CySicuro?
security

2K24 PRACTICE EXERCISE S10/L5

Prepared By :

Matteo Tedesco

Federico Biggi

Aldrovandi Max

Table of Contents

Track

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. What libraries are imported from the executable file?

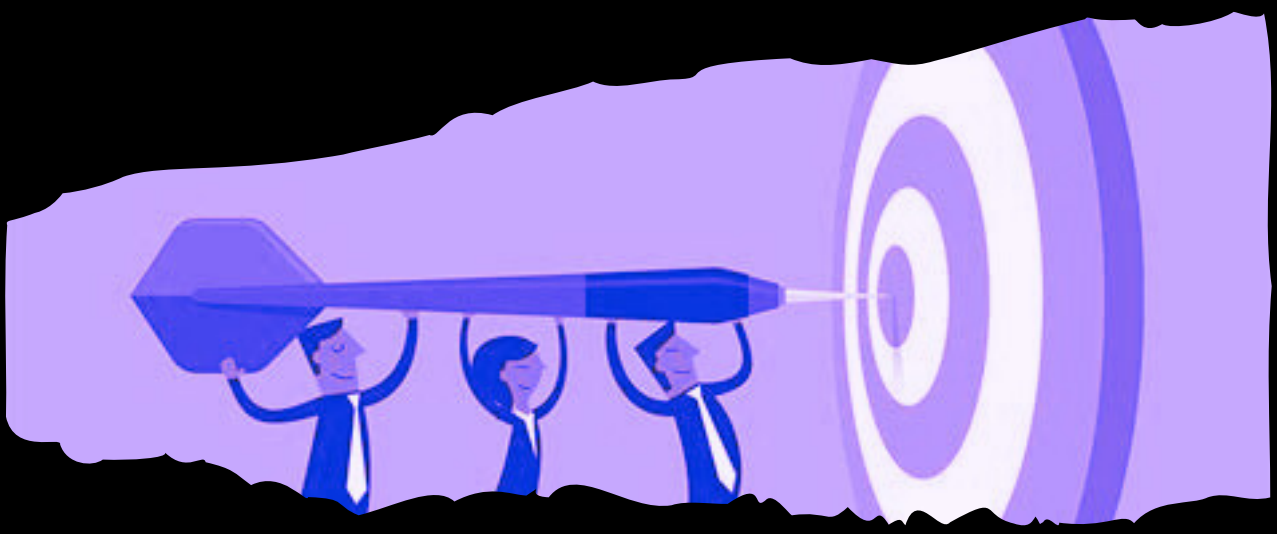
2. What sections does the malware executable file consist of?

With reference to the figure in slide 3, answer the following questions:

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

4. Hypothesize the behavior of the implemented feature.

5. BONUS -> Make table with meaning of individual lines of assembly code.



Lab Setup

Prerequisites:

When analyzing a malware, the setup of a virtual lab is extremely important.

But first thing first: what is a **malware?**

What is a MALWARE?

Malware (short for malicious software) is any software intentionally designed to cause damage to a computer, server, client, or computer network.
Malware can take many forms, including viruses, worms, Trojans, ransomware, spyware, adware, and more. Its primary purpose is to infiltrate, damage, or disable computers and computer systems, often by exploiting vulnerabilities in software.
Malware can disrupt operations, steal sensitive information, gain unauthorized access to systems, and cause significant financial and reputational damage. The spread of malware is typically done through phishing emails, malicious websites, or by exploiting security flaws in software.

Malware analysis is the process of **studying a malware sample to understand its behavior, functionality, and potential impact.** The goal is to dissect the malware to determine how it operates, how it spreads, what it does when executed, and how to detect, neutralize, or mitigate its effects.

There are two main types of Malware analysis: **Static** and **Dynamic.**

1. **Static Analysis:**
   - Involves examining the malware without executing it.
   - Techniques include inspecting the binary code, decompiling, disassembling, and using various tools to extract strings and other resources.
   - Provides insights into the malware's structure and components.

2. **Dynamic Analysis:**
   - Involves executing the malware in a controlled environment (such as a sandbox or virtual machine) to observe its behavior in real-time.
   - Techniques include monitoring file system changes, network activity, registry modifications, and system calls.
   - Reveals the malware's runtime behavior and interactions with the system.

Setup the lab:

Now we need to **setup the lab**.

Among the best practices for configuring a secure environment, we find:

NETWORK CONFIGURATION

The test environment should not have direct access to the internet and preferably not to other machines on the network either. The ideal configuration is:

1. **Disable Network Interfaces During Static Analysis**:
   - This prevents any unwanted network activity while performing static analysis on the malware, ensuring that the environment remains isolated.

2. Enable an Internal Network Interface for Dynamic Analysis:
- In VirtualBox, this is referred to as an "internal network." This setting is necessary to monitor the traffic generated by the malware during dynamic analysis. The internal network allows the VM to communicate with other VMs in a controlled and isolated environment, which is essential for observing how the malware behaves and interacts with network services without risking exposure to external systems

USB DEVICE

When a USB device is connected to the physical machine, it can also be recognized by the test environment. To avoid this behavior, it is good practice to either not enable or to disable the USB controller in the virtual machine. This precaution is important because malware could use the USB device to spread to your physical machine.

By disabling the USB controller, you prevent any potential interaction between the test environment and USB devices, thereby enhancing the security of your analysis setup.

SHARED FOLDERS

The same considerations apply to shared folders between your physical machine and the virtual lab. These shared folders could be used by malware to propagate outside the lab, potentially causing damage to your physical machine and other devices on your home network. Therefore, it is recommended not to share folders between the host and guest environments.

By avoiding shared folders, you minimize the risk of malware escaping the isolated test environment and compromising your primary system.

CREATE SNAPSHOTS

When analyzing malware, it often happens that the test environment gets damaged or permanently compromised. A good practice is to create snapshots of the virtual machine in its initial state before starting any analysis. This way, you can easily restore the environment if needed.

By creating snapshots, you ensure that you can quickly revert to a clean state, allowing for a safe and efficient continuation of your analysis work.

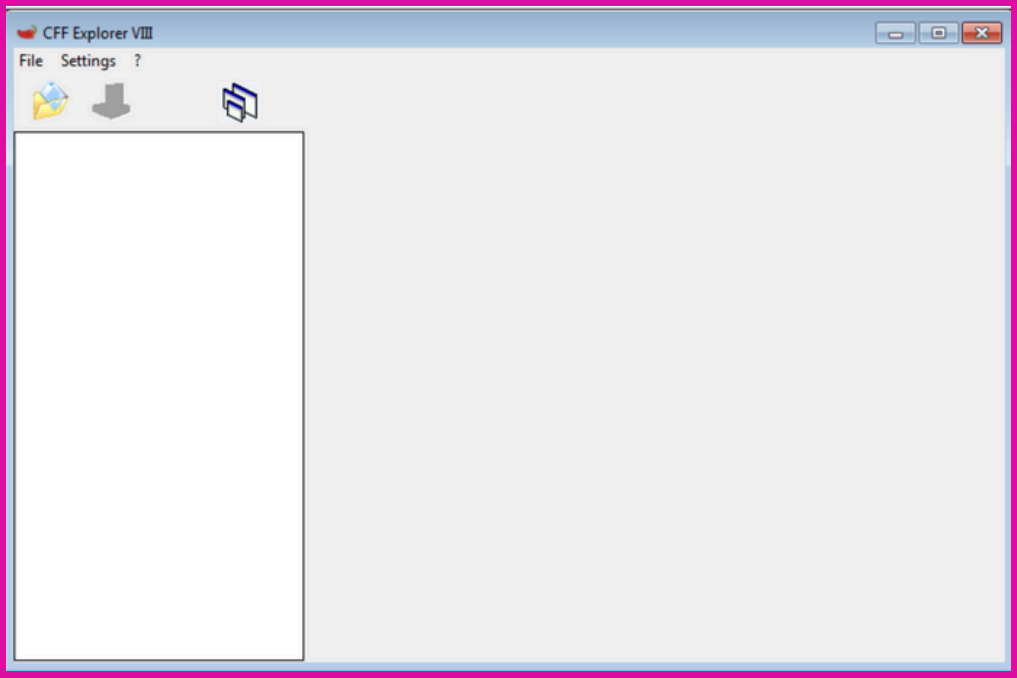
Analysis with CFF Explorer

Now that our lab is up and running, we will analyze the first part of the exercise.
The file “**Malware_U3_W2_L5**” is an executable file in our Windows 7 VM. Windows uses the **Portable Executable (PE)** format for most executable files. The PE format contains information necessary for the operating system to understand how to manage the file's code, such as **libraries**.
Modules or libraries contain a set of **functions**. When a program needs a function, it **"calls"** a library where the required function is defined.

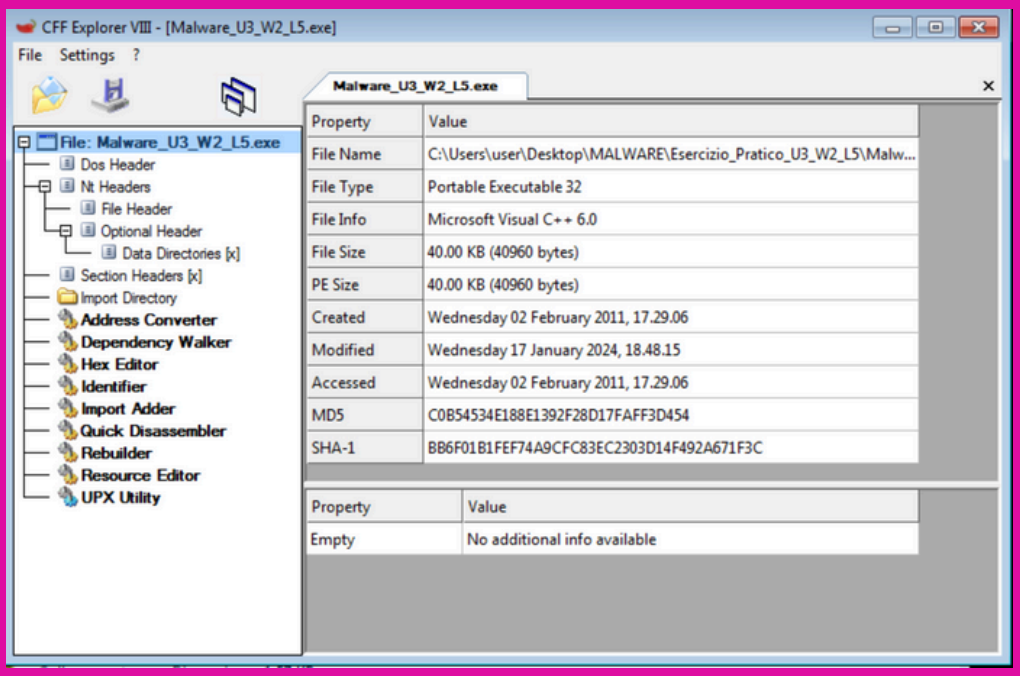
We are going to use the program **CFF Explorer**, a software tool primarily used for examining and modifying the internals of executable files in order to examine the **sections** and the **libraries** imported by the malware.

Analysis:

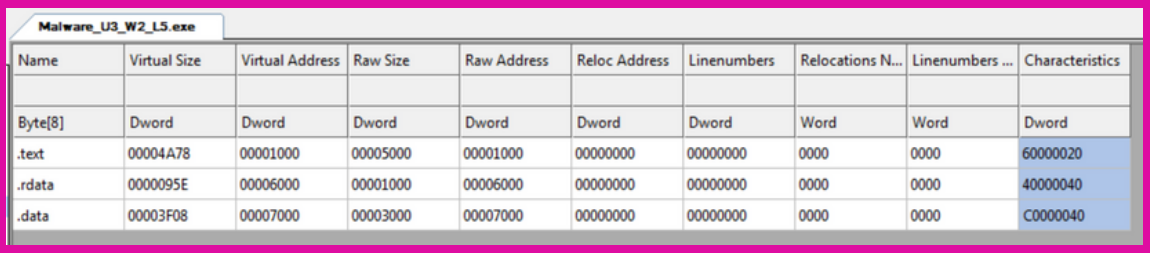
First, we’ll open the tool CFF Explorer inside our VM.



After that, we'll open the malware inside the program.



As we can see, on the left, we have the various headers categories. What we need to do is to go into the **Section Headers** one, in which we'll find what we are looking for.



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00004A78	00001000	00005000	00001000	00000000	00000000	0000	0000	60000020
.rdata	0000095E	00006000	00001000	00006000	00000000	00000000	0000	0000	40000040
.data	00003F08	00007000	00003000	00007000	00000000	00000000	0000	0000	C0000040

We can see that the sections of the malware are 3:

- **.text**: Contains executable code.
- **.rdata**: Contains read-only data, such as constants and strings.
- **.data**: Contains initialized global and static variables.

In this case, the sections are not obfuscated so we can clearly see what sections from the malware. **Obfuscating the sections** is a technique used for confusing and slowing down the analysis process.

Now, we'll move into the **imported directory.**

Malware_U3_W2_L5.exe

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

The imported libraries are two:

- **Kernel32.dll:** a core Dynamic Link Library (DLL) file in the Windows operating system, providing essential functions and services for user-mode applications and system components.
- **Wininet.dll**: Another core DLL in Windows that provides functions related to Internet protocols and services.

Importing these two libraries, typically indicates that the malware intends to leverage **system-level and internet-related functionalities** provided by these libraries to achieve its malicious objectives, such as file registry creation, file system manipulation, network communication and DNS manipulation, just to name a few.

Assembly - Code Analysis

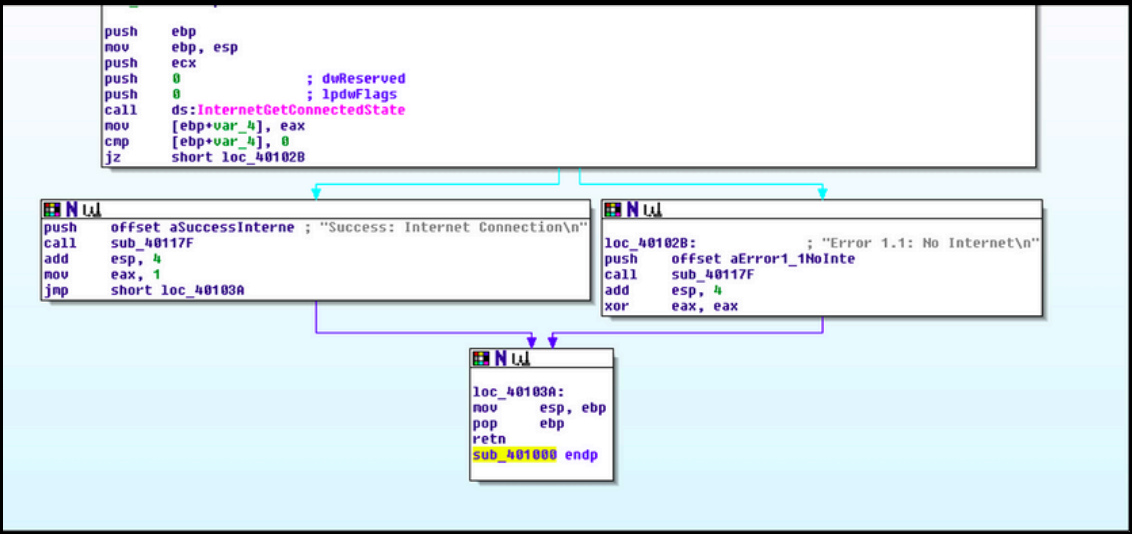
Assembly Language:

Assembly language is a low-level programming language that is closely related to machine code, which is the set of instructions executed directly by a computer's CPU.

Unlike high-level programming languages that are more abstract and closer to human languages, assembly language provides a more direct way to control hardware by using mnemonic codes and labels to represent machine-level instructions and data locations.

Assembly is used in malware analysis to reverse-enginner the code of the malware that we need to understand. By reverse-engineering it, we can see how the malware works in detail.

Code Analysis:



We need to identify the constructs, hypothesize the functionality and create a table to explain each line of code.

Analysis of the code:

1. Identifying the Constructs

Stack Creation: The code starts by setting up the stack frame with the instructions:

- **Function Call:** The function `**InternetGetConnectedState**` is called to check the internet connection status.
- **Conditional Jump**: The code uses `cmp` and `jz` (jump if zero) to decide the flow based on the internet connection status.
- **Message Display:** Depending on the internet connection status, it pushes different messages onto the stack and calls another function to handle the display.
- **Stack Frame Destruction**: The code ends by restoring the stack frame and returning control to the caller.

2. Functionality Hypothesis

The assembly code appears to check whether there is an internet connection and then prints an appropriate message. Here's a step-by-step breakdown:

1. **Set Up Stack Frame**: Saves the current base pointer and sets up a new stack frame.

2. **Check Internet Connection:** Calls the **`InternetGetConnectedState`** function.

3. **Branch Based on Result:** If the result is equal to 0, there is no internet connection and it jumps to the section that handles the error message.

4. **Print Success Message**: If the result is equal to 1, there is internet connection available, it pushes the success message and calls a subroutine to print it.

5. **Print Error Message**: If there is no internet connection, it pushes the error message and calls the same subroutine to print it.

6. **Clean Up and Return**: Restores the stack frame and return.

3. Line-by-Line Explanation (Optional Table)

Line of Code	Explanation
`push ebp`	Save the current base pointer.
`mov ebp, esp`	Set the base pointer to the current stack pointer.
`push ecx`	Save the `ecx` register on the stack.
`push 0`	Push the argument `0` for `dwReserved` onto the stack.
`push 0`	Push the argument `0` for `lpdwFlags` onto the stack.
`call ds:InternetGetConnectedState`	Call the function to check the internet connection status.
`mov [ebp+var_4], eax`	Store the result of the function call in a local variable.
`cmp [ebp+var_4], 0`	Compare the result to `0` (no internet connection).
`jz short loc_A01028`	Jump to `loc_A01028` if the result is `0`.
`push offset aSuccessInterne`	Push the address of the success message onto the stack.
`call sub_A0117F`	Call a subroutine to print the message.

Line of Code	Explanation
`add esp, 4`	Clean up the stack by removing the message pointer.
`mov eax, 1`	Set the return value to `1` (indicating success).
`jmp short loc_A0103A`	Jump to the end of the function.
`loc_A01028:`	Label for the error handling code.
`push offset aError1_1NoInte`	Push the address of the error message onto the stack.
`call sub_A0117F`	Call a subroutine to print the message.
`add esp, 4`	Clean up the stack by removing the message pointer.
`xor eax, eax`	Set the return value to `0` (indicating failure).
`loc_A0103A:`	Label for the end of the function.
`mov esp, ebp`	Restore the original stack pointer.
`pop ebp`	Restore the original base pointer.
`retn`	Return from the function.

Summary

The assembly code checks for an internet connection and prints a success or error message based on the result. The use of stack operations, function calls, and conditional jumps are typical constructs in this type of functionality.