SQL Injection Attack Lab

Level : Medium

Elgg Based Labs: SQL Injection Attack Lab: Medium level

Link : http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Web/Web_SQL_Injection/

Goal: Launching the SQL injection attack on a vulnerable web application. Conducting experiments with several countermeasures.

SQL Injection: It is a code injection technique which exploits the vulnerabilities in the interface between web application and database servers.

When SQL queries are not properly constructed, SQL injection vulnerabilities can occur.

2.1 Lab Environment Configuration:

Starting Apache Server:

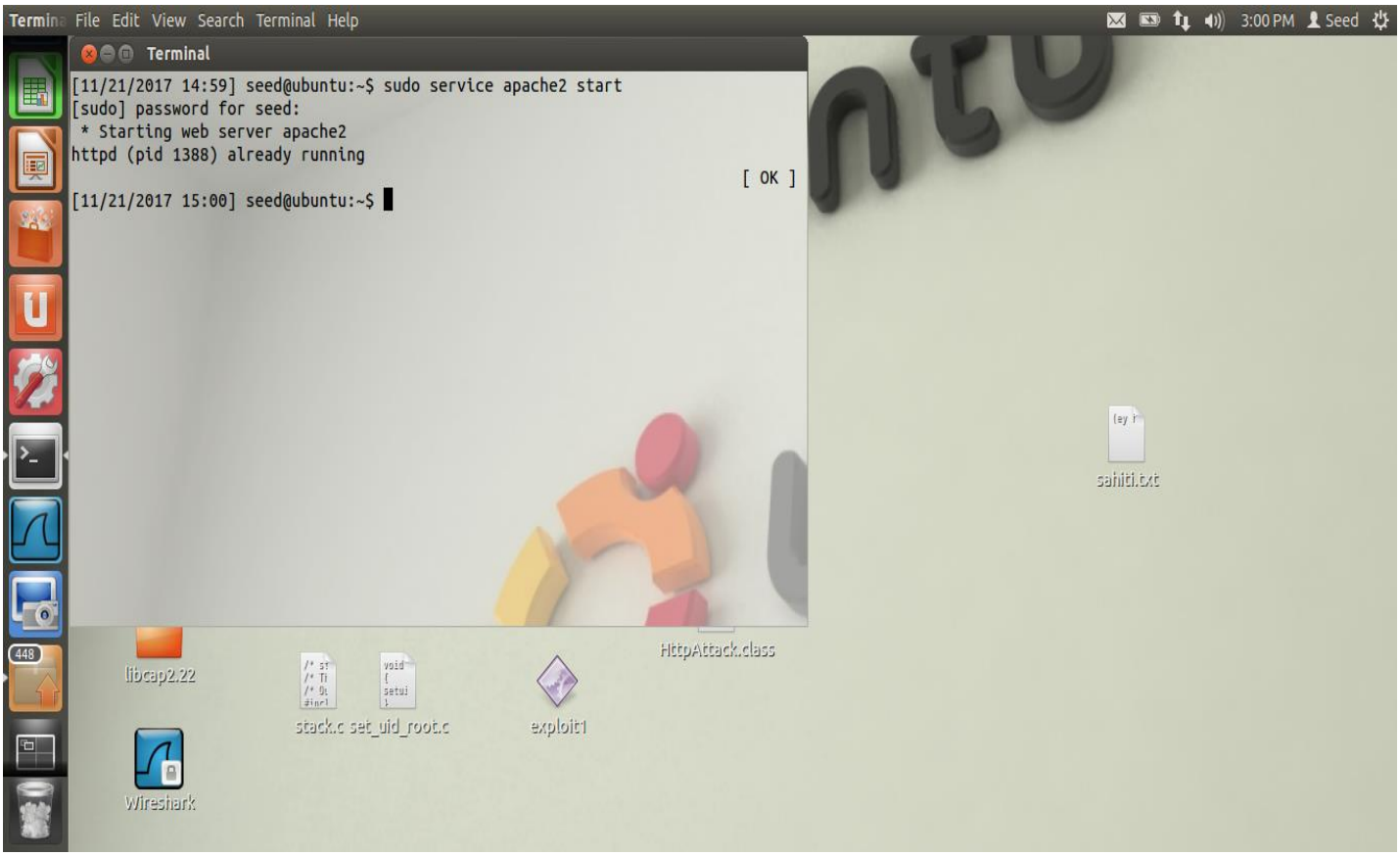


Fig: Starting the Apache Server

The apache server is started using the command : sudo service apache2 start

Configuring DNS:

URL: http://www.SEEDLabSQLInjection.com

Folder: /var/www/SQLInjection/

Mapped domain name to a particular IP address using /etc/hosts.

Appended 127.0.0.1 www.example.com entry to /etc/hosts

Configured Apache Server using Virtual host.

2.2 Turn off the countermeasure:

PHP uses magic quote mechanism to defend against SQL injection attacks. Turned off this protection in the following way as shown in the screenshot

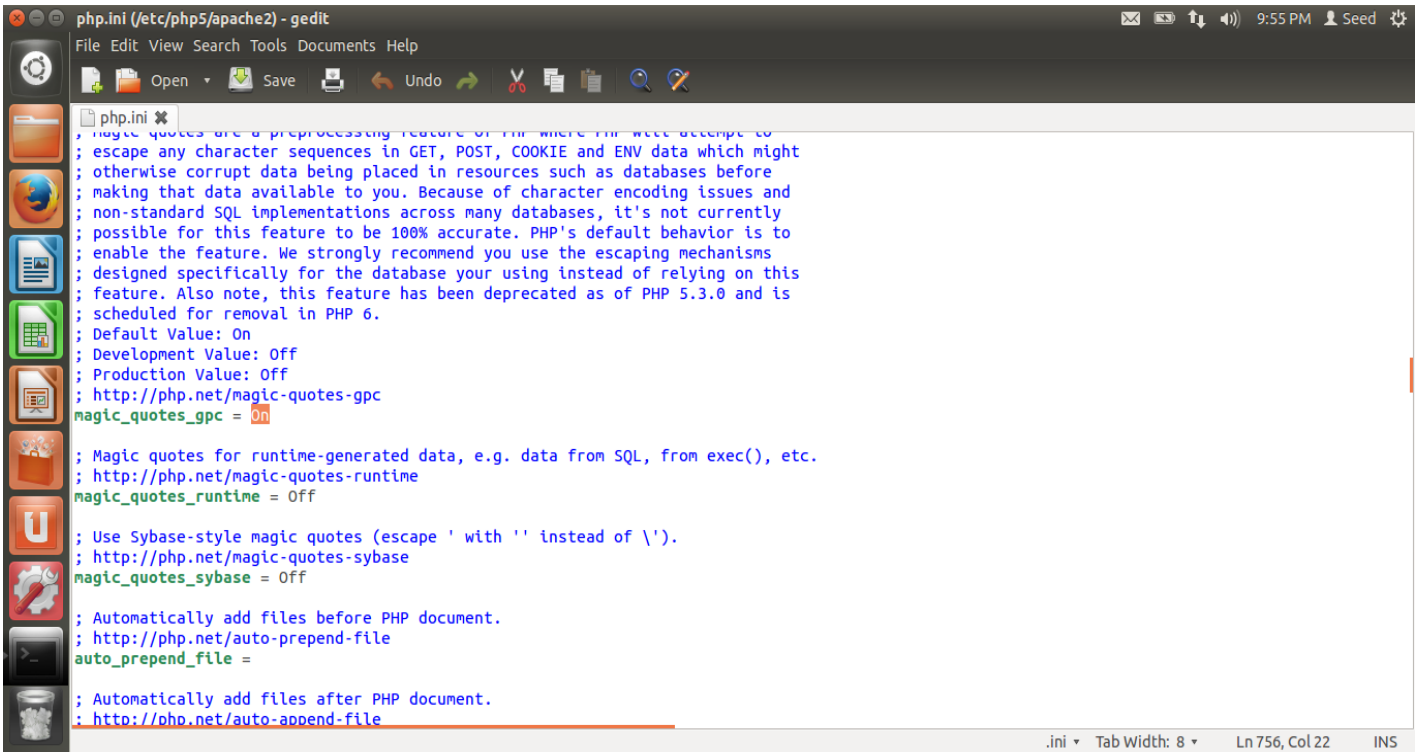


Fig: Magic quote protection is on.

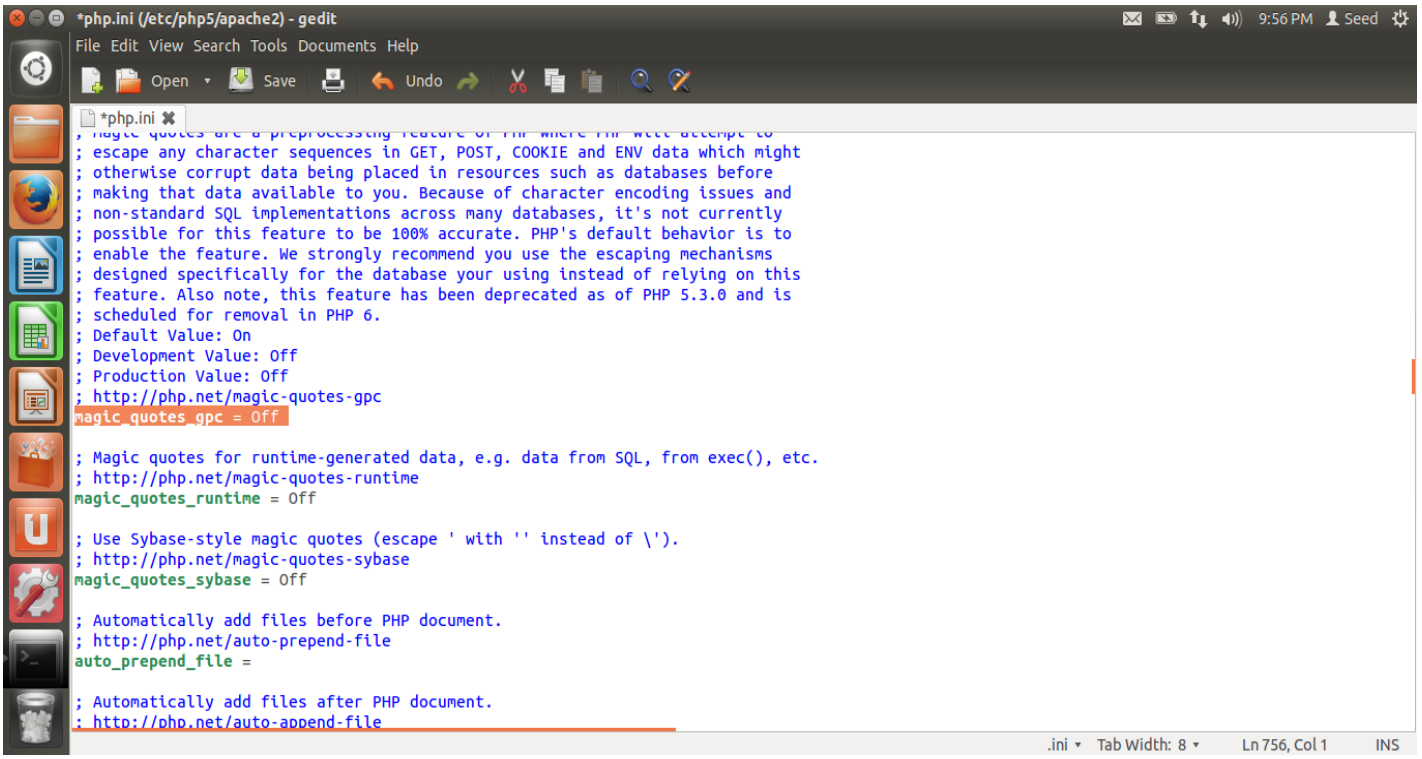


Fig: Turning off magic quote protection

2.3 Patch the Existing VM to add the web application:

Fig: Patching the existing VM to add the web application

Results: The environment is configured and the new web application is added to the VM through patching it. The protection magic quote is turned off in order to enable SQL injection attacks.

3. Lab Tasks

3.1 Task 1: MySQL Console :

Objective: Running SQL commands to print all profile information of the employee Alice

- **Web application:** www.SEEDLabSQLInjection.com
  - This is an 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.
- **Database:** Users
- **Table:** Credential
  - The table stores the personal information (e.g. eid, password, salary, ssn, etc.) of every employee.

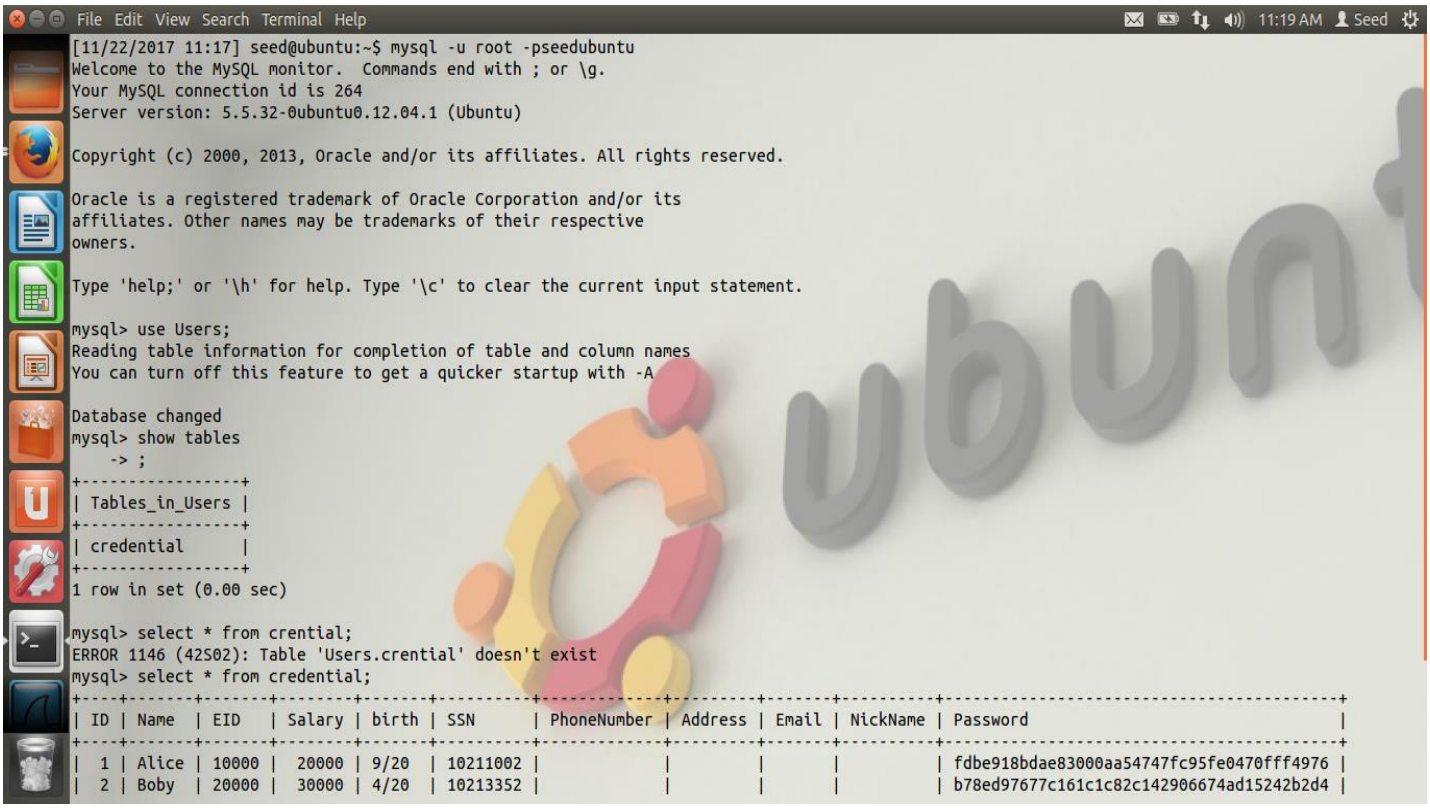


Fig: Database Users with table credential

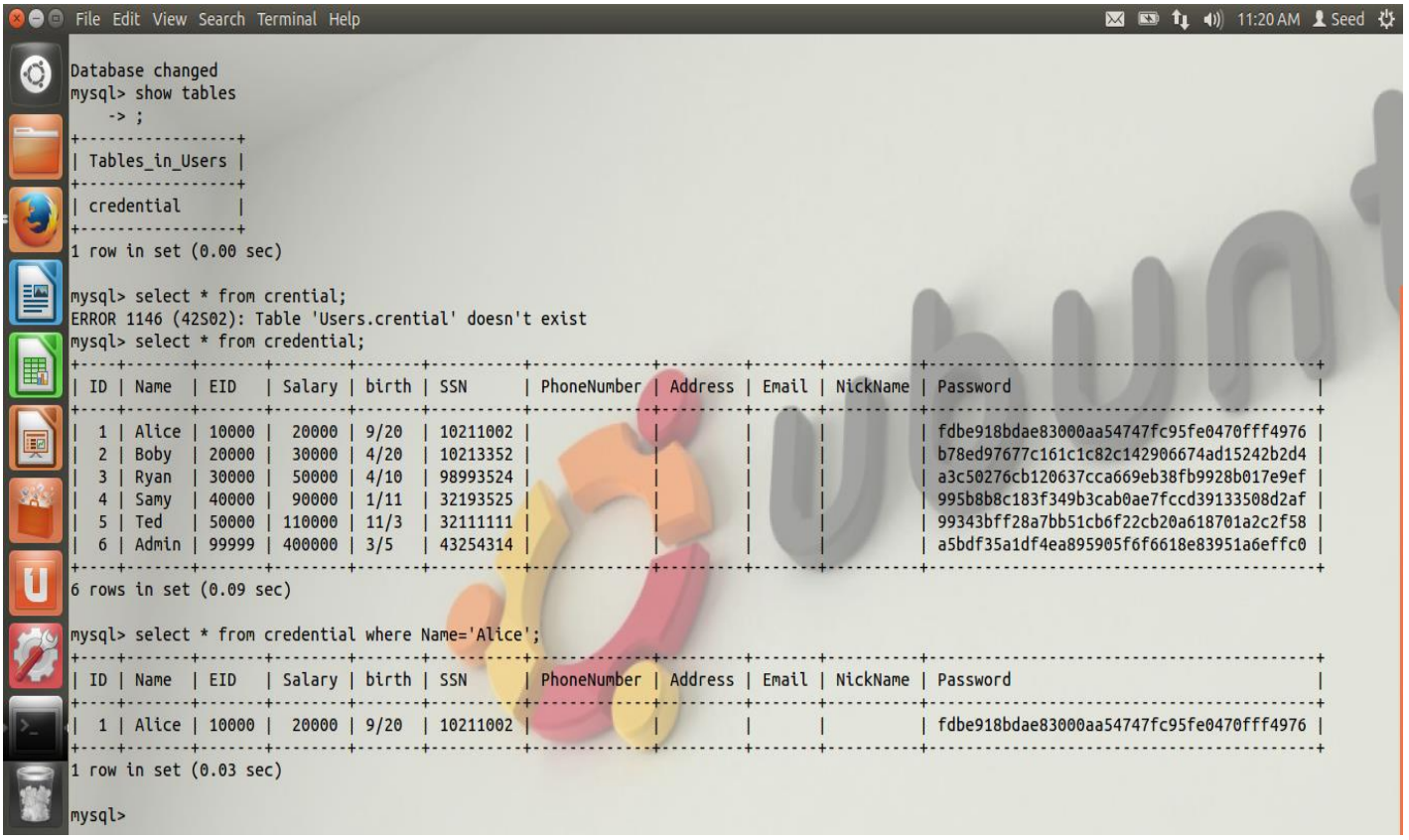


Fig: SQL command to print the profile information of the employee Alice.

3.2: Task 2: SQL injection attack on SELECT statement:

Objective: Log in to the application without knowing any employee's credential.

Procedure:

- Usually authentication is based on Employee ID and password.
- The attacker could modify the SQL statement to access the data. Attacker can execute their own code.
- The PHP code unsafe_credential.php, located in the /var/www/SQLInjection directory, is used to conduct user authentication.

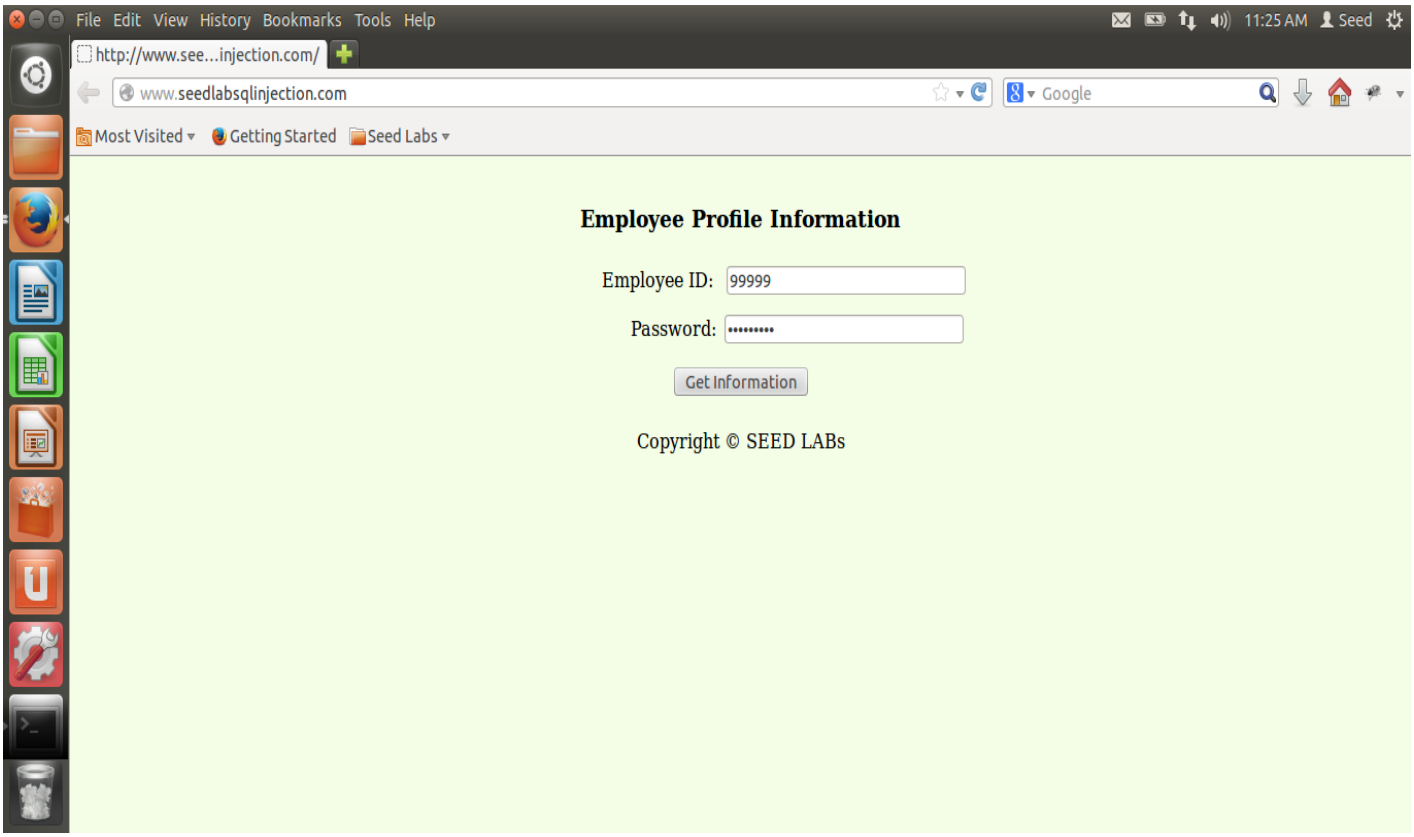


Fig: Web application which gives authentication based on the employee ID and password.

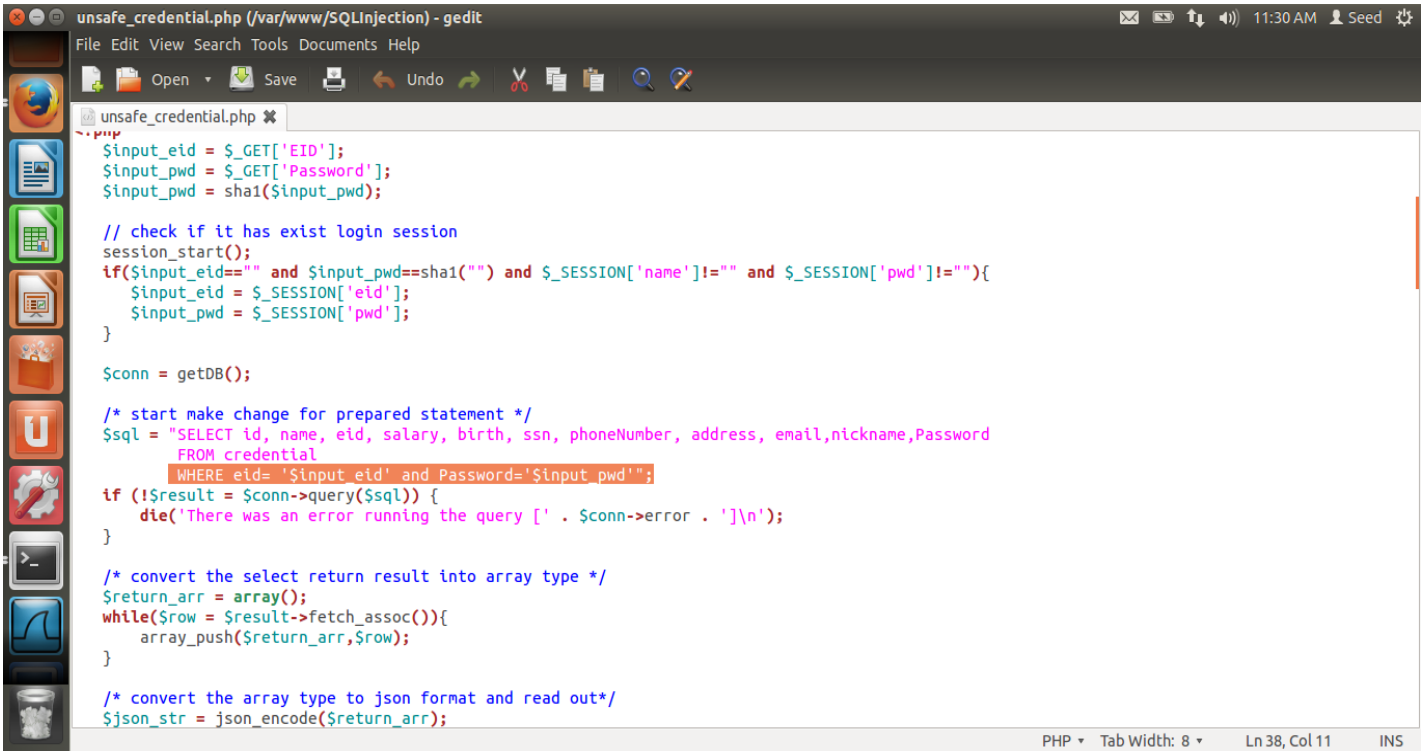


Fig: PHP code for user authentication

Task 2.1: SQL Injection Attack from webpage:

Objective: To get admin access to view all the employee's information.

Procedure:

- We do not know the ID and password for admin. We just know the name as admin.
- Using this details i.e with the name as admin, pass the malicious code in the employee ID region and comment out the password region as we do not know any other details apart from the name as admin. Pass ' or name='admin' ;-- in employee ID region to perform attack.

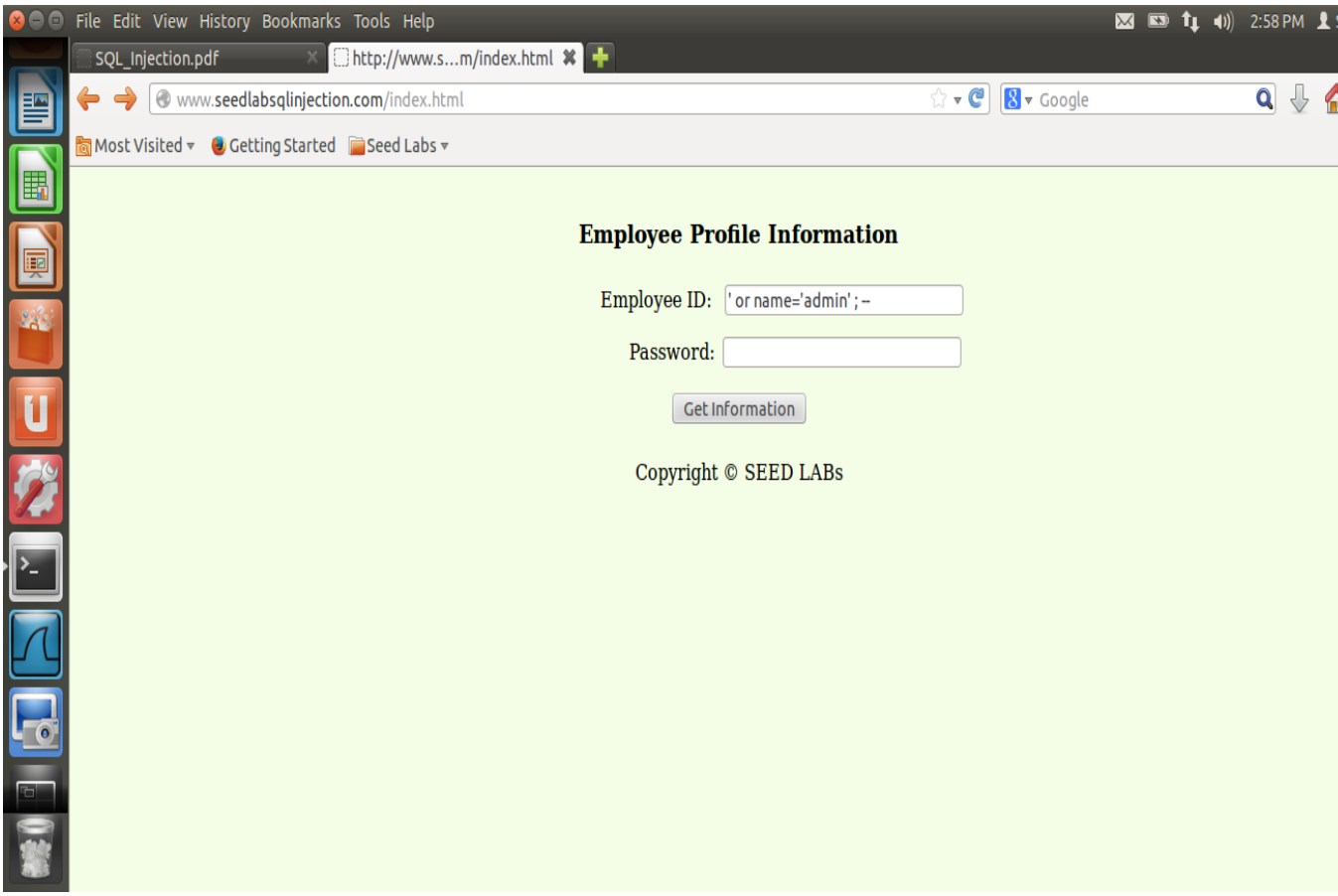


Fig: Attacking the web application with malicious code to get admin access

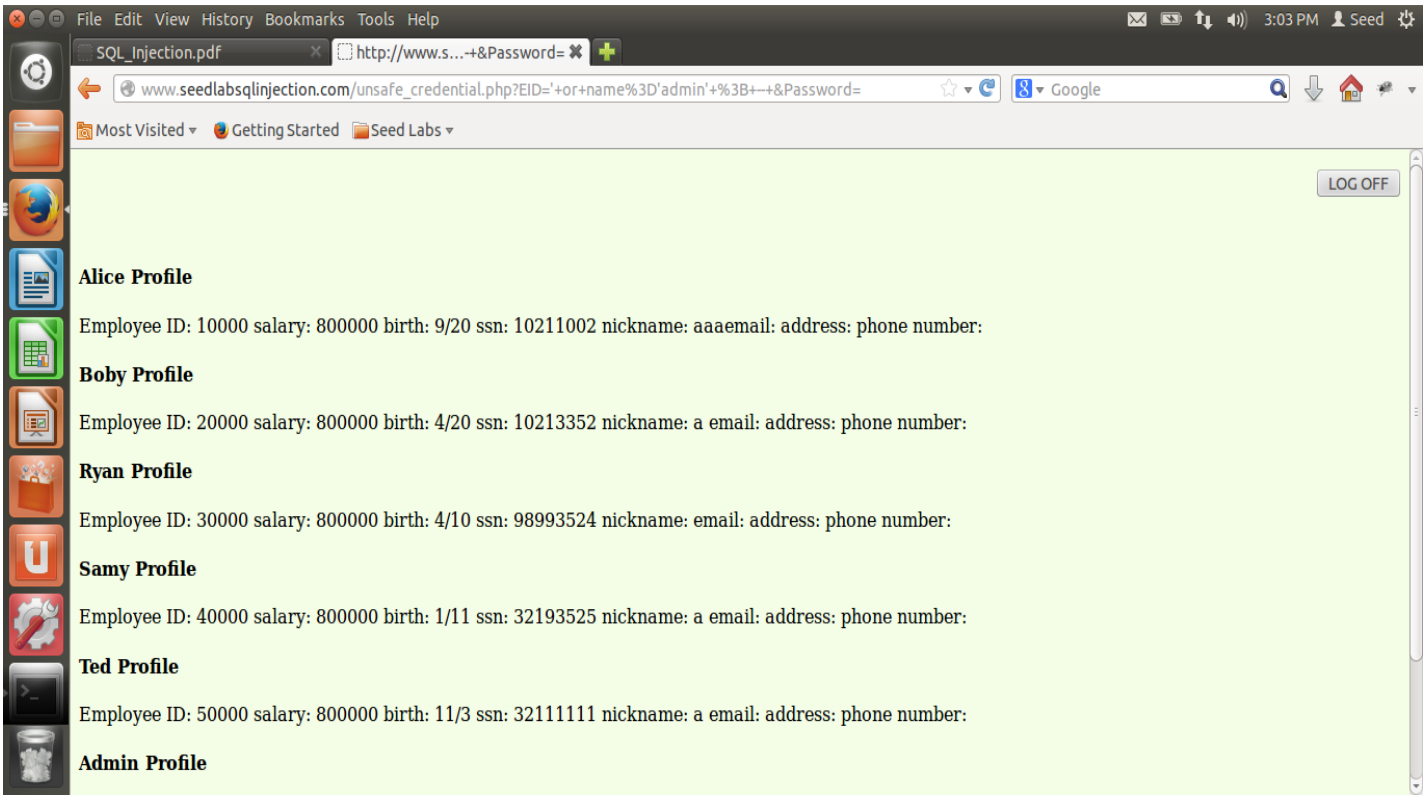


Fig: Admin access is achieved successfully with the malicious code injected by the attacker.

Results:

- The malicious code in the employee ID region is passed into the query region in the PHP code and in that name is taken as admin and the password part of the query is commented out.
- This turns out the query in a way to get admin access without knowing the Employee ID and password for admin.
- The attacker successfully gained admin privileges. Now the attacker has access to entire information and can do malicious activities with the sensitive data.

Task 2.2: SQL Injection Attack from command line.

Objective: Repeat the attack from task 1 using a command line tool like CURL which can send http requests instead of a web application.

Procedure:

- Install CURL.
- Pass the malicious URL in the CURL command line which sends HTTP requests.
- As you can see from the screenshot below, the EID and password regions are empty which means we do not know ID and password. We are aware of only one thing which is the name as admin. The malicious code passed as CURL command is used for fetching the data with admin access level.

Fig: Malicious URL sending as a parameter to the CURL command.

Fig: Admin level access is achieved and the data is displayed.

Results:

- SQLInjection attack is successful from the command line using CURL to get admin level access.

Results: We have seen how to steal information from the database using the vulnerability in login page from task 2.1 and task 2.2.

Task 2.3: Append a new SQL statement:

Objective: Modify the database using the same vulnerability in login page.

Procedure:

- Use SQL injection attack to turn one SQL statement into two, with second one being delete statement.
- Used semicolon to separate multiple SQL statements.
- Deleted a record from the database

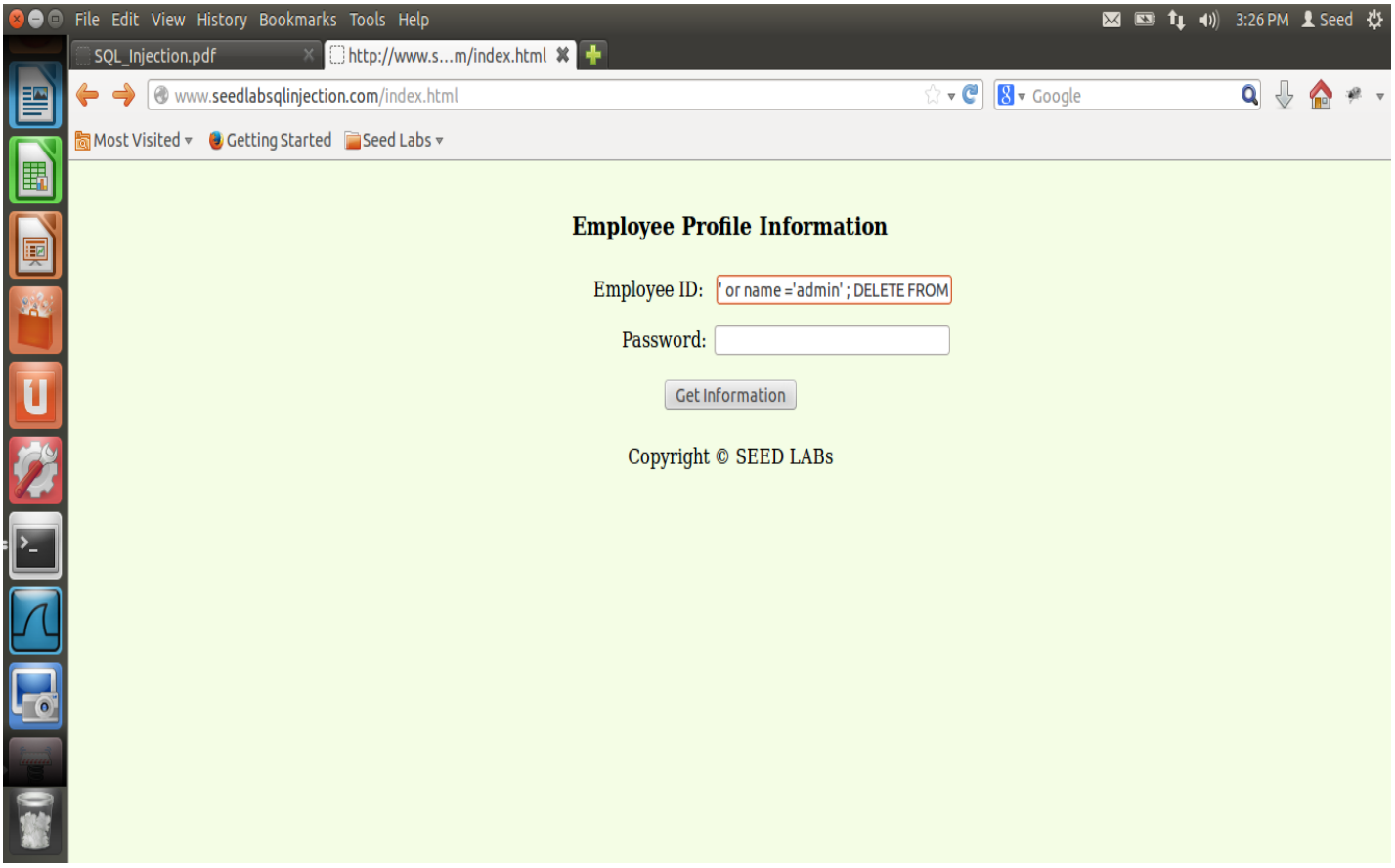


Fig: Append new SQL statement to attack to get admin access and delete a row in the database.

- Query to perform SQL injection to attack: ‘ or name=’admin’ ; DELETE FROM credential where name=’Ted’ –
- Changing the query as multi_query in unsafe_credential.php to execute multiple SQL statements.

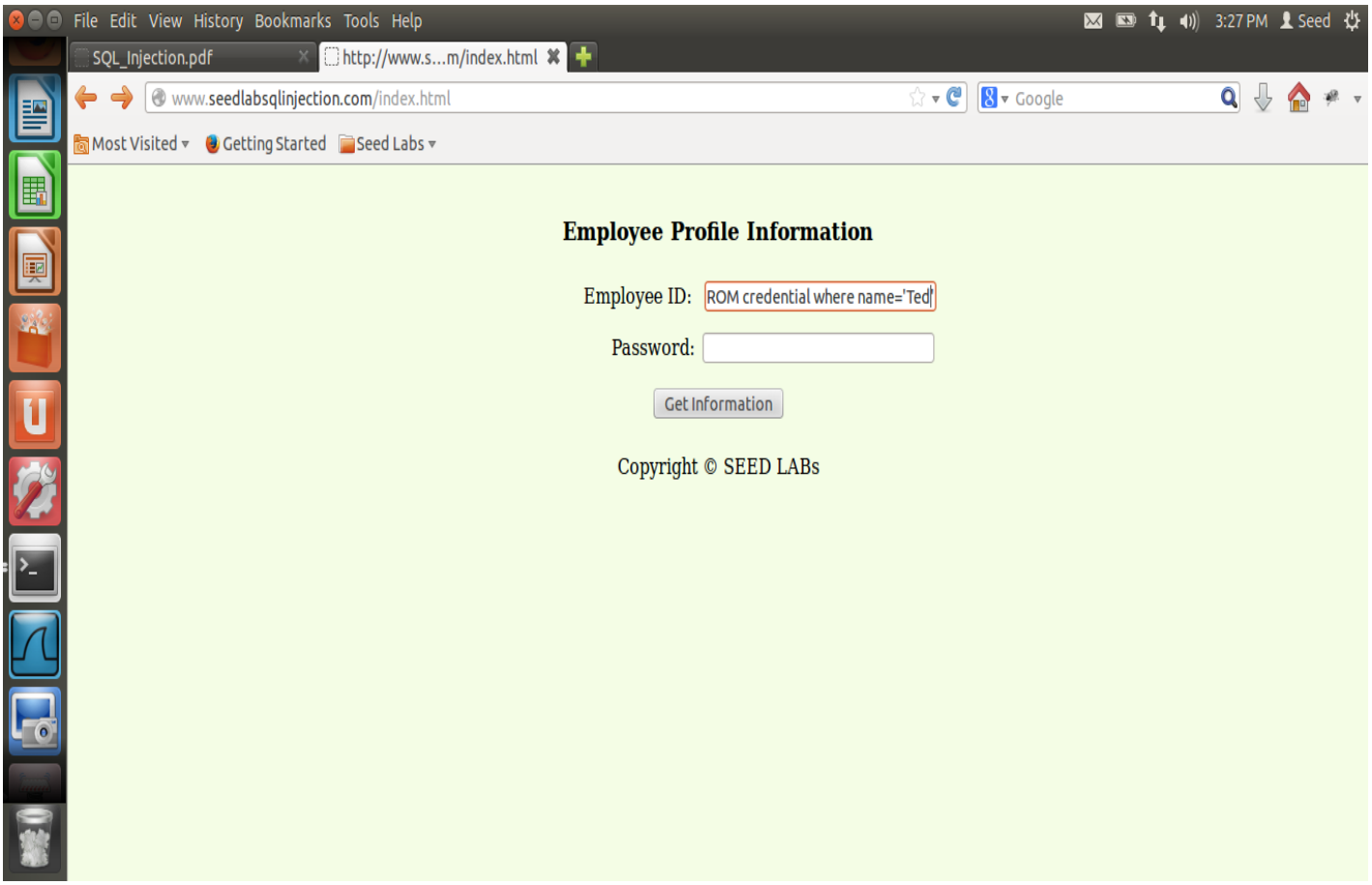


Fig: Deleting the record with name as TED.

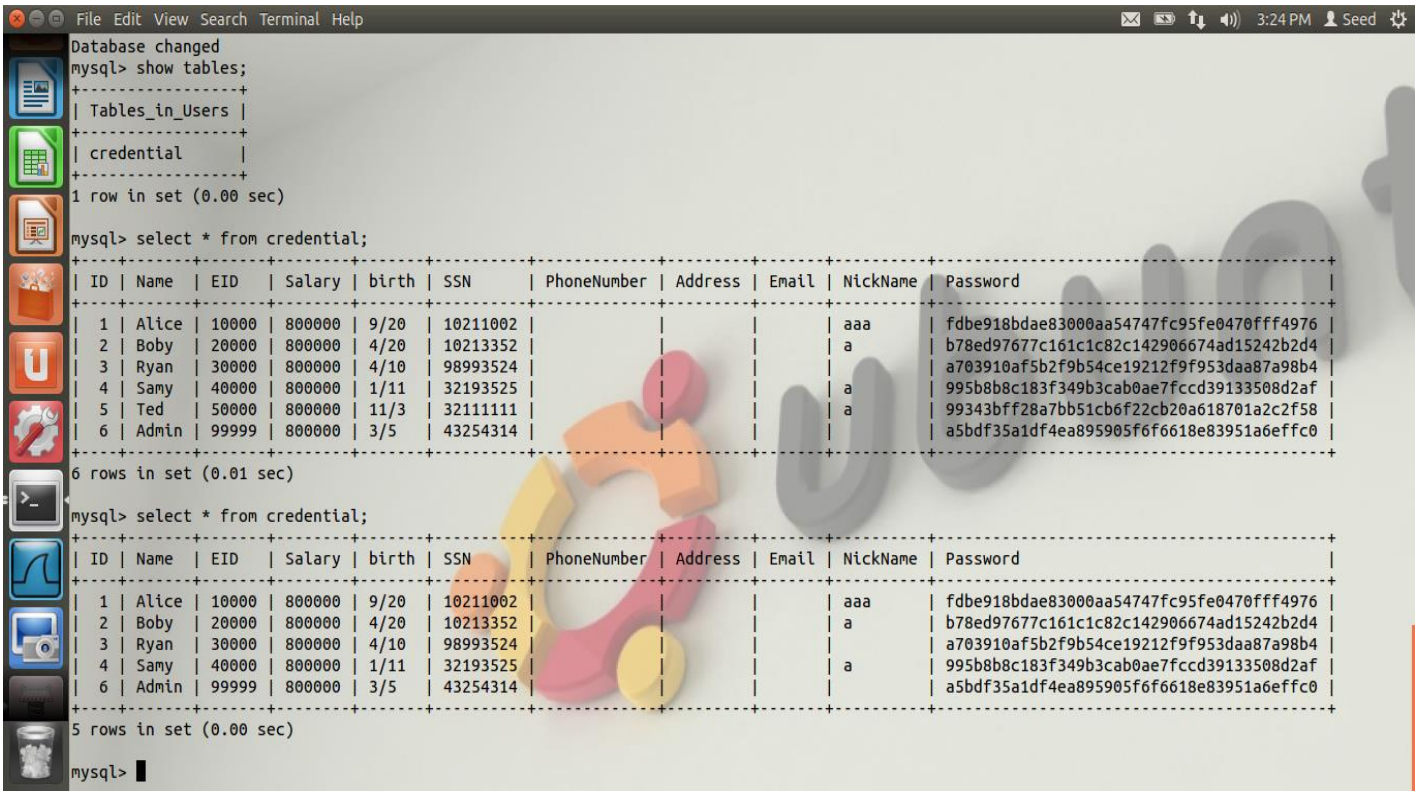


Fig: Database before and after deletion of the record "Ted" with malicious code SQL injection.

Results:

- Modified the database using SQL injection attack by exploiting the vulnerabilities in login page to delete a record from the table in the database.
- The record named as "TED" is deleted.

3.3 Task 3: SQL Injection Attack on UPDATE Statement

Objective: To update employee information using SQL injection attack using the vulnerability in Edit profile page of Employee Management application.

Procedure:

- In the Employee Management application, there is an Edit Profile page that allows employees to update their profile information, including nickname, email, address, phone number, and password.
- To go to this page, employees need to login first.
- When employees update their information through the Edit Profile page.
- The PHP code implemented in unsafe edit.php file is used to update employee's profile information.
- The PHP file is located in the /var/www/SQLInjection directory.

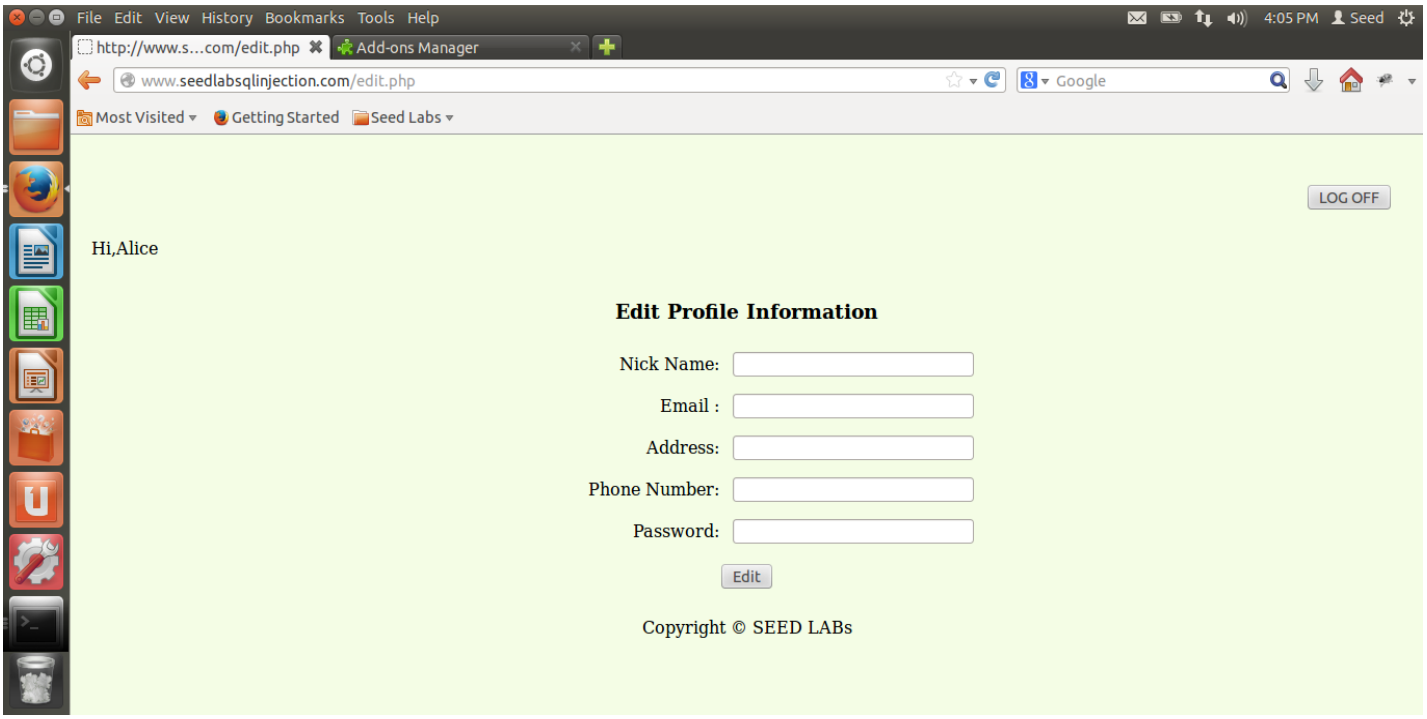


Fig: Edit Profile page for Alice

Task 3.1: SQL Injection Attack on UPDATE Statement — modify salary.

Objective: As a malicious employee (say Alice), the goal in this task is to increase Alice's salary via Edit Profile page.

Procedure:

- Only administrator is allowed to make changes to salaries.
- Alice as a malicious makes modifications to the SQL query which updates the salary of Alice to a huge amount thereby increasing her salary.

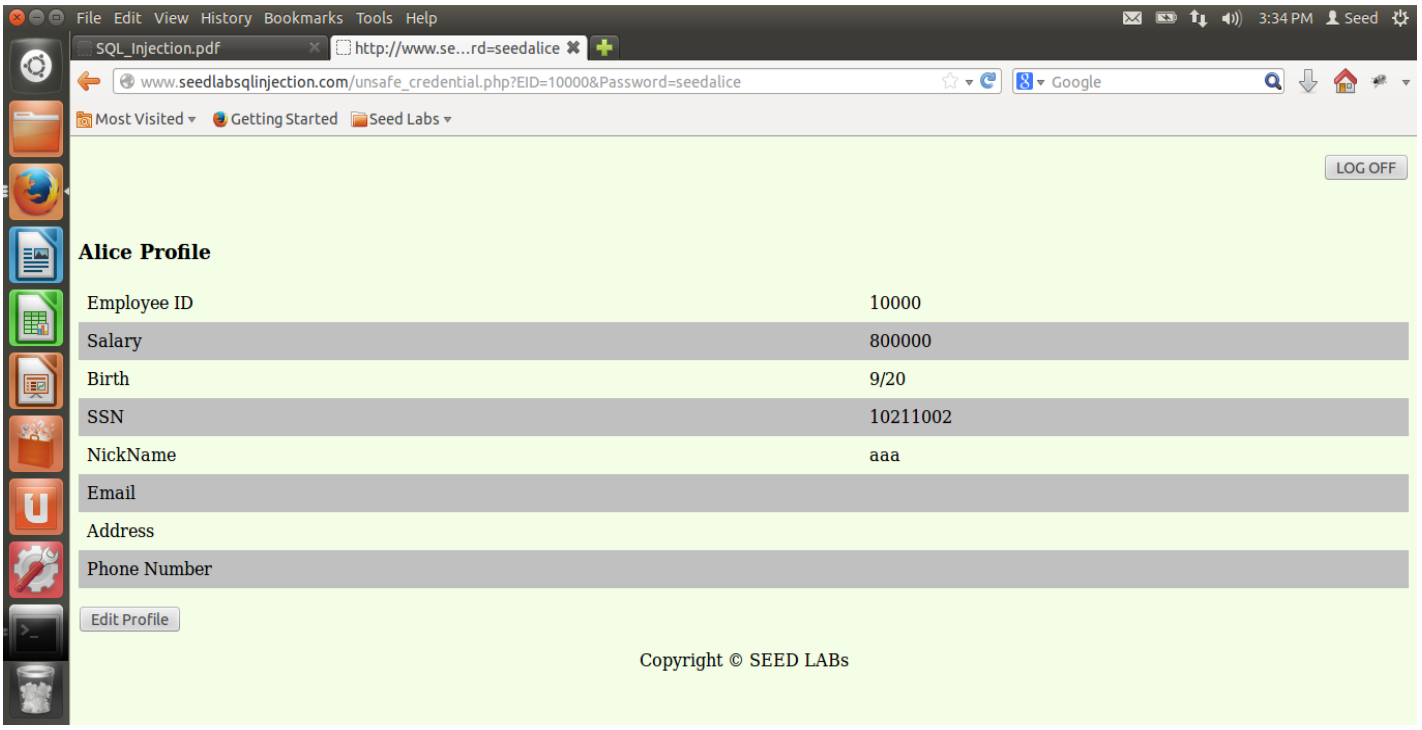


Fig: Alice Information before modifying salary.

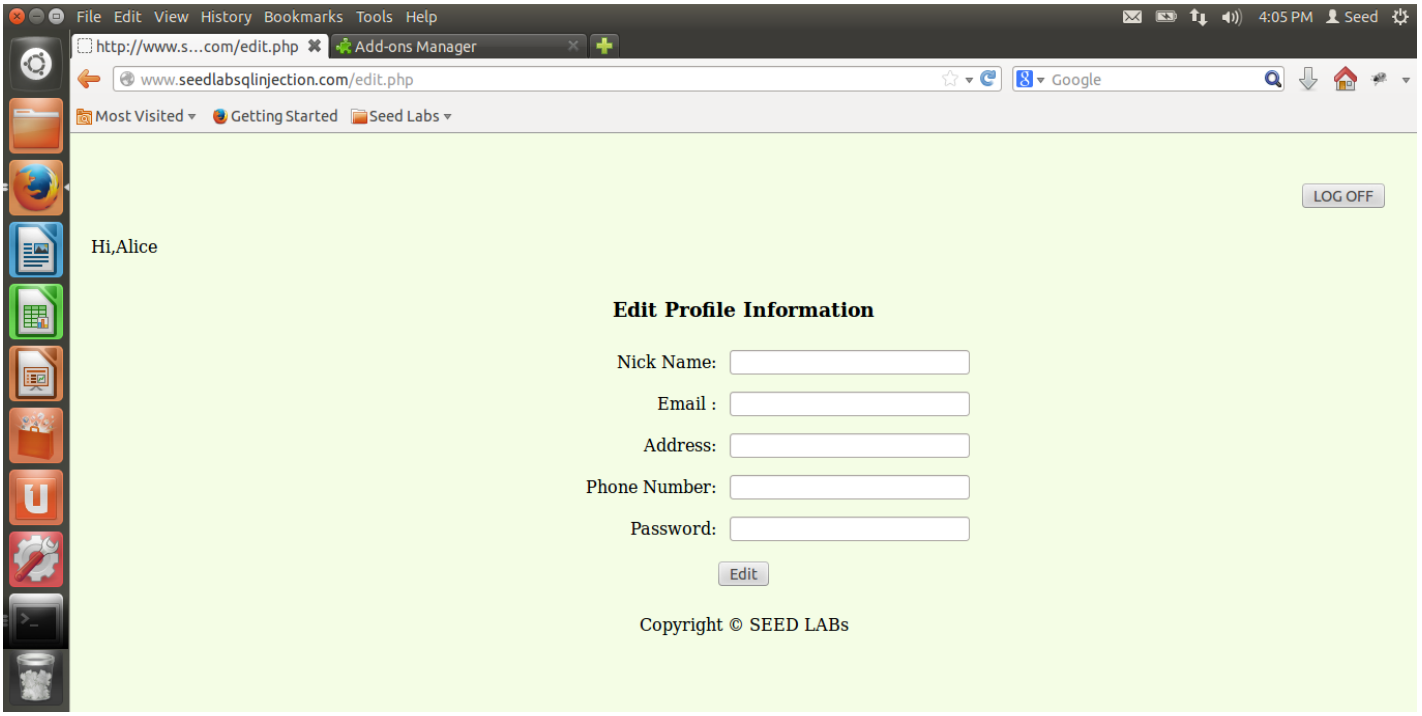


Fig: Alice's Edit Profile Page

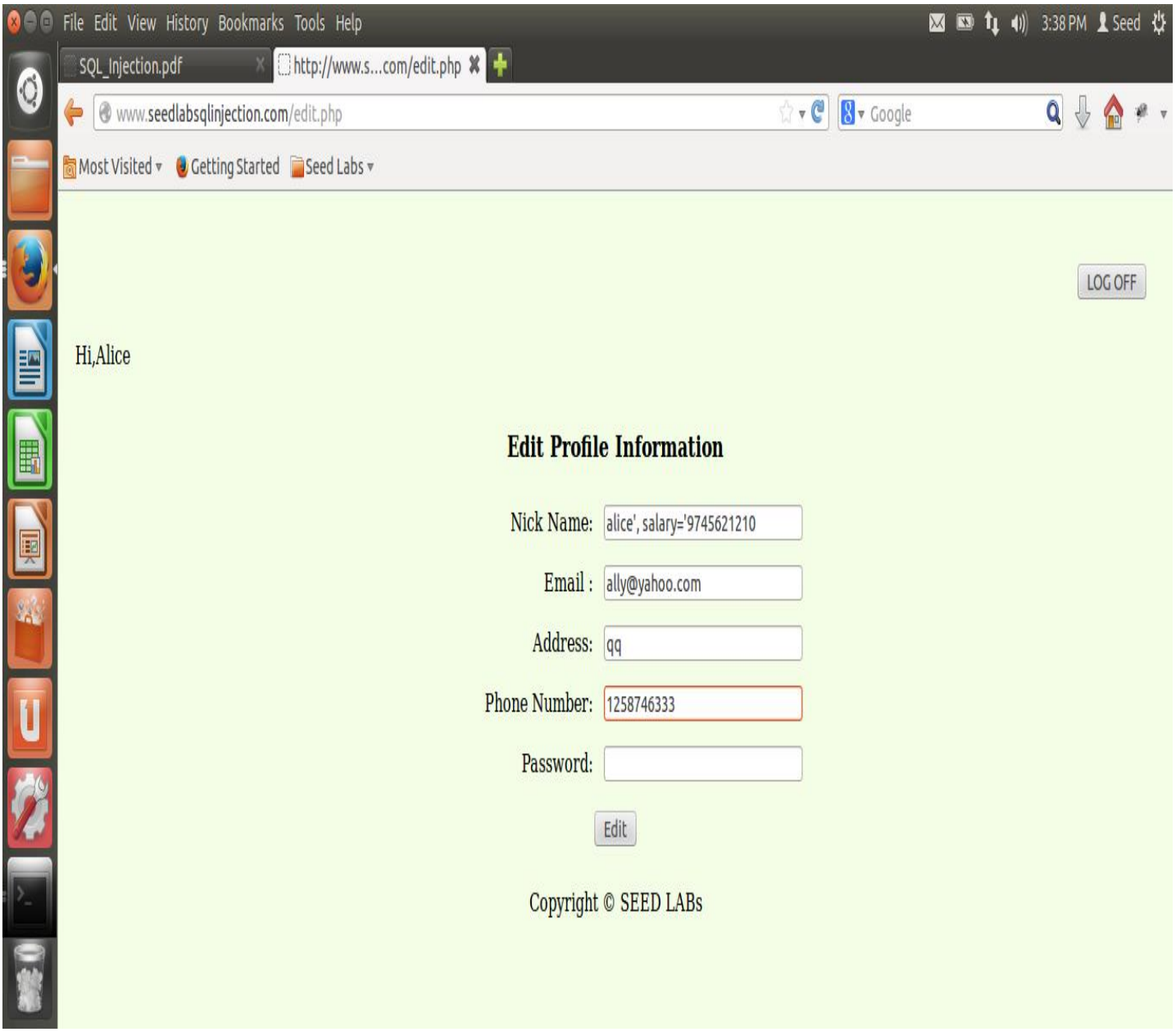


Fig: Edit profile information attack by introducing malicious code in nickname field to modify the salary of Alice.

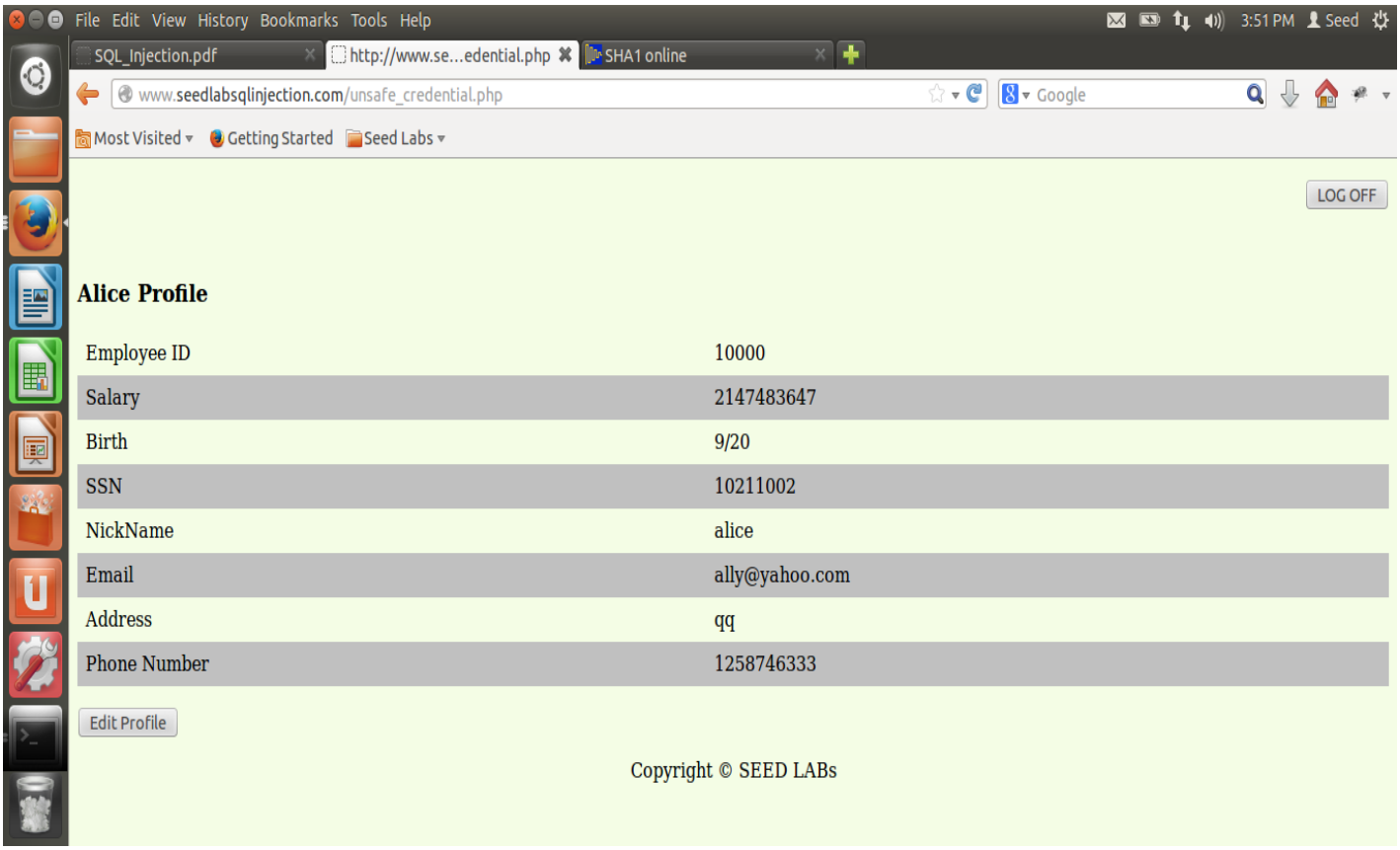


Fig: Alice's salary is increased because of SQL injection attack on update statement to modify salary.

Results:

- 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.
- As you have seen above, the salary of Alice is modified.

Task 3.2: SQL Injection Attack on UPDATE Statement — modify other people' password.

- Alice(attacker) wants to modify the password of Ryan(victim)
- Ryan's original password – seedryan
- Ryan's modified password – password
- Hash value for password: 5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8

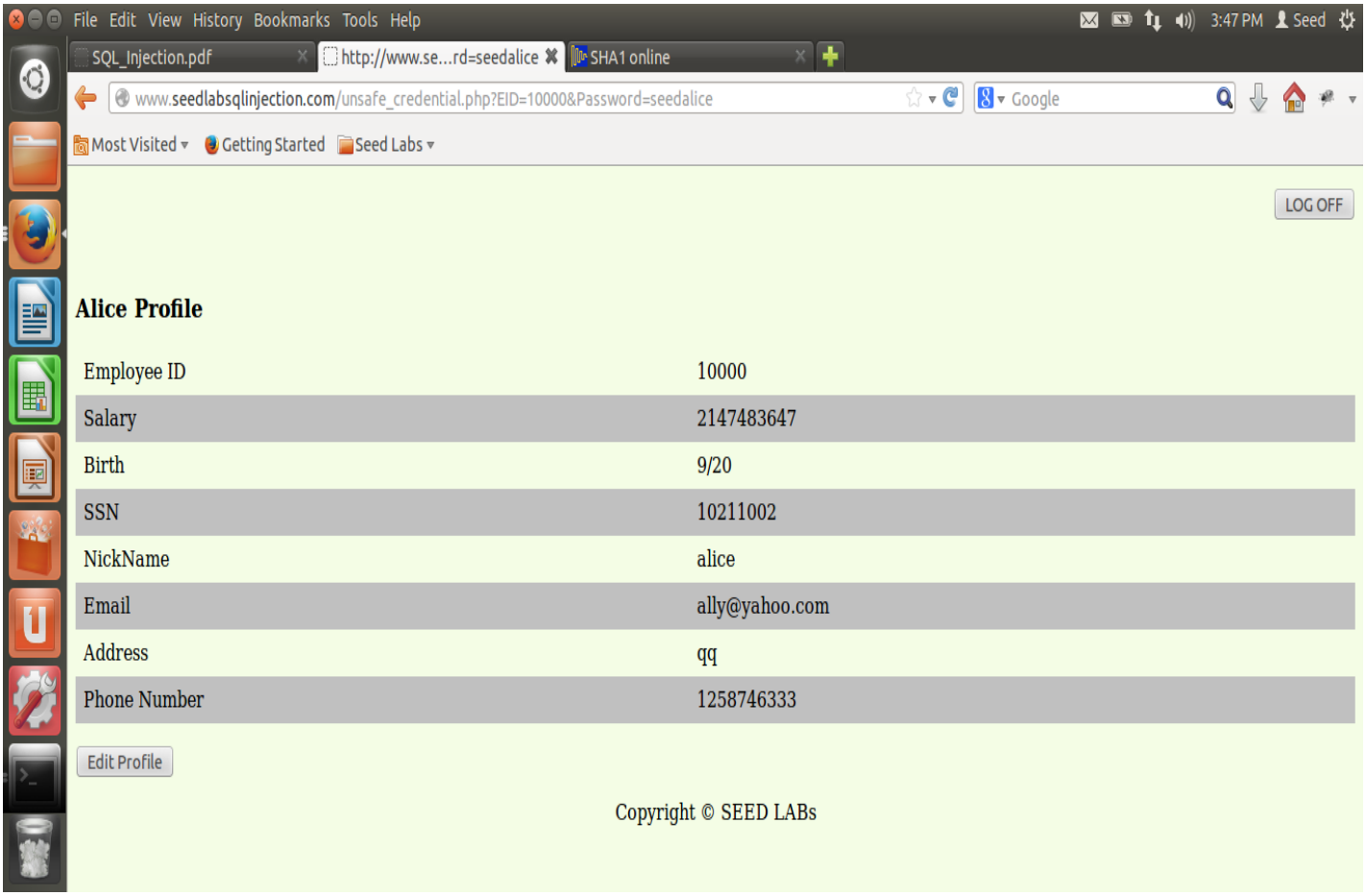


Fig: Alice edit profile information

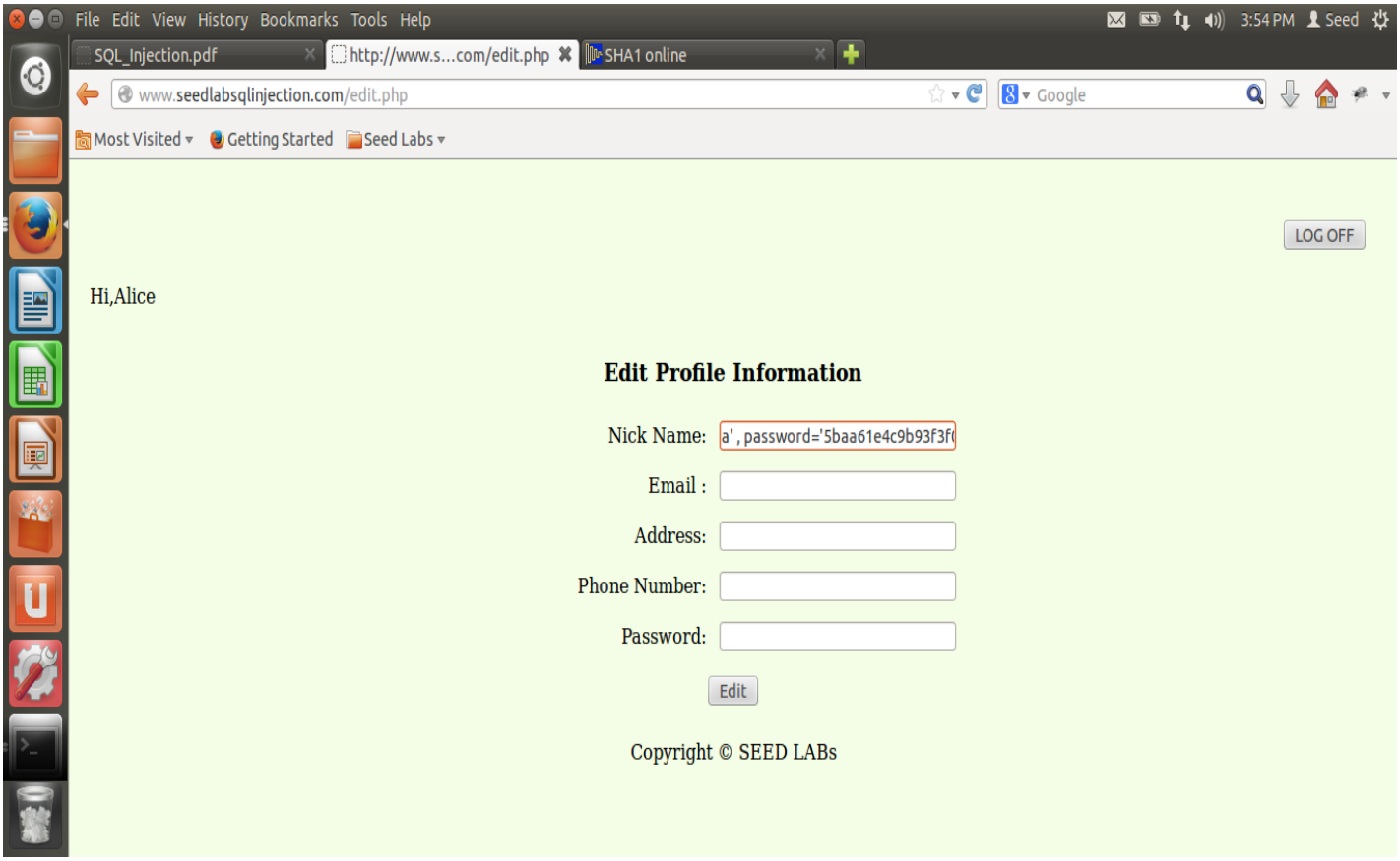


Fig: Modifying Ryan's password from Alice's profile by introducing malicious code for SQL injection attack.

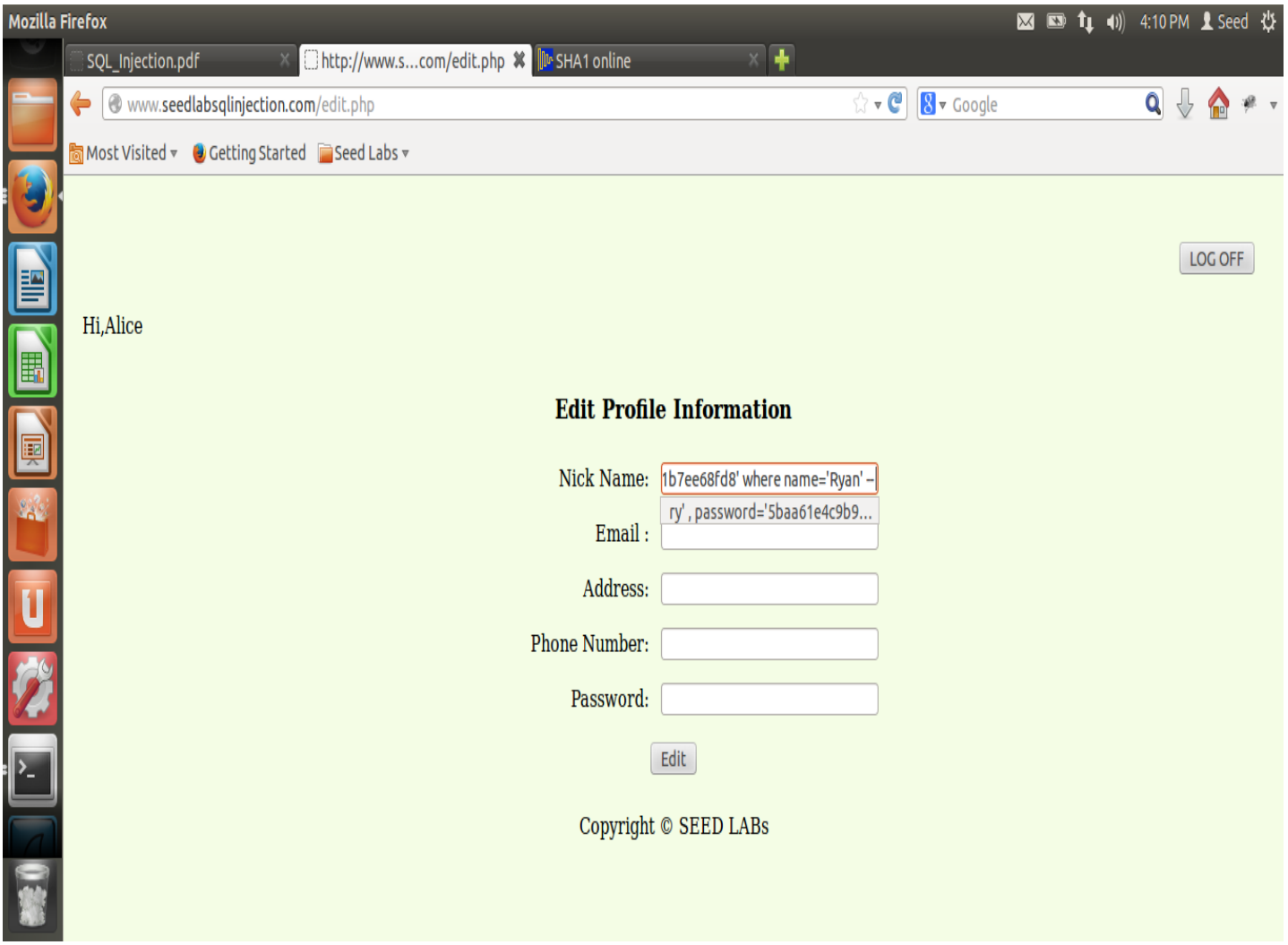


Fig: Modifying Ryan's password from Alice's profile by introducing malicious code for SQL injection attack.

- Query to perform SQLInjection attack to modify Ryan's password: a', password=' 5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8' where name='Ryan' --

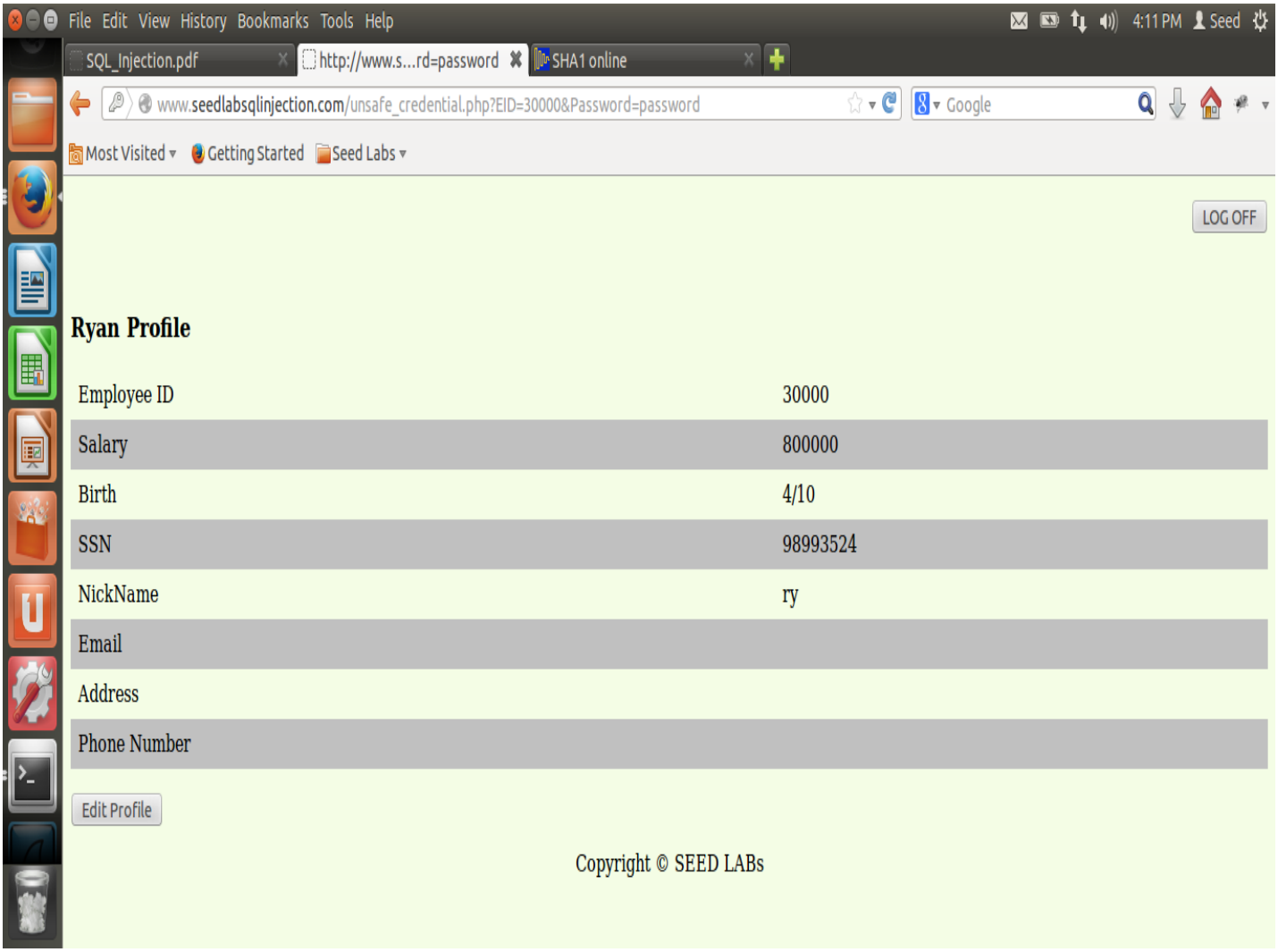


Fig: Ryan's profile with modified password.

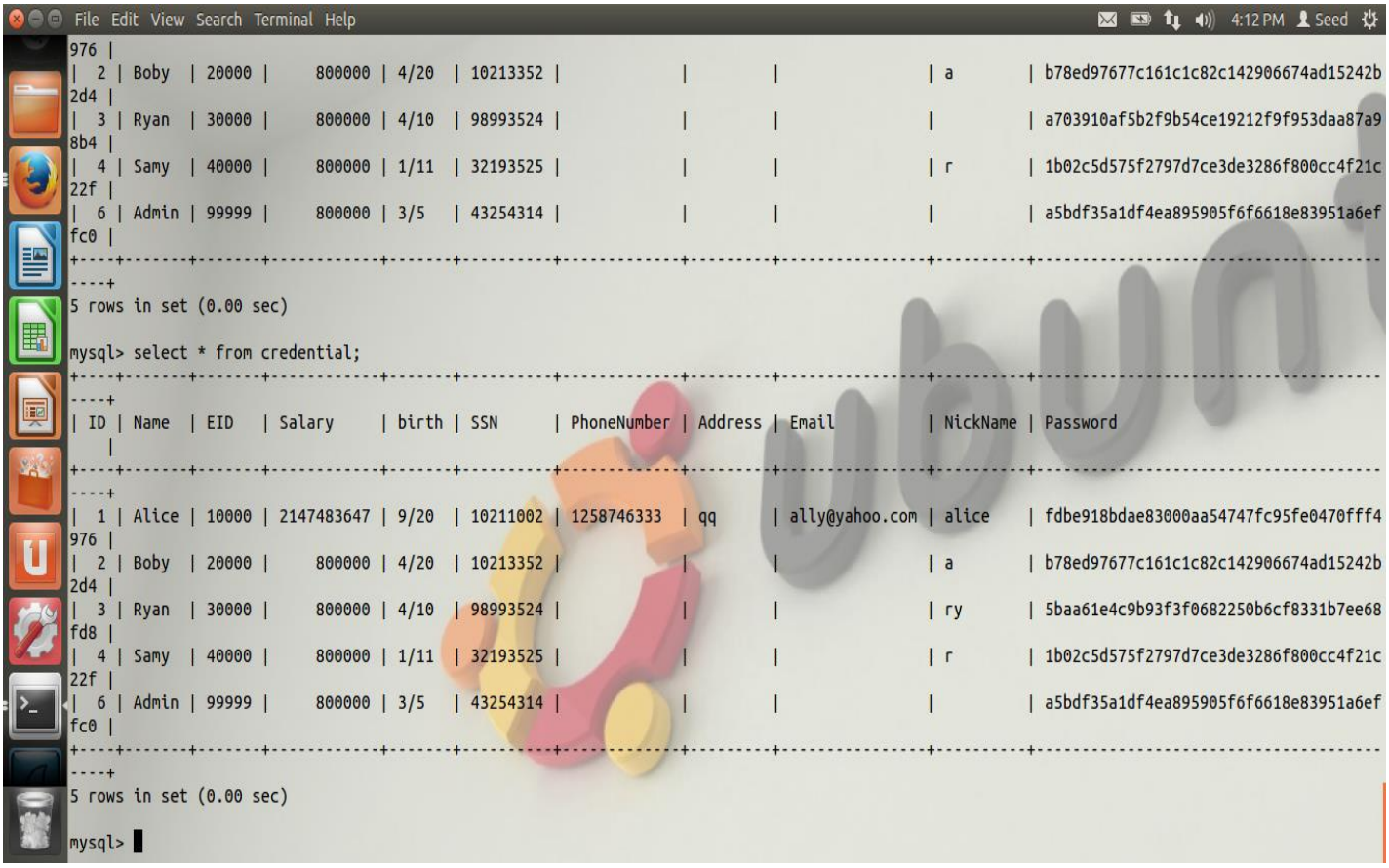


Fig: Password for Ryan is changed in the database.

Results:

- SQL Injection Attack on UPDATE Statement to modify Ryan's password from Alice's profile.

3.4 Task 4: Countermeasure — Prepared Statement

Objective: Use prepared statement as a countermeasure for SQLInjection attacks.

Procedure:

- Using prepared statement for queries, the actual data and code is separated as shown in the screenshot of code below.
- 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.
- The actual data are replaced by question marks (?).
- After this step, we then send the data to the database using bind param() as shown below in the screenshot.
- With the parameters bound, we used $stmt->execute(), to execute the prepared query.

- To retrieve the results of the query, they must also be bound: $stmt->bind_result($output_1, $output_2, ..., $output_n), where the bound variables match the data expected to be returned from the query.
- Finally, actually getting the query's results requires calling $results=$stmt->fetch().

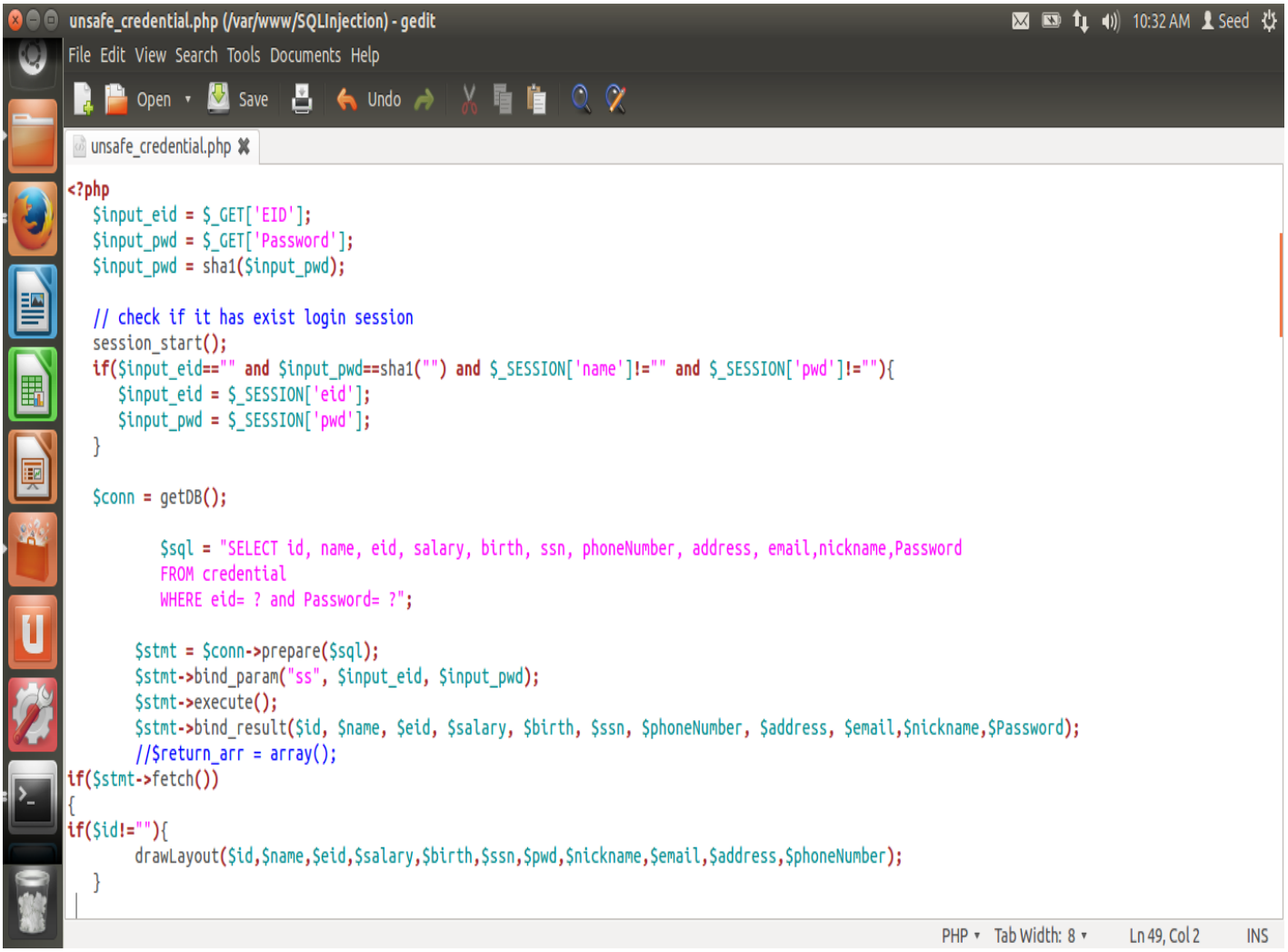
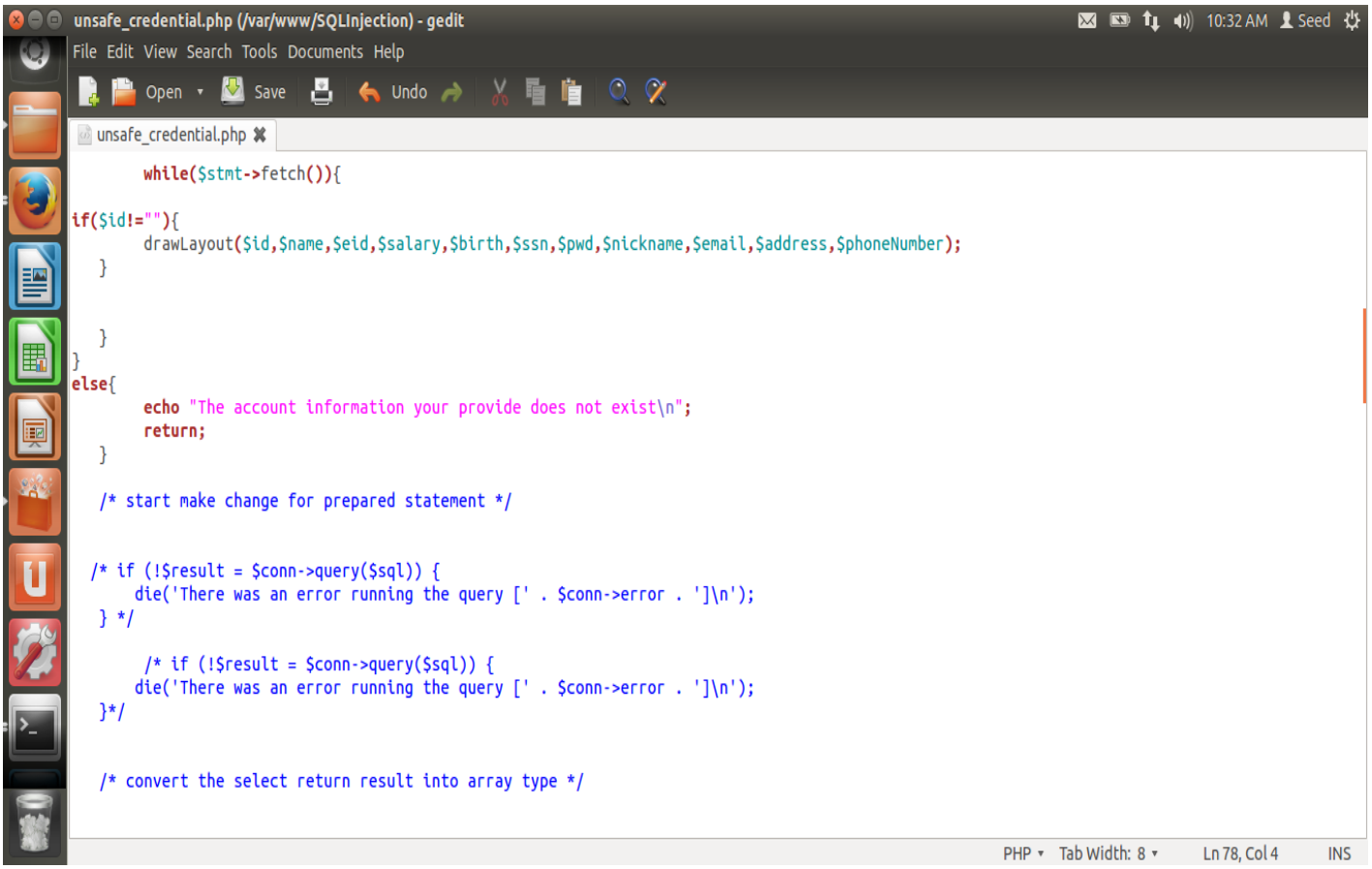
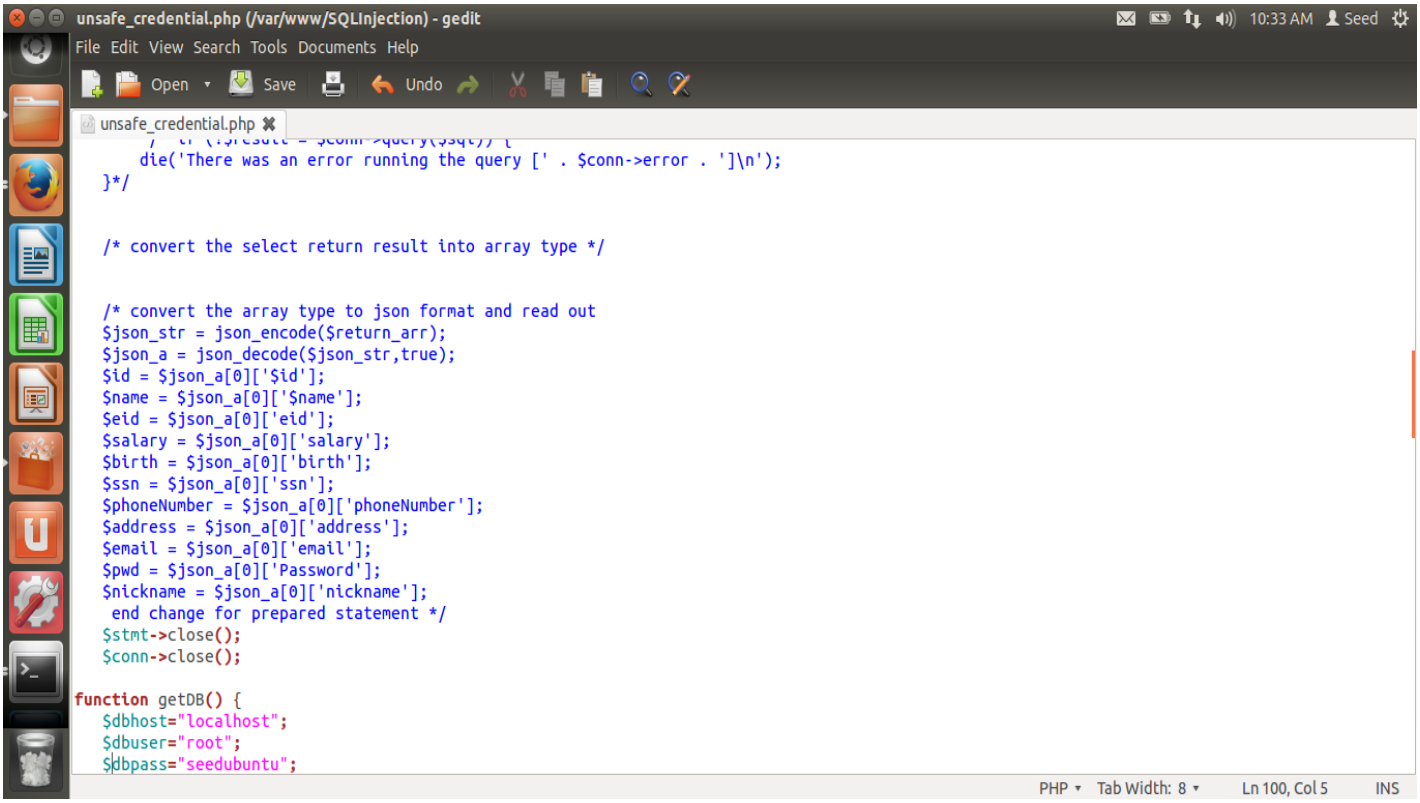


Fig: unsafe_credential.php with prepared statement code.


unsafe_credential.php (/var/www/SQLInjection) - gedit
10:32 AM Seed
File Edit View Search Tools Documents Help
Open Save Undo
unsafe_credential.php
while($stmt->fetch()){
if($id!=""){
drawLayout($id,$name,$eid,$salary,$birth,$ssn,$pwd,$nickname,$email,$address,$phoneNumber);
}
}
else{
echo "The account information your provide does not exist\n";
return;
}
/* start make change for prepared statement */
/* if (!$result = $conn->query($sql)) {
die('There was an error running the query [' . $conn->error . ']\n');
} */
/* if (!$result = $conn->query($sql)) {
die('There was an error running the query [' . $conn->error . ']\n');
}*/
/* convert the select return result into array type */
PHP Tab Width: 8 Ln 78, Col 4 INS


unsafe_credential.php (/var/www/SQLInjection) - gedit
10:33 AM Seed
File Edit View Search Tools Documents Help
Open Save Undo
unsafe_credential.php
die('There was an error running the query [' . $conn->error . ']\n');
}*/
/* convert the select return result into array type */
/* 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'];
$birth = $json_a[0]['birth'];
$ssn = $json_a[0]['ssn'];
$phoneNumber = $json_a[0]['phoneNumber'];
$address = $json_a[0]['address'];
$email = $json_a[0]['email'];
$pwd = $json_a[0]['Password'];
$nickname = $json_a[0]['nickname'];
end change for prepared statement */
$stmt->close();
$conn->close();
function getDB() {
$dbhost="localhost";
$dbuser="root";
$dbpass="seedubuntu";
PHP Tab Width: 8 Ln 100, Col 5 INS

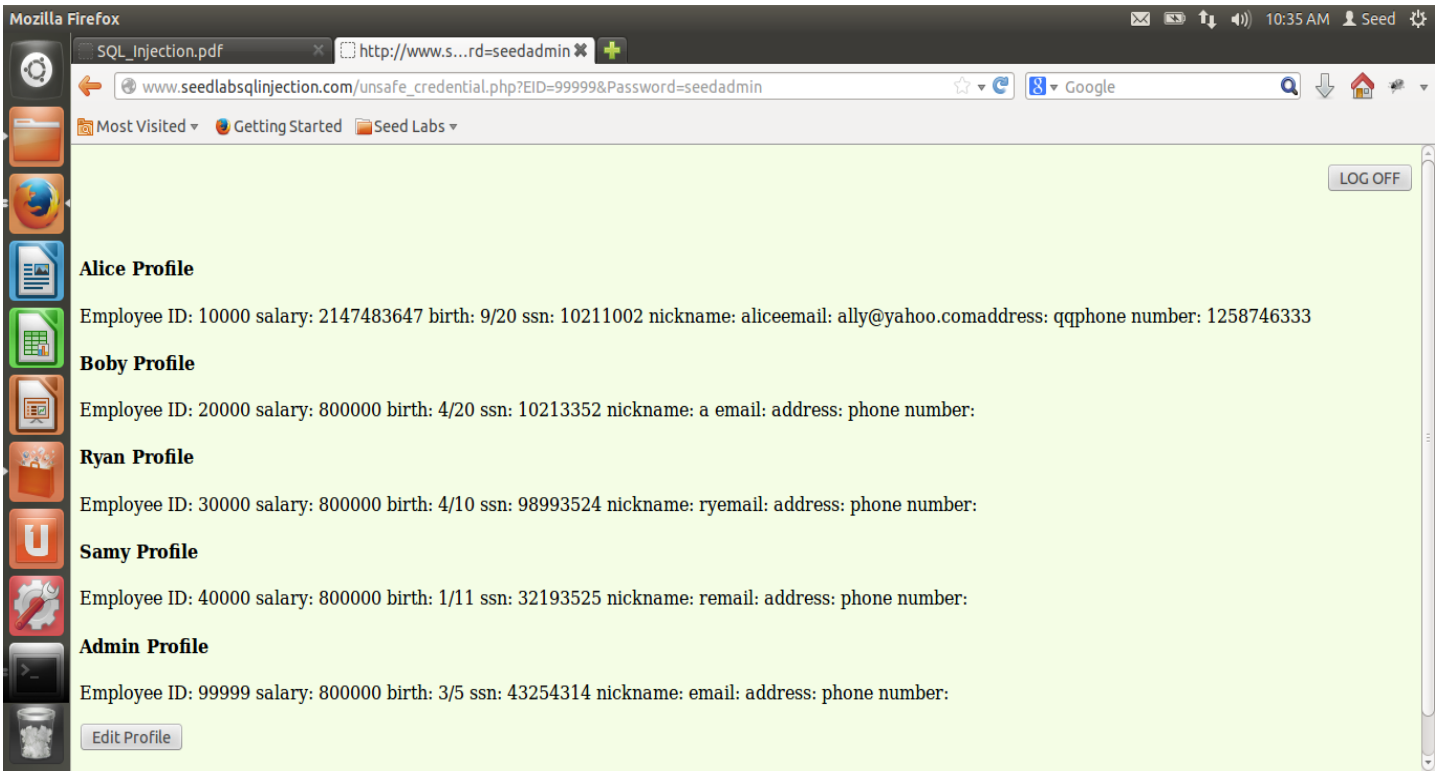


Fig: Admin access for prepared statement with correct admin ID and password

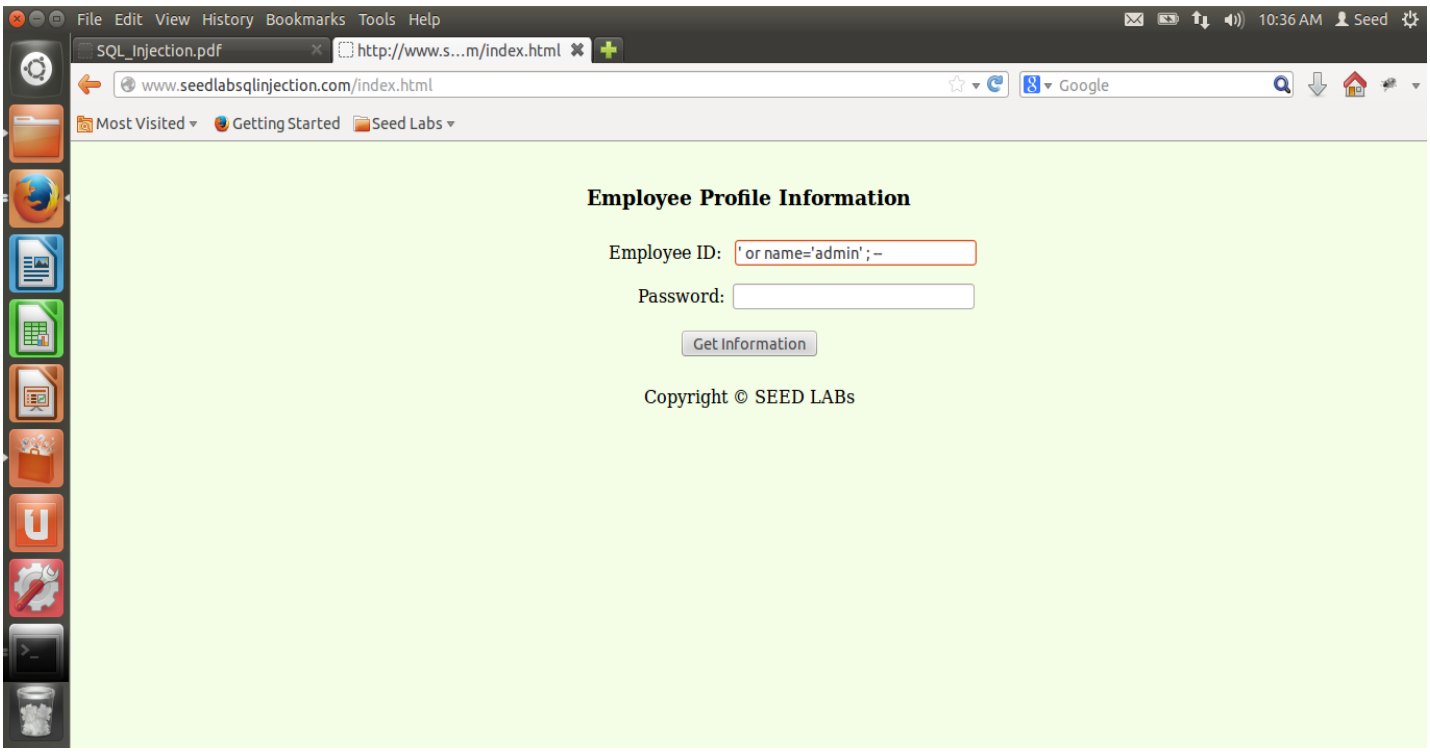


Fig: Trying SQLInjection to get admin access without knowing ID or password for prepared statement code i.e trying task 1 with prepared statement.

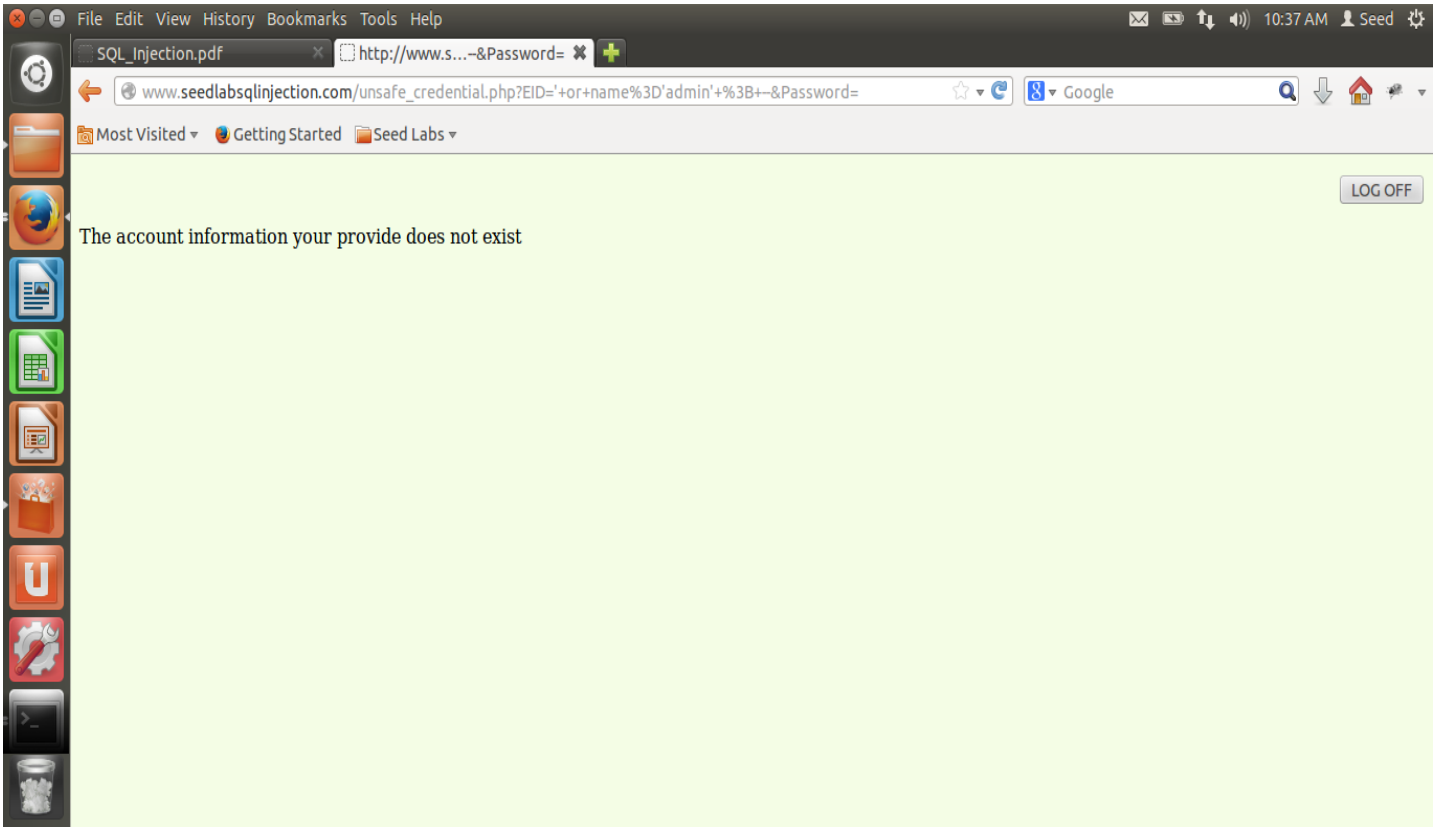


Fig: No access is achieved.

Results:

- The query is written using prepared system.
- Prepared statement is used to separate data and code.
- Tried SQLInjection task 1 i.e. to get admin access without knowing admin's employee ID or password.
- Access is not achieved because of prepared statement.
- Prepared statement is used as a countermeasure to prevent SQLInjection attacks
- Prepared statements allow a developer to separate SQL logic from user input logic.
- With this separation, user input types can be explicitly specified, making them strongly typed as far as the database is concerned.

Task 3.4: Use prepared statement for task 2 i.e. to modify another person's salary

Objective: To use prepared statement in the query in EDIT profile to see if an attacker can modify victim's salary or not.

Procedure:

- Modify unsafe_edit.php using prepared statements as shown in the screenshot of the code above.
- Go to alice's profile and edit it.

- Check if prepared statement is useful and is working well enough to edit/update Alice's information.

Now, try as an attacker. Alice is the attacker who wishes to increase her salary as shown in the screenshot below.

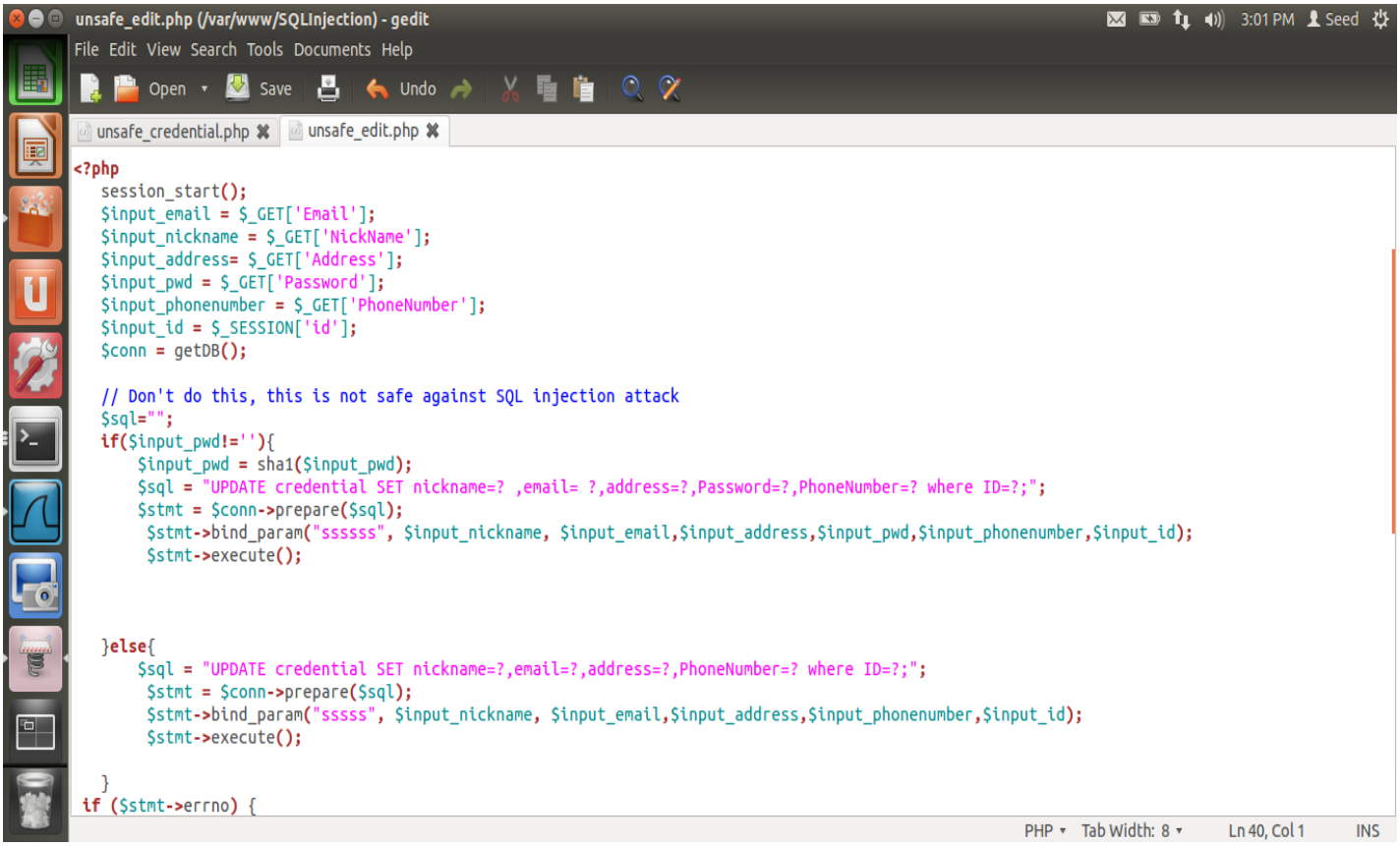
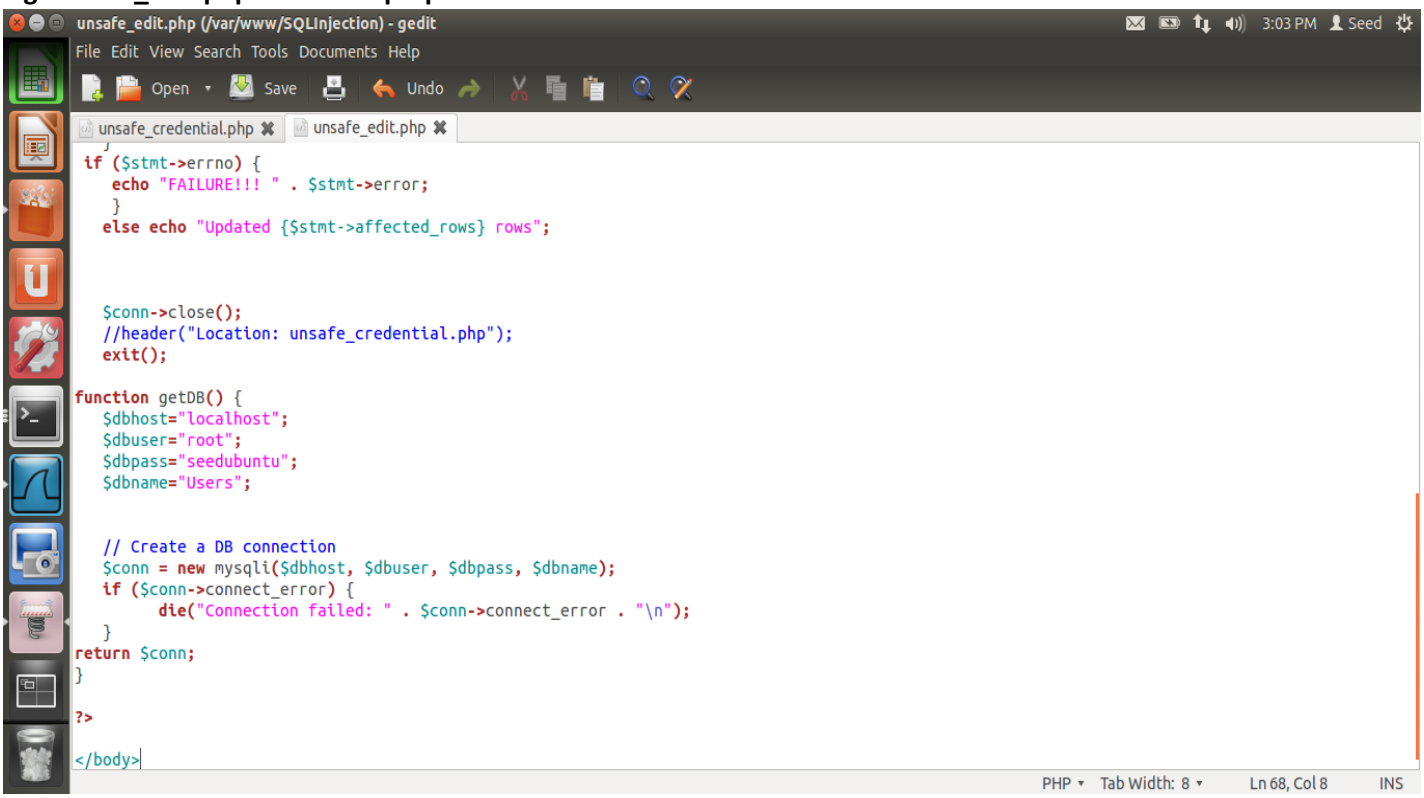


Fig: unsafe_edit php code with prepared statement



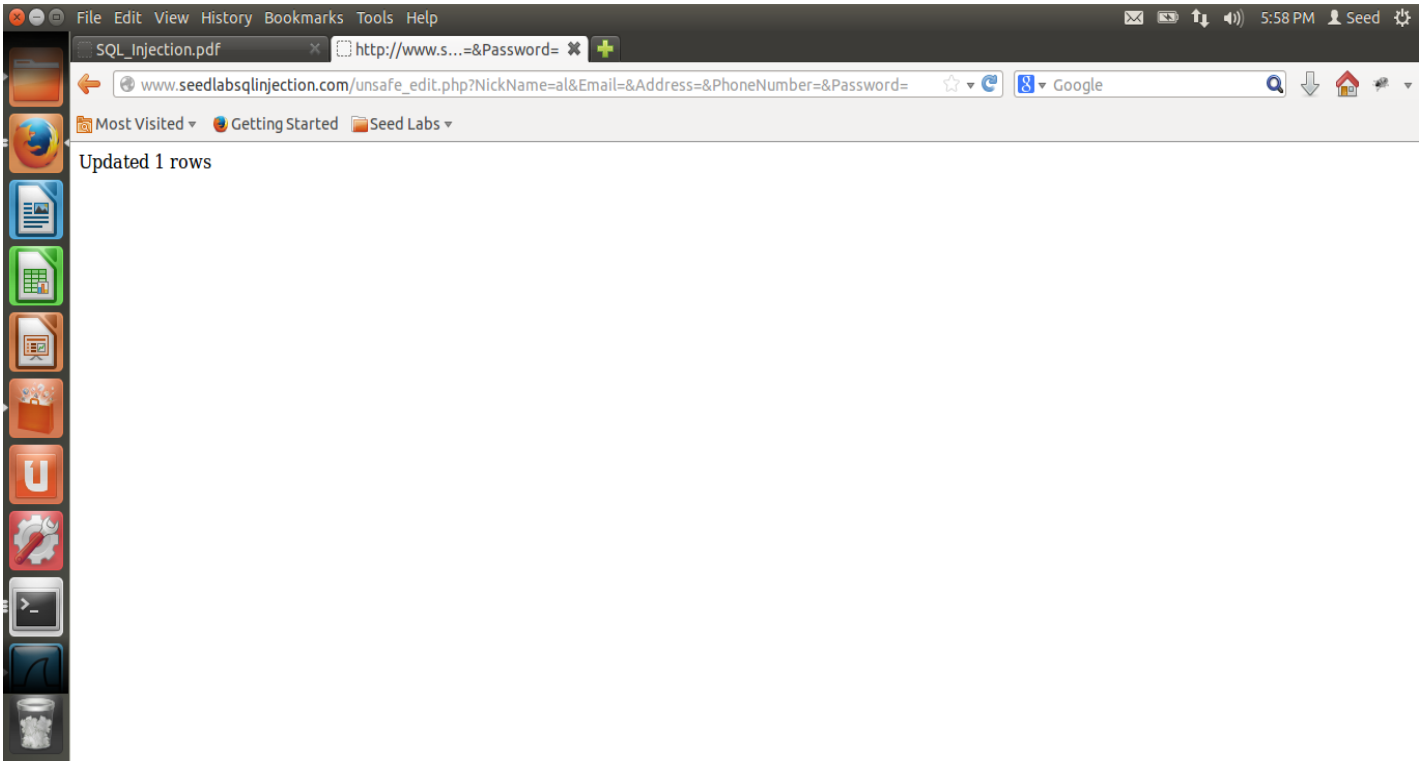


Fig: Prepared statement worked perfectly to edit the nickname of alice.

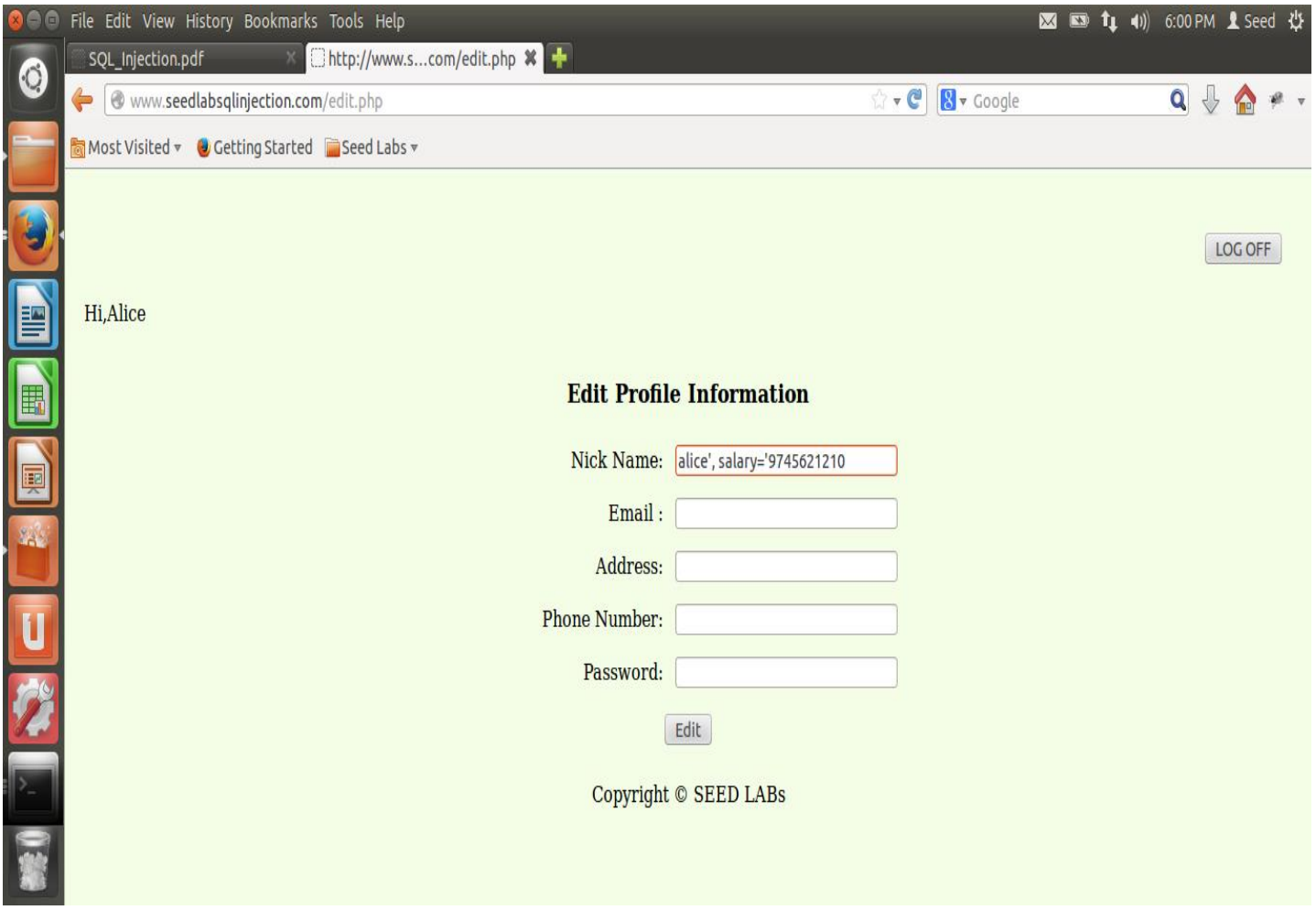


Fig: Trying the SQLInjection attack to modify salary of Alice for prepared statement code.

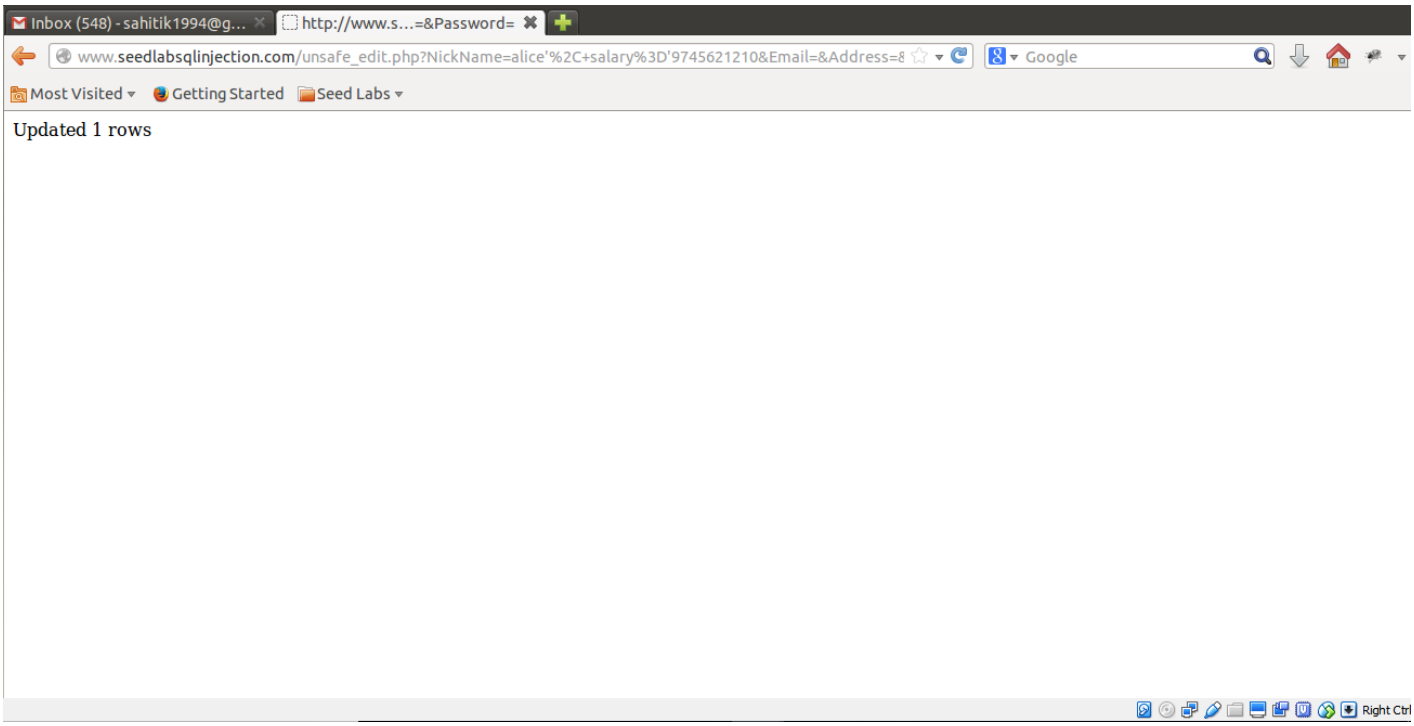


Fig: The above SQL injection injection attack tries to edit the nickname field instead of attacking.

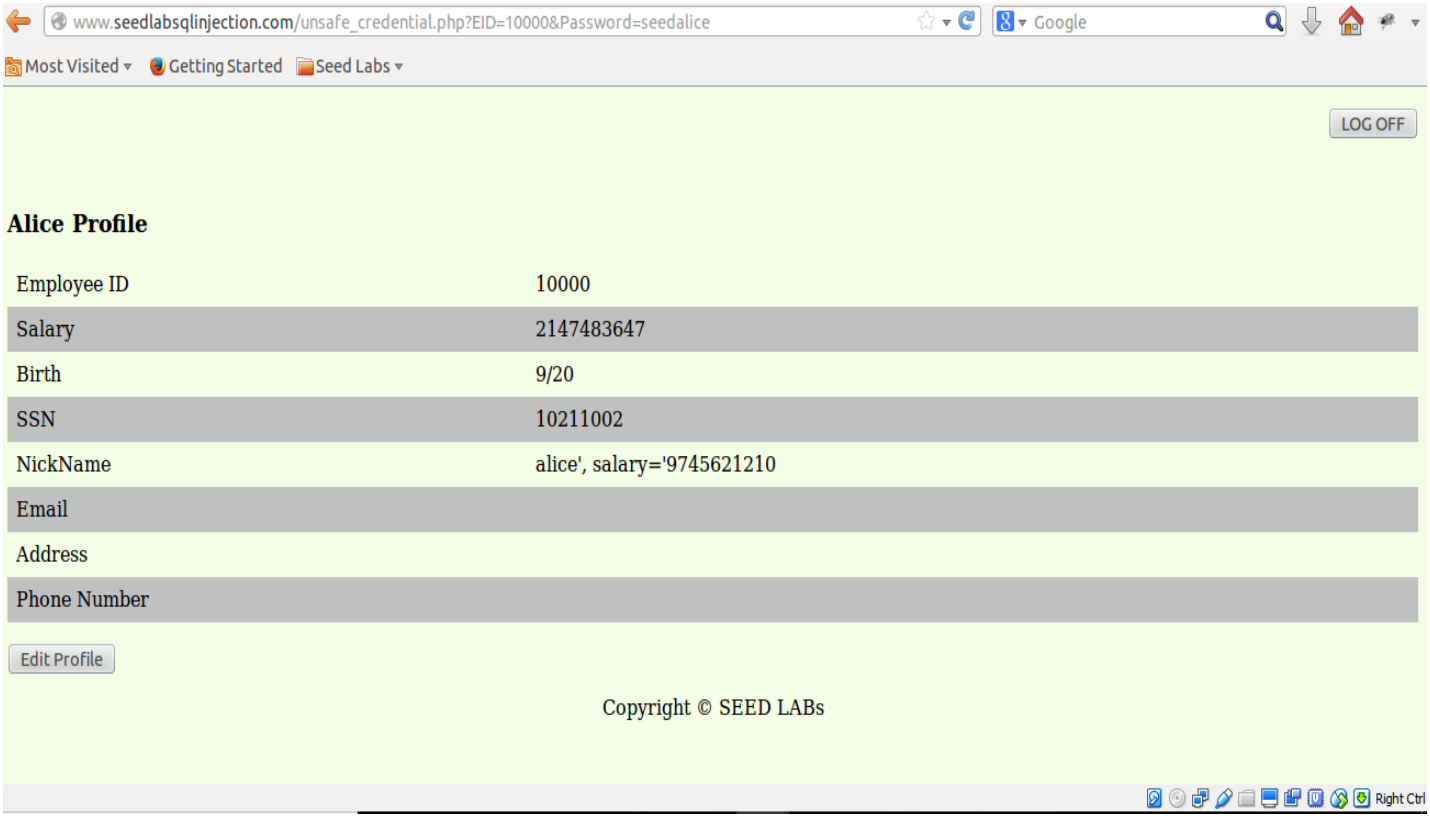


Fig: Salary of Alice is not modified. Instead of the SQL injection, the nickname is only modified.

Results:

- Code is modified with prepared statement as shown above.
- Tried normal editing of nickname of Alice. Alice nickname got modified as shown in the screenshot above upon editing with prepared statement code.
- Now, as an attacker(Alice) tried to modify her salary.
- As seen from the above screenshots, salary is not modified due to the countermeasure of using prepared statement to prevent SQLInjection attacks.
- Instead of performing SQL Injection attack, it performs normal edit operation which updates the nickname.

Task 3.4: Use prepared statement for task 2 i.e. to modify another person's salary

Objective: To use prepared statement in the query in EDIT profile to see if an attacker can modify another victim's password or not.

Procedure:

- Modify unsafe_edit.php using prepared statements as shown in the screenshot of the code above.
- Now, try as an attacker. Alice is the attacker who wishes to modify Ryan's salary as shown in the screenshot below.
- old password: password
- Generate hash value for new password: hey : 7f550a9f4c44173a37664d938f1355f0f92a47a7

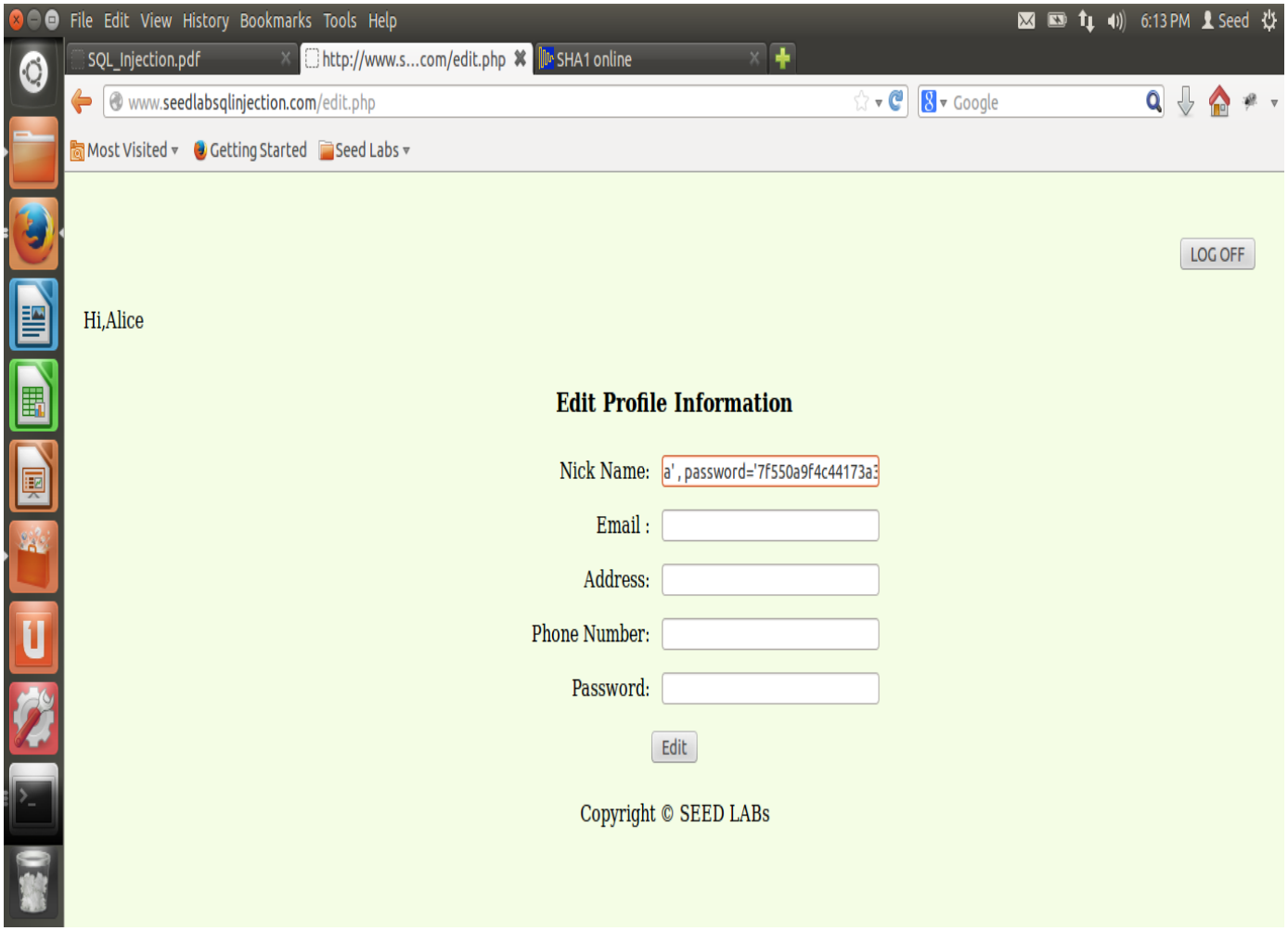


Fig: Alice as an attacker is trying to change Ryan's password as "hey" using has value of hey through SQLInjection for prepared statements.

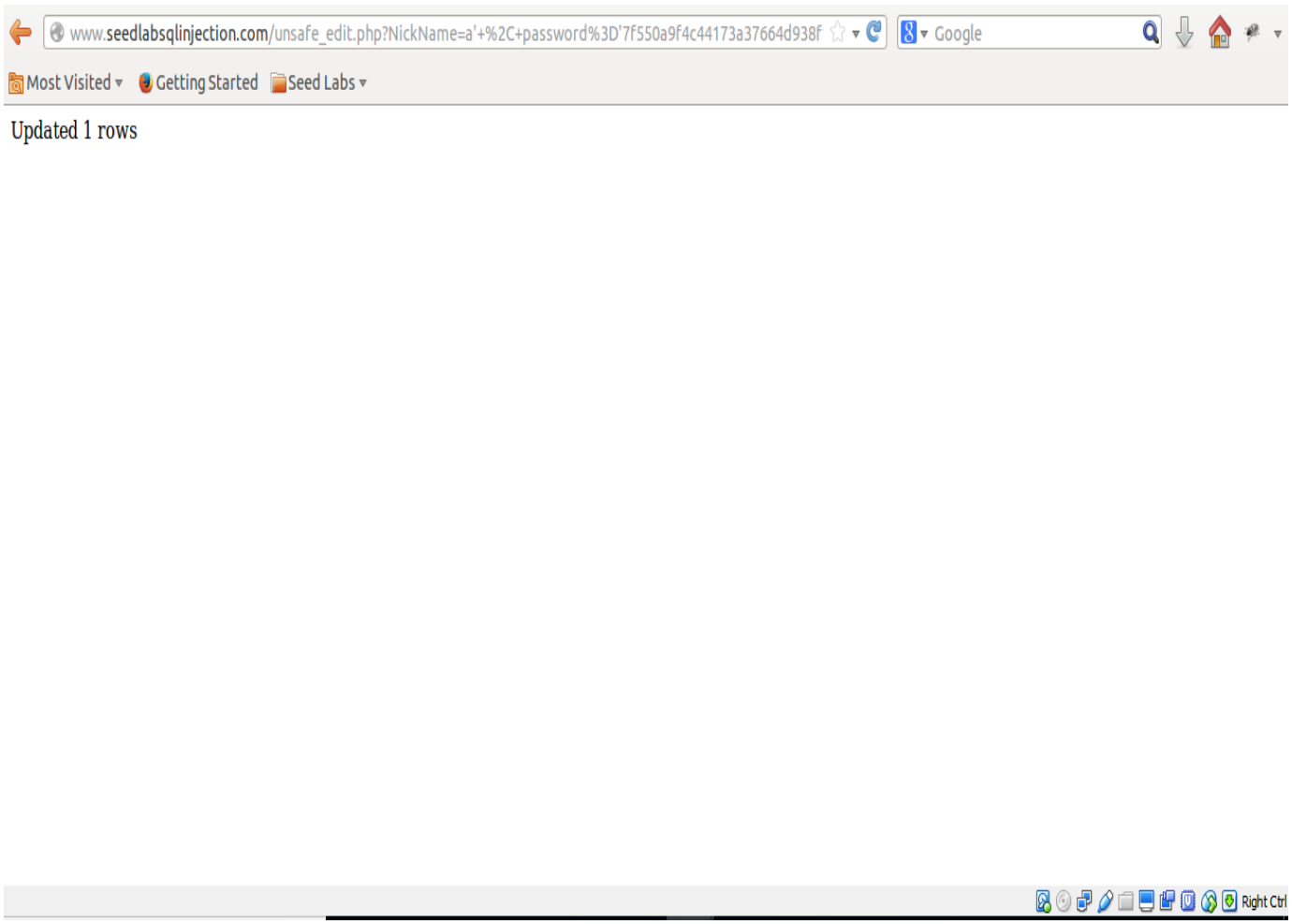


Fig: Instead of performing the SQLInjection attack, normal edit profile function is performed which results in updating the row.

Fig: The prepared statement does not allow for SQLinjection attack as the prepared statement code is used in unsafe_edit.php which prevents the attack. Instead it performs normal edit operations and updates the nickname.

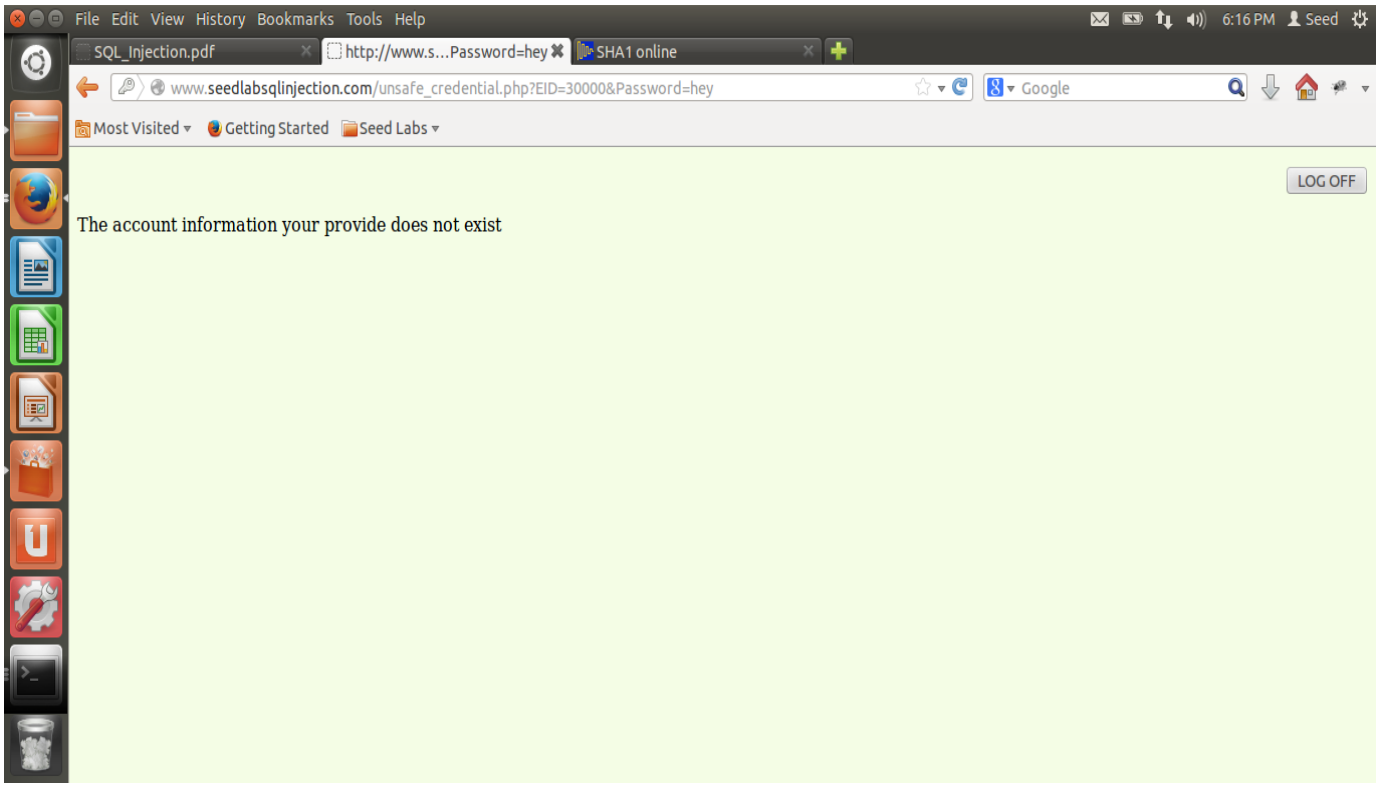


Fig: Alice could not modify Ryan's password.

Results:

- The attacker(Alice) tried to modify Ryan's password as "hey" using its hash value as data in the database for password is stored in the form of hash values.
- The SQLInjection technique is used as shown in the screenshot.
- However, due to the prepared statements used in the code, the attacker could modify the victim's password.
- Instead, it performs normal edit operation and updates the nickname as shown in the screenshot.
- This shows that prepared statements are used as an effective countermeasure against SQL injection attacks.