WannaCry Malware Analysis - How YOU Could have Saved the World

Published: 2026-04-19 · Archived: 2026-05-06 02:00:47 UTC

Some details

Everyone knows and has heard about the WannaCry ransomware. One of the biggests and most well known attacks to ever happen in cybersecurity, the range was so big that even people outside of the cybersecurity community knew about it.

WannaCry? :'(

If somehow you were the first analyst to stumble upon WannaCry. You could have stopped this attack with just a little bit of experience on Reverse Engineering/Malware Analysis.

After looking for a while for the "original" WannaCry file I found this sample on MalwareBazaar. I verified some of the strings and function calls to make sure they match with this article that does some analysis on WannaCry. This same article was mentioned by **MalwareTech** himself (**Marcus Hutchins**) on his now famous blog entry, in case you don't know who he is, he is the guy who stopped WannaCry.

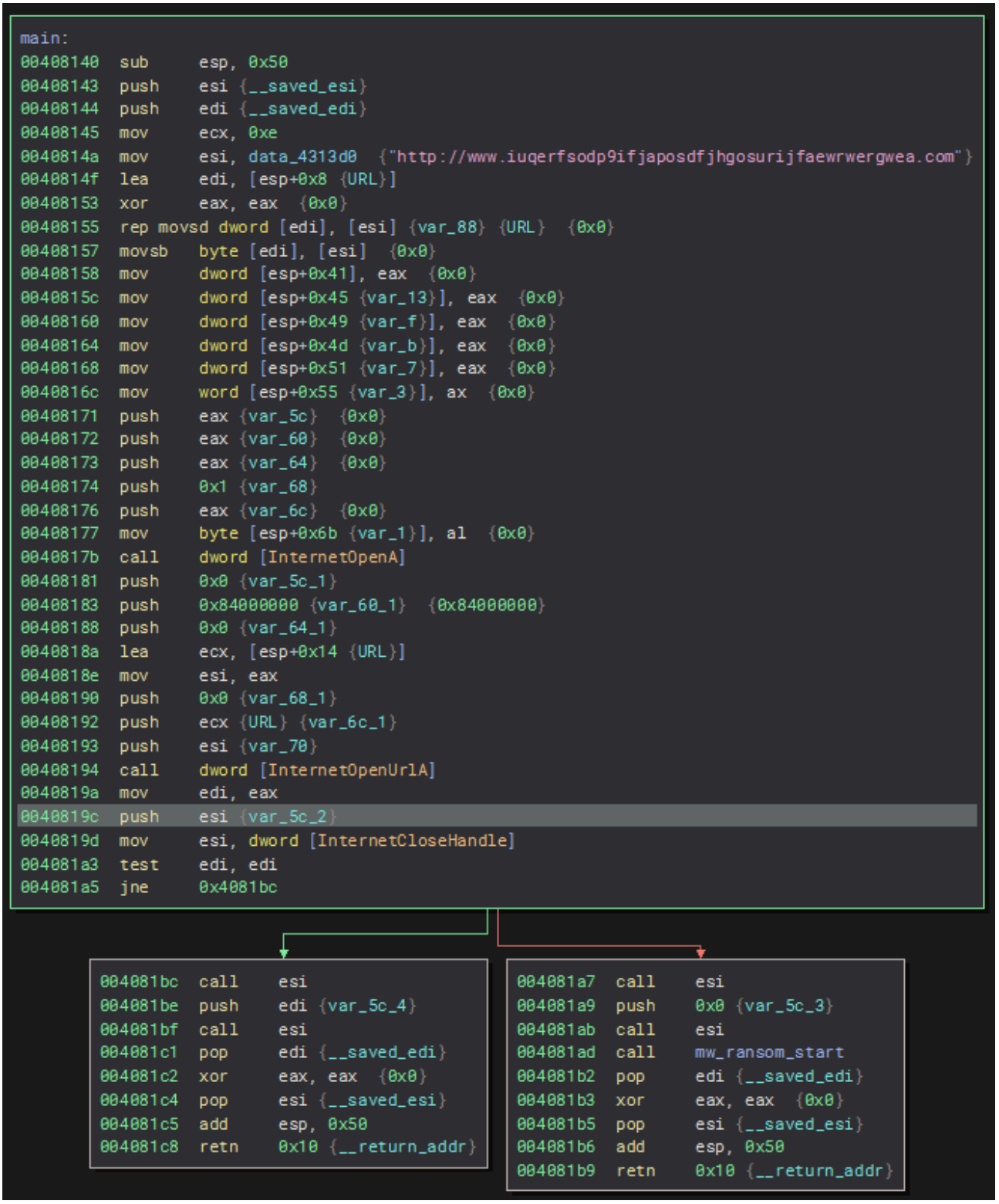
```
SHA256 hash: 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c
SHA1 hash: e889544aff85ffaf8b0d0da705105dee7c97fe26
MD5 hash: db349b97c37d22f5ea1d1841e3c89eb4
humanhash: berlin-angel-beer-quebec
File name: invoice_greenanimals.pdf.exe
```

It was a little bit hard to find the original file due to the big amount of variants that came after this attack, multiple variants were distributed using a different domain as a killswitch and after those were useless they deleted the killswitch from the file.

After opening the file using Binary Ninja you will be located at `0x409a16` or `_start()`. This is just a normal looking start of a C++ file. This function just tries to initialize the program correctly in a Windows environment.

```
00409b09        while (true)
00409b09        {
00409b09            i = *(uint8_t*)esi;
00409b09
00409b11            if (!i || i > 0x20)
00409b11            {
00409b13                STARTUPINFOA startupInfo;
00409b13                startupInfo.dwFlags = 0;
00409b1a                GetStartupInfoA(&startupInfo);
00409b24                uint32_t wShowWindow;
00409b24
00409b24                wShowWindow = !(startupInfo.dwFlags & 1) ? 0xa
00409b24                    : (uint32_t)startupInfo.wShowWindow;
00409b24
00409b3a                uint32_t wShowWindow_1 = wShowWindow;
00409b3b                char* var_90 = esi;
00409b3c                int32_t var_94_1 = 0;
00409b44                HMODULE var_98_1 = GetModuleHandleA(nullptr);
00409b45                main();
00409b4a                int32_t var_6c = 0;
00409b4e                exit(0);
00409b4e                /* no return */
00409b11            }
00409b11
00409b05        label_409b05:
00409b05            esi = &esi[1];
00409b06            char* var_78_2 = esi;
00409b09        }
00409a16    }
```

But we can also see at the very end of the a function that I renamed to `main()`, this is the main logic behind the Wannacry ransomware. This function is located at address `0x408140`.



Now, if you don’t have any experience on reverse engineering you might think this is something hard or impossible. Even if you have some experience doing programming, assembly code looks really weird. It doesn’t matter if you don’t understand assembly or even if you have very little experience in programming, you can understand the graph in the photo. But first let’s examine this on detail.

```
0040814a  mov      esi, data_4313d0  {"http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com"}
0040814f  lea      edi, [esp+0x8 {URL}]
00408153  xor      eax, eax  {0x0}
00408155  rep movsd dword [edi], [esi] {var_88} {URL}  {0x0}
00408157  movsb    byte [edi], [esi]  {0x0}
00408158  mov      dword [esp+0x41], eax  {0x0}
0040815c  mov      dword [esp+0x45 {var_13}], eax  {0x0}
00408160  mov      dword [esp+0x49 {var_f}], eax  {0x0}
00408164  mov      dword [esp+0x4d {var_b}], eax  {0x0}
00408168  mov      dword [esp+0x51 {var_7}], eax  {0x0}
0040816c  mov      word [esp+0x55 {var_3}], ax  {0x0}
```

This first instructions is just used to copy the URL `hXXp://www[.]iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com` into a local variable that will be used later on the function.

InternetOpenA - Get those Protocols

The next set of instructions are just a call to a the function `InternetOpenA`. This is an API Windows function that can be used by multiple programs. The `push` instruction is used to send the parameters that the function needs to work, here we have 5 `push` instructions, which means we have 5 parameters.

```
00408171  push     eax {var_5c}  {0x0}
00408172  push     eax {var_60}  {0x0}
00408173  push     eax {var_64}  {0x0}
00408174  push     0x1 {var_68}
00408176  push     eax {var_6c}  {0x0}
00408177  mov      byte [esp+0x6b {var_1}], al  {0x0}
0040817b  call     dword [InternetOpenA]
```


```
1
2
3
4
5
```

```
00408171  push    eax {var_5c}  {0x0}
00408172  push    eax {var_60}  {0x0}
00408173  push    eax {var_64}  {0x0}
00408174  push    0x1 {var_68}
00408176  push    eax {var_6c}  {0x0}
```


If we search for the function InternetOpenA we can find the official Microsoft documentation for the function.


```
1
2
3
4
5
```

```
HINTERNET InternetOpenA(
  [in] LPCSTR lpszAgent,
  [in] DWORD  dwAccessType,
  [in] LPCSTR lpszProxy,
  [in] LPCSTR lpszProxyBypass,
```

```
6
7
```

```
  [in] DWORD  dwFlags
);
```

In that same page we can find what each parameter is used for, and a summary of what the function does. From the Microsoft documentation page:

> Initializes an application's use of the WinINet functions.

If we search what `WinINet` means we can find the following explanation, I got this from the official Microsoft documentation:

> The Windows Internet (WinINet) application programming interface (API) enables your application to interact with FTP and HTTP protocols to access Internet resources. As standards evolve, these functions handle the changes in underlying protocols, enabling them to maintain consistent behavior.

After enabling the interaction with certain protocols to access Internet resources, the next step is to use that new resource to connect to the previous URL we saw.

InternetOpenUrlA - Is this thing on?

The next function is InternetOpenUrlA.

```
00408181  push    0x0 {var_5c_1}
00408183  push    0x84000000 {var_60_1}  {0x84000000}
00408188  push    0x0 {var_64_1}
0040818a  lea     ecx, [esp+0x14 {URL}]
0040818e  mov     esi, eax
00408190  push    0x0 {var_68_1}
00408192  push    ecx {URL} {var_6c_1}
00408193  push    esi {var_70}
00408194  call    dword [InternetOpenUrlA]
```

In the documentation we can see the following:

> Opens a resource specified by a complete FTP or HTTP URL.

```
1
2
3
4
5
6
7
8
```

```
HINTERNET InternetOpenUrlA(
[in] HINTERNET hInternet,
[in] LPCSTR    lpszUrl,
[in] LPCSTR    lpszHeaders,
[in] DWORD     dwHeadersLength,
[in] DWORD     dwFlags,
[in] DWORD_PTR dwContext
);
```

In the screenshot we saw previously we saw some interesting parameters being used to call the function.

First we see these instructions (I skipped a few instructions that were not relevant):

```
call    dword [InternetOpenA]
mov     esi, eax
...
push    esi
call    dword [InternetOpenUrlA]
```

We have to know that after we call a function the return value or the result of that action is stored in `eax`. So after a `call` instruction the return value is in `eax`, that's why the next instruction `mov esi, eax` is used. After we copy that to `esi` it is pushed to be used by `call dword [InternetOpenUrlA]` as a parameter.

1. Call `InternetOpenA`
2. Create a way for the program to have connection capabilities
3. Use return value as a parameter to call `InternetOpenUrlA`

We also can see another `push` instruction where it uses the value `0x84000000`. This is a flag value, these are values that are used to turn on options. In this case we can search for the API Flag documentation for `Wininet.h`.

From that website I got this:

> INTERNET_FLAG_RELOAD 0x80000000 Forces a download of the requested file, object, or directory listing from the origin server, not from the cache. The FtpFindFirstFile, FtpGetFile, FtpOpenFile, FtpPutFile, HttpOpenRequest, and InternetOpenUrl functions use this flag.

> INTERNET_FLAG_NO_CACHE_WRITE 0x04000000 Does not add the returned entity to the cache. This flag is used by , HttpOpenRequest, and InternetOpenUrl.

I can just add these values and it gives me `0x84000000`. This means that it uses both of these functions, you can sometimes watch these options like this:

```
INTERNET_FLAG_RELOAD | INTERNET_FLAG_NO_CACHE_WRITE
```

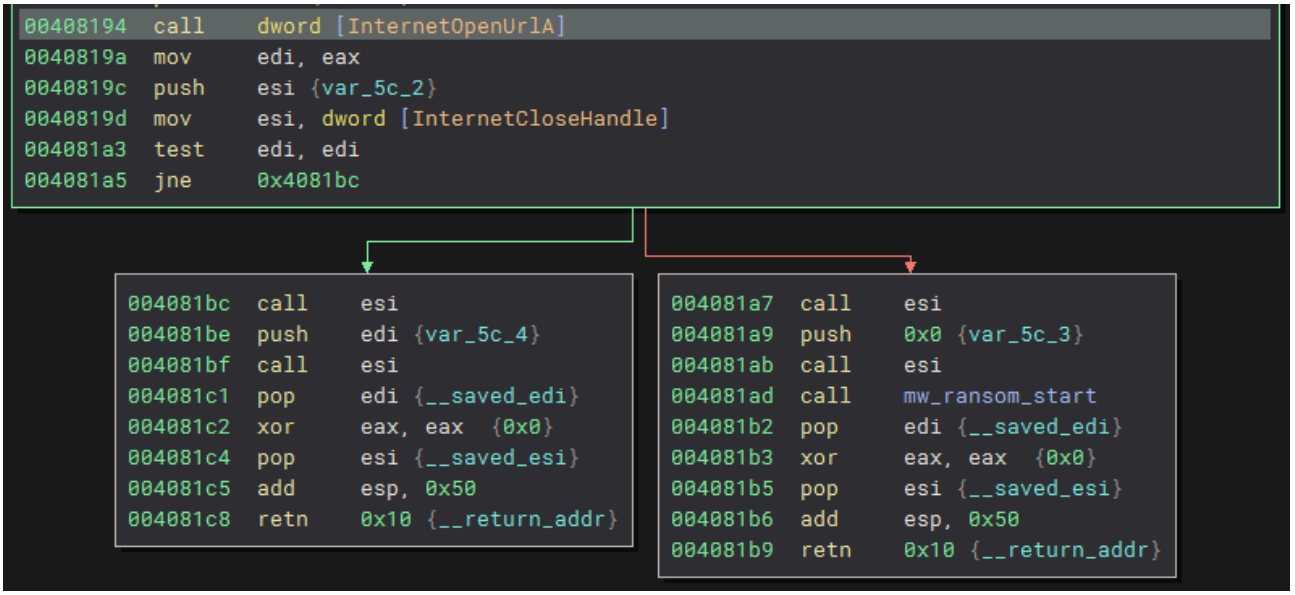
This means that it will always get the newest version of the website and it will not save the cache of that website to make sure the information is always up to date.

Summary

This just creates a connection with Windows API to make sure the malware can use Internet capabilities of the system and then tries to access the domain `hXXp://www[.]iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea[.]com`. Depending on the result of these requests the malware is going to choose the next instructions that will be executed.

The branches

SOo0oOo no website?



As I mentioned before this is were the most important decision is taken. What happens after we call the function `InternetOpenUrlA` to connect to the website?

Based on the official Microsoft documentation:

> Returns a valid handle to the URL if the connection is successfully established, or NULL if the connection fails.

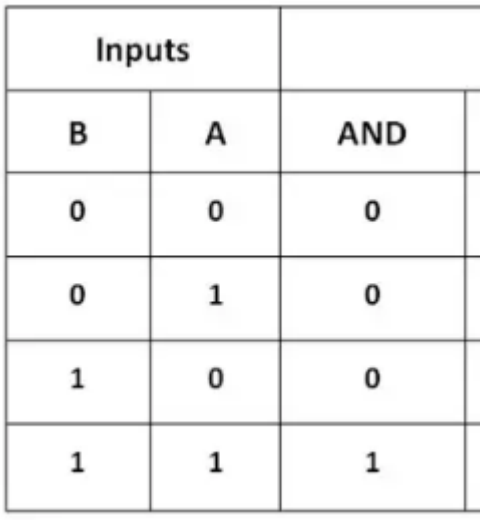
`NULL` is just a representation of the absense of a value or a reference. This is not the same as zero. In assembly language `NULL` is represented by the value zero `0x0`.

So, if the function call to `InternetOpenUrlA` is not able to successfully establish a connection to the domain it will return `NULL` or `0x0`.

Now that value is stored in `eax`. We use a `mov` to store the value on the register `edi`. Then it does a comparison with `test`.

```
call    dword [InternetOpenUrlA]
mov     edi, eax
...
test    edi, edi
```

The `test` instruction is an `AND` operation. It takes the value `0x0` and then it uses this little table to get the result. If we do an `0x0 AND 0x0` it will result on `0x0`.



Inputs		
B	A	AND
0	0	0
0	1	0
1	0	0
1	1	1

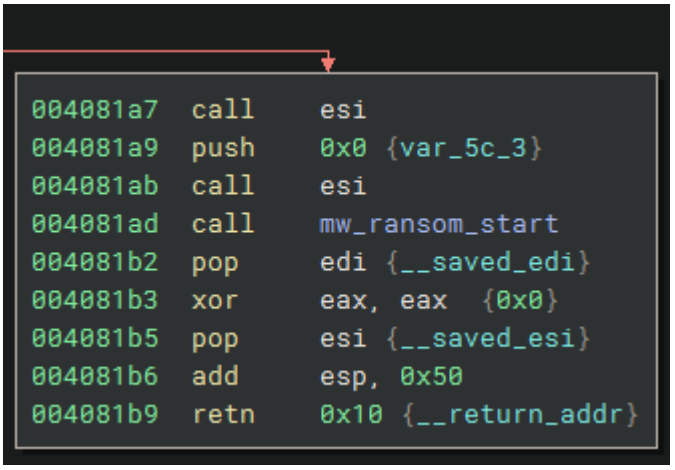
`test` affects the `Zero Flag (ZF)`. If the result of the operation is zero, then it sets the flag to `ZF = 1`. The instruction `JNE` (Jump if not Equal) and `JNZ` (Jump if not Zero) are the alternate representations of the same instruction.

- `ZF = 1` (**Set**): The result of the operation was zero.
- `ZF = 0` (**Cleared**): The result of the operation was non-zero.

Both `JNE` and `JNZ` execute the same machine code operation, jumping only when the `Zero Flag` is equal to `1`.

```
test  edi, edi    ; NULL AND NULL = 0, sets ZF=1
jne   0x4081bc    ; jump ONLY if ZF=0 (not zero)
```

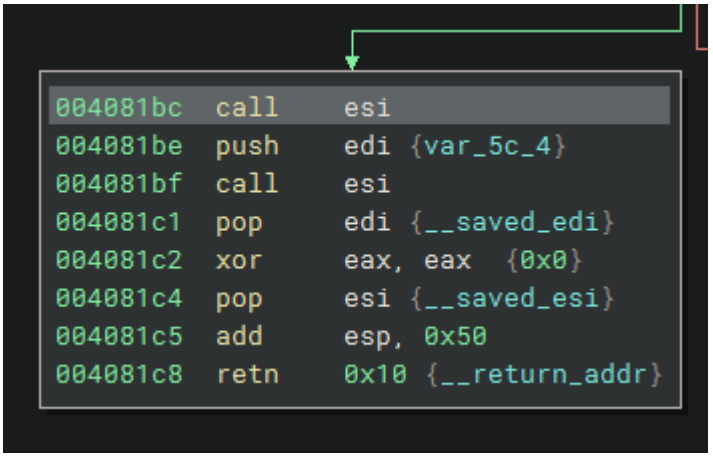
If we are not able to connect to the domain then the flag is `ZF=1`, meaning we will jump to this block:



It's a very short block of code, it makes some calls to `esi`, at this time `esi` is holding the function `InternetCloseHandle`. It closes the connection and then calls a function I renamed to `mw_ransom_start`, this is pretty self explanatory, I already did a quick analysis on that specific function to know what it does, but based on the call to another function we can induce that this is most likely the infection function that will trigger the ransomware.

It's alive!!!

If the connection to the domain is succesful then the value will be a non zero value, if we use the table mentioned before the result of the `AND` operation will be a non zero value. Meaning that it will jump to `0x4081bc`.

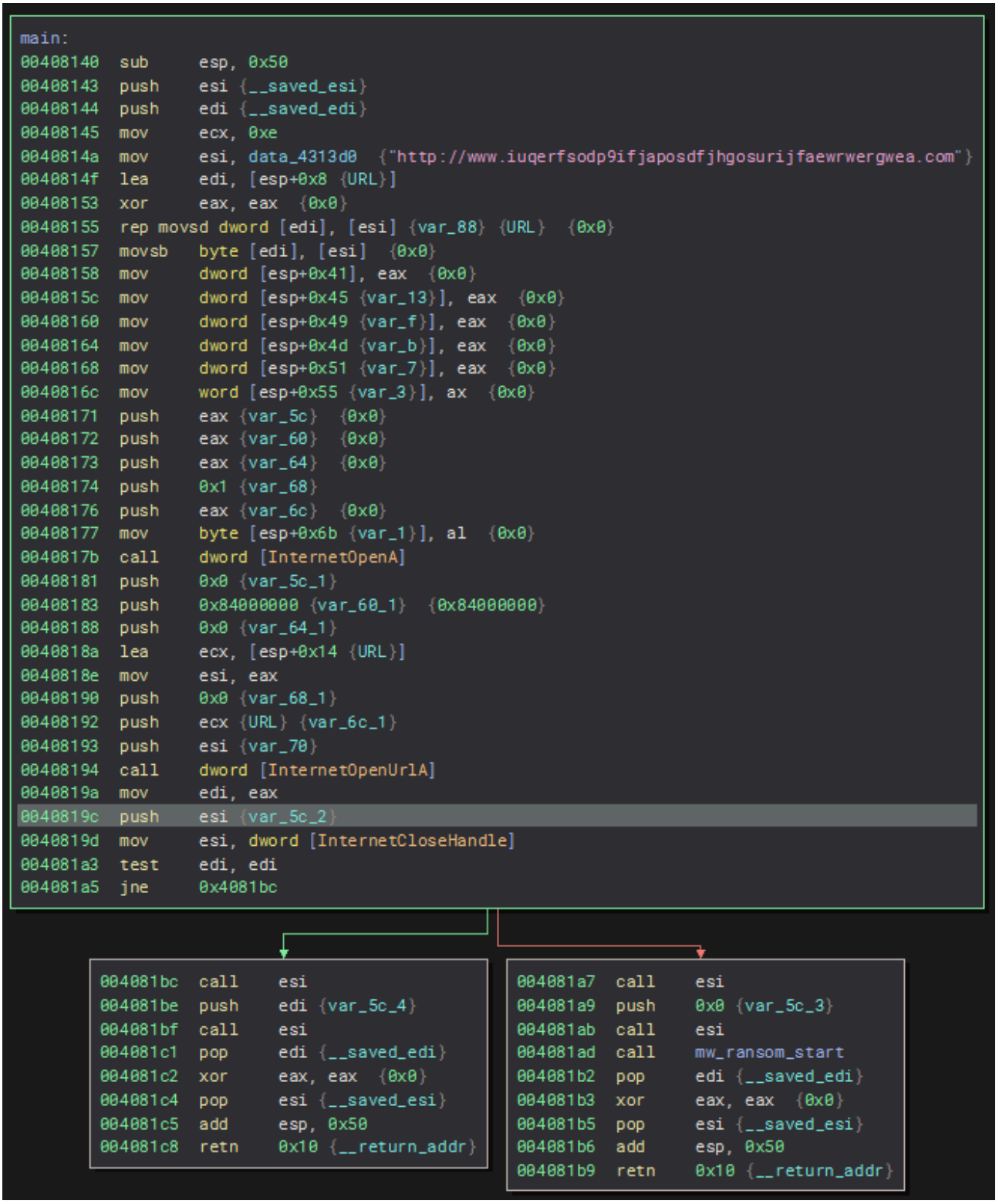


These instructions are pretty simple as well, it just calls the same function that is stored in `esi` which is once again `InternetCloseHandle`. The whole block of code is made up of clean up instructions and doesn't do anything meaningful and then it returns, which in other words means that the program closes itself.

Thanks to this analysis we know that the result of a connection to a certain domain will determine if the malware closes itself without doing anything harmful to the victim's computer.

Just watch the little graph

Is there a faster and easier way to get the same result? Yes, just watch the graph, you can notice easily that there are two different branches, meaning that a decision will be taken.



1. See the branches from a decision at the very end of the function
2. Notice that the left branch just closes the program without any suspicious function calls
3. See the function call to `InternetOpenUrlA` and the suspicious URL
4. Look for the value `edi` that is used by `test`. This is just a simple 'if'
5. `mov edi, eax` after the `call` to `InternetOpenUrlA`
6. Notice that the execution of the malware depends on the result of the `InternetOpenUrlA`

Conclusion

This could have been done by you. You just had to watch the graph, notice the branch and just know what an `if` statement is, search for some API that are well documented and use logic to know that the URL is used to determine the execution of the malware.
