NHL STENDEN
university of applied sciences

Written Communication 2
Research Report

Research Report Title: **Buffer Overflow Vulnerability, Understanding the Vulnerability of the Decade**

Made by:

Christopher Sulistiyo (4850025)

Supervised by:

Jakob Loer

Version Management

Version Number	Date	Remarks
1.0	10/02/2023	Initial research topic idea
2.0	17/02/2023	Initial research sub-topics and proposal format
3.0	24/02/2023	Initial introduction
4.0		

Acknowledgement

The researcher would like to express a sincere gratitude Jakob Loer for his invaluable guidance and extensive resources throughout this research that has made a great deal of contribution and preparation to this paper in the form of remarks and suggestions. The tribute also goes out to the colleagues at the institute of NHL Stenden for their helpful feedback and encouragement.

Finally, the researcher would like to thank God, the families, and the loved ones for their unwavering support and understanding during this research. Without their support, this work would not have been possible.

Abstract

Buffer overflow vulnerability exists when a program attempts to put more data in its buffers than it can hold or when a program attempts to put data in a memory area past a buffer. A buffer itself is a sequential section of memory allocated to contain anything from a character string to an array of integers. Writing outside the bounds of a block of allocated memory can lead to potential issues such as data corruption, program cash, or most dangerous of all an execution of malicious code from hackers. If an attacker manages to make the program execute their malicious code, it will result in complete control (remote shell execution) of the victim's program, and therefore would lead to ransomware attack, data theft, and spyware, viruses, and worm injection. (Buffer overflow is one of the oldest, best-known form of software security vulnerability, existing in programming language as old as C. The problem unfortunately still exists in software applications in both legacy and newly developed applications. Main cause of this reason is due to the wide variety of ways buffer overflows can occur, with most professional software developers know about the vulnerability but are still implementing poorly developed and error-prone techniques often used to prevent them because of the lack of prevention knowledge. There are 5 different types of buffer overflows: heap-based, stack-based, integer, Unicode, and format string overflow attacks.

In a classic buffer flow exploit, also called a stack-based overflow, an attacker sends data to a program in which it will store in an undersized stack buffer. The result is that the information on the call stack is overwritten, including the function's return pointer. The data will then set the value of the return pointer so that when the function returns, it transfers control to malicious code contained in the attacker's data. At the code level, buffer overflow vulnerabilities are usually the fault of the programmers themselves. Some of the programming languages that are susceptible for buffer overflows are C, C++, Fortran, and Assembly. In C and C++, many memory manipulation functions do not perform bounds checking and can easily overwrite the allocated bounds of the buffers they operate upon. Prevention techniques that can used to mitigate the attack are the following: stack canaries, ASLR, always validate user's input, and use memory safe programming languages such as Python, Java, C#, and JavaScript.

Table of Contents

Chapter 1: Introduction

Software development is hard and a broad topic to cover. It is a complex and challenging process, with many issues that the junior developers will likely encounter. From miscommunication and lack of proper coordination, time management, program's scalability and complexity, integration problem, all the way to further maintenance and upgradation issue, these are some of the that will likely be encountered while developing a software application in a collective process that behaves in accordance with SDLC. Software development itself is a constantly growing field, filled with new technologies, ideas, platforms, and programming languages emerging on a regular basis. While these advancements bring many benefits, they also might create new challenges and complexities for developers. But one of the biggest challenge remains in software development regardless about the advancement of technologies is about the application's security. Even the most skilled developers can sometimes struggle to make their software secure and free from vulnerabilities. Looking at some of the highlights of the infamous cyber incidents, it is nearly impossible to build a software that is completely immune to attacks, as there are countless potential vulnerabilities and attack vectors to consider, by the virtue of the ever growing and constantly evolving field of technology. From input validation issues, SQL Injection attacks, DDoS attacks, to buffer overflows, software vulnerabilities can lead to serious security breaches and data loss, causing significant harm to both individuals and organizations that are using the exploited application. Furthermore, identifying and addressing vulnerabilities can be time-consuming, as it is also a tedious task for developers to keep looking at their code over and over until they finally find what part of their code that led to the cyber-attack. It is also can be costly, requiring extensive testing and debugging efforts. As such, it is crucial for software developers to be aware of the existing solved vulnerabilities in the past to anticipate further potential risks and to take proactive measures to minimize vulnerabilities and ensure the security of their applications. For this specific research report, it will investigate one of the oldest vulnerabilities ever existed in the programming world, buffer overflow vulnerability.

Chapter 2: Methodology

Instruments

During this research, only qualitative data was collected, as well as all the research questions are answered through the needed data by the means of desk research (*see ICT Library research method*). However, it is possible that each of the research question's answer may be expanded upon if needed or requested by the aforementioned research supervisor, as the content of the research is limited, and the research answers are purposely condensed.

Population and Samples

The population of this research will be limited to the people with a basic understanding of coding, which range from the university/ college students to an adult senior software developer. That means that a little bit of technical knowledge about programming and operating system is necessary to fully understand the context of this research.

The research samples were selected based on their purpose and convenience. Multiple articles online and books through the form of electric or physical have been used regarding the research as well as sampling.

Limitations

This research will only talk about buffer overflow and subjects related to it in 32-bit application in Linux operating system, as these topics are complex and widely researched. While this study is limited to Linux, the principles and concepts discussed can be applied to other operating systems as well. However, this research will not explore and go into details for these broader applications and is limited for the specific context of Linux. The same concept also applies to the 64-bit applications, which will not be discussed in this research, as their anatomy in a computer's memory and virtual address pace look essentially the same with the 32-bit applications and the components are also in the same space. The only difference is that the 64-bit applications have more memory and there might be more spacing between components, but the same concept will apply and it will be easier for the new readers to understand the concept starting from simpler subjects.

Data Analysis

The research data regarding this paper will be qualitative and primarily acquired from many sources. The validity of the information is medium to high, due to many than one sources explaining the subject through numerous ways, but the main certainty comes from official documentation provided by OWASP.

Research Subject and Its Following Inquiries

The research topic is described as follows: Buffer Overflow Vulnerability, Understanding the Vulnerability of the Decade.

Main question: What are the countermeasures and mitigation techniques for countering different modern buffer overflow vulnerability attacks in programming languages like C?

Firstly, a established knowledge of what a buffer overflow is will be needed from the readers. Then, the readers will be invited to learn what can this attack do to computer system when being let happen. Additionally, different types of this attack will be discussed, along with different tactics to prevent or combat this specific attack.

From here, the main question will be divided into 4 sub-questions. In doing so, it will then create a way of understanding the topic as well as creating knowledge about the subject. All those sub-questions are included in the following:

- What is a buffer overflow vulnerability and how does it occur in computer systems?
- What Are the Consequences of a Successful Buffer Overflow Attack on a Computer System and Its Users? Who is Susceptible to Buffer Overflow Attacks?
- What Are the Different Types of Buffer Overflow Attack in Computer Systems?
- How Can Software Developers Prevent Buffer Overflow Vulnerabilities in Their Code Through Design and Testing Practices Considering the Current State-Of-Art Technologies?

Chapter 3: Research Result

Research Question #1: What Is a Buffer Overflow Vulnerability and How Does It Occur in Computer System?

Before diving in and start discussing about the vulnerability itself, a basic understanding of program segments/ sections and virtual address spaces are expected of the reader to understand the gist of memory corruption which is the main association of a buffer overflow vulnerability. Those topics will be discussed and included in this chapter.

Compiled programs are typically organized into at least 3 sections (or sometimes called segments), and they are logical blocks within a compiled program that help developers to organize the contents of a program. When writing a code, it is common to have a number of instructions and variables; with some of the variables might be initialized to a certain values, some of them will be uninitialized, some of the variables will be global, and some of them will be local. And when compiling a program with the knowledge that it eventually be loaded into a virtual address space, it would be logical to take the various components inside the program and split them up logically inside of the compiled program.

- .text – this holds the program's instructions. With the instructions being basic lines of code. This is where developers put a calling to a function, an instruction of an addition of two variables, etc.
- .data – this holds the initialized global variables. For example, at the top of the program the developer specified int x = 5, a memory region would be defined that has the value 5 associated to it across 4 bytes (because it is an integer).
- .bss – this holds uninitialized global variables. For example, a developer just initialized int x; instead of int x =5. Those uninitialized variables will be grouped together and put in this file.

There are more sections to it but these 3 will suffice to understand the main topic of this paper. When writing a code, developer will have several instructions and variables, with some of the variables being initialized or uninitialized to certain values.

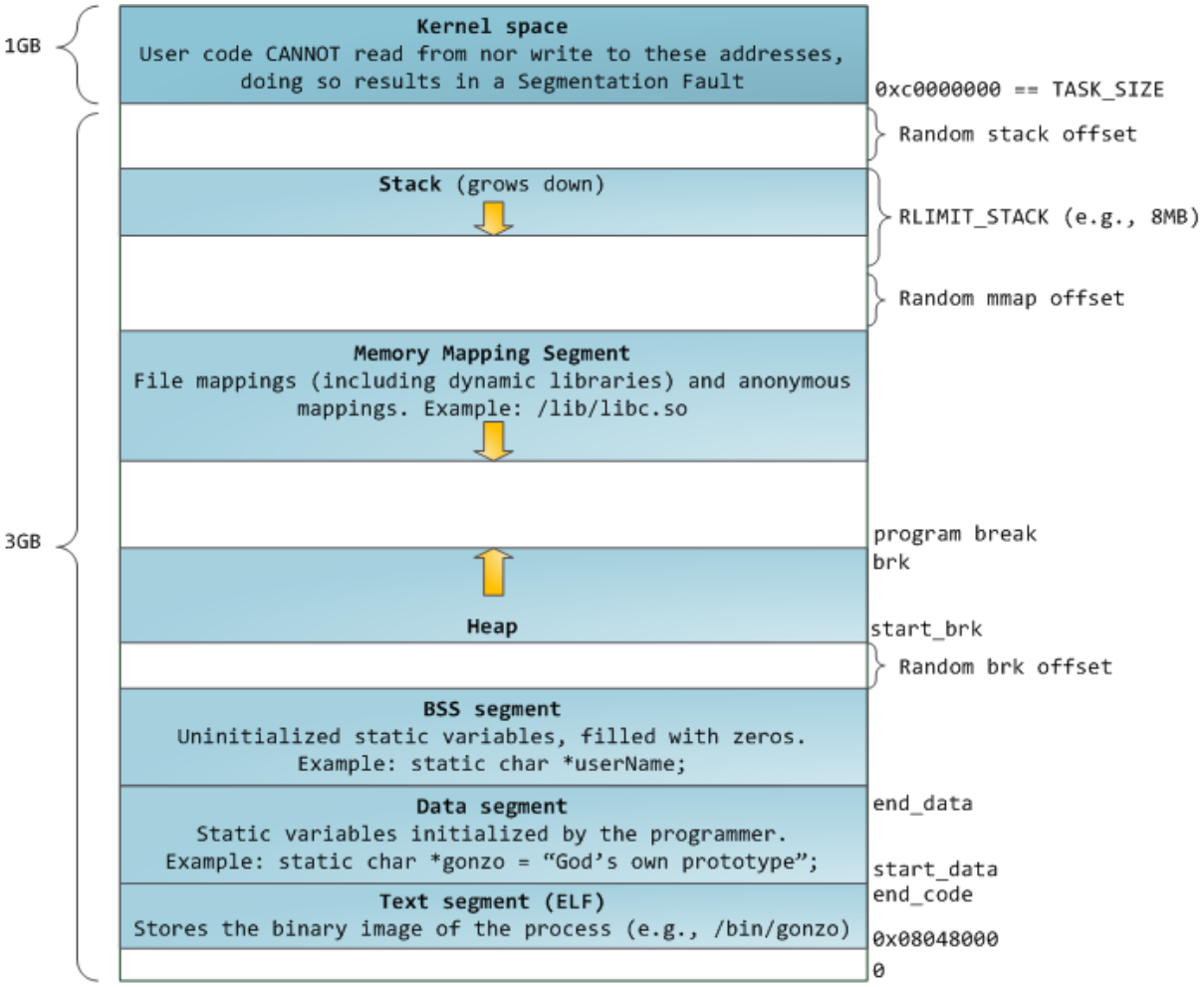

Figure 1: Virtual address space for a 32-bit application in Linux OS (Duarte, 2009)

When a program gets loaded and turns into a process, in which eventually it will create child-processes according to the program's functionality, a virtual address space will be created for that child process. Some explanations about Figure 1 diagram will be discussed for a lengthy part of this chapter. The diagram consists of different part of virtual address space, in which from now on they will be called rectangles. The first rectangle shows an area of memory that is allocated for the kernel's use. The memory address of 0xFFFFFFFF (which would be the largest number in 32-bit system could represent) down to 0xc0000000 is an area of memory that is allocated for the kernel's use. This is where the kernel's code lives, including all the drivers and schedulers it needs. Basically, all kinds of portions of the kernel lives in there. The user mode programs cannot read from or write to this address or otherwise a fault within a program will be created because this part of memory (kernel) is not supposed to be modified or accessed directly. This part of the memory can only be accessed through the interface that is often called system calls (*see Operating System Concepts*).

The next part of the diagram (*Figure 1*) is where the things get progressed from a high to low memory addresses in a virtual address space. The second part is called process's stack. It is a type of data structure that the process uses for managing functions and variables. The bottom of the stack is represented as the peak of the second rectangle where the diagram is typically drawn, and as the elements of the stack get pushed the stack will grow downwards towards the lower memory addresses.

After the stack, represented by the third rectangle is a region of memory where dynamically loaded libraries are put. If a program imports standard library, standard I/O, $string.h$ (typically in C/ C++ library), or any kind of a third-party library that is required for the execution of a program, meaning the code that the developers did not write, lives in this middle region.

The next thing in the address space is the heap, which represented in the fourth rectangle. Heap is where developers draw a dynamically allocated memory from. For example, if $malloc()$ function is called in a C/ C++ program, then some numbers of bytes are allocated to a dynamically generated string, those bytes/ memory addresses that have been given all live in the heap. The heap growth is the exact opposite of the stack, where the low memory addresses is at the bottom of the heap and it grows upwards.

After the heap is where the components that would have been loaded in from a compiled executable/ a program that is running. All 3 of the instructions that have been discussed above live in here.

Buffer (Thakur, n.d.) is a space of physical storage memory used to hold temporary data while traveling from one location to another. It is the area of memory used temporarily keep data. This might occur, for instance, when a program tries to store input from a user or a network in a buffer. These buffers are typically found in RAM. Buffers are frequently used by computers to increase performance; most contemporary hard drives make use of the advantages of buffering to retrieve data quickly, and a number of online application services also make use of buffers. For instance, buffers are frequently employed to avoid interference when streaming videos online. When a video is streaming, the video player downloads at once before starting to play it. Therefore, the performance of the video streaming is unaffected by a slight decrease in connection speed or a brief service outage.

A simple example of a buffer is a fixed size array, for example an array of 10 integers. It is an array that is fixed in size so that it can be populated with elements.

However, because different applications use the buffer's mechanism, buffers have a limited capacity and can only store a specific quantity of data for a certain amount of time. As a result, a situation where more data is pushed into a buffer arises; this referred to as a buffer overflow. Buffer overflow also known as buffer overrun vulnerability (Smith, n.d.) happens when a program tries to write more data to a buffer than the buffer is intended to contain, overwriting nearby memory regions. In other words, too much data is sent to a repository that does not have adequate room, and this data is gradually replaced with data from nearby repositories.

Figure 2: A graphical representation of an overflowed buffer (Imperva, n.d.)

Below are some of the functions in C and C++ that are commonly associated with not doing the appropriate buffer lengths checking:

- `gets() -\> fgets()` - read characters
- `strcpy() -\> strncpy()` - copy content of the buffer
- `strcat() -\> strncat()` - buffer concatenation
- `sprintf() -\> snprintf()` - fill buffer with data of different types
- `(f)scanf()` - read from STDIN
- `getwd()` - return working directory
- `realpath()` - return absolute (full) path

Figure 3: Dangerous C and C++ functions that are associated with buffer overflows (Smith, n.d.)

Buffer overflow in web applications

Web applications can also be affected by the corruption of the execution stack by buffer overflows. By sending a carefully crafted input to a web application, an attacker may cause the web application to execute arbitrary code – effectively taking over the machine. The buffer overflow vulnerabilities can be present in both web server and application server products that serve the static and dynamic aspects of a website, or a web application itself. Buffer overflows found in widely used server products are likely to become widely known and can pose a significant risk to users of these products. When web application use libraries, such as a graphics library to generate images, they open themselves to potential buffer overflow attacks.

Buffer overflows in custom web applications are less likely to be detected because there are far few hackers trying to find and exploit such flaws in a specific application. If discovered in a custom application, the ability to exploit the vulnerability (other than to crash the program) is significantly reduced by the fact that the source code and detailed error messages for the application are normally not available to the hacker.

Research Question #2: What Are the Consequences of a Successful Buffer Overflow Attack on a Computer System and Its Users? Who is Susceptible to Buffer Overflow Attacks?

Access control loss (instruction processing)

A successful attempt to a buffer overflow will result in the attacker gaining access to a victim's computer. By carefully overflowing the buffer into a certain limit, the attacker would then insert a malicious code that will be read and executed by the targeted program, resulting in full remote control and backdoor installation. This would then lead to a further bigger problem, for instance data theft, data loss, etc. An overflow attack will often involve the use of arbitrary code, which is often outside of the scope of program's security policies.

System crashes

Typically, while not being handled by a professional attacker, a buffer overflow attack itself will often result in system crashing as it is trying to read and execute the next line of code which will be just a region of overflowed data being put in the input stream, that will result into system crashing. Another common thing that usually happens during buffer overflow attack is to make the program jump to another point in code, in which sometimes the logic would not be applied to the program, because the unprofessional attacker does not know the structure of the code of the target program written by the developer and therefore cause a system crash. It may also result in lack of availability and programs being put into an infinite loop.

Further security issues

An attacker may utilize arbitrary code execution from a buffer overflow attack to exploit additional flaws and undermine more security measures.

Vulnerabilities of the buffer are overwhelmed in programming languages like C, who trade their security for efficiency, and do not control memory access. In environments written in interpreted languages like Python, PHP, Perl, Java, or JavaScript which are often used to build web applications, are more immune to the attack except for overflows in their interpreter.

If the input is larger than the buffer, the extra data can overwrite other parts of the program's memory, potentially causing the program to crash or behave unpredictably. Even worse, an attacker can potentially exploit this vulnerability by carefully crafting input data that overwrites parts of the program's memory with malicious code. If the program then jumps to that code and executes it, the attacker can gain control of the program and potentially even the entire system.

Nearly all applications, web servers, and web application environments are vulnerable to buffer overflows. One of the main causes of the buffer overflow attacks is that the program written by the developers does not allocate buffers of the proper size and does not check for overflow issues. These issues are particularly severe to programming languages like C and C++ due to the lack of built-in buffer overflow protection.

But even for developers who code in the high-level programming language should take extra care around them. Their programs frequently use C-coded operating systems or runtime environments, which still makes them susceptible to such attacks.

Research Question #3: What Are the Different Types of Buffer Overflow Attack in Computer Systems?

There are many available ways and techniques to exploit buffer overflow vulnerabilities, and they are also vary based on the OS and programming language. However, the objective is always to interfere with or regulate program execution by messing with a computer's memory. There are 5 types of buffer overflows that have been known so far: stack-based, heap-based buffer overflows, integer overflow, Unicode overflow, and format string overflow attacks; with the first two being the most common types of buffer overflow attacks. They have been categorized based on the location of the buffer in the process memory (Cobb, July 2021).

Heap-based buffer overflow attack

A memory structure for controlling dynamic memory is called a heap. When the needed memory is too large to fit on the stack or is meant to be used between function calls, programmers frequently use the heap to allocate memory whose size is unknown at compile time. Attacks using heaps flood memory space reserved for an application or process. This type of attack is harder to carry out and happens rarer than stack attacks as it is more difficult to exploit.

The heap is one of the components of the virtual address space allocated by the kernel when a process starts. The heap grows towards the stack, so the "bottom" of the heap is at lower addresses, and it grows towards higher addresses (*see Figure 1*). Heaps are typically constructed using linked lists of allocated and unallocated blocks of memory. An attacker's goal in a heap overflow is to fill one allocated "chunk" of memory past its capacity with the hope that the overflowed data will leak into an adjacent variable's chunk of memory. This can lead to code execution or just plain variable corruption.

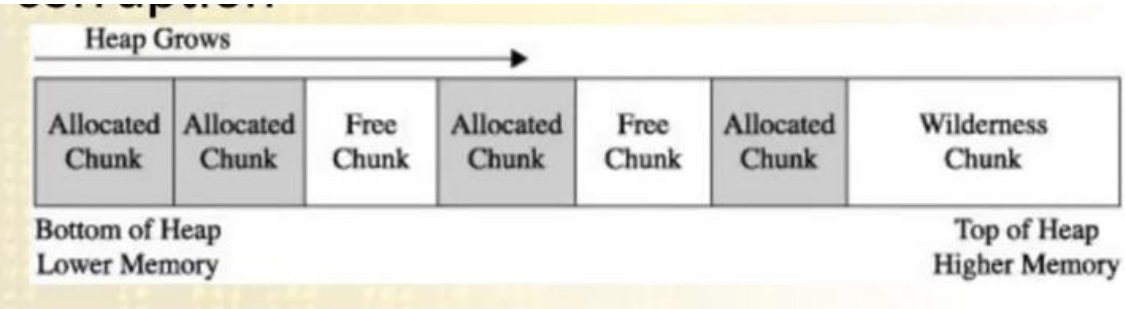

Figure 4: An anatomy of how heap grows in memory

Stack-based buffer overflow attack

Data is stored on the stack using a LIFO (last-in, first-out) structure. It is a continuous space in memory that is used to organize data related to function calls, such as function local variables and management information, such as frame and instruction pointers. Typically, until the targeted program requires user input, the stack is empty. The user's input is then placed on top of the return memory address that the program had previously written to the stack. The user's input is transmitted to the return address designated by the program after the stack is processed.

However, a stack has a limited capacity. The stack must be given a certain amount of space by the programmer who developed the code. The stack will overflow if the user's input is longer than the space allocated for it in the stack and the program does not verify that the input will fit. This is not a major issue by itself, but when paired with malicious input, it poses a serious security risk.

Integer overflow attack

For integers, the majority of programming languages specify their maximum sizes. When certain limit is exceeded, the result can cause an error or deliver incorrect result within the bounds of an integer.

Word Size	Signed Values (+ or -)	Unsigned Values (+ only)
8 bits	-128 to +127	0 to 255
16 bits	-32,768 to +32,767	0 to 65,535
32 bits	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,296

Figure 5: A common integral data types (MTSU, n.d.)

Unicode overflow attack

This type of overflow is a security exploit that targets applications or systems that handle user input using Unicode encoding. Each character used in different writing systems is given a unique number by the character encoding standard known as Unicode.

In Unicode overflow attack, an attacker exploits a flaw in a application or system that allows the input of too many Unicode characters, leading to a buffer overflow. This overflow may cause the system to crash or cause an unexpected behaviour. In some cases, it may also be leveraged to run malicious code or obtain unauthorized access to the system.

Format string attack

A format string attack is when a program or system accepts user input as a format string function that is not designed to handle the input. This security vulnerability can cause a buffer overflow, which might result in unexpected behaviour or even arbitrary code execution on the system.

An attacker exploits a vulnerability in an application or program that accepts user input and uses it as a format string parameter in a function such as $printf()$ or $fprintf()$ to carry out a format string overflow attack. The function will attempt to read additional data from the stack if the input contains format string characters such as %s, %d, or %x because they are interpreted as format directives by these functions.

The function may read too much data from the stack if the input is not properly sanitized, which could potentially change the way an application flows and lead to access and manipulation of other memory spaces.

Format function	Description
fprint	Writes the printf to a file
printf	Output a formatted string
sprintf	Prints into a string
snprintf	Prints into a string checking the length
vfprintf	Prints the a va_arg structure to a file
vprintf	Prints the va_arg structure to stdout
vsprintf	Prints the va_arg to a string
vsnprintf	Prints the va_arg to a string checking the length

Figure 6: Format functions in C that if not treated, can expose the application to Format String attack (Meir, n.d)

Parameters	Output	Passed as
%%	% character (literal)	Reference
%p	External representation of a pointer to void	Reference
%d	Decimal	Value
%c	Character	
%u	Unsigned decimal	Value
%x	Hexadecimal	Value
%s	String	Reference
%n	Writes the number of characters into a pointer	Reference

Figure 7: Common parameters used in Format String Overflow attack (Meir, n.d.)

Research Question #4: How Can Software Developers Prevent Buffer Overflow Vulnerabilities in Their Code Through Design and Testing Practices Considering the Current State-Of-Art Technologies?

Validate data (input validation)

The most effective technique to prevent buffer overflow attacks is to validate all input data before it is processed. This involves evaluating the amount and type of input data to ensure that it is within expected range and does not contain dangerous code. Another way of countering buffer overflows using this method is to avoid using the get user input functions specified in *Figure 3* while writing a program in C and C++.

Address Space Randomization (ASLR)

It is a specific computer security technique involved in preventing exploitation of memory corruption vulnerabilities such as buffer overflows. In a buffer overflow attack, attacker feeds a function as much junk data as it can handle, followed by a malicious payload. The payload will then overwrite data the program intends to access. Instructions to jump to another point in code are a common payload.

For a successful buffer overflow attack by a professional attacker, it requires the attacker to know where each part of the program is in memory. Figuring this out can be a difficult process requiring trial and error. After determining that, they must make a payload and find a suitable place to inject it. If the attacker does not know where their target code is located, it can be difficult or impossible to exploit it.

ASLR works alongside virtual memory management and randomly arranges address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap, and libraries. Every time the program is run, the components are moved to a different address in virtual memory. Attackers can no longer learn where their target is through trial and error, because the address will be different every time. Generally, applications need to be compiled with ASLR support, but this is becoming the default requirement, including compilers like CLANG and GNU Compiler Collection (GCC) for C and C++.

However, a paper has been presented by the researchers from University of California and SUNY Binghamton (*Evtyushkin, 2016*) shows that it is still possible to determine locations of known branch instructions in a running program using an attack on the BTB (Branch Target Buffer). BTB is part of the processor that speeds up if statements by predicting the outcome. With, newer, fine-grain ASLR techniques and increasing the amount of entropy (randomness) can make the Jump Over (buffer overflow) attack infeasible and would require more effort from the attacker.

Stack Canary

Named after the bird canary that is used by humans in coal mine. The idea is to place this warm-blooded animal in a mine to detect carbon monoxide level. The bird being more sensitive to this toxic gas would become sick before the miners. The bird is kept within a cage, and after several hours the miners would check whether the bird is still alive or not. If it is, it would indicate that the specific mine is safe for miners to work and put on some protective respirators or abandon the mine. In some cases, the canaries were kept in a cage with a dedicated oxygen tank so the birds could survive after their illness provided a warning.

The same concept applies to this buffer overflow counter tactic. The original developers who code the program would put a small integer at the beginning of their code lines. The value of the integer can be chosen specifically by the developer or is randomly chosen by the program at start, in memory just before the stack return pointer. Most buffer overflow will overwrite memory from lower to higher memory addresses, so in order for a successful control of the target program/ process by the attackers

the attack needs to overwrite the return pointer and the canary value must also be overwritten. The value would therefore be checked to make sure it has not changed before a routine uses the return pointer on the stack. This technique can greatly increase the difficulty of exploiting a stack buffer overflow because it forces the attacker to gain control of the instruction pointer by some non-traditional means such as corrupting other important variables on the stack. The attacker could also guess the value of the canary, although this would be made impossible especially if the value is randomly generated by the program, but this does not mean that using this concept will 100% make the program invulnerable to the buffer overflow attack. There are 3 types of canaries that is often in used: terminator, random, and random XOR canaries.

Nonexecutable stack

Another approach of countering buffer overflow is to enforce a memory policy on the stack memory region that would disallow execution from the stack (Write XOR Execute, or W^X). This would mean that an attacker must either find a way to disable the execution from memory or find a way to put their shellcode payload in a non-protected region or memory to execute their malicious shellcode from the stack.

Runtime Bounds Checking

Furthermore, an additional way of countering buffer overflow vulnerabilities is to utilize memory-safe programming languages that have built-in memory safety feature, for example Python and Java.

Chapter 4: Summary

A buffer overflow is a type of software application's security vulnerability that occurs when a program or process attempts to write more data to a fixed-length block of memory, or buffer, than the buffer is allocated to hold. Buffer itself is a finite contiguous allocated block of memory contain a defined amount of data; any extra data will overwrite data value in memory addresses adjacent to the destination buffer. The C language, unlike most moder languages, has no default bounds-checking behaviour on its various buffer construct. This vulnerability can easily be avoided if the program includes sufficient bounds checking to flag or discard data when too much is sent to a memory buffer.

Buffer overflow typically occurs in a code that:

- Relies on external data to control its behaviour.
- Depends upon properties of the data that are enforced outside of the immediate scope of the code.
- Is so complex that a programmer cannot accurately predict its behaviour.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating system: Windows, MacOS, and Linux (all).

Countermeasures

- Runtime bounds checking.
- Stack canary.
- Address space randomization (ASLR).
- Nonexecutable stack.
- Code in programming languages that implement runtime bounds checking.

Most common ways to prevent buffer overflows is to use OS runtime protection, such as ASLR to randomly arrange the address spare positions of key data areas of a process. This includes the base of the executable and positions of the heap, stack, and libraries. This approach will make it very difficult for an attacker to reliably jump to a particular function in memory. Another way of preventing this vulnerability is to make nonexecutable stack or marks the areas of memory as either executable or nonexecutable with Data Execution Prevention.

Another way to counter buffer overflow for a non-developer point of view is to always keep the devices patched as vendors usually issue software patches and updates to fix all the known and discovered security breaches. A user should keep up to date with the latest bug reports of their web, application server products, and other products that they use in their internet infrastructure.

Reference List

Anonymous. (April 8th, 2021). *Address Space Layout Randomization*. IBM. Retrieved on March 1st, 2023, from https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization

Anonymous. (February 14th, 2023). *Defining Buffer Overflow Attacks & How to Defend Against Them*. Retrieved on March 17th, 2023, from https://www.okta.com/identity-101/buffer-overflow-attacks/

Anonymous. (n.d.). *Buffer Overflow*. ENISA. Retrieved on March 11th, 2023, from https://www.enisa.europa.eu/topics/incident-response/glossary/buffer-overflow

Anonymous. (n.d.). *Buffer Overflow*. Fortinet. Retrieved on March 13, 2023, from https://www.fortinet.com/resources/cyberglossary/buffer-overflow

Anonymous. (n.d.). *Buffer Overflow*. OWASP. Retrieved on March 9th, 2023, from https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

Anonymous. (n.d.), *Buffer Overflow*. Wikipedia. Retrieved on March 5th, 2023, from https://en.wikipedia.org/wiki/Buffer_overflow

Anonymous. (n.d.). *Buffer Overflow Attack*. Imperva. Retrieved on March 10th, 2023, from https://www.imperva.com/learn/application-security/buffer-overflow/

Anonymous. (n.d.). *Buffer Overflow Attack*. NordVPN. Retrieved on March 17th, 2023, from https://nordvpn.com/cybersecurity/glossary/buffer-overflow-attack/

Anonymous. (n.d.). *Buffer Overflow Attacks*. Hacksplaining. Retrieved on March 14th, 2023, from https://www.hacksplaining.com/prevention/buffer-overflows

Anonymous. (n.d.). *Buffer Overflow Protection*. Wikipedia. Retrieved on March 21st, 2023, from https://en.wikipedia.org/wiki/Buffer_overflow_protection

Anonymous. (n.d.). *Data Types in C++*. MTSU. Retrieved on March 24th, 2023, from https://www.cs.mtsu.edu/~xyang/2170/datatypes.html

Anonymous. (n.d.). *What is a Buffer Overflow?* CheckPoint. Retrieved on March 4th, 2023, from https://www.checkpoint.com/cyber-hub/cyber-security/what-is-cyber-attack/what-is-a-buffer-overflow/

Anonymous. (n.d.*). What is a Buffer Overflow Attack?* ⚔ *Types, Examples. Part 1*. Wallarm. Retrieved on March 18th, 2023, from https://www.wallarm.com/what/buffer-overflow-attack-definition-types-use-by-hackers-part-1

Anonymous. (n.d.). *What is Buffer Overflow?* ContrastSecurity. Retrieved on March 16th, 2023, from https://www.contrastsecurity.com/glossary/buffer-overflow

Anonymous. (n.d.). *What is Buffer Overflow?* Cloudflare. Retrieved on March 15th, 2023, from https://www.cloudflare.com/en-gb/learning/security/threats/buffer-overflow/

Anonymous. (n.d.). *What Is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks*. Veracode. Retrieved on March 7th, 2023, from https://www.veracode.com/security/buffer-overflow

Bhargav, Nikhil. (January 25th, 2023). *What's a Buffer?* Baeldung. Retrieved on March 2nd, 2023, from https://www.baeldung.com/cs/buffer

Chapin, Steve J. & Lhee, Kyung-Suk. (April 25th, 2003). Department of Electrical Engineering & Computer Science in Syracause University: Software – Practice and Experience. *Buffer Overflow and Format String Overflow Vulnerabilities*, 33(5), 423-460. https://surface.syr.edu/cgi/viewcontent.cgi?httpsredir=1&article=1095&context=eecs

Cobb, Michael. (July 2021). *Buffer Overflow*. TechTarget. Retrieved on March 13th, 2023, from https://www.techtarget.com/searchsecurity/definition/buffer-overflow

Constantin, Lucian. (January 22nd, 2020). *What is A Buffer Overflow? And How Hackers Exploit These Vulnerabilities*. CSO. Retrieved on March 18th, 2023 from https://www.csoonline.com/article/3513477/what-is-a-buffer-overflow-and-how-hackers-exploit-these-vulnerabilities.html

Du, Wenliang. (October 12th, 2017). *Computer Security: A Hands-on Approach*. (1st edition). CreateSpace Independent Publishing Platform

Duarte, Gustavo. (January 27th, 2009). *Anatomy of a Program in Memory*. ManyButFinite. Retrieved on March 21, 2023, from https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

Evtyushkin, Dmitry et all. (October 15, 2016). *Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR*. University of California, Riverside & SUNY Binghamton. https://www.cs.ucr.edu/~nael/pubs/micro16.pdf

Holt, Rene. (December 6th, 2021). *What are Buffer Overflow Attacks and How are They Thwarted?* WeLiveSecurity. Retrieved on March 19th, 2023, from https://www.welivesecurity.com/2021/12/06/what-are-buffer-overflow-attacks-how-are-they-thwarted/

Kaczanowski, Megan. (April 5th, 2021). *What is a Buffer Overflow Attack – and How to Stop it*. freeCodeCamp. Retrieved on March 20th, 2023, from https://www.freecodecamp.org/news/buffer-overflow-attacks/

Meir. (n.d.). *Format String Attack*. OWASP. Retrieved on March 23rd, 2023, from https://owasp.org/www-community/attacks/Format_string_attack

Nidecki, Tomasz Andrzej. (June 17th, 2019). *What Is a Buffer Overflow*. Acunetix. Retrieved on March 8, 2023 from https://www.acunetix.com/blog/web-security-zone/what-is-buffer-overflow/

Silberschatz, Abraham et all. (June 26th, 2001). *Operating System Concepts*. (6th edition). Wiley.

Smith, Andrew et all. (n.d.). *Buffer Overflow Attack*. OWASP. Retrieved on March 6th, 2023, from https://owasp.org/www-community/attacks/Buffer_overflow_attack

Srinivas. (August 27th, 2020). *How to Mitigate Buffer Overflow Vulnerabilities*. Infosec. Retrieved on March 15, 2023 from https://resources.infosecinstitute.com/topic/how-to-mitigate-buffer-overflow-vulnerabilities/

Stewart, Dennis. (October 26th, 2016). *What Is ASLR, and How Does It Keep Your Computer Secure?* How-To-Geek. Retrieved on March 21st, 2023, from https://www.howtogeek.com/278056/what-is-aslr-and-how-does-it-keep-your-computer-secure/

Thakur, Dinesh. (n.d.). *What is Buffer – Definition*. ComputerNotes. Retrieved on March 21st, 2023, from https://ecomputernotes.com/fundamental/input-output-and-memory/buffer

Watters, Brendan. (February 19th, 2019). *Stack-Based Buffer Overflow Attacks: Explained and Examples*. Rapid7. Retrieved on March 4th, 2023, from https://www.rapid7.com/blog/post/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/

Welekwe, Amakiri. (August 24th, 2020). *Buffer Overflow Vulnerabilities and Attacks Explained*. Comparitech. Retrieved on March 12th, 2023 from https://www.comparitech.com/blog/information-security/buffer-overflow-attacks-vulnerabilities/

Appendix