BUFFER OVERFLOW MITIGATION TOOL FOR ANALYSIS OF VULNERABILITY IN C++/C CODES.

A PROJECT REPORT

## Submitted by

**ABHIYANK YADAV 22MIP10074**

**BHARAT RAGHUVANSHI 22MEI10006**

**HIREN SHUKLA 22MIP10070**

**ABHIMANYU NARAYAN KANJOLIA 22MIP10055**

*in partial fulfilment for the award of the degree*

*of*

**C++ PROJECT**

**INTEGRATED MASTER OF TECHNOLOGY**



**SCHOOL OF COMPUTING SCIENCE AND ENGINEERING**

**VIT BHOPAL UNIVERSITY**

**KOTHRIKALAN, SEHORE**

**MADHYA PRADESH – 466114**

**MAY 2024**

**BONAFIDE CERTIFICATE**

Certified that this project report titled” BUFFER OVERFLOWMITIGATION TOOL FOR ANALYSIS OF VULNERABILITY IN C++/C CODES.” is the bonafide work of “ABHIYANK YADAV (22MIP10074), BHARAT RAGHUVANSHI (22MEI10006), HIREN SHUKLA (22MIP10070), ABHIMANYU NARAYAN KANJOLIA (22MIP10055)”who carried out the project work under my supervision. Certified further that to the best of my knowledge the work reported at this time does not form part of any other project/research work based on which a degree or award was conferred on an earlier occasion on this or any other candidate.

**ACKNOWLEDGEMENT**

First and foremost I would like to thank the Lord Almighty for His presence and immense blessings throughout the project work.

I would like to thank my internal guide Mr. T Venkateshwar Rao for continually guiding and actively participating in my project, giving valuable suggestions to complete the project work.

I would like to thank all the technical and teaching staff of the School of Aeronautical Science, who extended directly or indirectly all support.

Last, but not least, I am deeply indebted to my parents who have been the greatest support while I worked day and night for the project to make it a success.

# Motivation

For all the heading levels, their spacing, fonts, numbering, nested level, etc. have been set. In chapter one, at each heading level, corresponding heading number in styles is given at the first instance of use. Similarly, for captions (tables, figures, etc.) everything has been set, including their alignment, spacing, etc. So, from the Styles on the home tab in the ribbon in MS Word, select the appropriate style required for your text. Table of contents, tables, figures, etc. have been automated but will be useful only *if you remember to use automated captions while labelling the same*. For this, use insert caption tab under references tab on the MS word ribbon.

Thanks!

# 

# 

# Abstract

This research paper offers a comprehensive exploration of buffer overflow vulnerabilities in software systems, with a primary focus on analyzing C++ code through parsing and utilizing tools like GDB and Valgrind. The study extends to an exhaustive examination of the causes and consequences of buffer overflow incidents, emphasizing the importance of secure coding practices and rigorous code analysis techniques.

The paper begins with an in-depth discussion of buffer overflow vulnerabilities, providing insights into their nature and potential impact on software security. Following this, the research proceeds to describe the methodology employed for parsing and analyzing C++ code using tools like GDB and Valgrind, offering a detailed background for the ensuing analysis.

In the software analysis phase, C++ code samples are meticulously examined using GDB and Valgrind to detect and mitigate buffer overflow vulnerabilities. The results of these analyses are thoroughly documented and assessed, providing valuable insights into the effectiveness of various mitigation strategies and the relative security of different coding practices.

This research addresses a critical need in software security by extending the analysis to include both manual code examination . By combining these approaches, the study offers a comprehensive understanding of buffer overflow vulnerabilities and effective mitigation techniques.

The outcomes of this research not only contribute to a deeper understanding of buffer overflow vulnerabilities in C++ code but also provide practical guidance for developers and security professionals in identifying and remedying such vulnerabilities. As software security remains a significant concern in modern computing, this research serves as a valuable resource for enhancing the security of software systems through proactive vulnerability analysis and mitigation strategies.

# Table of Contents

[Submitted by i](#_Toc166246331)

[Motivation iv](#_Toc166246332)

[Project Approval v](#_Toc166246333)

[Declaration v](#_Toc166246334)

[Abstract vii](#_Toc166246335)

[Table of Contents ix](#_Toc166246336)

[Table of Figures xii](#_Toc166246337)

[Table of Tables xiii](#_Toc166246338)

[Abbreviation xiii](#_Toc166246339)

[Chapter 1 Introduction of Buffer Overflow 1](#_Toc166246340)

[1.1 What is Buffer Overflows 1](#_Toc166246341)

[1.1.1 Normal Memory layout 1](#_Toc166246342)

[1.2 Detecting Buffer Overflow in CPP 2](#_Toc166246343)

[1.2.1 Input Validation: 2](#_Toc166246344)

[1.2.2 Object Size Checking: 2](#_Toc166246345)

[1.2.3 Stack Canaries: 2](#_Toc166246346)

[1.2.4 Pointer Notation: 3](#_Toc166246347)

[1.2.5 Safe Unsafe Functions: 3](#_Toc166246348)

[1.2.5.1 String Copy in C: 3](#_Toc166246349)

[1.2.5.2 String Concatenation in C: 5](#_Toc166246350)

[1.3 Function chart: 8](#_Toc166246351)

[Chapter 2 10](#_Toc166246352)

[Literature Review 10](#_Toc166246353)

[Chapter 3 13](#_Toc166246354)

[Debugging Buffer Overflow in C++ using GDB and Valgrind 13](#_Toc166246355)

[3.1 Introduction 13](#_Toc166246356)

[3.2 Understanding Buffer Overflow 13](#_Toc166246357)

[3.3 Debugging with GDB 13](#_Toc166246358)

[3.3.1 Compiling the Program for Debugging 13](#_Toc166246359)

[3.3.2 Starting GDB 14](#_Toc166246360)

[3.3.3 Setting Breakpoints 14](#_Toc166246361)

[3.3.4 Running the Program 14](#_Toc166246362)

[3.3.5 Examining Variables 14](#_Toc166246363)

[3.4 Debugging with Valgrind 15](#_Toc166246364)

[3.4.1 Installing Valgrind 15](#_Toc166246365)

[3.4.2 Running Valgrind 15](#_Toc166246366)

[Chapter 4 17](#_Toc166246367)

[Analyzing the code of Buffer Overflow mitigation tool 17](#_Toc166246368)

[4.1 bound\_checking.cpp: 17](#_Toc166246369)

[4.1.1 Header Inclusions: 17](#_Toc166246370)

[4.1.2 Function Definition: 17](#_Toc166246371)

[4.1.3 Opening Input File: 18](#_Toc166246372)

[4.1.4 Opening Output File: 18](#_Toc166246373)

[4.1.5 Reading and Writing Lines: 18](#_Toc166246374)

[4.1.6 Closing Files: 19](#_Toc166246375)

[4.1.7 Success Message: 19](#_Toc166246376)

[4.2 parsing.cpp: 19](#_Toc166246377)

[4.2.1 Header Inclusions: 20](#_Toc166246378)

[4.2.2 Function Definition: 20](#_Toc166246379)

[4.2.3 Opening Input File: 20](#_Toc166246380)

[4.2.4 Parsing Array Accesses: 21](#_Toc166246381)

[4.2.5 Closing Input File and Returning Result: 21](#_Toc166246382)

[4.3 safe\_unsafe\_fn.cpp: 22](#_Toc166246383)

[4.3.1 Header Inclusions: 22](#_Toc166246384)

[4.3.2 Global Variables: 23](#_Toc166246385)

[4.3.3 Function to Check Unsafe Function: 23](#_Toc166246386)

[4.3.4 Function to Check Safe Function: 23](#_Toc166246387)

[4.4 static\_code\_analysis.cpp: 24](#_Toc166246388)

[4.4.1 Header Inclusions: 24](#_Toc166246389)

[4.4.2 Function to Check Proper Variable Name: 24](#_Toc166246390)

[4.4.3 Function to Perform Static Code Analysis: 25](#_Toc166246391)

[4.4.4 Summary Report: 25](#_Toc166246392)

[4.5 main.cpp: 26](#_Toc166246393)

[4.5.1 Header Inclusions: 26](#_Toc166246394)

[4.5.2 Custom Header Inclusions: 26](#_Toc166246395)

[4.5.3 Function to Remove Comments: 26](#_Toc166246396)

[4.5.4 Main Function: 27](#_Toc166246397)

[Chapter 5 28](#_Toc166246398)

[5.1 Utilizing Valgrind and GDB Debugger for Comprehensive Code Analysis and Issue Detection 28](#_Toc166246399)

[5.1.1 Info Registers: 28](#_Toc166246400)

[5.1.2 Adding Breakpoints and Running: 28](#_Toc166246401)

[5.1.3 Valgrind Summary: 28](#_Toc166246402)

[5.1.4 Disassembling Code: 28](#_Toc166246403)

[5.2 Lets Analyze the code snippet without errors:- 29](#_Toc166246404)

[5.3 Lets Analyze the code snippet with errors:- 31](#_Toc166246405)

[References 35](#_Toc166246406)

[Acknowledgements 37](#_Toc166246407)

# Table of Figures

[Figure 1:Program “Pointer Safe String Concatenation 3](#_Toc166246270)

[Figure 2:Program of Buffer Overflow with String Copy 4](#_Toc166246271)

[Figure 3:Program Safe String N-Copy 4](#_Toc166246272)

[Figure 4:Examples of a Buffer Overflow with String Concatenation “strcat” 6](#_Toc166246273)

[Figure 5:Program “Safe String Concatenation” sfstrcat 7](#_Toc166246274)

[Figure 6: GDB installation 15](#_Toc166246275)

[Figure 7: Valgrind installation 16](#_Toc166246276)

[Figure 8 The code Snippet 29](#_Toc166246277)

[Figure 9: Analyzing using Valgrind 29](#_Toc166246278)

[Figure 10 the disas value of code during analysis 30](#_Toc166246279)

[Figure 11 Analyzing the code 31](#_Toc166246280)

[Figure 12 Implementation with gcc 31](#_Toc166246281)

[Figure 13 Segmentatio fault 32](#_Toc166246282)

[Figure 14 The disas analysis 32](#_Toc166246283)

[Figure 15 Valgrind Summary 33](#_Toc166246284)

[Figure 16 Break Points 34](#_Toc166246285)

[Figure 17The Register Values 34](#_Toc166246286)

# Table of Tables

[Table 1:standard functions 2](#_Toc166240792)

[Table 2:String Copy Functions 5](#_Toc166240793)

[Table 3:Truncating Copy Functions 5](#_Toc166240794)

[Table 4:String Concatenation Functions 7](#_Toc166240795)

[Table 5:Safe and unsafe functions 8](#_Toc166240796)

# Abbreviation

|  |  |
| --- | --- |
| BO | Buffer Overflow |
| GCC | GNU Compiler Collection |
| SRC | Source |
| DEST | Destination |
| DISAS | Disassemble |
| EIP | Extended instruction pointer |
| EBP | Extended base pointer |
| GDB | GNU based debugger |

# Chapter 1 Introduction of Buffer Overflow

## 1.1 What is Buffer Overflows

Buffer overflows occur when data is written outside of the boundaries of the memory allocated to a particular data structure. C and C++ are susceptible to buffer overflows because these languages

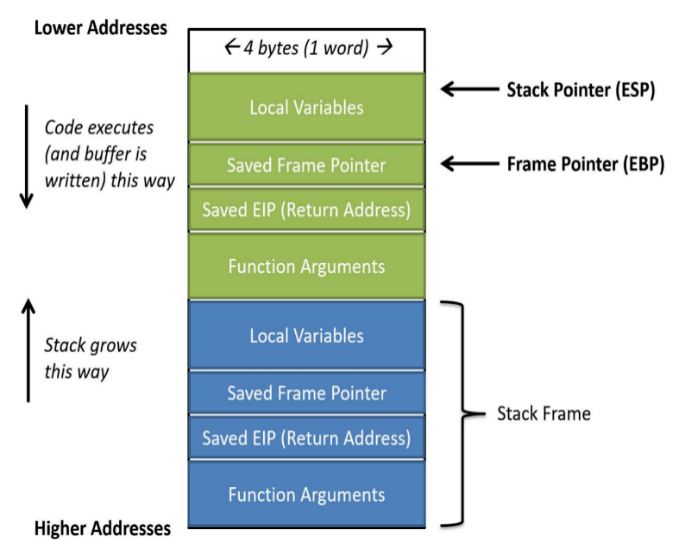
1. Define strings as null-terminated arrays of characters
2. Do not perform implicit bounds checking
3. Provide standard library calls for strings that do not enforce bounds checking

Depending on the location of the memory and the size of the overflow, a buffer overflow may go undetected but can corrupt data, cause erratic behavior, or terminate the program abnormally.

A buffer with finitely many contiguously-grouped free-storage or uninitialized elements a1, a2, a3,…,an is properly used if a C function does not access elements greater than an.

A buffer overflow occurs when a C or C++ program proceeds to access and write on to elements m where m > n. Once the m > n overwriting condition is fulfilled, a BUFFER OVERFLOW occurs, and the program ceases to function normally. Some C programs will abruptly terminate while others will continue overwriting computer information that was never part of the contiguously designed buffer. If the overwriting persists, the computer hard drive will be noticeably altered. BO’s can occur with several functions in the standard C library. The functions strcat, “string concatenation;” strcpy, “string copy;” gets, “get string;” and fgets, “format get string” can produce buffer overflows over the right circumstances and overwrite computer memory.

### Normal Memory layout



### 

Common Mistakes which can be a cause of BO- basic\_string Usage

the basic\_string class include:

1. Using an invalidated or uninitialized iterator
2. Passing an out-of-bounds index
3. Using an iterator range that really is not a range
4. Passing an invalid iterator position

## 1.2 Detecting Buffer Overflow in CPP

1.2.1 Input Validation:

Buffer overflows can be prevented by ensuring that input data does not exceed the size of the smallest buffer in which it is stored. Given below a simple function that performs input validation.

1.2.2 Object Size Checking:

The \_builtin\_object \_size() function is used to add lightweight buffer overflow protection to the following standard functions when \_FORTIFY\_SOURCE is defined:

Table 1:standard functions

|  |  |  |
| --- | --- | --- |
| vprintf() | vsprintf() | memset() |
| sprintf() | snprintf() | memcpy() |
| printf() | strncat() | vsnprintf() |
| fprintf() | strncpy() | memmove() |

Many operating systems that support GCC turn on object size checking by default. Others provide a macro (such as \_FORTIFY \_SOURCE) to enable the feature as an option. On Red Hat Linux, for example, no protection is performed by default.

1.2.3 Stack Canaries:

Stack canaries are another mechanism used to detect and prevent stack-smashing attacks. Instead of performing generalized bounds checking, canaries are used to protect the return address on the stack from sequential writes through memory (for example, resulting from a call to strcpy()). Canaries consist of a value that is difficult to insert or spoof and are written to an address before the section of the stack being protected.

1.2.4 Pointer Notation:

Arrays, must always hold contiguous locations in memory-space and are vulnerable to stack BOs—the variant of BOs under discussion. A pointer-based program for string concatenation might be useful to developers as a succinct, compact alternative to array notation.

For Example: If a program uses the ptrsfstrcat() function poorly, it could lead to unexpected outcomes when incrementing pointers or array subscripts. This can happen because the pointers might end up pointing to memory locations that are not intended, which can cause serious problems when accessing or modifying data stored there.

It's difficult to implement a way to check if these pointers are accessing valid memory locations. In C, there's no built-in mechanism to enforce bounds checking on the arguments passed to functions like ptrsfstrcat(). This means the responsibility falls on the programmer to ensure that pointers are properly managed to avoid accessing invalid memory locations.

char\* ptrstrcat(char\* s, int lens, char\* t, int lent) {

    int i = 0, j = 0, maxt, spaces = lens - strlen(s);

    char \*sstart = s, \*tstart = t;

    if (spaces-- > 0 && (maxt = lent <= strlen(t) + 1 ? lent : strlen(t) + 1) > 0) {

        while (\*s)

            s++;

        char\* sbeginCat = s;

        while (s - sbeginCat < spaces && (\*s++ = \*t++) && t - tstart < maxt)

            ;

    }

    return sstart;

}

Figure 1:Program “Pointer Safe String Concatenation

1.2.5 Safe Unsafe Functions:

In C++, there are both safe and unsafe functions in terms of buffer overflow.

Safe functions, such as std: :string: :append() or std: :vector: :push\_back(), automatically handle memory management and bounds checking, reducing the risk of buffer overflow. These functions dynamically allocate memory as needed and ensure that data is safely appended to the buffer without exceeding its boundaries.

1.2.5.1 String Copy in C:

There are times in the writing of a C program that the developer may wish to copy one string to another string. Passwords, e-mails, and other important quantities manifest themselves as strings in C programs.

Ex- As an example of BO, the culprit string copy manifests itself well in a computer program designed to display a two-by-two matrix with each element on a new line. However, a BO occurs when the developer selects an 8-byte value or fewer for the final character array.

void strcpyMatrix() {

    char exit;

    int qty = 10;

    char original[] = "1 1\n2 2"; // 8 characters long

    char final[9]; // must be 9 or greater!

    strcpy(final, original); // copy original matrix; buffer overflow.

    for (int i = 0; original[i] != '\0'; ++i) {

        printf("%c", final[i]);

        printf("\n");

    }

}

Figure 2:Program of Buffer Overflow with String Copy

char\* sfstrncpy(char\* d, int lend, char\* s, int lens) {

    int choices = lens <= strlen(s) + 1 ? lens : strlen(s) + 1;

    char\* dbegin = d;

    if (lend <= choices) {

        while (lend-- > 1 && (\*d++ = \*s++))

            ;

        \*d = '\0';

    } else {

        while (\*d++ = \*s++)

            ;

    }

Figure 3:Program Safe String N-Copy

Table 2:String Copy Functions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Functions** | **Standard/TR** | **Buffer Overflow Protection** | **Guarantees Null Termination** | **May Truncate String** | **Allocates Dynamic Memory** |
| strcpy() | C99 | No | No | No | No |
| strncpy() | C99 | Yes | No | Yes | No |
| strlcpy() | OpenBSD | Yes | Yes | Yes | No |
| strdup() | TR 24731-2 | Yes | Yes | No | Yes |
| strcpy\_s() | C11 | Yes | Yes | No | No |

**Truncating Copy Functions:** The allocated string must be reclaimed by passing the returned pointer to free().

Table 3:Truncating Copy Functions

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Functions** | **Standard/TR** | **Buffer Overflow Protection** | **Guarantees Null Termination** | **May Truncate String** | **Allocates Dynamic Memory** | **Checks Runtime Constraints** |
| strncpy() | C99 | Yes | No | Yes | No | No |
| strlcpy() | OpenBSD | Yes | Yes | Yes | No | No |
| strndup() | TR 24731-2 | Yes | Yes | Yes | Yes | No |
| strncpy\_s() | C11 | Yes | Yes | No | No | Yes |

1.2.5.2 String Concatenation in C:

In the program, the user passes two char arrays to strcat. One array receives information while the other gives information. The net impact of strcat’s normal operating instructions on the array receiving information becomes evident when the computer writes over the final character in the receiving array called the null-terminator and then proceeds to “add” the contents to the end of the receiving array.

A dangerous problem arises when the receiving array is too small to hold the entirety of t; a pernicious BO occurs leading to undefined behavior. While strncat (“string n concatenation”) is part of the standard library, copying only “n” characters from the C-String submitting information to the passive or “receiving” CString, advocating another solution for this problem might prove to be fruitful for certain applications.

#include <stdio.h>

#include <string.h>

void strcatSandwich() {

    char type[20];

    char sandwich[] = " sandwich";

    printf("Enter the name of the sandwich you want:\n");

    scanf("%19s", type);

    printf("Here is your receipt:\n");

    strcat(type, sandwich);

    printf("%s\n", type);

}

int main() {

    char bose[] = "Bose";

    char chandrasekhar[] = "Chandrasekhar";

    char greetUser[25] = "Greetings ";

    char greetUserTwo[25] = "Greetings ";

    strcat(greetUser, bose);

    printf("%s\n", greetUser);

    strcat(greetUserTwo, chandrasekhar);

    printf("%s\n", greetUserTwo);

    strcatSandwich();

    return 0;

}

Figure 4:Examples of a Buffer Overflow with String Concatenation “strcat”

void sfstrcat(char s[], int lens, char t[], int lent) {

    int i, j;

    i = j = 0;

    if ((lens - strlen(s) - 2) >= lent) {

        while (s[i] != '\0')

            i++;

        while (s[i++] = t[j++])

            ;

    } else if ((lens - strlen(s) - 2) < lent) {

        while (s[i] != '\0')

            i++;

        int y = lens - strlen(s) - 2;

        while (y-- >= 0) {

            s[i++] = t[j++];

            if (y == 0) {

                s[i] = '\0';

                break;

            }

        }

    }

}

Figure 5:Program “Safe String Concatenation” sfstrcat

Table 4:String Concatenation Functions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Functions** | **Standard/TR** | **Buffer Overflow Protection** | **Guarantees Null Termination** | **May Truncate String** | **Allocates Dynamic Memory** |
| strcat() | C99 | No | No | No | No |
| strncat() | C99 | Yes | No | Yes | No |
| strlcat() | OpenBSD | Yes | Yes | Yes | No |
| strcat\_s() | C11 | Yes | Yes | No | No |

## 1.3 Function chart:

Table 5:Safe and unsafe functions

|  |  |
| --- | --- |
| strcpy(): char\*strcpy(char \*s1, const char\*s2); | The strcpy() function copies the string pointed to by s2(including the terminating null character) into the array pointed to by s1. If copying occurs between Objects that overlap, the behavior is undefined. the function strcpy() does not allocate any storage.  The caller must insure that the buffer pointed to by sl is long enough to hold string s2 and its terminating null character. To avoid buffer overflow we use strncpy(). |
| strcat() : char strcat(char \*s1, const char \*s2); | The strcat() function appends a copy of the string pointed to by s2 (including the terminating null character) to the end Of the string pointed to by sl. The initial character of s2 overwrites the null character at the end of sl. If copying occurs between objects that overlap, the behaviour is undefined. The function strcat() does not allocate any storage. The caller must insure that the buffer pointed to by s1 is long enough for string s2 and its terminating null character. TO avoid buffer overflow we use strncat(). |
| memcpy(): void \* memcpy (void \* dest, const void src, size\_t num ); | It copies num bytes from src buffer to memory location pointed by dest. |
| memccpy(): void \*memccpy(void \*sl, const void \*s2, int c, size\_t n); | The memccpy() function operates as efficiently as possible on memory areas. It does not check for overflow of any receiving memory area. Specifically, memccpy() copies bytes from memory area s2 into s], stopping after the first occurrence Of c has been copied, or after n bytes have been copied, whichever comes first. |
| **s**printf(): int sprintf(char \* buffer, const char \*format [ , argument , ...j ); | Writes a sequence of arguments to the given buffer formatted as the format argument specifies (i.e, it calculates how many characters that are being printed). |
| gets(): char \* gets (char \* buffer) | Reads characters from stdin and stores them into buffer until a new line (\n) or EOF character is encountered. The ending new line  character (\n) is not included in the string returned, instead of that a null character (\0) is appended at the end of the resulting string. |
| scanf(): int scanf( const char \*format [ , argument , ...1 ); | The scan/ function reads data from the standard input stream stdin and writes the data into the location given by argument. When reading a string with scanf, always specify a width for the format (for example, “032%s" instead of “ %s"); otherwise, improperly formatted input can easily cause a buffer overrun. Alternately, consider using fgets() to avoid buffer overflow. |
| strecpy(): char \*strecpy (char \*output, const char \*input, const char\*exceptions); | Copies the input string, up to a null byte, to the output string, expanding non-graphic characters to their equivalent C-language escape sequences (for example, \n, \001) |

# Chapter 2

# Literature Review

Dor, Rodeh and Sagiv have developed a system that detects unsafe string operations in C programs [1]. Their system performs a source-to-source trans­for­ma­tion that instruments a program with additional variables that describe string attributes and contains assert statements that check for unsafe string op­er­a­tions. The instrumented program is then analyzed statically using integer analysis to determine possible assertion failures. This approach can handle many com­plex properties such as over­lapping pointers. However, in the worst case the number of variables in the instrumented program is quadratic in the number of variables in the original program. To date, it has only been used on small example programs.

Wagner, et al. have developed a system to statically detect buffer overflows in C [2]. They used their tool effectively to find both known and unknown buffer overflow vulnerabilities in a version of sendmail. Their approach formulates the problem as an integer range analysis problem by treating C strings as an abstract type accessed through library functions and modeling pointers as integer ranges for allocated size and length. A consequence of modeling strings as an abstract data type is that buffer overflows involving non-character buffers cannot be detected.

Nong et al. evaluate vulnerability detection tools (both static and dynamic analysis tools) to identify memory-related vulnerabilities [3]. They also use the dataset created by Toyota, with the same 638 test cases and consider one SAT (i.e., CBMC) and four dynamic analysis tools (i.e., AddressSanitizer, Valgrind, MemorySanitizer, DrMemory). Their results show that SATs accuracy needs to be improved and that it is difficult to obtain both good precision and recall for the same tool. Moreover, tools that use a hybrid approach (static and dynamic techniques) usually have better accuracy. Once again, due to the limitations of the dataset selected, it may not be representative of vulnerabilities in real projects.

Jia et al. propose an offline analysis solution called HOTrace to identify heap vulnerabilities [4]. To do that, HOTrace uses programs’ execution traces and identifies taint attributes during the execution. The whole process is done in the programs themselves, without the source code. The evaluation was performed in 17 Windows x86/x64 applications. Using their prototype, they identified 47 previously unknown heap overflow vulnerabilities, including two vulnerabilities in Microsoft Word.

Liu et al. analyzed five open source C/C++ projects (Linux Kernel, FFmpeg, ImageMagick, OpenSSL, and php) and presented 12 findings, which were applied to find 10 zero-day vulnerabilities [5]. The authors wanted to understand if more vulnerabilities can be found close to identified vulnerabilities. Using the commits to fix the identified vulnerabilities, they built the call-graph for the vulnerable code snippets.

The most seminal works on C (Kernighan & Ritchie, 1978) (Kernighan & Ritchie, 1988) [6], through in their presentation of standard C library functions such as fgets, strcpy, and strcat, do not give further advice or workarounds for BOs.

“Buffer overflow has been the most important vulnerability faced by developers” (Shaw, 2014) [7] reads one article before delving into attempts to automatically transform some of the unsafe C functions noted in the last paragraph into program-safe alternatives immune to BO. The two methods of checking for BO in the literature involve static or dynamic schemas: static for the C code checking that occurs before the C program is compiled and dynamic for checking at the time at which the program is executed.

A static-analysis paper published near the writing of this thesis declared that the technique “results in a high number of false positives” (Shahab, Alenezi, Nadeem, & Asif, 2020) [8]. A Naval Postgraduate M.S. thesis (Wikman, 2020) outlines many facets of static BO analysis, including many contemporary programs that are used for static analysis

# Chapter 3

# Debugging Buffer Overflow in C++ using GDB and Valgrind

## Introduction

Buffer overflow is a common issue in C++ programming, where an operation writes more data into a fixed-length block of memory (buffer) than it can hold. This chapter will guide you on how to use two powerful tools, GDB and Valgrind, to debug and handle buffer overflow issues in C++ code.

## 3.2 Understanding Buffer Overflow

Before diving into debugging, it’s crucial to understand what buffer overflow is. A buffer overflow occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program may crash or, even worse, it could lead to a security vulnerability where a malicious user can execute arbitrary code.

#include<iostream>

using namespace std;

int main() {

char buffer[10];

strcpy(buffer, "This string is too long for the buffer");

return 0;

}

In the above code, the string is too long for the buffer, causing a buffer overflow.

## 3.3 Debugging with GDB

GDB, the GNU Project Debugger, allows you to see what is going on ‘inside’ a program while it executes.

### Compiling the Program for Debugging

Before we can debug, we need to compile our program with the -g flag to include debugging information.

g++ -g -o test program.cpp

### Starting GDB

You can start GDB with your program as follows:

gdb test

### 3.3.3 Setting Breakpoints

You can set breakpoints at specific lines of your code. When the program execution reaches a breakpoint, it pauses, allowing you to examine the state of the program.

break main

### Running the Program

You can run the program inside GDB using the run command. If a breakpoint is hit, the program will pause.

run

### Examining Variables

You can print the contents of a variable with the print command.

print buffer

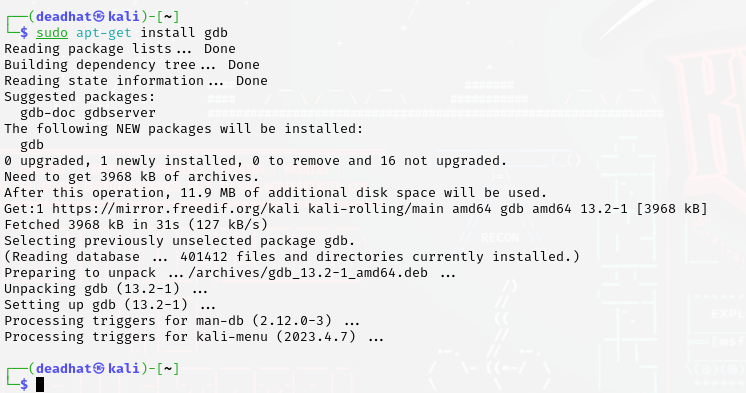


Figure 6: GDB installation

## Debugging with Valgrind

Valgrind is a powerful tool that can detect memory leaks and buffer overflows.

### Installing Valgrind

You can install Valgrind using the package manager for your system. For Ubuntu, you can use:

sudo apt-get install valgrind

### Running Valgrind

You can run your program with Valgrind as follows:

valgrind --leak-check=yes ./test

Valgrind will then launch your program, and monitor its memory usage for buffer overflows and leaks.



Figure 7: Valgrind installation

# Chapter 4

# Analyzing the code of Buffer Overflow mitigation tool

## **bound\_checking.cpp**:

1. This file defines a function `applyBoundChecking` that takes two parameters: `inputFilename` and `outputFilename`.
2. It opens the input and output files, checks if they are successfully opened, and prints an error message if not.
3. It reads each line from the input file and writes it to the output file.
4. Finally, it closes both files and prints a success message.

Let's break down the `**bound\_checking.cpp**` code step by step:

### Header Inclusions:

```cpp

#include <iostream>

#include <fstream>

#include <string>

```

1. This code includes necessary header files for input/output operations (`iostream`), file stream operations (`fstream`), and string manipulation (`string`).

### Function Definition:

```cpp

void applyBoundChecking(const std::string& inputFilename, const std::string& outputFilename)

```

1. This function is named `applyBoundChecking` and takes two parameters: `inputFilename` and `outputFilename`, both of type `std::string`.
2. It's designed to apply bound checking to a file specified by `inputFilename` and write the result to another file specified by `outputFilename`.

### Opening Input File:

```cpp

std::ifstream inputFile(inputFilename);

if (!inputFile.is\_open()) {

std::cerr << "Error: Unable to open input file " << inputFilename << std::endl;

return;

}

```

* 1. This code segment attempts to open the input file specified by `inputFilename` using an input file stream (`ifstream`).
  2. If the file cannot be opened, it prints an error message to the standard error stream (`cerr`) and exits the function.

### Opening Output File:

```cpp

std::ofstream outputFile(outputFilename);

if (!outputFile.is\_open()) {

std::cerr << "Error: Unable to open output file " << outputFilename << std::endl;

return;

}

```

1. Similar to the input file, this segment attempts to open the output file specified by `outputFilename` using an output file stream (`ofstream`).
2. If the file cannot be opened, it prints an error message to `cerr` and exits the function.

### Reading and Writing Lines:

```cpp

std::string line;

while (std::getline(inputFile, line)) {

outputFile << line << std::endl; }

1. Inside a loop, this code reads each line from the input file using `std::getline`.
2. Each line read from the input file is immediately written to the output file using the output file stream `outputFile`.
3. This effectively copies the contents of the input file to the output file.

### Closing Files:

```cpp

inputFile.close();

outputFile.close();

```

1. After the copying process is complete, both the input and output files are closed using the `close()` method of file streams.

### Success Message:

```cpp

std::cout << "Bound checking applied successfully. Output written to " << outputFilename << std::endl; ```

1. If the function executes without errors, it prints a success message to the standard output stream (`cout`), indicating that bound checking has been successfully applied and specifying the name of the output file.

## parsing.cpp:

1. This file defines a function `parseInputFile` that takes a filename as input and returns a vector of strings representing array accesses found in the file.
2. It opens the input file, reads each line, and searches for array accesses using string manipulation.
3. Array accesses are identified based on the presence of square brackets `[ ]`.
4. It returns a vector containing the identified array accesses.

let's analyze the `**parsing.cpp**` code step by step:

### Header Inclusions:

```cpp

#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <string>

```

1. This code includes necessary header files for input/output operations (`iostream`), file stream operations (`fstream`), string stream operations (`sstream`), and data structures like vectors (`vector`) and strings (`string`).

### Function Definition:

```cpp

std::vector<std::string> parseInputFile(const std::string& filename)

```

1. This function is named `parseInputFile` and takes a single parameter: `filename`, which is a constant reference to a string.
2. It's designed to parse an input file specified by `filename` and extract array accesses from each line.

### Opening Input File:

```cpp

std::ifstream inputFile(filename);

if (!inputFile.is\_open()) {

std::cerr << "Error: Unable to open input file " << filename << std::endl;

return arrayAccesses;

}

```

1. This code segment attempts to open the input file specified by `filename` using an input file stream (`ifstream`).
2. If the file cannot be opened, it prints an error message to the standard error stream (`cerr`) and returns an empty vector of strings.

### Parsing Array Accesses:

```cpp

std::string line;

while (std::getline(inputFile, line)) {

std::istringstream iss(line);

std::string word;

while (iss >> word) {

if (word.find('[') != std::string::npos) {

arrayAccesses.push\_back(word);

}

}

}

```

1. Inside a loop, this code reads each line from the input file using `std::getline`.
2. For each line, it creates a string stream (`istringstream`) to tokenize the line into individual words.
3. Within an inner loop, it extracts each word from the line and checks if it contains a left square bracket `[` using the `find` method.
4. If a word contains a left square bracket, it's considered an array access, and it's added to the `arrayAccesses` vector.

### Closing Input File and Returning Result:

```cpp

inputFile.close();

return arrayAccesses;

```

1. After parsing is complete, the input file is closed using the `close()` method of the file stream.
2. The function returns the vector `arrayAccesses`, which contains all the identified array accesses from the input file.

In summary, the `parseInputFile` function reads the contents of an input file, tokenizes each line into words, and identifies array accesses by searching for left square brackets. It then returns a vector containing all the identified array accesses.

## safe\_unsafe\_fn.cpp:

1. This file contains lists of unsafe and safe functions commonly associated with buffer overflow vulnerabilities.
2. It defines functions `containsUnsafeFunction` and `containsSafeFunction` to check for the presence of unsafe and safe functions in a given line of code, respectively.
3. It performs string searches within each line to identify the occurrence of unsafe or safe functions based on predefined lists.

let's analyze the `**safe\_unsafe\_fn.cpp**` code step by step:

### Header Inclusions:

```cpp

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include <algorithm>

#include <sstream>

```

1. This code includes necessary header files for input/output operations (`iostream`), file stream operations (`fstream`), string manipulation (`string`), data structures like vectors (`vector`), algorithms (`algorithm`), and string stream operations (`sstream`).

### Global Variables:

```cpp

std::vector<std::string> unsafeFunctions = {...};

std::vector<std::string> safeFunctions = {...};

```

1. This code defines two global vectors: `unsafeFunctions` and `safeFunctions`, which contain lists of unsafe and safe functions, respectively.

### Function to Check Unsafe Function:

```cpp

bool containsUnsafeFunction(const std::string& line) {...}

```

1. This function checks whether a given line of code contains any of the unsafe functions listed in the `unsafeFunctions` vector.
2. It iterates through the `unsafeFunctions` vector and uses the `find` method to search for each function name in the input line.
3. If any unsafe function is found, it returns `true`; otherwise, it returns `false`.

### Function to Check Safe Function:

```cpp

bool containsSafeFunction(const std::string& line) {...}

```

1. This function checks whether a given line of code contains any of the safe functions listed in the `safeFunctions` vector.
2. Similar to the `containsUnsafeFunction` function, it iterates through the `safeFunctions` vector and searches for each function name in the input line.
3. Additionally, it includes some extra checks based on specific conditions for certain safe functions (e.g., checking for `sizeof` and `- 1` for `memcpy\_s`, or checking for `sizeof` in combination with `fgets`).

## static\_code\_analysis.cpp:

* 1. This file defines functions for static code analysis to detect potential issues like memory leaks, coding standards violations, bugs, and security vulnerabilities.
  2. Function `isProperVariableName` checks if a variable name follows proper naming conventions.
  3. Function `performStaticCodeAnalysis` reads each line of code from a file, analyzes it for potential issues, and generates a summary report.

let's break down the `**static\_code\_analysis.cpp**` code step by step:

### 4.4.1 Header Inclusions:

```cpp

#include <iostream>

#include <fstream>

#include <string>

#include <cctype>

```

1. This code includes necessary header files for input/output operations (`iostream`), file stream operations (`fstream`), string manipulation (`string`), and character handling functions (`cctype`).

### Function to Check Proper Variable Name:

```cpp

bool isProperVariableName(const std::string& variableName) {...}

```

1. This function checks whether a given variable name follows proper naming conventions.
2. It verifies if the variable name starts with a lowercase letter, contains only alphanumeric characters or underscores, and has at least one letter or digit.
3. If the conditions are met, it returns `true`; otherwise, it returns `false`.

### Function to Perform Static Code Analysis:

```cpp

void performStaticCodeAnalysis(const std::string& filename) {...}

1. This function performs static code analysis on a file specified by `filename`.
2. It opens the file for reading using an input file stream (`ifstream`).
3. Inside a loop, it reads each line from the file and analyzes it for potential issues.
4. It checks for various issues such as potential memory leaks, coding standards violations, potential bugs (like division by zero, null pointer dereference, out-of-bounds array access), and security vulnerabilities (like buffer overflow).
5. For each detected issue, it prints a message indicating the type of issue and its location within the file.
6. After analyzing all lines, it generates a summary report summarizing the analysis results, including the total number of issues found and the breakdown of issues by type.

### Summary Report:

cpp

std::cout << "Summary Report:" << std::endl;

...

1. This code segment prints a summary report summarizing the analysis results.
2. It includes the total number of issues found, as well as the counts for potential memory leaks, coding standards violations, potential bugs, performance issues, and security vulnerabilities.

In summary, `static\_code\_analysis.cpp` provides functionality to perform static code analysis on a given file. It checks for various types of issues and generates a summary report outlining the analysis results. The analysis covers potential memory leaks, coding standards violations, potential bugs, and security vulnerabilities, providing valuable insights for improving code quality and security.

## main.cpp:

* 1. This file orchestrates the entire tool's functionality by integrating the other components.
  2. It includes functions to remove comments from the input code, parse array accesses, apply bound checking, perform static code analysis, and check for safe and unsafe functions.
  3. The `main` function reads input from a file, applies various mitigation techniques, and performs analysis to identify and report potential vulnerabilities.

### Header Inclusions:

```cpp

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

```

1. This code includes necessary header files for input/output operations (`iostream`), file stream operations (`fstream`), string manipulation (`string`), and vector data structure (`vector`).

### Custom Header Inclusions:

```cpp

#include "bound\_checking.cpp"

#include "parsing.cpp"

#include "static\_code\_analysis.cpp"

#include "safe\_unsafe\_fn.cpp"

```

1. This code includes custom header files (`bound\_checking.cpp`, `parsing.cpp`, `static\_code\_analysis.cpp`, `safe\_unsafe\_fn.cpp`) that contain functions and logic for specific tasks like bound checking, parsing, static code analysis, and checking safe/unsafe functions.

### Function to Remove Comments:

cpp

std::string removeComments(const std::string& line) {...}

1. This function removes comments from a given line of code. It iterates through the line character by character, skipping characters within block comments (`/\* ... \*/`) or line comments (`// ...`). It also handles comments within double quotes to avoid removing commented out code within strings.

### Main Function:

```cpp

int main() {...}

```

1. The `main` function orchestrates the entire process.
2. It opens an input file (`input1.cpp`) for reading and an output file (`removed\_comments.cpp`) for writing.
3. Inside a loop, it reads each line from the input file, removes comments using the `removeComments` function, and writes the modified line to the output file.
4. After processing all lines, it closes both input and output files.
5. It then parses the modified file (`removed\_comments.cpp`) to extract array accesses using the `parseInputFile` function.
6. It applies bound checking to the modified code using the `applyBoundChecking` function, creating an output file (`output.cpp`).
7. It performs static code analysis on the modified file using the `performStaticCodeAnalysis` function.

Finally, it checks for safe and unsafe functions within the modified code using the `containsUnsafeFunction` and `containsSafeFunction` functions respectively, and prints out any found occurrences.

In summary, `main.cpp` is the central component that drives the buffer overflow mitigation tool. It reads an input file, removes comments, applies various analyses and modifications (such as parsing, static code analysis, and bound checking), and then reports on potential issues and the presence of safe/unsafe functions.

These components work together to mitigate buffer overflow vulnerabilities by identifying, analyzing, and securing code against common attack vectors.

# Chapter 5

## 5.1 Utilizing Valgrind and GDB Debugger for Comprehensive Code Analysis and Issue Detection

GDB (GNU Debugger) is a powerful tool used for debugging and analyzing programs written in various programming languages, including C, C++, and assembly. It allows developers to inspect the internal state of a program, track down bugs, and analyze its behavior during execution.

Here's how we'll use GDB to analyze the code snippets and identify any issues:

### Info Registers:

- View CPU register values to detect any anomalies or signs of corruption.

### 5.1.2 Adding Breakpoints and Running:

- Set breakpoints at key points in the code to halt execution and inspect the program's state.

- Run the program and observe its behavior step by step.

### Valgrind Summary:

- Employ Valgrind to detect memory-related issues such as leaks or errors.

- Analyze the Valgrind summary for any memory-related problems.

### Disassembling Code:

- Disassemble the code to inspect the low-level assembly instructions.

- Look for unexpected behavior or instructions that could indicate errors.

By utilizing GDB's features, we can systematically analyze the code snippets, identify potential flaws, and ensure the reliability and security of the programs.

## Lets Analyze the code snippet without errors:-

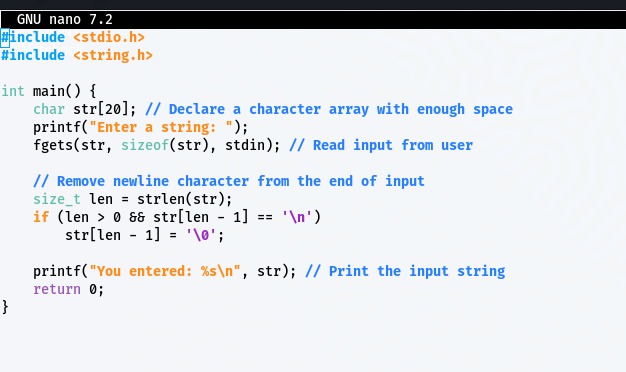


Figure 8 The code Snippet

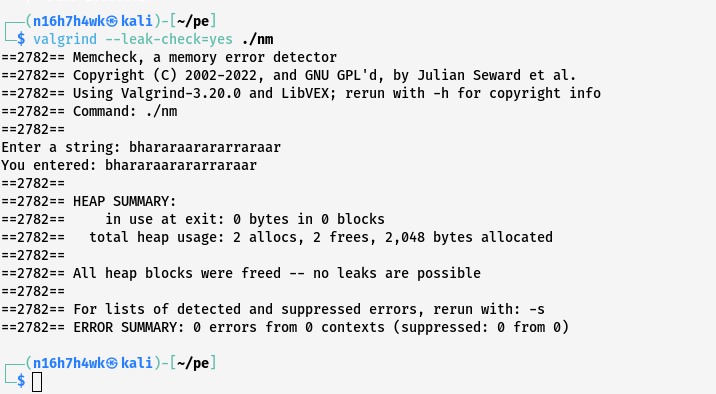


Figure 9: Analyzing using Valgrind



Figure 10 the disas value of code during analysis

## Lets Analyze the code snippet with errors:-

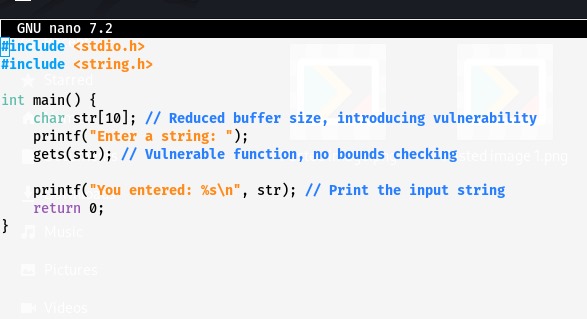


Figure 11 Analyzing the code

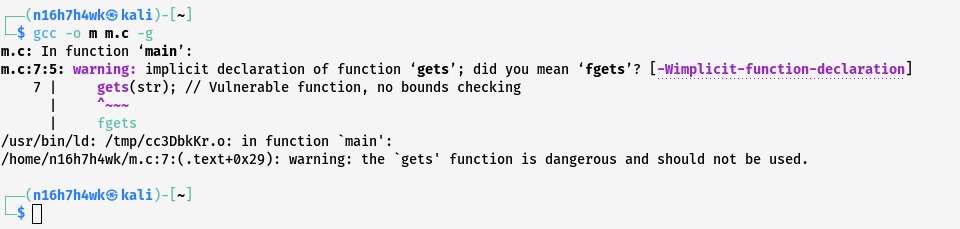


Figure 12 Implementation with gcc

Here We are Encountering gcc error because vulnerable function is used.

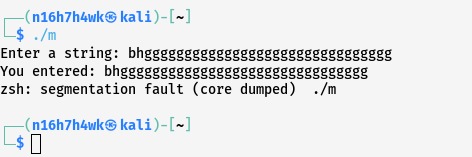


Figure 13 Segmentatio fault

There is a segementation fault identified in the given snippet.



Figure 14 The disas analysis

During the analysis of code, Valgrind provides a disas code which provides diffent values of code in the memory.



Figure 15 Valgrind Summary

This is the valgrind summary after analyzing the given code snippet.

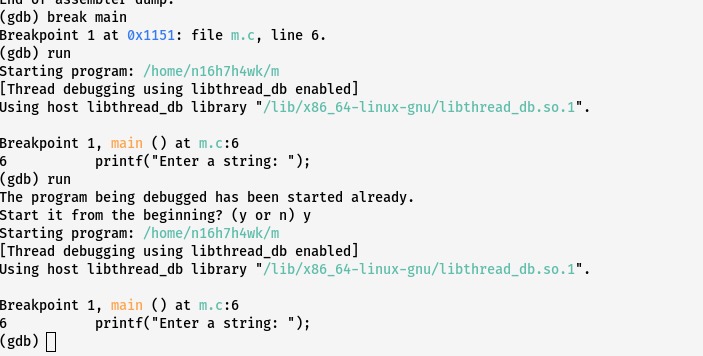


Figure 16 Break Points

Now let’s add the break points and then run the code again.

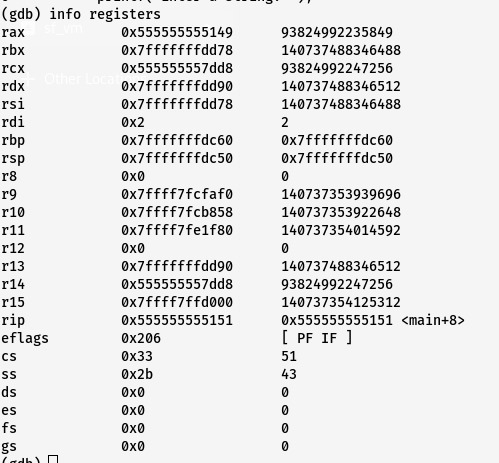


Figure 17The Register Values

Here we got the values of registers

# References

* 1. Nurit Dor, Michael Rodeh and Mooly Sagiv. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. 8th International Static Analysis Symposium. To appear, July 2001.
  2. David Wagner, Jeffrey S. Foster, Eric A. Brewer and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed System Security Symposium. February 2000.
  3. Y. Nong, H. Cai, P. Ye, L. Li, and F. Chen, ‘‘Evaluating and compar[1]ing memory error vulnerability detectors,’’ Inf. Softw. Technol., vol. 137, Sep. 2021, Art. no. 106614.
  4. X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, ‘‘Towards efficient heap overflow discovery,’’ in Proc. 26th USENIX Secur. Symp. (USENIX Secur.), 2017, pp. 989–1006.
  5. B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, ‘‘A large-scale empirical study on vulnerability distribution within projects and the lessons learned,’’ in Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., New York, NY, USA, Jun. 2020, pp. 1547–1559.
  6. Kernighan, B. W., & Ritchie, D. M. (1978). The C Programming Language. Englewood Cliffs, New Jeresy: Prentice Hall
  7. Shaw, A. (2014). Program Transformations to Fix C Buffer Overflow. ICSE Companion 2014: Companion Proceedings of the 36th International Conference on Software Engineering, 733-735.
  8. Wikman, E. C. (2020). Static Analysis Tools for Detecting Stack-Based Buffer Overflows. Monterey: Naval Postgraduate School

# Acknowledgements

If we thought anyone cared, if we thought anyone would even be reading this, we'd probably make an effort to keep up appearances until the last possible moment. But no one does, and no one will. So, we can pretty much say exactly what we think.

Oh, yes, the *acknowledgements.* We think not. We did it. We did it all, by ourself.

|  |  |  |
| --- | --- | --- |
| |  | | --- | | ABHIYANK YADAV 22MIP10074  BHARAT RAGHUVANSHI 22MEI10006  HIREN SHUKLA 22MIP10070  ABHIMANYU NARAYAN KANJOLIA 22MIP10055 | |  | |