

Lab 5 – SQL Injection

Goal: To fully understand the weakness in SQL semantics and know how to exploit the vulnerabilities in the interface between web applications and database servers, for retrieval of un-allowed data.

Instructions: Please refer to attached lab instructions with this document.

Deliverable: A lab report, an electronic submission to **Canvas**, is expected to **document** and **explain** all the **injections** that you use, and **include the screen shots** when you achieve the major milestones in the lab, such as when you retrieve the data based on injections. A demo may be requested when necessary.

Requirement: The report will all be evaluated based on the following grading criteria.

Correctness	25%
Completeness	25%
Clarity	25%
Quality of English writing	25%

SQL Injection Attack Lab

Copyright © 2006 - 2016 Wenliang Du, Syracuse University.

The development of this document was partially funded by the National Science Foundation under Award No. 1303306 and 1318814. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. A human-readable summary of (and not a substitute for) the license is the following: You are free to copy and redistribute the material in any medium or format. You must give appropriate credit. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You may not use the material for commercial purposes.

1 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 the web applications 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. The SQL injection attack 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 type of attacks.

2 Lab Environment

You need to use our provided virtual machine image for this lab. The name of the VM image that supports this lab is SEEDUbuntu12.04.zip. If you happen to have an older version of our pre-built VM image, you need to download the most recent version, as the older version does not support this lab. Go to our SEED web page (<http://www.cis.syr.edu/~wedu/seed/>) to get the SEEDUbuntu VM image.

2.1 Environment Configuration

In this lab, we need three tools, which should be installed in the provided SEEDUbuntu VM image: (1) the Firefox web browser, (2) the Apache web server, and (3) the Employee Management web application. For the browser, we need to use the LiveHTTPHeaders extension for Firefox to inspect HTTP requests and responses. The pre-built SEEDUbuntu VM image provided to you has already installed the Firefox web browser with the required extensions. The Employee Management application is not installed in the VM, but we will show you later how to install it.

Starting the Apache Server. The Apache web server is also included in the pre-built Ubuntu image and is started by default. The following command is used to start the web server manually.

```
% sudo service apache2 start
```

Configuring DNS. We have configured the following URL needed for this lab. To access the URL, the Apache server needs to be started first:

```
URL: http://www.SEEDLabSQLInjection.com
Folder: /var/www/SQLInjection/
```

The above URL is only accessible from inside of the virtual machine, because we have modified the `/etc/hosts` file to map the domain name of each URL to the virtual machine's local IP address (127.0.0.1). You may map any domain name to a particular IP address using `/etc/hosts`. For example, you can map `http://www.example.com` to the local IP address by appending the following entry to `/etc/hosts`:

```
127.0.0.1      www.example.com
```

If your web server and browser are running on two different machines, you need to modify `/etc/hosts` on the browser's machine accordingly to map these domain names to the web server's IP address, not to 127.0.0.1.

Configuring Apache Server. In our pre-built VM image, we have used Apache server to host all the web sites used in the lab. The name-based virtual hosting feature in Apache could be used to host several web sites (or URLs) on the same machine. A configuration file named `default` in the directory `"/etc/apache2/sites-available"` contains the necessary directives for the configuration:

1. The directive `"NameVirtualHost *"` instructs the web server to use all IP addresses in the machine (some machines may have multiple IP addresses).
2. Each web site has a `VirtualHost` block that specifies the URL for the web site and directory in the file system that contains the sources for the web site. For example, to configure a web site with URL `http://www.example1.com` with sources in directory `/var/www/Example_1/`, and to configure a web site with URL `http://www.example2.com` with sources in directory `/var/www/Example_2/`, we can use the following blocks:

```
<VirtualHost *>
    ServerName http://www.example1.com
    DocumentRoot /var/www/Example_1/
</VirtualHost>

<VirtualHost *>
    ServerName http://www.example2.com
    DocumentRoot /var/www/Example_2/
</VirtualHost>
```

You may modify the web application by accessing the source in the mentioned directories. For example, with the above configuration, the web application `http://www.example1.com` can be changed by modifying the sources in the directory `/var/www/Example_1/`.

2.2 Turn Off the Countermeasure

PHP provides a mechanism to automatically defend against SQL injection attacks. The method is called **magic quote**. Let us turn off this protection.

1. Go to `/etc/php5/apache2/php.ini`.
2. Find the line: `magic_quotes_gpc = On`.
3. Change it to this: `magic_quotes_gpc = Off`.
4. Restart the Apache server by running `"sudo service apache2 restart"`.

2.3 Patch the Existing VM to Add the Web Application

We have developed the simple Employee Management web application for this lab. The web application is used to store employee profile information. We have created several employee accounts for this application. To see all the employee's account information, you can log into `www.SEEDLabSQLInjection.com` as the administrator (the employee ID is 99999).

The SEEDUbuntu12.04 image that you have downloaded from our course website does not include this web application. You need to patch the VM for this lab. You can download the patch file called `patch.tar.gz` from our course website. The file includes the web application and a script that will install all of the required files needed for this lab. Place `patch.tar.gz` in any folder, unzip it, and run a script called `bootstrap.sh`. The VM will now be ready for this lab.

```
$ tar -zxvf ./patch.tar.gz
$ cd patch
$ chmod a+x bootstrap.sh
$ ./bootstrap.sh
```

The `bootstrap.sh` script creates a new database named `Users` in our existing SEEDUbuntu VM, loads data into the database, sets up the virtual host URL, and finally modifies the local DNS configuration. If you are interested in doing the setup manually, please refer to the Appendix section for details.

2.4 Note for Instructors

If the instructor plans to hold lab sessions for this lab, we suggest that the following background materials be covered in the lab sessions:

1. How to use the virtual machine, Firefox web browser, and the `LiveHTTPHeaders` extension.
2. Brief introduction of SQL. Only need to cover the basic structure of the `SELECT`, `UPDATE`, and `INSERT` statements. A useful online SQL tutorial can be found at `http://www.w3schools.com/sql/`.
3. How to operate the `MySQL` database (only the basics). The account information about the `MySQL` database can be found in the user manual of the VM, which can be downloaded from our SEED web page.
4. Brief introduction of `PHP`. Only need to cover the very basics. Students with a background in C/C++, Java, or other language should be able to learn this script language quite quickly.

3 Lab Tasks

We have created a web application, and host it at `www.SEEDLabSQLInjection.com`. This web application 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 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. All employee information is described in the following table.

User	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				

3.1 Task 1: MySQL Console

The objective of this task is to get familiar with SQL commands by playing with the provided database. We have created a database called `Users`, which contains a table called `credential`; the table stores the personal information (e.g. `eid`, `password`, `salary`, `ssn`, etc.) of every employee. `Administrator` is allowed to change the profile information of all employees, but each employee can only change his/her own information. In this task, you need to play with the database to get familiar with SQL queries.

MySQL is an open-source relational database management system. We have already setup MySQL in our SEEDUbuntu VM image. The user name is `root` and password is `seedubuntu`. Please login to MySQL console using the following command:

```
$ mysql -u root -pseedubuntu
```

After login, you can create new database or load an existing one. As we have already created the `Users` database for you, you just need to load this existing database using the following command:

```
mysql> use Users;
```

To show what tables are there in the `Users` database, you can use the following command to print out all the tables of the selected database.

```
mysql> show tables;
```

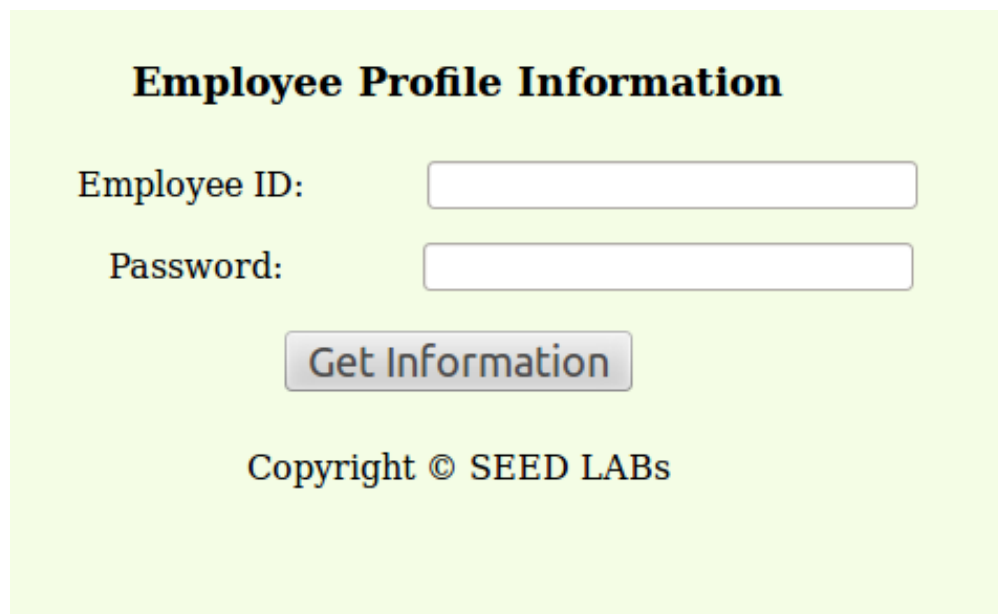
After running the commands above, you need to use a SQL command to print all the profile information of the employee `Alice`. Please provide the screenshot of your results.

3.2 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 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.

You can go to the entrance page of our web application at www.SEEDLabSQLInjection.com, where you will be asked to provide Employee ID and Password to log in. The login page is shown in

Figure 1. The authentication is based on Employee ID and Password, so only employees who know their IDs and passwords are allowed to view/update their profile information. Your job, as an attacker, is to log into the application without knowing any employee's credential.



The image shows a web form titled "Employee Profile Information" on a light green background. It contains two input fields: "Employee ID:" and "Password:". Below these fields is a button labeled "Get Information". At the bottom of the form, it says "Copyright © SEED LABs".

Figure 1: The Login Page

To help you started with this task, we explain how authentication is implemented in our web application. The PHP code `unsafe_credential.php`, located in the `/var/www/SQLInjection` directory, is used to conduct user authentication. The following code snippet show how users are authenticated.

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

// The following is psuedo code
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 variables `input_eid` and `input_pwd` hold the strings typed by users in the login page. Basically, the program checks whether any record matches with the employee ID 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 ID or the password. You need to decide what to type in the `Employee ID` and `Password` fields to succeed in the attack.
- **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 example shows how to send an HTTP GET request to our web application, with two parameters (`SUID` and `Password`) attached:

```
curl 'www.SeedLabSQLInjection.com/index.php?SUID=10000&Password=111'
```

If you need to include special characters in the `SUID` and `Password` fields, you need to encode them properly, or they can change the meaning of your requests. If you want to include 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`.

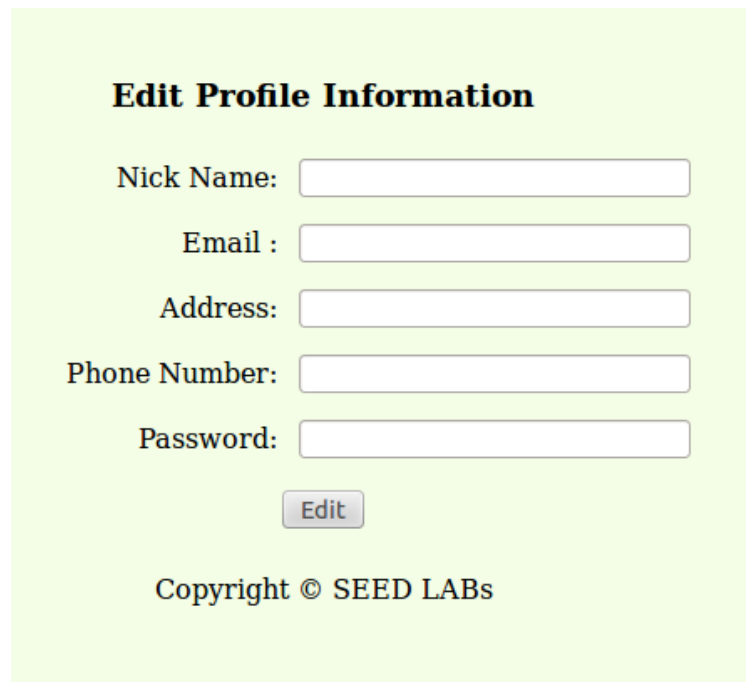
- **Task 2.3: Append a new SQL statement.** In the above two attacks, we can only steal information from the database; it will be better if we can modify the database using the same vulnerability in the login page. An idea is to use the SQL injection attack to turn one SQL statement into two, with the second one being the update or delete statement. In SQL, `semicolon (;)` is used to separate two SQL statements. Please describe how you can use the login page to get the server run two SQL statements. Try the attack to delete a record from the database, and describe your observation.

3.3 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 login 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.php` file is used to update employee's profile information. The PHP file is located in the `/var/www/SQLInjection` directory.

```
$conn = getDB();
$sql = "UPDATE credential SET nickname='$nickname',
                                email='$email',
                                address='$address',
                                phonenumber='$phonenumber',
                                Password='$pwd'
                                WHERE id= '$input_id' ";
$conn->query($sql)
```



Edit Profile Information

Nick Name:

Email :

Address:

Phone Number:

Password:

Copyright © SEED LABS

Figure 2: Edit Profile

- **Task 3.1: SQL Injection Attack on UPDATE Statement — modify 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. Only administrator is allowed to make changes to salaries. If you are a malicious employee (say Alice), your goal in this task is to increase your own salary via this Edit Profile page. We assume that you do know that salaries are stored in a column called `salary`.
- **Task 3.2: SQL Injection Attack on UPDATE Statement — modify other people's password.** Using the same vulnerability in the above UPDATE statement, malicious employees can also change other people's data. The goal for this task is to modify another employee's password, and then demonstrate that you can successfully log into the victim's account using the new password. The assumption here is that you already know the name of the employee (e.g. Ryan) on whom you want to attack. 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.php` code to see how password is being stored. It uses **SHA1** hash function to generate the hash value of password.

To make sure your injection string does not contain any syntax error, you can test your injection string on MySQL console before launching the real attack on our web application.

3.4 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 was set by the

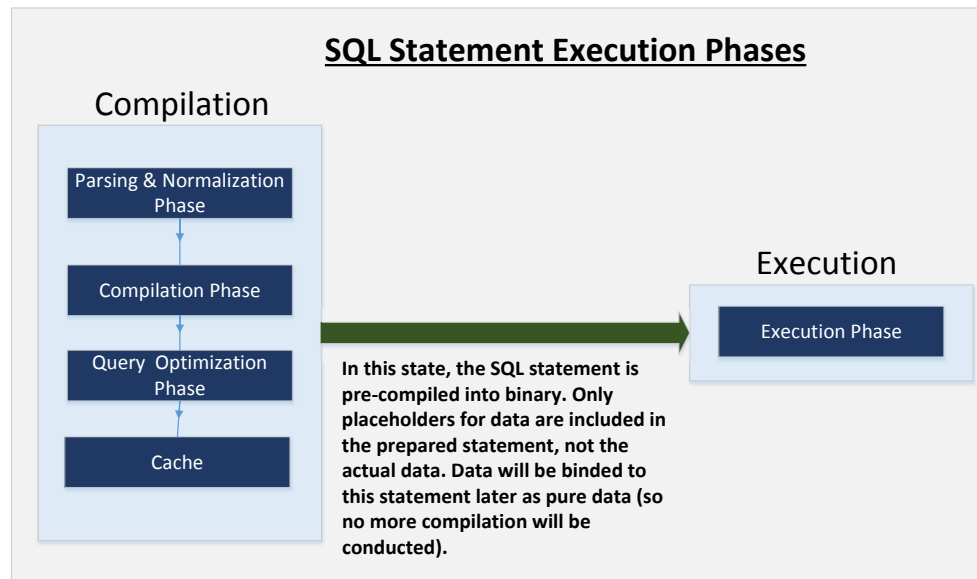


Figure 3: Prepared Statement Workflow

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 statement**.

To understand how prepared statement prevents 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. 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, query is interpreted. In the query optimization phase, the number of different plans are considered to execute the query, out of which the best optimized plan is chosen. The chosen plan is store 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 need to be provided, but these 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 prepared statement prevents SQL injection attacks.

Here is an example of how to write a prepared statement in PHP. We use a SELECT statment in the following example. We show how to use prepared statement to rewrite the code that is vulnerable to SQL injection attacks.

```
$conn = getDB();
$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

```
$conn = getDB();
$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 prepare step. As we can see from the above code snippet, the actual data are 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.

For this task, please use the prepared statement mechanism to fix the SQL injection vulnerabilities exploited by you in the previous tasks. Then, check whether you can still exploit the vulnerability or not.

4 Guidelines

Test SQL Injection String. In real-world applications, it may be hard to check whether your SQL injection attack contains any syntax error, because usually servers do not return this kind of error messages. To conduct your investigation, you can copy the SQL statement from php source code to the MySQL console. Assume you have the following SQL statement, and the injection string is ' or 1=1;#.

```
SELECT * from credential
WHERE name='$name' and password='$pwd';
```

You can replace the value of `$name` with the injection string and test it using the MySQL console. This approach can help you to construct a syntax-error free injection string before launching the real injection attack.

5 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed. Please provide details using screen shots and code snippets. You also need to provide explanation to the observations that are interesting or surprising.

A Patching the Lab Environment

In the environment setup section, we described how to run our patch script to set up the SQL injection lab environment. If you are interested to setting up the environment manually, this section provides step-by-step instructions.

A.1 Load data into database

We need to load some existing data into MySQL. The following commands log into the MySQL database, create a database called `Users`, and load the data from `Users.sql` into the newly created database.

```
$ mysql -u root -pseedubuntu
mysql> CREATE DATABASE Users;
mysql> quit
$ mysql -u root -pseedubuntu Users < Users.sql
```

A.2 Set up the website

Four steps are needed to set up the Employee Management web application inside the VM.

Step 1: Create a new folder `/var/www/SQLInjection` and copy all the php, html,css files to `/var/www/SQLInjection`. Please go to the directory where you store the patch folder and type the following commands.

```
$ sudo mkdir /var/www/SQLInjection
$ sudo cp *.css *.php *.html /var/www/SQLInjection
```

Step 2: In the SEEDUbuntu VM, we use Apache to host all the web sites used in the lab. The name-based virtual hosting feature in Apache can be used to host several web sites (or URLs) on the same machine. A configuration file `/etc/apache2/sites-available/default` contains the necessary directives for the configuration. We can add a new url to `/etc/apache2/sites-available/default` file as follows:

```
<VirtualHost *>
    ServerName http://www.SeedLabSQLInjection.com
    DocumentRoot /var/www/SQLInjection/
</VirtualHost>
```

Step 3: We need to modify the local DNS file `/etc/hosts` to provide an IP address (127.0.0.1) for the host name `www.SeedLabSQLInjection.com`.

```
127.0.0.1          www.SeedLabSQLInjection.com
```

Step 4: Restart the Apache server.

```
$ sudo service apache2 restart
```