Task 13: Secure API Testing & Authorization Validation

Introduction

In this task, I learned how REST APIs work and how security testing is performed on APIs. I created my own API using Flask in Ubuntu and tested it using Postman. The main goal of this task was to test authentication, authorization, input validation, and response handling. I also checked how APIs behave when valid and invalid requests are sent. This helped me understand common API security misconfigurations and OWASP API risks.

Tools Used

Primary Tool:

- Postman

Alternative Tools (Studied but not used):

- cURL
- Insomnia

Other Technologies Used:

- Python Flask (To create test API)
- Ubuntu Terminal (To run API server)

1.Understanding REST API and HTTP Methods

REST APIs allow applications to communicate with backend servers using HTTP methods.

The main HTTP methods are:

GET → Used to fetch data
POST → Used to create data
PUT → Used to update data
DELETE → Used to delete data

In my testing, I mainly used GET method to retrieve user data from my API.

2.How I Setup My Own API

First, I opened Ubuntu terminal. Then I created a project folder to store my API files.

I created a virtual environment because Ubuntu restricts direct pip installation. After activating the virtual environment, I installed Flask. Then I created a Python file and wrote API code. After that, I ran the API server locally and tested it using Postman.

Steps I Performed to Setup API

Step 1 — Create Project Folder

I created a folder for API project and moved into it.

Step 2 — Create Virtual Environment

I created virtual environment to safely install Python packages.

Step 3 — Activate Virtual Environment

I activated the virtual environment so Flask installs locally.

Step 4 — Install Flask

I installed Flask using pip.

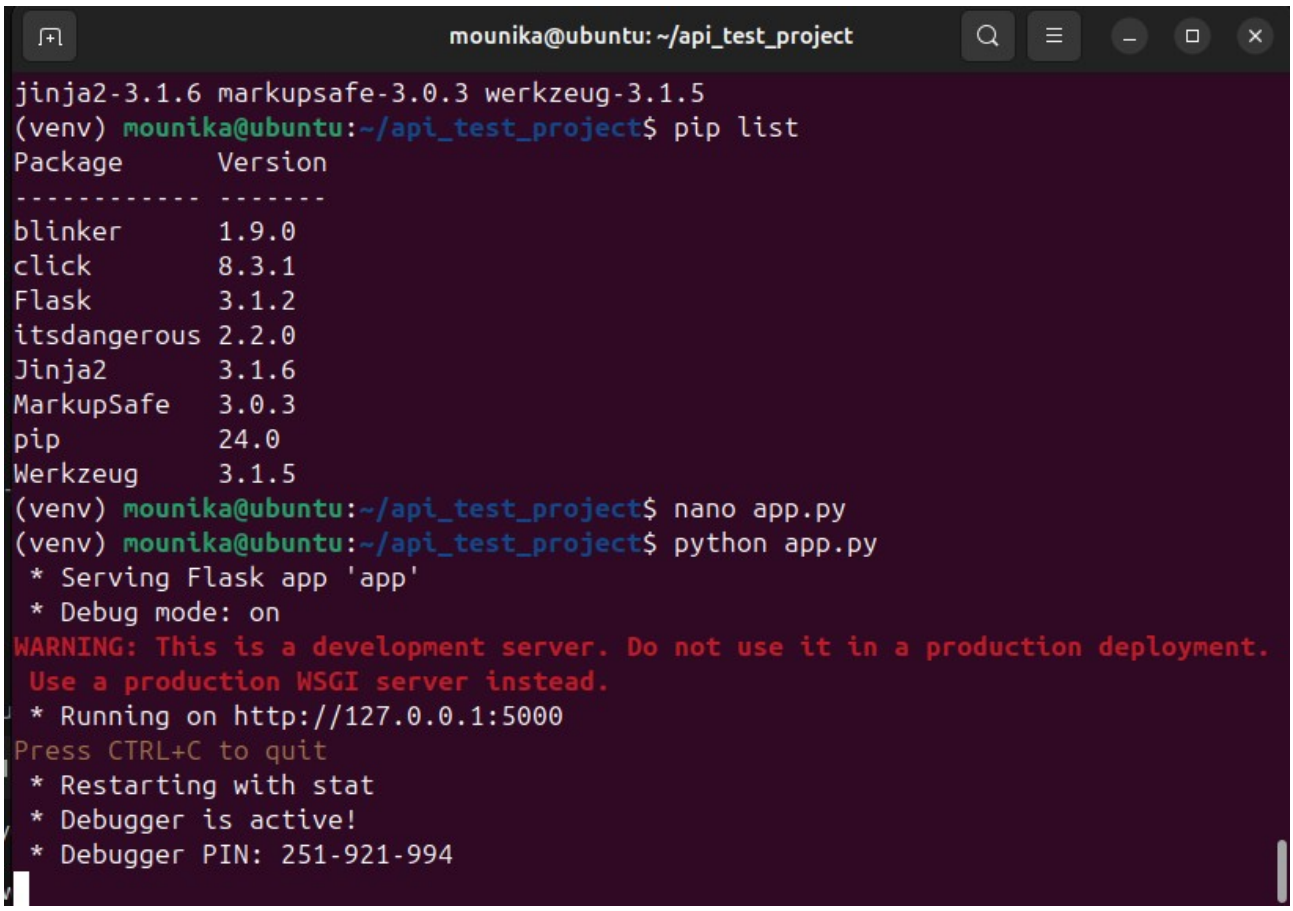
Step 5 — Create API File

I created app.py file and wrote API code.

Step 6 — Run API

I ran API using python command and verified it was running on localhost.

Screenshot — API Running in Terminal :


mounika@ubuntu: ~/api_test_project
jinja2-3.1.6 markupsafe-3.0.3 werkzeug-3.1.5
(venv) mounika@ubuntu:~/api_test_project$ pip list
Package Version
blinker 1.9.0
click 8.3.1
Flask 3.1.2
itsdangerous 2.2.0
Jinja2 3.1.6
MarkupSafe 3.0.3
pip 24.0
Werkzeug 3.1.5
(venv) mounika@ubuntu:~/api_test_project$ nano app.py
(venv) mounika@ubuntu:~/api_test_project$ python app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 251-921-994

API Code I Wrote

app.py > ...

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

users = {
    1: {"name": "Alice"},
    2: {"name": "Bob"}
}

VALID_TOKEN = "securetoken123"

@app.route("/users/<int:user_id>", methods=["GET"])
def get_user(user_id):
    auth_header = request.headers.get("Authorization")

    if not auth_header:
        return jsonify({"error": "Missing token"}), 401

    token = auth_header.replace("Bearer ", "")

    if token != VALID_TOKEN:
        return jsonify({"error": "Invalid token"}), 403

    if user_id in users:
        return jsonify(users[user_id])

    return jsonify({"error": "User not found"}), 404


if __name__ == "__main__":
    app.run(debug=True)
```

API Testing Using Postman

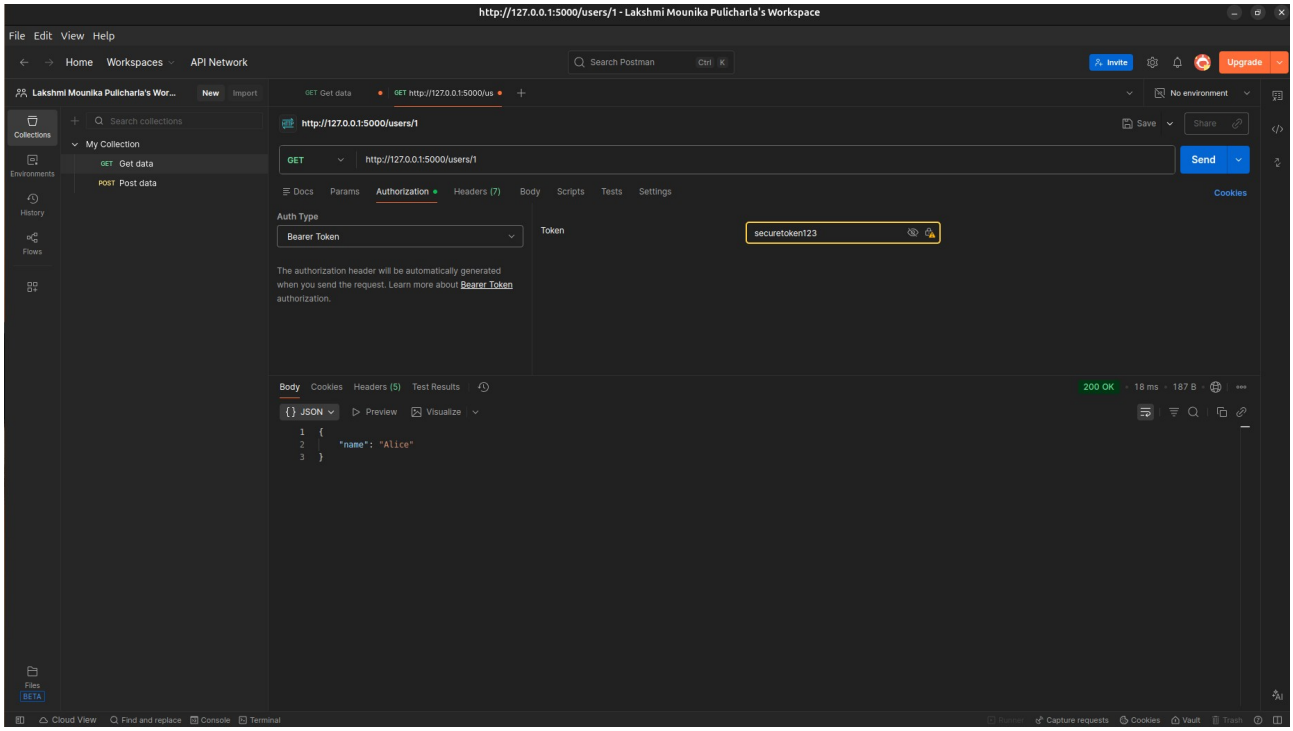
After running the API locally, I opened Postman and configured requests using localhost URL.

3.Authentication Testing

Valid Credentials Testing

I sent request using valid bearer token. API returned status 200 and returned user data. This shows authentication works correctly.

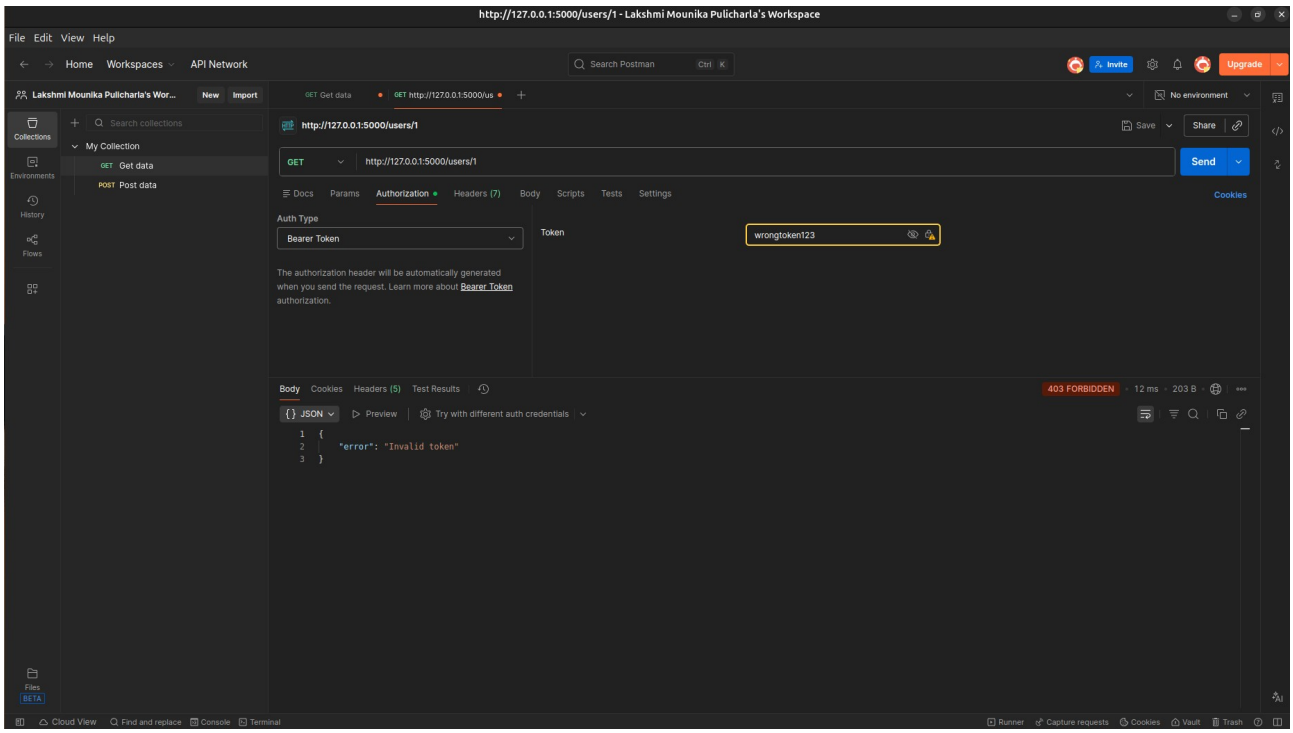
Screenshot — Valid Token Test



Invalid Credentials Testing

I changed token value and sent request. API returned status 403 Forbidden. This shows API correctly blocks invalid authentication.

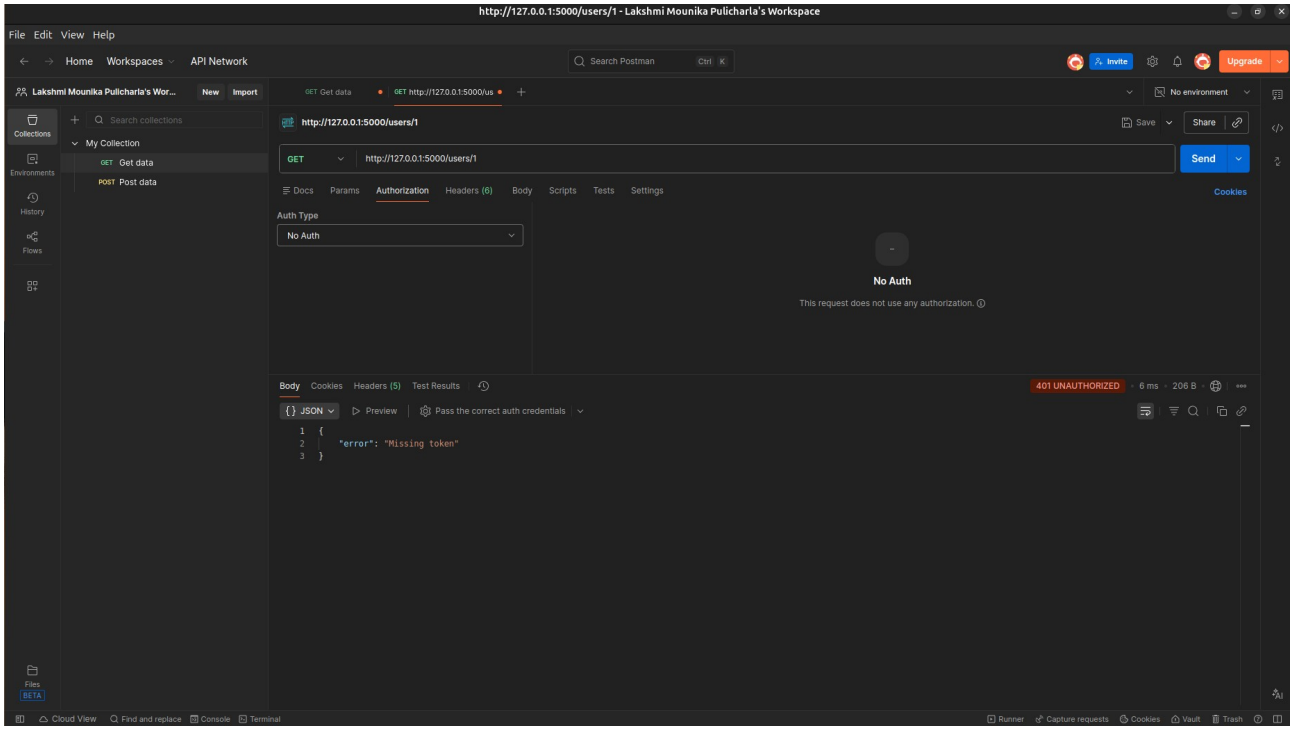
Screenshot — Invalid Token Test



4.No Authentication Testing

I removed authentication token and sent request. API returned 401 Unauthorized error. This shows API blocks unauthenticated users.

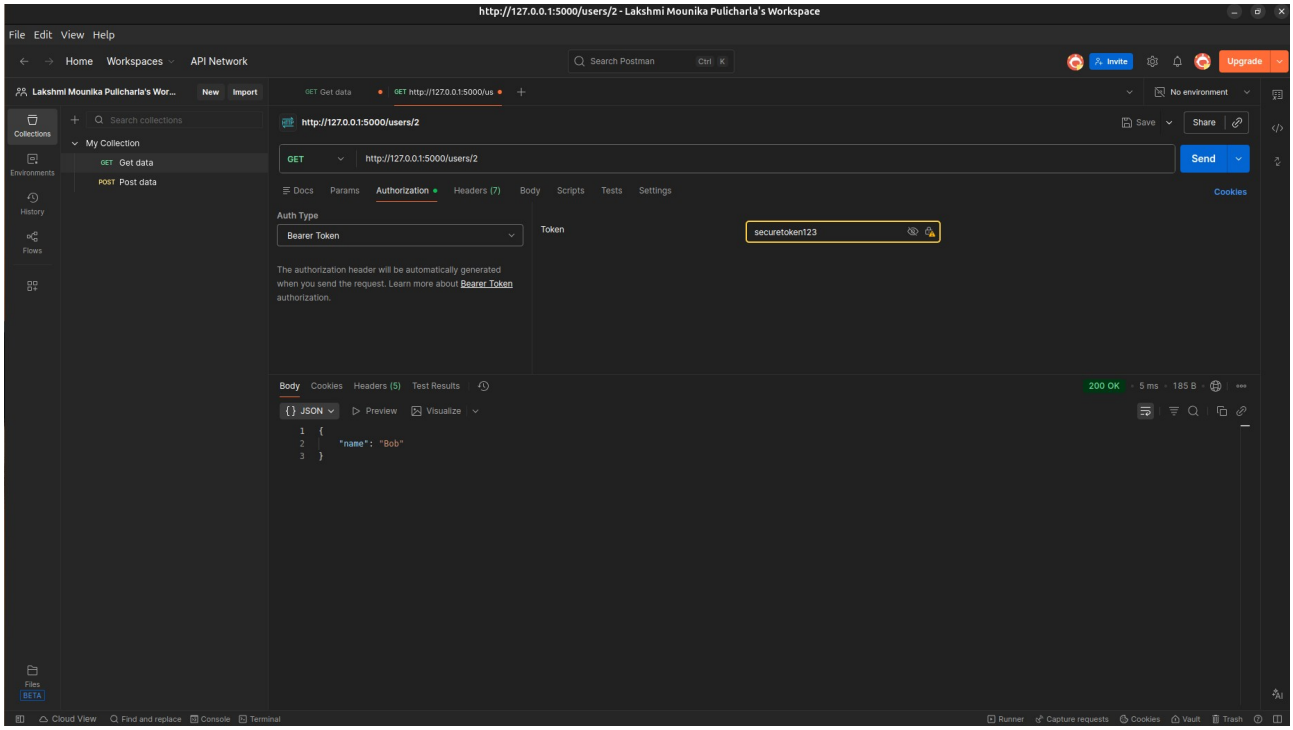
Screenshot — No Token Test



5.Authorization Testing (Broken Object Level Authorization Check)

I changed user ID in API request and tested if API returns data. API returned requested user data. In real systems, if users can access other user data, it can cause Broken Object Level Authorization vulnerability.

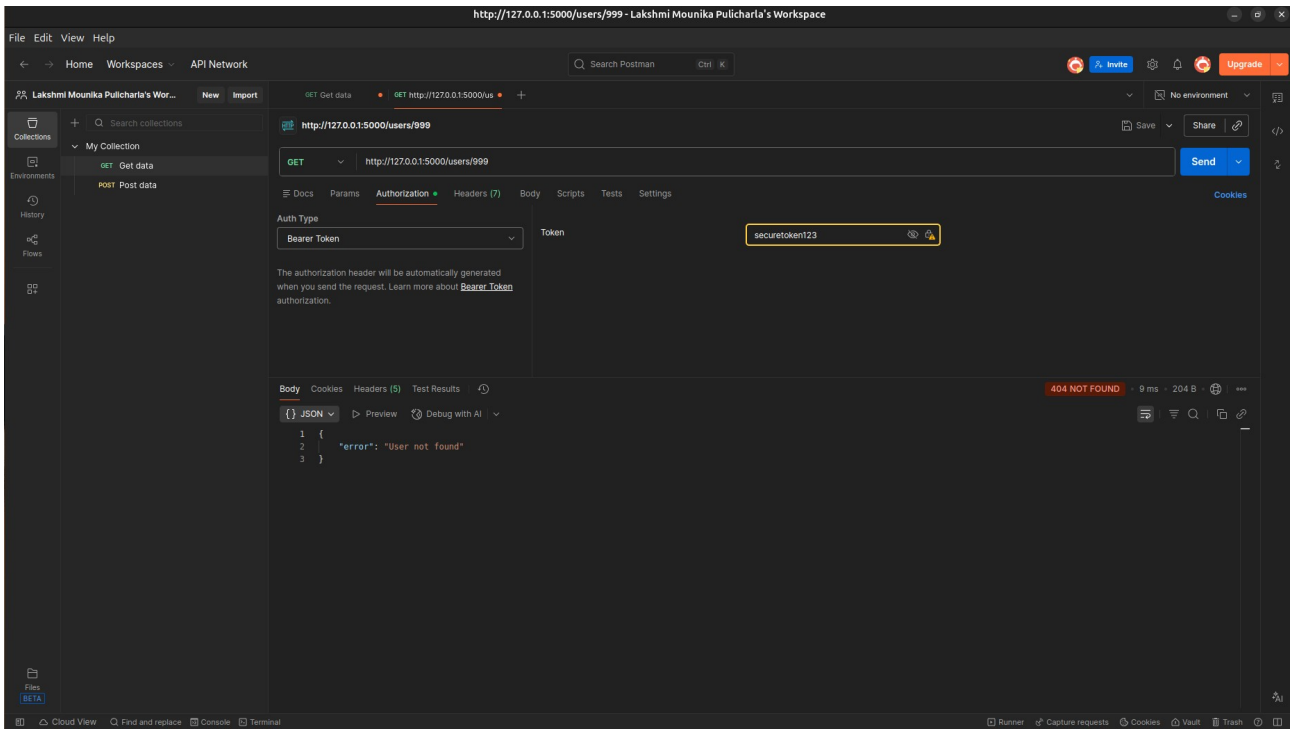
Screenshot — User ID Change Test



6.Input Validation Testing

I tested API by requesting non-existing user ID. API returned 404 error with proper message. This shows API properly handles invalid input.

Screenshot — Invalid User Test



7.Rate Limiting Testing

Since rate limiting was not implemented in my API, multiple rapid requests were accepted. In real-world applications, lack of rate limiting can lead to denial of service attacks.

8.HTTP Response Analysis

I analyzed response codes and messages.

200 → Successful request
401 → Unauthorized access
403 → Forbidden access
404 → Resource not found

API returned proper error messages and status codes.

9.OWASP API Risk Mapping

Broken Authentication
→ Tested using valid, invalid, and missing tokens

Broken Object Level Authorization
→ Tested using user ID modification

Input Validation Issues
→ Tested using invalid user ID

Security Misconfiguration
→ Checked error handling and responses.

10.Conclusion

In this task, API security testing was performed using Postman and a locally created Flask API. Different types of tests like valid token testing, invalid token testing, no authentication testing, authorization testing, and input validation testing were done. The API responses were checked using HTTP status codes and error messages. Based on the testing results, possible security risks related to authentication, authorization, and input validation were analyzed using OWASP API risk concepts. Overall, the task was successfully completed by testing the API for common security misconfigurations and authorization issues.