SQL Injection and XSS: Mitigation in a Deliberately Vulnerable PHP/MySQL Web Application

Abdulla Alsaadi
Email: aalsaadi@purdue.edu

***Abstract*—SQL injection (SQLi) and cross-site scripting (XSS) remain a droment and lurking type threats that looms on the head of any website [1]. Recent research has shown that while the number of SQLi attacks have decreased, they are still prevalent especially in quickly assembled private scripts and websites that are not intended for public used [2], [3]. This paper describes the design choices that lead to the creation of a deliberately vulnerable website The demo includes a login page that query the `users` table and a products search page that query the `products` table. To exploit the vulnerable website, the attacker would need to exploit concatenated SQL queries to (1) bypass the login authentication logic, (2) create a new user, (3) inject a XSS payload that execute once the user accidentally search for it, (4) inject a drop command that drop the products and the users table. Finally, the underlying vulnerabilities are outline and hardened using preperaed statements, output encoding, and database-level hardening. The goal of this project is not to propose a new deference but to provide concrete introductory example on how small design choices could lead to real SQLi and XSS attacks.**

***Index Terms*—SQL injection, cross-site scripting, web security, PHP, MariaDB, XAMPP, prepared statements.**

I. Introduction

SQL injection (SQLi) is still one of the most serious risks for web applications. SQLi is a web vulnerability that allows the attacker to execute SQL commands using unprepared and vulnerable input fields. Alternatively, the attacker could use the URL pattern to inject the SQL command. However, this is outside the scope of this paper. There are many tools out there that attempt to test SQLi attacks and automate the whole process [4].

SQL injection remain high on OWASP's top 10 [5] and are often used as the first step to steal, alter, or destroy data in production environment. In practice, many vulnerable applications are simple custom PHP scripts that directly concatenate user input into SQL queries and render database content back into HTML without proper encoding. Many of these scripts are used as backend tools intended for developer use only and they are not hardened enough since the expectation of outside low is non-existent. This hands-on labs help beginners understand why this exploit is dangerous, what causes it, and how to remedy it.

In this project, we built a small, self-contained, deliberately vulnerable demo site that runs entirely on a local XAMPP stack. The site is realistic enough to show real damage and demonstrate the severity of this type of exploits. including creating new administrator accounts, injecting JavaScript into search results, and dropping tables—but small enough that anyone can read and understand every line of code. The website was created using Claude AI [6]. The AI model generated a safe website then further modified was required to the make it vulnerable to SQLi and XSS.

This paper documents the design of that demo application, walks through the attacks step-by-step, and explains the code changes that fix the vulnerabilities. The focus is educational: make the root cause of SQLi and XSS obvious, then show the minimal but correct fixes using prepared statements, output encoding, and database hardening.

The full source code is available at
https://github.com/22AKMS/SQLi-project.

II. Literature Summary of the Topic

The security literature body has analyzed SQL injection for decades. Originally, SQLi and XSS were more prevelant in 90s and well into the 2010s [7], since frameworks and website building tools like Angular, Next.js, and Svelte were only created recently in 2016 [8], [9], [10]. These tools have streamlined website creation and would handle the security of the input fields for the developer. This does not mean that they are foolproof, but they are an improvement to the old way of building a website from scratch.

Work by Halfond et al. divides SQLi into different types, including tautology-based attacks, union queries, piggy-backed queries, and inference attacks [11]. However, This research paper will only focus on simple SQLi methods that does not require advanced scripting and deep website creation insight and knowhow.

A large portion of research focuses on automatic detection and prevention. Some approaches focus on the application, others use proxy firewalls that recognize injection requests, and others analyze query structure to detect illegal changes. However, this type of research is outside the scope of this project [12], [13], [14], [15].

The OWASP's Top 10 Guidelines, includes SQLi, as a top risk for many web applications [5]. Across sources, the recommended defenses are: use parameterized queries or what we call prepared statements in this paper, avoid passing SQL strings from raw input, enforce least privilege for database

accounts by creating views and changing the default user, and validate input where it makes sense [16], [17].

III. Specific Methodologies

The project methodology is devided into two main phases: *attack* and *defense*. Both use the same PHP/MySQL application but with different configuration and code.

A. Environment Setup

The environment is built on a local XAMPP installation that bundles:

- Apache for serving the PHP pages,
- MariaDB/MySQL for the database,
- PHP as the server-side scripting language,
- phpMyAdmin for inspecting and editing database tables.

The database `demo_site` contains at least two tables:

- **Users**
  which stores `id`, `username`, `password`, `full_name`, `role`, and `timestamps`.

- **Products**
  which stores `id`, `name`, `category`, `price`, `description`, `stock`, and `timestamps`.

Sample data is inserted via SQL scripts so that attackers start with a working login (for example, an `admin` account) and some realistic product entries. The sample SQL file will be included in the project's GitHub repository, as well as a detailed, step by step, guide to installing the project [18].

B. Artificial Intelligence use

Claude AI was used to generate a safe website that was later altered to become vulnerable [6]. The AI model was prompted to create a PHP website with a login.php and search.php files. Then the model was instructed to communicate with MariaDB and query for the required data. The model returned a fully functional website that is secure to SQLi and XSS. For the sake of this project, we have rigged the query and XSS checks to simulate a minimally guarded setup.

C. Attack Phase

In the attack phase, the application uses intentionally insecure PHP code:

- SQL queries are built with string concatenation of user input.
- The database connection uses the default user which is `root`.

attackers carry out a series of attack scenarios:

1) Login bypass using a boolean statement.
2) Account creation using stacked SQL statements to insert new rows into the `users` table.
3) Stored XSS statements by inserting JavaScript into the `products` table and triggering it via search.

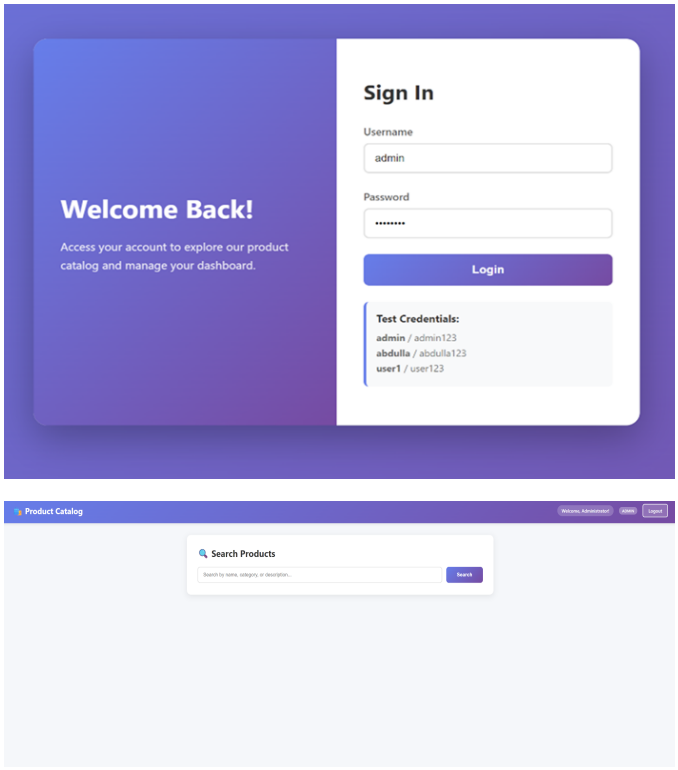

Fig. 1. Generated login page (top) and generated search page (bottom).

4) Destructive SQLi by injecting `DROP TABLE` statements.

Each scenario is verified visually in the browser and in phpMyAdmin. The idea is to make the impact obvious: the attacker can log in as admin, see new rows, see scripts execute, and watch tables disappear.

D. Defense Phase

In the defense phase, the same site is modified with standard, well-known mitigations informed by our literature review:

- Prepared statements with bound parameters replaces passing raw SQL commands.
- `htmlspecialchars()` is used before printing any data into HTML to prevent XSS from running.
- The database user is changed from `root` to a dedicated account with limited privileges.

The same attack inputs from the first phase are then re-used. attackers observe that login bypass attempts now fail, stacked statements do not execute, XSS payloads render as plain text, and destructive `DROP TABLE` commands are blocked by lack of privilege.

IV. Implementations

This section describes the key implementation details in the PHP and SQL files.

A. Database Configuration and SQL Files

The SQL setup file creates the `demo_site` database, defines the `users` and `products` tables, and inserts initial rows. A simple example for the `users` table is:

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(50) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL,
  full_name VARCHAR(100),
  role VARCHAR(20),
  created_at TIMESTAMP DEFAULT
  CURRENT_TIMESTAMP
);
```

The script also inserts users (for example, `admin` with a demo password) and several normal users. The `products` table is defined with realistic fields and a few rows to display in search results.

```
INSERT INTO users (username, password, full_name,
    role) VALUES
('admin', 'admin123', 'Administrator', 'admin'),
('abdulla', 'abdulla123', 'Abdulla Alsaadi', '
    manager'),
('user1', 'user123', 'John Smith', 'user');

INSERT INTO products (name, category, price,
    description, stock) VALUES
('Wireless Mouse', 'Electronics', 29.99, 'Ergonomic
    wireless mouse with USB receiver', 50),
('Mechanical Keyboard', 'Electronics', 89.99, 'RGB
    backlit mechanical gaming keyboard', 30),...;
```

B. Vulnerable PHP: Login and Search

The PHP files share a simple configuration file for database access:

```
// config.php
define('DB_HOST', 'localhost');
define('DB_USER', 'root');
define('DB_PASS', '');
define('DB_NAME', 'demo_site');
```

A helper file `db.php` returns a `mysqli` connection using these constants.

In the vulnerable version of `login.php`, user input is read and used directly in the SQL string:

```
$username = $_POST['username'];
$password = $_POST['password'];

$conn = getConnection();

$sql = "SELECT * FROM users
        WHERE username = '$username'
          AND password = '$password'";

$result = $conn->query($sql);
```

Because the data is dropped straight into the query, any quotes or semicolons in `$username` or `$password` end up in the final SQL.

The vulnerable search function works in a similar way:

```

$sql = "SELECT * FROM products
        WHERE name LIKE '%$term%'
           OR description LIKE '%$term%'";

$result = $conn->query($sql);
```

The results are then rendered directly into HTML:

```
<div class="product-name">
  <?php echo $row['name']; ?>
</div>
<p class="product-description">
  <?php echo $row['description']; ?>
</p>
```

Here, any HTML or JavaScript stored in `name` or `description` will run in the browser.

C. Example Attack Payloads

With this implementation, classic payloads work exactly as expected:

- **Login bypass:**

```
Username: ' OR '1'='1
Password: anything
```

- **Create new user with stacked query:**

```
cit'; INSERT INTO users VALUES (
  NULL, 'attacker', 'cit',
  'Attacker User', 'admin', NOW()
);
```

- **Stored XSS in products:**

```
cit'; INSERT INTO products VALUES (
  NULL,
  '<script>alert("XSS")</script>',
  'Evil', 1,
  'Malicious', 1, NOW()
);
```

- **Drop products/users :**

```

cit'; DROP TABLE products;
```

```

cit'; DROP TABLE users;
```

All of these are sent through the normal web forms and stored or executed by the application.

D. Hardened PHP: Prepared Statements

In the fixed version of `login.php`, the SQL statement is constant and user data is bound as parameters:

```
$conn = getConnection();

$stmt = $conn->prepare(
  "SELECT * FROM users
   WHERE username = ? AND password = ?"
);
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
$result = $stmt->get_result();
```

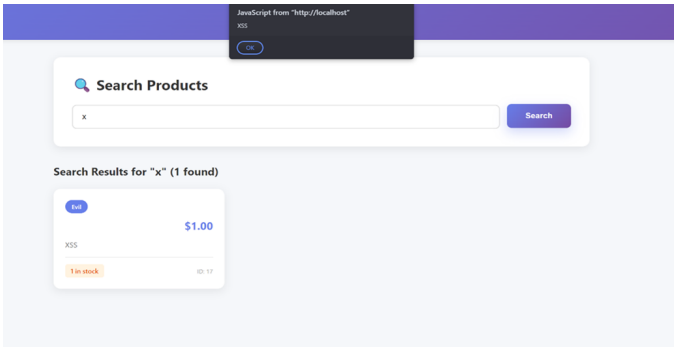

Fig. 2. Stored XSS payload executing when the injected product is rendered in the browser.

The database now knows exactly where the parameters go and treats them only as data. If the attacker submits `' OR '1'='1`, it is matched as a literal username, not as part of the SQL structure. Stacked statements separated by semicolons are not interpreted as additional queries.

E. Hardened PHP: Output Encoding

To stop XSS, the search results are encoded before output:

```
<div class="product-name">
  <?php echo htmlspecialchars(
      $row['name']); ?>
</div>
<p class="product-description">
  <?php echo htmlspecialchars(
      $row['description']); ?>
</p>
```

`htmlspecialchars()` converts special characters like < and > into safe entities. Injected scripts now show up as text in the page instead of running inside the browser.

F. Database Least Privilege

Finally, the database user account is changed. Instead of using `root`, a dedicated account such as `demo_app` is created with only the needed privileges on the `demo_site` database. This account does not have global `DROP` or `ALTER` permissions. Even if new SQLi bugs appear later, the damage is limited.

V. Conclusion

This project has demonstrated SQL injection and XSS vulnerabilities using a small but readable PHP web application. In the first phase of the project, the attacker uses a simple SQL statement which allows full login bypass, unauthorized account creation, stored XSS, and destructive queries like dropping tables. In the second phase, we harden the same exploitable application by switching to prepared statements, output encoding, and a least-privilege database user.

References

[1] I. Hydara *et al.*, "Current state of research on cross-site scripting (xss)," *Information and Software Technology*, 2015, systematic literature review on XSS research and prevalence.

[2] Aikido Security. (2024) The state of sql injections. Gives statistics on the share of SQLi among discovered vulnerabilities in 2024. [Online]. Available: https://www.aikido.dev/blog/the-state-of-sql-injections

[3] P. Momeni and J. Ligatti, "Measuring the absolute and relative prevalence of sql concatenations and sql injection vulnerabilities," *Journal of Computer Security*, 2025, empirical study of SQL concatenations and SQLi patterns in real code and CVEs.

[4] sqlmap Project. (2025) sqlmap: Automatic sql injection and database takeover tool. Official sqlmap project site describing automated SQL injection testing. [Online]. Available: https://sqlmap.org/

[5] OWASP Foundation. (2021) Owasp top 10:2021. Standard awareness document of critical web application security risks. [Online]. Available: https://owasp.org/Top10/2021/

[6] Anthropic. (2025) Intro to claude. Official documentation describing Claude as an AI platform for language, coding, and analysis tasks. [Online]. Available: https://platform.claude.com/docs/en/intro

[7] A. S. Buyukkayhan *et al.*, "What's in an exploit? an empirical analysis of reflected server xss exploitation techniques," in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[8] InfoQ. (2016) Angular 2 final released, adopts semantic versioning. Announces the final release of Angular 2 in September 2016. [Online]. Available: https://www.infoq.com/news/2016/09/angular-2-final-released/

[9] Auth0. (2017) Next.js 3.0 release: What's new? States that Next.js was first released on October 25, 2016. [Online]. Available: https://auth0.com/blog/nextjs-3-release-what-is-new/

[10] Devoteam. (2020) Svelte: a new approach to building user interfaces. Explains that the first version of Svelte was released in November 2016. [Online]. Available: https://www.devoteam.com/expert-view/svelte-framework-user-interfaces/

[11] W. G. J. Halfond, J. Viegas, and A. Orso, "A classification of SQL injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006, also available as a technical report from Georgia Institute of Technology. [Online]. Available: https://viterbi-web.usc.edu/~halfond/papers/halfond06issse.pdf

[12] M. Nasereddin, A. ALKhamaiseh, M. Qasaimeh, and R. Al-Qassas, "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, 2021, systematic review of SQL injection detection and mitigation techniques. [Online]. Available: https://doi.org/10.1080/19393555.2021.1995537

[13] P. Kumar and R. K. Pateriya, "A survey on SQL injection attacks, detection and prevention techniques," in *2012 Third International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2012, pp. 1–5, survey of SQL injection attacks and various detection / prevention approaches. [Online]. Available: https://doi.org/10.1109/ICCCNT.2012.6396096

[14] S. F. Hidhaya and A. Geetha, "Intrusion protection against SQL injection and cross site scripting attacks using a reverse proxy," in *Recent Trends in Computer Networks and Distributed Systems Security*, ser. Communications in Computer and Information Science. Springer, 2012, vol. 335, pp. 252–263, reverse-proxy based protection behaving like a web application firewall. [Online]. Available: https://doi.org/10.1007/978-3-642-34135-9_26

[15] M.-Y. Kim and D. H. Lee, "Data-mining based SQL injection attack detection using internal query trees," *Expert Systems with Applications*, vol. 41, no. 11, pp. 5416–5430, 2014, analyzes query structure to detect SQL injection. [Online]. Available: https://doi.org/10.1016/j.eswa.2014.02.041

[16] OWASP Foundation. (2021) A03:2021 – injection. Details injection vulnerabilities including SQL injection. [Online]. Available: https://owasp.org/Top10/A03_2021-Injection/

[17] ——. (2023) Sql injection prevention cheat sheet. Recommends prepared statements, avoiding dynamic SQL from user input, and enforcing least privilege on database accounts. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[18] A. Alsaadi. (2025) Sqli-project: Deliberately vulnerable php/mysql web application. Accessed: 7 Dec. 2025. [Online]. Available: https://github.com/22AKMS/SQLi-project