```python
@app.route("/client_registeration", methods = ['POST'])
def registerToSite():
    #Paramters -> [fullName, userName, email, password, phone]
    fullName = request.form['fullName']
    userName = request.form['userName']
    email = request.form['email']
    password = request.form['password']
    phone = request.form['phone']
    if fullName != '' and userName != '' and email != '' and password != '' and phone != '':
        # Check if email already exist
        dbConn = get_db_connection()
        dbCursor = dbConn.cursor()
        q = 'select userName from users where email = "' + email + '"'
        dbData = dbCursor.execute(q).fetchall()
        if len(dbData) > 0:
            return {'msg':'Email already Exist'}

        dbCursor.execute("INSERT INTO users (fullName, userName, email, password, phone, privillage) VALUES (?, ?, ?, ?, ?, ?)",
            (fullName, userName, email, password, phone, 2)
            )
        dbConn.commit()
        dbConn.close()
        return {'msg':'User Registered'}
    else:
        return {'msg':'Invalid Data'}
```

Security Vulnerabilities for /client_registeration

ID	Vulnerability Name (Risk Score)	Issue	Risk	Steps to Reproduce	Suggested Mitigation
V-1	**SQL Injection Vulnerability** (Risk Score: 10/10)	The query string concatenates user input directly: `q = 'select userName from users where email = "' + email + '"'`	Allows attackers to inject malicious SQL code, potentially exposing or altering the database.	1. Use a tool like Burp Suite or Postman to send a POST request with email set to `" OR 1=1--`. 2. Observe the SQL query executed and verify if it bypasses the email uniqueness check.	Use parameterized queries for all database operations.
V-2	**Cleartext Password Storage** (Risk Score: 9/10)	Passwords are stored directly in the database without encryption.	If the database is compromised, all passwords are exposed in plaintext.	1. Inspect the database after a user registration. 2. Observe that passwords are stored in plaintext.	Use a secure hashing algorithm (e.g., bcrypt) to store passwords.

V-3	**Lack of HTTPS Enforcement** (Risk Score: 9/10)	No enforcement for HTTPS usage.	User data, including passwords, could be intercepted in transit.	1. Run the app without HTTPS (default Flask development mode). 2. Capture the network request with tools like Wireshark. 3. Observe sensitive data in plaintext.	Use a library like Flask-Talisman to enforce HTTPS and secure headers.
V-4	**No Rate Limiting** (Risk Score: 8/10)	No mechanism to limit the number of requests from a client.	Vulnerable to brute force and denial-of-service attacks.	1. Use a tool like Postman or JMeter to send rapid repeated requests to /client_registeration. 2. Observe that there is no throttling or blocking of repeated requests.	Use rate-limiting middleware like Flask-Limiter to restrict request rates.
V-5	**Error Disclosure** (Risk Score: 7/10)	The error message reveals whether an email exists in the system: `{'msg':'Email already Exist'}`	Exposes valid emails to attackers during enumeration attempts.	1. Send a POST request with an existing email. 2. Observe the specific error message disclosing that the email already exists.	Return generic error messages to prevent information disclosure.
V-6	**Lack of Input Validation** (Risk Score: 6/10)	No validation of input fields like email, password strength, or phone format.	Increases the risk of invalid or malicious data being stored in the database.	1. Send a POST request with invalid input, such as an incorrect email format or weak password. 2. Observe that the data is accepted and stored without validation.	Use libraries like Cerberus or Marshmallow for input validation. Ensure passwords meet complexity requirements and validate formats for email and phone.
V-7	**Weak Password Enforcement** (Risk Score: 6/10)	No mechanism enforces strong password policies.	Weak passwords increase the risk of account compromise.	1. Send a registration request with a weak password like "12345". 2. Observe that the system accepts the weak password.	Enforce a strong password policy requiring minimum length, uppercase, lowercase, numbers, and special characters.
V-8	**Lack of Role-Based Access Control (RBAC)** (Risk Score: 6/10)	Hardcoded privilege value does not align with a robust RBAC mechanism.	No granular control over what users can do, increasing the risk of unauthorized access or privilege escalation.	1. Review the privilege column values in the database. 2. Observe the lack of flexibility or proper validation for assigning roles.	Implement a proper RBAC system and ensure roles and privileges are dynamically validated during registration.
V-9	**Missing Logging and Monitoring** (Risk Score: 5/10)	No evidence of logging or monitoring for suspicious activity or registration failures.	Makes it harder to detect and respond to attacks or anomalies in real time.	1. Review the codebase and logs for any record of failed registration attempts or suspicious activity. 2. Observe the lack of logging and monitoring.	Implement logging and monitoring mechanisms to track registration activity and flag anomalies (e.g., rate of failed attempts).

Logical Vulnerabilities for /client_registeration

ID	Vulnerability Name (Risk Score)	Issue	Risk	Steps to Reproduce	Suggested Mitigation
V-10	**Unrestricted Registration** (Risk score: 7/10)	No mechanism to restrict user registration (e.g., CAPTCHA, email verification).	Allows bots or malicious users to create fake accounts, leading to potential spam or abuse.	1. Write a simple script to automate registration requests. 2. Observe that there is no verification mechanism like CAPTCHA or email validation to block automated attempts.	Implement CAPTCHA and email verification during registration to prevent abuse.
V-11	**Hardcoded Privilege Value** (Risk score: 6/10)	Privilege is hardcoded as 2 without validation or flexibility.	May allow unintended privilege escalation or misuse if the privilege system changes.	1. Inspect the privilege value stored in the database for all new users. 2. Observe that it is always hardcoded to 2.	Dynamically determine or validate the privilege level rather than hardcoding it.
V-12	**Email Uniqueness Check Logic** (Risk score: 5/10)	Email uniqueness is verified using a raw query and fetching all rows.	Inefficient for large datasets and prone to performance issues.	1. Add a large number of records to the database. 2. Observe performance degradation when checking email uniqueness.	Add a UNIQUE constraint to the email column in the database for efficiency and integrity.
V-13	**Duplicate User Registration** (Risk score: 5/10)	No mechanism prevents users with the same userName from registering, potentially causing conflicts.	Can lead to user confusion and ambiguous identification within the system.	1. Send two registration requests with the same userName but different emails. 2. Observe that the system allows both registrations, leading to ambiguity.	Add a UNIQUE constraint to the userName column in the database to prevent duplicates.
V-14	**Missing DB Error Handling** (Risk score: 4/10)	No handling for potential database failures, such as connection issues or query errors.	If a database error occurs, it could crash the server or return unhandled exceptions to the client.	1. Intentionally disrupt the database connection (e.g., by stopping the DB service). 2. Send a POST request and observe that the app crashes or returns unhandled exceptions.	Use try-except blocks around database operations to handle exceptions gracefully and provide meaningful error messages.
V-15	**Unvalidated Phone Number Format** (Risk score: 4/10)	Phone numbers are not validated, allowing invalid or improperly formatted data to be entered.	Reduces the quality of data stored and increases the risk of invalid contact information.	1. Send a registration request with an invalid phone number format. 2. Observe that the system accepts invalid phone numbers without validation.	Use a library like phonenumbers to validate phone number formats during registration.