Malware Analysis and Incident Forensics (Ms Cybersecurity)
Systems and Enterprise Security (Ms Eng. in CS)
Practical test - 10/07/2024

First name: Davide Last name: Del Vecchio Enrollment num.: 1779973

Email: delvecchio.1779973@studenti.uniroma1.it

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-20240710*.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?

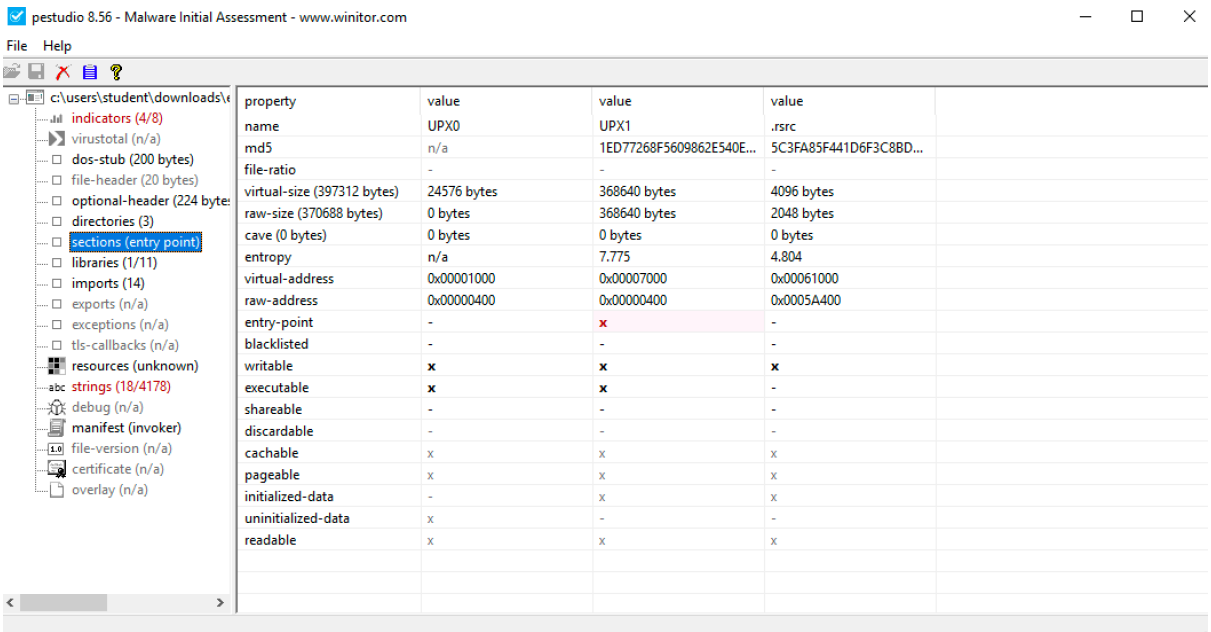
Examining the sample with **PEStudio**, we can retrieve from it many interesting things:

- Indicators of packing:
  - Sections names UPX0, UPX1
  - 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 LoadLibraryA 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 UPX1
  - Virtual size of the first section is much larger than its raw size

Maybe since the entry point is in UPX1, it will contains the decompression stub that will unpack the original executable at runtime and will store it in UPX0

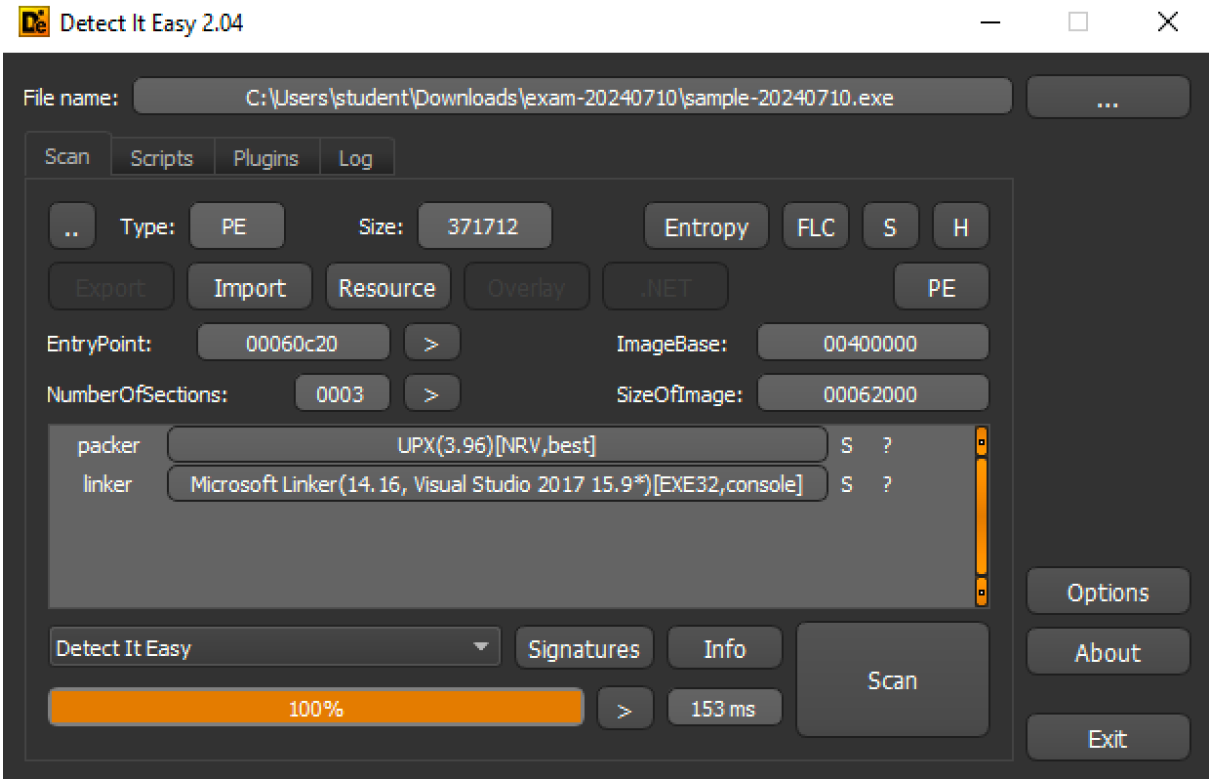
- The *imports* section contains some functions that are potential indicators or could reveal the sample's behavior. They are the following:
  - GetProcAddress, LoadLibraryA which are typical of packed software that has to rebuild the IAT
  - MessageBoxA, which means that the sample shows some message box
  - CryptStringToBinar, which is probably used for obfuscation 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, LoadLibraryA, SHGetFolderPath,

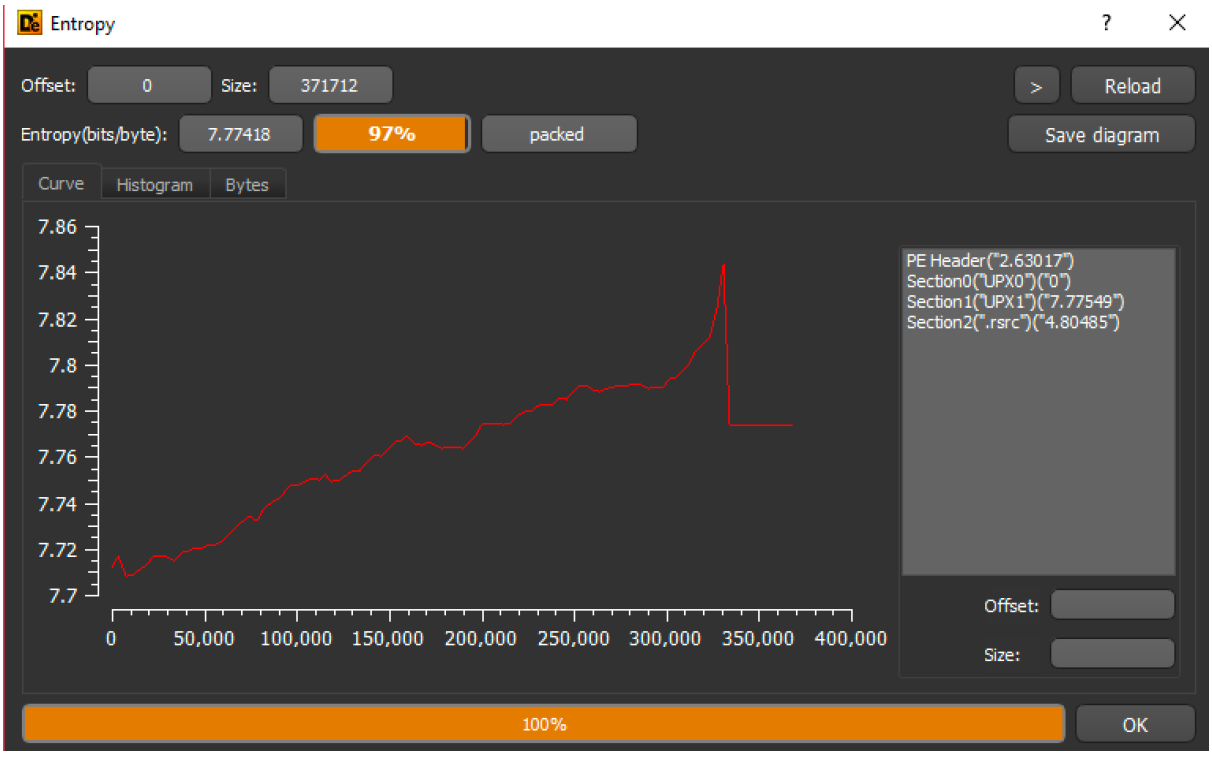
CryptStringToBinary, MessageBoxA)
    - Library names (NETAPI32.dll, kernel32.dll, shell32.dll, user32.dll)
    - Extentions (.jpg, .dll)
    - ?mafuba&%s\%s.jpgUghn, probably the sample create a file .jpg in a specific folder
- The library section contains some libraries that are potential indicators or could reveal the sample's behavior. They are the following:
    - shell32.dll, which likely means that the sample interacts with other processes
    - crypt32.dll, which likely means that the malware performs some kind of encryption
    - user32.dll, which likely means that the sample performs user-level interactions such as showing a message box
    - kernel32.dll, which likely means that the sample uses some important function like CreateFile, WriteFile, VirtualAlloc, WriteProcessMemory and others.



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 UPX (version 3.96). Furthermore, after clicking “Entropy”, it confirms the packing.



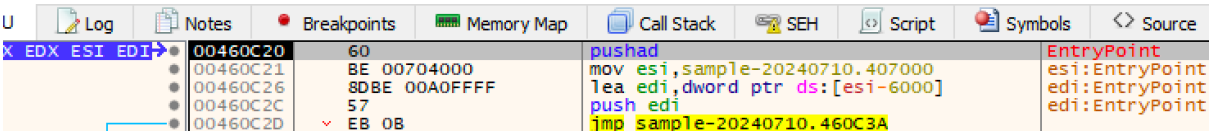


To find the OEP of a packed sample it's necessary to locate the tail jump, that is the jump that the packed sample performs to the beginning of the unpacked code after the unpacking stub has finished its operations.

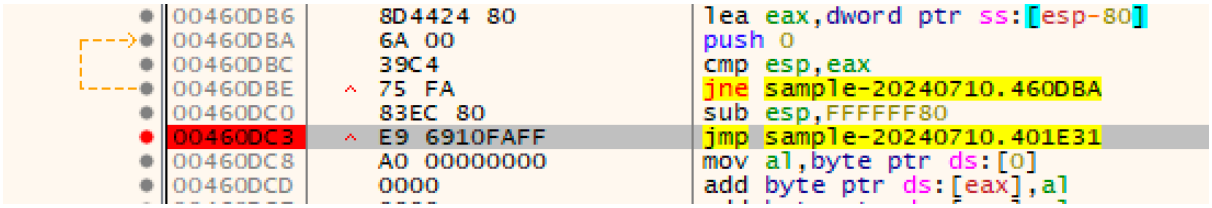
There are some indicators useful to recognize the tail jump that will allow us to fine the OEP:

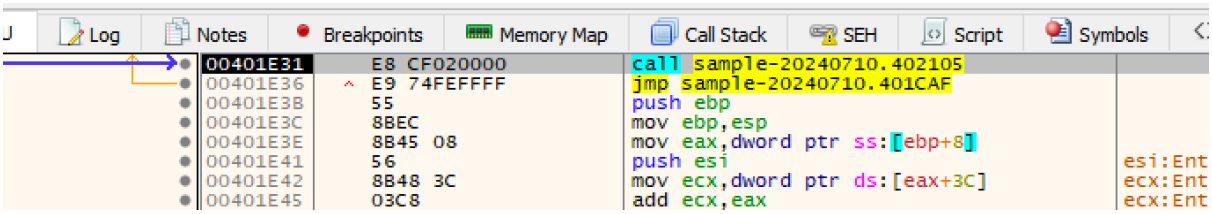
- The instruction jumps to another section (in this case from UPX1 to UPX0)
- After the tail jump should be a bunch of garbage bytes.
- The destination was previously modified by the unpacking stub

After opening the sample in x32dbg and starting at the entry point in UPX1 (0x460C20), 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 @ 0x460DC3
OEP @ 0x401E31

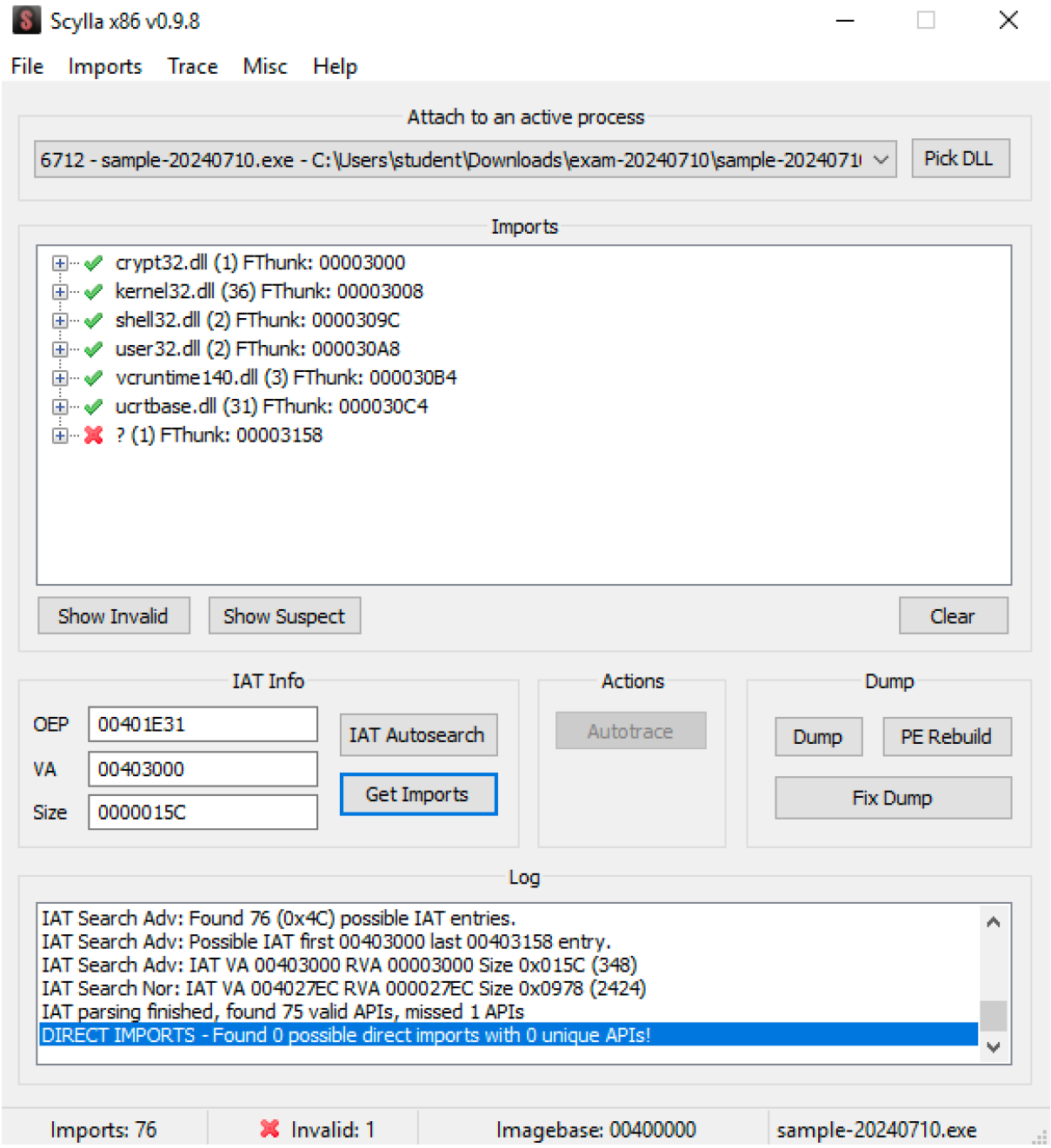




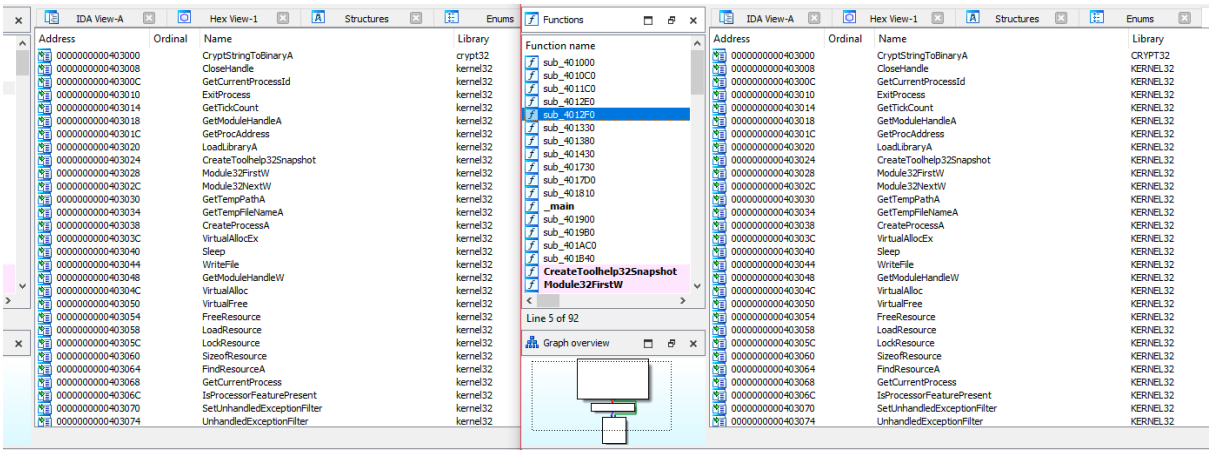
3 - Provide details about the IAT reconstruction process that you carried out to unpack the code. *HINTS: the answer should cover methodological aspects and facts on your output; also, validate it! (e.g., check API calls, compare with sample-20240710-unpacked.exe).*

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

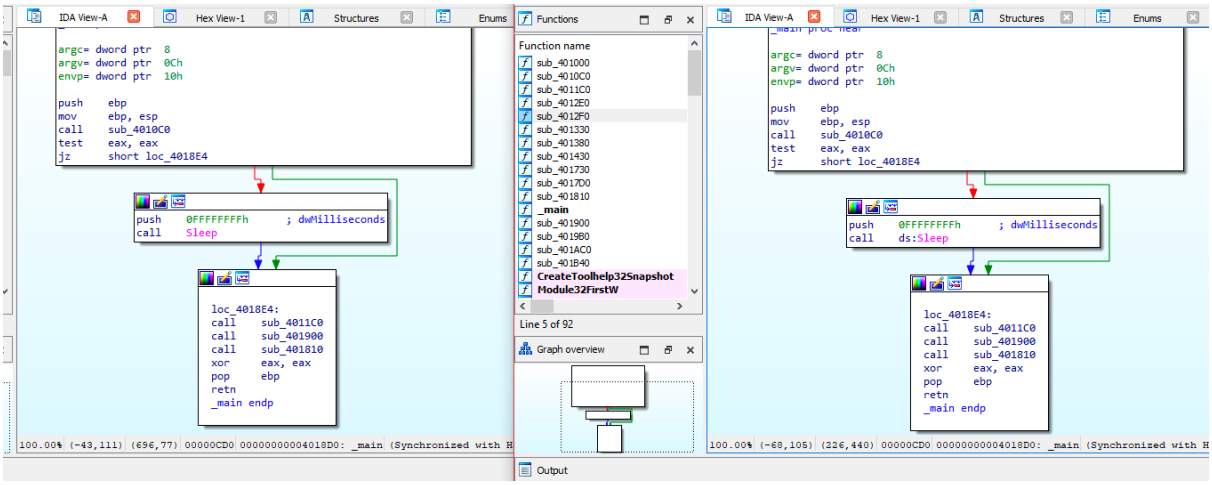
- Pressing **IAT Autosearch** we can obtain the IAT information starting from the OEP (0x401E31). At this point Scylla retrieves its virtual address and the size;
- Then, with **Get import** we can retrieve the list of imports. There is an invalid entry, as we can see in the screenshot, that can be deleted.
- 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-20241007-unpacked.exe).



I compare also the IDA View and we can see that there are the same functions.



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):

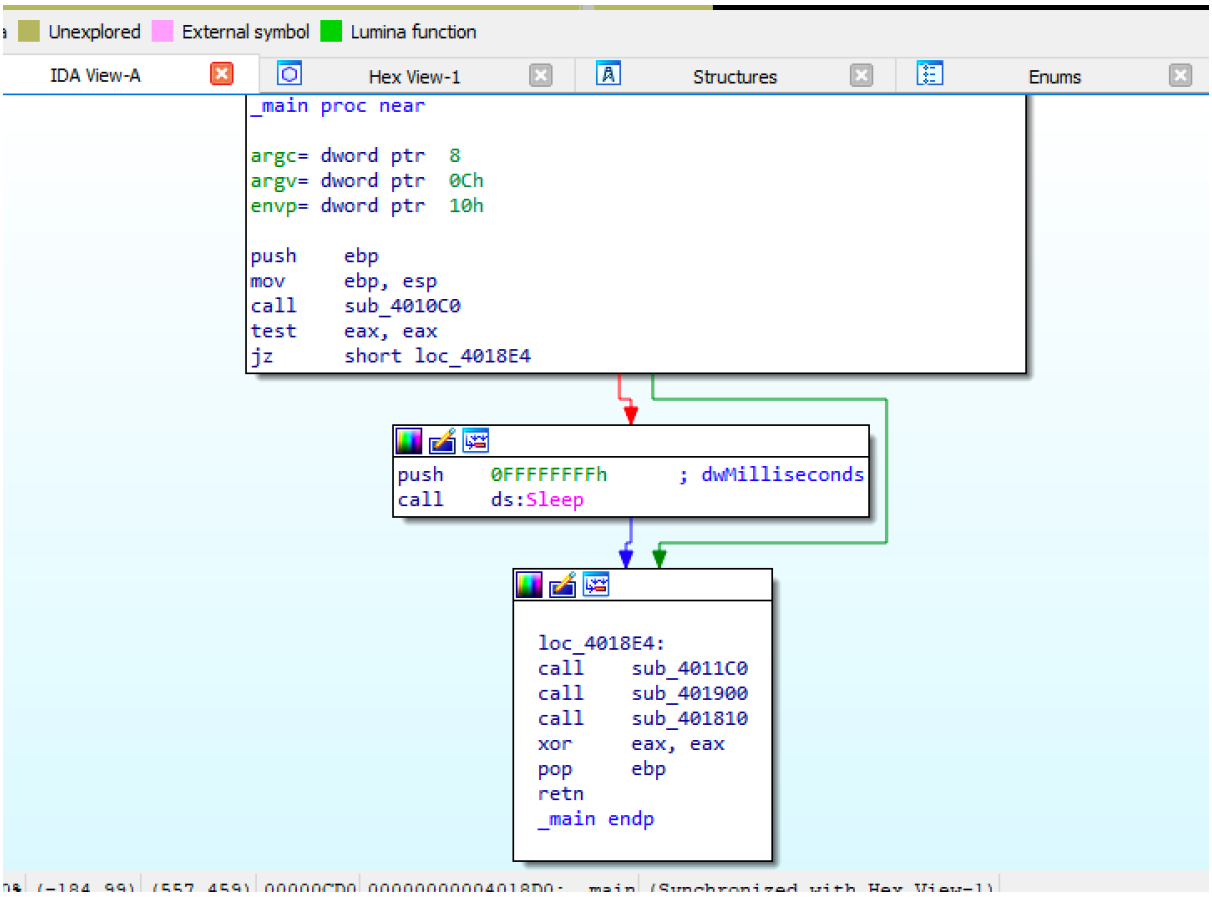
1. The sample checks if the system is running since a short time (1800000 ms). If so, it goes to sleep for a long period of time.
2. The sample checks if it has one or more modules with a specific name. If so, it exits after having changed the wallpaper.
3. The sample copies itself in the startup folder, to achieve persistence on the victim Machine and calls the file 7z.exe.
4. The sample injects a dll into the victim process: “explorer.exe”
5. The dll tries to connect to host 34.44.19.185 on port 80 and sends a specific message to it. Then it offers no C2 functionality.

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	mafuba.jpg	Image created in a specific case and set as wallpaper	IDA
Executable	7z.exe	Copy of the malware created for persistence purposes	IDA
File	<temp path>\scz<random-number>.dll	Dll created by the sample and file with the resource to inject it in process “explorer.exe”	IDA
Process	Explorer.exe	Victim process	IDA, process hacker
Network connection	34.44.19.185:80	Connection performs by the dll	IDA, process explorer, wireshark

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 0x4018D0:



sub_4010C0 (time check)

- It calls GetTickCount that is xor-encrypted from kernel32.dll that are also xor-encrypted and it decrypts them with key 30h. it uses GetTickCount to retrieve the number of millisecond that have passed since system startup.
- Then it compares this millisecond with 1800000 ms and if the millisecond of the system is high than 1800000 it returns 0 and the sample continuous the execution. Otherwise it goes to sleep for a long time period (0FFFFFFFFh)

sub_4011C0 (check occurrence)

- It calls GetCurrentProcessId to retrieve the id of the calling process.
- It calls CreateToolhelp32Snapshot with dwFlags = 8 to iterate the list of running processes. It calls Module32FirstW and Module32NextW to iterate the list of modules.
- It checks if there is an occurrence of any of these string in the modules:
    - daimao
    - piccolo
    - zamasu
    - dercori
    - frost
- if it founds an occurrence, it calls the function sub_401000:
    - in that function, the sample calls SHGetFolderPathA with csidl = 27h to retrieve the pictures folder: C:\Users\student\Pictures

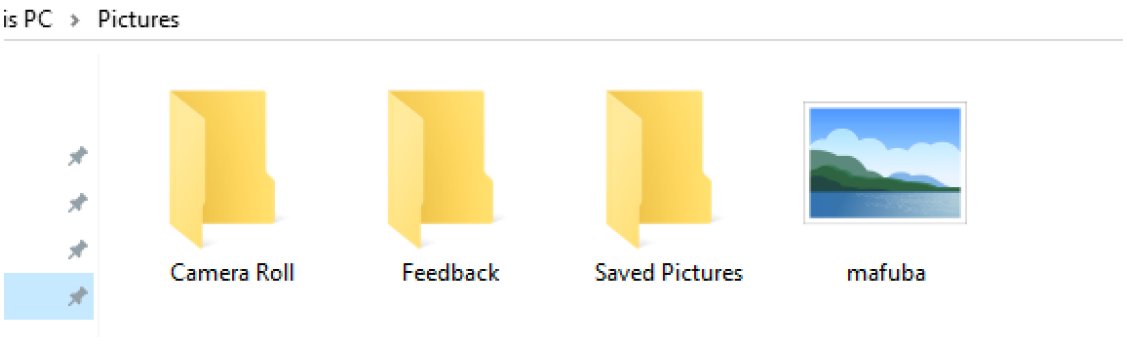
```
00  ..@.....'.......
73  ........C:\Users
65  \student\Picture
E7  s...............
```

- Then it calls CreateFileA with dwDesiredAccess = 40000000h (write permission) and WriteFile to insert the file mafuba.jpg in that folder

```
.text:0040103F add     esp, 14h
.text:00401042 push    0               ; hTemplateFile
.text:00401044 push    80h             ; dwFlagsAndAttributes
.text:00401049 push    2               ; dwCreationDisposition
.text:0040104B push    0               ; lpSecurityAttributes
.text:0040104D push    0               ; dwShareMode
.text:0040104F push    40000000h       ; dwDesiredAccess
.text:00401054 lea     ecx, [ebp+pszPath]
.text:0040105A push    ecx             ; lpFileName
.text:0040105B call    ds:CreateFileA
.text:00401061 mov     [ebp+hFile], eax
.text:00401064 push    0               ; lpOverlapped
.text:00401066 lea     edx, [ebp+NumberOfBytesWritten]
.text:00401069 push    edx             ; lpNumberOfBytesWritten
.text:0040106A mov     eax, [ebp+nNumberOfBytesToWrite]
.text:0040106D push    eax             ; nNumberOfBytesToWrite
.text:0040106E mov     ecx, [ebp+lpBuffer]
.text:00401071 push    ecx             ; lpBuffer
.text:00401072 mov     edx, [ebp+hFile]
.text:00401075 push    edx             ; hFile
.text:00401076 call    ds:WriteFile
.text:0040107C mov     eax, [ebp+hFile]
.text:0040107F push    eax             ; hObject
.text:00401080 call    ds:CloseHandle
(754,357) 0000045B 000000000040105B: sub_401000+5B (Synchronized with EIP)
```

```
00 24 00 00 00  24 00 00 00 AC F9 19 00  ....$...$.......
00 C8 F9 19 00  04 01 00 00 5C 20 45 00  ..@.........\·E.
00 24 00 00 00  00 00 00 00 D4 FA 19 00  ....$...........
00 C8 F9 19 00  04 01 00 00 5C 20 45 00  ?.@.........\·E.
00 54 20 45 00  43 3A 5C 55 73 65 72 73  ....T·E.C:\Users
75 64 65 6E 74  5C 50 69 63 74 75 72 65  \student\Picture
61 66 75 62 61  2E 6A 70 67 00 FF FF E7  s\mafuba.jpg....
00 FF FF FF FF  01 00 00 00 FF 57 5C 1F  ..............
```

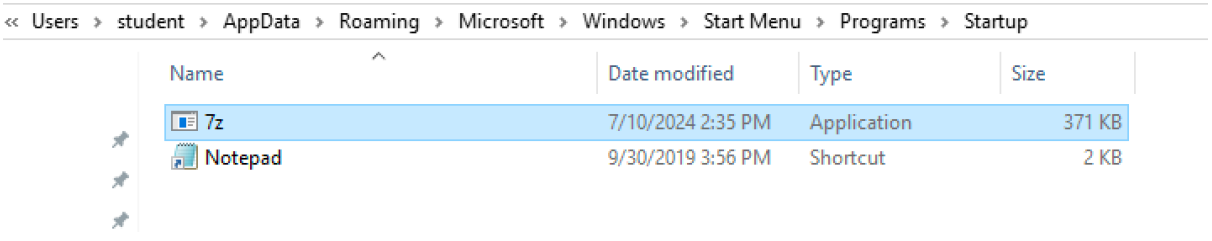
- Then it calls SystemParametersInfoA with uiAction = 14h to set the value of the system-wide parameter
- The image size is 4EE5Eh bytes



- Finally, it prints a message using MessageBoxA

sub_401900 (persistence)

- It calls SHGetFolderPathW with csidl = 0x07 to retrieve the path of the startup folder: “C:\Users\student\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup“
- It creates a copy of itself in that folder and calls this copy “7z.exe” using CopyfileW. In this way the sample can survive after reboot.

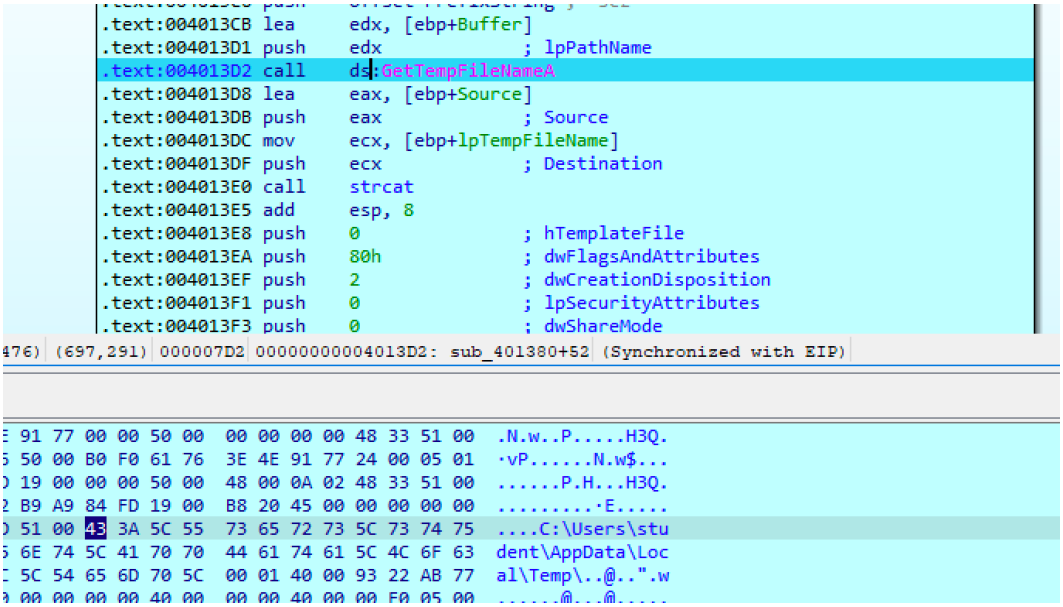


sub_401810 (dll injection)

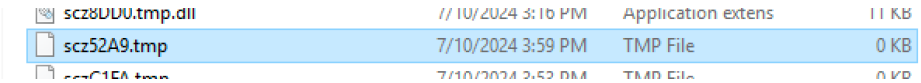
- It first calls the function sub_4017D0:
  - In that function, the sample calls the functions:
    - FindResourceA
    - LoadResource
    - LockResource
    - SizeofResource

- VirtualAlloc
- FreeResource

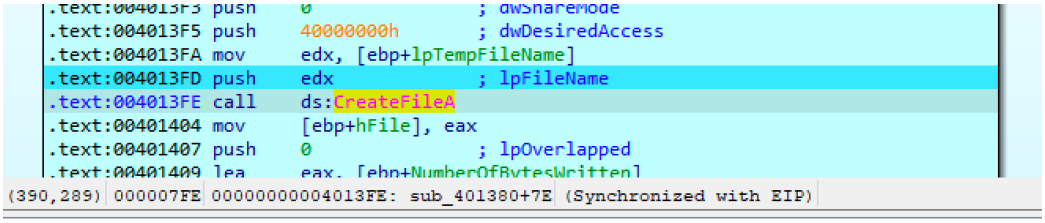
- Then it calls the function sub_401380:
  - In that function, the sample calls GetTempPathA to retrieve the path: “C:\Users\student\AppData\Local\Temp”.

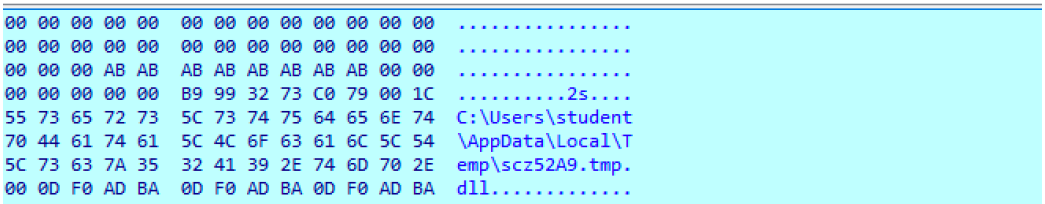


  - Then it calls GetTempFileNameA with lpPathName = temp path, to create the name of the temporary file in the tmp folder (in my case is scz52A9.tmp).

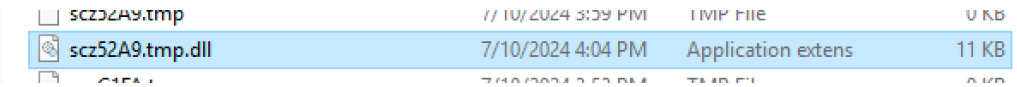


  - Then it calls CreateFileA with dwDesiredAccess = 40000000h (write permission) to create the dll file: “scz52A9.tmp.dll”.

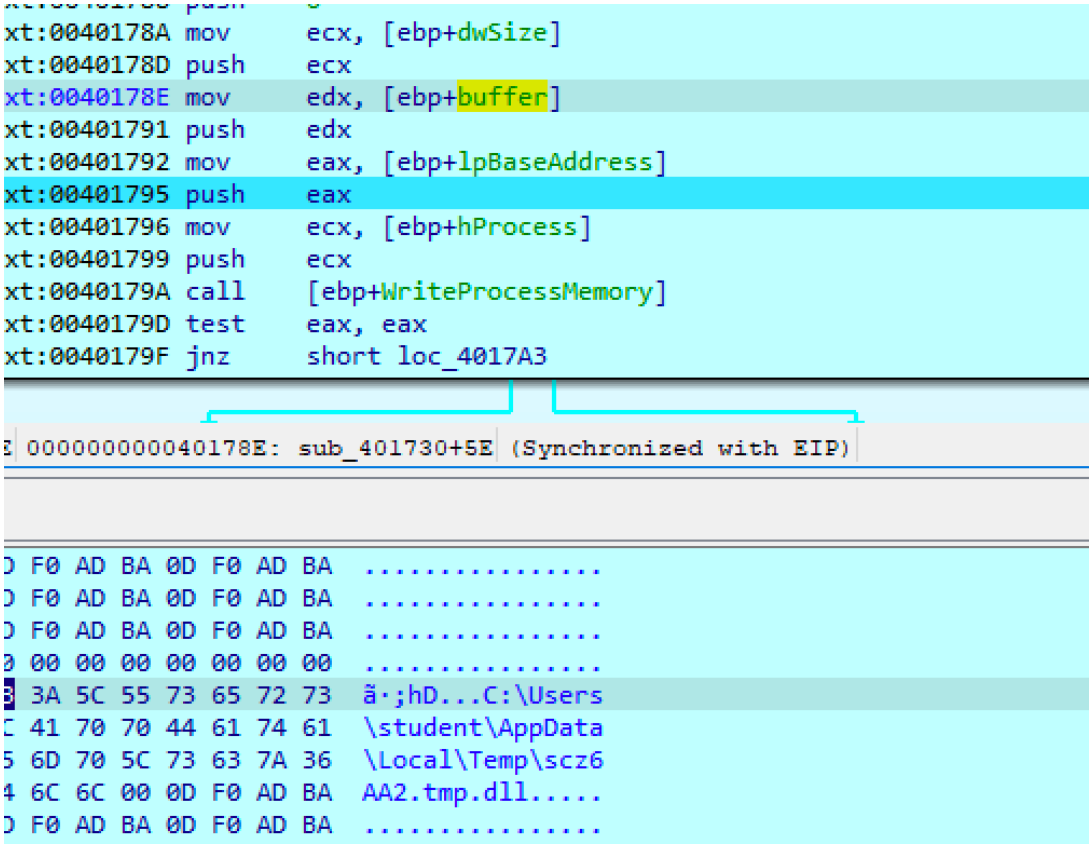




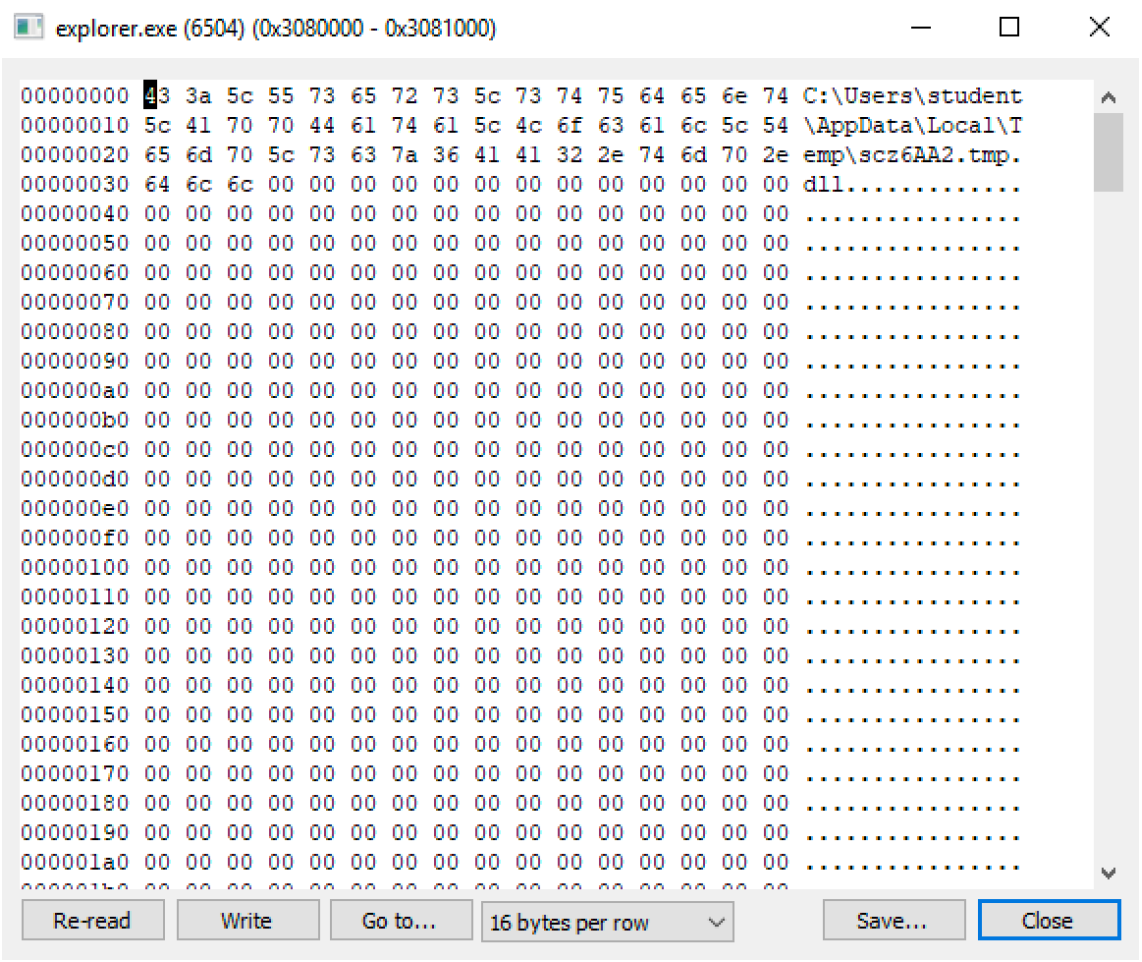
  - At the end, it calls WriteFile to write 2A00h bytes of malicious code in that dll file.



- It calls CreateProcessA to run the process “explorer.exe” in suspended state (dwCreationFlags = 4).
- Then it calls the function sub_401730:
    - In that function, it calls sub_401430:
        - It uses sub_401AC0 to decrypt the following functions:
            - NtUnmapViewOfSection
            - SetThreadContext
            - ResumeThread
            - NtCreateThreadEx
            - SuspendThreadEx
            - VirtualAllocEx
            - GetThreadContext
            - WriteProcessMemory
            - LoadLibraryA
    - In sub_401730 it calls VirtualAllocEx to allocate 0x34 bytes of memory in explorer.exe.



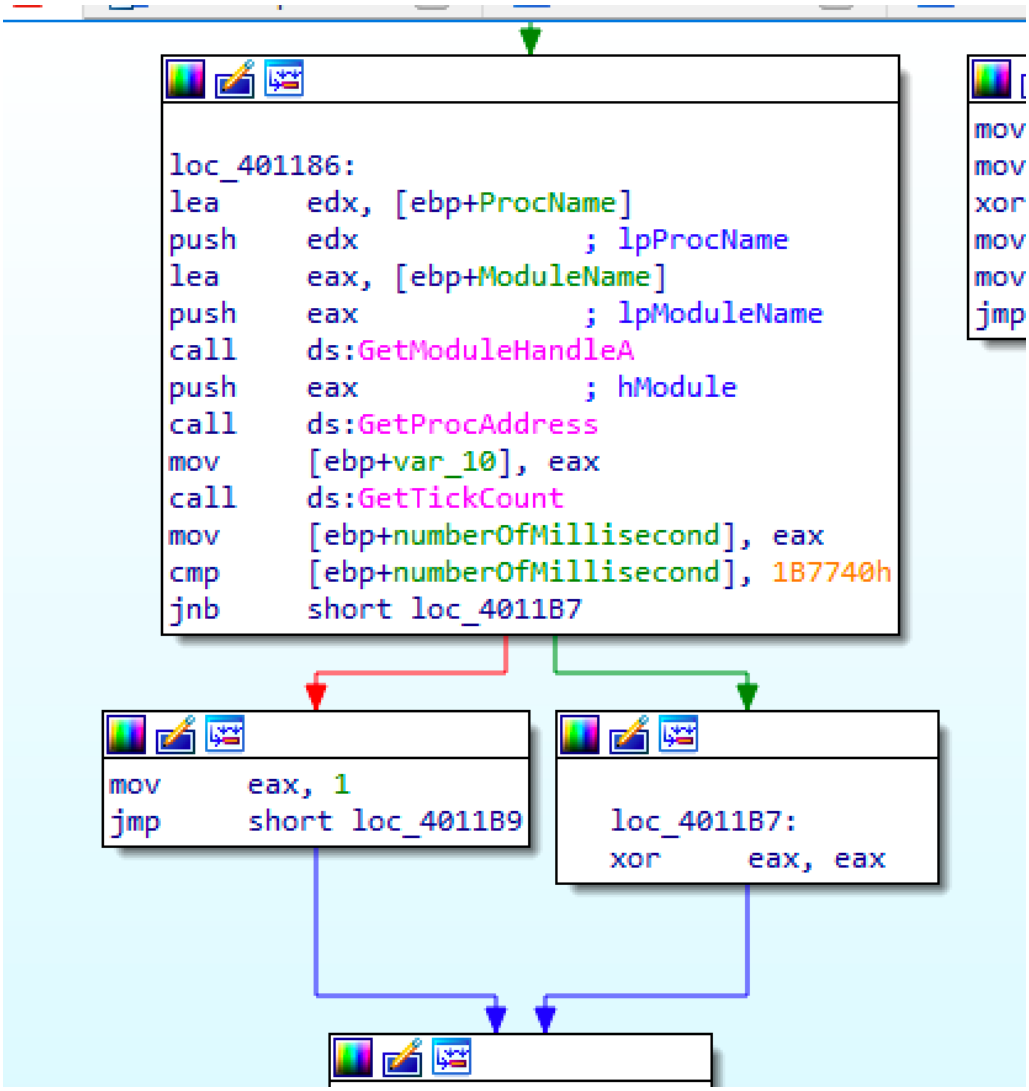
    - Then it calls WriteProcessMemory to inject the malicious code stored in scz<random number>.tmp.dll at the memory address 0x3080000 into explorer.exe.

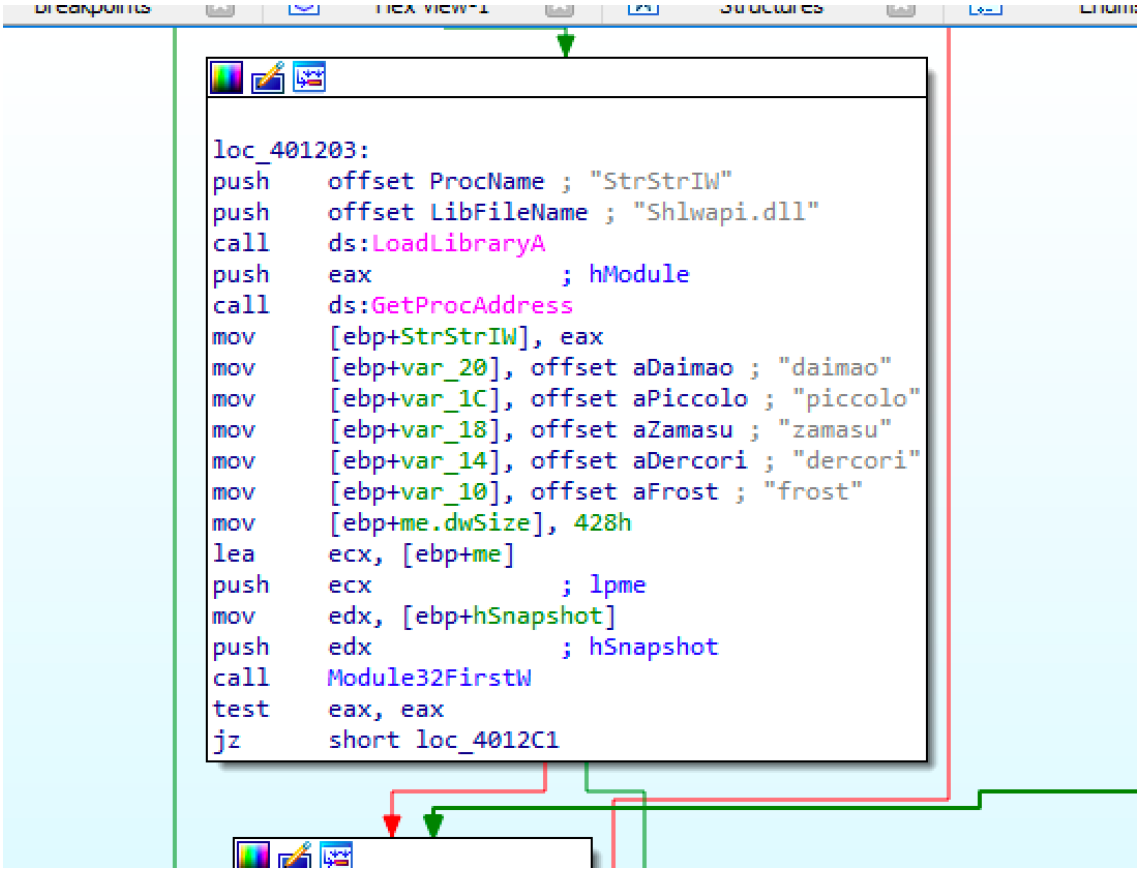


- At the end, it calls NtCreateThreadEx to create the thread for that victim process.

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.

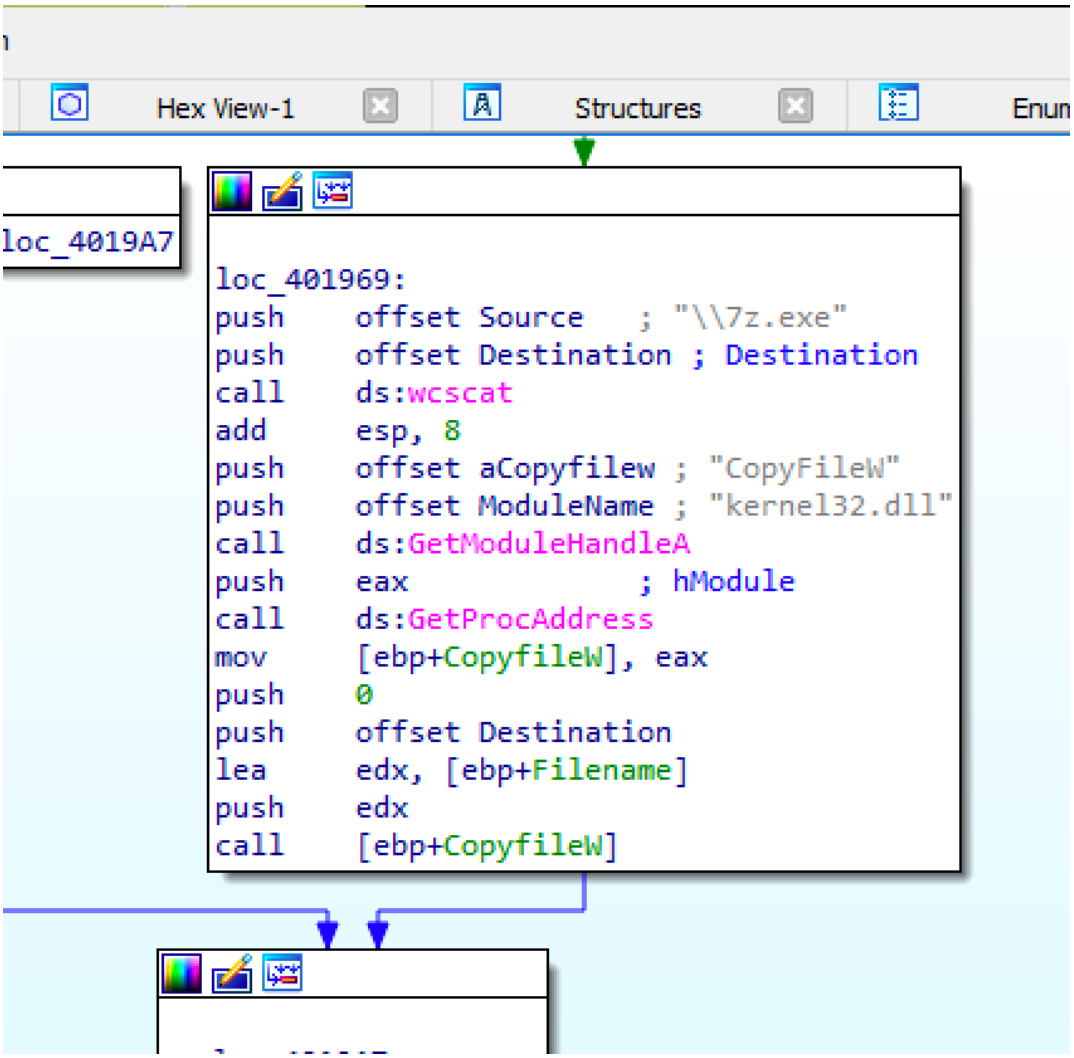
As show before in question 6, the sample, before performing its malicious activities, check if the system is running since a short time (1800000 ms) in **sub_4010C0**. Then it checks also if it has one or more modules with a specific name in **sub_4011C0**. Details in answer 6.


loc_401186:
lea edx, [ebp+ProcName]
push edx ; lpProcName
lea eax, [ebp+ModuleName]
push eax ; lpModuleName
call ds:GetModuleHandleA
push eax ; hModule
call ds:GetProcAddress
mov [ebp+var_10], eax
call ds:GetTickCount
mov [ebp+numberOfMillisecond], eax
cmp [ebp+numberOfMillisecond], 1B7740h
jnb short loc_4011B7
mov eax, 1
jmp short loc_4011B9
loc_4011B7:
xor eax, eax



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

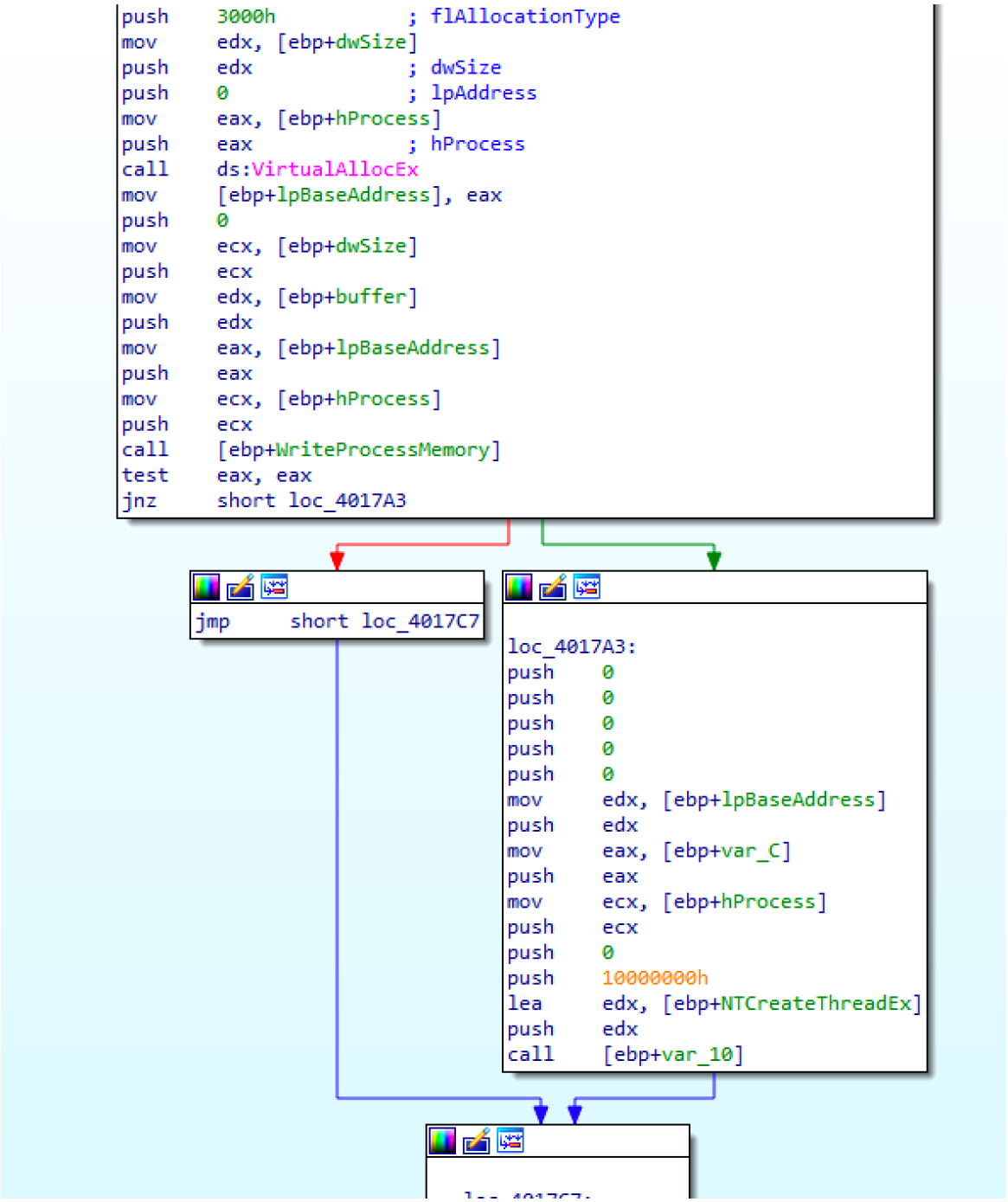
Persistence is achieved in this way: In the function **sub_401900**, the malware creates a copy of itself in the startup folder and calls this copy “7z.exe”. In this way, the malware will be automatically run at the startup of the machine. Details in answer 6.



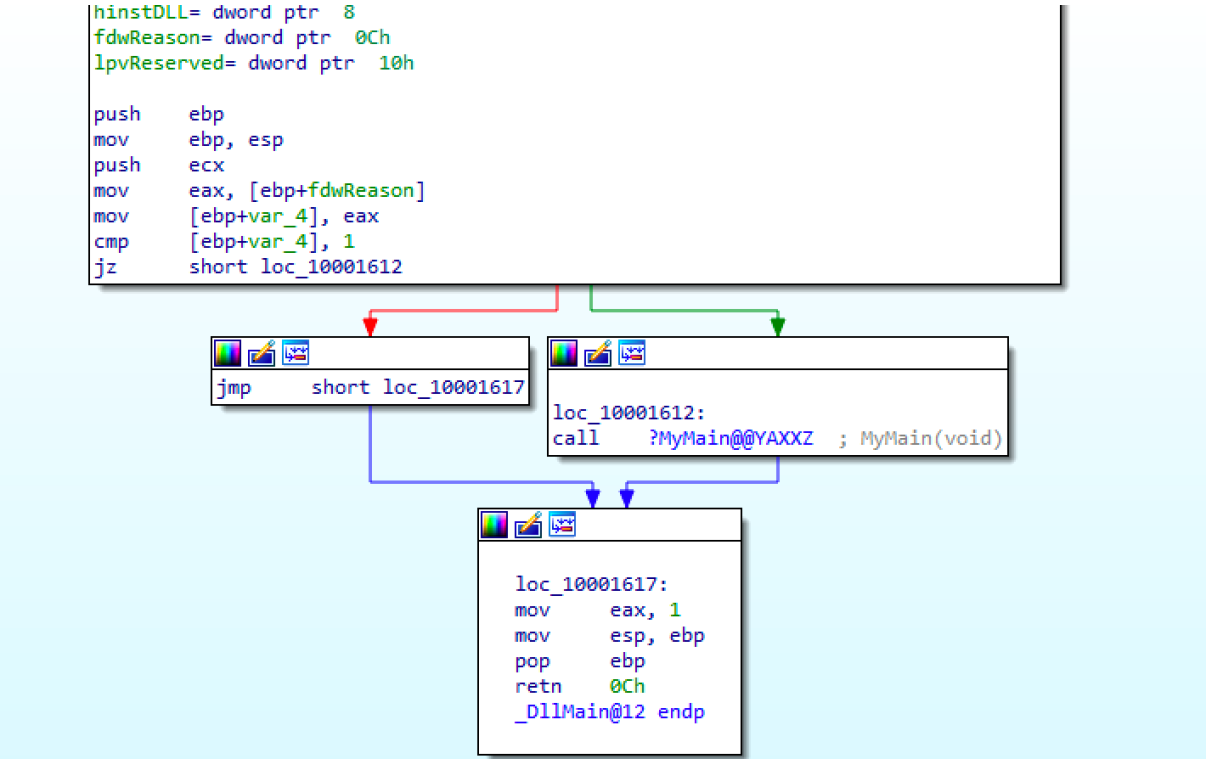
9 - Does the sample perform any code injection activities? Which kind of injection pattern do you recognize? Describe the characteristics and behavior of the injected payload, stating also where it is originally stored within the sample.

The sample performs dll injection. In fact, in **sub_401810** and in particular in sub routine **sub_401730**:

- It retrieves a dll from C:\Users\student\AppData\Local\Temp\SCZ<random-number>.dll and injected it in explorer.exe
- It loads, at run time, many functions, among which there are the ones necessary for dll injcetion (virtualAllocEx, WriteProcessMemory, NTCreateThreadEx, LoadLibraryA)
- it allocates memory to the victim process using VirtualAllocEx
- The memory allocated is 0x34 bytes
- It writes the path to the dll with WriteProcessMemory on this allocated memory at address 0x3080000 (this dynamically changes at each execution of the malware)
- It calls NTCreateThreadEx passing LoadLibraryA as function to execute, so that the victim process will load the dll and execute its malicious code.



When we open the dll file, the code is this:



The injected dll behaves as follow:

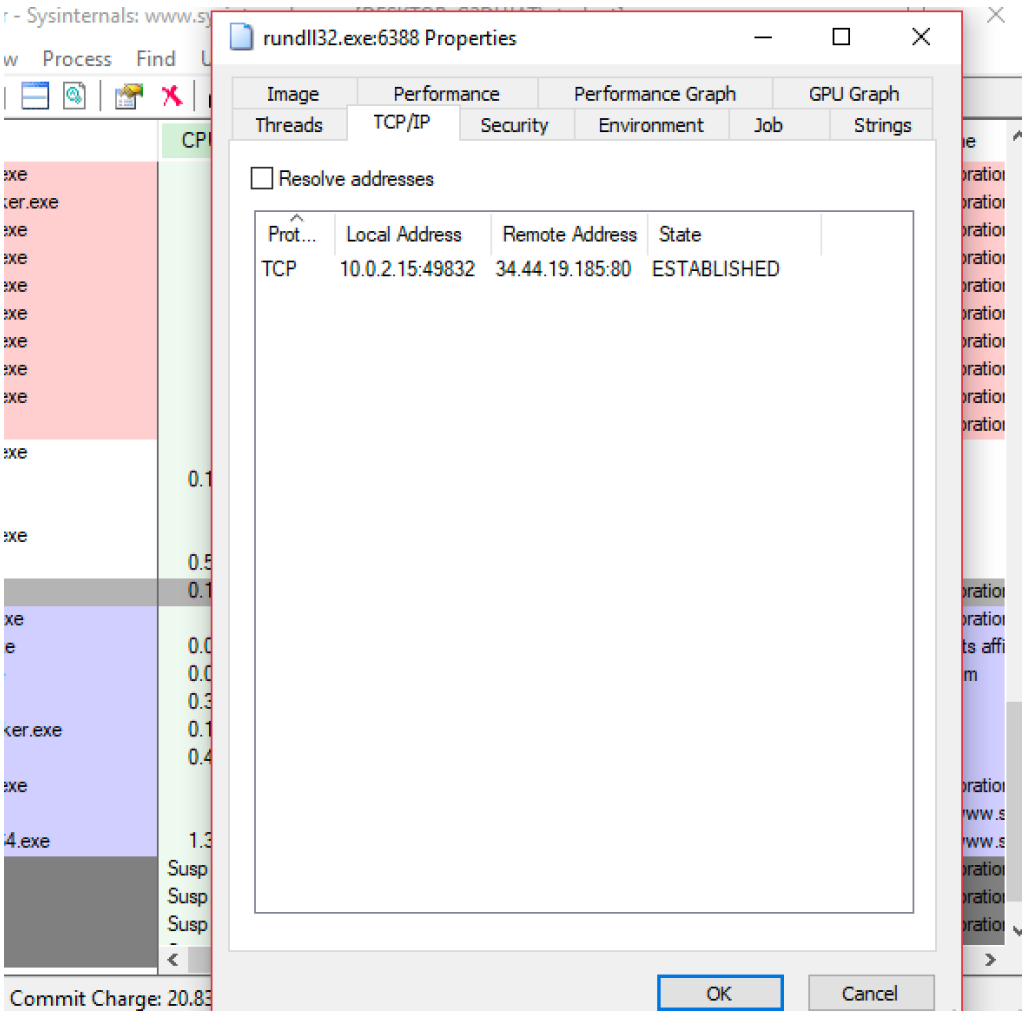
After opening the extracted file with **IDA** and using **rundll32** to load and execute the code in the dll, it is possible to analyze the dll. From DLLmain, we can click in lpStartAddress.

The operations done in lpStartAddress, that are the actual malicious operations, are the following:

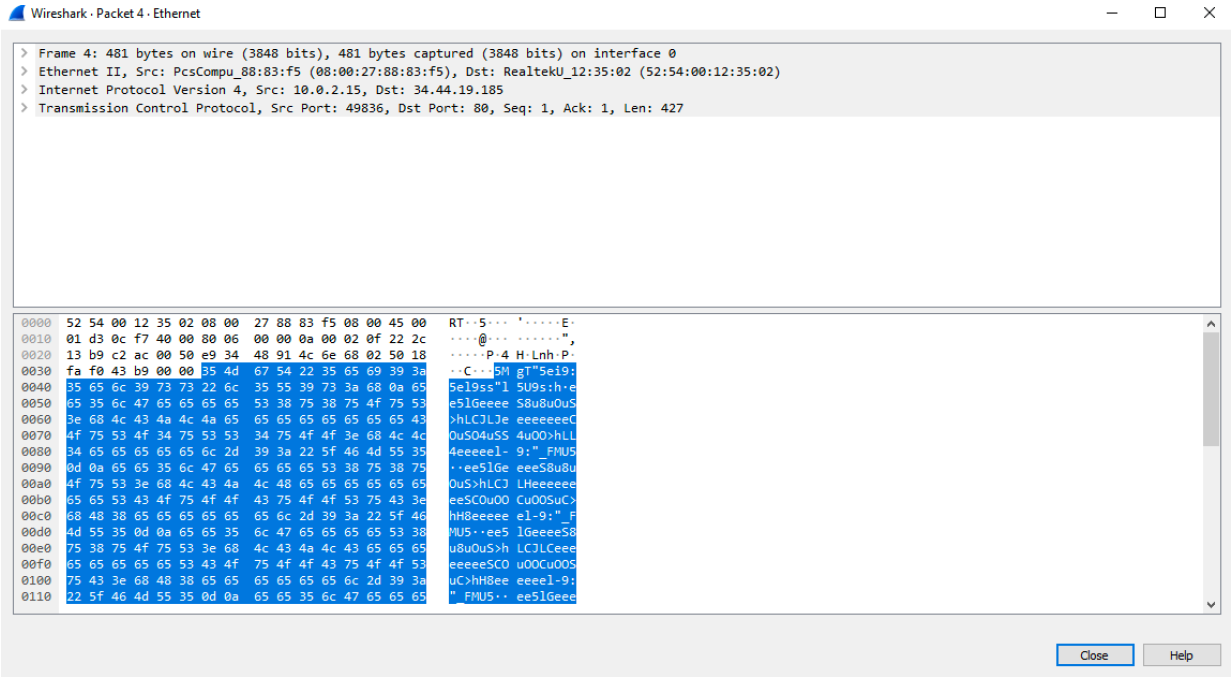
- It calls sub_10001000 and here it calls WSAStartup to retrieve all the network functions.
- It tries to connect to host 34.44.19.185 on port 80. And if it is successful, it prints “TARGET HOST CONNECTIONS:\n%s\n” where %s is the IPV4 address, and the port chosen by the windows OS:
    - In my case is: 10.0.2.15:49808

```
00  ................
47  ............TARG
49  ET·HOST·CONNECTI
30  ONS:.··TCP····10
20  .0.2.15:49808···
```

- Then it sends a particular string to the server. After that it sends BYE to the server and closes the connection.



	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.2.15	34.44.19.185	TCP	66	49836 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
2	0.173622	34.44.19.185	10.0.2.15	TCP	60	80 → 49836 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
3	0.173774	10.0.2.15	34.44.19.185	TCP	54	49836 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0
4	22.413565	10.0.2.15	34.44.19.185	TCP	481	49836 → 80 [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=427
5	22.413829	34.44.19.185	10.0.2.15	TCP	60	80 → 49836 [ACK] Seq=1 Ack=428 Win=65535 Len=0
6	24.601435	34.44.19.185	10.0.2.15	TCP	60	80 → 49836 [PSH, ACK] Seq=1 Ack=428 Win=65535 Len=1 [TCP segment of a re



In this screen, we can notice that the port is different for the previous one, this because they are screen related to other execution and the port chosen by the OS changes randomly.

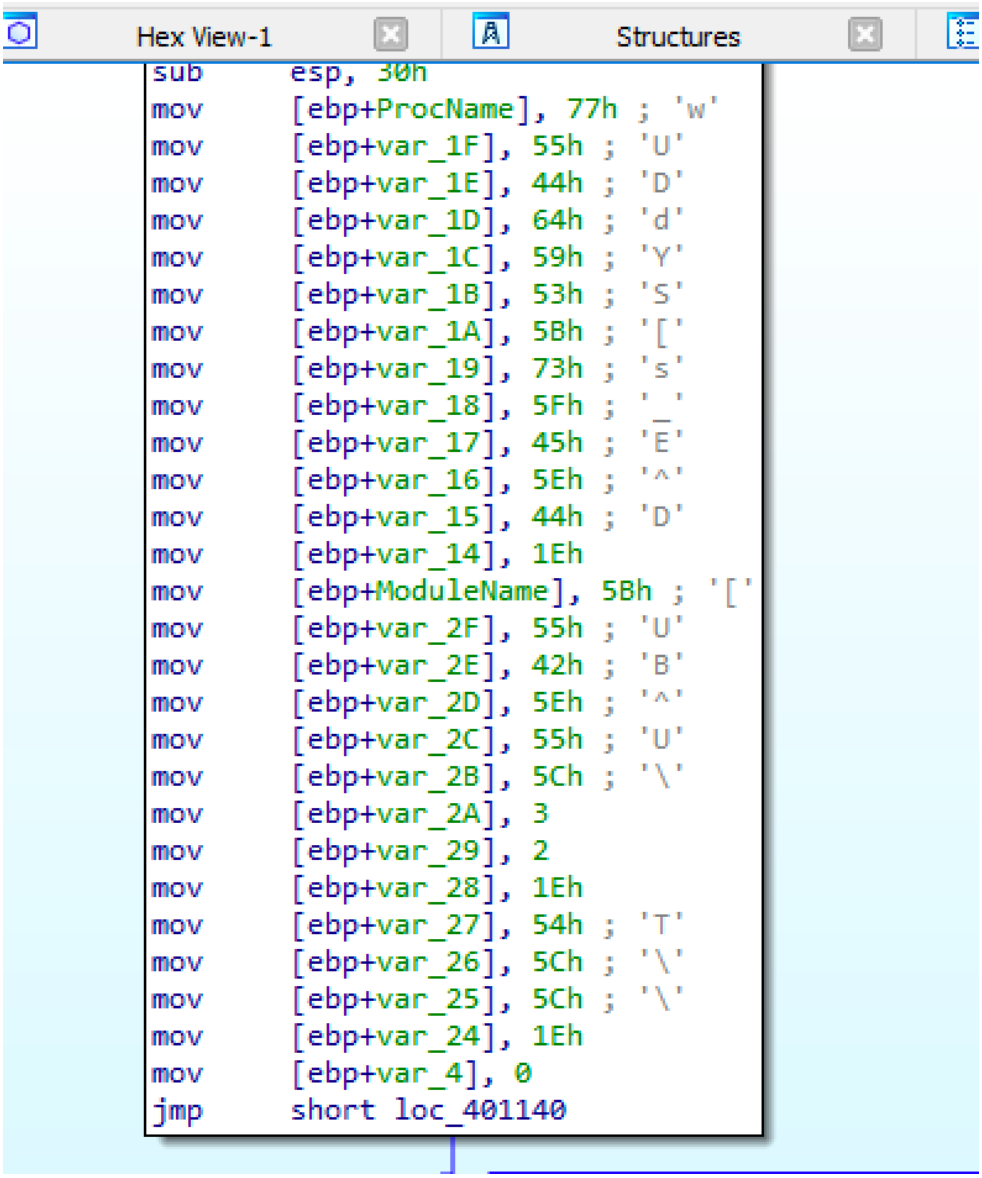
10 - Does the sample beacon an external C2? Which kind of beaconing does the malware use? Which information is sent with the beacon? Does the sample implement any communication protocol with the C2? If so, describe the functionalities implemented by the protocol.

Yes, after connecting to the address 34.44.19.185, it sends a beacon “TARGET HOST CONNECTIONS: 10.0.2.15:49808”. The main binary only beacons and offers no C2 functionality. After it sends “BYE” and closes the connection.

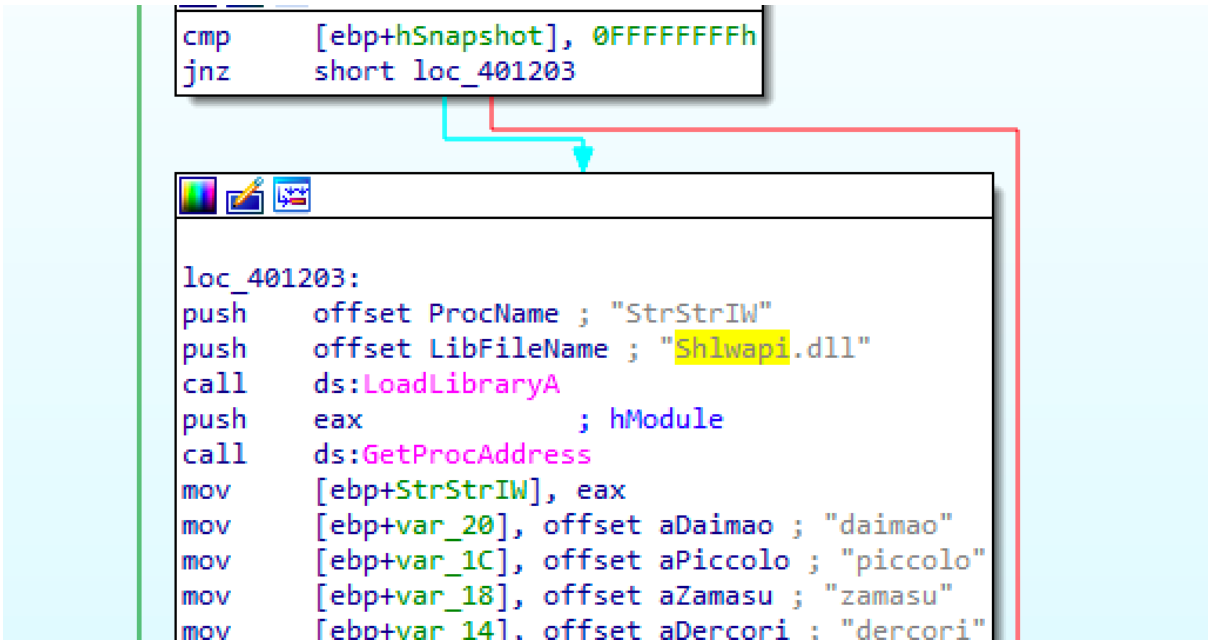
Details in answer 9

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.

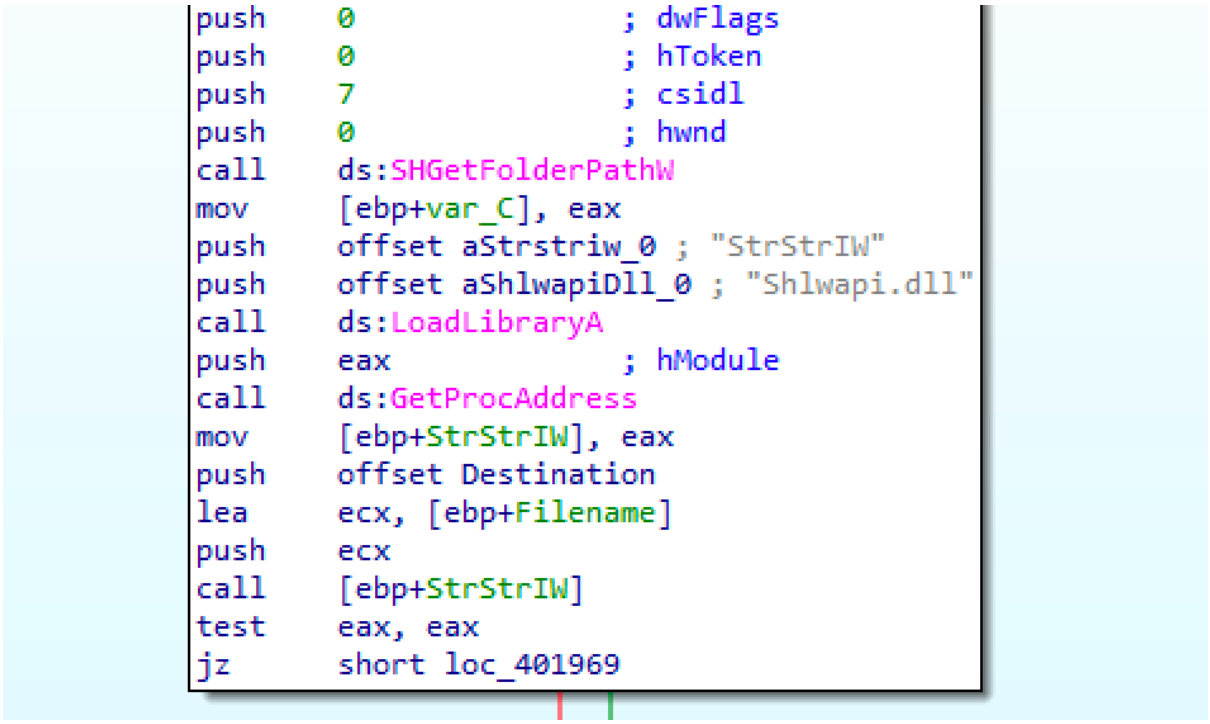
In function **sub_4010C0**, the strings used to load “getTickCount” are pushed to stack byte-per-byte not to be visible at a basic static analysis. They are also xor-encrypted not to be easily understandable even at a more advanced static analysis



In function **sub_4011C0**, the function StrStrIW and the library Shlwapi.dll is loaded at runtime, to avoid it being visible among the import.



Something very similar happens, for the function sub_401900



In function **sub_401810**, the name of the process to be injected (i.e. “explorer.exe”) is pushed on stack byte-per-byte not to be visible at a plain static analysis:

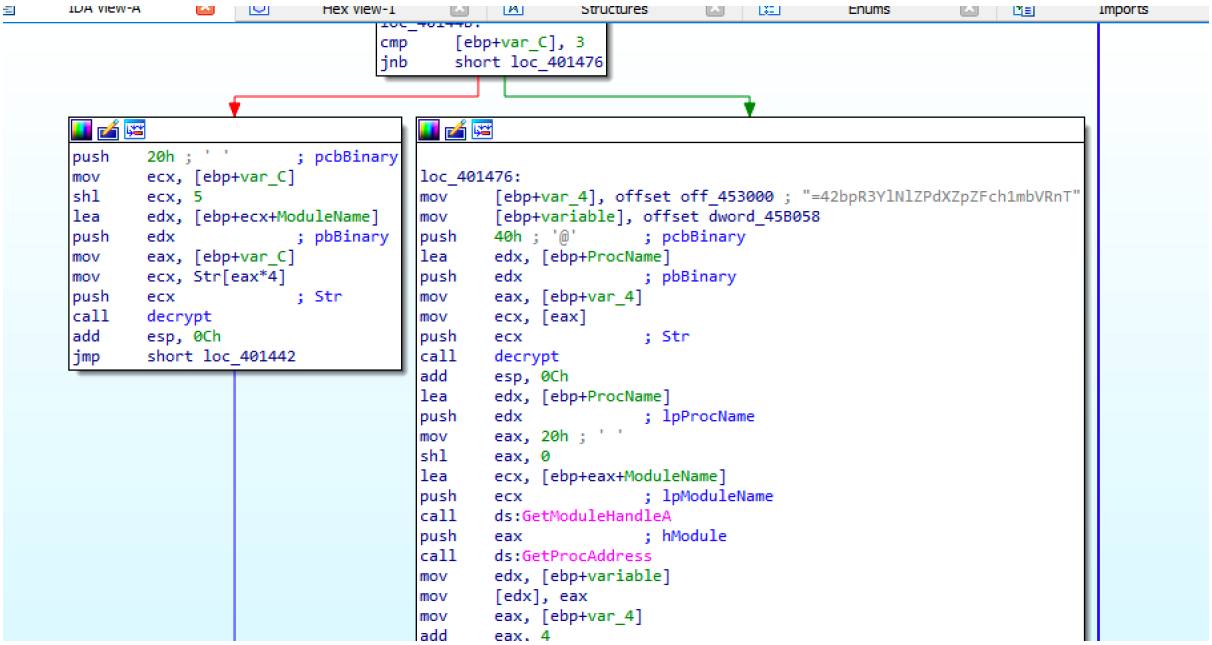
```
call    sub_401700
mov     [ebp+Str], eax
mov     eax, [ebp+Str]
push    eax             ; Str
call    strlen
add     esp, 4
add     eax, 1
mov     [ebp+dwSize], eax
mov     [ebp+CommandLine], 65h ; 'e'
mov     [ebp+var_1F], 78h ; 'x'
mov     [ebp+var_1E], 70h ; 'p'
mov     [ebp+var_1D], 6Ch ; 'l'
mov     [ebp+var_1C], 6Fh ; 'o'
mov     [ebp+var_1B], 72h ; 'r'
mov     [ebp+var_1A], 65h ; 'e'
mov     [ebp+var_19], 72h ; 'r'
mov     [ebp+var_18], 2Eh ; '.'
mov     [ebp+var_17], 65h ; 'e'
mov     [ebp+var_16], 78h ; 'x'
mov     [ebp+var_15], 65h ; 'e'
mov     [ebp+var_14], 0
mov     [ebp+dwCreationFlags], 4
push    44h ; 'D'       ; Size
push    0               ; Val
lea     ecx, [ebp+StartupInfo]
push    ecx             ; void *
call    memset
add     esp, 0Ch
```

In the **sub_401380** the extension .dll is pushed byte-per-byte on stack.

```
push    ebp
mov     ebp, esp
sub     esp, 118h
mov     [ebp+Source], 2Eh ; '.'
mov     [ebp+var_B], 64h ; 'd'
mov     [ebp+var_A], 6Ch ; 'l'
mov     [ebp+var_9], 6Ch ; 'l'
mov     [ebp+var_8], 0
push    104h            ; Size
call    ds:malloc
add     esp, 4
mov     [ebp+lpTempFileName], eax
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
push    104h            ; nBufferLength
call    ds:GetTempPathA
mov     ecx, [ebp+lpTempFileName]
```

In the **sub_401430** the functions: NtUnmapViewOfSection, SetThreadContext, ResumeThread, NtCreateThreadEx, SuspendThreadEx, VirtualAllocEx, GetThreadContext,

WriteProcessMemory, LoadLibraryA are encrypted and not to be visible at a plain static analysis.




```
.data:00453000 _data           segment para public 'DATA' use32
.data:00453000                 assume cs:_data
.data:00453000                 ;org 453000h
.data:00453000 off_453000      dd offset a42bpr3ylnlzpdx
.data:00453000                                         ; DATA XREF: sub_401430:loc_401476↑o
.data:00453000                                         ; "=42bpR3YlNlZPdXZpZFch1mbVRnT"
.data:00453004                 dd offset aAd4vgdu92qkfwz ; "==Ad4VGdu92QkFWZyhGV0V2U"
.data:00453008                 dd offset aKfwzyhgvl1wdzv ; "kFWZyhGVl1WdzVmU"
.data:0045300C                 dd offset aAefrwyljhauvgd ; "==AeFRWYlJHaUVGdhVmcDRnT"
.data:00453010                 dd offset aAzhvmcorfzuvgc ; "==AZhVmcoRFZuVGczV3U"
.data:00453014                 dd offset aGxrj9gbsfebhvh ; "=gXRj9GbsFEbhVHdylmV"
.data:00453018                 dd offset aAd4vgdu92qkfwz_0 ; "==Ad4VGdu92QkFWZyhGV0V2R"
.data:0045301C                 dd offset a5j3btvwtznxzj9 ; "5J3btVWTzNXZj9mcQVGdpJ3V"
.data:00453020                 dd offset aBlnchjnypxezh9 ; "BlnchJnYpxEZh9GT"
.data:00453024 ; char *Str
```