

# SQL Injection Attack Lab

## Table of Contents:

<b>Overview</b>	<b>1</b>
<b>Lab Environment</b>	<b>2</b>
<b>Container Setup and Commands</b>	<b>2</b>
<b>About the Web Application</b>	<b>3</b>
<b>Task 1: Get Familiar with SQL Statements</b>	<b>4</b>
<b>Task 2: SQL Injection Attack on SELECT Statement</b>	<b>4</b>
<b>Task 2.1: SQL Injection Attack from webpage</b>	<b>6</b>
<b>Task 2.2: SQL Injection Attack from command line</b>	<b>6</b>
<b>Task 2.3: Append a new SQL statement</b>	<b>6</b>
<b>Task 3: SQL Injection Attack on UPDATE Statement</b>	<b>7</b>
<b>Task 3.1: Modify your own salary</b>	<b>7</b>
<b>Task 3.2: Modify other people' salary</b>	<b>8</b>
<b>Task 3.3: Modify other people' password</b>	<b>8</b>
<b>Task 4: Countermeasure — Prepared Statement</b>	<b>9</b>
<b>Submission</b>	<b>9</b>

# 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 types of attacks. This lab covers the following topics:

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

**This lab has been tested within the seed labs 20.04 VM. You are to download the Labsetup.zip file within this VM and perform the tasks here. The Labsetup.zip file can be found at [https://seedsecuritylabs.org/Labs\\_20.04/Web/Web\\_SQL\\_Injection/](https://seedsecuritylabs.org/Labs_20.04/Web/Web_SQL_Injection/).**

## Lab Environment

We have developed a web application for this lab, and we use containers to set up this web application. There are two containers in the lab setup, one for hosting the web application, and the other for hosting the database for the web application. The IP address for the web application container is 10.9.0.5, and The URL for the web application is the following:

```
http://www.seed-server.com
```

We need to map this hostname to the container's IP address. Please add the following entry to the **/etc/hosts** file. You need to use the root privilege to change this file (using sudo). It should be noted that this name might have already been added to the file due to some other labs. If it is mapped to a different IP address, the old entry must be removed.

```
10.9.0.5 www.seed-server.com
```

## Container Setup and Commands

Please download the Labsetup.zip file to your VM from teams or the seed labs website, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment using the commands:

```
$ docker-compose build
$ docker-compose up
```

**MySQL database.** Containers are usually disposable, so once it is destroyed, all the data inside the containers are lost. For this lab, we do want to keep the data in the MySQL database, so we do not lose our work when we shutdown our container. To achieve this, we have mounted the mysql data folder on the host machine (inside Labsetup, it will be created after the MySQL container runs once) to the /var/lib/mysql folder inside the MySQL container. This folder is where MySQL stores its database.

Therefore, even if the container is destroyed, data in the database is still kept. **If you do want to start from a clean database**, you can remove this folder:

```
$ sudo rm -rf mysql_data
```

## About the Web Application

We have created a web application, which 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 privileged 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 Table 1:

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				

Table 1: Database

# 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. The data used by our web application is stored in a MySQL database, which is hosted on our MySQL container. We have created a database called sqllab 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. **Please get a shell on the MySQL container called mysql-10.9.0.6.** Then use the mysql client program to interact with the database. The user name is root and password is dees.

```
$ dockps
```

```
$ docksh [container id]
```

```
[02/15/23] seed@VM:~/.../Labsetup$ dockps  
88af3384118b  www-10.9.0.5  
34cbb7755d31  mysql-10.9.0.6
```

```
[02/15/23] seed@VM:~/.../Labsetup$ docksh 34cbb7755d31  
root@34cbb7755d31:/# mysql -u root -pdees
```

Inside the mysql container that we just got a shell in run:

```
$ mysql -u root -pdees
```

After login, you can create a new database or load an existing one. As we have already created the sqllab users database for you, you just need to load this existing database using the use command. To show what tables are there in the sqllab users database, you can use the show tables command to print out all the tables of the selected database.

```
mysql> use sqllab_users;
```

```
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 a screenshot of your results.

```
mysql> select * from credential where Name='Alice';
```

## 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 to 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.seed-server.com](http://www.seed-server.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 authenticates 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.



Figure 1: Login page

To help you get 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/SQL_Injection` directory, is used to conduct user authentication. The following code snippet shows 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.

## 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 know the password. You need to decide what to type in the Username and Password fields to succeed in the attack. Enter the following text in the username field and leave the password field blank or filled with any arbitrary value:

**Admin'#**

## 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 command shows how to send an HTTP GET request to our web application, with two parameters (username and Password) attached:

**\$ curl 'www.seed-server.com/unsafe\_home.php?username=alice&Password=11'**

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 a 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.

Run the following command in a terminal to send the HTTP request:

```
$ curl 'http://www.seed-server.com/unsafe_home.php?username=Admin%27%23&Password='
```

## 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 try to run two SQL statements via the login page.

There is a countermeasure preventing you from running two SQL statements in this attack. Please use the SEED book or resources from the Internet to figure out what this countermeasure is, and describe your discovery in the lab report.

Enter the following text in the username field and leave the password field blank or filled with any arbitrary value and describe your results:

**Admin'; select 1;#**

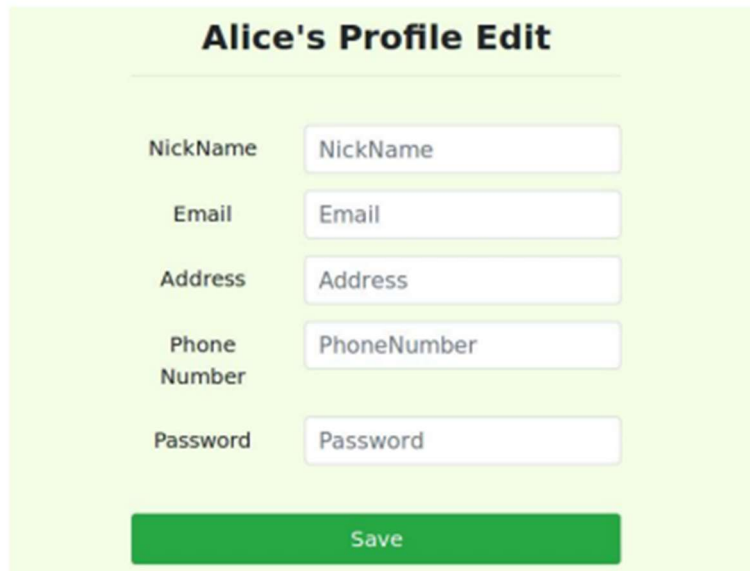
## Task 3: SQL Injection Attack on UPDATE Statement

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);
```



The image shows a web form titled "Alice's Profile Edit". It contains five input fields, each with a label to its left: "NickName", "Email", "Address", "Phone Number", and "Password". The input fields are white with rounded corners and a light gray border. Below the input fields is a green button with the text "Save" in white. The entire form is set against a light green background.

Figure 2: The edit profile page

### 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 Bobby 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.

Login to the alice user using the username **alice** and password **seedalice**. Paste the following text into the Phone number field of Alice's Edit profile page:

**123', salary = 99999 where Name='Alice'##**

### Task 3.2: Modify other people's salary



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

Paste the following text into the Phone number field of Alice's Edit profile page:

**123', salary = 1 where Name='Bobby'#**

**Verify that the salary has been changed and provide screenshots.**

## Task 3.3: Modify other people's password

After changing Bobby's salary, you are still disgruntled, so you want to change Bobby'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 Bobby'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 the password is being stored. It uses the SHA1 hash function to generate the hash value of the password.

Use the link below to calculate the SHA1 hash value of the password you wish to change Bobby's password to.

<https://xorbin.com/tools/sha1-hash-calculator>

Paste the following text into the Phone number field of Alice's Edit profile page:

**123', Password ='[ SHA 1 Hash of the password ]' where Name='Bobby'#**

## 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 were 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 statements*.

To understand how prepared statements prevent 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.

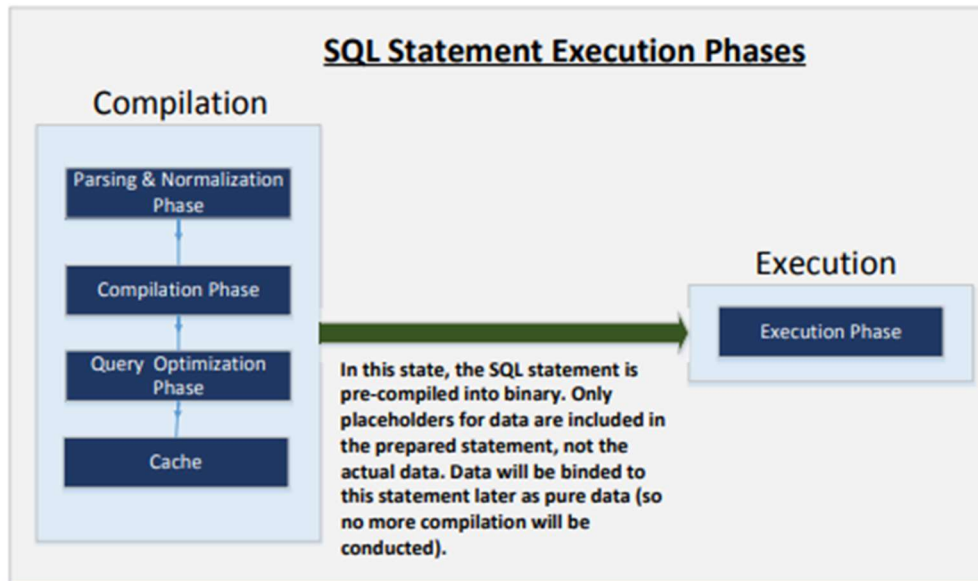


Figure 3: Prepared Statement Workflow

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, the query is interpreted. In the query optimization phase, a number of different plans are considered to execute the query, out of which the best optimized plan is chosen. The chosen plan is stored 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 needs to be provided, but this 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 a 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 statements 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 data. This is the preparation step. As we can see from the above code snippet, the actual data is 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.

**Task.** In this task, we will use the prepared statement mechanism to fix the SQL injection vulnerabilities. For the sake of simplicity, we created a simplified program inside the defense folder. We will make changes to the files in this folder. If you point your browser to the following URL, you will see a page similar to the login page of the web application. This page is where we will test whether our prepared statement works as intended or not.

URL: <http://www.seed-server.com/defense/>

The data typed in this login page will be sent to the server program `getinfo.php`, which invokes a program called `unsafe.php`. The SQL query inside this PHP program is vulnerable to SQL injection attacks. Your job is to modify the SQL query in `unsafe.php` using the prepared statement, so the program can defeat SQL injection attacks. Inside the lab setup folder, the `unsafe.php` program is in the `image_www/Code/defense` folder. You can directly modify the program there. After you are done, you need to rebuild and restart the container, or the changes will not take effect.

Within your labsetup folder locate the following directory:  
**image\_www/Code/defense**

Inside this directory you will find the `unsafe.php` file that needs to be modified. First comment out the following lines in this file:

```
$result = $conn->query("SELECT id, name, eid, salary, ssn
                        FROM credential
                        WHERE name= '$input_uname' and Password= '$hashed_pwd'");
if ($result->num_rows > 0) {
    // only take the first row
    $firstrow = $result->fetch_assoc();
    $id       = $firstrow["id"];
    $name     = $firstrow["name"];
    $eid      = $firstrow["eid"];
    $salary   = $firstrow["salary"];
    $ssn      = $firstrow["ssn"];
}
```

Now paste the following code right after the previous if condition that was just commented out.

```
$result = $conn->prepare("SELECT id, name, eid, salary, ssn FROM
credential WHERE name= ? and Password= ?");
$result->bind_param("ss", $input_uname, $hashed_pwd);
$result->execute();
$result->bind_result($id, $name, $eid, $salary, $ssn);
$result->fetch();
$result->close();
```

Save your changes and run the following commands:

```
$ docker-compose down
$ docker-compose build --no-cache
$ docker-compose up
```

Now in the username field for the following URL try the attack using the same input used for **Task 2.1** and report your observations.

<http://www.seed-server.com/defense/>

## Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.