```python
@app.route("/client_login", methods = ['Post'])
def loginToSite():
    userName = request.form['userName']
    email = request.form['email']
    password = request.form['password']
    qMail = 'select privillage from users where email = "' + email +'" and password = "' + password + '"'
    qUser = 'select privillage from users where userName = "' + userName +'"'
    dbConn = get_db_connection()
    dbCursor = dbConn.cursor()
    if email != '':
        dbData = dbCursor.execute(qMail).fetchall()
        if len(dbData) > 0:
            role = dbData[0][0]
            payload = {'userName':userName,'email':email,'role':role}
            token = generateJWT(payload)
            return {'token':token}
        else:
            return {'msg':'In correct email or password'}
    elif userName != '':
        dbData = dbCursor.execute(qUser).fetchall()
        if len(dbData) > 0:
            role = dbData[0][0]
            payload = {'userName':userName,'email':email,'role':role}
            token = generateJWT(payload)
            return {'token':token}
        else:
            return {'msg':'In correct username or password'}
    else:
        return {'msg':'Failed'}
```

Security Vulnerabilities for /client_login

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: `qMail = 'select privillage from users where email = "' + email +'" and password = "' + password + '"'`	Allows attackers to inject malicious SQL code, potentially exposing or altering the database.	1. Send a POST request with email as `test@example.com' OR 1=1--`. 2. Observe that the SQL query is vulnerable to injection and bypasses login checks.	Use parameterized queries for all database operations.

V-2	**Lack of HTTPS Enforcement** (Risk Score: 9/10)	No enforcement for HTTPS usage.	Sensitive user credentials such as passwords can be intercepted if transmitted over HTTP.	1. Run the app in development mode with HTTP. 2. Use a packet sniffer to capture the credentials in transit.	Enforce HTTPS for all endpoints using libraries like Flask-Talisman or reverse proxy configuration (e.g., Nginx).
V-3	**Sensitive Data in Token** (Risk Score: 7/10)	JWT token contains sensitive information like `userName`, `email`, and `role` which may be easily decoded.	If an attacker intercepts the token, they could decode it to obtain sensitive user details.	1. Send a POST request and capture the JWT token returned. 2. Decode the JWT token using tools like jwt.io.	Do not include sensitive data such as email and role in JWT payload. Only include necessary claims and use additional encryption.
V-4	**Direct Password Handling** (Risk Score: 8/10)	The password is read directly from the request and is not encrypted or protected during transit.	If the password is captured (e.g., through a man-in-the-middle attack), it will be exposed.	1. Capture the password in transit using a packet sniffer during the login request. 2. Observe that the password is exposed in plain text.	Encrypt passwords during transmission using HTTPS, and use secure password hashing mechanisms (e.g., bcrypt, PBKDF2) when storing them.
V-5	**Insecure Token Generation** (Risk Score: 6/10)	The token is generated using JWT without ensuring secure key management and algorithm choice.	If the secret key is exposed or weak, attackers could forge valid tokens.	1. Inspect the token generation code and verify the strength of the secret key. 2. Attempt to brute-force or guess the secret key.	Use a strong secret key for JWT generation and consider rotating keys periodically.
V-6	**Lack of Brute Force Protection** (Risk Score: 7/10)	No rate limiting or protection mechanism against brute force login attempts.	Attackers could attempt to guess passwords through brute force, potentially gaining unauthorized access.	1. Use a script to repeatedly send login attempts with incorrect credentials. 2. Observe that no protection against multiple failed attempts is applied.	Implement rate limiting or account lockout mechanisms to limit brute-force login attempts.
V-7	**No Session Expiry or Invalidation** (Risk Score: 6/10)	No expiry or invalidation mechanism for the JWT tokens.	If a token is compromised, it could be used indefinitely without any way to invalidate it.	1. Generate a JWT token and store it. 2. Check that the token does not expire, leading to indefinite validity.	Implement token expiry with a reasonable timeout and allow token invalidation (e.g., using refresh tokens).

Logical Vulnerabilities for /client_login

ID	Vulnerability Name (Risk Score)	Issue	Risk	Steps to Reproduce	Suggested Mitigation
V-8	**Lack of Account Lockout Mechanism** (Risk Score: 7/10)	There is no mechanism to lock an account or limit failed login attempts.	This could lead to brute force attacks, allowing attackers to guess credentials.	1. Attempt multiple failed login attempts. 2. Observe that the system allows unlimited attempts without any lockout or delay.	Implement account lockout or throttling after a certain number of failed login attempts to prevent brute force attacks.
V-9	**Unvalidated User Input** (Risk Score: 6/10)	User input (username, email, and password) is not validated or sanitized before querying the database.	If user input is not properly validated, it could result in unexpected behavior, such as SQL injection or logic errors.	1. Send a login request with special characters, such as `username' OR 1=1--`. 2. Observe if the system is vulnerable to unexpected results or SQL injection.	Validate and sanitize all user input to ensure they conform to expected formats (e.g., proper email format, password length).
V-10	**No Logging or Monitoring for Failed Login Attempts** (Risk Score: 6/10)	Failed login attempts are not logged, and there is no monitoring for suspicious login patterns.	Failure to monitor login attempts can make it difficult to detect or prevent brute force attacks.	1. Send a large number of failed login requests. 2. Observe that there is no logging or notification of such attempts.	Implement logging for failed login attempts and monitor for unusual patterns, such as multiple rapid failed attempts.
V-11	**Inconsistent Username and Email Handling** (Risk Score: 5/10)	The logic checks for email and username separately, allowing the possibility of ambiguous user identification and processing only one at a time	Users with the same username or email could be registered in multiple records, creating conflicts.	1. Send two login requests with the same userName but different emails or vice versa. 2. Observe that the system allows login for both records, which could cause confusion. 3. Also send a request with both `userName` and `email` fields filled.	Implement a check that ensures both email and userName are unique in the system or choose one as the primary key for login.
V-12	**Confusing Error Message for Incorrect Credentials** (Risk Score: 5/10)	The error message `return {'msg':'In correct email or password'}` is misleading and could confuse the user.	The error message doesn't specify whether details are correct or incorrect, reducing usability and user guidance.	1. Attempt to login with incorrect credentials. 2. Observe that the ambiguous error message is shown to the user.	Provide more descriptive error messages like `Invalid username or password` without revealing specific details.
V-13	**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.