

**FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING**

**Department of Computer Engineering**

**1. Course, Subject & Experiment Details**

<b>Practical No:</b>	
<b>Title:</b>	<b>Buffer Overflow</b>
<b>Name of the Student:</b>	<b>Warren Fernandes</b>
<b>Roll No:</b>	<b>8940</b>
<b>Date of Performance:</b>	<b>21-03-2022</b>
<b>Date of Submission:</b>	<b>09-04-2022</b>

**Evaluation:**

<b>Sr. No.</b>	<b>Rubric</b>	<b>Grade</b>
<b>1</b>	<b>On time submission/completion (2)</b>	
<b>2</b>	<b>Preparedness (2)</b>	
<b>3</b>	<b>Skill (4)</b>	
<b>4</b>	<b>Output (2)</b>	

**Signature of the Teacher**

## Buffer Overflow:

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations. For example, a buffer for log-in credentials may be designed to expect username and password inputs of 7 bytes, so if a transaction involves an input of 9 bytes (that is, 2 bytes more than expected), the program may write the excess data past the buffer boundary. Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes.



## Buffer Overflow Attacks:

Attackers exploit 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.

If attackers know the memory layout of a program, they can intentionally feed input that the buffer cannot store, and overwrite areas that hold executable code, replacing it with their own code. For example, an attacker can overwrite a pointer (an object that points to another area in memory) and point it to an exploit payload, to gain control over the program.

## 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.

## Program Implementation:

### Program 1:

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

int main()
{
    char buf1[16]="secret";
    char buf2[16];
    printf("Enter something for buf2:");
    scanf("%s",buf2);
    printf("Value of buf1: %s\n",buf1);
    printf("Value of buf2: %s\n",buf2);
    printf("Address ofbuf1: %p\n",buf1);
    printf("Address of buf2: %p\n",buf2);
    return 0;
}
```

Here we have created two strings of size 16 bytes each and printed the value of strings and the addresses of strings.

```
warren@warren:~/Desktop/CSS$ cc buffer_overflow.c
warren@warren:~/Desktop/CSS$ ./a.out
Enter something for buf2:abcd
Value of buf1: secret
Value of buf2: abcd
Address ofbuf1: 0xbfdaf2ec
Address of buf2: 0xbfdaf2fc
warren@warren:~/Desktop/CSS$
```

Here as the size of input is less than 16 bytes the program there is no problem of buffer overflow and when the size of input is more than 16 bytes the remaining string is saved in adjacent address. In this case the address of “buf1” and buf2” are adjacent so the input string will overwrite the value of buf1.

```
warren@warren:~/Desktop/CSS$ ./a.out
Enter something for buf2:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
Value of buf1: secret
Value of buf2: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAa
Address ofbuf1: 0xbfca2f7c
Address of buf2: 0xbfca2f8c
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
```

## Program 2:

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

int main()
{
    int auth=0;

    char system_pass[16] = "secret";

    char user_pass[16];

    printf("Enter your password: ");
    scanf("%s",user_pass);

    if(strcmp(user_pass, system_pass) == 0)
    {
        auth = 1;
    }

    printf("Value at user pass: %s\n",user_pass);
    printf("Value at syster pass: %s\n",system_pass);
    printf("Address at user pass: %p\n",user_pass);
    printf("Address at system pass: %p\n",system_pass);
    printf("Address at auth: %p\n",&auth);
    printf("Auth variable: %d\n",auth);
    if(auth)
    {
        printf("Password is correct");
    }
    return 0;
}
```

In addition to previous program we have created auth variable and we are comparing both the strings if both the strings are same then auth variable is set to 1 and if auth is not 0 it will print password is correct.

```
warren@warren:~/Desktop/CSS$ cc buffer_overflow_exploit.c
warren@warren:~/Desktop/CSS$ ./a.out
Enter your password: secret
Value at user pass: secret
Value at syster pass: secret
Address at user pass: 0xbfd5354c
Address at system pass: 0xbfd5353c
Address at auth: 0xbfd53538
Auth variable: 1
Password is correctwarren@warren:~/Desktop/CSS$
```

Here we have entered correct password and in output it shows password is correct but in addition we can also see that the address of all the three variables are adjacent so it will be easy to implement buffer overflow on this program.

```
warren@warren:~/Desktop/CSS$ ./a.out
Enter your password: sbcd
Value at user pass: sbcd
Value at syster pass: secret
Address at user pass: 0xbfaa9ecc
Address at system pass: 0xbfaa9ebc
Address at auth: 0xbfaa9eb8
Auth variable: 0
```

Here we have entered incorrect password and the value of auth variable is 0 so it shows that password doesn't matches

```
Enter your password:aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Value at user_pass: aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Value at system_pass: aaaaaaaaaaaaaaaa
Address at user_pass: 0xffffd57c
Address at system_pass: 0xffffd58c
Address at auth: 0xffffd59c
Auth variable: 0
```

Here we have executed buffer overflow but we have not reached the address of auth variable so we it doesn't prints "Password is correct".

```
Enter your password:aaaaaaaaaaaaaaaaaaaaaaaaaaaaab
Value at user_pass: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
Value at system_pass: aaaaaaaaaaaaaaaaab
Address at user_pass: 0xffffd57c
Address at system_pass: 0xffffd58c
Address at auth: 0xffffd59c
Auth variable: 98      I
Password is correct!
```

Here we have overwritten the value of auth variable i.e. ASCII value of b and it shows password is correct.

### Program 3:

---

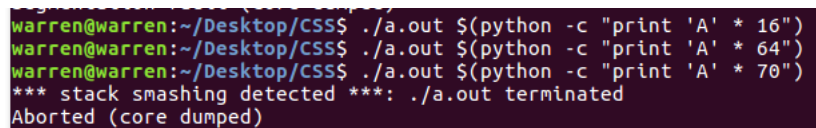
```
#include<stdio.h>
#include<string.h>

int main(int argc, char ** argv)
{
    char buffer[64];
    strcpy(buffer, argv[1]);

    return 0;
}
```

Here we have created string of 64 bytes and we will give the input in command line and then copy the input to the buffer string.

So



```
warren@warren:~/Desktop/CSS$ ./a.out $(python -c "print 'A' * 16")
warren@warren:~/Desktop/CSS$ ./a.out $(python -c "print 'A' * 64")
warren@warren:~/Desktop/CSS$ ./a.out $(python -c "print 'A' * 70")
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
```

Here when we have given input using python command line and in 1<sup>st</sup> three cases code has been executed successfully and in 4<sup>th</sup> case we got segmentation fault that shows from the user input at what offset the return address is located and then hacker can get access to shell and use the machine.

## Postlab:

### 1. How to Prevent Buffer Overflows?

Developers can protect against buffer overflow [vulnerabilities](#) via security measures in their code, or by using languages that offer built-in protection.

In addition, modern operating systems have runtime protection. Three common protections are:

**Address space randomization (ASLR)**—randomly moves around the address space locations of data regions. Typically, buffer overflow attacks need to know the locality of executable code, and randomizing address spaces makes this virtually impossible.

**Data execution prevention**—flags certain areas of memory as non-executable or executable, which stops an attack from running code in a non-executable region.

**Structured exception handler overwrite protection (SEHOP)**—helps stop malicious code from attacking Structured Exception Handling (SEH), a built-in system for managing hardware and software exceptions. It thus prevents an attacker from being able to make use of the SEH overwrite exploitation technique. At a functional level, an SEH overwrite is achieved using a stack-based buffer overflow to overwrite an exception registration record, stored on a thread's stack.

### 2. What Programming Languages are More Vulnerable?

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. Mac OSX, Windows, and Linux all use code written in C and C++.

Languages such as PERL, Java, JavaScript, and C# use built-in safety mechanisms that minimize the likelihood of buffer overflow.