Malware Analysis and Incident Forensics (Ms Cybersecurity)
Systems and Enterprise Security (Ms Eng. in CS)
Practical test - 01/16/2024

First name: Last name: Enrollment num.: Email: (it will be used to send you the outcome of this test)

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-202401016*.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?

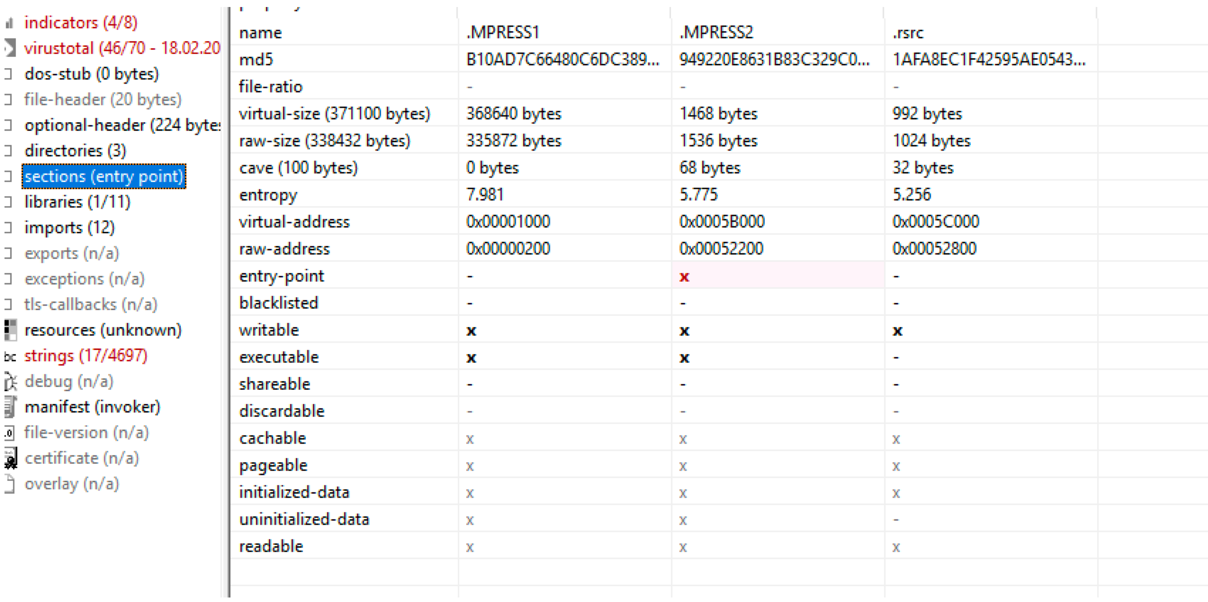
Examining the sample with PEStudio, we can retrieve from it many interesting things:

- 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 originale executable at runtime and will store it in .MPRESS1

- Interesting strings:
    - Function names (GetProcAddress, GetModuleHandle)
    - Library names (NETAPI32.dll, kernel32.dll, shell32.dll…)
    - Extentions (.dll)
    - CryptStringToBinary, probably the malware uses this function to crypt and encrypt something
    - Shlwapi.dll, probably the malware injects this dll in a process
    - SHGetFolderPath, probably the malware retrieves the folder where there is the malicious code
- Libraries:
    - There are some import like GetProcAddress from kernel32.dll
    - CryptStringToBinaryA from crypt32.dll

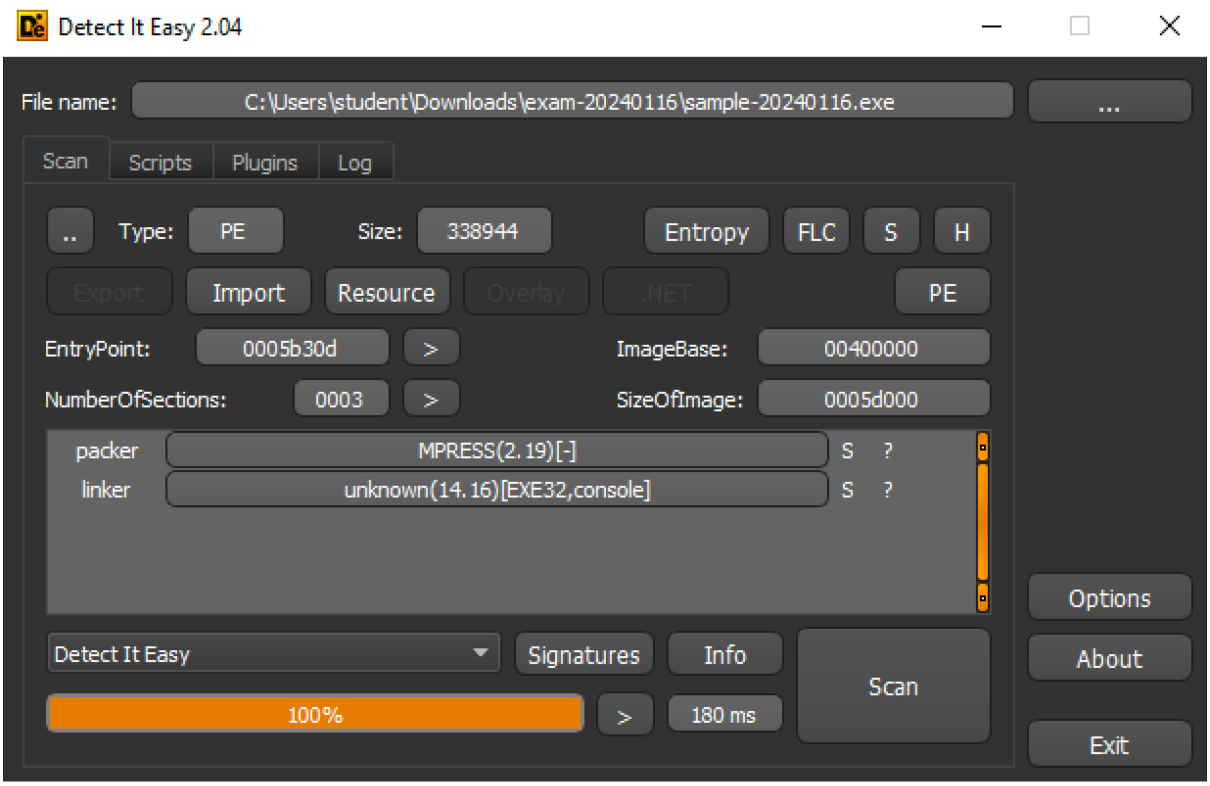
- SHGetFolderPathA from shell32.dll

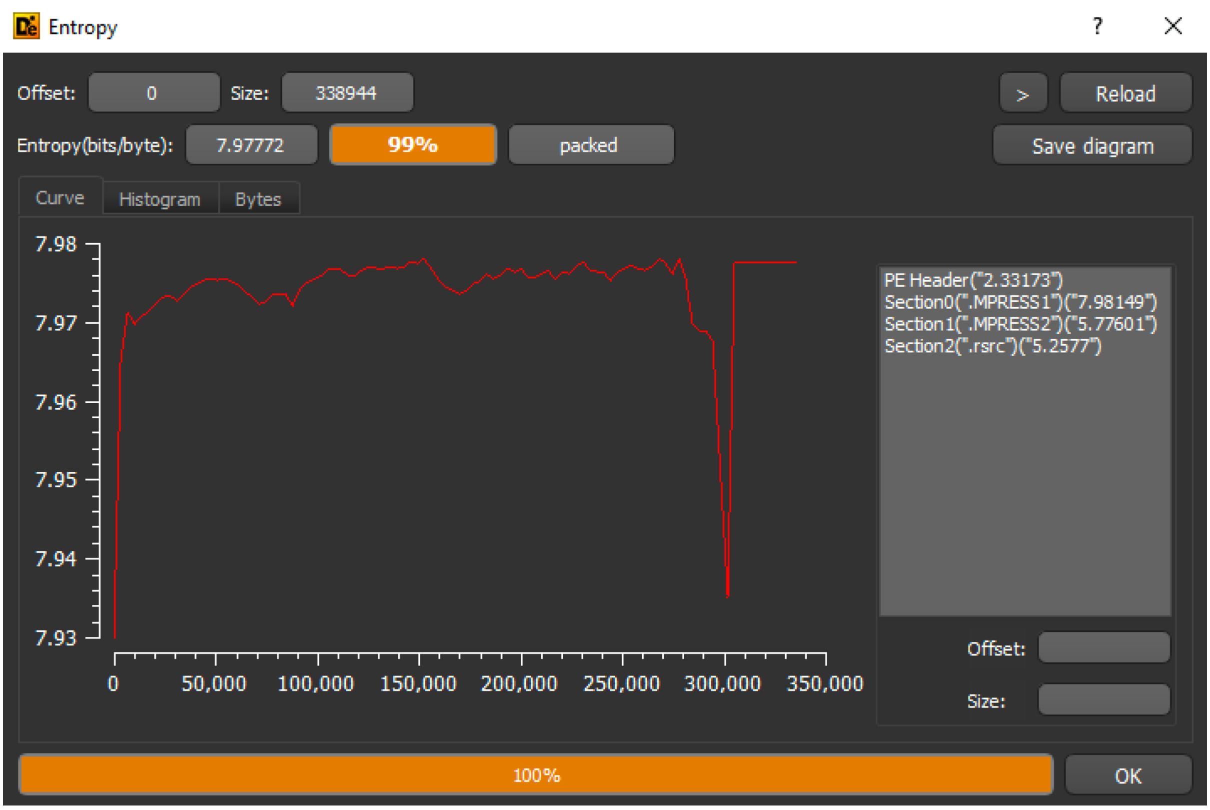


name	.MPRESS1	.MPRESS2	.rsrc
md5	B10AD7C66480C6DC389...	949220E8631B83C329C0...	1AFA8EC1F42595AE0543...
file-ratio	-	-	-
virtual-size (371100 bytes)	368640 bytes	1468 bytes	992 bytes
raw-size (338432 bytes)	335872 bytes	1536 bytes	1024 bytes
cave (100 bytes)	0 bytes	68 bytes	32 bytes
entropy	7.981	5.775	5.256
virtual-address	0x00001000	0x0005B000	0x0005C000
raw-address	0x00000200	0x00052200	0x00052800
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.

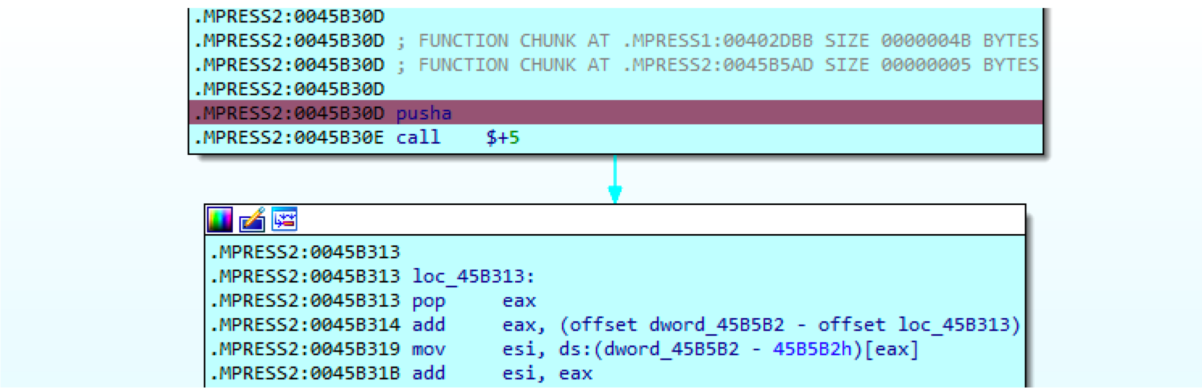




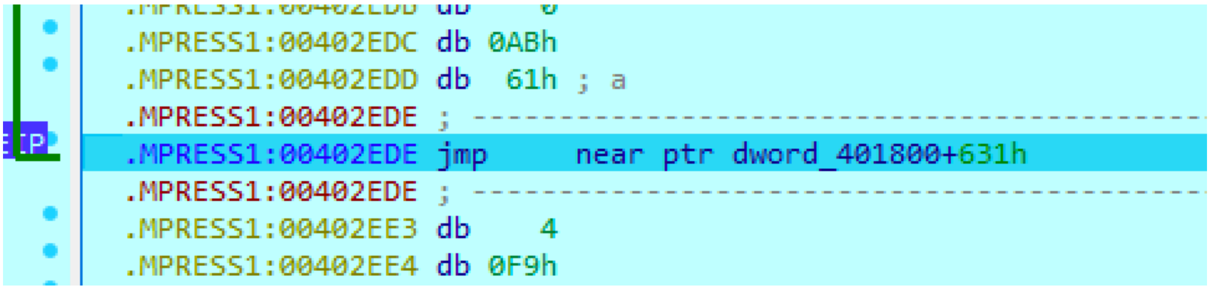
There are some indicators useful to recognize the tail jump that will allow us to fine the OEP:

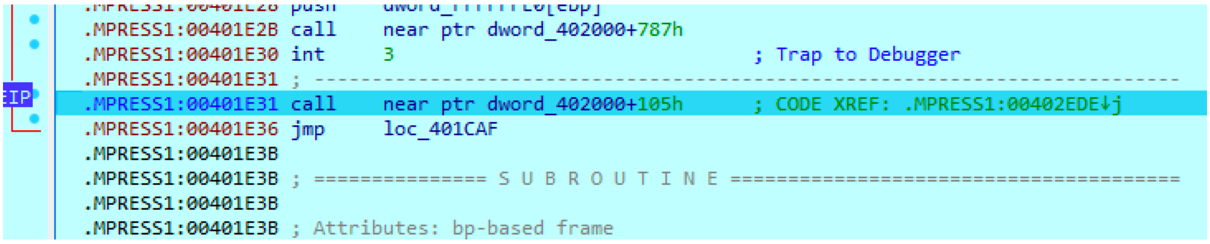
- The instruction jumps to another section (in this case from .MPRESS2 to .MPRESS1)
- After the tail jump should be a bunch of garbage bytes.
- The destination was previously modified by the unpacking stub

After opening the sample in IDA and starting at the entry point in .MPRESS2 (0x45B30D), 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 @ 0x402EDE
OEP @ 0x401E31

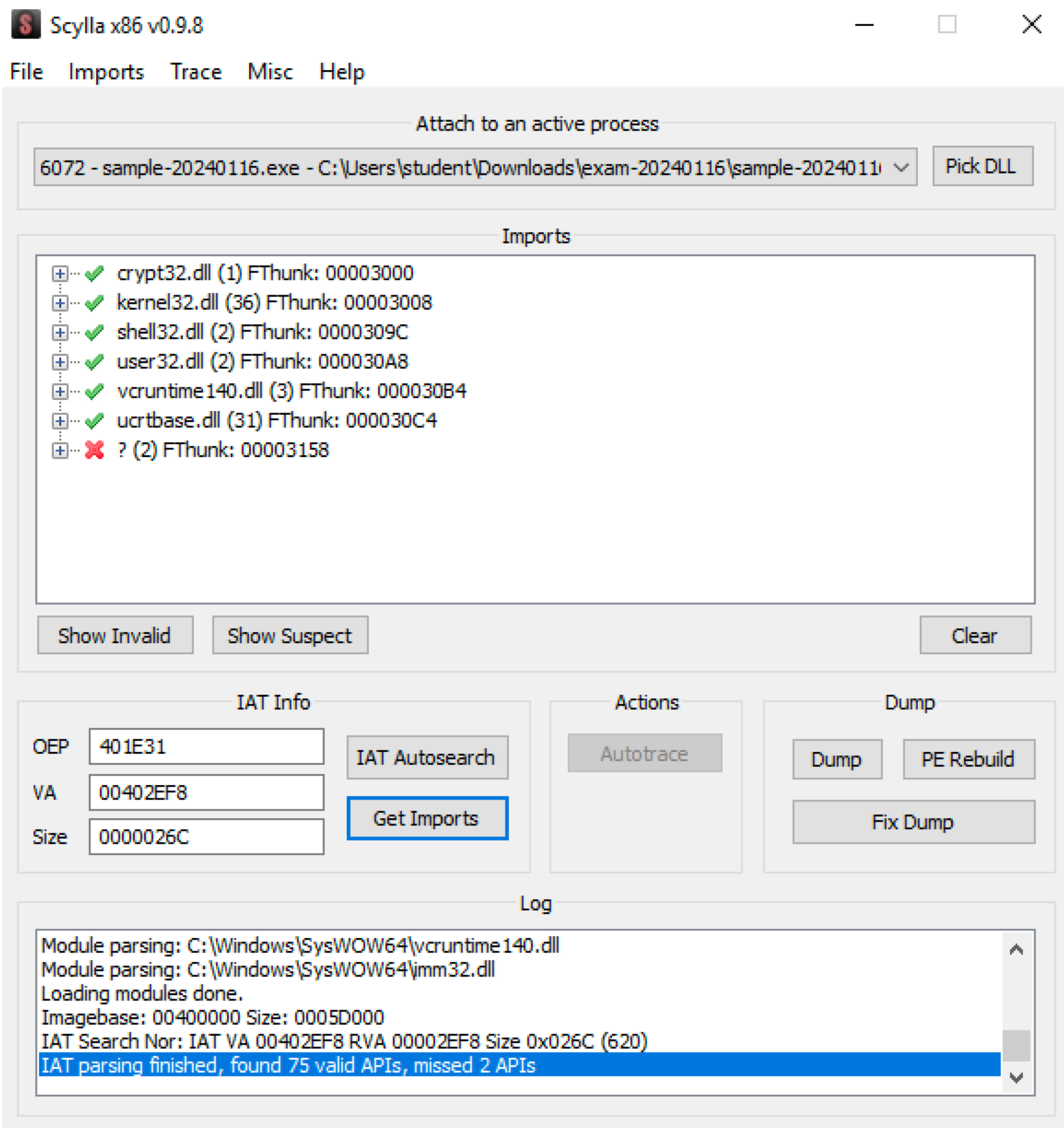


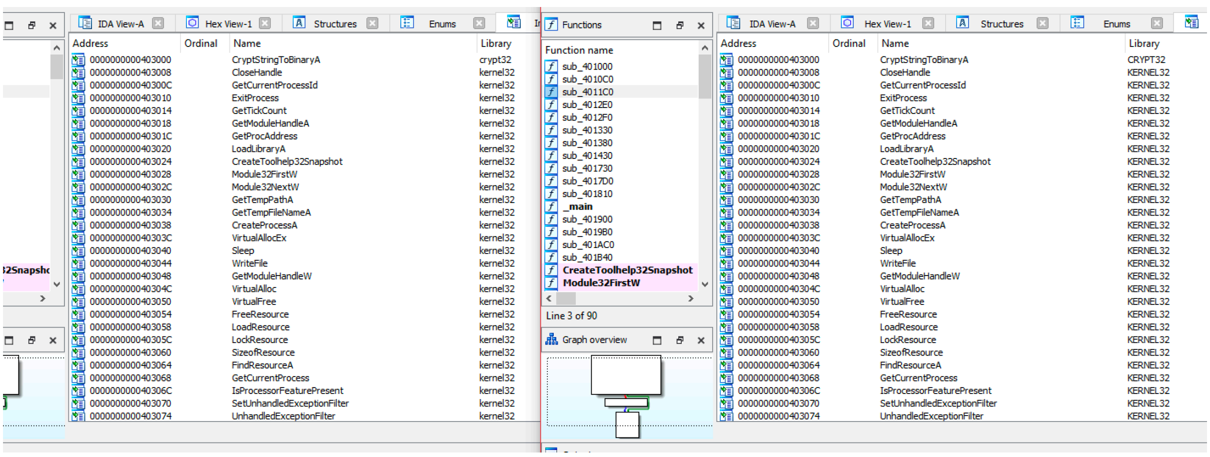


3 - Provide details about the IAT reconstruction process that you carried out to unpack the code. *<u>HINTS:</u> the answer should cover methodological aspects and facts on your output; also, validate it! (e.g., check API calls, compare with sample-XXXXXXXX-unpacked.exe).*

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;
- Comparing with imports of sample-20240116-unpacked in IDA we can see that the operation was successful





on the left the SCYLLA dump fixed, on the right the unpacked version

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.

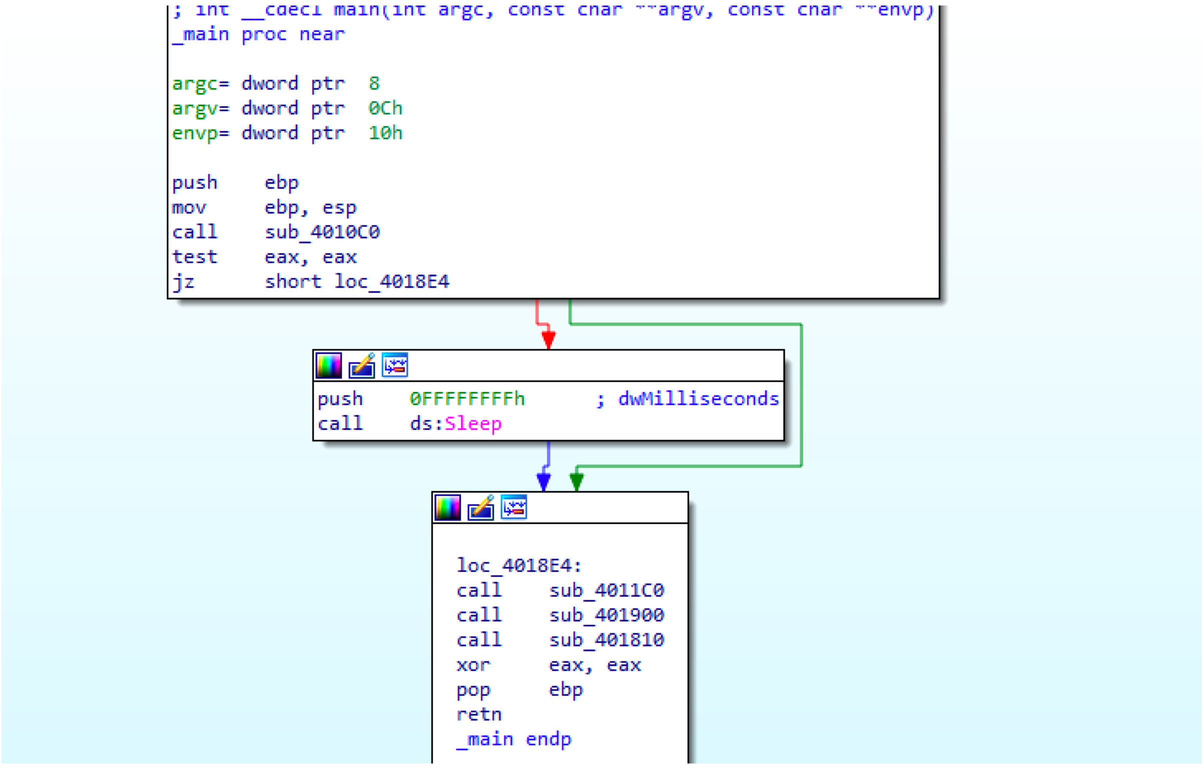
In general, the sample works as follows (for details see answer 6): It checks the milliseconds after startup to decide if goes to sleep. The sample create a copy of itself in the startup folder in this way it can survive after reboot. At the end, it execute explorer.exe and inject a dll in its portion of memory. The dll performs a conntection to a specific host on port 80.

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).

- Files:
  - $STARTUP_FOLDER\7z.exe (discovery method: IDA)
  - $PICTURES_FOLDER\ mafuba.jpg (discovery method: IDA)
- Processes:
  - explorer.exe (dll injection) (discovery method: IDA/process hacker)
- Network connections:
  - CC @ 34.136.147.60 (discovery method: IDA)

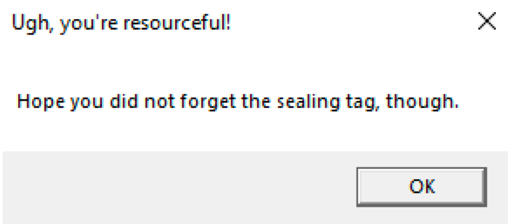
6 - List the subroutines used by the sample and its byproducts (e.g., injected payloads, second-stage executables), if any, to implement its main functionalities and provide a sketch of the execution transfers among them (e.g sketch a tree/graph). **NOTE**: listing such parts is optional only in the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

- main:

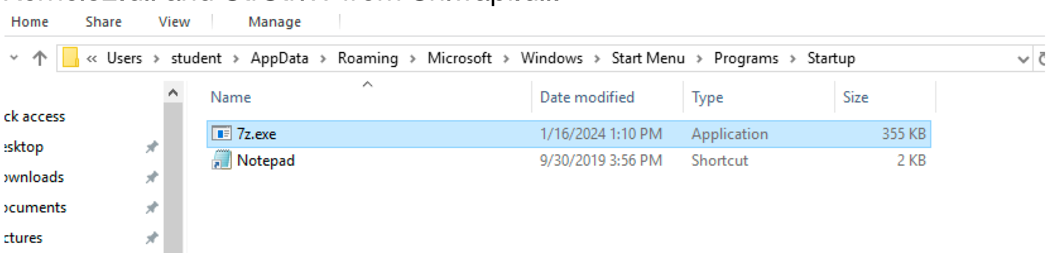


- sub_4010C0 (time check)
  - It uses GetModuleHandleA and GetProcAddress to retrieve GetTickCount from kernel32.dll. If the number of milliseconds is lower than 1800000, it goes to sleep for a long period of time (0FFFFFFFFh). Otherwise, it continuous the execution.
- sub_4011C0 (check module)
  - Iterate the modules of the process using CreateToolhelp32Snapshot, Module32FirstW and Module32NextW. In this iteration, it retrieves the function StrStrIW from Shlwapi.dll using LoadLibraryA and GetProcAddress. Then it checks if there is a match with the first occurrence of the following strings: “daimao”, “piccolo”, “zamasu”, “dercori”, “frost”. If the match is found, it executes the function sub_401000.
    - sub_401000
      - It uses the function SHGetFolderPathA to retrieve the path: C:\Users\student\Pictures
      - Then it stores the file mafuba.jpg in this path using CreateFileA and WriteFile.
      - The data are stored in unk_403190

- Finally printf a message box



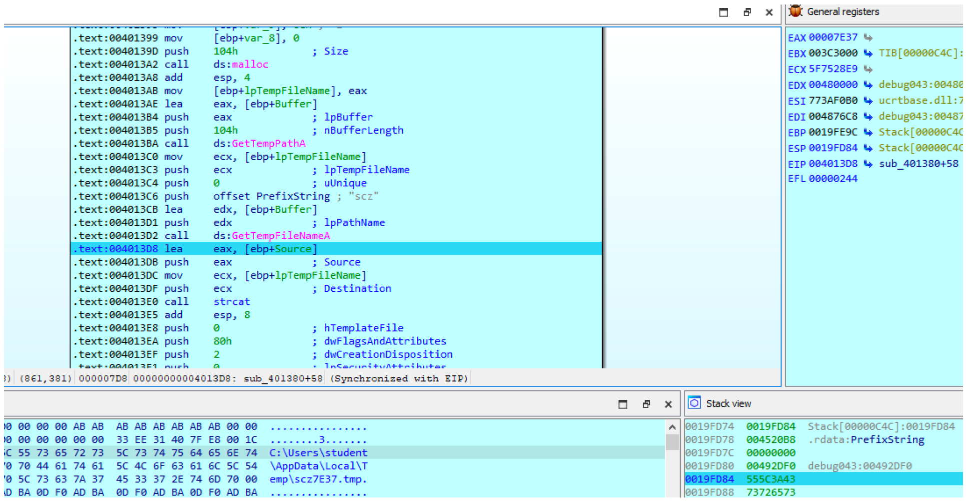
- sub_401900 (persistence)
  - It retrieves the current path: C:\Users\student\Downloads\exam-20240116\sample-20240116-unpacked.exe
    using SHGetFolderPathW and it creates a copy of itself in C:\Users\student\APPData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup and it calls the copy “7z.exe”. In this way, the malware will be automatically run at the startup of the machine. To perform this task the sample uses GetProcAddress obtains the address of several interesting functions like CopyFileW from Kernel32.dll and StrStrIW from Shlwapi.dll.



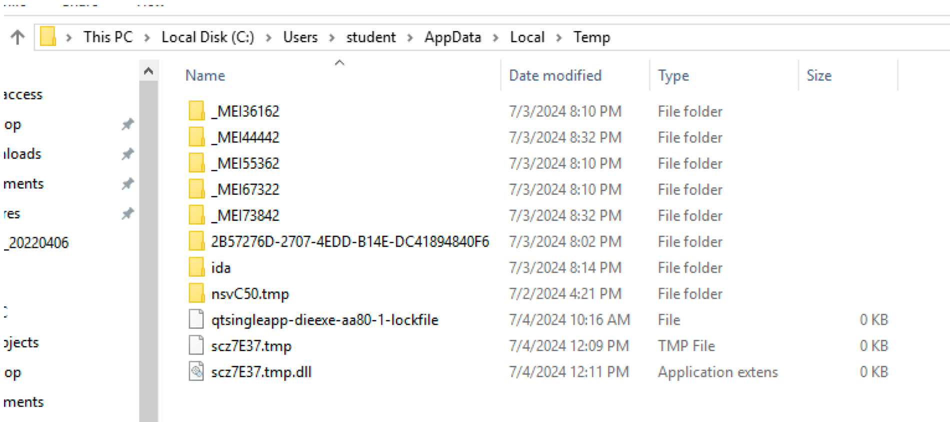
- sub_401810 (DLL injection)
  - The sample calls first the function sub_4017D0
    - In this function, it calls sub_4019B0 and it passes the name of the recourse “65” and the type “SCZ”. This is done using the
      functions: FindResourceA, LoadResource, LockResource and SizeofResource. Then it uses VirtualAlloc to allocate the

resourse in 0x001D0000.

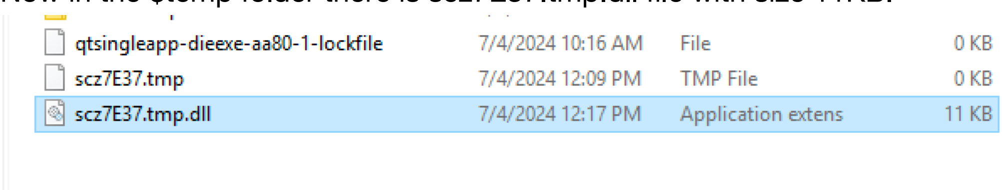
- Then the sample calls the function sub_401380
  - It uses the function GetTempPathA to retrieve the path to the directory designated for temporary files (C:\Users\student\AppData\Local\Temp).
  - It uses the function GetTempFileNameA to create a file in this path that has a name scz and then there is a unique numeric value (7E37) returned by GetTempFileNameA.



  - Then it calls CreateFileA to create this file in this directory.



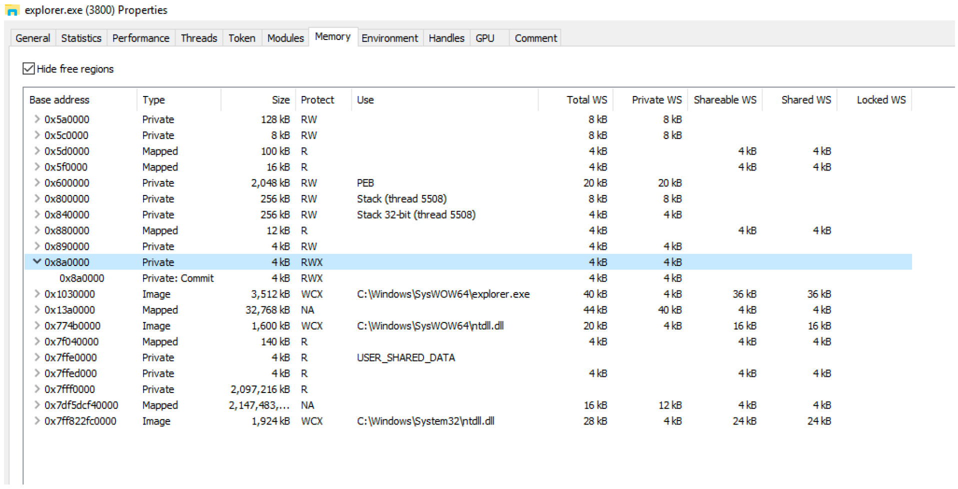
  - At the end, it calls WriteFileA to write 10752 bytes and the data are stored in 0x1D0000.
- Now in the $temp folder there is scz7E37.tmp.dll file with size 11KB.



- The sample calls CreateProcessA to create the process explorer.exe in suspended state (dwCreationFlags = 0x4)
- It calls sub_401730
  - In this function it calls sub_401430
    - It uses the function sub_401AC0 to decrypt the string
    - It retrieves the address of the function

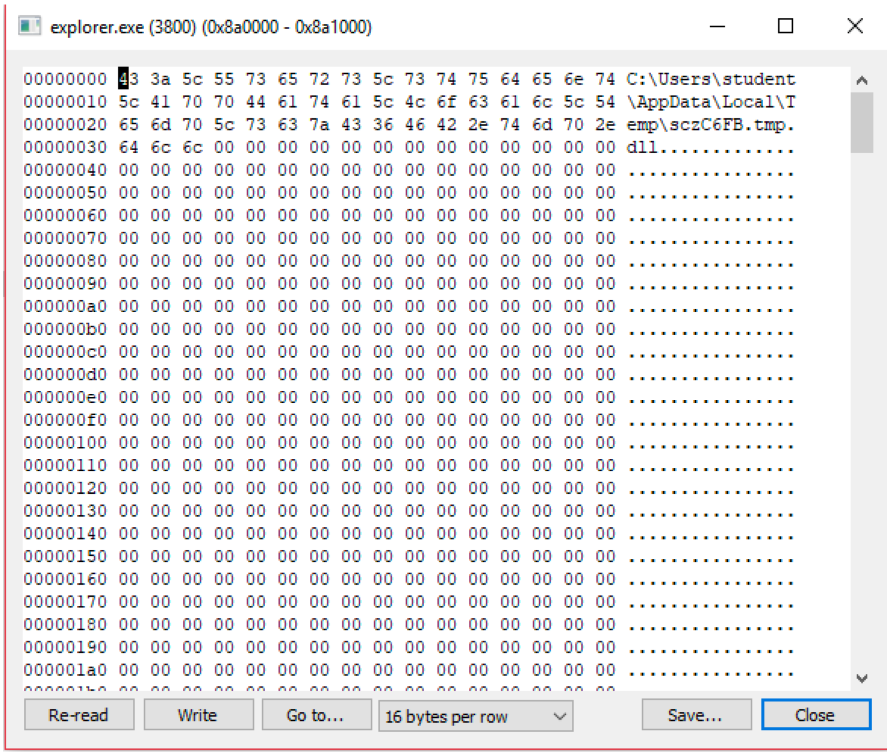
NtUnmapViewOfSection, SetThreadContext, Resume Thread, NtCreateThreadEx, SuspendThreadEx, VirtualAllocEx, GetThreadContext, WriteProcessMemory, LoadLibraryA using GetProcAddress. These functions are obfuscated and encrypted.

- It calls VirtualAllocEx to allocate memory in the process explorer.exe at memory address 0x8a0000 (this address change dynamically).



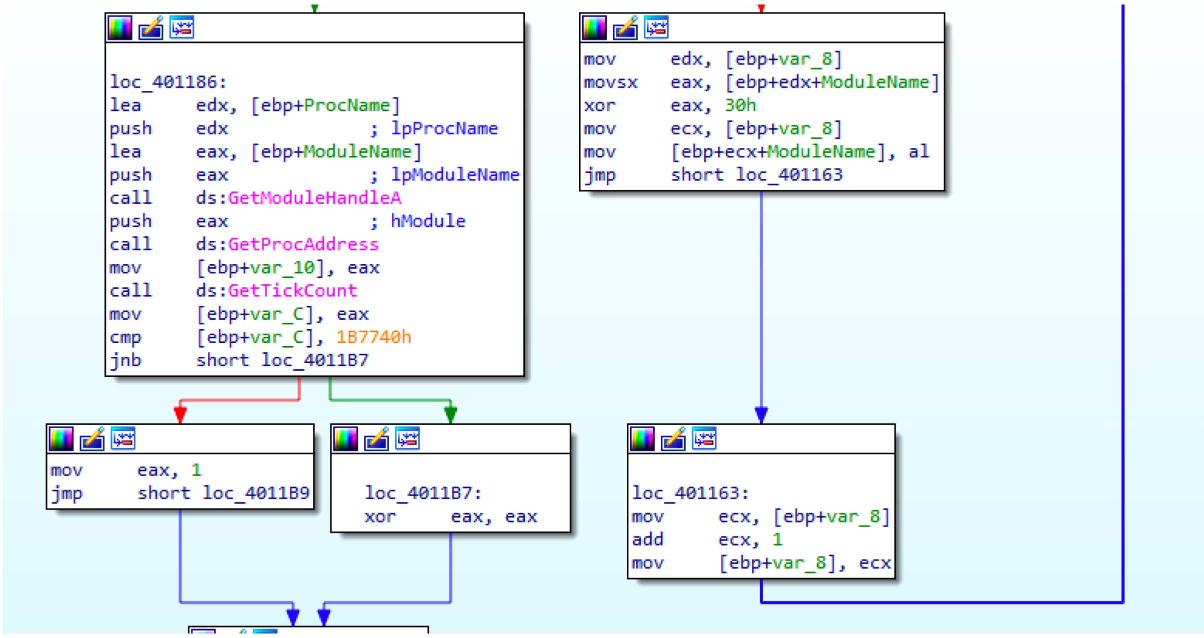
Base address	Type	Size	Protect	Use	Total WS	Private WS	Shareable WS	Shared WS	Locked WS
0x5a0000	Private	128 kB	RW		8 kB	8 kB			
0x5c0000	Private	8 kB	RW		8 kB	8 kB			
0x5d0000	Mapped	100 kB	R		4 kB		4 kB	4 kB	
0x5f0000	Mapped	16 kB	R		4 kB		4 kB	4 kB	
0x600000	Private	2,048 kB	RW	PEB	20 kB	20 kB			
0x800000	Private	256 kB	RW	Stack (thread 5508)	8 kB	8 kB			
0x840000	Private	256 kB	RW	Stack 32-bit (thread 5508)	4 kB	4 kB			
0x880000	Mapped	12 kB	R		4 kB		4 kB	4 kB	
0x890000	Private	4 kB	RW		4 kB	4 kB			
0x8a0000	Private	4 kB	RWX		4 kB	4 kB			
0x8a0000	Private: Commit	4 kB	RWX		4 kB	4 kB			
0x1030000	Image	3,512 kB	WCX	C:\Windows\SysWOW64\explorer.exe	40 kB	4 kB	36 kB	36 kB	
0x13a0000	Mapped	32,768 kB	NA		44 kB	40 kB	4 kB	4 kB	
0x774b0000	Image	1,600 kB	WCX	C:\Windows\SysWOW64\ntdll.dll	20 kB	4 kB	16 kB	16 kB	
0x7f040000	Mapped	140 kB	R		4 kB		4 kB	4 kB	
0x7ffe0000	Private	4 kB	R	USER_SHARED_DATA					
0x7ffed000	Private	4 kB	R		4 kB		4 kB	4 kB	
0x7fff0000	Private	2,097,216 kB	R						
0x7df5dcf40000	Mapped	2,147,483,...	NA		16 kB	12 kB	4 kB	4 kB	
0x7ff822fc0000	Image	1,924 kB	WCX	C:\Windows\System32\ntdll.dll	28 kB	4 kB	24 kB	24 kB	

- It calls WriteProcessMemory to write the scz.tmp.dll in this memory location



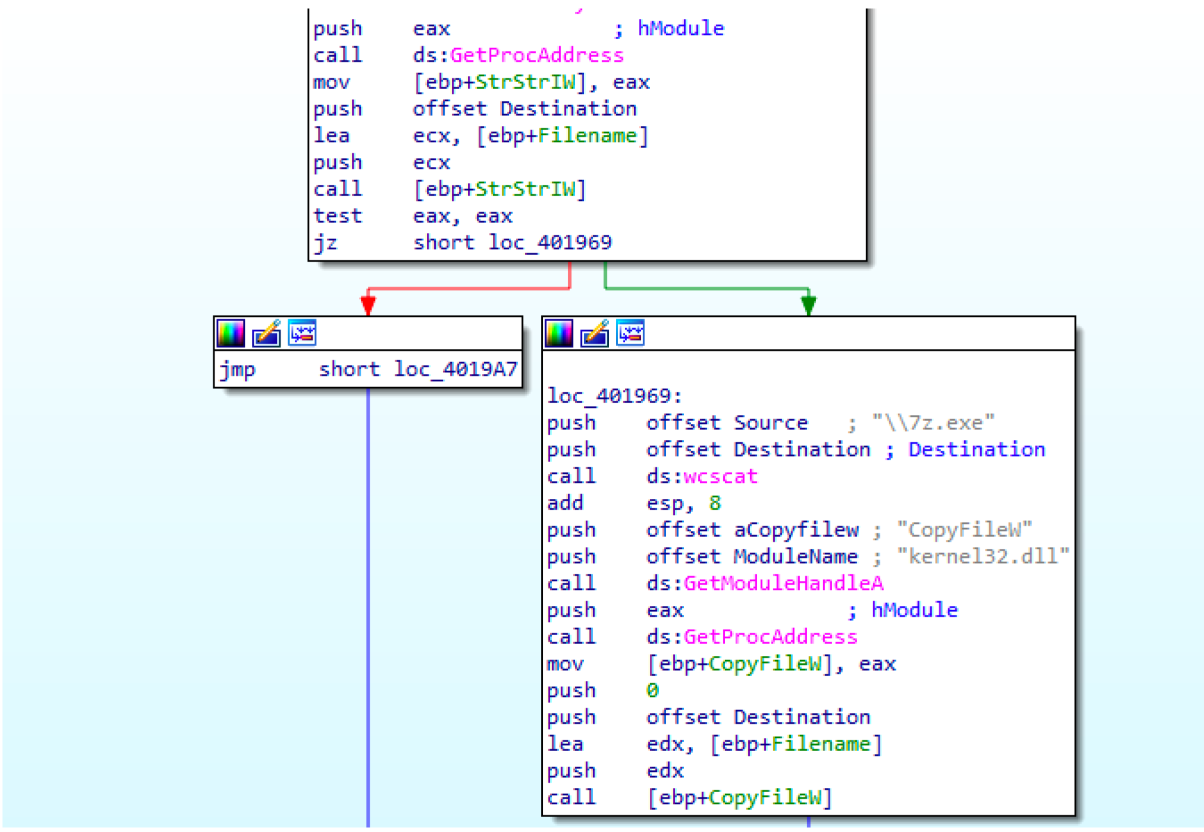
7 - Does the sample make queries about the surrounding environment before unveiling its activities? If yes, describe them and pinpoint specific instructions/functions in the code.

As show before in question 6, it checks the number of milliseconds after startup.



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. In this way, the malware will be automatically run at the startup of the machine.



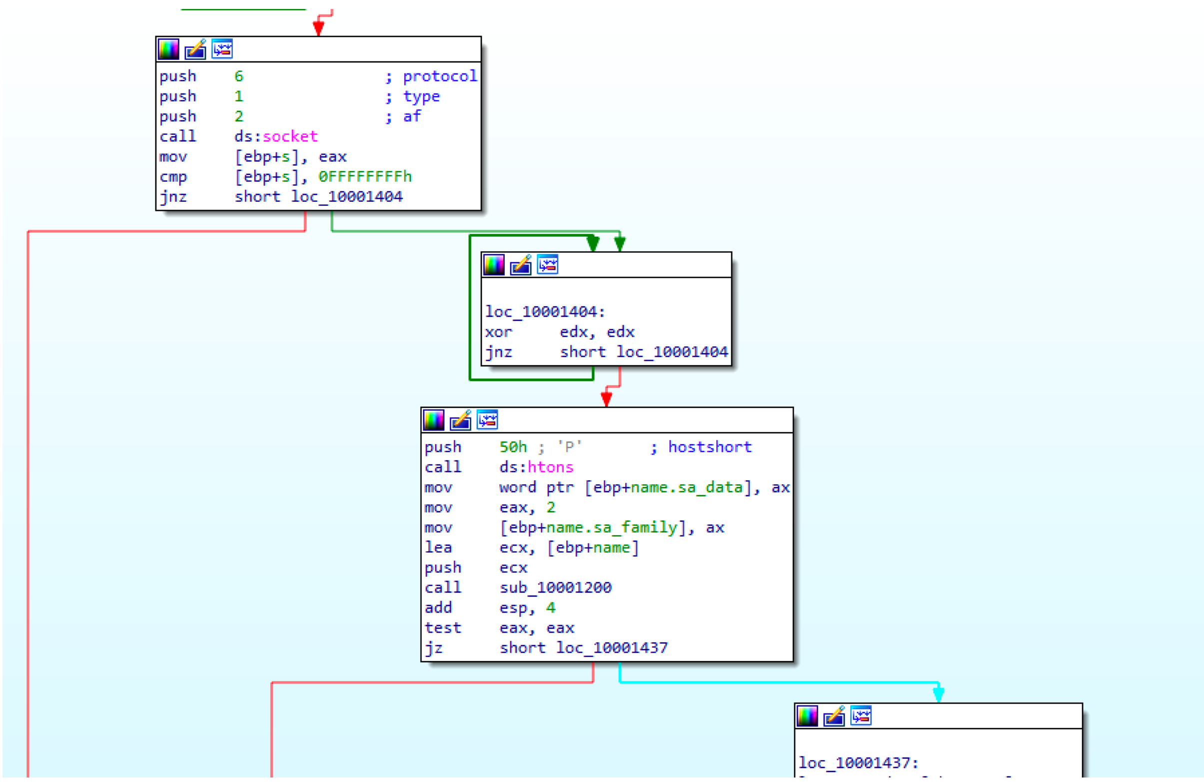
9 - Does the sample perform any code injection activities? Which kind of injection pattern do you recognize? Describe the characteristics and behavior of the injected payload, stating also where it is originally stored within the sample.

The sample performs dll injection. In fact, in function sub_401900 and retrieve the dll using the function sub_4017D0:

- it retrieves a dll scz<casual numbers>.tmp.dll and injected it into explorer.exe
- it loads, at run time, many functions, among which there are the ones necessary for dll injection (virtualAllocEx, WriteProcessMemory, NTCreateThreadEx, LoadLibraryA)
- it allocates memory to the victim process using VirtualAllocEx
- it writes the path to the dll with WriteProcessMemory on this allocated memory
- it calls NTCreateThreadEx passing LoadLibraryA as function to execute, so that the victim process will load the dll and execute its malicious code.

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 IpStartAddress. In this function we can see that the dll performs some functions. in fact it creates a socket and tries to connect to a specific ip address 34.136.147.60 on port 80. It calls the function sub_10001070 and calls CreateProcessA with the command “netstat -n” and send to the server the network configuration.



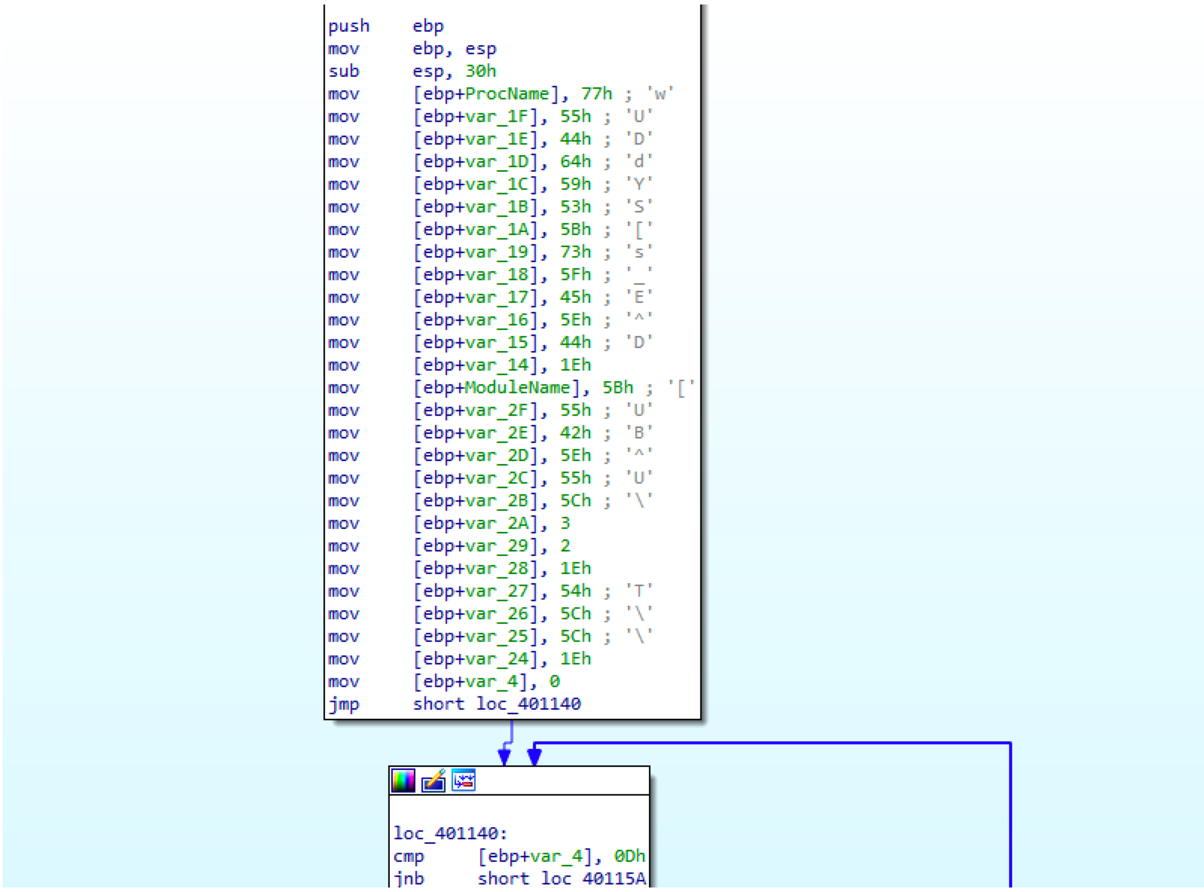
10 - Does the sample beacon an external C2? Which kind of beaconing does the malware use? Which information is sent with the beacon? Does the sample implement any communication protocol with the C2? If so, describe the functionalities implemented by the protocol.

The sample does not perform some specific communication protocol in which the server sends commands to the target. After connecting to the address 34.136.147.60 on port 80, it sends a beacon “TARGET HOST CONNECTIONS:\n%s\n”.

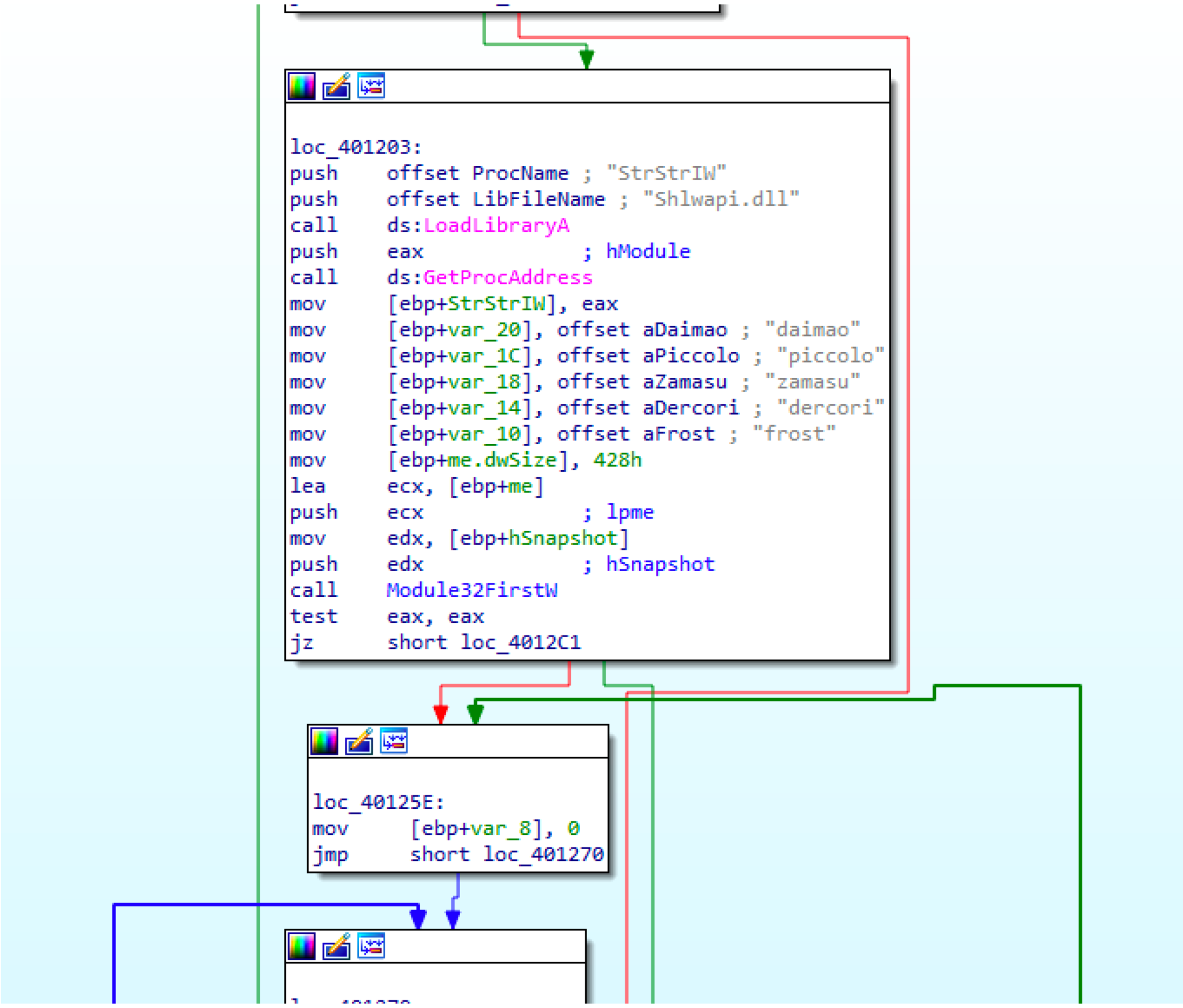
If the connection fails, it sends BYE\n and close the connection.

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. The encryption key is 30h



In function sub_4011C0, the function StrStrIW is loaded at runtime, to avoid it being visible among the import.



Something very similar happens, for the persistence function

```
lea     ecx, [ebp+Filename]
push    ecx
call    [ebp+StrStrIW]
test    eax, eax
jz      short loc_401969
```

```
jmp     short loc_4019A7
```

```
loc_401969:
push    offset Source    ; "\\7z.exe"
push    offset Destination ; Destination
call    ds:wcscat
add     esp, 8
push    offset aCopyfilew ; "CopyFileW"
push    offset ModuleName ; "kernel32.dll"
call    ds:GetModuleHandleA
push    eax              ; hModule
call    ds:GetProcAddress
mov     [ebp+CopyFileW], eax
push    0
push    offset Destination
lea     edx, [ebp+Filename]
push    edx
call    [ebp+CopyFileW]
```

```
loc_4019A7:
mov     esp, ebp
pop     ebp
retn
sub_401900 endp
```

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_4017D0
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
xor     edx, edx
mov     [ebp+ProcessInformation.hProcess], edx
```

In the sub_401380, the string “.dll” is pushed byte-per-byte on stack

```
lpBuffer= dword ptr  8
nNumberOfBytesToWrite= dword ptr  0Ch

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]
push    ecx             ; lpTempFileName
push    0               ; uUnique
push    offset PrefixString ; "scz"
lea     edx, [ebp+Buffer]
push    edx             ; lpPathName
call    ds:GetTempFileNameA
lea     eax, [ebp+Source]
push    eax             ; Source
mov     ecx, [ebp+lpTempFileName]
push    ecx             ; Destination
call    strcat
add     esp, 8
push    0               ; hTemplateFile
push    80h             ; dwFlagsAndAttributes
push    2               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    0               ; dwShareMode
push    40000000h       ; dwDesiredAccess
mov     edx, [ebp+lpTempFileName]
```