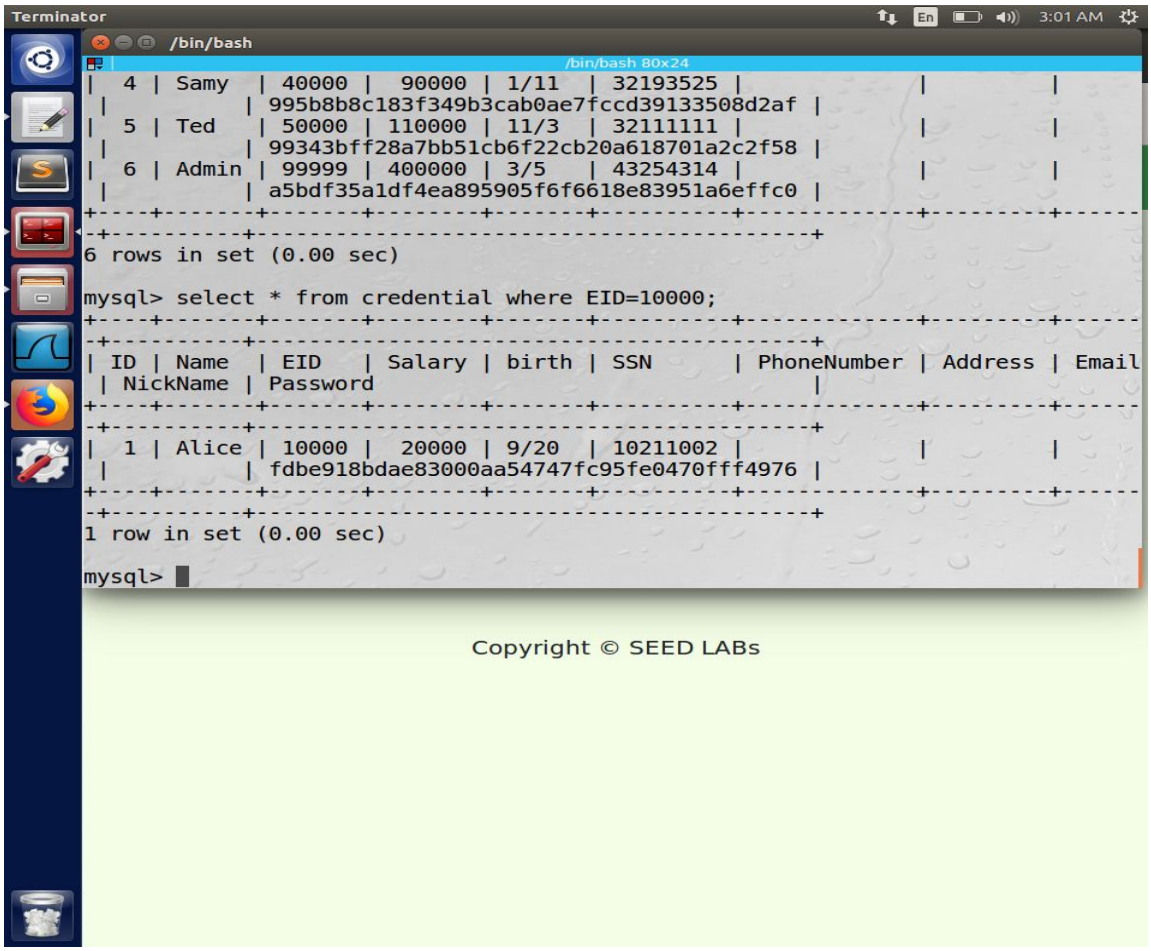
Ethan Booker

SQL Injection Attack Lab

Spring 2020

Description of overall goals of lab:
To learn how a website that uses an SQL database could be vulnerable to an attack called SQL injection, see the damage of an SQL injection, and to see how to defend such an attack.

3.1, Task 1
Description: Get familiar with SQL statements
Evidence:

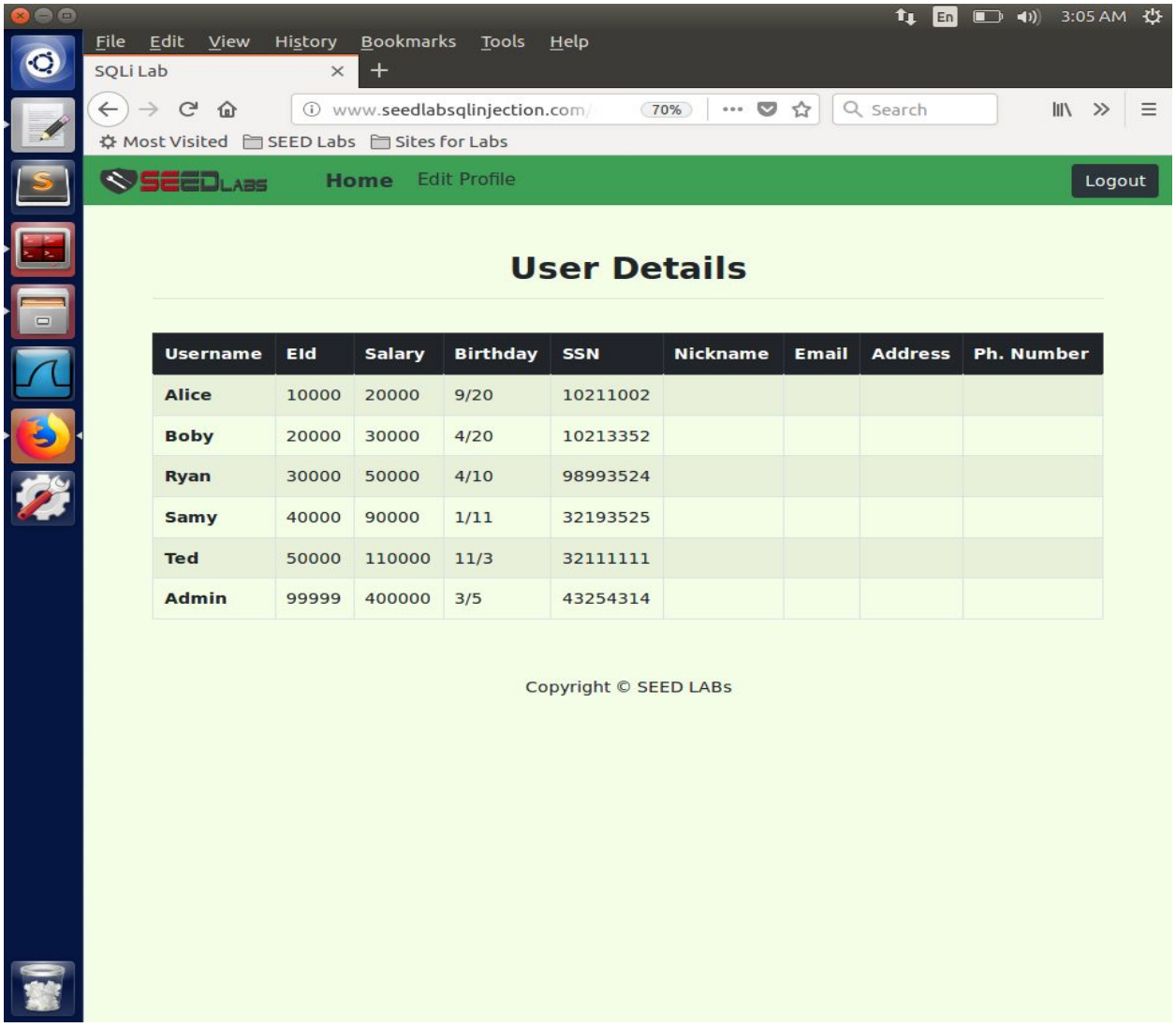


Analysis: These are simple SQL query statements and not much to say. The query used was “select * from credential where EID=10000;”

3.2, Task 2.1

Description: SQL Injection Attack on SELECT Statement

Evidence:



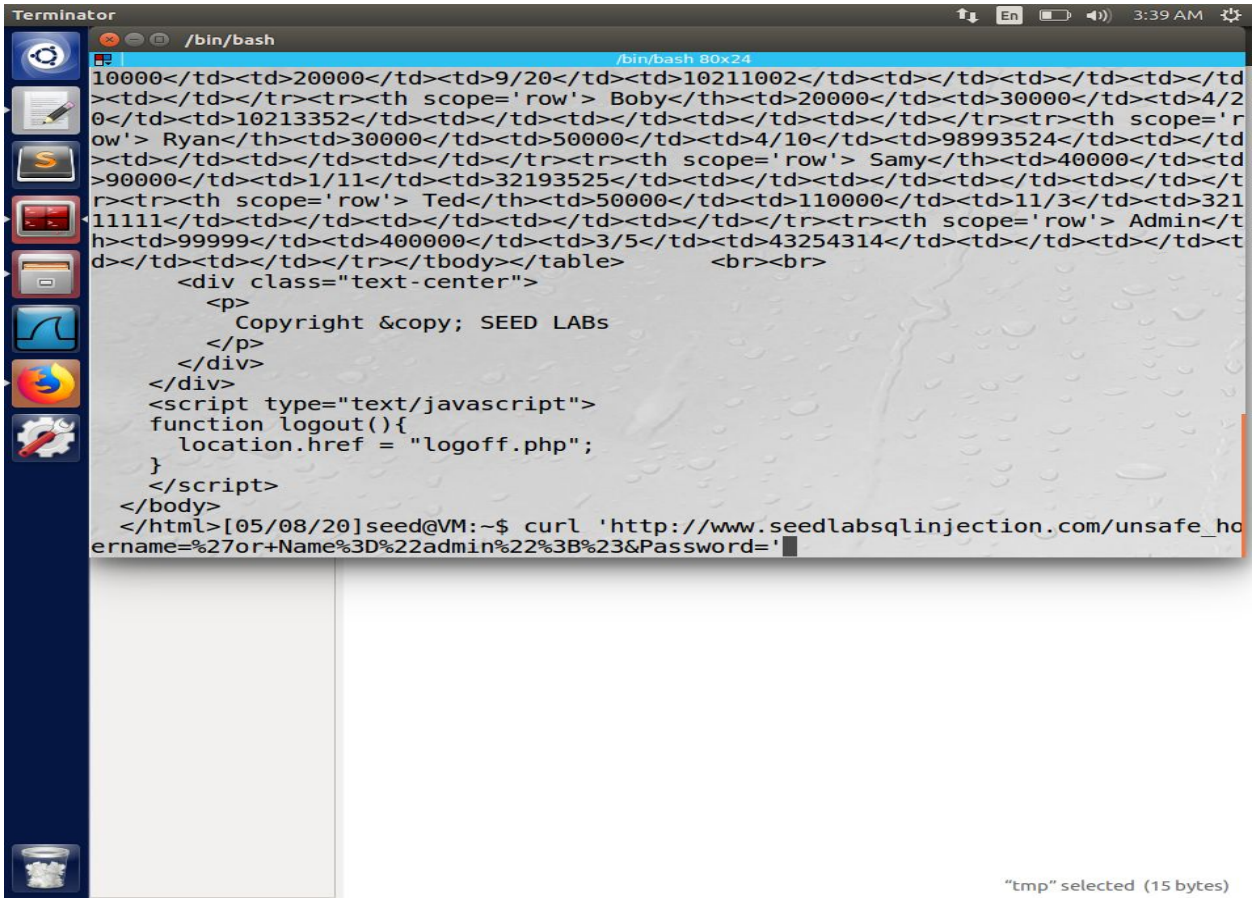
Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

Analysis: In the Username text box I typed in ‘ or Name=”admin”;# since the single quote has special meaning in a normal statement then I am able to inject my own statements in there then get rid of any other statements afterwards using the comment symbol #. I already knew that Username mapped to the name column so I can specifically look for a name that is admin.

3.2, Task 2.2

Description: SQL Injection Attack from command line

Evidence:

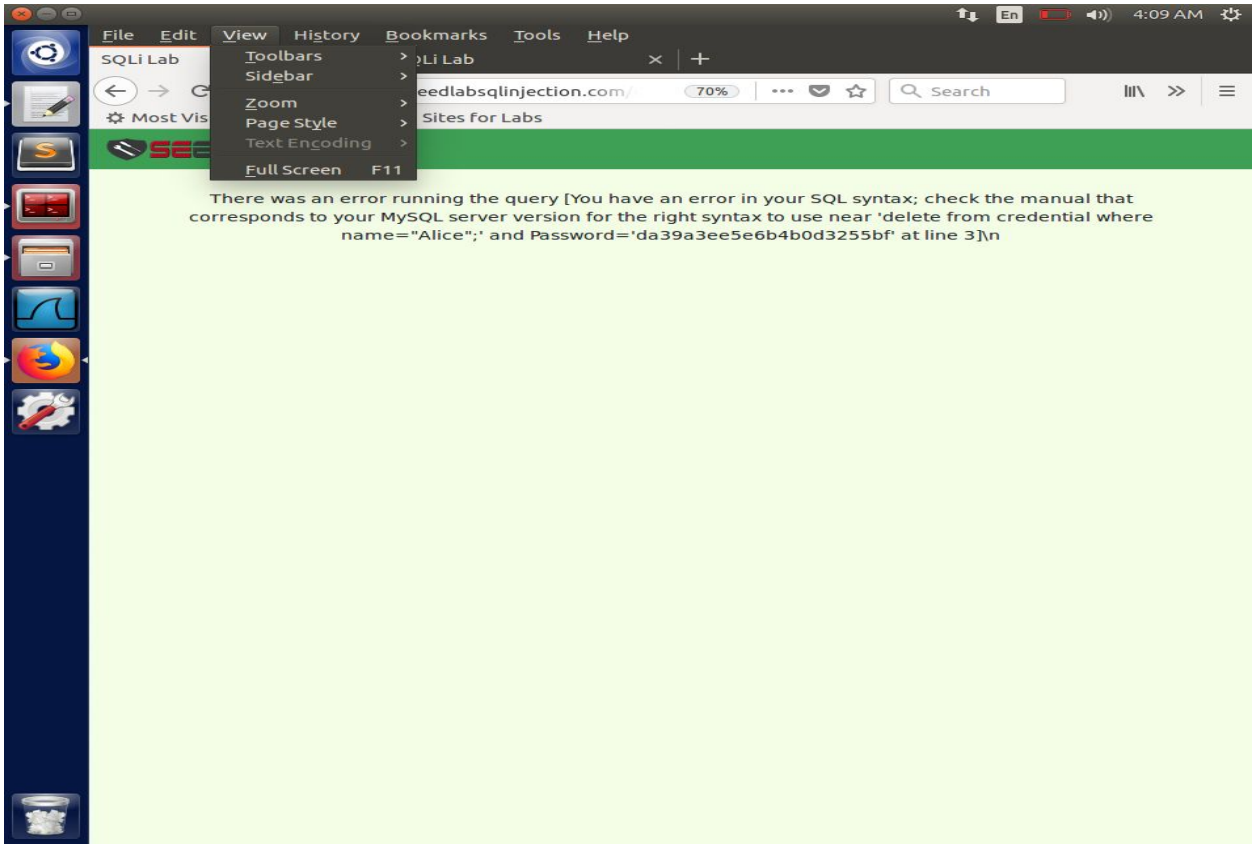


Analysis: I was able to obtain the url through the website's current url header. By doing that, I was able to call curl and mimic the same attack but through the command line.

3.2, Task 2.3

Description: Append a new SQL statement

Evidence:

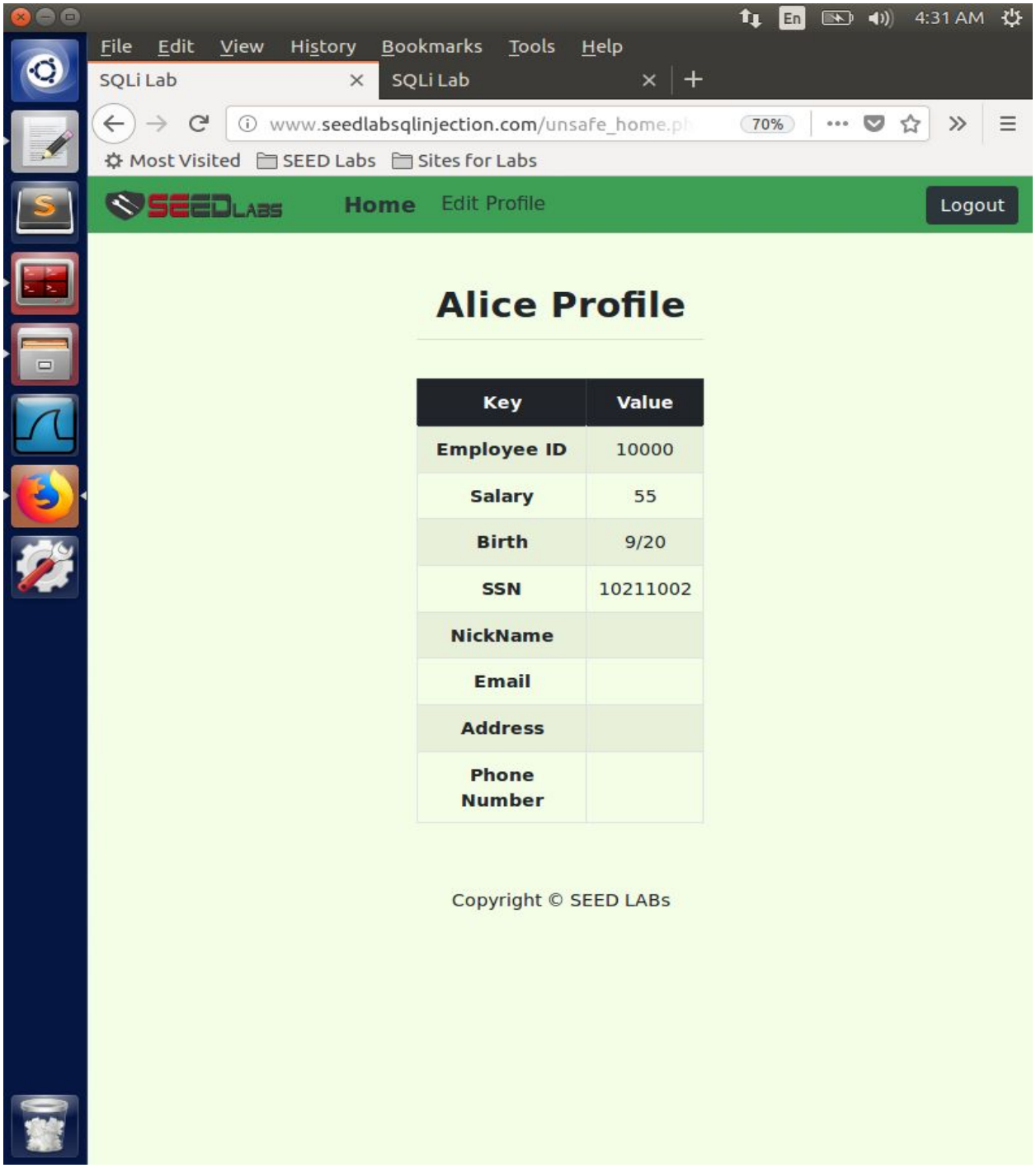


Analysis: I got an error when I tried to append another SQL statement after my first statement in logging in. This happened because after the first statement (‘ or 1=1;) is executed, we get a syntax error due to how SQL compiles the query. So even after adding a # after my second statement there will still be an error because you cannot execute two statements in one query.

3.3, Task 3.1

Description: Modify Alice's salary

Evidence:

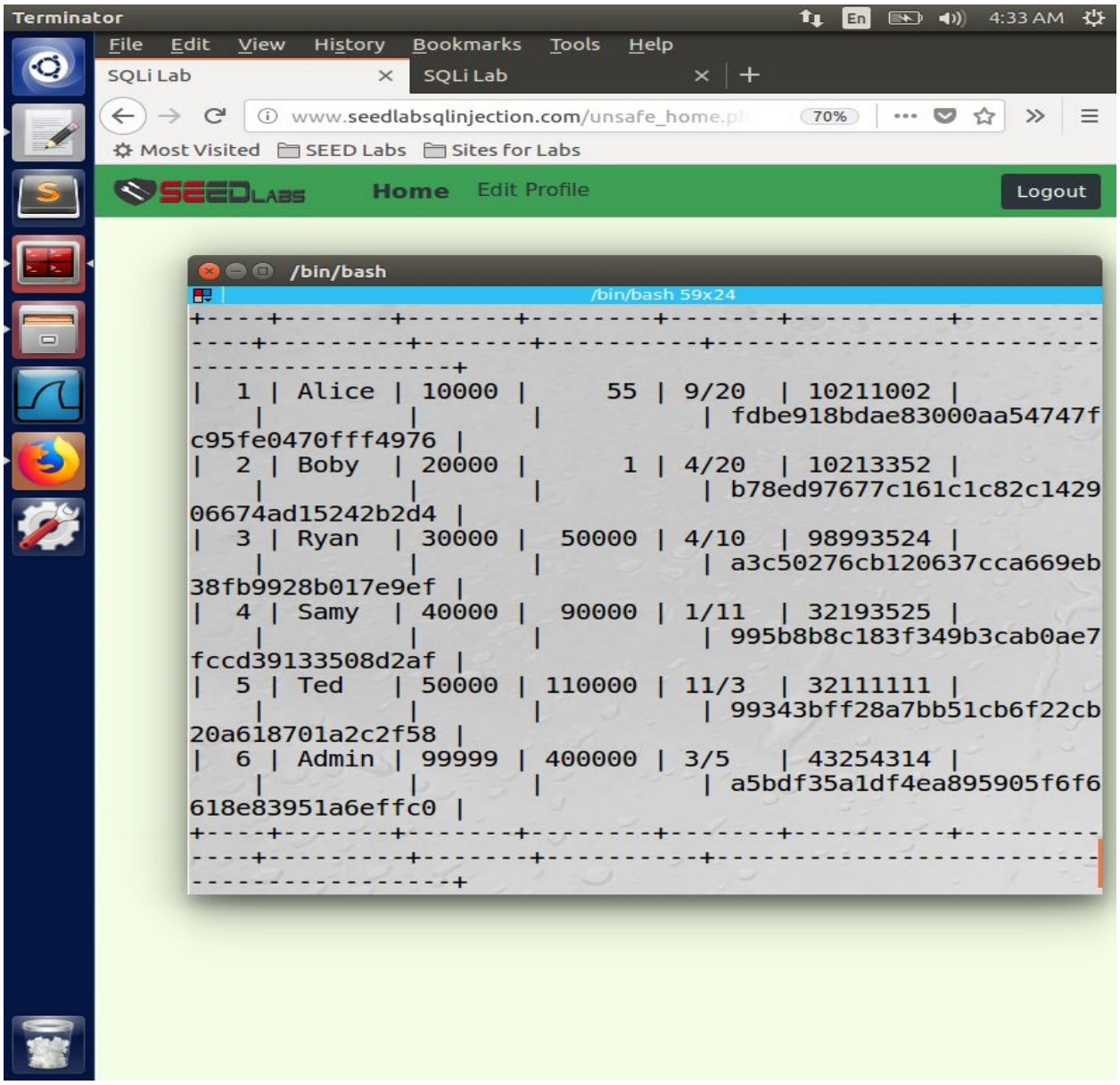


Analysis: In the NickName text box, I inputted ', Salary="55" where Name="Alice";# .Due to how the query was set up, I used a comma then I set which attribute I wanted to change then added the where statement to select a specific row resulting in Alice's salary being updated.

3.3, Task 3.2

Description: Modify Boby's salary to 1.

Evidence:

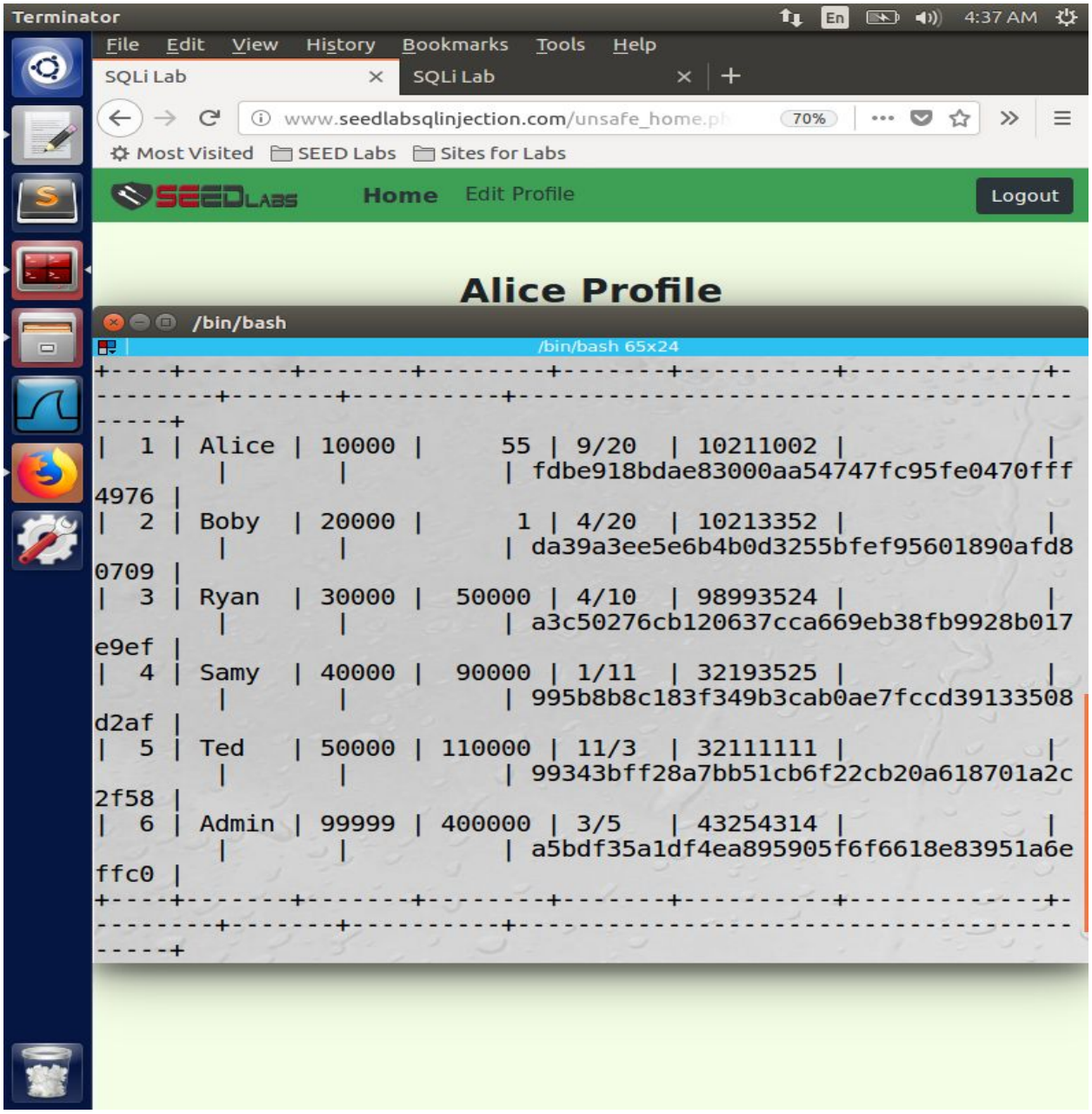


Analysis: Similar to task 3.1, I just changed the query to ', Salary="1" where Name="Boby";# and this changed Boby's salary. I showed the database to show the change. I did not need to leave Alice's edit profile in order to do this.

3.3, Task 3.3

Description: Modify Boby's password

Evidence:

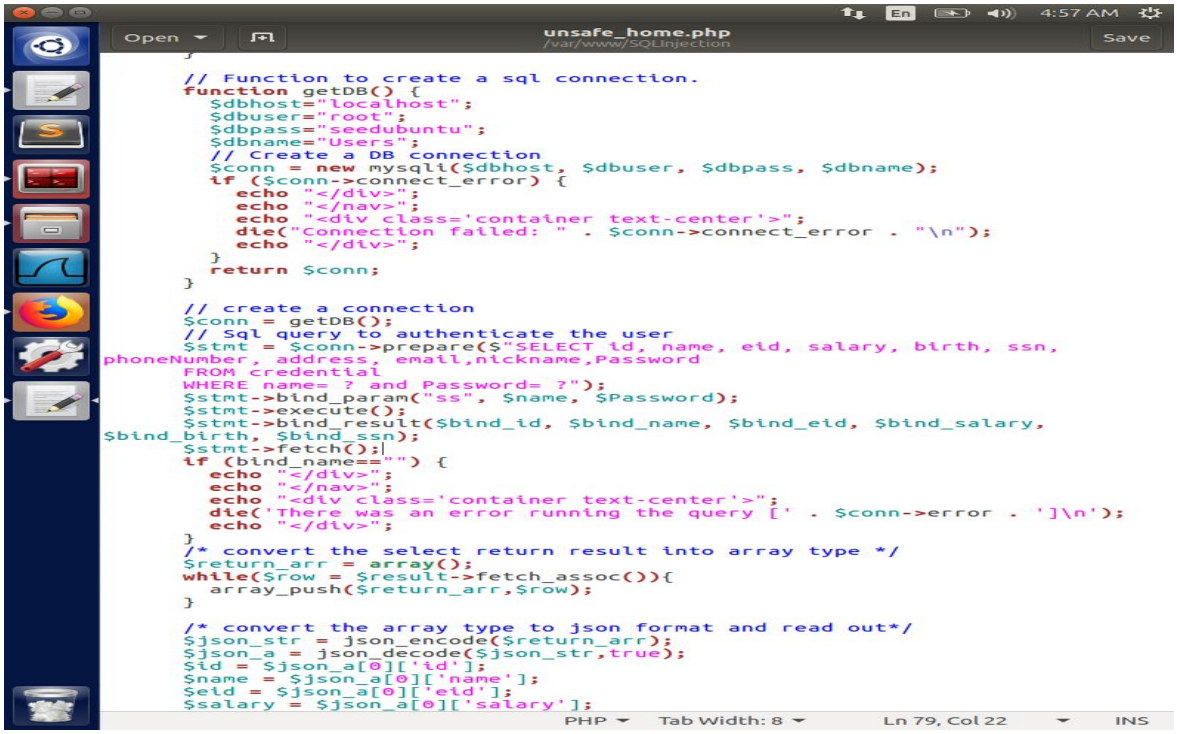


Analysis: This task is similar to task 3.2, but rather than saying Salary, I used Password="...". The hash is different and from task 3.2 for Boby. How I obtained the hash was due to opening the terminal and typing echo test | openssl sha1.

3.4, Task 4

Description: Countermeasure - prepared statement

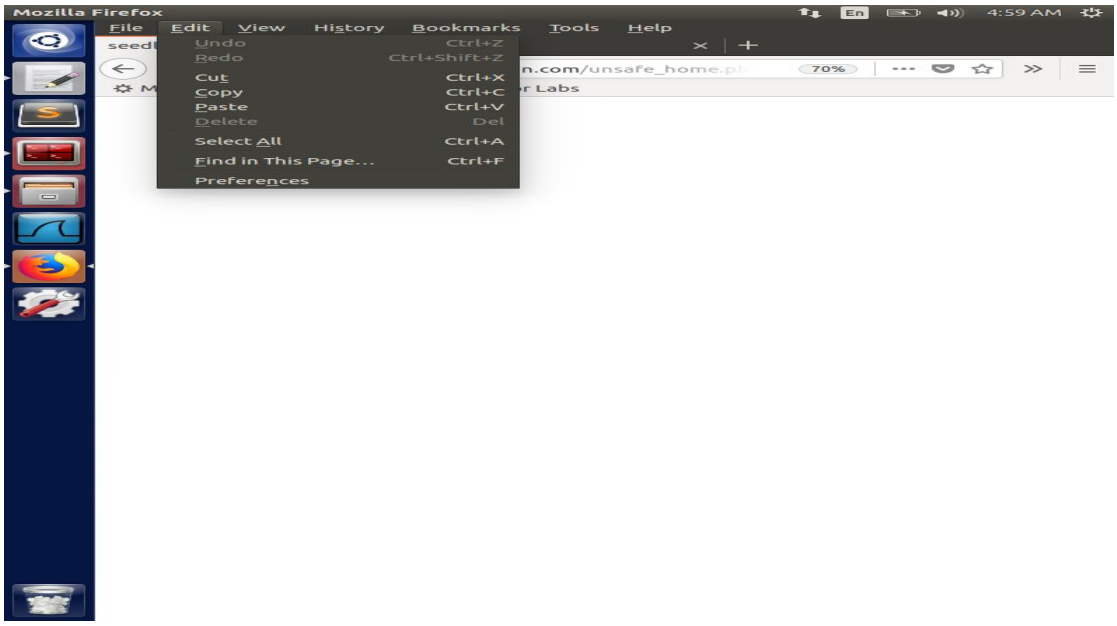
Evidence:




```
        // Function to create a sql connection.
        function getDB() {
          $dbhost="localhost";
          $dbuser="root";
          $dbpass="seedubuntu";
          $dbname="Users";
          // Create a DB connection
          $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
          if ($conn->connect_error) {
            echo "</div>";
            echo "</nav>";
            echo "<div class='container text-center'>";
            die("Connection failed: " . $conn->connect_error . "\n");
            echo "</div>";
          }
          return $conn;
        }

        // create a connection
        $conn = getDB();
        // Sql query to authenticate the user
        $stmt = $conn->prepare($"SELECT id, name, eid, salary, birth, ssn,
phoneNumber, address, email,nickname,Password
        FROM credential
        WHERE name= ? and Password= ?");
        $stmt->bind_param("ss", $name, $Password);
        $stmt->execute();
        $stmt->bind_result($bind_id, $bind_name, $bind_eid, $bind_salary,
$bind_birth, $bind_ssn);
        $stmt->fetch();
        if (bind_name=="") {
          echo "</div>";
          echo "</nav>";
          echo "<div class='container text-center'>";
          die('There was an error running the query [' . $conn->error . ']\n');
          echo "</div>";
        }
        /* convert the select return result into array type */
        $return_arr = array();
        while($row = $result->fetch_assoc()){
          array_push($return_arr,$row);
        }

        /* convert the array type to json format and read out*/
        $json_str = json_encode($return_arr);
        $json_a = json_decode($json_str,true);
        $id = $json_a[0]['id'];
        $name = $json_a[0]['name'];
        $eid = $json_a[0]['eid'];
        $salary = $json_a[0]['salary'];
```



Analysis: By changing how the sql statement is executed by using a prepared statement in order to send only the query part without the data then the ? marks are replaced with the actual binded parameter data. The prepared statements prevent special meanings from occurring with characters as data inputs. As for the second image, this is due to me testing the login page with

the successful ‘ or Name=”admin”;# but it didn’t work this time and showed a white screen due to the if ($bind_name=””) … as this gave an error. Perhaps if this was correct then the correct screen would show up, but regardless the attack did not work anymore.

Conclusion: In this lab I learned how to apply some SQL commands through a website and through the terminal to create an SQL Injection. I have a better understanding on how to test if a website is using SQL by trying to execute a simple SQL injection statement. Most importantly, I learned how to defend against this attack using prepared statements and I have a better understanding as to why an SQL Injection is possible when they could easily just use a prepared statement or sanitize the input data.