



**Abertay  
University**

# **Malware Analysis Report**

An investigation into the capabilities and behaviors of the  
WannaCry ransomware

**Sarah Gardiner**

CMP320: Advanced Ethical Hacking

BSc Ethical Hacking Year 3

2022/23

*Note that Information contained in this document is for educational purposes.*

# Abstract

---

Despite the best efforts of cybersecurity professionals, malware remains a persistent and ever-evolving threat to the hardware and software that runs many of the systems used in our daily lives. Because of this, analysis of said malware plays an important role in protecting these machines, and by unveiling the methods and tools used by malicious software we can prevent even more devices from becoming compromised.

The following report details the analysis of a malware sample, which was quickly unveiled to be from the WanaCry ransomware strain. The stages of the analysis consisted of three main parts; static analysis was performed to allow for the code and structure of the malware to be assessed without the malware ever being run. This was done by comparing the malware's hash with scanning results from other security vendors and identifying possible functionality from strings held within the malware, and analyzing the malware's metadata for any possible information that could be used later. During disassembly of the malware, a disassembler was used to show the x86 assembly code used to run the malware at a low level. This code was analyzed, and information first learned in the static analysis phase was used to help identify key functions and procedures. Finally, dynamic analysis involved running the malware and monitoring various operations and network transmissions that take place during its runtime.

The information found during the above steps allowed for an in-depth analysis of the malware's capabilities to execute, encrypt, and persist within an infected machine. It was found that the sample could encrypt an entire device's files within seconds and had the capability to rewrite the Windows registry itself to allow for persistent access to the device and its data.

# Contents

---

1	Introduction .....	1
1.1	Background .....	1
1.2	Aims.....	3
2	Methodology.....	4
2.1.1	Overview of Static Analysis .....	4
2.1.2	Disassembly.....	5
2.1.3	Dynamic Analysis.....	5
3	Procedure and Results .....	7
3.1	Overview of Procedure .....	7
3.2	Static Analysis.....	8
3.2.1	Signature Identification.....	8
3.2.2	String Analysis .....	9
3.2.3	Packer Detection .....	11
3.2.4	Library and Function Identification.....	12
3.2.5	Portable Executable Analysis .....	14
3.3	Disassembly.....	15
3.3.1	Identifying Windows API Calls.....	15
3.3.2	Identifying Executable Functionality.....	17
3.4	Dynamic Analysis .....	19
3.4.1	Executing the Malware .....	19
3.4.2	Process Monitoring .....	23
3.4.3	Registry Analysis.....	26
3.4.4	Network Analysis.....	27
4	Discussion.....	28
4.1	Overall Discussion .....	28
4.1.1	Evasion Capabilities.....	28
4.1.2	Persistence Capabilities .....	28
4.2	Countermeasures.....	29
4.3	Future Work .....	29
5	References .....	30
	Appendices.....	33

Appendix A..... 33

5.1.1 VirusTotal Results..... 33

5.1.2 Dependency Walker Results ..... 34

5.1.3 Functions Imported From Ntdll.dll..... 35

5.1.4 Functions Imported From Kernel32.dll ..... 48

# 1 INTRODUCTION

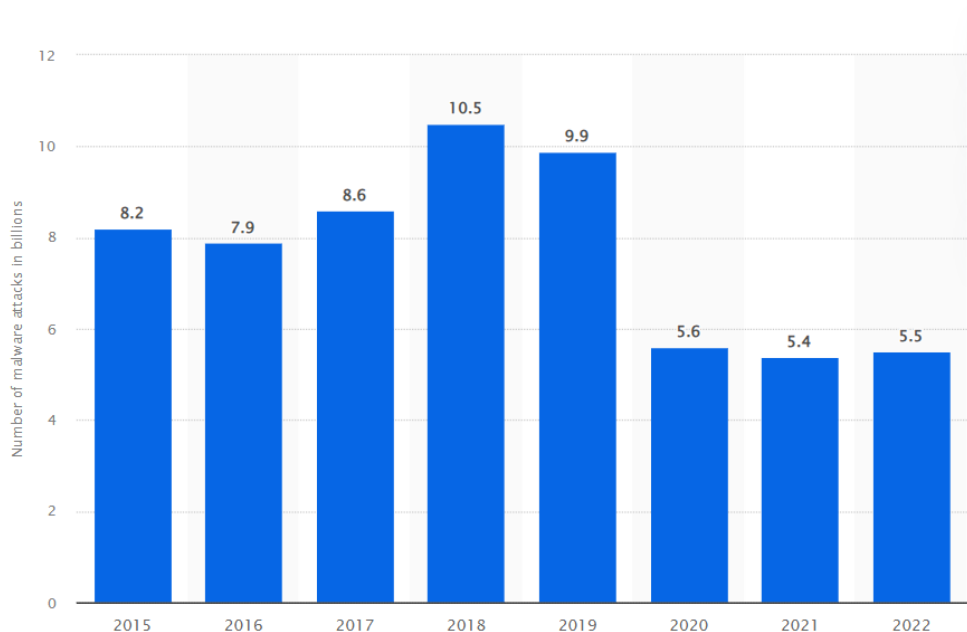
## 1.1 BACKGROUND

---

As the world turns to putting its data online, the world of traditional crime and robbery has shifted dramatically. No longer is it worth planning a bank heist, or intercepting a jewelry delivery, when the same amount of money or more can be made from the comfort of your own home.

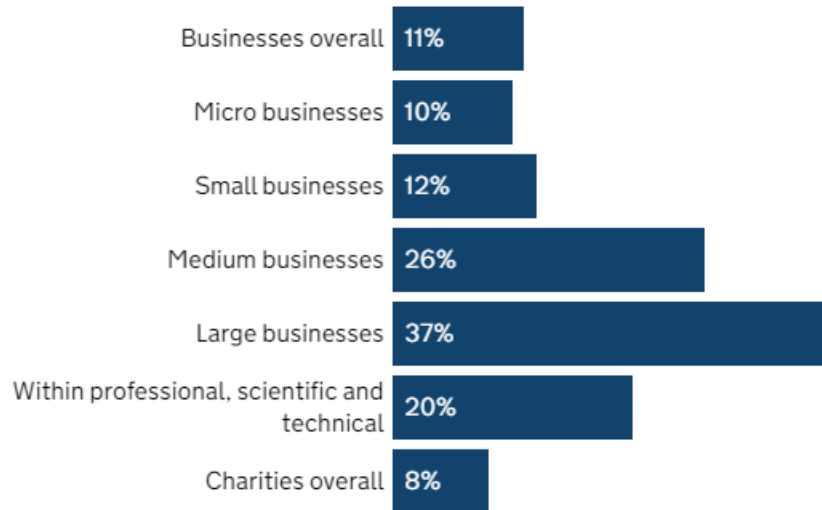
Malware has been a part of digital life for almost as long as digital life has been a conceivable concept. While the first inventors of malware were primarily computer scientists looking to push the boundaries of their own devices, criminals quickly took up the art of using software to hack computers and, powered by the invention of cryptocurrencies in the late 2000s, quickly took over the once idyllic and peaceful digital world.

Currently, the malware industry is massive, and hackers are managing to push the number of malware-based attacks into the billions. While trends over the past three years have gone down, the numbers are still incomprehensively massive, as Figure 1.1 shows.



*Figure 1.1: The number of malware attacks each year dropped in the 2020s, but remains significantly high. (Statista, 2023)*

While the number of attacks has decreased in recent years, malware's cost remains high for victims. As a survey by the UK government highlighted, the cost of cybercrime for businesses is, on average, £15300 (UK Government, 2023). With at least 11% of all UK businesses experiencing cyber crime between 2022 and 2023 (see Figure 1.2), it is clear that malware is prevalent within our society.



*Figure 1.2: The percentage of organizations that have experienced any cybercrime between 2022 and 2023 (UK Government, 2023).*

A major prevention for malware is discovering the techniques it uses to infect its victims. When antiviruses first became mainstream, many of them used signature-based detection to spot a potential virus (Milosevic, 2013). However, this method does not account for changes made to the malicious files, and as hashing algorithms used to generate typical file signatures are so sensitive, a threat actor would only need to change a few lines of code to successfully bypass these early detection systems.

To stop this, malware analysis is used to identify what processes, functions, and activities are created by malware. Current major antiviruses monitor almost all elements of a device and are constantly checking for malicious activity. Microsoft Defender, for example, constantly logs filesystem, login, network, and even kernel-level events for malicious activity (Paul Huijbregts, 2023). However, modern-day antiviruses still need regularly updated with the malicious processes that they need to look for, meaning that cybersecurity professionals need to spend time regularly analyzing the newest strains of malware available on the black market. Without malware analysis, almost all other cybersecurity protections would quickly be outdated.

## 1.2 AIMS

---

The aim of this report is to conduct a comprehensive analysis of a malware sample in a controlled environment to identify its behavior, key functions, and potential impact on the infected system by using tools that can outline key malicious activities at both low-level and high-level interactions with the device. To reach a conclusion on the malware's overall abilities, the following aims will be met:

- Identify interactions between the operating system and the malware.
- Outline key functions and procedures used to handle malicious activity.
- Use tools that can uncover all areas of malware's behaviors.
- Identify persistence and evasive methods in use.

## 2 METHODOLOGY

The procedure used to analyze the provided malware sample can be broken down into three main parts, static analysis, analysis of assembly code, and dynamic analysis. The methodology used for this report follows the one outlined in the Malware Reverse Engineering Handbook, written by the NATO Cooperative Cyber Defence Centre of Excellence.

### 2.1.1 Overview of Static Analysis

Static analysis allowed for the structure and function of the malware to be analyzed without allowing the executable to be run. Several steps were taken to allow for as much information about the sample to be gathered as possible. These steps are outlined below.

#### 2.1.1.1 *Signature Identification*

The first step in static analysis of the malware sample involved identifying if the malware's signature had been seen by any other antivirus scanners. The website VirusTotal allowed for the MD5 hash of the file to be uploaded, returning the results of any antivirus scanners that have previously detected that file to be malicious.

While VirusTotal allows for files themselves to be uploaded, it was decided to stick to uploading just a hash as the virtual machine used to analyze the malware was deliberately disconnected to the internet, and uploading the file itself would require connecting the machine to the internet, risking the malware being spread to the host device and other devices on the network.

As a scanning tool, VirusTotal was particularly powerful as it provides real-time updates on malware samples (VirusTotal, 2023). Therefore, the results from this stage are representative of what each antivirus vendor would analyze the sample to be at the time of writing, instead of what they may have identified the malware to be as when the sample was first detected.

To get the file's MD5 hash, the tool md5 Hash was used. This tool was chosen purely because it makes copying the file's hash a simple process when compared to other tools available on the device.

#### 2.1.1.2 *String Analysis*

Two main tools were used to analyze the strings of the malware sample: Strings, and 010 Editor. Strings allowed for the strings of the file to be gathered and analyzed, potentially showing any function names, imported DLLs, and executable files that the sample used.

Gathering strings at this stage would allow for cross-examination of other program outputs in later stages, such as within the dynamic analysis section. Additionally, it allowed for a general overview of the program's functionality.

#### 2.1.1.3 *Packer Detection*

Detecting if the malware analysis had been packed was a key part of the static analysis process. Packing, which is an obfuscation method, prevents the malware from being analyzed fully as the program is compressed to such an extent that typical static analysis methods are no longer effective (Michael Sikorski, 2012).



To detect if the program had been packed, the tool PEiD was used, as it would allow for the compiler or packer used to build the application.

#### **2.1.1.4 Library and Function Identification**

While string analysis provided some insight into the functions and libraries used by the sample, it was important to dive deeper into the DLLs imported during the program's runtime. This was done using Dependency Walker, which could list all dynamically linked functions. Doing so allowed for some of the most important parts of the program to be analyzed (Michael Sikorski, 2012), giving an extremely deep insight into the malware's full functionality.

Additionally, the DLLs would also show how the program intended on working with key areas of the underlying operating system, such as files, the registry, and the user.

#### **2.1.1.5 Portable Executable Analysis**

Analysis of portable executable (PE) headers would allow for the sample's metadata to be analyzed. To do this step, the program PE-Bear was used.

### **2.1.2 Disassembly**

The information gathered during the static analysis stage allowed for a basic overview of the sample's functionality, but a deeper investigation was needed to discover the inner workings of the malware (Michael Sikorski, 2012). To get a full picture and walk through the malware step-by-step, the tool IDA Free was used to disassemble the malware sample and reveal the underlying x86 assembly code, which was then analyzed to discover how functions imported from DLLs identified in previous analysis sections were used within the program.

### **2.1.3 Dynamic Analysis**

Dynamically analyzing the malware involved executing the sample and monitoring the processes taking place. Doing this allowed for information to be gathered that would be otherwise difficult to find through static analysis, such as the contents of files or the process used to execute the malicious payload.

One important factor of this stage is the virtual machine (VM) used to hold the malware. VMware Workstation was used to host a machine running Windows 10, with the network adaptor set to 'host-only', preventing the VM from communicating with any other devices on the network and therefore preventing the malware from spreading.

The separate stages of dynamic analysis are detailed below.

#### **2.1.3.1 Executing the malware**

The first step in dynamic analysis involved executing the sample and noting the visible changes a user would see while the malware runs. Malware functionality was tested and demonstrated on a text file, which allowed for the final outcome of the encryption process to be noted.

#### **2.1.3.2 Process Monitoring**

To monitor each process and operation created by the malware sample, the tool Process Monitor, or Procmon, was used. Procmon is an advanced monitoring tool for Windows that captures live data on events happening within the operating system as they happen (Michael Sikorski, 2012). As it is built and

shipped with Windows, it is highly specialised to detecting almost all events occurring within the OS, therefore revealing in full detail the steps taken by the malware as it runs.

#### **2.1.3.3 Registry Analysis**

Analysing the registry allowed for the discovery of changes made by the malware, which could indicate any persistence mechanisms used to allow the malware to stay hidden within the victimised device.

To do this stage, the tool Regshot was used to take a snapshot of the registry before and after the malware was run. Regshot allowed for a comparison between the two versions and allowed for identification of any changed registry keys or files.

#### **2.1.3.4 Network Analysis**

To analyse the malware's networking capabilities, the packet-capture tool Wireshark was used. This allowed for any attempted communication between the malware and other hosts on the network, or on the internet, to be captured and analysed for revealing clues on the malware's communication abilities.

# 3 PROCEDURE AND RESULTS

## 3.1 OVERVIEW OF PROCEDURE

---

This section of the report will show the process used and results found from the steps outlined within the methodology.

To note, the analysis of the malware sample was conducted on a Windows 10 (version 1909) virtual machine, hosted by VMWare Workstation Pro (version 16.2.4).

To complete the methodology, the following tools were used:

- **Md5 Hash** (version 6.1.7601.17514)
- **Strings** (unknown version)
- **VirusTotal** (Unknown version, accessed between April and May 2023)
- **010 Editor** (version 12.0.1)
- **PEiD** (version 0.95)
- **Dependency Walker** (version 2.2.6000)
- **PE-Bear** (version 0.6.1)
- **IDA Free** (version 7.6.210526)
- **Process Monitor (Procmon)** (version 3.92)
- **Regshot** (version 1.9.1)
- **Wireshark** (version 4.0.3)

## 3.2 STATIC ANALYSIS

### 3.2.1 Signature Identification

Using the program md5 Hash, the hash of the file was obtained (see Figure 3.1) and was used within VirusTotal to find if the file had been previously found by other security analysts and antiviruses. Doing so confirmed that the sample was malicious, and that 67 security vendors were able to detect it, as can be seen in Figure 3.2.

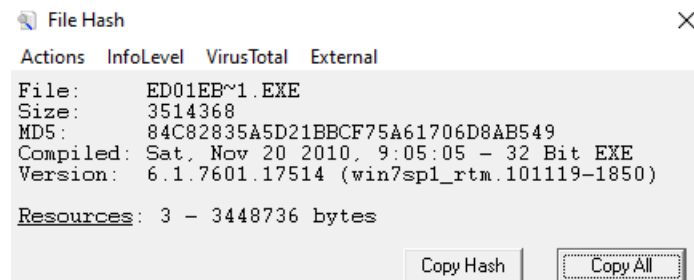


Figure 3.1: MD5 Hash showed the hash of the file, as well as the date it was compiled.

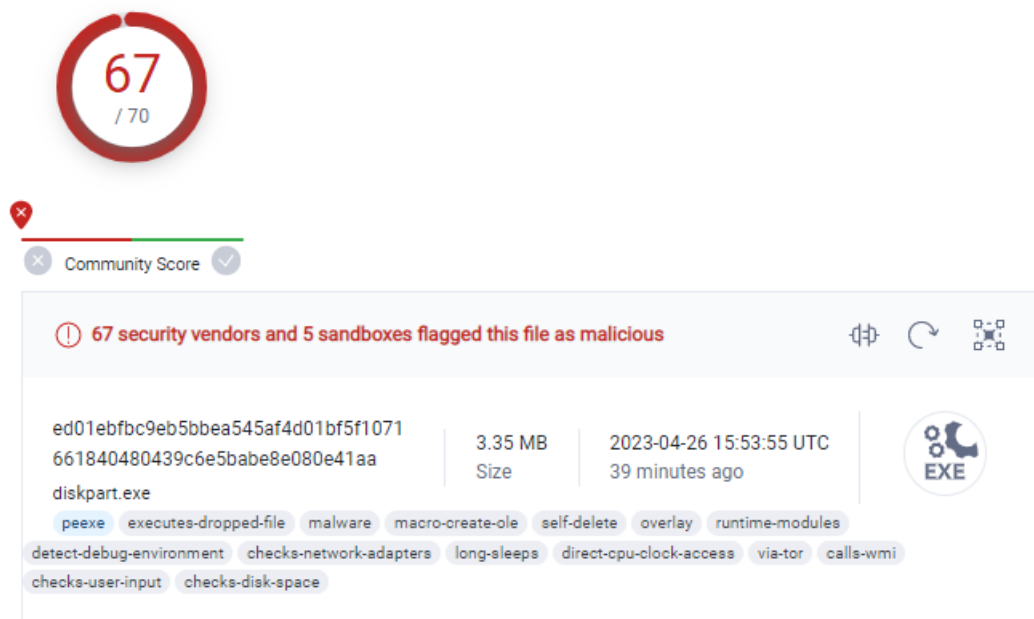


Figure 3.2: Analysis results from VirusTotal

Several of the security vendors that detected the sample identified it as being a ransomware, specifically a strain known as WannaCry, as can be seen in Figure 3.3. The full results from VirusTotal can be found in Appendix A, section 5.1.1.

AhnLab-V3	❗ Trojan.Win32.WannaCryptor.R200571	Alibaba	❗ Ransom:Win32/WannaCry.ali1020010
ALYac	❗ Trojan.Ransom.WannaCryptor	Antiy-AVL	❗ Trojan[Ransom]/Win32.Scatter
Arcabit	❗ Trojan.Ransom.WannaCryptor.A	Avast	❗ Win32:WanaCry-A [Trj]
AVG	❗ Win32:WanaCry-A [Trj]	Avira (no cloud)	❗ TR/Ransom.JB
Baidu	❗ Win32.Trojan.WannaCry.c	BitDefender	❗ Trojan.Ransom.WannaCryptor.A

Figure 3.3: Several of the vendors on VirusTotal identified the sample as being a strain of the ransomware WannaCry.

## 3.2.2 String Analysis

### 3.2.2.1 Hex

First, the hex of the program was analyzed using the tool 010 Editor. This was done to confirm the file format signature, which is represented in the ‘magic bytes’, meaning the first two to four bytes of the file (NetSPI, 2013). As Figure 3.4 shows, these magic bytes were ‘MZ’, meaning the file is a MS-DOS executable (Pietrek, 2002).

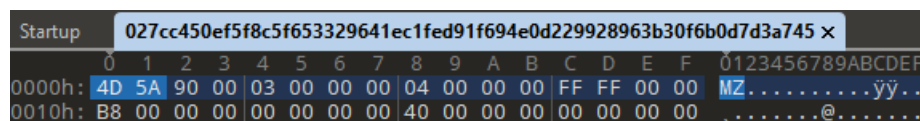


Figure 3.4: The magic bytes of the file were 'MZ', proving the file was a portable executable (PE) file.

### 3.2.2.2 Strings

The program Strings was used to extract string values from the sample, which could be used to provide an idea into the functionality of the program. Looking through String’s output, many function names could be seen that pointed to the potential functionality of the malware, including functions originating from Microsoft Enhanced RSA Cryptographic Provider (see Figure 3.5) and functions that indicated they were used for reading to, writing from, and deleting files (see Figure 3.6).

```
0000F08C Microsoft Enhanced RSA and AES Cryptographic Provider
0000F0C4 CryptGenKey
0000F0D0 CryptDecrypt
0000F0E0 CryptEncrypt
0000F0F0 CryptDestroyKey
0000F100 CryptImportKey
0000F110 CryptAcquireContextA
```

Figure 3.5: Strings showed that the sample used functions from Microsoft's Cryptographic Provider.

```

0000D884 SetCurrentDirectoryA
0000D89C GetCurrentDirectoryA
0000D8B4 GetComputerNameW
0000D8C8 SetFileTime
0000D8D6 SetFilePointer
0000D8E8 MultiByteToWideChar
0000D8FE GetFileAttributesW
0000D914 GetFileSizeEx
0000D924 CreateFileA
0000D932 InitializeCriticalSection
0000D94E DeleteCriticalSection
0000D966 ReadFile
0000D972 GetFileSize
0000D980 WriteFile
0000D98C LeaveCriticalSection
0000D9A4 EnterCriticalSection
0000D9BC SetFileAttributesW
0000D9D2 SetCurrentDirectoryW
0000D9EA CreateDirectoryW
0000D9FE GetTempPathW
0000DA0E GetWindowsDirectoryW

```

*Figure 3.6: Some of the function names indicated that they were used for creating and deleting files.*

Strings also found multiple of the dynamic link libraries that the program uses, such as Kernel32.dll and User32.dll (see Figure 3.7).

```

0000DBAA KERNEL32.dll
0000DBBA wsprintfA
0000DBC4 USER32.dll

```

*Figure 3.7: DLLs found using Strings.*

The names of multiple executable files were also found through strings. These include 'tasksche.exe', 'taskdl.exe', and 'diskpart.exe'.

```

0035962D taskdl.exe
00359689 taskse.exe

```

*Figure 3.8: Two executable file names found using Strings.*

### 3.2.3 Packer Detection

To continue further with static analysis, the file needed to be checked to see if it had been obfuscated using a packer. If this was the case, then extra steps would need to be taken to unpack the sample to see the file in its original format.

Using PEiD, it was possible to see that the sample had been compiled using Microsoft's Visual C++ compiler, indicating it had not been obfuscated using packing (see Figure 3.9).

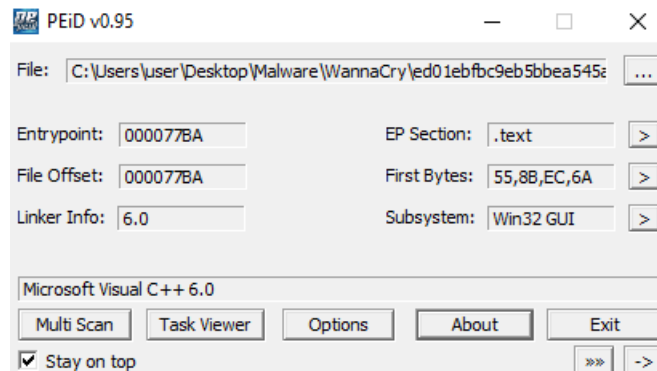


Figure 3.9: The bottom text box shows that the program had been compiled using Microsoft's C++ compiler.

3.2.4 Library and Function Identification

Analysis of the malware’s imported libraries and used functions was critical to furthering the understanding of the malware’s capabilities. While some of these had already been identified in the previous steps (examples of which can be seen in Figure 3.6 and Figure 3.7 ) using the Strings tool, the full extent of the sample’s functions and libraries were identified using the tools Dependency Walker,

3.2.4.1 Identification of Dynamically Linked Libraries

Using the program Dependency Walker, it was possible to only see the libraries that are dynamically linked to the sample. The program identified several libraries used by the program, all of which can be seen in Appendix A, section 5.1.2.

All of the identified libraries come from KernelBase.dll, which is used to launch .exe files (Taj, 2019), and from within that, Ntdll.dll which is a special system support library for using subsystem DLLs (David Solomon, 2012). As can be seen in Appendix A, section 5.1.2, all of the listed libraries are accompanied by yellow circle with a question mark, that indicated they are a missing modules and could not be found in the search path (Dependency Walker, n.d).

Analysis of the functions imported from Kernelbase.dll allowed for further understanding of the sample’s capabilities. To start, the malware imported several functions that allowed for it to create new processes, as shown in Figure 3.10. These processes are often used to run additional programs alongside the main program. The first two are simply to create a new process that runs in the security context of the calling process (Microsoft, 2023), while the last two are for creating a process that runs in the security context of a specified user (Microsoft, 2023).





	N/A	59 (0x003B)	CreateProcessInternalA
	N/A	60 (0x003C)	CreateProcessInternalW
	N/A	58 (0x003A)	CreateProcessAsUserW
	N/A	57 (0x0039)	CreateProcessAsUserA

Figure 3.10: The sample imported four functions allowing it to create new processes during its runtime.

Several of the imported functions indicated that the sample is able to create new files and directories, such as NtCreateFile, which is used to create a new file or directory and can be used for opening existing files and directories (Microsoft, 2021), and NtCreateKey, which is used to create a new registry key, as well as opening an existing one (Microsoft, 2023).

Several functions could be seen in use primarily for evading the detection of an antivirus, allowing the program to go potentially unnoticed, such as NtCreateUserProcess, which is the lowest and last function accessible in user mode that can allow for the creation of new processes, which prevents it from being detected by antiviruses (Gentiles, 2022).

In fact, by ordering the functions alphabetically, it was seen that the sample used a significant number of ‘Nt’ functions, more than any other functions called from Ntdll.dll, a small sample of which can be seen in Figure 3.11 A full list of these can be found in Appendix A, section 5.1.3.



	N/A	191 (0x00BF)	NtAccessCheck
	N/A	192 (0x00C0)	NtAccessCheckAndAuditAlarm
	N/A	193 (0x00C1)	NtAccessCheckByType
	N/A	194 (0x00C2)	NtAccessCheckByTypeAndAuditAlarm
	N/A	195 (0x00C3)	NtAccessCheckByTypeResultList
	N/A	196 (0x00C4)	NtAccessCheckByTypeResultListAndAuditAlarm
	N/A	197 (0x00C5)	NtAccessCheckByTypeResultListAndAuditAlarmByHandle
	N/A	203 (0x00CB)	NtAdjustGroupsToken
	N/A	204 (0x00CC)	NtAdjustPrivilegesToken
	N/A	209 (0x00D1)	NtAllocateLocallyUniqueId
	N/A	211 (0x00D3)	NtAllocateUserPhysicalPages
	N/A	213 (0x00D5)	NtAllocateVirtualMemory
	N/A	214 (0x00D6)	NtAllocateVirtualMemoryEx
	N/A	244 (0x00F4)	NtCancelFile
	N/A	245 (0x00F5)	NtCancelFileEx
	N/A	246 (0x00F6)	NtCancelSynchronousIoFile
	N/A	247 (0x00F7)	NtCancelTimer
	N/A	250 (0x00FA)	NtClearEvent
	N/A	251 (0x00FB)	NtClose
	N/A	252 (0x00FC)	NtCloseObjectAuditAlarm
	N/A	258 (0x0102)	NtCompareObjects
	N/A	265 (0x0109)	NtConvertBetweenAuxiliaryCounterAndPerformanceCounter
	N/A	269 (0x010D)	NtCreateDirectoryObjectEx
	N/A	272 (0x0110)	NtCreateEvent
	N/A	274 (0x0112)	NtCreateFile
	N/A	276 (0x0114)	NtCreateIoCompletion
	N/A	279 (0x0117)	NtCreateKey
	N/A	280 (0x0118)	NtCreateKeyTransacted
	N/A	282 (0x011A)	NtCreateLowBoxToken
	N/A	284 (0x011C)	NtCreateMutant
	N/A	285 (0x011D)	NtCreateNamedPipeFile
	N/A	289 (0x0121)	NtCreatePrivateNamespace
	N/A	296 (0x0128)	NtCreateSection
	N/A	297 (0x0129)	NtCreateSectionEx
	N/A	298 (0x012A)	NtCreateSemaphore
	N/A	299 (0x012B)	NtCreateSymbolicLinkObject
	N/A	301 (0x012D)	NtCreateThreadEx
	N/A	302 (0x012E)	NtCreateTimer
	N/A	303 (0x012F)	NtCreateTimer2
	N/A	308 (0x0134)	NtCreateUserProcess
	N/A	311 (0x0137)	NtCreateWnfStateName
	N/A	315 (0x013B)	NtDelayExecution
	N/A	320 (0x0140)	NtDeleteKey
	N/A	321 (0x0141)	NtDeleteObjectAuditAlarm

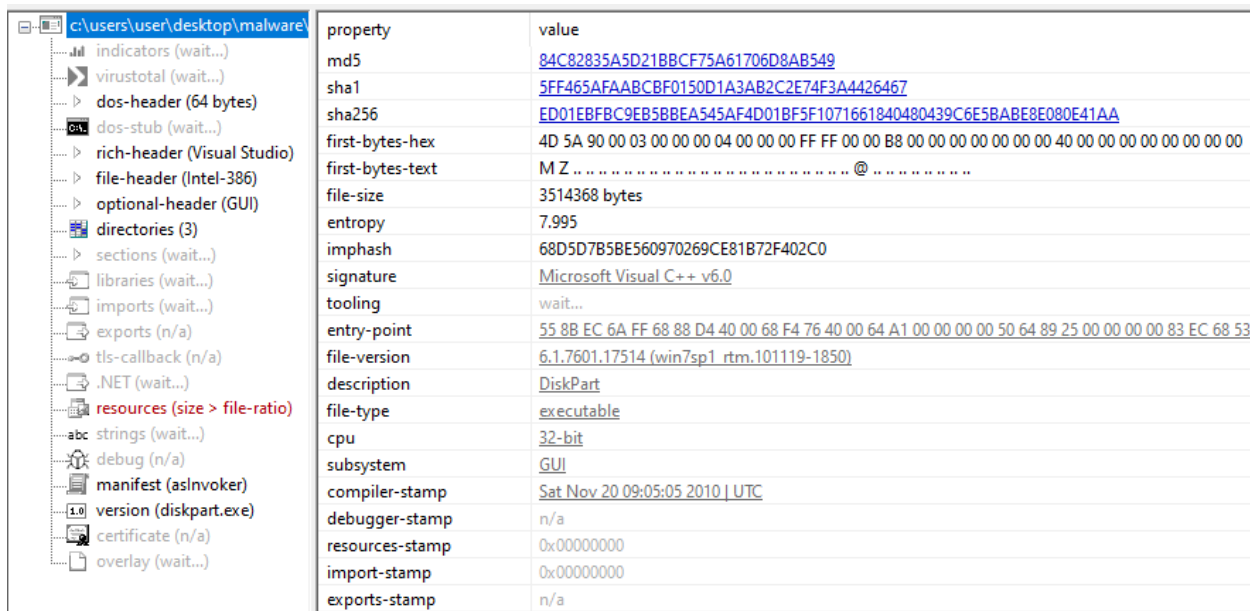
Figure 3.11: A small sample of the imported functions beginning with 'Nt'.

Analyzing the functions imported from Kernel32.dll revealed the sample's heavy reliance on working with files and directories on the compromised device. These functions can be found in Appendix A, 5.1.4. Functions such as CreateDirectoryA, CreateFileA, GetFileAttributesA, GetFileSize, and WriteFile, show that the sample was created with file handling and iteration in mind.

### 3.2.5 Portable Executable Analysis

Using the tool PEStudio allowed for analysis of the sample's metadata through its portable executable (PE) file headers.

A look at the information revealed by PEStudio on the main tab gives several clues around the creation and current state of the file. As seen in Figure 3.12, PEStudio confirms that the sample was compiled using Microsoft Visual Studio. Interestingly, PEStudio also indicates here that the sample may be packed, as the entropy of the file is valued to be 7.995 on a scale of 0 to 8. Values between 7 and 8 typically mean that the file is packed (Fox, 2023), however information that was found during previous steps of the analysis, such as those seen in sections 3.2.2.2, and 3.2.4, indicates that the file was not packed.



property	value
md5	84C82835A5D21BBCF75A61706D8AB549
sha1	5FF465AFAABCBF0150D1A3AB2C2E74F3A4426467
sha256	ED01EBFBC9EB5B8EA545AF4D01BF5F1071661840480439C6E5B8BE8E080E41AA
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 40 00 00 00 00 00 00 00
first-bytes-text	M Z . . . . . @ . . . . .
file-size	3514368 bytes
entropy	7.995
imphash	68D5D7B5BE560970269CE81B72F402C0
signature	Microsoft Visual C++ v6.0
tooling	wait...
entry-point	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 00 83 EC 68 53
file-version	6.1.7601.17514 (win7sp1_rtm.101119-1850)
description	DiskPart
file-type	executable
cpu	32-bit
subsystem	GUI
compiler-stamp	Sat Nov 20 09:05:05 2010   UTC
debugger-stamp	n/a
resources-stamp	0x00000000
import-stamp	0x00000000
exports-stamp	n/a

Figure 3.12: The information on the Main tab within PEStudio.

Using PE-Bear, it was possible to see the size allocations for the file. As Figure 3.13 shows, the virtual and raw sizes of the sections did not significantly differ, confirming that the sample was definitely not packed, as if they had been then the sizes would have a significant difference, especially within the .text section (Michael Sikorski, 2012).

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
▼ .text	1000	7000	1000	69B0	60000020	0	0	0
> 8000	^		8000	mapped: 7000	r-x			
▼ .rdata	8000	6000	8000	5F70	40000040	0	0	0
> E000	^		E000	mapped: 6000	r--			
▼ .data	E000	2000	E000	1958	C0000040	0	0	0
> 10000	^		10000	mapped: 2000	rw-			
▼ .rsrc	10000	34A000	10000	349FA0	40000040	0	0	0
> 35A000	^		35A000	mapped: 34A000	r--			

Figure 3.13: The virtual and raw size allocations for the sample, as shown through PE-Bear.

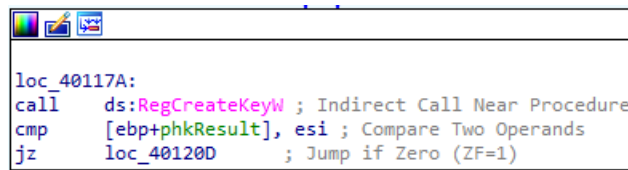
## 3.3 DISASSEMBLY

Using IDA Free, the malware sample was disassembled and its x86 assembly code was analyzed, which allowed for identification of key functions within the code.

### 3.3.1 Identifying Windows API Calls

In section 3.2.4, the dynamically linked libraries used by the program were identified, many of which came from Windows DLLs. Upon analyzing the assembly code of the program, it was possible to see the functions from which some of these libraries are called.

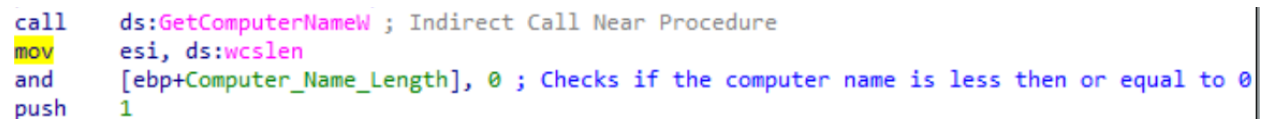
Firstly, it could be seen that the program uses the RegCreateKeyW (see Figure 3.14) to create a new registry key named "Software\Microsoft\WanaCrypt0r\wd".



```
loc_40117A:
call    ds:RegCreateKeyW ; Indirect Call Near Procedure
cmp     [ebp+phkResult], esi ; Compare Two Operands
jz      loc_40120D      ; Jump if Zero (ZF=1)
```

Figure 3.14: The sample was found to create a new registry key using RegCreateKeyW

The sample also uses GetComputerNameW (see Figure 3.15) to get the infected device's name, which it then obfuscates. This process can be seen in Figure 3.16, which includes comments to outline each step of the code better.



```
call    ds:GetComputerNameW ; Indirect Call Near Procedure
mov     esi, ds:wcslen
and     [ebp+Computer_Name_Length], 0 ; Checks if the computer name is less then or equal to 0
push    1
```

Figure 3.15: The sample used the linked function GetComputerNameW to get the compromised device's name.

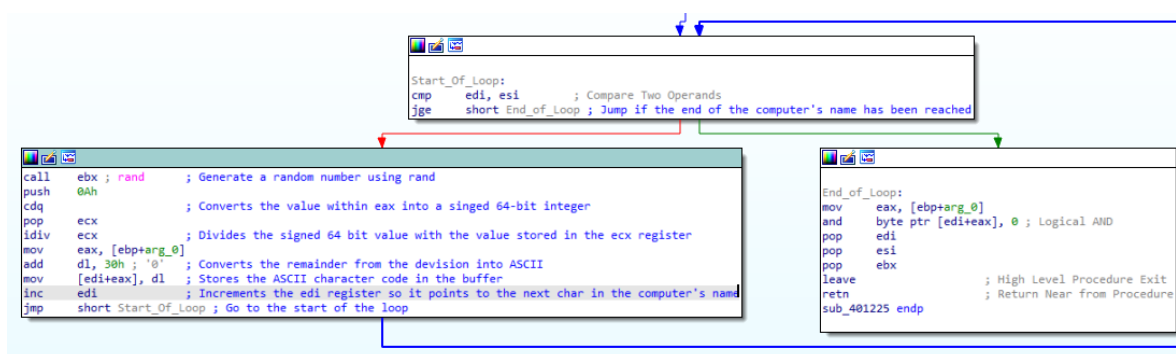


Figure 3.16: The loop used to change each character in the computer's name into a random ASCII character.

There were also multiple instances of the sample creating, writing to, and reading from files using functions imported from Windows DLLs, some of which can be seen in Figure 3.17

```

.text:0040173D      mov     esi, ds:GetProcAddress
.text:00401743      push    offset ProcName ; "CreateFileW"
.text:00401748      push    edi             ; hModule
.text:00401749      call    esi ; GetProcAddress ; Indirect Call Near Procedure
.text:0040174B      push    offset aWritefile ; "WriteFile"
.text:00401750      push    edi             ; hModule
.text:00401751      mov     dword_40F878, eax
.text:00401756      call    esi ; GetProcAddress ; Indirect Call Near Procedure
.text:00401758      push    offset aReadfile ; "ReadFile"
.text:0040175D      push    edi             ; hModule
.text:0040175E      mov     dword_40F87C, eax
.text:00401763      call    esi ; GetProcAddress ; Indirect Call Near Procedure
.text:00401765      push    offset aMovefilew ; "MoveFileW"
.text:0040176A      push    edi             ; hModule
.text:0040176B      mov     dword_40F880, eax
.text:00401770      call    esi ; GetProcAddress ; Indirect Call Near Procedure
.text:00401772      push    offset aMovefileexw ; "MoveFileExW"
.text:00401777      push    edi             ; hModule
.text:00401778      mov     dword_40F884, eax
.text:0040177D      call    esi ; GetProcAddress ; Indirect Call Near Procedure
.text:0040177F      push    offset aDeletefilew ; "DeleteFileW"
.text:00401784      push    edi             ; hModule

```

Figure 3.17: A sample of the code used by the malware to read, write, delete, and move files.

There were multiple instances found where the malware was using Window's threading functionality to assist in handling files. Functions such as EnterCriticalSection and LeaveCriticalSection regularly appeared throughout the assembly code, and analysis of TaskStart, which will be further explained in section 3.3.2.1, showed that the program creates a global mutex named "MsWinZonesCacheCounterMutexA" (shown in Figure 3.18). This mutex is regularly checked within the executable Tasksche.exe (which is also detailed in section 3.3.2.1), suggesting that the sample uses this mutex to check if it is already running on the system.

```

.text:00401F08      push    offset aGlobalMswinon ; "Global\\MsWinZonesCacheCounterMutexA"
.text:00401F0D      lea     eax, [ebp+Buffer] ; Load Effective Address

```

Figure 3.18: Creation of the global mutex used by the malware sample.

Encryption capabilities were also primarily done through the use of imported Windows functions. As will be detailed in section 3.3.2.1, the program uses Microsoft's RSA and AES Cryptographic Provider for its main encryption processes. It also uses multiple other encryption related functions (shown in Figure 3.19), such as CryptImportKey, CryptEncrypt, and CryptGenKey, which are all imported from the now depreciated 'wincrypt.h' header (Microsoft, 2021).

```

push    esi
mov     esi, ds:GetProcAddress
push    offset aCryptacquireco ; "CryptAcquireContextA"
push    edi             ; hModule
call    esi ; GetProcAddress ; Indirect Call Near Procedure
push    offset aCryptimportkey ; "CryptImportKey"
push    edi             ; hModule
mov     dword_40F894, eax
call    esi ; GetProcAddress ; Indirect Call Near Procedure
push    offset aCryptdestroyke ; "CryptDestroyKey"
push    edi             ; hModule
mov     dword_40F898, eax
call    esi ; GetProcAddress ; Indirect Call Near Procedure
push    offset aCryptencrypt ; "CryptEncrypt"
push    edi             ; hModule
mov     dword_40F89C, eax
call    esi ; GetProcAddress ; Indirect Call Near Procedure
push    offset aCryptdecrypt ; "CryptDecrypt"
push    edi             ; hModule
mov     dword_40F8A0, eax
call    esi ; GetProcAddress ; Indirect Call Near Procedure
push    offset aCryptgenkey ; "CryptGenKey"

```

Figure 3.19: The cryptography functions imported from Microsoft DLLs.

### 3.3.2 Identifying Executable Functionality

As was detailed in section 3.2.2.2, the Strings program was able to find multiple names of executable files from the sample, two of which can be seen in Figure 3.8. During analysis of the sample's assembly code, these file names were found to be programs that the sample uses to perform key functions.

#### 3.3.2.1 Analysis of Tasksche.exe

The executable Tasksche.exe is one of the first called when the program runs. When Tasksche.exe runs, it first obfuscates the device's name, as detailed in Figure 3.16, and then goes on to load three strings which appear to potentially be legacy Bitcoin wallet addresses, as they fit in with the naming conventions used by Bitcoin's P2PKH addresses (Sedgwick, 2019).

```
mov [ebp+Source], offset a13am4vw2dhxygx ; "13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94"  
mov [ebp+var_8], offset a12t9ydpgwuez9n ; "12t9YDPgwuez9NyMgw519p7AA8isjr6SMw"  
mov [ebp+var_4], offset a115p7ummngo1p ; "115p7UMMngo1pMvKpHijcRdfJNXj6LrLn"
```

Figure 3.20: Three strings loaded by Tasksche.exe.

Tasksche.exe then goes on to grant all users on the device full control permissions for the directory and subdirectories that the executable exists within, as shown in Figure 3.21. Notably, the malware suppresses any success messages and forces the command to continue regardless of any errors (Microsoft, 2023). This means that anyone using the device will not be notified that they have been granted full permissions to the directory the malware is currently operating within, allowing the malicious software to continue unnoticed.

```
push offset aIcacIsGrantEve ; "icacIs . /grant Everyone:F /T /C /Q"
```

Figure 3.21: Tasksche.exe granting all users full control access to its directory.

This executable's full purpose is highlighted when it calls the function "TaskStart" once various DLLs and files are setup and in place. This function, which can be seen in Figure 3.22, handles the encryption of files on the system using Microsoft's RSA and AES Cryptographic Provider, the use of which was first uncovered in section 3.2.2.2. Figure 3.23 shows this function and loop in detail.

```
push offset String1 ; "TaskStart"  
push eax ; int  
call sub_402924 ; Call Procedure  
pop ecx  
cmp eax, ebx ; Compare Two Operands  
pop ecx  
jz short loc_40215A ; Jump if Zero (ZF=1)
```

Figure 3.22: Tasksche.exe calling "TaskStart".

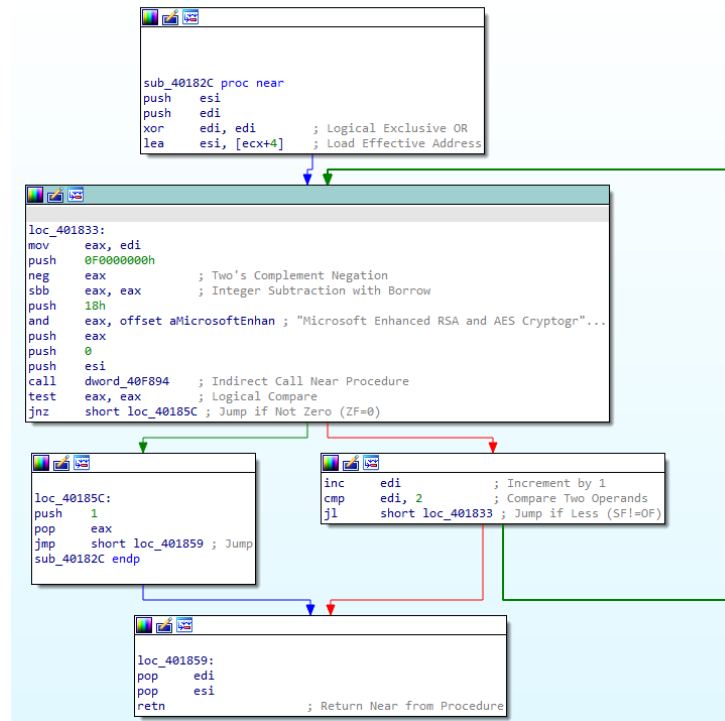


Figure 3.23: The loop using Microsoft's Cryptographic Provider to encrypt the contents of a file.

### 3.3.2.2 Other Executables

Checks were made to see if the other executable files that were identified using Strings could be found within the assembly code, such as 'taskdl.exe', and 'diskpart.exe' (which are shown in Figure 3.8). However, it was discovered that these executables were not identified in IDA Free, as they could not be found using the text search functionality or through manual searching of the strings found by IDA.

Because of this, it is possible that IDA has not completely disassembled the program, meaning that some potentially key functionality of the program could not be identified.

## 3.4 DYNAMIC ANALYSIS

To begin the dynamic analysis stage, the PE file previously analyzed was renamed to WannaCry, the name of the ransomware strain that had previously been identified during the Static Analysis and Disassembly stages of the analysis. Doing this allowed for the sample's activity to be easily filtered and tracked.

### 3.4.1 Executing the Malware

The malware sample was first run to gain a general idea of its functionality during runtime.

Upon clicking on the executable, several files were created within seconds, all of which are shown in Figure 3.24. Multiple of which were specific to the malware sample, such as the WNRy and WNCrY files. Additionally, a read me text file was created, alongside an executable that was used as the main user interface for the malware, which is shown in Figure 3.25.

msg	5/2/2023 5:09 PM	File folder	
@Please_Read_Me@.txt	5/2/2023 5:09 PM	Text Document	1 KB
@WanaDecryptor@.exe	5/12/2017 2:22 AM	Application	240 KB
@WanaDecryptor@.exe	5/2/2023 5:09 PM	Shortcut	1 KB
00000000.eky	5/2/2023 5:09 PM	EKY File	0 KB
00000000.pkty	5/2/2023 5:09 PM	PKY File	1 KB
00000000.res	5/2/2023 5:09 PM	RES File	1 KB
223261683043782.bat	5/2/2023 5:09 PM	Windows Batch File	1 KB
223261683043782.bat.WNCrY	5/2/2023 5:09 PM	WNCrY File	1 KB
A+5½O	5/2/2023 5:09 PM	File	0 KB
b.wnry	5/11/2017 8:13 PM	WNRy File	1,407 KB
c.wnry	5/2/2023 5:09 PM	WNRy File	1 KB
f.wnry	5/2/2023 5:09 PM	WNRy File	1 KB
m.vbs	5/2/2023 5:09 PM	VBScript Script File	1 KB
r.wnry	5/11/2017 3:59 PM	WNRy File	1 KB
s.wnry	5/9/2017 4:58 PM	WNRy File	2,968 KB
t.wnry	5/12/2017 2:22 AM	WNRy File	65 KB
taskdl.exe	5/12/2017 2:22 AM	Application	20 KB
taskse.exe	5/12/2017 2:22 AM	Application	20 KB
u.wnry	5/12/2017 2:22 AM	WNRy File	240 KB
WannaCry.exe	5/14/2017 3:29 PM	Application	3,432 KB

Figure 3.24: Files created upon execution of the malware.



Figure 3.25: The main user interface of the malware.

This user interface included several buttons that allowed for contacting the malware’s owner, getting information on buying bitcoin, and for decrypting the files once payment was successful. It was also possible to send messages through the “Contact Us” link, which when clicked produced a popup window allowing users to type short messages (see Figure 3.26)

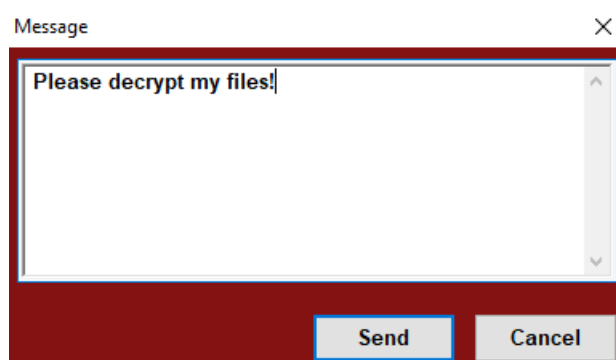


Figure 3.26: The popup window allowing users to contact the malware’s owners.

To test the malware’s encryption capabilities, a text file was created and saved to the malware’s parent directory. The original content of this file is shown in Figure 3.27. After the malware had run, this file was seen to have the extension “.WNCRY” added, and the contents of the file had been changed to be random Unicode characters with the word “WANACRY!” at the beginning (see Figure 3.28).



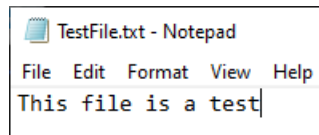


Figure 3.27: The test text file.

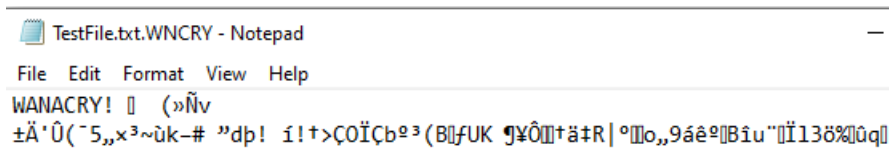


Figure 3.28: The text file after the malware had run.

Analysis of the machine's folders showed that all files had been converted to the WNCRY file type, and every folder with encrypted files had the executable and read me file inserted into them.

The read me file itself contained some simple answers on what had happened to the files and how a user could get their data back. The text was written in broken English, suggesting that it had been machine translated from another language. Figure 3.29 shows this file.

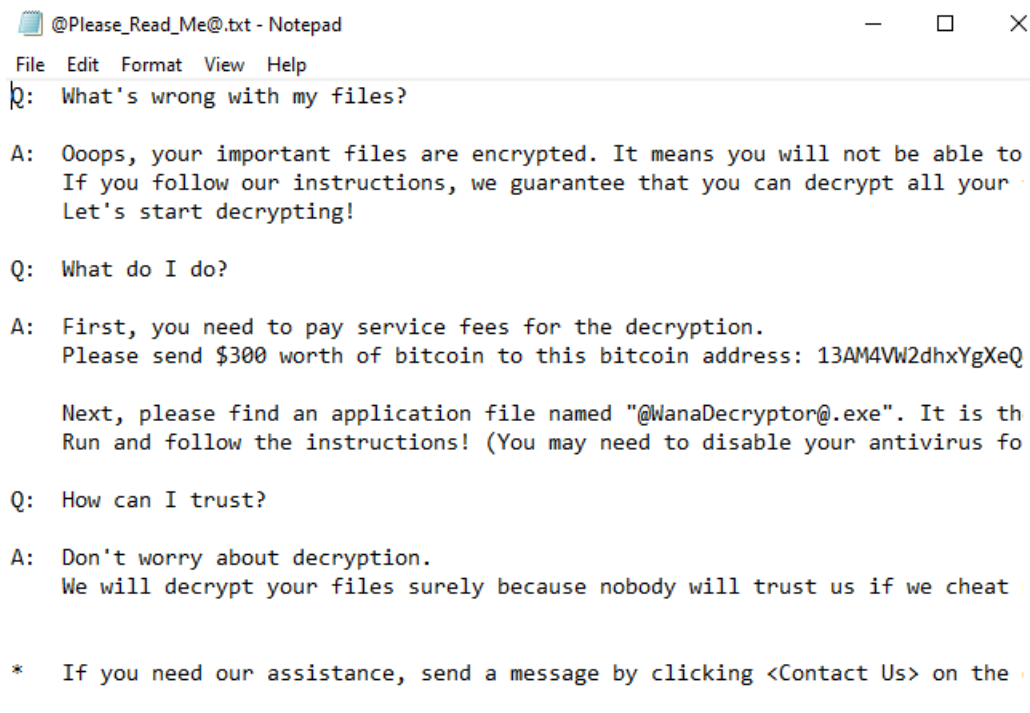


Figure 3.29: The read me file created by the malware.

The last notable change made by the malware was the changed desktop wallpaper, which had previously been a standard Windows wallpaper. After running the sample, it had been changed to notify the user that their files were encrypted, and to run the WannaCry executable to get their files back, as shown in Figure 3.30.

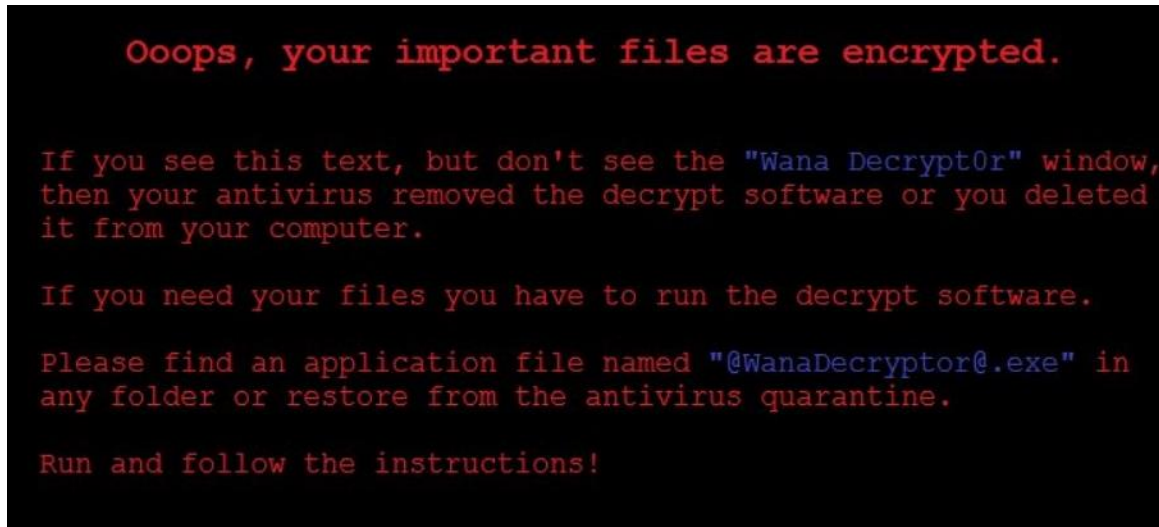


Figure 3.30: The desktop wallpaper now informed the user that their files had been encrypted.

### 3.4.2 Process Monitoring

Using Process Monitor (Procmon), it was possible to see the creation of processes by the sample and analyze information gathered about the process itself.

First, several operations were added to Procmon's filter, which ensured that some key functionality of the malware could be easily observed. These included filters that could show what files were being created and written to, what areas of the registry were being accessed and changed, the DLLs that are being loaded, and the malware's network activity. Figure 3.31 shows all filters added, which are generally the most common operations performed by malware (Bencherchali, 20202) and would give a general oversight to the malicious processes performed by the sample.

Column	Relation	Value	Action
<input checked="" type="checkbox"/> Operation	is	CreateFile	Include
<input checked="" type="checkbox"/> Operation	is	WriteFile	Include
<input checked="" type="checkbox"/> Operation	is	SetRenameInformationFile	Include
<input checked="" type="checkbox"/> Operation	is	SetDispositionInformationFile	Include
<input checked="" type="checkbox"/> Operation	is	RegCreateKey	Include
<input checked="" type="checkbox"/> Operation	is	RegSetValue	Include
<input checked="" type="checkbox"/> Operation	is	RegDeleteKey	Include
<input checked="" type="checkbox"/> Operation	is	RegDeleteValue	Include
<input checked="" type="checkbox"/> Operation	is	TCP Connect	Include
<input checked="" type="checkbox"/> Operation	is	TCP Receive	Include
<input checked="" type="checkbox"/> Operation	is	UDP Send	Include
<input checked="" type="checkbox"/> Operation	is	UDP Receive	Include
<input checked="" type="checkbox"/> Operation	is	Load Image	Include
<input checked="" type="checkbox"/> Operation	is	Process Create	Include
<input checked="" type="checkbox"/> Operation	is	CreatePipe	Include

Figure 3.31: Operations added to Procmon's filter.

The final step in setting up Procmon was to add the name of the malware executable, WannaCry.exe, to the highlighting feature. This was done to make the sample's activities easier to distinguish from other background processes and is shown in Figure 3.32.

Highlight entries matching these conditions:

Process Name is WannaCry.exe then Include

Reset Add Remove

Column	Relation	Value	Action
<input checked="" type="checkbox"/> Process N...	is	WannaCry.exe	Include

Figure 3.32: WannaCry's file path was added to Procmon's highlighting feature.

### 3.4.2.1 Startup Events

Upon running the sample, several events that were identified in previous sections of the analysis were seen to be occurring. The first of which is the loading in of DLLs, which can be seen in Figure 3.33.

WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\user32.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\win32u.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\gdi32.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\gdi32full.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\msvc_p_win.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\userbase.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\advapi32.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\msvcrt.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\sechost.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\rpcrt4.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\sspicli.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\cryptbase.dll	SUCCESS
WannaCry.exe	1964	Load Image	C:\Windows\SysWOW64\bcryptprimitives.dll	SUCCESS

Figure 3.33: The malware first loaded in several DLLs.

The malware then went on to create a new key within the registry, a functionality which was first noted when the sample's assembly code was analyzed (detailed in section 3.3.1). Notably, the malware was denied access on two attempts of creating a new registry key, however, it was successful on its third attempt as is shown in Figure 3.34.

WannaCry.exe	1964	RegCreateKey	HKLM\Software\WOW6432Node\WanaCrypt0r	ACCESS DENIED	Desired Access: Maximum Allowed
WannaCry.exe	1964	RegCreateKey	HKLM\Software\WOW6432Node	SUCCESS	Desired Access: Maximum Allowed,
WannaCry.exe	1964	RegCreateKey	HKLM\SOFTWARE\WOW6432Node\WanaCrypt0r	ACCESS DENIED	Desired Access: Maximum Allowed
WannaCry.exe	1964	RegCreateKey	HKCU\Software\WanaCrypt0r	SUCCESS	Desired Access: Maximum Allowed,
WannaCry.exe	1964	RegSetValue	HKCU\Software\WanaCrypt0r\wd	SUCCESS	Type: REG_SZ, Length: 78, Data: C

Figure 3.34: The malware made multiple attempts to create a new registry key and was successful on the third try.

An interesting detail highlighted by Procmon was the sample's ability to handle errors that occurred during runtime. For example, there were multiple instances where the sample attempted to create or read from a file that did not exist. Instead of failing to proceed with the process, the malware was able to switch to a different directory and attempt to locate the file again. Figure 3.35 shows an example of this, where `lcacls.exe` was attempted to be accessed to run the `lcacls` command shown in Figure 3.21. The malware first tried to run the command from its original directory, but upon error it was able to switch to the `Windows\SysWOW64` directory where `lcacls.exe` is located, allowing it to successfully run the command.

WannaCry.exe	1964	CreateFile	C:\Windows\apppatch\sysmain.sdb	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Malware\WannaCry\lcacls.exe	NAME NOT FOUND
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Malware\WannaCry\lcacls.exe	NAME NOT FOUND
WannaCry.exe	1964	CreateFile	C:\Windows\SysWOW64\lcacls.exe	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Windows\SysWOW64\lcacls.exe	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Windows\SysWOW64\lcacls.exe	SUCCESS

Figure 3.35: The sample often had to go through multiple attempts to create files.

### 3.4.2.2 Active Encryption

Multiple processes were captured showing the malware actively encrypting files on the device, allowing for the full encryption process to be revealed. Using Figure 3.36 as an example of this, it was seen that the malware would attempt to create a new file with the prefix “.WNCRY” attached. This always resulted in “NAME NOT FOUND” being returned. The program would then go on to access the file it was currently encrypting, before creating a new file of the exact same name but with the prefix “.WYNCRYT” attached. It would then write to this file, rename it, and then write to the original file.

WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip.WNCRY	NAME NOT FOUND
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
cscrip.exe	524	CreateFile	C:\Users\user\Desktop\Malware\WannaCry	SUCCESS
cscrip.exe	524	Load Image	C:\Windows\SysWOW64\kernel32.dll	SUCCESS
cscrip.exe	524	Load Image	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip	SUCCESS
WannaCry.exe	1964	SetRenameInformationFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	NAME NOT FOUND
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip	SUCCESS
WannaCry.exe	1964	CreateFile	C:\Users\user\Desktop\Samples\7.7.zip	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip	SUCCESS
WannaCry.exe	1964	WriteFile	C:\Users\user\Desktop\Samples\7.7.zip	SUCCESS

Figure 3.36: The processes captured during the sample's encryption process.

Looking at the details of these events, it was possible to see the “.WYNCRYT” file size gradually increasing (an example of which is shown in Figure 3.37) , until it was roughly the same size as the original file. The multiple write operations are potentially due to the file’s encryption being done in stages.

CreateFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS	Desired Access: Generic Write, Read Attribut
WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS	Offset: 0, Length: 8, Priority: Normal
WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS	Offset: 8, Length: 4, Priority: Normal
WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS	Offset: 12, Length: 256, Priority: Normal
WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS	Offset: 268, Length: 4, Priority: Normal
WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS	Offset: 272, Length: 8, Priority: Normal
WriteFile	C:\Users\user\Desktop\Samples\7.7.zip.WYNCRYT	SUCCESS	Offset: 280, Length: 63,376, Priority: Normal

Figure 3.37: The sample would take multiple attempts to write to an encrypted file.

The encryption of files was incredibly verbose, and the sample was found to have encrypted everything it could access. The operations shown within Procmon indicated that the malware would begin encrypting files within its own directory, and then go on to target the user’s Documents file first. After that, the order of files seemed to be done alphabetically.

### 3.4.3 Registry Analysis

As was discussed in section 3.4.2.2, the malware was seen to be making significant changes to the Windows Registry during its run time. To analyze the exact changes made, the tool Regshot was used to capture a snapshot before and after the malware was run. Looking at the comparison between the two snapshots, which is shown in Figure 3.38, there were a significant number of registry keys that were deleted and added, indicating that the malware almost entirely rewrote the device's registry. However, a notable problem with Regshot's output is that no files or folders were detected to have been deleted, added, or changed by the malware, despite previous analysis heavily indicating file editing did take place (an example of which can be seen in Figure 3.36). It is possible that this false negative stems from the malware changing the entire C:\ directory, meaning Regshot was attempting to analyze a directory that no longer existed.

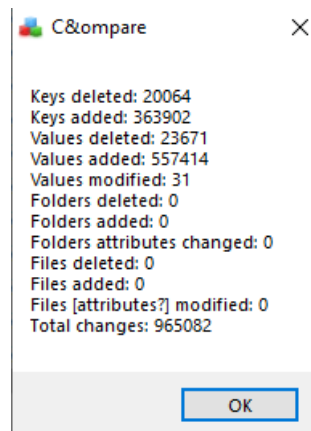


Figure 3.38: Regshot's analysis of registry files and keys that were modified by the malware.

### 3.4.4 Network Analysis

Analysis of the requests made over the network was done using Wireshark, with packets being captured over the loopback traffic capture interface.

It was possible to see multiple requests from the malware making NetBIOS Name Service (NBNS) requests, which is a service that can translate human-readable names to IP addresses (shown in Figure 3.39).

169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>
169.254.131.29	169.254.255.255	NBNS	82 Name query NB DESKTOP-14QC1L8<1c>

Figure 3.39: Name query requests originating from the malware.

It was also possible to see the malware using the Browser protocol to send host announcement packets over the network (shown in Figure 3.40).

216 382.612912	169.254.131.29	169.254.255.255	BROWSER	233 Host Announcement DESKTOP-14QC1L8, Workstation, Server, NT Workstation
Frame 216: 233 bytes on wire (1864 bits), 233 bytes captured (1864 bits) on interface 0				
Internet Protocol Version 4, Src: 169.254.131.29, Dst: 169.254.255.255				
User Datagram Protocol, Src Port: 138, Dst Port: 138				
NetBIOS Datagram Service				
Message Type: Direct_group datagram (17)				
Flags: 0x02, This is first fragment, Node Type: B node				
Datagram ID: 0xc34d				
Source IP: 169.254.131.29				
Source Port: 138				
Datagram length: 187 bytes				
Packet offset: 0 bytes				
Source name: DESKTOP-14QC1L8<20> (Server service)				
Destination name: WORKGROUP<1d> (Local Master Browser)				
SMB (Server Message Block Protocol)				
SMB MailSlot Protocol				
Microsoft Windows Browser Protocol				

Figure 3.40: A packet sent using the Browser protocol, containing a host announcement.

Some packets also included Onion URLs. When testing the functionality of the Wana Decrypt0r program, clicking on the “Contact Us” button resulted in several TCP packets being sent, some of which included the URL of the server presumably used to hold information about each infected device. Figure 3.41 shows one of these packets, with the URL highlighted in blue.

388 804.951788	127.0.0.1	127.0.0.1	TCP	74 49710 → 9050 [PSH, ACK] Seq=4 Ack=3 Win=327424 Len=30
Frame 388: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0				
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1				
Transmission Control Protocol, Src Port: 49710, Dst Port: 9050, Seq: 4, Ack: 3, Win: 327424, Len: 30				
Data (30 bytes)				
Data: 05010003176377776e6877686c7a35326d61716d372e6f6e696f6e000050				
Length: 30				

Figure 3.41: A TCP packet containing Onion links.

# 4 DISCUSSION

## 4.1 OVERALL DISCUSSION

---

Overall, it was possible to identify many of the functionalities used by the malware sample using the tools and techniques outlined in the methodology.

The malware is almost solely encryption-based and was created with the intent of encrypting as many of a victim's files as possible, to the extent that it would even encrypt some of its own files. It clearly makes use of Windows's own encryption algorithms to do this, as was shown in Figure 3.23. One key aspect of maliciousness in this area is the malware's targeting of the user's Documents directory, where most users will keep their most important files and folders. As was detailed in section 3.4.2.2, the Documents folder was targeted first, meaning that even if the user realized their files were about to be encrypted and shut down their device before the malware was able to complete the encryption process, the chances of their most important files already being encrypted were still significantly high.

Overall, the majority of the aims set out at the beginning of the report were met, and it was possible to outline the main functionality of the malware sample. Key functions were identified during analysis of the sample's assembly code, and the impact on the system was measured during the dynamic analysis.

One aim that could have been better met was the uncovering of all areas of the malware's behavior, especially the networking functionality. As the malware was running on a machine that was not connected to any other network, it was not possible to see the malware attempting to and successfully infecting other devices, a functionality that other malware analysts have noted that WannaCry is capable of doing (Hunt, 2017).

### 4.1.1 Evasion Capabilities

The sample is clearly written with evasion in mind, and analysis of the functions detailed in section 3.2.4.1 indicates that the malware's developers were intent on keeping the malicious activities hidden for as long as possible. For example, the high number of functions imported from Ntdll.dll means that the sample is able to force the CPU to switch from User-mode, which can and is highly monitored by antiviruses, to Kernel-mode, which cannot be monitored by antiviruses (Mosch, 2021), allowing for the sample to run code without detection. The significant number of functions from Ntdll.dll that specifically begin with 'Nt' and 'Zw' also acts as proof of detection evasion, as these functions can only be called by kernel-mode drivers and not by programs running in User-mode (Microsoft, 2021), indicating that the program is designed to run within the kernel itself.

### 4.1.2 Persistence Capabilities

The malware's persistence capabilities became evident while analyzing the assembly code. To start, the sample creates a new registry key within the "Software/Microsoft" registry (shown in Figure 3.14). This is a common tactic for malware to do, allowing the sample to launch after a reboot, and to integrate with existing legitimate processes (Grimes, 2017). The significant number of changes made to the registry, which were outlined in Figure 3.38, clearly show that the malware aims to exert a high level of control over the infected device.



Additionally, the malware was seen to regularly suppress messages or popups that would typically appear after certain commands are run, preventing the user from knowing that a setting that is critical to the security of the device had been changed. A key example of this is the suppression of the *'icacls'* success message, which was detailed in section 3.3.2.1.

## 4.2 COUNTERMEASURES

---

Preventing this strain of WannaCry from infecting devices can be done using one notable piece of software – an antivirus. Having one present and running on all systems within the network would almost certainly protect any device that WannaCry attempts to infect. As was shown in Figure 3.2, 95% of antiviruses on VirusTotal could identify this malware as being malicious, meaning that using any of those security vendors would prevent this malware from totally encrypting a system.

## 4.3 FUTURE WORK

---

If more time and resources were available, further analysis of the sample's networking capabilities would be an important next step that, unfortunately, was not possible due to the time and technological constraints surrounding this report.

If possible, a simple network could be set up to allow for full analysis of the malware as it runs. This environment would be more realistic than the ones present in most homes and offices and would give an idea of the functionality of the virus when presented on devices and operating systems other than Windows.

Additionally, further analysis of the other executables (such as the ones detailed in Figure 3.8) created during the runtime would allow for further expansion into the full capabilities of the malware. This could potentially be done after execution of the malware, as these executables did not get identified during the static disassembly stage.

## 5 REFERENCES

- Bencherchali, N., 2020. *Hunting Malware with Windows Sysinternals — Process Monitor*. [Online]  
Available at: <https://nasbench.medium.com/hunting-malware-with-windows-sysinternals-process-monitor-e67476f44514>  
[Accessed 2 May 2023].
- David Solomon, M. R. A. I., 2012. *Windows® Internals, Sixth Edition, Part 1*. 6th ed. Redmond: Microsoft Press.
- Dependency Walker, n.d. *Module Dependency Tree View*. [Online]  
Available at: [https://www.dependencywalker.com/help/html/hidr\\_module\\_tree\\_view.htm](https://www.dependencywalker.com/help/html/hidr_module_tree_view.htm)  
[Accessed 28 April 2023].
- Fox, N., 2023. *PeStudio Overview: Setup, Tutorial and Tips*. [Online]  
Available at: <https://www.varonis.com/blog/pestudio>  
[Accessed 29 April 2023].
- Gentiles, E., 2022. *Making NtCreateUserProcess Work*. [Online]  
Available at: <https://captmeelo.com/redteam/maldev/2022/05/10/ntcreateuserprocess.html>  
[Accessed 28 April 2023].
- Grimes, R., 2017. *Infected with malware? Check your Windows registry*. [Online]  
Available at: <https://www.csoonline.com/article/2894520/are-you-infected-with-malware-check-windows-registry-keys.html>  
[Accessed 30 April 2023].
- Hunt, T., 2017. *Everything you need to know about the WannaCry / Wcry / WannaCrypt ransomware*. [Online]  
Available at: <https://www.troyhunt.com/everything-you-need-to-know-about-the-wannacrypt-ransomware/>  
[Accessed 2 May 2023].
- Michael Sikorski, A. H., 2012. *Practical Malware Analysis*. 1st ed. San Francisco: No Starch Press.
- Microsoft, 2021. *CryptEncrypt function (wincrypt.h)*. [Online]  
Available at: <https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptencrypt>  
[Accessed 2 May 2023].
- Microsoft, 2021. *NtCreateFile function (winternl.h)*. [Online]  
Available at: <https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntcreatefile>  
[Accessed 28 April 2021].
- Microsoft, 2021. *Using Nt and Zw Versions of the Native System Services Routines*. [Online]  
Available at: <https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/using-nt-and-zw-versions-of-the-native-system-services-routines>  
[Accessed 29 April 2023].

- Microsoft, 2023. *CreateProcessA function (processthreadsapi.h)*. [Online]  
Available at: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa>  
[Accessed 28 April 2023].
- Microsoft, 2023. *CreateProcessAsUserA function (processthreadsapi.h)*. [Online]  
Available at: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessasusera>  
[Accessed 28 April 2023].
- Microsoft, 2023. *icaccls*. [Online]  
Available at: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/icaccls>  
[Accessed 1 May 2023].
- Microsoft, 2023. *ZwCreateKey function (wdm.h)*. [Online]  
Available at: <https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-zwcreatekey>  
[Accessed 28 April 2023].
- Milosevic, N., 2013. *History of malware*. [Online]  
Available at: [https://pure.manchester.ac.uk/ws/portalfiles/portal/32297162/FULL\\_TEXT.PDF](https://pure.manchester.ac.uk/ws/portalfiles/portal/32297162/FULL_TEXT.PDF)  
[Accessed 2 May 2023].
- Mosch, F., 2021. *A tale of EDR bypass methods*. [Online]  
Available at: <https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/>  
[Accessed 29 April 2023].
- NetSPI, 2013. *Magic Bytes – Identifying Common File Formats at a Glance*. [Online]  
Available at: <https://www.netSPI.com/blog/technical/web-application-penetration-testing/magic-bytes-identifying-common-file-formats-at-a-glance/>  
[Accessed 25 April 2023].
- Paul Huijbregts, J. A. a. J. G., 2023. *Microsoft Defender for Endpoint in Depth*. 1st ed. Birmingham: Packt Publishing Ltd..
- Pietrek, M., 2002. *An In-Depth Look into the Win32 Portable Executable File Format*. [Online]  
Available at: [https://learn.microsoft.com/en-us/previous-versions/bb985992\(v=msdn.10\)?redirectedfrom=MSDN](https://learn.microsoft.com/en-us/previous-versions/bb985992(v=msdn.10)?redirectedfrom=MSDN)  
[Accessed 25 April 2023].
- Sedgwick, K., 2019. *Everything You Should Know About Bitcoin Address Formats*. [Online]  
Available at: <https://news.bitcoin.com/everything-you-should-know-about-bitcoin-address-formats/>  
[Accessed 30 April 2023].
- Statista, 2023. *Annual number of malware attacks worldwide from 2015 to 2022 (in billions)*. [Online]  
Available at: <https://www.statista.com/statistics/873097/malware-attacks-per-year-worldwide/>  
[Accessed 2 May 2023].

Taj, R., 2019. *Applications Fail to Start with Kernelbase.dll error*. [Online]

Available at: <https://answers.microsoft.com/en-us/windows/forum/all/applications-fail-to-start-with-kernelbasedll/44a4c2ad-a43f-479b-b026-bcdc2ff01285>

[Accessed 28 April 2023].

UK Government, 2023. *Cyber security breaches survey 2023*. [Online]

Available at: <https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2023/cyber-security-breaches-survey-2023>

[Accessed 2 May 2023].

VirusTotal, 2023. *How it works*. [Online]

Available at: <https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works>

[Accessed 2 May 2023].

# APPENDICES

## APPENDIX A

### 5.1.1 VirusTotal Results

Popular threat  ransomware.wannacry/wannacryptor Threat categ Family labels label			
Security vendors' analysis		Do you want to automate checks?	
ALYac  Trojan.Ransom.Wa...	Antiy-AVL  Trojan[Ransom]/Wi...	Rising  Ransom.WanaCryp...	Sangfor Engine Zero  Ransom.Win32.Wa...
Arcabit  Trojan.Ransom.Wa...	Avast  Win32:WanaCry-A...	SecureAge  Malicious	SentinelOne (Static ML)  Static AI - Suspici...
AVG  Win32:WanaCry-A...	Avira (no cloud)  TR/Ransom.JB	Sophos  Troj/Ransom-EMG	Symantec  Ransom.Wannacry
Baidu  Win32.Trojan.Wan...	BitDefender  Trojan.Ransom.Wa...	TACHYON  Ransom/W32.Wan...	TEHTRIS  Generic.Malware
BitDefenderThreat Gen:NN.ZexaF.361...	Bkav Pro  W32.WannaCrypLT...	Tencent  Trojan-Ransom.Win...	Trapmine  Malicious.high.ml...
ClamAV  Win.Ransomware...	CrowdStrike Falcon  Win/malicious_con...	Trellix (FireEye)  Generic.mg.84c82...	TrendMicro  Ransom_WANA.A
Cylance  Unsafe	Cynet  Malicious (score: ...	TrendMicro-HouseCall  Ransom_WANA.A	VBA32  TrojanRansom.Wa...
Cyren  W32/Trojan.ZTSA...	DeepInSinct  MALICIOUS	VIPRE  Trojan.Ransom.Wa...	VirIT  Trojan.Win32.Wan...
DrWeb  Trojan.Encoder.11...	Elastic  Malicious (high Co...	ViRobot  Trojan.Win32.S.Wa...	Webroot  W32.Ransom.Wan...
Emsisoft  Trojan.Ransom.Wa...	eScan  Trojan.Ransom.Wa...	Xcitium  TrojWare.Win32.Ra...	Yandex  Trojan.Igent.bUj9p...
ESET-NOD32  Win32/Filecoder.W...	F-Secure  Trojan.TR/Ranso...	Zillya  Trojan.WannaCry...	ZoneAlarm by Check Point  Trojan-Ransom.Wi...
Fortinet  W32/WannaCryptor...	GData  Win32.Trojan-Rans...	Zoner  Trojan.Win32.556...	Acronis (Static ML)  Undetected
Google  Detected	Gridinsoft (no cloud)  Malware.Win32.Ge...	CMC  Undetected	SUPERAntiSpyware  Undetected
Ikarus  Trojan-Ransom.W...	Jiangmin  Trojan.Wanna.eo	Avast-Mobile  Unable to process ...	BitDefenderFalx  Unable to process...
K7AntiVirus  Trojan ( 0050d71...	K7GW  Trojan ( 0050d71...	Symantec Mobile Insight  Unable to process ...	Trustlook  Unable to process ...
Kaspersky  Trojan-Ransom.Wi...	Lionic  Trojan.Win32.Wan...		
Malwarebytes  Generic.Ransom.Fi...	MAX  Malware (ai Score...		
MaxSecure  Trojan.Ransom.W...	McAfee  Ransom-O.g		
McAfee-GW-Edition  BehavesLike.Win32...	Microsoft  Ransom.Win32/W...		

Figure 3.0.1: The security vendor's analysis of the sample, as provided by VirusTotal

### 5.1.2 Dependency Walker Results

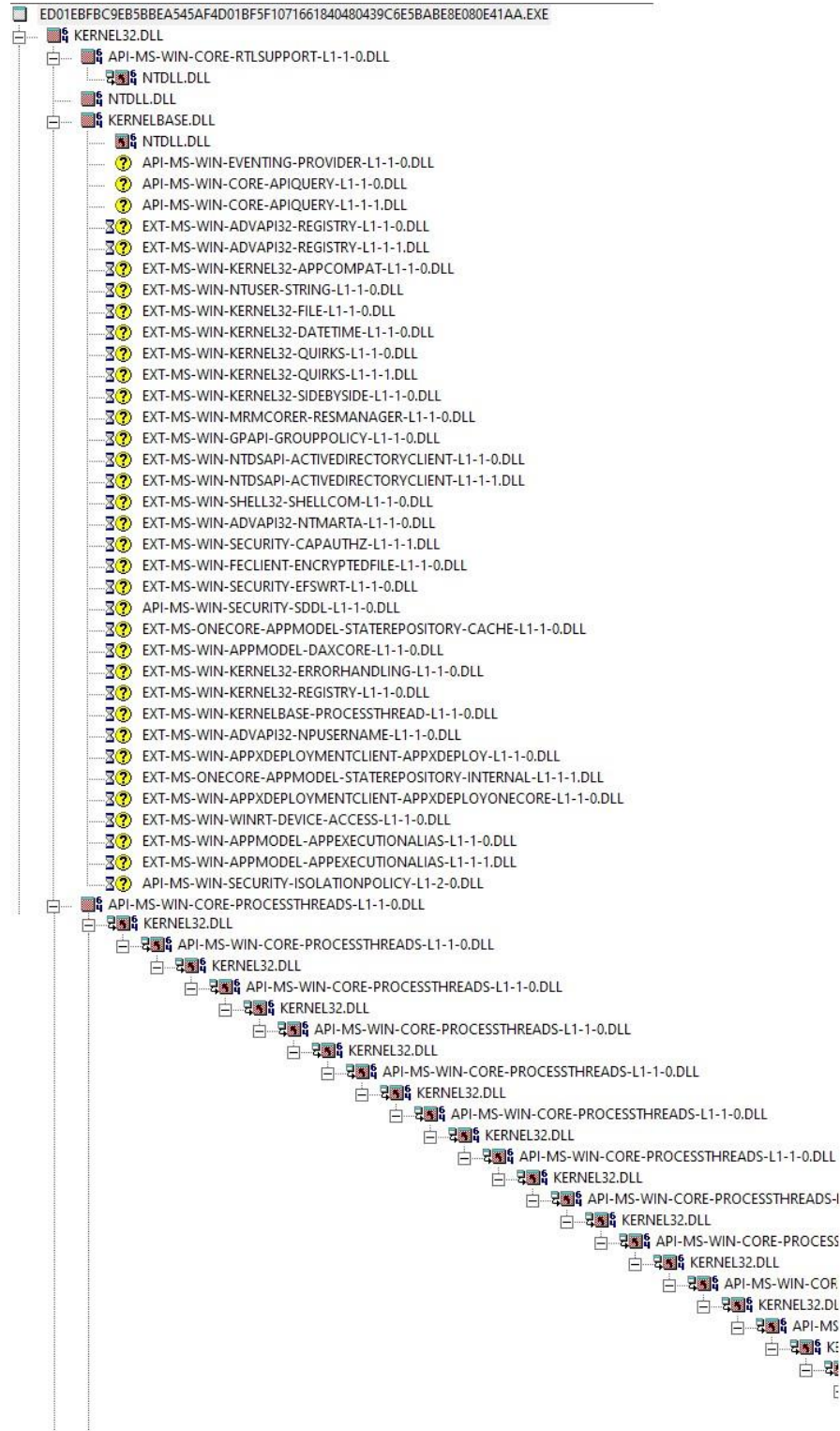


Figure 0.2: The sample's dynamically linked libraries as analysed by Dependency Walker.

### 5.1.3 Functions Imported From Ntdll.dll

- `__C_specific_handler`
- `__chkstk`
- `CsrAllocateCaptureBuffer`
- `CsrAllocateMessagePointer`
- `CsrCaptureMessageBuffer`
- `CsrCaptureMessageMultiUnicodeStringsInPlace`
- `CsrClientCallServer`
- `CsrClientConnectToServer`
- `CsrFreeCaptureBuffer`
- `CsrGetProcessId`
- `DbgPrint`
- `DbgPrintEx`
- `DbgUiConnectToDbg`
- `DbgUiContinue`
- `DbgUiConvertStateChangeStructure`
- `DbgUiConvertStateChangeStructureEx`
- `DbgUiDebugActiveProcess`
- `DbgUiGetThreadDebugObject`
- `DbgUiStopDebugging`
- `DbgUiWaitStateChange`
- `EtwCheckCoverage`
- `EtwEventEnabled`
- `EtwEventRegister`
- `EtwEventUnregister`
- `EtwEventWrite`
- `EtwEventWriteNoRegistration`
- `EtwEventWriteTransfer`
- `isalpha`
- `iswalpha`
- `_itow_s`
- `LdrAccessResource`
- `LdrAddDllDirectory`
- `LdrAddLoadAsDataTable`
- `LdrAddRefDll`
- `LdrAppxHandleIntegrityFailure`
- `LdrCallEnclave`
- `LdrCreateEnclave`
- `LdrDeleteEnclave`
- `LdrDisableThreadCalloutsForDll`
- `LdrFindResource_U`
- `LdrFindResourceEx_U`
- `LdrGetDllDirectory`
- `LdrGetDllFullName`
- `LdrGetDllHandle`
- `LdrGetDllHandleByMapping`
- `LdrGetDllHandleByName`
- `LdrGetDllPath`
- `LdrGetFileNameFromLoadAsDataTable`
- `LdrGetProcedureAddress`
- `LdrGetProcedureAddressForCaller`

- LdrInitializeEnclave
- LdrLoadAlternateResourceModule
- LdrLoadAlternateResourceModuleEx
- LdrLoadDll
- LdrLoadEnclaveModule
- LdrpResGetMappingSize
- LdrpResGetResourceDirectory
- LdrQueryImageFileExecutionOptionsEx
- LdrQueryImageFileKeyOption
- LdrQueryOptionalDelayLoadedAPI
- LdrRemoveDllDirectory
- LdrRemoveLoadAsDataTable
- LdrResFindResource
- LdrResFindResourceDirectory
- LdrResGetRCConfig
- LdrResolveDelayLoadedAPI
- LdrResolveDelayLoadsFromDll
- LdrResRelease
- LdrResSearchResource
- LdrRscIsTypeExist
- LdrSetDefaultDllDirectories
- LdrUnloadAlternateResourceModule
- LdrUnloadDll
- LdrUpdatePackageSearchPath
- \_local\_unwind
- memcmp
- memcpy
- memcpy\_s
- memmove
- memset
- NlsMbCodePageTag
- NtAccessCheck
- NtAccessCheckAndAuditAlarm
- NtAccessCheckByType
- NtAccessCheckByTypeAndAuditAlarm
- NtAccessCheckByTypeResultList
- NtAccessCheckByTypeResultListAndAuditAlarm
- NtAccessCheckByTypeResultListAndAuditAlarmByHandle
- NtAdjustGroupsToken
- NtAdjustPrivilegesToken
- NtAllocateLocallyUniqueId
- NtAllocateUserPhysicalPages
- NtAllocateVirtualMemory
- NtAllocateVirtualMemoryEx
- NtCancelIoFile
- NtCancelIoFileEx
- NtCancelSynchronousIoFile
- NtCancelTimer
- NtClearEvent
- NtClose
- NtCloseObjectAuditAlarm



- NtCompareObjects
- NtConvertBetweenAuxiliaryCounterAndPerformanceCounter
- NtCreateDirectoryObjectEx
- NtCreateEvent
- NtCreateFile
- NtCreateIoCompletion
- NtCreateKey
- NtCreateKeyTransacted
- NtCreateLowBoxToken
- NtCreateMutant
- NtCreateNamedPipeFile
- NtCreatePrivateNamespace
- NtCreateSection
- NtCreateSectionEx
- NtCreateSemaphore
- NtCreateSymbolicLinkObject
- NtCreateThreadEx
- NtCreateTimer
- NtCreateTimer2
- NtCreateUserProcess
- NtCreateWnfStateName
- NtDelayExecution
- NtDeleteKey
- NtDeleteObjectAuditAlarm
- NtDeletePrivateNamespace
- NtDeleteValueKey
- NtDeleteWnfStateName
- NtDeviceIoControlFile
- NtDuplicateObject
- NtDuplicateToken
- NtEnumerateKey
- NtEnumerateValueKey
- NtFilterToken
- NtFlushBuffersFile
- NtFlushKey
- NtFlushVirtualMemory
- NtFreeUserPhysicalPages
- NtFreeVirtualMemory
- NtFsControlFile
- NtGetCachedSigningLevel
- NtGetContextThread
- NtGetNlsSectionPtr
- NtGetWriteWatch
- NtImpersonateAnonymousToken
- NtIsProcessInJob
- NtLoadEnclaveData
- NtLoadKey
- NtLoadKeyEx
- NtLockFile
- NtLockVirtualMemory
- NtMapUserPhysicalPages

- NtMapViewOfSection
- NtMapViewOfSectionEx
- NtNotifyChangeDirectoryFile
- NtNotifyChangeDirectoryFileEx
- NtNotifyChangeKey
- NtNotifyChangeMultipleKeys
- NtOpenDirectoryObject
- NtOpenEvent
- NtOpenFile
- NtOpenKey
- NtOpenKeyEx
- NtOpenKeyTransactedEx
- NtOpenMutant
- NtOpenObjectAuditAlarm
- NtOpenPrivateNamespace
- NtOpenProcess
- NtOpenProcessToken
- NtOpenProcessTokenEx
- NtOpenSection
- NtOpenSemaphore
- NtOpenSymbolicLinkObject
- NtOpenThread
- NtOpenThreadToken
- NtOpenTimer
- NtPrivilegeCheck
- NtPrivilegedServiceAuditAlarm
- NtPrivilegeObjectAuditAlarm
- NtProtectVirtualMemory
- NtPulseEvent
- NtQueryAttributesFile
- NtQueryAuxiliaryCounterFrequency
- NtQueryDefaultLocale
- NtQueryDirectoryFile
- NtQueryDirectoryFileEx
- NtQueryDirectoryObject
- NtQueryEaFile
- NtQueryEvent
- NtQueryFullAttributesFile
- NtQueryInformationFile
- NtQueryInformationJobObject
- NtQueryInformationProcess
- NtQueryInformationThread
- NtQueryInformationToken
- NtQueryInstallUILanguage
- NtQueryKey
- NtQueryLicenseValue
- NtQueryMultipleValueKey
- NtQueryObject
- NtQuerySecurityAttributesToken
- NtQuerySecurityObject
- NtQuerySymbolicLinkObject

- NtQuerySystemInformation
- NtQuerySystemInformationEx
- NtQueryValueKey
- NtQueryVirtualMemory
- NtQueryVolumeInformationFile
- NtQueryWnfStateData
- NtQueueApcThread
- NtRaiseException
- NtRaiseHardError
- NtReadFile
- NtReadFileScatter
- NtReadVirtualMemory
- NtReleaseMutant
- NtReleaseSemaphore
- NtRemoveIoCompletion
- NtRemoveIoCompletionEx
- NtRemoveProcessDebug
- NtResetEvent
- NtResetWriteWatch
- NtRestoreKey
- NtResumeThread
- NtSaveKeyEx
- NtSetCachedSigningLevel
- NtSetContextThread
- NtSetDefaultLocale
- NtSetEvent
- NtSetInformationFile
- NtSetInformationKey
- NtSetInformationObject
- NtSetInformationProcess
- NtSetInformationThread
- NtSetInformationToken
- NtSetInformationVirtualMemory
- NtSetIoCompletion
- NtSetSecurityObject
- NtSetSystemInformation
- NtSetSystemTime
- NtSetTimerEx
- NtSetValueKey
- NtSignalAndWaitForSingleObject
- NtSuspendThread
- NtTerminateEnclave
- NtTerminateProcess
- NtTerminateThread
- NtUnloadKey
- NtUnlockFile
- NtUnlockVirtualMemory
- NtUnmapViewOfSection
- NtUnmapViewOfSectionEx
- NtWaitForMultipleObjects
- NtWaitForSingleObject

- NtWriteFile
- NtWriteFileGather
- NtWriteVirtualMemory
- NtYieldExecution
- PssNtCaptureSnapshot
- PssNtDuplicateSnapshot
- PssNtFreeRemoteSnapshot
- PssNtFreeSnapshot
- PssNtFreeWalkMarker
- PssNtQuerySnapshot
- PssNtValidateDescriptor
- PssNtWalkSnapshot
- qsort
- RtlAbsoluteToSelfRelativeSD
- RtlAcquirePebLock
- RtlAcquirePrivilege
- RtlAcquireSRWLockExclusive
- RtlAcquireSRWLockShared
- RtlActivateActivationContextEx
- RtlActivateActivationContextUnsafeFast
- RtlAddAccessAllowedAce
- RtlAddAccessAllowedAceEx
- RtlAddAccessAllowedObjectAce
- RtlAddAccessDeniedAce
- RtlAddAccessDeniedAceEx
- RtlAddAccessDeniedObjectAce
- RtlAddAce
- RtlAddAuditAccessAce
- RtlAddAuditAccessAceEx
- RtlAddAuditAccessObjectAce
- RtlAddMandatoryAce
- RtlAddResourceAttributeAce
- RtlAddScopedPolicyIDAce
- RtlAddSIDToBoundaryDescriptor
- RtlAllocateActivationContextStack
- RtlAllocateAndInitializeSid
- RtlAllocateHandle
- RtlAllocateHeap
- RtlAnsiCharToUnicodeChar
- RtlAnsiStringToUnicodeString
- RtlAppendUnicodeStringToString
- RtlAppendUnicodeToString
- RtlAreAllAccessesGranted
- RtlAreAnyAccessesGranted
- RtlAreBitsSet
- RtlBarrier
- RtlCapabilityCheck
- RtlCaptureContext
- RtlCaptureStackBackTrace
- RtlCharToInteger
- RtlCheckTokenCapability

- RtlCheckTokenMembershipEx
- RtlCleanUpTEBLangLists
- RtlClearBits
- RtlCompactHeap
- RtlCompareMemory
- RtlCompareUnicodeString
- RtlCompareUnicodeStrings
- RtlConvertSidToUnicodeString
- RtlConvertToAutoInheritSecurityObject
- RtlCopyContext
- RtlCopySid
- RtlCopyUnicodeString
- RtlCreateAcl
- RtlCreateBoundaryDescriptor
- RtlCreateEnvironmentEx
- RtlCreateHeap
- RtlCreateProcessParametersEx
- RtlCreateProcessParametersWithTemplate
- RtlCreateSecurityDescriptor
- RtlCreateTagHeap
- RtlCreateTimer
- RtlCreateTimerQueue
- RtlCreateUnicodeString
- RtlCreateUnicodeStringFromAsciiz
- RtlCreateUserFiberShadowStack
- RtlCreateUserStack
- RtlCultureNameToLCID
- RtlCutoverTimeToSystemTime
- RtlDeactivateActivationContextUnsafeFast
- RtlDecodePointer
- RtlDecodeSystemPointer
- RtlDefaultNpAcl
- RtlDeleteAce
- RtlDeleteBoundaryDescriptor
- RtlDeleteCriticalSection
- RtlDeleteElementGenericTableAvl
- RtlDeleteSecurityObject
- RtlDeleteTimer
- RtlDeleteTimerQueueEx
- RtlDeregisterWaitEx
- RtlDeriveCapabilitySidsFromName
- RtlDestroyEnvironment
- RtlDestroyHeap
- RtlDestroyProcessParameters
- RtlDetermineDosPathNameType\_U
- RtlDllShutdownInProgress
- RtlDnsHostNameToComputerName
- RtlDosApplyFileIsolationRedirection\_Ustr
- RtlDosPathNameToNtPathName\_U
- RtlDosPathNameToNtPathName\_U\_WithStatus
- RtlDosPathNameToRelativeNtPathName\_U

- RtlDosPathNameToRelativeNtPathName\_U\_WithStatus
- RtlDosSearchPath\_Ustr
- RtlDowncaseUnicodeString
- RtlEncodePointer
- RtlEnterCriticalSection
- RtlEnumerateGenericTableAvl
- RtlEqualPrefixSid
- RtlEqualSid
- RtlEqualUnicodeString
- RtlExitUserProcess
- RtlExitUserThread
- RtlExpandEnvironmentStrings
- RtlExpandEnvironmentStrings\_U
- RtlFindAceByType
- RtlFindClearBitsAndSet
- RtlFindMessage
- RtlFirstFreeAce
- RtlFlsAlloc
- RtlFlsFree
- RtlFlsGetValue
- RtlFlsSetValue
- RtlFlushSecureMemoryCache
- RtlFormatCurrentUserKeyPath
- RtlFormatMessageEx
- RtlFreeActivationContextStack
- RtlFreeAnsiString
- RtlFreeHandle
- RtlFreeHeap
- RtlFreeSid
- RtlFreeUnicodeString
- RtlFreeUserFiberShadowStack
- RtlFreeUserStack
- RtlGetAce
- RtlGetActiveActivationContext
- RtlGetAppContainerNamedObjectPath
- RtlGetAppContainerParent
- RtlGetAppContainerSidType
- RtlGetControlSecurityDescriptor
- RtlGetCurrentDirectory\_U
- RtlGetCurrentProcessorNumberEx
- RtlGetCurrentServiceSessionId
- RtlGetCurrentTransaction
- RtlGetCurrentUmsThread
- RtlGetDaclSecurityDescriptor
- RtlGetDeviceFamilyInfoEnum
- RtlGetEnabledExtendedFeatures
- RtlGetExePath
- RtlGetExtendedContextLength2
- RtlGetExtendedFeaturesMask
- RtlGetFileMUIPath
- RtlGetFullPathName\_U

- RtlGetFullPathName\_UEx
- RtlGetFullPathName\_UstrEx
- RtlGetGroupSecurityDescriptor
- RtlGetInterruptTimePrecise
- RtlGetLastNtStatus
- RtlGetLastWin32Error
- RtlGetLocaleFileMappingAddress
- RtlGetNativeSystemInformation
- RtlGetNtSystemRoot
- RtlGetOwnerSecurityDescriptor
- RtlGetPersistedStateLocation
- RtlGetProcessHeaps
- RtlGetProcessPreferredUILanguages
- RtlGetProductInfo
- RtlGetSaclSecurityDescriptor
- RtlGetSearchPath
- RtlGetSecurityDescriptorRMControl
- RtlGetSystemPreferredUILanguages
- RtlGetSystemTimePrecise
- RtlGetThreadErrorMode
- RtlGetThreadPreferredUILanguages
- RtlGetUILanguageInfo
- RtlGetUserPreferredUILanguages
- RtlGetVersion
- RtlGuardCheckLongJumpTarget
- RtlGUIDFromString
- RtlIdentifierAuthoritySid
- RtlIdnToAscii
- RtlIdnToNameprepUnicode
- RtlIdnToUnicode
- RtlImageDirectoryEntryToData
- RtlImageNtHeader
- RtlImageNtHeaderEx
- RtlImpersonateSelf
- RtlInitAnsiString
- RtlInitAnsiStringEx
- RtlInitBarrier
- RtlInitializeCriticalSection
- RtlInitializeCriticalSectionAndSpinCount
- RtlInitializeCriticalSectionEx
- RtlInitializeExtendedContext2
- RtlInitializeGenericTableAvl
- RtlInitializeHandleTable
- RtlInitializeSid
- RtlInitializeSRWLock
- RtlInitString
- RtlInitUnicodeString
- RtlInitUnicodeStringEx
- RtlInsertElementGenericTableAvl
- RtlIntegerToChar
- RtlIntegerToUnicodeString

- RtlIsCapabilitySid
- RtlIsCurrentProcess
- RtlIsDosDeviceName\_U
- RtlIsMultiSessionSku
- RtlIsMultiUsersInSessionSku
- RtlIsNormalizedString
- RtlIsPackageSid
- RtlIsParentOfChildAppContainer
- RtlIsProcessorFeaturePresent
- RtlIsStateSeparationEnabled
- RtlIsValidHandle
- RtlKnownExceptionFilter
- RtlLCIDToCultureName
- RtlLcidToLocaleName
- RtlLeaveCriticalSection
- RtlLengthRequiredSid
- RtlLengthSecurityDescriptor
- RtlLengthSid
- RtlLoadString
- RtlLocaleNameToLcid
- RtlLocateExtendedFeature
- RtlLocateLegacyContext
- RtlLockHeap
- RtlLookupElementGenericTableAvl
- RtlLookupFunctionEntry
- RtlMakeSelfRelativeSD
- RtlMapGenericMask
- RtlMultiByteToUnicodeN
- RtlNewSecurityObject
- RtlNewSecurityObjectEx
- RtlNewSecurityObjectWithMultipleInheritance
- RtlNormalizeString
- RtlNtStatusToDosError
- RtlNtStatusToDosErrorNoTeb
- RtlOemStringToUnicodeString
- RtlOpenCurrentUser
- RtlpCheckDynamicTimeZoneInformation
- RtlpCreateProcessRegistryInfo
- RtlPcToFileHeader
- RtlpGetLCIDFromLangInfoNode
- RtlpGetNameFromLangInfoNode
- RtlpGetSystemDefaultUILanguage
- RtlpInitializeLangRegistryInfo
- RtlpIsQualifiedLanguage
- RtlpLoadMachineUIByPolicy
- RtlpLoadUserUIByPolicy
- RtlpMergeSecurityAttributeInformation
- RtlpMuiFreeLangRegistryInfo
- RtlpQueryDefaultUILanguage
- RtlPrefixString
- RtlPrefixUnicodeString



- RtlProcessFlsData
- RtlpTimeFieldsToTime
- RtlpTimeToTimeFields
- RtlPublishWnfStateData
- RtlQueryActivationContextApplicationSettings
- RtlQueryEnvironmentVariable
- RtlQueryEnvironmentVariable\_U
- RtlQueryHeapInformation
- RtlQueryInformationAcl
- RtlQueryInformationActivationContext
- RtlQueryPackageClaims
- RtlQueryPackageIdentityEx
- RtlQueryProtectedPolicy
- RtlQueryRegistryValuesEx
- RtlQuerySecurityObject
- RtlQueryWnfStateData
- RtlQueueWorkItem
- RtlRaiseCustomSystemEventTrigger
- RtlRaiseException
- RtlRaiseStatus
- RtlRandomEx
- RtlReAllocateHeap
- RtlRegisterWait
- RtlReleaseActivationContext
- RtlReleasePath
- RtlReleasePebLock
- RtlReleasePrivilege
- RtlReleaseRelativeName
- RtlReleaseSRWLockExclusive
- RtlReleaseSRWLockShared
- RtlReportSilentProcessExit
- RtlRunOnceBeginInitialize
- RtlRunOnceComplete
- RtlRunOnceExecuteOnce
- RtlRunOnceInitialize
- RtlSelfRelativeToAbsoluteSD
- RtlSelfRelativeToAbsoluteSD2
- RtlSetControlSecurityDescriptor
- RtlSetCurrentDirectory\_U
- RtlSetCurrentTransaction
- RtlSetDaclSecurityDescriptor
- RtlSetEnvironmentStrings
- RtlSetEnvironmentVar
- RtlSetEnvironmentVariable
- RtlSetExtendedFeaturesMask
- RtlSetGroupSecurityDescriptor
- RtlSetHeapInformation
- RtlSetInformationAcl
- RtlSetLastWin32Error
- RtlSetLastWin32ErrorAndNtStatusFromNtStatus
- RtlSetOwnerSecurityDescriptor

- RtlSetProcessPreferredUILanguages
- RtlSetProtectedPolicy
- RtlSetSaclSecurityDescriptor
- RtlSetSecurityDescriptorRMControl
- RtlSetSecurityObject
- RtlSetSecurityObjectEx
- RtlSetThreadErrorMode
- RtlSetThreadPreferredUILanguages
- RtlSetUnhandledExceptionFilter
- RtlSetUserValueHeap
- RtlSizeHeap
- RtlSleepConditionVariableCS
- RtlSleepConditionVariableSRW
- RtlStringFromGUID
- RtlStringFromGUIDEx
- RtlSubAuthorityCountSid
- RtlSubAuthoritySid
- RtlSubscribeWnfStateChangeNotification
- RtlTimeFieldsToTime
- RtlTimeToTimeFields
- RtlTryAcquirePebLock
- RtlUnhandledExceptionFilter
- RtlUnicodeStringToAnsiString
- RtlUnicodeStringToInteger
- RtlUnicodeStringToOemString
- RtlUnicodeToMultiByteN
- RtlUnicodeToMultiByteSize
- RtlUnicodeToOemN
- RtlUnicodeToUTF8N
- RtlUnlockHeap
- RtlUnsubscribeWnfNotificationWaitForCompletion
- RtlUnsubscribeWnfStateChangeNotification
- RtlUpcaseUnicodeChar
- RtlUpcaseUnicodeString
- RtlUpdateTimer
- RtlUTF8ToUnicodeN
- RtlValidAcl
- RtlValidateHeap
- RtlValidRelativeSecurityDescriptor
- RtlValidSecurityDescriptor
- RtlValidSid
- RtlVerifyVersionInfo
- RtlVirtualUnwind
- RtlWaitOnAddress
- RtlWalkHeap
- RtlWow64EnableFsRedirectionEx
- RtlWow64GetProcessMachines
- RtlWow64GetSharedInfoProcess
- RtlWow64IsWowGuestMachineSupported
- RtlWow64PopCrossProcessWorkFromFreeList
- RtlWow64PushCrossProcessWorkOntoFreeList

- RtlWow64PushCrossProcessWorkOntoWorkList
- RtlWow64RequestCrossProcessHeavyFlush
- RtlxAnsiStringToUnicodeSize
- RtlxOemStringToUnicodeSize
- RtlxUnicodeStringToAnsiSize
- RtlxUnicodeStringToOemSize
- SbSelectProcedure
- sqrt
- strchr
- \_stricmp
- \_strlwr
- strncat
- strncat\_s
- \_strnicmp
- swprintf\_s
- toupper
- tolower
- TpAllocCleanupGroup
- TpAllocIoCompletion
- TpAllocPool
- TpAllocTimer
- TpAllocWait
- TpAllocWork
- TpCallbackMayRunLong
- TpCancelAsyncIoOperation
- TpCaptureCaller
- TpCheckTerminateWorker
- TpPostWork
- TpQueryPoolStackInformation
- TpReleaseIoCompletion
- TpReleaseWait
- TpReleaseWork
- TpSetPoolMinThreads
- TpSetPoolStackInformation
- TpSetWait
- TpSimpleTryPost
- TpStartAsyncIoOperation
- TpWaitForIoCompletion
- TpWaitForWait
- \_ui64tow\_s
- VerSetConditionMask
- \_vsnprintf
- \_vsnwprintf
- vswprintf\_s
- wcscat\_s
- wcschr
- wcscmp
- wcscpy\_s
- wcscspn
- \_wcsicmp
- \_wcslwr

- wcsncmp
- wcsncpy\_s
- \_wcsnicmp
- wcsnlen
- wcsrchr
- wcssp
- wcsstr
- wcstoul
- WinSqmAddToStreamEx
- WinSqmEndSession
- WinSqmIncrementDWORD
- WinSqmIsOptedIn
- WinSqmSetDWORD
- WinSqmSetString
- WinSqmStartSession
- \_wtoi
- \_wtol
- ZwClose
- ZwCreateKey
- ZwOpenKey
- ZwQueryInformationToken
- ZwQueryLicenseValue
- ZwQueryValueKey
- ZwQueryWnfStateData
- ZwQueryWnfStateNameInformation
- ZwSetValueKey
- ZwUpdateWnfStateData
- N/A

#### 5.1.4 Functions Imported From Kernel32.dll

- CloseHandle
- CopyFileA
- CreateDirectoryA
- CreateDirectoryW
- CreateFileA
- CreateProcessA
- DeleteCriticalSection
- EnterCriticalSection
- FindResourceA
- FreeLibrary
- GetComputerNameW
- GetCurrentDirectoryA
- GetExitCodeProcess
- GetFileAttributesA
- GetFileAttributesW
- GetFileSize
- GetFileSizeEx
- GetFullPathNameA
- GetModuleFileNameA
- GetModuleHandleA

- GetProcAddress
- GetProcessHeap
- GetStartupInfoA
- GetTempPathW
- GetWindowsDirectoryW
- GlobalAlloc
- GlobalFree
- HeapAlloc
- HeapFree
- InitializeCriticalSection
- IsBadReadPtr
- LeaveCriticalSection
- LoadLibraryA
- LoadResource
- LocalFileTimeToFileTime
- LockResource
- MultiByteToWideChar
- OpenMutexA
- ReadFile
- SetCurrentDirectoryA
- SetCurrentDirectoryW
- SetFileAttributesW
- SetFilePointer
- SetFileTime
- SetLastError
- SizeofResource
- Sleep
- SystemTimeToFileTime
- TerminateProcess
- VirtualAlloc
- VirtualFree
- VirtualProtect
- WaitForSingleObject
- WriteFile