CS 201P Lab Assignment #4
Sandhya Chandramohan UCI ID: 61167891

SQL Injection Attack Lab

3.1 Task 1: Get Familiar with SQL Statements

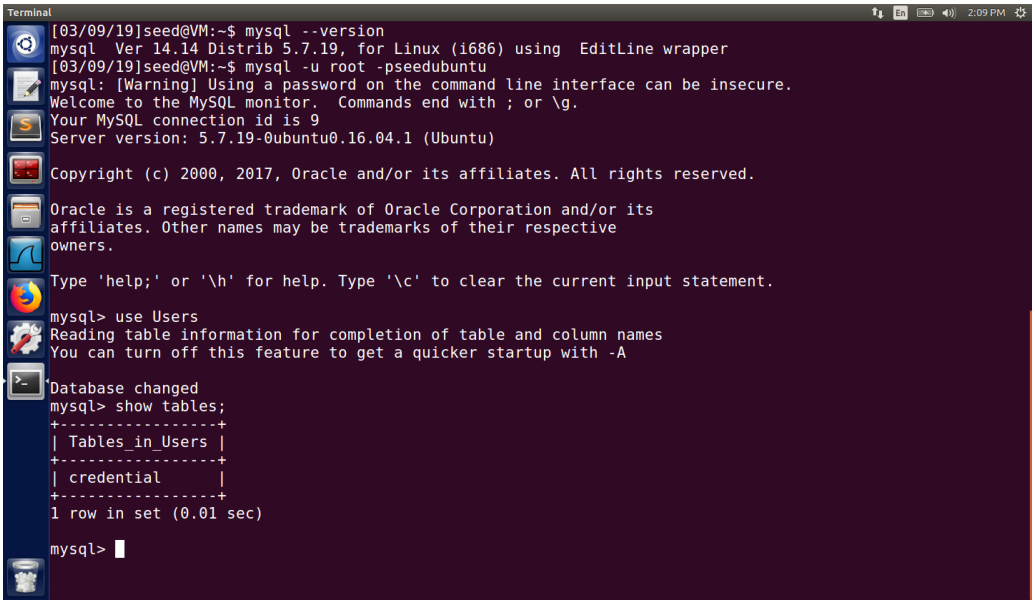
Procedure:

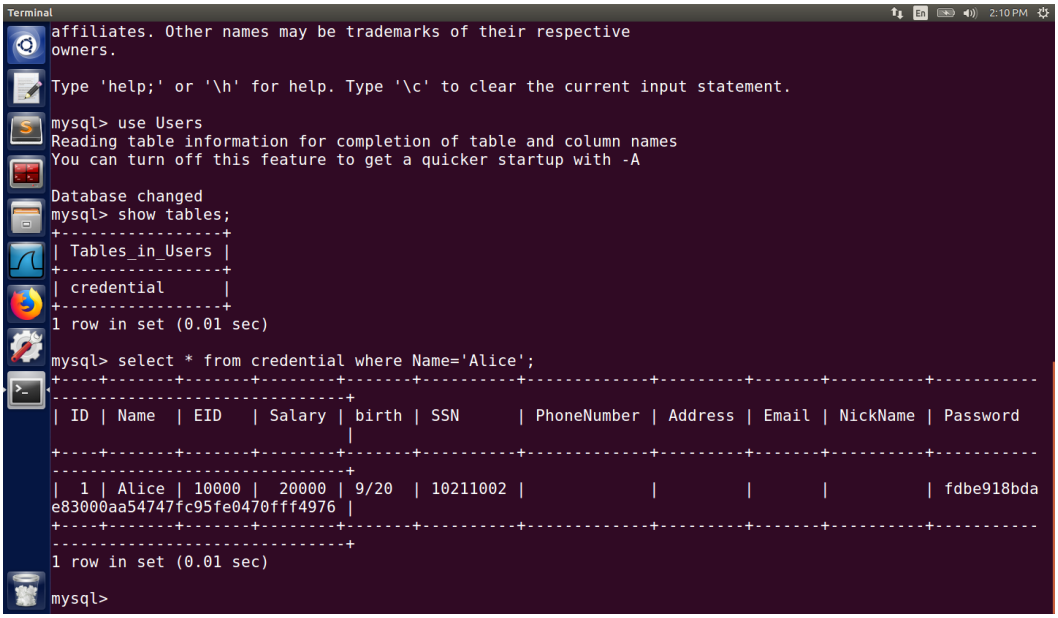
i. Login to the MySQL Console using the command:
$ mysql -u root –pseedubuntu

ii. Create or Load an existing database Users:
mysql> use Users;

iii. Show the existing tables in the Users database using the following command:
mysql> show tables;

iv. Print all the profile information of Alice using the command:
Select * from credential where Name=`Alice`;





Please Turn Over…

Observation and Explanation:

i. We are able to login to the MySQL console via the command:
   $ mysql -u root –pseedubuntu
ii. When we run the mysql> use Users; command, we are able to load the Users database
iii. Running mysql> show tables; displays the existing tables in the database
iv. By running the select statement shown below we are able to all the profile information of Alice:
   Select * from credential where Name=`Alice`;

3.2 Task 2: SQL Injection Attack on SELECT Statement

Task 3.2.1: SQL Injection Attack from webpage

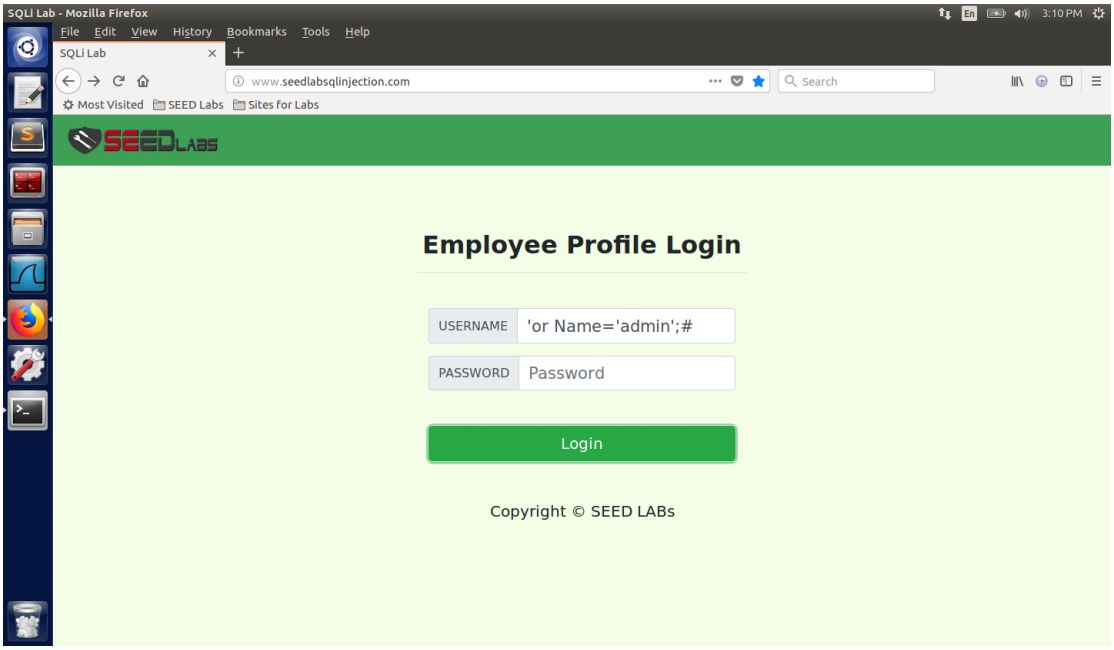
Procedure:

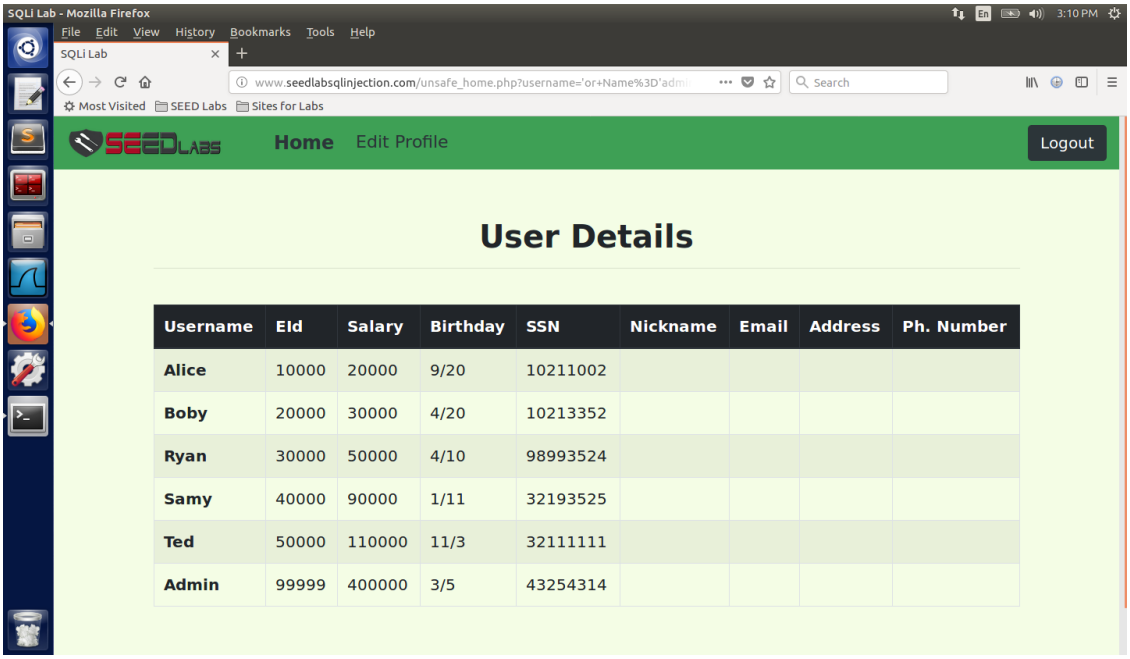
i. Knowing only the username ‘admin’ we try to login to the webapp www.SEEDLabSQLInjection.com via their Employee Profile Login Page.

ii. We type in the malicious SQL payload in the USERNAME field in the login page and leave the password field blank

iii. The mailicious payload typed in the USERNAME field is:
‘ or Name=’admin’;#

iv. We then click on submit





Please Turn Over…

Observation and Explanation:

i. When the mailicious payload is typed in the USERNAME field:
‘ or Name=’admin’;#

i. We are able to login to the web application.

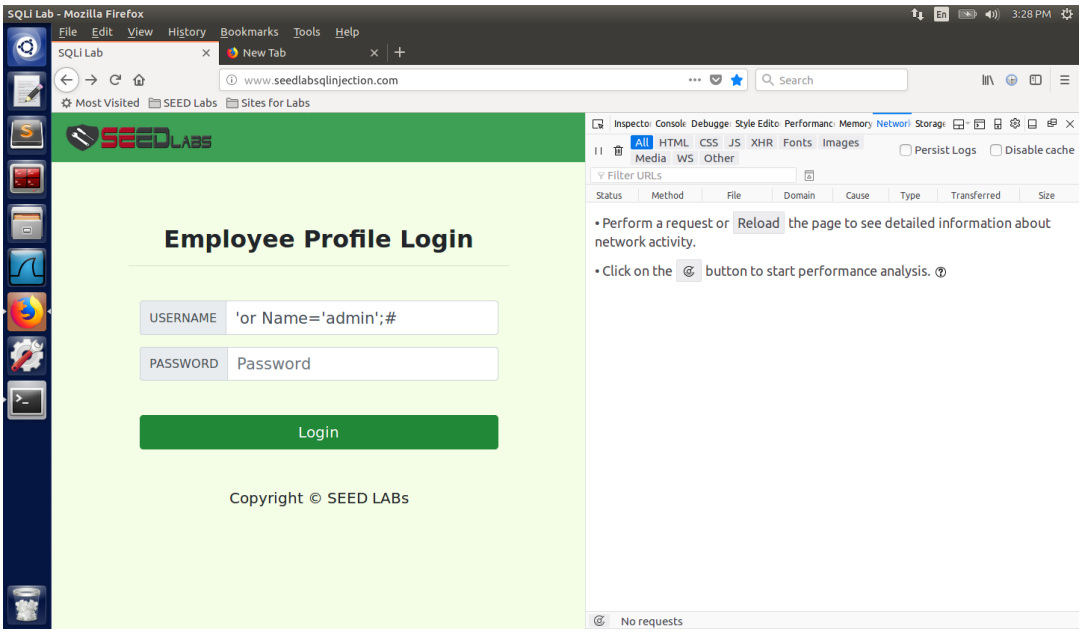
ii. This is because, the malicious payload typed in the USERNAME field gets injected in the SQL Query for fetching the login information and gets compiled, performing a SQL Injection attack.

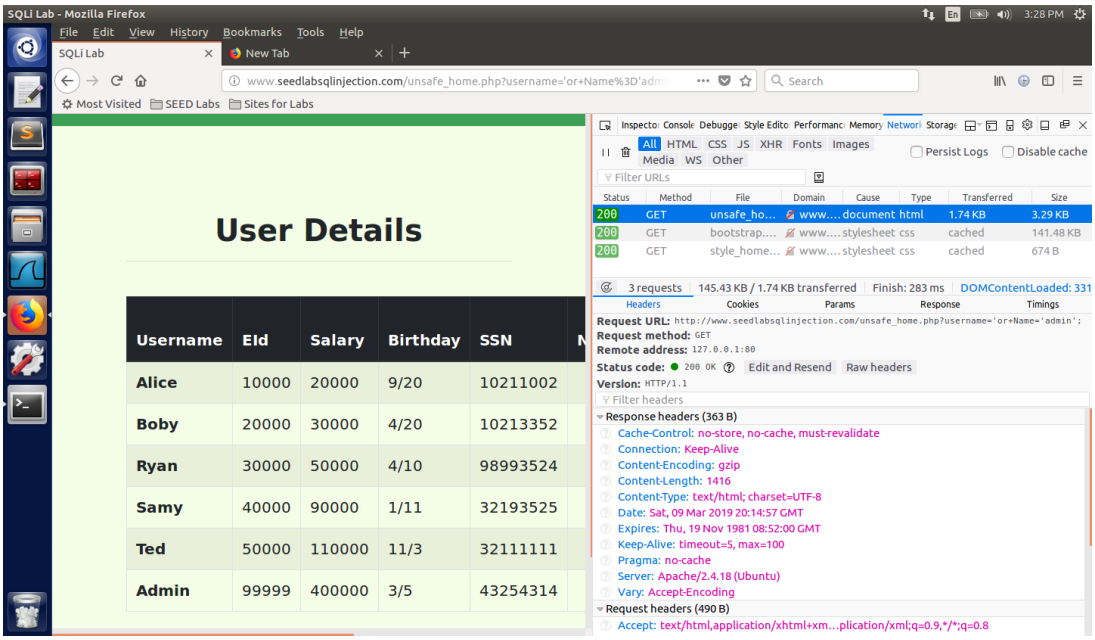
iii. The leading single quote and condition *or Name=’admin*’ leads to the query returning all fields where name is admin. And semi-colon signifies the end of the statement and the **#** at the end leads to commenting out the rest of the statement.

Task 3.2.2: SQL Injection Attack from command line

Procedure:

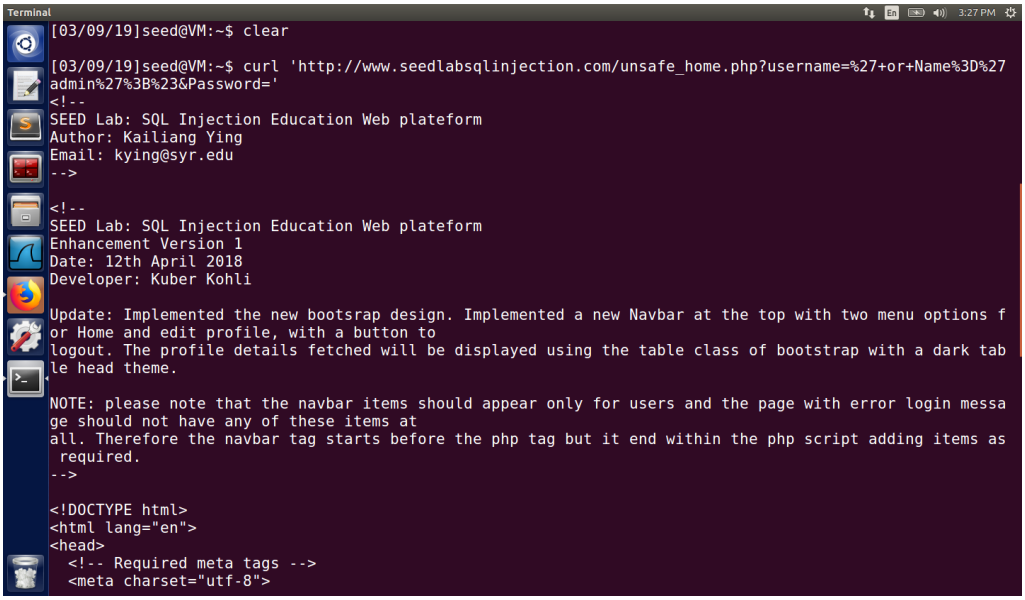
i. We first open the Developer Tools Network Tab to note the URL being used for the Profile Login, which is http://www.seedlabsqlinjection.com/unsafe_home.php?username=username&Password=password





Please Turn Over…

ii. The following command is run from the terminal command line:
Curl
'http://www.seedlabsqlinjection.com/unsafe_home.php?username=%27+or+Name%3D%27admin%27%3B%23&Password='
Which is basically the HTTP request for login with the special characters in the query parameters such as ***"#;*** being HTTP URL encoded



Observation and Explanation:

i. When the above-mentioned Curl command for HTTP request for Profile login is run from the command line, the query parameter passed to the username field is the HTTP url encoded malicious payload which results in SQL injection.

ii. The HTML of the web page after successful login is displayed on the console.

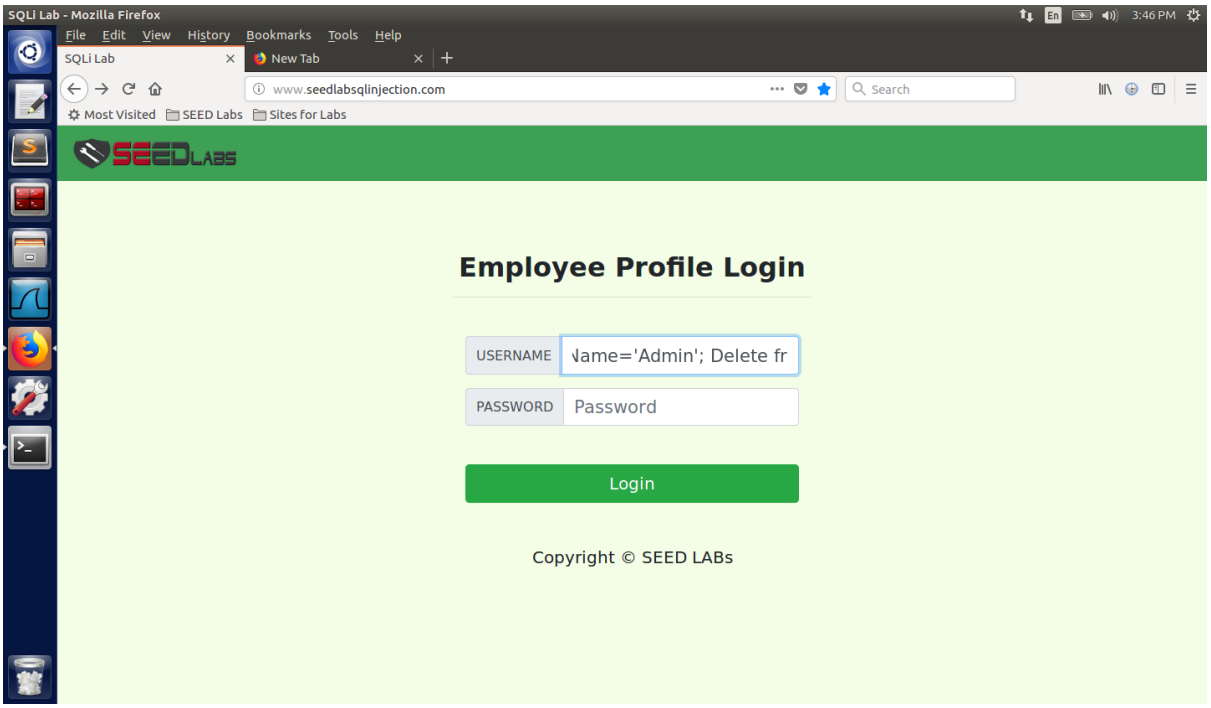
Please Turn Over…

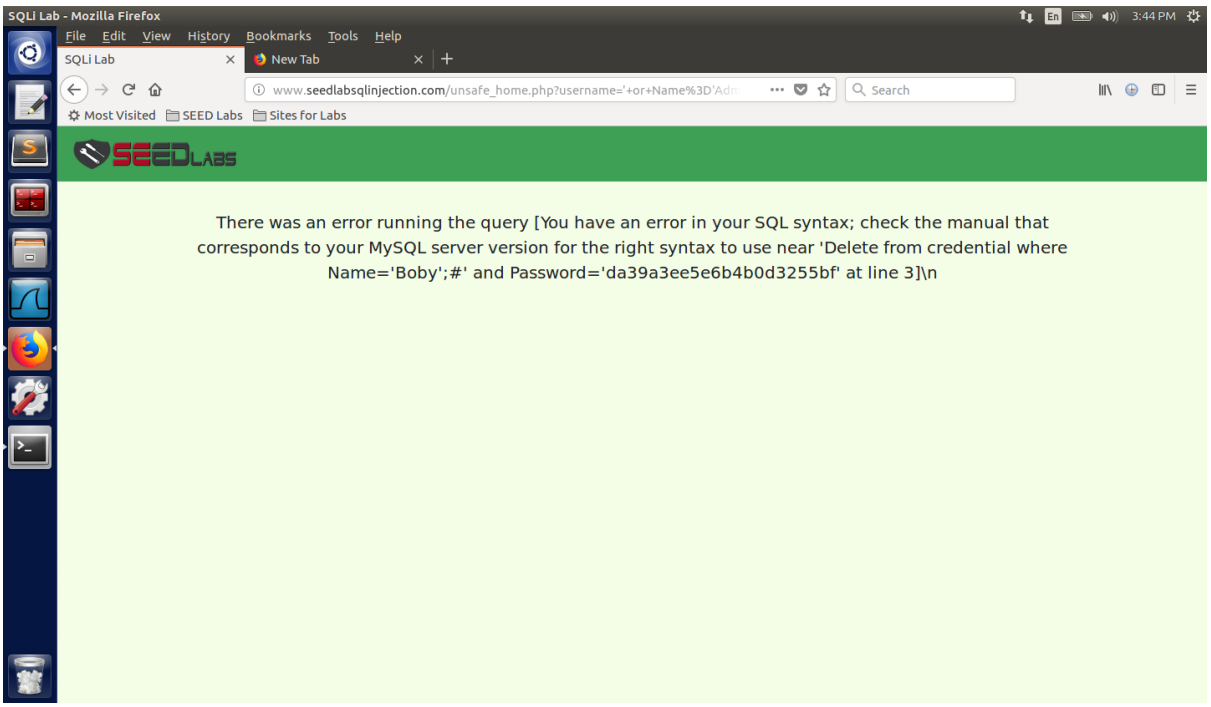
Task 3.2.3: Append a new SQL statement

Procedure:

i. In this attack, a Delete SQL statement is appended at the end of the Select SQL statement used for login.

ii. After the semi-colon which signifies the end of the first statement. The second statement: *Delete from credential where Name='Boby';#* is appended and the login button is clicked.





Please Turn Over…

Observation and Explanation:

i. We note that when a second statement is appended at the end of the first statement in the malicious payload which is injected into the SQL. It does not get executed and throws an error.

ii. Also, when the same two SQL statements are executed in the MySQL console, they execute successfully without any errors. This shows that there is no error in the SQL syntax of the two SQL statements and consequently malicious payload injected.

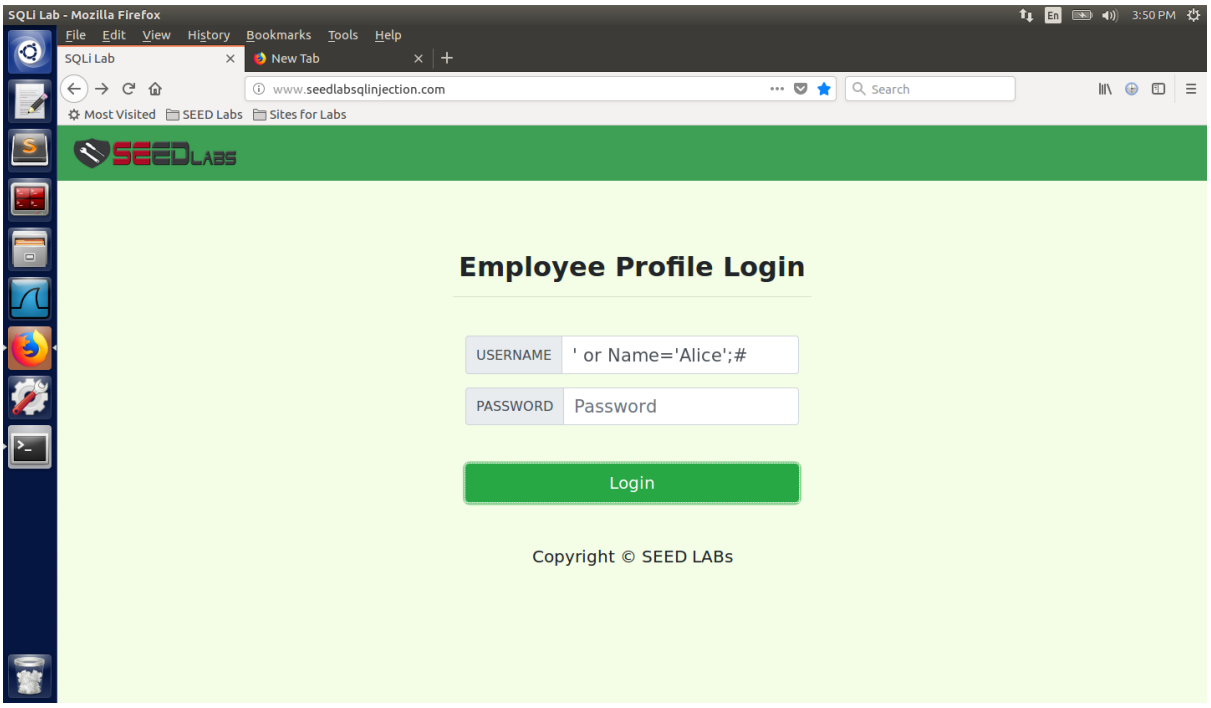
iii. The SQL injection of two SQL statements appended fails because, PHP does not allow the execution of two SQL statements at once. Therefore, it throws an error at the end of the first SQL statement.

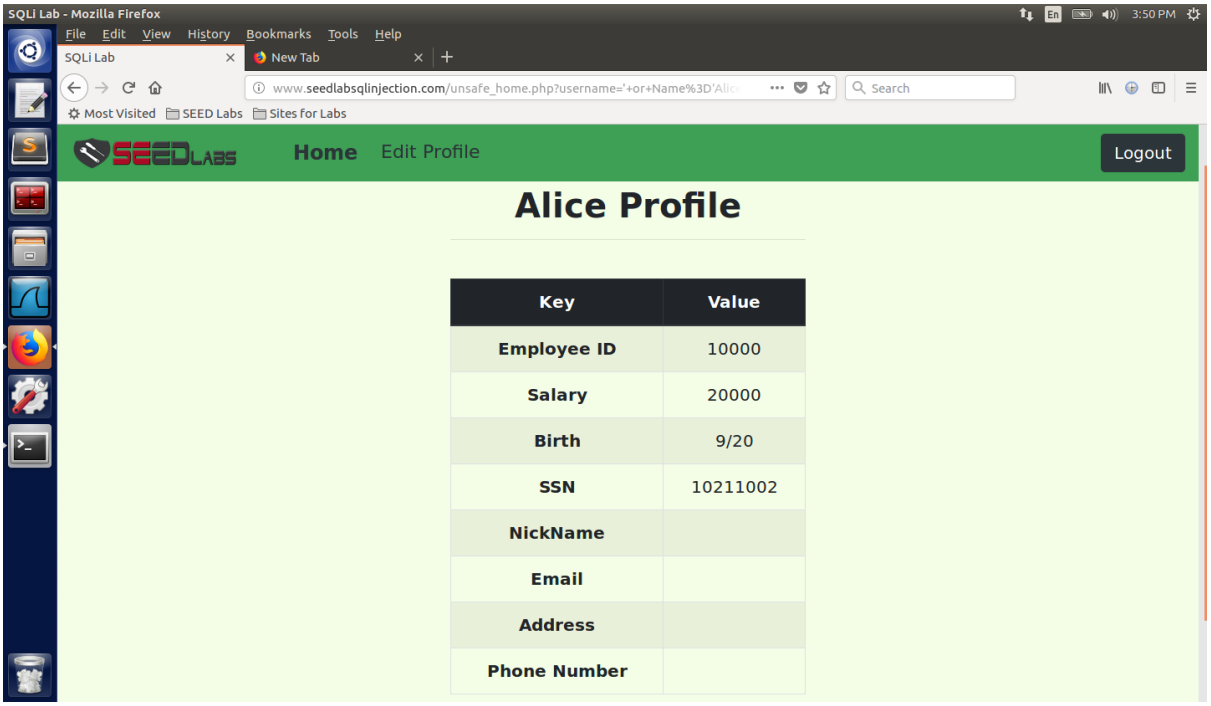
3.3 Task 3: SQL Injection Attack on UPDATE Statement

Task 3.3.1: Modify your own salary

Procedure:

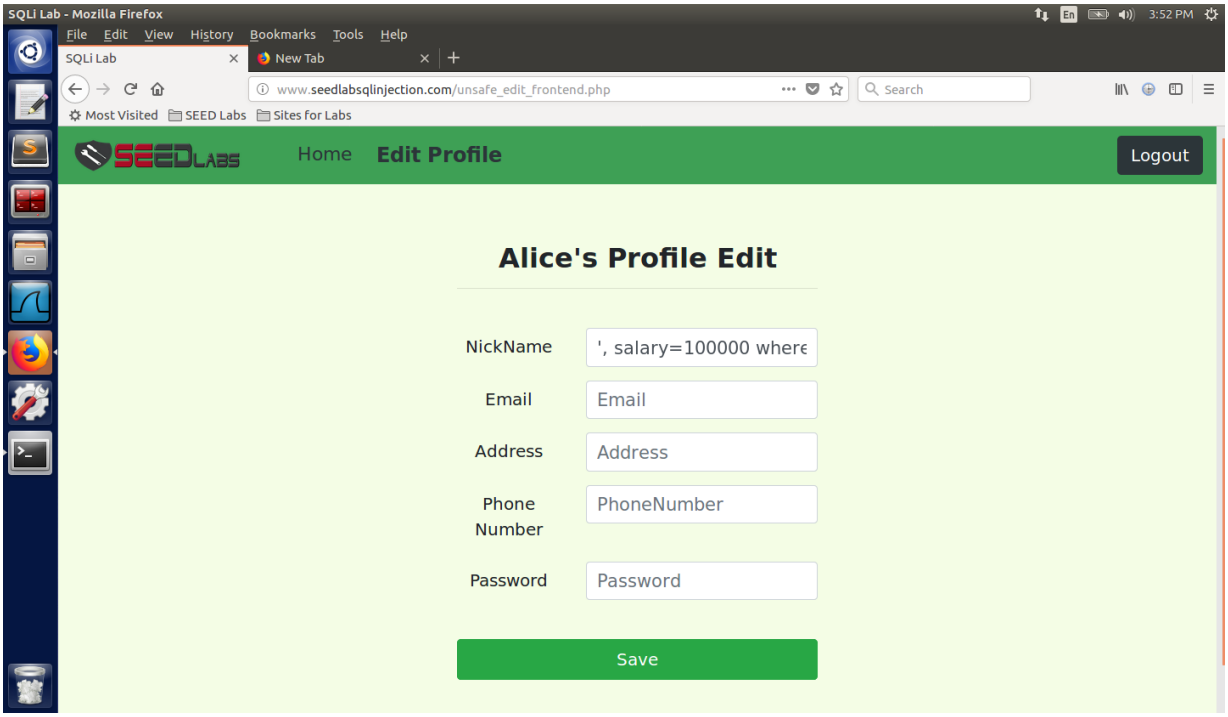
i. We login into the web application as Alice through the Employee Profile Login with the malicious payload injected: ***' or Name='Alice';#***

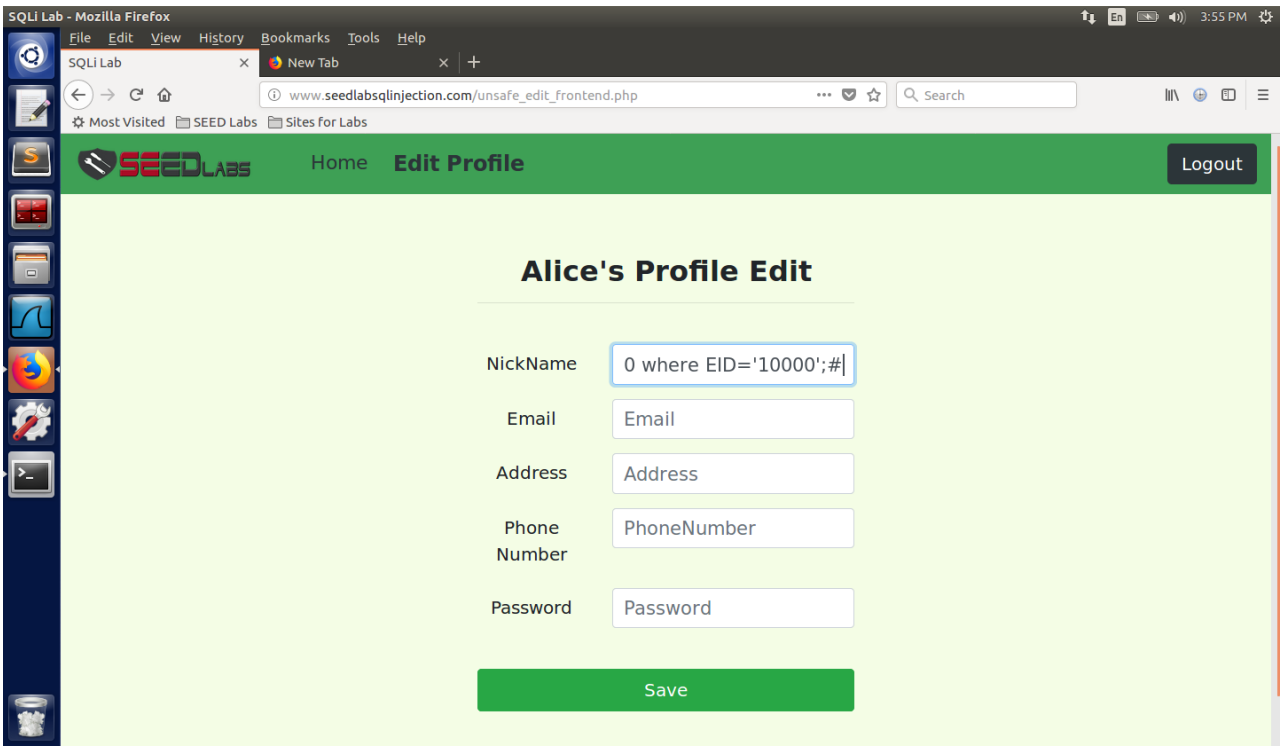




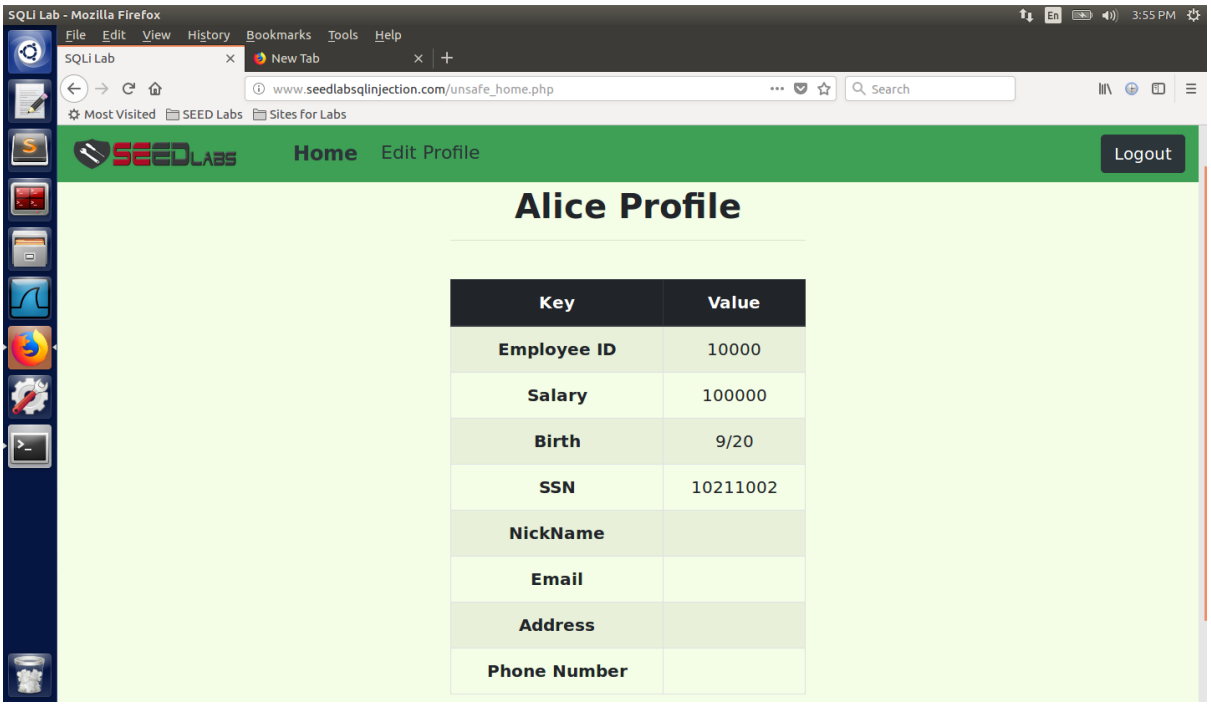
Please Turn Over…

ii. We then inject the following malicious payload into the Profile Edit SQL by typing it into the Nickname field and leaving the other fields blank and clicking on save.
', salary=100000 where EID='10000';#





Please Turn Over…



Observation and Explanation:

i. We login into the web application successfully as the user ‘Alice’ using the malicious payload ***‘ or Name=’Alice’;#***

ii. We then go to the Edit Profile option and successfully inject the malicious payload that updates the salary of Alice to 100000 from the previous 20000

iii. The injected SQL is ***‘, salary=100000 where EID=’10000’;#*** because we know the EID of Alice is 10000 from the previous screenshot.

iv. Finally, we note that Alice’s profile has been successfully edited.

v. This works because, the Profile Edit form uses the SQL Update query in the backend and the part of the original query after the # gets commented out.

Please Turn Over…

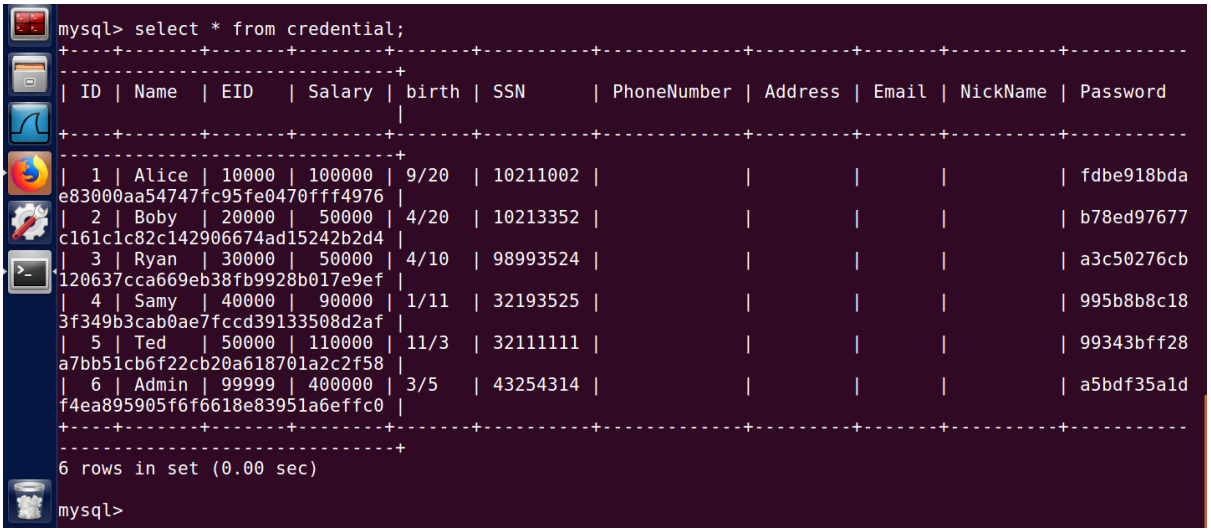
Task 3.3.2: Modify other people' salary

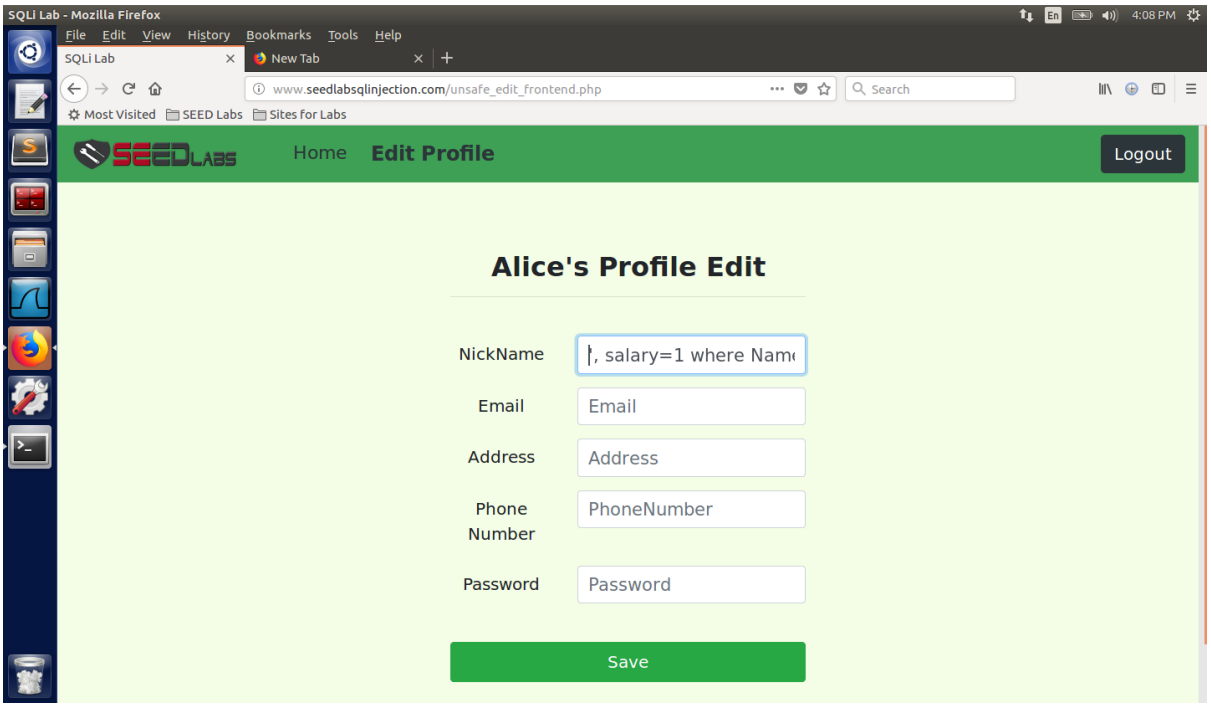
Procedure:

i. We first check the current salary of other people in the MySQL console by typing the command: ***Select * form credential;***

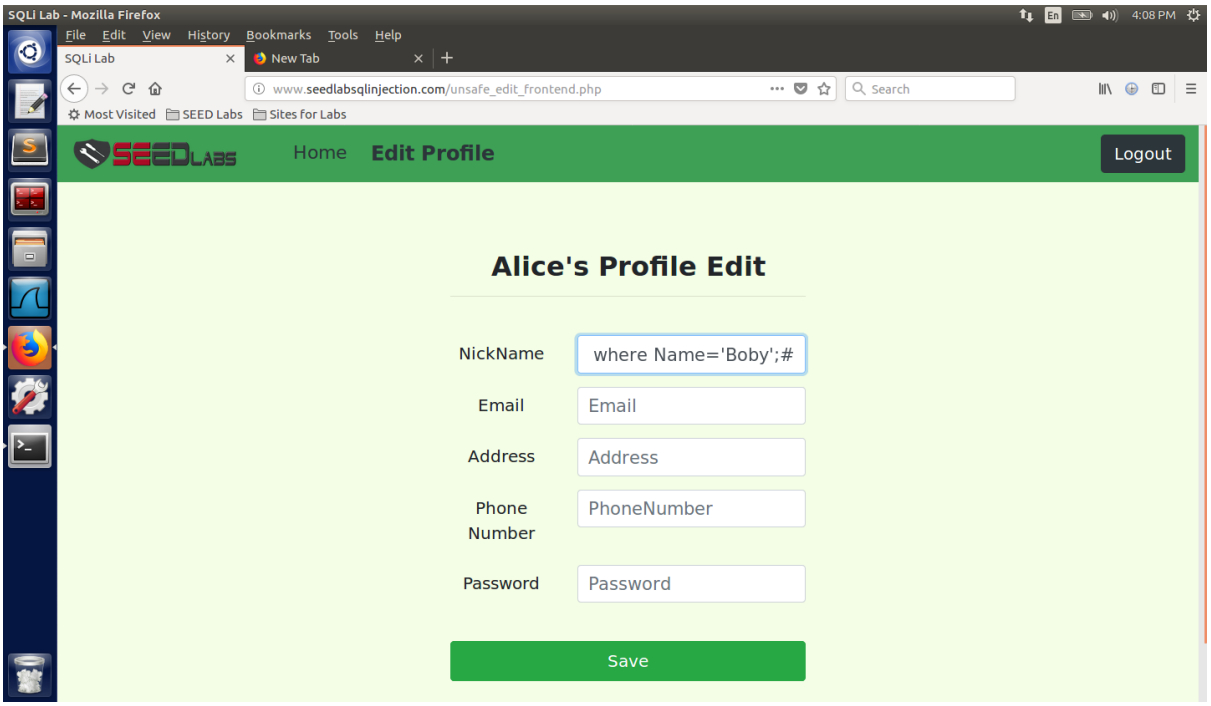
ii. Then as logged in user Alice, we go to the Profile Edit page and type in the following malicious payload in the NickName field to modify Boby's (Alice's boss) salary to 1:
', salary=1 where Name='Boby';#

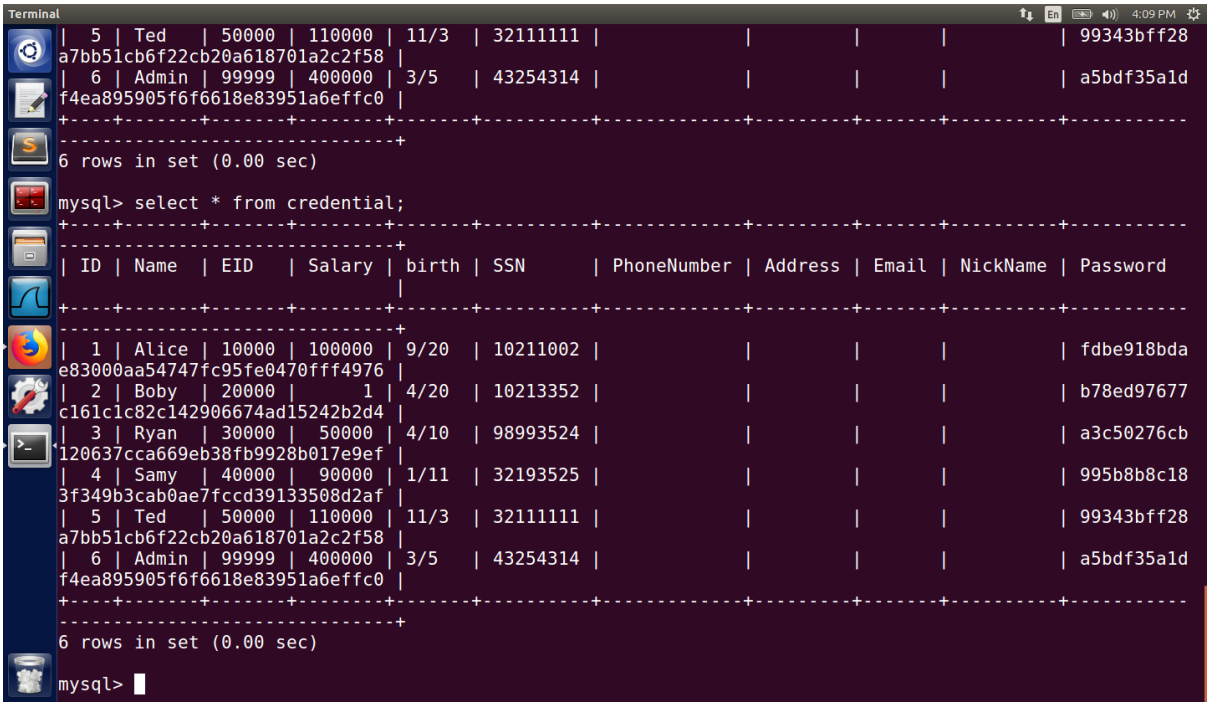
iii. We then click on save and check the updated values of salary of Boby by again running the MySQL command: Select * from credential;





Please Turn Over…





Observation and Explanation:

i. We note that the earlier salary of Boby was 50000 as can be seen in the MySQL console.
After SQL injection of malicious payload ***', salary=1 where Name='Boby';#*** Boby's salary gets updated to 1 as can be seen in the last screenshot.

ii. This is because, once we are logged in as any user, we can edit/update any other person's record as long as we have access to the web application form.

iii. The Edit form uses the Update SQL query and the part after the # is commented out.

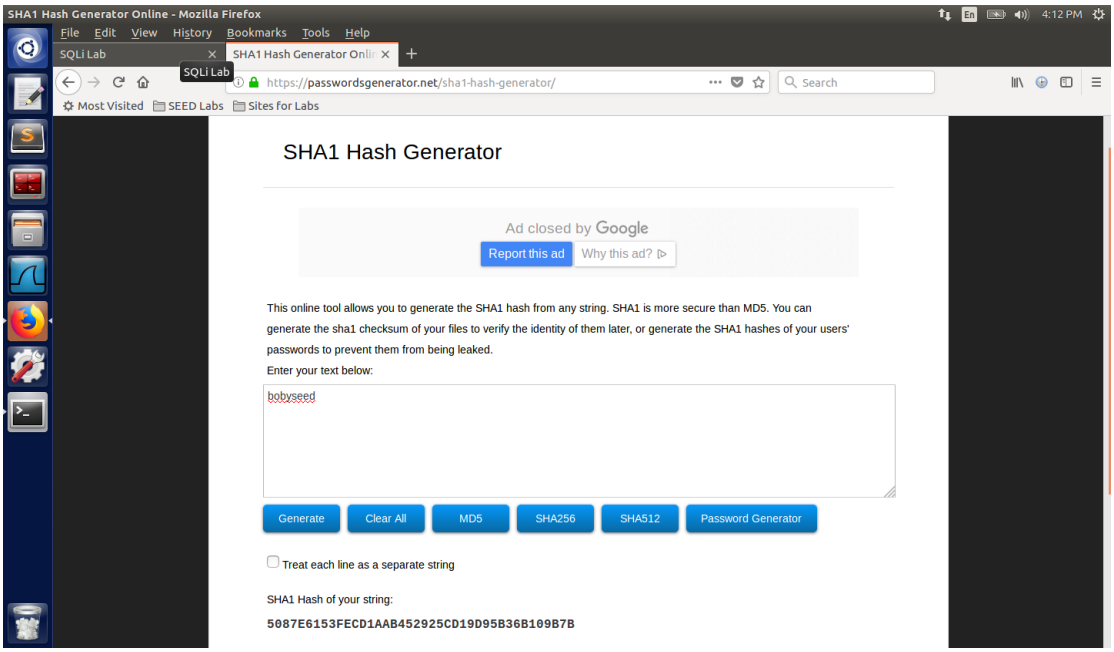
Please Turn Over…

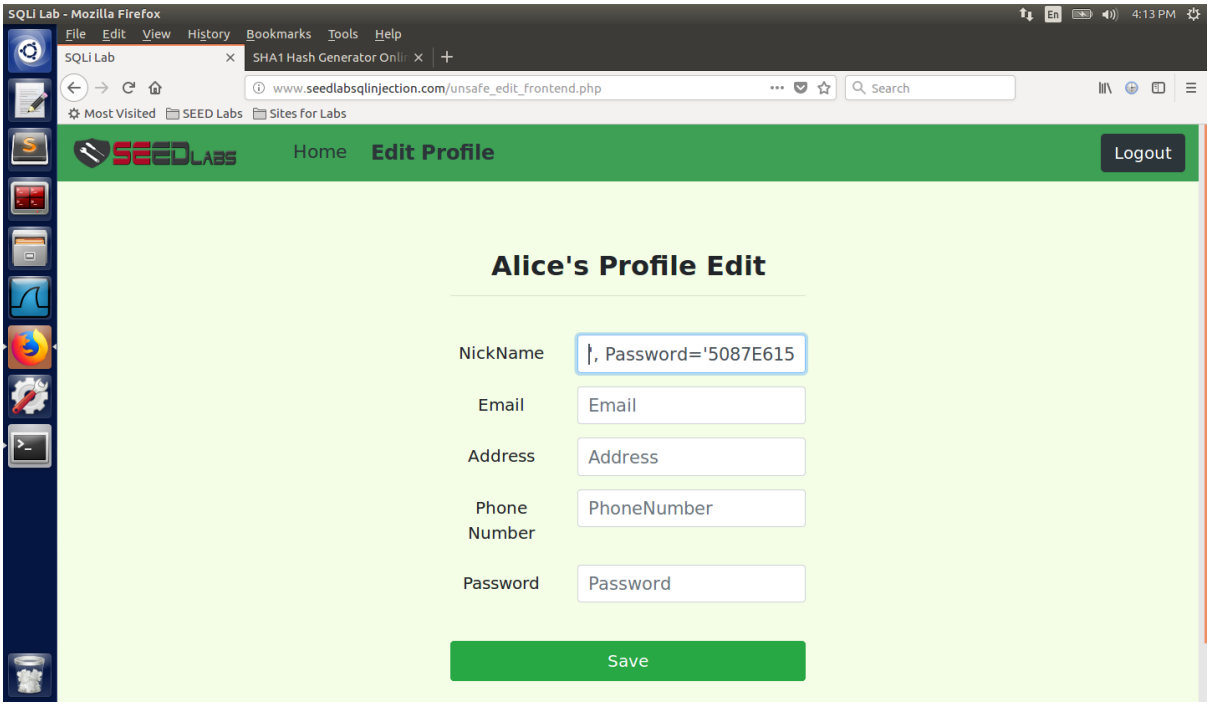
Task 3.3.3: Modify other people' password

Procedure:

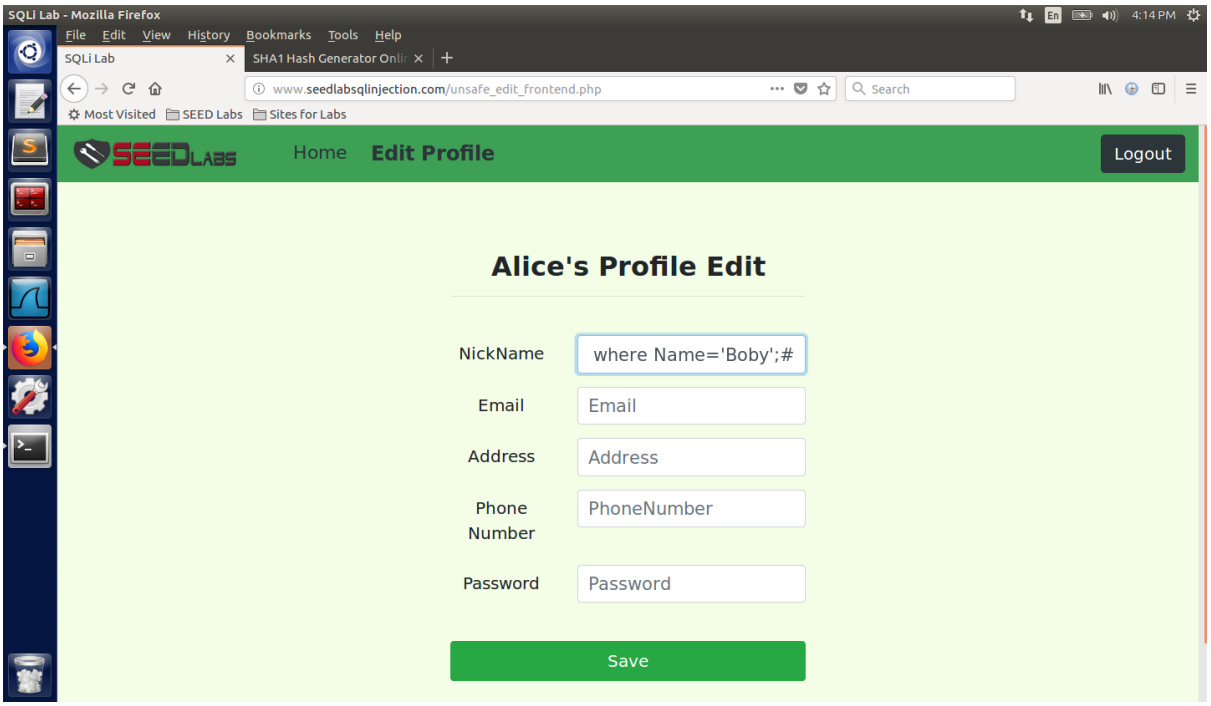
i. We first generate a password of our choosing by generating the SHA1 hash of our chosen password ***bobyseed***

ii. We then inject the following malicious payload into the SQL Update statement via the Profile Edit form:
', Password='5087E6153FECD1AAB452925CD19D95B36B109B7B' where Name='Boby';#

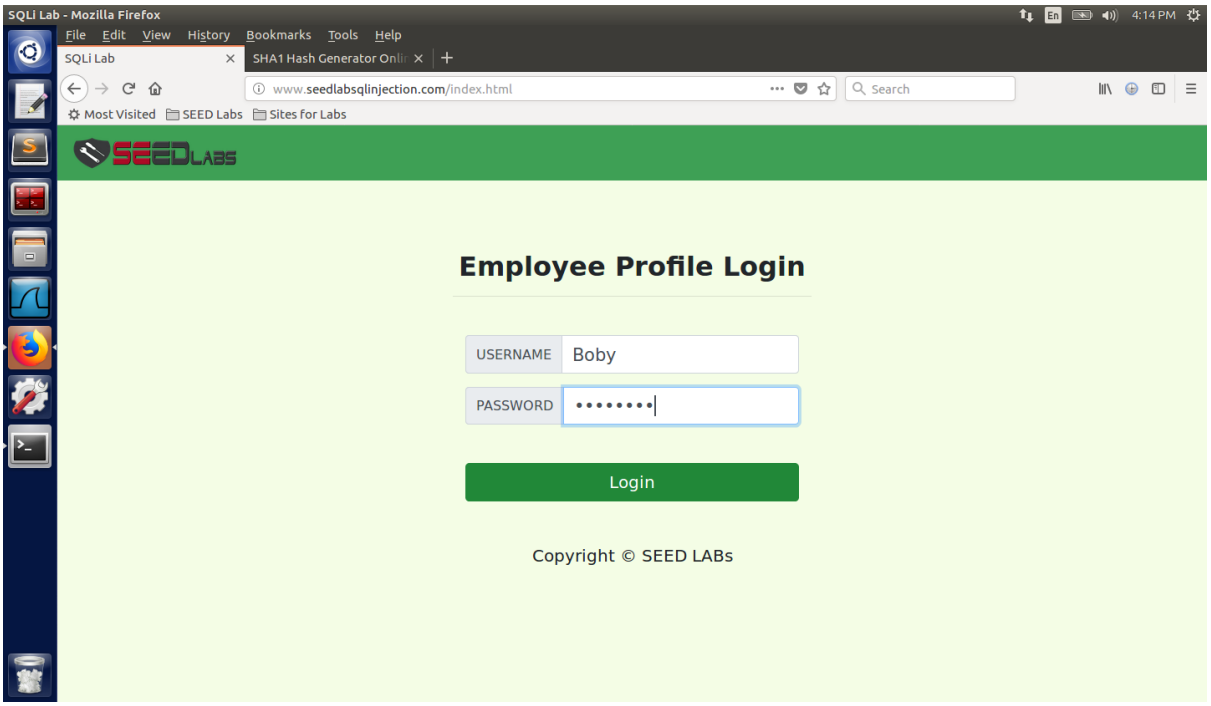


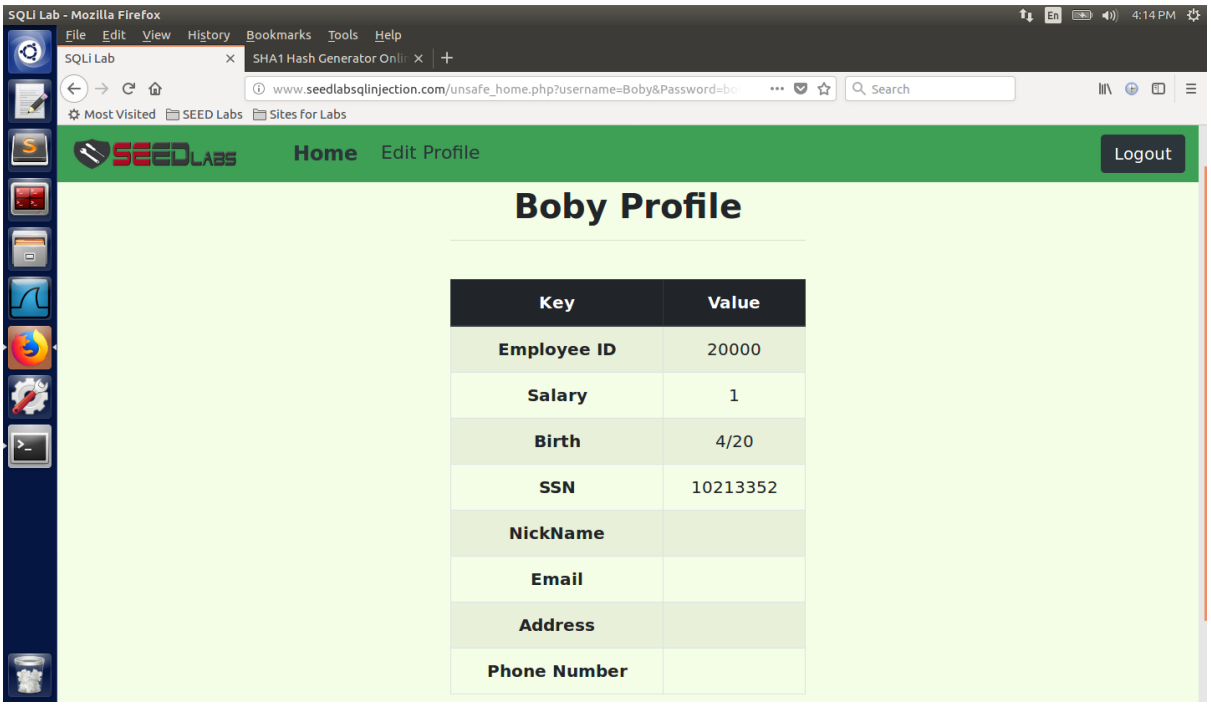


Please Turn Over…



iii. We then logout as Alice and login again as the user Boby using the new password of our choosing bobyseed that we injected into the SQL in the previous step to update the database.





Observation and Explanation:

i. The SHA1 hash of our chosen password bobyseed is successfully generated via an online hash generator.

ii. We note that the SQL injection attack to update the password of user Boby as **bobyseed** is successful via the malicious payload: ***', Password='5087E6153FECD1AAB452925CD19D95B36B109B7B' where Name='Boby';#***

iii. The SHA1 hash of the chosen password is used in the malicious payload injected since the password is stored as SHA1 hash in the database.

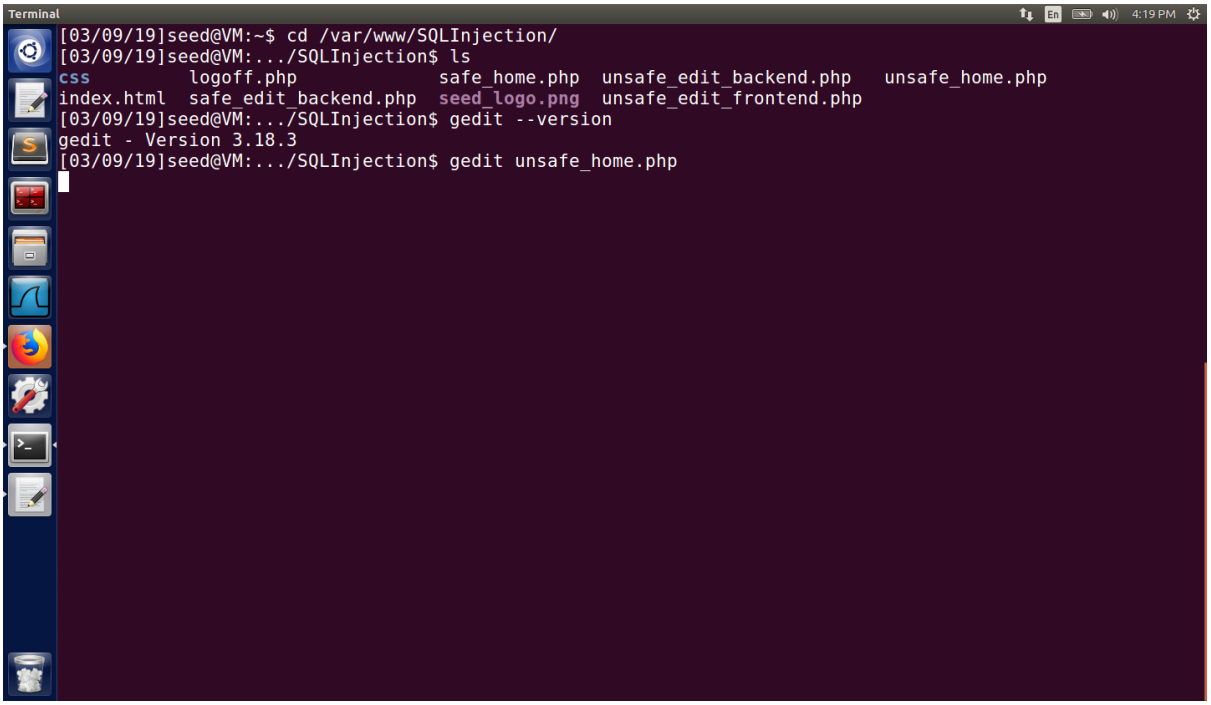
iv. We are also able to successfully login as username Boby and password bobyseed to see the profile details of Boby and edit any other fields we want.

Please Turn Over…

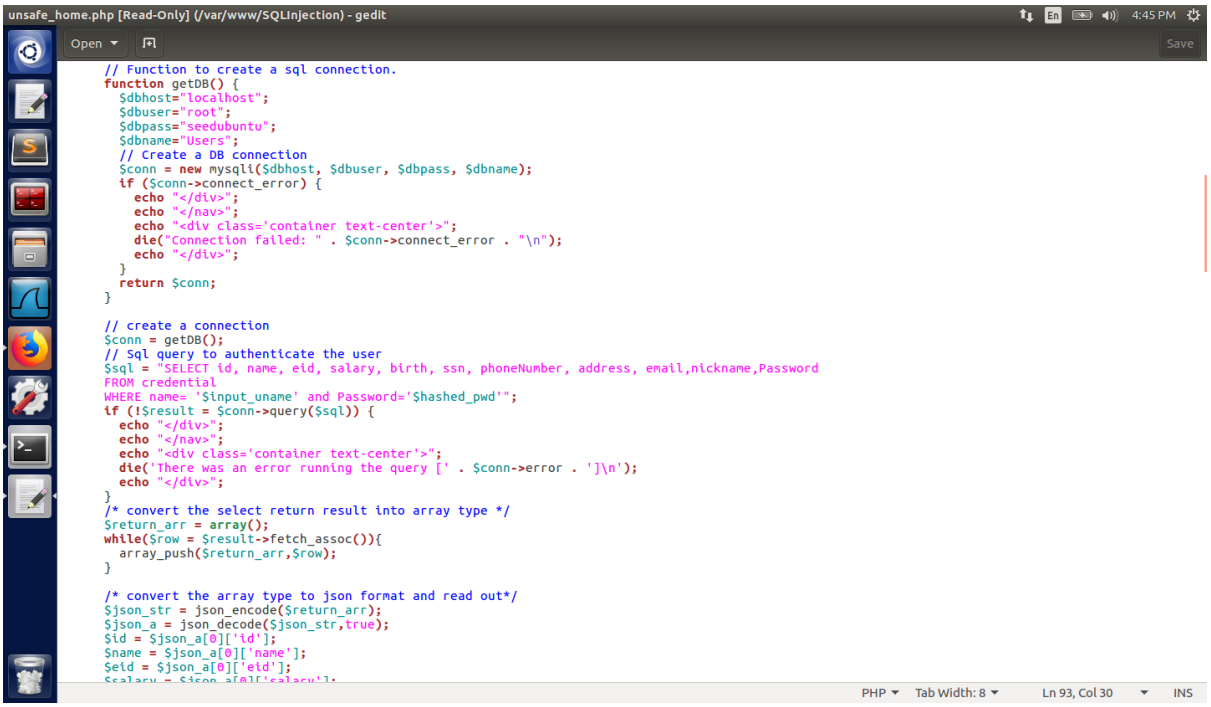
3.4 Task 4: Countermeasure — Prepared Statement

Procedure:

i. We navigate to the root folder where the web application files are hosted: /var/www/SQLInjection and open the unsafe_home.php file and the unsafe_edit_backend.php files for making them safe by making the SQL statements as Prepared Statements.

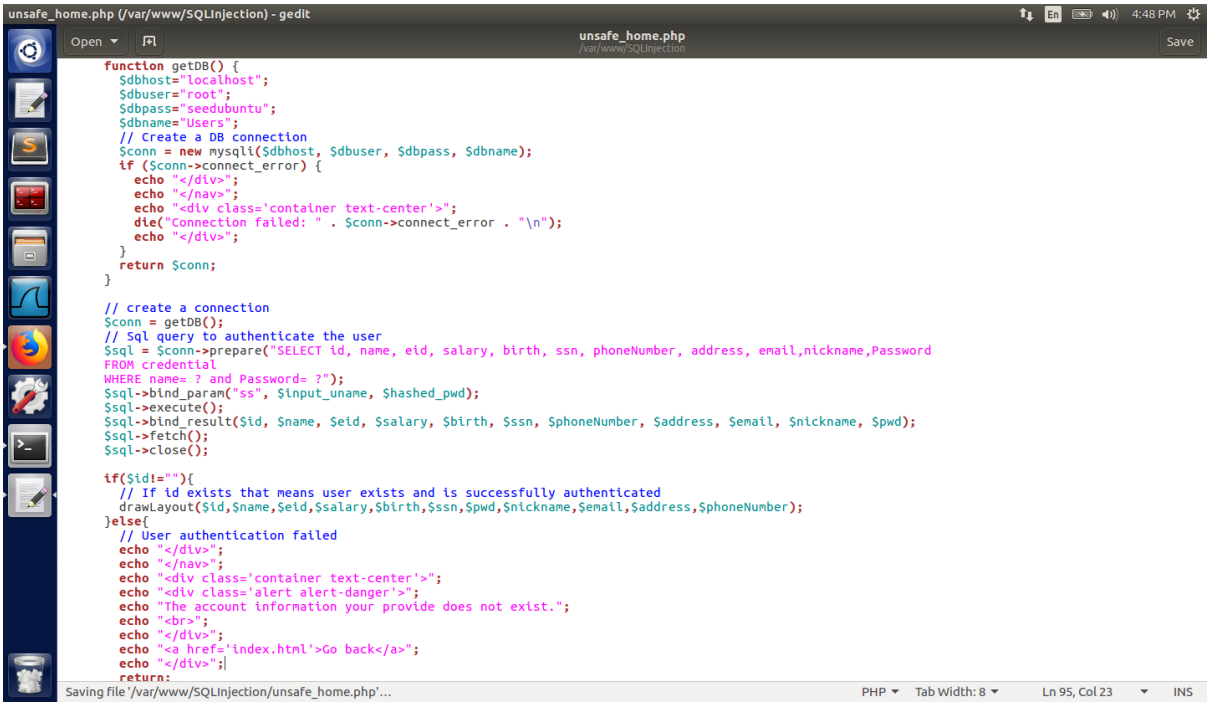


unsafe_home.php (Before Prepared Statement changes)

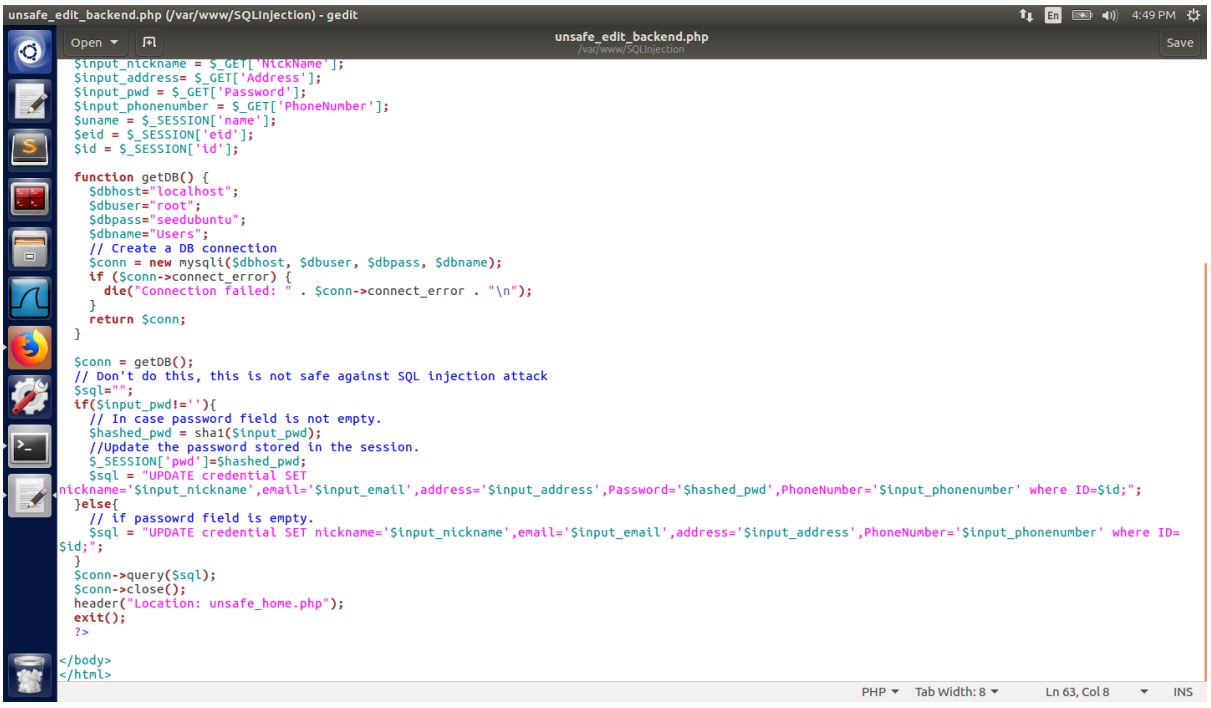


Please Turn Over…

unsafe_home.php (After Prepared Statement changes)



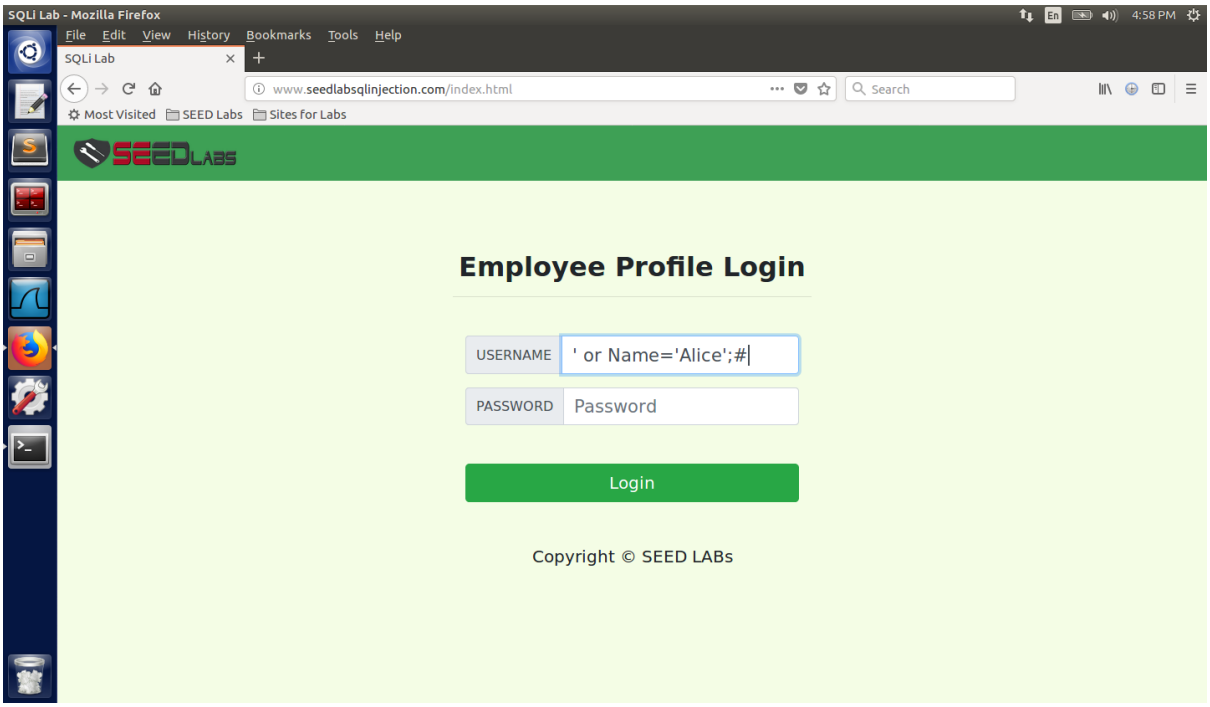
unsafe_edit_backend.php (Before Prepared Statement changes)

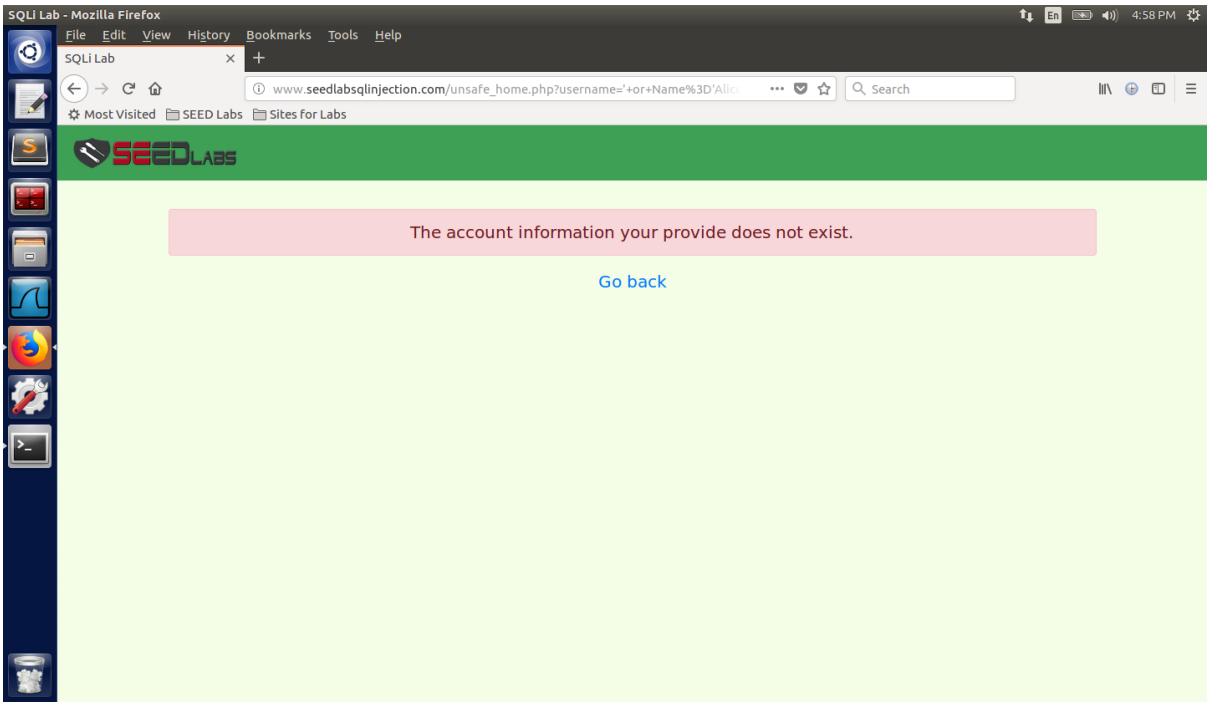


Please Turn Over…

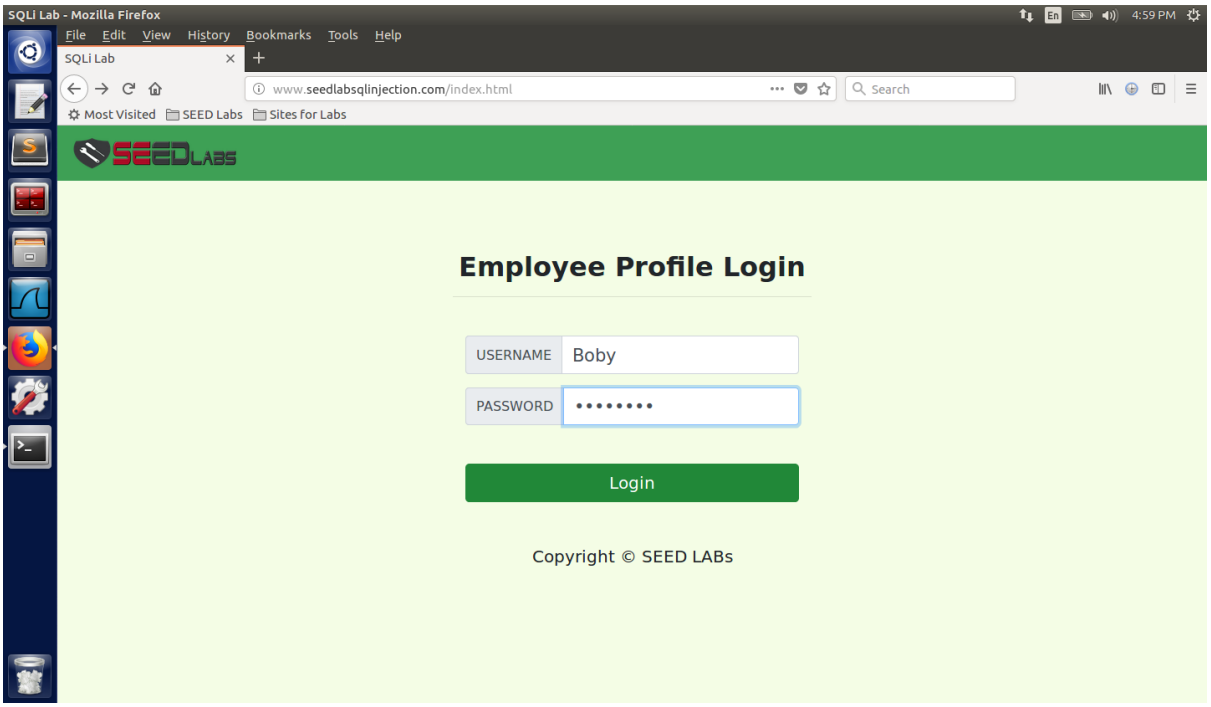
unsafe_edit_backend.php (After Prepared Statement changes)

ii. The SQL Injection attacks carried out in the previous steps are repeated – The SQL injection attack for login (Select statement).
Malicious payload used in Username field: ***' or Name='Alice';#***

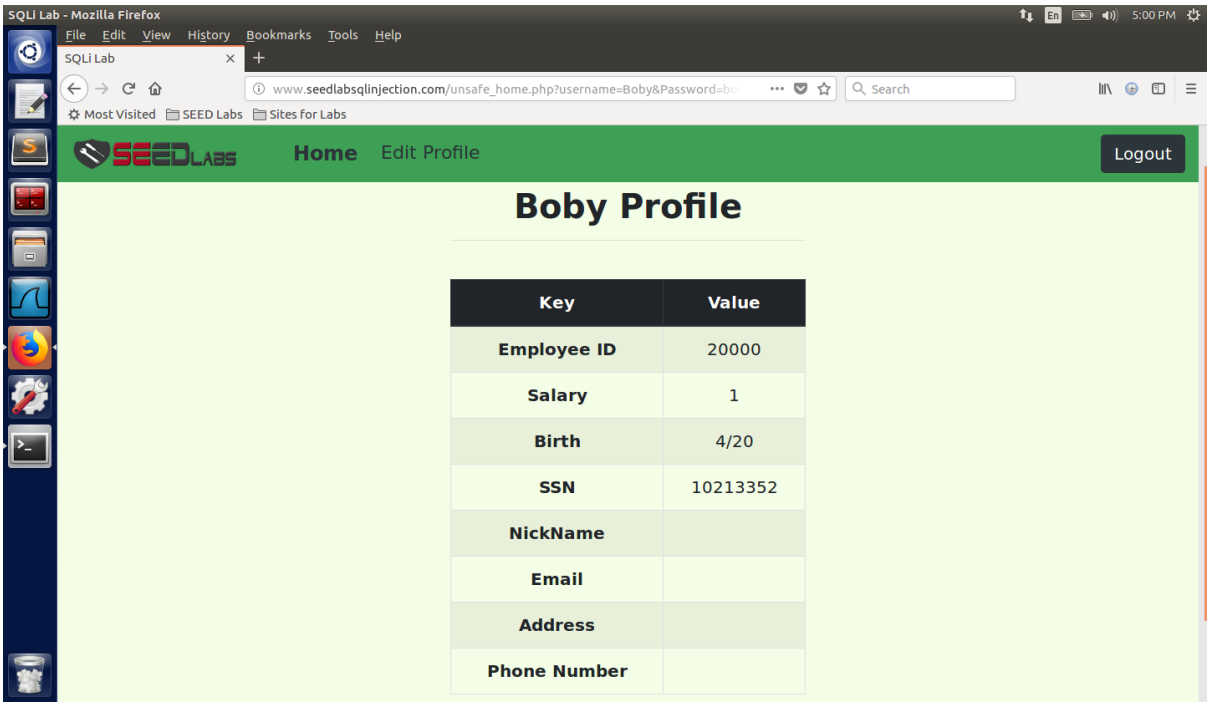


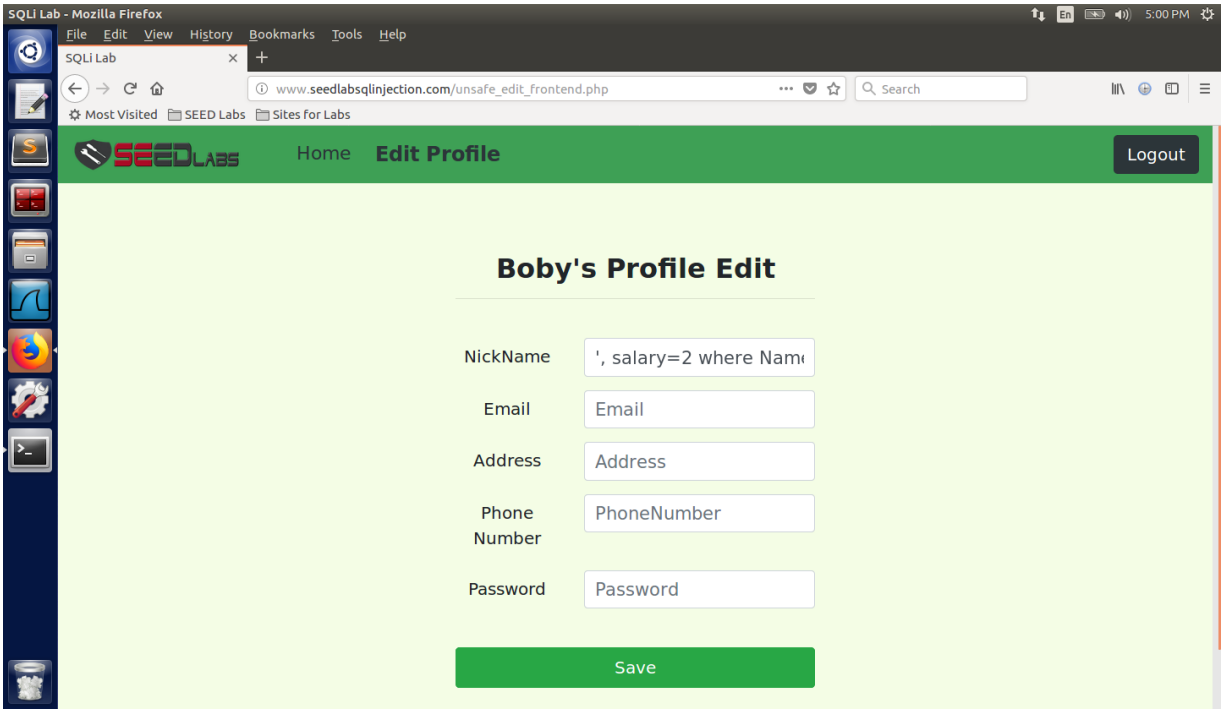


iii. Next we login as user Boby for whom we know both the username and password (of our choosing we set in a previous attack) and try the SQL injection attack for Update statement (Edit Profile).

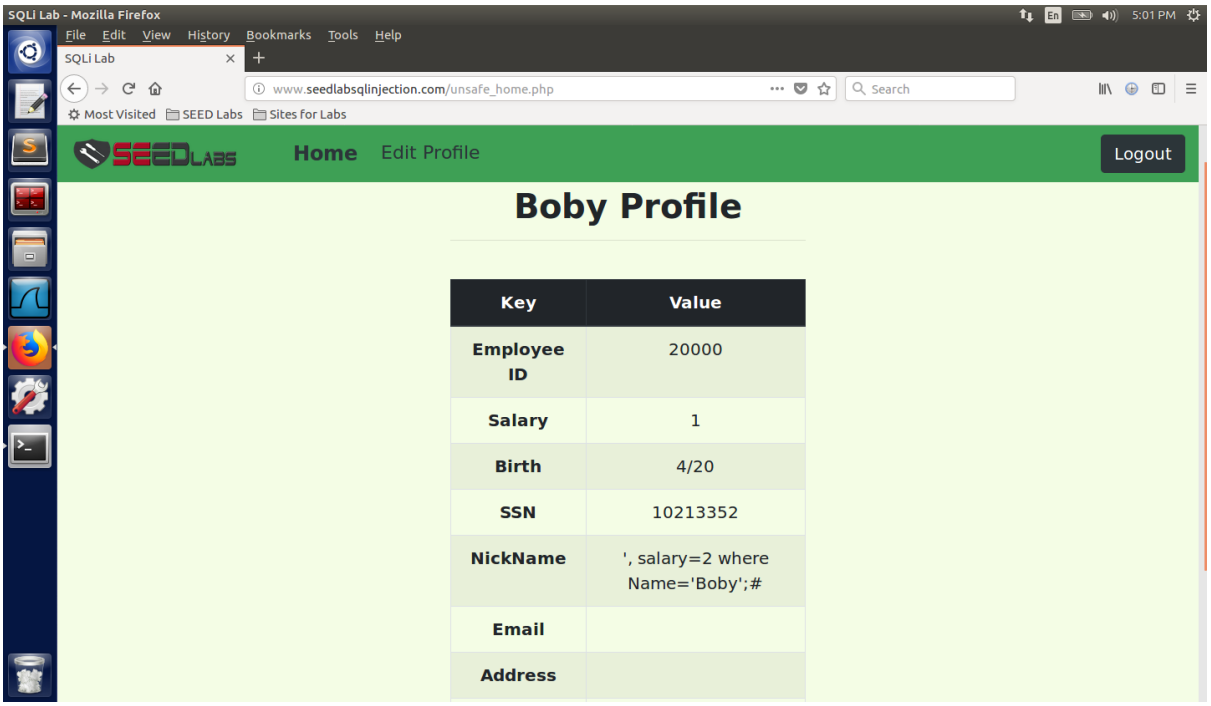


Please Turn Over…





Please Turn Over…



Observation and Explanation:

i. After changing the unsafe_home.php file and unsafe_edit_backend.php files using prepared statements, when we perform SQL injection attack on the login page, the attack does not succeed. Instead we see the error message “The account information that you requested does not exist.”

ii. Similarly, when we perform the SQL Injection on the Profile Edit form, the attack does not succeed. Instead we find that the malicious payload that we injected in the NickName field is stored whole as a string in the NickName field as can be seen from the last screenshot.

iii. This is because, prepared statements, compile the SQL statements with placeholders for the data and the data is plugged in at the end. As a result, any injected SQL can not be compiled and hence gets treated as the bound datatype which is string in this case.

Please Turn Over…