

**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](mailto:querzoni@diag.uniroma1.it) and [delia@diag.uniroma1.it](mailto: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

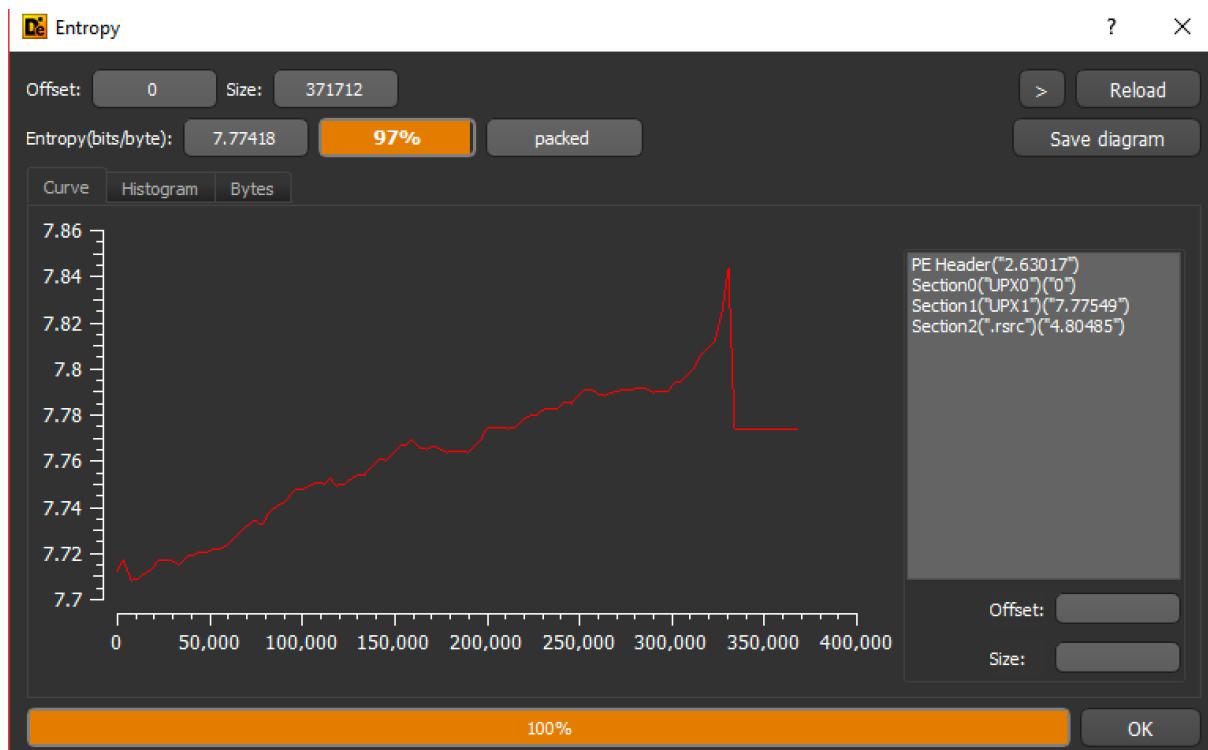
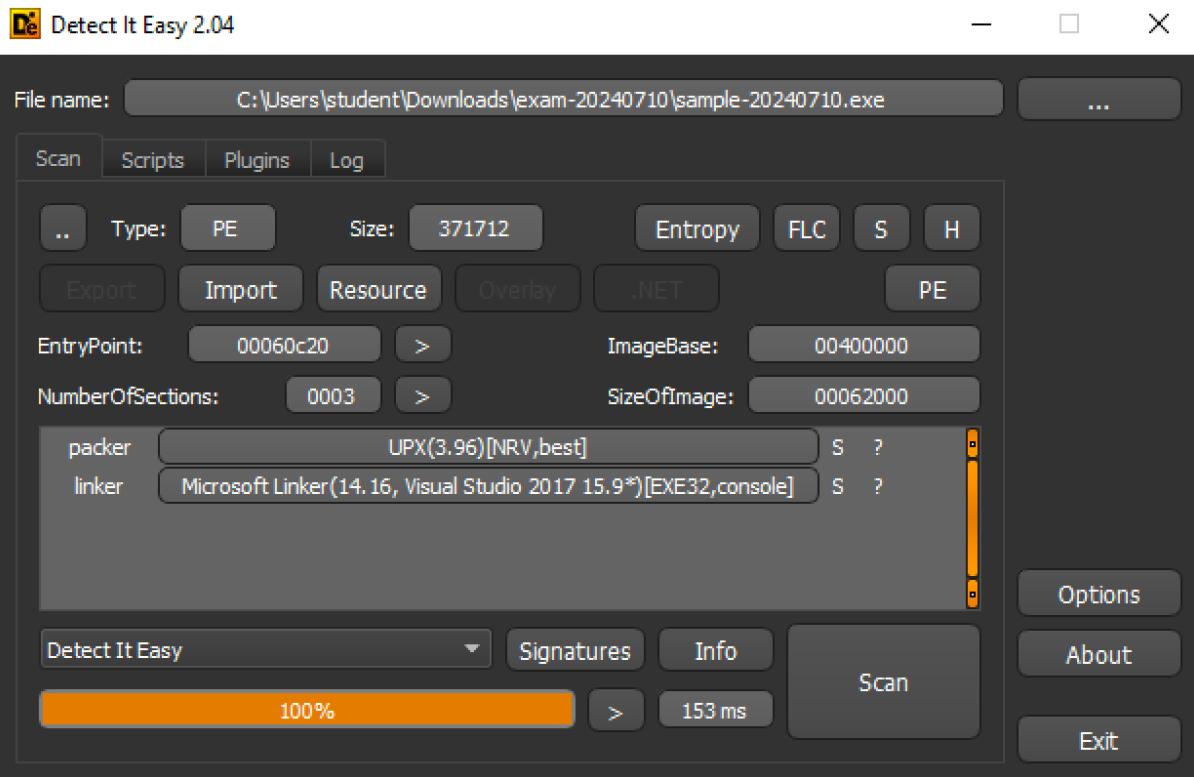
- The *imports* section contains some functions that are potential indicators or could reveal the sample's behavior. They are the following:
  - GetProcAddress, LoadLibraryA which are typical of packed software that has to rebuild the IAT
  - MessageBoxA, which means that the sample shows some message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

property	value	value	value
name	UPX0	UPX1	.rsrc
md5	n/a	1ED77268F5609862E540E...	5C3FA85F441D6F3C8BD...
file-ratio	-	-	-
virtual-size (397312 bytes)	24576 bytes	368640 bytes	4096 bytes
raw-size (370688 bytes)	0 bytes	368640 bytes	2048 bytes
cave (0 bytes)	0 bytes	0 bytes	0 bytes
entropy	7.775	4.804	-
virtual-address	0x00001000	0x00007000	0x00061000
raw-address	0x00000400	0x00000400	0x0005A400
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
uninitialized-data	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 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.

U	Log	Notes	Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols	Source
X	EDX	ESI	EDI						
				00460C20	60	pushad			EntryPoint
				00460C21	BE 00704000	mov esi,sample-20240710.407000			esi:EntryPoint
				00460C26	8DBE 00A0FFFF	lea edi,dword ptr ds:[esi-6000]			edi:EntryPoint
				00460C2C	57	push edi			edi:EntryPoint
				00460C2D	EB 0B	jmp sample-20240710.460C3A			

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

●	00460DB6	8D4424 80	lea eax,dword ptr ss:[esp-80]
●	00460DBA	6A 00	push 0
●	00460DBC	39C4	cmp esp,eax
●	00460DBE	^ 75 FA	jne sample-20240710.460DBA
●	00460DC0	83EC 80	sub esp,FFFFFF80
●	00460DC3	^ E9 6910FAFF	jmp sample-20240710.401E31
●	00460DC8	A0 00000000	mov al,byte ptr ds:[0]
●	00460DCD	0000	add byte ptr ds:[eax],a

J	Log	Notes	Breakpoints	Memory Map	Call Stack	SEH	Script	Symbols	Source
				00401E31	E8 CF020000	call sample-20240710.402105			
				00401E36	^ E9 74FFFF	jmp sample-20240710.401CAF			
				00401E38	55	push ebp			
				00401E3C	8BEC	mov ebp,esp			
				00401E3E	8B45 08	mov eax,dword ptr ss:[ebp+8]			
				00401E41	56	push esi			esi:Ent
				00401E42	8B48 3C	mov ecx,dword ptr ds:[eax+3C]			ecx:Ent
				00401E45	03C8	add ecx, eax			ecx:Ent

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.

Scylla x86 v0.9.8

File Imports Trace Misc Help

Attach to an active process  
6712 - sample-20240710.exe - C:\Users\student\Downloads\exam-20240710\sample-20240710 | Pick DLL

Imports

- + crypt32.dll (1) FThunk: 00003000
- + kernel32.dll (36) FThunk: 00003008
- + shell32.dll (2) FThunk: 0000309C
- + user32.dll (2) FThunk: 000030A8
- + vcruntime140.dll (3) FThunk: 000030B4
- + ucrtbase.dll (31) FThunk: 000030C4
- + ? (1) FThunk: 00003158

Show Invalid Show Suspect Clear

IAT Info Actions Dump

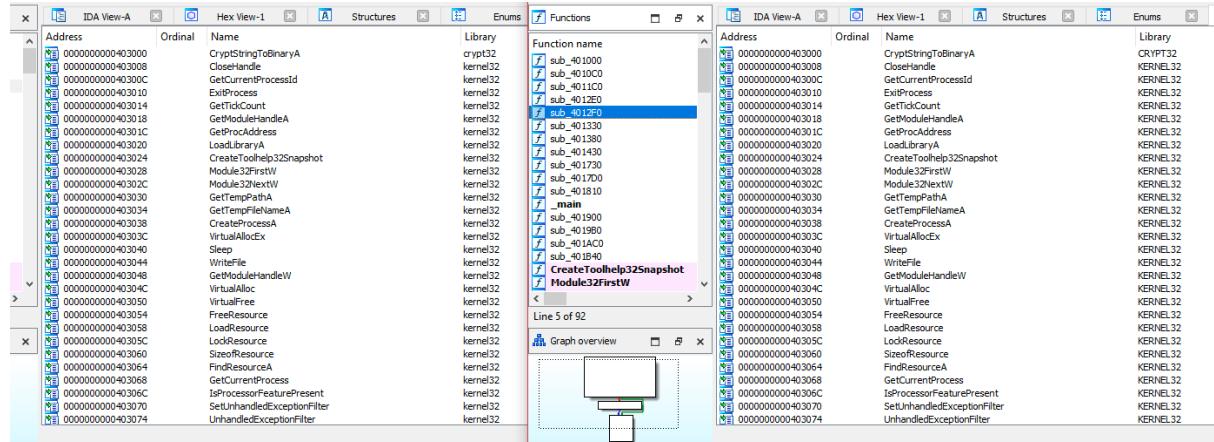
OEP	00401E31	IAT Autosearch	Autotrace	Dump
VA	00403000			PE Rebuild
Size	0000015C	Get Imports		Fix Dump

Log

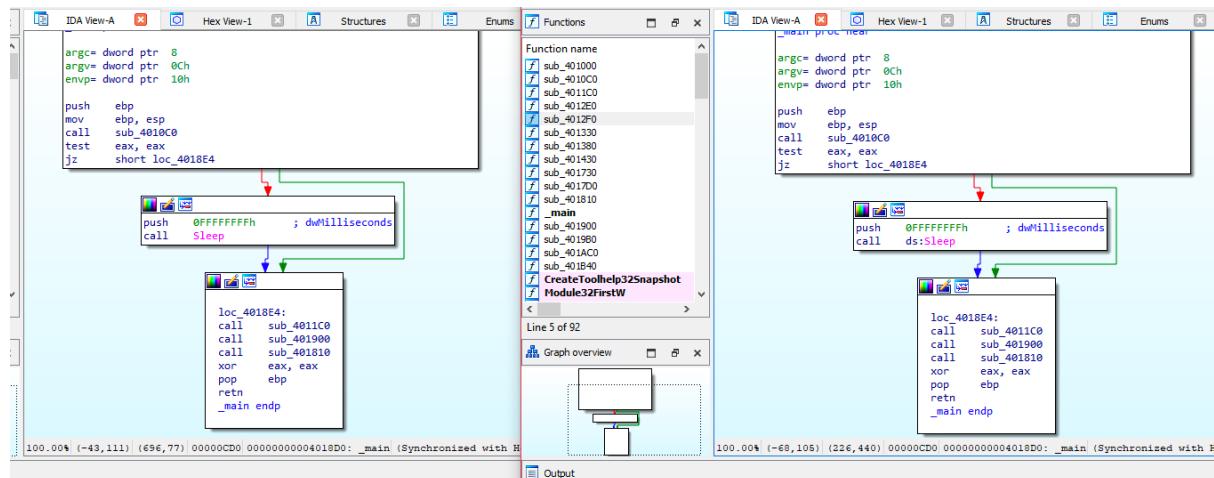
```
IAT Search Adv: Found 76 (0x4C) possible IAT entries.  
IAT Search Adv: Possible IAT first 00403000 last 00403158 entry.  
IAT Search Adv: IAT VA 00403000 RVA 00003000 Size 0x015C (348)  
IAT Search Nor: IAT VA 004027EC RVA 000027EC Size 0x0978 (2424)  
IAT parsing finished, found 75 valid APIs, missed 1 APIs  
DIRECT IMPORTS - Found 0 possible direct imports with 0 unique APIs!
```

Imports: 76 Invalid: 1 Imagebase: 00400000 sample-20240710.exe ...

- 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):

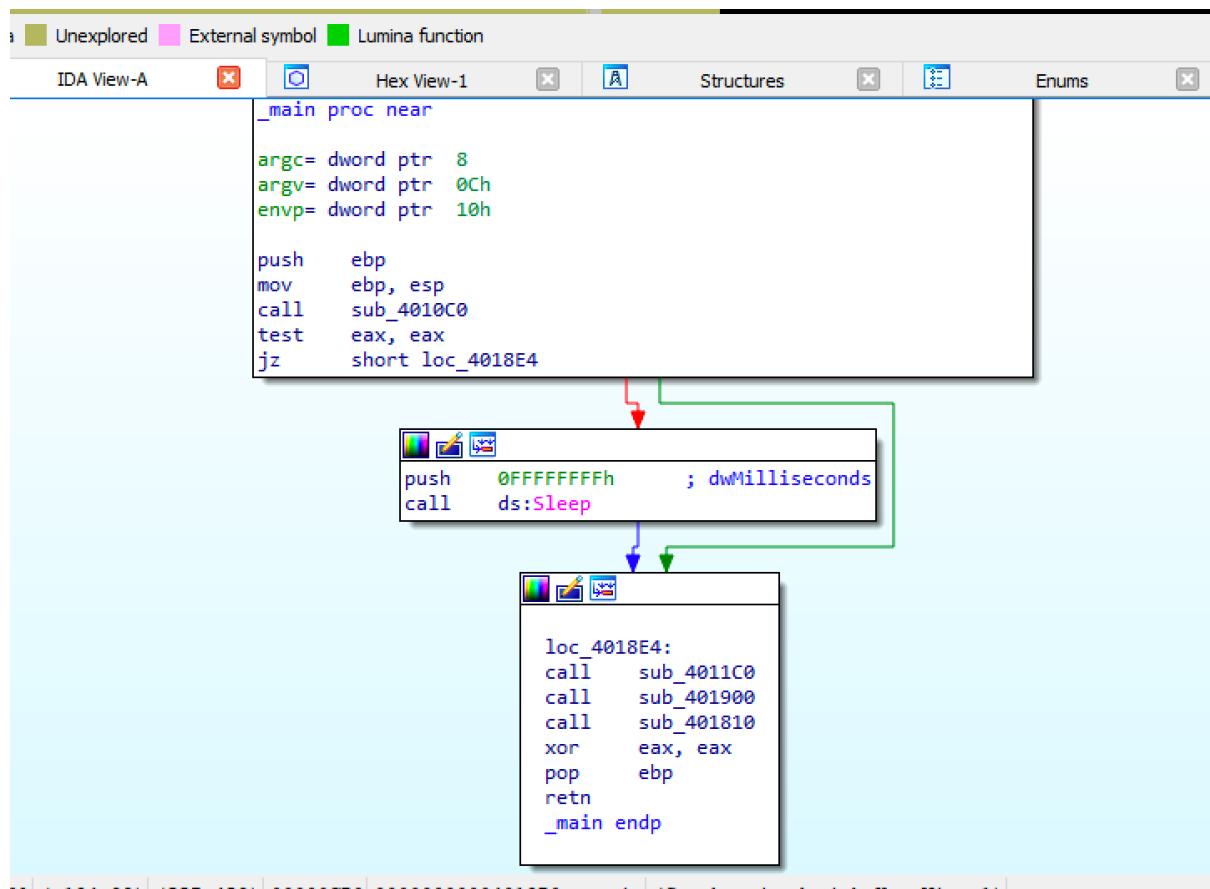
1. The sample checks if the system is running since a short time (1800000 ms). If so, it goes to sleep for a long period of time.
2. The sample checks if it has one or more modules with a specific name. If so, it exits after having changed the wallpaper.
3. The sample copies itself in the startup folder, 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 **0xXXXXXXXX**. Code at **0xXXXXXXXX** 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 (0xFFFFFFFFh)

### **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 finds 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

```

3 00  ..@.....'
2 73  .....C:\Users
2 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:0040105A cd11  sub_4012F0
.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
.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
.ds:WriteFile
.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)

```

---

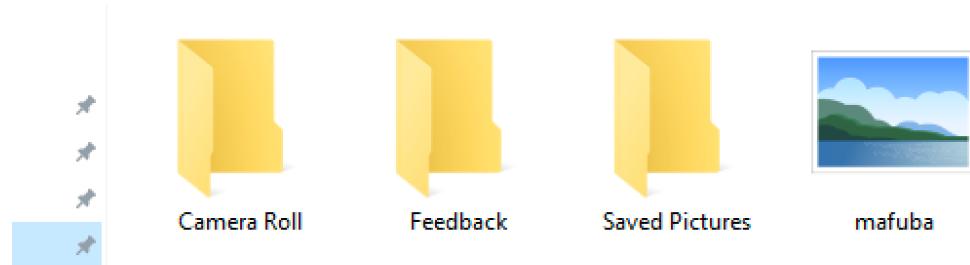
```

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 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

is PC > Pictures





- o 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.

C:\Users\student\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup				
	Name	Date modified	Type	Size
	7z	7/10/2024 2:35 PM	Application	371 KB
	Notepad	9/30/2019 3:56 PM	Shortcut	2 KB

#### **sub\_401810 (dll injection)**

- It first calls the function sub\_4017D0:
  - o 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”.

```

.text:004013CB lea    edx, [ebp+Buffer]
.text:004013D1 push   edx          ; lpPathName
.text:004013D2 call   ds:GetTempFileNameA
.text:004013D8 lea    eax, [ebp+Source]
.text:004013DB push   eax          ; Source
.text:004013DC mov    ecx, [ebp+lpTempFileName]
.text:004013DF push   ecx          ; Destination
.text:004013E0 call   strcat
.text:004013E5 add    esp, 8
.text:004013E8 push   0             ; hTemplateFile
.text:004013EA push   80h           ; dwFlagsAndAttributes
.text:004013EF push   2             ; dwCreationDisposition
.text:004013F1 push   0             ; lpSecurityAttributes
.text:004013F3 push   0             ; dwShareMode

```

476) (697, 291) 000007D2|00000000004013D2: sub\_401380+52 (Synchronized with EIP)

```

E 91 77 00 00 50 00 00 00 00 00 48 33 51 00 .N.w..P....H3Q.
5 50 00 80 F0 61 76 3E 4E 91 77 24 00 05 01 .vP.....N.w$...
0 19 00 00 00 50 00 48 00 0A 02 48 33 51 00 .....P.H...H3Q.
2 B9 A9 84 FD 19 00 B8 20 45 00 00 00 00 00 .....E....
0 51 00 43 3A 5C 55 73 65 72 73 5C 73 74 75 ...C:\Users\stu
5 6E 74 5C 41 70 70 44 61 74 61 5C 4C 6F 63 dent\AppData\Loc
2 5C 54 65 6D 70 5C 00 01 40 00 93 22 AB 77 al\Temp\..\@.."w
0 00 00 00 40 00 00 00 40 00 00 00 F0 05 00 .....@.....

```

- 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).

scz8DDU.dll	11 KB
scz52A9.tmp	0 KB
scz52A9.dll	0 KB

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

```

.text:004013F3 push   0             ; dwShareMode
.text:004013F5 push   40000000h       ; dwDesiredAccess
.text:004013FA mov    edx, [ebp+lpTempFileName]
.text:004013FD push   edx          ; lpFileName
.text:004013FE call   ds>CreateFileA
.text:00401404 mov    [ebp+hFile], eax
.text:00401407 push   0             ; lpOverlapped
.text:00401409 lea    eax, [ebp+NumberOfBytesWritten]

```

(390, 289) 000007FE|00000000004013FE: sub\_401380+7E (Synchronized with EIP)

```

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00 00 00 AB AB AB AB AB AB AB AB 00 00 ..... .
00 00 00 00 00 B9 99 32 73 C0 79 00 1C ..... 2s....
55 73 65 72 73 5C 73 74 75 64 65 6E 74 C:\Users\student
70 44 61 74 61 5C 4C 6F 63 61 6C 5C 54 \AppData\Local\T
5C 73 63 7A 35 32 41 39 2E 74 6D 70 2E emp\scz52A9.tmp.
00 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA dll..... .

```

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

SCZ52A9.tmp	11 KB
scz52A9.tmp.dll	0 KB
C:\Windows\system32\cmd.exe	0 KB

- 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.

The screenshot shows a debugger interface with two main panes. The top pane displays assembly code:

```

xt:0040178A mov     ecx, [ebp+dwSize]
xt:0040178D push    ecx
xt:0040178E mov     edx, [ebp+buffer]
xt:00401791 push    edx
xt:00401792 mov     eax, [ebp+lpBaseAddress]
xt:00401795 push    eax
xt:00401796 mov     ecx, [ebp+hProcess]
xt:00401799 push    ecx
xt:0040179A call    [ebp+WriteProcessMemory]
xt:0040179D test    eax, eax
xt:0040179F jnz     short loc_4017A3

```

The instruction at address 0x0040178E, `sub_401730+5E`, is highlighted in yellow and labeled "(Synchronized with EIP)". Below the assembly code, the bottom pane shows a memory dump of the allocated buffer:

D F0 AD BA 0D F0 AD BA	.....
D F0 AD BA 0D F0 AD BA	.....
D F0 AD BA 0D F0 AD BA	.....
0 00 00 00 00 00 00 00	.....
3 A 5C 55 73 65 72 73	ä..;hD...C:\Users
C 41 70 70 44 61 74 61	\student\AppData
5 6D 70 5C 73 63 7A 36	\Local\Temp\scz6
4 6C 6C 00 0D F0 AD BA	AA2.tmp.dll.....
D F0 AD BA 0D F0 AD BA	.....

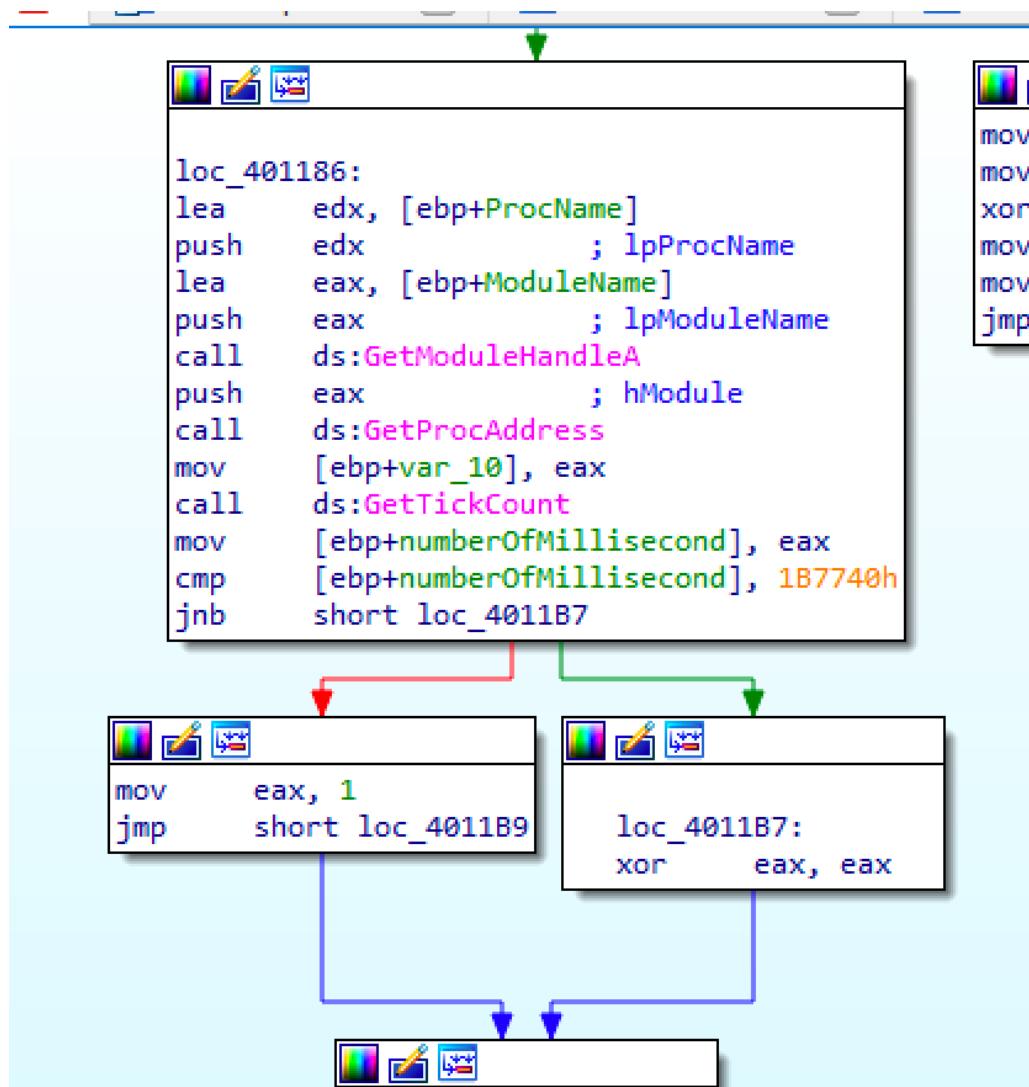
- Then it calls WriteProcessMemory to inject the malicious code stored in scz<random number>.tmp.dll at the memory address 0x3080000 into explorer.exe.

The screenshot shows a debugger's memory dump window for the process explorer.exe (6504). The title bar reads "explorer.exe (6504) (0x3080000 - 0x3081000)". The main area displays memory starting at address 00000000. The assembly code is mostly zeros, with some instruction bytes visible. The memory dump shows a large block of zeros. The bottom of the window has several buttons: "Re-read", "Write", "Go to...", "16 bytes per row" (with a dropdown arrow), "Save...", and "Close".

- 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 shown before in question 6, the sample, before performing its malicious activities, checks 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_401203:
push    offset ProcName ; "StrStrIW"
push    offset LibFileName ; "Shlwapi.dll"
call    ds:LoadLibraryA
push    eax           ; hModule
call    ds:GetProcAddress
mov     [ebp+StrStrIW], eax
mov     [ebp+var_20], offset aDaimao ; "daimao"
mov     [ebp+var_1C], offset aPiccolo ; "piccolo"
mov     [ebp+var_18], offset aZamasu ; "zamasu"
mov     [ebp+var_14], offset aDercori ; "dercori"
mov     [ebp+var_10], offset aFrost ; "frost"
mov     [ebp+me.dwSize], 428h
lea     ecx, [ebp+me]
push    ecx           ; lpme
mov     edx, [ebp+hSnapshot]
push    edx           ; hSnapshot
call    Module32FirstW
test   eax, eax
jz     short loc_4012C1
```

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.

The screenshot shows a debugger interface with two windows. The top window is titled 'Hex View-1' and contains tabs for 'Structures' and 'Enum'. Below these tabs is a toolbar with icons for file operations. A specific memory location, 'loc\_4019A7', is highlighted in blue. The assembly code for the routine at loc\_401969 is displayed:

```
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]
```

A blue bracket is drawn under the assembly code, indicating the scope of the question. The bottom window is also a hex viewer, showing a continuation of the memory dump.

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 injection (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.

```

push    3000h          ; flAllocationType
mov     edx, [ebp+dwSize]
push    edx          ; dwSize
push    0           ; lpAddress
mov     eax, [ebp+hProcess]
push    eax          ; hProcess
call   ds:VirtualAllocEx
mov     [ebp+lpBaseAddress], eax
push    0
mov     ecx, [ebp+dwSize]
push    ecx
mov     edx, [ebp+buffer]
push    edx
mov     eax, [ebp+lpBaseAddress]
push    eax
mov     ecx, [ebp+hProcess]
push    ecx
call   [ebp+WriteProcessMemory]
test   eax, eax
jnz    short loc_4017A3

```

jmp short loc\_4017C7

loc\_4017A3:

```

push    0
push    0
push    0
push    0
push    0
mov     edx, [ebp+lpBaseAddress]
push    edx
mov     eax, [ebp+var_C]
push    eax
mov     ecx, [ebp+hProcess]
push    ecx
push    0
push    10000000h
lea     edx, [ebp+NTCreateThreadEx]
push    edx
call   [ebp+var_10]

```

1 - loc\_4017C7

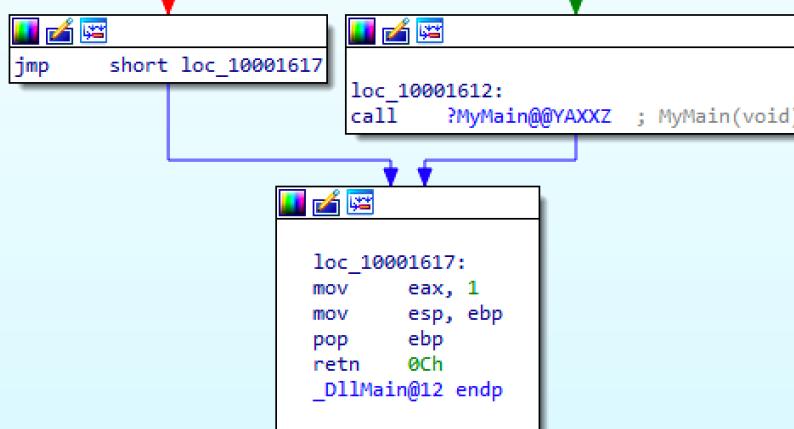
When we open the dll file, the code is this:

```

hinstDLL= dword ptr  8
fdwReason= dword ptr  0Ch
lpvReserved= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
mov     eax, [ebp+fdwReason]
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 1
jz      short loc_10001612

```



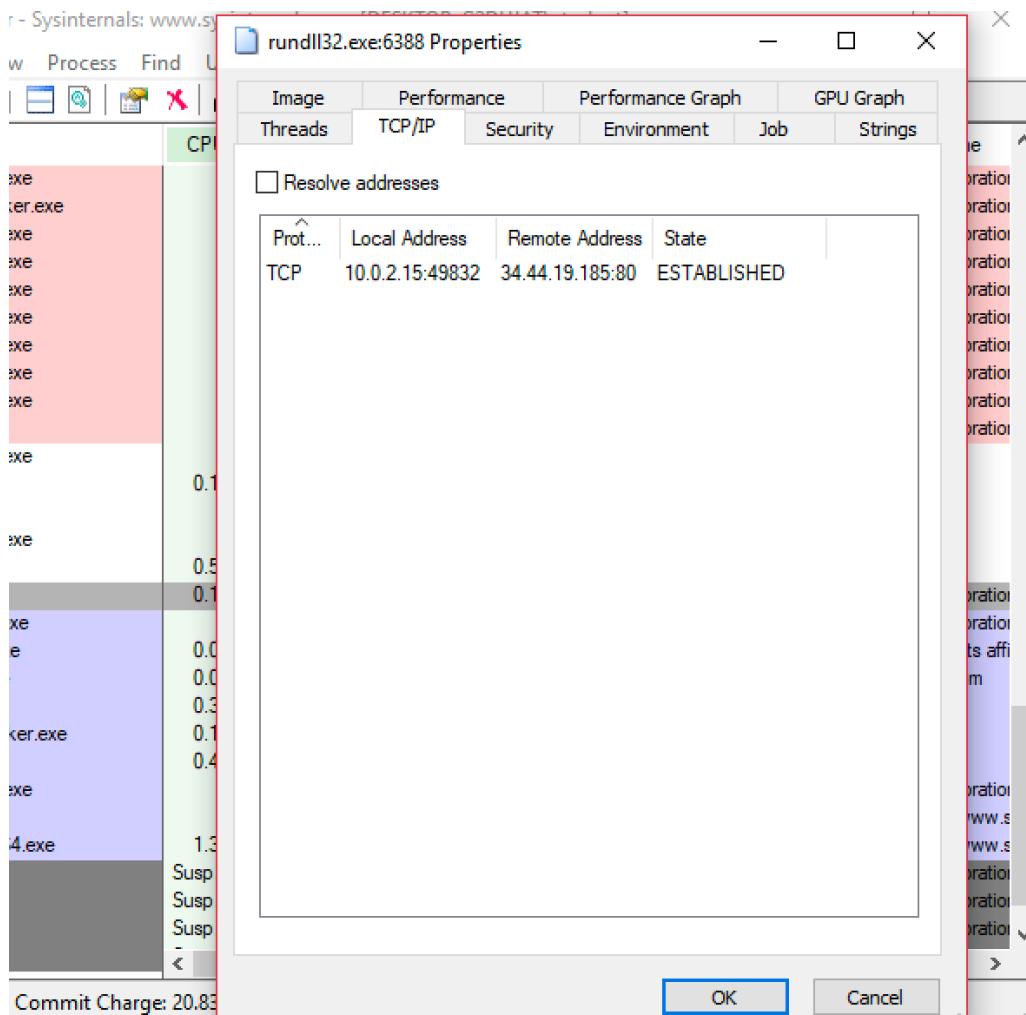
The injected dll behaves as follow:

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`.

The operations done in `IpStartAddress`, 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
7	24.651150	10.0.2.15	34.44.19.185	TCP	54	49836 → 80 [ACK] Seq=1 Ack=2 Win=64240 Len=0

```

> Frame 4: 481 bytes on wire (3848 bits), 481 bytes captured (3848 bits) on interface 0
> Ethernet II, Src: PcsCompu_88:83:f5 (08:00:27:88:83:f5), Dst: RealtekU_12:35:02 (52:54:00:12:35:02)
> Internet Protocol Version 4, Src: 10.0.2.15, Dst: 34.44.19.185
> Transmission Control Protocol, Src Port: 49836, Dst Port: 80, Seq: 1, Ack: 1, Len: 427

0000  52 54 00 12 35 02 08 00 27 88 83 f5 08 00 45 00 RT-·5... ····E-
0010  01 d3 0c f7 40 00 80 06 00 00 0a 00 02 0f 22 2c .....@... ·····
0020  13 b9 c2 ac 00 50 e9 34 48 91 4c 6e 68 02 50 18 .....P-4 H-Lnh-P
0030  fa f0 43 b9 00 00 35 4d 67 54 22 35 65 69 39 3a ..C-·5N g7"5ei9:
0040  35 65 6c 39 73 73 22 6c 35 55 39 73 3a 68 0a 65 Sel9ss"1 S09s:h·e
0050  65 35 6c 47 65 65 65 65 53 38 75 38 75 4f 75 53 e51Geee S8u8u0uS
0060  3e 68 4c 43 4a 4c 46 65 65 65 65 65 65 43 >HLCJLj. eeeeeeeC
0070  4f 75 53 4f 34 75 53 53 34 75 4f 4f 3e 68 4c 4c OuSO4uSS 4u00>LL
0080  34 65 65 65 65 65 6c 2d 39 3a 22 5f 46 4d 55 35 4eeeeee1- 9;" FMU5
0090  0d 0e 65 65 35 6c 47 65 65 65 53 38 75 38 75 ..ee51Ge eee58u8u
00a0  4f 75 53 3e 68 4c 43 48 4e 48 65 65 65 65 65 Ou$jhLCJ Leeeeeee
00b0  65 65 53 43 4f 75 4f 4f 43 75 4f 4f 53 75 43 eeSC0u00 CU00SuC9
00c0  68 48 38 65 65 65 65 65 6c 2d 39 3a 22 5f 46 hh8eeeeee e1-9;"_F
00d0  4d 55 35 0d 0a 65 65 35 6c 47 65 65 65 65 53 38 MU5-·ee5 1GeeeeS8
00e0  75 38 75 4f 75 53 3e 68 4c 43 4a 4c 43 65 65 u8u0uS>h LCJLCeee
00f0  65 65 65 65 65 53 43 4f 75 4f 4f 43 75 4f 4f 53 eeeee5CO uOCuOOS
0100  75 43 3e 68 48 38 65 65 65 65 65 65 65 65 65 65 u>H8ee eeee1-9;
0110  22 5f 46 4d 55 35 0d 0a 65 65 35 6c 47 65 65 65 65 " FMU5-· ee51Geed

```

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

Hex View-1    A    Structures

```
sub    esp, 30h
mov    [ebp+ProcName], 77h ; 'w'
mov    [ebp+var_1F], 55h ; 'U'
mov    [ebp+var_1E], 44h ; 'D'
mov    [ebp+var_1D], 64h ; 'd'
mov    [ebp+var_1C], 59h ; 'Y'
mov    [ebp+var_1B], 53h ; 'S'
mov    [ebp+var_1A], 58h ; '['
mov    [ebp+var_19], 73h ; 's'
mov    [ebp+var_18], 5Fh ; '_'
mov    [ebp+var_17], 45h ; 'E'
mov    [ebp+var_16], 5Eh ; '^'
mov    [ebp+var_15], 44h ; 'D'
mov    [ebp+var_14], 1Eh
mov    [ebp+ModuleName], 58h ; '['
mov    [ebp+var_2F], 55h ; 'U'
mov    [ebp+var_2E], 42h ; 'B'
mov    [ebp+var_2D], 5Eh ; '^'
mov    [ebp+var_2C], 55h ; 'U'
mov    [ebp+var_2B], 5Ch ; '\\'
mov    [ebp+var_2A], 3
mov    [ebp+var_29], 2
mov    [ebp+var_28], 1Eh
mov    [ebp+var_27], 54h ; 'T'
mov    [ebp+var_26], 5Ch ; '\\'
mov    [ebp+var_25], 5Ch ; '\\'
mov    [ebp+var_24], 1Eh
mov    [ebp+var_4], 0
jmp    short loc_401140
```

In function **sub\_4011C0**, the function StrStrIW and the library Shlwapi.dll is loaded at runtime, to avoid it being visible among the import.

```
cmp    [ebp+hSnapshot], 0FFFFFFFh
jnz    short loc_401203

loc_401203:
push   offset ProcName ; "StrStrIW"
push   offset LibFileName ; "Shlwapi.dll"
call   ds:LoadLibraryA
push   eax           ; hModule
call   ds:GetProcAddress
mov    [ebp+StrStrIW], eax
mov    [ebp+var_20], offset aDaimao ; "daimao"
mov    [ebp+var_1C], offset aPiccolo ; "piccolo"
mov    [ebp+var_18], offset aZamasu ; "zamasu"
mov    [ebp+var_14], offset aDercori : "dercori"
```

Something very similar happens, for the function sub\_401900

```
push   0           ; dwFlags
push   0           ; hToken
push   7           ; csidl
push   0           ; hwnd
call   ds:SHGetFolderPathW
mov    [ebp+var_C], eax
push   offset aStrStrIW_0 ; "StrStrIW"
push   offset aShlwapiDll_0 ; "Shlwapi.dll"
call   ds:LoadLibraryA
push   eax           ; hModule
call   ds:GetProcAddress
mov    [ebp+StrStrIW], eax
push   offset Destination
lea    ecx, [ebp+Filename]
push   ecx
call   [ebp+StrStrIW]
test  eax, eax
jz    short loc_401969
```

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:

```

4011    sub_4011/00
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.

The screenshot shows the IDA Pro interface with two windows open. The top window is the 'HEX VIEW' showing assembly code. The bottom window is the 'DATA' window showing memory dump details. A red arrow points from the assembly code to the memory dump window, indicating a specific memory location being analyzed.

Assembly code (top window):

```
loc_401476:
    cmp    [ebp+var_C], 3
    jnb    short loc_401476

push 20h ; ' '
mov   ecx, [ebp+var_C]
shl   ecx, 5
lea   edx, [ebp+ecx+ModuleName]
push  edx ; pbBinary
mov   eax, [ebp+var_C]
mov   ecx, Str[eax*4]
push  ecx ; Str
call  decrypt
add   esp, 0Ch
jmp   short loc_401442
```

Memory dump (bottom window):

```
loc_401476:
    mov   [ebp+var_4], offset off_453000 ; "=42bpR3Y1N1ZPdXZpZFch1mbVRnT"
    mov   [ebp+variable], offset dword_45B058
    push  40h ; '@'
    lea   edx, [ebp+ProcName]
    push  edx ; pbBinary
    mov   eax, [ebp+var_4]
    mov   ecx, [eax]
    push  ecx ; Str
    call  decrypt
    add   esp, 0Ch
    lea   edx, [ebp+ProcName]
    push  edx ; lpProcName
    mov   eax, 20h ; ''
    shl   eax, 0
    lea   ecx, [ebp+eax+ModuleName]
    push  ecx ; lpModuleHandleA
    call  ds:GetModuleHandleA
    push  eax ; hModule
    call  ds:GetProcAddress
    mov   edx, [ebp+variable]
    mov   [edx], eax
    mov   eax, [ebp+var_4]
    add   eax, 4
```

Symbol table (bottom window):

```
.data:00453000 , segment permissions, read/write
.data:00453000 _data
.data:00453000
.data:00453000
.data:00453000 ;org 453000h
dd offset a42bpR3Y1N1ZPdXZpZFch1mbVRnT
; DATA XREF: sub_401430:loc_401476↑o
; "42bpR3Y1N1ZPdXZpZFch1mbVRnT"
dd offset aAd4vgdu92qkfwz ; ==Ad4VGdu92QkFWZyhGV0V2U"
dd offset aKfwzyhgvl1wdzv ; "kFWZyhGVl1WdzVmU"
dd offset aAefrwyljhauvgd ; ==AeFRWYlJHaUVGdhVmDRnT"
dd offset aAzhvmcorfzuvgc ; ==AZhVmcoRFZuVGczV3U"
dd offset aGxrj9gbsfebhv ; "gXRj9GbsFEbhVHdylmV"
dd offset aAd4vgdu92qkfwz_0 ; ==Ad4VGdu92QkFWZyhGV0V2R"
dd offset a5j3btwtznxzj9 ; "5J3btWTzNXZj9mcQVGdpJ3V"
dd offset aBlncjhjnypxezh9 ; "BlncJnYpxEZh9GT"
.data:00453024 ; char *Str
```