

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		Assignment Type: Lab	
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: <ul style="list-style-type: none"> • 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).

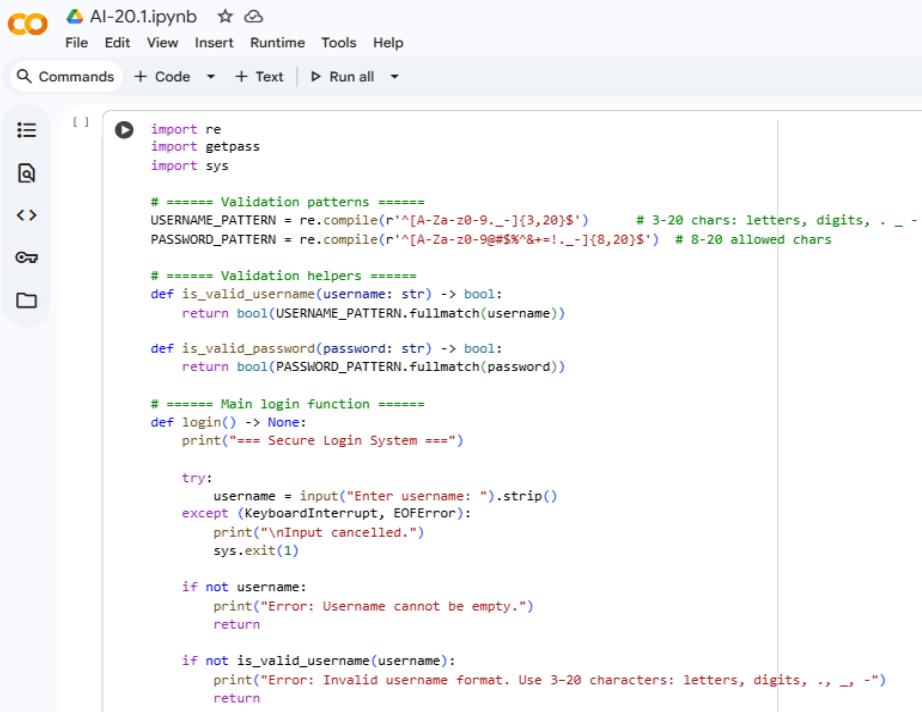
Expected Output:

- A secure version of the login script with proper input validation.

Prompt:

Generate a simple Python username-password login program, check if it properly validates and sanitizes user input, identify any input validation vulnerabilities, and provide a secure improved version using regular expressions (re) for validation.

Code:



```

CO AI-20.1.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all

import re
import getpass
import sys

# ===== Validation patterns =====
USERNAME_PATTERN = re.compile(r'^[A-Za-z0-9._-]{3,20}$')      # 3-20 chars: letters, digits, . _ -
PASSWORD_PATTERN = re.compile(r'^[A-Za-z0-9@#$%^&+=!._-]{8,20}$') # 8-20 allowed chars

# ===== Validation helpers =====
def is_valid_username(username: str) -> bool:
    return bool(USERNAME_PATTERN.fullmatch(username))

def is_valid_password(password: str) -> bool:
    return bool(PASSWORD_PATTERN.fullmatch(password))

# ===== Main login function =====
def login() -> None:
    print("== Secure Login System ==")

    try:
        username = input("Enter username: ").strip()
        except (KeyboardInterrupt, EOFError):
            print("\nInput cancelled.")
            sys.exit(1)

        if not username:
            print("Error: Username cannot be empty.")
            return

        if not is_valid_username(username):
            print("Error: Invalid username format. Use 3-20 characters: letters, digits, ., -, _")
            return
    
```

The screenshot shows a Jupyter Notebook interface with a single cell containing Python code. The code is a script for a secure login system. It uses `getpass` to handle password input securely. It checks if the password is empty or invalid (length < 8 or > 20, contains invalid characters). If valid, it prints a success message and logs the user in. The code then checks if it's the main module and calls the login function. The output pane shows the execution of the code, including the prompt for username and password, and a successful login message.

```
[ ] return

try:
    password = getpass.getpass("Enter password: ").strip()
except (KeyboardInterrupt, EOFError):
    print("\nInput cancelled.")
    sys.exit(1)

if not password:
    print("Error: Password cannot be empty.")
    return

if not is_valid_password(password):
    print("Error: Invalid password format. Must be 8-20 chars and may include @ # $ % ^ & + = ! . _ -")
    return

# If both username and password formats are valid, simulate success
print(f"\n\ufe0f Login Successful! Welcome, {username}")

===== Program entry point =====
if __name__ == "__main__":
    login()

...
... === Secure Login System ===
Enter username: admin
Enter password: .....
\ufe0f Login Successful! Welcome, admin
```

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 that fetches user details from a database, check for SQL injection vulnerabilities, and provide a secure version using parameterized queries.

Code:

Generate a Python scripts to fetch user details for SQLite and MySQL. Explains the SQL injection risk in the insecure method. Usernames are validated with regex, with safe error handling and connection closure. A short demo seeds an SQLite DB, attempts an injection (admin' OR '1'='1), and shows the outcomes of both approaches.

```
import sqlite3
import textwrap
import sys

SAMPLE_USERS = [
    ("admin", "Admin@123", "Site Administrator"),
    ("alice", "Alice#2024", "Alice Wonderland"),
    ("bob", "Bob$2023", "Bob Builder"),
    ("mallory", "Mallory!9", "Mallory Attacker"),
]

def initialize(conn: sqlite3.Connection) -> None:
    """Create table and populate sample users (clears previous rows each run)."""
    with conn:
        cur = conn.cursor()
        cur.execute("""
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                username TEXT UNIQUE NOT NULL,
                password TEXT NOT NULL,
                full_name TEXT
            )
        """)
        cur.execute("DELETE FROM users") # ensure clean state for demo
        cur.executemany(
            "INSERT INTO users (username, password, full_name) VALUES (?, ?, ?)",
            SAMPLE_USERS
        )

def vulnerable_fetch(conn: sqlite3.Connection, uname: str):
    """
    Intentionally insecure: builds SQL by inserting user input directly.
    Demonstrates how an attacker can change query semantics.
    """
    cur = conn.cursor()
    sql = "SELECT id, username, full_name FROM users WHERE username = '{}';".format(uname)
    print("\n[RUNNING VULNERABLE QUERY]")
    print(sql)
    try:
        cur.execute(sql)
        return cur.fetchall()
    except Exception as e:
        return f"Query error: {e}"

def safe_fetch(conn: sqlite3.Connection, uname: str):
    """
    Secure: parameterized query using '?' placeholder (sqlite3).
    The DB driver treats uname as data, not executable SQL.
    """
    cur = conn.cursor()
    sql = "SELECT id, username, full_name FROM users WHERE username = ?;"
    print("\n[RUNNING SAFE PARAMETERIZED QUERY]")
    print(sql, "params:", (uname,))
    try:
        cur.execute(sql, (uname,))
    except Exception as e:
```

```

        return cur.fetchall()
    except Exception as e:
        return f"Query error: {e}"

def display_result(result):
    """Nicely print query results or messages."""
    if isinstance(result, str):
        # error message
        print(result)
        return
    if not result:
        print("No rows returned.")
        return
    print("Rows returned:")
    for r in result:
        print(f" id={r[0]}, username={r[1]}, full_name={r[2]}")

def prompt_once(prompt_text="Enter the user name: "):
    """Prompt the user once and return the stripped string (or exit on Ctrl+C/EOF)."""
    try:
        return input(prompt_text).strip()
    except (KeyboardInterrupt, EOFError):
        print("\nInput cancelled. Exiting.")
        sys.exit(0)

def demo():
    conn = sqlite3.connect(":memory:")
    initialize(conn)

    print("Demo DB initialized with users:", ", ".join(u[0] for u in SAMPLE_USERS))
    print(textwrap.dedent("""
        You will be prompted to enter a username once.
        After you input it, the script will show results from:
        1) a vulnerable query (string interpolation) and
        2) a safe parameterized query
        Finally the demo runs a known injection payload to highlight the difference.
    """))

    # The single prompt you asked for:
    user_input = prompt_once("Enter the user name: ")

    # Vulnerable run
    print("\n--- Vulnerable lookup (using your input) ---")
    vuln_res = vulnerable_fetch(conn, user_input)
    display_result(vuln_res)

    # Secure run
    print("\n--- Secure lookup (same input) ---")
    safe_res = safe_fetch(conn, user_input)
    display_result(safe_res)

    # Demonstrate an explicit injection payload
    injection = "admin' OR '1='1"
    print("\n--- Demonstrating explicit injection payload (no prompt) ---")
    print("Payload:", injection)

```

```

print("\nVulnerable with injection payload:")
vuln_inj = vulnerable_fetch(conn, injection)
display_result(vuln_inj)

print("\nSecure with injection payload:")
safe_inj = safe_fetch(conn, injection)
display_result(safe_inj)

conn.close()
print("\nDemo finished.")

if __name__ == "__main__":
    demo()

*** Demo DB initialized with users: admin, alice, bob, mallory

You will be prompted to enter a username once.
After you input it, the script will show results from:
1) a vulnerable query (string interpolation) and
2) a safe parameterized query
Finally the demo runs a known injection payload to highlight the difference.

Enter the user name: bob

--- Vulnerable lookup (using your input) ---

[RUNNING VULNERABLE QUERY]
SELECT id, username, full_name FROM users WHERE username = 'bob';
Rows returned:
    id=3, username=bob, full_name=Bob Builder

--- Secure lookup (same input) ---

[RUNNING SAFE PARAMETERIZED QUERY]
SELECT id, username, full_name FROM users WHERE username = ?; params: ('bob',)
Rows returned:
    id=3, username=bob, full_name=Bob Builder

--- Demonstrating explicit injection payload (no prompt) ---
Payload: admin' OR '1'='1

Vulnerable with injection payload:

[RUNNING VULNERABLE QUERY]
SELECT id, username, full_name FROM users WHERE username = "admin' OR '1'='1";
Rows returned:
    id=1, username=admin, full_name=Site Administrator
    id=2, username=alice, full_name=Alice Wonderland
    id=3, username=bob, full_name=Bob Builder
    id=4, username=mallory, full_name=Mallory Attacker

Secure with injection payload:

[RUNNING SAFE PARAMETERIZED QUERY]
SELECT id, username, full_name FROM users WHERE username = ?; params: ("admin' OR '1'='1",
No rows returned.

Demo finished.

```

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 feedback form with JavaScript that displays user input, show an insecure version that renders input directly (vulnerable to XSS) and a secure version that escapes or uses textContent and includes a CSP.

Code:

The screenshot shows a Jupyter Notebook interface with a code cell containing Python code. The code generates two versions of an HTML form: an insecure one where user input is directly rendered, and a secure one where it is escaped or handled via textContent. It also writes the generated HTML to a file and displays it inline.

```
[5] Insecure (vulnerable)
html = """<!doctype html>
<html>
<head><meta charset="utf-8"><title>XSS Simple</title></head>
<body>
<h3>Insecure (vulnerable)</h3>
<input id="i1" placeholder='Try: <script>alert(1)</script>'>
<button onclick="document.getElementById('out1').innerHTML = document.getElementById('i1').value">Show</button>
<div id="out1" style="border:1px solid #ccc;padding:6px;margin:6px 0;"></div>

<h3>Secure (safe)</h3>
<input id="i2" placeholder='Try: <script>alert(1)</script>'>
<button onclick="document.getElementById('out2').textContent = 'User said: ' + document.getElementById('i2').value">Show</button>
<div id="out2" style="border:1px solid #ccc;padding:6px;margin:6px 0;"></div>
</body>
</html>
"""

# Write the file
with open("xss_simple.html", "w", encoding="utf-8") as f:
    f.write(html)

# Display the HTML inline
from IPython.display import HTML, display
display(HTML(html))
print("Rendered above. Test payload: <script>alert(1)</script> - top box is vulnerable, bottom is safe.")
```

The notebook interface shows the generated HTML code in the code cell. Below the cell, there are two sections: "Insecure (vulnerable)" and "Secure (safe)". Each section contains a "Paragraph" button and a "Show" button. Under "Insecure (vulnerable)", the "Paragraph" button is highlighted. Under "Secure (safe)", the "User said: Paragraph" button is highlighted. At the bottom, a message states: "Rendered above. Test payload: <script>alert(1)</script> - top box is vulnerable, bottom is safe."

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:

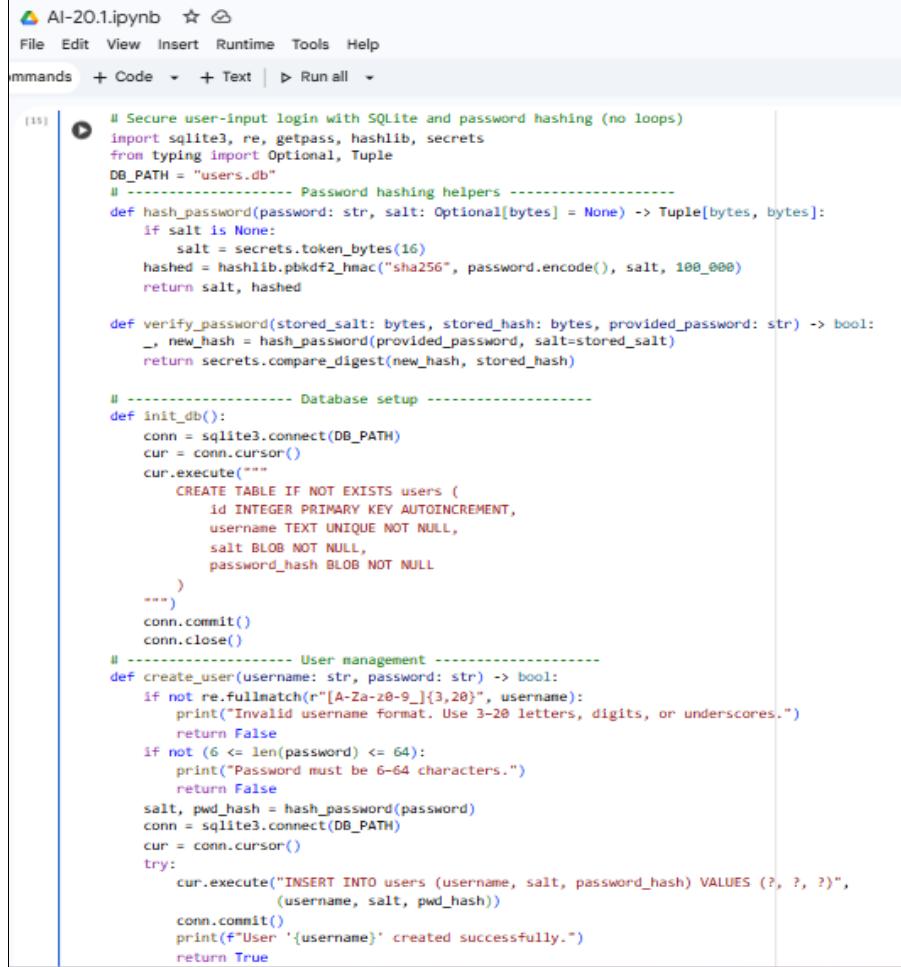
- Perform a security audit to detect possible vulnerabilities.
- Prompt AI to suggest **secure coding practices** to fix issues.
- Compare insecure vs secure versions side by side.

Expected Output:

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

Prompt: Generate a short AI-based project snippet (login form, API call). Check it for common security vulnerabilities, including input validation issues, insecure API handling.

Code:



```
# Secure user-input login with SQLite and password hashing (no loops)
import sqlite3, re, getpass, hashlib, secrets
from typing import Optional, Tuple
DB_PATH = "users.db"
# ----- Password hashing helpers -----
def hash_password(password: str, salt: Optional[bytes] = None) -> Tuple[bytes, bytes]:
    if salt is None:
        salt = secrets.token_bytes(16)
    hashed = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, 100_000)
    return salt, hashed

def verify_password(stored_salt: bytes, stored_hash: bytes, provided_password: str) -> bool:
    _, new_hash = hash_password(provided_password, salt=stored_salt)
    return secrets.compare_digest(new_hash, stored_hash)

# ----- Database setup -----
def init_db():
    conn = sqlite3.connect(DB_PATH)
    cur = conn.cursor()
    cur.execute("""
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT UNIQUE NOT NULL,
            salt BLOB NOT NULL,
            password_hash BLOB NOT NULL
        )
    """)
    conn.commit()
    conn.close()

# ----- User management -----
def create_user(username: str, password: str) -> bool:
    if not re.fullmatch(r"[A-Za-z0-9]{3,20}", username):
        print("Invalid username format. Use 3-20 letters, digits, or underscores.")
        return False
    if not (6 <= len(password) <= 64):
        print("Password must be 6-64 characters.")
        return False
    salt, pwd_hash = hash_password(password)
    conn = sqlite3.connect(DB_PATH)
    cur = conn.cursor()
    try:
        cur.execute("INSERT INTO users (username, salt, password_hash) VALUES (?, ?, ?)",
                   (username, salt, pwd_hash))
        conn.commit()
        print(f"User '{username}' created successfully.")
        return True
    except sqlite3.Error as e:
        print(f"Error creating user: {e}")
        return False
```

```

    except sqlite3.IntegrityError:
        print("Username already exists.")
        return False
    finally:
        conn.close()
def get_user_record(username: str):
    conn = sqlite3.connect(DB_PATH)
    cur = conn.cursor()
    cur.execute("SELECT id, username, salt, password_hash FROM users WHERE username = ?", (username,))
    row = cur.fetchone()
    conn.close()
    return row
# ----- Login flow -----
def login_prompt():
    username = input("Enter username: ").strip()
    if not re.fullmatch(r"[A-Za-z0-9_]{3,20}", username):
        print("Invalid username format.")
        return
    password = getpass.getpass("Enter password: ")
    record = get_user_record(username)
    if not record:
        print("No such user or wrong credentials.")
        return
    _, salt, pwd_hash = record
    if verify_password(salt, pwd_hash, password):
        print("Login successful!")
    else:
        print("Invalid username or password.")
if __name__ == "__main__":
    init_db()
    # Create a single user
    print("\n--- Create a New User ---")
    new_user = input("New username: ").strip()
    new_pass = getpass.getpass("New password: ")
    create_user(new_user, new_pass)
    # Attempt login once
    print("\n--- Login ---")
    login_prompt()

"""
--- Create a New User ---
New username: srikan
New password: .....
Username already exists.

--- Login ---
Enter username: srikan
Enter password: .....
Login successful!

```

Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.