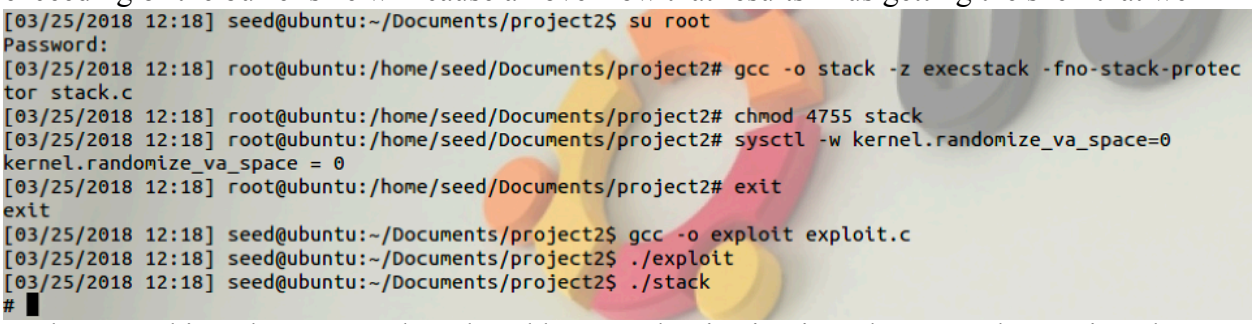
Lab 2 Report
Andrew Cox, Sebastian Martin, Joy Ray, Xiyuan Zheng

Task 1: The key to the stack buffer overflow is to write so many chars to the buffer that the content of the buffer becomes greater than the actual size. To do this, we fill the start of the buffer with multiple iterations of the return address, which is where the pointer is supposed to return to after running the shell code. We use the return address to fill in this space because using anything else would run the risk of altering the data values in the registers in a way that we do not want to. Once the return address iterations have been written, we simply finish the buffer off with the provided shell code. When the assembly code is run from the compiled "badfile", the exceeding of the buffer size will cause an overflow that results in us getting the shell that we



Task 2: For this task, we turned on the address randomization in order to combat against the stack buffer overflow attack. With this activated the address that we are trying to target with the malicious code with continuously move around combating the attack. In order to get around this it is mostly just a process of trial and error, until we have correctly hit the address thus, calling the malicious code and receiving the shell. Through our testing, there is quite a wide range where this will actually happen, sometimes very quickly, and sometimes it will take a long time for the correct address to appear, and for the exploit to be called. Because we do eventually get the shell, we see that our exploit.c is correct.

```
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
#
```

Task 3: No, I am not able to get the shell. In order to perform this task, I need to turn off address randomization defense. Then I repeated the buffer overflow attack again with GCC's Stack Guard. Stack Guard enhances the code by detecting buffer overflow attacks against stack. It allows us to prevent the attempt of changing address before the function returns. Therefore, the program was terminated, since the smash attack was detected.

```
03/23/2018 15:31] seed@ubuntu:~/csce548/project2$ rm stack
m: remove write-protected regular file `stack'? yes
03/23/2018 15:31] seed@ubuntu:~/csce548/project2$ ls
adfile  call_shellcode  call_shellcode.c  exploit  exploit.c  stack.c
03/23/2018 15:32] seed@ubuntu:~/csce548/project2$ sudo sysctl -w kernel.randomi
e_va_space=0
sudo] password for seed:
ernel.randomize_va_space = 0
03/25/2018 16:42] seed@ubuntu:~/csce548/project2$ gcc -0 stack -z execstack stack.c
cc: error: stack: No such file or directory
cc: error: unrecognized option '-0'
03/25/2018 16:45] seed@ubuntu:~/csce548/project2$ gcc -o stack -z execstack stack.c
03/25/2018 16:45] seed@ubuntu:~/csce548/project2$ chmod 4755 stack
03/25/2018 16:46] seed@ubuntu:~/csce548/project2$ ls
adfile  call_shellcode  call_shellcode.c  exploit  exploit.c  stack  stack.c
03/25/2018 16:46] seed@ubuntu:~/csce548/project2$ ./stack
```

```
[0x90909090]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1582559    /home/seed/csce548/project2/stack
08049000-0804a000 r-xp 00000000 08:01 1582559    /home/seed/csce548/project2/stack
0804a000-0804b000 rwxp 00001000 08:01 1582559    /home/seed/csce548/project2/stack
0804b000-0806c000 rwxp 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.1
b7e0b000-b7e0c000 r-xp 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.1
b7e0c000-b7e0d000 rwxp 0001c000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so.1
b7e1f000-b7e20000 rwxp 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc5000 r-xp 001a3000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc6000 rwxp 001a5000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.so
b7fc6000-b7fc9000 rwxp 00000000 00:00 0
b7fd9000-b7fdd000 rwxp 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-xp 0001f000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rwxp 00020000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rwxp 00000000 00:00 0          [stack]
Aborted (core dumped)
[03/25/2018 16:47] seed@ubuntu:~/csce548/project2$
```

Task 4: No, I cannot get to the shell when using "noexecstack." When I make the addresses in the stack non-executable it prevents the exploit.c code because all the writable addresses in the stack are now non-executable. Since the addresses now cannot be executed the code in exploit.c will not work because it is dependent on being able to get to the return address of the stack. Although this makes it difficult to cause a buffer overflow it does not prevent the buffer overflow completely. The "return-to-libc" attack would get around the "noexecstack" and this is done because it calls functions that are in libc and do not reside in the stack.

```
Terminal
[03/24/2018 12:44] seed@ubuntu:~/Project2$ sudo sysctl -w kernel.randomize_va_sp
ace=0
kernel.randomize_va_space = 0
[03/24/2018 12:44] seed@ubuntu:~/Project2$ su root
Password:
[03/24/2018 12:44] root@ubuntu:/home/seed/Project2# gcc -o stack -fno-stack-prot
ect                                tack.c
Firefox Web Browser
[03/24/2018 12:45] root@ubuntu:/home/seed/Project2# ls
badfile           exploit     -fno-stack-protector  stack.c
call_shellcode    exploit.c   Lab 2 Report.odt      stack-fno-stack-protector
call_shellcode.c  exploit.c~  stack
[03/24/2018 12:45] root@ubuntu:/home/seed/Project2# chmod 4755 stack
[03/24/2018 12:45] root@ubuntu:/home/seed/Project2# exit
exit
[03/24/2018 12:45] seed@ubuntu:~/Project2$ ./exploit
[03/24/2018 12:45] seed@ubuntu:~/Project2$ ./stack
Segmentation fault (core dumped)
[03/24/2018 12:45] seed@ubuntu:~/Project2$
```