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
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 **Category: Not Found**

The provided Python code is a simple Flask application with SQLAlchemy for database management, JWT for authentication, and password hashing. Let’s analyze it for potential security vulnerabilities:

1. **SQL Injection (CWE-89):** The code uses SQLAlchemy ORM for database queries which, by default, uses parameterized queries. This generally mitigates the risk of SQL injection. However, it’s still important to validate or sanitize inputs, especially if raw SQL is used elsewhere in the application.

2. **Insecure Direct Object References (IDOR) / Broken Access Control (CWE-284):** The `/change-email` endpoint decodes the JWT token and trusts the `username` within it to identify the user. This could be a vulnerability if the JWT token is not validated properly or if the JWT implementation is flawed. An attacker might manipulate the token to change the email of another user.

3. **Use of Hard-coded Credentials (CWE-798):** The `SECRET_KEY` is imported from a configuration file. This is fine as long as the key is kept secret and is strong enough. However, if the key is weak, predictable, or hardcoded in a shared codebase, it could lead to security issues.

4. **Insufficient Logging and Monitoring (CWE-778):** The code lacks logging, especially for failed login attempts or other potentially malicious activities. This could make it difficult to detect or investigate malicious activities.

5. **Exposure of Sensitive Information to an Unauthorized Actor (CWE-200):** While not directly evident from the given code, if any sensitive information (like the full error messages or stack traces) is exposed to the users, it could lead to

information leakage.

6. **JWT Implementation Details:** It's crucial to ensure that the JWT implementation is secure. This includes proper validation of tokens, handling token expiration, and ensuring the tokens are stored securely on the client side.

7. **Password Storage (CWE-916):** The application uses `check_password_hash` for password validation, which suggests that passwords are hashed. It's important to ensure that the hashing is implemented securely (using a strong hash function and salt).

8. **Cross-Site Request Forgery (CSRF) (CWE-352):** Flask does not protect against CSRF by default. If the application is used in a browser context, CSRF protection should be added.

9. **Error Handling:** The code does not properly handle exceptions that could occur during database operations or JWT encoding/decoding, which might lead to unintended behaviors.

10. **Cross-Origin Resource Sharing (CORS):** If the API is intended to be accessed from web pages not served by the same domain, CORS policies need to be correctly implemented to avoid security risks.

11. **Sensitive Data Exposure through Caching (CWE-524):** If HTTP caching headers are not set properly, sensitive data could be cached by browsers or intermediate proxies.

In summary, while there are no glaring SQL injection vulnerabilities, there are several areas, such as JWT handling, error handling, and CSRF protection, where the application's security could potentially be improved.

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 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: Partially Found**

The provided Python code, which uses Flask and SQLAlchemy, seems mostly well-structured, but there are a couple of potential security concerns to address:

1. **Insecure Database URI in Source Code (CWE-259: Use of Hard-coded Password)**: The line `app.config.from_mapping(SQLALCHEMY_DATABASE_URI=f"sqlite:///{path.parent}/db.sqlite")` suggests that the database URI is hardcoded in the source code. This can be a security issue, especially if the code is stored in a v<ersion control system that is accessible to unauthorized users. In a production environment, it's recommended to use environment variables or a configuration file that is not included in the source code repository to store sensitive information like database URIs.

2. **Insufficient Error Handling (CWE-209: Information Exposure Through an Error Message)**: The exception handling in `get_shops` and `get_revenue` functions is very generic and does not log the errors. While it's good that it doesn't expose detailed error information to the end user, this could also mean that important information about exceptions is not logged, making it difficult to diagnose issues. Additionally, using a generic catch-all except statement can mask other unexpected issues, which might be security-related.

3. **Potential Insecure Direct Object References (CWE-932: OWASP Top Ten 2021 Category A05 - Security Misconfiguration)**: The endpoint `/shops/<string:shop_name>/revenue_data.json` directly uses the `shop_name` parameter from the URL to query the database. If there are no additional checks or controls, this could potentially be exploited to access data that the user should not have access to. Implementing proper access control or validation checks can mitigate this risk.

4. **Lack of Input Validation (CWE-20: Improper Input Validation)**: There is no evident input validation for the `shop_name` parameter in the `get_revenue` function. Malformed or malicious inputs could potentially lead to issues like SQL injection, although SQLAlchemy does mitigate this risk to an extent by using parameterized queries.

5. **Database Schema Exposure (CWE-200: Information Exposure)**: The structure of the database (such as table names and columns) is defined within the code. If this code is exposed, it could give potential attackers insights into the database structure, which could be used in SQL injection attacks. While this is a

lower risk because SQLAlchemy is being used, it’s still a consideration for overall security posture.

Remember, addressing these vulnerabilities involves not just modifying the code, but also implementing best practices in configuration management, access control, and error handling/logging.

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: Found**

The Python code provided is a simple Flask application with SQLAlchemy for database interactions, specifically for a user login system. Let’s analyze the potential security vulnerabilities:

1. **SQL Injection (CWE-89)**: The use of SQLAlchemy ORM (Object-Relational Mapping) in querying the database mitigates the risk of SQL injection, as it handles parameterized queries internally. However, if there are other parts of the code (not shown) that manually construct SQL queries without proper sanitation, there might be a risk of SQL injection.

2. **Storing Passwords in Plain Text (CWE-256)**: The code uses `werkzeug.security.check_password_hash`, which implies that the passwords are expected to be stored as hashed values. This is a good practice. However, if the actual password hashing is not implemented correctly elsewhere in the code (for example, using weak hashing algorithms or not using salts), it could lead to vulnerability. As provided, there’s no direct indication that passwords are stored insecurely, but this aspect needs careful review.

3. **Insufficient Logging and Monitoring (CWE-778)**: The code does not seem to have any logging mechanism for successful or failed authentication attempts. Without proper logging, it’s difficult to track potential malicious activities, like brute-force attacks.

4. **Lack of Rate Limiting (CWE-307)**: There is no implementation of rate limiting on the login endpoint. This can allow an attacker to perform brute force attacks on the login endpoint.

5. **Sensitive Data Exposure (CWE-200)**: The error message `"Invalid username or password.!"` could potentially expose user information. For example, if the message was different depending on whether the username exists or not, it could allow an attacker to enumerate valid usernames.

6. **Insecure Configuration (CWE-16)**: The `app.config.from_mapping` line sets the database URI. Depending on the deployment environment, if the database is not properly secured or if this configuration is not managed securely, it could lead to unauthorized access to the database.

7. **Lack of Input Validation (CWE-20)**: There's no visible input validation on the `username` and `password` fields. This could potentially lead to various issues, including but not limited to, script injection or other forms of input-based attacks if these inputs are used insecurely elsewhere in the application.

8. **Hardcoded Secret (CWE-798)**: There's no indication of a hardcoded secret in the provided code snippet, but this is a common issue in applications. It's important to ensure that sensitive information like database URIs, API keys, etc., are not hardcoded and are securely managed.

In summary, while the code snippet shows basic security measures like password hashing, there are areas like logging, rate limiting, and input validation that could be improved to enhance security. Further review of the complete codebase is necessary to ensure comprehensive security.

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, 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**

The provided Python API code, which uses Flask and SQLAlchemy, demonstrates a vulnerability related to the handling of user authentication. Specifically, it concerns the potential for timing attacks.

Vulnerability Details:

- **Vulnerability Type:** Timing Attack
- **CWE ID:** CWE-208 (Time of Check Time of Use (TOCTOU) Race Condition)
- **Related OWASP API TOP 10 Category:** API2:2019 — Broken User Authentication

Explanation:

- **Timing Attack:** The code uses `check_password_hash` for password verification. However, it does not implement constant-time comparison. This can lead to a timing attack, where an attacker measures the time taken to validate user credentials. Small time differences can be used to infer correct vs incorrect password characters.
- **OWASP Category - Broken User Authentication:** This vulnerability falls under the OWASP category of Broken User Authentication, as it affects the security mechanisms implemented to manage authentication and session management.

It's important to note that while this code snippet does demonstrate a potential vulnerability, the real-world impact would depend on various factors including network conditions and the server's performance. To mitigate such vulnerabilities, it's advised to use constant-time comparison functions for password checks and other sensitive operations.

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**

The Python API code you provided exhibits a vulnerability related to **Insecure Direct Object References (IDOR)**, which falls under the category of **Broken Object Level Authorization (BOLA)** as per the **OWASP API Top 10** list. Specifically, the issue is with the `/change-email` endpoint.

Here's a breakdown of the vulnerability:

1. **Endpoint:** `/change-email`
2. **Issue:** The code changes the email address based on the username obtained from the JWT token, without verifying if the requester is authorized to change that specific user's email. There is no check to ensure that the token belongs to the user whose email is being changed.
3. **Consequence:** An authenticated user can change the email address of any user, not just their own, by manipulating the JWT token.
4. **CWE (Common Weakness Enumeration):** This vulnerability aligns with CWE-285: Improper Authorization.
5. **OWASP API Top 10 Category:** This falls under A1:2019 - Broken Object Level Authorization.

To mitigate this vulnerability, the application should implement proper authorization checks to ensure that the authenticated user (the one whose identity is in the token) is the same as the user whose email is being changed.

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

The provided Python API code contains a vulnerability related to improper input validation. This is identified as:

- **CWE Code**: The vulnerability can be classified under CWE-20: Improper Input Validation.

- **OWASP API Top 10 Category**: It falls under the category ‘A01:2021 - Broken Object Level Authorization’ in the OWASP API Top 10.

This vulnerability is present in the `/shops/<string:shop_name>/revenue_data.json` endpoint. The function `get_revenue(shop_name)` takes a `shop_name` as input directly from the URL and uses it in a query without sufficient validation or encoding. This could potentially allow an attacker to manipulate the query, leading to issues like SQL Injection if the underlying database interaction were not handled safely (although in this code SQLAlchemy is used, which mitigates typical SQL Injection risks). However, it still poses a risk of unauthorized data access or manipulation at the object level due to insufficient authorization checks.