## **API endpoints - Logical and Security vulnerabilities**

### **Endpoint: /client\_registeration**

**Bug 1: SQL Injection**

**Description:**  
The email parameter is directly concatenated into the SQL query, making it vulnerable to SQL Injection.

**Impact:**  
An attacker can execute arbitrary SQL commands, potentially accessing, modifying, or deleting data in the database.

**Recommendation:**  
Use parameterized queries or ORM (Object Relational Mapping) libraries to prevent SQL injection.

**Risk Score:**  
**Critical (9.8/10)** - Exploitable remotely, impacts data confidentiality, integrity, and availability.

**Bug 2: Plaintext Password Storage**

**Description:**  
The password is stored directly in the database without hashing, which is insecure.

**Impact:**  
If the database is compromised, user passwords can be easily retrieved and reused elsewhere.

**Recommendation:**  
Hash passwords using a secure algorithm (e.g., bcrypt or Argon2) before storing them in the database.

**Risk Score:**  
**Critical (9.1/10)** - Direct access to sensitive data.

**Bug 3: Inadequate Input Validation**

**Description:**  
The API only checks if fields are non-empty but does not validate formats for email, password.

**Impact:**  
Invalid or malicious input could bypass intended functionality or introduce vulnerabilities.

**Recommendation:**

* Validate email format using regex.
* Enforce strong password policies (e.g., length, special characters).

**Risk Score:**  
**High (7.4/10)** - Could be a stepping stone for other attacks.

**Bug 4: User Enumeration**

**Description:**  
The API responds with "Email already Exist" if the email is in the database, allowing attackers to enumerate registered users.

**Impact:**  
An attacker could identify valid emails for targeted phishing or brute-force attacks.

**Recommendation:**  
Always return a generic error message (e.g., "Invalid input").

**Risk Score:**  
**Medium (6.5/10)** - Facilitates reconnaissance.

### **Endpoint: /client\_login**

**Bug 1: SQL Injection**

**Description:**  
The email and password parameters are directly concatenated into SQL queries, making them vulnerable to SQL Injection.

**Impact:**  
An attacker can bypass authentication or retrieve sensitive data.

**Recommendation:**  
Use parameterized queries or an ORM to prevent SQL injection.

**Risk Score:**  
**Critical (9.8/10)** - Exploitable remotely and impacts authentication.

**Bug 2: Weak Authentication Logic**

**Description:**  
Separate checks for email and userName allow potential bypasses:

* A user with just a username and no email can log in without validating the password for the email.

**Impact:**  
Authentication logic is inconsistent and prone to exploitation.

**Recommendation:**  
Unify authentication logic to ensure both email and password are required, and check them together.

**Risk Score:**  
**High (8.2/10)** - Authentication bypass impacts account security.

**Bug 3: No Rate Limiting**

**Description:**  
The login endpoint does not implement rate limiting, making it susceptible to brute-force attacks.

**Impact:**  
An attacker can guess passwords for accounts without restriction.

**Recommendation:**  
Implement rate limiting and account lockouts after several failed login attempts.

**Risk Score:**  
**High (7.5/10)** - Could lead to account compromise.

**Bug 4: Missing Input Validation**

**Description:**  
The API does not validate email or username formats.

**Impact:**  
Malicious input can lead to injection attacks or unexpected behavior.

**Recommendation:**  
Add input validation to ensure fields follow expected formats.

**Risk Score:**  
**Medium (6.2/10)** - Amplifies other vulnerabilities.

**Bug 5: Insecure Token Generation**

**Description:**  
The generateJWT function is not defined in the provided code. If insecure practices are used (e.g., weak secret keys, no expiration), tokens may be vulnerable.

**Impact:**  
An attacker could forge tokens or use stolen tokens indefinitely.

**Recommendation:**

* Use strong secret keys.
* Set short token expiration times.
* Implement token invalidation mechanisms.

**Risk Score:**  
**High (8.0/10)** - Impacts session security.

### **Summary of Findings**

|  |  |  |  |
| --- | --- | --- | --- |
| Bug | Endpoint | Risk Score | Impact |
| SQL Injection | Both | 9.8 (Critical) | Database compromise, data theft |
| Plaintext Password | /client\_registeration | 9.1 (Critical) | User passwords are exposed |
| Weak Authentication | /client\_login | 8.2 (High) | Potential login bypass |
| No Rate Limiting | /client\_login | 7.5 (High) | Enables brute-force attacks |
| User Enumeration | /client\_registeration | 6.5 (Medium) | Facilitates reconnaissance |
| Input Validation | Both | 7.4 (High) | Increased attack surface |
| Insecure Token Handling | /client\_login | 8.0 (High) | Session hijacking or token forgery |

By addressing these findings with the recommended fixes, we can significantly improve the logical and security robustness of the API endpoints.

### **Risk Score Calculation**

The **risk scores** in the analysis were estimated using a simplified **CVSS (Common Vulnerability Scoring System)** approach. CVSS provides a structured method for calculating the severity of vulnerabilities based on a combination of exploitability and impact metrics. Here's how the scores were derived:

**Factors Considered**

1. **Exploitability:**
   * **Attack Vector (AV):** How easily the vulnerability can be exploited (e.g., remotely, locally).
   * **Attack Complexity (AC):** The difficulty level of exploiting the vulnerability.
   * **Privileges Required (PR):** Whether an attacker needs access or credentials.
   * **User Interaction (UI):** Whether the user must do something for the exploit to succeed.
2. **Impact:**
   * **Confidentiality Impact (C):** How much sensitive data can be exposed.
   * **Integrity Impact (I):** How much the vulnerability can tamper with data.
   * **Availability Impact (A):** Whether the vulnerability affects service availability.
3. **Context-Specific Considerations:**
   * Applicability to the provided APIs.
   * Potential for real-world exploitation.

**Steps in Risk Scoring**

1. **Assign Scores for Exploitability & Impact:**
   * Scores are set between **0 (None)** to **10 (Critical)**.
   * Each factor is weighted based on its criticality.
2. **Calculate the Risk:**
   * A simple weighted average approach was used for this estimate.
   * More severe vulnerabilities (e.g., SQL Injection) are rated higher because they are easier to exploit and have broader impacts.

**Example Calculation**

**1. SQL Injection**

* **Attack Vector:** Remote (**Score: 10**) – Can be exploited remotely via API calls.
* **Attack Complexity:** Low (**Score: 10**) – Simple concatenation-based injection.
* **Privileges Required:** None (**Score: 10**) – No authentication required.
* **Confidentiality Impact:** High (**Score: 10**) – Full database exposure.
* **Integrity Impact:** High (**Score: 10**) – Allows arbitrary modifications to the database.
* **Availability Impact:** High (**Score: 10**) – Can disrupt or delete the database.

**Using a basic average:**

**Risk Score** = **Sum of Scores** / **Number of Metrics** = (10 + 10 + 10 + 10 + 10 + 10) / 6 = 10

Adjusted for context (real-world exploitability, assumed database protections): **9.8/10**

**2. User Enumeration**

* **Attack Vector:** Remote (**Score: 10**) – Can be exploited remotely.
* **Attack Complexity:** Low (**Score: 10**) – Simple response differentiation.
* **Privileges Required:** None (**Score: 10**) – No authentication needed.
* **Confidentiality Impact:** Medium (**Score: 6**) – Exposes emails but no sensitive data.
* **Integrity Impact:** None (**Score: 0**) – No tampering is possible.
* **Availability Impact:** None (**Score: 0**) – No service disruption.

**Risk Score** = **Sum of Scores** / **Number of Metrics** = (10 + 10 + 10 + 6 + 0 + 0) / 6 = 6.0

Adjusted for context (less critical due to minimal impact): **6.5/10**