Spring 2020

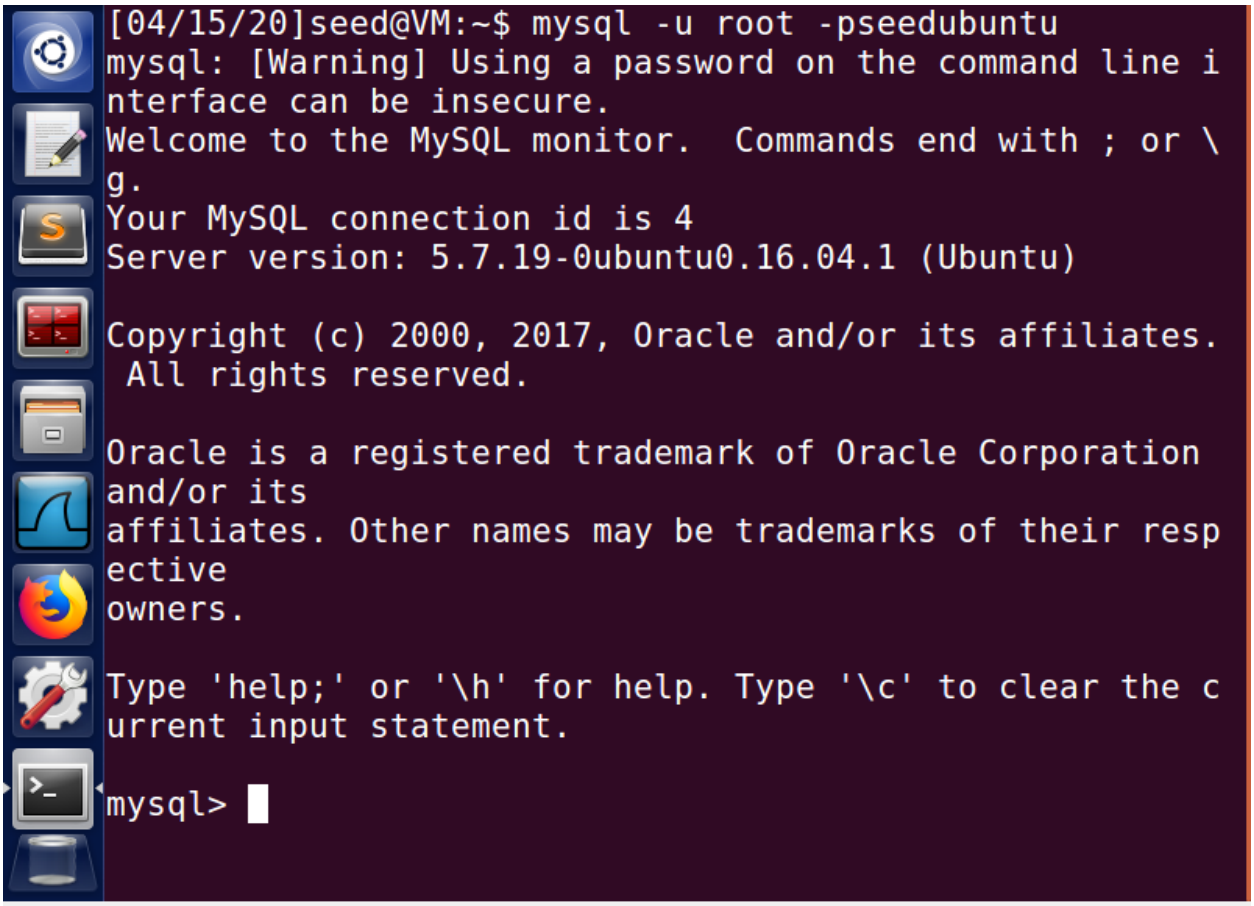
CSE-5382-001 – Secure Programming

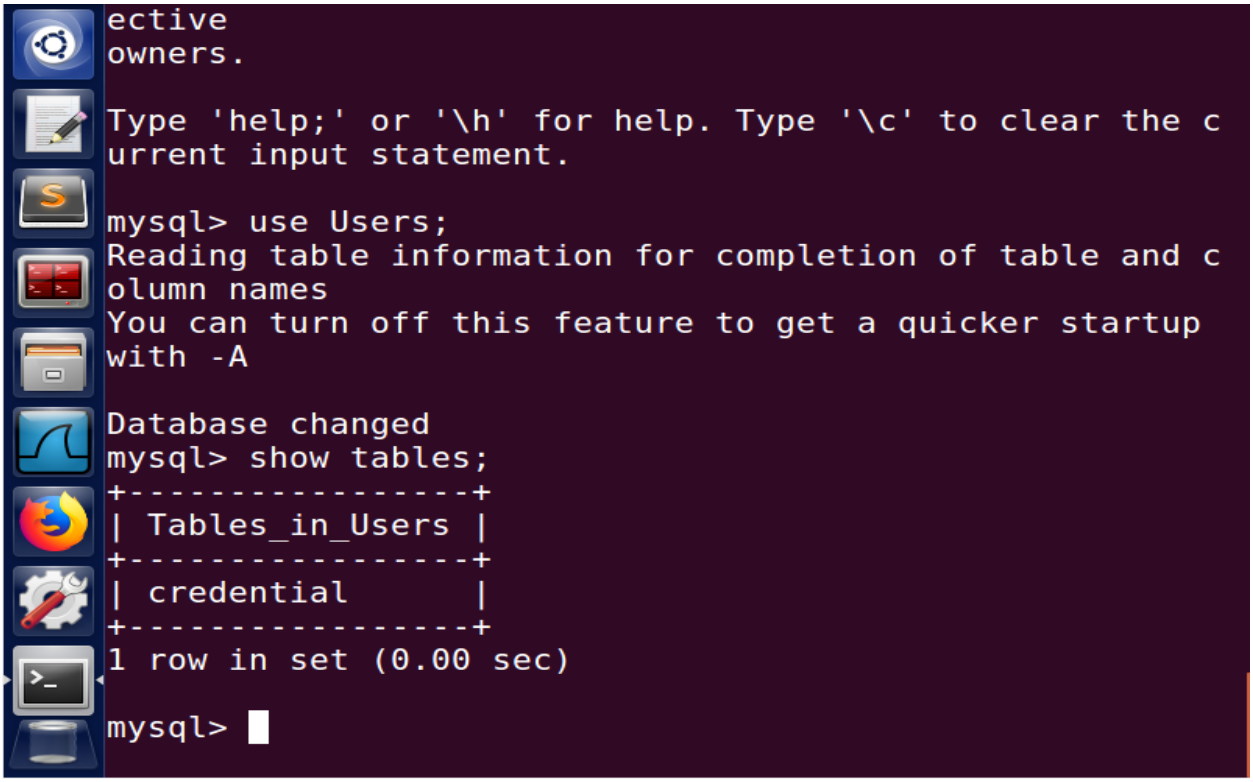
Homework Assignment 9 – SQL Injection Attack

Name	UTA ID
Goutami Padmanabhan	1001669338

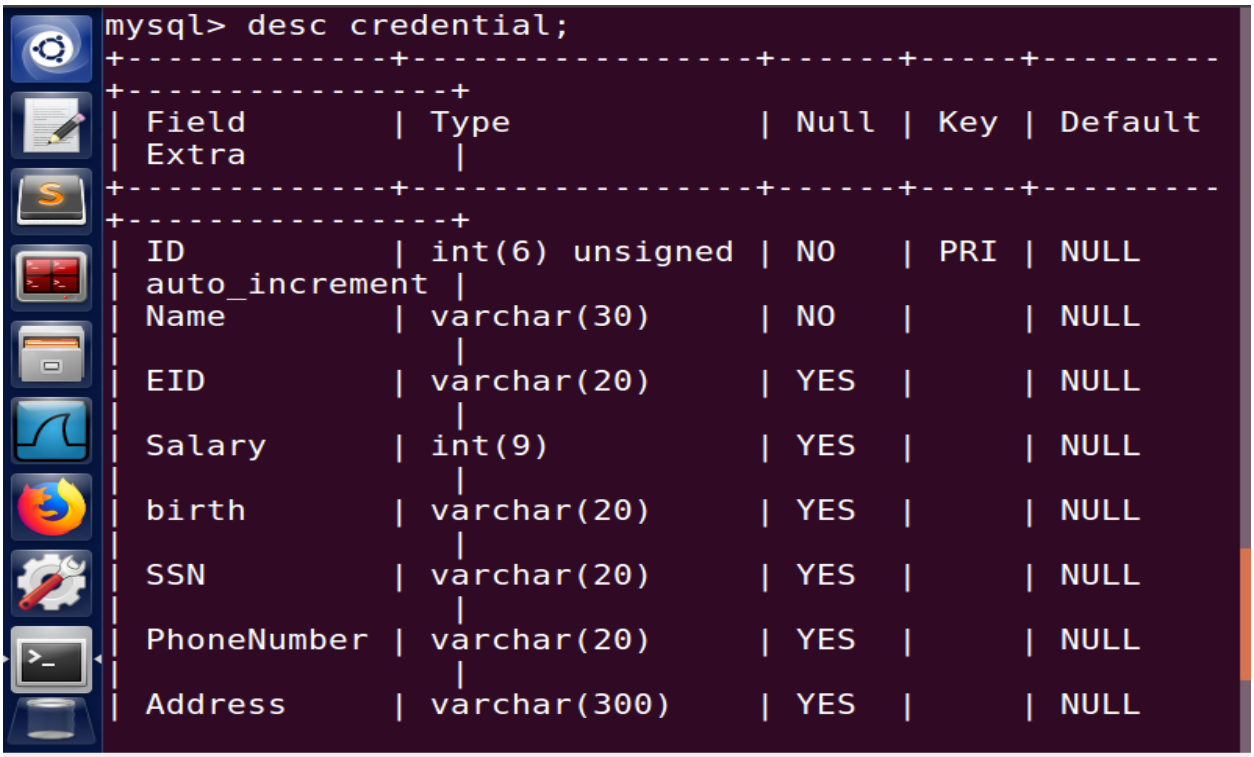
3.1 Task 1: Get Familiar with SQL Statements

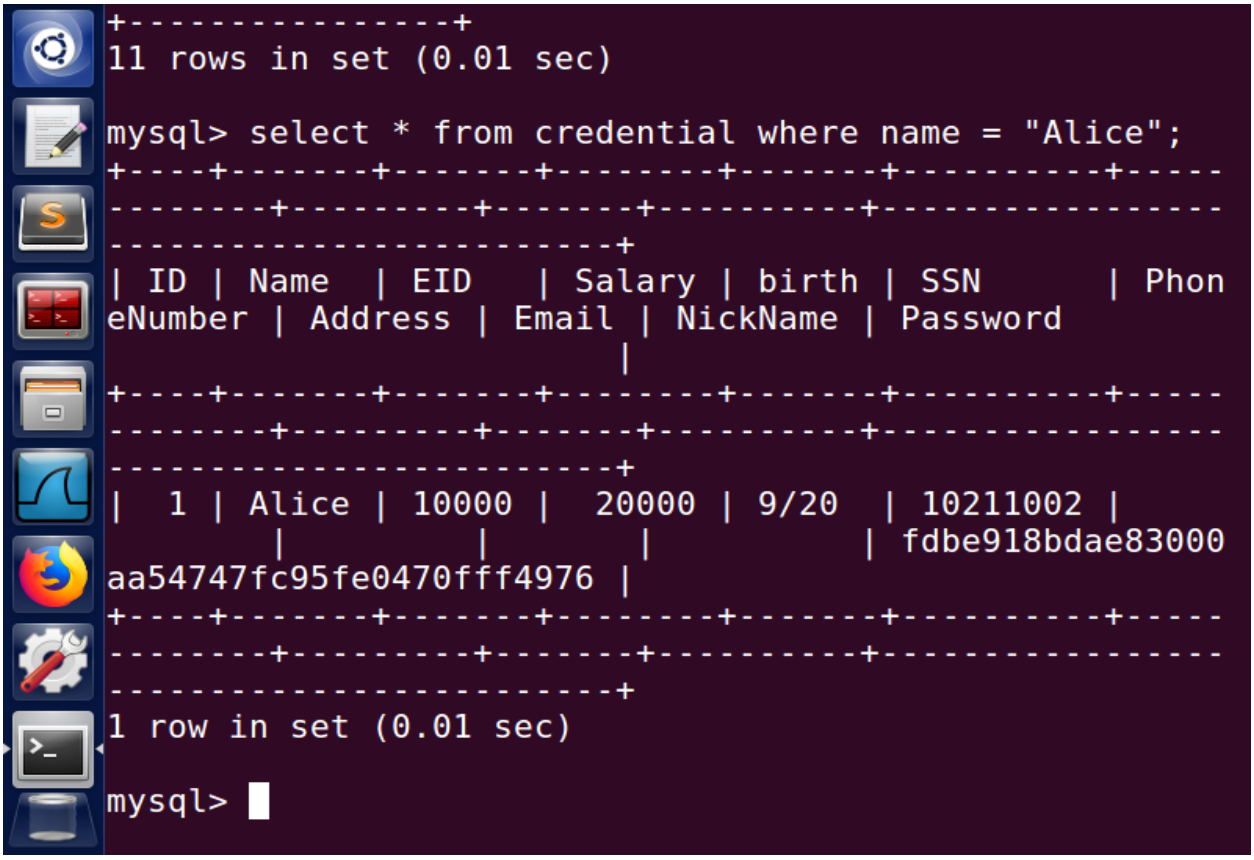
The purpose of this task is to understand how to use MySQL environment and understand how to get the output of SQL commands. For this, we login to mysql environment as a root user by giving its password. Every time we want to use the mysql environment in this assignment, we use the following steps shown in the screenshots below.





We load the Users database in the mysql environment and find the table name ‘credential’. We find out the column names of the credential table.



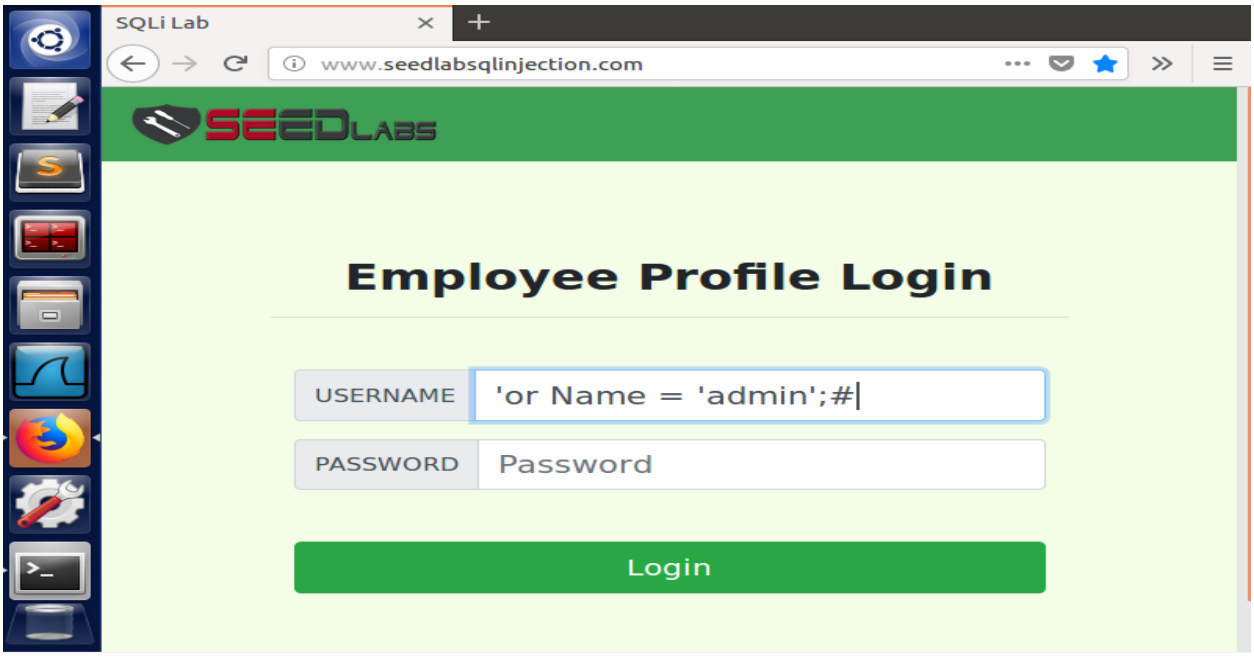


Conclusion: We login to mysql environment and load the Users database. We find that it has one table named ‘credential’ and find out the column names used in this table. We then query the credential or record of the person named ‘Alice’. By doing this task, we make ourselves comfortable with mysql environment and use the same knowledge throughout the assignment.

3.2 Task 2: SQL Injection Attack on SELECT Statement

Task 2.1: SQL Injection Attack from webpage.

The purpose of this task is to understand how SQL injection attack is done using the SELECT statement in MySQL. Our task, as an attacker, is to log into the web application, http://www.seedlabsqlinjection.com/ without knowing any employee's credential. In this task we do the attack from the webpage.



We exploit this web application's PHP code unsafe home.php, located in the /var/www/SQLInjection directory by trying to modify the SQL query from the Username field of this login page. We modify the `$input_uname` variable in this code to `'or Name = 'admin';#`. This makes any user to login as admin and comments out the rest of conditions in the SQL statement.

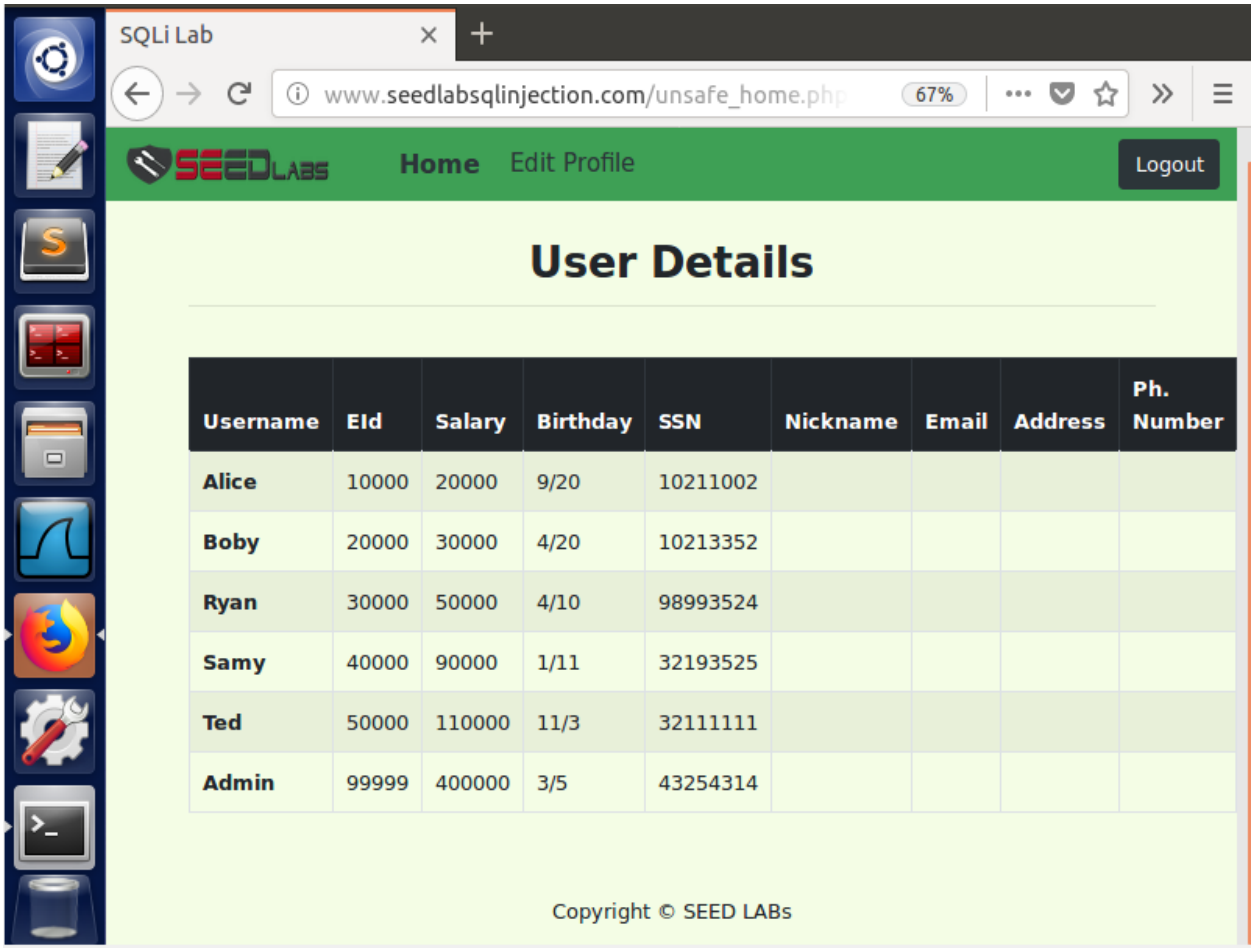
SQL code before the attack

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

SQL code after the attack – After replacing `$input_uname` variable to `'or Name = 'admin';#`

```
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email,
nickname, Password
FROM credential
WHERE name= ''or Name = 'admin';#' and Password='$hashed_pwd'";
$result = $conn -> query($sql);
```

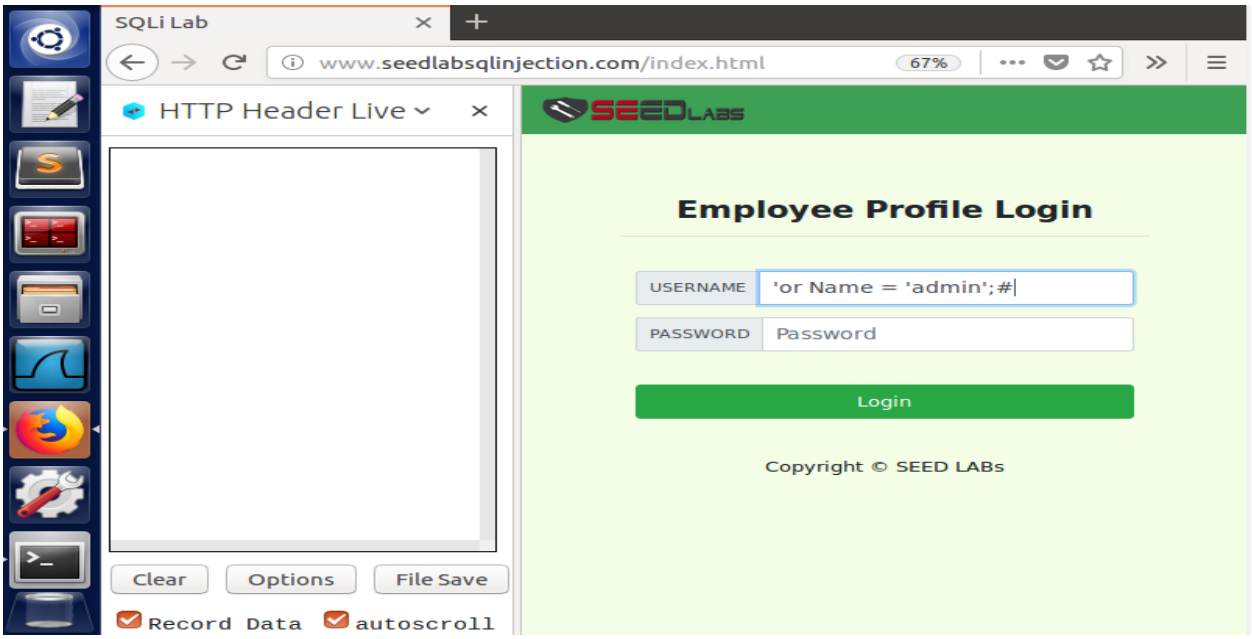
The # in this code comments out `' and Password='$hashed_pwd'";` this line.



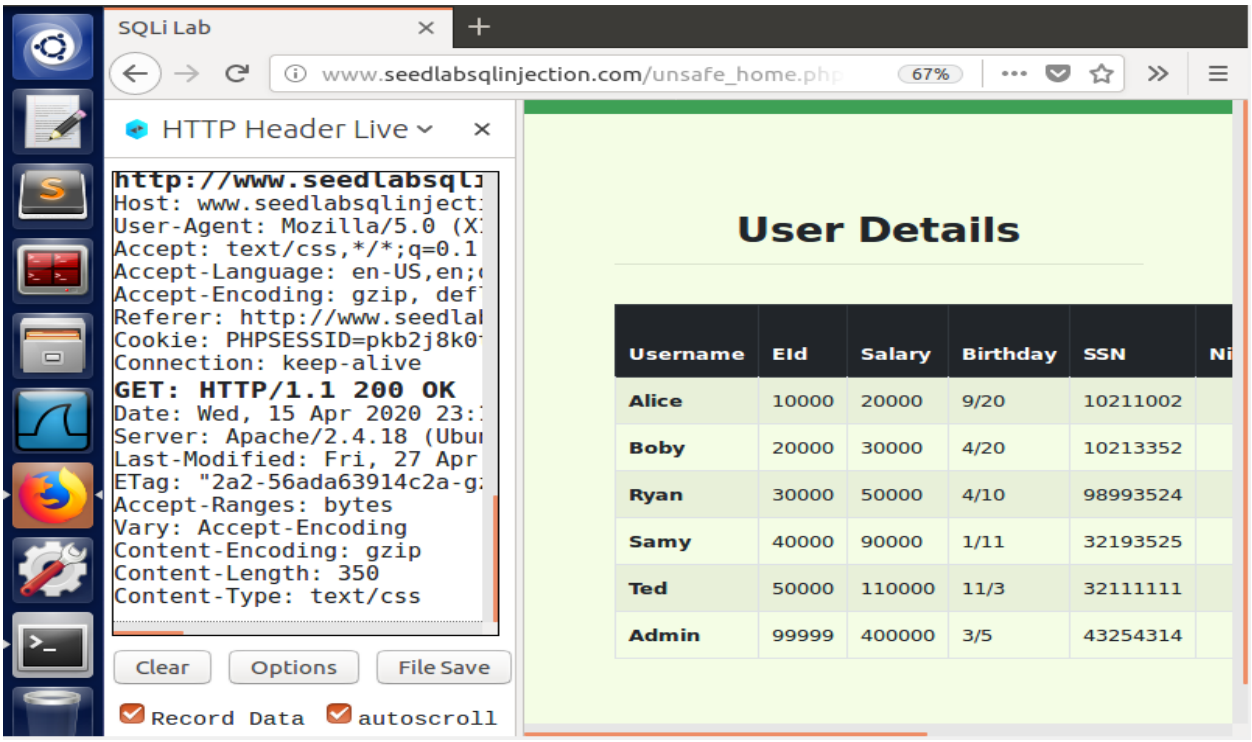
Conclusion: Since the PHP code of this web application was not coded in a secure way, it is vulnerable to SELECT statement SQL injection attack. SQL statement was not coded with prepared statement. This led to using the $input_uname variable to make the attack. We simply write the code we want in the Username field of the login page and use a # at the end of the code to exploit this vulnerability. By using this, we login as admin and get access to the entire web application without even knowing the login credentials.

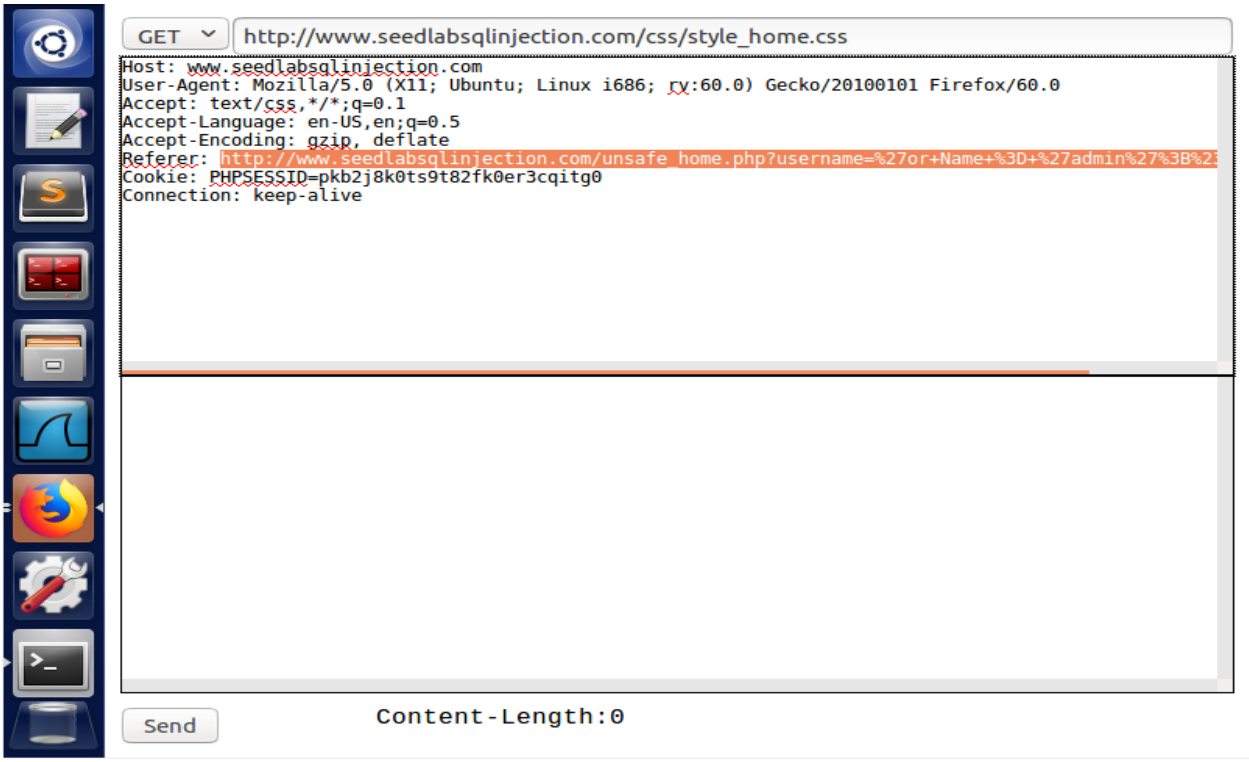
Task 2.2: SQL Injection Attack from command line.

The purpose of this task is to make SQL injection attack from the command line. Our task, as an attacker, is to log into the web application, http://www.seedlabsqlinjection.com/ from the command line without knowing any employee's credential and display all the results in the command line.

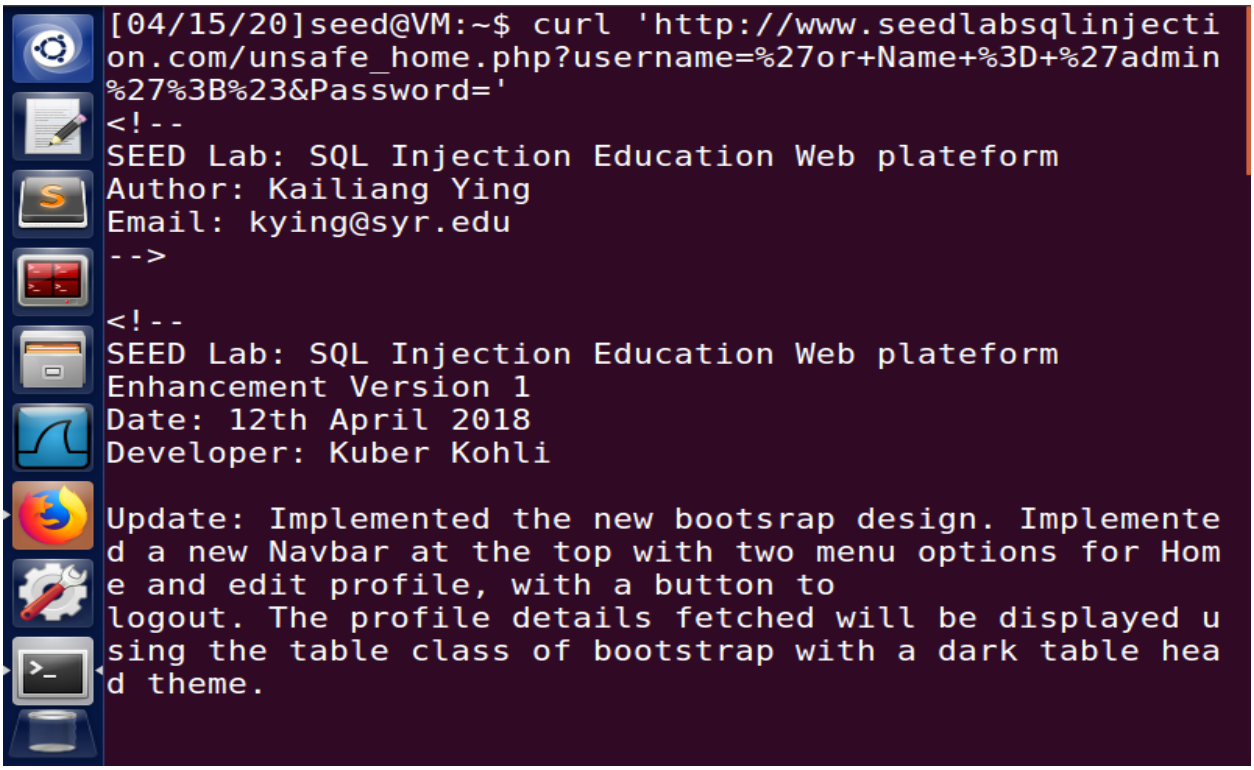


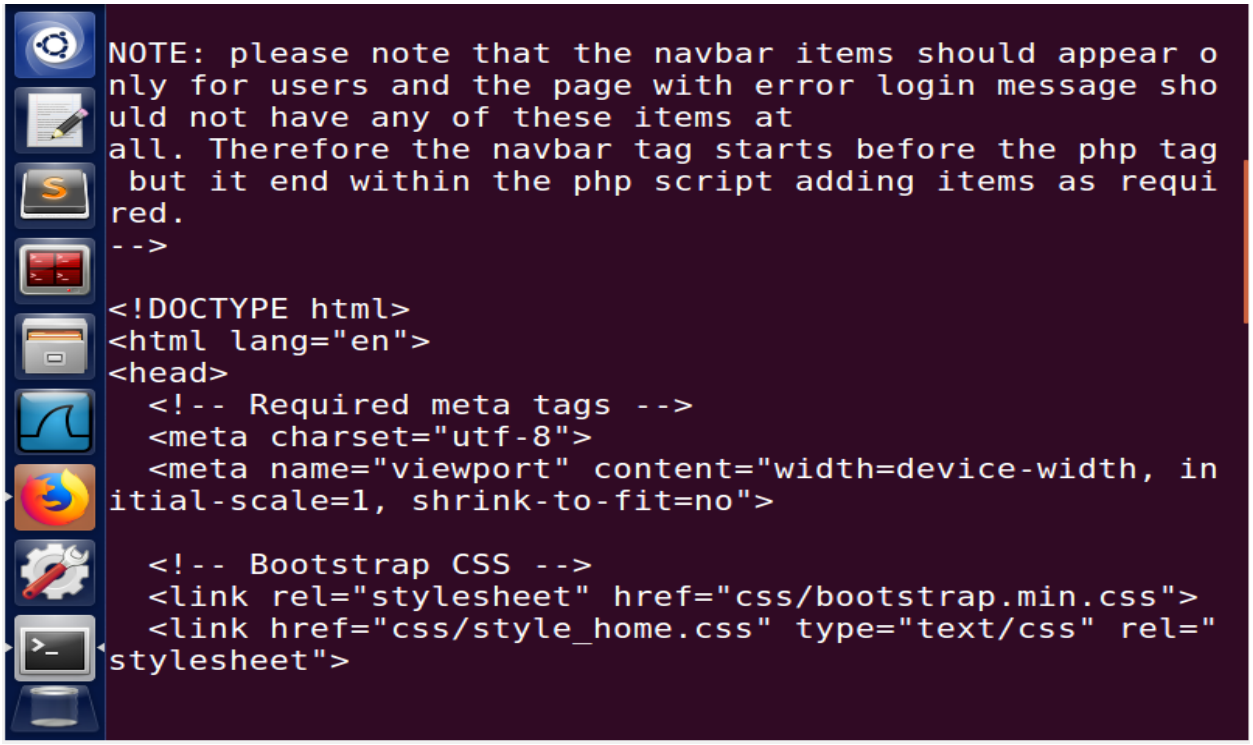
In order to achieve this, we must know the exact GET request URL that is used in this web application to login as admin. We find this out by using HTTP Header Live add on in Firefox browser. We perform the same webpage attack we did in Task 2.1 again to get the GET request URL.





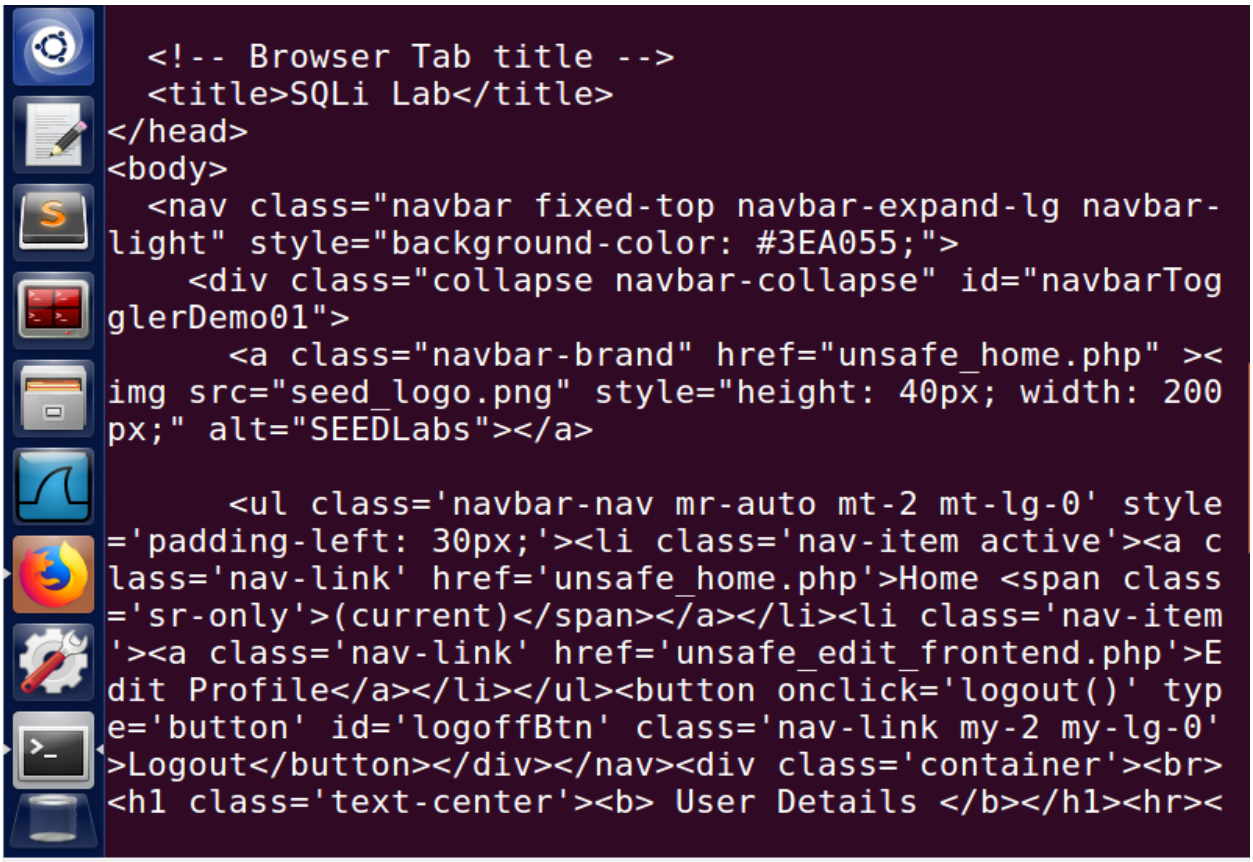
The GET request URL highlighted in the screenshot is noted down and we use the command line curl command to make the same SQL injection attack from SELECT statement. We use the command `curl http://www.seedlabsqlinjection.com/unsafe_home.php?username=%27or+Name+%3D+%27admin%27%3B%23&Password=`. We use the username field to write our query in his URL.



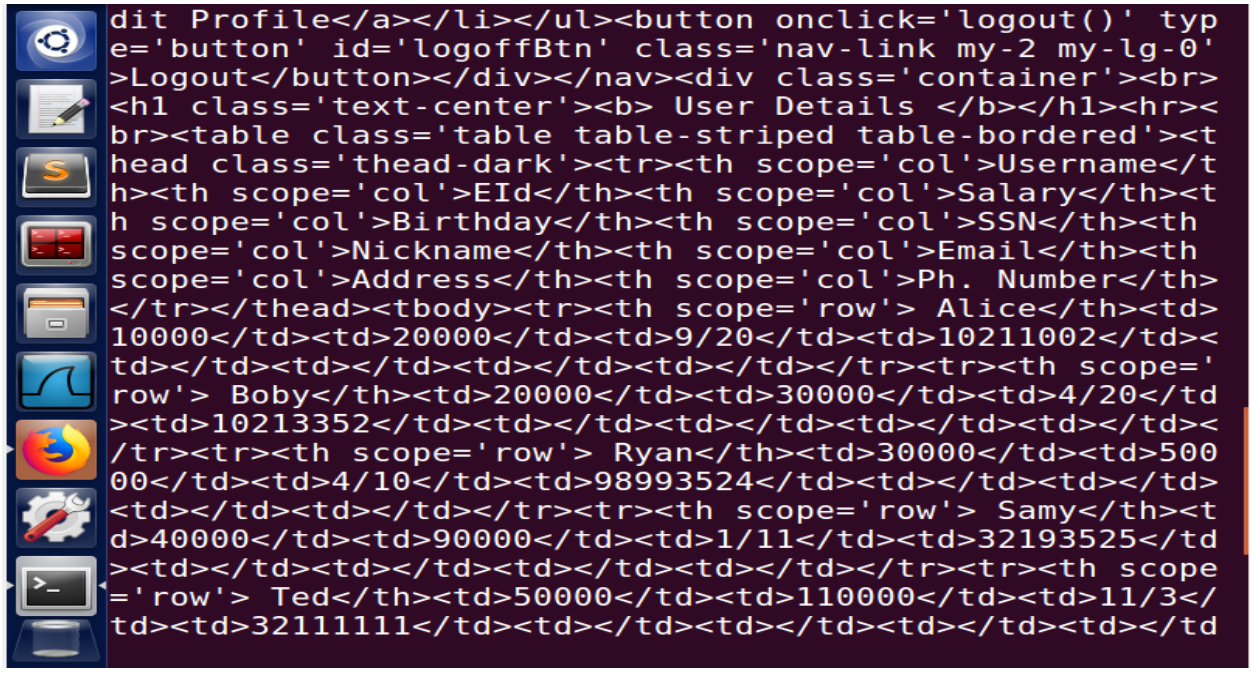


```
NOTE: please note that the navbar items should appear o
nly for users and the page with error login message sho
uld not have any of these items at
all. Therefore the navbar tag starts before the php tag
 but it end within the php script adding items as requi
red.
-->

<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, in
itial-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">
```

When we use the curl command, we can login to the application just like admin and view all the details.



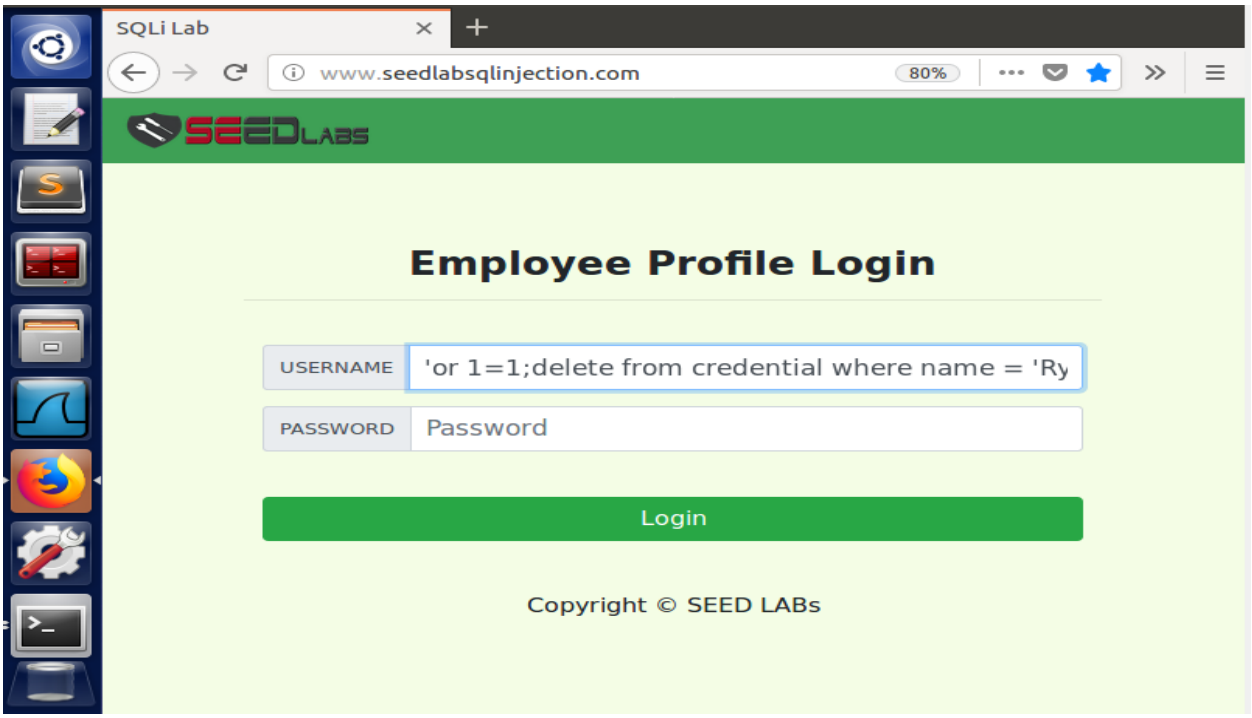
```
  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>
<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-
light" style="background-color: #3EA055;">
    <div class="collapse navbar-collapse" id="navbarTog
glerDemo01">
      <a class="navbar-brand" href="unsafe_home.php" ><
img src="seed_logo.png" style="height: 40px; width: 200
px;" alt="SEEDLabs"></a>

      <ul class='navbar-nav mr-auto mt-2 mt-lg-0' style
='padding-left: 30px;'><li class='nav-item active'><a c
lass='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'>E
dit Profile</a></li></ul><button onclick='logout()' typ
e='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><
```



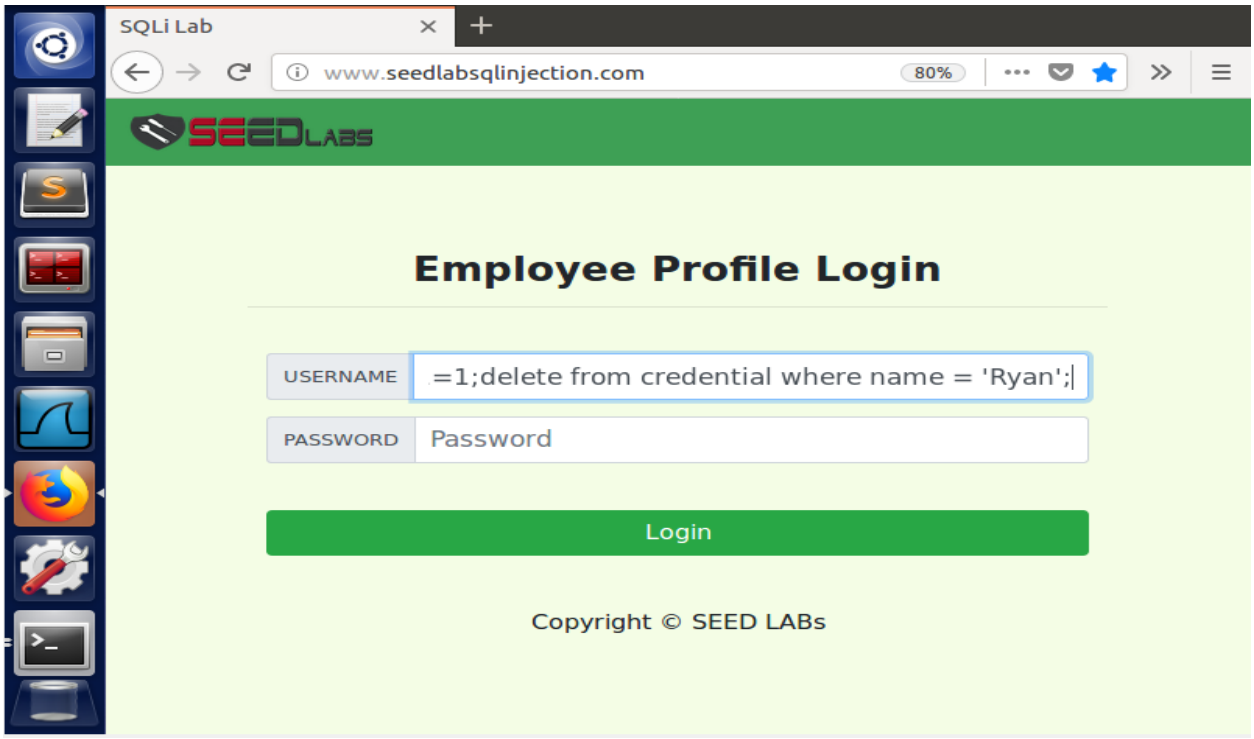
Conclusion: By knowing the exact parameters to be used in GET request URL using the HTTP Header Live add on, we can easily make the SQL injection attack using the curl command in the command line as well. We exploit the SELECT statement in the PHP code of the web application and use the same conditions in the URL, we used while making the attack from the webpage. We login to the application as admin and all the data is displayed. We did not require any credentials to login to the application.

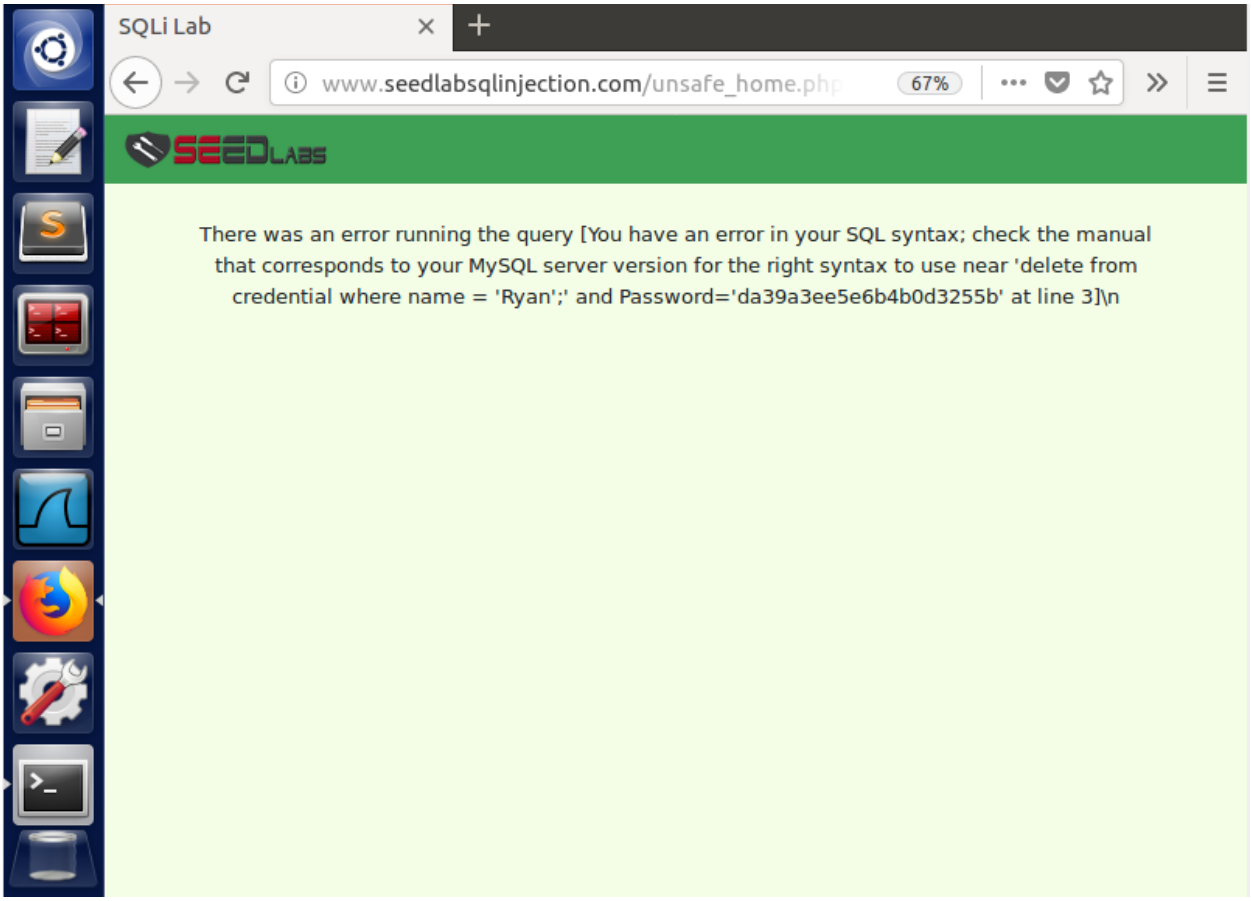
Task 2.3: Append a new SQL statement.

The purpose of this task is to modify the database by appending a new SQL statement by using the vulnerabilities in the web application. In this task we try to delete the credentials of the record whose name is ‘Ryan’.



We enter the extra SQL code as shown in the screenshots and try to exploit the SQL code vulnerability.



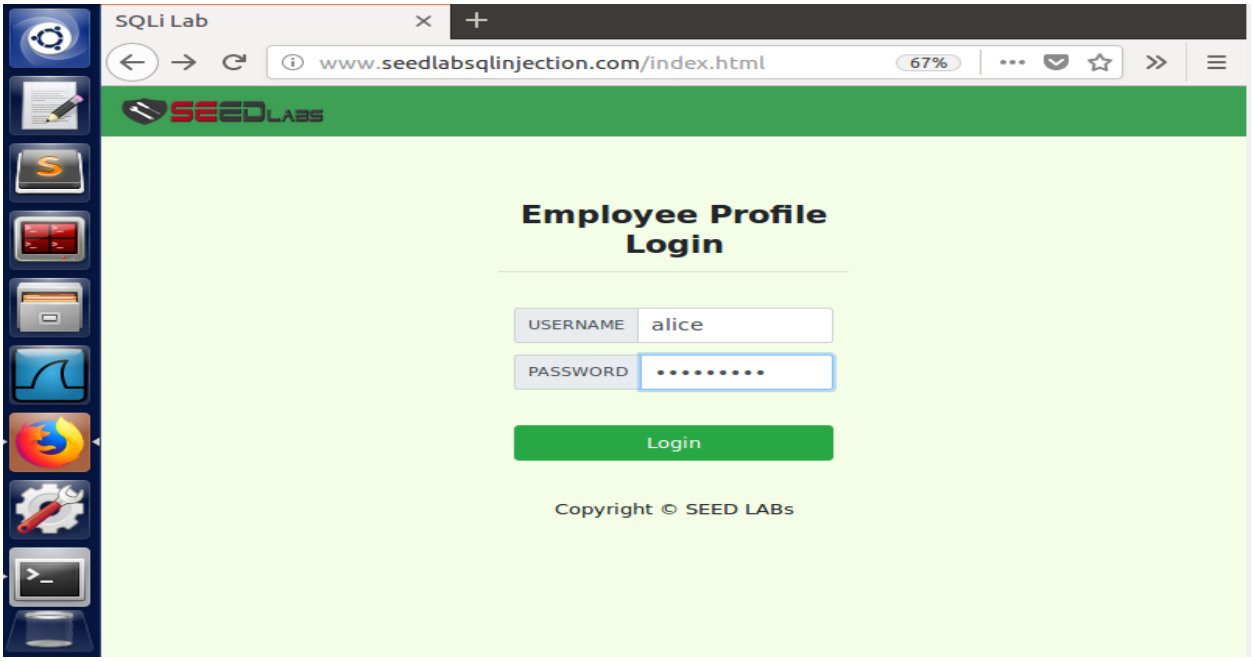


Conclusion: We append another SQL statement in the username field of the webpage and try to delete Ryan's record from the database. When we try to login, it throws an error saying that it is an SQL syntax error. This is because of the in-built countermeasure in MySQL that prevents multiple SQL statements from executing invoked from the same PHP code. Because of this countermeasure, we are unable to perform the SQL injection attack by using the SELECT statement in SQL and modify anything in the database.

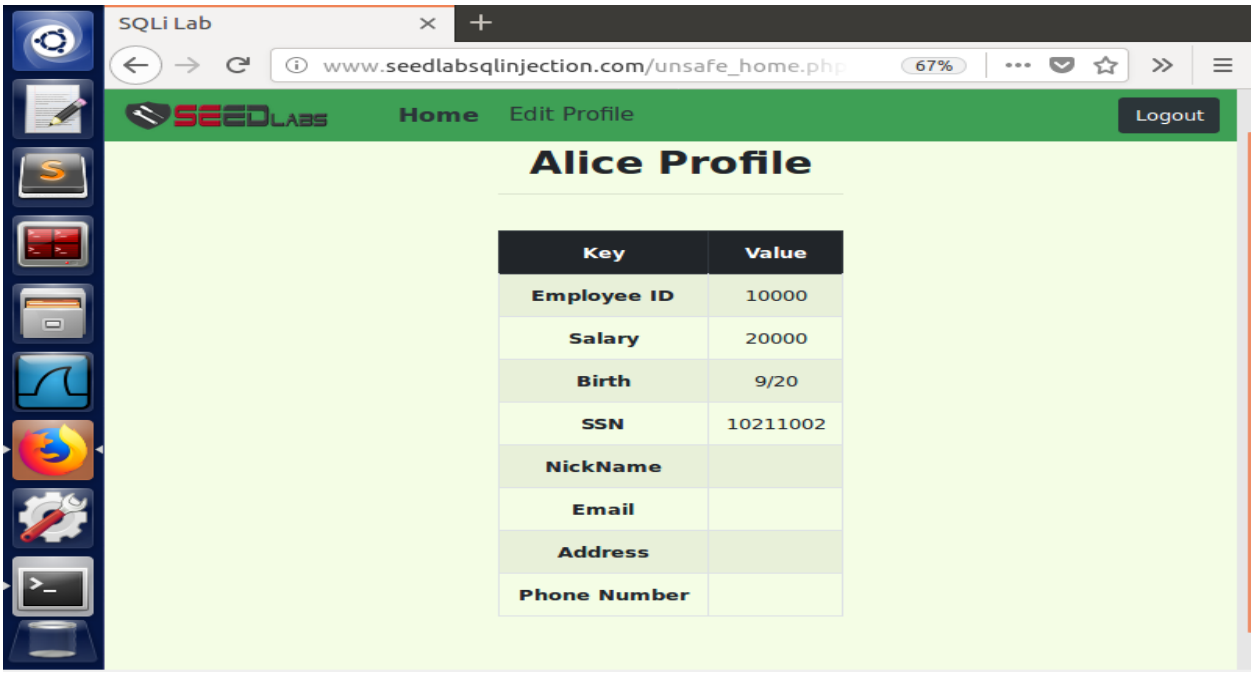
3.3 Task 3: SQL Injection Attack on UPDATE Statement

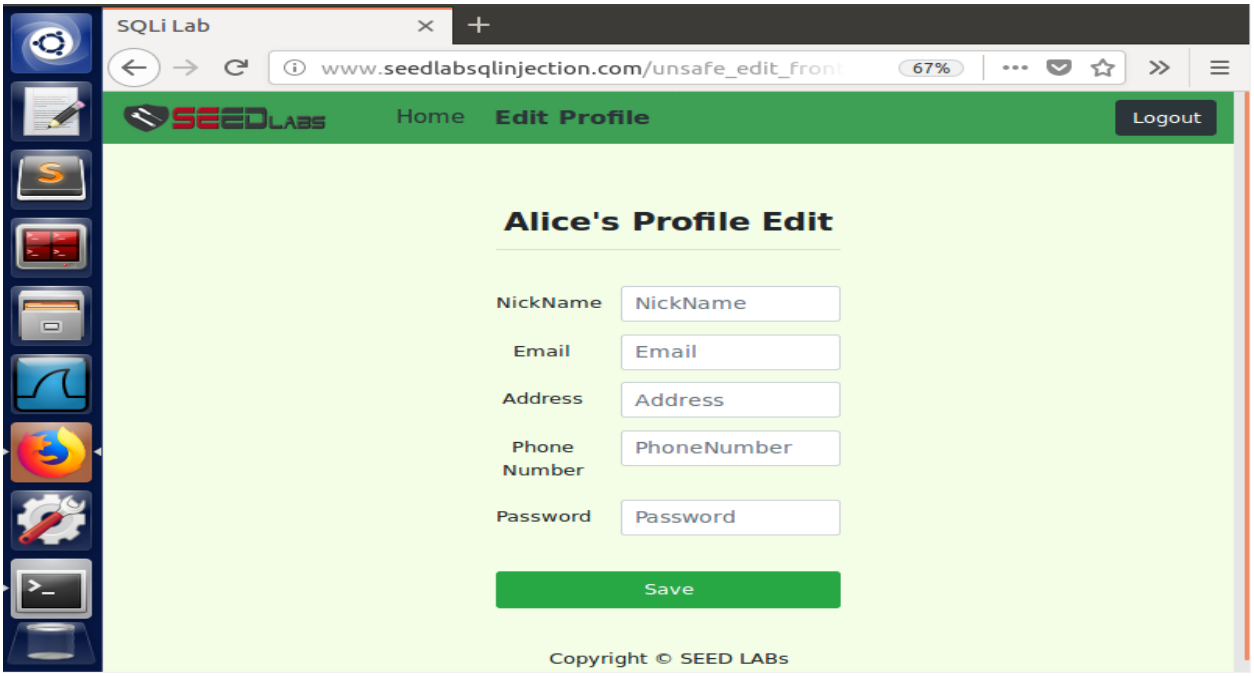
Task 3.1: Modify your own salary.

The purpose of this task is to make the SQL injection attack on UPDATE statement of SQL. Our task as an attacker is to modify the database by using the vulnerable UPDATE statement.

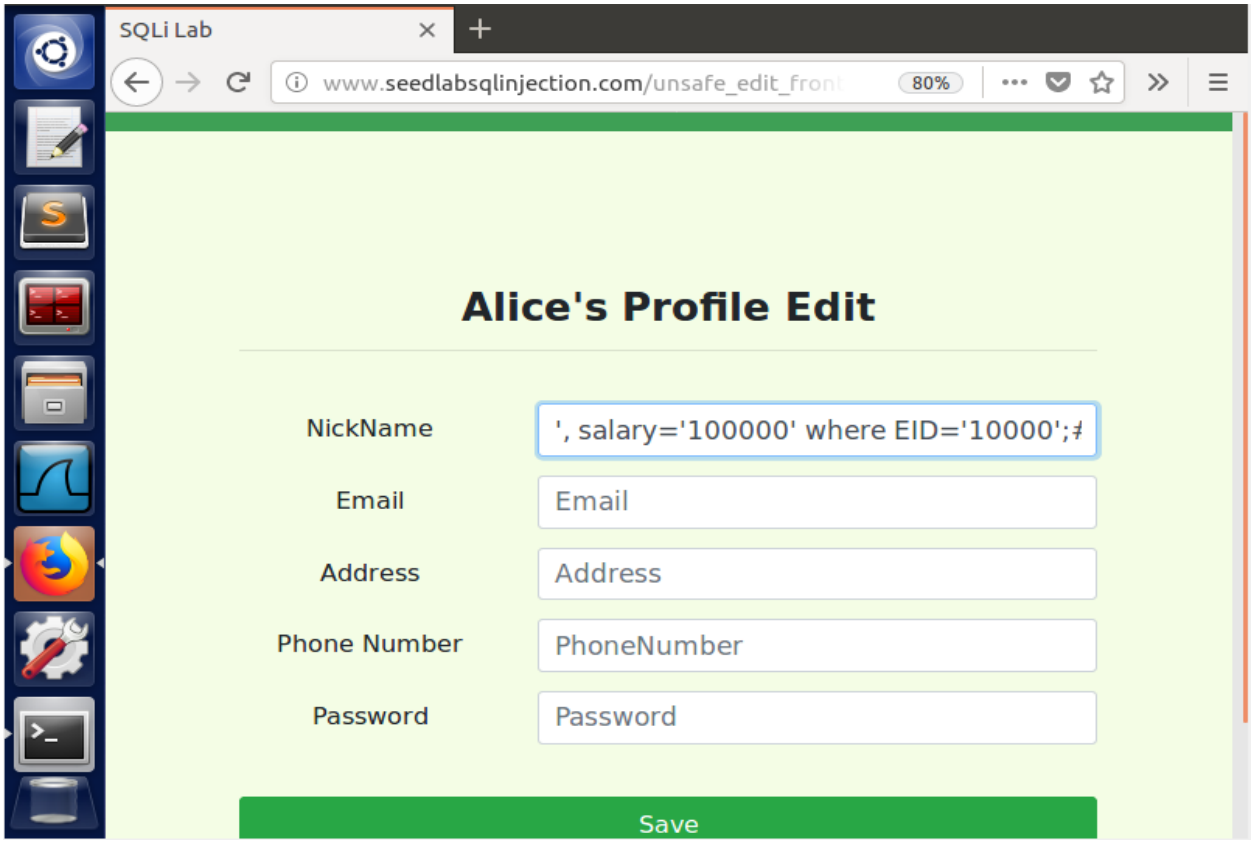


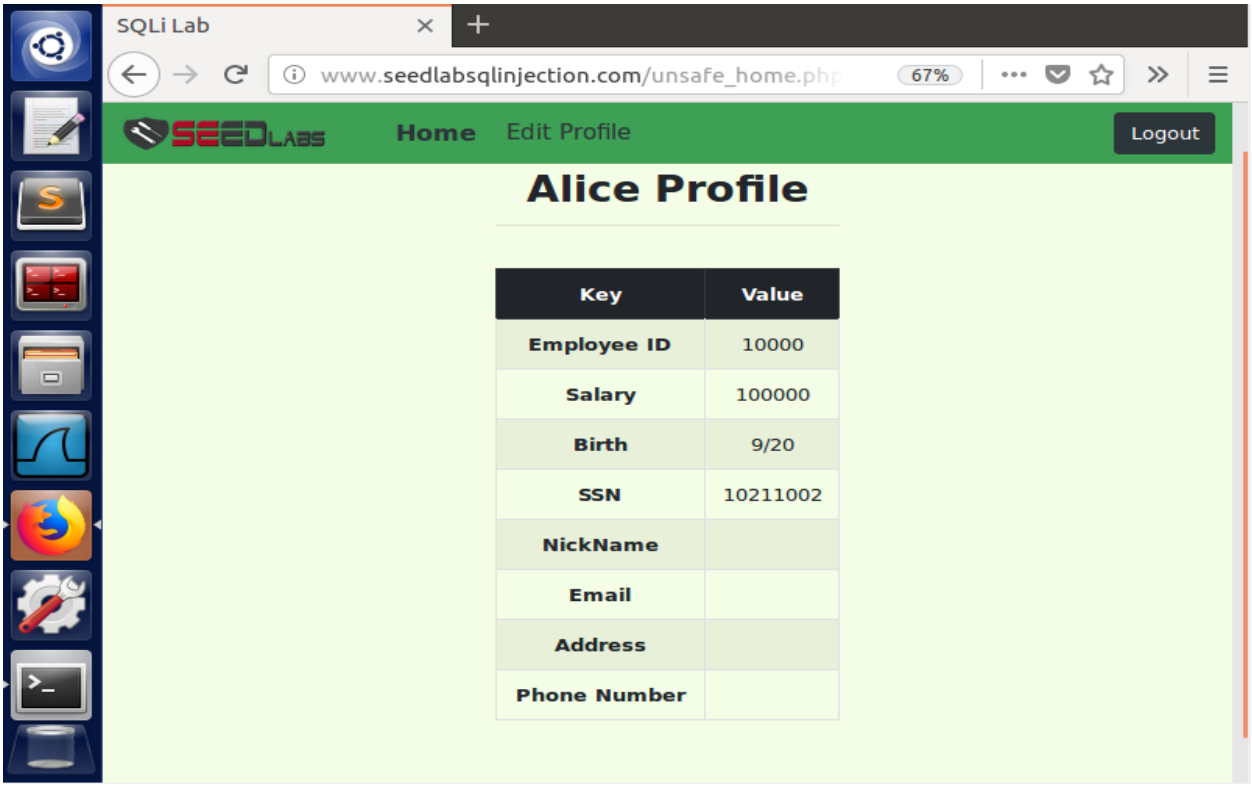
We login as Alice using her credentials. Alice sees that her salary is 20000 and is too low. She wants to modify her salary. But normally salary is not an editable field for the employee.



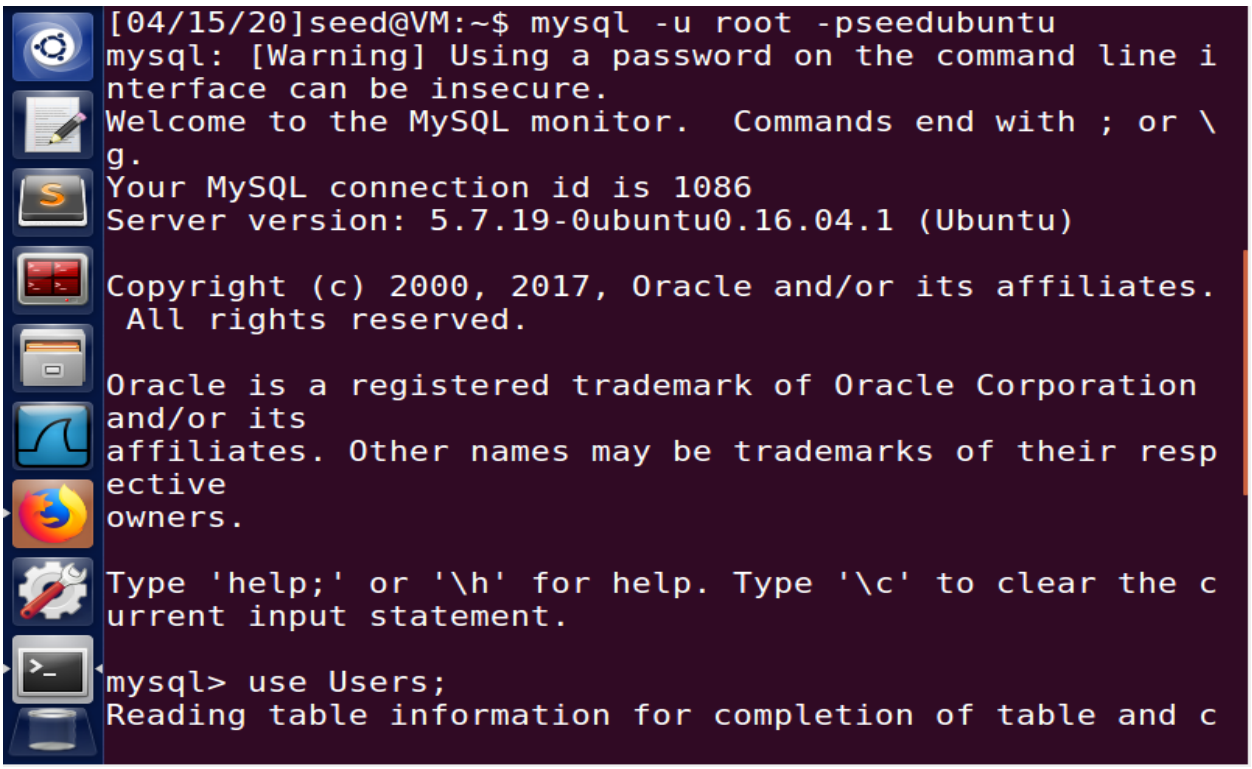


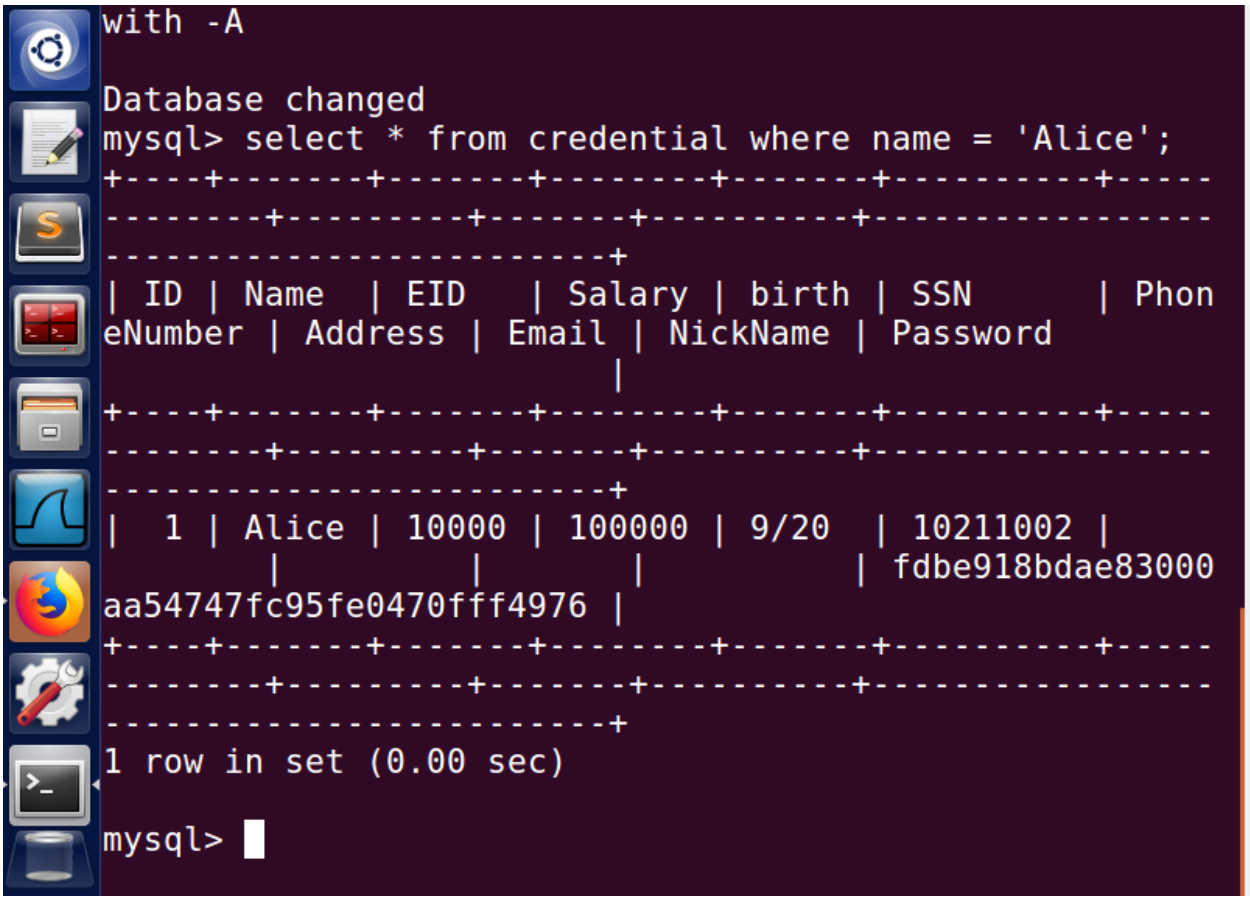
Alice edits her profile and uses the Nickname field to inject her SQL to update her salary. She replaces the $input_nickname variable indirectly in the SQL statement to ‘, salary = ‘100000’ where EID = ‘10000’;# by using the Nickname field in the edit profile page. Here Alice’s EID is 10000.





When Alice saves her changes in the Edit profile, she sees that her Salary has been updated to her desired value. We can also go and check the database in the terminal using the command line to cross verify if Alice's salary has been changed.





The above screenshot confirms that Alice's salary has been changed.

Conclusion: When Alice injects an SQL statement in the Nickname field of the Edit profile page, she is easily able to change the fields that is not editable normally. She injects her new Salary as an update and makes changes to her salary.

SQL before the attack at the backend.

```
$sql="UPDATE credential SET
nickname='$input_nickname',email='$input_email',address='$input_addres
s',Password='$hashed_pwd',Phonenumber='$input_phonenumber' where
ID=$id;";
```

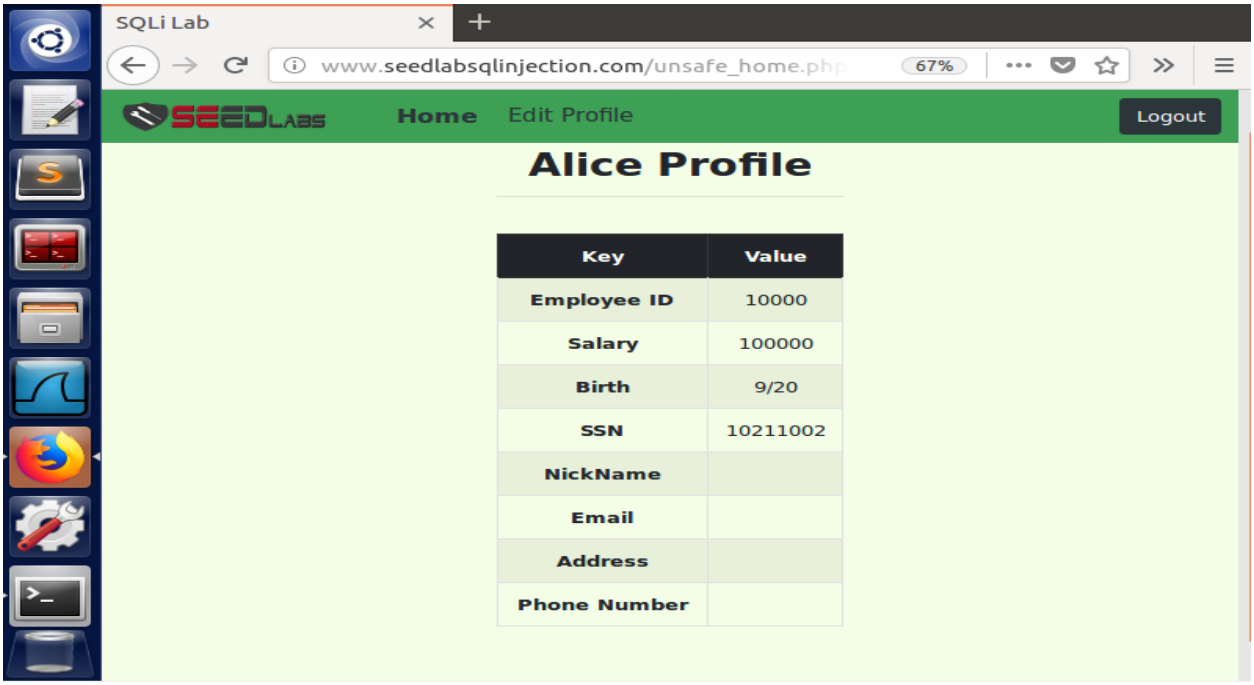
SQL after the attack using the Nickname field of the Edit Profile page.

```
$sql="UPDATE credential SET nickname='', salary = '100000' where EID =
'10000';#',email='$input_email',address='$input_address',Password='$ha
shed_pwd',Phonenumber='$input_phonenumber' where ID=$id;";
```

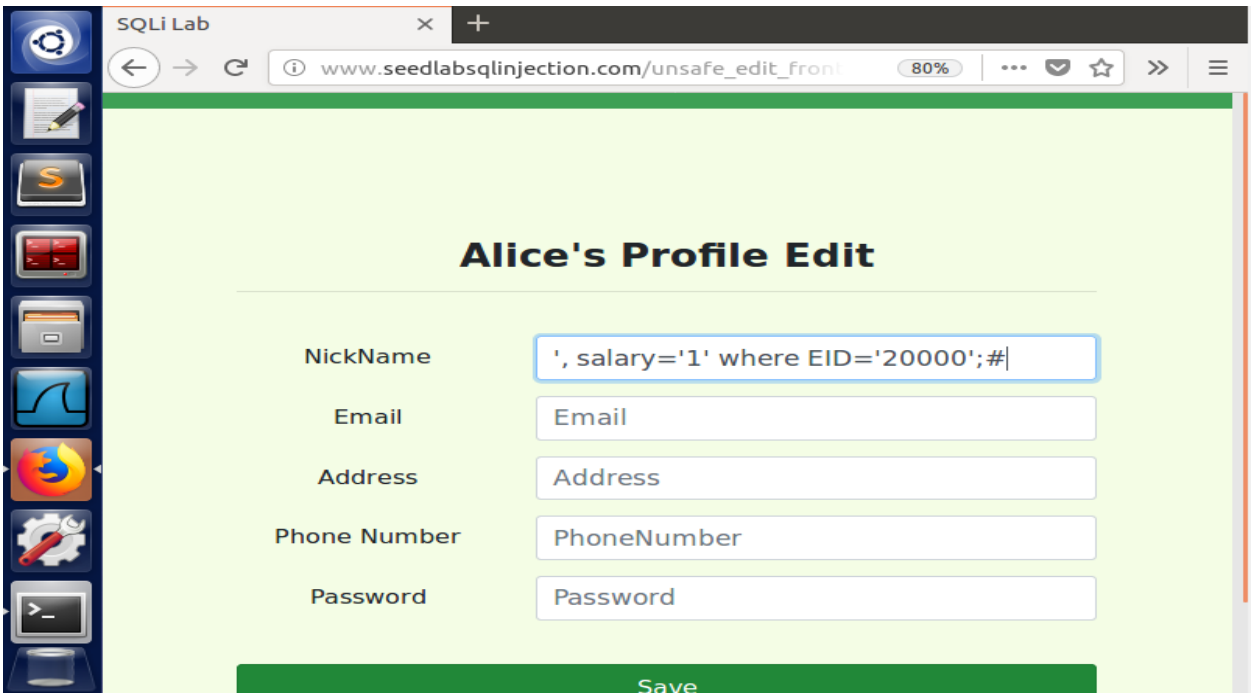
The UPDATE statement is easily replaced with the values we want by just using the Nickname field $input_nickname. The above modified SQL statement changes the meaning of the update totally. The # comments out the rest of the SQL condition specified. This can be avoided using the prepared statement.

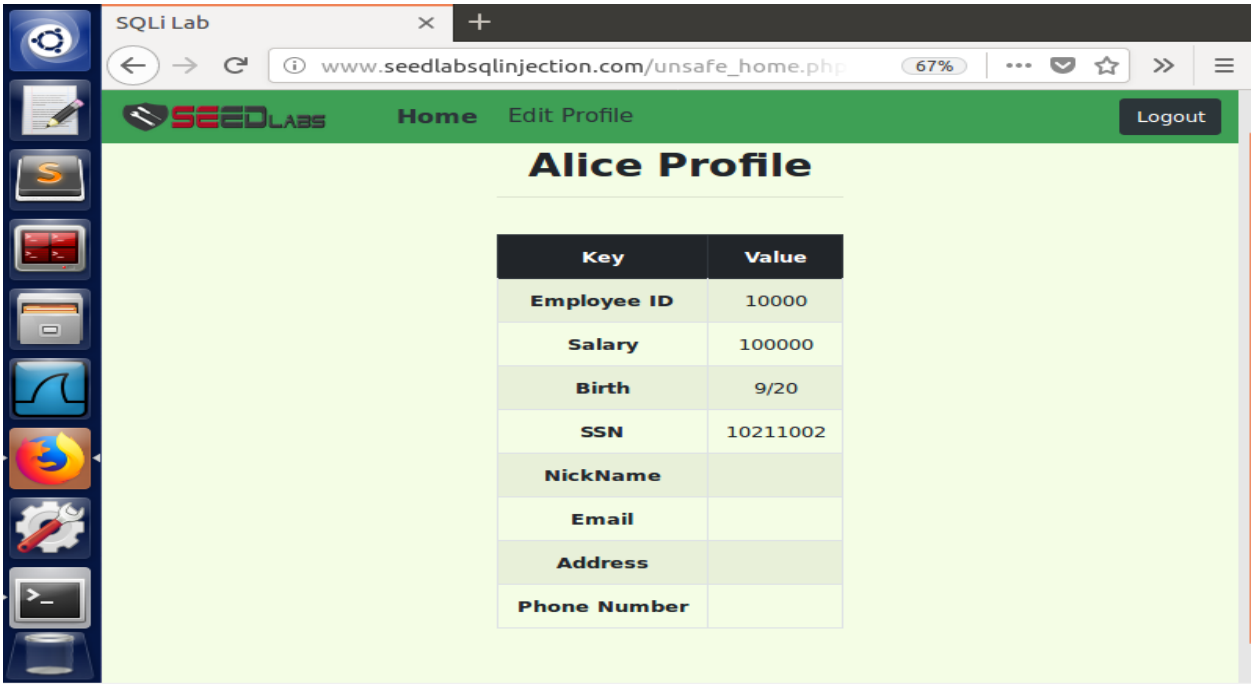
Task 3.2: Modify other people' salary.

The purpose of this task is to make Alice to change her boss, Boby's salary to $1. In order to achieve this we login as Alice.

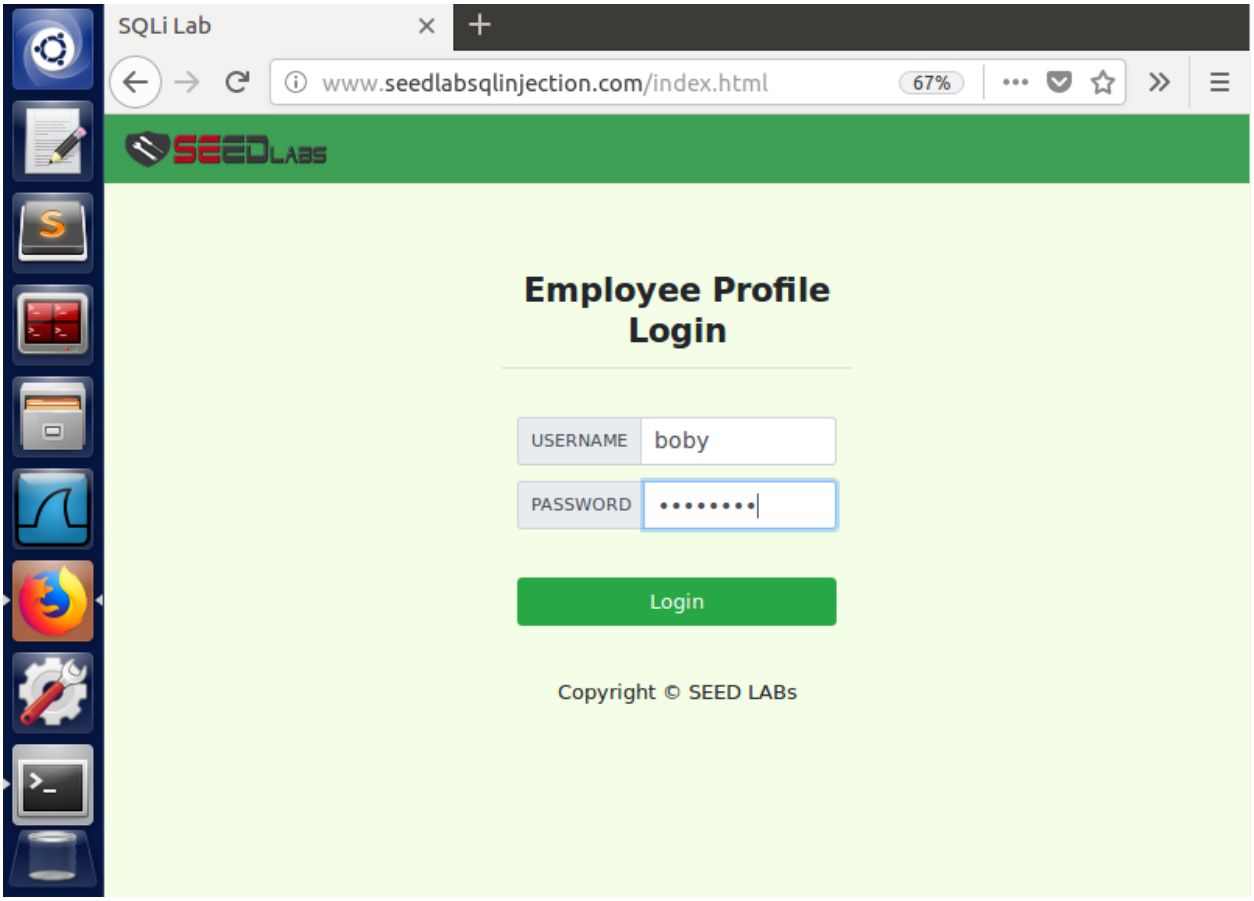


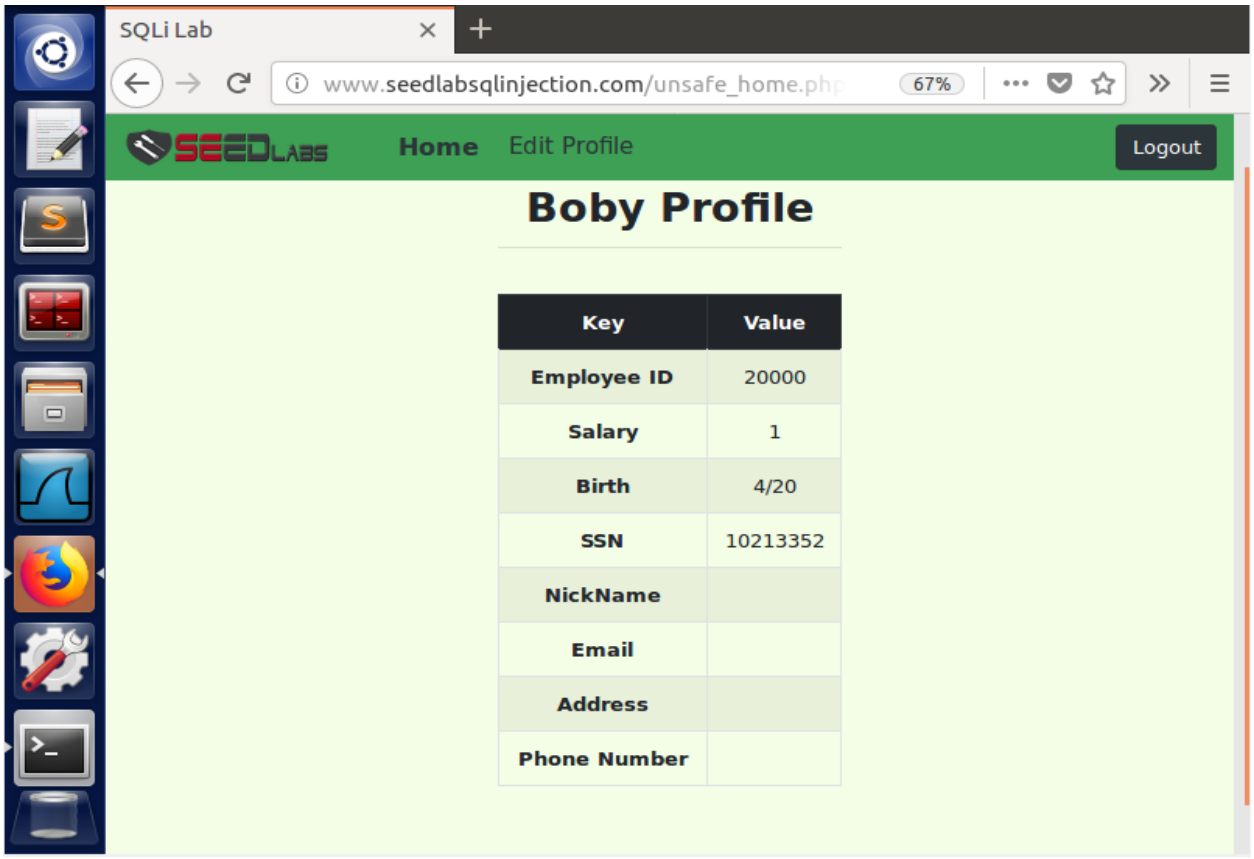
Alice edits her profile and applies the same SQL injection, but this time she uses Boby's EID (20000) to change his salary. She then saves her profile as shown in the screenshot.



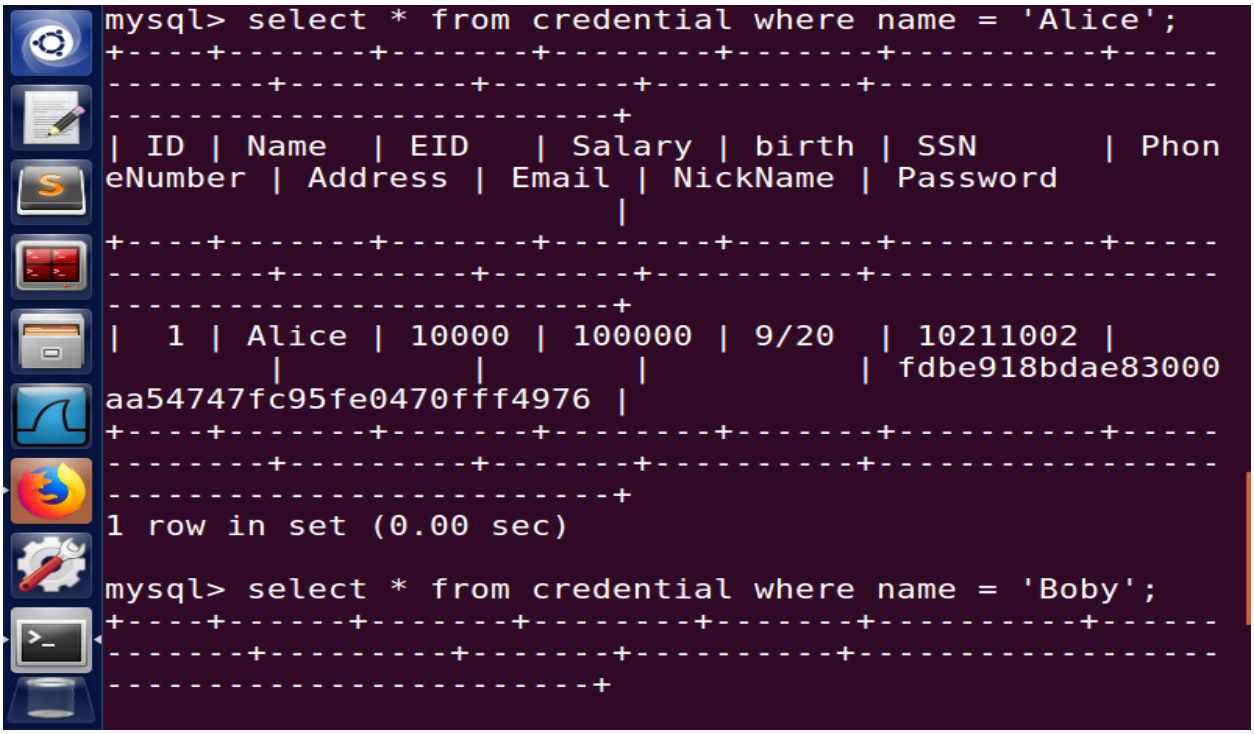


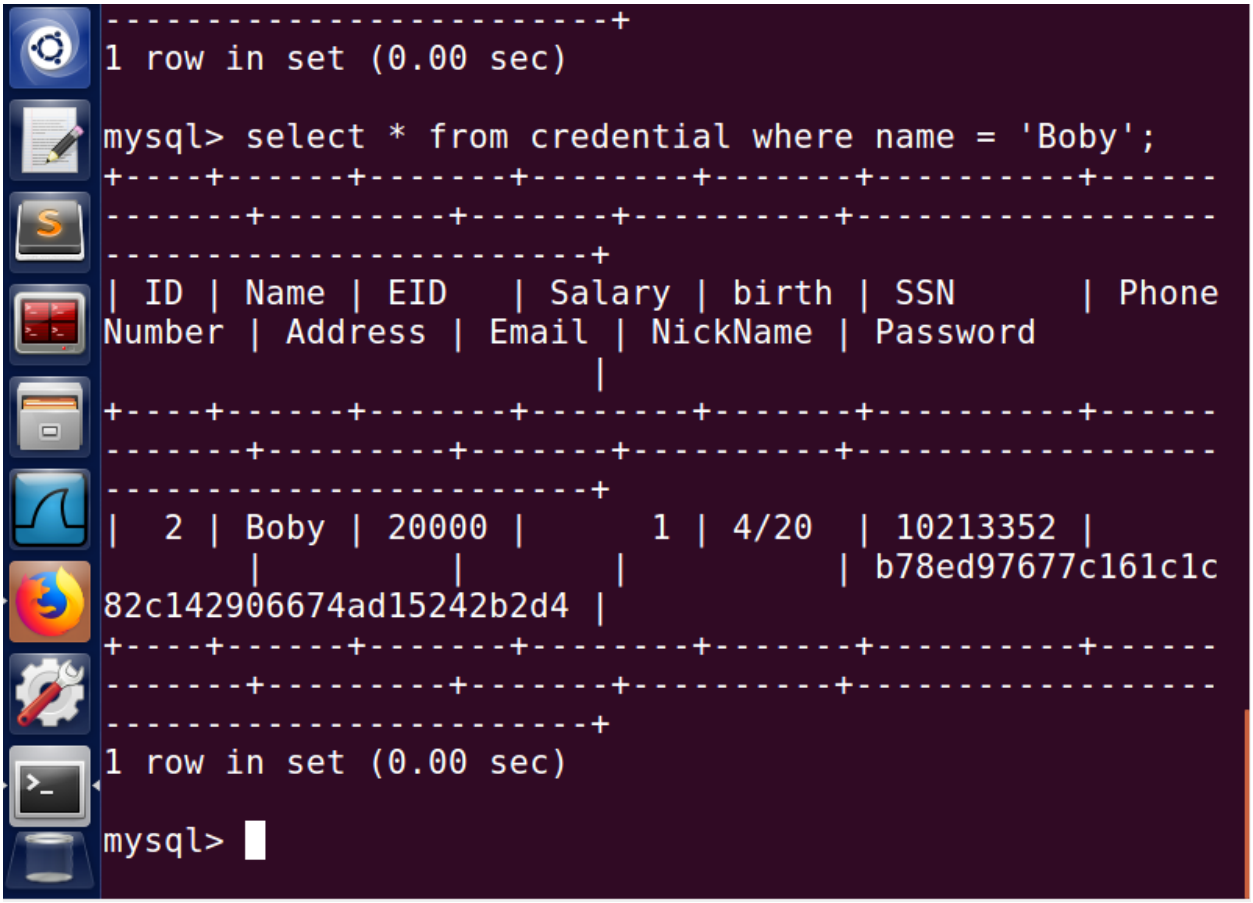
There are no changes to Alice's profile after SQL injection. Boby makes a login to his profile.





Boby sees that his salary has been changed to $1 in his profile. To double check this data, we also check in the terminal command line by using SQL select statements.





Conclusion: When Alice injects an SQL statement in the Nickname field of the Edit profile page, she is easily able to change the fields that is not editable normally and she is able to modify the salary of other people. She injects her boss's new Salary as an update and makes changes to his salary without making any changes to her profile.

SQL before the attack at the backend.

```
$sql="UPDATE credential SET
nickname='$input_nickname',email='$input_email',address='$input_addres
s',Password='$hashed_pwd',Phonenumber='$input_phonenumber' where
ID=$id;";
```

SQL after the attack using the Nickname field of the Edit Profile page.

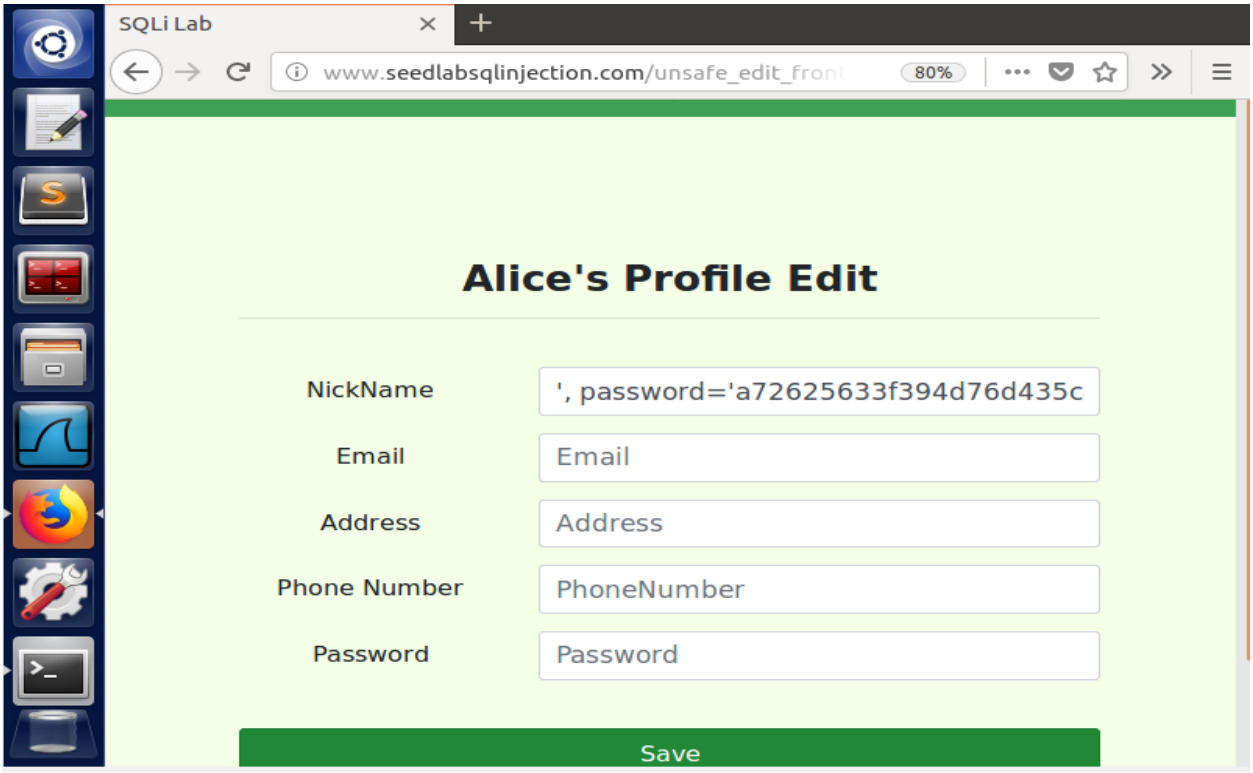
```
$sql="UPDATE credential SET nickname='', salary = '100000' where EID =
'20000';#',email='$input_email',address='$input_address',Password='$ha
shed_pwd',Phonenumber='$input_phonenumber' where ID=$id;";
```

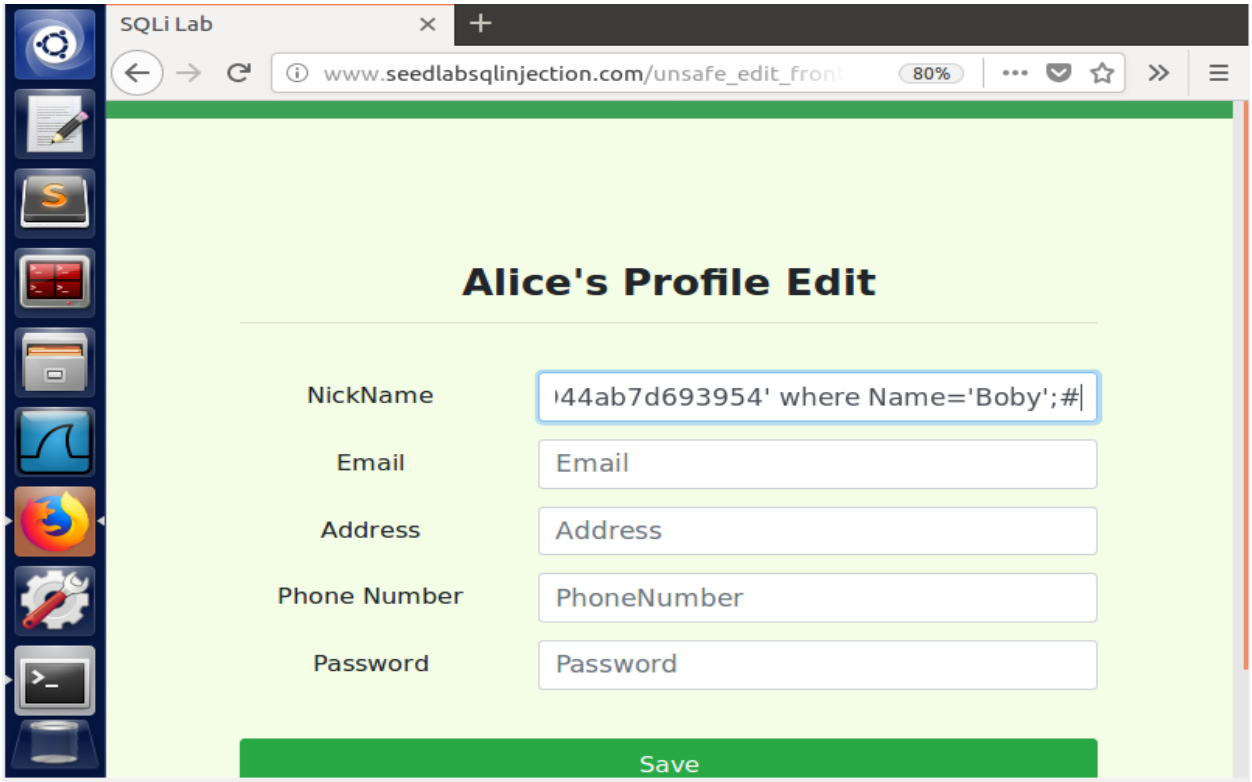
The UPDATE statement is easily replaced with the values we want by just using the Nickname field $input_nickname. The above modified SQL statement changes the meaning of the update totally. The # comments out the rest of the SQL condition specified.

Task 3.3: Modify other people' password.

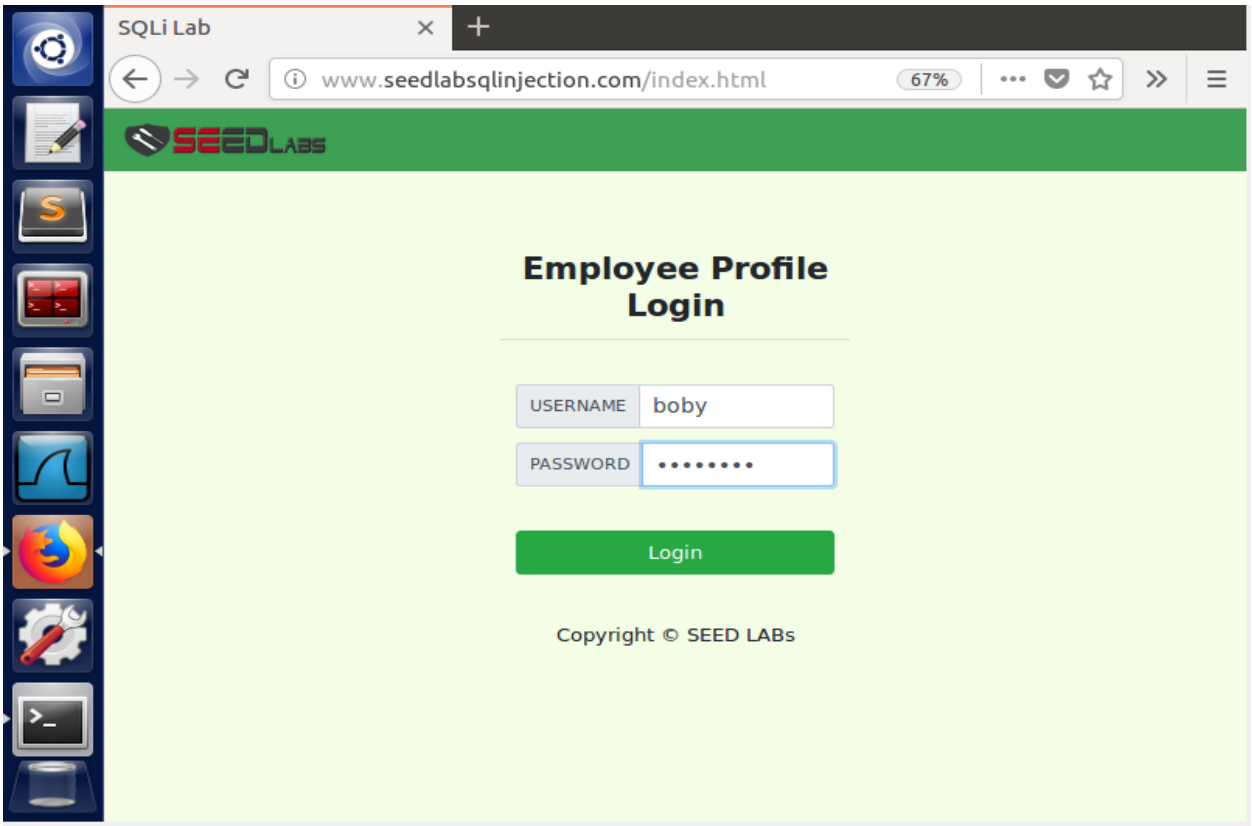
The purpose of this task is to modify Boby's password from Alice's account. In order to achieve this we check Boby;s record in the database Users in the credential table. Boby's password is represented in the form of a hashed value in his record. We cannot reverse a hashed value

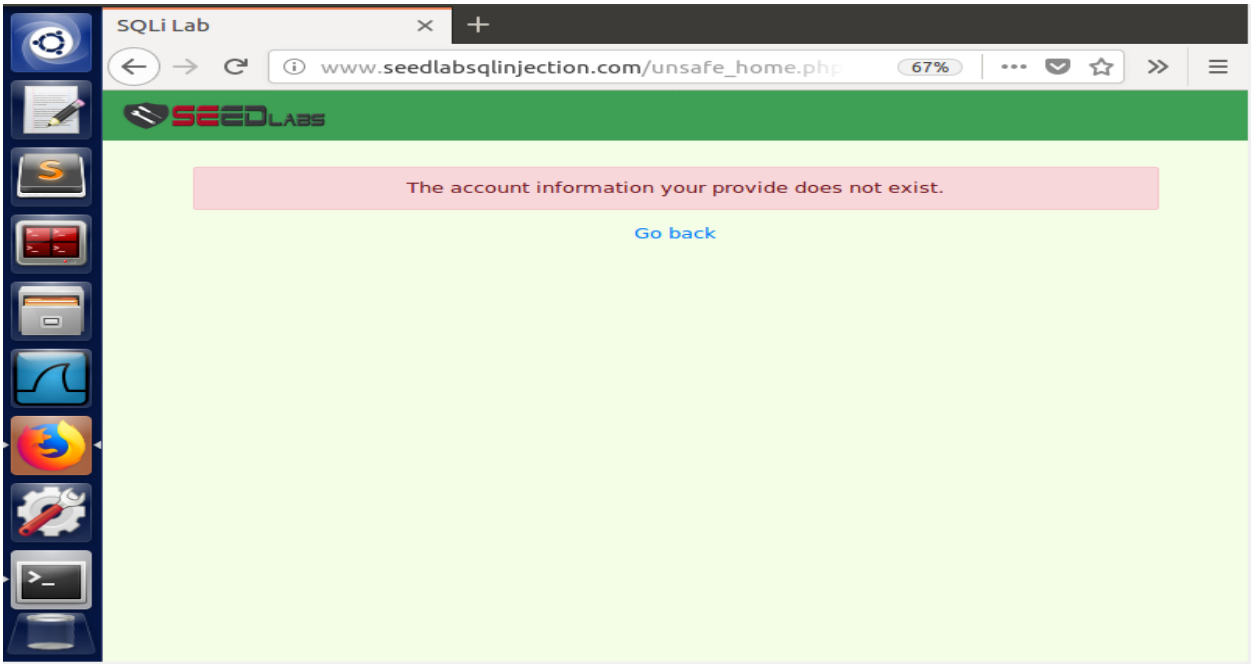
Alice wants to find the hashed value of the password she wants to change Boby's password to. Alice wants to change Boby's password to 'iamchangingpwd'. But she does not know the current password of Boby. In order to achieve this, she finds the hashed value of 'iamchangingpwd' by using openssl sha1.



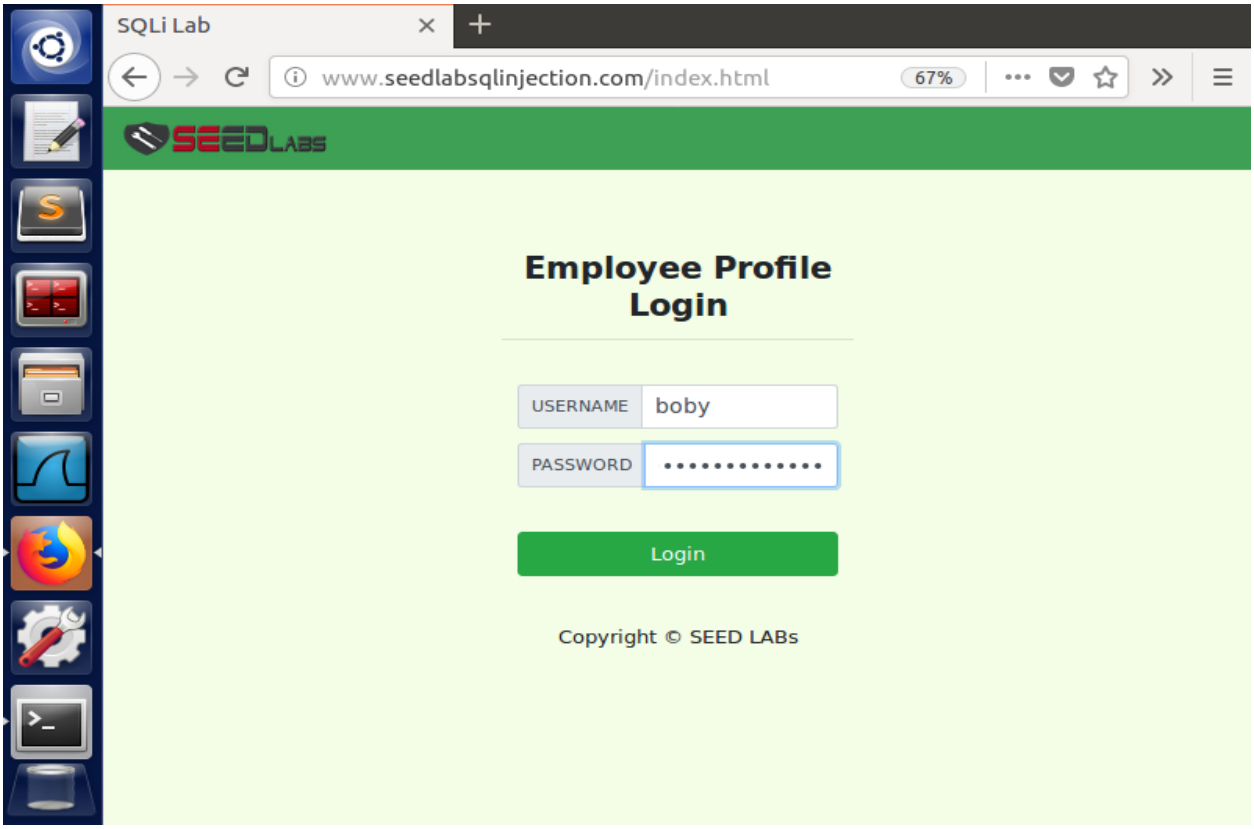


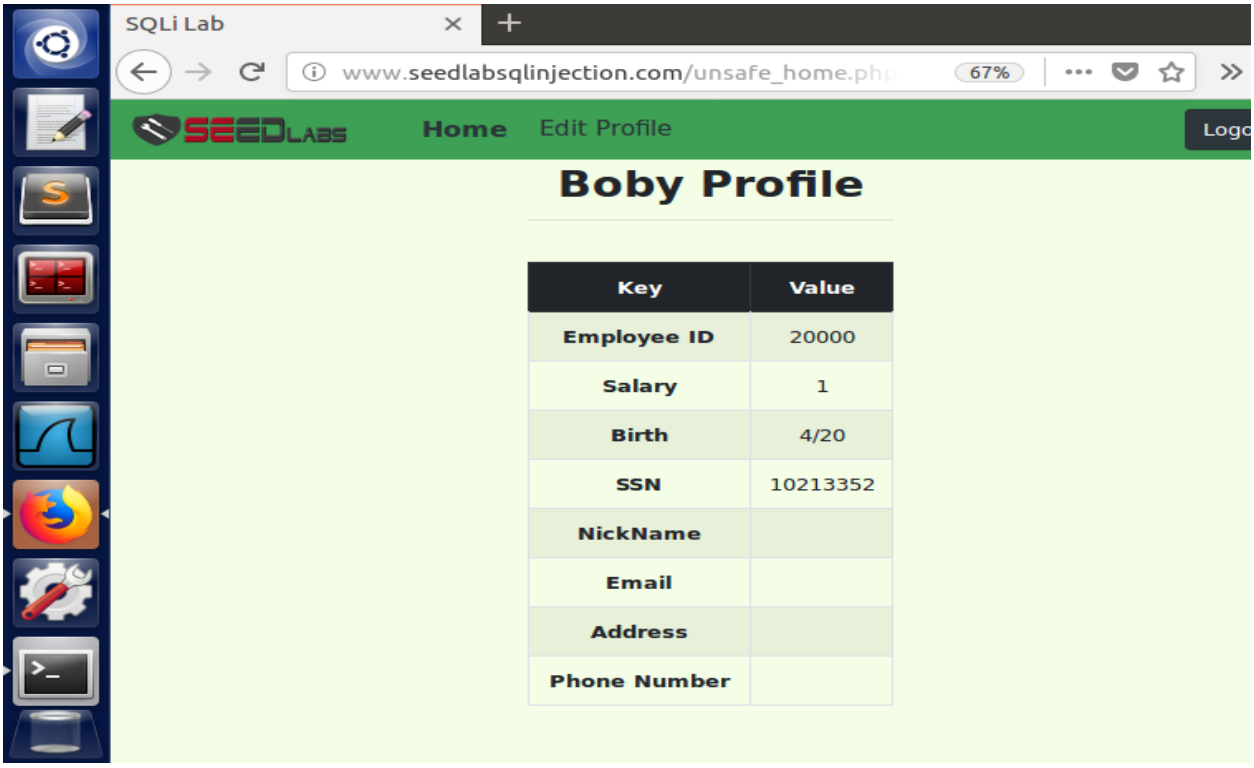
Alice edits her profile and makes use of the Nickname field to exploit the UPDATE statement in the vulnerable application code in the same way as previous tasks. She updates her profile and saves it.



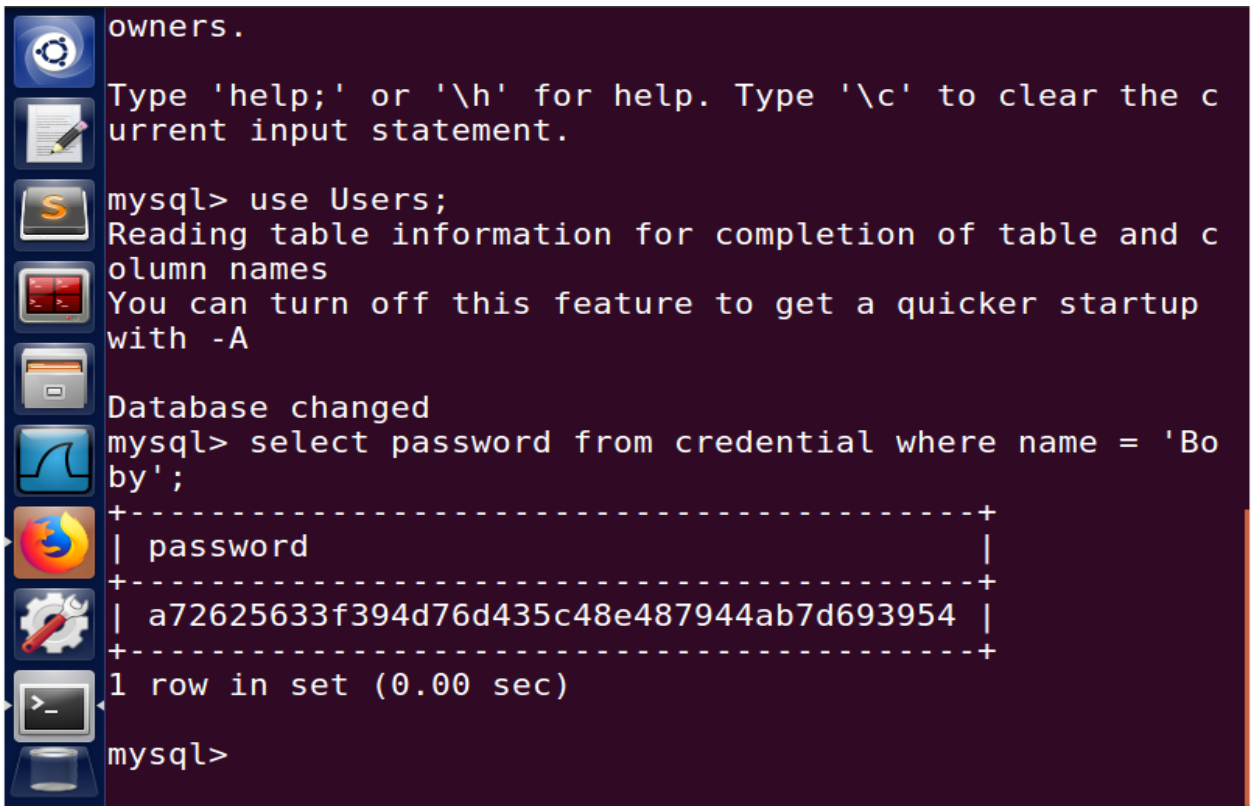


When Boby tries to login to his profile with the old password, it shows an error saying that 'The account information your provide does not exist'. This error occurs when the password entered is wrong.





When Alice tries to login to Boby's profile using his newly changed password, she is able to login to his profile. We check the password using the command line terminal as well. This shows the same hashed password in Boby's record which Alice gave as value in the update field.



Conclusion: When Alice injects an SQL statement in the Nickname field of the Edit profile page, she is easily able to change the fields that is not editable normally and she is able to modify the password of other people. She injects her boss's new hashed password as an update and makes changes to his password without making any changes to her profile and without the knowledge of Boby.

SQL before the attack at the backend.

```
$sql="UPDATE credential SET
nickname='$input_nickname',email='$input_email',address='$input_addres
s',Password='$hashed_pwd',Phonenumber='$input_phonenumber' where
ID=$id;";
```

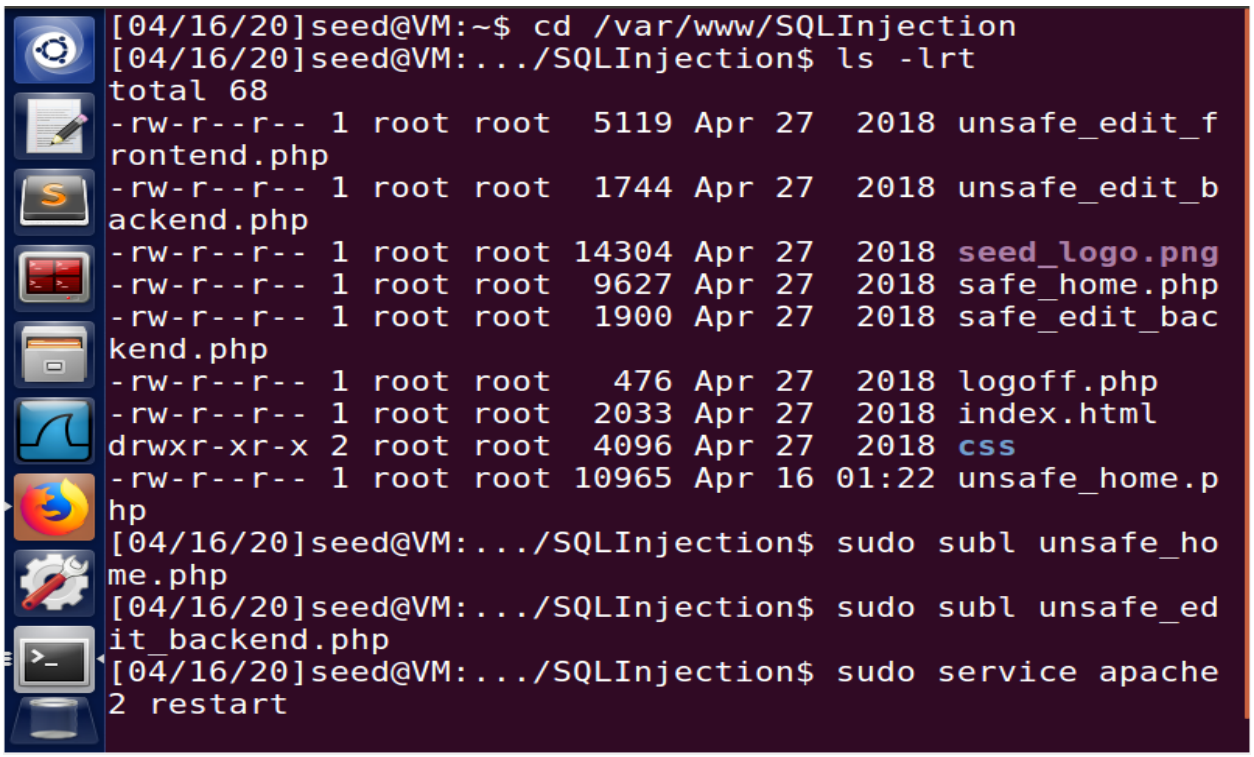
SQL after the attack using the Nickname field of the Edit Profile page.

```
$sql="UPDATE credential SET nickname='', password =
'a72625633f394d76d435c48e487944ab7d693954' where Name =
'Boby';#',email='$input_email',address='$input_address',Password='$has
hed_pwd',Phonenumber='$input_phonenumber' where ID=$id;";
```

The UPDATE statement is easily replaced with the values we want by just using the Nickname field $input_nickname. The above modified SQL statement changes the meaning of the update totally. The # comments out the rest of the SQL condition specified.

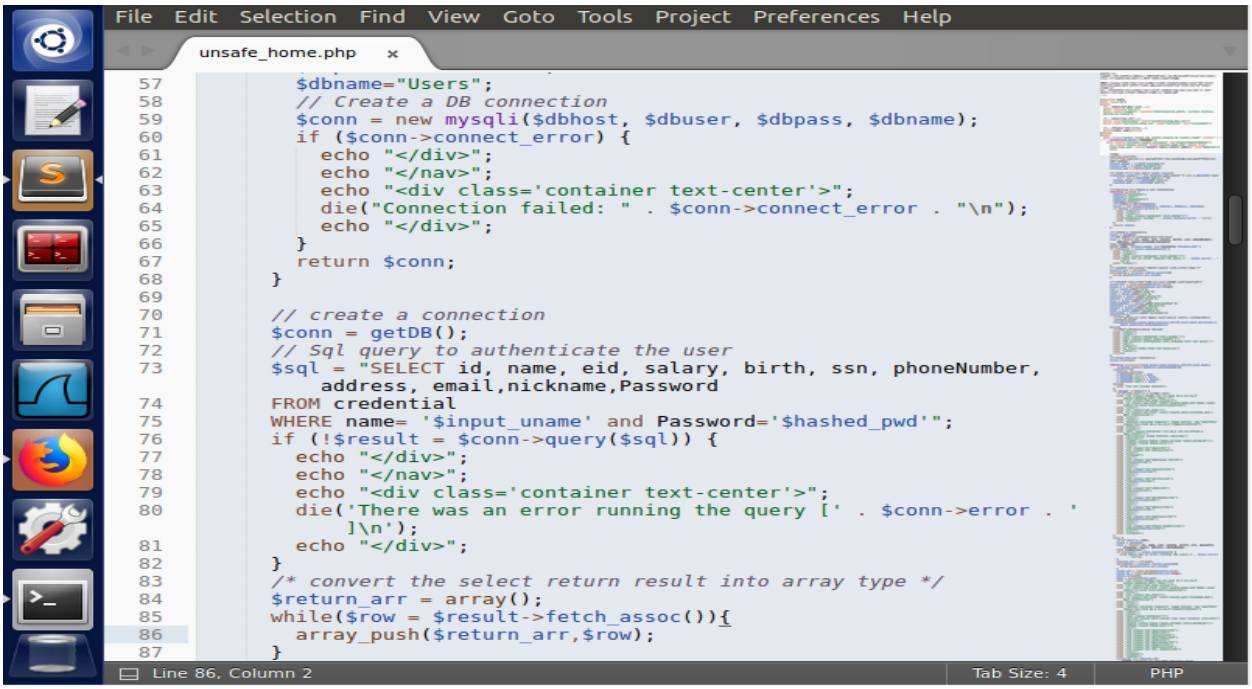
3.4 Task 4: Countermeasure — Prepared Statement

The purpose of this task is to use the prepared statement mechanism to fix the SQL injection vulnerabilities exploited in the previous tasks and perform the same attacks as the previous tasks.

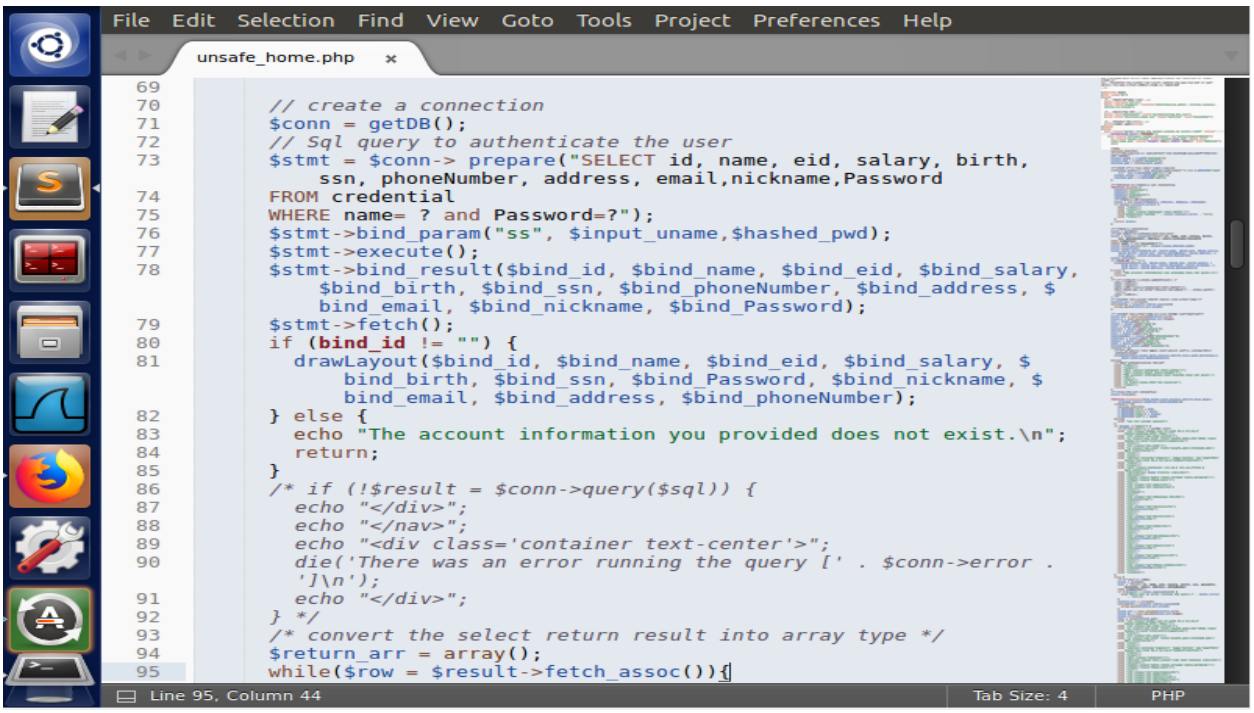


In order to implement the countermeasure to prevent SQL injection attack, we have to include prepared statements in the SQL SELECT and UPDATE queries. To do this, we update the unsafe_home.php and unsafe_edit_backend.php files as shown in the screenshots.

Vulnerable code - unsafe_home.php before implementing countermeasure: This file contains SELECT statemnets

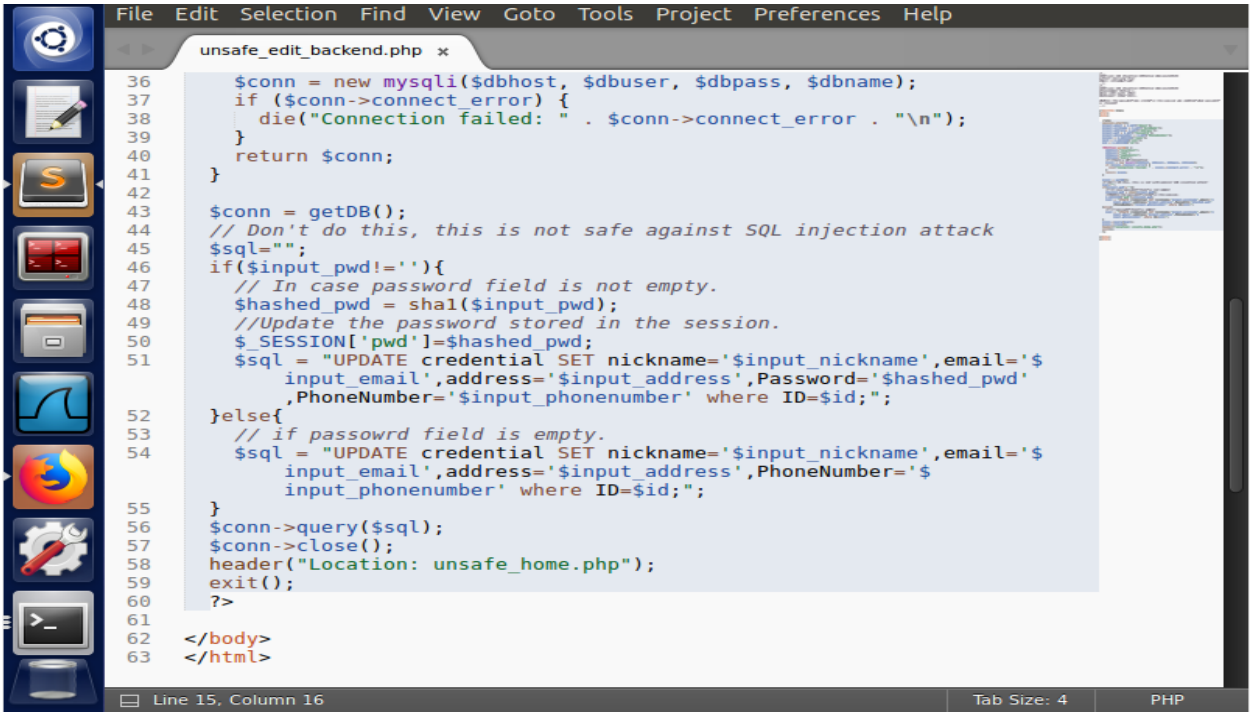


unsafe_home.php after implementing countermeasure:

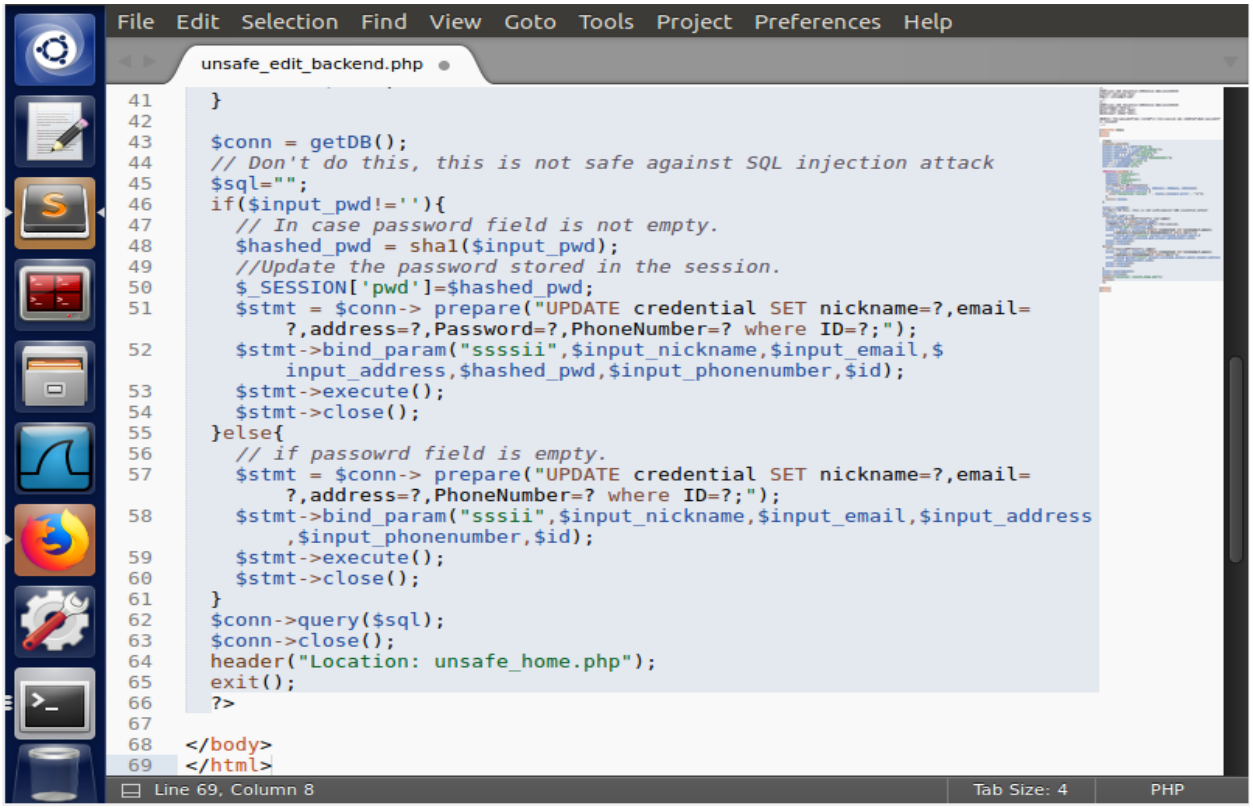


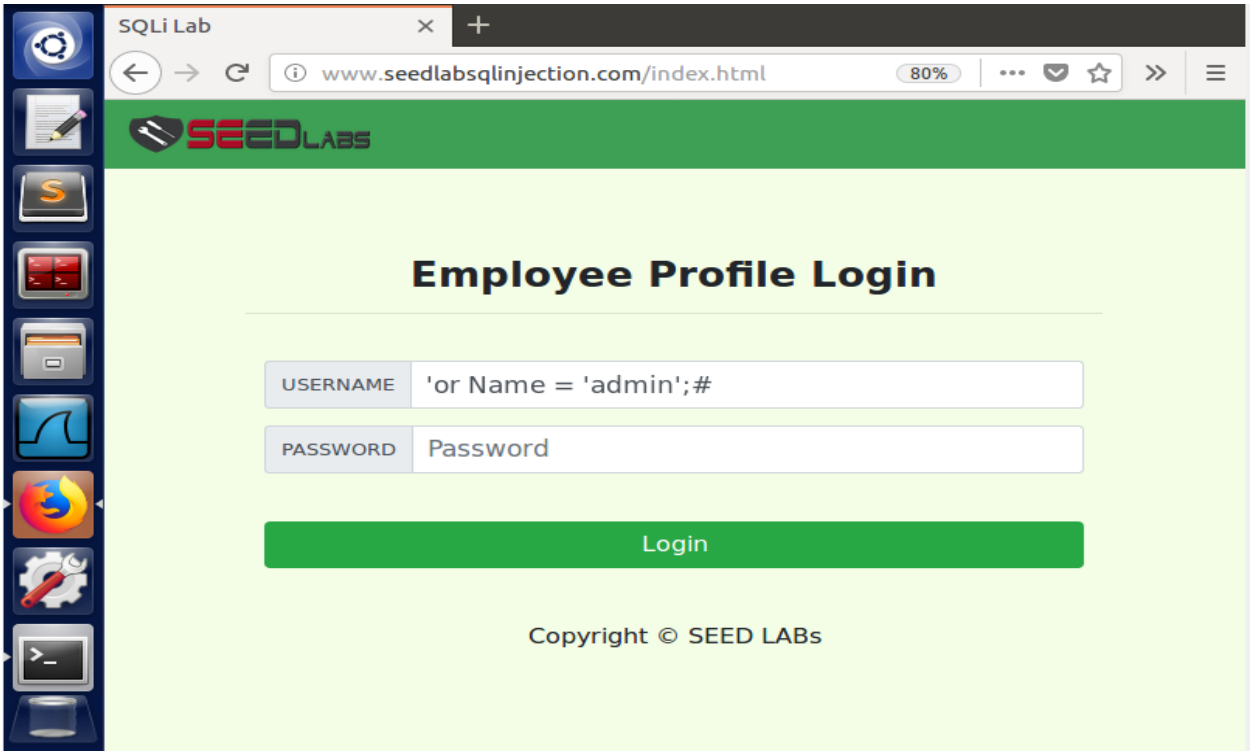
Vulnerable code - unsafe_edit_backend.php before implementing countermeasure:

This file contains UPDATE statements

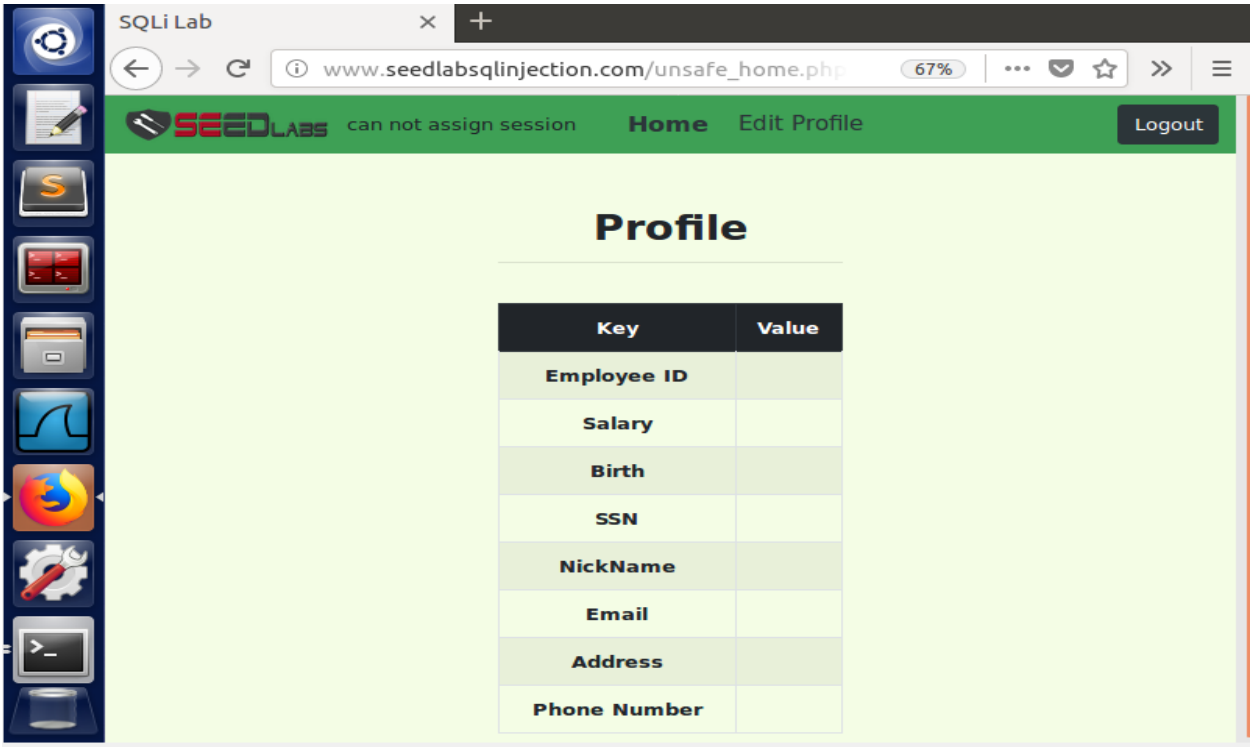


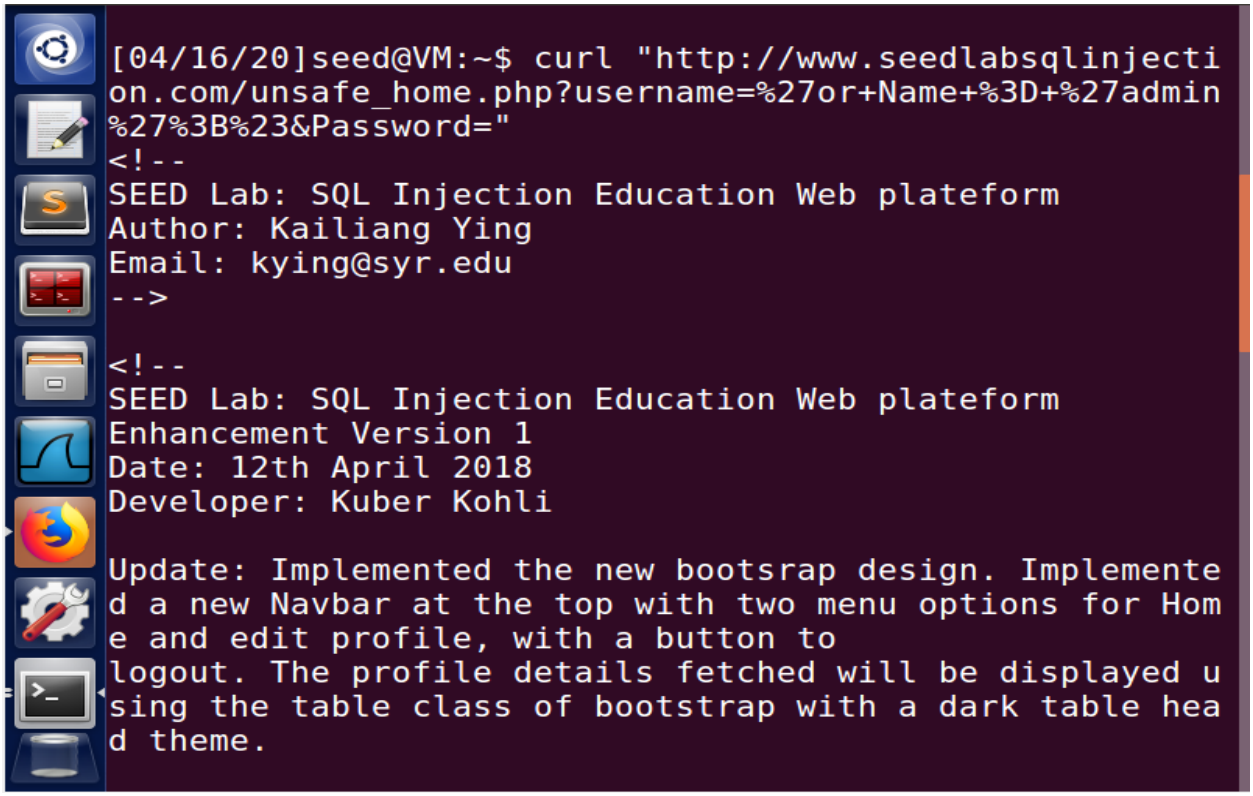
unsafe_edit_backend.php after implementing countermeasure:



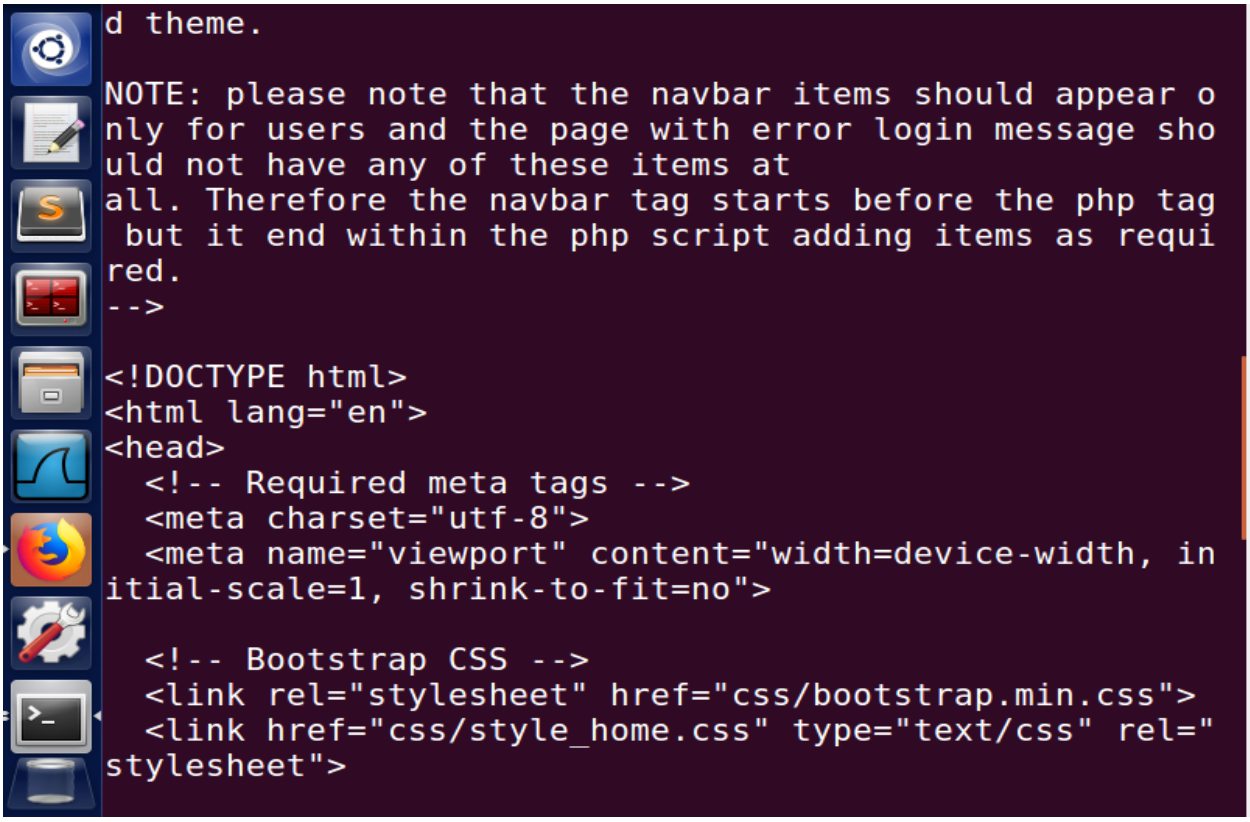


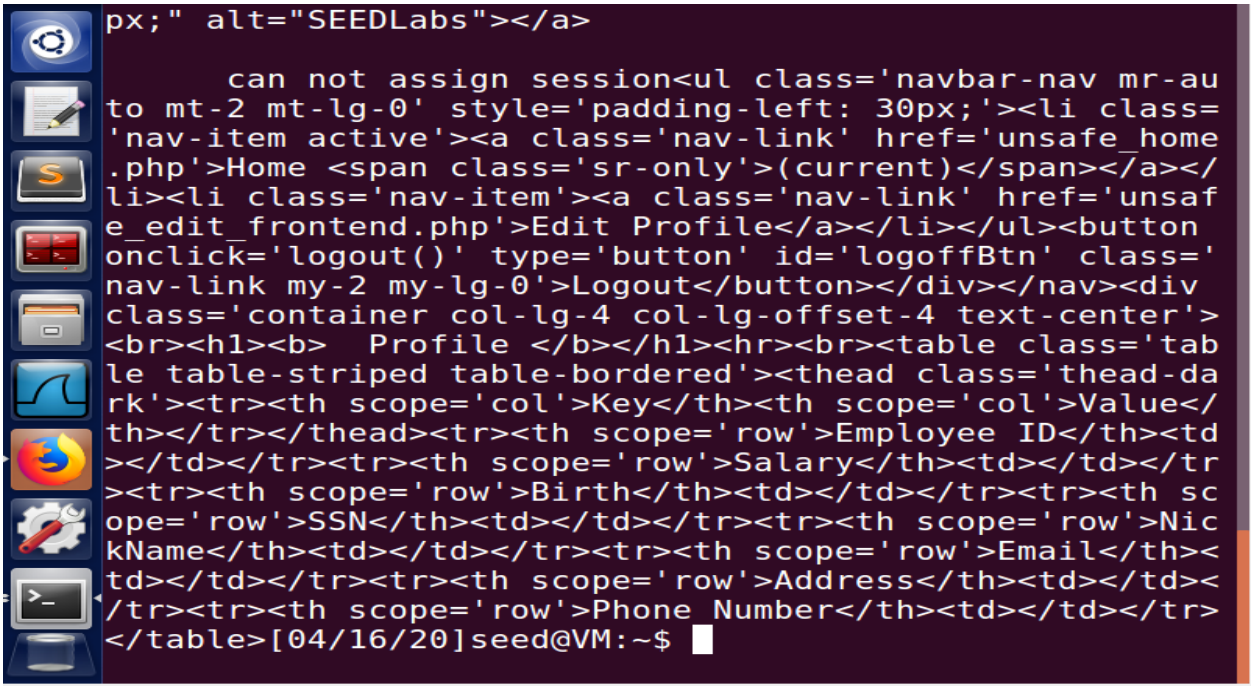
We perform the previous task 2.1 again. We try to login to the application as admin without using any login credentials and we see a blank page getting displayed. This is because of the countermeasure we have enabled.



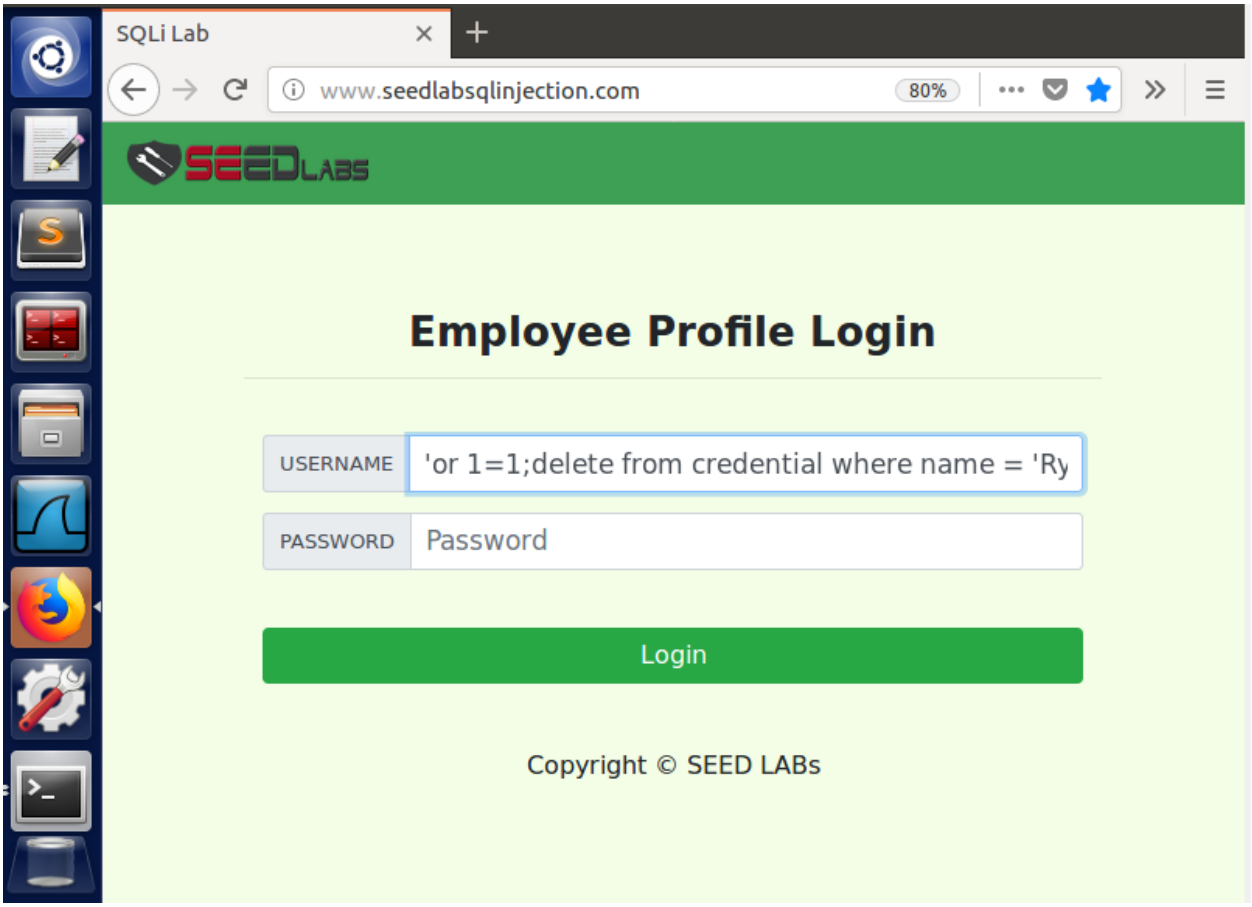


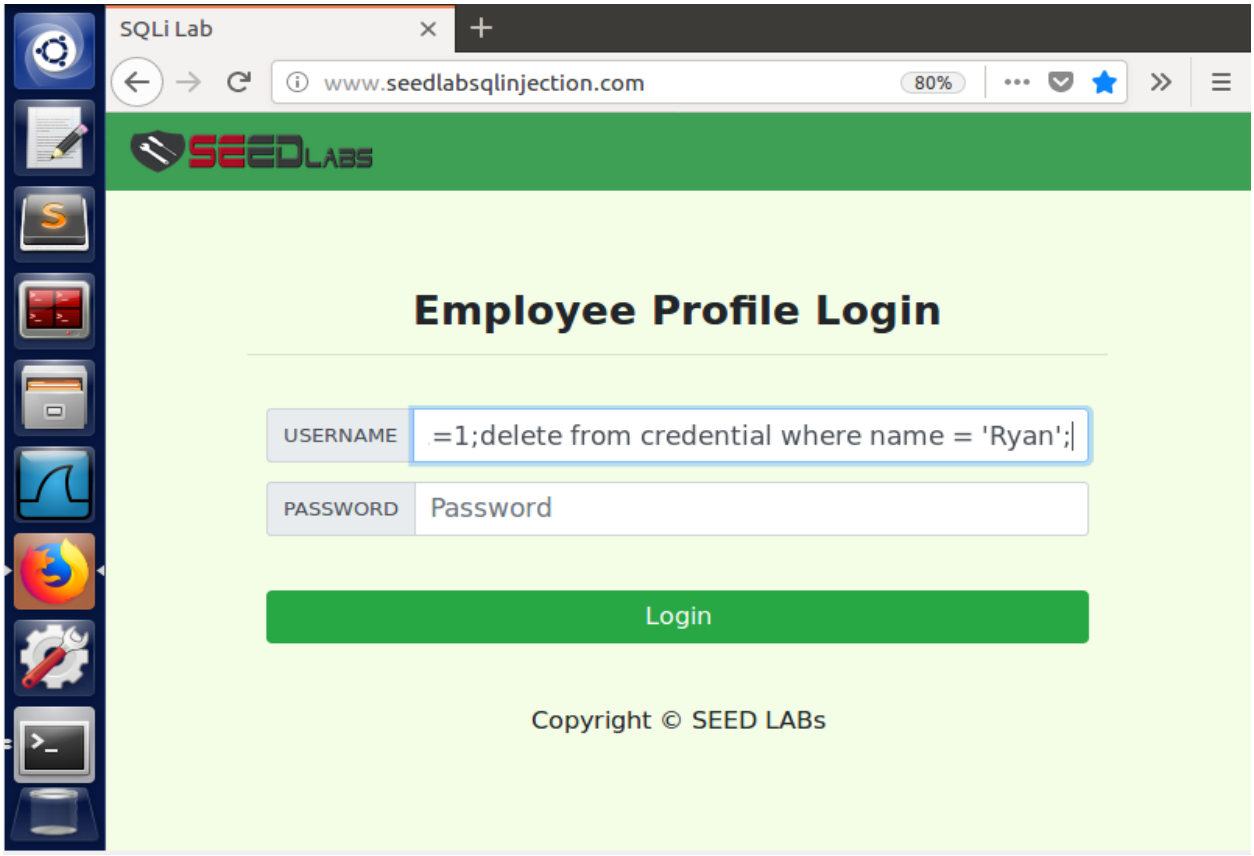
We try this attack using the command line terminal using curl command. We see that we are not able to access the data of the application or login as admin.



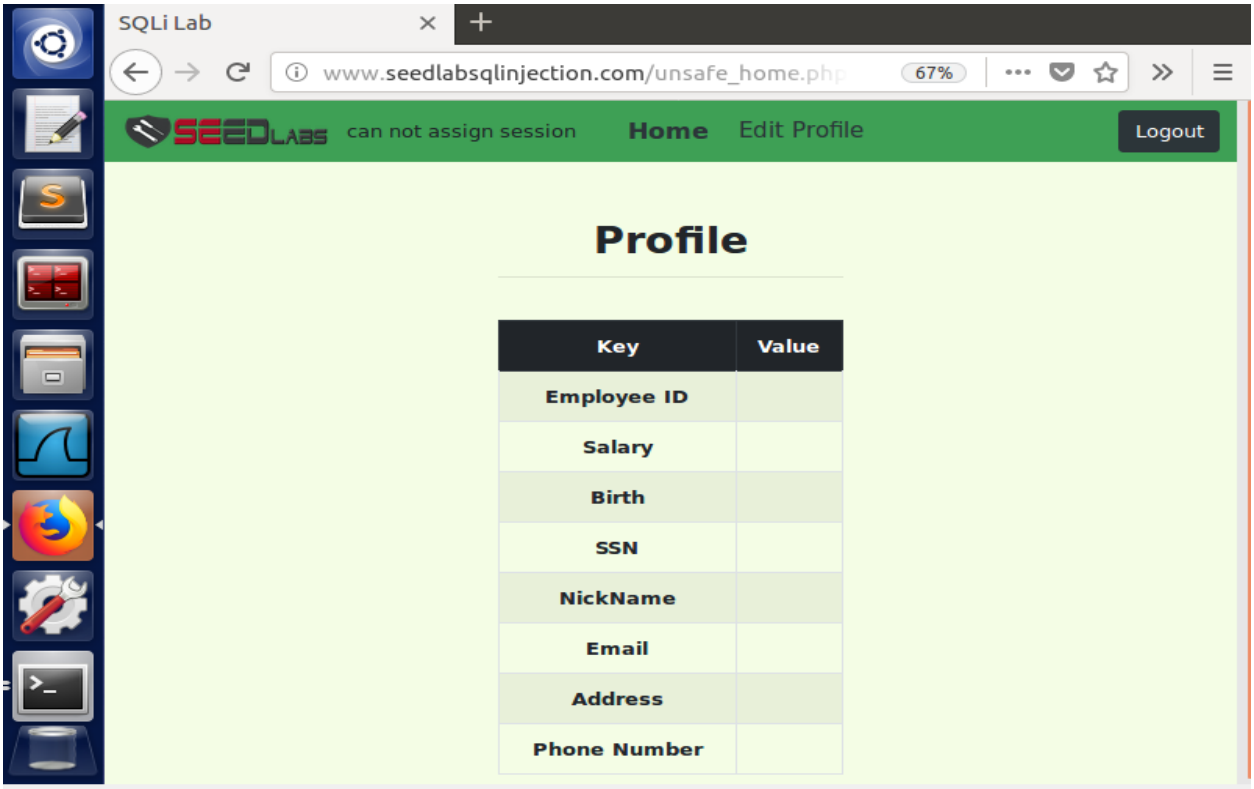


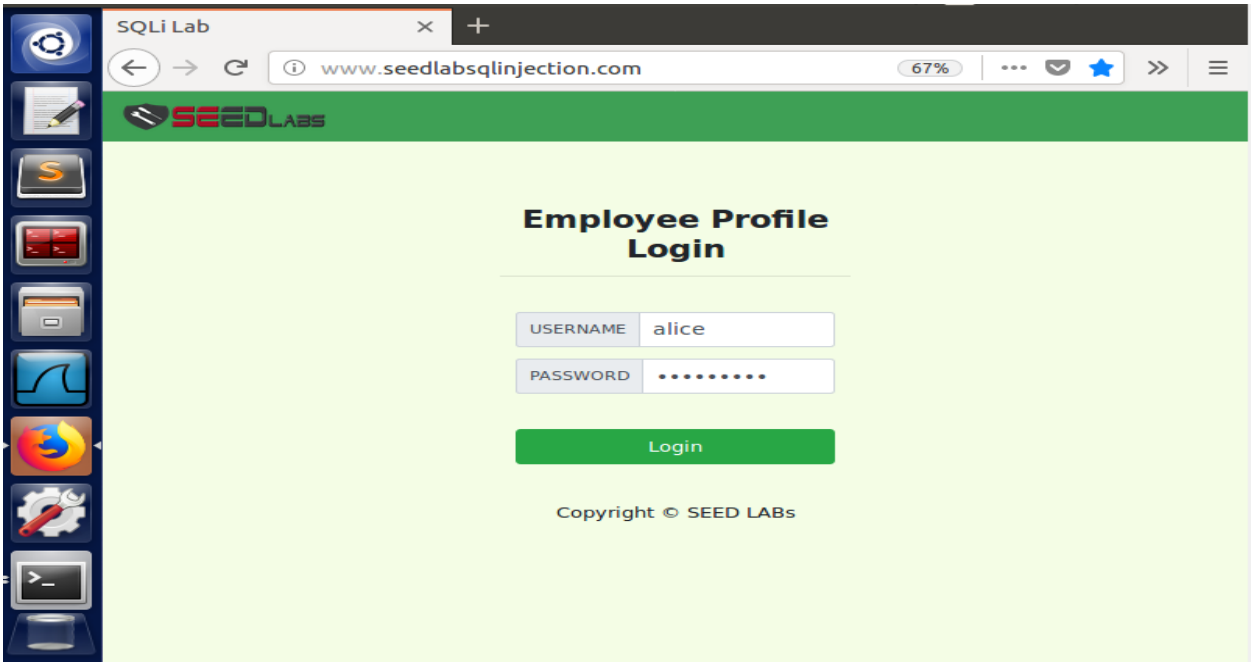
Then we try to delete the Ryan's record from the credential table.



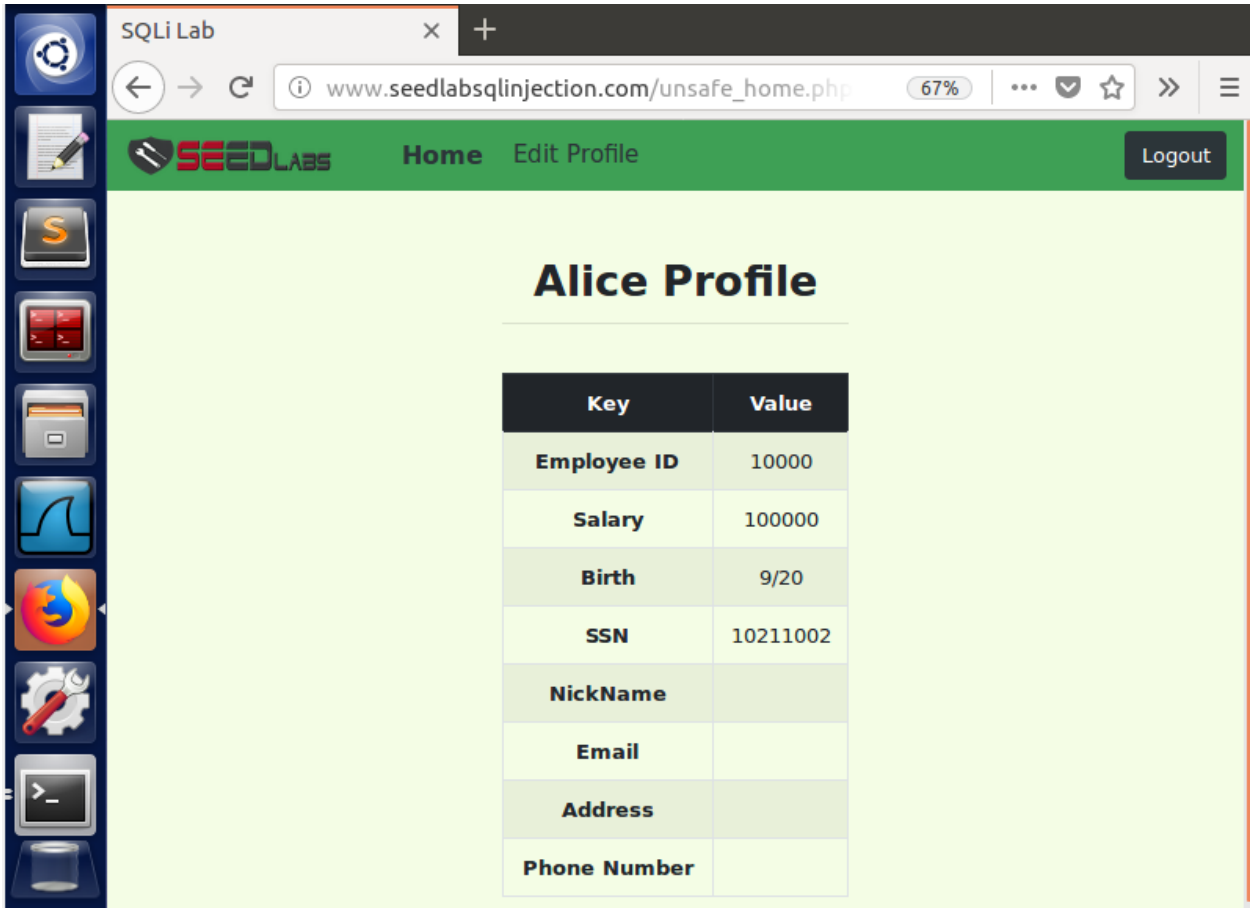


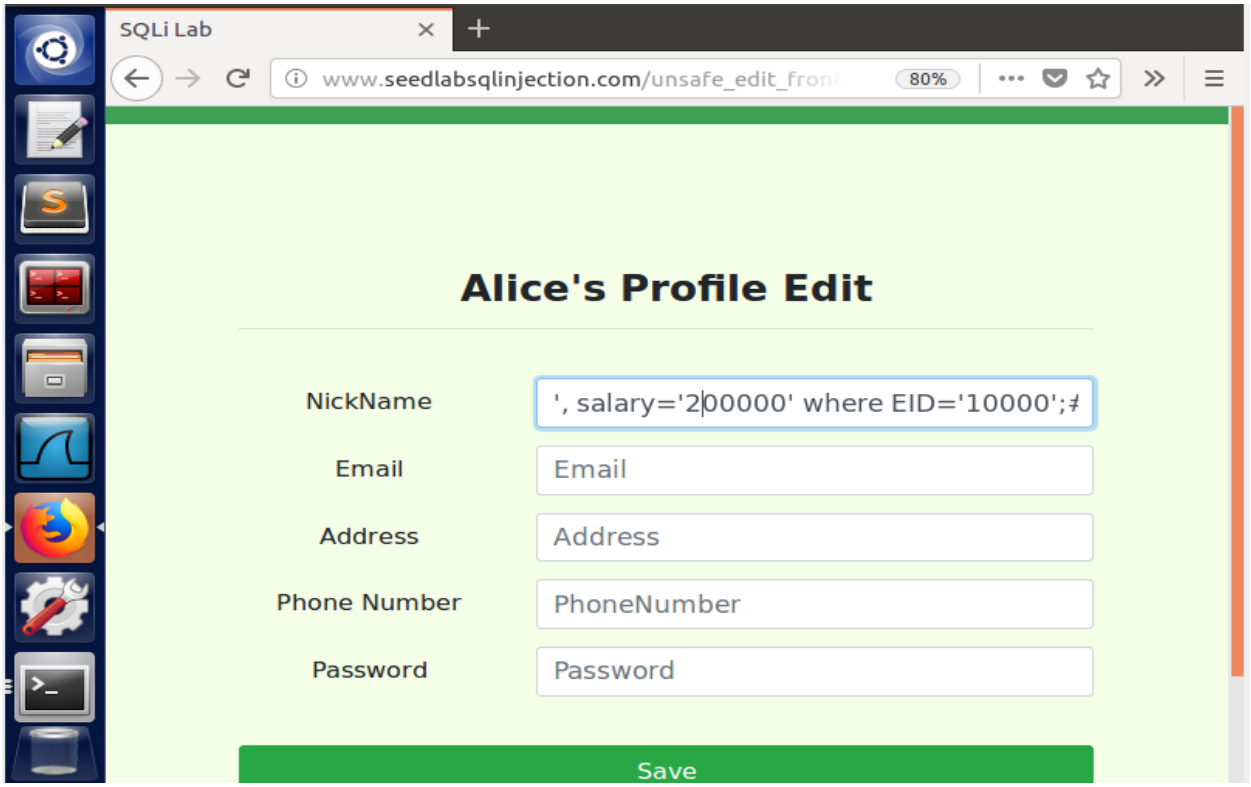
This task also fails because the countermeasure is implemented. The delete operation does not happen.



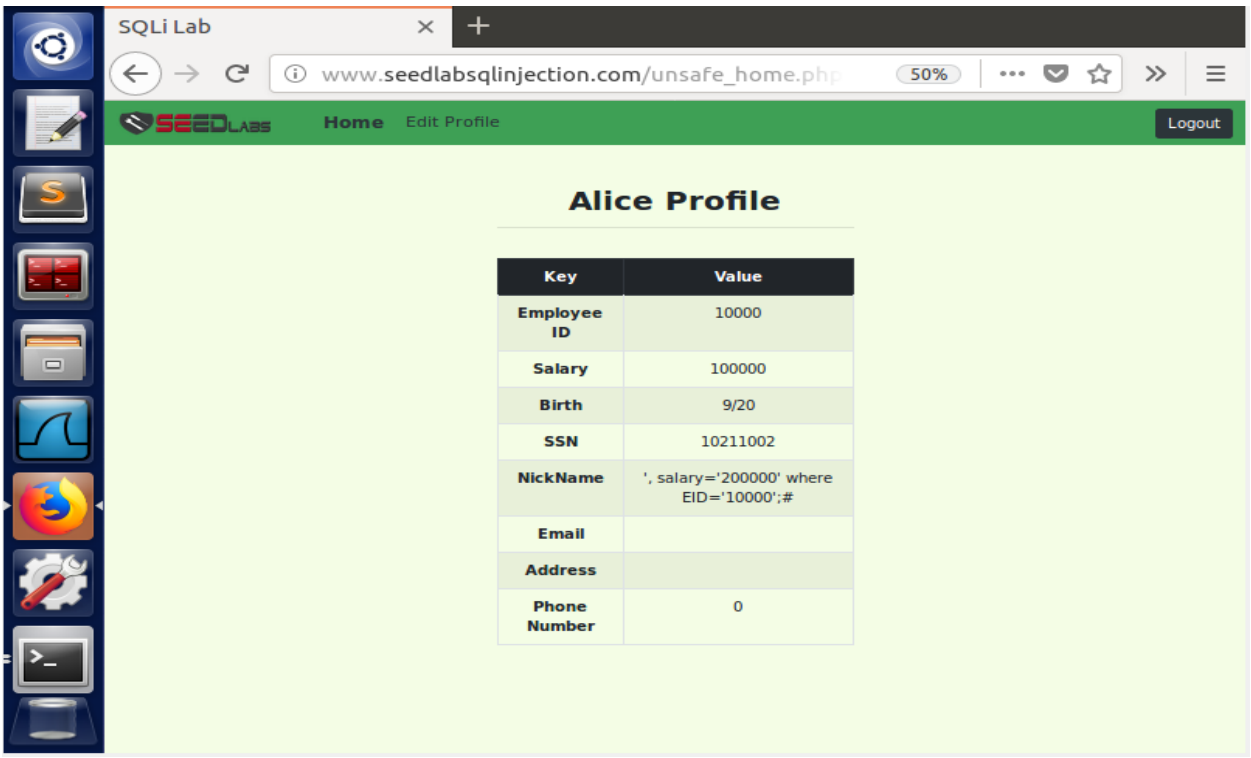


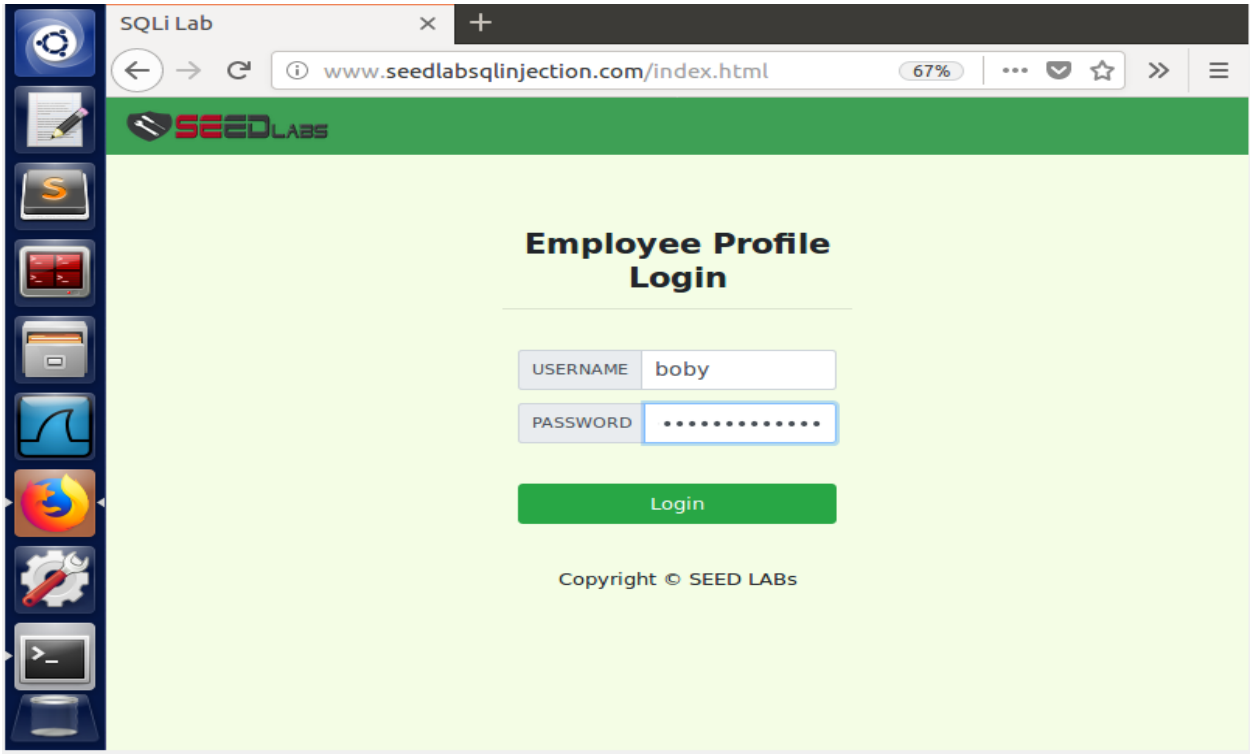
We now login as Alice and try to change her salary that is not an editable field.



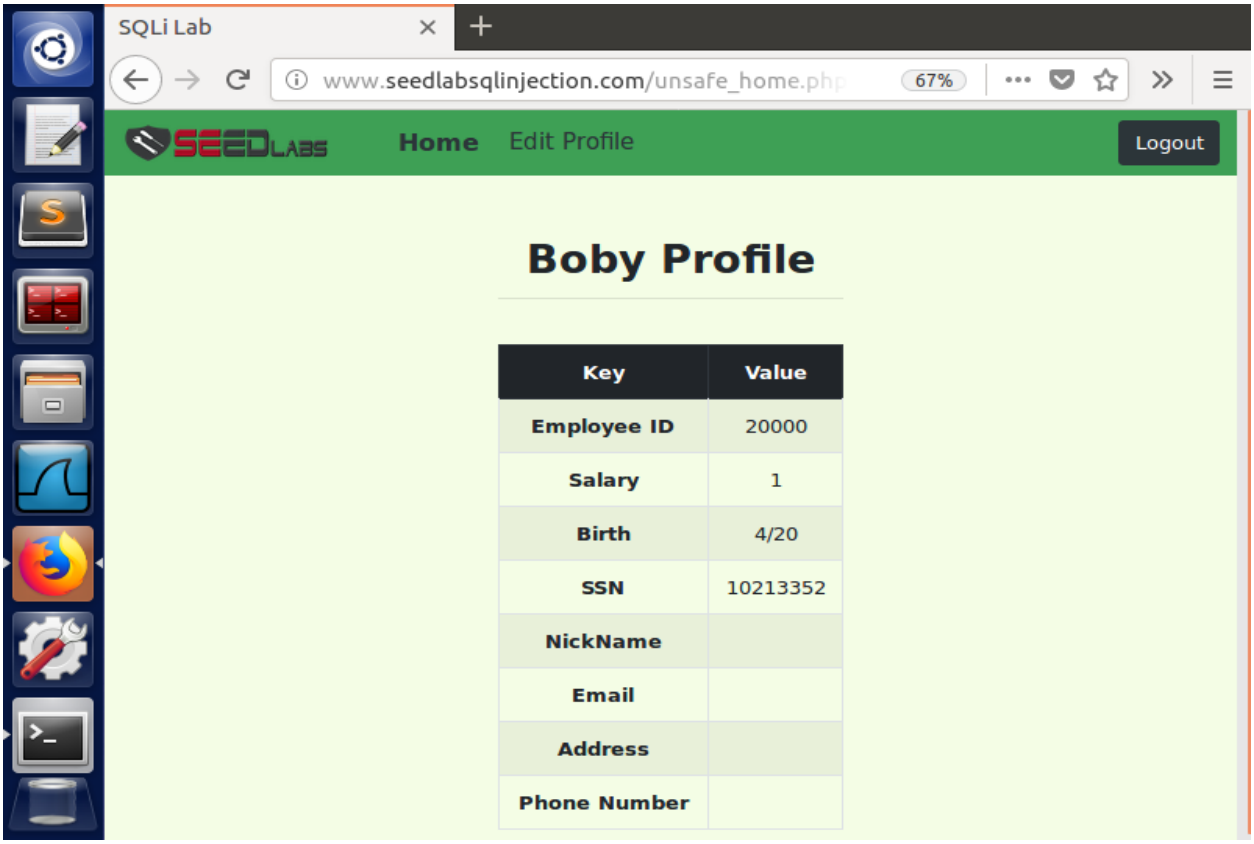


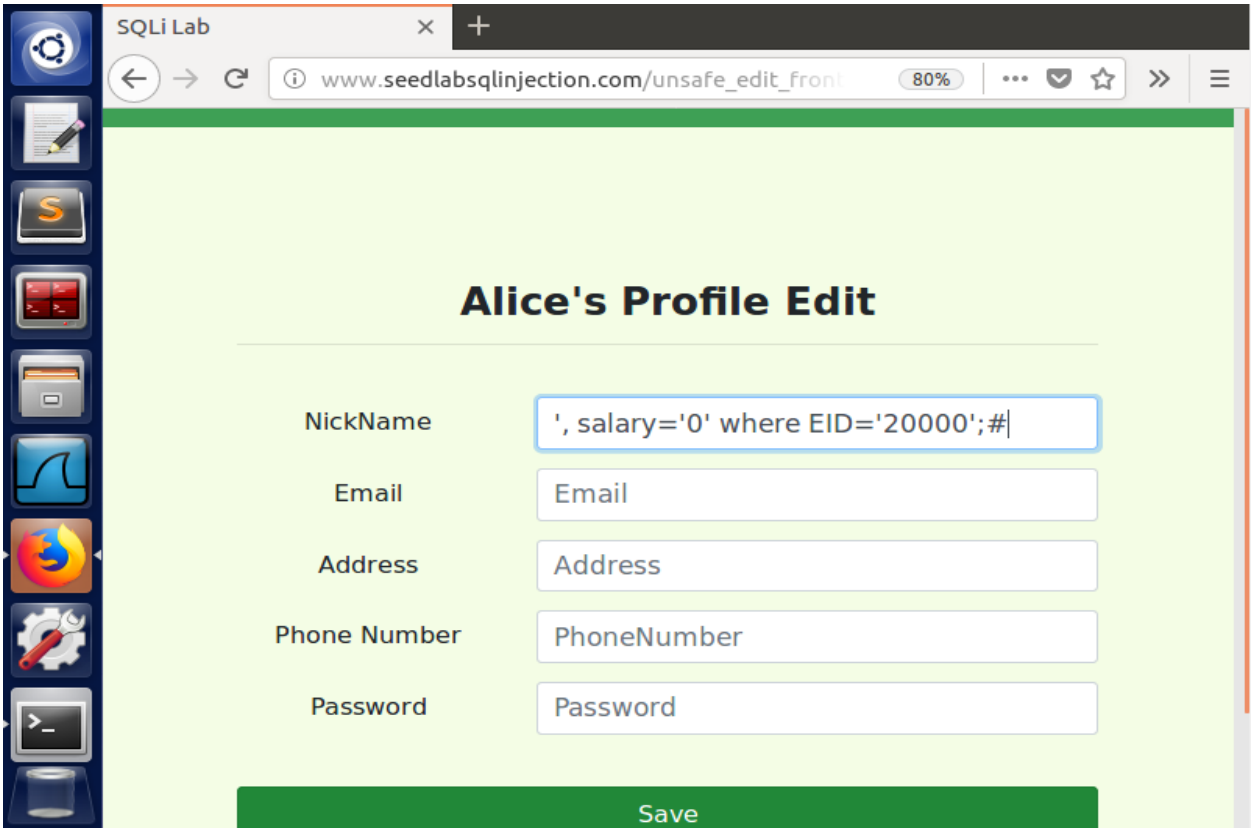
When Aluce tries to exploit the code by writing her Salary update query in the Nickname field, she sees that her Nickname gets changed instead of making any changes to her salary. She is unable to do SQL injection attack of UPDATE statement.



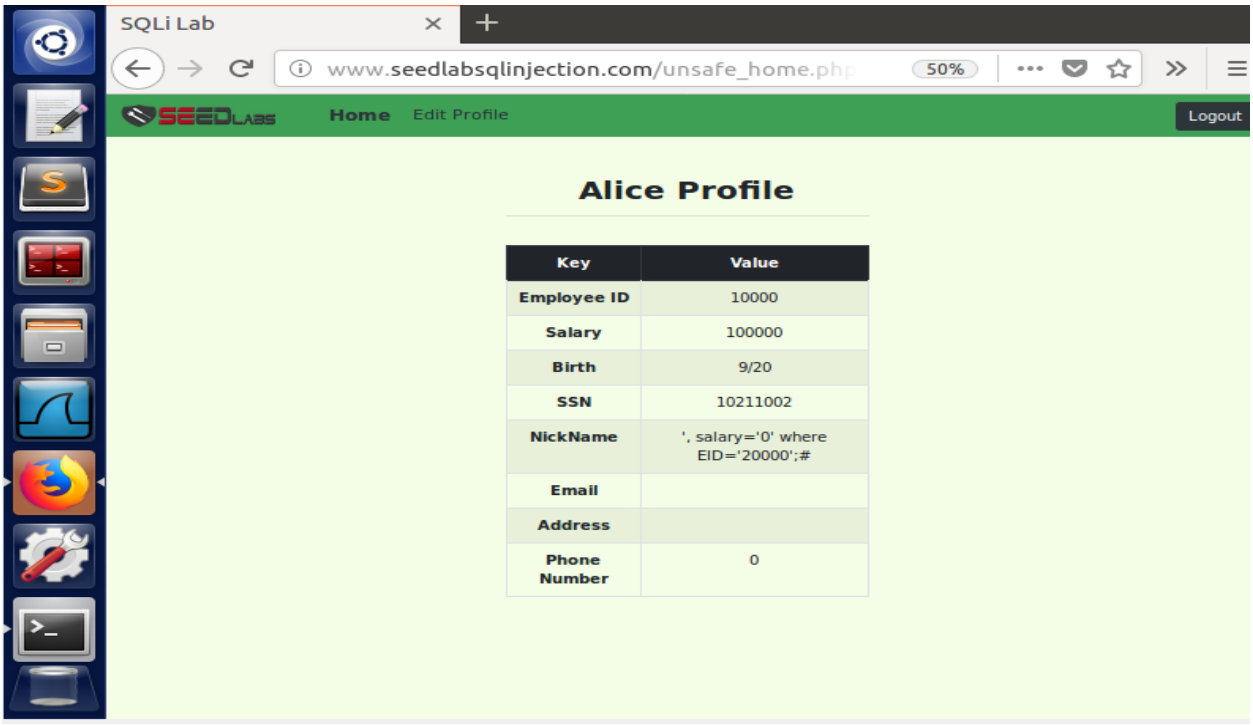


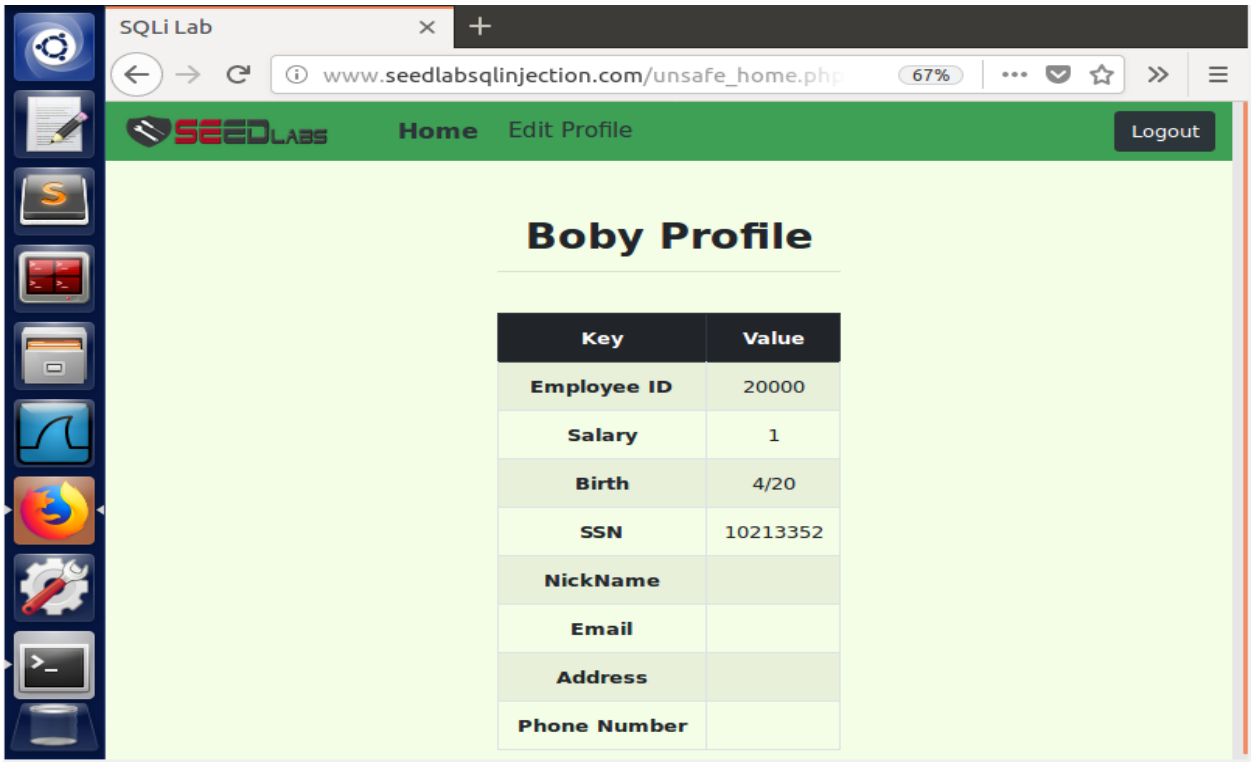
Boby logs in to his profile and checks that his salary and profile are unchanged.



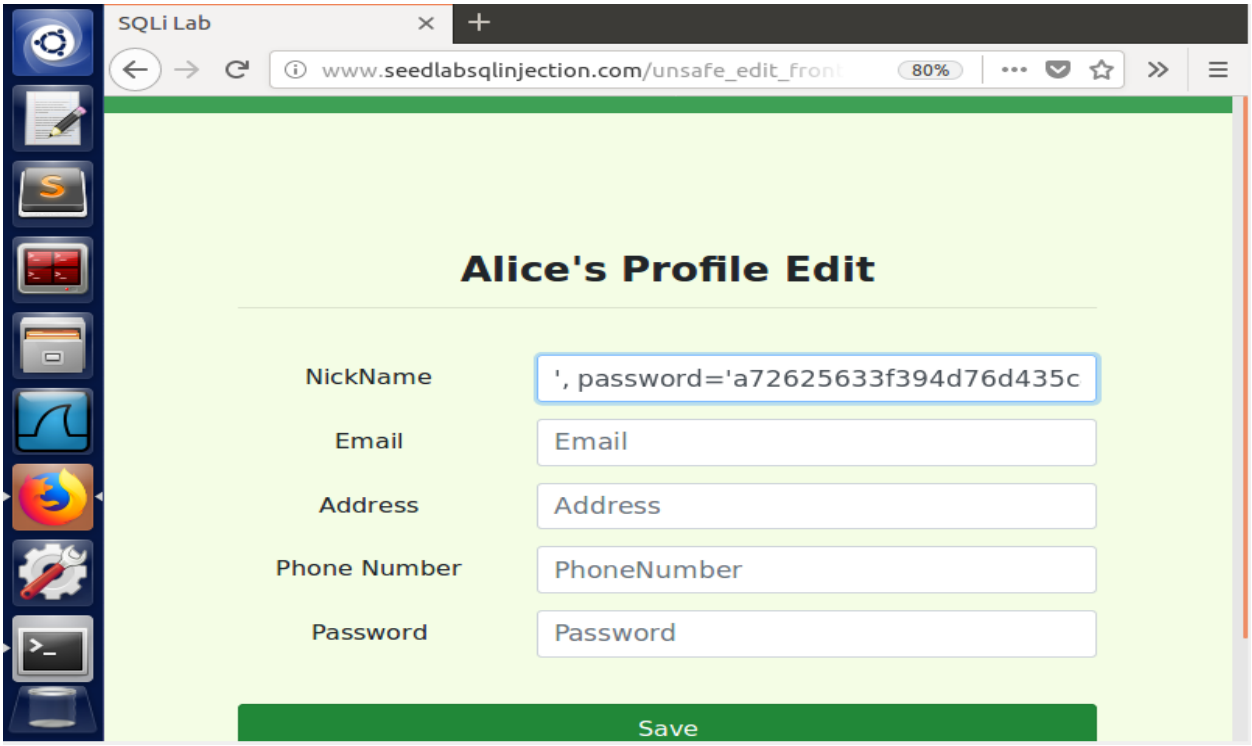


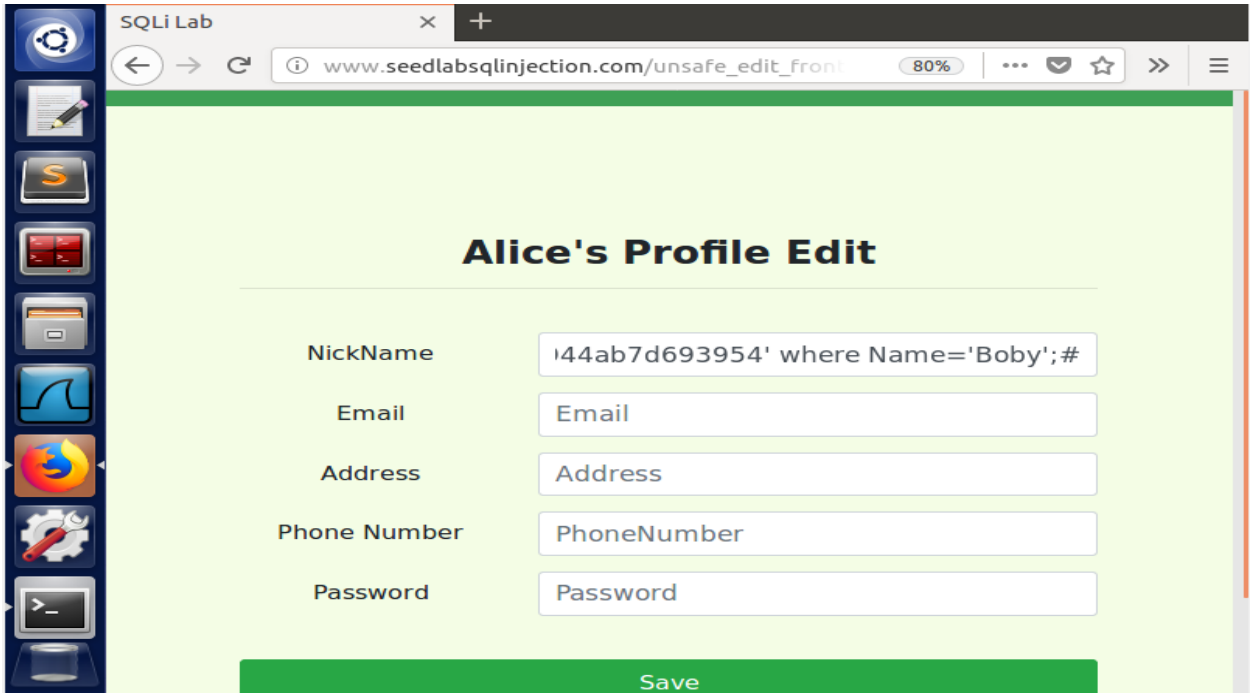
Alice tries to modify Boby's salary again to change his salary to $0. She logs in to her profile and Edits the Nickname field with the query she wants to execute. When she saves her profile, only her Nickname changes.



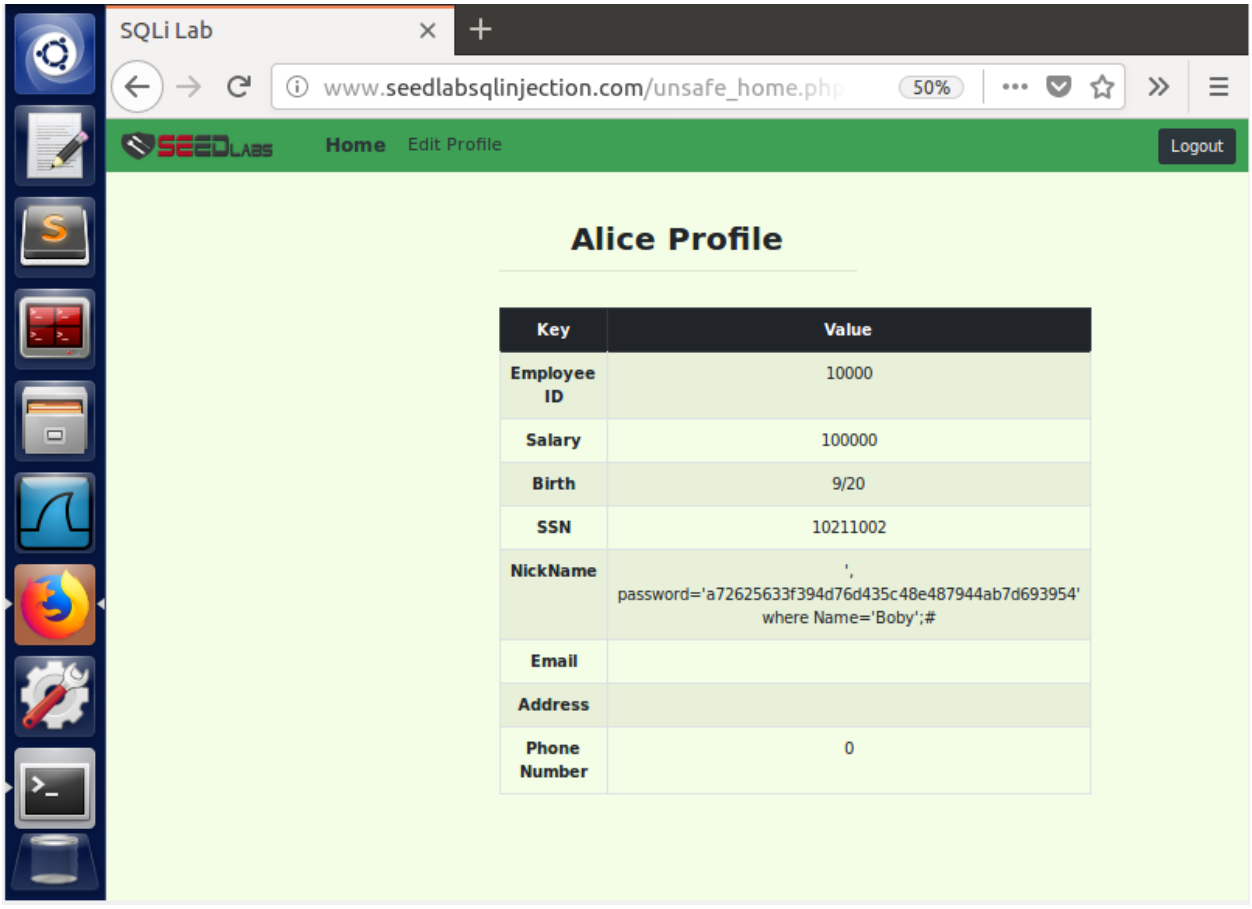


Boby's profile remains unchanged by this SQL injection attack. Alice then tries to change Boby's password using the hashed value she found out. She uses the same trick as before and tries to update Boby's profile from her Edit profile page in the Nickname field.

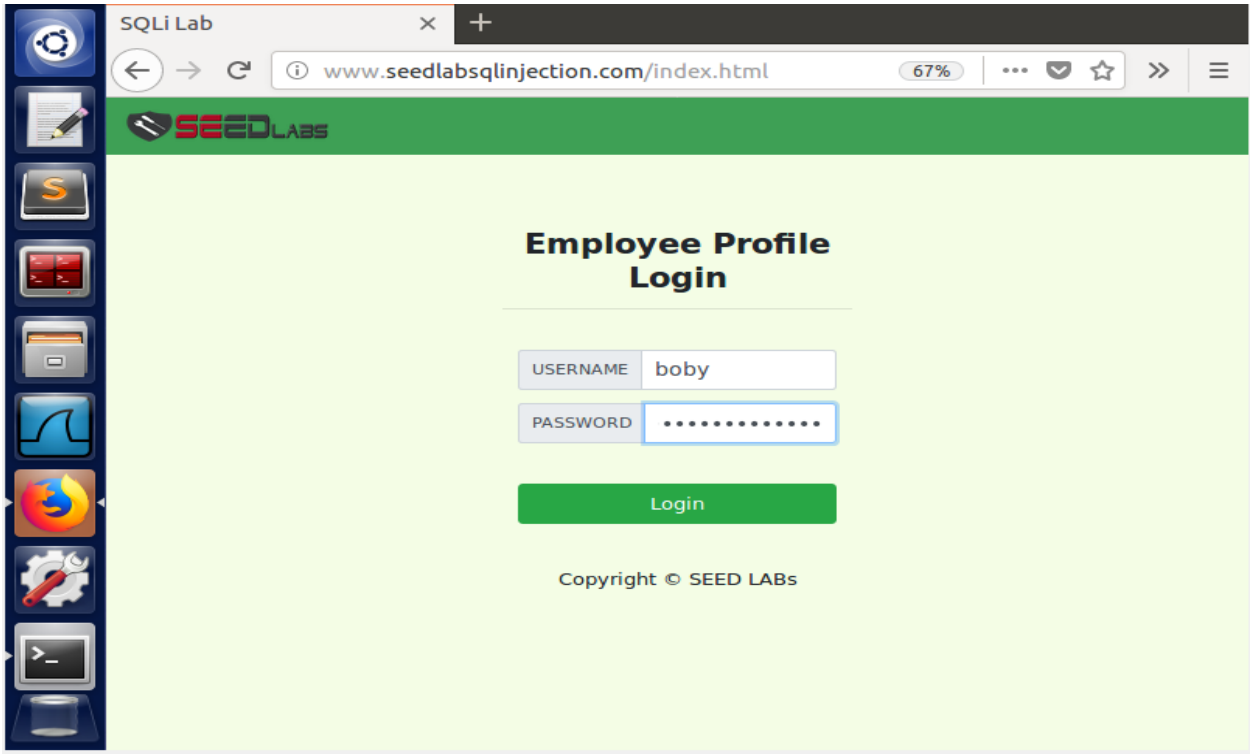


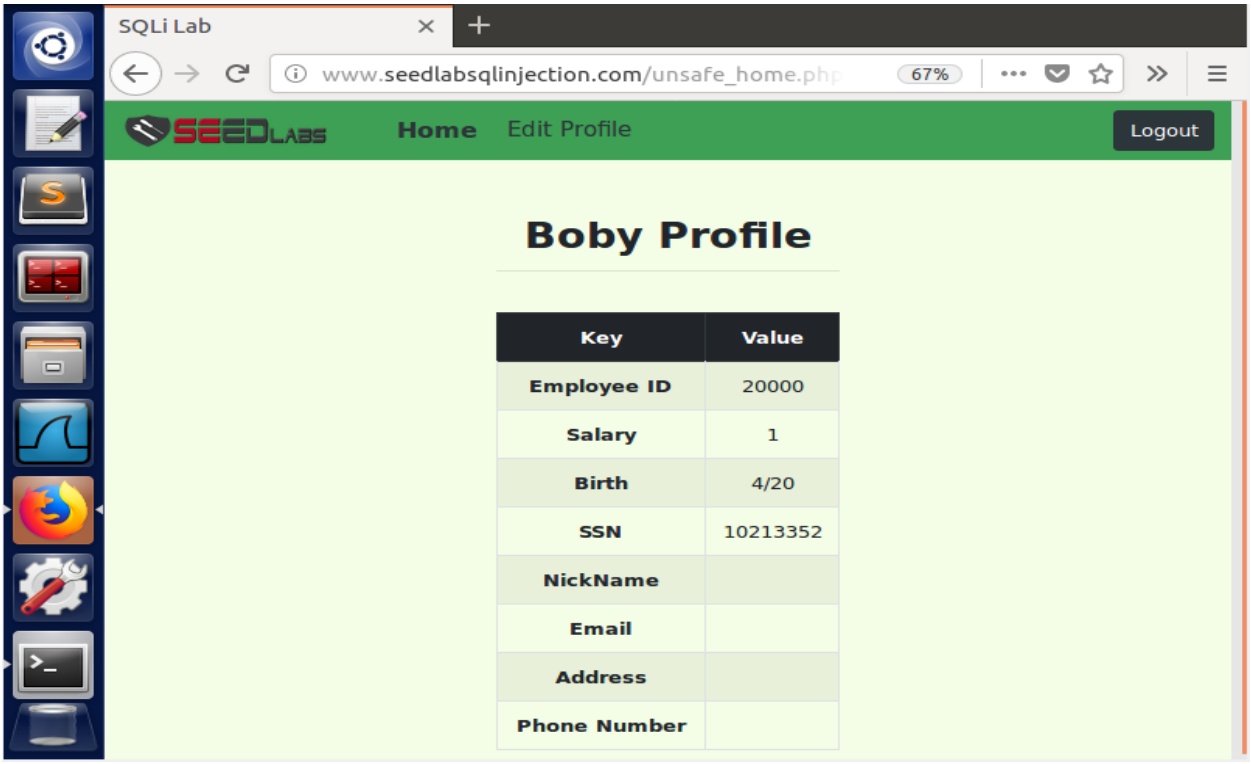


When she saves her profile, only her Nickname field is changed.



Boby's password remains unchanged by this update. When Boby los in, he finds no change in his profile.





Conclusion: By implementing the countermeasure, all the previous tasks 2 and 3 failed. This proves that by using the prepared statement in the SQL in both the PHP files (unsafe_home.php and unsafe_edit_backend.php), we see that we are able to prevent SQL injection attack of any type. Both

SELECT and UPDATE statements are secured and no kind of attack can happen. The Query and the data parts are separated from each other in the compilation phase. The data is directly fed into the compiled statements and then results are produced. This totally negates all the chances of changing the code indirectly or making an SQL injection attack. The changed statements shown below:

SELECT statement with countermeasure:

```
$stmt = $conn-> prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
      FROM credential
      WHERE name= ? and Password=?");
      $stmt->bind_param("ss", $input_uname,$hashed_pwd);
      $stmt->execute();
      $stmt->bind_result($bind_id, $bind_name, $bind_eid, $bind_salary, $bind_birth, $bind_ssn, $bind_phoneNumber, $bind_address, $bind_email, $bind_nickname, $bind_Password);
      $stmt->fetch();
```

UPDATE statement with countermeasure:

```
$stmt = $conn-> prepare("UPDATE credential SET nickname=?,email=?,address=?,Password=?,PhoneNumber=? where ID=?;");
    $stmt->bind_param("ssssii",$input_nickname,$input_email,$input_address,$hashed_pwd,$input_phonenumber,$id);
    $stmt->execute();
    $stmt->close();
```

These SQL statements use prepare statement. This prevents any kind of SQL injection attack. This countermeasure used prevented all the attacks using SELCT and UPDATE statements in SQL.

References:

https://github.com/aasthayadav/CompSecAttackLabs/blob/master/10.%20SQL%20Injection%20Attack/Lab%2010%20SQL%20Injection%20Attack.pdf