



UNIVERSITY OF PIRAEUS – DEPARTMENT OF INFORMATICS
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Distributed Systems, Security and Emerging Information Technologies»

ΠΜΣ «Κατανεμημένα Συστήματα, Ασφάλεια και Αναδυόμενες Τεχνολογίες Πληροφορίας»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	Distributed Advanced Shellcode Detection System Κατανεμημένο Σύστημα Εντοπισμού Σύγχρονου Shellcode
Student's Name-Surname: Όνοματεπώνυμο φοιτητή:	Dimitrios Tsilis Δημήτριος Τσίλης
Father's Name: Πατρώνυμο:	Evangelos Ευάγγελος
Student's ID: Αριθμός Μητρώου:	ΜΠΚΣΑ - 20006
Supervisor: Επιβλέπων:	Panayiotis Kotzanikolaou, Associate Professor Παναγιώτης Κοτζανικολάου Αναπληρωτής Καθηγητής

March 2024/ Μάρτιος 2024

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Christos Douligeris

Professor

Χρήστος Δουλιγέρης
Καθηγητής

Panayiotis Kotzanikolaou

Associate Professor

Παναγιώτης Κοτζανικολάου
Αναπληρωτής Καθηγητής

Michael Psarakis

Associate Professor

Μιχαήλ Ψαράκης
Αναπληρωτής Καθηγητής

Acknowledgments

I would like to warmly thank my supervisor, Associate Professor Mr. Panagiotis Kotzanikolaou, for the support and guidance he provided me during the completion of my master thesis. I believe that the experience gained from this thesis is not only valuable but also essential for my future career.

I am grateful to Dr. Dimitrios Glynos for generously sharing his experience, contributing significantly to this master's thesis. His expertise and thoughtful suggestions have significantly enriched the quality of this research.

Additionally, I extend my gratitude to my parents and siblings, who are always by my side, no matter what I do.

Lastly, to my second family – my friends – who support the choices I make in my life.

Περίληψη

Τα Συστήματα Εντοπισμού Διαδικτυακής Διείσδυσης είναι εξειδικευμένες συσκευές υλικού ή προγράμματα λογισμικού, τα οποία έχουν σχεδιαστεί και κατασκευαστεί με σκοπό τον εντοπισμό κακόβουλης κίνησης στο δίκτυο. Οι τεχνικές που χρησιμοποιούνται κατά τον έλεγχο των πακέτων της κίνησης στο δίκτυο είναι σε θέση να ανιχνεύσουν οποιαδήποτε δυνητική απειλή. Για αυτόν τον λόγο θεωρούνται σημαντικές, καθώς είναι το πιο κρίσιμο μέρος προκειμένου να είναι αποτελεσματικό ένα Σύστημα Εντοπισμού Δικτυακής Διείσδυσης. Οι επιτιθέμενοι, από την πλευρά τους, για να παρακάμψουν αυτά τα συστήματα, άρχισαν να δημιουργούν φορτία που δεν εντοπίζονται. Τα εν λόγω φορτία, τα οποία αποκαλούνται πολυμορφικά shellcodes, είναι σε θέση να κρυφθούν ή να μεταλλαχθούν ώστε να παρακάμψουν αυτούς τους μηχανισμούς ασφαλείας. Επομένως, σε αυτήν τη διατριβή παρουσιάζεται μία νέα Μηχανή, η οποία είναι ικανή να ανιχνεύει αυτού του είδους τα shellcodes ειδικά για 32-bit Windows λειτουργικά συστήματα σε ένα δίκτυο. Με το να προσομοιώνει στοιχεία ενός λειτουργικού συστήματος Windows, αυτή η Μηχανή αυξάνει την ανθεκτικότητά του και την αποτελεσματικότητά του σε σύγκριση με εναλλακτικές λύσεις. Επιπλέον, η εν λόγω Μηχανή ενσωματώνεται στο SEDUCE, ένα κατανεμημένο σύστημα ανίχνευσης shellcode που χρησιμοποιεί προσομοίωση CPU για τον έλεγχο της κίνησης στο δίκτυο.

Λέξεις Κλειδιά: Ασφάλεια Δικτύων, Συστήματα Εντοπισμού Δικτυακής Διείσδυσης, Πολυμορφικά Shellcodes, Εσωτερικές Λειτουργίες Windows OS

Abstract

The Network Intrusion Detection Systems (NIDS) are specialized systems which may be either hardware devices or software programs that are designed and built for detecting malicious traffic in the networks. The techniques used when inspecting the packets of a network traffic, are able to detect any potential threat. For this reason, these techniques are the most critical part for a NIDS system to be effective. The attackers by their side to circumvent those systems, started to craft undetectable payloads. These payloads, so-called polymorphic shellcodes, are able to either hide or mutate themselves and may bypass these security mechanisms. As such, in this thesis a new Engine is introduced which is capable of detecting this kind of shellcodes specifically for 32-bit Windows operating systems in a network environment. By emulating components of a Windows operating system, this engine improves its resilience and effectiveness in comparison to alternative solutions. Additionally, this Engine is integrated in SEDUCE, a distributed shellcode detection system using CPU emulation for inspecting the network traffic.

Key Words: Network Security, Network Intrusion Detection Systems, Polymorphic Shellcodes, Windows OS Innerworkings

Table of Contents

1. Introduction.....	7
1.1 Buffer Overflow Attacks.....	7
1.1 General.....	7
1.2 Technical Aspect.....	7
1.2.1 Registers.....	8
1.2.2 The Stack.....	8
1.2.3 Program Functions.....	10
1.2.4 Stack Overflow.....	11
1.2.5 NOP Sleds.....	12
1.3 Shellcode Insights.....	13
1.4 Metamorphic and Polymorphic Shellcodes.....	13
1.5 Thesis Contribution.....	14
1.6 Thesis Structure.....	14
2. Field Research.....	15
2.1 Network Intrusion Detection System (NIDS).....	15
2.2 Detection Techniques of NIDS.....	15
2.2.1 Signature-based Detection.....	15
2.2.2 Anomaly-based Detection.....	16
2.2.3 Emulation-based Detection.....	16
2.3 Shellcode Detection Using CPU Emulation.....	16
3. Design.....	17
3.1 Windows Internals.....	17
3.2 PE and Win32 API (DLLs).....	19
3.2.1 Portable Executable (PE).....	19
3.2.2 Win32 API (DLLs).....	21
3.3 PEB and TEB structures.....	21
3.3.1 Process Environment Block.....	21
3.3.2 Thread Environment Block.....	22
3.4 PEB_LDR_DATA and LDR_MODULE.....	23
3.5 Importance of FS and GS Registers.....	24
3.5.1 Global Descriptor Table.....	25
3.5.2 Segment Registers.....	25
3.6 Overall Design.....	26
4. Implementation.....	27
4.1 Library Inclusions.....	27
4.1.1 Unicorn Engine.....	27

4.1.2 libpe	27
4.2 Basic Windows structures	27
4.3 Windows Detection Engine	34
4.3.1 Memory Layout	34
4.3.2 Helper Functions	35
4.3.3 Main functions	44
5. User's Manual	51
6. Experimental Evaluation	52
6.1 Dataset Preparation	52
6.2 Proof-of-Concept	52
6.3 Time Duration	53
7. Conclusions and Future Work	54
7.1 Conclusions	54
7.2 Future Enhancements	54
8. Bibliography - References	55

1. Introduction

A major security research issue is risen regarding the detection of polymorphic shellcodes in the network. Network traffic may include various data, which is not an easy task to distinguish benign traffic from malicious. Moreover, various NIDS solutions designed and built for this purpose, but none of them are bulletproof. Organizations or individuals keep researching on how to bypass these systems. As such, many vendors designing NIDS solutions, are trying their best to keep their systems up to date. But they are facing multiple issues. Despite of finding the optimal solution to handle the network traffic efficiently, vendors are researching for techniques capable of detecting advanced attacks.

This thesis is providing a solution which results on building a robust and effective NIDS system, especially for the shellcodes detection. The proposed solution mitigates various challenges associated with commonly used techniques for detecting malicious traffic. One basic key factor in its implementation is that it is designed based on the innerworkings of a windows operating system. As such, it provides an advantage in detecting previously unknown attacks and is less susceptible to false positives.

1.1 Buffer Overflow Attacks (BoF)

For the purposes of this thesis, a detailed explanation of what a buffer overflow attack is will be provided, as polymorphic shellcodes are predominantly utilized in such exploits. Furthermore, it should be noted that this thesis solution is not limited to detect shellcodes in such forms of attacks.

1.1 General

A buffer overflow attack is a type of security vulnerability that occurs when a program writes more data to a buffer than it can hold, as a result it overruns the buffer's boundary and overwrites adjacent memory locations. This vulnerability can cause the program to behave in an unintended way, which can be exploited by attackers.

The history of buffer overflow attacks can be tracked back to the 1960s, with early instances occurring in the form of errors in the developing process rather than intentional exploitation. However, on the 1980s and 1990s this vulnerability gained significant attention in the context of cybersecurity. In the 1988, this type of attack was utilized by the famous "internet worm" created by Robbert Tappan Morris, but it became widely recognized in the 1996 when the supervisor of the BugTraq, Aleph One, wrote an article for the electronic magazine with the title "Smashing The Stack For Fun and Profit"[1].

The fundamental idea behind a buffer overflow attack is that an attacker is able to overwrite critical program data, like return addresses, function pointers, and other control information in order to redirect the program's operations, allowing them to either execute malicious code to gain unauthorized access to the systems or instantly terminate the program's execution resulting to a denial of service.

1.2 Technical Aspect of BoF

In this chapter, buffer overflow attacks will be described in IA-32 architectures.

1.2.1 Registers

The x86 processors have eight (8) 32-bit general purpose registers, as illustrated in Figure 1.[2] The naming of the registers are mostly historical. For example, EAX used to be called the accumulator since it was used for arithmetic operations. From these registers, the stack pointer (ESP) and the base pointer (EBP) are special purpose registers associated with the stack's boundaries.

For the EAX, EBX, ECX, and EDX registers can be subdivided into smaller units. For instance, the least significant 2 bytes of EAX can be used as a 16-bit register called AX. The least significant byte of AX can be used as a single 8-bit register called AL, while the most significant byte of AX can be used as a single 8-bit register called AH. The use of sub-registers is when dealing with data. For instance, when working with different portions of data within the larger 32-bit registers.[2]

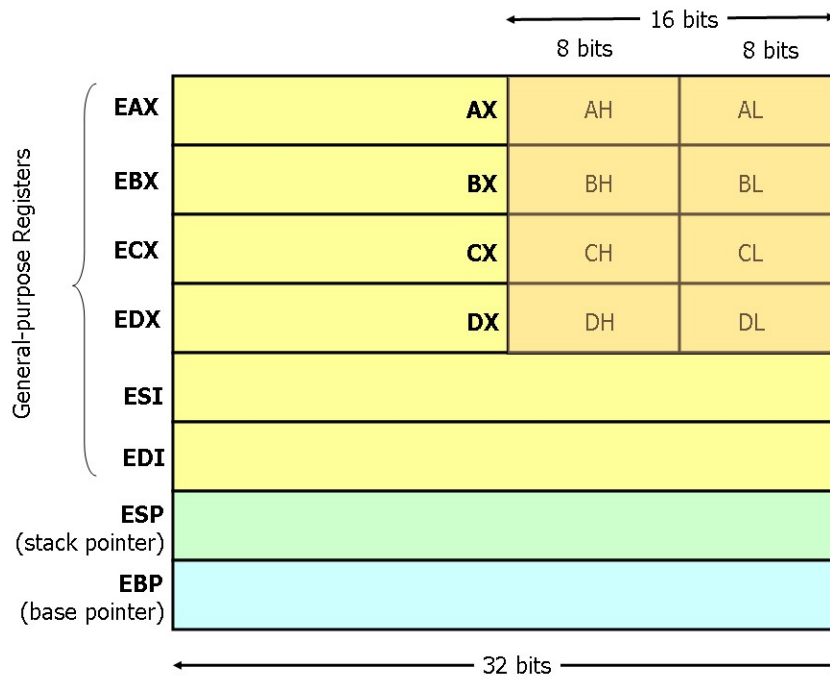


Figure 1 General-purpose Registers

1.2.2 The Stack

The stack operates as a Last-In-First-Out (LIFO) data structure, meaning that the most recent element placed onto the stack, also must be the first element to be removed.[3] The extended stack pointer (ESP) register points to the top of it, defining its boundaries. The stack-specific instructions, PUSH and POP, utilize ESP to determine the stack's location in memory. The following figure illustrates the result of ESP register after executing the instructions (Figure 2, 3):


```
push 3
push 2
push 1
```

Figure 2 Push instructions executed

Address	Value
0xfaff0008	3
0xfaff0004	2
0xfaff0000	1

← ESP pointing to this address

Figure 3 Result of ESP register when push

And the following Figures 4,5 illustrates the result of ESP when the POP instruction is encountered:

```
pop 3
pop 2
```

Figure 4 Pop instructions executed

Address	Value
0xfaff0008	3
0xfaff0004	2
0xfaff0000	1

← ESP pointing to this address

Figure 5 Result of ESP register when pop

As it is observed, the pop instruction only decrements its value without writing or erasing data.

Moreover, there is the extended base pointer (EBP) register. EBP register is used to calculate an address relative to another address and along with ESP, plays a crucial role in forming the stack frame. The EBP, often referred to as the frame pointer, serves as a stable reference point within a function's stack frame. This reference point is crucial for accessing local variables and parameters efficiently. It is set up by pointing to the stack's base address. Figure 6

Address	Value
0xfaff0008	3
0xfaff0004	2
0xfaff0000	1
0xfafeffc	Base stack address

← ESP pointing to this address

← EBP pointing to this address

Figure 6 Indication of EBP register

In x86 assembly, the stack typically grows from higher to lower addresses, with each stack slot often being 4 bytes. So, if a variable is two stack slots away from the base pointer (EBP), it is accessed as [EBP + 8].

1.2.3 Program Functions

When developing a program, the creation of functions is very important. They are offering to the developers better readability and also the ability to reuse the same code. Moreover, functions allow developers having their code more manageable.

During a program's execution, when a function is invoked, the control flow is transferred to the beginning of that function and as a result, a new stack frame is allocated on the stack. This newly created stack frame holds local variables, function parameters, and control data specific to the called function. The stack pointer as mentioned above, represented by the ESP register in x86 architectures, always pointing to the top of the stack frame and it is adjusted to create space for the newly allocated stack frame. Furthermore, the return address, indicating the next instruction to execute after the function completes its tasks, is pushed onto the stack. The return address is represented by the EIP register. [3]

It's important to note that, before any instructions within the function are executed, the function prologue is taking place. The prologue is a set of instructions at the beginning of a function, typically responsible for setting up the function's stack frame. Figure 7

<code>push ebp</code>	; Save the previous value of EBP
<code>mov esp, ebp</code>	; Set up the new base pointer

Figure 7 Prologue of a function

During the execution of the function, the return address remains on the stack, and local variables, function parameters, or control-specific data may be manipulated to accomplish the function's tasks.

Once the function completes its tasks and is ready to return, the result, if any, is often stored in a designated register. Subsequently, the stack frame of the current function is then popped off the stack, freeing up the allocated space. Simultaneously, the return address is retrieved from the function's stack frame, allowing the program to resume execution from the correct point in the calling code. This task is called epilogue (Figure 8). More specifically, the process of the epilogue involves saving the previous value of the base pointer (EBP) and the new base pointer is set to the current value of the stack pointer (ESP).

<code>pop ebp</code>	; Restore the previous value of EBP
<code>ret</code>	; Return to the calling function

Figure 8 Epilogue of a function

In a buffer overflow vulnerability, an attacker, taking into account the previously mentioned details, will try to manipulate the return address in a way that redirects the program execution to their intended destination. More specifically, before a function returns, they will try to take control of the return address by overwriting it with their intended data. Such scenarios may be accomplished when local variables are controlled by an attacker.

1.2.4 Stack Overflow

The following program (Figure 9) is taking as input some characters from the command line (controlled by the user). Then, it copies the content of the input in a declared array of 12-character size with the name "c". As it is observed, the vulnerable "strcpy" function is not checking the size of the given input. As a result, by giving more than 12 characters from the command line, the stack of the "vulnerableFunction" is overflowed.

```
#include <string.h>

void vulnerableFunction(char *attackerInput){
    char c[12];

    strcpy(c, attackerInput); // no bounds checking
}

int main(int argc, char **argv){
    vulnerableFunction(argv[1]);
    return 0;
}
```

Figure 9 Stack after prologue

An attacker may control the return address by giving sixteen (16) "A" characters and four (4) characters – bytes (since each character is a byte). The last four characters- bytes will overwrite the return address and point to a controlled destination. A visual representation is given when "vulnerableFunction" is initiated (Figure 10) and the result after the execution of the program by taking as input sixteen (16) "A" and 4 bytes contain the "0x0835C080" hex bytes (Figure 11).

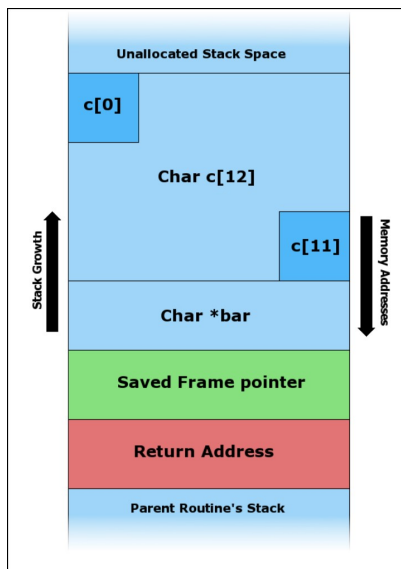


Figure 10 Initialized Stack before execution

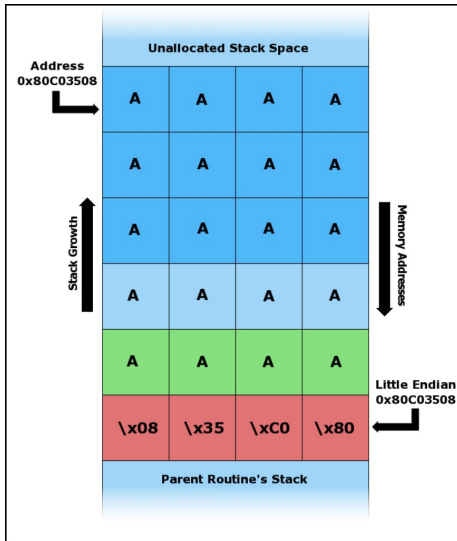


Figure 11 Overwriting return address

1.2.5 NOP Sleds

A NOP sled, alternatively known as a NOP slide, refers to an extensive sequence of No-Operation (NOP) instructions positioned before the actual shellcode.[4] Figure 12. They are used for increasing the chances for a successful exploitation of the program. Although, not mandatory, they frequently used. Their purpose is to increase the reliability of an exploit. More specifically, if the execution of the program directed to any point within the NOP sled, the shellcode will eventually be executed.

Traditional NOP sleds consist of extended sequences of the NOP (No-Operation) instruction, often represented as 0x90 in hexadecimal. Attackers from their side, crafting those NOP instruction alternatively in order to avoid detection patterns. The opcodes in the 0x40 to 0x4f range are often used because they correspond to single-byte instructions that increment or decrement general-purpose registers. For instance, the “inc eax” is the “0x40” hex representation.

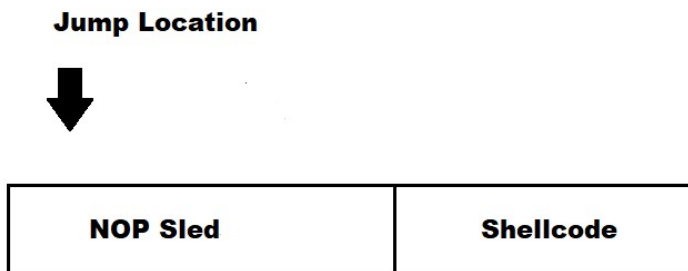


Figure 12 Nop Sleds and shellcode

1.3 Shellcode Insights

Shellcode is a piece of executable code, generally referred as payload, to exploit vulnerabilities within a system or execute malicious commands. The term originates from its typical function of launching a command shell, providing attackers with control over a compromised system. Moreover, shellcodes are specifically crafted to take advantage of specific software vulnerabilities, allowing attackers to bypass security measures and gain control over compromised systems. In the context of buffer overflows, an attacker, upon gaining control of the return address, typically injects a shellcode payload (raw-byte instructions) to redirect the program as desired.

Various types of shellcodes include:

- **Reverse Shell Shellcode:** Establishes a connection between the compromised system and an attacker-controlled system, facilitating remote access to the compromised machine.
- **Bind Shell Shellcode:** Establishes a network on the compromised system to initiate a connection with the attacker, allowing them to control the targeted system.
- **File Download Shellcode:** Exploits vulnerabilities to download and execute a malicious file from a remote server onto the compromised system.
- **Shellcode for Local Privilege Escalation:** Exploits operating system vulnerabilities to elevate the attacker's privileges, granting administrative or root access.

1.4 Metamorphic and Polymorphic Shellcodes

In order to defend against these kinds of attacks, the defenders created security mechanism to detect these shellcodes. As a response, the attackers start crafting more sophisticated shellcodes to bypass these mechanisms and achieve their goals. These shellcodes can be categorized to metamorphic and polymorphic shellcodes.

Polymorphism refers to a type of malicious code or shellcode that has the ability to change its appearance dynamically while maintaining its original functionality. More specifically, the polymorphic shellcode may decrypt its code, execute it and then propagate itself encrypted with a different key. Then, when executed on different machine, it executes the same code. Instead, metamorphic shellcodes simply execute their code and then, during propagation, they mutate itself resulting to a different functionality code. The key difference between them is that metamorphism changes its underlying code. [5]

Attackers, most of the times, craft shellcodes combining both of these categories. For the purpose of this thesis, when we mention polymorphic shellcodes, we will refer in their combination. Moreover, in this thesis we are addressing how to detect payloads that they have an encrypted shellcode along with their decryption routine and in top of that the NOP Sleds. Figure 13.



Figure 13 Under Inspection Payload

1.5 Thesis Contribution

This thesis introduces a Windows Detection Engine capable of detecting windows specific shellcodes in a network traffic. Its purpose is to detect advanced shellcodes such as metamorphic or polymorphic shellcodes for a 32-bit Windows operating system. This Engine is integrated in the “Shellcode Detection Using CPU Emulation” (SEDUCE), a Network Intrusion Detection System detecting shellcodes based on the technique of CPU emulation. More specifically, the Engine will be fed traffic from the network (raw bytes) and it will be capable of detecting if a system call in a Windows environment occurred. Furthermore, the focus on the 32-bit architecture is intentional due to the fact that many legacy systems and certain environments still rely on 32-bit versions of the Windows operating system.

1.6 Thesis Structure

Since most of these shellcodes exploit vulnerabilities such as buffer overflows, this thesis, in Chapter 1, is describing some basic concepts for an attacker to achieve a buffer overflow attack to compromise a system. More precisely, there is a briefly historical review regarding how buffer overflows occurred. Later, the technical aspects of a buffer overflow attack are presented such as how the stack or program functions work and most importantly how attackers achieve to exploit it. Then, a detailed explanation is given about what the shellcodes are among with how attackers hide them and how they seem from a network traffic perspective.

In Chapter 2, the inner workings of a Network Intrusion Detection System are presented along with their currently techniques used. The advantages of this thesis solution compared to other Network Intrusion Detection Systems is also examined.

In Chapter 3, some basic concepts regarding the Windows OS inner functionality are described, since the introduced Engine of this thesis act as a Windows OS emulator. Moreover, following each functionality description, a shellcode’s action will be presented. The basic concepts consist of: (a) how a Windows OS executing system calls, (b) which are the basic structures of a Windows OS along with their purposes, and (c) what actions are taken when a Windows OS operates with memory regarding some basic registers which a Windows OS frequently uses.

In Chapter 4, detailed analysis of the thesis’s Engine implementation will be presented. More specifically, it will be described in details the structure of the memory map and how the memory layout is created. Additionally, all the helper and basic functions will be analyzed as they are the core functionality of this Engine.

In Chapter 5, the User’s Manual is provided. More specifically, all the mandatory dependencies are presented which they should be installed and detailed instructions on how to build the thesis’s Engine.

In Chapter 6, a presentation of the dataset preparation is provided. Moreover, it is presented a Proof-of-Concept that the Engine is capable of detecting polymorphic shellcodes and a general discussion regarding the results based on the detection time duration of them.

In Chapter 7, conclusions based on the results from the preceding chapter will be presented. Furthermore, suggestions for future enhancements will be given for making the Engine more robust and effective.

2. Field Research

In the network level, generally it is very difficult to detect the abovementioned shellcodes due to their capability to hide themselves with sophisticated techniques. Multiple systems with different techniques created for inspecting such malicious network traffics. This thesis, although introduces an Engine capable for detecting advanced shellcodes, it is integrated in a NIDS system using a technique which is more robust than other NIDS for detecting such shellcodes in a network traffic.

2.1 Network Intrusion Detection System (NIDS)

Network Intrusion Detection Systems (NIDS) is designed to identify and respond to potentially malicious traffic in the network. More specifically, they are capable to inspect the network traffic and, if any suspicious packet identified, it would report this incident to a Security Information and Event Management (SIEM) system. At the same time, it will drop this packet before reaching the destination system. NIDS may be either a hardware device or a software application monitoring inbound or outbound network traffic and usually placed behind a firewall. Figure 14. Multiple techniques are invented for detecting the malicious traffic. [6]

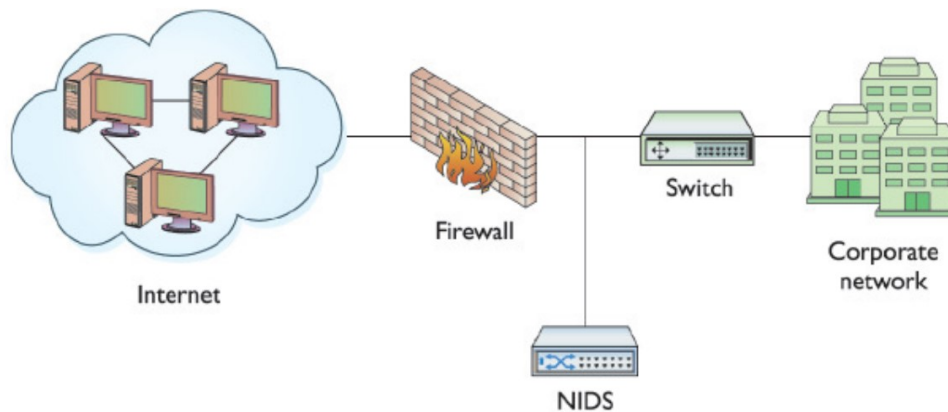


Figure 14 NIDS Placement

2.2 Detection Techniques of NIDS

2.2.1 Signature-based Detection

The Signature-based technique is the most common used by most NIDS. It is inspecting the network traffic based on signatures, patterns or known identified events. [7] In the context of a polymorphic shellcodes, it may try to match the decryption routine or the nop sleds based on existing data in its database. For this reason, these NIDS always should have their database up to date. The consequences of this technique are that it relies on already identified events, meaning that it cannot detect new attack vectors. Moreover, the database of such systems is increasing as new event-entries should be inserted in. As a result, the processing time, for each packet inspection, will be increased causing high delays when packets are transmitted.

2.2.2 Anomaly-based Detection

NIDS using anomaly-based detection, are creating a base-line network traffic. More specifically, they inspect the network traffic how it normally should be and if any abnormally traffic detected, they will report it to the Security Information and Event Management (SIEM) system. The malicious traffic if should be rejected or not, is based on their configuration. The consequences of this technique are that these types of NIDS are inspecting the traffic generally and not the actual payload. Moreover, they are prone to false positives. For instance, previously unknown traffic, but legitimate behavior can also get flagged accidentally.

2.2.3 Emulation-based Detection

The Emulation-based technique was firstly introduced by Polychronakis [9]. NIDS systems using this technique, are inspecting the network traffic by executing portions of traffic packets based on an emulated environment. More specifically, this environment may try to execute the network traffic as CPU instructions and if a malicious shellcode detected, it will report the event to a SIEM system. Simultaneously, based on configuration settings, these packets will either be dropped to prevent the threat or omitted. These kinds of NIDS are very effective because: (a) they may detect previously unknown polymorphic shellcodes- resilient to new type of attacks, (b) is not relying to any kind of database due to their dynamic inspection nature, (c) they inspect the actual traffic and focusing on the payloads and (d) they are less prone to false positives because the network traffic is executed in an emulated environment. Generally, this technique overcomes the consequences of the other two techniques.

2.3 Shellcode Detection Using CPU Emulation

SEDUCE is a tiny network intrusion detection system which is using the emulation-based technique. It consists of a sensors-agents schema. The sensors are placed strategically in the front line of the network. They are responsible for collecting based on filters the network traffic and distributing it to the agents. On the other side, the agents are placed after the sensors. They are responsible for inspecting the network traffic delivered by the sensors, and upon any malicious data found, they will immediately report to a SIEM system.[10]. The solution implemented in this thesis, is extending the capabilities of an agent by detecting 32-bit windows specific polymorphic shellcodes in a network. More specifically, combining the emulation-based technique with the integration of the engine within the agent - which functions as an emulator of a Windows operating system - significantly enhances the robustness and effectiveness of SEDUCE compared to other solutions, particularly when deployed in front of Windows host devices.

3. Design

3.1 Windows Internals

To design a Windows engine that detects shellcodes specific to Windows x86, it is crucial to understand how Windows facilitates system calls. In contrast to Linux, where applications can “directly” access system calls through interrupt software functions, Windows applications operate differently. They follow a distinct, lengthier route to utilize system calls. More specifically, they are designed to rely on functions from the Windows API. These Windows API functions, in turn, invoke functions from the Native API, which utilize the system calls. [11] The Native API functions are undocumented and are implemented in the ntdll.dll file. Moreover, they represent the lowest level of abstraction in the user-mode level. The following Figure 15 illustrates the abstraction layer between user and kernel mode.

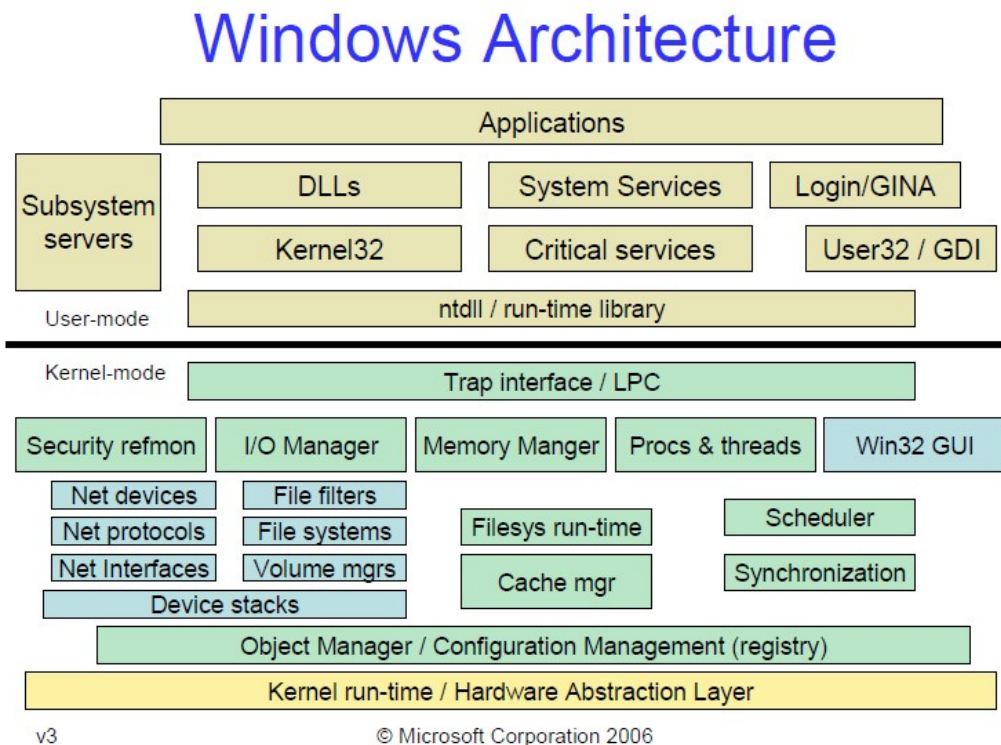


Figure 15 Windows Architecture

The documented functions, utilized by developers through the Windows API, are residing inside the dll files. Some of the libraries are the kernel32.dll, advapi32.dll, gdi32.dll and are many more, depending on the specific task to be executed. The most commonly used functions, crucial for interacting with core services like working with file systems, processes, devices, etc, are provided by kernel32.dll.

Attackers in order to craft stable shellcodes for Windows, may utilize functions generally from the Windows API functions. The three most vital libraries kernel32.dll, kernelbase.dll and

ntdll.dll, are so fundamental that they are imported by almost every process. As a Proof-of-Concept Figure 16 and Figure 17 illustrate the loaded DLLs from some common processes “explorer.exe” and “notepad.exe”, using the tool ListDLLs contained in the Sysinternals Suite of Windows.[12]

```

Windows PowerShell
PS C:\Users\tsili\Downloads> .\Listdlls.exe explorer.exe

Listdlls v3.2 - Listdlls
Copyright (C) 1997-2016 Mark Russinovich
Sysinternals

-----
explorer.exe pid: 20348
Command line: C:\Windows\Explorer.EXE

Base          Size          Path
0x00000000e2eb0000 0x513000 C:\Windows\Explorer.EXE
0x0000000018d00000 0x1f8000 C:\Windows\SYSTEM32\ntdll.dll
0x0000000003900000 0xbd0000 C:\Windows\System32\KERNEL32.DLL
0x00000000ff470000 0x2f6000 C:\Windows\System32\KERNELBASE.dll

```

Figure 16 Listing loaded DLLs for explorer.exe

```

Windows PowerShell
PS C:\Users\tsili\Downloads> .\Listdlls.exe notepad.exe

Listdlls v3.2 - Listdlls
Copyright (C) 1997-2016 Mark Russinovich
Sysinternals

-----
notepad.exe pid: 14356
Command line: "C:\Windows\system32\notepad.exe"

Base          Size          Path
0x00000000611f0000 0x380000 C:\Windows\system32\notepad.exe
0x0000000018d00000 0x1f8000 C:\Windows\SYSTEM32\ntdll.dll
0x0000000003900000 0xbd0000 C:\Windows\System32\KERNEL32.DLL
0x00000000ff470000 0x2f6000 C:\Windows\System32\KERNELBASE.dll

```

Figure 17 Listing loaded DLLs for notepad.exe

It should be noted that the base addresses of DLLs are the same across processes on the same machine because DLLs are loaded into memory once and shared among multiple processes using a technique called DLL sharing. However, they will differ across machines and across reboots.

3.2 PE and Win32 API (DLLs)

3.2.1 Portable Executable (PE)

Every executable file follows a standard format known as Common Object File Format (COFF). The PE file format, a specific COFF format, is available for executable files, object code, DLLs, etc. for both 32-bit and 64-bit versions of Windows operating systems. In contrast, Unix-based systems, including Linux, the available files are the Executable Link File (ELF) format.

Portable Executable file format is a data structure which is providing with all the necessary information the Windows OS loader how to manage the wrapped executable code. This includes dynamic library references for linking, the tables containing the import and export APIs, data dedicated to resource management, Thread Local Storage (TLS) data etc. Moreover, when PE files are loaded into memory via the Windows loader, the in-memory version is known as a module and it consists of the DOS Header, DOS Stub, PE File Header, Image Optional Header, Section Table, Data Dictionaries, and Sections. Figure 18. [13]. An important term "Relative Virtual Address (RVA)" is the distance between the beginning of a section or data structure and the base address of the module.

Regarding the DLLs in-memory modules, after a shellcode is injected, may traverse several headers of these modules in order to find the base address of functions that it will then use for utilizing systems calls. For instance, the path that an attacker's shellcode is usually following is (finding the WinExec function inside the kernel32.dll):

- 1) Determine the Relative Virtual Address (RVA) of the PE signature by adding the base address and an offset of 0x3C bytes.
- 2) Calculate the address of the PE signature by adding the base address to the previously obtained RVA of the PE signature.
- 3) Identify the RVA of the Export Table by adding the address of the PE signature to an offset of 0x78 bytes.
- 4) Determine the address of the Export Table by combining the base address with the RVA of the Export Table.
- 5) Find the count of exported functions by adding 0x14 bytes to the address of the Export Table.
- 6) Obtain the RVA of the Address Table by adding 0x1C to the address of the Export Table.
- 7) Find the address of the Address Table by combining the base address with the RVA of the Address Table.
- 8) Determine the RVA of the Name Pointer Table by adding 0x20 bytes to the address of the Export Table.
- 9) Find the address of the Name Pointer Table by adding the base address to the RVA of the Name Pointer Table.
- 10) Identify the RVA of the Ordinal Table by adding 0x24 bytes to the address of the Export Table.
- 11) Determine the address of the Ordinal Table by combining the base address with the RVA of the Ordinal Table.
- 12) Iterate through the Name Pointer Table, comparing each string (name) with "WinExec" and keeping track of the position.
- 13) Find the WinExec ordinal number from the Ordinal Table by adding (position * 2) bytes to the address of the Ordinal Table (each entry is 2 bytes).
- 14) Obtain the function RVA from the Address Table by adding (ordinal_number * 4) bytes to the address of the Address Table (each entry is 4 bytes).

15) Find the function address by adding the base address to the obtained function RVA.

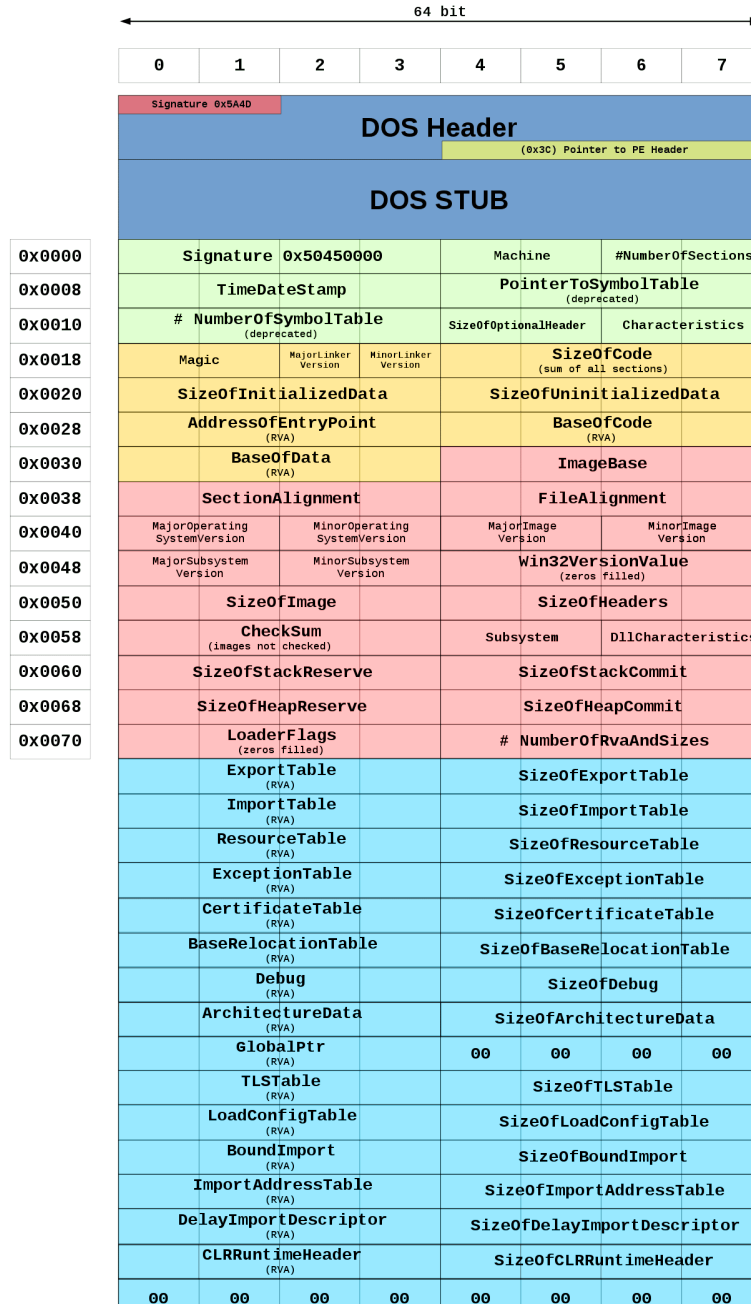


Figure 18 PE Format

3.2.2 Win32 API (DLLs)

The Win32 API, or Windows API, refers specifically to the application programming interface provided by Microsoft for developing applications on the Microsoft Windows operating system. [11]. The "32" in Win32 refers to the 32-bit architecture of the Windows operating system. While modern versions of Windows support both 32-bit and 64-bit architectures, the term "Win32" is still commonly used to refer to the API as a historical convention. It is a set of functions and data structures that a Windows program can use to ask Windows to do a specific task, like opening a file, displaying a message, etc. Figure illustrates the chained reaction when "VirtualAlloc" function is called from kernel32.dll file.

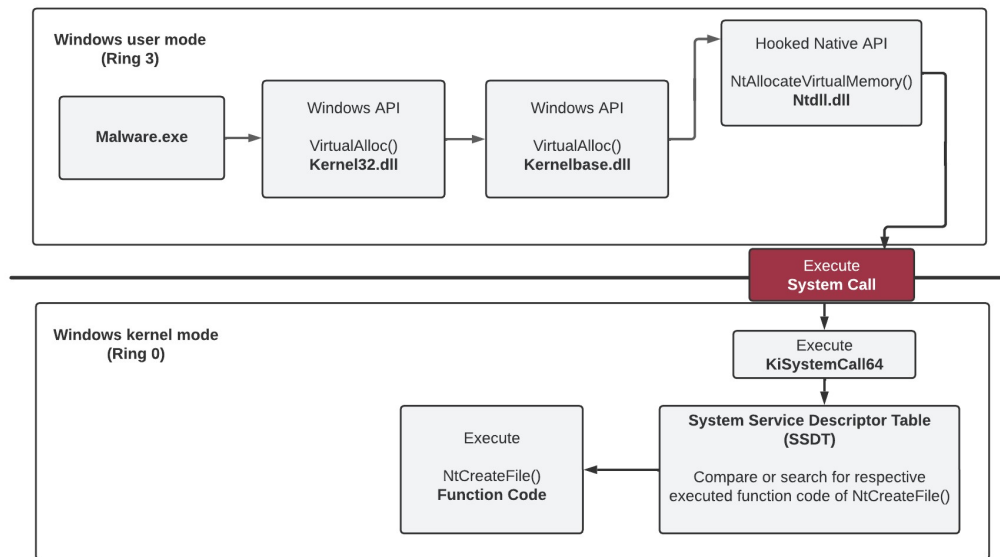


Figure 19 "VirtualAlloc" API Path

As described in the 3.1 Windows Internals chapter, crafted shellcodes are utilizing functions from Win32 API residing in specific dlls. Knowing that dlls are PE files, they are mostly interested and searching for those functions inside the export table of these files as described previously.

3.3 PEB and TEB structures

3.3.1 Process Environment Block

The PEB (Thread Environment Block) is a data structure that holds information about a process and is present in memory for every running process. It has been present in Windows since the introduction of Win2k (Windows 2000). [11]

It provides information about a process at both kernel mode and user mode. Moreover, it is a structure available for every process at a fixed address in memory. For x86 processes, it can be found at fs:[0x30] in the Thread Environment Block (TEB), and for x64 processes, it's typically located at gs:[0x60].

This structure contains useful information regarding the process, including the process name, process ID, pointers to other structures such as to PEB_LDR_DATA, and much more. The following Figure 20 illustrates some of its members.

```

typedef struct _PEB {
    BYTE           Reserved1[2];
    BYTE           BeingDebugged;
    BYTE           Reserved2[1];
    PVOID          Reserved3[2];
    PPEB_LDR_DATA  Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    PVOID          Reserved4[3];
    PVOID          AtlThunkSListPtr;
    PVOID          Reserved5;
    ULONG          Reserved6;
    PVOID          Reserved7;
    ULONG          Reserved8;
    ULONG          AtlThunkSListPtr32;
    PVOID          Reserved9[45];
    BYTE           Reserved10[96];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE           Reserved11[128];
    PVOID          Reserved12[1];
    ULONG          SessionId;
    -----More Members-----
} PEB, *PPEB;

```

Figure 20 Process Environmental Block Structure

3.3.2 Thread Environment Block

In a process, there can be one or more threads, and typically, each process begins with a single primary thread. Additional threads are created as needed. Despite sharing the same virtual addresses, each thread possesses its own set of resources, including exception handlers, local storage, and more. So, like PEB, each thread has a TEB (Thread Environment Block), residing in the process address space. It is also known as Thread Information Block (TIB). [11]

Moreover, the address of the TEB can be retrieved through the FS register for 32-bit Windows operating systems and via the GS register for 64-bit Windows operating systems. This mechanism allows for efficient access to thread-specific details and supports the functionality of a multi-threaded process.

The TEB is a crucial data structure containing information essential to the execution context of each thread. Within the TEB, valuable details include thread-specific data such as the thread ID and initial stack address, exception handling information for managing errors, details about loaded DLLs including base addresses and entry points, and an array for Thread Local Storage

(TLS) allowing threads to have their own set of variables. The following Figure 21 illustrates some of its members.

```
typedef struct _TEB {
    PVOID Reserved1[12];
    PPEB ProcessEnvironmentBlock;
    PVOID Reserved2[399];
    BYTE Reserved3[1952];
    PVOID TlsSlots[64];
    BYTE Reserved4[8];
    PVOID Reserved5[26];
    PVOID ReservedForOle;
    PVOID Reserved6[4];
    PVOID TlsExpansionSlots;
    -----More Members-----
} TEB, *PTEB;
```

Figure 21 Thread Environmental Block Structure

3.4 PEB_LDR_DATA and LDR_MODULE

The PEB Loader Data, or PEB_LDR_DATA, structure is a Windows Operating System structure that contains information about all of the loaded modules (DLLs) in the current process.[14]. The operating system links to the PEB_LDR_DATA structure in the Process Environment Block (PEB) at a specific offset. More specifically, the PEB structure includes a field that points to the PEB_LDR_DATA structure, and it is located at offset 0x0C from the beginning of the PEB. The following Figure 22 illustrates its structure.

```
typedef struct _PEB_LDR_DATA {
    uint32_t Length;
    _Bool Initialized;
    uint32_t SsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Figure 22 PEB Loader Data Structure

The LIST_ENTRY structure typically contains two pointers, Flink and Blink, which stand for "Forward link" and "Backward link", respectively (Figure 23). These pointers allow elements to be linked in both directions, forming a doubly linked list.

```
typedef struct _LIST_ENTRY {
    uint32_t Flink;
    uint32_t Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

Figure 23 LIST_ENTRY Structure

Also, each DLL loaded into the process has its own LDR_MODULE structure (Figure 24). The LDR_MODULE structure holds essential information about a loaded module, including its base address, entry point, size, and names. This structure is part of the linked list of modules and is often accessed through the PEB_LDR_DATA structure.

```
typedef struct _LDR_MODULE {
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    uint32_t BaseAddress;
    uint32_t EntryPoint;
    uint32_t SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    uint32_t Flags;
    int16_t LoadCount;
    int16_t TlsIndex;
    LIST_ENTRY HashTableEntry;
    uint32_t TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;
```

Figure 24 Loader Module Structure

PEB_LDR_DATA is the head of the list. It has both forward and backward links to other elements, forming a doubly linked list. These links help traverse the list in both directions. As for LDR_MODULE, these entries are interconnected with PEB_LDR_DATA, forming a dynamic chain which is traversable in order to find any loaded LDR_MODULE.

Windows constructs the list of the loaded DLLs based on the order they (1) were loaded by the windows loader (InLoadOrderModuleList), (2) are found in the memory layout (InMemoryOrderModuleList), (3) were initialized (InInitializationOrderModuleList).

Polymorphic shellcodes typically traverse the PEB_LDR_DATA structure and the linked LDR_MODULE structures in order to find the base address of loaded dlls needed to utilize functions inside them.

3.5 Importance of FS and GS Registers

In Windows operating systems, the FS (Frame Segment) and GS (General Segment) registers play important roles. For 32-bit windows operating systems, the FS register is used to point to TEB structure for each created process/thread. In 64-bit versions the GS register is used for same purpose. [11]. Polymorphic shellcodes very often will contain instructions using the FS or GS registers to reach TEB and, in turn PEB with the purpose of finding dlls loaded into memory.

These registers contain a value called selector holding the index in the Global Descriptor Table where TEB structure may be found. Figure 25.

15	3	2	1	0
Index	Ti	RPL		

Figure 25 Segment Selector

3.5.1 Global Descriptor Table

The Global Descriptor Table (GDT) is a binary data structure specific to the IA-32 and x86-64 architectures. It contains entries telling the CPU about memory segments and also defines base access privileges for certain parts of memory. [15] The GDT's role is to manage memory segments and set up the necessary descriptors for code, data, and other segments. Moreover, there is a special register (GDTR) holding the value pointing to GDT. The following Figure 26 and Figure 27 illustrating the GDTR register and GDT Table, respectively.

79 (64-bit mode)			
48 (32-bit mode)	16	15	0
Offset	Size		
63 (64-bit mode)			
31 (32-bit mode)	0	15	0

Figure 26 GDTR Register

Address	Content
GDTR Offset + 0	Null
GDTR Offset + 8	Entry 1
GDTR Offset + 16	Entry 2
GDTR Offset + 24	Entry 3
...	...

Figure 27 Global Descriptor Table

The x86 architecture has two methods of memory protection and of providing virtual memory - segmentation and paging. In the 32-bit version of Windows, each process has its own set of segment descriptors in the GDT, which are managed by the operating system. The GDT is responsible for defining the memory segments for both user-mode and kernel-mode execution.

3.5.2 Segment Registers

In x86 architecture, segment registers are special-purpose registers that are used in conjunction with memory segmentation [15]. The most used segment registers are:

1. CS (Code Segment): Points to the base address of the code segment. The instruction pointer (EIP or RIP in 32-bit or 64-bit mode, respectively) contains the offset within this segment.

2. DS (Data Segment): Points to the base address of the data segment. Used for data accesses.
3. ES (Extra Segment): Historically used as an additional data segment in certain operations.
4. FS and GS (F and S letters comes after the E – Extra Segment): Additional segment registers introduced to provide extra segments for certain operations.

3.6 Overall Design

A shellcode's main purpose is to reach the desired function in order to execute a system call. To achieve this goal, it has to traverse several structures when injected into a process. The traversal path usually is "TEB->PEB->PEB_Ldr_DATA->InMemoryOrderLoadList->currentProgram->ntdll base address->kernel32 base address". Then, when finding the appropriate dll's base address (kernel32.dll in our case), it will search for the desired function in it as described in the Portable Executable (PE) chapter. So, the constructed Engine should consist of the following:

- a) a Process Environment Block structure along with its structured members.
- b) a Thread Environment Block structure along with its structured members.
- c) a PEB Loader Data structure.
- d) a structure for the loaded in-memory modules (kernel32.dll)
- e) a structure for the Global Descriptor Table.

All these structures should be loaded into a virtual memory and interconnect with each other, allowing a shellcode to traverse as it wants. Lastly, the FS register should hold the value index for finding the appropriate entry (TEB structure) in the GDT.

4. Implementation

In the SEDUCE project, the “detection_engine_unicorn_windows_x86.c” and “winternl.h” files are created. The first one is responsible for the initialization and, generally, the running process of the Engine. Also, it is responsible to map the memory properly and write into it all the necessary structures needed. Regarding the “winternl.h” file, all the windows specific structures are declared and are used by the routines of the Engine.

4.1 Library Inclusions

The basic libraries used to build the Windows Engine are the Unicorn Engine project and the libpe library under the project “readpe - PE Utils”.

4.1.1 Unicorn Engine

Unicorn Engine is a lightweight, multi-platform, multi-architecture CPU emulator framework, based on QEMU. [16] It is designed to provide a flexible and efficient platform for emulating instructions across various architectures. A notable difference between Unicorn and QEMU is that we only focus on emulating CPU operations, but do not handle other parts of computer machine like QEMU.

Its core provides an API in C such as:

- Opening and closing Unicorn instance created.
- Starting and stopping the emulation based on end-address, time or instructions count given parameters.
- Reading and writing into memory based on given memory regions given.
- Read and writing to the cpu registers.
- Providing memory management, such as hook memory events for invalid memory access or dynamically map memory at runtime in order to handle invalid/missing memory regions.

4.1.2 libpe

The libpe library under the project “readpe - PE Utils” is used to handle the parsing and analysis of PE files. [17] It provides an API in C to extract information from PE files programmatically, such as headers, sections, and other details embedded within these files. More specifically, for our purpose, it will be used to parse the kernel32.dll for writing its raw-byte content in the memory. Also, it will be used for parsing the exported functions along with their relative virtual addresses and writing them into memory, in case a windows system call occurred.

4.2 Basic Windows structures

The “winternl.h” header file contains all the necessary structures that will be initialized by the functions declared in the “detection_engine_unicorn_windows_x86.c” file. More specifically, it contains the structures for the Process Environment Block, Thread Environment Block, PEB Loader Data and Loader Modules (dll loaded in memory). Moreover, it contains the structures which are declared as members inside the aforementioned structures. It should be noted that for the structures Process Environment Block and the Thread Environment Block, the “#pragma pack(push, 8)” preprocessor directive is in effect. This directive instructs the compiler to adjust the alignment of structure members. The number represents the byte alignment and setting it to 8 means that the structure members will be aligned on 8-byte boundaries. The following Figures 28-37 illustrate the contained structures in this file.

```
typedef struct _LIST_ENTRY {
    uint32_t Flink;
    uint32_t Blink;
} LIST_ENTRY, *PLIST_ENTRY;
```

Figure 28 LIST_ENTRY Structure

```
typedef struct _UNICODE_STRING {
    uint16_t Length;
    uint16_t MaximumLength;
    uint32_t Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

Figure 29 UNICODE_STRING Structure

```
typedef struct _LDR_MODULE {
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
    uint32_t BaseAddress;
    uint32_t EntryPoint;
    uint32_t SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    uint32_t Flags;
    int16_t LoadCount;
    int16_t TlsIndex;
    LIST_ENTRY HashTableEntry;
    uint32_t TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;
```

Figure 30 LDR_MODULE Structure

```
typedef struct _PEB_LDR_DATA {
    uint32_t Length;
    _Bool Initialized;
    uint32_t SsHandle;
    LIST_ENTRY InLoadOrderModuleList;
    LIST_ENTRY InMemoryOrderModuleList;
    LIST_ENTRY InInitializationOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Figure 31 PEB_LDR_DATA Structure

```
typedef struct _RTL_USER_PROCESS_PARAMETERS {
    uint8_t Reserved1[16];
    uint32_t Reserved2[10];
    UNICODE_STRING ImagePathName;
    UNICODE_STRING CommandLine;
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;
```

Figure 32 RTL_USER_PROCESS_PARAMETERS Structure

```
typedef struct _NT_TIB {
    uint32_t ExceptionList;
    uint32_t StackBase;
    uint32_t StackLimit;
    uint32_t Reserved1;
    uint32_t Reserved2;
    uint32_t Reserved3;
    uint32_t Self;
} NT_TIB, *PNT_TIB;
```

Figure 33 NT_TIB Structure

```
typedef struct _CLIENT_ID {
    uint32_t UniqueProcess;
    uint32_t UniqueThread;
} CLIENT_ID, *PCLIENT_ID;
```

Figure 34 CLIENT_ID Structure

```
typedef struct _GDI_TEB_BATCH {
    uint32_t Offset;
    uint32_t HDC;
    uint32_t Buffer[310];
} GDI_TEB_BATCH, *PGDI_TEB_BATCH;
```

Figure 35 GDI_TEB_BATCH Structure

```
typedef struct _PROCESSOR_NUMBER {
    uint16_t Group;
    uint8_t Number;
    uint8_t Reserved;
} PROCESSOR_NUMBER, *PPROCESSOR_NUMBER;
```

Figure 36 PROCESSOR_NUMBER Structure

```
typedef struct _GUID {
    uint32_t Data1;
    uint16_t Data2;
    uint16_t Data3;
    uint8_t Data4[8];
} GUID, *PGUID;
```

Figure 37 GUID Structure

```
#pragma pack(push, 8) // Set packing to 8 bytes
typedef struct __TEB {
    NT_TIB NtTib;
    uint32_t EnvironmentPointer;
    CLIENT_ID ClientId;
    uint32_t ActiveRpcuint32_t;
    uint32_t ThreadLocalStoragePointer;
    uint32_t ProcessEnvironmentBlock; // PPEB
    uint32_t LastErrorValue;
    uint32_t CountOfOwnedCriticalSections;
    uint32_t CsrClientThread;
    uint32_t Win32ThreadInfo;
    uint32_t User32Reserved[26];
    uint32_t UserReserved[5];
    uint32_t WOW32Reserved; // ptr to wow64cpu!X86SwitchTo64BitMode
    uint32_t CurrentLocale;
    uint32_t FpSoftwareStatusRegister;
    uint32_t SystemReserved1[54];
    uint32_t ExceptionCode;
    uint32_t ActivationContextStackPointer; // PACTIVATION_CONTEXT_STACK
    uint8_t SpareBytes[36];
    uint32_t TxFsContext;
    GDI_TEB_BATCH GdiTebBatch;
    CLIENT_ID RealClientId;
    uint32_t GdiCachedProcessuint32_t;
    uint32_t GdiClientPID;
    uint32_t GdiClientTID;
    uint32_t GdiThreadLocalInfo;
    uint32_t Win32ClientInfo[62];
    uint32_t glDispatchTable[233];
    uint32_t glReserved1[29];
    uint32_t glReserved2;
    uint32_t glSectionInfo;
    uint32_t glSection;
    uint32_t glTable;
    uint32_t glCurrentRC;
```

```
uint32_t glContext;
uint32_t LastStatusValue;
UNICODE_STRING StaticUnicodeString;
uint16_t StaticUnicodeBuffer[261];
uint32_t DeallocationStack;
uint32_t TlsSlots[64];
LIST_ENTRY TlsLinks;
uint32_t Vdm;
uint32_t ReservedForNtRpc;
uint32_t DbgSsReserved[2];
uint32_t HardErrorMode;
uint32_t Instrumentation[9];
GUID ActivityId;
uint32_t SubProcessTag;
uint32_t EtwLocalData;
uint32_t EtwTraceData;
uint32_t WinSockData;
uint32_t GdiBatchCount;
PROCESSOR_NUMBER CurrentIdealProcessor;
uint32_t IdealProcessorValue;
uint8_t ReservedPad0;
uint8_t ReservedPad1;
uint8_t ReservedPad2;
uint8_t IdealProcessor;
uint32_t GuaranteedStackBytes;
uint32_t ReservedForPerf;
uint32_t ReservedForOle;
uint32_t WaitingOnLoaderLock;
uint32_t SavedPriorityState;
uint32_t SoftPatchPtr1;
uint32_t ThreadPoolData;
uint32_t TlsExpansionSlots; // Ptr32 Ptr32 Void
uint32_t MuiGeneration;
_Boolean IsImpersonating;
uint32_t NlsCache;
uint32_t pShimData;
uint32_t HeapVirtualAffinity;
uint32_t CurrentTransactionuint32_t;
uint32_t ActiveFrame; // PTEB_ACTIVE_FRAME
uint32_t FlsData;
uint32_t PreferredLanguages;
uint32_t UserPrefLanguages;
uint32_t MergedPrefLanguages;
_Boolean MuiImpersonation;
uint16_t CrossTebFlags;
```

```

uint16_t SameTebFlags;
uint32_t TxnScopeEnterCallback;
uint32_t TxnScopeExitCallback;
uint32_t TxnScopeContext;
uint32_t LockCount;
uint32_t SpareUlong0;
uint32_t ResourceRetValue;
} TEB, *PTEB;

```

Figure 38 TEB Structure

```

typedef struct __PEB {
    uint8_t InheritedAddressSpace;
    uint8_t ReadImageFileExecOptions;
    uint8_t BeingDebugged;
    uint8_t BitField;
    uint32_t Mutant;
    uint32_t ImageBaseAddress;
    uint32_t Ldr; //PEB_LDR_DATA Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    uint32_t SubSystemData;
    uint32_t ProcessHeap;
    uint32_t FastPebLock;
    uint32_t AtlThunkSListPtr;
    uint32_t IFEOKey;
    uint32_t CrossProcessFlags;
    uint32_t UserSharedInfoPtr;
    uint32_t SystemReserved;
    uint32_t AtlThunkSListPtr32;
    uint32_t ApiSetMap;
    uint32_t TlsExpansionCounter;
    uint32_t TlsBitmap;
    uint32_t TlsBitmapBits[2];
    uint32_t ReadOnlySharedMemoryBase;
    uint32_t SharedData;
    uint32_t ReadOnlyStaticServerData;
    uint32_t AnsiCodePageData;
    uint32_t OemCodePageData;
    uint32_t UnicodeCaseTableData;
    uint32_t NumberOfProcessors;
    uint32_t NtGlobalFlag;
    int64_t CriticalSectionTimeout;
    uint32_t HeapSegmentReserve;
    uint32_t HeapSegmentCommit;
    uint32_t HeapDeCommitTotalFreeThreshold;

```



```
uint32_t HeapDeCommitFreeBlockThreshold;
uint32_t NumberOfHeaps;
uint32_t MaximumNumberOfHeaps;
uint32_t ProcessHeaps;
uint32_t GdiSharedHandleTable;
uint32_t ProcessStarterHelper;
uint32_t GdiDCAttributeList;
uint32_t LoaderLock;
uint32_t OSMajorVersion;
uint32_t OSMinorVersion;
uint16_t OSBuildNumber;
uint16_t OSCSDVersion;
uint32_t OSPlatformId;
uint32_t ImageSubsystem;
uint32_t ImageSubsystemMajorVersion;
uint32_t ImageSubsystemMinorVersion;
uint32_t ActiveProcessAffinityMask;
uint32_t GdiHandleBuffer[34]; // or [60] depending on the ptr_size
uint32_t PostProcessInitRoutine;
uint32_t TlsExpansionBitmap;
uint32_t TlsExpansionBitmapBits[32];
uint32_t SessionId;
uint64_t AppCompatFlags;
uint64_t AppCompatFlagsUser;
uint32_t pShimData;
uint32_t AppCompatInfo;
UNICODE_STRING CSDVersion;
uint32_t ActivationContextData;
uint32_t ProcessAssemblyStorageMap;
uint32_t SystemDefaultActivationContextData;
uint32_t SystemAssemblyStorageMap;
uint32_t MinimumStackCommit;
uint32_t FlsCallback;
LIST_ENTRY FlsListHead;
uint32_t FlsBitmap;
uint32_t FlsBitmapBits[4];
uint32_t FlsHighIndex;
uint32_t WerRegistrationData;
uint32_t WerShipAssertPtr;
uint32_t pUnused; // pContextData
uint32_t plmageHeaderHash;
uint64_t TracingFlags;
uint64_t CsrServerReadOnlySharedMemoryBase;
uint32_t TppWorkerpListLock;
LIST_ENTRY TppWorkerpList;
```

```
uint32_t WaitOnAddressHashTable[128];
} PEB, * PPEB;
```

Figure 39 PEB Structure

4.3 Windows Detection Engine

All the necessary functions needed to initialize and run the Windows Engine is declared in the “detection_engine_unicorn_windows_x86.c” file. It is the core component of the implementation in which the memory map, initialization of the structures, starting the Engine take place. Moreover, it contains the function which is used to detect the malicious traffic and also the function which is responsible for its termination.

4.3.1 Memory Layout

For the memory layout, the following macros – symbolic constants are declared. Figure 40

```
#define BASE_ADDR    0x400000
#define EXEC_SIZE    0x600000
#define STACK_BASE   0xD00000
#define STACK_SIZE   0x010000
#define HEAP_BASE    0xD50000
#define HEAP_SIZE    0x010000

#define DLL_META     0x60000000
#define DLL_META_SZ  0x00010000
#define DLL_BASE     0x70000000
#define PEB_LDR_ADDR 0x77dff000
#define TEB_ADDR     0x79000000
#define PEB_ADDR     0x7A000000
#define GDT_BASE     0x80000000
#define GDT_SIZE     0x1000
```

Figure 40 Defining the Macros

From the abovementioned, the BASE_ADDR is where the emulation is going to start. More specifically, it is the starting point of execution and also the address that the emulated binary will be written. The stack’s and heap’s base address and size declared from the STACK_BASE, STACK_SIZE, HEAP_BASE, HEAP_SIZE as their names denote. Moreover, the “kernel32.dll” file’s structure (LDR_MODULE) and its raw byte content are written to DLL_META and DLL_BASE, respectively. Lastly, the structures of the PEB_LDR_DATA, TEB, PEB and GDT will be written to PEB_LDR_ADDR, TEB_ADDR, PEB_ADDR and GDT_BASE.

Based on the above the memory map is illustrated in the following Figure 41:

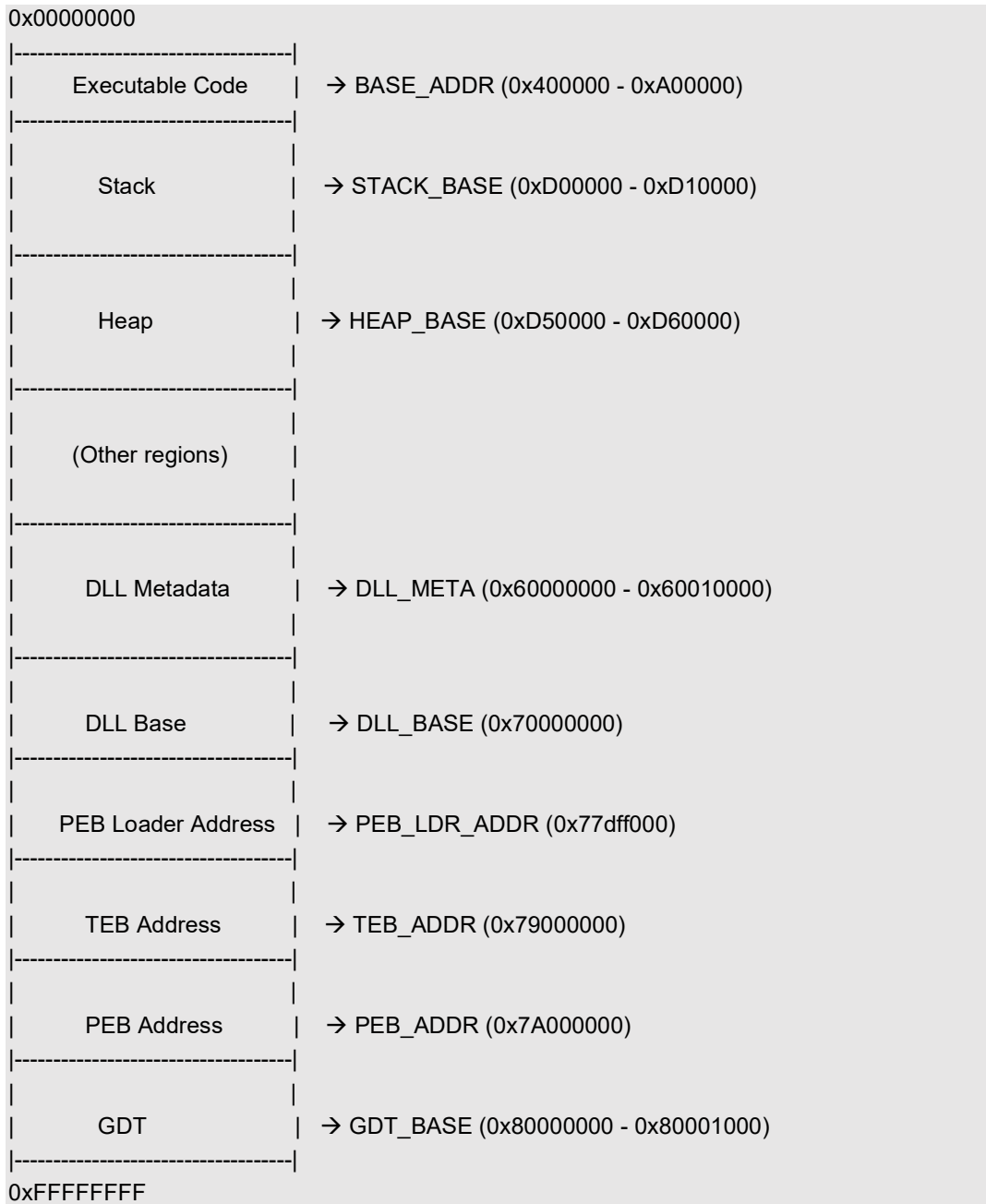


Figure 41 Memory Map in Virtual RAM

4.3.2 Helper Functions

The following functions are declared which they help in the process for either aligning the memory or initializing the structures and write them into the memory. It should be noted that in some of these functions there are additional routines which will be explained in details.

a) The “align4k” function’s purpose is to ensure that a given size is rounded up to the nearest multiple of 4 kilobytes, aligning it to a specific boundary. Figure 42

```
static inline size_t align4k(size_t size){
    size_t alignment = 0x1000;
    size_t mask = ((size_t)-1) & -alignment;
    return (size + (alignment - 1)) & mask;
}
```

Figure 42 “align4k” Function

b) The “create_LDR_Module” function is responsible to align the exported functions’ addresses of a dll and then create and write in memory its structure. More specifically, at first it iterates over the exported functions from the parsed dll file and adjusts their addresses based on the dll base address (DLL_BASE). Then, it initializes the LDR_MODULE structure with information about the dll, such as base address, entry point, and size of the image. Finally, it writes the LDR_MODULE into the memory at specific address along with their dll’s meta data (DLL_BASE, DLL_META addresses). Figure 43

```
/* return value: 1 everything ok
 *              0 an error occurred */
static int create_LDR_Module(uc_engine * uc, pe_ctx_t ctx,
                            uint64_t dll_base_address, uint64_t
dll_meta_addr){
    uc_err err;
    char *dll_name;
    char full_dll_name[PATH_MAX];
    size_t base_dll_name_len, full_dll_name_len, base_dll_name_wide_len,
          full_dll_name_wide_len;
    const char *directory_path = "C:\\Windows\\System32\\";
    LDR_MODULE _dataDLL;
    int retval = 1;

    // Align function addresses based on DLL Base address
    for (int i = 0; i < pe_exported_functions->functions_count; i++) {
        pe_exported_functions->functions[i].address +=
dll_base_address;
    }

    // Creating LDR_MODULE struct for the DLL

    dll_name = strrchr(ctx.path, '/');
    if (!dll_name) {
        return 0;
    }
    dll_name++; // walk past the /
```

```

string      base_dll_name_len = strlen(dll_name);          // length of the narrow character
            snprintf(full_dll_name, sizeof(full_dll_name), "%s%s", directory_path,
                    dll_name);

string      full_dll_name_len = strlen(full_dll_name);    // length of the narrow character
            uint16_t *wideString_base_dll_name =
            (uint16_t *) malloc((base_dll_name_len + 1) * sizeof(uint16_t));
            uint16_t *wideString_full_dll_name =
            (uint16_t *) malloc((full_dll_name_len + 1) * sizeof(uint16_t));

            // Convert the narrow strings to a wide string
            for (size_t i = 0; i < base_dll_name_len; i++) {
                wideString_base_dll_name[i] = (uint16_t) dll_name[i];
            }
            wideString_base_dll_name[base_dll_name_len] = 0;

            for (size_t i = 0; i < full_dll_name_len; i++) {
                wideString_full_dll_name[i] = (uint16_t) full_dll_name[i];
            }
            wideString_full_dll_name[full_dll_name_len] = 0;

            base_dll_name_wide_len = (base_dll_name_len) * sizeof(uint16_t);
            // size of the wide character string in bytes
            full_dll_name_wide_len = (full_dll_name_len) * sizeof(uint16_t); // size of the
wide character string in bytes

            memset(&_dataDLL, 0, sizeof(LDR_MODULE));

            _dataDLL.BaseAddress = (uint32_t) dll_base_address;
            _dataDLL.EntryPoint = (uint32_t) ctx.pe.entrypoint;
            _dataDLL.SizeOfImage = (uint32_t) ctx.map_size;
            _dataDLL.FullDllName.Length = (uint16_t) full_dll_name_wide_len;
            _dataDLL.FullDllName.MaximumLength =
            (uint16_t) full_dll_name_wide_len + 2;
            _dataDLL.FullDllName.Buffer = dll_meta_addr + sizeof(_dataDLL);
            _dataDLL.BaseDllName.Length = (uint16_t) base_dll_name_wide_len;
            _dataDLL.BaseDllName.MaximumLength =
            (uint16_t) base_dll_name_wide_len + 2;
            _dataDLL.BaseDllName.Buffer =
            dll_meta_addr + sizeof(_dataDLL) +
            _dataDLL.FullDllName.MaximumLength;

            _dataDLL.InInitializationOrderModuleList.Flink =
            (uint32_t) PEB_LDR_ADDR + 0xc;

```

```

        _dataDLL.InInitializationOrderModuleList.Blink =
            (uint32_t) PEB_LDR_ADDR + 0xc;
        _dataDLL.InMemoryOrderModuleList.Flink = (uint32_t) PEB_LDR_ADDR +
0x14;
        _dataDLL.InMemoryOrderModuleList.Blink = (uint32_t) PEB_LDR_ADDR +
0x14;
        _dataDLL.InInitializationOrderModuleList.Flink =
            (uint32_t) PEB_LDR_ADDR + 0x1c;
        _dataDLL.InInitializationOrderModuleList.Blink =
            (uint32_t) PEB_LDR_ADDR + 0x1c;

        err = uc_mem_write(uc, _dataDLL.FullDllName.Buffer,
                           wideString_full_dll_name,
full_dll_name_wide_len);
        if (err != UC_ERR_OK) {
            retval = 0;
            goto exit;
        }

        err = uc_mem_write(uc, _dataDLL.BaseDllName.Buffer,
                           wideString_base_dll_name,
base_dll_name_wide_len);
        if (err != UC_ERR_OK) {
            retval = 0;
            goto exit;
        }

        err = uc_mem_write(uc, dll_meta_addr, &_dataDLL, sizeof(_dataDLL));
        if (err != UC_ERR_OK) {
            retval = 0;
        }
    exit:
        free(wideString_full_dll_name);
        free(wideString_base_dll_name);
        return retval;
}

```

Figure 43 “create_LDR_Module” Function

c) The “setup_PEB_LDR” function is used for setting up the Process Environment Block’s (PEB) Loader Data (PEB_LDR_DATA) structure within the emulated environment. The PEB_LDR_DATA structure will be written in the PEB_LDR_ADDR address. Figure 44

```

/* return value: 0 an error occurred
 *      1 everything OK */
static int setup_PEB_LDR(uc_engine * uc){

```

```

uc_err err;
size_t size;
PEB_LDR_DATA _dataPEB_LDR;

memset(&_dataPEB_LDR, 0, sizeof(PEB_LDR_DATA));
_dataPEB_LDR.InInitializationOrderModuleList.Flink =
    (uint32_t) DLL_META;
_dataPEB_LDR.InInitializationOrderModuleList.Blink =
    (uint32_t) DLL_META;
_dataPEB_LDR.InMemoryOrderModuleList.Flink = (uint32_t) DLL_META + 0x8;
_dataPEB_LDR.InMemoryOrderModuleList.Blink = (uint32_t) DLL_META + 0x8;
_dataPEB_LDR.InInitializationOrderModuleList.Flink =
    (uint32_t) DLL_META + 0x10;
_dataPEB_LDR.InInitializationOrderModuleList.Blink =
    (uint32_t) DLL_META + 0x10;
size = align4k(sizeof(_dataPEB_LDR));
err = uc_mem_map(uc, PEB_LDR_ADDR, size, UC_PROT_READ);
if (err != UC_ERR_OK)
    return 0;

err = uc_mem_write(uc, PEB_LDR_ADDR, &_dataPEB_LDR,
                  sizeof(_dataPEB_LDR));

if (err != UC_ERR_OK) {
    uc_mem_unmap(uc, PEB_LDR_ADDR, size);
    return 0;
}

return 1;
}

```

Figure 44 “setup_PEB_LDR” Function

d) The structures of the Process Environment Block and Thread Environment Block will be initialized by the “create_PEB_TEB” function and these structures will be written into the memory at PEB_ADDR and TEB_ADDR addresses, respectively. Figure 45

```

/* return value: 0 an error occurred
 *      1 everything OK */
static int create_PEB_TEB(uc_engine * uc){
    uc_err err;
    size_t size;
    PEB_PEB;
    TEB_TEB;

    memset(&_PEB, 0, sizeof(PEB));
    memset(&_TEB, 0, sizeof(TEB));
}

```

```

_PEB.ImageBaseAddress = (uint32_t) BASE_ADDR;
_PEB.Ldr = (uint32_t) PEB_LDR_ADDR;
_PEB.ProcessHeap = (uint32_t) HEAP_BASE;
_TEB.NtTib.StackBase = (uint32_t) STACK_BASE;
_TEB.NtTib.StackLimit = (uint32_t) STACK_BASE - (uint32_t) STACK_SIZE;
_TEB.NtTib.Self = (uint32_t) TEB_ADDR;
_TEB.ThreadLocalStoragePointer = (uint32_t) TEB_ADDR;
_TEB.ProcessEnvironmentBlock = (uint32_t) PEB_ADDR;

size = align4k(sizeof(_PEB));

err = uc_mem_map(uc, PEB_ADDR, size, UC_PROT_READ);
if (err != UC_ERR_OK)
    return 0;

err = uc_mem_write(uc, PEB_ADDR, &_PEB, size);
if (err != UC_ERR_OK) {
    uc_mem_unmap(uc, PEB_ADDR, size);
    return 0;
}

size = align4k(sizeof(TEB));
err = uc_mem_map(uc, TEB_ADDR, size, UC_PROT_READ);
if (err != UC_ERR_OK) {
    uc_mem_unmap(uc, PEB_ADDR, align4k(sizeof(PEB)));
    return 0;
}

err = uc_mem_write(uc, TEB_ADDR, &_TEB, size);
if (err != UC_ERR_OK) {
    uc_mem_unmap(uc, TEB_ADDR, align4k(sizeof(TEB)));
    uc_mem_unmap(uc, PEB_ADDR, align4k(sizeof(PEB)));
    return 0;
}

return 1;
}

```

Figure 45 “create_PEB_TEB” Function

e) The initialization of the Global Descriptor Table’s structure will be executed by the “create_GDT” function. Also, it will write the GDT structure into the memory along with its registered entries at GDT_BASE address. Figure 46


```

/* return value: 0 an error occurred
 *      1 everything OK */
static int create_GDT(uc_engine * uc){
    uc_err err;

    err = uc_mem_map(uc, GDT_BASE, GDT_SIZE, UC_PROT_READ);
    if (err != UC_ERR_OK) {
        fprintf(stderr, "could not memory map GDT\n");
        return 0;
    }
    /* relevant registers are init by setup_segment_registers */

    const uint8_t a[] = "\xff\xff\x00\x00\x00\xff\xcf\x00";
    err = uc_mem_write(uc, GDT_BASE + 4 * 8, a, sizeof(a));
    if (err != UC_ERR_OK) {
        fprintf(stderr, "could not write at offset 32 of GDT\n");
        uc_mem_unmap(uc, GDT_BASE, GDT_SIZE);
        return 0;
    }

    const uint8_t a1[] = "\xff\xff\x00\x00\x00\xf3\xcf\x00";
    err = uc_mem_write(uc, GDT_BASE + 5 * 8, a1, sizeof(a1));
    if (err != UC_ERR_OK) {
        fprintf(stderr, "could not write at offset 40 of GDT\n");
        uc_mem_unmap(uc, GDT_BASE, GDT_SIZE);
        return 0;
    }

    const uint8_t a2[] = "\xff\xff\x00\x00\x00\x97\xcf\x00";
    err = uc_mem_write(uc, GDT_BASE + 6 * 8, a2, sizeof(a2));
    if (err != UC_ERR_OK) {
        fprintf(stderr, "could not write at offset 48 of GDT\n");
        uc_mem_unmap(uc, GDT_BASE, GDT_SIZE);
        return 0;
    }

    /* was const uint8_t a3[] = "\xff\x0f\x00\xd0\xb7\xf3\x40\x00"; */
    const uint8_t a3[] = "\xff\x0f\x00\x00\x00\xf3\x40\x79";
    err = uc_mem_write(uc, GDT_BASE + 10 * 8, a3, sizeof(a3));
    if (err != UC_ERR_OK) {
        fprintf(stderr, "could not write at offset 80 of GDT\n");
        uc_mem_unmap(uc, GDT_BASE, GDT_SIZE);
        return 0;
    }
}

```

```
        return 1;
    }
```

Figure 46 “create_GDT” Function

f) The process of setting up the segment registers will take place in the “setup_segment_registers” function. The registers are the code segment, data segment, extra segment, stack segment and file segment. Figure 47

```
static inline int setup_segment_registers(void){
    uc_err err;

    memset(&gdtr, 0, sizeof(uc_x86_mmr));
    gdtr.base = GDT_BASE;
    gdtr.flags = 0;
    gdtr.limit = GDT_SIZE;
    gdtr.selector = 0;

    err = uc_reg_write(uc, UC_X86_REG_GDTR, &gdtr);
    if (err != UC_ERR_OK) {
        return 0;
    }

    int b = 35;
    err = uc_reg_write(uc, UC_X86_REG_CS, &b);
    if (err != UC_ERR_OK) {
        return 0;
    }

    int b1 = 43;
    err = uc_reg_write(uc, UC_X86_REG_DS, &b1);
    if (err != UC_ERR_OK) {
        return 0;
    }

    err = uc_reg_write(uc, UC_X86_REG_ES, &b1);
    if (err != UC_ERR_OK) {
        return 0;
    }

    err = uc_reg_write(uc, UC_X86_REG_GS, &b1);
    if (err != UC_ERR_OK) {
        return 0;
    }

    int b2 = 48;
```

```

        err = uc_reg_write(uc, UC_X86_REG_SS, &b2);
        if (err != UC_ERR_OK) {
            return 0;
        }

        int b3 = 83;
        err = uc_reg_write(uc, UC_X86_REG_FS, &b3);
        if (err != UC_ERR_OK) {
            return 0;
        }

        return 1;
    }
}

```

Figure 47 “setup_segment_registers” Function

g) The “hook_dll_functions” function will be called its time an instruction is executed in the unicorn engine. Its main purpose is to detect if the EIP register is in the boundaries of the parsed dll in memory and also checking if this address is belonging to any of the Windows API function. Figure 48

```

static void hook_dll_functions(uc_engine * uc, uint64_t address, uint32_t size,
                               void *user_data){
    EmulationResult *er;
    Threat *threat;
    char threat_msg[400];

    if ((uint32_t) address < (uint32_t) DLL_BASE)
        return;

    er = (EmulationResult *) user_data;
    threat = er->threat;

    for (int i = 0; i < pe_exported_functions->functions_count; i++) {
        uint32_t funcAddress =
            (uint32_t) pe_exported_functions->functions[i].address;
        if ((uint32_t) funcAddress == (uint32_t) address) {
            er->gotcha = 1;
            threat->severity = SEVERITY_HIGH;
            snprintf(threat_msg, sizeof(threat_msg),
                    "Windows x86 kernel32.dll call
detected (%s)",
                    pe_exported_functions->functions[i].name);
            threat->msg = strdup(threat_msg);
            if (!threat->msg) {

```

```
memory for thread string\n");
                                                                    fprintf(stderr, "could not allocate
                                                                    er->gotcha = -1;
                                                                    }
                                                                    uc_emu_stop(uc);
                                                                    return;
                                                                    }
                                                                    }
                                                                    }
```

Figure 48 “hook_dll_functions” Function

4.3.3 Main functions

The following functions are responsible for the initialization of the unicorn engine and also what should be done while this engine is running.

a) The “uni_engine_init” function is responsible of the Unicorn Engine’ s initialization. Its primary purpose is to set up the emulation environment, load a DLL (Dynamic Link Library) into memory, and initialize various data structures related to the emulation of a Windows environment. Firstly, it will create an instance of the Unicorn Engine. Then, it will parse and load in a variable the exported functions of the kernel32.dll and, consequently write dll’ s raw byte content into memory. Lastly, it will set up the memory as described in the previous chapters by calling the abovementioned functions. Figure 49

```
static int uni_engine_init(void){
    uc_err err;
    const void *raw_data;
    char *path = DLL_DIR "/kernel32.dll";

    // Initialize engine
    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        fprintf(stderr, "could not open unicorn engine in x86 mode\n");
        return 0;
    }

    disposable_mem = mmap(NULL, HEAP_BASE + HEAP_SIZE - BASE_ADDR,
PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (disposable_mem == MAP_FAILED) {
        perror("could not allocate disposable emulation memory");
        goto err_mmap;
    }

    err = uc_mem_map_ptr(uc, BASE_ADDR, HEAP_BASE + HEAP_SIZE -
BASE_ADDR, UC_PROT_ALL, disposable_mem);
    if (err != UC_ERR_OK) {
```

```

        perror("could not map host disposable memory to emulator");
        goto err_uc_mem_map_ptr;
    }

    // Parsing kernel32.dll
    pe_err_e err_loading = pe_load_file(&ctx, path);
    // if we don't find the file in the standard place, search
    // a local directory (this is useful when trying out things from
    // a non-installed version)
    if (err_loading == LIBPE_E_OPEN_FAILED) {
        path = "./DLL/windows-x86/system32/kernel32.dll";
        err_loading = pe_load_file(&ctx, path);
    }
    if (err_loading != LIBPE_E_OK) {
        pe_error_print(stderr, err_loading);
        goto err_pe_load_file;
    }

    // parse the loaded PE file(e.g. kernel32.dll) from previous step
    err_loading = pe_parse(&ctx);
    if (err_loading != LIBPE_E_OK) {
        pe_error_print(stderr, err_loading);
        goto err_pe_parse;
    }
    // Pointer to raw data of PE file
    raw_data = ctx.map_addr;
    //Size of PE file
    raw_pe_size = ctx.map_size;

    // Save globally the exported functions from the previous parsed PE file
    pe_exported_functions = pe_exports(&ctx);

    // Load parsed file in memory
    // size of memory block; MUST be 4 KB (4 * 1024) aligned (size=1,2,...
    // otherwise will cause fail) --> In our case raw_pe_size

    err = uc_mem_map(uc, DLL_BASE, align4k(raw_pe_size), UC_PROT_READ |
UC_PROT_EXEC);
    if (err != UC_ERR_OK) {
        fprintf(stderr, "could not create space for loaded DLLs\n");
        goto err_uc_mem_map_dll;
    }

    err = uc_mem_write(uc, DLL_BASE, raw_data, raw_pe_size);
    if (err != UC_ERR_OK) {

```

```

        fprintf(stderr,
                "could not write DLL data to emulator
memory\n");
        goto err_uc_mem_write_dll;
    }

    err = uc_mem_map(uc, DLL_META, DLL_META_SZ, UC_PROT_READ);
    if (err != UC_ERR_OK) {
        fprintf(stderr, "could not map memory for DLL metadata\n");
        goto err_uc_mem_map_dll_meta;
    }

    // Creating the LDR_Module struct for the DLL
    if (create_LDR_Module(uc, ctx, DLL_BASE, DLL_META) == 0) {
        fprintf(stderr, "failed to create LDR module\n");
        goto err_create_ldr_module;
    }

    if (setup_PEB_LDR(uc) == 0) {
        fprintf(stderr, "failed to setup PEB and LDR\n");
        goto err_setup_peb_ldr;
    }

    if (create_PEB_TEB(uc) == 0) {
        fprintf(stderr, "failed to create PEB and TEB\n");
        goto err_create_peb_teb;
    }

    if (create_GDT(uc) == 0) {
        fprintf(stderr, "failed to create GDT\n");
        goto err_create_gdt;
    }

    // we don't pe_unload as we need the structures (functions)
    // exported by libpe for later
    return 1;

err_create_gdt:
    uc_mem_unmap(uc, TEB_ADDR, align4k(sizeof(TEB)));
    uc_mem_unmap(uc, PEB_ADDR, align4k(sizeof(PEB)));
err_create_peb_teb:
    uc_mem_unmap(uc, PEB_LDR_ADDR, align4k(sizeof(PEB_LDR_DATA)));
err_setup_peb_ldr:
err_create_ldr_module:
    uc_mem_unmap(uc, DLL_META, DLL_META_SZ);

```

```

err_uc_mem_map_dll_meta:
err_uc_mem_write_dll:
    uc_mem_unmap(uc, DLL_BASE, align4k(raw_pe_size));
err_uc_mem_map_dll:
err_pe_parse:
    pe_unload(&ctx);
err_pe_load_file:
    uc_mem_unmap(uc, BASE_ADDR, HEAP_BASE + HEAP_SIZE -
BASE_ADDR);
err_uc_mem_map_ptr:
    munmap(disposable_mem, HEAP_BASE + HEAP_SIZE - BASE_ADDR);
err_mmap:
    uc_close(uc);
    return 0;
}

```

Figure 49 “uni_engine_init” Function

b) When the Unicorn Engine is set up properly, the “uni_engine_process” function will be executed in order for the Unicorn Engine to start running. Its main purpose is to set the hook function (hook_dll_functions) and for each block (raw traffic byte) will do the inspection if any system call is detected. Moreover, it starts the engine from a specific point of the constructed memory and before that is responsible to set up the esp and ebp registers for the stack. Figure 50.

```

static int uni_engine_process(char *data, size_t len, Threat * threat){
    uc_err err;
    uc_hook trace_handle;
    EmulationResult er;
    const char *p;
    int block_size, i, block_num = 0;
    int ret = 0;
    uint32_t stack_top, ebp;

    if ((data == NULL) || (len == 0))
        return 0;

    er.gotcha = 0;
    er.threat = threat;

    stack_top = STACK_BASE; // that's where it starts off from
    ebp = stack_top + sizeof(void *);

    err = uc_hook_add(uc, &trace_handle, UC_HOOK_CODE, hook_dll_functions,
&er, BASE_ADDR, DLL_BASE + raw_pe_size
- 1);
/*

```

```

1);
    */
    if (err != UC_ERR_OK) {
        fprintf(stderr,
function hook");
            return -1;
    }

    while ((p = get_next_block(data, len, MIN_BLOCK_LENGTH, &block_size,
                                block_num++)) {
        if (block_size > EXEC_SIZE) {
            fprintf(stderr, "block size larger than available "
emulation\n");
                ret = -1;
                goto exit_loop;
        }

        // Start of disposable_mem is BASE_ADDR in emulator.
        // This is where we copy the payload.
        memcpy(disposable_mem, p, block_size);

        for (i = 0; i < block_size; i++) {
            err = uc_reg_write(uc, UC_X86_REG_ESP,
&stack_top);
                if (err != UC_ERR_OK) {
                    fprintf(stderr, "could not set
ESP\n");
                        ret = -1;
                        goto exit_loop;
                }

                err = uc_reg_write(uc, UC_X86_REG_EBP,
&ebp);
                    if (err != UC_ERR_OK) {
                        fprintf(stderr, "could not set
EBP\n");
                            ret = -1;
                            goto exit_loop;
                    }

                    if (setup_segment_registers() == 0) {
                        fprintf(stderr, "error at setting up
segment"

```



```

        " registers\n");
        ret = -1;
        goto exit_loop;
    }

    DPRINTF_MD5(disposable_mem+i,
                "checking
offset %d\n", i);

    // emulate with 200000 microseconds timeout
    err =
    uc_emu_start(uc, BASE_ADDR+i, 0,
200000, 0);
    if (er.gotcha <= -1) { // callback
        ret = -1;
        goto exit_loop;
    }

    if (er.gotcha == 1) { // found shellcode
        DPRINTF_MD5(p, block_size,
                    "detection at
offset %d\n", i);
        malloc(block_size);
        threat->payload =
        if (!threat->payload) {
            perror("could
not allocate memory for malicious payload");
            ret = -1;
            goto exit_loop;
        }
        memcpy(threat->payload, p,
block_size);
        threat->length = block_size;
        ret = 1;
        goto exit_loop;
    }

    // In all other emulation errors do nothing

    // flush QEMU translations just before the next
round

```

```
BASE_ADDR + block_size - 1);          err = uc_ctl_remove_cache(uc, BASE_ADDR,
                                        if (err != UC_ERR_OK) {
                                            ret = -1;
                                            goto exit_loop;
                                        }
                                        } // for-loop for offsets
                                        // while-loop for blocks
}
exit_loop:
    uc_hook_del(uc, trace_handle);
    return ret;
}
```

Figure 50 “uni_engine_process” Function

5. User's Manual

SEDUCE [18] can be built in a Debian based operating system. First of all, the dependency packages should be downloaded it and built. The command that should be issued is “sudo apt install <PackageName>”. The following packages should be installed:

- git gcc automake autoconf libtool autoconf-archive make
- libglib2.0-dev libconfuse-dev default-mysql-server libprelude-dev
- libpreludedb-dev libpcap-dev libnet1-dev libnids-dev prelude-utils
- libpreludedb7-mysql prelude-manager libyara-dev pkg-config cmake

After the successful installation of the dependencies. The project should be fetched and installed. The following command should be issued:

- a) “sudo git clone https://github.com/seducelDS/seducer.git”
- b) ./autogen.sh
- c) ./configure --enable-win32 --disable-linux64
- d) make install

The (a) command is fetching the project from its repository. The (b) command is fetching all the libraries which will be included such as the Unicorn Engine and libpe. At the same time, it is doing some checks that everything is installed and not any library is missing in order for the user to build the project. The command (c) will configure the project to work only with the Windows Engine created in this thesis. Finally, the command (d) will build the whole project.

As described in 2.3 Shellcode Detection Using CPU Emulation chapter, SEDUCE consists of the sensors and agents nodes. For the detection capabilities of the Windows Engine created, the binary under the “agent folder” will be used. This binary is designed to emulate an agent when set up to a network.

To start a specific Engine to inspect a given network traffic, the “./seduce win32 /{pathToBinaryTraffic}” command should be issued. The option /{pathToBinaryTraffic} is the path of the raw-binary network traffic file having also the shellcode. The results of the Experimental Evaluation chapter will be based on the execution of this command.

6. Experimental Evaluation

For the purpose of the experimental evaluation of the thesis's Windows Engine, different kind of shellcodes created and fed it. Moreover, the dataset for the Engine to be tested, the "msfvenom" application program was used.

6.1 Dataset Preparation

"msfvenom" is a part of the Metasploit Framework, a widely used penetration testing and exploitation toolkit. Metasploit is open-source and provides tools for developing, testing, and executing exploits against remote targets. [19] It is commonly used by security professionals, ethical hackers, and penetration testers to assess the security of systems. "msfvenom" specifically is a combination of the words "Metasploit" and "payload generation." It is a powerful payload generator that is used to create various types of malicious shellcodes for a variety of operating systems and architectures.

For the thesis purpose, the command "msfvenom -p windows/exec CMD=calc.exe -a x86 --platform windows -e x86/shikata_ga_nai -i 100 -f raw" will be issued for the payload creation.

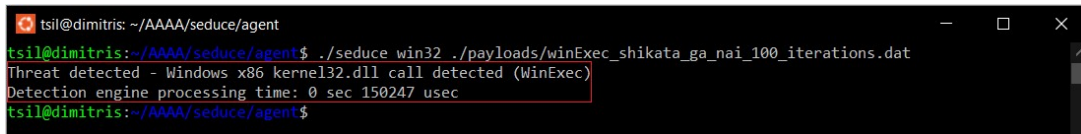
Options given:

- a) **-p windows/exec**: Specifies the payload to be generated. In this case, it's a Windows command execution payload.
- b) **CMD=calc.exe**: Specifies the command to be executed. In this example, it's set to execute the Windows Calculator (calc.exe). You can replace this with any other command you want to execute on the target system.
- c) **-a x86**: Specifies the architecture of the payload. Here, it's set to x86, indicating a 32-bit payload.
- d) **--platform windows**: Specifies the target platform, which is Windows in this case.
- e) **-e x86/shikata_ga_nai**: Specifies the encoder to be used. In this example, it's set to use the Shikata Ga Nai encoder, which is a polymorphic XOR additive feedback encoder for x86 instructions.
- f) **-i 100**: Specifies the number of iterations for the encoder. In this case, it's set to 100 iterations.
- g) **-f raw**: Specifies the format of the output payload. In this case, it's set to raw, which means the payload will be generated in raw binary format.

Putting it all together, this msfvenom command generates a Windows payload that executes the Calculator (calc.exe) when executed on a 32-bit Windows system. The payload is encoded using the Shikata Ga Nai encoder with 100 iterations, and the output is in raw binary format.

6.2 Proof-of-Concept

As a Proof-of-Concept the previous command will be issued and the raw data output will be saved in the "winExec_shikata_ga_nai_100_iterations.dat" file. Then, executing the command ".\seduce win32 ./payloads/winExec_shikata_ga_nai_100_iterations.dat", would result of detecting the shellcode's system call. Figure 51.



```

tsil@dimitris: ~/AAAA/seduca/agent
tsil@dimitris:~/AAAA/seduca/agent$ ./seduce win32 ./payloads/winExec_shikata_ga_nai_100_iterations.dat
Threat detected - Windows x86 kernel32.dll call detected (WinExec)
Detection engine processing time: 0 sec 150247 usec
tsil@dimitris:~/AAAA/seduca/agent$

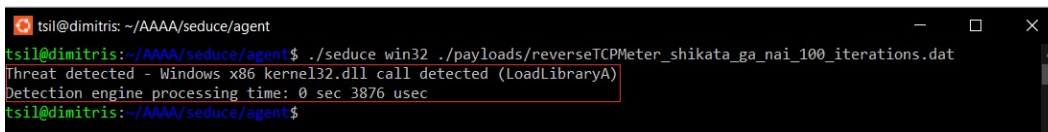
```

Figure 51 Detecting WinExec System Call

As another Proof-of-Concept with a different system call, the command “msfvenom -p windows/meterpreter/reverse_tcp -a x86 --platform windows -e x86/shikata_ga_nai -i 100 -f raw” issued. Regarding the option given in conjunction with the previous one is:

- p windows/meterpreter/reverse_tcp**: Specifies the payload to be generated. In this case, it's a Windows Meterpreter reverse TCP shell payload. Meterpreter is an advanced, dynamically extensible payload that provides a wide range of post-exploitation capabilities.

This msfvenom command generates a Windows Meterpreter reverse TCP shell payload for a 32-bit Windows system. The payload is encoded using the Shikata Ga Nai encoder with 100 iterations, and the output is in raw binary format. The raw data output will be saved in the “reverseTCPMeter_shikata_ga_nai_100_iterations.dat” file. By executing the command “./seduce win32 ./payloads/reverseTCPMeter_shikata_ga_nai_100_iterations.dat”, would result, as previously, of detecting the shellcode's system call. Figure 52.



```

tsil@dimitris: ~/AAAA/seduca/agent
tsil@dimitris:~/AAAA/seduca/agent$ ./seduce win32 ./payloads/reverseTCPMeter_shikata_ga_nai_100_iterations.dat
Threat detected - Windows x86 kernel32.dll call detected (LoadLibraryA)
Detection engine processing time: 0 sec 3876 usec
tsil@dimitris:~/AAAA/seduca/agent$

```

Figure 52 Detecting LoadLibraryA System Call

6.3 Time Duration

The conducted test for time duration, a dataset with different iteration of the -i option will be generated. The command for the generated payloads is as the first PoC with different iterations. Upon the test, the results with seven (7) attempts for each iteration along with the average time are illustrated in the following table:

Iterations	1 st Attempt	2 nd Attempt	3 rd Attempt	4 th Attempt	5 th Attempt	6 th Attempt	7 th Attempt	Average
1	1506	1292	1612	1207	1217	1442	1119	1345.14
5	1803	1530	2027	1527	1568	1783	1710	1679.29
20	7185	5558	7562	6760	6060	6959	6127	6521.71
50	28314	28674	26530	27419	25964	29368	25538	27476.29
100	147851	145152	145124	142965	143608	144847	140916	144762.71

Based on the previous table the following figure may be constructed, illustrating the average time per iteration. Figure 53.

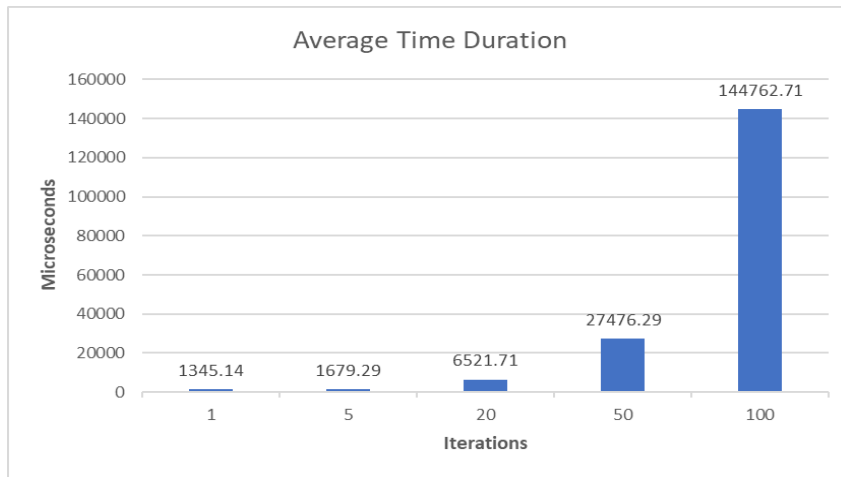


Figure 53 Average time duration line chart

7. Conclusions and Future Work

7.1 Conclusions

As revealed by the results of the conducted test, the thesis Engine is capable for detecting polymorphic shellcodes. More specifically, the Engine continues to effectively emulate the behavior of polymorphic shellcodes regardless the kind of the shellcode (whichever system call wants to execute) or the iterations applied to them. As a result, it demonstrates its robustness and leads to the conclusion that it is resilient to unknown polymorphic shellcodes.

A clear and noteworthy positive correlation exists between the number of iterations applied by the Shikata Ga Nai encoder and the time duration required for detection. Based on the Figure 53, while Engine's emulation is generally more resilient to polymorphism, there is a trade-off in terms of time required for detection. As more iterations applied to the polymorphic shellcode, the time duration for detection is growing. This has the potential to create a bottleneck in the network traffic and overall impacting the systems performance.

7.2 Future Enhancements

The Windows Engine that has been presented in this thesis is capable to detect polymorphic shellcodes which are utilizing functions from the kernel32.dll file. Because other polymorphic shellcodes may utilize functions from other dll files, it is possible to extent its capabilities by integrating more DLLs among with their exported functions. While many shellcodes may attempt dynamic loading of absent DLLs using the LoadLibrary function in kernel32.dll, a more straightforward approach would be to load them during the Engine initialization phase. This strategy minimizes the detection time, optimizing the overall effectiveness of the detection process.

Another extension to SEDUCE Engine may be to support detection capabilities for polymorphic shellcodes for x64 Windows. More specifically, the same Engine may be built by emulating a x64 Windows environment which executing both x86 and x64 programs. In a Windows

environment, the WOW64 (Windows-on-Windows 64-bit) subsystem allows the execution of 32-bit Windows-based applications on 64-bit Windows systems. The result would be to have one Engine for detecting both x86 and x64 polymorphic shellcodes.

Finally, it should be noted that during our test conduction, the Engine for some invalid instructions have been stopped with the exception "Segfault". [20] The root cause of this misbehavior is belonging to the Unicorn library for which developers will provide a fix in the Unicorn version 2.1.0. [21]

8. Bibliography - References

- [1] Aleph One. "Smashing The Stack For Fun And Profit." Phrack Magazine, vol. 14, no. 49, November 1996.
- [2] Wong, Reginald. "Mastering Reverse Engineering". Birmingham, UK: Packt, 2018. ISBN: 9781788838849.
- [3] Anley, Chris, John Heasman, Felix "FX" Linder, and Gerardo Richarte, "The Shellcoder's Handbook: Discovering and Exploiting Security Holes". 2nd ed. Indianapolis, Indiana: Wiley Publishing, 2007. ISBN: 978-0-470-08023-8
- [4] Sikorski Michael and Honig Andrew, "Practical Malware Analysis", San Francisco, CA: No Starch Press, 2007. ISBN-10: 1-59327-290-1
- [5] Nbou Omar, "Detecting and modeling polymorphic shellcode: A new approach". VDM Verlag Dr. Müller, August 26, 2011. ISBN-13: 978-3639377736.
- [6] Northcutt Stephen and Novak Judy, "Network Intrusion Detection: An Analyst's Handbook". 3rd. ed. USA: New Riders Publishing, 2002.
- [7] Meng, Weizhi, and Lam For Kwok. "Enhancing the performance of signature-based network intrusion detection systems: an engineering approach." HKIE Transactions 21, no. 4 (2014): 209–222. <http://dx.doi.org/10.1080/1023697X.2014.970750>.
- [8] Shimeall, Timothy J., and Jonathan M. Spring. "Recognition Strategies: Intrusion Detection and Prevention." In Introduction to Information Security, edited by Timothy J. Shimeall and Jonathan M. Spring, 253–274. Syngress, 2014. ISBN 9781597499699. <https://doi.org/10.1016/B978-1-59749-969-9.00012-2>.
- [9] Polychronakis, Michalis, Kostas G. Anagnostakis, and Evangelos P. Markatos. "Network-Level Polymorphic Shellcode Detection Using Emulation." In Detection of Intrusions and Malware & Vulnerability Assessment, edited by Rainer Büschkes and Pavel Laskov, 4064. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006. https://doi.org/10.1007/11790754_4.
- [10] Δ. Α. Γλυνός, "Αρχιτεκτονική, διαχείριση και ασφάλεια προγραμματιζόμενων δικτυακών υποδομών" Mar. 2013, Last Accessed: Dec 17, 2023. [Online]. Available: <https://dione.lib.unipi.gr/xmlui/handle/unipi/5263>.

- [11] Russinovich, M. E., Solomon, D. A., & Ionescu, A., "Windows Internals". Pearson Education. 2017
- [12] Russinovich, M. E., & Margosis, A. "Troubleshooting with the Windows Sysinternals Tools (IT Best Practices - Microsoft Press)". 2nd ed. Microsoft Press.2016. ISBN-13: 978-0735684447
- [13] Pietrek, Matt. "Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format." MSDN Magazine, February 2002. URL: <https://learn.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail> (Last accessed Dec 17, 2023).
- [14] Microsoft. "PEB_LDR_DATA structure (winternl.h)." Windows Dev Center. URL: https://learn.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data. (Last accessed Dec 17, 2023).
- [15] Silberschatz Abraham; Galvin Peter Baer; Gagne Greg. "Operating System Concepts". 3rd ed. USA: John Wiley & Sons, Inc. 2013. ISBN: 978-1-118-06333-0.
- [16] Unicorn Engine. URL: <https://www.unicorn-engine.org/>. Last Accessed [Dec 17, 2023].
- [17] readpe - PE Utils. URL: <https://github.com/mentebinaria/readpe>. Last Accessed [Dec 17, 2023]
- [18] SEDUCE. URL: <https://github.com/seduceIDS/seduce>. Last Accessed [Dec 17, 2023]
- [19] Rapid 7 Metasploit. URL: <https://www.metasploit.com>. Last Accessed [Dec 17, 2023]
- [20] Unicorn Engine Repository. URL: <https://github.com/unicorn-engine/unicorn/issues/1869>. Last Accessed [Dec 17, 2023]
- [21] Unicorn Engine Repository. URL: <https://github.com/unicorn-engine/unicorn/milestone/3>. Last Accessed [Dec 17, 2023]