ASSIGNMENT-20.3

NAME:CHANDA HARINI

ROLL NO.:2403A510E1

BATCH NO.:05

Task 1 – Input Validation Check

PROMPT:

"Generate a simple Python script for a user login system that takes a username and password as input."

CODE:

20.3.1.py

```python
# insecure_login.py

def login():
    """
    A simple, insecure login function that checks against hardcoded credentials.
    WARNING: This code is for demonstration of vulnerabilities and is NOT secure.
    """
    # Hardcoded credentials (In a real app, this is also a vulnerability)
    correct_username = "admin"
    correct_password = "password123"

    # Get user input without any validation or sanitization
    print("--- Insecure Login System ---")
    username = input("Enter username: ")
    password = input("Enter password: ")

    # Directly compare the raw input
    if username == correct_username and password == correct_password:
        print("\n✅ Login successful! Welcome.")
    else:
        print("\n❌ Invalid username or password.")

if __name__ == "__main__":
    login()

```

OUTPUT:

```
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai\.vscode> & "C:/Users/Praneeeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "
c:/Users/Praneeeth Cheekati/OneDrive/Desktop/ai/.vscode/20.3.1.py"
--- Insecure Login System ---
Enter username: admin
Enter password: admin123

❌ Invalid username or password.
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai\.vscode>
```

OBSERVATION:

The initial script is a classic example of trusting user input, which is a primary source of security vulnerabilities.

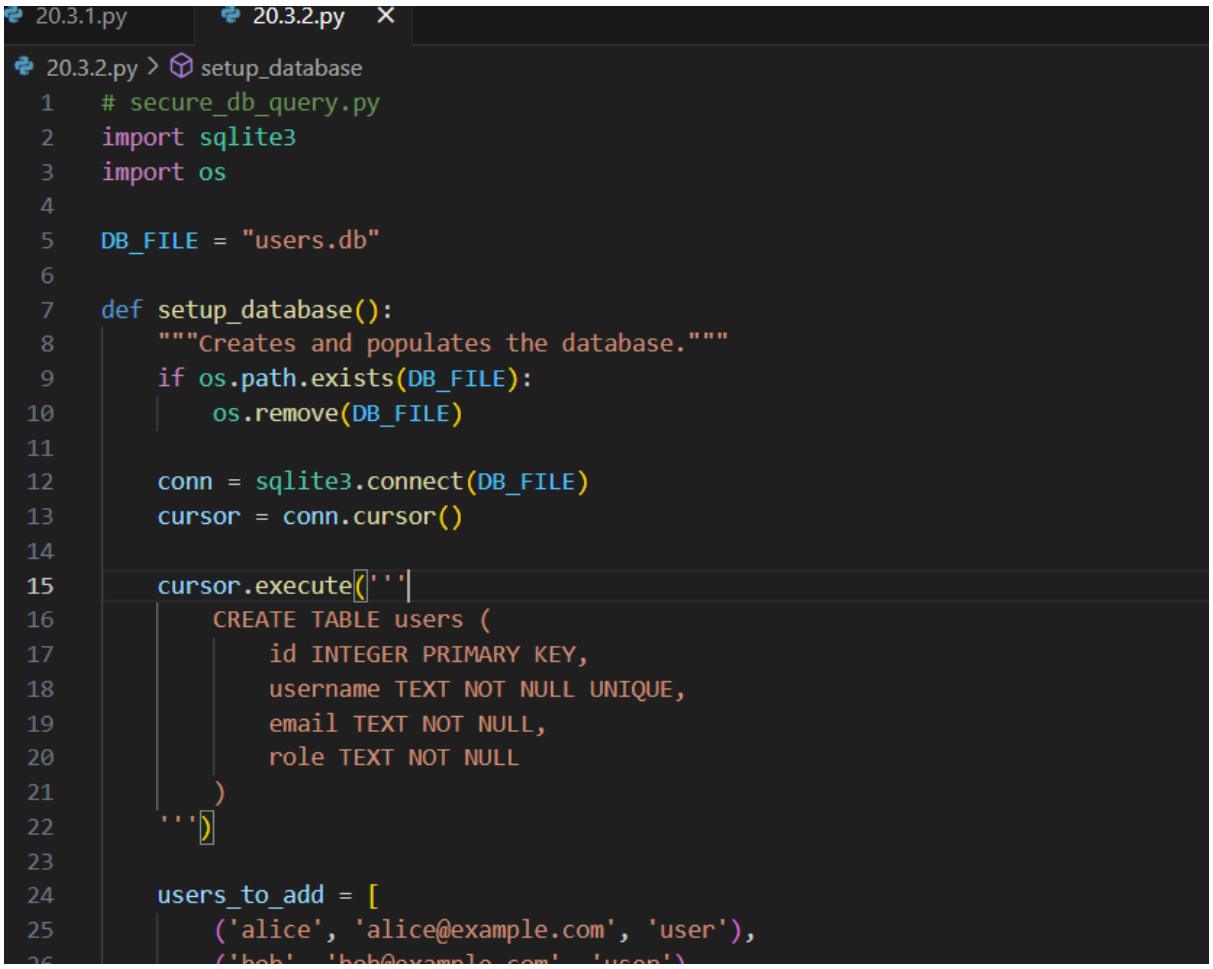
1. **No Input Validation:** The script does not check the format, length, or type of characters in the `username` and `password` fields. A user could enter an empty string, an extremely long string (which could lead to Denial of Service in some systems), or special characters.
2. **No Input Sanitization:** The input is used directly. If this `username` were used to construct a database query (e.g., `f"SELECT * FROM users WHERE username = '{username}'"`), it would be wide open to **SQL Injection**. An attacker could input `' OR '1'='1` as the username to bypass the login.
3. **Vague Error Messages:** The message "Invalid username or password" is good practice as it doesn't reveal which one was incorrect (preventing username enumeration). However, the system gives no feedback to a legitimate user about *why* their input might be invalid (e.g., "Username must be alphanumeric").
4. **Hardcoded Credentials:** Storing credentials directly in the source code is a major security risk. They should be stored securely, for example, as hashed and salted values in a database.

Task 2 – SQL Injection Prevention

PROMPT:

"Generate a Python script using the `sqlite3` library. The script should create a simple `users` table, populate it with a few users, and include a function that takes a username from user input to fetch and display that user's details from the database."

CODE:


20.3.1.py
20.3.2.py
20.3.2.py > setup_database
# secure_db_query.py
import sqlite3
import os
DB_FILE = "users.db"
def setup_database():
"""Creates and populates the database."""
if os.path.exists(DB_FILE):
os.remove(DB_FILE)
conn = sqlite3.connect(DB_FILE)
cursor = conn.cursor()
cursor.execute('''
CREATE TABLE users (
id INTEGER PRIMARY KEY,
username TEXT NOT NULL UNIQUE,
email TEXT NOT NULL,
role TEXT NOT NULL
)
''')
users_to_add = [
('alice', 'alice@example.com', 'user'),

20.3.1.py 20.3.2.py

20.3.2.py > setup_database

```
def setup_database():
        ('alice', 'alice@example.com', 'user'),
        ('bob', 'bob@example.com', 'user'),
        ('charlie_admin', 'charlie@example.com', 'admin')
    ]
    cursor.executemany('INSERT INTO users (username, email, role) VALUES (?, ?, ?)', users_to_add)

    conn.commit()
    conn.close()
    print("Database created and populated.")

def get_user_details_secure(username):
    """
    Fetches user details from the database using a secure parameterized query.
    """
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()

    # SECURE METHOD: Use a placeholder (?) for user input.
    query = "SELECT id, username, email, role FROM users WHERE username = ?"
    print(f"\nExecuting secure query with parameter: ('{username}',)")

    try:
        # Pass the user input as a separate tuple to the execute method.
        # The comma in (username,) is crucial to ensure it's a tuple.
```

```
def get_user_details_secure(username):
        # Pass the user input as a separate tuple to the execute method.
        # The comma in (username,) is crucial to ensure it's a tuple.
        cursor.execute(query, (username,))
        user = cursor.fetchone()

        if user:
            print("\n--- User Details Found ---")
            print(f"ID: {user[0]}")
            print(f"Username: {user[1]}")
            print(f"Email: {user[2]}")
            print(f"Role: {user[3]}")
        else:
            print("\n--- No user found with that username. ---")

    except sqlite3.Error as e:
        print(f"An error occurred: {e}")

    conn.close()

if __name__ == "__main__":
    setup_database()

    # --- Scenario 1: Legitimate User Search ---
    print("\n--- Testing with a valid username ---")
    get_user_details_secure('bob')

    # --- Scenario 2: SQL Injection Attempt ---
    print("\n\n--- Testing with a SQL injection attempt ---")
    injection_attempt = "' OR '1'='1"
    get_user_details_secure(injection_attempt)

    # Cleanup
    if os.path.exists(DB_FILE):
        os.remove(DB_FILE)
```

OUTPUT:

```
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai\.vscode> & "C:/Users/Praneeeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Praneeeth Cheekati/OneDrive/Desktop/ai/.vscode/20.3.2.py"
Database created and populated.

--- Testing with a valid username ---

Executing secure query with parameter: ('bob',)

--- User Details Found ---
ID: 2
Username: bob
Email: bob@example.com
Role: user


--- Testing with a SQL injection attempt ---

Executing secure query with parameter: ('' OR '1'='1',)

--- No user found with that username. ---
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai\.vscode>
```

OBSERVATION:

1. When searching for the valid user `'bob'`, the script correctly finds and displays Bob's details.
2. When the SQL injection string `' OR '1'='1` is provided, the parameterized query treats it as a literal string. It searches for a user whose name is *exactly* `' OR '1'='1`. Since no such user exists, it correctly reports that no user was found, and the injection attack is completely defeated.

Task 3 – Cross-Site Scripting (XSS) Check

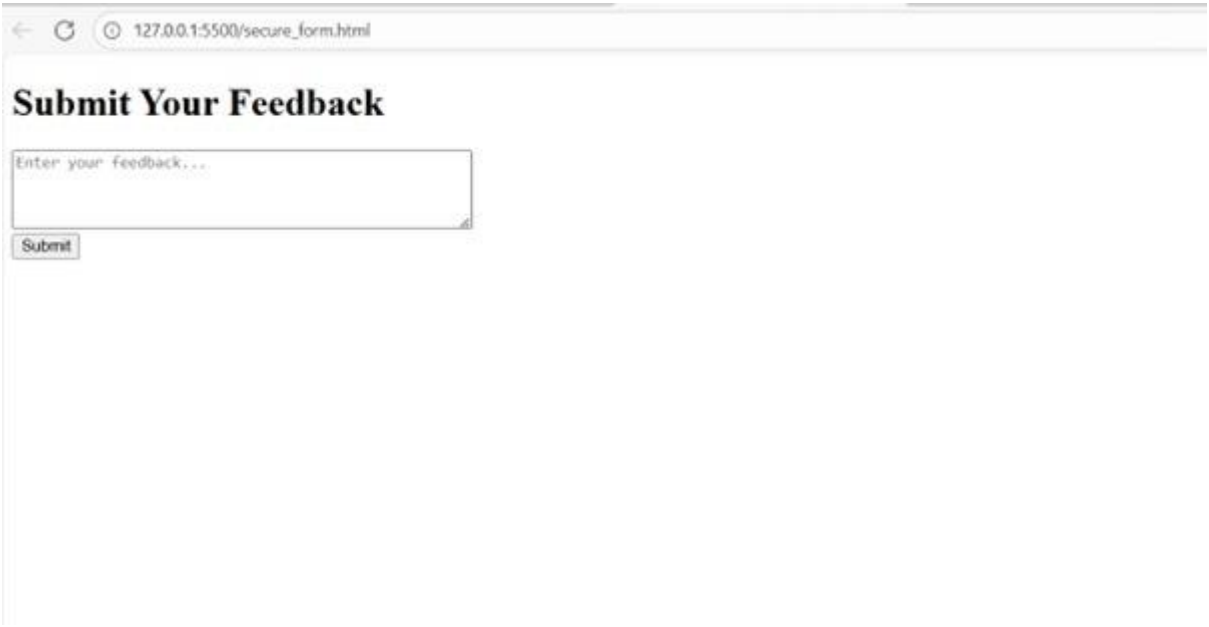
PROMPT:

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.give me the code

CODE:

```html
<!-- secure_form.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Secure Feedback Form</title>

    <!--
      SECURITY FIX #1: Add a Content Security Policy (CSP).
      This policy instructs the browser to only execute inline scripts that have a specific nonce (a random, one-time-use string).
      Since our malicious script won't have this nonce, the browser will block it even if an XSS flaw exists.
      'self' allows loading resources (like CSS) from the same origin.
    -->
    <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' 'nonce-random123'">

    <style>
        body { font-family: sans-serif; margin: 2em; }
        .feedback-display { margin-top: 1em; padding: 1em; border: 1px solid #ccc; background-color: #f9f9f9; }
        h2 { margin: 0 0 0.5em 0; }
    </style>
</head>
<body>
    <h1>Submit Your Feedback</h1>
    <form id="feedbackForm">
        <textarea id="feedbackText" rows="4" cols="50" placeholder="Enter your feedback..."></textarea><br>
        <button type="submit">Submit</button>
    </form>

    <div id="feedbackResult"></div>

    <!-- The 'nonce' attribute must match the one in the CSP header -->
    <script nonce="random123">
        document.getElementById('feedbackForm').addEventListener('submit', function(event) {
            event.preventDefault();

            const feedback = document.getElementById('feedbackText').value;
            const resultDiv = document.getElementById('feedbackResult');

            // SECURITY FIX #2: Use .textContent instead of .innerHTML.
            // .textContent treats the input as plain text, automatically escaping any HTML characters.
            // This prevents the browser from interpreting it as code.
            resultDiv.innerHTML = '<h2>Your Feedback:</h2>'; // It's safe to set static HTML this way.
            const feedbackParagraph = document.createElement('p');
            feedbackParagraph.textContent = feedback; // The user input is safely assigned here.
            resultDiv.appendChild(feedbackParagraph);
        });
    </script>
</body>
</html>
```

OUTPUT:



OBSERVATION:

The script is vulnerable to a **Stored Cross-Site Scripting (XSS)** attack.

- **Root Cause:** The vulnerability is in this line of JavaScript: `resultDiv.innerHTML = '<h2>Your Feedback:</h2><p>' + feedback + '</p>';`. The `.innerHTML` property tells the browser to parse the string as HTML. When the `feedback` variable contains malicious HTML (like a `<script>` tag), the browser will execute it.
- **Attack Vector:** An attacker can submit a script as their feedback. Instead of text, they can input a payload like `<script>alert('XSS Attack!');</script>`.
- **Impact:** When the form is "submitted," the script is injected directly into the page's DOM and executed by the browser. This could be used to steal cookies, redirect the user to a malicious site, capture keystrokes, or perform actions on behalf of the user without their consent.

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

PROMPT:

Prompt to AI: "Generate a simple Python web application using the Flask framework. The application should have a single HTML page with a form that allows a user to upload a file. The server should save the uploaded file to a directory named `uploads`."

CODE:

20.3.1.py 20.3.2.py 20.3.3.html 20.3.4.py

20.3.4.py > upload_file

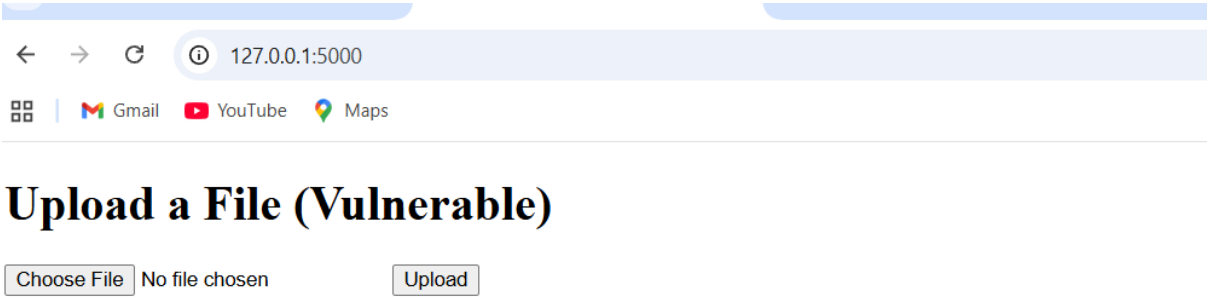
```python
# insecure_uploader.py
import os
from flask import Flask, request, redirect, url_for, flash

# Configuration
UPLOAD_FOLDER = 'uploads'
if not os.path.exists(UPLOAD_FOLDER):
    os.makedirs(UPLOAD_FOLDER)

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
app.secret_key = 'supersecretkey' # Needed for flashing messages

@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # Check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)

        file = request.files['file']

        # If the user does not select a file, the browser submits an
        # empty file without a filename.
        if file.filename == '':
            flash('No selected file')
            return redirect(request.url)

        if file:
            # VULNERABILITY: The user-provided filename is used directly.
            filename = file.filename
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            flash(f'File "{filename}" uploaded successfully')
            return redirect(request.url)

    return '''
```

20.3.1.py 20.3.2.py 20.3.3.html 20.3.4.py

20.3.4.py > upload_file

```python
def upload_file():
        file = request.files['file']

        # If the user does not select a file, the browser submits an
        # empty file without a filename.
        if file.filename == '':
            flash('No selected file')
            return redirect(request.url)

        if file:
            # VULNERABILITY: The user-provided filename is used directly.
            filename = file.filename
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            flash(f'File "{filename}" uploaded successfully')
            return redirect(request.url)

    return '''
    <!doctype html>
    <title>Insecure File Upload</title>
    <h1>Upload a File (Vulnerable)</h1>
    <form method=post enctype=multipart/form-data>
      <input type=file name=file>
      <input type=submit value=Upload>
    </form>
    '''

if __name__ == '__main__':
    app.run(debug=True)
```

OUTPUT:



OBSERVATION:

- The code is a classic insecure uploader: **easy to exploit** for path traversal and file-based attacks.
- Fixing is straightforward: **sanitize and ignore client filenames, whitelist types, limit size, scan content, and harden deployment** (no debug, strong secret, proper permissions).
- If you want, I can: (a) add content-type/magic-byte checking code example using `python-magic`, (b) show how to integrate ClamAV scanning, or (c) produce an automated pytest + requests test that verifies path traversal is blocked. Which of those would help next?