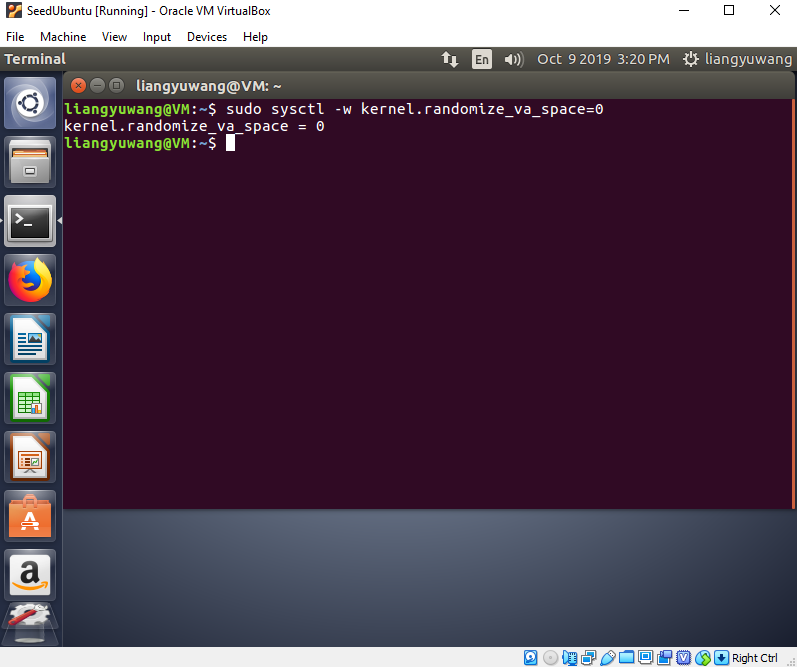
Buffer Overflow Vulnerability Lab

Liangyu W

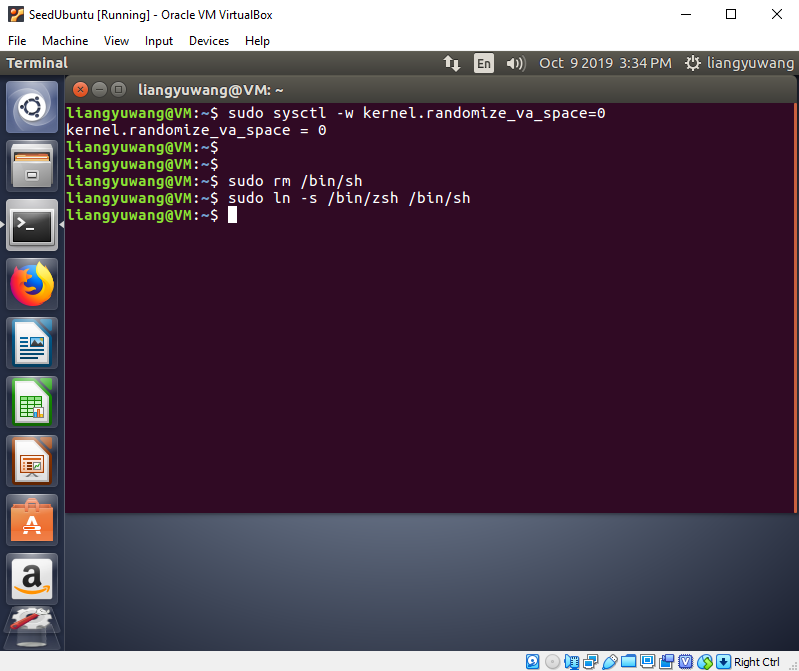
In this lab, we exploit a program with buffer overflow vulnerability by making the program write data to its buffer that is beyond the boundaries of its pre-allocated buffer size, in order to gain root access. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

Turn off countermeasures: Ubuntu have implemented several security mechanisms to make the buffer-overflow attack difficult, so we need to disable them first.

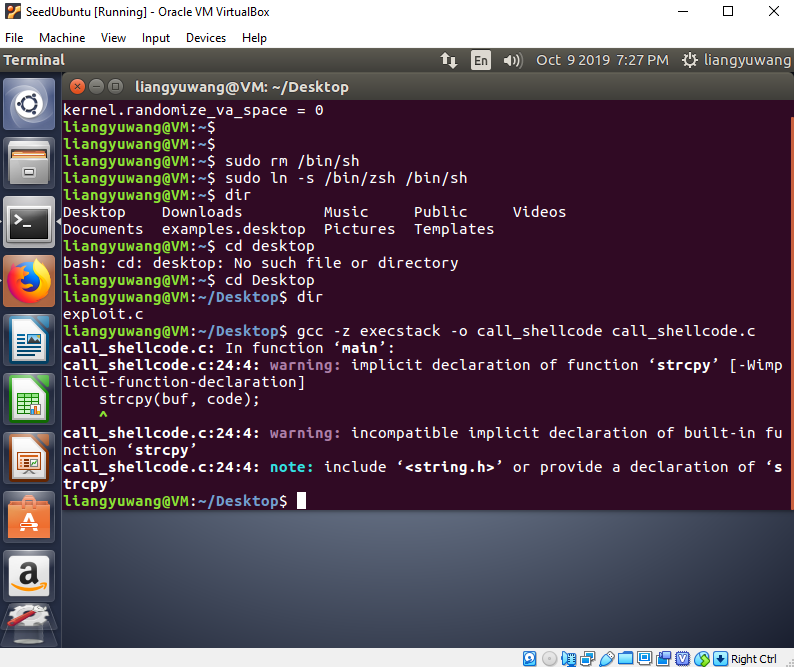
Disable address space randomization using the command “sudo sysctl -w kernel.randomize\_va\_space=0”



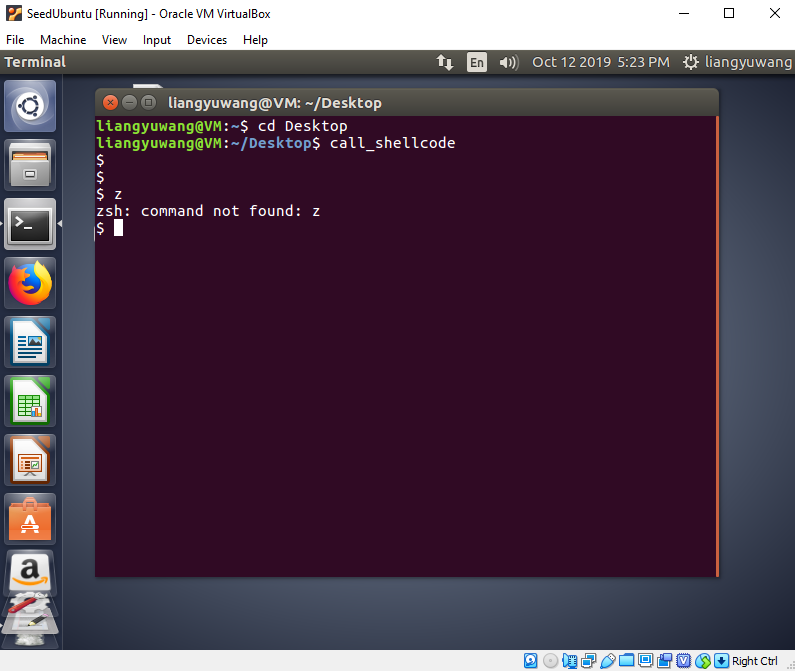
Configuring/bin/sh: Link /bin/sh to zsh



Compile the provided call\_shellcode program using the command “gcc -z execstack -o call\_shellcode call\_shellcode.c”. By default, stacks are non-executable, which prevents the injected malicious code from getting executed. The “-z execstack” option turns this countermeasure off and makes the stack of the compiled program executable.



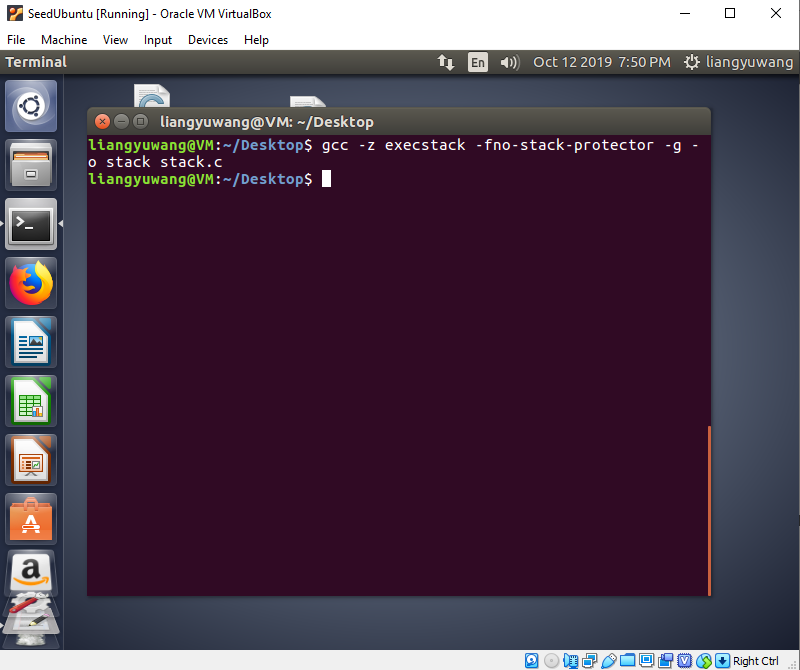
Run the compiled call\_shellcode program.



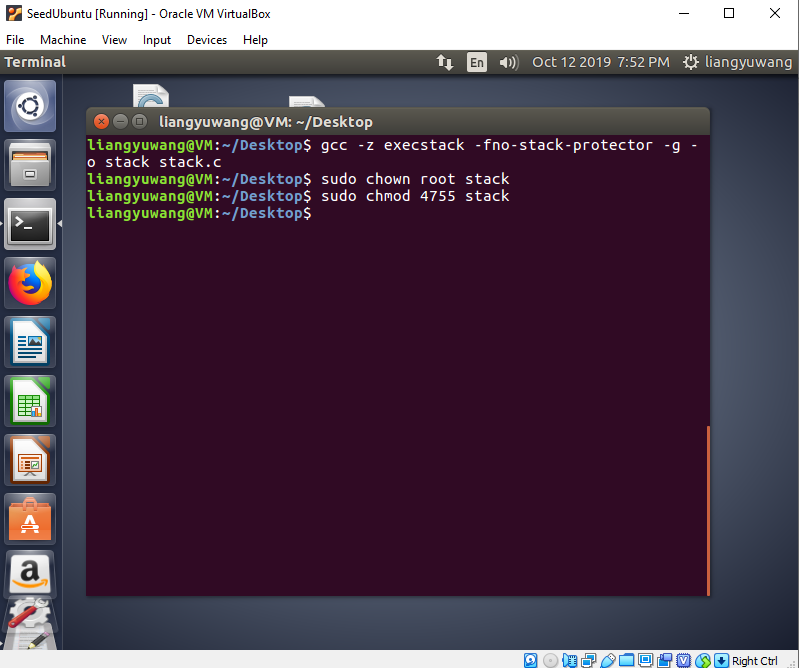
Call\_shellcode program invokes the execve() system call to call up the zsh shell program.

We are provided with a vulnerable stack program, the program reads 517 bytes of data from a file called "badfile", and then copies the data to a buffer of size 24. Clearly, there is a buffer overflow problem. To exploit this vulnerability, we need to get our malicious code into the memory of the running program. To do this, we will place our malicious code in "badfile", so when the stack program reads from the file, the code is loaded into the str[] array; when the program copies str to the target buffer, the code will then be stored on the stack.

Compile the provided vulnerable stack program with command “gcc -z execstack -fno-stack-protector -g -o stack stack.c”. The “-fno-stack-protector” option disables GCC’s countermeasure called StackGuard Protection Scheme. If not disabled, the buffer overflow attack will not work. The “-z execstack” option makes the stack of the compiled program executable. We use the -g flag to compile the program, so debugging information is added to the binary.

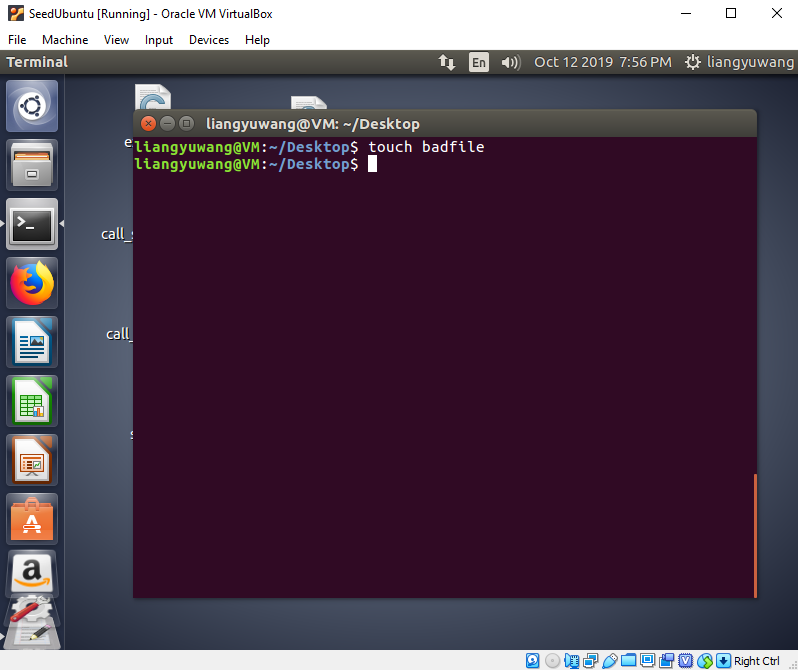


Change the ownership and permission of the stack program to make the program a root-owned Set-UID program. To achieve this, first change the ownership of the program to root, then change the permission to 4755 to enable the Set-UID bit.



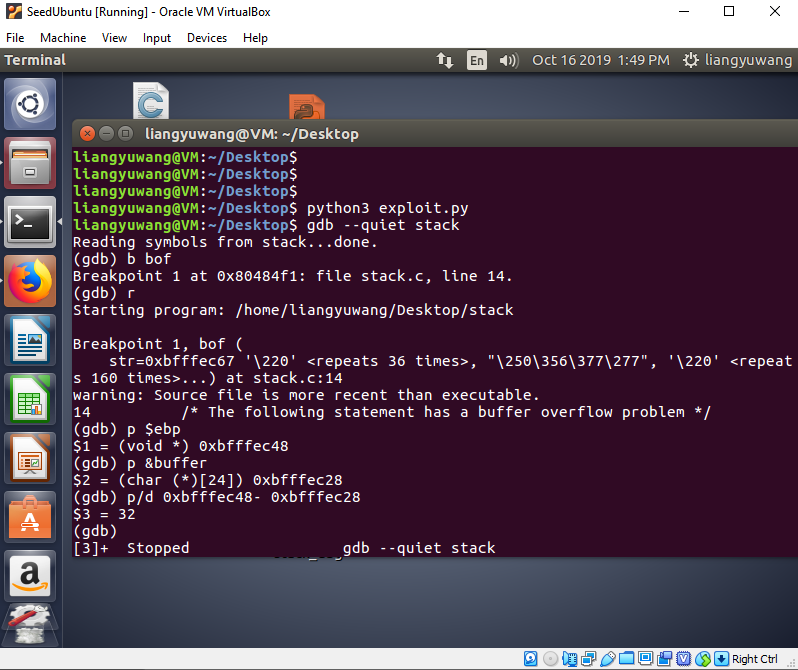
Now we need to force the program to jump to our malicious code when it’s running in the memory. To do that, we need to know the memory address of the malicious code. Unfortunately, we do not know where exactly our malicious code is. We only know that our code is copied into the target buffer on the stack, but we do not know the buffer’s memory address, because its exact location depends on the program’s stack usage. We will use debugging method to find out where the stack frame resides on the stack, and use that to derive the memory address of our code.

We use the “touch badfile” command to create an empty “badfile”.



We disassemble the vulnerable stack program with gnu debugger and set a breakpoint at bof() function. And then we run the program and the program stops inside bof() function:

We now print out the value of the frame pointer $ebp and the address of the &buffer using gdb’s p command:



From the above, we can see that the value of the frame pointer is 0xbfffec48, therefore the return address is stored in 0xbfffec48 + 4 = BFFFEC4C. And the first address that we can jump to 0xbfffec48 + 8 (the memory regions starting from this address is filled with NOPs). Therefore, we can put 0xbfffec48 + 8 inside the return address field.

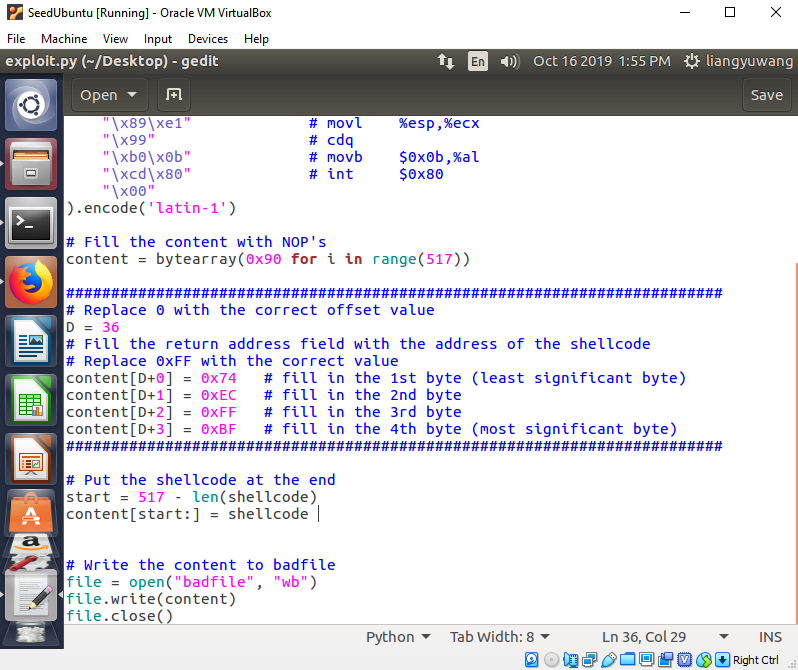
From the debugging results, we can calculate the distance between $ebp and the &buffer’s starting address. We get 32. Since the return address field is 4 bytes above where $ebp points to, the distance between the buffer’s starting point and the return address field is 36.

We are provided with a exploit.py file that generates the “badfile” that contains the malicious code. The exploit.py program fills the region above the new return address with NOP values to create multiple entry points for our malicious code.

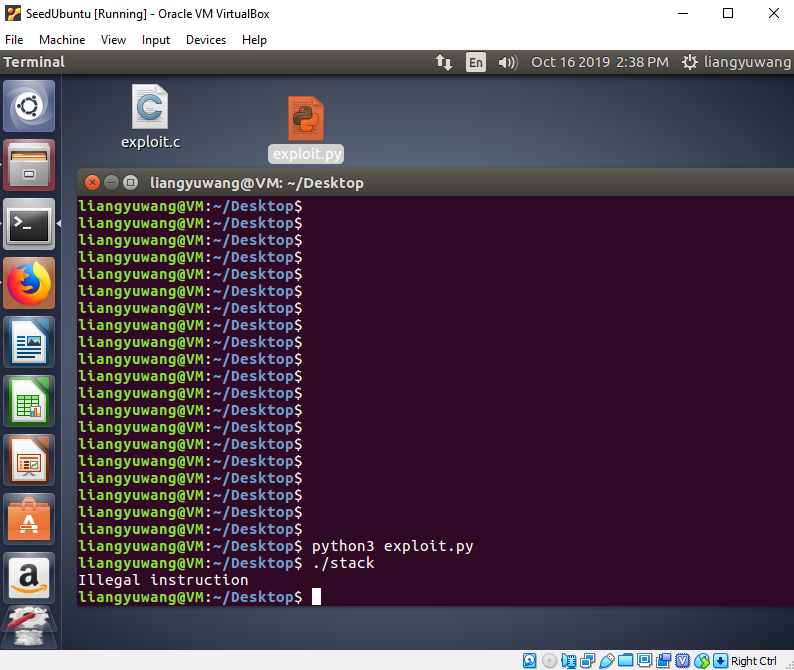
We use 36 as offset, so our code address is:

0xbfffec48 + 36 + 8 = BFFFEC74

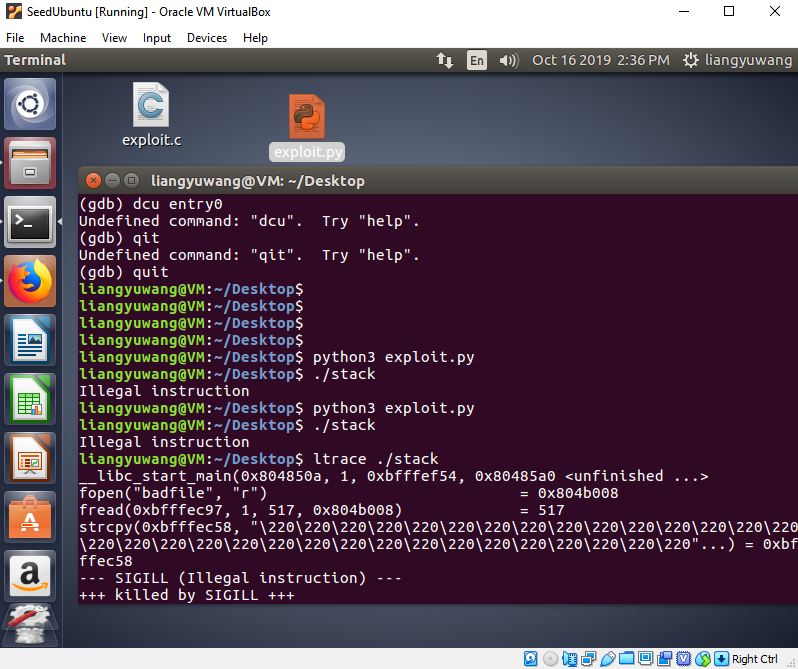
Modify the exploit.py file as follows:



Run the exploit program to generate badfile, then run the stack program, but we get “illegal instruction”.



So we run the stack program through ltrace:



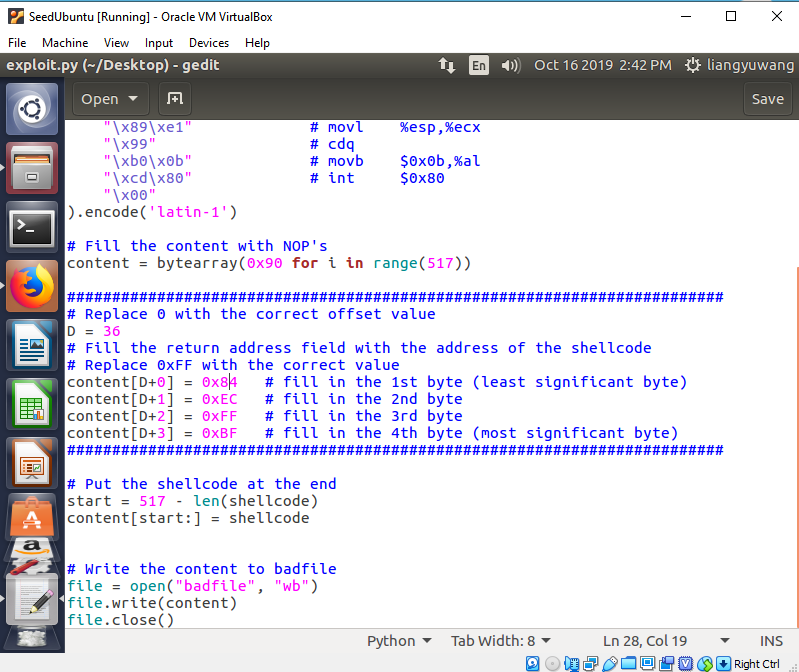
we note that the buffer variable which reads from badfile has address: 0xbfffec58

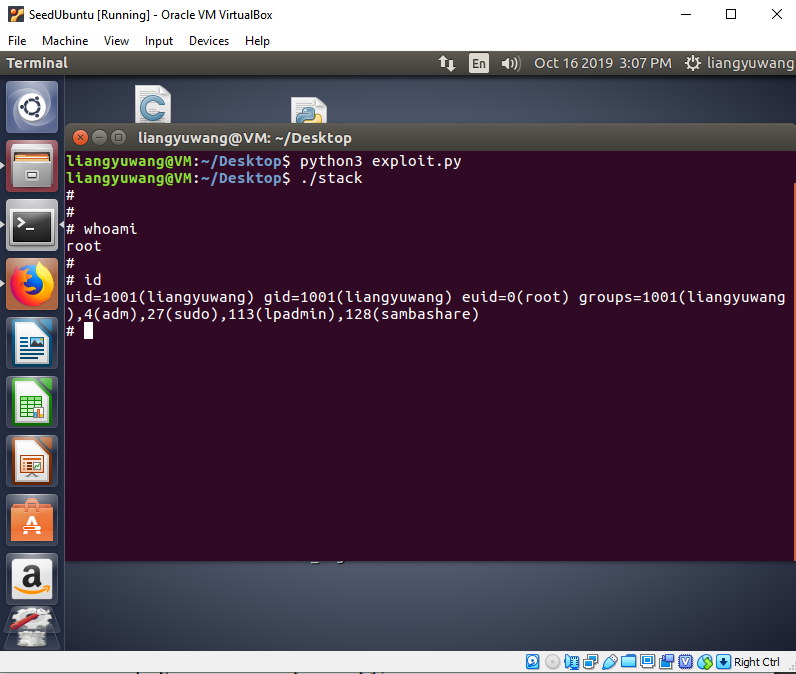


So we re-calculate our malicious code address:

0xbfffec58 + 36 + 8 = BFFFEC84

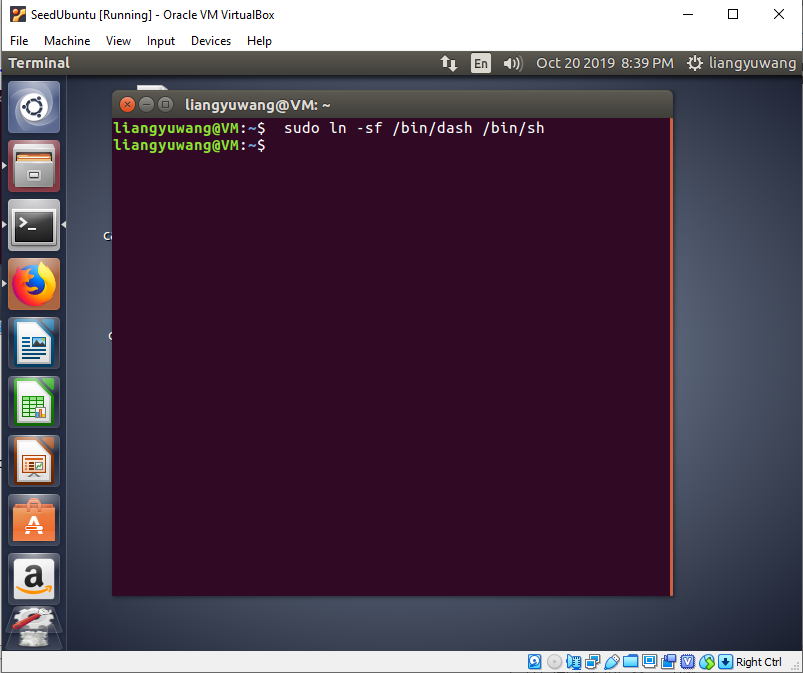
And we modify our exploit.py file as follows:



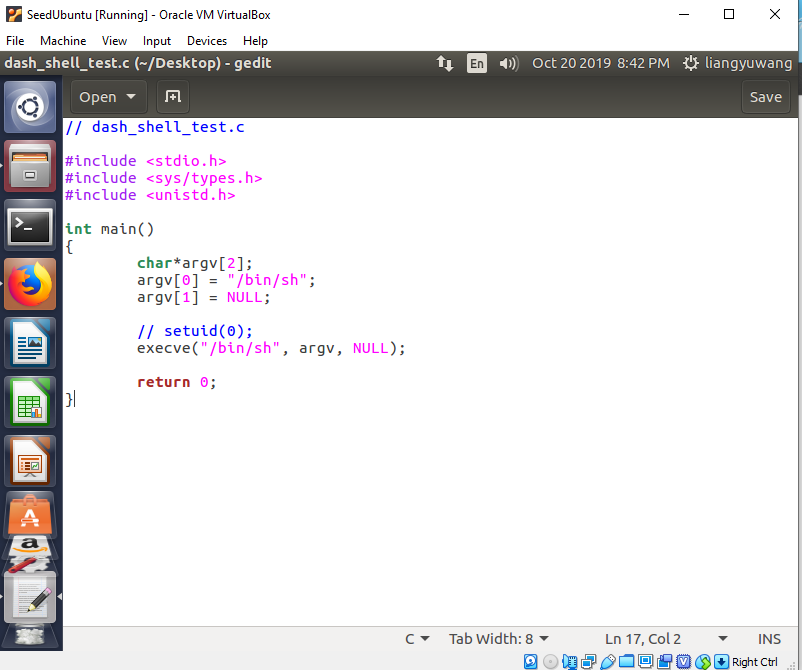
We regenerate badfile, run the stack program, and we successfully obtains root privilege: 

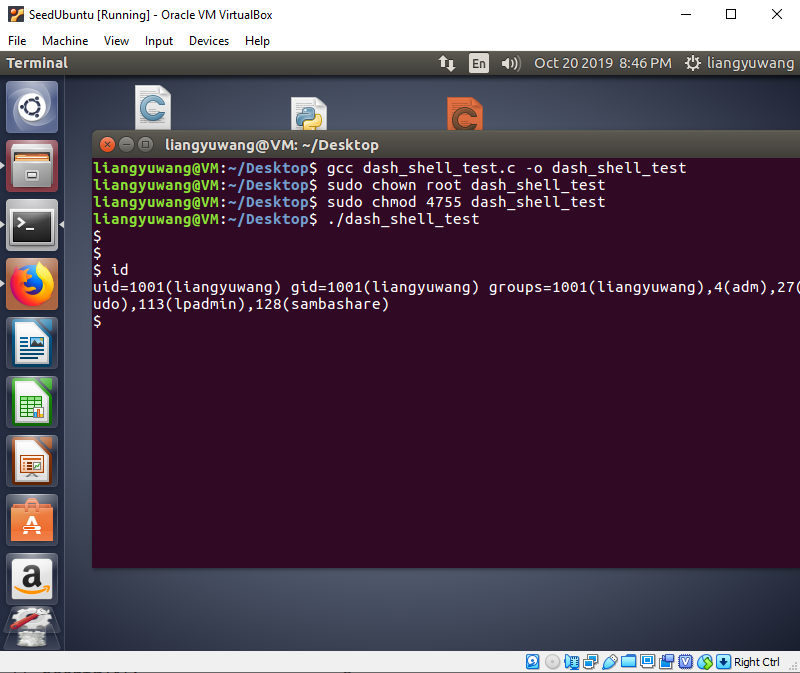
**Defeating dash’s Countermeasure**

We will first change the/bin/sh symbolic link, so it points back to /bin/dash using command “ sudo ln -sf /bin/dash /bin/sh” :

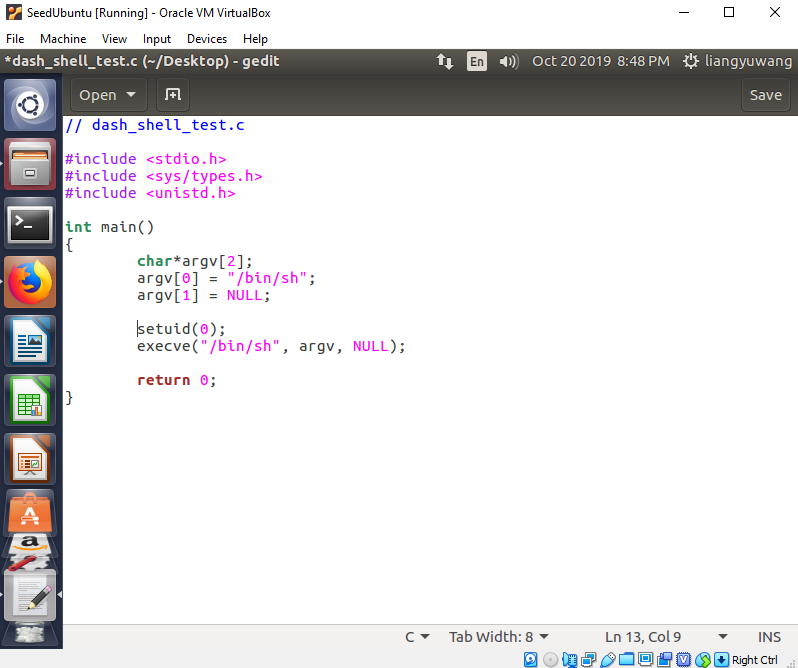


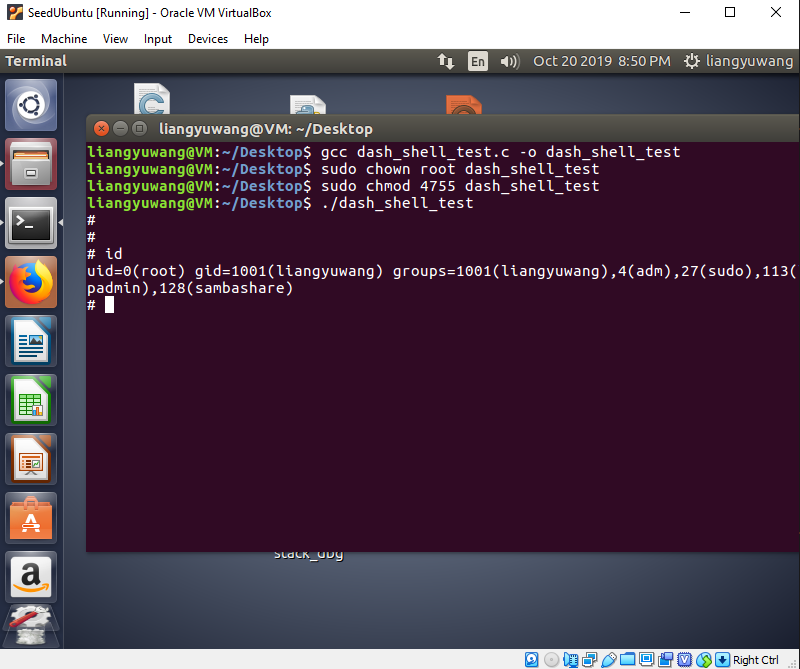
We compile and run the dash\_shell\_test program without the setuid(0) line:





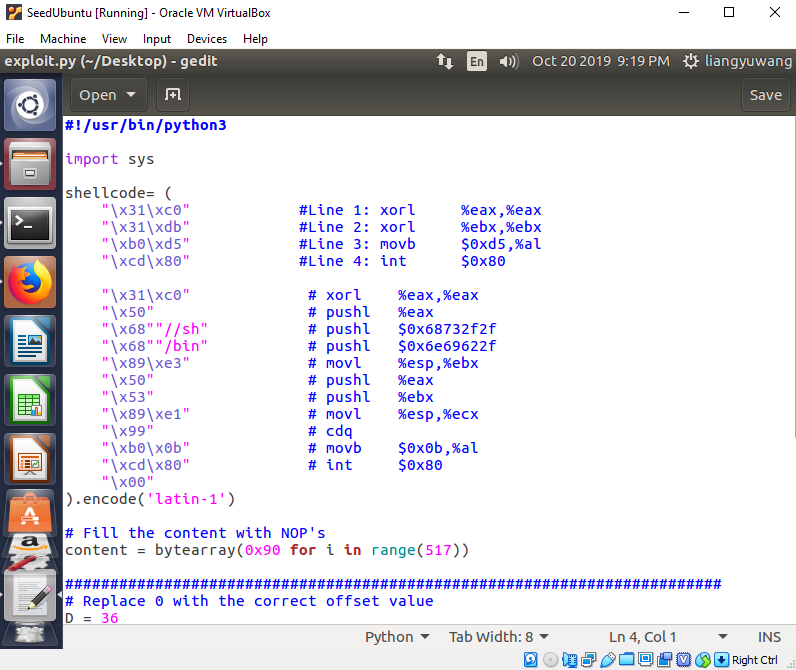
We then run and compile dash\_shell\_test program with the setuid(0) line:

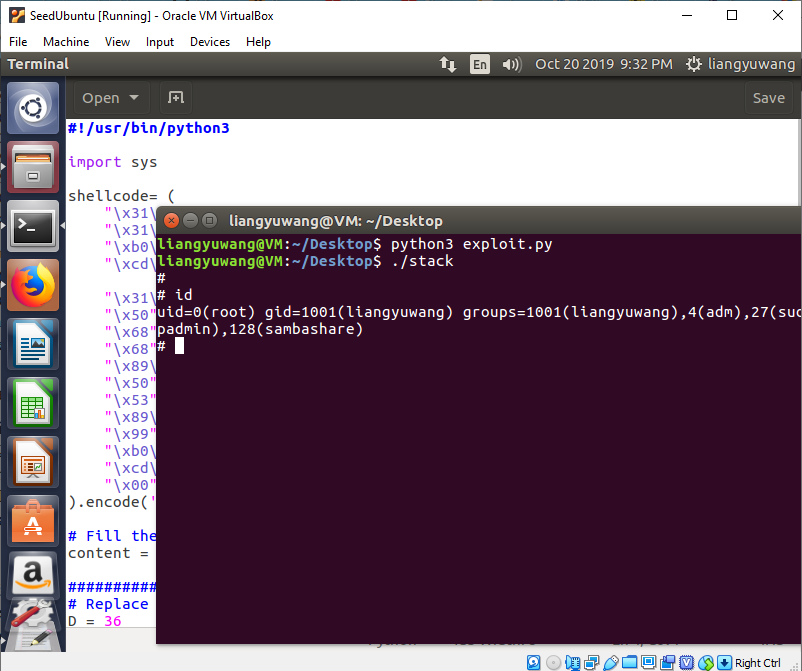




With the setuid(0) line, we are able to get root; without it, we are unable to get root. It shows that dash shell can detect when the effective UID is not the real UID.

We add the assembly code for invoking seuid(0) to our exploit.py program as follows:





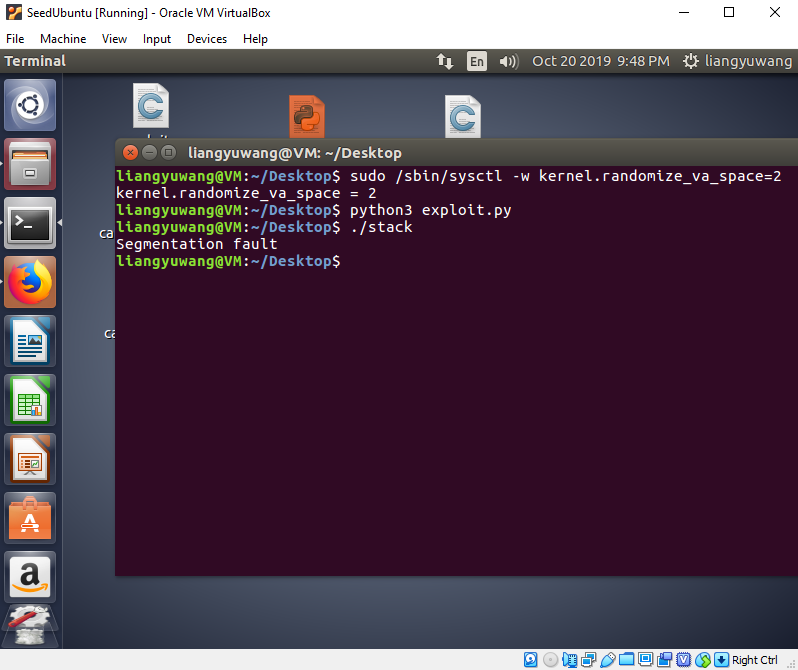
We run exploit.py and stack program and obtains root, and this time our UID is root(0) instead of 1001, this means we now have a real root process, more powerful than before.

**Defeating Address Randomization**

We now want to use brute force approach to defeat Linux’s address space randomization countermeasure.

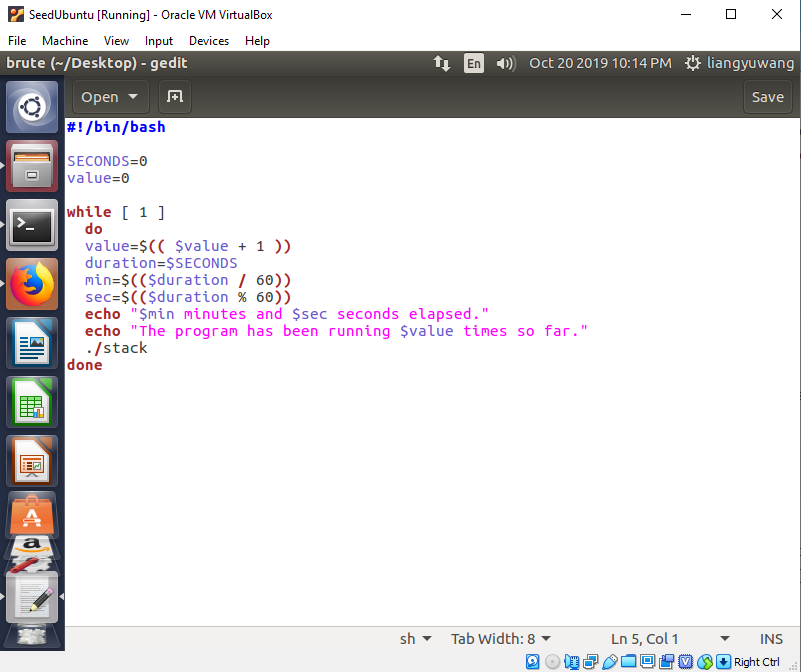
We enable Ubuntu’s address space randomization using the command: sudo /sbin/sysctl -w kernel.randomize\_va\_space=2

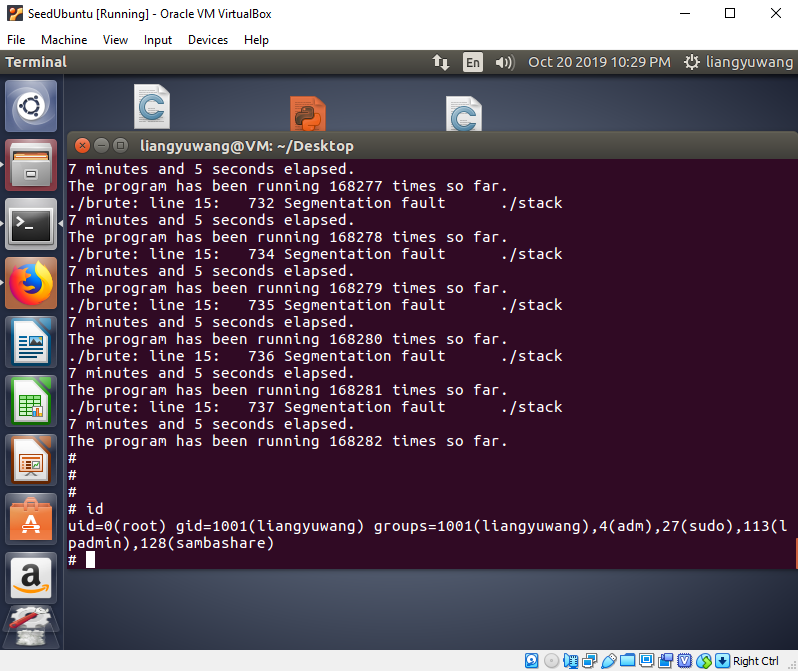
Then we run the buffer overflow attack and it fails and we get segmentation fault error:



This happens because address randomization randomizes the return address, so a hard-coded attack won’t work.

So we create a bash shell script that runs the the stack program in a loop, that will hopefully hit upon the correct return address:



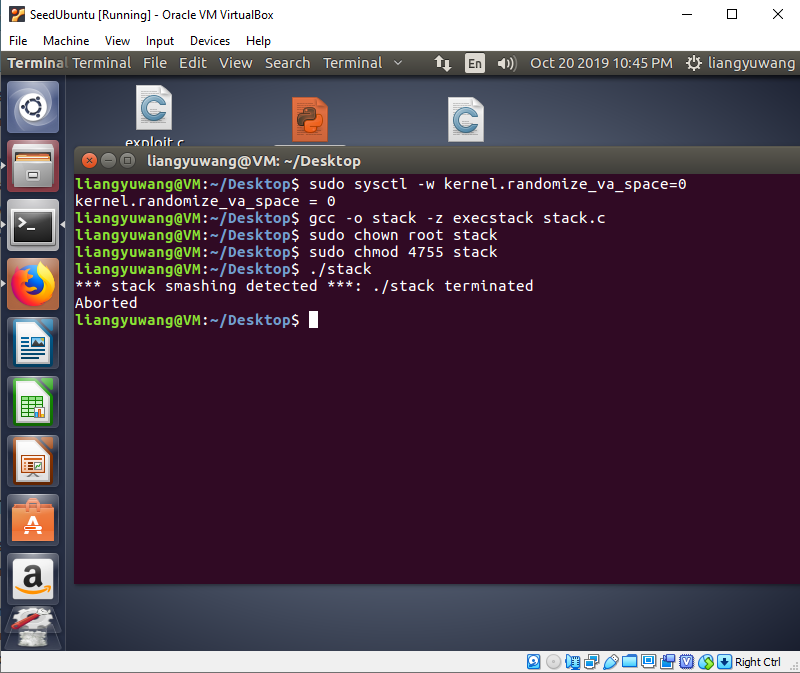


We run the script, and got root after 7 minutes 5 seconds and 168282 attempts.

**Turn on the StackGuard Protection**

We turn off address space randomization: sudo sysctl -w kernel.randomize\_va\_space=0

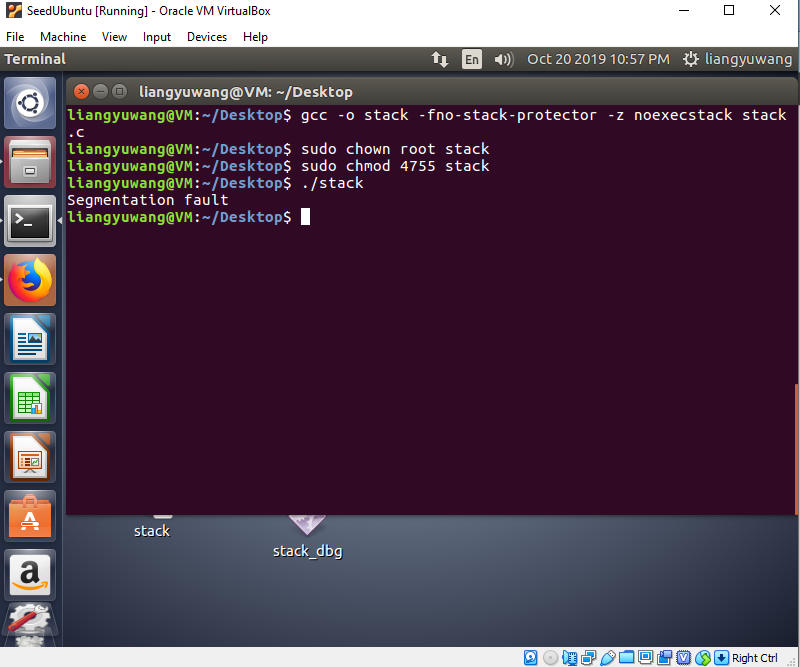
We recompile stack without the -fno-stack-protector option and run the stack program:



We get a “stack smashing detected” error and then the program core dumps.

**Turn on the Non-executable Stack Protection**

We intentionally make stacks executable and recompile stack program using the noexecstack option: gcc -o stack -fno-stack-protector -z noexecstack stack.c



We run stack and get segmentation fault, so we cannot get a shell because non-executable stack protection blocks shell code from executing on the stack.