Vulnerability Report: `/client_registeration` & `/client_login` Endpoints

Steps to Generate a Successful Registration Response (HTTP 200 OK):

1. **Construct the HTTP POST Request:**
    - Prepare an HTTP POST request targeted at the `/client_registeration` endpoint of the Flask application.
    - Set the request's `Content-Type` header to `application/x-www-form-urlencoded`.
2. **Populate the Request Body (Form Data):**
    - Include the following key-value pairs in the request's body:
        - `fullName`: "ABC"
        - `userName`: "testuser"
        - `email`: "ABC@XYZ.com"
        - `password`: "Test@123"
        - `phone`: "2222"
    - Ensure that the data is encoded correctly according to the `application/x-www-form-urlencoded` format.
3. **Send the Request:**
    - Transmit the constructed POST request to the Flask application's server.
4. **Verify the Response:**
    - Check the HTTP response status code.
    - A successful registration should result in an HTTP 200 OK status code.
    - Verify the response body.
    - The response body should contain the JSON data `{"msg": "User Registered"}`.

Expected Outcome:

- If the registration is successful, the Flask application will:
    - Insert the provided user data into the `users` table within the `database.db` SQLite database.
    - Return an HTTP 200 OK response.
    - Return a json body of `{"msg": "User Registered"}`.
- If the email already exists, the server will return `{"msg": "Email already Exist"}`.

- If any of the required fields are missing, the server will return `{"msg": "All fields are required"}`

1. SQL Injection (Moderate Severity)

- **Affected Endpoints:** `/client_registeration`, `/client_login`
- **Description:**
  - The application constructs SQL queries through string concatenation of user-supplied data (input - email, username). This pattern introduces a direct SQL injection vulnerability.
  - Specifically, the queries used to check for existing email addresses during registration and to authenticate users during login are susceptible.
- **Impact:**
  - Potential for unauthorized data retrieval, modification, or deletion.
  - Compromised database integrity.
- **Recommendation:**
  - Implement parameterized queries or prepared statements to ensure proper separation of SQL code and user-provided data.

2. Insufficient Password Hashing (Critical Severity)

- **Affected Endpoints:** `/client_registeration`, `/client_login`
- **Description:**
  - Passwords are stored in plaintext within the database.
  - Login authentication performs direct plaintext comparison.
- **Impact:**
  - Complete exposure of user credentials in the event of database compromise.
  - Significant data breach risk.
- **Recommendation:**
  - Employ a robust, one-way adaptive hashing/encrypt algorithm for password storage.
  - Hash input passwords during authentication for secure comparison.

3. Insecure JWT Implementation (Critical Severity)

- **Affected Endpoints:** `/client_login`, `/update_info`
- **Description:**
  - Hardcoded, predictable JWT secret key ("123456").
  - The decodeNoneJwt function disables signature verification.
- **Impact:**
  - Token forgery, enabling unauthorized access to protected resources.
  - Complete compromise of authentication.
- **Recommendation:**

    - Generate and securely store a ***cryptographically strong***, random secret key.
    - Never disable JWT verification in production.

4. Inadequate Input Validation (Low Severity)

- **Affected Endpoint:** `/client_registeration`
- **Description:**
    - Limited input validation; primarily checks for empty fields.
    - Lacks comprehensive validation for data formats (e.g., email, phone number).
- **Impact:**
    - Potential for data corruption and application instability.
    - Allows malformed data to persist in the database.
- **Recommendation:**
    - Implement stricter input validation using regular expressions or dedicated validation libraries.

5. Detailed Error Reporting (Low Severity)

- **Affected Endpoints:** `/client_registeration`, `/client_login`
- **Description:**
    - Error messages provide specific details that could aid hackers in information gathering.
- **Impact:**
    - Increased attack surface through information leakage.
- **Recommendation:**
    - Implement generic error responses without specific details in production environments.
    - Maintain detailed error logs for debugging purposes, ***but restrict access.***