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	Week10 - Monday

outputs.

- 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:

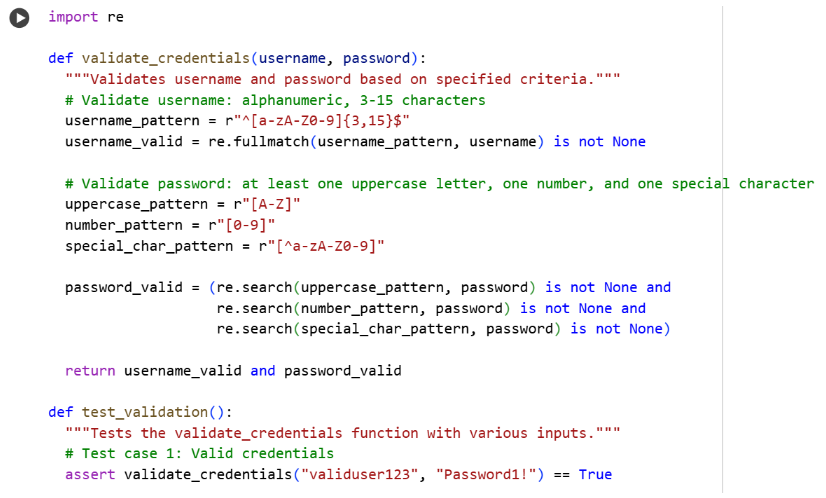
Analyze an AI-generated **Python login script** for input validation vulnerabilities.

Instructions:

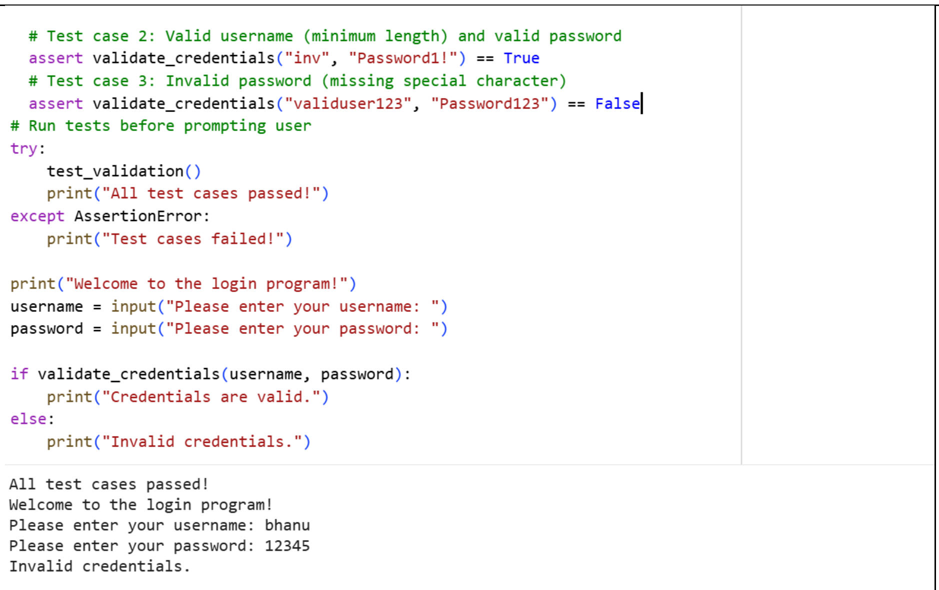
- Prompt AI to generate a simple username-password login program.
- Review whether input sanitization and validation are implemented.
- Suggest secure improvements (e.g., using re for input validation).

Expected Output:

- A secure version of the login script with proper input validation.
- **Prompt:** Generate a Python login program that reads username and password from user input, validates usernames to be alphanumeric 3–15 characters and passwords to contain at least one uppercase letter, one number, and one special character using re, prints whether the credentials pass validation, and includes three assert test cases for the validation logic.

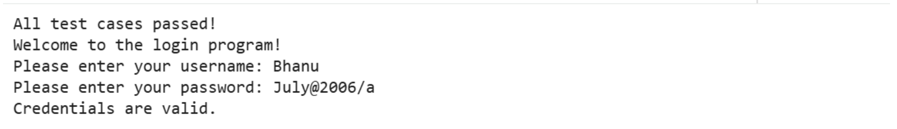


```python
import re

def validate_credentials(username, password):
  """Validates username and password based on specified criteria."""
  # Validate username: alphanumeric, 3-15 characters
  username_pattern = r"^[a-zA-Z0-9]{3,15}$"
  username_valid = re.fullmatch(username_pattern, username) is not None

  # Validate password: at least one uppercase letter, one number, and one special character
  uppercase_pattern = r"[A-Z]"
  number_pattern = r"[0-9]"
  special_char_pattern = r"[^a-zA-Z0-9]"

  password_valid = (re.search(uppercase_pattern, password) is not None and
                    re.search(number_pattern, password) is not None and
                    re.search(special_char_pattern, password) is not None)

  return username_valid and password_valid

def test_validation():
  """Tests the validate_credentials function with various inputs."""
  # Test case 1: Valid credentials
  assert validate_credentials("validuser123", "Password1!") == True
```



```
  # Test case 2: Valid username (minimum length) and valid password
  assert validate_credentials("inv", "Password1!") == True
  # Test case 3: Invalid password (missing special character)
  assert validate_credentials("validuser123", "Password123") == False
# Run tests before prompting user
try:
    test_validation()
    print("All test cases passed!")
except AssertionError:
    print("Test cases failed!")

print("Welcome to the login program!")
username = input("Please enter your username: ")
password = input("Please enter your password: ")

if validate_credentials(username, password):
    print("Credentials are valid.")
else:
    print("Invalid credentials.")
```

```
All test cases passed!
Welcome to the login program!
Please enter your username: bhanu
Please enter your password: 12345
Invalid credentials.
```



```
All test cases passed!
Welcome to the login program!
Please enter your username: Bhanu
Please enter your password: July@2006/a
Credentials are valid.
```

Task 2 – SQL Injection Prevention

Task:

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

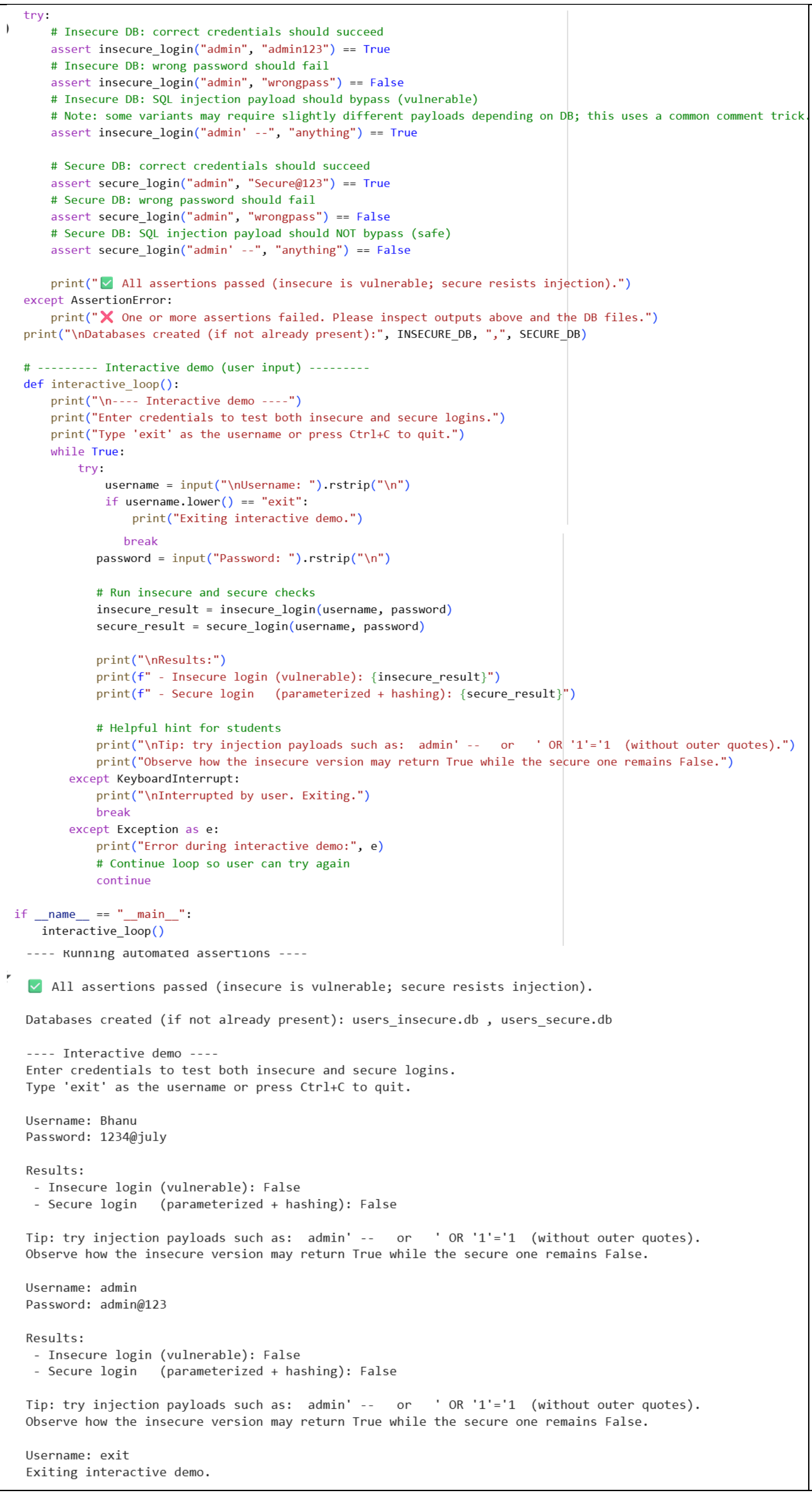
Instructions:

- 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 script using SQLite that accepts a username from user input, demonstrates an insecure user lookup using string-concatenated SQL and the resulting SQL injection (e.g., ' OR '1'='1), then refactors to a secure version using parameterized queries (?), and includes three assert tests proving the secure version resists injection.

```python
#!/usr/bin/env python3
"""
Interactive SQL injection demo + secure fix.

- Creates two SQLite DBs: users_insecure.db (plain-text, vulnerable) and users_secure.db (hashed passwords + parameterized queries
- Runs assertions to demonstrate vulnerability (insecure) and protection (secure).
- THEN enters an interactive loop where the user can enter username/password to see results for both insecure and secure logins.
- Use Ctrl+C or enter 'exit' as username to quit the interactive loop.
"""

import sqlite3
import hashlib
import os
import sys

# --------- Utilities ---------
def hash_password(password: str) -> str:
    """Return SHA-256 hex digest for the given password."""
    return hashlib.sha256(password.encode()).hexdigest()

# --------- Insecure DB & Login (vulnerable to SQL injection) ---------
INSECURE_DB = "users_insecure.db"

def setup_insecure_db():
    """Create an insecure DB and add a demo user (plain-text password)."""
    conn = sqlite3.connect(INSECURE_DB)
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS users (username TEXT PRIMARY KEY, password TEXT)")
    # Use INSERT OR IGNORE to avoid duplicate insert on repeated runs
    cursor.execute("INSERT OR IGNORE INTO users(username, password) VALUES (?, ?)", ('admin', 'admin123'))

    conn.commit()
    conn.close()

def insecure_login(username: str, password: str) -> bool:
    """
    Vulnerable login: builds SQL with string concatenation — demonstrates SQL injection risk.
    Returns True if login succeeds (row found), False otherwise.
    """
    conn = sqlite3.connect(INSECURE_DB)
    cursor = conn.cursor()
    # WARNING: String concatenation leads to SQL injection vulnerability.
    query = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password + "'"
    try:
        cursor.execute(query)
        result = cursor.fetchone()
    except Exception as e:
        # If the malicious input broke the SQL syntax, treat as failed login
        print("Insecure login query error:", e)
        result = None
    conn.close()
    return bool(result)

# --------- Secure DB & Login (parameterized queries + hashing) ---------
SECURE_DB = "users_secure.db"

def setup_secure_db():
    """Create a secure DB and add a demo user with hashed password."""
    conn = sqlite3.connect(SECURE_DB)
    cursor = conn.cursor()
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS users (
            username TEXT PRIMARY KEY,
            password TEXT
        )
    """)
    # Insert hashed admin password if not already present
    cursor.execute("INSERT OR IGNORE INTO users(username, password) VALUES (?, ?)",
                   ('admin', hash_password('Secure@123')))
    conn.commit()
    conn.close()

def secure_login(username: str, password: str) -> bool:
    """
    Secure login: uses parameterized queries and compares hashed password.
    Returns True if login succeeds, False otherwise.
    """
    conn = sqlite3.connect(SECURE_DB)
    cursor = conn.cursor()
    hashed_pw = hash_password(password)
    cursor.execute("SELECT * FROM users WHERE username = ? AND password = ?", (username, hashed_pw))
    result = cursor.fetchone()
    conn.close()
    return bool(result)

# --------- Setup both databases ---------
setup_insecure_db()
setup_secure_db()

# --------- Automated Assertions demonstrating behavior ---------
print("---- Running automated assertions ----\n")
```



```python
try:
    # Insecure DB: correct credentials should succeed
    assert insecure_login("admin", "admin123") == True
    # Insecure DB: wrong password should fail
    assert insecure_login("admin", "wrongpass") == False
    # Insecure DB: SQL injection payload should bypass (vulnerable)
    # Note: some variants may require slightly different payloads depending on DB; this uses a common comment trick.
    assert insecure_login("admin' --", "anything") == True

    # Secure DB: correct credentials should succeed
    assert secure_login("admin", "Secure@123") == True
    # Secure DB: wrong password should fail
    assert secure_login("admin", "wrongpass") == False
    # Secure DB: SQL injection payload should NOT bypass (safe)
    assert secure_login("admin' --", "anything") == False

    print("✅ All assertions passed (insecure is vulnerable; secure resists injection).")
except AssertionError:
    print("❌ One or more assertions failed. Please inspect outputs above and the DB files.")
print("\nDatabases created (if not already present):", INSECURE_DB, ",", SECURE_DB)

# --------- Interactive demo (user input) ---------
def interactive_loop():
    print("\n---- Interactive demo ----")
    print("Enter credentials to test both insecure and secure logins.")
    print("Type 'exit' as the username or press Ctrl+C to quit.")
    while True:
        try:
            username = input("\nUsername: ").rstrip("\n")
            if username.lower() == "exit":
                print("Exiting interactive demo.")
                break
            password = input("Password: ").rstrip("\n")

            # Run insecure and secure checks
            insecure_result = insecure_login(username, password)
            secure_result = secure_login(username, password)

            print("\nResults:")
            print(f" - Insecure login (vulnerable): {insecure_result}")
            print(f" - Secure login   (parameterized + hashing): {secure_result}")

            # Helpful hint for students
            print("\nTip: try injection payloads such as:  admin' --   or   ' OR '1'='1  (without outer quotes).")
            print("Observe how the insecure version may return True while the secure one remains False.")
        except KeyboardInterrupt:
            print("\nInterrupted by user. Exiting.")
            break
        except Exception as e:
            print("Error during interactive demo:", e)
            # Continue loop so user can try again
            continue

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

```
---- Running automated assertions ----

✅ All assertions passed (insecure is vulnerable; secure resists injection).

Databases created (if not already present): users_insecure.db , users_secure.db

---- Interactive demo ----
Enter credentials to test both insecure and secure logins.
Type 'exit' as the username or press Ctrl+C to quit.

Username: Bhanu
Password: 1234@july

Results:
 - Insecure login (vulnerable): False
 - Secure login   (parameterized + hashing): False

Tip: try injection payloads such as:  admin' --   or   ' OR '1'='1  (without outer quotes).
Observe how the insecure version may return True while the secure one remains False.

Username: admin
Password: admin@123

Results:
 - Insecure login (vulnerable): False
 - Secure login   (parameterized + hashing): False

Tip: try injection payloads such as:  admin' --   or   ' OR '1'='1  (without outer quotes).
Observe how the insecure version may return True while the secure one remains False.

Username: exit
Exiting interactive demo.
```

Task 3 – Cross-Site Scripting (XSS) Check

Task:

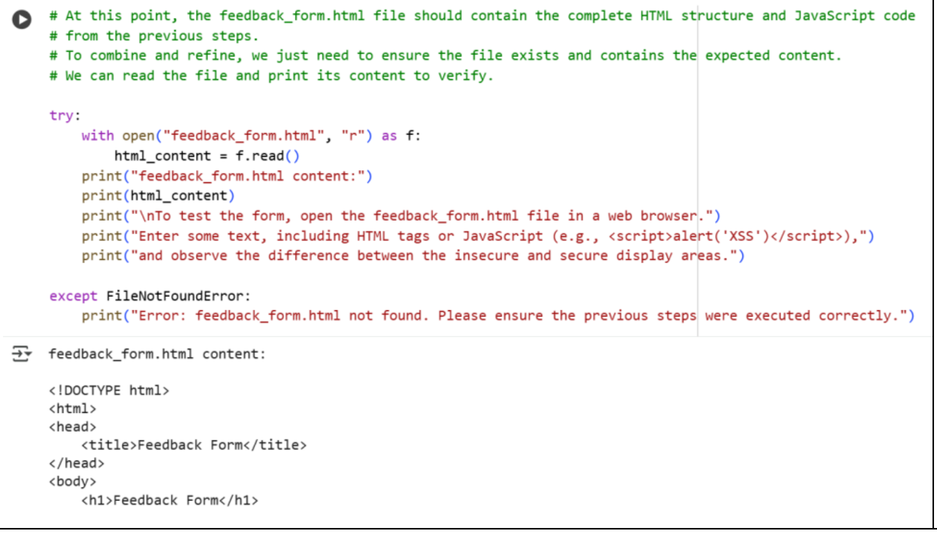
Evaluate an AI-generated **HTML form with JavaScript** for XSS vulnerabilities.

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).

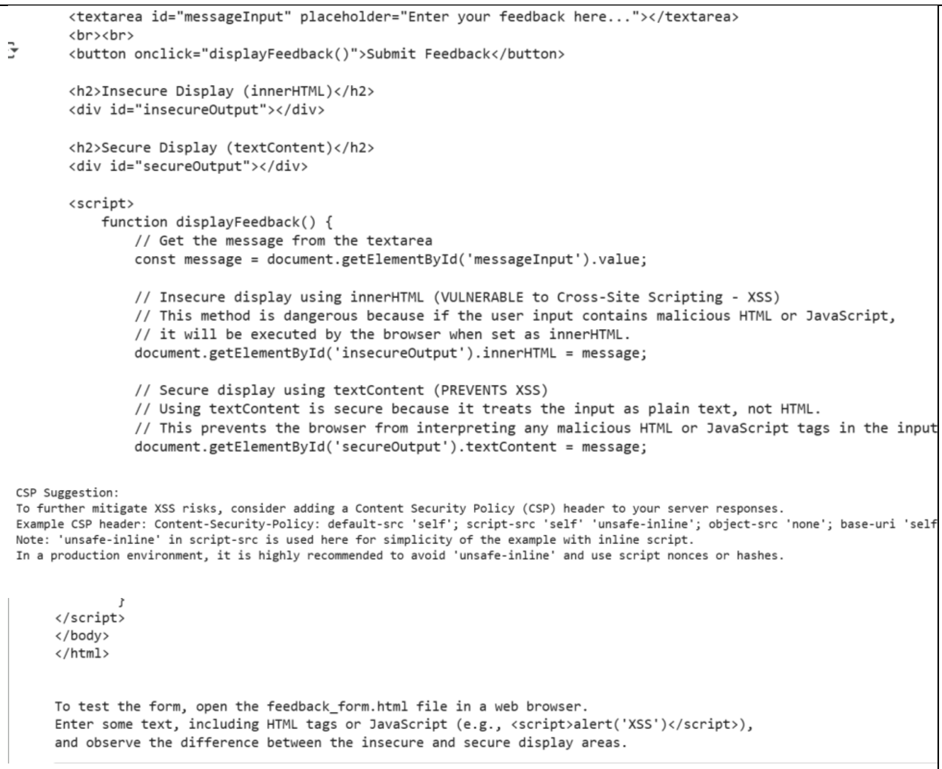
Expected Output:

- A secure form that prevents XSS attacks.
- **Prompt:**Generate an HTML + JavaScript feedback form that **takes a user-entered message**, first displays it insecurely using innerHTML to demonstrate XSS risk, then shows a secure version using textContent or proper HTML-entity escaping and a CSP suggestion, with inline comments explaining the vulnerability and fix.

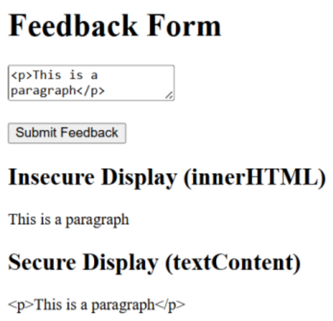


```python
# At this point, the feedback_form.html file should contain the complete HTML structure and JavaScript code
# from the previous steps.
# To combine and refine, we just need to ensure the file exists and contains the expected content.
# We can read the file and print its content to verify.

try:
    with open("feedback_form.html", "r") as f:
        html_content = f.read()
    print("feedback_form.html content:")
    print(html_content)
    print("\nTo test the form, open the feedback_form.html file in a web browser.")
    print("Enter some text, including HTML tags or JavaScript (e.g., <script>alert('XSS')</script>),")
    print("and observe the difference between the insecure and secure display areas.")

except FileNotFoundError:
    print("Error: feedback_form.html not found. Please ensure the previous steps were executed correctly.")
```

```
feedback_form.html content:

<!DOCTYPE html>
<html>
<head>
    <title>Feedback Form</title>
</head>
<body>
    <h1>Feedback Form</h1>
```



```
        <textarea id="messageInput" placeholder="Enter your feedback here..."></textarea>
        <br><br>
        <button onclick="displayFeedback()">Submit Feedback</button>

        <h2>Insecure Display (innerHTML)</h2>
        <div id="insecureOutput"></div>

        <h2>Secure Display (textContent)</h2>
        <div id="secureOutput"></div>

        <script>
            function displayFeedback() {
                // Get the message from the textarea
                const message = document.getElementById('messageInput').value;

                // Insecure display using innerHTML (VULNERABLE to Cross-Site Scripting - XSS)
                // This method is dangerous because if the user input contains malicious HTML or JavaScript,
                // it will be executed by the browser when set as innerHTML.
                document.getElementById('insecureOutput').innerHTML = message;

                // Secure display using textContent (PREVENTS XSS)
                // Using textContent is secure because it treats the input as plain text, not HTML.
                // This prevents the browser from interpreting any malicious HTML or JavaScript tags in the input
                document.getElementById('secureOutput').textContent = message;

CSP Suggestion:
To further mitigate XSS risks, consider adding a Content Security Policy (CSP) header to your server responses.
Example CSP header: Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline'; object-src 'none'; base-uri 'self
Note: 'unsafe-inline' in script-src is used here for simplicity of the example with inline script.
In a production environment, it is highly recommended to avoid 'unsafe-inline' and use script nonces or hashes.

            }
    </script>
    </body>
    </html>

    To test the form, open the feedback_form.html file in a web browser.
    Enter some text, including HTML tags or JavaScript (e.g., <script>alert('XSS')</script>),
    and observe the difference between the insecure and secure display areas.
```



Feedback Form

```
<p>This is a paragraph</p>
```

Insecure Display (innerHTML)

This is a paragraph

Secure Display (textContent)

<p>This is a paragraph</p>

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).

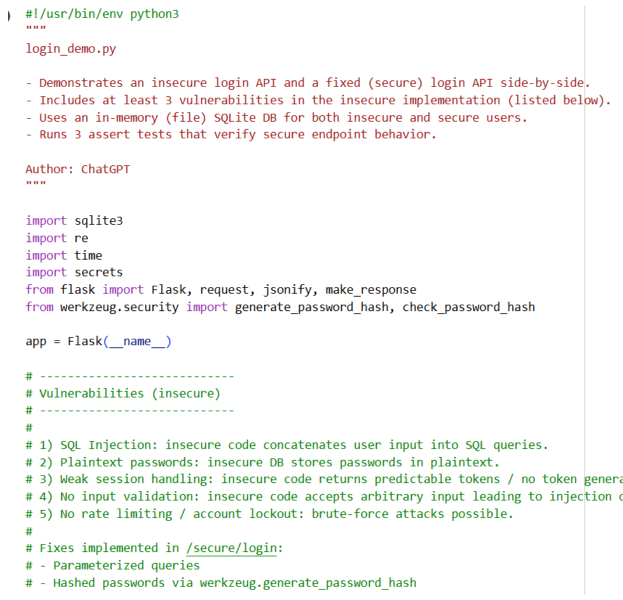
Instructions:

- 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.

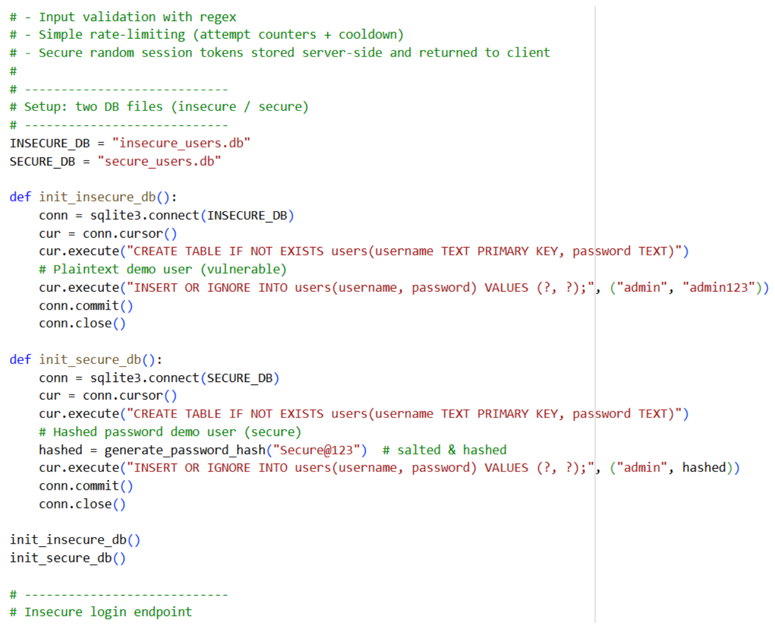
Expected Output:

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

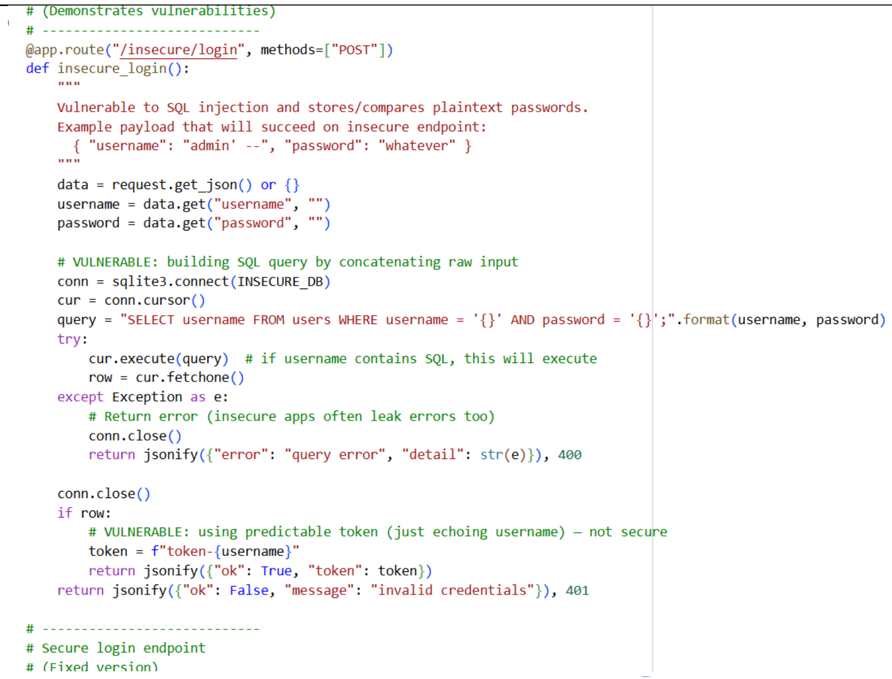
Prompt: Generate an AI-created project snippet (choose a login API endpoint or file-upload handler) that accepts user input, list at least three security vulnerabilities found in the snippet, produce side-by-side insecure and fixed code with inline comments for each fix, and include three assert tests verifying the fixed version's behavior.



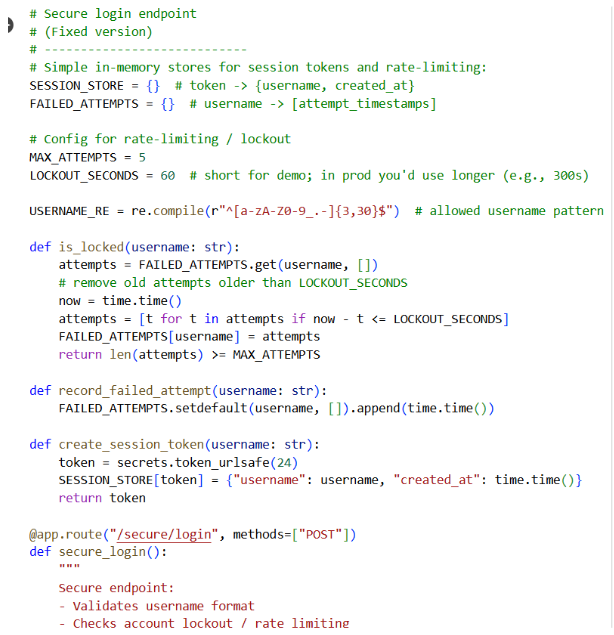
```python
#!/usr/bin/env python3
"""
login_demo.py

- Demonstrates an insecure login API and a fixed (secure) login API side-by-side.
- Includes at least 3 vulnerabilities in the insecure implementation (listed below).
- Uses an in-memory (file) SQLite DB for both insecure and secure users.
- Runs 3 assert tests that verify secure endpoint behavior.

Author: ChatGPT
"""

import sqlite3
import re
import time
import secrets
from flask import Flask, request, jsonify, make_response
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__)

# ---------------------------
# Vulnerabilities (insecure)
# ---------------------------
#
# 1) SQL Injection: insecure code concatenates user input into SQL queries.
# 2) Plaintext passwords: insecure DB stores passwords in plaintext.
# 3) Weak session handling: insecure code returns predictable tokens / no token generation.
# 4) No input validation: insecure code accepts arbitrary input leading to injection or malformed requests.
# 5) No rate limiting / account lockout: brute-force attacks possible.
#
# Fixes implemented in /secure/login:
# - Parameterized queries
# - Hashed passwords via werkzeug.generate_password_hash
```



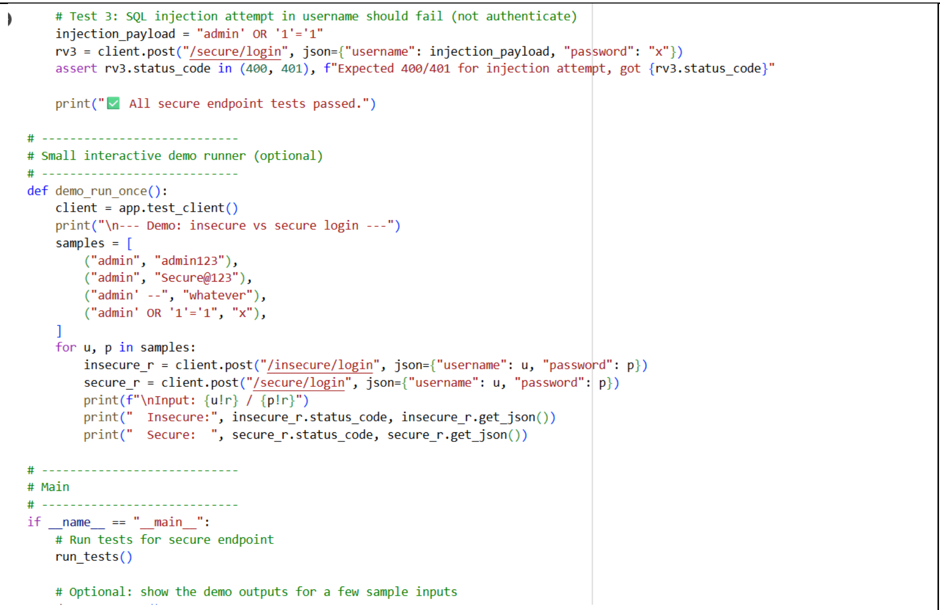
```python
# - Input validation with regex
# - Simple rate-limiting (attempt counters + cooldown)
# - Secure random session tokens stored server-side and returned to client
#
# ---------------------------
# Setup: two DB files (insecure / secure)
# ---------------------------
INSECURE_DB = "insecure_users.db"
SECURE_DB = "secure_users.db"

def init_insecure_db():
    conn = sqlite3.connect(INSECURE_DB)
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS users(username TEXT PRIMARY KEY, password TEXT)")
    # Plaintext demo user (vulnerable)
    cur.execute("INSERT OR IGNORE INTO users(username, password) VALUES (?, ?);", ("admin", "admin123"))
    conn.commit()
    conn.close()

def init_secure_db():
    conn = sqlite3.connect(SECURE_DB)
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS users(username TEXT PRIMARY KEY, password TEXT)")
    # Hashed password demo user (secure)
    hashed = generate_password_hash("Secure@123")  # salted & hashed
    cur.execute("INSERT OR IGNORE INTO users(username, password) VALUES (?, ?);", ("admin", hashed))
    conn.commit()
    conn.close()

init_insecure_db()
init_secure_db()

# ---------------------------
# Insecure login endpoint
```



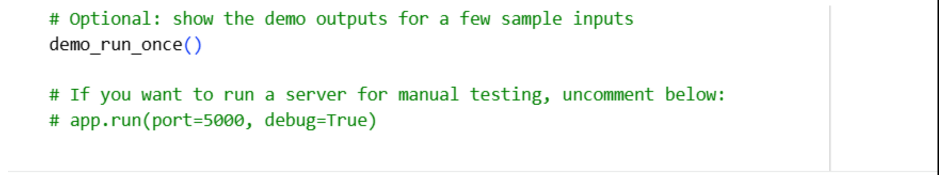
```python
# (Demonstrates vulnerabilities)
# ----------------------------
@app.route("/insecure/login", methods=["POST"])
def insecure_login():
    """
    Vulnerable to SQL injection and stores/compares plaintext passwords.
    Example payload that will succeed on insecure endpoint:
      { "username": "admin' --", "password": "whatever" }
    """
    data = request.get_json() or {}
    username = data.get("username", "")
    password = data.get("password", "")

    # VULNERABLE: building SQL query by concatenating raw input
    conn = sqlite3.connect(INSECURE_DB)
    cur = conn.cursor()
    query = "SELECT username FROM users WHERE username = '{}' AND password = '{}';".format(username, password)
    try:
        cur.execute(query)  # if username contains SQL, this will execute
        row = cur.fetchone()
    except Exception as e:
        # Return error (insecure apps often leak errors too)
        conn.close()
        return jsonify({"error": "query error", "detail": str(e)}), 400

    conn.close()
    if row:
        # VULNERABLE: using predictable token (just echoing username) — not secure
        token = f"token-{username}"
        return jsonify({"ok": True, "token": token})
    return jsonify({"ok": False, "message": "invalid credentials"}), 401

# ----------------------------
# Secure login endpoint
# (Fixed version)
```



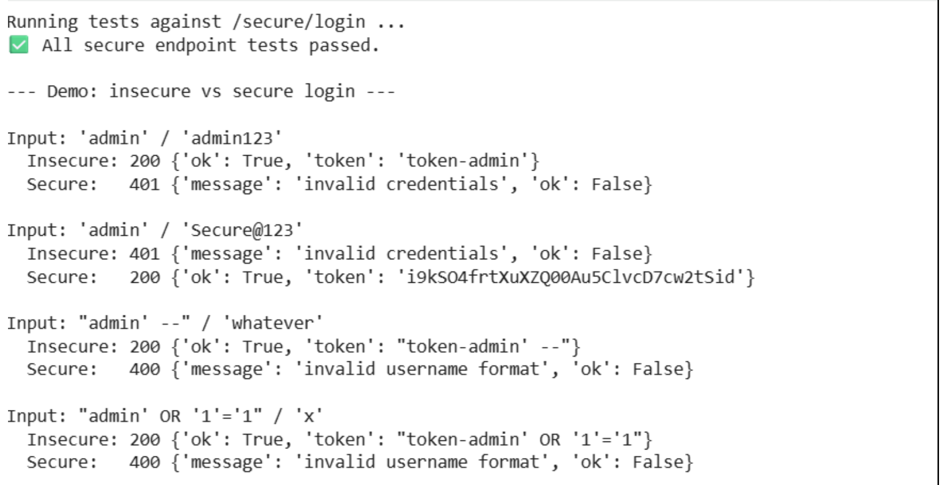
```python
# Secure login endpoint
# (Fixed version)
# ----------------------------
# Simple in-memory stores for session tokens and rate-limiting:
SESSION_STORE = {}  # token -> {username, created_at}
FAILED_ATTEMPTS = {}  # username -> [attempt_timestamps]

# Config for rate-limiting / lockout
MAX_ATTEMPTS = 5
LOCKOUT_SECONDS = 60  # short for demo; in prod you'd use longer (e.g., 300s)

USERNAME_RE = re.compile(r"^[a-zA-Z0-9_.-]{3,30}$")  # allowed username pattern

def is_locked(username: str):
    attempts = FAILED_ATTEMPTS.get(username, [])
    # remove old attempts older than LOCKOUT_SECONDS
    now = time.time()
    attempts = [t for t in attempts if now - t <= LOCKOUT_SECONDS]
    FAILED_ATTEMPTS[username] = attempts
    return len(attempts) >= MAX_ATTEMPTS

def record_failed_attempt(username: str):
    FAILED_ATTEMPTS.setdefault(username, []).append(time.time())

def create_session_token(username: str):
    token = secrets.token_urlsafe(24)
    SESSION_STORE[token] = {"username": username, "created_at": time.time()}
    return token

@app.route("/secure/login", methods=["POST"])
def secure_login():
    """
    Secure endpoint:
    - Validates username format
    - Checks account lockout / rate limiting
    - Uses parameterized SQL queries (prevents injection)
    - Compares hashed password using werkzeug.check_password_hash
    - Issues a secure random session token (server-side stored)
    """
    data = request.get_json() or {}
    username = data.get("username", "")
    password = data.get("password", "")

    # Input validation: reject suspicious characters / lengths early
    if not isinstance(username, str) or not isinstance(password, str):
        return jsonify({"ok": False, "message": "invalid input types"}), 400
    username = username.strip()
    if not USERNAME_RE.match(username):
        return jsonify({"ok": False, "message": "invalid username format"}), 400

    # Check lockout / rate limiting
    if is_locked(username):
        return jsonify({"ok": False, "message": "account temporarily locked due to failed attempts"}), 429

    # Secure DB access: parameterized query (prevents SQL injection)
    conn = sqlite3.connect(SECURE_DB)
    cur = conn.cursor()
    cur.execute("SELECT password FROM users WHERE username = ?;", (username,))
    row = cur.fetchone()
    conn.close()

    if not row:
        # No such user — record failed attempt to slow enumerate attempts
        record_failed_attempt(username)
        return jsonify({"ok": False, "message": "invalid credentials"}), 401

    stored_hash = row[0]
    if not check_password_hash(stored_hash, password):
        record_failed_attempt(username)
        return jsonify({"ok": False, "message": "invalid credentials"}), 401

    # Success: create secure token and reset failed attempts
    token = create_session_token(username)
    FAILED_ATTEMPTS.pop(username, None)
    # Return token in JSON (in production prefer setting secure HttpOnly cookie and using TLS)
    resp = jsonify({"ok": True, "token": token})
    # (Example) set secure flags on cookie if desired (demonstration only)
    resp.set_cookie("session_token", token, httponly=True, samesite="Lax")
    return resp

# ---------------------------
# Tests for secure endpoint
# ---------------------------
def run_tests():
    """
    Three assert tests verifying the fixed version's behavior:
    1) correct credentials -> 200 and token present
    2) wrong password -> 401
    3) SQL injection payload should NOT authenticate via secure endpoint -> 401
    """
    print("Running tests against /secure/login ...")
    client = app.test_client()

    # Test 1: correct credentials
    rv = client.post("/secure/login", json={"username": "admin", "password": "Secure@123"})
    assert rv.status_code == 200, f"Expected 200 for correct credentials, got {rv.status_code}"
    data = rv.get_json()
    assert data.get("ok") == True and "token" in data, "Secure login should return token for correct credentials."

    # Test 2: wrong password
    rv2 = client.post("/secure/login", json={"username": "admin", "password": "wrongpass"})
    assert rv2.status_code == 401, f"Expected 401 for wrong password, got {rv2.status_code}"
```



```
    # Test 3: SQL injection attempt in username should fail (not authenticate)
    injection_payload = "admin' OR '1'='1"
    rv3 = client.post("/secure/login", json={"username": injection_payload, "password": "x"})
    assert rv3.status_code in (400, 401), f"Expected 400/401 for injection attempt, got {rv3.status_code}"

    print("✅ All secure endpoint tests passed.")

# ---------------------------
# Small interactive demo runner (optional)
# ---------------------------
def demo_run_once():
    client = app.test_client()
    print("\n--- Demo: insecure vs secure login ---")
    samples = [
        ("admin", "admin123"),
        ("admin", "Secure@123"),
        ("admin' --", "whatever"),
        ("admin' OR '1'='1", "x"),
    ]
    for u, p in samples:
        insecure_r = client.post("/insecure/login", json={"username": u, "password": p})
        secure_r = client.post("/secure/login", json={"username": u, "password": p})
        print(f"\nInput: {u!r} / {p!r}")
        print("  Insecure:", insecure_r.status_code, insecure_r.get_json())
        print("  Secure:   ", secure_r.status_code, secure_r.get_json())

# ---------------------------
# Main
# ---------------------------
if __name__ == "__main__":
    # Run tests for secure endpoint
    run_tests()

    # Optional: show the demo outputs for a few sample inputs
```



```
    # Optional: show the demo outputs for a few sample inputs
    demo_run_once()

    # If you want to run a server for manual testing, uncomment below:
    # app.run(port=5000, debug=True)
```



```
Running tests against /secure/login ...
✅ All secure endpoint tests passed.

--- Demo: insecure vs secure login ---

Input: 'admin' / 'admin123'
  Insecure: 200 {'ok': True, 'token': 'token-admin'}
  Secure:   401 {'message': 'invalid credentials', 'ok': False}

Input: 'admin' / 'Secure@123'
  Insecure: 401 {'message': 'invalid credentials', 'ok': False}
  Secure:   200 {'ok': True, 'token': 'i9kSO4frtXuXZQ00Au5ClvcD7cw2tSid'}

Input: "admin' --" / 'whatever'
  Insecure: 200 {'ok': True, 'token': "token-admin' --"}
  Secure:   400 {'message': 'invalid username format', 'ok': False}

Input: "admin' OR '1'='1" / 'x'
  Insecure: 200 {'ok': True, 'token': "token-admin' OR '1'='1"}
  Secure:   400 {'message': 'invalid username format', 'ok': False}
```