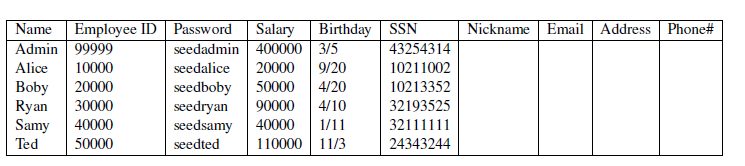
**ASSIGNMENT – 9**

Name: **Sudharsan Srinivasan**

UTA ID: **1001755919**

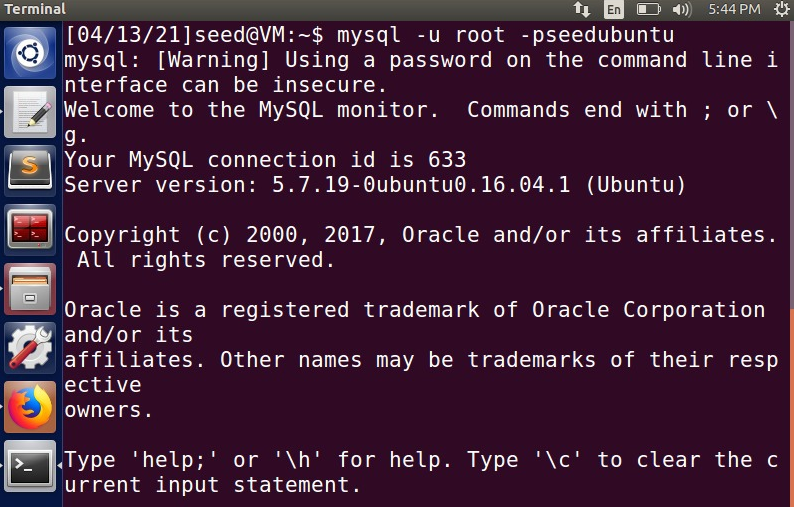
**Environment:**

* The websites used for the tasks are [**www.SEEDLabSQLInjection.com**](http://www.SEEDLabSQLInjection.com)to login to user accounts etc and the corresponding folder accessed is **var/www/SQLInjection.**
* The following image shows the employee database which will be used for the tasks in the assignment to carry out SQL activities.

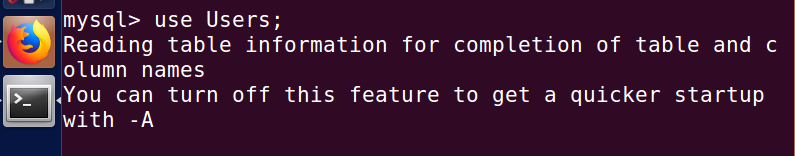


**Task 1 – Familiarizing with SQL Statements:**

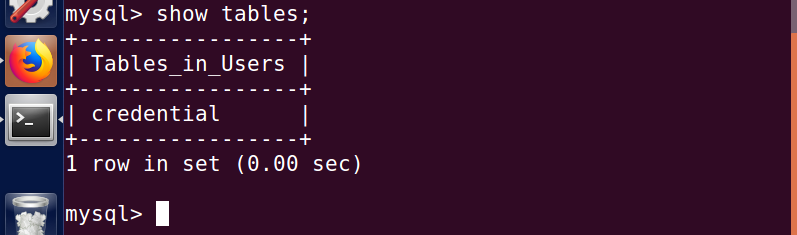
* This is just to get used to the SQL Database that is shown above. We first login to the database using **root** as username and **seedubuntu** as password using the below command.



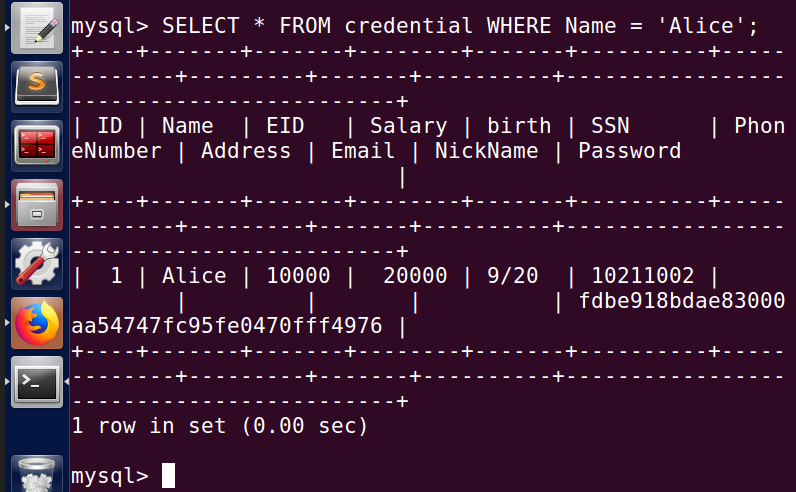
* Next, we load the existing database we saw above using the **use** command.



* To print all the tables that are present in the above database, we use **show tables** command.



* The above result shows we have a single table named **credential.** To pull up the information of employee ‘Alice’, we use SELECT statement.



* As seen above, the employee ‘Alice’ information gets printed on the terminal.

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

* We try to launch SQL Injection Attack in this statement by 3 different methods.

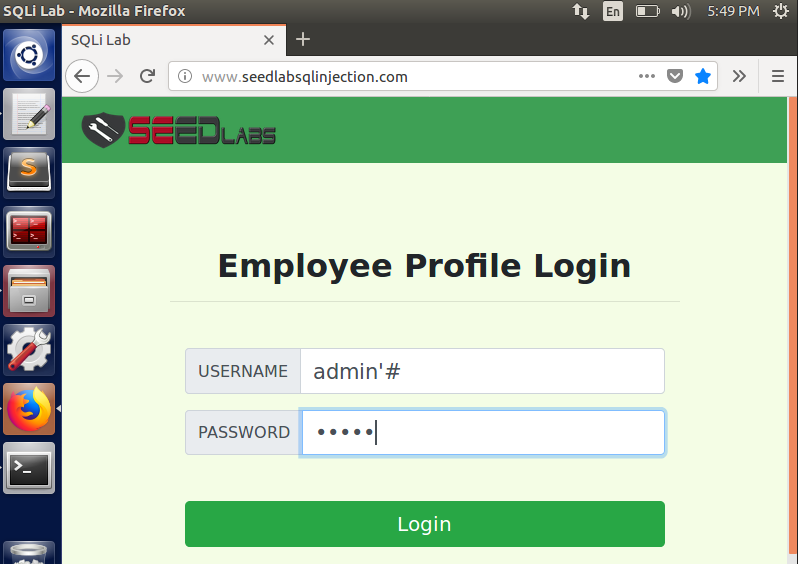
**SQL Injection from Web Page**

**SQL Injection from Command Line**

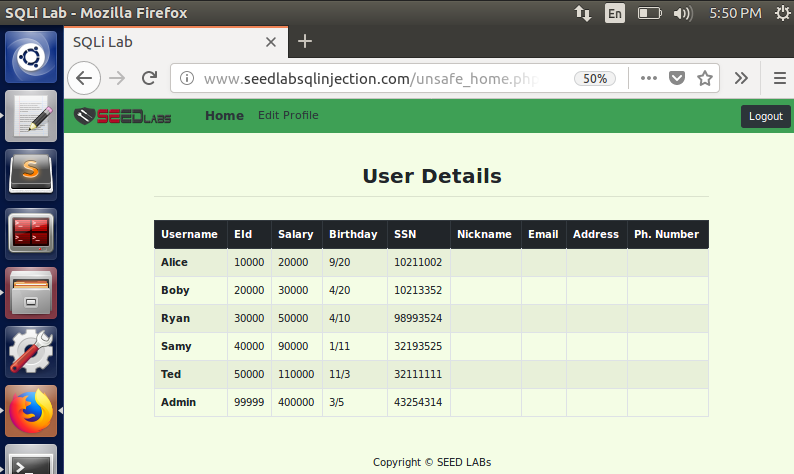
**Append SQL Statement**

**2.1 – Web Page SQL Injection:**

* Open the **seedlabsqlinjection.com** site in the browser. On the Employee Profile Login page, we try login as the **admin** and we enter some Password because we don’t know it.



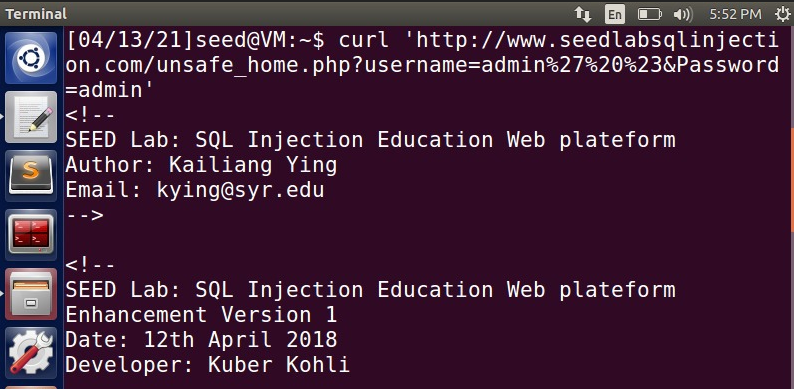
* After entering the above credentials, we see below that we were able to login to the database.



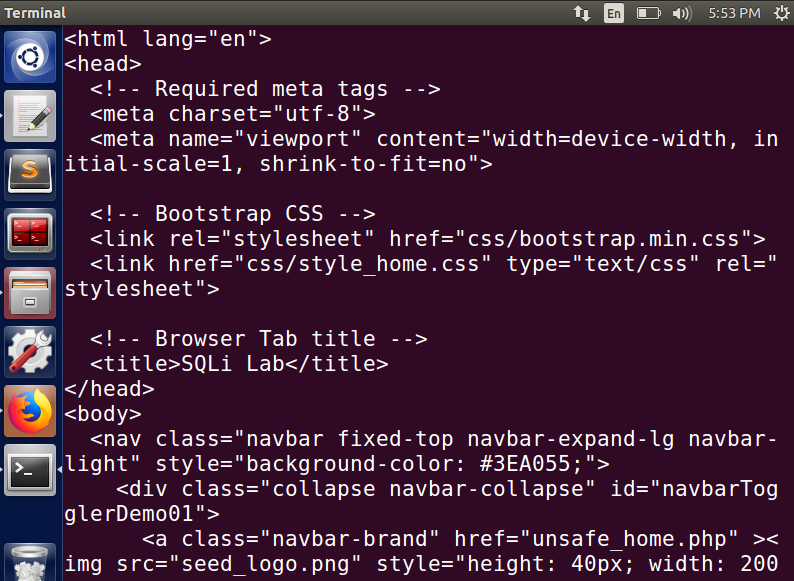
* We observe the login credential we used above – ‘**admin’#**’
* Since we don’t know the password, we enter # in the username column. This # makes every field after username to be commented out and does not check for column such as Password. So, it does not matter what value we enter in the Password field since its never going to be checked with the database.
* Some value is entered in the Password field just as a filler so that JavaScript error does not popup, which will halt our SQL Injection Attack.
* Thus, we get the information of all the employees as admin even without the password by launching SQL Injection attack through Web Page.

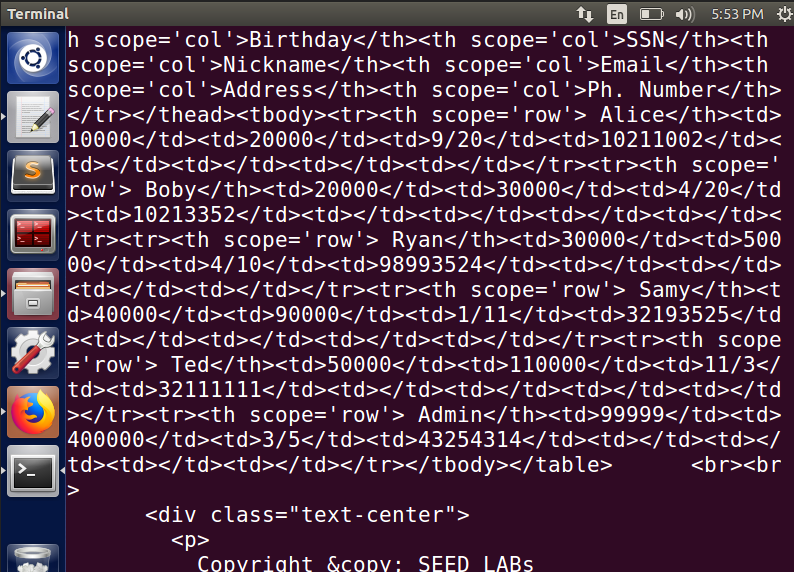
**2.2 – Command Line SQL Injection:**

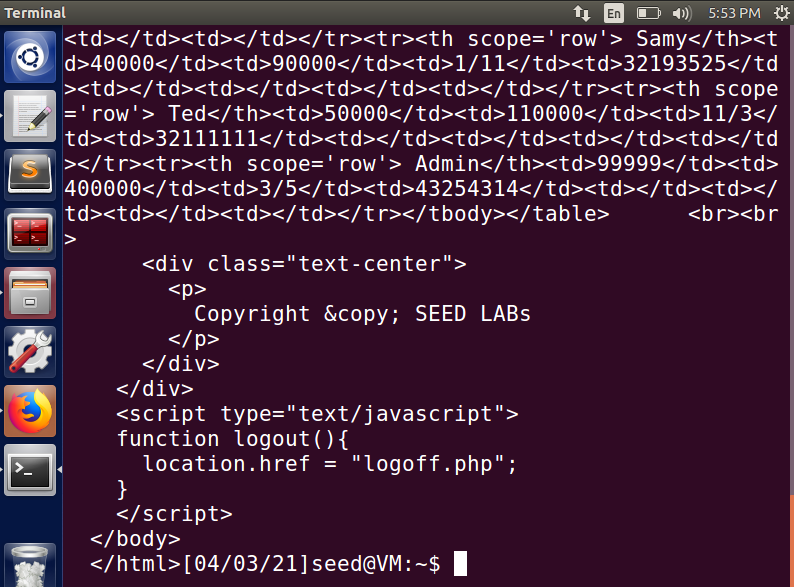
* We perform the same login activity, this time using Command Line through **curl** command.
* We use **curl** command along with specifying the website to access, in our case seedlabsqlinjection and the file **unsafe\_home.php** which checks for the username with the database, followed by the credentials for username and password.



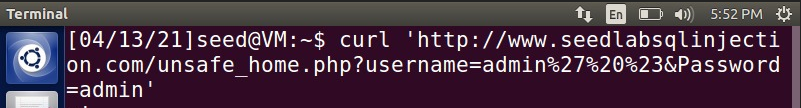
* After giving the above command, we get access to the database and all the individuals employee information is printed out in command line in table format as seen.







* This is because of the command we entered using **curl**



* Notice for the username, we gave **admin%27%20%23**

Here, %27 is the ASCII equivalent of **‘**

%20 is the ASCII equivalent of **space**

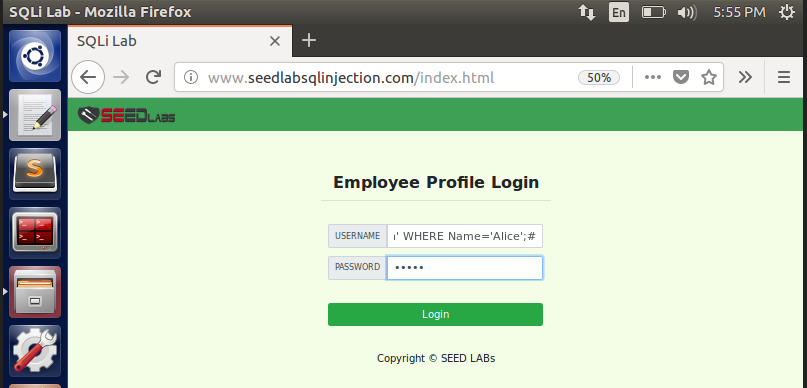
%23 is the ASCII equivalent of **#**

Which makes the username admin’ # (similar to 2.1)

thus, enabling us to bypass Password field without the check and getting all the employee information retrieved.

**2.3 – SQL Statement Append:**

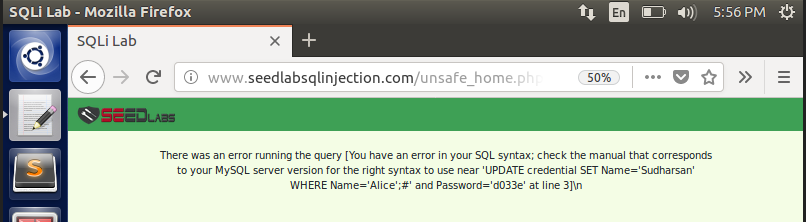
* Here, we try to append an additional SQL Statement along with the value we give for username, thus making the command run two SQL statements.



* As seen above, in the **username** field of the login page, We enter the below in username field where DELETE query is separated from admin using semicolon(;) making it 2 SQL statements, one is username check and other UPDATE query.

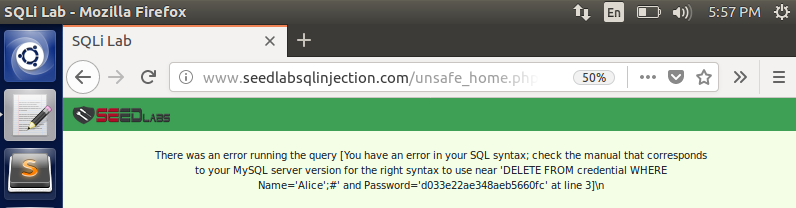
**admin'; UPDATE credential SET Name = 'Sudharsan' WHERE Name = 'Alice'; #**

* On trying to run the command, we get the following error.

****

* Next, we try to see if we are able to delete an employee information from the database using the **DELETE** command.We enter the below in username field where DELETE query is separated from admin using semicolon(;) making it 2 SQL statements, one is username check and other DELETE query.

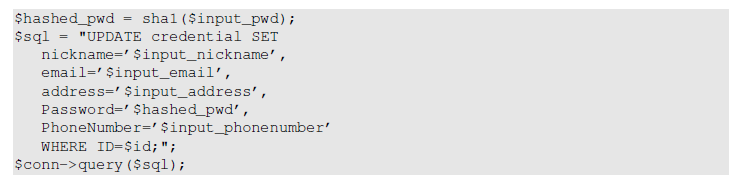
**admin'; DELETE FROM credential WHERE Name = 'Alice'; #**

****

* On both the above cases, we were unsuccessful in trying to UPDATE/DELETE employee information. This is because the MySQL PHP does not provide for running 2 SQL Statements together in the database using **query()** function we have in **unsafe\_home.php** file.
* This prevents SQL Injection from happening by limiting only one SQL Query to run. To overcome this and launch a successful attack, the function should be **multiquery()** to enable it to run multiple queries simultaneously.

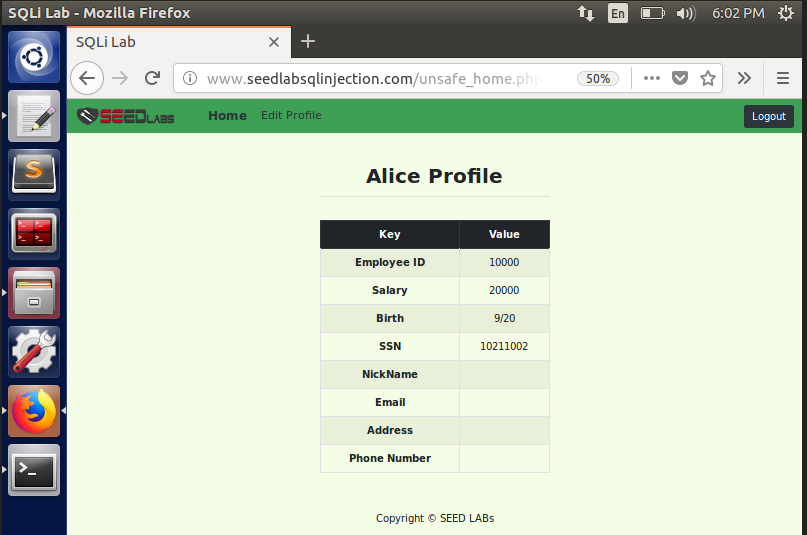
**Task 3 – UPDATE Statement SQL Injection Attack:**

* This involves using UPDATE statement to launch a SQL Injection attack to make modifications to the entries of the employees in the database.
* This process makes use of **unsafe\_edit\_backend.php** file to run the UPDATE query and change the employee information.

****

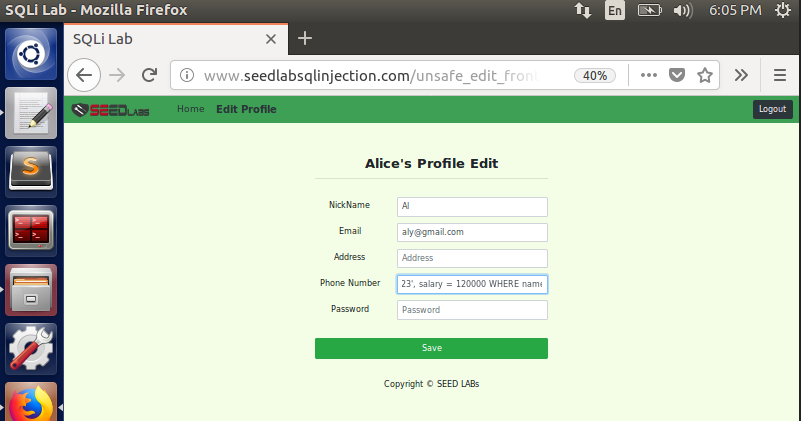
**3.1 – Modification to own’s salary:**

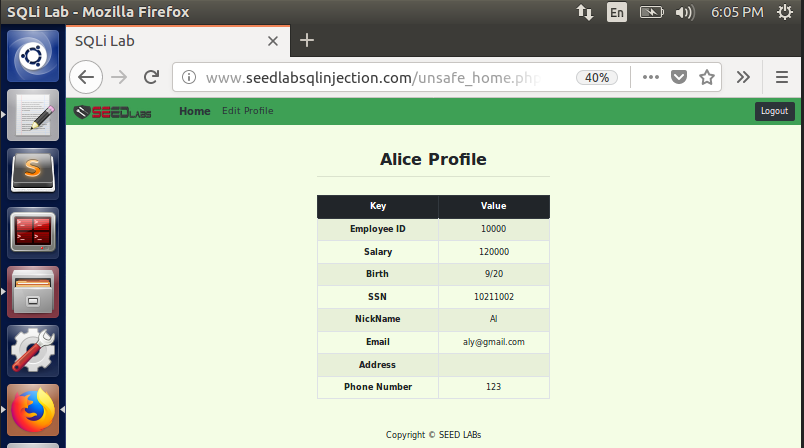
* We launch an SQL attack using the Phone Number field. First we login to our own profile, in this case, Alice account.



* As seen above, right now the salary is **20000**. Launching SQL Attack can be done now by clicking on **Edit Profile** and entering the following command in Phone Number field.

**123’, salary = 120000 WHERE name = ‘Alice’ #**

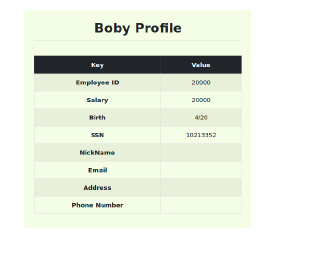




* On saving the changes, we see the query has been successful in updating the Salary from **20000** to  **120000**. As seen earlier with the snippet of unsafe\_edit\_backend.php file, the salary column gets updated with the value we specify, thus successful in launching the attack.

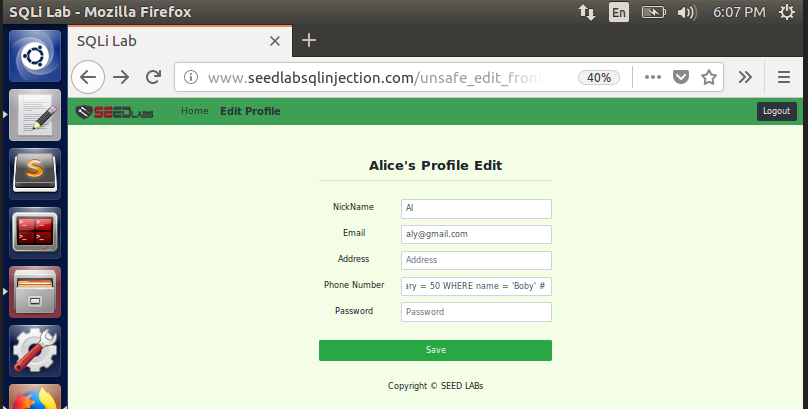
**3.2 – Modification to other’s salary:**

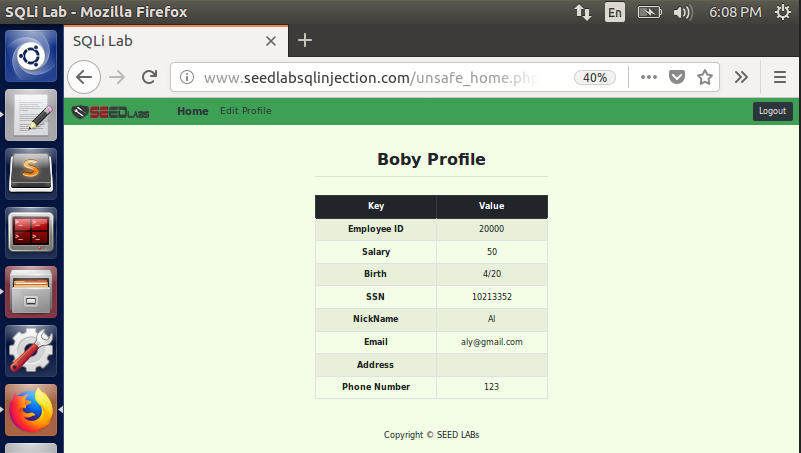
* We use the same attack to make changes to another employee’s salary, in this case **Boby**. Boby’s profile before launching the attack is shown below.



* We try to update Boby’s salary by using the below SQL command in the Phone Number field just like previous task.

**123', salary = 50 WHERE name = 'Boby' #**

****

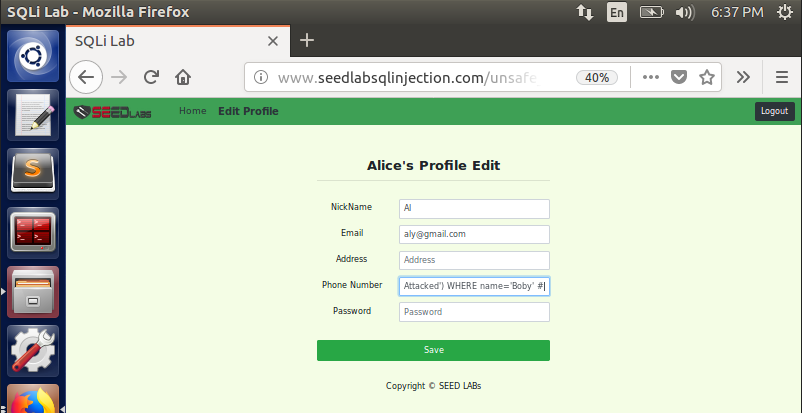
****

* We are successful in changing Boby’s salary by launching SQL attack from Phone Number field.

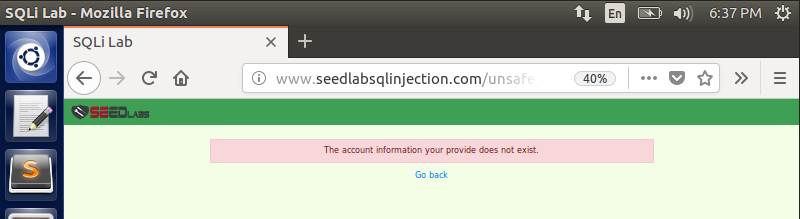
**3.3 – Modification to Password:**

* To make things worse for Boby, we go one step further and update his password as well, so that he won’t be able to login to his account.
* We use the Phone Number field and enter the below SQL command

**', Password = sha1('Attacked') WHERE name= 'Boby' #**



* After making the changes, we check to see if the attack is successful by trying to login to Boby’s account with his old password because he has no idea about the SQL attack.



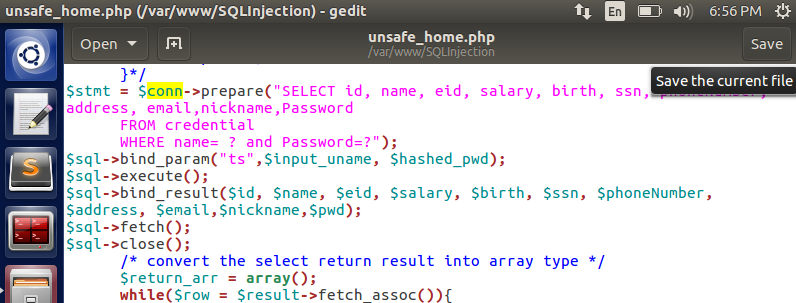
* On trying to login to the account, Boby could not login because the password has been changed, thus proving a successful SQL attack.

**Task 4 – Prepare Statement to serve as CounterMeasure:**

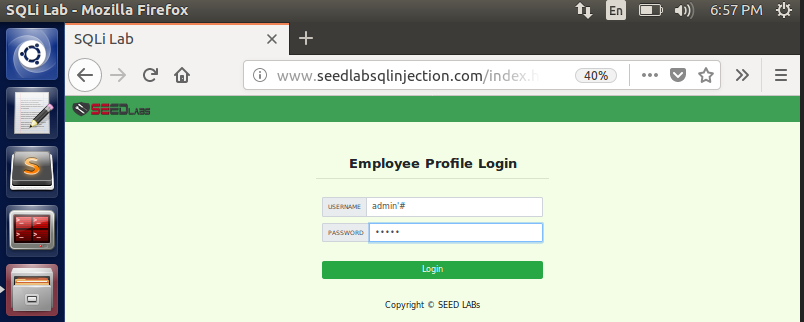
* The problem with the previous attacks is that the program cannot differentiate code and date when SQL query is sent to the database.
* To overcome this issue and to prevent SQL Injection attacks, we make use of prepare statements to send the code part separately and then bind the data part, send them separately making it easy for the program to identify them.

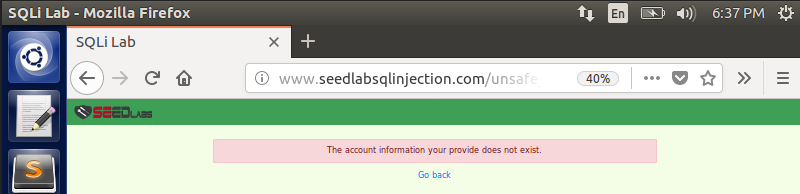
**Task 2.1 with code changes:**

* We first edit the **unsafe\_home.php** file as shown below.



* We try to perform Task 2.1 now again where we try to login without knowing the password by using SQL Injection attack.

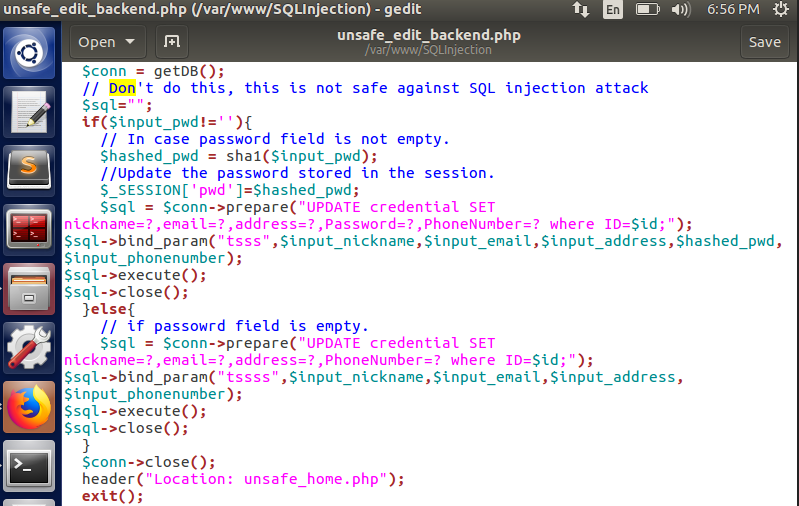




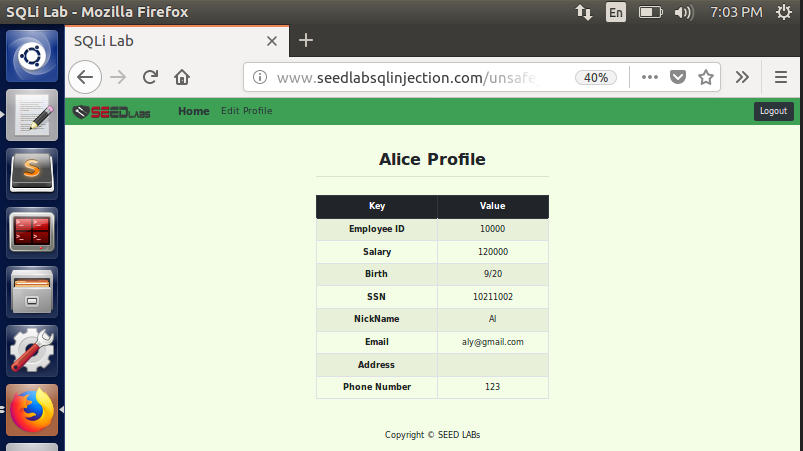
* We are not able to bypass the password field by giving # in the username field as we did in Task 2.1. This is because, since we send code and data separately in this case, the username **admin’#**  and password **admin** is sent to the database. Since the username and password does not match with any records in the database, login fails. The countermeasure to separate code and data prevents SQL injection in this case.

**Task 3.1 with code changes:**

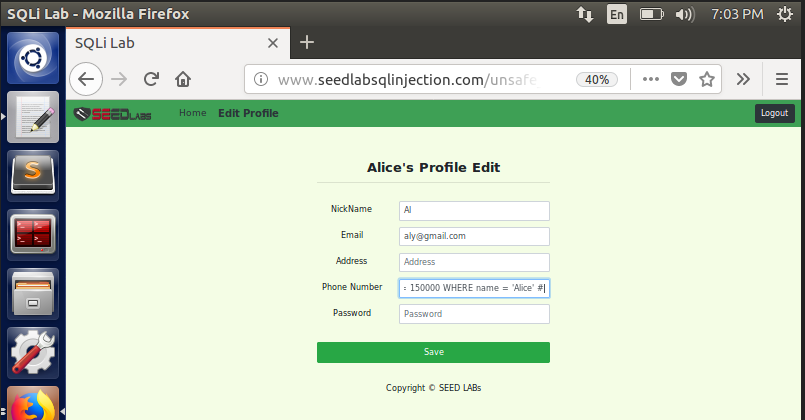
* We update the code in the file **unsafe\_edit\_backend.php** file which is used for UPDATE queries to update Alice’s salary.

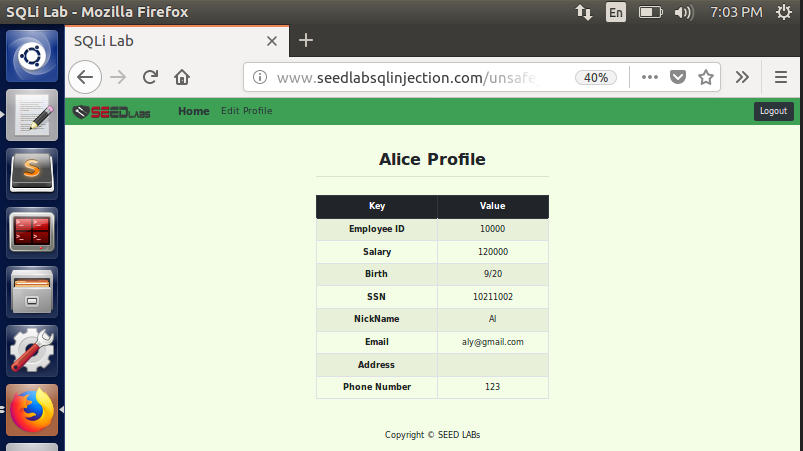


* On performing the Task 3.1 where Alice tries to update her salary, we try to perform the same attack. Alice account before trying to update.



* Now we use the same SQL command used earlier, to update salary from **120000** to **150000**





* As seen above, the salary is still the same as before, the UPDATE query did not make any impact. This is because of the updated php file to send code and data separately. Since on the data part of Phone Number field, we append this SQL statement and try to update, the program identifies that and prevents the SQL attack from happening.