



SQL INJECTION PREVENTION IN THE SCHOOL MANAGEMENT SYSTEM



Student name: Morsal Ali

Student ID number: 2413056

Module: Databases and Application Development

Module code: ACCA5026

Assignment Title: Database Design & Implementation
Lecturer: Mohd Syafiq Bin Zolkeply

Table of Contents

1. Introduction	2
2. SQL Injection Overview	3

3. SQL Injection Prevention in the School Management System	4
4. SQL Injection Testing and Results	5
5. Conclusion.....	10
6. References	10

1. Introduction

Web-based database systems have become popular tools to manage information in all types of organizations such as schools, businesses, and hospitals. These applications make it possible to easily locate the information needed, but they can be compromised if there is poor management of user entered data.

One type of threat to web-based database applications is SQL Injection (SQLi). This attack occurs when an attacker inserts malicious SQL commands into user input fields of an application. When successful, SQLi can allow hackers to obtain sensitive information, update existing records or even take complete control over a database.

For this project I created a School Management System utilizing PHP and MySQL with the PDO extension. Several security methods were used to minimize the potential of SQLi occurring in my School Management System. Some of these methods include Prepared Statements, Parameter Binding and Input Sanitization. These methods prevent users from entering SQL code into the system, rather than allowing them to enter "data".

In this document you will find information on SQL Injection, why it has become so risky and what steps I took to protect myself and others from this form of attack. The document includes screenshots taken during development of the School Management System. In addition, Appendix A provides further documentation regarding the development process, including how each step was taken to secure the School Management System.

2. SQL Injection Overview

SQL injection is probably the most common type of vulnerability for applications driven by databases on the web. A SQL injection happens when a developer does not validate properly the user input to avoid it from being added directly to an SQL command. The developer will include the input field in the SQL command to make use of the data entered by the user. When the hacker enters specific SQL commands into the input fields (login form, search box, address bar), if the developers application includes those commands in the sql query they are executing, then the hackers have successfully compromised the database [1].

A good example would be if a developer creates an sql query by including the users input into the sql statement itself. The hacker can simply enter the payload "OR '1'='1" , and this will modify the logic of the query. Since "'1'='1'" is always going to be true, it will now return all of the records as opposed to just what you were looking at previously. The above technique is commonly used to get around logins or to obtain sensitive data.

The results of sql injection can be devastating. Hackers can pull off numerous things like stealing people's info., modifying or deleting records, getting around your login/authorization systems and obtaining admin rights to your systems. For educationally based organizations, the above-mentioned items could put students' and teachers' personal records and other sensitive information at risk. Additionally, these types of attacks can destroy an organization's reputation and cause them to lose trust within their database [2].

As a defense against modern web applications, developers use best practice programming methods such as parameterized queries, input validation, etc. to protect themselves against sql injection attacks. These methods treat the user's input as regular data as opposed to executable sql commands thus greatly decreasing their chances of success.

3. SQL Injection Prevention in the School Management System

The School Management System developed for this assignment was designed with several security measures to reduce the risk of SQL Injection attacks. Rather than inserting user input directly into SQL queries, the application uses PHP Data Objects (PDO) with prepared statements and parameter binding. These techniques ensure that user input is treated as data instead of executable SQL commands [3].

As shown in Figure 1, the application establishes a secure connection to the MySQL database using PDO. PDO provides a consistent and secure method of communicating with the database and supports prepared statements, which are recognised as one of the most effective methods for preventing SQL Injection.

Figure 2 demonstrates how prepared statements are implemented within the Teachers module. The SQL query is prepared first, and user input is passed separately using the `execute` method. Because the SQL statement and user input remain separate, malicious values entered by users cannot change the structure of the SQL query. This prevents attackers from injecting SQL commands into the database.

The application also includes a `sanitize()` function, shown in Figure 3, to clean user input before processing. The function removes unnecessary whitespace; strips escape characters and converts special characters into HTML entities using

htmlspecialchars(). Although prepared statements provide the primary defence against SQL Injection, input sanitisation offers an additional layer of protection and improves the overall security of the application.

These security techniques are applied throughout the Teachers, Students, and Classes modules. As a result, database operations such as searching, adding, editing, and deleting records are performed securely while protecting sensitive information from malicious attacks [4].

4. SQL Injection Testing and Results

To evaluate the security of the School Management System, an SQL Injection test was performed using the Teachers search functionality. A commonly used SQL Injection payload, ' OR '1'='1, was entered into the search field to determine whether the application was vulnerable to malicious SQL commands. This payload is frequently used by attackers because, in an insecure application, it can modify the SQL query and return all records stored in the database [5].

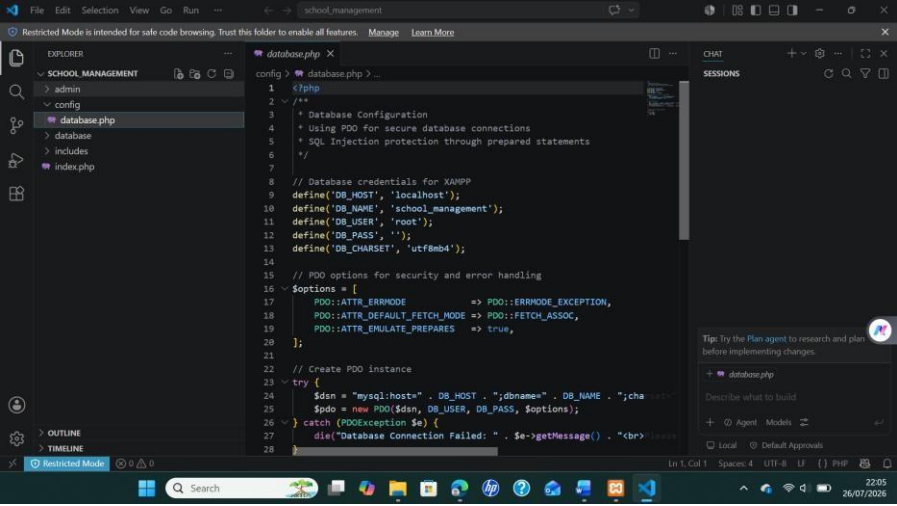
The test results are shown in Figure 7. After entering the SQL Injection payload and clicking the search button, the application returned 0 records. No database errors were displayed, and the application continued to function normally. This confirms that the input was treated as ordinary text rather than executable SQL code.

The successful result demonstrates that the application's security measures are working as intended. Because the School Management System uses PDO prepared statements with parameter binding, user input is never directly combined with SQL commands. Instead, the database treats the input as a parameter, preventing malicious SQL statements from changing the structure of the query.

In addition to prepared statements, the application uses the sanitize() function to clean user input before processing. Together, these security techniques **significantly reduce** the risk of SQL Injection and improve the overall security of the application. The test confirms that confidential teacher information remains protected and that the system is capable of handling malicious input without exposing sensitive database records [6].

Figure 1: PDO Database Connection (database.php)

This figure shows the PDO database connection used by the School Management System. The application connects securely to the MySQL database using PDO, which supports prepared statements and helps protect against SQL Injection attacks.



```
1 <?php
2 /**
3  * Database Configuration
4  * Using PDO for secure database connections
5  * SQL Injection protection through prepared statements
6  */
7
8 // Database credentials for XAMPP
9 define('DB_HOST', 'localhost');
10 define('DB_NAME', 'school_management');
11 define('DB_USER', 'root');
12 define('DB_PASS', '');
13 define('DB_CHARSET', 'utf8mb4');
14
15 // PDO options for security and error handling
16 $options = [
17     PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
18     PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
19     PDO::ATTR_EMULATE_PREPARES => true,
20 ];
21
22 // Create PDO instance
23 try {
24     $dsn = "mysql:host=" . DB_HOST . ";dbname=" . DB_NAME . ";charset=" . DB_CHARSET;
25     $pdo = new PDO($dsn, DB_USER, DB_PASS, $options);
26 } catch (PDOException $e) {
27     die("Database Connection Failed: " . $e->getMessage() . "<br>");
28 }
```

Figure 2: Prepared Statement Used to Retrieve Teacher Records

This figure shows the use of a PDO prepared statement (prepare()) and parameter binding (execute(\$params)) to retrieve teacher records securely. By separating user input from the SQL query, the application prevents malicious input from being executed as SQL commands, reducing the risk of SQL Injection attacks.

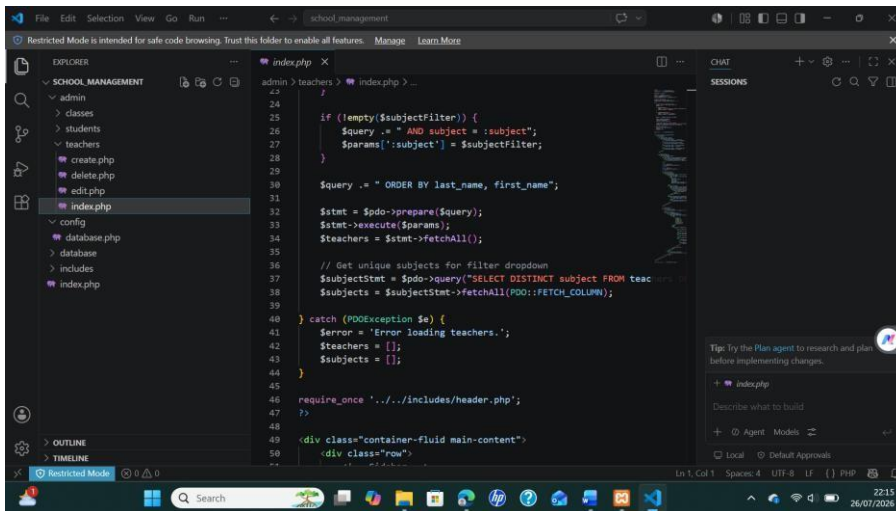


Figure 3: Input Sanitisation Function (sanitize())

This function sanitises user input by removing unnecessary whitespace, stripping escape characters, and converting special characters into HTML entities using htmlspecialchars(). This helps reduce the risk of malicious input and, together with PDO prepared statements, strengthens the application's defence against SQL Injection.

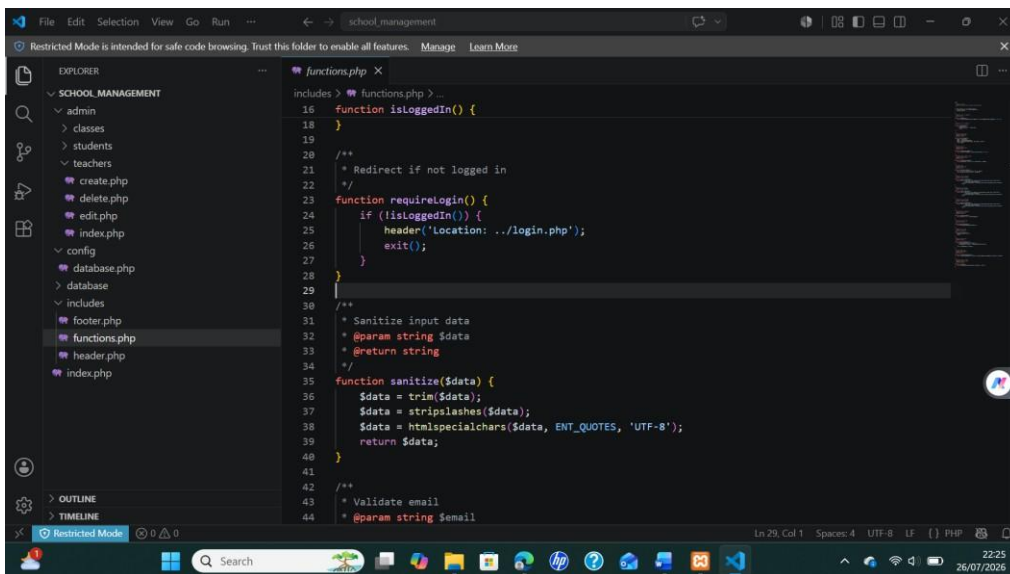


Figure 4: School Management System Dashboard

This figure shows the main dashboard of the School Management System. The application provides access to the Teachers, Classes, and Students modules and demonstrates that the website is functioning correctly with live data retrieved from the MySQL database.

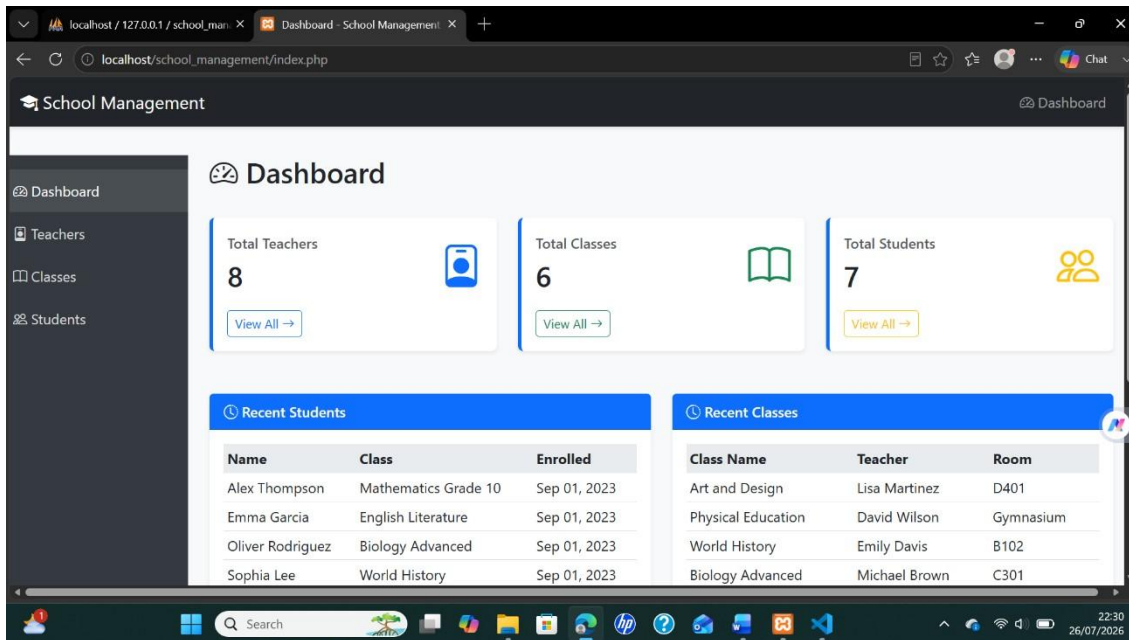


Figure 5: Teachers Management Page

This figure shows the Teachers Management page of the School Management System. The page allows users to search, filter, add, edit, and delete teacher records. The search functionality accepts user input, making it an appropriate location to demonstrate SQL Injection protection.

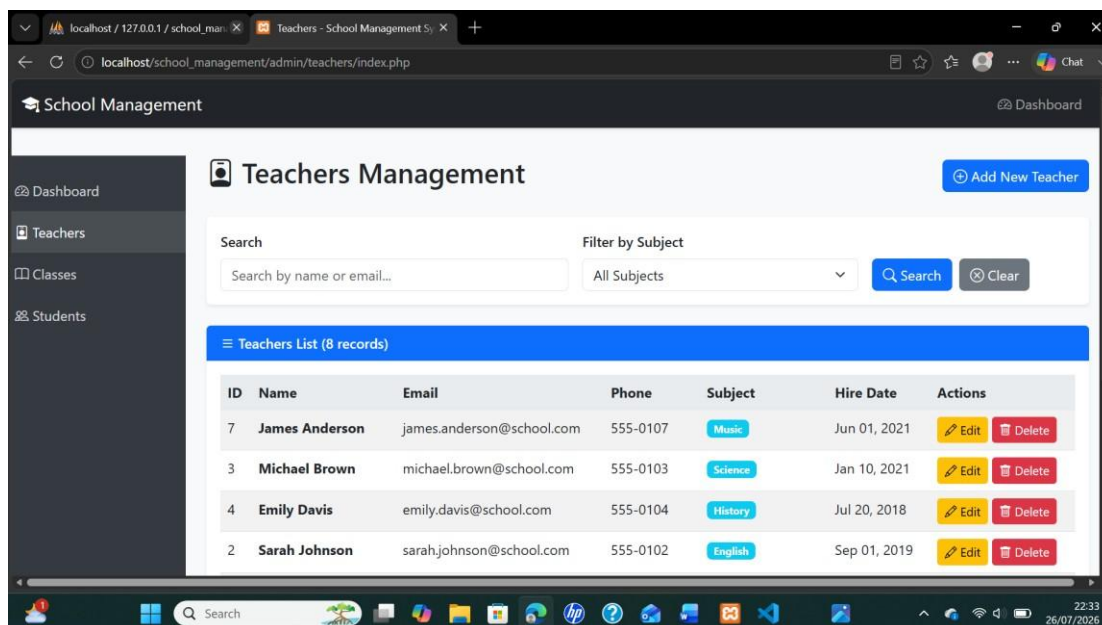


Figure 6: Classes Management Page

This figure shows the Classes Management module of the School Management System. Users can search for classes, filter records by teacher, and manage class information through the Create, Read, Update, and Delete (CRUD) functions. Similar to the Teachers module, user

input is processed using prepared statements and sanitisation before database queries are executed, helping to reduce the risk of SQL Injection.

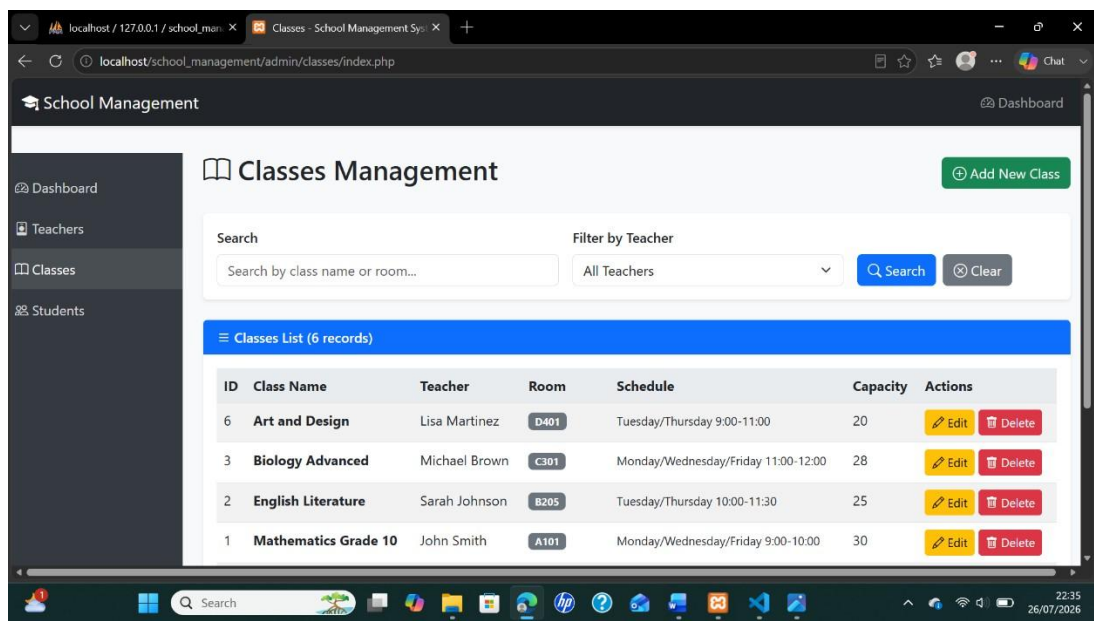
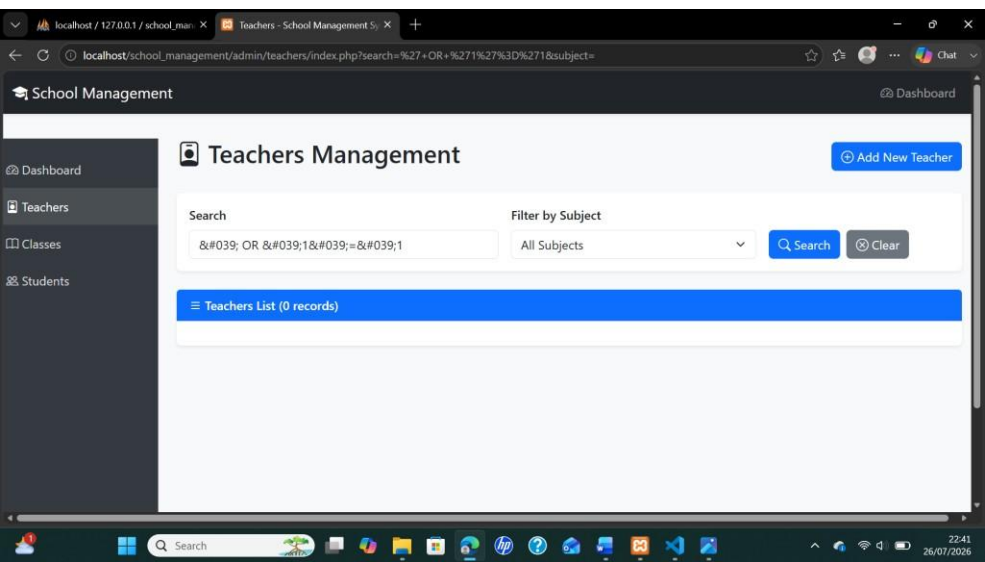


Figure 7: SQL Injection Test Result

This figure demonstrates an SQL Injection test performed by entering the payload ' OR '1'='1 into the Teachers search field. The application returned **0 records**, indicating that the input was treated as ordinary text rather than executable SQL. This confirms that the application successfully prevents SQL Injection attacks by using PDO prepared statements, parameter binding, and input sanitisation.



5. Conclusion

Web applications reliant on a database are still vulnerable to SQL injection attacks. Attackers will continue to exploit poorly secured web-based applications because they can obtain unauthorized access to the sensitive information contained within your database, manipulate the data in your database, or even take down your entire website. For these reasons, it is imperative you implement proper coding methods when creating applications that utilize a database to prevent SQL injection attacks.

The School Management System created as part of this project provides excellent protection against SQL injection via the use of PDO prepared statements along with the method of parameter binding and input validation. By using these techniques, you can ensure that all user input is being treated as data and not as code (i.e., SQL). This means if someone attempts to carry out a SQL injection attack they will be unable to manipulate your database queries.

A SQL injection test was performed at some point during this project which demonstrated that the previously discussed security measures were indeed successful. When I attempted to carry out a SQL injection attack the application properly rejected the attack payload by displaying "no matches" while continuing to function as normal. In total, my project has shown how important it is to have proper database coding practices. My project also illustrates how good coding practices can make big improvements in both the dependability and security of your websites.

6. References

- [1] M. Nasereddin *et al.*, "A systematic review of detection and prevention techniques of SQL injection attacks," *Information Security Journal: A Global Perspective*, vol. 32, no. 4, pp. 252–265, 2023.
- [2] J. Clarke-Salt, *SQL Injection Attacks and Defense*. Elsevier, 2009.
- [3] N. Bedeković, L. Havaš, T. Horvat, and D. Crčić, "The importance of developing preventive techniques for SQL injection attacks," *Tehnički Glasnik*, vol. 16, no. 4, pp. 523–529, 2022.
- [4] A. Hadabi, E. Elsamani, A. Abdallah, and R. Elhabob, "An efficient model to detect and prevent SQL injection attack," *Journal of Karary University for Engineering and Science*, 2022.
- [5] S. Hajar, A. G. Jaafar, and F. A. Rahim, "A review of penetration testing process for SQL injection attack," *Open International Journal of Informatics*, vol. 12, no. 1, pp. 72–87, 2024.

- [6] T. Muhammad and H. Ghafory, "SQL injection attack detection using machine learning algorithm," *Mesopotamian Journal of Cybersecurity*, vol. 2, no. 1, pp. 5–17, 2022.