CSCI 476: Lab 06

Nathan Stouffer

March 5, 2020

Task 1

Task 1 asks us to familiarize ourselves with SQL statements. This is shown below in Figures 1 and 2. Specifically, Figure 1 shows opening mysql and Figure 2 displays a command that views a specific row of the database.

Figure 1: Opening mysql

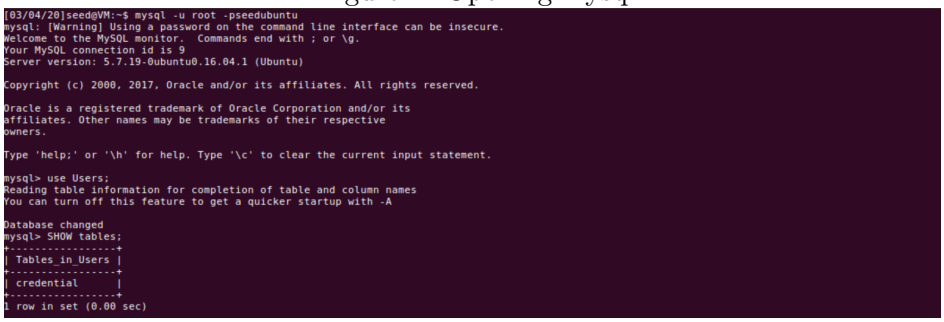
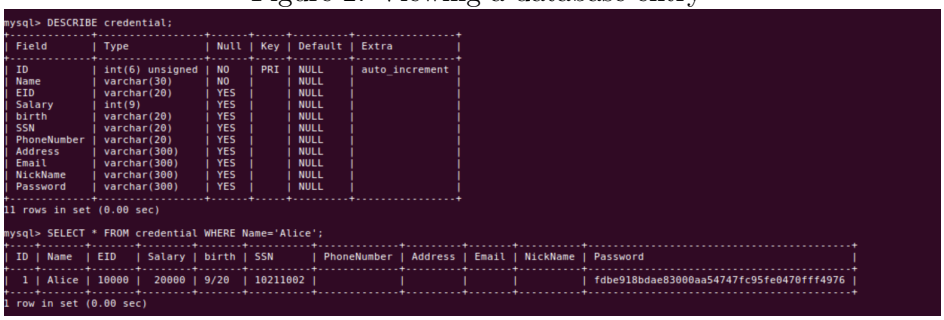


Figure 2: Viewing a database entry



Task 2

In Task 2, we perform the SQL injection attack on a vulnerable website set up by SEED Labs. The vulnerable php code for the log in page is shown in Figure 3. The php code takes whatever is entered by the user and compiles it. This means a specially crafted input can change what is required to log in. This is shown in the subtasks below.

Figure 3: Vulnerable php code

```
// create a connection
$conn = getDB();
// Sql query to authenticate the user
$sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
FROM credential
WHERE name= '$input_uname' and Password='$hashed_pwd'";
```

Task 2.1

In Task 2.1, we are asked to log in as the administrator (with a username of admin) without knowing the password. This can be done by entering **admin’; #** in the username portion. This works since the database now ignores the password field and searches only for the username, which is admin. Figure 5 shows the result of logging in as admin.

Figure 4: Logging in as admin on a webpage

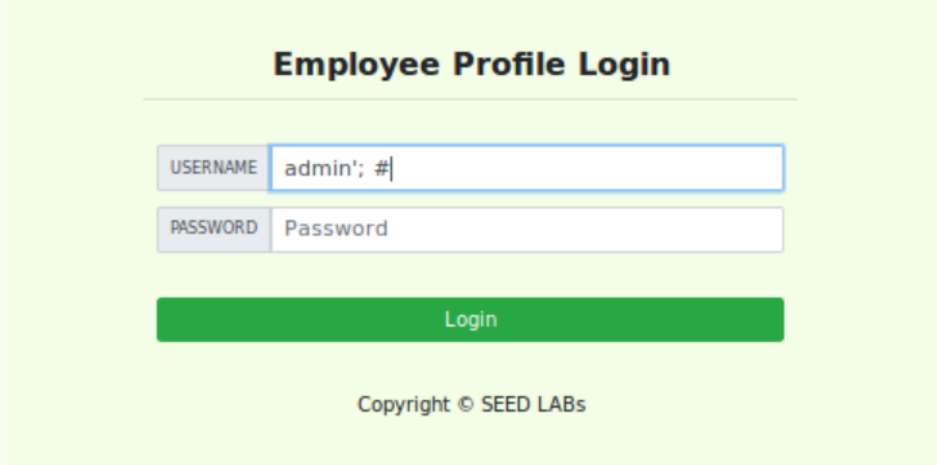
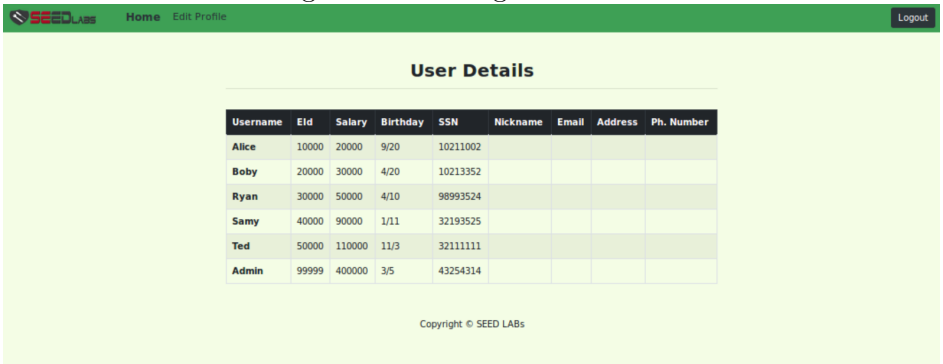


Figure 5: Showing admin result



Task 2.2

In Task 2.2, we perform the same attack as in Task 2.1 but from the command line instead of a webpage. This requires only a slight change in our attack strategy: we must now manually encode special characters such as whitespace and quotes. Figure 6 shows an execution of the command and also redirects the output to a file called task2-out.html. We can then open the task2-out.html using a browser and see what we have in the output. This is shown in Figure 7. So we have the expected output of all the users!

Figure 6: Logging in as admin from command line

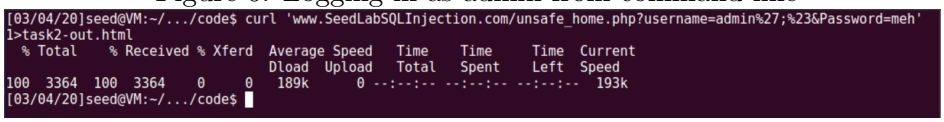
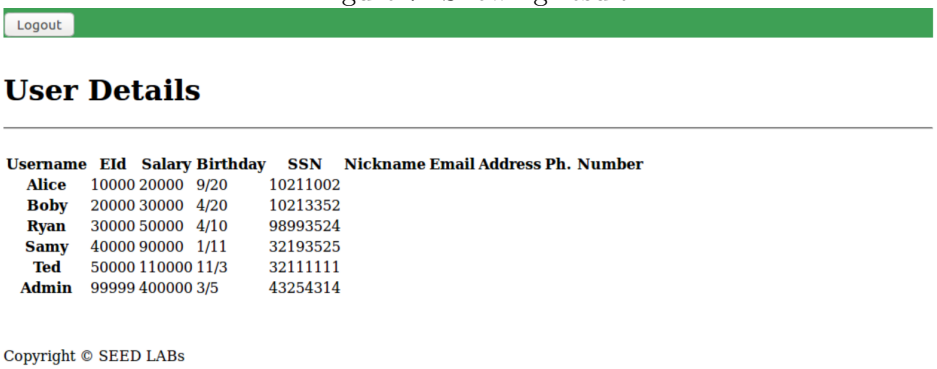


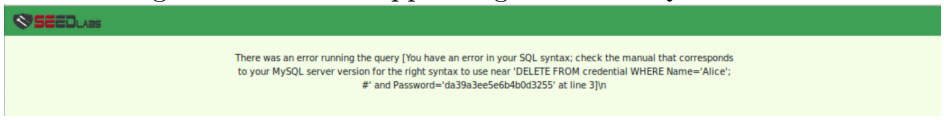
Figure 7: Showing result



Task 2.3

Task 2.3 looks to chain multiple SQL statements together. SQL statements are separated only by a semicolon, so entering **admin'; DELETE FROM credential WHERE Name='Alice'; #** as the username in the log in page would login as admin and then delete the user Alice from the data base. I chose not to display a screenshot entering this since the username field on the webpage would not have shown the entire input. However, there is one issue: the above statement did not work (Figure 8 displays the output of entering that as the username). This is because the php code uses a sql function that only allows an execution of one command. This is a step in the right direction from a protecting perspective.

Figure 8: Result of appending a second SQL statement



Task 3

Instead of just viewing information, Task 3 looks to update the data base with new values. In this task, we assume that we are a disgruntled employee named Alice. The subsequent subtasks show how we can use the SQL injection attack for malicious purposes. The attack is based on knowing the vulnerable php code in Figure 9.

Figure 9: Vulnerable php code

```
$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='$hashed_pwd',PhoneNumber='$in
put_phonenumber' where ID=$id;";
 }else{
   // if passowrd field is empty.
   $sql = "UPDATE credential SET nickname='$input_nickname',email='$input_email',address='$input_address',PhoneNumber='$input_phonenumber' where
ID=$id;";
 }
```

Task 3.1

Task 3.1 asks that we modify our own (Alice's) salary. Our first step is to log in as Alice, which we do using the technique form Task 2. Figure 10 shows the login screen and Figure 11 shows Alice's profile.

Figure 10: Logging in as Alice

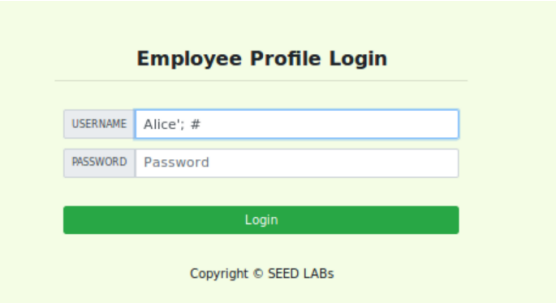
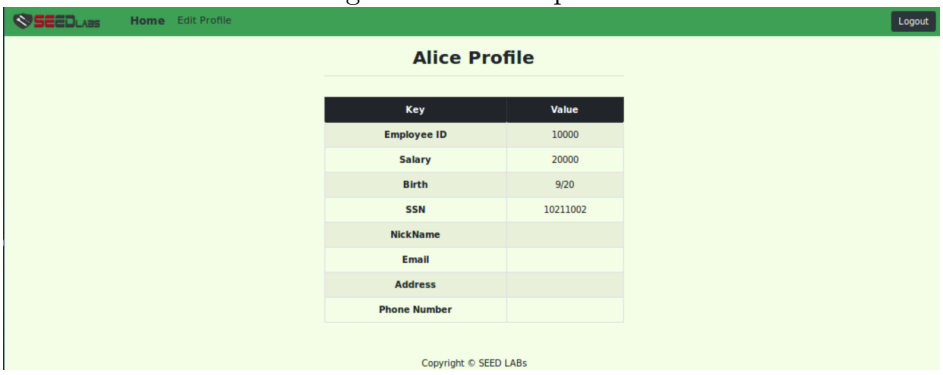


Figure 11: Alice's profile



We then navigate to the edit profile page and enter **', salary=100000 WHERE Name='Alice';** # in the nickname field. This command will change Alice's salary to 100000 and comment out the rest of the changes. The command is shown in Figure 12 and the result can be seen in Figure 13.

Figure 12: Changing Alice's salary

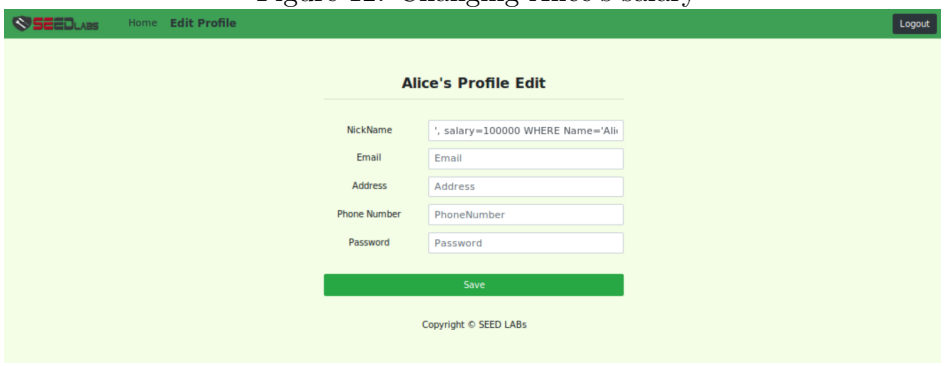
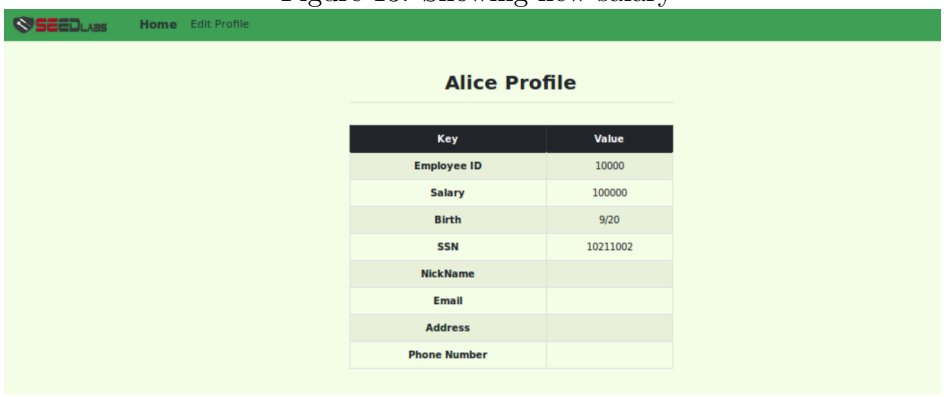


Figure 13: Showing new salary



Task 3.2

Task 3.2 asks us to modify other people's salary. We begin by again logging in as Alice and navigating to the edit profile page. Now, instead of entering **', salary=100000 WHERE Name='Alice';** # in the nickname field, we enter **', salary=1 WHERE Name='Boby';** # to change Boby's salary to $1. The input is shown in Figure 14 and the successful salary change is shown in Figure 15 (I logged in as admin to view everyone's salary).

Figure 14: Changing Boby's salary

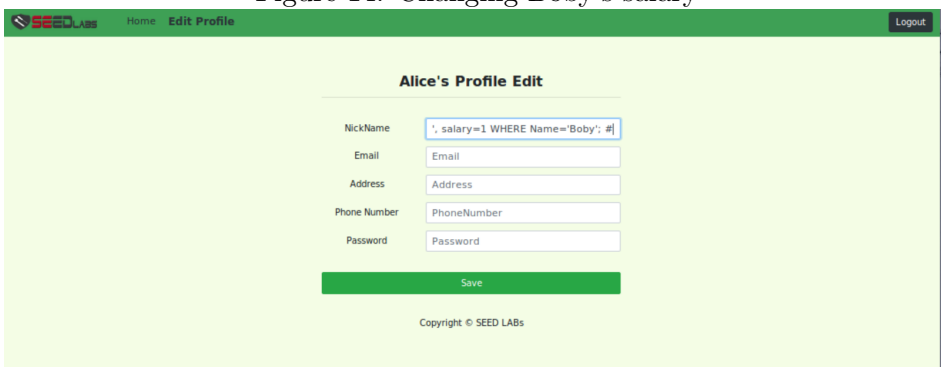
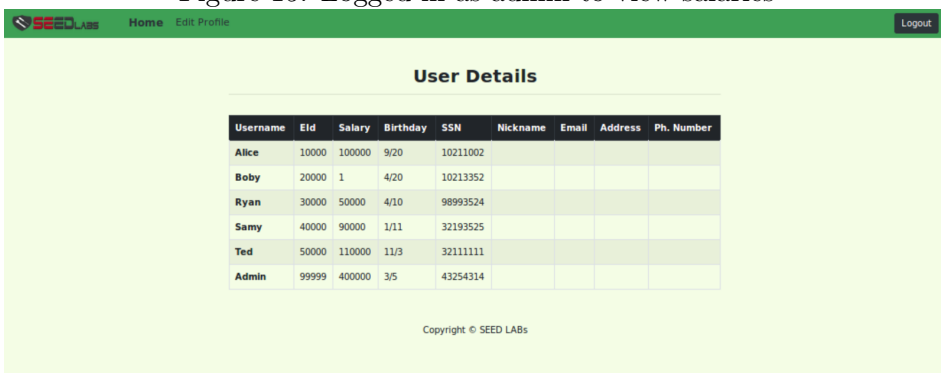


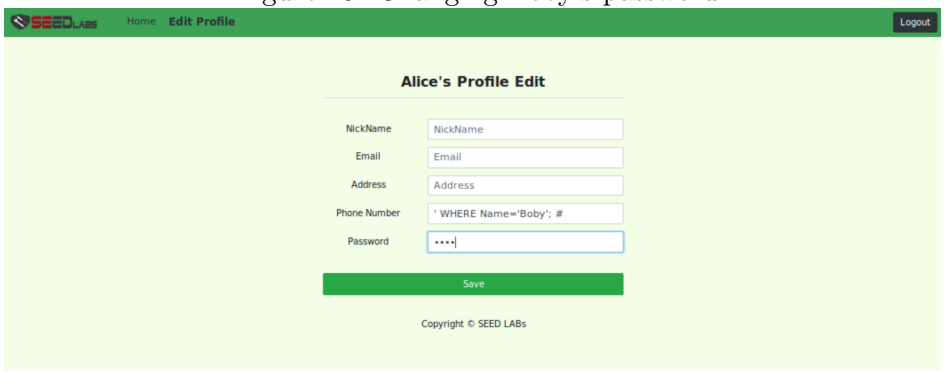
Figure 15: Logged in as admin to view salaries



Task 3.3

Task 3.3 asks that we modify Boby's password. We again do this from Alice's account on the edit profile page. However, changing a password is slightly different than updating salaries. We have to keep in mind that the password is hashed. A quick glance at the vulnerable php code in Figure 9 shows that we can enter the password that we want into the new password field and then some malicious SQL statements in the Phone Number field to change whose password gets updated. Figure 16 shows this where the SQL statement is **' WHERE Name='Boby'; #** and a new password is entered.

Figure 16: Changing Boby's password



The above information gives the error shown in Figure 17, however, Boby's password does get updated (which can be seen in Figure 19).

Figure 17: Error message

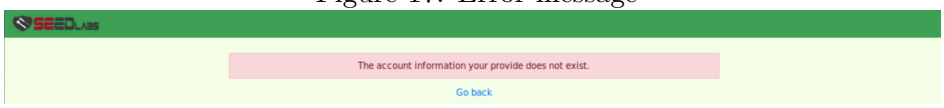


Figure 18 shows the login page for Boby and then Figure 19 shows us successfully viewing Boby's profile.

Figure 18: Logging in as Boby

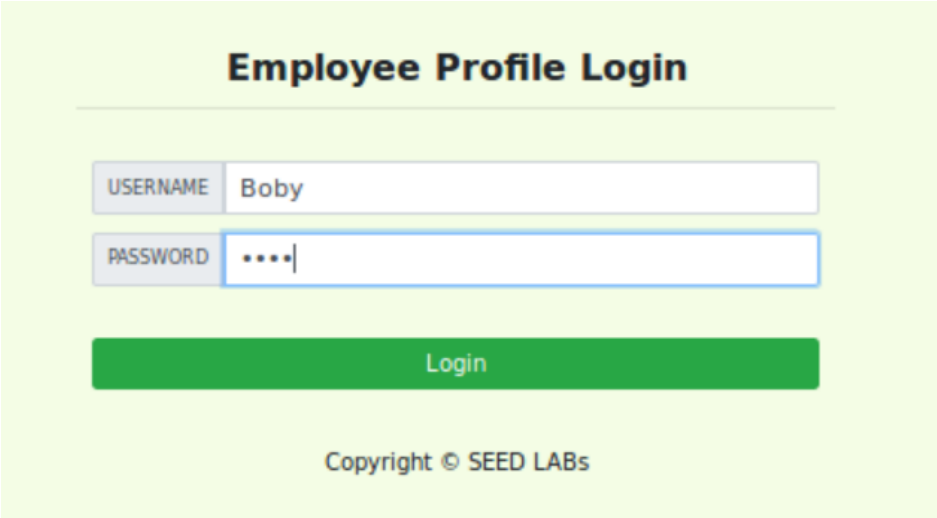
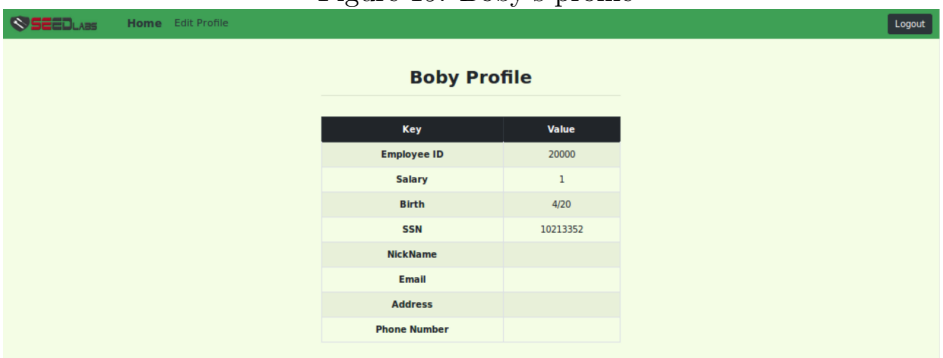


Figure 19: Boby's profile



Task 4

Task 4 asks us to edit the code to make the website no longer vulnerable by using prepared statements. We do this for each of the vulnerable portions below. Figure 20 shows the new code used for the log in page.

Figure 20: Updated login php code

```
$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 21 displays the same login string that was passed in during Task 2, however, Figure 22 shows that the attack is no longer successful and no such user is found.

Figure 21: Attempted SQL Injection

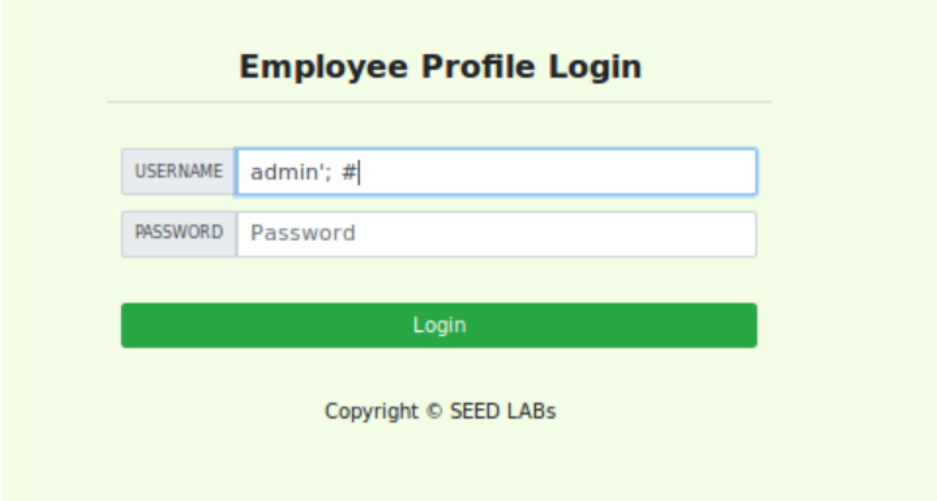
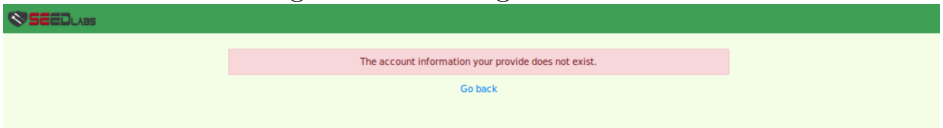


Figure 22: Showing attack failure



We now give the edited code behind updating the profile (shown in Figure 23). This, again, uses prepared statements.

Figure 23: Updated edit profile code

```
$sql="";
if($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 = $conn->prepare("UPDATE credential SET nickname= ?,email= ?,address= ?,Password= ?,PhoneNumber= ? where ID=$id;");
  $sql->bind_param("sssss",$input_nickname,$input_email,$input_address,$hashed_pwd,$input_phonenumber);
  $sql->execute();
  $sql->close();
}else{
  // if passowrd field is empty.
  $sql = $conn->prepare("UPDATE credential SET nickname=?,email=?,address=?,PhoneNumber=? where ID=$id;");
  $sql->bind_param("ssss",$input_nickname,$input_email,$input_address,$input_phonenumber);
  $sql->execute();
  $sql->close();
}
```

In Figure 24, we have logged in as Boby (since we know his password) and attempt to change his salary using the same attack described in Task 3. However, Figure 25 shows that the attack no longer works. The salary is not updated and the exact string passed in for the Nickname field is now Boby's nickname.

Figure 24: Attempted SQL Injection

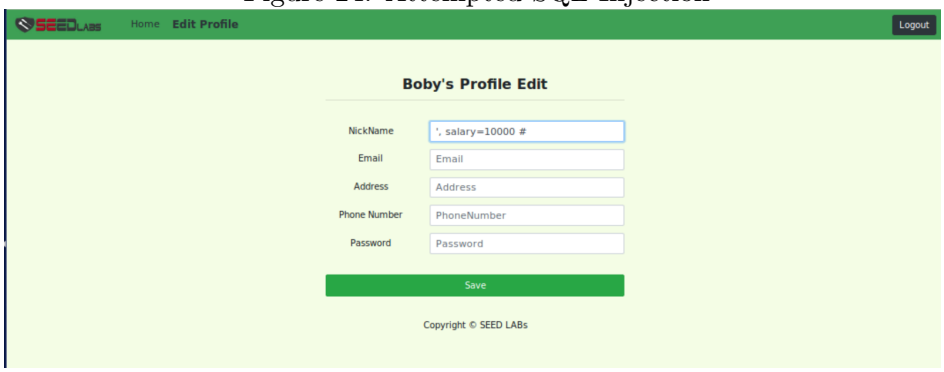
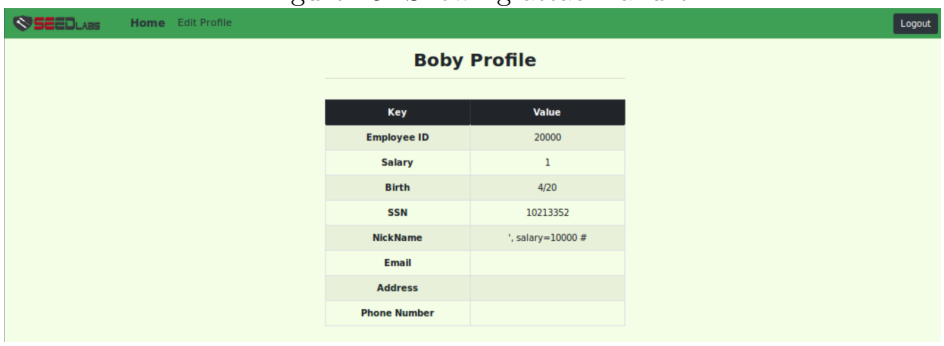


Figure 25: Showing attack failure



Overall, the SQL Injection attack is especially dangerous since an attacker does not need to know much information about the system. There is no need for figuring out what passwords are or even knowing a lot of user names (since they could try variations of admin). However, good software engineering standards, such as separating code and data, provide safe countermeasures against such attacks