# Vulnerability Explanation – Vulnerable Flask App

This document explains the intentionally built vulnerabilities in the vulnerable Flask application as part of the cybersecurity assessment task.  
Each vulnerability includes its description, location in the code, the associated risk, and recommended mitigation strategies.

## 1. Broken Access Control (OWASP A01:2021)

* **Location**: /login endpoint (login() function)
* **Description**:  
  After a successful login, the application exposes the full list of users, including usernames and roles, without enforcing any access control based on user privileges.
* **Risk**:  
  Any authenticated user can access sensitive user data, leading to information leakage and potential privilege abuse.
* **Recommended Mitigation**:  
  Implement strict role-based access control (RBAC). Only authorized users (e.g., users with an "admin" role) should be allowed to access user lists or sensitive data.

## 2. Cryptographic Failures (OWASP A02:2021)

* **Location**: /login endpoint (login() function)
* **Description**:  
  Passwords are compared directly as plaintext without any cryptographic protection, and no secure password storage practices are implemented.
* **Risk**:  
  Handling plaintext passwords exposes the system to credential theft, replay attacks, and offline brute-force attacks if a breach occurs.
* **Recommended Mitigation**:  
  Implement secure password hashing using algorithms such as bcrypt, Argon2, or PBKDF2 before storing passwords or comparing them during authentication.

## 3. Injection – SQL Injection (OWASP A03:2021)

* **Location**: /query endpoint (query() function)
* **Description**:  
  User input from the 'username' query parameter is embedded directly into SQL queries without sanitization or parameterization, making it vulnerable to SQL Injection.
* **Risk**:  
  Attackers can manipulate SQL queries to access unauthorized data, retrieve the entire database, or even modify or delete records.
* **Recommended Mitigation**:  
  Always use parameterized queries (prepared statements) to safely handle user input in database operations.

## 4. Insecure Design – Command Injection (OWASP A04:2021)

* **Location**: /ping endpoint (ping() function)
* **Description**:  
  The user input 'target' is passed directly into an OS command (ping) without validation, allowing arbitrary command injection.
* **Risk**:  
  Attackers can execute unauthorized OS-level commands, leading to remote code execution and complete server compromise.
* **Recommended Mitigation**:  
  Validate and sanitize user input to allow only valid IP addresses or domain names.  
  Avoid using raw os.system calls; instead, use safer libraries like subprocess.run() with strict argument lists.

## 5. Security Misconfiguration – Cloud Storage (OWASP A05:2021)

* **Location**: /upload endpoint (upload() function)
* **Description**:  
  AWS access keys and secret keys are hardcoded directly into the application code. The simulated file upload does not enforce any access control or encryption.
* **Risk**:  
  Exposure of cloud credentials can lead to unauthorized access to storage resources, data breaches, service hijacking, and full account compromise.
* **Recommended Mitigation**:  
  Store sensitive credentials securely using environment variables or secret management services like AWS Secrets Manager.  
  Apply strict IAM policies and bucket-level access controls.

## 6. Insecure Deserialization (OWASP A08:2021)

* **Location**: /deserialize endpoint (deserialize() function)
* **Description**:  
  The application deserializes user-supplied data using pickle.loads() without any verification or validation.
* **Risk**:  
  Attackers can craft malicious serialized payloads to execute arbitrary code on the server, leading to remote code execution and system compromise.
* **Recommended Mitigation**:  
  Avoid deserializing untrusted data with unsafe libraries like pickle. Use safer alternatives such as JSON, and always validate and sanitize incoming data structures.

## Simulated Vulnerability (Real-World CVE Demonstration)

### Simulated Use-After-Free (CVE-2025-29824 Concept)

* **Location**: /allocate, /free, /use endpoints (allocate(), free(), use() functions)
* **Description**:  
  The application simulates a use-after-free vulnerability by allowing the use of deleted objects without proper re-validation or null checks.
* **Risk**:  
  In real systems (especially at the kernel level), use-after-free vulnerabilities can lead to memory corruption, privilege escalation, arbitrary code execution, or complete system crashes.
* **Recommended Mitigation**:  
  In production systems:
  + Validate the object state before using it.
  + Nullify or invalidate object references after freeing.
  + Implement strict memory management and reference tracking to avoid use-after-free conditions.