SQL Injection Attack Lab

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 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 type of attacks. This lab covers the following topics:

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

Readings.   Detailed coverage of SQL injection can be found in Chapter 11 of the SEED book, *Computer Security: A Hands-on Approach*, by Wenliang Du.

Lab Environment.   This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

2   Lab Environment

We have developed a web application for this lab. The folder where the application is installed and the URL to access this web application are described in the following:

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

The above URL is 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`.

Apache Configuration.   In our pre-built VM image, we 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 `000-default.conf` in the directory `"/etc/apache2/sites-available"` contains the necessary directives for the configuration:

Inside the configuration file, 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. The following examples show how to configure a website with URL `http://www.example1.com` and another website with URL `http://www.example2.com`:

```
<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 `/var/www/Example_1/` directory. After a change is made to the configuration, the Apache server needs to be restarted. See the following command:

```
$ sudo service apache2 start
```

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.

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				

3.1 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. 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. 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.

We will use the login page from `www.SEEDLabSQLInjection.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 authenticate 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.

To help you 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/SQLInjection` directory, is used to conduct user authentication. The following code snippet show 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) {
```
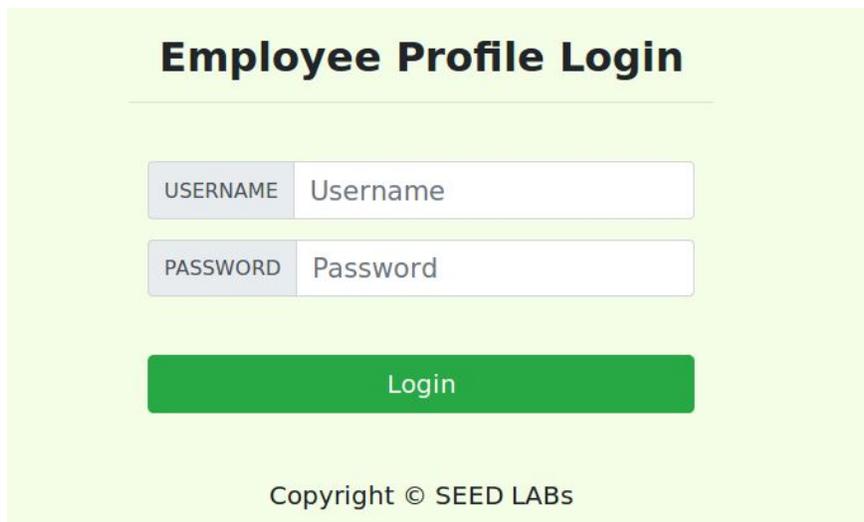
Figure 1: The Login page

```
  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 the password. You need to decide what to type in the `Username` 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 (`username` and `Password`) attached:

```
$ curl
```

```
'www.SeedLabSQLInjection.com/index.php?username=alice&Password=111'
```

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 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 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);
```

- **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 Boby 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. Please demonstrate how you can achieve that. We assume that you do know that salaries are stored in a column called `'salary'`.

- **Task 3.2: Modify other people' salary**. After increasing your own salary, you decide to punish your boss Boby. You want to reduce his salary to 1 dollar. Please demonstrate how you can achieve that.

- **Task 3.3: Modify other people' password**. After changing Boby's salary, you are still disgruntled, so you want to change Boby'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 Boby's account using the new password. One thing worth
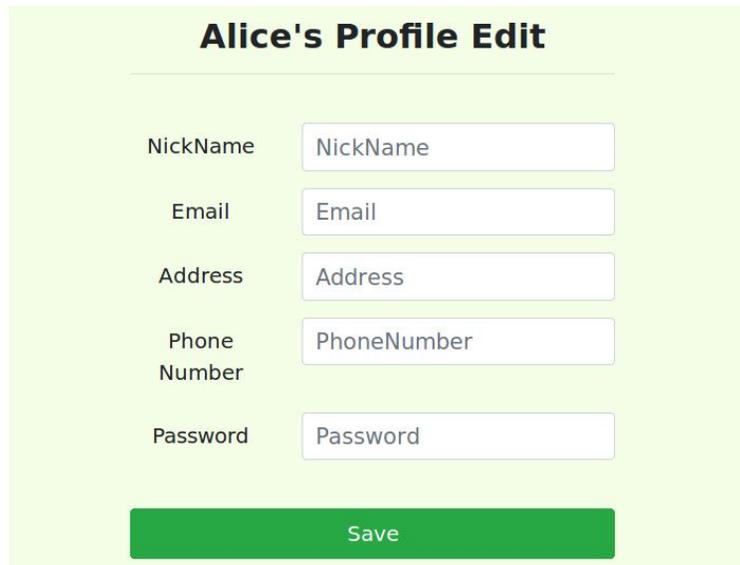
Alice's Profile Edit

NickName   [NickName   ]

Email   [Email   ]

Address   [Address   ]

Phone   [PhoneNumber   ]
Number

Password   [Password   ]

[**Save**]

Figure 2: The Edit-Profile page

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 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 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
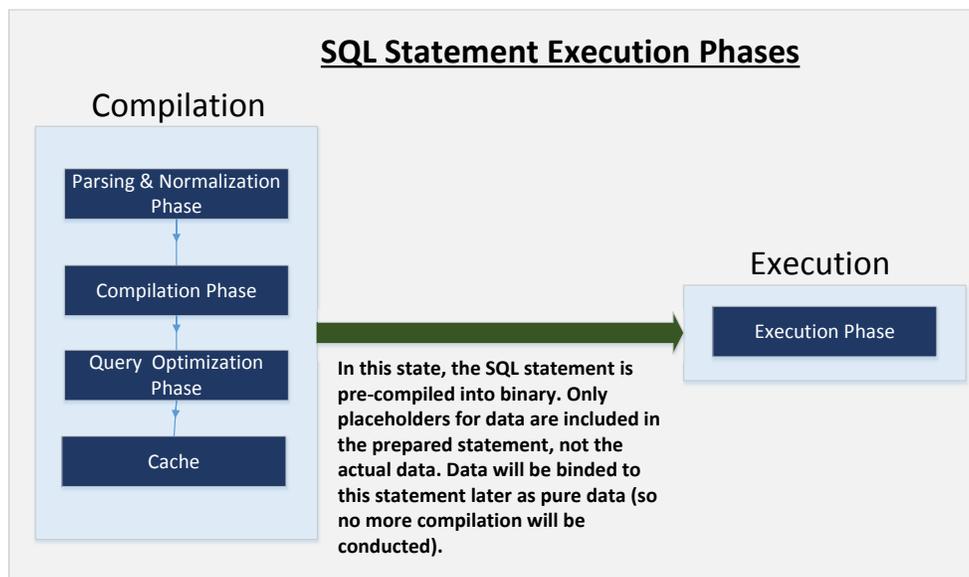
Figure 3: Prepared Statement Workflow

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.

```
$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 the 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.