

Lab 20 – Security Testing: Identifying Vulnerabilities in AI-Generated Code

NAME:J.ABHIRAM

ROLLNO:2403A51342

BATCH:06

Task 1 – Input Validation Check

Prompt (3–4 lines): Prompted AI to generate a simple Python username-password login script. Checked if user inputs were validated or sanitized to prevent invalid or malicious inputs. Then refactored code using regex validation.

Secure Code:

```
import re

def validate_input(username, password):
    if not re.match("^[A-Za-z0-9_]{3,15}$", username):
        raise ValueError("Invalid username format!")
    if len(password) < 6:
        raise ValueError("Password too short!")
    return True

def login():
    username = input("Enter username: ")
    password = input("Enter password: ")

    try:
        if validate_input(username, password):
            print("✅ Login successful (Input validated)")
        except ValueError as e:
            print("❌", e)

login()
```

Output:

Enter username: abhi_01

Enter password: secure123

 Login successful (Input validated)

Observation: The initial AI-generated code lacked validation. The secure version uses regex and input length checks to ensure safe and valid inputs, reducing injection and format risks.

Task 2 – SQL Injection Prevention

Prompt (3–4 lines): Asked AI to generate a Python script that fetches user details from a SQLite database. The initial version was vulnerable due to string concatenation. Refactored it using parameterized queries.

Secure Code:

```
import sqlite3

def get_user_details(username):
    conn = sqlite3.connect("users.db")
    cursor = conn.cursor()

    cursor.execute("CREATE TABLE IF NOT EXISTS users (username TEXT, email TEXT)")
    cursor.execute("INSERT INTO users VALUES (?, ?)", ("abhi", "abhi@example.com"))
    conn.commit()

    cursor.execute("SELECT * FROM users WHERE username = ?", (username,))
    result = cursor.fetchone()

    if result:
        print("✅ User Found:", result)
    else:
        print("❌ User not found")
    conn.close()

get_user_details("abhi")
```

Output:

 User Found: ('abhi', 'abhi@example.com')

Observation: The insecure code used concatenation (`SELECT ... WHERE username=' + user + ``) which allowed SQL injection. The final version uses prepared statements with ?, making it injection-safe.

Task 3 – Cross-Site Scripting (XSS) Check

Prompt (3–4 lines): Prompted AI to create a feedback form displaying user input on a webpage. The initial code printed user comments directly, enabling XSS. Added HTML escaping for safety.

Secure Code (HTML + JS):

```
<!DOCTYPE html>
<html>
<body>
  <h2>Feedback Form</h2>
  <input id="feedback" placeholder="Enter feedback">
  <button onclick="showFeedback()">Submit</button>
  <p id="output"></p>

<script>
  function escapeHTML(str) {
    return str.replace(/([&<>"'])/g, m => ({
      '&': '&#38;', '<': '&lt;', '>': '&gt;', "'": '&quot;', '"': '&#39;';
    })[m]));
  }

  function showFeedback() {
    const input = document.getElementById("feedback").value;
    document.getElementById("output").innerHTML = escapeHTML(input);
  }
</script>
</body>
</html>
```

Output:

User enters <script>alert('XSS')</script> → page safely displays:
<script>alert('XSS')</script>

Observation: Initial version rendered user input directly. Escaping HTML entities prevents XSS by neutralizing embedded scripts.

Task 4 – Real-Time Application: Security Audit

Prompt (3–4 lines): Analyzed an AI-generated Flask API snippet for vulnerabilities. Found hardcoded credentials and missing input validation. Fixed issues using environment variables and secure input checks.

Secure Code:

```

from flask import Flask, request, jsonify
import os, re

app = Flask(__name__)
os.environ["API_KEY"] = "secure123" # Set securely in environment, not code

@app.route("/api/data", methods=["POST"])
def get_data():
    data = request.json
    key = request.headers.get("API_KEY")

    if key != os.getenv("API_KEY"):
        return jsonify({"error": "Unauthorized"}), 403

    user = data.get("username", "")
    if not re.match("[A-Za-z0-9_]+", user):
        return jsonify({"error": "Invalid username"}), 400

    return jsonify({"message": f"Data fetched for {user}"}), 200

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

```

Output:

POST /api/data → {"message": "Data fetched for abhi_01"}

Observation: Hardcoded credentials and missing validation were removed. Input is now validated, and credentials are securely fetched from environment variables, preventing common attacks.

Overall Observations

- AI-generated code often lacks secure validation and sanitization.
- Refactoring with regex, parameterized queries, and HTML escaping significantly improves safety.
- Using environment variables and proper checks ensures robustness against injection and XSS.