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

```python
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**

There are several potential security vulnerabilities in the given Python code:

1. Insecure Direct Object Reference (IDOR):
The `User` class has a direct object reference to the `password` column, which can lead to an IDOR vulnerability. An attacker can potentially manipulate the `password` column directly, bypassing the check_password_hash function. To mitigate this risk, it's recommended to use a secure password hashing algorithm like Argon2 or Bcrypt.

CWE-772: Insecure Direct Object Reference (IDOR)

2. SQL Injection:
The code uses string concatenation to build the SQL query for creating the database table. This could lead to a SQL injection vulnerability if an attacker were to manipulate the `SQLALCHEMY_DATABASE_URI` configuration parameter. To mitigate this risk, it's recommended to use parameterized queries or a safe subset of SQL.

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

3. Path Traversal:
The code uses `path.parent` to build the path to the database file. This could lead to a path traversal vulnerability if an attacker were to manipulate the `path` object. To mitigate this risk, it's recommended to use a secure method for building file paths.

CWE-22: Path Traversal

4. Insecure Password Hashing:
The code uses the `check_password_hash` function from the `werkzeug.security` module to verify passwords. However, this function is not recommended for new applications, as it uses a weak password hashing algorithm (MD5). To mitigate this risk, it's recommended to use a secure password hashing algorithm like Argon2 or Bcrypt.

CWE-758: Insecure Password Hashing

5. Information Disclosure:
The code returns a plaintext error message containing the username and password entered by the user. This could lead to an information disclosure vulnerability if an attacker were to exploit this feature to obtain sensitive information. To mitigate this risk, it's recommended to return a generic error message that does not reveal sensitive information.

CWE-200: Information Disclosure

It's essential to address these security vulnerabilities promptly to protect the application and its users from potential attacks.

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:

```python
from pathlib import Path
from dataclasses import dataclass
from flask import Flask, request
from flask_sqlalchemy import SQLAlchemy
import hashlib

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

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

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)


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


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

    user = User(username=username, password=hashlib.md5(password))
    db.session.add(user)
    db.session.commit()

    return {"message": "User successfully created.!"}
    except Exception as e:
    print(e)
    return {"error": "User couldn't be created.!"}
```

OUTPUT **Category: Partially Found**

There are several potential security vulnerabilities in the given Python code:

1. Insecure Direct Object Reference (IDOR):
The `User` class has a direct object reference to the `db.Model` class, which can lead to potential security vulnerabilities. An attacker can potentially manipulate the `User` class and gain unauthorized access to the database. CWE-915: Insecure Direct Object Reference (IDOR).
2. SQL Injection:
The code uses string concatenation to create the SQL query for creating the user table. This can lead to SQL injection vulnerabilities if an attacker can inject malicious input into the query. CWE-895: Improper Neutralization of Special Elements in Output.
3. Password Hashing:
The code uses the `hashlib.md5()` function to hash the user's password. This is not a secure way to hash passwords, as it is vulnerable to collision attacks. A better approach would be to use a password hashing library such as `passlib` or ` cryptography`. CWE-758: Weak Password Hashing.
4. Information Disclosure:
The code returns a message to the user indicating whether the user was created successfully or not. This can potentially leak information about the user's existence, which may be sensitive. CWE-200: Information Disclosure.
5. Lack of Input Validation:
The code does not validate the input data properly, which can lead to potential security vulnerabilities. For example, the `username` and `password` fields are not validated, which can lead to SQL injection or buffer overflow attacks. CWE-113: Insufficient Logging and Monitoring.
6. Lack of Error Handling:
The code does not handle errors properly, which can lead to potential security vulnerabilities. For example, if an attacker injects malicious input, the code may not handle the error properly, leading to unexpected behavior. CWE-110: Improper Error Handling.

To fix these vulnerabilities, the code should be refactored to use secure coding practices. For example, the code should use parameterized queries to prevent SQL injection, and a secure password hashing library should be used to store the user's password securely. Additionally, the code should validate input data properly and handle errors securely.

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:

```python
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 occured."}


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 **Category: Found**

As a cyber security specialist, I have analyzed the given Python code and found a few potential security vulnerabilities. Here are my findings, along with the corresponding CWE (Common Weakness Enumeration) identifiers:

1. SQL Injection (CWE-89):
The code uses raw user input (shop_name) in a SQL query without proper sanitization or escaping. This could allow an attacker to inject malicious SQL code, potentially leading to unauthorized access or manipulation of data.
2. Path Traversal (CWE-22):
The code uses `pathlib.Path(__file__)` to get the current file path, which could potentially be manipulated by an attacker to access sensitive information or execute arbitrary code.
3. Information Disclosure (CWE-200):
The code returns sensitive information (shop revenue data) without proper authorization or access control. This could allow an unauthorized user to access sensitive data.
4. Insecure Direct Object Reference (IDOR):
The code uses `Shop.query.filter_by(name=shop_name).first()` to retrieve a shop object by name. This could potentially allow an attacker to manipulate the shop object or access sensitive information by manipulating the `name` parameter.
5. Insecure deserialization (CWE-502):
The code uses `json.loads()` to deserialize a JSON response. This could potentially allow an attacker to exploit a deserialization vulnerability to execute arbitrary code.

It's important to note that these are just potential vulnerabilities, and a thorough security assessment should be conducted to identify any actual vulnerabilities and recommend appropriate mitigations.

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:

```python
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 **Code: Not Found**

There are several potential vulnerabilities in the given Python API code:

1. Insecure Direct Object Reference (IDOR):

The `User` class has a direct object reference to the `path` variable, which could potentially allow an attacker to access sensitive information or perform unauthorized actions. This vulnerability is related to the OWASP API TOP 10 vulnerability category: A3:2017 - Sensitive Data Exposure.

2. Insecure Password Hashing:

The `check_password_hash()` function is used to verify the password, but it's not sufficient to prevent rainbow table attacks. A stronger hashing algorithm such as Argon2 or Bcrypt should be used instead. This vulnerability is related to the OWASP API TOP 10 vulnerability category: A5:2017 - Broken Authentication.

3. SQL Injection:

The code uses raw SQL queries and string concatenation, which makes it vulnerable to SQL injection attacks. An attacker could potentially inject malicious SQL code to extract sensitive data or perform unauthorized actions. This vulnerability is related to the OWASP API TOP 10 vulnerability category: A1:2017 - Injection.

4. Lack of Input Validation:

The code doesn't validate the input properly, which could lead to potential security vulnerabilities. For example, the `username` and `password` parameters in the `@app.post("/login")` function are not validated, which could allow an attacker to inject malicious data. This vulnerability is related to the OWASP API TOP 10 vulnerability category: A2:2017 - Broken Authentication.

Therefore, the CWE code for this vulnerability would be:

CWE-200: Information Exposure

CWE-307: Insecure Password Hashing

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

CWE-20: Improper Input Validation

The related OWASP API TOP 10 vulnerability category would be:

A3:2017 - Sensitive Data Exposure

A5:2017 - Broken Authentication

A1:2017 - Injection

A2:2017 - Broken Authentication

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, request
from werkzeug.security import check_password_hash
from flask_sqlalchemy import SQLAlchemy
import jwt

from config import SECRET_KEY


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)
      email = db.Column(db.String(50), nullable=False)


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

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

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


@app.post("/change-email")
def change_email():
      token = request.json.get("token")
      email = request.json.get("email")

      if token:
      try:
            payload = jwt.decode(token, app.secret_key,
algorithms=["HS256"])
      except:
            return {"error": "Please provide a valid token.!"}
      else:
      return {"error": "Token must be provided.!"}

      user = User.query.filter_by(username=payload["username"]).first()

      if user:
      user.email = email

      db.session.commit()

      return {"message": "Successfully changed email.!"}

      return {"Invalid username.!"}
```

OUTPUT **Code: Partially Found**

I have identified several potential vulnerabilities in the given Python API code. Here are the vulnerabilities and their corresponding CWE codes and OWASP API TOP 10 vulnerabilities categories:

1. Insecure Direct Object Reference (IDOR):

The `User` class has a direct object reference to the `db` object, which can lead to unauthorized access to the database. An attacker can potentially manipulate the `db` object to execute arbitrary SQL queries, leading to a variety of attacks, such as data tampering or elevation of privileges.

CWE-73: Insecure Direct Object Reference

OWASP API TOP 10-A3:2017

1. Insecure Password Hashing:The `check_password_hash()` function is used to verify the password, but it is not sufficient to prevent rainbow table attacks. The password hash should be salted and stretched to slow down the hashing process and increase the computational overhead.

CWE-522: Insecure Password Hashing

OWASP API TOP 10-A3:2017

1. Insufficient Logging and Monitoring:The code does not have sufficient logging and monitoring mechanisms in place. Logging and monitoring are essential for detecting and responding to security incidents.

CWE-261: Insufficient Logging and Monitoring

OWASP API TOP 10-A5:20171. Broken Authentication:The `login()` function uses a plain JSON Web Token (JWT) to authenticate users. JWTs are vulnerable to tampering and replay attacks. A secure authentication mechanism should use a secure token, such as a JSON Web Token signed with a secret key, and implement proper token validation and revocation mechanisms.

CWE-614: Broken Authentication

OWASP API TOP 10-A3:2017

1. Insecure Direct Access to Backend Resources:The `change_email()` function allows users to change their email address without proper authorization. An attacker can potentially manipulate the email address of a user, leading to a variety of attacks, such as phishing or identity theft.

CWE-603: Insecure Direct Access to Backend Resources

OWASP API TOP 10-A3:2017

It is essential to address these vulnerabilities to protect the API from potential attacks and ensure the security and privacy of users' data.

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 occured."}


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 Code: Found

I have identified several potential vulnerabilities in the given Python API code. Here are the vulnerabilities and their corresponding CWE codes and OWASP API TOP 10 vulnerabilities categories:

1. SQL Injection (CWE-89) - OWASP API TOP 10: A3:2017

The code uses raw SQL queries and string concatenation, which can lead to SQL injection vulnerabilities. An attacker can exploit this by injecting malicious SQL code to extract sensitive data or escalate privileges.

2. Path Traversal (CWE-22) - OWASP API TOP 10: A4:2017

The code uses `pathlib.Path(__file__)` to construct a path to the database file. However, this can be exploited by an attacker to traverse the file system, potentially leading to unauthorized access to sensitive files.

3. Sensitive Data Exposure (CWE-538) - OWASP API TOP 10: A5:2017

The code stores sensitive data, such as monthly and annual revenue, in plaintext in the database. An attacker who gains access to the database can extract this sensitive information, potentially leading to financial loss or reputational damage.

4. Broken Authentication (CWE-522) - OWASP API TOP 10: A7:2017

The code uses Flask-SQLAlchemy for authentication, but it does not implement proper access control mechanisms. An attacker can potentially exploit this by gaining unauthorized access to sensitive data or perform actions as an authenticated user.

5. Cross-Site Scripting (CWE-79) - OWASP API TOP 10: A8:2017

The code uses raw user input in the `shop_name` parameter of the `get_revenue` function without proper validation or encoding. An attacker can exploit this by injecting malicious JavaScript code, potentially leading to a cross-site scripting (XSS) attack.

6. Insecure Direct Object Reference (CWE-91) - OWASP API TOP 10: A6:2017

The code uses direct object references (IDOR) to access data in the database. An attacker can potentially exploit this by manipulating the IDOR to access sensitive data or perform actions as an authenticated user.

7. Missing Function Level Access Control (CWE-276) - OWASP API TOP 10: A7:2017

The code does not implement proper access control mechanisms for functions. An attacker can potentially exploit this by accessing sensitive data or performing actions as an authenticated user.

8. Unnecessary Data Exposure (CWE-538) - OWASP API TOP 10: A5:2017

The code returns sensitive data, such as monthly and annual revenue, in the response of the `get_revenue` function. An attacker can potentially extract this sensitive information, potentially leading to financial loss or reputational damage.

It's important to note that these are just potential vulnerabilities, and a comprehensive security assessment should be conducted to identify and prioritize actual vulnerabilities. Additionally, it's important to follow best practices for secure coding and security testing to mitigate these vulnerabilities.