W10-PS

BY MICHAEL ANDREOLI & OTMAN HMICH

MALWARE ANALYSIS

Trace:

Referring to the file Malware_U3_W2_L5 present inside the folder Esercizio_Pratico_U3_W2_L5 on the desktop of the virtual machine dedicated to malware analysis, answer the following questions:

1. Which libraries are imported by the executable file?
2. What are the sections that make up the executable file of the malware?

Referring to the figure in slide 3, answer the following questions:

1. Identify the known constructs (stack creation, possible loops, and other constructs) implemented in assembly.
2. Hypothesize the behavior of the implemented functionality.
3. BONUS: Create a table with the meaning of individual lines of code.

MALWARE ANALYSIS

Malware Analysis is the process of studying the behavior of malware. There are two types of analysis: static, which involves analyzing the code without running the malware, and dynamic, by executing it.

MALWARE ANALYSIS STATICA

Studying malware without running it, through disassembling the code and analyzing its static characteristics such as text strings, file sections, and imported libraries.

Basic Static Analysis

Basic static analysis focuses on a superficial examination of malware without delving too deeply into technical details. Key activities include:

- Identification of Metadata: Obtaining basic information about the file, such as the name, size, and hashes (MD5, SHA-1, SHA-256).
- String Extraction: Using tools like strings to extract and analyze the text strings present in the file. This can reveal URLs, domain names, error messages, and other useful information.
- File Header Examination: Checking the file format (e.g. PE, ELF) and obtaining basic information from the headers.
- Imported Libraries: Identifying the libraries and functions imported from the file by examining the Import Address Table (IAT).
-

Advanced Static Analysis

Advanced static analysis goes beyond superficial examination and requires more in-depth skills and sophisticated tools. Key activities include:

- Code Disassembly: Using tools like IDA Pro or Ghidra to disassemble the binary code into assembly language and analyze the instructions in detail.
- File Section Analysis: Examining the file sections (e.g. .text, .data, .rsrc) to identify executable code, data, and resources.
- Recognition of Constructs: Identifying common constructs such as loops, function calls, and stack management operations in the assembly code.
- Obfuscation Techniques Analysis: Detecting and attempting to deobfuscate the obfuscated or compressed code to understand the actual behavior of the malware.
- Comparison with Known Signatures: Using YARA rules and malware signature databases to identify known code patterns.

Undocumented Functions Analysis: Identifying the use of undocumented APIs or advanced evasion techniques used by the malware.

MALWARE ANALYSIS STATICA

Pro

- Security: There is no need to run the malware, so there is no risk of infecting the analysis system.
- Speed: It is often faster than dynamic analysis.
- Detection of hidden threats: It can identify malware components that may not be activated in a dynamic analysis.

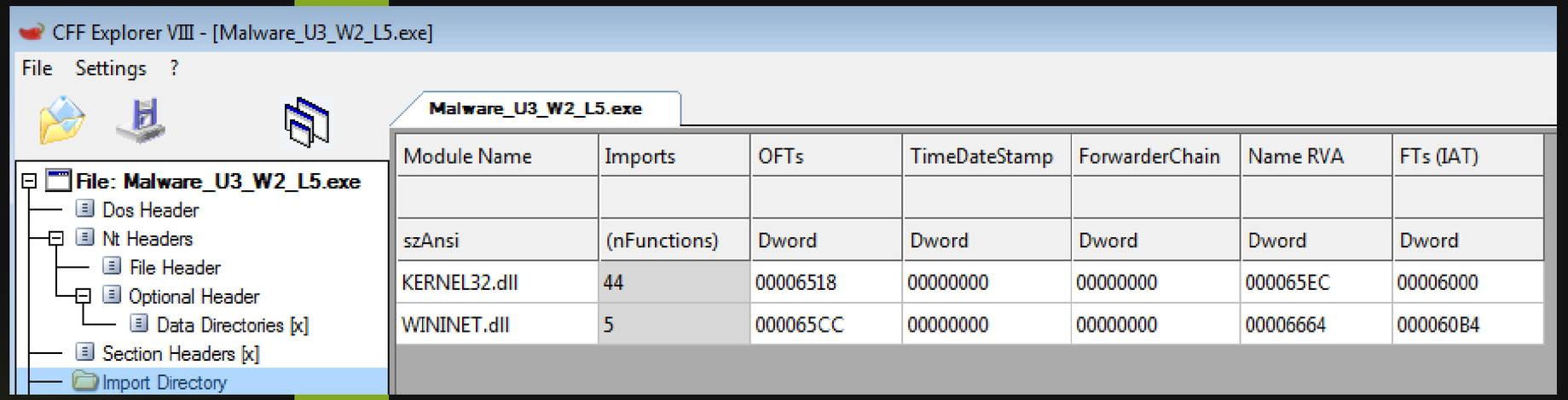
Cons

- Obfuscation: Malware can use obfuscation techniques to hide its code, making static analysis more difficult.
- Packers: The use of packers (software that compresses and encrypts the executable) can hide the true code of the malware.
- Lack of dynamic behavior: It does not provide information on the runtime behavior of the malware, which could be crucial for a complete understanding of the threat.

CFF EXPLORER

CFF Explorer is a freeware suite offering a comprehensive Portable Executable (PE) editor, CFF Explorer, alongside a process viewer. CFF Explorer itself provides in-depth analysis and modification capabilities for PE files, including support for 32-bit and 64-bit architectures. Key functionalities include: Detailed PE Structure Exploration: Examine headers, sections, imports, exports, and resources of PE files. Editing and Customization: Modify metadata fields and flags, enabling customization of program behavior. Advanced Functionality: Tools like a disassembler, hex editor, import adder, and scripting language empower advanced editing and analysis. .NET Support: CFF Explorer is the first PE editor with full support for editing .NET internal structures and resources.

A DLL (Dynamic Link Library) is a toolbox for computer programs. It contains tools (functions and data) that many programs can use at once, instead of each program having its own copy. This saves space and memory. Many programs can use this same tool from the DLL, instead of each writing their own version.



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	44	00006518	00000000	00000000	000065EC	00006000
WININET.dll	5	000065CC	00000000	00000000	00006664	000060B4

In Import Directory we can see there are two libraries: KERNEL32 and WININET.

- KERNEL32.DLL: A fundamental Windows library. It provides basic functions, including memory management, process management, and thread execution. It is imported by many operating system executable files.
- WININET.dll: Provides functions for Internet access, such as sending and receiving HTTP requests and downloading files. It is imported by executable files that need to connect to the Internet.

CFF EXPLORER

As we can see in the in the lower part there are the functions of the corresponding libraries

Malware_U3_W2_L5.exe

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
000065EC	N/A	000064DC	000064E0	000064E4	000064E8	000064EC
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
000065E4	000065E4	0296	Sleep
00006940	00006940	027C	SetStdHandle
0000692E	0000692E	0156	GetStringTypeW
0000691C	0000691C	0153	GetStringTypeA

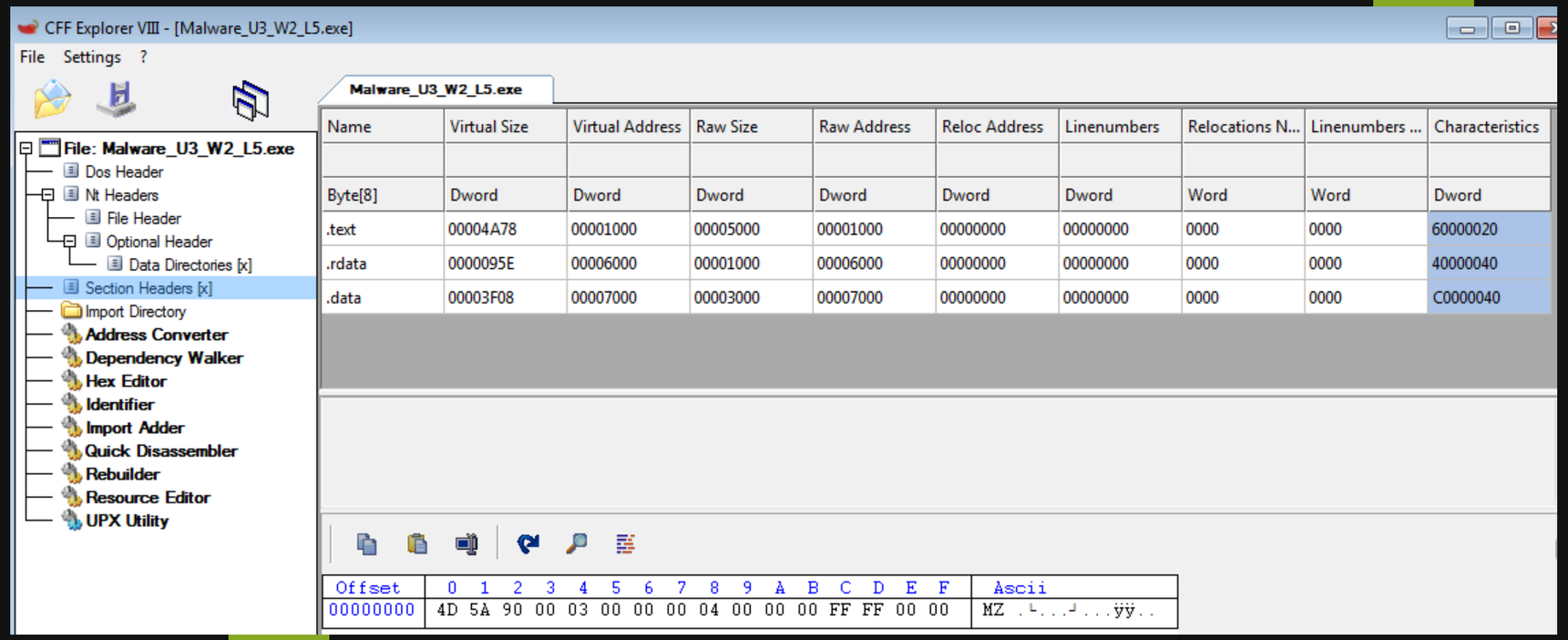
Malware_U3_W2_L5.exe

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

CFF EXPLORER

Now we can see the parts of the malware, .text, .rdata, .data;
.text This is the part of the code that will be executed by the CPU.
.rdata This is the section that contains information about the imported libraries.
.data Contains the global variables that the code will act upon.



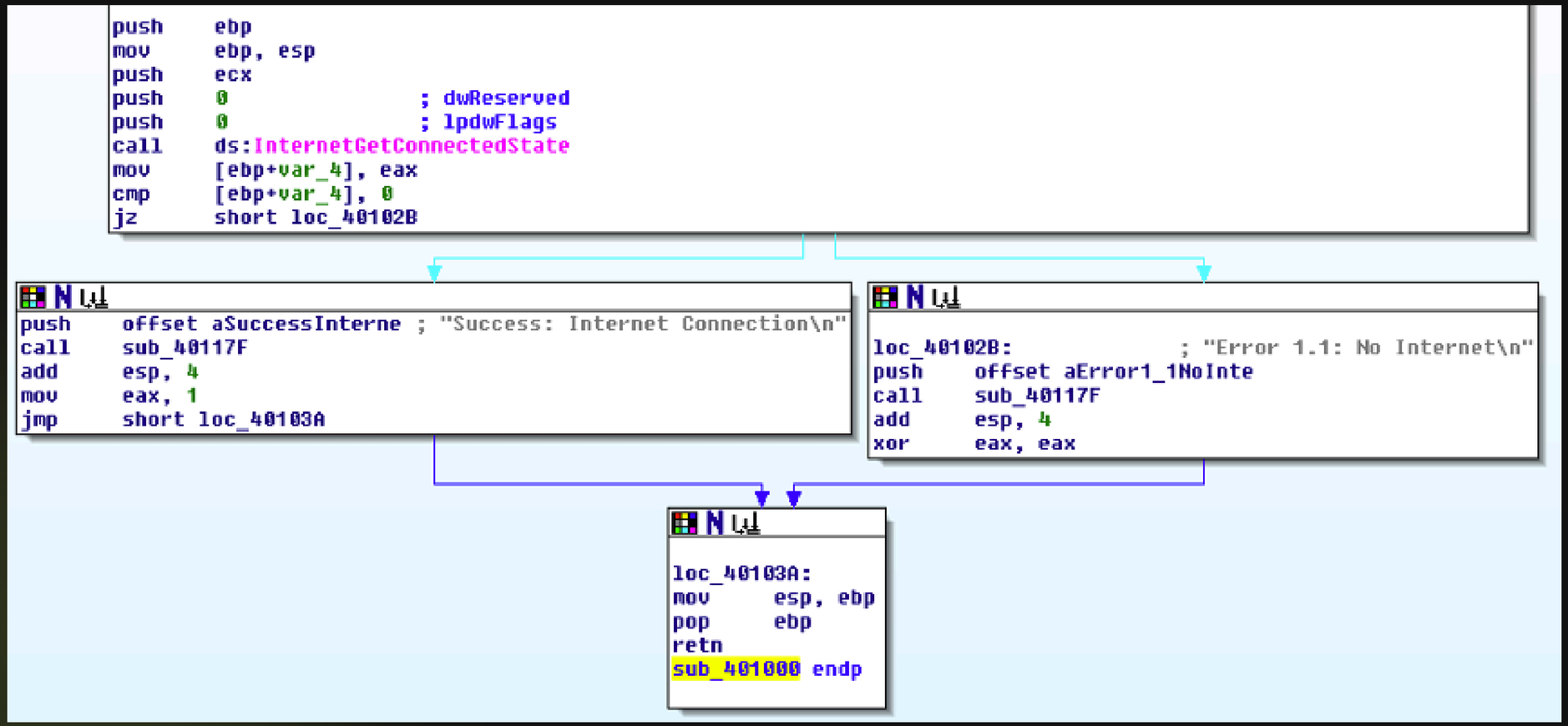
Section Headers

Each section header describes a section within the PE file.

- Name: Name of the section (e.g., .text, .data).
- VirtualSize: Total size of the section when loaded into memory.
- VirtualAddress: Address of the section when loaded into memory.
- SizeOfRawData: Size of the section's data on disk.
- PointerToRawData: File pointer to the section's data.
- PointerToRelocations: File pointer to the relocation entries.
- PointerToLinenumbers: File pointer to the line number entries.
- NumberOfRelocations: Number of relocation entries.
- NumberOfLinenumbers: Number of line number entries.
- Characteristics: Flags indicating the characteristics of the section.

ASSEMBLY

Assembly language, is a low-level programming language closely related to a computer's architecture. Low-Level: It is called "low-level" because it operates directly on the processor's instructions, using a language very close to machine code (which the CPU can execute directly). Simple Instructions: Each instruction in assembly represents a very simple and specific operation, such as moving data between registers, performing calculations, or controlling the program's execution flow. Registers and Memory: Operations work with registers (small memory units within the CPU) and memory addresses. For example, you can load a value into a register, perform a calculation with it, and then save the result in another register or in memory. Hardware-Linked: Each type of processor (x86, ARM, etc.) has its own set of assembly instructions, which means that assembly code written for one type of processor will not work on another type without modifications. Main Use: It is mainly used when precise control over hardware is needed, such as in operating systems, drivers, firmware, and some high-performance applications.




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

STACK CREATION

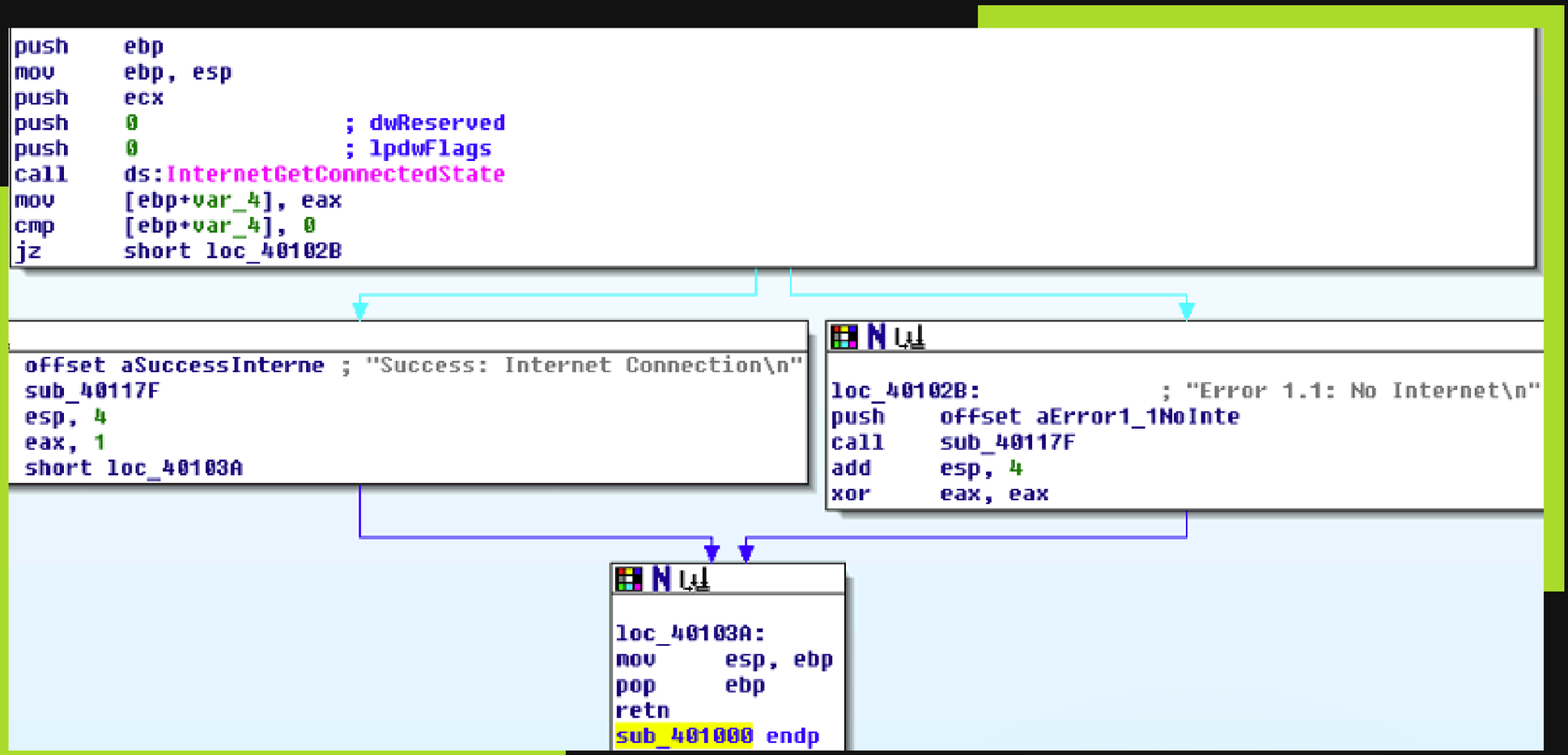
```
cmp     [ebp+var_4], 0
jz      short loc_40102B
```

iF CONDITION

```
mov     esp, ebp
pop     ebp
```

STACK CLEANING

ASSEMBLY



After analyzing the code, we have concluded that it aims to check whether an internet connection is established or not, using an if statement to determine the connection status.

With the line of code cmp [ebp+var_4], 0, we perform the check where, if the value is 0, the connection is not established, and it jumps to the memory location 40102B, informing us of the failed connection.

If the result is not 0, the connection is established, and the message “Success: Internet Connection” is displayed.

ASSEMBLY

```
push    ebp
mov     ebp, esp
push    ecx
push    0                   ; dwReserved
push    0                   ; lpdwFlags
call    ds:InternetGetConnectedState
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jz      short loc_40102B
```

Copy the content of ***ESP*** (which points to the top of the stack) into EBP.
Inserts ***ECX*** into the stack
Inserts the value 0 into the stack
IInserts the value 0 into the stack (***lpdwFlags*** could be a signal flag)
Call the function ***InternetGetConnectedState***
Inserts the value of ***EAX*** inside ***VAR_4***
Compare 0 with the value inside ***VAR_4***
Jump conditioned by the previous comparison (if 0),
we skip to the specified memory location

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

Push the established connection string into the stack
Call the function at the memory address specified
Adding 4 to ***ESP*** registry
Change the ***EAX*** registry value with 1
Jump to the memory address

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

Push the ***printf*** function telling theere is no connection
Call the function at the memory address specified
Add 4 to the ***ESP*** registry
Zero the ***EAX*** registry

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

Copy the value of ***EBP*** in ***ESP***
Remove ***EBP*** pointer
Close the program

ASSEMBLY

```
push    ebp
mov     ebp, esp
push    ecx
push    0                   ; dwReserved
push    0                   ; lpdwFlags
call    ds:InternetGetConnectedState
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jz      short loc_40102B
```

Si punta alla base dello stack
Si copia il contenuto di ESP (che punta alla cima dello stack) in EBP
Inserts ECX into the stack
Inserts the value 0 into the stack
IInserts the value 0 into the stack (lpdwFlags could be a signal flag)
Call the function InternetGetConnectedState
Inserts the value of EAX inside VAR_4
Compare 0 with the value inside VAR_4
Jump conditioned by the previous comparison (if 0), we skip to the specified memory location

Push the established connection string into the stack
Call the function at the memory address specified
Adding 4 to ESP registry
Change the EAX registry value with 1
Jump to the memory address

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

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

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