CODEX Code Review

20/10

Summary

The provided `login` function is critically vulnerable to SQL injection due to the direct concatenation of user-supplied input into the SQL query string. This allows attackers to bypass authentication or execute arbitrary database commands. Additionally, the function implies that passwords are being stored and compared in plaintext, which is a severe security risk.

Security Issues

Severity	Issue
High	
High	

Suggestions

- **Prevent SQL Injection:** Always use parameterized queries (prepared statements) provided by your database driver. This separates the SQL command from the data, preventing malicious input from being interpreted as code.
- **Secure Password Hashing:** Never store passwords in plaintext. Use strong, one-way cryptographic hash functions like bcrypt, scrypt, or Argon2 (with appropriate salts) to store password hashes. During login, hash the user-provided password and compare it with the stored hash.
- **Error Handling:** Implement robust error handling for database operations to gracefully manage connection issues or query failures.
- **Principle of Least Privilege:** Ensure the database user account used by the application has only the necessary permissions (e.g., `SELECT` for authentication, not `INSERT`, `UPDATE`, `DELETE` unless explicitly required elsewhere).

Refactored Code

```
def login(user, pwd): # IMPORTANT: Replace 'db.execute' with your actual database driver's method # for executing parameterized queries. This is a conceptual example. # Also, this refactoring assumes 'pwd' is already a hash to be compared. # For real-world, you'd fetch the stored hash for 'user' and then compare. # Example using parameterized
```

query (syntax may vary based on actual DB library): query = "SELECT * FROM users WHERE username = ? AND password = ?" # Or for named parameters: query = "SELECT * FROM users WHERE username = :username AND password = :password" # Assuming 'db.execute' can take parameters separately # If 'pwd' is meant to be a hash, you'd typically fetch the stored hash for the user # and then use a password verification function (e.g., bcrypt.checkpw) # For demonstration, assuming 'pwd' is the plaintext password to be hashed and compared # In a real scenario, you'd fetch the stored hash for the user from the DB first. # For this refactor, we'll assume `pwd` passed in is the plaintext and we're comparing against a stored hash. # A more realistic login flow would be: # 1. Fetch user by username # 2. If user exists, retrieve their stored hashed password and salt # 3. Hash the provided 'pwd' using the same salt # 4. Compare the newly generated hash with the stored hash # Simplified refactor focusing ONLY on SQL injection prevention based on original function's structure: try: # This assumes 'db.execute' accepts a query string and a tuple/list of parameters result = db.execute(query, (user, pwd)) # Using positional parameters # Or for named parameters: result = db.execute(query, {'username': user, 'password': pwd}) return True if result else False except Exception as e: # Log the error appropriately print(f"Database error during login: {e}") return False