Lab Environment:

```
# vim: syntax=apache ts=4 sw=4 sts=4 sr noet
<VirtualHost *:80>
        ServerName http://www.SeedLabSQLInjection.com
        DocumentRoot /var/www/SQLInjection
</VirtualHost>

[06/12/19]seed@VM:.../sites-available$ sudo service apache2 start
[06/12/19]seed@VM:.../sites-available$
```

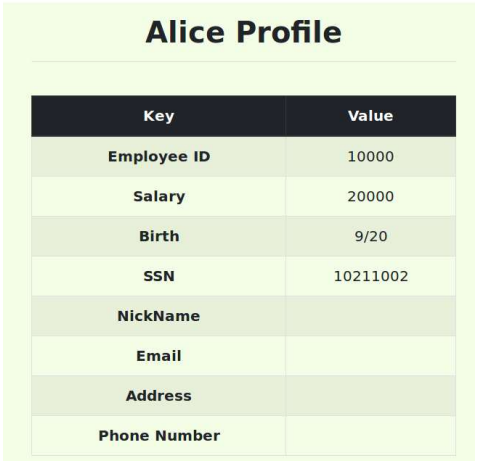
We verify that the SQL injection URL is linked to our source files, this is done in /etc/apache2/… We also restart the apache server to make sure everything is running correctly.

Task 1: Get Familiar with SQL Statements

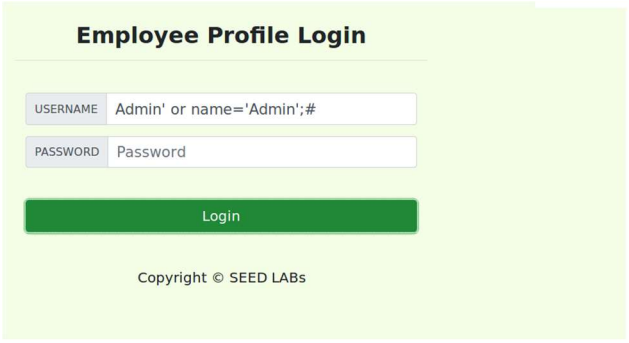
```
mysql> select * from credential where name like 'Alice';
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
1 row in set (0.00 sec)
```

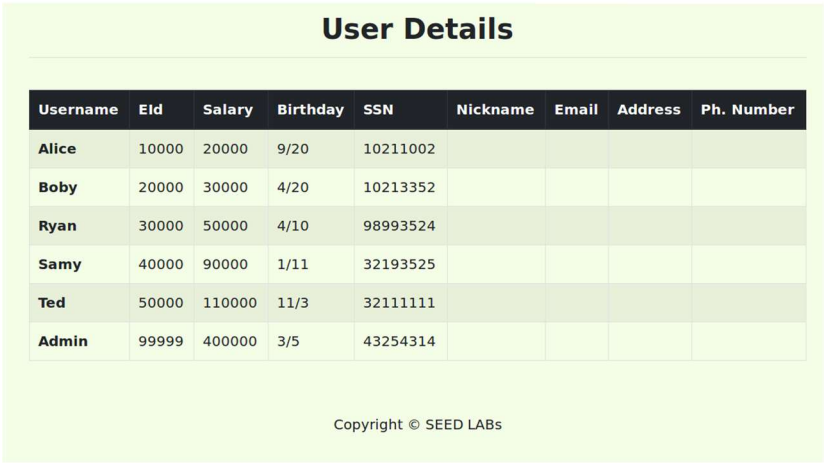
After loading in the User database, we print all the fields of Alice in the credential table.

Task 2.1: SQL Injection Attack from webpage



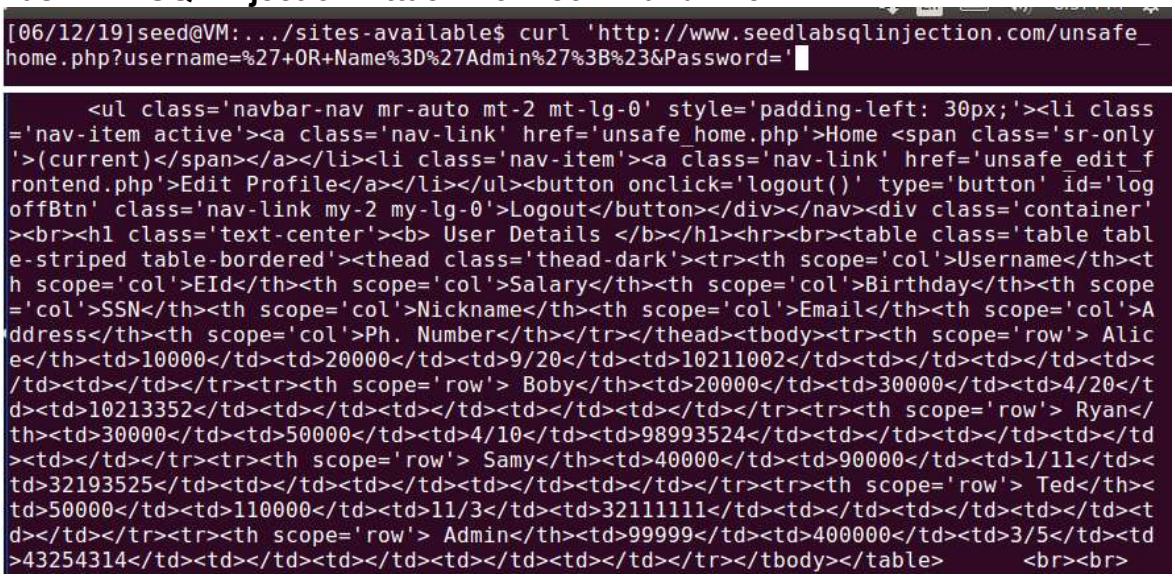
At first, we tried the well-known 1=1 attack however this was only partially successful as we were only able to log into Alice but not the admin account. This is because we log into the first account that was added into the table which happened to be Alice. Often times admin will be the first account to be added and, in that case, we would have been successful.





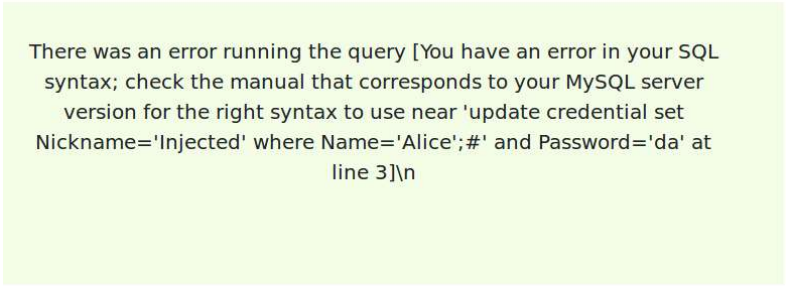
We can modify the previous statement and directly choose which account we want to access. After logging in, we see the information of all the users meaning we truly have the admin account.

Task 2.2: SQL Injection Attack from command line



The curl statement is exactly the same as what we put into the username field the only difference being that the input is HTML URL encoded. The results might be a little bit harder to read since its being read from an HTML source, but the information is still there. We can use other Linux commands to search for specific names and such.

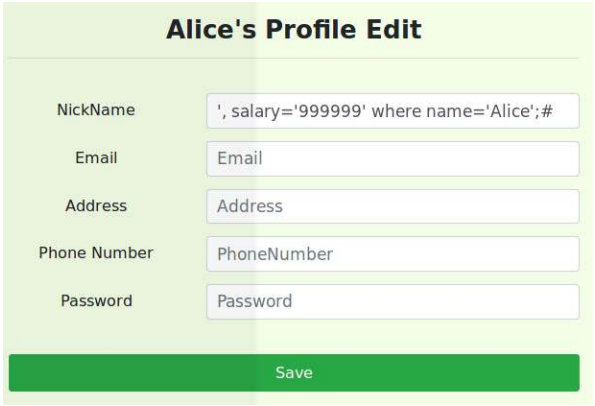
Task 2.3: Append a new SQL statement



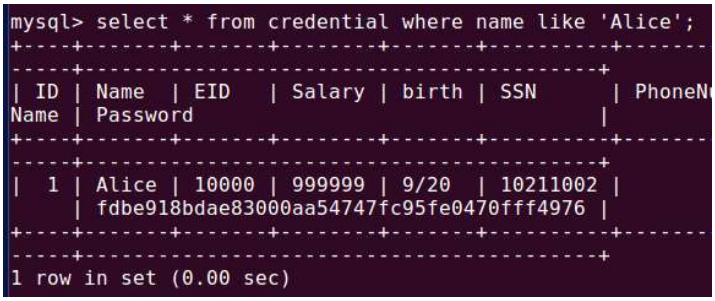
We simply add a semicolon to complete an SQL query which allows us to add another query right after. Unfortunately, our attack didn't work because MySql has countermeasures to detect and block multiple queries from being executed from PHP.

Task 3.1: Modify your own salary

We login into Alice's account using her own username and password. We observe that her current salary is only 20000, lets change that.



We enter *', salary='999999' where name='Alice';#* into the nickname field. This allows us to directly set our salary.

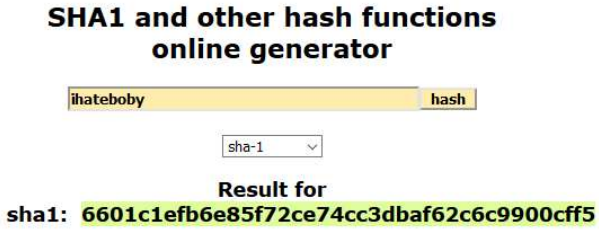


Using MySql DBMS we can verify that indeed Alice's new salary is 999999.

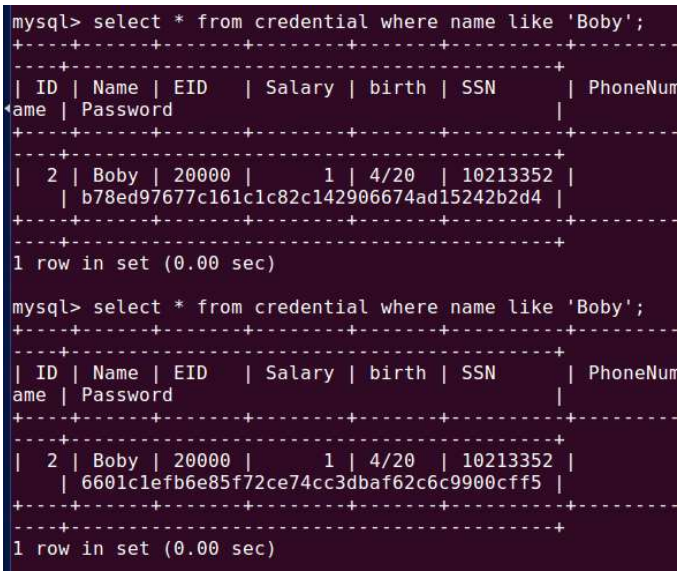
Task 3.2: Modify other people' salary

We enter ', salary='1' where name='Boby';# into the nickname field of Alice's profile. This query changes the salary of anyone named "Boby" to 1. We can be more precise and use Boby's ID to specifically target him. In our case, the database only has one Boby so this query will suffice.

Task 3.3: Modify other people' password



Looking into the PHP code we can clearly see that a SHA-1 hashed password is store. These and many other hash functions are widely available online, so we simply generate a hash of the desired password.



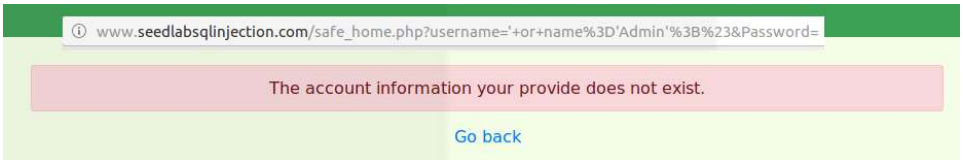
We verify that the hash changed by using the MySql DBMS. Using the nickname field we type *', password='6601c1efb6e85f72ce74cc3dbaf62c6c9900cff5' where name='Boby';#*

Using the username "Boby" and the password "ihateboby", we can now successfully log into his account.

Task 4: Countermeasure — Prepared Statement

```
      $conn = getDB();
      // Sql query to authenticate the user
      $sql = $conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumb
er, address, email,nickname,Password
      FROM credential
      WHERE name= ? and Password= ?");
      $sql->bind_param("ss", $input_uname, $hashed_pwd);
      $sql->execute();
      $sql->bind_result($id, $name, $eid, $salary, $birth, $ssn, $phoneNumber, $
address, $email, $nickname, $pwd);
      $sql->fetch();
      $sql->close();
```

The file safe_home.php includes the prepared statement code we need to protect our server from injection attacks.



When we try using the same query from task 2.1 we get an error. This is because our input is not treated as a string input, as data rather than code. With this protection enabled we are unable to perform any kind of injection on this server.

Code:

```
      $sql = $conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber,
address, email,nickname,Password
      FROM credential
      WHERE name= ? and Password= ?");
      $sql->bind_param("ss", $input_uname, $hashed_pwd);
      $sql->execute();
      $sql->bind_result($id, $name, $eid, $salary, $birth, $ssn, $phoneNumber,
$address, $email, $nickname, $pwd);
      $sql->fetch();
```