**Secure Coding Review**

**SQL Injection Vulnerability**

Project: Basic Python Application (Simulated Database Interaction)

Date: July 25, 2025

**1. Executive Summary**

This review focuses on a critical web application security vulnerability: SQL Injection. Through a simplified Python application, we demonstrated how insecure coding practices can lead to data breaches, unauthorized access, or data manipulation. The review highlights the importance of using secure coding techniques, specifically parameterized queries, to mitigate this threat.

**2. Vulnerability Identified: SQL Injection**

**Description**

SQL Injection (SQLi) is a code injection technique used to attack data-driven applications, in which malicious SQL statements are inserted into an entry field for execution (e.g., to dump database content to the attacker). It occurs when user-supplied input is directly concatenated into a SQL query without proper sanitization or escaping.

**Impact**

If exploited, SQL Injection can lead to:

* Data Breach: Unauthorized access to sensitive information (e.g., user credentials, personal data).
* Data Manipulation/Corruption: Modification or deletion of database records.
* Authentication Bypass: Gaining access to user accounts or administrative panels without valid credentials.
* Remote Code Execution: In some severe cases, an attacker might be able to execute arbitrary commands on the server.

**Vulnerable Code Snippet (Conceptual Example from** vulnerable\_app.py**)**

# Insecure: Directly concatenating user input

username = user\_input # e.g., "' OR 1=1; --"

query = f"SELECT \* FROM users WHERE username = '{username}' AND password = 'some\_password';"

# Database executes: SELECT \* FROM users WHERE username = '' OR 1=1; --' AND password = 'some\_password';

# The 'OR 1=1' becomes part of the SQL logic, always evaluating to true.

**Demonstration of Attack**

Using the vulnerable code, an attacker could input payloads like:

* ' OR 1=1; -- (to bypass authentication)
* '; DROP TABLE users; -- (to delete data)

These inputs manipulate the SQL query's logic, leading to unintended and malicious database operations.

**3. Remediation and Secure Coding Best Practices**

**Recommended Fix: Parameterized Queries (Prepared Statements)**

The most effective and widely recommended method to prevent SQL Injection is to use Parameterized Queries (also known as Prepared Statements). This technique separates the SQL code from the user-supplied data.

**Secure Code Snippet (Example from** app.py**)**

# Secure: Using a placeholder and passing parameters separately

username\_param = user\_input # e.g., "' OR 1=1; --"

query\_template = "SELECT \* FROM users WHERE username = ? AND password = 'some\_password';"

parameters = (username\_param,) # Data passed as a separate parameter

# Database driver safely handles the 'username\_param' as literal data,

# not as part of the SQL command.

**Explanation of the Fix**

With parameterized queries, the database engine first parses the SQL query structure (with placeholders) and then, separately, binds the user-provided values to those placeholders. This ensures that any special characters in the input are treated as literal data, not as executable SQL commands, effectively neutralizing injection attempts.

**General Secure Coding Recommendations**

1. Always Use Parameterized Queries: This is the golden rule for any database interaction involving user input. Never concatenate user input directly into SQL strings.
2. Input Validation: While parameterized queries prevent SQL injection, validating user input (e.g., checking for expected data types, lengths, or formats) is still a good practice for overall application robustness and to prevent other types of attacks.
3. Principle of Least Privilege: Database users should only have the minimum necessary permissions required for their tasks. For example, a web application user should not have permissions to drop tables.
4. Error Handling: Avoid displaying verbose database error messages to users, as these can provide attackers with valuable information about your database structure.
5. Keep Software Updated: Regularly update your database systems, programming languages, and associated libraries to patch known vulnerabilities.
6. Security Audits and Testing: Regularly perform code reviews, static analysis (SAST), dynamic analysis (DAST), and penetration testing to identify and remediate vulnerabilities.

**4. Conclusion**

Understanding and preventing SQL Injection is fundamental to building secure applications. By adopting practices like parameterized queries, developers can significantly reduce the risk of this prevalent and dangerous vulnerability. This review serves as a foundational step in building a secure coding mindset.