**IS LAB ASSIGNMENT#03**

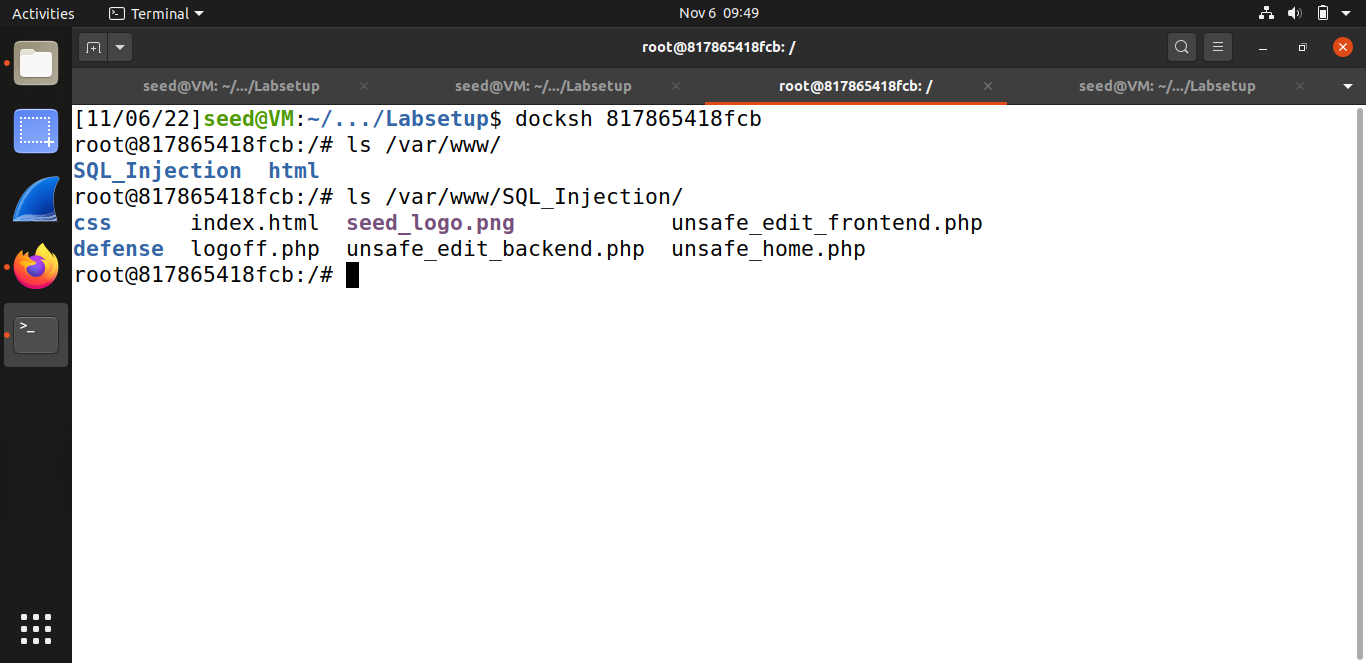
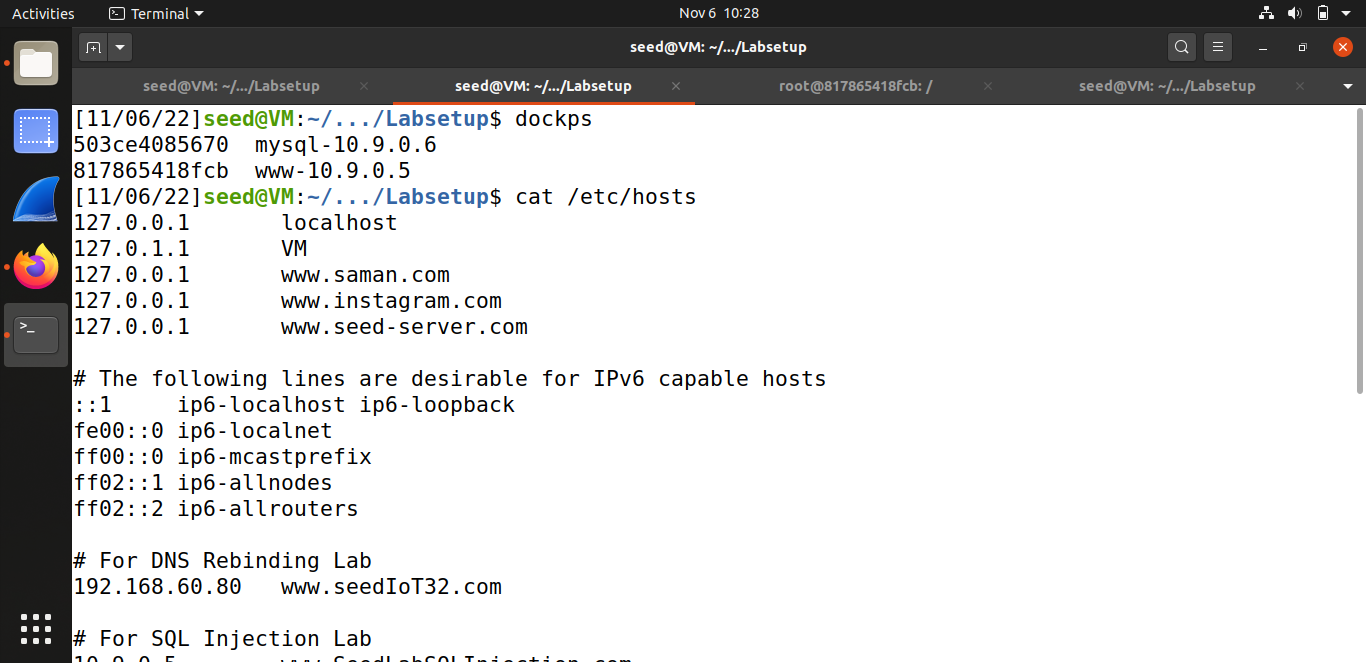
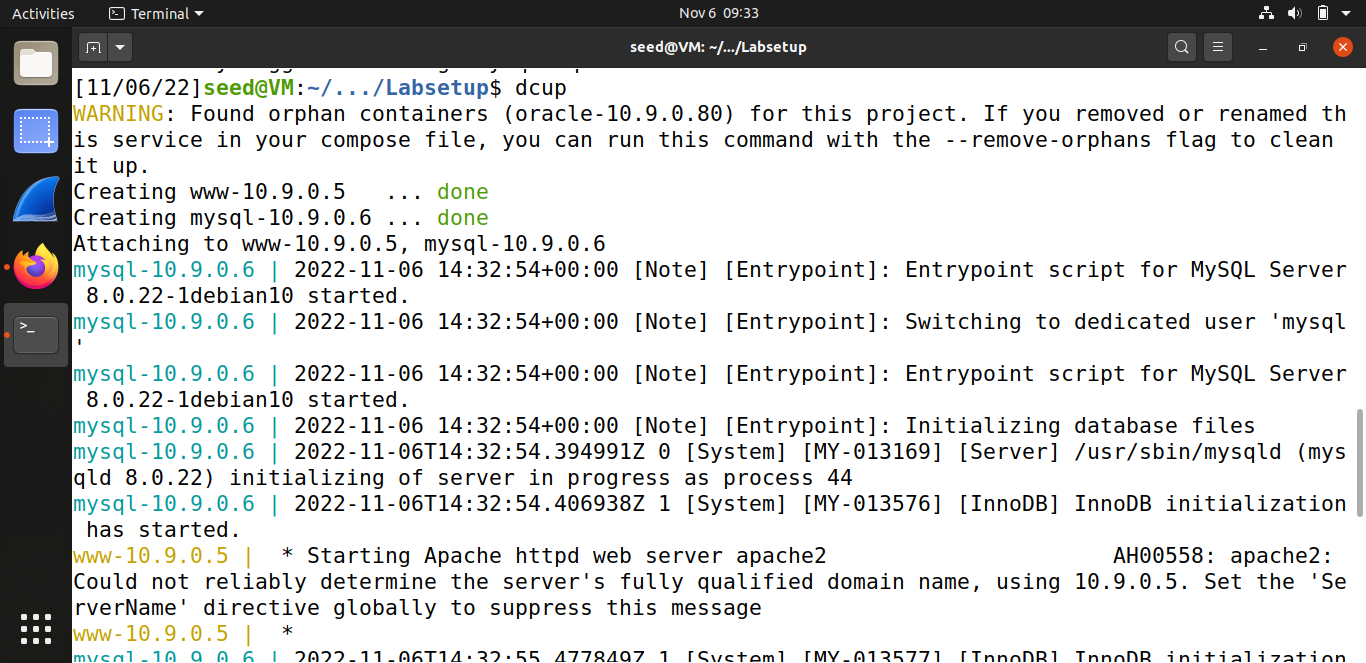
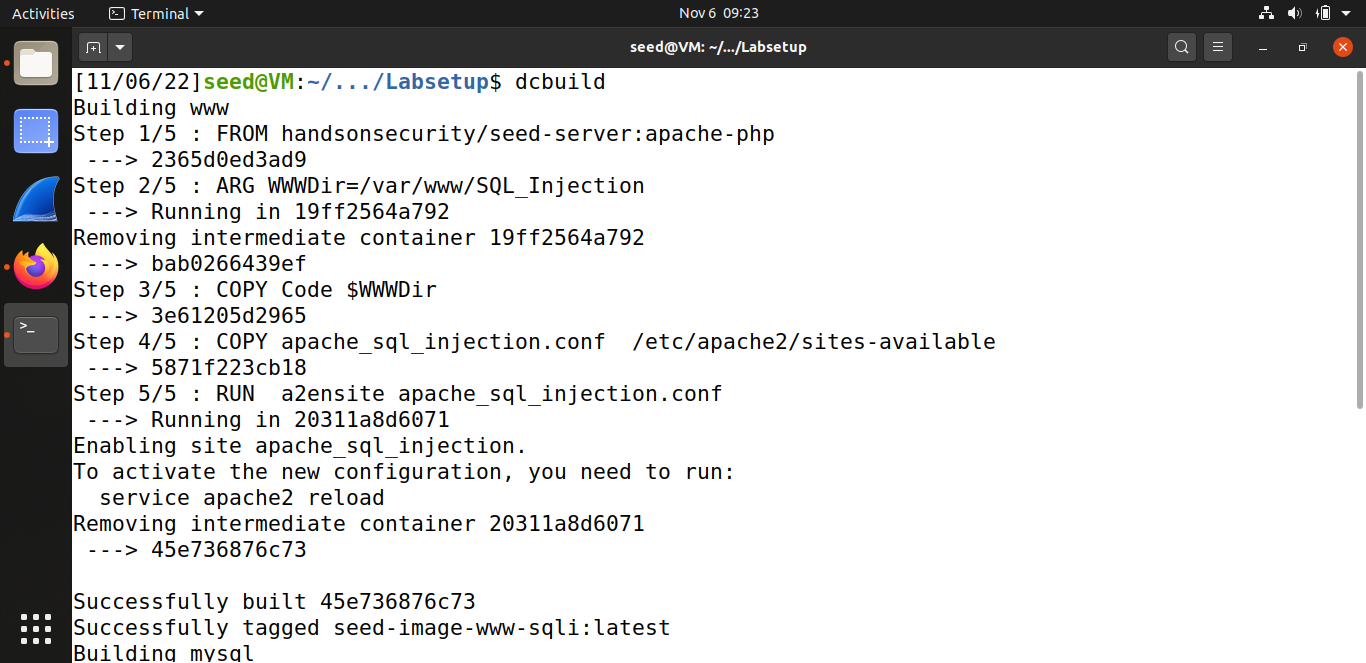
**Name: Saman Khan**

**ID: 19K-0354**

**Section: H**

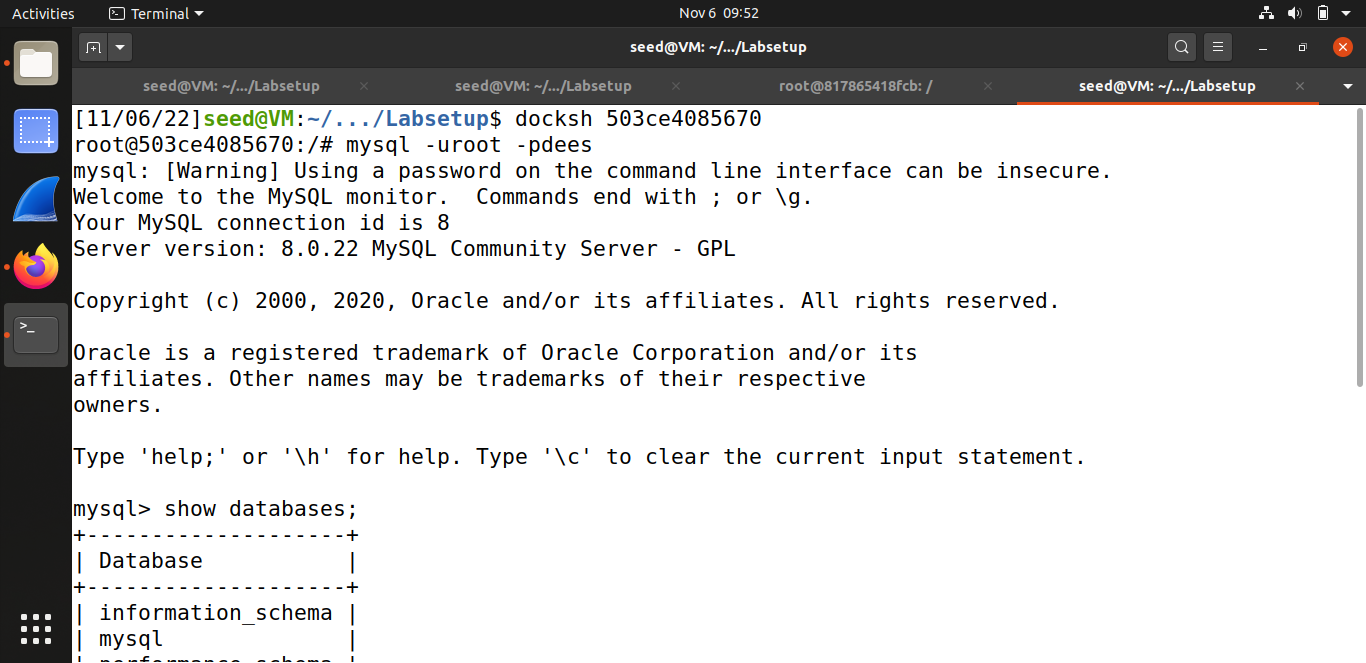
**Lab Environment Setup:**

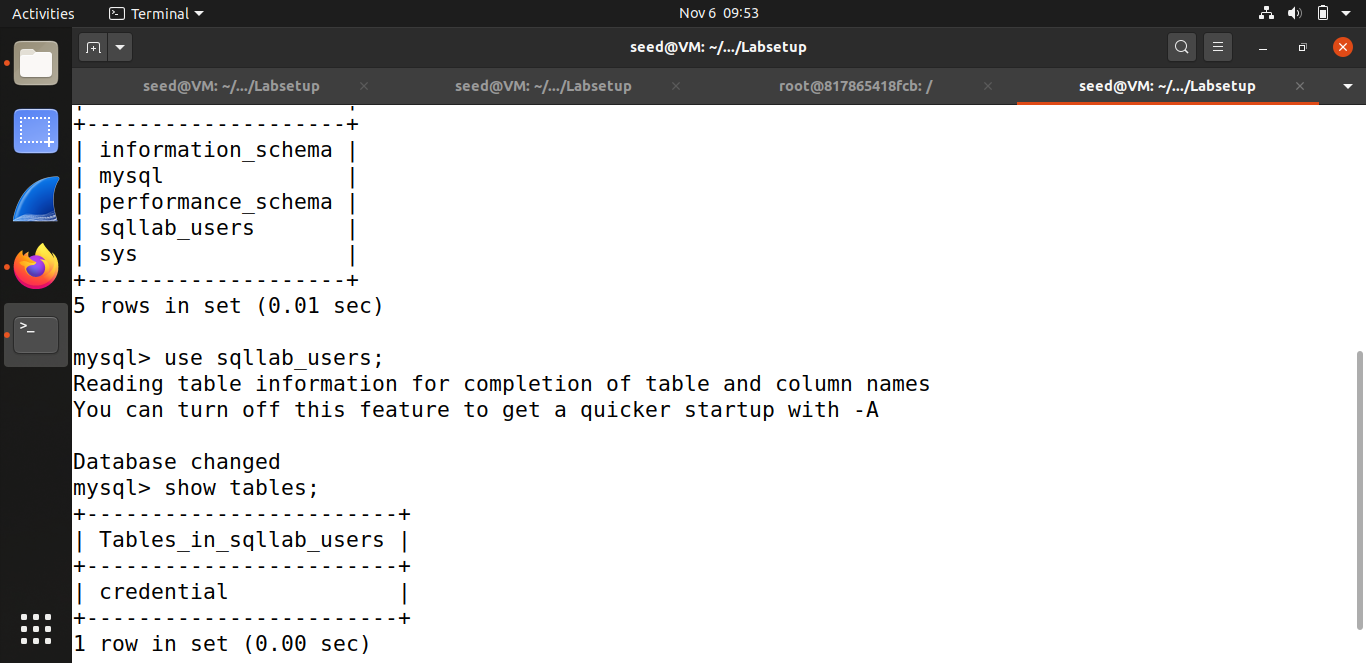
The default web application URL for this lab is is only accessible from inside of the virtual machine, as we have modified the /etc/hosts file to map the domain name of each URL to the virtual machine’s local IP address (127.0.0.1).

****

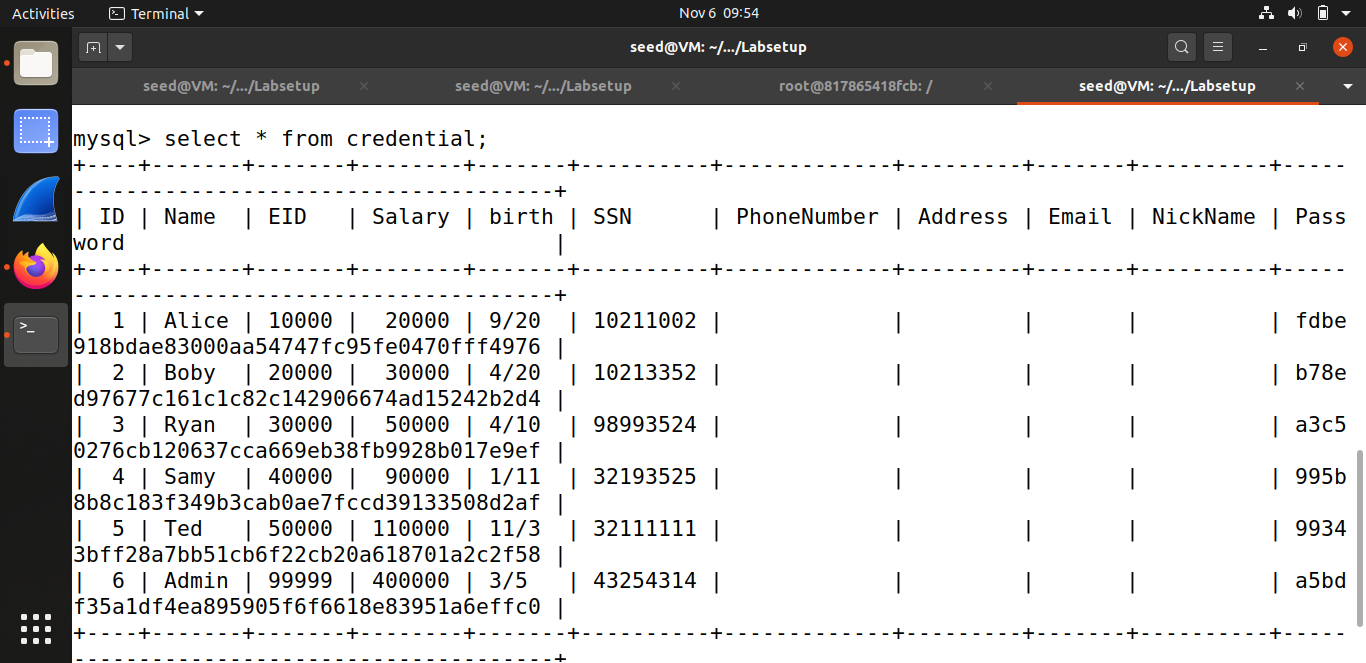
**TASK#01: Get Familiar with SQL Statements**

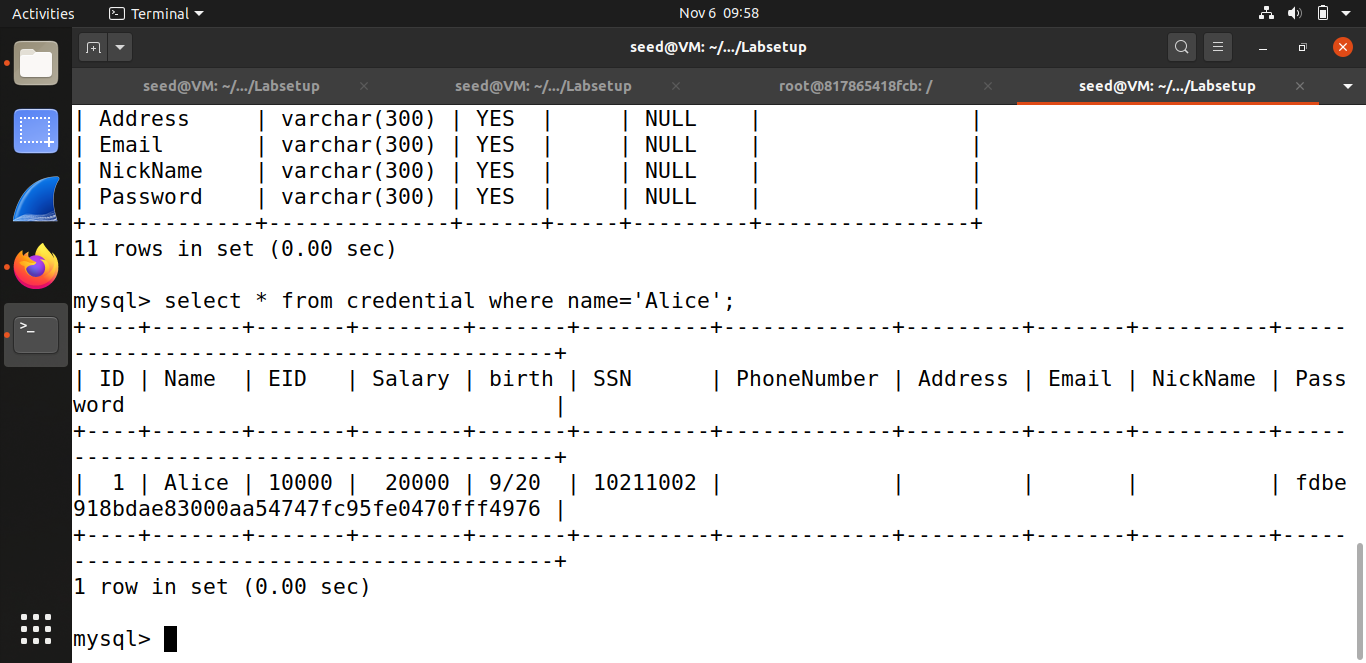
We first log into the MYSQL console, by using the username and password provided, and switch the database in use to Users so that we can access the credential table.







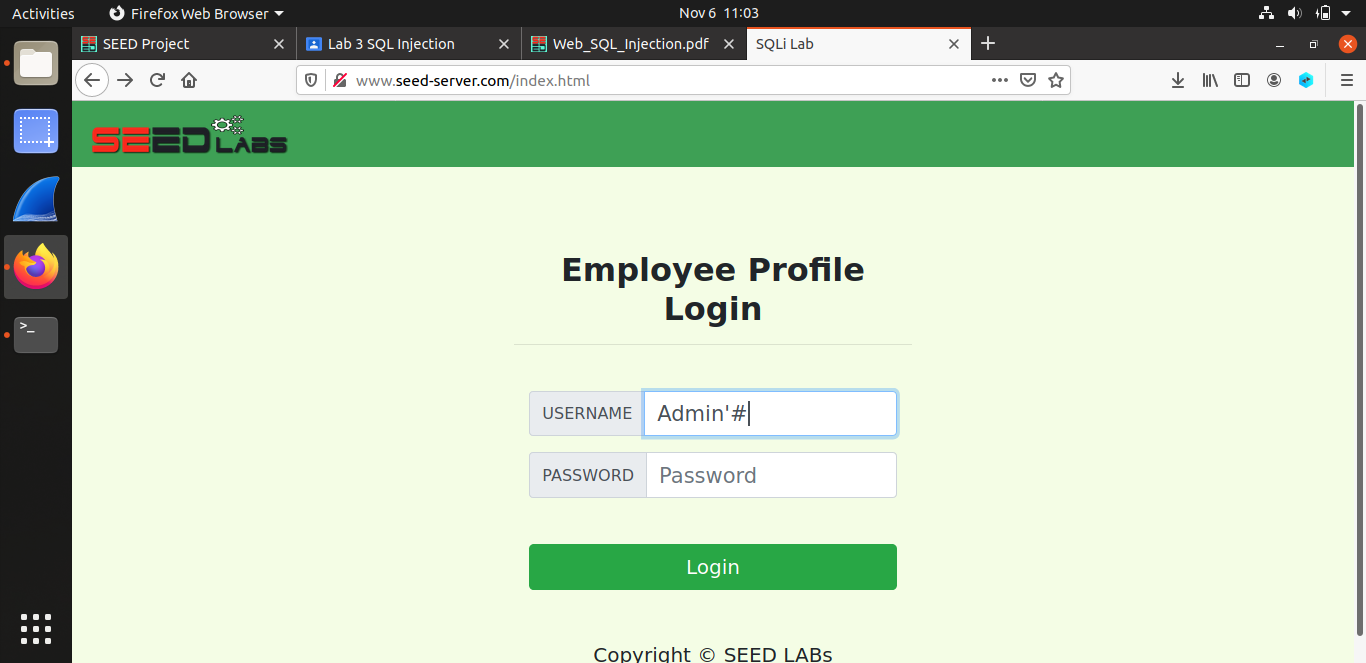


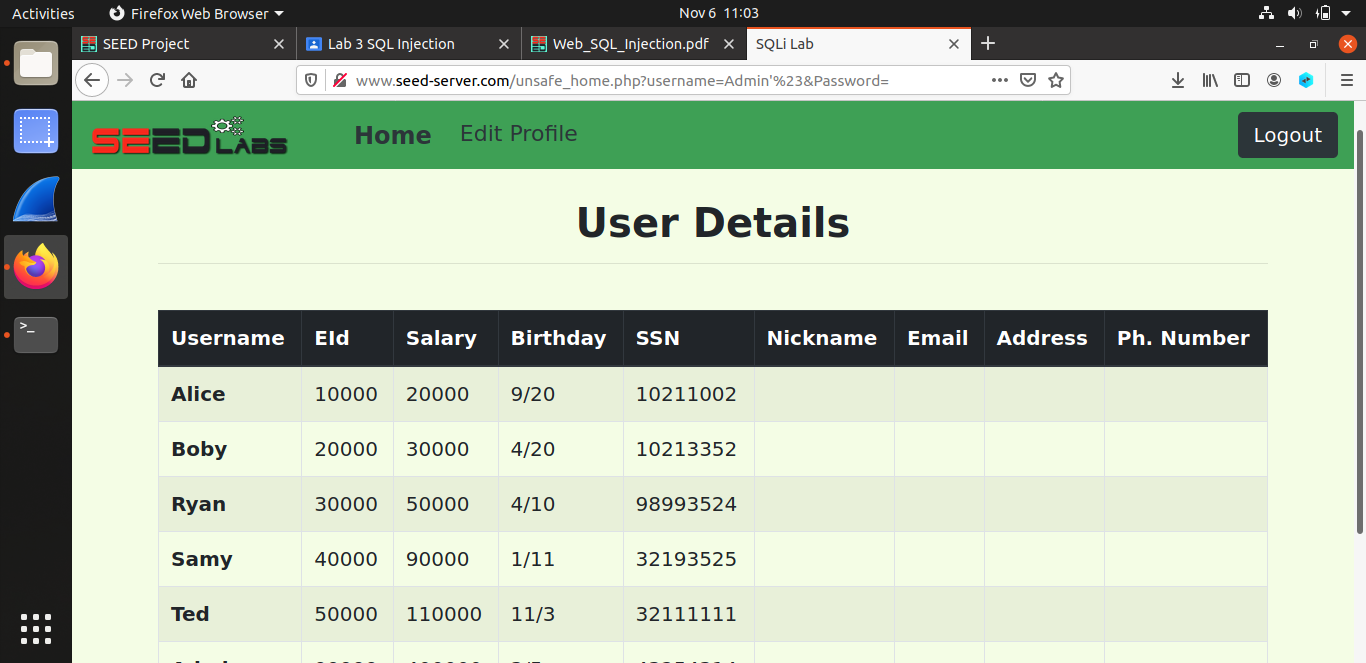
Using SQL command to print all the profile information of the employee named Alice.

**TASK#02: SQL Injection Attack on SELECT Statement**

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

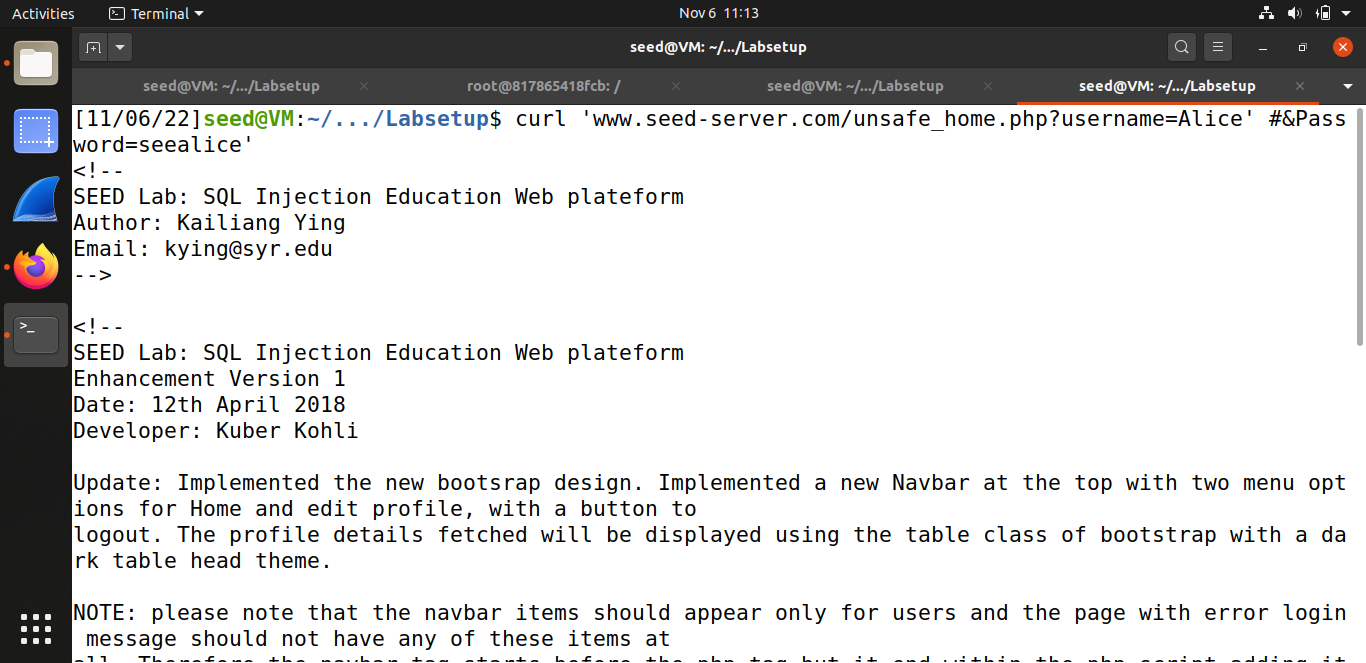
Logging into the web application as the administrator from the login page, using Admin’# in the input field of username and leaving password field blank. The ‘#’ marks the query that follows after it as a comment hence password field is not even executed.

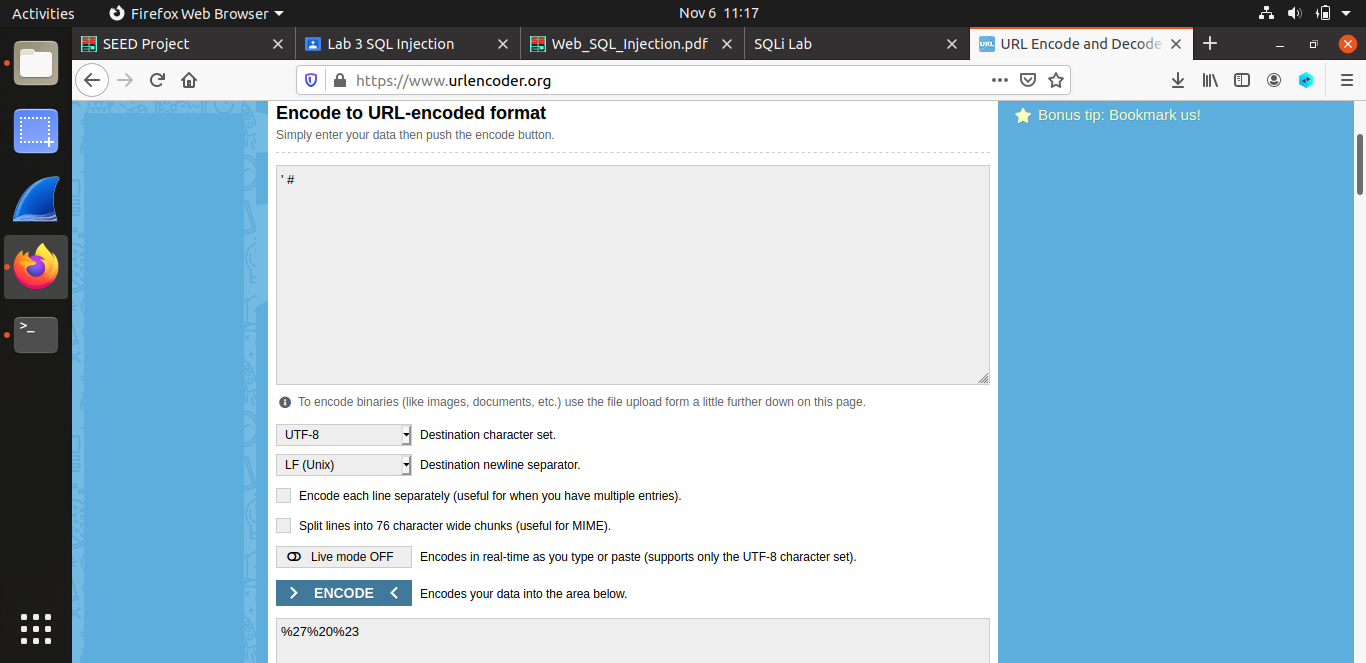


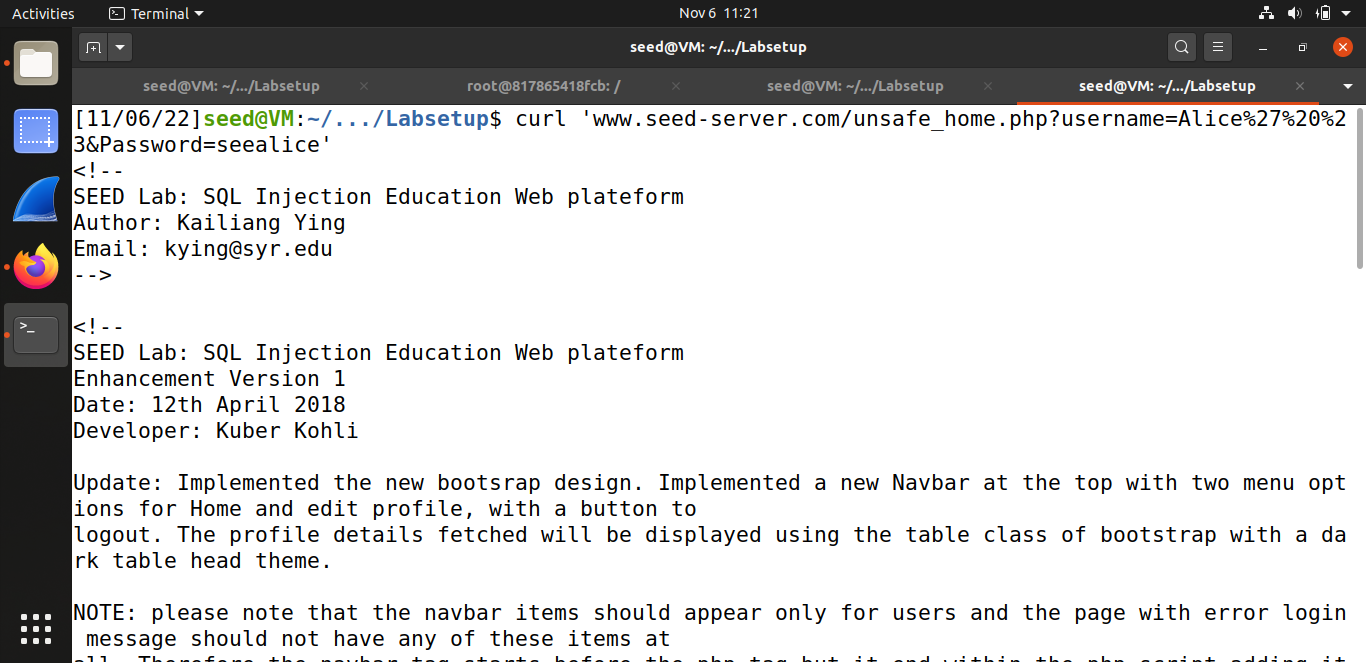


* **Task#2.2: SQL Injection Attack from command line**

Here the curl command is used to place an HTTP request from the terminal and perform the same login operation. We get the HTML page in return. First we used simple ‘ #, but then we encoded these characters using urlencoder.org and then used the encoded output.

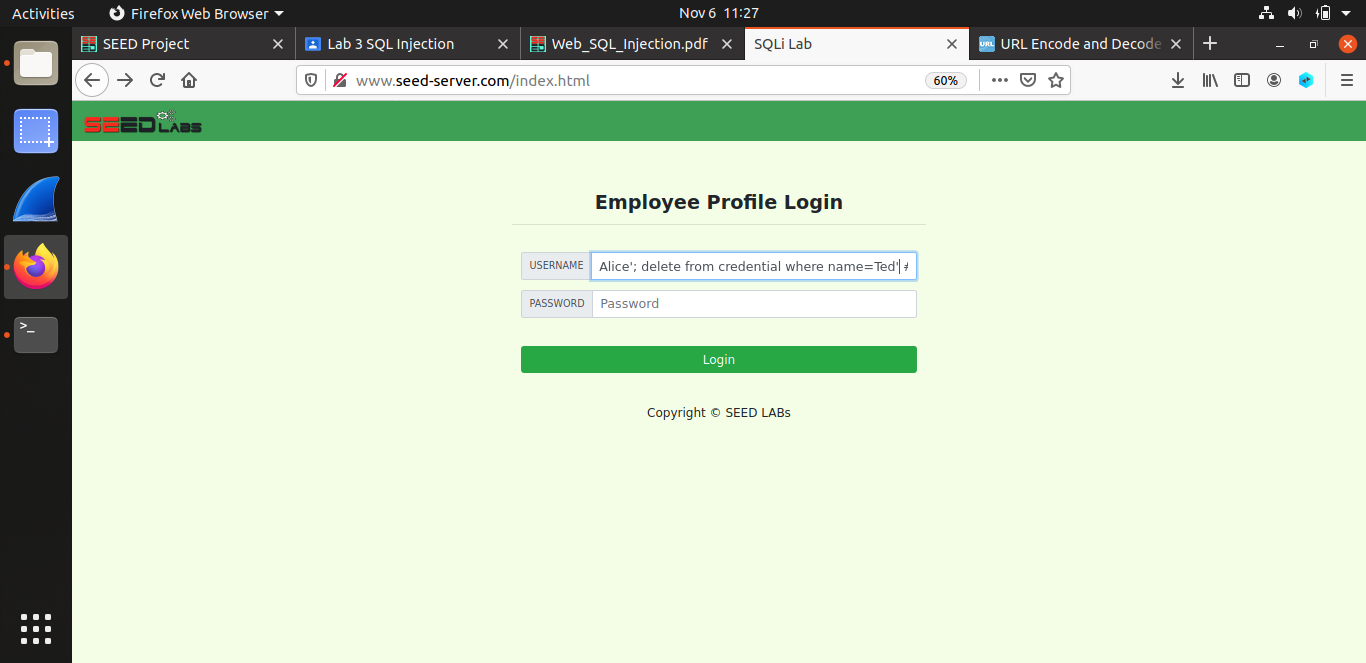
****

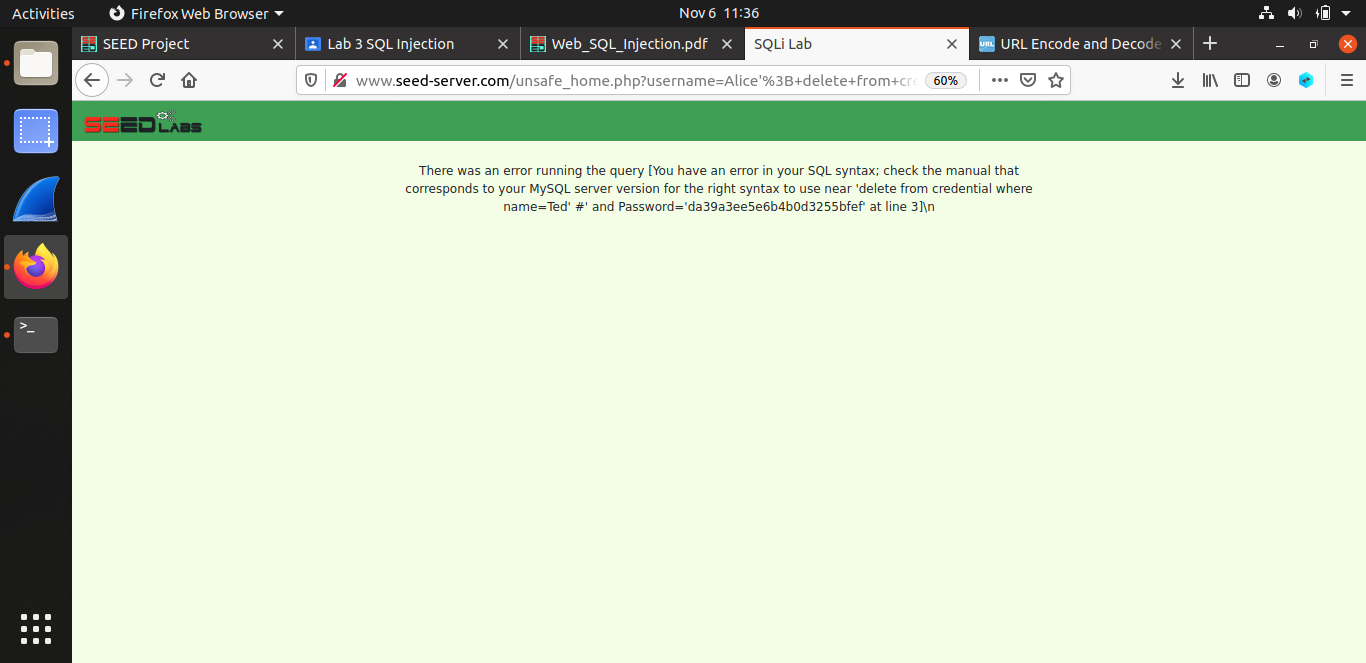
****

****

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

In order to append a new SQL query we place a ‘;’ to separate the two queries. This method is known as piggy-backed queries where the user injects malicious queries like delete or update. However, our attempt at running the query was unsuccessful because this SQLi doesn’t work against MySql. In PHP’s MySql extension the mysql::query() API doesn’t allow multiple queries to run in the database server.





**TASK#03: SQL Injection Attack on UPDATE Statement**

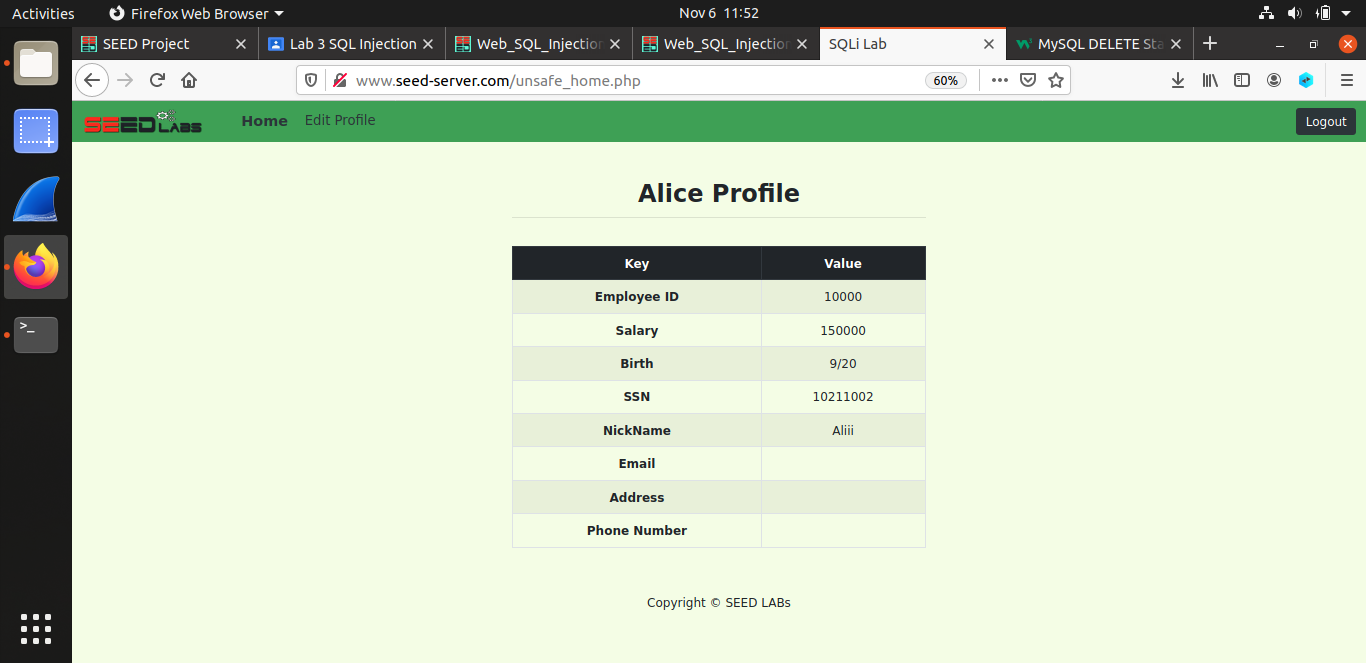
* **Task#3.1: Modify your own salary**

In order to modify Alice’s salary, we first log in to her account and then edit the profile. By entering “Aliii’, salary=150000 where ID=1 #”, in the nickname field and then hitting save, we are basically forcing the update query to also modify the salary. So the query will be something like:

UPDATE credential SET nickname=’Aliii’, salary=150000 where ID=1 # (the rest of the query is commented)

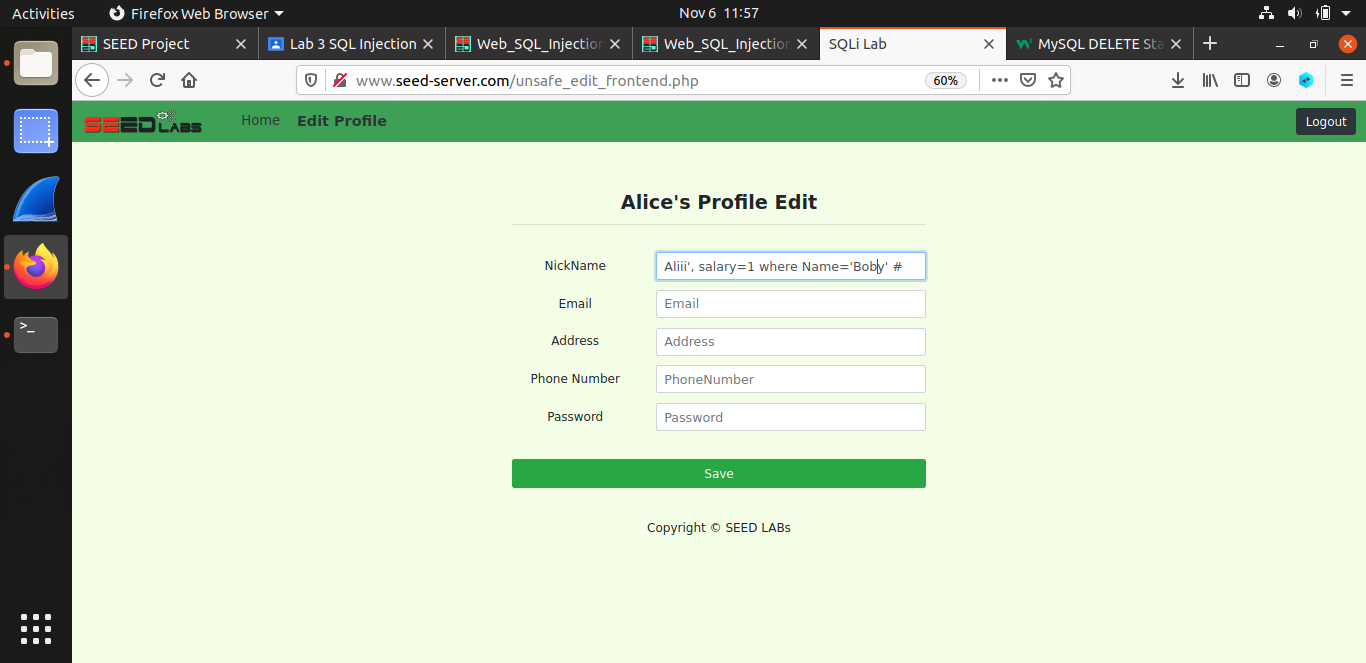
This perform the update action and the salary is successfully updated from 20000 to 150000.

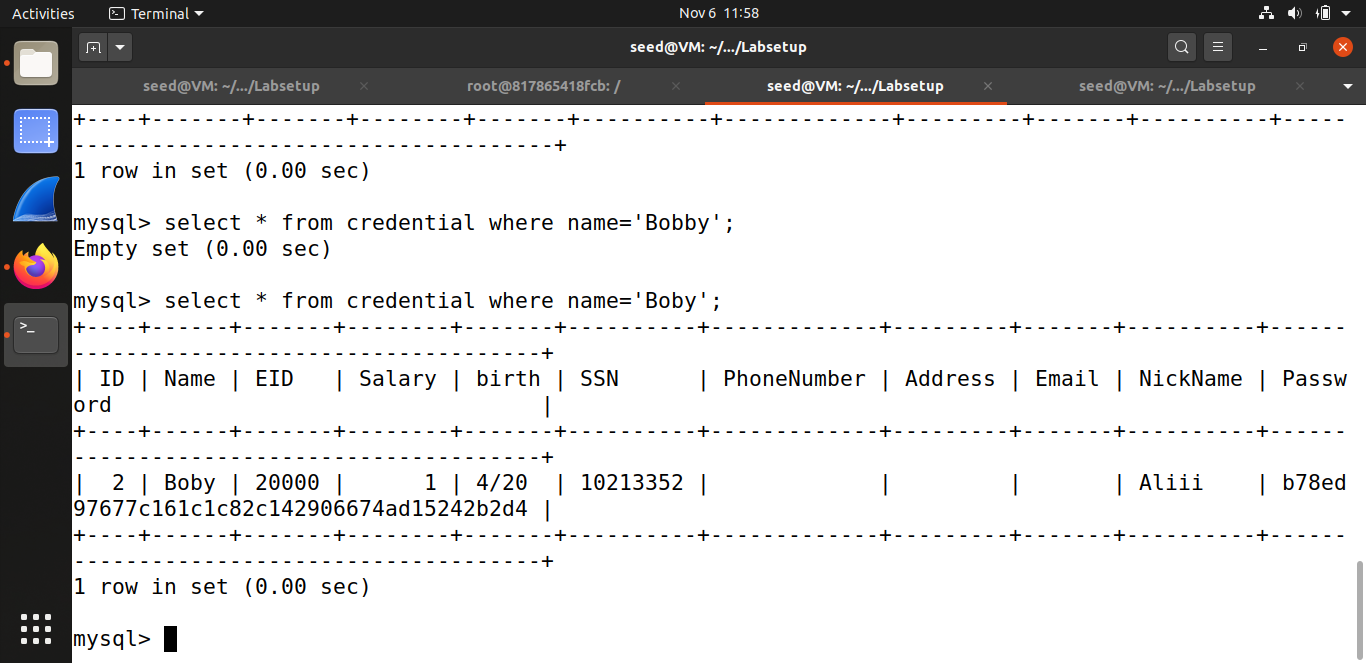




* **Task#3.2: Modify other people’ salary**

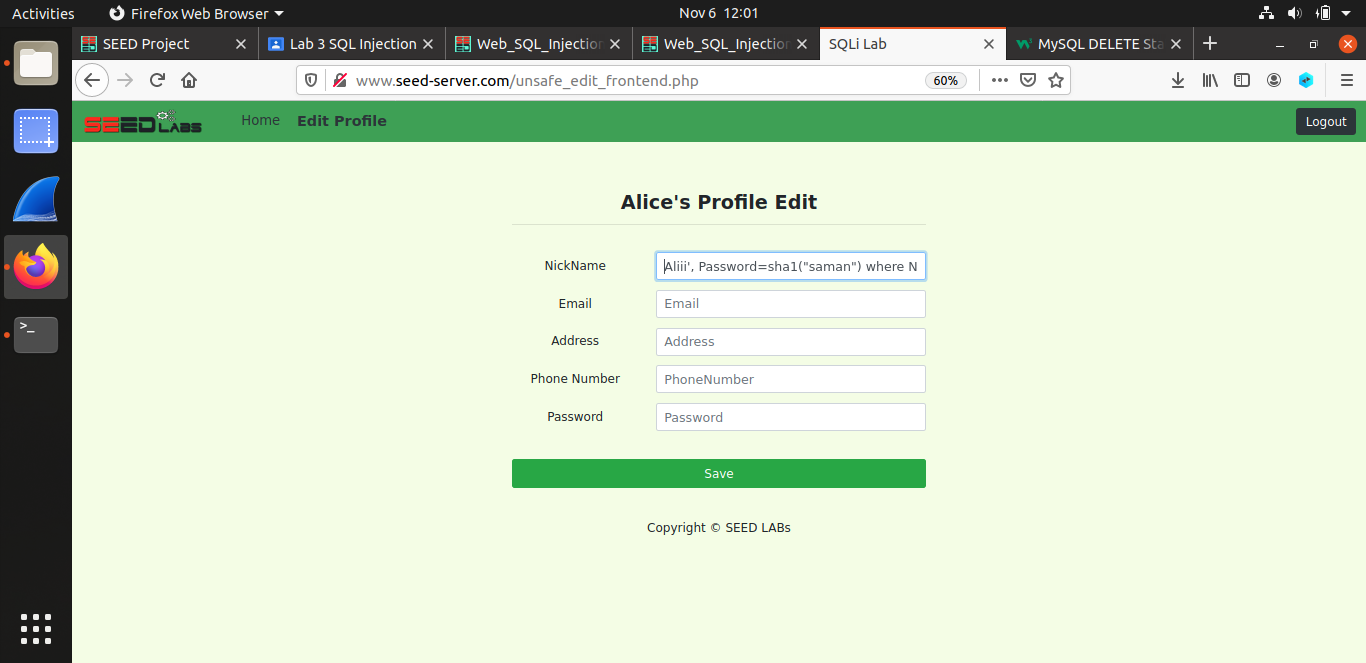
Using the same action as performed in the task above, but instead of using ID=1 we have used name=’Boby’ as we are now changing Boby’s salary. After saving the information we go to command prompt to check whether Boby’s salary has changed or not. We can see that Boby’s salary is now 1.

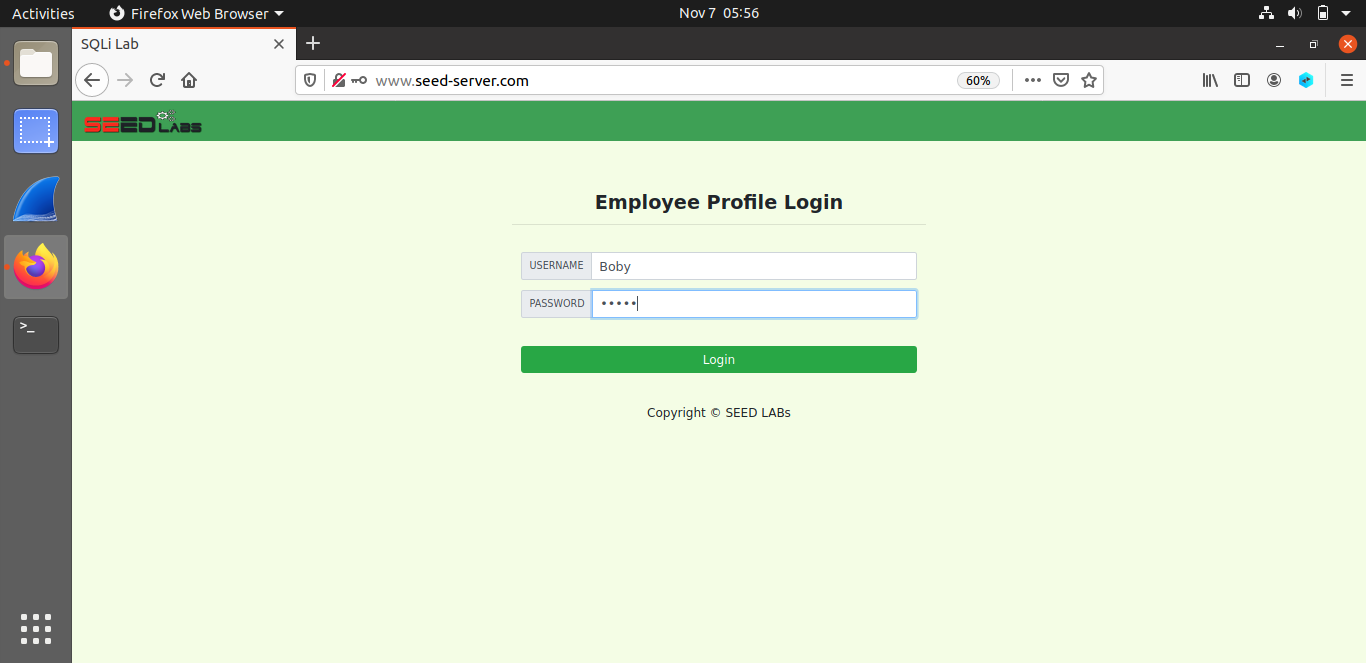


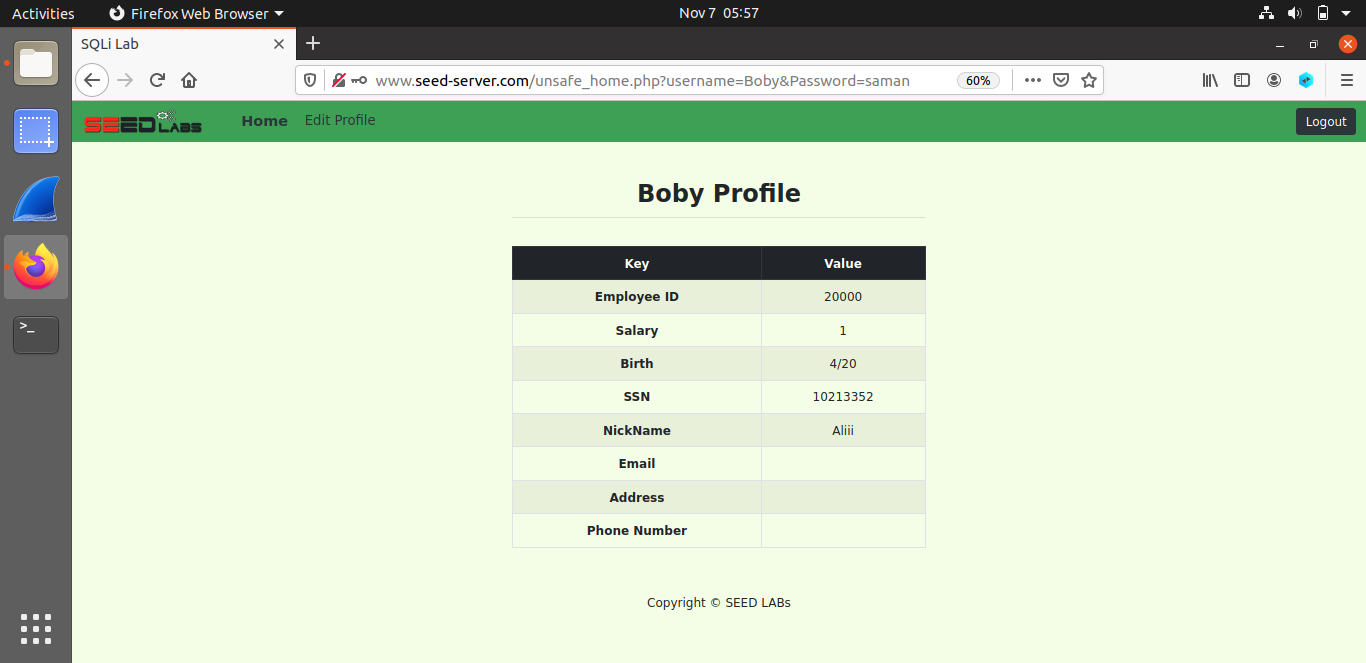


* **Task#3.3: Modify other people’ password**

Here same approach is used but since password is hashed we have used sha1() to hash the password passed. We then tried to log into Boby’s account using the new password. The output is as followed.

****

****

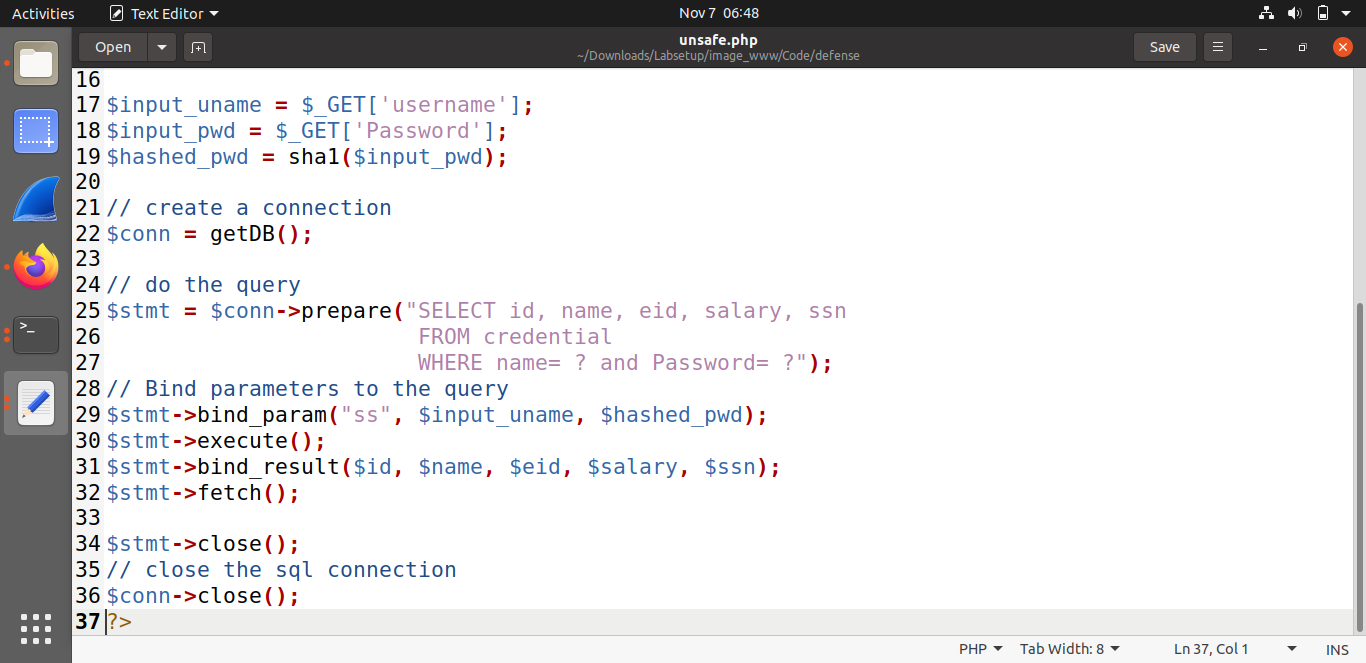
****

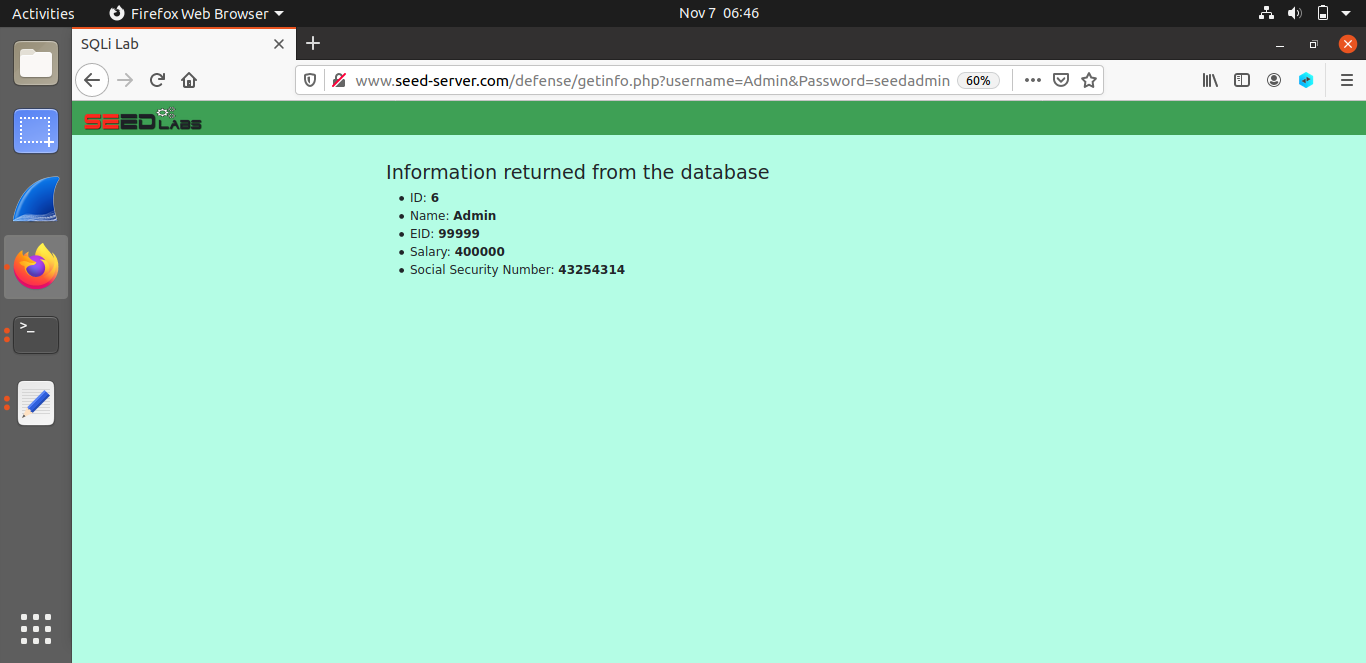
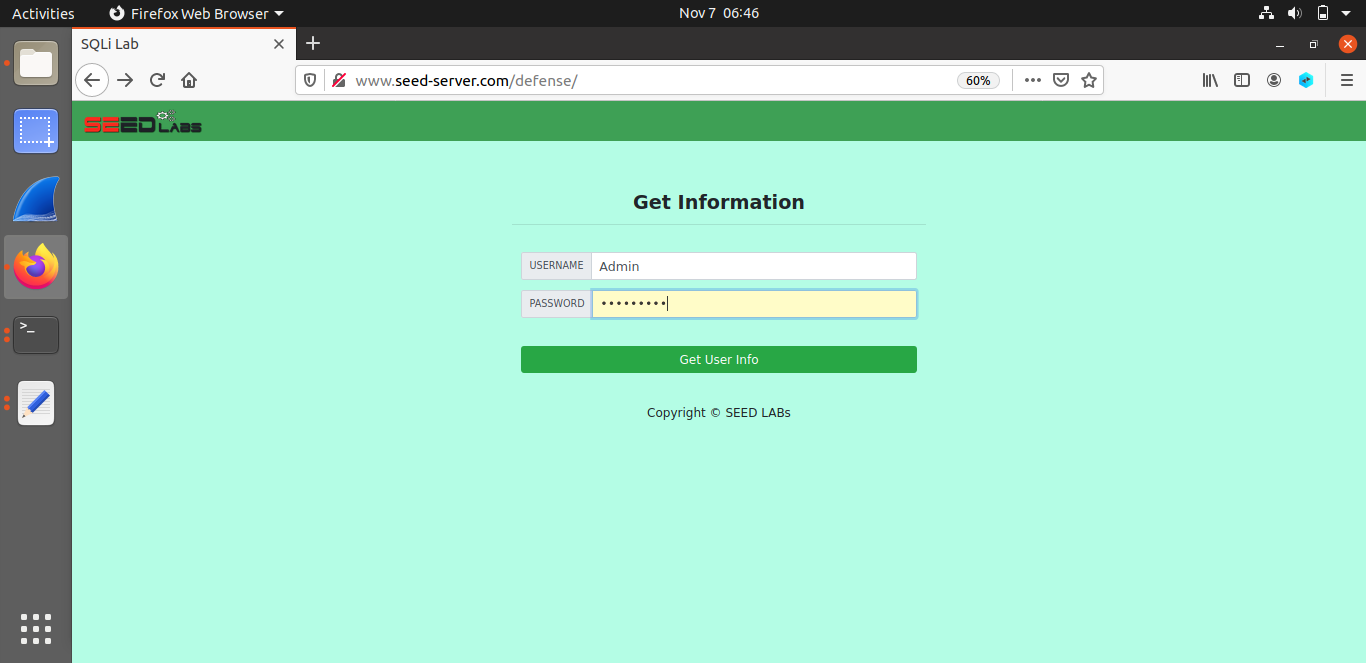
****

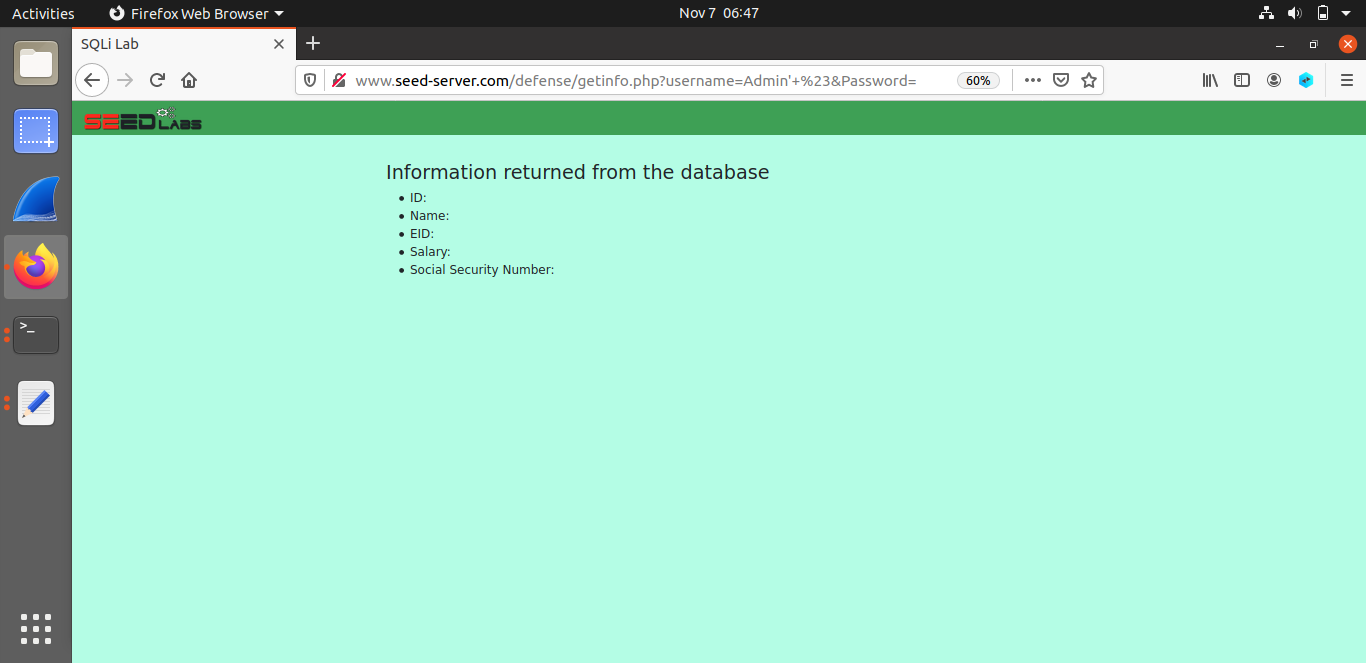
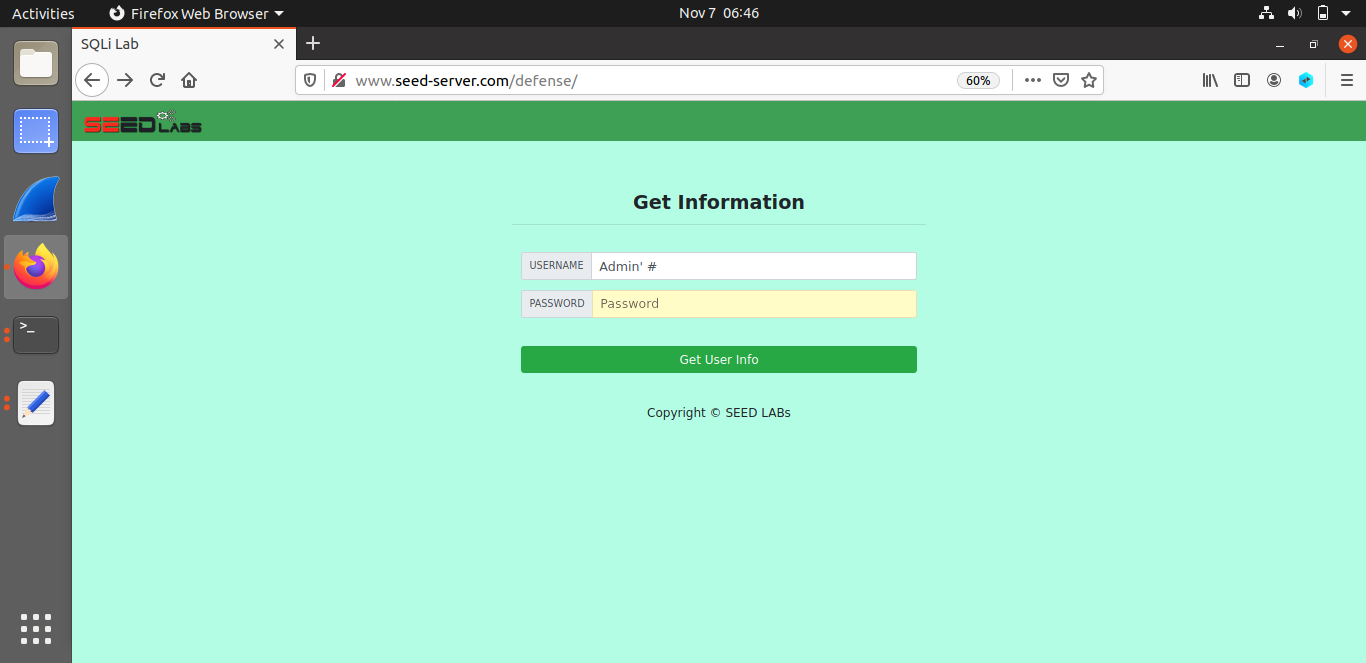
**TASK#04: Countermeasure — Prepared Statement**

In this task, the vulnerabilities encountered above are fixed. Instead of executing the query at compile time we are using the prepared statements. A prepared statement goes through the compilation step and turns into a pre-compiled query with empty placeholders for data. To run this pre-compiled query, we need to provide data to it, but this data will no more go through the compilation step; instead, it will get plugged directly into the pre-compiled query, and will be sent directly to the execution engine. Therefore, even if there is SQL code inside data, without going through the compilation step, the code will be simply treated as part of the data, without any special meaning. This is how prepared statements prevent SQLi attacks.

1. First I edited unsafe.php and stored it in defense folder. The output is:



Entering correct username and password rendered the complete information. 

By making use of same attack as in task#2.1, we can see that the page displayed no information.

1. Here we have rewritten unsafe\_home.php and unsafe\_edit\_backend to prevent the attacks made in task#2 and task#3. The output produced as a result of the same attacks made are:



