# C Buffer Overflow, Heap and Stack Corruption and Analysis

Secure Software Systems | Research paper

Thissera P. T. D. | IT 19022666

12th of May 2020

Email: it19022666@my.sliit.lk

***ABSTRACT***

***In information security and information programming, a buffer overflow vulnerability is an anomaly where a program writes data to a buffer that overwrites the boundary of the buffer and adjacent memory locations. Over the past few years, buffer overflows vulnerability have become the most common form of security risk. Also, buffer overflow risk status is dominated by remote network penetration risk attempts to gain partial or complete control over an anonymous Internet user. If the buffer can effectively eliminate the risk of overflow, it can eliminate a large number of its most serious security threats. This article provides an in-depth survey of the different types of buffer overflow risk and attack. It also surveys the various safety measures required to reduce the risk of buffer overflows. Also, the methods presented in this article have the effect of accurately predicting the risk of software buffer overflow in C / C ++ and Java programs. By experimenting with techniques that can eliminate the problem of buffer overflow, it is possible to get rid of those safety hazards.***

***Key terms:*** *Heap corruption, Stack corruption, buffer overflow attack, vulnerability*

## 1. INTRODUCTION

Over the past few years, buffer overflows have been identified as the most common security risk, with buffer overflow risk dominating remote network penetration risk. What happens here is that an anonymous Internet user attempts to gain partial or complete control over a host. They represent one of the most serious class security threats, as they allow any individual to gain complete control of a host using such a buffer overflow attack. Of all the defense attacks, a significant portion of the buffer overflow threat is made up simply because the risk of buffer overflow is high. Furthermore, this vulnerability is easy to exploit.

While the risk of buffer overflow in the remote penetration class is particularly high, the risk of buffer overflow properly presents what the attacker needs. The injected attack code runs with the privileges of the program at risk, allowing the attacker to initiate any action necessary to control a host computer. Overflow buffers separated by heap or datasets are also a threat. Such a situation is usually difficult for an attacker to exploit. An attacker must find a critical security reserve, such as a user ID or file name, that is, the value of the rewritable memory. Sometimes a function indicator stored on the heap can be arbitrarily redirected or modified. The indicator is then distorted, and the attacker can then execute the selection code. Such attacks have occurred in recent years. Risks of buffer overflow can be easily detected using traditional testing methods, such as branch covering or segment. But there are problems here. The problem is that program values and execution routes that are required to detect a security error are not displayed during regular operation. Therefore, it is also very difficult to show during testing. It is advisable to use a test method that can monitor the length of the possible buffer when the program is executing. There are preconditions for library work when a buffer does not overflow. If length can cause conditions to be violated, a buffer overflow may occur.

Throughout this research, C focuses on the programming language, and C provides a small syntactic test. Limitations, and c Programs and functions are tedious with minimal error checking. Therefore, many C programs suffer from security issues including buffer overflow.

## 2. RESEARCH STATEMENT / OBJECTIVES

The core concept of this research is studying more research articles and find a topic related to a theoretical foundation of secure software system and apply it to a real-world practical scenario. It provides an in-depth study of topics such as static analysis and dynamic analysis of C buffer overflow, how an anonymous attacker can find buffer overflow vulnerability and ways to arrange for suitable code to be in the program's address space and ways to cause the program to jump to the malicious code. Using that buffer overflow vulnerability hacker can overflow the buffer and can overwrite the adjacent program state with a near arbitrary sequence of bytes.

## 3. ANALYSIS

Programs written with a greater focus on programming efficiency and code length are more vulnerable to attacks than a security language written using the C language. According to the programming, the C language can be considered as a very flexible and powerful language. It is sufficient to specify a pointer-based call, either by direct memory mode or by a text string approach, and the latter suggests that the length of the actual buffer may not be controlled even during library work on text threads. The potential for exposure to the stated length overflow is very high. Before looking further into the mechanism by which an attack is performed, it is important to focus on several technical aspects of program execution and memory management functions.

### 3.1 STATIC ANALYSIS

A number of tools can be used to check the source code in the event of a buffer overflow. The ITS4 tool is similar to a large class, scanning C and C ++ source codes for known dangerous library calls, as well as statistically scanning secure-critical C source code for insecurities. It also tests the logic of minor calls and reports the severity of the threat.

For example: fixed length threads to a copying library call buffer are less intensely graded than the contents of the array buffer than copying library calls. It also looks for competing terms for other potential issues. Other tools include RATS and Splint.

An integer range analysis should be used to detect potential buffer overflows. C-threads can be considered as abstract data types and can be assumed to be *"strcpy"* and *"strcat"* guided by C standard library functions. An allocated buffer overflows whenever the length of a thread should exceed the maximum length of the thread. The flow-sensitive analysis, pointer transliteration, and functional call handling means that the thread length and the amount of memory allocated are approximately greater than the actual values required for each function. The primary problem with static analysis methods is their accuracy, and because of this common problem, scanning the source code does not usually determine the risk of buffer overflow, and all such tools can be used to determine the location of the buffer overflow. Dynamic tools take a different approach.

### 3.2 DYNAMIC ANALYSIS

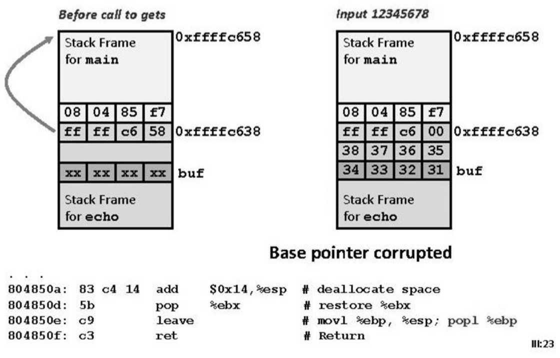
Dynamic analysis programs are activated and tested to determine if a buffer overflow occurs. Exceeding the limits set for a buffer may result in hardware errors, compilers needing code to check the limits, or to configure data structures in memory. Those additional tools are often disabled in the name of efficiency. Various tools enhance or replace these capabilities.

For example: The Purify tool detects many types of erroneous memory, including access to a buffer beyond the limit.

Program testing also takes place as an alternative approach. Fuss, a tool used to test standard Unix utilities, also provides inputs containing random spellings. Problems with indicators and array distortion, including defects in a buffer overflow, can be considered as the primary cause.

A major problem in dynamic analysis is the need for test data that can cause overflow. This dynamic occurs when an overflow is given to a program without data, and the potential for buffer overflow to occur cannot be detected by technical means.

## 4. BUFFER OVERFLOW VULNERABILITY AND THE ATTACK



```
. . .
804850a:  83 c4 14   add     $0x14,%esp   # deallocate space
804850d:  5b         pop     %ebx         # restore %ebx
804850e:  c9         leave                # movl %ebp, %esp; popl %ebp
804850f:  c3         ret                  # Return
```

The overall goal of a buffer overflow attack is to suppress the activity of a privileged program. The attacker has the ability to control the program, and if the program has enough privileges, the attacker can then control the host. Normally an attacker launches an attack to get the root privileges in a program. Although not always, an attacker sometimes uses code similar to "exec (sh)" to retrieve the root shell. If you want to get it that way, an attacker must score two sub-goals and then the attacker can get what he wants.

### 4.1 WAT TO ARRANGE THE SUITABLE CODE TO BE IN THE PROGRAM'S ADDRESS SPACE

There are basically two ways to set up an attack code and there are two basic ways to set up this attack. The attack code should be written in the address space of the victim program, and the first is whether the injection was already in use or not.

#### 4.1.1 INJECT

The attacker provides a thread to the program as an input. Also, the threads stored in the program buffer will contain native CPU bytes. The victim's program buffer is used to store attack codes for the attacker while instructing them to attack the platform. The attacker does not need to overflow any type of buffer and can inject into a complete and fair buffer after adequate payment. Also, the buffer can be found anywhere, on the stock with automated variables and piles with malloc variables, and some subtle methods and facts that exist in an initial or non-initial static data area.

#### 4.1.2 ALREADY IN USE

Things to do in most cases Programs about the needs of a code attacker may already exist in the address space. The attacker only needs the parameter code and then the program jumps to that parameter code. For example: To execute the code created and entered by the attacker, go to "Exec (" / bin / sh ")" and "arg" means "exec (arg)" which is an argument embedded in the string pointter. A code that exists in libc. All an attacker needs to do is change the index to point to "/ bin / sh" and jump to the appropriate instructions in the libc library.

### 4.2 WAYS TO CREATE A MALICIOUS CODE CREATION PROGRAM.

The primary method is to overflow a buffer with weak or non-existent bounds when testing its input with the intent of corrupting the status of an adjacent part of the program.

For an example: by overflowing the buffer, the attacker or hacker can overwrite the adjacent program state with a near arbitrary sequence of bytes.

Notable factors among the overflows are the state where the state is located and where the state is located in the memory structure. Activation records, performance indicators and lounge jumper buffers are almost all at risk. Each time a function is called, an active record is kept, including the return address to which the program should exit, or the point of the injected code.

Addressing Reports in Corrupted Activity Reports Automatic variables overflow or applying local buffer to the function. Corruption of the return address in the activation report causes the attacker to jump into the attack code when the victim's activity returns, and the return address is canceled. This buffer overflow is called a "stack smashing attack." And it has now become the majority of buffer overflows.

If there is an overflow buffer next to the function index, that index can be rewritten. At the same time, the next time the program calls the function, it must jump to the attack code. Similarly, buffers such as "setjmp / longjmp" attached to an overflow buffer can be overwritten.

**setjmp – saves the current environment into the variable environment for later use by the function longjmp( )*

**longjmp - The longjmp( ) function shall restore the environment saved by the most recent invocation of setjmp( ) in the same thread, with the corresponding jump_buffer argument,*

## 5. CONTROL FLOW CORRUPTION TECHNIQUES

The most common and simplest way to buffer is to combine the overflow attack with the injection technology. It also reports corruption on a single thread for activation. The attacker finds an automatic variable that can cause it to overflow and feeds a large thread into the program at once. The buffer will be left full to modify the activation record containing the injected attack code. This can be considered a temporary plate for an attack. This is because allocating a small local buffer or parameter input to get a C user is so common. There are many codes that can be affected by this type of attack, and injection and contamination will never work at the same time. An attacker can inject one buffer without overflow using the code, and overflow into another buffer to corrupt a code index. This is usually done to check the boundaries of an overflow buffer. Where it falls on error and, therefore, the buffer overflows only up to a certain number of bytes. The attacker will not be able to retain the code of the vulnerable buffer. For that reason, the code is inserted into a buffer after it has been sufficiently modified. If the attacker attempts to use a resident code without using an injection, the parameters are generally essential for an attacker.

### 2.1 HEAP CORRUPTION

Heap corruption in piles begins during a program that seeks to damage the distributor's view of the piles. The relative consequence of this would be to consider the negative as well as the memory leakage. In those cases, some memories do not come to mind. Then there is the inability to access programs. Otherwise, it can be fatal. Memory loss is also unavoidable. A memory error may occur in the allocator when manipulating one or more of the free lists included in the pool, even after they have been corrupted. Identifying the source of the corruption is a difficult task when the source of the error is located in another part of the code base.

This heap corruption is caused by a program trying to free up memory, attempting to separate those memories after they have been released, corrupting piles to block memory, and the error occurring from a later part of the memory. These situations can also be caused by the use of conflicting memory components, due to the multitasking of the program, and by changing the strategy for allocating memories.

The contradictory partitions contained herein, when used in contradictory chunks, have the potential to contaminate allocator information, pile information, and distributor views regarding the use of a program that writes out of bounds and blocks the memory they use. Part of the memory may be included before or after the blocks are used for display. It is possible to set it aside or not. In such a case, the allocator may cause an error when the memory is set aside or released, i.e., in an unrelated attempt. This is called Contiguous memory blocks.

*//example for heap corruption*

*char *buffer1 = malloc(12 *sizeOf(char));*

*char *buffer2 = malloc(12 *sizeof(char));*

*strcpy(buffer2, “mywordshare”);*

*strycpy(buffer1, “12345678901234561234567890123456”);*

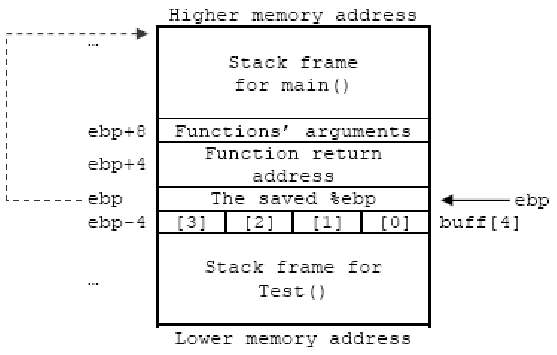
*printf(“buffer1 value: %s\n”, buffer1);*

*printf(“buffer2 value: %s\n”, buffer2);*

*printf(“buffer1 address: %p\n”,(void *) buffer1);*

*printf(“buffer2 address: %p\n”,(void *) buferf2);*

### 2.2 STACK CORRUPTION



Stack is a special memory area within the memory. It uses store local functions variables and context during function calls. Store temporary variables, store variables declaration and declare variables and mainly it is a temporary storage memory.

Bulk corruption refers to memory corruption as a result of a program operation. You have to rewrite the vertical element using a copy of the infinite array. That is, local variables and the return address can be pointed out. As a result, the program crashes or exhibits undefined behavior. When there is a difference in the value of a parameter passed by the call function, the stack corruption can be suspected. Local variables involved in calling and calling someone should be considered when a bulk corruption is detected to find the possible sources of memory corruption.

```
// example for stack corruption
Int main(void)
{
        Int buffer2[16]="overwriteme";
        Char buffer1 = 16;
        Strcpy(buffer1, " ");
  Printf("buffer1 value: %s\n", buffer1);
  Printf("buferf2 value: %s\n", buffer2);
  Printf("buffer1 address: %p\n", (void *)buffer1);
  Printf("buffer2 address: %p\n", (void *)buffer2);
}
```

## 6. BUFFER OVERFLOW DEFENSES

The buffer consists of four basic approaches to protecting against the risk of overflow and attack. Of the four approaches, the first to describe the brute force method of *correct code* writing. Second one is the creation of storage areas for *buffers non-executable*. It also shows how to prevent the

attacker from injecting the attack code. This approach stops most buffers from overflowing. However, attackers do not need to inject attack codes to make a buffer against the overflow attack.

Third, check out the *array border checking* for all array access. This method does not prevent overflow and therefore eliminates the problem of buffer overflow. But there are significant costs involved. Finally, the indirect compiler approach is to *test the integrity of the code* indicators before dereferencing. The buffer overflow attacks cannot be done while this technique is being performed and it is difficult to create many buffer overflows and the attacks it launches without stopping.

The buffer overflow protection organizes and modifies the data and information that is divided into sections. This will add a canary value. In cases where the bulk buffer is destroyed by an overflow, it can be seen that the buffer overflows before recollection. Verification of canary values can execute and terminate the affected program. Being the end of the real estate market. Limit checks include buffer overflow protection methods. It checks the access to the reserved partition and cannot go beyond the reserved space. Excessive filling of a stacked buffer and execution of programs are more likely to have a greater impact than filling a buffer on a pile. Equipped with excavation-specific protection against pile-only overflows.

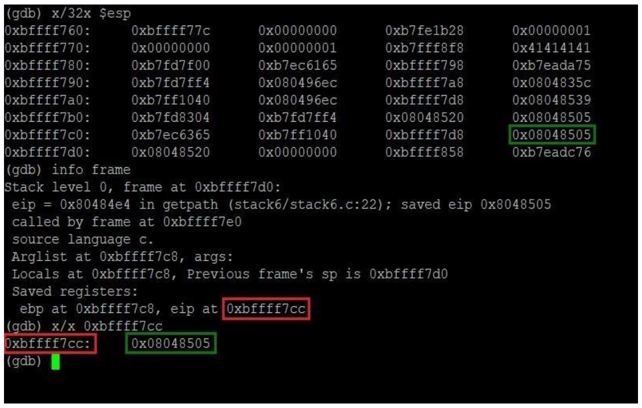
### 6.1 WRITE CORRECT CODE

Writing the correct code is more important, but there should be a better understanding of how to write those codes when writing using the C language. Frequent safe and vulnerable programs are carried out. Accordingly, various techniques and tools have been developed to assist developers. The simplest way to reduce the risk of buffer overflow is to capture the source code for *strcpy* and potential library calls. Versions of the C standard library that are complained about when joining *sprintf* programs that do not check the length of the arguments should also be developed for risky functions such as *strcpy* and *sprintf*. The primary purpose of manually auditing a large number of codes is to appear with code audit teams to look for common security vulnerabilities, such as buffer overflow and file system running conditions. The buffer overflow risk is subtle, and security codes that use security alternatives such as *snprintf* and *strncpy* may contain buffer overflow information.

### 6.2 NON-EXECUTABLE BUFFERS

The basic premise is to stop the execution of datasets on the address space of a victim program. The inputs that are inserted into a victim program make it impossible for the attacker to execute the codes injected into the buffer. This technology was created in older computer systems. But nowadays, programs included in the MS Windows system and the UNIX system include various performance optimizations based on their ability to emit dynamic code from data sets. Accordingly, it is necessary to make it possible to execute data sections within all programs without compromising compatibility for significant programs. However, stack segment activation can be made unworkable and must be done to protect the compatibility of most programs. Can download kernel patches for both LINUX and Solaris (Oracle Solaris). This can cause compatibility issues as the addresses created as part of the program cannot be executed in space and do not contain the bulk codes that are legally relevant to the program. There are two exceptional cases in LINUX operating system where executable code must be placed on the stack.

I. Signal Delivery
II. GCC Trampolines



```
(gdb) x/32x $esp
0xbffff760:     0xbffff77c      0x00000000      0xb7fe1b28      0x00000001
0xbffff770:     0x00000000      0x00000001      0xb7fff8f8      0x41414141
0xbffff780:     0xb7fd7f00      0xb7ec6165      0xbffff798      0xb7eada75
0xbffff790:     0xb7fd7ff4      0x080496ec      0xbffff7a8      0x0804835c
0xbffff7a0:     0xb7ff1040      0x080496ec      0xbffff7d8      0x08048539
0xbffff7b0:     0xb7fd8304      0xb7fd7ff4      0x08048520      0x08048505
0xbffff7c0:     0xb7ec6365      0xb7ff1040      0xbffff7d8      0x08048505
0xbffff7d0:     0x08048520      0x00000000      0xbffff858      0xb7eadc76
(gdb) info frame
Stack level 0, frame at 0xbffff7d0:
 eip = 0x80484e4 in getpath (stack6/stack6.c:22); saved eip 0x8048505
 called by frame at 0xbffff7e0
 source language c.
 Arglist at 0xbffff7c8, args:
 Locals at 0xbffff7c8, Previous frame's sp is 0xbffff7d0
 Saved registers:
  ebp at 0xbffff7c8, eip at 0xbffff7cc
(gdb) x/x 0xbffff7cc
0xbffff7cc:     0x08048505
(gdb)
```

### 6.3 ARRAY BOUNDS CHECKING

It is essential to inject one or more codes for an overflow attack on a buffer, and contamination of the control flow will also be essential. Such an inoperable array is bordering and not like a buffer. Checking shows the risk of an overflow in a buffer and that the attack is almost complete. An array can never overflow, but this array overflow can be used for adjacent contamination program status. In order to check the boundaries of an array, almost everyone who writes and reads to the array must be checked and they are in range. Checking references in all arrays are called direct access, and in the most cases optimization methods can be used to clear multiple checks. There are several project implementation approaches for checking array boundaries, illustrated in detail below.

#### 6.3.1 COMPAQ C COMPILER

Using C and C ++ compilers, can create entries for each array, check the boundaries of the arrays, and patch them. The time taken for this can be termed as a process and it can also be termed as an optimization. Designed for alpha CPUs, the Compaq C compiler supports limited array border checks during check bounds. Boundary checks are limited, and alternatives are used. The following methods will confirm this.

- Only referenced clear arrays will be properly scrutinized. That is, "A [3]" is checked but "* (a + 3)" is not checked.
- It has been argued and adopted as to when all the C arrays present in a program are converted into indicators. There will be no proper checks on the restrictions on subroutine entries.
- Compiled using strcpy () function to check the boundaries of dangerous library functions. Checking the limits here would be dangerous if they were enabled.

Index arithmetic is one of the most widely used programs today, and therefore uses arithmetic arrays for array access and array arguments for functions, and these limitations are strict. While the limit check feature is used to a limited extent to debug programs, it can be difficult to ensure that existing buffers in a program are not exploited at risk of overflow.

#### 6.3.2 PURIFY: MEMORY ACCESS CHECKING

Purify is meant to be a tool for debugging memory usage for C programs. This Purify uses the encryption of object codes in devices used for all memory accesses. After interacting with Purify Linker and libraries, you will receive a program that can be implemented by standardized native ones. To ensure the legitimacy of the program, all existing arrays are scanned. Purify-security programs are generally functional, even when not in a special environment. But purify cannot be described as a product security tool. Although purify was tedious to build, one copy is said to show a purchase price of approximately $ 5,000.

#### 6.3.3 TYPE-SAFE LANGUAGES

Type-safe operations can only be performed on a variable due to potential hazards due to the lack of a buffer overflow C-type safety condition. Where the creative input to variable fu cannot be used to make arbitrary changes to the variable column. New security-sensitive code is written in secure languages, such as "Java or ML". The vast majority of code invested in existing security-sensitive applications and operating systems, consisting of millions of lines of code, is written in the C language. However, Java virtual machines can be referred to as a C program, and one of the ways in which JVM can be attacked is by pointing the buffer overflow to JVM itself. For this reason, applying a buffer for overflow type-protected languages can have better results by enforcing security system-type security.

### 6.4 USING CANARY VALUE

Buffer overflow protection modifies the organization of data within active call frames by inserting a "canary" value. It also indicates that a buffer overflowed before memory, when destroyed. This gives the whole class the advantage of attack prevention. The term canary refers to the values between a buffer and control data to monitor an overflow in a buffer. When the buffer overflows, the first data to be corrupted will usually be canary. Failure to verify Canary data will result in an overflow warning. And then handle it. For example, a canary value should not be confused with a sentinel value by deleting corrupt data. There are three types of canaries in use:

I. Terminator
II. Random
III. Random XOR

## 7. CONCLUSION

Provided a detailed classification of buffer overflow risk, attacks, and defense tactics in this test. Buffer overflows are suitable for this level of analysis because the majority of them are security risk issues, and the significant majority are remote penetration safety risk issues.

There is considerable potential to obscure addresses to limit the threat of attacks such as widespread buffer overflows. When this is done, random rearrangement of a computer program and its data storage space puts the primary risk of being exploited by buffer overflow attacks. Unlike many existing techniques, address blurring is a common mechanism that has a wide range of applications for many attacks related to memory errors. Thereby, an attack-specific mechanism can be applied to avoid known attack opportunities.

Because each system is obscured in different ways, even if an attacker successfully overwhelms one system on a network, the attack must be started from the beginning and the second will require a lot of effort before the system can be overthrown. In the context of autoimmunity, this factor will greatly reduce the spread of worms and viruses. Thus, this test provides simple and effective solutions to prevent the spread of viruses and worms by obscuring addresses.

## 8. FUTURE RESEARCHERS

The information provided in this section outlines future researchers. The first of these experiments was to find out the relationship between program size and speed when a buffer overflow occurs, and additional code is added to protect it. That extra code must be specific to the buffer overflow code. It is also important that it is critical to large programs. Knowledge and training on prior security coding can be effectively measured in these cases when training under the topic of secure C / C ++ encoding or under appropriate security encoding. Perfect exception handling of C / C ++ programs can also be applied during implementation.

Another interesting thing to explore in the future is the development of C / C ++ editors with educational information on the problem of buffer overflow, such as the "Intelligence feature". This research could be used in the future to reduce the risk of problems with the buffer overflow and to assess its effectiveness during operation. This project does not place much emphasis on compilation, runtime identification, and prevention solutions, other than the implementation of extensive exceptional handling, as most research is heavily funded.

For example: the convention used for C functional calling can be modified to the best and safest methods by reviewing how the bulk frame can be created and destroyed, i.e., how a processor supports the mechanism through its functional environment.

**AKNOWLADGEMENT**

I would like to pay my respects to Sri Lanka Institute of Information Technology Secure Software System (SSS) Model Lecturer Ms. Chethana Liyanapathirana and Mr. Lakmal Rupasignhe. First, we were able to gain a better understanding of her concept through her research paper article on a theoretical foundation and apply it to a real-world practical scenario. She would also like to share her knowledge with us and honor us for guiding us in the right path. Also, it was great to be able to complete this research properly and to pay more attention to it by giving a detailed explanation in order to enhance our knowledge properly. I would also like to extend my appreciation to my colleague Eradh Jayasundara, who has been instrumental in helping me in my research. Inspired by him, I was able to carry out this research better and see this research in new dimensions. I would also like to express my gratitude to all my family members for their innumerable methods and for saving me from many mistakes. I was able to enhance this study because of the generosity and uniqueness of everyone. Finally, I would like to thank my university for furthering my education. I would like to pay my respects to creating thousands of people like me and paving the way for future conquests of Sri Lanka and the world. With my 100% commitment and full effort for any assignment given, I look forward to successfully completing and striving to conquer the world of tomorrow.

## REFERENCES

[1] "What Is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks," Veracode, 2021. [Online]. Available: https://www.veracode.com/security/buffer-overflow.

[2] A. Goel, "An Introduction to Buffer Overflow Vulnerability," 2 December 2019. [Online]. Available: https://betterprogramming.pub/an-introduction-to-buffer-overflow-vulnerability-760f23c21ebb.

[3] "CHAPTER FIVE:CONCLUSION AND FUTURE WORK," 2008. [Online]. Available: https://www.tenouk.com/Bufferoverflowc/bufferoverflowvulexploitdemo5.html.

[4] M. B. Eric Haugh, "Testing C Programs for Buffer Overflow Vulnerabilities," University of California, California.

[5] P. W. C. P. S. B. Crispin Cowan, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," January 2000. [Online]. Available: https://www.researchgate.net/publication/232657947_Buffer_Overflows_Attacks_and_Defenses_for_the_Vulnerability_of_the_Decade.

[6] J. T. B. M. T. K. John Viega, "A static vulnerability scanner for C and C++ code," Reliable software technologies.

[7] "Debugging software crashes," EventHelix, 2021. [Online]. Available: https://www.eventhelix.com/embedded/debugging-software-crashes/#Stack%20Memory%20Corruption.

[8] E. Man, "C Buffer Overflow, Heap/Stack Corruption and Analysis," Engineer Man, Jun 13, 2017.

[9] D. Aspinal, "Secure Programming Lecture 4: Memory," Informatics @ Edinburgh, 25th January 2018. [Online]. Available: https://www.inf.ed.ac.uk/teaching/courses/sp/2017/lecs/04-overflow2-6up.pdf.

[10] W. Chen, "Heap Overflow Exploitation on Windows 10 Explained," Rapid7, Jun 12, 2019. [Online]. Available: https://www.rapid7.com/blog/post/2019/06/12/heap-overflow-exploitation-on-windows-10-explained/.

[11] "Buffer Overflow Attack," Imperva, 2021 . [Online]. Available: https://www.imperva.com/learn/application-security/buffer-overflow/.

[12] "Buffer Overflow," OWASP, 2021. [Online]. Available: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.

***AUTHOR PROFILE***

**Thissera P. T. D.**

*Born in Colombo, Sri Lanka on 23rd of July 1999. Studying at University of Sri Lanka Institute of Information Technology, Malabe, Sri Lanka. Received the certificate in INTRODUCTION TO CYBERSECURITY COURSE at Cisco Networking Academy, Sri Lanka, in 2020, INTRODUCTION TO IOT COURSE at Cisco Networking Academy in 2020* *and NDG LINUX UNHATCHED COURSE at Cisco Networking Academy in year 2020. Following BSc (Hons) in Information Technology Specializing in Cyber Security degree, Sri Lanka Institute of Information Technology (SLIIT) University, Malabe, Sri Lanka, From February 2019 to February 2023. She worked as COMPUTER APPLICATION ASSISTANT at WebstazyOne software company, Colombo 07. Her primary research interests include Critical infrastructure security in healthcare sector, Artificial intelligence, Internet of Things (IOT), Buffer overflowing attacks, security vulnerabilities, and Cyber Crimes and so on. Ms. Thissera is a member of IEEE community at SLIIT, Member of ISACA community in SLIIT and a member of Cyber Security Community in SLIIT. And she has number of achievements like ALL ISLAND, JUNIOR NATIONAL, INTER SCHOOL CHAMPIONSHIPS in BADMINTON. Ms. Thissera held number of positions in school level. Currently, she is in 3rd year 1st semester in her university life.*