

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
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

Category: Not Found

The provided Python code appears to be a Flask web application with SQLAlchemy integration to interact with a SQLite database. Overall, the code structure seems well-organized and does not show any blatant security vulnerabilities. However, I'll point out a few areas to consider for potential improvements:

1. Error Handling:

In the `get_shops` and `get_revenue` functions, generic error messages are returned without specifying the actual error that occurred. This lack of detailed error handling might make it difficult to troubleshoot issues and could potentially expose sensitive information. It's recommended to log detailed error messages during development and return user-friendly error messages without exposing implementation details in a production environment.

2. Exception Handling:

In the `get_shops` and `get_revenue` functions, there is a blanket `except` clause capturing all exceptions. It's better to catch specific exceptions to handle different error scenarios more appropriately. This would make it easier to identify and address potential issues.

3. SQL Injection:

The code uses SQLAlchemy, which helps protect against SQL injection by parameterizing queries. However, it's important to ensure that all user inputs are properly validated and sanitized. In the provided code, the `get_revenue` function takes `shop_name` directly from the URL parameters, but it is used in the query without validation. While Flask and SQLAlchemy provide some protection, it's good practice to validate and sanitize user inputs explicitly.

4. Sensitive Information Exposure:

The code returns a generic error message in the event of an exception. While this is good for security, it might not be very helpful for developers during debugging. Consider logging more detailed error messages during development and providing generic messages to users in production.

5. Access Control:

The code does not implement access control mechanisms. Depending on the application's requirements, you might want to add authentication and authorization to restrict access to certain routes or resources.

6. Security Headers:

Consider adding security headers to the Flask application to enhance security. For example, you could include headers like Content Security Policy (CSP), Strict-Transport-Security (HSTS), and others to mitigate certain types of attacks.

As for Common Weakness Enumeration (CWE) identifiers, the mentioned issues are more about security best practices rather than specific vulnerabilities. The recommendations provided above should help improve the security posture of the application.

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

from config import SECRET_KEY

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

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

```

app.config.from_mapping(SECRET_KEY=SECRET_KEY)

db = SQLAlchemy(app)

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)

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!"}

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

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

```

```

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

if token:
    try:
        if jwt.decode(token, app.secret_key,
algorithms=["HS256"]):
            user =
User.query.filter_by(username=username).first()

            if user:
                user.password = generate_password_hash(password)
                db.session.commit()

                return {"message": "Password has been changed.!"}
            else:
                return {"message": "User couldn't found.!"}
        else:
            raise Exception("Invalid token.!")
    except:
        return {"error": "Error occurred.!"}
    else:
        return {"error": "Token must be supplied.!"}

```

OUTPUT

Category: Partially Found

The provided Python code appears to be a Flask application that implements user authentication and password change functionality using Flask-SQLAlchemy for database operations and JWT (JSON Web Tokens) for authentication. Overall, the code structure seems sound, but there are a couple of security considerations and potential improvements:

1. Insecure Password Storage:

- The code uses `generate_password_hash` for storing passwords securely, which is good. However, it's essential to ensure that strong hashing algorithms are used. The code uses the default algorithm, which is generally secure, but you may want to explicitly specify it to be sure.
- CWE: [CWE-759](#) - Use of a One-Way Hash with a Predictable Salt

2. Token Handling:

- The code uses JWT for authentication, which is a good practice. However, there is a potential issue in the `change_password` function. The JWT token is verified using `jwt.decode`, but the result is not used. A valid token only ensures that the token is well-formed and hasn't expired. It doesn't necessarily mean the user is authenticated.
- CWE: [CWE-285](#) - Improper Authorization

3. Error Handling:

- The generic `except` block in the `change_password` function catches all exceptions, making it challenging to identify specific issues. It would be better to catch only the specific exceptions you expect (e.g., `jwt.ExpiredSignatureError`, `jwt.InvalidTokenError`).
- CWE: [CWE-703](#) - Improper Check or Handling of Exceptional Conditions

4. Security of Secret Key:

- The `SECRET_KEY` is loaded from an external configuration file (`config.py`). Ensure that this file is adequately protected and not exposed to unauthorized users.
- CWE: [CWE-798](#) - Use of Hard-coded Credentials

5. Logging:

- Ensure that sensitive information, such as passwords or detailed error messages, is not logged to prevent potential information disclosure.
- CWE: [CWE-532](#) - Insertion of Sensitive Information into Log File

6. HTTPS:

- If the application is deployed in a production environment, consider using HTTPS to encrypt communication between the client and server.
- CWE: [CWE-319](#) - Cleartext Transmission of Sensitive Information

It's important to note that security is a complex and evolving field, and the suggestions provided here are based on common best practices at the time of writing. It's recommended to stay updated on security best practices and consider using security tools and libraries for additional layers of protection.

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 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: Found**

Yes, there are security vulnerabilities in the given Python code:

1. Weak Password Hashing:

The code uses the `hashlib.md5` function to hash passwords, which is considered a weak and insecure hashing algorithm. MD5 is vulnerable to collision attacks, and its hash values can be easily cracked. It's recommended to use stronger and more secure hashing algorithms such as SHA-256 or bcrypt.

Common Weakness Enumeration (CWE): CWE-916 (Use of Weak Hash Function for Password)

2. Exception Handling:

The code uses a generic `except Exception as e` statement, which can be risky. Catching all exceptions might hide potential issues or security-related errors. It is better to catch specific exceptions and handle them accordingly.

Common Weakness Enumeration (CWE): CWE-248 (Uncaught Exception)

3. Missing Salt for Password Hashing:

The code does not use any salt when hashing passwords. Salting is an important practice to add randomness to each password hash, making it more difficult for attackers to use precomputed tables (rainbow tables) for cracking passwords.

Common Weakness Enumeration (CWE): CWE-759 (Use of a One-Way Hash without a Salt)

To improve the code's security:

- Use a stronger password hashing algorithm like SHA-256 or bcrypt.
- Implement proper exception handling by catching specific exceptions.
- Include a unique and random salt for each password when hashing.

Remember that for actual production code, it's crucial to follow best practices for securing user credentials and sensitive data.

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 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: Not

The provided Python API code appears to be well-written and doesn't contain any obvious vulnerabilities. However, it's important to note that the code provided is incomplete, as some of the functions like `_get_revenue` are missing their actual implementation details. Without the complete code, it's challenging to identify specific vulnerabilities.

As of my last knowledge update in January 2022, based on the provided code, it doesn't exhibit any common vulnerabilities that correspond to specific CWE codes or OWASP API TOP 10 categories.

It's worth noting that code security is a complex topic, and vulnerabilities may arise from various factors, including input validation, authentication, authorization, and data handling. It's recommended to perform a comprehensive security analysis, including code reviews, static analysis, and dynamic testing, to identify potential vulnerabilities in a complete codebase.

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
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
Found

Code: Partially

Yes, there is a vulnerability in the provided Python API code. The vulnerability is related to weak encryption in the password storage.

1. **CWE Code:** CWE-916 (Use of Password Hash With Insufficient Computational Effort)
2. **OWASP API TOP 10 Category:** A2 - Broken Authentication (Specifically, "A2:2017 - Broken Authentication: Using Components with Known Vulnerabilities" as per the CWE code)

Explanation:

- The code uses the `hashlib.md5` function to hash passwords, which is considered a weak hashing algorithm. MD5 is vulnerable to various attacks, and it's not suitable for securely hashing passwords.
- CWE-916 is associated with using a weak password-hashing algorithm that does not provide sufficient computational effort, making it easier for attackers to brute

force or use other attacks to recover passwords.

- The related OWASP API TOP 10 category is “A2 - Broken Authentication,” specifically addressing the use of components with known vulnerabilities in authentication mechanisms. In this case, the weak encryption (MD5) for password storage represents a vulnerability in the authentication process.

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
import os

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

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=generate_password_hash(password))

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

        os.system(f"echo 'User with username {username} has been
created.' >> {path.parent}/log.txt")

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

```

OUTPUT
Found
Code:

The provided Python API code has a potential security vulnerability related to Command Injection.

CWE Code: CWE-78 (Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'))

OWASP API TOP 10 Category: A1: Injection

Vulnerability Description:

The vulnerability lies in the following line of code:

```
os.system(f"echo 'User with username {username} has been created.' >> {path.parent}/log.txt")
```

The `os.system` function is susceptible to command injection if the `username` parameter is not properly sanitized. An attacker could potentially manipulate the `username` input to execute arbitrary commands, leading to unauthorized access or unintended consequences.

To mitigate this vulnerability, it is recommended to use proper input validation and avoid incorporating user input directly into system commands. Preferably, use logging mechanisms provided by the framework or libraries to achieve the desired functionality without introducing security risks.