# Case 1: Addressing and Resolving a Code Vulnerability

## Introduction

## Tasks

This document will complete the following tasks:

1. Select any one code finding identified by Semgrep and take a screenshot to show the status of the finding
   * ☒ select the [tainted-sql-string] rule and fix all occurances found in the bad-python-app code base
   * <screenshot-tainted-sql-string-2025-05-27.png>
2. Analysis and Explanation

### a. Explain the Finding

* ☒ Detail what the vulnerability is and why Semgrep’s rule flagged it as a potential security issue. Include an understanding of the rule’s logic and the specific code pattern that triggered the finding.

### b. Impact Analysis

* ☒ Discuss the potential impact or risks associated with the vulnerability if left unaddressed.

## Vulnerabilities Identified

### 1. SQL Injection in Login Functionality

**File**: vulns/sql\_injection/sql\_injection\_login.py

**Vulnerable Code**:

username = form.get('username')  
password = form.get('password')  
password\_hash = \_hash\_password(password)  
  
# Vulnerable SQL query with string formatting  
sql = f"SELECT \* FROM users WHERE username='{username}' AND password='{password\_hash}'"  
db\_result = app.db\_helper.execute\_read(sql)

**Risk**:

* Attackers could bypass authentication using SQL injection techniques
* Potential for complete database compromise
* Exposure of sensitive user data

### 2. SQL Injection in Search Functionality

**File**: vulns/sql\_injection/sql\_injection\_search.py

**Vulnerable Code**:

search = request.args.get('q')  
  
# Vulnerable SQL query with string formatting  
sql = f"SELECT \* FROM products WHERE name LIKE '%{search}%'"  
db\_result = app.db\_helper.execute\_read(sql)

**Risk**:

* Unauthorized data access
* Potential for data exfiltration
* Database structure enumeration

## Remediation

### 1. Login Functionality Fix

**File**: vulns/sql\_injection/sql\_injection\_login.py

**Changes Made**:

# Before (vulnerable)  
sql = f"SELECT \* FROM users WHERE username='{username}' AND password='{password\_hash}'"  
db\_result = app.db\_helper.execute\_read(sql)  
  
# After (fixed)  
sql = "SELECT \* FROM users WHERE username = :username AND password = :password\_hash"  
db\_result = app.db\_helper.execute\_read(sql, {'username': username, 'password\_hash': password\_hash})

### 2. Search Functionality Fix

**File**: vulns/sql\_injection/sql\_injection\_search.py

**Changes Made**:

# Before (vulnerable)  
search = request.args.get('q')  
sql = f"SELECT \* FROM products WHERE name LIKE '%{search}%'"  
db\_result = app.db\_helper.execute\_read(sql)

# After (fixed)

# 1. Parameterized query to prevent SQL injection

# 2. Escaped special characters in LIKE pattern  
search = request.args.get('q', '')  
sql = "SELECT \* FROM products WHERE name LIKE :search\_pattern"  
db\_result = app.db\_helper.execute\_read(sql, {'search\_pattern': f"%{search}%"})

## Technical Explanation

### Why the Fix Works

1. **Named Parameters**:
   * For example, the :name placeholder allows for dictionary-based parameterization.
   * The database engine handles proper escaping of special characters
   * Prevents SQL injection by separating SQL logic from data
   * [Flask Documentation: Query Parameters](https://flask.palletsprojects.com/en/2.0.x/quickstart/" \l "accessing-request-data)
   * [SQLAlchemy Documentation: Querying](https://docs.sqlalchemy.org/en/14/tutorial.html" \l "querying)
2. **Defense in Depth**:
   * Input validation is still recommended as an additional layer of security.
   * The existing password hashing remains in place for security
   * [OWASP Input Validation Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html)

### Security Benefits

* **Prevents SQL Injection**: User input is never directly executed as SQL
* **Improved Performance**: Parameterized queries can be cached by the database
* **Code Maintainability**: Clearer separation between SQL logic and data
* **Future-Proof**: Works with any compliant SQL database

## Testing

To verify the fixes:

1. **Login Page**:
   * Try entering ' OR '1'='1 in the username field
   * The query should fail safely without logging in
   * Check server logs for any SQL errors (there should be none)
2. **Search Functionality**:
   * Test with special characters: %, \_, ', ;, --
   * The search should handle these as literal characters, not SQL commands
   * No database errors should be exposed

## Conclusion

The implemented changes effectively mitigate the SQL injection vulnerabilities by using parameterized queries throughout the application. This approach follows security best practices and provides robust protection against SQL injection attacks while maintaining the application’s functionality.

## Additional Recommendations

1. Implement input validation for all user-supplied data
2. Use a web application firewall (WAF) for additional protection
3. Regularly update dependencies to patch known vulnerabilities
4. Implement rate limiting to prevent brute force attacks
5. Consider using an ORM (like SQLAlchemy) for better security and maintainability