



## **Experiment 01**

### **Aim:**

To implement and evaluate the effectiveness of **parameterized queries** in preventing **SQL injection attacks** in web and database-connected applications.

---

### **Objectives:**

- Understand the mechanism of SQL injection attacks.
  - Identify vulnerable points in application code.
  - Implement parameterized queries in different languages (e.g., Python, PHP, Java).
  - Evaluate the system before and after mitigation.
  - Ensure secure interaction between user input and database.
- 

### **Tools:**

- **Languages:** Python, PHP, Java
  - **Databases:** SQLite, MySQL
  - **Tools:** SQLMap, DB Browser, IDE (VS Code/IntelliJ)
- 

### **Theory:**

#### **SQL Injection**

SQL injection occurs when an attacker can **inject malicious SQL** into queries executed by an application, typically by manipulating user inputs. If not handled properly, it can lead to:

- Unauthorized data access
- Data manipulation or deletion
- Complete DB compromise

## Parameterized Queries

Also known as **prepared statements**, these:

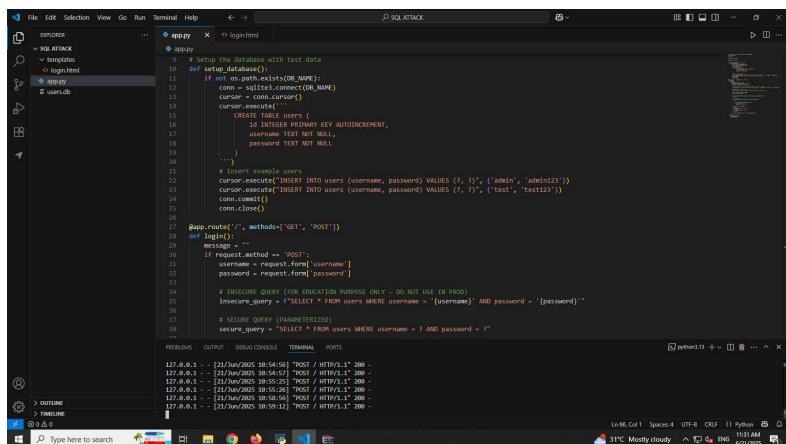
- Use placeholders (**?**, **:param**) instead of inserting user input directly into the query string.
- Prevent user input from being interpreted as SQL code.
- Ensure **separation of SQL logic and data**, which is the foundation of injection prevention.

## Results:

SQL injection is a type of attack where a hacker inserts or "injects" malicious SQL code into input fields (like login forms or search bars) to access or manipulate the database. This can lead to serious issues such as data leaks, unauthorized logins, or even deletion of data.

To prevent this, developers can use **parameterized queries** (also called prepared statements). These queries work by using placeholders instead of directly inserting user input into SQL statements. The database treats user input as data only, not as part of the SQL command, so the injected code cannot be executed.

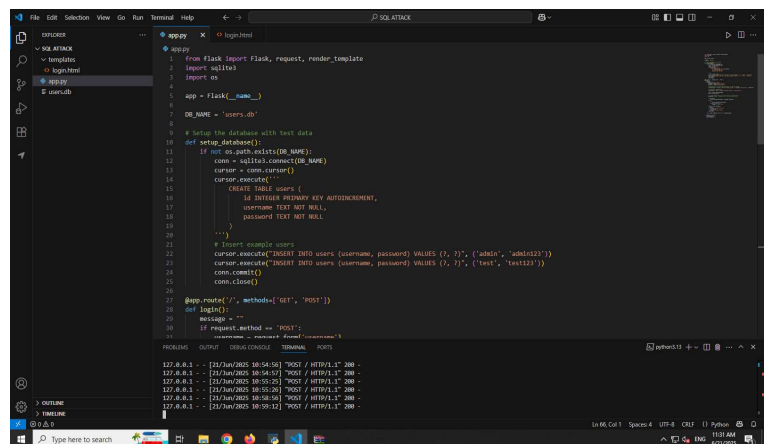
## Screenshots of Step-wise execution of tool



```

1 # Setup the database with test data
2 def setup_database():
3     if not os.path.exists(DB_NAME):
4         conn = sqlite3.connect(DB_NAME)
5         cursor = conn.cursor()
6         cursor.execute("""
7             CREATE TABLE users (
8                 id INTEGER PRIMARY KEY AUTOINCREMENT,
9                 username TEXT NOT NULL,
10                password TEXT NOT NULL
11            )
12        """)
13        # Insert example users
14        cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", ("admin", "admin123"))
15        cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", ("test", "test123"))
16        conn.commit()
17        conn.close()
18
19 @app.route('/', methods=['GET', 'POST'])
20 def login():
21     message = ""
22     if request.method == "POST":
23         username = request.form['username']
24         password = request.form['password']
25
26         # Insecure query (FOR EDUCATION PURPOSE ONLY - DO NOT USE IN PROD)
27         insecure_query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
28         # Secure query (PARAMETERIZED)
29         secure_query = "SELECT * FROM users WHERE username = ? AND password = ?"

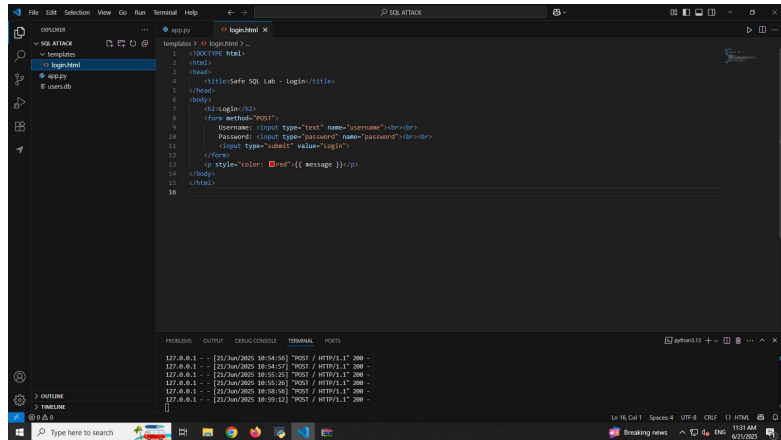
```



```

1 # Setup the database with test data
2 def setup_database():
3     if not os.path.exists(DB_NAME):
4         conn = sqlite3.connect(DB_NAME)
5         cursor = conn.cursor()
6         cursor.execute("""
7             CREATE TABLE users (
8                 id INTEGER PRIMARY KEY AUTOINCREMENT,
9                 username TEXT NOT NULL,
10                password TEXT NOT NULL
11            )
12        """)
13        # Insert example users
14        cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", ("admin", "admin123"))
15        cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", ("test", "test123"))
16        conn.commit()
17        conn.close()
18
19 @app.route('/', methods=['GET', 'POST'])
20 def login():
21     message = ""
22     if request.method == "POST":
23         username = request.form['username']
24         password = request.form['password']
25
26         # Insecure query (FOR EDUCATION PURPOSE ONLY - DO NOT USE IN PROD)
27         insecure_query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
28         # Secure query (PARAMETERIZED)
29         secure_query = "SELECT * FROM users WHERE username = ? AND password = ?"

```



The screenshot shows a web browser window with a login page. The page has a title "Safe SQL Lab - Login" and a form with fields for "Username" and "Password". Below the form is a "Login" button. The terminal window below the browser shows network traffic, including a POST request to "/login" with a body containing "username=admin" and "password=admin".

### Learning Outcomes:

- Effectiveness: Parameterized queries fully prevent SQL injection.
- Easy to implement in modern languages.
- Must be used consistently across the app.
- Often supported by ORMs by default.

**Conclusion:** Parameterized queries are a simple, efficient, and essential defense against SQL injection. They ensure inputs are treated as data, not executable code.

### QUESTION:

1. What is SQL injection ?
2. Steps to prevent SQL injection ?



**TCET**  
**BE COMPUTER SCIENCE & ENGINEERING (CYBER SECURITY)**  
Choice Based Credit Grading System (CBCGS)  
Under TCET Autonomy



NAME:

ROLLNO:

BE-CSE

For Faculty Use:

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				