Logical and Security testing

Endpoint: /client_registeration

▶ #1 SQL Injection Vulnerability

Issue: The query string in the following line directly concatenates user input without sanitization:

```
q = 'select userName from users where email = "' + email + '"'
```

Risk: Allows attackers to inject malicious SQL code, potentially exposing or altering the database.
Risk Score: 10/10 (Critical)
Suggested Mitigation: Use parameterised queries for all database operations

▶ #2 Cleartext Password Storage

Issue: The password is stored directly in the database without encryption

```
dbCursor.execute("INSERT INTO users (fullName, userName,
    email, password, phone, privillage) VALUES
    (?, ?, ?, ?, ?, ?)",…)
```

Risk: If the database is compromised, all passwords are exposed in plaintext
Risk Score: 9/10 (Critical)
Suggested Mitigation: Use secure hashing algorithm to store passwords

▶ #3 Lack of HTTPS Enforcement

Issue: The code does not check if the request is made over HTTPS
Risk: User data, including passwords, could be intercepted in transit
Risk Score: 9/10 (Critical)
Suggested Mitigation: Force HTTS (eg: use library like Flask-Talisman)

▶ #4 No Rate Limiting

Issue: The endpoint has no mechanism to limit the number of requests from a client
Risk: Makes the application vulnerable to brute force and denial-of-service attacks
Risk Score: 8/10 (High)
Suggested Mitigation: Use rate-limiting middleware (eg: use library like flask-limiter)

▶ #5 Error Disclosure

Issue: The message returned for duplicate email is predictable:

```
return {'msg':'Email already Exist'}
```

Risk: Exposes sensitive information (e.g., valid emails in the system) to attackers during enumeration attempts
Risk Score: 7/10 (High)
Suggested Mitigation: Return generic error messages to prevent information disclosure

▶ #6 Hardcoded Privilege Value

Issue: Privilege is hardcoded as `2` without validation or flexibility

```
dbCursor.execute("...", (fullName, userName, email, password, phone, 2))
```

Risk: May allow unintended privilege escalation if the default privilege level changes in the system and may give sensitive operations access to unauthorised users
Risk Score: 6/10 (Moderate)
Suggested Mitigation: Dynamically determine or validate the privilege level instead of hardcoding it

▶ #7 Lack of Input Validation

Issue: No validation of the input fields beyond checking if they are empty.
Examples:

- `email` format is not validated
- `password` strength (length, special characters, etc.) is not checked.
- `phone` format is not validated.

Risk: Increases the likelihood of invalid or malicious data being stored in the database.
Risk Score: 6/10 (Moderate)
Suggested Mitigation: Use a library like `cerberus` or `marshmallow` for input validation
Examples:

- Validate email with a regex or dedicated library
- Ensure passwords meet strength requirements
- Validate phone numbers with a format check

▶ #8 Email Uniqueness Check Logic

Issue: The check for email uniqueness uses a raw query and fetches all rows:

```
dbData = dbCursor.execute(q).fetchall()
if len(dbData) > 0:
```

Risk: Inefficient for large datasets and prone to performance issues.
Risk Score: 5/10 (Medium)
Suggested Mitigation: Add a `UNIQUE` constraint to the `email` column in the database for efficiency and integrity instead of querying all

▶ #9 Missing DB Error handling

Issue: The function does not handle potential failures/exceptions that may arise during database operations (eg: connections error, execution errors)
Risk: If db error occurs, it may cause server crash or return unhandled exception which can lead to poor user experience
Risk Score: 4/10 (Low)
Suggested Mitigation:Implement **try-except** blocks around database operations to handle exceptions gracefully and provide meaningful error messages to the client