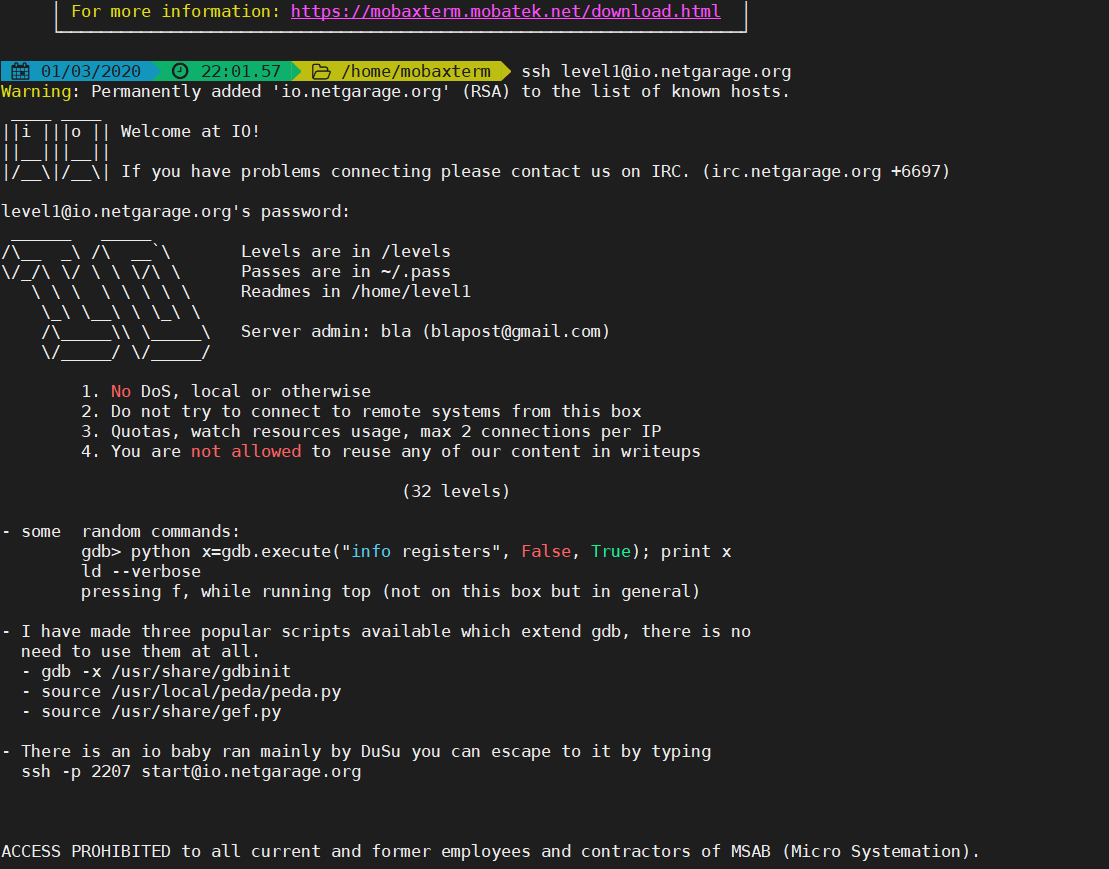
# OHTS – LAB 01

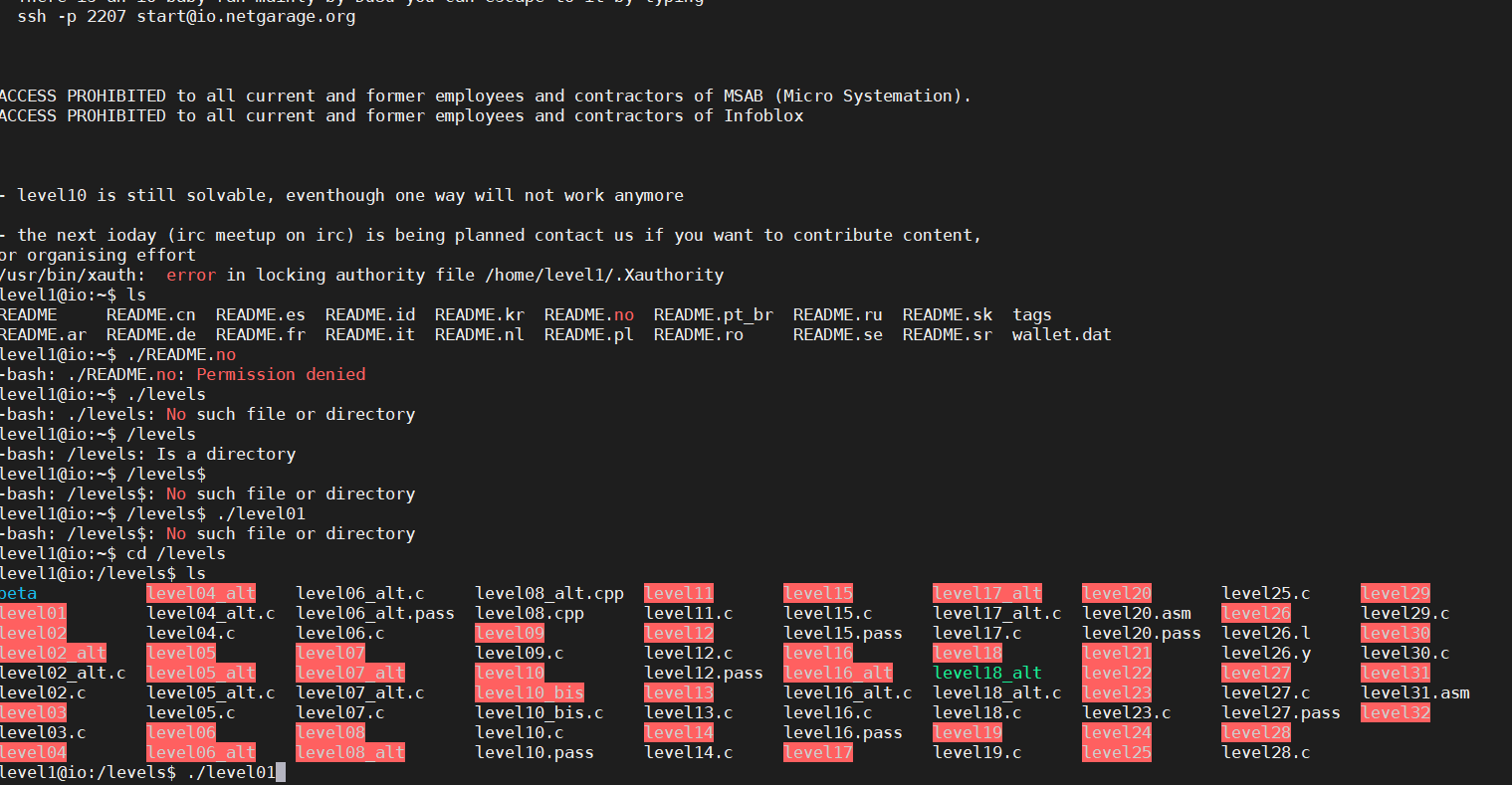
# It17011808 – K.B.S.B. Kudagoda.

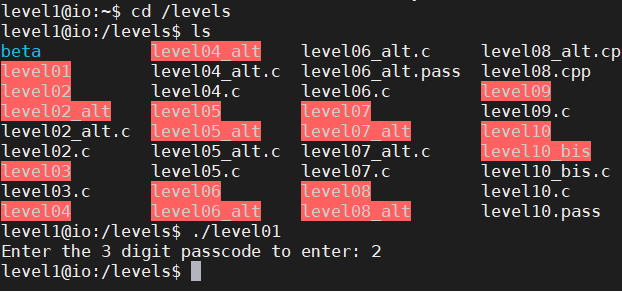
## Level 01



Password was given for level 01. In here we have to find password for level 2

Firstly, find out what are the files according to particular levels.



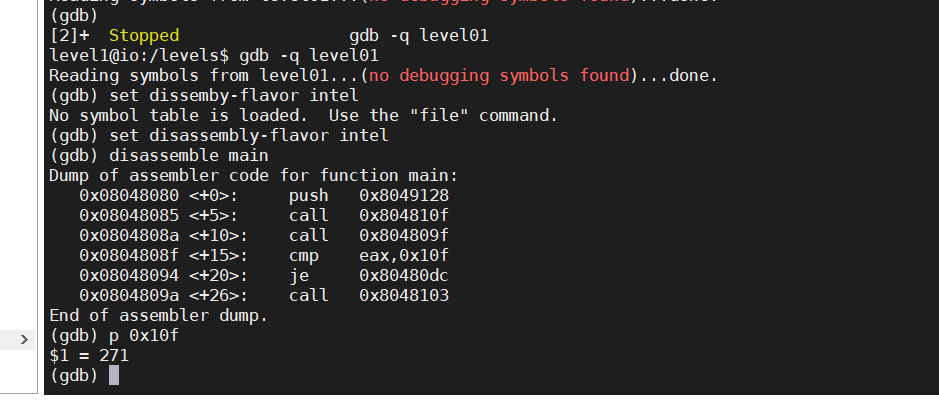


In here, there is only one file for level 01. Then execute that file for more details.

Its prompt to enter 3 digit number.

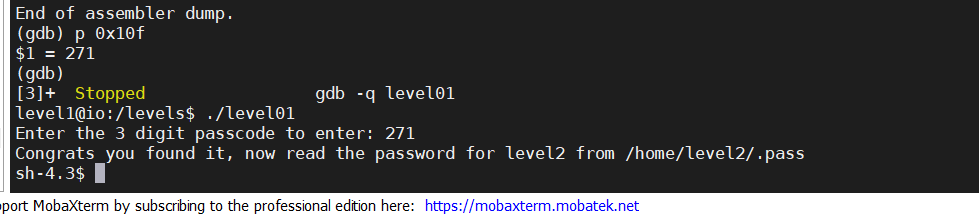
Then launching the program under gdb .

Get disassembly code for main program to find out inputs.

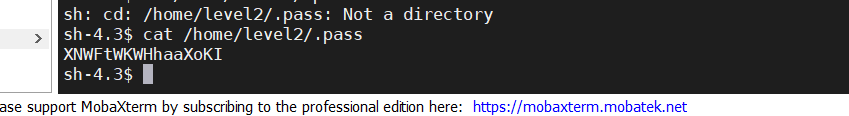


In here, we can see second call ask for the user input (the password). Then the program compare a fixed value with the value of the register eax.

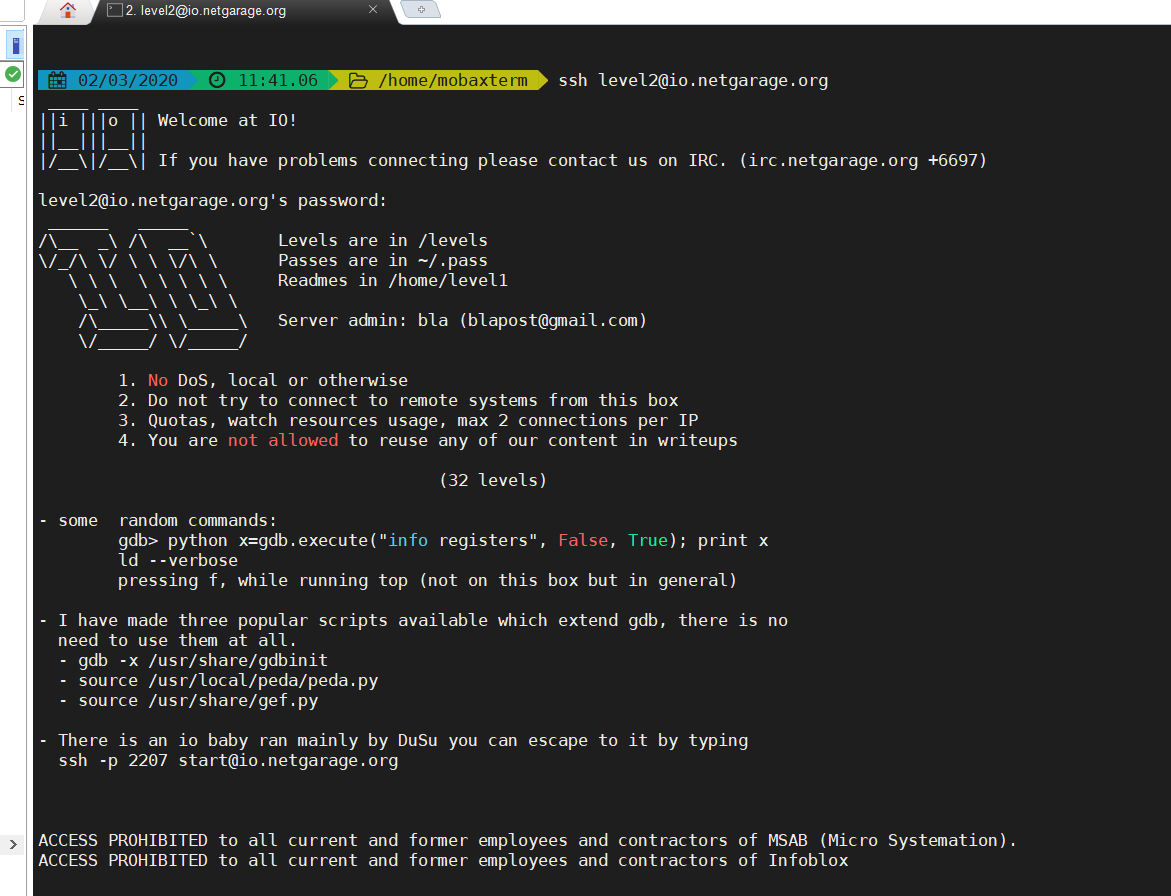
The value is a hexadecimal value, we can display its decimal value with p in gdb



Then execute level1 and enter the password for get level 2 password.



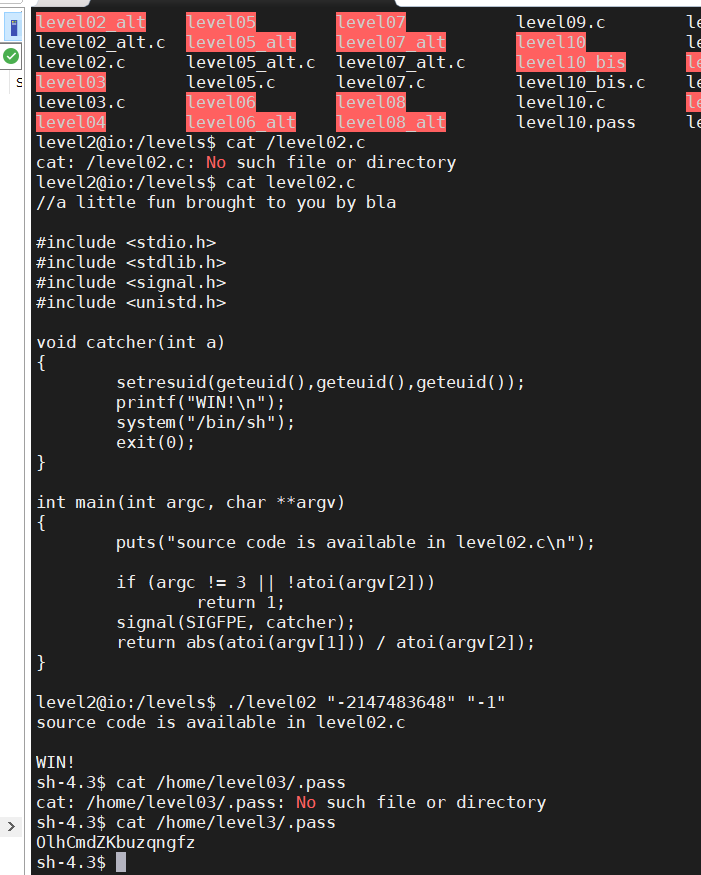
# Level 02



Input the password and enter to the next level of the series.

As previous level looking for file regarding level 2.

We can clearly see there is a source code for level 02



In here, there is a catcher function. If the catcher function is called, it will set the current user identity, and print a win message, before spawning a new shell. This is clearly the indication that we need to raise a SIGFPE exception.

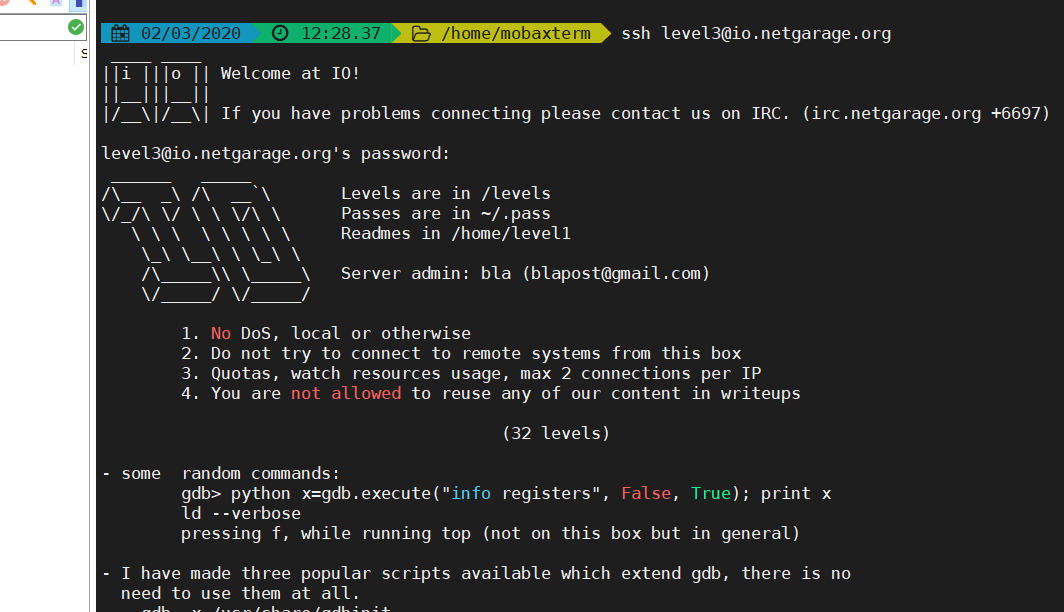
The SIGFPE can me triggered with a 1/0 or a sqrt(-1)

What we can do instead is try to use an integer value outside of the bound of the integer definition. We can see on the abs reference page that the most-negative value to be out of range is -2147483648

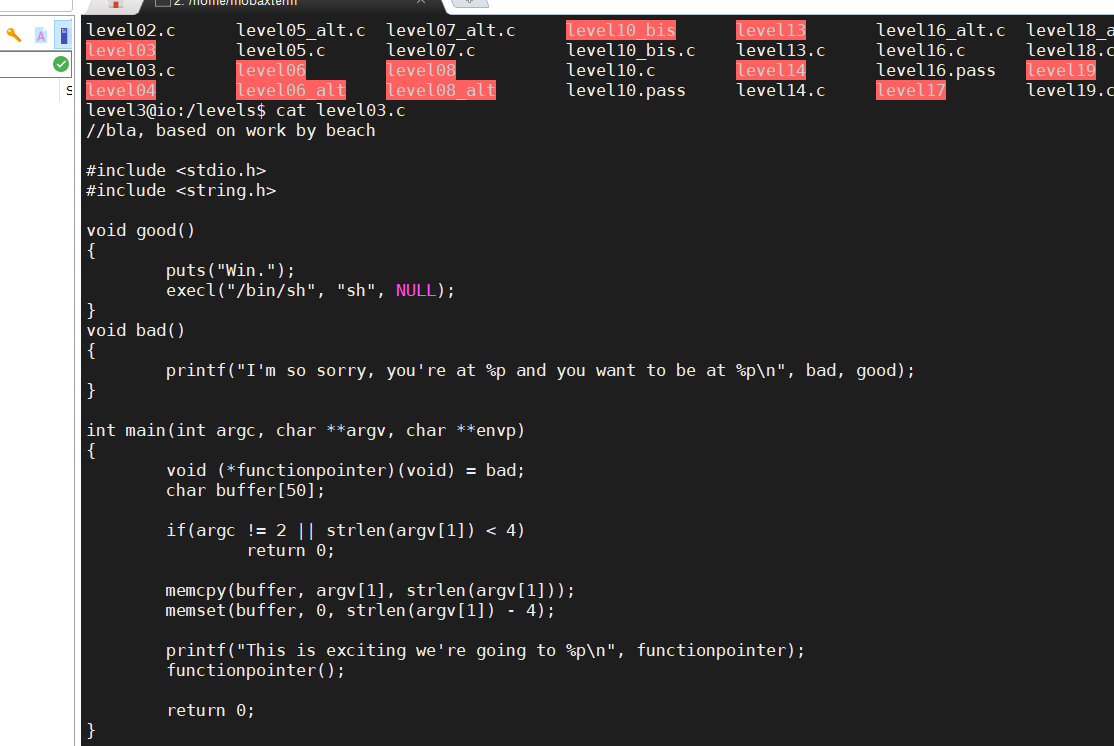
But, if we send the value 2147483648 to abs, which is an incorrect value, it should raise an error.

Then search password in /home/level3/.pass .

# Level 03



In here also we can find source code for level 3.



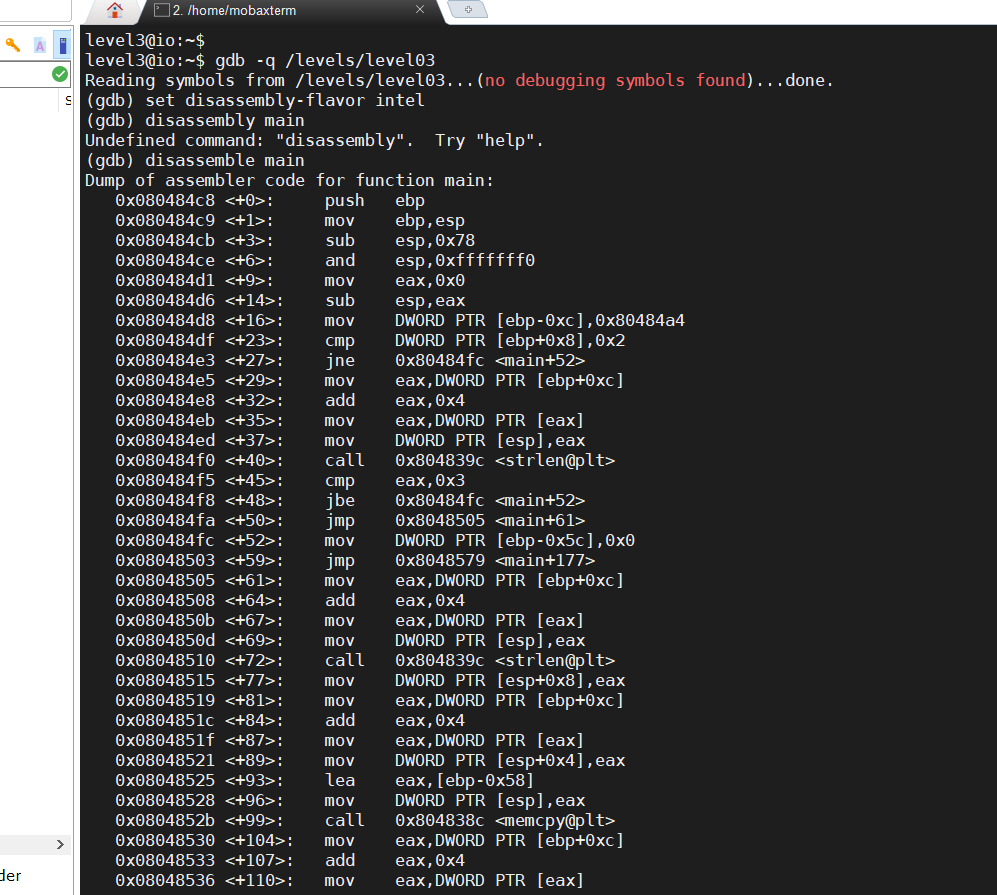
There are two functions called good and bad. As we can see good function is the thing, what gives us more reliable result for our expectation. We want to find a way to execute good function.

Moving on to the main function, we can see that we start by allocated a function pointer which points to the address of the 'bad' function. We can reason that we want to find some way to make the function pointer point to the 'good' function instead of the 'bad' function. We then create a 50 byte character array called 'buffer'.

We can see that the number of bytes we copy into the buffer depends only on the size of our input.

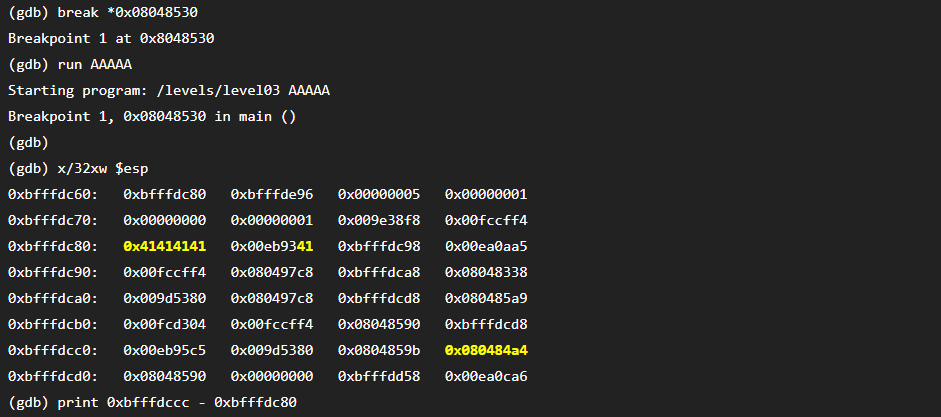
If we overflow buffer from more than the size of buffer memcpy overwrites the memory until our argument has been stored.

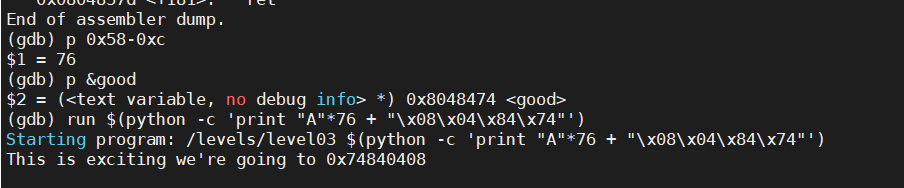
Lets, exploit the vulnerability .



Let's put a breakpoint right after the memcpy function is executed, run the program with the argument 'AAAAA', and see what we find

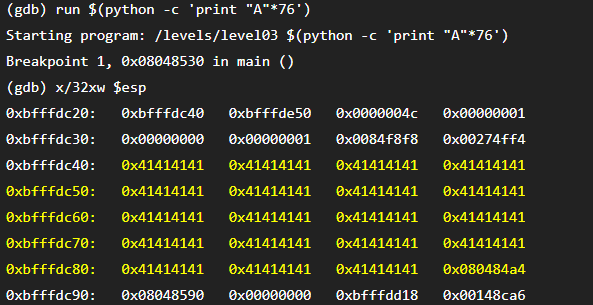
We first set our breakpoint to the address containing the instruction after the memcpy function. Then, we run the program using 'AAAAA' as the provided argument, and once the breakpoint is reached, we tell GDB that we want to examine 32 hex-words starting at ESP and going towards higher memory addresses. This allows us to see the stack.





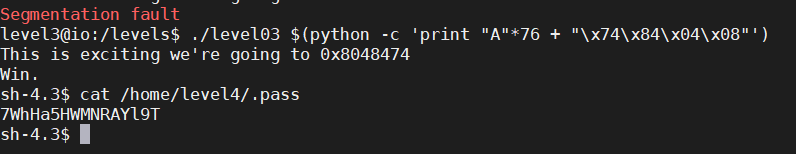
We can print 76 'A's using Python with the following command:

python -c 'print "A"\*76'

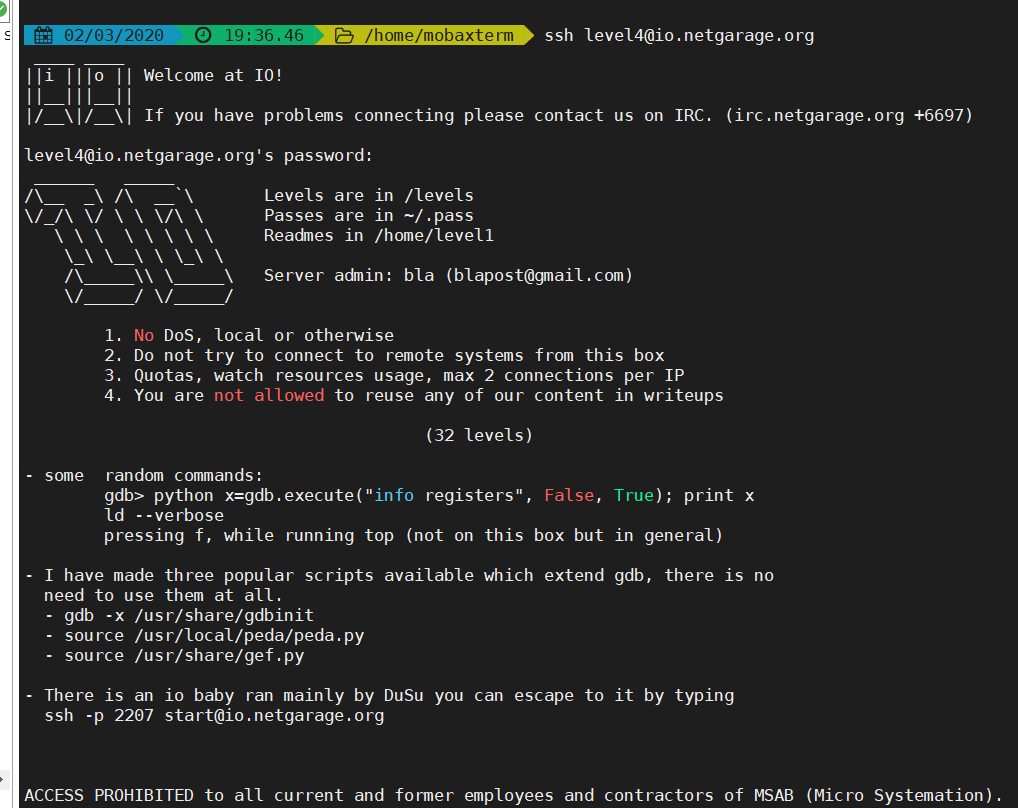


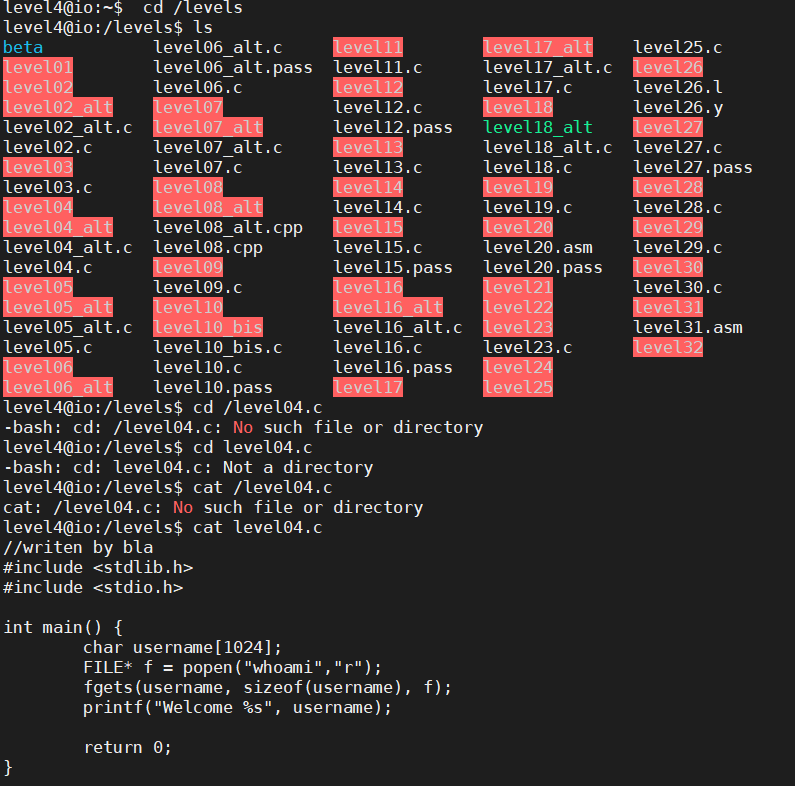
successfully overwritten data to the start of the function pointer. Now we just need to overwrite the address of the function pointer to the correct address.

we will use Python to fill the first 76 bytes, and then we will overwrite the function pointer with the correct address and run.



# Level 04





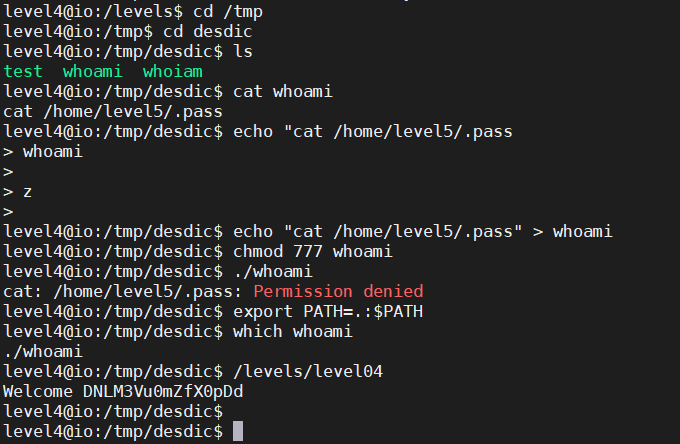
This program executes a shell command and prints the results.

Then, substitue the command called by popen with one that will help us advance, like

cat /home/level5/.pass

after that,find whoami file location and find vaiable path.

And execute level04 and get password.



# Level 05

