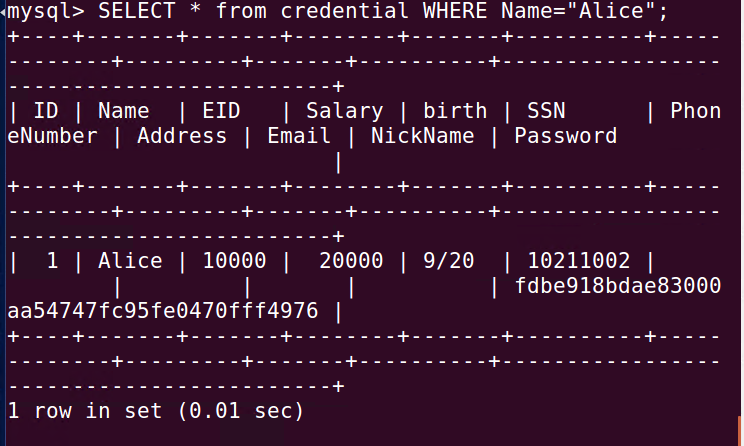
**CPS 633 - Lab 4 Report**

Section 4 Group 10

**Task 1: Get Familiar with SQL Statements**

We use a simple SELECT statement that specifies Name=’Alice’ to get information from the table about Alice.

**SELECT \* FROM credential WHERE Name=“Alice”;**

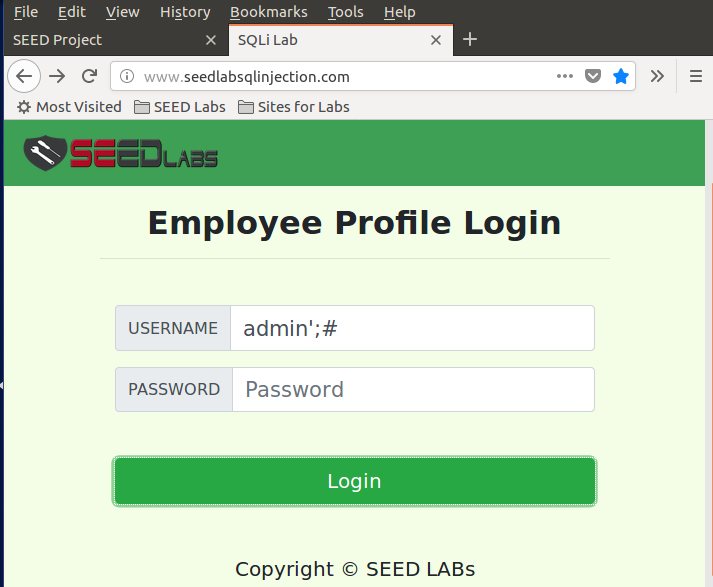


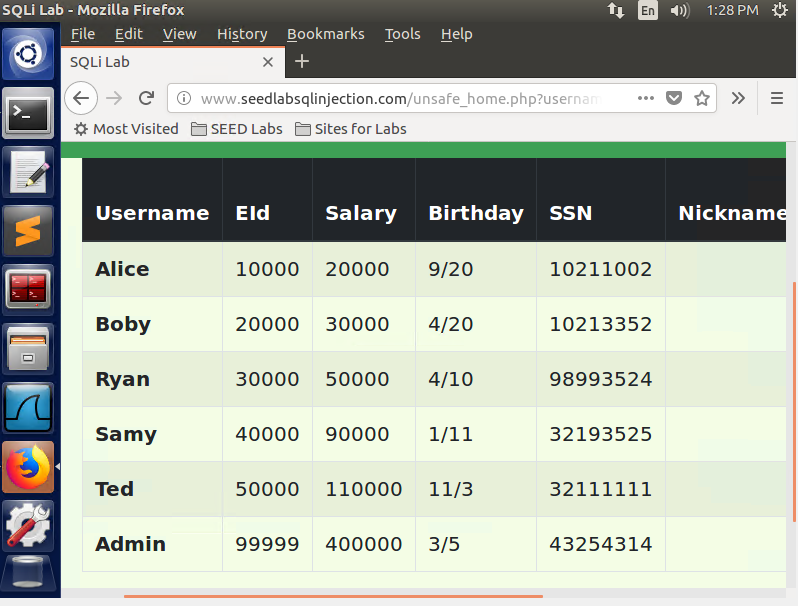
**Task 2: SQL Injection Attack on SELECT Statement**

**Task 2.1: SQL Injection Attack from webpage**

After inspecting the SELECT statement that is implemented in unsafe\_home.php, we can see that all we need to do to gain access to the system is to remove the last part of the query where it checks for the password. So to do this, we close off **admin** with a **‘** (single quotation mark)so that it is recognized as part of the query. Now that it is part of the query, we end the query with **;** (semicolon). Then finally to get rid of the rest of the query and avoid an error, we simply append **#** to the end of our statement to comment out the rest of the code in that line. This works because after we end the query with the semicolon we are not back to typing in PHP, therefore because # comments out code in PHP it works.

Our login will therefore look like this: **admin’; #**

****



**Task 2.2: SQL Injection Attack from command line**

To login using CURL we essentially execute the same admin’;# command into the Username login box, but using curl we need to replace the special characters with the encoding characters so Shell doesn't get confused.

%27 = ‘ (single quotation mark)

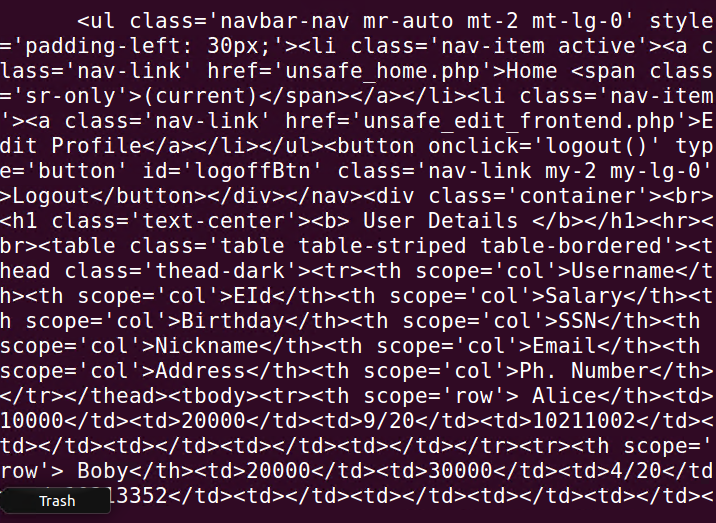
%23 = #

So our CURL command will look just like this:

**curl 'www.SeedLabSQLInjection.com/unsafe\_home.php?username=admin%27+;+%23'**

****

As we can see below, we have access to the Admin page. It is a little hard to read through all the HTML but we can see it displays information for all the Users which indicates we have Admin access.

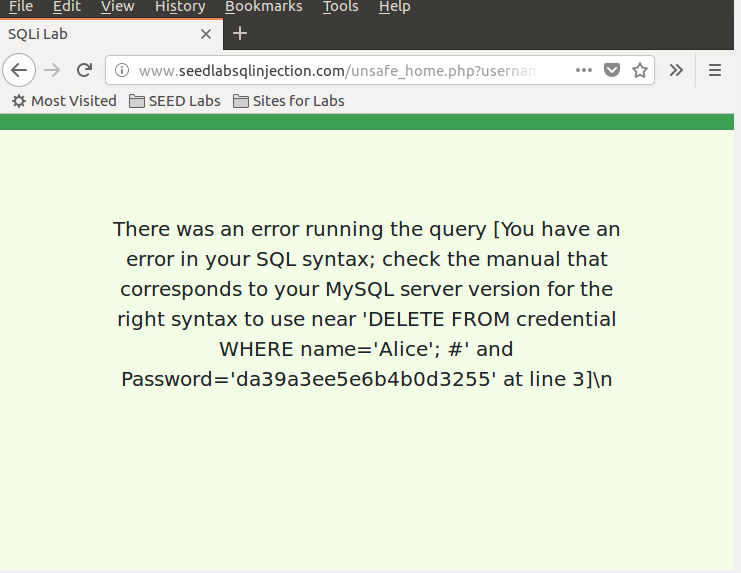
****

**Task 2.3: Append a new SQL statement**

Unfortunately, query stacking is not possible in MySQL PHP. It blocks the second query after the SELECT statement to be less vulnerable to SQL Injection Attacks. Using MySQL Connect does not allow multiple queries unless a flag is passed in the initial setup.

In theory, the command below should delete the user Alice from the database. It follows the same idea in Task 2.1, where we use the single quote to force the system to recognize it as part of the query. Then, proceeds to end the query and in theory execute the DELETE query that follows. Then at the end of the DELETE query we end it using the semicolon and append the # to the end to comment out the rest of the query that requires the password.

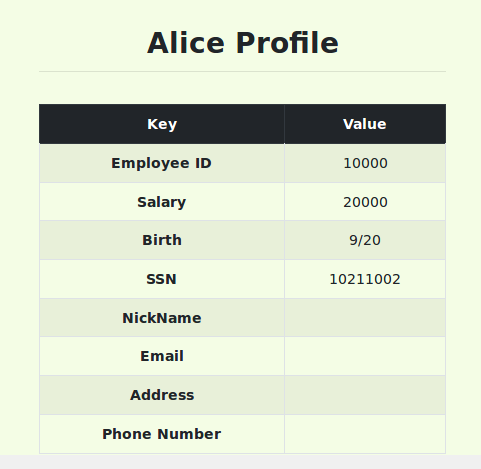
**admin’; DELETE FROM credential WHERE Name=‘Alice’; #**



**Task 3: SQL Injection Attack on UPDATE Statement**

**Task 3.1: Modify your own salary**

First we login to Alice’s account and look at it before the UPDATE statement.



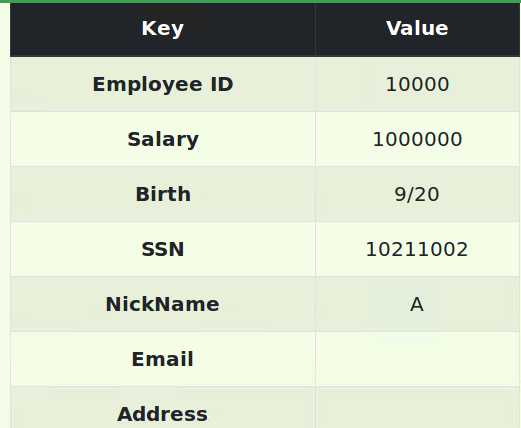
On the Edit Profile page we follow a similar idea to Task 2 where we trick the PHP program into thinking the input in the Nickname box is part of the SQL Query, then we continue it and make the changes we want.

In this case we want to change our salary to 100000, so we input the following command into the Nickname input box:

**A',salary=1000000;#**

****

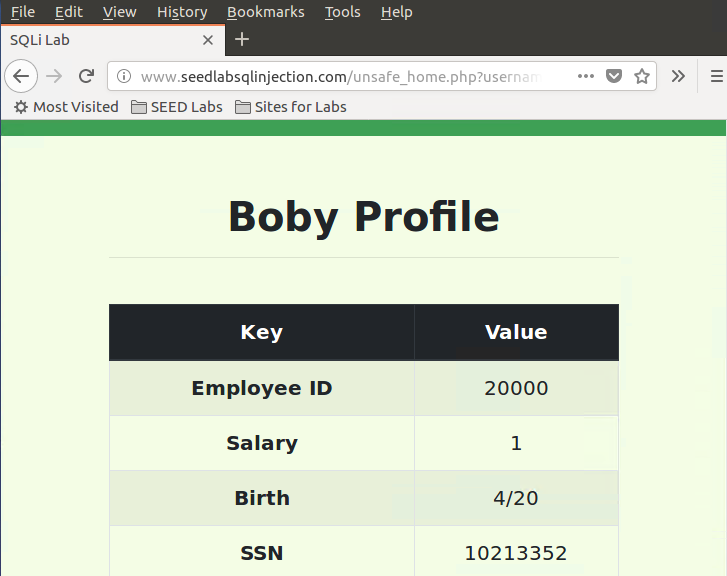
We follow the layout of the query shown in unsafe\_edit\_backend.php. After we append the single quotation mark, we continue by appending a comma that is used to separate the update statements after SET, so we attach **salary=1000000** after the comma. Then close the query with a semicolon and then append the # to the end which comments out the rest of the query.



**Task 3.2: Modify other people salary**

Again we follow the same format we did in the previous task, but this time we add a specification after salary=1 that only updates for users with Name=’Boby’ who in this case is our victim.

**',salary=1 WHERE Name='Boby';#**

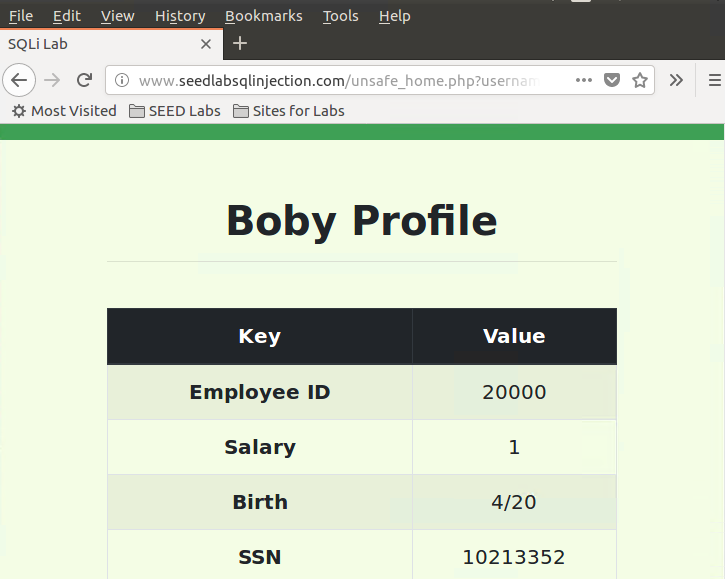
****

**Task 3.3: Modify other people password**

In this case, for us to assign a password to a user it must be encrypted in SHA1. So we went online and used an online SHA1 encryption tool to encrypt the password “123” and assigned it to Boby.

**A’,Password=40bd001563085fc35165329ea1ff5c5ecbdbbeef WHERE Name=’Boby;#**

We are able to login into Boby’s account using the password 123.



**Task 4: Countermeasure Prepared Statement**

In the /var/www/SQLInjection directory we find the file safe\_home.php that implements prepare statements that area used to counter SQL Injection attacks. We also find a safe\_edit\_backend.php file that also uses prepare statements for the safe version of the edit page. We then attempt the same SQL Injection Attack on the safe home page that we did in Task 2 to try and login as the Admin without using any passwords and it fails.

safe\_home.php (Just the implementation of the prepare statements):

// create a connection

$conn = getDB();

// Sql query to authenticate the user

$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();

safe\_edit\_backend (Just the implementation of the prepare statements):

$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();