Here’s a **formatted** guide with **instructions** for implementing SQL injection in both **Python (SQLite)** and **C++ (SQLite)** while also demonstrating how to **prevent it** using parameterized queries.

# **SQL Injection Demonstration & Prevention (Python & C++ with SQLite)**

## **1. Setting Up the SQLite Test Database**

Before implementing the Python or C++ script, let's create a simple SQLite database.

### **Create a Sample Database (test.db)**

Run the following SQL commands in an SQLite shell or script to set up the database:

-- Create a simple users table

CREATE TABLE users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT NOT NULL,

password TEXT NOT NULL

);

-- Insert some sample users

INSERT INTO users (username, password) VALUES ('alice', 'password123');

INSERT INTO users (username, password) VALUES ('bob', 'securepass');

# **2. Python Implementation**

## **Vulnerable Python Script (Prone to SQL Injection)**

import sqlite3

def get\_user\_details(username):

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

cursor = conn.cursor()

# \*\*Vulnerable Query (Prone to SQL Injection)\*\*

query = f"SELECT \* FROM users WHERE username = '{username}'"

cursor.execute(query)

result = cursor.fetchall()

conn.close()

return result

# \*\*Test SQL Injection\*\*

print(get\_user\_details("' OR '1'='1")) # This will return all users!

### **Explanation**

* The function constructs an **SQL query** using f"SELECT \* FROM users WHERE username = '{username}'"

If an attacker inputs "' OR '1'='1", the query becomes:  
 SELECT \* FROM users WHERE username = '' OR '1'='1'

* Since '1'='1' is always **true**, all users are returned.

## **Safe Python Script (Using Parameterized Queries)**

import sqlite3

def get\_user\_details\_safe(username):

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

cursor = conn.cursor()

# \*\*Safe Query (Using Parameterized Queries)\*\*

query = "SELECT \* FROM users WHERE username = ?"

cursor.execute(query, (username,)) # Parameterized query prevents injection

result = cursor.fetchall()

conn.close()

return result

# \*\*Test SQL Injection (This should NOT return all users)\*\*

print(get\_user\_details\_safe("' OR '1'='1")) # Returns nothing or error

### **Why is this safe?**

* **Parameterized queries** ensure that input is treated as **data**, not SQL code.
* Prevents **SQL injection** regardless of the input provided.

# **3. C++ Implementation**

### **Dependencies**

* Install the **SQLite3** library.
  + On Linux: sudo apt install libsqlite3-dev
  + On Windows: Download SQLite3 and link against sqlite3.lib

## **Vulnerable C++ Code (Prone to SQL Injection)**

#include <iostream>

#include <sqlite3.h>

void getUserDetails(const std::string& username) {

sqlite3\* db;

sqlite3\_open("test.db", &db);

sqlite3\_stmt\* stmt;

// \*\*Vulnerable Query\*\*

std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";

if (sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, nullptr) == SQLITE\_OK) {

while (sqlite3\_step(stmt) == SQLITE\_ROW) {

std::cout << "User: " << sqlite3\_column\_text(stmt, 1) << "\n";

}

}

sqlite3\_finalize(stmt);

sqlite3\_close(db);

}

int main() {

// \*\*Test SQL Injection\*\*

getUserDetails("' OR '1'='1"); // This will return all users!

return 0;

}

### **Why is this vulnerable?**

* The query **concatenates user input** directly into SQL.
* Injecting "' OR '1'='1" results in **unauthorized access** to all users.

## **Safe C++ Code (Using Parameterized Queries)**

#include <iostream>

#include <sqlite3.h>

void getUserDetailsSafe(const std::string& username) {

sqlite3\* db;

sqlite3\_open("test.db", &db);

sqlite3\_stmt\* stmt;

// \*\*Safe Query Using Parameterized Statements\*\*

std::string query = "SELECT \* FROM users WHERE username = ?";

if (sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, nullptr) == SQLITE\_OK) {

sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_STATIC);

while (sqlite3\_step(stmt) == SQLITE\_ROW) {

std::cout << "User: " << sqlite3\_column\_text(stmt, 1) << "\n";

}

}

sqlite3\_finalize(stmt);

sqlite3\_close(db);

}

int main() {

// \*\*Test SQL Injection (This should NOT return all users)\*\*

getUserDetailsSafe("' OR '1'='1");

return 0;

}

### **Why is this safe?**

* Uses **sqlite3\_prepare\_v2** to precompile the SQL statement.
* **sqlite3\_bind\_text** ensures that input is treated as **data**.
* Prevents SQL injection, as the input **cannot alter the SQL structure**.

# **4. Key Takeaways**

| **Approach** | **Vulnerable?** | **Explanation** |
| --- | --- | --- |
| **Python - String Concatenation** | ✅ Yes | Directly embeds user input into SQL. |
| **Python - Parameterized Queries** | ❌ No | Uses ? placeholders to prevent SQL injection. |
| **C++ - String Concatenation** | ✅ Yes | Directly constructs SQL query from input. |
| **C++ - Prepared Statements** | ❌ No | Uses sqlite3\_bind\_text to safely bind input. |

# **5. Summary & Best Practices**

✅ **Always use parameterized queries** (Prepared Statements) to prevent SQL injection.  
 ✅ **Never concatenate user input** directly into an SQL query.  
 ✅ **Validate and sanitize input** before passing it to the database.  
 ✅ **Use least privilege access** when connecting to the database to minimize risk.  
 ✅ **Monitor logs** for unusual query patterns that may indicate an attack.

By following these practices, you can **effectively protect** your applications from SQL injection attacks! 🚀

Certainly! Below is a **step-by-step guide** on setting up and testing SQL injection **on VS Code (Windows) using SQLite**, starting **from opening VS Code** to running **both Python and C++ implementations**.

# **🛠️ Setting Up SQL Injection Demo in VS Code (Windows) with SQLite**

## **🔹 1. Install Prerequisites**

Before we begin, ensure you have the following installed:

### **✅ Required Software**

1. **VS Code** – Download from [here](https://code.visualstudio.com/)
2. **SQLite** – Download the SQLite tools from [here](https://www.sqlite.org/download.html)
   * Extract sqlite-tools-win32-x86 and **note the folder path**.
3. **SQLite VS Code Extension** (optional but recommended)
   * Open **VS Code** → Go to **Extensions** (Ctrl + Shift + X)
   * Search for **SQLite** and install the one by **Alex Covizzi** (SQLite extension).
4. **Python** – Install [Python for Windows](https://www.python.org/downloads/)
   * Ensure python is added to your system PATH.
5. **C++ Compiler** (for C++ implementation)
   * Install **MinGW-w64** from [Mingw-w64](https://www.mingw-w64.org/)
   * Add MinGW to the system PATH.

## **🔹 2. Set Up SQLite in VS Code**

### **✅ Create a New VS Code Project**

1. **Open VS Code**
2. Click **File** → **Open Folder...** → Select or create a folder (e.g., SQLiteProject)
3. **Create a new SQLite database:**
   * Open **VS Code terminal** (Ctrl + ~)

Navigate to the folder using:  
 cd path\to\SQLiteProject

Create the database by running:  
 sqlite3 test.db

Inside the SQLite prompt, run:  
 CREATE TABLE users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT NOT NULL,

password TEXT NOT NULL

);

INSERT INTO users (username, password) VALUES ('alice', 'password123');

INSERT INTO users (username, password) VALUES ('bob', 'securepass');

**Exit SQLite shell:** .exit

## **🔹 3. Writing the Python Code**

### **✅ Vulnerable Python Script (Prone to SQL Injection)**

1. In VS Code, **create a new Python file**:  
   * vulnerable\_sql.py

**Copy the following code:** import sqlite3

def get\_user\_details(username):

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

cursor = conn.cursor()

# \*\*Vulnerable Query (Prone to SQL Injection)\*\*

query = f"SELECT \* FROM users WHERE username = '{username}'"

cursor.execute(query)

result = cursor.fetchall()

conn.close()

return result

# \*\*Test SQL Injection\*\*

print(get\_user\_details("' OR '1'='1")) # This will return all users!

**Run the script in VS Code Terminal** python vulnerable\_sql.py

1. If successful, you’ll see **all users printed**, proving SQL injection works.

### **✅ Secure Python Script (Using Parameterized Queries)**

1. **Create a new file:** secure\_sql.py

**Copy this secure version:** import sqlite3

def get\_user\_details\_safe(username):

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

cursor = conn.cursor()

# \*\*Safe Query Using Parameterized Queries\*\*

query = "SELECT \* FROM users WHERE username = ?"

cursor.execute(query, (username,)) # Parameterized query

result = cursor.fetchall()

conn.close()

return result

# \*\*Test SQL Injection (This should NOT return all users)\*\*

print(get\_user\_details\_safe("' OR '1'='1")) # Should return nothing

**Run it:** python secure\_sql.py

1. This time, the attack **will not work**—only a valid username will return results.

## **🔹 4. Writing the C++ Code**

### **✅ Vulnerable C++ Code (Prone to SQL Injection)**

1. **Create a new file**: vulnerable\_sql.cpp

**Copy this code:** #include <iostream>

#include <sqlite3.h>

void getUserDetails(const std::string& username) {

sqlite3\* db;

sqlite3\_open("test.db", &db);

sqlite3\_stmt\* stmt;

// \*\*Vulnerable Query\*\*

std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";

if (sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, nullptr) == SQLITE\_OK) {

while (sqlite3\_step(stmt) == SQLITE\_ROW) {

std::cout << "User: " << sqlite3\_column\_text(stmt, 1) << "\n";

}

}

sqlite3\_finalize(stmt);

sqlite3\_close(db);

}

int main() {

// \*\*Test SQL Injection\*\*

getUserDetails("' OR '1'='1"); // This will return all users!

return 0;

}

1. **Compile and run the program**:

Compile using MinGW:  
 g++ vulnerable\_sql.cpp -o vulnerable\_sql.exe -lsqlite3

Run:  
 ./vulnerable\_sql.exe

* + **All users should be printed**, proving SQL injection works.

### **✅ Secure C++ Code (Using Prepared Statements)**

1. **Create a new file**: secure\_sql.cpp

**Copy this secure version:** #include <iostream>

#include <sqlite3.h>

void getUserDetailsSafe(const std::string& username) {

sqlite3\* db;

sqlite3\_open("test.db", &db);

sqlite3\_stmt\* stmt;

// \*\*Safe Query Using Parameterized Statements\*\*

std::string query = "SELECT \* FROM users WHERE username = ?";

if (sqlite3\_prepare\_v2(db, query.c\_str(), -1, &stmt, nullptr) == SQLITE\_OK) {

sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_STATIC);

while (sqlite3\_step(stmt) == SQLITE\_ROW) {

std::cout << "User: " << sqlite3\_column\_text(stmt, 1) << "\n";

}

}

sqlite3\_finalize(stmt);

sqlite3\_close(db);

}

int main() {

// \*\*Test SQL Injection (This should NOT return all users)\*\*

getUserDetailsSafe("' OR '1'='1");

return 0;

}

**Compile and run the program**:  
  
 g++ secure\_sql.cpp -o secure\_sql.exe -lsqlite3

./secure\_sql.exe

1. **SQL injection no longer works!**

## **🔹 5. Summary**

| **Approach** | **Language** | **Vulnerable?** | **Protection Used** |
| --- | --- | --- | --- |
| **String Concatenation** | Python | ✅ Yes | None (Prone to injection) |
| **Parameterized Query** | Python | ❌ No | ? Placeholder |
| **String Concatenation** | C++ | ✅ Yes | None (Prone to injection) |
| **Prepared Statements** | C++ | ❌ No | sqlite3\_bind\_text |

## **🔹 6. Final Thoughts**

✅ **Always use prepared statements** to prevent SQL injection.  
 ✅ **Avoid string concatenation** for SQL queries.  
 ✅ **Use SQLite extensions in VS Code** for easier database management.

This guide ensures **hands-on experience** with both **vulnerable and secure implementations** of SQL queries in **Python and C++** on **VS Code (Windows)**. 🚀 Let me know if you need further assistance!

# References

1. Stallings, W., & Brown, L. (2025). SQL injection attacks and injection attacks. In *Computer Security Principles and Practice (5th ed., pp. 256–264, 607–622).* Pearson.

# Acknowledgements

The authors acknowledge the use of AI-assisted tools, including **ChatGPT-4o** and **DeepSeek R1**, in supporting aspects of drafting, conceptualization, and refinement of this work. These tools were utilized to enhance clarity and structure; however, all content was critically reviewed and validated by the authors to ensure accuracy, integrity, and originality. The final responsibility for any errors or omissions remains solely with the authors.

## Prompts

### DeepSeek R1

* Referencing just these two documents (Computer Security Principles and Practice 5th edition; ch. 5.4 and 11.2), I want you to guide me as I attempt to write a basic python script for SQL injection. For now, don't write any code, just give me either pseudo code, steps, suggestions, and guidance

### Chat GPT 4o

* (directly copied output from deekSeek r1 response and pasted in chat gpt) “Certainly! Writing a basic Python script to demonstrate SQL injection involves understanding how SQL injection works and how to simulate it in a controlled environment. Since SQL injection is a serious security vulnerability, it's important to only perform this in a safe, controlled environment, such as a local test database, and never on a live system or without proper authorization. Here’s a step-by-step guide to help you write a basic Python script for SQL injection:…” “format it and provide instructions also for c++ and sqlite”
* can you tailor the instructions to do it on vs code on windows with sqlite extension (start all the way from opening vscode)