Malware Analysis: WannaCry (The Encryptor)

Introduction

This paper will analyze a well known piece of malware WannaCry. The malware was downloaded from https://github.com/ytisf/theZoo/tree/master/malware/Binaries/Ransomware.WannaCry. Evidently, in the link it says this is Ransomware, but in the analysis we will prove this. The malware has two parts to it:

1. The Worm
2. The Encryptor

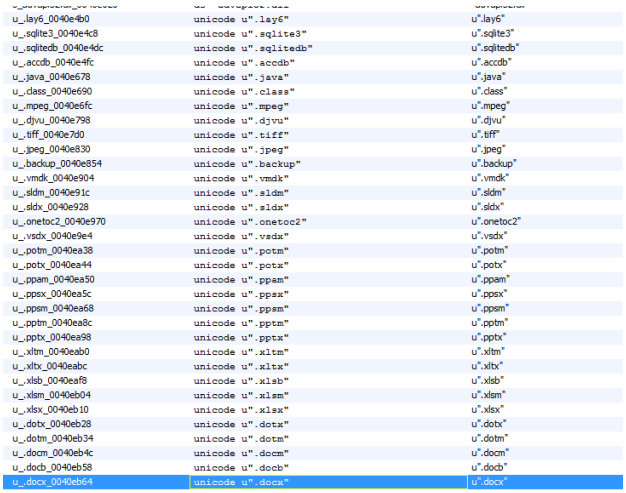
After going through the analysis, the downloaded sample contains only the encryptor. It'll be evident as we continue with the analysis that the sample's goal is to encrypt, persist, and get money from victims. At the end, I'll reference some vulnerabilities found in the malware that allow for the recovery of encrypted data in special circumstances.

Unpacking

I initially loaded the malware into CFF Explorer. Looking through the section's headers, there were no signs of packing. Analyzing the PE headers revealed standard section names indicating that the malware is likely not entirely packed with common packers like UPX.

Strings of Interest

Using Ghidra, I was able to find a few clues supporting the fact that this is ransomware. There are several extensions listed in the executable file. These are likely file types targeted by the malware authors. The file extensions ".sqlite3," for databases, ".vmdk," for virtual machines, etc. These could contain important information, and therefore are the primary file types targeted by WannaCry. A catalog of these extensions can be found in figure 1.



u_lay6_0040e4b0	unicode u".lay6"	u".lay6"
u_sqlite3_0040e4c8	unicode u".sqlite3"	u".sqlite3"
u_sqlitedb_0040e4dc	unicode u".sqlitedb"	u".sqlitedb"
u_accdb_0040e4fc	unicode u".accdb"	u".accdb"
u_java_0040e678	unicode u".java"	u".java"
u_class_0040e690	unicode u".class"	u".class"
u_mpeg_0040e6fc	unicode u".mpeg"	u".mpeg"
u_djvu_0040e798	unicode u".djvu"	u".djvu"
u_tiff_0040e7d0	unicode u".tiff"	u".tiff"
u_jpeg_0040e830	unicode u".jpeg"	u".jpeg"
u_backup_0040e854	unicode u".backup"	u".backup"
u_vmdk_0040e904	unicode u".vmdk"	u".vmdk"
u_sldm_0040e91c	unicode u".sldm"	u".sldm"
u_sldx_0040e928	unicode u".sldx"	u".sldx"
u_onetoc2_0040e970	unicode u".onetoc2"	u".onetoc2"
u_vsdx_0040e9e4	unicode u".vsdx"	u".vsdx"
u_potm_0040ea38	unicode u".potm"	u".potm"
u_potx_0040ea44	unicode u".potx"	u".potx"
u_ppam_0040ea50	unicode u".ppam"	u".ppam"
u_ppsx_0040ea5c	unicode u".ppsx"	u".ppsx"
u_ppsm_0040ea68	unicode u".ppsm"	u".ppsm"
u_pptm_0040ea8c	unicode u".pptm"	u".pptm"
u_pptx_0040ea98	unicode u".pptx"	u".pptx"
u_xltm_0040eab0	unicode u".xltm"	u".xltm"
u_xltx_0040eabc	unicode u".xltx"	u".xltx"
u_xlsb_0040eaf8	unicode u".xlsb"	u".xlsb"
u_xlsm_0040eb04	unicode u".xlsm"	u".xlsm"
u_xlsx_0040eb10	unicode u".xlsx"	u".xlsx"
u_dotx_0040eb28	unicode u".dotx"	u".dotx"
u_dotm_0040eb34	unicode u".dotm"	u".dotm"
u_docm_0040eb4c	unicode u".docm"	u".docm"
u_docb_0040eb58	unicode u".docb"	u".docb"
u_docx_0040eb64	unicode u".docx"	u".docx"

Figure 1: This screenshot shows the file extension strings found in the executable using the search for strings feature in Ghidra.

Furthermore, there were filenames found, and interesting strings that may be significant in other functions. The command “icacls ./grant Everyone…” is used so the malware is able to gain access to all files in a recursive and silent manner.

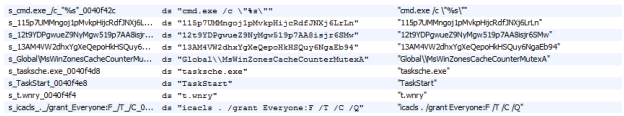


Figure 2: This screenshot shows miscellaneous strings and commands that are of particular interest to the function of the malware as we dissect it later..

Lastly, the two strings below were seen in the malware as a sort of attribution to existing compression/decompression libraries. The strings mentioning “inflate 1.1.3” confirm that the malware is using a data compression library from Zlib. This is likely to decompress some embedded resources at runtime.

- "- unzip 0.15 Copyright 1998 Gilles Vollant "
- " inflate 1.1.3 Copyright 1995-1998 Mark Adler "

Imported Methods

I next noticed a lot of function names. These are dynamically loaded functions that would modify, create, or delete files. It’s likely that the file modifying functions are used to obfuscate analysis. It’s also likely that these modifications are used to delete the original file and keep only the encrypted files on the disk. All of this would support the function of ransomware.

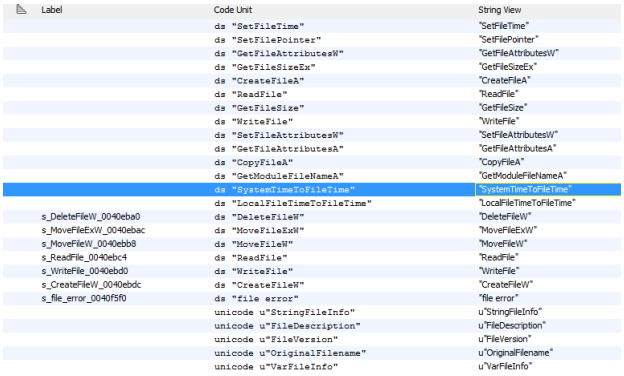


Figure 3: This screenshot shows the file function names found in the executable using the search for strings feature in Ghidra.

There were also function names dynamically loaded for cryptographic operations. These strings are of special interest because it gives an idea of what library and methods WannaCry is using to encrypt files.

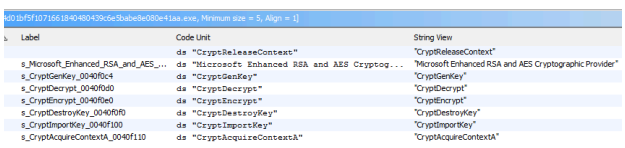


Figure 4: This screenshot shows the cryptography function names found in the executable using the search for strings feature in Ghidra.

Scrolling further revealed memoryapi.h functions. These functions are generally used to unpack or reveal a second portion of the malware's purpose at runtime. This likely means there's a hidden layer of functionality in this malware's payload.

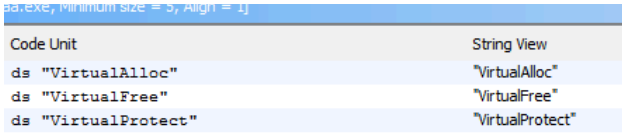


Figure 5: This screenshot shows the memoryapi.h function names found in the executable using the search for strings feature in Ghidra.

Reversed Engineered Disassembly

Figure 6: This screenshot shows a disassembled WinMain() function from WannaCry in Ghidra. The function InitializeRansomware is emphasized.

Using Ghidra, I found the WinMain() function in the entry point identified by Ghidra (Figure 6). By examining the nature of this malware up to this point, I thought the most interesting method to disassemble and analyze would be where it sets up the public cryptography keys. In doing so, I identified the function InitializeRansomware() below.

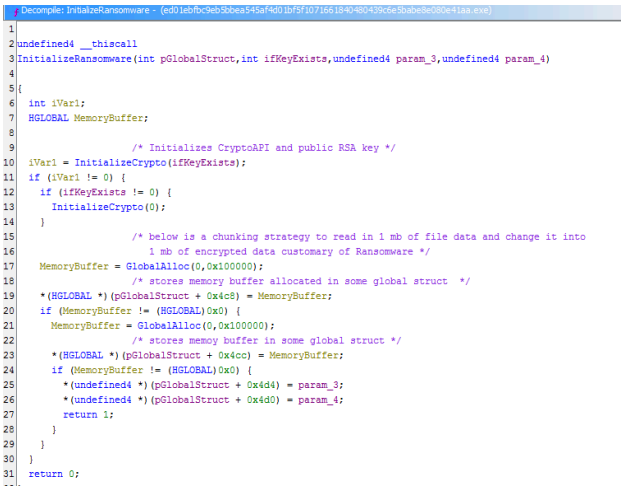


Figure 7: This screenshot shows a fully disassembled and analyzed InitializeRansomware() function from WannaCry in Ghidra. Annotated C-code is provided to assist with claims.

The function can be broken down into two main parts:

1. Importing a public RSA key
2. Memory allocation for a chunking strategy to encrypt files.

The first part is facilitated by the function InitializeCrypto(). Depending on the argument passed, it will either import the public RSA key from a file or use CryptImportKey() from Windows Cryptography API.

The memory allocation works by creating two 1 mb chunks. One chunk will contain the original data, and the other will contain the encrypted data. This ensures consistent memory efficiency as the malware won't try to encrypt large files all at once.

Figure 8: This screenshot shows a disassembled InitializeCrypto() function from WannaCry in Ghidra.

Network Traffic

By running the executable, I was able to learn how it communicates to a central server, and I was able to find some interesting packets being sent to some external IP through wireshark. Through further research into this specific malware and the port it uses, this is likely trying to connect to a TOR network [1].

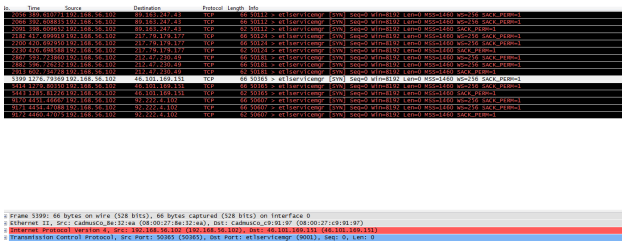


Figure 9: This screenshot shows the malware trying to reach an external ip address on port 9001 likely on the TOR network [1].

Upon interaction with the GUI that popped up, I noticed that a DNS request was sent to the REMNUX for dns.msfncsi.com, the network connectivity status indicator. This is a microsoft network diagnostics tool, but it shows that the malware attempts to communicate over the network, likely with some central server [1]. This is characteristic of most Ransomware.

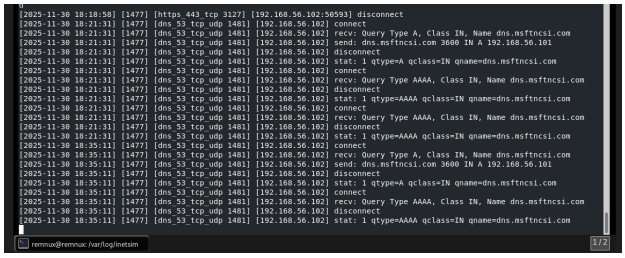


Figure 10: This screenshot shows a network request for *dns.msftnsci.com* which is the windows network status connectivity indicator.

Registry

The malware uses an interesting persistence method, so it will attempt to place itself in the registry for persistence, but upon failure it has failsafes it uses.. For my experiment the malware ended up installing itself into the HKCU/Windows/Microsoft/CurrentVersion/Run registry as seen below. The malware creates a persistence entry in the Run key. The value name at first may appear randomized, but when analyzing the subroutines in WinMain() we find a call to GetComputerName(). This returns the computer name which then goes through a unique id generation algorithm so each computer maps to a unique identifier which creates the value name.

Figure 11: This screenshot shows a command run by a subprocess of the malware. It adds a value to the HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Files Modified

The malware creates several files, but due to the nature of Ransomware the most important and obvious is the encryption of files that end in ".txt, .java, .docx, .sqllite3, etc….: This uses the chunking strategy mentioned earlier to optimize memory usage. It will append a .WNCRY file extension to each encrypted file.

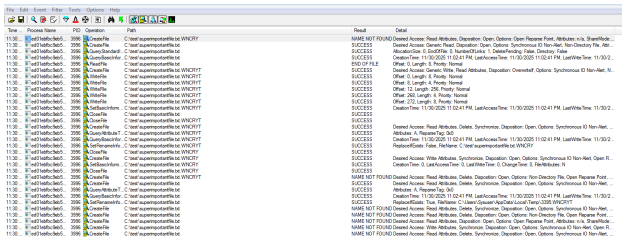


Figure 12: This screenshot shows one of the files I made being encrypted by the source executable. The original file is “superimportantfile.txt” and it is later replaced with “superimportantfile.txt.WNCRY.”

Furthermore, the malware will also create a lot of processes. Each one has a unique task, but they all originate from the Ransomware.Wannacry file. Interestingly, the GUI that pops up is likely a part of the packed payload hidden in the original file. There are also other processes like taskse.exe and taskdl.exe that seem to be responsible for querying the registry and maintaining persistence among other possible things that require further analysis.

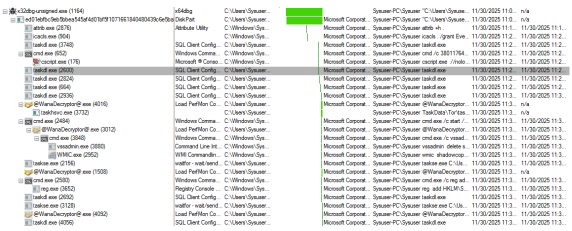


Figure 13: This screenshot shows the process tree on Procmon. It shows the subprocesses created by the main malware executable starting with “ed01…..exe.”

Debugger Analysis

Running the malware in a debugger, I wanted to see what the VIrtualAlloc function was unpacking. From looking at the hex dump, we can see that it's unpacking a portable executable. The common PE header is shown in the dump after setting a breakpoint at VirtualAlloc.. By running the executable in x32dbg, we’re able to confirm the unpacking behaviour of WannaCry to dynamically unload a hidden payload.

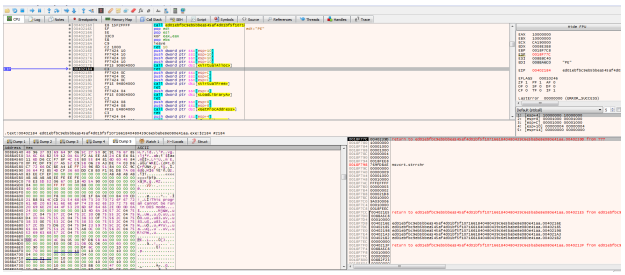


Figure 14: This screenshot shows a breakpoint at VirtualAlloc and the hex dump that contains a PE Header confirming that the malware unpacks an executable.

Cleaning an Infected Machine

In order to disinfect a machine from this specific version of the malware, you would first need to disconnect it from the internet. This prevents the malware from sending encryption keys to the central command center utilized by the authors.

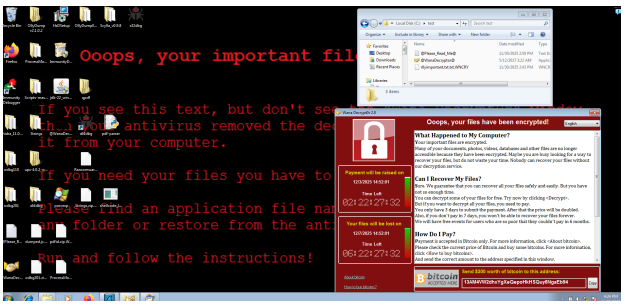


Figure 15: This screenshot shows the result of running the malware. The wallpaper changes, and there's a GUI for interacting with the authors. On the top right, I show an encrypted file I prepared before running the malware to see its encryption.

The next goal is to eradicate the malware's persistence methods and active processes. In order to accomplish this, we'll terminate the GUI process, use regedit to delete any altered or new registry keys, and remove the executable files of the malware.

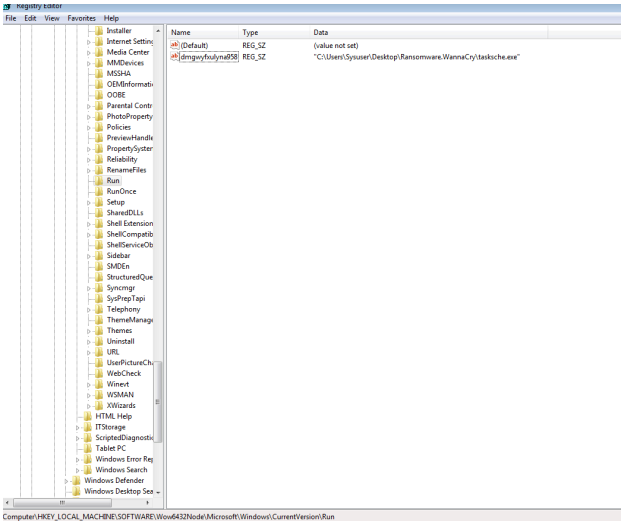


Figure 16: This screenshot shows the location of the persistence method. Earlier we observed a command to add it to a different registry, but since this is 32 bit malware running on a 64 bit windows operating system it was redirected to the Wow6432Node folder.

Unfortunately, a user will likely be unable to recover their files. If the user has a backup, we'll simply use those files to recover the files. If they don't, it's likely not possible to decrypt these files without the private key (the authors have this). There is some hope however for devices that haven't been rebooted since being infected. There was a group of researchers that found a vulnerability in the windows cryptography api. CryptoDestroyKey() did not actually delete the key; it only deletes a handler for the key, so the two prime numbers p and q used to create the public and private key still exist on the drive as long as the computer hasn't restarted. They developed a tool to scan for these two prime numbers and

calculate the public-private key pair [2]. This technique could be used on infected systems that haven't been rebooted yet. The tool is known as WannaKey.

In short, if the user wishes to restore the current version of their system (post attack), I would suggest implementing the steps above. On the other hand, a user with an externally backed up image could reset their computer from that point. If the user only backed up their system on their computer, then unfortunately we wouldn't be able to recover the data. This is because the malware deletes all shadow copies on the drive. I determined this when the windows system prompted permissions to run the command to delete all shadow copies on the computer.

References

[1] D. Sajan, 'A Comprehensive Analysis of WannaCry Ransomware', *INTERNATIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT*, vol. 08, pp. 1–5, 08 2024.

[2] A. Greenberg, "A WannaCry flaw could help some victims get files back," Wired, https://www.wired.com/2017/05/wannacry-flaw-help-windows-xp-victims-get-files-back/ (accessed Dec. 2, 2025).