A Report on Managing SQL Injections using Prepared Statements

192559, Omenda Benir Odeny

Introduction

The aim of this report will be to describe the various practices which I can use to develop a secure web application with proper database functionality, with a focus on enhancing security by means of preventing SQL injections using mysqli as well as prepared statements. To do this, I focus on the various functionality that occurs during management of the database as users interact with the system, mainly CRUD (Create, Read, Update, Delete). To do this, we need a mysqli connection that verifies connectivity between the database and the application, and then I will use code snippets to demonstrate functionality. Once our database is connected, we then make sure all operations are secure as we use CRUD. I use PHP as it is server-side friendly, and we can utilize object-oriented concepts as we secure our operations.

Definition of Terms

SQL Injection - An attack whereby the hacker inserts malicious input that can be interpreted as an SQL statement.

Prepared Statement - An SQL query that is precompiled and is sent to the database before you fill in the actual values.

Transaction – It is a sequence of one or more SQL statements that are executed as a single logical unit of work. Usually, a transaction is only complete if and only if it executes to completion (All or Nothing Property).

It then follows that SQL injections can occur through malicious input, especially via forms that submit data (usually through POST requests). They occur when malicious input is interpreted to be part of a prepared SQL statement and therefore run as an actual SQL query. This is disastrous as it allows unauthorized access to the database.

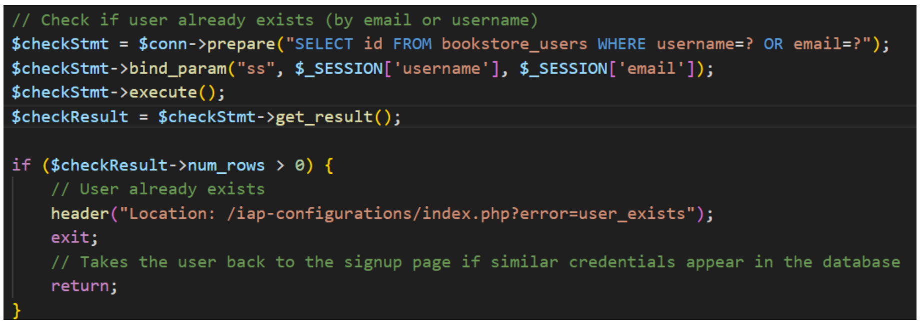
1. Connecting to the Database

To connect to the database, we create a file, preferrable out of the root folder. The file can take the name: database_connect.php and then we write the code that is responsible for establishing connection.

```php
<?php
try {
    $dsn = "mysql:host=localhost;dbname=testdb;charset=utf8mb4";
    $username = "root";
    $password = "strongPas3w0rd";

    // Create PDO instance
    $pdo = new PDO($dsn, $username, $password);

    // Set error mode to exception (important!)
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    echo " Database connection successful!";
} catch (PDOException $e) {
    echo " Connection failed: " . $e->getMessage();
}
```

2. Database Configuration

As my project focuses on a bookstore application, the default table will be my bookstore_users table. Functionality will be handling user input when the user signs in or signs up, therefore we require a POST form that submits to a file with a prepared statement. The image describes the case in which the user signs up. First, we need to check if the user already exists, a SELECT SQL statement comes in handy. In this case, I used mysqli connect, rather than PDO.



```
// Check if user already exists (by email or username)
$checkStmt = $conn->prepare("SELECT id FROM bookstore_users WHERE username=? OR email=?");
$checkStmt->bind_param("ss", $_SESSION['username'], $_SESSION['email']);
$checkStmt->execute();
$checkResult = $checkStmt->get_result();

if ($checkResult->num_rows > 0) {
    // User already exists
    header("Location: /iap-configurations/index.php?error=user_exists");
    exit;
    // Takes the user back to the signup page if similar credentials appear in the database
    return;
}
```

Thwarting SQL Injection Using prepared Statements.

Prepared statements use placeholder values to run SQL queries. This means that whatever the user inputs as data in a form is treated as a value and not SQL code. In the case of malicious input such as: "OR '1'='1', the intention is for this to be interpreted as an SQL query. This makes unauthorized users to gain access to an entire database. Prepared Statements solve this issue by treating the whole user input as a single placeholder value in an SQL query, therefore eliminating the risk of it being executed as raw SQL, making database access more secure.

Bind Variables.

Moreover, prepared statements usually require bind variables to run. The bind variables correspond to the placeholders (?) and their data types. This means the user input is translated into a data type in which a specific operation can be performed. This further eliminates any malicious input as ultimately it will belong to a specific data type. The four data types allowed in bind variables are integer, string, double and blob.

Usually, the first letter of the data type is taken in in the **bind_param()** function. An example is for a select statement in which the placeholder value has two strings, say name and email then the **bind_param** function takes in "ss" as a parameter to the function, followed by the two corresponding variables containing the string user input.

Running Transactions Using PDO

At times, depending on how we have configured our system while users interact with it, there maybe multiple SQL queries that we may need to consider. For example, any time a user buys a book, a money transaction occurs. This means that we need to update the SQL books table containing the catalogue of books, as well as the transactions table, at the same time. If one query is unsuccessful, then we need to abort the whole operation and roll back to the initial state before the transaction.

Example

Using PDO, let us say two users within the same database have participated in a transaction, where a user in the same database sends money to another user within the database. This means that money is deducted from one user but added to another. For this, we need a prepared statement and the two SQL statements. We also use PDO functionality that signals the beginning and end of a transaction in the following way:

```php
<?php
try {
    $pdo = new PDO('mysql:host=localhost;dbname=bank', 'root', 
'password', [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    ]);

    // Start transaction
    $pdo->beginTransaction();

    // Withdraw from account 1
    $stmt = $pdo->prepare("UPDATE accounts SET balance = balance - ? 
WHERE id = ?");
    $stmt->execute([500, 1]);

    // Deposit into account 2
    $stmt = $pdo->prepare("UPDATE accounts SET balance = balance + ? 
WHERE id = ?");
    $stmt->execute([500, 2]);

    // Commit (save) the changes
    $pdo->commit();
    echo "Transaction successful!";
} catch (Exception $e) {
    // Roll back if any error occurs
    $pdo->rollBack();
    echo "Transaction failed: " . $e->getMessage();
}
?>
```

Production Changes

For production, we want to make sure that our database configurations are in an environment file that is not available to the public. Typically, a .env file that is then loaded into php. Also, we store useful information in variables. Instead of our host being localhost, we change it to the actual host variable.

```
# .env
DB_HOST=127.0.0.1
DB_NAME=myapp
DB_USER=myuser
DB_PASS=SuperSecret123
```

We then load it into php using a get_env function, in the following manner:

```php
$db_host = getenv('DB_HOST');
```