Task-1 Report: Buffer Overflow Vulnerability and Patching in `func0`

Introduction:

This report documents the process of demonstrating and fixing a buffer overflow vulnerability in a simple C program (`func0`). The program in question was designed to showcase how a common coding error can lead to security vulnerabilities, particularly buffer overflow.

Vulnerable Program (`task_1_function_0_vulnerable.c`)

Program Description:

The vulnerable program was designed to take a command line argument and copy it into a buffer without checking the length of the input. This was achieved using the `strcpy()` function, which does not provide bounds checking.

Source Code:

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

int main(int argc, char *argv[]) {
    char buffer[10];
    if (argc < 2) {
        printf("Usage: %s <input string>\\n", argv[0]);
        return 1;
    }
    strcpy(buffer, argv[1]);
    printf("You entered: %s\\n", buffer);
    return 0;
}
```

Vulnerability Analysis:

The vulnerability lies in the use of `strcpy()`, which leads to a buffer overflow if the input string exceeds the buffer size (10 characters). This was demonstrated by providing a longer input string, causing the program to crash or behave unexpectedly.

Patched Program (`task_1_function_0_patched.c`)

Program Description:

The patched program addresses the buffer overflow vulnerability by using `strncpy()` instead of `strcpy()`. This function limits the number of characters copied to the size of the buffer, preventing an overflow.

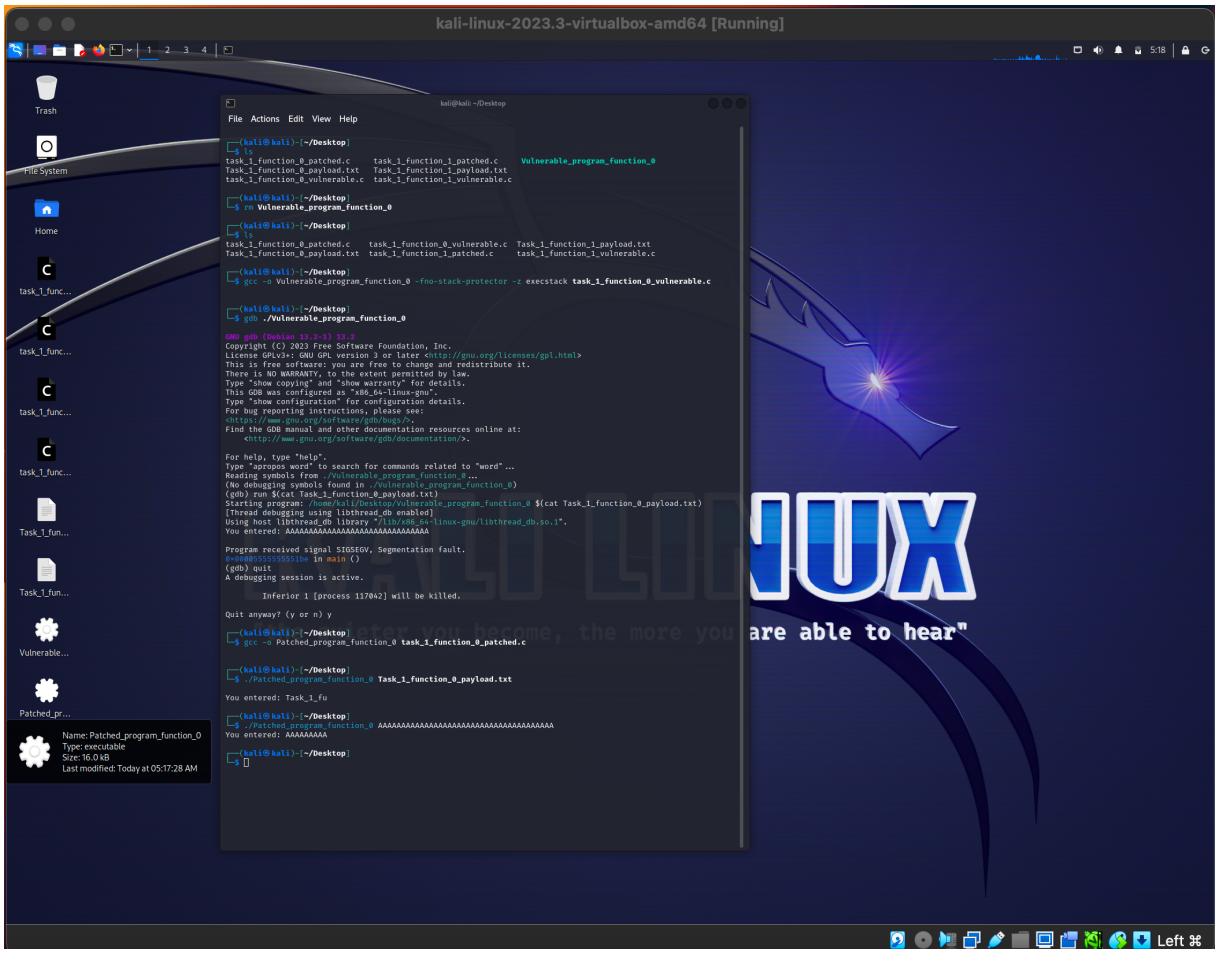
Source Code:

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

int main(int argc, char *argv[]) {
    char buffer[10];
    if (argc < 2) {
        printf("Usage: %s <input string>\\n", argv[0]);
        return 1;
    }
    strncpy(buffer, argv[1], sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\\0'; // Ensure null termination
    printf("You entered: %s\\n", buffer);
    return 0;
}
```

Fixation Verification:

The patched program was tested with inputs longer than the buffer size. The program successfully limited the copied characters to the buffer's length and did not crash, demonstrating that the buffer overflow vulnerability was effectively patched.



Conclusion:

The `func0` exercise provided a valuable lesson in the importance of secure coding practices. The use of unsafe functions like `strcpy()` can lead to critical vulnerabilities like buffer overflows. By replacing `strcpy()` with `strncpy()`, the vulnerability was mitigated, showcasing a simple yet effective method for enhancing the security of a C program.

Task-1 Report: Buffer Overflow Vulnerability and Resolution in `func1`

Introduction:

This report outlines the identification and rectification of a buffer overflow vulnerability in a simple C program referred to as `func1`. The objective is to demonstrate the critical nature of secure coding practices to prevent common vulnerabilities.

Vulnerable Program (`task_1_function_1_vulnerable.c`)

Program Overview:

The initial version of the program utilized `sprintf()` to copy user input into a buffer. However, `sprintf()` does not check the size of the input against the buffer's capacity, leading to potential buffer overflow vulnerabilities.

Source Code Snippet:

```
sprintf(buffer, "Input: %s", argv[1]);
```

Vulnerability Demonstration:

The program was compiled with GCC using flags to disable certain security features (`-fno-stack-protector -z execstack`) and executed with a long string input. A segmentation fault (`SIGSEGV`) occurred, indicative of a buffer overflow.

GDB Analysis:

Running the program in GDB with the payload from `Task_1_function_1_payload.txt` confirmed the buffer overflow, as evidenced by a segmentation fault at the point of executing the `sprintf()` command.

Patched Program (`task_1_function_1_patched.c`)

Program Modification:

To rectify the vulnerability, the program was modified to use `snprintf()` instead of `sprintf()`. `snprintf()` safely limits the number of characters written to the size of the buffer.

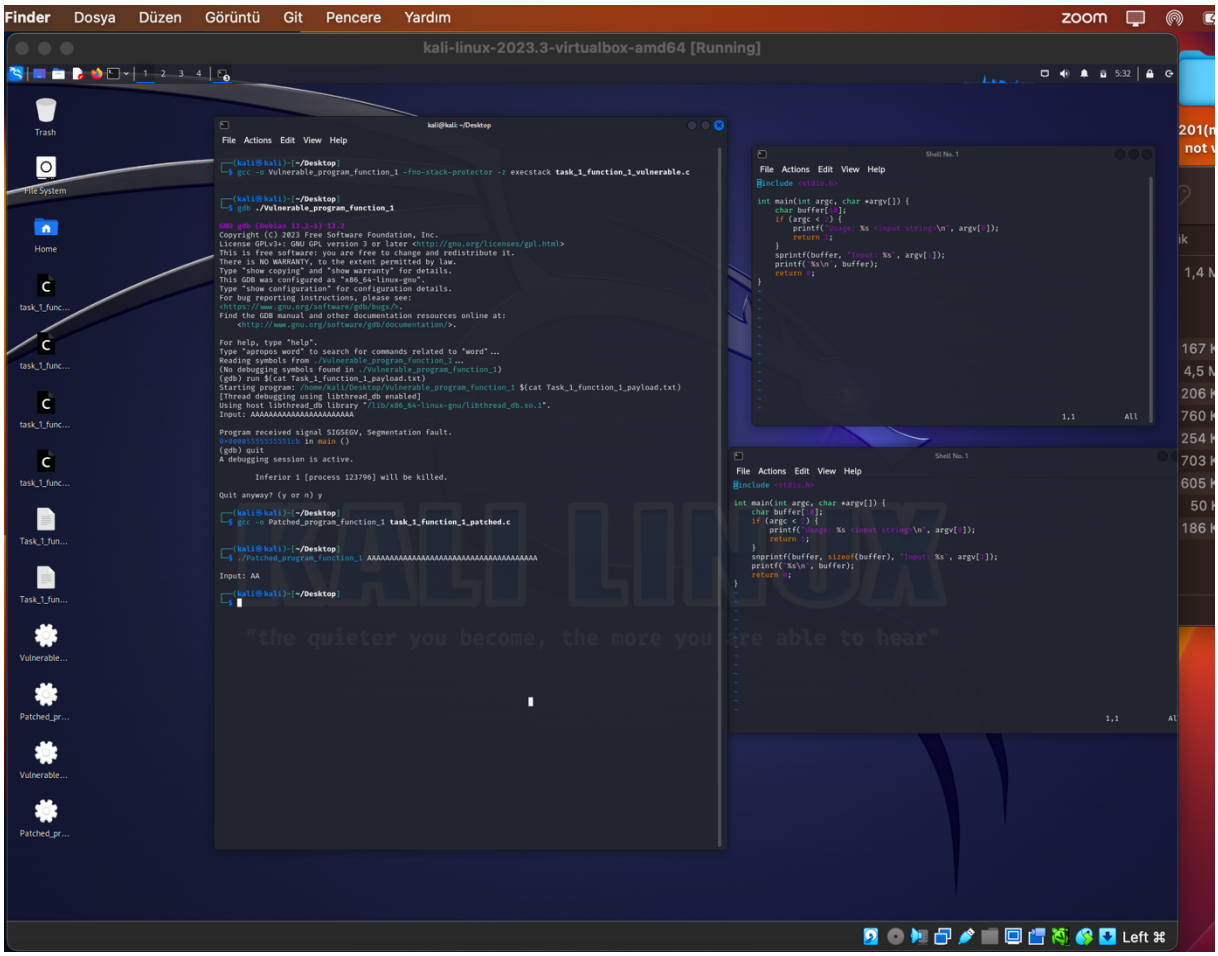
Source Code Snippet:

```
snprintf(buffer, sizeof(buffer), "Input: %s", argv[1]);
```

Verification of Fix:

The patched program was then compiled and executed with a similar long string input. The output confirmed that only a portion of the input (up to the

buffer's limit) was processed, effectively preventing a buffer overflow. The program executed without any segmentation faults, demonstrating the effectiveness of the fix.



Conclusion:

The exercise with `func1` clearly illustrates the dangers of buffer overflow vulnerabilities in C programs and the importance of using safe functions like `snprintf()` for buffer operations. The modification from `sprintf()` to `snprintf()` successfully prevented buffer overflow, showcasing a vital practice in secure coding.

Task 2:

Link of my Chatgpt prompt:

https://chat.openai.com/share/90e71340-c125-4da7-8507-7ccbe8988822

Prompt 1 **Integer Overflow Vulnerability**:
“Write a C program that performs a financial calculation, such as calculating the balance after multiple transactions, using unsigned integers.”

Introduction:

This report details the integer overflow vulnerability demonstrated through a simple C program. Integer overflow occurs when an arithmetic operation results in a numeric value that exceeds the maximum representable value for a given data type, causing the value to wrap around to the minimum representable value.

Vulnerable Code Analysis:

Source Code Snippet:

```
// ... [Initial setup code]
if (operation == 'D' || operation == 'd') {
    balance += transaction;
}
```

Vulnerability Demonstration:

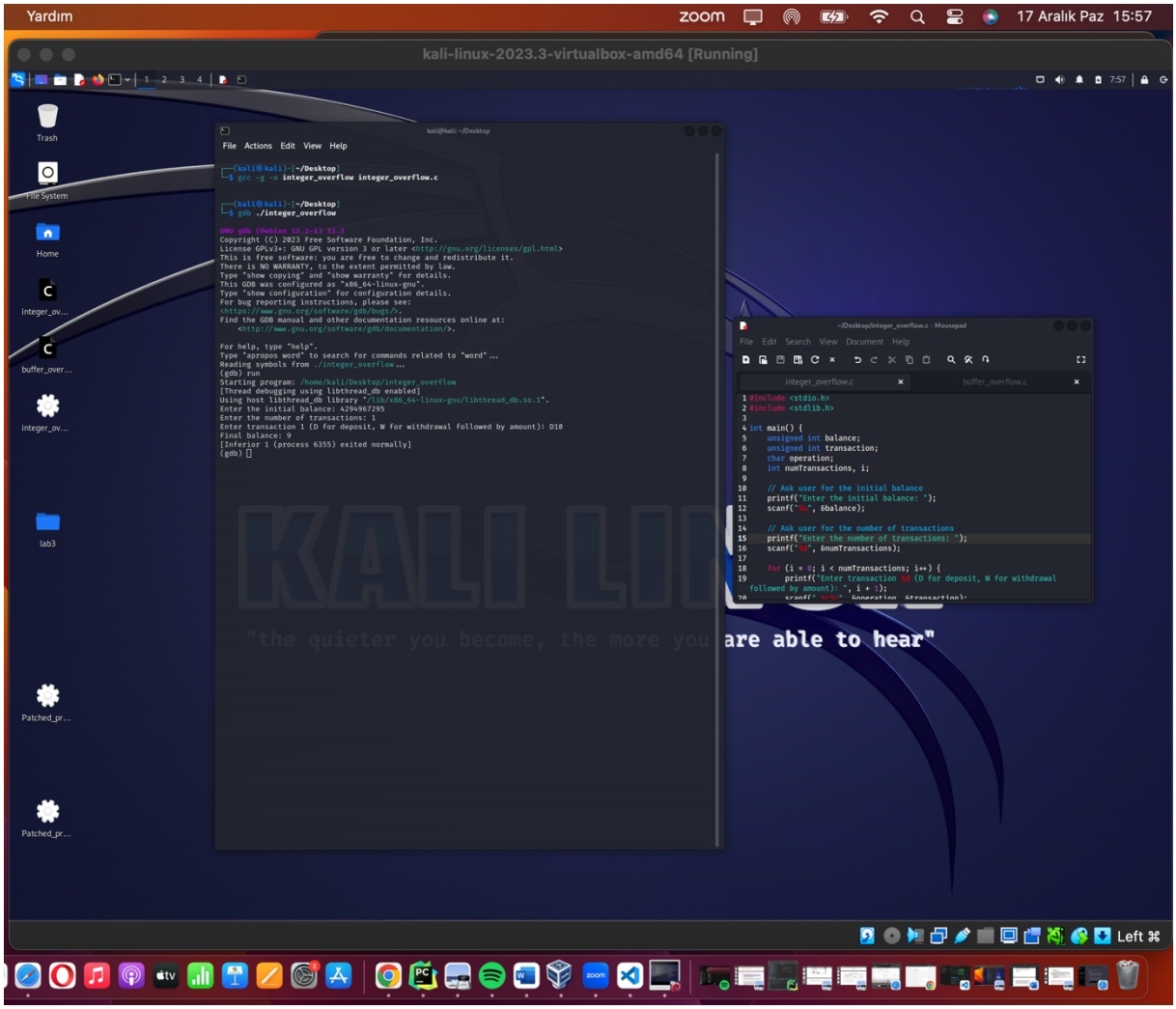
The vulnerability is illustrated in the above snippet where the balance is updated with a transaction value. An integer overflow occurs when adding any positive amount to the balance if the balance is already at its maximum value for an `unsigned int`.

Compilation and Execution:

The program was compiled with the following command:
gcc -g -o integer_overflow integer_overflow.c

The `-g` option was used to include debugging symbols for analysis with GDB.

The program was executed, and the following inputs were provided:



Result and Analysis:

After executing the deposit operation, the balance was expected to increase. However, due to the integer overflow, the final balance displayed was `9`, indicating that the balance wrapped around past the maximum value for an unsigned 32-bit integer (`UINT_MAX` or `4294967295`) and continued from `0`.

This confirms the presence of an integer overflow vulnerability, as the program logic does not account for the possibility that the balance can overflow, leading to an incorrect and potentially exploitable financial calculation.

GDB Analysis:

The program was also analyzed using GDB, and the same inputs were provided. The program did not crash, and GDB confirmed normal program termination, which is consistent with the defined behavior of unsigned integer overflow in C:

```
(gdb) run
Starting program: /home/kali/Desktop/integer_overflow
[...]
Final balance: 9
[Inferior 1 (process 6355) exited normally]
```

Conclusion:

The program provided for this task effectively demonstrates an integer overflow vulnerability. It shows that even when a program does not crash, logical errors due to integer overflow can lead to significant issues, particularly in domains where accurate numerical computations are critical.

Prompt 2 **Buffer Overflow Vulnerability**:

“Create a C function that concatenates two user-provided strings without using standard library functions for string manipulation.”

When I run this Prompt it does not shown any vulnerable activity so I skipped and asked another questions and I decided to pick last prompt.

String Format Injection Vulnerability Report:

Last Promt: “Can you write a simple C program that takes a string input from the user and then uses printf to display it? The program should directly pass the user's input to printf without any formatting or validation.”

Code Overview (`string_format.c`):

Source Code:

```
#include <stdio.h>

int main() {
    char inputString[100];

    printf("Enter a string: ");
    fgets(inputString, sizeof(inputString), stdin);

    // Vulnerable usage: printing the input directly without format specifiers
    printf(inputString);

    return 0;
}
```

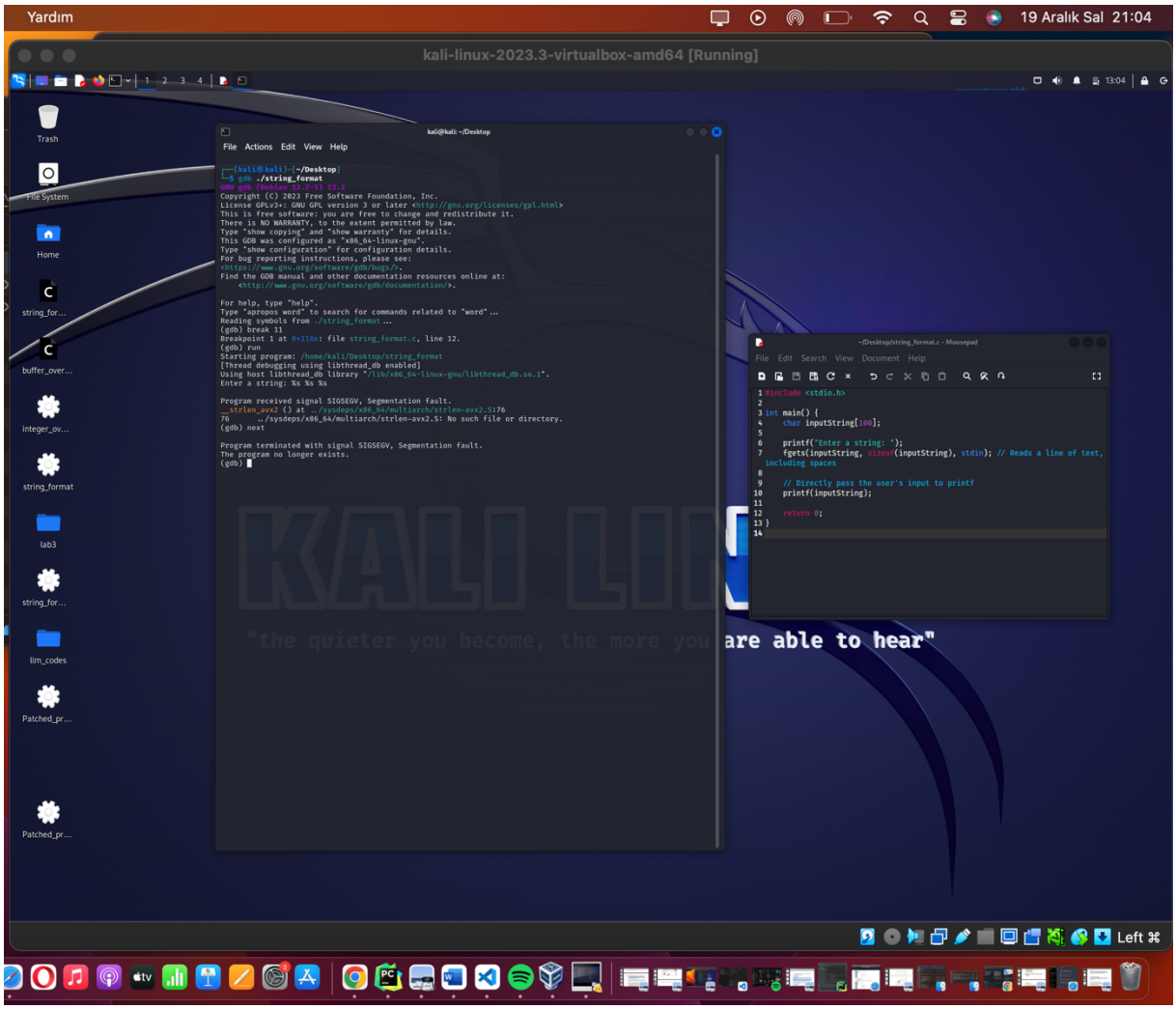
Compilation and Execution:

The program was compiled with GCC and executed to demonstrate the vulnerability.

Testing Methodology:

1. Direct Execution:

- Direct Execution: **%s %s %s** was input to test for format string vulnerabilities. The program echoed the input, leading to a segmentation fault (**SIGSEGV**), indicating the presence of the vulnerability.
- Execution within GDB: The program was also run in GDB with the same input, where it experienced a segmentation fault at **printf(inputString);**, further confirming the vulnerability.



Vulnerability Analysis:

The vulnerability arises from using user input directly in **printf(inputString);**. This can lead to security issues as an attacker could provide a format string that causes **printf** to access arbitrary memory locations.

Observation and Conclusion:

- The behavior of **printf** with **%s %s %s** input typically would result in printing out values from the stack, indicative of a string format injection vulnerability.
- The observed segmentation fault confirms the vulnerability, highlighting the importance of secure coding practices, especially in handling user input and format strings in functions like **printf**.