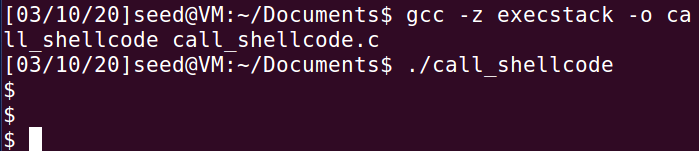
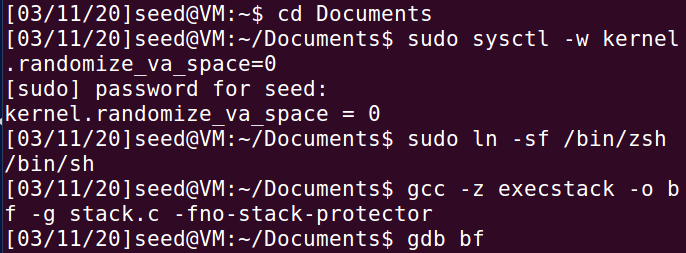
**2.2 Task 1: Running Shellcode**



The screenshot above shows that the /bin/sh is executed. The shellcode provided in the lab document invokes the execve() system call to launch a shell.

**2.4 Task 2: Exploiting the Vulnerability**

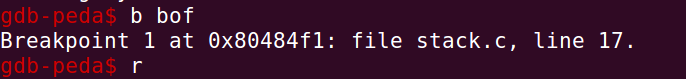


The line of command of sudo sysctl -w kernel.randomize\_va\_space=0 above demonstrates how to disable the address space randomization in the heap and stack in Linux-based system. Guessing addresses is critical in buffer-overflow attacks, which can be performed easily if the address space randomization feature is disabled.

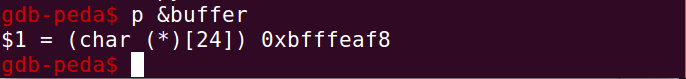
The line of command of sudo ln -sf /bin/zsh /bin/sh above links /bin/sh to zsh shell program that does not have a countermeasure that makes the buffer-overflow attacks difficult. The command is executed also because the victim program is a Set-UID program and the attack relies on running a shell program.

The stack.c program is compiled with -z execstack option to make the stack of the running program executable. It is also compiled with -fno-stack-protector option to disable the StackGuard protection, which will allow buffer overflows attack.

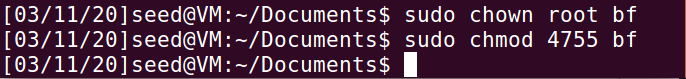
The command gdb allows user to debug the executable program and find the buffer address of the program in the terminal.



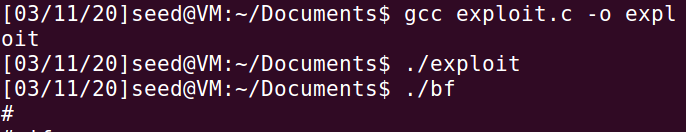
The breakpoint is set at the bof function in the stack.c program during the program debugging process.



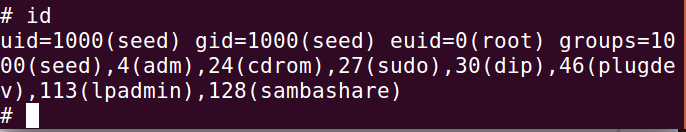
The screenshot above shows the buffer address in the bof function in the stack.c program.



The screenshot above demonstrates how to make the executable program of stack.c a root-owned Set-UID program.



The badfile is created when the exploit.c program is executed with the command ./exploit . The command ./bf executes the stack.c program and launches the buffer overflow attack. The attack is resulted in having access to the root shell.



The screenshot shows that even the user has access to the root shell, the real user id still remains the same but the effective user id is root.

﻿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 \*/

//1.overwrite the return address

long addr = 0xbfffeb78;

long \*ptr = (long \*) (buffer + 36);

\*ptr = addr;

//2.copy shellcode into buffer

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);

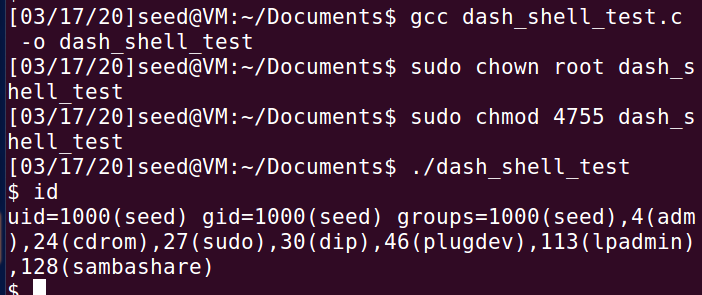
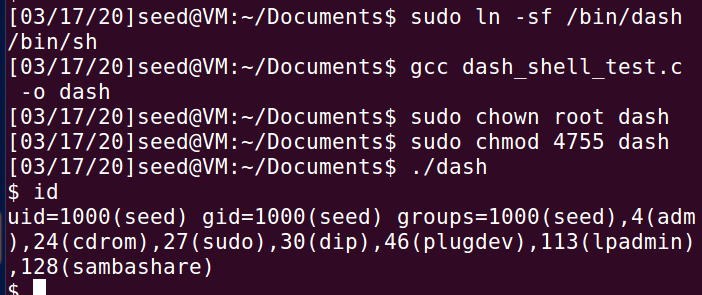
fclose(badfile);

}

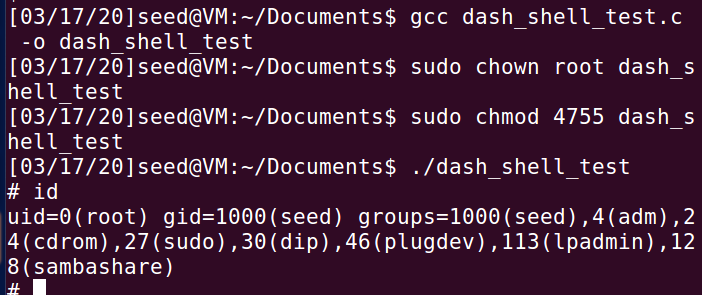
The codes above are inserted in the exploit.c program to construct contents for badfile.

The return address is overwritten by an address that is created by adding 0x80 to the buffer address (0xbffeaf8). The shellcode is also copied into buffer by using the memcpy function.

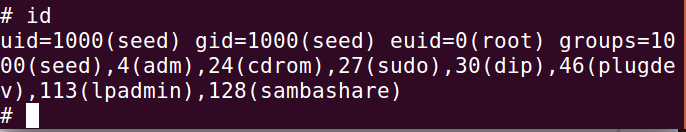
**2.5 Task 3: Defeating dash’s Countermeasure**



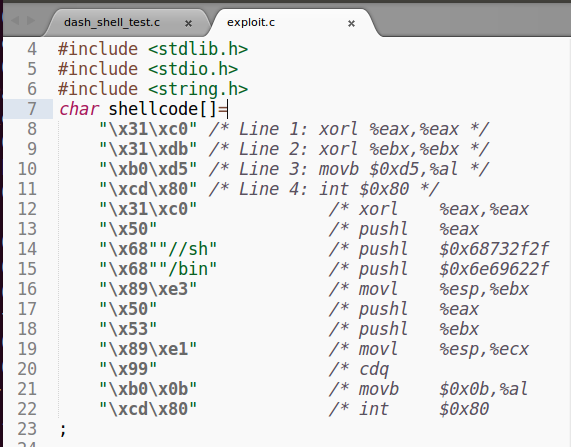
The screenshot above shows the result when the line of code of setuid(0) is commented out. The uid is 1000 and the shell is a non-root user shell.



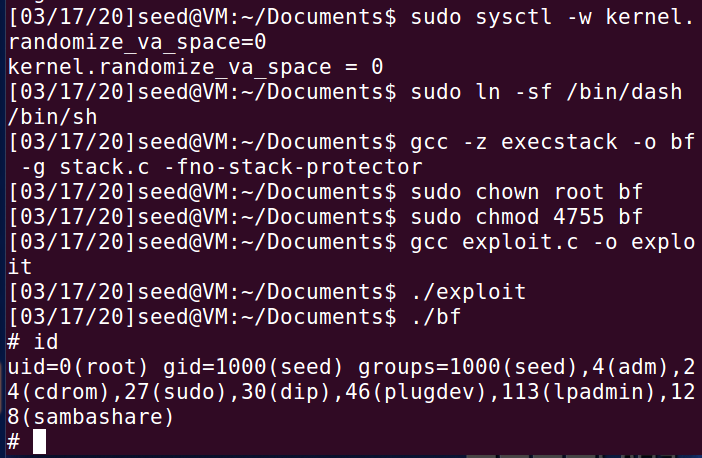
The screenshot above shows the result when the line of code of setuid(0) is uncommented. The uid is no longer 1000 and it is 0 instead. The shell is a root shell. This is because the uid is set to 0 before invoking the dash program.



The screenshot demonstrates the result of Task 2 before the shellcode is updated in exploit.c. The root shell is obtained but the uid is 1000.

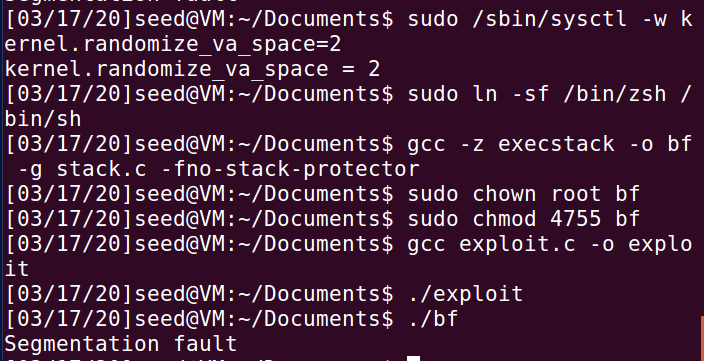


The shellcode in exploit.c is updated with 4 additional instructions.

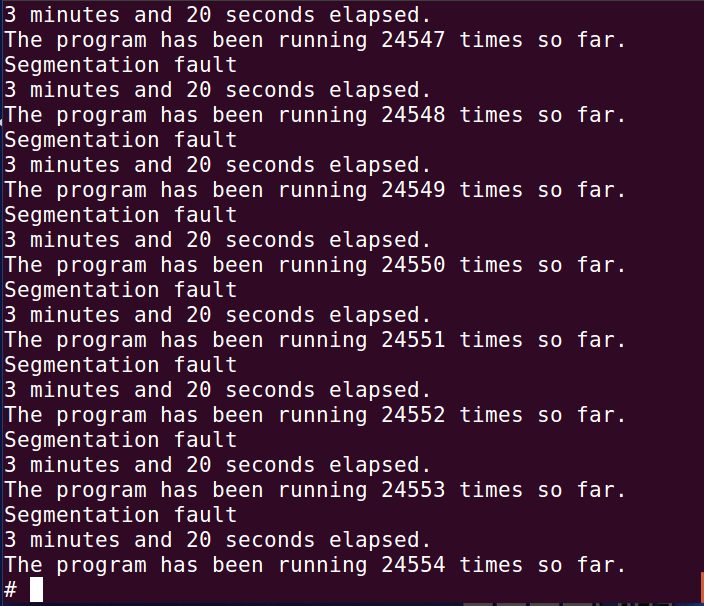


The /bin/sh is linked to /bin/dash using the command sudo ln -sf /bin/dash /bin/sh . The screenshot indicates that the root shell is obtained after the attack from Task 2 is launched. However, the uid is changed from 1000 (seed) to 0 (root). This is because the uid is set to 0 by the updated shell code in exploit.c.

**2.6 Task 4: Defeating Address Randomization**

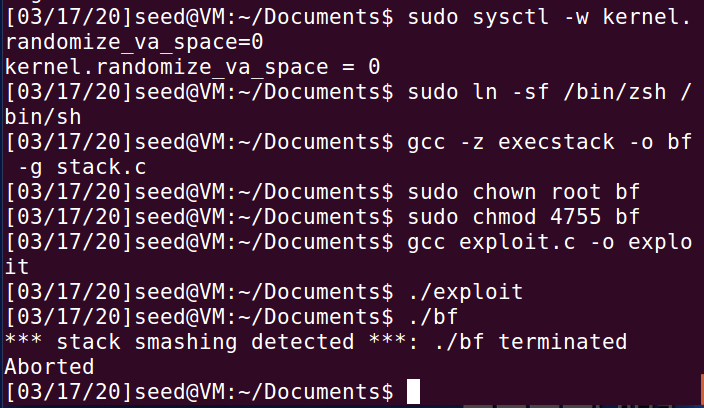
****

The attack developed in Task 2 is not launched successfully because the Ubuntu’s address randomization feature is turned on. Therefore, the address needed to perform the attack in Task 2 is no longer accurate due to the address randomization.



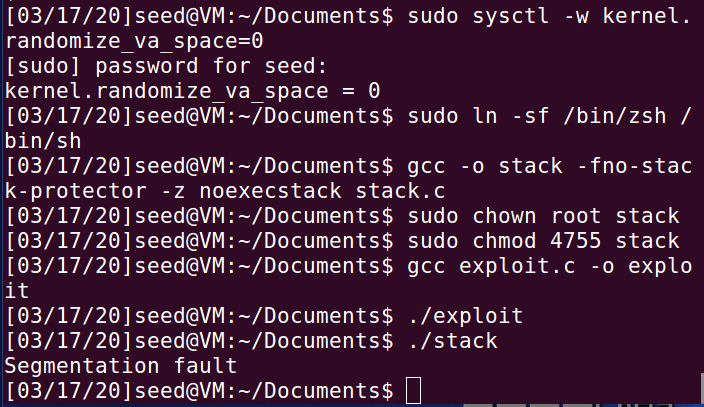
The brute-force approach is used to attack the vulnerable program in Task 2 repeatedly to make sure that the address in badfile is correct. The shell script provided in the lab document allows us to run the vulnerable program in an infinite loop until the attack in Task 2 succeeds. The screenshot shows that it took me 3 minutes and 20 seconds to ensure that the address in badfile is correct in order to launch the attack in Task 2 successfully.

**2.7 Task 5: Turn on the StackGuard Protection**



The address randomization feature is disabled and the StackGuard protection mechanism in GCC when compiling the vulnerable program in Task 2 is enabled. The attack fails to launch due to the enabled StackGuard protection feature in GCC. The error message shown in the screenshot above indicates that error detected while executing the attack in Task 2 with the StackGuard protection feature being turned on.

**2.8 Task 6: Turn on the Non-executable Stack Protection**



The screenshot illustrates that the program in Task 2 fails to launch a successful attack due to non-executable stack protection as the error of segmentation fault occurs.