# CODEX Code Review

**1/10**

## Summary

The provided login function is critically vulnerable to SQL injection due to direct string concatenation of user inputs into the SQL query. It also appears to handle passwords insecurely by comparing them directly, implying they might be stored in plain text or weakly hashed.

## Security Issues

| Severity | Issue | Fix |
|---|---|---|
| High | SQL Injection vulnerability. The 'user' and 'pwd' parameters are directly embedded into the SQL query string without proper sanitization or parameterization. This allows an attacker to manipulate the query, potentially bypassing authentication, accessing sensitive data, or executing arbitrary database commands. | Use parameterized queries (prepared statements) provided by your database driver or ORM. This separates the SQL logic from the user-supplied data, preventing injection attacks. For example, if 'db.execute' supports it, pass parameters as a separate argument. |

| | | |
|---|---|---|
| High | Insecure Password Handling. The code compares the provided password directly with a stored value, which strongly suggests that passwords are either stored in plain text or are hashed without proper salting and a strong hashing algorithm. This makes passwords vulnerable to brute-force attacks, rainbow table attacks, and compromise if the database is breached. | Never store plain text passwords. Always hash and salt passwords using a strong, slow hashing algorithm like bcrypt, scrypt, or Argon2. When a user attempts to log in, retrieve the stored hash and salt, and then hash the provided password with the retrieved salt, comparing the resulting hash to the stored hash. |

## Suggestions

- Implement parameterized queries for all database operations involving user input.
- Adopt a robust password hashing and salting scheme (e.g., bcrypt) instead of direct password comparison.
- Add proper error handling for database operations.
- Consider using an Object-Relational Mapper (ORM) like SQLAlchemy for more secure and abstract database interactions.
- Log failed login attempts to detect potential brute-force attacks.
- Return more specific error messages (e.g., 'Invalid username' vs. 'Invalid password') only after successful authentication, to avoid aiding attackers in enumeration.

## Refactored Code

```
import bcrypt # Assuming bcrypt is installed for password hashing def login(user, pwd): # In a real application, 'db' would
be a proper database connection object # that supports parameterized queries and secure password retrieval. # 1.
Retrieve the stored hashed password and salt for the given username. # This assumes 'db.execute' can fetch a single
user's data securely. # The query should only retrieve the hash, not the plaintext password. # Example: SELECT
hashed_password, salt FROM users WHERE username = ? # Placeholder for fetching user data securely try: # Example
of a parameterized query to fetch user details # This assumes db.execute takes a query string and a tuple/list of
parameters user_data = db.execute("SELECT id, hashed_password, salt FROM users WHERE username = ?", (user,))
except Exception as e: print(f"Database error: {e}") # Log the error appropriately return False if not user_data: return False
# User not found # Assuming user_data is a dictionary-like object or a tuple stored_hashed_password =
user_data['hashed_password'] # Or user_data[1] if it's a tuple stored_salt = user_data['salt'] # Or user_data[2] if it's a tuple
# 2. Hash the provided password with the retrieved salt and compare. # bcrypt.checkpw handles hashing the provided
password with the provided salt # and comparing it to the stored hash securely. try: # Ensure pwd is bytes for bcrypt if
bcrypt.checkpw(pwd.encode('utf-8'), stored_hashed_password.encode('utf-8')): return True else: return False except
Exception as e: print(f"Password verification error: {e}") # Log the error appropriately return False # Example of how a
password might be stored initially (for context, not part of login function): # def register_user(user, pwd): # salt =
bcrypt.gensalt() # hashed_password = bcrypt.hashpw(pwd.encode('utf-8'), salt) # # Store user, hashed_password, and
```

```
salt in the database # db.execute("INSERT INTO users (username, hashed_password, salt) VALUES (?, ?, ?)", (user,
hashed_password.decode('utf-8'), salt.decode('utf-8')))
```