Software Security Lab Report

for

Buffer Overflow Attack (Set-UID Version)

Submitted To

Md. Iftekharul Alam Efat
Assistant Professor
Institute of Information Technology (IIT)
Noakhali Science and Technology University

Submitted by

Ishrat Jahan Rintu (BFH1925002F)
Prosanto Deb (ASH1925005M)
Abdullah Alif (ASH1925009M)
Md. Alamgir Hossain (ASH1925016M)
Md. Rokonuzzaman (ASH1925018M)

Date of Submission: 28/02/2023

Table of Contents

Table of Figures

1. Introduction

1.1 Program Memory Layout

To fully understand how buffer overflow attacks work, we need to understand how the data memory is arranged inside a process. When a program runs, it needs memory space to store data.

- 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.
- 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.

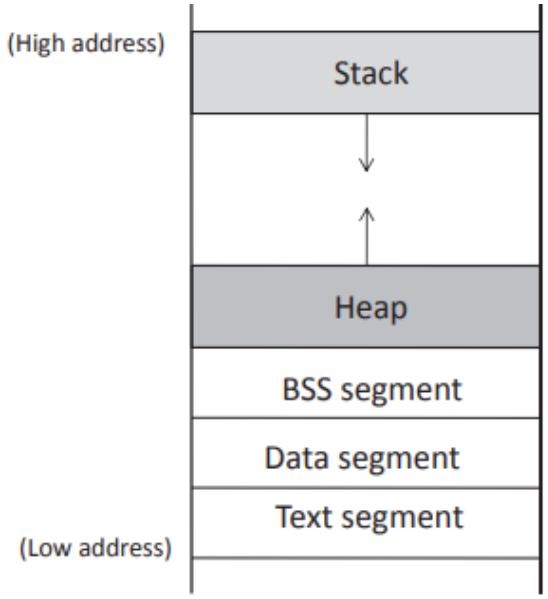

Figure 1 Program Memory Layout

1.2 Stack and Function Invocation

Buffer overflow can happen on both stack and heap. But in this lab we focus on the stack based buffer overflow.

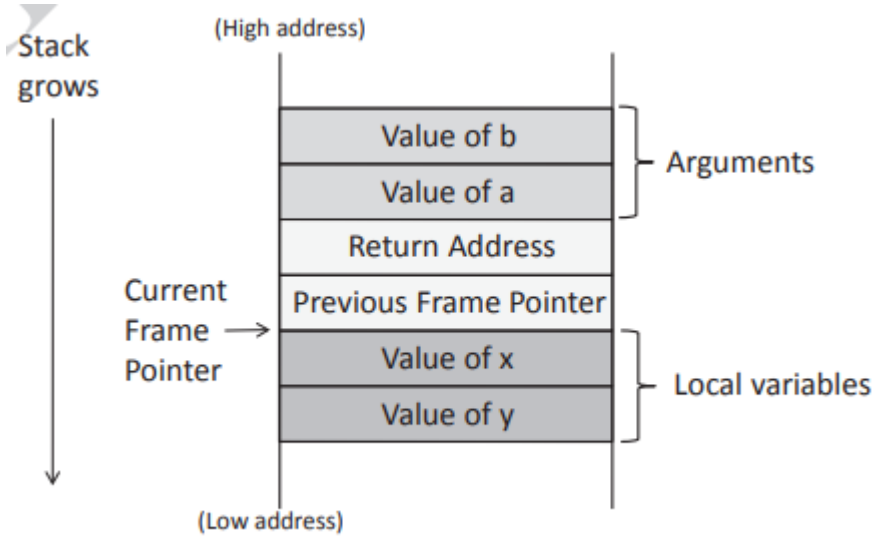

Figure 2 Layout for a function's stack frame

Figure 3 C Program for Demonstration

1.3 Buffer Overflow

As can be seen in Figure 4, the region above the buffer includes critical values, including the return address and the previous frame pointer. The return address affects where the program should jump to when the function returns. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place. Several things can happen.

- First, the new address, which is a virtual address, may not be mapped to any physical address, so the return instruction will fail, and the program will crash.
- Second, the address may be mapped to a physical address, but the address space is protected, such as those used by the operating system kernel; the jump will fail, and the program will crash.
- Third, the address may be mapped to a physical address, but the data in that address is not a valid machine instruction (e.g. it may be a data region); the return will again fail and the program will crash.
- Fourth, the data in the address may happen to be a valid machine instruction, so the program will continue running, but the logic of the program will be different from the original one.

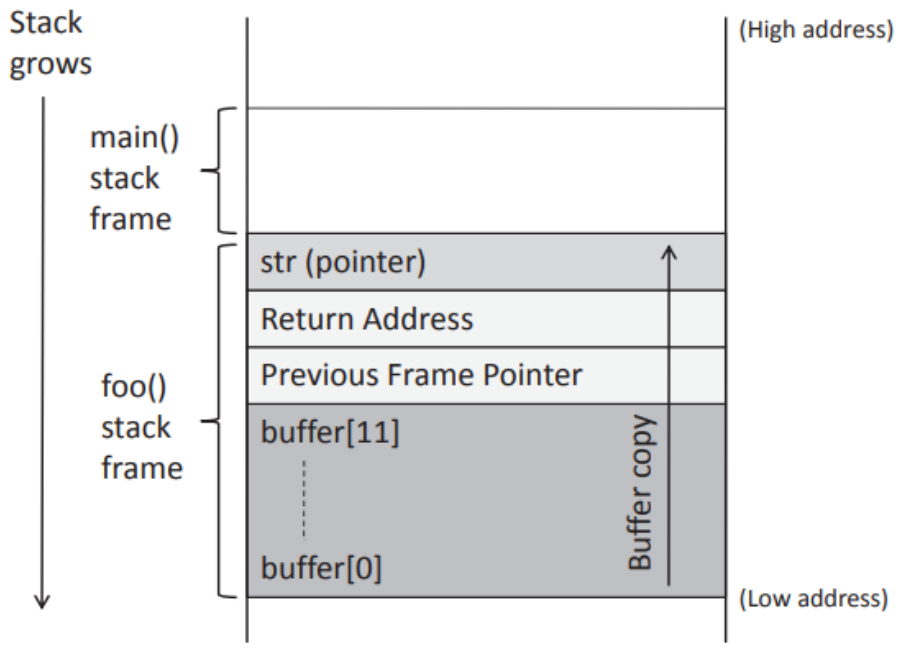

Figure 4 Buffer Overflow

1.4 Exploiting a Buffer Overflow Vulnerability

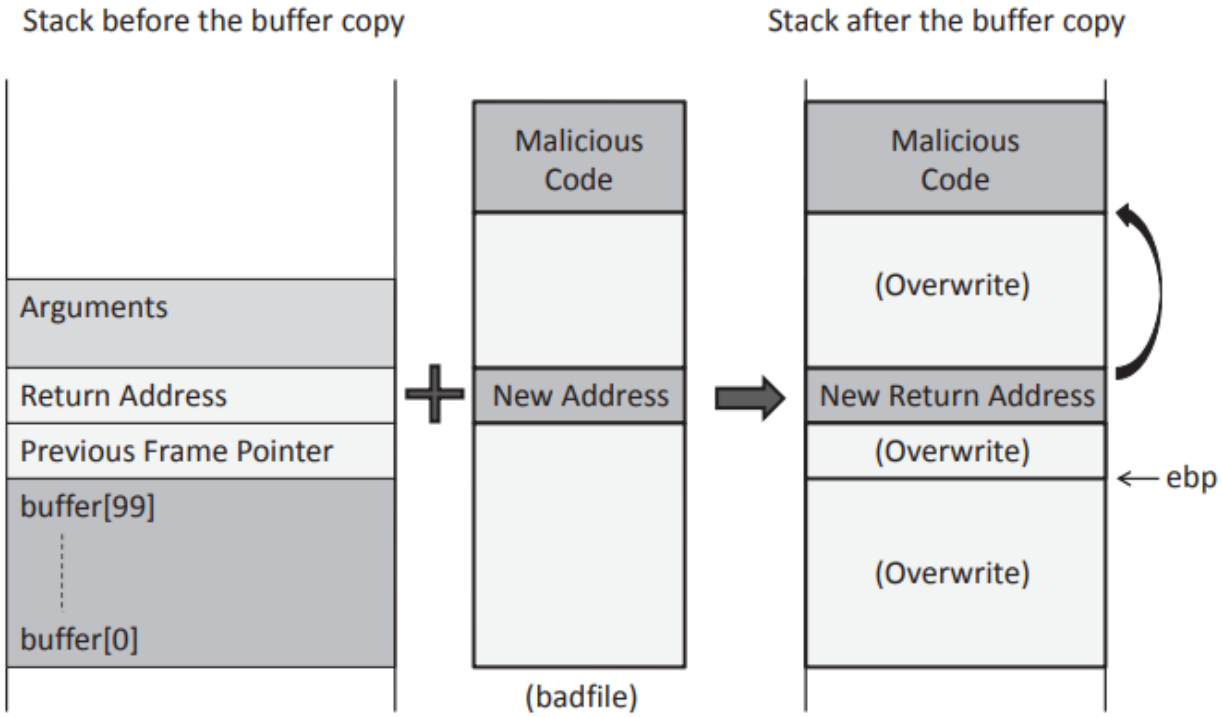

Figure 5 Insert and Jump to malicious code

2. Setup for Our Experiment

2.1 Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first.

2.1.1 Address Space Randomization

Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks.

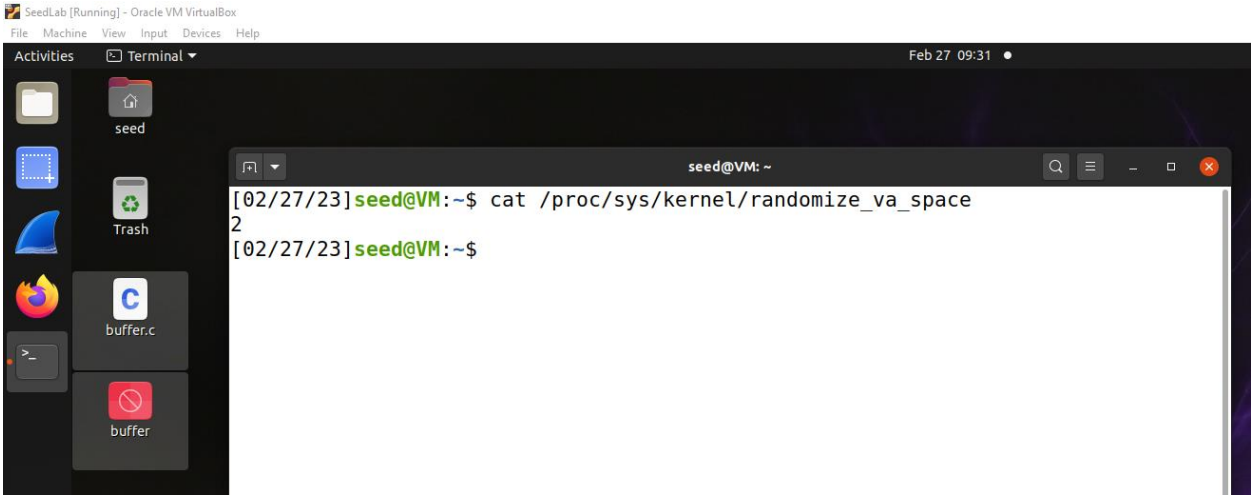

Figure 6 Address Space Randomization

Figure 7 Checking Address Space Randomization

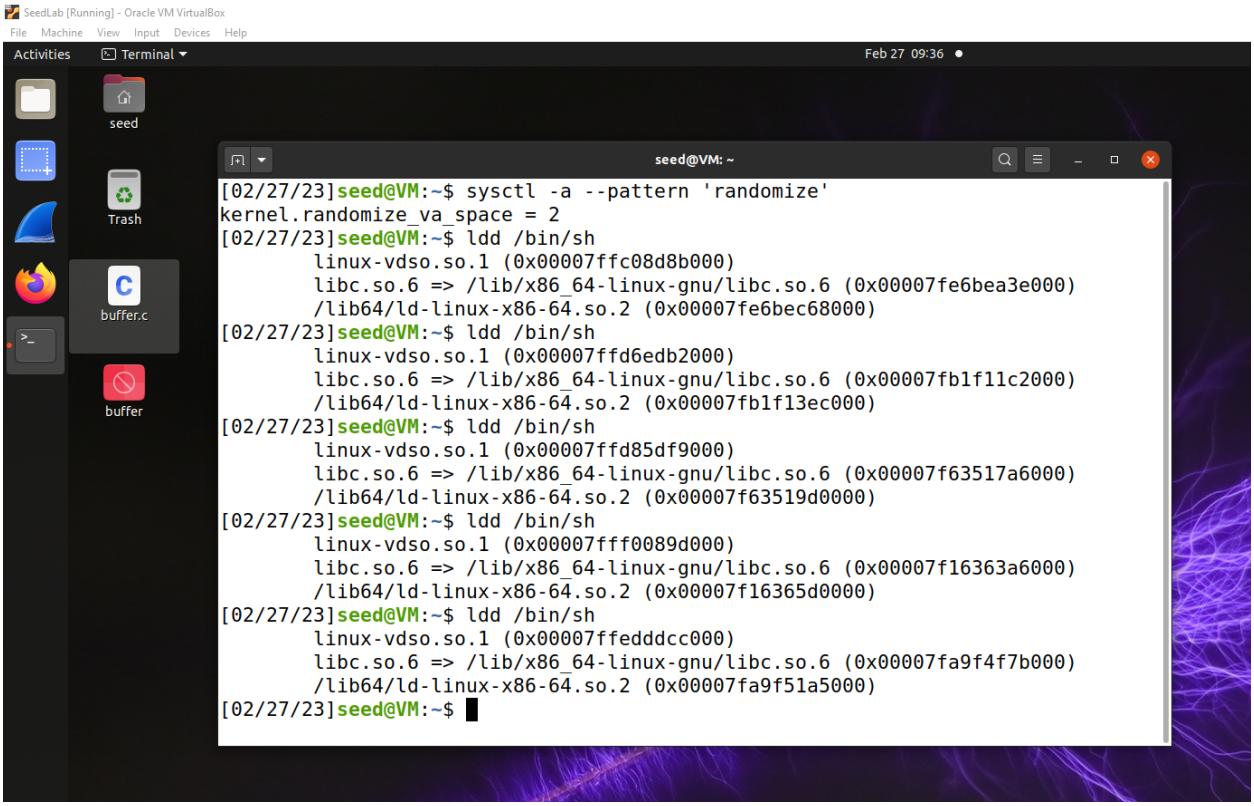

Figure 8 Random Addresses

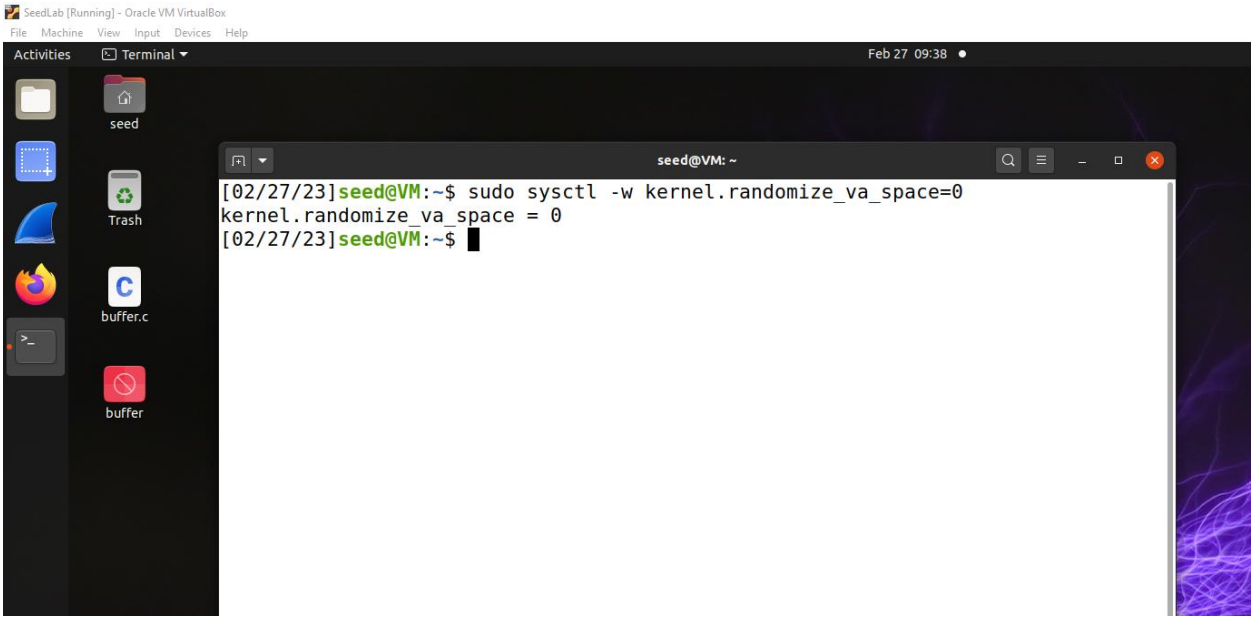

Figure 9 Turning Off Random Addresses

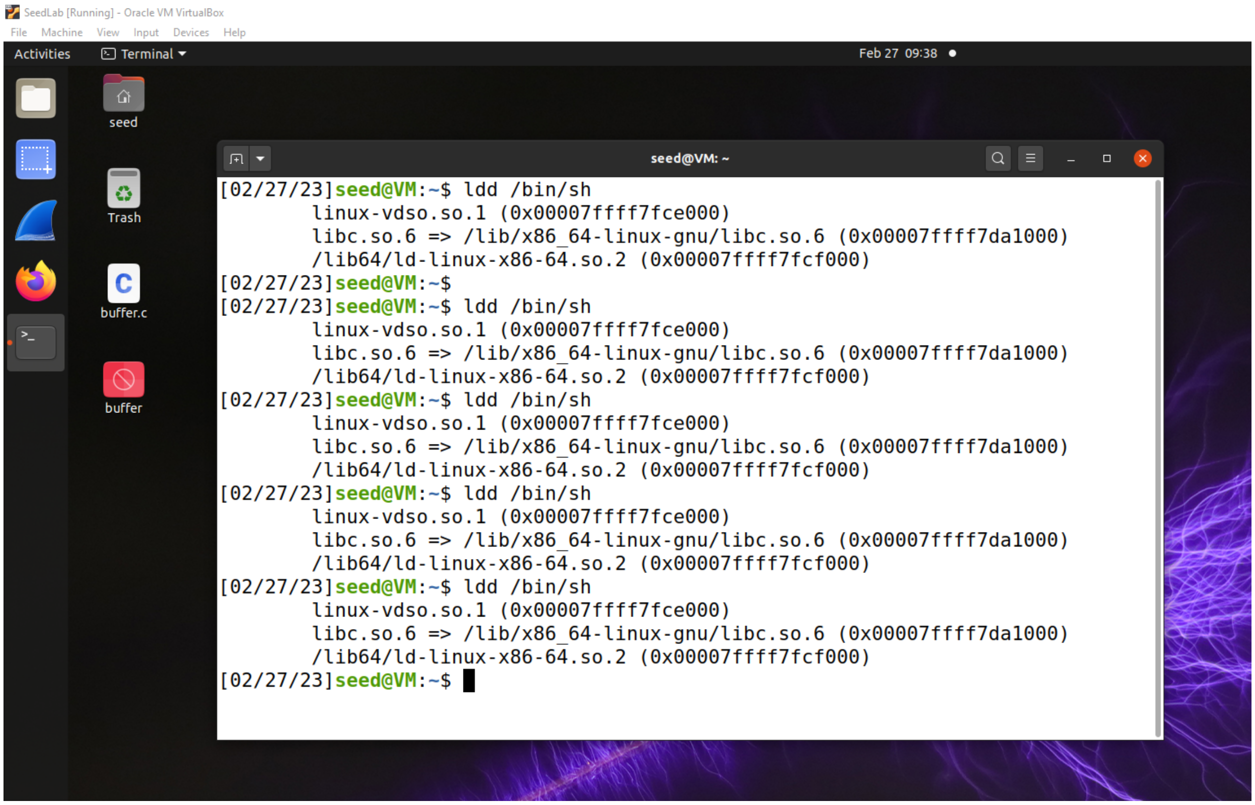

Figure 10 Same Addresses

2.1.2 Configuring /bin/sh

In the recent versions of Ubuntu OS, the /bin/sh symbolic link points to the /bin/dash shell. The dash program, as well as bash, has implemented a security countermeasure that prevents itself from being executed in a Set-UID process. Basically, if they detect that they are executed in a Set-UID process, they will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in /bin/dash can be easily defeated). We have installed a shell program called zsh in our Ubuntu 20.04 VM. The following command can be used to link /bin/sh to zsh:

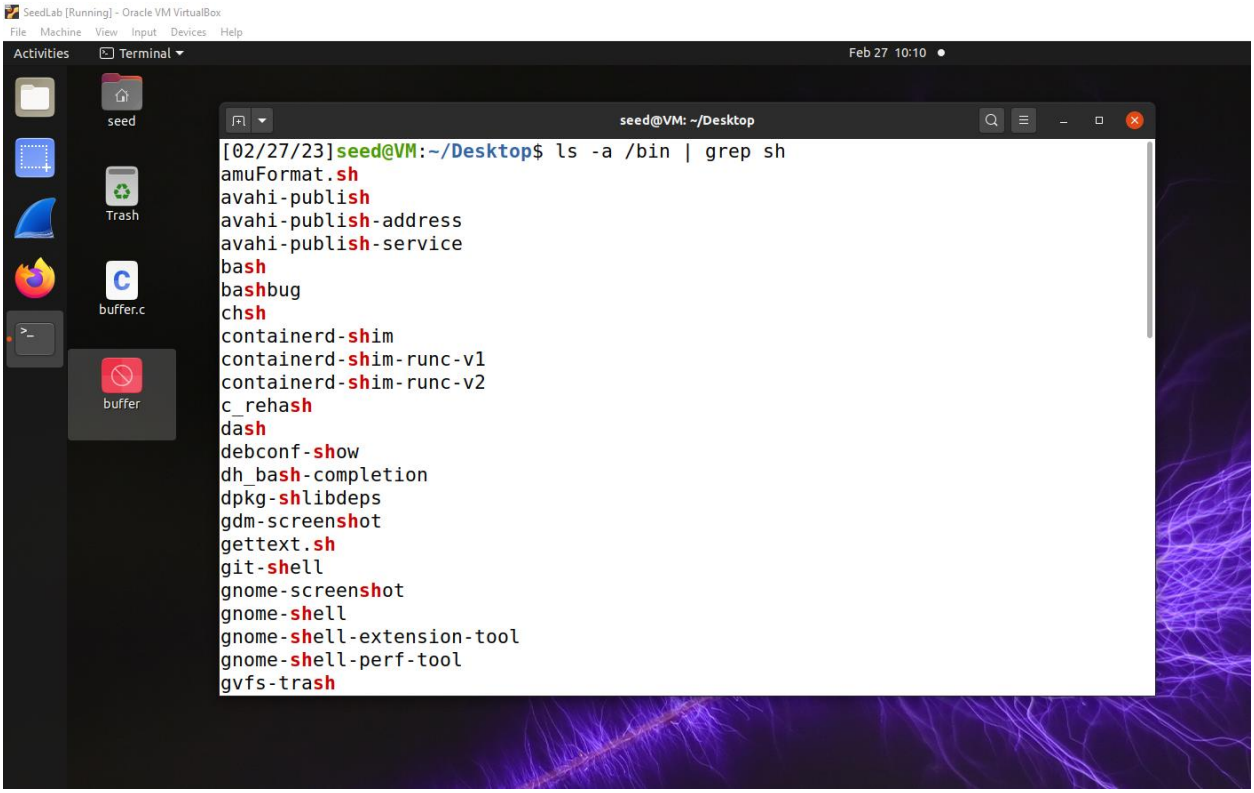

Figure 11 Configuring /bin/sh

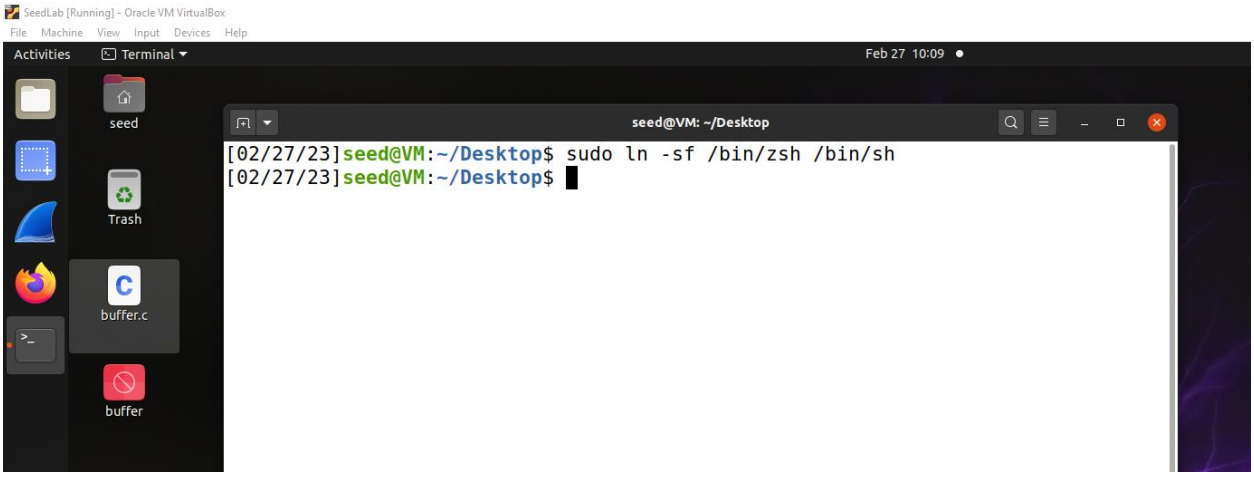

Figure 12 Configuration

3. Getting Familiar with Shellcode

The ultimate goal of buffer-overflow attacks is to inject malicious code into the target program, so the code can be executed using the target program's privilege. Shellcode is widely used in most code-injection attacks. Let us get familiar with it in this task.

3.1 The C Version of Shellcode

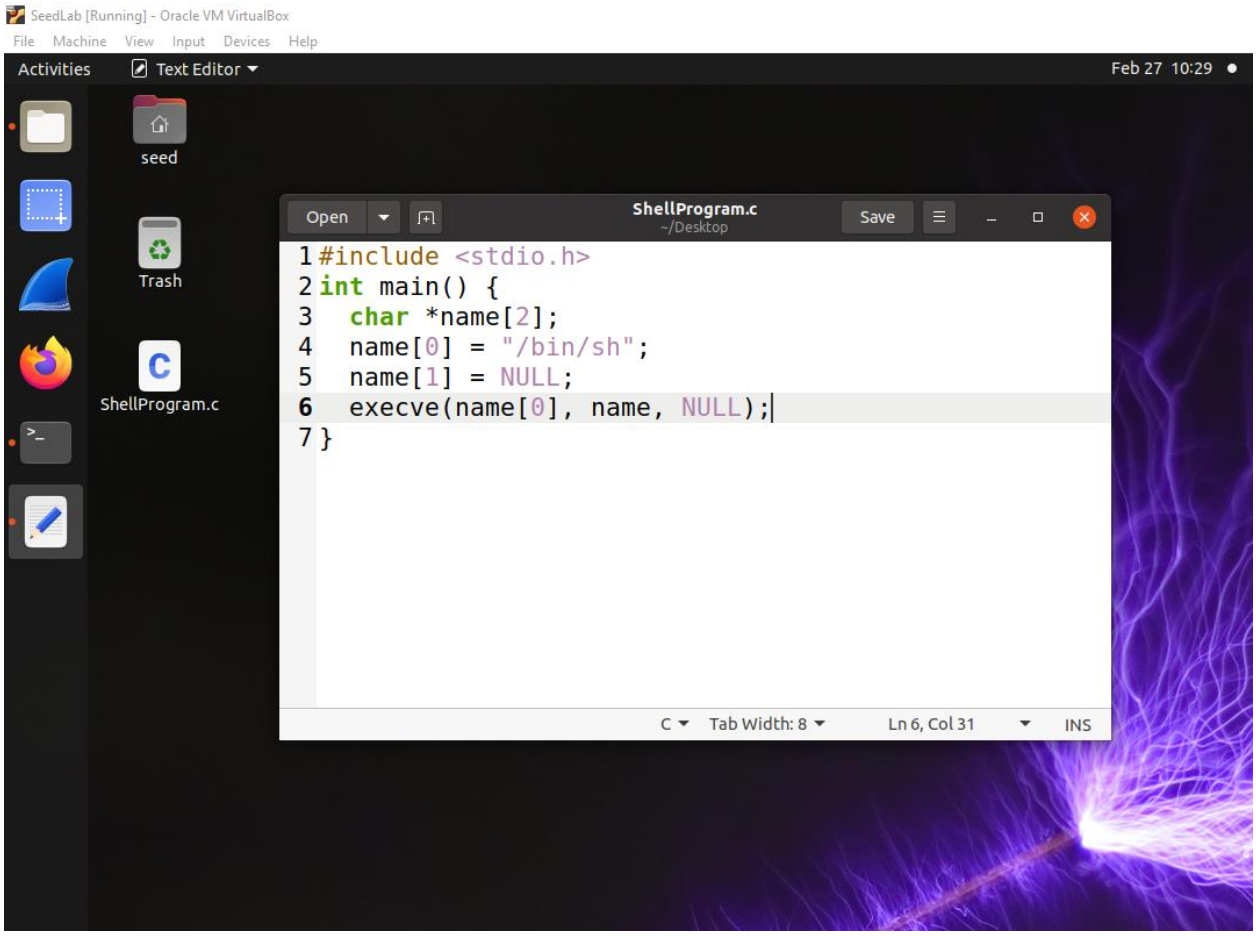

Figure 13 C Version of Shell Code

Unfortunately, we cannot just compile this code and use the binary code as our shellcode. The best way to write a shellcode is to use assembly code. In this lab, we are provided the binary version of a shellcode.

3.2 32-bit Shellcode

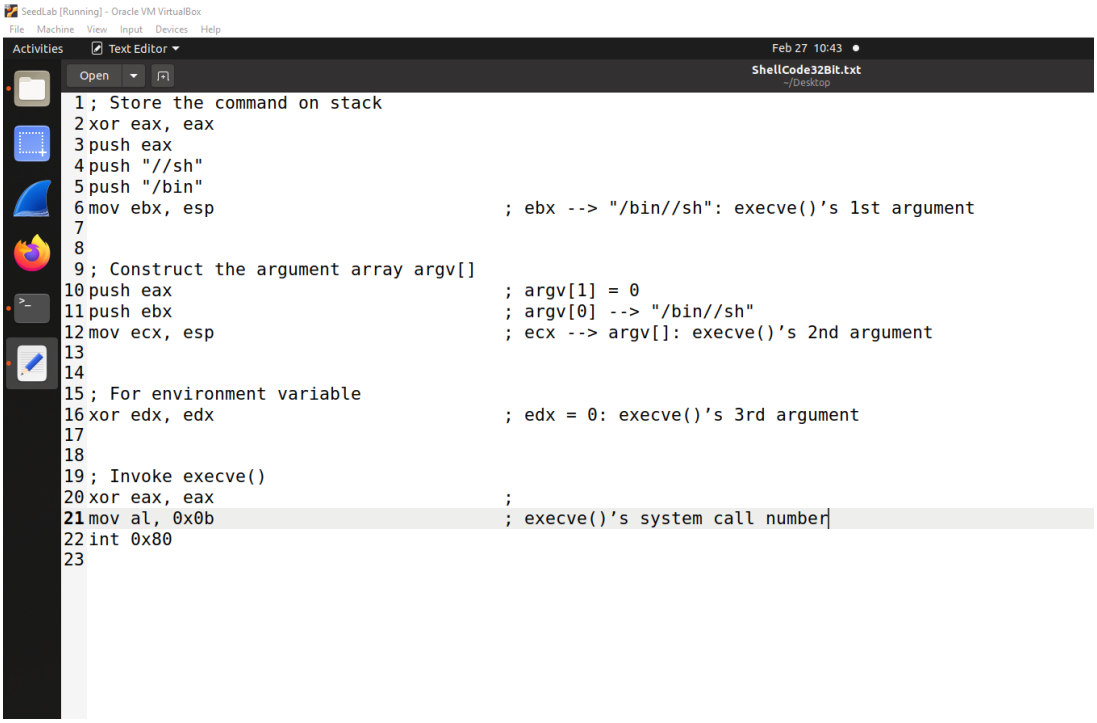

Figure 14 32-bit Shellcode

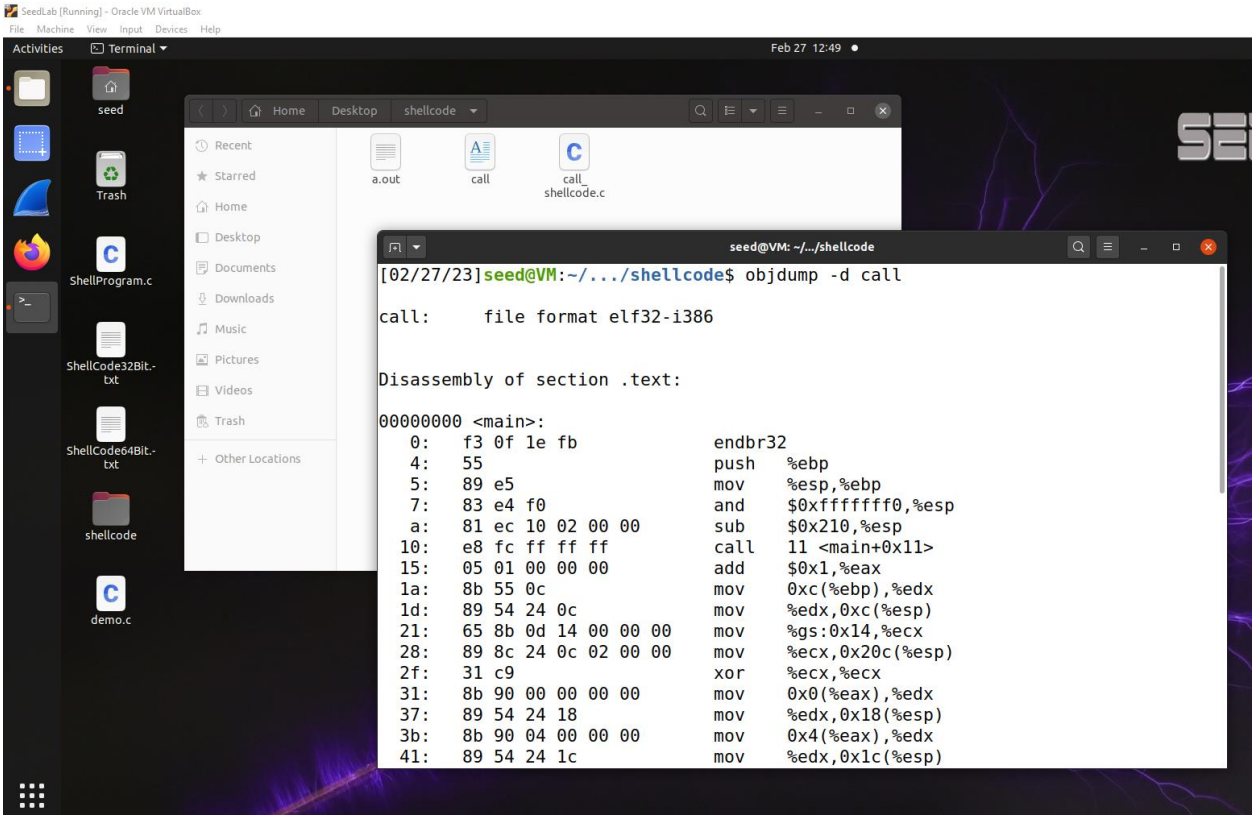

Figure 15 Assembly Language output

3.3 64-bit Shellcode

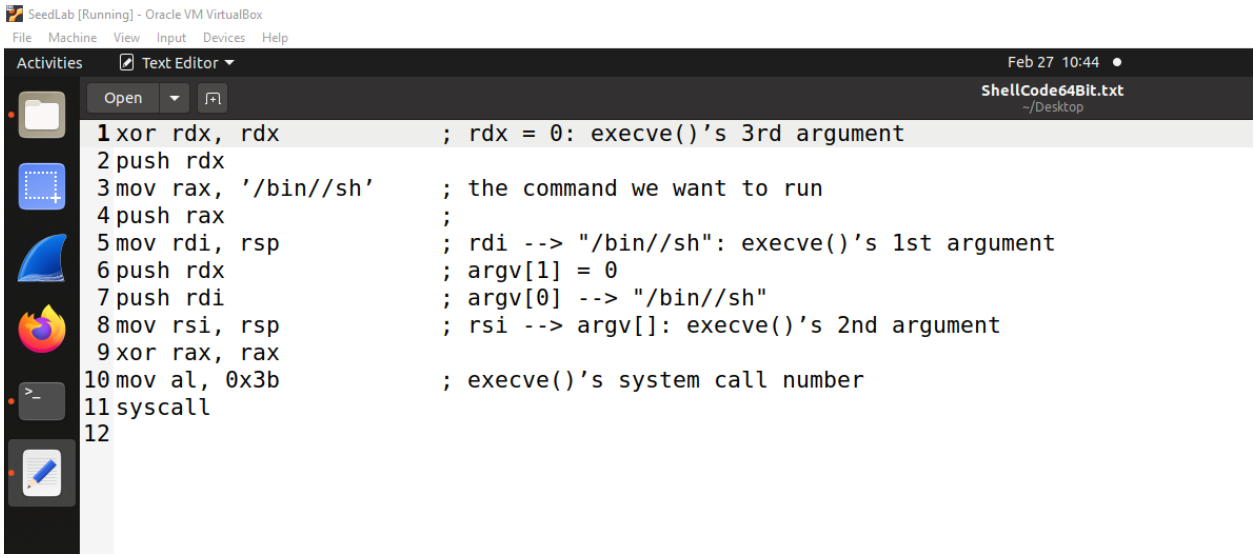

Figure 16 64-bit Shellcode

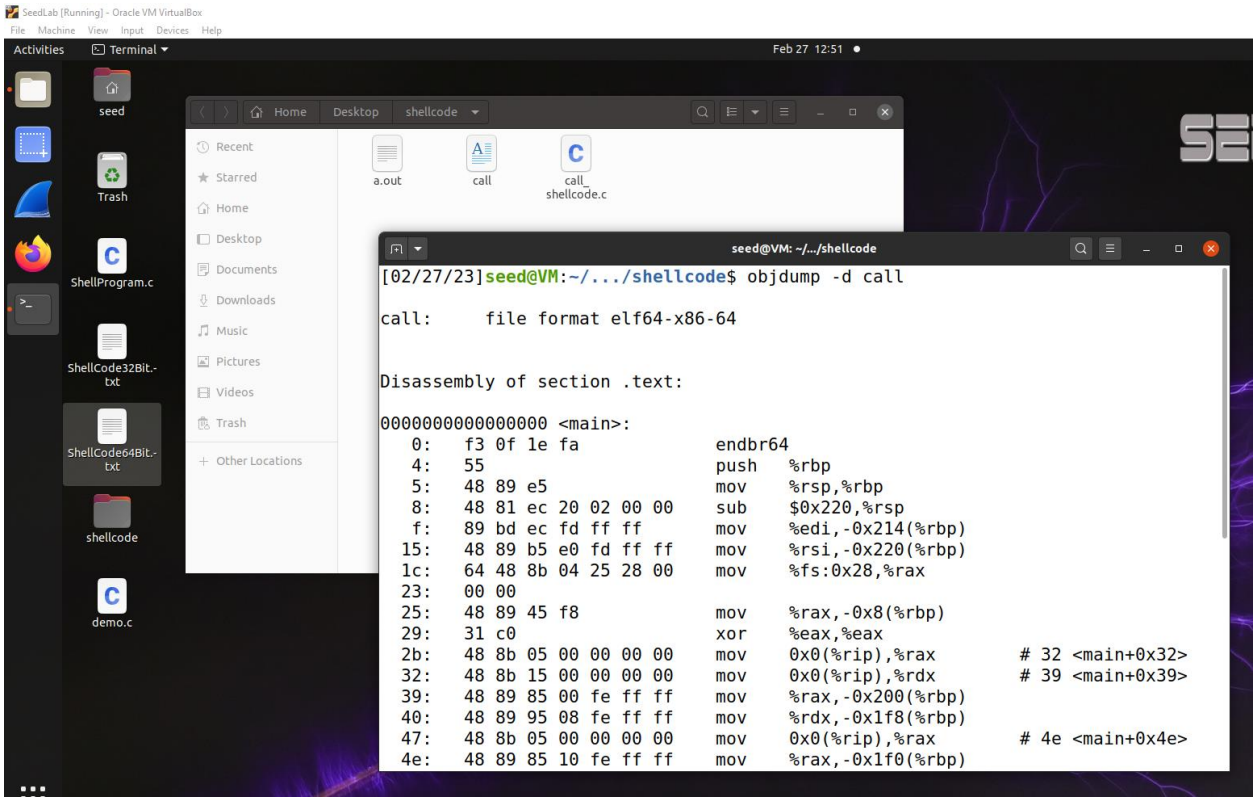

Figure 17 Assembly Language Output

3.4 Invoking the Shellcode

In the given document, we have found the binary code from the assembly code above, and we have put the code in a C program called call shellcode.c inside the shellcode folder.

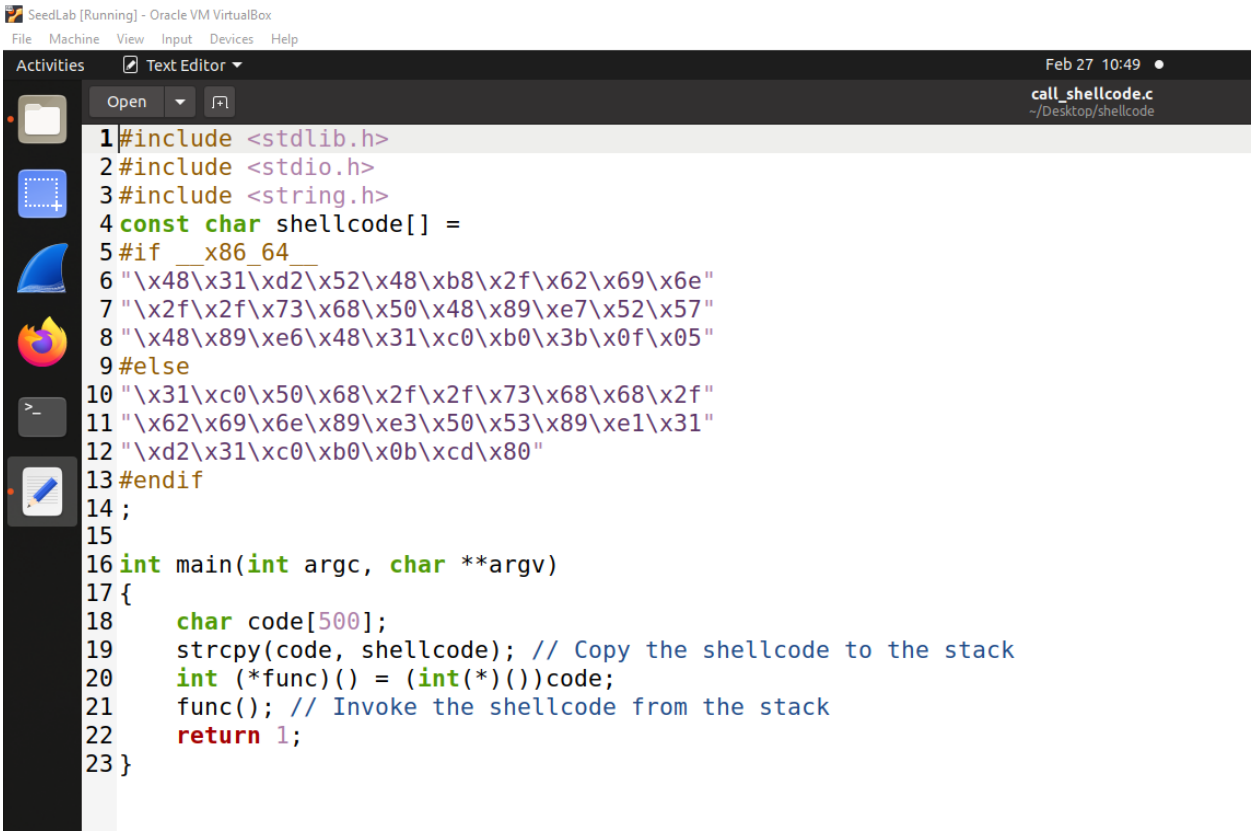

Figure 18 C Program with binary code

4. Understanding the Vulnerable Program

The vulnerable program used in this lab is called stack.c, which is in the code folder. This program has a buffer-overflow vulnerability, and our job was to exploit this vulnerability and gain the root privilege.

The above program has a buffer overflow vulnerability. It first reads an input from a file called badfile, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() is only BUF SIZE bytes long, which is less than 517. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a root-owned Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its

input from a file called badfile. This file is under users' control. Now, our objective is to create the contents for badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

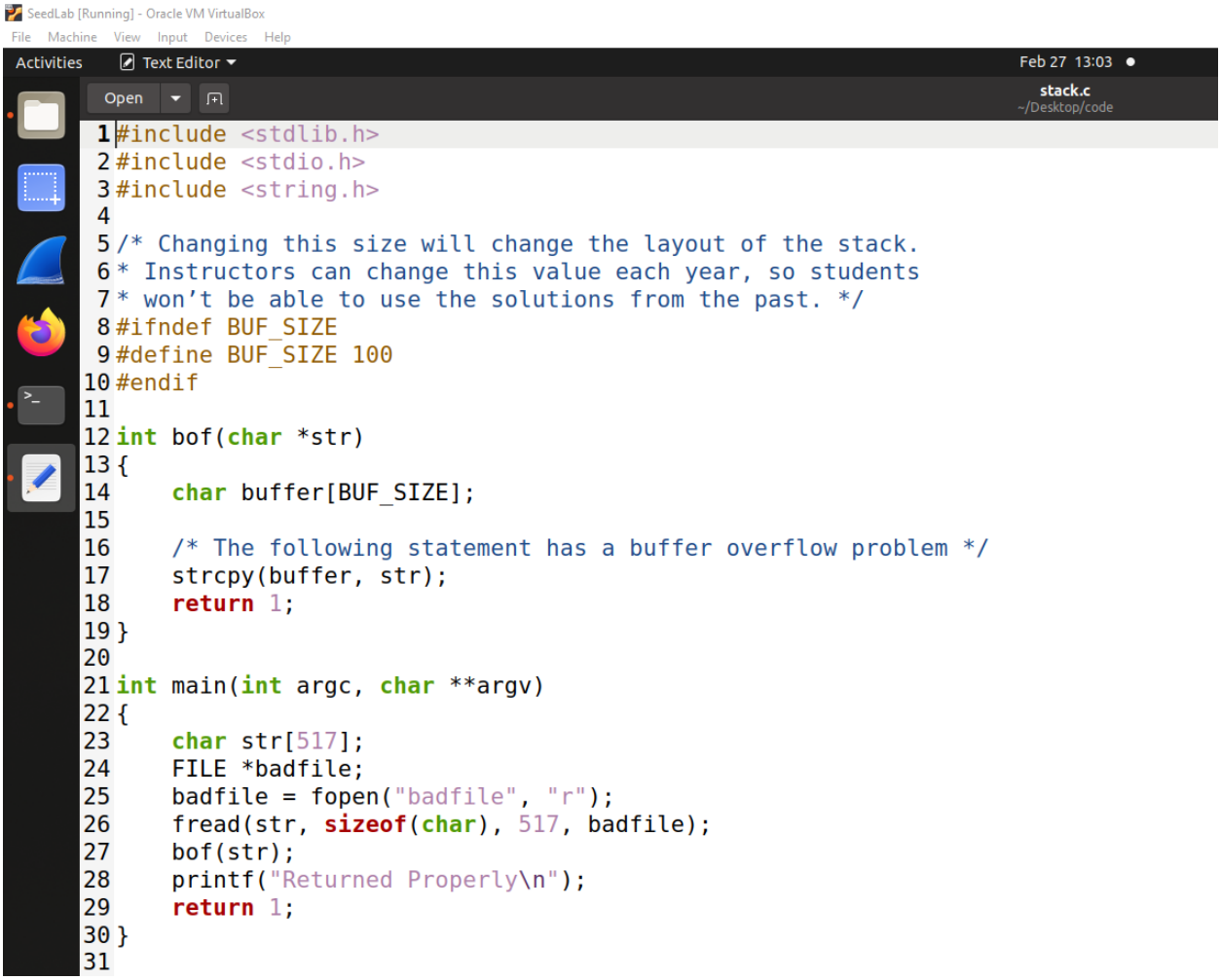

Figure 19 Given C Program

Figure 20 Initial Testing

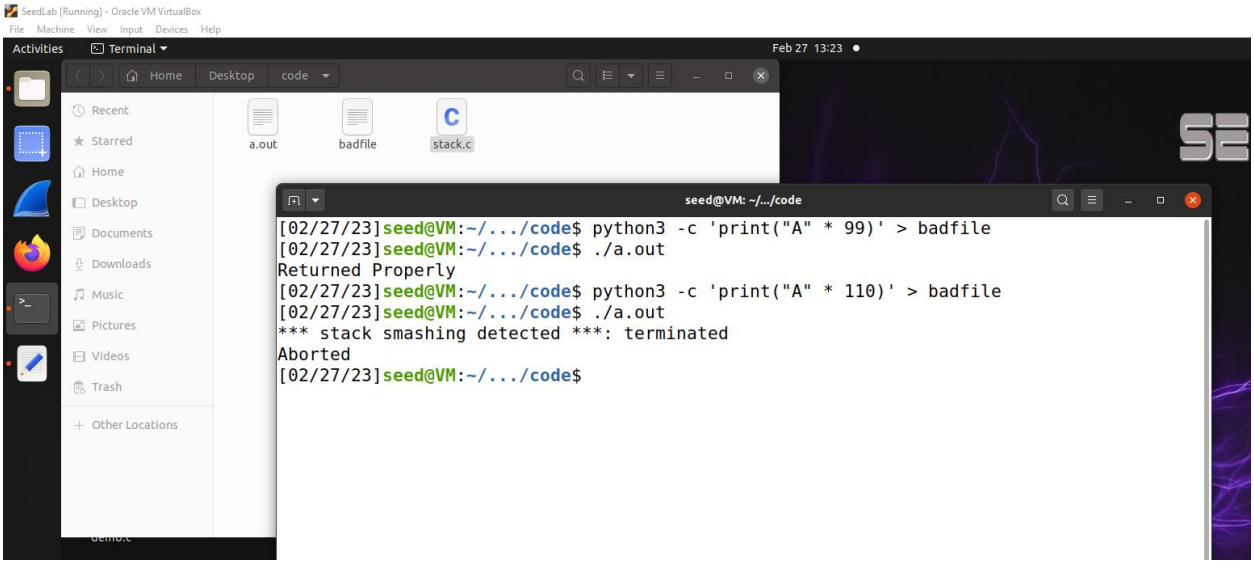

Figure 21 Facing Buffer Overflow

4.1 Compilation

To compile the above vulnerable program, we have to turn off the StackGuard and the non-executable stack protections using the -fno-stack-protector and "-z execstack" options.

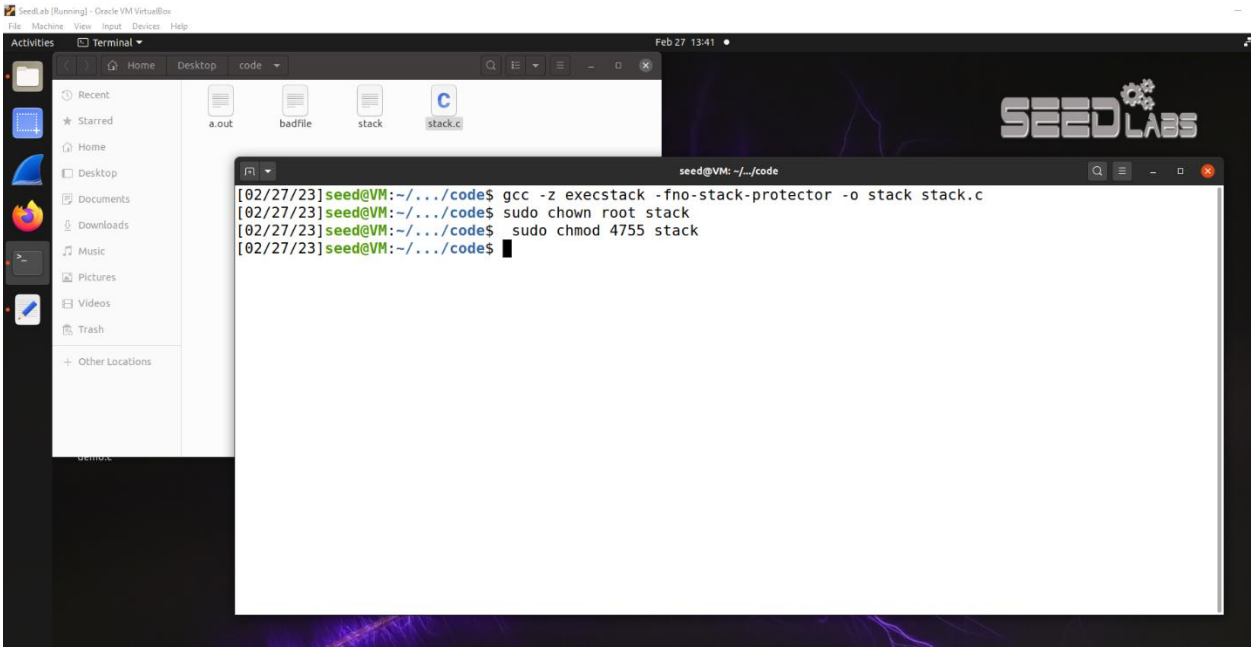

Figure 22 Compilation

After the compilation, we need to make the program a root-owned Set-UID program. We can achieve this by first change the ownership of the program to root (Line), and then change the permission to 4755 to enable the Set-UID bit (Line).

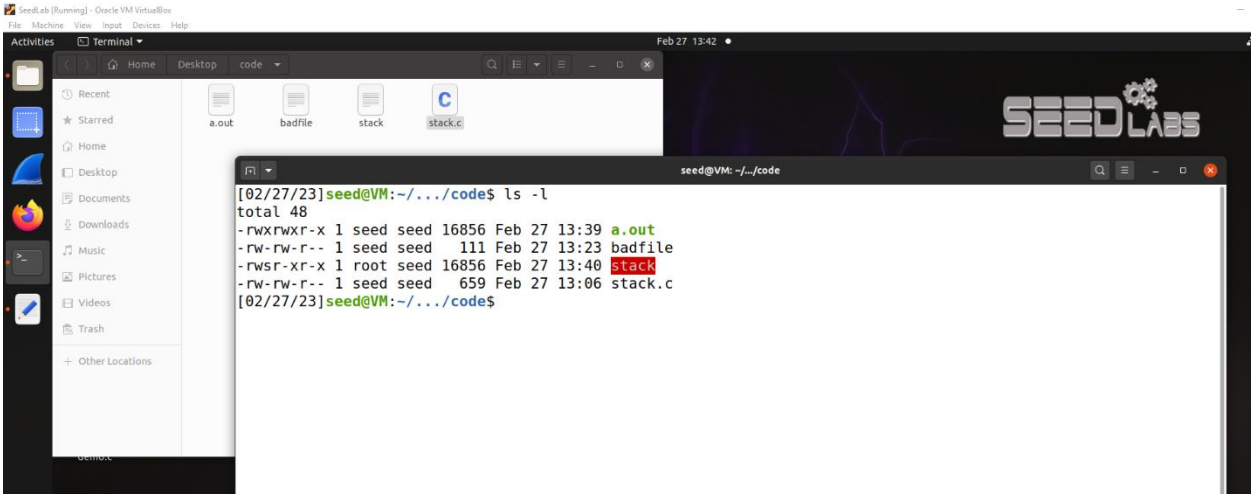

Figure 23 Permission Granted

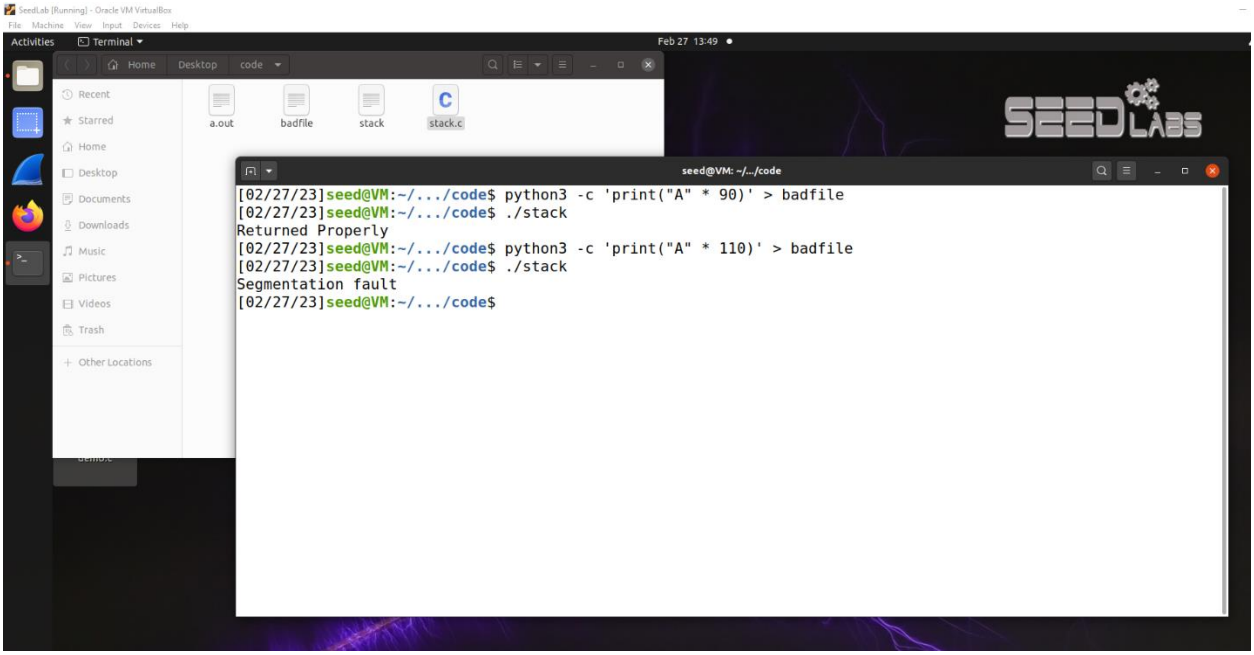

Figure 24 Showing Segmentation Fault

5. Launching Attack on 32-bit Program

To exploit the buffer-overflow vulnerability in the target program, the most important thing to know is the distance between the buffer's starting position and the place where the return-address is stored. We will use a debugging method to find it out. Since we have the source code of the target program, we can compile it with the debugging flag turned on.

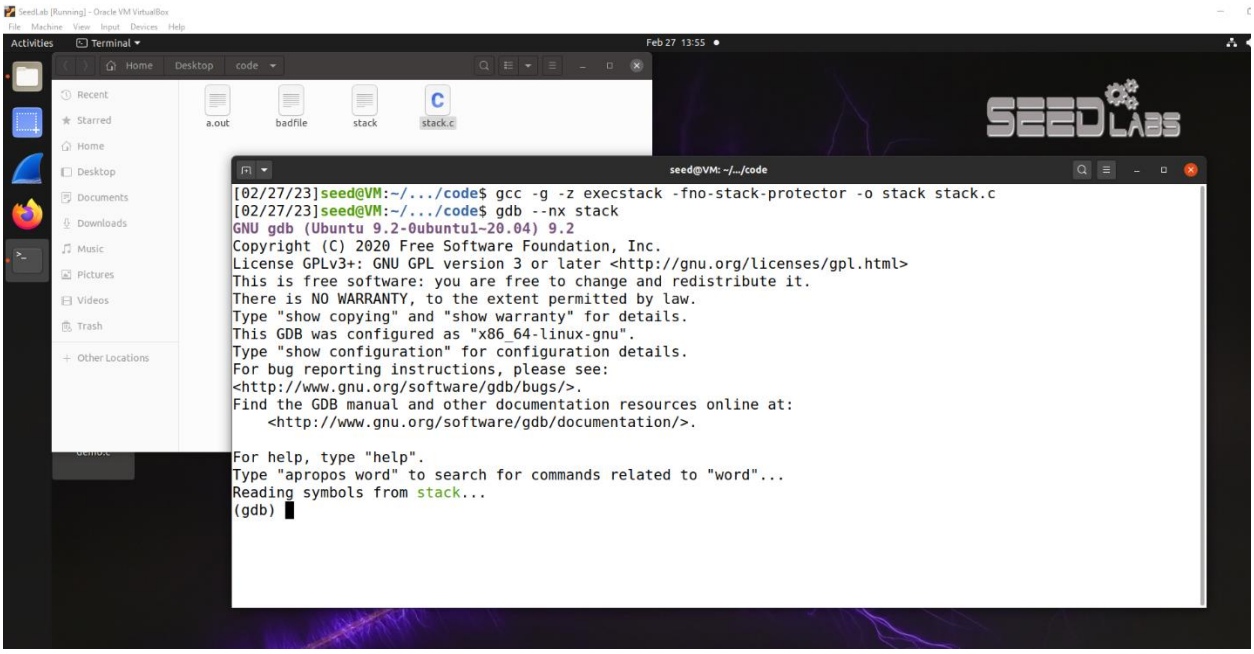

Figure 25 Debugging the code

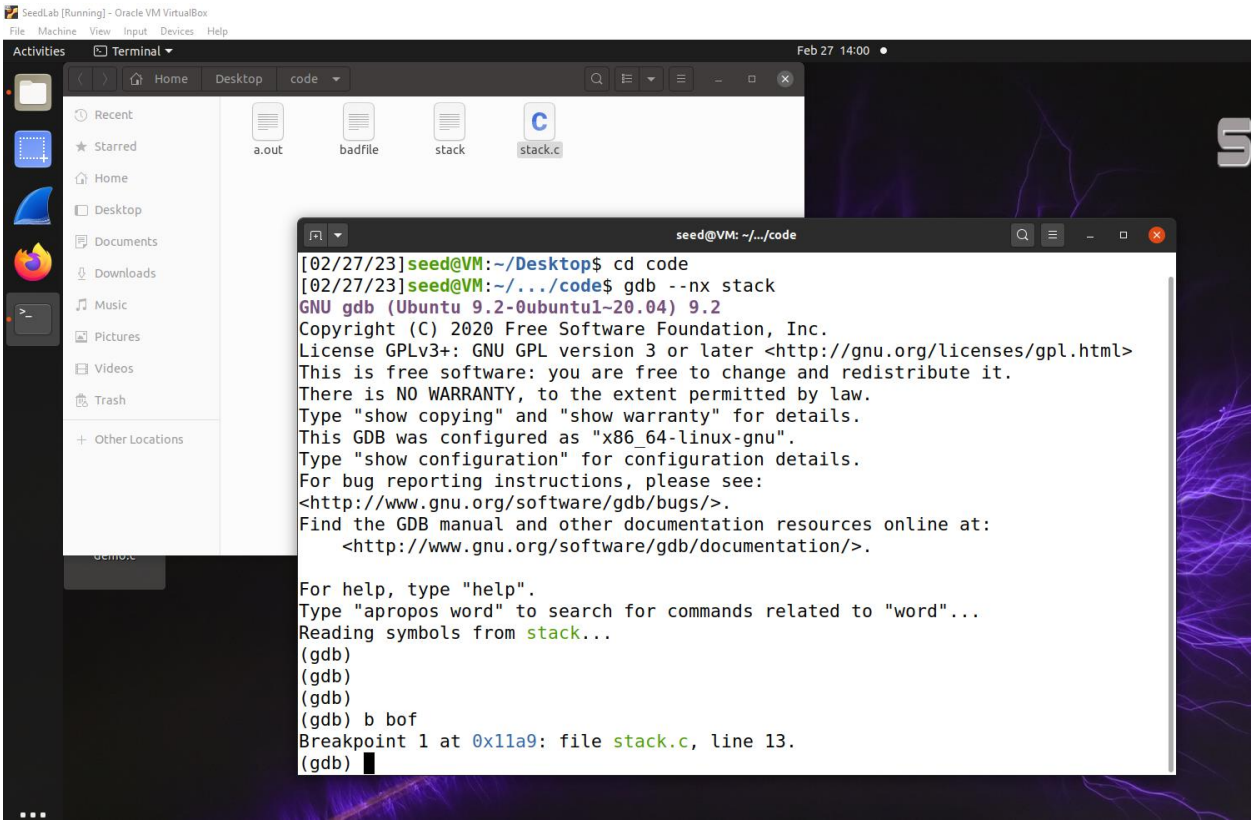

Figure 26 Setting Breakpoint

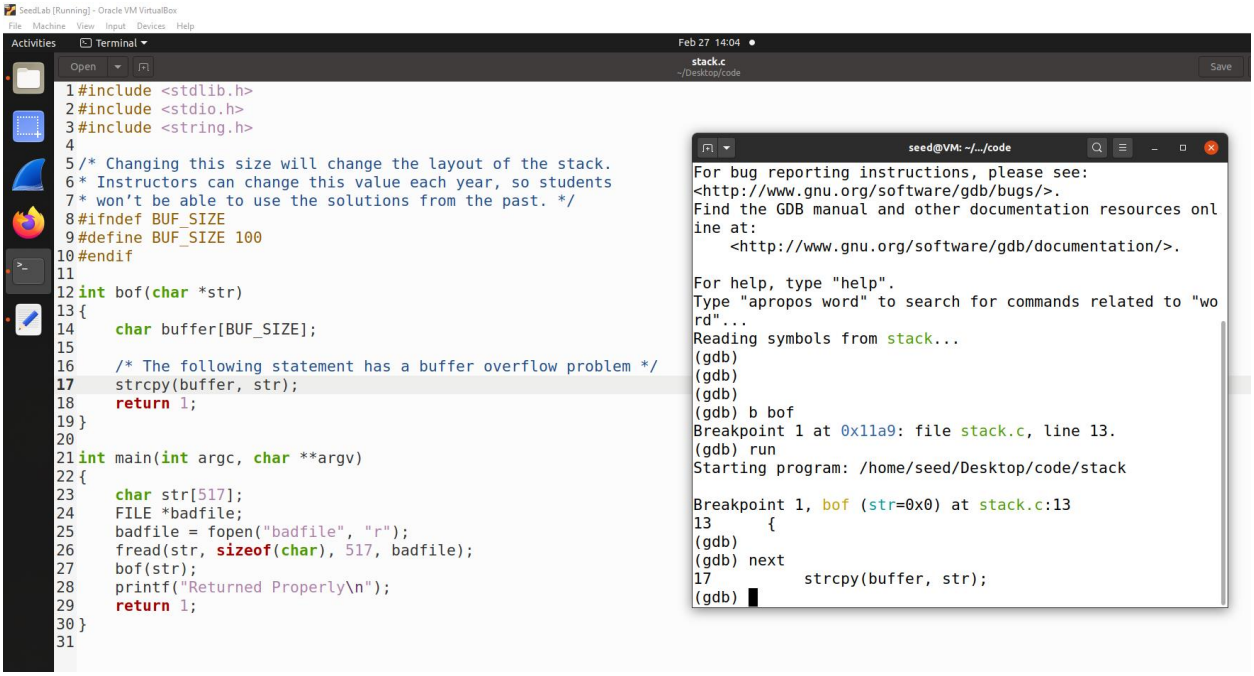

Figure 28 Matching the breakpoint with the code

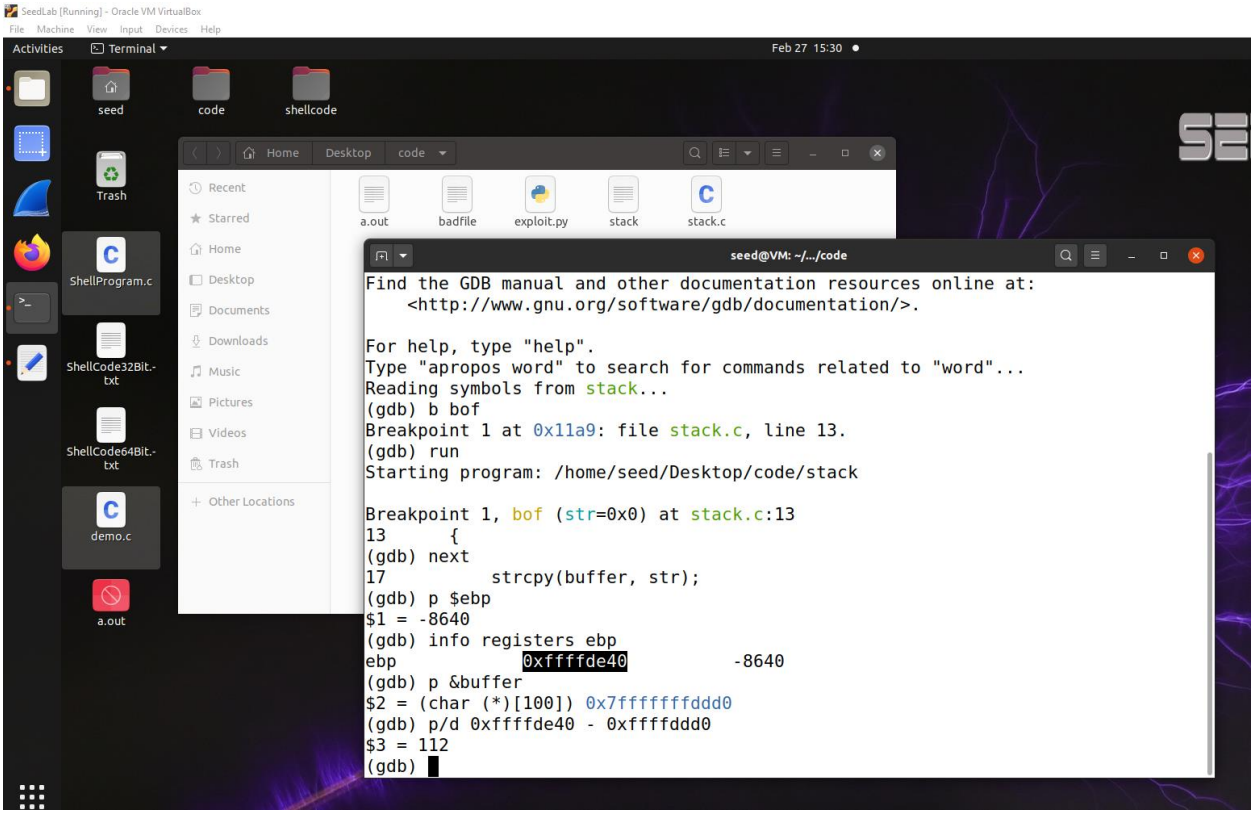

Figure 27 Finding the starting index

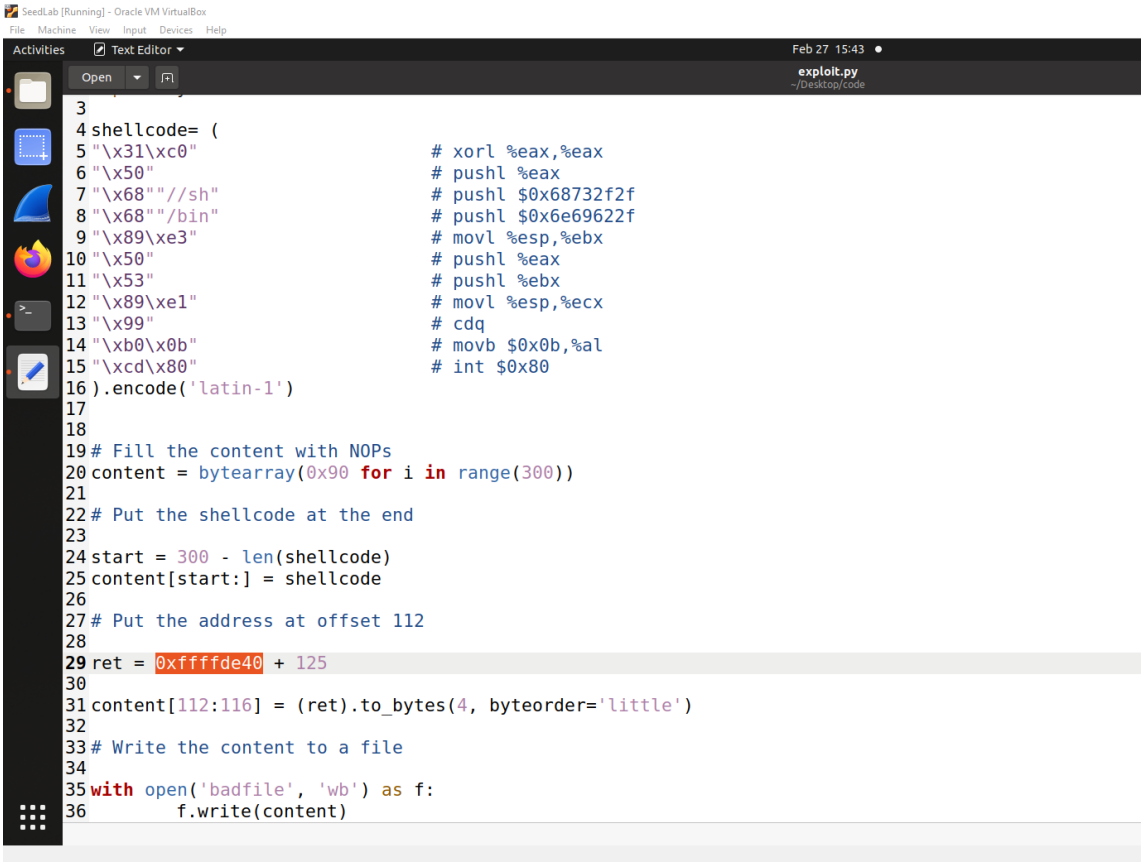

Figure 29 Python File Configuration

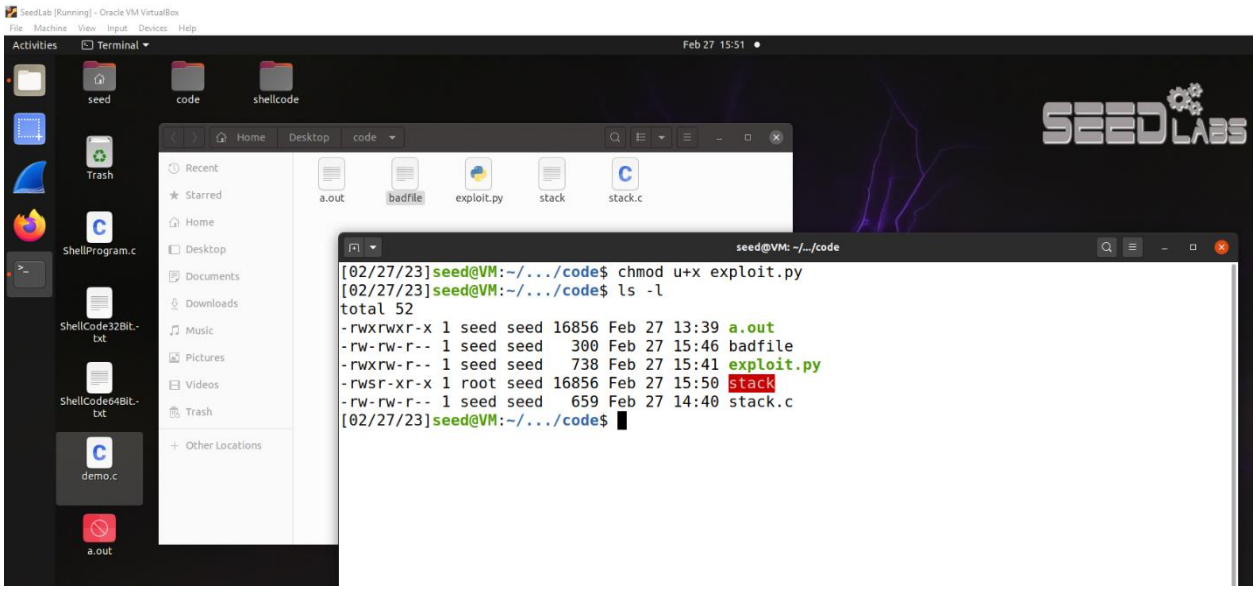

Figure 30 Taking Root Permission

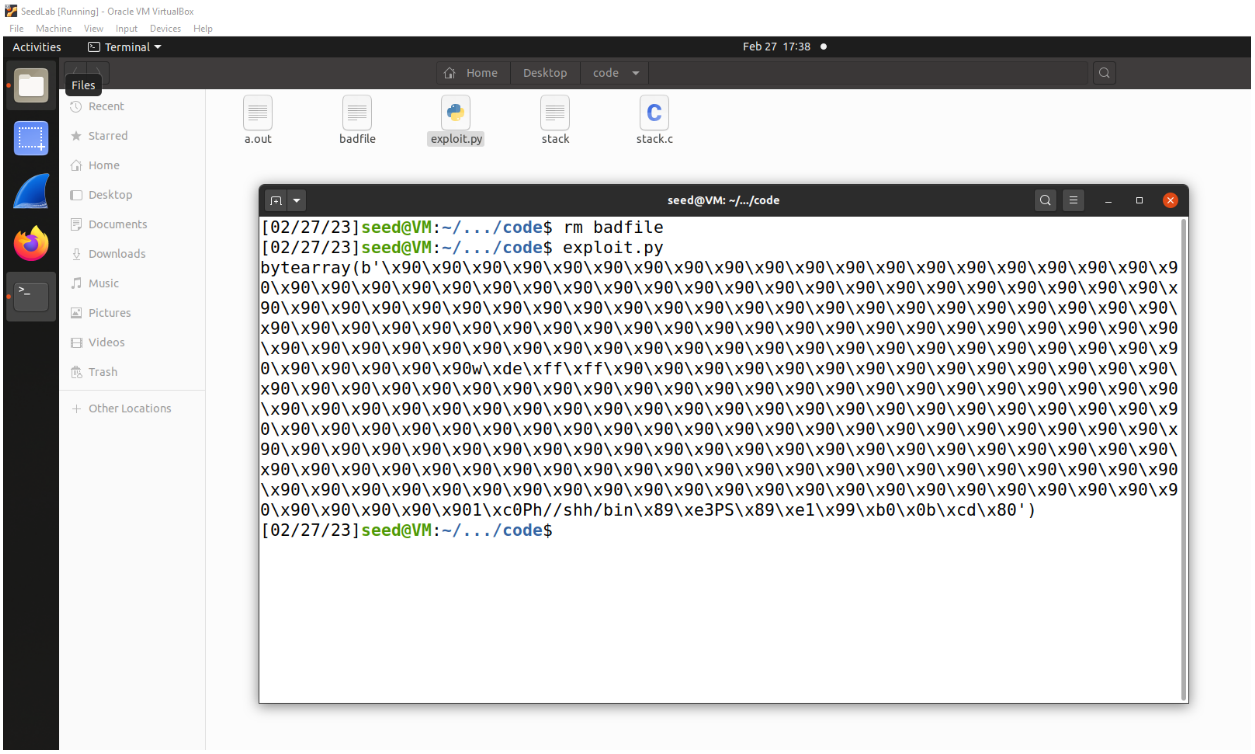

Figure 31 Final Output with malicious code

```
$./exploit.py      // create the badfile
$./stack-L1        // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

6. Defeating dash's Countermeasure

The dash shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to the real UID (which is the case in a Set-UID program). This is achieved by changing the effective UID back to the real UID, essentially, dropping the privilege. In the previous tasks, we let /bin/sh points to another shell called zsh, which does not have such a countermeasure. In this task, we will change it back, and see how we can defeat the countermeasure.

Figure 32 Change back to dash

```
; Invoke setuid(0): 32-bit
xor ebx, ebx      ; ebx = 0: setuid()'s argument
xor eax, eax
mov  al, 0xd5     ; setuid()'s system call number
int 0x80

; Invoke setuid(0): 64-bit
xor rdi, rdi      ; rdi = 0: setuid()'s argument
xor rax, rax
mov  al, 0x69     ; setuid()'s system call number
syscall
```

Experiment. Compile `call_shellcode.c` into root-owned binary (by typing "`make setuid`"). Run the shellcode `a32.out` and `a64.out` with or without the `setuid(0)` system call. Please describe and explain your observations.

Figure 33 Instruction to defeat dash

7. Defeating Address Randomization

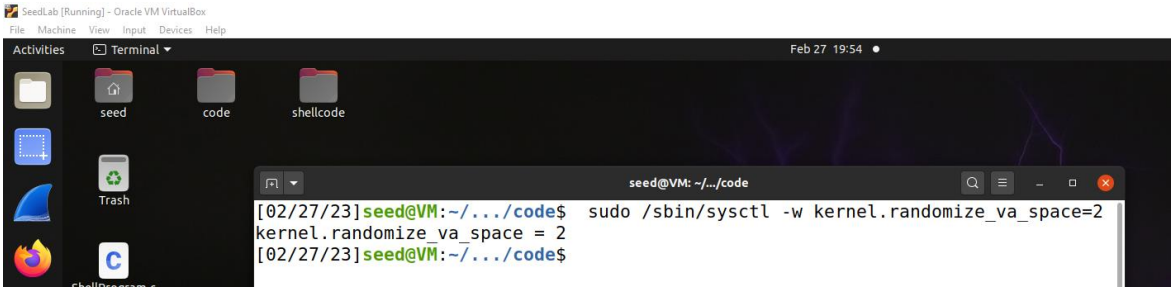

Figure 34 Address Randomization

We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the badfile can eventually be correct. We will only try this on stack-L1, which is a 32-bit program. We can use the following shell script to run the vulnerable program in an infinite loop. If our attack succeeds, the script will stop; otherwise, it will keep running.

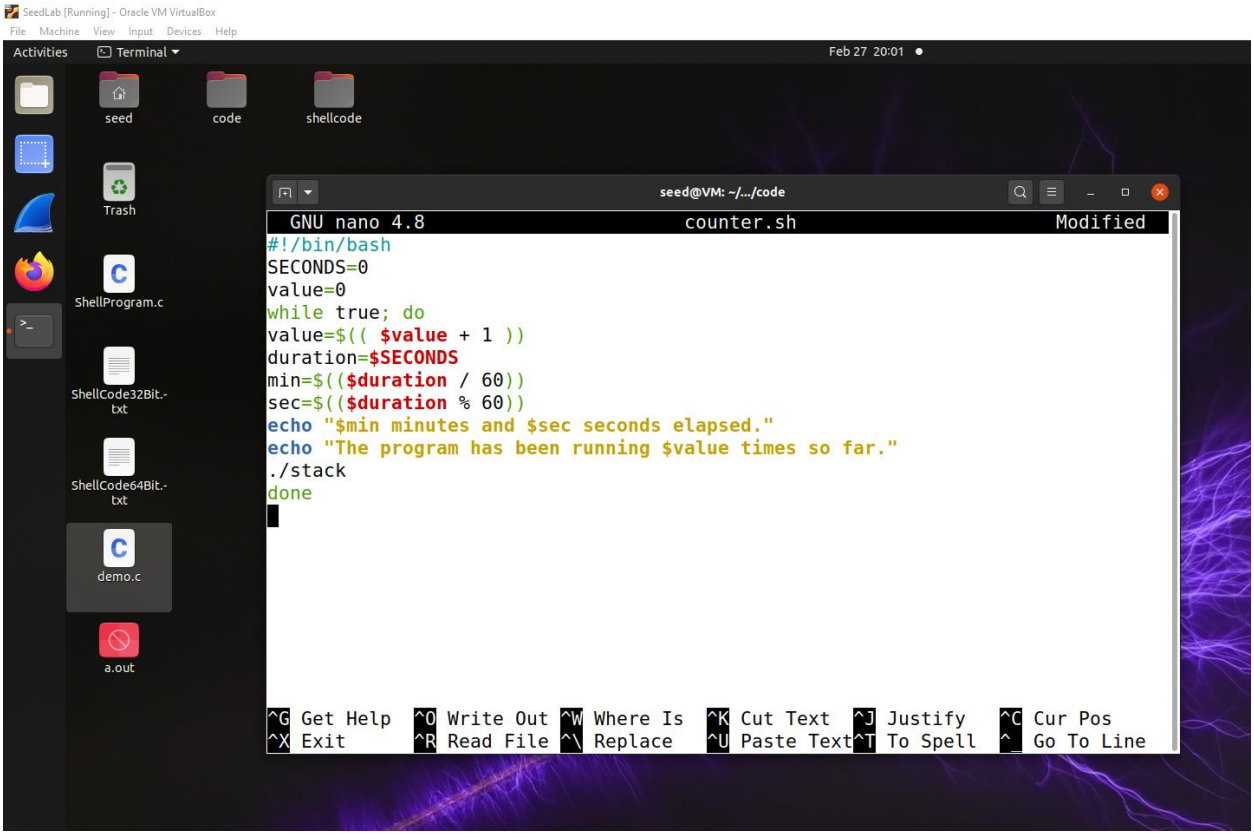

Figure 35 Shell program with infinite loop

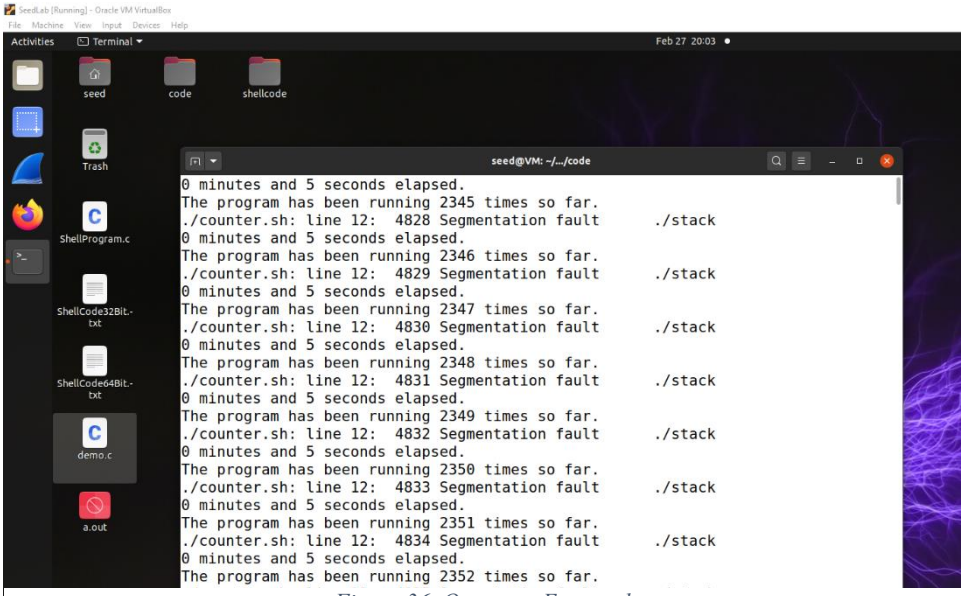

Figure 36 Output as Expected

8. Turn on the StackGuard Protection

Many compiler, such as gcc, implements a security mechanism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. In our previous tasks, we disabled the StackGuard protection mechanism when compiling the programs. In this task, we will turn it on and see what will happen.

9. Turn on the Non-executable Stack Protection

Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the gcc, which by default makes stack non-executable.

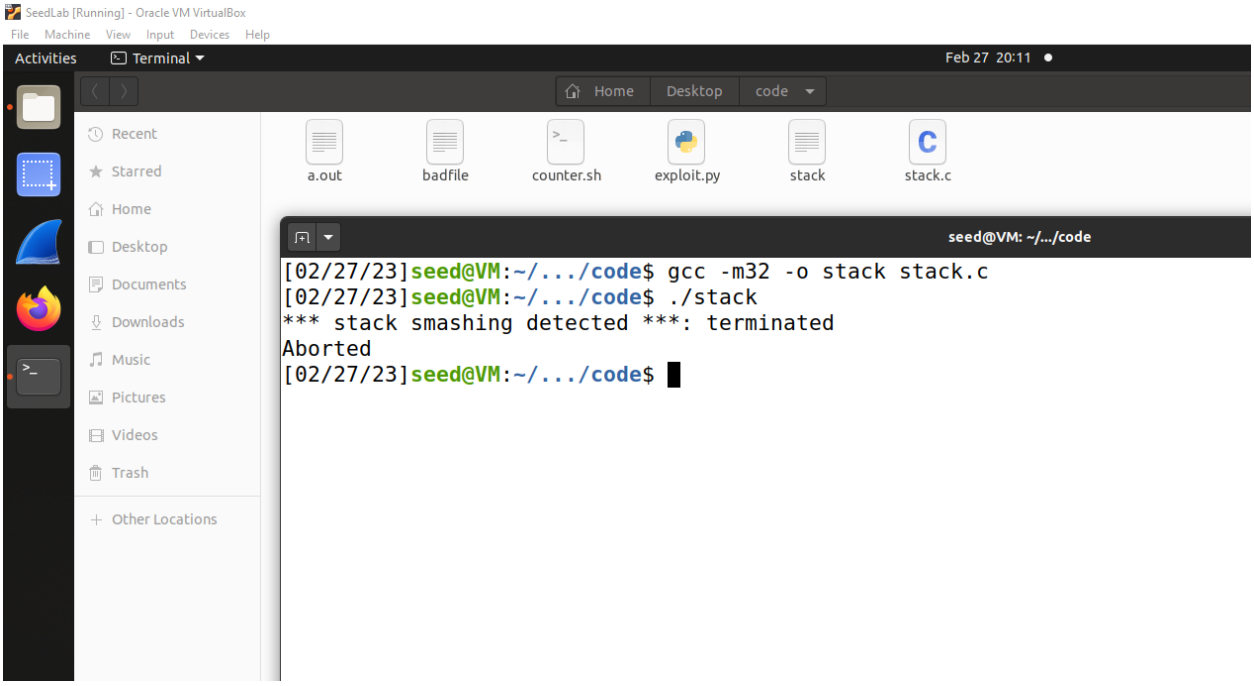

Figure 37 After Turning on Stack Guard and non-executable stack protection