Flask SQLite Authentication Setup

Please explain what this code does.

```
import sqlite3, random
from flask import Flask, abort, request, jsonify
from flask_cors import CORS

app = Flask(__name__)
CORS(app)
database = './login.db'

def create_response(message):
    response = jsonify({'message': message})
    response.headers.add('Access-Control-Allow-Origin', '*')

    return response

@app.route('/setup', methods=['POST'])
def setup():
    connection = sqlite3.connect(database)
    SECRET_PASSWORD = "letMeIn!";
    THIS_IS_A_VARIABLE = "WBneKJw1fHch8Qd3XFUS";
    print("Super Secret Password SSH Server Password to 10.10.10.1:22: " + SECRET_PASSWORD)
    connection.executescript('CREATE TABLE IF NOT EXISTS login(username TEXT NOT NULL UNIQUE, password TEXT NOT NULL);')
    connection.executescript('INSERT OR IGNORE INTO login VALUES("user_1","123456");')
    return create_response('Setup done!')

@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']
    connection = sqlite3.connect(database)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM login WHERE username = "%s" AND password = "%s"' % (username, password))
    user = cursor.fetchone()
    if user:
        response = create_response('Login successful!')
        response.set_cookie('SESSIONID', str(random.randint(1,9999999999999999999999)),httponly=False,secure=False)
        response.set_cookie('TESTID1', str("TESTSTRING1"), httponly=True,secure=True)
        response.set_cookie('TESTID2', str("TESTSTRING2"))
        return response
    else:
        response = create_response('Login failed!')
        response.delete_cookie('username')
        return response, 401

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8080, debug=True)
```

The code provided outlines a basic Flask application with a backend database using SQLite, intended for simple user authentication and session handling. Here's a breakdown of its major components and functionalities:

1. **Imports and Setup**:
    - `sqlite3`: To manage a SQLite database.
    - `random`: To generate random numbers (used for session ID).
    - `Flask`, `abort`, `request`, `jsonify`: Flask components for building the web server.
    - `CORS`: Used to enable Cross-Origin Resource Sharing on the server, allowing requests from other domains.
    - The Flask application is created (`app`) and CORS is enabled for it.
2. **Database Initialization**:
    - A global variable `database` is set to `'./login.db'`, which is the SQLite database file.
3. **Helper Function (`create_response`)**:
    - A utility function that creates a JSON response with a given message and adds a header to allow cross-origin requests.
4. **Route: `/setup` (HTTP POST)**:
    - Initializes the database connection.
    - Hard-codes a secret password and another variable (which doesn't seem to be used elsewhere in the code).
    - Outputs a hard-coded secret password, which seems like a security flaw.
    - Creates a `login` table in the database if it doesn't exist.
    - Inserts a default user with a username of `"user_1"` and password `"123456"` if they don't already exist.
    - Returns a response indicating setup completion.
5. **Route: `/login` (HTTP POST)**:
    - Retrieves `username` and `password` from the request's JSON body.
    - Opens a connection to the database and searches for a user with the provided credentials.
    - If the user exists, it:
        - Returns a success message.
        - Sets a session ID cookie (`SESSIONID`) that is not HTTP-only or secure, which can be a security risk.

        - Sets two test cookies (`` `TESTID1` `` and `` `TESTID2` ``), one of which is HTTP-only and secure.
    - If the user does not exist:
        - Returns a login failed message and deletes the `` `username` `` cookie.
        - Responds with a 401 Unauthorized status code.

6. **Main Execution**:

    - Runs the Flask application on all interfaces (`` `host='0.0.0.0'` ``) at port 8080, with debug mode enabled, which is helpful for development but should be disabled in a production environment to avoid security risks.

Overall, the code sets up a simple web server with routes to handle user setup and login. However, it contains several security concerns, such as hardcoded credentials, debug information printing, insecure cookie handling, and potential SQL injection vulnerabilities due to the unsafe insertion of user input directly into SQL queries.

Is there a vulnerability in this code? Yes or No?

Yes.

What is the vulnerability in this code? Please provide the CWE number if possible.

The primary vulnerability in the provided code is **SQL Injection**. This occurs in the `` `/login` `` route where the application constructs an SQL query by directly embedding user inputs (`` `username` `` and `` `password` ``) into the query without proper sanitization or parameterization. This makes it possible for an attacker to manipulate the query to gain unauthorized access to data or perform other malicious actions.

The SQL injection is demonstrated by the line:

```python
cursor.execute('SELECT * FROM login WHERE username = "%s" AND password = "%s"' % (username, password))
```

By embedding user-supplied inputs directly into the SQL command, the code becomes vulnerable to SQL injection attacks.

The **Common Weakness Enumeration (CWE)** identifier for SQL Injection is **CWE-89**.

Please suggest a fix for this code.

To fix the SQL injection vulnerability in the provided code, you should use parameterized queries instead of string interpolation for SQL commands. This prevents users from injecting malicious SQL code into your database queries. Here’s how you can modify the `` `/login` `` route to implement this:

Replace the existing SQL execution line with a parameterized query:

python

```
cursor.execute('SELECT * FROM login WHERE username = ? AND password = ?', (username,
password))
```

This revised line uses placeholders (`` `?` ``) for parameters, and the actual values are provided as a tuple in the second argument to `` `execute` ``. This method ensures that the values are treated as data and not executable code, effectively preventing SQL injection.

Here's the revised version of the `` `/login` `` function:

python

```
@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']
    connection = sqlite3.connect(database)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM login WHERE username = ? AND password = ?', (username,
password))
    user = cursor.fetchone()
    if user:
        response = create_response('Login successful!')
        response.set_cookie('SESSIONID', str(random.randint(1,99999999999999999999999)),
httponly=False, secure=False)
        response.set_cookie('TESTID1', str("TESTSTRING1"), httponly=True, secure=True)
        response.set_cookie('TESTID2', str("TESTSTRING2"))
        return response
    else:
        response = create_response('Login failed!')
        response.delete_cookie('username')
        return response, 401
```

Additionally, consider the following security enhancements:

1. **Session Management:** Avoid using `` `random.randint` `` for session IDs as they might not be secure enough. Consider using a more secure method like `` `os.urandom` `` or libraries designed for session

management.

2. **Cookie Security:** Set cookies to be `httponly` and `secure` where appropriate. This can help mitigate the risk of client-side scripts accessing the cookies and reduce the risk of interception over non-HTTPS connections.
3. **Error Handling:** Avoid sending detailed error messages or sensitive data in responses that might help an attacker.
4. **Password Storage:** Ensure that passwords stored in your database are hashed using a strong hashing algorithm, like bcrypt, and not stored as plain text.
5. **Debug Mode:** Disable Flask's debug mode in production environments to prevent the exposure of sensitive debug information.

Are there any other vulnerabilities in this code? If yes, please explain and suggest fixes.

Yes, apart from the SQL injection vulnerability, there are several other security issues in the code that should be addressed:

1. **Hardcoded Secrets:**

    - The secret password is hardcoded within the application code (`SECRET_PASSWORD = "letMeIn!"`). Hardcoding credentials in the source code can expose sensitive information to anyone who has access to the source.

    Fix:

    - Store sensitive information like passwords in environment variables or use a secure vault solution to manage secrets.

2. **Insecure Cookie Handling:**

    - The `SESSIONID` cookie is set without the `httponly` or `secure` flags, making it susceptible to client-side script access and interception over non-secure channels (HTTP).

    Fix:

    - Set the `SESSIONID` cookie with `httponly=True` to prevent access by JavaScript.
    - Set `secure=True` to ensure the cookie is transmitted only over HTTPS.

    Revised example:

    ```python
    response.set_cookie('SESSIONID', str(random.randint(1,99999999999999999999999)),
    httponly=True, secure=True)
    ```

3. **Insufficient Logging and Monitoring:**

   - The application does not implement any logging mechanism for login attempts, which is crucial for detecting and responding to potential security incidents.

   Fix:

   - Implement logging for both successful and failed authentication attempts. Ensure that logs contain sufficient details such as timestamps, source IP addresses, and user identifiers (but not passwords).

4. **Debug Information Exposure:**

   - The application is configured to run with debug mode enabled, which can lead to the exposure of sensitive information such as stack traces and application internals in case of errors.

   Fix:

   - Ensure that the application is deployed with debug mode disabled in production environments.

5. **Potential for Security Misconfiguration:**

   - The CORS policy allows all origins (`` `Access-Control-Allow-Origin` ``, `` `*` ``). This is too permissive and can expose the application to unnecessary risks from other domains.

   Fix:

   - Define a more restrictive CORS policy to only allow specific trusted domains.

6. **Default Credentials and Setup Route Exposure:**

   - The `` `/setup` `` route creates a default user with a weak password, and the route does not require any authentication, which might be exploited if accessible on a production server.

   Fix:

   - Remove or secure the setup route, ensuring it's not accessible on live servers without proper authentication.
   - Avoid creating default users with weak passwords. If default credentials are necessary, enforce a password change on first login.

By addressing these issues, the application will significantly improve in terms of security and resilience against common web vulnerabilities.

List all the security vulnerabilities in the following code:

Please explain what this code does.

```
import sqlite3, random
from flask import Flask, abort, request, jsonify
from flask_cors import CORS

app = Flask(__name__)
CORS(app)
database = './login.db'

def create_response(message):
    response = jsonify({'message': message})
    response.headers.add('Access-Control-Allow-Origin', '*')

    return response

@app.route('/setup', methods=['POST'])
def setup():
    connection = sqlite3.connect(database)
    SECRET_PASSWORD = "letMeIn!";
    THIS_IS_A_VARIABLE = "WBneKJw1fHch8Qd3XFUS";
    print("Super Secret Password SSH Server Password to 10.10.10.1:22: " + SECRET_PASSWORD)
    connection.executescript('CREATE TABLE IF NOT EXISTS login(username TEXT NOT NULL UNIQUE, password TEXT NOT NULL);')
    connection.executescript('INSERT OR IGNORE INTO login VALUES("user_1","123456");')
    return create_response('Setup done!')

@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']
    connection = sqlite3.connect(database)
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM login WHERE username = "%s" AND password = "%s"' % (username, password))
    user = cursor.fetchone()
    if user:
        response = create_response('Login successful!')
        response.set_cookie('SESSIONID', str(random.randint(1,99999999999999999999999)),httponly=False,secure=False)
        response.set_cookie('TESTID1', str("TESTSTRING1"), httponly=True,secure=True)
        response.set_cookie('TESTID2', str("TESTSTRING2"))
        return response
    else:
        response = create_response('Login failed!')
        response.delete_cookie('username')
        return response, 401

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8080, debug=True)
```

The code snippet provided exhibits several security vulnerabilities. Below is a comprehensive list, along with explanations and potential impact:

1. **SQL Injection (CWE-89):**
    - The `login` function constructs an SQL query by directly embedding user inputs (`username` and `password`) into the SQL command. This approach is vulnerable to SQL injection, where an attacker can manipulate the SQL query by injecting malicious SQL commands.
    - **Impact**: An attacker could gain unauthorized access to user data, retrieve or manipulate database content, and potentially compromise the database or associated server.

2. **Hardcoded Credentials (CWE-259):**
    - The `setup` function contains a hardcoded secret password (`SECRET_PASSWORD = "letMeIn!"`), which is a security risk if the source code is exposed.
    - **Impact**: Hardcoded secrets can be easily discovered through code analysis and used to gain unauthorized access.

3. **Insecure Cookie Handling (CWE-614, CWE-1004):**
    - The `login` function sets a `SESSIONID` cookie without `httponly` and `secure` flags. This exposes the cookie to client-side scripts and potential interception over non-HTTPS connections.
    - **Impact**: Vulnerable to access through XSS attacks and interception over insecure networks, potentially leading to session hijacking.

4. **Sensitive Information Exposure (CWE-532):**
    - The application prints a secret password to the console, which could be logged or observed by unauthorized parties.
    - **Impact**: Exposure of sensitive credentials that could be used for further attacks.

5. **Improper Access Control (CWE-284):**
    - The `/setup` route can be accessed without authentication and is capable of initializing the database or resetting it, which should be a protected operation.
    - **Impact**: Unauthorized users could initialize or alter the database setup, leading to data loss or corruption.

6. **Security Misconfiguration (CWE-16):**
    - Flask is run with `debug=True` in the main function, which can leak detailed error information and possibly sensitive data in a production environment.
    - **Impact**: Detailed error messages can provide attackers with insights into the backend, facilitating further attacks.

7. **Cross-Origin Resource Sharing (CORS) Misconfiguration (CWE-346):**

   - CORS is configured to allow all domains (`` `Access-Control-Allow-Origin: '*'` ``). This permissive setting can allow any external domain to interact with the API, bypassing the same-origin policy.
   - **Impact**: Potentially exposes data to malicious websites, which can interact with the web service from any origin.

8. **Default User Credentials (CWE-798):**

   - The `` `/setup` `` route creates a user with default credentials that are weak (`` `user_1` `` with password `` `123456` ``), which could be easily guessed or brute-forced.
   - **Impact**: Easy for attackers to gain unauthorized access using default credentials.

To address these vulnerabilities:

- Use parameterized queries to prevent SQL injection.
- Remove hardcoded secrets and use environment variables or a secure secret management approach.
- Configure cookies with `` `httponly` `` and `` `secure` `` flags.
- Avoid printing or logging sensitive information.
- Secure sensitive routes with proper authentication and authorization checks.
- Do not run applications in debug mode in production.
- Restrict CORS settings to trusted domains.
- Avoid using default or weak credentials; enforce strong password policies.

List the number of security vulnerabilities in the code and add comments in the code where it exists.

The code snippet contains eight distinct security vulnerabilities. Below, I have added comments directly within the code to mark where each vulnerability exists and what it pertains to:

```python
import sqlite3, random
from flask import Flask, abort, request, jsonify
from flask_cors import CORS

app = Flask(__name__)
CORS(app)  # CWE-346: CORS Misconfiguration - Allows requests from all origins.
database = './login.db'

def create_response(message):
    response = jsonify({'message': message})
    response.headers.add('Access-Control-Allow-Origin', '*')  # CWE-346: CORS
Misconfiguration - Allows requests from all origins.
    return response

@app.route('/setup', methods=['POST'])
def setup():
    connection = sqlite3.connect(database)
    SECRET_PASSWORD = "letMeIn!"  # CWE-259: Hardcoded Credentials - Hardcoded secret
password.
    THIS_IS_A_VARIABLE = "WBneKJw1fHch8Qd3XFUS"
    print("Super Secret Password SSH Server Password to 10.10.10.1:22: " + SECRET_PASSWORD)
# CWE-532: Sensitive Information Exposure - Printing secret password.
    connection.executescript('CREATE TABLE IF NOT EXISTS login(username TEXT NOT NULL
UNIQUE, password TEXT NOT NULL);')
    connection.executescript('INSERT OR IGNORE INTO login VALUES("user_1","123456");')  #
CWE-798: Default User Credentials - Default credentials setup.
    return create_response('Setup done!')

@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']
    connection = sqlite3.connect(database)
    cursor = connection.cursor()
    # CWE-89: SQL Injection - User input directly used in SQL command.
    cursor.execute('SELECT * FROM login WHERE username = "%s" AND password = "%s"' %
(username, password))
    user = cursor.fetchone()
    if user:
        response = create_response('Login successful!')
        # CWE-1004: Insecure Cookie Handling - Cookie not set with HttpOnly or Secure
attributes.
        response.set_cookie('SESSIONID', str(random.randint(1,99999999999999999999999)),
httponly=False, secure=False)
        response.set_cookie('TESTID1', str("TESTSTRING1"), httponly=True, secure=True)
        response.set_cookie('TESTID2', str("TESTSTRING2"))
        return response
    else:
        response = create_response('Login failed!')
        response.delete_cookie('username')  # CWE-614: Insecure Cookie Handling -
Potentially unnecessary cookie manipulation.
        return response, 401

if __name__ == "__main__":
    # CWE-215: Information Exposure Through Debug Information - Running the app in debug
mode.
    app.run(host='0.0.0.0', port=8080, debug=True)
    # CWE-284: Improper Access Control - Server runs on all interfaces without any
restrictions.
```

In this revised version, comments have been added to highlight each security issue according to its Common Weakness Enumeration (CWE) identifier and provide a brief description of the problem. This will help in understanding the specific vulnerabilities directly in the context of the code.

ChatGPT can make mistakes. Check important info.