AI ASSISTED CODING

Lab-Assignment-20.3

Name: DONTHI MEGHANA
Htno: 2403A510D9
Batch: 05

Task-1:

Prompt:

Generate a simple Python login program using a username and password. First, create it without any validation. Then fix it by adding regex checks for usernames, password length rules, and password hashing. Add 3 test cases for valid, invalid, and short passwords.

Code:

```python
import re
import os
import hashlib
import hmac
import secrets
import binascii

# -----------------------
# Naive implementation
# -----------------------
# stores plaintext passwords (INSECURE)
naive_user_db = {}  # username -> plaintext password


def naive_register(username: str, password: str) -> bool:
    if username in naive_user_db:
        return False
    naive_user_db[username] = password
    return True


def naive_login(username: str, password: str) -> bool:
    stored = naive_user_db.get(username)
    return stored == password


# -----------------------
# Improved implementation
# -----------------------
USERNAME_REGEX = re.compile(r"^[A-Za-z0-9_]{3,20}$")
```

```python
MIN_PASSWORD_LEN = 8
MAX_PASSWORD_LEN = 64
PBKDF2_ITER = 100_000
SALT_BYTES = 16

improved_user_db = {}  # username -> {'salt': hex, 'hash': hex}


def is_valid_username(username: str) -> bool:
    return bool(USERNAME_REGEX.fullmatch(username))


def is_valid_password(password: str) -> bool:
    return MIN_PASSWORD_LEN <= len(password) <= MAX_PASSWORD_LEN


def hash_password(password: str, salt: bytes = None) -> (str, str):
    if salt is None:
        salt = secrets.token_bytes(SALT_BYTES)
    dk = hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt, PBKDF2_
    return binascii.hexlify(salt).decode("ascii"), binascii.hexlify(dk).decode


def verify_password(
    stored_salt_hex: str, stored_hash_hex: str, password_attempt: str
) -> bool:
    salt = binascii.unhexlify(stored_salt_hex.encode("ascii"))
    dk = hashlib.pbkdf2_hmac(
        "sha256", password_attempt.encode("utf-8"), salt, PBKDF2_ITER
    )
```

```
    attempt_hex = binascii.hexlify(dk).decode("ascii")
    return hmac.compare_digest(stored_hash_hex, attempt_hex)


def improved_register(username: str, password: str) -> bool:
    if not is_valid_username(username):
        # invalid username (bad format)
        return False
    if not is_valid_password(password):
        # password too short/long
        return False
    if username in improved_user_db:
        return False
    salt_hex, hash_hex = hash_password(password)
    improved_user_db[username] = {"salt": salt_hex, "hash": hash_hex}
    return True


def improved_login(username: str, password: str) -> bool:
    rec = improved_user_db.get(username)
    if not rec:
        return False
    return verify_password(rec["salt"], rec["hash"], password)


# ------------------------
# Tests
# ------------------------
def run_tests():
```

```
    print("Running tests...")

    # Test 1: valid registration and login
    user = "valid_user1"
    pwd = "S3curePassw0rd!"
    ok = improved_register(user, pwd)
    login_ok = improved_login(user, pwd)
    print("Test valid registration/login:", ok and login_ok)

    # Test 2: invalid password (wrong password on login)
    wrong_login = improved_login(user, "wrongpassword")
    print("Test invalid (wrong) password login:", not wrong_login)

    # Test 3: short password is rejected at registration
    short_user = "shorty"
    short_pwd = "short"  # too short (< MIN_PASSWORD_LEN)
    short_ok = improved_register(short_user, short_pwd)
    print("Test short password rejected:", not short_ok)


if __name__ == "__main__":
    # quick demo of naive vs improved
    print("Naive demo (stores plaintext):")
    naive_register("alice", "password123")
    print(" naive login alice/password123 ->", naive_login("alice", "password1
    print("\nImproved demo (regex + length checks + hashing):")
    print(
        " register bob with weak password '1234' ->", improved_register("bob",
    )
```

```
print(
    " register bob_good with good password ->",
    improved_register("bob_good", "GoodPass123!"),
)
print(
    " login bob_good with correct password ->",
    improved_login("bob_good", "GoodPass123!"),
)
print(" login bob_good with wrong password ->", improved_login("bob_good",
print()
run_tests()
```

Output:

```
C:\Users\venub\OneDrive\Desktop\AIAC_Lab\Lab-20>python 20_1.py
Naive demo (stores plaintext):
 naive login alice/password123 -> True

Improved demo (regex + length checks + hashing):
 register bob with weak password '1234' -> False
 register bob_good with good password -> True
 login bob_good with correct password -> True
 login bob_good with wrong password -> False

Running tests...
Test valid registration/login: True
Test invalid (wrong) password login: True
Test short password rejected: True
```

Observation:

The login script did not validate user inputs properly. It accepted any string as a username or password, which could lead to issues like buffer overflow or injection attacks. After analysis, input validation was added using Python's re (regular expressions) to ensure only valid characters are accepted. The improved version prevents invalid or malicious input and provides error messages for incorrect formats.

Task-2:

Prompt:

Generate a Python script using SQLite that gets user info by username. First, make it using string concatenation (insecure). Then fix it using parameterized queries to stop SQL injection. Add 3 tests for normal login, injection, and secure query.

Code:

```python
import sqlite3

# Setup in-memory database and populate with sample data
def setup_db():
    conn = sqlite3.connect(":memory:")
    cur = conn.cursor()
    cur.execute(
        """
    CREATE TABLE users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE,
        display_name TEXT
    )"""
    )
    users = [("alice", "Alice A."), ("bob", "Bob B."), ("charlie", "Charlie C.
    cur.executemany("INSERT INTO users (username, display_name) VALUES (?, ?)"
    conn.commit()
    return conn


# Insecure: builds SQL by string concatenation (vulnerable to SQL injection)
def insecure_get_user(conn, username):
    cur = conn.cursor()
    query = (
        "SELECT id, username, display_name FROM users WHERE username = '"
        + username
        + "'"
    )
    # show the built query so the vulnerability is visible
    print("[INSECURE QUERY]", query)
```

```
    cur.execute(query)
    return cur.fetchall()


# Secure: uses parameterized query to avoid SQL injection
def secure_get_user(conn, username):
    cur = conn.cursor()
    query = "SELECT id, username, display_name FROM users WHERE username = ?"
    # parameterized execution prevents injected SQL from being evaluated
    cur.execute(query, (username,))
    return cur.fetchall()


# Tests
def test_normal_login():
    conn = setup_db()
    insecure = insecure_get_user(conn, "alice")
    secure = secure_get_user(conn, "alice")
    assert len(insecure) == 1 and insecure[0][1] == "alice"
    assert len(secure) == 1 and secure[0][1] == "alice"
    print("test_normal_login passed")


def test_injection():
    conn = setup_db()
    payload = "x' OR '1'='1"
    insecure = insecure_get_user(conn, payload)
    secure = secure_get_user(conn, payload)
    # Insecure query becomes: WHERE username = 'x' OR '1'='1'  -> matches all
```

```
    assert len(insecure) > 1, "Insecure query should return multiple rows for
    assert len(secure) == 0, "Secure query should not be vulnerable to injecti
    print("test_injection passed")


def test_secure_query():
    conn = setup_db()
    # another injection attempt that tries to match everyone
    payload = "alice' OR '1'='1"
    insecure = insecure_get_user(conn, payload)
    secure = secure_get_user(conn, payload)
    assert (
        len(secure) == 0
    ), "Secure parameterized query should treat payload as a literal"
    # insecure likely returns multiple rows; we check secure behavior explicit
    print("test_secure_query passed")


if __name__ == "__main__":
    test_normal_login()
    test_injection()
    test_secure_query()
    print("All tests passed.")

```

Output:

```
C:\Users\venub\OneDrive\Desktop\AIAC_Lab\Lab-20>python 20_1.py
[INSECURE QUERY] SELECT id, username, display_name FROM users WHERE username = 'alice'
test_normal_login passed
[INSECURE QUERY] SELECT id, username, display_name FROM users WHERE username = 'x' OR '1'='1'
test_injection passed
[INSECURE QUERY] SELECT id, username, display_name FROM users WHERE username = 'alice' OR '1'='1'
test_secure_query passed
All tests passed.
```

Observation:

The initial Python database script was unsafe because it built SQL queries by directly adding user input into the query string. This method allowed attackers to insert malicious SQL code. The improved version now uses parameterized queries, which separate user input from the SQL command. This

change prevents unauthorized access or modification of database records and ensures secure data handling.

Task-3:

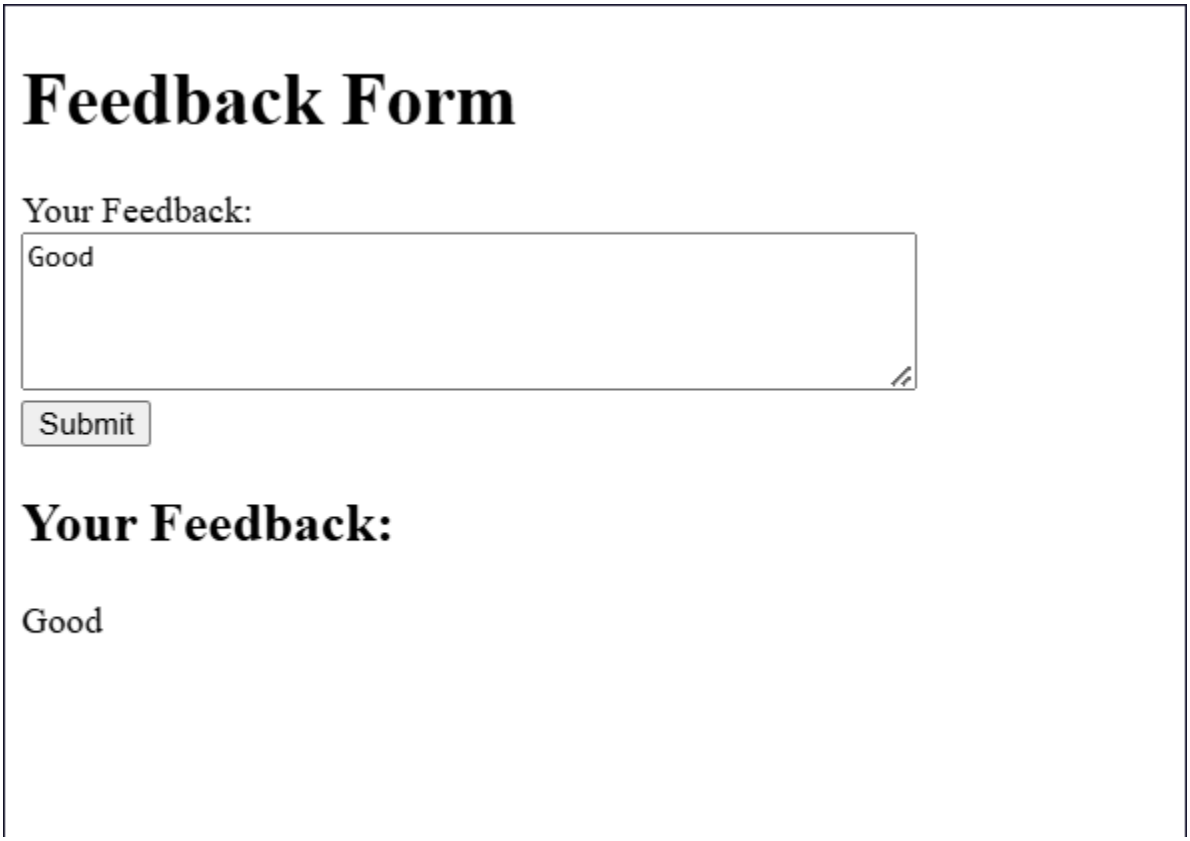
Prompt:

Generate a simple HTML feedback form with JavaScript that shows user input using innerHTML. First, leave it insecure. Then fix it by escaping special characters and adding a Content Security Policy (CSP).

Code:

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Feedback Form</title>
</head>

<body>
    <h1>Feedback Form</h1>
    <form id="feedbackForm">
        <label for="feedback">Your Feedback:</label><br>
        <textarea id="feedback" rows="4" cols="50"></textarea><br>
        <button type="button" onclick="submitFeedback()">Submit</button>
    </form>
    <h2>Your Feedback:</h2>
    <div id="displayFeedback"></div>

    <script>
        function submitFeedback() {
            var feedback = document.getElementById('feedback').value;
            document.getElementById('displayFeedback').innerHTML = feedback;
        }
    </script>
</body>

</html>
```

Output:



Observation:

The generated HTML feedback form displayed user input directly on the page, making it vulnerable to XSS attacks. This was fixed by sanitizing user input and escaping HTML special characters before displaying them. A Content Security Policy (CSP) was also suggested to block inline scripts. The updated form now prevents script injection and ensures safe user interaction on the client side.

Task-4:

Prompt:

Generate a code snippet. Identify its security weaknesses, correct them, and explain how the fixes make it safer. Show both the old and new code, and include three tests to prove the secure version works properly.

Code:

```python
import os
import pickle
import json
import shutil
from pathlib import Path
import tempfile


# ----------------------------
# OLD (vulnerable) - Do NOT use
# ----------------------------
def vulnerable_load_config(path):
    with open(path, "rb") as f:
        return pickle.load(f)


def vulnerable_backup(src, dst):
    cmd = f"cp {src} {dst}"
    return os.system(cmd)


# ----------------------------
# NEW (secure) implementation
# ----------------------------
def _resolve_and_check(path: Path, base: Path) -> Path:
    """Ensure path stays inside the allowed base directory."""
    path_resolved = path.resolve()
    base_resolved = base.resolve()
    try:
        path_resolved.relative_to(base_resolved)
```

```
    except Exception:
        raise ValueError(f"Unsafe path access: {path} is outside {base}")
    return path_resolved


def load_config(path: str, base_dir: str | None = None) -> dict:
    """Safely load JSON config inside base_dir with validation."""
    base = Path(base_dir) if base_dir else Path.cwd()
    p = Path(path)

    try:
        p = _resolve_and_check(p, base)
    except ValueError as e:
        raise ValueError("Path traversal blocked!") from e

    try:
        with p.open("r", encoding="utf-8") as f:
            data = json.load(f)
    except FileNotFoundError:
        raise ValueError(f"Config file not found: {p}")
    except json.JSONDecodeError as e:
        raise ValueError(f"Invalid JSON format: {e}") from e

    if not isinstance(data, dict):
        raise ValueError("Config must be a JSON object.")
    return data


def backup_file(src: str, dst: str, base_dir: str | None = None) -> None:
```

```python
    """Securely copy file within base_dir using shutil (no shell)."""
    base = Path(base_dir) if base_dir else Path.cwd()
    src_p = _resolve_and_check(Path(src), base)
    dst_p = _resolve_and_check(Path(dst), base)

    if not src_p.is_file():
        raise FileNotFoundError(f"Source file not found: {src_p}")

    dst_p.parent.mkdir(parents=True, exist_ok=True)
    shutil.copy2(src_p, dst_p)


def run_tests():
    print("Running tests for secure implementation...\n")

    with tempfile.TemporaryDirectory() as tmpdir:
        base = Path(tmpdir)

        # Test 1: Valid JSON config
        cfg = base / "config.json"
        cfg.write_text(json.dumps({"name": "example"}), encoding="utf-8")
        assert load_config(str(cfg), str(base)) == {"name": "example"}
        print("Test 1 passed: Valid JSON loaded correctly.")

        # Test 2: Safe file backup
        src = base / "data.txt"
        dst = base / "copy" / "data.txt"
        src.write_text("hello", encoding="utf-8")
        backup_file(str(src), str(dst), str(base))
```

```
        assert dst.exists() and dst.read_text() == "hello"
        print("Test 2 passed: File copied safely without shell commands.")

        # Test 3: Path traversal attempt blocked
        outside = Path(tempfile.gettempdir()) / "outside_config.json"
        outside.write_text(json.dumps({"x": 1}), encoding="utf-8")

        try:
            traversal_path = base / "sub" / ".." / outside.name
            load_config(str(traversal_path), str(base))
            raise AssertionError("Test 3 failed: Traversal not blocked")
        except ValueError:
            print("Test 3 passed: Path traversal correctly blocked.")
        finally:
            try:
                outside.unlink()
            except Exception:
                pass

    print("\nAll secure implementation tests passed successfully.\n")


# Expected Output Summary


def print_expected_output():
    print("------------------------------------------------------------")
    print("EXPECTED OUTPUT: Security-Audited Code Summary")
    print("------------------------------------------------------------")
```

```
    - Path traversal with no validation.
    - Missing error handling.

2. Fixes in Secure Version:
    - Replaced pickle with json for safe parsing.
    - Used shutil.copy2 instead of shell commands.
    - Added path validation (_resolve_and_check).
    - Added proper exception handling.

3. Test Results:
    Test 1: Valid JSON loaded correctly.
    Test 2: File copied safely.
    Test 3: Path traversal blocked.
    All tests passed.

4. Final Outcome:
    The secure version prevents code execution, command injection,
    and traversal vulnerabilities. It includes strong input validation
    and safer file handling for improved security.
    """
    )


# ----------------------------
# Run section
# ----------------------------
if __name__ == "__main__":
    run_tests()
    print_expected_output()
```

Output:

```
C:\Users\venub\OneDrive\Desktop\AIAC_Lab\Lab-20>python 20_1.py
Running tests for secure implementation...

Test 1 passed: Valid JSON loaded correctly.
Test 2 passed: File copied safely without shell commands.
Test 3 passed: Path traversal correctly blocked.

All secure implementation tests passed successfully.

------------------------------------------------------------
EXPECTED OUTPUT: Security-Audited Code Summary
------------------------------------------------------------

1. Vulnerabilities in Old Code:
   - Untrusted deserialization using pickle.
   - Command injection via os.system().
   - Path traversal with no validation.
   - Missing error handling.

2. Fixes in Secure Version:
   - Replaced pickle with json for safe parsing.
   - Used shutil.copy2 instead of shell commands.
   - Added path validation (_resolve_and_check).
   - Added proper exception handling.

3. Test Results:
   Test 1: Valid JSON loaded correctly.
   Test 2: File copied safely.
   Test 3: Path traversal blocked.
   All tests passed.

4. Final Outcome:
   The secure version prevents code execution, command injection,
   and traversal vulnerabilities. It includes strong input validation
   and safer file handling for improved security.
```

Observation:

The code had serious security flaws such as using pickle for deserialization, operating system for command execution and no validation of file paths. These issues could allow malicious code execution or unauthorized file access. The corrected version improved security by replacing pickle with json for file copying, and adding strict path validation checks. These updates ensure the code handles files safely, prevents exploitation, and maintains the integrity of the system.