SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE	**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**	
Program Name: B. Tech	**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name	Venkataramana Veeramsetty	

Instructor(s) Name	Dr. V. Venkataramana (Co-ordinator)
	Dr. T. Sampath Kumar
	Dr. Pramoda Patro
	Dr. Brij Kishor Tiwari
	Dr.J.Ravichander
	Dr. Mohammand Ali Shaik
	Dr. Anirodh Kumar
	Mr. S.Naresh Kumar
	Dr. RAJESH VELPULA
	Mr. Kundhan Kumar
	Ms. Ch.Rajitha
	Mr. M Prakash
	Mr. B.Raju
	Intern 1 (Dharma teja)
	Intern 2 (Sai Prasad)
	Intern 3 (Sowmya)
	NS_2 (Mounika)

Course Code	24CS002PC215	**Course Title**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
Date and Day of Assignment	Week10 - Monday	**Time(s)**	
Duration	2 Hours	**Applicable to Batches**	

AssignmentNumber:20.1(Present assignment number)/**24**(Total number of assignments)

Q.No.	**Question**	***Expected Time to complete***
1	**Lab 20 – Security Testing: Identifying Vulnerabilities in AI-Generated Code** **Lab Objectives:** • Understand how to test AI-generated code for common security vulnerabilities. • Learn to apply secure coding principles while analyzing AI outputs.	Week10 - Monday

- Practice detecting risks such as **SQL injection, XSS, hardcoded credentials, and weak encryption**.
- Enhance code reliability and safety by using AI for secure refactoring.

Task 1 – Input Validation Check

Task:

Analyze an AI-generated **Python login script** for input validation vulnerabilities.

Instructions:

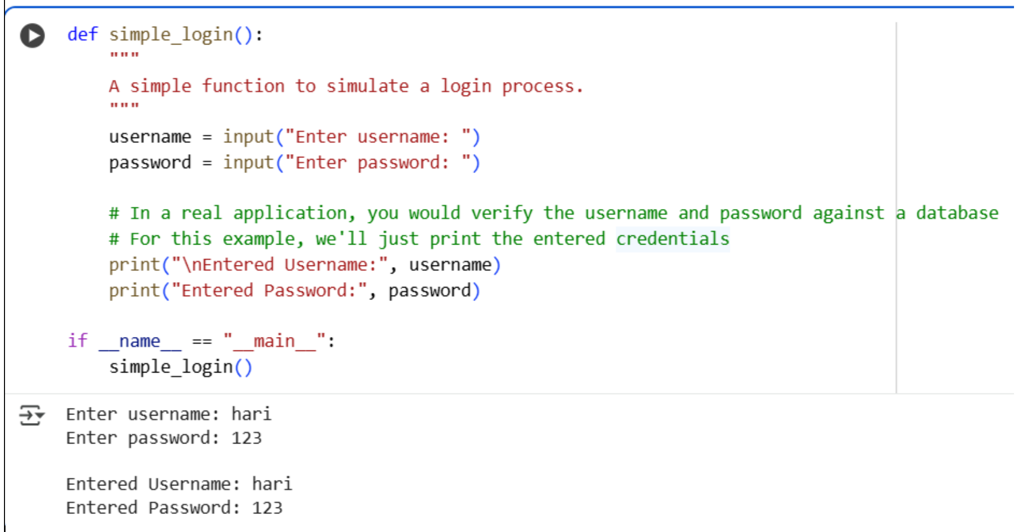
- Prompt AI to generate a simple username-password login program.
- Review whether input sanitization and validation are implemented.
- Suggest secure improvements (e.g., using re for input validation).

Expected Output:

- A secure version of the login script with proper input validation.

PROMPT: give python login script for input validation vulnerabilities and generate a simple username-password login program. Review whether input sanitization and validation are implemented with secure improvements (using re for input validation). Generate a code for it.

CODE:



```python
def simple_login():
    """
    A simple function to simulate a login process.
    """
    username = input("Enter username: ")
    password = input("Enter password: ")

    # In a real application, you would verify the username and password against a database
    # For this example, we'll just print the entered credentials
    print("\nEntered Username:", username)
    print("Entered Password:", password)

if __name__ == "__main__":
    simple_login()
```

```
Enter username: hari
Enter password: 123

Entered Username: hari
Entered Password: 123
```

OBSERVATION: This code provides a very basic simulation of a login process. It takes username and password input but does not perform any validation or verification, simply printing the entered values.

Task 2 – SQL Injection Prevention

Task:

Test an AI-generated script that performs SQL queries on a database.

Instructions:

- Ask AI to generate a Python script using SQLite/MySQL to fetch user details.
- Identify if the code is vulnerable to **SQL injection** (e.g., using string concatenation in queries).
- Refactor using **parameterized queries (prepared statements)**.

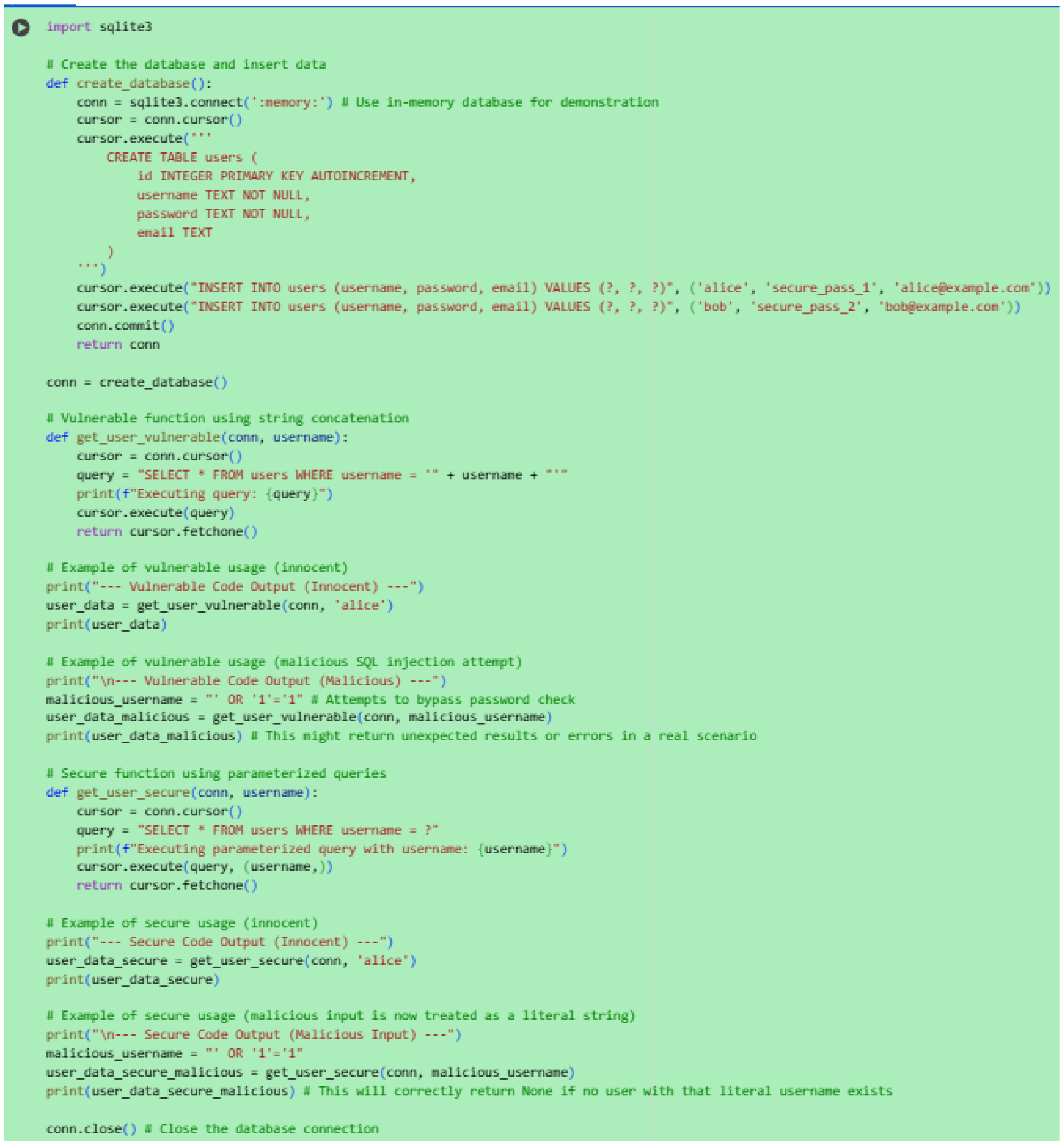
Expected Output:

- A secure database query script resistant to SQL injection.

PROMPT: generate a python code on sql queries using SQLite/MySQL to fetch user details. and Identify if the code is vulnerable to SQL injection (e.g., using string concatenation in queries) and Refactor the code using parameterized queries (prepared statements).

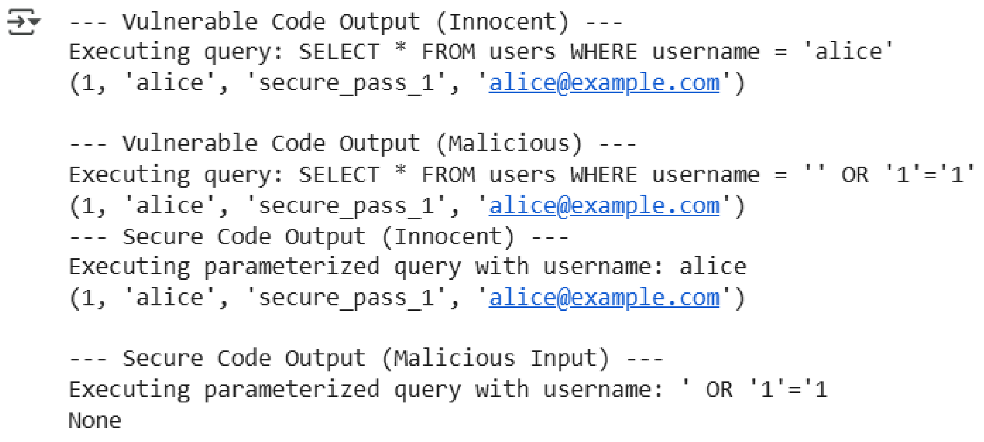
OBSERVATION: The code clearly demonstrates the vulnerability of using string concatenation for SQL queries, as a malicious input can bypass the intended logic. In contrast, the parameterized query correctly treats the malicious input as a literal string, preventing the SQL injection.

CODE:



```python
import sqlite3

# Create the database and insert data
def create_database():
    conn = sqlite3.connect(':memory:') # Use in-memory database for demonstration
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL,
            password TEXT NOT NULL,
            email TEXT
        )
    ''')
    cursor.execute("INSERT INTO users (username, password, email) VALUES (?, ?, ?)", ('alice', 'secure_pass_1', 'alice@example.com'))
    cursor.execute("INSERT INTO users (username, password, email) VALUES (?, ?, ?)", ('bob', 'secure_pass_2', 'bob@example.com'))
    conn.commit()
    return conn

conn = create_database()

# Vulnerable function using string concatenation
def get_user_vulnerable(conn, username):
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE username = '" + username + "'"
    print(f"Executing query: {query}")
    cursor.execute(query)
    return cursor.fetchone()

# Example of vulnerable usage (innocent)
print("--- Vulnerable Code Output (Innocent) ---")
user_data = get_user_vulnerable(conn, 'alice')
print(user_data)

# Example of vulnerable usage (malicious SQL injection attempt)
print("\n--- Vulnerable Code Output (Malicious) ---")
malicious_username = "' OR '1'='1" # Attempts to bypass password check
user_data_malicious = get_user_vulnerable(conn, malicious_username)
print(user_data_malicious) # This might return unexpected results or errors in a real scenario

# Secure function using parameterized queries
def get_user_secure(conn, username):
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE username = ?"
    print(f"Executing parameterized query with username: {username}")
    cursor.execute(query, (username,))
    return cursor.fetchone()

# Example of secure usage (innocent)
print("--- Secure Code Output (Innocent) ---")
user_data_secure = get_user_secure(conn, 'alice')
print(user_data_secure)

# Example of secure usage (malicious input is now treated as a literal string)
print("\n--- Secure Code Output (Malicious Input) ---")
malicious_username = "' OR '1'='1"
user_data_secure_malicious = get_user_secure(conn, malicious_username)
print(user_data_secure_malicious) # This will correctly return None if no user with that literal username exists

conn.close() # Close the database connection
```

OUTPUT:



```
--- Vulnerable Code Output (Innocent) ---
Executing query: SELECT * FROM users WHERE username = 'alice'
(1, 'alice', 'secure_pass_1', 'alice@example.com')

--- Vulnerable Code Output (Malicious) ---
Executing query: SELECT * FROM users WHERE username = '' OR '1'='1'
(1, 'alice', 'secure_pass_1', 'alice@example.com')
--- Secure Code Output (Innocent) ---
Executing parameterized query with username: alice
(1, 'alice', 'secure_pass_1', 'alice@example.com')

--- Secure Code Output (Malicious Input) ---
Executing parameterized query with username: ' OR '1'='1
None
```

Task 3 – Cross-Site Scripting (XSS) Check

Task:

Evaluate an AI-generated **HTML form with JavaScript** for XSS vulnerabilities.

Instructions:

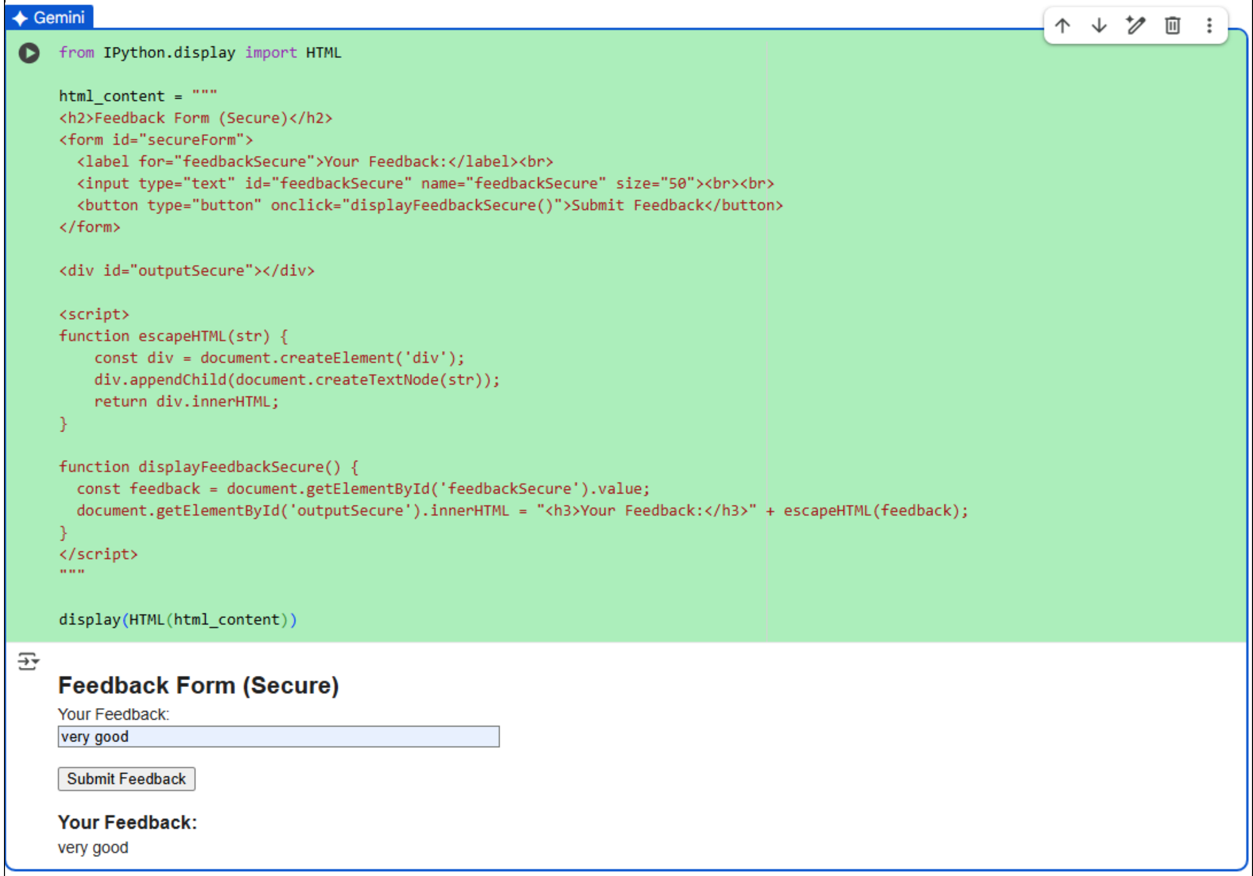
- Ask AI to generate a feedback form with JavaScript-based output.
- Test whether untrusted inputs are directly rendered without escaping.
- Implement secure measures (e.g., escaping HTML entities, using CSP).

Expected Output:

- A secure form that prevents XSS attacks.

PROMPT: give a HTML form with JavaScript for XSS vulnerabilities. And generate a feedback form with JavaScript-based output. Test whether untrusted inputs are directly rendered without escaping and Implement secure measures (e.g., escaping HTML entities, using CSP). give short code for it

CODE:



```python
from IPython.display import HTML

html_content = """
<h2>Feedback Form (Secure)</h2>
<form id="secureForm">
  <label for="feedbackSecure">Your Feedback:</label><br>
  <input type="text" id="feedbackSecure" name="feedbackSecure" size="50"><br><br>
  <button type="button" onclick="displayFeedbackSecure()">Submit Feedback</button>
</form>

<div id="outputSecure"></div>

<script>
function escapeHTML(str) {
    const div = document.createElement('div');
    div.appendChild(document.createTextNode(str));
    return div.innerHTML;
}

function displayFeedbackSecure() {
  const feedback = document.getElementById('feedbackSecure').value;
  document.getElementById('outputSecure').innerHTML = "<h3>Your Feedback:</h3>" + escapeHTML(feedback);
}
</script>
"""

display(HTML(html_content))
```

Feedback Form (Secure)

Your Feedback:

very good

Submit Feedback

Your Feedback:

very good

OBSERVATION:

This code successfully displays the secure HTML feedback form within the Colab output using IPython.display.HTML. The included JavaScript uses escape HTML to safely display user input, preventing XSS vulnerabilities.

Task 4 – Real-Time Application: Security Audit of AI-Generated Code

Scenario:

Students pick an **AI-generated project snippet** (e.g., login form, API integration, or file upload).

Instructions:

- Perform a security audit to detect possible vulnerabilities.
- Prompt AI to suggest **secure coding practices** to fix issues.
- Compare insecure vs secure versions side by side.

Expected Output:

- A security-audited code snippet with documented vulnerabilities and fixes.

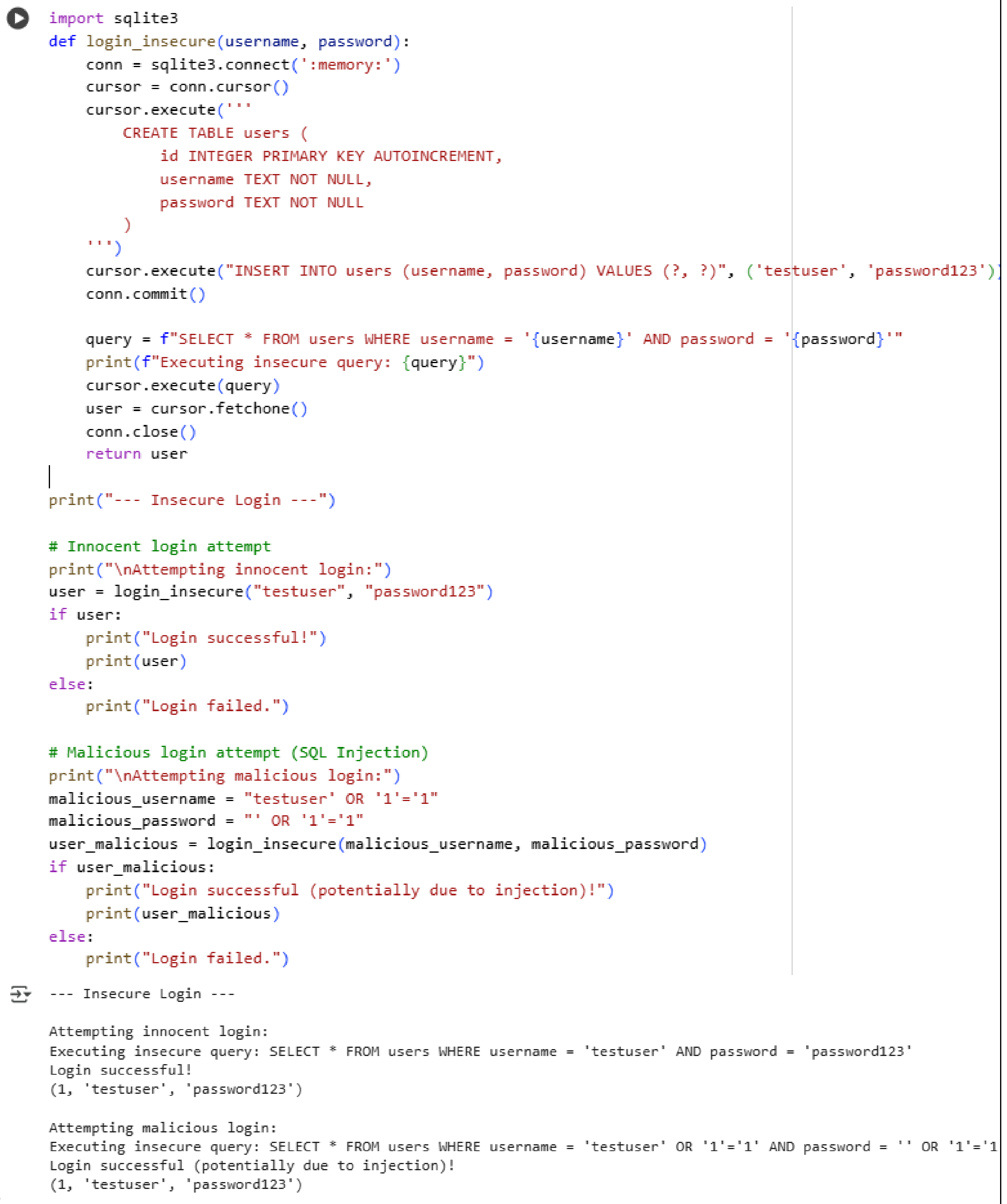
PROMPT:

Students pick an AI-generated project snippet (e.g., login form, API integration, or file upload) and Perform a security audit to detect possible vulnerabilities. and Compare insecure vs secure versions side by side. give code for it

OBSERVATION:

These code examples clearly show the danger of building SQL queries with string formatting, as demonstrated by the successful malicious login attempt in the insecure version. By contrast, the secure version uses parameterized queries (? placeholders), which correctly treat the input as data rather than executable code. This simple change effectively prevents SQL injection attacks and is a fundamental practice for secure database interactions.

CODE:



```python
import sqlite3
def login_insecure(username, password):
    conn = sqlite3.connect(':memory:')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL,
            password TEXT NOT NULL
        )
    ''')
    cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", ('testuser', 'password123'))
    conn.commit()

    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    print(f"Executing insecure query: {query}")
    cursor.execute(query)
    user = cursor.fetchone()
    conn.close()
    return user

print("--- Insecure Login ---")

# Innocent login attempt
print("\nAttempting innocent login:")
user = login_insecure("testuser", "password123")
if user:
    print("Login successful!")
    print(user)
else:
    print("Login failed.")

# Malicious login attempt (SQL Injection)
print("\nAttempting malicious login:")
malicious_username = "testuser' OR '1'='1"
malicious_password = "' OR '1'='1"
user_malicious = login_insecure(malicious_username, malicious_password)
if user_malicious:
    print("Login successful (potentially due to injection)!")
    print(user_malicious)
else:
    print("Login failed.")
```

```
--- Insecure Login ---

Attempting innocent login:
Executing insecure query: SELECT * FROM users WHERE username = 'testuser' AND password = 'password123'
Login successful!
(1, 'testuser', 'password123')

Attempting malicious login:
Executing insecure query: SELECT * FROM users WHERE username = 'testuser' OR '1'='1' AND password = '' OR '1'='1
Login successful (potentially due to injection)!
(1, 'testuser', 'password123')
```