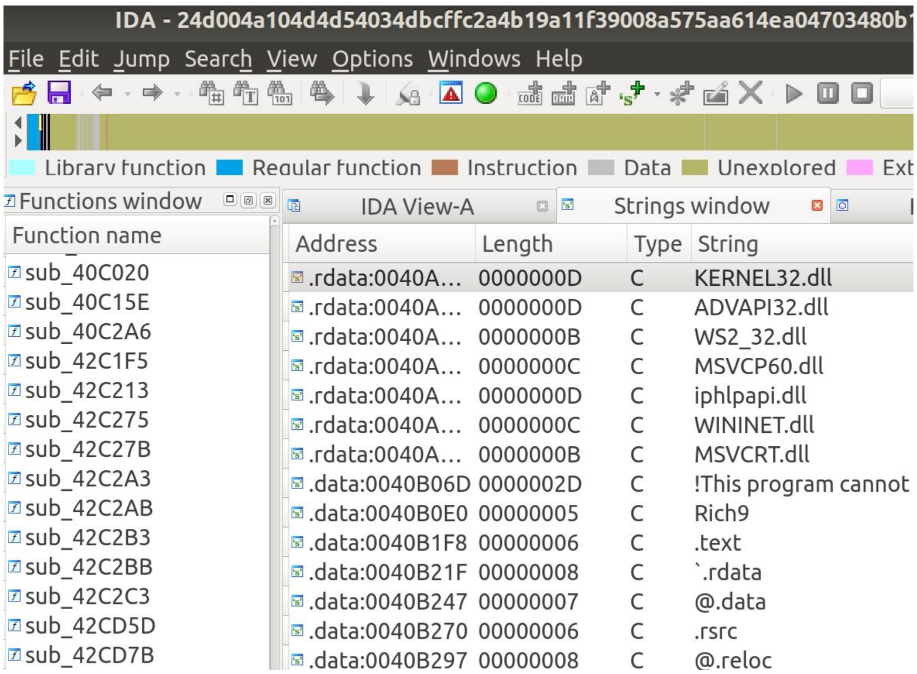
Extracting some Wannacry IOCs through static analysis

Pre-requisites:
- IDA Pro 7 (Free edition)
- Sample: 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c

Starting from the strings

Retrieving strings

First of all, get to the strings window (Shift+F12):



Let's see some of the interesting strings we can find here:

.data:004312B8	0000000A	C	\\\\%s\\IPC$
.data:004312C4	0000002B	C	Microsoft Base Cryptographic Provider v1.0
.data:004312F0	0000000C	C	%d.%d.%d.%d
.data:004312FC	0000000C	C	mssecsvc2.0
.data:00431308	00000028	C	Microsoft Security Center (2.0) Service
.data:00431330	0000000F	C	%s -m security
.data:00431344	00000011	C	C:\\%s\\qeriuwjhrf
.data:00431358	00000009	C	C:\\%s\\%s
.data:00431364	00000008	C	WINDOWS
.data:0043136C	0000000D	C	tasksche.exe
.data:004313D0	00000039	C	http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com

We already got many IOCs just by checking the strings. Looks like this sample is not packed/encrypted or its strings encoded/encrypted.

Analyzing the “mssecsvc2.0” string

Defining the string (if it’s not defined)

Double click on the on the “mssecsvc2.0” string:

```
.data:004312FC ; CHAR unk_4312FC[]
.data:004312FC unk_4312FC      db  6Dh ; m            ; DATA XREF: sub_407C40+55↑o
.data:004312FC                                        ; .text:00408008↑o ...
.data:004312FD                 db  73h ; s
.data:004312FE                 db  73h ; s
.data:004312FF                 db  65h ; e
.data:00431300                 db  63h ; c
.data:00431301                 db  73h ; s
.data:00431302                 db  76h ; v
.data:00431303                 db  63h ; c
.data:00431304                 db  32h ; 2
.data:00431305                 db  2Eh ; .
.data:00431306                 db  30h ; 0
.data:00431307                 db    0
```

This looks like the starting address of an ASCII string. Each byte representing a character in the string and ending with a null (0x0) byte.

You can define that list of bytes by right clicking on the starting address and selecting “mssecsvc2.0”, or just click “a” after putting the cursor in the address. Afterwards, it will look like this:

```
.data:004312FC ; CHAR aMssecsvc20[]
.data:004312FC aMssecsvc20     db 'mssecsvc2.0',0      ; DATA XREF: sub_407C40+55↑o
.data:004312FC                                         ; .text:00408008↑o ...
```

Now the string has a memorable name that will make it easier to identify once you look at the disassembly.

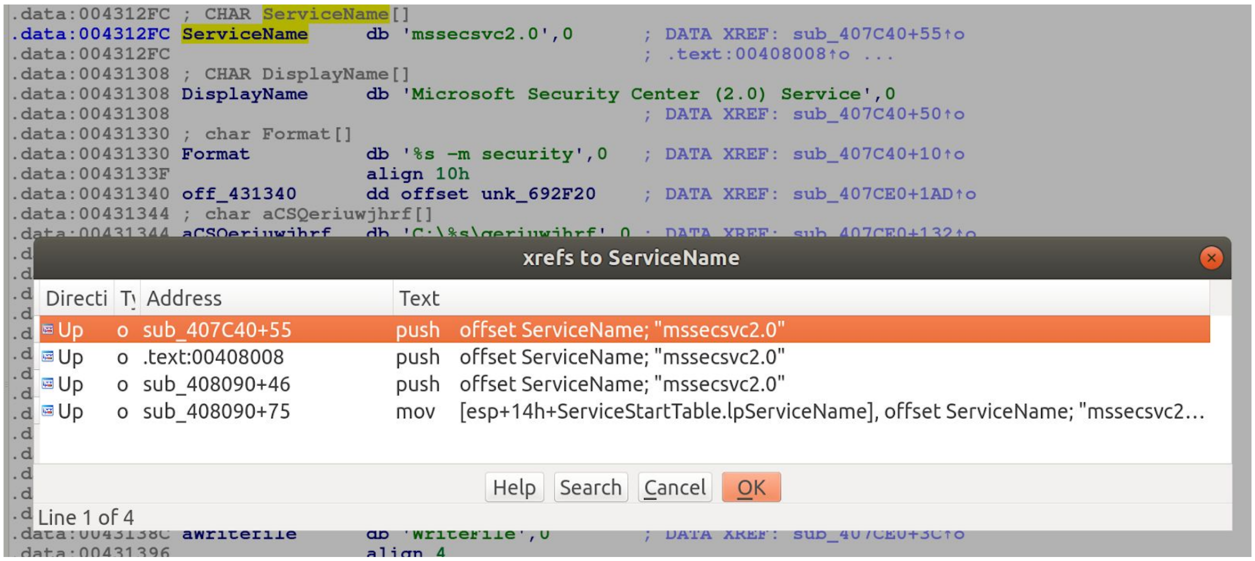
If you are lucky and your disassembler is clever enough, it might already have picked a meaningful name:

```
.data:004312FC ; CHAR ServiceName[]
.data:004312FC ServiceName     db 'mssecsvc2.0',0      ; DATA XREF: sub_407C40+55↑o
.data:004312FC                                         ; .text:00408008↑o ...
```

In this case, IDA Pro named this variable “ServiceName”.

Finding Cross References

With your cursor on the name of the string, click “x”:



Looks like there are 4 different places from where that string is used.

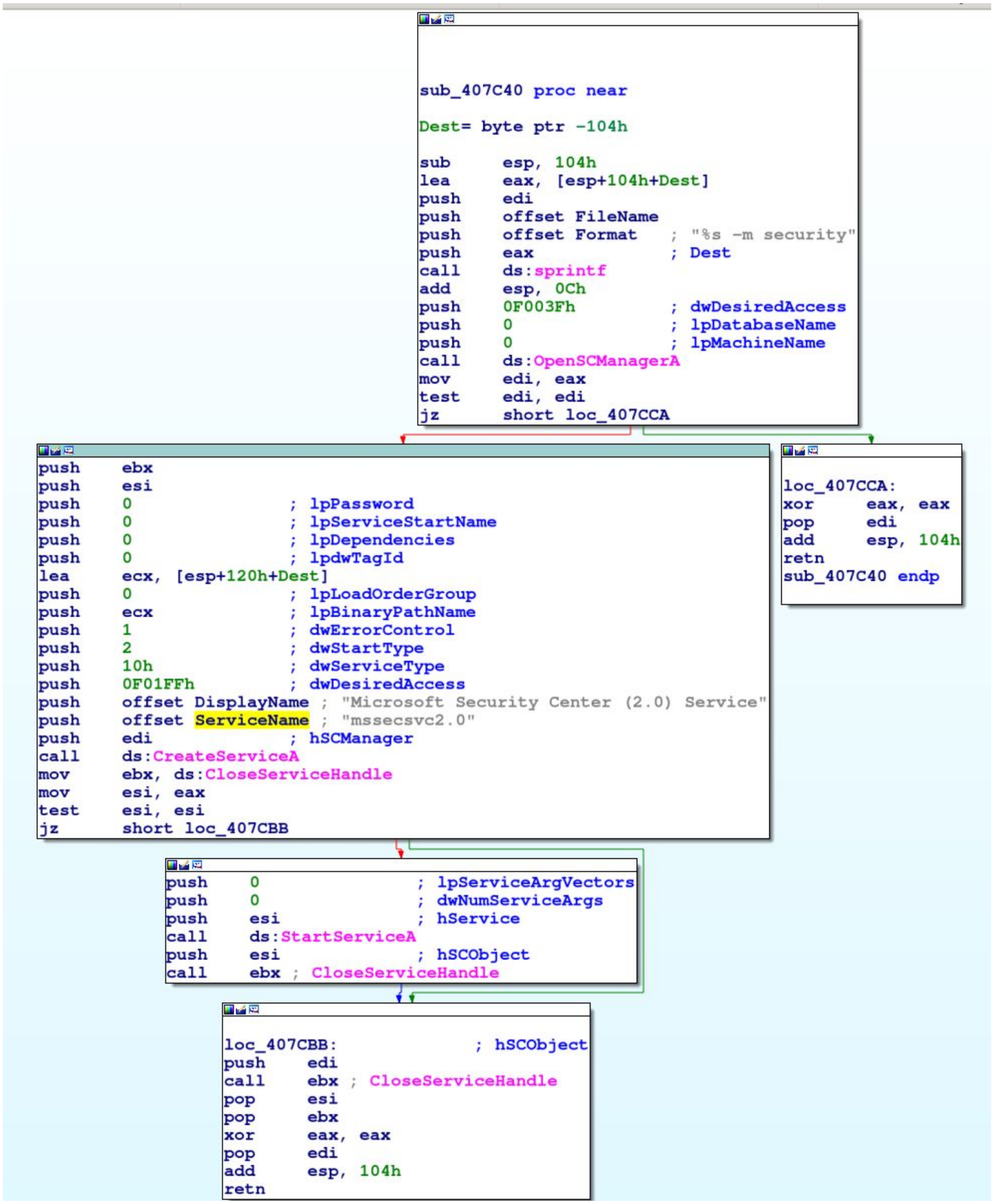
Let’s select the first entry in that list. By double clicking in the entry we’ll get to the code:

```
push    0               ; lpLoadOrderGroup
push    ecx             ; lpBinaryPathName
push    1               ; dwErrorControl
push    2               ; dwStartType
push    10h             ; dwServiceType
push    0F01FFh         ; dwDesiredAccess
push    offset DisplayName ; "Microsoft Security Center (2.0) Service"
push    offset ServiceName ; "mssecsvc2.0"
push    edi             ; hSCManager
call    ds:CreateServiceA
```

Analyzing the function

Now that we listed the code that uses that string, let’s analyze the function to which that code belongs.

By right clicking in anywhere outside the blocks with code (in the background), you’ll see the “Fit Window” option (“w”). By selecting that option, you’ll get a view of the full graph:



This function is rather small and it’s only formed by 5 basic blocks.

The following functions can be called:

- snprintf
- OpenSCManagerA
- CreateServiceA
- StartServiceA
- CloseServiceHandle

To get used to the Windows API, the MSDN should be your best friend. For example, to know what the OpenSCManagerA function does, visit this link:

“Establishes a connection to the service control manager on the specified computer and opens the specified service control manager database.”

From all those functions, the most interesting ones is the CreateServiceA and StartServiceA. Let’s look at them:

- CreateServiceA

Look at its MSDN entry:

Creates a service object and adds it to the specified service control manager database.

Syntax

C++

```
SC_HANDLE WINAPI CreateService(
  _In_      SC_HANDLE hSCManager,
  _In_      LPCTSTR   lpServiceName,
  _In_opt_  LPCTSTR   lpDisplayName,
  _In_      DWORD     dwDesiredAccess,
  _In_      DWORD     dwServiceType,
  _In_      DWORD     dwStartType,
  _In_      DWORD     dwErrorControl,
  _In_opt_  LPCTSTR   lpBinaryPathName,
  _In_opt_  LPCTSTR   lpLoadOrderGroup,
  _Out_opt_ LPDWORD   lpdwTagId,
  _In_opt_  LPCTSTR   lpDependencies,
  _In_opt_  LPCTSTR   lpServiceStartName,
  _In_opt_  LPCTSTR   lpPassword
);
```

This function creates a new service and receives ¡13! Parameters. Let’s look at the basic block that calls the CreateServiceA function:

```
push    ebx
push    esi
push    0               ; lpPassword
push    0               ; lpServiceStartName
push    0               ; lpDependencies
push    0               ; lpdwTagId
lea     ecx, [esp+120h+Dest]
push    0               ; lpLoadOrderGroup
push    ecx             ; lpBinaryPathName
push    1               ; dwErrorControl
push    2               ; dwStartType
push    10h             ; dwServiceType
push    0F01FFh         ; dwDesiredAccess
push    offset DisplayName ; "Microsoft Security Center (2.0) Service"
push    offset ServiceName ; "mssecsvc2.0"
push    edi             ; hSCManager
call    ds:CreateServiceA
mov     ebx, ds:CloseServiceHandle
mov     esi, eax
test    esi, esi
jz      short loc_407CBB
```

Reminder:
The calls to Windows API functions in x86 follow the stdcall calling convention. Function parameters are pushed to the stack, from right to left (first pushed parameter is the right-most), callee cleans the parameters from the stack.

IDA Pro shows all the API parameter names as comments.

Let’s analyze some of the most interesting parameters:

- dwServiceType

Its value is SERVICE_WIN32_OWN_PROCESS = 0x10, which tells us, for example, that this is a “regular” user-process service (not a driver).

- dwStartType

Its value is SERVICE_AUTO_START = 0x2 which tells us that this is, most likely, the **persistence technique** used by Wannacry, defined by ATT&CK with the T1050 id.

- DisplayName

The DisplayName of the service is actually the “description” of the service. Its value: “Microsoft Security Center (2.0) Service”

- ServiceName

This is the name of the service. Its value: “mssecsvc2.0”

- BinaryPathName

The binary path name is a more complex thing to identify, let's go step by step.
BinaryPathName is set as follows:

```
lea     ecx, [esp+120h+Dest]
push    0                   ; lpLoadOrderGroup
push    ecx                 ; lpBinaryPathName
```

The binary path name is set from ecx, which contains the address of esp+0x120+Dest. The last time that "Dest" was touched in this function was in:

```
lea     eax, [esp+104h+Dest]
push    edi
push    offset FileName
push    offset Format       ; "%s -m security"
push    eax                 ; Dest
call    ds:sprintf
```

1. In eax, the address to the variable "Dest" is stored. IDA Pro names it "Dest" because it is used as the destination buffer in "sprintf".
2. The "push edi" instruction can be ignored (edi is popped at the end of the function).
3. The FileName address is pushed. IDA Pro names this variable FileName after analyzing the functions that are executed before.
4. The format string address is pushed.
5. The destination buffer address is pushed.

- sprintf(Dest, "%s -m security", FileName);

We know that the service binary will be something like:

- <FileName> -m security

- What is FileName?

We see that IDA Pro names that specific offset as FileName. Let's identify why. We can look at the cross references (the third) for this variable or just backtrack a couple of functions until we get to this code:

```
push    104h                ; nSize
push    offset FileName     ; lpFilename
push    0                   ; hModule
call    ds:GetModuleFileNameA
```

We can see that GetModuleFileNameA is called with an hModule value of 0.

```
DWORD WINAPI GetModuleFileName(
  _In_opt_ HMODULE hModule,
  _Out_    LPTSTR  lpFilename,
  _In_     DWORD   nSize
);
```

> The GetModuleFilename function “retrieves the fully qualified path for the file that contains the specified module. The module must have been loaded by the current process.”
>
> “If this parameter [hModule] is NULL, GetModuleFileName retrieves the path of the executable file of the current process.”

So, there you go. The BinaryPathName will contain the full path of the current binary being executed plus the “-m security” string.

- StartServiceA

The call to StartServiceA looks like this:

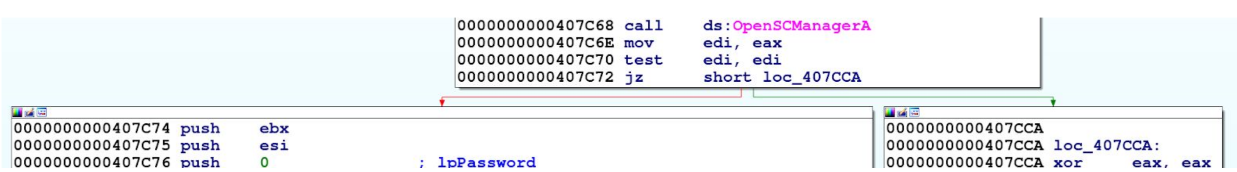
```
0000000000407CAD push    0                   ; lpServiceArgVectors
0000000000407CAF push    0                   ; dwNumServiceArgs
0000000000407CB1 push    esi                 ; hService
0000000000407CB2 call    ds:StartServiceA
```

“esi” contains a handle to the service. The other parameters are null.

We can assume the service will be started after its creation, but we’ll analyze it further.

Analyzing the function execution flow

- The first conditional can be found here:



```
call    ds:OpenSCManagerA
mov     edi, eax
test    edi, edi
jz      short loc_407CCA
```

1. The return value of the OpenSCManagerA call is stored in eax. If the function succeeds, it will return a handle, otherwise it will return NULL (zero).
2. Then eax is moved to edi.
3. Then edi is tested (a logical AND is performed). If edi is 0 the ZF will be set to 1.
4. Finally, there's a “jump if zero” instruction. Meaning that if edi was zero, the jump to loc_407CCA will be taken.

At loc_407CCA the function finishes.
Therefore, if the OpenSCManagerA is successful we jump to 0x407C74.

- The following conditionals are here:

```
07C9B call    ds:CreateServiceA
07CA1 mov     ebx, ds:CloseServiceHandle
07CA7 mov     esi, eax
07CA9 test    esi, esi
07CAB jz      short loc_407CBB

0000000000407CAD push    0                  ; lpServiceArgVectors
0000000000407CAF push    0                  ; dwNumServiceArgs
0000000000407CB1 push    esi                ; hService
0000000000407CB2 call    ds:StartServiceA
0000000000407CB8 push    esi                ; hSCObject
0000000000407CB9 call    ebx ; CloseServiceHandle

0000000000407CBB
```

The CreateServiceA call returns a handle if successful, it returns zero if it fails.
The same as before happens. If the call to CreateServiceA is successful, the service is started by calling to StartServiceA.

Conclusion

Most likely WannaCry uses a Windows service for persistence.

A service with the ability to get started each time that the system boots is created (and started after its creation).

The service name is “mssecsvc2.0” and its display name is “Microsoft Security Center (2.0) Service”.

Each time that the service gets started it will execute a binary (for which we don’t know the name) with the following options: “-m security”.

We could call this function **serviceCreation**.

Starting from known user code

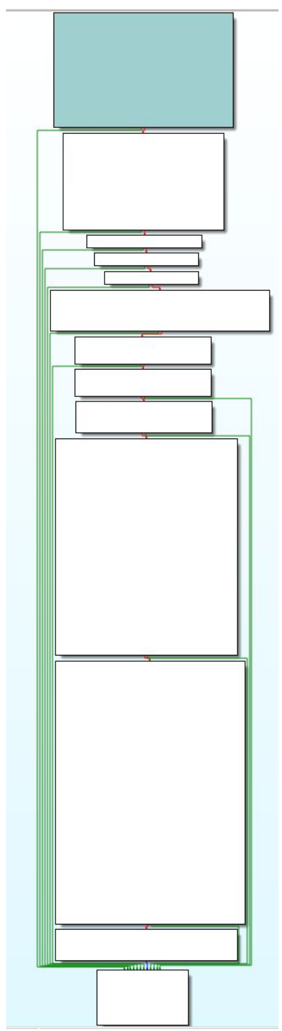
Choosing the target

If we look at the cross references for the newly analyzed serviceCreation function, we find that there’s only one place from which it gets called. The function at 0x407F20.

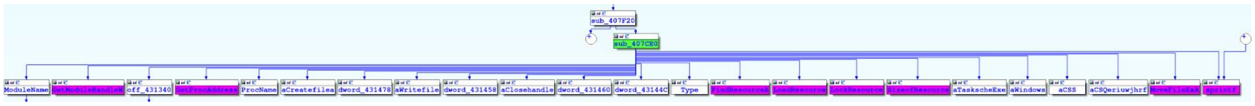
This function only calls to serviceCreation and to another function at **0x407F2C**. Let’s analyze it!

Visual analysis

The graph looks like this:



And the proximity browser gives us this layout:



We can see calls to:

- GetModuleHandleW
- GetProcAddress
- FindResourceA
- LoadResourceA
- LockResource
- SizeOfResource
- MoveFileExA
- sprintf

From this information we can infer that (maybe) some library functions will be resolved by the GetModuleHandleW + GetProcAddress, that there will be some work related to resources and that some files will be moved around.

From the proximity browser we can also see some interesting symbols:

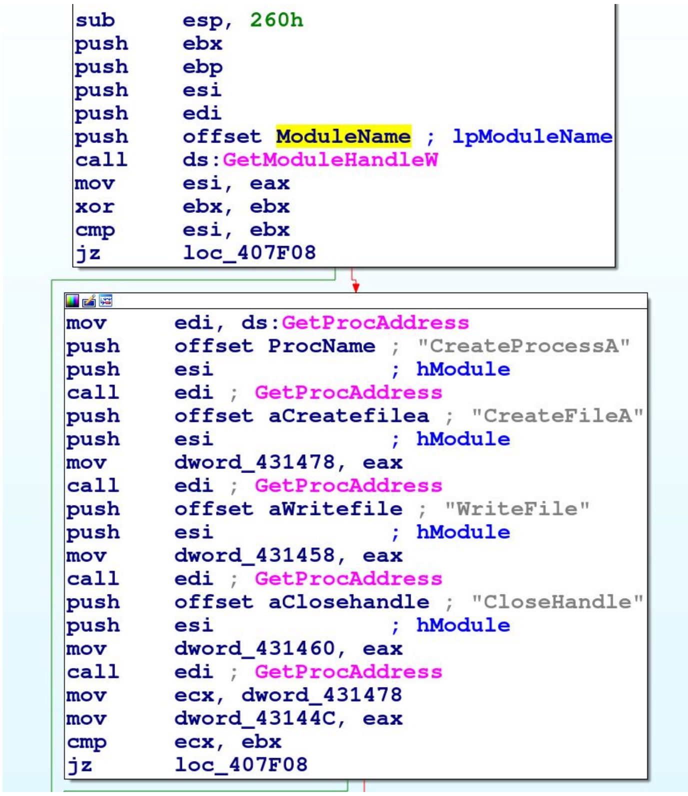
- ProcName
- aCreateFilea
- aWritefile
- TaskcheExe
- aWindows
- aCSS
- aCSQeriuwjhrf

1. aCreateFilea, aWritefile might be functions that get resolved using the GetProcAddress call.
2. TaskcheExe and aWindows look like system related artifacts.
3. For aCSS and aCSQeriuwjhrf we cannot infer anything yet.

Detailed analysis

- Dynamically resolving functions

The function starts with:



```
sub     esp, 260h
push    ebx
push    ebp
push    esi
push    edi
push    offset ModuleName ; lpModuleName
call    ds:GetModuleHandleW
mov     esi, eax
xor     ebx, ebx
cmp     esi, ebx
jz      loc_407F08
```

```
mov     edi, ds:GetProcAddress
push    offset ProcName ; "CreateProcessA"
push    esi             ; hModule
call    edi ; GetProcAddress
push    offset aCreatefilea ; "CreateFileA"
push    esi             ; hModule
mov     dword_431478, eax
call    edi ; GetProcAddress
push    offset aWritefile ; "WriteFile"
push    esi             ; hModule
mov     dword_431458, eax
call    edi ; GetProcAddress
push    offset aClosehandle ; "CloseHandle"
push    esi             ; hModule
mov     dword_431460, eax
call    edi ; GetProcAddress
mov     ecx, dword_431478
mov     dword_43144C, eax
cmp     ecx, ebx
jz      loc_407F08
```

In the first basic block we see a call to GetModuleHandleW whose parameter is ModuleName which is a memory location that does not have any value set yet (it will be set dynamically - we know that because if we double click on the name we see there's no data defined yet).

The GetModuleHandleW returns the handle to the module if the call is successful, it returns null otherwise.

After the call we have the following instructions:

```
.text:00407CF5    mov    esi, eax
.text:00407CF7    xor    ebx, ebx
.text:00407CF9    cmp    esi, ebx
```

```
.text:00407CFB                jz      loc_407F08
```

1. The returned value stored in eax is moved to esi.
2. ebx is set to 0.
3. esi (the returned value from GetModuleHandleW) is compared against 0 (ebx).
4. If esi is zero (if GetModuleHandleW failed) we jump to loc_407F08. Otherwise, we continue to the next basic block.

After that, we see four times a similar code construct:

```
mov     edi, ds:GetProcAddress
push    offset ProcName ; "CreateProcessA"
push    esi             ; hModule
call    edi ; GetProcAddress
```

Remember that in esi there's the return value of the GetModuleHandleW (a handle to a given unknown "ModuleName" module).

1. The address of GetProcAddress is stored in edi.
2. An address to the "CreateProcessA" string is pushed as a parameter to GetProcAddress.
3. The handle to the "ModuleName" is pushed as a parameter to GetProcAddress.
4. GetProcAddress is called through edi.

Basically, GetProcAddress resolves the address where a given function defined by its name is loaded in memory. In this case, the CreateProcessA function is being resolved.

This means that the "ModuleName" passed to the GetModuleHandleW function will likely be "Kernel32.dll" (the library that exports the CreateProcessA function).

A couple of instructions later we can see that the return address of this GetProcessAddress call is stored in a specific memory location:

```
mov     dword_431478, eax
```

We can rename this memory location as "CreateProcessA". This will most likely get called in the future.

We can do the same for the other resolved functions. The basic block will look like this:

```
mov     edi, ds:GetProcAddress
push    offset ProcName ; "CreateProcessA"
push    esi             ; hModule
call    edi ; GetProcAddress
push    offset aCreatefilea ; "CreateFileA"
push    esi             ; hModule
mov     CreateProcessA, eax
call    edi ; GetProcAddress
push    offset aWritefile ; "WriteFile"
push    esi             ; hModule
mov     CreateFileA_0, eax
call    edi ; GetProcAddress
push    offset aClosehandle ; "CloseHandle"
push    esi             ; hModule
mov     WriteFile, eax
call    edi ; GetProcAddress
mov     ecx, CreateProcessA
mov     CloseHandle_0, eax
cmp     ecx, ebx
jz      loc_407F08
```

If we put the cursor in the CreateProcessA rename location and scroll down, we will see at 0x00407EE8 the following call to that location:

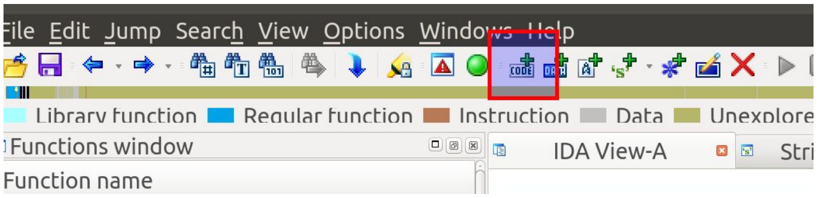
```
lea     ecx, [esp+274h+var_24C]
lea     edx, [esp+274h+Dest]
push    ecx
push    ebx
push    ebx
push    8000000h
push    ebx
push    ebx
push    ebx
push    edx
push    ebx
mov     [esp+298h+var_24C], 44h
mov     [esp+298h+var_21C], bx
mov     [esp+298h+var_220], 81h
call    CreateProcessA
test    eax, eax
jz      short loc_407F08
```

If we double click on the CreateProcessA string we see that IDA Pro recognizes it as a function and prints its signature:

```
.data:00431478 ; BOOL __stdcall CreateProcessA(LPCSTR lpApplicationName, LPSTR lpCommandLine,
.data:00431478 CreateProcessA  dd 0                    ; DATA XREF: sub_407CE0+35↑w
```

However, it’s not doing the same for the disassembly view.

To fix that, click on the CreateProcessA string again in the disassembly view and click in the highlighted icon (“Convert to Instruction”) - you can also type “c” to achieve the same outcome or go to Edit → Code:

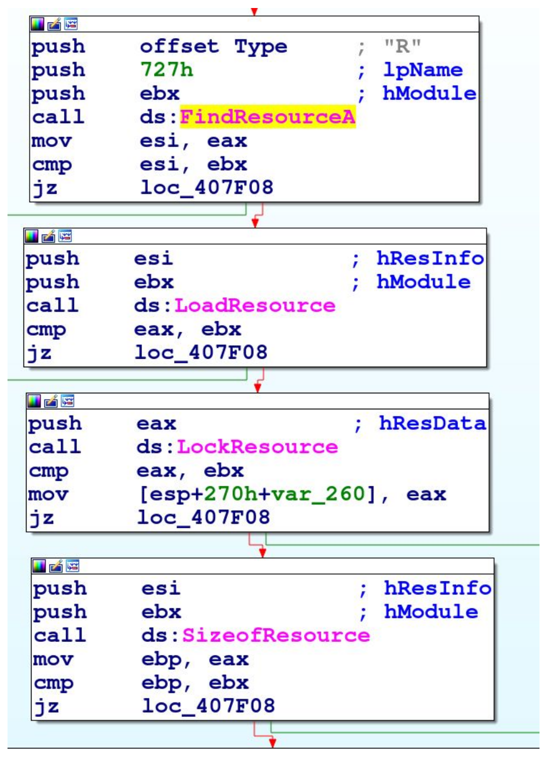


After typing “c”, IDA Pro will modify the disassembly view and show the parameter information for the CreateProcessA call:

```
lea     ecx, [esp+274h+StartupInfo]
lea     edx, [esp+274h+Dest]
push    ecx             ; lpStartupInfo
push    ebx             ; lpCurrentDirectory
push    ebx             ; lpEnvironment
push    8000000h        ; dwCreationFlags
push    ebx             ; bInheritHandles
push    ebx             ; lpThreadAttributes
push    ebx             ; lpProcessAttributes
push    edx             ; lpCommandLine
push    ebx             ; lpApplicationName
mov     [esp+298h+StartupInfo.cb], 44h
mov     [esp+298h+StartupInfo.wShowWindow], bx
mov     [esp+298h+StartupInfo.dwFlags], 81h
call    CreateProcessA
```

- Reading resources

The code that follows after resolving those function addresses is this:



We can see four basic blocks each one calling:

- FindResourceA: Retrieves a handle to a given resource (necessary for LoadResource).
- LoadResource: “Retrieves a handle that can be used to obtain a pointer to the first byte of the specified resource in memory.” (necessary for LockResource)
- LockResource: “Retrieves a pointer to the specified resource in memory.”
- SizeOfResource: “Retrieves the size, in bytes, of the specified resource.”

The FindResource call receives the hModule parameter which in this case is null, given that ebx is 0 (from a previous xor ebx, ebx).

```
.text:00407CF7          xor     ebx, ebx
```

Then, the IpName specifies the name of the resource. In this case, the name of the resource is the id 0x727.

Let’s look at the resources using radare2:

```
➜ analysis_2 r2 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c_wannacry
 -- When in doubt, try 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa; pd;'
[0x00409a16]> iR
Resource 0
        name: 1831
        timestamp: Thu Jan  1 01:00:00 1970
        vaddr: 0x007100a4
        size: 3.4M
        type: UNKNOWN
        language: LANG_ENGLISH
Resource 1
        name: 1
        timestamp: Thu Jan  1 01:00:00 1970
        vaddr: 0x00a6a0a4
        size: 944
        type: VERSION
        language: LANG_ENGLISH

[0x00409a16]> px @ 0x7100a4
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x007100a4  4d5a 9000 0300 0000 0400 0000 ffff 0000  MZ..............
0x007100b4  b800 0000 0000 0000 4000 0000 0000 0000  ........@.......
0x007100c4  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x007100d4  0000 0000 0000 0000 0000 0000 f800 0000  ................
0x007100e4  0e1f ba0e 00b4 09cd 21b8 014c cd21 5468  ........!..L.!Th
0x007100f4  6973 2070 726f 6772 616d 2063 616e 6e6f  is program canno
0x00710104  7420 6265 2072 756e 2069 6e20 444f 5320  t be run in DOS
0x00710114  6d6f 6465 2e0d 0d0a 2400 0000 0000 0000  mode....$.......
0x00710124  e0c5 3ad1 a4a4 5482 a4a4 5482 a4a4 5482  ..:...T...T...T.
0x00710134  dfb8 5882 a6a4 5482 cbbb 5f82 a5a4 5482  ..X...T..._...T.
0x00710144  27b8 5a82 a0a4 5482 cbbb 5e82 afa4 5482  '.Z...T...^...T.
0x00710154  cbbb 5082 a0a4 5482 67ab 0982 a9a4 5482  ..P...T.g.....T.
0x00710164  a4a4 5582 07a4 5482 9282 5f82 a3a4 5482  ..U...T..._...T.
0x00710174  63a2 5282 a5a4 5482 5269 6368 a4a4 5482  c.R...T.Rich..T.
0x00710184  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00710194  0000 0000 0000 0000 5045 0000 4c01 0400  ........PE..L...
[0x00409a16]> q
```

We list all the resources in the PE through the “iR” command. Then we print a hexdump (px) starting at the first byte of the resource with the name 1831 (0x727) which radare2 tells us starts at 0x7100a4.

Interestingly enough, we can see that **WannaCry contains another PE embedded as a resource** and this code is going to (at least) read it through the FindResource, LoadResource and LockResource.

The LockResource is the function call that gives the address (a pointer) to the first byte of the resource in memory.

```
push    eax                 ; hResData
call    ds:LockResource
cmp     eax, ebx
mov     [esp+270h+var_260], eax
jz      loc_407F08
```

This means that var_260 can be renamed to something like “embeddedBinary”.

The return value of the call to SizeOfResource is stored in ebp, which will be used later.

- Dropping the resource

After getting a pointer to the resource in memory and obtaining its size, the following basic block looks like this:

```
mov     ecx, 40h
xor     eax, eax
lea     edi, [esp+270h+var_207]
mov     [esp+270h+Dest], bl
rep stosd
stosw
stosb
mov     ecx, 40h
xor     eax, eax
lea     edi, [esp+270h+var_103]
mov     [esp+270h+NewFileName], bl
rep stosd
mov     esi, ds:sprintf
push    offset aTaskscheExe ; "tasksche.exe"
stosw
stosb
push    offset aWindows ; "WINDOWS"
lea     eax, [esp+278h+Dest]
push    offset aCSS     ; "C:\\%s\\%s"
push    eax             ; Dest
call    esi ; sprintf
add     esp, 10h
lea     ecx, [esp+270h+NewFileName]
push    offset aWindows ; "WINDOWS"
push    offset aCSQeriuwjhrf ; "C:\\%s\\qeriuwjhrf"
push    ecx             ; Dest
call    esi ; sprintf
add     esp, 0Ch
lea     edx, [esp+270h+NewFileName]
lea     eax, [esp+270h+Dest]
push    1               ; dwFlags
push    edx             ; lpNewFileName
push    eax             ; lpExistingFileName
call    ds:MoveFileExA
push    ebx             ; hTemplateFile
push    4               ; dwFlagsAndAttributes
push    2               ; dwCreationDisposition
push    ebx             ; lpSecurityAttributes
push    ebx             ; dwShareMode
lea     ecx, [esp+284h+Dest]
push    40000000h       ; dwDesiredAccess
push    ecx             ; lpFileName
call    CreateFileA_0
mov     esi, eax
cmp     esi, 0FFFFFFFFh
jz      loc_407F08
```

Let's start from the beginning:

```
mov     ecx, 40h
xor     eax, eax
lea     edi, [esp+270h+var_207]
mov     [esp+270h+Dest], bl
rep stosd
stosw
stosb
mov     ecx, 40h
xor     eax, eax
lea     edi, [esp+270h+var_103]
mov     [esp+270h+NewFileName], bl
rep stosd
```

The "stosd", "stosw", "stosb" instructions store the content of eax (dword), ax (word), al (byte) to the address given by EDI or DI. DI is incremented/decremented by 1.

As explained before, the "REP" will repeat the instruction that follows ecx times.

Let's analyze the following instructions:

```
.text:00407DB9          mov     ecx, 40h
.text:00407DBE          xor     eax, eax
.text:00407DC0          lea     edi, [esp+270h+var_207]
.text:00407DC4          mov     [esp+270h+Dest], bl
.text:00407DC8          rep stosd
.text:00407DCA          stosw
.text:00407DCC          stosb
```

1. ecx is set to 0x40 so the "rep" modifier will execute stosd 0x40 times.
2. eax is set to zero.
3. edi contains the address to var_207.
4. The mov instruction can be ignored for now, but it puts a null byte (ebx is 0) in Dest.
5. 0x40*32 null (eax is 0) bits are written to address var_207.
6. A null word is written at var_207+0x40*32
7. A null byte is written at var_207+0x40*32+16.

Basically, we've seen **how a memset(var_2017, 0, 0x103) is implemented.**

- Both var_207 and var_103 got initialized to 0.
- And Dest and NewFileName are "cleared out" by putting a null byte in the beginning of the buffers.

The code that follows is:

```
mov     esi, ds:sprintf
push    offset aTaskscheExe ; "tasksche.exe"
stosw
stosb
push    offset aWindows ; "WINDOWS"
lea     eax, [esp+278h+Dest]
push    offset aCSS      ; "C:\\%s\\%s"
push    eax              ; Dest
call    esi ; sprintf
```

We have already seen this code construct.
The “C:\\WINDOWS\\tasksche.exe” is stored in the “Dest” buffer.

We can rename Dest by taskcheBinaryPath.

And the same happens with the code that follows:

```
lea     ecx, [esp+270h+NewFileName]
push    offset aWindows ; "WINDOWS"
push    offset aCSQeriuwjhrf ; "C:\\%s\\qeriuwjhrf"
push    ecx              ; Dest
call    esi ; sprintf
```

The NewFileName buffer is set to “C:\\WINDOWS\\qeriuwjhrf”.

Given that we don’t know what this artifact will be, we can leave the NewFileName variable name as it is, given that is already quite meaningful.

- Moving files around

After preparing the previous file paths, we find the following code:

```
lea     edx, [esp+270h+NewFileName]
lea     eax, [esp+270h+taskcheBinaryPath]
push    1                ; dwFlags
push    edx              ; lpNewFileName
push    eax              ; lpExistingFileName
call    ds:MoveFileExA
push    ebx              ; hTemplateFile
push    4                ; dwFlagsAndAttributes
push    2                ; dwCreationDisposition
push    ebx              ; lpSecurityAttributes
push    ebx              ; dwShareMode
lea     ecx, [esp+284h+taskcheBinaryPath]
push    40000000h        ; dwDesiredAccess
push    ecx              ; lpFileName
call    CreateFileA_0
mov     esi, eax
cmp     esi, 0FFFFFFFFh
jz      loc_407F08

mov     eax, [esp+270h+embeddedBinary]
lea     edx, [esp+270h+embeddedBinary]
push    ebx              ; lpOverlapped
push    edx              ; lpNumberOfBytesWritten
push    ebp              ; nNumberOfBytesToWrite
                         ; ebp contains sizeOfResource
push    eax              ; lpBuffer
push    esi              ; hFile
call    WriteFile
push    esi              ; hObject
call    CloseHandle_0
```

First of all, MoveFileExA is called with the address of NewFileName in edx (which will be the lpNewFileName parameter) and the address of taskcheBinaryPath in eax (which will be the lpExistingFileName).

So the sample is moving the file “C:\\\\WINDOWS\\\\tasksche.exe” to “C:\\\\WINDOWS\\\\qeriuwjhrf”. The “tasksche.exe” file is not checked for its existence and the return code for MoveFileExA is not checked either.

After the call to MoveFileExA, a call to CreateFileA_0 is executed. CreateFileA_0 is the memory location where the address of the CreateFileA import was written. This call is just a required call before calling the next WriteFile function.

The WriteFile function receives ebp as the nNumberOfBytesToWrite. ebp contained the size of the resource. The contents that will be written to the file created with the CreateFileA_0 call are specified by the address to embeddedBinary, which contained the pointer in memory for the resource.

Basically, **what is happening is that the resource that was embedded in the PE is being written in C:\\\\WINDOWS\\\\tasksche.exe.**

- Finalizing the trick

After writing the data to the taskche.exe file, this code follows:

```
xor     ecx, ecx
xor     eax, eax
mov     [esp+270h+ProcessInformation.hThread], ecx
lea     edi, [esp+270h+StartupInfo.lpReserved]
mov     [esp+270h+ProcessInformation.dwProcessId], ecx
lea     edx, [esp+270h+taskcheBinaryPath]
mov     [esp+270h+ProcessInformation.dwThreadId], ecx
mov     ecx, 10h
rep stosd
mov     edi, offset off_431340
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     [esp+270h+ProcessInformation.hProcess], ebx
mov     esi, edi
mov     ebp, ecx
mov     edi, edx
or      ecx, 0FFFFFFFFh
repne scasb
mov     ecx, ebp
dec     edi
shr     ecx, 2
rep movsd
mov     ecx, ebp
lea     eax, [esp+270h+ProcessInformation]
and     ecx, 3
push    eax             ; lpProcessInformation
rep movsb
lea     ecx, [esp+274h+StartupInfo]
lea     edx, [esp+274h+taskcheBinaryPath]
push    ecx             ; lpStartupInfo
push    ebx             ; lpCurrentDirectory
push    ebx             ; lpEnvironment
push    8000000h        ; dwCreationFlags
push    ebx             ; bInheritHandles
push    ebx             ; lpThreadAttributes
push    ebx             ; lpProcessAttributes
push    edx             ; lpCommandLine
push    ebx             ; lpApplicationName
mov     [esp+298h+StartupInfo.cb], 44h
mov     [esp+298h+StartupInfo.wShowWindow], bx
mov     [esp+298h+StartupInfo.dwFlags], 81h
call    CreateProcessA
```

There’s some mambo jambo here:

```
xor     ecx, ecx
xor     eax, eax
mov     [esp+270h+ProcessInformation.hThread], ecx
lea     edi, [esp+270h+StartupInfo.lpReserved]
mov     [esp+270h+ProcessInformation.dwProcessId], ecx
lea     edx, [esp+270h+taskcheBinaryPath]
mov     [esp+270h+ProcessInformation.dwThreadId], ecx
mov     ecx, 10h
rep stosd
mov     edi, offset off_431340
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     [esp+270h+ProcessInformation.hProcess], ebx
mov     esi, edi
mov     ebp, ecx
mov     edi, edx
or      ecx, 0FFFFFFFFh
repne scasb
mov     ecx, ebp
dec     edi
shr     ecx, 2
rep movsd
mov     ecx, ebp
lea     eax, [esp+270h+ProcessInformation]
and     ecx, 3
push    eax             ; lpProcessInformation
rep movsb
```

Basically, some structures are initialized. You can go through it knowing the following bits of information:

- strlen()

The “repne scasb” is used to implement the strlen function.

- memset()

The “rep stosd” is used to write eax value where edi points, ecx times (incrementing edi per one byte in each iteration).

- strcpy()

The “rep movsd” copies dwords from esi to edi, ecx times (incrementing esi and edi per one byte in each iteration).

What really matters is what happens next, which is the last thing that happens!

```
lea     ecx, [esp+274h+StartupInfo]
lea     edx, [esp+274h+taskcheBinaryPath]
push    ecx             ; lpStartupInfo
push    ebx             ; lpCurrentDirectory
push    ebx             ; lpEnvironment
push    8000000h        ; dwCreationFlags
push    ebx             ; bInheritHandles
push    ebx             ; lpThreadAttributes
push    ebx             ; lpProcessAttributes
push    edx             ; lpCommandLine
push    ebx             ; lpApplicationName
mov     [esp+298h+StartupInfo.cb], 44h
mov     [esp+298h+StartupInfo.wShowWindow], bx
mov     [esp+298h+StartupInfo.dwFlags], 81h
call    CreateProcessA
```

CreateProcessA is a function to execute binaries. What matters most here is the lpCommandLine parameter. This parameter is set in edx and edx contains an address pointing to tascheBinaryPath. The tascheBinaryPath contained “C:\\WINDOWS\\tasksche.exe”.

Basically, the resource embedded in inside the PE, after being dropped to the filesystem it got executed.

Analyze the “main” function

First of all we need to identify what’s the main function.

Finding “main”

From the strings, we discovered the function at 0x407C40 which we named serviceCreation. We know for sure that this is user code, so let’s backtrack until we find the main function.

By putting the cursor in the “serviceCreation” string (function name) and pressing “x” we find that there is only one cross reference at 0x407F20 from a function that the only thing it does is call “serviceCreation” and another function.

- Is 0x407F20 our main?

Let’s find the cross reference to this function and if it looks like user code, this won’t be our main.

Again, there is only one cross reference at the 0x408090 function.

- Is 0x408090 our main?

Even if this function mentions argc, it still looks like user code (OpenSCManagerA, CloseServiceHandle, GetModuleFileName, StartServiceCtrlDispatcherA…).

Let’s find the cross reference to this function and if it looks like user code, this won’t be our main.

Again, there is only one cross reference at the 0x408140 function.

- Is 0x408140 our main?

This is still user code (InternetOpenA and working with user strings).

Let’s find the cross reference to this function and if it looks like user code, this won’t be our main.

Again, there is only one cross reference at the 0x409A16 function.

- Is 0x409A16 our main?

No, it’s not!

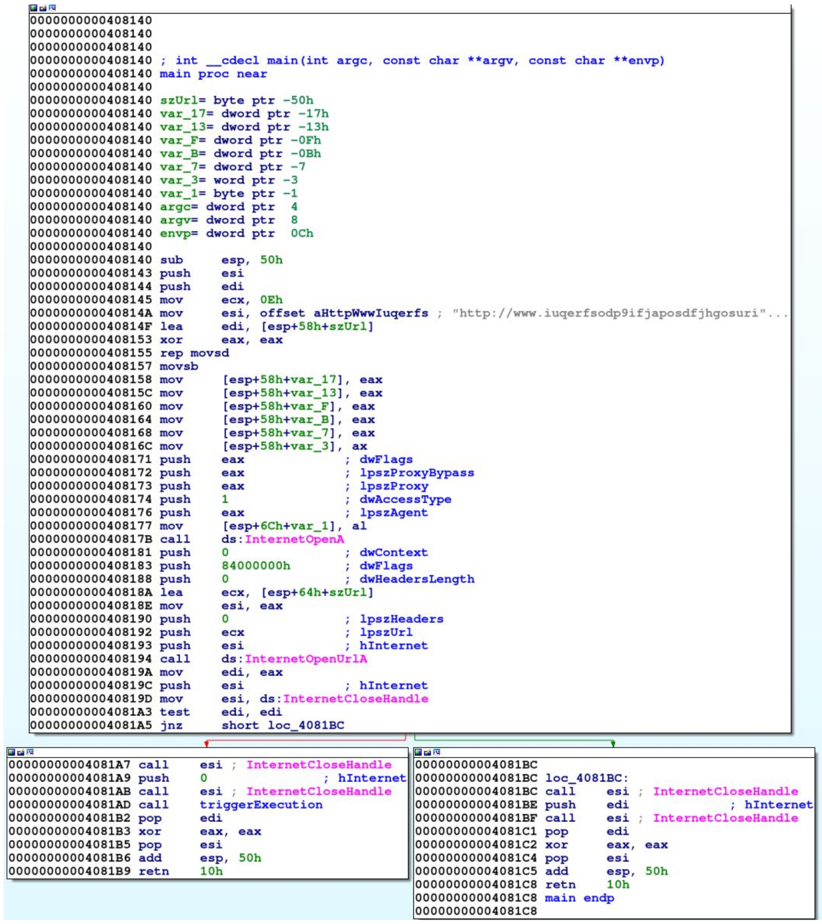
This is actually the “start” function as IDA Pro recognizes it. This means that this function is not written by a developer, making our previously analyzed function the first function written by a developer, hence our main function!

Even if IDA Pro didn’t appropriately named this function as “start”, we could have inferred that the previous function we analyzed was the main function because before being called, calls to GetModuleHandleA and GetStartupInfoA functions are executed. This is a common pattern before calling main. After “main” is called, we have an “exit” call.

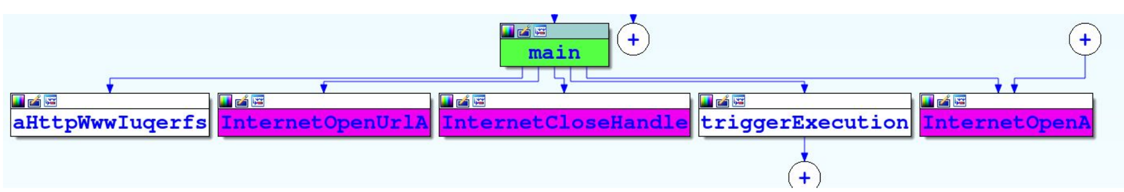
The main function is at 0x408140.

Analyzing “main”

The main function looks like this:



It only has three basic blocks. The “proximity browser” (right click on a basic block and select “Proximity Browser”) shows us the most important artifacts for “main”:



We have:

- A string
- Call to InternetOpenUrlA
- InternetCloseHandle
- User function

- InternetOpenA

The function starts with:

```
mov     ecx, 0Eh
mov     esi, offset aHttpWwwIuqerfs ; "http://www
lea     edi, [esp+58h+szUrl]
xor     eax, eax
rep movsd
movsb
```

We see that it works with the string “aHttpWwwIuqerfs” which is:

- http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com

To explain “rep movsd” instruction we need to divide the explanation in two:

1. “rep” will execute ecx (in this case 0xE) times the instruction that follows (in this case movsd).
2. “movsd” moves 32 bits (d) from the address pointed by “esi” to the address pointed by “edi”. Then both esi and edi get incremented by 32 bits.

Therefore, this code:

1. Moves 0xE to ecx to prepare for the rep movsd
2. Points esi to the beginning of the URL string to prepare for rep movsd
3. Points edi to the beginning of the local variable szUrl buffer.
4. xor eax, eax can be ignored right now
5. It executes rep movsd which will copy 0xE*32 bits = 448 bits = 56 bytes to szUrl. Just in case you didn’t notice, the URL string is exactly 56 bytes.
6. It executes movsb, which will move one extra byte to the end of the szUrl buffer. This byte will be null given that this is how the URL string ends.

Basically, we’ve seen how a simple strcpy() is implemented.

Then all local variables are set to 0 given that eax is 0 from the previous xor eax, eax.

```
mov     [esp+58h+var_17], eax
mov     [esp+58h+var_13], eax
mov     [esp+58h+var_F], eax
mov     [esp+58h+var_B], eax
mov     [esp+58h+var_7], eax
mov     [esp+58h+var_3], ax
```

The InternetOpenA and InternetOpenUrl are called:

```
push    eax                 ; dwFlags
push    eax                 ; lpszProxyBypass
push    eax                 ; lpszProxy
push    1                   ; dwAccessType
push    eax                 ; lpszAgent
mov     [esp+6Ch+var_1], al
call    ds:InternetOpenA
push    0                   ; dwContext
push    84000000h           ; dwFlags
push    0                   ; dwHeadersLength
lea     ecx, [esp+64h+szUrl]
mov     esi, eax
push    0                   ; lpszHeaders
push    ecx                 ; lpszUrl
push    esi                 ; hInternet
call    ds:InternetOpenUrlA
```

InternetOpenA only “Initializes an application's use of the WinINet functions”, however, InternetOpenUrlA “Opens a resource specified by a complete FTP or HTTP URL”.

The address used in InternetOpenUrlA is stored in ecx, which comes from:

- lea ecx, [esp+64h+szUrl]

In ecx we have the address of szUrl, the local variable containing the URL.

InternetOpenUrlA is the last call in this basic block. Its return value (in eax) is stored in “edi”.

```
0000000000408194 call    ds:InternetOpenUrlA
000000000040819A mov     edi, eax
000000000040819C push    esi             ; hInternet
000000000040819D mov     esi, ds:InternetCloseHandle
00000000004081A3 test    edi, edi
00000000004081A5 jnz     short loc_4081BC
```

```
00000000004081A7 call    esi ; InternetCloseHandle
00000000004081A9 push    0               ; hInternet
00000000004081AB call    esi ; InternetCloseHandle
00000000004081AD call    triggerExecution
00000000004081B2 pop     edi
00000000004081B3 xor     eax, eax
00000000004081B5 pop     esi
00000000004081B6 add     esp, 50h
00000000004081B9 retn    10h
```

```
00000000004081BC
00000000004081BC loc_4081BC:
00000000004081BC call    esi ; InternetCloseHandle
00000000004081BE push    edi             ; hInternet
00000000004081BF call    esi ; InternetCloseHandle
00000000004081C1 pop     edi
00000000004081C2 xor     eax, eax
00000000004081C4 pop     esi
00000000004081C5 add     esp, 50h
00000000004081C8 retn    10h
00000000004081C8 main endp
```

After storing the InternetOpenUrlA return value in edi, the next time edi gets used is for the conditional jump:

- test edi, edi
- jnz loc_4081BC

If edi is not zero, we will jump to loc_4081BC (right basic block), which will only call InternetCloseHandle and finish the program execution.

- Is edi zero?

InternetOpenURL “returns a valid handle to the URL if the connection is successfully established, or NULL if the connection fails”.

This means that edi will only be zero if the connection to the given URL failed.

- The execution of the program will finish if edi is not zero
- The execution of the program will finish if that URL can be reached
- The execution of WannaCry will finish if http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com is online.

This is what is known as the WannaCry killswitch.

The WannaCry killswitch was registered by MalwareTech.