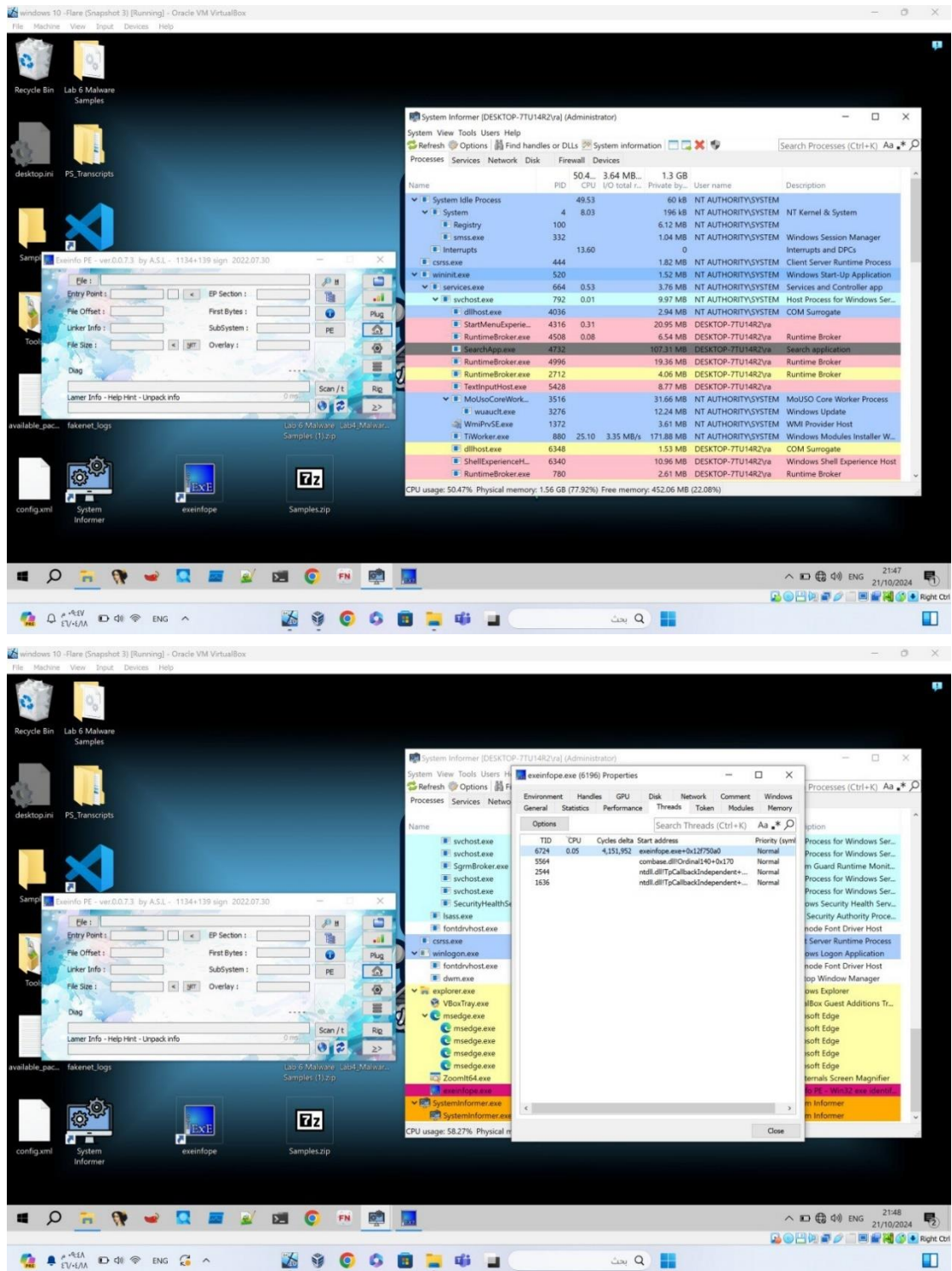


CCCY432 – Reverse Engineering and Malware Analysis
Lab 6 – Unpacking Process Injection Code Using Debugger
Raghad Lafe -2111941

Task 1: Injected a DLL in a running process using Process Hacker:

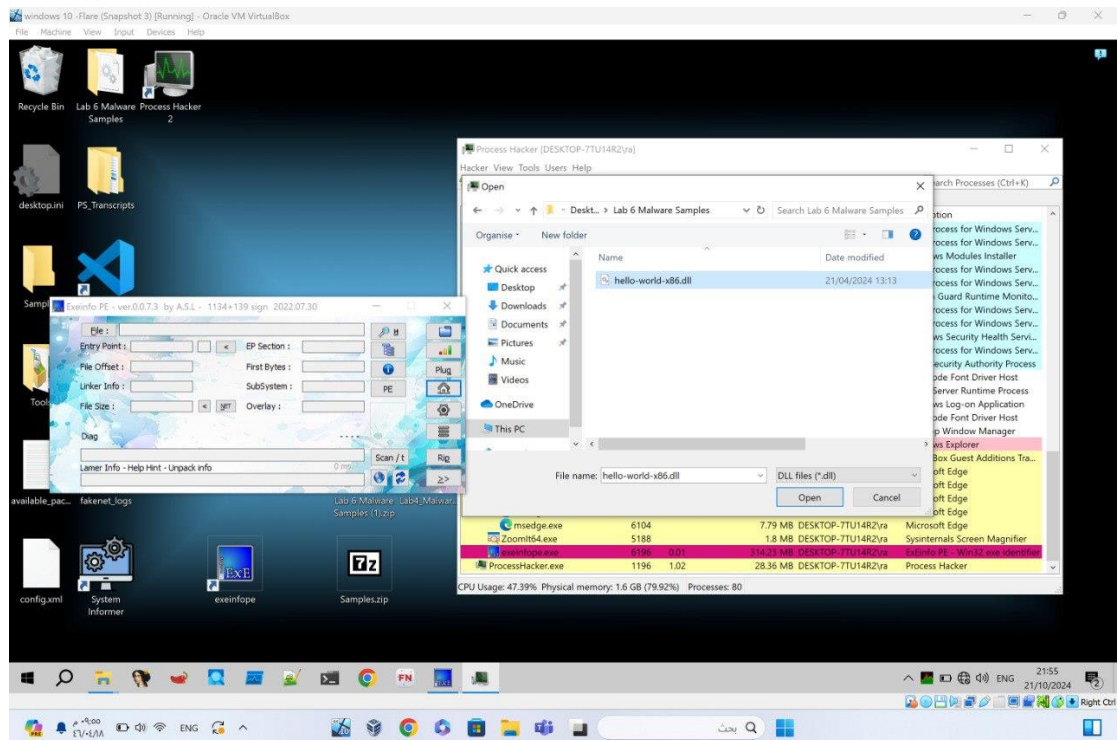
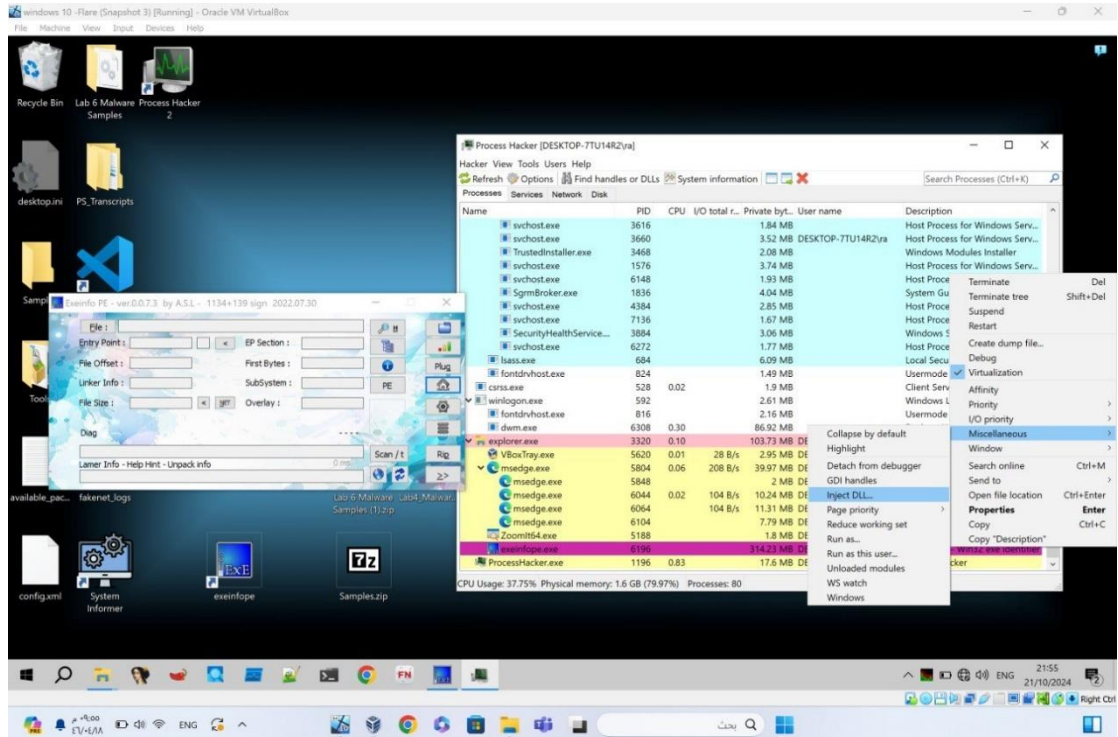
1- Screenshot of exeinfo Threads Before Injection:

- Before the DLL was injected, I took a screenshot of the running threads in the exeinfo process.



2- Inject a DLL into a Running Process:

- I used Process Hacker to inject a hello-world-x86.dll into an active exeinfo process.

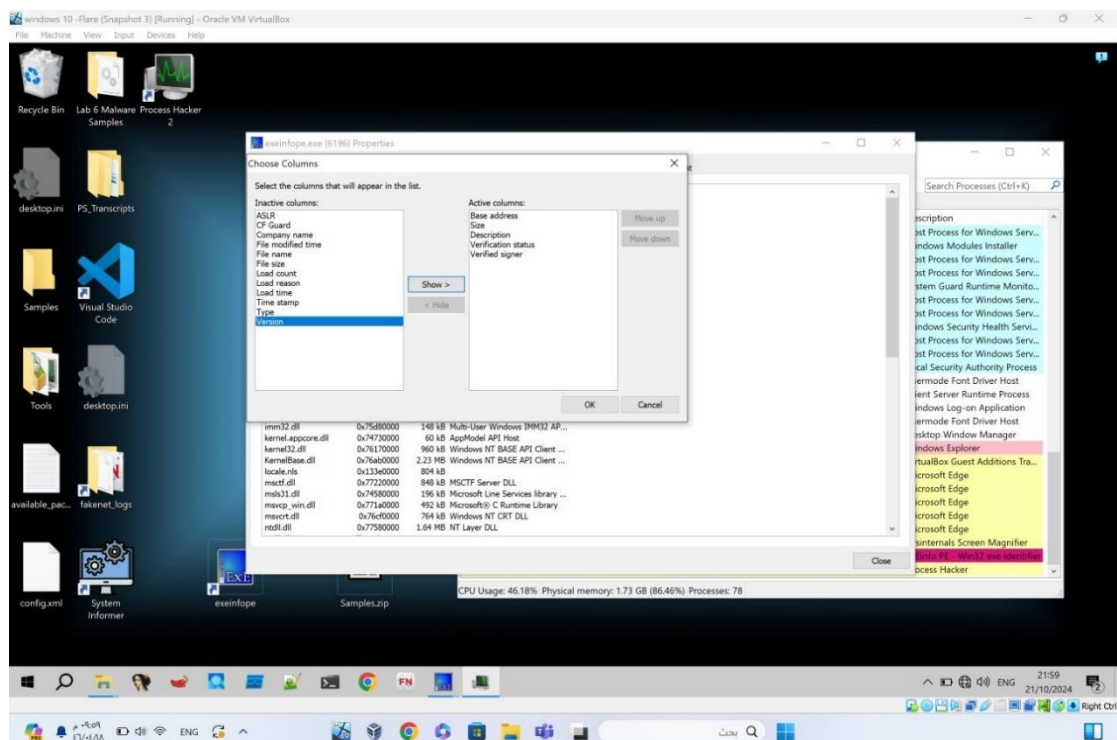
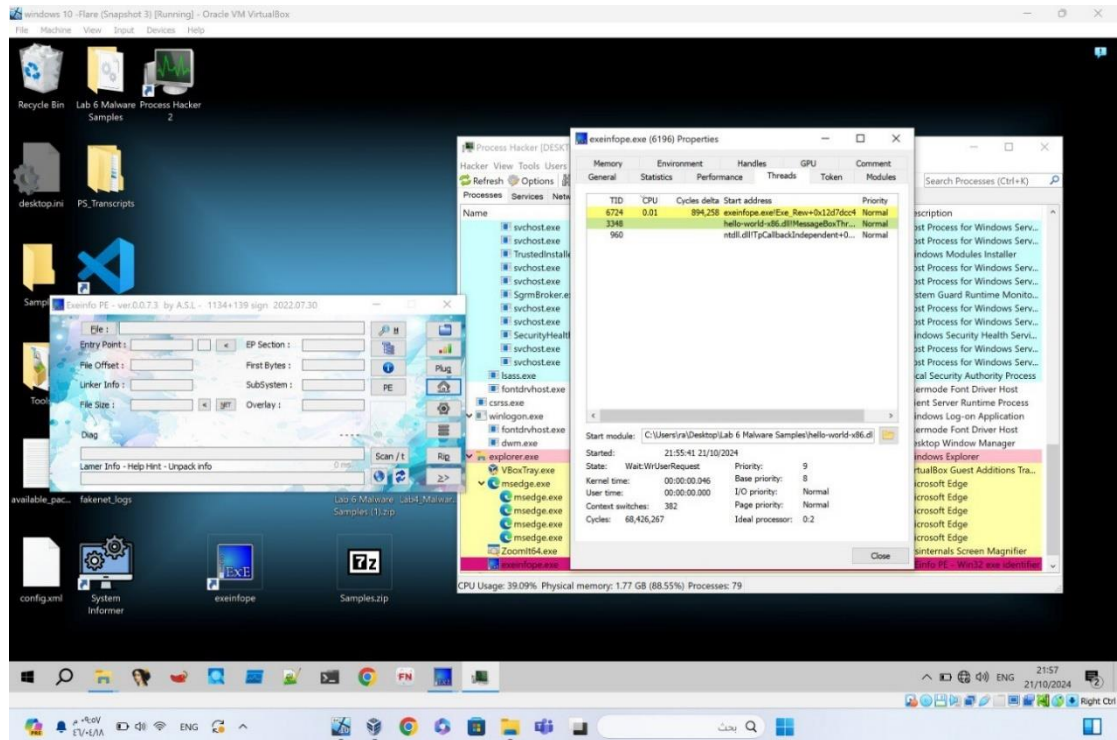


3- Screenshot of `exeinfo` Threads After Injection:

- After injecting the DLL, I took another screenshot to compare the thread activity.

4- Screenshot of Trusted and Not Trusted Modules:

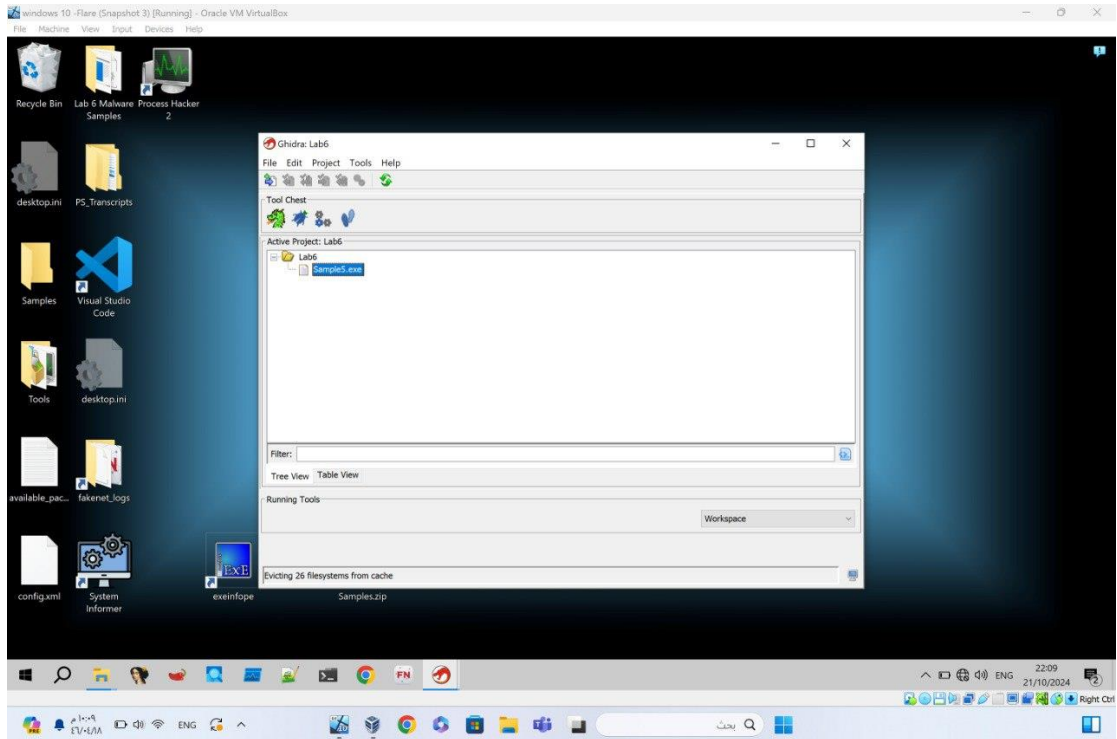
- I analyzed the modules in my system to differentiate between trusted and potentially malicious modules.



Task 2: Recognizing process injection pattern in disassembler (Ghidra):

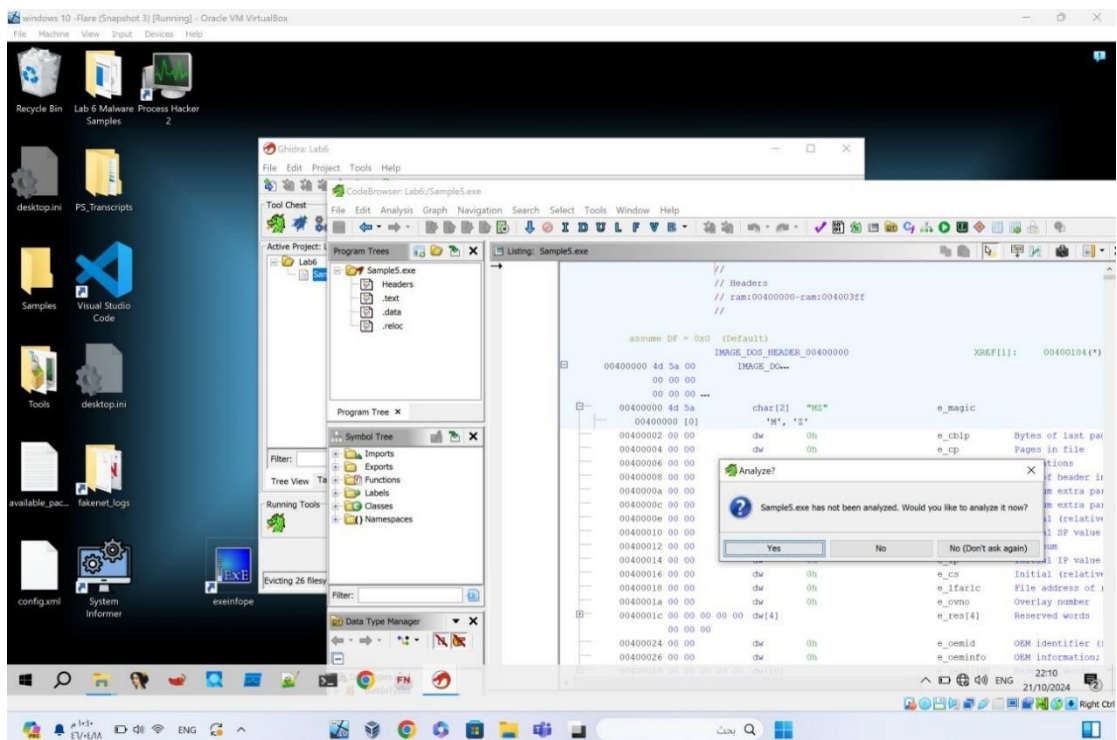
1- Recognizing Process Injection in Ghidra with Sample5.exe:

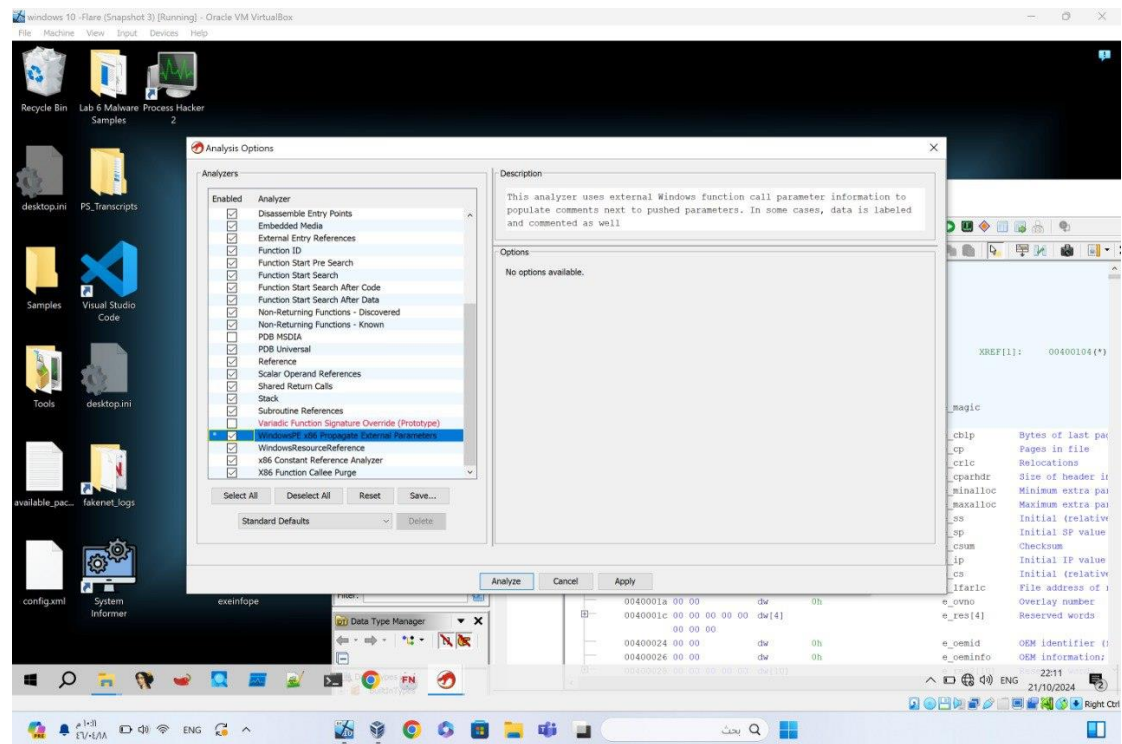
- I loaded Sample5.exe into Ghidra to analyze the injection pattern.



2- Screenshot of Enabling Windows PE Propagate External Parameters Option:

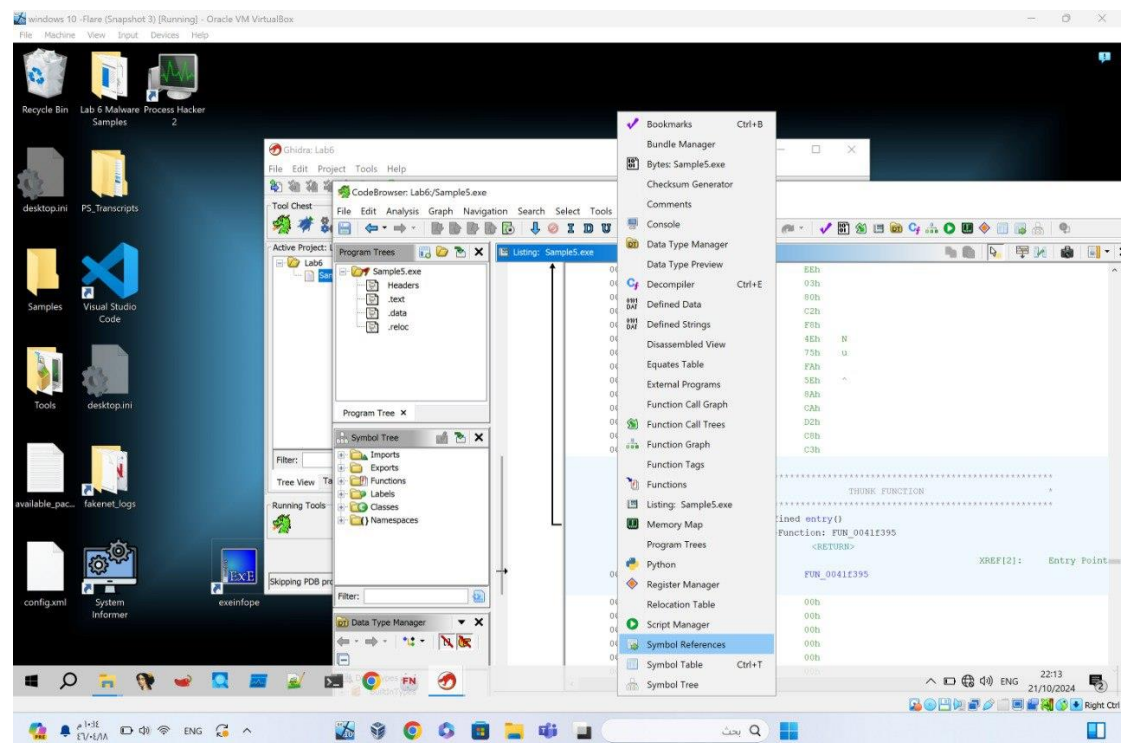
- I enabled this option in Ghidra to help analyze the process's parameters.

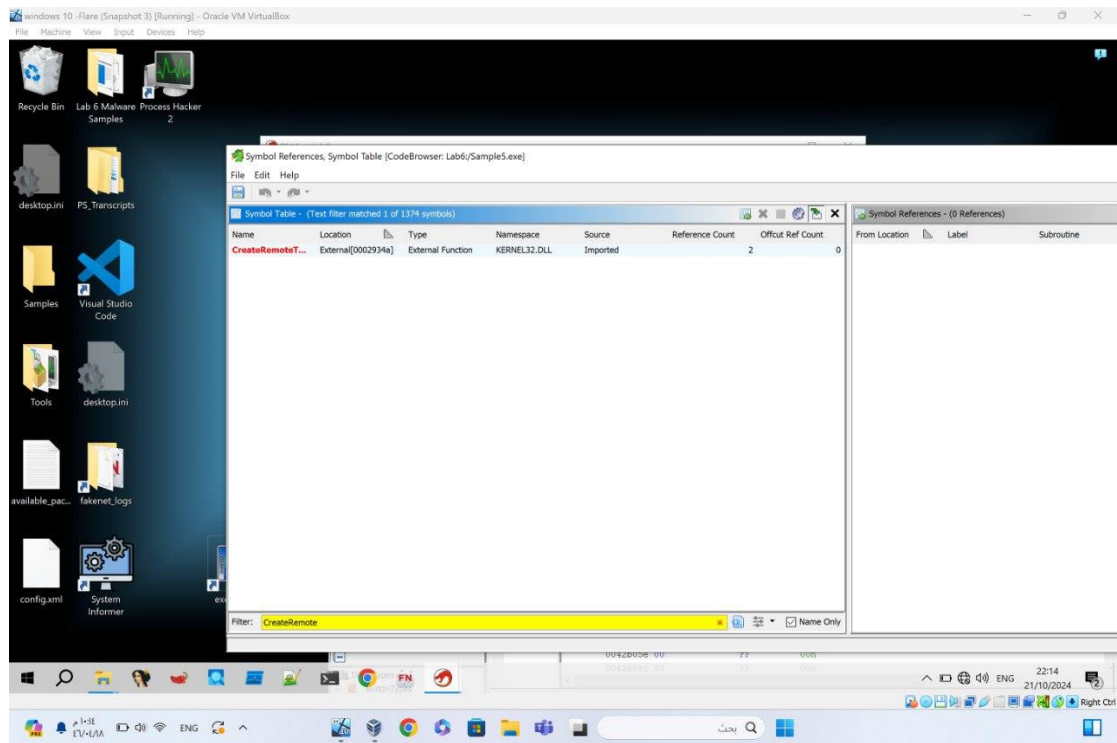




3- API to Identify Process Injection in Ghidra:

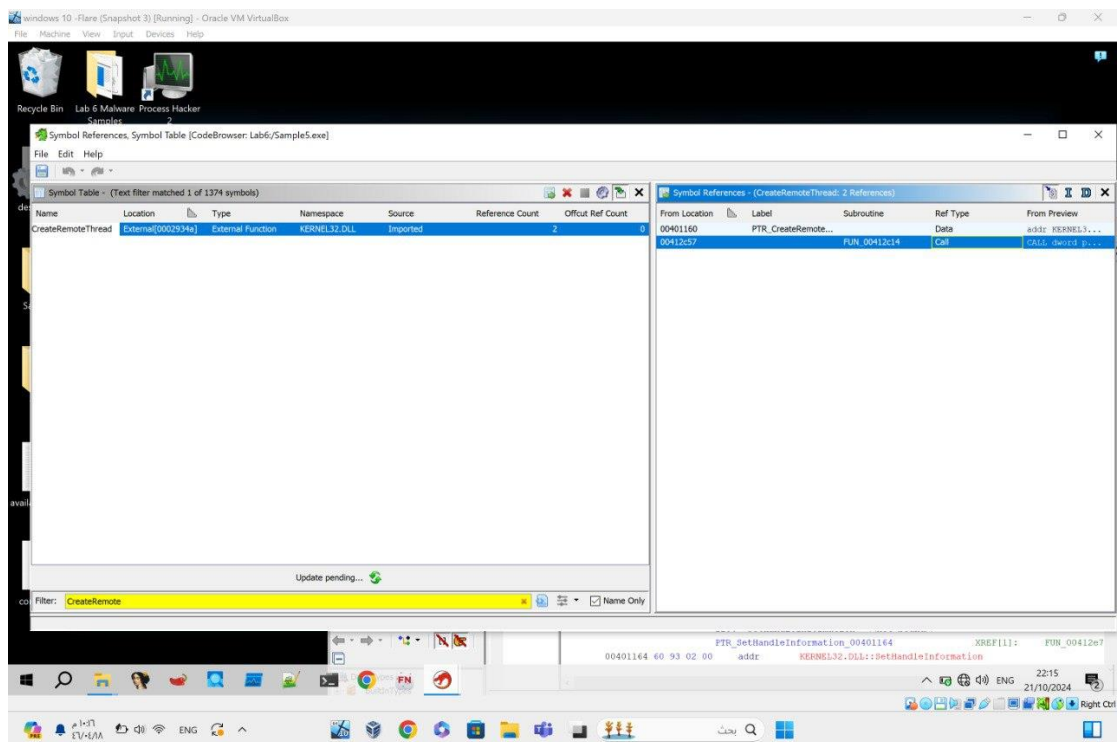
- Expected Pattern for Process Injection: Typical patterns include the use of suspicious APIs like VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread.
- The key API to start looking for is CreateRemoteThread.

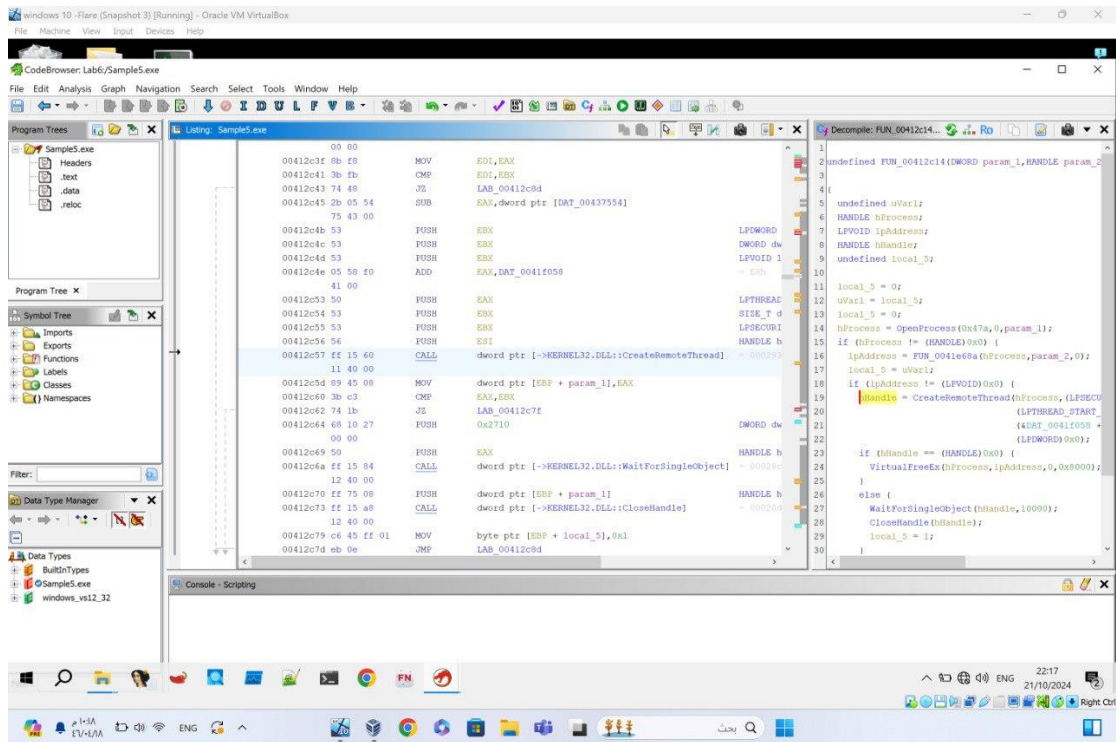




4- Screenshot of the CreateRemoteThread API Call:

- I captured the CreateRemoteThread API call in my analysis.








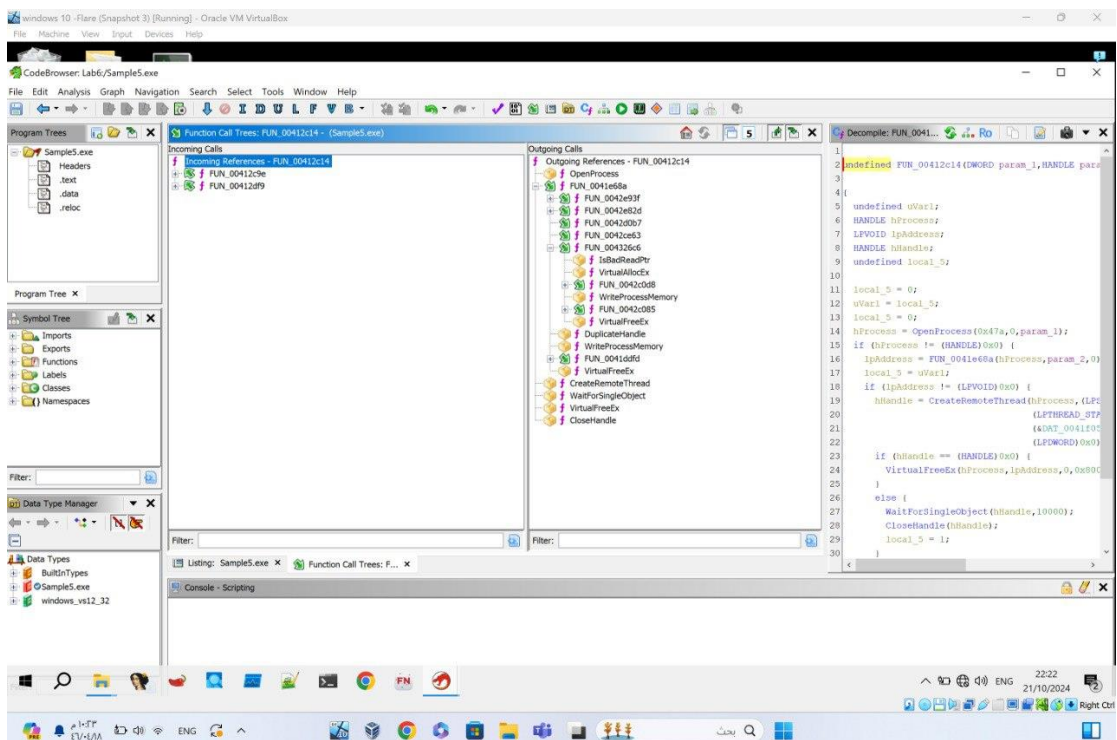
5- Screenshot of Function Call Tree for CreateProcessThread:

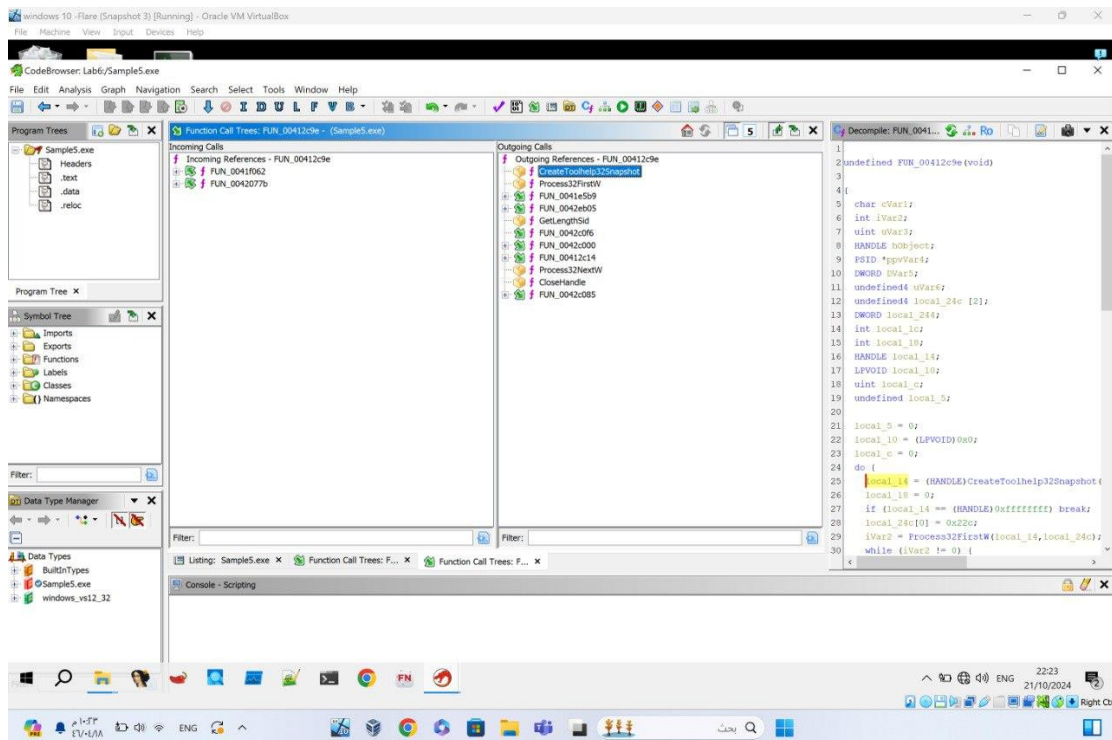
- I visualized the function call tree, showing the injection flow from one function to another. Our goal was to look in disassembler for these API patterns and we found it.

Code Analysis –
Process Injection and API Hooking (ref.2.ch5)

Simple DLL injection:

1. EnumProcesses or CreateToolhelp32Snapshot
2. OpenProcess
3. VirtualAllocEx
4. WriteProcessMemory
5. GetModuleHandle to get a pointer to the  which implements LoadLibrary
6. GetProcAddress to locate the address of the  LoadLibrary function in the 
7. CreateRemoteThread





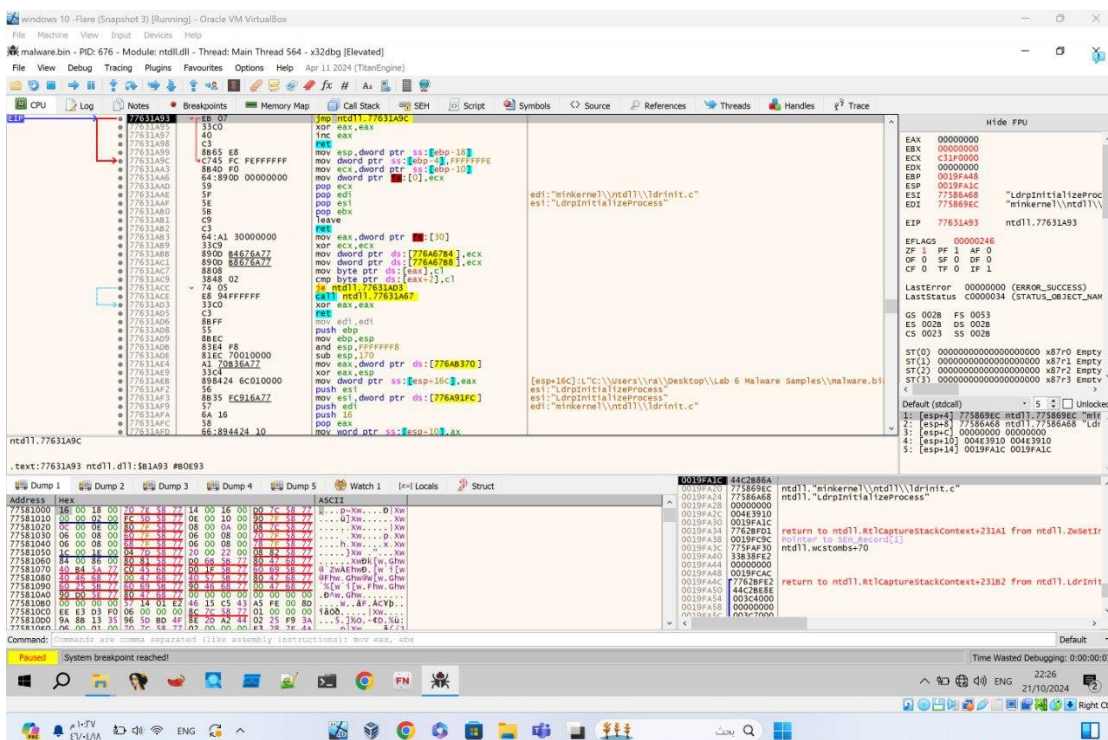
Task 3: Use the debugger (x32dbg) to unpack/extract the code before being injected:

1- Unpack/Extract Code in x32dbg with Malware.bin:

- I used the malware.bin sample and x32dbg to analyze code injection.

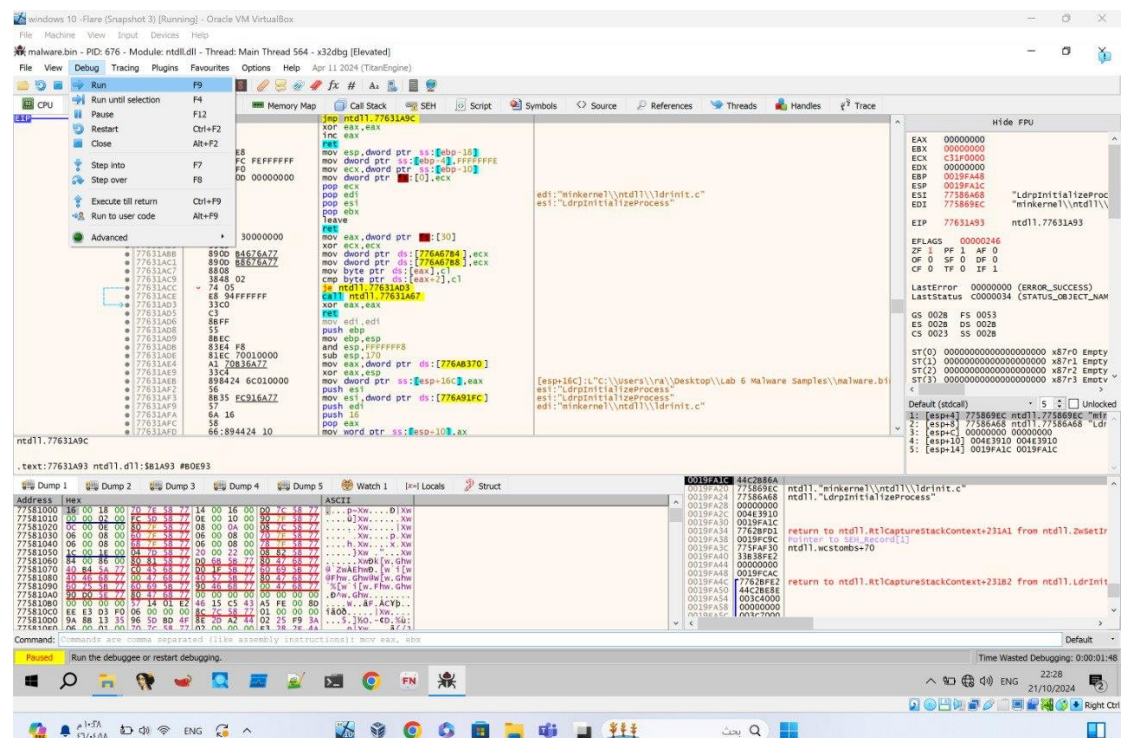
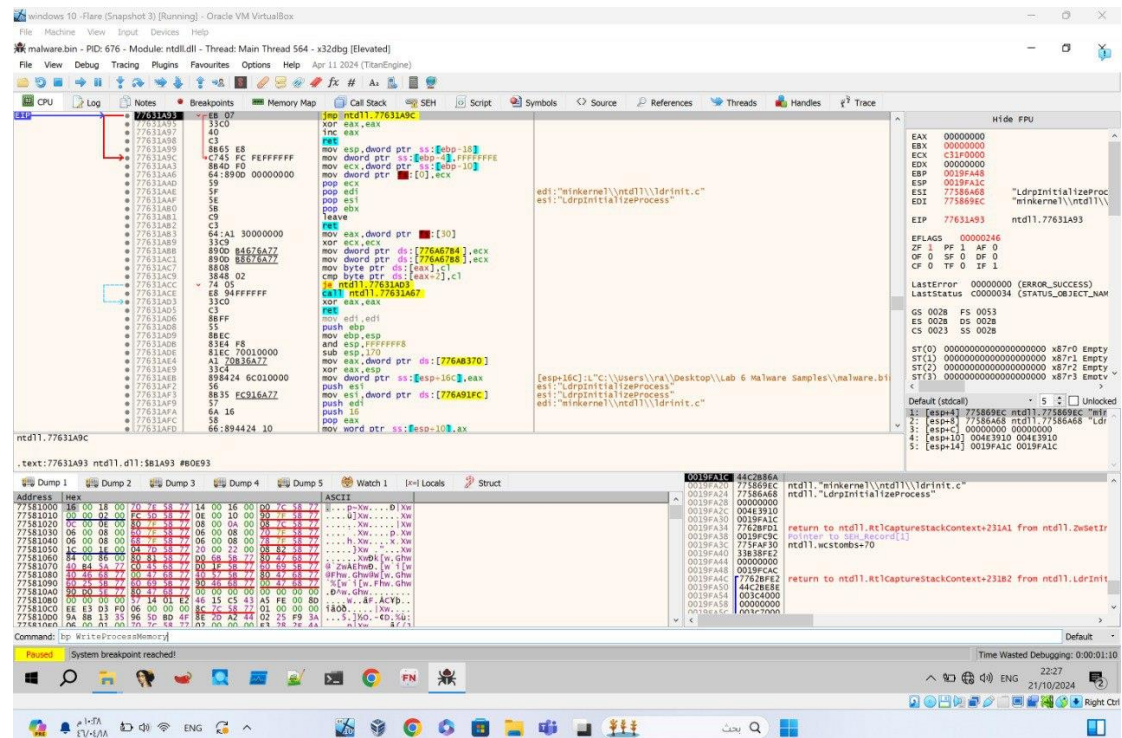
2- Screenshot of Debugger Before Adding Breakpoints:

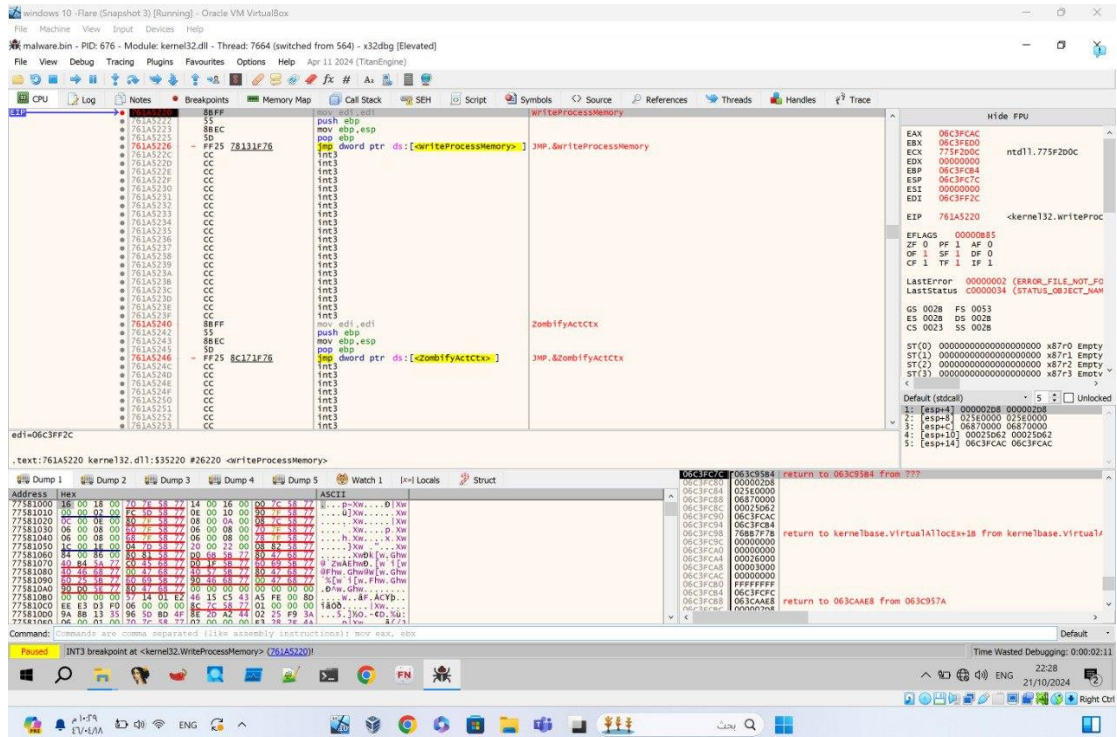
- I captured a screenshot of the debugger state before setting any breakpoints.



3- Screenshot of API Parameters:

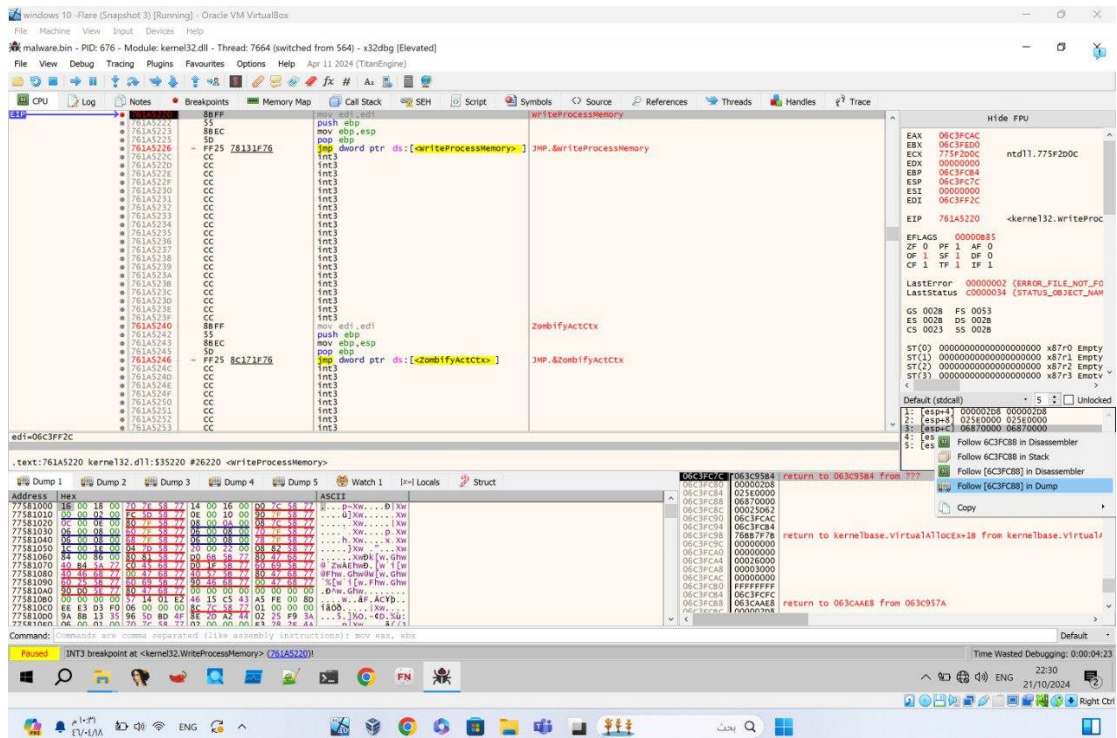
- I captured the parameters passed to the injection API during runtime.





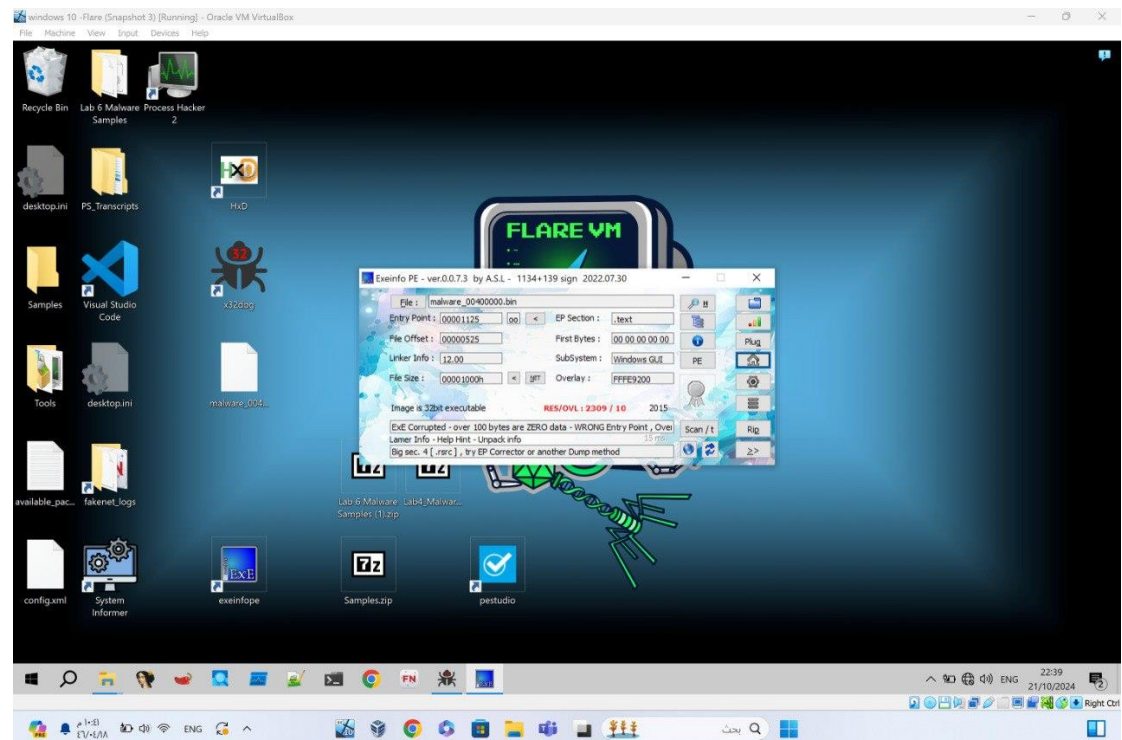
4- Screenshot of Third Parameter in Dump:

5- Screenshot of Contents of Dump1 and Dump2:



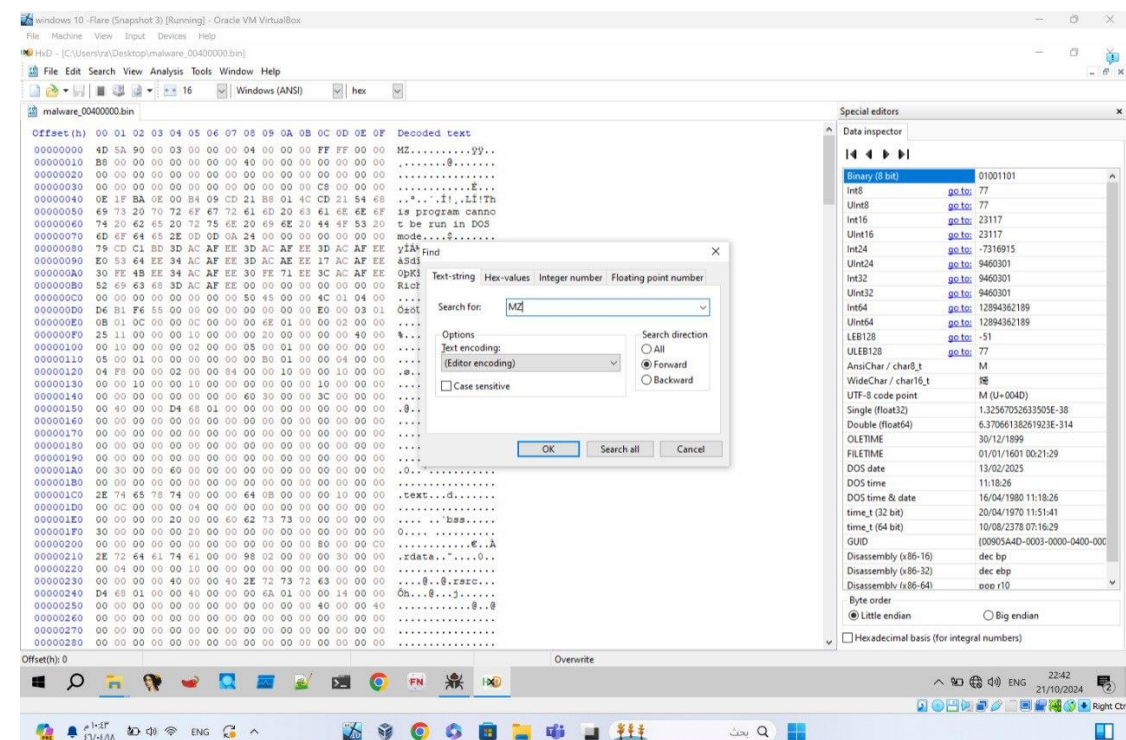
6- Analyzing Extracted Code with `exeinfo`:

- I analyzed the code with `exeinfo` but found that it did recognize the format when it shouldn't, but it said it is corrupted.



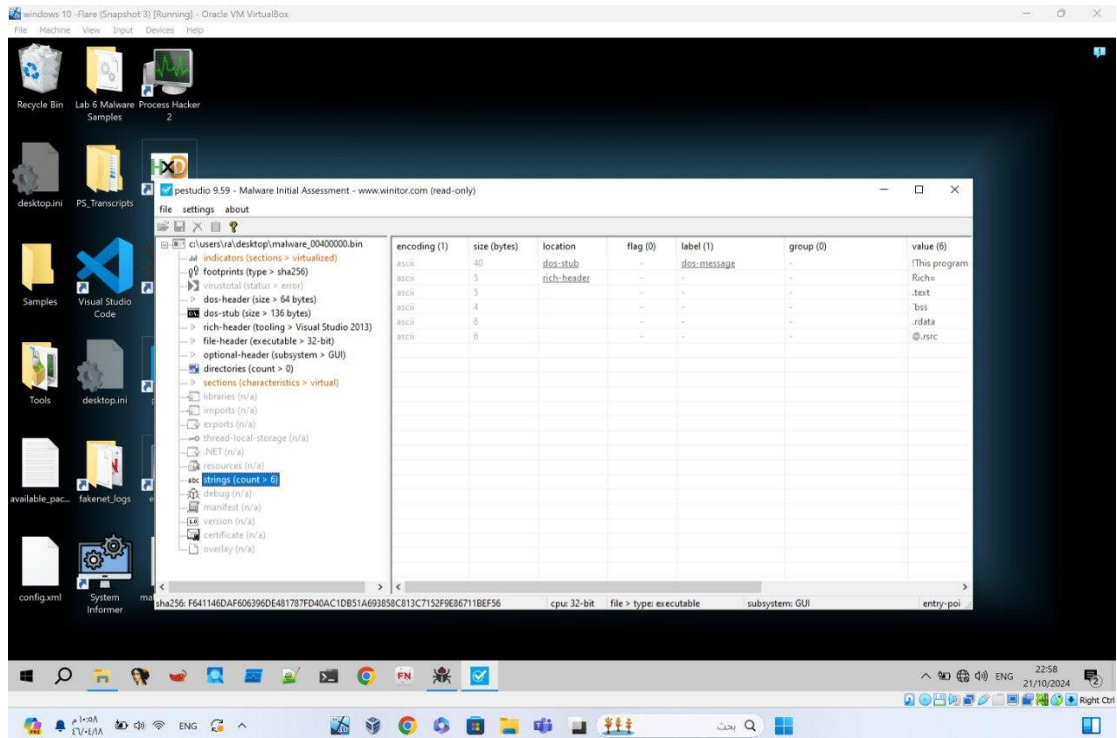
7- Screenshot of Extracted Code with HxD:

- I opened the code in HxD (a hex editor) to inspect and correct the data but it already in correct format?!



8-Screenshot of Corrected Code in `pestudio`:

- I took another screenshot after analyzing the file with `pestudio` but it did not show anything.



What is the difference between EXE and DLL?

EXE (Executable) is a file that directly runs a program, while a **DLL** (Dynamic Link Library) is a file used by programs to extend their functionality.

From this step, how can you detect malicious DLL?

Malicious DLLs can be detected by checking for unusual or unsigned modules.

Why was the process terminated after a while?

Corruption from the DLL, or a crash caused by improper injection.

List the pattern that you would expect in process injection technique.

Typical patterns include the use of suspicious APIs like VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread.

Which API you should start looking for, to identify process injection in Ghidra?

The key API to start looking for is CreateRemoteThread.

Write the instruction at 0x00412C53 ?

PUSH EAX

What parameter is pushed at 0x00412C53 ?

lpStartAddress for the CreateRemoteThread function

Go to Microsoft documentation and explain what does the parameter do at 0x00412C53?

- lpBaseAddress is a pointer to the base address in the specified process where data will be written.
- Before the system transfers data to this location, it verifies that all data in the base address and the memory of the specified size are accessible for writing. If the memory is not accessible, the function will fail.

Which API we should investigate to locate the injected code?

The API to investigate is WriteProcessMemory, which is key for memory injection.

Why do we need to create a breakpoint at that API in b?

We need a breakpoint to pause execution when the API is called, allowing us to inspect the injected code.

Which parameter is at [esp+C]? check the documentation of the API from Microsoft website.

lpBuffer typically points to a region of memory that contains the data to be written to the target process or used during execution.

What can you conclude from this message “This program can not be run in DOS mode”?

This indicates that the extracted code is likely a Windows executable, not a DOS binary.

Why exeinfoPE did not recognize the extracted code?

ExeinfoPE likely did not recognize the extracted code because the extra data before the **MZ** header confused it. Tools like ExeinfoPE rely on recognizing specific file signatures (such as the **MZ** header) to identify the type of file and its format. If there is non-standard data or additional content placed before the **MZ** signature, the tool may fail to detect it as a valid executable, leading to misidentification or failure to analyze the file.