STEFANO CESARONI

PROJECT

S10/L5

TRACE

With reference to the Malware_U3_W2_L5 file present within the folder "Exercise_Practical_U3_W2_L5 " on the desktop of the virtual machine dedicated for malware analysis, answer the following questions:

1) Which libraries are imported from the executable file?

2) What sections does the malware executable file consist of?

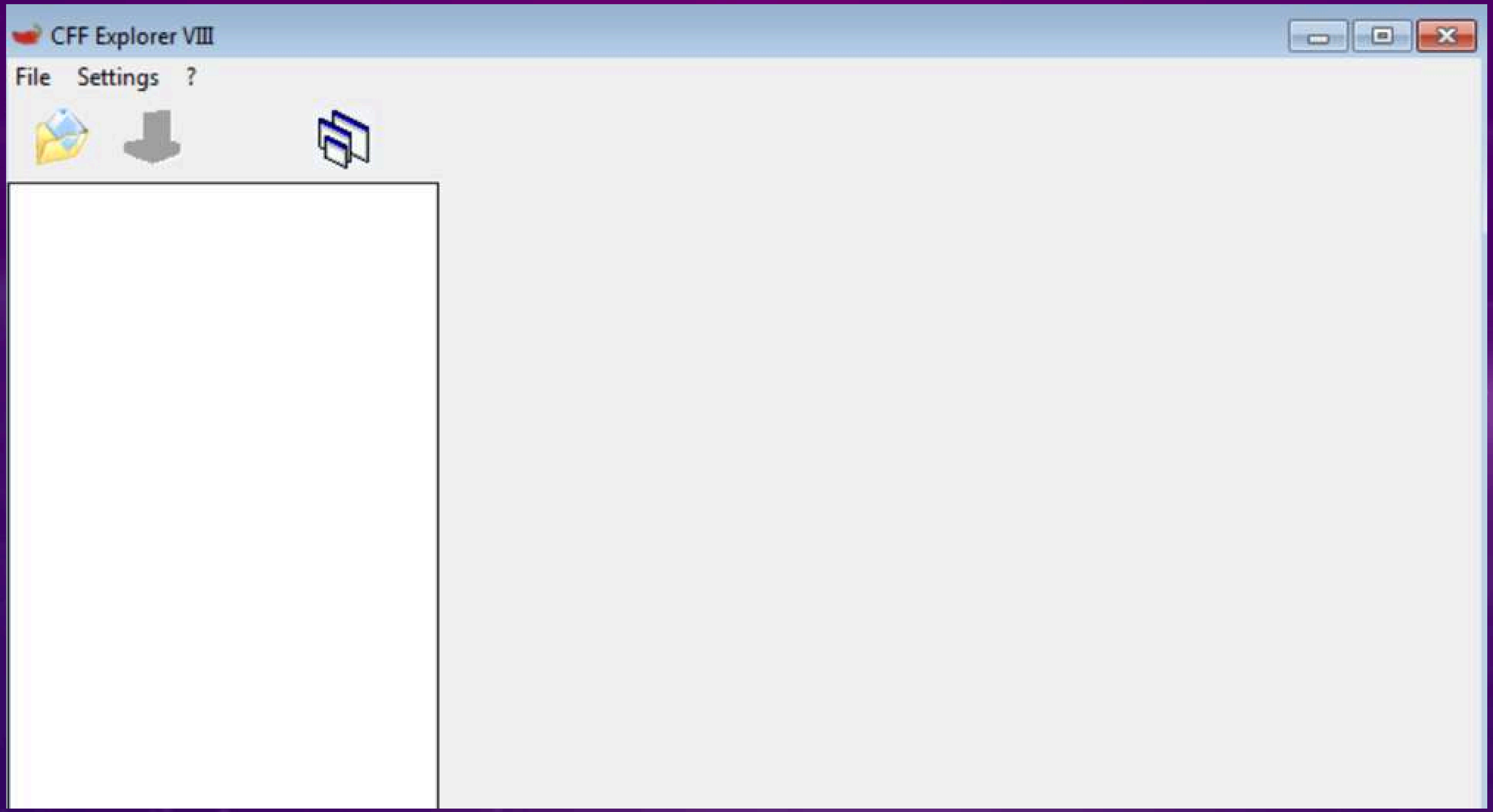
With reference to the figure on PAGE20 , answer the following questions:

3) Identify the known constructs (stack creation, possible loops);

4) Hypothesize the behavior of the feature;

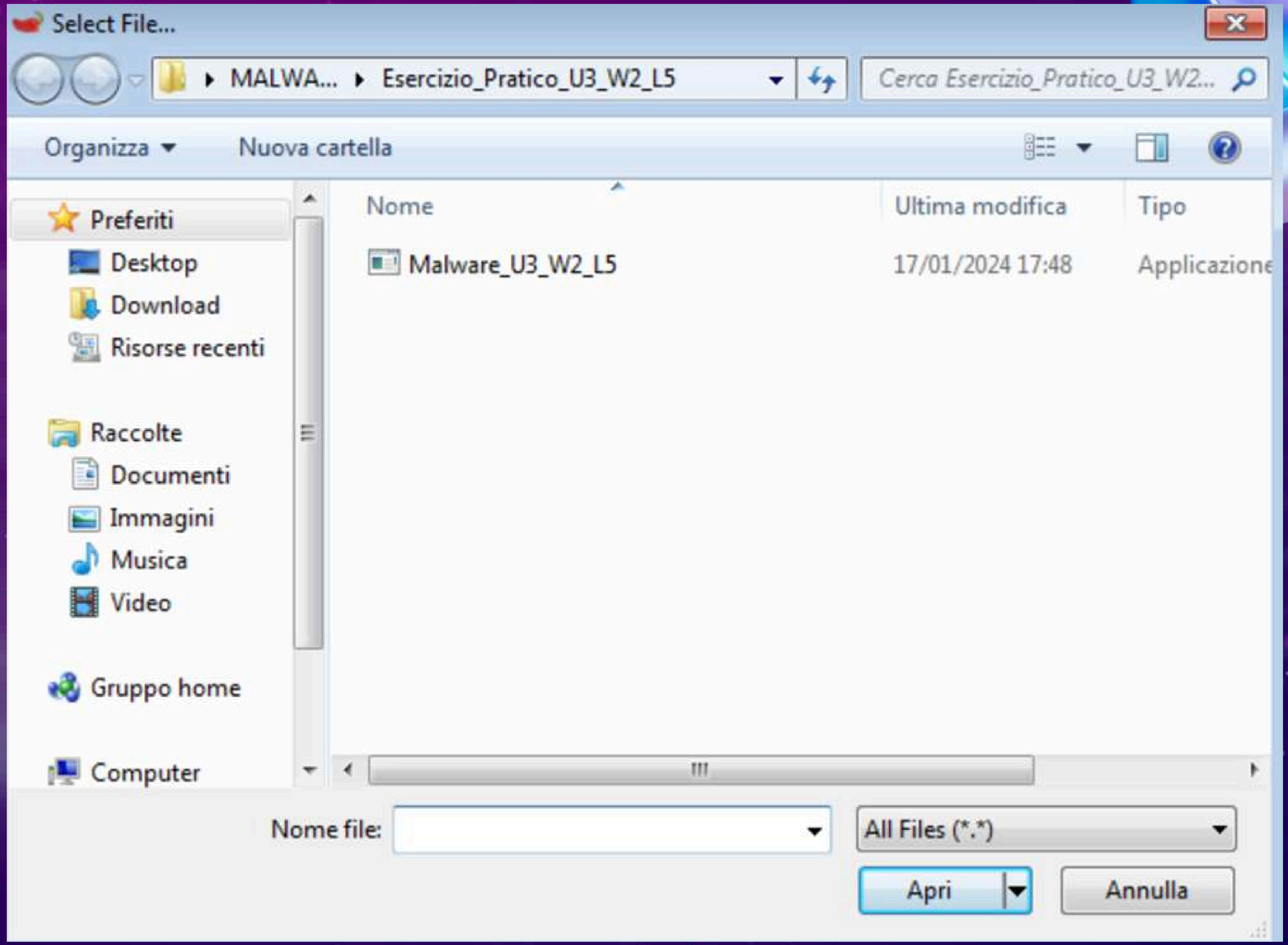
5) BONUS make table with meaning of individual lines of code.

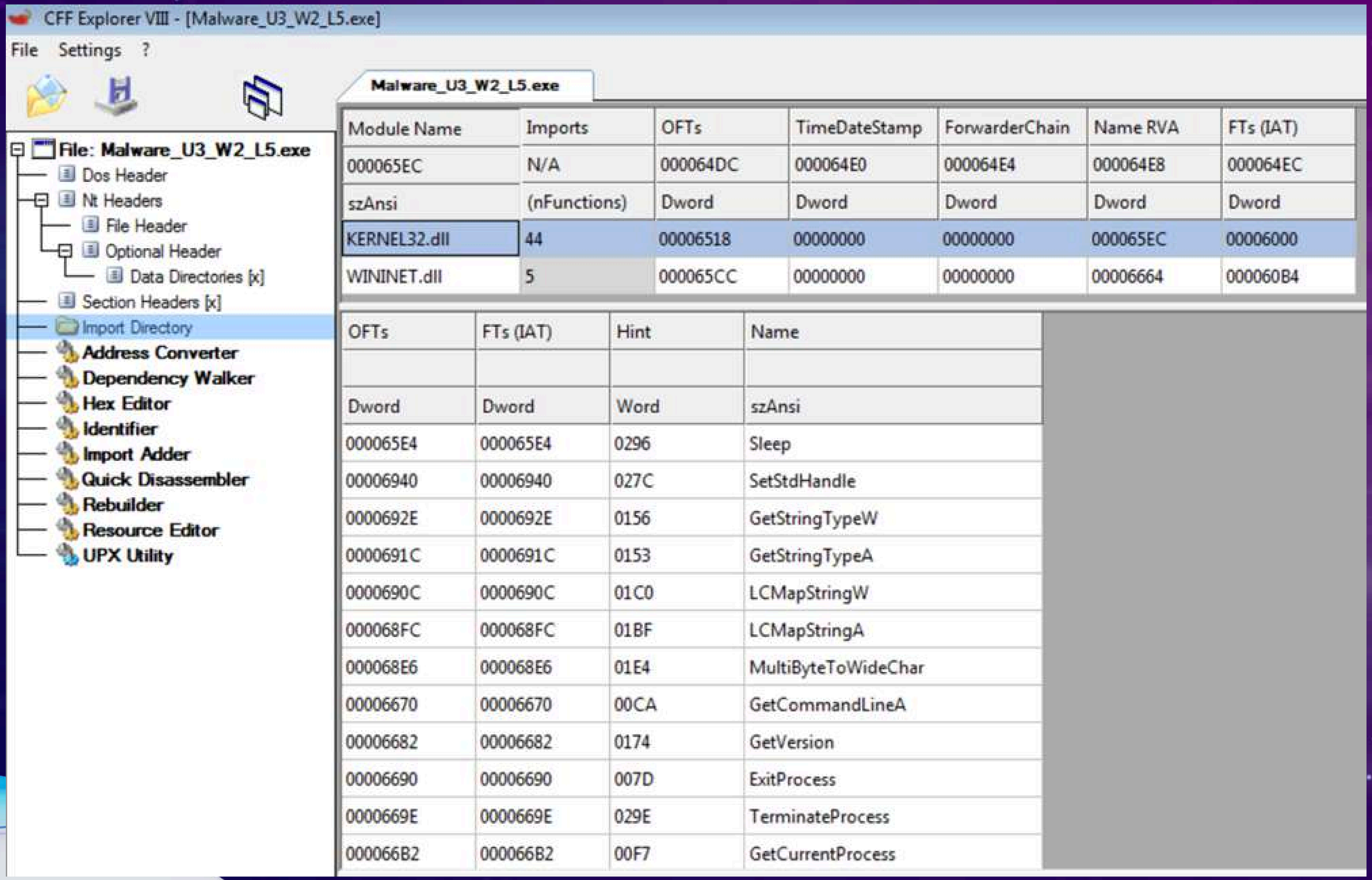
RESOLUTION FIRST PART



We are asked to do a basic static analysis for a Malware.
As a first step we go to open the TOOL, which is already installed by default on our window7, CFF EXPLORER. It is a very powerful tool that allows us to analyze and check the libraries and functions imported by a Malware, it also allows us to understand how many sections the malicious executable is composed of.
It is good to know that each section plays a particular role and it is of paramount importance to know what they are.

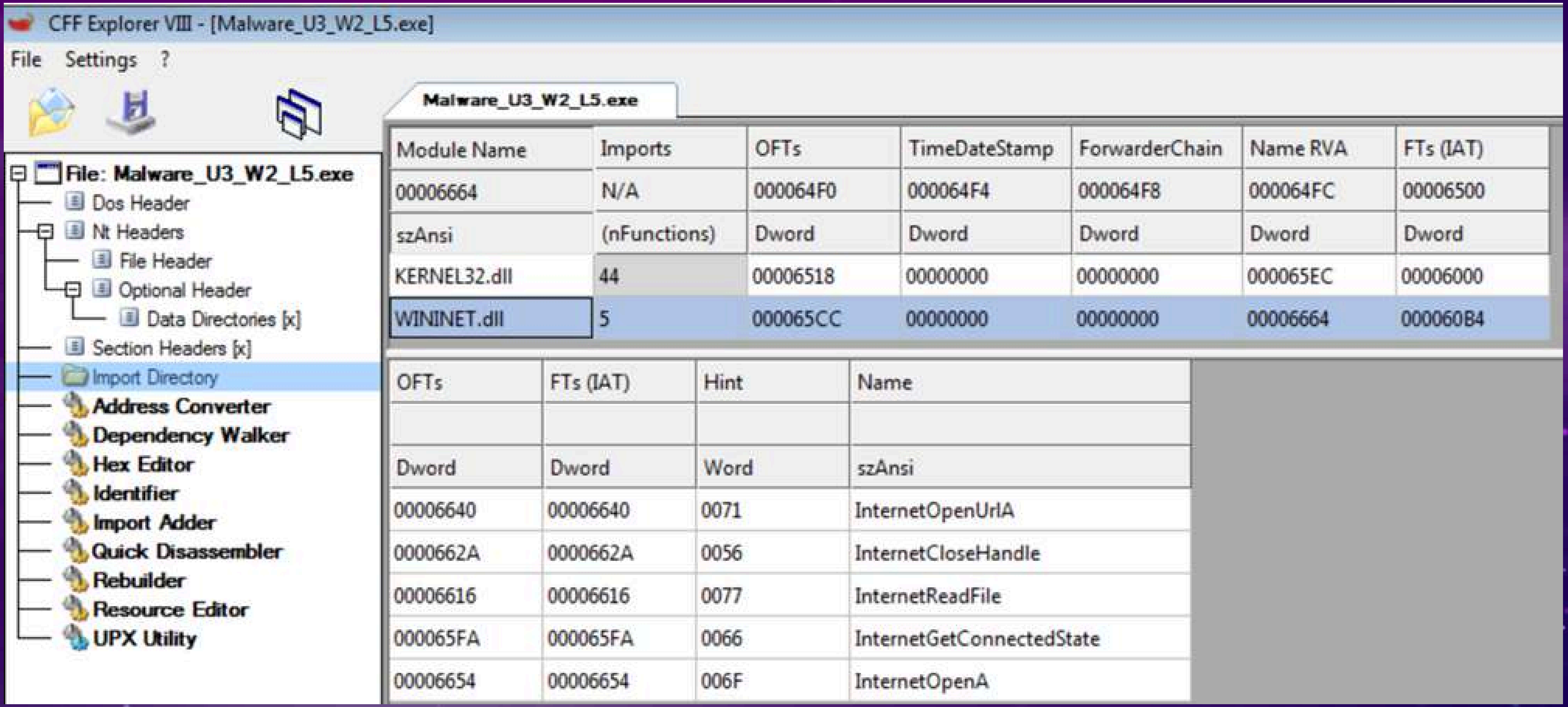
We browse into the file system looking for our malware to analyze and select it.





Once the malware is selected, this screen will appear to us. The Tool returns a lot of information, for the moment the ones we are interested in are the libraries it imported, the functions it uses, and figuring out how many sections the malware is composed of. In the left column we are going to select the "Import Directory" field. Once selected on the right we will see a table that returns the libraries used. If we select one, a table will be returned to us at the bottom showing the functions used for each library.

The image opposite shows the functions of the WININET.dll library.



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
00006664	N/A	000064F0	000064F4	000064F8	000064FC	00006500
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	44	00006518	00000000	00000000	000065EC	00006000
WININET.dll	5	000065CC	00000000	00000000	00006664	000060B4

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
00006640	00006640	0071	InternetOpenUrlA
0000662A	0000662A	0056	InternetCloseHandle
00006616	00006616	0077	InternetReadFile
000065FA	000065FA	0066	InternetGetConnectedState
00006654	00006654	006F	InternetOpenA

The libraries imported by the malware in question are:

KERNEL32.dll= contains the main functions for interacting with the operating system;

WININET.dll= contains the functions for implementing some network protocols(HTTP, HTTPS, FTP).

KERNEL32.DLL

The kernel32.dll library is one of the core system libraries of the Windows operating system. The functions included in kernel32.dll provide support for managing processes, memory, files, input/output, and other low-level system operations.

Many functions essential to the operation of the Windows operating system and applications depend on kernel32.dll. Any problems with or corruption of this library can lead to system or application malfunctions.

In addition, the kernel32.dll library is present in almost all versions of Windows, ensuring application compatibility across different versions of the operating system. As a critical system library, it is often a target for malware and exploits that try to take control of the system through kernel32.dll vulnerabilities.

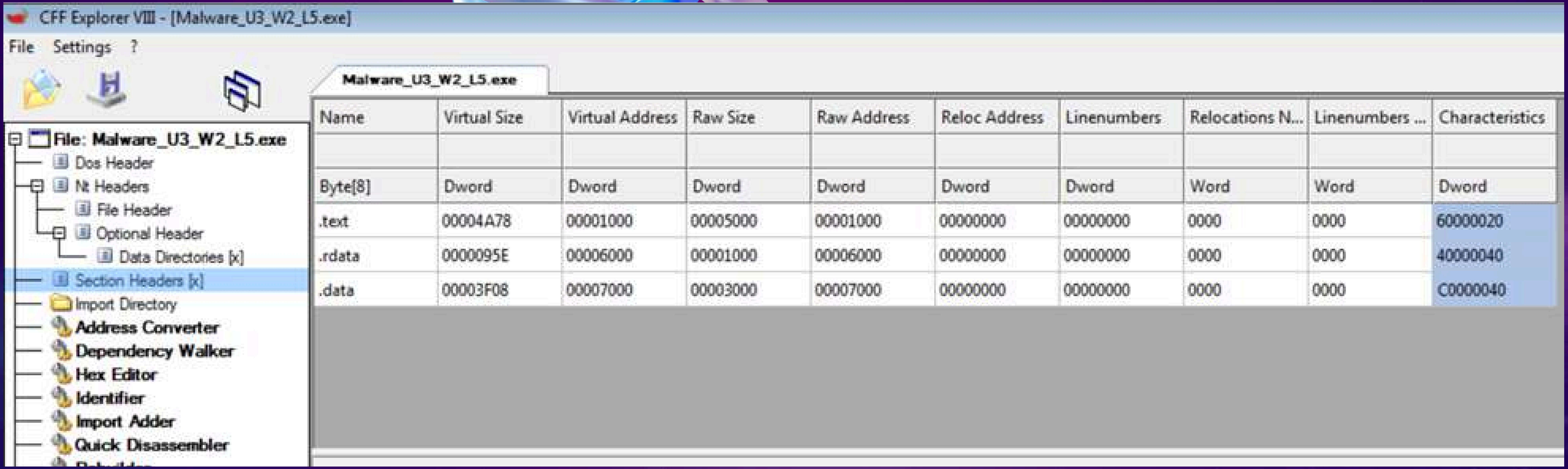
When analyzing malware, kernel32.dll is often monitored for function calls that may indicate suspicious behavior, such as creating new processes, manipulating system files, abnormally allocating memory, or modifying the registry

WININET.DLL

The WININET.DLL library is a Windows Dynamic Link Library and provides functions for Internet access. It is an integral part of the Windows architecture for network communication and is used by many applications to perform network operations, such as downloading files, Web browsing, and managing Internet connections.

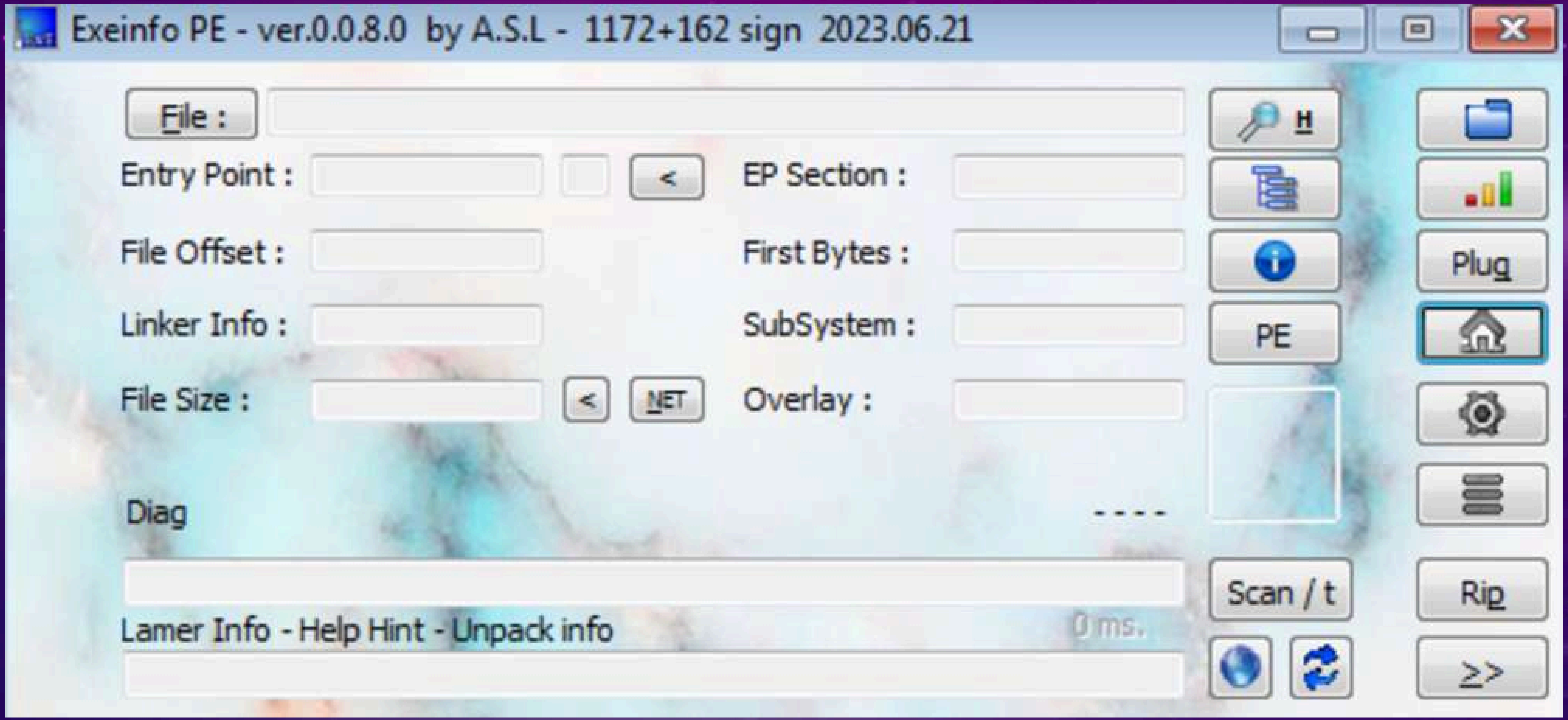
This library is often the target of malware that attempts to intercept or manipulate network traffic.

Malware that imports the WININET.DLL library is very dangerous because it could use functions that can download malicious payloads or files from remote servers, or it could send stolen data to a server controlled by an attacker. There are functions within this library that could intercept, steal, or manipulate session cookies resulting in compromised online accounts, identity theft, or access to sensitive information.

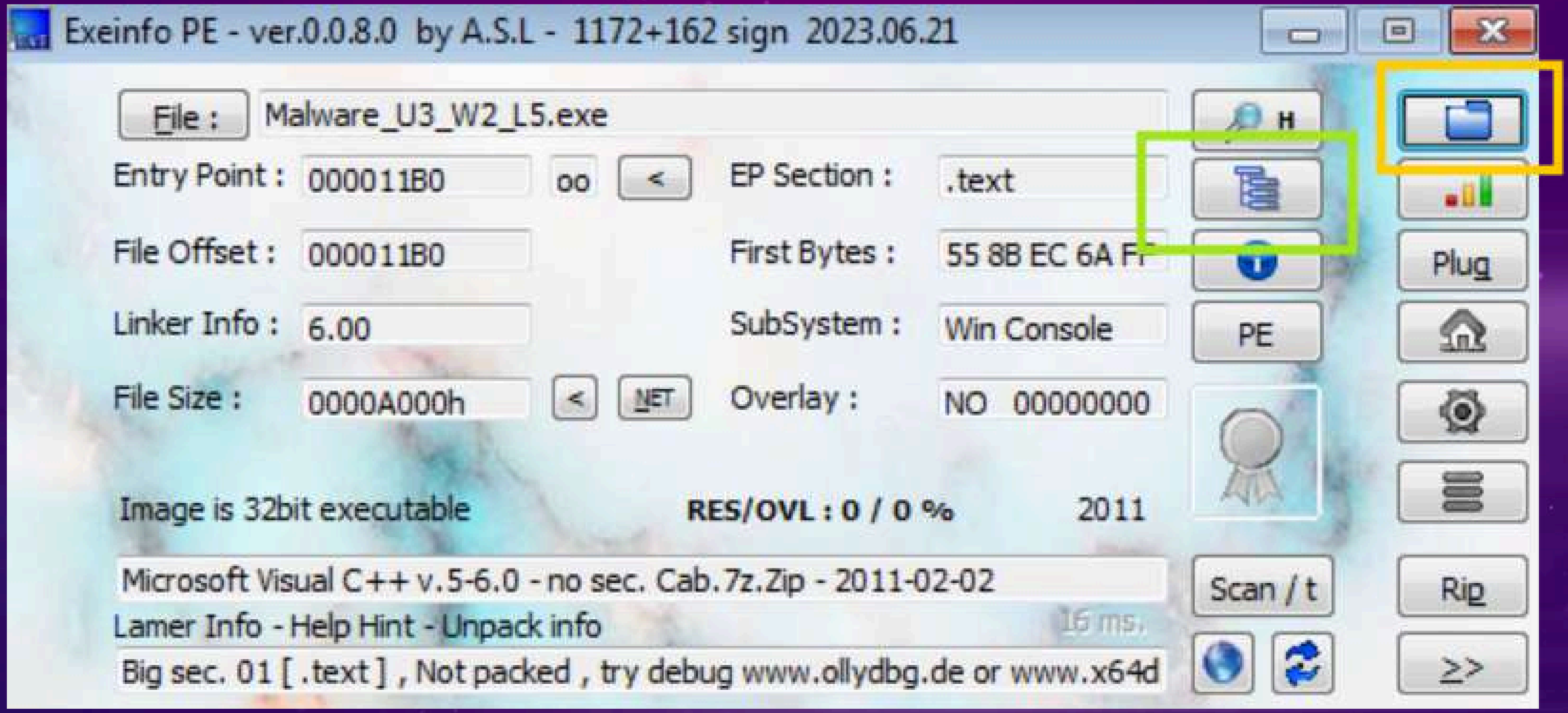


To go and see how many sections the malware is composed of we will simply click on the left column "Section Headers." Once done the Tool will return a table on which we find the information about the sections. This particular malware is composed of 3 main sections:

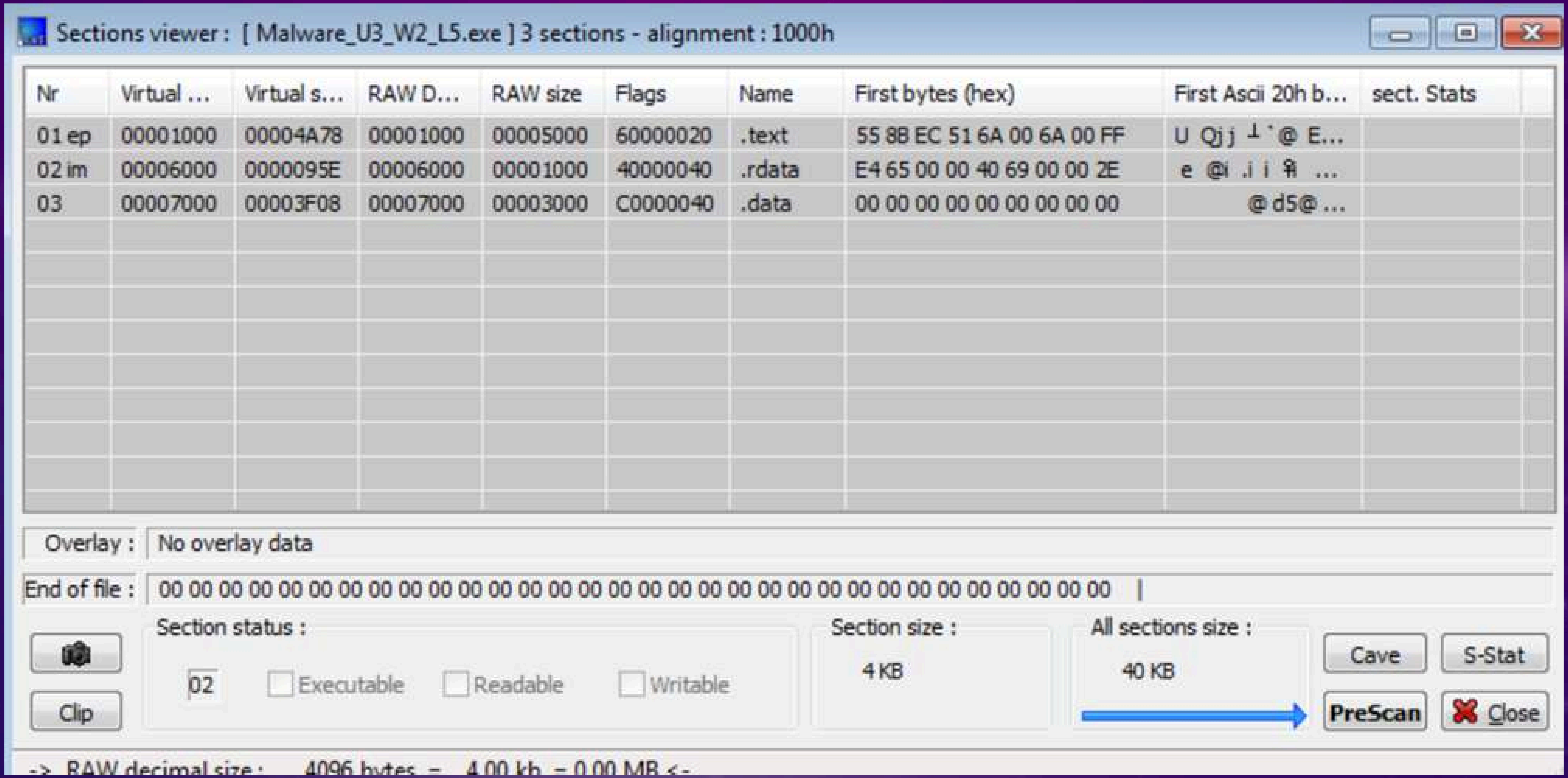
.text = contains the lines of code that the CPU will execute once the software is started;
.rdata = includes information about the libraries and functions imported and exported by the malware;
.data = contains the global data/variables of the executable program, which must be available from anywhere in the program.



In addition to CFF EXPLORER there are a myriad of Tools that are capable of analyzing malicious files, one of which is EXEINFO PE.

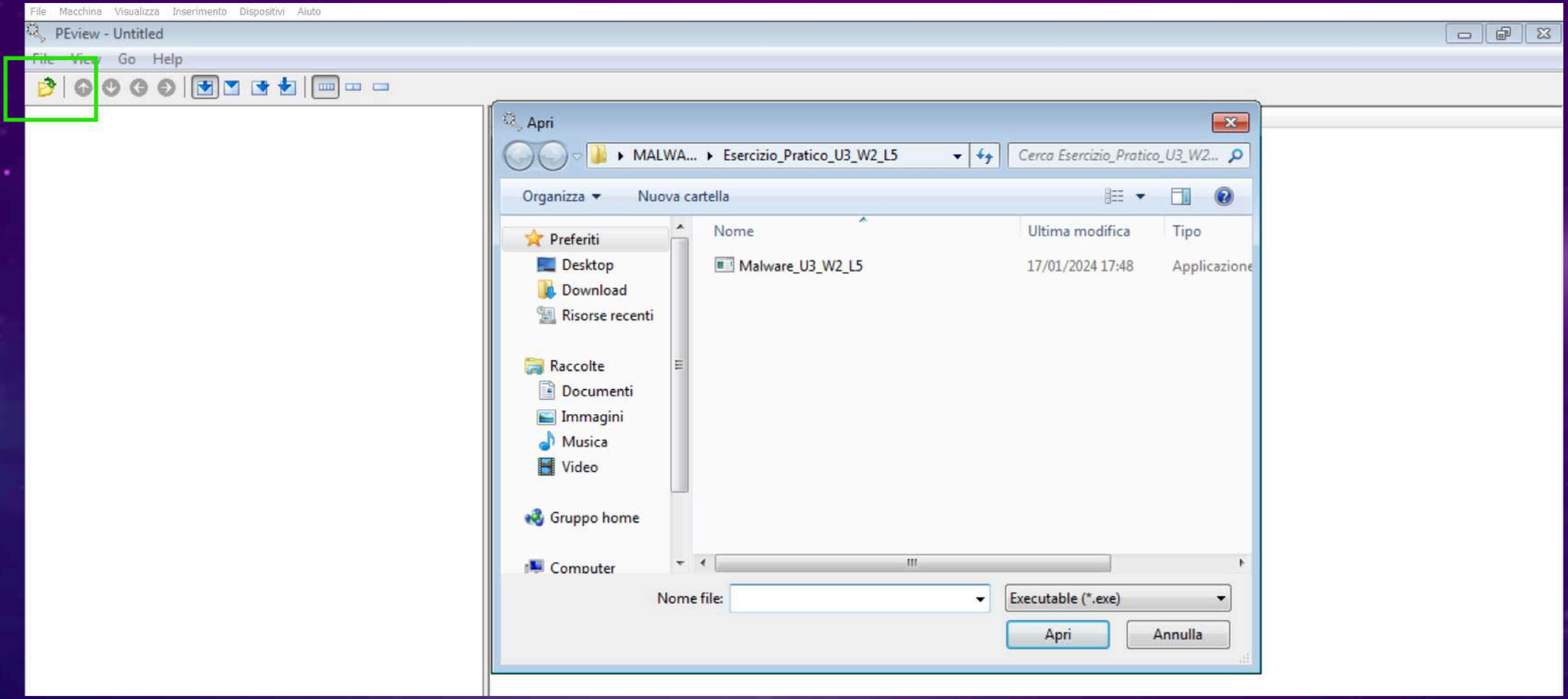


Once we open the Tool(installed by default on our windows7) we go to load the file to be analyzed, to do this we just click on the folder icon, browse the File system and select the executable. Once found, as can be seen in the figure, the Tool returns several pieces of information to us. To figure out how many sections the malware is composed of, we will simply click on the icon under the magnifying glass.



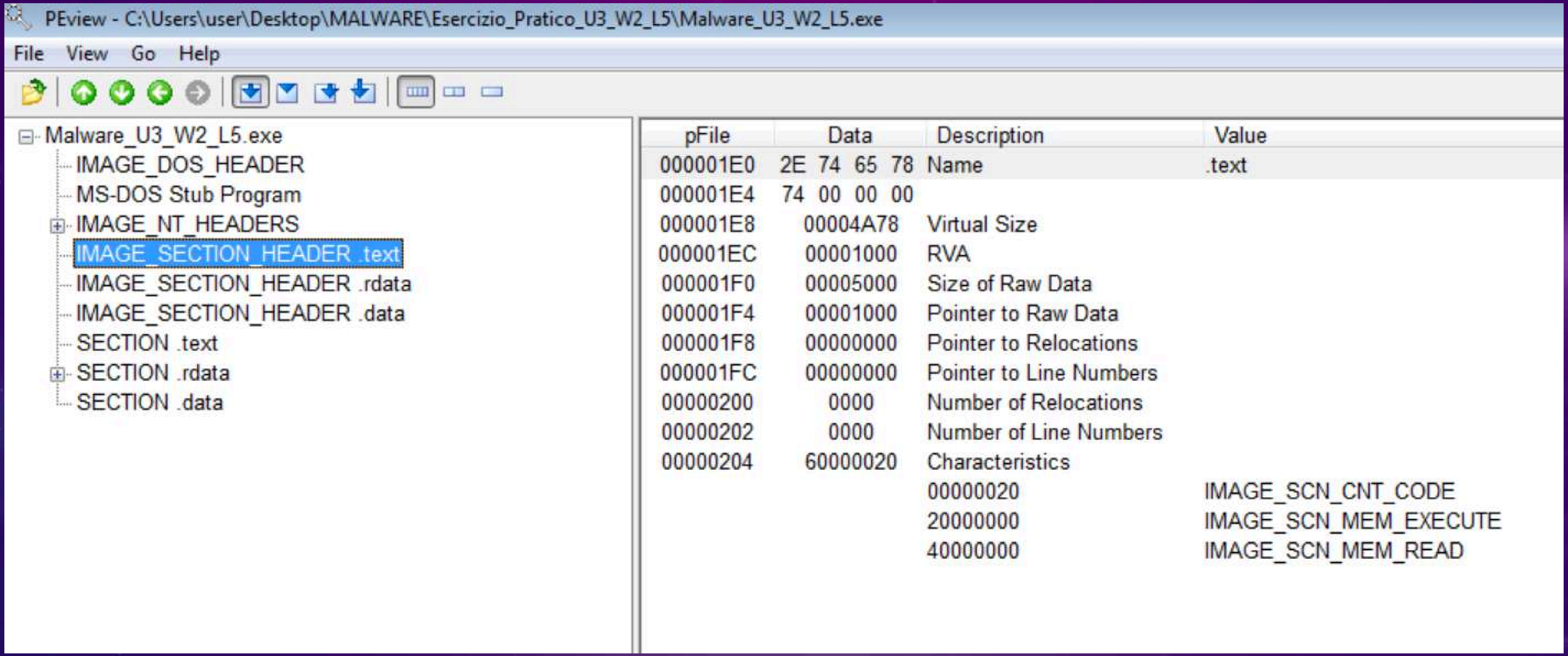
Nr	Virtual ...	Virtual s...	RAW D...	RAW size	Flags	Name	First bytes (hex)	First Ascii 20h b...	sect. Stats
01 ep	00001000	00004A78	00001000	00005000	60000020	.text	55 8B EC 51 6A 00 6A 00 FF	U Qj j ┴ `@ E...	
02 im	00006000	0000095E	00006000	00001000	40000040	.rdata	E4 65 00 00 40 69 00 00 2E	e @i .i i ...	
03	00007000	00003F08	00007000	00003000	C0000040	.data	00 00 00 00 00 00 00 00 00	@ d5@ ...	

Once this is done, the tool will return us a table with the sections of which the Malware consists.



Another tool you can use for basic static analysis is "PEview"

In the top left corner we find a folder icon, click it and go to choose the malware to analyze.



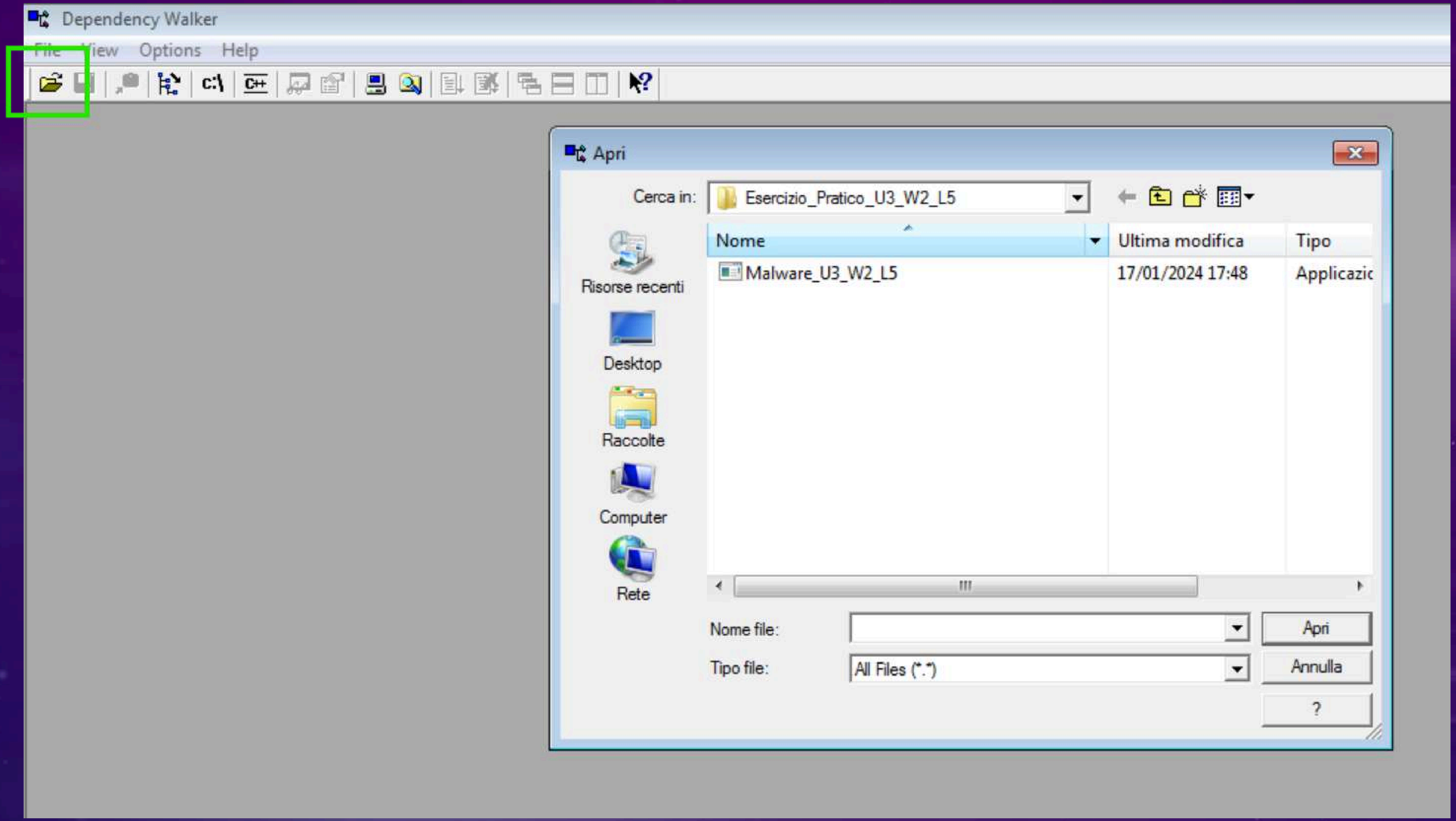
pFile	Data	Description	Value
000001E0	2E 74 65 78	Name	.text
000001E4	74 00 00 00		
000001E8	00004A78	Virtual Size	
000001EC	00001000	RVA	
000001F0	00005000	Size of Raw Data	
000001F4	00001000	Pointer to Raw Data	
000001F8	00000000	Pointer to Relocations	
000001FC	00000000	Pointer to Line Numbers	
00000200	0000	Number of Relocations	
00000202	0000	Number of Line Numbers	
00000204	60000020	Characteristics	
		00000020	IMAGE_SCN_CNT_CODE
		20000000	IMAGE_SCN_MEM_EXECUTE
		40000000	IMAGE_SCN_MEM_READ

This tool also returns us the malware sections with all their characteristics such as for example, the Size of Raw Data which stands for the size stored on the hard disk of the .text section data and we know it has a size of 00005000. The value is written in Hex(hexadecimal) format, in decimal format it becomes 20480 bytes.

"Virtual Size" is another field that we find in the .text section. This field specifies the size (in bytes) of the section in memory. Unlike the "Size of Raw Data", which represents the size of the section on disk, "Virtual Size" represents how much space the section will occupy when it is loaded into memory by the operating system. The virtual size of this malware section(doing the conversion from hexadecimal to decimal) turns out to be 19064 bytes.

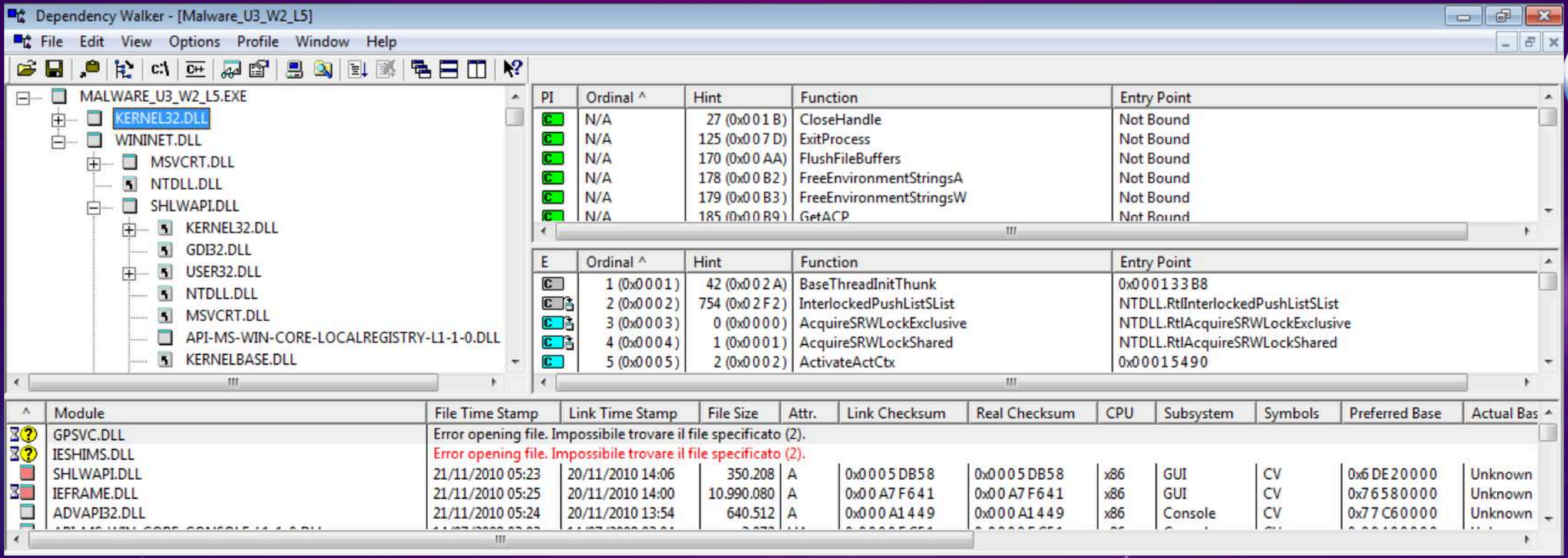
We note that in terms of size there is about 1000 bytes of difference, this could indicate that there may be data in the section that is compressed or that there is unused data.

Malware often uses compression or encryption techniques to hide the actual code. Significant differences(although this is not the case) between "Size of Raw Data" and "Virtual Size" may suggest the use of these "code obfuscation" techniques.



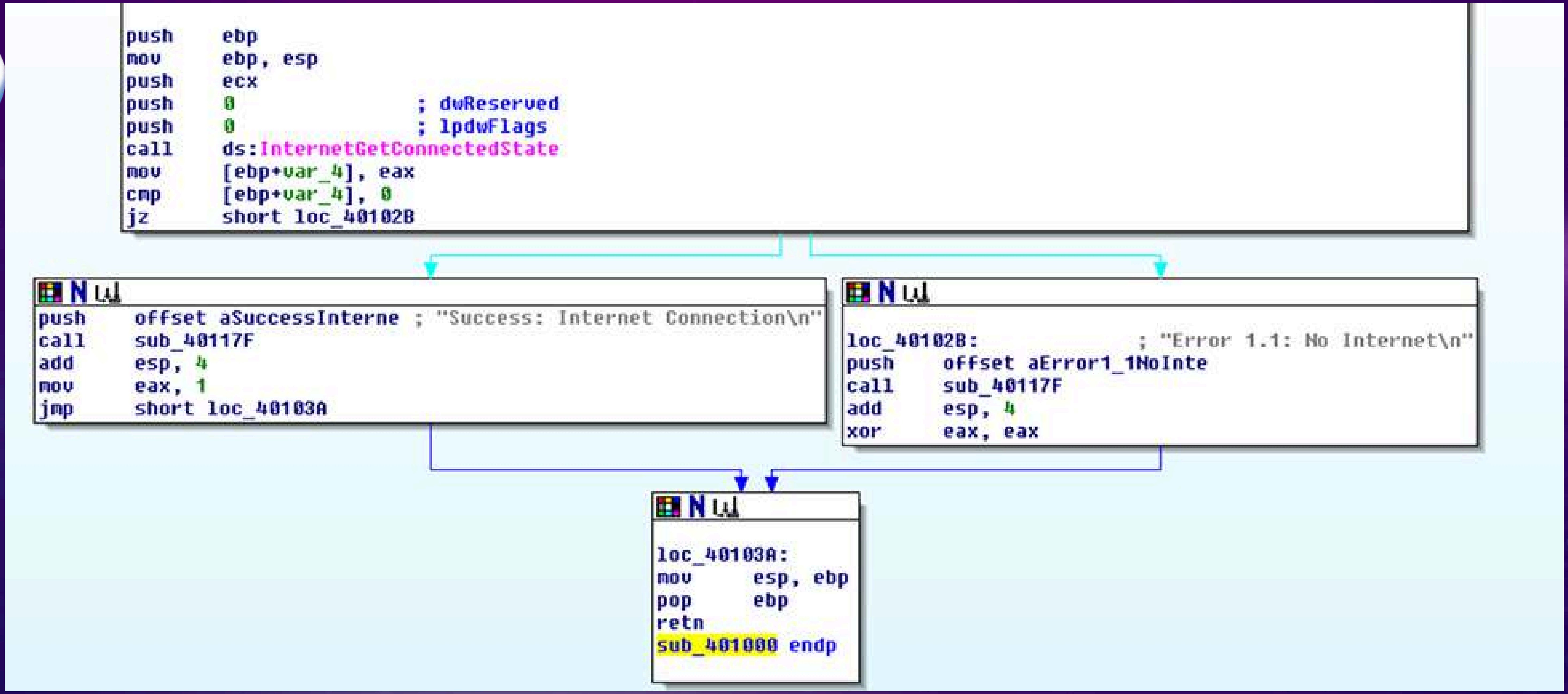
Another tool you can use for basic static analysis is "Dependency Walker "

In the upper left corner we find a folder icon, click it and go to choose the malware to be analyzed



The Dependency Walker tool is mainly used to analyze dependencies of executable, DLL, and other Windows files. Although Dependency Walker is not designed specifically for malware analysis, it can be useful for understanding the dependencies and imports/exports of a suspicious file. It displays a tree structure of the file's dependencies, showing all DLLs required by the analyzed file, Lists all modules (DLLs and other executable files) that the file loads, and shows the functions exported and imported by each module.

RESOLUTION SECOND PART



Answer the following questions:

3) Identify the known constructs (stack creation, possible loops).

4) Hypothesize the behavior of the feature.

5) BONUS make table with meaning of individual lines of code.

RETURN TO PAGE 2

FUNCTIONALITY BEHAVIOR

The code, using the InternetGetConnectedState function, checks to see if there is a connection to the Internet. If the connection is present the return value will be a number other than 0 and the success message "Succes: Internt Connection" will be printed. If the connection is not present, the return value will be a number equal to 0 and the error message "Error 1.1: No internet" will be printed.

TO IDENTIFY THE CONSTRUCTS WE CAN START BY DIVIDING THE CODE INTO SECTIONS SO THAT THEIR IDENTIFICATION IS MORE STRAIGHTFORWARD.

1) CREATING THE STACK

```
push    ebp
mov     ebp, esp
push    ecx
```

This first part of code is responsible for creating the stack.

push ebp = saves the contents of the ebp (Base Pointer) register on the stack. This is the first step in creating a new stack frame.

mov ebp, esp = copies the value of the esp (Stack Pointer) register to the ebp register. Esp will now point to the newly created stack frame and ebp will act as the stack base.

push ecx = will save the contents of the ecx register on the stack.

2) FUNCTION PARAMETERS AND FUNCTION CALL

```
push    0                       ; dwReserved
push    0                       ; lpdwFlags
call    ds:InternetGetConnectedState
```

In the part of the code that deals with handling the function parameters and the function call, the parameters needed for the call to the InternetGetConnectedState function are prepared and passed.

When calling a function in assembly, the parameters must be passed onto the stack before the call is made.

push 0 ;dwReserved = puts the value 0 on the stack, which is used as the first parameter of the function.

push 0 ;lpdwFlags = puts another 0 on the stack, which is used as the second parameter of the function.

call ds:InternetGetConnectedState = Calls the function to check the state of the Internet connection.

3) FUNCTION RETURN MANAGEMENT

```
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jz      short loc_40102B
```

Handling of the function return occurs immediately after the call to the InternetGetConnectedState function and relies on checking the return value to determine the next flow of the program.

mov [ebp+var_4], eax = stores the return value of the function contained in the Eax register in the local variable [ebp+var_4].

cmp [ebp+var_4], 0 = compares this value with 0.

jz short loc_40102B = handles a condition in fact if the comparison results in zero (if InternetGetConnectedState returned 0, indicating no connection), the program flow jumps to the loc_40102B label where the error portion of the code will be handled.

4) ERROR MANAGEMENT

```
loc_40102B:                        ; "Error 1.1: No Internet\n"
push    offset aError1_1NoInte
call    sub_40117F
add     esp, 4
xor     eax, eax
```

This part of the code handles the case where the return value of the function is equal to 0.
loc_40102B = the label where the error part of the code will be handled.
push offset aError1_1NoInte = "will push" an error string "Error 1.1: No internet" onto the stack .
call sub_40117F = calls an auxiliary function that will handle the display of the error message.
add esp, 4 = here we "clean" the stack, basically balancing the stack by removing the parameter passed by the auxiliary function.
xor eax, eax = Sets the eax register to 0, which will be the return value of the function indicating an error (no connection).

The "xor" instruction in assembly is generally used to reset a register to zero. xor is a bit to bit operation that performs an exclusive OR on two operands, if it is used with the same register for both operands each bit of the register will be compared with itself. the OR between 2 equal bits will always result in zero and the result will be to reset the register effectively and especially quickly compared to the "mov" instruction.

mov eax, 0 and xor eax, eax are the same thing only, as mentioned earlier, the xor instruction turns out to be faster and more compact in terms of machine code.

5) SUCCESS MANAGEMENT

```
push    offset aSuccessInterne ; "Success: Internet Connection\n"
call    sub_40117F
add     esp, 4
mov     eax, 1
jmp     short loc_40103A
```

This step of the code handles the case where the return value of the InternetGetConnectedState function is other than 0.

push offset aSuccessInterne = here will "push" the success message "Succes: Internt Connection" onto the stack.

call sub_40117F = call an auxiliary function to handle the message display.

add esp, 4 = "clears" the stack

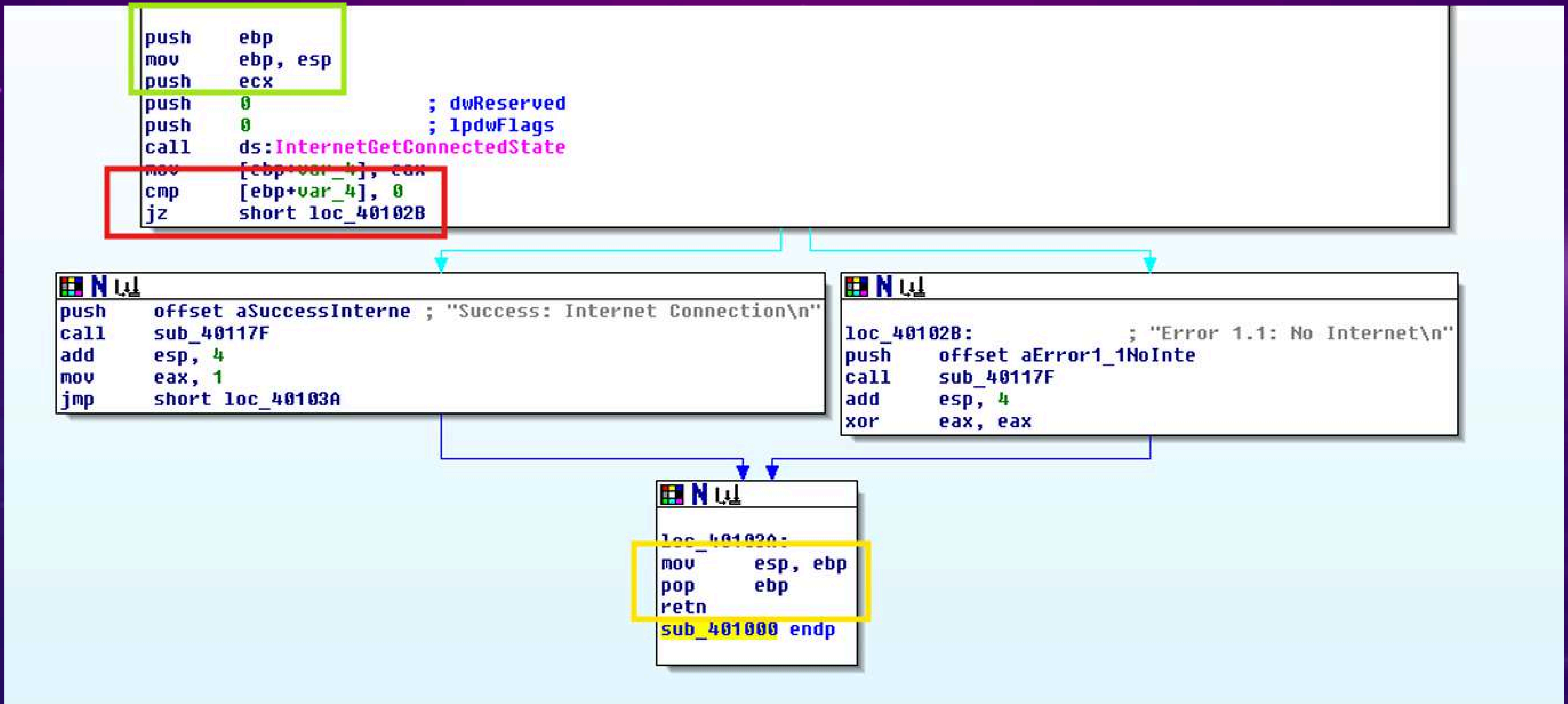
mov eax, 1 = sets the eax register to 1 indicating a successful return.

jmp short loc_40103A = handles the "jump" to label loc_40103 to proceed to the function output.

6) EXIT FROM THE FUNCTION

```
loc_40103A:
mov     esp, ebp
pop     ebp
retn
sub_401000 endp
```

This is the last section of the code and handles the output.
loc_40103A = the label where the part of the code regarding exit from the function will be handled;
mov esp, ebp = Restores the esp register to the original value saved in ebp;
pop ebp = Restores the ebp register to its original value saved at the beginning of the function;
retn = terminates the program.
These last lines of code are very important as they" clean up" the stacks and registers used to prevent corruptions or other problems.
The sub_401000 endp line simply indicates the end of the subroutine called sub_401000. It is a convention of many assemblers to mark the end of a procedure. It does not perform any operations at the binary code level; it serves only as a marker for the assembler.



Now that we have divided the code into sections we can see the use of 3 main constructs:

1) the first construct is about creating the stack;

2) the second is conditional construct of type "IF";

3) the third is about "cleaning" and removing the stack.

THANK YOU FOR YOUR ATTENTION

STEFANO CESARONI