

A detailed analysis of the STOP/Djvu Ransomware – CYBER GEEKS

Published: 2021-10-31 · Archived: 2026-04-05 21:28:24 UTC

Summary

STOP/Djvu ransomware is not a very known ransomware like Conti, REvil or BlackMatter, however ESET ranked it on the 3rd place in the top ransomware families in Q2 2020 (https://www.welivesecurity.com/wp-content/uploads/2020/07/ESET_Threat_Report_Q22020.pdf). This ransomware can run with one of the following parameters: “-Admin”, “-Task”, “-AutoStart”, “-ForNetRes”, and “-Service”. The process doesn’t target specific countries based on their country code, and also decrypts a list of files, file extensions and folders that will be skipped. Two persistence mechanisms are implemented: a Run registry key and a scheduled task created using COM objects. The malware computes the MD5 hash of the MAC address and performs a GET request to the C2 server based on it. The binary also acts as a downloader for 2 malicious files called build2.exe and build3.exe. The victim ID is decrypted using the XOR operator and then written to a file called PersonalID.txt. Both local drives and network shares are targeted by the malware, and the files are encrypted using the Salsa20 algorithm. The Salsa20 matrix used for encrypting files is based on a UUID generated using the UuidCreate API, which is encrypted using an embedded RSA public key (if the C2 server is unreachable) or a public key downloaded from the C2 server. The RSA implementation found in the executable is taken from the OpenSSL project hosted at <https://github.com/openssl/openssl>.

Analyst: [@GeeksCyber](#)

Technical analysis

SHA256: 4380c45fd46d1a63cffe4d37cf33b0710330a766b7700af86020a936cdd09cbe

The following PDB path can be found in the binary:

“C:\xudihiguhe\jegovicatusoca\jijetogez\winucet\xusev\kucor.pdb”. There is a call to GlobalAlloc that allocates several bytes from the heap:



Figure 1

The malware calls the LoadLibraryW function in order to load the “kernel32.dll” file into the address space of the process:

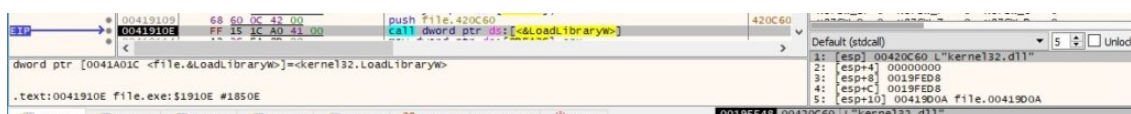


Figure 2

The GetProcAddress API is utilized to retrieve the address of the “VirtualProtect” function:

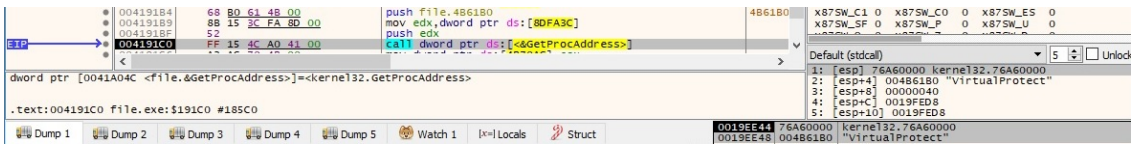


Figure 3

The memory area allocated above is filled in by the malware, and the VirtualProtect routine is used to change its protection to 0x40 = PAGE_EXECUTE_READWRITE:

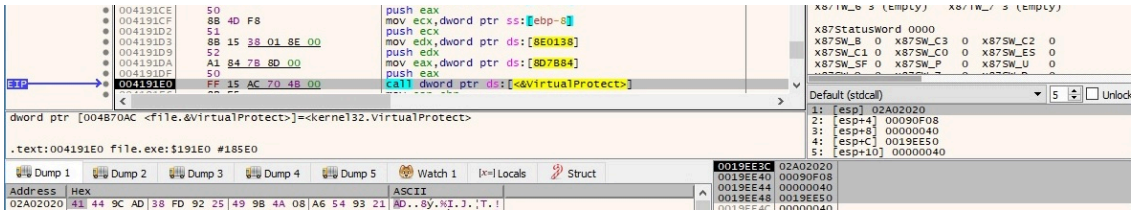


Figure 4

There is also a lot of garbage code in the binary that is never executed, as shown in figure 5:

```

.text:00419708 lea    ecx, [ebp+NumberOfEventsRead]
.text:0041970E push   ecx          ; lpNumberOfEventsRead
.text:0041970F push   0           ; nLength
.text:00419711 lea    edx, [ebp+Buffer]
.text:00419717 push   edx          ; lpBuffer
.text:00419718 push   0           ; hConsoleInput
.text:0041971A call   ds:ReadConsoleInputA
.text:00419720 push   0           ; uSize
.text:00419722 lea    eax, [ebp+var_A78]
.text:00419728 push   eax          ; lpBuffer
.text:00419729 call   ds:GetSystemWow64DirectoryW
.text:0041972F push   0           ; uSize
.text:00419731 lea    ecx, [ebp+var_1278]
.text:00419737 push   ecx          ; lpBuffer
.text:00419738 call   ds:GetSystemWindowsDirectoryA
.text:0041973E lea    edx, [ebp+CPInfoEx]
.text:00419744 push   edx          ; lpCPInfoEx
.text:00419745 push   0           ; dwFlags
.text:00419747 push   0           ; CodePage
.text:00419749 call   ds:GetCPInfoExW
.text:0041974F push   0           ; lpValue
.text:00419751 push   0           ; lpName
.text:00419753 call   ds:SetEnvironmentVariableA
.text:00419759 push   0           ; lpStartupInfo
.text:0041975B call   ds:GetStartupInfoA
.text:00419761 push   offset szFile ; "bojosoboxufevitabanufu lodan"
.text:00419766 push   0           ; uSizeStruct
.text:00419768 lea    eax, [ebp+Struct]
.text:0041976E push   eax          ; lpStruct
.text:0041976F push   offset szKey  ; "dobacu vunubeficapixozeyorolezowodaw ja"...
.text:00419774 push   offset szSection ; "volozowepuyuyigokakifurizigucas sedinum"...
.text:00419779 call   ds:GetPrivateProfileStructA

.text:004197D7 push   0           ; lpSecurityAttributes
.text:004197D9 push   0           ; nDefaultTimeOut
.text:004197DB push   0           ; nInBufferSize
.text:004197DD push   0           ; nOutBufferSize
.text:004197DF push   0           ; nMaxInstances
.text:004197E1 push   0           ; dwPipeMode
.text:004197E3 push   0           ; dwOpenMode
.text:004197E5 push   offset aVocikizifonefa ; "vocikizifonefakifohihezederomaritefibaf"...
.text:004197EA call   ds:CreateNamedPipeA
    
```

Figure 5

The process jumps at the beginning of the new shellcode, as highlighted below:

Jump is taken
dword ptr [file.008D7B84]=02A02020

```

.text:00419846 file.exe:$19846 #18C46
    
```

Address	Hex	ASCII
02A02020	E8 01 00 00	...
02A02030	E8 00 00 00	...
02A02040	C9 C3 55 8B	EAU.i.18SVW.E.A.
02A02050	00 83 65 FC	...e.u.e...X.E.D.E
02A02060	F0 CB 07 00	0E...E...M.O.H..E
02A02070	F9 83 C0 3D	0.A..M..A..h..W..h
02A02080	88 4E 00 00	..N..e...Eohj.4
02A02090	00 68 88 4E	..h..e...EiEu
02A020A0	00 00 00 55	...U..i.SVWQdy50..
02A020B0	00 58 88 40	..X..0..H...A0j..
02A020C0	7D 08 57 50	}.WPè[...At..Èè
02A020D0	E7 88 41 18	..A..P..Xc..A..XXXP.
02A020E0	D8 88 48 1C	}.WPè[...At..Èè
02A020F0	88 32 58 50	..2XP..0j..yu..Vè#..
02A02100	00 85 C0 74	..At..A..A..eàx30
02A02110	66 88 13 C1	E..A..A..E..Y..^].a
02A02120	5D C2 08 00	}.A..U..i.QSR3E3030
02A02130	88 45 08 8A	..E...E..UNà.E..
02A02140	08 84 C9 E0	..Eàj3A..M..;Ut..@Z
02A02150	58 59 88 E5	[Y..à]A...E...Mo.H
02A02160	10 88 45 08	..E..NT..H..E..

Figure 6

The binary retrieves the address of the following functions using GetProcAddress: “GlobalAlloc”, “GetLastError”, “Sleep”, “VirtualAlloc”, “CreateToolhelp32Snapshot”, “Module32First”, “CloseHandle”. CreateToolhelp32Snapshot is utilized to take a snapshot of the current process that includes all its modules (0x8 = TH32CS_SNAPMODULE):

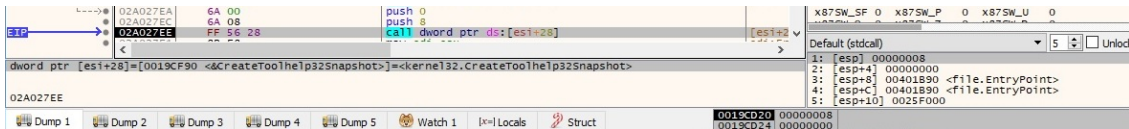


Figure 7

The ransomware extracts information about the first module of the process using the Module32First API:

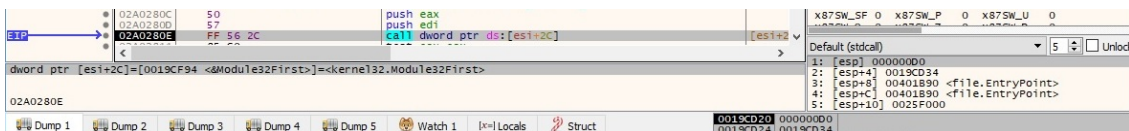


Figure 8

The malicious process allocates and populates a new memory area via a function call to VirtualAlloc (0x1000 = MEM_COMMIT and 0x40 = PAGE_EXECUTE_READWRITE):



Figure 9

The process jumps at the beginning of the new shellcode, as highlighted below:

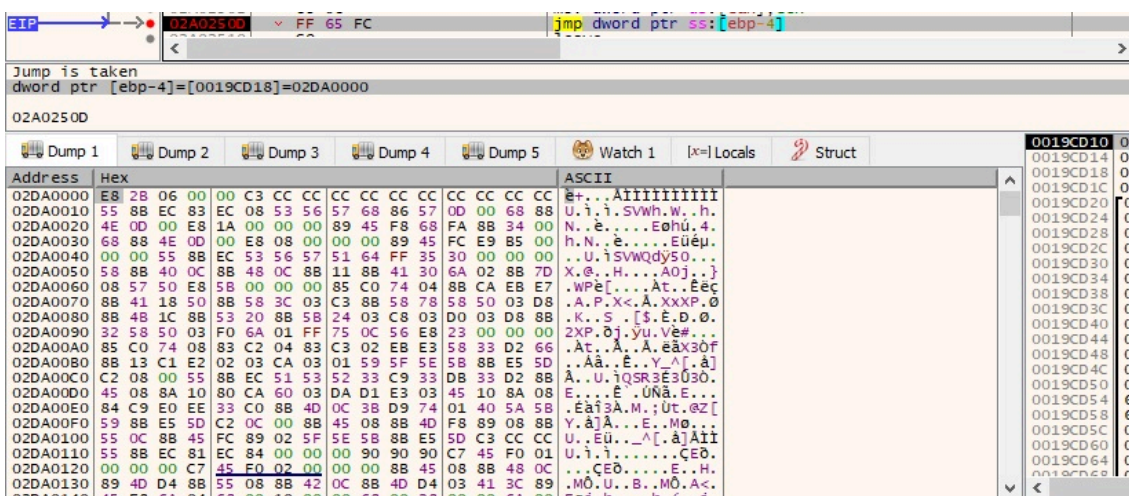


Figure 10

The malware calls the LoadLibraryA API to load the following DLLs into memory: user32.dll, kernel32.dll and ntdll.dll. It also retrieves the address of the following functions: “MessageBoxA”, “GetMessageExtraInfo”, “WinExec”, “CreateFileA”, “WriteFile”, “CloseHandle”, “CreateProcessA”, “GetThreadContext”, “VirtualAlloc”,

“VirtualAllocEx”, “VirtualFree”, “ReadProcessMemory”, “WriteProcessMemory”, “SetThreadContext”, “ResumeThread”, “WaitForSingleObject”, “GetModuleFileNameA”, “GetCommandLineA”, “NtUnmapViewOfSection”, “NtWriteVirtualMemory”, “RegisterClassExA”, “CreateWindowExA”, “PostMessageA”, “GetMessageA”, “DefWindowProcA”, “GetFileAttributesA”, “GetStartupInfoA”, “VirtualProtectEx”, “ExitProcess”.

From our perspective, the malware developers have implemented some actions that don’t influence the main execution flow as an anti-analysis mechanism. GetFileAttributesA is used to retrieve file system attributes for a non-existent file:

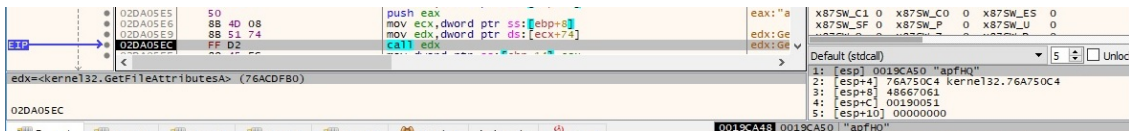


Figure 11

The file registers a window class called “saodkfnosa9uin” using the RegisterClassExA routine:

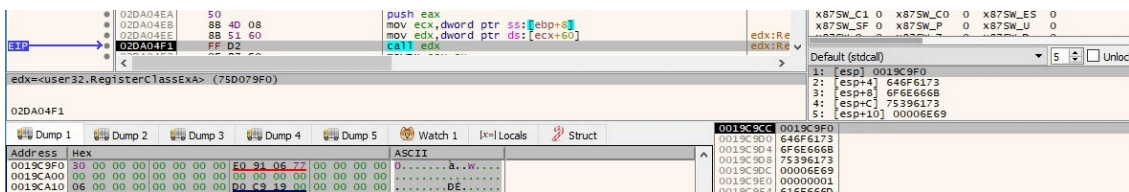


Figure 12

The CreateWindowExA function is utilized to create a new window (0x200 = WS_EX_CLIENTEDGE, 0xCF0000 = WS_OVERLAPPEDWINDOW, 0x80000000 = CW_USEDEFAULT):

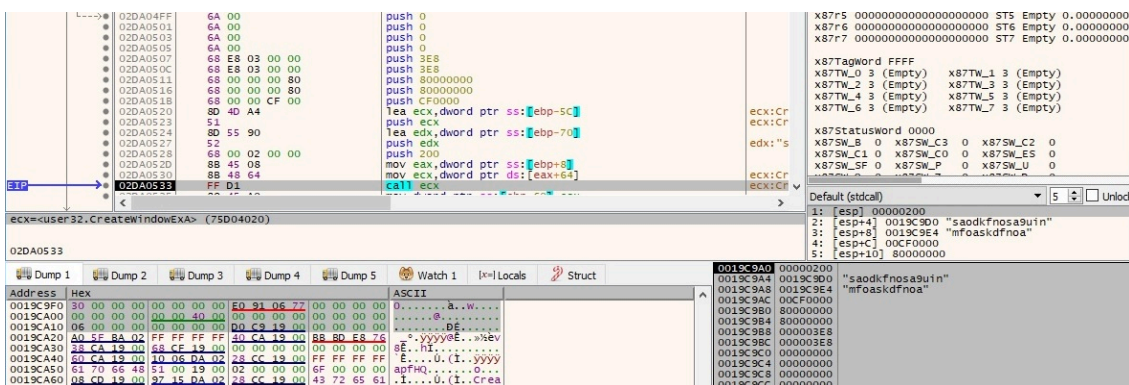


Figure 13

The process allocates a new memory area via a function call to VirtualAlloc (0x1000 = MEM_COMMIT and 0x4 = PAGE_READWRITE):



Figure 14

The ransomware extracts the content of the STARTUPINFO structure:

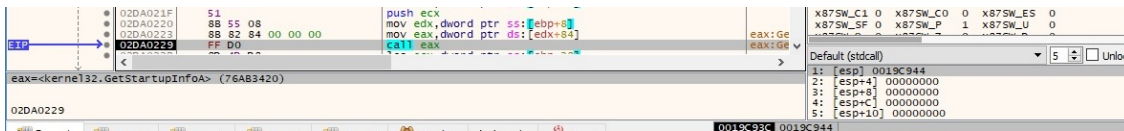


Figure 15

The malware creates a copy of itself in a suspended state via a call to CreateProcessA (0x08000004 = CREATE_NO_WINDOW | CREATE_SUSPENDED):

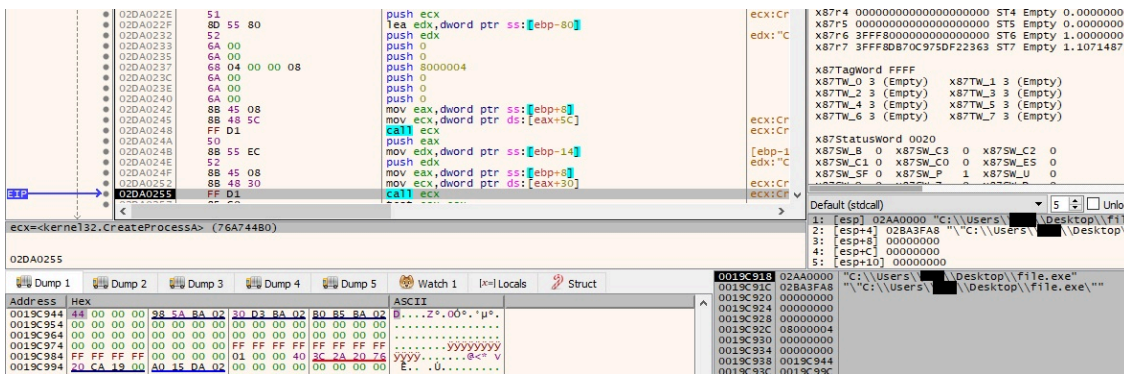


Figure 16

GetThreadContext is used to retrieve the context of a specific thread:

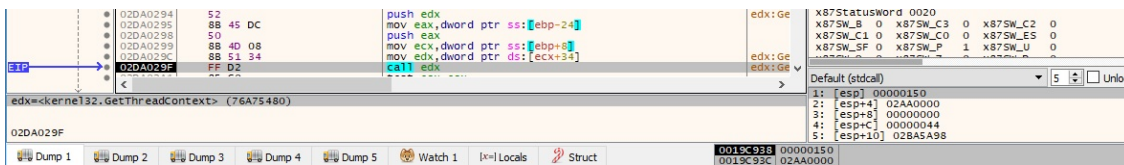


Figure 17

The malicious binary unmaps a view of a section from the address of the newly created process using ZwUnmapViewOfSection:



Figure 18

The VirtualAllocEx routine is utilized to allocate new space in the newly created process (0x3000 = MEM_COMMIT | MEM_RESERVE and 0x40 = PAGE_EXECUTE_READWRITE):

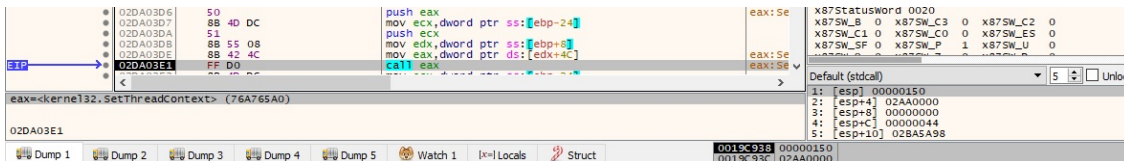


Figure 21

The binary resumes the main thread of the suspended process using ResumeThread:

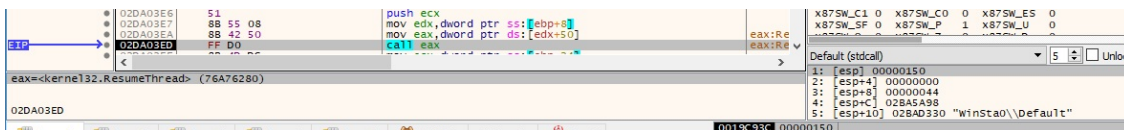


Figure 22

We’ve extracted the executable from memory, and we continue to analyze this file. The following PDB path has been found: “e:\doc\my work (c++)_git\encryption\release\encrypt_win_api.pdb”. The binary initializes the use of the WinINet functions by calling the InternetOpenW API (the user agent being “Microsoft Internet Explorer”):

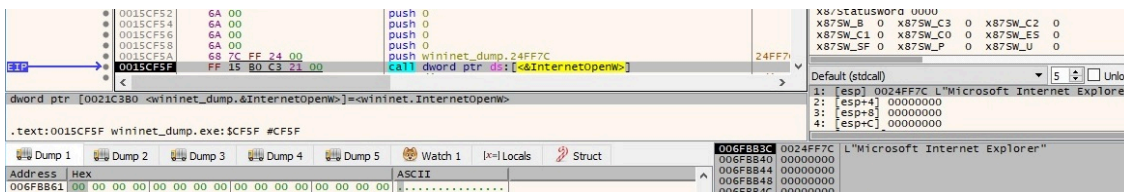


Figure 23

The malware performs a GET request to <https://api.2ip.ua/geo.json>, which reveals details about the location of the IP address:

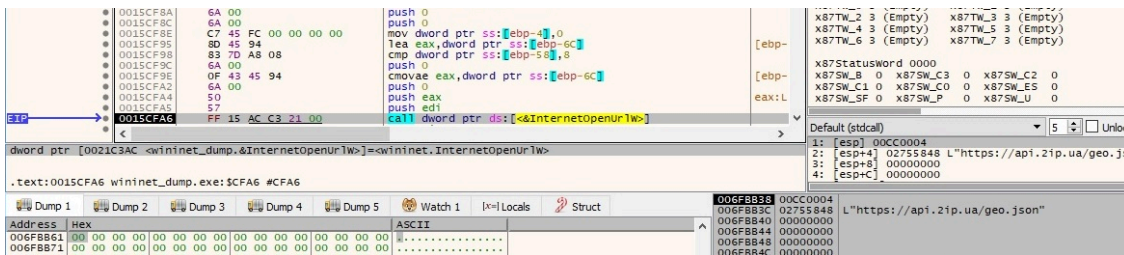


Figure 24

InternetReadFile is used to read the response from the server, and an example of a JSON form is displayed below:

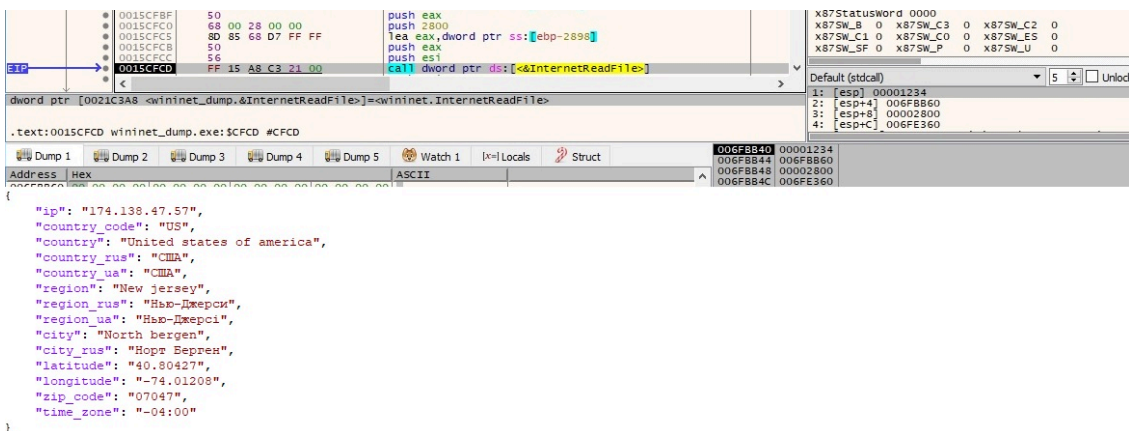


Figure 25

The “country_code” element is compared with “RU” (Russian language), “BY” (Belarusian language), “UA” (Ukrainian language), “AZ” (Azerbaijani language), “AM” (Armenian language), “TJ” (Tajik language), “KZ” (Kazakh language), “KG” (Kyrgyz language), “UZ” (Uzbek language) and “SY” (Syriac language):

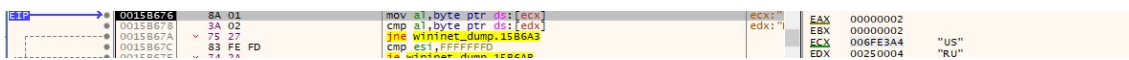


Figure 26

The systems that have one of the languages enumerated above will not be encrypted. The priority for the current process is set to high by calling the SetPriorityClass routine (0x80 = **HIGH_PRIORITY_CLASS**):

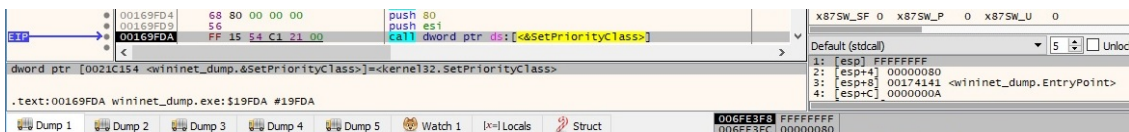


Figure 27

The executable retrieves the command-line string for the process and then returns an array of pointers to the command-line arguments:

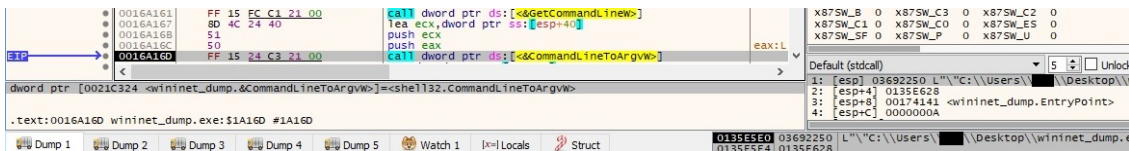


Figure 28

It’s important to mention that the malware can run with one of the following parameters: “-Admin”, “-Task”, “-AutoStart”, “-ForNetRes”, and “-Service”. We’ll describe the execution flows with different parameters later on.

All process IDs that correspond to the processes on the system are retrieved by calling the EnumProcesses API:

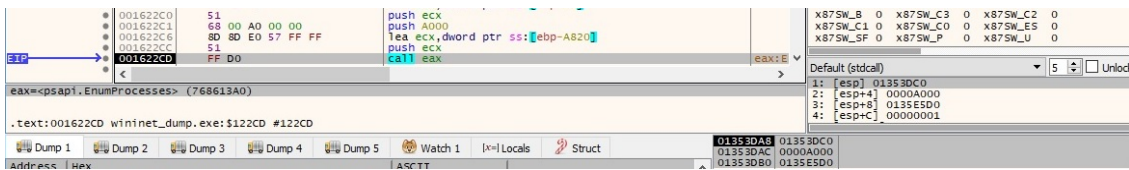


Figure 29

Each process object is opened by the ransomware using OpenProcess (0x410 = **PROCESS_QUERY_INFORMATION | PROCESS_VM_READ**):

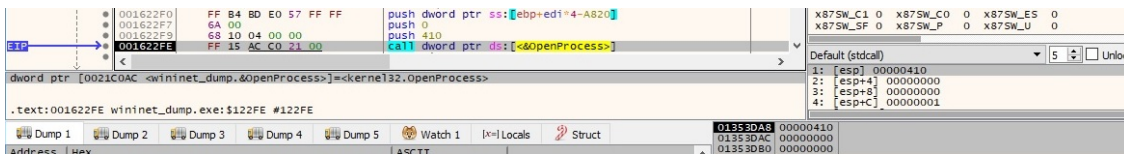


Figure 30

The malware extracts a handle for each module from a process that was successfully opened:

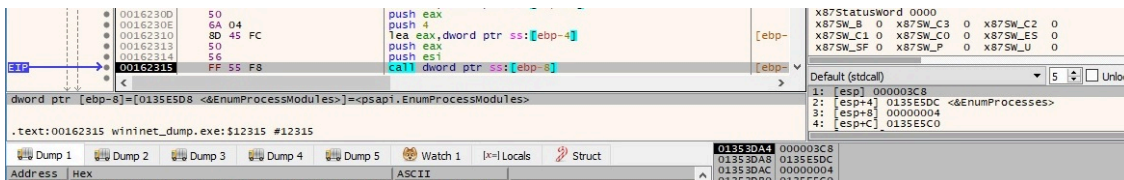


Figure 31

The GetModuleBaseNameW function is used to retrieve the base name of a module that is compared with the name of the executable (in our case, “wininet_dump.exe”):

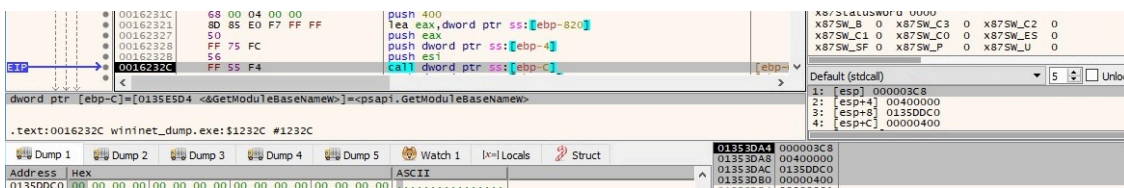


Figure 32

The binary performs a lot of XOR operations (key = 0x80) in order to decrypt relevant strings. The next figure contains a buffer with the C2 server securebiz[.]org:

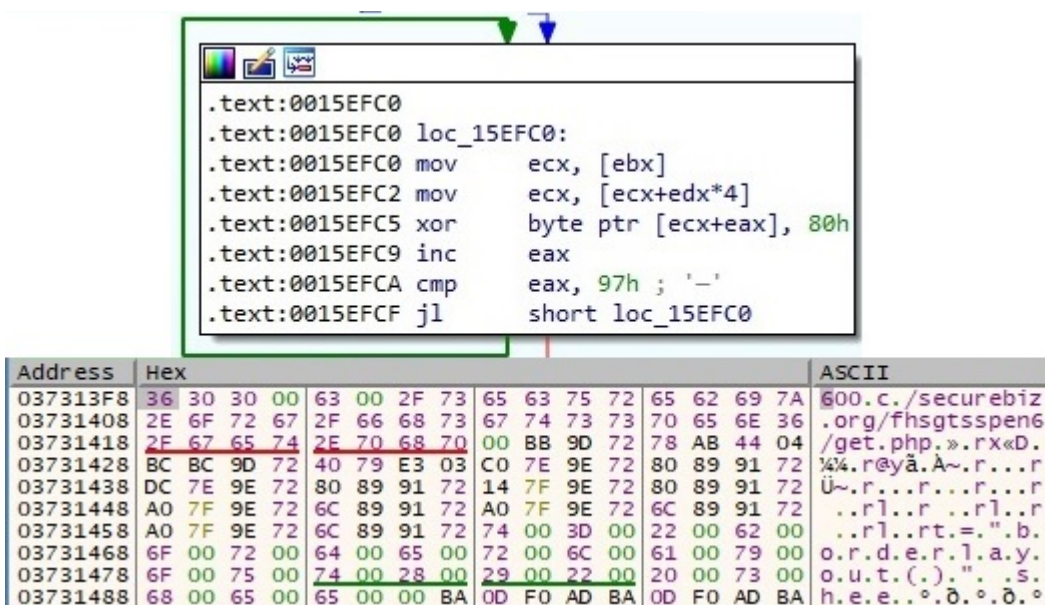


Figure 33

The ransomware opens the Run registry key using RegOpenKeyExW (0x80000001 = HKEY_CURRENT_USER and 0xF003F = KEY_ALL_ACCESS):

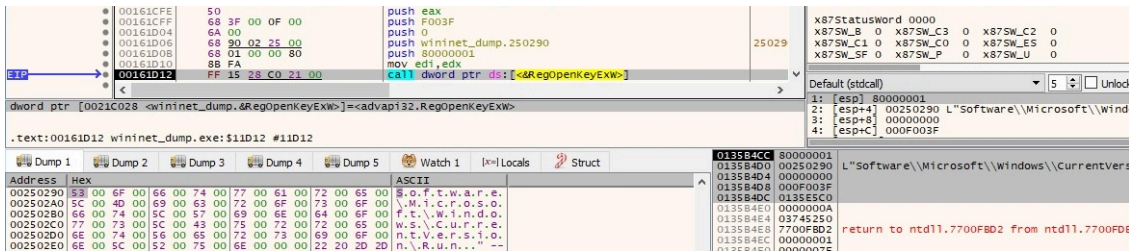


Figure 34

The process is looking for a value called “SysHelper”, which doesn’t exist at this time:

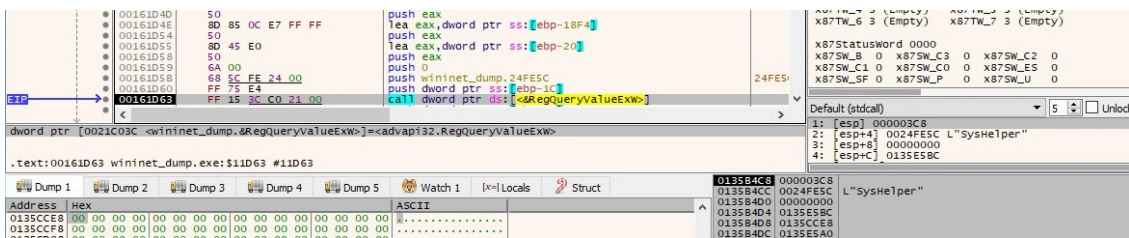


Figure 35

The UuidCreate function is used to generate a new UUID (16 random bytes):

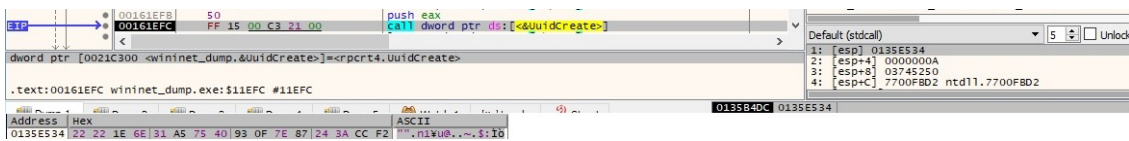


Figure 36

The process converts the UUID to a string using the UuidToStringW API:

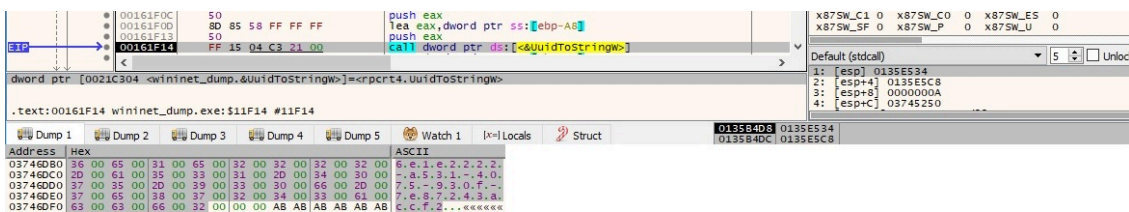


Figure 37

A new directory based on the UUID is created by the malware:

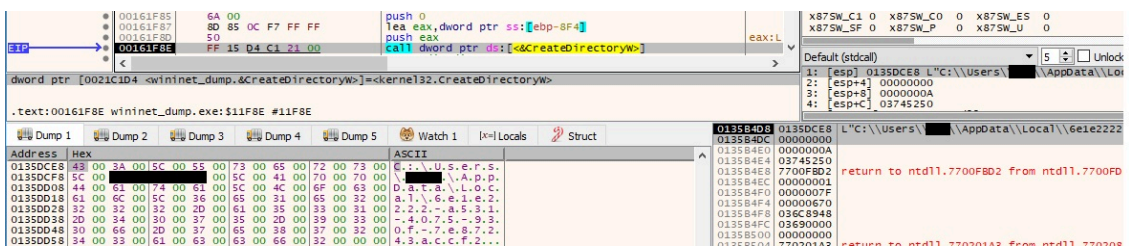


Figure 38

The CopyFileW routine is utilized to copy the executable to a new file in the above directory:

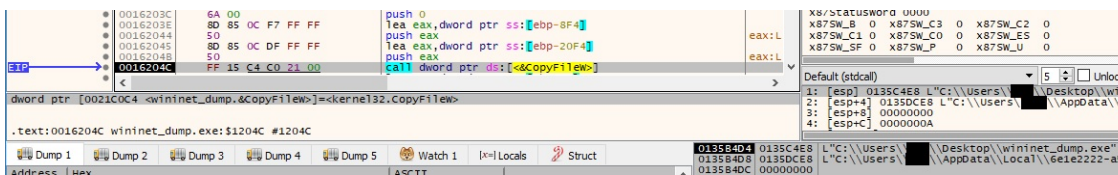


Figure 39

The ransomware establishes persistence on the host by creating an entry called “SysHelper” under the Run registry key, which will run the executable with the “-AutoStart” parameter whenever the user logs on:

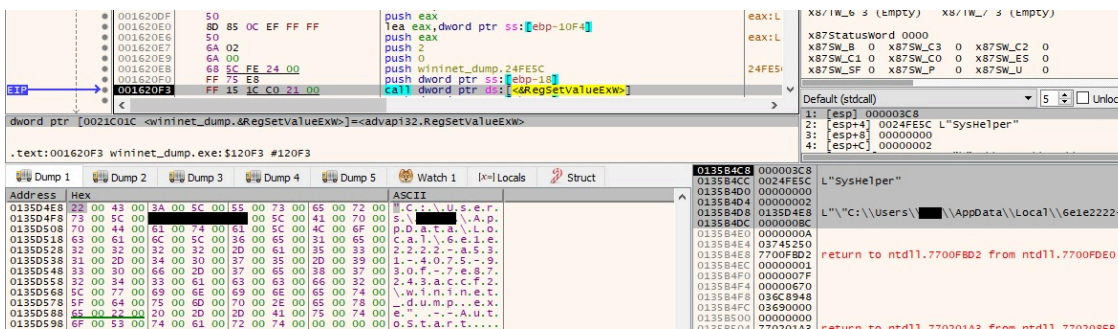


Figure 40

The binary denies “Everyone” to delete the folder created above using the icacls command, as highlighted in figure 41:

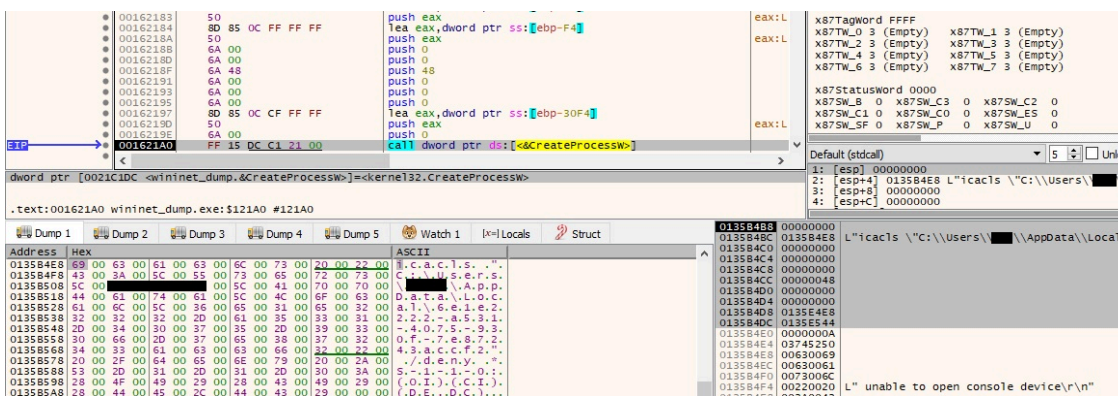


Figure 41

A second persistence mechanism consists of creating a scheduled task (using COM objects) that will run the ransomware every 5 minutes.

The malicious file initializes the COM library on the current thread using the CoInitialize function:



Figure 42

We have observed that the implementation is similar to the one presented at <https://docs.microsoft.com/en-us/windows/win32/taskschd/time-trigger-example-c>, however we'll dig deeper and explain how the assembly code looks like.

The CoInitializeSecurity routine is used to register and set the default security values for the process (0x6 = **RPC_C_AUTHN_LEVEL_PKT_PRIVACY** and 0x3 = **RPC_C_IMP_LEVEL_IMPERSONATE**):

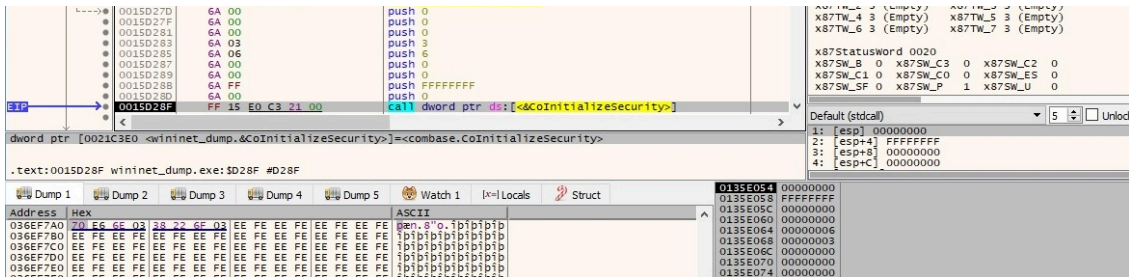


Figure 43

The process creates an object with the CLSID {0F87369F-A4E5-4CFC-BD3E-73E6154572DD}, which implements the Schedule.Service class for operating the Windows Task Scheduler Service:

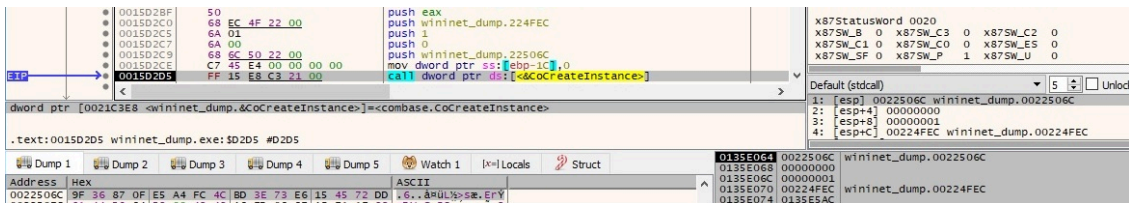


Figure 44

You can notice if you follow the C++ implementation mentioned above that in a case of a function call such as `p -> f(a,b)`, the assembly representation contains 3 parameters pushed on the stack (because the pointer `p` is pushed as well). An example of such a call is represented by the `ITaskService::GetFolder` method, which gets a folder of registered tasks:

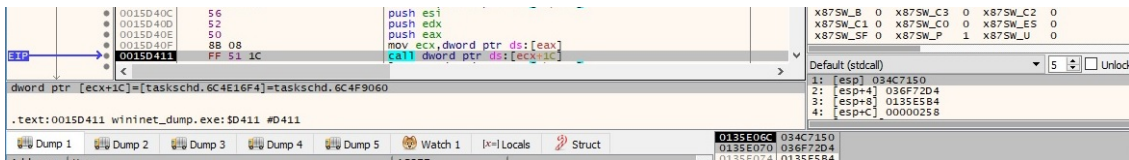


Figure 45

A task called "Time Trigger Task" is deleted using the `ITaskFolder::DeleteTask` method:

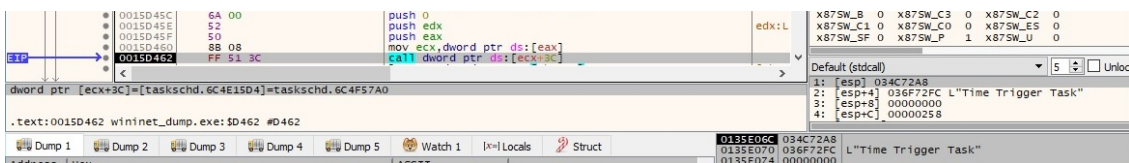


Figure 46

The `ITaskService::NewTask` function is utilized to create an empty task definition object:

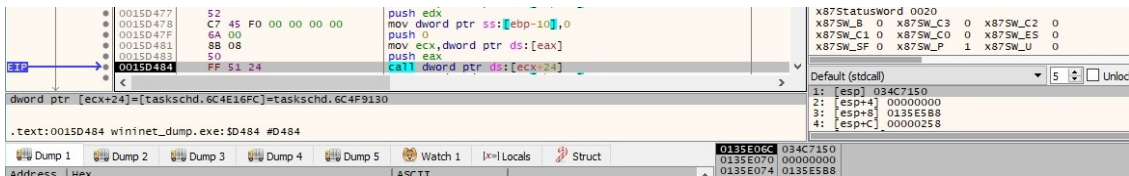


Figure 47

An example of a safe release when the pointer is no longer used is shown in figure 48:

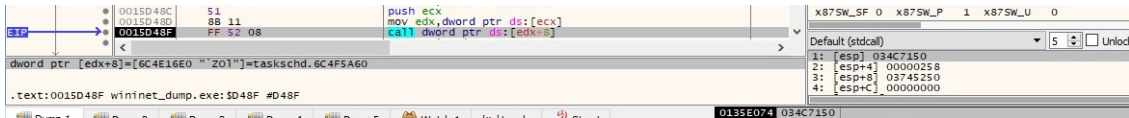


Figure 48

The binary retrieves the registration information of the task (the description, the author, and the date the task is registered) by calling the `ITaskDefinition::get_RegistrationInfo` method:

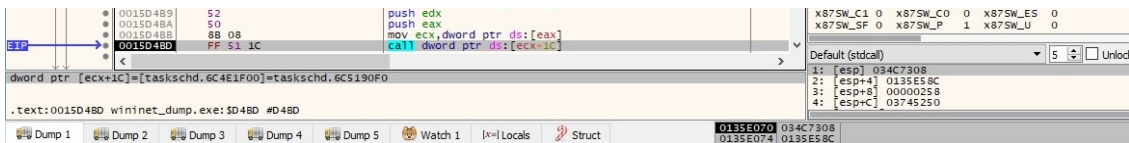


Figure 49

`IRegistrationInfo::put_Author` is used to set the author of the task to “Author Name”:

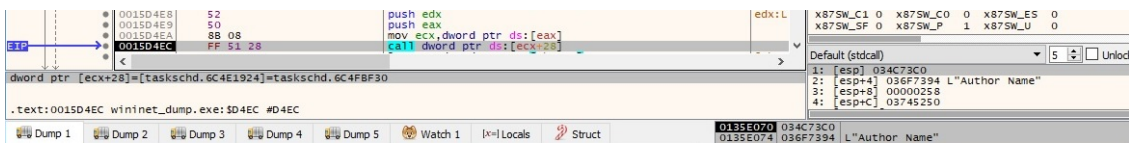


Figure 50

The ransomware retrieves the principal for the task (which provides the security credentials) by calling the `ITaskDefinition::get_Principal` function:

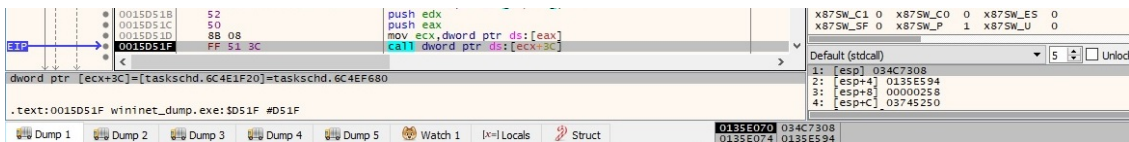


Figure 51

The security logon type is set to 0x3 (**TASK_LOGON_INTERACTIVE_TOKEN**), which means that the task will be run only in an existing interactive session:

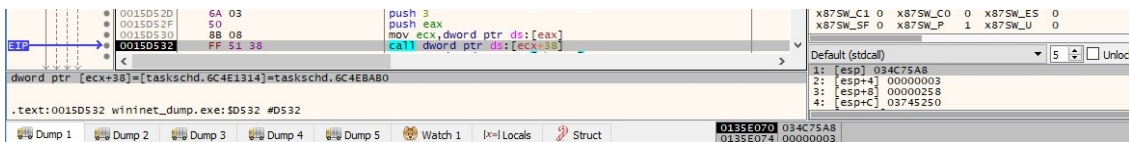


Figure 52

ITaskDefinition::get_Settings is utilized to retrieve the settings that describe how the Task Scheduler performs the task:

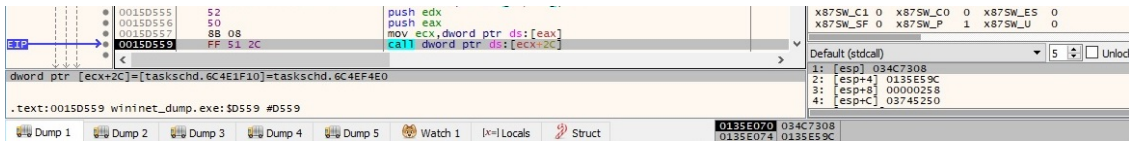


Figure 53

The file sets a Boolean value to 0xFFFFFFFF (VARIANT_TRUE) that indicates the Task Scheduler can start the task at any time after its scheduled time has elapsed using the ITaskSettings::put_StartWhenAvailable method:

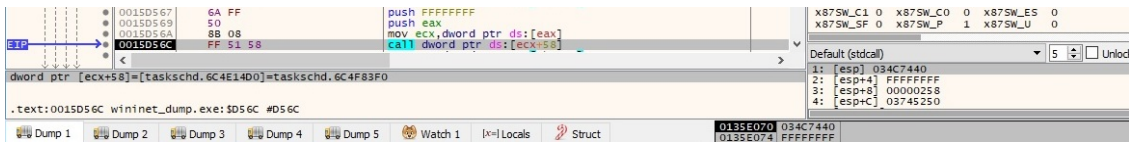


Figure 54

The amount of time the Task Scheduler will wait for an idle condition to occur is set to 5 minutes via a function call to IdleSettings::put_WaitTimeout:

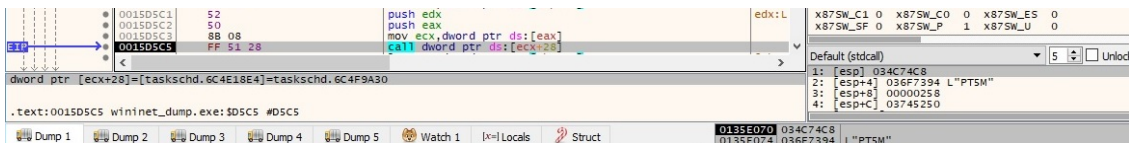


Figure 55

ITaskDefinition::get_Triggers is used to get a collection of triggers used to start the task:

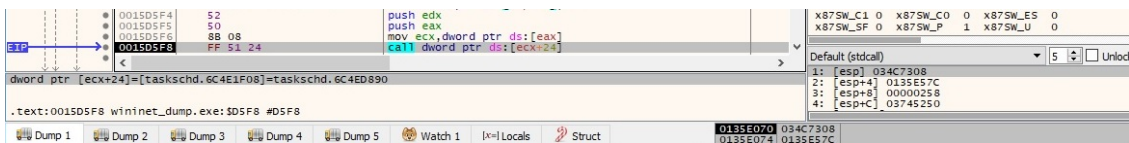


Figure 56

The executable creates a new trigger for the task using the ITriggerCollection::Create method (0x1 = TASK_TRIGGER_TIME):

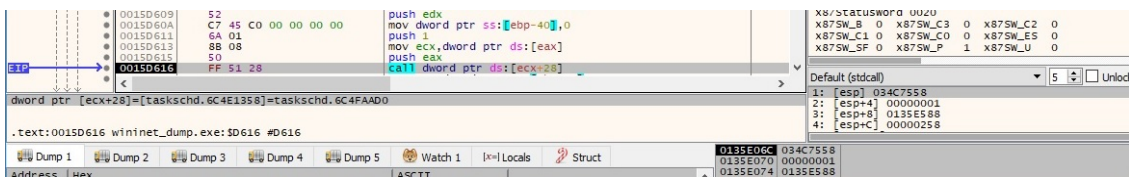


Figure 57

There is a QueryInterface call with a parameter set as CLSID {B45747E0-EBA7-4276-9F29-85C5BB300006} – IID_ITimeTrigger:

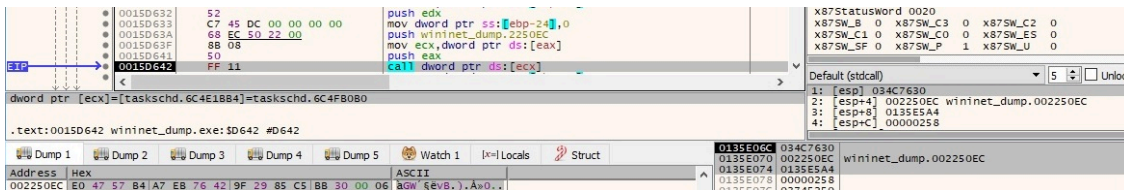


Figure 58

The identifier for the trigger is set to “Trigger1” using the ITrigger::put_Id function:

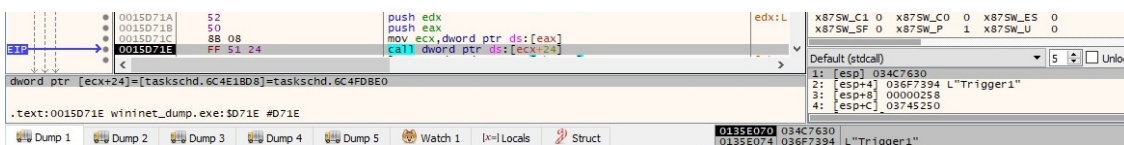


Figure 59

The ransomware sets the date and time when the trigger is deactivated by calling the ITrigger::put_EndBoundary method:

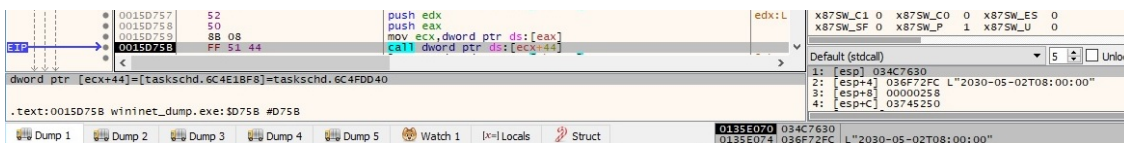


Figure 60

The system time is extracted via a call to the _time64 function:



Figure 61

The malware formats the system time into a human-readable form using strftime:

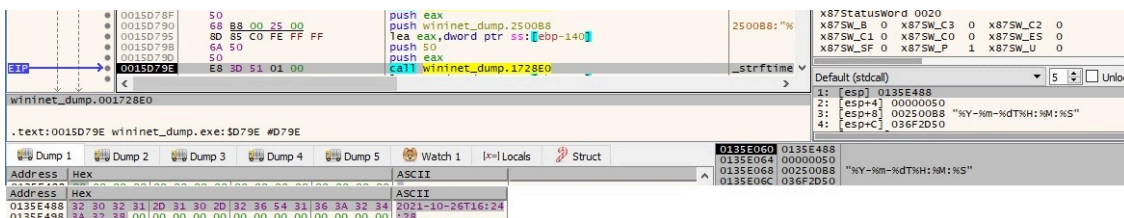


Figure 62

The malicious binary sets the date and time when the trigger is activated by calling the ITrigger::put_StartBoundary method:

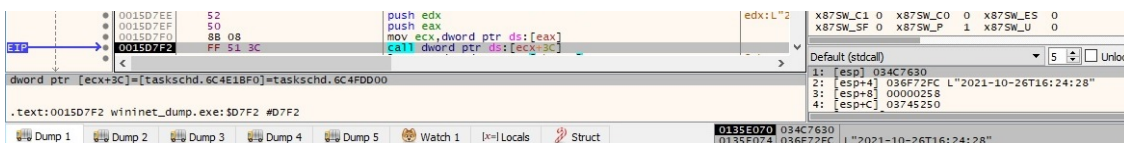


Figure 63

IActionCollection::Create is utilized to create and add a new action to the collection (0x0 = TASK_ACTION_EXEC):

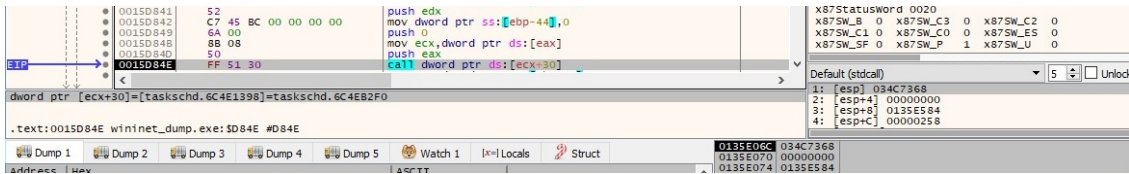


Figure 64

There is a QueryInterface call with a parameter set as CLSID {4c3d624d-fd6b-49a3-b9b7-09cb3cd3f047} – IID_IExecAction:

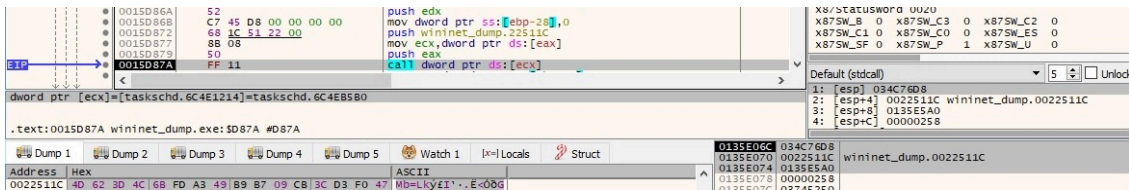


Figure 65

The path of the executable is set to the copied file using the IExecAction::put_Path method:

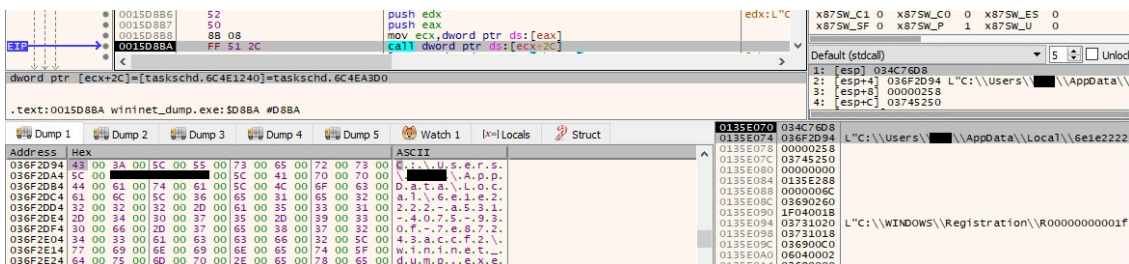


Figure 66

The “-Task” argument is added by calling the IExecAction::put_Arguments function:

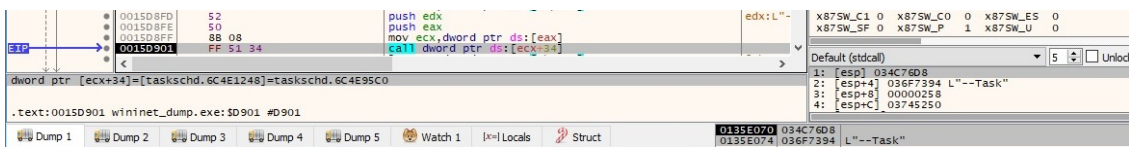


Figure 67

Finally, the malware uses the ITaskFolder::RegisterTaskDefinition method to create the task called “Time Trigger Task” (0x6 = TASK_CREATE_OR_UPDATE):

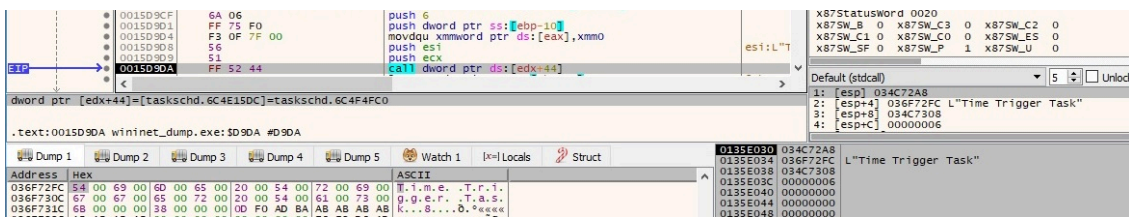


Figure 68

Here is the newly created scheduled task in Windows Task Scheduler:



Figure 69

The ransomware launches itself with the following parameters “-Admin IsNotAutoStart IsNotTask” (IsNotAutoStart = malware didn’t run based on the Run registry key, IsNotTask = malware didn’t run based on the scheduled task):

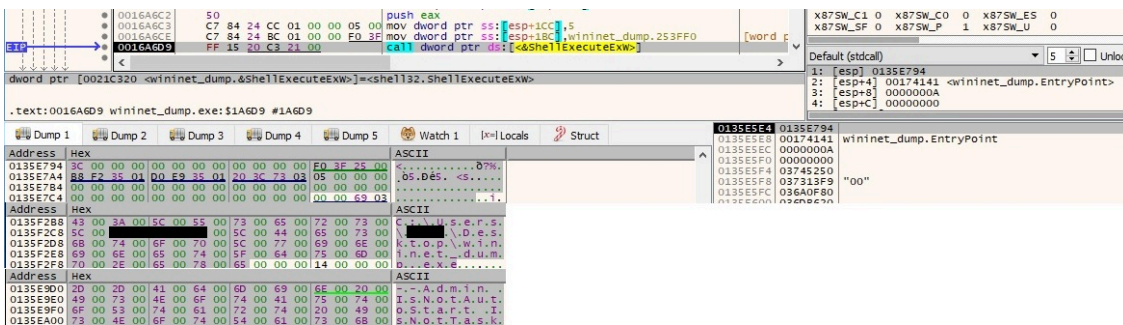


Figure 70

“-Task“ parameter

We’ll only highlight different actions that are performed by the ransomware running with this parameter without mentioning the same actions as in the case of running with no parameters.

GetAdaptersInfo is utilized to retrieve adapter information (including the MAC address) for the localhost:

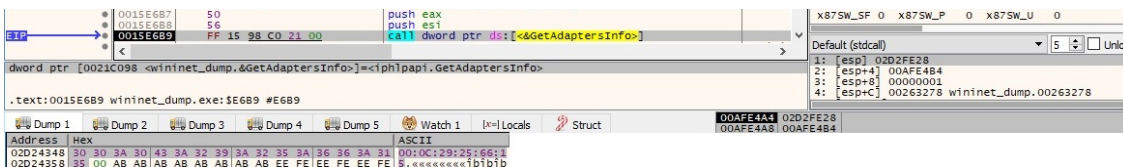


Figure 71

The malware calls the CryptAcquireContextW API in order to obtain a handle to a particular key container within a cryptographic service provider (0x1 = **PROV_RSA_FULL** and 0xF0000000 = **CRYPT_VERIFYCONTEXT**):

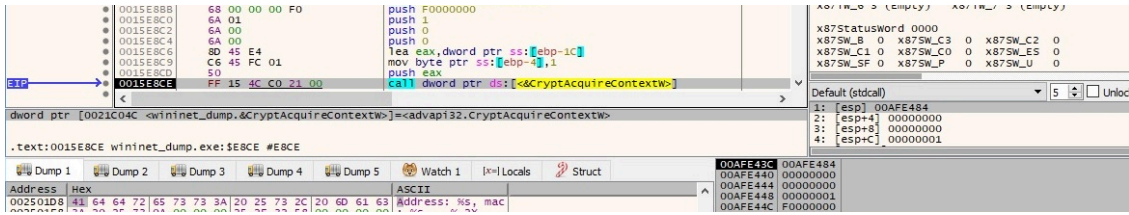


Figure 72

The binary creates a handle to a CSP hash object using the CryptCreateHash API (0x8003 = CALG_MD5):

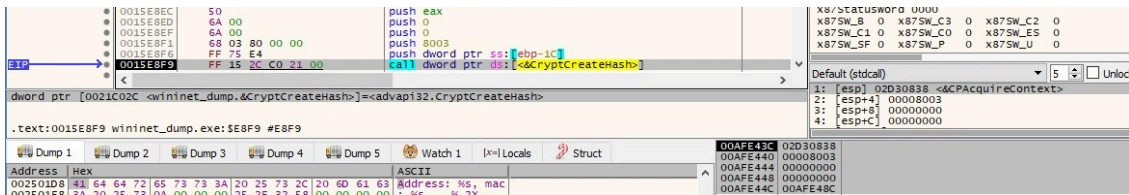


Figure 73

The ransomware hashes a buffer that contains the MAC address extracted above via a function call to CryptHashData:

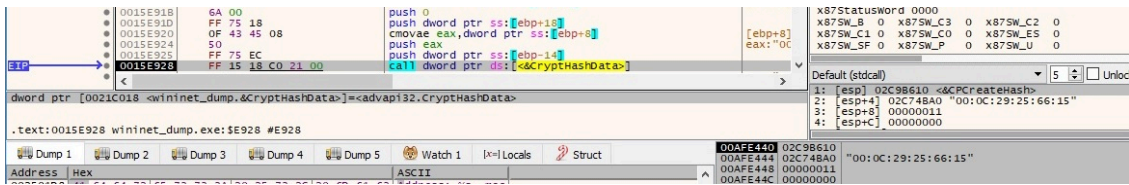


Figure 74

The MD5 hash value is extracted by calling the CryptGetHashParam routine (0x2 = HP_HASHVAL):



Figure 75

A new thread is created by calling the CreateThread API:

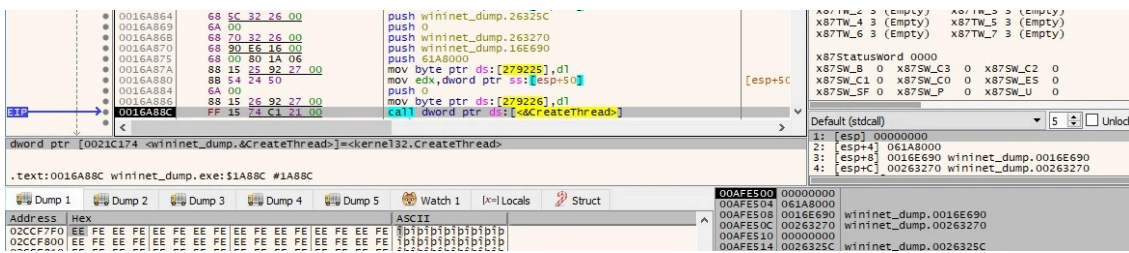


Figure 76

Thread activity – sub_16E690

The RegOpenKeyExW function is used to open the “Software\Microsoft\Windows\CurrentVersion” registry key (0x80000001 = HKEY_CURRENT_USER and 0xF003F = KEY_ALL_ACCESS):

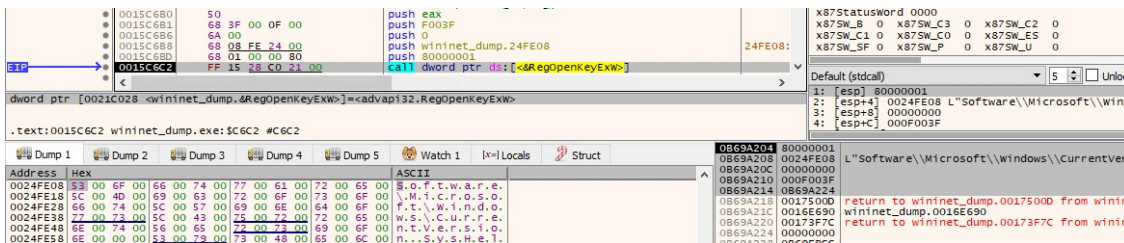


Figure 77

The process is looking for a value named “SysHelper”, which doesn’t exist at this time (this is different from the one in figure 34):

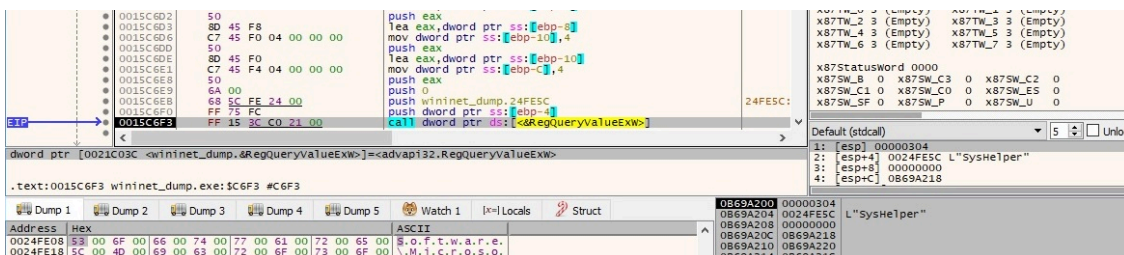


Figure 78

The entry from above is created, and its value is set to 1 using the RegSetValueExW API:

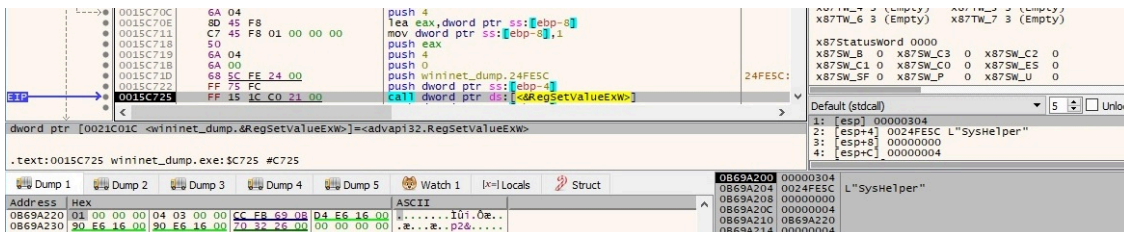


Figure 79

The executable tries to locate a file called “bowsakkdstx.txt” in the “C:\Users\

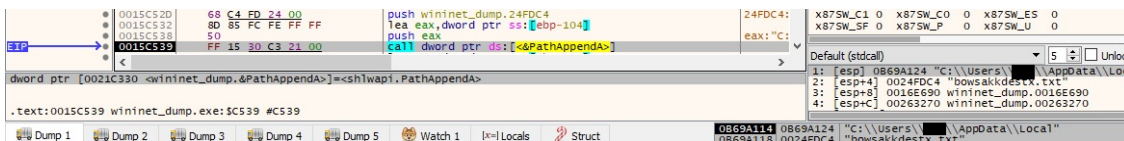


Figure 80

There is a function call to InternetOpenW similar to the one presented in figure 23 (with the same user agent). The binary performs a GET request to the C2 server securebiz[.]org with the parameter pid = MD5(MAC address):

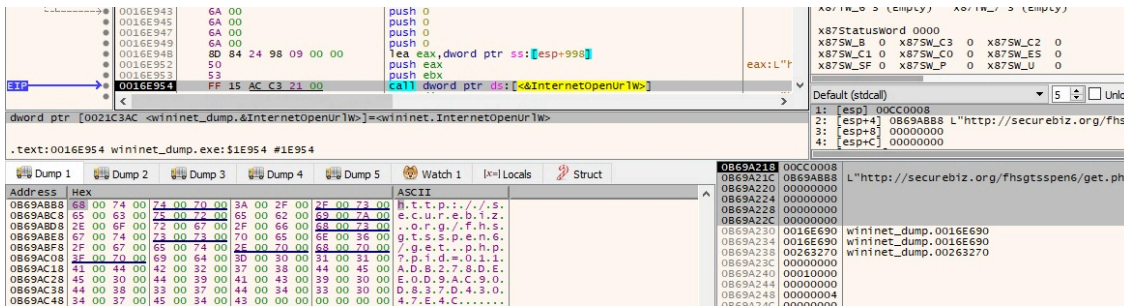


Figure 81

The response from the server is read using the InternetReadFile function:

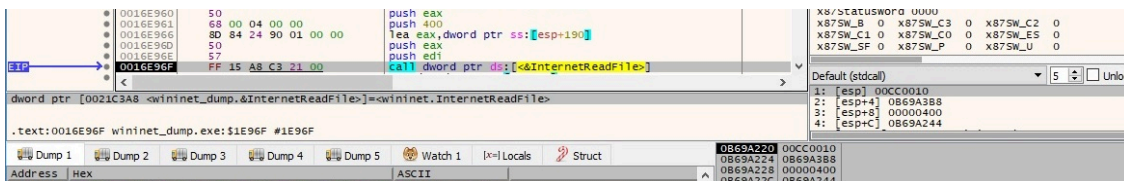


Figure 82

The binary creates the file called “bowsakkdestx.txt” using fopen:

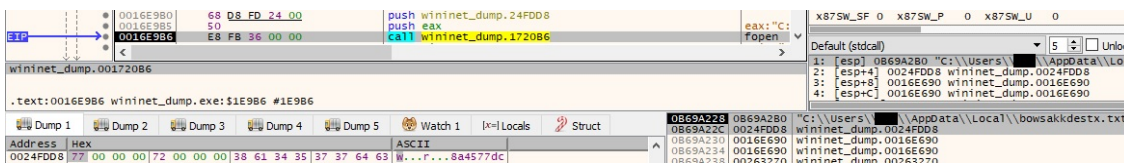


Figure 83

The file is populated using a function call to fwrite (the C2 server was down during our analysis, so we emulated the network communications using FakeNet):

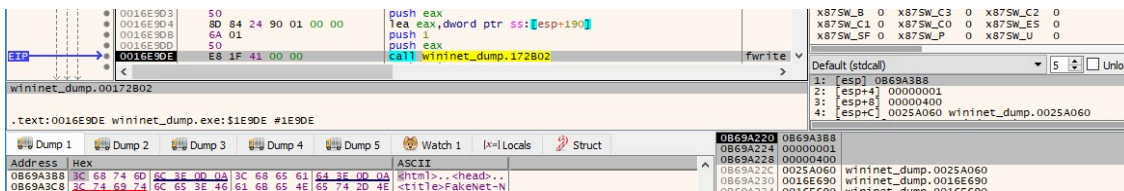


Figure 84

An example of a real response can be seen at <https://app.any.run/tasks/900f626a-2bf6-48b2-85f9-2328f2b2d0d2/> and contains 2 elements: “public_key” and “id”. The malware wants to extract the “public_key” value from the response:

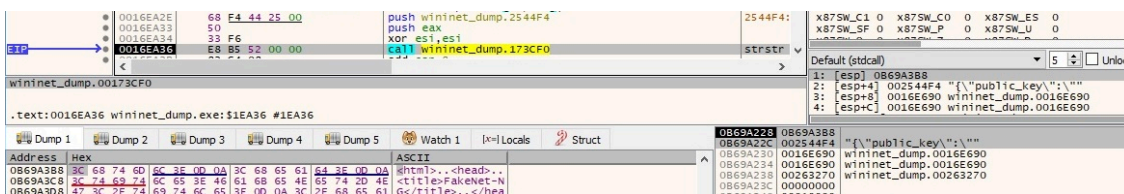


Figure 85

Even though the C2 server was down, the binary comes with a hard-coded RSA public key. The file from above is deleted in any case:

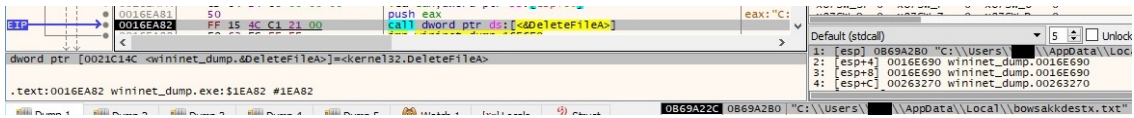


Figure 86

Using multiple XOR operations with 0x80, the ransomware decrypts the RSA public key in PKCS1 format, a victim ID, and a URL that leads to another malicious file at [http://securebiz\[.\]org/files/1/build3.exe](http://securebiz[.]org/files/1/build3.exe):

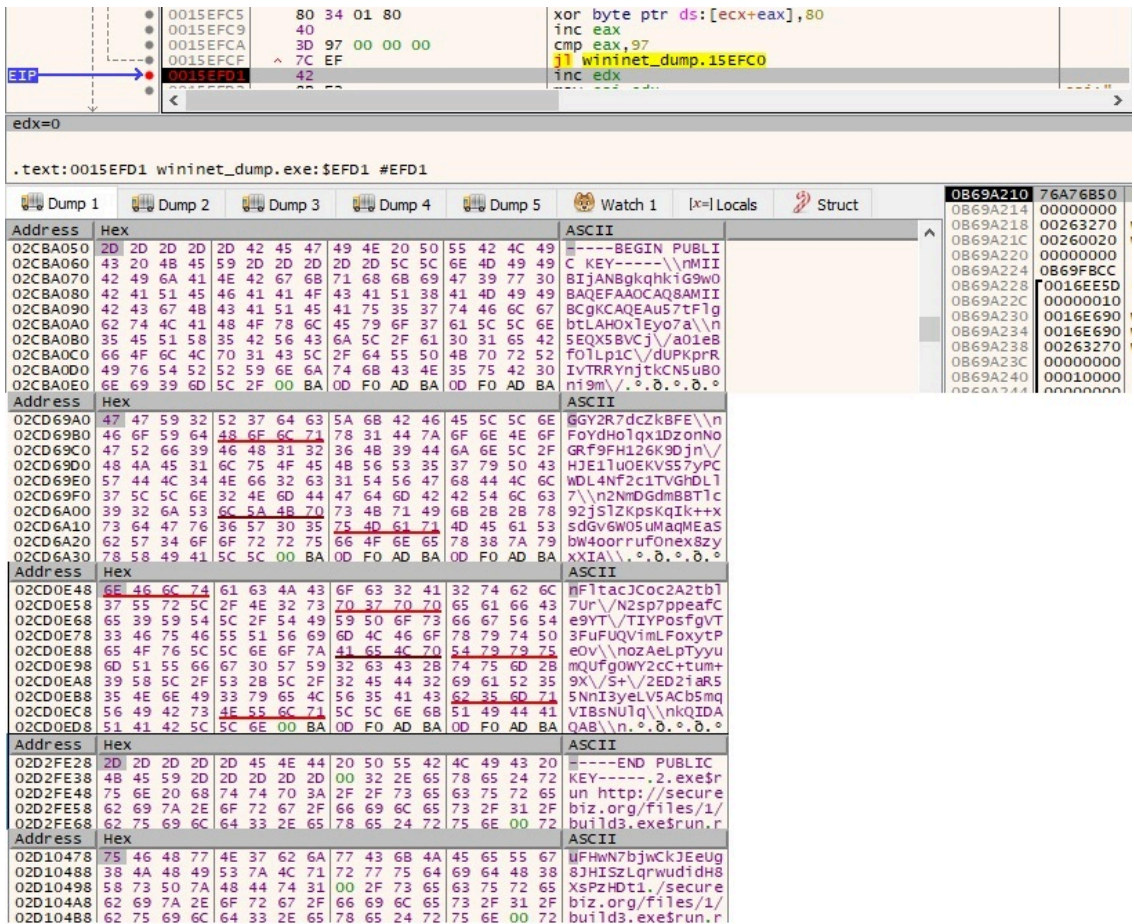


Figure 87

We continue to analyze the main thread. A mutex called “{1D6FC66E-D1F3-422C-8A53-C0BBCF3D900D}” is created via a function call to CreateMutexA:

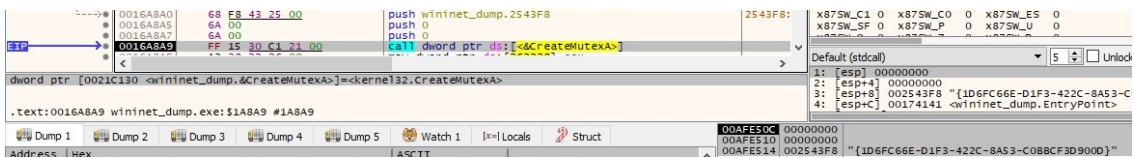


Figure 88

The malware decrypts the ransom note using the XOR operator:

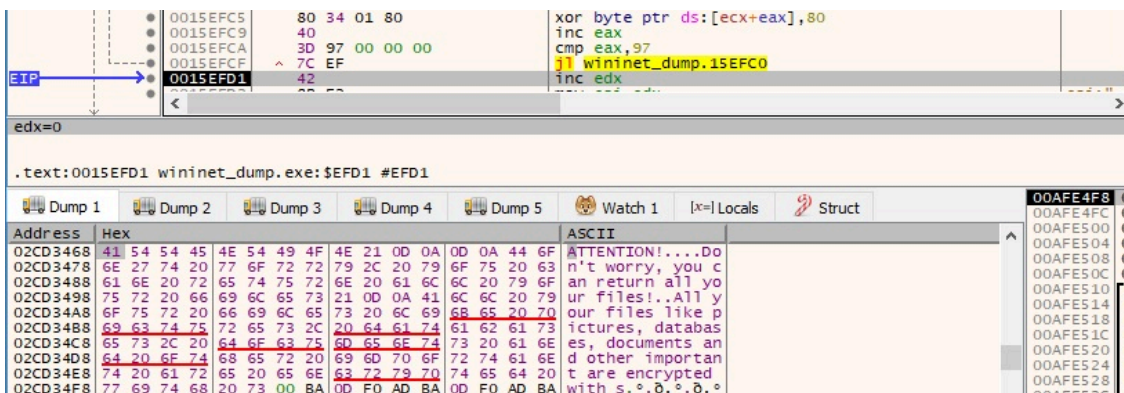


Figure 89

The following information is also decrypted (a list of files to be skipped, a list of extensions to be skipped, and a list of directories to be skipped):

- ntuser.dat, ntuser.dat.LOG1, ntuser.dat.LOG2, ntuser.pol
- .sys, .ini, .DLL, .dll, .blf, .bat, .lnk, .regtrans-ms
- C:\SystemID\, C:\Users\Default User\, C:\Users\Public\, C:\Users\All Users\, C:\Users\Default\, C:\Documents and Settings\, C:\ProgramData\, C:\Recovery\, C:\System Volume Information\, C:\Users\%username%\AppData\Roaming\, C:\Users\%username%\AppData\Local\, C:\Windows\, C:\PerfLogs\, C:\ProgramData\Microsoft\, C:\ProgramData\Package Cache\, C:\Users\Public\, C:\\$Recycle.Bin\, C:\\$WINDOWS.~BT\, C:\dell\, C:\Intel\, C:\MSOCache\, C:\Program Files\, C:\Program Files (x86)\, C:\Games\, C:\Windows.old\
- D:\Users\%username%\AppData\Roaming\, D:\Users\%username%\AppData\Local\, D:\Windows\, D:\PerfLogs\, D:\ProgramData\Desktop\, D:\ProgramData\Microsoft\, D:\ProgramData\Package Cache\, D:\Users\Public\, D:\\$Recycle.Bin\, D:\\$WINDOWS.~BT\, D:\dell\, D:\Intel\, D:\MSOCache\, D:\Program Files\, D:\Program Files (x86)\, D:\Games\
- E:\Users\%username%\AppData\Roaming\, E:\Users\%username%\AppData\Local\, E:\Windows\, E:\PerfLogs\, E:\ProgramData\Desktop\, E:\ProgramData\Microsoft\, E:\ProgramData\Package Cache\, E:\Users\Public\, E:\\$Recycle.Bin\, E:\\$WINDOWS.~BT\, E:\dell\, E:\Intel\, E:\MSOCache\, E:\Program Files\, E:\Program Files (x86)\, E:\Games\
- F:\Users\%username%\AppData\Roaming\, F:\Users\%username%\AppData\Local\, F:\Windows\, F:\PerfLogs\, F:\ProgramData\Desktop\, F:\ProgramData\Microsoft\, F:\Users\Public\, F:\\$Recycle.Bin\, F:\\$WINDOWS.~BT\, F:\dell\, F:\Intel\

The executable retrieves the user name associated with the current thread by calling the GetUserNameW API:

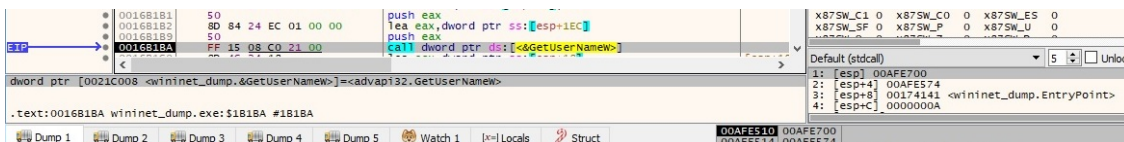


Figure 90

The malicious process is looking for a file called “PersonalID.txt” that doesn’t exist at this time:

Figure 91

CreateDirectoryW is utilized to create a directory called “C:\SystemID”:

Figure 92

The ransomware creates the file “C:\SystemID\PersonalID.txt” and writes the victim ID to it:

Figure 93

It’s very uncommon that the malware searches the system for a file called “I:\5d2860c89d774.jpg” (0xC0000000 = **GENERIC_READ | GENERIC_WRITE**, 0x1 = **FILE_SHARE_READ**, 0x3 = **OPEN_EXISTING** and 0x80 = **FILE_ATTRIBUTE_NORMAL**):

Figure 94

LoadCursorW is used to load the standard arrow resource from the executable (0x7F00 = **IDC_ARROW**):

Figure 95

The binary registers a window class using the RegisterClassExW routine:

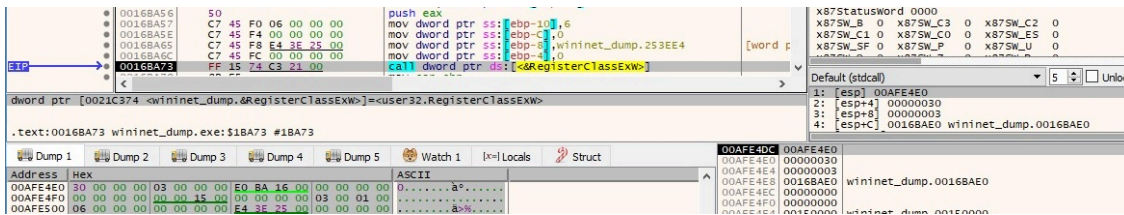


Figure 96

CreateWindowExW is utilized to create a new window called “LPCWSTRszTitle” (0xCF0000 = WS_OVERLAPPEDWINDOW and 0x80000000 = CW_USEDEFAULT):

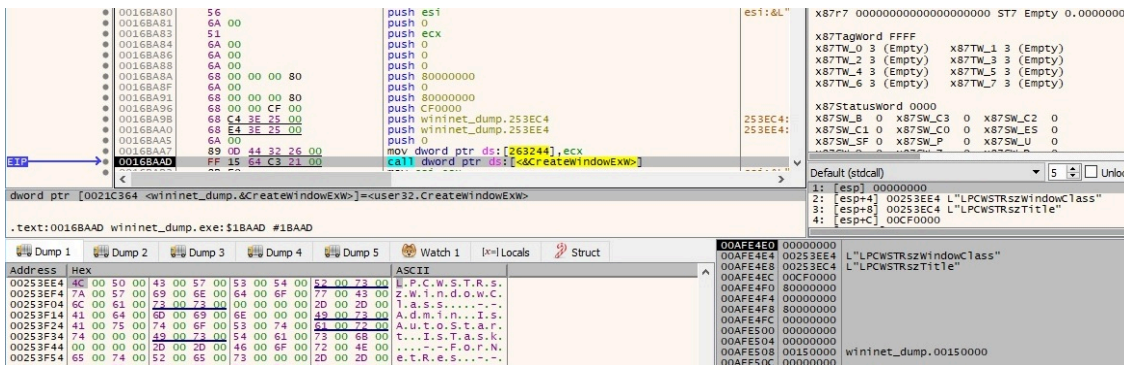


Figure 97

The window created earlier is hid by calling the ShowWindow routine (0x0 = SW_HIDE):



Figure 98

We need to analyze the window procedure defined in figure 96 (sub_16BAE0).

The malware uses the ntDllDefWindowProcW API in order to call the default window procedure whenever a particular message needs to be processed (0x24 = WM_GETMINMAXINFO, 0x81 = WM_NCCREATE, 0x83 = WM_NCCALCSIZE and 0x1 = WM_CREATE):



Figure 99

`GetLogicalDrives` is used to retrieve a bitmask that represents the available disk drives:

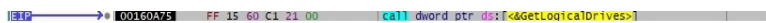


Figure 100

The ransomware forces the system not to display the critical-error message box and sending these errors to the calling process (`0x1 = SEM_FAILCRITICALERRORS`):

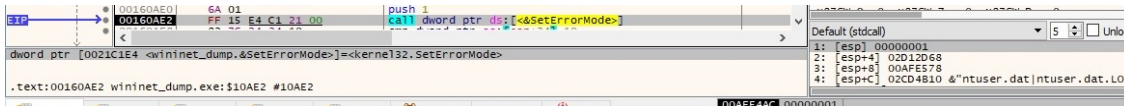


Figure 101

The file extracts the type of the drives by calling the `GetDriveTypeA` API and compares it with 2 (`DRIVE_REMOVABLE`), 3 (`DRIVE_FIXED`), 4 (`DRIVE_REMOTE`) and 6 (`DRIVE_RAMDISK`):

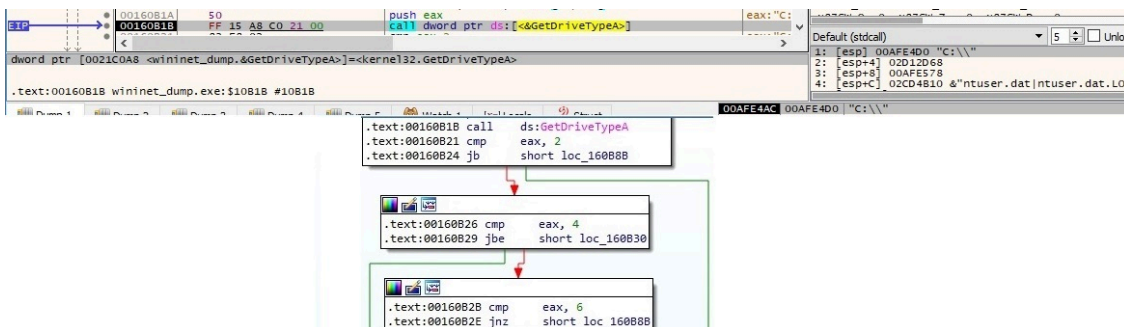


Figure 102

Two new threads are created using the `CreateThread` function:

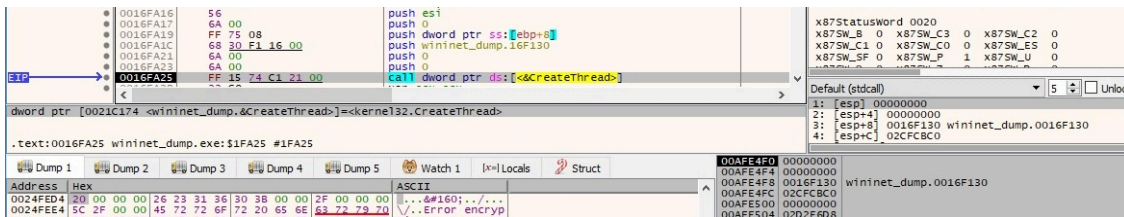


Figure 103

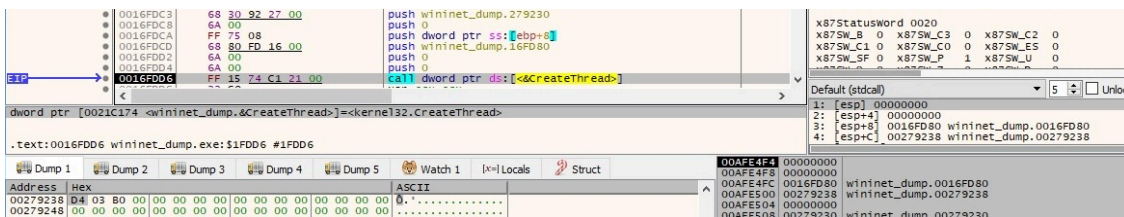


Figure 104

The file retrieves a message from the message queue by calling the GetMessageW routine, translates virtual-key messages into character messages using TranslateMessage, and finally dispatches a message to a window procedure using DispatchMessageW:

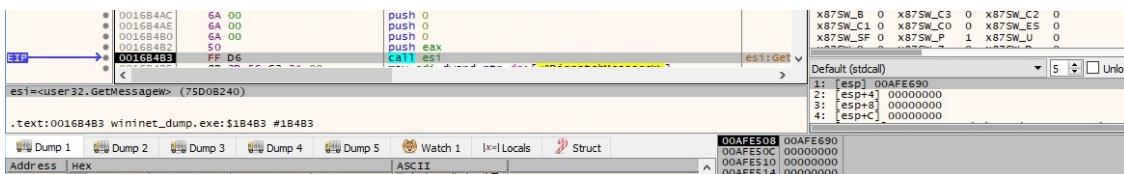


Figure 105

Thread activity – sub_16FD80

The malware enumerates all resources on the network via a function call to WNetOpenEnumW (0x2 = RESOURCE_GLOBALNET):

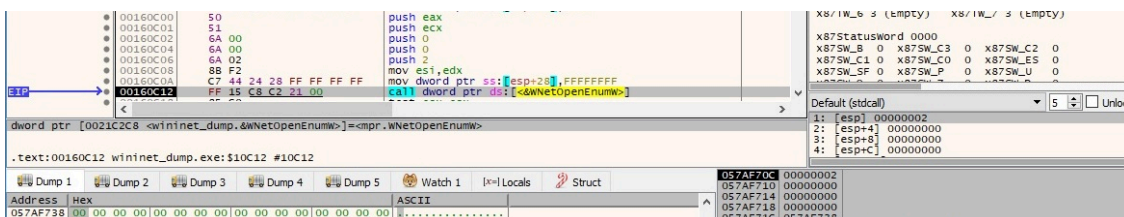


Figure 106

WNetEnumResourceW is utilized to further enumerate the network resources:

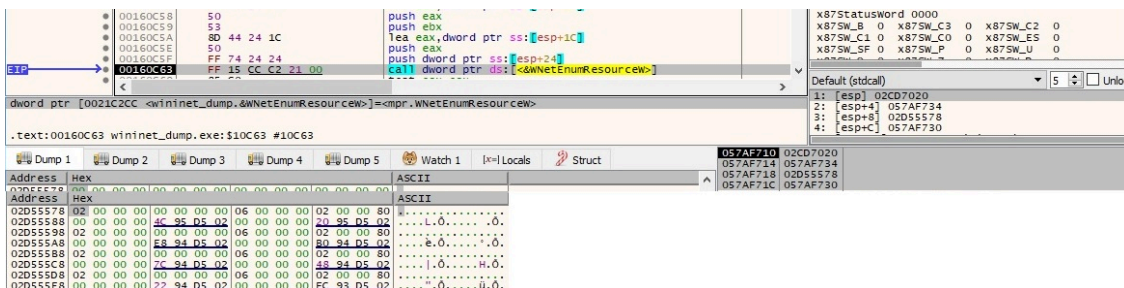


Figure 107

The message **DBT_DEVICEREMOVECOMPLETE** (“A device or piece of media has been removed”) is sent to the window created earlier, and its procedure will handle it:

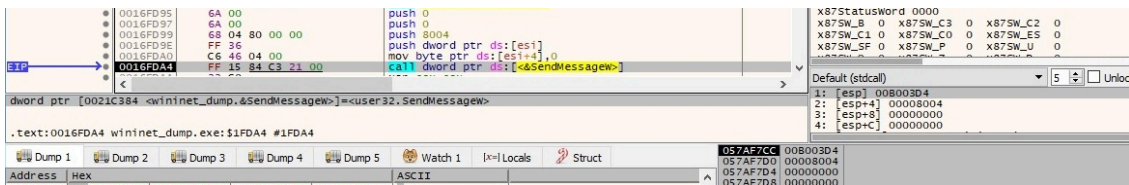


Figure 108

When the window procedure receives the message, it calls the **GetComputerNameW** API in order to get the NetBIOS name of the local machine:

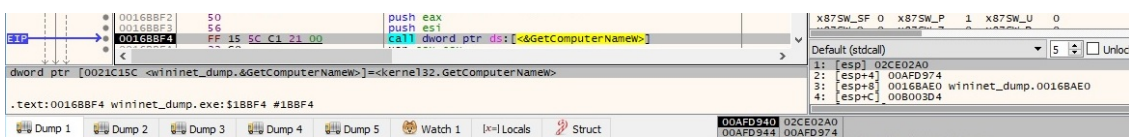


Figure 109

Thread activity – sub_16F130

The ransomware creates the ransom note called “_readme.txt” in every directory that it encrypts:

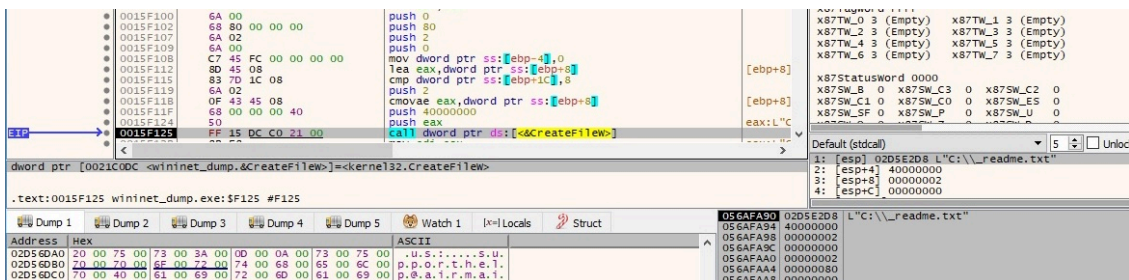


Figure 110

The ransom note is populated by calling the **WriteFile** function, as shown in figure 111:

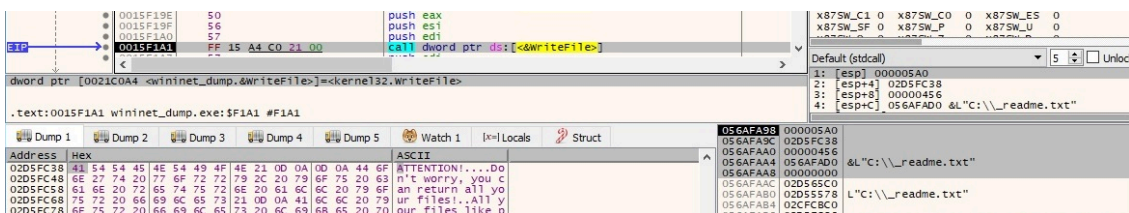


Figure 111

An example of a ransom note is highlighted below:

```

1  ATTENTION!
2
3  Don't worry, you can return all your files!
4  All your files like pictures, databases, documents and other important are encrypted with strongest encryption and unique key.
5  The only method of recovering files is to purchase decrypt tool and unique key for you.
6  This software will decrypt all your encrypted files.
7  What guarantees you have?
8  You can send one of your encrypted file from your PC and we decrypt it for free.
9  But we can decrypt only 1 file for free. File must not contain valuable information.
10 You can get and look video overview decrypt tool:
11 https://we.tl/t-1JwFK5rT39
12 Price of private key and decrypt software is $980.
13 Discount 50% available if you contact us first 72 hours, that's price for you is $490.
14 Please note that you'll never restore your data without payment.
15 Check your e-mail "Spam" or "Junk" folder if you don't get answer more than 6 hours.
16
17
18 To get this software you need write on our e-mail:
19 manager@mailtemp.ch
20
21 Reserve e-mail address to contact us:
22 supportthehelp@airmail.cc
23
24 Your personal ID:
25 0336gSd743duFHWn7bjwCkJEeUg8JHISzLqrwudidH8XsPzHDt1
    
```

Figure 112

The files are enumerated using the FindFirstFileW and FindNextFileW APIs:

Figure 113

The directories mentioned under figure 89 will not be encrypted. The file extension is extracted by calling the PathFindExtensionW routine:

Figure 114

The files and file extensions mentioned under figure 89 will be skipped. The ransomware also avoids files that have the “.tisc” extension because this will be appended after the encryption is complete:

Figure 115

Each targeted file is opened using the CreateFileW routine:

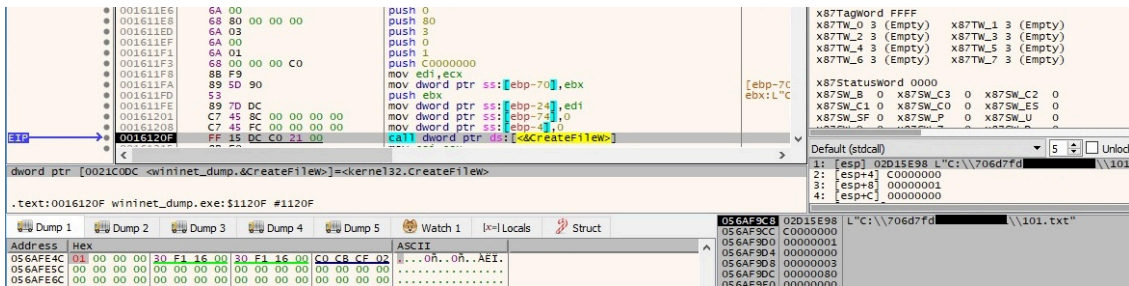


Figure 116

The file content is read by calling the ReadFile function:

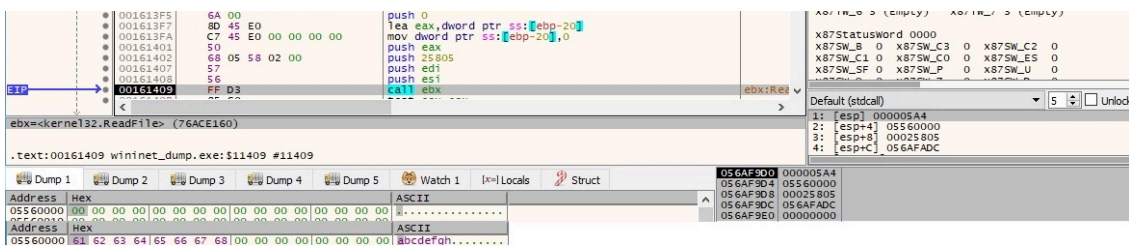


Figure 117

There is a function call to CryptAcquireContextW (as in figure 72) and another one to CryptCreateHash (as in figure 73). The malware hashes a buffer that contains the first 5 bytes from the targeted file and the RSA public key, as shown in figure 118:

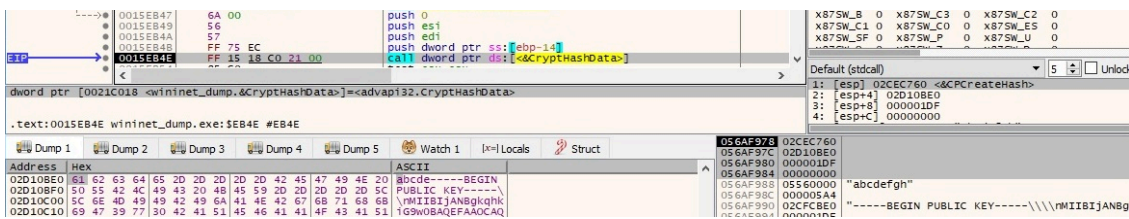


Figure 118

The MD5 hash value is extracted by calling the CryptGetHashParam routine (0x2 = HP_HASHVAL):



Figure 119

The binary creates a new UUID (16 random bytes) by calling the UuidCreate API (which internally uses CryptGenRandom):

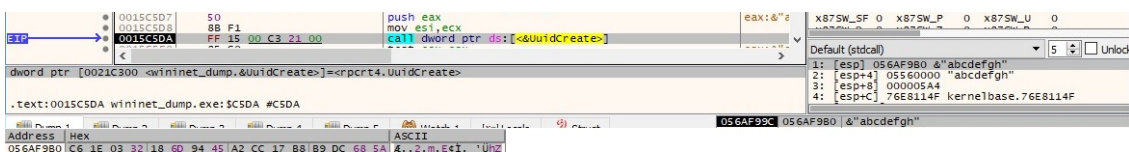


Figure 120

The process converts the UUID to a string using the UuidToStringA API:

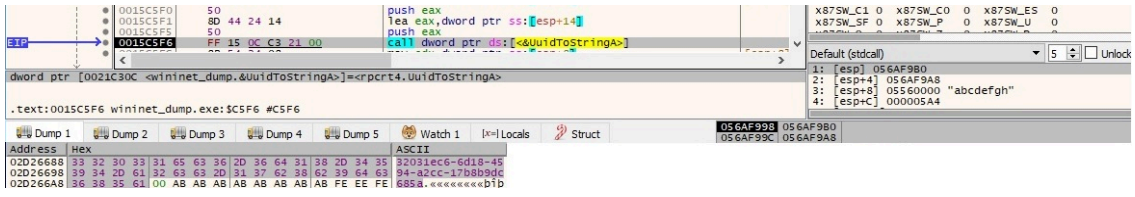


Figure 121

Based on the value generated above, the ransomware constructs the following Salsa20 matrix:

Address	Hex	ASCII
056AF9F4	65 78 70 61 33 32 30 33 31 65 63 36 2D 36 64 31	expa32031ec6-6d1
056AFA04	38 2D 34 35 6E 64 20 33 33 32 30 33 31 65 63 36	8-45nd 332031ec6
056AFA14	00 00 00 00 00 00 00 00 32 2D 62 79 39 34 2D 612-by94-a
056AFA24	32 63 63 2D 31 37 62 38 62 39 64 63 74 65 20 68	2cc-17b8b9dcte k

Figure 122

A snippet of the Salsa20 algorithm implemented by the malware is presented below:

```
.text:0015C142  
.text:0015C142 loc_15C142:  
.text:0015C142 add     eax, ebx  
.text:0015C144 mov     ebx, [ebp+var_30]  
.text:0015C147 rol     eax, 7  
.text:0015C14A xor     [ebp+var_8], eax  
.text:0015C14D mov     eax, [ebp+var_8]  
.text:0015C150 add     eax, [ebp+var_10]  
.text:0015C153 rol     eax, 9  
.text:0015C156 xor     [ebp+var_C], eax  
.text:0015C159 mov     eax, [ebp+var_C]  
.text:0015C15C add     eax, [ebp+var_8]  
.text:0015C15F rol     eax, 0Dh  
.text:0015C162 xor     ebx, eax  
.text:0015C164 mov     eax, [ebp+var_C]  
.text:0015C167 add     eax, ebx  
.text:0015C169 mov     [ebp+var_30], ebx  
.text:0015C16C ror     eax, 0Eh  
.text:0015C16F xor     [ebp+var_10], eax  
.text:0015C172 mov     eax, [ebp+var_14]  
.text:0015C175 add     eax, [ebp+var_1C]  
.text:0015C178 rol     eax, 7  
.text:0015C17B xor     [ebp+var_20], eax  
.text:0015C17E mov     eax, [ebp+var_20]  
.text:0015C181 add     eax, [ebp+var_1C]  
.text:0015C184 rol     eax, 9  
.text:0015C187 xor     esi, eax  
.text:0015C189 mov     ebx, [ebp+var_28]  
.text:0015C18C mov     eax, [ebp+var_20]  
.text:0015C18F add     eax, esi  
.text:0015C191 rol     eax, 0Dh  
.text:0015C194 xor     [ebp+var_14], eax  
.text:0015C197 mov     eax, [ebp+var_14]  
.text:0015C19A add     eax, esi  
.text:0015C19C ror     eax, 0Eh  
.text:0015C19F xor     [ebp+var_1C], eax  
.text:0015C1A2 mov     eax, [ebp+var_4]  
.text:0015C1A5 add     eax, ebx  
.text:0015C1A7 rol     eax, 7  
.text:0015C1AA xor     edx, eax  
.text:0015C1AC mov     eax, [ebp+var_4]  
.text:0015C1AF add     eax, edx  
.text:0015C1B1 rol     eax, 9
```

Figure 123

The process encrypts the file content using the Salsa20 algorithm, however the first 5 bytes from the targeted file are not encrypted. Based on the strings presented in figure 124 and our analysis of the RSA implementation, we believe that the malware developers have included the OpenSSL code found at <https://github.com/openssl/openssl> (or similar code taken from other projects):

```

.text:001B9C8E
.text:001B9C8E loc_1B9C8E:
.text:001B9C8E call    sub_1A1630
.text:001B9C93 push    102h
.text:001B9C98 push    offset aCryptoEngineEn_0 ; ".\\crypto\\engine\\eng_table.c"
.text:001B9C9D push    1Eh
.text:001B9C9F push    9
.text:001B9CA1 call    sub_1A47A0
.text:001B9CA6 mov     ecx, [edi]
.text:001B9CA8 add     esp, 10h
.text:001B9CAB test   ecx, ecx
.text:001B9CAD jz     loc_1B9D70

.text:001AA782 push    64h ; 'd'
.text:001AA784 push    offset aCryptoEngineEn ; ".\\crypto\\engine\\eng_init.c"
.text:001AA789 push    1Eh
.text:001AA78B push    9
.text:001AA78D call    sub_1A47A0
.text:001AA792 add     esp, 10h

.text:001A1D40 push    esi
.text:001A1D41 push    8Bh ; '<' ; int
.text:001A1D46 push    offset aCryptoRsaRsaLi ; ".\\crypto\\rsa\\rsa_lib.c"
.text:001A1D4B push    58h ; 'X' ; size_t
.text:001A1D4D call    sub_1A4E50
.text:001A1D52 mov     esi, eax
.text:001A1D54 add     esp, 0Ch
.text:001A1D57 test   esi, esi
.text:001A1D59 jnz    short loc_1A1D77

.text:001AA2D0
.text:001AA2D0
.text:001AA2D0
.text:001AA2D0 sub_1AA2D0 proc near
.text:001AA2D0 mov     eax, offset off_25B754 ; "Eric Young's PKCS#1 RSA"
.text:001AA2D5 retn
.text:001AA2D5 sub_1AA2D0 endp
.text:001AA2D5

```

Figure 124

The binary encrypts the UUID generated before using the RSA public key embedded in the file:

The screenshot shows a debugger window with the following content:

- Assembly View:**
 - 001A9252: 6A 00 push 0
 - 001A9254: 6A 00 push 0
 - 001A9256: FF 74 24 28 push dword ptr ss:[esp+28]
 - 001A925A: FF 74 24 30 push dword ptr ss:[esp+30]
 - 001A925E: 55 push ebp
 - 001A925F: 53 push ebx
 - 001A9260: E8 88 F5 00 00 call Wininet.Dump.1B87F0
- Memory Dump:**
 - Address: 02CD6FD8, Hex: 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA, ASCII: \0.\0.\0.\0.\0.\0.\0.\0
 - Address: 02CD6FE8, Hex: 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA, ASCII: \0.\0.\0.\0.\0.\0.\0.\0
 - Address: 02CD6FF8, Hex: 7D F8 25 54 C0 78 EB C9 08 CC 2C 5A 7C 64 76 C7, ASCII: 18ST[EE.I.Z]lvQ
 - Address: 02CD6FF8, Hex: 8F D5 8E FF 12 C6 CA B1 06 EE 2A 75 FB 6A E2 3B, ASCII: 0.Y.Ez.TUjA;
 - Address: 02CD7008, Hex: 8F DF DC 29 57 20 83 D3 8E E9 14 97 F4 A6 2D, ASCII: 20Uj\0ne.\0-
 - Address: 02CD7018, Hex: C9 77 99 7F 91 D8 41 39 9A 0E 07 12 E1 3D 95 29, ASCII: Ew...DA9...a=)
 - Address: 02CD7028, Hex: 14 48 26 96 98 54 74 D3 60 8C 67 C3 2A 6E 48 7C, ASCII: .kE..TTO'wA"mH
 - Address: 02CD7038, Hex: 0C 9A 8E C8 89 87 7D 91 49 78 FC 64 70 1C 34 06, ASCII: .NE'.1{udp.4.
 - Address: 02CD7048, Hex: 72 B3 5D 8E 78 44 FE 5C 93 85 A7 E5 25 E3 08 EB, ASCII: *)\XD\..s&A.e
 - Address: 02CD7058, Hex: 07 D1 D9 64 5C 82 AC B3 F3 29 E8 66 6F 53 15 98, ASCII: xNud\..0)efos..
 - Address: 02CD7068, Hex: 8F D8 33 A7 08 8D 03 01 CF 55 80 FF 22 F3 3A 5D, ASCII: 20S...iU'0j
 - Address: 02CD7078, Hex: 28 8A 14 94 18 D0 27 D1 B2 CD 66 C9 08 80 88 2A, ASCII: +...Y^N'ifE..?
 - Address: 02CD7088, Hex: 7D 91 B2 A4 F2 48 66 EA 91 38 6E E6 D0 86 E0 9F, ASCII: j.*0HfE..n00A.
 - Address: 02CD7098, Hex: 8F 4E 3C E6 C2 AA 07 F4 56 97 6F D7 6A 54 E2 DA, ASCII: j\m&*.0v.dxj1AU
 - Address: 02CD70A8, Hex: DA DE A5 36 04 E0 98 05 23 2D 2D 55 E3 00 5D 76, ASCII: 0p46.1.-.--Ua.V
 - Address: 02CD70B8, Hex: E5 D2 6F DC C8 0A 1E 5D B5 8D 89 EB DC 2E 5F C0, ASCII: 000Ue..j\..eU..A
 - Address: 02CD70C8, Hex: 7D F9 5F EA 85 E4 06 E0 E1 D5 35 D9 26 34 E1, ASCII: pJU.e.a.8a0Ue4
- Registers:**
 - ESP: 02D18230
 - EAX: 00000100
 - ECX: 02D26568
 - EDX: 00000025
- String List:**
 - 056AF4F4: 00000100
 - 056AF4F8: 02D26568 "32031ec6-6d18-4594-a2cc-17b8b9dc685a"
 - 056AF4FC: 00000025
 - 056AF500: 00000000
 - 056AF504: 00000000

Figure 125

The encrypted content is written to the file using WriteFile, as shown below:

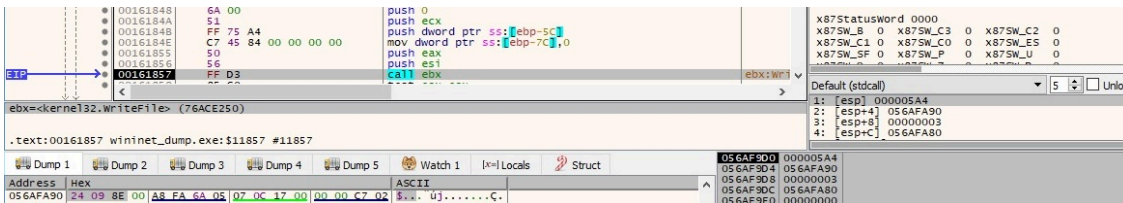


Figure 126

The malicious binary writes the encrypted UUID using the same API:

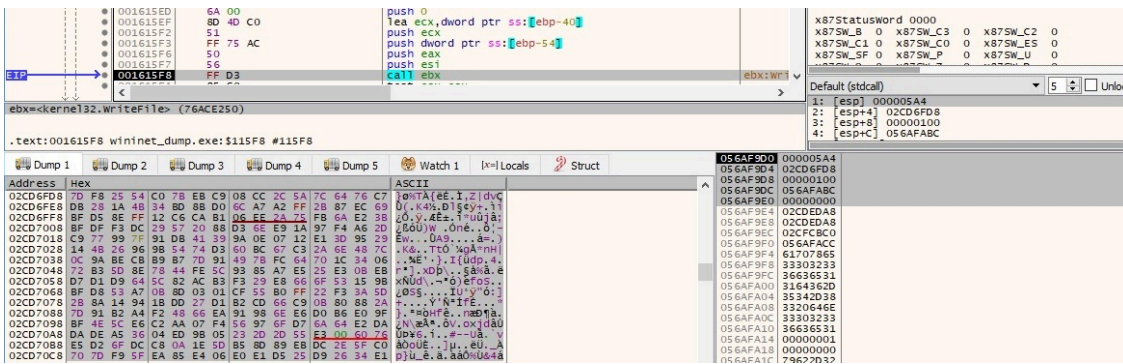


Figure 127

The offline ID is also added to the encrypted file:

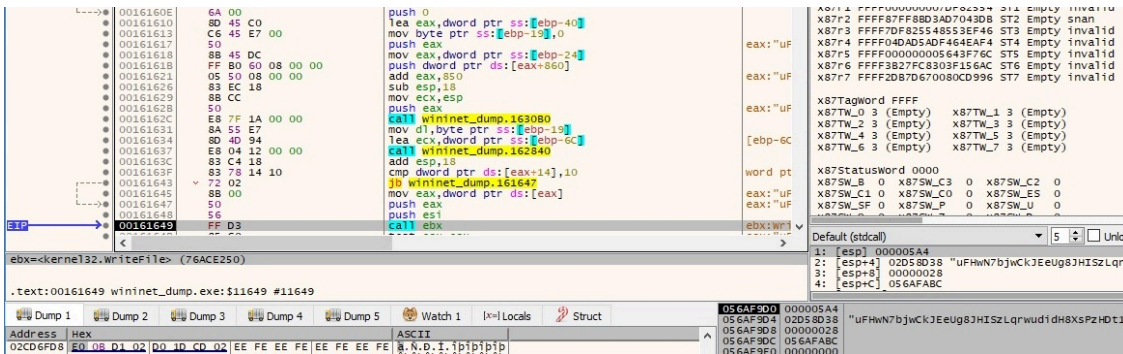


Figure 128

The value “{36A698B9-D67C-4E07-BE82-0EC5B14B4DF5}” is also added to the encrypted file:

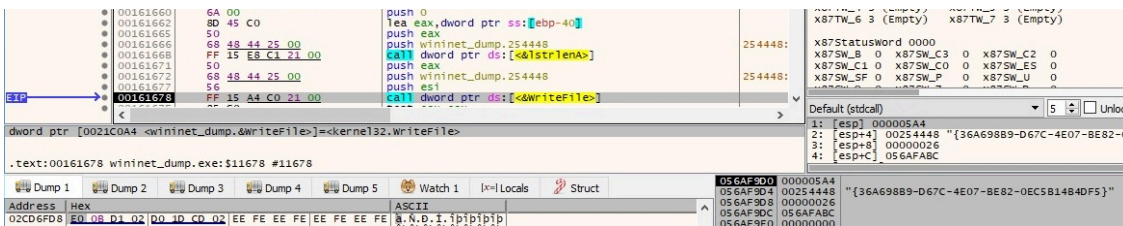


Figure 129

The encrypted file extension is changed to “.tisc” by the ransomware:

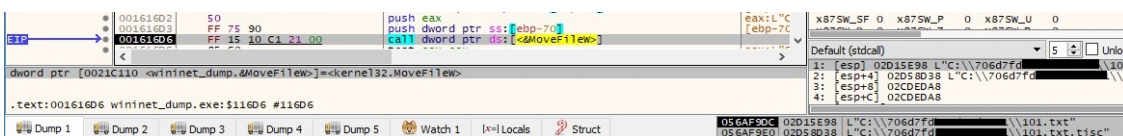


Figure 130

The encrypted file has the following structure that highlights different elements:

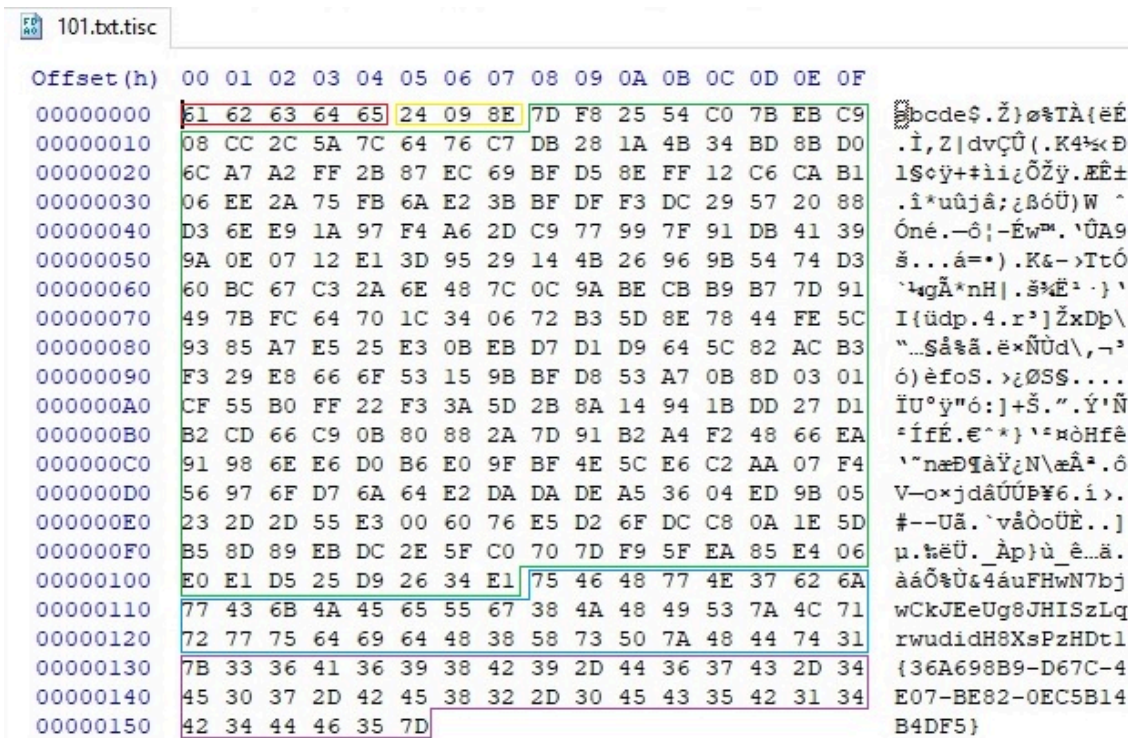


Figure 131

“-AutoStart” parameter

The activity is similar to the case discussed above.

“-Admin IsNotAutoStart IsNotTask” parameters

The binary establishes a connection to the service control manager by calling the OpenSCManagerW routine (0x1 = SC_MANAGER_CONNECT):

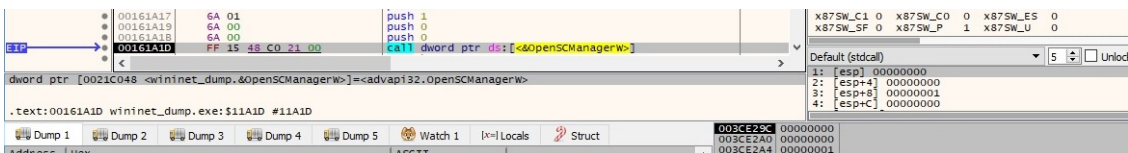


Figure 132

A service called “MYSQL” is opened by the process via a function call to OpenServiceW (0x20 = SERVICE_STOP):

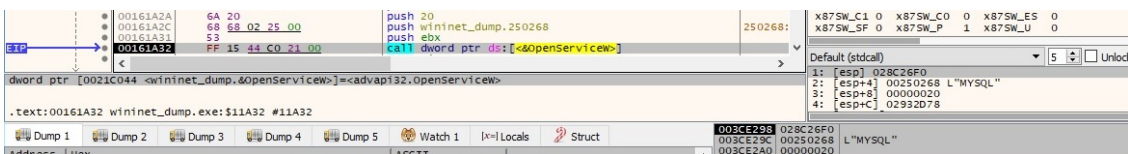


Figure 133

Whether the service would exist on a host, the ransomware would stop it using the ControlService API:

```

.text:00161A3E push    edi
.text:00161A3F lea    eax, [ebp+ServiceStatus]
.text:00161A42 push    eax                ; lpServiceStatus
.text:00161A43 push    1                ; dwControl
.text:00161A45 push    esi                ; hService
.text:00161A46 call   ds:ControlService
.text:00161A4C mov    edi, ds:CloseServiceHandle
.text:00161A52 test   eax, eax
.text:00161A54 jz     short loc_161AA0

.text:00161A56 lea    eax, [ebp+ServiceStatus]
.text:00161A59 push    eax                ; lpServiceStatus
.text:00161A5A push    esi                ; hService
.text:00161A5B call   ds:QueryServiceStatus
.text:00161A61 test   eax, eax
.text:00161A63 jz     short loc_161A9D
    
```

Figure 134

The file decrypts another URL that will be used to download more malicious files, [http://znpst\[.\]top/dl/build2.exe](http://znpst[.]top/dl/build2.exe):

Address	Hex	ASCII
028C6188	68 74 74 70	http://znpst.top
028C6198	2F 64 6C 2F	/dl/build2.exe\$
028C61A8	75 6E 20 68	un http://secure
028C61B8	62 69 7A 2E	biz.org/files/1/
028C61C8	62 75 69 6C	build3.exe\$run.r

Figure 135

A new thread is created by calling the CreateThread function:

Figure 136

Thread activity – StartAddress (sub_16DBD0)

UuidCreate is utilized to generate a new UUID:

Figure 137

The UuidToStringA routine is used to convert the UUID to a string:



Figure 138

The malicious process creates a new directory based on the UUID generated above:

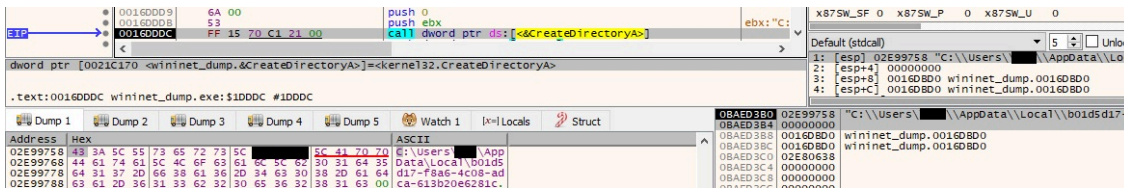


Figure 139

The binary performs a GET request to http[:]//znpst[.]top/dl/build2.exe using InternetOpenUrlA:

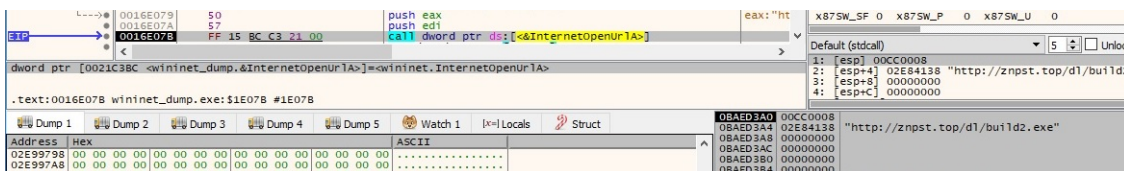


Figure 140

According to the analysis from

<https://any.run/report/cd6bf2f554a9aa446cb36d28e374e1010268cbda8f55eb0043f86e2724128be/152e55c2-5e8f-4fe2-a764-7876ba00f03e>, the above executable is a malware called Ursnif (banking Trojan).

The status code is extracted by calling the HttpQueryInfoW routine (0x20000013 = **HTTP_QUERY_FLAG_NUMBER | HTTP_QUERY_STATUS_CODE**):

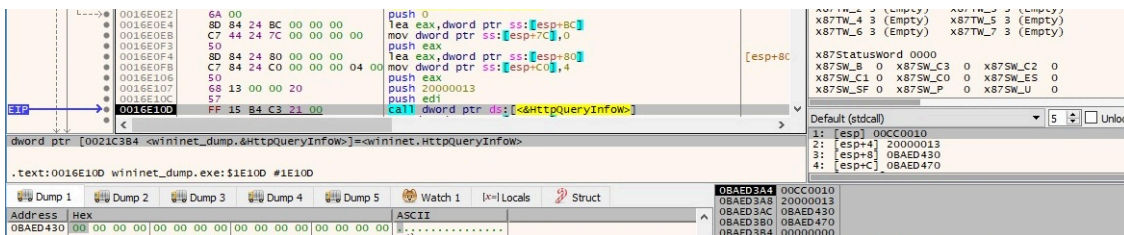


Figure 141

A file called “build2.exe” is created in the new directory:

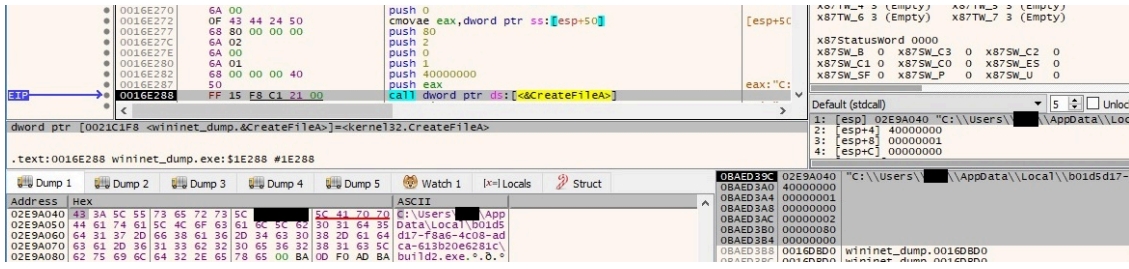


Figure 142

The InternetReadFile routine is utilized to read the executable from the server, as displayed in figure 143:

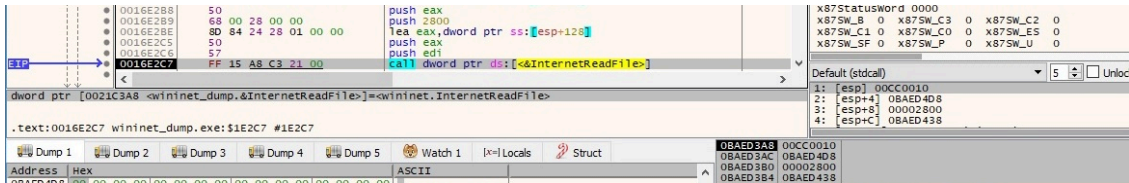


Figure 143

ShellExecuteA is used to run the newly created executable:

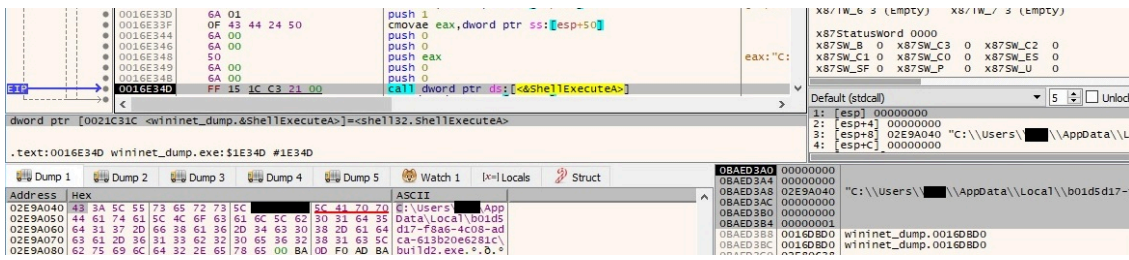


Figure 144

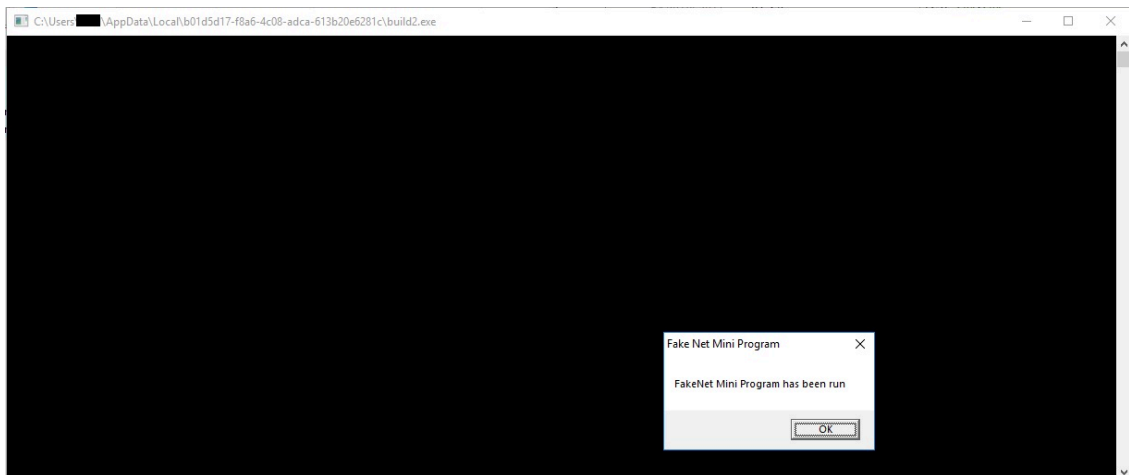


Figure 145

The binary performs a GET request to [http://securebiz\[.\]org/files/1/build3.exe](http://securebiz[.]org/files/1/build3.exe) using InternetOpenUrlA:

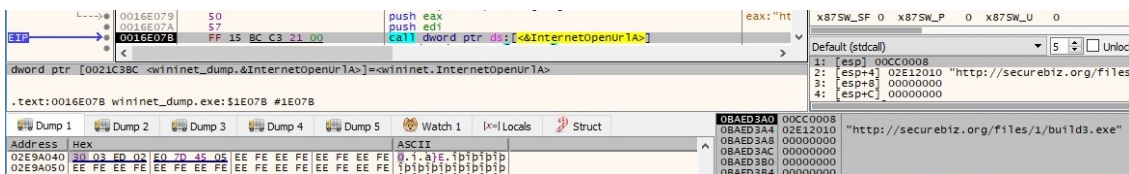


Figure 146

According to multiple online resources, the above file is supposed to be an infamous info-stealer called Vidar. The process of reading data from the server, creating the malicious file, etc. is the same as above and isn't explained again.

For completeness, we will also provide details about the other parameters that can be used, as displayed at <https://app.any.run/tasks/635cd7df-e4b7-4d1a-a937-e8d8599e6c72/>.

“-ForNetRes “jwvfPPgZoQyg6Q6he8weP7iDsH9FKc74ICjysAt2” r77yXePcnmrctJPWrZCcbJgUIAtOa1FC9Na710t1 IsNotAutoStart IsNotTask” parameters

The binary creates a mutex called “{FBB4BCC6-05C7-4ADD-B67B-A98A697323C1}” using the CreateMutexA API:

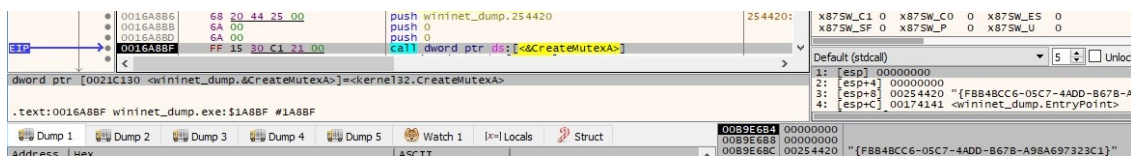


Figure 147

According to online sources, the first parameter can be considered as a Key and the second one as a Personal ID. The malware performs a hashing operation (MD5) on the Key:

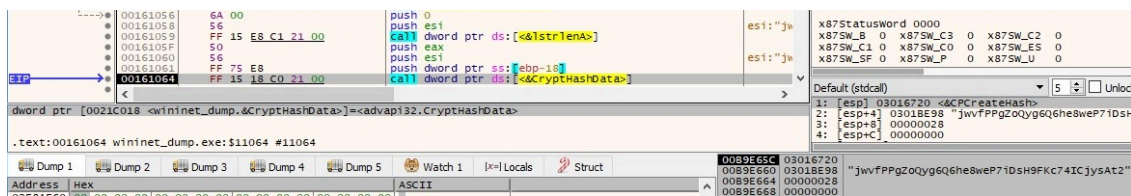


Figure 148

The hash value is extracted using the CryptGetHashParam function (0x2 = HP_HASHVAL):

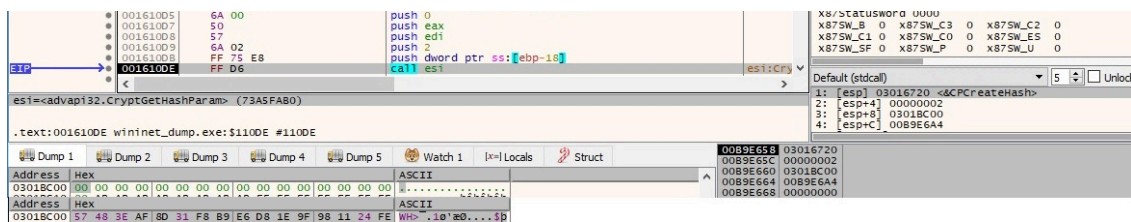


Figure 149

The execution flow is similar to the one starting with figure 90 and will not be reiterated.

“-Service 4904 “jwvfPPgZoQyg6Q6he8weP7iDsH9FKc74ICjysAt2” r77yXePcnmrctJPWrZCcbJgUIAtOa1FC9Na710t1” parameters

The above value represents the parent process ID, which is converted from string to a long integer value:

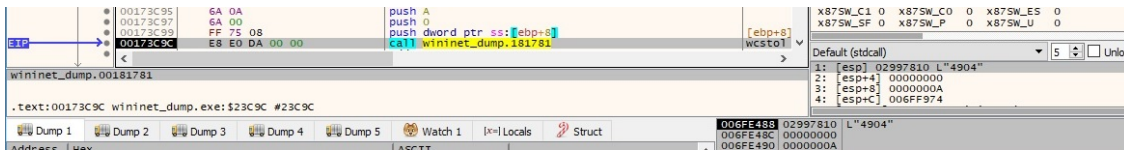


Figure 150

The ransomware opens the local process object using the OpenProcess routine (0x100000 = SYNCHRONIZE):

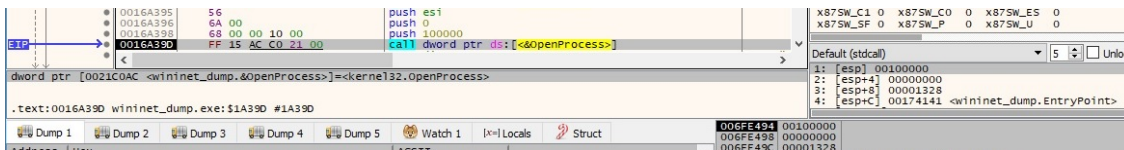


Figure 151

After the parent process enters the signaled state, the file dispatches incoming sent messages, checks for posted messages, and then retrieves the messages:



Figure 152

The malicious binary retrieves the exit code of the current process and then kills itself using TerminateProcess:

```
.text:0016A415 call ds:GetCurrentProcess
.text:0016A41B mov esi, eax
.text:0016A41D mov [esp+14D0h+ExitCode], 0
.text:0016A425 lea eax, [esp+14D0h+ExitCode]
.text:0016A429 push eax ; lpExitCode
.text:0016A42A push esi ; hProcess
.text:0016A42B call ds:GetExitCodeProcess
.text:0016A431 push [esp+14D0h+ExitCode] ; uExitCode
.text:0016A435 push esi ; hProcess
.text:0016A436 call ds:TerminateProcess
```

Figure 153

Finally, we describe the case when the country code belongs to the following list: “RU”, “BY”, “UA”, “AZ”, “AM”, “TJ”, “KZ”, “KG”, “UZ” and “SY”.

CreateMutexA is utilized to create a mutex called “{FBB4BCC6-05C7-4ADD-B67B-A98A697323C1}”:

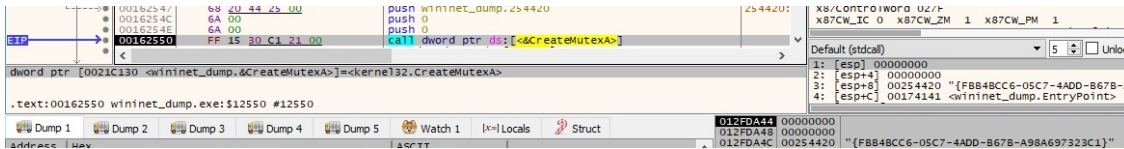


Figure 154

A batch file called “delselb.bat” is created in the %TEMP% directory:

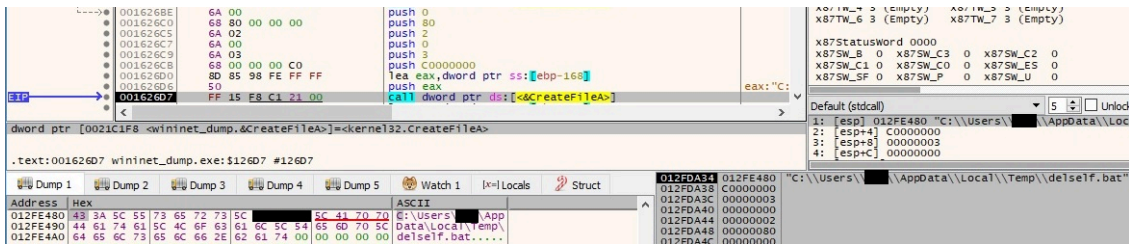


Figure 155

The above file is populated using the WriteFile API, and its content is displayed below:

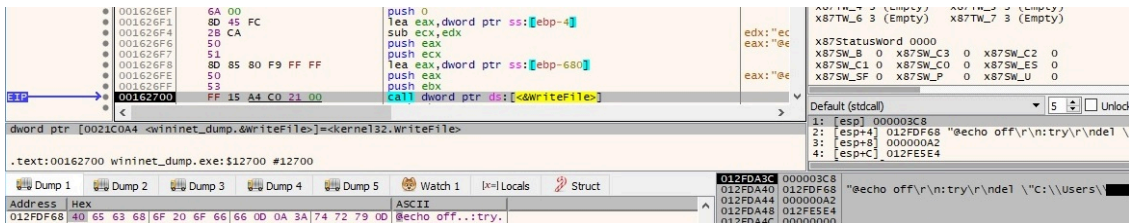


Figure 156

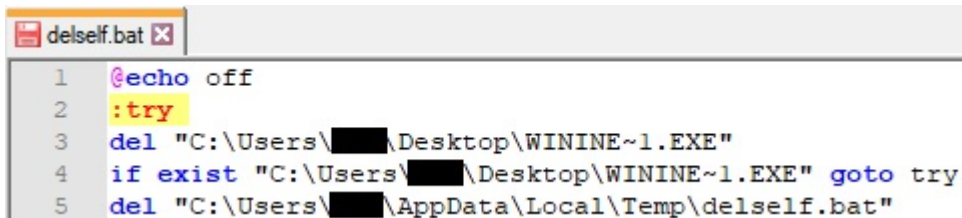


Figure 157

After the batch file finishes its execution, the malicious file and the script are deleted:

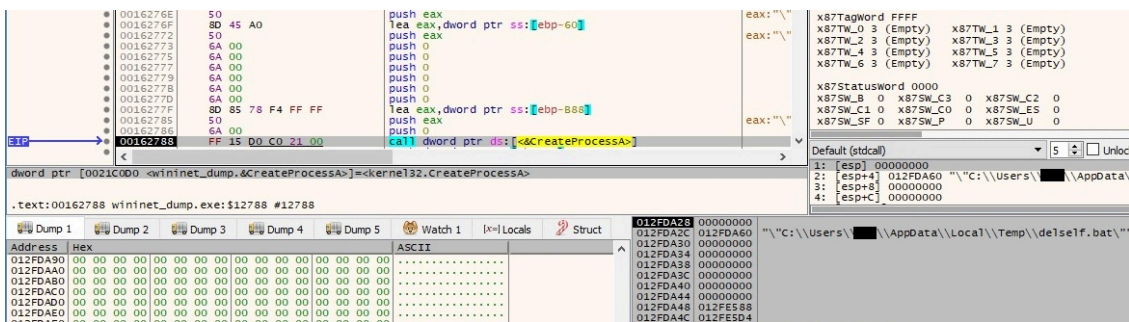


Figure 158

References

MSDN: <https://docs.microsoft.com/en-us/windows/win32/api/>, <https://docs.microsoft.com/en-us/windows/win32/taskschd/time-trigger-example-c—>

Fakenet: <https://github.com/fireeye/flare-fakenet-ng>

Any.run: <https://app.any.run/tasks/635cd7df-e4b7-4d1a-a937-e8d8599e6c72/>

VirusTotal:

<https://www.virustotal.com/gui/file/4380c45fd46d1a63cffe4d37cf33b0710330a766b7700af86020a936cdd09cbe>

MalwareBazaar:

<https://bazaar.abuse.ch/sample/4380c45fd46d1a63cffe4d37cf33b0710330a766b7700af86020a936cdd09cbe/>

OpenSSL: <https://github.com/openssl/openssl>

INDICATORS OF COMPROMISE

C2 domains:

- [securebiz\[.\]org](https://securebiz[.]org)
- [znpst\[.\]top](https://znpst[.]top)

SHA256: 4380c45fd46d1a63cffe4d37cf33b0710330a766b7700af86020a936cdd09cbe

Scheduled Task: “Time Trigger Task”

Registry key: HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run\SysHelper

User-agent: “Microsoft Internet Explorer”

PDB paths:

- “C:\xudihiguhe\jegovicatusoca\jijetogez\winucet\xusev\kucor.pdb”
- “e:\doc\my work (c++)_git\encryption\release\encrypt_win_api.pdb”

URLs:

- [http://securebiz\[.\]org/fhsgtsspen6/get.php](http://securebiz[.]org/fhsgtsspen6/get.php)
- <http://securebiz.org/files/1/build3.exe>
- <http://znpst.top/dl/build2.exe>
- <https://api.2ip.ua/geo.json>

Source: <https://cybergeeks.tech/a-detailed-analysis-of-the-stop-djvu-ransomware/>