LIFARS
your digital world, **secured**

A Detailed Analysis of Lazarus' RAT Called FALLCHILL

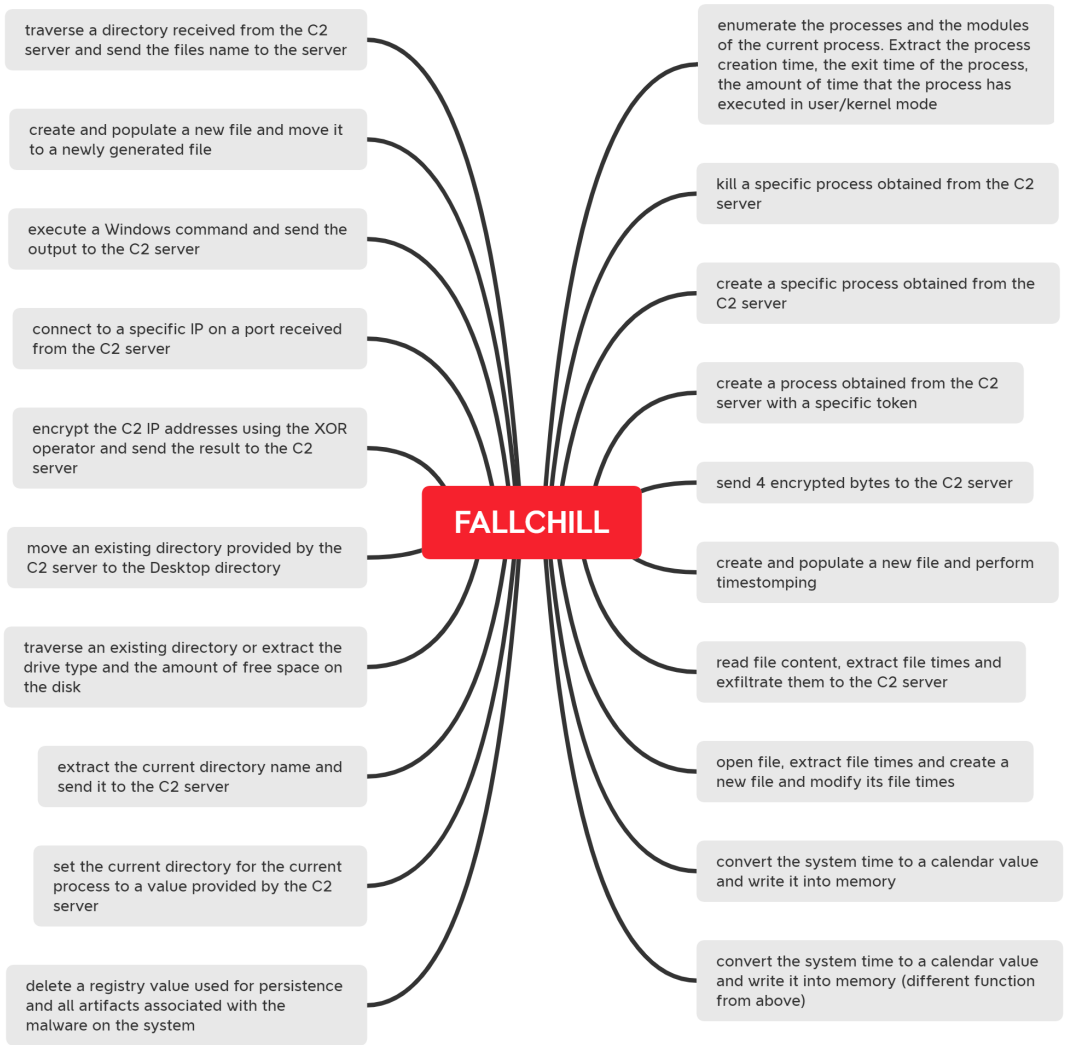
\

Prepared by: LIFARS, LLC
Date: 09/07/2021

EXECUTIVE SUMMARY

FALLCHILL is a RAT that has been used by Lazarus Group since 2016. The malware decrypts multiple strings at runtime using the XOR algorithm and the RC4 hard-coded key "0D 06 09 2A 86 48 86 F7 0D 01 01 01 05 00 03 82". It implements a custom algorithm that is used to decode multiple DLL names and export functions, which will be imported at runtime. The process collects the following data from the machine and generates a victim ID: OS version information, MAC address, host name, host IP address. The following IP addresses represent the C2 servers, which will instruct the malware on what command to perform: 175.100.189.174 and 125.212.132.222. The diagram presented below presents all the functionalities implemented by this RAT.

FALLCHILL DIAGRAM



ANALYSIS AND FINDINGS

SHA256:a606716355035d4a1ea0b15f3bee30aad41a2c32df28c2d468eafd18361d60d6

The malware writes multiple RC4 and XOR encrypted strings to the memory. One such example is shown in figure 1:

```
.text:0040501D mov     al, 71h ; 'q'
.text:0040501F mov     [esp+48h+var_44], 0Bh
.text:00405024 mov     [esp+48h+var_F], al
.text:00405028 mov     [esp+48h+var_C], al
.text:0040502C mov     al, 31h ; '1'
.text:0040502E mov     [esp+48h+var_43], 0FBh ; 'û'
.text:00405033 test    esi, esi
.text:00405035 mov     [esp+48h+var_42], 3Eh ; '>'
.text:0040503A mov     [esp+48h+var_41], 0B7h ; '·'
.text:0040503F mov     [esp+48h+var_40], 0C8h ; 'È'
.text:00405044 mov     [esp+48h+var_3F], 0BFh ; '¿'
.text:00405049 mov     [esp+48h+var_3E], 94h ; '”'
.text:0040504E mov     [esp+48h+var_3D], 0E2h ; 'â'
.text:00405053 mov     [esp+48h+var_3B], 6
.text:00405058 mov     [esp+48h+var_3A], 0DAh ; 'Ú'
.text:0040505D mov     [esp+48h+var_39], 1Eh
.text:00405062 mov     [esp+48h+var_38], 88h ; 'ˆ'
.text:00405067 mov     [esp+48h+var_37], 14h
.text:0040506C mov     [esp+48h+var_36], 0Fh
.text:00405071 mov     [esp+48h+var_35], 74h ; 't'
.text:00405076 mov     [esp+48h+var_34], 83h ; 'ƒ'
.text:0040507B mov     [esp+48h+var_33], 8Eh ; 'Ž'
.text:00405080 mov     [esp+48h+var_32], 52h ; 'R'
.text:00405085 mov     [esp+48h+var_31], 0A7h ; '§'
.text:0040508A mov     [esp+48h+var_30], 0D4h ; 'Ô'
.text:0040508F mov     [esp+48h+var_2F], 19h
.text:00405094 mov     [esp+48h+var_2E], 8Bh ; '‹'
.text:00405099 mov     [esp+48h+var_2D], 96h ; '–'
.text:0040509E mov     [esp+48h+var_2B], 6Dh ; 'm'
.text:004050A3 mov     [esp+48h+var_2A], 0F3h ; 'ó'
.text:004050A8 mov     [esp+48h+var_29], 97h ; '—'
.text:004050AD mov     [esp+48h+var_28], 2Eh ; '.'
.text:004050B2 mov     [esp+48h+var_27], 23h ; '#'
.text:004050B7 mov     [esp+48h+var_26], 0A2h ; '¢'
.text:004050BC mov     [esp+48h+var_25], dl
.text:004050C0 mov     [esp+48h+var_24], 92h ; '’'
.text:004050C5 mov     [esp+48h+var_23], 0B8h ; '¸'
.text:004050CA mov     [esp+48h+var_22], 7Ch ; '|'
.text:004050CF mov     [esp+48h+var_21], 8Dh
.text:004050D4 mov     [esp+48h+var_1F], 0C1h ; 'Á'
.text:004050D9 mov     [esp+48h+var_1D], 0FFh
.text:004050DE mov     [esp+48h+var_1C], 99h ; '™'
.text:004050E3 mov     [esp+48h+var_1B], 21h ; '!'
.text:004050E8 mov     [esp+48h+var_1A], 6Eh ; 'n'
```

Figure 1

The hard-coded RC4 key "0D 06 09 2A 86 48 86 F7 0D 01 01 01 05 00 03 82" is used to decrypt multiple strings at runtime:

```
.text:00402EB0 mov     cl, 0Dh
.text:00402EB2 mov     al, 86h ; '†'
.text:00402EB4 mov     [esp+1Ch+var_10], cl
.text:00402EB8 mov     [esp+1Ch+var_C], al
.text:00402EBC mov     [esp+1Ch+var_A], al
.text:00402EC0 mov     [esp+1Ch+var_8], cl
.text:00402EC4 mov     ecx, 1
.text:00402EC9 lea     eax, [ebp-1]
.text:00402ECC cmp     eax, ecx
.text:00402ECE mov     [esp+1Ch+var_F], 6
.text:00402ED3 mov     [esp+1Ch+var_E], 9
.text:00402ED8 mov     [esp+1Ch+var_D], 2Ah ; '*'
.text:00402EDD mov     [esp+1Ch+var_B], 48h ; 'H'
.text:00402EE2 mov     [esp+1Ch+var_9], 0F7h ; '÷'
.text:00402EE7 mov     [esp+1Ch+var_7], cl
.text:00402EEB mov     [esp+1Ch+var_6], cl
.text:00402EEF mov     [esp+1Ch+var_5], cl
.text:00402EF3 mov     [esp+1Ch+var_4], 5
.text:00402EF8 mov     [esp+1Ch+var_3], 0
.text:00402EFD mov     [esp+1Ch+var_2], 3
.text:00402F02 mov     [esp+1Ch+var_1], 82h ; ','
```

Figure 2

There is a custom implementation of the RC4 algorithm provided by the sample, as shown below:

```
.text:0040302E
.text:0040302E loc_40302E:
.text:0040302E mov     eax, [esi+4]
.text:00403031 xor     ecx, ecx
.text:00403033 mov     dl, [eax+100h]
.text:00403039 inc     dl
.text:0040303B mov     [eax+100h], dl
.text:00403041 mov     eax, [esi+4]
.text:00403044 mov     cl, [eax+100h]
.text:0040304A mov     dl, [ecx+eax]
.text:0040304D mov     cl, [eax+101h]
.text:00403053 add     cl, dl
.text:00403055 xor     edx, edx
.text:00403057 mov     [eax+101h], cl
.text:0040305D mov     eax, [esi+4]
.text:00403060 xor     ecx, ecx
.text:00403062 mov     cl, [eax+101h]
.text:00403068 mov     dl, [eax+100h]
.text:0040306E add     ecx, eax
.text:00403070 add     edx, eax
.text:00403072 push    ecx
.text:00403073 push    edx
.text:00403074 mov     ecx, esi
.text:00403076 call    sub_402E80
.text:0040307B mov     eax, [esi+4]
.text:0040307E xor     ecx, ecx
.text:00403080 xor     edx, edx
.text:00403082 mov     cl, [eax+101h]
.text:00403088 mov     dl, [eax+100h]
.text:0040308E mov     cl, [ecx+eax]
.text:00403091 add     cl, [edx+eax]
.text:00403094 and     ecx, 0FFh
.text:0040309A mov     dl, [ecx+eax]
.text:0040309D mov     al, [ebx+edi]
.text:004030A0 xor     dl, al
.text:004030A2 mov     [edi], dl
.text:004030A4 inc     edi
.text:004030A5 dec     ebp
.text:004030A6 jnz     short loc_40302E
```

Figure 3

An example of a string decrypted using a XOR operation, and the RC4 algorithm is displayed in figure 4:

Address	Hex	ASCII
0019FEB4	35 00 6D 00 6B 00 66 00 4A 00 59 00 37 00 6B 00	5.m.k.f.J.Y.7.k.
0019FEC4	6A 00 6D 00 48 00 63 00 6A 00 34 00 6A 00 6C 00	j.m.H.c.j.4.j.l.
0019FED4	78 00 47 00 36 00 6A 00 68 00 73 00 64 00 52 00	x.G.6.j.h.s.d.R.
0019FEE4	54 00 37 00 46 00 61 00 77 00 37 00 66 00 6A 00	T.7.F.a.w.7.f.j.

Figure 4

The binary uses the SetErrorMode function in order to force the system not to display the critical-error-handler message box and the Windows Error Reporting dialog (0x3 = **SEM_FAILCRITICALERRORS** | **SEM_NOGPFAULTERRORBOX**):

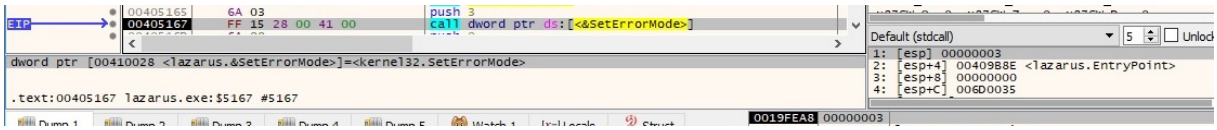


Figure 5

A new thread is created by the malware using the CreateThread API:

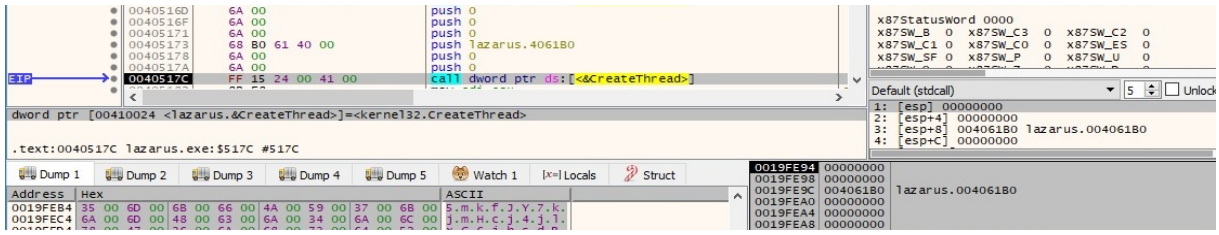


Figure 6

THREAD ACTIVITY – START ADDRESS FUNCTION

We can use CyberChef (https://gchq.github.io/CyberChef/) to confirm that the algorithm used to decrypt strings is indeed RC4:

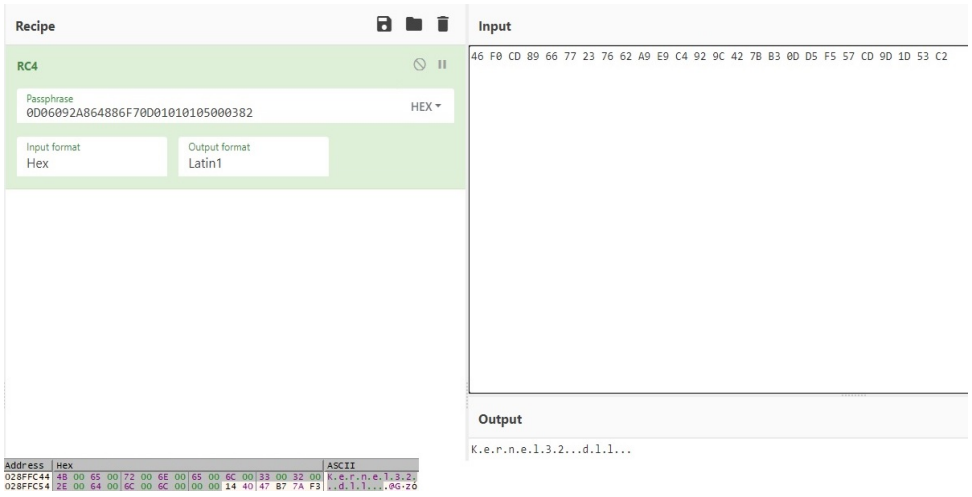


Figure 7

The LoadLibraryW routine is utilized to load multiple DLLs into the address space:

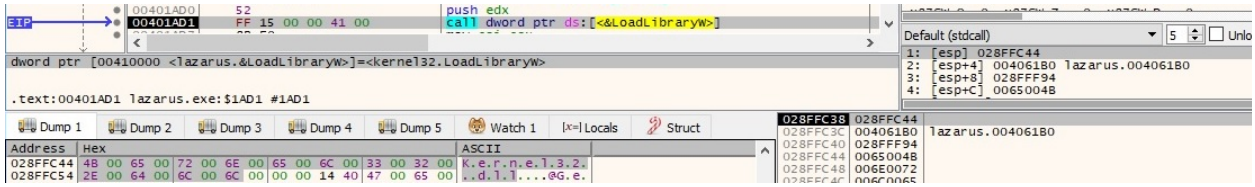


Figure 8

The executable retrieves the address of multiple exported functions by calling the GetProcAddress function:

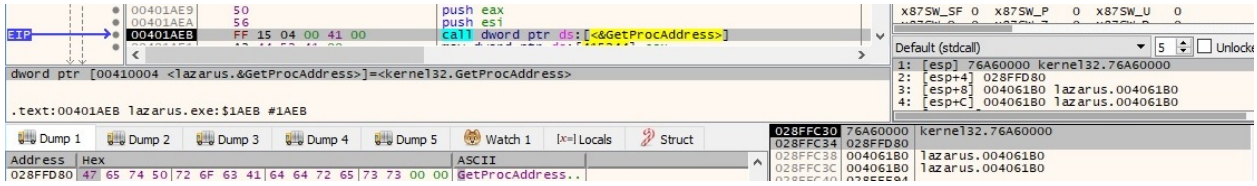


Figure 9

A simple encoding algorithm that consists of subtracting a hex number from 0xDB is implemented by the file (the decryption algorithm is implemented in Python and presented in the appendix):

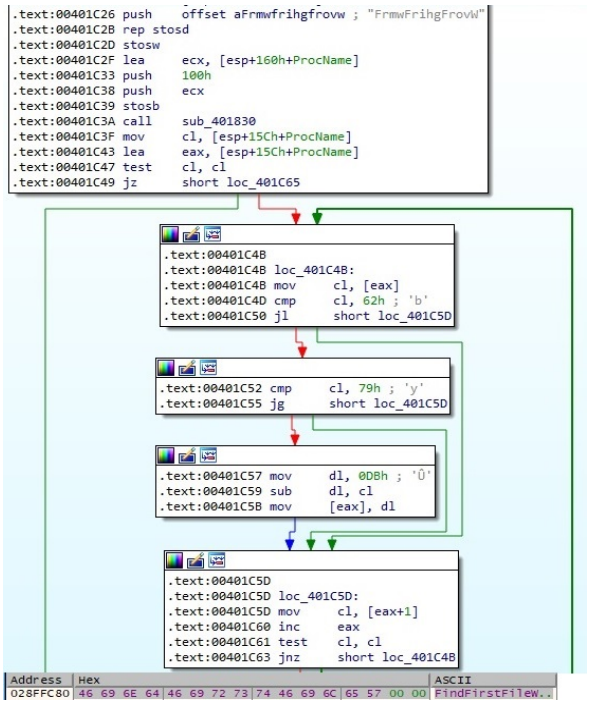


Figure 10

The following DLLs are also loaded by the malicious process: wtsapi32.dll, Advapi32.dll, ws2_32.dll and iphlpapi.dll. The process decrypts the following function names and gets the address of them via a GetProcAddress function call:

- Module32FirstW, WinExec, FindFirstFileW, LocalAlloc, CreateThread, ReadFile, GetFileSize, GetExitCodeProcess, CloseHandle, GetTempFileNameW, Process32FirstW, DeleteFileW, LoadLibraryW, GetExitCodeThread, GetFileTime, TerminateThread, LocalFree, WaitForSingleObject, WaitForMultipleObjects, GetModuleFileNameW, WriteFile, Process32NextW, Sleep, MapViewOfFile, ReadProcessMemory, SetFilePointer, CreateToolhelp32Snapshot, GetTempPathW, CreateProcessW, GetFileAttributesW, GetLocalTime, GetSystemDirectoryW, GetVolumeInformationW, GetCurrentProcess, UnmapViewOfFile, GetVersionExW, SetFileTime, GetLogicalDrives, GetCurrentDirectoryW, SetCurrentDirectoryW, OpenProcess, CreateFileW, TerminateProcess, FreeLibrary, VirtualProtectEx, WriteProcessMemory, GetComputerNameW, FindNextFileW, GetModuleHandleW, MoveFileExW, FindClose, CreateFileMappingW, VirtualQueryEx, GetDriveTypeW, GetDiskFreeSpaceExW, GetLastError, SetLastError, VirtualAllocEx, CreateRemoteThread, FindResourceW, LoadResource, LockResource, GetTickCount
- WTSQueryUserToken, WTSEnumerateSessionsW
- OpenProcessToken, RegOpenKeyW, ControlService, SetServiceStatus, CloseServiceHandle, AdjustTokenPrivileges, LookupPrivilegeValueW, GetTokenInformation, LookupAccountSidW, OpenServiceW, RegDeleteKeyW, DeleteService, RegDeleteValueW, ChangeServiceConfig2W, OpenSCManagerW, CreateServiceW, StartServiceW, RegSetValueExW, RegCloseKey, RegisterServiceCtrlHandlerW, RegCreateKeyW, RegOpenKeyExW, RegQueryValueExW, GetUserNameW, CreateProcessAsUserW

- WSACleanup, recv, setsockopt, WSAStartup, listen, shutdown, gethostbyname, getpeername, accept, ioctlsocket, connect, closesocket, socket, htons, select, send, __WSAFDIsSet, bind, inet_addr
- GetAdaptersInfo

The malicious executable initiates the usage of Winsock DLL using the WSAStartup routine:

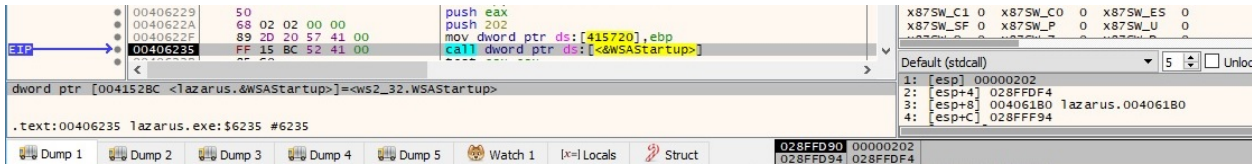


Figure 11

The file tries to open a registry key that doesn't exist on our machine.

According to an article published by US-CERT at

https://us-cert.cisa.gov/sites/default/files/publications/MAR-10135536-A_WHITE_S508C.pdf, the data stored in this key is RC4 encrypted, and XOR encoded (0x80000002 = **HKEY_LOCAL_MACHINE** and 0x20019 = **KEY_READ**):

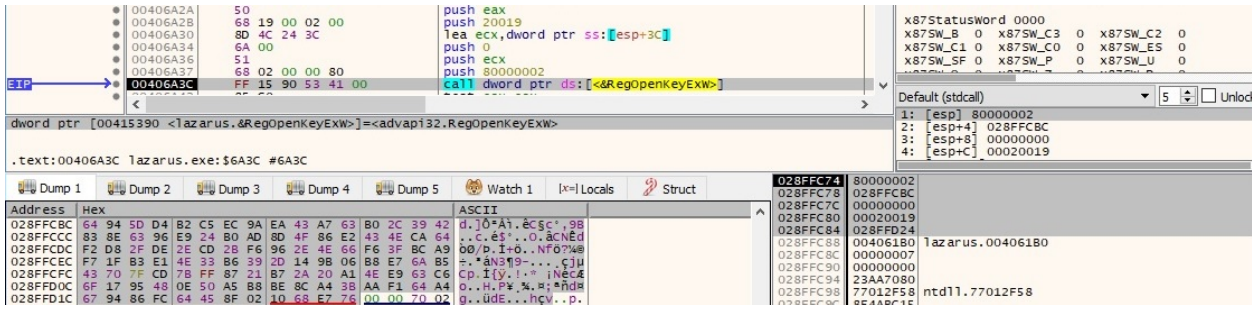


Figure 12

The major/minor version and the build number of the operating system are extracted via a GetVersionExW API call, as highlighted below:

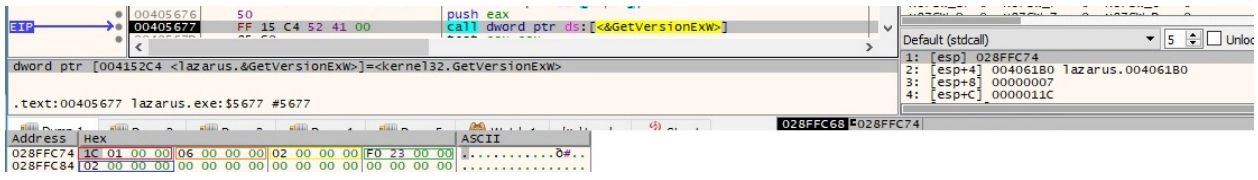


Figure 13

The GetAdaptersInfo routine is used to retrieve adapter information for the local machine. The binary extracts the hardware address (MAC) from the result and stores it in a separate buffer:

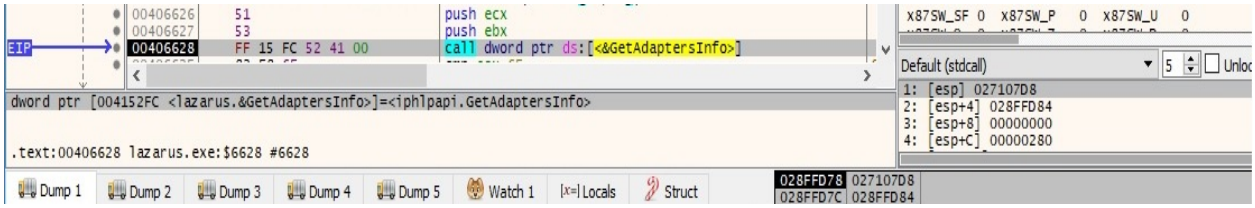


Figure 14

The NetBIOS name of the computer is extracted using GetComputerNameW:

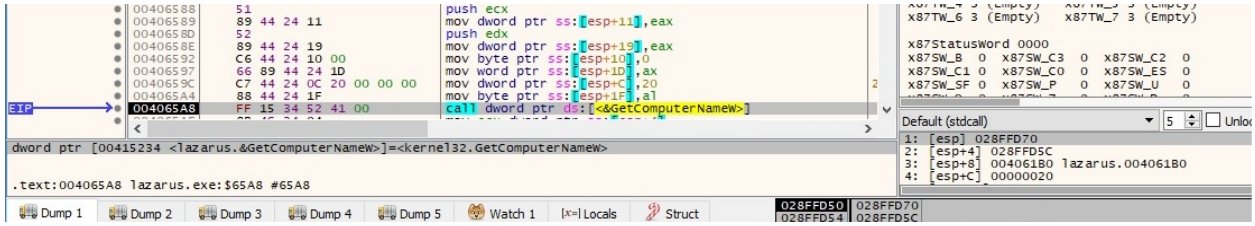


Figure 15

The private IP address of the host along with other information is extracted using the gethostbyname function:

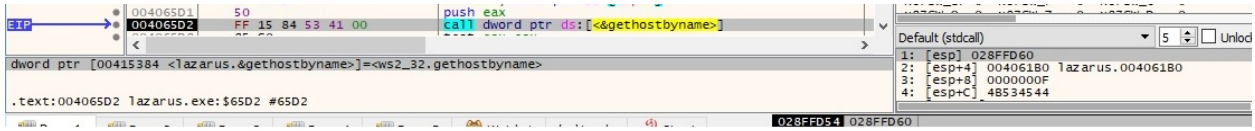


Figure 16

The following buffer contains the IP address extracted earlier, the host name, and different information about the operating system extracted above:

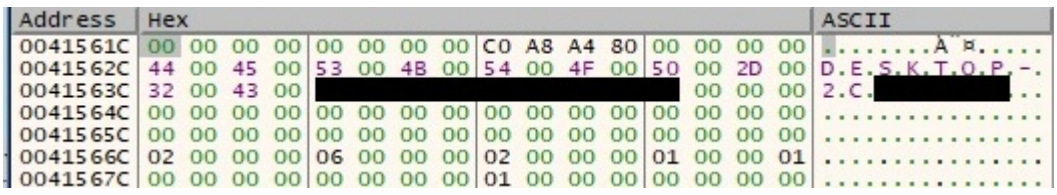


Figure 17

The executable generates a unique ID based on a GetTickCount function call and the MAC address. The algorithm utilized to obtain the ID is custom and consists of a lot of operations (a snippet of it is displayed below):

```
Address   Hex                                              ASCII
028FFCEC  00 0C 29 25 66 15 2D 00 20 F1 88 A4 03 00 00 00  ..)%f.-. ñ.¤....
.text:004037BE xor     ebx, ebp
.text:004037C0 mov     [esp+0B4h+var_40], edi
.text:004037C4 and     ebx, edx
.text:004037C6 mov     edx, [ecx+0Ch]
.text:004037C9 mov     edi, esi
.text:004037CB xor     ebx, edx
.text:004037CD mov     edx, [esp+0B4h+var_40]
.text:004037D1 rol     edi, 5
.text:004037D4 add     ebx, edi
.text:004037D6 add     ebx, edx
.text:004037D8 mov     edx, [ecx+10h]
.text:004037DB lea     edi, [edx+ebx+5A827999h]
.text:004037E2 mov     ebx, [eax+4]
.text:004037E5 mov     edx, [ecx+4]
.text:004037E8 mov     [esp+0B4h+var_3C], ebx
.text:004037EC rol     edx, 1Eh
.text:004037EF mov     ebx, ebp
.text:004037F1 mov     [esp+0B4h+var_A0], edi
.text:004037F5 xor     ebx, edx
.text:004037F7 and     ebx, esi
.text:004037F9 rol     edi, 5
.text:004037FC xor     ebx, ebp
.text:004037FE add     ebx, edi
.text:00403800 mov     edi, [esp+0B4h+var_3C]
.text:00403804 add     ebx, edi
.text:00403806 mov     edi, [ecx+0Ch]
.text:00403809 rol     esi, 1Eh
.text:0040380C lea     edi, [edi+ebx+5A827999h]
.text:00403813 mov     ebx, [eax+8]
.text:00403816 mov     [esp+0B4h+var_38], ebx
.text:0040381A mov     ebx, edi
.text:0040381C rol     ebx, 5
.text:0040381F mov     [esp+0B4h+var_A4], ebx
```

Figure 18

The corresponding ID of our machine is highlighted in figure 19:

```
Address   Hex                                              ASCII
00415688  37 00 39 00 37 00 33 00 33 00 33 00 30 00 33 00  7.9.7.3.3.3.0.3.
00415698  35 00 38 00 31 00 34 00 33 00 33 00 00 00 00 00  5.8.1.4.3.3.....
```

Figure 19

Two C2 servers and the port number have been decrypted by the process:

```
Address   Hex                                              ASCII
028FFCEC  34 00 34 00 33 00 00 00 95 65 F4 7D 50 27 58 2E  4.4.3....eô}P'X.
Address   Hex                                              ASCII
028FFD3C  31 00 32 00 35 00 2E 00 32 00 31 00 32 00 2E 00  1.2.5...2.1.2...
028FFD4C  31 00 33 00 32 00 2E 00 32 00 32 00 32 00 00 00  1.3.2...2.2.2...
Address   Hex                                              ASCII
028FFD1C  31 00 37 00 35 00 2E 00 31 00 30 00 30 00 2E 00  1.7.5...1.0.0...
028FFD2C  31 00 38 00 39 00 2E 00 31 00 37 00 34 00 00 00  1.8.9...1.7.4...
```

Figure 20

The inet_addr routine is utilized to convert the IP addresses of the C2 servers into proper addresses for the IN_ADDR structure:

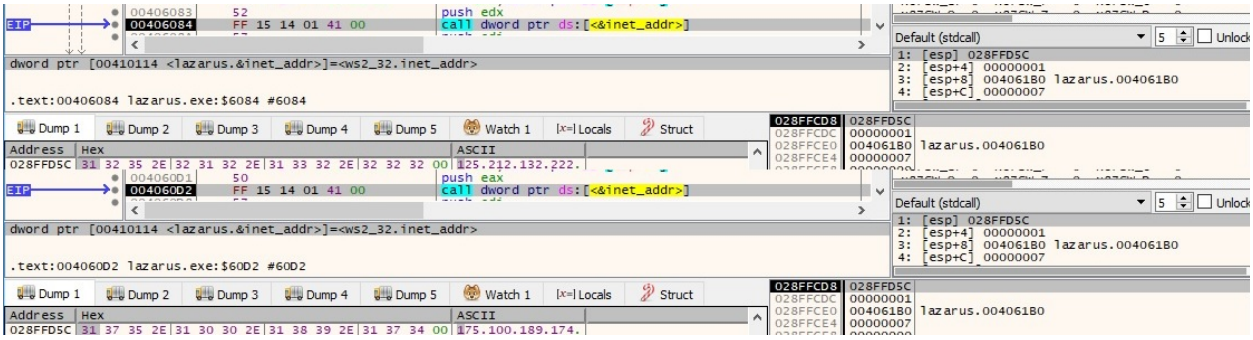


Figure 21

The sample extracts the valid drives on the system using the GetLogicalDriveStringsW API:

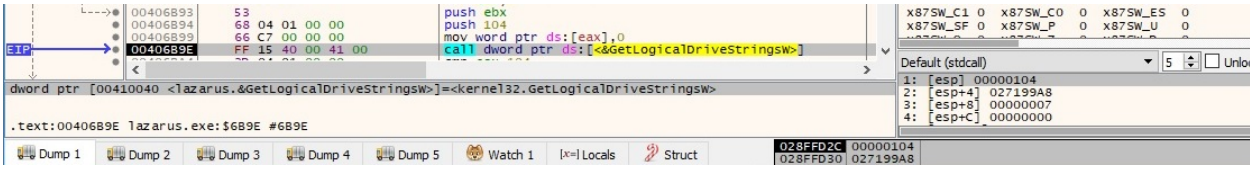


Figure 22

GetDriveTypeW is used to retrieve the type of the drives extracted above. The drives name and their type are saved to a buffer in the following form ("C 3" and "D 5"):

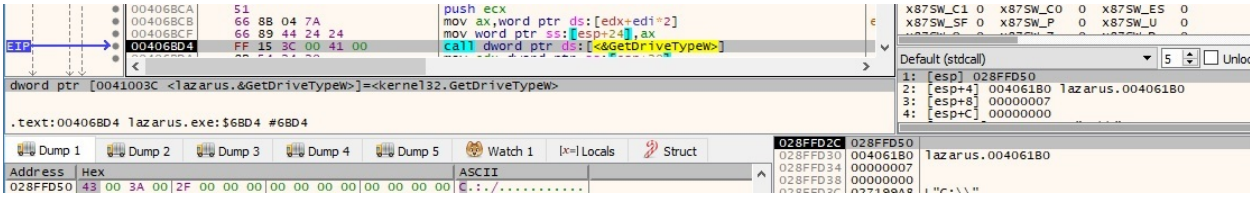


Figure 23

A new socket is created by the process (0x2 = **AF_INET**, 0x1 = **SOCK_STREAM** and 0x6 = **IPPROTO_TCP**):

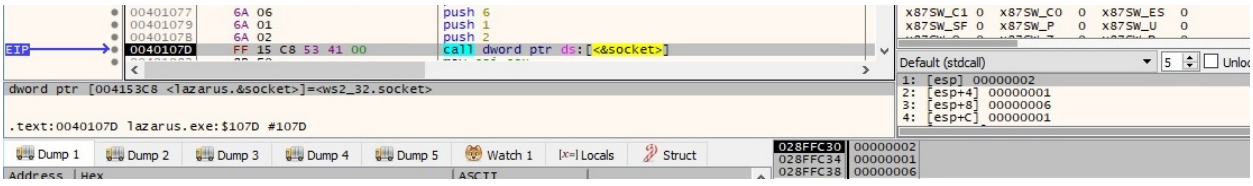


Figure 24

The malicious file enables the non-blocking mode for the socket using the ioctlsocket routine (0x8004667e = **FIONBIO**):

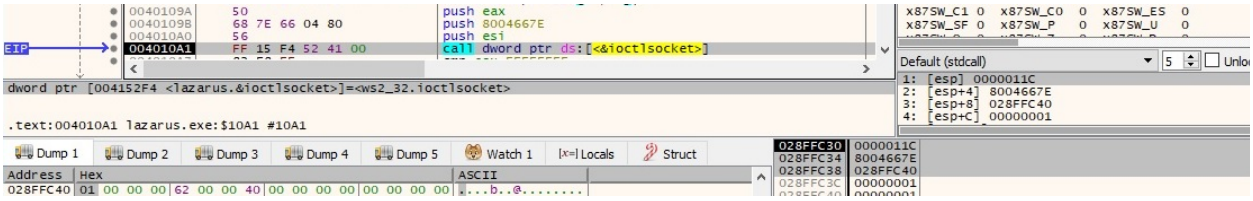


Figure 25

A new connection to 175.100.189.174 on port 443 is established by the process (if it's unsuccessful, it tries to connect to 125.212.132.222). It's important to mention that the network connections are simulated using FakeNet (https://github.com/fireeye/flare-fakenet-ng):

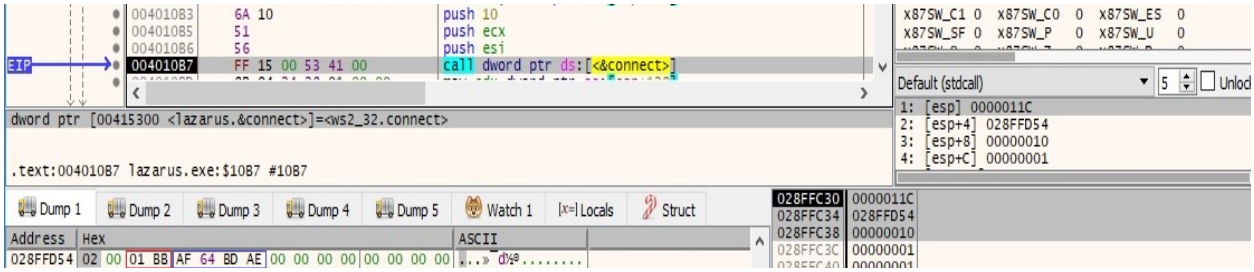


Figure 26

The select API is utilized to determine the status of the socket:

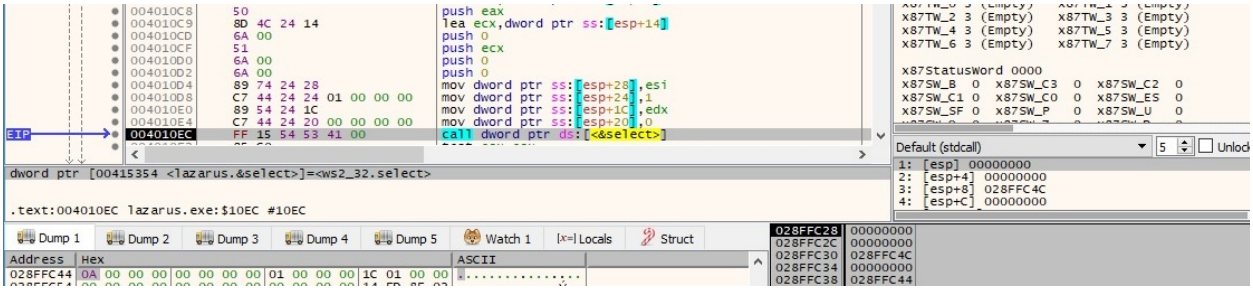


Figure 27

The blocking mode for the socket is enabled using the ioctlsocket routine:

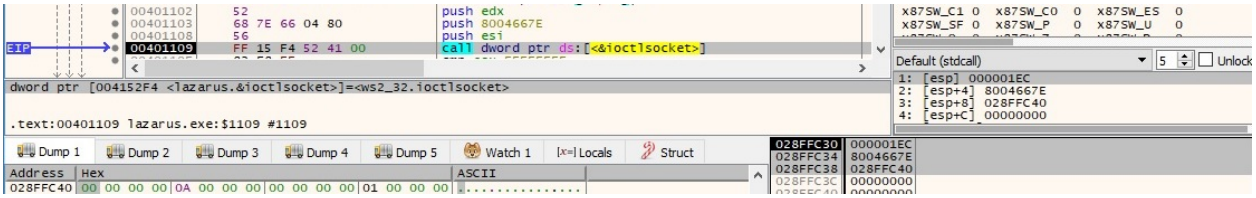


Figure 28

There is a call to GetTickCount followed by another one to the _rand function. The result of the operations is encrypted using the RC4 key presented before. The structure of the data sent to the server is "17 03 01 00 <buffer length> buffer":

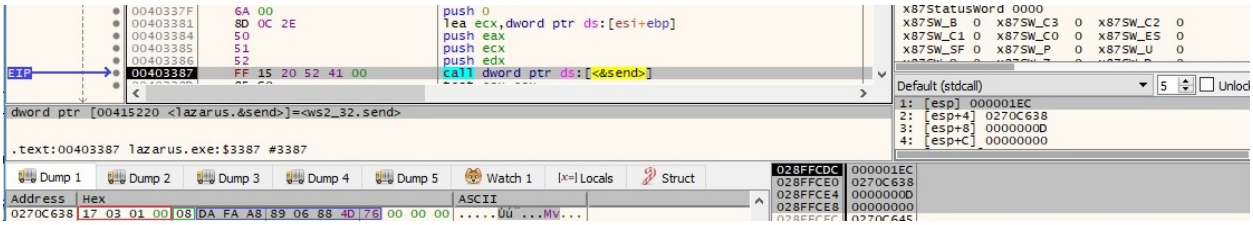


Figure 29

The process receives data from the socket by calling the recv function. It expects a structure such as "17 03 01 00 <buffer length>", and then other recv calls follow:

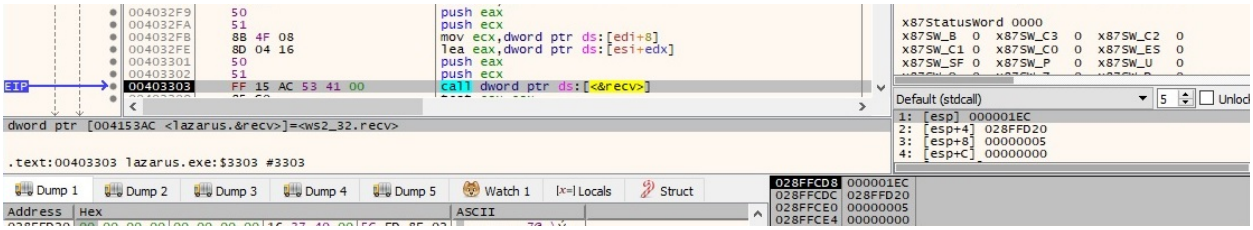


Figure 30

A new thread that will handle the RAT capabilities of the malware is created via a CreateThread API call:

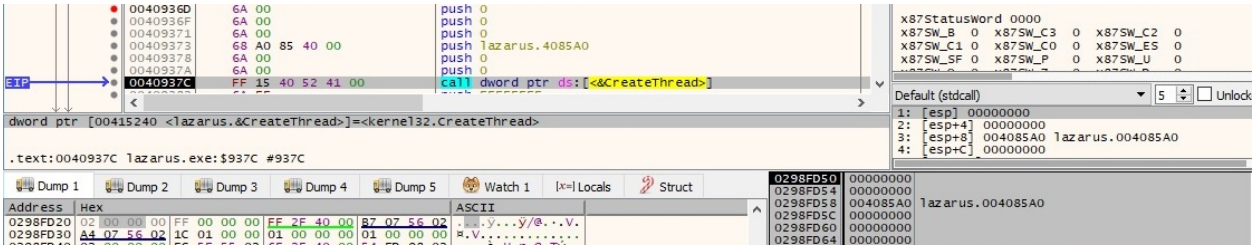


Figure 31

OllyDumpEx plugin is used to dump the process memory for further analysis, however, we still need to fix the IAT (import address table):

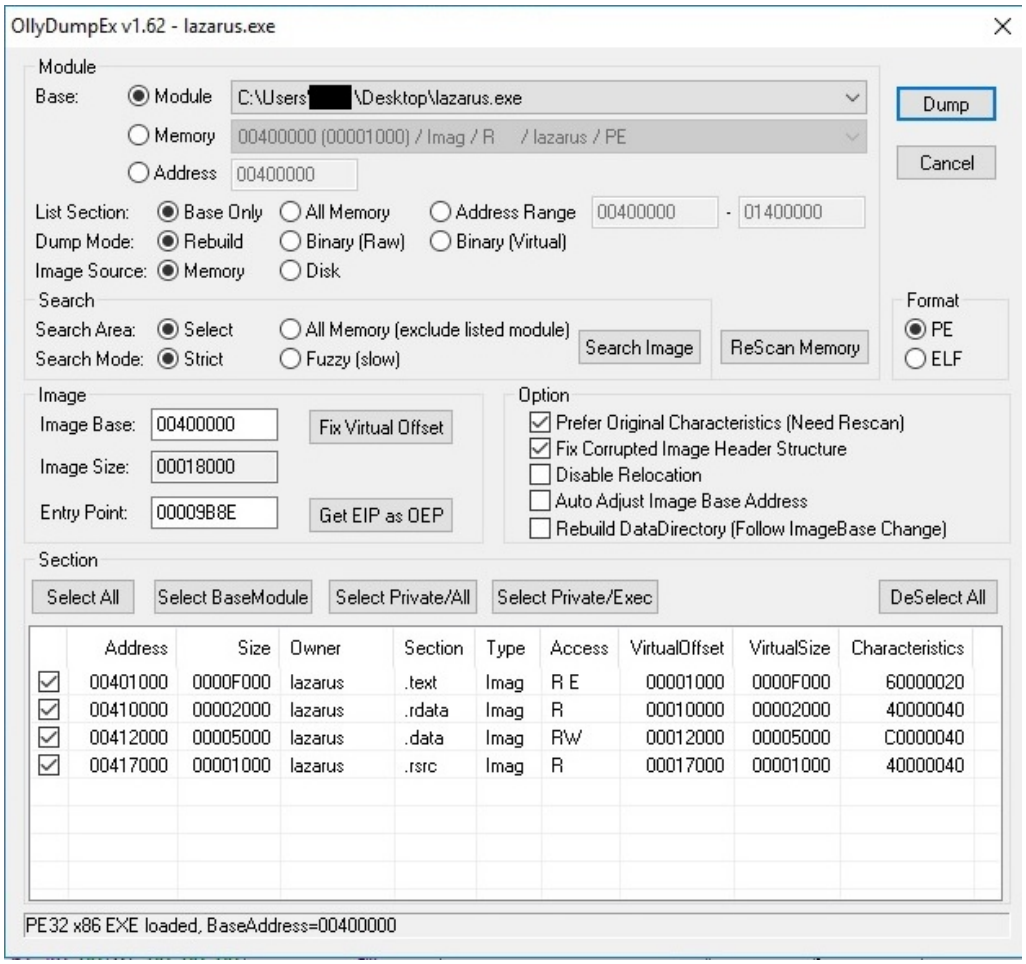


Figure 32

Scylla (https://github.com/NtQuery/Scylla) didn't help us in fixing the IAT, however Imports Fixer 1.6 (https://forum.tuts4you.com/files/file/1205-imports-fixer-legacy-archives/) has performed this task successfully:

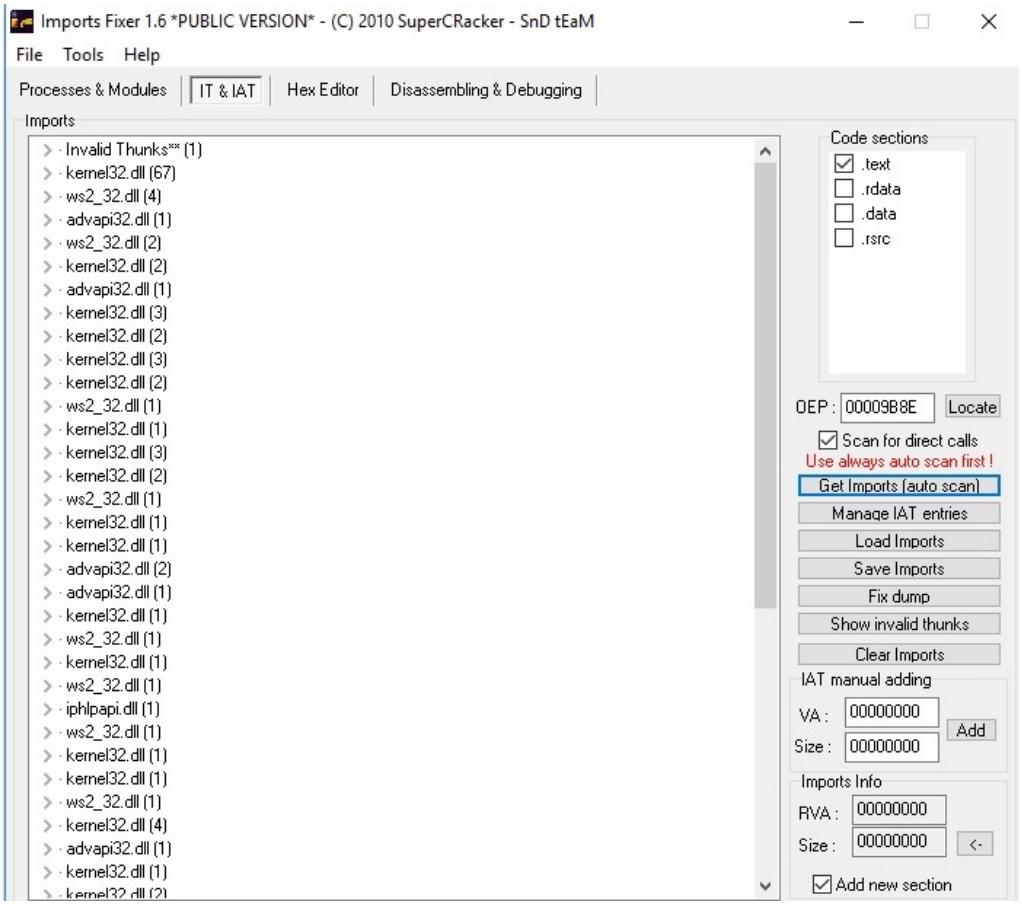


Figure 33

Depending on what command it receives from the C2 server, the malware implements 34 different cases regarding RAT functionalities (some of them have the same execution flow as we'll see later on):

```
.text:004085EF mov     edx, [esp+14h+var_4]
.text:004085F3 shr     edx, 1
.text:004085F5 mov     [ebx+edx*2], ax
.text:004085F9 mov     eax, [esp+14h+var_8]
.text:004085FD add     eax, 0FFFF00CCh ; switch 34 cases
.text:00408602 cmp     eax, 21h
.text:00408605 ja      def_408613      ; jumptable 00408613 default case, cases 65337,65340-65342,65347,65351-65354,65358,65361,65362
```

Figure 34

THREAD ACTIVITY – SUB_4085A0 FUNCTION

We will describe each execution flow depending on the EAX value, which is computed based on the data the malware receives from the C2 server (figure 34).

EAX = 0 – traverse a directory received from the C2 server and send the files name to the C2

The process traverses the targeted directory using the FindFirstFileW and FindNextFileW functions:

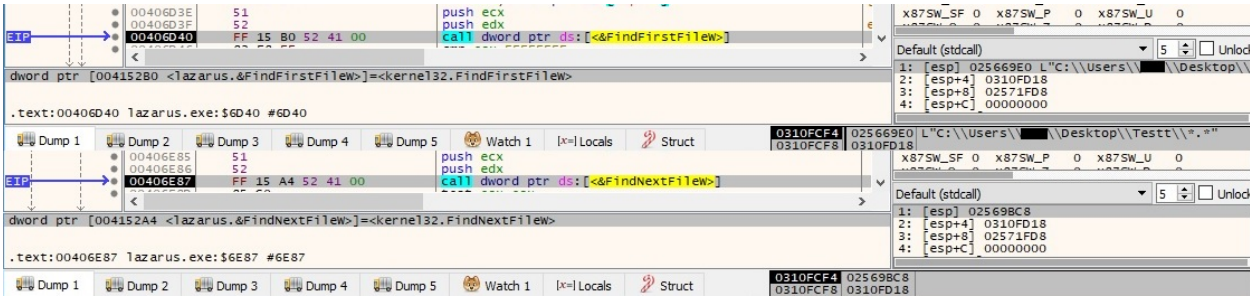


Figure 35

The directory name is encrypted using the XOR algorithm and sent to the C2 server. The file name is encrypted as well (note the case number):

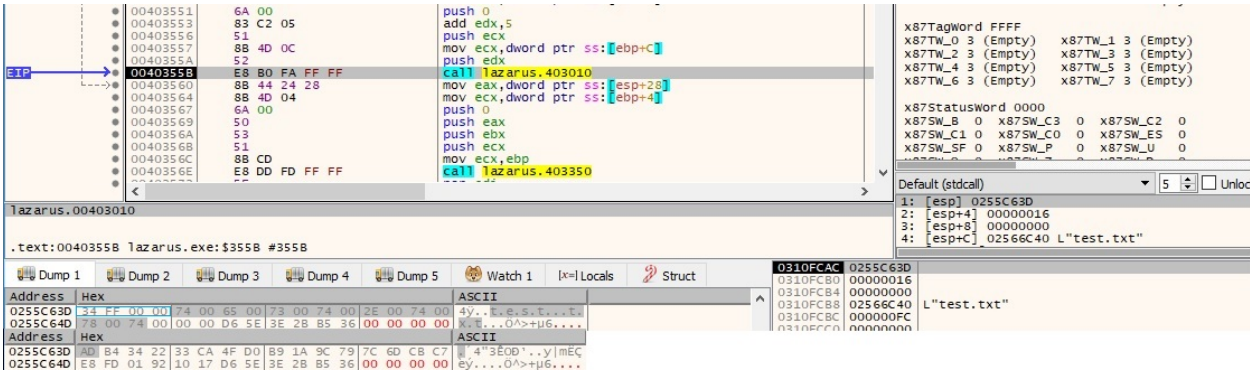


Figure 36

The encrypted file name is transmitted to the server in the structure "17 03 01 00 <encrypted filename length> encrypted filename", as shown in figure 37.

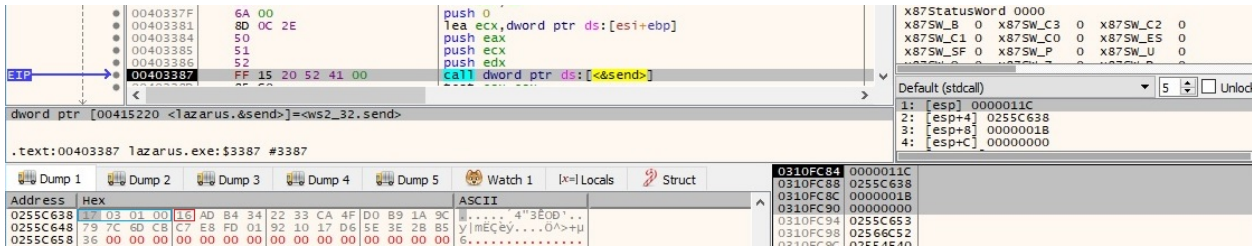


Figure 37

EAX = 1 – enumerate the processes and the modules of the current process. Extract the process creation time, the exit time of the process, the amount of time that the process has executed in user/kernel mode. Open the access token associated with a process and determine if the user belongs to a privileged group

The binary takes a snapshot of the processes (0x2 = **TH32CS_SNAPPROCESS**):

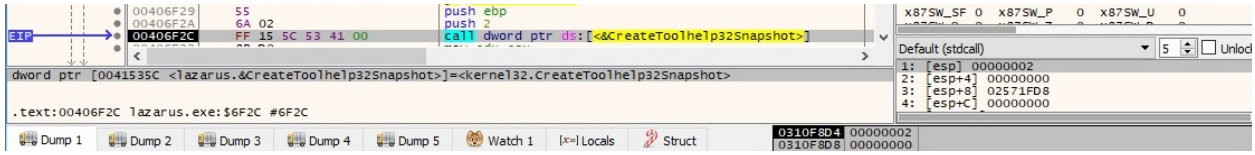


Figure 38

The processes are enumerated using the Process32FirstW and Process32NextW APIs:

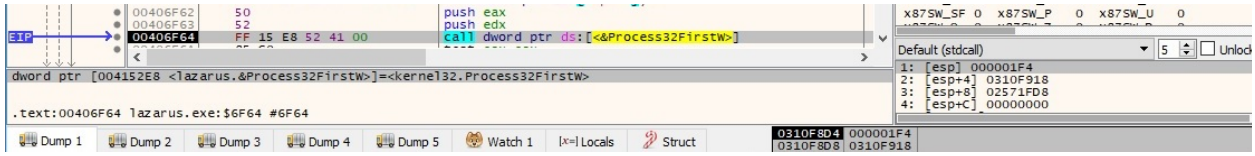


Figure 39

OpenProcess is utilized to open the local process object (0x410 = **PROCESS_QUERY_INFORMATION** | **PROCESS_VM_READ**):

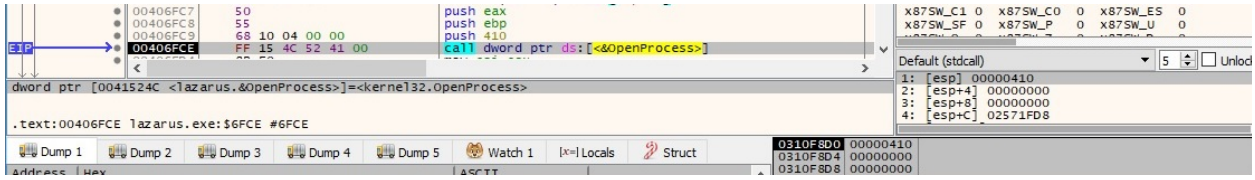


Figure 40

The process name is XOR-ed and exfiltrated to the C2 server using the send routine:

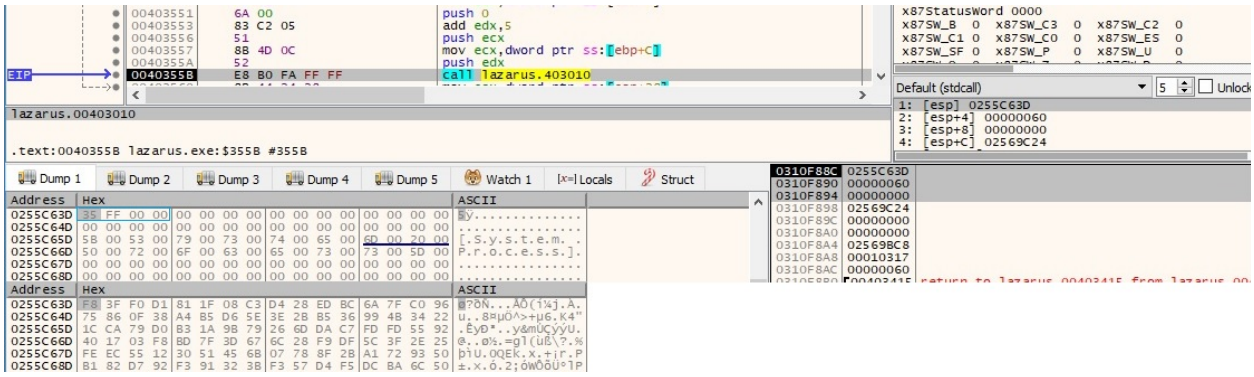


Figure 41

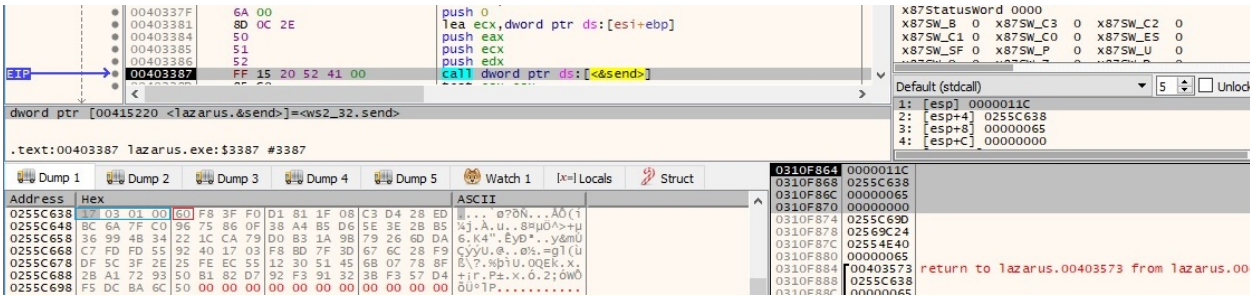


Figure 42

The file takes a snapshot of the current process that includes all its modules (0x8 = **TH32CS_SNAPMODULE**):

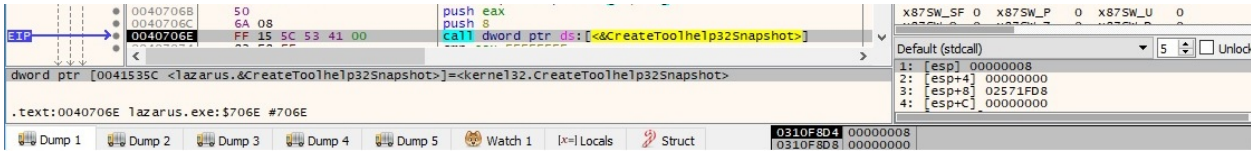


Figure 43

The modules are enumerated using the Module32FirstW and Module32NextW APIs:

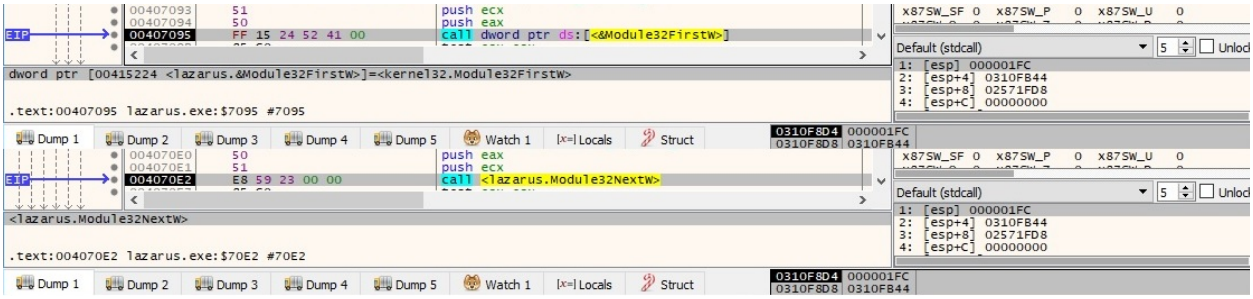


Figure 44

The module name is XOR-ed and exfiltrated to the C2 server using the send routine:

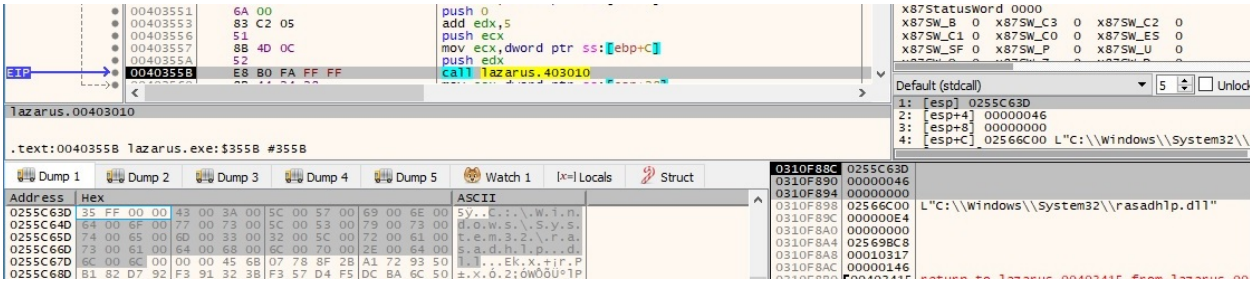


Figure 45

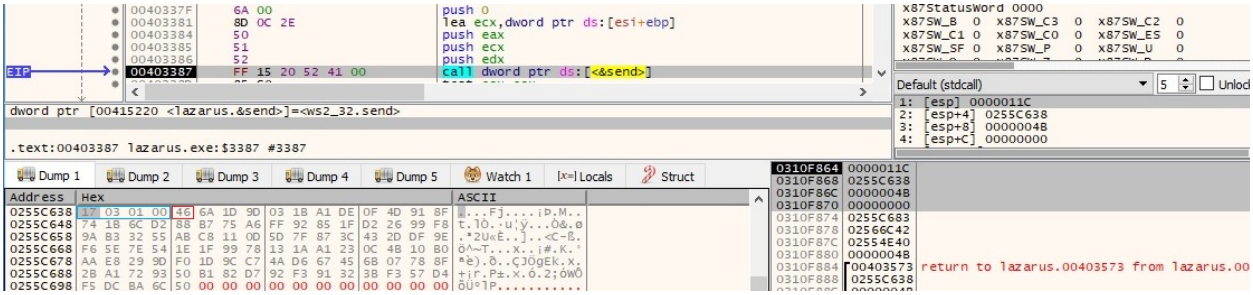


Figure 46

GetProcessTimes is used to retrieve timing information for the enumerated process:

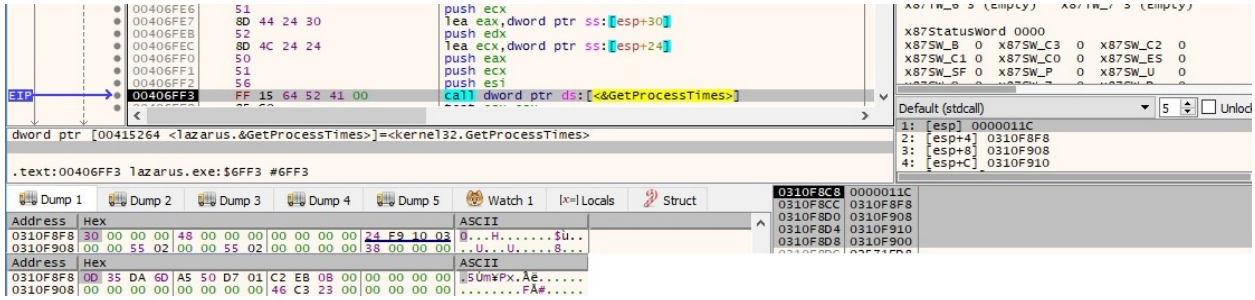


Figure 47

The malicious process converts the creation time of the enumerated process to system time format:

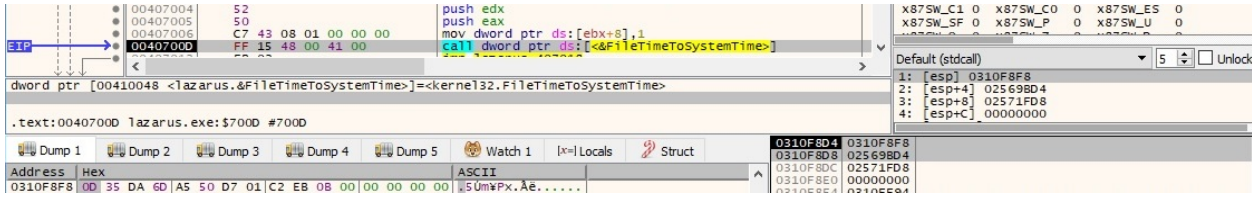


Figure 48

The OpenProcessToken routine is used to open the access token associated with the enumerated process (0x8 = **TOKEN_QUERY**):

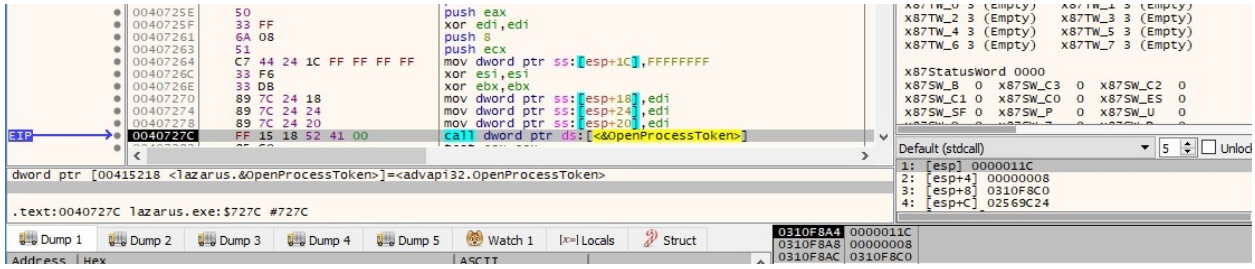


Figure 49

GetTokenInformation is utilized to retrieve the user account of the token, as shown below (0x1 = **TokenUser**):

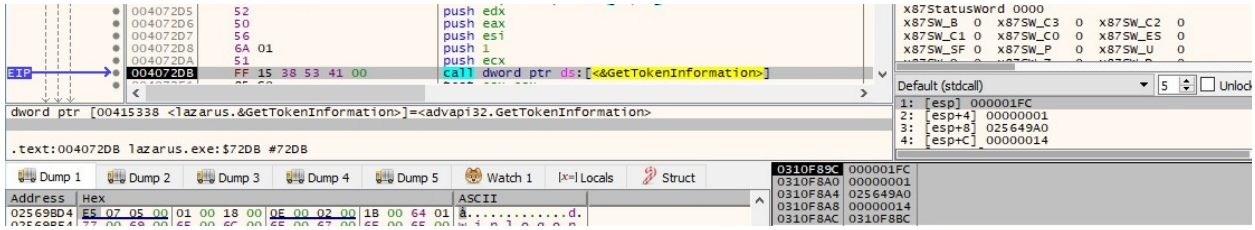


Figure 50

The binary uses the LookupAccountSidW API to retrieve the account that corresponds to a SID and the name of the first domain on which the SID was found:

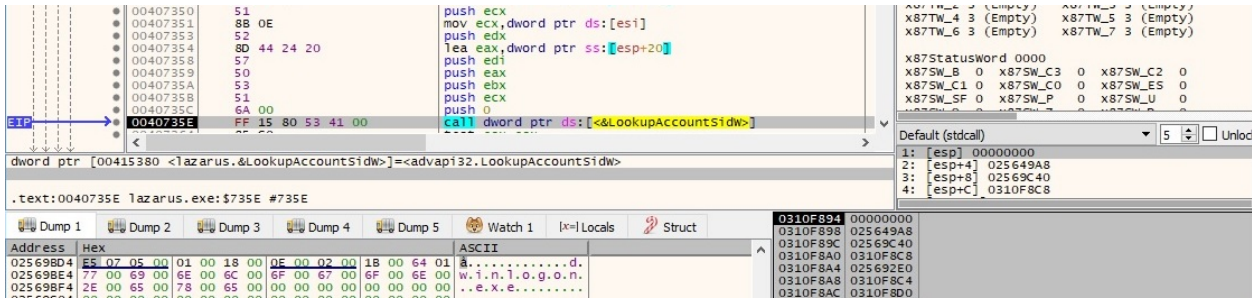


Figure 51

The Terminal Services session identifier associated with the token from above is extracted using the GetTokenInformation function (0xc = **TokenSessionId**):

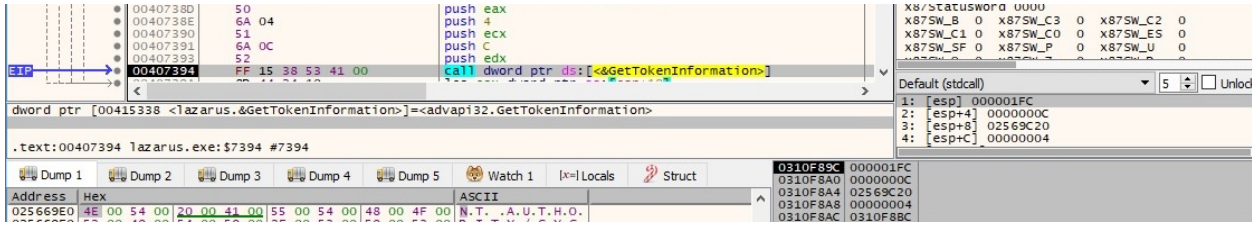


Figure 52

Whether the malware has successfully opened a process and extracted its creation time, the process ID along with the creation time and process name are encrypted using the XOR algorithm and transmitted to the C2 server:

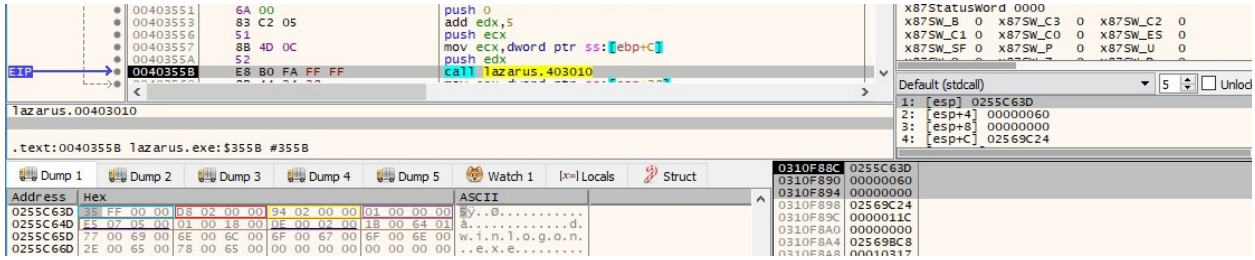


Figure 53

EAX = 2 – kill a specific process obtained from the C2 server

The processes are enumerated using the Process32FirstW and Process32NextW functions:

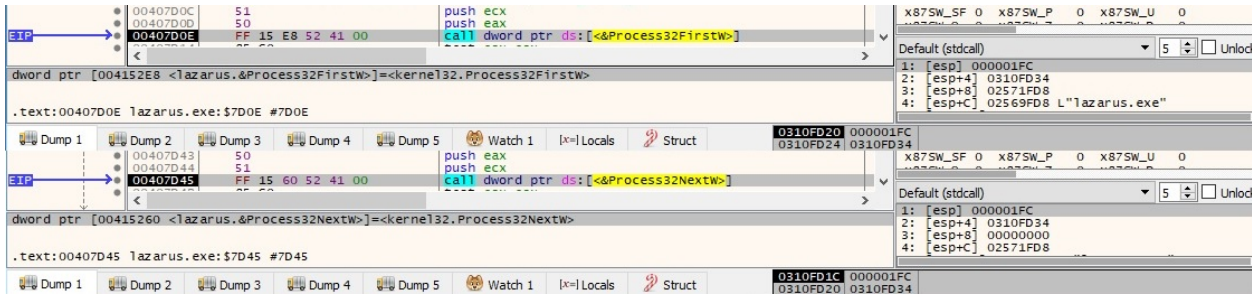


Figure 54

The malware opens the targeted process via a call to OpenProcess (0x100001 = **SYNCHRONIZE** | **PROCESS_TERMINATE**):

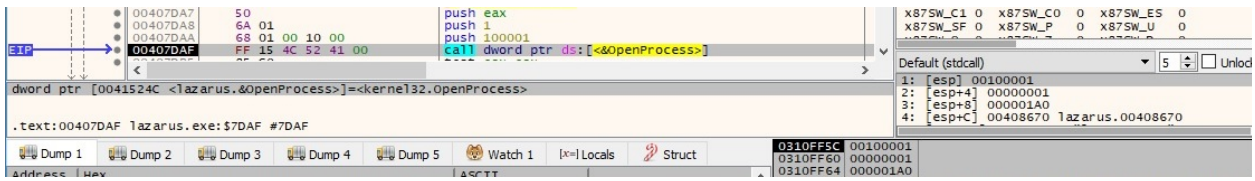


Figure 55

TerminateProcess is utilized to kill the targeted process and all of its threads:

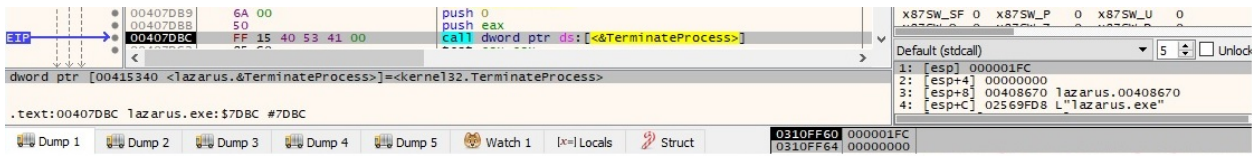


Figure 56

EAX = 3 – create a specific process obtained from the C2 server

A new process whose name is obtained from the C2 server is created using the CreateProcessW API (0x8000000 = **CREATE_NO_WINDOW**):

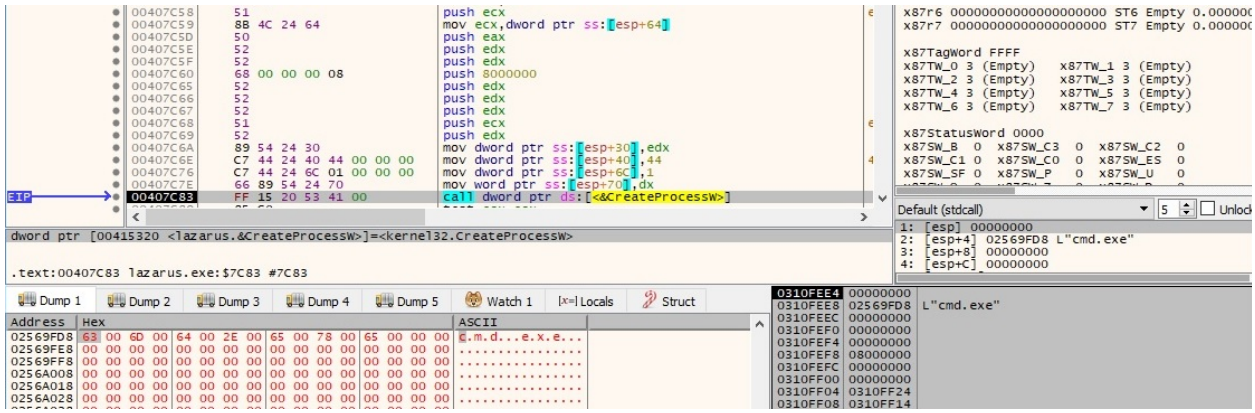


Figure 57

EAX = 4 – create a process obtained from the C2 server with a specific token

The WTSQueryUserToken routine is utilized to obtain the primary access token of the user specified by session 0:

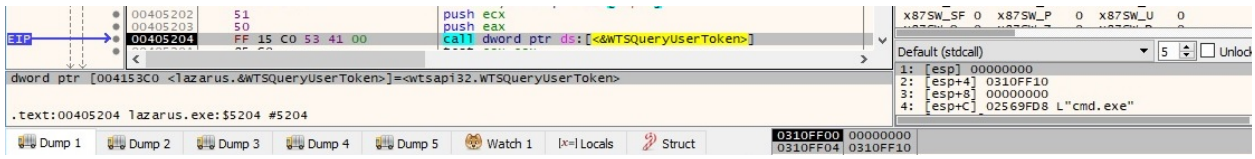


Figure 58

The file creates a new process that runs in the security context of the user represented by the above token:

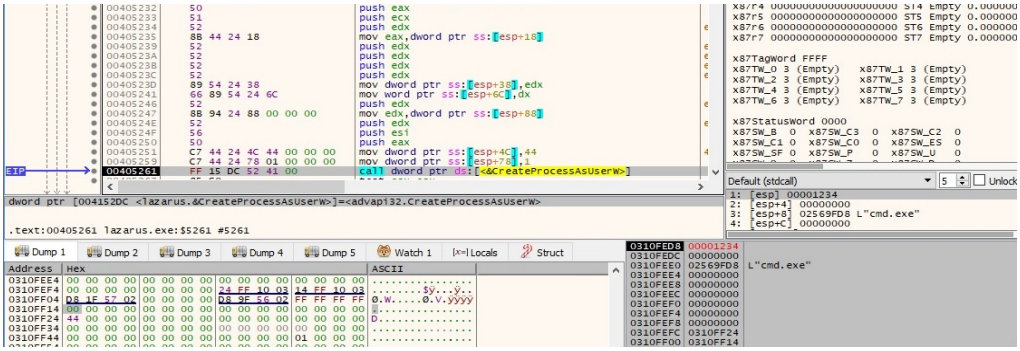


Figure 59

EAX = 5, 8, 9, 10, 15, 19, 20, 21, 22, 23, 25, 26, 29, 30 – send 4 encrypted bytes to the C2 server

The executable XOR-ed the "0xFFFFFFFF" number with some key bytes and sends the result to the C2 server:

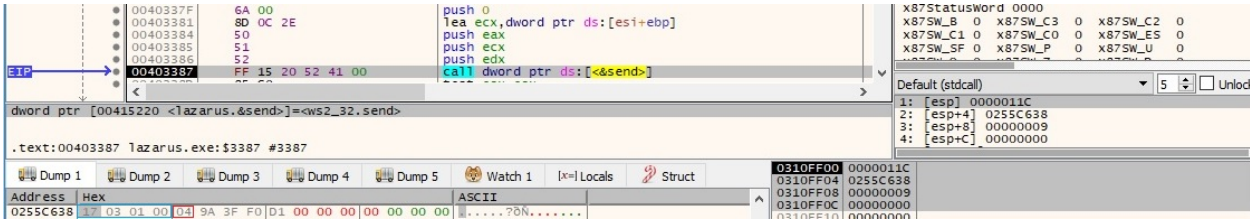


Figure 60

EAX = 6 – create and populate a new file and perform timestomping

A new file whose name is received from the C2 server is created by the malware:

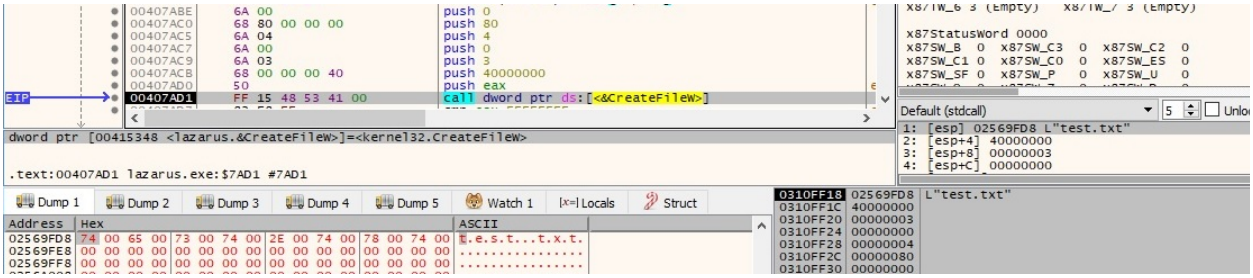


Figure 61

The malicious process opens the "cmd.exe" file:

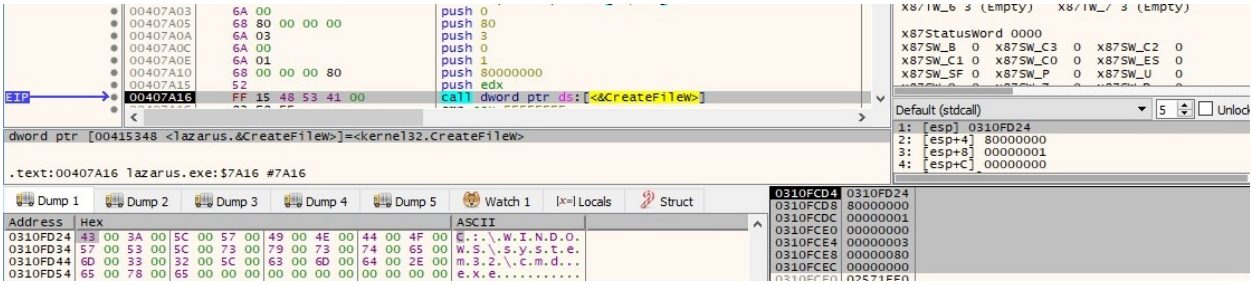


Figure 62

The created, last accessed and last modified times of the "cmd.exe" file are extracted using the GetFileTime API:

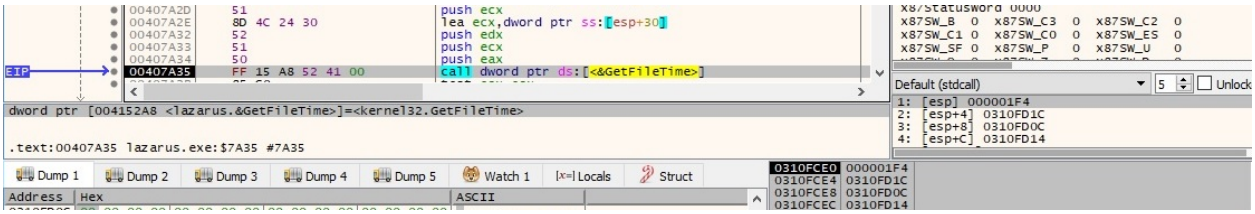


Figure 63

The created, last accessed, and last modified times of the newly created file are set to the ones extracted above:

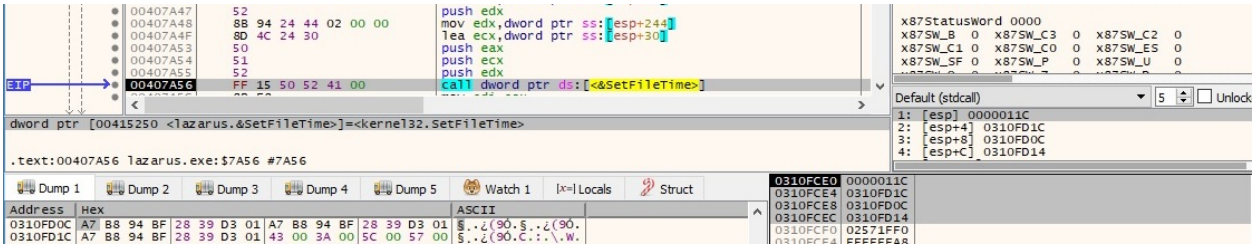


Figure 64

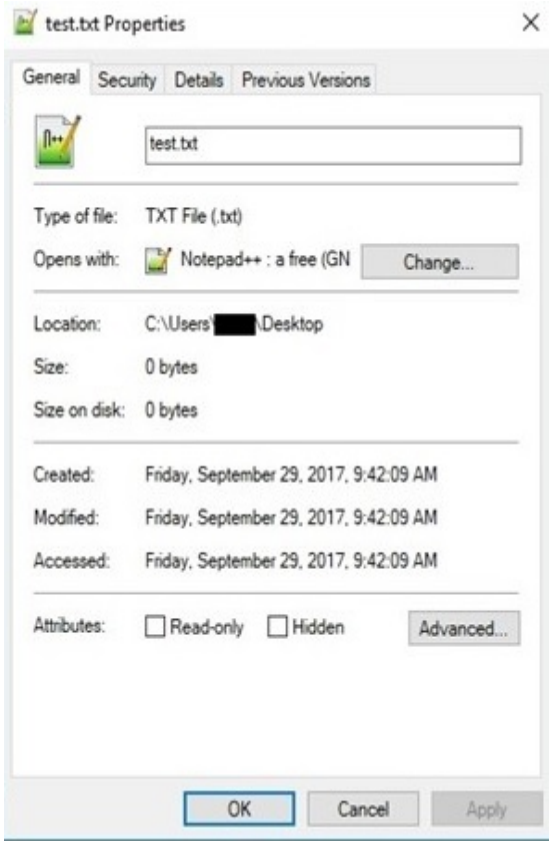


Figure 65

The file is populated with content received from the server, as shown in figure 66:

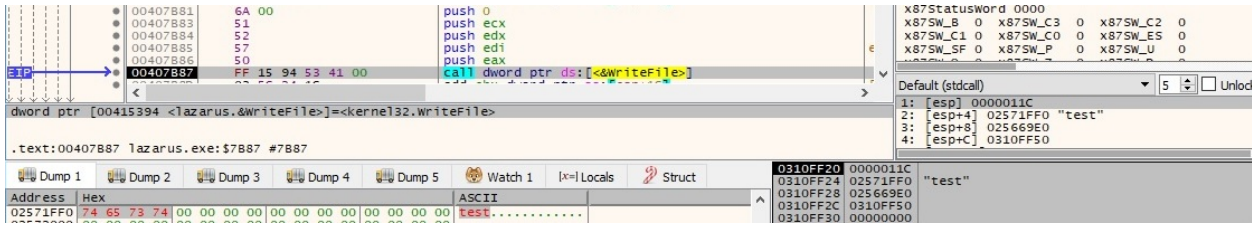


Figure 66

EAX = 7 – read file content, extract file times and exfiltrate them to the C2 server

The process opens the targeted file using the CreateFileW routine:

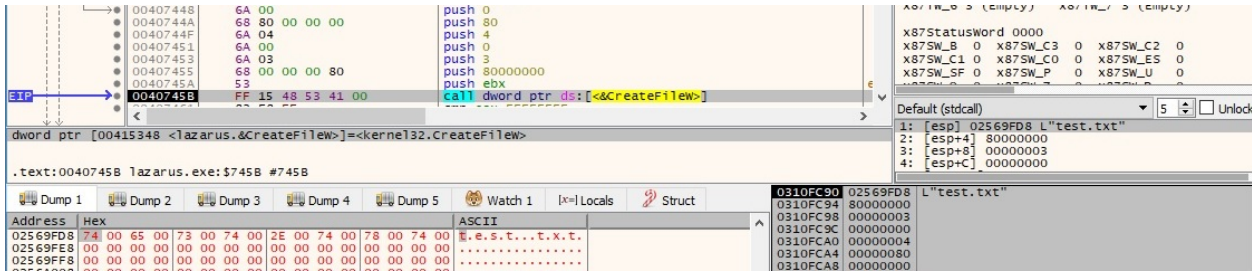


Figure 67

The created, last accessed, and last modified times of the above file are extracted using the GetFileTime API:

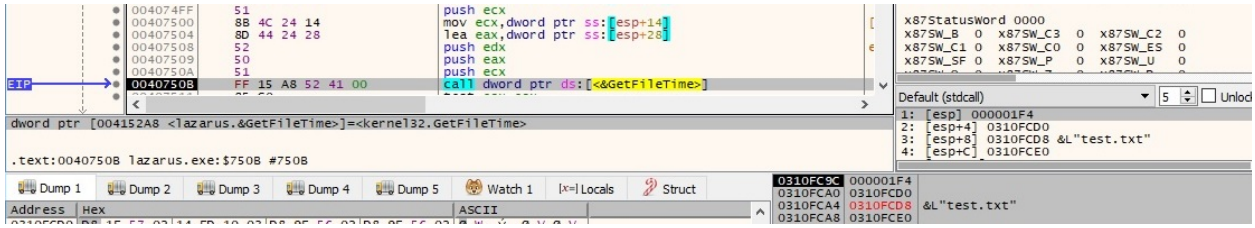


Figure 68

ReadFile is utilized to retrieve the file content:

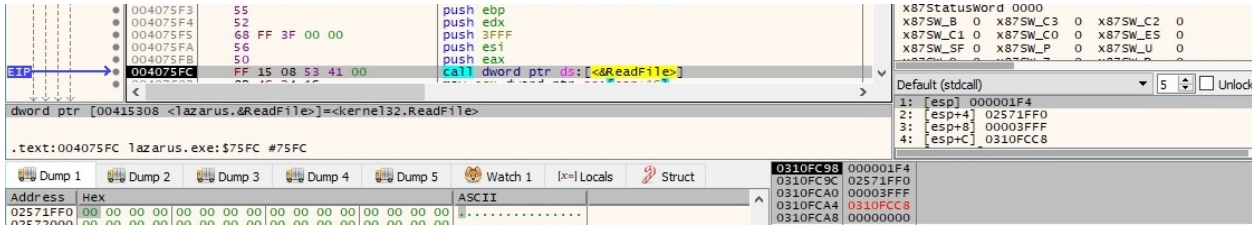


Figure 69

The filename, file times, and file content are encrypted using the XOR operator and sent to the C2 server:

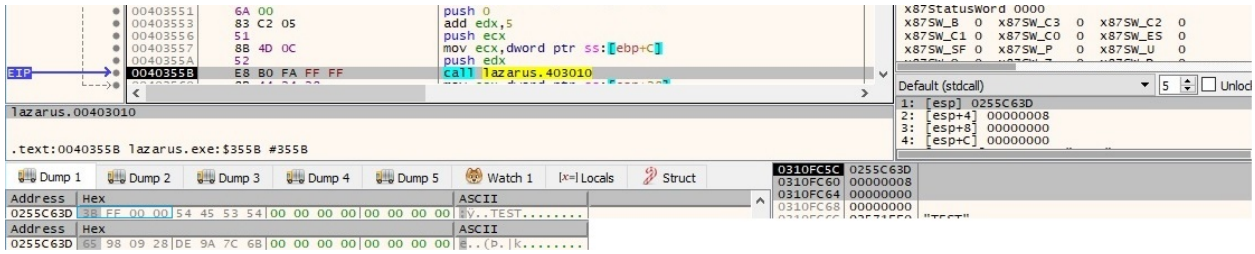


Figure 70

EAX = 11 – open file, extract file times and create a new file and modify its file times

CreateFileW is used to open a file specified by the C2 server:

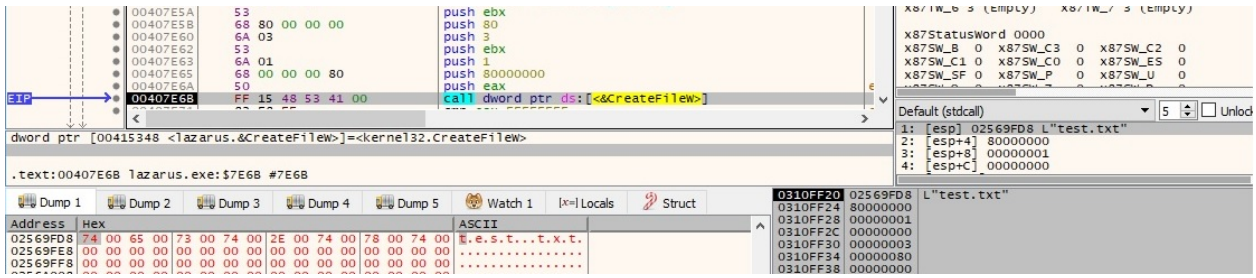


Figure 71

The created, last accessed, and last modified times of the above file are extracted using the GetFileTime function:

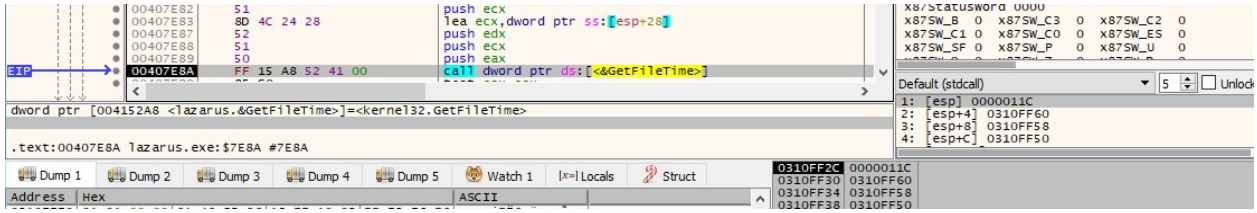


Figure 72

A new file designated by the C2 server is created by the binary:

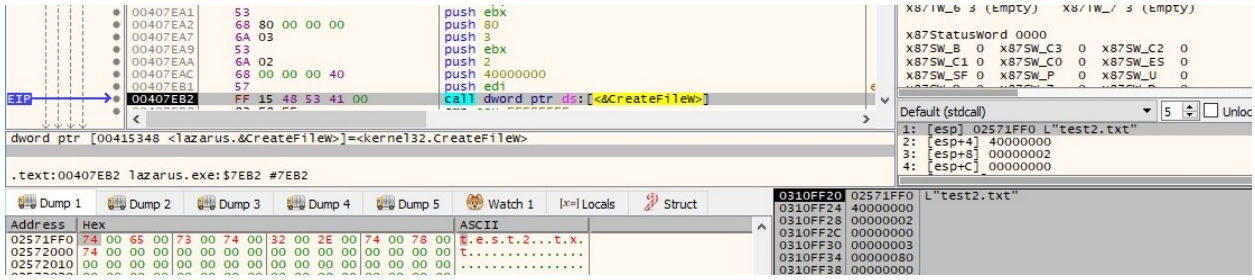


Figure 73

The SetFileTime routine is utilized to set the created, last accessed, and last modified times for the new file to the values extracted before:

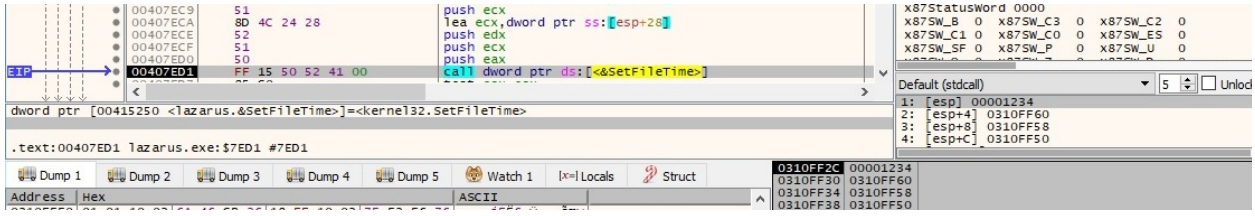


Figure 74

EAX = 12, 14 – convert the system time to a calendar value and write it into memory

The malware extracts the system time and converts it to a calendar value:

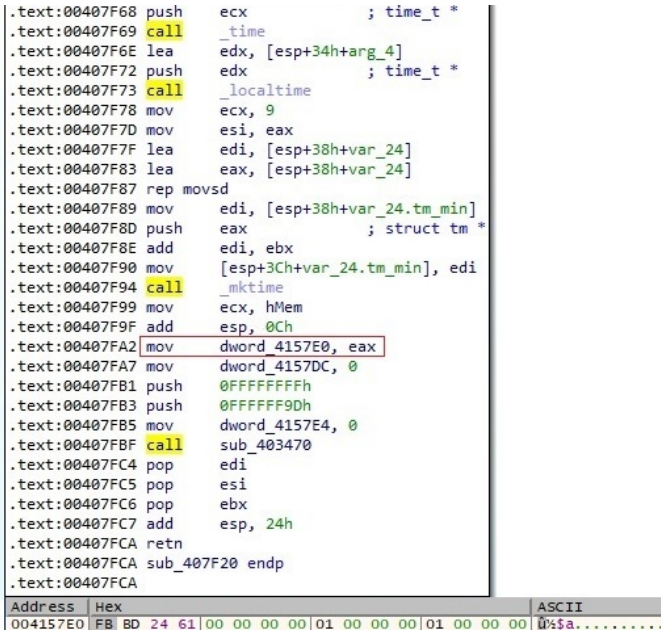


Figure 75

EAX = 13 – convert the system time to a calendar value and write it into memory (different function from above)

The malware extracts the system time and converts it to a calendar value:

```
.text:00407FE6 push    ecx             ; time_t *
.text:00407FE7 call    _time
.text:00407FEC lea     edx, [esp+38h+arg_0]
.text:00407FF0 push    edx             ; time_t *
.text:00407FF1 call    _localtime
.text:00407FF6 mov     ecx, 9
.text:00407FFB mov     esi, eax
.text:00407FFD lea     edi, [esp+3Ch+var_24]
.text:00408001 lea     eax, [esp+3Ch+var_24]
.text:00408005 rep movsd
.text:00408007 mov     edi, [esp+3Ch+var_24.tm_min]
.text:0040800B push    eax             ; struct tm *
.text:0040800C add     edi, ebx
.text:0040800E mov     [esp+40h+var_24.tm_min], edi
.text:00408012 call    _mktime
.text:00408017 mov     ecx, hMem
.text:0040801D add     esp, 10h
.text:00408020 mov     dword_4157E0, eax
.text:00408025 mov     dword_4157DC, 1
.text:0040802F push    0FFFFFFFFh
.text:00408031 push    0FFFFFF9Dh
.text:00408033 mov     dword_4157E4, 0
.text:0040803D call    sub_403470
.text:00408042 pop     edi
.text:00408043 pop     esi
.text:00408044 pop     ebx
.text:00408045 add     esp, 24h
.text:00408048 retn
.text:00408048 sub_407FD0 endp
.text:00408048

Address  Hex                                              ASCII
004157E0 CD A0 23 61 00 00 00 00 01 00 00 00 01 00 00 00  I #a............
```

Figure 76

EAX = 16 – create and populate a new file and move it to a newly generated file

The FindFirstFileW API is utilized to search the current directory for a file name pushed as a parameter:

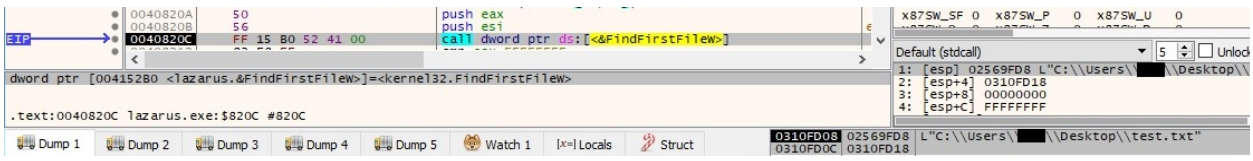


Figure 77

The process creates the new file mentioned above using the CreateFileW function:

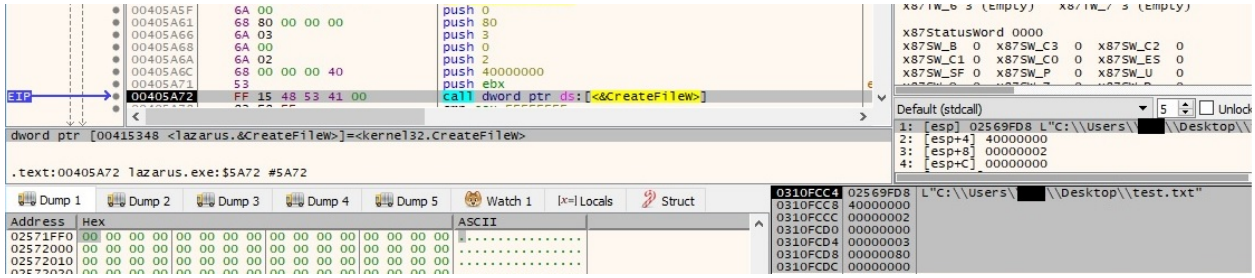


Figure 78

Four NULL bytes are written in the file created above:

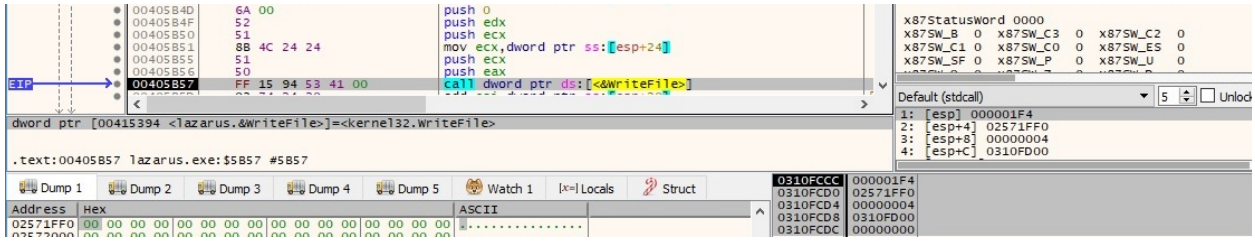


Figure 79

The GetTickCount and _rand functions are used to generate eight pseudo-random low characters. The binary moves the file from above to a new one (0x8 = **MOVEFILE_WRITE_THROUGH**):

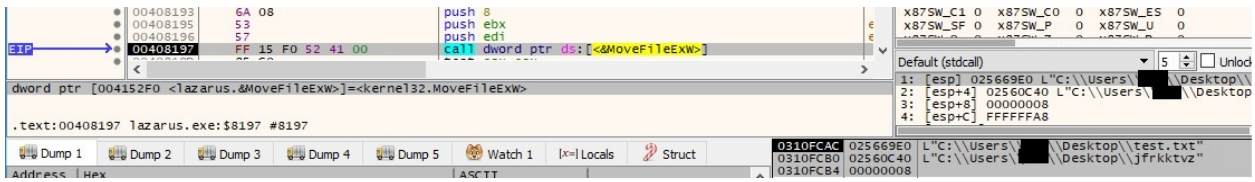


Figure 80

EAX = 17 – execute a Windows command and send the output to the C2 server

The %TEMP% directory is retrieved using the GetTempPathW routine:

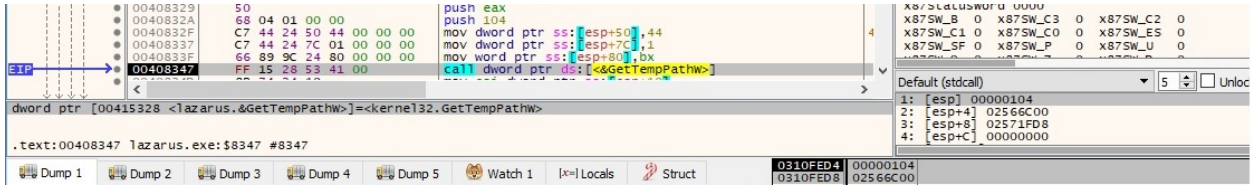


Figure 81

The executable creates a new temporary file, which starts with "CM", as shown in figure 82.

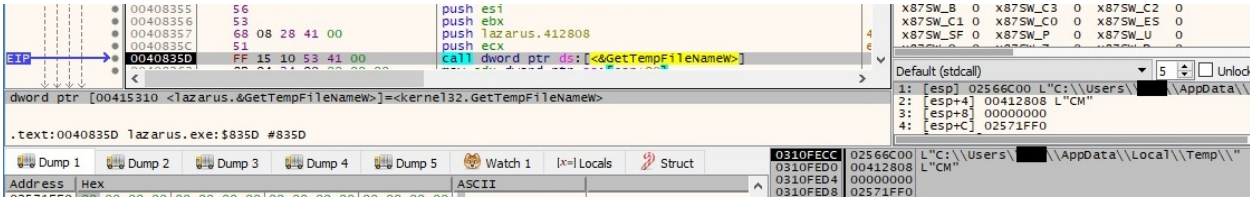


Figure 82

The malware executes a Windows command received from the C2 server and stores the output into the temporary file created above:

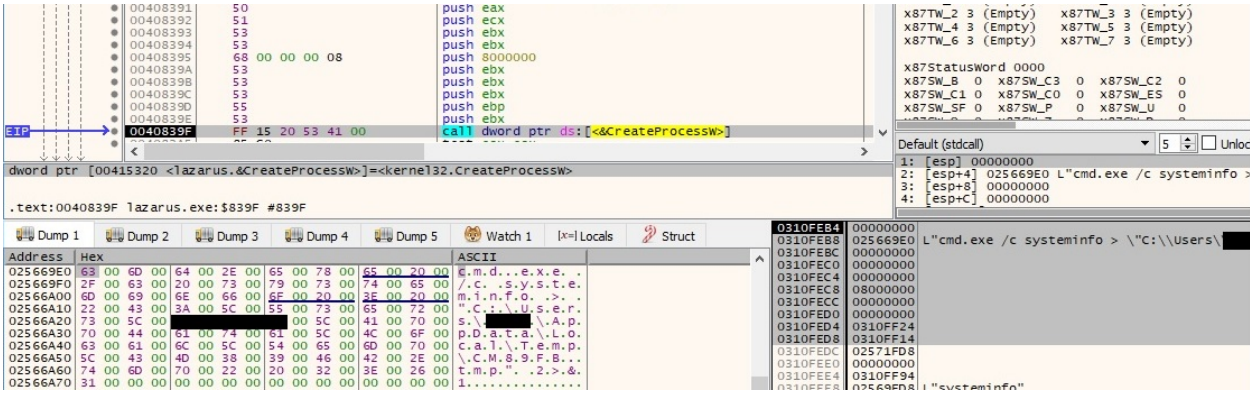


Figure 83

ReadFile is utilized to read data from the above file and store it into memory:

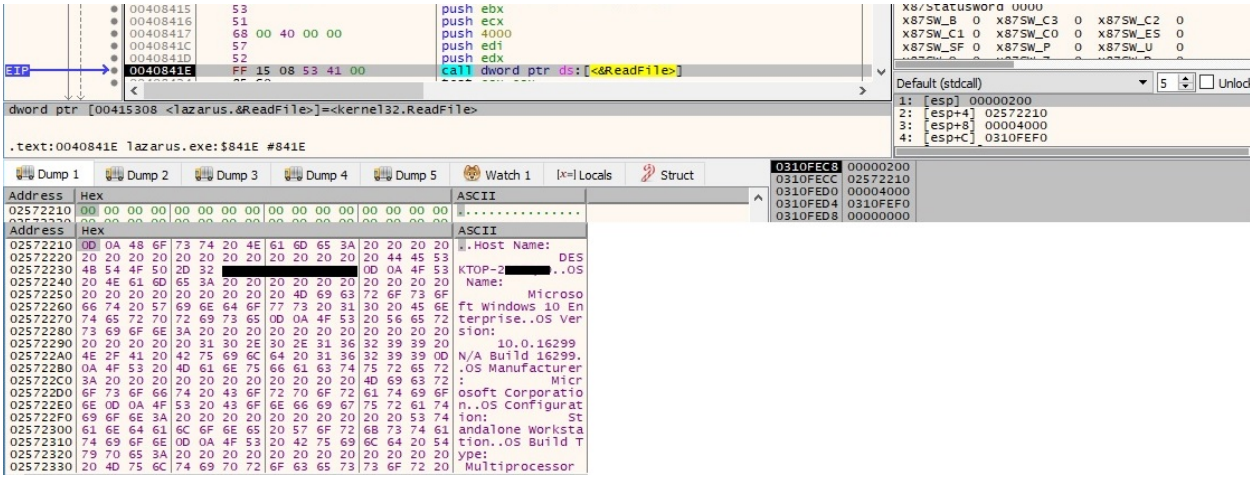


Figure 84

The output of the Windows command is XOR-ed with a buffer that was used during multiple XOR operations and exfiltrated to the C2 server:

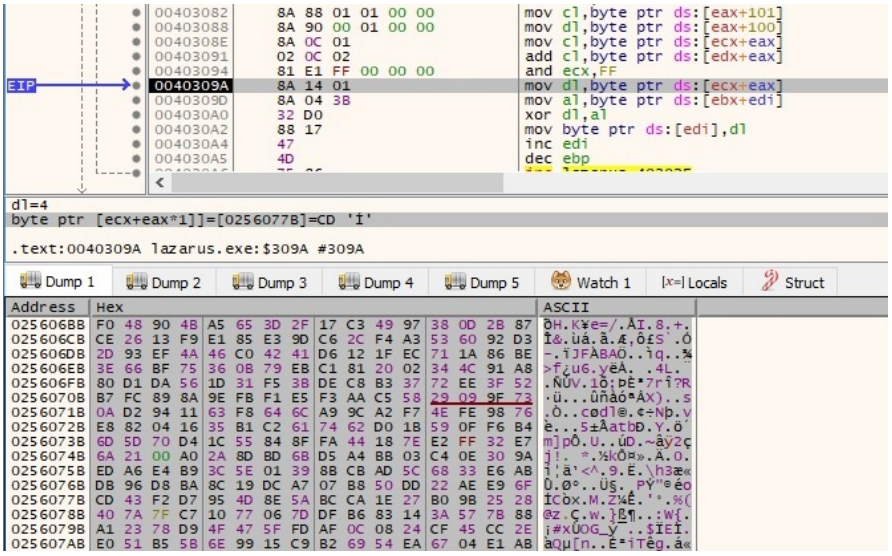


Figure 85

The binary kills the spawned process if it's still running:

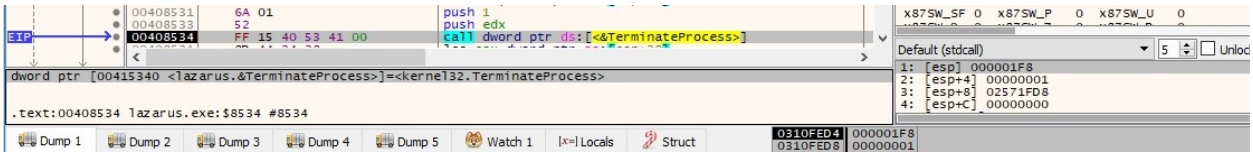


Figure 86

The temporary file is deleted by the malware:

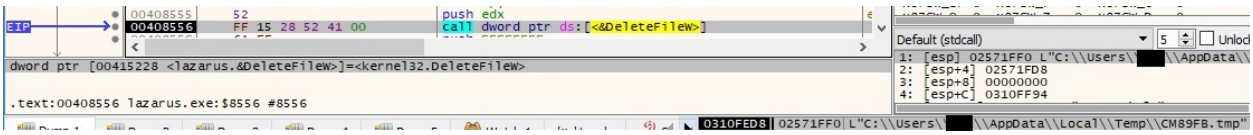


Figure 87

EAX = 18 – connect to a specific IP on a port received from the C2 server

The binary expects an argument such as "100.101.102.103:5555". It converts the port number from string to integer:

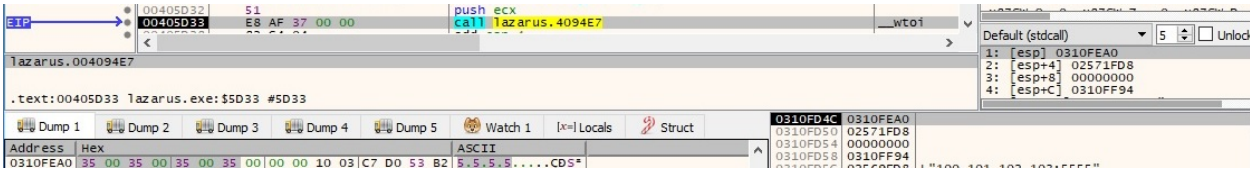


Figure 88

The inet_addr function is used to transform the IP address into a proper address for the IN_ADDR structure:

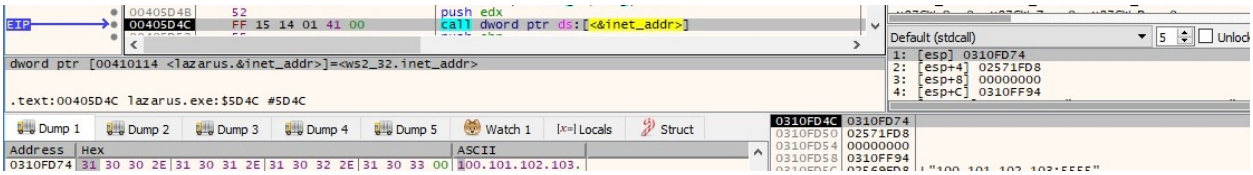


Figure 89

A new socket is created by the executable (0x2 = **AF_INET**, 0x1 = **SOCK_STREAM** and 0x6 = **IPPROTO_TCP**):

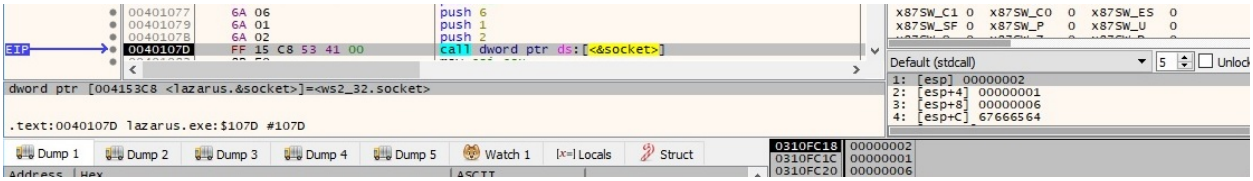


Figure 90

The file enables the non-blocking mode for the socket using the ioctlsocket routine (0x8004667e = **FIONBIO**):

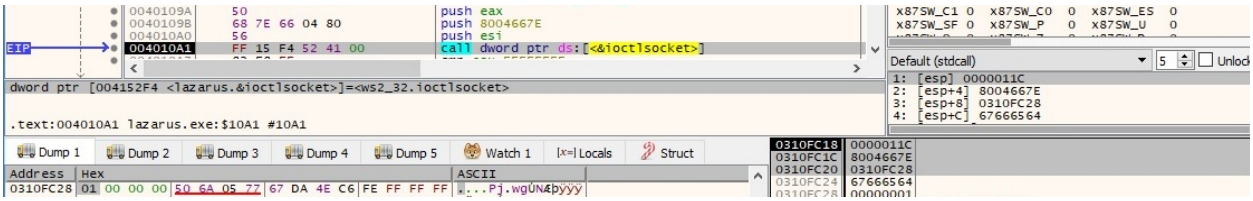


Figure 91

A new connection to the socket is established by the malware:

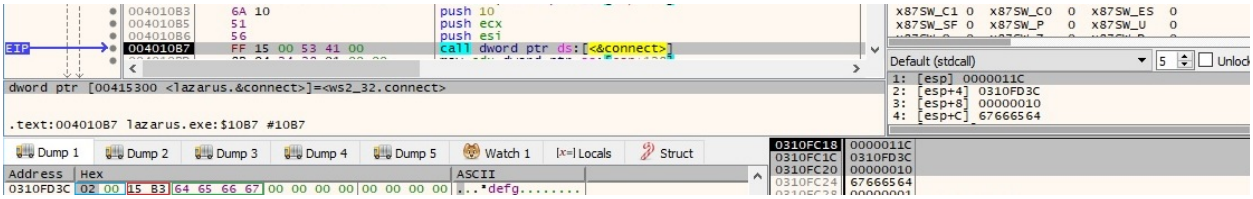


Figure 92

The TCP linger is set to 1 using the setsockopt API (0xffff = **SOL_SOCKET** and 0x80 = **SO_LINGER**), as shown in figure 93.

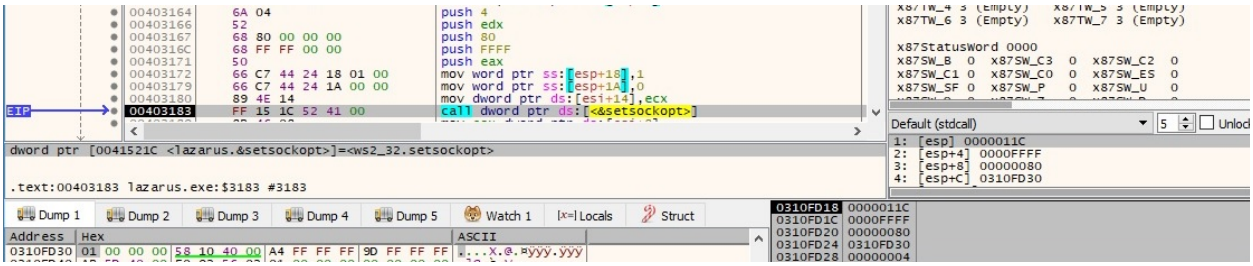


Figure 93

EAX = 24 – encrypt the C2 IP addresses using the XOR operator and send the result to the C2 server

The buffer that contains the C2 IP addresses and the port number is encrypted with the XOR operation:

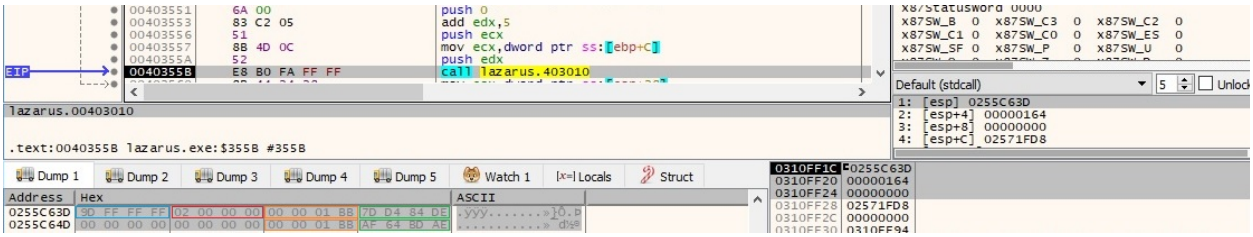


Figure 94

The encrypted content is transmitted to the C2 server via a send function call:

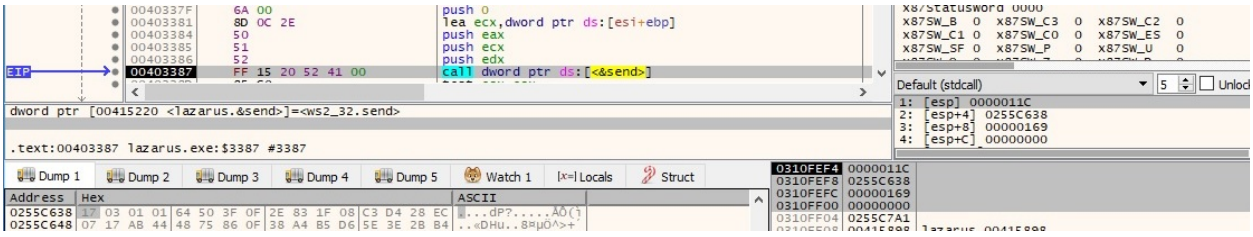


Figure 95

EAX = 27 – move an existing directory provided by the C2 server to the Desktop directory

The FindFirstFileW function is used to search the current directory for a subdirectory pushed as a parameter, as shown in figure 96.

Figure 96

The process tries to move the above directory to the Desktop folder (0x8 = **MOVEFILE_WRITE_THROUGH**):

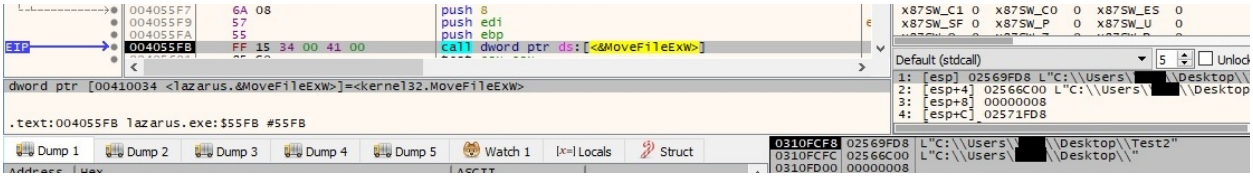


Figure 97

EAX = 28 – traverse an existing directory or extract the drive type and the amount of free space on the disk

Whether the parameter provided by the server is a folder name, then the process traverses the directory using the FindFirstFileW and FindNextFileW APIs and send the status (an encrypted buffer) to the C2 server:

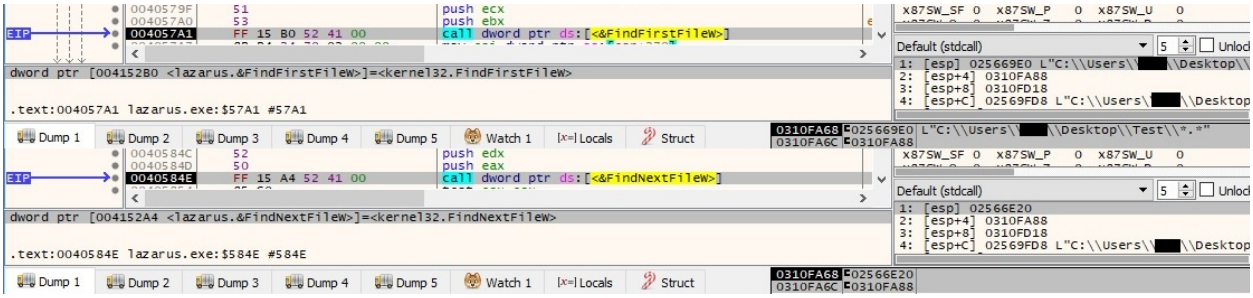


Figure 98

Whether the parameter provided by the server is a disk drive, the file retrieves the drive type using the GetDriveTypeW routine, as shown in figure 99.

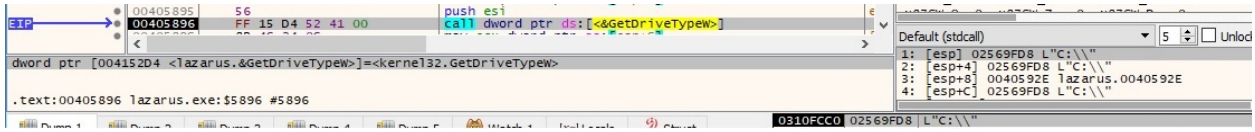


Figure 99

The binary gets the total amount of space and the total amount of free space that is available on the "C:\" drive:

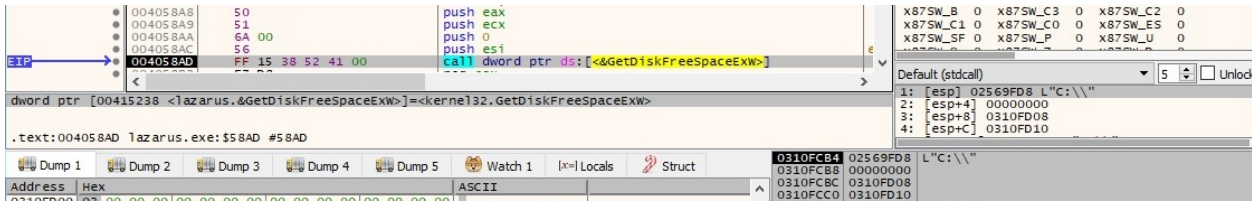


Figure 100

The case number and the drive type, along with the amount of space and the amount of free space, are encrypted using the XOR operator and send to the C2 server:

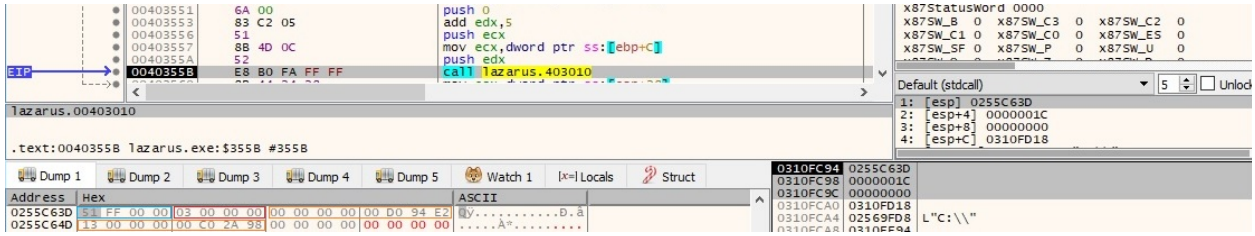


Figure 101

EAX = 31 – extract the current directory name and send it to the C2 server

The binary retrieves the current directory using the GetCurrentDirectoryW routine:

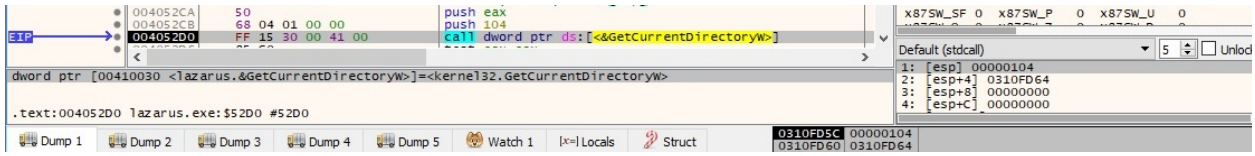


Figure 102

The case number and the directory name are encrypted using XOR operation and transmitted to the C2 server, as shown in the figure below.

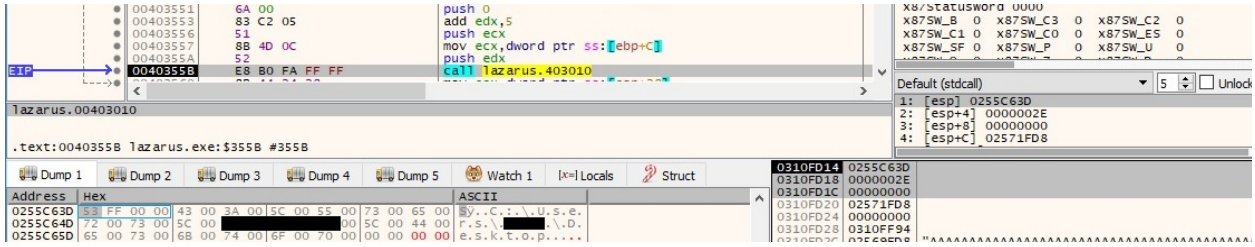


Figure 103

EAX = 32 – set the current directory for the current process to a value provided by the C2 server

The executable calls the FindFirstFileW API with the directory as a parameter:

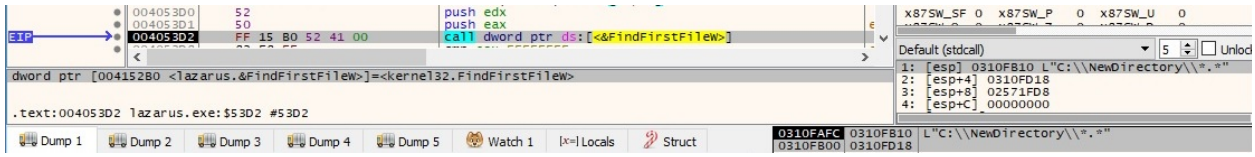


Figure 104

The current directory for the process is changed using the SetCurrentDirectoryW API:

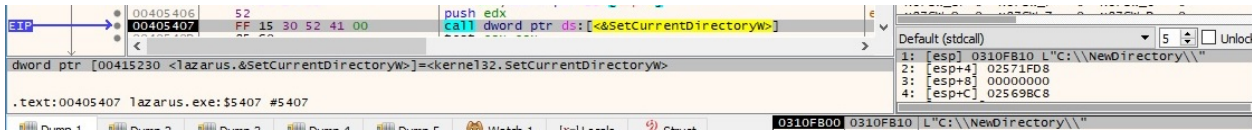


Figure 105

EAX = 33 – delete a registry value used for persistence and all artifacts associated with the malware on the system

GetTempPathA is utilized to retrieve the %TEMP% directory:

Figure 106

The process creates a batch file called CMUPD.bat, as highlighted in figure 107.

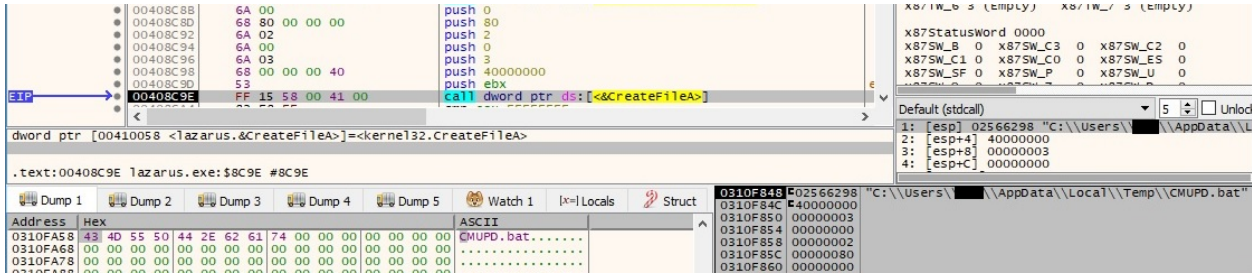


Figure 107

The binary opens the Run registry key that is commonly used for persistence purposes (0x80000002 = **HKEY_LOCAL_MACHINE** and 0xF003F = **KEY_ALL_ACCESS**):

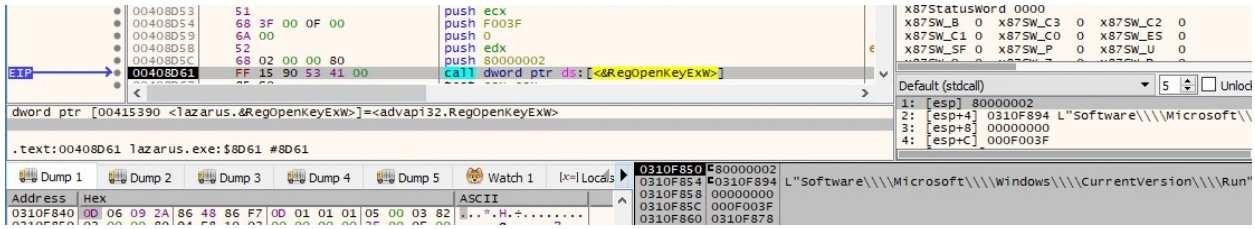


Figure 108

A value with the same name as the executable (which we generically called "lazarus") is deleted by the malware using RegDeleteValueW:

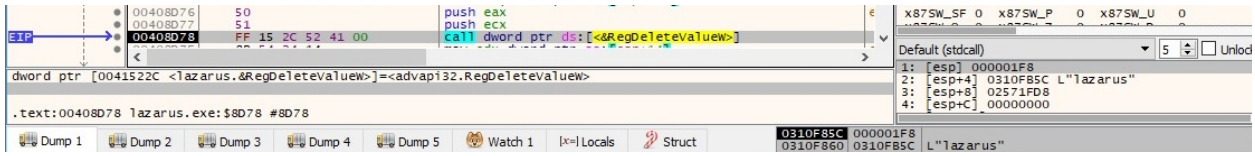


Figure 109

The content of the batch file is displayed below. It is used to delete the malicious file and afterwards the batch file:

```
CMUPD.bat
1  @echo off
2  :Loop
3  del "C:\Users\[redacted]\Desktop\lazarus.exe"
4  if exist "C:\Users\[redacted]\Desktop\lazarus.exe" goto Loop
5  del "C:\Users\[redacted]\AppData\Local\Temp\CMUPD.bat"
```

Figure 110

A new process that runs the batch file is created by the malware, and this concludes our analysis:

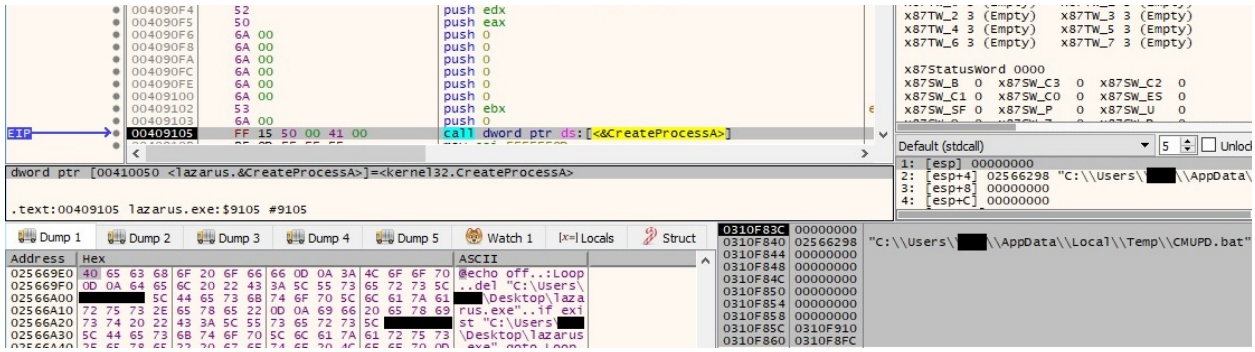


Figure 111

INDICATORS OF COMPROMISE

SHA256: a606716355035d4a1ea0b15f3bee30aad41a2c32df28c2d468eafd18361d60d6

C2 IP addresses:

125.212.132.222

175.100.189.174

APPENDIX

Decryption algorithm for strings (Python)

```
l = ["GvgVvihrlmEcW",
"GvgVlofnvImulinagrlmW",
"GvgUhviNanvW",
"GvgTvnkPagsW",
"GvgTvnkFrovNanvW",
"GvgTrxpClfmg",
"GvgTlpvmImulinagrlm",
"GvgSbhgvnDrivxglibW",
"GvgPilxvhhTrnvh",
"GvgMlwfovHamwovW",
"GvgMlwfovFrovNanvW",
"GvgLlxaoTrnv",
"GvgLltrxaoDirevh",
"GvgLahgEiili",
"GvgFrovTrnv",
"GvgFrovSrzv",
"GvgFrovAggiryfgvhW",
"GvgEcrgClwvTsivaw",
"GvgEcrgClwvPilxvhh",
"GvgDrhpFivvSkaxvEcW",
"GvgDirevTbkvW",
```

```
"GvgClnkfgviNanvW",
"GvgCfiivmgPilxvhh",
"GvgCfiivmgDrivxglibW",
"GvgAwakgvihImul",
"RvtCivagvKvbW"]

for j in range (0, len(l)):
        s = ''
        a = l[j]
        for i in range (0, len(a)):
                b = hex(ord(a[i]))
                b = int(b,16)
                if (b > 0x61) and (b < 0x7a):
                        c = int("0xdb",16) - b
                        s = s + str(bytearray.fromhex(str(hex(c))[2:]).decode())
                else:
                        s = s + a[i]

        print s+"\n"
```

Yara rule for detecting the threat

```
rule Lazarus_FALLCHILL_RAT
{
meta:
        author = "Vlad Pasca - LIFARS LLC"
        Date = "2021-08-25"
        Reference = "https://us-cert.cisa.gov/sites/default/files/publications/MAR-10135536-A_WHITE_S508C.pdf"
strings:
        $s1 = "GvgFrovSrzv" fullword ascii
        $s2 = "LlxpRvhlfixv" fullword ascii
        $s3 = "Pilxvhh32FrihgW" fullword ascii
        $s4 = "WirgvPilxvhhMvnlib" fullword ascii


        $t1 = "@echo off" fullword ascii
        $t2 = "c%sd.e%sc %s > \"%s\" 2>&1" fullword wide
        $t3 = "- -" fullword wide
        $t4 = "REGSVR32.EXE.MUI" fullword wide
condition:
   (uint16(0) == 0x5A4D) and (3 of ($s*) or 3 of ($t*))
}
```