Malware Analysis and Incident Forensics (Ms Cybersecurity)
Systems and Enterprise Security (Ms Eng. in CS)
Practical test - 18/07/2023

First name: Last name: Enrollment num.:

Email:

Rules: You can use the textbook, written notes or anything "physical" you brought from home. You have full internet access that you can use to access online documentation. Communicating with other students or other people in ANY form, or receiving unduly help to complete the test, is considered cheating. Any student caught cheating will have their test canceled. To complete the test, **copy the following questions in a new Google Docs file and fill it in with your answers**. Please write your answer immediately after each question. Paste screenshots and code snippets to show whenever you think they can help comprehension. BEFORE the end of the test, produce a PDF and send it via e-mail to both querzoni@diag.uniroma1.it and delia@diag.uniroma1.it **with subject "MAIF-test-*<your surname>*-*<your enrollment number>*" (use the same pattern for the PDF file name)**.

Consider the sample named *sample-20230718*.exe and answer the following questions:

1 - What does a basic inspection of the PE file (e.g., header, sections, strings, resources) reveal about this sample?

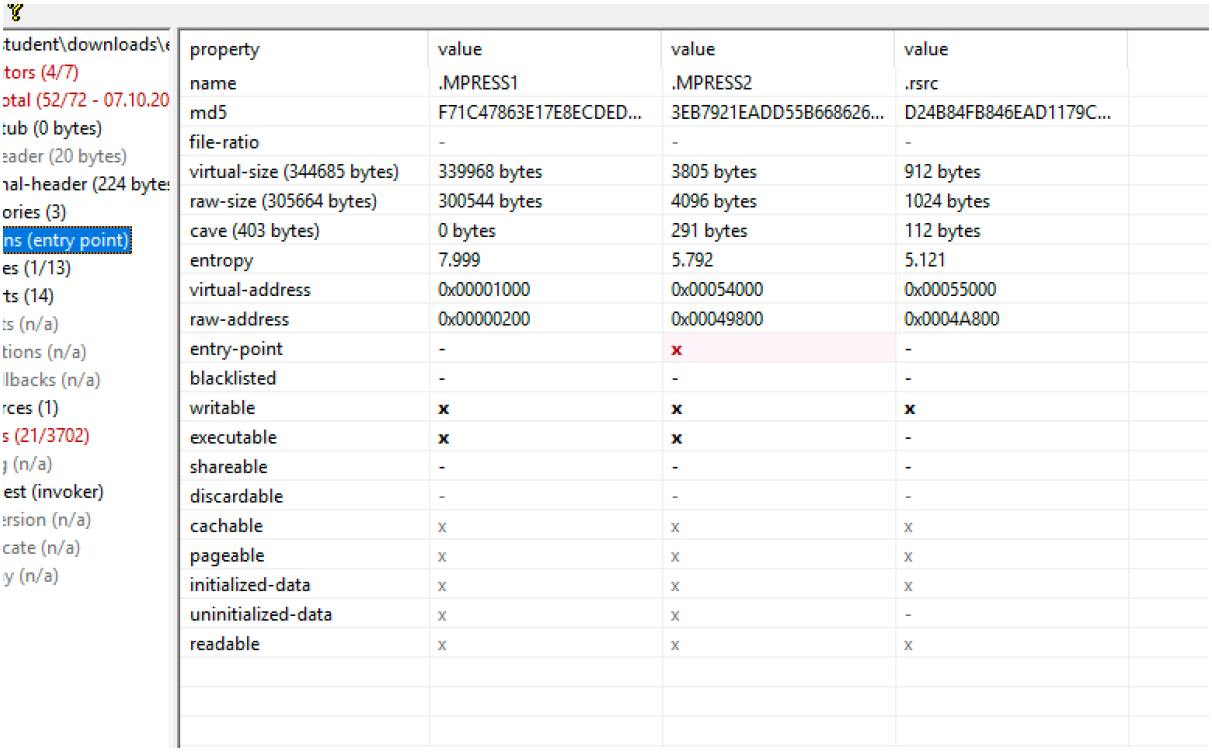
Examining the sample with **PEStudio**, we can retrieve from it many interesting things:

- Indicators of packing:
  - Sections names .MPRESS1, .MPRESS2
  - The entry point is not in the first section
  - Both sections have write and execute permissions
  - Few imports per library, but there is the presence of GetProcAddress and GetModuleHandle which are used to load and gain access to additional functions.
  - There are a lot of junk-like strings, maybe they are just compressed or obfuscated
  - High level of entropy in section .MPRESS1
  - Virtual size of the first section is larger than its raw size

Maybe since the entry point is in .MPRESS2, it will contains the decompression stub that will unpack the original executable at runtime and will store it in .MPRESS1

- The *imports* section contains some functions that are potential indicators or could reveal the sample's behavior. They are the following:
  - GetProcAddress, GetModuleHandle which are typical of packed software that has to rebuild the IAT
  - MessageBoxA, which means that the sample shows some message box
  - RegOpenKeyExA, which is probably used for opening the registry key and insert some value key for persistence purposes.
  - SHGetFolderPathA, which means that the malware interacts with the filesystem
- The strings section contains some strings that are potential indicators or that could reveal something about the behavior of the sample. They are the following:

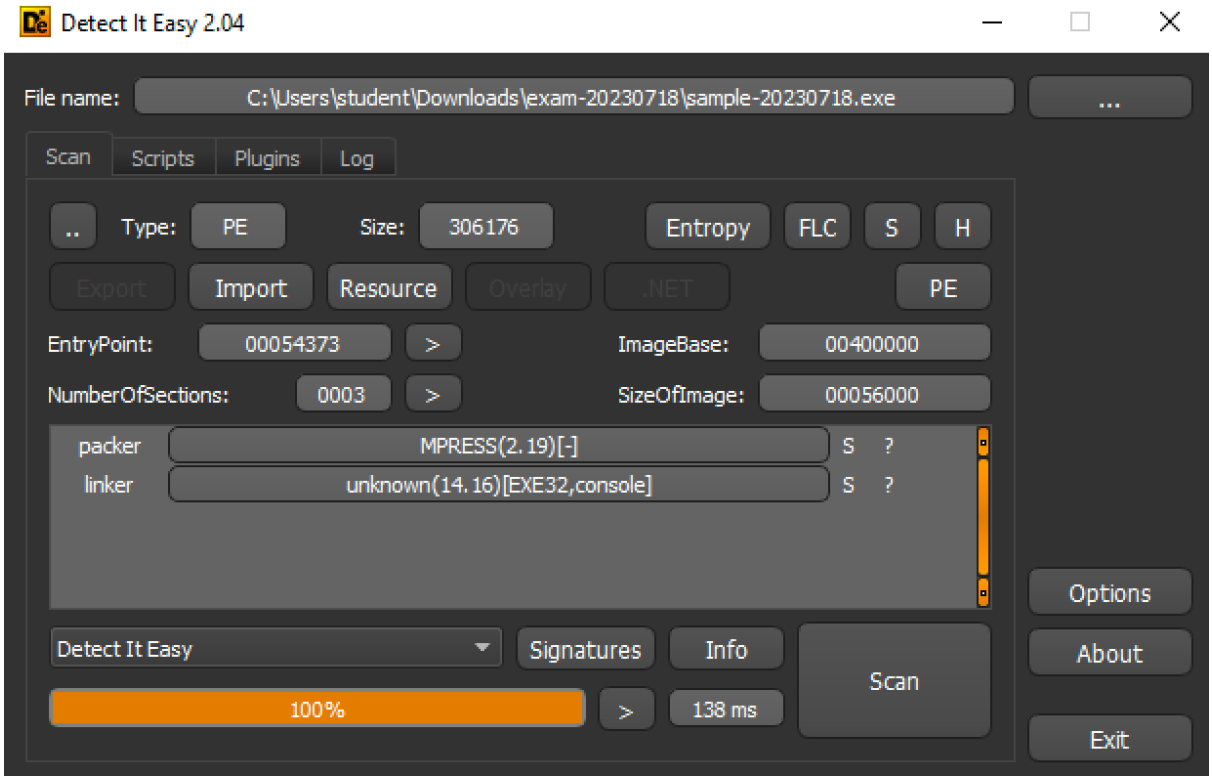
    - Function names (GetProcAddress, GetModuleHandle…)
    - Library names (NETAPI32.dll, kernel32.dll, shell32.dll, user32.dll, ws2_32.dll)
    - Extentions (.dll)
- The library section contains some libraries that are potential indicators or could reveal the sample's behavior. They are the following:
    - shell32.dll, which likely means that the sample interacts with other processes
    - ws2_32.dll, which likely means that the malware performs network activities
    - user32.dll, which likely means that the sample performs user-level interactions such as showing a message box



property	value	value	value
name	.MPRESS1	.MPRESS2	.rsrc
md5	F71C47863E17E8ECDED...	3EB7921EADD55B668626...	D24B84FB846EAD1179C...
file-ratio	-	-	-
virtual-size (344685 bytes)	339968 bytes	3805 bytes	912 bytes
raw-size (305664 bytes)	300544 bytes	4096 bytes	1024 bytes
cave (403 bytes)	0 bytes	291 bytes	112 bytes
entropy	7.999	5.792	5.121
virtual-address	0x00001000	0x00054000	0x00055000
raw-address	0x00000200	0x00049800	0x0004A800
entry-point	-	x	-
blacklisted	-	-	-
writable	x	x	x
executable	x	x	-
shareable	-	-	-
discardable	-	-	-
cachable	x	x	x
pageable	x	x	x
initialized-data	x	x	x
uninitialized-data	x	x	-
readable	x	x	x

2 - Which packer was used to pack this sample? Provide the original entry point (OEP) address, where the tail jump instruction is located, and detail how you identified them.

As section names suggested and as **Detect It Easy** confirmed, the sample was packed with MPRESS (version 2.19). Furthermore, after clicking “Entropy”, it confirms the packing.

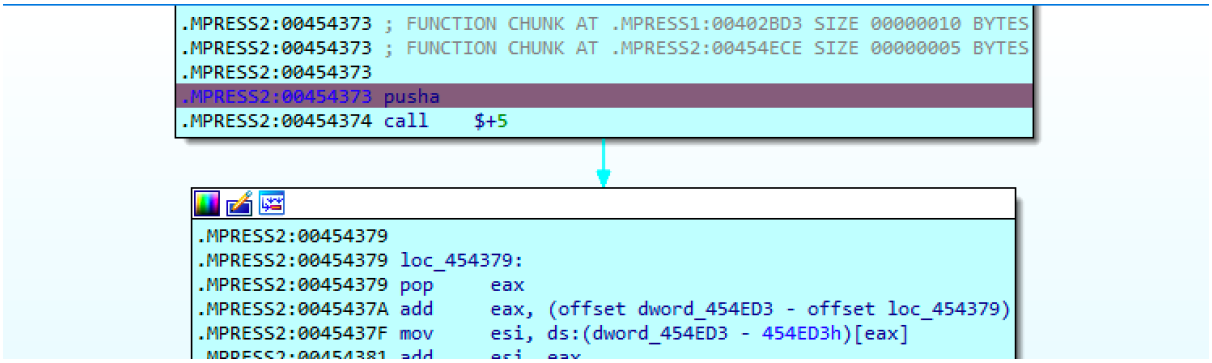


To find the OEP of a packed sample it's necessary to locate the tail jump, that is the jump that the packed sample performs to the beginning of the unpacked code after the unpacking stub has finished its operations.

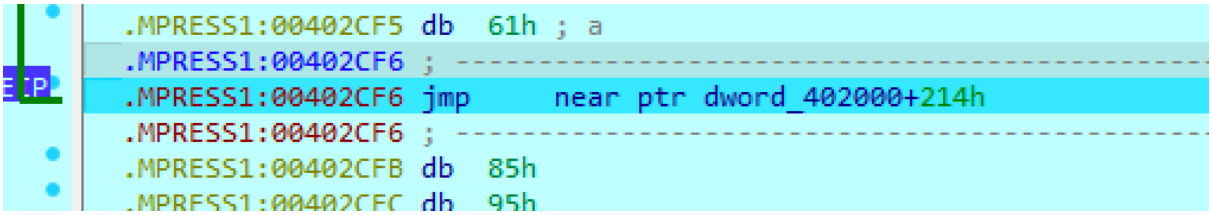
There are some indicators useful to recognize the tail jump that will allow us to fine the OEP:

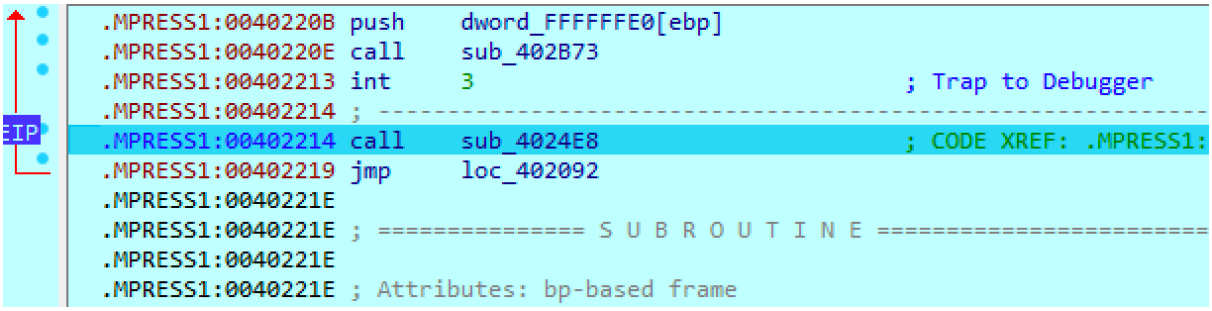
- The instruction jumps to another section (in this case from .MPRESS2 to .MPRESS1)
- After the tail jump should be a bunch of garbage bytes.
- The destination was previously modified by the unpacking stub

After opening the sample in IDA and starting at the entry point in .MPRESS2 (0x454373), the first instruction is a pusha, used to save the register values at startup. Most likely, there will be a corresponding popa instruction just before the tail jump.



There is a practical and reliable technique to identify the tail jump: place an HW breakpoint on memory access on the data pushed on the stack after the first pusha instruction. Before the jump there will be a popa instruction to restore the saved execution context.
Tail_jump @ 0x402CF6
OEP @ 0x402214

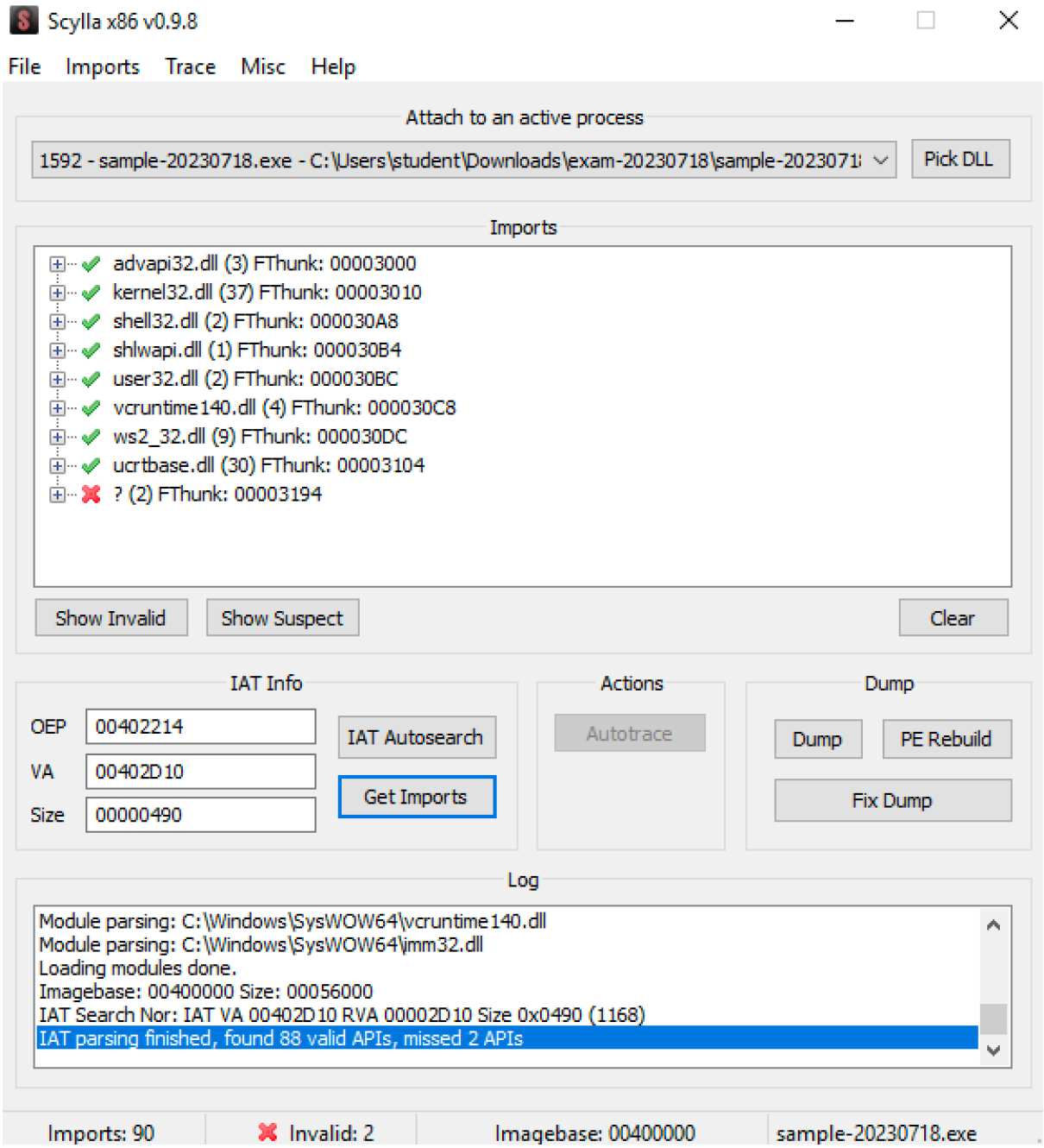




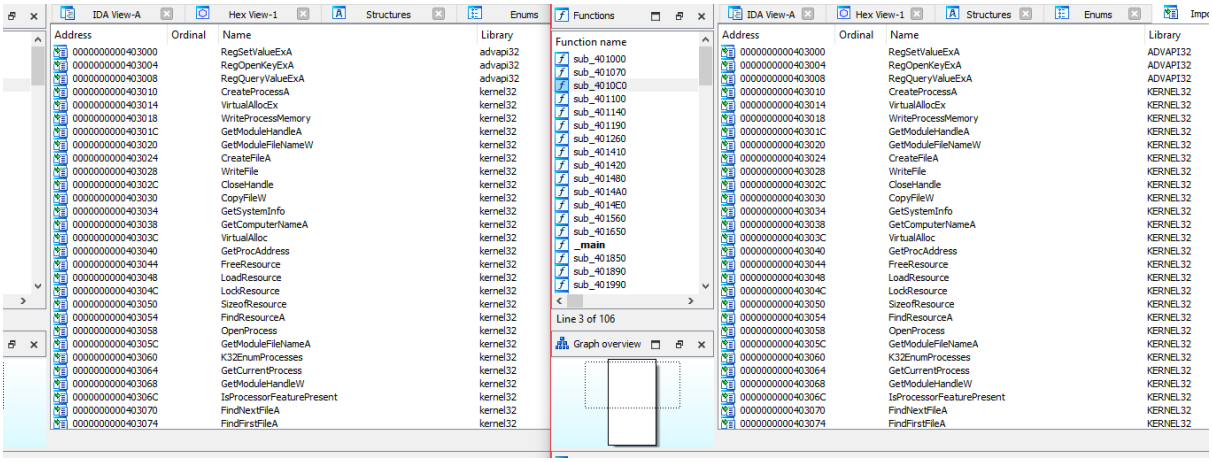
3 - Provide details about the IAT reconstruction process that you carried out to unpack the code. *HINTS: the answer should cover methodological aspects and facts on your output; also, validate it! (e.g., check API calls, compare with sample-20240710-unpacked.exe).*

Once the OEP is discovered, we can open **Scylla** to dump the binary

- Pressing **IAT Autosearch** we can obtain the IAT information starting from the OEP (0x402214). At this point Scylla retrieves its virtual address and the size;
- Then, with **Get import** we can retrieve the list of imports. There is an invalid entry, as we can see in the screenshot, that can be deleted.
- At this point, we have to click on Dump to dump the memory of the process (a file with the suffix_dump will be created).
- Finally, click Fix Dump loading the file created at step 3. A new file (with the suffix-SCY) is created, and it will contain the dump of the process with the reconstructed IAT.



- I compared the version of the sample unpacked by me with the already unpacked version provided for this exam. Using **IDA**, I inspected the imports performed by both versions. As can be seen in the following image, the imports are the same. In the image, on the left, there are the imports of the sample unpacked by me and on the right the imports performed by the already unpacked sample (i.e. sample-*20230718*-unpacked.exe).



4 - Provide a brief, high-level description of the functionalities implemented by the sample (what it does, when, how). Try to keep it short (like 10 lines). Reference answers to other questions wherever you see fit.

In general, the sample works as follows (for details see answer 6):

1. The sample checks if there is the presence of an antivirus. If so, it exits. It also checks that there is only one instance of the malware running in the system.
2. The sample copies itself in the startup folder, to achieve persistence on the victim Machine and calls the file pwned.exe.
3. The sample injects a shellcode in colorcpl.exe
4. The sample performs some network activities trying to connect to host 34.173.12.156 on port 80 and it waits the command.
5. The shellcode uses the network function to perform some network activities.

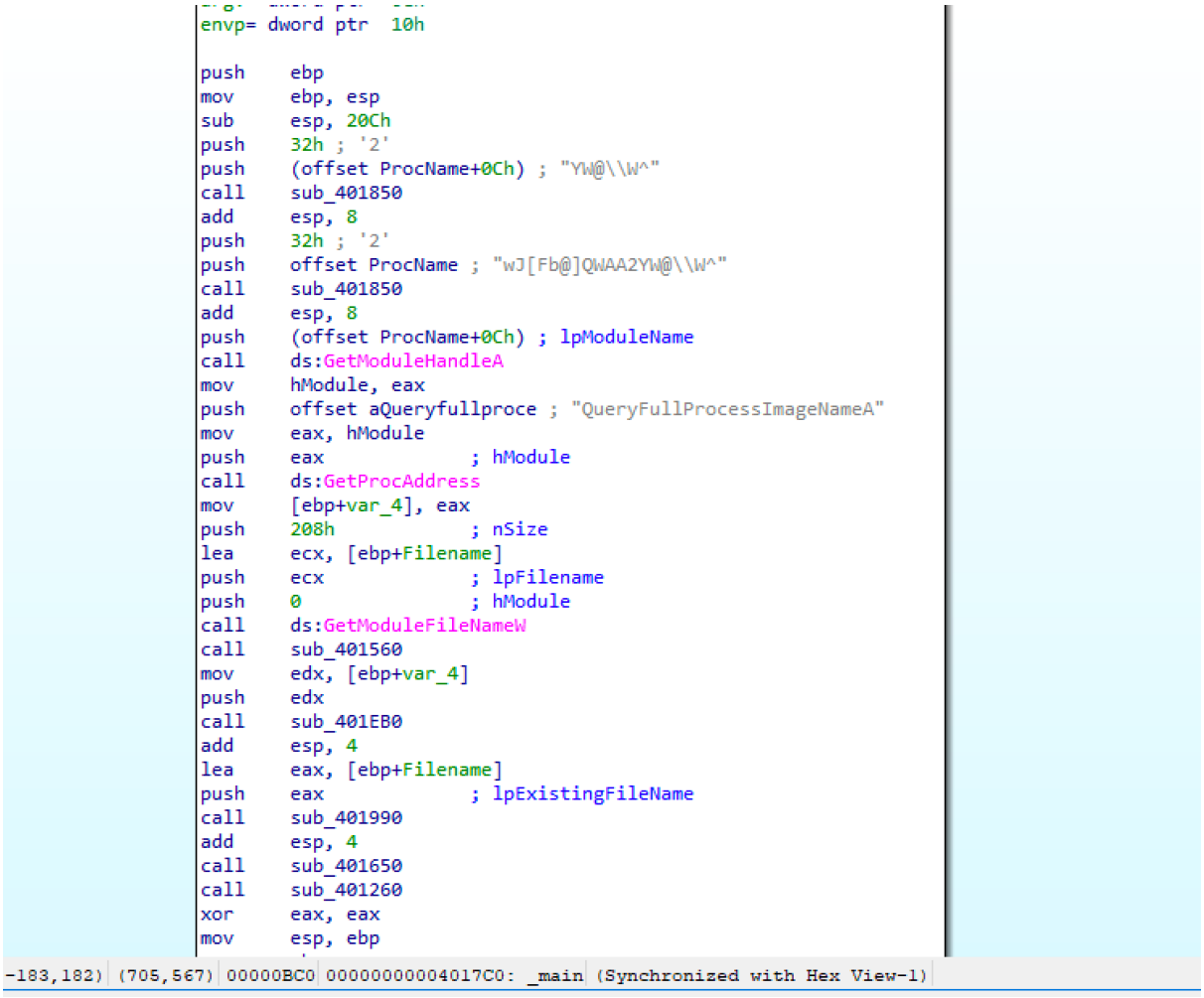
5 - List the processes, registry keys, files, and network connections created/manipulated by the sample and its byproducts (e.g., injected payloads, second-stage executables), if any, during their functioning. Detail the methodology you used to acquire this list. (Come back to this question to complete it as you acquire further details during the test).

Type	**Indicator**	**Description**	**Discovery method**
File	frieza-sama.jpg	Image created in a specific case and set as	IDA

		wallpaper	
Executable	pwned.exe	Copy of the malware created for persistence purposes	IDA
Process	colorcpl.exe	Victim process in which the sample injects the shellcode	IDA, process hacker
Registry key	HKEY_CURRENT_USER\Control Panel\Desktop	Inserts the registry key “TileWallpaper” and “WallpaperStyle” to set the image as a wallpaper	IDA, regedit
Network connection	34.173.12.156:80	Connection performs by the sample	IDA, process explorer
Network connection	10.0.2.15:49843	Connection performs by the shellcode	IDA, process explorer

6 - List the subroutines used by the sample and its byproducts (e.g., injected payloads, second-stage executables), if any, to implement its main functionalities and provide a sketch of the execution transfers among them (e.g sketch a tree/graph). **NOTE**: listing such parts is optional only in the case of shellcodes. *HINTS: Main code starts at **0xXXXXXXX**. Code at 0xXXXXXXX and higher addresses can be safely ignored.*

The **main** starts at 0x4017C0:



sub_401850 (decryption)

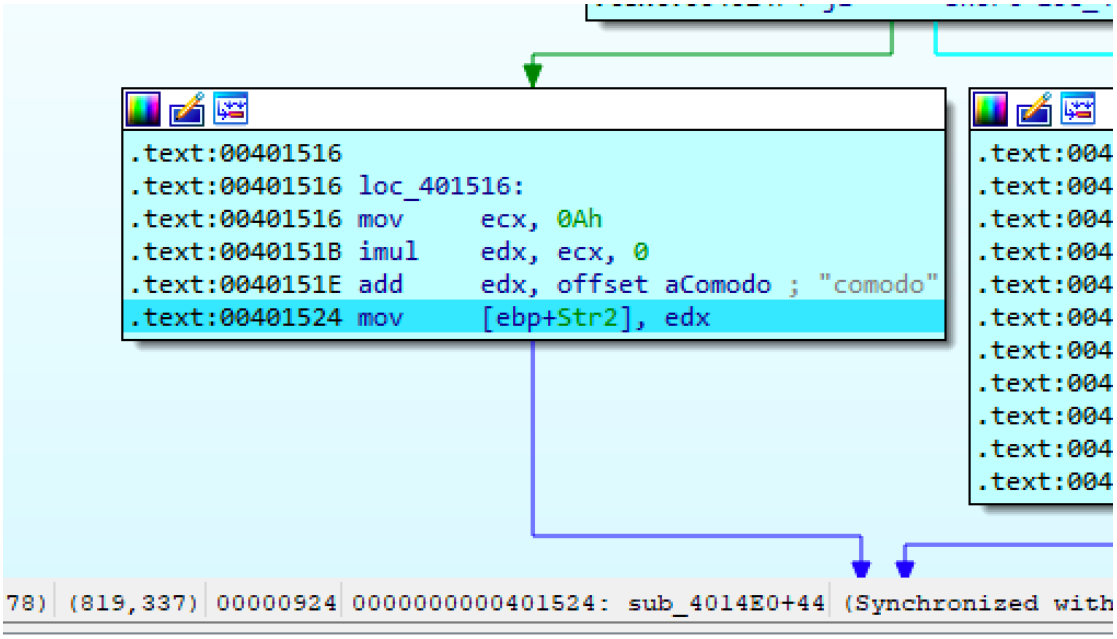
- The sample starts to decrypt some strange strings with:
  - “YW@\\W^”
  - “wJ[Fb@]QWAA2YW@\\W^”

  That are respectively:
  - Kernel
  - ExitProcess\x00kernel
- It calls GetModuleHandleA and GetProcAddress to retrieve the function QueryFullProcessImageNameA from kernel32.dll.
- Then it calls GetModuleFileNameW to retrieve the path of the executable file: C:\Users\student\Downloads\exam-20230718\sample-20230718-unpacked.exe.

sub_401560 (anti detection)

- In that function, it calls SHGetFolderPathA with csidl = 26h (program file folder) to retrieve the path: C:\Program File (x86)
- It calls FindFirstFileA and FindNextFileA to iterate the list of directory and file into program file.

- It calls the function sub_4014E0 that is used to check if there is the presence of: comodo, defender, intezer, Kaspersky, Sophos, webroot.



  - If there is at least one of these antiviruses, it calls ExitProcess.

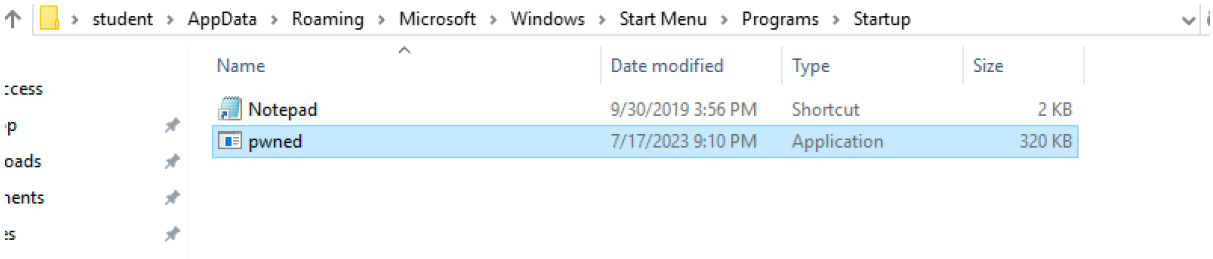
sub_401EB0 (check that there is only one instance)

- It calls PathFindFileNameA to take the path of the malicious code and takes the executable file (sample.exe).
- It calls K32EnumProcesses to iterate the processes in the system by ID. It uses this function to see if there already is an instance of the malware running in the system. If it is true, exit otherwise it calls OpenProcess in sub_401E30.

sub_401990 (persistence)

- It calls SHGetFolderPathW with csidl that is calculated with xor operation (xor ecx, 2Ch) and correspond to 0x07 (The file system directory that corresponds to the user's Startup program group). It uses this function to retrieve the path: C:\Users\student\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup.
- It checks if the file in the path: C:\Users\student\Downloads\exam-20230718\sample-20230718-unpacked.exe is in that directory. If it is false, the sample creates a copy of itself in the folder C:\Users\student\AppData\Roaming\Microsoft\Windows\Start

Menu\Programs\Startup and calls this file “pwned.exe”. In this way the malware can survive after reboot.

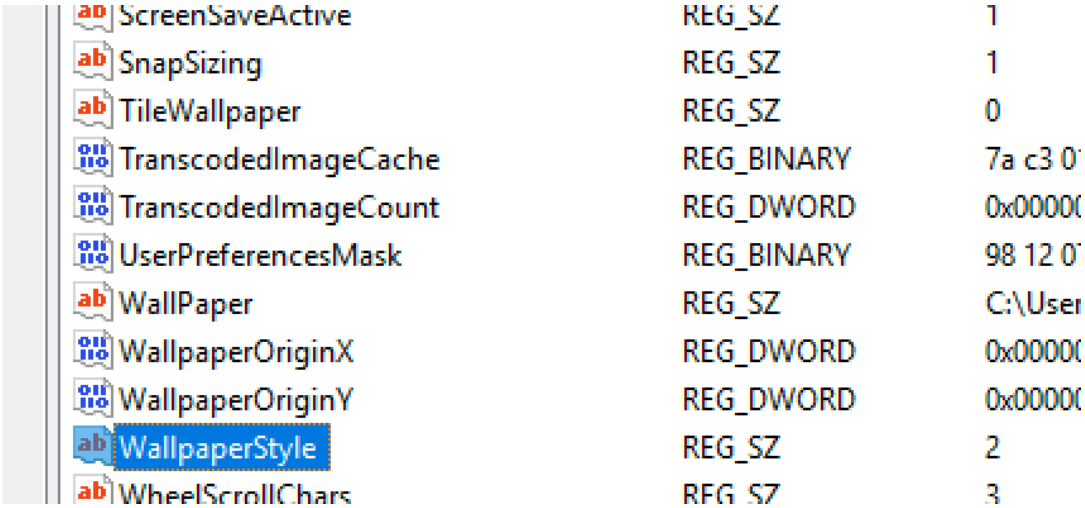


- Otherwise, it calls the function sub_401890. In that function, it calls SHGetFolderPathA with csidl = 27h (The file system directory that serves as a common repository for image files.) to retrieve the path: C:\Users\student\Pictures and it creates in that folder the file “frieza-sama.jpg” using CreateFileA and WriteFile.

```
.text:004018A8 call    ds:SHGetFolderPathA
.text:004018AE push    offset aFriezaSama ; "frieza-sama"
.text:004018B3 lea     ecx, [ebp+pszPath]
.text:004018B9 push    ecx             ; char
.text:004018BA push    offset aSSJpg   ; "%s\\%s.jpg"
.text:004018BF push    104h            ; BufferCount
.text:004018C4 lea     edx, [ebp+pszPath]
.text:004018CA push    edx             ; Buffer
.text:004018CB call    sub_401AF0
.text:004018D0 add     esp, 14h
.text:004018D3 push    0               ; hTemplateFile
.text:004018D5 push    80h             ; dwFlagsAndAttributes
.text:004018DA push    2               ; dwCreationDisposition
.text:004018DC push    0               ; lpSecurityAttributes
.text:004018DE push    0               ; dwShareMode
.text:004018E0 push    40000000h       ; dwDesiredAccess
.text:004018E5 lea     eax, [ebp+pszPath]
.text:004018EB push    eax             ; lpFileName
.text:004018EC call    ds:CreateFileA
.text:004018F2 mov     [ebp+hFile], eax
.text:004018F5 push    0               ; lpOverlapped
.text:004018F7 lea     ecx, [ebp+NumberOfBytesWritten]
.text:004018FA push    ecx             ; lpNumberOfBytesWritten
.text:004018FB push    4876Eh          ; nNumberOfBytesToWrite
.text:00401900 push    offset unk_4033A8 ; lpBuffer
.text:00401905 mov     edx, [ebp+hFile]
00000CAE 00000000004018AE: sub_401890+1E (Synchronized with EIP)
```

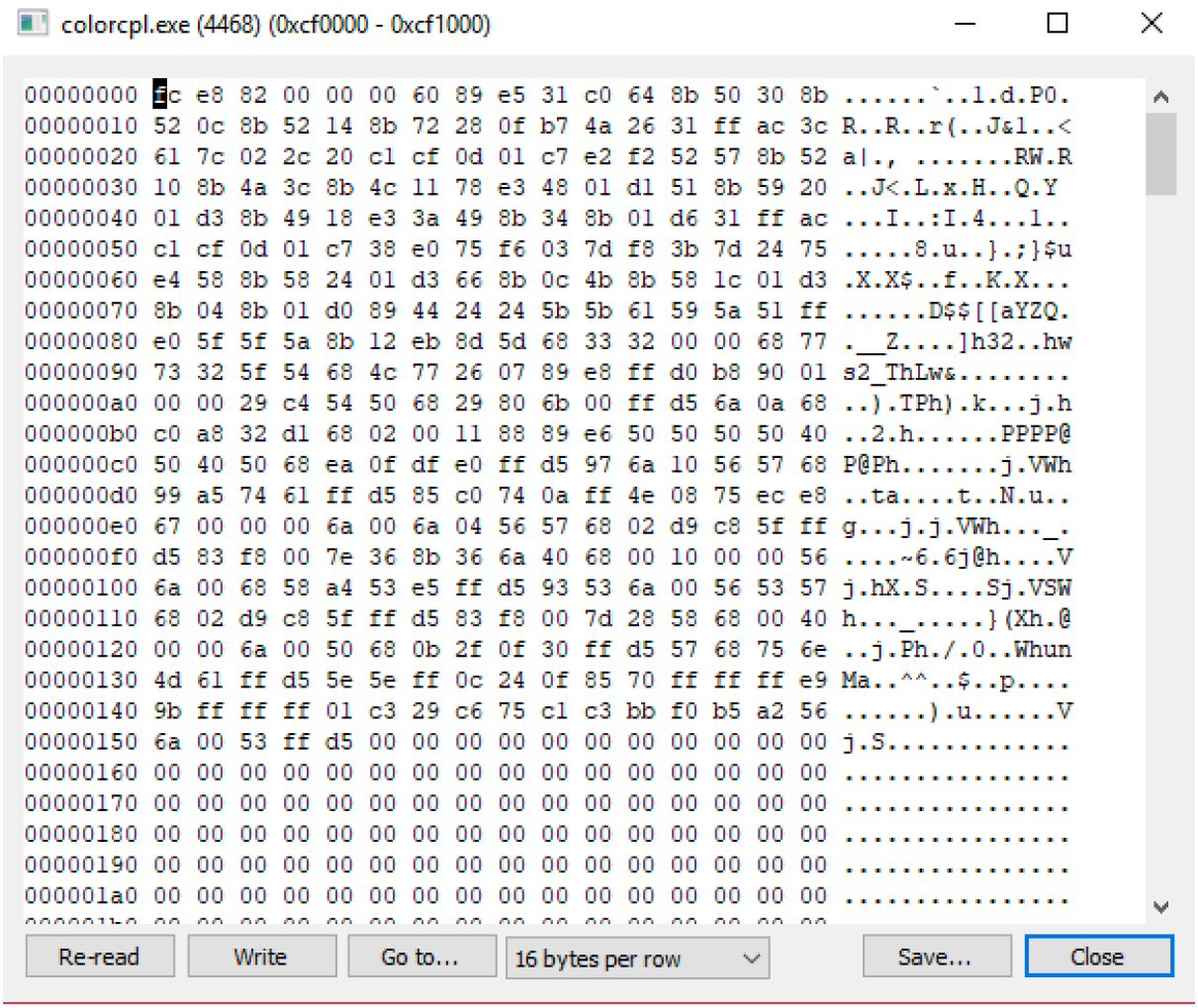
```
00 00  00 00 00 00 08 FB 19 00  ....).....
19 00  04 01 00 00 24 BB 44 00  ..@.....$.D.
44 00  43 3A 5C 55 73 65 72 73  .....D.C:\Users
6E 74  5C 50 69 63 74 75 72 65  \student\Picture
7A 61  2D 73 61 6D 61 2E 6A 70  s\frieza-sama.jp
63 00  88 FA 19 00 0B EE B4 73  g.c...c.....
```

- Then it calls RegOpenKeyExA with hkey = 80000001h (HKEY_CURRENT_USER) to open the registry key “Control Panel\Desktop” and set the key “TileWallpaper” and “WallpaperStyle” using RegSetValueExA.



sub_401650 (shellcode injection)

- It calls CreateProcessA with the parameter commandLine = “colorcpl.exe” to create the process “colorcpl.exe” in suspended state (dwCreationFlags = 4).
- It calls VirtualAllocEx to allocate 341 bytes of memory in that process.
- It calls WriteProcessMemory to write the malicious code in that memory at memory address 0xCF0000.



- Then it calls GetProcAddress to retrieve the function NtCreateThreadEx that is xor-encrypted (the key is 1Ah) to create a new thread of the process.

sub_401260 (C&C)

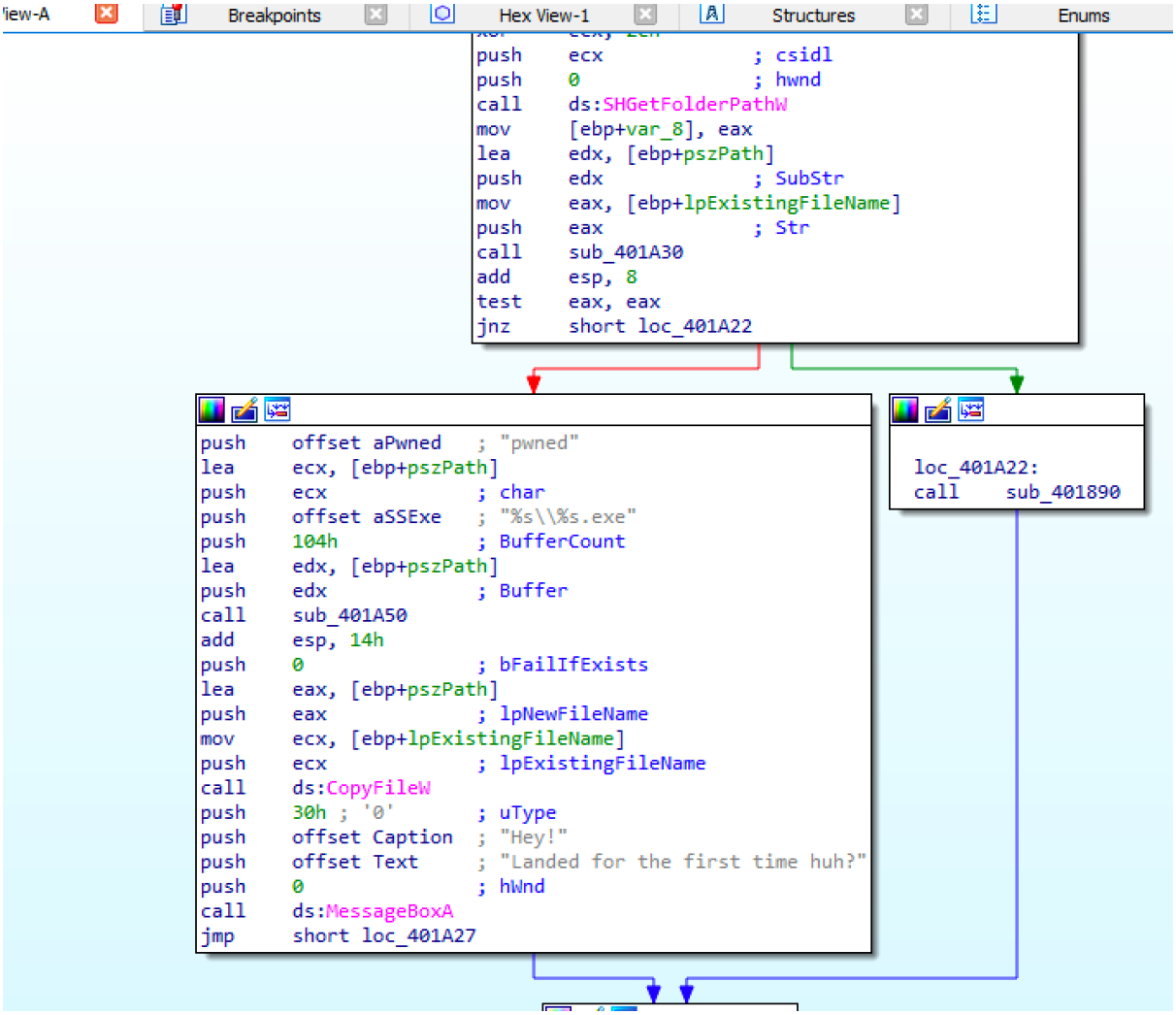
- It calls the function sub_401000 to call WSAStartup to retrieve the network functions.
- It tries to connect to host “34.173.12.156” on port 80. And if it is successful, it sends “RDY\n” to the server and waits for command.
- The commands are:
    - Q: it sends BYE\n to the server and close the connection
    - I: it opens the registry key Software\Microsoft\Cryptography using RegOpenKeyExA with hkey = 80000002h (HKEY_LOCAL_MACHINE) and it calls RegQueryValueExA to check if there is the value key “MachineGuid” and takes the corresponded data.
    - G: it calls GetComputerName and send the computer name in this form: Computer name: %s\n
    - H: it calls GetSystemInfo to retrieve the information of the system as number of processors, architecture (x64, x86, Wow64)
- The last 3 commands are in sub_401190

7 - Does the sample make queries about the surrounding environment before unveiling its activities? If yes, describe them and pinpoint specific instructions/functions in the code.

As show before in question 6, the sample, before performing its malicious activities, check if there is an antivirus in the program file (x86). It also checks that there is not another instance running in the system, to ensure that only one instance of the malware is running every execution.

8 - Does the sample include any persistence mechanisms? If yes, describe its details and reference specific instructions/functions in the code.

Persistence is achieved in this way: In the function **sub_401990**, the malware creates a copy of itself in the startup folder. In this way, the malware will be automatically run at the startup of the machine.



Details in answer 6.

9 - Does the sample perform any code injection activities? Which kind of injection pattern do you recognize? Describe the characteristics and behavior of the injected payload, stating also where it is originally stored within the sample.

Shellcode stored encrypted at unk_403218 and injected in colorcpl.exe

Then, the sample performs the following operations:

- creates a new process “colorcpl.exe” in suspended state

- makes space in memory for the payload using VirtualAllocEX

- copies the payload stored at location unk_403218 using WriteProcessMemory with Size “0x155” inside the process memory oxCF0000 (in my case, it changes every time dynamically)

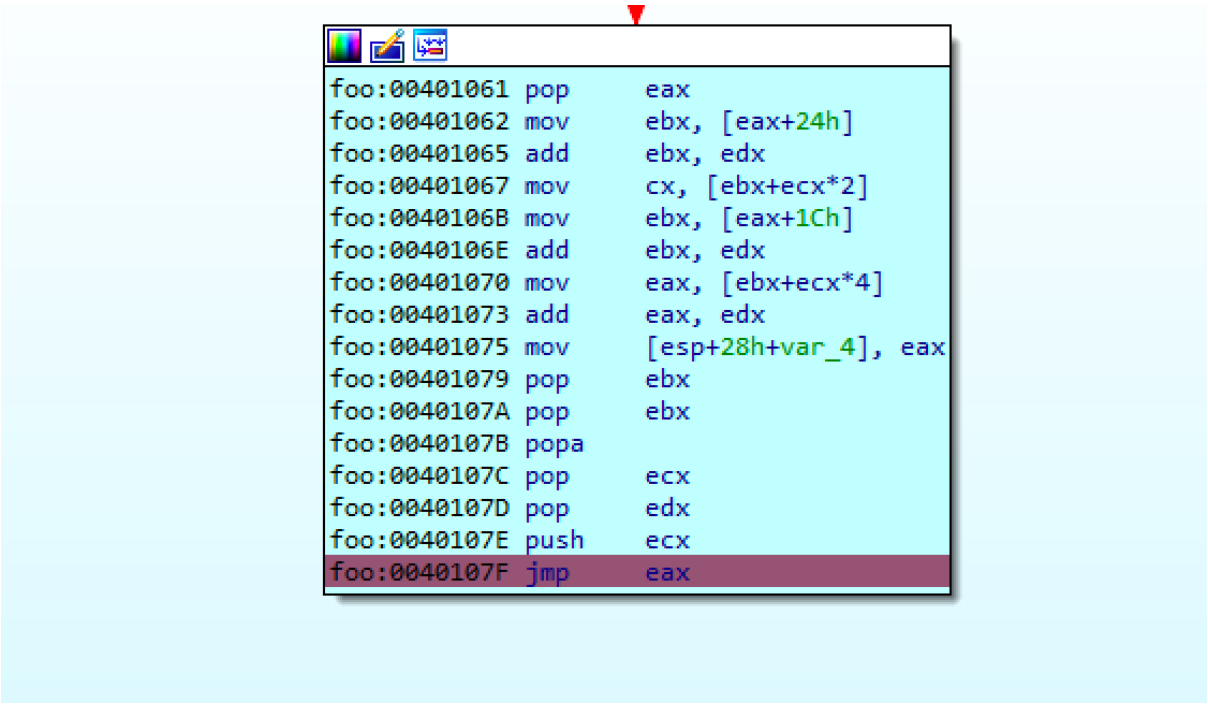
- deciphers an obfuscated string that will happen to be “NtCreateThreadEx” and then initiates the payload invoking it

How extract the payload:

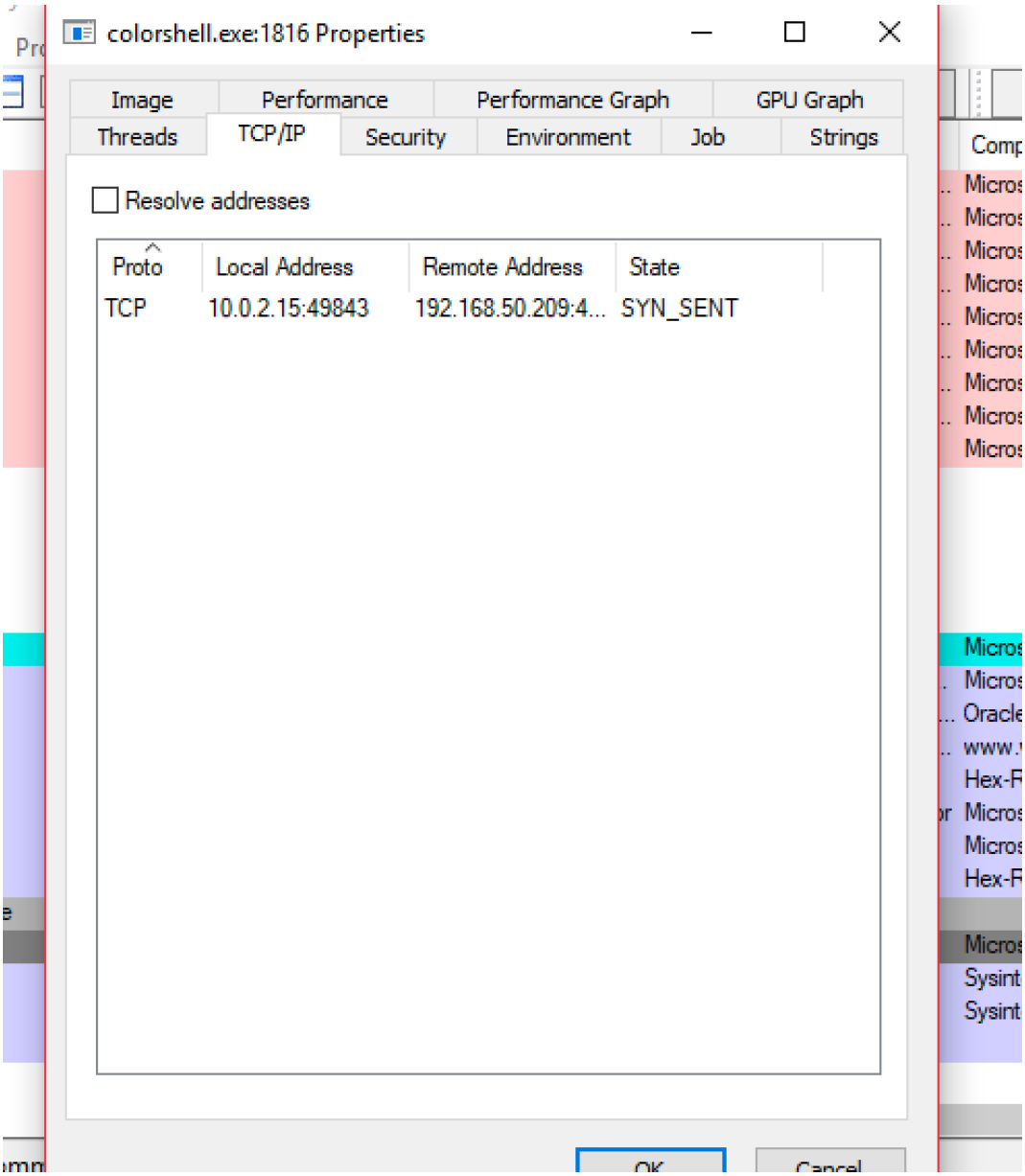
1. In Process Hacker, inspect the colorcpl.exe in which the shellcode is injected,
2. Find the address of lpBaseAddress in Memory,
3. Double click to see read/write memory,
4. Step over,
5. Refresh memory,
6. Select the bytes (including terminator) and save.

Inspect payload:

1. Convert the payload in an exectuable file using shellcode2exe
    a. `shellcode2exe.bat 32/64 <shellcode.bin> <shellcode.exe>`
2. Put a breakpoint on the last jmp eax instruction,
3. Execute the program a few times looking at the EAX register value.



We can see that the shellcode calls the network function: WSAStartup, WSASocketA, connect. It connects on port 49843.



10 - Does the sample beacon an external C2? Which kind of beaconing does the malware use? Which information is sent with the beacon? Does the sample implement any communication protocol with the C2? If so, describe the functionalities implemented by the protocol.

Yes, after connecting to the address 34.173.12.156 on port 80 via TCP, it sends a beacon “RDY” and waits (it calls recv) for a command:

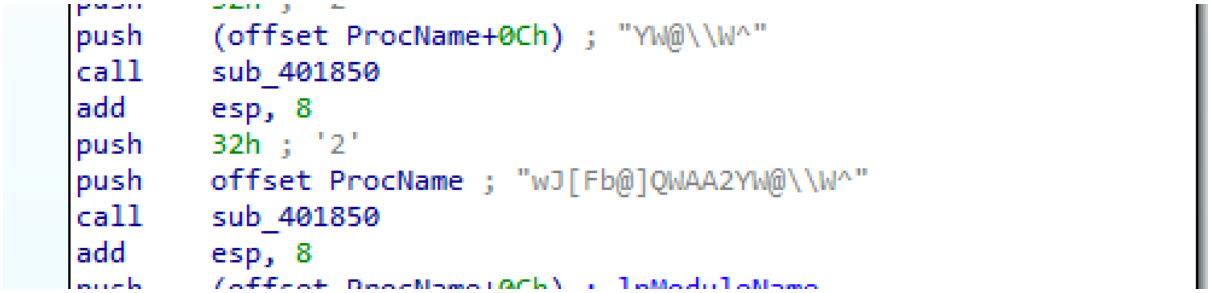
- Q: it sends BYE\n to the server and close the connection
- I: it opens the registry key Software\Microsoft\Cryptography using RegOpenKeyExA with hkey = 80000002h (HKEY_LOCAL_MACHINE) and it calls RegQueryValueExA to check if there is the value key “MachineGuid” and takes the corresponded data.
- G: it calls GetComputerName and send the computer name in this form: Computer name: %s\n
- H: it calls GetSystemInfo to retrieve the information of the system as number of

processors, architecture (x64, x86, Wow64)

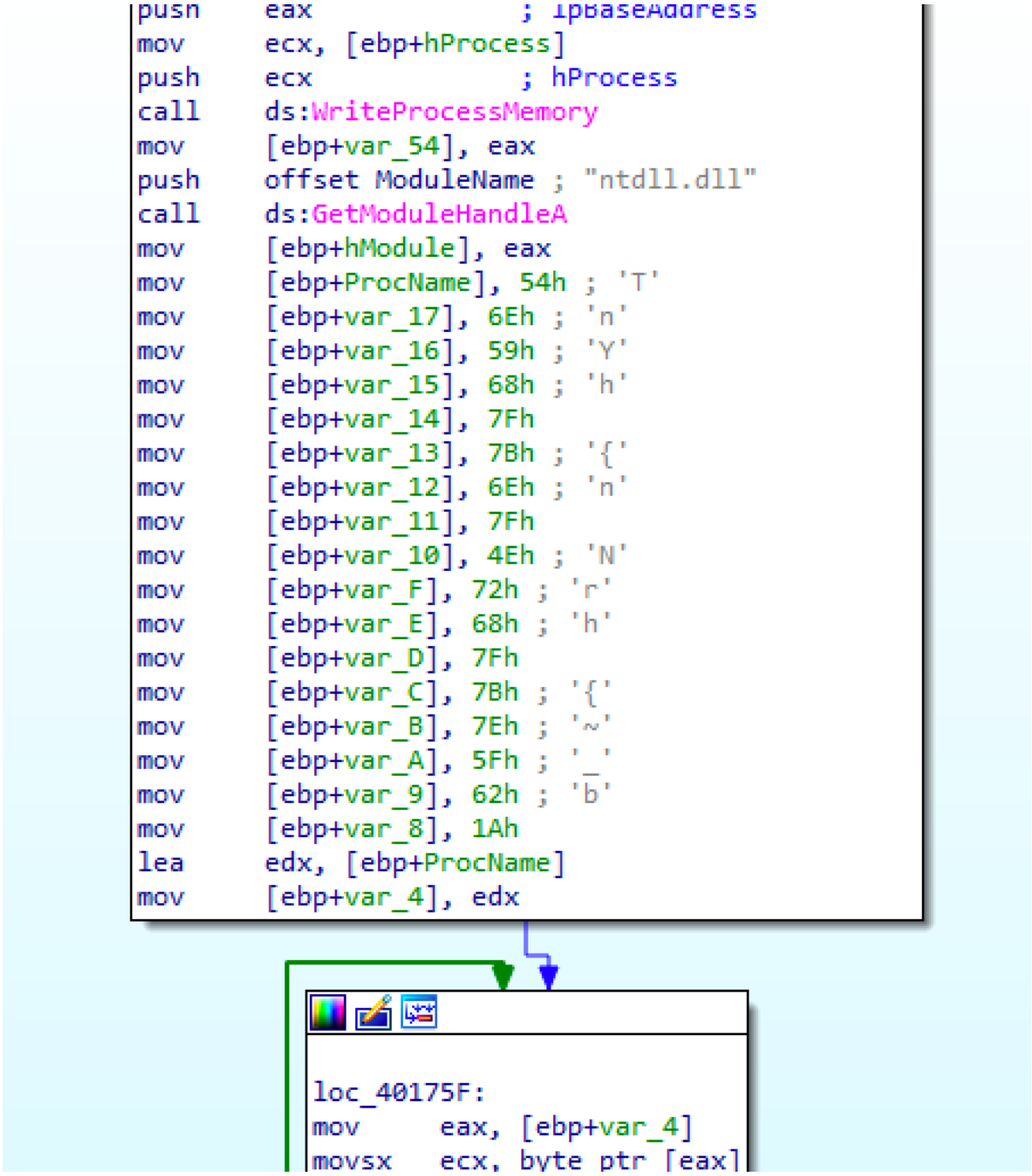
Details in answer 6

11 - List the obfuscation actions (if any) performed by the sample to hide its activities from a plain static analysis. Pinpoint and describe specific code snippets.

In the function main, the strings kernel and ExitProcess are xor-encrypted and not to be easily understandable even at a more advanced static analysis. The sample use the function sub_401850 to decrypt that strings.



In the sub_401650, the function “NtCreateThreadEx” is pushed byte-per-byte on stack, and is xor-encrypted not to be visible at a plain static analysis:


push eax ; lpBaseAddress
mov ecx, [ebp+hProcess]
push ecx ; hProcess
call ds:WriteProcessMemory
mov [ebp+var_54], eax
push offset ModuleName ; "ntdll.dll"
call ds:GetModuleHandleA
mov [ebp+hModule], eax
mov [ebp+ProcName], 54h ; 'T'
mov [ebp+var_17], 6Eh ; 'n'
mov [ebp+var_16], 59h ; 'Y'
mov [ebp+var_15], 68h ; 'h'
mov [ebp+var_14], 7Fh
mov [ebp+var_13], 7Bh ; '{'
mov [ebp+var_12], 6Eh ; 'n'
mov [ebp+var_11], 7Fh
mov [ebp+var_10], 4Eh ; 'N'
mov [ebp+var_F], 72h ; 'r'
mov [ebp+var_E], 68h ; 'h'
mov [ebp+var_D], 7Fh
mov [ebp+var_C], 7Bh ; '{'
mov [ebp+var_B], 7Eh ; '~'
mov [ebp+var_A], 5Fh ; '_'
mov [ebp+var_9], 62h ; 'b'
mov [ebp+var_8], 1Ah
lea edx, [ebp+ProcName]
mov [ebp+var_4], edx
loc_40175F:
mov eax, [ebp+var_4]
movsx ecx, byte ptr [eax]