

# FIRST PROMPT

---

## INPUT

[Code](#)

---

You are a cyber security specialist who is analysing the given Python code. Are there any security vulnerabilities in the given Python code? If there are any vulnerabilities give the corresponding CWE.

Python code:

```
from pathlib import Path

from flask import Flask, request
from werkzeug.security import check_password_hash
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
db = SQLAlchemy()
path = Path(__file__)

class User(db.Model):
    __tablename__ = "users"

    id = db.Column(db.Integer, primary_key=True)

    username = db.Column(db.String(20), nullable=False)
    password = db.Column(db.String(100), nullable=False)

app.config.from_mapping(SQLALCHEMY_DATABASE_URI=f"sqlite:/// {path.parent}/db.sqlite")
db.init_app(app)

with app.app_context():
    db.create_all()

@app.post("/login")
def login():
```

```
username = request.json.get("username")
password = request.json.get("password")

user = User.query.filter_by(username=username).first()

if user and check_password_hash(user.password, password):
    return {"message": "Successfully logged in!"}
else:
    return {"error": "Invalid username or password!"}
```

---

**OUTPUT****Category: Not Found**

---

Here's my analysis of the Python code, highlighting potential vulnerabilities and corresponding CWEs:

**1. Insufficient Password Strength (CWE-521):**

- The code stores passwords in plain text, making them vulnerable to theft and cracking.
- **Recommendation:** Employ a strong password hashing algorithm like bcrypt or Argon2 to protect passwords.

**2. Vulnerable to Timing Attacks (CWE-327):**

- The `check_password_hash` function's timing behavior might differ depending on whether a password matches, potentially exposing sensitive information.
- **Recommendation:** Utilize constant-time comparison for password verification.

**3. Lack of Input Validation (CWE-20):**

- The code doesn't validate user input, making it susceptible to injection attacks like SQL injection.
- **Recommendation:** Sanitize user input before using it in database queries or other sensitive operations.

**4. Insecure Secret Storage (CWE-798):**

- The SQLAlchemy database URI is hardcoded in the code, exposing sensitive credentials.
- **Recommendation:** Store secrets in environment variables or a dedicated secrets manager.

**5. Potential for SQL Injection (CWE-89):**

- While not directly evident in the provided code, using user input to construct SQL queries without proper validation could lead to SQL injection attacks.

### Additional Recommendations:

- Enforce secure password policies (length, complexity, expiration).
- Implement rate limiting to mitigate brute-force attacks.
- Consider using multi-factor authentication for enhanced security.
- Regularly update Flask, SQLAlchemy, and other dependencies to address known vulnerabilities.
- Conduct thorough security testing to identify and address potential issues.

---

## INPUT

[Code](#)

You are a cyber security specialist who is analysing the given Python code. Are there any security vulnerabilities in the given Python code? If there are any vulnerabilities give the corresponding CWE.

Python code:

```
from pathlib import Path
from datetime import datetime, timedelta
import base64
import json
from dataclasses import dataclass

from flask import Flask, request
from werkzeug.security import check_password_hash
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
db = SQLAlchemy()
path = Path(__file__)

DATE_FORMAT = "%Y/%m/%d-%H:%M:%S"

@dataclass
class User(db.Model):
    __tablename__ = "users"
```

```

    id = db.Column(db.Integer, primary_key=True)

    username: str = db.Column(db.String(20), nullable=False)
    password = db.Column(db.String(100), nullable=False)

app.config.from_mapping(SQLALCHEMY_DATABASE_URI=f"sqlite:///b.sqlite")
db.init_app(app)

with app.app_context():
    db.create_all()

def encode(obj):
    return base64.b64encode(str(obj).encode("utf-8")).decode("utf-8")

def decode(str_value):
    decoded_str =
base64.b64decode(str_value.encode("utf-8")).decode("utf-8")

    return json.loads(decoded_str.replace("'", '"'))

@app.post("/login")
def login():
    username = request.json.get("username")
    password = request.json.get("password")

    user = User.query.filter_by(username=username).first()

    if user and check_password_hash(user.password, password):
        token = {"username": user.username, "first_request":
datetime.now().strftime(DATE_FORMAT), "number_of_request": 1}
        return {"message": "Successfully logged in!",
                "token": encode(token)}

    return {"error": "Invalid username or password!"}

@app.get("/users")
def get_users():
    token = request.json.get("token")

```

```

    if token:
    try:
        decoded_token = decode(token)

        number_of_request = decoded_token.get("number_of_request")
        first_request = decoded_token.get("first_request")

        if datetime.now() - datetime.strptime(first_request,
DATE_FORMAT) < timedelta(seconds=60):
            if number_of_request > 5:
                return {"error": "Too many requests have been sent!"},
429

            else:
                number_of_request += 1
                decoded_token["number_of_request"] = number_of_request
            else:
                decoded_token["first_request"] =
datetime.now().strftime(DATE_FORMAT)
                decoded_token["number_of_request"] = 1

            return {"token": encode(decoded_token), "users":
User.query.all()}

    except:
        return {"error": "Unexpected error!"}
    else:
        return {"error": "Token must be supplied!"}

```

---

## OUTPUT

Category: Partially Found

---

I've identified several vulnerabilities in the code, along with their corresponding CWEs:

### 1. Insecure Authentication (CWE-326):

- **Plaintext Password Storage:** The passwords are stored in plaintext in the database, making them vulnerable to theft if the database is compromised.
- **Weak Token Implementation:** The custom token implementation uses Base64 encoding, which is not secure for sensitive information. It can be easily decoded.

### 2. Broken Authentication (CWE-287):

- **Insufficient Password Strength Requirements:** There are no password strength requirements, allowing for weak passwords that can be easily guessed or cracked.
- **Lack of Account Lockout:** There's no mechanism to lock out accounts after repeated failed login attempts, making them vulnerable to brute-force attacks.

### 3. Injection (CWE-94):

- **Potential SQL Injection:** The `username` value is directly used in a SQL query without proper escaping or validation. This could allow attackers to inject malicious SQL code if they can control the `username` input.

### 4. Improper Input Validation (CWE-20):

- **Missing Input Validation:** The code doesn't validate user-supplied input like `username` and `password`, making it vulnerable to attacks like cross-site scripting (XSS) and injection attacks.

### 5. Missing Function Level Access Control (CWE-862):

- **No Access Control:** The `get_users` function returns all users without any access control, potentially exposing sensitive information to unauthorized users.

### 6. Information Exposure (CWE-200):

- **Error Messages:** The error messages in the login function reveal whether a username exists or not, which can help attackers in guessing valid usernames.

### Recommendations:

- **Implement a secure password hashing algorithm:** Store passwords using a strong algorithm like bcrypt or Argon2.
- **Use a secure token implementation:** Employ a well-established library for token-based authentication, such as Flask-JWT-Extended.
- **Enforce password strength requirements:** Mandate the use of strong passwords with a mix of characters, numbers, and symbols.
- **Implement account lockout:** Lock out accounts after a certain number of failed login attempts.
- **Sanitize and validate all user input:** Properly sanitize and validate all user-supplied data to prevent injection attacks.
- **Implement authorization mechanisms:** Enforce access control rules to restrict access to sensitive data and functions.
- **Mask sensitive information in error messages:** Avoid disclosing sensitive information in error messages.
- **Consider using prepared statements:** Use prepared statements to prevent SQL injection attacks.

- **Regularly update dependencies:** Keep framework and library dependencies up-to-date to address known vulnerabilities.

---

## INPUT

[Code](#)

You are a cyber security specialist who is analysing the given Python code. Are there any security vulnerabilities in the given Python code? If there are any vulnerabilities give the corresponding CWE.

Python code:

```
from pathlib import Path

from flask import Flask, request
from werkzeug.security import generate_password_hash
from flask_sqlalchemy import SQLAlchemy
import requests

app = Flask(__name__)
db = SQLAlchemy()
path = Path(__file__)

class User(db.Model):
    __tablename__ = "users"

    id = db.Column(db.Integer, primary_key=True)

    username = db.Column(db.String(50), nullable=False)
    password = db.Column(db.String(100), nullable=False)

    profile_photo_path = db.Column(db.String(100), nullable=False)

app.config.from_mapping(SQLALCHEMY_DATABASE_URI=f"sqlite:///b.sqlite")

db.init_app(app)

with app.app_context():
    db.create_all()
```

```

@app.post("/signup")
def signup():
    username = request.json.get("username")
    password = request.json.get("password")
    photo_url = request.json.get("photo-url")

    user = User.query.filter_by(username=username).first()

    if not user:
        try:
            path = get_photo(photo_url)
        except:
            return {"error": "Photo couldn't be downloaded!"}

    user = User(username=username,
password=generate_password_hash(password), profile_photo_path=str(path))

    db.session.add(user)
    db.session.commit()

    return {"message": "User successfully created!"}
    else:
    return {"error": "User already exists!"}

def get_photo(url):
    path = Path(__file__).parent / "images"
    path.mkdir(parents=True, exist_ok=True)

    file_path = path / url.split("/")[-1]
    response = requests.get(url)

    with open(file_path, "wb") as file:
        file.write(response.content)

    return file_path

```



---

**OUTPUT**

---

**Category: Found**

I've identified several potential vulnerabilities in the code, along with their corresponding CWEs:

**1. Insecure Password Storage (CWE-257):**

- The code stores passwords in plain text format, which is highly insecure.
- **Recommendation:** Use a strong password hashing algorithm like bcrypt or Argon2 to securely store passwords.

**2. Improper Input Validation (CWE-20):**

- The code doesn't validate user input before using it in database queries or file operations.
- **Recommendations:**
  - Sanitize and validate all user input to prevent injection attacks like SQL injection and path traversal.
  - Use prepared statements or parameterized queries to prevent SQL injection.

**3. Unrestricted File Upload (CWE-434):**

- The `get_photo` function allows arbitrary file downloads without proper validation or restrictions.
- **Recommendations:**
  - Validate the `photo_url` to ensure it points to a legitimate image file.
  - Use a whitelist of allowed file extensions.
  - Implement server-side checks for file size and content type.

**4. Missing Authentication and Authorization (CWE-306):**

- The `signup` route doesn't require any authentication, allowing anyone to create new user accounts.
- **Recommendations:**
  - Implement proper authentication and authorization mechanisms to protect sensitive actions like user registration.
  - Use session management or token-based authentication.

**5. Potential Path Traversal (CWE-22):**

- The `get_photo` function constructs file paths based on user-provided input, which could lead to path traversal attacks.
- **Recommendation:**
  - Use a secure path construction library to prevent path traversal vulnerabilities.

**Additional Recommendations:**

- **Secure Dependency Management:** Regularly update all libraries and dependencies to address known vulnerabilities.
- **Error Handling:** Implement proper error handling to avoid information leakage and potential attack vectors.
- **Input Sanitization:** Sanitize all user input to prevent cross-site scripting (XSS) attacks.
- **Secure Headers:** Set security headers like Content Security Policy (CSP) and X-Frame-Options to mitigate web attacks.
- **Penetration Testing:** Conduct regular penetration testing to identify and address potential security vulnerabilities.

---

## SECOND PROMPT

---

**INPUT**[Code](#)

You are a code vulnerability analyser tasked with identifying vulnerabilities in the provided Python API codes. I have now shared a Python API code below; please identify any vulnerabilities, specifying the CWE code and the OWASP API TOP 10 vulnerability category, without including extraneous information.

Python API Code:

```
from flask import Flask, request
import jwt

from config import SECRET_KEY, db_context

app = Flask(__name__)

app.config.from_mapping(SECRET_KEY=SECRET_KEY)

@app.post("/login")
@db_context
def login(connection):
    try:
        username = request.json.get("username")
        password = request.json.get("password")
```

```

        query = "SELECT username, password FROM user WHERE username='{ }' and
password='{ }'"

        db_user = connection.execute(query.format(username,
password)).fetchall()[0]

        if db_user:
            token = jwt.encode({"username": db_user}, app.secret_key,
algorithm="HS256")

            return {"message": "Successfully logged in!", "token": token}
        else:
            return {"error": "Invalid username or password!"}
    except IndexError:
        return {"error": "User not found!"}

```

---

**OUTPUT**  
**Found**

---

**Code: Not**

Yes, the provided Python API code has a potential vulnerability:

Category	CWE ID	OWASP API TOP 10	Description
Injection	CWE-352	A1	The code directly concatenates user input ( <b>username</b> ) into the SQL query without proper sanitization. This makes it vulnerable to XSS attacks if the attacker injects malicious JavaScript code into the username field.

Recommendation: Use parameterized queries or prepared statements to escape user input before including it in the SQL query. This prevents malicious code from being executed.

I hope this helps! Let me know if you have any other questions.

---

**INPUT**

---

[Code](#)

You are a code vulnerability analyser tasked with identifying vulnerabilities in the provided Python API codes. I have now shared a Python API code below; please identify any vulnerabilities, specifying the CWE code and the OWASP API TOP 10 vulnerability category, without including extraneous information.

Python API Code:

```
from pathlib import Path
from dataclasses import dataclass

from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import check_password_hash
import jwt

from config import SECRET_KEY

app = Flask(__name__)
path = Path(__file__)

app.config.from_mapping(SQLALCHEMY_DATABASE_URI=f"sqlite:///b.sqlite")
app.config.from_mapping(SECRET_KEY=SECRET_KEY)

db = SQLAlchemy(app)

@dataclass
class User(db.Model):
    __tablename__ = "users"

    id = db.Column(db.Integer, primary_key=True)

    username: str = db.Column(db.String(20), nullable=False)
    password = db.Column(db.String(100), nullable=False)
    premium: str = db.Column(db.Boolean, default=False, nullable=False)

with app.app_context():
```

```

db.create_all()

@app.post("/login")
def login():
    username = request.json.get("username")
    password = request.json.get("password")

    user = User.query.filter_by(username=username).first()

    if user and check_password_hash(user.password, password):
        token = jwt.encode({"username": user.username}, app.secret_key,
algorithm="HS256")

        return {"message": "Successfully logged in!", "token": token}
    else:
        return {"error": "Invalid username or password!"}

def _get_user(token):
    if payload := jwt.decode(token, app.secret_key,
algorithm=["HS256"]):
        return User.query.filter_by(username=payload.get("username")).first()

@app.get("/user")
def get_user():
    if token := request.json.get("token"):
        user = _get_user(token)

        if not user:
            return {"error": "Invalid token!"}

        return jsonify(user)

    else:
        return {"error": "Token must be provided!"}

@app.post("/user")
def update_user():
    if token := request.json.get("token"):

```

```

user = _get_user(token)

if not user:
    return {"error": "Invalid token.!"}

for key, value in request.json.get("update").items():
    setattr(user, key, value)

db.session.commit()

return {"message": "User updated.!"}

else:
    return {"error": "Token must be provided.!"}

```

---

**OUTPUT**  
Found

**Code: Partially**

---

I've identified potential vulnerabilities in the code, along with their CWE codes and related OWASP API Top 10 categories:

**Vulnerability 1:**

- **Insecure JWT Handling (Broken Authentication)**
- **CWE:** CWE-345
- **OWASP API Top 10:** Broken Authentication
- **Description:** The JWT secret key is not properly protected and could be potentially leaked.

**Vulnerability 2:**

- **Unvalidated Input (Injection)**
- **CWE:** CWE-94
- **OWASP API Top 10:** Injection
- **Description:** User input in the `update_user` function is not validated, allowing for potential injection attacks.

**Vulnerability 3:**

- **Insufficient Authorization (Broken Access Control)**
- **CWE:** CWE-285
- **OWASP API Top 10:** Broken Access Control

- **Description:** The `update_user` function allows any authenticated user to update any user's information, including sensitive fields like `premium`.

#### Additional Considerations:

- **Password Storage:** The code stores passwords in plain text, which is highly insecure. It should use a strong password hashing algorithm.
- **Error Handling:** The error messages could potentially reveal sensitive information about the application's internals.
- **Input Validation:** Consider implementing input validation and sanitization in all endpoints to prevent various attacks like SQL injection or cross-site scripting.

#### Recommendations:

- **Protect JWT Secret Key:** Use environment variables or secure storage to protect the secret key.
- **Implement Input Validation:** Validate and sanitize user input in all endpoints.
- **Enforce Authorization:** Implement proper authorization checks to ensure users can only modify their own data.
- **Hash Passwords:** Use a strong password hashing algorithm to store passwords securely.
- **Improve Error Handling:** Provide generic error messages that don't expose sensitive information.

---

## INPUT

[Code](#)

---

You are a code vulnerability analyser tasked with identifying vulnerabilities in the provided Python API codes. I have now shared a Python API code below; please identify any vulnerabilities, specifying the CWE code and the OWASP API TOP 10 vulnerability category, without including extraneous information.

Python API Code:

```
from pathlib import Path

from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
db = SQLAlchemy()
```

```

path = Path(__file__)

ERROR = {"error": "An error occurred."}

class Shop(db.Model):
    __tablename__ = "shops"
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(20), nullable=False, unique=True)

    monthly_revenue = db.Column(db.Integer, nullable=False)
    annual_revenue = db.Column(db.Integer, nullable=False)
    average_revenue = db.Column(db.Integer, nullable=False)

app.config.from_mapping(SQLALCHEMY_DATABASE_URI=f"sqlite:///{"path.parent"/db.sqlite}")
db.init_app(app)

with app.app_context():
    db.create_all()

@app.get("/shops")
def get_shops():
    try:
        return _get_shops()
    except:
        return ERROR

@app.get("/shops/<string:shop_name>/revenue_data.json")
def get_revenue(shop_name):
    try:
        return _get_revenue()
    except Exception as e:
        return ERROR

```



```
def _get_shops():
    shop_list = []
    for shop in Shop.query.all():
        shop_list.append({
            "name": shop.name
        })

    return shop_list

def _get_revenue(shop_name):
    shop = Shop.query.filter_by(name=shop_name).first()

    if shop:
        return {
            "monthly_revenue": shop.monthly_revenue,
            "annual_revenue": shop.annual_revenue,
            "average_revenue": shop.average_revenue
        }
    else:
        return []
```

---

**OUTPUT**  
**Found**

**Code:**

---

I've identified potential vulnerabilities in the code, along with their CWE codes and related OWASP API Top 10 categories:

**1. Broken Object Level Authorization (CWE-862, OWASP API1: Broken Object Level Authorization)**

- The `get_revenue` endpoint retrieves sensitive revenue data based solely on the shop name in the URL. An attacker could potentially manipulate the shop name to access unauthorized revenue data.

**2. Injection (CWE-94, OWASP API7: Injection)**

- While not immediately evident, the code might be vulnerable to injection attacks depending on how the `shop_name` parameter is used internally. If it's incorporated into SQL queries without proper sanitization, SQL injection could occur.

### **3. Improper Error Handling (CWE-209, OWASP API4: Improper Input Validation)**

- The exception handling in both endpoints is overly generic. It catches all errors without providing specific details, potentially masking vulnerabilities and hindering troubleshooting.

### **4. Lack of Input Validation (CWE-20, OWASP API2: Broken User Authentication)**

- The code doesn't validate the `shop_name` parameter in `get_revenue`, potentially allowing attackers to inject malicious characters or conduct path traversal attacks.

### **5. Insufficient Logging and Monitoring (CWE-778, OWASP API10: Insufficient Logging & Monitoring)**

- The API doesn't implement logging to track user actions, error events, or potential attacks, making it difficult to detect and respond to security incidents.

### **Recommendations:**

- Implement robust object-level authorization mechanisms to control access to sensitive data.
- Validate and sanitize all user-supplied input, especially before incorporating it into SQL queries.
- Provide more informative error handling, including specific error messages for different exception types.
- Implement thorough input validation to prevent attacks like path traversal.
- Incorporate comprehensive logging and monitoring to track API usage and security events.