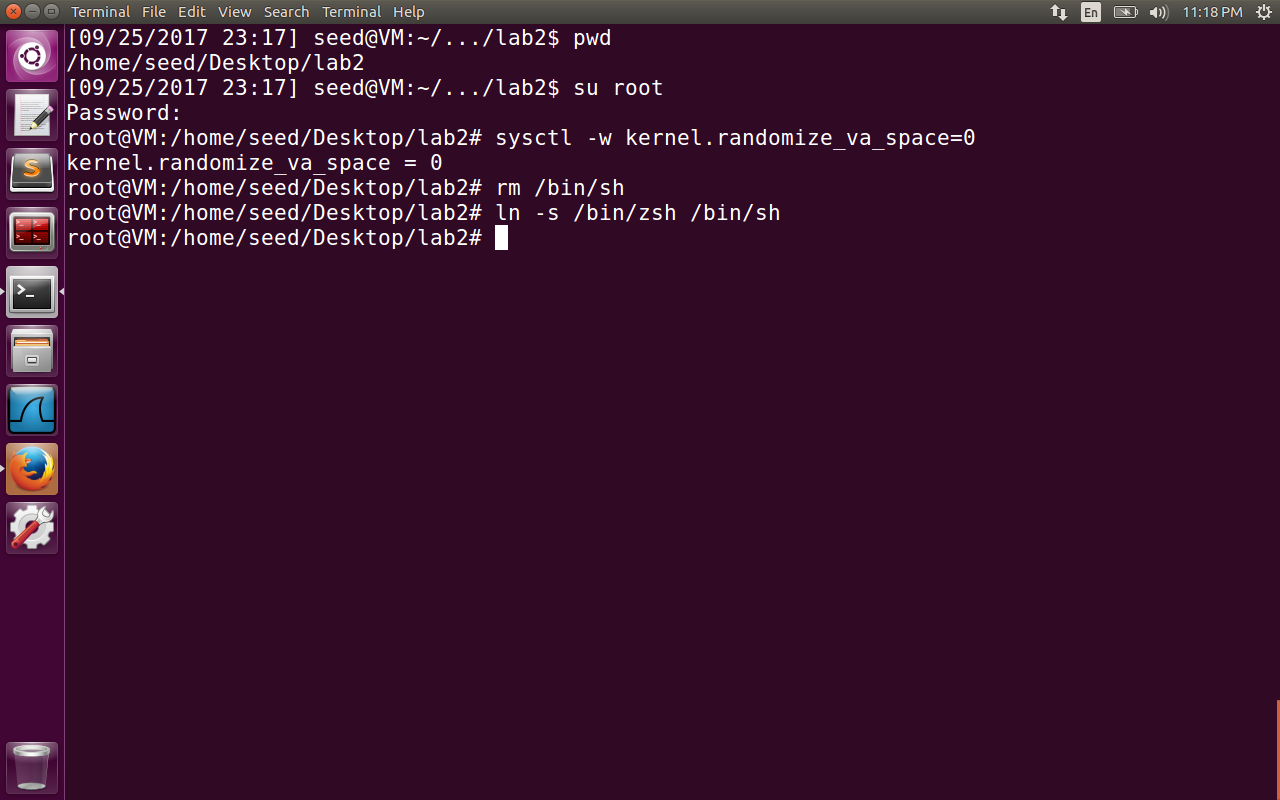
Computer Security- Lab2

Buffer Overflow Attack

NAME: JASHWANTH REDDY GANGULA

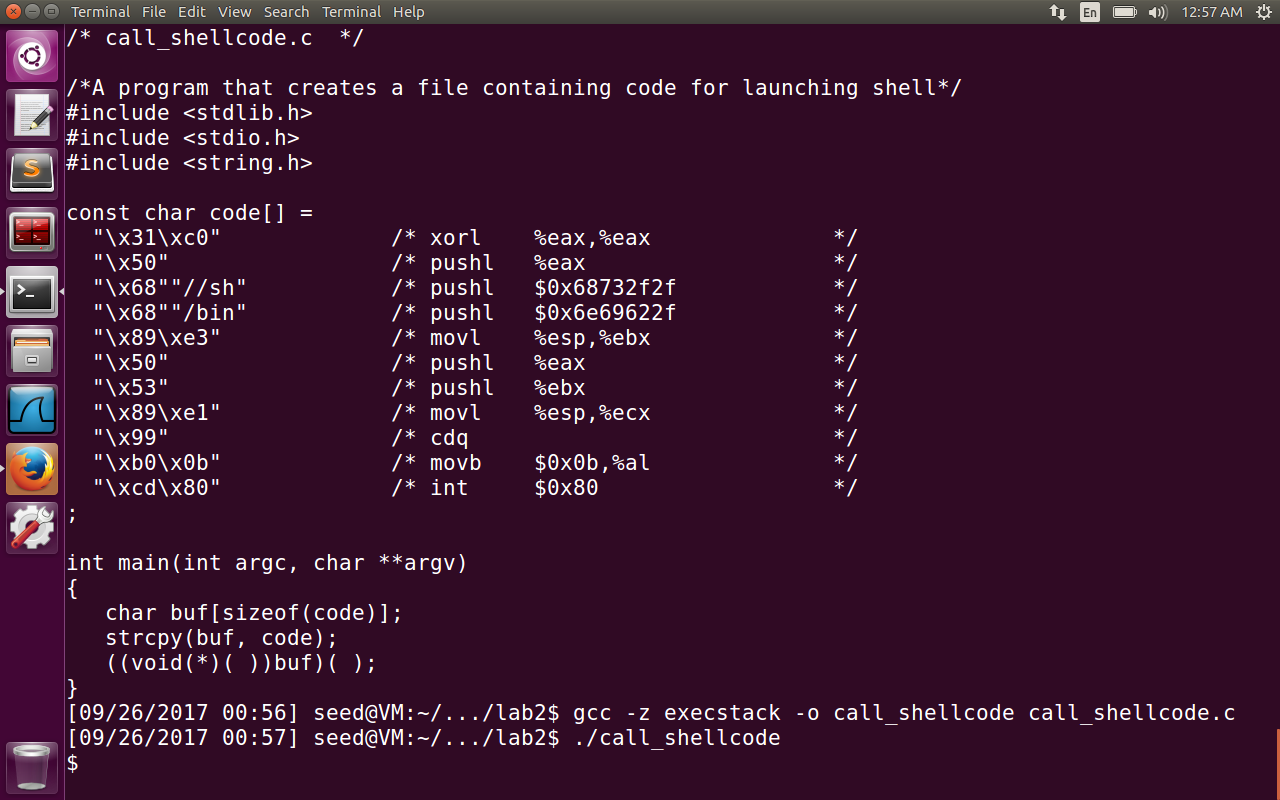
SUID: 646254141

**2.1 Initial Setup**



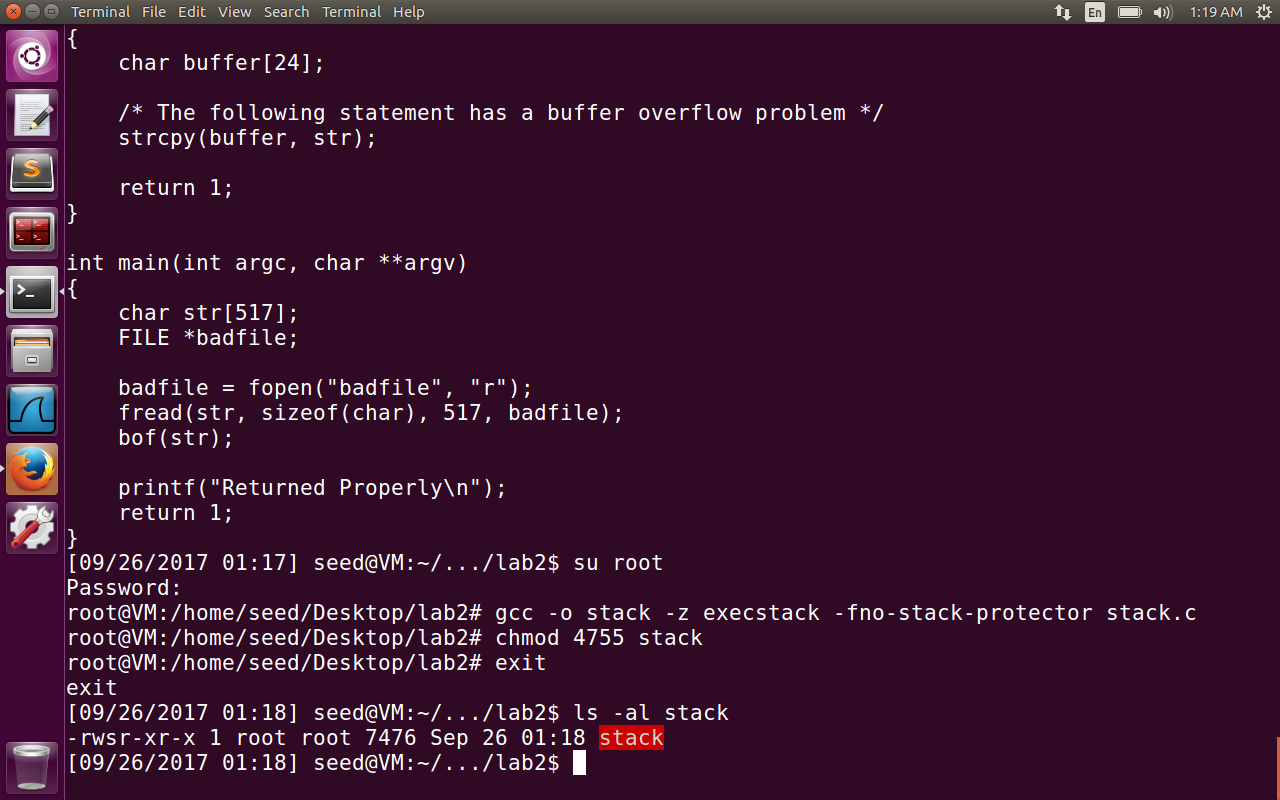
In the initial lab setup, we have disabled the address randomization, because guessing the address is one of the key steps of the buffer overflow attack. Ubuntu makes the stacks non executable by default. In future to make the stacks executable we need to execute the command gcc –z execstack -o somefile somefile.c . Finally /bin/sh is linked to /bin/zsh because dash shell is modified to drop set-uid privilege.

**2.2 Shell Code to launch a shell.**

****

The program call\_shellcode is compiled by making it an executable stack. The shell is launched by executing shellcode stored in a buffer.

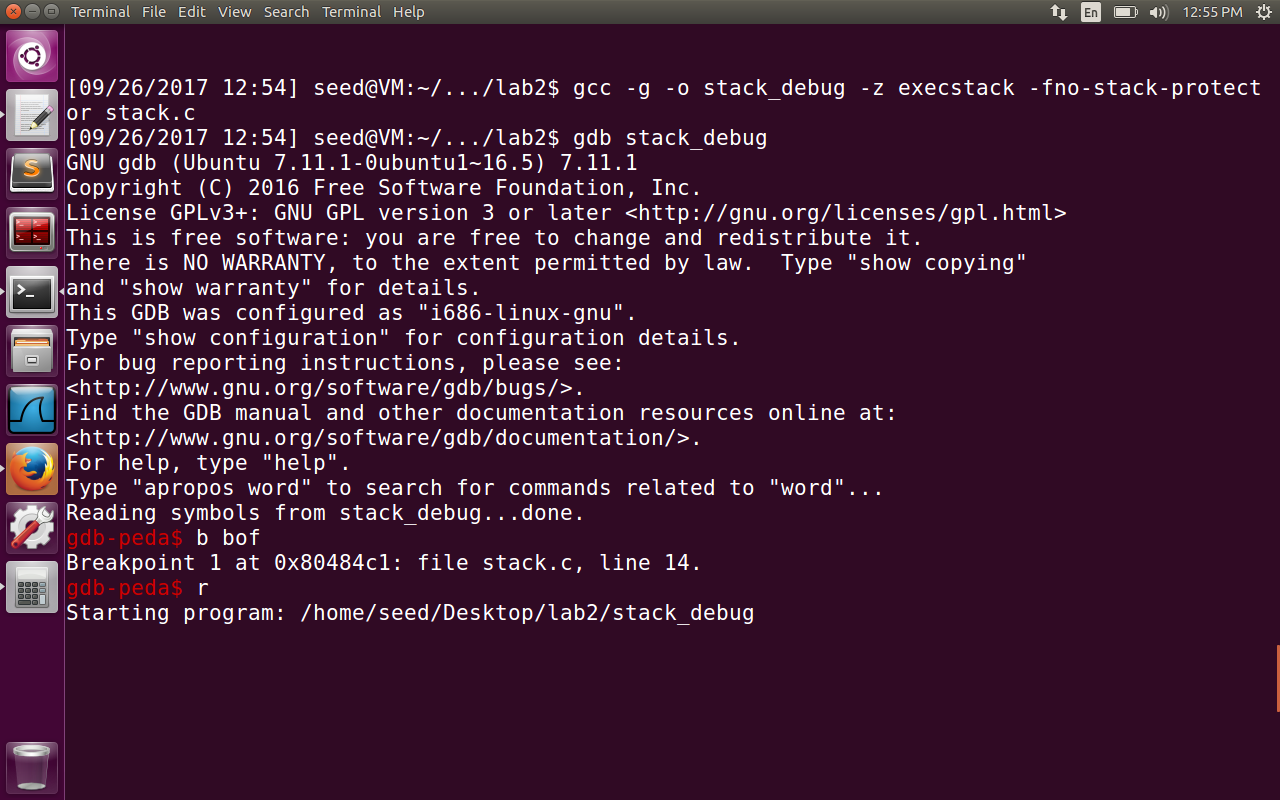
**2.3: The Vulnerable Program**

****

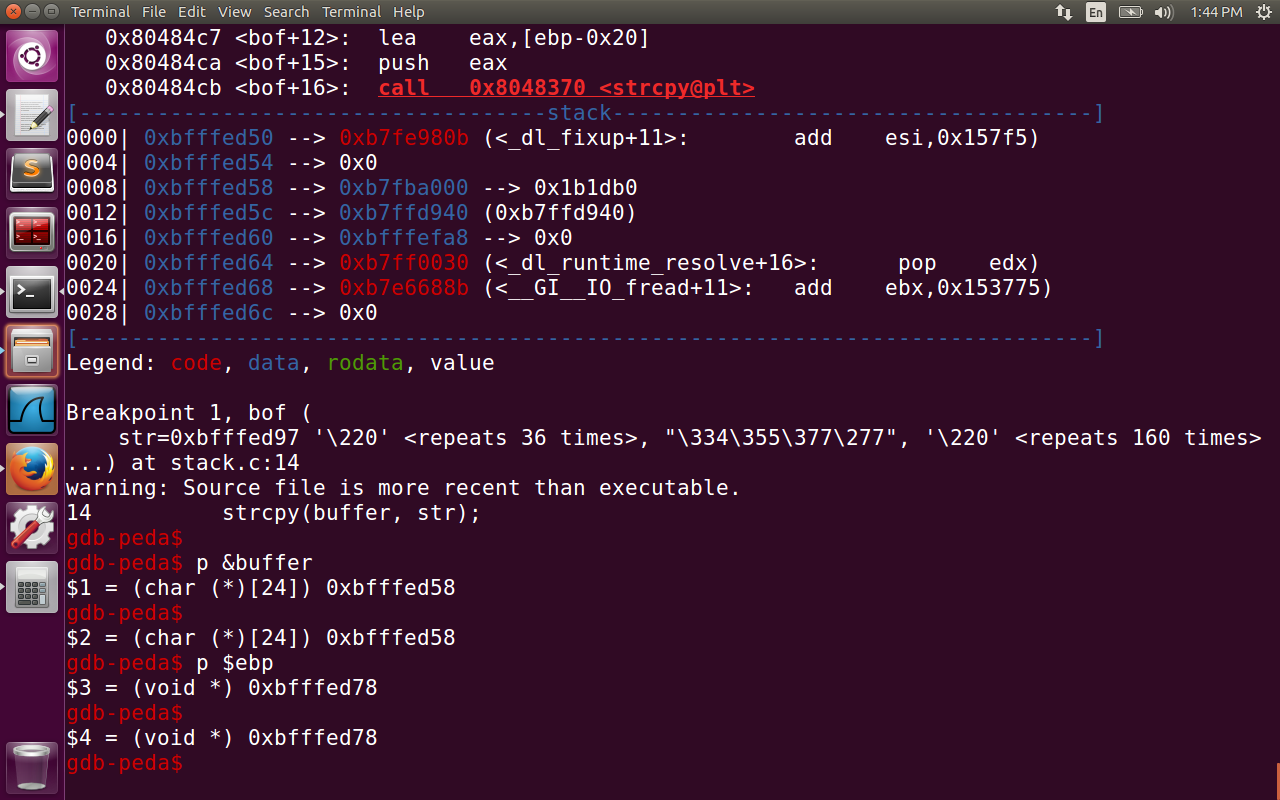
The stack.c has buffer overflow program. The buffer size is 24, but the size of badfile is 517 bytes. Strcpy has no boundary checking, hence the error. The program is compiled with root privileges with removal of stack guard protection and turning off non-executable stack options.

The program is also made a set-uid program. The normal user can affect this program by editing the contents of badfile and execute the malicious code.

**Task 1: Exploiting the Vulnerability**

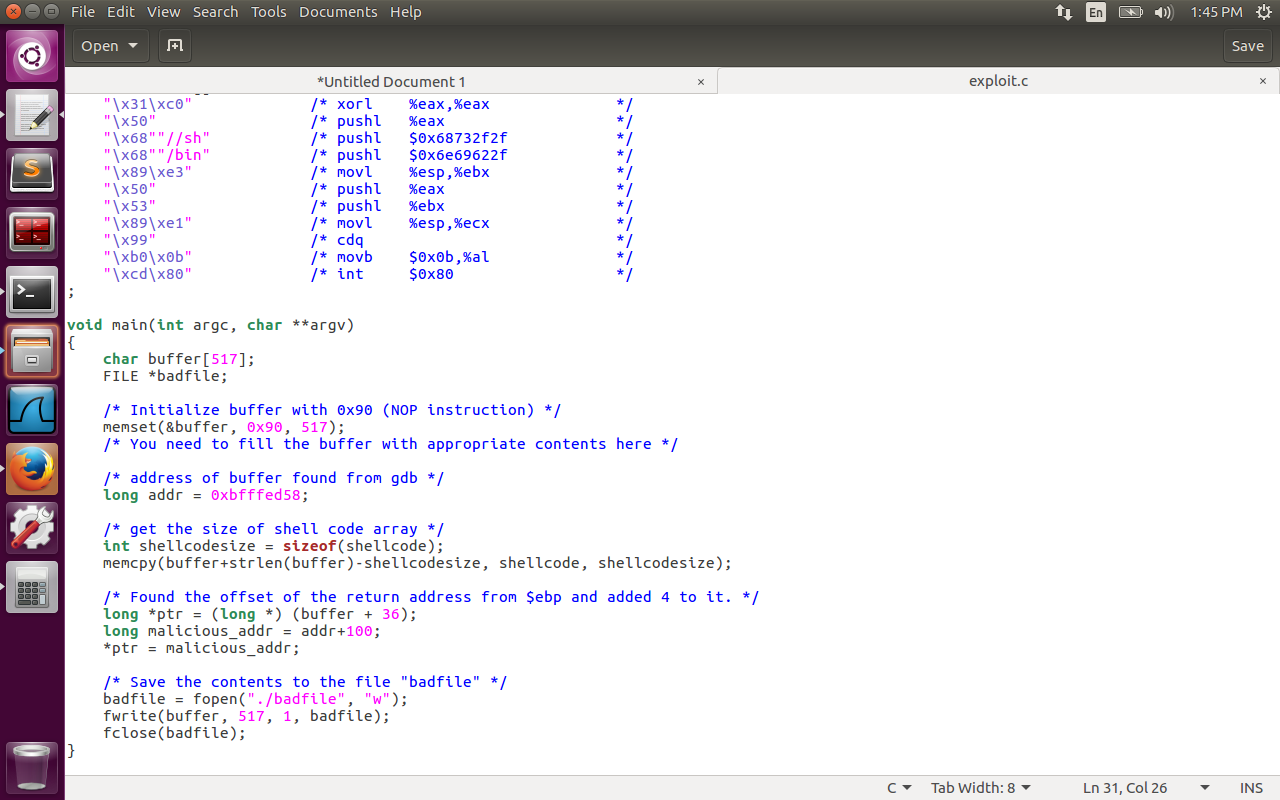
****

First the stack program is compiled with –g option to make it a debuggable program in gdb. The executable stack\_debug is created with normal user privileges. The breakpoint is inserted at function bof as shown in the image above.

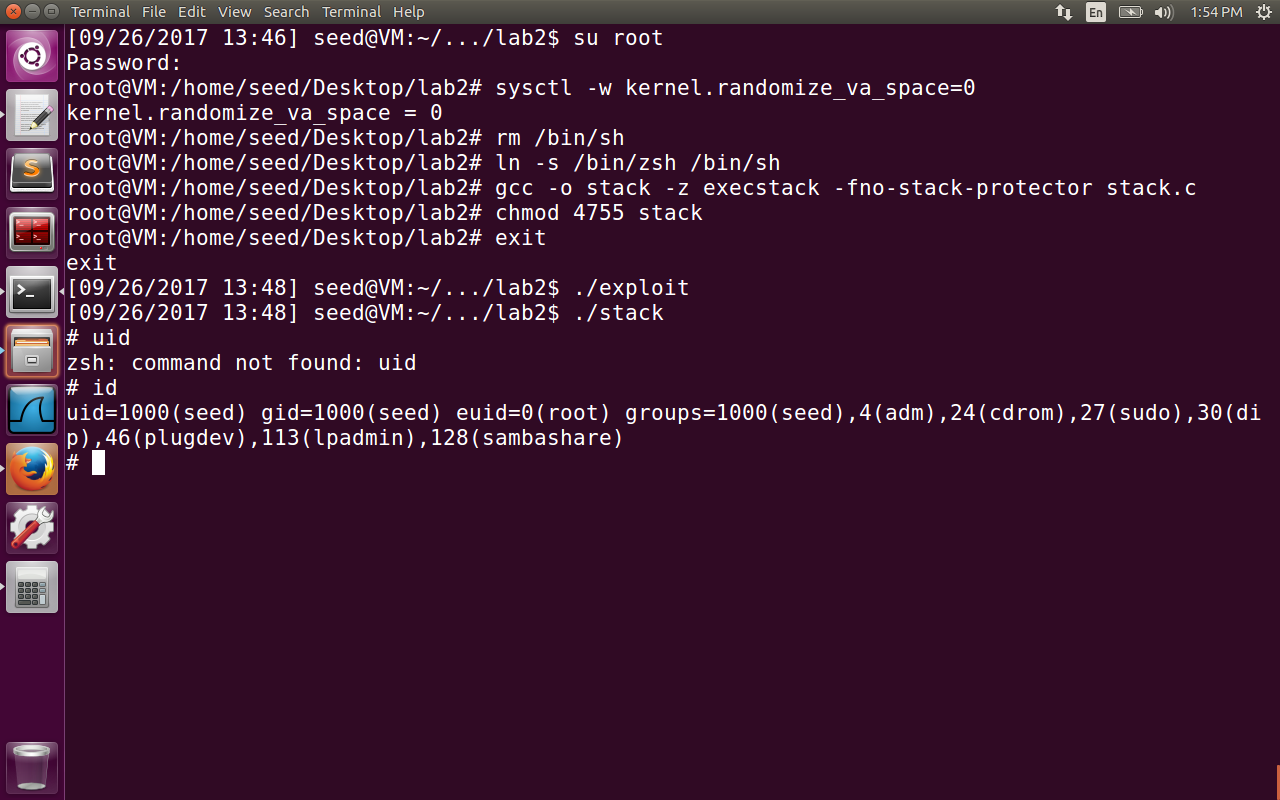


Now the objective is to get the address of buffer variable and also the variable ebp. The difference between the addresses of ebp and buffer = 0xbfffed78 – 0xbfffed58 = 0x20 = 32

So now we know that the return addresses is at the offset of 32+4 = 36 bytes from the buffer address. If we inject the malicious code in this address, we can exploit the vulnerability.



The exploit.c program is changed with the following logic. First the address of buffer is stored in the addr variable. memcpy is used to copy the shellcode content to the end of buffer string. The return address is at 36 offset from buffer[0] address. The malicious code addresses is put at the return address. The offset of 100 is added to buffer variable address found from gdb. The entire buffer is filled with NOP instructions. Even if we jump to some section of NOP instructions, eventually the malicious code is executed. Hence 100 is added roughly with this estimate.



First we logged in to the root account. We disabled the address randomization of heap and stack, because guessing the buffer address is one of the important steps of buffer overflow attack. Next we compile the stack.c program by enabling it as a executable stack and disabling stack guard protection. The stack.c is compiled by root and made it a set-uid program.

The exploit program writes the malicious code content to the badfile. By executing the stack program , we have access to root shell. But if we observe, real user id is still seed(1000), the effective user id is root(0). We can run the following program

void main() {

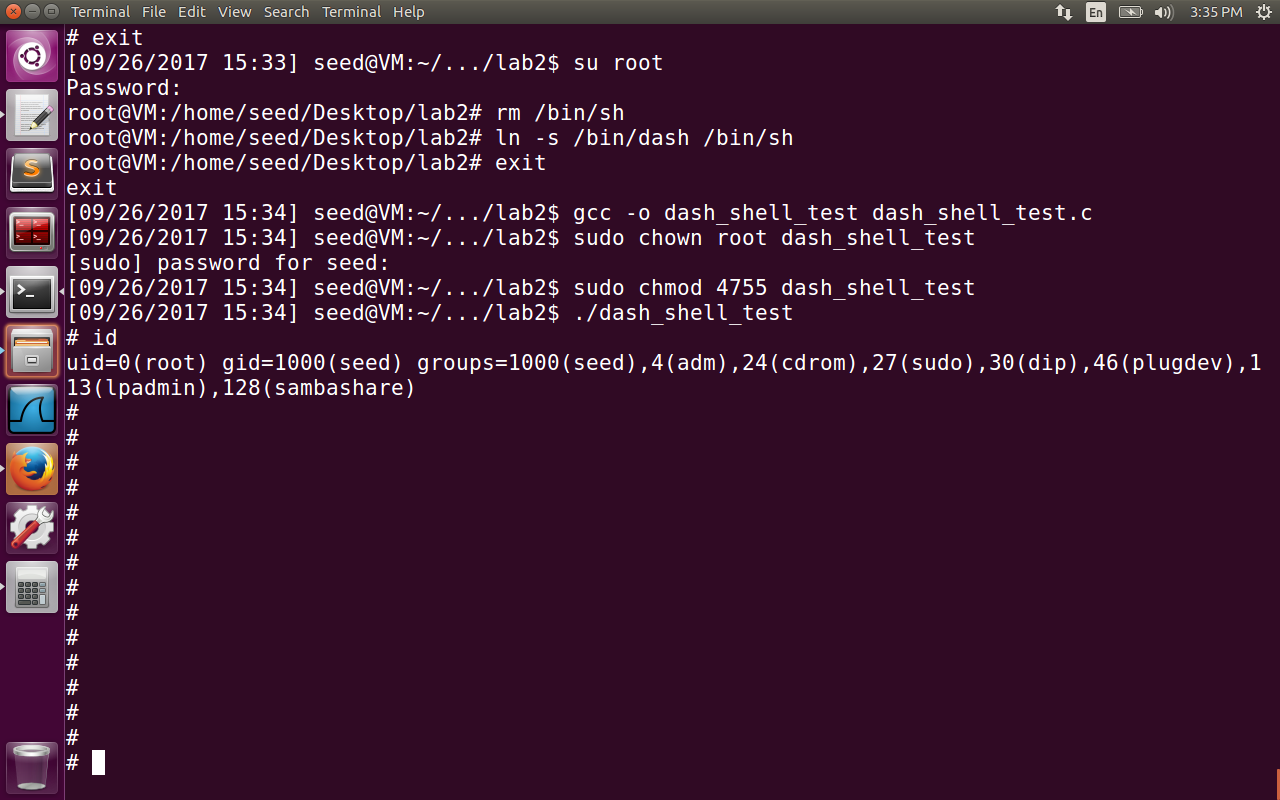
setuid(0);

system("/bin/sh");

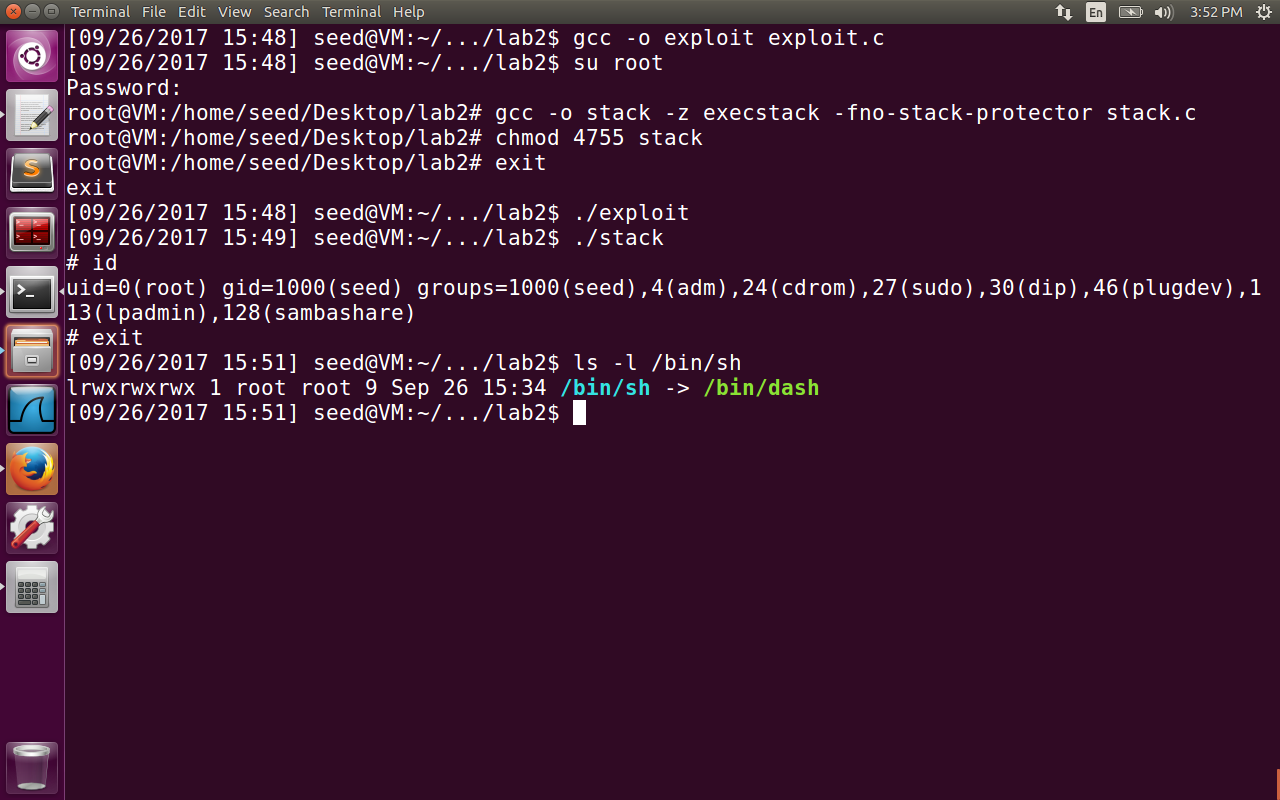
}

The shell we acquire now is more powerful because both real user id and effective id are root user.

**2.5 Task 2: Overcoming Dash Shell Privilege Drop**

****

The dash\_shell\_test program has setuid(0) inside the program. Hence after making the program a set-uid program, the uid is shown as root. So now the shell is linked to /bin/dash.



In the figure above we observed that uid = root(0). This is because we have changed the shell\_code to execute setuid(0) before executing execv(“/bin/sh”). The /bin/sh is now pointing to /bin/dash. The shell we get after setting userid to zero is more powerful, because the dash shell is known to drop privileges if it observes that real user id and effective user id are different. The problem is solved by adding the code to set uid to zero in exploit.c program.

char shellcode[] = "\x31\xc0" /\* Line 1: xorl %eax,%eax \*/

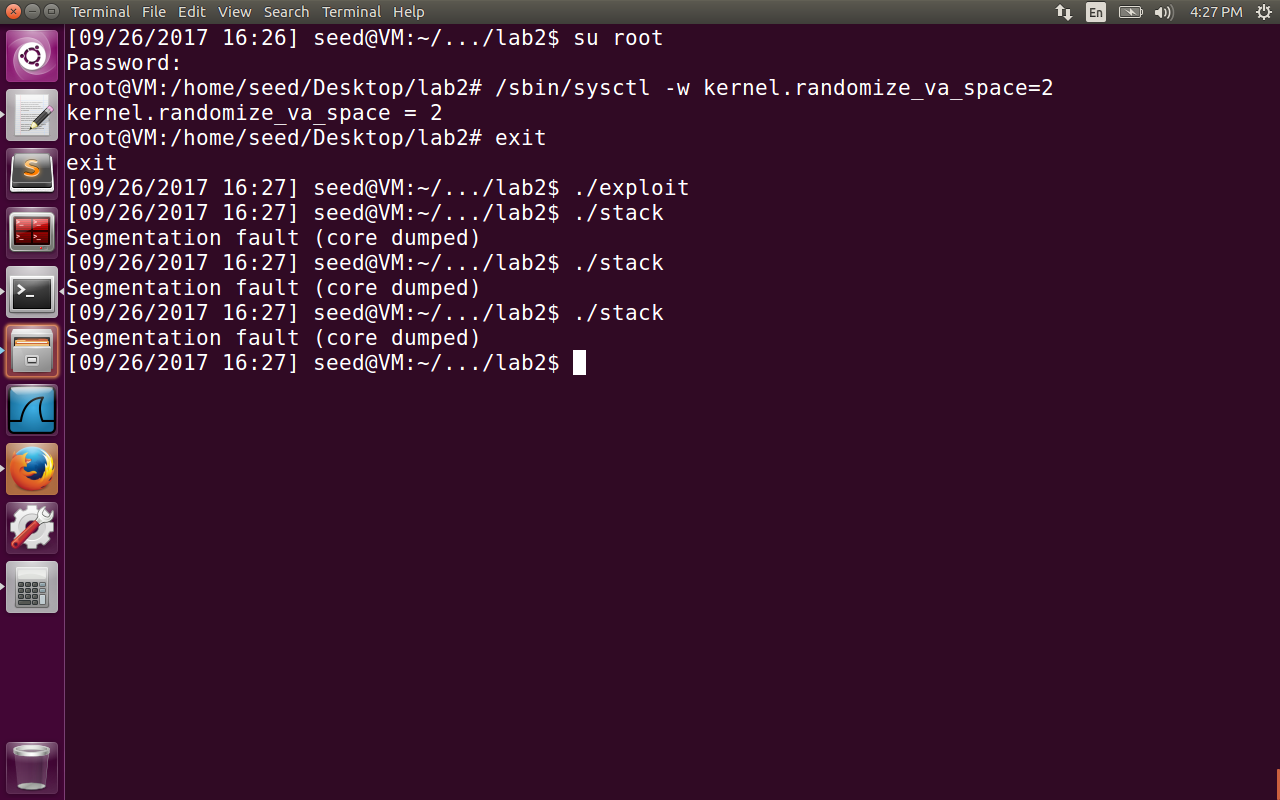
"\x31\xdb" /\* Line 2: xorl %ebx,%ebx \*/

"\xb0\xd5" /\* Line 3: movb $0xd5,%al \*/

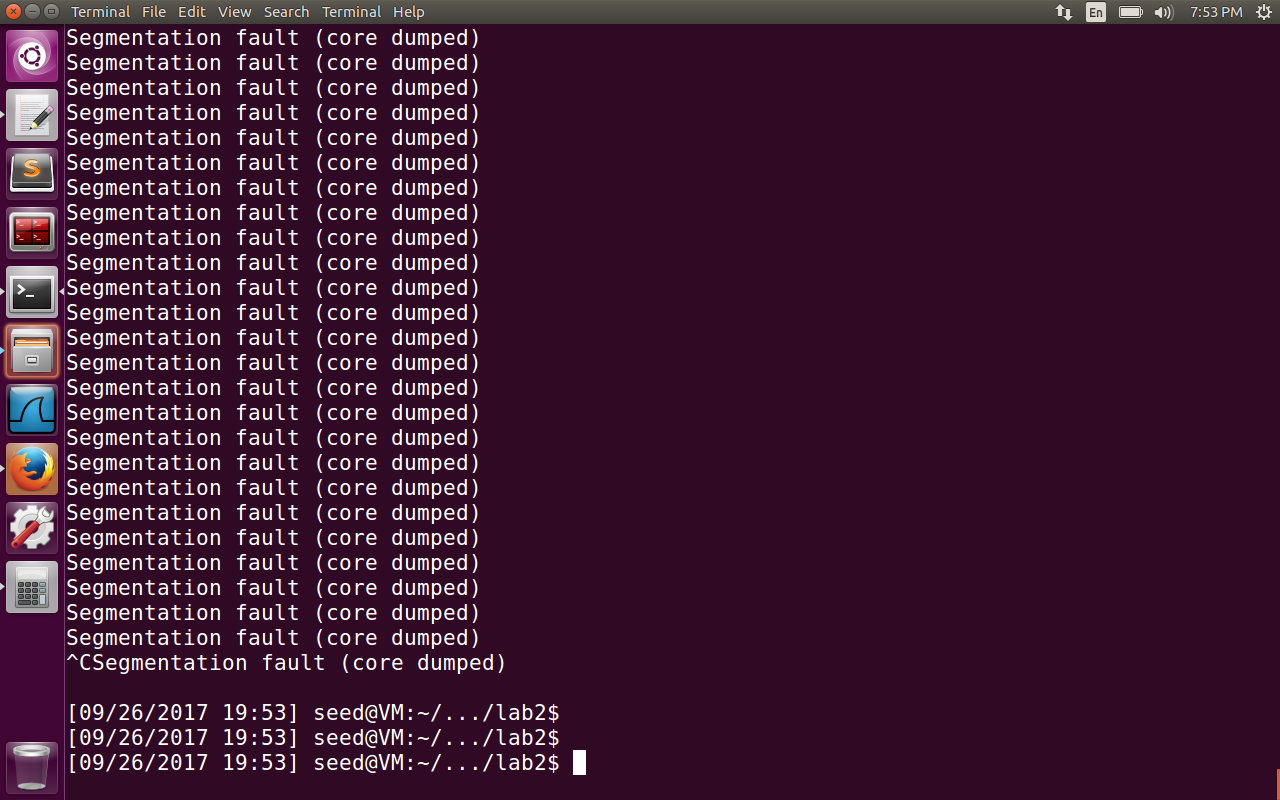
"\xcd\x80" /\* Line 4: int $0x80 \*/…………..

The updated shellcode adds 4 instructions: (1) sets ebx to zero in Line 2, (2) sets eax to 213 via Line 1 and 3 (213 is setuid’s system call number), and (3) executes the system call in Line 4

**2.6 Task 3: Address Randomization**

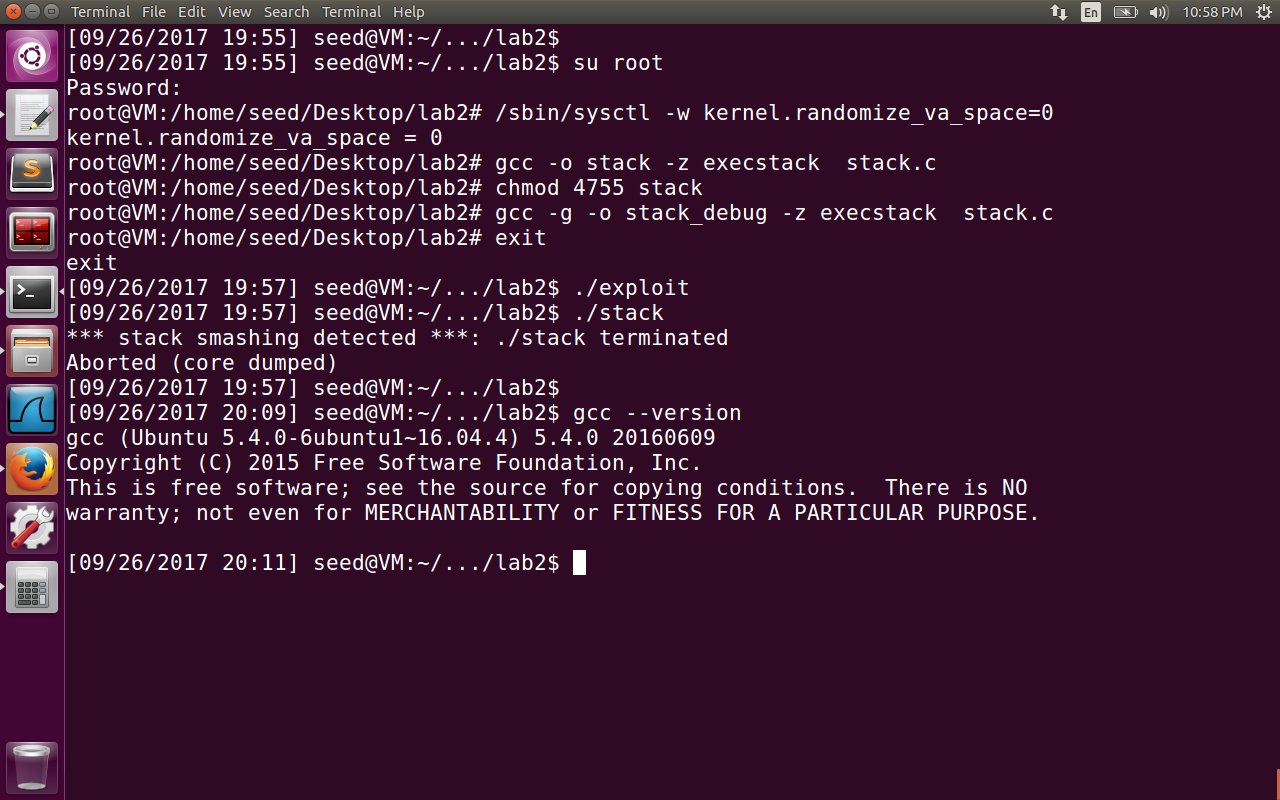
****

Now we have logged in as root user, and enabled address space randomization by kernel. The attack won’t be successful now. We can run a script which continuously tries to get the root shell sh -c "while [ 1 ]; do ./stack; done;”



Even after running for 3 hr couldn’t open the shell program (see the timestamp with previous images) . The probability of stack starting at the address given in the exploit.c program is very low. Hence the chance of invoking the shell program is very less. If we are lucky we can get the shell in 30 min. Some other times, even after running the while script for days, we cannot get the shell. The current picture I have run for 4 hours but couldn’t get the root shell. The advantage with address space randomization is that every time , the program is loaded to memory, stack and heap addresses are different from previous time the program is run.

**2.7 Task 4: Stack Guard**

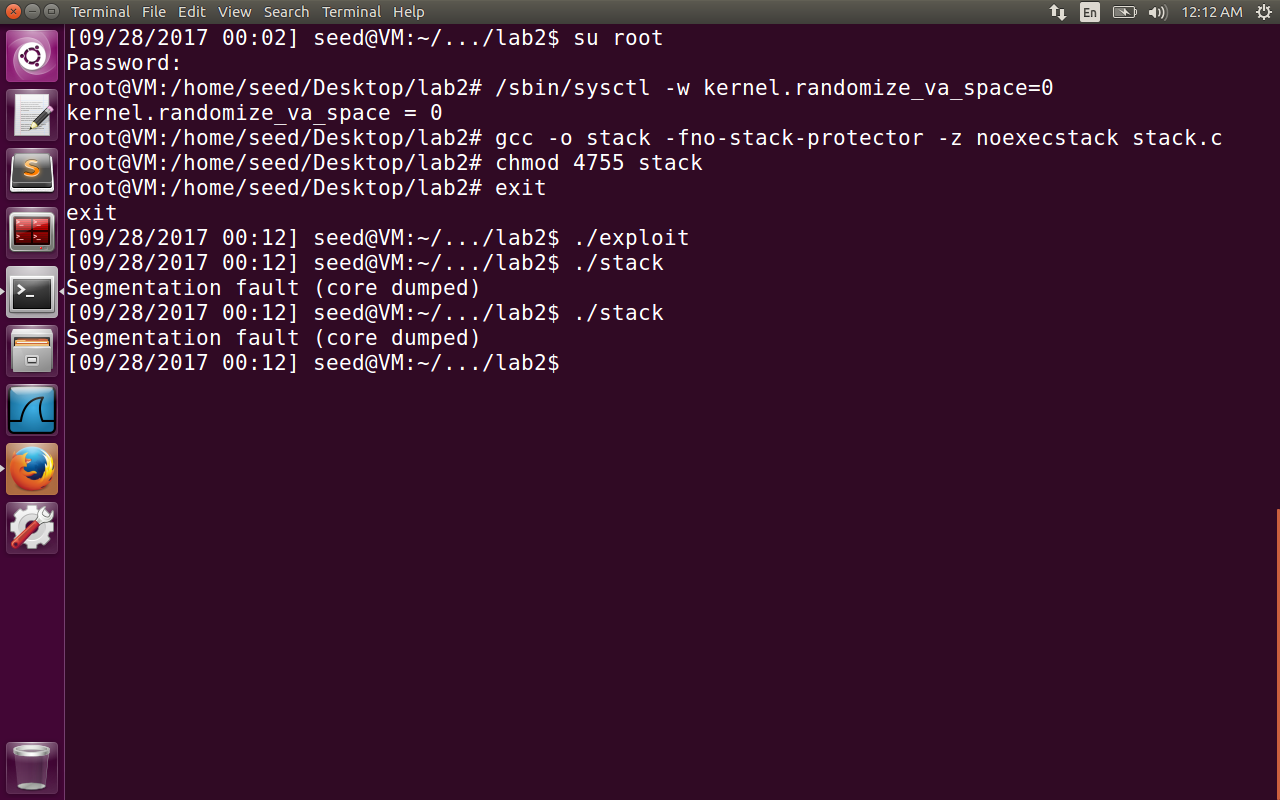
****

We logged in as root and disabled the address space randomization. The stack.c program is compiled without the option –fno-stack-protector. So we are enabling stack guard protection for this task. (It is enabled by default because the current gcc version on my machine is 5.4.0) and it is enabled on the machines with gcc versions >= 4.3.3.

Reference: <https://lwn.net/Articles/584225/>

The basic idea behind stack protection is to push a canary (“a randomly chosen integer”) just after function return pointer is pushed onto stack. The canary value is checked before the function returns. If the value of canary is changed, the program will abort. Generally, stack buffer overflow (aka "stack smashing") attacks will have to change the value of the canary as they write beyond the end of the buffer before they can get to the return pointer. Since the value of the canary is unknown to the attacker, it cannot be replaced by the attack. Thus, the stack protection allows the program to abort when that happens rather than return to wherever the attacker wanted it to go.

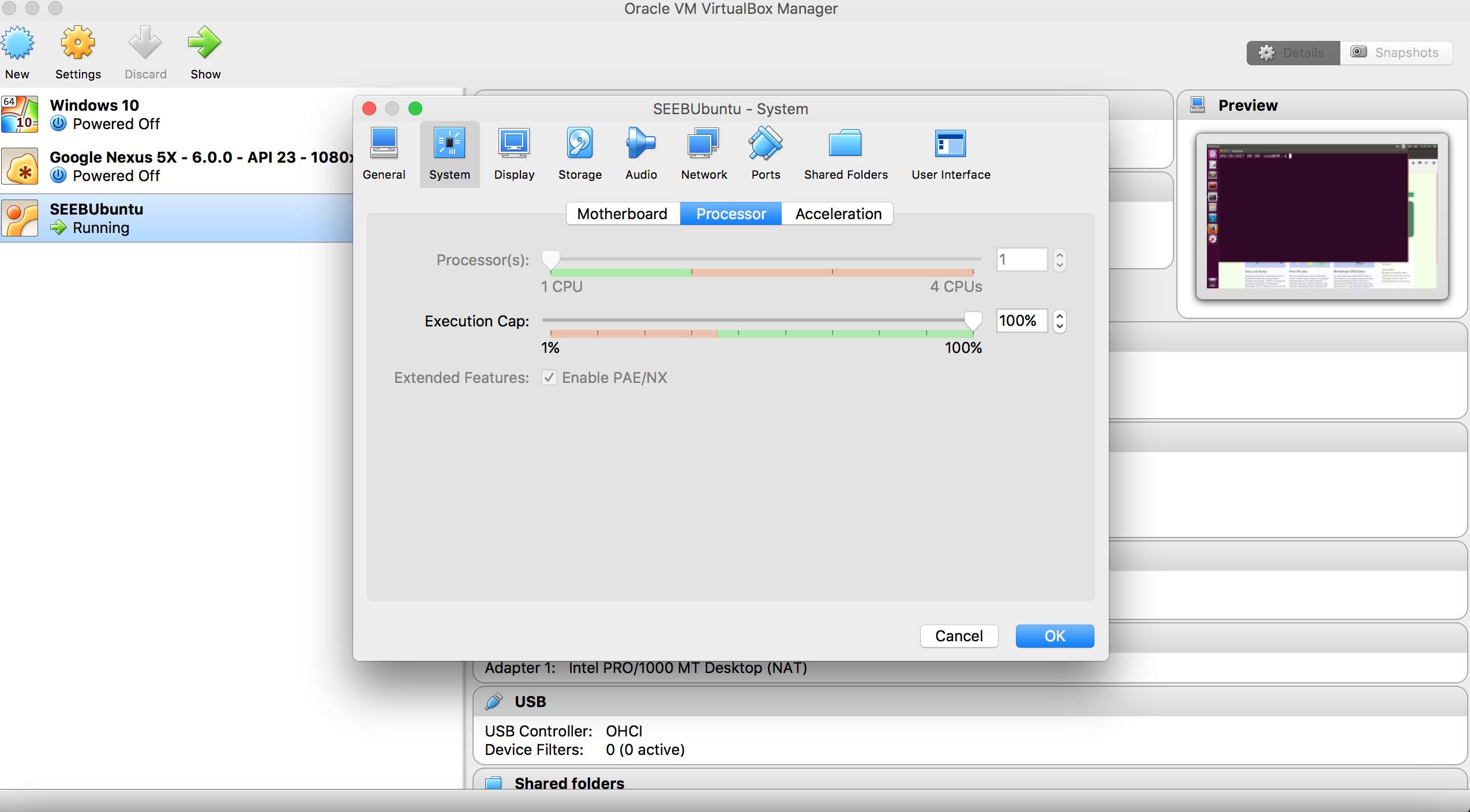
**2.8 Task 5: Non-executable Stack**

****

**Reference:** [**http://www.cis.syr.edu/~wedu/seed/Labs\_12.04/Files/NX.pdf**](http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Files/NX.pdf)

First we have logged in as root, disabled the address space randomization by kernel. Then we compiled the stack.c program with the option **noexecstack.** This protects against the code getting executed on the non-executable memory regions such as stack, heap. The output is a expected, we get the segmentation fault, because we couldnot execute the code on the stack region.

The NX bit (Never execute), known also as XD (execute Disable) is a technology supported by the CPU to disable such execution even when we don’t provide the option noexectsack. I tried to disable the feature on the virtual machine shown below under settings->system . But the virtual machine has thrown some error because of which I couldnot boot Ubuntu with NX bit disabled.



So the NX bit is always needed to be enabled to boot Ubuntu 16.04 and hence the buffer overflow attack cannot be exploited as the stack now becomes non-executable. As mentioned in the lad, we can use return-to-libc attack in such cases to get root shell access.