CODEX Code Review

2/10

Summary

The provided Python code for the login function has a critical security vulnerability due to SQL injection. It directly concatenates user input into the SQL query, making it susceptible to malicious input that could bypass authentication or access/modify database content.

Security Issues

Severity	Issue	Fix
High	SQL Injection Vulnerability: The SQL query is constructed by directly concatenating user-provided `user` and `pwd` strings. An attacker can inject malicious SQL code through these parameters, leading to unauthorized access, data manipulation, or data exfiltration.	Use parameterized queries (prepared statements) to separate SQL logic from user input. Most database drivers support this by passing parameters separately to the execute method, preventing them from being interpreted as part of the SQL command.
High	Plain Text Password Comparison: The code compares the password directly from user input against a value in the database. Storing or comparing passwords in plain text is a significant security risk, as it makes them vulnerable to theft if the database is compromised.	Store only cryptographically hashed and salted passwords in the database. When a user attempts to log in, hash and salt the provided password using the same method (and salt, if applicable) and compare the resulting hash to the stored hash. Never store or transmit plain text passwords.

Suggestions

- Implement proper error handling for database operations. What happens if `db.execute` fails or raises an exception?
- Consider adding rate limiting to login attempts to mitigate brute-force attacks.
- Ensure that the `db.execute` method is part of a secure database interaction layer that handles connection pooling, transaction management, and resource cleanup.
- Provide more informative feedback to the caller than just `True`/`False` (e.g., specific error codes for invalid user, wrong password, or internal error, though be careful not to reveal too much to attackers).

Refactored Code

```
import hashlib def login(user, pwd): # This is a conceptual refactoring. 'db' and its 'execute' method # would need to support parameterized queries, e.g., using '?' or '%s'. # Also, proper password hashing and salting should be implemented. # --- Improved Password Handling (Conceptual) --- # In a real application, you'd retrieve the salt for the 'user' # from the database, then hash the provided 'pwd' with that salt. # For this example, we'll assume a 'get_stored_hash_and_salt' function. # stored_hash, stored_salt = get_stored_hash_and_salt(user) # if not stored_hash: return False # User not found # hashed_pwd = hashlib.sha256((pwd + stored_salt).encode()).hexdigest() # query = "SELECT id FROM users WHERE username = ? AND password_hash = ?" # result = db.execute(query, (user, hashed_pwd)) # --- SQL Injection Fix (Parameterized Query) --- # Assuming 'db.execute' supports parameterized queries with '?' placeholders # and takes parameters as a tuple/list. query = "SELECT id FROM users WHERE username = ? AND password = ?" try: # The actual 'db' object and its 'execute' method's signature # will depend on the database library being used (e.g., psycopg2, sqlite3, mysql-connector). # Example for sqlite3: cursor.execute("SELECT ... WHERE a=? AND b=?", (a_val, b_val)) result = db.execute(query, (user, pwd)) # Assuming result is a cursor or similar object that can be checked for rows return True if result and result.fetchone() else False except Exception as e: # Log the error for debugging, but don't expose sensitive info to the user print(f"Database error during login: {e}") return False
```