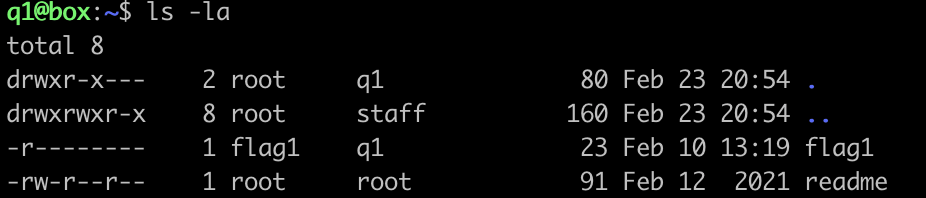
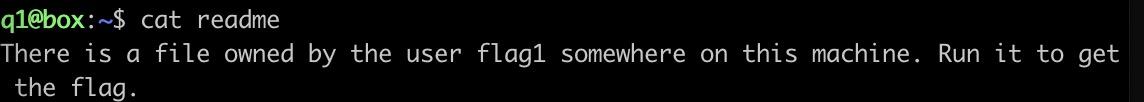
Q1:

The flag is flag{gDR5TVkUyHBGtcwE}

At first I logged into the VM via ssh. Then I use ls -la to check if there are some hint file:



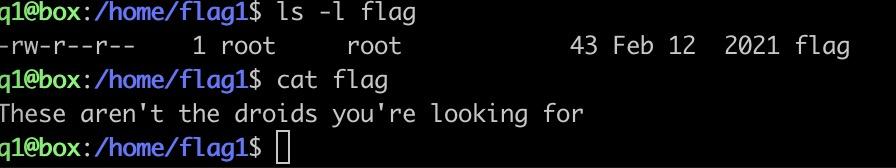
It seems that there are only two files in this directory, which are ‘flag1’ and ‘readme’ and we don’t have permission to execute and write for ‘flag1’. Then I try to open the ‘readme’ first.



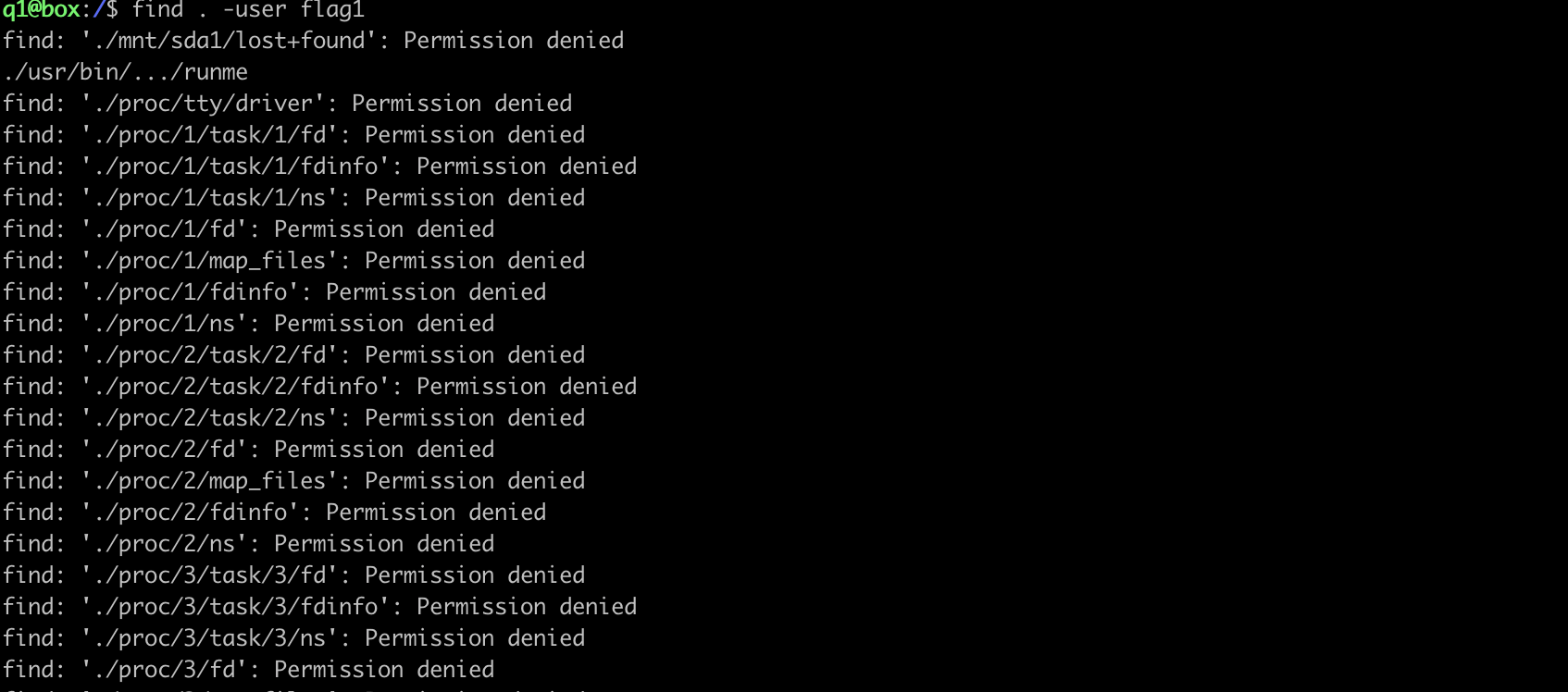
It shows me that in the directory, there is a file we can run to get the flag. However, I already checked all files in this directory. There are no hidden files or folders here. So I thought maybe in the upper directory, I use the command ‘cd ..’ back to the upper folder. I found a folder called flag1 and there’s a file called flag, at that time I thought that should be the target file.

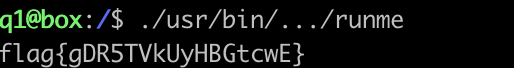


When I open it by ‘cat flag’, it show me:



Well,my thinking is right, so I want to go directly to the root directory and see how many files user: flag1 has. I use the command ‘cd //’ to go back to the root directory. Find is a command line for traversing the entire structure of a file. It looks for specific files based on our requirements. In q1, we need to find the files belonging to the user ‘flag1’.

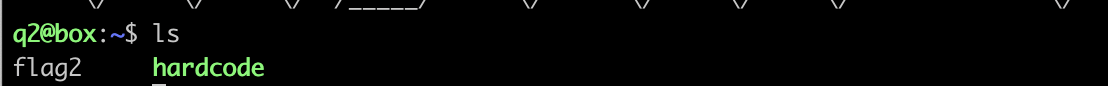


As we can see, there’s a file called runme that can be executed. Then I run it to get the flag:

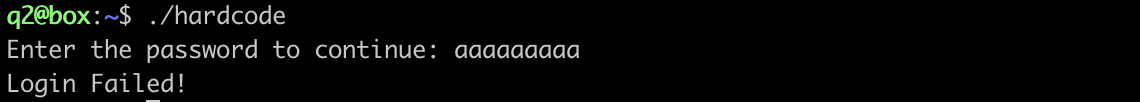
I didn't know how to solve this problem at first. I have tried many ways to run flag1 directly, such as downloading it locally and using chmod to change permissions. However, these methods all failed, and I thought at the time that maybe I was in the wrong place. So I reoriented, as mentioned in the readme, as user1, trying to find a file that works.

Q2:

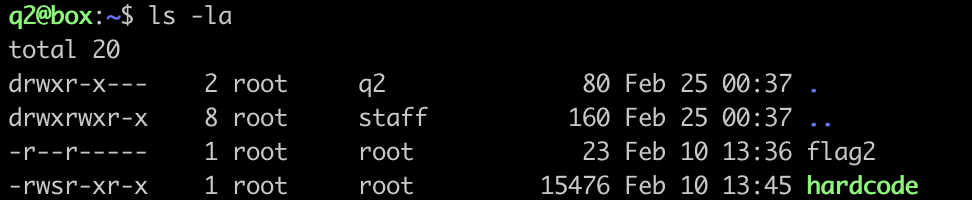
The flag is flag{H8egp5TUDrDZMrAm}



As the instruction said, we need to run the ‘hardcode’ then enter the correct password.



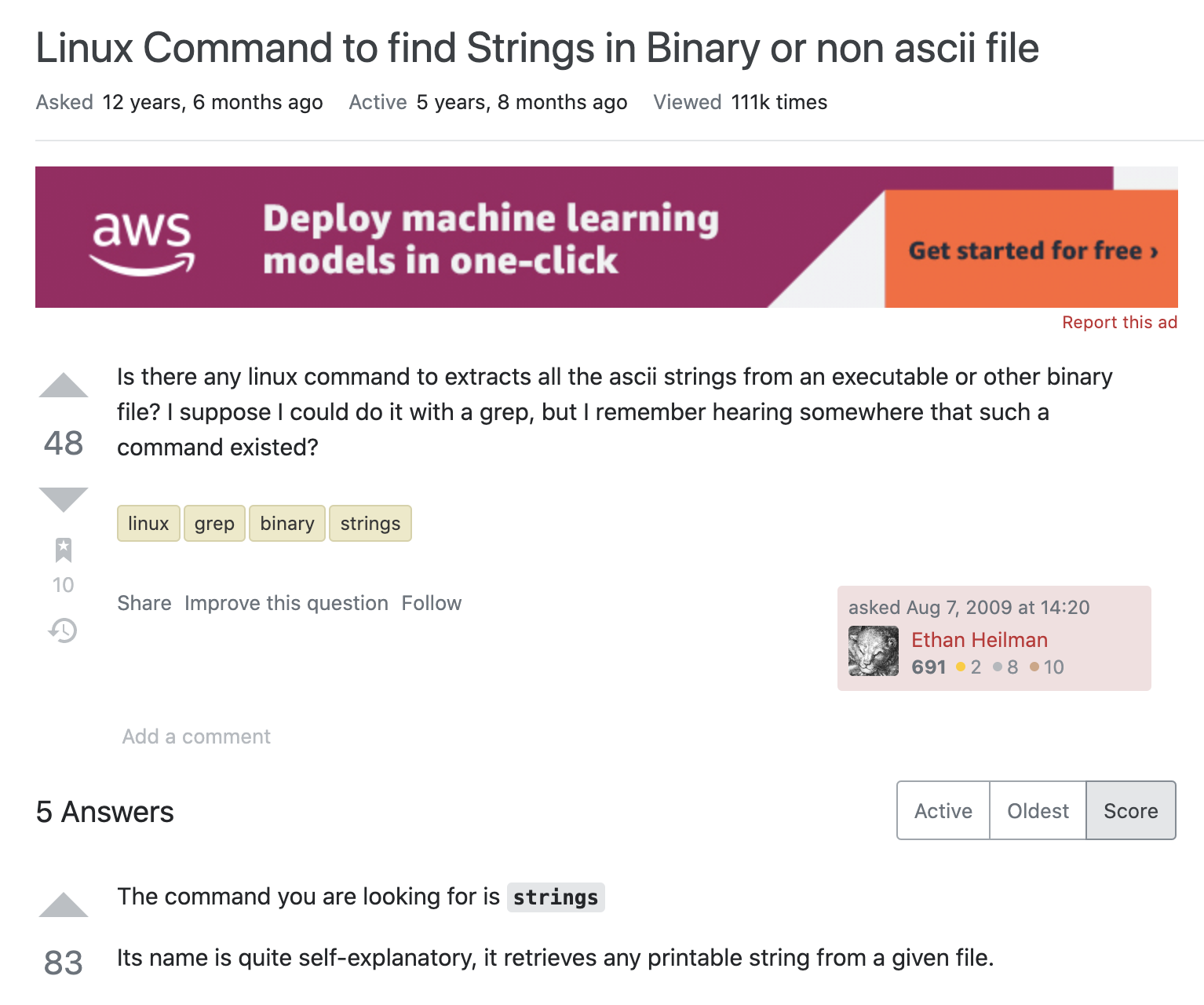
I tried the simple password and it showed the login failed. I want to check the file permission so that I know what we can do for the file.



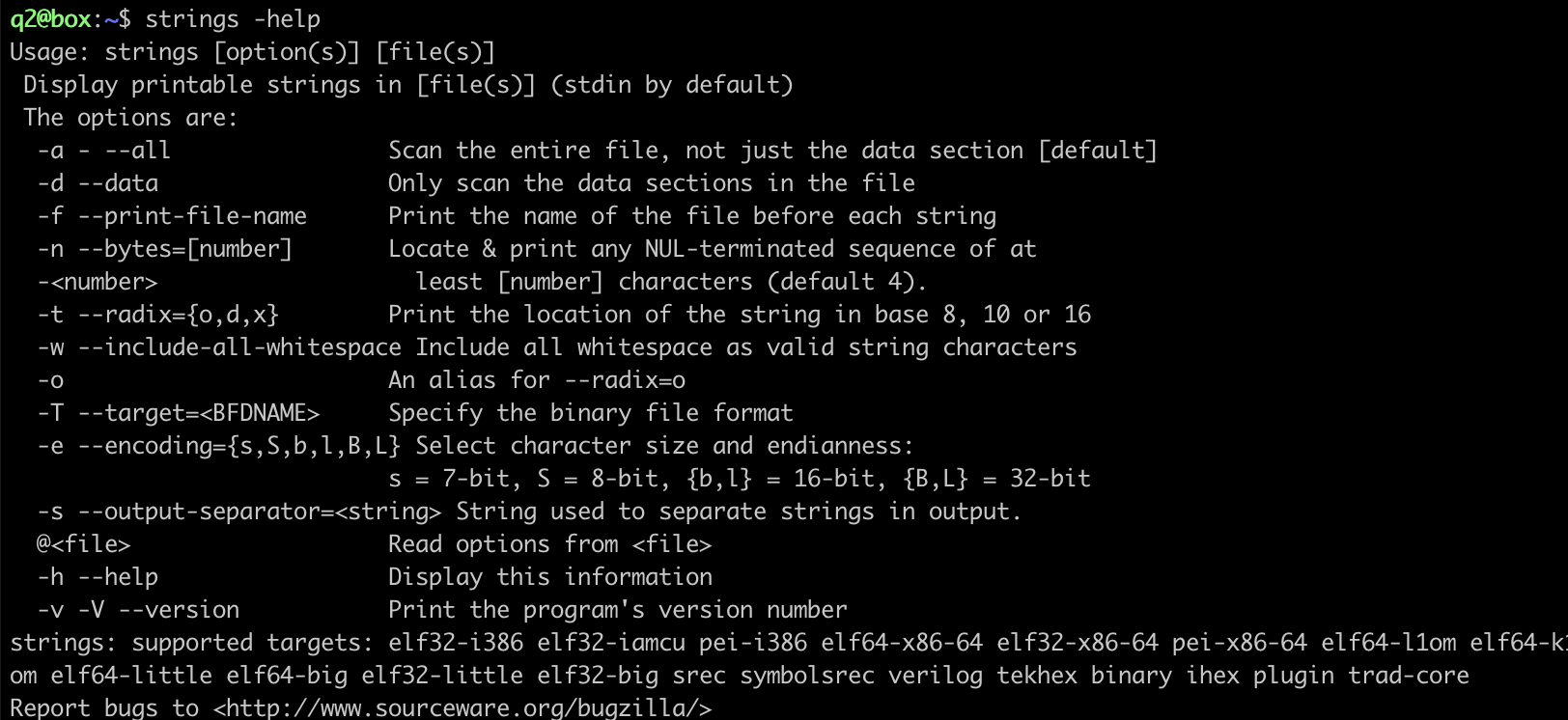
It seems that we can read and execute the hardcode, so I open the hardcode:



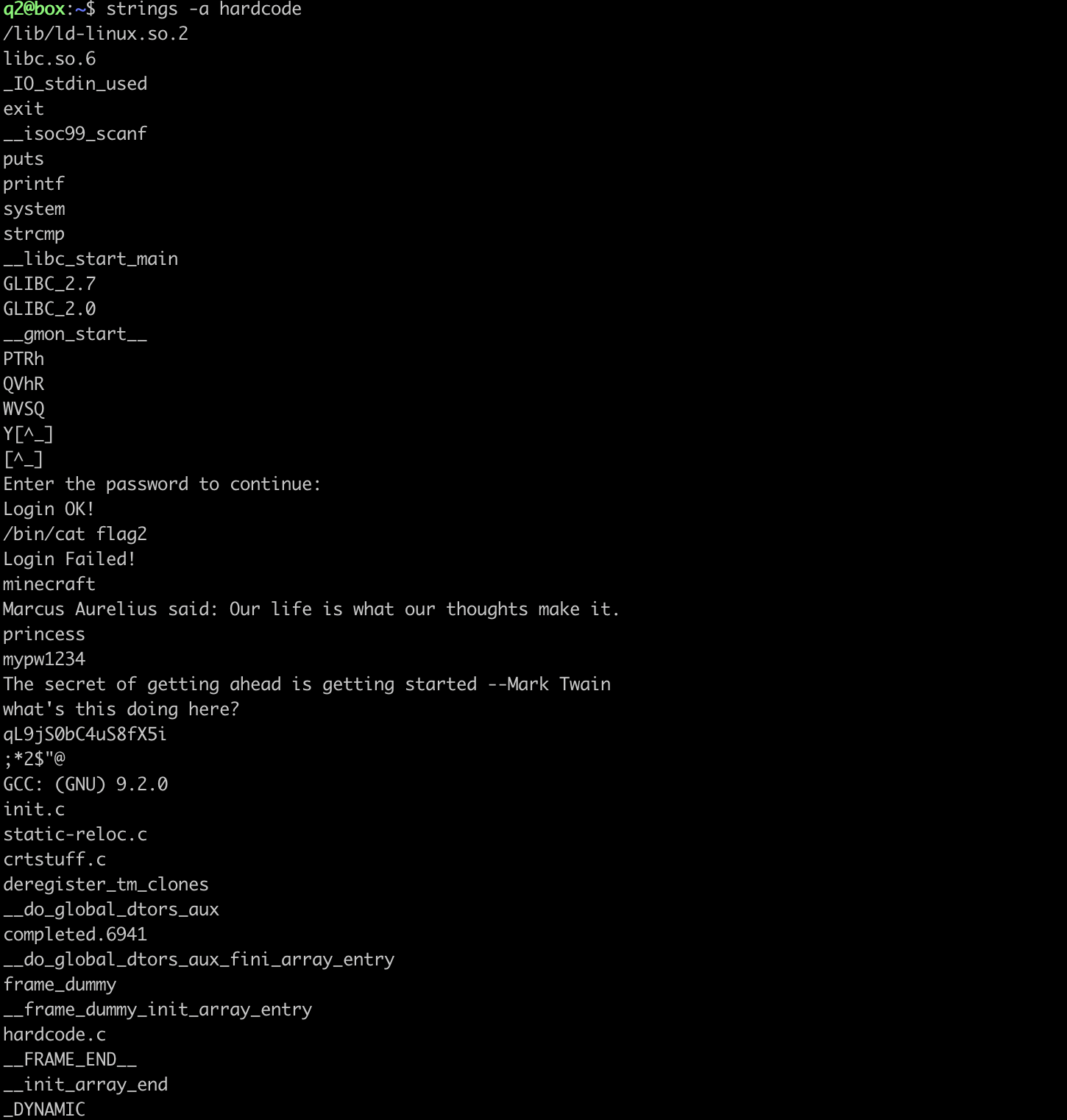
This looks like a binary file that humans can't read. I guess the challenge is mainly to extract text from binary files. So I went to search how to convert binary to text in Linux.



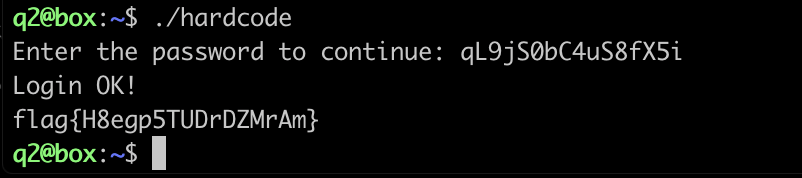
The strings command is part of the GNU Binutils set of binary tools for printing printable strings in files.



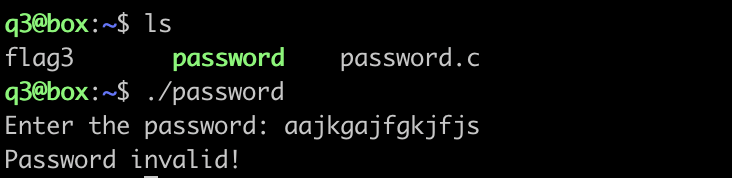
I need to use -a to scan the whole file, so the command is ‘strings -a hardcode’



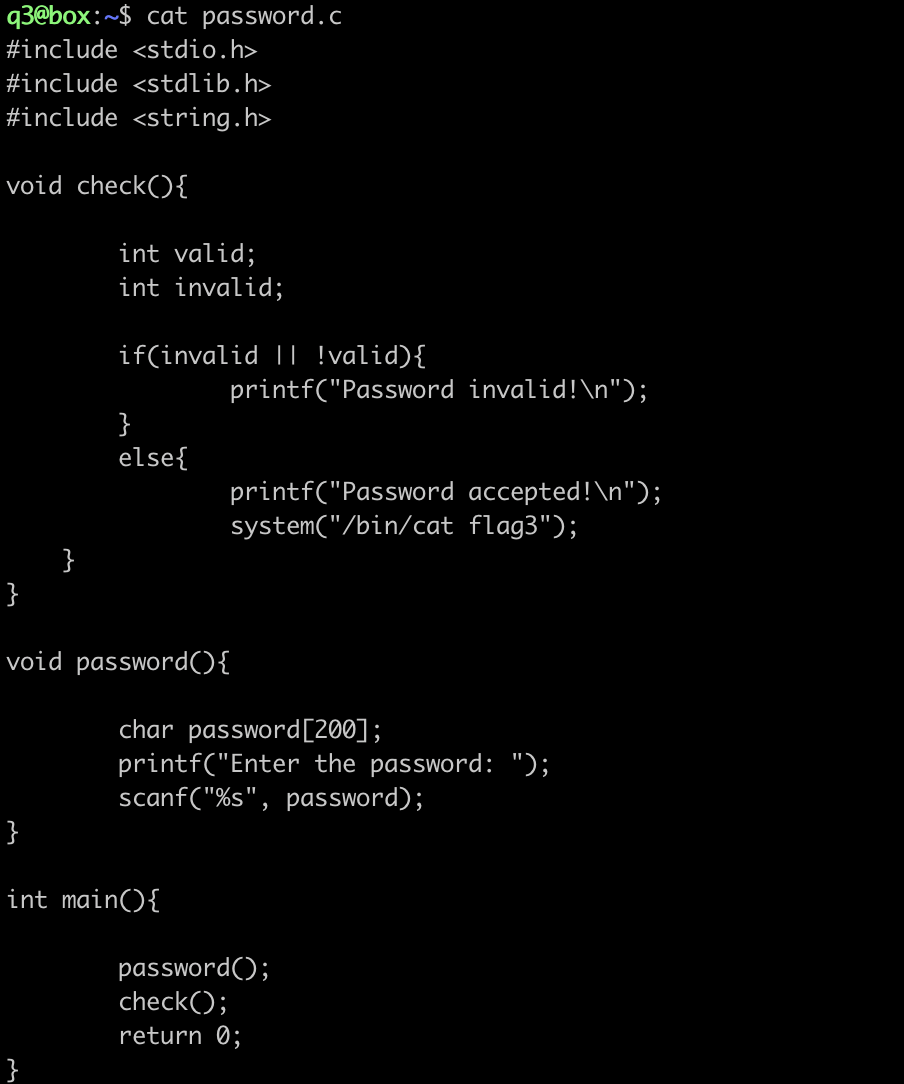
The password is hidden there, and I have tried many possibilities. Eventually I found out that the password was qL9jS0bC4uS8fX5i.



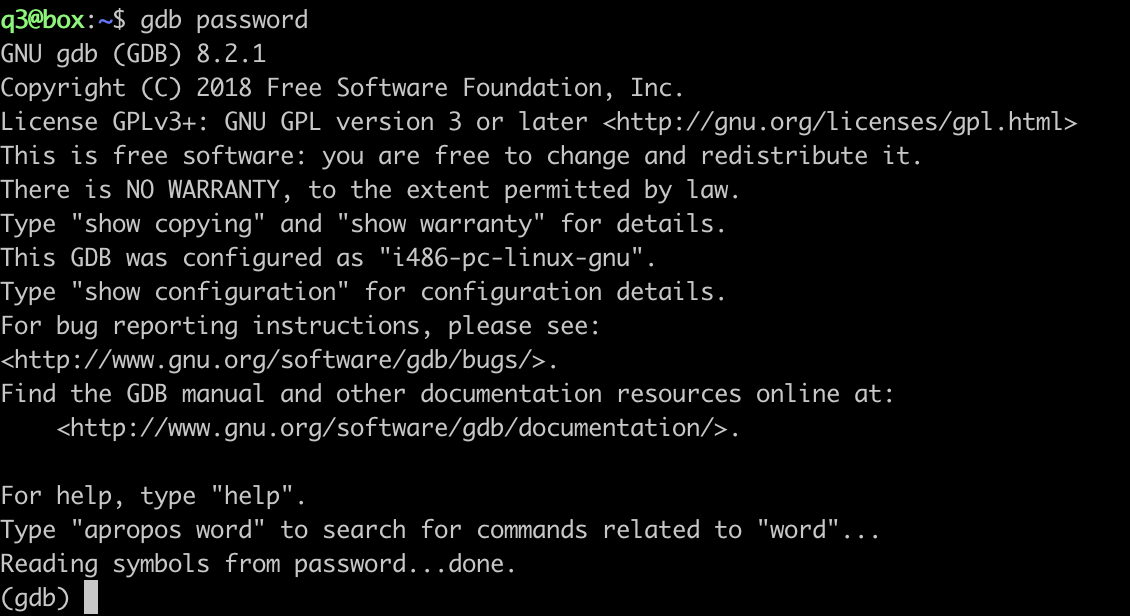
Q3:



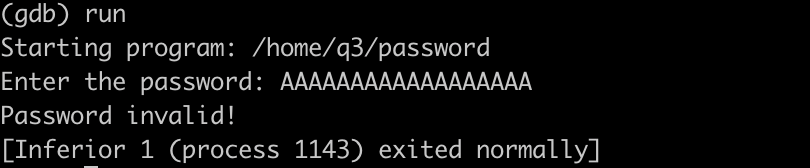
At first I list all files in the directory, there are three files and it seems like the file called password is the operating file for the password.c. Then I try to execute it by the command “./password” It shows me that I need to enter the password, then I try some random password to see what feedback it will give me. Well, let me try to open the source file to see what happened:



As the hint said, this question is similar to the bof in assignment2, I think this question will be about the overflow attack. First thing I need to do is to use gdb.



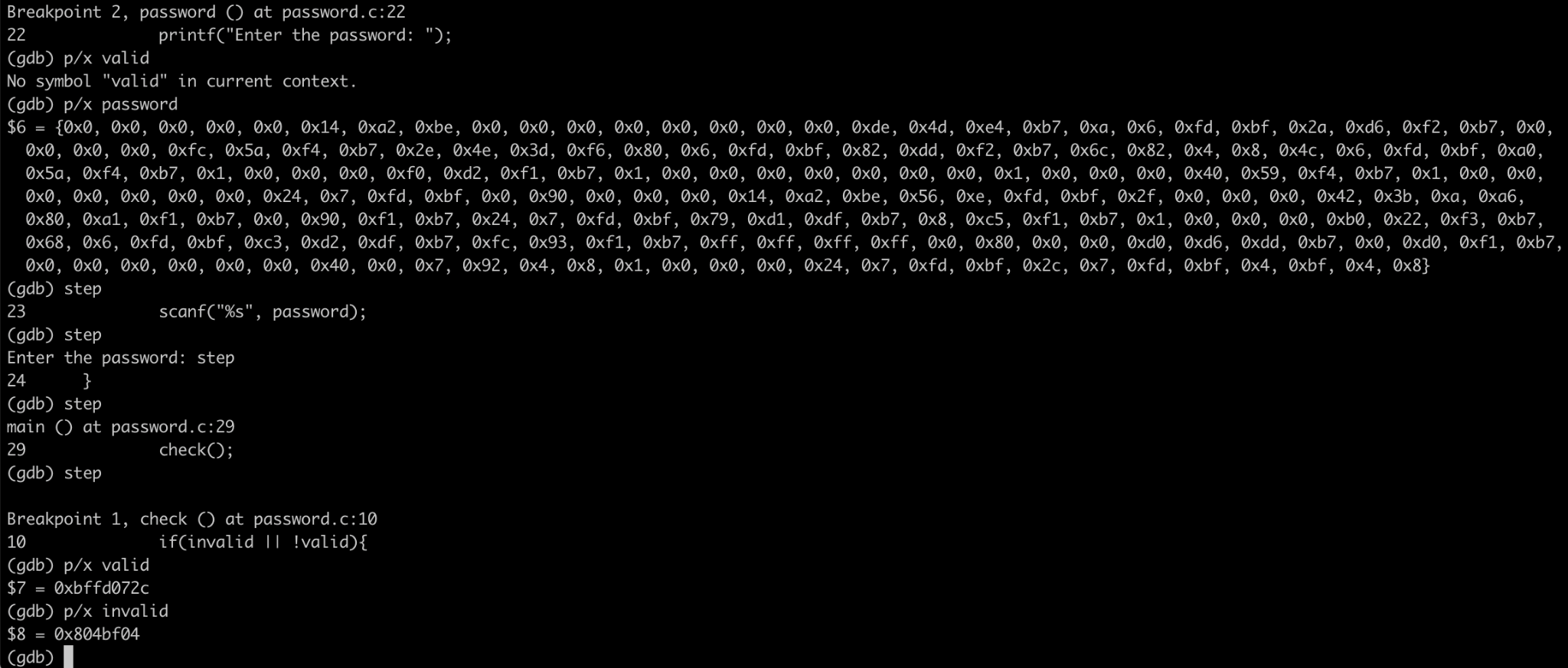
Then I run the program in gdb:



I set a breakpoint at the if statement so that I can see the content about the valid and invalid in hexadecimal. Then I want to see the password content at the same time. However, it only shows me 0x55; that’s weird, because I input AAAAAAAAAAAA. It should not be 0x55. I try to research what’s going on. I found the password already saved in memory so I can’t read it.



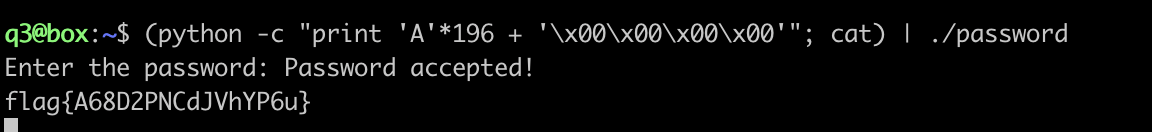
The solution is simple, just set a breakpoint at the password() function.



Just like ‘overflowme’ in bof, although it has 200 digits, the eighth to last five digits are the same as Valid's Little Endian; the fourth to last digits are the same as Invalid's little-endian format. So it seems that passwords can control valid and invalid.

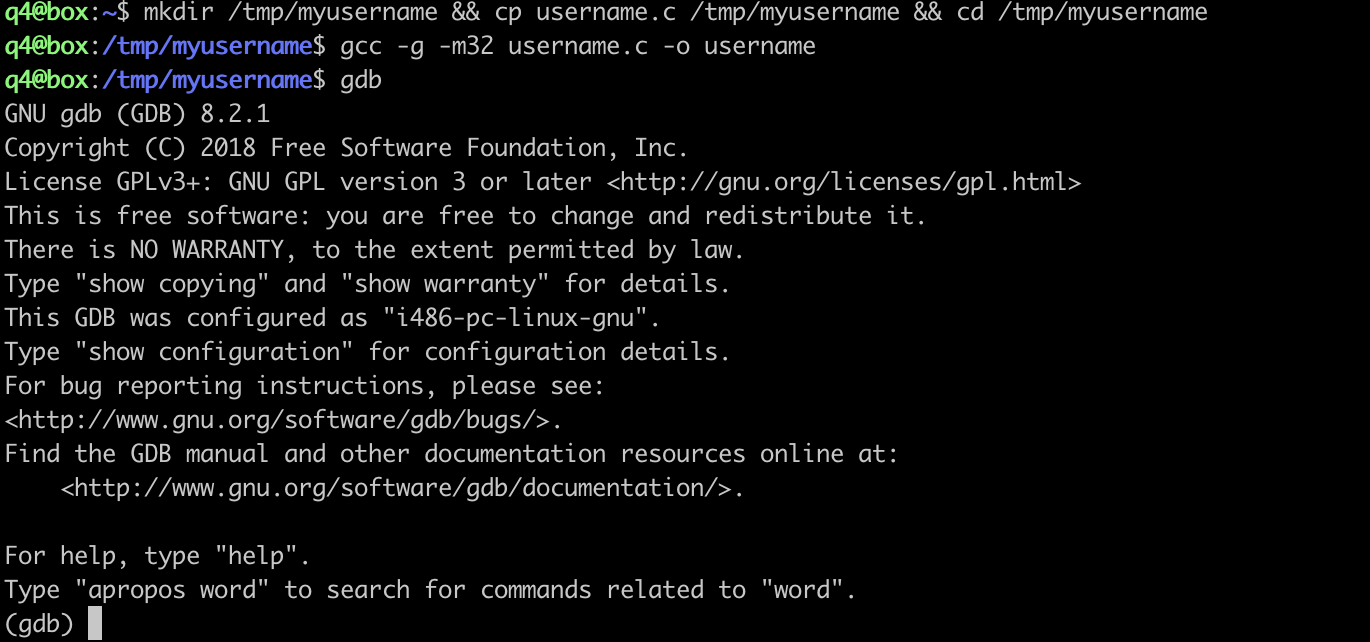
Next, I went back and looked at the if statement in check(). In order to jump into the else, we must set the if statement to 0. Inside the 'if' is an OR operator, we need to make 'invalid' or '!valid' becomes 0. I chose to make invalid 0x00000000.

We will enter 'A' to the 196th digit. Then we will input the same bytes as invalid. I'm writing a simple little Python program as an input method that generates this mix of ASCII characters and raw hex bytes:



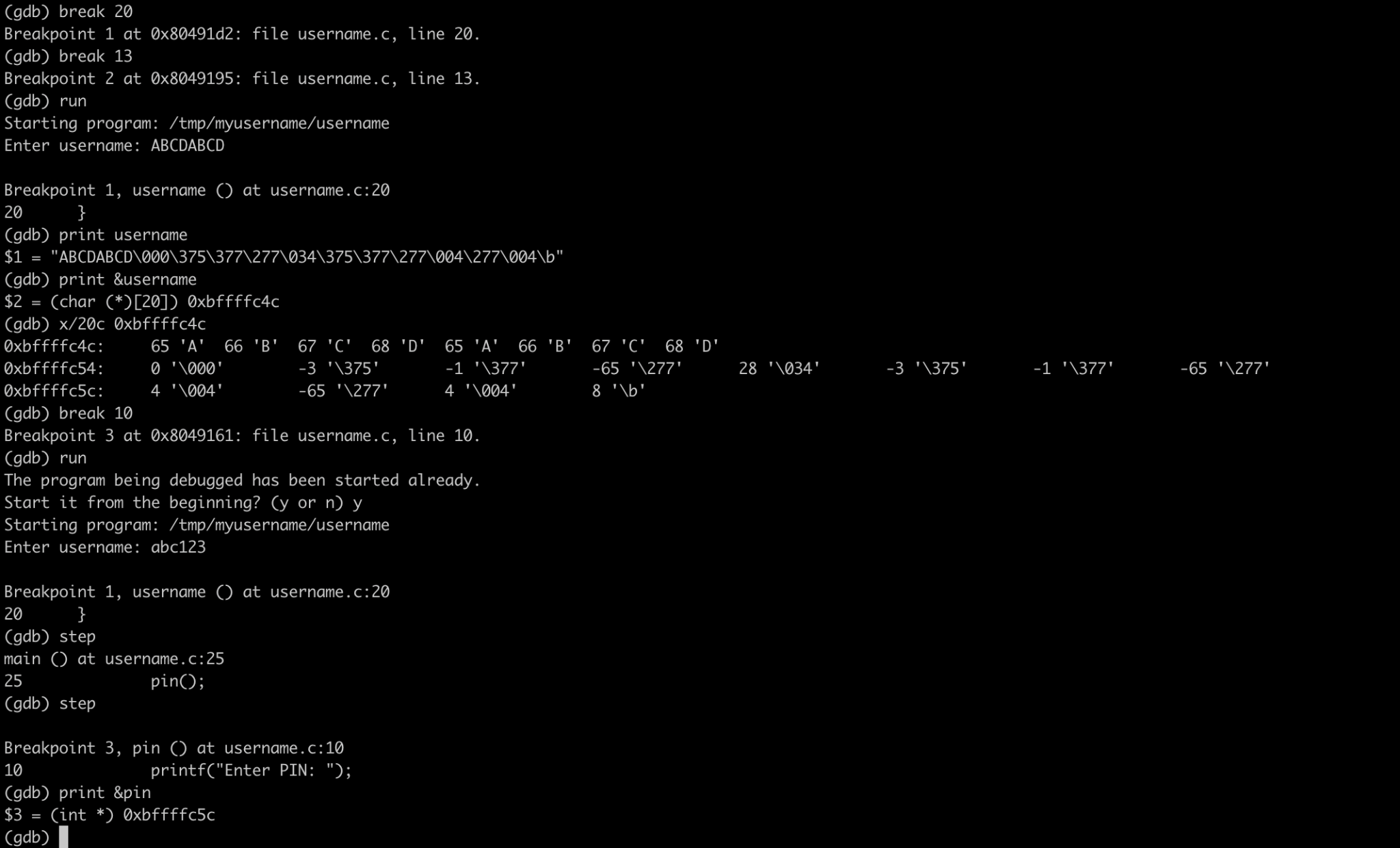
Q4:

As the hint said, the question is similar to the practice question ‘password’. At first I create a temp directory, compile it and use the gdb:



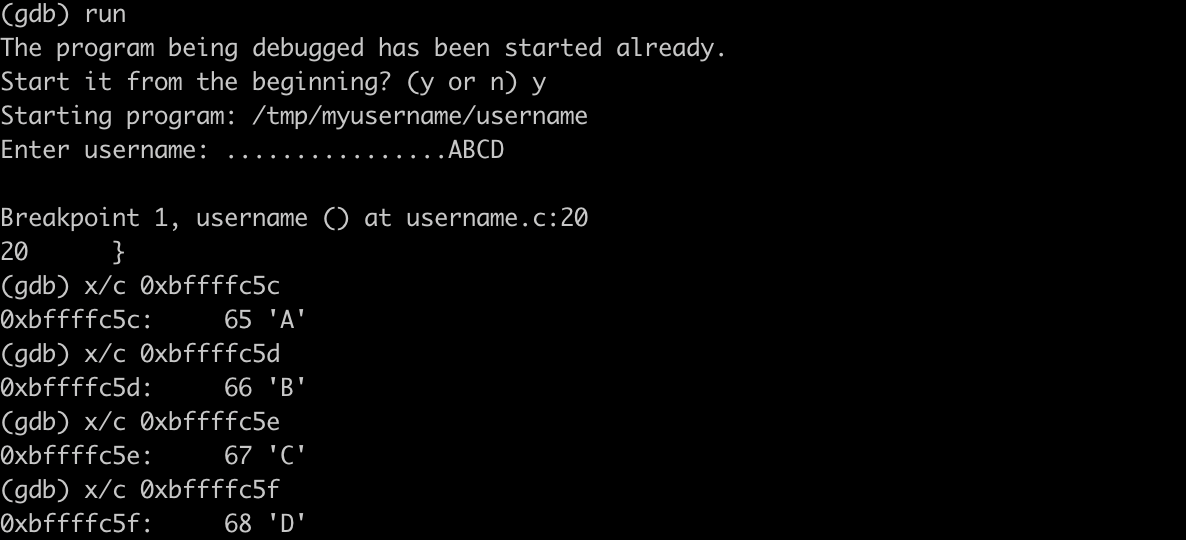
I set up a breakpoint to see the content of the username after it has been written. To find out where in memory the username buffer is stored, I find out by printing the contents of the pointer to username. It starts from 0xbffffc4c to 0xbffffc5c. Then I set another breakpoint in the function pin(). Then I print the content of the pointer to the pin which is 0xbffffc5c.

At this point, I think the problem is similar to the question ‘password’. Although the username and pin are two functions, local variable will be allocated when the function is called.

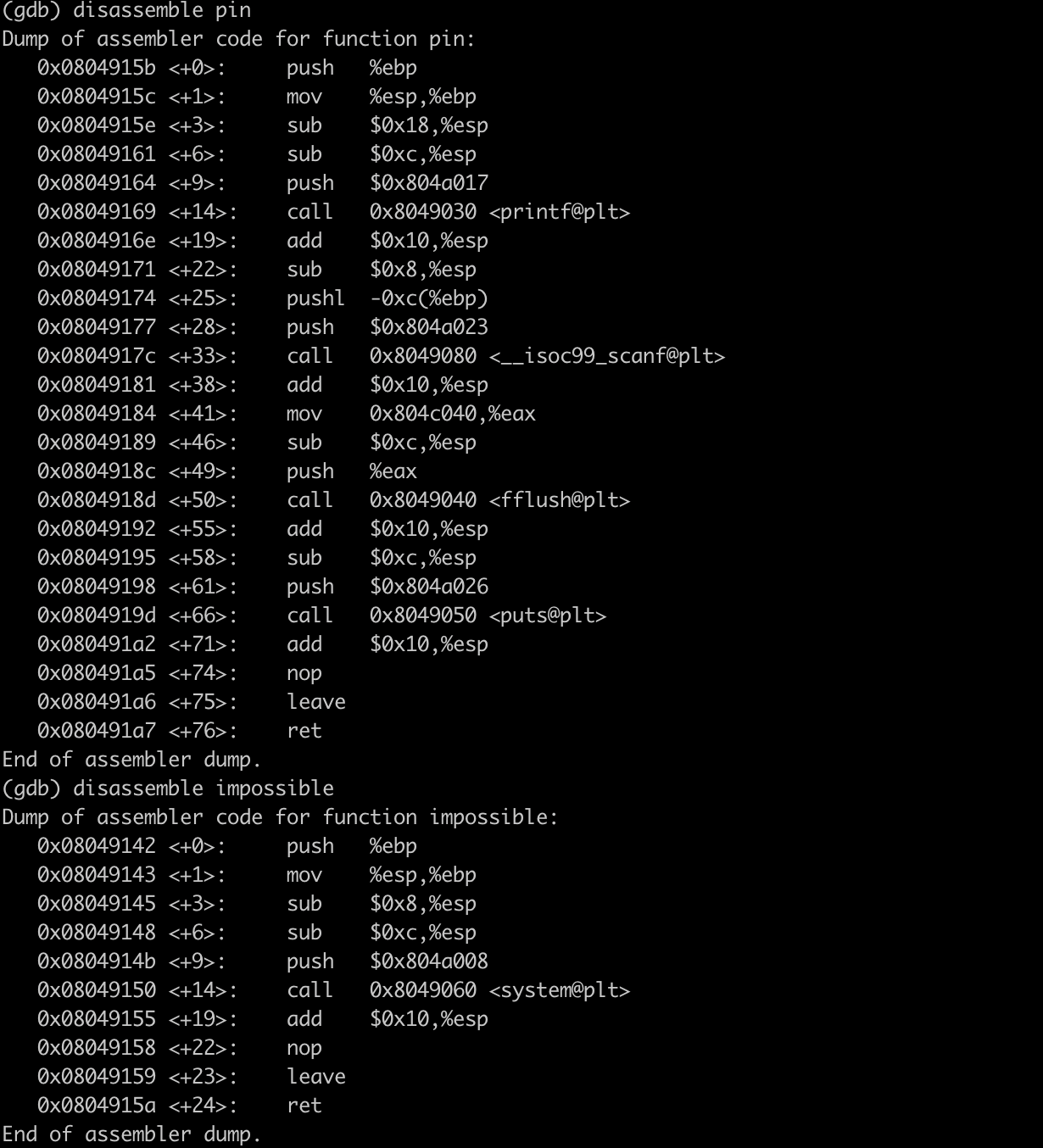


Next, I will check that when the address is set in username(), when we call pin(), the value in the address is still there. Let me enter a name consisting of 16 "." followed by "ABCD." Then we will use x/c to check the contents of the pin.

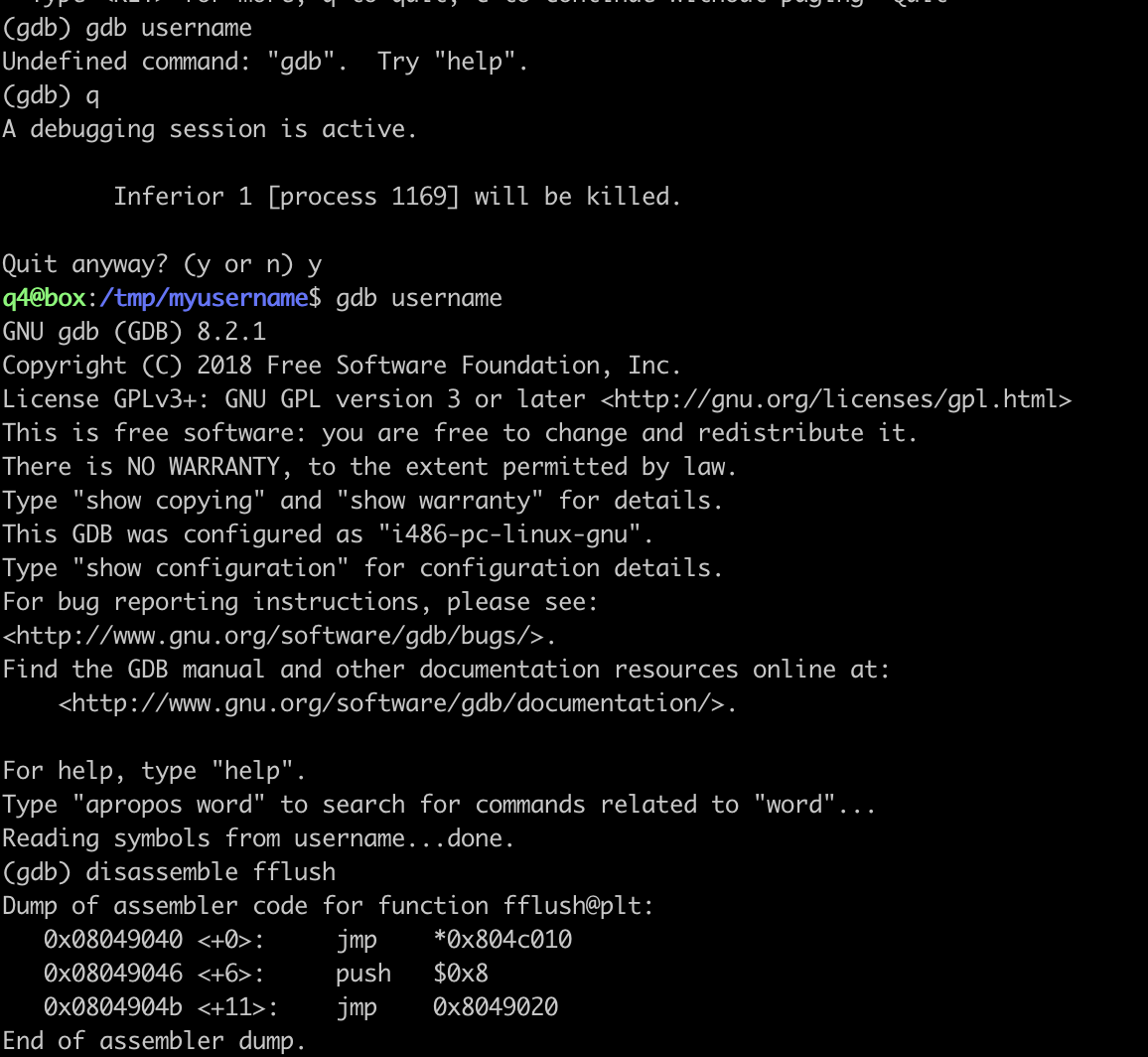
I re-run gdb and found that the contents stored in the address are the last four digits of my username. It proves that I can control the pin when calling the username() function.



Then I try to hijack the function. We will use the disassemble command in gdb to see the memory address of each operation. When calling a function, the program will jump to the specified address. If I overwrite this address, the program will jump to the address I expect to execute. In this problem, I want it to jump directly to the address of the impossible().



In the function impossible, we will find the address before executing the system() which is 0x0804914b. Aim to jump to that address, I need to hijack the function after scanf the pin, which is fflush. I will use scanf to overwrite the address point to the fflush. The most important thing here is that we need to shut down the gdb first and then restart the gdb to disassemble the fflush. Otherwise we can not find the address that continues execution.



As we can see, the program will jump to where the fflush code is located, i.e. 0x804c010 and execute it. Next, I need to overwrite the contents in addresses 0x804c010d to 0x804c013 with the address of the system call. When the program loads the address to run the fflush code, it instead loads the address of the system call and jumps to that address.

I need to do several last steps. First, I need to set enough buffers into the ‘pin,’ 16 characters. Then store the address which we will jump to in little-endian. At last, transfer the jump address into a decimal number which means transfer 0x0804914b to 134517067. The only thing I need to do is put all things together:

