Bypassing Authentication and Exploiting Buffer Overflows: A Security Analysis

5653630

February 5, 2025

Contents

1 Task 1

1.1 Stack-Based Buffer Overflows in x86 vs x86-64

1.1.1 Why x86-64 over x86?

1) Address Space & Randomisation

Firstly, x86-64 systems present greater challenges for binary exploitation compared to x86 systems, mainly due to their use of a 64-bit address space instead of the 32-bit [**?**]. This is because 64-bit systems use a larger address space, making it difficult for attackers to predict memory locations through computational methods, as the processor benefits from stronger address randomisation [**?**].

The same cannot be said for x86 systems, as they use a smaller address space, meaning their address randomisation is weaker compared to x86-64, allowing predictions about memory locations through computational methods to be much more feasible. This means that potential attackers will be forced to use side-channel attacks to speed up the process, increasing the attack complexity and reducing the chance of success [**?**].

2) Registers

Secondly, x86-64 is known for using a faster calling convention, which is beneficial as it reduces reliance on the stack for passing arguments [**?**]. The x86 system, on the other hand, relies more on stack-based argument passing, making exploitation easier as it increases the attack surface by storing function arguments in memory rather than registers.

Additionally, x86-64 systems use more general-purpose registers compared to x86 systems [**?**]. These registers are 64-bit, meaning not only can they hold larger values compared to x86 architecture, but they also reduce the need for programmers to use the stack to store values. This decreases the paths attackers can take to gain access to the stack from within the program, as programs simply do not access it as frequently. Furthermore, increased use of registers adds a layer of complexity for the programmer, as the attacker then needs to understand what each is used for and what values they store, further lowering the chance of success.

1.1.2 Defences

Compiler-Time Defences: Stack Canaries

To counteract stack-based buffer overflows, stack canaries are used as a detection mechanism to check whether unauthorised changes have been made to the stack while the function executes [**?**]. It works by placing a random value (canary) between local variables and the return address on the stack. Before a function returns, the system checks if this value has been altered. If it has, the program immediately terminates, preventing an attacker from executing malicious code.

Since modifying the return address requires overwriting the canary, an attacker would need to correctly guess its value. On modern 64-bit architectures, this is nearly impossible, with a success rate as low as 0.00000000000000000005421%. Furthermore, if the canary is randomised for each execution, brute-force attacks become infeasible.

However, this solution is not foolproof, since advanced hackers may attempt a byte-by-byte brute-force approach, reducing the number of guesses needed to 2,048 attempts [**?**]. Even so, this significantly increases the difficulty of successful exploitation since the success rate remains approximately at a low 0.0000000000000000111%.

Runtime Defences: Address Space Layout Randomisation (ASLR)

To further mitigate buffer overflow risks, ASLR randomises the memory layout of key regions such as the heap, stack, and executable code [**?**]. This forces attackers to work blindly, as the memory addresses change every time the program runs.

ASLR's effectiveness is better exploited in x86-64 architectures over x86, since 32-bit systems have smaller address spaces, making it feasible for attackers to brute-force their way around ASLR [**?**]. With x86-64,

however, the vastly larger address range makes brute-force impractical. To make things worse, if an exploit fails to execute, the program typically crashes rather than executing malicious code, which is a far better outcome than allowing a successful buffer overflow. Attackers attempting to bypass ASLR are then forced to resort to side-channel methods, which are more complex and less practical.

2 Task 2

2.1 Managing Directory Navigation and File Permissions

To first access the file, I navigated to my download directory where `Binary_Task2` was saved. Once inside, I gave myself all the necessary conditions by making the file executable, using the following command:

```
chmod +x Binary_Task2
```

After ensuring that the file is executable, I proceeded to the next step

2.2 Opening the File in gdb-peda

I analysed the file using gdb-peda, an enhanced debugging tool. The command used was:

```
gdb-peda ./Binary_Task2
```

Figure 1: Launching the Binary in gdb-peda.

2.3 Understanding the Main Function

Once inside the debugger, I disassembled main since it serves as the standard entry point in most compiled binaries. This provided the disassembly output shown in Figure 2, revealing the program's execution flow and key functionalities. The most notable function calls being at (main+39) and (main+75) as shown in Figure 2

This is because at (main+39), **printf@plt** is called, likely prompting the user for input. Since we were attempting to bypass a passphrase, it was reasonable to assume that this prompt was requesting the passphrase from the user. Once the passphrase was entered , it was processed and validated inside **check_passphrase** function at (main+75).

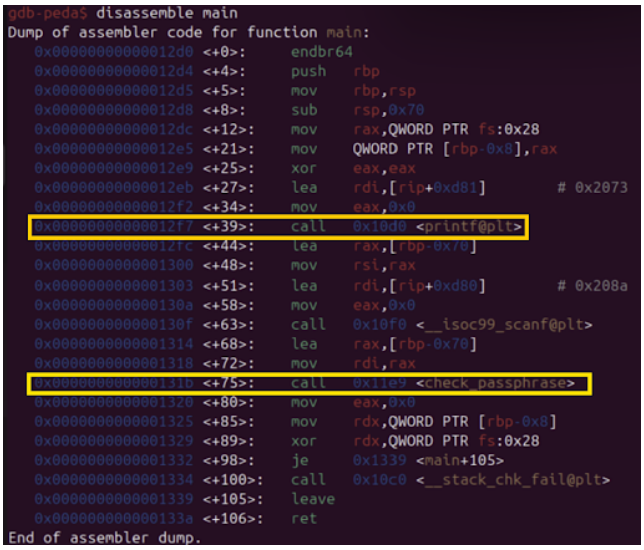

Figure 2: Inside of the main function.

2.4 Setting a Breakpoint

To investigate further, I set a breakpoint to analyse how the passphrase was checked, as shown in Figure 3

Figure 3: Setting a breakpoint

2.5 Disassembling the check_passphrase Function

With the breakpoint set, we continued to disassemble the function to analyse the internal structure, as shown in Figure 4

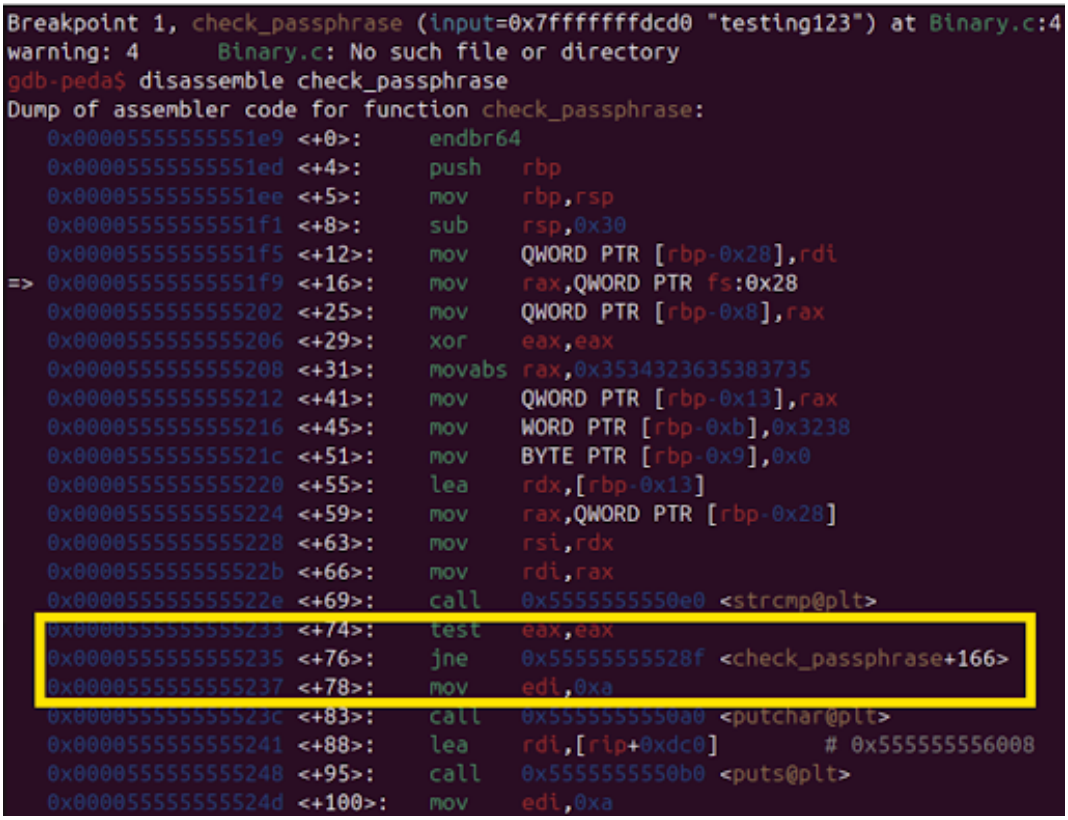

Figure 4: Examining the check_passphrase logic.

Inside the code, we saw multiple key functions and instructions being used, with the most standout ones covering from (main+69) to (main+78). This is because **strcmp@plt** was responsible for comparing strings character by character, and in our case, it was comparing our **rdi** (passphrase entered) with the **rsi** (correct passphrase). The result was then stored at **eax** and evaluated at (check_passphrase+74). If **eax = 0**, the zero flag (ZF) was set. Else, if **eax != 0**, then the zero flag (ZF) was cleared. At (check_passphrase+76), **jne** (jump not equal) checked the zero flag (ZF). If **ZF = 0**, it meant that the user input did not match the correct passphrase, and execution jumped to **check_passphrase+166**, likely handling failure. However, if **ZF = 1**, execution continued normally, likely granting access. Therefore, the key to bypassing this was to prevent **check_passphrase** from executing and instead force execution to continue from offset **(check_passphrase+78)**.

2.6 Bypassing Passphrase Validation

To do this, we set the instruction pointer **rip** to memory address **0x555555555237** at (check_passphrase+78). Since we already set a breakpoint at **check_passphrase** we simply continued execution. This forced the program to resume from (check_passphrase+78) as shown in Figure 5

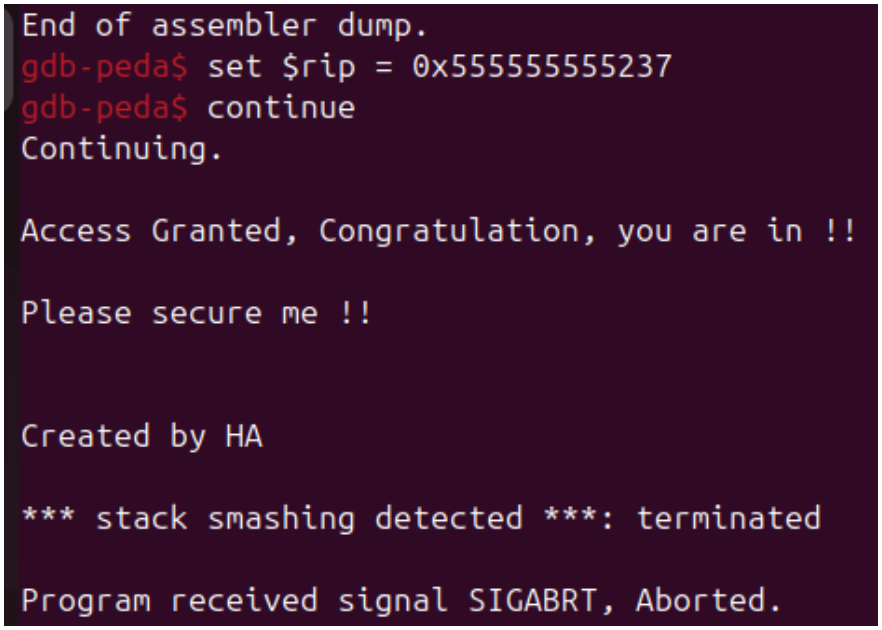

Figure 5: Confirming successful bypass.

With this method, I successfully bypass authentication and obtain the **"Access Granted"** message, confirming our exploit.

2.7 Mitigation

This section highlight the following protections, that if used correctly can make it harder for an attacker to bypass the program

2.8 Code Obfuscation

Most attackers will only be able to bypass the passphrase check once they understand the code. This is because seeing the way instructions jump from one memory address to another, enables the attackers to see the logic behind the program. For example, in our case, we were able to exploit the program once knowing we could simply jump the check_passphrase function.This insight came from disassembling the main function and identifying all the vulnerable call functions to exploit. Code obfuscation would work by utilising multiple jump instructions, function calls and by having sections of redundant code. Even though, this will not directly stop the attack, it will increase the complexity of the program, making it harder to follow the logic and exploit the vulnerabilities.

2.9 Encryption

Even though not shown here, there was a second way of bypassing the program. This involved viewing the password "5624582" from an insecure register that showed the password in plaintext. This would be prevented if a hashing algorithm such as SHA-256 got used, as it will hide the passphrase, preventing the attacker from using GDB to see the plaintext version of the password.

3 Task 3

3.1 Accessing the Binary

Following the same approach as Task 2, I first navigated to the Downloads directory to access the file. Once there, I launched the application using gdb-peda to begin debugging. These were the commands I used:

```
cd Downloads
gdb-peda ./Binary_Task3
```

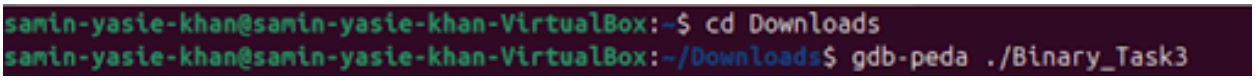

Figure 6: Checking security protections of the binary.

```
checksec
```

3.2 Checking Security Protections

I began the exploit by first checking the program's active defences using the **checksec** command. The results in figure 7 indicated that all major defenses were disabled, except for partial RELRO. However, this provided little protection in this case since it can only mitigate GOT-based buffer overflow attacks and not stack-based overflows.

Figure 7: Checking security protections of the binary.

3.3 Generating a Cyclic Pattern

Next I used the **pattern create**. to generate a 100 character cyclic pattern in order to identify the exact offset of the buffer overflow. Once the 100 characters were generated, I copied the output after the run command, in order to trigger a **SIGSEGV** signal (Segmentation Fault). This was to confirm that the program attempted to access an invalid memory address, likely overwriting the return address on the stack, causing a buffer overflow.

```
pattern_create 100
```

Figure 8: Generating a cyclic pattern.

I then checked the state of the stack frame through the **info frame** command. Here we confirmed that the input reached the return address since **RIP = 0x41334414165414149**, which was just the Hex representation of the cyclic cycle we used earlier on. We then used the pattern search command to find the offset value of **72**.

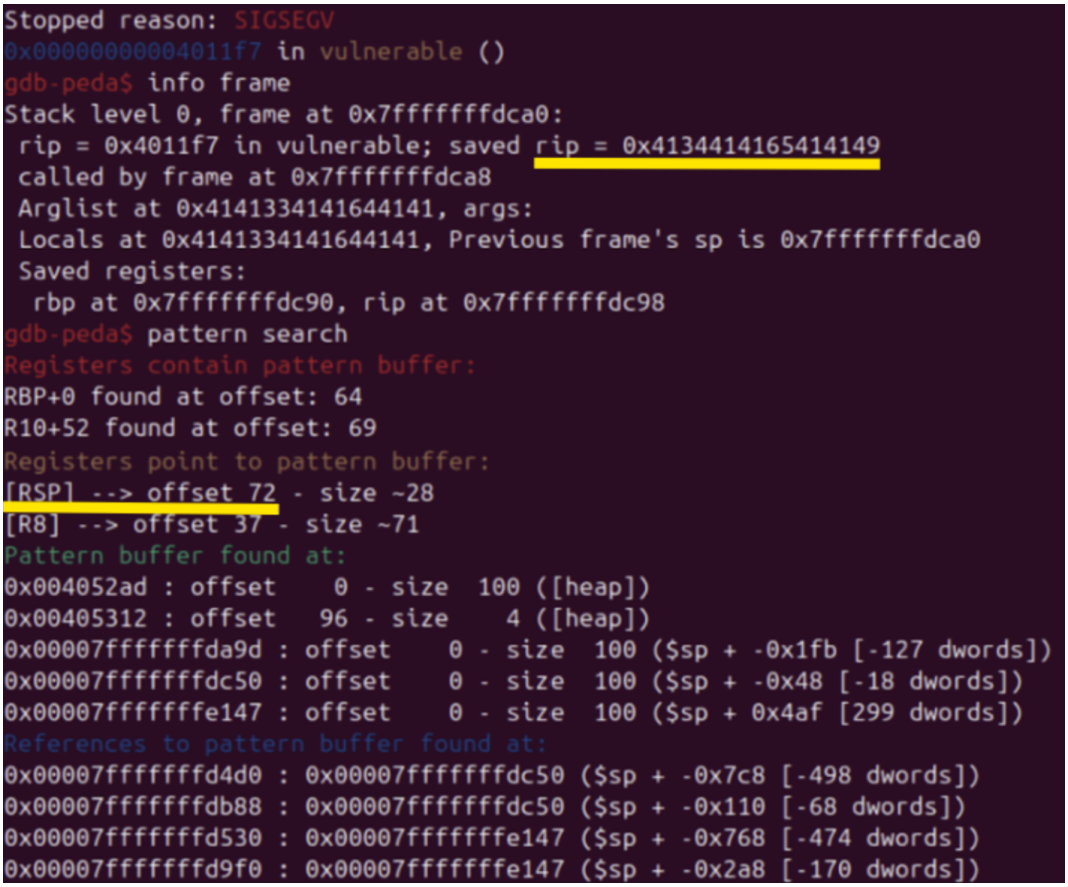

Figure 9: Inspecting the stack frame.

3.4 Finding the Buffer Overflow Offset

I then proceeded to overwrite the return address with a controlled value after determining the offset. In this case, `'BBBBBBB'` successfully overwrote the return address, as seen in the output above.

I then disassembled the shell code to take note of the entry point address, which will be used later to overwrite the return address in our exploit. This was to ensure that when execution reaches the overwritten return address, it will jump directly to the shell function, allowing us to gain control over the program.

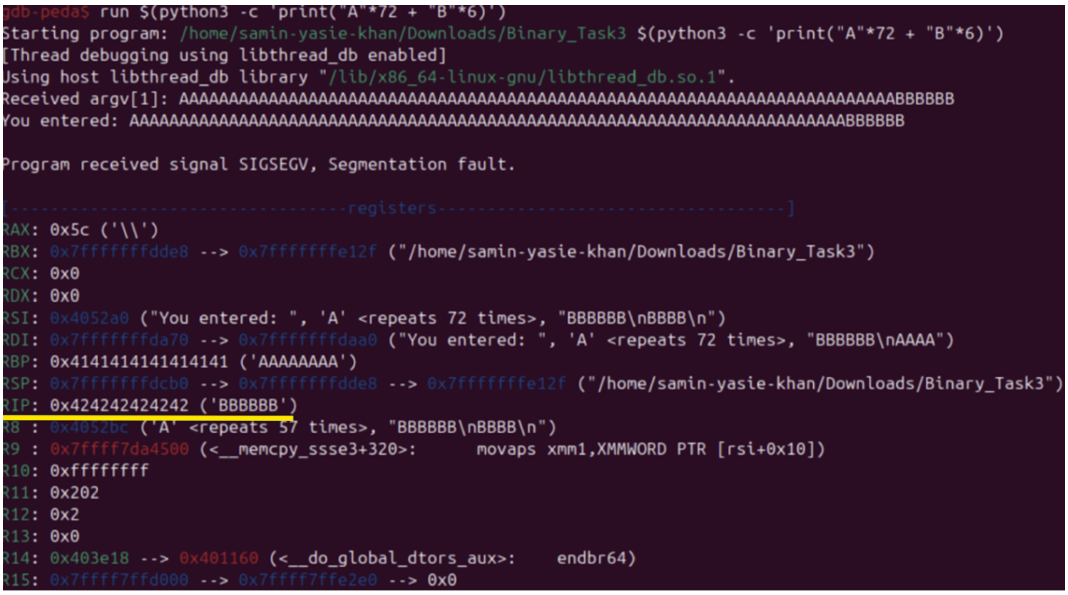

Figure 10: Overwriting the return address with a controlled.

I then disassembled the shell code to take note of the entry point address, which will be used later to overwrite the return address in our exploit. This was to ensure that when execution reaches the overwritten return address, it will jump directly to the shell function, allowing me to gain control over the program.

3.5 Disassembling the Shell Code

I then disassembled the shell code to take note of the entry point address, which will be used later to overwrite the return address in our exploit.

Figure 11: Disassembling the shell code and identifying the entry point address.

Next, we ran this code to trigger a buffer overflow.

3.6 Executing the Buffer Overflow

Next, we ran this code to trigger a buffer overflow.

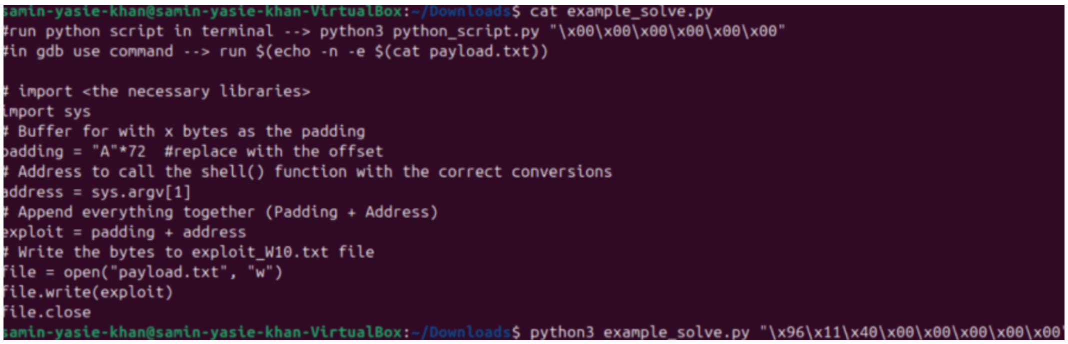

Figure 12: Code that triggers the overflow.

3.7 What This Script Does

What This Script Does:

1. Creates padding (`"A" * 72`) to reach the return address.
2. Takes the function address (`sys.argv[1]`) as an argument.
3. Appends the padding and address together to form the exploit.
4. Writes the payload into `payload.txt` for later use.

I then ran the script in little-endian format using the entry point address we noted earlier from the disassembled `shell()` function.

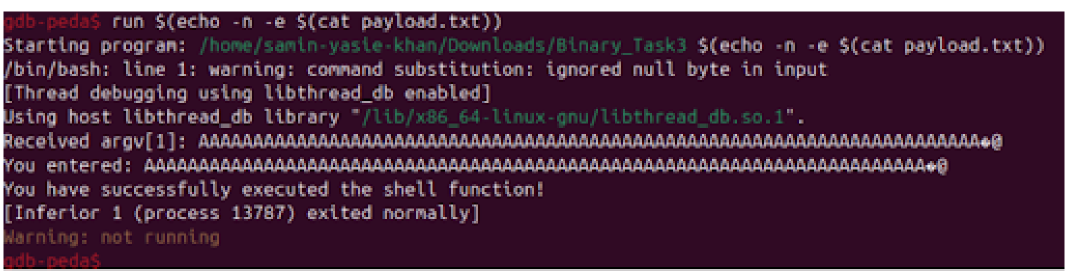

Figure 13: Executing the exploit successfully.

To finally execute the overflow, we ran the payload, which executed the binary (`Binary_Task3`) inside GDB with the exploit as input. The payload then successfully overwrote the return address (RIP) with `shell()`'s address. This was because the binary received a long string of “A” (before padding), and once the return address was successfully overwritten with the correct address, the program jumped to `shell()` instead of crashing and gave out the message `"You have successfully executed the shell function!"`, confirming execution of `shell()`.

3.7.1 Buffer Overflow Mitigations

3.8 Bounds Checking

Using safer functions like strncpy and memcpy will restrict input lengths to avoid overflows. In our case this would mean a restriction on the number of "A" we can input. This will mitigate the attack as the attacker will have less chance of creating a segmentation fault. Furthermore, compiler protections like enabling -fstack-protector on GCC, could also be used to prevent buffer overflow errors. If an overflow is detected, the program will terminate execution, preventing further exploitation.

References

[1] Marco-Gisbert, H., & Ripoll, I. (2014). 'On the Effectiveness of Full-ASLR on 64-bit Linux.' Universitat Politècnica de València. Available at: `https://web.archive.org/web/20150508073303/http://cybersecurity.upv.es/attacks/offset2lib/offset2lib-paper.pdf` (Accessed: 17 January 2024).

[2] Intel Corporation, 2023. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, pp. 3–10 - 3–11.

[3] Bowden, A., n.d. 'The Basics of Exploit Development 5: x86-64 Buffer Overflows.' Available at: `https://www.coalfire.com/the-coalfire-blog/the-basics-of-exploit-development-5-x86-64-buffer` (Accessed: 22 January 2024).

[4] Yoo, A. (2024). Stack Canary. YouTube. Available at: `https://www.youtube.com/watch?v=XXXXX` (Accessed: 19 January 2024).

[5] Tamir, C. (2019). 'WhatsApp Buffer Overflow Vulnerability: Under the Scope.' Zimperium. Available at: `https://www.zimperium.com/blog/whatsapp-buffer-overflow-vulnerability-under-the-scope/` (Accessed: 24 January 2024).

[6] Arpaci-Dusseau, R.H., & Arpaci-Dusseau, A.C. (2018). *Operating Systems: Three Easy Pieces*. Madison: Arpaci-Dusseau Books. Available at: `https://pages.cs.wisc.edu/~remzi/OSTEP/` (Accessed: 16 January 2024).