Malware Analysis and Incident Forensics (Ms Cybersecurity)
Systems and Enterprise Security (Ms Eng. in CS)
Practical test - 18/04/2024

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-20240418*.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:

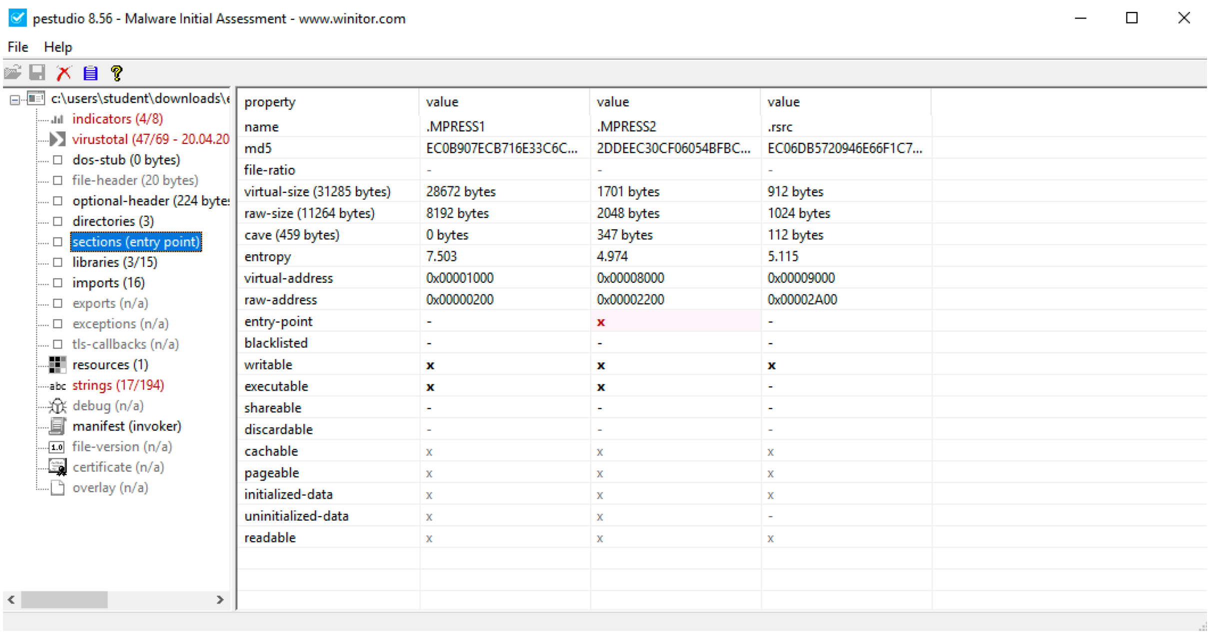
- Indicators of packing:
    - Sections names .MPRESS1, .MPRESS2
    - The entry point is not in the first section
    - Both sections have write and execute permissions
    - Few imports per library, but there is the presence of GetProcAddress and GetModuleHandle which are used to load and gain access 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 much 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
    - CyptStringToBinry, which is probably used for obfuscation purposes
    - SHGetKnownFolderPath, probably used to retrieve the full path of a known folder
- 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 (crypt32.dll, kernel32.dll, shell32.dll, netapi32.dll)
- Extentions (.dll)

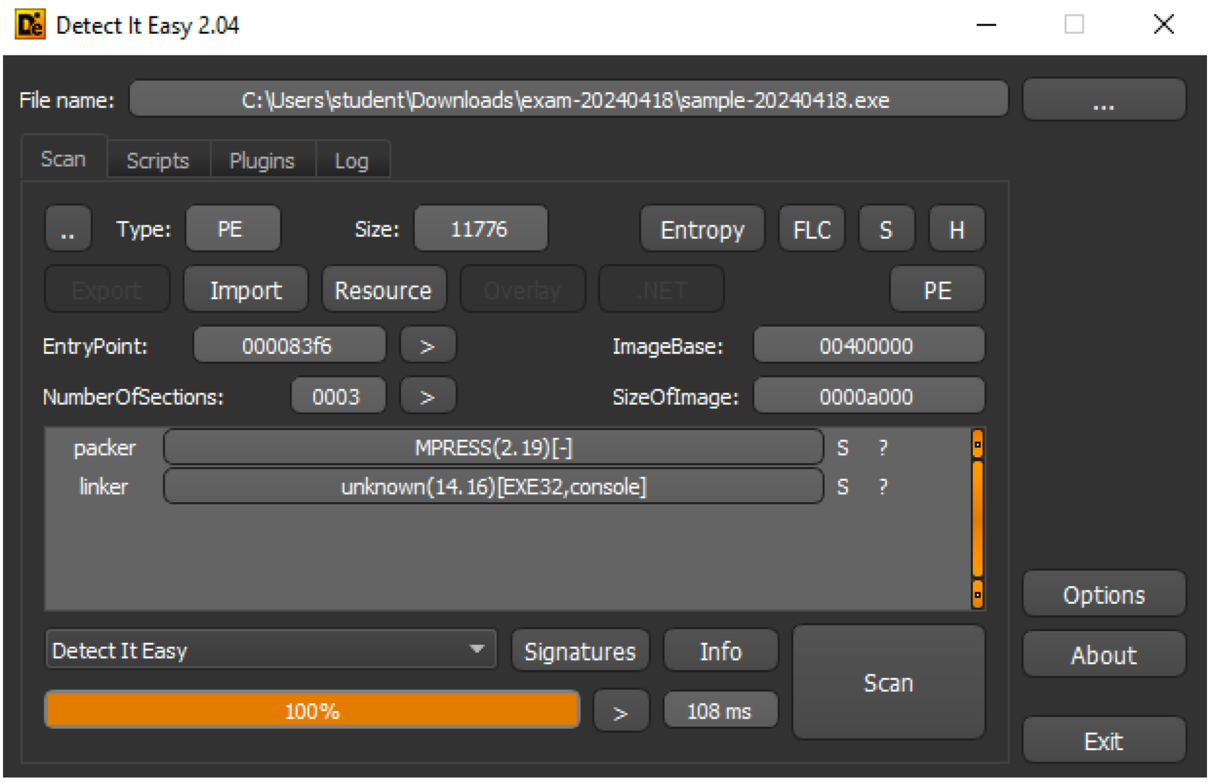
- The *library* section contains some libraries that are potential indicators or could reveal the sample's behavior. They are the following:
  - shell32.dll, which likely means that the sample interacts with other processes
  - crypt32.dll, which likely means that the malware performs some kind of encryption
  - user32.dll, which likely means that the sample performs user-level interactions such as showing a message box
  - advapi32.dll
  - ws2_32.dll which likely means that the sample performs some interaction on the network
- The *resources* section contains N resources used by the sample. Those are:
  - The “Manifest”



property	value	value	value
name	.MPRESS1	.MPRESS2	.rsrc
md5	EC0B907ECB716E33C6C...	2DDEEC30CF06054BFBC...	EC06DB5720946E66F1C7...
file-ratio	-	-	-
virtual-size (31285 bytes)	28672 bytes	1701 bytes	912 bytes
raw-size (11264 bytes)	8192 bytes	2048 bytes	1024 bytes
cave (459 bytes)	0 bytes	347 bytes	112 bytes
entropy	7.503	4.974	5.115
virtual-address	0x00001000	0x00008000	0x00009000
raw-address	0x00000200	0x00002200	0x00002A00
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). Furthermore, after clicking “Entropy”, it confirms the packing.

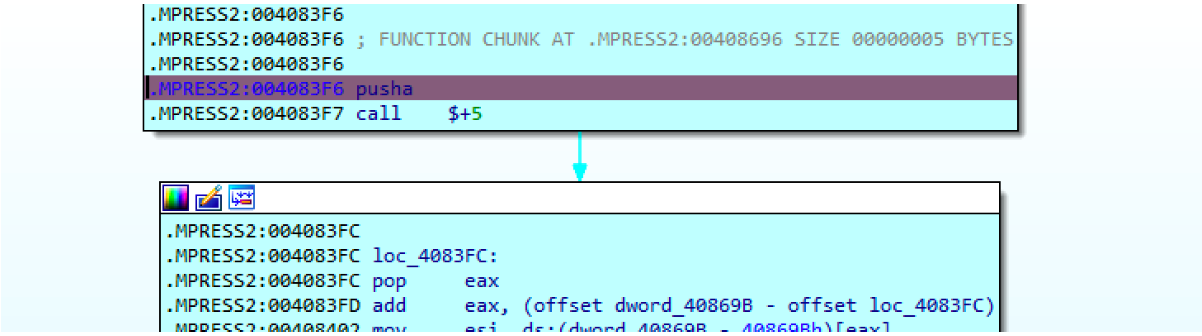


To find the OEP of a packed sample it's necessary to locate the tail jump, that is the jump that the packed sample performs to the beginning of the unpacked code after the unpacking stub has finished its operations.

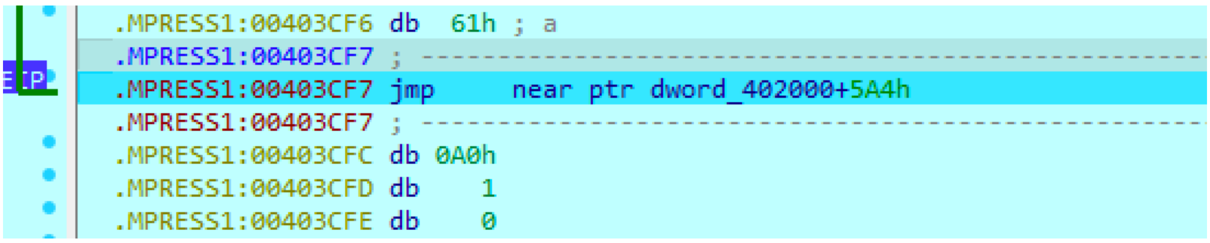
There are some indicators useful to recognize the tail jump that will allow us to fine the OEP:

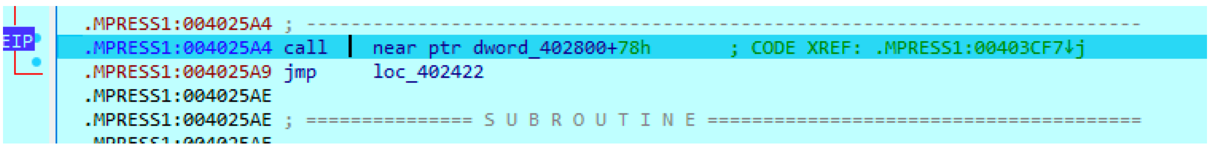
- The instruction jumps to another section (in this case from UPX1 to UPX0)
- After the tail jump should be a bunch of garbage bytes.
- The destination was previously modified by the unpacking stub

After opening the sample in IDA and starting at the entry point in .MPRESS2 (0x4083F6), 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 @ 0x403CF7
OEP @ 0x4025A4



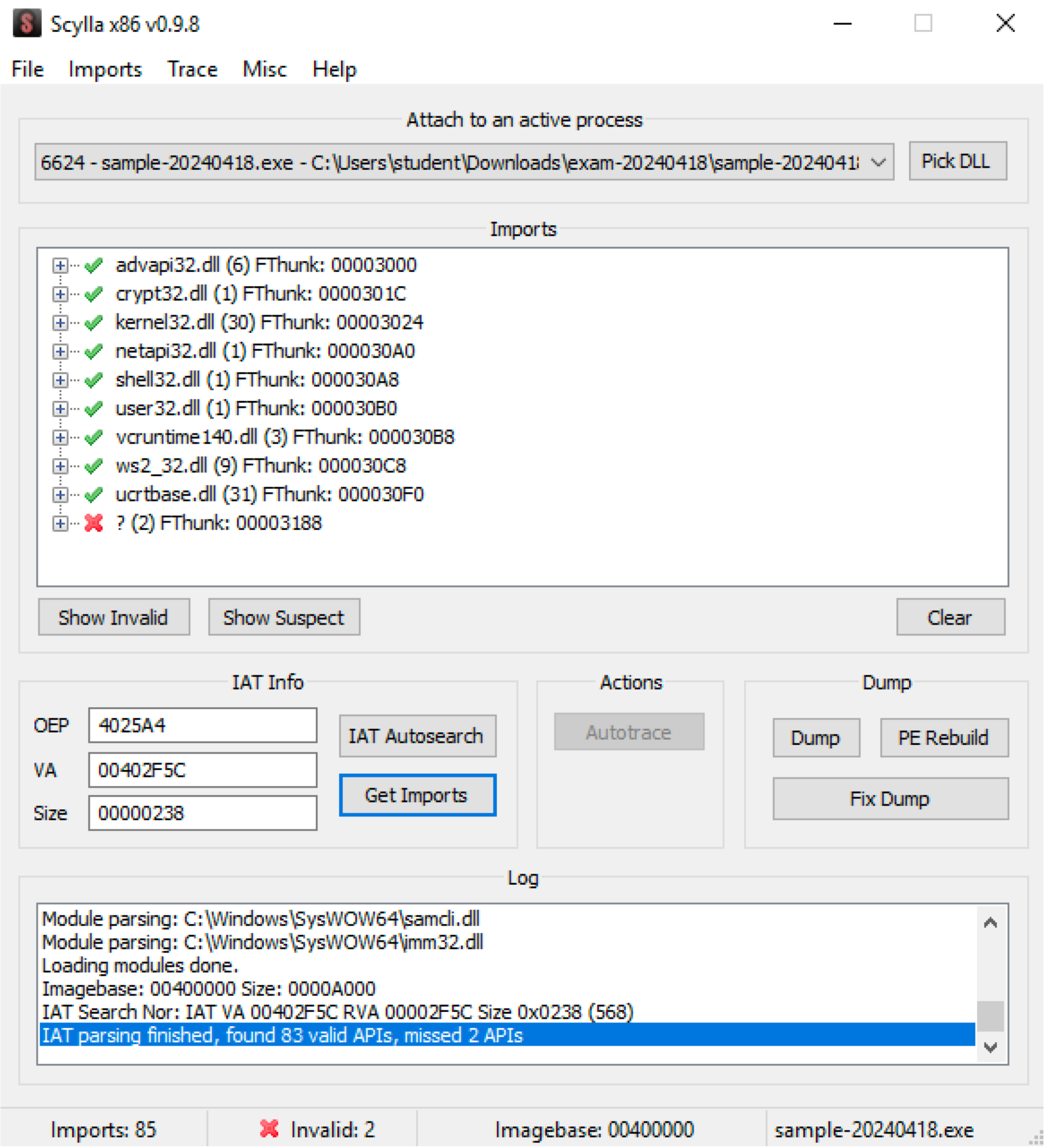


3 - Provide details about the IAT reconstruction process that you carried out to unpack the code. <u>*HINTS:*</u> *the answer should cover methodological aspects and facts on your output; also, validate it! (e.g., check API calls, compare with sample-20240710-unpacked.exe).*

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

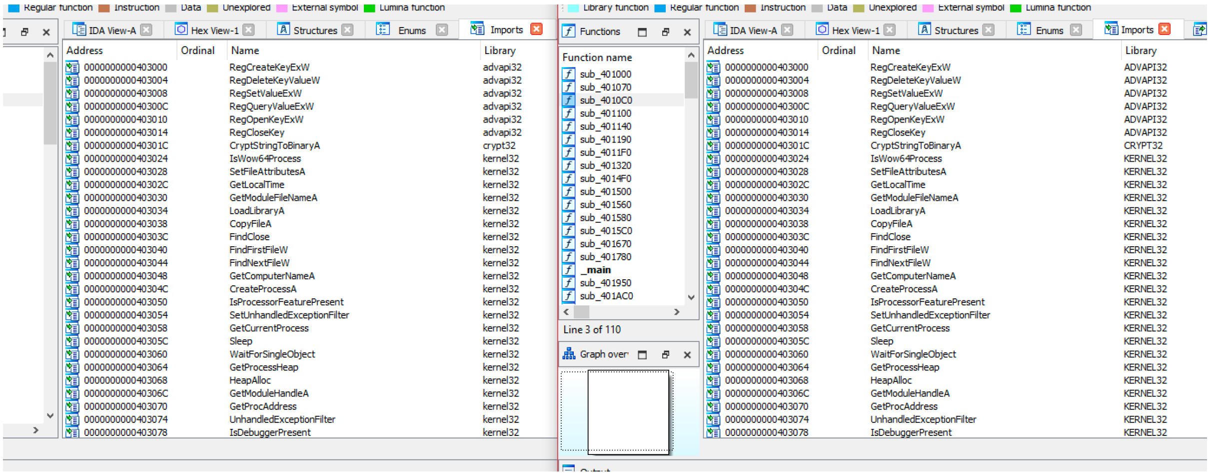
- Pressing **IAT Autosearch** we can obtain the IAT information starting from the OEP (0x4025A4). At this point Scylla retrieves its virtual address and the size;
- Then, with **Get import** we can retrieve the list of imports. There is an invalid entry, as we can see in the screenshot, that can be deleted.

- 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-

20240116-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 already is a registry key. If it is true, it exits otherwise, it creates the registry key Software\Microsoft\GDIPlus with the value UseThatMightToConquerJiren.
2. The sample checks the layout of the keyboard (Italian, UK, US, walles)
3. The sample checks the time of the machine and then copies itself in the desktop folder and hide the file
4. The sample performs the shellcode injects in DevicePairingWizard.exe
5. Finally the sample performs the network connections to the server 35.223.190.146 on port 80 and waits the comand

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**
Executable	timeaftertime.exe	Copy of the malware in desktop folder	IDA
Process	DevicePairingWizard	Victim process in which the sample injects	IDA, process hacker

		the shellcode	
Registry key	Software\Microsoft\GDIPlus\UseThatMightToConquerJiren	Registry key that is used for the persistence purpose	IDA
Network connection	35.223.190.146:80	Connection performs by the sample	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). **<u>NOTE</u>**: listing such parts is optional only in the case of shellcodes. <u>*HINTS:*</u> *Main code starts at* ***0xXXXXXXX****. Code at 0xXXXXXXX and higher addresses can be safely ignored.*

The **main** starts at 0x401900:

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    offset aCryptstringtob ; "CryptStringToBinaryA"
push    offset aCrypt32Dll_0 ; "crypt32.dll"
call    ds:GetModuleHandleA
push    eax             ; hModule
call    ds:GetProcAddress
mov     dword_406014, eax
mov     dword_406010, offset sub_402280
call    sub_402180
call    sub_4015C0
call    sub_401BF0
call    sub_401780
call    sub_401320
call    dword_406010
xor     eax, eax
pop     ebp
retn
_main endp
```

At the beginning the sample take the address of CryptStringToBinaryA using GetModuleHandleA and GetProcAddress.

sub_402280

- It deletes the value “UseThatMightToConquerJiren” from the registry key “Software\Microsoft\GDIPlus”, using RegDeleteKeyValueW and then exit

sub_402180 (check registry key)

- It calls RegOpenKeyExW to open the registry key “Software\Microsoft\GDIPlus” and checks if there is the subkey value in this key. If already exist, it calls a MessageBoxA and exit. If there not exit, it calls RegOpenKeyExW with subkey “Software\Microsoft” then it creates the key GDIPlus using RegCreateKeyExW and then it creates the value “UseThatMightToConquerJiren” using RegSetValueExW. Finally close the key.

sub_4015C0 (check keyboard)

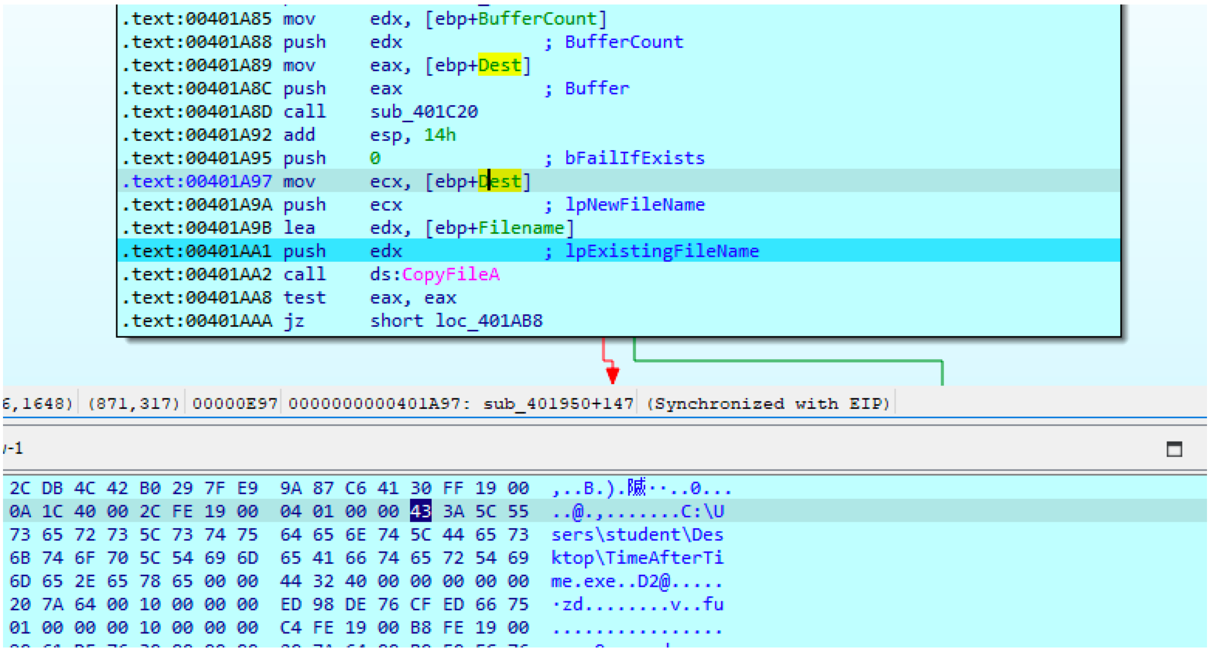
- It calls the function sub_4022B0 that used to decrypt some strings. The strings that are decrypt are:
  - “7573657233322e646c6c”: user32.dll
  - “4765744b6579626f6172644c61796f75744e616d6541”: GetKeyboardLayoutNameA

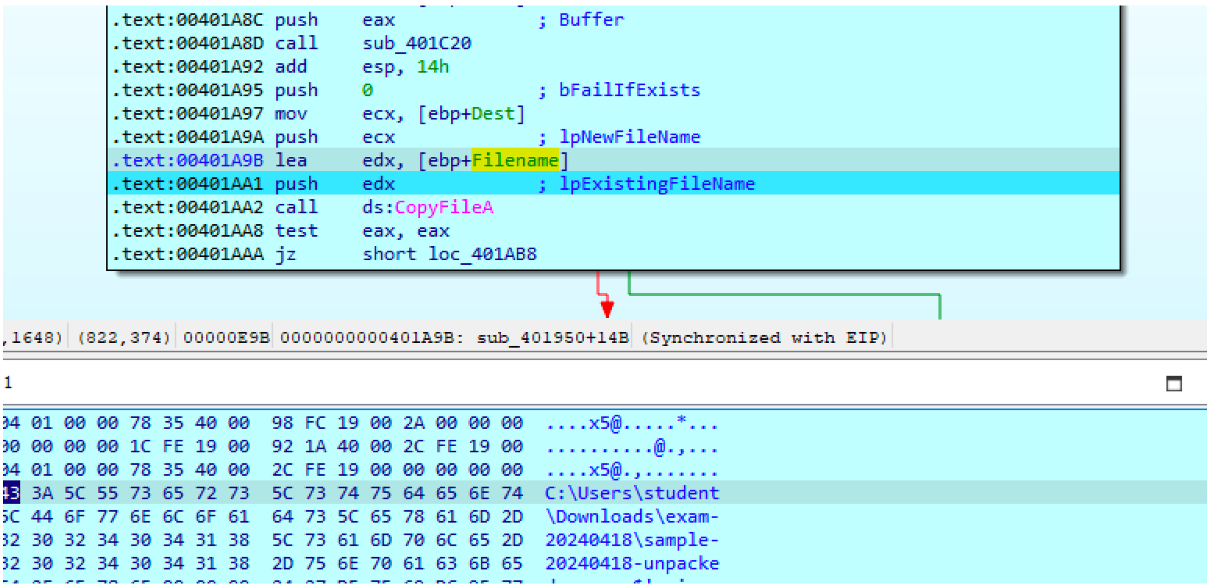
  Then it uses GetProcAddress to retrieve the address of the function GetKeyboardLayoutNameA. It calls this function to check the language of the keyboard based on some hex value:

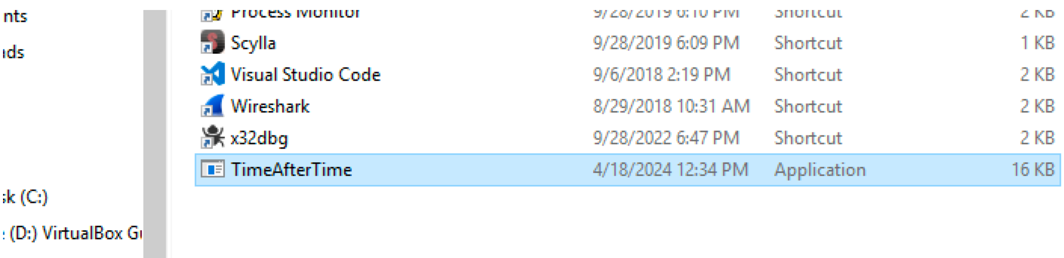
  - 0x410: Italian keyboard
  - 0x809: UK keyboard
  - 0x452: walles (UK) keyboard
  - 0x409: US keyboard

sub_401BF0 (check time, copy itself and persistence)

- The sample calls sub_401950. In this function the sample uses GetLocalTime to check if the day of the week is not 6 and lower or equal than 3. It decrypts the string shell32.dll and SHGetKnownFolderPath using the function sub_4022B0. It copies itself in the path: C:\Users\student\desktop and create the file timeaftertime.exe in that folder using CopyFileA. This executable timeaftertime.exe is located in 0x403564.





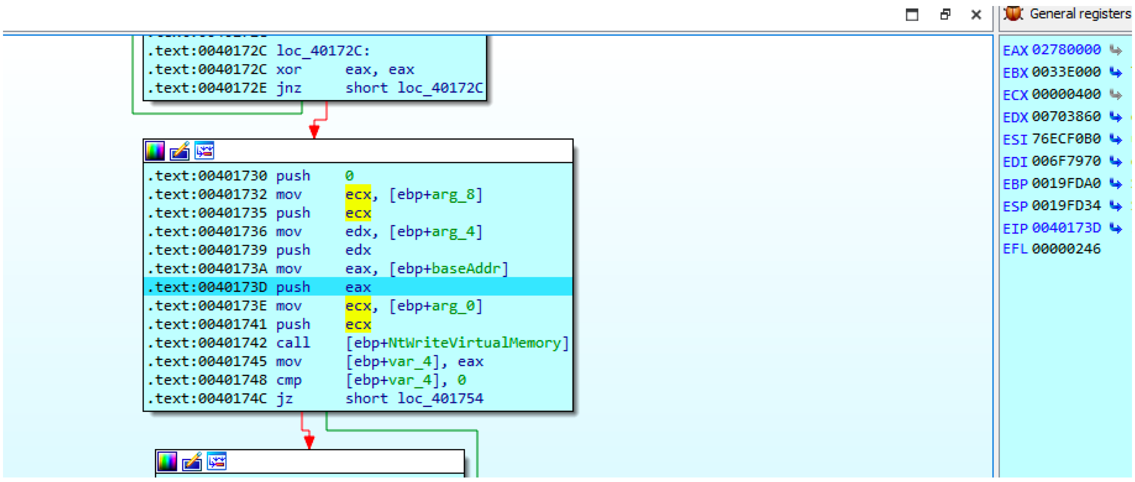


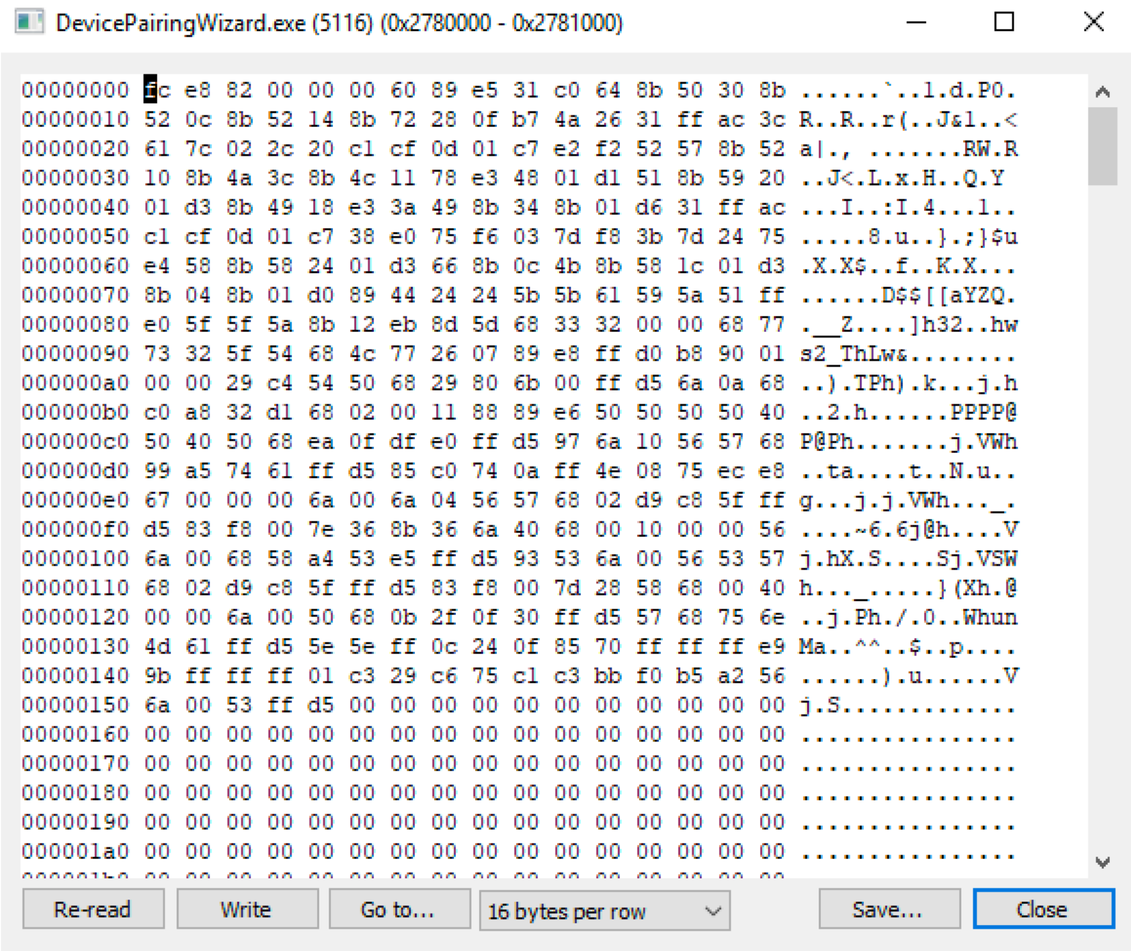
Then it calls SetFileAttributesA with dwFileAttributes = 0x2 to hide that file.

- It calls the function sub_401AC0
  - In that function it calls RegCreateKeyExA and RegSetValueExA to create a key Software\Microsoft\Windows\CurrentVersion\RunOnce named “MANGA Plus by SHUEISHA”. It this way the sample can survive after the reboot.

sub_401780 (shellcode injection)

- It calls GetProcessHeap and HeapAlloc to take 341 bytes on the heap of the process.
- It calls GetCurrentProcess to retrieve the process DevicePairingWizard.exe and it calls IsWow64Process to see if the environment is 32 bit or 64 bit.
- It calls CreateProcessA to create the process DevicePairingWizard.exe and it goes to sleep for 5sec (1388h) using the function Sleep.
- Then it performs a check to see if the process is already executed in the system. If it is true, it prints a message using MessageBoxA
- If it is false, it calls the function sub_401670
  - In that function, the sample performs the shellcode injection into DevicePairingWizard.exe.
  - It decrypts ntdll.dll and its functions:
    - NtAllocateVirtualMemory
    - NtWriteVirtualMemory
  - It calls NtAllocateVirtualMemory to allocate memory in the victim process and then it calls NtWriteVirtualMemory to write the shellcode in that memory (address = 0x2780000, this address change dynamically at each execution). The number of bytes written are 400 bytes





sub_401320 (C&C)

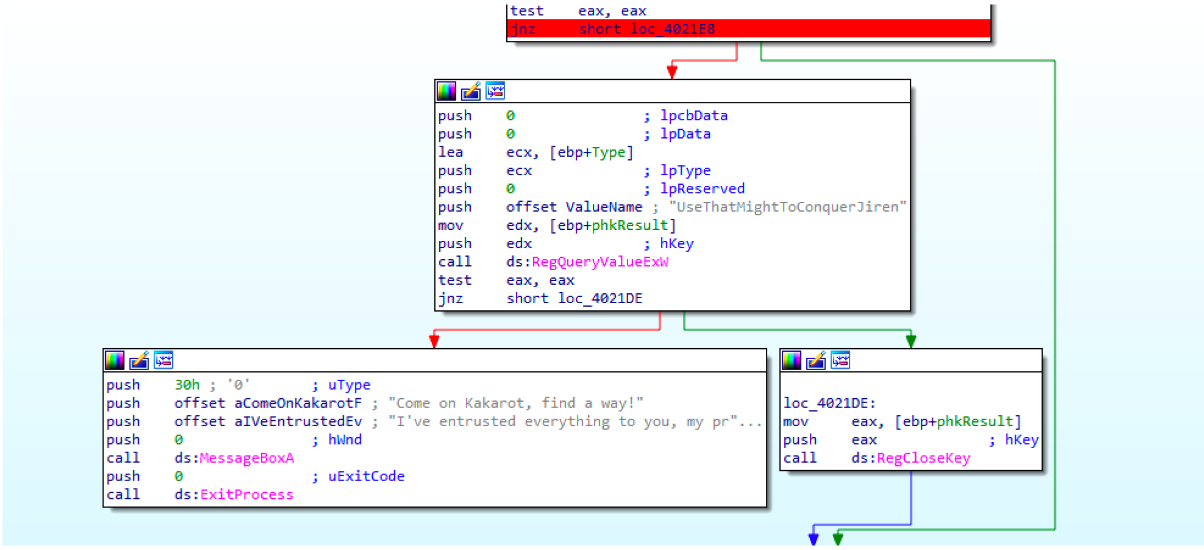
- The sample calls some network functions using WSAStartup and it tries to connect to server 35.223.190.146 on port 80.
- If it is successful it send RDY\n to the server and then waits command from the server:
    - If L is received, it sends BYE\n and close the connection
    - If Q is received, it enumerates the user account %s--%s
    - If H is received, it sends the computer name in this form: CN: %s
    - If G is received, gets the directory and files in student folder (using FindFirstFileW, FindNextFile)
    - If I is received, sends “ZZZ 20000” and goes to sleep
    - If S is received, gets the current time of the machine T: %02d:%02d\n
- The last five steps are in sub_4011F0

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.

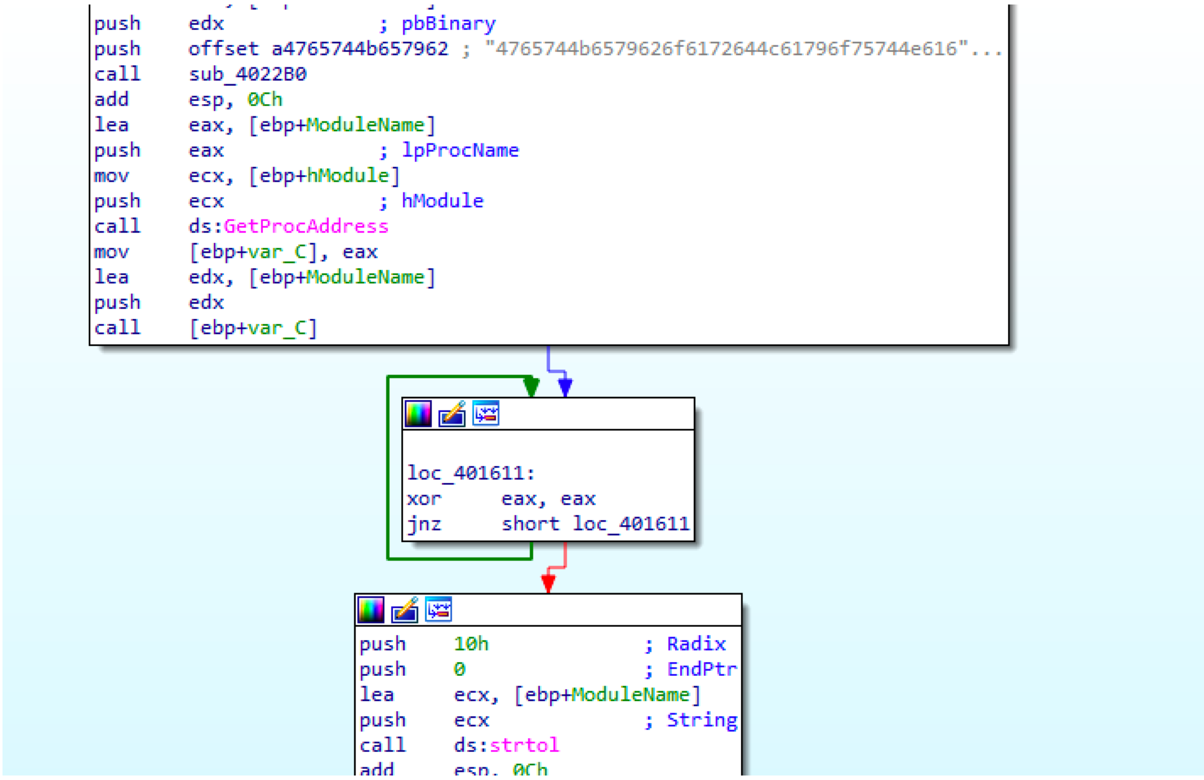
As show before in question 6, the sample, before performing its malicious activities, check if there is the registry Software\Microsoft\GDIPlus.

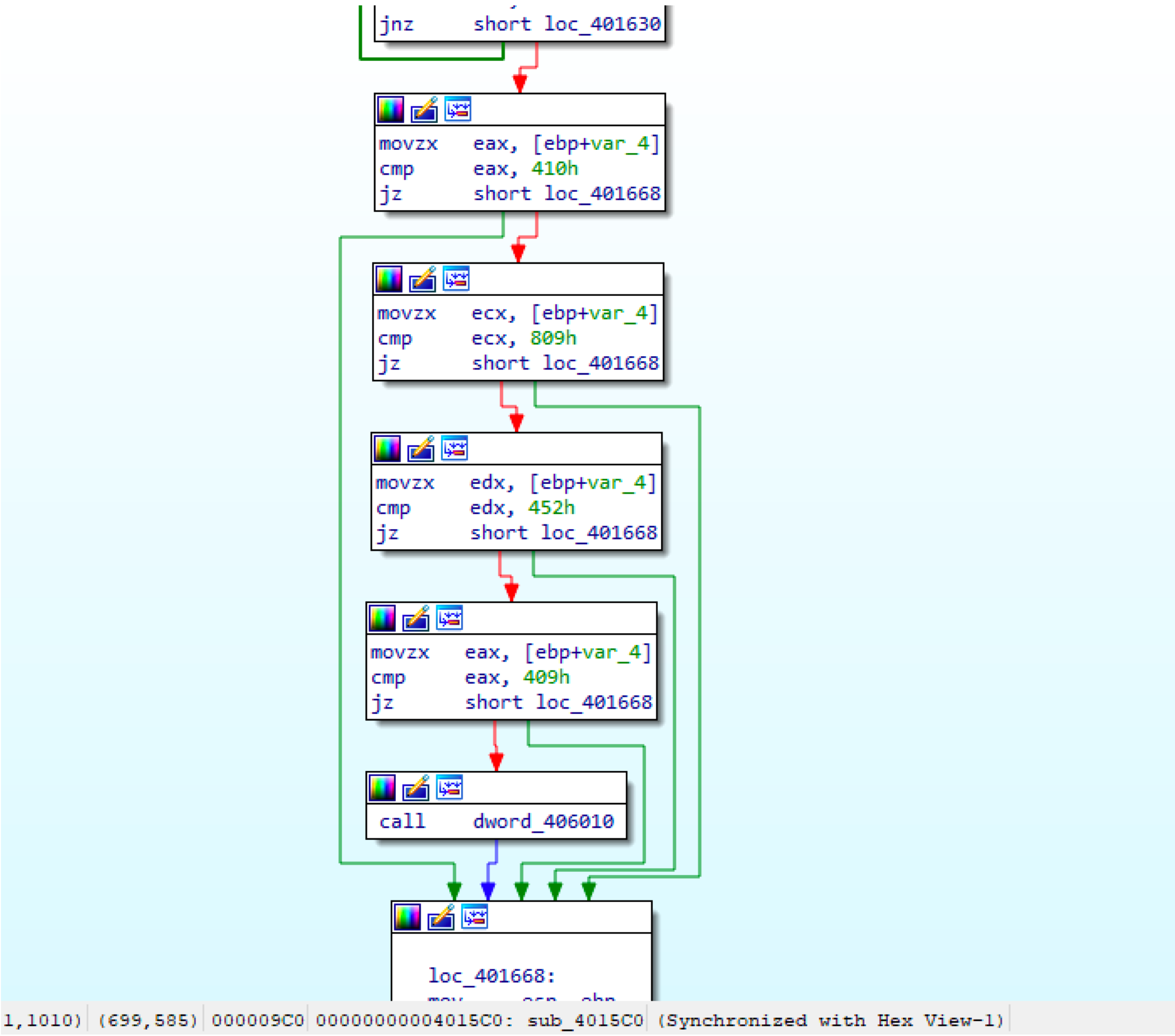
if another copy of the malware is already running then the registry key

Software\Microsoft\GDIPlus\UseThatMightToConquerJiren will be set. To ensure that only one instance of the malware is running every execution check for the presence of this key, if it finds it it terminates. The key is cleaned up thanks to function sub_402280, called on exit by the malware.

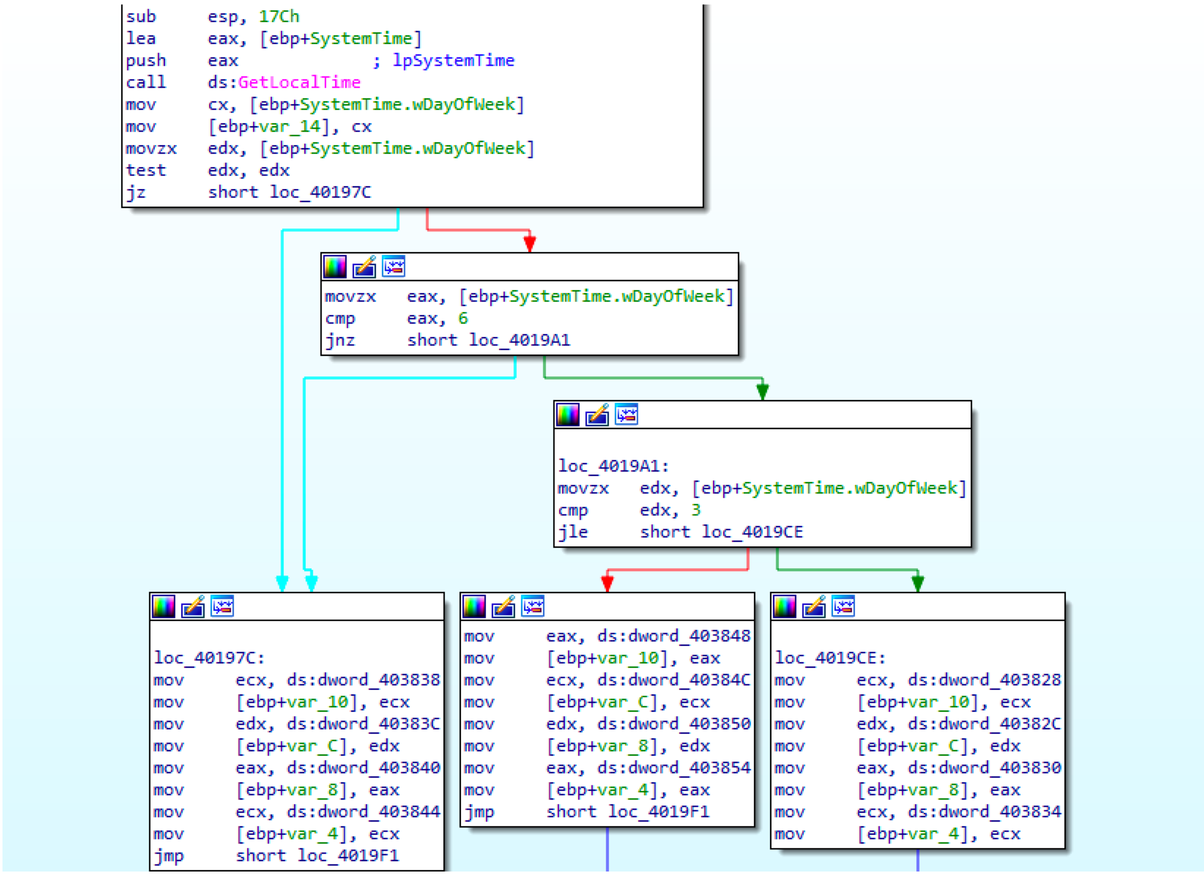


It checks the keyboard layout in the function sub_4015C0





It also checks the current data of the machine in function sub_401950



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_401AC0, the malware creates a copy of itself and create a registry key. In this way, the malware will be automatically run at the startup of the machine.

```
push    ecx             ; lpProcName
mov     edx, [ebp+hModule]
push    edx             ; hModule
call    ds:GetProcAddress
mov     [ebp+RegCreateKeyExA], eax
push    0
lea     eax, [ebp+var_C]
push    eax
push    0
push    0F003Fh
push    0
push    0
push    0
push    offset aSoftwareMicros ; "Software\\Microsoft\\Windows\\CurrentVe"...
push    80000001h
call    [ebp+RegCreateKeyExA]
lea     ecx, [ebp+var_30]
push    ecx             ; lpProcName
mov     edx, [ebp+hModule]
push    edx             ; hModule
call    ds:GetProcAddress
mov     [ebp+RegSetValueExA], eax
mov     eax, [ebp+Str]
push    eax             ; Str
call    strlen
add     esp, 4
push    eax
mov     ecx, [ebp+Str]
push    ecx
push    1
push    0
push    offset aMangaPlusByShu ; "MANGA Plus by SHUEISHA"
mov     edx, [ebp+var_C]
push    edx
call    [ebp+RegSetValueExA]
mov     esp, ebp
pop     ebp
retn
```

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 at unk_403278 and injected in DevicePairingWizard.exe

Then, the sample performs the following operations in function sub_401670:

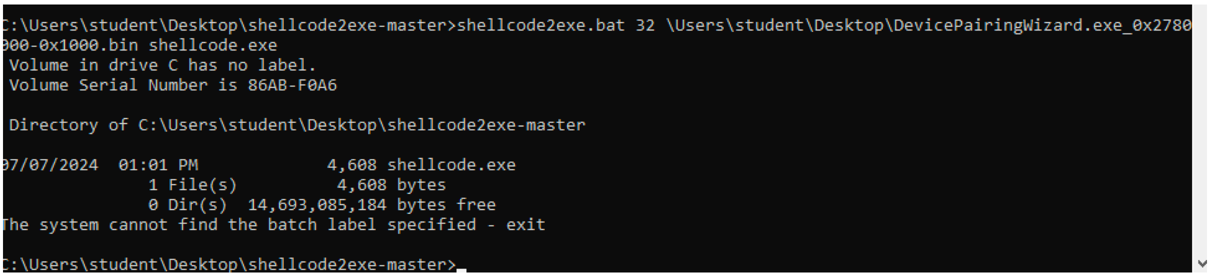
- creates a new process “DevicePairingWizard.exe”
- makes space in memory for the payload using: NtAllocateVirtualMemory
- copies the payload stored at location unk_403278 using NtWriteVirtualMemoery with Size 400 bytes inside the process memory 0x2780000 (in my case, it changes every time dynamically)
- deciphers an obfuscated string that will happen to be “RtlCreateUserThread” and then initiates the payload invoking it

How extract the payload:

1. In Process Hacker, inspect the DevicePairingWizard.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. Converti il payload in eseguibile usando 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.

```
foo:00401061 pop     eax
foo:00401062 mov     ebx, [eax+24h]
foo:00401065 add     ebx, edx
foo:00401067 mov     cx, [ebx+ecx*2]
foo:0040106B mov     ebx, [eax+1Ch]
foo:0040106E add     ebx, edx
foo:00401070 mov     eax, [ebx+ecx*4]
foo:00401073 add     eax, edx
foo:00401075 mov     [esp+28h+var_4], eax
foo:00401079 pop     ebx
foo:0040107A pop     ebx
foo:0040107B popa
foo:0040107C pop     ecx
foo:0040107D pop     edx
foo:0040107E push    ecx
foo:0040107F jmp     eax
```

We can see that the shellcode calls WSAStartup, WSASocketA, connect, bind, accept.

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 35.223.190.146, it sends a beacon “RDY” and waits (it calls recv) for a command:

- If L is received, it sends BYE\n and close the connection
- If Q is received, it enumerates the user account %s--%s
- If H is received, it sends the computer name in this form: CN: %s
- If G is received, gets the directory and files in student folder (using FindFirstFileW, FindNextFile)
- If I is received, sends “ZZZ 20000” and goes to sleep
- If S is received, gets the current time of the machine T: %02d:%02d\n

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 function sub_4015C0, the strings used to retrieve the dll and the function imported is incrypted before calling GetProcAddress and GetModuleHandle and not to be visible at a plain static analysis.

```
var_4= word ptr -4

push    ebp
mov     ebp, esp
sub     esp, 4Ch
push    40h ; '@'        ; pcbBinary
lea     eax, [ebp+ModuleName]
push    eax              ; pbBinary
push    offset Str       ; "7573657233322e646c6c"
call    sub_4022B0
add     esp, 0Ch
lea     ecx, [ebp+ModuleName]
push    ecx              ; lpModuleName
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
push    40h ; '@'        ; pcbBinary
lea     edx, [ebp+ModuleName]
push    edx              ; pbBinary
push    offset a4765744b657962 ; "4765744b6579626f6172644c61796f75744e616"...
call    sub_4022B0
add     esp, 0Ch
lea     eax, [ebp+ModuleName]
push    eax              ; lpProcName
mov     ecx, [ebp+hModule]
push    ecx              ; hModule
call    ds:GetProcAddress
mov     [ebp+var_C], eax
lea     edx, [ebp+ModuleName]
push    edx
call    [ebp+var_C]
```

Something very similar happens, for the function sub_401950

```
loc_4019F1:              ; pcbBinary
push    40h ; '@'
lea     edx, [ebp+ModuleName]
push    edx              ; pbBinary
push    offset a7368656c6c3332 ; "7368656c6c33322e646c6c"
call    sub_4022B0
add     esp, 0Ch
lea     eax, [ebp+ModuleName]
push    eax              ; lpModuleName
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
push    40h ; '@'        ; pcbBinary
lea     ecx, [ebp+ModuleName]
push    ecx              ; pbBinary
push    offset a53484765744b6e ; "53484765744b6e6f776e466f6c6465725061746"...
call    sub_4022B0
add     esp, 0Ch
lea     edx, [ebp+ModuleName]
push    edx              ; lpProcName
mov     eax, [ebp+hModule]
push    eax              ; hModule
call    ds:GetProcAddress
mov     [ebp+SHGetKnownFolderPath], eax
lea     ecx, [ebp+Source]
push    ecx
push    0
push    0
lea     edx, [ebp+var_10]
push    edx
call    [ebp+SHGetKnownFolderPath]
mov     [ebp+var_38], eax
push    104h             ; MaxCount
mov     eax, [ebp+Source]
push    eax              ; Source
mov     ecx, [ebp+Dest]
push    ecx              ; Dest
call    ds:wcstombs
```

In function used for the persistence, the name of the dll and functions are pushed on stack byte-per-byte not to be visible at a plain static analysis:

```
sub     esp, 40h
mov     [ebp+LibFileName], 61h ; 'a'
mov     [ebp+var_1F], 64h ; 'd'
mov     [ebp+var_1E], 76h ; 'v'
mov     [ebp+var_1D], 61h ; 'a'
mov     [ebp+var_1C], 70h ; 'p'
mov     [ebp+var_1B], 69h ; 'i'
mov     [ebp+var_1A], 33h ; '3'
mov     [ebp+var_19], 32h ; '2'
mov     [ebp+var_18], 2Eh ; '.'
mov     [ebp+var_17], 64h ; 'd'
mov     [ebp+var_16], 6Ch ; 'l'
mov     [ebp+var_15], 6Ch ; 'l'
mov     [ebp+var_14], 0
mov     [ebp+ProcName], 52h ; 'R'
mov     [ebp+var_3F], 65h ; 'e'
mov     [ebp+var_3E], 67h ; 'g'
mov     [ebp+var_3D], 43h ; 'C'
mov     [ebp+var_3C], 72h ; 'r'
mov     [ebp+var_3B], 65h ; 'e'
mov     [ebp+var_3A], 61h ; 'a'
mov     [ebp+var_39], 74h ; 't'
mov     [ebp+var_38], 65h ; 'e'
mov     [ebp+var_37], 4Bh ; 'K'
mov     [ebp+var_36], 65h ; 'e'
mov     [ebp+var_35], 79h ; 'y'
mov     [ebp+var_34], 45h ; 'E'
mov     [ebp+var_33], 78h ; 'x'
mov     [ebp+var_32], 41h ; 'A'
mov     [ebp+var_31], 0
mov     [ebp+var_30], 52h ; 'R'
mov     [ebp+var_2F], 65h ; 'e'
mov     [ebp+var_2E], 67h ; 'g'
mov     [ebp+var_2D], 53h ; 'S'
mov     [ebp+var_2C], 65h ; 'e'
mov     [ebp+var_2B], 74h ; 't'
mov     [ebp+var_2A], 56h ; 'V'
mov     [ebp+var_29], 61h ; 'a'
mov     [ebp+var_28], 6Ch ; 'l'
mov     [ebp+var_27], 75h ; 'u'
```

In the sub_401670, the name of dll and imported functions are encrypted before calling GetProcAddress and GetModuleHandle and not to be visible at a plain static analysis

```
mov     ebp, esp
sub     esp, 60h
push    40h ; '@'        ; pcbBinary
lea     eax, [ebp+ModuleName]
push    eax              ; pbBinary
push    offset a6e74646c6c2e64 ; "6e74646c6c2e646c6c"
call    sub_4022B0
add     esp, 0Ch
lea     ecx, [ebp+ModuleName]
push    ecx              ; lpModuleName
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
push    40h ; '@'        ; pcbBinary
lea     edx, [ebp+ModuleName]
push    edx              ; pbBinary
push    offset a4e74416c6c6f63 ; "4e74416c6c6f636174655669727475616c4d656"...
call    sub_4022B0
add     esp, 0Ch
lea     eax, [ebp+ModuleName]
push    eax              ; lpProcName
mov     ecx, [ebp+hModule]
push    ecx              ; hModule
call    ds:GetProcAddress
mov     [ebp+NtAllocateVirtualMemory], eax
push    40h ; '@'        ; pcbBinary
lea     edx, [ebp+ModuleName]
push    edx              ; pbBinary
push    offset a4e745772697465 ; "4e7457726974655669727475616c4d656d6f727"...
call    sub_4022B0
add     esp, 0Ch
lea     eax, [ebp+ModuleName]
push    eax              ; lpProcName
mov     ecx, [ebp+hModule]
push    ecx              ; hModule
call    ds:GetProcAddress
mov     [ebp+NtWriteVirtualMemory], eax
push    offset aRtlcreateusert ; "RtlCreateUserThread"
push    offset aNtdllDll ; "ntdll.dll"
call    ds:GetModuleHandleA
```