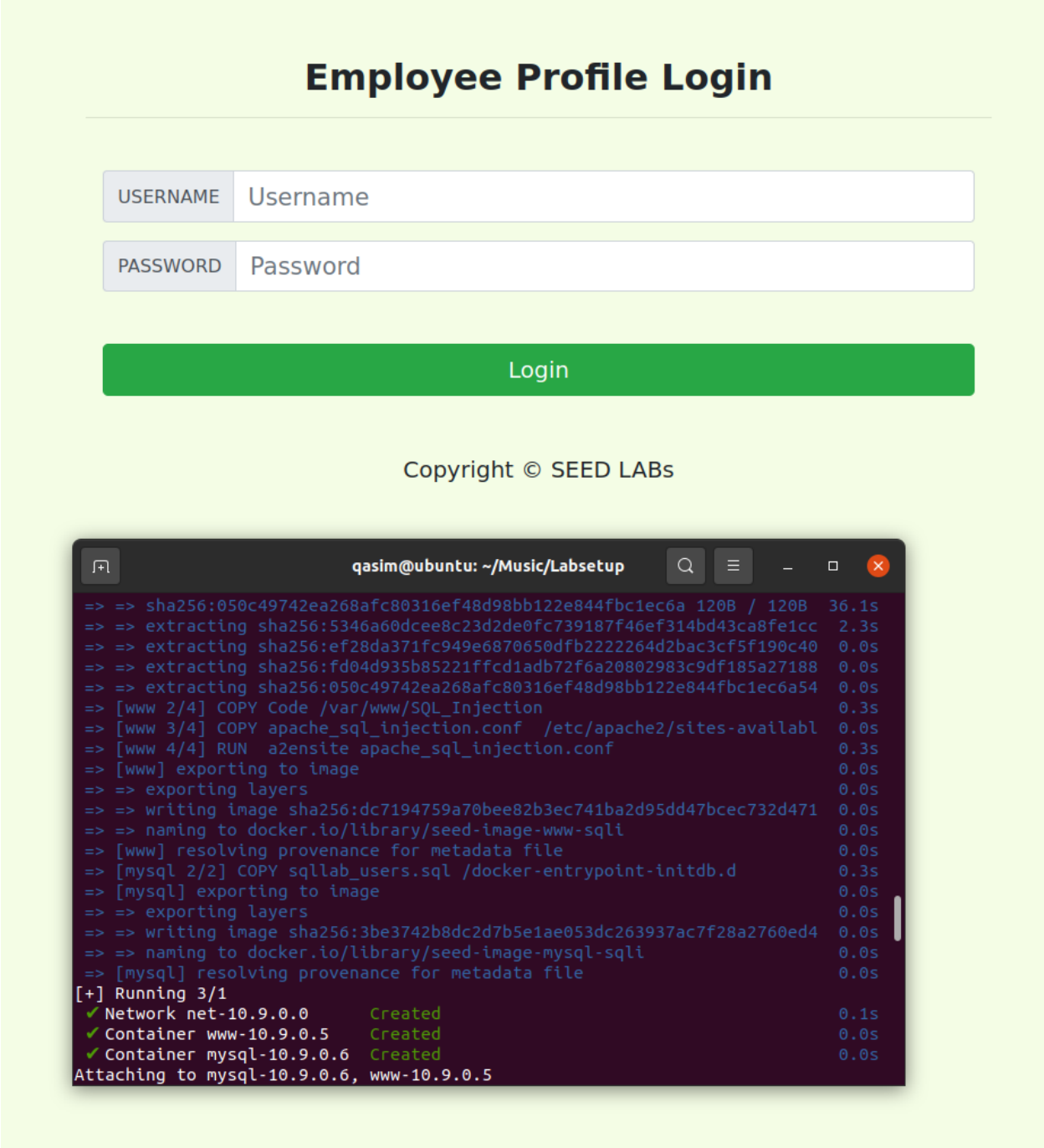
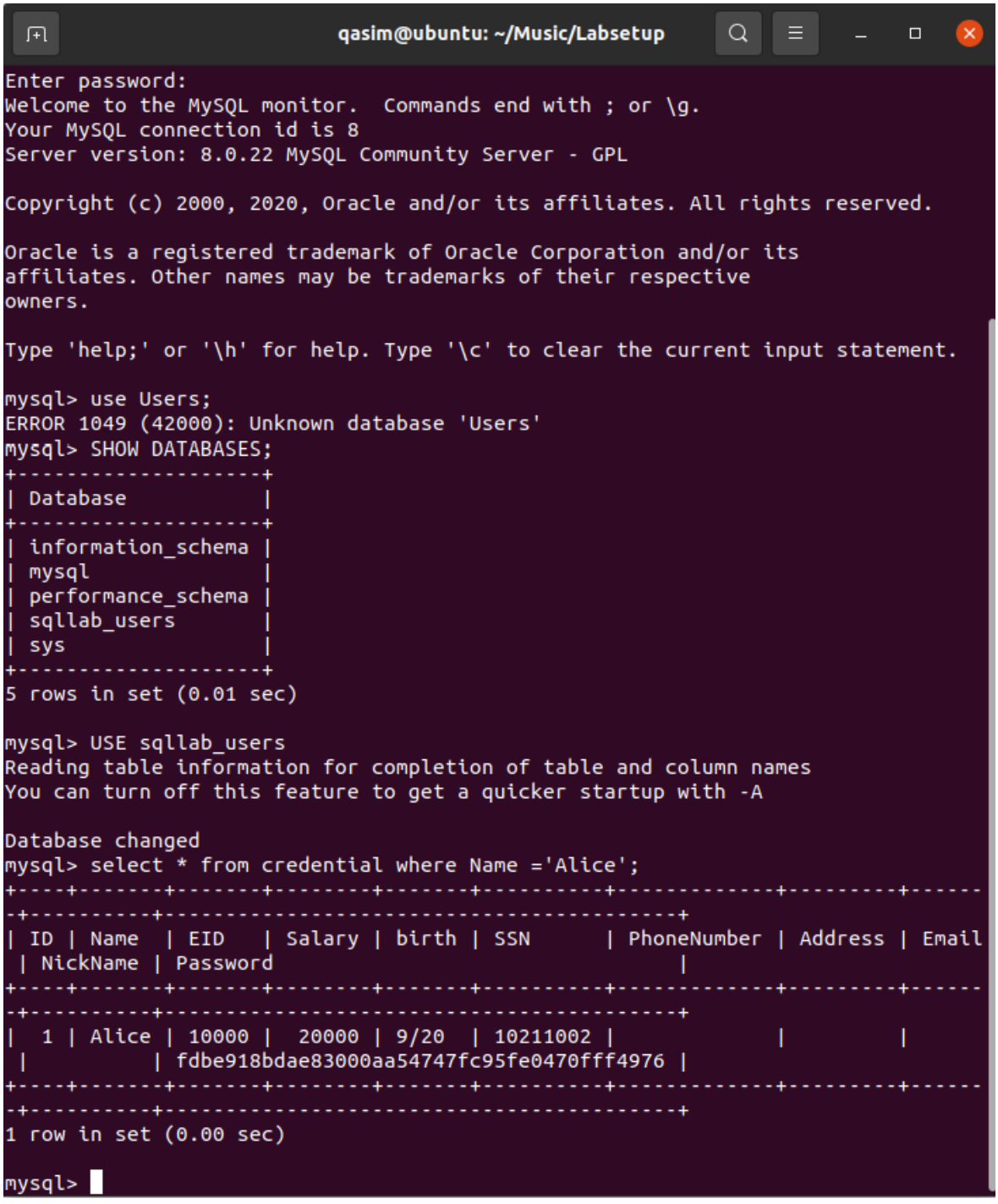
IS Lab 4

Setup docker containers and add seed-server ip to host file, go to the website



Task 1:

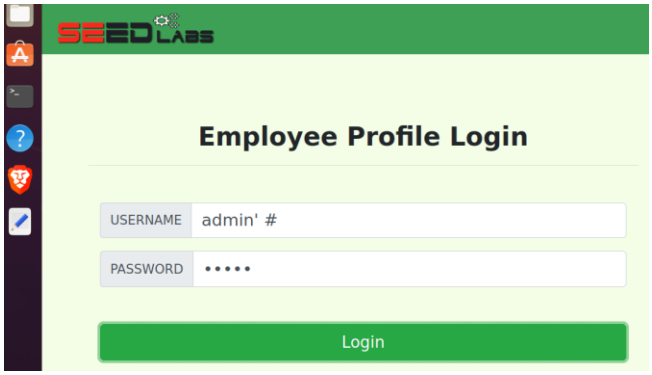
We first login into the MySQL console and switch the database(ps:dees) in use to Users:



Task 2:

Task 2.1:

Entering the username as Admin' # and password as admin as Everything after "admin" is commented out, including the password, thanks to the # symbol. As a result, we were able to use the admin ID to obtain all of the employee data.



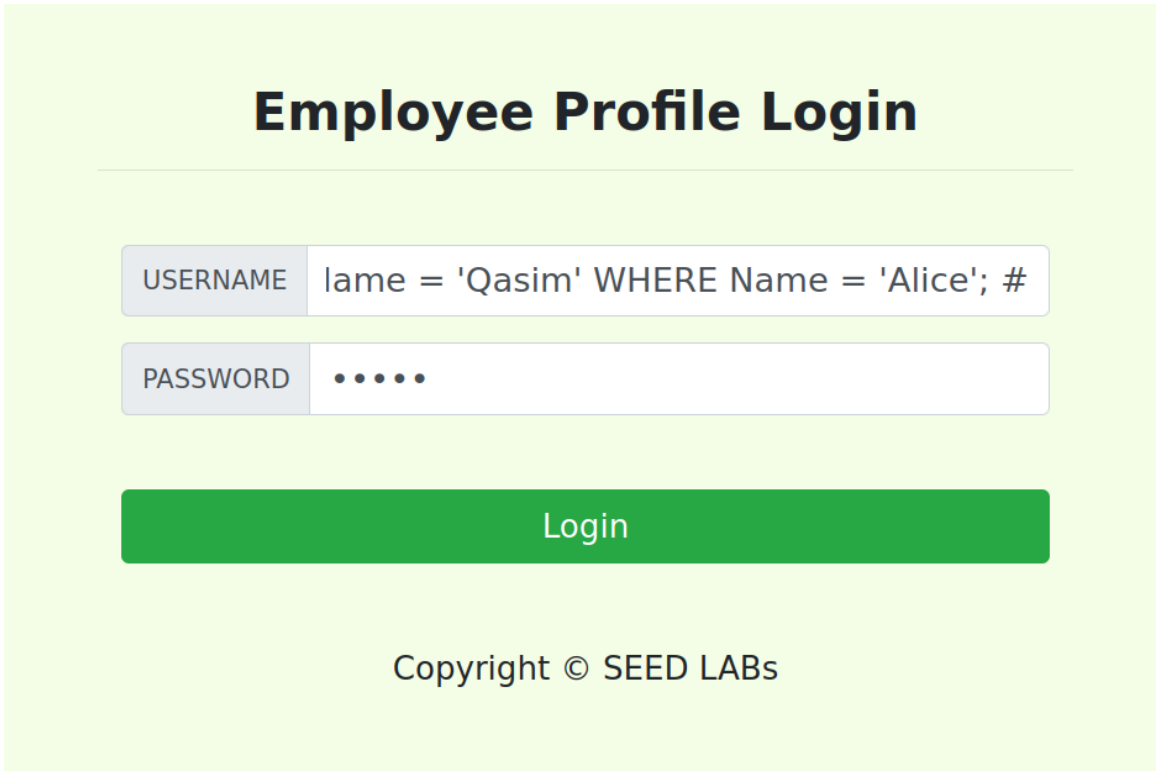

```
ysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+--------
--+
 ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickNam
   |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+--------
--+
  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |
6 |
  2 | Boby  | 20000 |  30000 | 4/20  | 10213352 |             |         |       |
4 |
  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |
f |
  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |
f |
  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |
8 |
  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |
0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+--------
--+
 rows in set (0.00 sec)

ysql>
```

Admin	99999	400000	3/5	43254314

Task 2.2:

We may observe that every employee's information is provided in a tabular HTML format. As a result, we could launch the identical attack as in Task 2.1. Whereas Web UI cannot automate the assault, CLI commands can. Encoding the special characters in the HTTP request using the curl command was one significant departure from the web user interface. We make use of the following: Space; Single, Quote and Hash symbols for SQL injections.

Task 2.3:



SQL Injection with ;: The semicolon (;) separates SQL statements, allowing attackers to append their own queries. In your case, attempting to update Name = 'Alice' to Name = ‘Qasim’ causes an error, indicating that multiple commands are being blocked.

Injection to Delete Data: The attacker inputs admin'; DELETE FROM credential WHERE Name = 'Alice'; # in the login form. This injects a command to delete records where the Name is 'Alice', exploiting SQL injection.

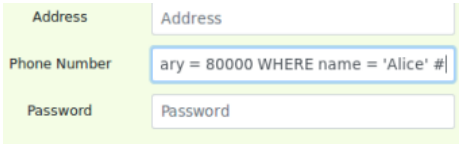
MySQLi Limitation: The mysqli::query() method in PHP blocks multiple queries for security reasons, preventing the injection from succeeding.

Bypassing with multi_query(): While mysqli::query() blocks multiple statements, the mysqli->multi_query() method can execute them. This is a security risk that allows SQL injection to perform actions like deleting data.

Task 3:

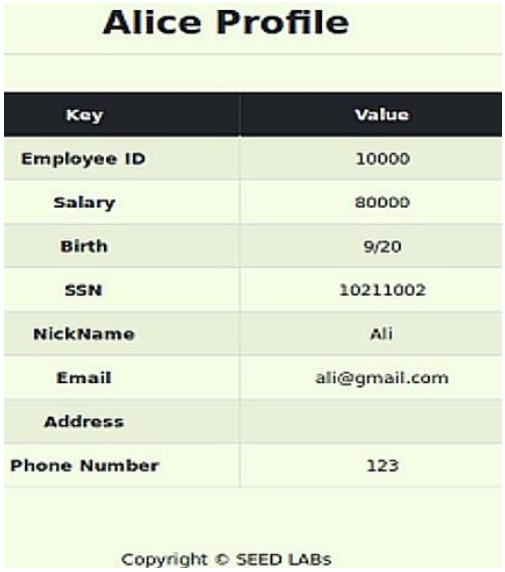
Task 3.1:

SQL Injection to Modify Data: The attacker enters 123’, salary = 80000 WHERE name = ‘Alice’ # in the profile form. This injects an additional SQL statement to modify Alice's salary.




```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------
---+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password
   |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------
---+
|  1 | Alice | 10000 |  80000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff49
76 |
|  2 | Boby  | 20000 |  20000 | 4/20  | 10213352 |             |         |       |          | b78ed97677c161c1c82c142906674ad15242b2
d4 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | a3c50276cb120637cca669eb38fb9928b017e9
ef |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | 995b8b8c183f349b3cab0ae7fccd39133508d2
af |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 99343bff28a7bb51cb6f22cb20a618701a2c2f
58 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | a5bdf35a1df4ea895905f6f6618e83951a6eff
c0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------
---+
6 rows in set (0.00 sec)

mysql>
```

Injected Query: The resulting query becomes:



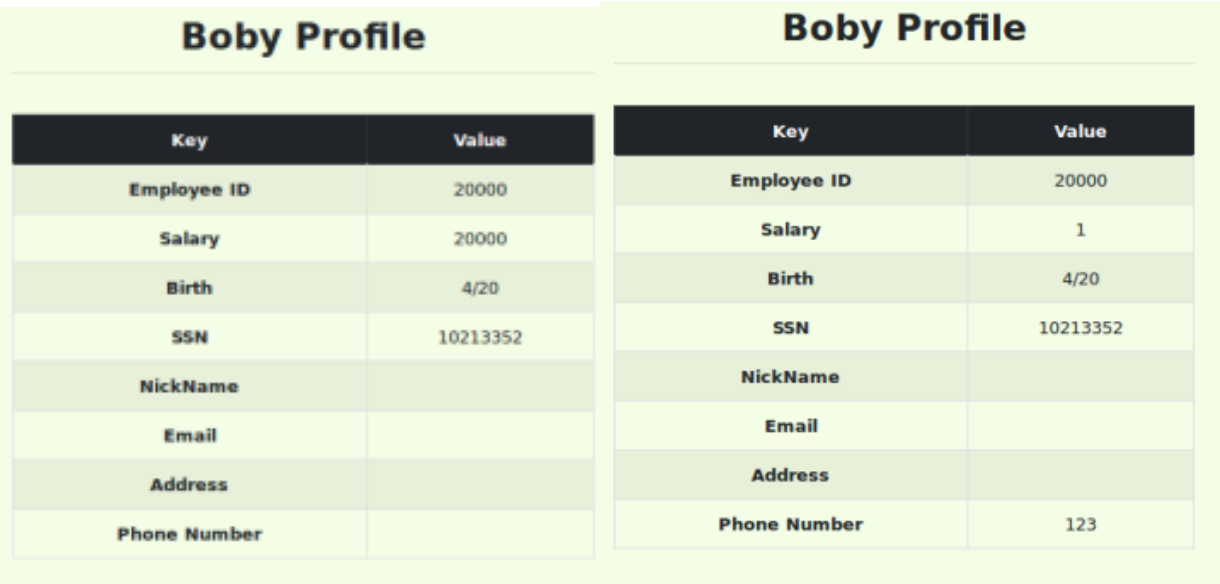
Alice Profile

Key	Value
Employee ID	10000
Salary	80000
Birth	9/20
SSN	10211002
NickName	Ali
Email	ali@gmail.com
Address	
Phone Number	123

Copyright © SEED LABs

Task 3.2:

We see that Boby's profile before any changes. Now, we try to change Boby's salary from Alice's account using the following string: 123', salary = 1 WHERE name = 'Boby' # On saving the changes, we log in into Boby's profile and check his details now and see that we have successfully changed his salary. We could enter that string in any of the other fields as well except password, because it is hashed.



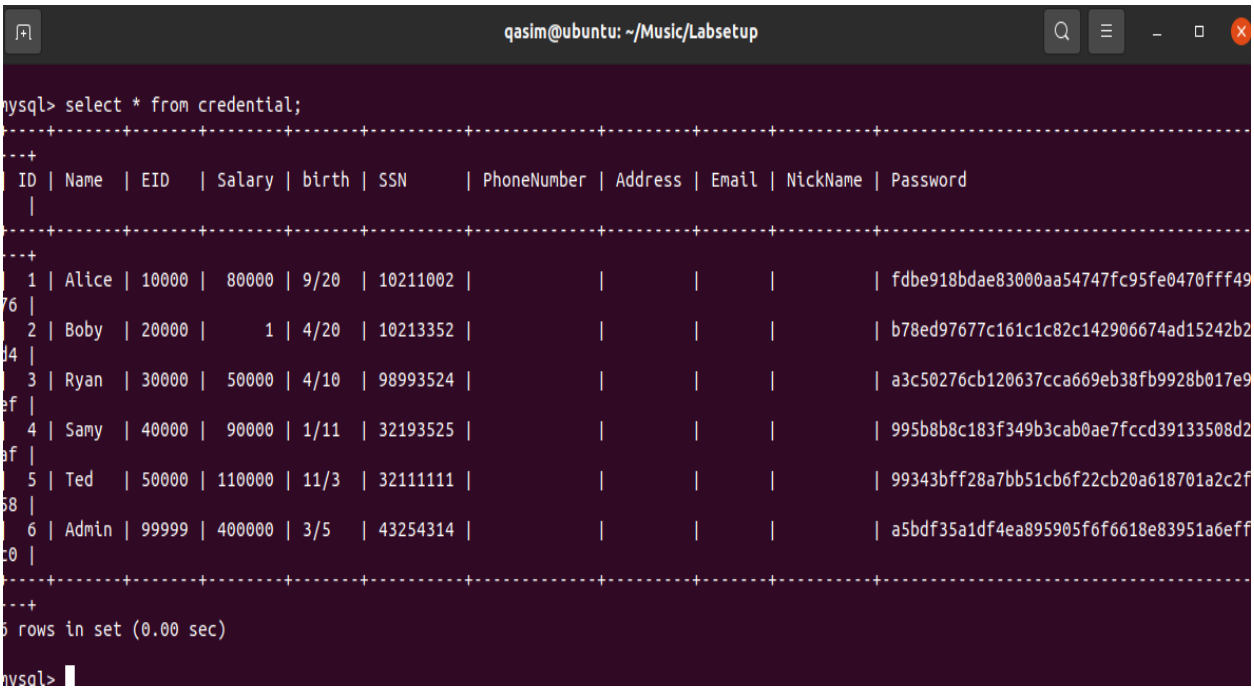
Boby Profile

Key	Value
Employee ID	20000
Salary	20000
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	

Boby Profile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	123

See changes in MySQL terminal:



Task 3.3:

Using SQL injection, we modified Boby's password by entering ', Password = sha1('Hacked') WHERE Name = 'Boby' # in Alice's profile. This updated Boby's password to "Hacked" using the sha1 function. The attacker could then log in to Boby's account with the new password.

Task 4:

```
// Function to create a sql connection.
function getDB() {
  $dbhost="10.9.0.6";
  $dbuser="seed";
  $dbpass="dees";
  $dbname="sqllab_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
$sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
FROM credential
WHERE name= '$input_uname' and Password='$hashed_pwd'";
if (!$result = $conn->query($sql)) {
  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);
```

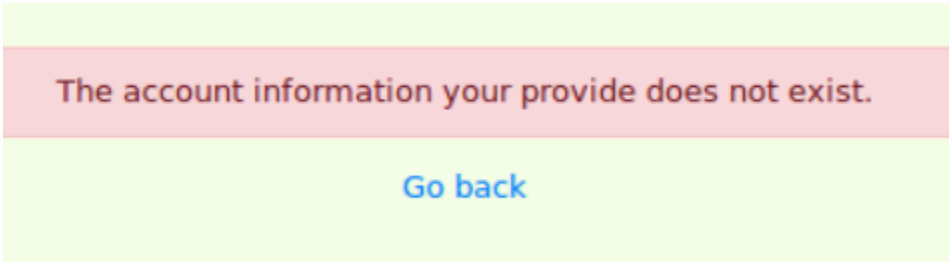
```
qasim@ubuntu: ~/Music/...
 ✔ Container www-10.9.0.5    Created
                             0.0s
Attaching to mysql-10.9.0.6, www-10.9.0.5
www-10.9.0.5    |  * Starting Apache httpd web serve
r apache2                   AH00558: apache2: Could n
ot reliably determine the server's fully qualified d
omain name, using 10.9.0.5. Set the 'ServerName' dir
ective globally to suppress this message
www-10.9.0.5    |  *
```

We now prepare statements of the previously attacked SQL statements in order to address this vulnerability. The unsafe_home.php file's SQL statement from job 2 is rewritten as follows:

```
// create a connection
$conn = getDB();
// Sql query to authenticate the user
$sql = $conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
FROM credential
WHERE name= ? and Password= ?");
$sql->bind_param("ss", $input_uname, $hashed_pwd);
$sql->execute();
$sql->bind_result($id, $name, $eid, $salary, $birth, $ssn, $phoneNumber, $address, $email, $nickname, $pwd);
$sql->fetch();
$sql->close();
```

We can no longer access the admin account, and we are no longer successful. The error message states that no user with the login credentials admin' # and admin password was found.

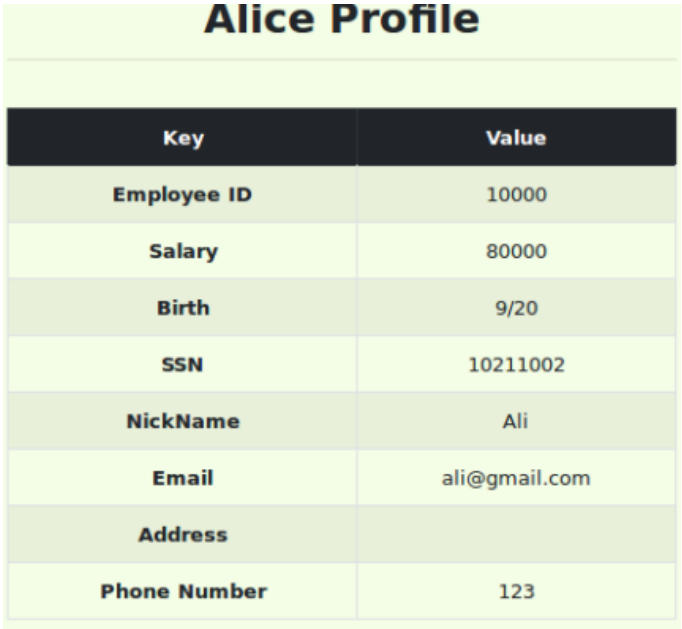
Attacking again:



Now to fail the attack on task 3, before that make these changes recommended for php security

```php
$conn = getDB();
// Don't do this, this is not safe against SQL injection attack
$sql="";
if($input_pwd!=''){
  // In case password field is not empty.
  $hashed_pwd = sha1($input_pwd);
  //Update the password stored in the session.
  $_SESSION['pwd']=$hashed_pwd;
  $sql = $conn->prepare("UPDATE credential SET nickname= ?,email= ?,address= ?,Password= ?,PhoneNumber= ?
here ID=$id;");
  $sql->bind_param("sssss",$input_nickname,$input_email,$input_address,$hashed_pwd,$input_phonenumber);
  $sql->execute();
  $sql->close();
}else{
  // if passowrd field is empty.
  $sql = $conn->prepare("UPDATE credential SET nickname=?,email=?,address=?,PhoneNumber=? where ID=$id;");
  $sql->bind_param("ssss",$input_nickname,$input_email,$input_address,$input_phonenumber);
  $sql->execute();
  $sql->close();
}
```

When attempting the same action as in Task 3 and saving the changes, we observe that the salary remains unchanged, indicating that the SQL injection attempt was unsuccessful.



After completing the compilation stage, a prepared statement becomes a pre-compiled query with blank data placeholders. We must supply data to this pre-compiled query for it to run, but this data will bypass the compilation stage and be sent straight to the execution engine after being put into the pre-compiled query. Therefore, even if SQL code is present in the data, it will be considered as part of the data without any special meaning if the compilation phase is skipped. Prepared statements guard against SQL injection attacks in this way.