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.

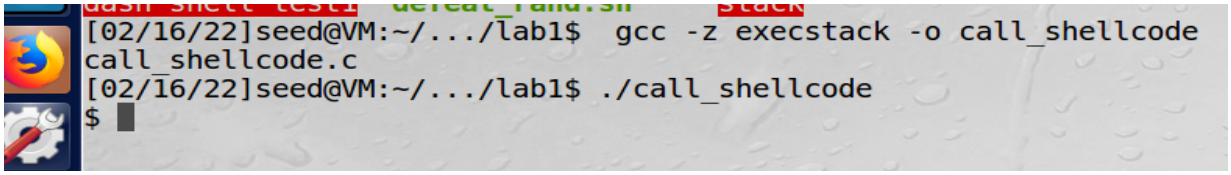

Figure 1 : executing shellcode

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.

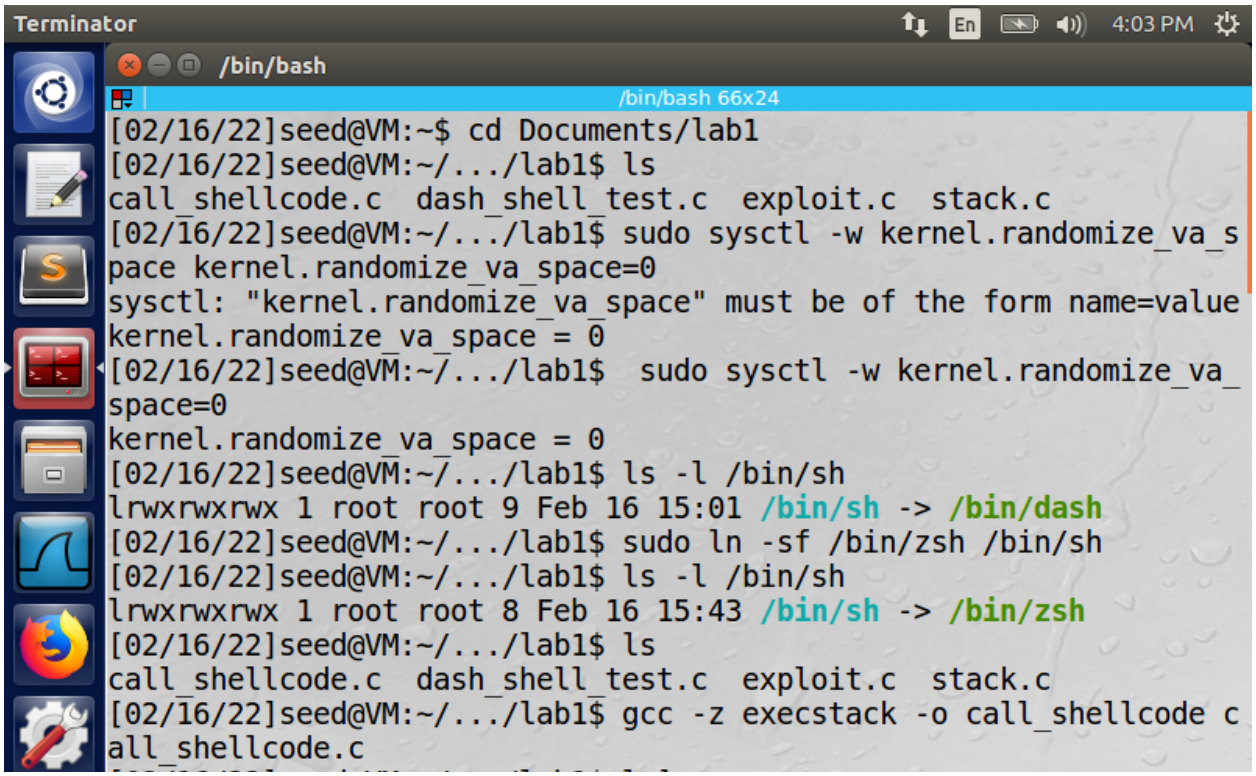

Figure 2

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.

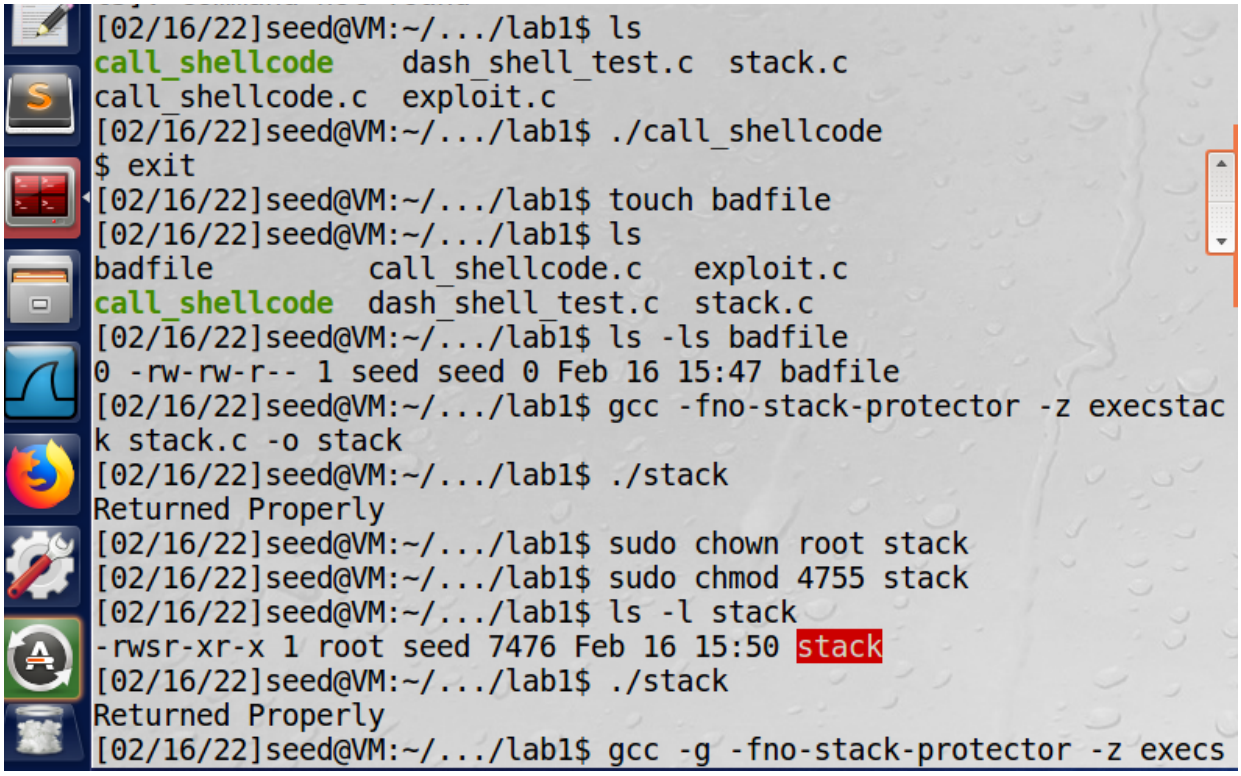

Figure 3

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.

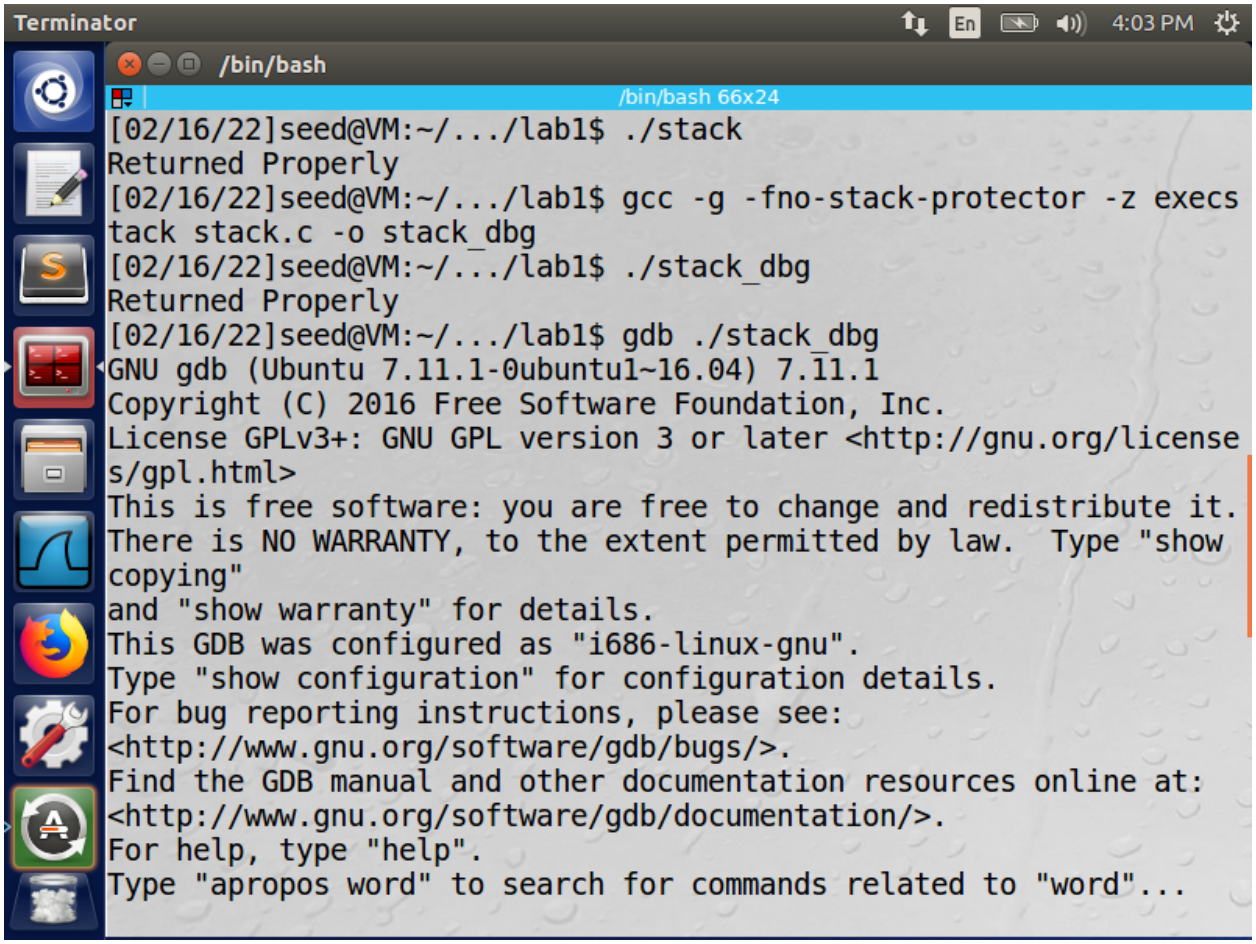

Figure 4

In Figure 5, we disassemble the function buf() which includes the vulnerable statement. First we find the breakpoint with ‘b bof’ 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.

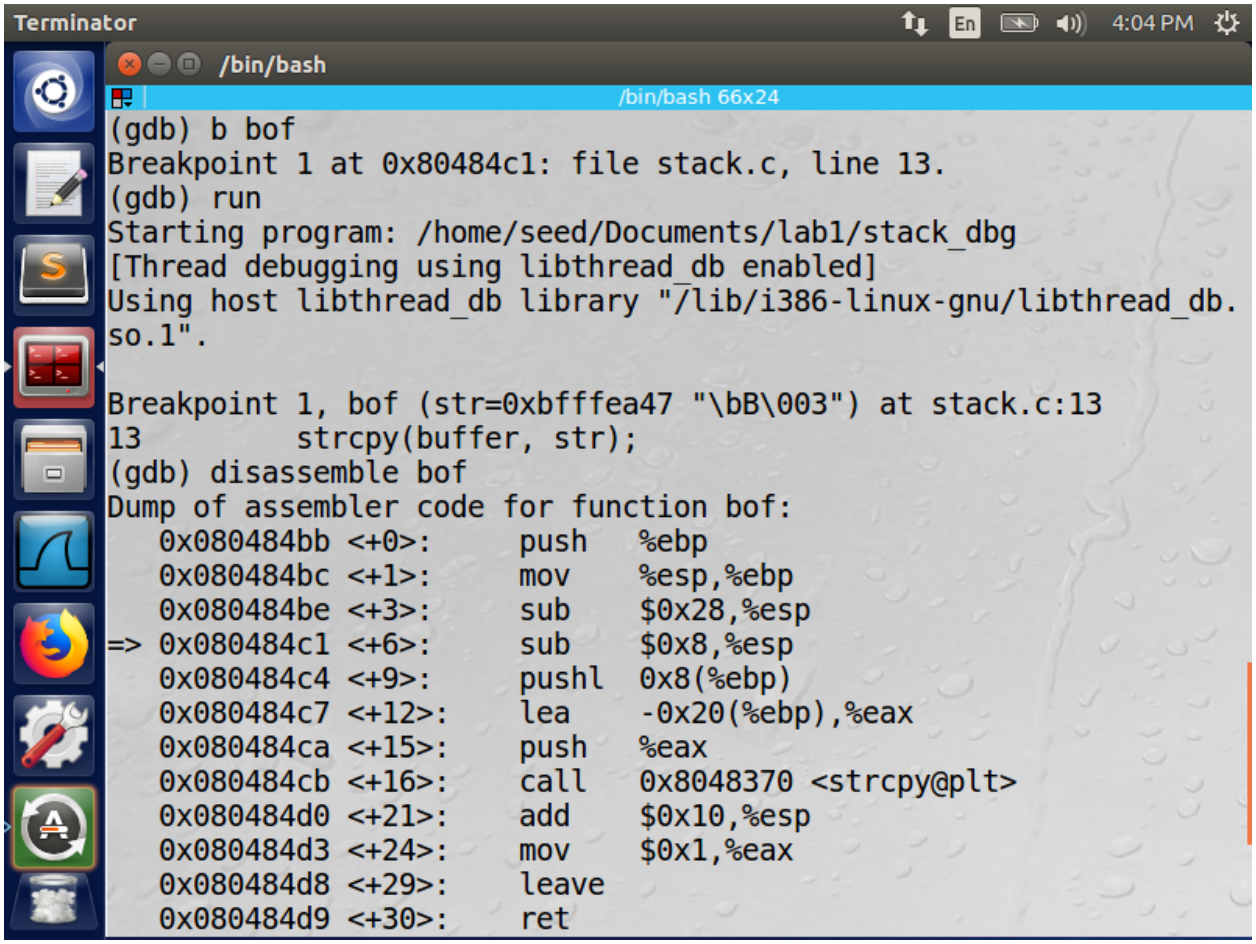

Figure 5

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.

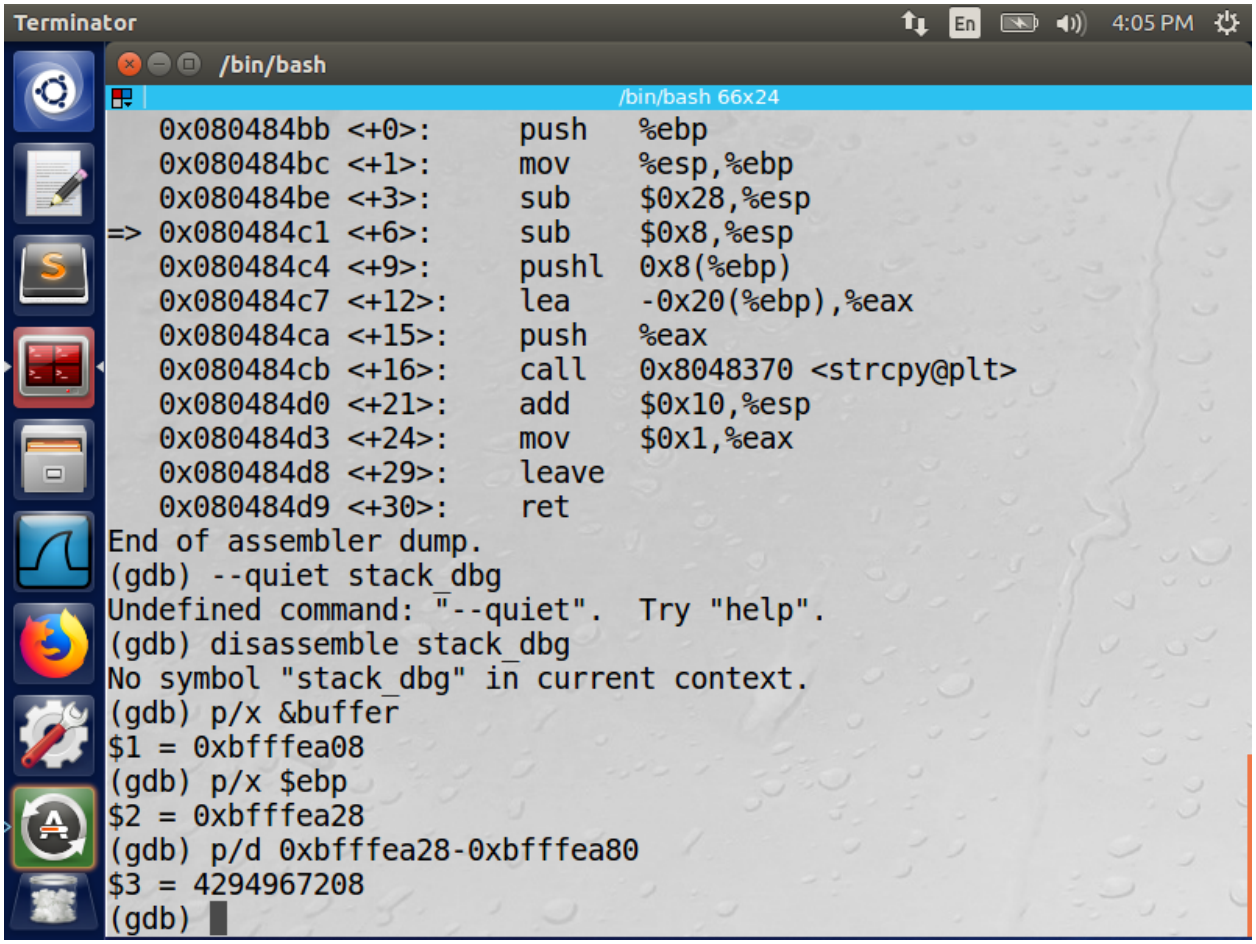

Figure 6

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 0xbfffea90. 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.

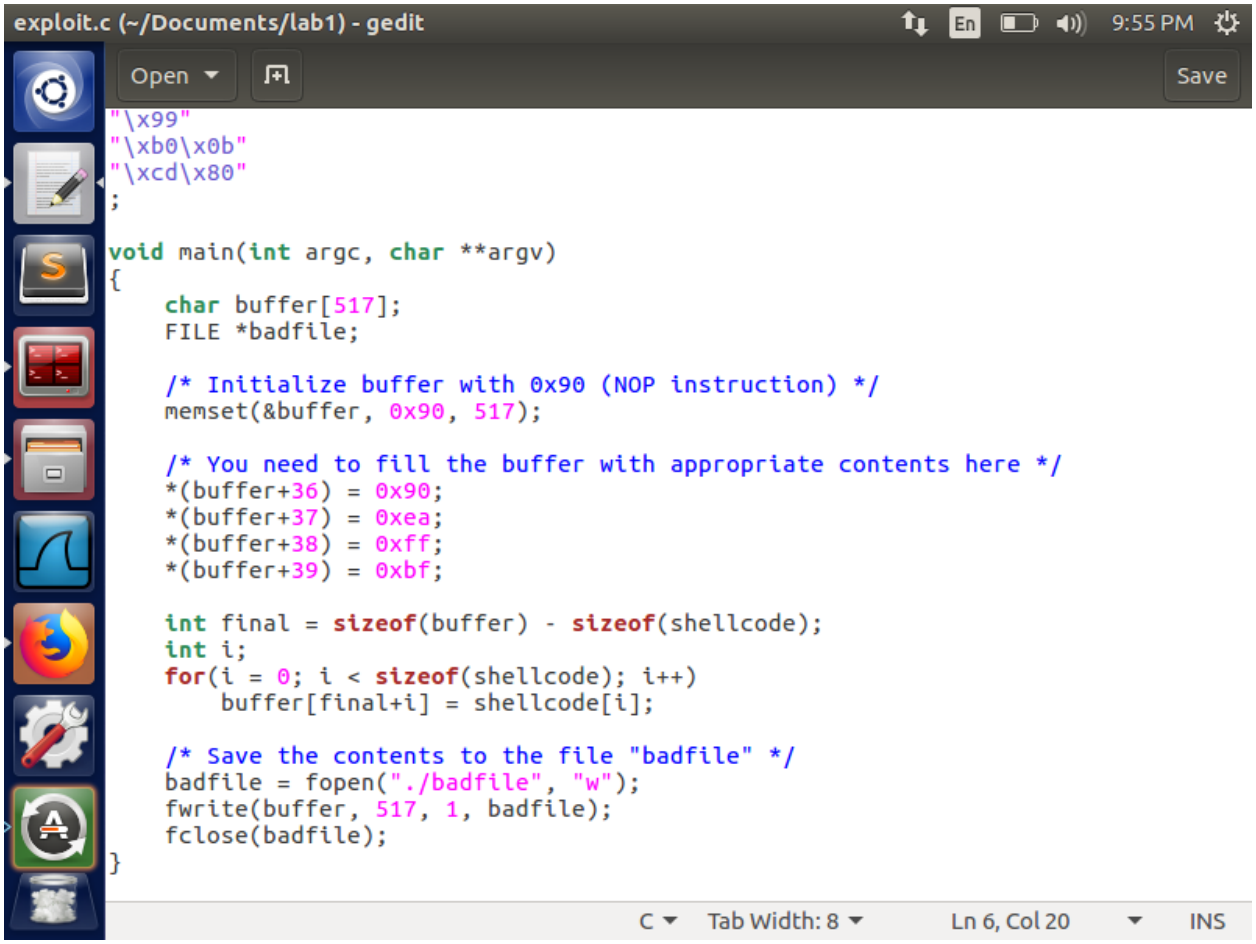

Figure 7: exploit.c

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).

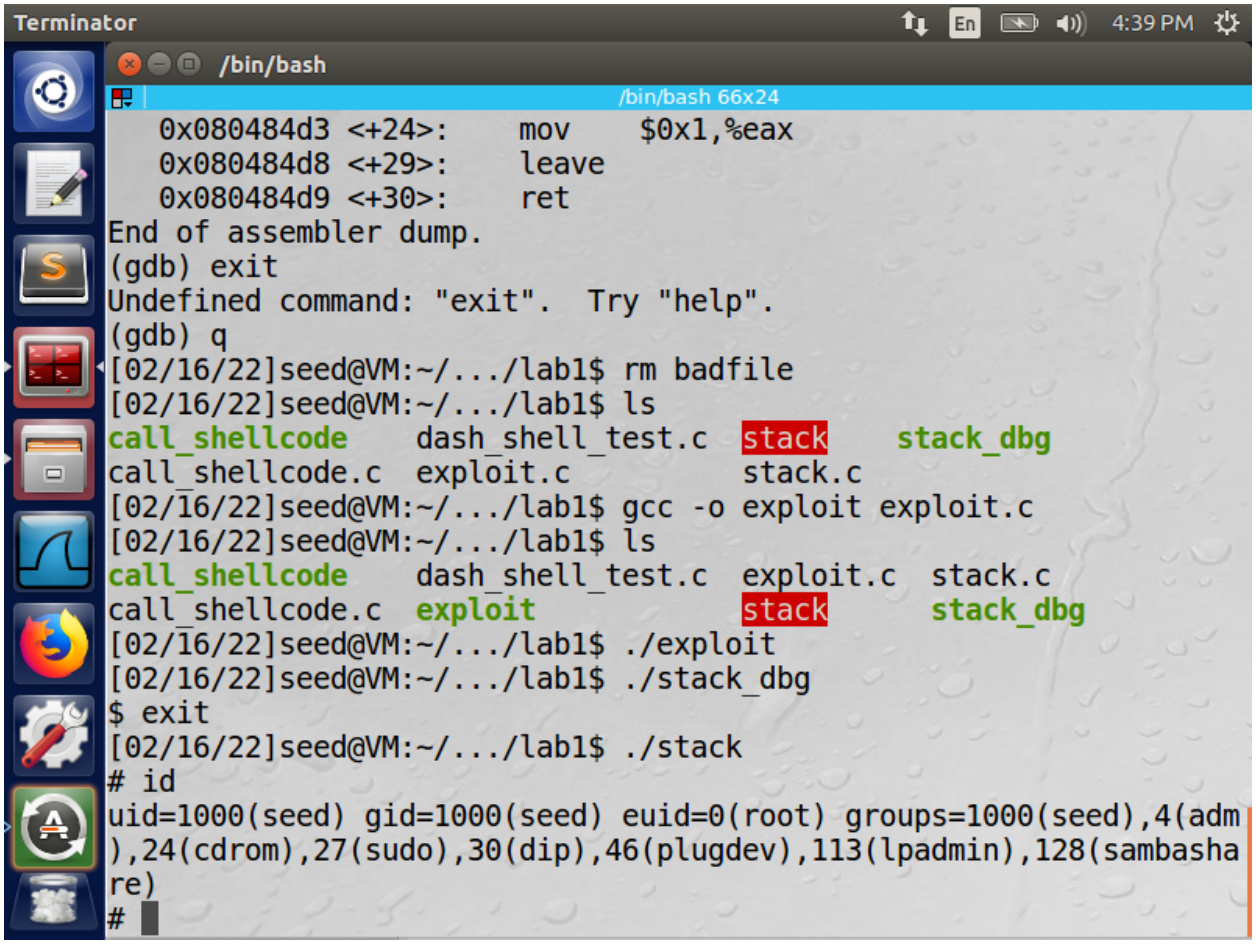
Terminator
En
4:39 PM
/bin/bash
/bin/bash 66x24
0x080484d3 <+24>: mov $0x1,%eax
0x080484d8 <+29>: leave
0x080484d9 <+30>: ret
End of assembler dump.
(gdb) exit
Undefined command: "exit". Try "help".
(gdb) q
[02/16/22]seed@VM:~/.../lab1$ rm badfile
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode dash_shell_test.c 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.

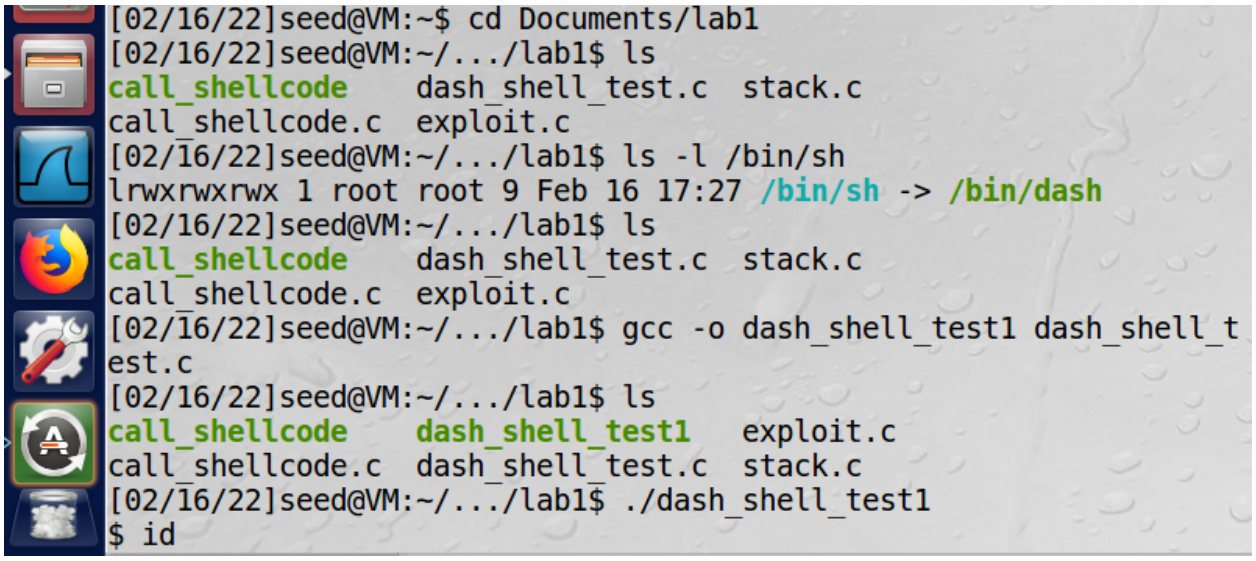

Figure 9

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_test1is 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.

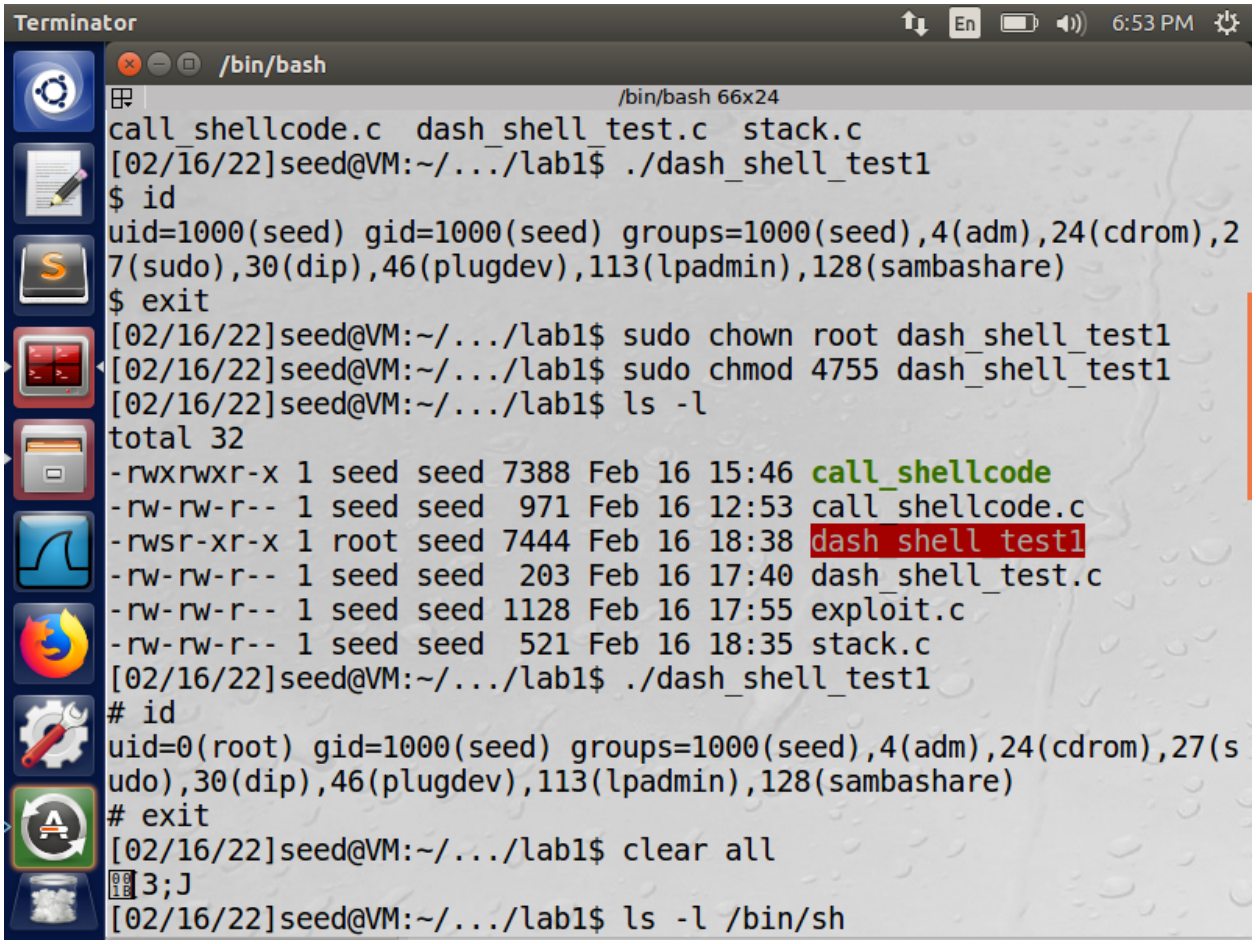

Figure 10

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.

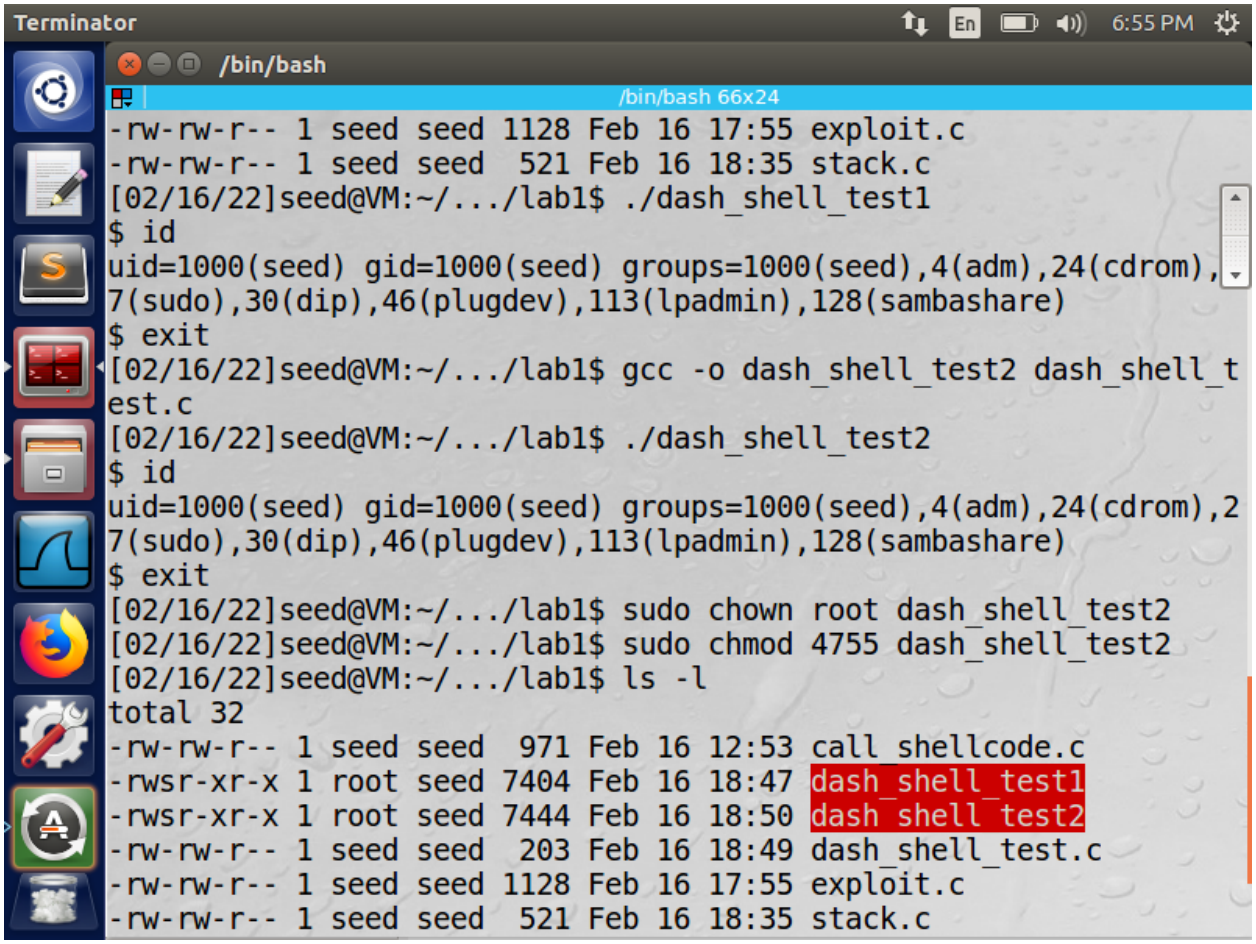

Figure 11

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.

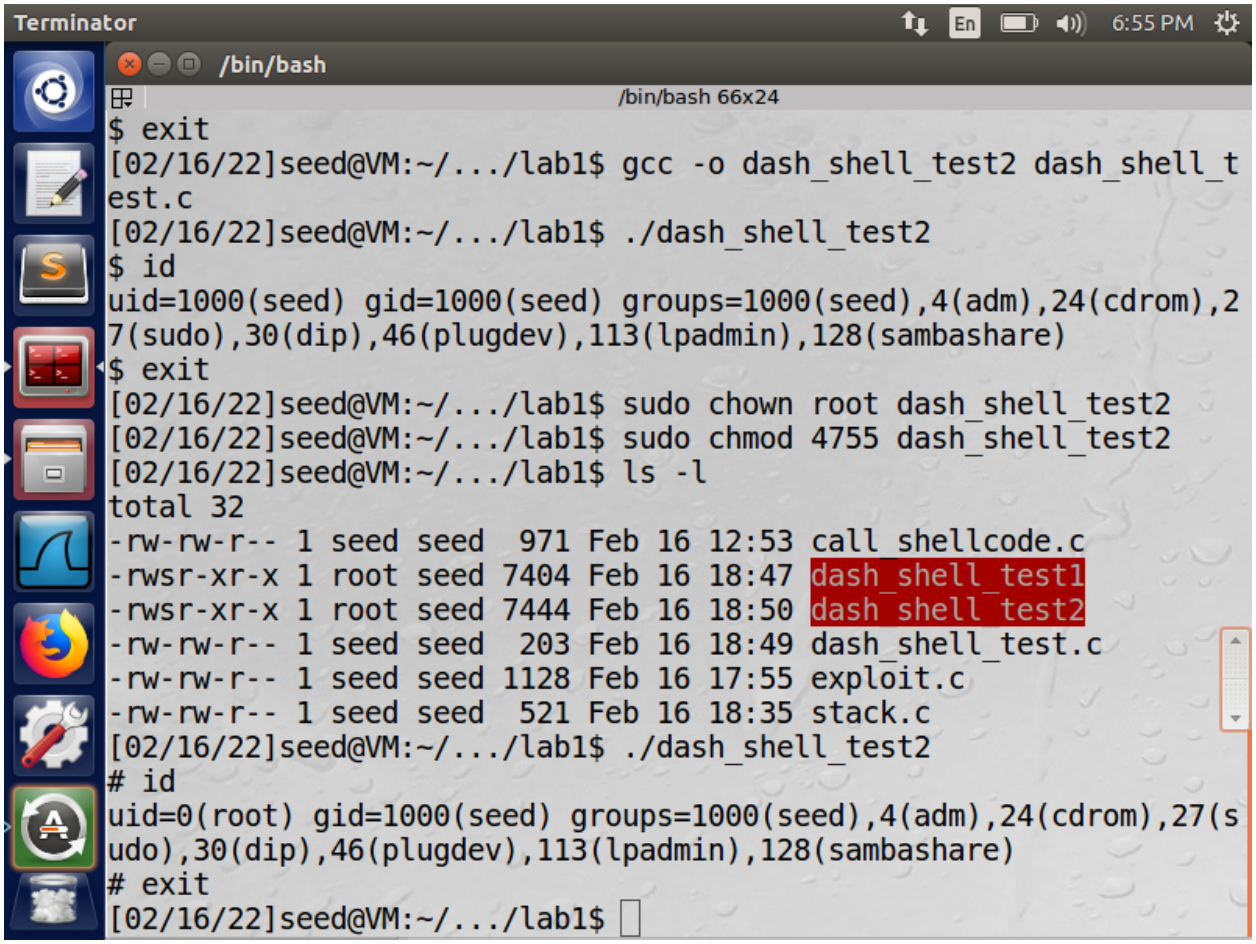

Figure 12

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.

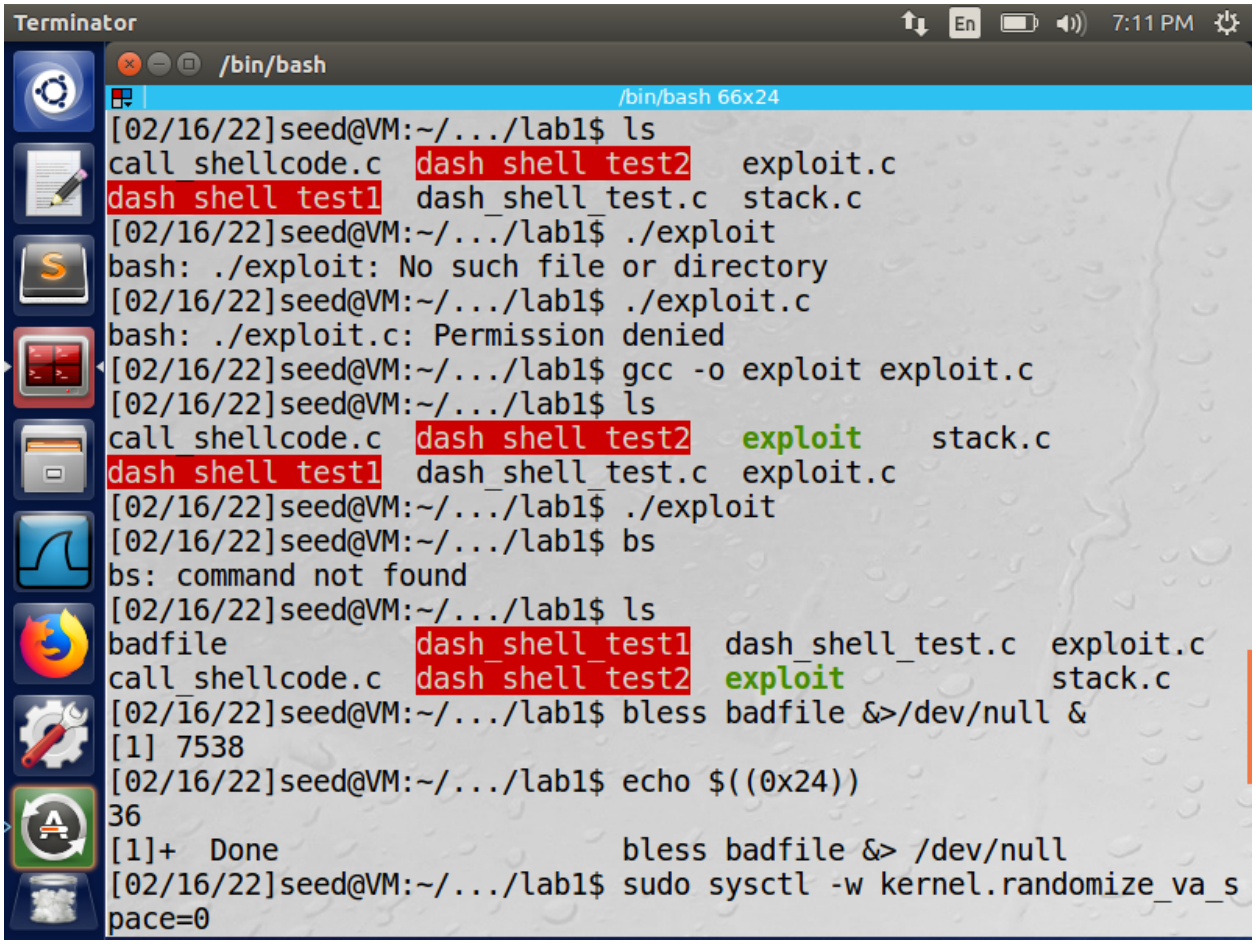
Terminator
En
7:11 PM
/bin/bash
/bin/bash 66x24
[02/16/22]seed@VM:~/.../lab1$ ls
call_shellcode.c dash_shell_test2   exploit.c
dash_shell_test1 dash_shell_test.c stack.c
[02/16/22]seed@VM:~/.../lab1$ ./exploit
bash: ./exploit: No such file or directory
[02/16/22]seed@VM:~/.../lab1$ ./exploit.c
bash: ./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_s
pace=0

Figure 13

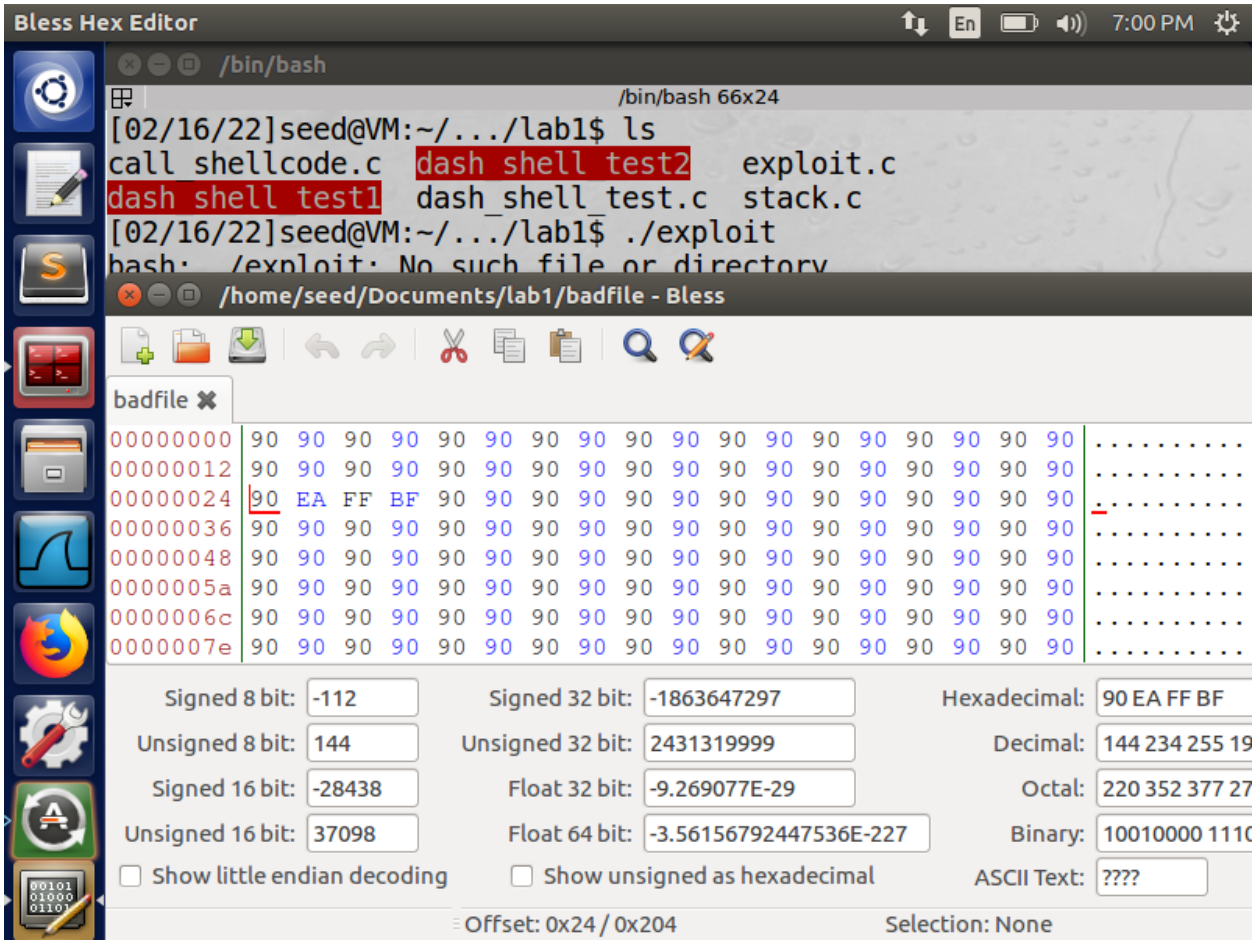

Figure 14

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

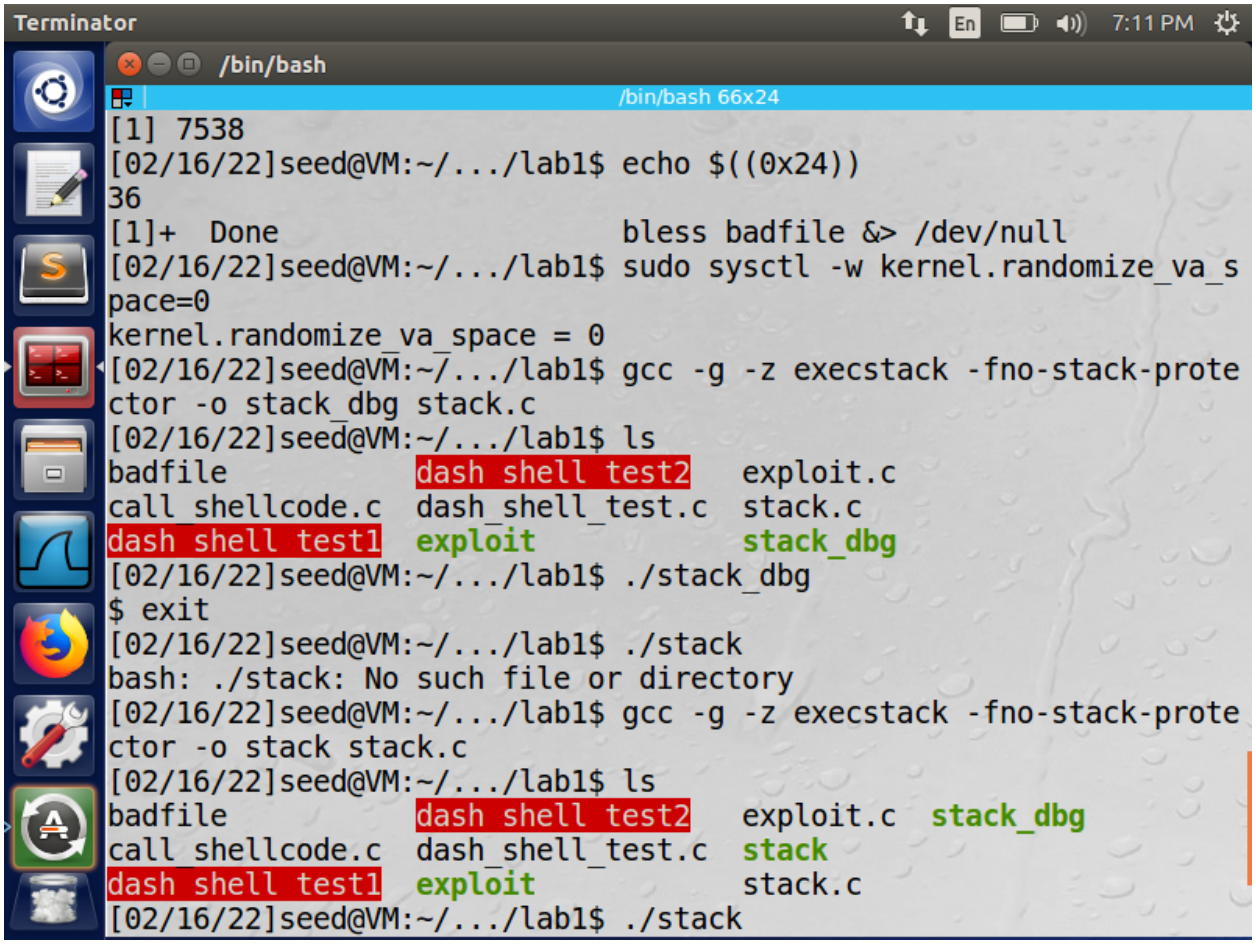

Figure 15

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

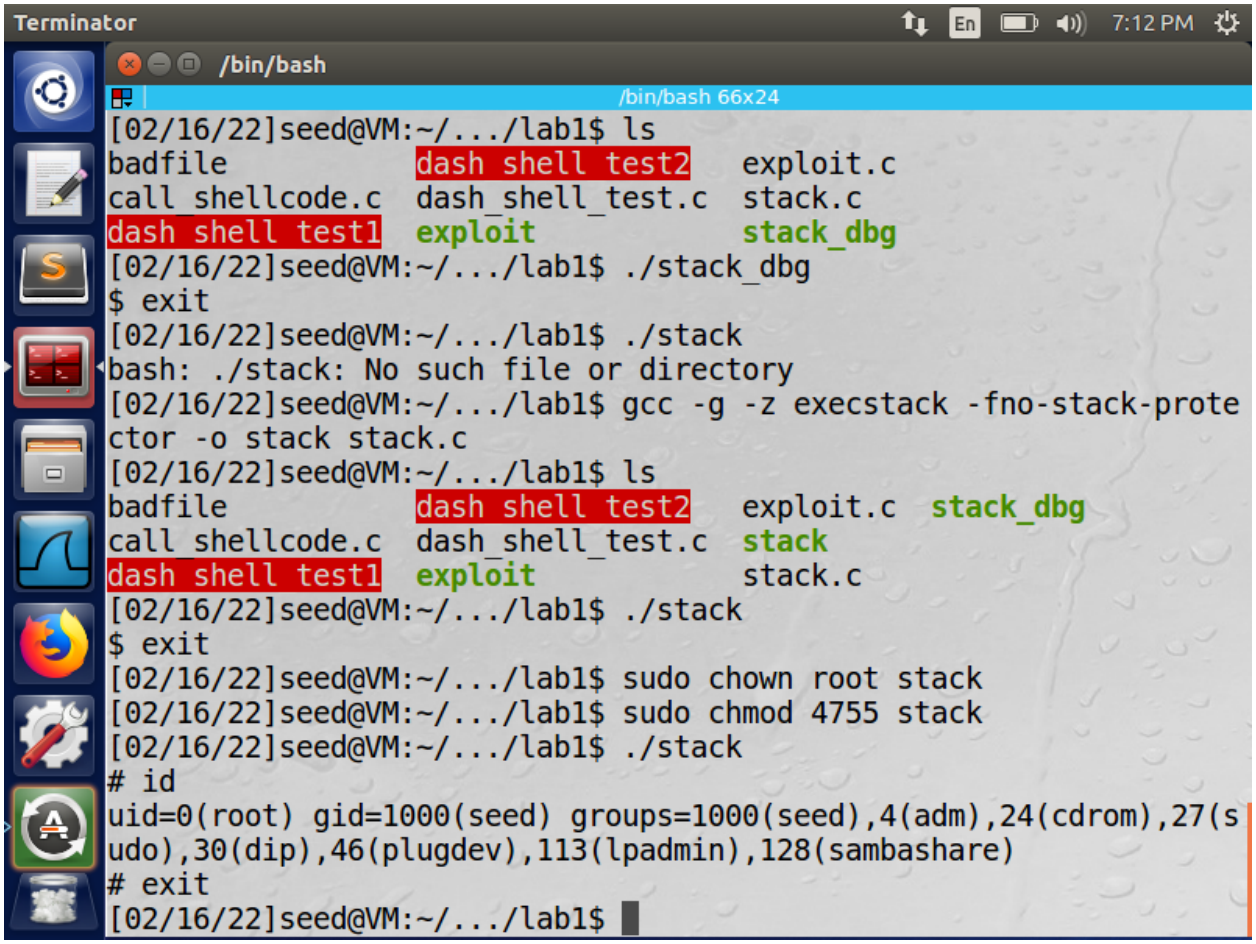

Figure 16

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.

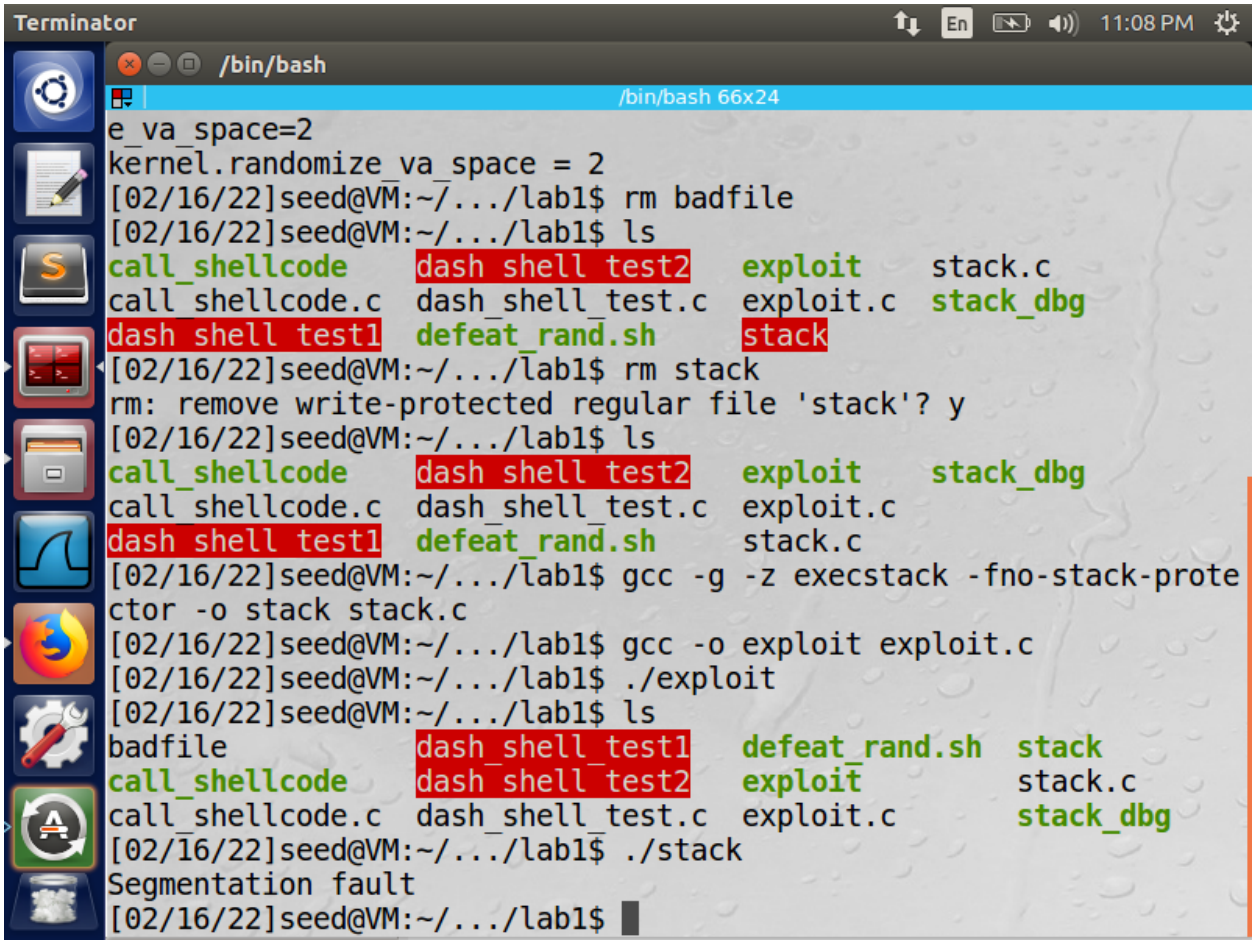

Figure 17

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.

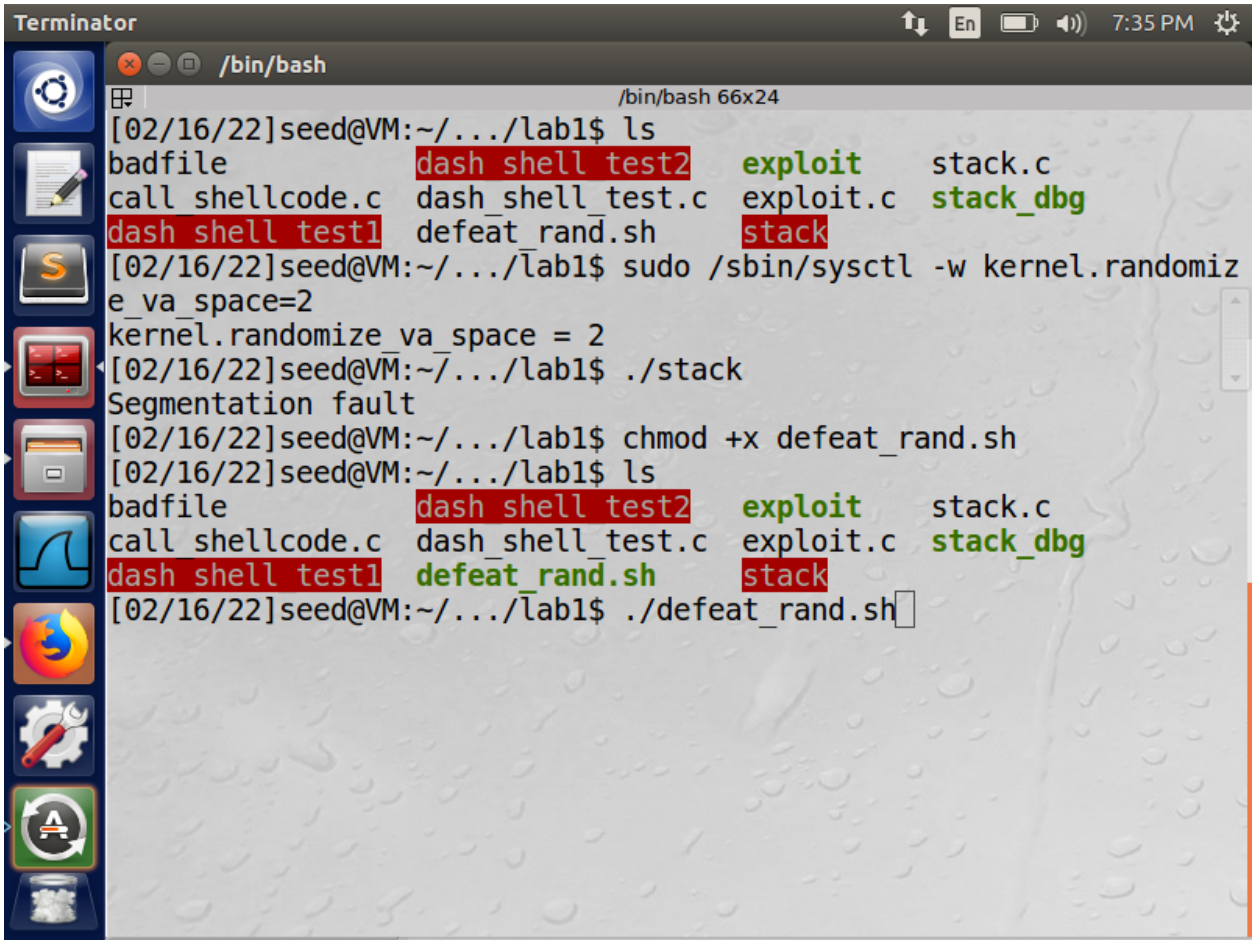
Terminator
En
7:35 PM
/bin/bash
/bin/bash 66x24
[02/16/22]seed@VM:~/.../lab1$ ls
badfile dash_shell_test2 exploit stack.c
call_shellcode.c dash_shell_test.c exploit.c stack_dbg
dash_shell_test1 defeat_rand.sh stack
[02/16/22]seed@VM:~/.../lab1$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 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

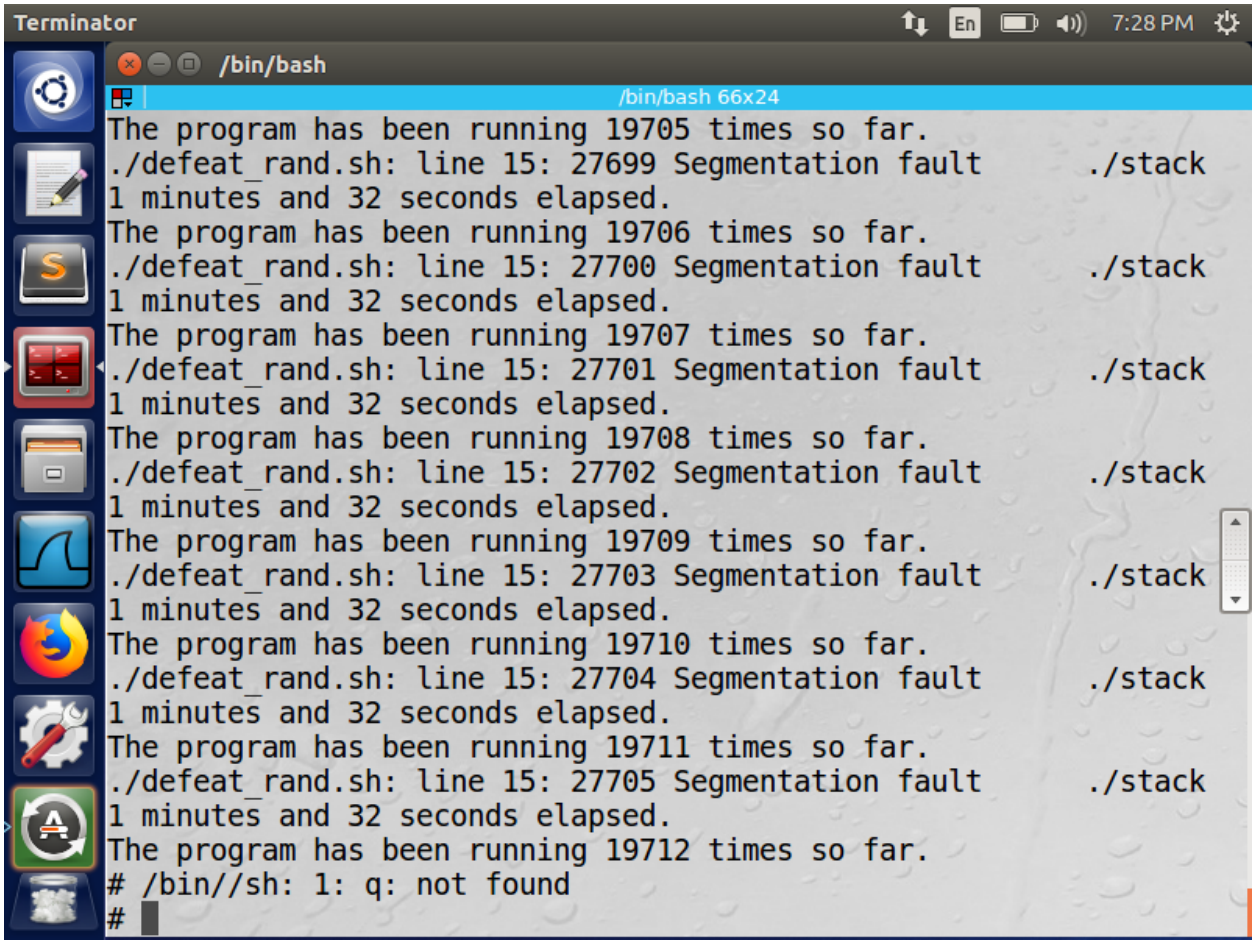
Terminator
En
7:28 PM
/bin/bash
/bin/bash 66x24
The program has been running 19705 times so far.
./defeat_rand.sh: line 15: 27699 Segmentation fault ./stack
1 minutes and 32 seconds elapsed.
The program has been running 19706 times so far.
./defeat_rand.sh: line 15: 27700 Segmentation fault ./stack
1 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