Security Assessment Report: SQL Injection Vulnerability in DVWA Application

Prepared by: Ikechukwu Justin Onyekwere; Cybersecurity Analyst

Date: September 29, 2025

Client: [Company Name] Stakeholders

Report Reference: SEC-2025-0929-DVWA-SQLI

Executive Summary

Dear Stakeholders,

As a Cybersecurity Analyst in penetration testing and vulnerability assessments, I have conducted a targeted analysis of the Damn Vulnerable Web Application (DVWA) instance running on your local development environment (localhost:8080). This assessment focused on the SQL Injection vulnerability module, using the open-source tool SQLmap to simulate real-world attack scenarios.

The results reveal critical **SQL injection vulnerabilities** in the GET parameter `id` of the endpoint `/vulnerabilities/sqli/`. These flaws allow unauthorized access to database information, potentially leading to data breaches, unauthorized data manipulation, or full system compromise. While the `Submit` parameter was tested and found non-vulnerable, the `id` parameter supports multiple injection techniques, including boolean-based blind, error-based, time-based blind, and UNION queries.

Although data extraction from the `users` table in the `security` database was unsuccessful due to environmental limitations (e.g., permission restrictions), this does not diminish the severity that the vulnerabilities are exploitable in principle and could be leveraged by attackers in a production-like setting.

Key Risks: High potential for sensitive data exposure (e.g., user credentials), compliance violations (e.g., GDPR, HIPAA if applicable), and reputational damage. Immediate remediation is recommended to mitigate these threats.

This report **(Attachment in the mail)** details the methodology, findings, risks, and actionable recommendations. I am available for a follow-up discussion to address any questions.

Best regards,

Ikechukwu Justin Onyekwere

Table of Contents

1.0 Introduction and Scope

This assessment was initiated to evaluate the security posture of the DVWA application, a deliberately vulnerable web app used for training and testing purposes. This configuration simulates a poorly secured web application, common in legacy systems or misconfigured environments. The primary objective was to identify injectable parameters, confirm the backend database, and attempt data extraction from sensitive tables (e.g., `users` in the `security` database). The assessment aligns with industry standards such as OWASP Top 10 (A03:2021 - Injection) and NIST SP 800-115 guidelines for technical security testing.

1.1 Assumptions and Limitations

- The environment is a controlled, local setup (Docker/Kali Linux on Debian 9 with Apache 2.4.25 and MySQL/MariaDB).
- No production data was at risk; this was a simulated test.
- The scan generated 6,907 HTTP requests, which could trigger alerts in a monitored environment.

1.2 Methodology

The assessment utilized SQLmap (version 1.9.6), an automated SQL injection and database takeover tool widely used in ethical hacking. SQLmap systematically probes for vulnerabilities by injecting payloads and analyzing responses for anomalies (e.g., errors, delays, or content changes).

Steps Followed

1. Connection and Stability Check: Verified the target's accessibility and response consistency.
2. Parameter Analysis: Tested GET parameters (`id` and `Submit`) for dynamic behavior and injectability.
3. Vulnerability Detection: Employed heuristic tests, followed by technique-specific probes (boolean-based, error-based, time-based, and UNION queries). Extended testing to MySQL-specific payloads based on DBMS fingerprinting.

4. 4.Exploitation Attempt: Once vulnerabilities were confirmed, attempted to enumerate and dump data from the specified table (`users` in `security` database).
5. Fallback Mechanisms: When full techniques failed, switched to partial methods and common column checks.

The scan was conducted with user inputs to include comprehensive MySQL tests and proceed despite warnings (e.g., non-dynamic parameters).

2.1 Detailed Findings

Environment and DBMS Identification

- Web Stack: Linux Debian 9 (stretch), Apache 2.4.25.
- Backend DBMS: MySQL >= 5.0 (MariaDB fork), confirmed via fingerprinting (e.g., error patterns and query responses).
- Additional Observations: Potential cross-site scripting (XSS) in `id` parameter, where inputs are reflected without sanitization; though not the focus, this compounds risks.

Vulnerable Parameters

- id (GET Parameter): Confirmed vulnerable to multiple SQL injection types. This parameter is likely used in unsanitised database queries (e.g., `SELECT * FROM table WHERE id = '$id'`), allowing attackers to append malicious SQL code.
- Submit (GET Parameter): Tested extensively (including boolean, error, time-based, and UNION variants across DBMS like PostgreSQL, MSSQL, Oracle). No vulnerabilities detected; deemed safe.

2.2 Identified Injection Techniques

The following techniques were successfully validated, each with example payloads. These allow attackers to bypass filters, extract data, or disrupt operations:

1. Boolean-Based Blind Injection:

- Description: Exploits true/false conditions to infer data bit-by-bit based on response differences (e.g., page content changes).

- Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL comment).
- Payload Example: `id=15' OR NOT 9909=9909#&Submit=Submit`.
- Implications: Stealthy; useful for extracting data without visible errors.

2. Error-Based Injection:

- Description: Forces database errors to leak information (e.g., via duplicate entries or type mismatches).
- Title: MySQL >= 5.0 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR).
- Implications: Fast data leakage if error messages are displayed; common in debug modes.

3. Time-Based Blind Injection:

- Description: Introduces delays (e.g., using SLEEP) to deduce data from response timings.
- Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP).
- Payload Example: `id=15' AND (SELECT 7090 FROM (SELECT(SLEEP(5)))VIxP)--WDVX&Submit=Submit`.
- Implications: Effective against output-filtered apps; harder to detect without monitoring.

4. UNION Query Injection:

- Description: Appends attacker-controlled results to the original query, extracting data directly.
- Title: MySQL UNION query (NULL) - 2 columns.
- Implications: Direct access to database contents; confirmed 2 columns in the query.

2.3 Data Extraction Attempt

Targeted the `users` table in the `security` database, which likely contains sensitive information (e.g., usernames, passwords).

1. Challenges Encountered:

- Full UNION technique failed due to potential row limits.
- Partial UNION fallback was attempted, but column names could not be enumerated.
- Common column check (using a wordlist like `id`, `name`, `password`) returned no matches.
- Permission issues detected (`'command denied'`), possibly from restricted DBMS privileges or containerized environment constraints.

Outcome: No data dumped, but this is not indicative of security—the vulnerabilities remain exploitable with refined techniques (e.g., using `--no-cast` or `--hex` switches).

3.1 Scan Metrics

- Total Requests: 6,907.
- Duration:13 minutes.
- Warnings: Reflective values (potential XSS), single-thread slowness, and data retrieval issues. Suggested mitigations include ignoring cookies or hex-encoding.

3.2 Risk Assessment and Business Implications

1. Severity Rating: Critical (CVSS Score Estimate: 9.8/10 – High impact on confidentiality, integrity, and availability).

- Potential Exploits:
- Data Theft: Extraction of user credentials, leading to account takeovers.
- Database Manipulation: Insertion/deletion of records (e.g., via stacked queries).
- Escalation: Chaining with XSS for broader attacks (e.g., session hijacking).

2. Business Risks:

- Financial: Breach costs (e.g., forensics, notifications) averaging $4.45M per incident (per 2025 Ponemon Institute data).
- Regulatory: Violations of data protection laws, risking fines (e.g., up to 4% of global revenue under GDPR).
- Reputational: Loss of customer trust if user data is exposed.
- Operationa: Downtime from denial-of-service via time-based delays.
- Threat Actors: Script kiddies (using tools like SQLmap), insiders, or advanced persistent threats targeting intellectual property.

In your development pipeline, these issues highlight gaps in secure coding practices, potentially propagating to production if not addressed.

3.3 Recommendations

The recommendations draw from established best practices, emphasizing prevention over detection. The goal is to eliminate root causes (e.g., unsanitized inputs) while implementing layered defenses. This approach aligns with OWASP guidelines and recent 2025 insights on evolving threats, such as AI-assisted injection attacks.

1. **Start by isolating and mitigating exposure to prevent exploitation:**

- Disable or Restrict Vulnerable Endpoints: Temporarily take the affected page offline or restrict access (e.g., via IP whitelisting). In DVWA's case, increase the security level to "high" for testing, but in production, apply access controls.
- Deploy Quick Fixes: Use input sanitization libraries (e.g., PHP's mysqli_real_escape_string) as a stopgap, though this is not foolproof. Monitor logs for anomalous queries using tools like Fail2Ban.

- Scan for Similar Issues: Run automated tools (e.g., SQLmap or ZAP) across all endpoints to identify other injectable parameters.

2. **Address the core vulnerability through secure coding:**

- Adopt Parameterized Queries: Replace dynamic SQL with prepared statements or parameterized queries (e.g., using PDO in PHP: $stmt = $db->prepare("SELECT * FROM users WHERE id = ?"); $stmt->execute([$id]);). This binds inputs as data, not code, preventing injection.
- Input Validation and Sanitization: Enforce strict whitelisting (e.g., ensure id is numeric via is_numeric() or regex). Reject invalid inputs early.
- Stored Procedures: Migrate queries to stored procedures, which encapsulate logic and reduce attack surfaces.
- Error Handling: Suppress detailed error messages (e.g., disable MySQL error display) to avoid leaking information during error-based attacks.

3. **Build defense-in-depth to handle future threats:**

- Implement a Web Application Firewall (WAF): Deploy tools like ModSecurity or Cloudflare WAF to block common SQLi payloads (e.g., UNION, SLEEP). Configure rules for MySQL-specific patterns.
- Least Privilege Principle: Create database users with minimal permissions (e.g., SELECT-only for read operations). Regularly audit and rotate credentials.
- Keep the Stack Updated: Patch MySQL/MariaDB, Apache, and PHP to the latest versions (e.g., address CVEs like CVE-2025-1094 in PostgreSQL analogs for MySQL).
- API Gateway for Modern Apps: If applicable, use an API gateway (e.g., Zuplo) to centralize input validation and rate limiting.

4. **Sustain security through proactive measures:**

- **Vulnerability Scanning**: Integrate automated scans (e.g., OWASP ZAP in CI/CD) and manual reviews. Test for emerging threats like AI-generated payloads.
- **Incident Response Plan**: Develop protocols for SQLi detection (e.g., via SIEM tools like Splunk) and response, including forensic analysis.
- **Developer Training**: Conduct workshops on secure coding (e.g., OWASP Top 10). Encourage code reviews focusing on injection risks.
- **Third-Party Audit**: Engage external experts annually to validate fixes.

Conclusion

This assessment underscores the dangers of unsanitized user inputs in web applications, as demonstrated by the exploitable SQL injection in DVWA. While the data dump failed, the confirmed vulnerabilities pose a clear and present threat that could be realized in a less restricted environment. By implementing the recommended measures, [Company Name] can significantly strengthen its security posture and protect valuable assets.

I recommend scheduling a debrief session to prioritize these actions. Please contact me at justinikechukwu19@gmail.com for further assistance.