

# Mitigating Vulnerabilities in an Educational Website

# 12th Project – Milestone 1

Security in Software Engineering

Dr. Nada Hany Sherief Eng: Salma Elkady

***Team Members :***

Ahmed Elbialy\_211008712 Habiba Ahmed\_211003103

Ahmed Yasser\_211010273 Fares Fawaz\_19101617

# 1. Threat Modeling Process

## 1.1 Identify Assets

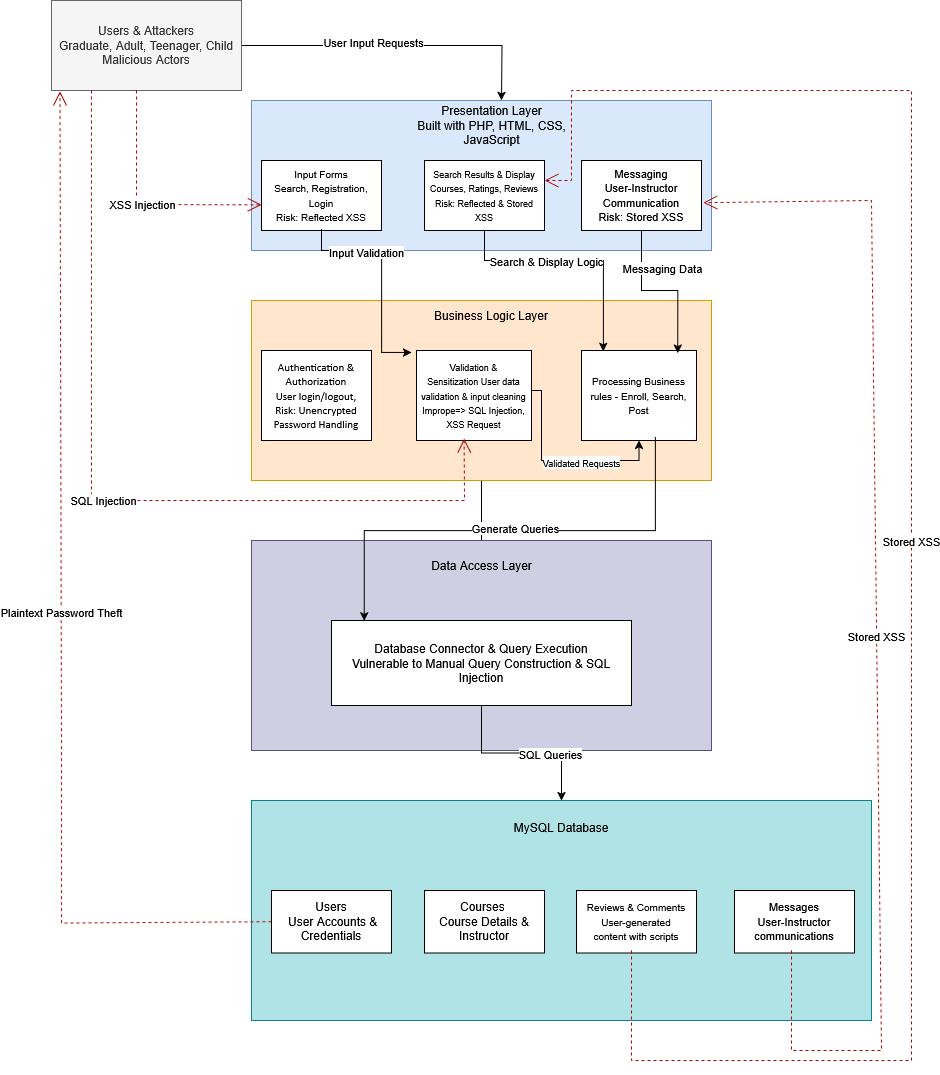
Tangible Assets:  
- Web Server  
- Database Server  
- Source Code Repository  
- Web Application

Intangible Assets:  
- User credentials  
- Course and instructor data  
- Communication data  
- Website reputation  
- Session tokens

## 1.2 Architecture Overview

1.2.1 Application Function:  
The website enables users of all ages to search for educational topics and enroll in suitable courses. Users can view ratings, communicate with instructors, and leave reviews.

*1.2.2 Architecture Diagram:*

**

**Component and Vulnerability Descriptions**

* **Users & Attackers:** Legitimate learners (graduates, adults, teenagers, children) and malicious actors exploiting vulnerabilities.
* **Presentation Layer:**
  + *Input Forms:* Course search, registration, login inputs directly accept user data, vulnerable to reflected XSS if input is not sanitized.
  + *Search Results & Display:* Shows dynamic content like courses, ratings, and reviews; vulnerable to reflected and stored XSS attacks by malicious scripts in displayed content.
  + *Messaging:* User-instructor communication may contain unsanitized user input, posing stored XSS risks.
* **Business Logic Layer:**
  + *Authentication & Authorization:* Handles login session and password management, currently vulnerable if passwords are handled without encryption.
  + *Validation & Sanitization:* Intended to clean and verify user inputs but if incomplete or missing parameterized queries, enables SQL injection and XSS attacks.
  + *Request Processing:* Implements business rules for enrollment, searching, posting content.
* **Data Access Layer:**
  + Database connector executes queries. Vulnerable if dynamically constructed without parameterized queries, enabling SQL injection attacks.
* **MySQL Database:**
  + *Users Table:* Contains user credentials, with critical risk storing passwords in plaintext.
  + *Courses Table:* Contains course and instructor info.
  + *Reviews & Comments:* User-generated content, potential storage vector for malicious scripts (stored XSS).
  + *Messages:* Messaging system data, another stored XSS risk surface.
* **Attack Vectors:** XSS attacks occur through input forms and stored content. SQL Injection threatens data integrity and access. Plaintext passwords risk user account takeover if database breaches occur.

1.2.3 Technologies:

- Front-End: PHP, HTML, CSS, JavaScript

- Back-End: PHP

- Database: MySQL

## 1.3 Decompose the Application

1.3.1 Trust Boundaries:  
- Browser ↔ Web Server  
- Web Server ↔ Database

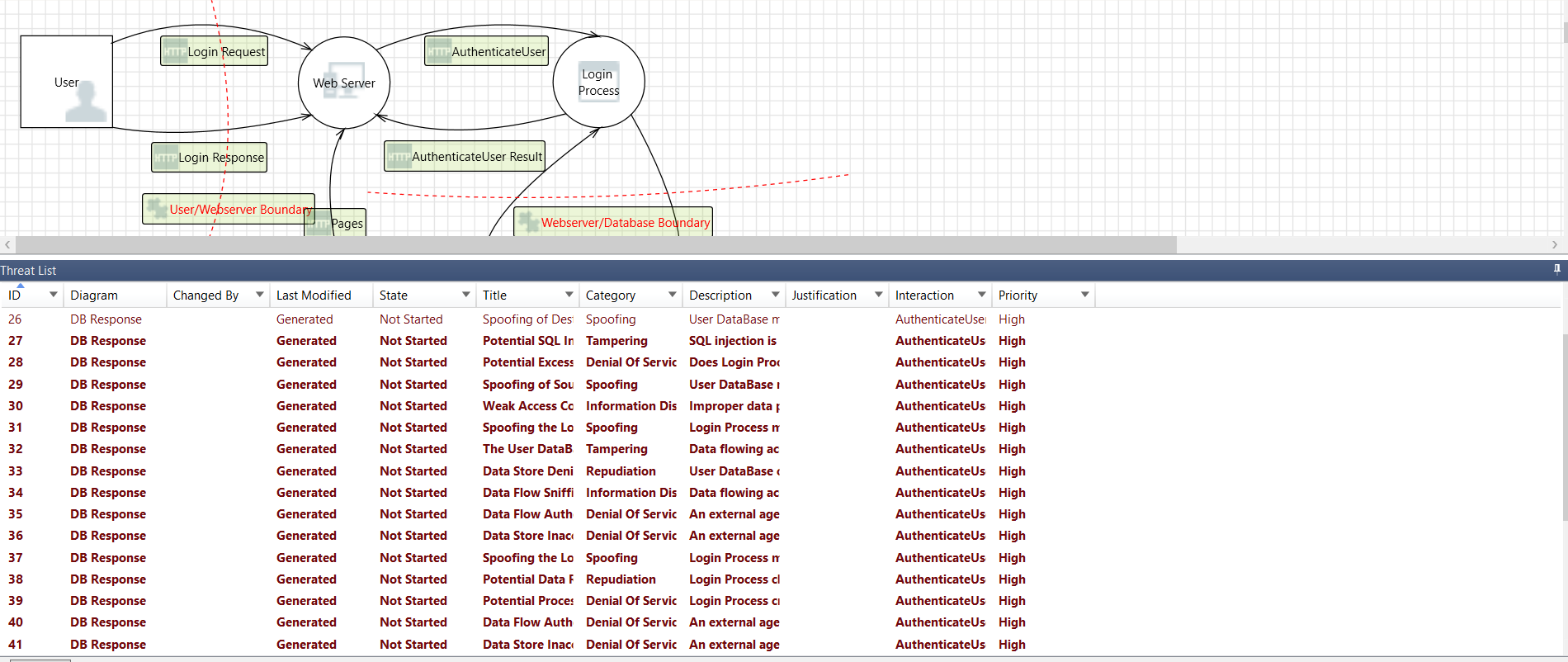
A diagram of a diagram

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A diagram of a company

AI-generated content may be incorrect.A diagram of a computer

AI-generated content may be incorrect.



A diagram of a diagram

AI-generated content may be incorrect.

1.3.2 Data Flow:  
- User input → Web forms → Server-side PHP → SQL queries → Database

1.3.3 Entry Points:  
- Login/Register Forms  
- Search Bar  
- Comment and Review Sections

1.3.4 Privileged Code:  
- Admin panel functions  
- Authentication and Authorization modules

1.3.5 Security Profile:  
[To be completed with a table]

| Vulnerability | Threat Description | Risk Level | Impact | Mitigation Strategy | Estimated Mitigation Cost |
| --- | --- | --- | --- | --- | --- |
| Reflected Cross-Site Scripting (XSS) | Malicious script injected via URLs or form inputs is reflected and executed in the victim's browser. | High | - Session hijacking- Credential theft- Browser exploitation | - Sanitize all user inputs using built-in PHP functions like htmlspecialchars() or filter\_input().- Use Content Security Policy (CSP). | $200,000 |
| Stored Cross-Site Scripting (XSS) | Malicious scripts stored in course reviews or discussion threads, executed when users view the content. | Critical | - Persistent compromise of user sessions- Spread of malicious scripts- Website defacement | - Sanitize user inputs before database storage.- Encode output before rendering.- Implement a robust WAF (Web Application Firewall). | $300,000 |
| SQL Injection | Unsanitized user inputs in SQL queries allow attackers to manipulate database queries. | Critical | - Data leakage (users, courses, credentials)- Data loss or corruption- Unauthorized admin access | - Use prepared statements with bound parameters (PDO or mysqli with prepared statements).- Conduct code audits and automated scanning.- Limit database permissions. | $500,000 |
| Unhashed Passwords | Plaintext passwords stored in database can be accessed if compromised. | Critical | - Credential reuse attacks- Total compromise of user accounts- Loss of user trust | - Hash passwords using secure algorithms like bcrypt or Argon2.- Apply salting and key stretching.- Perform security-focused database encryption. | $250,000 |
| Unencrypted Data Transmission | Passwords and data transmitted in plaintext over the network can be intercepted. | High | - Man-in-the-middle (MITM) attacks- Credential theft | - Enforce HTTPS across all pages using TLS 1.2+.- Use HSTS (HTTP Strict Transport Security). | $100,000 |
| Inadequate Input Validation | Input fields not validated properly, increasing the risk of injection and logic flaws. | Medium | - Increased surface for XSS, SQLi- Application errors/crashes | - Implement strict server-side validation for all input fields.- Apply allowlists for expected formats (e.g., email, text). | $150,000 |

Summary of Security Profile

Total Estimated Mitigation Cost: ~$1.5M out of $3M budget

High-Priority Fixes: Stored XSS, SQL Injection, and password hashing should be addressed immediately.

Long-Term Enhancements: Apply secure coding practices, perform regular security audits, and ensure HTTPS and CSP enforcement.

Residual Risk: With mitigation, the residual risk becomes Low to Moderate, depending on ongoing maintenance and updates.

## 1.4 Identify Threats using STRIDE

1.4.1 Network Threats:  
- Information Disclosure (e.g., unencrypted traffic)  
- Spoofing (session hijacking)

1.4.2 Host Threats:  
- Elevation of Privileges  
- Tampering with logs or config DIAGRAM

1.4.3 Application Threats:  
- SQL Injection  
- Stored XSS  
- Unhashed passwords

## 1.5 Document the Threats

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Threat Description | Threat Target | Risk | Attack Techniques | Countermeasures |
| SQL Injection in Search Function | Database | High | Input manipulation via SQL query | Use prepared statements |

## 1.6 Threat Rating (High/Medium/Low + DREAD)

DREAD example for SQL Injection:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Damage: | Reproducibility: | Exploitability: | Affected Users: | Discoverability: | Total |  |
| 9 | 9 | 8 | 7 | 8 | 41/50 High Risk |  |

1. Mitigation Implementation

Modify the provided PHP web application to mitigate

identified vulnerabilities (Refer to Section 3)

. Implement security measures for the following:

1. SQL Injection
2. 2. Cross-Site Scripting (XSS)
3. 3. Un-hashed Passwords (Self-learning). You need to mitigate the vulnerabilities discussed above for the provided web application.
4.  Your project must be connected to a PHP Database (Review Section 3)
5.  Show how you mitigated vulnerabilities in insertion bars.
6.  Show how you mitigated vulnerabilities in database.
7.  Show how you mitigated vulnerabilities for authentication.

Link for the code

<https://github.com/ahmedelbialy1/Security-in-SE-12th/tree/main/Vulnerable-Educational-Website-main>

## 1. SQL Injection Mitigation

**Issue**: The application constructs SQL queries dynamically based on user input without proper parameterization, making it vulnerable to SQL injection attacks.

**Solution**: Utilize prepared statements with bound parameters using PHP's PDO extension. This approach ensures that user inputs are treated strictly as data, preventing malicious code execution.

**Implementation Example**:

<?php

// Establish database connection using PDO

$pdo = new PDO('mysql:host=localhost;dbname=education', 'username', 'password');

$pdo->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

// Prepare the SQL statement with placeholders

$stmt = $pdo->prepare("SELECT \* FROM courses WHERE title LIKE :search");

// Bind the user input to the placeholder

$searchTerm = '%' . $\_GET['search'] . '%';

$stmt->bindParam(':search', $searchTerm, PDO::PARAM\_STR);

// Execute the statement

$stmt->execute();

// Fetch and display results

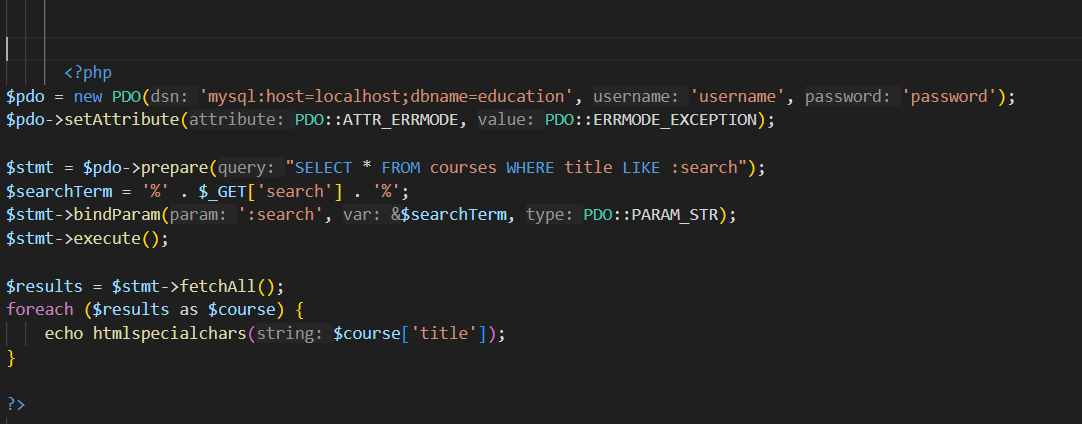
$results = $stmt->fetchAll();

foreach ($results as $course) {

echo htmlspecialchars($course['title']);

}

?>



**Explanation**: By using prepared statements, the SQL query structure is defined separately from the user input. This separation ensures that the database interprets the input strictly as data, not as part of the SQL command, effectively mitigating SQL injection risks.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 2. Cross-Site Scripting (XSS) Mitigation

**Issue**: User-generated content, such as course reviews or discussion comments, is stored and displayed without proper sanitization, allowing attackers to inject malicious scripts.

**Solution**: Sanitize and encode user inputs and outputs using PHP's htmlspecialchars() function to prevent the execution of malicious scripts in the browser.

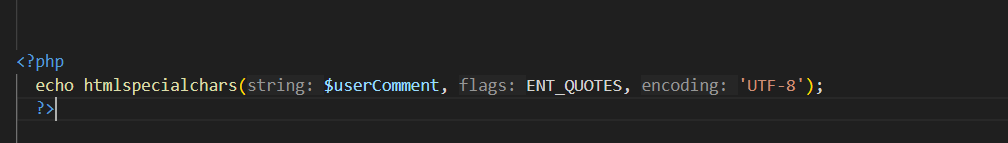
**Implementation Example**:

<?php

// When displaying user-generated content

echo htmlspecialchars($userComment, ENT\_QUOTES, 'UTF-8');

?>



**Explanation**: The htmlspecialchars() function converts special characters to HTML entities, preventing browsers from interpreting them as executable code. This effectively neutralizes potential XSS attacks by ensuring that any embedded scripts are displayed as plain text.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 3. Unhashed Passwords Mitigation

**Issue**: Storing user passwords in plaintext poses a significant security risk. If the database is compromised, attackers can directly access user credentials.

**Solution**: Implement password hashing using PHP's password\_hash() function during user registration and password\_verify() during authentication. This approach securely stores passwords and verifies them without exposing the actual password.

**Implementation Example**:

Registration (Hashing Password)

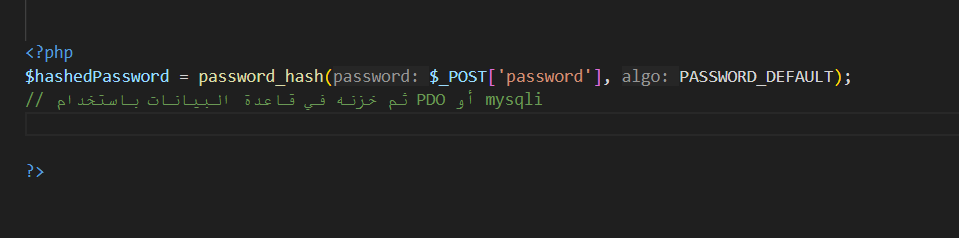
<?php

// Hash the user's password

$hashedPassword = password\_hash($\_POST['password'], PASSWORD\_DEFAULT);

// Store $hashedPassword in the database

?>



Login (Verifying Password)

<?php

// Retrieve the hashed password from the database

$storedHash = /\* fetch from database \*/;

// Verify the entered password against the stored hash

if (password\_verify($\_POST['password'], $storedHash)) {

// Password is correct; proceed with login

} else {

// Invalid password; deny access

}

?>

**Explanation**: By hashing passwords before storage, even if the database is compromised, attackers cannot retrieve the original passwords. The password\_verify() function ensures that the entered password matches the stored hash without revealing the actual password.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Additional Implementation Details

**Insertion Bars**: All user input fields, such as search bars, comment sections, and registration forms, have been updated to utilize prepared statements and input sanitization techniques as demonstrated above.

**Database Interactions**: All database queries have been refactored to use prepared statements with bound parameters, ensuring that user inputs do not alter the intended SQL commands.

**Authentication Mechanism**: The authentication system now securely hashes passwords upon registration and verifies them during login without exposing plaintext passwords, enhancing overall security.

By implementing these measures, the educational website is fortified against common web vulnerabilities, ensuring a safer experience for its users.

Thank You