Secure System Design – CS392

Assignment2: Overflow attack

Tanishq Malu 1901CS63

**The program given below has a buffer overflow vulnerability:**

**# include <stdlib.h>**

**# include <string.h>**

**# include <stdio.h>**

**int bof (char \*str) {**

**char buffer [24];**

**strcpy (buffer, str);**

**return 1;**

**}**

**int main (int arg c, char \*\*arg v) {**

**char str [517];**

**file \*badfile;**

**badfile = fopen("badfile", "r");**

**fread (str, sizeof(char), 517, badfile);**

**bof(str);**

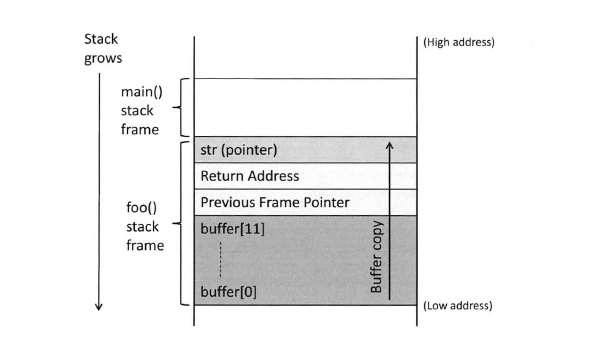
**print ("Returned Properly\n");**

**return 1;**

**}**

**Task (1): What is the reason for buffer overflow vulnerability in this code give a demo (execution snapshot) after execution of it and explain it properly. (5+5 =**

**10).**

Each function has local memory associated with it to hold incoming parameters, local variables, and (in some cases) temporary variables. This region of memory is called a stack frame and is allocated on the process’ stack. A schematic diagram of main stack is as shown here:

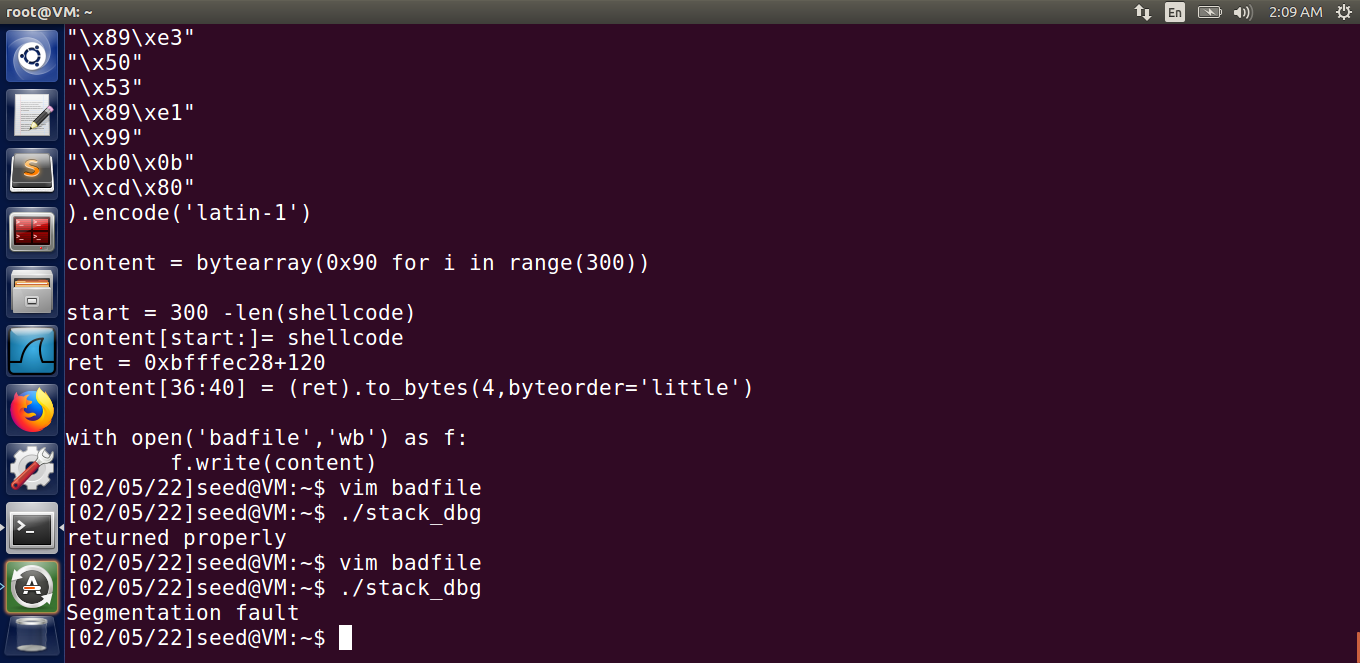
Thus when we call our function ‘bof’, it allocates the space for buffer array into the stack. However, the strcpy which we use to copy our string from badfile into buffer doesn’t stop copying the characters until it receives ‘/0’ character. Thus, if a string of size greater than the length of our buffer array is copied, the strcpy function keeps copying the data and this leads the other data above the the buffer array in the stack to be overwritten by the content of string.

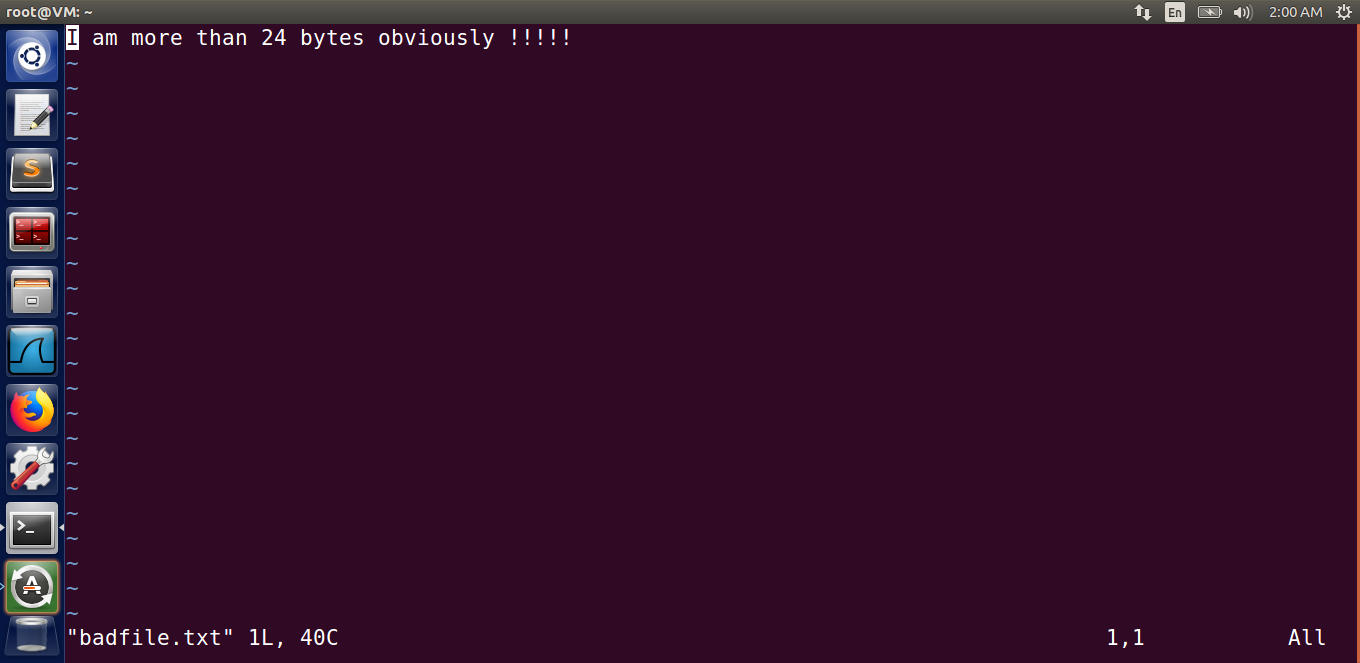
23

bof

**Demo:**

Let’s try to compile the code and set the length of input badfile firstly less than 24 char and than more than 24 characters.



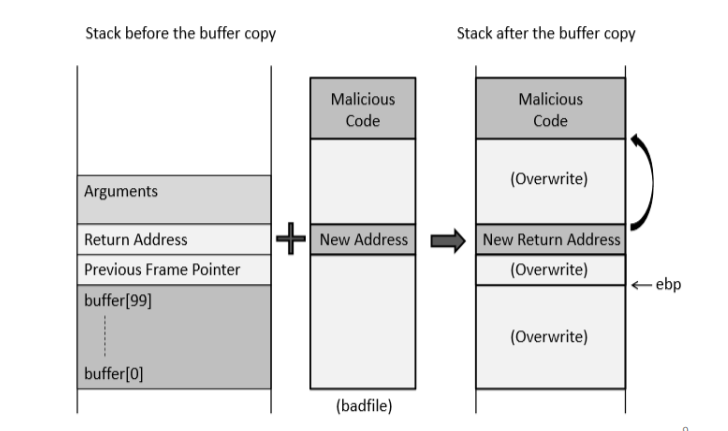


**Initial badfile with less than and more 24 char respectively**

We can clearly see that, when length of our badfile is greater than 24 char which is the size of our buffer, we receive a segmentation fault. Now let us try to understand why we received a segmentation fault.

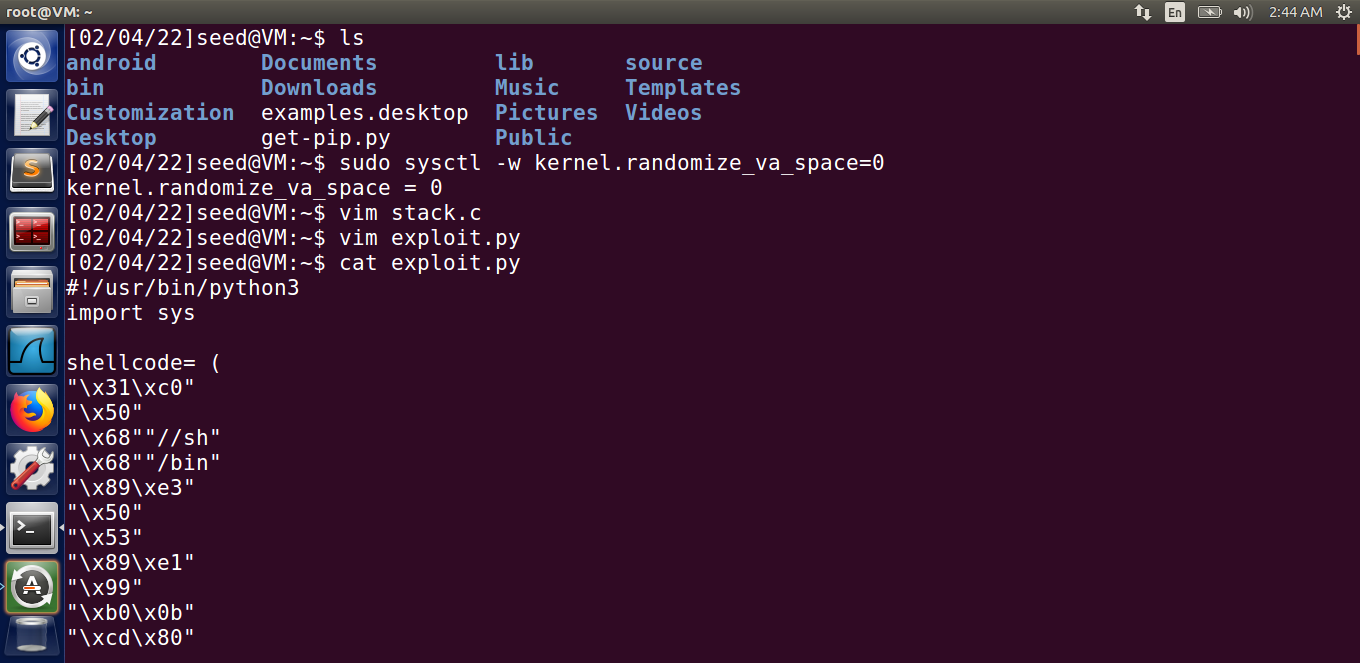
Due to the changes in address of return address while copying of string, it now points to a completely different address. This address may point to a valid address or invalid address as well. In our case, we fed some random characters in our badfile and thus our return address pointed to an invalid address and since it can’t be read, we got a segmentation fault.

However, if we properly feed the content of our badfile and make the return address point to our malicious code, we can easily breach the security. Thus, this code is clearly vulnerable to buffer-overflow attack and can be exploited by the attackers to take control of the system.



23

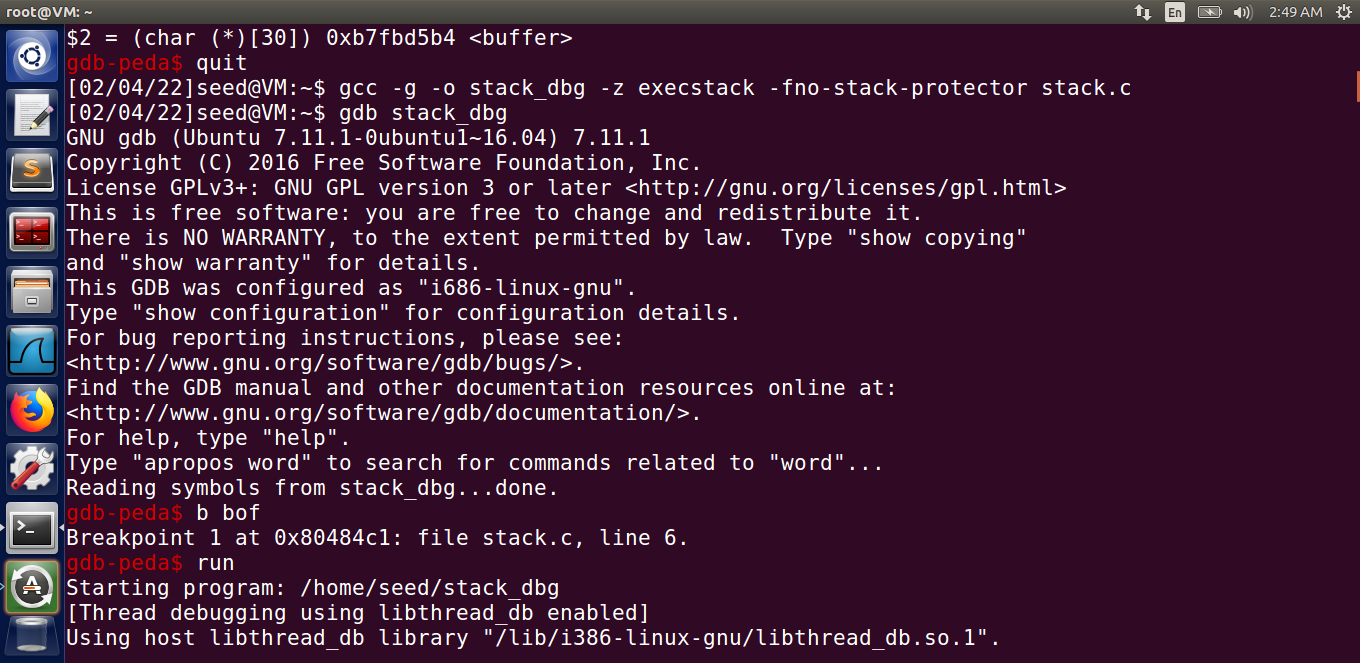
Let’s now try to exploit this vulnerability and try to get access to the root shell using it. Firstly let turn off the address randomization. It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for us to guess the address of the injected malicious code.



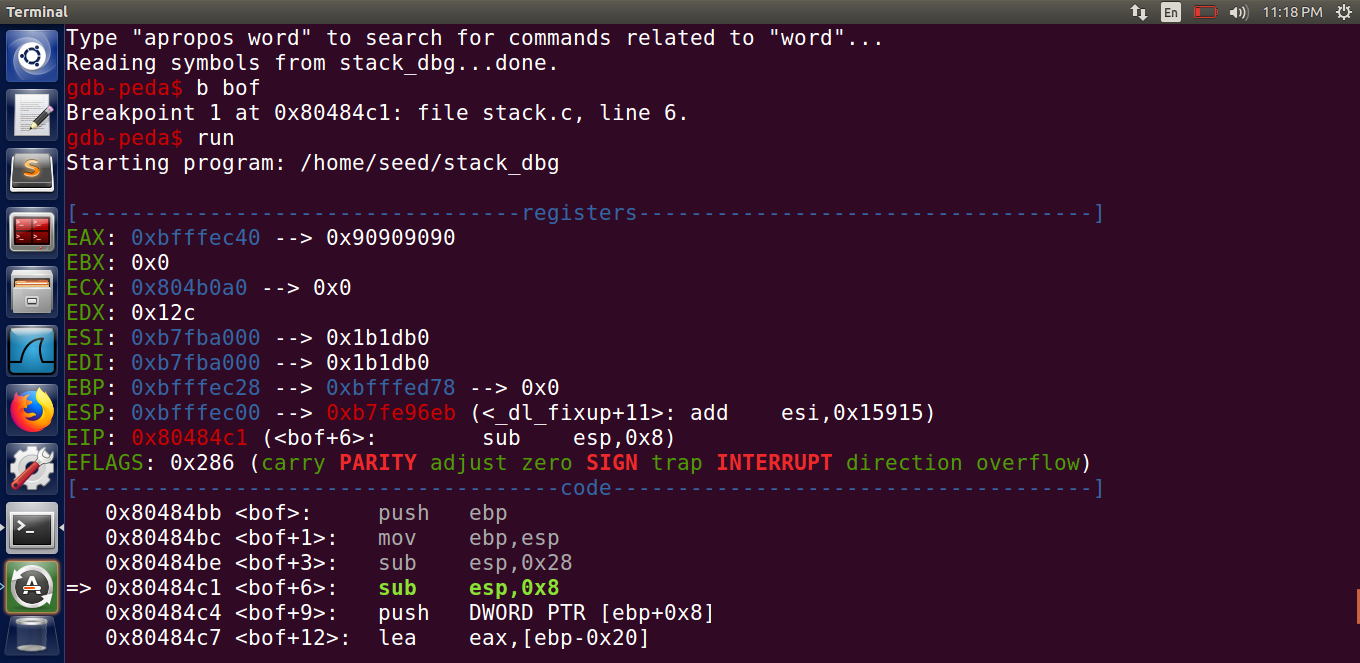
To make a successful buffer overflow attack, follow this steps:

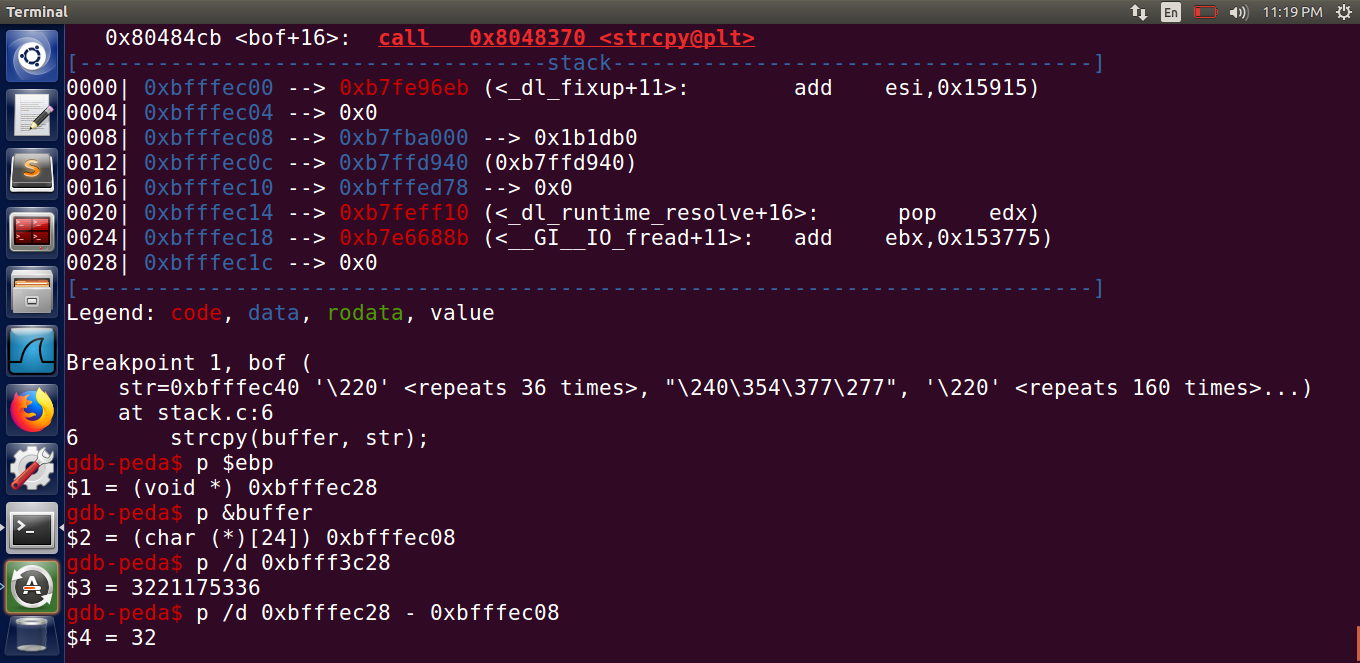
Step 1: Distance Between Buffer Base Address and Return Address

This can be done by using the gdb debugger tool and setting a breakpoint at our bof function. Once we know the frame pointer of bof, we can easily locate our malicious code with positive offset to this pointer.



Note: Countermeasures like execstack and stack needs to be switched off so that we can successfully launch our attack.





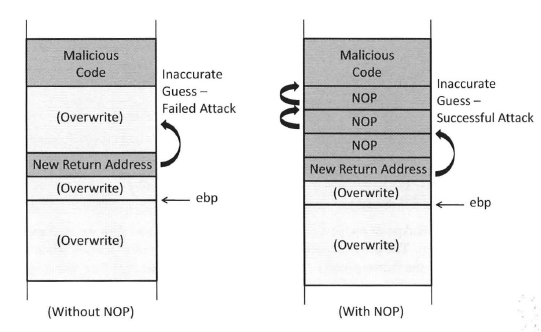
Distance between them

Getting the base address of buffer

Getting the address of ebp

We now know the distance between the base address and the ebp pointer. The return address to previous function is placed just above eb pie ebp +4. Thus we also get the address of ‘return address’. Thus we can easily place our intended address of malicious code at this address.

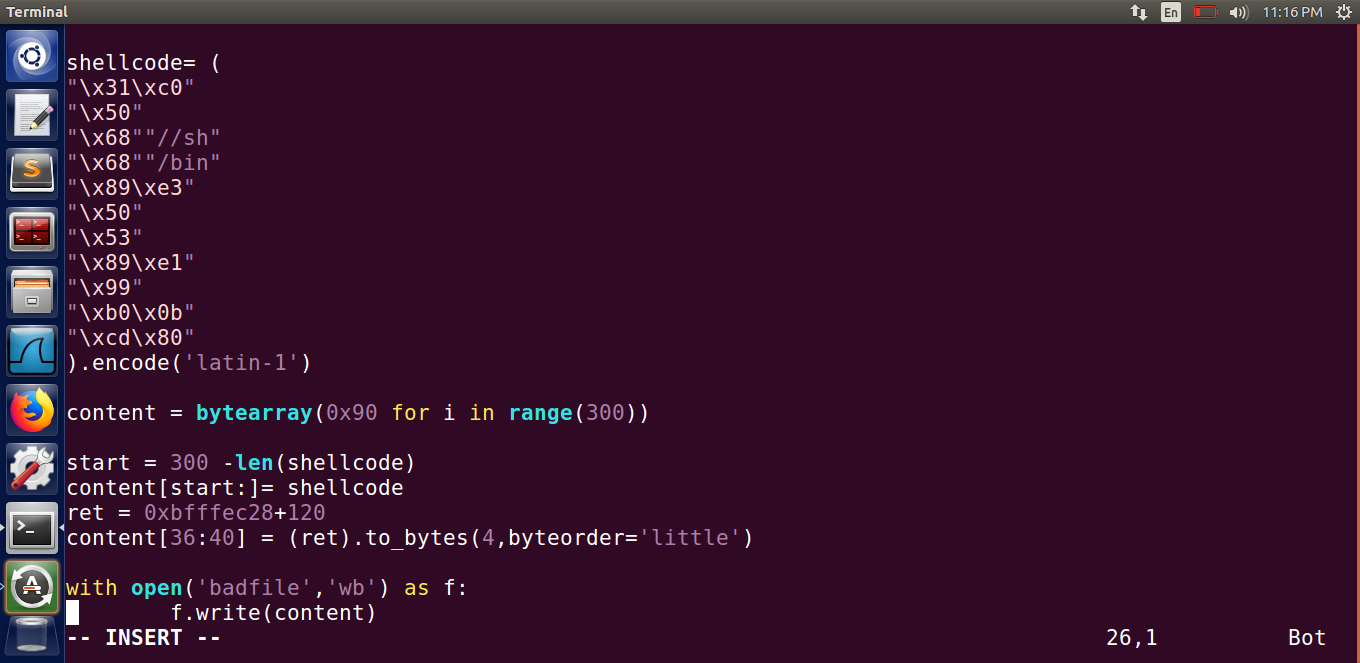
Step 2: Badfile construction

Proper creation of badfile is the most important step. If the badfile is not created properly and the return address is not mentioned clearly we might not reach to correct address of our malicious code and this could lead to failure of attack.

However we can overcome this problem by filling our badfile NOP instruction. This results in creation of multiple entry point for our malicious code and thus even if our return address points to address before our malicious code it will eventually get executed.

N-Op instruction are represented by 0x90 in assembly language and we will use it our badfile construction.

To create a badfile we will write a python script.



Shellcode: This code is basically the assembly language code for executing a shell.

i.e. execve(“/bin/sh”, argument, Null) ,where

“/bin/sh” represents the command to be executed

argument[0] = “/bin/sh”, argument[1]=0 represents the arguments to be passed.

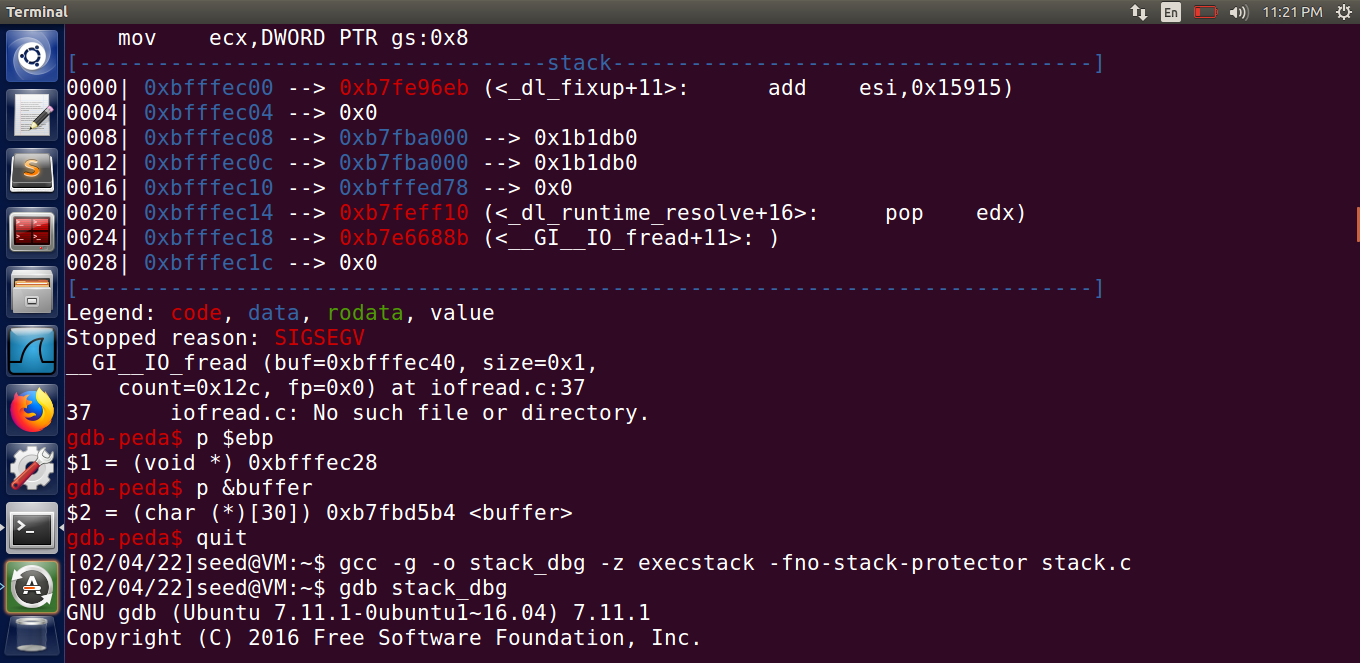
Null represents that no environment variables are passed.

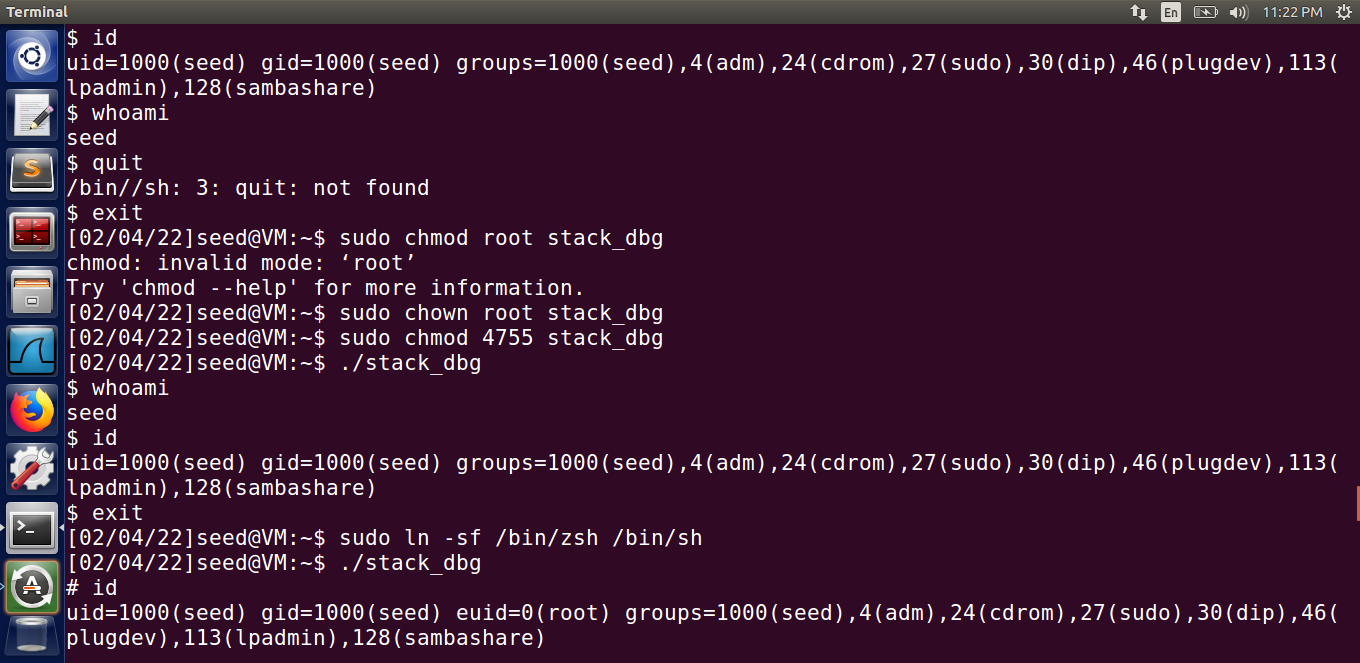
Ret = ebp + offset, it basically is the new return address

Byteorder =’little’ means that the data is stored in little endian order.

Once the python script is created, we run it to form our badfile. We know compile our vulnerable code and turn off the countermeasures.

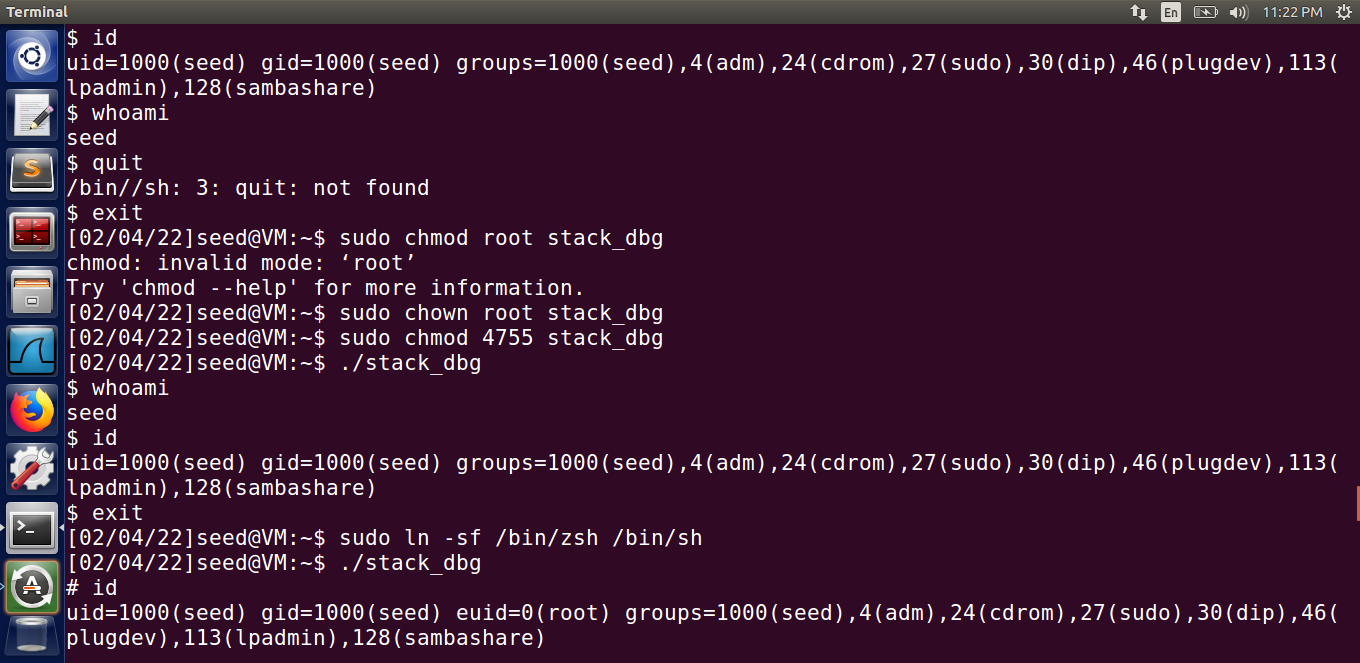
If a buffer overflow vulnerability can be exploited in a privileged Set – UID root program, the injected malicious code, if executed, can run with the root's privilege, we thus change the ownership and make it a set-uid program.

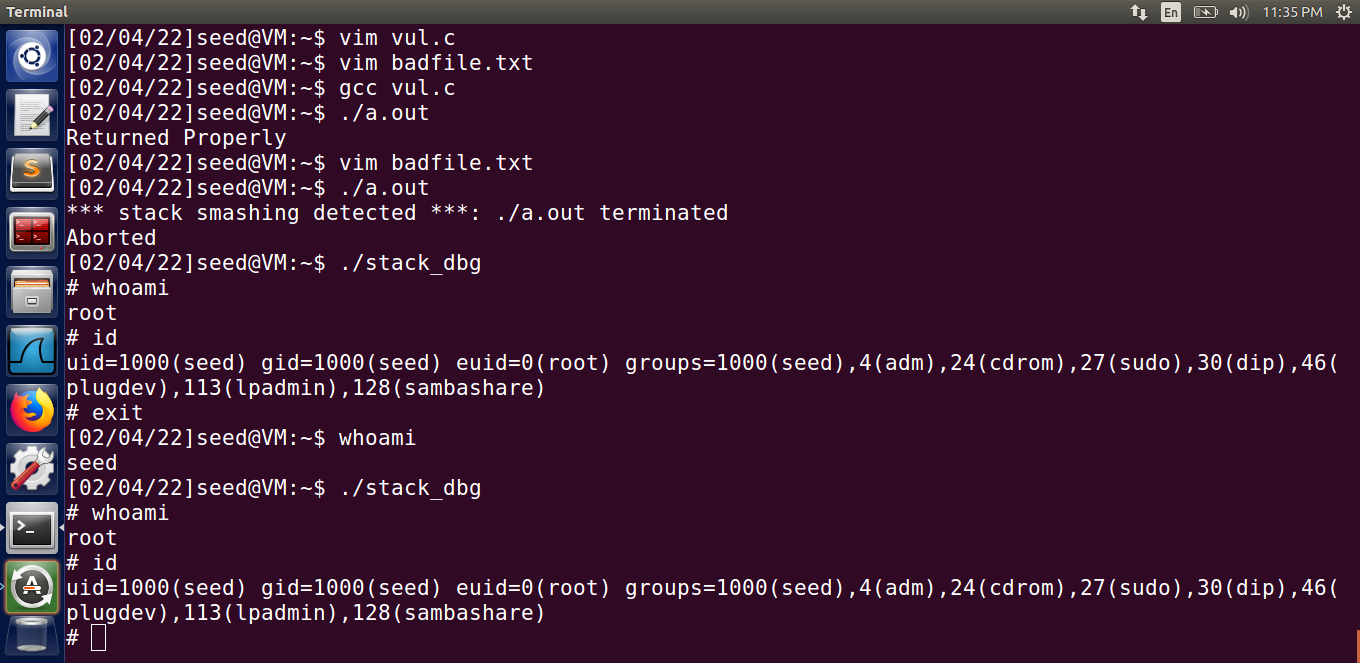




The bash shell has internal countermeasures to prevent buffer overflow attack. Thus we have to use a different shell to do this attack.

Linking /bin/sh to /bin/zsh





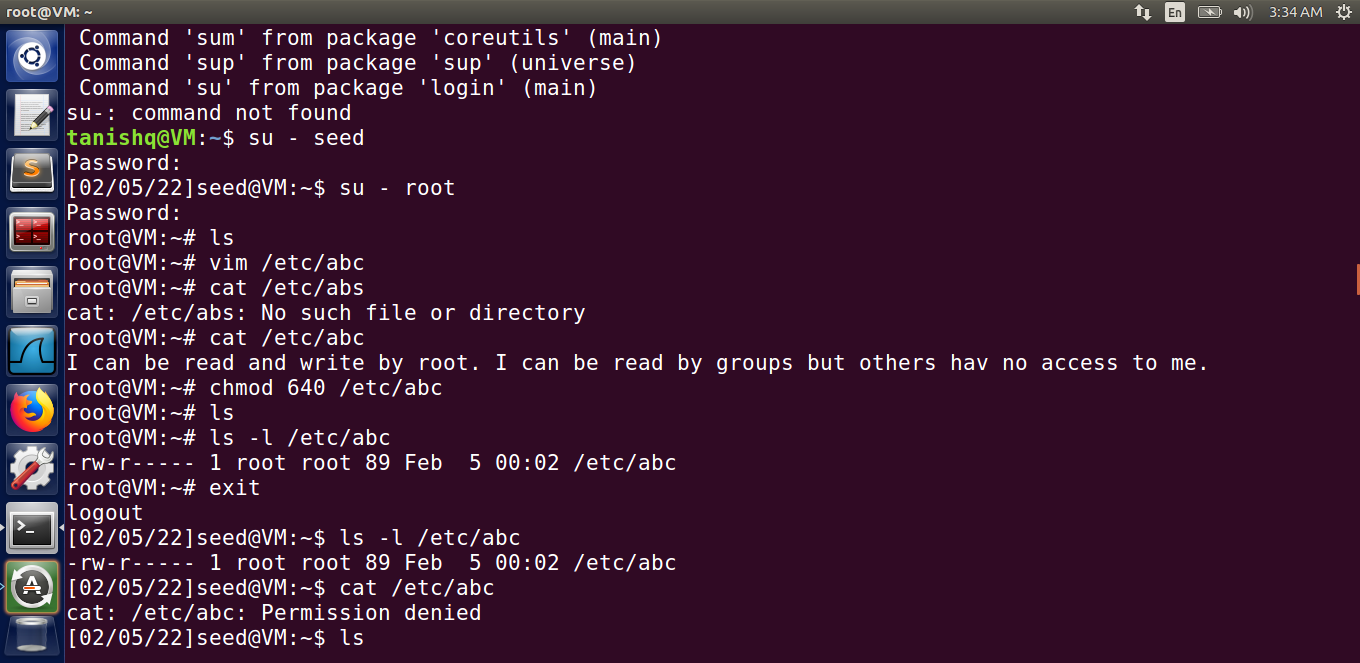
**Task (2):**

(1): Initially create "/etc/abc" file with permission so that only root can Read and write, Group can read and Others have no access. As a root user add some content into it. (10)

Let us break the task into:

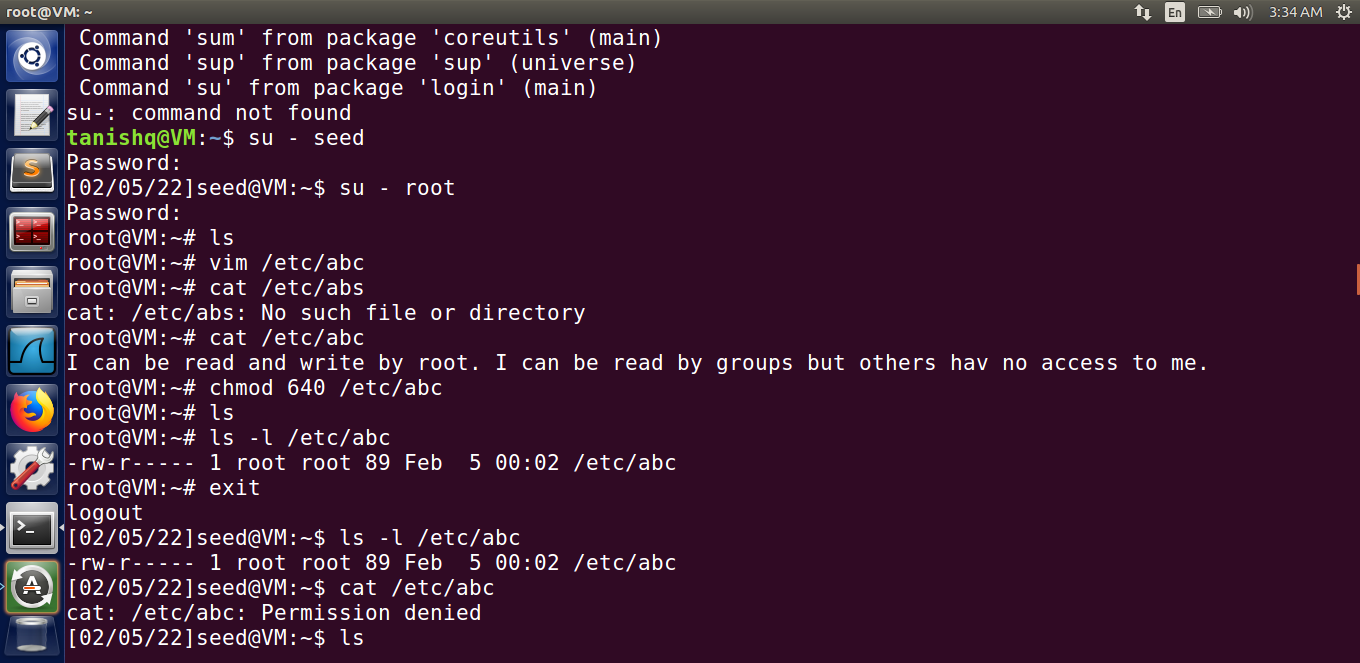
a) Logging in as root user

To create a file in etc, we need to be a root user. Logging in as root user.

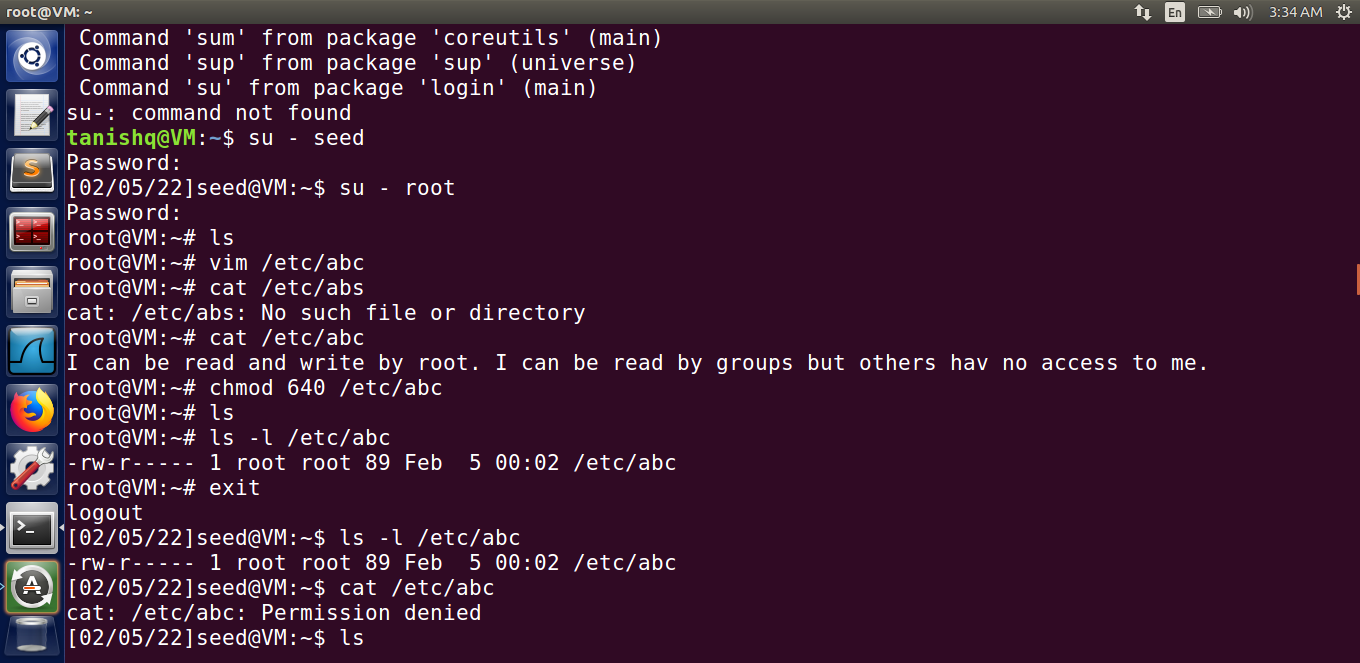


b) Creating /etc/abc file

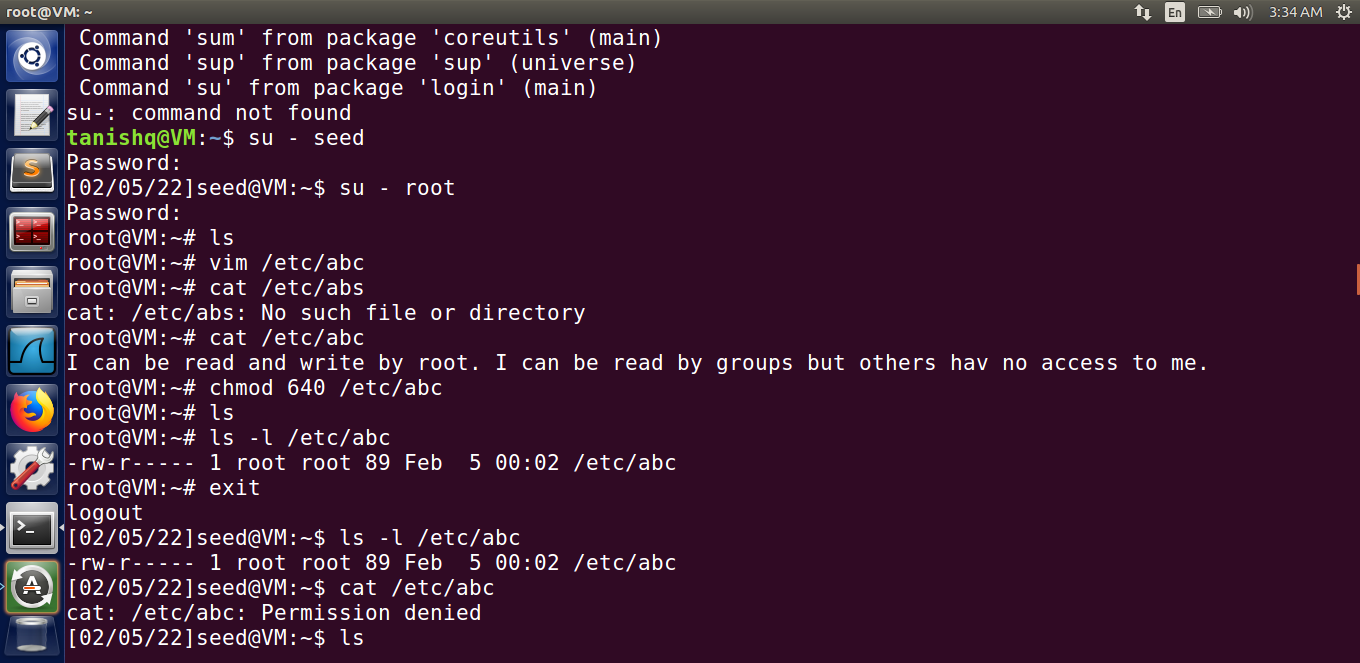
Creating the file using vim /etc/abc command and entering suitable content into the file. After creation of the file we can read it using cat /etc/abc since we are a root user.



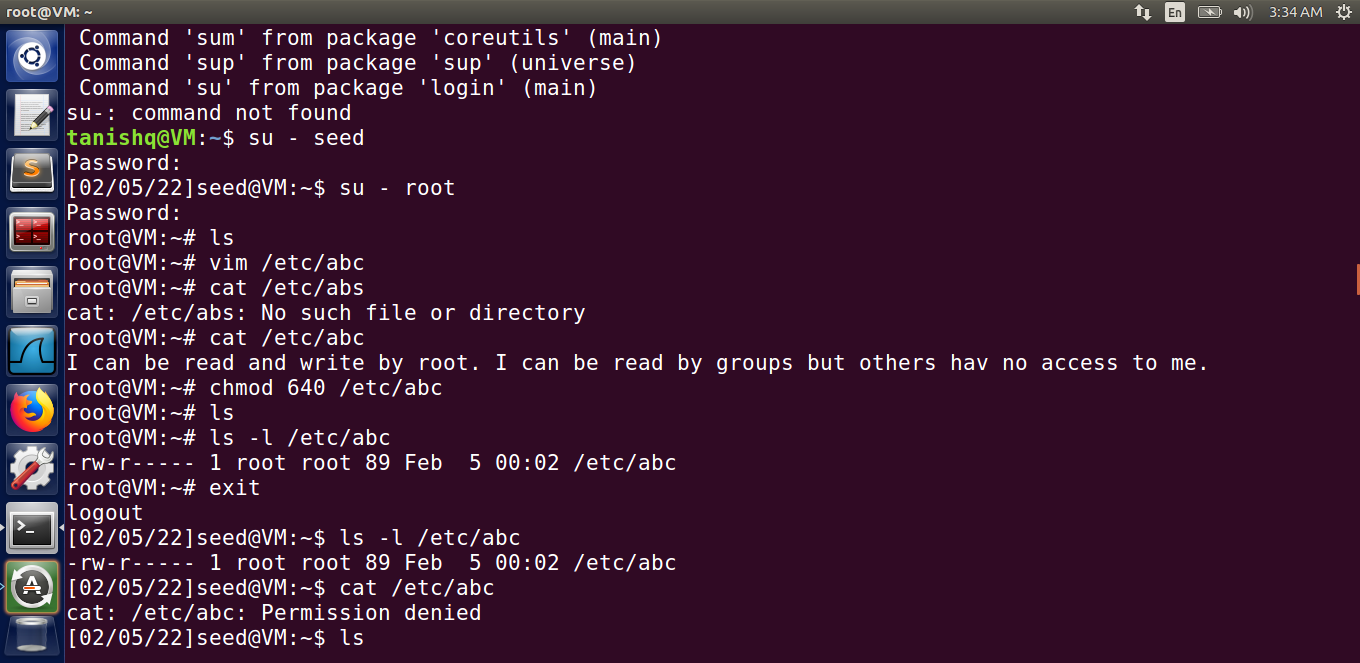
C) Granting permission as required in the question



D) Checking the file permission



We now try to read the file as a normal user, but since we have no permission we wont be able to read it, as can be seen below:



(2): Then switch to normal user and the contents of the badfile need to be prepared such that the malicious program can execute the following: -

execve("/bin/cat", argv,0)

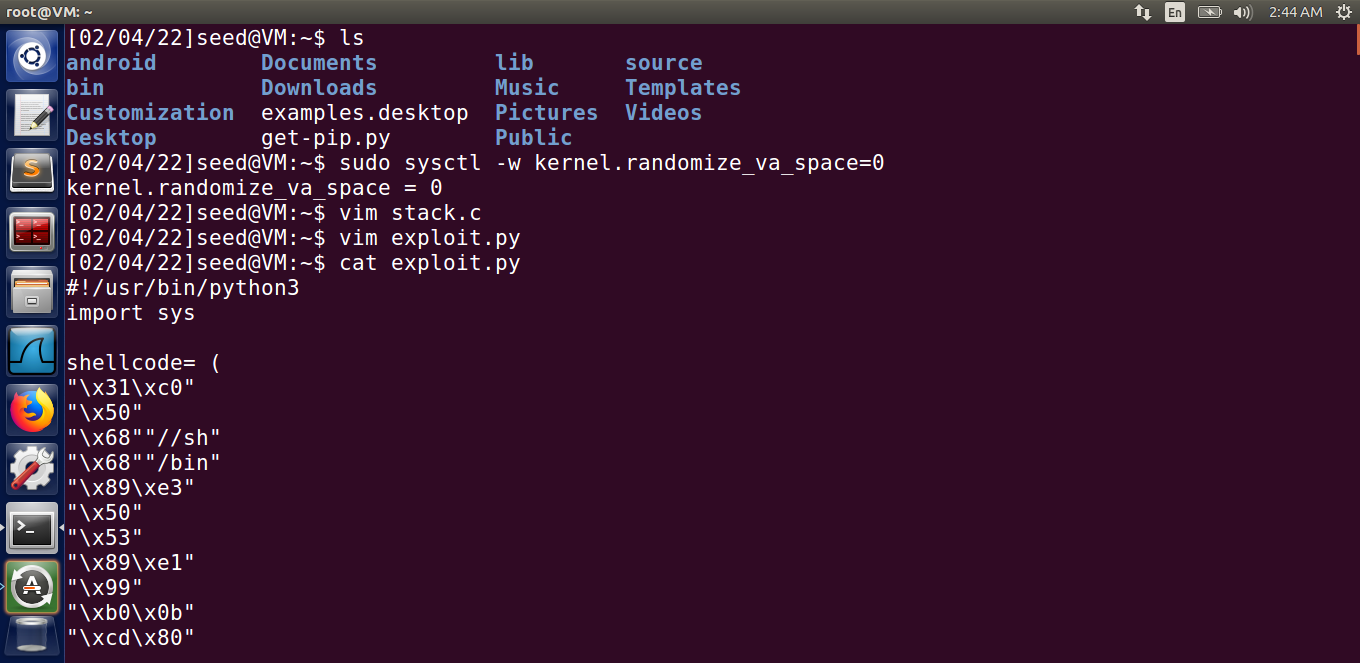
where argv[0]= the address of "bin/cat"

and argv[1]= "/etc/abc"

Thus, once the buffer overflow is successfully performed the normal user can read the contents of /etc/abc file. (20)

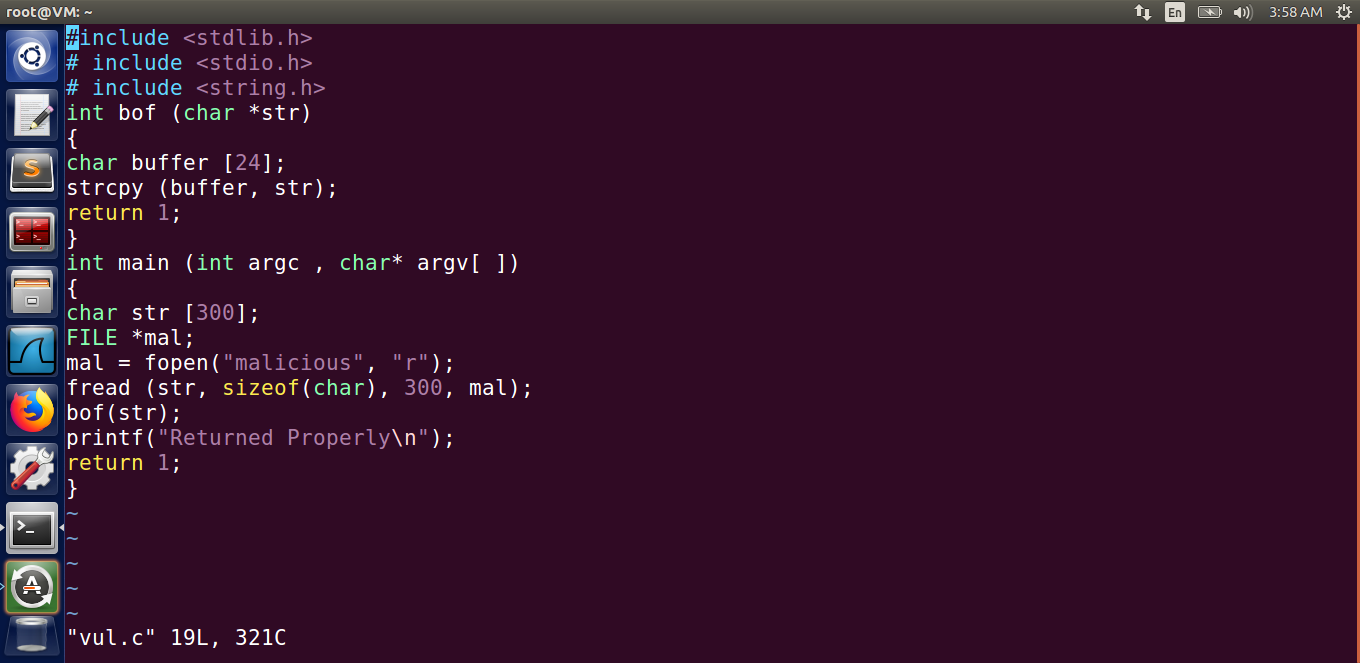
Let’s now try to exploit this vulnerability and try to run the above code so that we can see the /etc/abc file as normal user.

Firstly let’s turn off the address randomization. It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for us to guess the address of the injected malicious code.



To make a successful buffer overflow attack and to read the /etc/abc file, follow these steps:

Step1: Vulnerable code

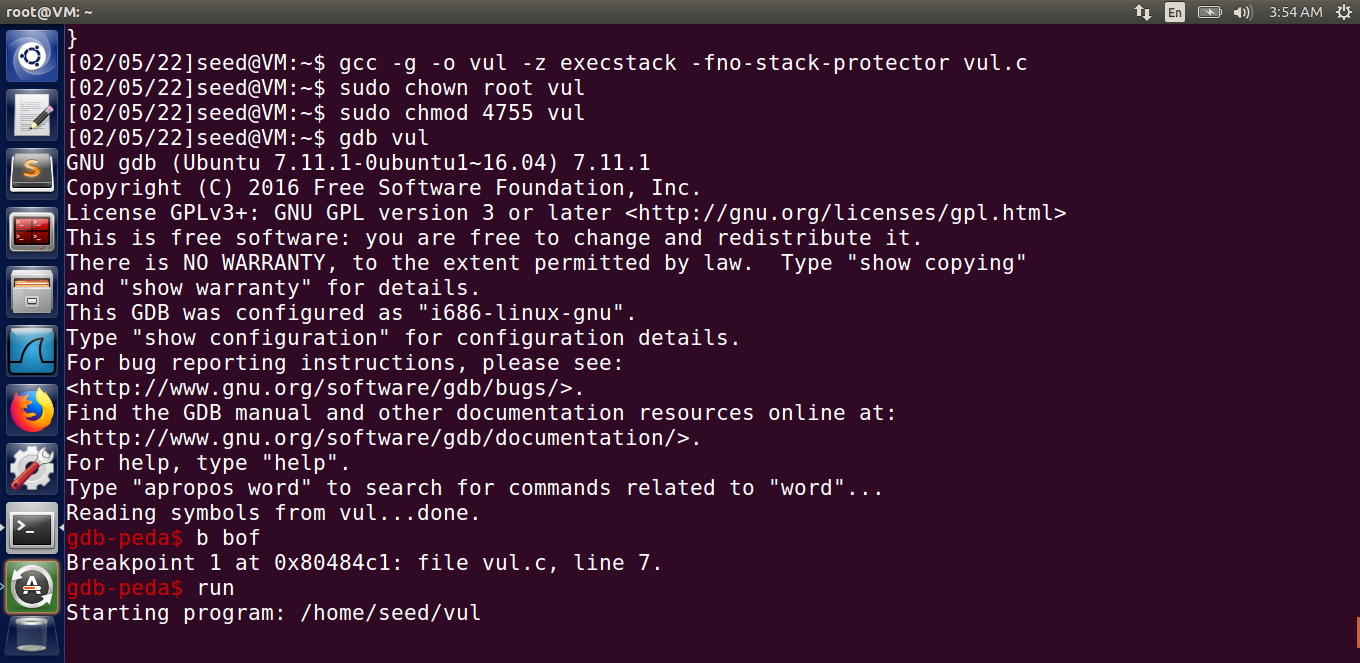


Creating a vulnerable code similar to previous vulnerable code to exploit buffer overflow attack.

If a buffer overflow vulnerability can be exploited in a privileged Set – UID root program, the injected malicious code, if executed, can run with the root's privilege, we thus change the ownership and make it a set-uid program

Note: Countermeasures like execstack and stack needs to be switched off so that we can successfully launch our attack.

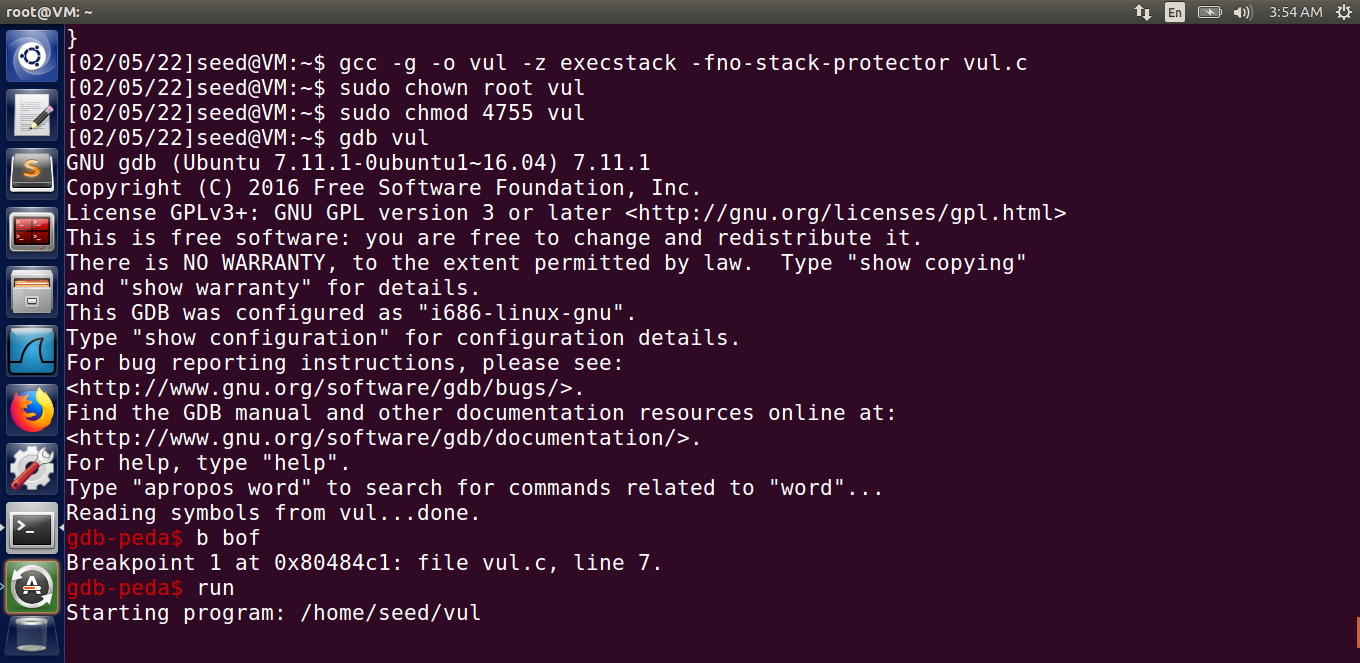
Compiling the code:

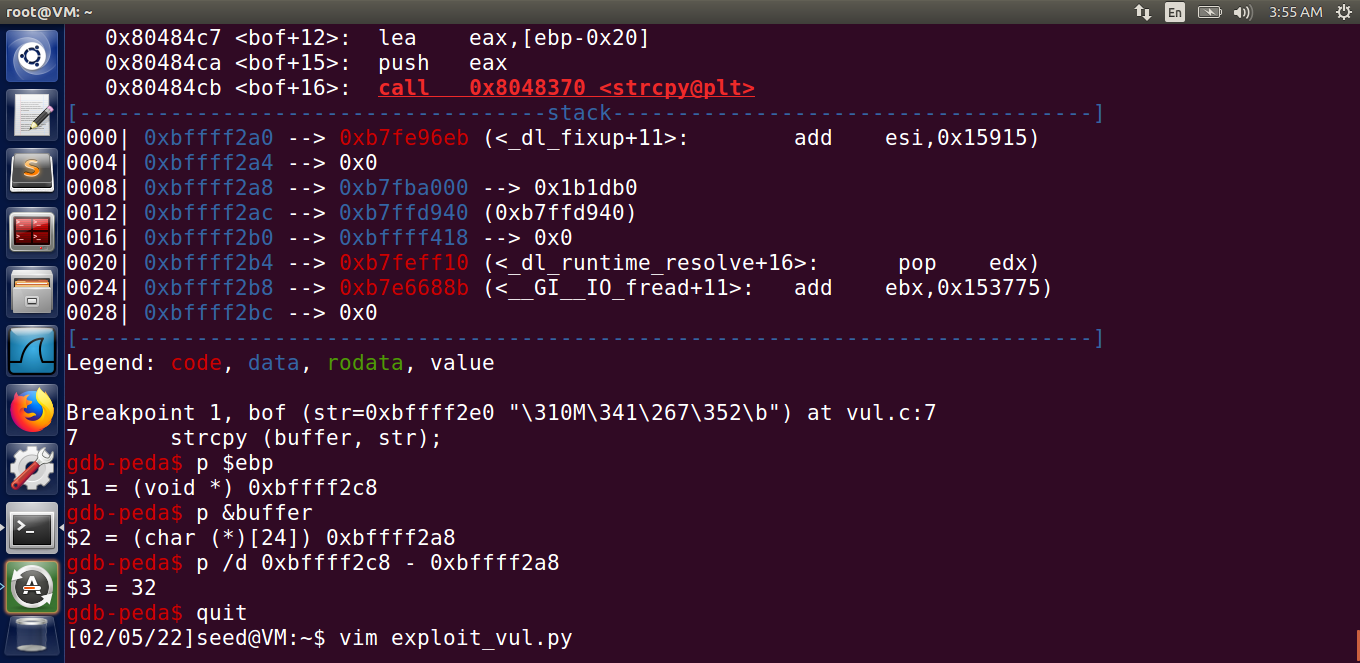


Step 2: Distance Between Buffer Base Address and Return Address

This can be done by using the gdb debugger tool and setting a breakpoint at our bof function. Once we know the frame pointer of bof, we can easily locate our malicious code with positive offset to this pointer.

Launching the GDB debugger tool to get the addresses:





Distance between them

Getting the base address of buffer

Getting the address of ebp

We now know the distance between the base address and the ebp pointer. The return address to previous function is placed just above eb pie ebp +4. Thus we also get the address of ‘return address’. Thus we can easily place our intended address of malicious code at this address.

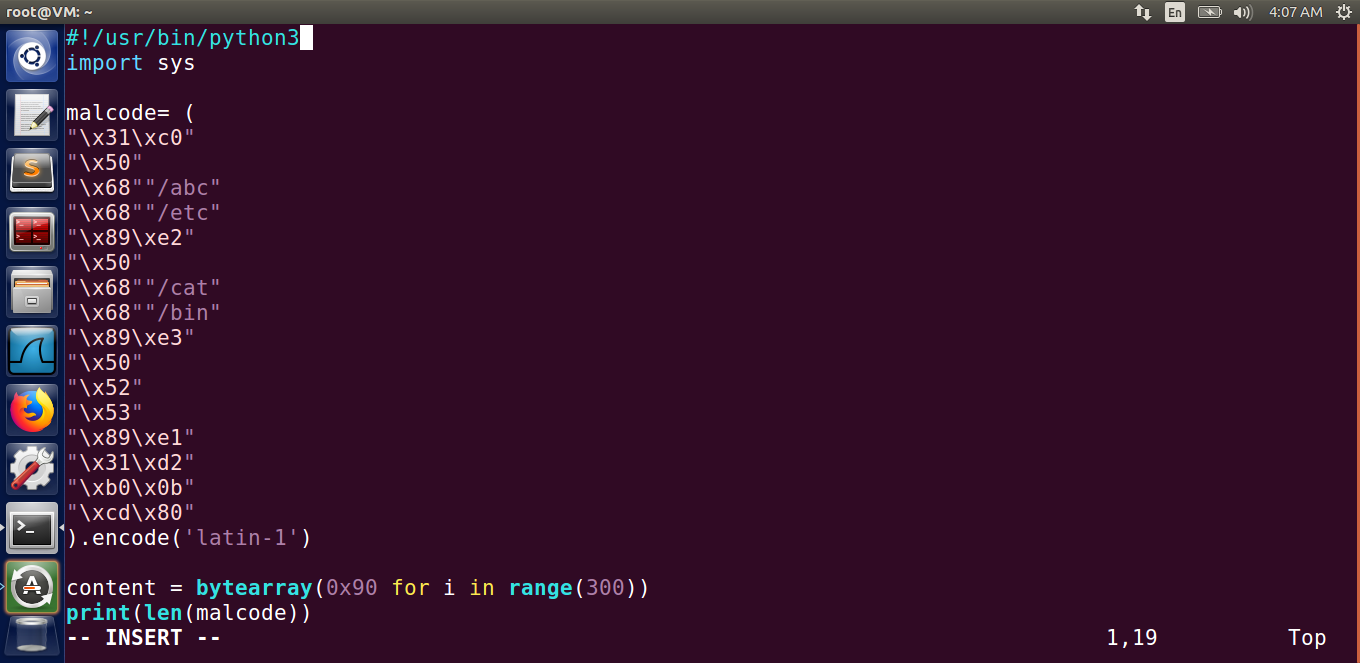
Step 2: Badfile construction

Proper creation of badfile is the most important step. If the badfile is not created properly and the return address is not mentioned clearly we might not reach to correct address of our malicious code and this could lead to failure of attack.

However we can overcome this problem by filling our badfile NOP instruction. This results in creation of multiple entry point for our malicious code and thus even if our return address points to address before our malicious code it will eventually get executed.

N-Op instruction are represented by 0x90 in assembly language and we will use it our badfile construction.

To create a badfile we will write a python script.



Our syntax to run execve:-

execve("/bin/cat", ["/bin/cat","/etc/abc", NULL], NULL)

Explaination:

malcode= (

"\x31\xc0" # xorl %eax , %eax

"\x50" # pushl %eax

"\x68""/abc" # pushing string /abc in stack

"\x68""/etc" # pushing string /etc in stack

"\x89\xe2" # movl %esp, %edx

"\x50" # pushl %eax

"\x68""/cat" # push string /cat in stack

"\x68""/bin" # push string /bin in stack

"\x89\xe3" # movl %esp, %ebx

"\x50" # pushl %eax # ( 0 )

"\x52" # pushl %edx # ( “/etc/abc”)

"\x53" # pushl %ebx # ( “/bin/cat” )

"\x89\xe1" # movl %esp, %ecx

"\x31\xd2" # xorl %edx, %edx

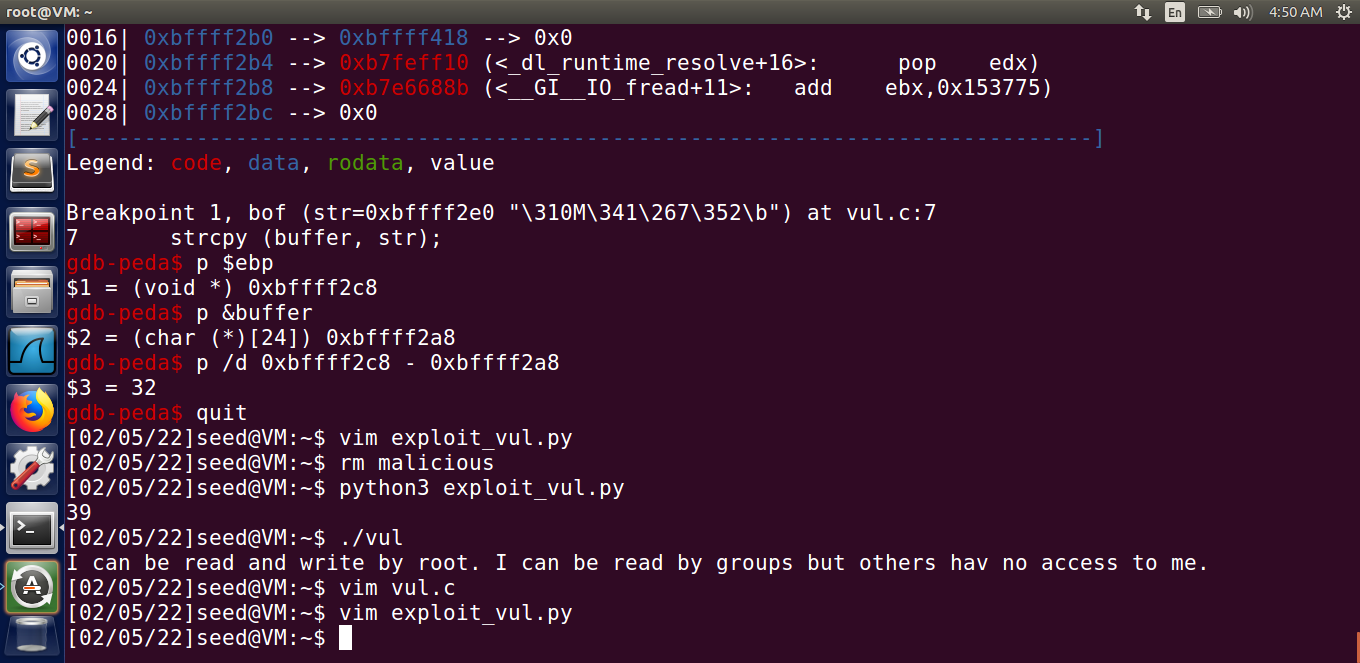
"\xb0\x0b" # movb $0x0b , %al

"\xcd\x80" # int $0x80

).encode('latin-1')

The later part of the code simply sets the return address and fills the malcode into the bad file. Also this return address should be placed at the difference which we got by subtracting ebp address and buffer base address.

After successfully running the python script we can create our malicious file. We can then run our vulnerable code to read the content of etc/abc file which was only readable to root and group.



Creating the malicious file. (output 39 here represents length of assemble cody (just for debugging purpose))

We can read a non-readable file !!

Thus, using buffer overflow attack, we are successfully able to read the files via normal user.

----------------------------------------------The End------------------------------------------------