# Python Flask Vulnerable App - Detailed Explanation and Fixes

## 1. SQL Injection

Vulnerable Code:

name = request.args.get('name')  
query = f"SELECT \* FROM users WHERE name = '{name}'"  
result = conn.execute(query).fetchall()

Secure Fix:

query = "SELECT \* FROM users WHERE name = ?"  
result = conn.execute(query, (name,)).fetchall()

Explanation:

String concatenation with user input allows an attacker to manipulate the SQL query (e.g., ' OR '1'='1'). Use parameterized queries to prevent this.

## 2. Cross-Site Scripting (XSS)

Vulnerable Code:

return render\_template\_string(f"<h1>You searched for {q}</h1>")

Secure Fix:

return render\_template("search.html", q=q)  
  
# search.html  
<h1>You searched for {{ q }}</h1>

Explanation:

Injecting user input directly into HTML enables XSS attacks. Jinja2 auto-escapes variables in templates.

## 3. Hardcoded Credentials

Vulnerable Code:

API\_KEY = "my\_super\_secret\_key"

Secure Fix:

import os  
API\_KEY = os.environ.get('API\_KEY')

Explanation:

Credentials should be stored securely using environment variables or secret managers, not hardcoded in source code.

## 4. Insecure Deserialization

Vulnerable Code:

obj = pickle.loads(data)

Secure Fix:

import json  
obj = json.loads(data)

Explanation:

pickle can execute arbitrary code during deserialization. Use safer formats like JSON for untrusted input.

## 5. Directory Traversal

Vulnerable Code:

with open(f"./uploads/{filename}") as f:

Secure Fix:

from werkzeug.utils import secure\_filename  
safe\_filename = secure\_filename(filename)  
with open(os.path.join("./uploads", safe\_filename)) as f:

Explanation:

Unsanitized file paths can allow attackers to access arbitrary files. Use secure\_filename to sanitize input.