

Lab 5: Mitigation of SQL Injection

Shivali Mate

I. INTRODUCTION

THE The lab focused on understanding both exploitation and mitigation strategies for injection flaws that commonly affect web applications. Using deliberately vulnerable training applications (WebGoat v8 & v7) the attacks are recreated to observe their impact, then applied developer-side mitigations to see how they prevent or reduce risk.

II. METHODOLOGY

Mitigation of SQL injection means using development and deployment practices that keep attacker-controlled input from being interpreted as SQL. A web developer can prevent injection by avoiding string concatenation of user input, using prepared/parameterized statements or safe DB APIs, validating or whitelisting inputs that affect structure, running the DB with least privilege, and avoiding dynamic SQL where possible.

Parameterized queries mitigate injection because the driver sends a fixed SQL template with placeholders while bound values are treated strictly as data and never parsed as SQL tokens. Parameterization does not protect when untrusted input is used to build SQL structure like table/column names, keywords, or fragments or other context-aware controls are required in addition to prepared statements.

A. SQL Injection (Mitigation)

This section documents the four mitigation exercises carried out in WebGoat 2023.8.

1) *Writing Safe Code 1*: The first exercise required completing a Java snippet that had originally concatenated user input into a SQL string. The concatenation is replaced with a prepared statement pattern: establishing connection getConnection(...) Connection.prepareStatement(?,?), bind calls such as statement.setString(1, username).

2) *Writing Safe Code 2*: The second exercise is to write a small Java program that opens a JDBC connection inside a try block, prepares a parameterized query like `SELECT * FROM users WHERE username = ?, [1]` binds the string parameter with `setString`, and `[2]` executes the query and handles exceptions in a catch block. The Fig. 1 shows the full Java code, the successful execution, and confirms immunity to classic injection payloads because the user input was only ever passed as a bound parameter

3) *Input Validation 1*: This exercise showed that naive input filters (for example, “remove spaces” or “strip punctuation”) can be bypassed. The attack is tested that retrieves password values from `user_system_data` by exploiting comment and whitespace parsing quirks. An example of trailing SQL comment operator to corrupt the intended statement:

```
select/**/**/from/**/user_system_data;--
```

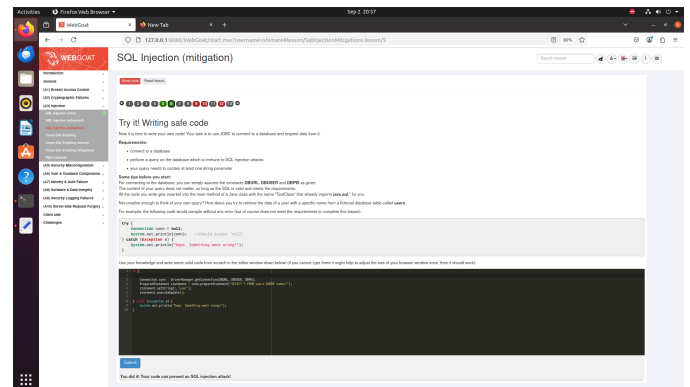


Fig. 1. Mitigation of SQL Injection: Writing Safe Code

4) *Input Validation 2*: In this exercise WebGoat’s HyperSQL (HSQLDB) parser acceptance of obfuscated keywords was demonstrated. In this case nested/obfuscated keywords and HSQLDB comment constructs can be used to evade non-recursive validators.

```
a';//seselectlect/**//frfromom//user_system_data;--
```

Fig. 2 shows the query being accepted and the resulting data.

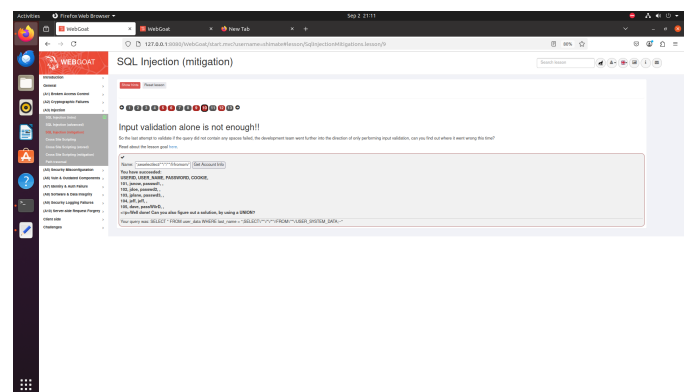


Fig. 2. Mitigation of SQL Injection: Input Validation

B. Command Injection

Command injection happens when an application places untrusted input into an OS command or script context, allowing an attacker to change the intended command flow; shell metacharacters such as `;`, `&&`, — let an attacker append or chain additional commands. The attempted injection payload

```
”;netstat -a;ifconfig;#
```

appends `netstat -a` and `ifconfig` to the original command via shell separators. The lab limited to only allowed commands like `netstat -a`, `dir`, `ls`, `ifconfig`, and `ipconfig`; Fig. 3 shows the server output. Note that `/bin/sh -c` is the usual way to invoke a new POSIX shell to execute a command string: the

-c option tells /bin/sh to read and run the following string as a shell command, which is exactly how chained payloads get interpreted by the OS.

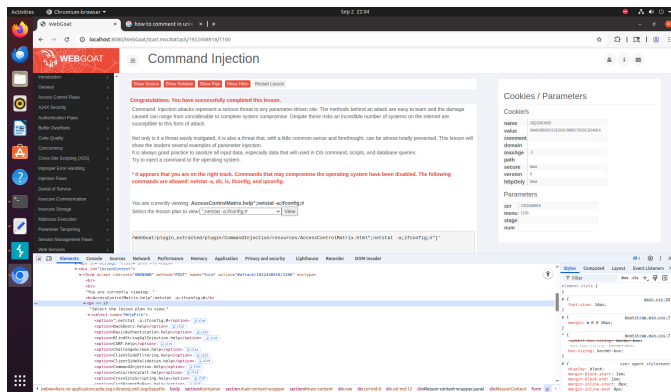


Fig. 3. Command Injection

C. Log Spoofing

Log spoofing means injecting characters into inputs so the resulting log file contains forged or misleading entries (for example, a line that appears to show admin successfully logged in). In the exercise percent-encoding for CR/LF to insert a new line into logs is used.

Smith%0d%0aLogin

Here %0d represents carriage return (CR) and %0a represents line feed (LF); when the server decodes these sequences into raw bytes, they create a new log line. The goal was to make the log show a username such as admin as having successfully logged in. Fig. 4 displays the crafted request and the resulting log snippet.

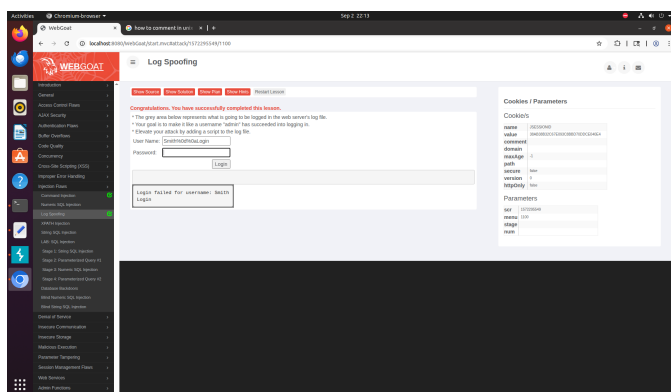


Fig. 4. Log Spoofing

D. XPATH Injection

XPath injection targets code that builds XPath queries from user input to select nodes in XML documents (for example an employee directory stored in XML). The exercise was to retrieve other employees' data by injecting boolean tautologies into the username/password fields. A typical payload used was:

Smith' or 1=1 or 'a'='a

The or 1=1 clause forces the username condition true, and the subsequent or 'a'='a' causes the password check to

evaluate true regardless of the supplied password. Fig. 5 shows the form input and the returned employee records.

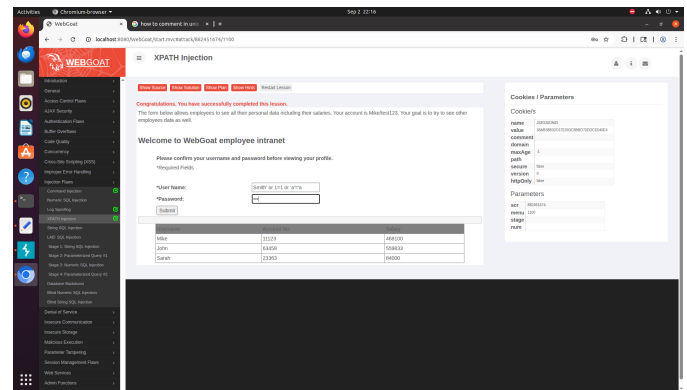


Fig. 5. XPATH Injection

III. TOOLS AND LEARNING EXPERIENCE

A Java/JDBC environment was used to implement and test prepared statements, and basic OS networking utilities to validate command-injection effects. WebGoat's exercises and the VM made it easy to reproduce attacks safely and observe results in real time. A limitation is that some bypass techniques relied on HSQLDB-specific parser behaviors that may not generalize to production databases. The labs also presumed Java/JDBC familiarity, and the log-spoofing exercise highlighted that many log viewers fail to escape entries, creating audit and XSS risks.

IV. FINDINGS AND DISCUSSION

The lab reinforced that parameterized queries are the primary defensive control against SQL injection because they separate code from data. However, the exercises also showed real limitations: parameterization cannot protect dynamic SQL structure and naive validators can be bypassed via obfuscation. Command injection and XPath injection interpolate user input into interpreter contexts, allows attackers to change semantics. Log spoofing demonstrated an operational risk: unescaped log entries can mislead administrators or become XSS vectors when logs are surfaced in web dashboards.

V. CONCLUSION

The lab demonstrates that while parameterized queries are highly effective for preventing classical SQL injection, security requires layered controls and operational hygiene. Developers should prefer safe APIs that avoid SQL composition, apply strict whitelisting for any dynamic identifiers, validate inputs in a context-aware way, and ensure logs and administrative interfaces properly escape untrusted content.

REFERENCES

- [1] OWASP Injection: <https://owasp.org/www-community/Injection>
- [2] XPath syntax: https://www.w3schools.com/xml/xpath_syntax.asp
- [3] XPath tutorial: https://www.w3schools.com/xml/xml_xpath.asp
- [4] XPath Recommendation: <https://www.w3.org/TR/2017/REC-xpath-31-20170321/>
- [5] HyperSQL (HSQLDB) official site / documentation: <https://hsqldb.org/>
- [6] Oracle Java SE java.sql.PreparedStatement API documentation: <https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html>