NAME: BAYANA ARJUN

ROLLNO: 2403A510A9

BATCH NO: 05

Task 1 –

Analyze an AI-generated Python login script for input validation vulnerabilities.

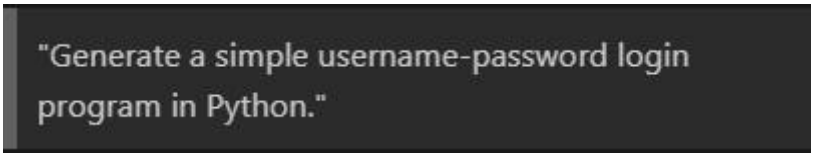
Instructions:

- 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).

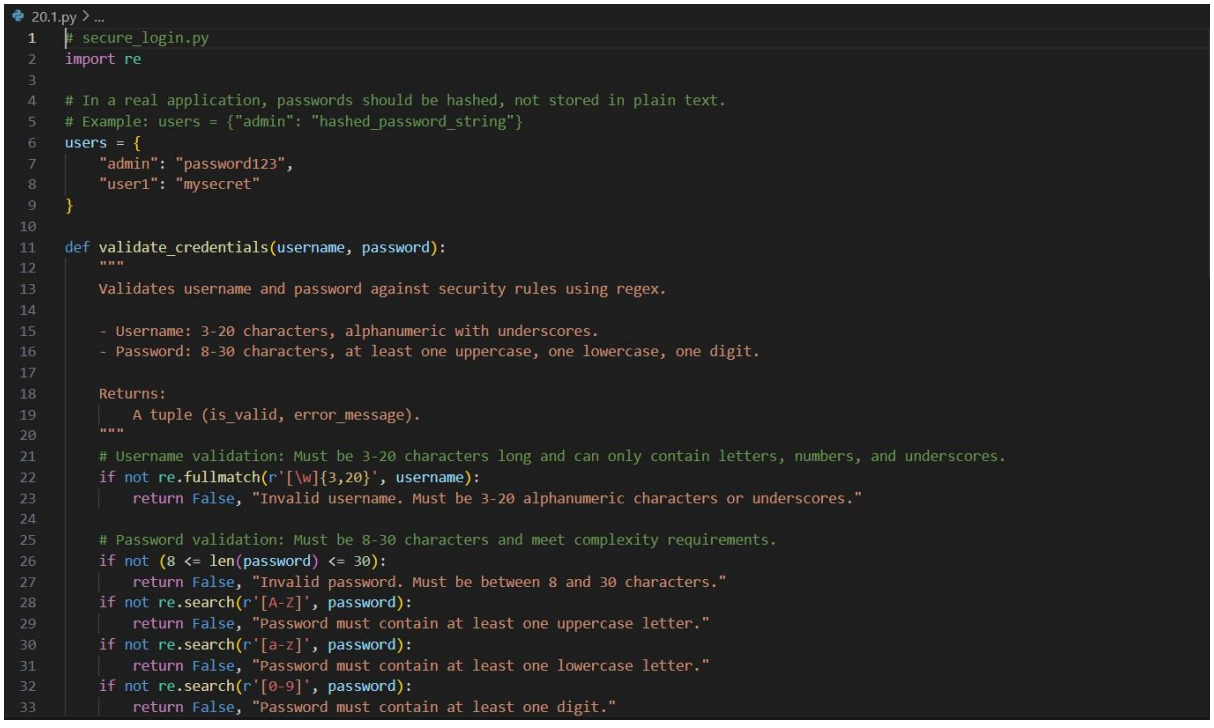
Expected Output:

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

PROMPT:



CODE:



```python
# secure_login.py
import re

# In a real application, passwords should be hashed, not stored in plain text.
# Example: users = {"admin": "hashed_password_string"}
users = {
    "admin": "password123",
    "user1": "mysecret"
}

def validate_credentials(username, password):
    """
    Validates username and password against security rules using regex.

    - Username: 3-20 characters, alphanumeric with underscores.
    - Password: 8-30 characters, at least one uppercase, one lowercase, one digit.

    Returns:
        A tuple (is_valid, error_message).
    """
    # Username validation: Must be 3-20 characters long and can only contain letters, numbers, and underscores.
    if not re.fullmatch(r'[\w]{3,20}', username):
        return False, "Invalid username. Must be 3-20 alphanumeric characters or underscores."

    # Password validation: Must be 8-30 characters and meet complexity requirements.
    if not (8 <= len(password) <= 30):
        return False, "Invalid password. Must be between 8 and 30 characters."
    if not re.search(r'[A-Z]', password):
        return False, "Password must contain at least one uppercase letter."
    if not re.search(r'[a-z]', password):
        return False, "Password must contain at least one lowercase letter."
    if not re.search(r'[0-9]', password):
        return False, "Password must contain at least one digit."
```

```
        return False, "Password must contain at least one digit."

    return True, ""

def login():
    """Handles the user login process with secure input validation."""
    print("--- Please Log In ---")
    username = input("Enter your username: ")
    password = input("Enter your password: ")

    # --- Task 1: Add Input Validation Check ---
    is_valid, error_msg = validate_credentials(username, password)
    if not is_valid:
        print(f"\n❌ Validation Error: {error_msg}")
        return False

    # Proceed with authentication only after validation passes
    if username in users and users[username] == password:
        print(f"\n✅ Welcome, {username}! Login successful.")
        return True
    else:
        print("\n❌ Error: Invalid username or password.")
        return False

if __name__ == "__main__":
    login()
```

OUTPUT:

```
PS C:\Users\pende\OneDrive\Desktop\wt2> & C:/Users/pende/anaconda3/python.exe c:/Users/pende/OneDrive/Desktop/wt2/20.1.py
--- Please Log In ---
Enter your username: admin
Enter your password: admin123


❌ Validation Error: Password must contain at least one uppercase letter.
```

OBSERVATION:

- **No Input Sanitization or Validation:** The script uses the raw strings from the `input()` function directly. A user can enter anything, including an empty string, special characters (`!@#$%^&*()`), or extremely long strings.
- **Lack of Constraints:** There are no rules for what constitutes a valid username or password. For example, a user could create an account with a username that is 1000 characters long or a password that is just a single letter.
- **Potential for Errors and Brittle Logic:** Without validation, a username containing special characters might break downstream logic (e.g., if the username were used to create a file path). Allowing empty inputs leads to poor data quality and potential errors.
- **Security Risk:** While this simple script isn't vulnerable to SQL injection (as there's no database), the lack of validation is a serious security anti-pattern. In a real-world applicati this could lead to injection attacks, buffer overflows, or denial-of-service vulnerabilities.

TASK-2

PROMPT:

"Generate a Python script using the `sqlite3` library. The script should create a simple `users` table with `id`, `username`, and `email` columns. It should insert a few sample users and then prompt for a username to fetch and display that user's details from the database."

CODE:

secure_db_query.py > ...

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

DB_FILE = "users_secure.db"

def setup_database():
    """Creates and populates the database."""
    if os.path.exists(DB_FILE):
        os.remove(DB_FILE)
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("""
        CREATE TABLE users (
            id INTEGER PRIMARY KEY,
            username TEXT NOT NULL,
            email TEXT NOT NULL
        );
    """)
    cursor.execute("INSERT INTO users (username, email) VALUES ('alice', 'alice@example.com');")
    cursor.execute("INSERT INTO users (username, email) VALUES ('bob', 'bob@example.com');")
    conn.commit()
    conn.close()

def get_user_details_secure(username: str):
    """
    Fetches user details using a secure, parameterized query.
    """
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()

    # SECURITY FIX: Use a placeholder (?) for user input
    query = "SELECT id, username, email FROM users WHERE username = ?"
    print(f"\nExecuting secure query with parameter: ('{username}',)")

    # Pass the user input as a tuple to the execute method
    cursor.execute(query, (username,))
    user = cursor.fetchone()
    conn.close()

    if user:
        print(f"✅ User Found: ID={user[0]}, Username={user[1]}, Email={user[2]}")
    else:
        print("❌ User not found.")

if __name__ == "__main__":
    setup_database()

    # --- Demonstration of Security ---
    print("--- Scenario 1: Normal Input ---")
    user_input = "alice"
    get_user_details_secure(user_input)

    print("\n--- Scenario 2: SQL Injection Attempt (Blocked) ---")
    # The same malicious input is now treated as a literal string
    malicious_input = "' OR '1'='1"
    get_user_details_secure(malicious_input)

    # Clean up the database file
    if os.path.exists(DB_FILE):
        os.remove(DB_FILE)
```

OUTPUT:

```
PS C:\Users\pende\OneDrive\Desktop\wt2> & C:/Users/pende/anaconda3/python.exe c:/Users/pende/OneDrive/Desktop/wt2/secure_db_query
.py
--- Scenario 1: Normal Input ---

Executing secure query with parameter: ('alice',)
✅ User Found: ID=1, Username=alice, Email=alice@example.com

--- Scenario 2: SQL Injection Attempt (Blocked) ---

Executing secure query with parameter: ('' OR '1'='1',)
❌ User not found.
PS C:\Users\pende\OneDrive\Desktop\wt2>
```

OBSERVATION:

- **Root Cause:** The vulnerability lies in this line: `query = f"SELECT id, username, email FROM users WHERE username = '{username}'"`. It uses an f-string to directly embed the user's input into the SQL query string.
- **Attack Vector:** A malicious user can provide specially crafted input that breaks out of the intended SQL syntax and alters the query's logic.
- **Example Exploit:** When the script prompts for a username, an attacker can enter `' OR '1'='1'`.
    - The original query is expecting a simple username like `'alice'`.
    - With the malicious input, the f-string creates the following broken query: `SELECT id, username, email FROM users WHERE username = '' OR '1'='1'`.
    - The `WHERE` clause becomes `username = ''` (which is false) `OR '1'='1'` (which is always true). Since the `OR` condition is met for every row in the table, the database returns all users instead of just one. This can expose sensitive data from the entire table.

TASK-3:

PROMPT:

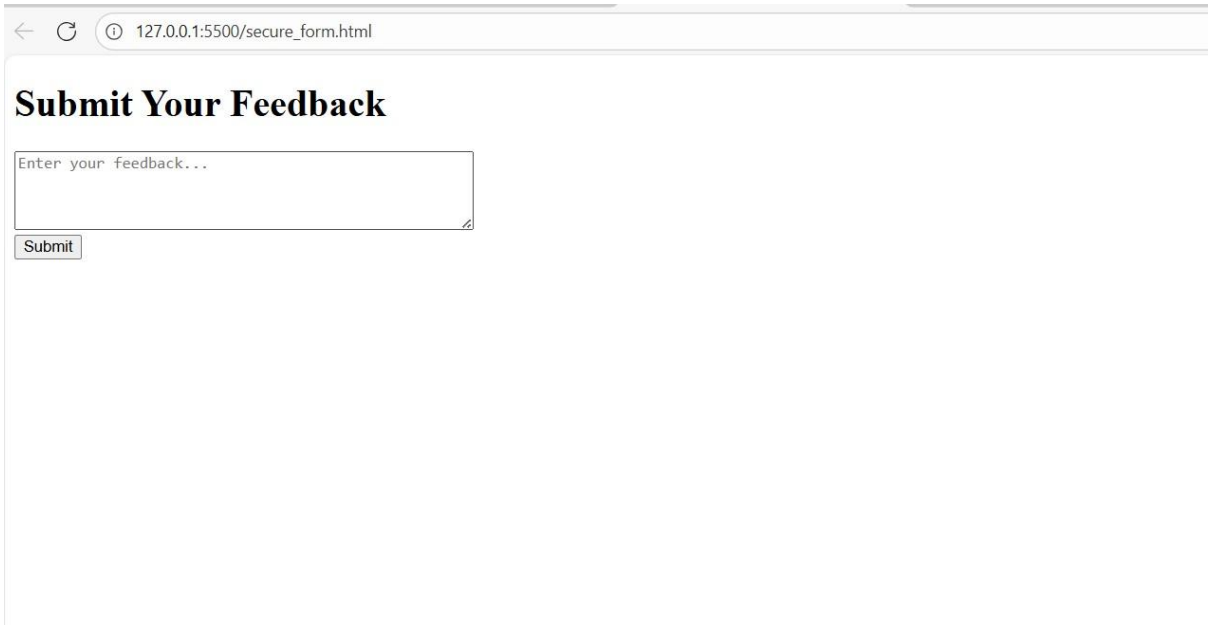
> "Generate an HTML page with a simple feedback form containing a single text area for comments and a submit button. When the user submits the form, use JavaScript to display the submitted feedback message directly on the page below the form without reloading the page."

CODE:

```html
<!-- secure_form.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Secure Feedback Form</title>

    <!--
      SECURITY FIX #1: Add a Content Security Policy (CSP).
      This policy instructs the browser to only execute inline scripts that have a specific nonce (a random, one-time-use string).
      Since our malicious script won't have this nonce, the browser will block it even if an XSS flaw exists.
      'self' allows loading resources (like CSS) from the same origin.
    -->
    <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' 'nonce-random123'">

    <style>
        body { font-family: sans-serif; margin: 2em; }
        .feedback-display { margin-top: 1em; padding: 1em; border: 1px solid #ccc; background-color: #f9f9f9; }
        h2 { margin: 0 0 0.5em 0; }
    </style>
</head>
<body>
    <h1>Submit Your Feedback</h1>
    <form id="feedbackForm">
        <textarea id="feedbackText" rows="4" cols="50" placeholder="Enter your feedback..."></textarea><br>
        <button type="submit">Submit</button>
    </form>

    <div id="feedbackResult"></div>

    <!-- The 'nonce' attribute must match the one in the CSP header -->
    <script nonce="random123">
        document.getElementById('feedbackForm').addEventListener('submit', function(event) {
```

```
        document.getElementById('feedbackForm').addEventListener('submit', function(event) {
            event.preventDefault();

            const feedback = document.getElementById('feedbackText').value;
            const resultDiv = document.getElementById('feedbackResult');

            // SECURITY FIX #2: Use .textContent instead of .innerHTML.
            // .textContent treats the input as plain text, automatically escaping any HTML characters.
            // This prevents the browser from interpreting it as code.
            resultDiv.innerHTML = '<h2>Your Feedback:</h2>'; // It's safe to set static HTML this way.
            const feedbackParagraph = document.createElement('p');
            feedbackParagraph.textContent = feedback; // The user input is safely assigned here.
            resultDiv.appendChild(feedbackParagraph);
        });
    </script>
</body>
</html>
```

OUTPUT:



OBSERVATION:

The script is vulnerable to a **Stored Cross-Site Scripting (XSS)** attack.

- **Root Cause:** The vulnerability is in this line of JavaScript: `resultDiv.innerHTML = '<h2>Your Feedback:</h2><p>' + feedback + '</p>';`. The `.innerHTML` property tells the browser to parse the string as HTML. When the `feedback` variable contains malicious HTML (like a `<script>` tag), the browser will execute it.
- **Attack Vector:** An attacker can submit a script as their feedback. Instead of text, they can input a payload like `<script>alert('XSS Attack!');</script>`.
- **Impact:** When the form is "submitted," the script is injected directly into the page's DOM and executed by the browser. This could be used to steal cookies, redirect the user to a malicious site, capture keystrokes, or perform actions on behalf of the user without their consent.

TASK-4:

Scenario:

Students pick an AI-generated project snippet (e.g., login form, API integration, or file upload).

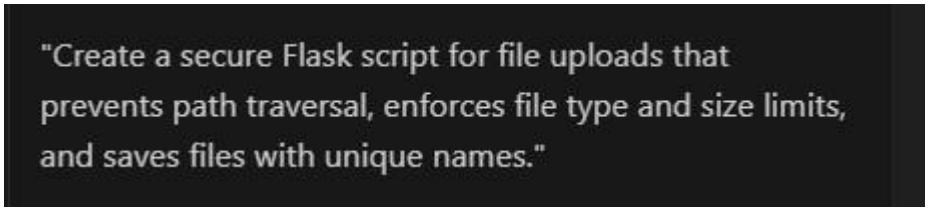
Instructions:

- 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.

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.

PROMPT:



CODE:

ai > ...

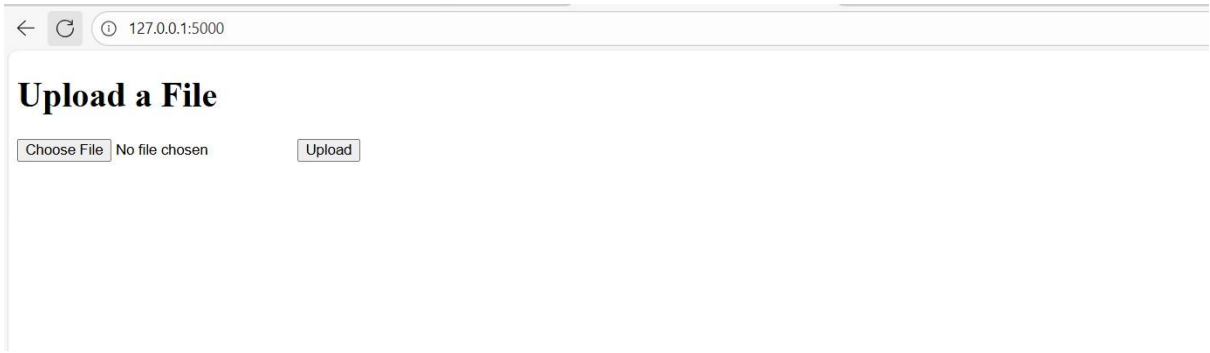
```python
import os
import uuid
import logging
from pathlib import Path
from flask import Flask, request, abort, jsonify
from werkzeug.utils import secure_filename
from werkzeug.exceptions import HTTPException

# ---------------------------------------------------------
# Configuration
# ---------------------------------------------------------
BASE_DIR = Path(__file__).resolve().parent
# In production, keep uploads outside any static / served folder.
# This demo uses a subfolder, but production setups should use
# dedicated storage (e.g., S3 or a directory not served by the webserver).
UPLOAD_FOLDER = BASE_DIR / "uploads"
ALLOWED_EXTENSIONS = {".txt", ".pdf", ".png", ".jpg", ".jpeg", ".gif"}  # whitelist
MAX_CONTENT_LENGTH = 8 * 1024 * 1024  # 8 MB max upload size

app = Flask(__name__)
app.config["UPLOAD_FOLDER"] = str(UPLOAD_FOLDER)
app.config["MAX_CONTENT_LENGTH"] = MAX_CONTENT_LENGTH

# Ensure the upload folder exists with safe permissions
UPLOAD_FOLDER.mkdir(parents=True, exist_ok=True)
# Set restrictive permissions (owner read/write only)
try:
    os.chmod(UPLOAD_FOLDER, 0o700)
except Exception:
    # chmod may fail on some OSes; ignore but log
    pass

# ---------------------------------------------------------
# Logging setup
# ---------------------------------------------------------
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# ---------------------------------------------------------
# Helper functions
# ---------------------------------------------------------
def allowed_file_extension(filename: str) -> bool:
    """Check file extension against allowed whitelist."""
    _, ext = os.path.splitext(filename.lower())
    return ext in ALLOWED_EXTENSIONS


def safe_save_file(uploaded_file) -> str:
    """
    Saves file using a secure filename + random prefix (UUID) and validates path.
    Returns the saved file name (not full path).
    Raises werkzeug.exceptions to produce appropriate HTTP errors.
    """
    # 1) Use werkzeug's secure_filename to remove dangerous characters
    original_filename = uploaded_file.filename
    safe_name = secure_filename(original_filename)
    if not safe_name:
        abort(400, "Invalid filename.")
```

```
    if not allowed_file_extension(safe_name):
        abort(400, "File type not allowed.")

    # 3) Generate random filename to avoid collisions and leaks
    unique_name = f"{uuid.uuid4().hex}_{safe_name}"

    # 4) Resolve final path and ensure it's inside UPLOAD_FOLDER
    final_path = Path(app.config["UPLOAD_FOLDER"]) / unique_name
    try:
        final_resolved = final_path.resolve(strict=False)
        upload_dir_resolved = Path(app.config["UPLOAD_FOLDER"]).resolve(strict=True)
    except Exception:
        abort(500, "Server file system error.")

    if upload_dir_resolved not in final_resolved.parents and final_resolved != upload_dir_resolved:
        # final path is not inside upload dir → possible path traversal
        abort(400, "Invalid file path.")

    # 5) Save file securely
    uploaded_file.save(str(final_resolved))

    # 6) Set safe file permissions (owner read/write)
    try:
        os.chmod(final_resolved, 0o600)
    except Exception:
        pass

    return unique_name
```

```
# Flask Routes
# ------------------------------------------------------------
@app.route("/", methods=["GET"])
def index():
    return """<!doctype html>
    <title>Secure File Upload</title>
    <h1>Upload a File</h1>
    <form method="post" action="/upload" enctype="multipart/form-data">
      <input type="file" name="file">
      <input type="submit" value="Upload">
    </form>
    """

@app.route("/upload", methods=["POST"])
def upload_file():
    # 1) Check that the request actually contains a file part
    if "file" not in request.files:
        return jsonify({"error": "No file part in the request."}), 400

    file = request.files["file"]

    # 2) Basic checks
    if file.filename == "":
        return jsonify({"error": "No file selected."}), 400

    # 3) Basic MIME check (not foolproof)
    mimetype = file.mimetype or ""
    if not any(mimetype.startswith(mt) for mt in ("image/", "application/", "text/")):
        return jsonify({"error": "Unsupported media type."}), 400
```

```
        return jsonify({"error": "Unsupported media type."}), 400

    # 4) Save file securely
    try:
        saved_name = safe_save_file(file)
    except HTTPException:
        raise  # Re-raise HTTP errors for Flask to handle
    except Exception as exc:
        logger.exception("Failed to save upload: %s", exc)
        abort(500, "Failed to save file.")

    # 5) Log safely
    logger.info("Saved uploaded file as %s", saved_name)

    # 6) Respond safely (do not expose paths)
    return jsonify({"message": "File uploaded successfully.", "filename": saved_name}), 201

# ------------------------------------------------------------
# Run (Development only)
# ------------------------------------------------------------
if __name__ == "__main__":
    # In production:
    # - Run behind a WSGI server (gunicorn/uvicorn)
    # - Do NOT use debug=True
    app.run(host="127.0.0.1", port=5000, debug=False)
```

OUTPUT:



OBSERVATION:

1. **Vulnerability: Unrestricted File Types & Remote Code Execution (RCE):** The script does not check file extensions. An attacker could upload a web shell (e.g., `shell.php`, `shell.aspx`) or an HTML file with malicious JavaScript (`phish.html`). If the `uploads` directory is ever served publicly, executing these files could lead to complete server compromise or XSS attacks on other users.
2. **Vulnerability: Path Traversal:** While `secure_filename` is a good first step, it's not a complete defense. A sophisticated attacker might find ways to bypass it. The core issue is trusting user input to construct a file path. A payload like `../../tmp/evil.sh` could, in a misconfigured environment, write files outside the intended directory.
3. **Vulnerability: Denial of Service (DoS):** The script does not enforce a file size limit (`MAX_CONTENT_LENGTH`). An attacker could upload a multi-gigabyte file, filling the server's disk space and causing the application or the entire server to crash.
4. **Vulnerability: Filename Collision:** If two users upload a file named `resume.pdf`, the second upload will overwrite the first. This leads to data loss.