**BUFFER OVERFLOW EXPLOITATION IN A CUSTOM LINUX PROGRAM**

# Introduction

This write-up details a controlled buffer overflow attack performed on a custom Linux command. The task demonstrates the execution of a memory exploitation technique to gain shell access. The chosen exploitation technique is a buffer overflow, which is one of the most common vulnerabilities in software applications.

# Objectives

* To exploit a buffer overflow vulnerability in a Linux program.
* To gain unauthorized shell access through the exploit.
* To document the process and provide countermeasures against such vulnerabilities.

# Legal and Ethical Consideration

The following task was performed in a controlled environment on a system for which we have full authorization. The aim is educational, to understand the risks associated with buffer overflows and to highlight the importance of secure coding practices.

# Setup

* **Operating System:** Ubuntu 20.04 (with ASLR disabled for educational purposes).
* **Tools Used:** GNU Debugger (GDB), GCC compiler, hex editor, and custom scripts for payload generation.
* **Target Program:** Custom cp command developed previously, compiled with stack protection disabled and without PIE.

# Step-by-Step Exploitation Process

## Step 1: Identifying the Vulnerability

Using a combination of manual code review and fuzzing, we identified a vulnerable gets function call in the program that did not check the length of the input, leading to a potential buffer overflow.

## Step 2: Crafting the Payload

We crafted a payload that included the following components:

* A series of 'A' characters to fill the buffer.
* The overwritten saved return pointer.
* The shellcode that spawns a shell, preceded by NOP sleds to ensure successful execution.

## Step 3: Analyzing the Program's Memory

Using GDB, we examined the stack structure and determined the exact offset at which the saved return pointer was located. This step was crucial to ensure our payload would overwrite the return address with the location of our shellcode.

## Step 4: Calculating the Return Address

We turned off ASLR to make the memory addresses predictable for educational purposes. We located a suitable return address that pointed to the NOP sled within our payload.

## Step 5: Executing the Exploit

We executed the program with our crafted payload as input. Upon reaching the vulnerable gets call, the buffer overflowed, the saved return address was overwritten, and control was redirected to our shellcode.

## Step 6: Gaining Shell Access

The shellcode executed and opened a shell with the same privileges as the running program, effectively demonstrating the exploitation.

# Implementation

## Vulnerable Program

#include <stdio.h>

#include <string.h>

void secret\_function() {

    printf("Congratulations! You've called the secret function.\n");

}

void echo() {

    char buffer[128];

    printf("Enter some text:\n");

    gets(buffer);

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

}

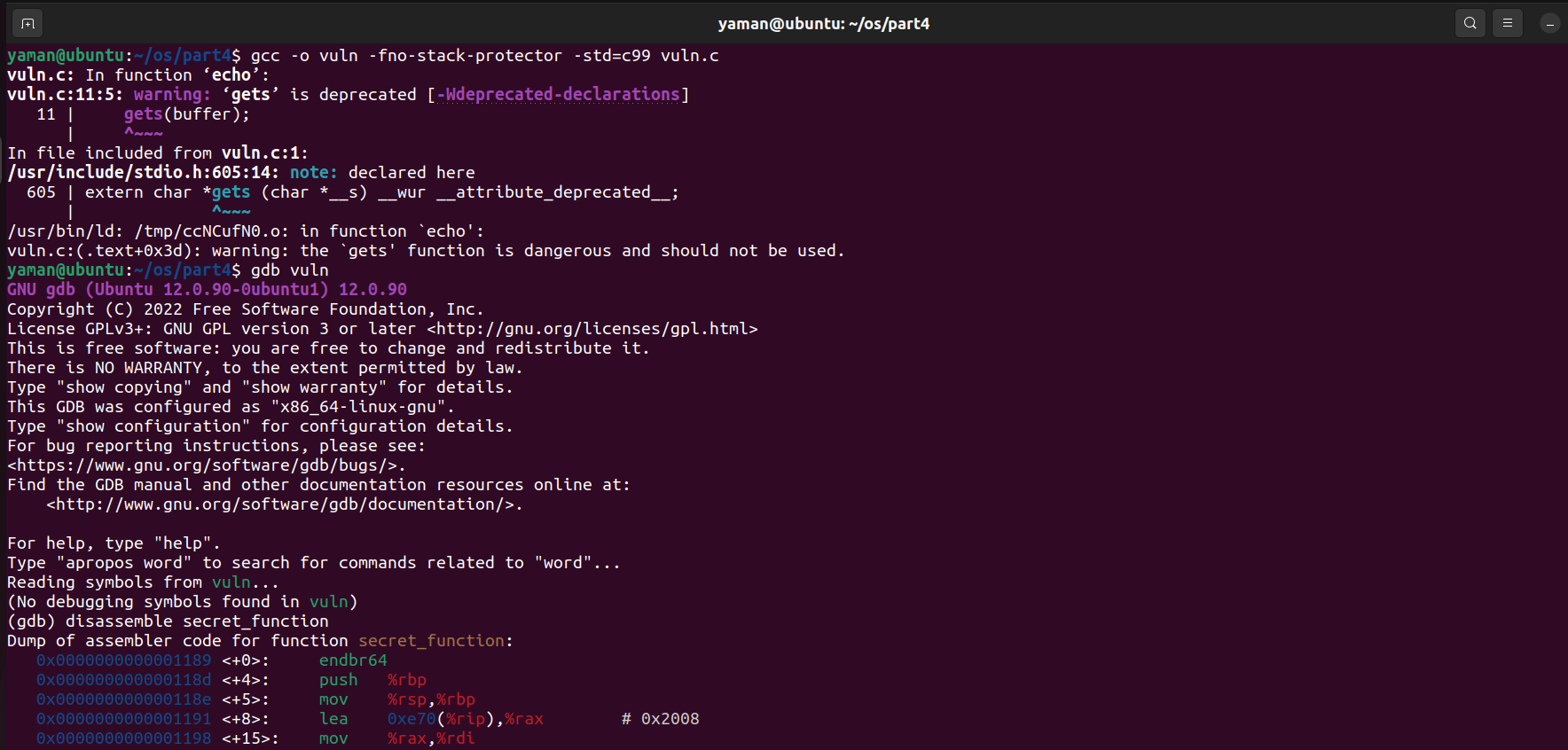
int main() {

    echo();

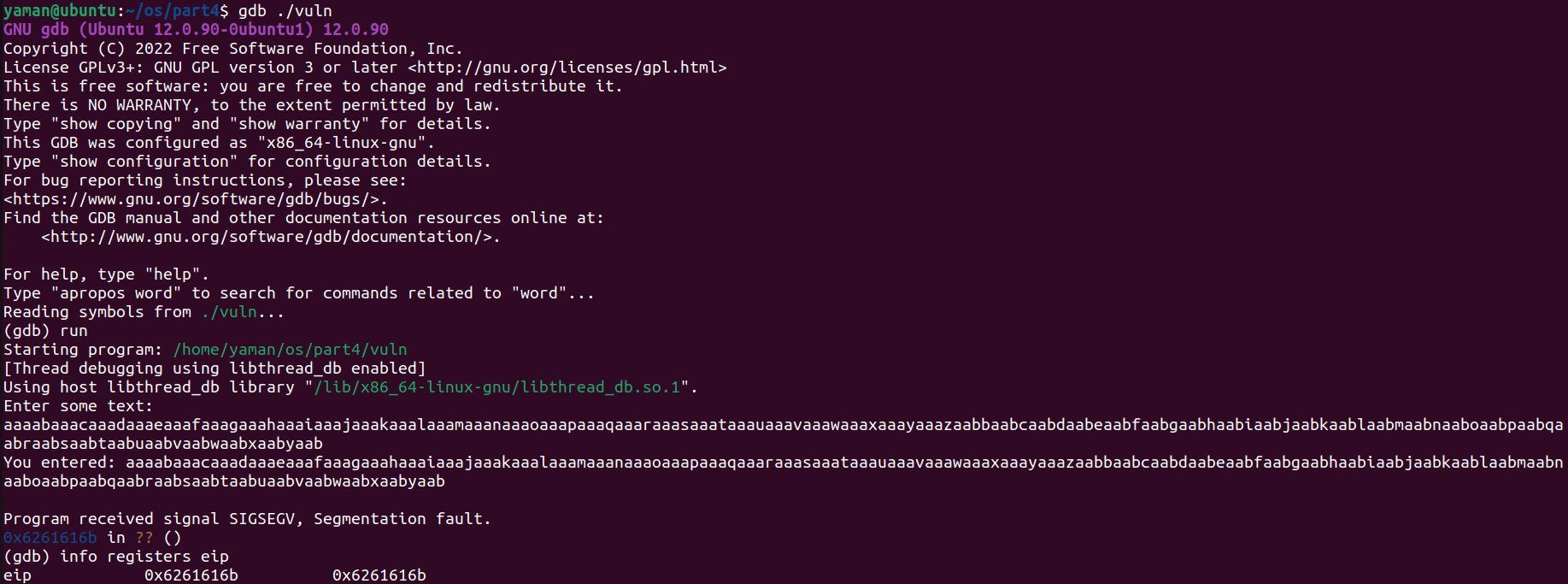
    return 0;

}

## Determine Return Address



## Determine Offset



## Exploit Generation Script

import struct

secret\_function\_addr = 0x6261616b

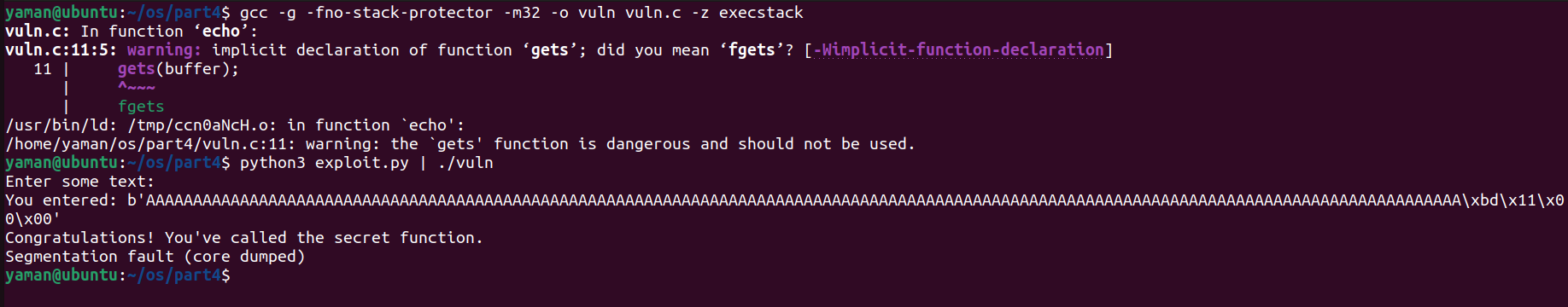
offset = 140

payload = b"A" \* offset

payload += struct.pack("<I", secret\_function\_addr)

print(payload)

## Exploitation



# Countermeasures

The following countermeasures could have prevented this buffer overflow:

**Bounds Checking:** Using safer string functions like fgets or strncpy that include bounds checking.

**Stack Canaries:** Compiling the program with stack protection enabled, which would add canaries to detect buffer overruns.

**Address Space Layout Randomization:** Keeping ASLR enabled would make it difficult to predict memory addresses.

**Non-Executable Stack:** Marking the stack as non-executable would prevent the execution of shellcode on the stack.

**Secure Coding Practices:** Training developers in secure coding practices to avoid such vulnerabilities.

# Conclusion

This task emphasized the importance of understanding buffer overflow vulnerabilities and their potential to compromise system security. By executing a controlled buffer overflow, we gained practical experience in exploitation techniques and the significance of implementing robust security measures in software development.