AI generated code review

created by SAIST

OpenAI:gpt-5 2025-08-10 22:53

Contents

1 Summary

Application Security Summary (app.py)

Overall Risk Assessment - Overall risk: Critical. Multiple vectors enable remote code execution, authentication/authorization bypass, and data exfiltration. Immediate remediation required before production use.

Critical Severity

1) Injection and Remote Code Execution - Issues: - SQL injection in authentication by interpolating user-supplied credentials into queries. - Command injection via attacker-controlled cmd parameter executed with os.popen. - Executing user-uploaded Python scripts (arbitrary code execution). - Running with debug=True exposes interactive debugger (remote code execution). - Impact: Full database compromise, server takeover, data theft, and persistence by attackers. - Key Recommendations: - Use parameterized queries and compare hashed passwords only. - Remove shell execution; if unavoidable, use subprocess with no shell, fixed arguments, and strict validation. - Never execute user uploads; remove feature or use strongly isolated sandbox workers with strict controls. - Disable debug in all non-local environments; deploy behind a production WSGI server with proper error handling.

2) Broken Authorization and Privilege Escalation - Issues: - Role taken from client input (base64) allowing self-assignment of admin. - Impact: Immediate privilege escalation to admin, full administrative control. - Key Recommendations: - Ignore client-provided roles; set roles server-side with safe defaults and restrict role changes to admins with server-side checks.

High Severity

1) Access Control Missing - Issues: - /admin/download lacks admin privilege verification. - Impact: Unauthorized access to sensitive files by any authenticated user. - Key Recommendations: - Enforce RBAC on this and all administrative routes (e.g., verify session role == admin).

2) Path and File Handling - Issues: - Directory traversal via unvalidated filename joined into file paths. - Unsafe file upload handling (unvalidated names/types, potential traversal). - Impact: Reading arbitrary files, potential execution or overwriting, data exfiltration. - Key Recommendations: - Normalize and validate paths; reject traversal sequences; restrict to a dedicated directory using safe-join. - Validate uploads by extension and MIME/content; use secure_filename; store outside web root; enforce size limits and scanning.

3) Credential and Secret Management - Issues: - Passwords stored/handled in plaintext. - Resetting passwords to a hard-coded weak value. - Static, hard-coded secret key. - Impact: Rapid account takeover, offline cracking risk, session tampering if key is leaked. - Key Recommendations: - Hash passwords with bcrypt, Argon2, or PBKDF2 with salts; compare hashes only. - Use strong random temporary credentials or tokenized reset flows; force immediate change via secure UX. - Load strong random secret keys per environment from secure storage; rotate periodically.

4) Transport and Session Security - Issues: - Session cookies allowed over HTTP (no Secure flag). - Impact: Session hijacking via network interception. - Key Recommendations: - Serve exclusively over HTTPS and set SESSION_COOKIE_SECURE = True.

5) CSRF on High-Impact Operations - Issues: - CSRF missing on admin-only password reset and on user deletion endpoints. - Impact: Cross-site attacks can reset passwords or delete users when a privileged user visits a malicious page. - Key Recommendations: - Require CSRF tokens on all state-changing routes; consider re-authentication for destructive actions; use SameSite cookies.

Medium Severity

1) CSRF on Other State-Changing Endpoints - Issues: - CSRF missing on additional POST routes including clearing chat history and other state changes. - Impact: Unauthorized actions via victim's browser. - Key Recommendations: - Implement a CSRF framework (e.g., Flask-WTF CSRFProtect) and validate tokens on all non-idempotent requests.

2) Cookie Hardening - Issues: - HttpOnly disabled on session cookies. - Impact: Increases impact of XSS by exposing session cookies to client-side scripts. - Key Recommendations: - Set SESSION_COOKIE_HTTPONLY = True.

Low Severity

1) Error Handling and Information Disclosure - Issues: - Returning raw exception messages to clients. - Impact: Leaks internal details useful for targeted attacks. - Key Recommendations: - Log detailed errors server-side; return generic error responses to clients.

Prioritized Remediation Plan

1) Immediately eliminate remote code execution and injection risks: - Disable debug mode. - Remove os.popen shell execution or replace with safe subprocess usage. - Stop executing uploaded scripts. - Fix SQL injection using parameterized queries and update auth to use password hashes.

2) Fix broken authorization: - Enforce server-side role management; remove trust in client-supplied roles. - Add strict RBAC checks to /admin/download and all admin routes.

3) Secure credentials and sessions: - Hash and salt all passwords; migrate existing passwords via reset flow. - Replace hard-coded secret key with a strong, rotated secret from secure storage. - Replace hard-coded password resets with secure token-based flows.

4) Enforce transport and cookie security: - Serve exclusively over HTTPS; set Secure and HttpOnly cookie flags appropriately.

5) Implement comprehensive CSRF defense: - Add CSRF tokens and validation to all state-changing endpoints. - Require re-authentication or step-up verification for destructive actions.

6) Harden file and path handling: - Implement safe path normalization and directory restrictions. - Validate upload filenames and content; store outside web root with size limits and scanning.

7) Improve error handling: - Return generic error messages and centralize server-side logging with proper sanitization.

2 Findings

2.1 CWE-89 - app.py - SQL Injection in login query

Priority: Critical

Issue: User-supplied credentials are interpolated directly into an SQL query, allowing attackers to inject arbitrary SQL to bypass authentication or exfiltrate data.

Recommendation: Use parameterized queries with placeholders (e.g., cursor.execute("SELECT * FROM users WHERE username = ? AND password = ?", (username, password))) and store/compare password hashes instead of plaintext.

```
80  def index():
81      return render_template('index.html')
82
83  @app.route('/login', methods=['GET', 'POST'])
84  def login():
85      if request.method == 'POST':
86          username = request.form.get('username')
87          password = request.form.get('password')
88
89
90          query = f"SELECT * FROM users WHERE username = '{username}' AND password =
            ↪  '{password}'"
91
92          try:
93              db = get_db()
94              user = db.execute(query).fetchone()
95
96              if user:
97                  session['user_id'] = user['id']
98                  session['username'] = user['username']
99                  session['role'] = user['role']
100                 return redirect(url_for('account'))
```

Figure 1: **app.py** on line **90**

2.2 CWE-78 - app.py - Remote Command Execution via system_info endpoint

Priority: Critical

Issue: An attacker-controlled cmd parameter is executed by the shell using os.popen, enabling arbitrary command execution on the server.

Recommendation: Remove this functionality or strictly whitelist allowed commands and avoid shell execution. If absolutely necessary, use subprocess with a fixed argument list and no shell, and validate inputs rigorously.

```
277 def system_info():
278
279     cmd = request.args.get('cmd', '')
280
281
282     if not cmd:
283         return jsonify({"output": "No command provided"}), 400
284
285     try:
286
287         result = os.popen(cmd).read()
288     except Exception as e:
289         result = f"Error: {str(e)}"
290
291     return jsonify({"output": result})
292
293
294 @app.route('/os_info', methods=['GET'])
295 def os_info():
296     default_file='testing.txt'
297
```

Figure 2: **app.py** on line **287**

2.3 CWE-94 - app.py - Arbitrary code execution by running user-uploaded .py files

Priority: Critical

Issue: The application executes uploaded Python scripts, allowing authenticated users to run arbitrary code on the server.

Recommendation: Never execute user-uploaded files. Remove this functionality or sandbox strictly isolated worker environments with strong restrictions if truly required.

```
234 filename = file.filename
235
236 # Save the uploaded .py file in the appropriate folder
237 file_path = os.path.join(app.config['PROFILE_PIC_FOLDER'], filename)
238 file.save(file_path)
239
240 # If it's a .py file, execute it
241 if filename.endswith('.py'):
242     try:
243         # Execute the Python script using subprocess
244         result = subprocess.run(['python', file_path], capture_output=True, text=True)
245
246         # Check if execution was successful
247         if result.returncode == 0:
248             print("Python script executed successfully.")
249             print(result.stdout)  # Optional: print the output of the script
250         else:
251             print("Python script execution failed.")
252             print(result.stderr)
253     except Exception as e:
254         print(f"Error executing Python script: {e}")
```

Figure 3: **app.py** on line **244**

2.4 CWE-22 - app.py - Path Traversal in os_info file read

Priority: High

Issue: Unvalidated filename from the request is joined into a file path, allowing directory traversal to read arbitrary files (e.g., ../../etc/passwd).

Recommendation: Normalize and validate the path, reject any path containing path traversal sequences, and restrict access to a specific directory using safe join utilities; consider using a whitelist of allowed files.

```
289         result = f"Error: {str(e)}"
290
291     return jsonify({"output": result})
292
293
294 @app.route('/os_info', methods=['GET'])
295 def os_info():
296     default_file='testing.txt'
297
298     filename= request.args.get('filename', 'testing.txt')
299     file_path = os.path.join(STATIC_FOLDER, filename)
300
301
302
303     try:
304         with open(file_path, 'r') as file:
305             content = file.read()
306     except Exception as e:
307         content = f"Error: {str(e)}"
308
309     return jsonify({"file_content": content})
```

Figure 4: **app.py** on line **299**

2.5 CWE-434 - app.py - Unrestricted file upload (no validation/sanitization of filename)

Priority: High

Issue: Uploaded file name is used directly without validation (e.g., no secure_filename, no extension/content checks), enabling upload of dangerous files and potential path traversal.

Recommendation: Validate file types with both extension and MIME/content checks, use secure_filename, store uploads outside the web root, and enforce size limits and scanning.

```
227
228 if not user:
229     return redirect(url_for('login'))
230
231 if request.method == 'POST':
232     file = request.files['profile_picture']
233     if file:
234         filename = file.filename
235
236         # Save the uploaded .py file in the appropriate folder
237         file_path = os.path.join(app.config['PROFILE_PIC_FOLDER'], filename)
238         file.save(file_path)
239
240         # If it's a .py file, execute it
241         if filename.endswith('.py'):
242             try:
243                 # Execute the Python script using subprocess
244                 result = subprocess.run(['python', file_path], capture_output=True, text=True)
245
246                 # Check if execution was successful
247                 if result.returncode == 0:
```

Figure 5: **app.py** on line **237**

2.6 CWE-269 - app.py - Privilege escalation via user-controlled role during registration

Priority: High

Issue: The role is taken from a client-supplied field (base64-encoded) and stored directly, allowing attackers to self-assign elevated roles like admin.

Recommendation: Ignore client-provided roles; set roles server-side to a safe default (e.g., 'user') and restrict role changes to admins with server-side validation.

```
184
185
186 @app.route('/register', methods=['GET', 'POST'])
187 def register():
188     if request.method == 'POST':
189         username = request.form['username']
190         password = request.form['password']
191         email = request.form['email']
192         role = request.form['role']
193
194         decoded_role = base64.b64decode(role).decode('utf-8')
195
196         db = get_db()
197         cursor = db.cursor()
198
199         # Check if username already exists
200         cursor.execute("SELECT * FROM users WHERE username = ?", (username,))
201         existing_user = cursor.fetchone()
202
203         if existing_user:
204             flash("Username already exists. Please choose a different one.", "danger")
```

Figure 6: **app.py** on line **194**

2.7 CWE-285 - app.py - Admin download endpoint lacks authorization checks

Priority: Medium

Issue: The /admin/download route does not verify admin privileges, allowing any authenticated user to download sensitive files.

Recommendation: Enforce role-based access control in this handler (e.g., check session role == 'admin') before serving files.

```
151     # clear messages KEEP THIS!!
152     chat_messages.clear()
153     return redirect(url_for('chat'))
154
155
156
157
158
159
160 @app.route('/admin/download/<int:file_id>', methods=['GET'])
161 def download(file_id):
162     # Define a mapping of file IDs to filenames
163     file_mapping = {
164         1: "top-secret-company-strategy-2024.doc",
165         2: "admin_notes.txt",
166         3: "payroll.csv",  # You can add more file IDs and filenames here
167     }
168
169     # Get the filename from the mapping based on the file ID
170     filename = file_mapping.get(file_id)
171
```

Figure 7: **app.py** on line **161**

2.8 CWE-256 - app.py - Plaintext password storage

Priority: Medium

Issue: Passwords are stored and handled in plaintext throughout the application, risking credential disclosure on compromise.

Recommendation: Hash passwords with a strong adaptive algorithm (bcrypt, Argon2, or PBKDF2) and use salts; update authentication logic to compare hashes.

```
198
199         # Check if username already exists
200         cursor.execute("SELECT * FROM users WHERE username = ?", (username,))
201         existing_user = cursor.fetchone()
202
203         if existing_user:
204             flash("Username already exists. Please choose a different one.", "danger")
205             return redirect(url_for('register'))
206
207
208         cursor.execute("INSERT INTO users (username, password, email, role) VALUES (?, ?, ?,
            ↪  ?)",
209                        (username, password, email, decoded_role))
210         db.commit()
211
212         flash("Registration successful! You can now log in.", "success")
213         return redirect(url_for('login'))
214
215     return render_template('register.html')
216
217
218 @app.route('/account', methods=['GET', 'POST'])
```

Figure 8: **app.py** on line **208**

2.9 CWE-352 - app.py - Missing CSRF protection on admin password reset

Priority: Medium

Issue: The admin-only POST endpoint does not include CSRF protections, enabling cross-site attacks to reset user passwords.

Recommendation: Require and validate CSRF tokens on this route and others that change state.

```
349
350     users = db.execute("SELECT id, username, email FROM users WHERE username != 'admin' LIMIT ?
        ↪  OFFSET ?", (per_page, offset)).fetchall()
351
352
353     total_users = db.execute("SELECT COUNT(*) FROM users WHERE username !=
        ↪  'admin'").fetchone()[0]
354     total_pages = (total_users // per_page) + (1 if total_users % per_page > 0 else 0)
355
356     return render_template('admin.html', users=users, page=page, total_pages=total_pages)
357
358
359 @app.route('/admin/reset_password/<int:user_id>', methods=['POST'])
360 def reset_password(user_id):
361     if 'username' not in session or session.get('role') != 'admin':
362         return redirect(url_for('login'))
363
364     db = get_db()
365
366     default_password = 'qwerty'
367     db.execute("UPDATE users SET password = ? WHERE id = ?", (default_password, user_id))
368     db.commit()
369
```

Figure 9: **app.py** on line **359**

2.10 CWE-352 - app.py - Missing CSRF protection on admin delete user

Priority: Medium

Issue: Deleting users is a sensitive operation exposed via POST without CSRF protection, susceptible to cross-site request forgery.

Recommendation: Add CSRF tokens and validate them server-side; consider same-site cookies and re-authentication for destructive actions.

```
362         return redirect(url_for('login'))
363
364     db = get_db()
365
366     default_password = 'qwerty'
367     db.execute("UPDATE users SET password = ? WHERE id = ?", (default_password, user_id))
368     db.commit()
369
370     return redirect(url_for('admin'))
371
372 @app.route('/admin/delete_user/<int:user_id>', methods=['POST'])
373 def delete_user(user_id):
374     if 'username' not in session or session.get('role') != 'admin':
375         return redirect(url_for('login'))
376
377     db = get_db()
378     db.execute("DELETE FROM users WHERE id = ?", (user_id,))
379     db.commit()
380
381     return redirect(url_for('admin'))
382
```

Figure 10: **app.py** on line **372**

2.11 CWE-489 - app.py - Flask debug mode enabled in production

Priority: Medium

Issue: Running with debug=True exposes the interactive debugger and detailed error information, potentially leading to remote code execution.

Recommendation: Disable debug mode in production and use a production WSGI server (e.g., gunicorn/uwsgi) with proper error handling/logging.

```
403             if result:
404                 return render_template('search_results.html', results=result)
405             else:
406                 return 'No results found.'
407         else:
408             return 'No parameter provided', 400
409 
410     return render_template('search.html')
411 
412 if __name__ == '__main__':
413     app.run(host='0.0.0.0', port=80, debug=True)
414 
```

Figure 11: **app.py** on line **413**

2.12 CWE-798 - app.py - Hard-coded Flask secret key

Priority: Medium

Issue: A static, hard-coded secret key weakens session integrity and enables cookie tampering if leaked.

Recommendation: Load a strong, random secret key from environment or secrets manager per deployment and rotate regularly.

```
2  import base64
3  import subprocess
4  import random
5  import sqlite3
6  import pickle
7  import os
8  import re
9  from werkzeug.utils import secure_filename
10
11 app = Flask(__name__)
12 app.secret_key = 'your_secret_key'
13
14 DATABASE = 'sql_injection_demo.db'
15
16 ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif'}
17 STATIC_FOLDER = 'static'
18
19 def allowed_file(filename):
20     return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
21
22
```

Figure 12: **app.py** on line **12**

2.13 CWE-1004 - app.py - HttpOnly flag disabled for session cookie

Priority: Medium

Issue: Disabling HttpOnly allows client-side scripts to access the session cookie, increasing the impact of XSS.

Recommendation: Set SESSION_COOKIE_HTTPONLY = True to prevent JavaScript access to the session cookie.

```
18
19 def allowed_file(filename):
20     return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
21
22
23 PROFILE_PIC_FOLDER = os.path.join(app.root_path, 'static', 'profile_pics')
24 if not os.path.exists(PROFILE_PIC_FOLDER):
25     os.makedirs(PROFILE_PIC_FOLDER)
26
27 app.config['PROFILE_PIC_FOLDER'] = PROFILE_PIC_FOLDER
28 app.config['SESSION_COOKIE_HTTPONLY'] = False
29 app.config['SESSION_COOKIE_SECURE'] = False
30
31 def get_db():
32     conn = sqlite3.connect(DATABASE)
33     conn.row_factory = sqlite3.Row
34     return conn
35
36 chat_messages = []
37
38 # set up the db
```

Figure 13: **app.py** on line **28**

2.14 CWE-614 - app.py - Secure flag disabled for session cookie

Priority: Medium

Issue: Session cookies can be transmitted over unencrypted HTTP, exposing them to interception.

Recommendation: Set SESSION_COOKIE_SECURE = True and serve the application exclusively over HTTPS.

```
19 def allowed_file(filename):
20     return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
21
22
23 PROFILE_PIC_FOLDER = os.path.join(app.root_path, 'static', 'profile_pics')
24 if not os.path.exists(PROFILE_PIC_FOLDER):
25     os.makedirs(PROFILE_PIC_FOLDER)
26
27 app.config['PROFILE_PIC_FOLDER'] = PROFILE_PIC_FOLDER
28 app.config['SESSION_COOKIE_HTTPONLY'] = False
29 app.config['SESSION_COOKIE_SECURE'] = False
30
31 def get_db():
32     conn = sqlite3.connect(DATABASE)
33     conn.row_factory = sqlite3.Row
34     return conn
35
36 chat_messages = []
37
38 # set up the db
39 def init_db():
```

Figure 14: **app.py** on line **29**

2.15 CWE-259 - app.py - Hard-coded weak default password in admin reset

Priority: Medium

Issue: Resetting user passwords to a hard-coded weak value enables easy account takeover if not changed immediately.

Recommendation: Generate strong random temporary passwords or password reset tokens and require the user to set a new password via a secure flow; never use hard-coded defaults.

```
356     return render_template('admin.html', users=users, page=page, total_pages=total_pages)
357
358
359 @app.route('/admin/reset_password/<int:user_id>', methods=['POST'])
360 def reset_password(user_id):
361     if 'username' not in session or session.get('role') != 'admin':
362         return redirect(url_for('login'))
363
364     db = get_db()
365
366     default_password = 'qwerty'
367     db.execute("UPDATE users SET password = ? WHERE id = ?", (default_password, user_id))
368     db.commit()
369
370     return redirect(url_for('admin'))
371
372 @app.route('/admin/delete_user/<int:user_id>', methods=['POST'])
373 def delete_user(user_id):
374     if 'username' not in session or session.get('role') != 'admin':
375         return redirect(url_for('login'))
376
```

Figure 15: **app.py** on line **366**

2.16 CWE-352 - app.py - Missing CSRF protection on password change

Priority: Medium

Issue: The state-changing POST endpoint lacks CSRF protections, allowing attackers to trigger actions via the victim's browser.

Recommendation: Implement CSRF tokens (e.g., Flask-WTF CSRFProtect) and verify them on all state-changing requests.

```
303     try:
304         with open(file_path, 'r') as file:
305             content = file.read()
306     except Exception as e:
307         content = f"Error: {str(e)}"
308 
309     return jsonify({"file_content": content})
310 
311 
312 
313 @app.route('/change_password', methods=['GET', 'POST'])
314 def change_password():
315     if 'user_id' not in session:
316         return redirect(url_for('login'))
317 
318     if request.method == 'POST':
319         user_id = session.get('user_id')
320         new_password = request.form.get('new_password')
321 
322         if user_id and new_password:
323             db = get_db()
```

Figure 16: **app.py** on line **313**

2.17 CWE-352 - app.py - Missing CSRF protection on clear messages

Priority: Low

Issue: A state-changing POST endpoint lacks CSRF protection, allowing attackers to clear chat history via the victim's browser.

Recommendation: Protect all state-changing routes with CSRF tokens and validate them.

```
139                 image_number = random.randint(1, 6)
140                 return render_template('blacklist.html', image_number=image_number)
141
142         message = re.sub(r'<img[^>]*>', '', message)
143         message = re.sub(r'alert', '', message)
144
145         chat_messages.append(message)
146
147     return render_template('chat.html', messages=chat_messages)
148
149 @app.route('/clear_messages', methods=['POST'])
150 def clear_messages():
151     # clear messages KEEP THIS!!
152     chat_messages.clear()
153     return redirect(url_for('chat'))
154
155
156
157
158
159
```

Figure 17: **app.py** on line **149**

2.18 CWE-209 - app.py - Verbose error messages returned to clients

Priority: Low

Issue: Returning raw exception messages to clients can leak internal details useful for attackers.

Recommendation: Log detailed errors server-side and return generic error messages to clients.

```
297
298     filename= request.args.get('filename', 'testing.txt')
299     file_path = os.path.join(STATIC_FOLDER, filename)
300
301
302
303     try:
304         with open(file_path, 'r') as file:
305             content = file.read()
306     except Exception as e:
307         content = f"Error: {str(e)}"
308
309     return jsonify({"file_content": content})
310
311
312
313 @app.route('/change_password', methods=['GET', 'POST'])
314 def change_password():
315     if 'user_id' not in session:
316         return redirect(url_for('login'))
317
```

Figure 18: **app.py** on line **307**