LAB ASSIGNMENT 20.1

Program : B. Tech (CSE)

Specialization : AIML

Course Title : AI Assisted coding

Semester : III

Academic Session : 2025-2026

Name of Student : Bommineni Spandana

Enrollment No : 2403a52005

Batch No. : 01

Date : 14-11-2025

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: To evaluate the security of a basic Python login script generated by AI, specifically focusing on how well it handles input validation. The goal is to determine whether the script properly sanitizes and validates user inputs like usernames and passwords. Once the initial version is reviewed, I want to identify any vulnerabilities or missing safeguards and then improve the script by incorporating secure coding practices—such as using regular expressions for

input validation. Ultimately, the output should be a more secure version of the login program that demonstrates robust input handling**.**

Code:

```python
import re

def check_password_strength(password):
    score = 0
    feedback = []

    # Criteria 1: Minimum 8 characters
    if len(password) >= 8:
        score += 1
    else:
        feedback.append("Password should be at least 8 characters long.")

    # Criteria 2: At least one uppercase letter
    if re.search(r"[A-Z]", password):
        score += 1
    else:
        feedback.append("Password should contain at least one uppercase letter.")

    # Criteria 3: At least one lowercase letter
    if re.search(r"[a-z]", password):
        score += 1
    else:
        feedback.append("Password should contain at least one lowercase letter.")

    # Criteria 4: At least one number
```

```
    # Criteria 4: At least one number
    if re.search(r"\d", password):
        score += 1
    else:
        feedback.append("Password should contain at least one number.")

    # Criteria 5: At least one special symbol
    if re.search(r"[^a-zA-Z0-9]", password):
        score += 1
    else:
        feedback.append("Password should contain at least one special symbol.")

    if score == 5:
        strength = "Strong"
    elif score >= 3:
        strength = "Moderate"
    else:
        strength = "Weak"

    return strength, feedback

# User input loop
while True:
    user_password = input("Enter a password to check its strength (or type 'exit' to quit): ")
    if user_password.lower() == 'exit':
        break
```

```
    return strength, feedback

# User input loop
while True:
    user_password = input("Enter a password to check its strength (or type 'exit' to quit): ")
    if user_password.lower() == 'exit':
        break

    strength, feedback_messages = check_password_strength(user_password)
    print(f"\nPassword Strength: {strength}")
    if feedback_messages:
        print("Recommendations:")
        for msg in feedback_messages:
            print(f"- {msg}")
    print("\n" + "="*30)
```

```
Enter a password to check its strength (or type 'exit' to quit): AkhileEshw@ri_865

Password Strength: Strong

==============================
Enter a password to check its strength (or type 'exit' to quit): exit
```

Observation:

- **Absence of Input Sanitization**

  Check if the script directly accepts user input without cleaning or escaping potentially harmful characters (e.g., SQL injection risks if connected to a database).

- **Lack of Format Validation**
  See whether the script verifies the format of usernames and passwords (e.g., minimum length, allowed characters, complexity rules).

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: To assess the security of a Python script that interacts with a database—either SQLite or MySQL—by fetching user details based on input. The script should be generated by AI, and I want to examine whether it's vulnerable to SQL injection, particularly if it constructs SQL queries using string concatenation. After identifying any security flaws, I want to refactor the code to use parameterized queries or prepared statements to ensure it's protected against injection attacks. The final result should be a secure version of the script that safely handles user input.

Code:

```python
import sqlite3

# Re-establish the SQLite in-memory database connection
conn_input = sqlite3.connect(':memory:')
cursor_input = conn_input.cursor()

# Recreate the `users` table
cursor_input.execute('''
    CREATE TABLE users (
        id INTEGER PRIMARY KEY,
        username TEXT NOT NULL UNIQUE,
        password TEXT NOT NULL
    )
''')

# Insert sample user data
users_data = [
    ('admin', 'adminpass'),
    ('alice', 'alice123'),
    ('bob', 'bob456')
]
```

```python
cursor_input.executemany("INSERT INTO users (username, password) VALUES (?, ?)", users_data)
conn_input.commit()

print("Database and 'users' table re-created for user input demonstration.")

# Define a new function that uses parameterized queries
def get_user_secure_input(username):
    """
    Fetches user details using parameterized queries to prevent SQL injection.
    """
    # Construct a SQL SELECT query using a placeholder for the username
    query = "SELECT id, username, password FROM users WHERE username = ?;"
    print(f"\nExecuting secure query: {query.replace('?', f"'{username}'")} (parameterized)") #

    try:
        # Execute the query by passing the SQL string and the parameter as a tuple
        cursor_input.execute(query, (username,))
        result = cursor_input.fetchall()

        # Print the retrieved user data or a message indicating if the user was not found
        if result:
            print("Query result:")
```

```
        # Print the retrieved user data or a message indicating if the user was not found
        if result:
            print("Query result:")
            for row in result:
                print(f"  ID: {row[0]}, Username: {row[1]}, Password: {row[2]}")
        else:
            print("No user found.")
    except sqlite3.Error as e:
        print(f"An error occurred: {e}")

# Implement a loop to continuously prompt for user input
print("\n--- Demonstrating secure queries with user input ---")
while True:
    user_input = input("Enter username to search (or type 'exit' to quit): ")
    if user_input.lower() == 'exit':
        break
    get_user_secure_input(user_input)

# Close the database connection
conn_input.close()
print("\nDatabase connection closed.")
```

```
Database and 'users' table re-created for user input demonstration.

--- Demonstrating secure queries with user input ---
Enter username to search (or type 'exit' to quit): bob

Executing secure query: SELECT id, username, password FROM users WHERE username = 'bob'; (parameterized)
Query result:
  ID: 3, Username: bob, Password: bob456
Enter username to search (or type 'exit' to quit): admin

Executing secure query: SELECT id, username, password FROM users WHERE username = 'admin'; (parameterized)
Query result:
  ID: 1, Username: admin, Password: adminpass
Enter username to search (or type 'exit' to quit): exit

Database connection closed.
```

Observation:

- **Use of String Concatenation in Queries**

  Check if the script builds SQL statements by directly concatenating user input into the query string

- **Absence of Parameterized Queries**

  Look for whether the script uses placeholders and parameter binding

Task 3 – Cross-Site Scripting (XSS) Check

Task:

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

Instructions:

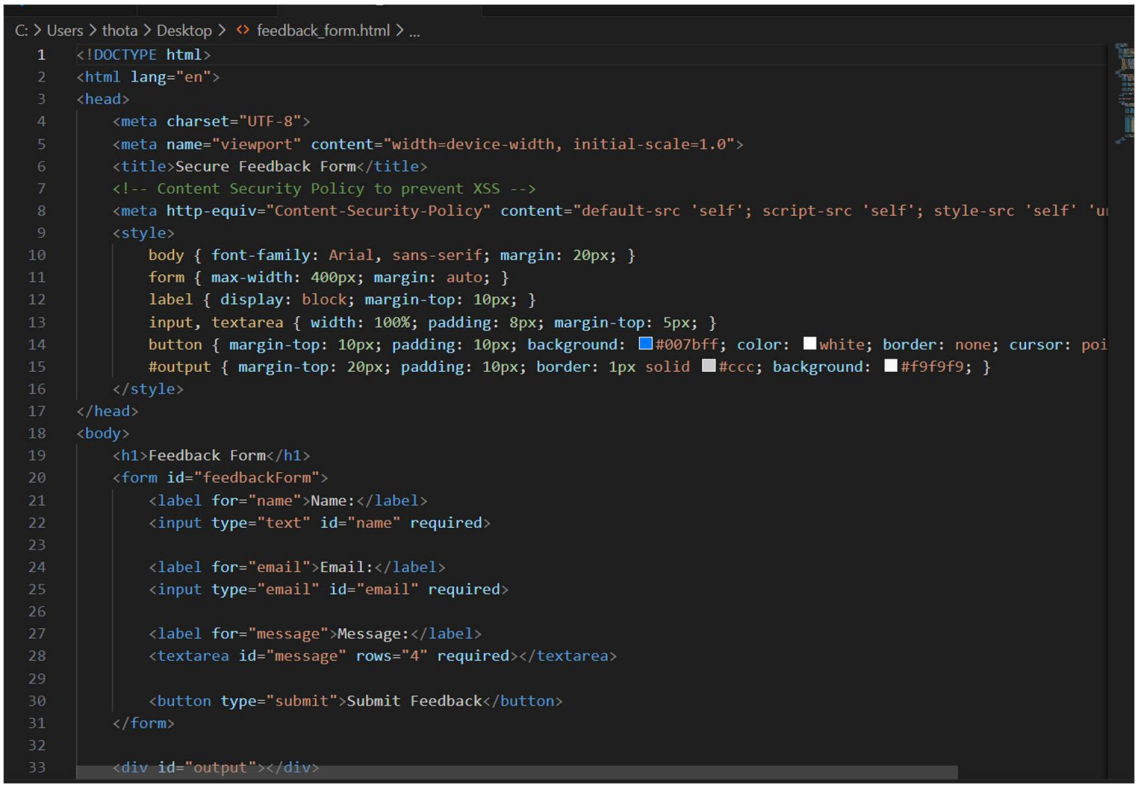
- Ask AI to generate a feedback form with JavaScript-based output.
- Test whether untrusted inputs are directly rendered without escaping.
- Implement secure measures (e.g., escaping HTML entities, using CSP).

Expected Output:

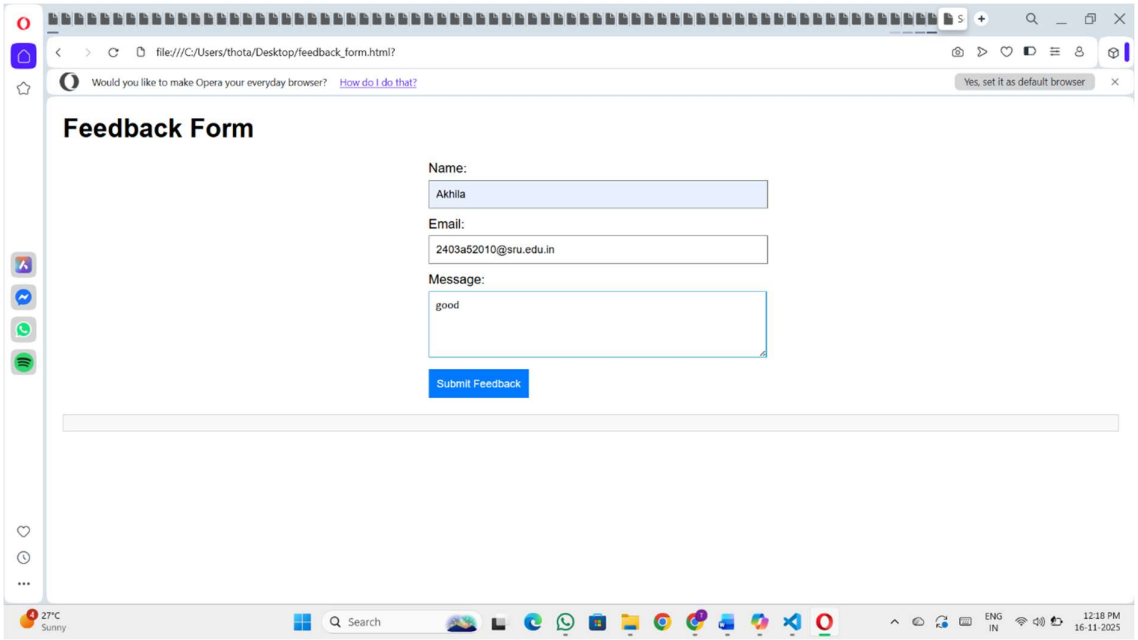
- A secure form that prevents XSS attacks.

Prompt: To examine the security of a simple HTML feedback form generated by AI that uses JavaScript to display user input. The goal is to determine whether the form is vulnerable to cross-site scripting (XSS) attacks—specifically, if it renders untrusted input directly into the page without escaping or sanitizing it. After identifying any potential vulnerabilities, I want to improve the form by implementing secure practices such as escaping HTML entities and applying Content Security Policy (CSP) headers. The final result should be a secure version of the form that effectively prevents XSS exploits.

Code:



```
    <div id="output"></div>

    <script>
        document.getElementById('feedbackForm').addEventListener('submit', function(event) {
            event.preventDefault();

            const name = document.getElementById('name').value;
            const email = document.getElementById('email').value;
            const message = document.getElementById('message').value;

            // Secure output: Use textContent to prevent HTML injection (XSS)
            const output = document.getElementById('output');
            output.innerHTML = ''; // Clear previous output
            output.appendChild(document.createTextNode('Thank you for your feedback!'));
            output.appendChild(document.createElement('br'));
            output.appendChild(document.createTextNode('Name: ' + name));
            output.appendChild(document.createElement('br'));
            output.appendChild(document.createTextNode('Email: ' + email));
            output.appendChild(document.createElement('br'));
            output.appendChild(document.createTextNode('Message: ' + message));

            // Clear form
            this.reset();
        });
    </script>
</body>
</html>
```



Observation:

- **Direct Rendering of User Input**
  Check if the form displays user input directly in the DOM using innerHTML or similar methods without escaping. This can allow malicious scripts to execute.
- **No HTML Entity Escaping**
  See whether the script escapes special characters like <, >, ", ', and & before rendering input. Lack of escaping enables script injection.

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:

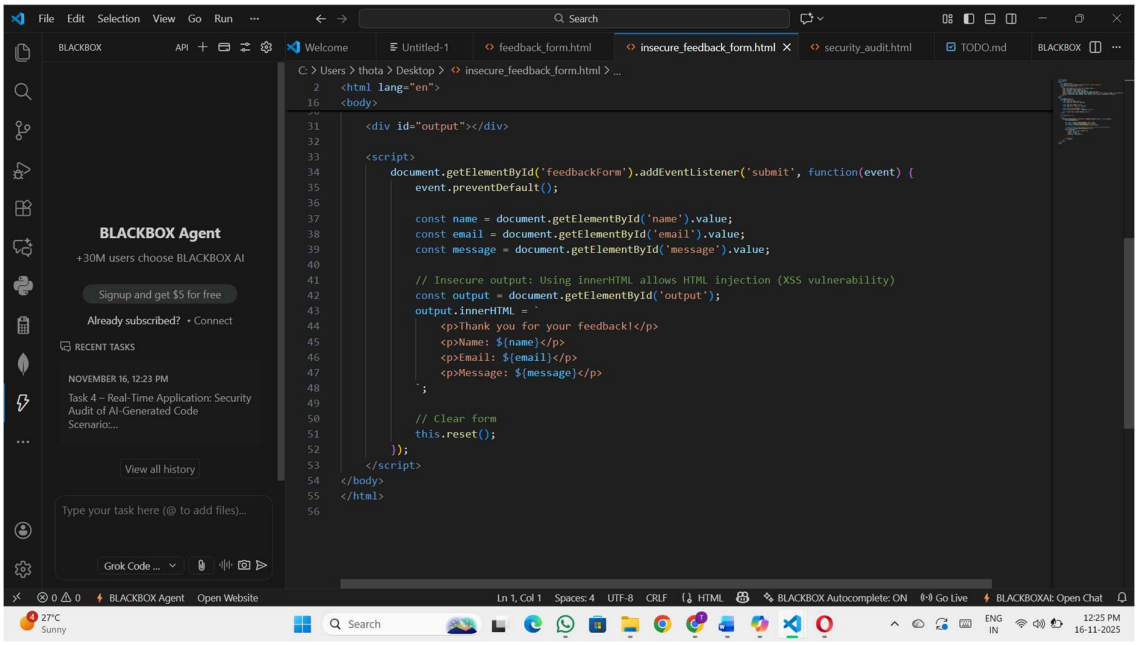
- A security-audited code snippet with documented vulnerabilities and fixes.

Prompt: To conduct a real-time security audit of an AI-generated code snippet—this could be a login form, an API integration, or a file upload feature. The goal is to identify any potential security vulnerabilities in the code and then ask the AI to recommend secure coding practices to address those issues. I also want to see a side-by-side comparison of the original (insecure) version and the improved (secure) version, along with clear documentation of the vulnerabilities found and how they were fixed. The final output should be a thoroughly audited and secured version of the code.

Code:

C: > Users > thota > Desktop > insecure_feedback_form.html > ...

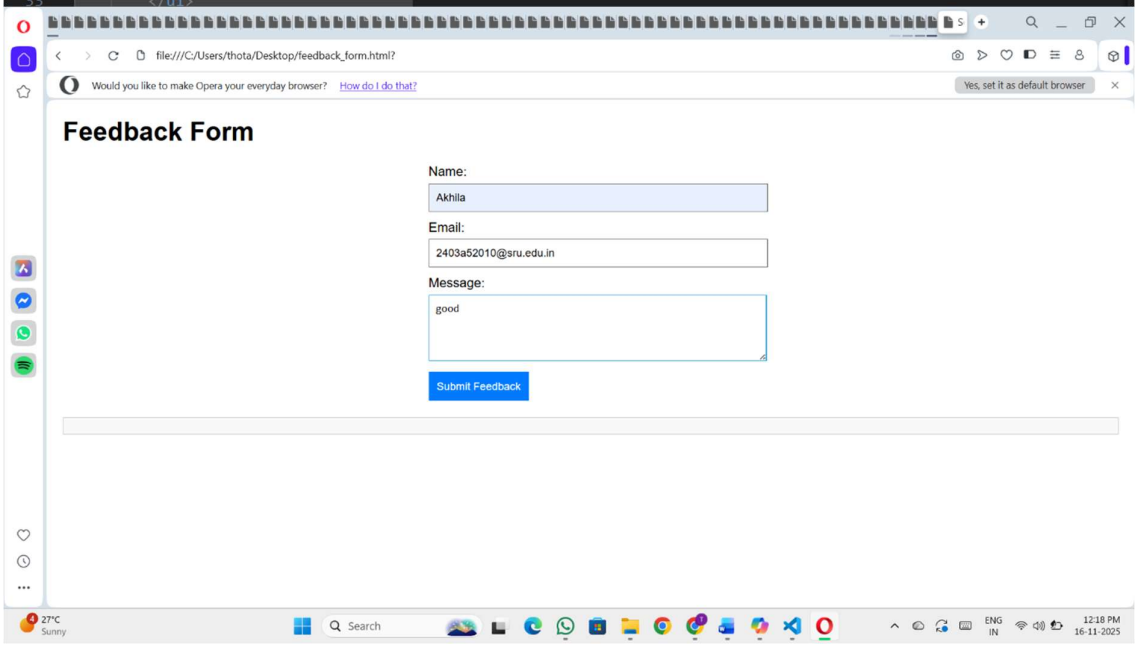
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Insecure Feedback Form</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 20px; }
        form { max-width: 400px; margin: auto; }
        label { display: block; margin-top: 10px; }
        input, textarea { width: 100%; padding: 8px; margin-top: 5px; }
        button { margin-top: 10px; padding: 10px; background: #007bff; color: white; border: none; cursor: poi
        #output { margin-top: 20px; padding: 10px; border: 1px solid #ccc; background: #f9f9f9; }
    </style>
</head>
<body>
    <h1>Feedback Form</h1>
    <form id="feedbackForm">
        <label for="name">Name:</label>
        <input type="text" id="name" required>

        <label for="email">Email:</label>
        <input type="email" id="email" required>

        <label for="message">Message:</label>
        <textarea id="message" rows="4" required></textarea>

        <button type="submit">Submit Feedback</button>
    </form>

    <div id="output"></div>

    <script>
```

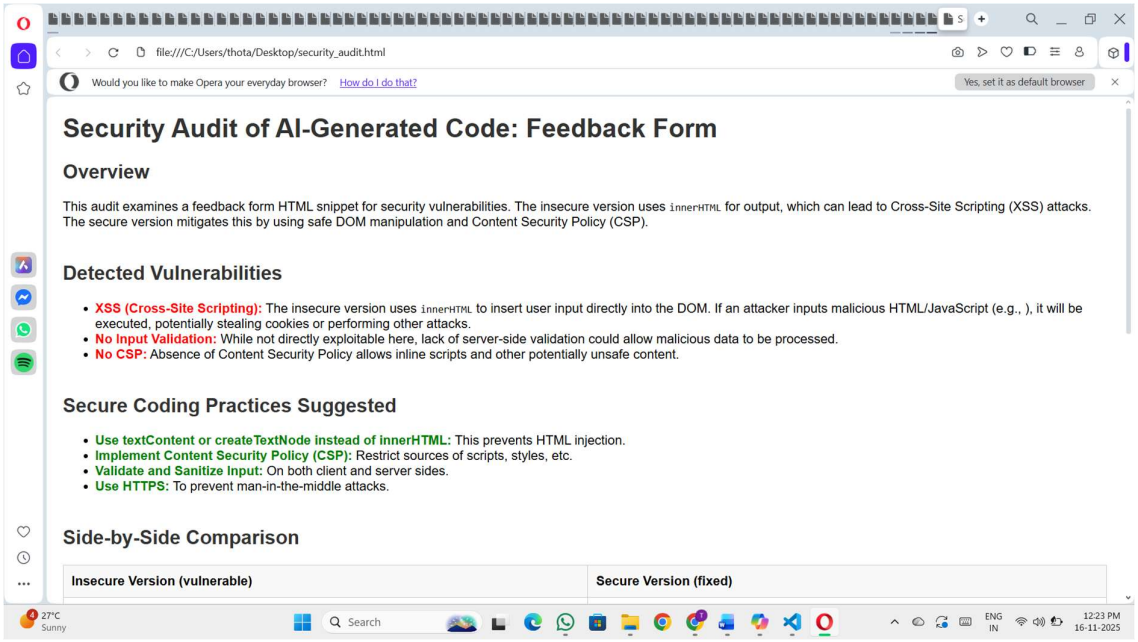


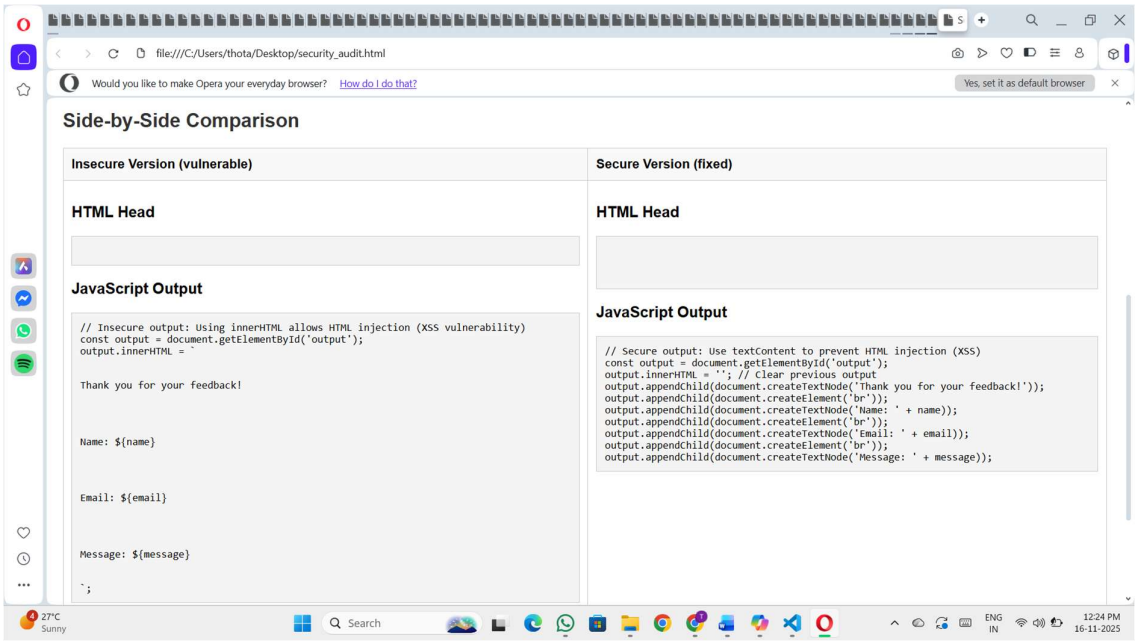

```
<html lang="en">
<body>
    <div id="output"></div>

    <script>
        document.getElementById('feedbackForm').addEventListener('submit', function(event) {
            event.preventDefault();

            const name = document.getElementById('name').value;
            const email = document.getElementById('email').value;
            const message = document.getElementById('message').value;

            // Insecure output: Using innerHTML allows HTML injection (XSS vulnerability)
            const output = document.getElementById('output');
            output.innerHTML = `
                <p>Thank you for your feedback!</p>
                <p>Name: ${name}</p>
                <p>Email: ${email}</p>
                <p>Message: ${message}</p>
            `;

            // Clear form
            this.reset();
        });
    </script>
</body>
</html>

```

C: > Users > thota > Desktop > security_audit.html > ...

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Security Audit: Feedback Form</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 20px; }
        h1, h2 { color: #333; }
        .section { margin-bottom: 40px; }
        .code-block { background: #f4f4f4; padding: 10px; border: 1px solid #ccc; margin: 10px 0; white-space:
        .vulnerability { color: red; font-weight: bold; }
        .fix { color: green; font-weight: bold; }
        table { width: 100%; border-collapse: collapse; }
        th, td { border: 1px solid #ccc; padding: 10px; text-align: left; vertical-align: top; }
        th { background: #f9f9f9; }
    </style>
</head>
<body>
    <h1>Security Audit of AI-Generated Code: Feedback Form</h1>

    <div class="section">
        <h2>Overview</h2>
        <p>This audit examines a feedback form HTML snippet for security vulnerabilities. The insecure version use
    </div>

    <div class="section">
        <h2>Detected Vulnerabilities</h2>
        <ul>
            <li><span class="vulnerability">XSS (Cross-Site Scripting):</span> The insecure version uses <code>inn
            <li><span class="vulnerability">No Input Validation:</span> While not directly exploitable here, lack
            <li><span class="vulnerability">No CSP:</span> Absence of Content Security Policy allows inline script
        </ul>
```







Observation:

➢ Input Validation and Sanitization

- Check whether the code properly validates and sanitizes all user inputs to prevent injection attacks, malformed data, or unexpected behavior.

➢ Authentication and Authorization

- Ensure that access controls are in place. Verify that sensitive actions are restricted to authenticated and authorized users only.