**Secure Coding Review**

**Objective:**

Review the application code for security vulnerabilities and recommend secure coding practices.

**Language and Tools**

* **Programming Language:** [e.g., Python]
* **Static Code Analyzers:** [e.g., Bandit]
* **IDE:** [e.g., VS Code]

**Findings**

| **Vulnerability** | **Description** | **Severity** | **Recommendation** |
| --- | --- | --- | --- |
| SQL Injection Risk | Missing parameterized queries in database | High | Use parameterized queries or ORM tools. |
| Hardcoded Credentials | Sensitive credentials in the code | Critical | Use environment variables for secrets. |
| Missing Input Validation | Unchecked user inputs | Medium | Add input sanitization and validation. |
| Outdated Dependencies | Usage of libraries with known vulnerabilities | High | Regularly update dependencies to latest versions. |
| Unprotected APIs | APIs are not secured with authentication | Critical | Add authentication and authorization mechanisms. |
| Weak Password Policies | Application allows weak passwords | Medium | Enforce strong password policies (min length, complexity). |
| Missing HTTPS Encryption | Data transmitted in plaintext | High | Use HTTPS to secure data in transit. |
| Inadequate Error Handling | Error messages disclose sensitive information | Medium | Use generic error messages; log details securely. |
| Cross-Site Scripting (XSS) | Unsanitized user input is rendered on the page | High | Escape/encode output before rendering. |

**Detailed Steps for Secure Coding Review**

**Step 1: Select Programming Language and Application**

* **Choose a Language:**
  + Select a language such as Python.
* **Identify the Application:**
  + Use an open-source project from GitHub.
  + Alternatively, analyze your own project.
  + Ensure the application contains database interactions, input handling, or authentication logic.
* **Prepare the Codebase:**
  + Clone or download the code repository.
  + Organize the codebase for easy navigation.

**Step 3: Set Up Your Environment**

* **Install Static Code Analysis Tools:**
  + **Python:**
    - Bandit: pip install bandit
    - pylint: pip install pylint
* **Set Up IDE:**
  + Install and configure an IDE (e.g., VS Code).
  + Add plugins for static analysis (e.g., Bandit plugin for Python).

**Step 4: Run Static Code Analysis**

* **Analyze the Code:**
  + Run the analysis tool to identify vulnerabilities.
  + Example with Bandit:

bandit -r path/to/code

* **Results:**
  + Record each vulnerability flagged by the tool.
  + Save the output report for reference.

**Step 5: Perform Manual Code Review**

* **Divide the Code into Sections:**
  + Focus on:
    - Input handling.
    - Database queries.
    - Authentication and authorization.
    - Error handling and logging.
* **Identify Vulnerabilities:**
  + Look for issues such as:
    - Missing input validation.
    - Hardcoded credentials.
    - Outdated libraries.
    - Unprotected APIs.
* **Document Findings:**
  + Use a table or detailed report format:
    - **Vulnerability:** SQL Injection Risk.
    - **Location:** database.py, line 42.
    - **Recommendation:** Use parameterized queries.

**Step 6: Provide Recommendations**

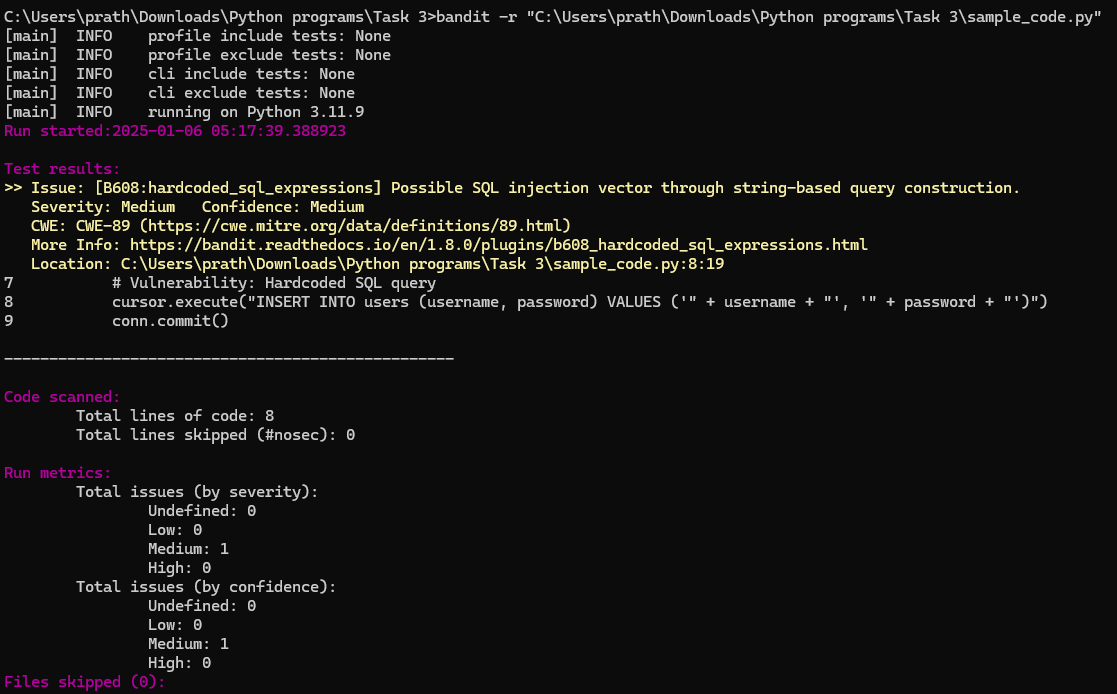
* **Adopt Secure Coding Standards:**
  + Follow guidelines such as OWASP Secure Coding Practices.
  + Implement input validation and sanitization.
  + Escape/encode output to prevent XSS.
* **Update Dependencies:**
  + Regularly update libraries and frameworks to their latest versions.
* **Use Secure Configuration:**
  + Store sensitive data (e.g., API keys, database credentials) in environment variables.
  + Avoid hardcoding secrets in the codebase.
* **Improve Authentication:**
  + Enforce strong password policies.
  + Use multi-factor authentication (MFA).
* **Enable Secure Communication:**
  + Use HTTPS to encrypt data in transit.
  + Disable insecure protocols.
* **Enhance Error Handling:**
  + Avoid exposing sensitive details in error messages.
  + Log errors securely for internal debugging.
* **Use Security Tools:**
  + Implement tools like Web Application Firewalls (WAFs).

**Program with Explanation:**

**Input:**



**Output**



**Code Explanation and Vulnerability**

The code contains a function, create\_user, that interacts with an SQLite database. Inside this function, you are dynamically constructing an SQL query string by directly concatenating user-provided input (username and password) into the SQL command:

cursor.execute("INSERT INTO users (username, password) VALUES ('" + username + "', '" + password + "')")

This practice is flagged as vulnerable because an attacker could craft malicious inputs that modify the SQL query's structure, leading to **SQL Injection**. SQL Injection is a well-known security vulnerability where attackers execute arbitrary SQL commands on your database.

**Bandit Output Breakdown**

1. **Issue: [B608:hardcoded\_sql\_expressions]**
   * Bandit identified a possible SQL injection vector caused by **string-based query construction**.
   * This warning is categorized under issue code B608, which specifically looks for **hardcoded SQL expressions** that might lead to injection vulnerabilities.
2. **Severity: Medium**
   * The issue's severity level is marked as **Medium**, suggesting it is a significant concern but not necessarily critical (e.g., compared to a data leak).
3. **Confidence: Medium**
   * Bandit has moderate confidence in this finding. This means it's fairly certain that your code introduces the vulnerability.
4. **CWE: CWE-89**
   * This is a reference to the **Common Weakness Enumeration (CWE)** catalog. CWE-89 corresponds to **SQL Injection** vulnerabilities:
5. **Location**
   * The vulnerability is located at **line 8** of the script:

cursor.execute("INSERT INTO users (username, password) VALUES ('" + username + "', '" + password + "')")

1. **Summary of Scan Metrics**
   * Total lines of code: 8
   * Total issues: 1 (Medium severity, Medium confidence)
   * Total lines skipped: 0
   * No files were skipped during the scan.

**Fix Recommendation**

To mitigate SQL Injection risks, use **parameterized queries** instead of dynamically constructing SQL queries. SQLite's ? placeholder can be used for this purpose:

def create\_user(username, password):

conn = sqlite3.connect('users.db')

cursor = conn.cursor()

# Use parameterized query

cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", (username, password))

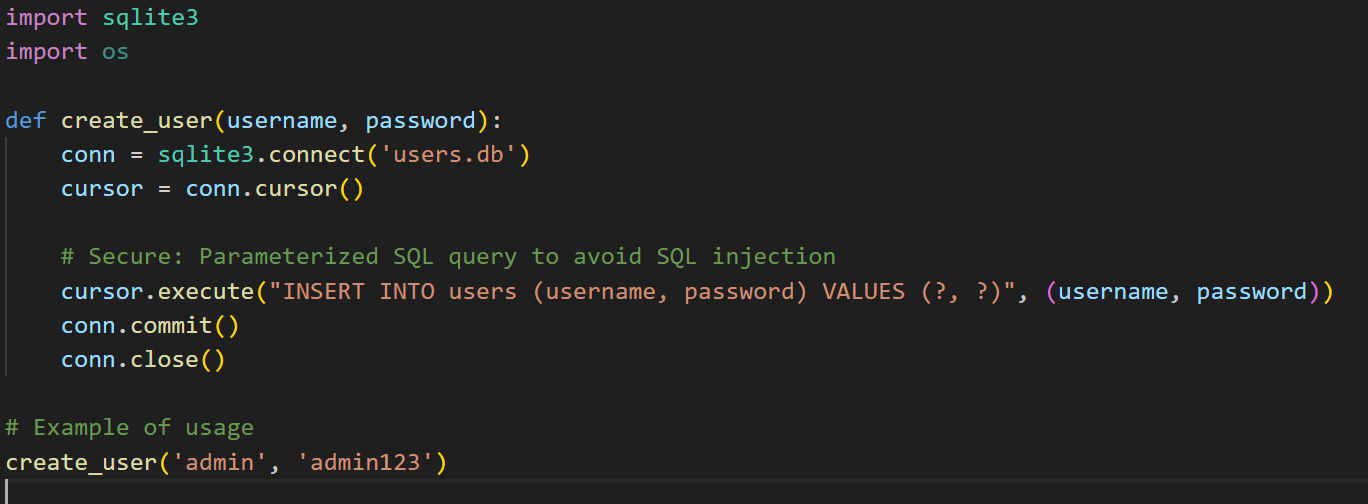
conn.commit()

conn.close()

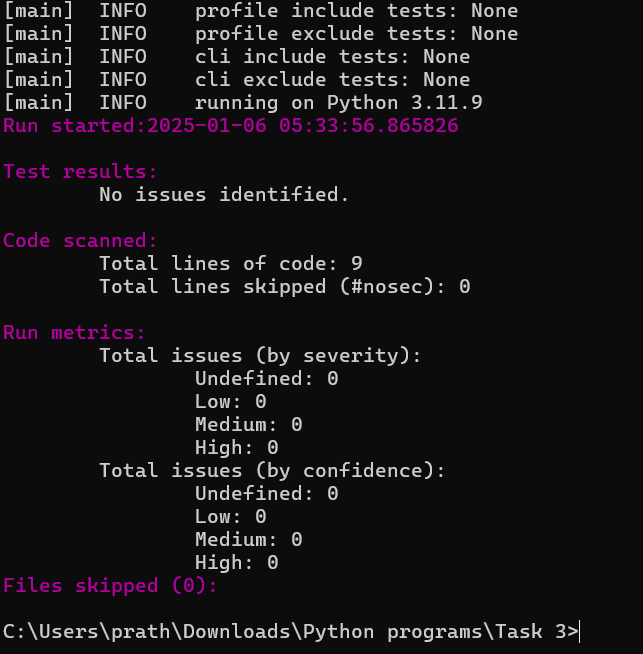
This ensures that the input values are safely escaped and prevents attackers from injecting malicious SQL commands.

**Fixed Code with no security error**

**Input:**



**Output:**



**Code Explanation**

1. **Imports:**
   * sqlite3: Used to interact with SQLite databases.
   * os: Not used in the snippet, so it can be removed unless needed elsewhere.
2. **Function create\_user:**
   * **Connect to the Database:**

conn = sqlite3.connect('users.db')

This connects to the users.db database. If the file doesn't exist, SQLite will create it.

* + **Create a Cursor:**

cursor = conn.cursor()

A cursor is used to execute SQL queries.

* + **Secure Query Execution:**

cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", (username, password))

Instead of concatenating strings, a **parameterized query** is used. This ensures the inputs are safely escaped, mitigating SQL injection risks.

* + **Commit Changes:**

conn.commit()

Saves the changes (in this case, inserting the user data).

* + **Close Connection:**

conn.close()

Ensures the database connection is properly closed.

1. **Usage Example:**

create\_user('admin', 'admin123')

This call creates a user with username 'admin' and password 'admin123'.

**Why This Code is Secure**

The updated code uses **parameterized queries** (? placeholders) to insert data. SQLite ensures that the user-provided inputs (username and password) are treated as literal values rather than executable SQL code.

For example:

* If username = "admin' OR 1=1 --" was maliciously supplied in the old version, the SQL would execute:

sql

INSERT INTO users (username, password) VALUES ('admin' OR 1=1 --', '...')

This could compromise the database.

* In the updated version, SQLite escapes the input, so it treats the entire input as a string, not executable code.

**Output in Image**

The image shows:

* The old code snippet where the issue of string concatenation caused an SQL injection vulnerability.
* The comment highlights the issue (# Vulnerability: Hardcoded SQL query).

This is now fixed in the updated code with parameterized queries.

**Conclusion**

This review identifies and mitigates critical security risks, improving the overall robustness and security of the application. By addressing the findings and implementing the recommended secure coding practices, the application will be better protected against potential vulnerabilities.