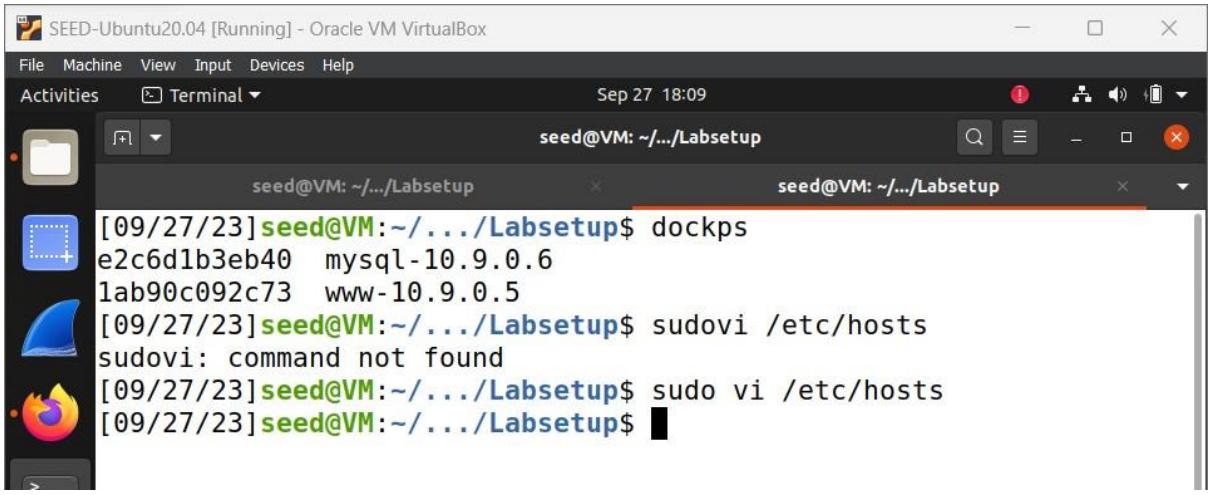
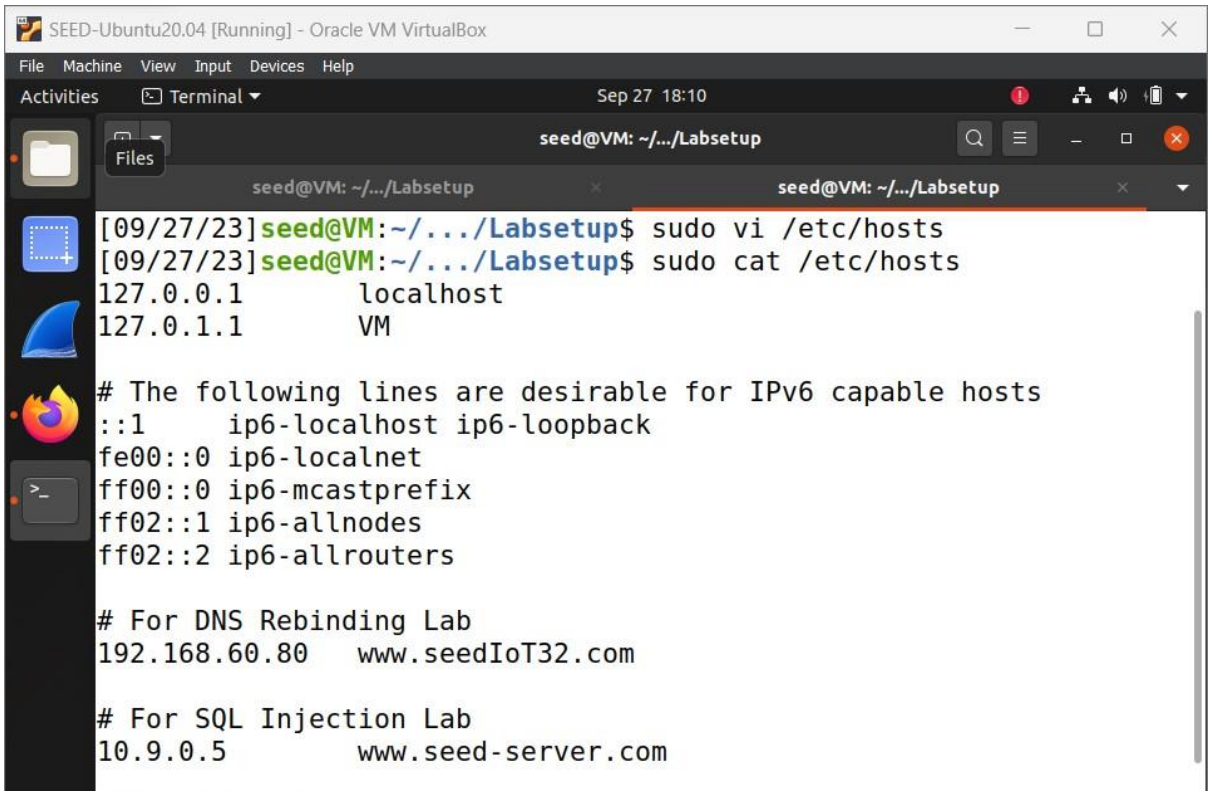
NAME: YASH SHETIYA

SUID: 9276568741

LAB SETUP:





TASK1:

Running basic mysql commands. First we go into the particular container and then login into sql.

We can either create databases or use the existing one. I chose to use an existing one and then performed basic commands to see the information.

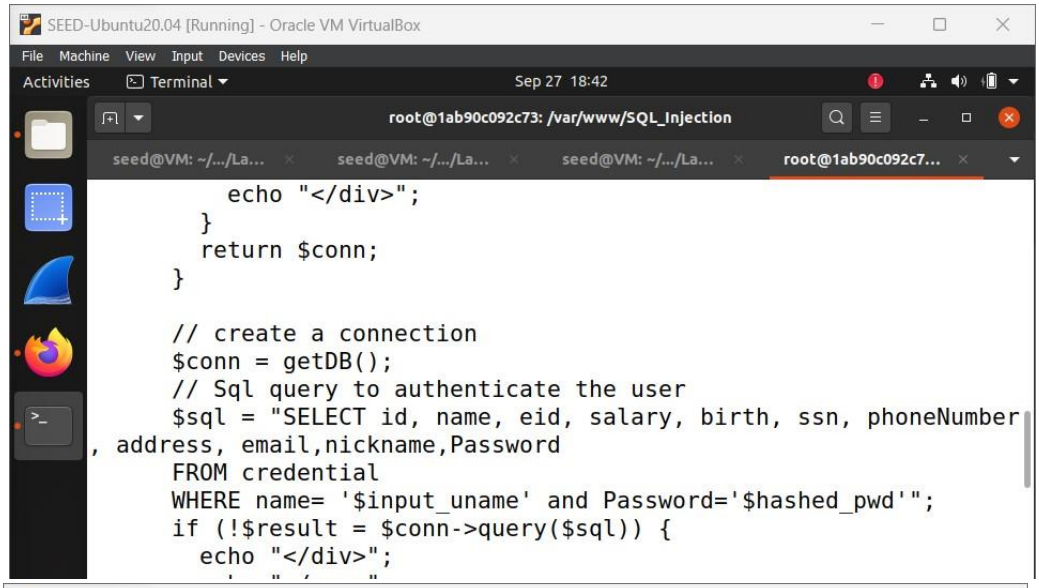
```
[09/27/23]seed@VM:~/.../Labsetup$ docksh e2
root@e2c6d1b3eb40:/# mysql -u root -pdees
mysql: [Warning] Using a password on the command line interface can
 be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.22 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
 statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
```

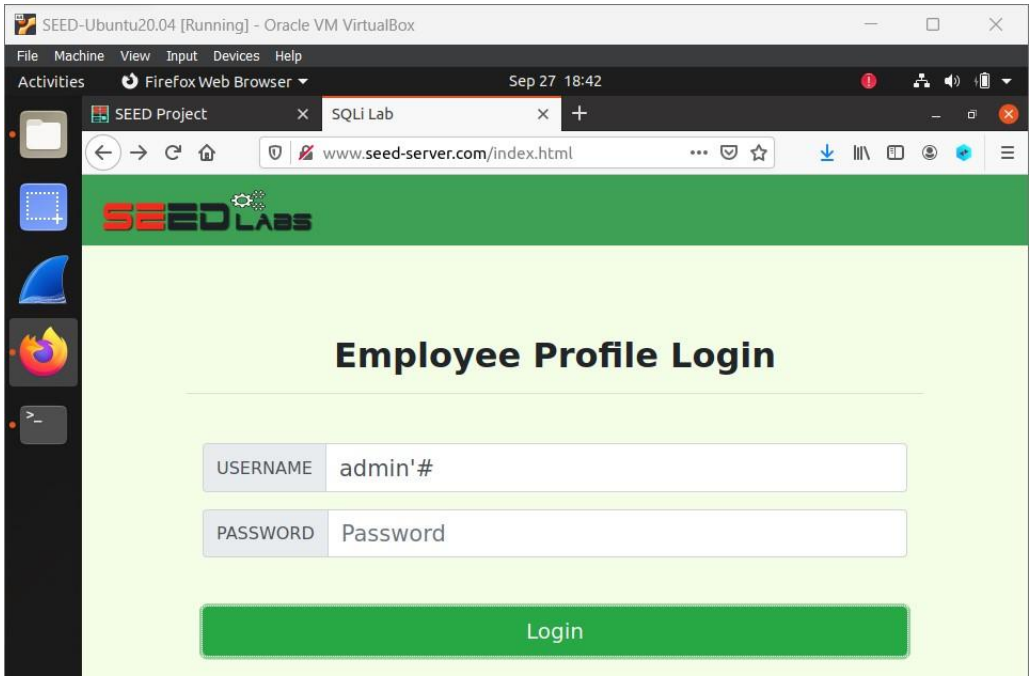
```
mysql> use sqllab_users;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+------------------------+
| Tables_in_sqllab_users |
+------------------------+
| credential             |
+------------------------+
1 row in set (0.00 sec)

mysql> select * from credential where Name = 'Alcie';
Empty set (0.01 sec)

mysql> select * from credential where Name = 'Alice';
+----+-------+-------+--------+-------+----------+-------------+---
------+-------+----------+----------------------------------------
-+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Ad
dress | Email | NickName | Password
 |
+----+-------+-------+--------+-------+----------+-------------+---
------+-------+----------+----------------------------------------
-+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |
      |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976
```

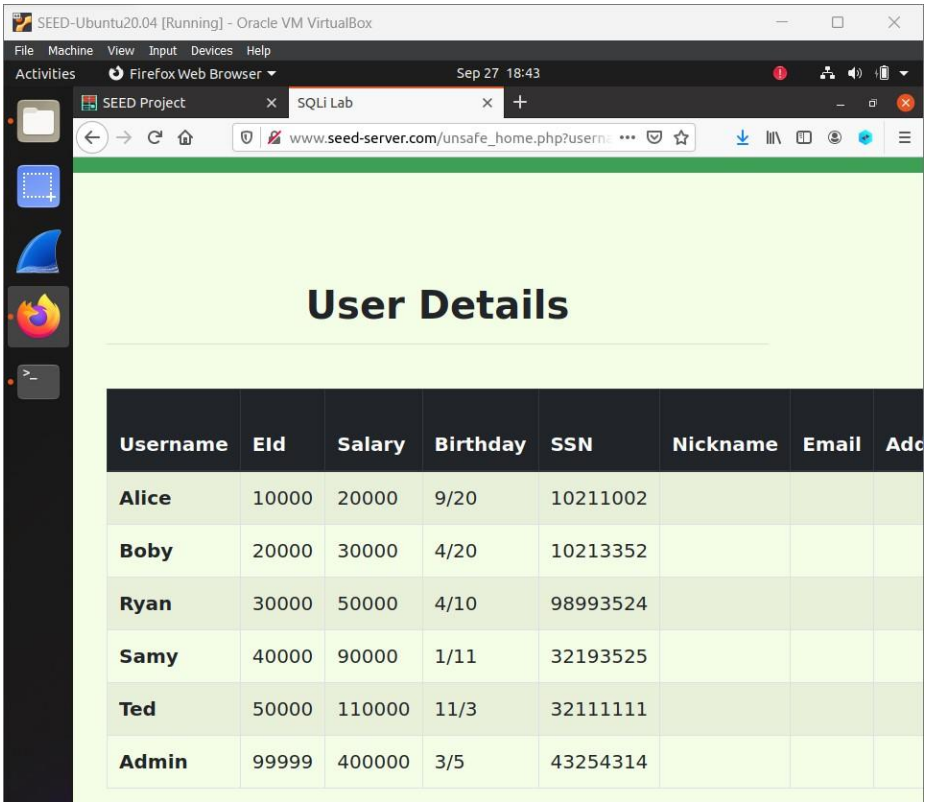
TASK 2

/// 2.1 ///

If we notice properly in the php file we can see that username and password are obtained using GET method. The password used is the sha1 value of the password entered by the user and not the password. So we can add (‘) to close the quoation marks and add (#) to comment out the content.







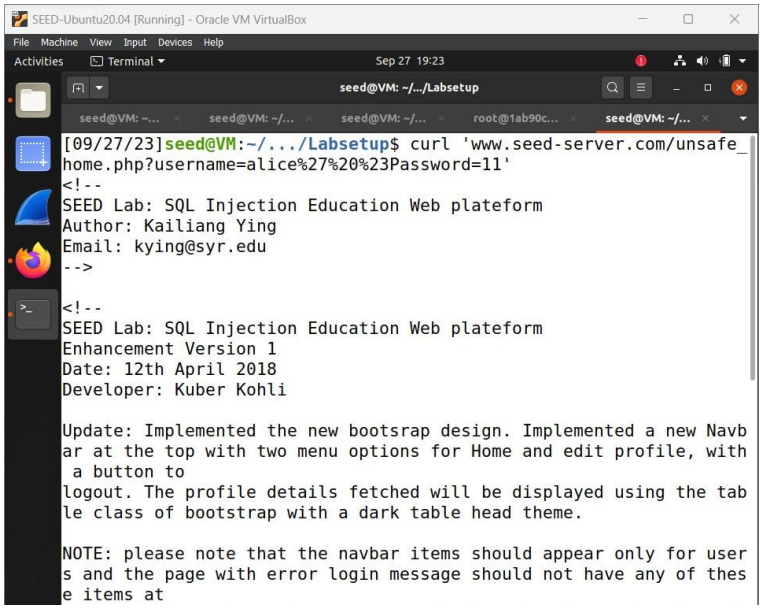
/// 2.2 ///

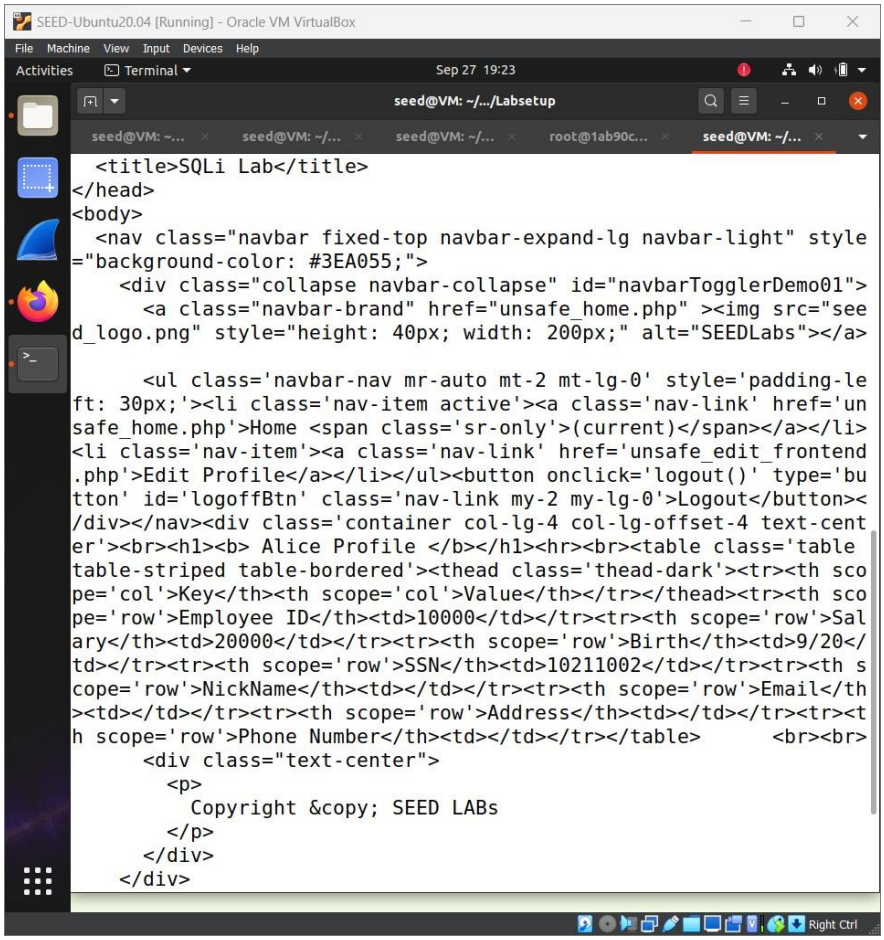
It should be noted that the HTTP request here needs to have special characters to be URL encoded.

= %23, & = %26, ‘ = %27

We need to handle HTTP encoding while sending request using curl. Once done we could see that Alice’s information was visible to us.

We used the following curl command that can be seen in the screenshot below to place a request to the website and login with the details provided.



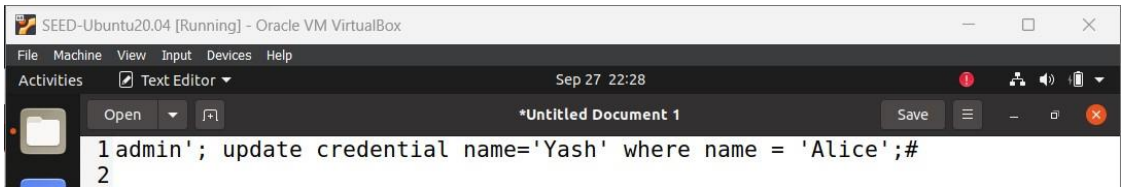


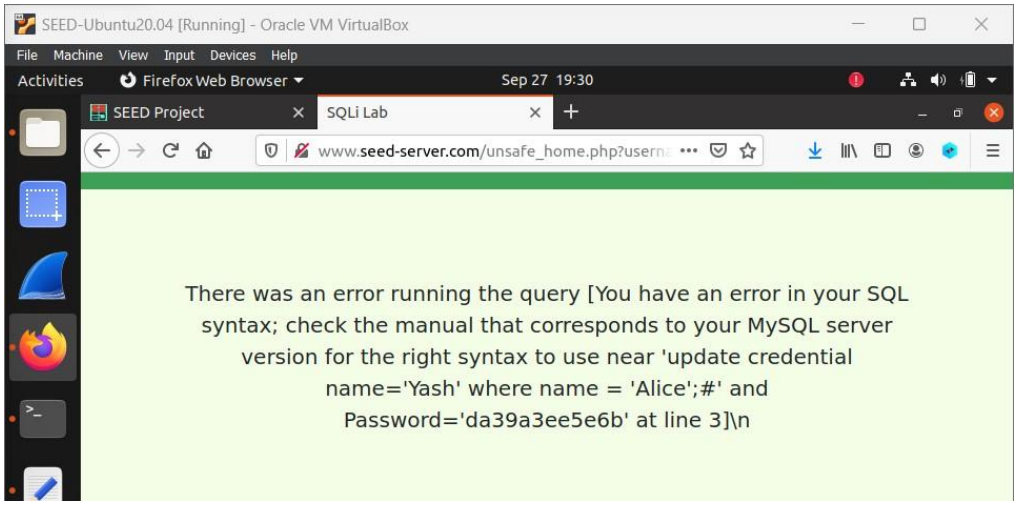
Here we can view details of Alice's profile and can verify that it is Alice and not somebody else.

/// 2.3 ///

Here we try to inject two commands in succession and we separate the two commands using (;).

This particular line we add in the Username cell.





This is happening because we have some kind of a syntax error. The erro being a multiquery error, we get this error because in our mysqli extension multiple queries are not allowed to run in the database server. This can be changed if we modify it to mysqli ->multiquery(). Not enabling multiquery in that field where in it uses GET request isnt a good praactice when we see it through a security perspective.

TASK 3

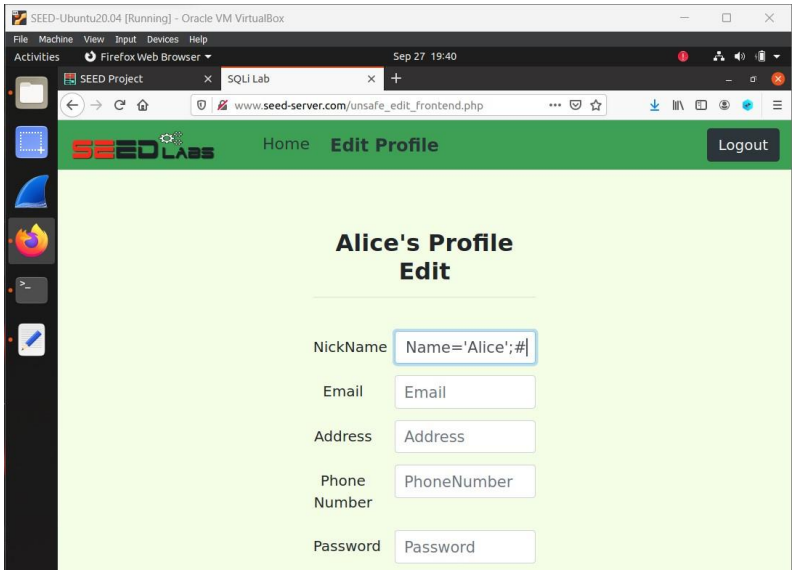
/// 3.1 ///

When we view the edit profile page, it can bee seen that we can only update our nickname, email, address, phone number and pasword. We are not given the privilege to update the salary field.

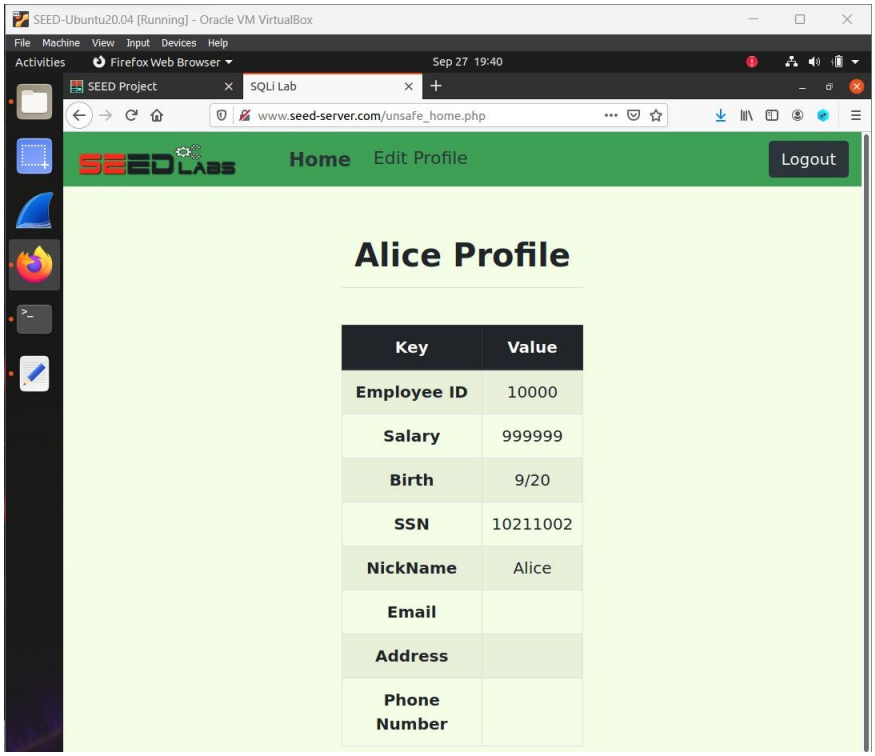
Suppose we are Alice and we want to change the salary field by exploiting the injection vulnerability. We can achieve our goal as follows:

First we login into Alice's account and then we can see Alice's profile

Now the salary here is (20000) which we mean to change to (999999). For this we go to edit profile and pass the following command: { *Alice' , Salary='999999' where Name='Alice';#* } in the nickname field.

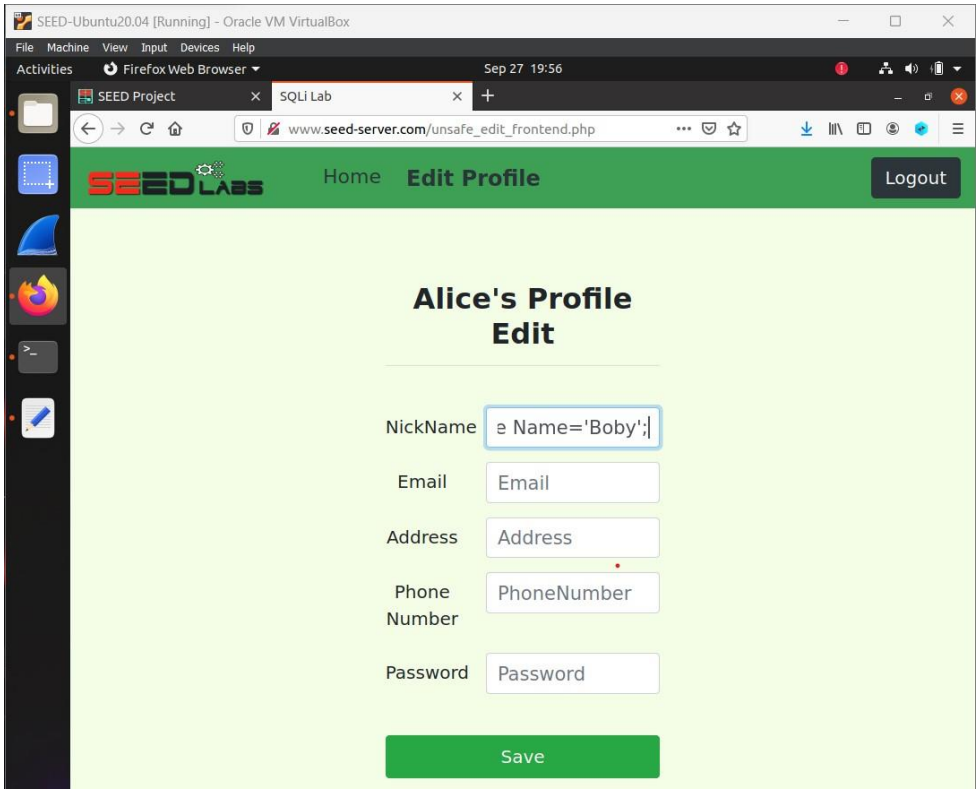


When we save these changes we can see that Alice's salary has been modified:

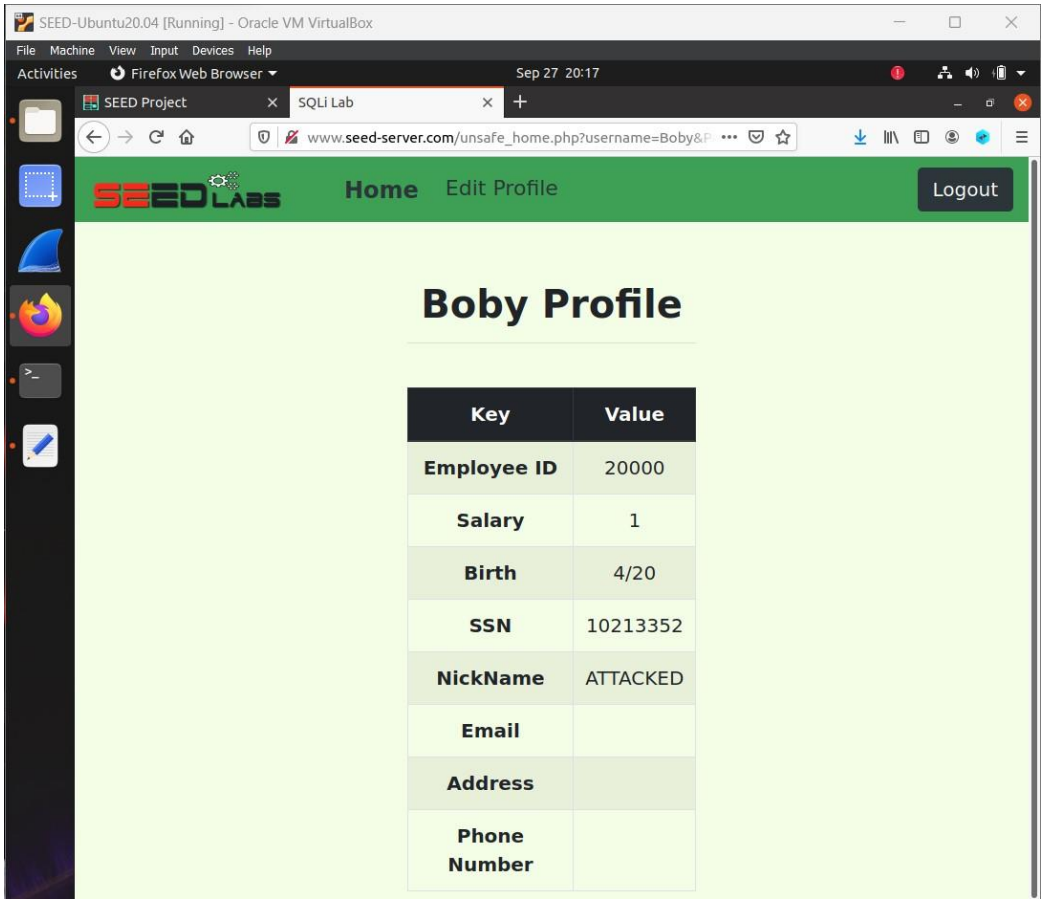


/// 3.2 ///

Now we want to change other's salary. Considering Boby to be Alice's Boss, she wants to punish boby by reducing his salary to a dollar. We pass the following command through Alice's account to update the salary of Boby: { *Attacked' ,Salary='1' where Name='Boby'#* }.



After saving these changes lets check Boby's profile, we can see that his salary has been changed to 1$.

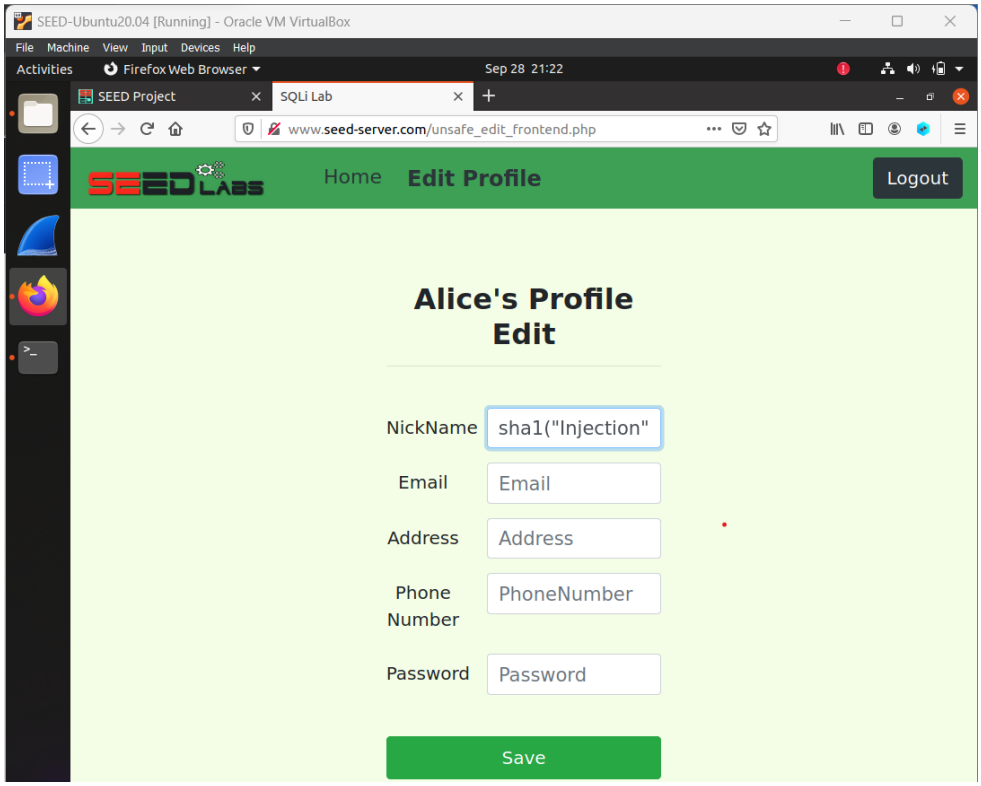


/// 3.3 ///

The query is : *{ Boby', Password=sha1("Injection") where name='Boby'# }*

We login into Alice's accountwho wants to cause damage to Boby and from her profile we put in our query.

One more way to go forward is that you could use any sha1 tool which will let you know the encrypted value and can be simply put as *password= (encrypted value).*

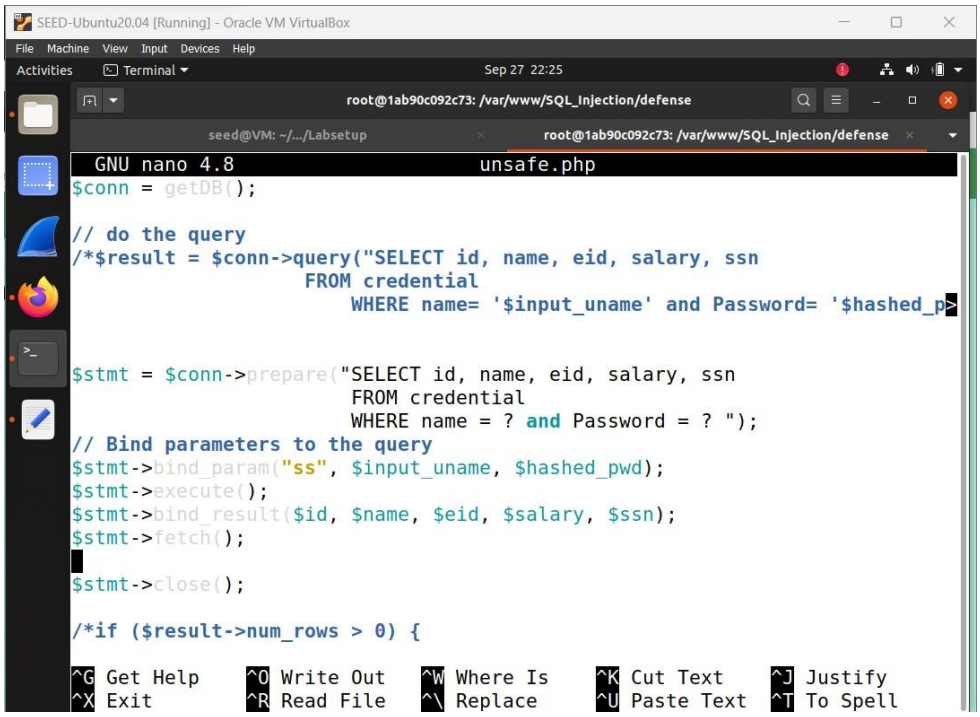


The password isnt stored in text but is passed through sha1 function. So we use the sha1 functionitself in our query. The databse is storing the hash value of passwords and it can also be seen in the unsafe_edit_backend.php.

Now when we check logging in as Boby with our updated password, it works and it can be seen below.

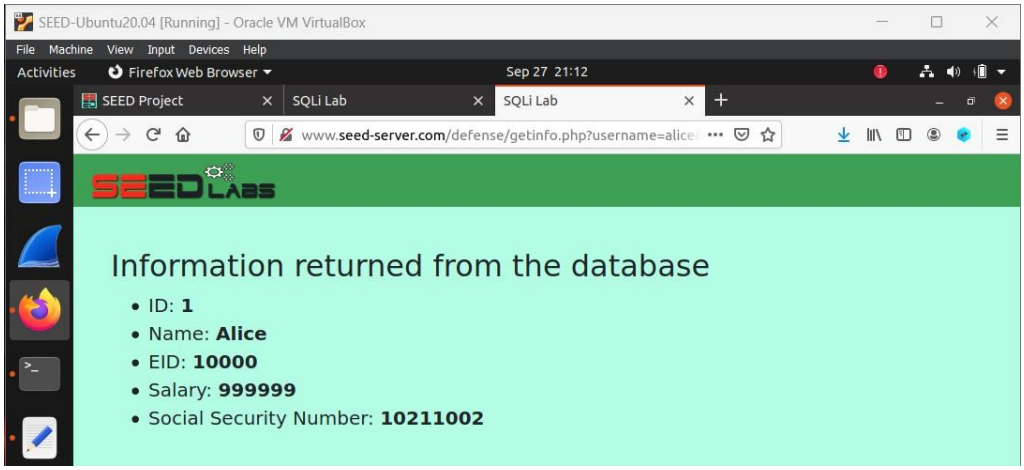
TASK 4:

For this task we need to modify the *unsafe.php*

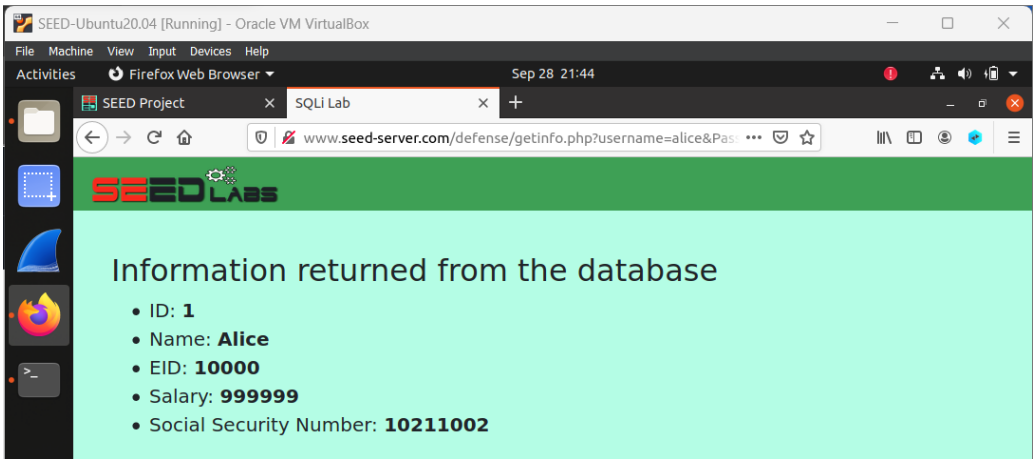


Here we can see that the original code has been commented out and our modificationshave been brought in.

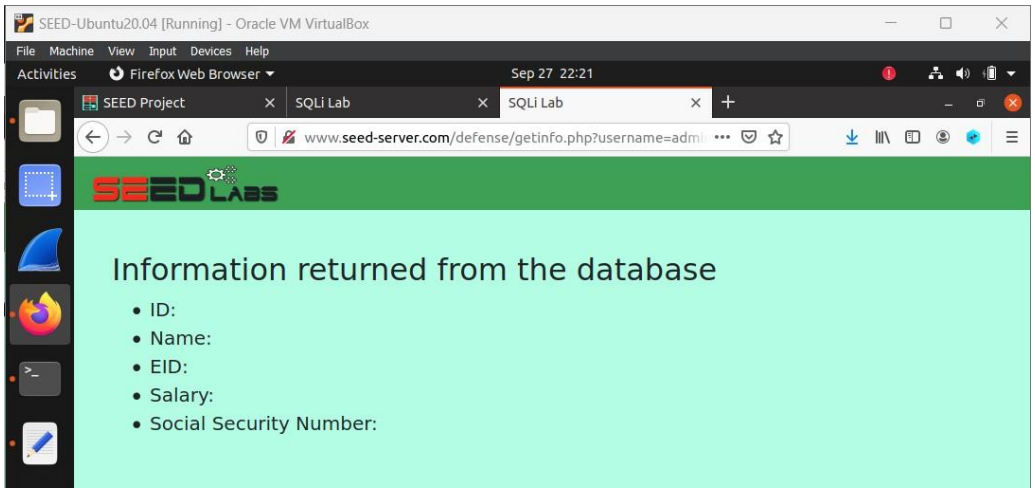
Before I made the changes the site would show the following details when logged in:



Now logging in using username and password: it can be seen that information is received



Now logging in using the bypass method:



From the above screenshot we can see that after modification when we tried to login using the bypass method we were not able to receive the information.

As per our modification the information is converted into string. Now the code and data are separated. With the changes we can say that SQL injection attack was avoided.