Name: Bharath Karumudi
Lab: SQL Injection Attack

Task 1: Get Familiar with SQL Statements

```
[11/16/19]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 1597
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;
ERROR 1049 (42000): Unknown database 'users'
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> describe credential;
+-------------+-----------------+------+-----+---------+----------------+
| Field       | Type            | Null | Key | Default | Extra          |
+-------------+-----------------+------+-----+---------+----------------+
| ID          | int(6) unsigned | NO   | PRI | NULL    | auto_increment |
| Name        | varchar(30)     | NO   |     | NULL    |                |
| EID         | varchar(20)     | YES  |     | NULL    |                |
| Salary      | int(9)          | YES  |     | NULL    |                |
| birth       | varchar(20)     | YES  |     | NULL    |                |
| SSN         | varchar(20)     | YES  |     | NULL    |                |
| PhoneNumber | varchar(20)     | YES  |     | NULL    |                |
| Address     | varchar(300)    | YES  |     | NULL    |                |
| Email       | varchar(300)    | YES  |     | NULL    |                |
| NickName    | varchar(300)    | YES  |     | NULL    |                |
| Password    | varchar(300)    | YES  |     | NULL    |                |
+-------------+-----------------+------+-----+---------+----------------+
11 rows 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 |             |         |
 |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-------------+---------+------
-+----------+------------------------------------------+
1 row in set (0.00 sec)

mysql>
```

Observation: Connected to MySQL with the credentials, used “Users” database and extracted the ‘Alice’ profile from the credential table.

Explanation: Connected to the MySQL using the MySQL command and with credentials. Then users the “use Users” to select the database “Users”. Listed the tables in the database using the show tables and from there identified the “credential” table and selected the data from the table using select statement with a where clause on Name. The database returned the Alice profile details.

Task 2: SQL Injection Attack on SELECT Statement

Task 2.1: SQL Injection Attack from webpage.

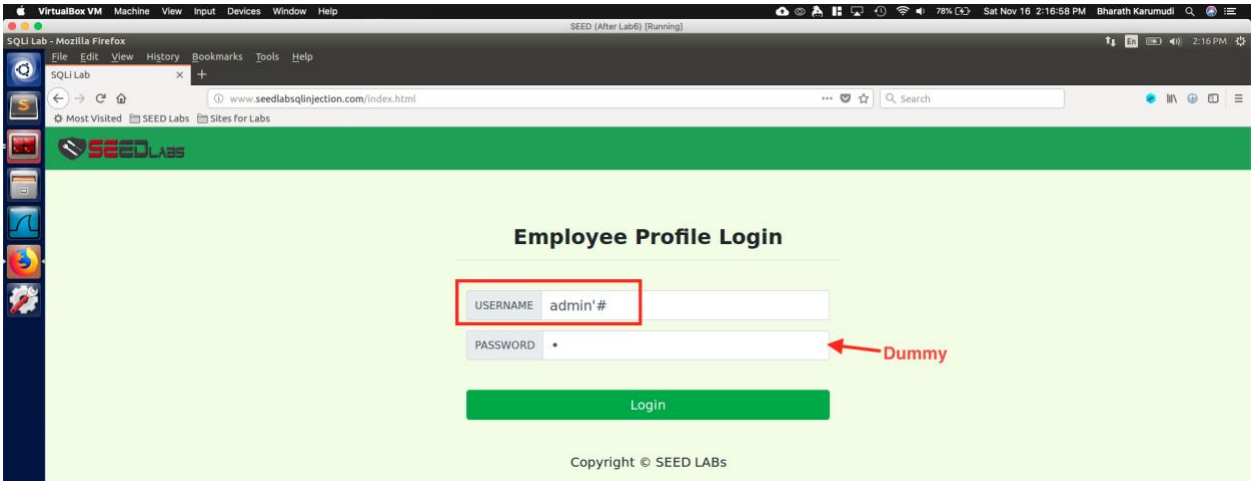
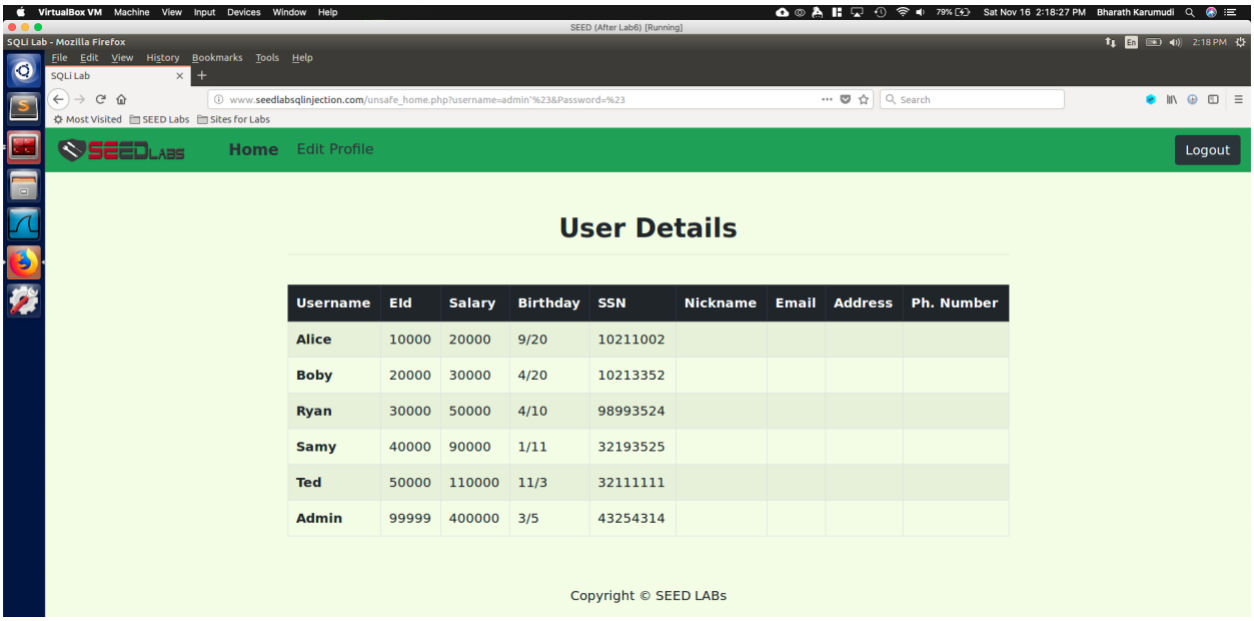


Fig: SQL Injection Attack on Admin id



Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

Fig: Successfully logged into Admin account

Observation: When logged in with username admin’# and an arbitrary password, able to login into the application and see all the information of all the employees.

Explanation: As the username was passed as admin’#, the single quote in the username helped in closing the name condition and the # helped in commenting the rest. So when the SQL framed on the application server, it parsed as name=’admin’#’ and Password= ’$hashed_pwd’"; So all the statement after the # was commented/ignored and bypassed the password validation. Thus, as an attacker able to login into the page even if I do not know the admin password.

Task 2.2: SQL Injection Attack from command line

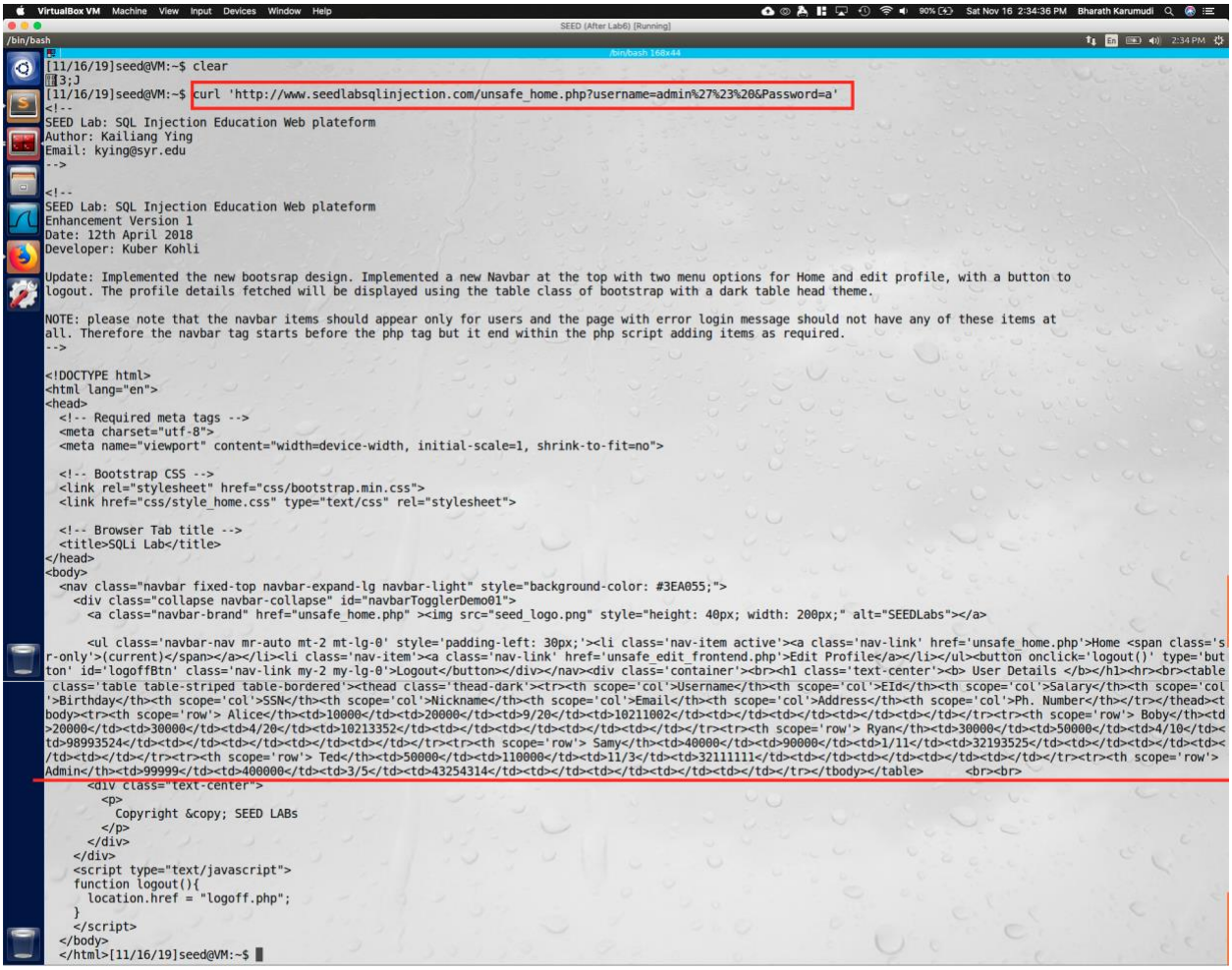


Fig: SQL injection from command line using CURL and the attack output

Observation: Using curl performed the attack to get the all employee details. Framed the curl URL as shown in the image and with a dummy password. After execution, I was able to see all the employee details.

Explanation: The curl command that we sent has the parameters that can bypass the password credentials. The parameters are framed in such a way, the username=admin’# &Password=a with this the password validation will be by passed and able to retrieve all the employee information.

Task 2.3: Append a new SQL statement

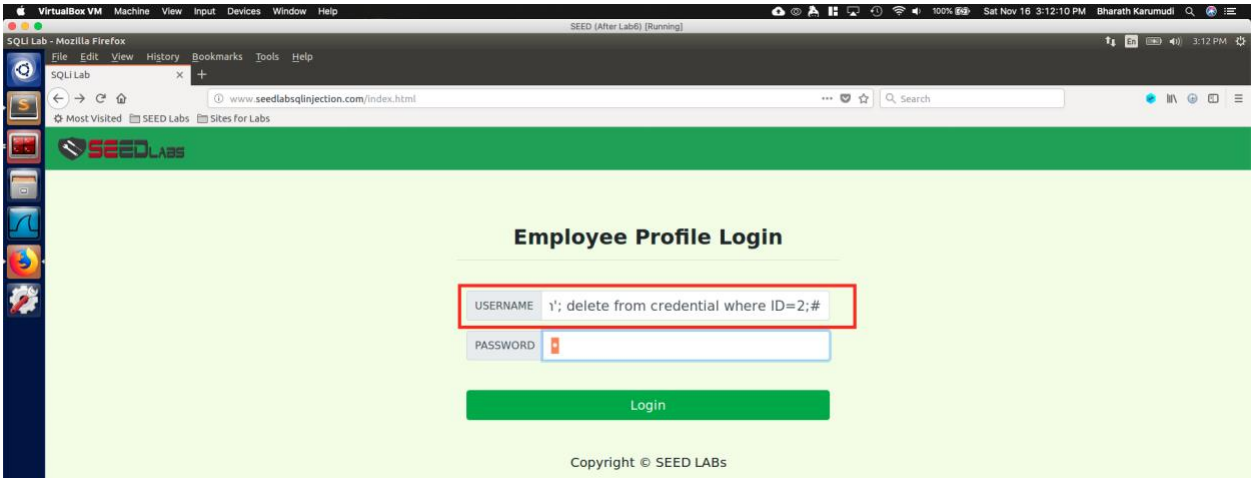


Fig: Attack from login page to delete a record with SQL Append

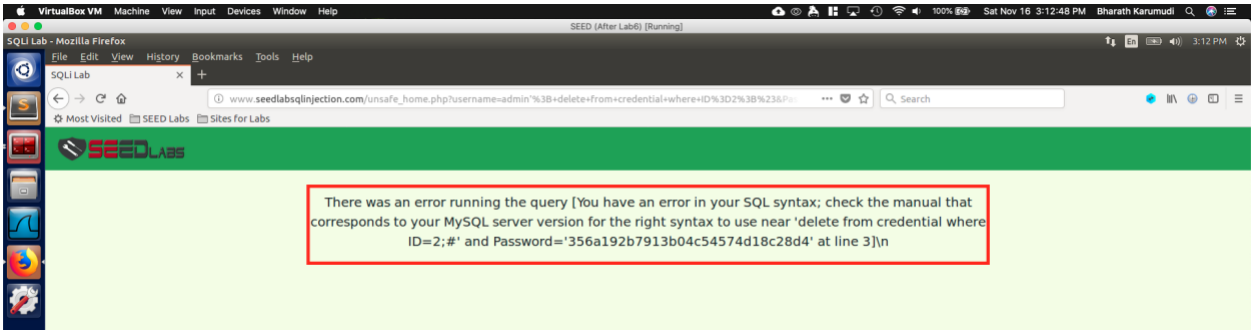


Fig: Attack failed

Observation: Tried to perform the SQL Injection by appending the delete statement with an intention to delete the ID=2 record. *admin'; delete from credential where ID=2;#* But the attack was unsuccessful.

Explanation: The attack was unsuccessful because the mysqli does not allow multiple queries and prevents this kind of SQL injection attack.

Task 3: SQL Injection Attack on UPDATE Statement

Task 3.1: Modify your own salary

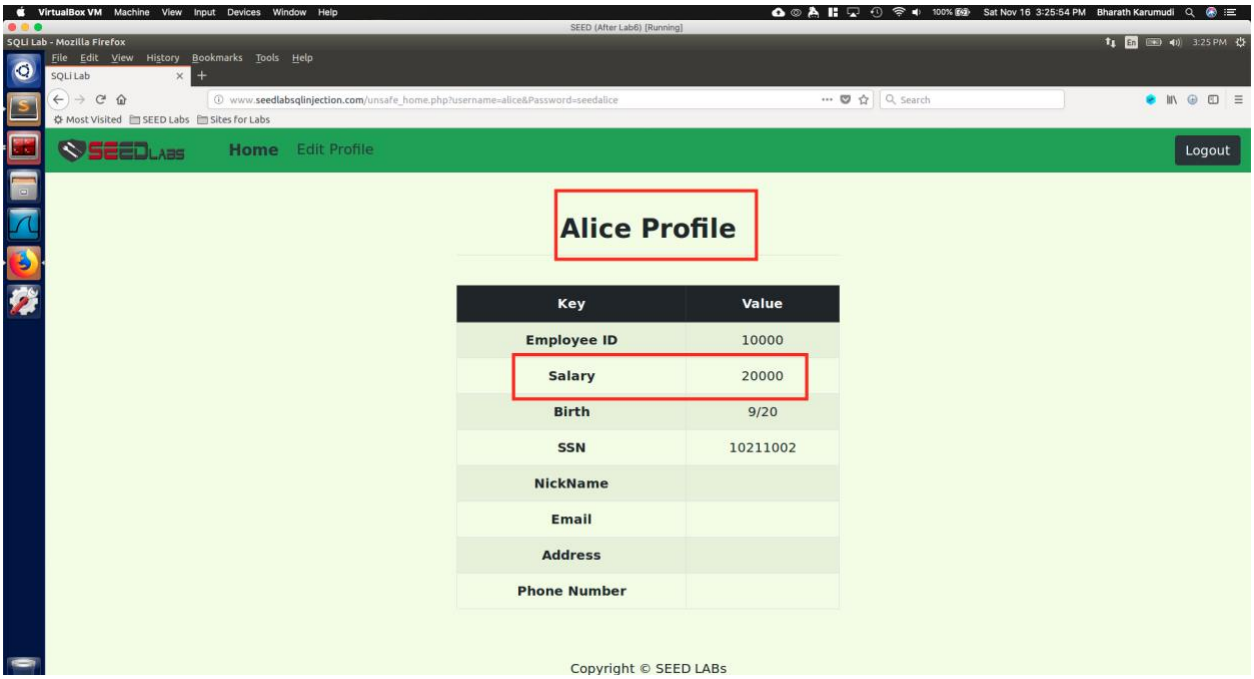


Fig: Alice Profile before updating the Salary and the value is 20000

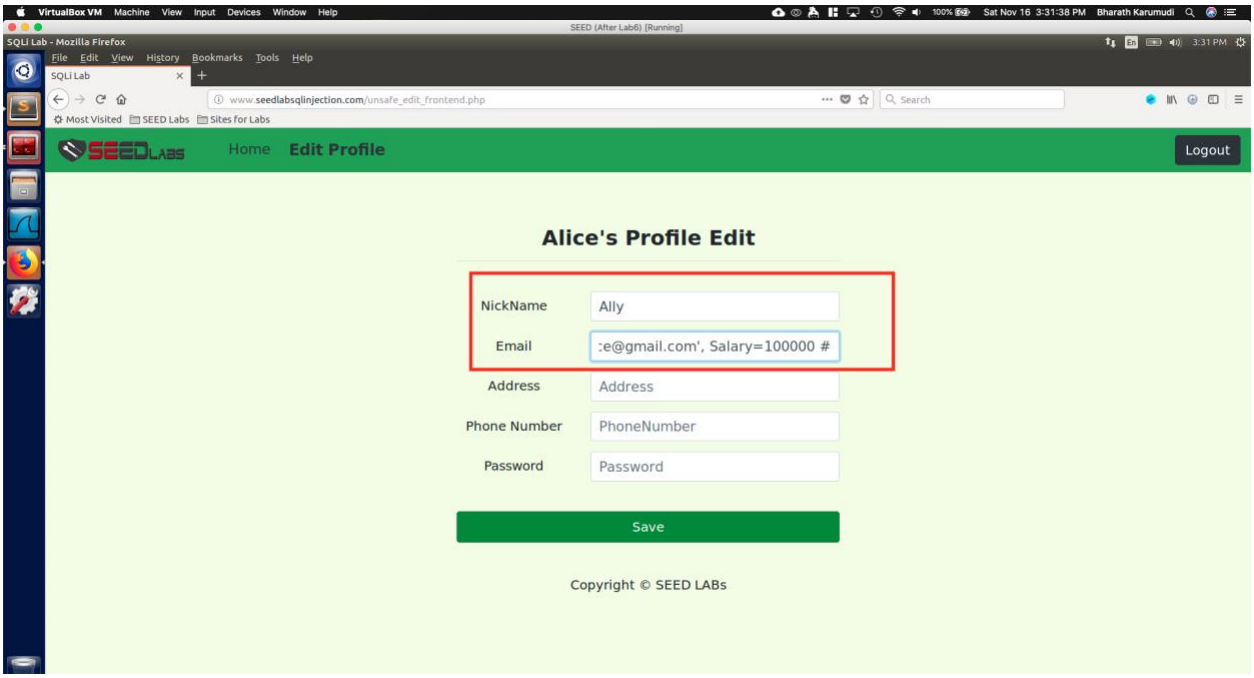


Fig: Alice trying the SQL Injection to update the Salary to 100000

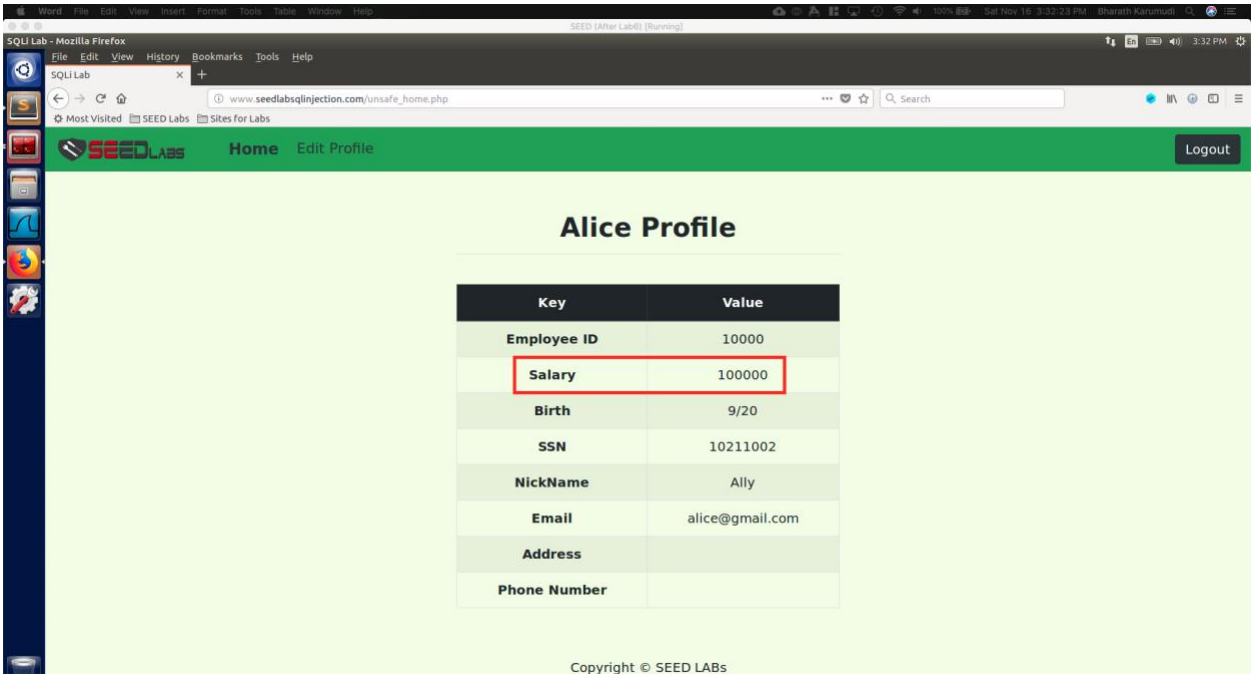


Fig: Alice successfully updated her Salary to 100000

Observation: Logged into Alice profile and went to edit profile page. In Edit profile page, key-in the details and in email field gave the input as *alice@gmail.com', Salary=100000 #* and once saved, the Salary was now updated to 100,000.

Explanation: This was due to SQL injection vulnerability, where Alice went to her profile page and gave the input such a way, where it will update her email id and also gave the Salary column information. The user input was interpreted as code and executed the SQL statement. Thus, she updated her Salary from 20000 to 100000.

Task 3.2: Modify other people' salary

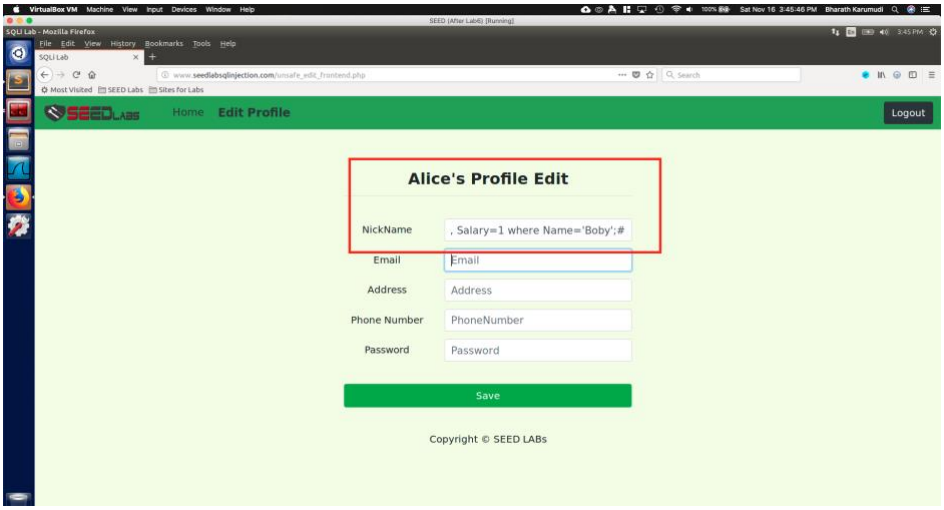


Fig: Alice updating the Boby Salary to 1 by SQL Injection

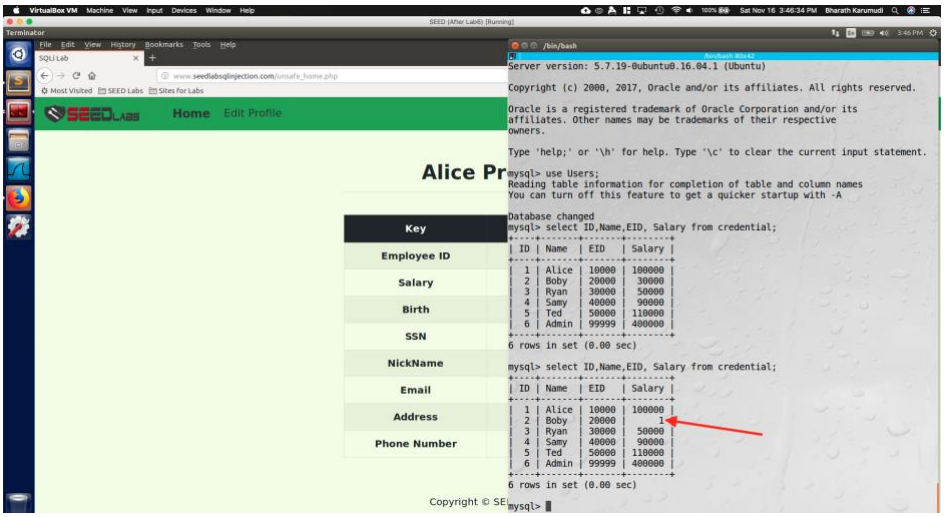


Fig: Boby Salary update to 1 successfully

Observation: As Alice, logged into account and in Edit Profile page, in the Nick Name field gave the input as: *Boby', Salary=1 where Name='Boby';#* and saved. When checked in the databas, the Salary of Boby was updated to 1.

Explanation: Using the SQL Injection and vulnerability in code, Alice key-in the details such a way, the SQL was executed with an intention to update the Boby Salary. The user data was interpreted as code and executed and thus Alice updated her boss Boby salary to 1.

Task 3.3: Modify other people' password.

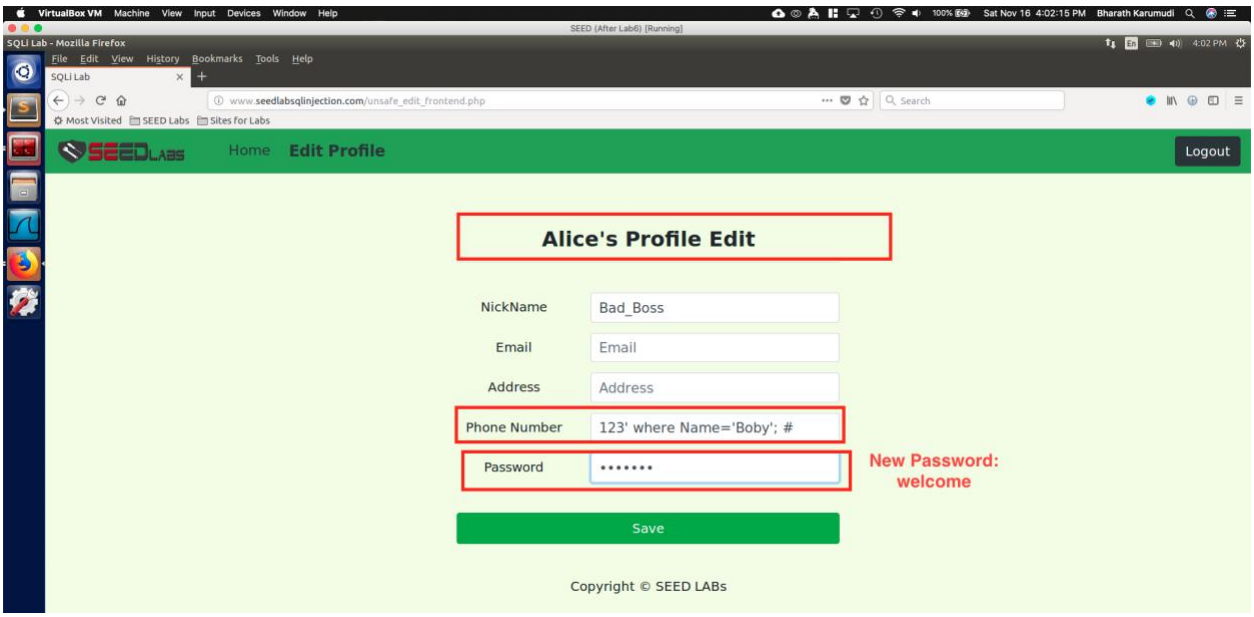


Fig: Alice updating the Boby password

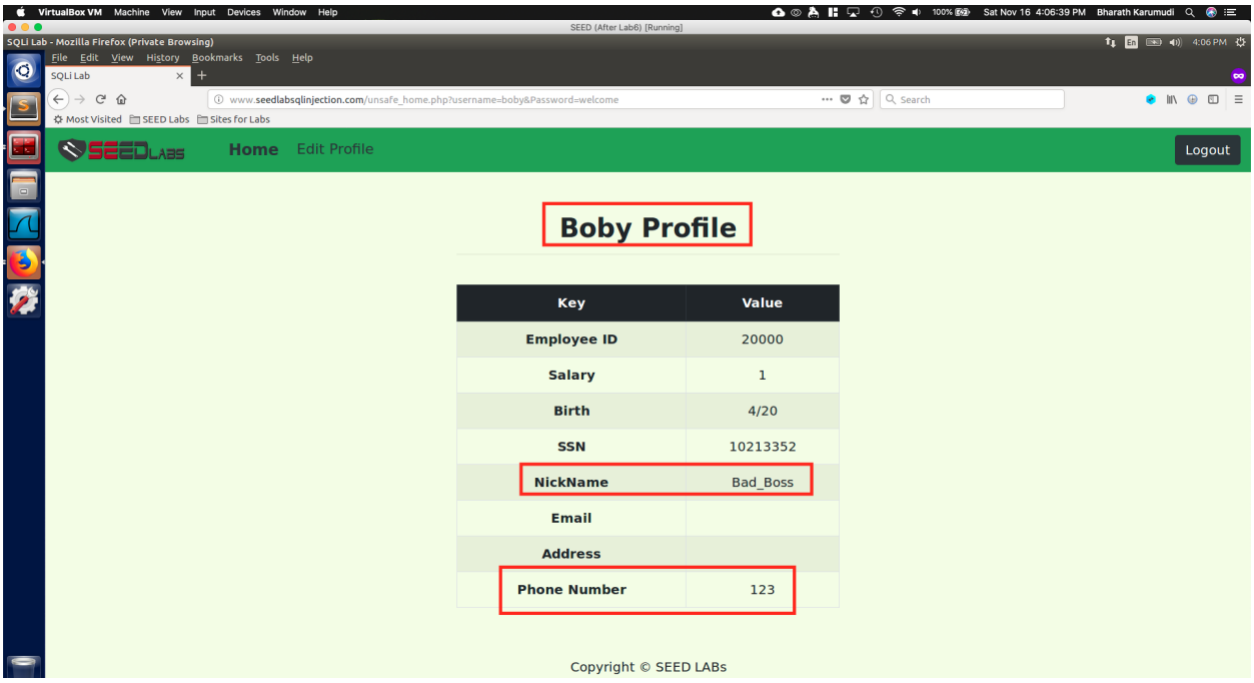


Fig: Profile details and password for Boby are updated. Logged in with new password

Observation: From Alice profile, using the Phone Number field, entered the values as *123' where Name='Boby'; #* and password set to welcome and also new NickName. Once submitted the values are updated and after that logged into Boby profile successfully using the new password.

Explanation: Using the SQL Injection and vulnerability in code, Alice key-in the details such a way, the SQL was executed with an intention to update the Boby password. The user data was interpreted as code and executed and thus Alice updated her boss Boby password.

Task 4: Countermeasure — Prepared Statement

a. Trying "SQL Injection Attack from webpage" to login with admin id.

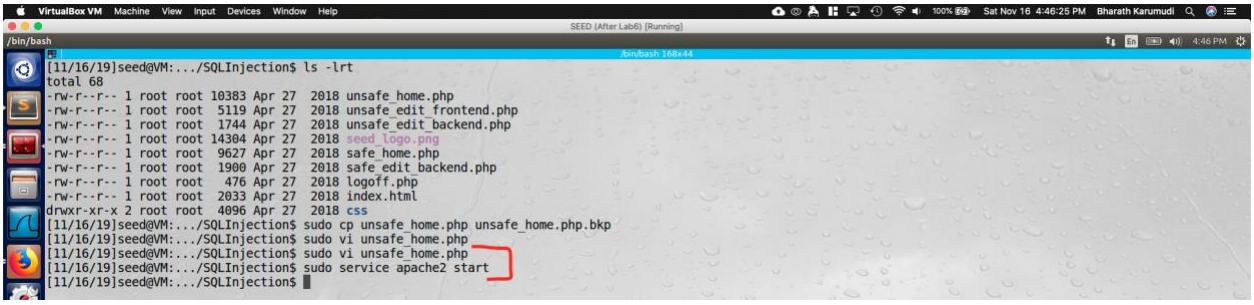


Fig: Updated the unsafe_home.php file

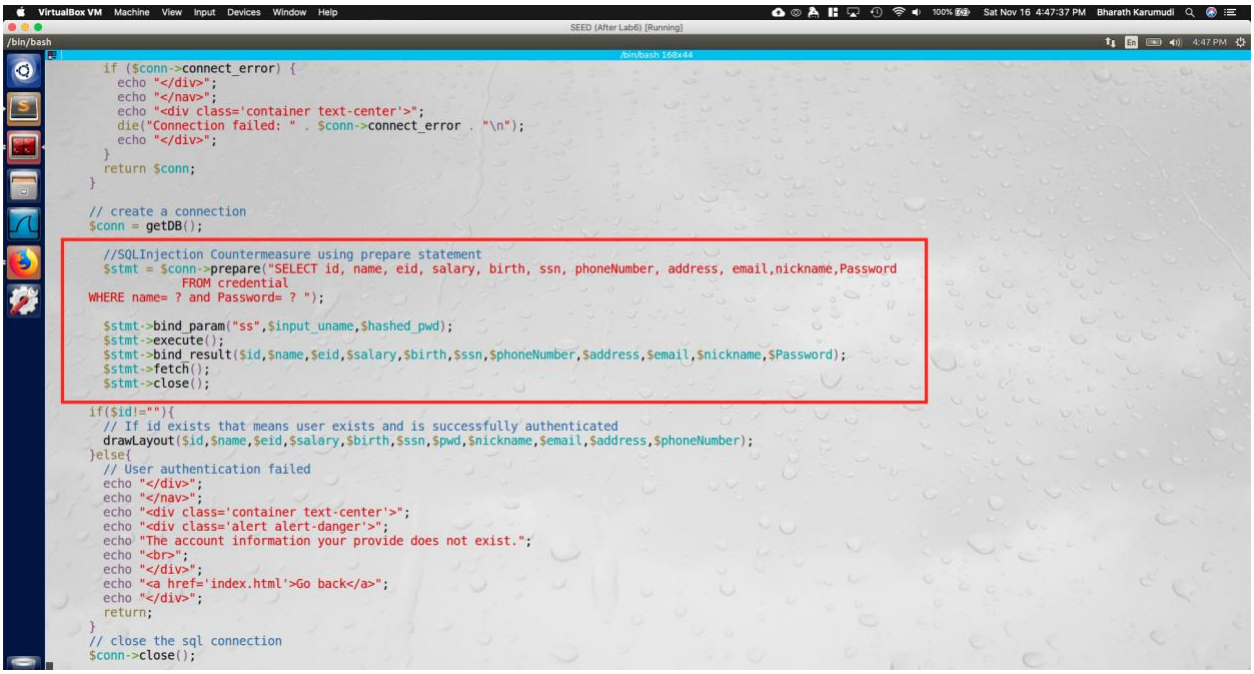


Fig: Implemented the SQL injection countermeasures in unsafe_home.php file

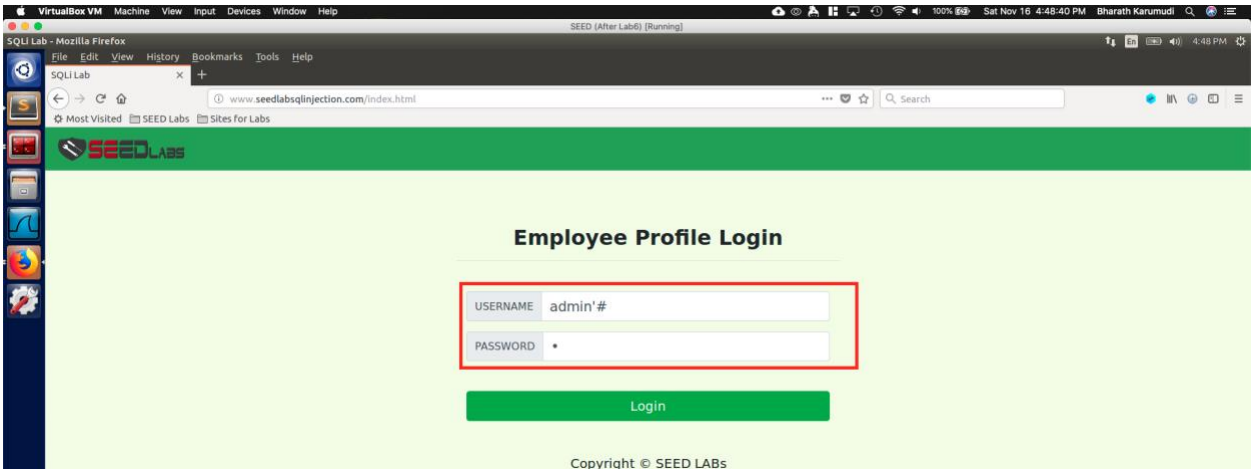


Fig: Trying the admin login attack

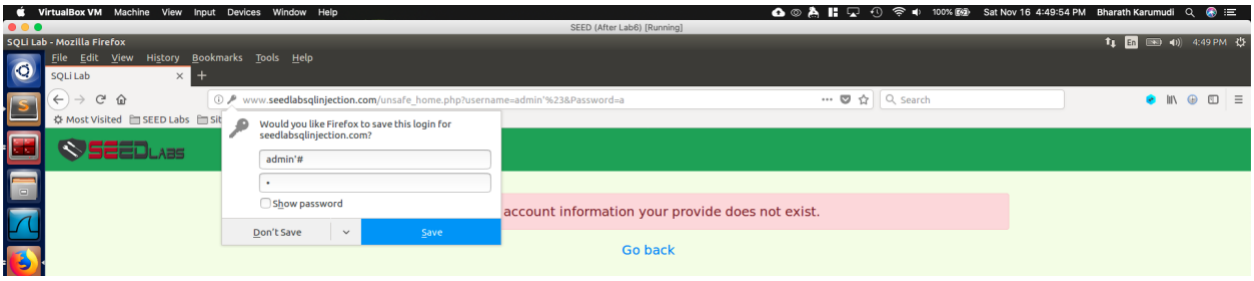


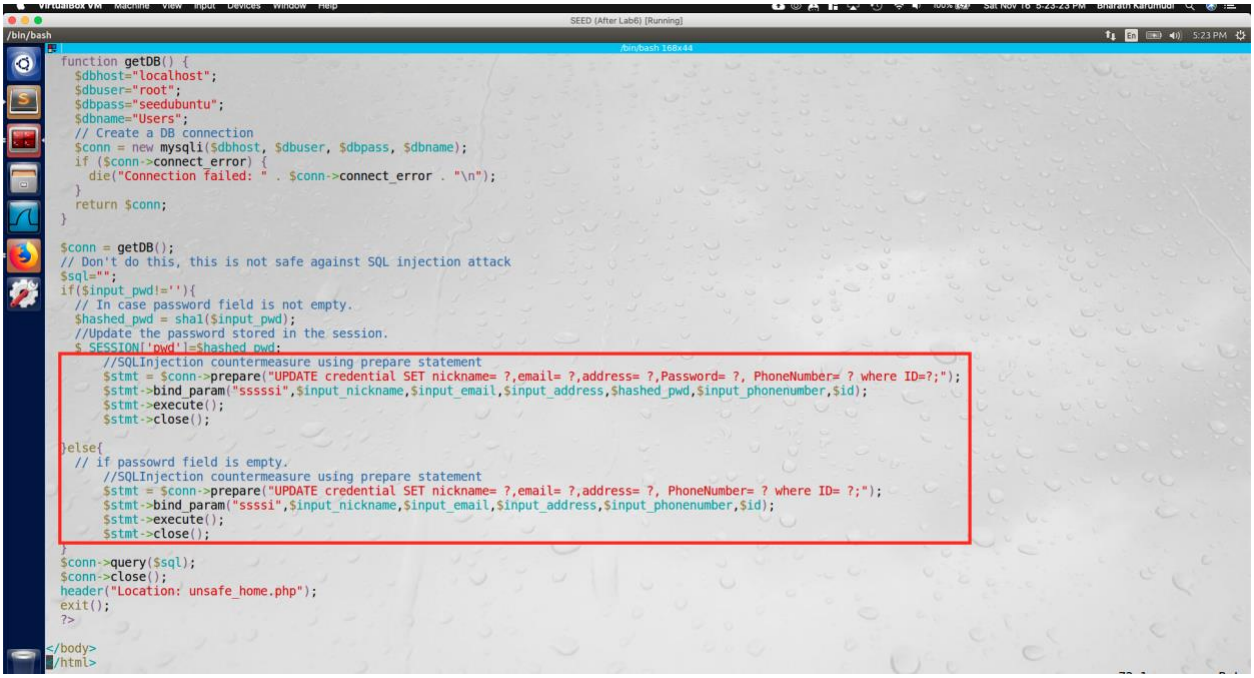
Fig: Attack failed

Observation: Modified the unsafe_home.php and added the prepare countermeasure for SQL statements. After modification, restarted the apache and tried to attempt the attack to login with admin as in Task 2.1; but this time the attack was failed.

Explanation: This is due to the countermeasure that was implemented in the unsafe_home.php ; unlike previous code, using the prepared statement mechanism, we divide the process of sending a SQL statement to the database into two steps:

The first step is to only send the code part, i.e., a SQL statement without the actual the data.
This is the prepare step. As we can see from the above code snippet, the actual data are replaced by question marks (?). After this step, we then send the data to the database using bind param(). The database will treat everything sent in this step only as data, not as code anymore. It binds the data to the corresponding question marks of the prepared statement.

b. Trying “Modify other people’ salary” attack



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

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

}else{
  // if passowrd field is empty.
      //SQLInjection countermeasure using prepare statement
      $stmt = $conn->prepare("UPDATE credential SET nickname= ?,email= ?,address= ?, PhoneNumber= ? where ID= ?;");
      $stmt->bind_param("ssssi",$input_nickname,$input_email,$input_address,$input_phonenumber,$id);
      $stmt->execute();
      $stmt->close();
}
$conn->query($sql);
$conn->close();
header("Location: unsafe_home.php");
exit();
?>

</body>
</html>
```

Fig: Modified unsafe_edit_backend.php file with countermeasures

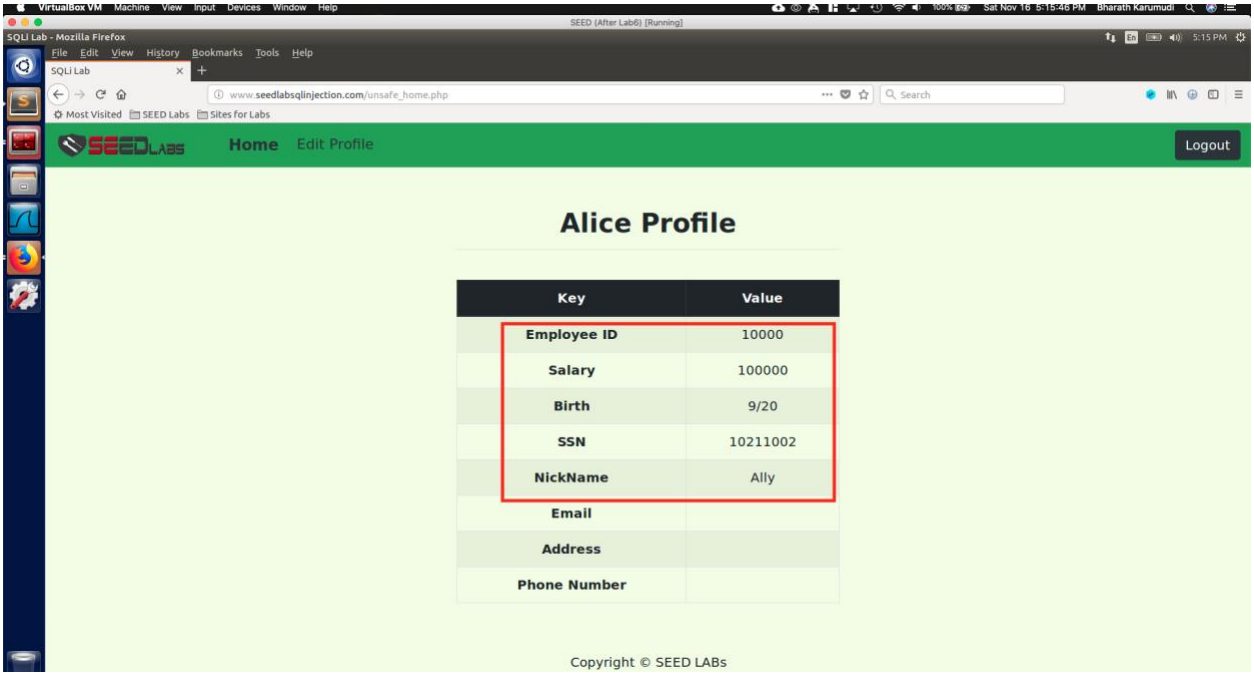


Fig: Alice Profile before attempting to update

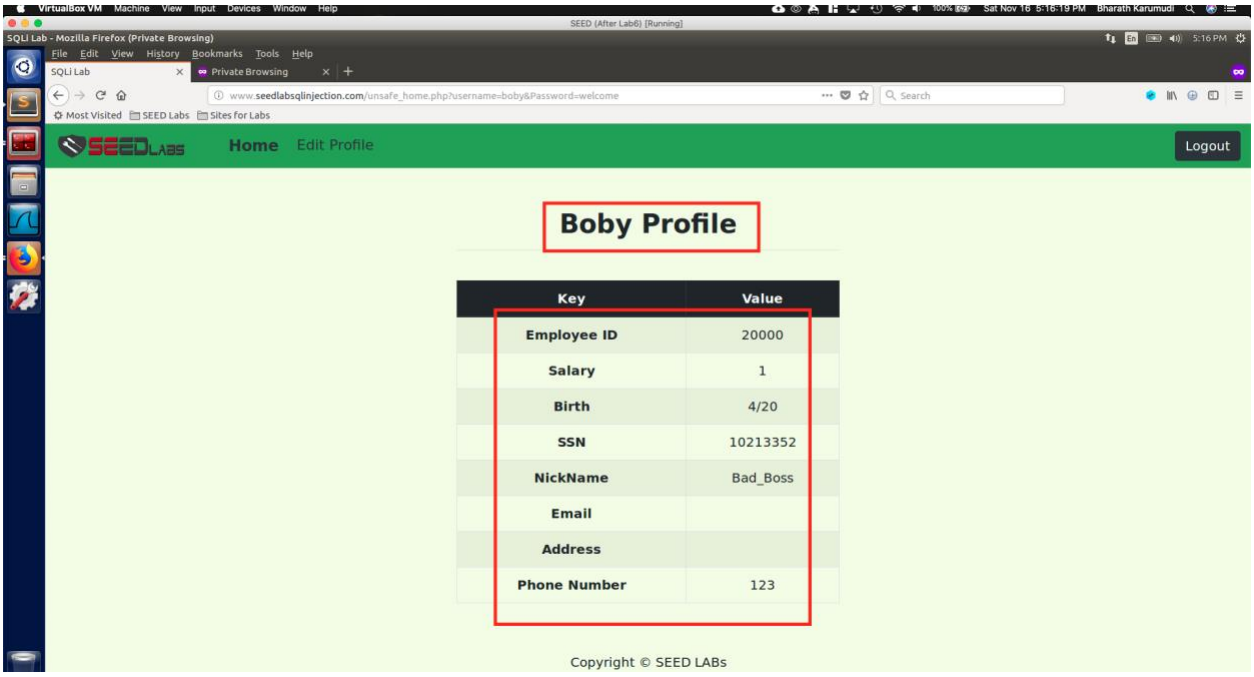


Fig: Boby profile before attack

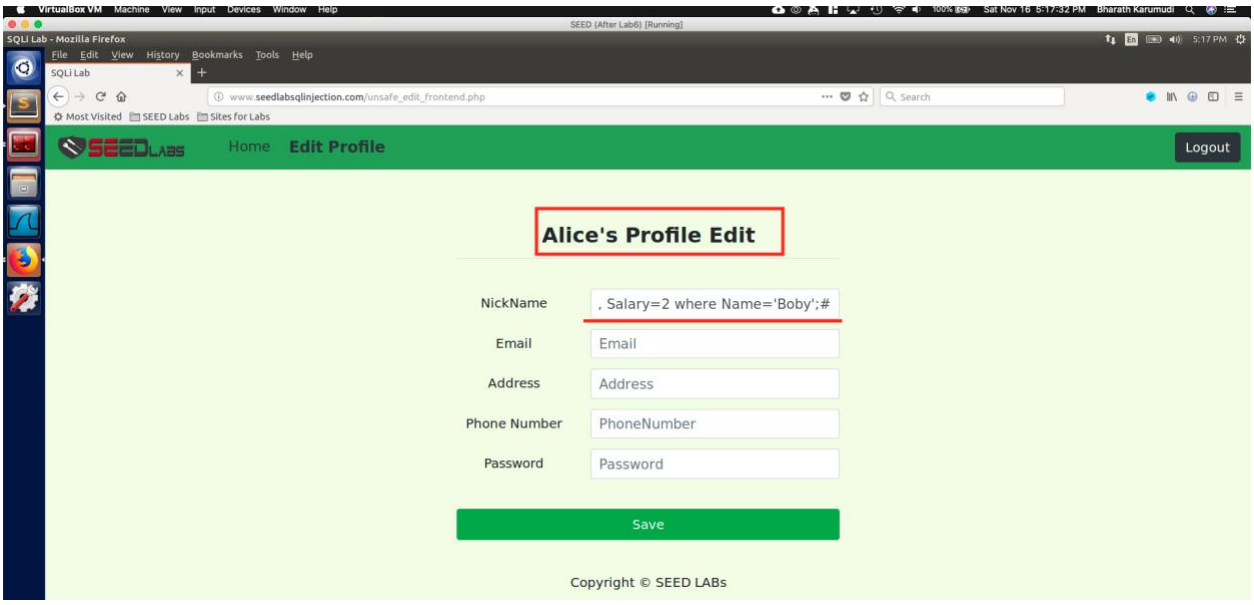


Fig: Alice trying the attack to update Boby Salary

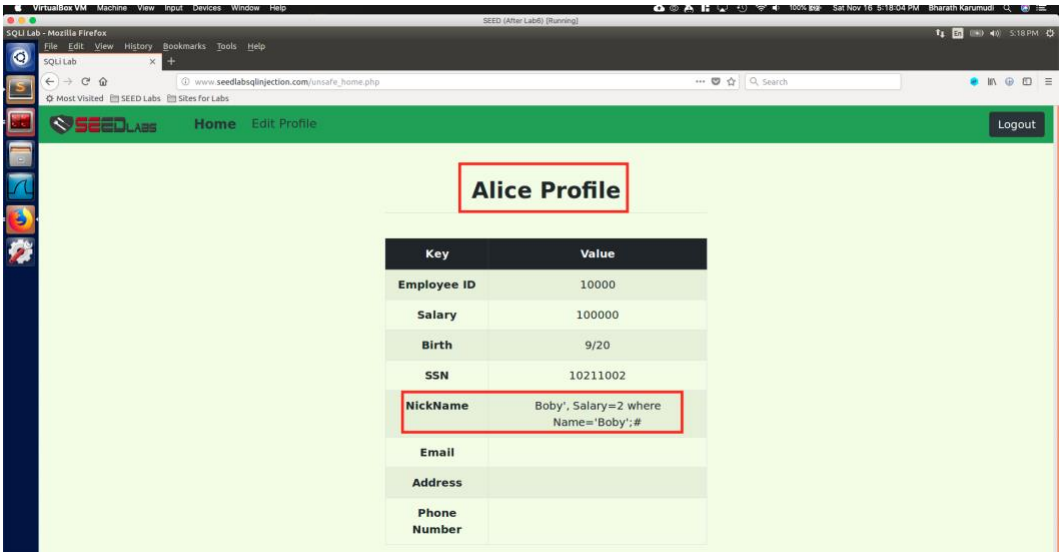


Fig: The update was happened to Alice profile – Attack failed

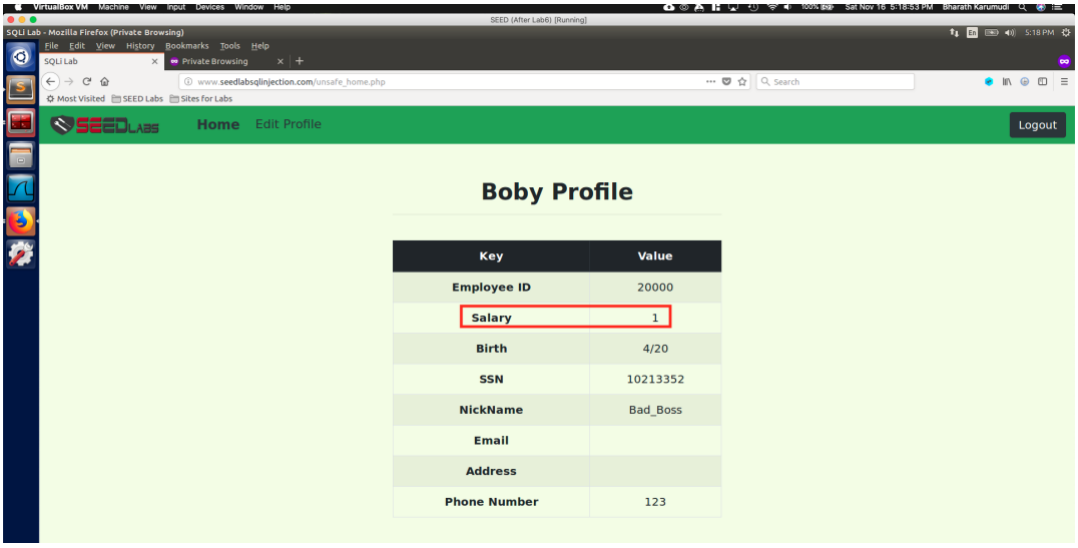


Fig: No impact on Boby profile after attack

Observation: Modified the unsafe_edit_backend.php and added the prepare countermeasure for SQL statements. After modification, restarted the apache and tried to attempt the attack to update the Boby Salary to 2 from Alice profile; but this time the attack was failed.

Explanation: This is due to the countermeasure that was implemented in the unsafe_edit_backend.php; unlike previous code, using the prepared statement mechanism, we divide the process of sending a SQL statement to the database into two steps:

The first step is to only send the code part, i.e., a SQL statement without the actual the data. This is the prepare step. As we can see from the above code snippet, the actual data are replaced by question marks (?). After this step, we then send the data to the database using bind param(). The database will treat everything sent in this step only as data, not as code anymore. It binds the data to the corresponding question marks of the prepared statement.