1) Target (sample vulnerable code)

(This is a short, intentionally vulnerable Flask snippet — I’ll reference lines below.)

```
from flask import Flask, request, jsonify
import sqlite3, os, hashlib, pickle
app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = '/tmp/uploads'
SECRET_KEY = "hardcoded-secret"
def get_db():
  conn = sqlite3.connect('app.db')
  return conn
@app.route('/login', methods=['POST'])
def login():
  data = request.json
  username = data.get('username')
  password = data.get('password')
  conn = get_db()
  cur = conn.cursor()
  cur.execute("SELECT id, password_hash FROM users WHERE username='%s'" %
username)
  row = cur.fetchone()
  if not row:
    return jsonify({'error': 'invalid'}), 40
  if hashlib.md5(password.encode()).hexdigest() == row[1]:
    session = {'user_id': row[0], 'role': 'user'}
    return pickle.dumps(session)
  return jsonify({'error':'invalid'}), 401
@app.route('/upload', methods=['POST'])
def upload():
  f = request.files['file']
  filename = f.filename
  path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
  f.save(path)
  return jsonify({'path': path})
@app.route('/run', methods=['POST'])
def run_command():
  cmd = request.json.get('cmd')
  os.system("echo %s > /tmp/usercmd" % cmd)
  return jsonify({'ok': True})
```

2) Security review — findings, risk, and remediation

Below each finding I include: what the issue is, why it's risky, where it appears, how to fix it

A. SQL Injection

Where: cur.execute("SELECT ... WHERE username='%s'" % username)

Why risky: User input is interpolated into SQL — attacker can run arbitrary queries (data leakage, modification, auth bypass).

Severity: High

Remediation: Use parameterized queries / prepared statements. Example:

```
cur.execute("SELECT id, password_hash FROM users WHERE username = ?", (username,))
```

Also validate username format (length, allowed chars) and limit returned columns.

B. Weak password hashing & insecure comparison

Where: hashlib.md5(password.encode()).hexdigest() and direct == comparison

Why risky: MD5 is broken for password storage (fast, no salt). Direct equality may allow timing attacks.

Severity: High

Remediation: Use a modern password hashing algorithm (bcrypt, Argon2, scrypt) with per-user salt and a well-tested library (e.g. passlib, bcrypt, or argon2-cffi). Use constant-time compare (libraries do that for you). Example:

```
from argon2 import PasswordHasher
ph = PasswordHasher()
```

C. Insecure session serialization (pickle)

Where: return pickle.dumps(session)

Why risky: pickle can execute arbitrary code when unpickling (remote code execution) — never trust pickled data from users. Also returning raw bytes as response might leak internals.

Severity: Critical

Remediation: Use secure, safe formats (JSON Web Tokens (JWT) with proper signing and short TTL, or server-side sessions stored e.g. Redis with secure session identifiers). If using Flask sessions, configure SECRET_KEY securely and use flask.session. Avoid pickle entirely for untrusted data

D. Hardcoded secret

Where: SECRET_KEY = "hardcoded-secret"

Why risky: Secrets in source are leaked via repo, backups, or logs.

Severity: High

Remediation: Use environment variables or a secrets manager (HashiCorp Vault, AWS Secrets Manager). Example: SECRET_KEY = os.environ.get('SECRET_KEY') and ensure deployment sets it securely. Rotate secrets and do not commit them.

E. Arbitrary file upload / path traversal

Where: filename = f.filename and os.path.join(..., filename) with no checks

Why risky: Attackers can upload malicious files (web shell), overwrite filesystem files via filename like ../../app.py, or upload huge files (DoS).

Severity: High

Remediation:

Validate filename: remove path components, restrict allowed extensions, sanitize names (use werkzeug.utils.secure_filename).

Enforce maximum file size (Flask MAX_CONTENT_LENGTH).

Store uploads outside of web root, and set safe permissions.

If files are executable (e.g. images), validate content type or reprocess (e.g. re-encode images).

Example:

```
from werkzeug.utils import secure_filename
filename = secure_filename(f.filename)
```

F. Command injection / unsafe system calls

Where: os.system("echo %s > /tmp/usercmd" % cmd)

Why risky: Unsanitized input used in shell command — attacker can execute arbitrary commands.

Severity: Critical

Remediation: Avoid shell invocation with string concatenation. Use safe APIs (no shell) or sanitize strictly. For example:

```
import subprocess
subprocess.run(["echo", cmd], check=True)  # avoids shell=True
```

But even then, consider whether running user-supplied commands is necessary; prefer whitelisting allowed actions.

G. Missing authentication / authorization checks

Where: upload, run, etc. routes have no auth enforcement.

Why risky: Unauthenticated users can perform privileged actions (upload files, execute commands).

Severity: High

Remediation: Enforce authentication (JWT sessions or server-side sessions) and proper role-based access control (RBAC). Validate permissions for each endpoint.

H. Poor error handling / information leakage

Where: Generic exceptions unhandled; DB errors may be returned to users.

Why risky: Stack traces reveal internal details (paths, SQL).

Severity: Medium

Remediation: Do not expose stack traces in production. Use structured error responses, centralized logging (securely), and ensure debug mode off in production. Flask: app.debug = False and configure log

I. No CSRF protection (if using cookies / sessions)

Where: No CSRF tokens; app might use session cookies.

Why risky: State-changing endpoints could be abused via CSRF.

Severity: Medium

Remediation: Use CSRF tokens for forms or set SameSite on cookies and require an authorization header for APIs (e.g., Bearer tokens) which are not sent by browsers in C

J. Insufficient input validation / rate limiting

Where: Plenty of endpoints accept arbitrary input.

Why risky: Leads to DoS, injection, oversized payloads.

Severity: Medium

Remediation: Validate JSON schema (use jsonschema or pydantic), limit request sizes, and apply rate limits (e.g., Flask-Limiter).

3) Secure code fixes (concrete examples)

Rewriting critical pieces from the sample to illustrate fixes:

```
from flask import Flask, request, jsonify, session

from werkzeug.utils import secure_filename

import os, sqlite3, bcrypt

from argon2 import PasswordHasher

import subprocess


app = Flask(__name__)

app.config['MAX_CONTENT_LENGTH'] = 4 * 1024 * 1024  # 4 MB

app.config['UPLOAD_FOLDER'] = '/srv/uploads'

app.secret_key = os.environ.get('SECRET_KEY')  # set in env

ph = PasswordHasher()

def get_db():

   conn = sqlite3.connect('app.db', isolation_level=None)  # configure as needed

   conn.row_factory = sqlite3.Row

   return conn

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

```
def login():
    data = request.get_json(force=True)
    username = data.get('username', '')
    password = data.get('password', '')
    if not username or not password:
        return jsonify({'error': 'missing creds'}), 400
    conn = get_db()
    cur = conn.cursor()
    cur.execute("SELECT id, password_hash FROM users WHERE username = ?",
(username,))
    row = cur.fetchone()
    if not row:
        return jsonify({'error': 'invalid'}), 401
stored_hash = row['password_hash']
    try:
        if ph.verify(stored_hash, password)
            session['user_id'] = row['id']
            return jsonify({'ok': True}), 200
    except Exception:
        return jsonify({'error': 'invalid'}), 40
         return jsonify({'error': 'invalid'}), 401


@app.route('/upload', methods=['POST'])
def upload():
    if 'user_id' not in session:
        return jsonify({'error': 'unauthenticated'}), 401
    f = request.files.get('file')
    if not f:
        return jsonify({'error': 'no file'}), 40
      filename = secure_filename(f.filename)
        ALLOWED = {'png','jpg','jpeg','pdf'}
    if '.' not in filename or filename.rsplit('.',1)[1].lower() not in ALLOWED:
        return jsonify({'error': 'invalid file type'}), 400
    path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
    f.save(path)
    os.chmod(path, 0o600)
    return jsonify({'path': path}), 201
@app.route('/run', methods=['POST'])
def run_command():
    if session.get('role') != 'admin':
        return jsonify({'error': 'forbidden'}), 403
    action = request.json.get('action')
    if action == 'write_temp':
        content = request.json.get('content','')
        with open('/tmp/usercmd', 'w') as fh:
            fh.write(content)
        return jsonify({'ok': True})
    else:
        return jsonify({'error': 'unsupported action'}), 400
```

4) Tools & commands for automated scans

Use a combination of static analyzers and dependency scanners.

Python static analyzers

Bandit (security-focused): pip install bandit

Run: bandit -r path/to/project

Semgrep (rule-based): pip install semgrep

Run: semgrep --config=p/security-audit path/to/project

Flake8 / MyPy for code quality/type checking.

Dependency & vulnerable scanning

pip-audit: pip install pip-audit then pip-audit

safety (pyup): pip install safety → safety check

Dependabot / Snyk for CI integration.

Secrets detection

detect-secrets (Yelp): pip install detect-secrets; run baseline and scan.

Dynamic testing

OWASP ZAP (active scanner) for API endpoints.

Burp Suite (manual pentest).

CI examples (GitHub Actions)

Run bandit, pip-audit, semgrep on push/PR and fail the job if high-severity issues found.

5) Prioritized remediation plan (practical)

1. Block immediate RCE / auth bypass (hours)

Replace pickle use; remove any eval/unpickling of user data.

Remove os.system usage with unsanitized input — disable or restrict.

Fix SQL injection by using parameterized queries.

2. Fix authentication & secrets (1–2 days)

Switch to Argon2/bcrypt for passwords and re-hash existing accounts (force password reset or re-hash on login).

Move secrets to env/secret manager and rotate.

3. File upload hardening & RBAC (1–2 days)

Enforce secure filenames, extension whitelist, content validation, max size, storage permissions.

Implement authentication on endpoints; deploy role-based checks.

4. Add automated scanning & CI (2–4 days)

Add Bandit, pip-audit, semgrep to CI; configure thresholds.

Add unit tests for auth, upload, and edge cases.

5. Operational hardening (ongoing)

Centralized logging, monitoring, WAF rules, rate limiting, secure headers (CSP, HSTS), and periodic pentests.

6) Secure coding best practices (short checklist)

Use parameterized queries for all DB access.

Use modern salted slow hashes for passwords (Argon2, bcrypt).

Never unpickle/unserialize attacker-controlled data.

Never execute shell commands with unsanitized input; prefer subprocess without shell=True.

Store secrets outside source control; use env vars or a secrets manager.

Validate and sanitize all inputs; use typed models (pydantic) and JSON schema validation.

Apply least privilege (RBAC), and verify auth on every endpoint.

Apply rate limiting and file-size limits.

Set secure cookies: Secure, HttpOnly, SameSite=Strict, and short session TTLs.

Turn off debug in production, centralize logging, and do not return stack traces.

Keep dependencies up to date; add dependency scanning to CI.

Implement automated SAST and DAST scanning in your deployment pipeline.