Secure Codebase Review

Below you see a simple web application developed with the python flask module which has some security vulnerabilities. We'll review the code to see what security vulnerabilities it has, what risks it poses, and how we can make it safer.

```
# main.py

from flask import Flask, request, render_template_string, redirect, session

import hashlib


app = Flask(__name__)

app.secret_key = 'hardcodedsecretkey'


# Fake in-memory database (for demo purposes)

users = {'admin': '5f4dcc3b5aa765d61d8327deb882cf99'}  # Password: 'password' (MD5 hash)


# Route: Login page

@app.route('/login', methods=['GET', 'POST'])

def login():

    if request.method == 'POST':

        username = request.form['username']

        password = request.form['password']


        hashed_password = hashlib.md5(password.encode()).hexdigest()

        if username in users and users[username] == hashed_password:

            session['user'] = username

            return redirect('/dashboard')

        else:

            return "Login Failed", 401


    return '''

        <form method="post">

            Username: <input type="text" name="username"><br>

            Password: <input type="password" name="password"><br>

            <input type="submit" value="Login">

        </form>
    '''

# Route: Dashboard
@app.route('/dashboard')
def dashboard():
    if 'user' not in session:
        return "You must be logged in!", 401
    return f"Welcome to your dashboard, {session['user']}!"


# Route: Search
@app.route('/search')
def search():
    query = request.args.get('query')


    return render_template_string(f"<h1>Search Results for: {query}</h1>")


# Route: Download File
@app.route('/download/<filename>')
def download(filename):
    return f"Downloading {filename}"


if __name__ == "__main__":
    app.run(debug=True)
```

Given "**main.py**" application has some security vulnerabilities and our task is identify and document these vulnerabilities as part of the secure code review and provide recommendations for fixing each issue.

Vulnerabilities:

1. **Hardcoded Secret Key:**
    - The **app.secret_key** is hardcoded and can easily be exposed, if an attacker gains access to the code, they can tamper with the session data, which compromises security. In a real application, secrets should be stored securely, such as in environment variables or secret management tools (e.g., AWS Secrets Manager).

2. **Authentication Bypass Risk:**
    - The authentication logic is weak. There is no **rate limiting** or **account lockout** for repeated failed attempts, which can lead to **brute force attacks**.

3. **Improper Password Hashing:**
    - The password is hashed using **MD5**, which is an **insecure hashing algorithm** vulnerable to collision attacks.

4. **Cross-Site Scripting (XSS):**
    - The search query is **directly inserted** into the HTML response without **sanitization**, which allows for **XSS attacks**. If a user enters **<script>**alert**('XSS');</script>** as the query, it will be executed in the browser, causing an XSS attack.

5. **Insecure Direct Object Access (IDOR):**
    - The download route allows users to specify any filename, which could lead to a **directory traversal attack**. If a user enters **../../etc/passwd** as the filename, they could potentially access system files.

6. **No Input Validation:**
    - There is **no validation on the user inputs** (e.g., username, password, search queries), making the application vulnerable to a variety of injection attacks, including **SQL injection** if interacting with a database.

7. **Weak Error Handling:**
    - Error messages reveal **sensitive information**, such as "**Login Failed**" in the login route, which can give attackers insight into whether the username or password was incorrect.

8. **Debug=True:**
    - The application runs in **debug mode**, which provides **detailed error messages** and can **expose sensitive information** to attackers.

Recommendations:

1. **Hardcoded Secret Key:**
    - Hardcoded secret key issue can be fixed by **using environment variables**, which is a more secure approach. The **app.secret_key** should be stored securely in an environment variable instead of being hardcoded in the codebase.

```
import os
from flask import Flask, request, render_template_string, redirect, session
import hashlib

app = Flask(__name__)

# Load the secret key from an environment variable (use 'os.getenv')
app.secret_key = os.getenv('FLASK_SECRET_KEY', 'defaultfallbackkey')  # 
'defaultfallbackkey' only as a fallback for local testing
```

2. **Authentication Bypass Risk:**
    - For mitigating the Authentication Bypass Risk, we can implement "**Rate Limiting**" (restricts the number of login attempts an IP address can make in a specific period, reducing the risk of brute-force attacks) and "**Account Lockout**" (after a certain number of failed attempts, temporarily lock the account to prevent further login attempts) strategies. In this case, we can import "**Flask-Limiter**" library for rate limiting and use a dictionary to track failed login attempts. This solution adds a strong layer of protection against brute-force login attacks and ensures better security.

```
from flask import Flask, request, render_template_string, redirect, session, abort
import hashlib
import time
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

app = Flask(__name__)

# Rate Limiter to prevent brute force attacks
limiter = Limiter(
    get_remote_address,
    app=app,
    default_limits=["10 per minute"]  # Limit: 10 login attempts per minute
)

# Store failed login attempts for account lockout
failed_attempts = {}
LOCKOUT_THRESHOLD = 3  # Maximum failed attempts before lockout
LOCKOUT_TIME = 60 * 5  # 5 minutes lockout period

@app.route('/login', methods=['GET', 'POST'])
@limiter.limit("10 per minute")  # Apply rate limit to login attempts
def login():
    ip_address = request.remote_addr
    current_time = time.time()

    # Check if the user is locked out
    if ip_address in failed_attempts:
        attempts, last_attempt_time = failed_attempts[ip_address]

        # If the account is still locked, deny access
        if attempts >= LOCKOUT_THRESHOLD and (current_time - last_attempt_time) < 
LOCKOUT_TIME:
            return "Account is temporarily locked. Try again later.", 403

        # Reset lockout if the lockout time has passed
        elif (current_time - last_attempt_time) > LOCKOUT_TIME:
            failed_attempts[ip_address] = (0, current_time)

    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
    # Secure password hashing (replace MD5)
    hashed_password = hashlib.sha256(password.encode()).hexdigest()

    # Vulnerable authentication logic
    if username in users and users[username] == hashed_password:
        session['user'] = username
        # Reset failed attempts after successful login
        if ip_address in failed_attempts:
            del failed_attempts[ip_address]
        return redirect('/dashboard')
    else:
        # Update failed attempts for the IP
        if ip_address in failed_attempts:
            attempts, last_attempt_time = failed_attempts[ip_address]
            failed_attempts[ip_address] = (attempts + 1, current_time)
        else:
            failed_attempts[ip_address] = (1, current_time)

        return "Login Failed", 401

return '''
    <form method="post">
        Username: <input type="text" name="username"><br>
        Password: <input type="password" name="password"><br>
        <input type="submit" value="Login">
    </form>
'''
```

3. **Improper Password Hashing:**
    - To fix the insecure password hashing (MD5) vulnerability, we can replace it with a more secure hashing algorithm such as **bcrypt**. Bcrypt is designed for password hashing because it includes a **salt** (a unique random value for each password) and has a configurable work factor to make brute-force attacks more difficult.

```
bcrypt.hashpw(password.encode(), bcrypt.gensalt()):
```

- This hashes the password with bcrypt. It automatically generates a salt and applies it to the password.
- The salt ensures that even if two users have the same password, their hashed values will be different.

```
bcrypt.checkpw(password.encode(), users[username]):
```

- This checks whether the password provided by the user matches the stored hashed password using bcrypt’s checkpw method.
- The encoded password is compared against the stored hashed password for verification.

```
import bcrypt
from flask import Flask, request, render_template_string, redirect, session

app = Flask(__name__)

# In-memory user database (passwords will be hashed with bcrypt)
users = {}

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        # Check if the user exists
        if username in users:
            # Verify the hashed password using bcrypt
            if bcrypt.checkpw(password.encode(), users[username]):
                session['user'] = username
                return redirect('/dashboard')
            else:
                return "Login Failed", 401
        else:
            return "Login Failed", 401

    return '''
        <form method="post">
            Username: <input type="text" name="username"><br>
            Password: <input type="password" name="password"><br>
            <input type="submit" value="Login">
        </form>
    '''

# Route to create a new user (for demonstration)
@app.route('/create_user', methods=['POST'])
def create_user():
    username = request.form['username']
    password = request.form['password']

    # Hash the password using bcrypt with a salt
    hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt())

    # Store the hashed password in the 'users' dictionary
    users[username] = hashed_password

    return "User created successfully!"
```

4. **Cross-Site Scripting (XSS):**
   - To fix the Cross-Site Scripting (XSS) vulnerability in the search route, we need to ensure that any user input is properly **escaped or sanitized** before rendering it to the HTML. This prevents attackers from injecting malicious code (e.g., JavaScript) into the response. Instead

of directly rendering the user input using **render_template_string** (which is insecure), we will either use Flask's built-in **render_template** method, which automatically escapes HTML characters. For example, if a user submits the query **<script>alert('XSS');</script>**, Flask would escape it like this: **&lt;script&gt;alert('XSS');&lt;/script&gt;**

```
# main.py
from flask import Flask, request, render_template, escape

app = Flask(__name__)

@app.route('/search')
def search():
    query = request.args.get('query', '')

    # Use render_template to safely render the query
    return render_template('search.html', query=query)
```

```
<!-- search.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Search Results</title>
</head>
<body>
    <h1>Search Results for: {{ query }}</h1>
</body>
</html>
```

5. **Insecure Direct Object Reference (IDOR) and Directory Traversal:**
   - To remedy the **Insecure Direct Object Reference (IDOR)** and **Directory Traversal** vulnerabilities in the download route, we need to implement several security measures to ensure that users can only download files they are allowed to access, and we prevent access to sensitive system files.
     1. Restrict file paths: Ensure that users cannot specify arbitrary paths (like **../../etc/passwd**) by sanitizing the filename input.
     2. Validate file paths: Ensure the file requested exists in a predefined directory (i.e., a directory that holds only the files the user is allowed to download).
     3. Use **send_from_directory**: Flask provides a safe method to send files from a specific directory, which avoids path traversal vulnerabilities.
     4. Whitelist files: Use a whitelist of **allowable file extensions** or names to further limit what users can access.

```
import os
from flask import Flask, request, send_from_directory, abort, safe_join

app = Flask(__name__)
```

```
# Define the directory where downloadable files are stored
DOWNLOAD_DIRECTORY = os.path.abspath("downloads")

# Allowed file extensions
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'jpg', 'png'}

def allowed_file(filename):
    """Check if the file has a valid extension."""
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/download/<filename>')
def download(filename):
    if not allowed_file(filename):
        return abort(403, description="File type not allowed")

    # Ensure the file path is safe and within the allowed directory
    safe_path = safe_join(DOWNLOAD_DIRECTORY, filename)
    if not os.path.isfile(safe_path):
        return abort(404, description="File not found")

    # Send the file from the allowed directory
    return send_from_directory(DOWNLOAD_DIRECTORY, filename)
```

6. **No Input Validation:**
   - To address the **"No Input Validation"** vulnerability, particularly regarding **SQL Injection** and similar attacks, it is essential to validate and sanitize user inputs effectively. The recommended approach for SQL queries is to utilize **parameterized queries or prepared statements**, as they help prevent SQL injection attacks.
     1. The query is using placeholders (?) and passing the user inputs as parameters ((username, password)), which ensures the database properly escapes the input, preventing SQL injection.
     2. We check if the **username** and **password** are provided and ensure they are not empty. This simple validation prevents issues like submitting an empty form or missing fields.
     3. The **get_db_connection()** function ensures that each request gets its own database connection, which is then closed after use.

```
import sqlite3

from flask import Flask, request, render_template_string



app = Flask(__name__)



# Establish connection to the database

def get_db_connection():

    conn = sqlite3.connect('example.db')

    conn.row_factory = sqlite3.Row
    return conn

@app.route('/login', methods=['POST'])
def login():
    # Get user inputs
    username = request.form.get('username')
    password = request.form.get('password')

    # Validate inputs: Ensure they are not empty
    if not username or not password:
        return "Username and password are required!", 400

    # Use a parameterized query to prevent SQL injection
    conn = get_db_connection()
    query = "SELECT * FROM users WHERE username = ? AND password = ?"
    result = conn.execute(query, (username, password)).fetchone()
    conn.close()

    if result:
        return "Login successful!"
    else:
        return "Invalid credentials!"
```

7. **Weak Error Handling:**
   - We need to ensure that error messages provided to users are **generic** and **do not reveal sensitive information**. In the case of a login route, disclosing whether the username or password is incorrect can give attackers useful information for brute force or enumeration attacks. The message "**Invalid credentials!**" is displayed whether the username or password is incorrect. This prevents attackers from knowing whether the username was valid but the password was wrong (or the opposite), which could otherwise help them in brute-force or enumeration attacks. Additionally, we can also internally **log the specific error** (e.g., whether it was a username or password issue) for administrators, while keeping the error message generic for the user.

```
import sqlite3
from flask import Flask, request, render_template_string

app = Flask(__name__)
```

```
def get_db_connection():
    conn = sqlite3.connect('example.db')
    conn.row_factory = sqlite3.Row
    return conn

@app.route('/login', methods=['POST'])
def login():
    # Get user inputs
    username = request.form.get('username')
    password = request.form.get('password')

    # Validate inputs: Ensure they are not empty
    if not username or not password:
        return "Username and password are required!", 400

    # Use a parameterized query to prevent SQL injection
    conn = get_db_connection()
    query = "SELECT * FROM users WHERE username = ?"
    user = conn.execute(query, (username,)).fetchone()

    conn.close()

    # Check if the user exists and password matches
    if user is None or user['password'] != password:
        # Generic error message to avoid revealing whether username or password was incorrect
        return "Invalid credentials!", 401

    # Successful login
    return "Login successful!"
```

Logging for Administrators:

```
import logging

# Log the error internally
def log_failed_login(username):
    logging.warning(f"Failed login attempt for user: {username}")

@app.route('/login', methods=['POST'])
def login():
    username = request.form.get('username')
    password = request.form.get('password')

    if not username or not password:
        return "Username and password are required!", 400

    conn = get_db_connection()
    query = "SELECT * FROM users WHERE username = ?"
    user = conn.execute(query, (username,)).fetchone()
    conn.close()
    if user is None or user['password'] != password:
        log_failed_login(username)  # Log the failed attempt
        return "Invalid credentials!", 401

    return "Login successful!"
```

8. **Debug=True:**
   - Debug mode **should never be enabled** in a production environment, setting **debug=False** or **omitting the debug** argument ensures that the application does not display **detailed error messages**, thus improving security in a production environment.

```
if __name__ == "__main__":

# Ensure debug mode is disabled in production
    app.run(debug=False)
```