Outline

1. Introduction

- Overview of PE files and their role in the Windows OS.
- Importance of understanding PE file structure for reverse engineering, debugging, and cybersecurity.

2. History of Evolution

- Evolution of executable file formats leading to PE:
    - COM Format: Early executable
    - MZ Format: basic headers.
    - NE Format: Supported dynamic linking and 16-bit Windows.
    - LE and LX Formats: Designed for 32-bit applications, used in OS/2.
    - COFF Format: Modular approach to manage sections and debug data.
    - PE Format: Combines COFF features with Windows-specific enhancements.

3. Why PE?

- Key advantages of the PE format:
    - Dynamic linking and modularity.
    - Efficient memory management and security features.
    - Support for embedded resources and cross-platform compatibility.

4. PE File Abstract

- DOS Header:
    - Magic number (MZ) and pointer to PE header (e_lfanew).
    - DOS stub: Basic compatibility message for DOS systems.
- NT Headers:
    - PE Signature: Identifies the file as a PE executable.

    - File Header: Contains machine type, number of sections, and timestamps.
    - Optional Header: Provides critical information like entry point, image base, and memory layout.
- Section Header:
    - Details metadata for each section (e.g., .text, .data, .rdata).
    - Key attributes like RVA, size, and characteristics.
- Data Directory:
    - Pointers to structures like Import Table, Export Table, Resource Table, and Relocation Table.

5. Tools for PE File Analysis

- Recommended tools for dissecting PE files:
    - CFF Explorer: GUI-based analysis.
    - PE-bear and PEview: Lightweight PE file explorers.
    - Hex editors (e.g., HxD) for manual inspection.

6. Detailed Analysis of Headers

- NT Header Structure:
    - Explanation of PE Signature, File Header, and Optional Header fields.
- Optional Header:
    - Magic number (PE32 vs. PE32+).
    - Address of entry point, image base, and sizes of code and data.
- Section Header:
    - Breakdown of section attributes and memory mapping.

7. Key Sections Explained

- .text Section: Contains executable code.
- .rdata Section: Stores constants and read-only data.

- .data Section: Holds writable variables and dynamic data.
- .pdata Section: Exception handling and function tables.
- .reloc Section: Handles relocation for flexible memory loading.

8. How a PE File Loads in Memory

- Step-by-step process:
    1. Loading the PE file.
    2. Reading headers and initializing memory allocation.
    3. Mapping sections into memory.
    4. Resolving imports and relocations.
    5. Setting up the entry point and running the program.
- Importance of memory layout for execution and debugging.

9. Strings in PE Files

- Definition and purpose of strings.
- Locations of strings in PE files:
    - .rdata, .data, Import Table, Resource Section.
- Methods for extracting strings using tools and scripts.
- Importance for reverse engineering and malware analysis.

10. Summary

Introduction

The PE (Portable Executable) file format is the standard for executable files, DLLs (Dynamic Link Libraries), and system files in the Windows operating system. Understanding the PE file structure is crucial for anyone involved in software development, reverse engineering, cybersecurity, or system administration. This file format allows Windows to load applications and execute them properly by providing critical information about how the file should be processed in memory.

When you analyze a PE file, you dive into a carefully structured blueprint of how a program is organized and executed. From the headers that guide the operating system to the specific sections where the code and data reside, the structure of the PE file is key to understanding its function.

By reading this, you'll gain insights into:

- How PE files are structured: Learn the different sections and headers that make up a PE file.
- Why this structure matters: Discover the role each part of the file plays in the execution process.
- Practical benefits: Understanding the structure helps in debugging, security analysis, and reverse engineering, allowing professionals to manipulate and optimize PE files effectively.

History of Evolution

1. COM Format (1970s – Early 1980s)

The COM format was one of the earliest and simplest file formats for executable programs. It came from the days of CP/M and later **MS-DOS**.

- What it was: Think of it as a raw file with just the program's code and nothing else. There were no headers or extra information—just the instructions that the computer would run.
- Why it worked: It was simple and efficient for small programs, limited to 64 KB in size. Perfect for the early days of computing.
- Why it fell short: As programs grew bigger and more complex, this format struggled. It didn't support features like splitting code and data or dynamic linking, and managing memory was a nightmare.

---

2. MZ Format (1980s)

Next came **the MZ** format, named after **Mark Zbikowski**, who worked at Microsoft. It was the first format used for MS-DOS executables.

- What changed: The **MZ** format added a small header at the start of the file. This header contained information like the program size, where to start running the code, and a magic number ("**MZ**") to identify the file type.
- Why it was better: Programs could now handle more than 64 KB by using segmented memory—dividing the program into sections for code, data, and stack.
- The problem: It was still limited to the DOS world and couldn't handle modern features like dynamic linking or relocation.

---

3. **NE Format (Mid-1980s)**

With the rise of Windows, the NE (New Executable) format appeared. It was made to support 16-bit Windows applications.

- The big improvements:
    - Dynamic linking: Programs could now share code using DLLs (Dynamic-Link Libraries).

    - Relocation: This meant the program could be loaded into different memory locations without breaking.
    - Resources: Developers could include icons, menus, and other UI elements directly in the file.
- Why it didn't last: It was tied to 16-bit systems, which became outdated as computing shifted to 32-bit and beyond.

---

4. LE and LX Formats (Early 1990s)

The LE and LX formats were designed for 32-bit applications on advanced operating systems like OS/2.

- What made them stand out:
    - They supported 32-bit architectures, which allowed for much larger and more powerful applications.
    - Demand paging: Only loaded parts of the program into memory as needed, saving resources.
    - Virtual memory: Made multitasking smoother and more efficient.
- Why they didn't dominate: They were mostly used in niche environments like OS/2, which lost the battle to Windows.

---

5. COFF Format (Late 1980s – Early 1990s)

The **COFF** (Common Object File Format) started on Unix systems and introduced a more modular way to manage program files.

- What it brought to the table:
    - Files were divided into sections, like code, data, and debug information.
    - It made linking programs easier and more reliable.
    - Included metadata, which made debugging and managing the files simpler.
- Where it struggled: While great for Unix, **COFF** didn't have built-in support for Windows-specific features, like embedding resources or dynamic linking.

---

6. PE Format (1993 – Present)

Finally, we get to the PE (Portable Executable) format, which became the backbone of Windows executables. It’s built on COFF but adds so much more to fit the needs of modern systems.

- Why it’s awesome:
  - Dynamic linking: Fully supports DLLs, making programs modular and efficient.
  - Section-based structure: Keeps code, data, and resources separate for better organization and memory management.
  - Memory permissions: Each section can have specific permissions (like read, write, or execute), adding a layer of security.
  - Relocation: Programs can be loaded anywhere in memory without breaking.
  - Resources: Developers can include images, icons, and additional files right in the executable.
  - Cross-platform flexibility: Works with multiple architectures (like x86, x64, and ARM).

Why PE ??

The **PE** (Portable Executable) format was developed to overcome the limitations of older file formats like MZ, which were insufficient for modern software needs. The PE format enhances organization, efficiency, and security in executable files. Here are the primary reasons it became the standard:

Separation of Code and Data: The PE format distinctly separates program code from data, improving memory management and making it easier to locate various elements within the code.

Improved Memory Management: PE files assign specific memory permissions to different sections, enhancing security and preventing malicious actions. The system uses smaller memory pages (typically 4KB) for efficient resource allocation.

Memory Efficiency: The format conserves space by leaving uninitialized variables unfilled, thus optimizing both memory and loading speed.

Support for Dynamic Linking: PE files can link to external libraries (DLLs), allowing for smaller executables, reduced memory usage, and simplified software updates.

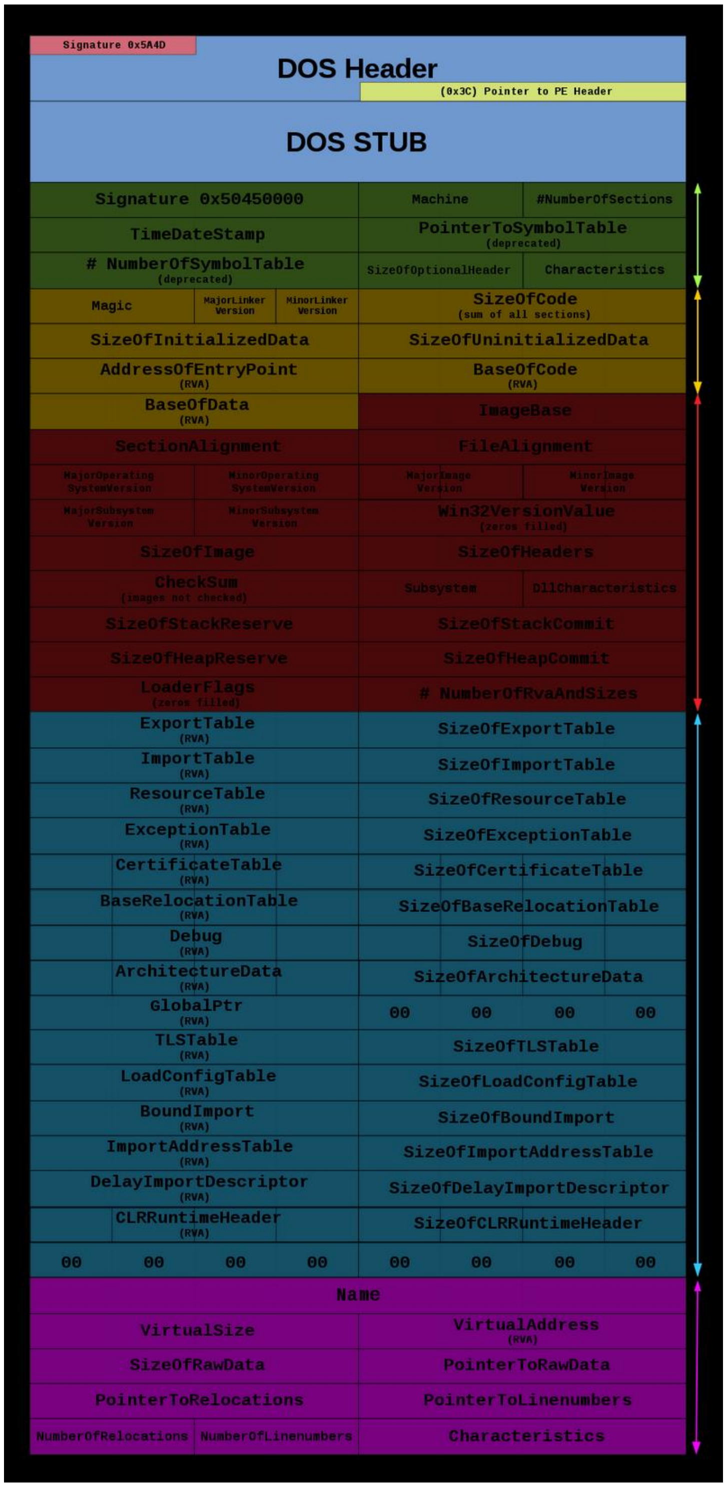
Flexibility in Memory Allocation: The relocatable nature of PE files allows the operating system to load applications at various memory addresses, improving overall memory usage.

Embedded Resources: PE format can incorporate various resources, such as icons and configuration files, directly into the executable, streamlining deployment.

Cross-Platform Compatibility: PE format is portable across multiple processors and platforms, including Windows environments.

Not only .exe files are PE files, dynamic link libraries (.dll), Kernel modules (.srv), Control panel applications (.cpl) and many others are also PE files.

PE File Abstract :



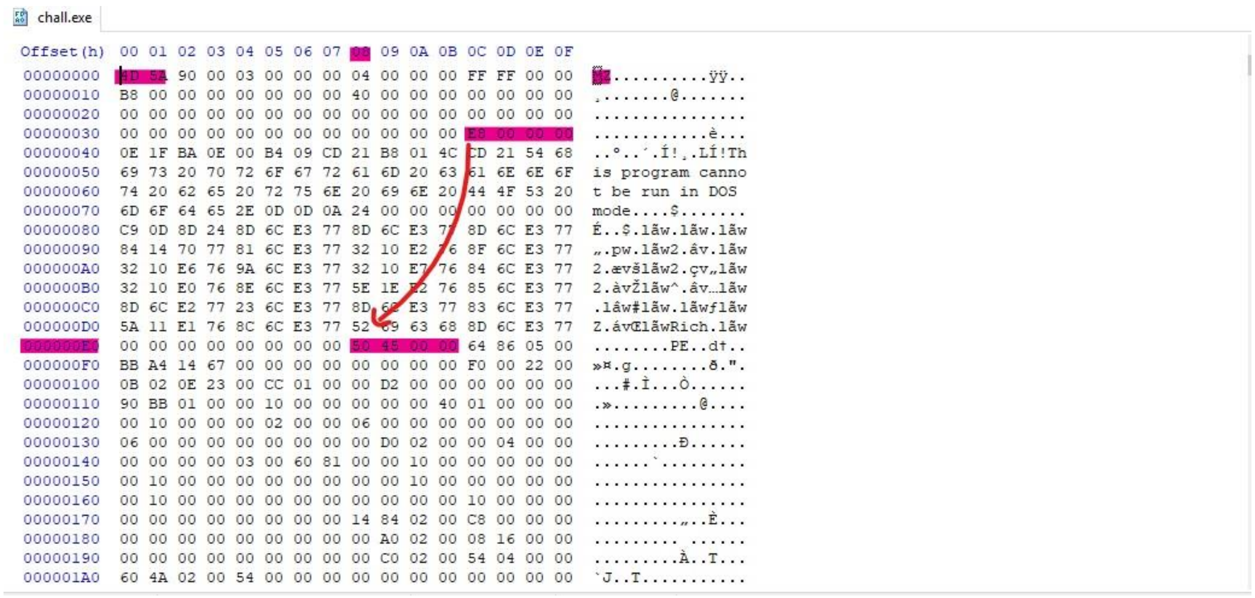
A PE file is a data structure that holds information necessary for the OS loader to be able to load that executable into memory and execute it.

I will use any hex-editor to explain in abstract pe structure then we will dive deep into pe file

In the first DOS Header:

The DOS header is located at the start of the PE file with a 64-bytes-long structure and always begins with the ASCII characters “MZ” (0x4D, 0x5A). This is known as the Magic Number, which identifies the file as an MS-DOS executable

The DOS header structure includes the following:



chall.exe

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00  ............è...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  C9 0D 8D 24 8D 6C E3 77 8D 6C E3 77 8D 6C E3 77  É..$.lãw.lãw.lãw
00000090  84 14 70 77 81 6C E3 77 32 10 E2 76 8F 6C E3 77  „.pw.lãw2.âv.lãw
000000A0  32 10 E6 76 9A 6C E3 77 32 10 E7 76 84 6C E3 77  2.ævšlãw2.çv„lãw
000000B0  32 10 E0 76 8E 6C E3 77 5E 1E E2 76 85 6C E3 77  2.àvŽlãw^.âv…lãw
000000C0  8D 6C E2 77 23 6C E3 77 8D 6C E3 77 83 6C E3 77  .lâw#lãw.lãwƒlãw
000000D0  5A 11 E1 76 8C 6C E3 77 52 69 63 68 8D 6C E3 77  Z.ávŒlãwRich.lãw
000000E0  00 00 00 00 00 00 00 00 50 45 00 00 64 86 05 00  ........PE..d†..
000000F0  BB A4 14 67 00 00 00 00 00 00 00 00 F0 00 22 00  »¤.g........ð.".
00000100  0B 02 0E 23 00 CC 01 00 00 D2 00 00 00 00 00 00  ...#.Ì...Ò......
00000110  90 BB 01 00 00 10 00 00 00 00 00 40 01 00 00 00  .».........@....
00000120  00 10 00 00 00 02 00 00 06 00 00 00 00 00 00 00  ................
00000130  06 00 00 00 00 00 00 00 00 D0 02 00 00 04 00 00  .........Ð......
00000140  00 00 00 00 03 00 60 81 00 00 10 00 00 00 00 00  ......`.........
00000150  00 10 00 00 00 00 00 00 00 00 10 00 00 00 00 00  ................
00000160  00 10 00 00 00 00 00 00 00 00 00 00 10 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 14 84 02 00 C8 00 00 00  ........„..È...
00000180  00 00 00 00 00 00 00 00 00 A0 02 00 08 16 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 C0 02 00 54 04 00 00  .........À..T...
000001A0  60 4A 02 00 54 00 00 00 00 00 00 00 00 00 00 00  `J..T...........
```

1. Magic Number (0x4D, 0x5A)

O . The first two bytes represent "MZ," identifying the file as an MS-DOS executable.

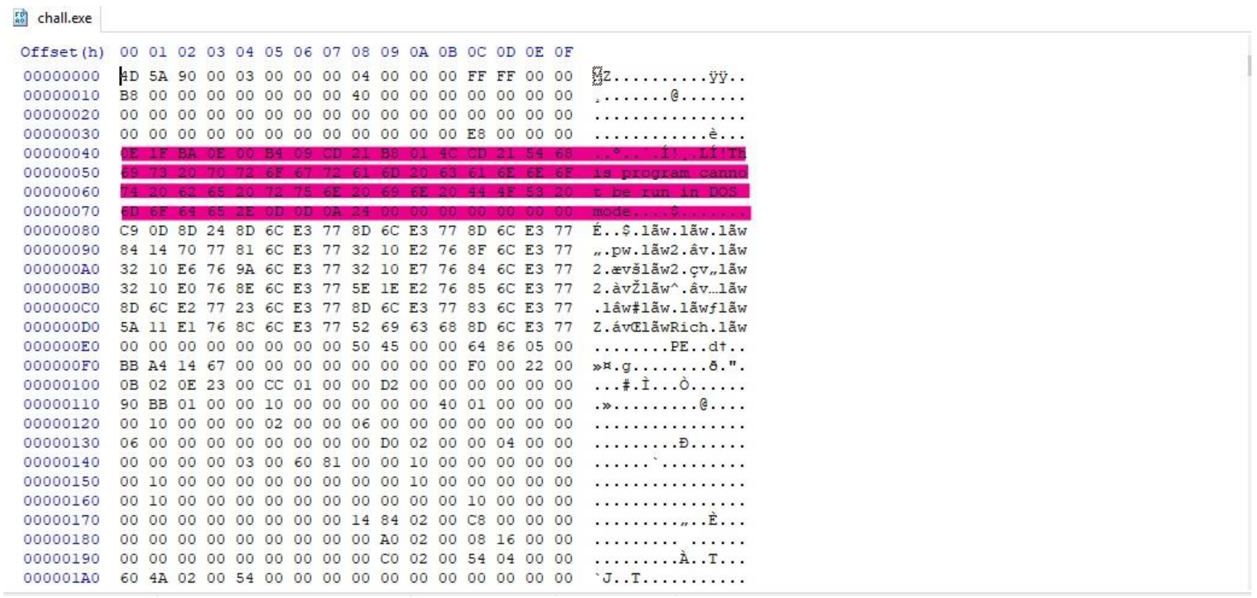
2. File Address of the PE Header (Offset 0x3C)

O . This field points to the location of the PE header, allowing the Windows operating system to locate it in the file which in our case is 000000E8.

The DOS header is followed by the **DOS_STUB**, a small program that typically displays an error message or instructions when the file is run on an unsupported system

DOS Stub:

After the DOS header comes the DOS stub which is a small MS-DOS 2.0 compatible executable that just prints an error message saying “This program cannot be run in DOS mode” when the program is run in DOS mode.



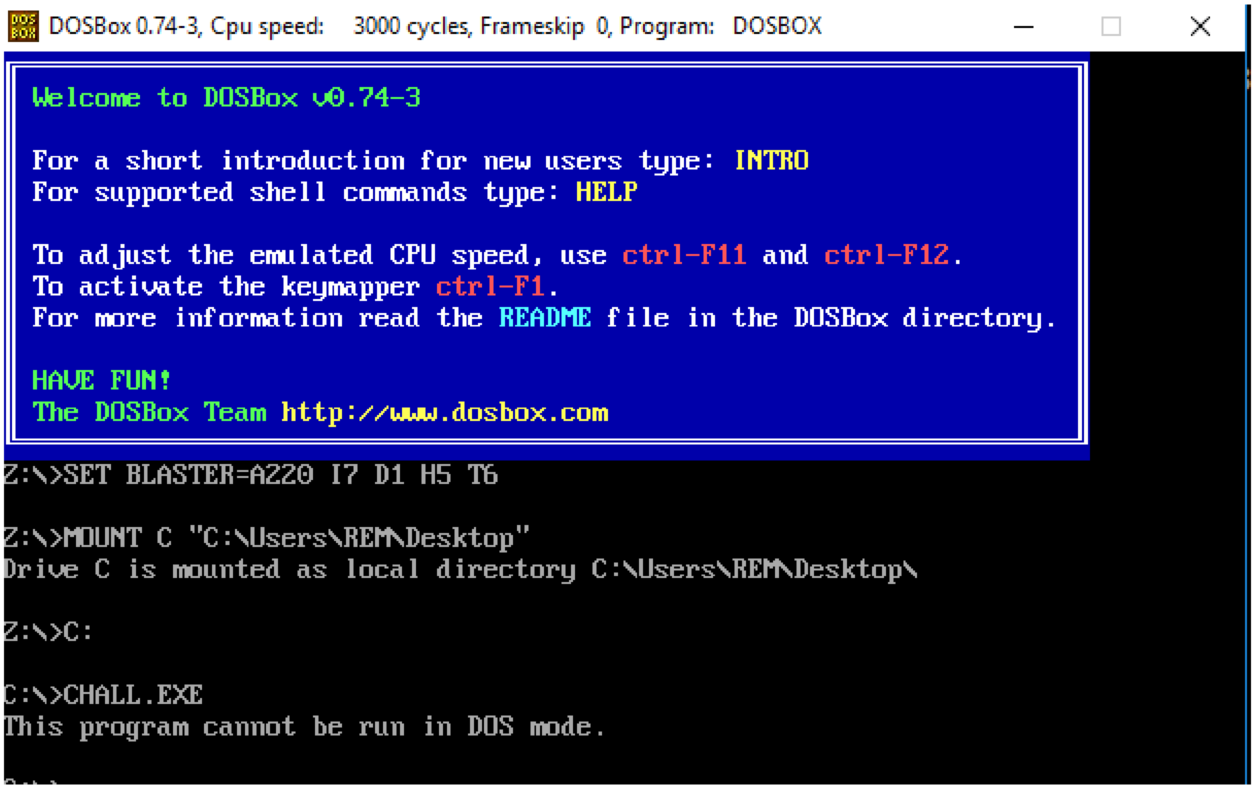
```
chall.exe

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00  ............è...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  C9 0D 8D 24 8D 6C E3 77 8D 6C E3 77 8D 6C E3 77  É..$.lãw.lãw.lãw
00000090  84 14 70 77 81 6C E3 77 32 10 E2 76 8F 6C E3 77  „.pw.lãw2.âv.lãw
000000A0  32 10 E6 76 9A 6C E3 77 32 10 E7 76 84 6C E3 77  2.ævšlãw2.çv„lãw
000000B0  32 10 E0 76 8E 6C E3 77 5E 1E E2 76 85 6C E3 77  2.àvŽlãw^.âv…lãw
000000C0  8D 6C E2 77 23 6C E3 77 8D 6C E3 77 83 6C E3 77  .lâw#lãw.lãwƒlãw
000000D0  5A 11 E1 76 8C 6C E3 77 52 69 63 68 8D 6C E3 77  Z.ávŒlãwRich.lãw
000000E0  00 00 00 00 00 00 00 00 50 45 00 00 64 86 05 00  ........PE..d†..
000000F0  BB A4 14 67 00 00 00 00 00 00 00 00 F0 00 22 00  »¤.g........ð.".
00000100  0B 02 0E 23 00 CC 01 00 00 D2 00 00 00 00 00 00  ...#.Ì...Ò......
00000110  90 BB 01 00 00 10 00 00 00 00 00 40 01 00 00 00  .».........@....
00000120  00 10 00 00 00 02 00 00 06 00 00 00 00 00 00 00  ................
00000130  06 00 00 00 00 00 00 00 00 D0 02 00 00 04 00 00  .........Ð......
00000140  00 00 00 00 03 00 60 81 00 00 10 00 00 00 00 00  ......`.........
00000150  00 10 00 00 00 00 00 00 00 00 10 00 00 00 00 00  ................
00000160  00 10 00 00 00 00 00 00 00 00 00 00 10 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 14 84 02 00 C8 00 00 00  ........„..È...
00000180  00 00 00 00 00 00 00 00 00 A0 02 00 08 16 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 C0 02 00 54 04 00 00  .........À..T...
000001A0  60 4A 02 00 54 00 00 00 00 00 00 00 00 00 00 00  `J..T...........
```

Size: Usually 60-70 bytes, but can vary depending on the implementation.

Code: The executable code that is run when a PE file is executed in a DOS environment. This code typically displays a message.

Message: Commonly includes text such as: This program cannot be run in DOS mode

to see the benefit of the previous section we will run it at dos mode to see the massege


DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
Welcome to DOSBox v0.74-3

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com
Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>MOUNT C "C:\Users\REM\Desktop"
Drive C is mounted as local directory C:\Users\REM\Desktop\

Z:\>C:

C:\>CHALL.EXE
This program cannot be run in DOS mode.
```

NT Headers:

It is difficult to see these headers in a hex editor, so we will cover them in detail later using tools to make it easier.

The NT Headers part contains three main parts:

PE signature: A 4-byte signature that identifies the file as a PE file.

File Header: A standard COFF File Header. It holds some information about the PE file.

Optional Header: The most important header of the NT Headers, its name is the Optional Header because some files like object files don't have it, however it's required for image files (files like .exe files). This header provides important information to the OS loader.

Section Header:

in a PE file contains important metadata about each section of the executable

Key Fields:

1. **Name:** The name of the section (e.g., .text, .data).
2. **Virtual Size:** The actual size of the section in memory.
3. **Virtual Address:** The memory address where the section will be loaded.

4. **Size of Raw Data:** The size of the section data in the file.
5. **Pointer to Raw Data:** The offset in the file where the section starts.
6. **Characteristics:** Flags defining section properties (e.g., executable, readable, writable).

Data Directorie:

To point to important data structures, such as imports, exports, or debugging information, used during program execution.

Common Directories:

1. **Export Table:** Details exported functions and data.
2. **Import Table:** Contains information about imported libraries and functions.
3. **Resource Table:** Stores resources like icons, strings, and bitmaps.
4. **Exception Table:** Holds information about exception handling.
5. **Certificate Table:** Contains security certificates for digital signing.
6. **Relocation Table:** Used for address adjustments during loading.
7. **Debug Directory:** Contains debugging information.
8. **TLS (Thread Local Storage) Table:** Holds data used for thread-specific storage.

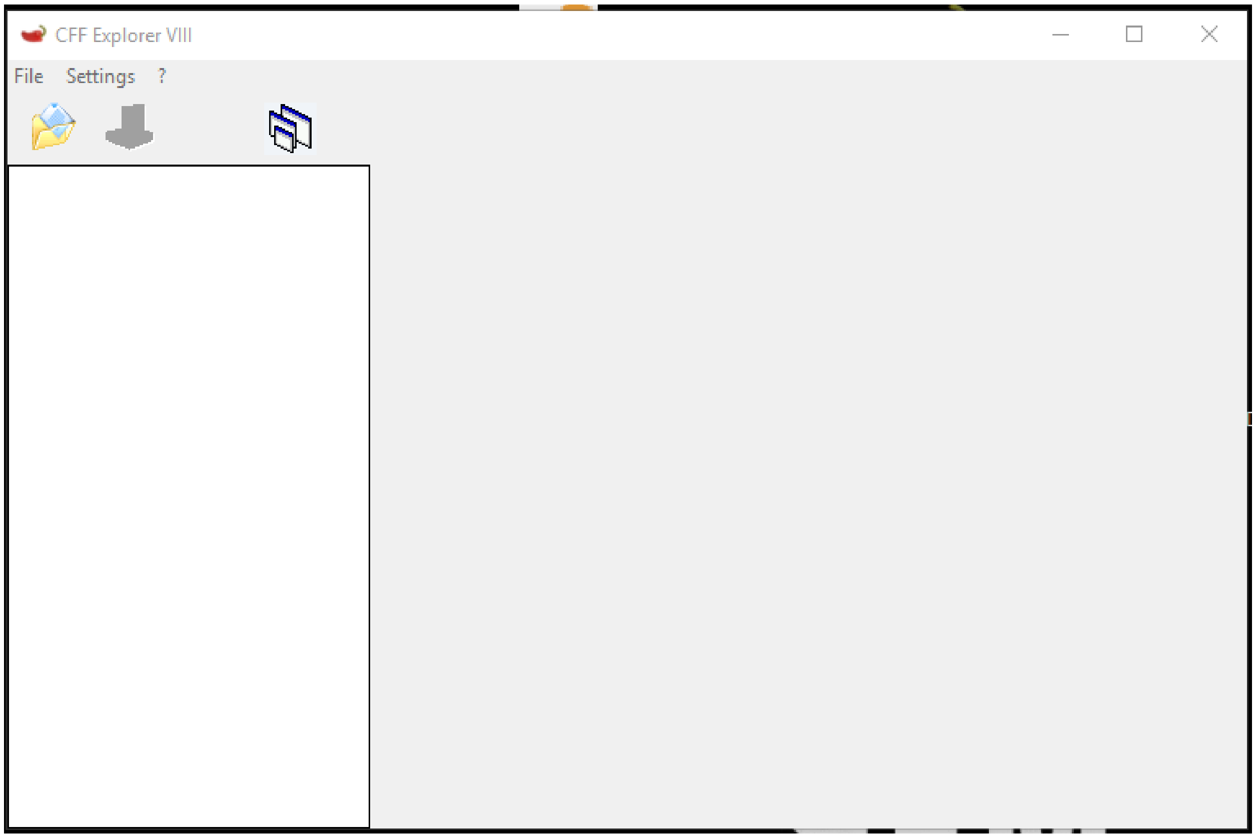
We will cover every part in details later

Now we will talk about tools that will help us to deep dive into PE file and understand the structure of PE file

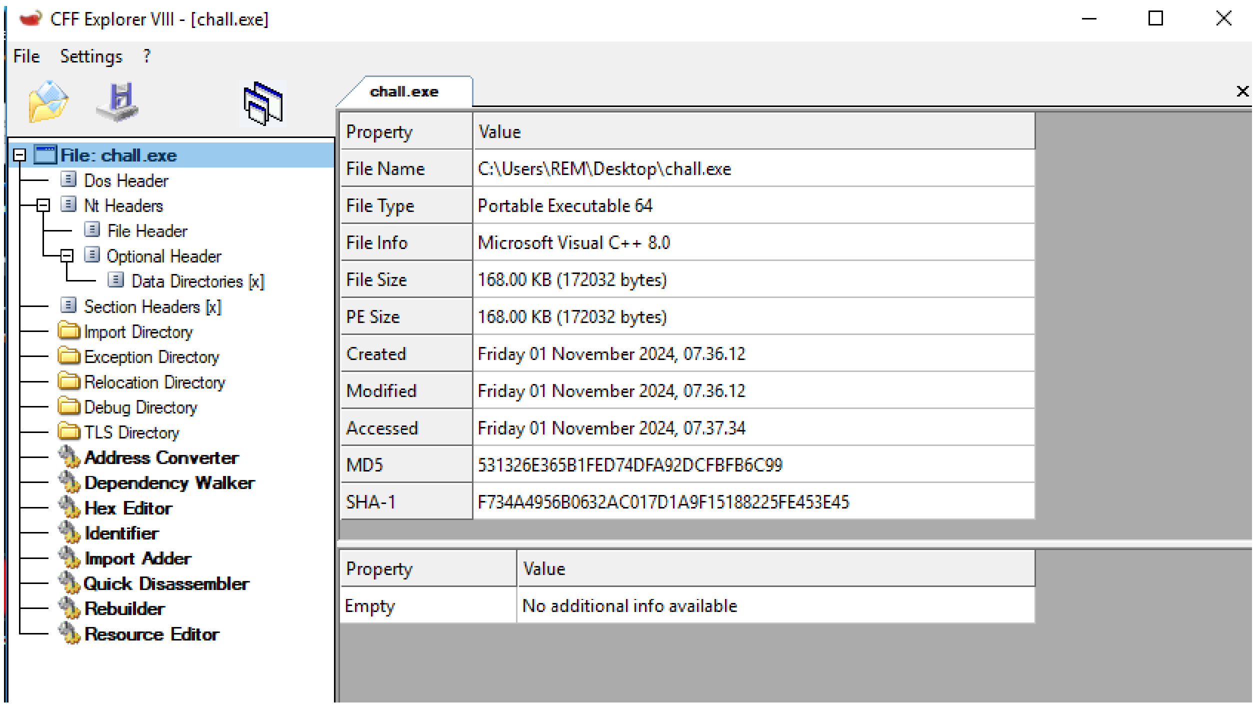
Pe file analysis tool :

CFF Explorer: We will use CFF Explorer. It is a good tool to analyze PE file headers, and it provides a good GUI, which makes the process very easy. You can download it from this "https://ntcore.com/files/CFF_Explorer.zip" and install it like any other program.

After installing it and running it, you will find the empty GUI looks like this :



When you drag the file you want to analyze and drop it, you will find this:



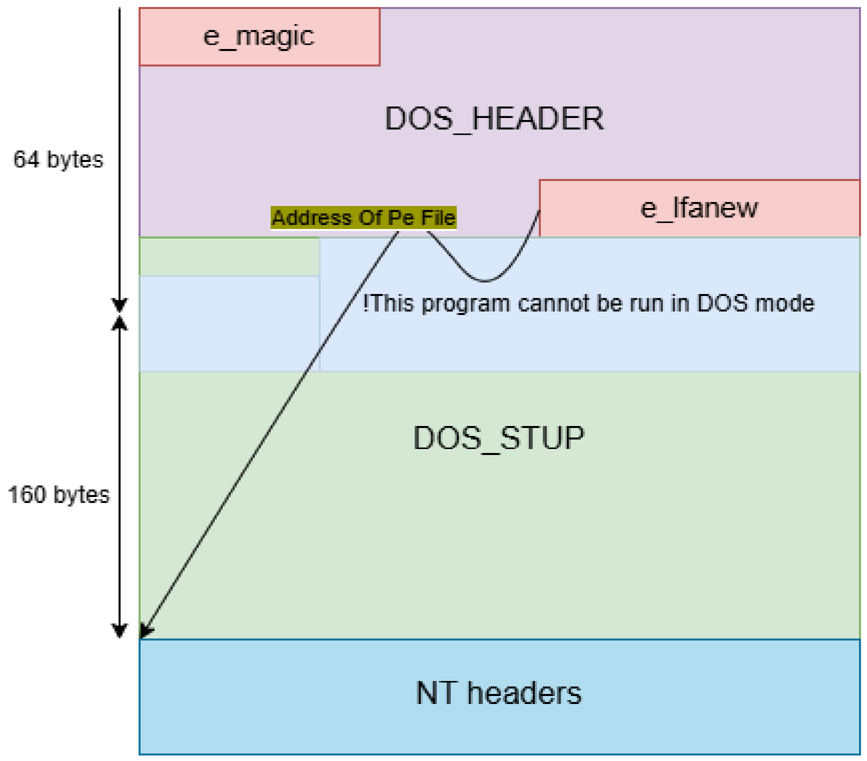
Property	Value
File Name	C:\Users\REM\Desktop\chall.exe
File Type	Portable Executable 64
File Info	Microsoft Visual C++ 8.0
File Size	168.00 KB (172032 bytes)
PE Size	168.00 KB (172032 bytes)
Created	Friday 01 November 2024, 07.36.12
Modified	Friday 01 November 2024, 07.36.12
Accessed	Friday 01 November 2024, 07.37.34
MD5	531326E365B1FED74DFA92DCFBFB6C99
SHA-1	F734A4956B0632AC017D1A9F15188225FE453E45

Property	Value
Empty	No additional info available

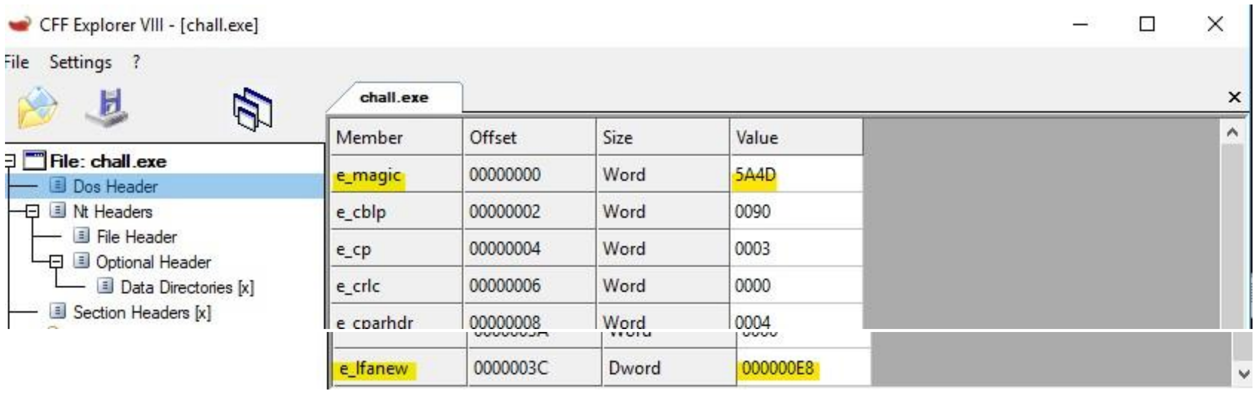
As you can see, there are a lot of things you already know, as we explained in previous sections. Other parts will be explained later

So, we will explain the important values in each header. Now, we will explain the **DOS header:** As mentioned before, it’s a 64-byte-long structure, we can take a look at the contents of that structure by looking at the IMAGE_DOS_HEADER structure definition from winnt.h:

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



But we can found the important values as showen in next image:



Member	Offset	Size	Value
e_magic	00000000	Word	5A4D
e_cblp	00000002	Word	0090
e_cp	00000004	Word	0003
e_crlc	00000006	Word	0000
e_cparhdr	00000008	Word	0004
e_lfanew	0000003C	Dword	000000E8

- **e_magic:** This is the first member of the DOS Header, it's a WORD so it occupies 2 bytes, it's usually called the magic number. It has a fixed value of 0x5A4D or MZ in ASCII, and it serves as a signature that marks the file as an MS-DOS executable.

  MZ: which stand for "Mark Zbikowski.

- **e_lfanew:** This is the last member of the DOS header structure, it's located at offset 0x3C into the DOS header and it holds an offset to the start of the NT headers. This member is important to the PE loader on Windows systems because it tells the loader where to look for the file header.

Essential Base:

Before diving into the details, it's critical to understand a key concept that will be frequently referenced: the **Relative Virtual Address (RVA)**.

What is an RVA?

An **RVA** represents an **offset** from the memory location where the image (PE file) is loaded, known as the **Image Base**. It is not an absolute memory address but rather a relative position within the loaded image.

To convert an **RVA** into an **absolute Virtual Address (VA)**, you simply add the RVA to the Image Base. This concept is crucial for understanding the Portable Executable (PE) file structure.

Breaking It Down

1. **Image Base:**
   - This is the starting address where the program or DLL is loaded into memory.
   - Example: If the Image Base is 0x400000, it means the PE file is loaded starting at this address in memory.
2. **Relative Virtual Address (RVA):**
   - The RVA is an offset from the Image Base. It tells you how far a specific piece of data or code is located from the start of the Image Base.
   - Example: If the RVA is 0x1000, the actual memory address is: VA=Image Base+RVA=0x400000+0x1000=0x401000
3. **Raw Offset (File Offset):**
   - This is the location of the data within the raw file on disk.
   - To find the Raw Offset from the RVA, you use the section headers. The formula is:

Raw Offset=(RVA−Virtual Address of Section)+Raw Address of Sect
ion
Section}Raw Offset=(RVA−Virtual Address of Section)+Raw Addres
s of Section

- The Raw Address of Section and Virtual Address of Section are provided in the section headers.

Section Attribute	
RVA	Offset from the Image Base. Shows where the data or code resides relative to the start of the image.
VA (Virtual Address)	Actual memory address where the section or data is loaded. VA=Image Base+RVA
Raw Offset (File Offset)	Actual location of the data in the file. Calculated using the formula: RVA−Section VA+Section Raw Address
Real Size	The size of the section or data **on disk** (raw data size, from the section header).
Virtual Size	The size of the section or data **in memory** (from the section header). This can be larger due to alignment.
Start of Section (VA)	The starting Virtual Address of the section (provided in the section header).
End of Section (VA)	The Virtual Address at the end of the section: End VA=Start VA+Virtual Size

Example Calculation

Suppose we have the following data:

- Image Base: 0x400000
- Section Info:
  - Section Virtual Address: 0x1E000
  - Section Raw Address: 0x1D000
  - RVA: 0x28414
  - Virtual Size: 0x2000

Step 1: Calculate the VA

The Virtual Address is calculated as:

VA=Image Base+RVA
VA=0x400000+0x28414=0x428414

Step 2: Calculate the Raw Offset

To calculate the Raw Offset from the RVA:

Raw Offset=(RVA−Virtual Address of Section)+Raw Address of Section

Plugging in the values:

Raw Offset=(0x28414−0x1E000)+0x1D000
Raw Offset=0xA414+0x1D000=0x27414

Step 3: Calculate the End VA

The End Virtual Address is calculated as:

End VA=StartVA+Virtual SizeEnd

VA=0x428414+0x2000=0x42A414

Summarize:

Relative Virtual Address (RVA): This is the offset from the base of the image in memory. It indicates where a section or data starts within the loaded image.

Virtual Address (VA): This is the actual address where the section will be loaded into memory. It can be calculated by adding the image base to the RVA.

Real Size: This refers to the actual size of the section on disk (the file size) versus the size of the section in memory (the memory size, which is typically aligned).

Nt Header:

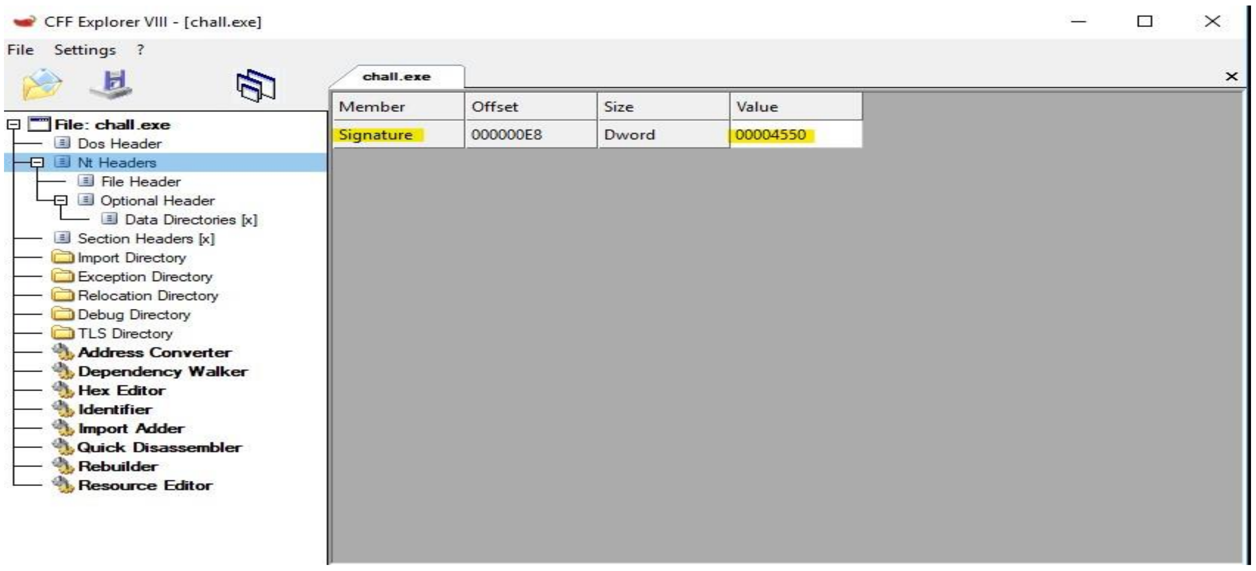
The NT headers are defined in the `winnt.h` file as the `IMAGE_NT_HEADERS` structure. By examining its definition, we can identify three primary members: a DWORD signature, an

`IMAGE_FILE_HEADER` structure called `FileHeader`, and an `IMAGE_OPTIONAL_HEADER` structure referred to as `OptionalHeader`.

It's important to note that this structure is defined in two versions: one for 32-bit executables (commonly referred to as PE32 executables) known as `IMAGE_NT_HEADERS`, and another for 64-bit executables (referred to as PE32+ executables) called `IMAGE_NT_HEADERS64`. The key distinction between these two versions lies in the corresponding `IMAGE_OPTIONAL_HEADER` structure, which also has two versions: `IMAGE_OPTIONAL_HEADER32` for 32-bit executables and `IMAGE_OPTIONAL_HEADER64` for 64-bit executables.

```
typedef struct _IMAGE_NT_HEADERS64 {
   DWORD Signature;
   IMAGE_FILE_HEADER FileHeader;
   IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;

typedef struct _IMAGE_NT_HEADERS {
   DWORD Signature;
   IMAGE_FILE_HEADER FileHeader;
   IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

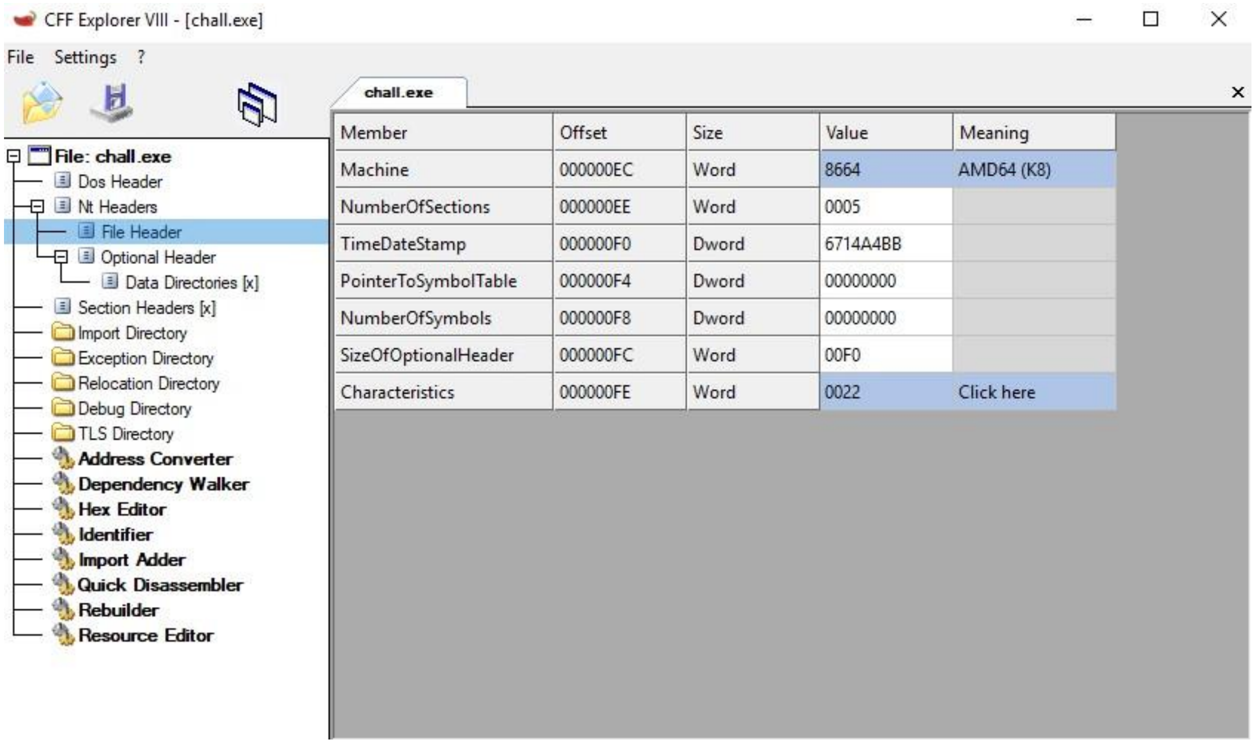
PE Signature: The PE Signature is a critical component of the NT Header in a PE (Portable Executable) file format. It marks the start of the NT Header and is used to identify the file as a valid PE file:



Member	Offset	Size	Value
Signature	000000E8	Dword	00004550

File Header : is part of the NT Header in a PE file and provides essential metadata about the executable It's defined as IMAGE_FILE_HEADER in winnt.h, here's the definition:

```
typedef struct _IMAGE_FILE_HEADER {
   WORD    Machine;
   WORD    NumberOfSections;
   DWORD   TimeDateStamp;
   DWORD   PointerToSymbolTable;
   DWORD   NumberOfSymbols;
   WORD    SizeOfOptionalHeader;
   WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```



Member	Offset	Size	Value	Meaning
Machine	000000EC	Word	8664	AMD64 (K8)
NumberOfSections	000000EE	Word	0005	
TimeDateStamp	000000F0	Dword	6714A4BB	
PointerToSymbolTable	000000F4	Dword	00000000	
NumberOfSymbols	000000F8	Dword	00000000	
SizeOfOptionalHeader	000000FC	Word	00F0	
Characteristics	000000FE	Word	0022	Click here

As you can see, it contains some information. We will explain each one of them.

1. **Machine(2-byte):**
   - Identifies the type of CPU the file is intended for.

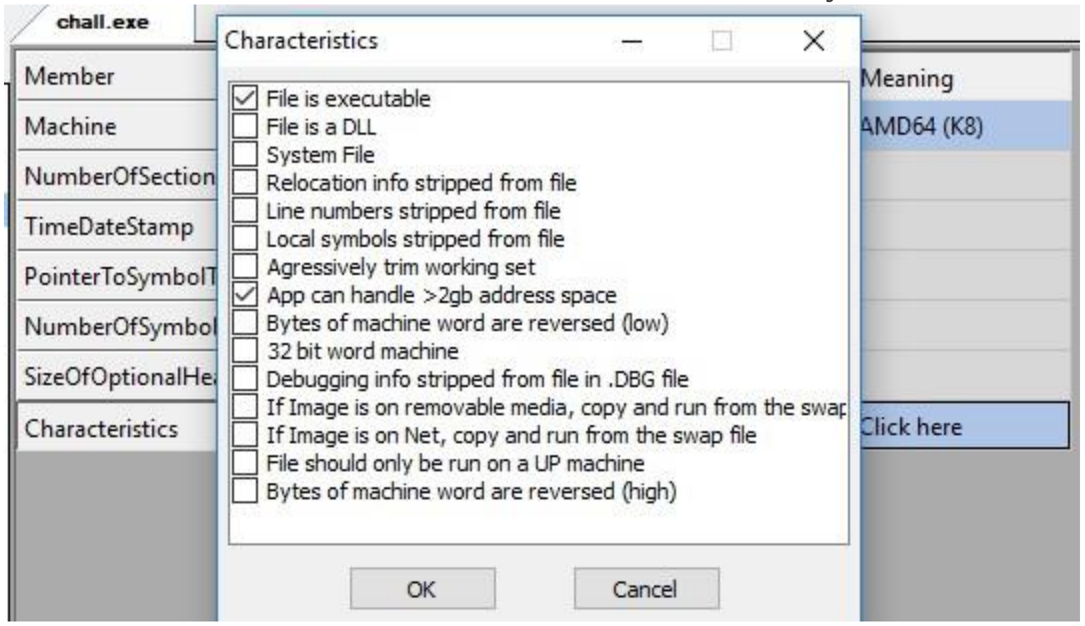
- Examples:
    - 0x014C: 32-bit (x86).
    - 0x8664: 64-bit (x64).
    - 0x0200: Intel Itanium.

  You can find more values is this page https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

2. **NumberOfSections(2-byte):**
    - Indicates the number of sections defined in the Section Table. Each section represents a part of the file, such as .text (code) or .data (data).
    - **Maximum number of sections**: **65535** (limited by the **2-byte field** in the File Header).
    - **Practical number of sections**: Typically between **1 to 20** sections in most PE files.
3. **TimeDateStamp:**
    - Helps determine when the file was compiled or last modified. Tools often use this to track versions or debugging purposes.
4. **PointerToSymbolTable:**

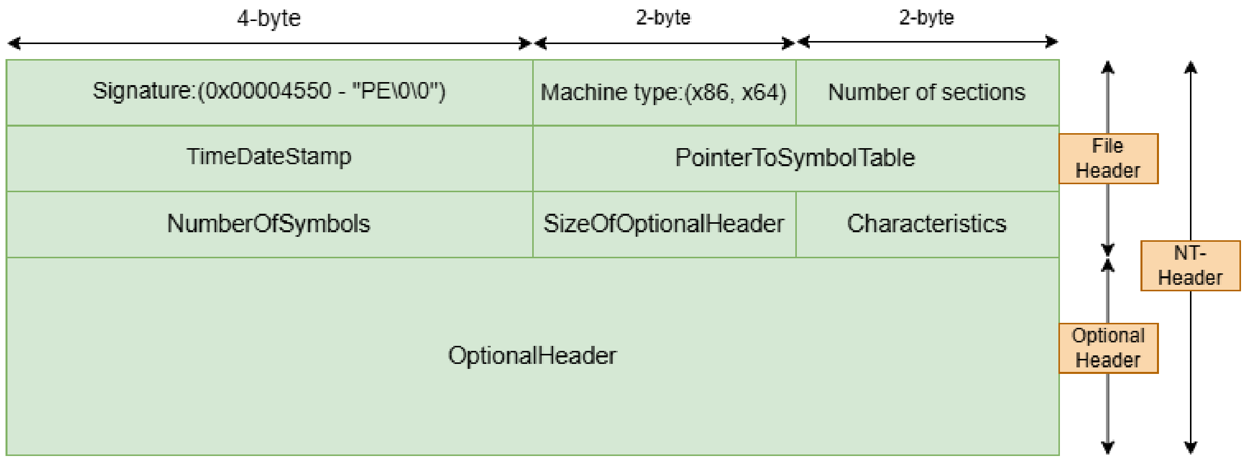
   Symbol Table : In older formats like COFF (Common Object File Format), the symbol table was used to store information about the symbols (such as functions, variables, and other program identifiers) within the executable. The symbol table allowed linkers and debuggers to map names to memory addresses and perform linking and debugging tasks. For modern Windows executables, symbol resolution is typically handled using PDB files, and the PointerToSymbolTable in the File Header is often set to 0.
5. **SizeOfOptionalHeader:**
    - Indicates the size of the Optional Header.
    - The Optional Header contains information like entry points, base addresses, and image sizes.
6. **Characteristics:**

- A bitfield defining the attributes of the file.
- You can see a button labeled "Click Here." When you press this button, it will convert the hex values to reveal what they refer to



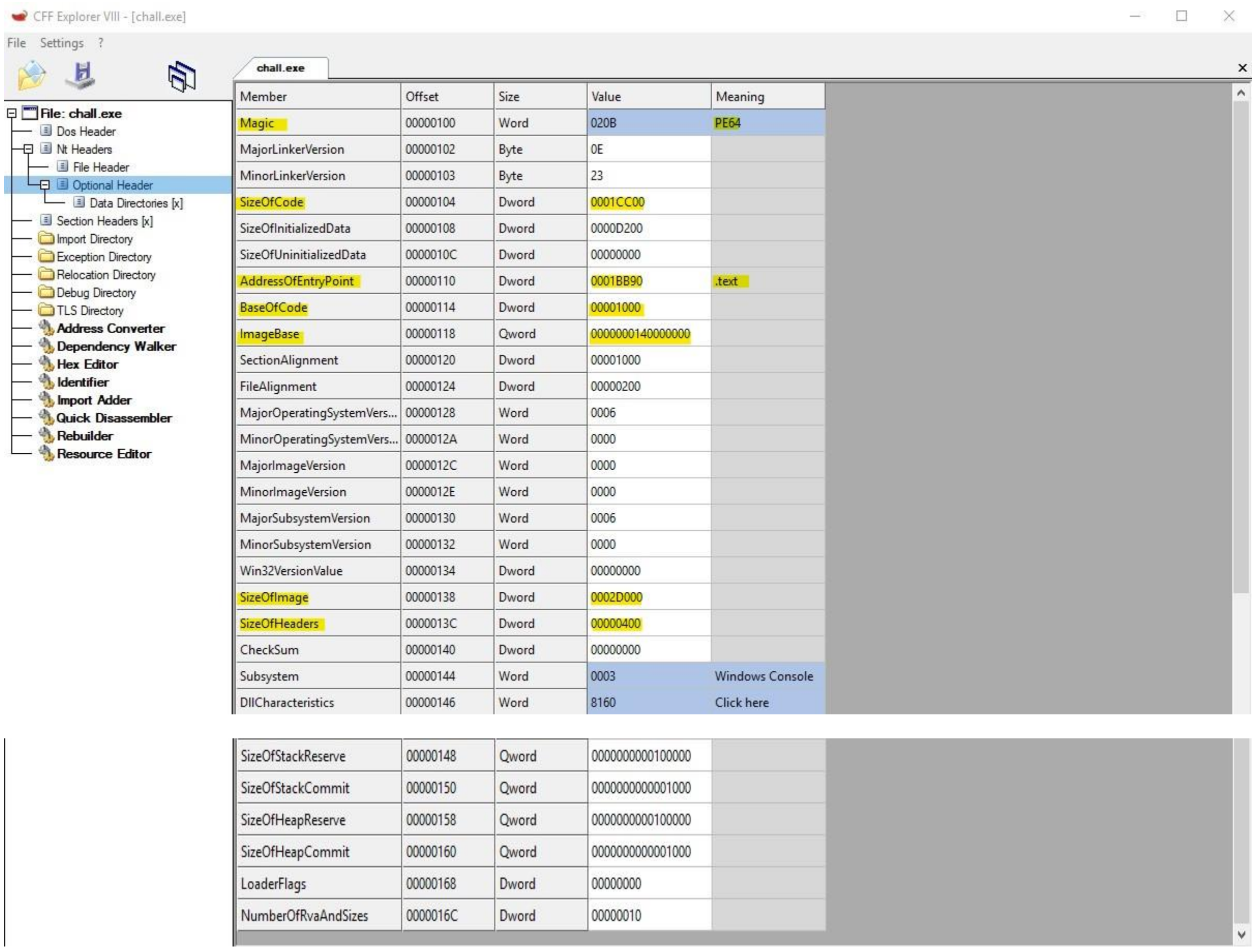
- [✓] File is executable
- [ ] File is a DLL
- [ ] System File
- [ ] Relocation info stripped from file
- [ ] Line numbers stripped from file
- [ ] Local symbols stripped from file
- [ ] Agressively trim working set
- [✓] App can handle >2gb address space
- [ ] Bytes of machine word are reversed (low)
- [ ] 32 bit word machine
- [ ] Debugging info stripped from file in .DBG file
- [ ] If Image is on removable media, copy and run from the swap
- [ ] If Image is on Net, copy and run from the swap file
- [ ] File should only be run on a UP machine
- [ ] Bytes of machine word are reversed (high)

- Common values:
  - 0x0002: Executable file.
  - 0x0100: 32-bit application.
  - 0x2000: DLL file



Optional Header: It's essential for the operating system to know how to handle the program. Think of it as a guidebook for the OS. We will explain the important values in it

Let’s take a look at the definition of both structures.



Member	Offset	Size	Value	Meaning
Magic	00000100	Word	020B	PE64
MajorLinkerVersion	00000102	Byte	0E	
MinorLinkerVersion	00000103	Byte	23	
SizeOfCode	00000104	Dword	0001CC00	
SizeOfInitializedData	00000108	Dword	0000D200	
SizeOfUninitializedData	0000010C	Dword	00000000	
AddressOfEntryPoint	00000110	Dword	0001BB90	.text
BaseOfCode	00000114	Dword	00001000	
ImageBase	00000118	Qword	0000000140000000	
SectionAlignment	00000120	Dword	00001000	
FileAlignment	00000124	Dword	00000200	
MajorOperatingSystemVers...	00000128	Word	0006	
MinorOperatingSystemVers...	0000012A	Word	0000	
MajorImageVersion	0000012C	Word	0000	
MinorImageVersion	0000012E	Word	0000	
MajorSubsystemVersion	00000130	Word	0006	
MinorSubsystemVersion	00000132	Word	0000	
Win32VersionValue	00000134	Dword	00000000	
SizeOfImage	00000138	Dword	0002D000	
SizeOfHeaders	0000013C	Dword	00000400	
CheckSum	00000140	Dword	00000000	
Subsystem	00000144	Word	0003	Windows Console
DllCharacteristics	00000146	Word	8160	Click here
SizeOfStackReserve	00000148	Qword	0000000000100000	
SizeOfStackCommit	00000150	Qword	0000000000001000	
SizeOfHeapReserve	00000158	Qword	0000000000100000	
SizeOfHeapCommit	00000160	Qword	0000000000001000	
LoaderFlags	00000168	Dword	00000000	
NumberOfRvaAndSizes	0000016C	Dword	00000010	

1. Magic

The **Magic** value is like a label that tells the system if the program is for a 32-bit or a 64-bit computer.

- If it says **PE32**, it’s a 32-bit program. "0x10B"
- If it says **PE32+**, it’s a 64-bit program. "0x20B"
- Identifies the image as a ROM image. "0x107"

It’s the first step for the operating system to know how to handle the program.

2. Size of Code

This value is the total size of all the program's instructions (the code that actually runs).

Think of it as the size of the "action plan" for the program. The OS needs this number to load all the instructions into memory correctly.

3. Address of Entry Point

This is the starting point of the program—the first instruction the system will execute when the program runs.

- For an executable, it’s where the program begins.
- For a DLL, it’s where the initialization happens.

If this address is wrong, the program won’t work because it won’t know where to start.

4. Image Base

The **Image Base** is like the program’s “home address” in memory.

- The program tells the OS, “I’d prefer to live at this address.”
- If no other program is using that address, the OS will load it there. If the address is taken, the OS will move it somewhere else (and fix any references internally).

5. Base of Code

This tells the system where the actual program instructions (code) begin within the program's memory layout.
Think of it as a bookmark that says, “Start reading the instructions from here.”

6. Size of Headers

The headers contain all the information about the program (like a table of contents). This value tells the OS how big that table of contents is so it knows where the actual program data starts.

7. Size of Image

This is the total space the program will take up in memory, including the headers and all sections (code, data, etc.).
It’s like saying, “I need this much space in memory to work properly.”

8. SizeOfStackReserve

- **Meaning**: This is the **total amount of memory** that the system should reserve for the stack, regardless of how much of it is used. It represents the size of the stack area in memory that is **reserved** for the application, but **not necessarily committed**.
- **Example Value**: 0x00100000 (1 MB)
- **Purpose**: When an application is loaded, the system will reserve 1 MB of virtual memory for the stack, but it won't allocate physical memory for this region until the stack space is actually used.

Stack Reserve: The amount of address space reserved for the stack.

9. SizeOfStackCommit

- **Meaning**: This specifies the **amount of memory** that the system should initially **commit** from the reserved stack space. When memory is committed, it means that the system actually **allocates physical memory** to the application (not just virtual address space).
- **Example Value**: 0x00010000 (64 KB)
- **Purpose**: Initially, only 64 KB of the reserved 1 MB stack space will be committed and backed by physical memory. As the application uses more stack space (for example, by calling more functions), additional physical memory will be committed in increments.

Stack Commit: The amount of memory that is actually made available in physical RAM for the stack.

10. SizeOfHeapReserve

- **Meaning**: Similar to **SizeOfStackReserve**, this is the **total amount of memory** that the system should reserve for the heap.
- **Example Value**: 0x00100000 (1 MB)
- **Purpose**: It tells the system to reserve 1 MB of virtual address space for the heap area. The system does not allocate actual physical memory until needed.

Heap Reserve: The amount of address space reserved for dynamic memory allocation (heap).

11. SizeOfHeapCommit

- **Meaning**: This defines the **initial amount of memory** the system should commit from the reserved heap memory when the program starts.
- **Example Value**: 0x00010000 (64 KB)
- **Purpose**: Just like **SizeOfStackCommit**, it defines how much of the heap memory is initially committed. Initially, the system allocates 64 KB of physical memory to the heap, and as more heap memory is used (through memory allocation calls), more physical memory can be committed.

Data Directory: in the NT Header is like an index pointing to special features or resources that the program might use. Think of it as a list of important addresses or locations where the operating system can find extra information or functionality the program needs.

Here's how it works:

The Data Directory doesn't store the actual data itself. Instead, it tells the operating system, "Hey, if you're looking for this specific thing, here's where you can find it in the program's memory layout."

As you can see in the next image that every one of directorys has RVA and Size Size is the real size at the desk we found the structure of it like this:

```
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
```

IMAGE_NUMBEROF_DIRECTORY_ENTRIES is a constant defined with the value 16, meaning that this array can have up to 16 :

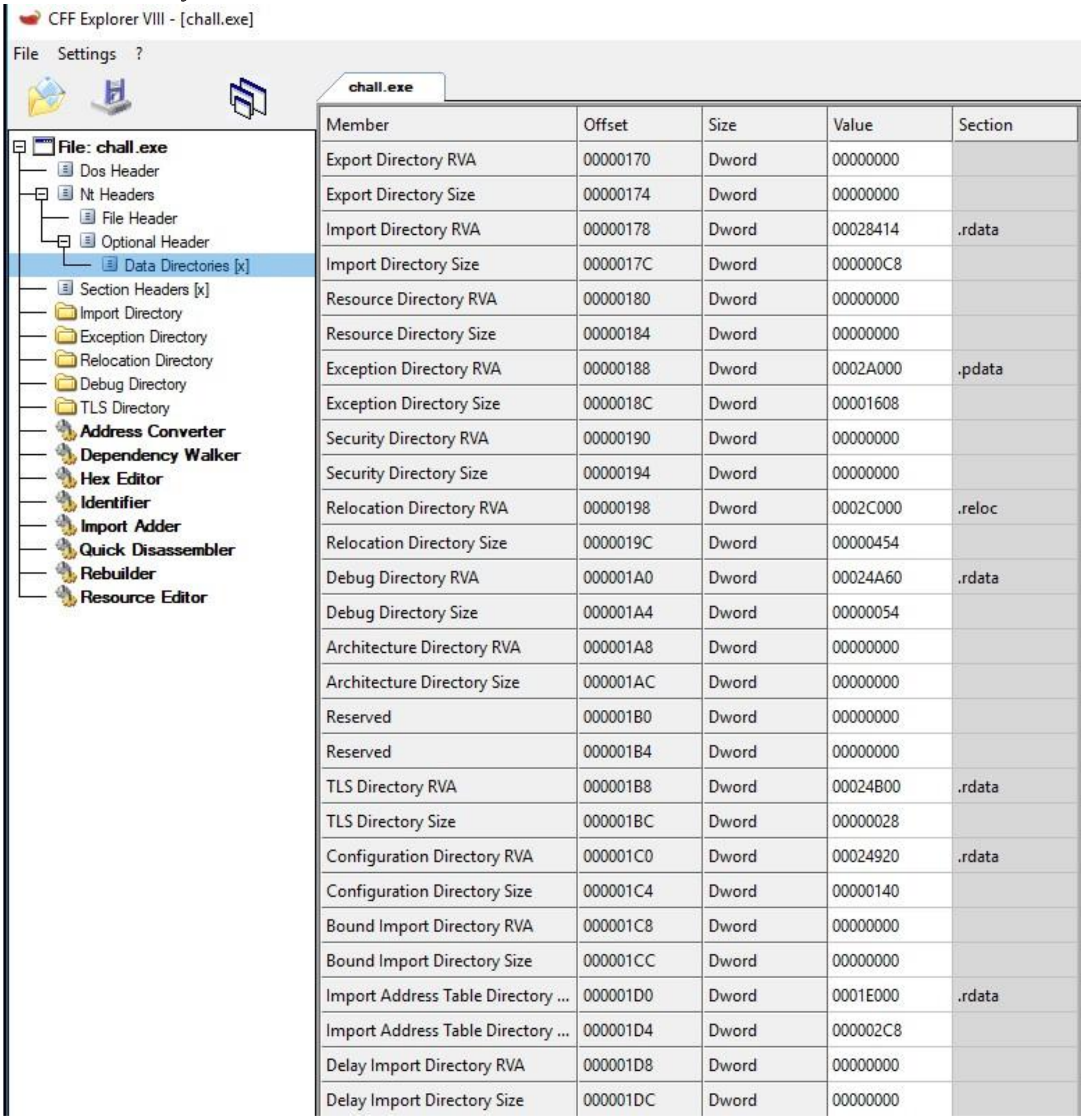
```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16
```

An IMAGE_DATA_DIRETORY structure is defines as follows:

It’s a very simple structure with only two members, first one being an RVA pointing to the start of the Data Directory and the second one being the size of the

```
typedef struct _IMAGE_DATA_DIRECTORY {
   DWORD   VirtualAddress;
   DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Data DirectoryIMAGE_DATA_DIRECTORY entries:



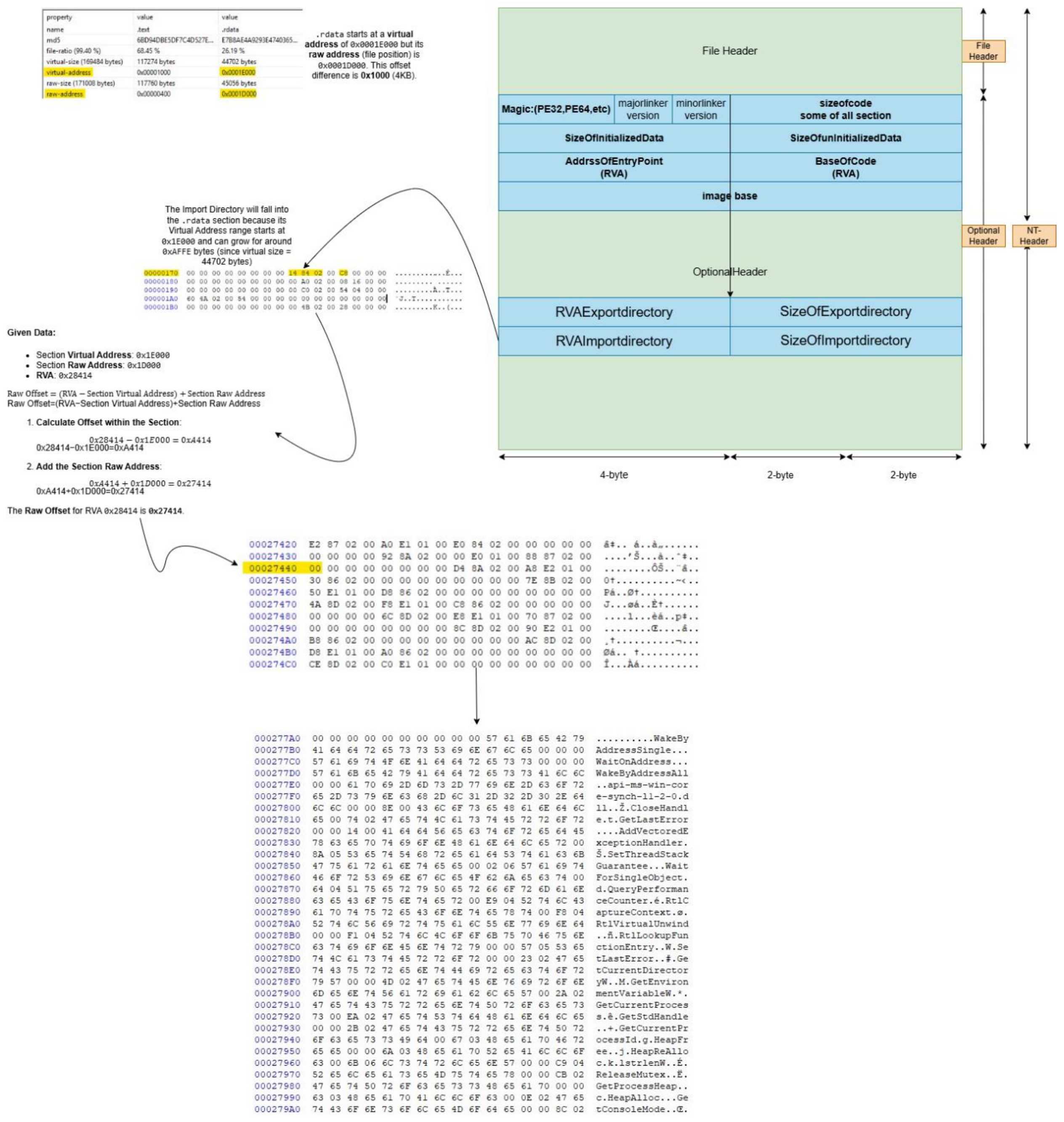
Member	Offset	Size	Value	Section
Export Directory RVA	00000170	Dword	00000000	
Export Directory Size	00000174	Dword	00000000	
Import Directory RVA	00000178	Dword	00028414	.rdata
Import Directory Size	0000017C	Dword	000000C8	
Resource Directory RVA	00000180	Dword	00000000	
Resource Directory Size	00000184	Dword	00000000	
Exception Directory RVA	00000188	Dword	0002A000	.pdata
Exception Directory Size	0000018C	Dword	00001608	
Security Directory RVA	00000190	Dword	00000000	
Security Directory Size	00000194	Dword	00000000	
Relocation Directory RVA	00000198	Dword	0002C000	.reloc
Relocation Directory Size	0000019C	Dword	00000454	
Debug Directory RVA	000001A0	Dword	00024A60	.rdata
Debug Directory Size	000001A4	Dword	00000054	
Architecture Directory RVA	000001A8	Dword	00000000	
Architecture Directory Size	000001AC	Dword	00000000	
Reserved	000001B0	Dword	00000000	
Reserved	000001B4	Dword	00000000	
TLS Directory RVA	000001B8	Dword	00024B00	.rdata
TLS Directory Size	000001BC	Dword	00000028	
Configuration Directory RVA	000001C0	Dword	00024920	.rdata
Configuration Directory Size	000001C4	Dword	00000140	
Bound Import Directory RVA	000001C8	Dword	00000000	
Bound Import Directory Size	000001CC	Dword	00000000	
Import Address Table Directory ...	000001D0	Dword	0001E000	.rdata
Import Address Table Directory ...	000001D4	Dword	000002C8	
Delay Import Directory RVA	000001D8	Dword	00000000	
Delay Import Directory Size	000001DC	Dword	00000000	

Before we dive deep into the section header, we need to explain where each directory is located.

As you can see, there are multiple columns containing values. These values provide information about the location of each directory

We have already explained the RVA and the Raw Size (real size). Now, we are going to explain the sections, but we will focus on the section columns that indicate where each directory is located.

We also found that every directory has two rows—one for the RVA and another for the size. The first row explains the location of the RVA of the directory. I will provide a graph that explains this.



property	value	value
name	.text	.rdata
md5	6BD94DBE5DF7C4D527E...	E7B8AE4A9293E4740365...
file-ratio (99.40 %)	68.45 %	26.19 %
virtual-size (169484 bytes)	117274 bytes	44702 bytes
virtual-address	0x00001000	0x0001E000
raw-size (171008 bytes)	117760 bytes	45056 bytes
raw-address	0x00000400	0x0001D000

`.rdata` starts at a **virtual address** of `0x0001E000` but its **raw address** (file position) is `0x0001D000`. This offset difference is **0x1000** (4KB).

The Import Directory will fall into the `.rdata` section because its Virtual Address range starts at `0x1E000` and can grow for around `0xAFFE` bytes (since virtual size = 44702 bytes)

```
00000170  00 00 00 00 00 00 00 00 14 84 02 00 C8 00 00 00  ..........„..È...
00000180  00 00 00 00 00 00 00 00 00 A0 02 00 08 16 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 C0 02 00 54 04 00 00  .........À..T...
000001A0  60 4A 02 00 54 00 00 00 00 00 00 00 00 00 00 00  `J..T...........
000001B0  00 00 00 00 00 00 00 00 00 4B 02 00 28 00 00 00  .........K..(...
```

Given Data:

- Section **Virtual Address**: `0x1E000`
- Section **Raw Address**: `0x1D000`
- **RVA**: `0x28414`

$$\text{Raw Offset} = (\text{RVA} - \text{Section Virtual Address}) + \text{Section Raw Address}$$
Raw Offset=(RVA−Section Virtual Address)+Section Raw Address

1. **Calculate Offset within the Section**:

$$0x28414 - 0x1E000 = 0xA414$$
0x28414−0x1E000=0xA414

2. **Add the Section Raw Address**:

$$0xA414 + 0x1D000 = 0x27414$$
0xA414+0x1D000=0x27414

The **Raw Offset** for RVA `0x28414` is **`0x27414`**.

```
00027420  E2 87 02 00 A0 E1 01 00 E0 84 02 00 00 00 00 00  â‡.. á..à„......
00027430  00 00 00 00 92 8A 02 00 00 E0 01 00 88 87 02 00  ....’Š...à..ˆ‡..
00027440  00 00 00 00 00 00 00 00 D4 8A 02 00 A8 E2 01 00  ........ÔŠ..¨â..
00027450  30 86 02 00 00 00 00 00 00 00 00 00 7E 8B 02 00  0†..........~‹..
00027460  50 E1 01 00 D8 86 02 00 00 00 00 00 00 00 00 00  Pá..Ø†..........
00027470  4A 8D 02 00 F8 E1 01 00 C8 86 02 00 00 00 00 00  J...øá..È†......
00027480  00 00 00 00 6C 8D 02 00 E8 E1 01 00 70 87 02 00  ....l...èá..p‡..
00027490  00 00 00 00 00 00 00 00 8C 8D 02 00 90 E2 01 00  ........Œ....â..
000274A0  B8 86 02 00 00 00 00 00 00 00 00 00 AC 8D 02 00  ¸†..........¬...
000274B0  D8 E1 01 00 A0 86 02 00 00 00 00 00 00 00 00 00  Øá.. †..........
000274C0  CE 8D 02 00 C0 E1 01 00 00 00 00 00 00 00 00 00  Î...Àá..........
```

```
000277A0  00 00 00 00 00 00 00 00 00 00 57 61 6B 65 42 79  ..........WakeBy
000277B0  41 64 64 72 65 73 73 53 69 6E 67 6C 65 00 00 00  AddressSingle...
000277C0  57 61 69 74 4F 6E 41 64 64 72 65 73 73 00 00 00  WaitOnAddress...
000277D0  57 61 6B 65 42 79 41 64 64 72 65 73 73 41 6C 6C  WakeByAddressAll
000277E0  00 00 61 70 69 2D 6D 73 2D 77 69 6E 2D 63 6F 72  ..api-ms-win-cor
000277F0  65 2D 73 79 6E 63 68 2D 6C 31 2D 32 2D 30 2E 64  e-synch-l1-2-0.d
00027800  6C 6C 00 00 8E 00 43 6C 6F 73 65 48 61 6E 64 6C  ll..Ž.CloseHandl
00027810  65 00 74 02 47 65 74 4C 61 73 74 45 72 72 6F 72  e.t.GetLastError
00027820  00 00 14 00 41 64 64 56 65 63 74 6F 72 65 64 45  ....AddVectoredE
00027830  78 63 65 70 74 69 6F 6E 48 61 6E 64 6C 65 72 00  xceptionHandler.
00027840  8A 05 53 65 74 54 68 72 65 61 64 53 74 61 63 6B  Š.SetThreadStack
00027850  47 75 61 72 61 6E 74 65 65 00 02 06 57 61 69 74  Guarantee...Wait
00027860  46 6F 72 53 69 6E 67 6C 65 4F 62 6A 65 63 74 00  ForSingleObject.
00027870  64 04 51 75 65 72 79 50 65 72 66 6F 72 6D 61 6E  d.QueryPerforman
00027880  63 65 43 6F 75 6E 74 65 72 00 E9 04 52 74 6C 43  ceCounter.é.RtlC
00027890  61 70 74 75 72 65 43 6F 6E 74 65 78 74 00 F8 04  aptureContext.ø.
000278A0  52 74 6C 56 69 72 74 75 61 6C 55 6E 77 69 6E 64  RtlVirtualUnwind
000278B0  00 00 F1 04 52 74 6C 4C 6F 6F 6B 75 70 46 75 6E  ..ñ.RtlLookupFun
000278C0  63 74 69 6F 6E 45 6E 74 72 79 00 00 57 05 53 65  ctionEntry..W.Se
000278D0  74 4C 61 73 74 45 72 72 6F 72 00 00 23 02 47 65  tLastError..#.Ge
000278E0  74 43 75 72 72 65 6E 74 44 69 72 65 63 74 6F 72  tCurrentDirector
000278F0  79 57 00 00 4D 02 47 65 74 45 6E 76 69 72 6F 6E  yW..M.GetEnviron
00027900  6D 65 6E 74 56 61 72 69 61 62 6C 65 57 00 2A 02  mentVariableW.*.
00027910  47 65 74 43 75 72 72 65 6E 74 50 72 6F 63 65 73  GetCurrentProces
00027920  73 00 EA 02 47 65 74 53 74 64 48 61 6E 64 6C 65  s.ê.GetStdHandle
00027930  00 00 2B 02 47 65 74 43 75 72 72 65 6E 74 50 72  ..+.GetCurrentPr
00027940  6F 63 65 73 73 49 64 00 67 03 48 65 61 70 46 72  ocessId.g.HeapFr
00027950  65 65 00 00 6A 03 48 65 61 70 52 65 41 6C 6C 6F  ee..j.HeapReAllo
00027960  63 00 6B 06 6C 73 74 72 6C 65 6E 57 00 00 C9 04  c.k.lstrlenW..É.
00027970  52 65 6C 65 61 73 65 4D 75 74 65 78 00 00 CB 02  ReleaseMutex..Ë.
00027980  47 65 74 50 72 6F 63 65 73 73 48 65 61 70 00 00  GetProcessHeap..
00027990  63 03 48 65 61 70 41 6C 6C 6F 63 00 0E 02 47 65  c.HeapAlloc...Ge
000279A0  74 43 6F 6E 73 6F 6C 65 4D 6F 64 65 00 00 8C 02  tConsoleMode..Œ.
```

So, we need to explain the benefits of these directories:

1. Export Table

- **Why it's useful**: The Export Table helps make your program modular. Think of it like a "menu" that tells other programs, "Hey, here are the functions or data I offer that you can use." It makes your code more flexible because other programs or DLLs can "borrow" these functions without needing to know the details of how they're built.
    - **Benefit**: It allows programs to share common functions or libraries, so instead of each program reinventing the wheel, they can just link to the same code. It also makes updates easier because you can change the function in the DLL without touching the main program.
- **Example**: A music player app could rely on a sound-processing library (DLL). The DLL exports a function to play a song, and other programs can use that same function without needing to know how it works.

2. Import Table

- **Why it's useful**: The Import Table is like a guest list for a party—telling your program where to find the things it needs from outside (like other DLLs or libraries). If your program needs to use something from another DLL, like opening a file or drawing a window, it looks at the Import Table to see what it should load and from where.
    - **Benefit**: This makes your program lighter and more adaptable. Instead of putting everything inside one big file, your program can "ask" external libraries for the pieces it needs. This saves space and allows the program to be updated or changed easily.
- **Example**: If your program needs to create a window or draw on the screen, it will check the Import Table to find the CreateWindow function in user32.dll. The system will load the necessary code for you.

3. Resource Table

- **Why it's useful**: The Resource Table is like the organizer of your program's "assets"—icons, images, strings, and anything else that isn't actual code. It helps you keep everything your program needs in one place, separate from the code, which makes managing and updating them easier.

    - **Benefit**: It helps developers manage resources (like text labels or icons) without needing to recompile the entire program. Plus, it makes the program more adaptable, for example by allowing different languages or themes.
- **Example**: A game might store all of its characters' names, dialogues, and icons in the Resource Table, so it can quickly switch between languages or update artwork without touching the core game code.

4. Relocation Table

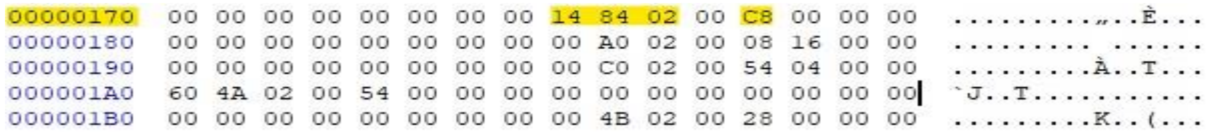
- **Why it's useful**: The Relocation Table is like a guide that tells the program how to adjust its memory addresses when it's loaded into RAM. This is important because the program might not always load at the same memory address, so the Relocation Table ensures everything points to the right place, no matter where it is in memory.
    - **Benefit**: It allows the program to be flexible and safe. The operating system can load it anywhere in memory without worrying about the program crashing because memory addresses don't match up. It also helps with security features like ASLR (Address Space Layout Randomization), which makes it harder for hackers to predict where the code is.
- **Example**: If a program is loaded into memory at a different location each time (thanks to ASLR), the Relocation Table makes sure that all function calls and variables still point to the right place in memory.

5. TLS (Thread Local Storage) Table

- **Why it's useful**: The TLS Table is like a special storage area for each thread in a multi-threaded program. Threads are like workers, and each one needs its own set of tools or data. The TLS Table ensures that each worker (thread) has its own set of data without stepping on each other's toes.
    - **Benefit**: It makes multi-threaded programs more efficient and avoids conflicts. Threads can safely access their own private data, without needing to share or lock data between them, which speeds up performance.

- **Example**: If your program is a web server handling multiple users, each thread could store session data (like user preferences) in the TLS Table, so each thread works independently without affecting others.

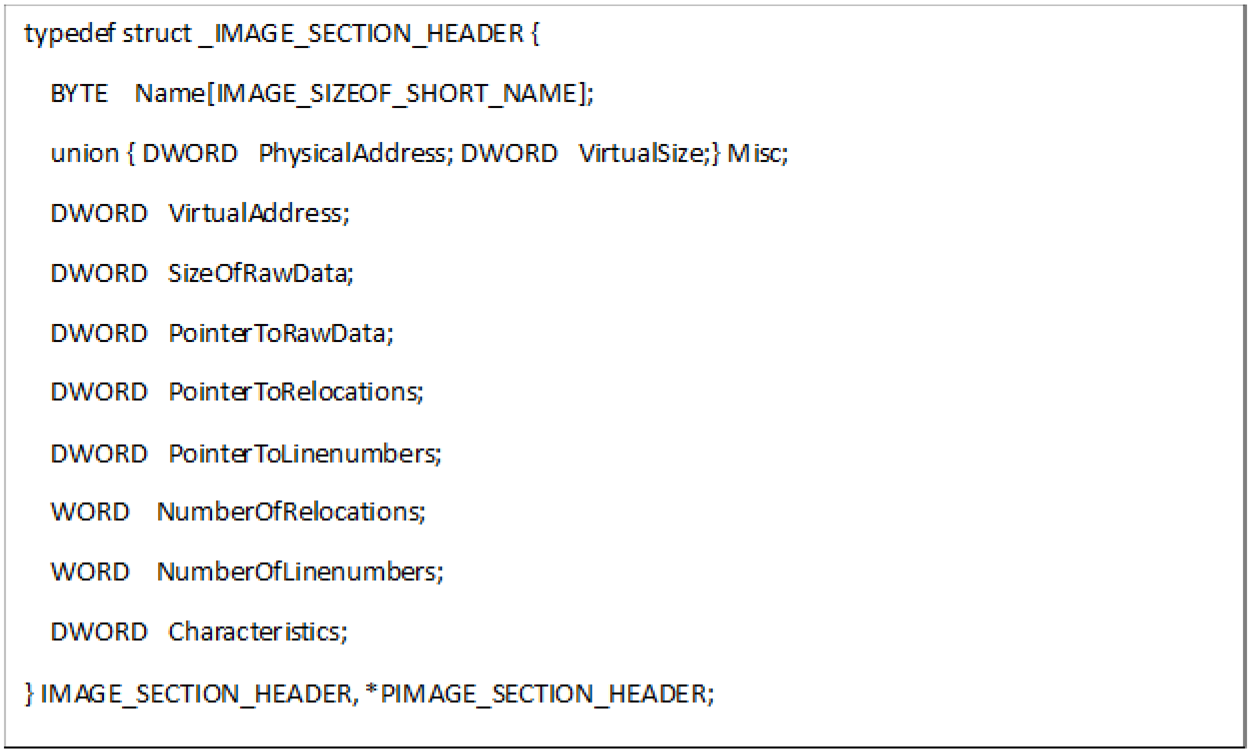
We will take the import directory example so we can find that offset of rva is 0x178 when we going to this offset at any hexeditor :



```
00000170  00 00 00 00 00 00 00 00 14 84 02 00 C8 00 00 00  .........„..È...
00000180  00 00 00 00 00 00 00 00 00 A0 02 00 08 16 00 00  ......... ......
00000190  00 00 00 00 00 00 00 00 00 C0 02 00 54 04 00 00  .........À..T...
000001A0  60 4A 02 00 54 00 00 00 00 00 00 00 00 00 00 00  `J..T...........
000001B0  00 00 00 00 00 00 00 00 00 4B 02 00 28 00 00 00  .........K..(...
```

Now we can find the value of the RVA 0x28414, which, when calculated, gives us the starting address of the import directory. Additionally, we can see the size of the directory. We have also found that the offset 0x178 is a pointer to the starting address of the import directory, and the offset 0x17C holds the value of its size.

Now we will explain the easiest and most important header, which is the **Section Header**. Section Header is a structure named IMAGE_SECTION_HEADER defined in winnt.h as follows:



```
typedef struct _IMAGE_SECTION_HEADER {

   BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];

   union { DWORD   PhysicalAddress; DWORD   VirtualSize;} Misc;

   DWORD   VirtualAddress;

   DWORD   SizeOfRawData;

   DWORD   PointerToRawData;

   DWORD   PointerToRelocations;

   DWORD   PointerToLinenumbers;

   WORD    NumberOfRelocations;

   WORD    NumberOfLinenumbers;

   DWORD   Characteristics;

} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Each section in a PE file, like .text (code), .data (data), and others, is described by such a header. Let’s break down the fields:

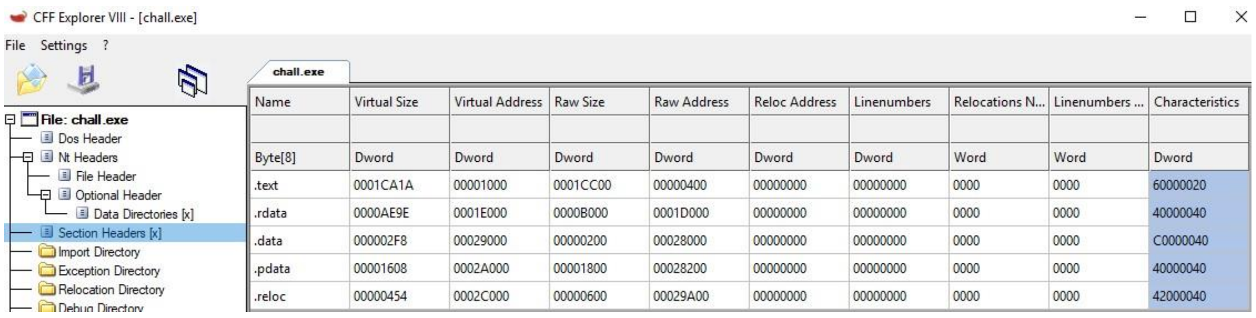
Field-by-Field Explanation:

1. **DWORD VirtualAddress**:
   - Represents the **relative virtual address (RVA)** of the section.
   - This is the address in memory where the section will be loaded, relative to the base address of the image.
2. **DWORD SizeOfRawData**:
   - Indicates the **size of the section's data** in the file.
   - This is the actual size of the data stored in the PE file on disk. It might be padded to align with a section alignment boundary.
3. **DWORD PointerToRawData**:
   - Points to the **starting location of the section's data** in the PE file.
   - This offset tells where to find the section's raw data on disk.
4. **DWORD PointerToRelocations**:
   - Offset to the section's **relocation entries** within the file (if applicable).
   - Relocations are used for adjusting addresses when the module is loaded at a different base address. Generally, this field is 0 for executable images, as relocations are not stored in the section headers.
5. **DWORD PointerToLinenumbers**:
   - Offset to **line-number information** for debugging.
   - In modern PE files, this field is often 0, as line numbers are typically stored in debug information rather than in the section headers.
6. **WORD NumberOfRelocations**:
   - Number of **relocation entries** for the section.
   - This is only relevant if the section contains relocations (e.g., for object files). For executables, this is often 0.
7. **WORD NumberOfLinenumbers**:
   - Number of **line-number entries**.

- Again, this is mostly 0 for modern PE files and is relevant primarily for debugging.

8. **DWORD Characteristics**:
    - Flags describing the **characteristics of the section**.
    - This field contains a combination of bit flags indicating properties like whether the section is executable, writable, readable, etc. Examples:
        - 0x20000000: Contains executable code.
        - 0x40000000: Contains initialized data.
        - 0x80000000: Contains uninitialized data.

I will take a screenshot of this header from CFF Explorer, then I will explain it using PE Studio.


CFF Explorer VIII - [chall.exe]

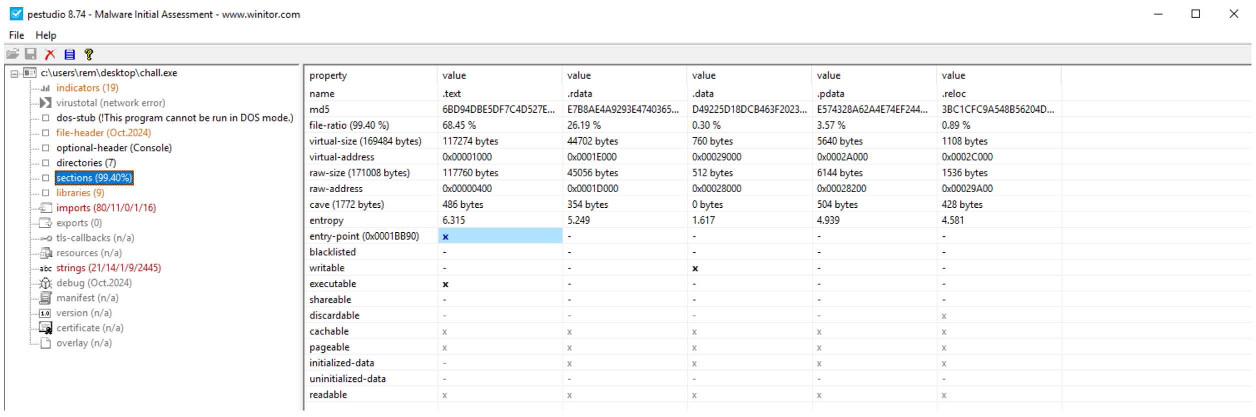
File Settings ?

chall.exe

File: chall.exe
- Dos Header
- Nt Headers
  - File Header
  - Optional Header
    - Data Directories [x]
- Section Headers [x]
- Import Directory
- Exception Directory
- Relocation Directory
- Debug Directory

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0001CA1A	00001000	0001CC00	00000400	00000000	00000000	0000	0000	60000020
.rdata	0000AE9E	0001E000	0000B000	0001D000	00000000	00000000	0000	0000	40000040
.data	000002F8	00029000	00000200	00028000	00000000	00000000	0000	0000	C0000040
.pdata	00001608	0002A000	00001800	00028200	00000000	00000000	0000	0000	40000040
.reloc	00000454	0002C000	00000600	00029A00	00000000	00000000	0000	0000	42000040

Now, we will take a look at this header in PE Studio and explain each field in detail.


pestudio 8.74 - Malware Initial Assessment - www.winitor.com

File Help

c:\users\rem\desktop\chall.exe
- indicators (19)
- virustotal (network error)
- dos-stub (!This program cannot be run in DOS mode.)
- file-header (Oct.2024)
- optional-header (Console)
- directories (7)
- sections (99.40%)
- libraries (9)
- imports (80/11/0/1/16)
- exports (0)
- tls-callbacks (n/a)
- resources (n/a)
- strings (21/14/1/9/2445)
- debug (Oct.2024)
- manifest (n/a)
- version (n/a)
- certificate (n/a)
- overlay (n/a)

property	value	value	value	value	value
name	.text	.rdata	.data	.pdata	.reloc
md5	6BD94DBE5DF7C4D527E...	E7B8AE4A9293E4740365...	D49225D18DCB463F2023...	E574328A62A4E74EF244...	3BC1CFC9A548B56204D...
file-ratio (99.40 %)	68.45 %	26.19 %	0.30 %	3.57 %	0.89 %
virtual-size (169484 bytes)	117274 bytes	44702 bytes	760 bytes	5640 bytes	1108 bytes
virtual-address	0x00001000	0x0001E000	0x00029000	0x0002A000	0x0002C000
raw-size (171008 bytes)	117760 bytes	45056 bytes	512 bytes	6144 bytes	1536 bytes
raw-address	0x00000400	0x0001D000	0x00028000	0x00028200	0x00029A00
cave (1772 bytes)	486 bytes	354 bytes	0 bytes	504 bytes	428 bytes
entropy	6.315	5.249	1.617	4.939	4.581
entry-point (0x0001BB90)	x	-	-	-	-
blacklisted	-	-	-	-	-
writable	-	-	x	-	-
executable	x	-	-	-	-
shareable	-	-	-	-	-
discardable	-	-	-	-	x
cachable	x	x	x	x	x
pageable	x	x	x	x	x
initialized-data	-	x	x	x	x
uninitialized-data	-	-	-	-	-
readable	x	x	x	x	x

As you can see there is 5 sections as we knowen before, we will explain it and claculate how can we know where it is starting and where the endinf of it

1. .text Section: The Brain Department

This is the most important department—it holds the actual instructions the program needs to follow. Think of it as the brain of the building where all decisions are made. It’s protected so that nobody can mess with it because changing these instructions would break the entire program.

2. .rdata Section: The Archive Room

This section is like a locked archive where the program stores permanent information it needs to refer to but doesn’t want to change. Things like phone directories (constants) or fixed rules (strings) are kept here. It also includes some important files that tell the program how to talk to other buildings (libraries or DLLs).

3. .data Section: The Workspace

Here’s the messy but essential workspace of the program. Employees (the program) use this area to keep notes, track numbers, or update variables that change during the day. Unlike the archive, this space is flexible and designed for day-to-day work.

4. .pdata Section: The Emergency Handbook

This department is the safety officer of the building. If something goes wrong—like an unexpected error or fire alarm—it knows exactly what to do. It contains the program’s emergency response plan, so it can handle crashes or exceptions without falling apart.

5. .reloc Section: The Moving Crew

This is the team that comes in when the office building can’t be set up in its usual spot. If another program is already using its preferred location in memory, this team helps it move everything to a new spot and still work perfectly. It’s all about flexibility and avoiding conflicts.

After we have seen the meaning of each section, we now need to explain the meaning of each value in the previous image.

MD5 Value

Think of the MD5 value like a fingerprint for the file. It's a unique string of numbers and letters that identifies the file. Just like every person has a unique fingerprint, every file has a unique MD5 hash. If the file changes even a little, the MD5 value will change too. It's a quick way to check if the file has been tampered with or altered.

File-Ratio

This is the comparison between the file size when it's in memory and the file size when it's stored on disk. If the file gets compressed or changes in any way during execution, the file-ratio tells us how much it's being altered as it runs.

Virtual Size

Imagine you're looking at a map of a city. The virtual size tells us the total amount of memory the program needs when it's running, like how much space the city (the program) needs on the map. It tells us how much "virtual" memory the program will take up in the system when it's loaded.

Virtual Address

This is the starting address of where the program's code or data will live in memory. It's like giving the program a specific spot on a street where it can set up its office (its space in RAM). It's not the actual memory location, but rather a logical address that the system uses to find it.

Raw Size

This is the actual size of the file as it exists on disk, before it's loaded into memory. It's like the file in its "storage" form. It doesn't take into account any adjustments or padding done when the file is running, just the raw amount of space it takes up.

Raw Address

Think of this as the "starting point" where the file begins on disk. It's like the starting point of a road trip—it tells the system where to begin looking for the file in its physical location on the hard drive.

Cave

A "cave" refers to hidden or unused areas in a file. It's like an empty room in a house that the file doesn't use, but it's there and available. Hackers might sometimes use these caves to hide malicious code or other secrets without it being noticed.

Entropy

Entropy is a fancy way of saying "how random is this file?" If a file has high entropy, it’s very unpredictable and hard to compress or analyze, like trying to read a scrambled message. Low entropy means the file is more organized and predictable, kind of like reading a well-structured document.

Entry Point

This is where the program begins its execution. It’s like the starting line in a race—the point where the operating system hands control to the program and tells it, "Go!" It marks the first instruction the program will run when it starts.

Now, we will dive deeper into the directories found in our example and explain them in detail.

Import Directory:

We have been know the benfits from this directory but not in details so we will go further to see what is it. The **Import Directory** in a PE file is basically a list of all the functions a program needs from other DLLs to work properly. and it is a Data Directory located at the beginning of the .idata section.Think of it as a "shopping list" for the program, telling Windows:

1. **Which DLLs to load** (like kernel32.dll or user32.dll).
2. **Which functions from those DLLs it needs** (like CreateFileA or MessageBoxA).

When you run the program, Windows looks at this list, loads the required DLLs, and connects the program to those functions, so it can use them during execution.

It’s like calling a friend for help and telling them:

- **Which friend to call** (the DLL).
- **What help you need** (the function name).

The image of IMAGE_IMPORT_DESCRIPTOR is defined as follows: each one of them is for a DLL.

It doesn't have a fixed size, so the last IMAGE_IMPORT_DESCRIPTOR of the array is zeroed-out (NULL-Padded) to indicate the end of the Import Directory Table.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
  union {
    DWORD  Characteristics;
    DWORD  OriginalFirstThunk;
  } DUMMYUNIONNAME;
  DWORD  TimeDateStamp;
  DWORD  ForwarderChain;
  DWORD  Name;
  DWORD  FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED
*PIMAGE_IMPORT_DESCRIPTOR;
```

1. DUMMYUNIONNAME:

This is a **union**, meaning it can store **either** Characteristics or OriginalFirstThunk, but not both at the same time.

- **Characteristics**: A reserved field, rarely used.
- **OriginalFirstThunk**: A pointer to the **OFT** (Original First Thunk), which is a list of all the functions the program wants from the DLL.
  Think of it as the "shopping list" for the program.

2. TimeDateStamp:

This field is for **timestamping or binding information**, but most of the time, it's set to 0.
If you think of the PE as a logbook, this might tell when the imports were last updated—but only if it's used.

3. ForwarderChain:

This is used if a function is **forwarded** to another DLL. (For example, one DLL might say, "I don't have this function, but someother.dll does.")
If there's no forwarding, this value is typically 0.

4. Name:

This is a **pointer to the name of the DLL** the program is importing from. For example, it might point to the string "user32.dll".

5. FirstThunk:

This is a pointer to the **FT** (First Thunk), which is the actual table of function pointers. After Windows resolves all the imports, this table will contain the **real memory addresses of the functions** from the DLL.

6. PIMAGE_IMPORT_DESCRIPTOR:

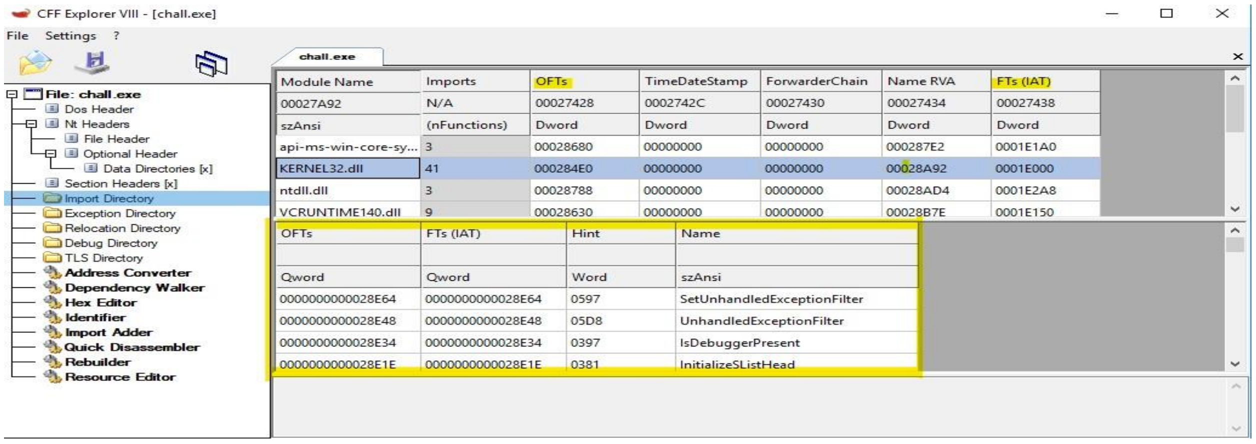
This is just a **pointer to the structure**. Adding P in front of a name is a common C-style way of saying, "This is a pointer to this type."

Imagine you're managing a "help list" for a program that says:

- "I need some help from **user32.dll**."
- "Here's the list of functions I need (OFT)."
- "Once you load the DLL, please write down the actual addresses for these functions so I can call them directly (FT)."

We need to explain some definitions to understand how the import table works.

As you can see in this screen, it shows which DLL is being called and also specifies the functions being referenced.



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
00027A92	N/A	00027428	0002742C	00027430	00027434	00027438
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
api-ms-win-core-sy...	3	00028680	00000000	00000000	000287E2	0001E1A0
KERNEL32.dll	41	000284E0	00000000	00000000	00028A92	0001E000
ntdll.dll	3	00028788	00000000	00000000	00028AD4	0001E2A8
VCRUNTIME140.dll	9	00028630	00000000	00000000	00028B7E	0001E150

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
0000000000028E64	0000000000028E64	0597	SetUnhandledExceptionFilter
0000000000028E48	0000000000028E48	05D8	UnhandledExceptionFilter
0000000000028E34	0000000000028E34	0397	IsDebuggerPresent
0000000000028E1E	0000000000028E1E	0381	InitializeSListHead

--We can now find the **import columns**, which show the number of imports being called from the module

Import Lookup Table (ILT)

Sometimes, the **Import Lookup Table (ILT)** is also referred to as the **Import Name Table (INT)**.

Each **imported DLL** has its own ILT, and the **IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk** holds the **Relative Virtual Address (RVA)** pointing to the ILT for the corresponding DLL.

The ILT is essentially a list that helps the loader identify which functions need to be imported from the DLL. It contains either the **name** or **ordinal** of each function.

ILT Structure:

- The ILT consists of an array of **32-bit numbers** (for PE32) or **64-bit numbers** (for PE32+). The last entry in the array is always **zeroed-out** to indicate the end of the ILT.

Each entry in the ILT encodes the following:

1. **Most Significant Bit (Bit 31/63)**:
   This bit is called the **Ordinal/Name flag**. It tells whether the function is being imported by **name** or by **ordinal**.
    - If the flag is **1**, the function is imported by **ordinal** (a unique number).
    - If the flag is **0**, the function is imported by **name**.
2. **Bits 15-0 (for PE32) or Bits 15-0 and 62-15 (for PE32+)**:
    - If the **Ordinal/Name flag** is **1**, these bits hold the **16-bit ordinal number** for the function to be imported. The rest of the bits (30-15/62-15) are set to **0**.
3. **Bits 30-0**:
    - If the **Ordinal/Name flag** is **0**, these bits contain the **RVA of the Hint/Name table**.

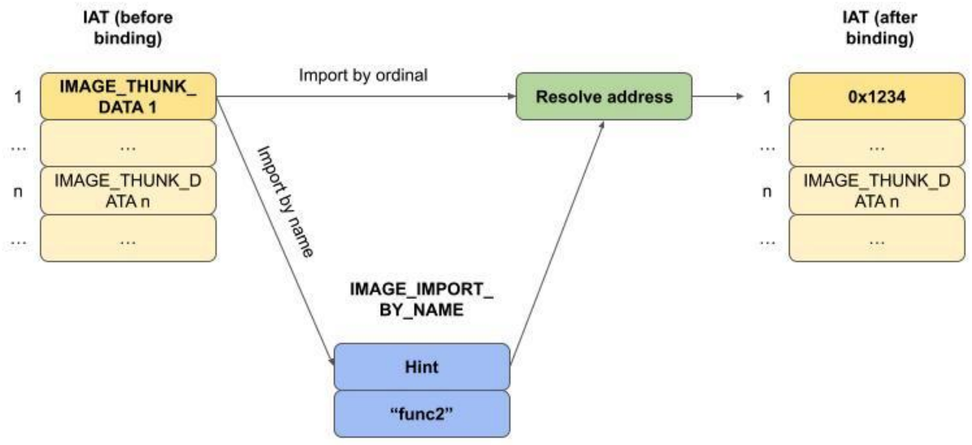
Hint/Name Table:

The **Hint/Name table** is a structure defined in **winnt.h** as IMAGE_IMPORT_BY_NAME. It contains the **name** of the function along with a **hint** (a short identifier to speed up name lookup).

```
typedef struct
_IMAGE_IMPORT_BY_NAME {

   WORD    Hint;

   CHAR   Name[1];

} IMAGE_IMPORT_BY_NAME.
```

Hint: A number used to look up a function. It's first used as an index in the export name pointer table, and if that fails, a binary search is performed on the table.

Name: A null-terminated string containing the function's name to import.



Relocation Directory:

We have explained this part in the Essential Base section, which explains how to calculate memory mapping and how the system loads the sections and other parts to their specific locations. But how does the system know the RVA for every section? This is where the Relocation Directory comes in. It functions like a table that contains the RVA locations for each part, and the system calculates these by adding the RVA to the base of the code. To summarize, it's divided into blocks, each block representing the base relocations for a 4KB page. Each block must start on a 32-bit boundary. Each block starts with an IMAGE_BASE_RELOCATION structure, followed by any number of offset field entries. The IMAGE_BASE_RELOCATION structure specifies the page RVA and the size of the relocation block.

```
typedef struct _IMAGE_BASE_RELOCATION
{DWORD   VirtualAddress;DWORD
SizeOfBlock;} IMAGE_BASE_RELOCATION;

typedef IMAGE_BASE_RELOCATION
UNALIGNED * PIMAGE_BASE_RELOCATION;
```

How a PE File is Loaded in Memory: abstract Explanation

When a **PE (Portable Executable)** file (like .exe or .dll) is executed or loaded into memory by the Windows operating system, the following steps occur. These steps describe how the OS maps the file's content (headers, code, data, etc.) into memory to prepare it for execution.

1. The Operating System Loads the File

- When you double-click an .exe file or a process explicitly loads a .dll, the Windows **Loader**:
    1. Opens the file on disk.
    2. Reads the file's **headers** to understand how to map the file into the process's virtual memory.

3. Creates a **new process environment** for the executable (if it's an .exe).

---

2. Memory Allocation

The OS allocates memory for the process based on the **ImageBase** specified in the PE file's **Optional Header**. This is the preferred address where the PE file expects to be loaded in memory.

- **ImageBase**: Starting memory address of the file (e.g., 0x400000 for most executables).
- If the **ImageBase** is unavailable (e.g., already occupied), the Loader relocates the PE file to a different address and adjusts absolute memory addresses using the **Relocation Table**.

---

3. Mapping Sections

The Loader reads the **Section Table** from the PE header and maps each section (like .text, .data, .rdata) into memory according to their **Virtual Address (VA)** and **Size**.

Key Concepts During Mapping

- **Virtual Address (VA)**: The address in memory where the section starts. Calculated as: VA=ImageBase+Section’s RVA (Relative Virtual Address)
- **Virtual Size**: The size of the section in memory (rounded to **SectionAlignment**, e.g., 0x1000 bytes).
- **Raw Size**: The size of the section on disk. If the section's virtual size is larger than its raw size, the Loader pads the section in memory with zeros.

Mapping Example

If the .text section has:

- **RVA**: 0x1000
- **Raw Size**: 0x600
- **Virtual Size**: 0x1000

- **ImageBase**: 0x400000

The section is mapped into memory at:

VA of .text=ImageBase+RVA=0x400000+0x1000=0x401000

Only the first 0x600 bytes are copied from the file; the remaining 0x400 bytes (to align to 0x1000) are filled with zeros.

4. Resolving Import Tables

The **Import Table** in the PE file lists external libraries (DLLs) and functions the executable depends on. The Loader performs the following:

1. Loads the required **DLLs** into memory.
2. Locates the addresses of the required functions in the DLLs.
3. Populates the **Import Address Table (IAT)** with the memory addresses of the required functions.

5. Relocation (If Needed)

If the PE file cannot be loaded at its preferred **ImageBase**, the Loader relocates it to a new memory address. This involves:

1. Calculating the difference between the **preferred ImageBase** and the **actual loaded base address**.
2. Adjusting all absolute memory addresses in the code using the **Relocation Table**.

6. AddressOfEntryPoint

The **Entry Point** is the starting instruction of the program, specified by:

- AddressOfEntryPoint (RVA in the Optional Header).
- Virtual Address (VA) is calculated as:
  EntryPoint VA=ImageBase+AddressOfEntryPoint\text{EntryPoint VA} =
  The OS transfers control to this address after loading is complete.

7. Execution Begins

- The Loader initializes the program's stack, heap, and global variables.
- Execution begins at the Entry Point.

Extra :

Strings:

What Are Strings in a PE File?

Strings in a PE (Portable Executable) file are sequences of readable characters stored within the binary that represent human-readable text used by the program. These strings provide insights into the program's operations and are useful for understanding or analyzing its behavior.

How Can You Find Strings in a PE File?

To locate strings, you can either extract them directly from the binary (static analysis) or analyze the program while it is running (dynamic analysis). Tools are typically used to automate this process, but the analysis relies on knowing where strings are commonly stored in the PE file.

Where Are Strings Stored in a PE File?

1. **.rdata Section:**
    - Stores constant, read-only strings such as error messages, prompts, or function names.
2. **.data Section:**
    - Stores writable strings that can be dynamically modified during the program's execution.
3. **Import Table:**
    - Contains names of the libraries and functions that the program depends on.
4. **Export Table:**
    - For DLLs, stores names of exported functions or symbols.
5. **Resource Section (.rsrc):**

- Stores metadata, dialog box text, version information, and other resources that often contain strings.

6. **Debug Sections:**
   - May include paths to source files, developer comments, or compilation metadata.
7. **Encoded or Obfuscated Strings:**
   - Strings stored in an encrypted or encoded format, often used by malware or advanced programs to evade analysis.

How Are Strings Extracted from a PE File?

1. **Identify Sections:**
   - Strings are typically found in .rdata, .data, or resource sections.
   - Locate these sections using a PE analysis tool or by examining the PE headers.
2. **Use Tools for Extraction:**
   - Static tools scan the binary for sequences of readable characters, identifying ASCII or Unicode strings.
   - Dynamic tools trace how strings are loaded or decoded during execution.
3. **Decoding Encoded Strings:**
   - If strings are obfuscated, reverse the encoding process by analyzing the decryption routines in the program.

When the PE file is loaded:

- Static strings are mapped into the program's memory as part of their respective sections (e.g., .rdata or .data).

- Dynamically generated strings are created or modified during runtime and stored in writable sections.
- Obfuscated strings are decoded or decrypted by specific functions when needed.

Summary

This guide takes you on a journey through the world of PE (Portable Executable) files, the backbone of executable programs on Windows. We start by looking at how PE files evolved from older formats like COM and MZ, showing how far things have come. You'll also see why the PE format is so powerful—offering features like dynamic linking, modularity, and strong memory management, making it a favorite for Windows applications.

We dive into the structure of a PE file, breaking down its key components like the DOS Header, NT Headers, and Section Headers. Each piece has a job to do, from guiding the OS on how to load the file to mapping its sections into memory. We also highlight tools like CFF Explorer and PE-bear that make analyzing PE files easier, even for beginners.

For those who love the nitty-gritty, there's a detailed look at headers and sections like .text, .rdata, and .data, explaining how they manage code, data, and even exceptions. We also walk through the entire process of how a PE file gets loaded into memory, step by step, making it clear why understanding this is so important for debugging or reverse engineering.

Lastly, we touch on something every reverse engineer loves: strings. From finding them in .rdata or the Import Table to extracting them for malware analysis, strings often hold the key to understanding a file's purpose.