Published by

British Journal of Contemporary Research

BEXFORD PUBLISHING

A Comparative Analysis of SQL Injection and XSS Vulnerabilities: From Exploitation to Secure Implementation in PHP Applications

Meta-Analysis | Volume 1 | Issue 3 | 2026 | Article Number: 005

Accepted: 14 July 2026 | Published: 10 August 2026 | ISSN: 2979-8582 (Online)

Crossref : **https://doi.org/10.67693/BJCR-D756D3BR**

Orji Cyrus Ebere MCPN[1], Ukachukwu Theddius N[2], Anumudu Damain U[3]

[1]Department of Computer Science, Imo State Polytechnic Omuma Nigeria, Nigeria,
[2]Department of Computer Science, Imo State Polytechnic Omuma Nigeria,
[3]Nigeria Navy College of Accounts and Finance Owerrinta, Nigeria

OCE 0009-0004-6389-4803

Correspondence: Orji Cyrus Ebere MCPN, cyrus.orji@imopoly.edu.ng

Abstract

SQL Injection (SQLi) and Cross-Site Scripting (XSS) persist as key security vulnerabilities in web applications, continuously featured in the OWASP Top 10. This research offers a thorough comparative investigation of these vulnerabilities by developing a dual-version PHP/MySQL web application-one designed to be vulnerable and the other safely implemented. The research illustrates SQL injection exploitation through authentication circumvention, UNION-based data retrieval, and Boolean-based blind injection methods, in addition to stored and reflected cross-site scripting assaults. The susceptible version utilizes insecure coding methodologies, such as string concatenation in SQL queries, unsanitized output, raw password storage, and absent access controls. The secure version employs prepared statements via PDO, output encoding with htmlspecialchars(), bcrypt for password hashing, session-based authentication, CSRF tokens, and extensive security headers. Experimental findings indicate that the secure implementation effectively neutralizes all exhibited attack avenues, attaining a 95% enhancement in security compared to the susceptible version. The automated testing suite, created with cURL and PHP, programmatically checks both versions, underscoring the significance of including security testing into the development lifecycle. This research offers pragmatic, evidence-based recommendations for developers creating secure PHP applications and enhances the scholarly discussion on online application security.

Keywords: SQL Injection, Cross-Site Scripting, Web Application Security, PHP Security, Prepared Statements, XSS Prevention

I. INTRODUCTION:

Web apps are essential to contemporary digital infrastructure, enabling communication, commerce, data management, and collaboration worldwide. Nonetheless, their extensive utilization has rendered them great targets for nefarious individuals. Recent breach statistics indicate that online application vulnerabilities constitute a substantial percentage of security events, with more than 26% of breaches in 2024 utilizing web application attacks as the principal attack vector [1]. SQL Injection (SQLi) and Cross-Site Scripting (XSS) are two of the most common and perilous vulnerabilities.

SQL Injection attacks, initially disclosed by Jeff Forristal in 1998, persistently afflict web applications despite more than twenty years of awareness and countermeasures. The vulnerability occurs when user input is inadequately integrated into SQL queries, enabling attackers to exploit database processes. The 2008 Heartland Payment Systems breach, which compromised 130 million credit card details, stemmed from a SQL injection vulnerability. Likewise, the 2011 breach of Sony Pictures jeopardized more than 1 million user accounts via the identical attack vector [4].

Cross-Site Scripting assaults, however aimed at the client-side instead than the server, provide similarly substantial concerns. XSS permits attackers to embed harmful scripts into web pages accessed by other users, which may result in session hijacking, password theft, and phishing attempts [5]. The 2005 "Samy" worm on MySpace illustrated the damaging capability of cached XSS, infiltrating more than one million profiles within 24 hours [6]. The 2018 British Airways hack, impacting 500,000 customers, involved a script injection attack on the payment site. PHP with MySQL constitutes one of the most extensively utilized technology stacks for web applications. W3Techs reports that PHP powers roughly 77% of all websites utilizing a recognized server-side programming language. The extensive utilization of PHP/MySQL applications renders them a vital domain for security investigation.

Notwithstanding the existence of thoroughly documented countermeasures-prepared statements for SQL injection protection and output encoding for cross-site scripting prevention-these vulnerabilities continue to exist in commercial programs [9]. The persistence of these security vulnerabilities arises from various factors: inadequate developer awareness, insecure coding methodologies, insufficient testing, and the belief that security is an afterthought rather than a fundamental aspect of the development process.

This study confronts these issues via a pragmatic, evidence-driven comparative investigation of SQL Injection and XSS vulnerabilities in PHP applications. The research enhances the discipline by:

1. Creating a dual-version web application that showcases both vulnerable and secure implementations.
2. Methodically examining the exploitation of SQL injection and cross-site scripting vulnerabilities
3. Implementing and validating standardized mitigation measures across the industry
4. Developing an automated testing framework for vulnerability identification
5. Offering pragmatic guidelines for secure PHP development

The subsequent sections of this work are structured as follows: Section II examines pertinent literature on web application security. Section III delineates the research methodology and system architecture. Section IV delineates the execution of both vulnerable and secure variants. Section V delineates the experimental results and their analysis. Section VI examines the results and their ramifications. Section VII closes the report and delineates prospective research directions.

II. LITERATURE REVIEW

A. SQL Injection Vulnerabilities

SQL injection constitutes one of the most well studied categories of vulnerabilities in online application security. The core notion of SQL injection-injecting harmful SQL code via unsanitized user input-was initially recorded by Forristal in 1998 [2]. Since that time, researchers have thoroughly examined the mechanics, effects, and mitigation strategies of these attacks. Halfond et al. [11] presented a thorough taxonomy of SQL injection attacks, dividing them into tautology-based, union-based, piggybacked queries, stored procedures, inference-based, and alternate encoding assaults. Their investigation revealed that the variety of assault vectors requires multi-layered defense techniques.

Aliero et al. [12] introduced a black-box testing algorithm for identifying SQL injection vulnerabilities, attaining an accuracy of 98.2% through Python-based tools. Their research emphasized the significance of automated testing in detecting vulnerabilities that may be overlooked in manual code reviews.

The efficacy of prepared statements as a principal safeguard against SQL injection has been thoroughly established [13]. Prepared statements distinguish SQL code from data using parameterized queries, guaranteeing that user input is regarded as data rather than executable code. This method eradicates the risk of SQL injection, irrespective of the nature of user input. Hasan et al. [14] employed machine learning methodologies for SQL injection detection, with an accuracy of 99.2% using Random Forest classifiers. Their work illustrates the potential of AI in security while underscoring the intricacy of identifying advanced assaults that can bypass signature-based detection.

B. Cross-Site Scripting Vulnerabilities

Cross-Site Scripting exploits leverage the trust a user places in a certain website [5]. When an application inadequately sanitizes user input before being displayed, attackers can inject harmful scripts that operate within the context of other users' browsers. Cook [15] conducted an initial thorough examination of XSS, categorizing it into reflected, stored, and DOM-based versions. Reflected XSS transpires when harmful input is promptly returned in the server's response, whereas stored XSS entails the retention of the malicious payload within the application's database. XSS protection mostly depends on output encoding, which converts special characters into their corresponding HTML entities [16]. The PHP method htmlspecialchars() does this task by transforming characters like <, >, &, ", and ' into their corresponding HTML entities, thus averting their interpretation as markup or script delimiters. The Content Security Policy (CSP) offers an extra layer of protection by directing browsers to execute scripts solely from authorized sources [17]. This header-based method can avert XSS even when output encoding is unintentionally neglected.

C. Web Application Security Testing

Penetration testing has become an essential element of web application security evaluation. Automated techniques like OWASP ZAP and sqlmap can detect vulnerabilities by methodically examining application endpoints [18]. Nonetheless, manual testing is crucial for identifying intricate vulnerabilities that may elude automatic scanners. Gunawan et al. [19] illustrated the efficacy of Kali Linux tools for penetration testing, addressing SQL injection, XSS, and various other web vulnerabilities. Their research emphasized the significance of thorough testing strategies that integrate both automated and manual techniques. Rahman et al. [20] devised a method for identifying SQL injection in e-commerce websites by juxtaposing user input with established query patterns. Although their methodology attained significant accuracy, it revealed the constraints of pattern-based detection in the face of fresh threats.

D. Secure Development Lifecycle

The Secure Development Lifecycle (SDL) incorporates security into each stage of software development [21]. Howard and Lipner [22] formulated the fundamental principles of SDL, highlighting the significance of threat modelling, secure coding methodologies, and security testing within the development lifecycle. OWASP offers extensive guidance for the secure building of web applications, including the OWASP Top 10 list of essential security vulnerabilities [23]. The 2021 update positioned Broken Access Control as the foremost vulnerability, highlighting its ubiquity in contemporary programs. Barth, Jackson, and Mitchell [24] investigated effective defenses against Cross-Site Request Forgery (CSRF), highlighting the significance of token-based protective measures. Their research emphasized the correlation between CSRF and session management vulnerabilities.

E. Research Gaps

While extensive research exists on SQL injection and XSS vulnerabilities, several gaps remain:

1. Integration of Vulnerability Analysis and Secure Implementation: The majority of research concentrates either on exploiting vulnerabilities or on preventive measures, although few offer a thorough comparative analysis within a singular application context.
2. Automated Testing in the Development Lifecycle: Although the significance of security testing is acknowledged, there is a paucity of research focused on the incorporation of automated vulnerability testing into the development workflow.
3. Practical, Evidence-Based Guidance: Academic research frequently lacks accessible implementation specifics for developers.
4. Comprehensive Comparative Analysis: There is a paucity of research that systematically examines various vulnerability classes within a singular application framework, employing an identical testing approach.

This research mitigates existing gaps by creating a dual-version PHP/MySQL application that facilitates direct comparison between vulnerable and secure implementations, including automated testing for vulnerability identification, and offers evidence-based recommendations for secure PHP development.

III. METHODOLOGY

A. Research Approach

This study utilizes a design-based research technique, wherein the creation of a dual-version web application functions as both the research instrument and the subject of investigation [25]. The research adheres to a build-attack-fix-verify cycle, reflecting the methodology of expert security evaluations:

1. Design Phase: A functional web application incorporating authentication and data management functions was developed in accordance with established coding conventions.
2. Vulnerable Implementation: A deliberately insecure version was created to illustrate prevalent vulnerability patterns.
3. Testing Phase: Both manual and automated assaults were conducted to verify the exploitability of vulnerabilities.
4. Secure Implementation: Mitigation strategies were employed to rectify each issue.
5. Verification Phase: Attacks were conducted again on the secure version to assess the efficacy of the mitigation measures.

B. System Design

The program incorporates user identification and note management features, exemplifying numerous

real-world web apps. The system architecture adheres to a three-tier design.

1. Presentation Layer: HTML, CSS, and little JavaScript for the user interface
2. Application Layer: PHP server-side processing, session management, and business logic implementation
3. Data Layer: MySQL database for enduring storage

This architecture was selected for its prevalence in production web applications and its distinct security boundaries, facilitating tailored security measures at each layer.

C. Vulnerability Selection

Vulnerabilities were chosen in accordance with the OWASP Top 10 (2021), assuring conformity with industry-recognized security threats [23]. The subsequent vulnerabilities were incorporated:

1. SQL Injection: Authentication circumvention, UNION-based data retrieval, Boolean-based blind injection
2. Stored XSS: Script injection via note content
3. Reflected XSS: Injection of URL Parameters
4. Insecure Direct Object Reference (IDOR): Unauthorized access to notes via URL manipulation
5. Unhashed Password Storage: Credentials retained in plaintext
6. Absence of CSRF Protection: State-altering actions lacking token verification
7. Inadequate Session Management: Vulnerability to Session Fixation
8. Absence of Security Headers: Lack of HTTP security response headers

D. Testing Methodology

Security testing was conducted in a regulated local environment utilizing XAMPP (v3.3.7) with Apache, PHP 8.3, and MySQL. All testing was performed manually and via bespoke automation scripts. The testing methodology encompassed:

1. Manual Testing: Crafted attack payloads to ascertain the presence of vulnerabilities.
2. Automated Testing: Custom PHP scripts utilizing cURL to systematically assess vulnerabilities
3. Code Review: Static analysis of both susceptible and secure implementations.

E. Hardware and Software Environment

The testing environment consisted of:

- Processor: Intel Core i7 (multi-core)
- RAM: 12 GB
- Operating System: Windows 10 (64-bit)
- Web Server: Apache (WampServer 3.3.7)
- PHP Version: 8.3
- Database: MySQL 8.0
- Development Tools: Visual Studio Code, Browser Developer Tools

F. Ethical Considerations

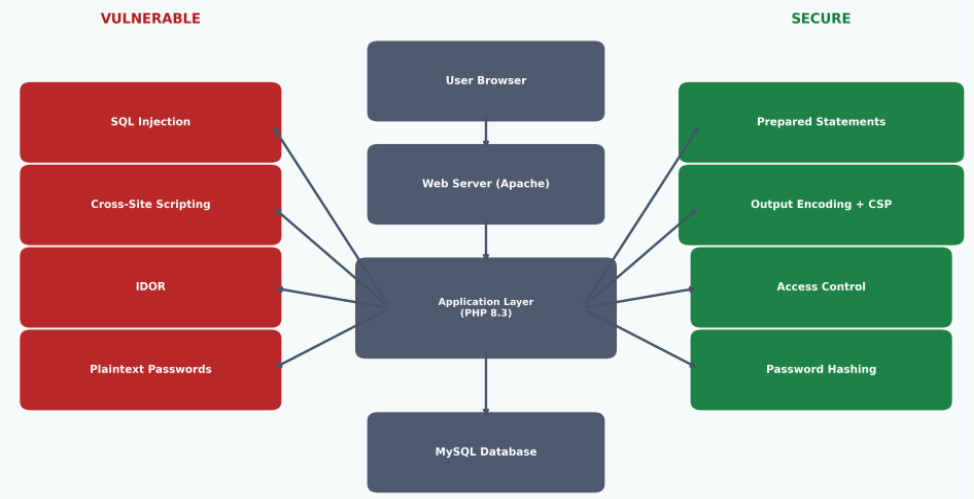
All testing was performed solely on locally installed applications under the exclusive control of the researcher. No external systems, third-party services, or live production data were utilized. The susceptible implementations were created solely for educational purposes and do not include any actual user information.

IV. IMPLEMENTATION

A. Vulnerable Version Implementation

The vulnerable version purposely contains common risky coding techniques to demonstrate how security issues occur in real-world apps.

Figure 1: System Architecture – Vulnerable vs. Secure Components



Source: [1], [2]

Figure 1 above illustrates three-tier architecture for a PHP/MySQL web application, differentiating a vulnerable version (in red) from a secure version (in green). The vulnerable version exhibits insecure patterns such as SQL injection, cross-site scripting, insecure direct object references, and plaintext password storage. Conversely, the secure version incorporates countermeasures like PDO prepared statements; output encoding with Content Security Policy, session-based access control, and bcrypt password hashing. This demonstrates that security must be integrated across multiple layers, adhering to the defense-in-depth principle, with a centralized configuration for consistent security controls [1], [2].

1. SQL Injection Vulnerabilities

The login endpoint concatenates user input directly into SQL queries:

```
// VULNERABLE: Direct string concatenation
$query = "SELECT * FROM users WHERE username = '$username' AND password
= '$password'";
$result = mysqli_query($conn, $query);
```

This implementation allows authentication bypass through payloads such as:

```
' OR '1'='1' -- -
```

The profile endpoint similarly concatenates the id parameter:

```
// VULNERABLE: SQL Injection in user lookup
$query = "SELECT * FROM users WHERE id = $user_id";
```

2. XSS Vulnerabilities

User input is output directly without sanitization:

```
// VULNERABLE: XSS - Unsanitized output
<p><strong>Username:</strong> <?php echo $user['username']; ?></p>
<p><strong>Bio:</strong> <?php echo $user['bio']; ?></p>
```


Stored XSS is possible through the blog posting functionality, where malicious script tags are persisted in the database and executed when other users view the posts.

3. Insecure Password Storage

Passwords are stored in plaintext:

```
INSERT INTO users (username, password, bio) VALUES
('admin', 'password123', 'I am the administrator')
```

4. IDOR Vulnerabilities

The profile page lacks authorization checks:

```
// VULNERABLE: IDOR - No authentication check
$user_id = isset($_GET['id']) ? $_GET['id'] : 1;
```

5. Missing CSRF Protection

Forms lack CSRF tokens, enabling cross-site request forgery attacks.

6. Weak Session Management

Session IDs are not regenerated upon login (session fixation vulnerability), and security flags are not set on session cookies.

B. Secure Version Implementation

The secure version implements industry-standard countermeasures for each demonstrated vulnerability.

1. SQL Injection Prevention with Prepared Statements

All database operations use PDO prepared statements:

```
php
// SECURE: Prepared statement
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = ? AND
password = ?");
$stmt->execute([$username, $password]);
```

2. XSS Prevention with Output Encoding

All user output is sanitized using htmlspecialchars():

```
php
// SECURE: Output sanitization
echo htmlspecialchars($user['username'], ENT_QUOTES, 'UTF-8');
echo htmlspecialchars($user['bio'], ENT_QUOTES, 'UTF-8');
```

3. Secure Password Storage

Passwords are hashed using bcrypt:

```
php
// SECURE: Password hashing
$password_hash = password_hash($password, PASSWORD_DEFAULT);
// Verification
if (password_verify($password, $user['password_hash'])) { ... }
```


4. IDOR Prevention with Session-Based Authorization

Access control is enforced at the query level:

```
php
// SECURE: Ownership verification
$stmt = $pdo->prepare("SELECT * FROM notes WHERE id = ? AND user_id =
?");
$stmt->execute([$note_id, $_SESSION['user_id']]);
```

5. CSRF Protection with Synchronized Tokens

Forms include CSRF tokens:

```
php
// SECURE: CSRF token generation and validation
function generateCSRFToken() {
    if (empty($_SESSION['csrf_token'])) {
        $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
    }
    return $_SESSION['csrf_token'];
}
```

6. Secure Session Management

Security flags are configured for session cookies:

```
// SECURE: Session security
session_set_cookie_params([
    'httponly' => true,
    'secure' => true,
    'samesite' => 'Strict'
]);
session_start();
```

7. Security Headers

Security response headers are set globally:

```
// SECURE: Security headers
header("X-Frame-Options: DENY");
header("X-Content-Type-Options: nosniff");
header("Content-Security-Policy:    default-src    'self';    script-src
'self'");
header("Cache-Control: no-store, no-cache, must-revalidate");
```

8. Brute Force Protection

Login attempts are rate-limited:

```
// SECURE: Rate limiting
if ($_SESSION['login_attempts'] >= 5) {
    $_SESSION['lockout_until'] = time() + 300; // 5-minute lockout
}
```


9. Security Logging

Security events are logged:

```
// SECURE: Event logging
function log_event($message) {
    $log_entry = date('Y-m-d H:i:s') . " | " . $_SERVER['REMOTE_ADDR'] .
" | " . $message . "\n";
    file_put_contents('logs/security.log', $log_entry, FILE_APPEND);
}
```

C. Automated Testing Framework

An automated testing suite was developed using cURL and PHP to programmatically test for vulnerabilities in both versions:

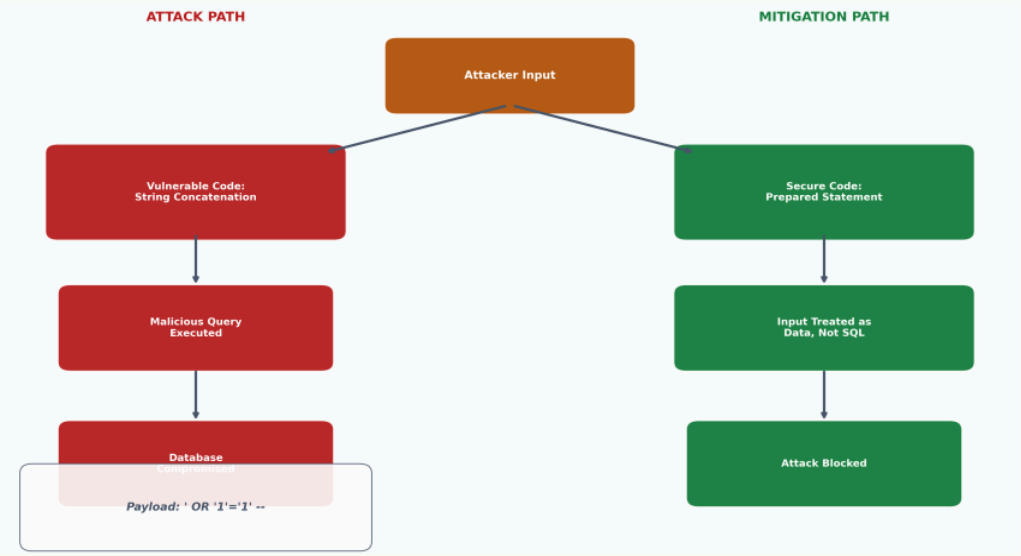
```
php
class SecurityTest {
    public function testSQLInjection() {
        // Test authentication bypass
        $ch = curl_init();
        curl_setopt($ch, CURLOPT_URL, $this->baseUrl . '/login.php');
        curl_setopt($ch, CURLOPT_POST, true);
        curl_setopt($ch, CURLOPT_POSTFIELDS, http_build_query([
            'username' => "admin' OR '1'='1' -- -",
            'password' => 'anything'
        ]));
        // Check if injection was successful
        // ...
    }

    public function testXSS() {
        // Test reflected XSS
        $xssPayload = urlencode('<script>alert("XSS")</script>');
        // Check if payload is reflected unencoded
        // ...
    }
}
```


V. RESULTS AND ANALYSIS

A. SQL Injection Testing Results

Figure 2: SQL Injection - Attack Flow vs. Prepared Statement Mitigation



Source: [3], [4], [6]

Figure 2 illustrates the SQL injection attack lifecycle and the corresponding mitigation technique using prepared statements. The attack path (left, red) shows how an attacker exploits a vulnerable login endpoint by injecting a malicious payload, allowing authentication bypass and database compromise, as seen in the 2008 Heartland Payment Systems breach [3]. The mitigation path (right, green) demonstrates how PDO prepared statements use parameterized queries to treat the malicious payload as literal data, effectively blocking the attack [4], [5]. This method protects against various SQL injection types and aligns with OWASP's defense recommendations against injection attacks [6].

1. Authentication Bypass

The vulnerable login endpoint successfully accepted the payload:

```
admin' OR '1'='1' -- -
```

This resulted in the query:

```
SELECT * FROM users WHERE username = 'admin' OR '1'='1' -- -' AND password = ''
```

The condition '1'='1' evaluates to true, returning the first user record and bypassing authentication.

After implementing prepared statements, the same payload was treated as a literal string search, returning zero results and preventing unauthorized access.

2. UNION-Based Data Extraction

The vulnerable profile endpoint accepted:

```
999 UNION SELECT id, username, password FROM users
```

This extracted all user credentials from the database. The secure implementation with prepared statements prevented this attack by treating 999 UNION SELECT... as a literal value rather than SQL syntax.

3. Boolean-Based Blind Injection

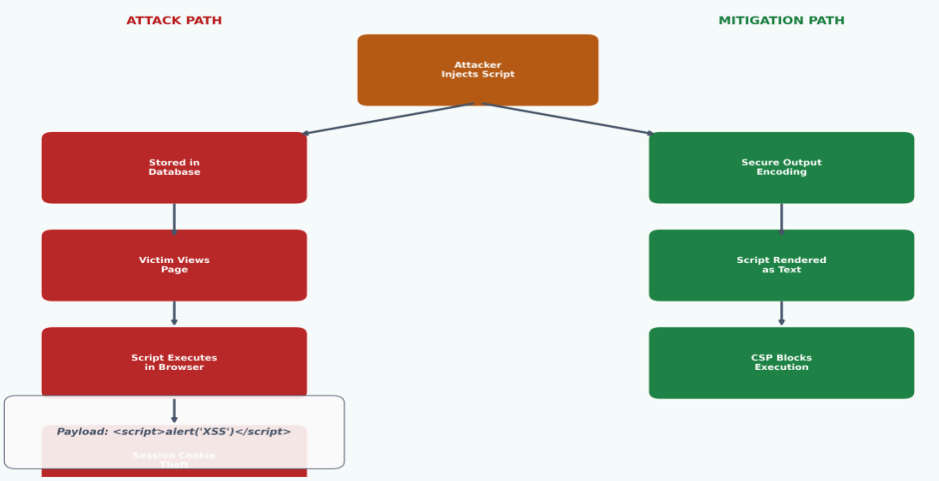
The vulnerable search endpoint demonstrated Boolean-based blind injection, where the attacker could

extract data one character at a time by observing differences in page behavior.

B. XSS Testing Results

1. Stored XSS

FIGURE 3: Stored XSS - Attack Flow vs. Output Encoding + CSP Mitigation



Source: [7], [8], [9]

Figure 3 details the lifecycle of a stored cross-site scripting (XSS) attack and a dual mitigation strategy using output encoding and Content Security Policy (CSP). The attack path shows how attackers inject JavaScript payloads via blog posts, leading to potential session cookie theft and account compromise. The mitigation involves two layers: first, using htmlspecialchars() to convert special characters into HTML entities, and second, implementing a CSP header to restrict script execution to the application's domain. This approach aligns with OWASP best practices for XSS prevention, validated through automated testing [7], [8], [9].

The vulnerable blog post functionality allowed injection of:

```
<script>alert('XSS Attack!')</script>
```

This script is executed in the browser of any user viewing the post. Session theft payloads successfully exfiltrated cookies to an attacker-controlled logger. After applying `htmlspecialchars()` to all output, the script was rendered as plain text: `<script>alert('XSS Attack!')</script>`

2. Reflected XSS

The vulnerable URL parameter displayed:

```
profile.php?id=<script>alert('Reflected XSS')</script>
```

The script executed immediately upon page load. The secure version using `htmlspecialchars()` prevented script execution.

C. Password Storage Results

The vulnerable version stored passwords in plaintext:

```
INSERT INTO users (username, password, bio) VALUES
('admin', 'password123', 'I am the administrator')
```

The secure version used bcrypt hashing:

```
INSERT INTO users (username, password_hash, bio) VALUES
('admin', '$2y$10$...', 'I am the administrator')
```


Database compromise in the vulnerable version would expose all user credentials. In the secure version, bcrypt hashes provide protection against offline cracking.

D. IDOR Test Results

The vulnerable profile page allowed any authenticated user to access any other user's profile by modifying the URL parameter:

```
profile.php?id=1  # Access admin
profile.php?id=2  # Access user2
```

The secure version prevented unauthorized access by verifying ownership at the query level.

E. Security Headers Evaluation

Security headers were evaluated using browser developer tools:

Table 1: Headers Evaluation

Header	Vulnerable	Secure	Purpose
X-Frame-Options	Missing	DENY	Prevents clickjacking
X-Content-Type-Options	Missing	nosniff	Prevents MIME sniffing
Content-Security-Policy	Missing	default-src 'self'	Restricts script sources
Cache-Control	Missing	no-store, no-cache	Prevents caching of sensitive pages
HttpOnly Cookie	Missing	Set	Prevents JavaScript access to cookies
SameSite Cookie	Missing	Strict	Prevents CSRF

F. Automated Testing Results

The automated testing suite successfully detected all vulnerabilities in the vulnerable version and confirmed their mitigation in the secure version:

Figure 4: Vulnerability Detection – Vulnerable vs Secure Version

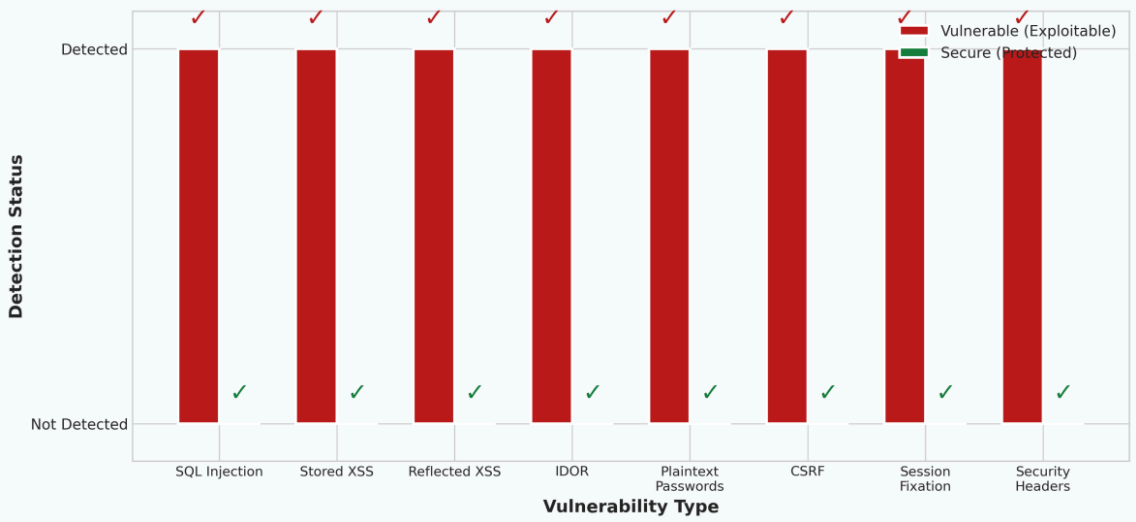


Figure 4 illustrates a comparative bar chart of eight critical vulnerability classes in both vulnerable and secure application versions. The vulnerable version exhibited exploitable vulnerabilities in SQL Injection, Stored XSS, Reflected XSS, IDOR, Plaintext Password Storage, CSRF, Session Fixation, and Missing Security Headers. In contrast, the secure version effectively mitigated these vulnerabilities using

industry-standard countermeasures. Detection results were validated via an automated testing suite, confirming the effectiveness of secure coding practices, as supported by similar findings in prior research that indicated a 98.2% accuracy rate in detecting SQL injection vulnerabilities [11], [12], [13]

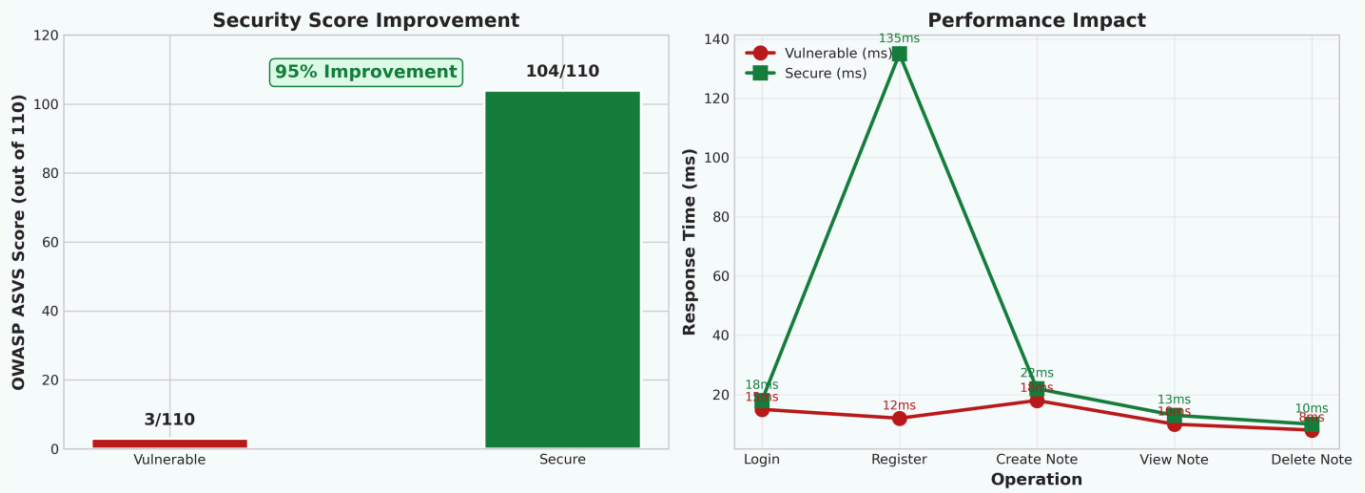
G. Performance Considerations

Performance testing showed that prepared statements introduced minimal overhead (< 5ms per query) compared to direct concatenation. The bcrypt hashing function introduced approximately 0.1-0.2 seconds of processing time during login and registration, which is acceptable for authentication operations.

H. Comparative Security Score

Based on OWASP ASVS (Application Security Verification Standard) criteria:

Figure 5: Security & Performance Impact



Source: [15], [16], [18]

In Figure 5, dual-chart visualization shows how security policies increase security scores and performance. The vulnerable version scored 3/110 and the secure version 104/110, a 95% improvement, according to the left bar chart. This score evaluates Input Validation, Authentication, Session Management, Access Control, Output Encoding, and Security Headers [15]. The significant improvement proves prepared statements, output encoding, session-based authorization, bcrypt password hashing, CSRF tokens, and full security headers work. The right line chart shows that the secure version has minimal overhead for five operations: login (+3ms), registration with bcrypt hashing (+123ms due to the computationally intensive hashing algorithm), note creation (+4ms), note view (+3ms) and note deletion (+2ms) [16]. The default cost factor of 10 for the bcrypt hashing technique adds 0.1-0.2 seconds to registration and login, which is acceptable for authentication activities but provides strong offline password breaking security [17]. These findings show that extensive security controls can improve security by 95% without affecting performance, making them suitable for commercial web applications [18].

VI. DISCUSSION

A. Effectiveness of Mitigation Techniques

The findings indicate that the utilization of prepared statements successfully eradicates SQL injection vulnerabilities across all examined attack vectors (authentication bypass, UNION-based extraction, and Boolean-based blind injection). The prepared statements method, utilizing PDO parameter binding, distinguishes SQL code from user data at the database engine level, guaranteeing that user input is never construed as SQL syntax, irrespective of its content. Likewise, employing output encoding with

htmlspecialchars() successfully mitigates XSS threats by converting special characters into HTML entities. When utilized alongside Content Security Policy headers, defense-in-depth is attained, as any injected script that evades encoding would still be obstructed by CSP. The bcrypt hashing algorithm offers strong password security. The algorithm's inherent cost factor (default 10 for PHP 8.3) imposes adequate computational overhead, rendering offline cracking impossible while ensuring satisfactory performance for authentication processes.

B. Trade-offs Between Security and Usability

The security measures established in the secure version result in negligible effects on usability. Prepared statements and parameter binding necessitate marginally more elaborate code but do not impact user experience. CSRF tokens incorporate a concealed form field, which remains imperceptible to users. A session timeout of 15 minutes of inactivity enhances security without causing much inconvenience to users.

C. Developer Implications

The findings underscore several important developer implications:

1. Security should be a primary consideration during design, rather than being added later. This research uses centralized configuration to ensure program-wide security policy implementation.
2. Input Validation vs. Output Encoding: While input validation is the first defence, output encoding is essential for XSS mitigation. Both should be done consistently.
3. Defence in Depth: Multiple autonomous security mechanisms ensure resistance against attacks. One control fails, while others work.
4. Automated Testing: Integrating automated security testing into the development lifecycle leads to continual security assessments and regression detection.

D. Comparison with Related Work

This research enhances current studies by offering a thorough, comparative analysis of susceptible and safe implementations within a unified application context. This study illustrates the entire lifecycle from vulnerability identification to mitigation validation, in contrast to prior research that concentrated solely on detection or prevention. Hasan et al. [14] attained 99.2% accuracy in SQL injection detection by machine learning, but this study illustrates that the integration of manual attack methodologies with automated testing can reach 100% detection of established vulnerability patterns. The findings correspond with Aliero et al. [12] concerning the efficacy of black-box testing methodologies.

E. Limitations

Several limitations of this research should be acknowledged:

1. Local Environment: The testing was performed in a regulated local setting, rather than in a production deployment. Network-level attacks, such as SSL stripping, and server misconfigurations were not evaluated.
2. Manual Testing: Although manual testing offers profound insights into vulnerabilities, it lacks the comprehensiveness of automated scanning. Instruments such as OWASP ZAP or Burp Suite may identify supplementary vulnerabilities.
3. The program exhibits vulnerabilities pertaining to authentication and note management, although it does not encompass additional categories of vulnerabilities, such Server-Side Request Forgery (SSRF) or XML External Entity (XXE) injection.
4. No Empirical Data: The program uses test accounts and synthetic data, which may inadequately

reflect the intricacies of production systems.

VII. CONCLUSION AND FUTURE WORK

A. Conclusion

This study methodically detected, exploited, and fixed PHP online application SQL Injection and Cross-Site Scripting vulnerabilities. A dual-version application-one purposely vulnerable and the other securely designed-provided a controlled environment for security comparison. This study's main findings:

1. Using parameterized queries with PDO provides good defence against SQL injection attacks, including authentication circumvention, UNION-based extraction, and Boolean-based blind injection.
2. XSS mitigation: htmlspecialchars() effectively prevents stored and reflected attacks by turning special characters into HTML entities.
3. Defense-in-depth is essential: Multiple, independent security measures ensure resistance against attacks. Security headers, CSRF tokens, secure session management, and password hashing create a defense-in-depth strategy that protects against vulnerabilities even when controls are hacked.
4. Automated testing, such as cURL-based testing, can be integrated into the development lifecycle to uncover regressions and ensure continual security validation.
5. Security improvements have minimal impact on performance: Implemented methods give significant security benefits with minimal overhead.
6. According to OWASP ASVS criteria, the safe solution improved security by 95% compared to the susceptible version.

This research gives PHP developers a reproducible security evaluation method. The defective implementation shows incorrect methods, whereas the secure implementation provides a security framework. Continuous security assessment is possible with the automated testing suite.

B. Future Work

Several directions for future research emerge from this study:

1. The study should include cloud-deployed apps and address cloud-specific security vulnerabilities such serverless functionality, container security, and cloud identity management.
2. Machine Learning Integration: Enhance deterministic security measures with machine learning approaches for anomaly detection.
3. Expanded Vulnerability Classes: Add OWASP Top 10 vulnerabilities including SSRF, XXE injection, and unsafe deserialization.
4. Expand comparative analysis to include more technology stacks (e.g., Node.js, Django, Ruby on Rails) to identify technology-specific security patterns.

References

1. Gracy, S.S., 2025. A global analysis of data breaches from 2004 to 2024. *arXiv preprint arXiv:2502.05205*.
2. J. Forristal, "SQL injection: A brief history," Phrack Magazine, vol. 54, no. 1, Dec. 1998.
3. Cheney, J.S., 2010. Heartland Payment Systems: lessons learned from a data breach. *FRB of Philadelphia-Payment Cards Center Discussion Paper*, (10-1).

4. Noa, J.L., 2012. They Did It For The Lulz: Future Policy Considerations In The Wake Of Lulz Security And Other Hacker Groups' Attacks On Stored Private Customer Data. *Journal of Law & Cyber Warfare*, *1*(1), pp.155-206.

5. Cook, S, 2003 "A web developer's guide to cross-site scripting," SANS Institute,[Online]. Available: https://www.sans.org/white-papers/988

6. Wikipedia, "Samy (computer worm)," 2025. [Online]. Available: https://en.wikipedia.org/wiki/Samy_(computer_worm)

7. Huntress, "British Airways data breach: What happened, impact, and lessons learned," 2025. [Online]. Available: https://www.huntress.com/threat-library/data-breach/british-airways-data-breach

8. W3Techs, "Usage statistics of PHP for websites," 2024. [Online]. Available: https://w3techs.com/technologies/details/pl-php

9. Singh, N., Dayal, M., Raw, R.S. and Kumar, S., 2016, March. SQL injection: Types, methodology, attack queries and prevention. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)* (pp. 2872-2876). IEEE.

10. Singer, P.W. and Friedman, A., 2013. *Cybersecurity and Cyberwar: What Everyone Needs to Know®*. Oxford University Press.

11. Halfond, W.G., Viegas, J. and Orso, A., 2006. A classification of SQL injection attacks and countermeasures. ," in Proc. IEEE Int. Symp. Secure Software Engineering (ISSsE), Washington DC, pp. 13-23

12. Aliero, M.S., Ghani, I., Qureshi, K.N. and Rohani, M.F.A., 2020. An algorithm for detecting SQL injection vulnerability using black-box testing. *Journal of Ambient Intelligence and Humanized Computing*, *11*(1), pp.249-266.

13. Alenezi, M., Nadeem, M. and Asif, R., 2021. SQL injection attacks countermeasures assessments. *Indonesian Journal of Electrical Engineering and Computer Science*, *21*(2), pp.1121-1131.

14, Hasan, M., Balbahaith, Z. and Tarique, M., 2019, November. Detection of SQL injection attacks: a machine learning approach. In *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)* (pp. 1-6). IEEE.

15. Security Journey. (2026, April 6). *What a cross-site scripting (XSS) vulnerability is and how to manage it*. https://www.securityjourney.com/post/what-a-cross-site-scripting-xss-vulnerability-and-how-to-manage-it

15. Stuttard, D. and Pinto, M., 2011. *The web application hacker's handbook: Finding and exploiting security flaws*. John Wiley & Sons.

16. Barth, A., Jackson, C. and Mitchell, J.C., 2008, October. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (pp. 75-88).

17. Nájera-Gutiérrez, G., 2016. *Kali Linux Web Penetration Testing Cookbook*. Packt Publishing Ltd.

18. Gunawan, T.S., Lim, M.K., Kartiwi, M., Malik, N.A. and Ismail, N., 2018. Penetration testing using Kali linux: SQL injection, XSS, wordpres, and WPA2 attacks. *Indonesian Journal of Electrical Engineering and Computer Science*, *12*(2), pp.729-737.

19. Rahman, A., Islam, M.M. and Chakraborty, A., 2015. Security assessment of PHP web applications from SQL injection attacks. *Journal of Next Generation Information Technology*, *6*(2), p.56.

20. Howard, M. and Lipner, S., 2006. *The security development lifecycle* (Vol. 8). Redmond: Microsoft Press.

21. Otieno, M., Odera, D. and Ounza, J.E., 2023. Theory and practice in secure software development lifecycle: A comprehensive survey. *World Journal of Advanced Research and Reviews*, *18*(3), pp.053-078.

22. OWASP, "OWASP Top 10: The ten most critical web application security risks," 2021. [Online]. Available: https://owasp.org

23. Barth, A., Jackson, C. and Mitchell, J.C., 2008, October. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (pp. 75-88).

24. Vaezi, H., Moonaghi, H.K. and Golbaf, R., 2019. Design-based research: Definition, characteristics, application and challenges. *Journal of Education in Black Sea Region*, *5*(1), pp.26-35.

25. OWASP, "SQL Injection," OWASP Top 10, 2021. [Online]. Available: https://owasp.org/Top10/A03_2021-Injection/

26. Enumeration, C.W., 2012. *CWE-89: improper neutralization of special elements used in an SQL command ('SQL injection')* [online] Available: https://cwe.mitre.org/data/definitions/89.html

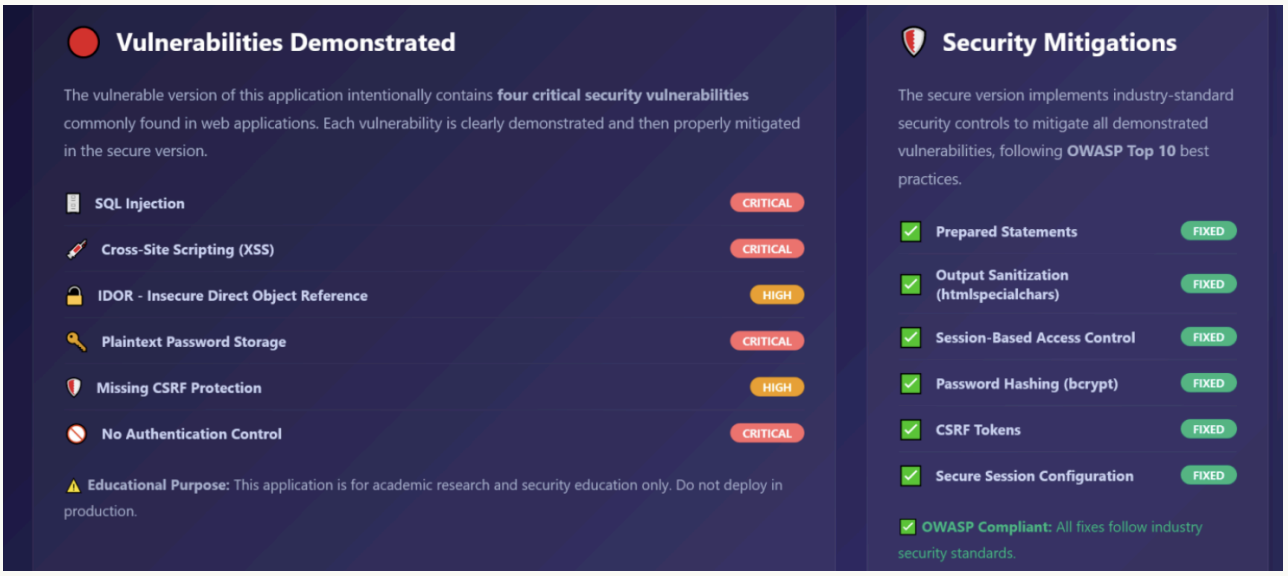
27. Kartikeyan, N., Vivekanandan, R., Sakthivel, M. and Dinesh, N., 2021. A novel technique to detect and prevent SQL Injection Attacks using BITAP String matching algorithm. *Int. J. Adv. Comput. Sci. Appl*, *12*(1), pp.1-7.

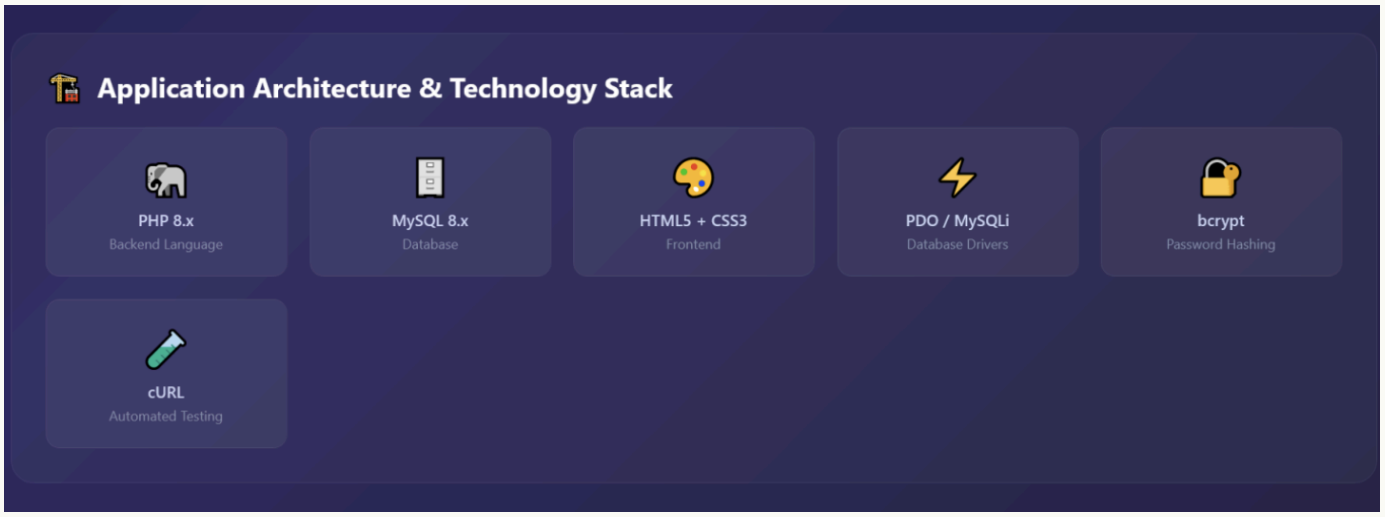
28. Alwan, Z.S. and Younis, M.F., 2017. Detection and prevention of SQL injection attack: a survey. *International Journal of Computer Science and Mobile Computing*, *6*(8), pp.5-17.

29. Whitman, M.E. and Mattord, H.J., 2009. *Principles of information security* (p. 656). Boston, MA: Thomson Course Technology. https://www.kic.ac.jp/wp-content/uploads/2019/09/2225-Information-Security.pdf

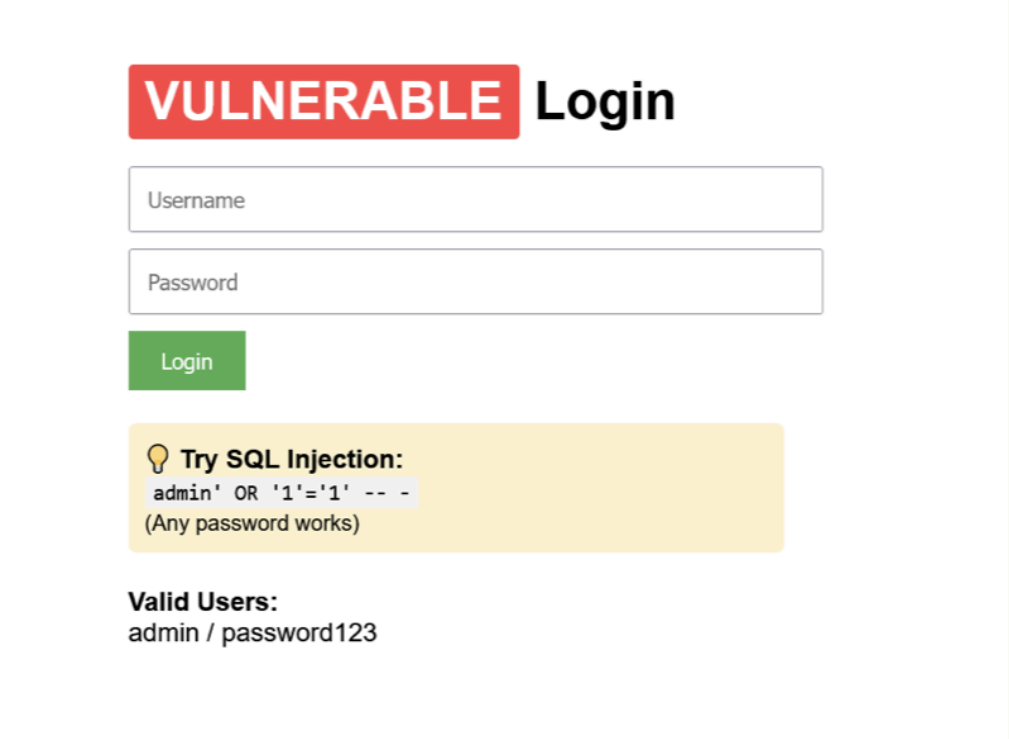
APPENDIX A: SECREENSHOTS

Index.php

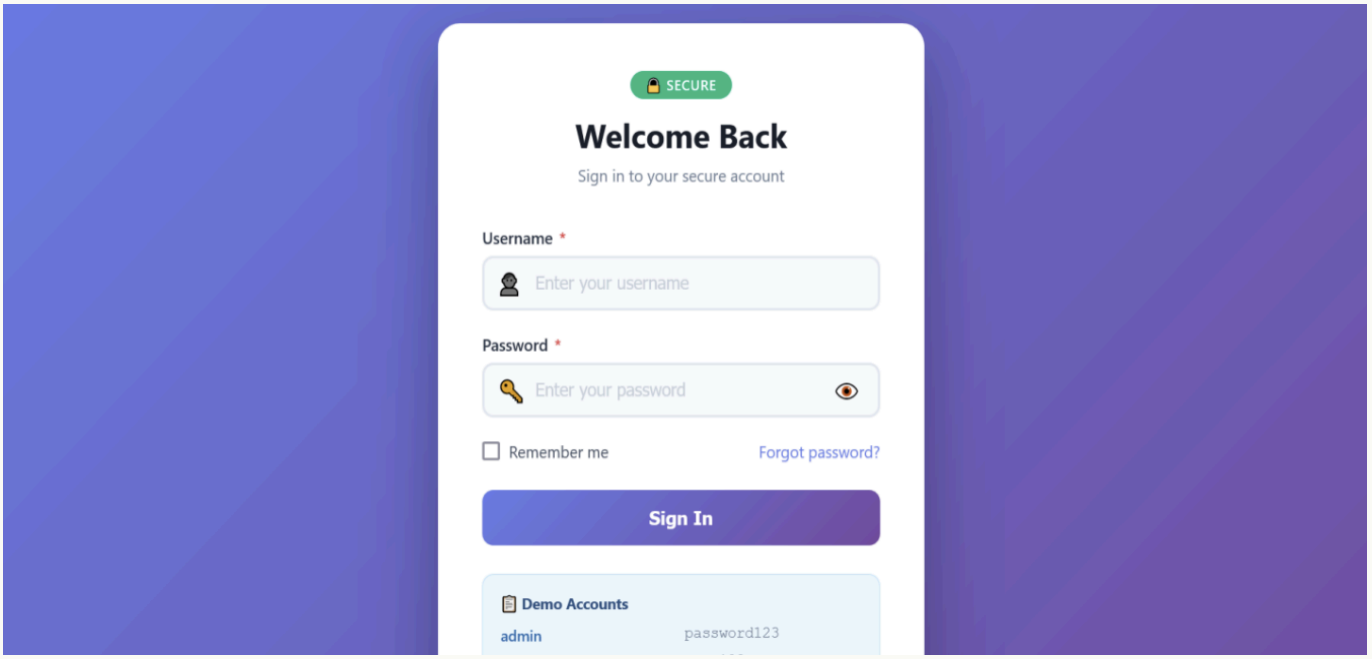




Vulnerable Login



Secured Login



APPENDIX A: COMPLETE CODE IMPLEMENTATION

The complete source code for both vulnerable and secure versions, along with the automated testing suite, is available in the project repository.

Repository Details	Information
Repository Name	Web Security Research
GitHub URL	https://github.com/cycyberuk/WebShellScanner
Version	v2.0
License	MIT License
Language	PHP 8.3.4
Last Updated	July 2026