De Montfort University Dubai

APRIL 2025

MALWARE ANALYSIS

Made by: Meyyappan Venkatesh
CTEC3754D
P2766441

4000 WORDS
(excluding cover page, Table of contents, Bibliography)

Contents

PART 1

Question 1

The file sus-file1.pdf was extracted from the archive sus-file1.7z and analyzed to check if it is harmful or suspicious. The tools used are PDFStreamDumper , pdfid , and peepdf.

By opening sus-file1.pdf using pdfid, we can clearly observe that the presence of only one trailer (**/Trailer: 1**) suggests that the file was created in a single operation and has not been modified since. Further analysis reveals that the file contains 146 objects, and by examining the details in the image below, it is evident that the document consists of only 1 page.

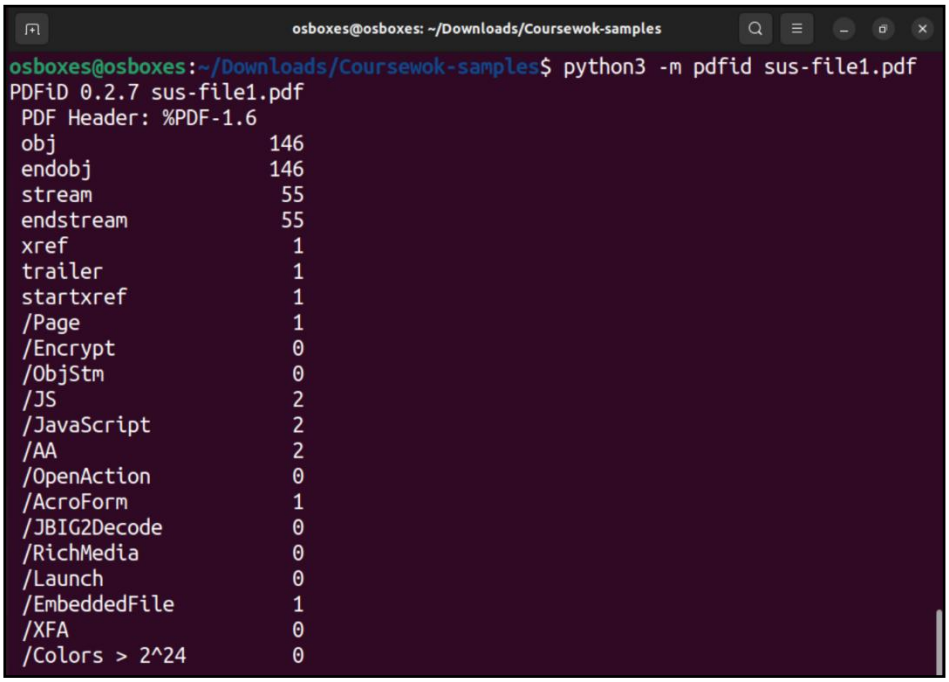

Figure 1: Pdfid Output

Upon opening the file in peepdf, we can see the results below in the image.

1. **AcroForm [8, 144]** – Represents interactive form fields, suggesting the PDF may have fillable forms.
2. **AA [11, 48]** – Defines user-triggered actions or automatic actions when opened (e.g., clicks), indicating possible dynamic behaviour.
3. **JavaScript [141, 142]** – Contains embedded scripts, which can enable dynamic features or pose security risks.
4. **Embedded Files [8, 144]** – Indicates attached files, which may include additional content or potential threats.

```
PPDF> open /home/osboxes/Downloads/Coursework-samples/sus-file1.pdf

File opened succesfully!!

Warning: PyV8 is not installed!!


File: sus-file1.pdf
MD5: 15d8b554bc3e87889c3199c4faa82d48
SHA1: 8b6e1fcad823d24c8b38a61d2a10c617ed2a8976
Size: 149387 bytes
Version: 1.6
Binary: False
Linearized: True
Encrypted: False
Updates: 0
Objects: 146
Streams: 55
URIs: 0
Comments: 0
Errors: 0

Version 0:
        Catalog: 8
        Info: 6
        Objects (146): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 6
4, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124,
125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146]
        Streams (55): [5, 17, 19, 20, 22, 24, 25, 35, 37, 38, 40, 42, 43, 52, 59, 61, 62, 64, 65, 66, 68, 69, 70, 71, 72, 73, 74, 75,
80, 81, 85, 88, 89, 93, 96, 98, 101, 103, 106, 108, 111, 113, 116, 118, 121, 123, 126, 128, 131, 133, 136, 138, 140, 143, 144]
                Encoded (37): [52, 62, 66, 68, 69, 70, 71, 72, 73, 74, 75, 80, 81, 85, 88, 89, 93, 96, 98, 101, 103, 106, 108, 111, 11
3, 116, 118, 121, 123, 126, 128, 131, 133, 136, 138, 143, 144]
        Objects with JS code (4): [5, 141, 142, 144]
        Suspicious elements:
                /Names (2): [4, 8]
                /AcroForm (2): [8, 144]
                /AA (2): [11, 48]
                /XFA (1): [144]
                /JS (2): [141, 142]
                /JavaScript (2): [141, 142]
                /EmbeddedFiles: [8]
                /EmbeddedFile: [144]

PPDF>
```

Figure 2: PPDF Output suspicious elements

Now using pdfStreamDumper we will first analyse the JavaScript objects present inside the pdf file

- Object 141: Uses **AFDate_FormatEx("mm/dd/yyyy")** for dynamic date formatting in interactive fields (Figure 3).
- Object 142: Employs **AFDate_KeystrokeEx("mm/dd/yyyy")** to validate user input for dates (Figure 4).

These scripts indicate the PDF supports interactive form functionality.

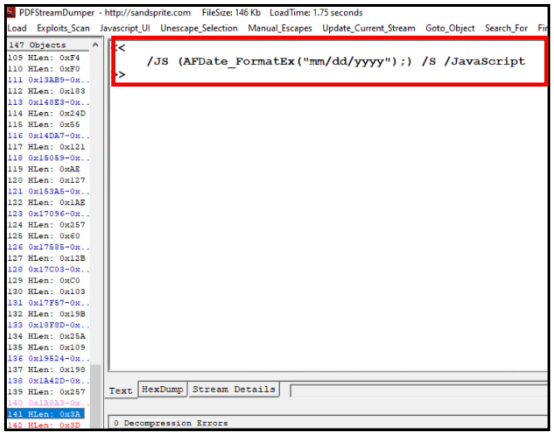

Figure 3: Object 141

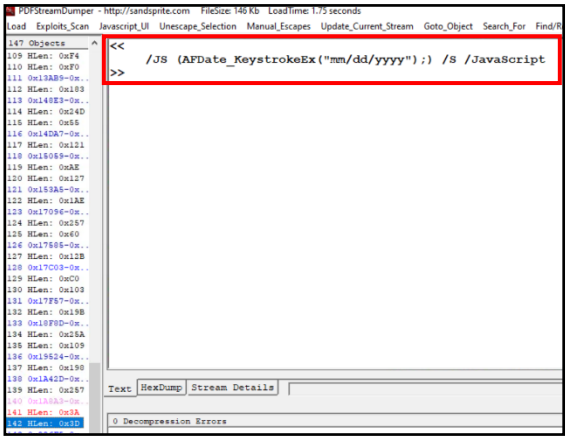

Figure 4: Object 142

Now moving on to the **/EmbeddedFiles** dictionary was located within Object 8, And when we scroll to the bottom, we can see the details as in (image 5). Upon further inspection, we observed the following details:

- The **/Names** array within the **/EmbeddedFiles** dictionary specifies the path to an embedded file:

  [/home/davidemaiorca/workspace/ProvaPDF/src/compressed/asdkjwx.pdf]

- This file (**asdkjwx.pdf**) is referenced as **145 0 R**, where **145** is the object number for the embedded file.

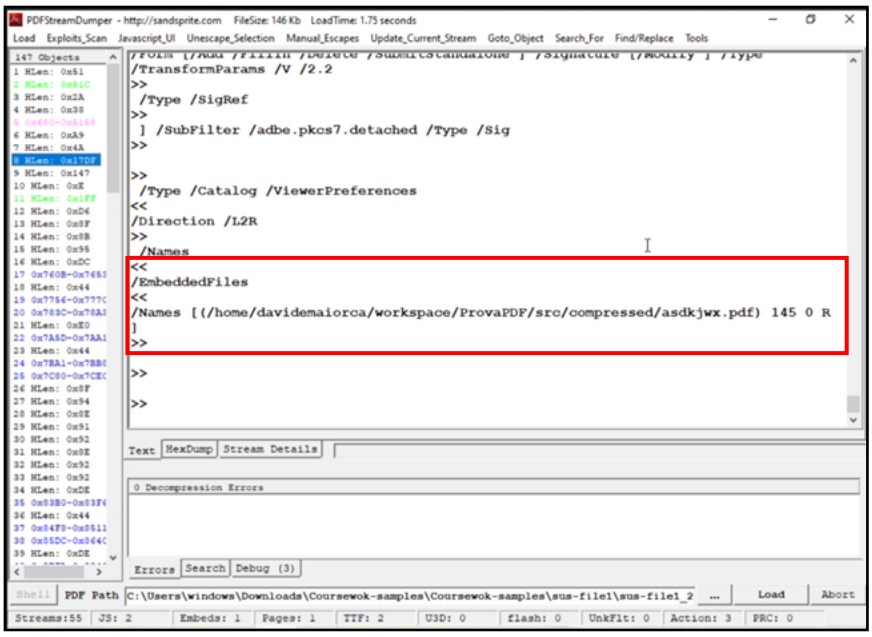

Figure 5: Object 8 leads to object 145

When we follow the reference to Stream 145, which corresponds to the embedded file **asdkjwx.pdf** (object **145 0 R**), we observe the following output as in the below (image 6)

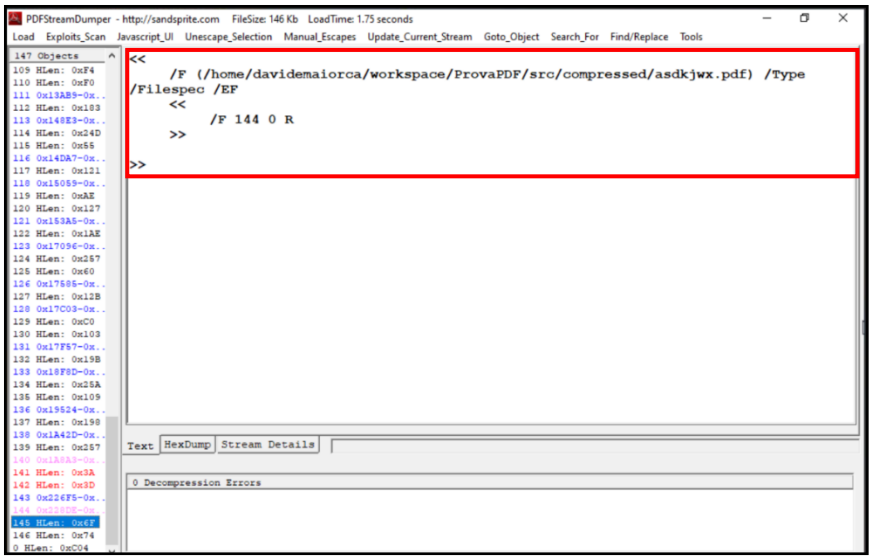

Figure 6: Object 145 referenced to Object 144

The nested dictionary contains a reference to another object (**/F 144 0 R**). This suggests that additional details about the embedded file, such as its content or metadata, may be stored in Object 144. And /F specifies the path to embedded file. Following 144 objects, the output is (fig 7)

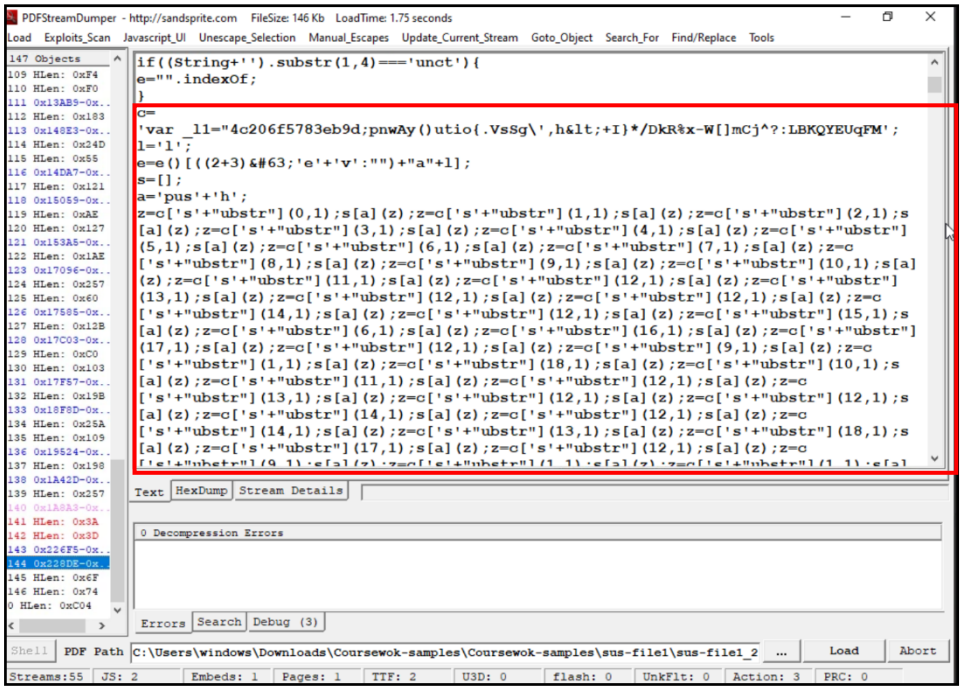

Figure 7: Object 144 encoded javascript code

1. **Dynamic String Building:** The code uses functions like substr and [] to build strings at runtime, likely to reconstruct hidden or malicious content. z = C['s' + 'ubstr'](0, 1); s[a](z);
2. **Obfuscated Variables:** Variable names and values are encoded in unusual ways to avoid detection. C='var _ll="4c206f5783eb9d;pnwAy()utio{.VsSg\'..... "';
3. **Conditional Code Checks:** It uses conditions to check parts of the code environment, For constructing and deconstructing the string again.

The presence of a randomly named embedded file (**asdkjwx.pdf**) and heavily obfuscated JavaScript code in Object 144 strongly suggests malicious intent. These elements are commonly used to evade detection and deliver harmful payloads, making the file highly suspicious and likely malicious. We can further confirm upon seeing the windows defender alert when the file is extracted in the pc.

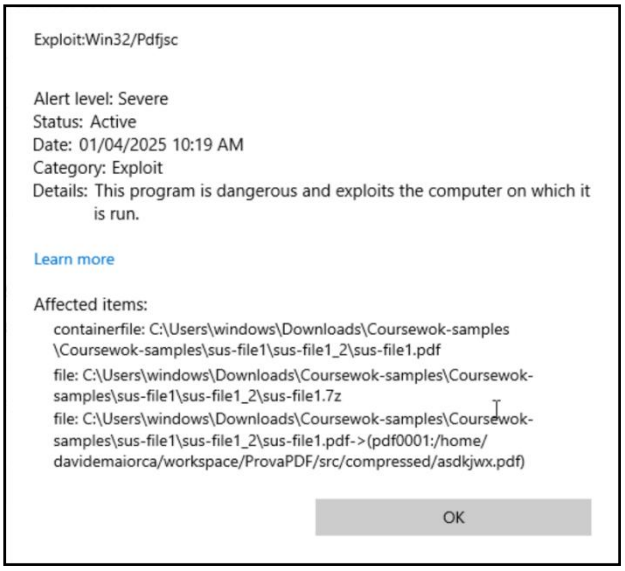

Figure 8: Windows Defender Alert

Question 2

The task involves retrieving sus_file2 from sus-file2.7z. Once extracted, the file appears in the format sus_file2.file, with an unknown extension. Upon analysing the file using PEview and Mitec EXE software, we conclude that it is a Portable Executable file, as indicated by the Hex values 4D5A (MZ), which are the signature of PE files.

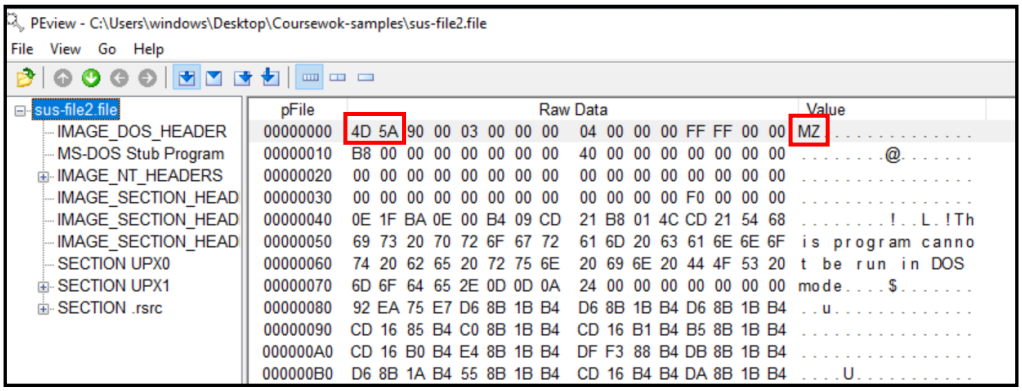

Figure 9: 4D5A MZ PEview Output

Upon opening sus-file2.file in PEiD, we observe a high entropy value of 7.98, which proves that the file is packed. And further PEiD confirms it is packed using UPX

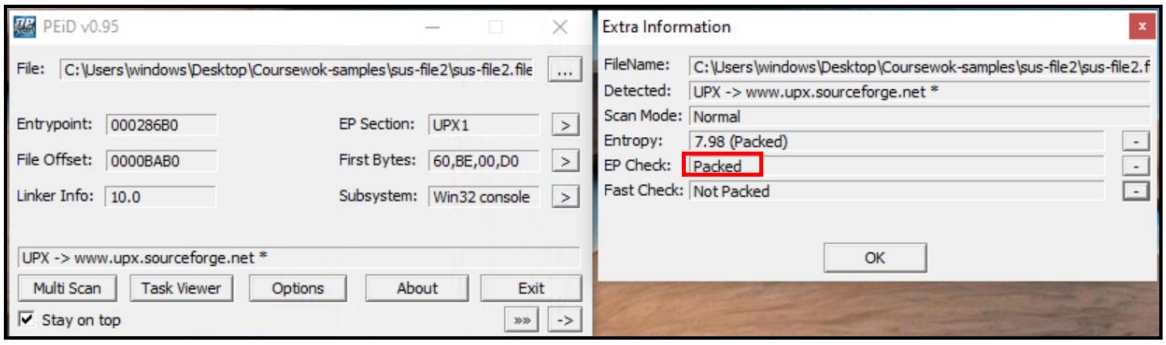

Figure 10: PEid Packed Confimation

From the below image, using MITeC EXE, it can be concluded that the file is a 32-bit Portable Executable. A key observable indicator is the variation between the Virtual Size and Raw Data Size of the sections (Image 11). Such variations are characteristic of packed files, where data is compressed or obfuscated.

UPX0 Section: The **Virtual Size** of the UPX0 section is **114,688**, while its **Raw Data Size** is **0**. The section is fully loaded into memory but contains no data on disk, which suggests it is compressed (packed)

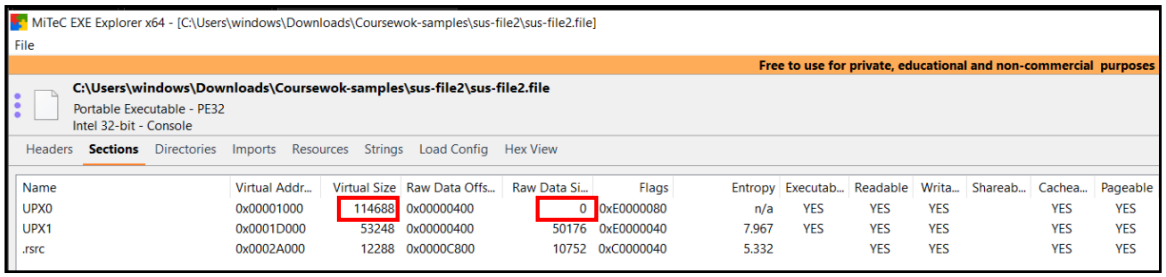

Figure 11: Mitec Indicating Packed file

These observations strongly suggest that the file has been packed, likely using a tool UPX, to reduce its disk size while preserving its runtime functionality. To unpack the file using the UPX tool and execute it, the following command is used (figure 12):

```
upx -d C:\Users\windows\Desktop\Coursewok-samples\sus-file2\sus-file2.file -o unpacked-sus-file2.exe
```

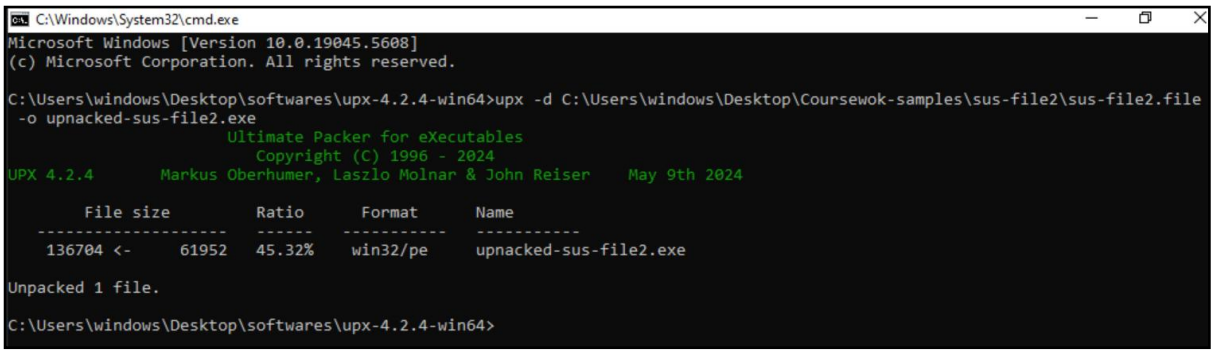

Figure 12: UPX unpacking

This command uses UPX to decompress the packed executable. The **-d** flag specifies decompression, and the **-o** flag defines the output file name (**unpacked-sus-file2.exe**). Successfully unpacking the file increased its size from 61 KB to 134 KB (figure 13). The analysis confirmed **sus_file2.file** as a UPX-packed PE32 executable.

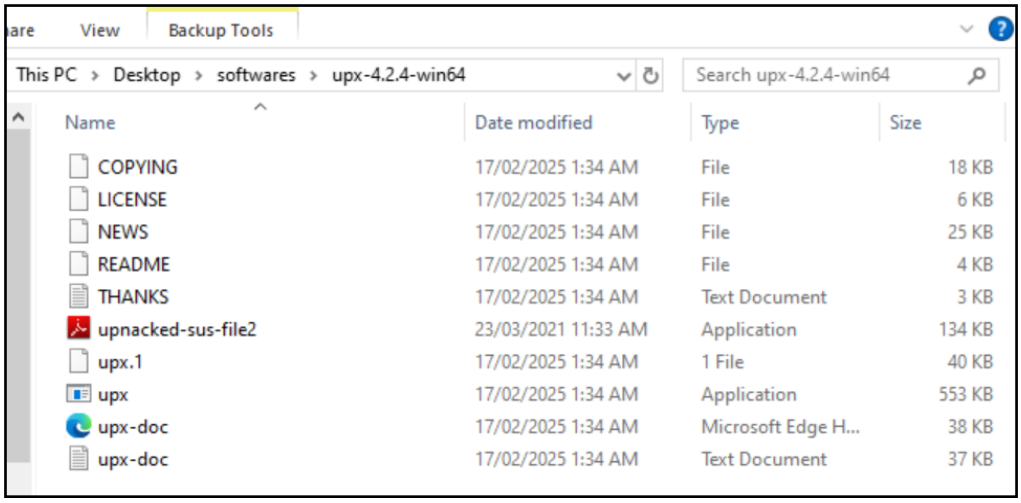

Figure 13: upx unpacked EXE

Question 3

If a user receives **sus_file3** (zipped in **sus-file3.7z**) via email and extracts it, revealing a .dll file, the system is unlikely to be infected. Windows does not execute .dll files automatically.

Additionally, attempting to open **sus_file3.dll** directly on a Windows system confirms its non-executable nature. When double-clicked, the system prompts the user to select an application to open the file, as shown below:

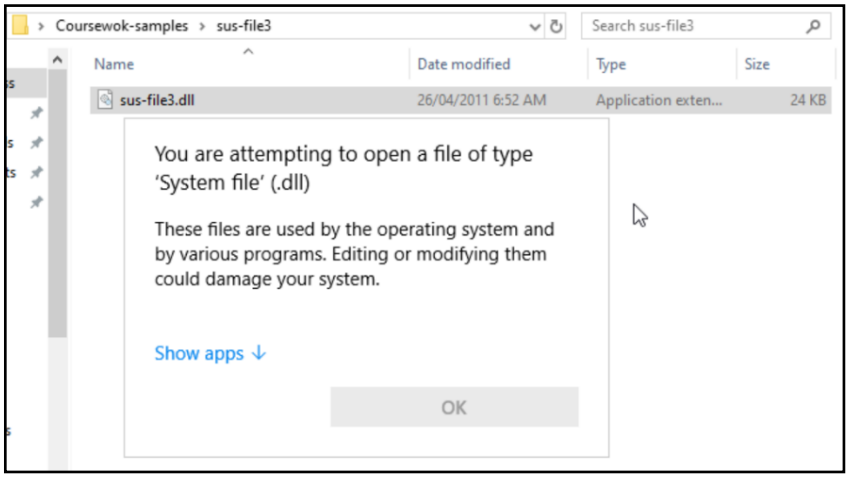

Figure 14: dll unable to open

To confirm that sus_file3.dll is indeed a Dynamic Link Library (DLL), we analyzed it using PEiD , a tool designed to identify packers and compilers in executable files. The PEiD output clearly indicates that sus_file3.dll is a **.dll file compiled using Microsoft Visual C++ 6.0** (figure 15).

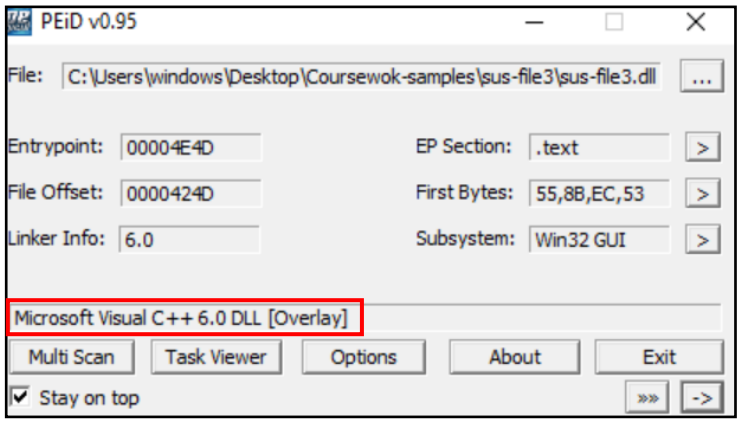

Figure 15: PEid Tool Output

A **.dll** (Dynamic Link Library) file cannot run by simply double-clicking it because it is not a standalone executable. Unlike **.exe** files, which launch programs directly, **.dll** files contain code, resources, or functions used by other programs at runtime. They act as modules that applications load when needed, rather than running independently (Ballderson, 2023). Hence .dll files remain inactive unless called by another program.

Question 4

The file **sus-file3.dll** DLL, as identified by MiTeC EXE Explorer. The static analysis focuses on extracting meaningful information from the file's structure and embedded strings to infer its potential behaviour.

Static Analysis of the DLL

1. The DLL installs as a persistent service named "IPRIP," with a misleading display name "Intranet Network Awareness (INA+)" and a description suggesting network data collection. It ensures persistence by modifying registry entries under SYSTEM\CurrentControlSet\Services, allowing it to run automatically at startup and survive reboots (Figure 16).

2. The service's ImagePath points to the legitimate binary %SystemRoot%\System32\svchost.exe -k netsvcs, loading the DLL within a shared svchost.exe process used for network services. By running under svchost.exe, a trusted Windows process, the DLL blends in with legitimate activity, avoiding suspicion while sharing resources with genuine services (Figure 16).

3. Functions like CreateServiceA, OpenSCManagerA, RegCreateKeyA, and RegSetValueExA show the DLL can register itself as a service and manipulate its configuration via the Service Control Manager (SCM). These APIs are used for installing and managing services. Editing registry keys boosts the malware's persistence and control (Figure 16).

4. The use of CreateProcessA and command strings like cmd.exe /c reveals the DLL can launch child processes to run arbitrary commands or scripts (Figure 17).

5. The DLL resolves **practicalmalwareanalysis.com** and requests serve.html, establishing an outbound connection to a command-and-control (C2) server. Using WININET.dll functions like InternetOpenA, HttpSendRequestA, and InternetReadFile, it communicates over HTTP/HTTPS (Figure 17).

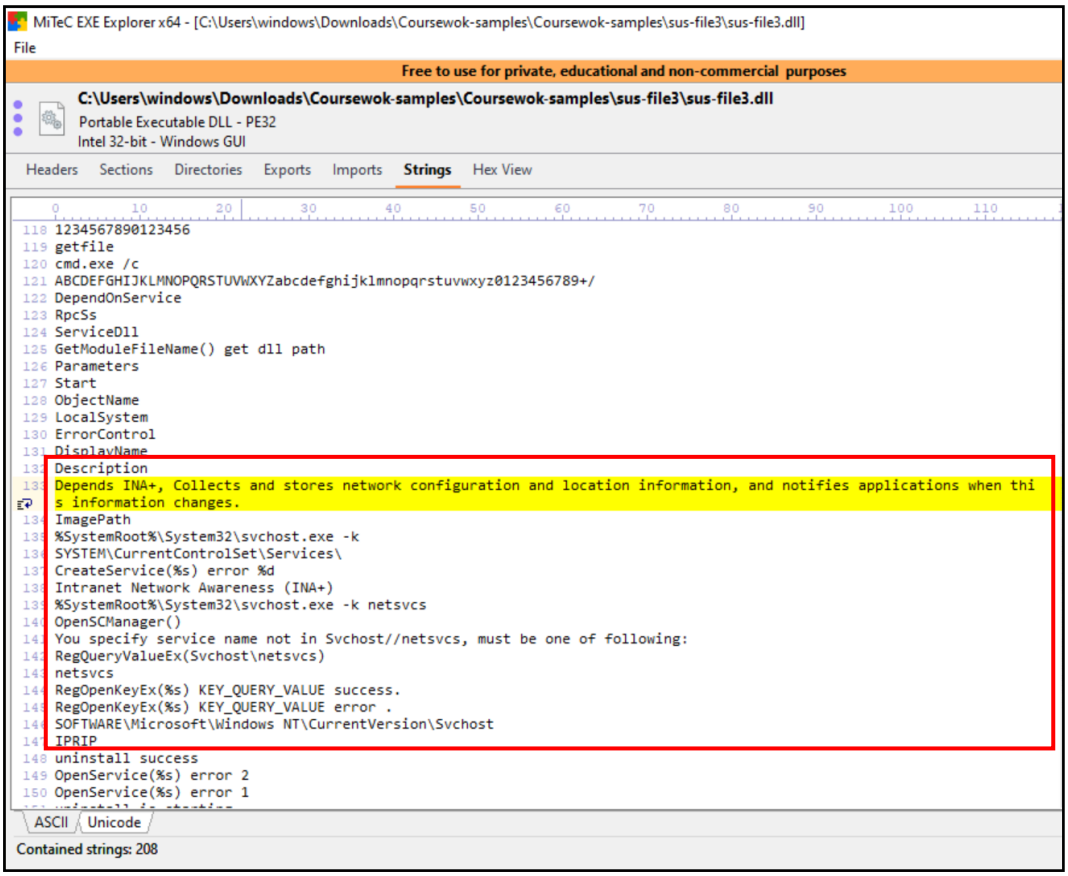

Figure 16: Static analysis of sus-file3.dll viewing strings

6. The DLL uses Base64-encoded strings to add obfuscation to evade detection, and analysis (Figure 17).
    - cXVpdA== → "quit"
    - Y29ubmVjdA== → "connect"
    - c2xlZXA= → "sleep"
    - Y21k → "cmd"
    - dW5zdXBwb3J0 → "unsupport"

7. The DLL relies on services like RpcSs (Remote Procedure Call) and defines dependencies in DependOnService to start only after critical Windows services are active (Figure 17).

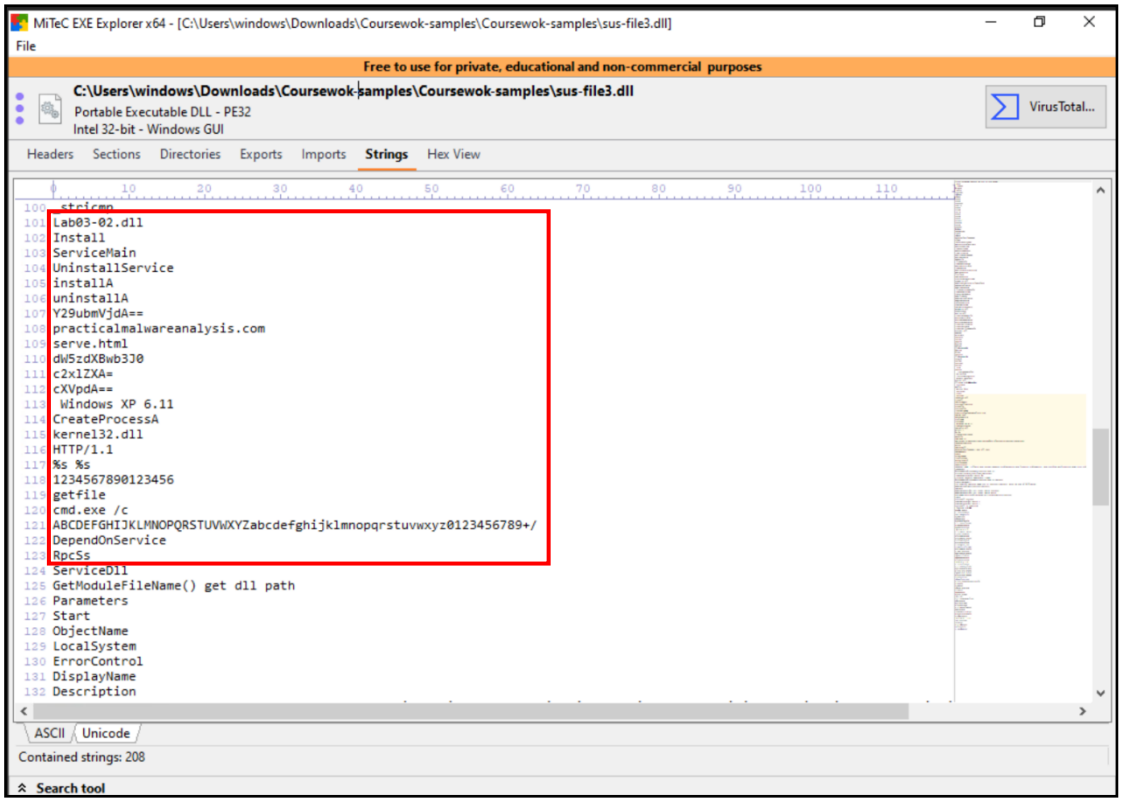

Figure 17: Base64 Encoded strings and process

Now, we move to dynamic analysis to observe the actual behaviour of the DLL, to observe its behavior in real-time, we prepare for dynamic analysis using fakenet, regshot, Process Explorer:

1. Regshot: Take an initial snapshot of the system state to track changes.
2. Process Explorer: Launch Process Monitor to capture file system, registry, and process activity.

Before running the DLL dynamically, we locate the exported functions using MiTeC EXE Explorer. In the Exports section (Figure 18), we observe several functions:

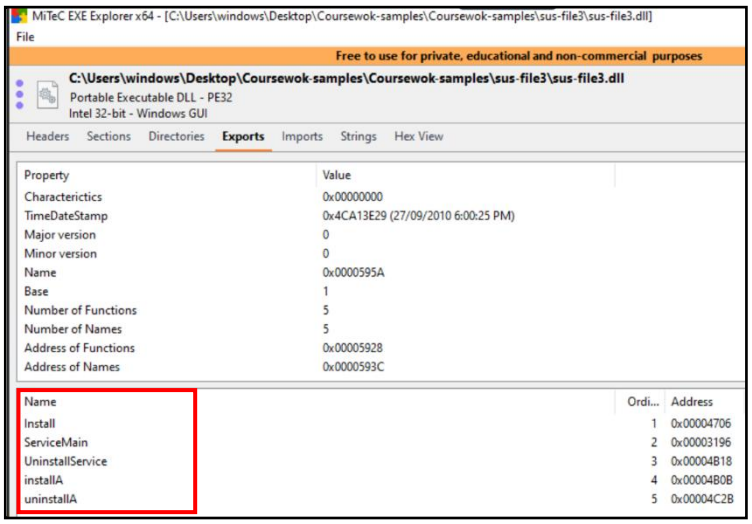

Figure 18: Exports of DLL file to find entrypoint

Among these, the function **install** stands out as a likely entry point for initializing the DLL's malicious behavior. This function is explicitly named to suggest its role in installing or configuring the malware on the system. To run the DLL dynamically, we use the following command figure 43.

```
rundll32.exe sus-file3.dll, install
```

Fakenet Observations

After executing the function 'installA', Fakenet did not capture any network activity related to the expected C2 communication with '**practicalmalwareanalysis.com**'. The absence of outbound connections suggests that the DLL may trigger its network-related functionality under different conditions. (Figure 19)

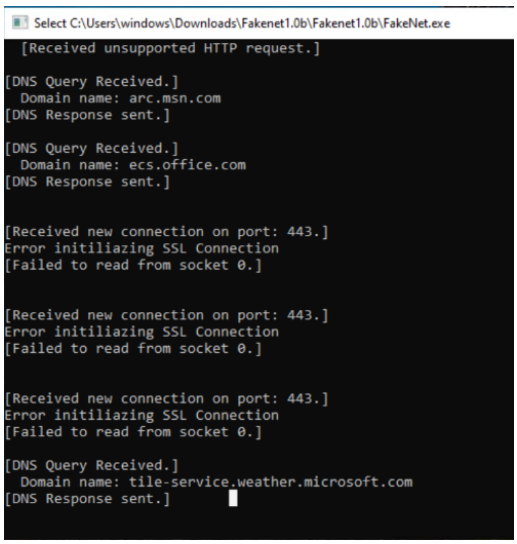

Figure 19: FakeNet No requests

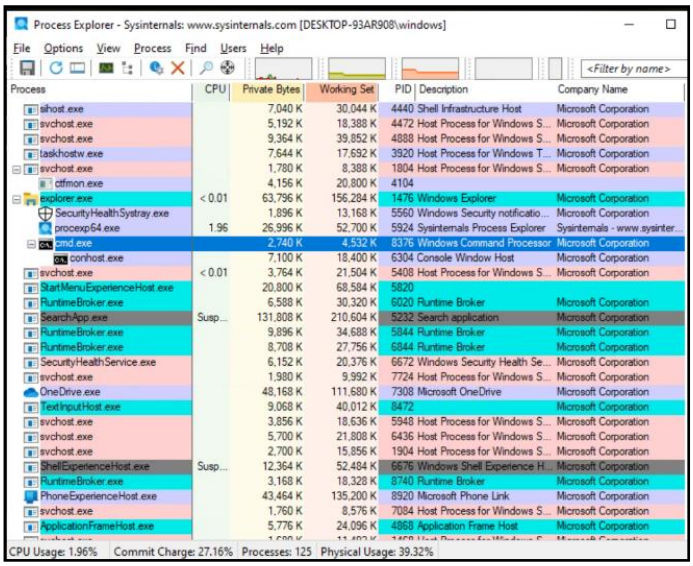

Figure 20: No INA+ Process created

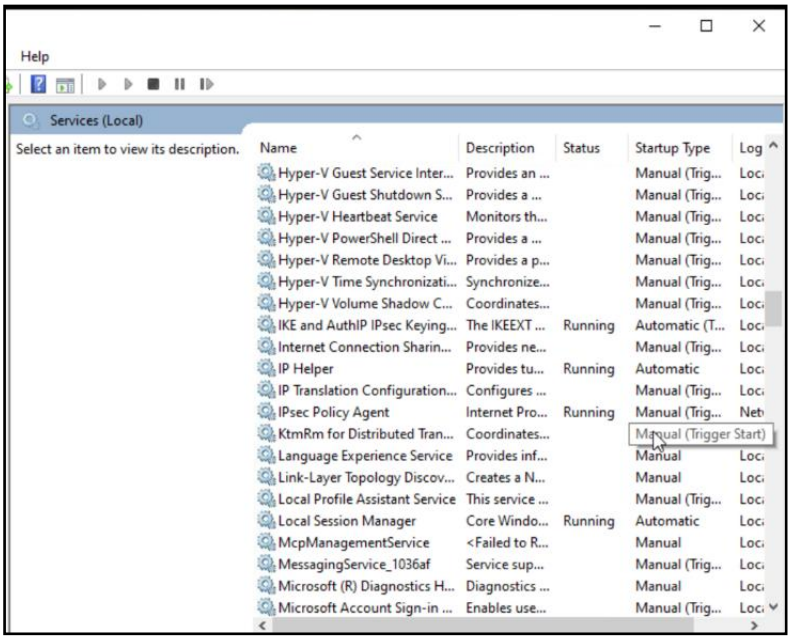

Figure 21: No INA+ Service created

Upon inspecting the Services and Process Explorer, no service named "Intranet Network Awareness (INA+)" was found. The DLL failed to install the service correctly (Figure 20 and 21).

Conclusion Based on the strings and analysis using MiTeC EXE Explorer and Bintext, **sus-file3.dll** is specifically designed for Windows XP (version 6.11) and is a 32-bit executable. This explains why it failed to function as expected during dynamic analysis as it was being run in windows 10 64-bit machine—its behavior is likely tailored for Windows XP and may only work correctly on 32-bit systems (Figure 22 and 23).

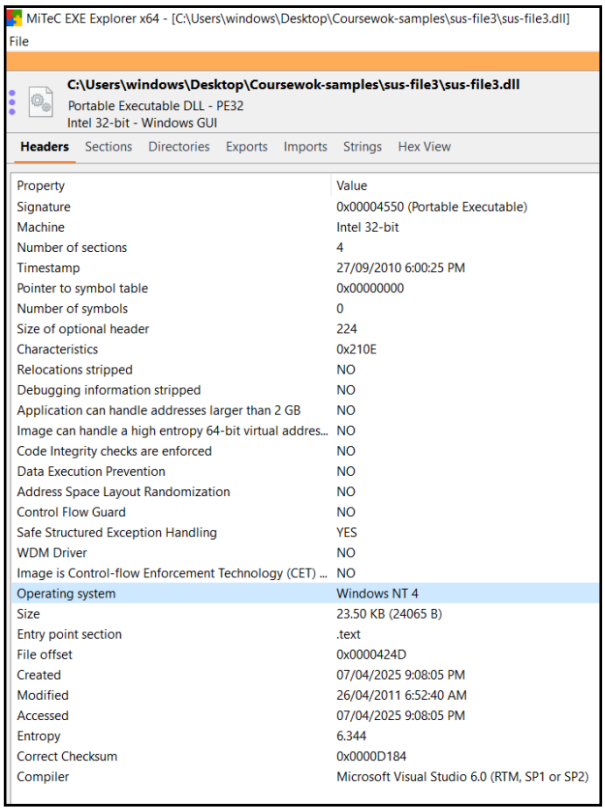

Figure 22: Proof works only for Windows NT 4

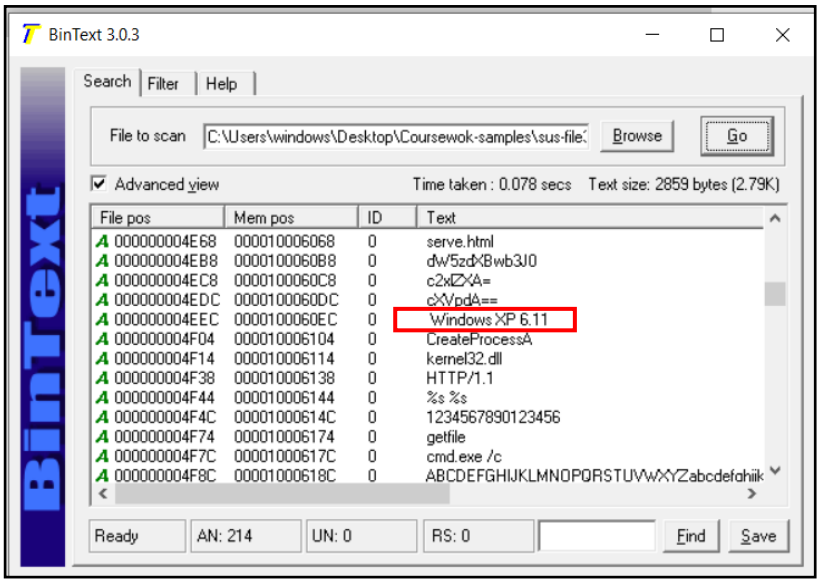

Figure 23: BinText string WindowsXP 6.11

PART 2

Question 1

The provided malware file, **sus-file4.exe**, was extracted from a ZIP archive and analysed using IDA Free. The goal is to identify encoding techniques, keys, and the functionality of the malware. Static analysis was performed first, focusing on identifying suspicious patterns.

During static analysis in IDA Free. XOR is a common method used by malware developers to obfuscate code or data, making it harder for analysts to reverse-engineer the malicious payload (Zalesskiy, 2024). IDA Free's search functionality was used to locate all occurrences of the XOR instruction in the binary.

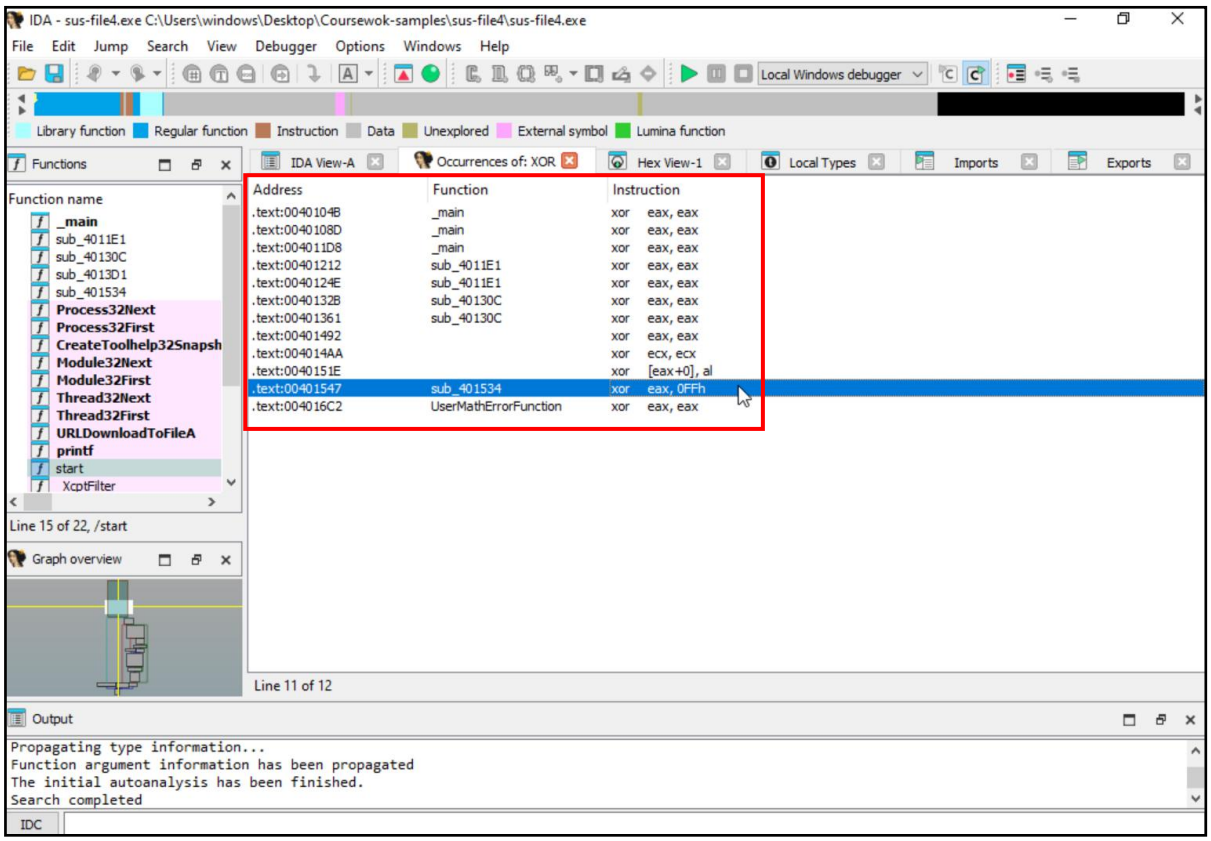

Figure 24: XOR called by function sub_401534

- Among the results, the function **sub_401534** the presence of **XOR [eax+0], al** and **XOR eax, 0FFh** suggests that the malware might be using XOR to encode or decode data. These patterns are often seen in malware where data is encrypted or obfuscated using XOR-based algorithms.
- Upon further examination of the disassembled code, the function **sub_401534** was identified as a critical component of the malware's decoding mechanism. This function processes a buffer or array of bytes, applying an XOR operation with the key **0xFFh**.

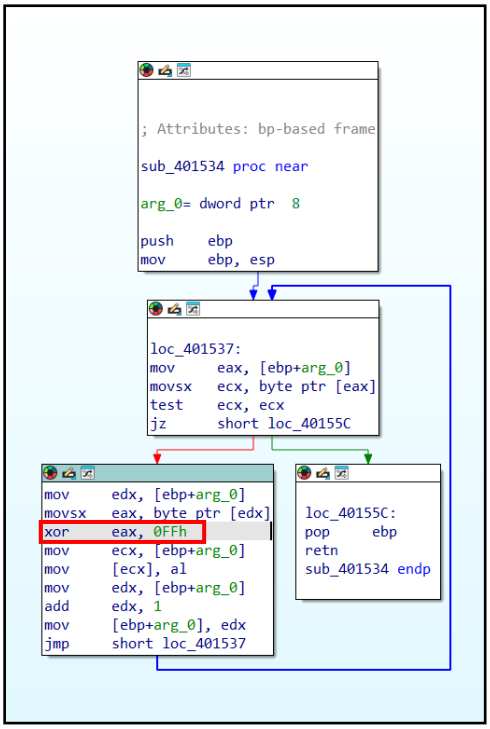

Figure 25: Detailed view what XOR is performing

- The function receives one input (arg_0), which is likely a pointer to a memory area holding some data.
- It uses a loop to go through each byte of the data stored at this memory location.
- Within the loop, the instruction **xor eax, 0FFh** applies a bitwise XOR operation to each byte.
- Most likely to be decoding characters in a loop

To understand the role of **sub_401534**, we first examine its cross-references (Xrefs) to determine where it is invoked within the binary. The analysis reveals that **sub_401534** is called twice:

1. At address **.text:004014EB**.
2. At address **.text:004014F8**.

These calls suggest that **sub_401534** is used to process different segments of encoded data during execution.

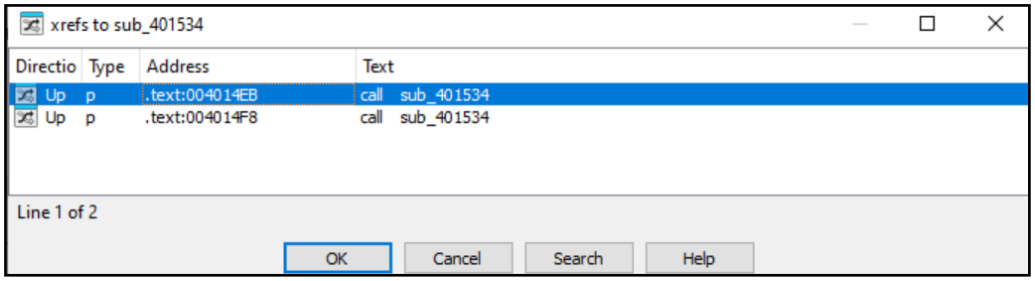

Figure 26: Two places where the sub_401534 is called

At address .text:004014EB, the malware pushes the memory address of unk_403010 onto the stack using the instruction push offset unk_403010. Immediately after, the function sub_401534 is called, indicating that unk_403010 is passed as an argument.

Similarly, at address .text:004014F8, the malware pushes the memory address of unk_403040 onto the stack using the instruction push offset unk_403040, followed by another call to sub_401534.

Hence unk_403010 and unk_403040 contain encoded data

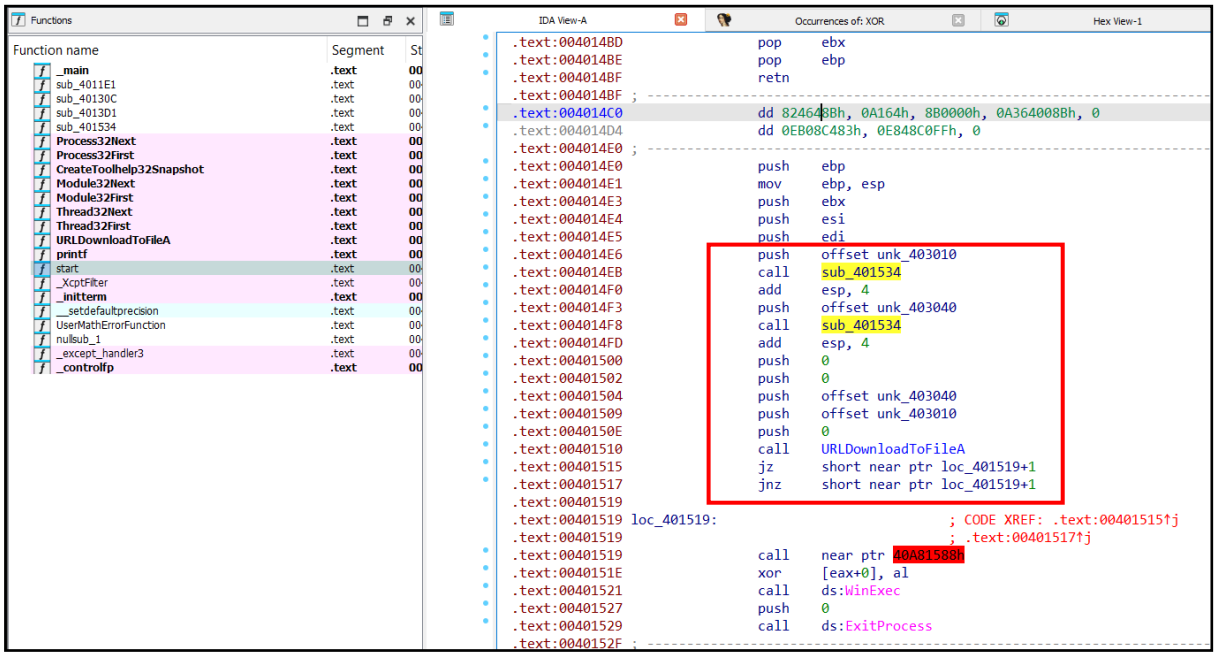

Figure 27: Encoded data at unk_403010 and unk_403040 passed as arguments to sub_401534, suggesting decoding or decryption logic

Analysing the First Call to sub_401534

At address .text:004014EB, the malware sends the address of unk_403010 to the sub_401534 function. This gives the function access to a buffer of data encoded with XOR using the key 0xFFh. The function then decodes the data by applying the XOR operation to each byte in the buffer.

Upon examining the contents of **unk_403010** (Figure 28), we observe a series of encoded bytes. By applying the XOR key **0xFFh** to each byte, we can decode the contents:

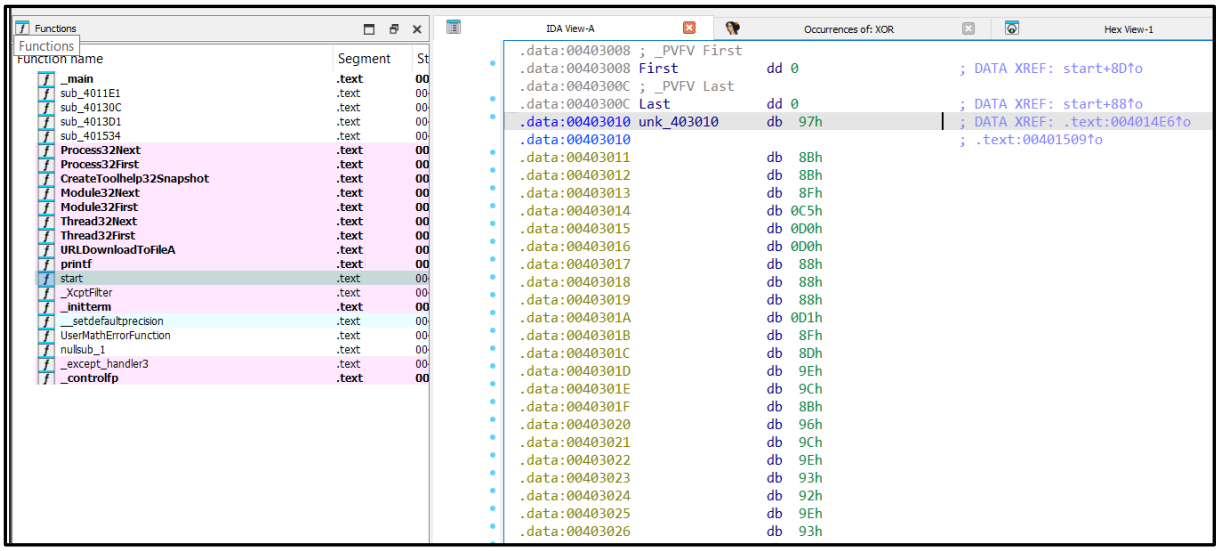

Figure 28: encoded data at unk_403010

- Encoded Bytes: **97h, 88h, 88h, 8Fh, 0C5h, 0D0h, 0D0h, 88h, …**
- **Decoded Value:** XORing each byte with 0xFFh reveals the decoded string:

http://www.practicalmalwareanalysis.com/tt.html

This URL represents the source from which the malware downloads its payload.

The second call to **sub_401534** occurs at address **.text:004014F8**. Like the first call, the malware pushes the memory address of **unk_403040** onto the stack:

Examining the contents of **unk_403040** (Figure 29), we see a series of encoded bytes. By applying the XOR key **0xFFh** to each byte, we can decode the contents:

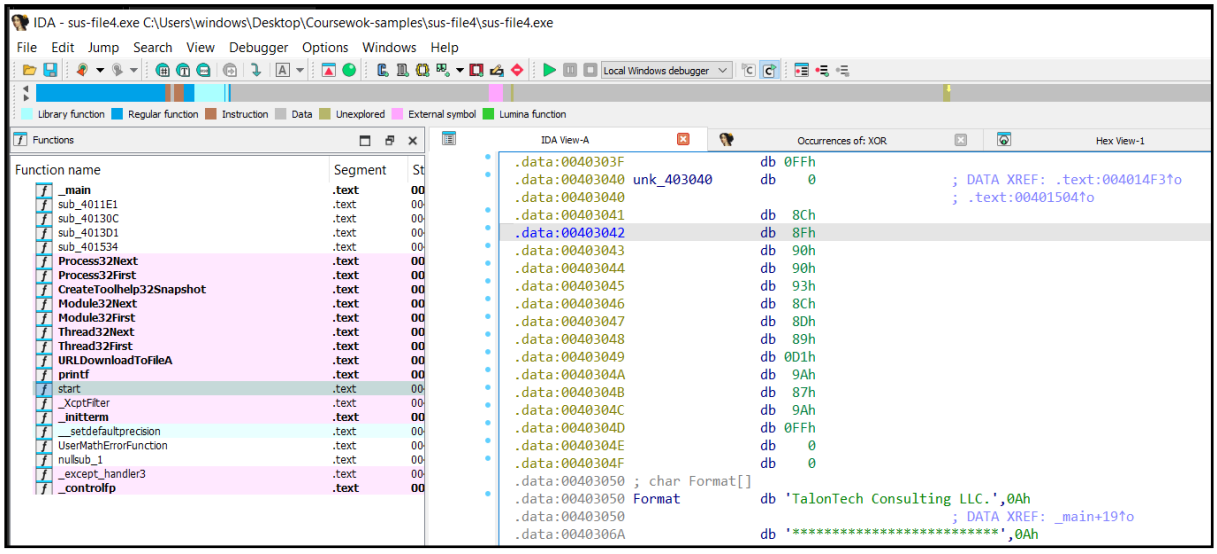

Figure 29: Encoded data at unk_403040

- Encoded Bytes: **8Ch,8Fh, 90h,90h, 93h, 8Ch, 89h, 8Dh, …**
- **Decoded Value:** XORing each byte with 0xFFh reveals the decoded string:

spoolsrv.exe

The malware uses the filename **spoolsrv.exe** to mimic a legitimate Windows process, thereby evading detection. This filename corresponds to the malicious executable downloaded by the malware.

Analysing the Second Call to sub_401534

After decoding both **unk_403010** (the URL) and **unk_403040** (the filename), the malware proceeds to download and execute the payload:

- The malware uses the URLDownloadToFileA function to retrieve the malicious executable (spoolsrv.exe) from the specified URL (http://www.practicalmalwareanalysis.com/tt.html).
- The decoded URL and filename are passed as arguments to this function, ensuring that the correct payload is downloaded.
- Once the file is downloaded, the malware executes it using the WinExec function.
- The decoded filename (spoolsrv.exe) is passed as an argument to WinExec, launching the malicious payload on the victim's system.

In conclusion the analysis of **sus-file4.exe** revealed XOR encoding with the key **0xFFh** as the primary obfuscation technique. Decoding the encoded data exposed its functionality: downloading and executing a malicious payload (**spoolsrv.exe**) from the URL **http://www.practicalmalwareanalysis.com/tt.html .**

Example of Decoding unk_403040

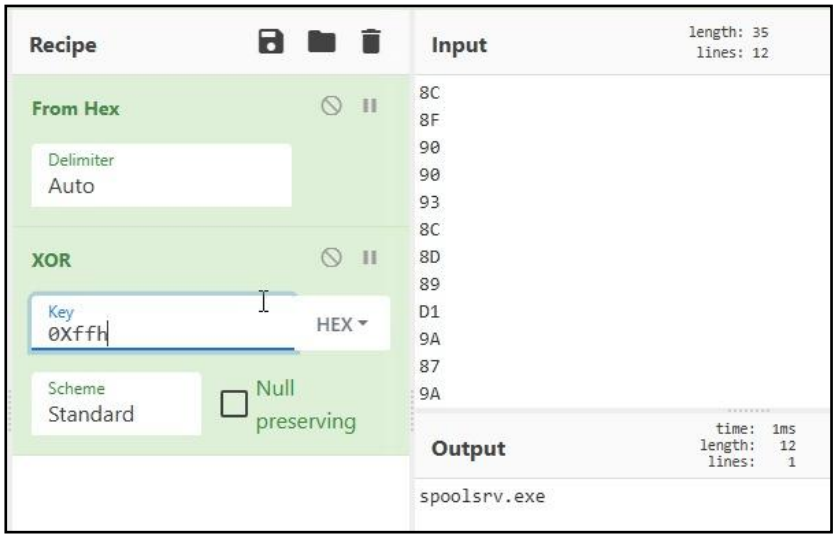

Figure 30: CyberChef decoded value for unk_403040

Question 2

The analysis starts with a file named sus-file5.zip, which contains an executable, sus-file5.exe.To determine the target environment for which the malware was compiled (either x64 or x86), the file was first analysed using Mitec EXE Explorer.

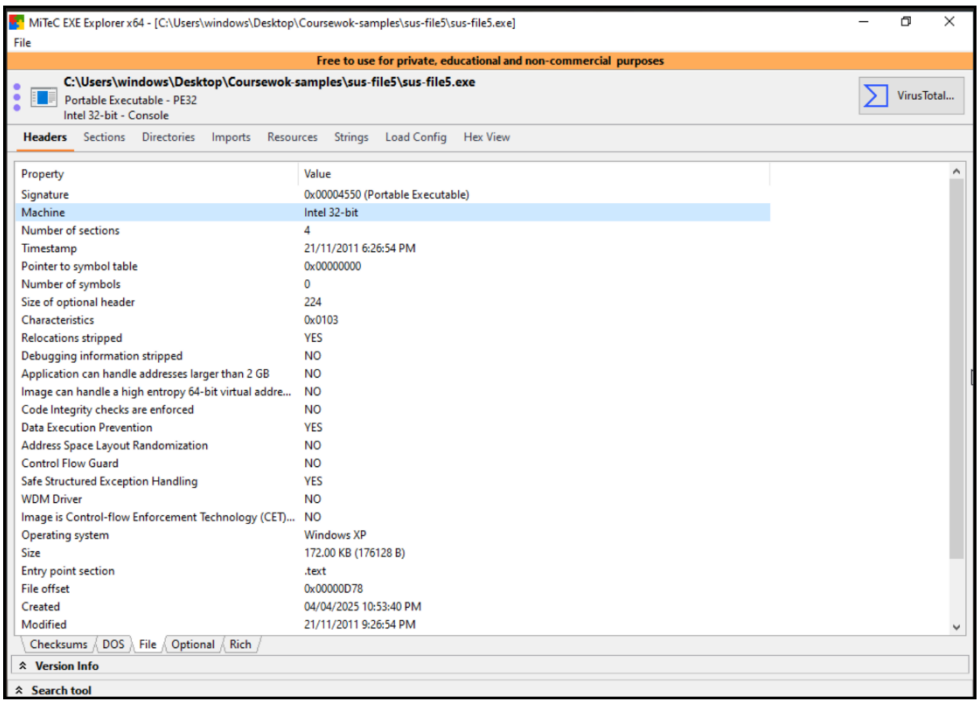

Figure 31: MiTeC showing sus-file5.exe is compiled for x86 (Intel 32-bit).

The Machine field shows Intel 32-bit which is x86 architecture, indicating that **sus-file5.exe was compiled for a x86 environment**, as confirmed by Mitec EXE Explorer.

Using Mitec EXE Explorer, the Resources section was checked and found to have three entries.

1. X64: Likely indicates a component or payload targeting the x64 (64-bit) environment.
2. X64DLL: Suggests a DLL (Dynamic Link Library) specifically designed for the x64 environment.
3. X86: Indicates a component or payload targeting the x86 (32-bit) environment.

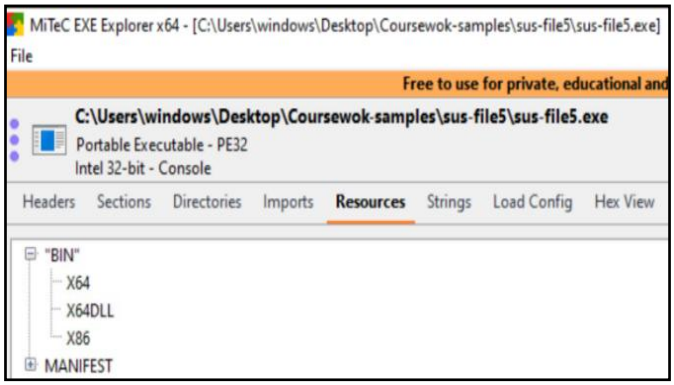

Figure 32: Resource section showing X86, X64, and X64DLL embedded files.

Now when the exe is imported to IDA free tool, we can observe few key things:

- The malware first loads kernel32.dll, Then the most important function is called **IsWow64Process** is a Windows API function that checks whether a 32-bit application is running on a 64-bit version of Windows.

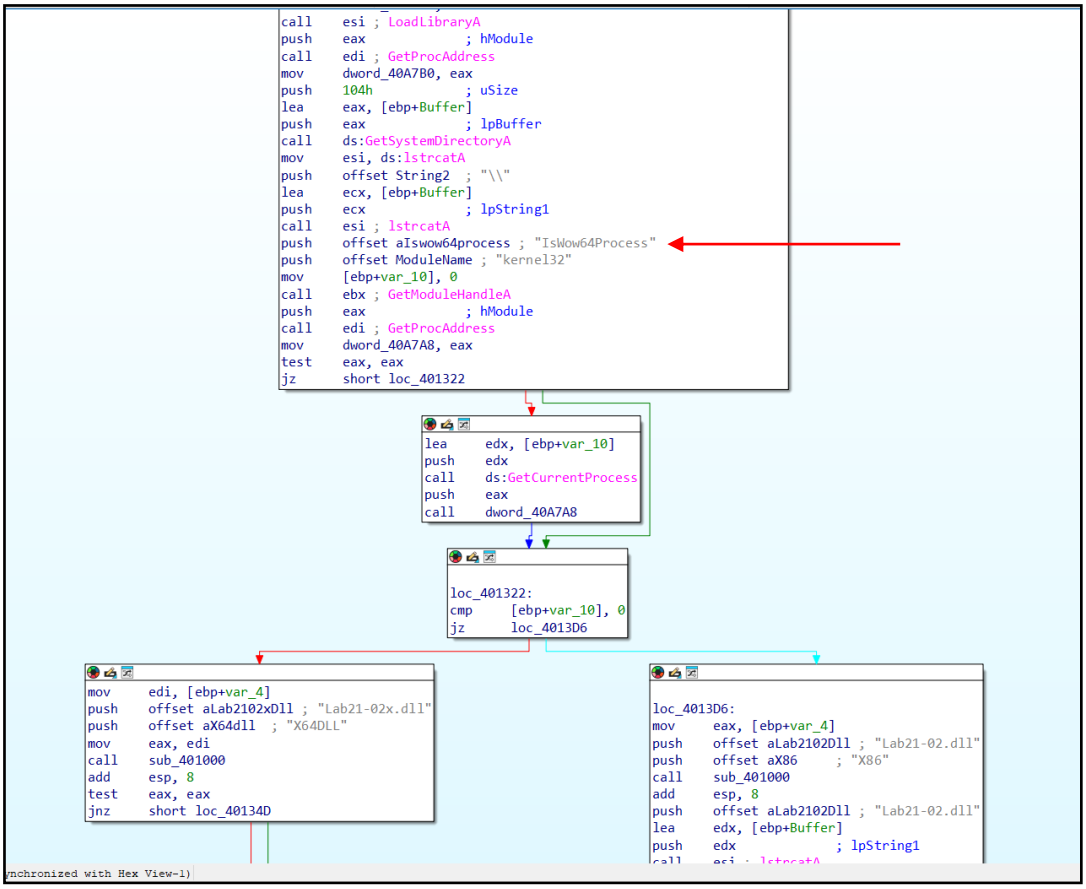

Figure 33: IDA view of IsWow64Process used to determine system architecture.

- If the malware is running under WOW64 (i.e., a 32-bit application on a 64-bit system), it follows the x64 branch. But if the program is running on a 32-bit machine it will follow the x86 branch

Behaviour in x86 (32-bit) Environment

In the x86 (32-bit) environment, the malware begins by determining the architecture of the host system using the IsWow64Process function. The result is stored in [ebp+var_10]. If the value is 0, it confirms the system is a native 32-bit environment (not running under WOW64). Execution then jumps to loc_40134D, which contains logic specifically designed for 32-bit systems.

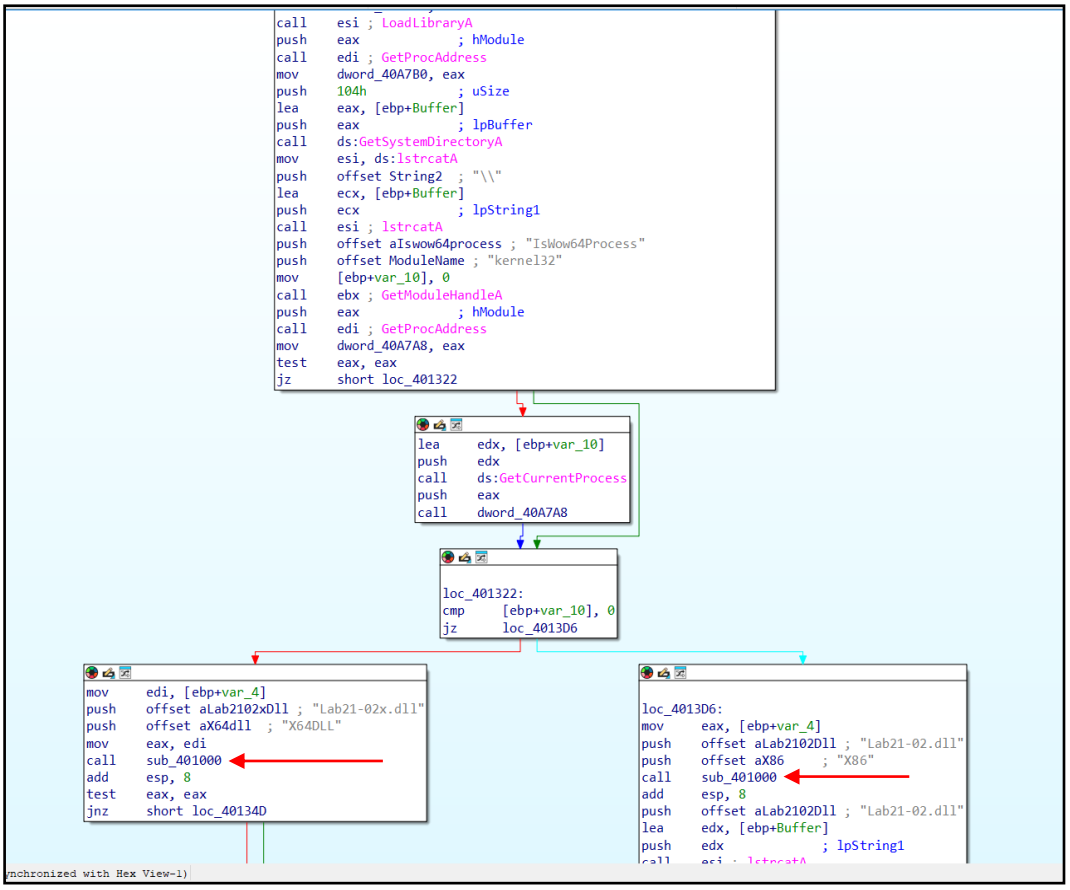

Figure 34: Conditional jump to x86 path if not running under WOW64

Role of sub_401000: This function is responsible for retrieving the "Lab21-02.dll" file from the binary's "X86" resource section and saving it to disk, specifically to C:\Windows\System32\. It uses standard Windows API functions like FindResourceA to locate the embedded file and VirtualAlloc to allocate memory for the extracted data (Figure 34).

Privilege Escalation with SeDebugPrivilege

Following the extraction of the DLL, the malware calls sub_401130 to elevate its privileges by enabling SeDebugPrivilege. This step is essential because it grants the malware permission to access and manipulate protected processes such as explorer.exe.

This subroutine works by invoking a chain of Windows API functions. First, it obtains a handle to its own process token using OpenProcessToken. Then, it calls LookupPrivilegeValueA to retrieve the locally unique identifier (LUID) for SeDebugPrivilege. Finally, AdjustTokenPrivileges is used to enable this privilege. This sequence of actions allows the malware to bypass standard user-level restrictions and interact directly with system processes (Figure 36).

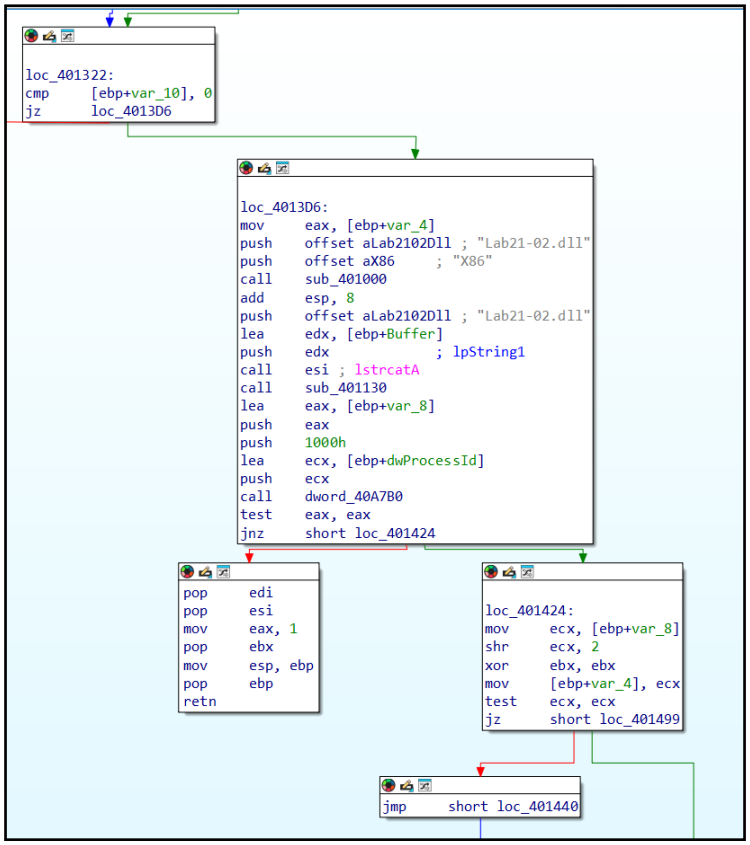

Figure 35: Entry into the x86-specific execution path. The malware extracts Lab21-02.dll from the "X86" resource and builds the full path for later injection.

Figure 36: Subroutine sub_401130 enabling SeDebugPrivilege using OpenProcessToken, LookupPrivilegeValueA, and AdjustTokenPrivileges

Process Enumeration and Target Identification

With elevated privileges enabled, the malware proceeds to locate its target process: explorer.exe. This is performed by calling a routine (likely sub_4011A0) that begins by enumerating all running processes using EnumProcesses. For each process ID retrieved, the malware opens the process and attempts to identify it by name.

To match the process name, the malware uses a case-insensitive comparison (strnicmp) against the hardcoded string explorer.exe. When a match is found, the corresponding process ID is retained for the injection stage (Figure 37).

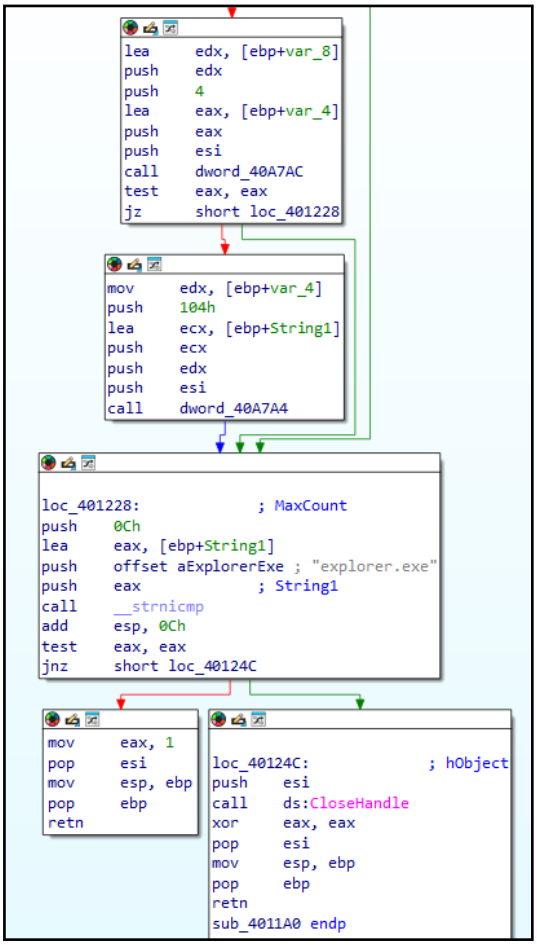

Figure 37: Process enumeration and case-insensitive comparison using strnicmp to identify the explorer.exe process as the target for injection.

DLL Injection into explorer.exe

Once the target process (explorer.exe) is identified, the malware initiates the injection phase. First, it calls OpenProcess with appropriate access rights to gain a handle to the process. After successfully opening the target, it uses VirtualAllocEx to allocate memory within the address space of explorer.exe.

The full path to the extracted DLL (C:\Windows\System32\Lab21-02.dll) is then written into the allocated memory using WriteProcessMemory. Finally, the malware creates a remote thread in the target process using CreateRemoteThread, passing LoadLibraryA as the thread start address

and the DLL path as its parameter. This results in the DLL being loaded and executed within the context of explorer.exe.

This technique allows the malware to run its payload inside a trusted system process, helping it evade detection and maintain persistence (Figure 38).

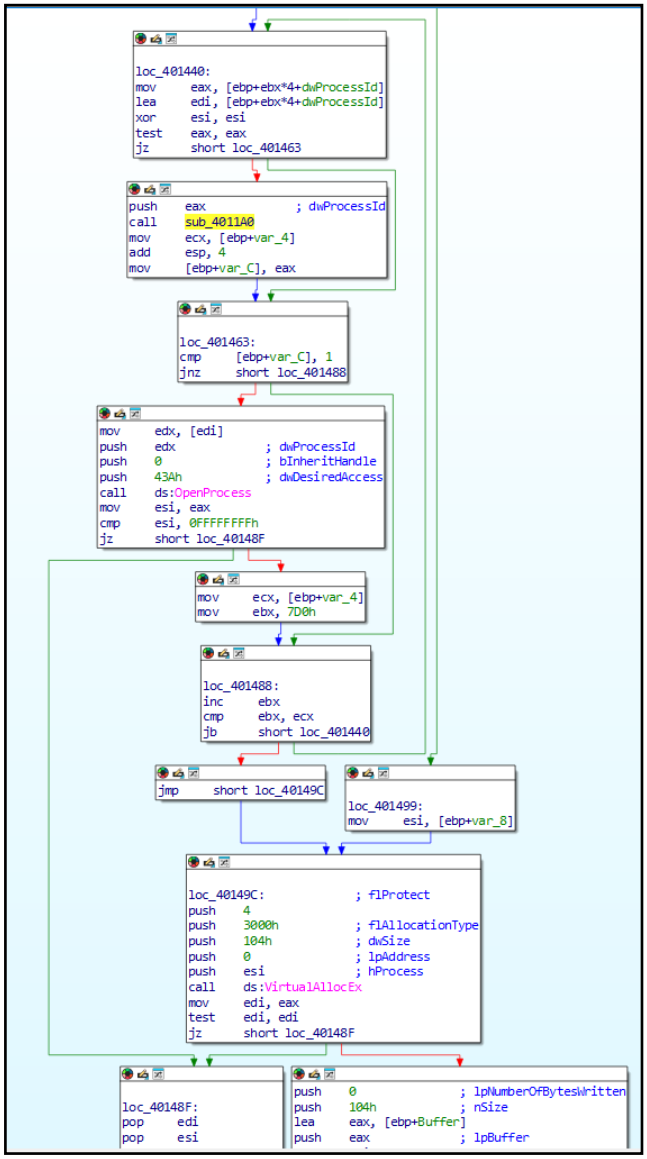

Figure 38: DLL injection steps using OpenProcess, VirtualAllocEx, and CreateRemoteThread

Behaviour in x64 Environment

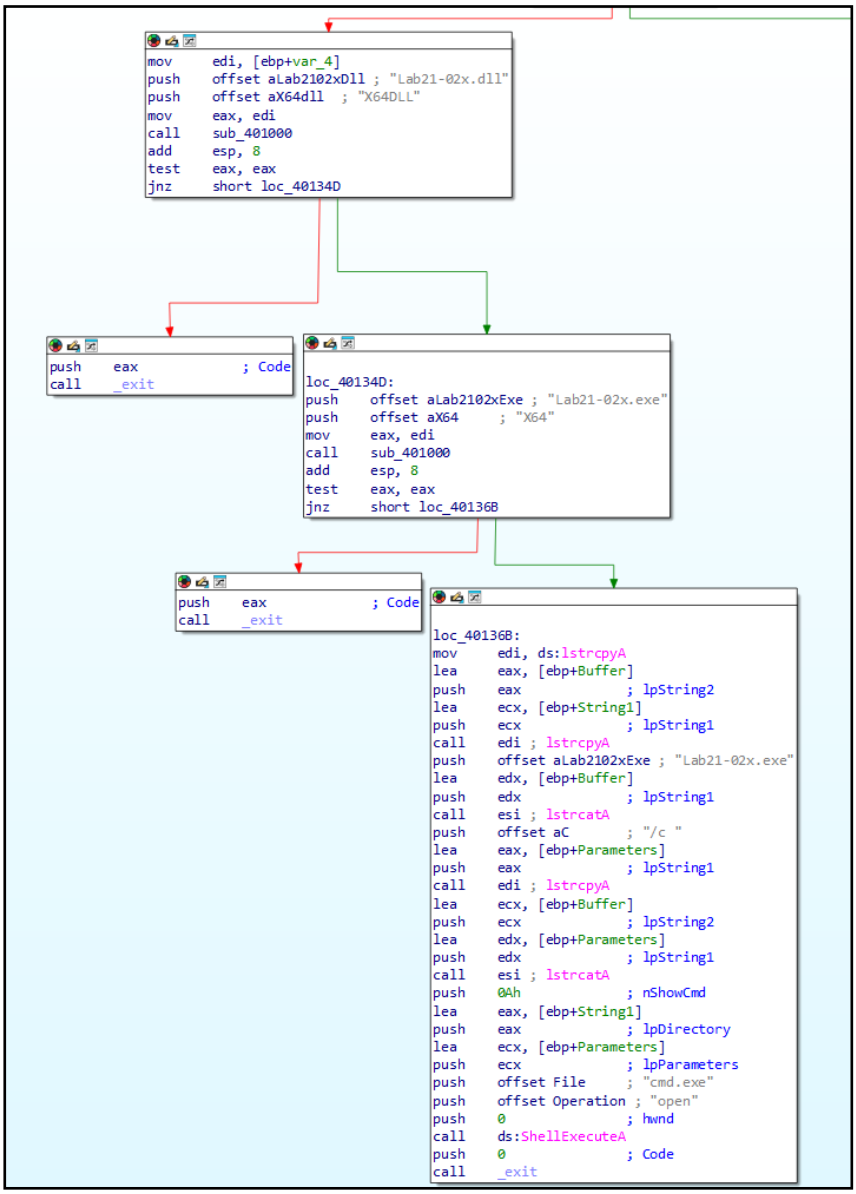

Figure 39: Extraction of Lab21-02x.dll and Lab21-02x.exe for execution in a 64-bit environment.

1. The function sub_401000 is called twice to extract two resources from the malware's binary: Lab21-02x.dll and Lab21-02x.exe. Each call uses specific identifiers ("X64DLL" for the DLL and "X64" for the executable) to locate and extract the files, which are saved in the C:\Windows\SysWOW64\ directory.

2. The malware constructs the DLL's full path (C:\Windows\SysWOW64\Lab21-02x.dll) using lstrcpyA and lstrcatA. First, lstrcpyA copies the base directory (C:\Windows\SysWOW64\) into a buffer at [ebp+Buffer]. Then, lstrcatA appends the DLL name (Lab21-02x.dll) to this buffer. This process ensures the malware dynamically builds the correct file path for referencing the extracted DLL during the injection process.

3. The malware executes Lab21-02x.exe using ShellExecuteA. It constructs the command 'cmd.exe /c Lab21-02x.exe', where 'cmd.exe' runs the Windows command interpreter, '/c' ensures the command is executed and then terminated, and 'Lab21-02x.exe' is the target file. The operation parameter is set to "open", instructing Windows to launch the executable in the system environment.

4. The malware dynamically resolves API functions using LoadLibraryA and GetProcAddress to load libraries (e.g., kernel32.dll) and retrieve function addresses (e.g., IsWow64Process, CreateRemoteThread). This avoids static imports, evading detection and ensuring compatibility. Functions like VirtualAllocEx and WriteProcessMemory are also resolved for DLL injection.

5. The malware injects its DLL into explorer.exe by first enumerating processes with EnumProcesses and identifying explorer.exe by its name. It then allocates memory in the target process using VirtualAllocEx, writes the DLL path with WriteProcessMemory, and creates a thread in explorer.exe using CreateRemoteThread. This thread calls LoadLibraryA to load the malicious DLL (Lab21-02x.dll) into the process's address space, allowing the malware to execute code within a trusted system process.

The malware extracts Lab21-02x.dll and Lab21-02x.exe for x64, stored in C:\Windows\SysWOW64\. It resolves APIs dynamically to evade detection and injects the DLL into explorer.exe using VirtualAllocEx and CreateRemoteThread, ensuring stealthy execution in 64-bit environments.

Final Comparison

Category	**x86 Environment**	**x64 Environment (WOW64 Detected)**
Initial Architecture Check	IsWow64Process returns false → follows x86 path	IsWow64Process returns true → follows x64 path
Extracted Files	Extracts Lab21-02.dll from "X86" resource	Extracts Lab21-02x.dll and Lab21-02x.exe from "X64DLL" and "X64" resources
Drop Location	C:\Windows\System32\Lab21-02.dll	C:\Windows\SysWOW64\Lab21-02x.dll and Lab21-02x.exe
Execution Flow	Direct DLL injection performed by main executable	Executes Lab21-02x.exe via ShellExecuteA to perform injection
Injection Method	Uses OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread	Same APIs used

Category	x86 Environment	x64 Environment (WOW64 Detected)
DLL Architecture	Injects 32-bit Lab21-02.dll	Injects 64-bit Lab21-02x.dll
Privilege Escalation	Enables SeDebugPrivilege to access system processes	Same method applied
String Assembly	Not required – DLL path known directly	Builds full DLL path using lstrcpyA + lstrcatA before writing to process memory
Process Name Matching	Uses strnicmp to match explorer.exe	Uses another function and a custom comparison routine

Conclusion

The malware sus-file5.exe is compiled for x86 and adapts its execution based on the system architecture. It performs resource extraction, enables SeDebugPrivilege, and injects DLLs into explorer.exe. On x86 systems, it injects a 32-bit DLL directly, while on x64 systems, it launches a 64-bit executable to inject a 64-bit DLL, ensuring stealth and persistence across environments.

Question 3

The smash.c program is a basic password checker that validates user input against two predefined passwords: Cyb3rTech! and AA@Ahmed. If the input matches either one, access is granted; otherwise, access is denied. Its primary function is simple authentication based on hardcoded credentials.

Key Components of the Code

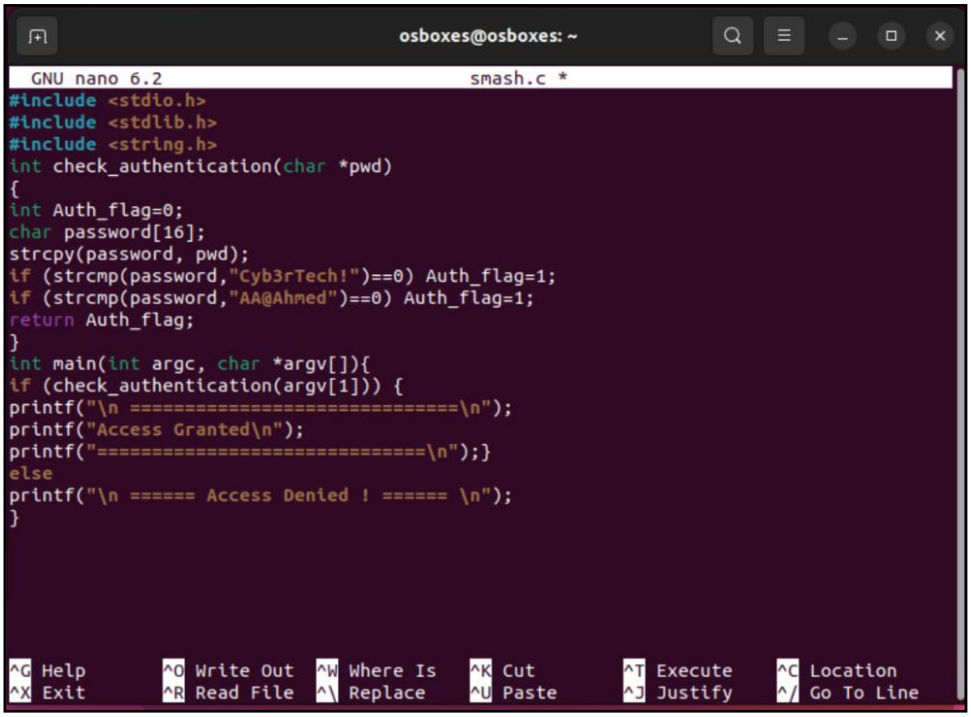

Figure 40

The **check_authentication** function takes a character pointer (**pwd**) as input and initializes a local variable **Auth_flag** to 0. Then char password[16] code declares a fixed-size array **password[16]** (It can only hold 16 characters) and uses **strcpy** to copy the contents of **pwd** into this buffer without ensuring the input size is safe, introducing a buffer overflow vulnerability. The function then compares the copied password with two valid passwords using **strcmp**, setting **Auth_flag** to 1 if a match is found. Finally, it returns the value of **Auth_flag**. In the **main** function, the program accepts command-line arguments and passes the first argument (**argv[1]**) to **check_authentication**. Based on the return value, it prints either "Access Granted" or "Access Denied". This design makes the program susceptible to exploitation through oversized input.

Conclusion

- The vulnerable part of the code is line 7, where **strcpy** is used to copy user input into the fixed-size **password[16]** buffer without ensuring the input length is within safe limits, enabling potential stack buffer overflow attacks.

Observations from Execution

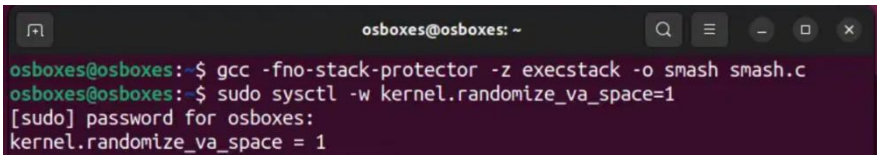

Figure 41

Pre-requisite before executing

gcc -fno-stack-protector -z execstack -o smash smash.c

1. **-fno-stack-protector**: Disables the stack protector feature, which is a security mechanism that adds a special value (known as a guard value) to the stack to detect and prevent buffer overflow attacks. By disabling this, the program becomes more vulnerable to stack-based overflows.
2. **-z execstack**: Marks the stack as executable, allowing any code written to the stack (e.g., via a buffer overflow) to be executed. This is typically disabled by default for security reasons, but enabling it makes exploitation of buffer overflows easier in this context.

To simplify the analysis while keeping some address randomization, the system's ASLR feature was temporarily set to partial mode using the following command:

sudo sysctl -w kernel.randomize_va_space=1

ASLR randomizes the memory addresses used by programs, making it harder to predict where specific data (e.g., the return address) resides. Setting it to partial mode allows for limited randomization during testing and debugging without fully disabling memory protection.

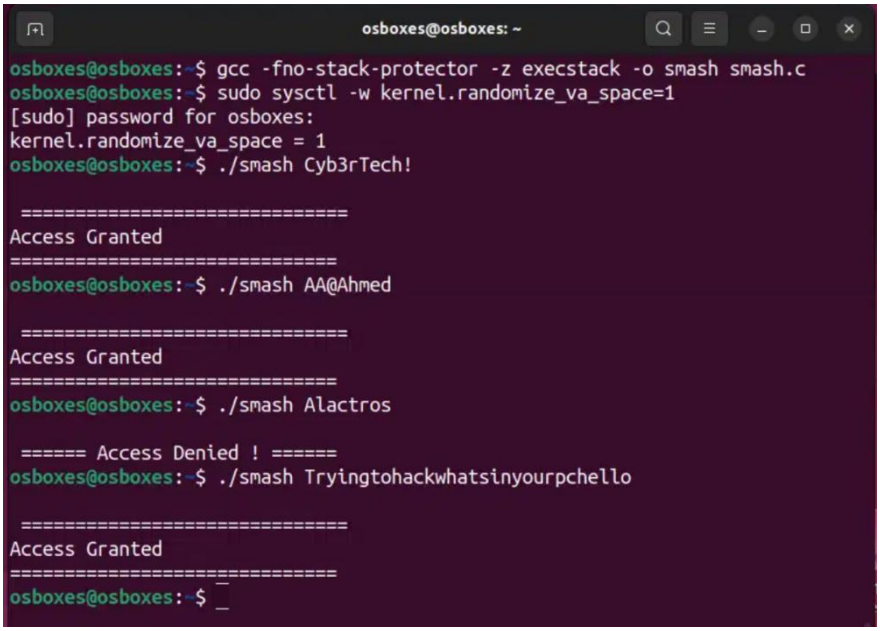

Figure 42

- When valid passwords like "Cyb3rTech!" or "AA@Ahmed" are entered, access is granted; otherwise, like with "Alactros," access is denied.
- Interestingly, providing an excessively long input (e.g., **"Tryingtohackwhatsinyourpchello"**) also results in "Access Granted." This behavior suggests that the program is vulnerable to buffer overflows, allowing unauthorized access under certain conditions.

Appendix

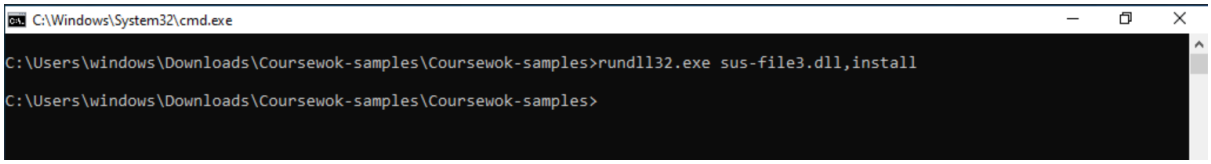

Figure 43: Rundll32.exe command being executed

Bibliography

Zalesskiy, J., 2024. *Understand Encryption in Malware: From Basics to XOR* . ANY.RUN. Available at: https://any.run/cybersecurity-blog/encryption-in-malware/ [Accessed: 30 March 2025].

Sikorski, M. and Honig, A., 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* . [ebook] Available at: https://ia600504.us.archive.org/25/items/humble_bundle_books/practicalmalwareanalysis.pdf [Accessed: 20 March 2025].

Balderson, K., 2024. *How To Run A DLL File: DLL File Open*. [online] Fortect. Available at: https://www.fortect.com/fix-dll-errors/run-dll-file/ [Accessed: 24 March 2025].