Table of Content

Introduction on Buffer Overflow

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations.

A buffer overflow occurs when a program writes more data to a buffer (temporary storage area) than it can hold. This can lead to overwriting adjacent memory, which may contain important data like function pointers or flags.

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

Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes.

Attackers exploit buffer overflows to overwrite memory in a way that changes a program's behavior, often gaining unauthorized access or executing malicious code.

Understanding Buffer Overflow Vulnerabilities

How Does It Happen?

Buffer overflow often occurs when a program doesn't check the size of the input it's receiving, allowing excessive data to overflow into other memory areas.

Common Vulnerable Functions in C

Functions like gets(), strcpy(), and scanf() don't check the size of input. Using them without caution can lead to buffer overflow vulnerabilities.

Key Concepts of Buffer Overflow

- This error occurs when there is more data in a buffer than it can handle, causing data to overflow into adjacent storage.
- This vulnerability can cause a system crash or, worse, create an entry point for a cyberattack.
- C and C++ are more susceptible to buffer overflow.
- Secure development practices should include regular testing to detect and fix buffer overflows. These practices include automatic protection at the language level and bounds-checking at run-time.
- Veracode's binary SAST technology identifies code vulnerabilities, such as buffer overflow, in all code — including open source and third-party components —so that developers can quickly address them before they are exploited.

Types of Buffer Overflow

1. Stack-based buffer overflows

Stack-based buffer overflow exploits are likely the shiniest and most common form of exploit for remotely taking over the code execution of a process. These exploits were extremely common 20 years ago, but since then, a huge amount of effort has gone into mitigating stack-based overflow attacks by operating system developers, application developers, and hardware manufacturers, with changes even being made to the standard libraries developers use.

Stack-based overflows occur when a program writes more data to a local variable (like an array or buffer in a function) than it can hold. This extra data can overwrite other parts of the stack, like function pointers, return addresses, or other local variables.

Example: Suppose a function has a local buffer for a password. If a user inputs more characters than this buffer can hold, the overflow can overwrite the return address on the stack, allowing an attacker to control where the program goes next. By overwriting the return address, attackers can redirect program execution to a location of their choosing (like injected malicious code).

Here's a simple example of a stack-based buffer overflow in C. This program has a vulnerability because it doesn't check the size of user input, allowing the input to overflow the buffer.

Source Code:

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

void vulnerableFunction() {
   char buffer[10];  // A small buffer, can only hold 10 characters

   printf("Enter some text: ");
   gets(buffer);     // Unsafe function, does not check input size

   printf("You entered: %s\n", buffer);
}

int main() {
   vulnerableFunction();
   return 0;
}
```

When the user enters more than 10 characters, buffer overflows, and the extra data spills over into nearby parts of the stack, possibly affecting other variables, function pointers, or the return address.
By controlling this overflow, an attacker could potentially alter the program's execution by overwriting the return address with a different memory address.

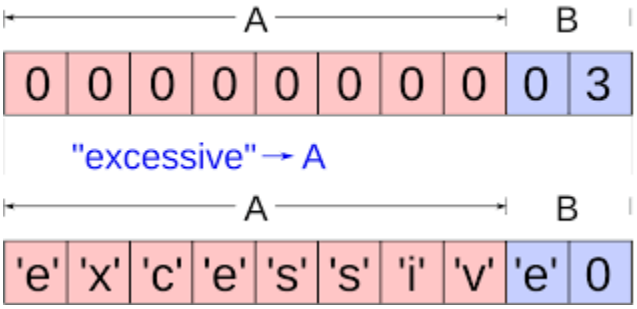
If you run this program and enter a short input (like "hello"), it works as expected.
However, if you enter a long string (e.g., "AAAAAAAAAAAAAAA"), it will overflow buffer, writing extra data into adjacent stack memory, possibly causing a crash or unpredictable behavior.

2. Heap-based attacks

This type of overflow happens in the heap, which is the part of memory used for dynamically allocated variables (like variables created with malloc() in C). malloc() (memory allocation) is a function used to dynamically allocate memory on the heap at runtime. This is useful when you don't know the

exact amount of memory your program will need in advance, or when you need more control over memory allocation.

Heap-based overflows occur when a program allocates a chunk of memory on the heap but writes more data to it than allocated, overwriting adjacent memory regions.



Example: If a program allocates 20 bytes on the heap but writes 30 bytes, the extra data might corrupt neighboring data in the heap. Heap overflows can allow attackers to manipulate metadata that manages the heap or overwrite pointers, potentially leading to arbitrary code execution.

Here's a simple example of a heap-based buffer overflow in C. Heap-based attacks exploit vulnerabilities in dynamically allocated memory, like memory created with malloc() or calloc().

Source Code

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

void heapOverflowExample() {
   char *buffer = (char *)malloc(10 * sizeof(char));  // Allocate 10 bytes on the heap

   if (buffer == NULL) {
      printf("Memory allocation failed.\n");
      return;
   }
```

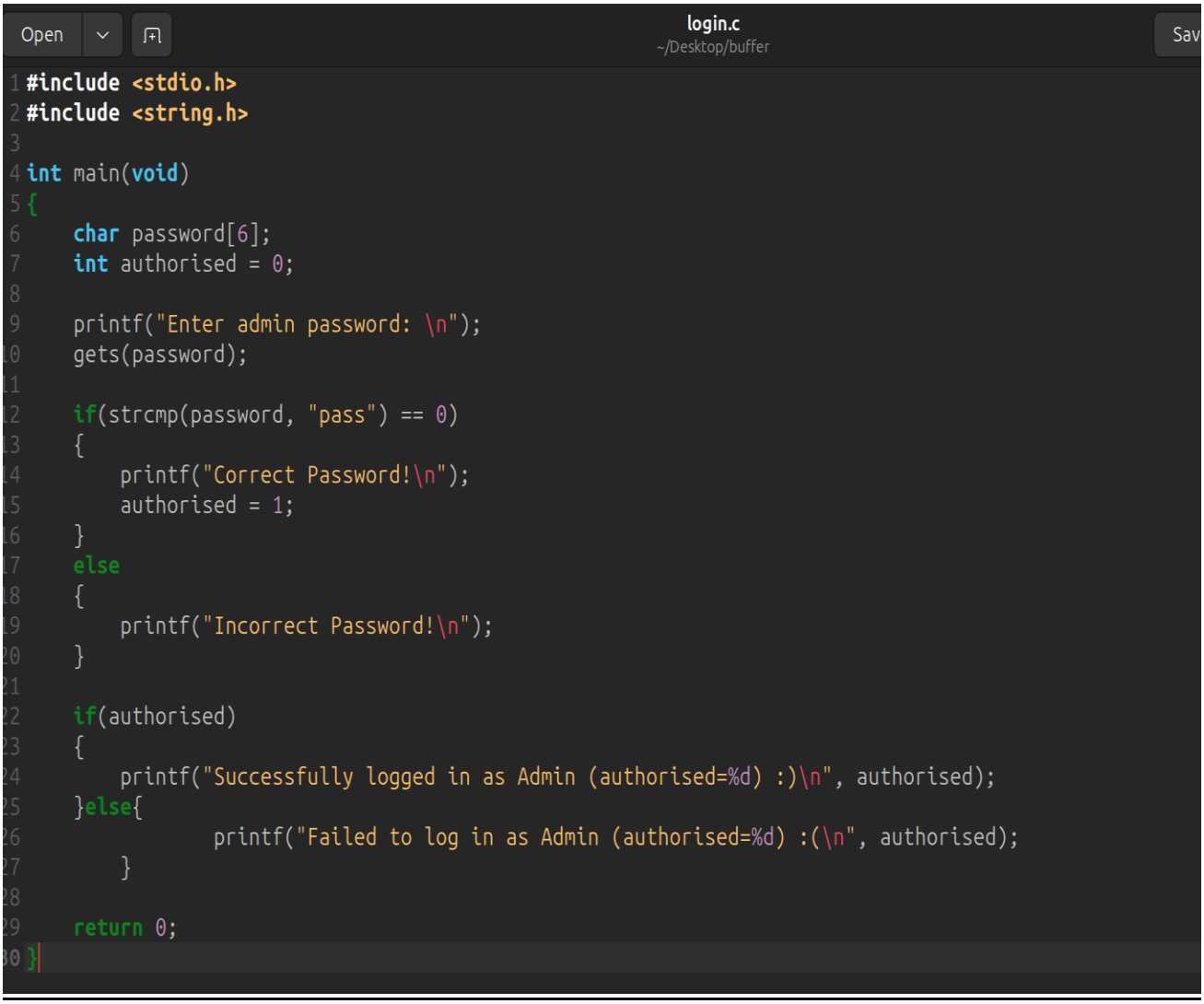
```
    // Intentionally copy a large string that overflows the allocated buffer
    strcpy(buffer, "This is a very long string that overflows the buffer!");

    printf("Buffer contains: %s\n", buffer);

    // Free allocated memory
    free(buffer);
}

int main() {
    heapOverflowExample();
    return 0;
}
```

When you copy a string that's too large into the allocated buffer, it doesn't fit. The extra data overflows into neighboring memory on the heap, which can overwrite important information or control structures.
This type of overflow can lead to heap corruption, potentially allowing an attacker to manipulate heap metadata, change pointers, or alter other critical data, leading to arbitrary code execution or crashes.

Demo

Vunerable C code:

Source Code:



This C program is a simple authentication example where a user is asked to enter an admin password. If the entered password matches a predefined value, the user is marked as "authorized." However, the code contains a vulnerability due to unsafe input handling, specifically a buffer overflow vulnerability.

The function gets() doesn't check the size of password, so a user could enter more than 5 characters, causing a **buffer overflow**. When this happens:

- The extra data overflows beyond the memory allocated to password and may overwrite the next variable in memory, which here is authorised.
- If authorised is overwritten with a non-zero value, the program might display "Successfully logged in as Admin" even though the correct password wasn't entered.

To exploit this program using GDB (GNU Debugger), we can try to overflow the password buffer and overwrite the authorised variable. This is a classic example of a **stack-based buffer overflow** where, due to unsafe handling of the buffer, we attempt to overwrite adjacent memory and bypass the authentication check.

Step-by-Step Exploitation Using GDB

1. Compile the Program without Stack Protection

Make sure to compile the program without stack protection (like stack canaries) and without Address Space Layout Randomization (ASLR), which would prevent or complicate buffer overflow exploitation

gcc -fno-stack-protector -z execstack -o login login.c -m32

Note:

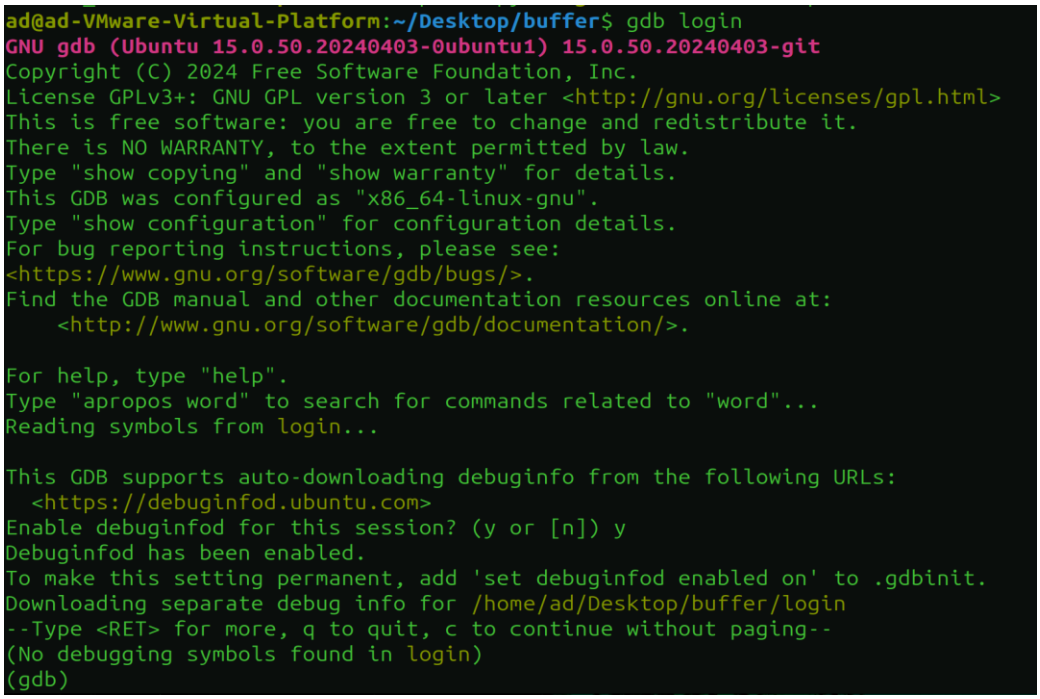
-fno-stack-protector : Disables stack protection mechanism

-z execstack : Allows the stack to be executable

-m32 : Compiles the code as a 32-bit binary

2. **Run the Program in GDB**

   gdb login



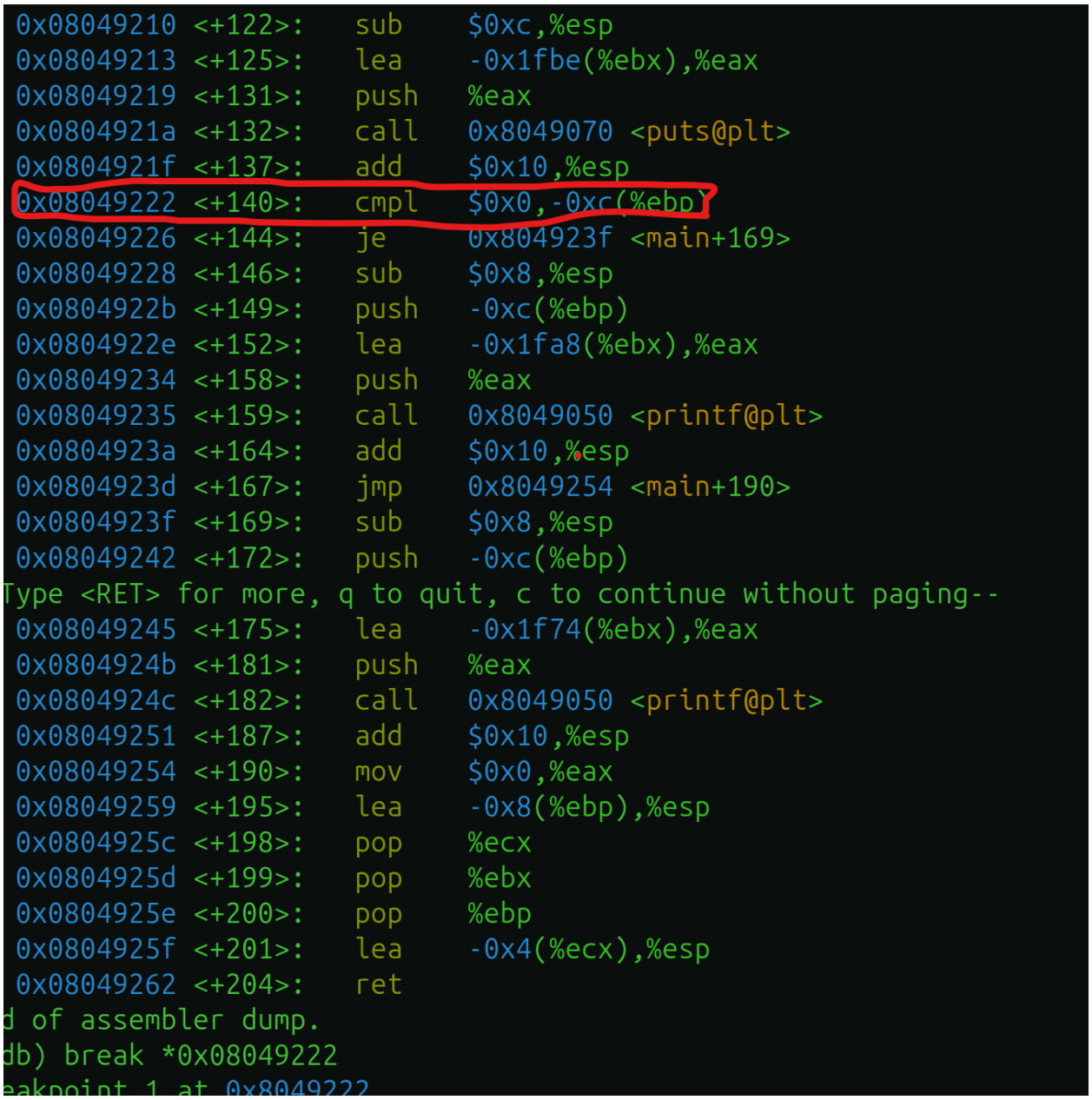
3. **Set a Breakpoint**

   First of all Here we have convert the code in to binary language.

   disas main

```
--Type <RET> for more, q to quit, c to continue without paging--
(No debugging symbols found in login)
(gdb) disas main
Dump of assembler code for function main:
   0x08049196 <+0>:     lea    0x4(%esp),%ecx
   0x0804919a <+4>:     and    $0xfffffff0,%esp
   0x0804919d <+7>:     push   -0x4(%ecx)
   0x080491a0 <+10>:    push   %ebp
   0x080491a1 <+11>:    mov    %esp,%ebp
   0x080491a3 <+13>:    push   %ebx
   0x080491a4 <+14>:    push   %ecx
   0x080491a5 <+15>:    sub    $0x10,%esp
   0x080491a8 <+18>:    call   0x80490d0 <__x86.get_pc_thunk.bx>
   0x080491ad <+23>:    add    $0x2e47,%ebx
   0x080491b3 <+29>:    movl   $0x0,-0xc(%ebp)
   0x080491ba <+36>:    sub    $0xc,%esp
   0x080491bd <+39>:    lea    -0x1fec(%ebx),%eax
   0x080491c3 <+45>:    push   %eax
   0x080491c4 <+46>:    call   0x8049070 <puts@plt>
   0x080491c9 <+51>:    add    $0x10,%esp
   0x080491cc <+54>:    sub    $0xc,%esp
   0x080491cf <+57>:    lea    -0x12(%ebp),%eax
   0x080491d2 <+60>:    push   %eax
   0x080491d3 <+61>:    call   0x8049060 <gets@plt>
   0x080491d8 <+66>:    add    $0x10,%esp
   0x080491db <+69>:    sub    $0x8,%esp
   0x080491de <+72>:    lea    -0x1fd5(%ebx),%eax
   0x080491e4 <+78>:    push   %eax
   0x080491e5 <+79>:    lea    -0x12(%ebp),%eax
   0x080491e8 <+82>:    push   %eax
```

Now setting the breakpoint.



Here we have set the break point. In that point it is comparing the user input password with the admin password.

```
   0x08049262 <+204>:    ret
End of assembler dump.
(gdb) break *0x08049222
Breakpoint 1 at 0x8049222
(gdb) run
Starting program: /home/ad/Desktop/buffer/login
Downloading separate debug info for system-supplied DSO at 0x
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthr
Enter admin password:
test
Incorrect Password!

Breakpoint 1, 0x08049222 in main ()
(gdb)
```

Here we run our program after setting the break point.

We entered the password but it is incorrect.

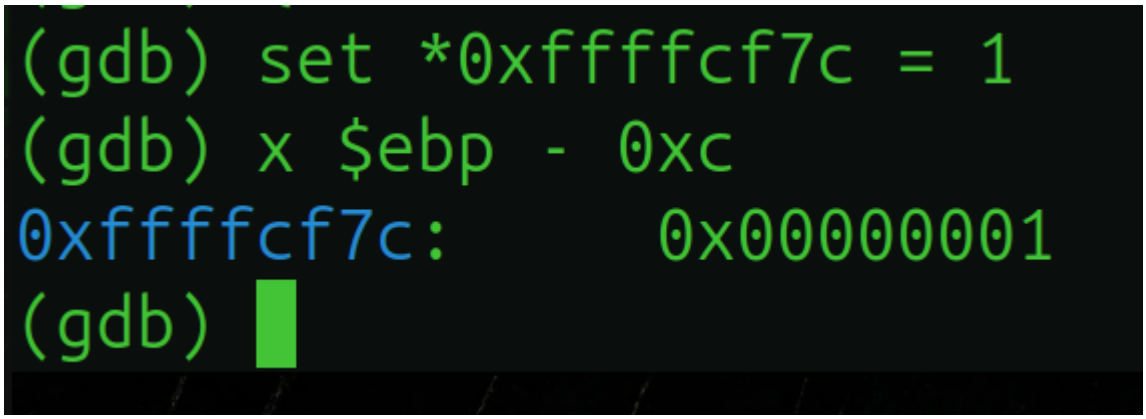
```
Breakpoint 1, 0x08049222 in main ()
(gdb) x $ebp - 0xc
0xffffcf7c:     0x00000000
(gdb)
```

Ebp is the extended base pointer.

The EBP register is often used to reference local variables and function parameters, which are stored on the stack.

0xffffcf7c is the address of the variable stored.

So we will change the value of 0xffffcf7c in order to run buffer overflow.



Here we have successfully changed the value of the ebp to 1

Now we will further run the program.

Here we have successfully login as the admin without the right password.

Defence Technique

Buffer overflow vulnerabilities can lead to severe security issues, such as unauthorized code execution, privilege escalation, and system crashes. To protect against these vulnerabilities, developers, compilers, and operating systems have implemented several effective defense mechanisms. Here are the main techniques in detail:

1. Stack Canaries

Stack canaries are small values (often random) placed between the local variables and the return address on the stack. When a function starts, the canary is placed just before the return address in memory.

If a buffer overflow occurs, the overflow is likely to overwrite this canary before reaching the return address. Before the function returns, the program checks if the canary value is intact. If it has been modified, the program detects an overflow and terminates execution or raises an alert.

For example, if a function has a stack canary set to 0xDEADBEEF, any change in this value when the function returns signals a potential overflow, allowing the program to respond immediately.

Limitations:

This protection works only for stack-based overflows, not for heap-based overflows or other memory corruption issues. Sophisticated attackers may try to guess the canary value, although modern canary values are often random and difficult to predict.

2. Address Space Layout Randomization (ASLR)

ASLR randomizes the memory addresses used by processes, including the stack, heap, and libraries. Every time a program runs, the starting addresses of these memory sections are randomized.

This randomization makes it difficult for an attacker to predict where specific functions or buffers are located in memory, which is essential for successful exploitation.

For example, In a system with ASLR, a buffer in one execution may be located at address 0x7fffd123 and in another run at 0x6fffc456. This unpredictability disrupts buffer overflow attacks that rely on specific memory addresses.

Limitations:

ASLR is less effective on systems that don't randomize all memory regions. It can be partially bypassed if an attacker has another way to leak memory addresses.

3. Safe Functions and Standard Library Alternatives

Using safer alternatives to functions that are vulnerable to buffer overflows, such as gets(), strcpy(), and sprintf(). Unsafe functions, like gets(), do not limit input size, making overflows easy to exploit. Safer functions include fgets() (which limits input size) and snprintf() (which safely formats strings without overflow).

Modern compilers and libraries provide safer alternatives that ensure the bounds of buffers are respected**.**

Example, Replace strcpy(buffer, userInput); with strncpy(buffer, userInput, sizeof(buffer) - 1);, ensuring that the function copies a limited amount of data and leaves space for the null terminator.

Role as a Ethical Hacker

As an ethical hacker, your role in understanding and addressing buffer overflow vulnerabilities is critical to ensuring system security. Ethical hackers work to identify, exploit, and mitigate such vulnerabilities in a controlled, legal manner to protect systems from malicious attacks. Here's an overview of the key responsibilities an ethical hacker has regarding buffer overflows:

1. Identify Vulnerabilities: Look for places in code where buffer overflows could happen.
2. Test Exploits Safely: Use controlled methods to show how an overflow might be exploited.
3. Develop Fixes: Suggest ways to fix or protect against buffer overflows, like safer code practices and security features.
4. Educate and Document: Teach others about buffer overflow risks and document your findings.
5. Work with Security Teams: Collaborate to ensure all areas are secure.

As an ethical hacker, your role in managing buffer overflows is multifaceted. You not only find and test vulnerabilities but also contribute to stronger defenses and create a security-first mindset within your organization. By detecting these flaws before attackers can exploit them, you actively protect systems and improve overall security resilience.

Conclusion

Buffer overflow vulnerabilities are critical security issues that can lead to serious consequences like unauthorized code execution, data corruption, and system compromise. Over the years, various defense mechanisms have been developed to mitigate the risk of buffer overflows, and they work effectively when combined.

It can lead to severe security risks, allowing attackers to manipulate memory and execute malicious code. Avoiding unsafe functions and enabling protection mechanisms like stack canaries and ASLR are essential for developing secure applications.

Reference

1) https://chatgpt.com/c/672d9cc7-7558-800f-aaca-08bd6e944c77
2) https://www.imperva.com/learn/application-security/buffer-overflow/
3) https://www.fortinet.com/resources/cyberglossary/buffer-overflow
4) https://www.cloudflare.com/learning/security/threats/buffer-overflow/
5) https://www.techtarget.com/searchsecurity/definition/buffer-overflow
6) https://www.veracode.com/security/buffer-overflow