|  |
| --- |
| FutureInterns |
| 📈 Penetration Testing Report: DVWA Vulnerability Assessment |
|  |

|  |
| --- |
| Jagadeesh Kommineni  7-26-2025 |

**📈 Penetration Testing Report: DVWA Vulnerability Assessment**

**Analyst:** Jagadeesh Kommineni  
**Role:** CyberSecurity Intern  
**Environment:** DVWA (Damn Vulnerable Web Application)  
**OS:** Kali Linux  
**Tools Used:** Burp Suite, OWASP ZAP, Apache2, MySQL, curl, Chromium, DVWA  
**Date:** 26/07/25

## 🔧 Environment Setup

To replicate a controlled offensive security environment, DVWA was deployed within a Kali Linux VM. This facilitated manual vulnerability discovery and automated scanning techniques. Apache2 and MySQL services were configured alongside necessary PHP modules (e.g., php-mysqli, php-gd).

* Application URL: http://localhost/dvwa
* Web Server: Apache/2.4 on Kali Linux
* Database Server: MySQL
* Security level: **Low** (for demonstrative exploitation)
* Configured config.inc.php from .dist template with correct DB credentials

## 🛡️ Vulnerability Testing Summary

This section outlines the technical exploitation of core web application vulnerabilities as per OWASP Top 10 guidelines. Each vulnerability includes PoC (Proof of Concept), analysis, and defensive security controls.

### ⚡ 1. Command Injection

**Objective:** Exploit improper input validation to execute arbitrary shell commands.

**Steps:** - Accessed endpoint: /dvwa/vulnerabilities/exec/ - Input payload: 127.0.0.1 && whoami - Returned output: www-data, confirming shell execution in server context.

**Technical Insight:** Unescaped input passed to shell\_exec() or system() functions leads to RCE (Remote Command Execution).

**Countermeasures:** - Employ escapeshellcmd() and strict whitelisting - Avoid direct shell interaction; use secure alternatives (e.g., APIs) - Implement WAF rules for command signature detection

### 🔢 2. SQL Injection

**Objective:** Extract sensitive data through injection in unsanitized SQL queries.

**Steps:** - URL endpoint: /dvwa/vulnerabilities/sqli/ - Payload: 1' OR '1'='1 - Server response revealed full user credential table

**Technical Insight:** Input is directly concatenated into a dynamic SQL query, exposing the backend to exploitation.

**Mitigation:** - Use parameterized queries (mysqli\_stmt, PDO) to prevent injection - Employ least privilege access to DB accounts - Enable detailed DB logging for anomaly detection

### 🚩 3. Stored XSS (Cross-Site Scripting)

**Objective:** Persist malicious JavaScript to execute in the browser of any user viewing stored data.

**Steps:** - Access: Guestbook form with Name/Message fields - Payload: <script>alert('XSS')</script> - Execution: Alert triggered upon loading stored entry

**Technical Insight:** User input is stored in DB and injected into the DOM without sanitization or context-sensitive encoding.

**Defenses:** - Apply output encoding using htmlspecialchars() - Enable Content Security Policy (CSP) - Validate and sanitize on both client and server side

### 🚩 4. Reflected XSS

**Objective:** Execute injected JS reflected via unsanitized query parameters.

**Steps:** - Injected payload: <script>alert('XSS')</script> into URL/form - Immediate alert on submission confirmed execution

**Technical Insight:** Reflected XSS commonly occurs in search forms, error messages, or feedback endpoints.

**Recommendations:** - Apply input validation and output encoding in response context - Restrict inline scripting via CSP headers - Leverage libraries like DOMPurify to sanitize inputs

### 🗃️ 5. Unrestricted File Upload (RCE)

**Objective:** Upload a malicious payload (PHP web shell) and achieve remote code execution.

**Steps:** - File uploaded: shell.php via upload interface - URL accessed: /dvwa/hackable/uploads/shell.php?cmd=whoami - Output confirmed: www-data

**Technical Detail:** Lack of MIME type checking and absence of server-side extension filtering enabled arbitrary code execution.

**Hardening Measures:** - Allow only whitelisted file extensions (e.g., jpg, png) - Store uploads outside the web root - Disable exec() and related functions via PHP config

### 🔐 6. Cross-Site Request Forgery (CSRF)

**Objective:** Craft an auto-submitting form that changes a user’s password without their consent.

**Steps:** - Created external HTML (trick.html) with hidden POST form to /dvwa/vulnerabilities/csrf/ - While authenticated, opened HTML file triggering automatic submission - Password did not change (due to lack of active session or CSRF token verification)

**Analysis:** Modern protections like SameSite cookies and missing session handling limited exploitability.

**Security Practices:** - Implement synchronizer tokens (CSRF-Tokens) - Use SameSite=Strict and HttpOnly cookie flags - Validate origin and referer headers on sensitive actions

### 🤔 7. Brute Force

**Objective:** Systematically guess credentials using automated tools.

**Steps:** - Tool: Burp Suite Intruder - Target: /dvwa/vulnerabilities/brute/ - Payloads: Common usernames (admin, user), weak passwords (1234, password) - Successful login with admin:password

**Findings:** The server response length changed upon successful login, aiding detection.

**Recommendations:** - Implement account lockout or CAPTCHA after failed attempts - Log IP address and device metadata for anomaly detection - Enforce MFA (Multi-Factor Authentication)

## 📊 Risk Matrix

| Vulnerability | OWASP Category | CVSS Estimate | Exploited? |
| --- | --- | --- | --- |
| Command Injection | A1: Injection | 9.8 Critical | ✅ |
| SQL Injection | A1: Injection | 9.0 High | ✅ |
| Stored XSS | A7: XSS | 6.1 Medium | ✅ |
| Reflected XSS | A7: XSS | 5.4 Medium | ✅ |
| File Upload (RCE) | A8: Insecure Deserialization | 10.0 Critical | ✅ |
| CSRF | A5: Broken Access Control | 4.3 Low | ❌ |
| Brute Force | A2: Broken Auth | 7.5 High | ✅ |

## 🔺 Strategic Remediation Plan

* Enforce **input validation and output encoding** on all user-controlled fields
* Implement **prepared statements** across all database operations
* Use **WAFs** (e.g., ModSecurity) to block common exploit patterns
* Configure **secure headers** (X-XSS-Protection, X-Frame-Options, Content-Security-Policy)
* Enforce **role-based access control** and use **principle of least privilege**
* Audit and patch software dependencies regularly

## 📅 Conclusion

This DVWA assessment demonstrates critical gaps that exist when secure coding standards are not enforced. By leveraging both manual and automated testing techniques, this report simulates real-world adversarial behaviors across OWASP Top 10 vulnerabilities. Each exploit provides insight into insecure application logic and underscores the importance of defense-in-depth, secure configuration, and application hardening.

Future work should include elevating DVWA security levels and integrating advanced tools like sqlmap, nikto, and Metasploit for deeper exploration.

Screebshots































