Security Assessment Report

Project: Web Application Vulnerability Assessment
Task: SQL Injection Testing (DVWA – Low, Medium, High Levels)
Intern: Abdelalim SAADA
Program: Cyber Security Internship – Future Interns
Date: August 2025

Executive Summary

This report documents a structured SQL Injection (SQLi) test conducted on the Damn Vulnerable Web Application (DVWA). The objective was to assess the effectiveness of security mechanisms implemented at different difficulty levels — **Low**, **Medium**, and **High** — and to provide detailed findings, analysis, and recommendations for securing web applications against injection attacks.

Test Environment

Component	Details
Target Application	DVWA (Damn Vulnerable Web App)
Hosting	Local VMware VM (`192.168.100.96`)
Testing Platform	Kali Linux
Tools Used	Browser, Burp Suite, SQLMap
Authentication	DVWA login (admin / password)
Security Levels	Low, Medium, High

Tools Used

- **Browser** – Manual injection via URL parameters
- **Burp Suite Community Edition** – HTTP interception and manipulation
- **SQLMap** – Automated SQL Injection testing
- **Linux Shell** – Command-line testing and data logging

Test Cases Summary

Level	Injection Result	Payload Example	Risk Level	Bypass Method	Protection Mechanism
Low	Successful	`' OR '1'='1`	Critical	None (direct input)	No protection
Medium	Successful	`0 or 1=1`	High	Logic-only injection	addslashes(), no quotes
High	Blocked	`1' OR '1'='1`, `0 or 1=1`	Low	None (sanitized)	Escaping + numeric check

Detailed Findings

Level 1 – Low Security

Payload Used:

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

Result:
Returned multiple user records. Classic SQL Injection worked due to direct embedding of user input in SQL.

Technical Explanation:
Input is inserted directly into the SQL query without sanitization:

```
SELECT first_name, last_name FROM users WHERE user_id = '1' OR '1'='1';
```

Impact:

- Full data exposure
- Possibility of further exploitation (e.g., UNION-based injection, authentication bypass)

Recommendations:

- Use **prepared statements** (parameterized queries)
- Apply **server-side input validation**
- Avoid building SQL queries from raw input

Level 2 – Medium Security

Failed Payloads:

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

Successful Payload:

```
0 or 1=1
```

Result:
Logic-only injection worked. Escaping of quotes prevented basic SQLi, but logic injection succeeded.

Explanation:
PHP code uses `addslashes()` to escape `'`, `"`, etc., but logic-only payloads avoid quotes entirely:

```
SELECT first_name, last_name FROM users WHERE user_id = 0 or 1=1;
```

Impact:

- Partial bypass of sanitization
- Attacker still extracts full user records

OWASP ID	Title	Affected Levels
A03:2021	Injection	Low, Medium
Mitigated		High

Recommendations:

- Use **strict type checking** and **prepared statements**
- Avoid logic operators in numeric fields
- Apply **whitelisting** instead of blacklisting

Level 3 – High Security

Payloads Attempted:

```
1' OR '1'='1
0 or 1=1
```

Tool:
SQLMap command:

```
sqlmap -u "http://192.168.100.96/vulnerabilities/sqli/?id=1&Submit=Submit#"
\
--cookie="security=high; PHPSESSID=..." \
--level=5 --risk=3 --threads=5 --dump
```

Result:

- All injections blocked
- SQLMap failed to identify injectable parameters
- Manual injection returned error: `Invalid SQL syntax`

Source Code Review:

```
$id = stripslashes($id);
$id = mysql_real_escape_string($id);
if (is_numeric($id)) {
    ...
}
```

Security Features Observed:

- Input escaping (`mysql_real_escape_string`)
- Removal of backslashes (`stripslashes`)
- Whitelisting: only numeric input is accepted (`is_numeric`)

Impact:
No injection possible. Input sanitized, logic rejected, and dangerous characters neutralized.

Recommendations:

- Maintain this multi-layered approach
- Replace deprecated `mysql_*` with PDO or MySQLi
- Monitor logs for suspicious patterns

General Recommendations

1. **Use Parameterized Queries Everywhere**
   Avoid string-based SQL. Prepared statements are the industry standard.
2. **Enforce Input Types Strictly**
   Use `is_numeric()` or casting on numeric fields.
3. **Log Suspicious Behavior**
   Monitor failed input attempts for signs of tampering.
4. **Escape & Sanitize All Inputs**
   Apply consistent server-side sanitization for all user data.

Conclusion

This assessment demonstrates the varying levels of SQL injection protection in DVWA. The exercise highlights the importance of **defense-in-depth**, particularly the need for **input validation**, **prepared statements**, and **error suppression**.

- **Low and Medium** levels were vulnerable and exploited.
- **High** level was resistant due to strong input control.
- Findings are documented, and remediation is proposed for each level.