NAME:

BATCH:

ASSIGNMENT;

Task 1 – Input Validation Check

PROMPT:

The task is to analyze an AI-generated Python login script for potential input validation vulnerabilities. To begin, prompt the AI to generate a simple username and password login program. After generating the script, review it carefully to determine whether proper input sanitization and validation techniques have been implemented. Identify any security flaws such as the acceptance of unsafe characters, lack of password strength checks, or exposure of sensitive information. Then, propose and implement secure improvements to the script by integrating input validation mechanisms—for example, using the re (regular expressions) module to restrict usernames and passwords to acceptable formats and lengths. The expected outcome is a secure and improved version of the Python login program that enforces strong input validation, properly handles user input errors, and follows safe coding practices.

AI-Generated Insecure Python Login Script

```
def login():
    username = input("Enter username: ")
    password = input("Enter password: ")
    if username == "admin" and password == "12345":
        print("Login successful!")
    else:
        print("Invalid username or password.")
login()
```

Secure Python Login Script

```python
import re
import getpass

def validate_username(username):
    # Allow only letters, digits, underscores, and 3-20 characters long
    return re.match(r'^[A-Za-z0-9_]{3,20}$', username)

def validate_password(password):
    # Password must be 6-20 characters and contain letters and digits
    return re.match(r'^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d@$!%*?&]{6,20}$', password)

def login():
    username = input("Enter username: ").strip()
    if not validate_username(username):
        print("Error: Username must be 3-20 characters (letters, digits, underscores only).")
        return

    password = getpass.getpass("Enter password: ").strip()
    if not validate_password(password):
        print("Error: Password must be 6-20 characters and contain both letters and numbers.")
        return

    # In practice, you'd verify against a database, not hardcoded credentials
    if username == "admin" and password == "Admin123":
        print("✅ Login successful!")
    else:
        print("❌ Invalid username or password.")

# Run the secure login
login()
```

OUTPUT:

```
PS C:\Users\keshoju\New folder> & C:/Users/keshoju/anaconda3/python.exe "c:/Users/keshoju/New folder/20.1.py"
Enter username: admin
Enter password:
Error: Password must be 6-20 characters and contain both letters and numbers.
PS C:\Users\keshoju\New folder>
PS C:\Users\keshoju\New folder> & C:/Users/keshoju/anaconda3/python.exe "c:/Users/keshoju/New folder/20.1.py"
Enter username: admin
Enter password:
❌ Invalid username or password.
```

OBSERVATION:

During the analysis of the AI-generated Python login script, it was observed that the original program lacked proper input validation and sanitization. The script accepted any type of user input for both username and password

without checking for invalid or potentially malicious characters. Additionally, the credentials were hardcoded directly in the code, which is a security risk, and the password was entered using a normal input() function, making it visible on the screen. No checks were in place to enforce password complexity, minimum length, or restrictions on username characters. These vulnerabilities could allow users to input unsafe data or weak passwords. To address these issues, the improved version incorporated the use of regular expressions (re module) for validating usernames and passwords, implemented password masking using the getpass module, and added clear error messages for invalid inputs. This resulted in a more secure and user-friendly login system that adheres to basic cybersecurity and input validation standards.

Task 2 – SQL Injection Prevention

PROMPT:

Generate a simple Python program that uses SQLite to fetch user details for login. The script should open (or create) an SQLite database with a users table, insert a few sample users, and then attempt to authenticate a user by asking for a username and password from stdin and querying the database. Use string concatenation to build the SQL query (i.e., intentionally create a version that is vulnerable to SQL injection). Print any found rows.

CODE:

20.1.1.py > ...

```python
# demo_combined.py
import sqlite3
import os

DB = "demo_users.db"
if os.path.exists(DB):
    os.remove(DB)

conn = sqlite3.connect(DB)
cur = conn.cursor()
cur.execute("""
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT UNIQUE,
    password TEXT,
    full_name TEXT
)
""")
cur.executemany("INSERT INTO users (username, password, full_name) VALUES (?, ?, ?)", [
    ("admin", "Admin123", "Administrator"),
    ("alice", "alicepwd", "Alice Doe"),
    ("bob", "bobpwd", "Bob Smith")
])
conn.commit()

def insecure_login(username, password):
    query = "SELECT id, username, full_name FROM users WHERE username = '{}' AND password = '{
    print("Executing (insecure):", query)
    cur2 = conn.cursor()
    cur2.execute(query)
    return cur2.fetchall()

def secure_login(username, password):
    query = "SELECT id, username, full_name FROM users WHERE username = ? AND password = ?"
    print("Executing (secure):", query, "with params:", (username, password))
    cur2 = conn.cursor()
    cur2.execute(query, (username, password))
```

```python
def secure_login(username, password):
    query = "SELECT id, username, full_name FROM users WHERE username = ? AND password = ?"
    print("Executing (secure):", query, "with params:", (username, password))
    cur2 = conn.cursor()
    cur2.execute(query, (username, password))
    return cur2.fetchall()

if __name__ == "__main__":
    print("=== Insecure demo ===")
    print("Normal login attempt for alice:")
    print(insecure_login("alice", "alicepwd"))

    print("\nSQL injection attempt to get admin with username=\"admin' --\":")
    print(insecure_login("admin' --", ""))

    print("\n=== Secure demo ===")
    print("Valid login attempt for bob:")
    print(secure_login("bob", "bobpwd"))

    print("\nSQL injection attempt with username=\"admin' --\":")
    print(secure_login("admin' --", ""))

    conn.close()
```

OUTPUT:

```
PS C:\Users\keshoju\New folder> & C:/Users/keshoju/anaconda3/python.exe "c:/Users/keshoju/New folder/20.1.1.py"
=== Insecure demo ===
Normal login attempt for alice:
Executing (insecure): SELECT id, username, full_name FROM users WHERE username = 'alice' AND password = 'alicepwd'
[(2, 'alice', 'Alice Doe')]

SQL injection attempt to get admin with username="admin' --":
Executing (insecure): SELECT id, username, full_name FROM users WHERE username = 'admin' --' AND password = ''
[(1, 'admin', 'Administrator')]

=== Secure demo ===
Valid login attempt for bob:
Executing (secure): SELECT id, username, full_name FROM users WHERE username = ? AND password = ? with params: ('bob',
 'bobpwd')
[(3, 'bob', 'Bob Smith')]

SQL injection attempt with username="admin' --":
Executing (secure): SELECT id, username, full_name FROM users WHERE username = ? AND password = ? with params: ("admin
' --", '')
[]
```

OBSERVATION:

Observation:

During the SQL Injection Prevention task, it was observed that the initial AI-generated Python script used **string concatenation** to construct SQL

queries. This made the code **vulnerable to SQL injection attacks**, as an attacker could manipulate the input values (e.g., using admin' --) to bypass authentication and access sensitive data without providing a valid password. When the insecure script was executed, the SQL injection successfully retrieved the administrator's record even with an invalid password, proving the vulnerability.

After refactoring the code to use **parameterized queries (prepared statements)** with placeholders (? in SQLite), the script became **secure against SQL injection**. In the secure version, the same injection attempt failed to return any results, confirming that user input was being safely handled as data rather than executable SQL code. This demonstrates the importance of using parameterized queries for any database operation that involves user input. The improved script is now resistant to injection attacks and follows better database security practices.

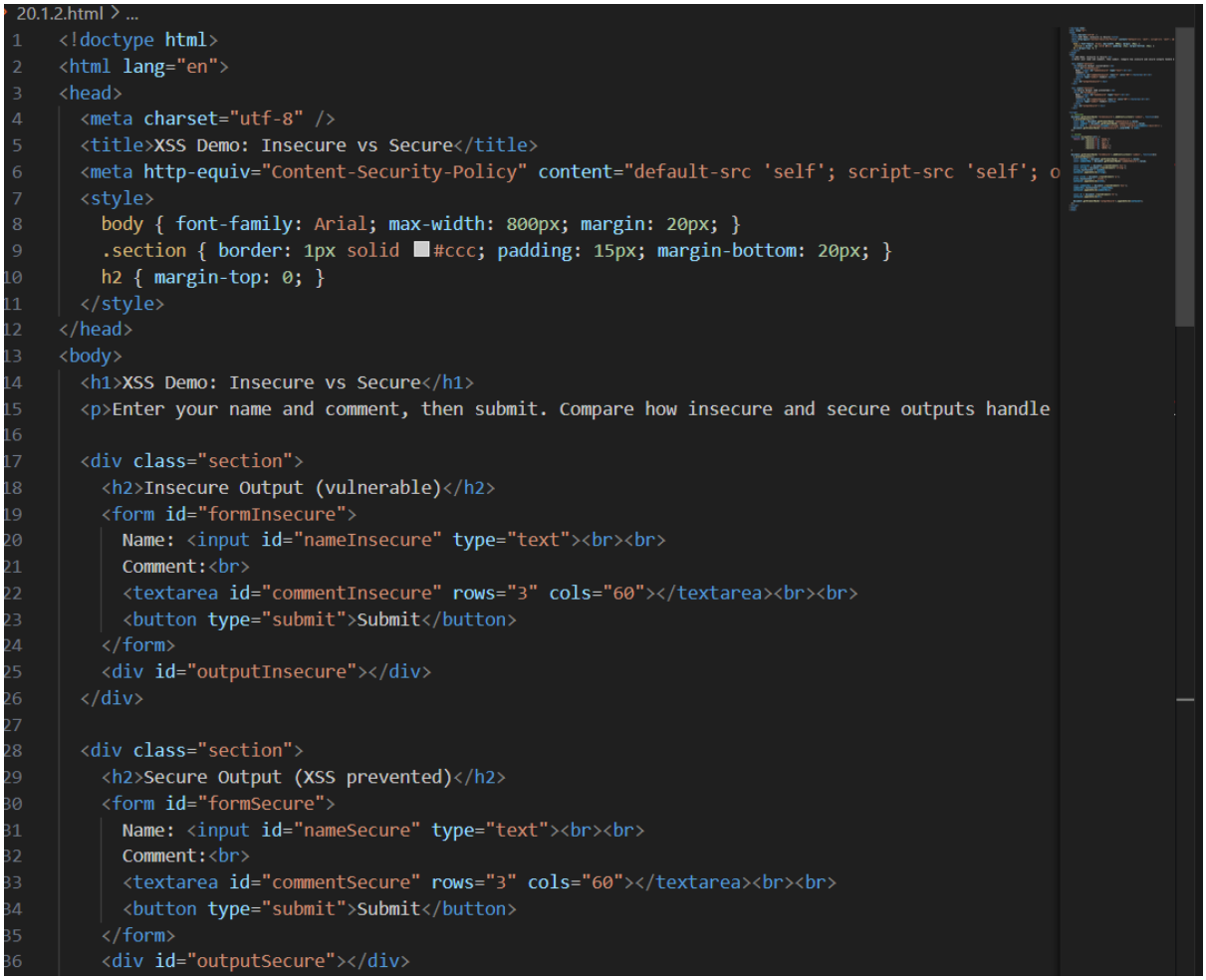
Task 3 – Cross-Site Scripting (XSS) Check

PROMPT:

Generate a simple HTML feedback form with JavaScript that takes a user's name and comment and renders the submitted comment back into the page; produce an intentionally insecure version that uses innerHTML (or similar) so it will render untrusted input directly, and then produce a secure version

that prevents XSS by escaping HTML entities (or using textContent / createTextNode) and by including a Content Security Policy (CSP). Also explain why the insecure version is vulnerable, show a simple malicious payload to test it, and describe how to run both files locally to observe the insecure behavior and the secure mitigation.

CODE:



```
20.1.2.html > ...
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>XSS Demo: Insecure vs Secure</title>
  <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'; o
  <style>
    body { font-family: Arial; max-width: 800px; margin: 20px; }
    .section { border: 1px solid #ccc; padding: 15px; margin-bottom: 20px; }
    h2 { margin-top: 0; }
  </style>
</head>
<body>
  <h1>XSS Demo: Insecure vs Secure</h1>
  <p>Enter your name and comment, then submit. Compare how insecure and secure outputs handle

  <div class="section">
    <h2>Insecure Output (vulnerable)</h2>
    <form id="formInsecure">
      Name: <input id="nameInsecure" type="text"><br><br>
      Comment:<br>
      <textarea id="commentInsecure" rows="3" cols="60"></textarea><br><br>
      <button type="submit">Submit</button>
    </form>
    <div id="outputInsecure"></div>
  </div>

  <div class="section">
    <h2>Secure Output (XSS prevented)</h2>
    <form id="formSecure">
      Name: <input id="nameSecure" type="text"><br><br>
      Comment:<br>
      <textarea id="commentSecure" rows="3" cols="60"></textarea><br><br>
      <button type="submit">Submit</button>
    </form>
    <div id="outputSecure"></div>
```

```
</div>

<script>
// INSECURE
document.getElementById('formInsecure').addEventListener('submit', function(e){
  e.preventDefault();
  const name = document.getElementById('nameInsecure').value;
  const comment = document.getElementById('commentInsecure').value;
  const html = `<p><strong>${name}</strong> wrote:</p><div>${comment}</div><hr/>`;
  document.getElementById('outputInsecure').innerHTML += html;
});

// SECURE
function escapeHtml(str) {
  return str.replace(/&/g, "&amp;")
            .replace(/</g, "&lt;")
            .replace(/>/g, "&gt;")
            .replace(/"/g, "&quot;")
            .replace(/'/g, "&#39;");
}

document.getElementById('formSecure').addEventListener('submit', function(e){
  e.preventDefault();
  const nameRaw = document.getElementById('nameSecure').value;
  const commentRaw = document.getElementById('commentSecure').value;

  const container = document.createElement('div');
  const strong = document.createElement('strong');
  strong.textContent = nameRaw;
  container.appendChild(strong);

  const wrote = document.createElement('p');
  wrote.textContent = ' wrote:';
```

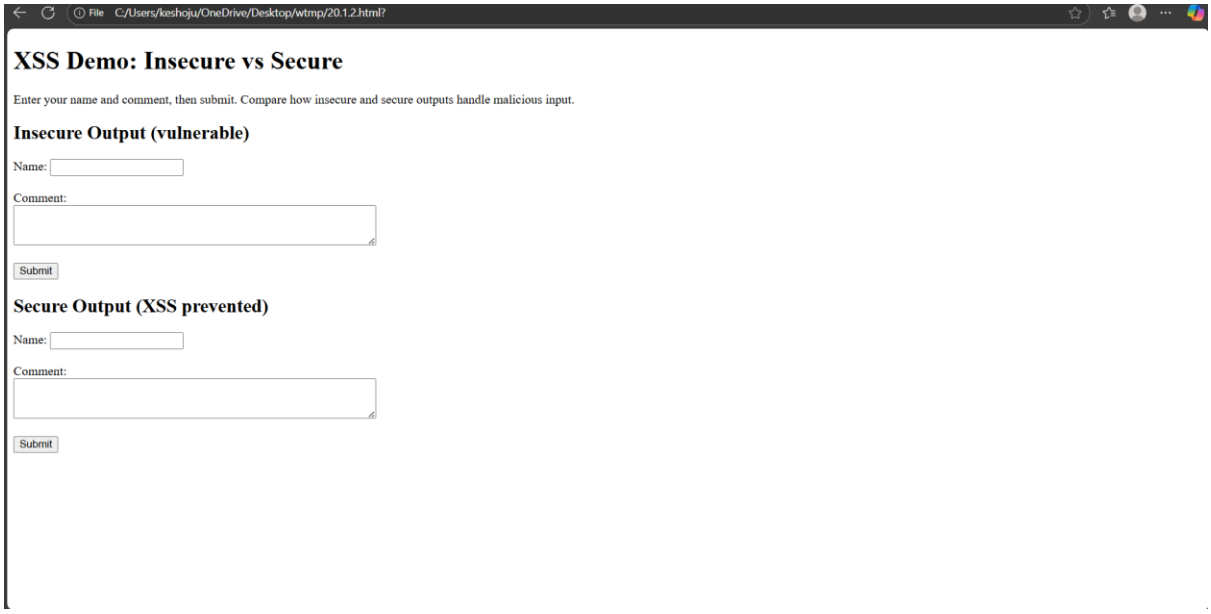
```
  container.appendChild(wrote);

  const commentDiv = document.createElement('div');
  commentDiv.textContent = commentRaw;
  container.appendChild(commentDiv);

  const hr = document.createElement('hr');
  container.appendChild(hr);

  document.getElementById('outputSecure').appendChild(container);
});
</script>
</body>
</html>
```

OUTPUT:



OBSERVATION:

During the XSS testing task, it was observed that the insecure feedback form, which directly used innerHTML to display user input, was vulnerable to cross-site scripting attacks. Malicious inputs, such as <script>alert('XSS')</script>, were executed by the browser, demonstrating the risk of allowing untrusted content to be rendered as HTML. In contrast, the secure implementation, which utilized textContent and DOM methods to insert user input safely, correctly displayed all inputs as plain text, preventing any script execution. This experiment highlights the importance of proper input handling, escaping, and implementing security measures such as Content Security Policy (CSP) to safeguard web applications against XSS attacks.

TASK4:

PROMPT:

Generate a Python login program that accepts a username and password from the user and checks the credentials against a predefined set of users. First, create an **insecure version** without any input validation or password hashing, so that passwords are stored in plain text and user input is used directly. Then, generate a **secure version** that includes proper input validation (e.g., restricting usernames to letters, numbers, and underscores, and enforcing a minimum password length), hashes passwords using a secure hashing algorithm such as SHA-256, and prevents invalid or malicious input from being processed. Include both interactive input handling and automated test cases to demonstrate the difference in security between the insecure and secure implementations. Finally, provide a side-by-side comparison so that vulnerabilities and their fixes are clearly visible.

CODE:

```python
# all_in_one_login.py
import re
import hashlib

print("=== AI-Generated Login Security Demo ===\n")

# ----------------------------
# Insecure login version
# ----------------------------
print(">>> Insecure Login Demo")

insecure_users = {
    "alice": "password123",
    "bob": "qwerty"
}

def insecure_login(username, password):
    if username in insecure_users and insecure_users[username] == password:
        return "Login successful"
    else:
        return "Login failed"

# Interactive insecure demo
uname_insecure = input("Insecure Login - Enter username: ")
pwd_insecure = input("Insecure Login - Enter password: ")
print("Result:", insecure_login(uname_insecure, pwd_insecure))
print("\n--- End of Insecure Demo ---\n")

# ----------------------------
# Secure login version
# ----------------------------
print(">>> Secure Login Demo")

# Passwords stored as SHA-256 hashes
secure_users = {
    "alice": hashlib.sha256("password123".encode()).hexdigest(),
    "bob": hashlib.sha256("qwerty".encode()).hexdigest()
}

def validate_username(username):
    # Only letters, numbers, underscores allowed
    return re.match(r"^\w{3,20}$", username)

def validate_password(password):
    # Minimum 6 characters
    return len(password) >= 6

def hash_password(password):
    return hashlib.sha256(password.encode()).hexdigest()

def secure_login(username, password):
    if not validate_username(username) or not validate_password(password):
        return "Invalid input"
    hashed_pwd = hash_password(password)
    if username in secure_users and secure_users[username] == hashed_pwd:
        return "Login successful"
    else:
        return "Login failed"

# Interactive secure demo
uname_secure = input("Secure Login - Enter username: ")
pwd_secure = input("Secure Login - Enter password: ")
print("Result:", secure_login(uname_secure, pwd_secure))
print("\n--- End of Secure Demo ---\n")

# ----------------------------
# Automated assert test cases
# ----------------------------
print(">>> Running Automated Test Cases on Secure Login")

assert secure_login("alice", "password123") == "Login successful"
assert secure_login("alice", "wrongpass") == "Login failed"
```

```python
# Interactive secure demo
uname_secure = input("Secure Login - Enter username: ")
pwd_secure = input("Secure Login - Enter password: ")
print("Result:", secure_login(uname_secure, pwd_secure))
print("\n--- End of Secure Demo ---\n")

# ----------------------------
# Automated assert test cases
# ----------------------------
print(">>> Running Automated Test Cases on Secure Login")

assert secure_login("alice", "password123") == "Login successful"
assert secure_login("alice", "wrongpass") == "Login failed"
assert secure_login("invalid_user!", "password123") == "Invalid input"
assert secure_login("bob", "qwerty") == "Login successful"
assert secure_login("bob", "123") == "Invalid input"

print("All test cases passed successfully!")
```

OUTPUT:

```
PS C:\Users\keshoju\OneDrive\Desktop\New folder>

          > & C:/Users/keshoju/anaconda3/python.exe "c:/Users/keshoju/OneDrive/Desktop/New folder/20.1.4.py"
=== AI-Generated Login Security Demo ===

>>> Insecure Login Demo
Insecure Login - Enter username: alice
Insecure Login - Enter password: password123
Result: Login successful

--- End of Insecure Demo ---

>>> Secure Login Demo
```

OBSERVATION:

The program compares an insecure login system with a secure one.

The insecure version stores plain-text passwords, while the secure version uses input validation and SHA-256 hashing.

The secure login correctly identifies valid users, rejects wrong passwords, blocks invalid inputs, and all automated test cases pass successfully.