EXPLORING API SECURITY – WEEK 1

IDENTIFYING VULNERABILITIES IN THE OPENAPI V3 SPECS FOR VAMPI

1. INTRODUCTION

This report documents my Week One progress in the CyberGirls API Security training, where I explored key concepts and conducted hands-on vulnerability testing on a sample API. The goal is to understand common API security threats and apply practical methods to identify them.

2. WHAT I LEARNT

This week, I completed the API Security Fundamentals course on APIsec University. Through this, I learned:

- The structure of modern API security threats and how they differ from traditional web security issues.
- The OWASP API Security Top 10 and real-world implications of each category
- The importance of API authentication, authorization, proper rate limiting and input validation, avoiding exposure of internal objects, especially in error responses etc.
- The value of secure-by-design approaches in API development and deployment

From the practical assignment, I was able to apply what I learnt in theory hands-on in

- JWT-based authentication
- Using Postman with Bearer tokens
- Identifying vulnerabilities like Excessive Data Exposure, Mass Assignment and Broken Object Level Authorization (BOLA)
- Understanding how unprotected endpoints and weak access control models can be exploited in real-world APIs

3. LAB PROGRESS

a) Initial Setup

I converted the provided openapi3.yaml file to JSON using Swagger Editor and imported it into Postman for easier API interaction.

I initially faced connection errors with Postman pointing to the local API. To test endpoint functionality, I created a mock server in Postman and verified basic request behavior.

b) Vulnerability 1 Identified: Excessive Data Exposure

I first identified the endpoint */createdb*, which creates and populates the database with dummy data to provide some base data to start with.

Now I proceeded to register a new user using the Register new user endpoint */users/v1/register*.

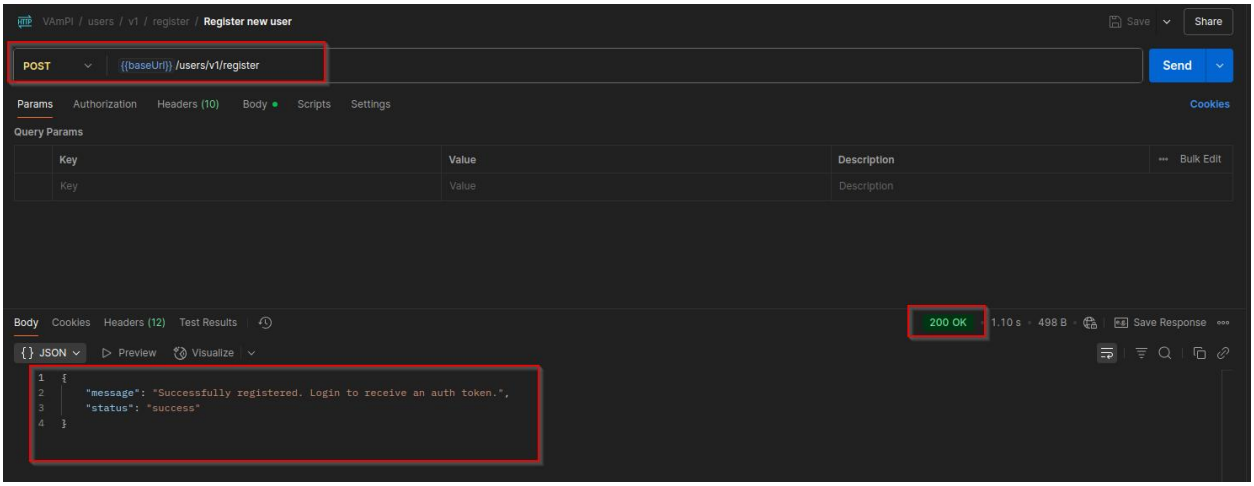

Figure 1: Created new user 'name1'

Then I logged this new user in using the Login to endpoint */users/v1/login*.

After logging in, I received an auth token, message and status.

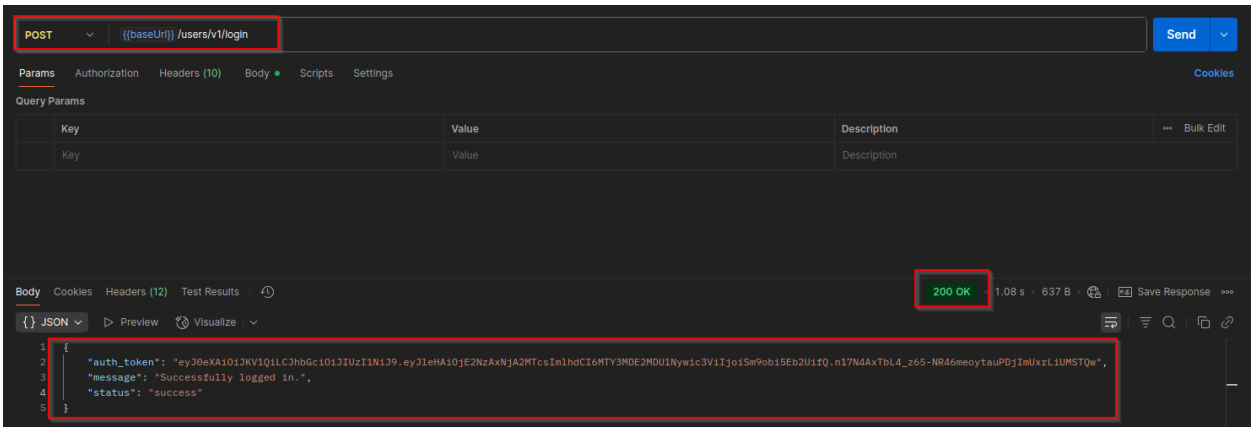

Figure 2: Login successful

In the response body, the username, password and email of the user are returned.

You can see that a JSON Web Token (JWT) is provided in the response body.

I copied this JWT and pasted it into a new collection variable.

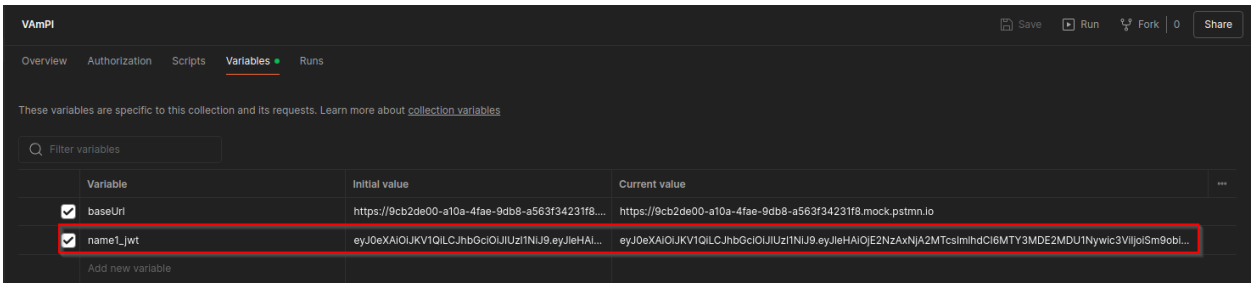

Figure 3: Jwt token variable

I navigated to the Authorization tab. In the Type drop-down menu, I selected Bearer Token and then set the variable I just created as the token value.

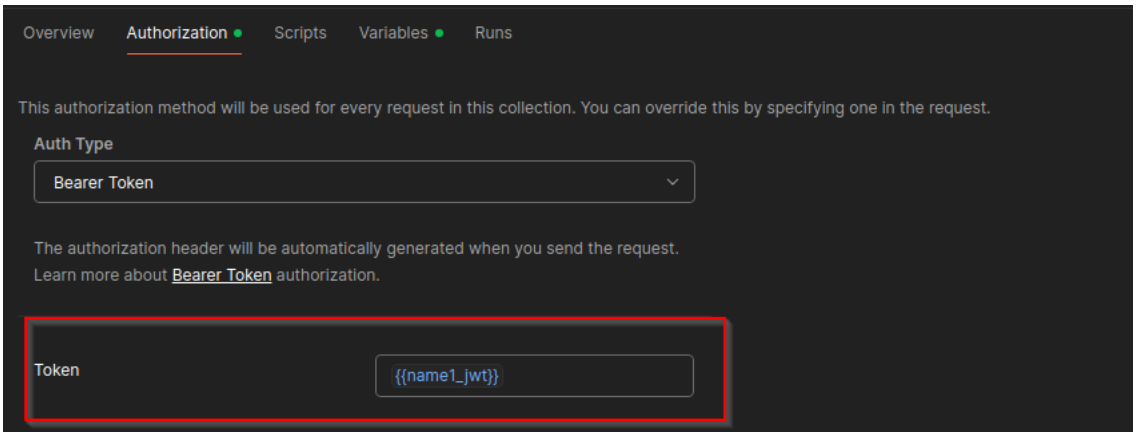

Figure 4: Token for authorization

Now, whenever I need to reissue a new token to this user, this token will be used in the Authorization request header for API calls.

Once this was set up, I tested the Retrieves all data for all users' endpoint */users/v1/_debug*. This endpoint is typically used for debugging purposes and may have been left behind unintentionally by the developer before releasing the production version of the API.

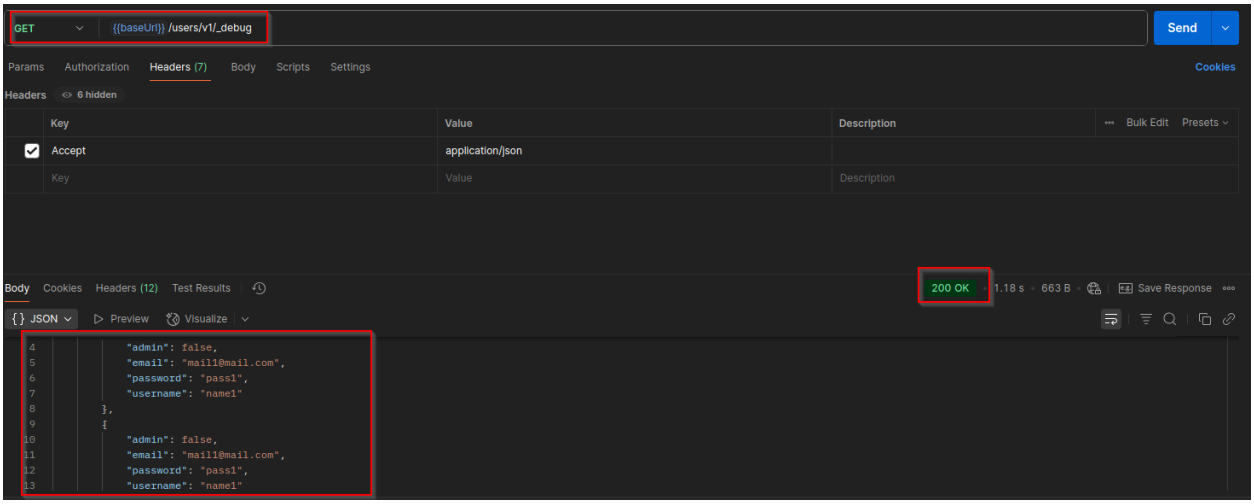

Figure 5: Excessive data exposure

As shown in the response, this endpoint returns the email, password and username of all users in the database.

It also reveals a parameter called admin, which is a boolean and can be either true or false.

This is **the Excessive Data Exposure** through the debug endpoint vulnerability.

c) Vulnerability 2 Identified: Mass Assignment

Now that I have found this admin parameter, I attempted to see whether I could use it to create an admin user.

I returned to the Register new user endpoint and created a new user.

This time, I added an "admin": true line to the request body, as shown below:

Figure 6: Creating an admin user

You can see the request was accepted successfully.

To verify this, I sent another request to the Retrieves all data for all users' endpoint */users/v1/_debug*.

In the response body, the newly created user appears at the bottom of the list with an admin status set to true.

Figure 7: Admin user created

B

d) Vulnerability 3 Identified: Broken Object Level Authorization (BOLA)

I logged in as the new user I just created and used this user's JWT in the Authorization request header.

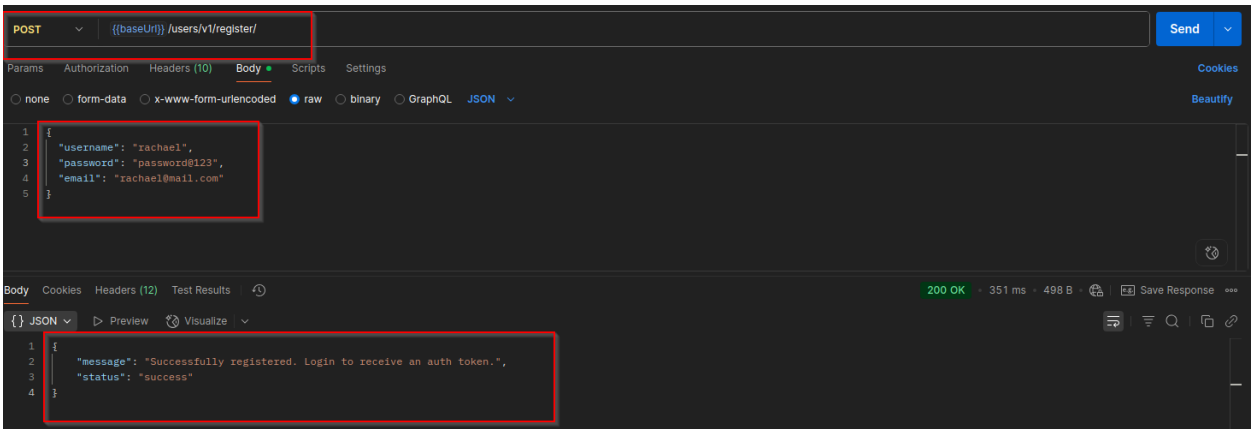

Figure 8: Second user created

Since I will be switching between the JWTs of multiple users, I created a collection variable for this second user's JWT and used that variable in the Token field of the Authorization tab in the collection.

Next, I went to the Add new book endpoint *POST /books/v1*. As user Rachael, I created a new book.

Then I added a book successfully using her JWT.

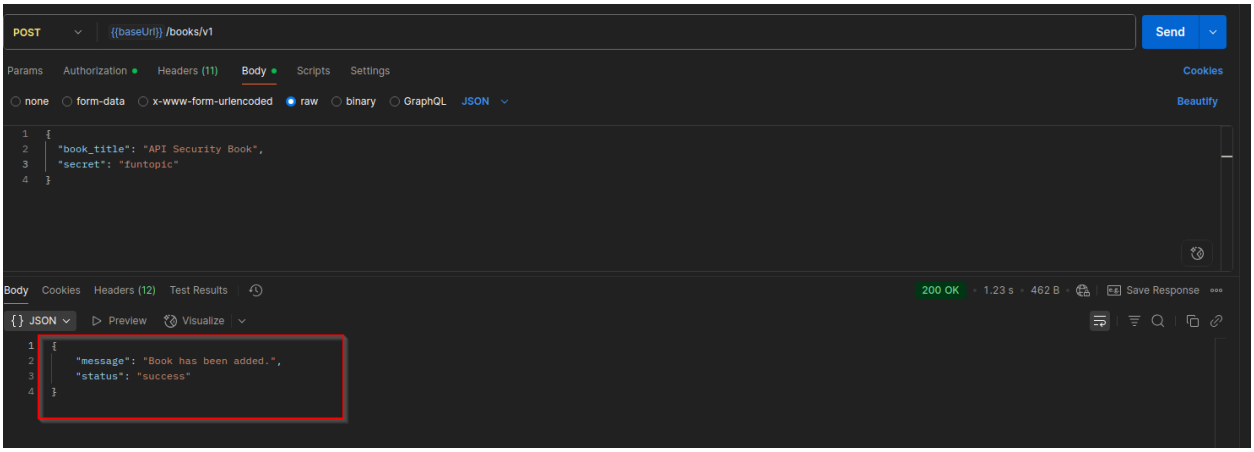

Figure 9: Added a book for the new user successfully

After that, I retrieved all books using the appropriate endpoint.

In the response body, the list of all books in the database is returned with their titles and corresponding user, but the *secret* is not displayed.

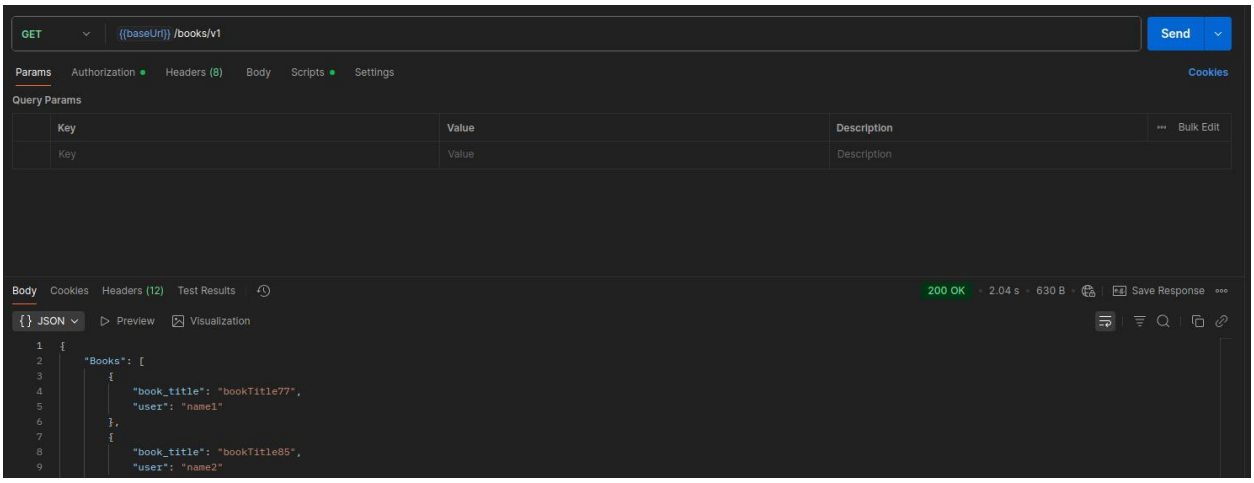

Figure 10: Retrieves all books but secret is not displayed

Each book is unique to its respective user and only the owner should be allowed to view the secret.

Then, I accessed the Retrieves book by title along with secret endpoint */books/v1/:book_title* and switched to user1's JWT in the collection's Authorization tab. The request was successful and returned the book's title, the submitting user and the secret.

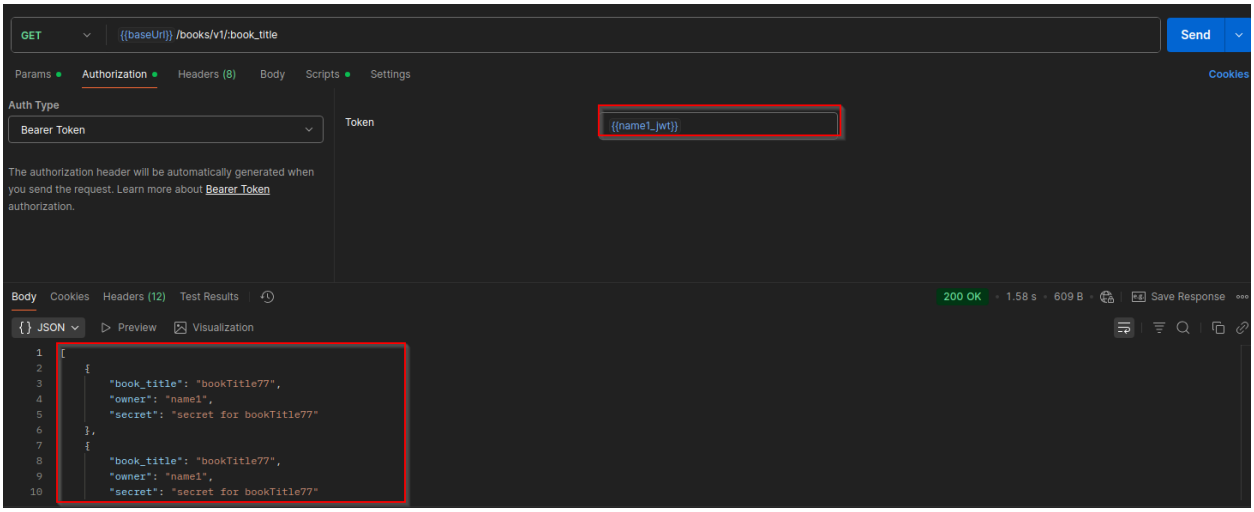

Figure 11: Secret is now returned

This confirms a **Broken Object Level Authorization (BOLA) vulnerability**. A user is able to access sensitive resources of other users by simply switching JWTs and targeting objects not owned by them.

4. RISK SUMMARY AND REALISTIC MITIGATION RECOMMENDATIONS

a) Excessive Data Exposure

This occurs when an API returns more data than necessary, exposing sensitive information (like email, password and admin status) in API responses. This is especially risky if such endpoints are left enabled in production.

Fixes and Mitigations

- Remove or secure debugging endpoints (/users/v1/_debug) from production environments.
- Use response filtering on the server side to return only the minimum necessary data based on user role or request context.
- Never return sensitive fields (passwords, admin status) unless absolutely required and never return passwords even in hash form.
- Implement access controls that restrict who can view debug or administrative data.

b) Mass Assignment

This vulnerability occurs when the API blindly accepts and maps client-provided input to internal object properties without filtering, allowing users to modify protected fields (setting admin: true during registration).

Fixes and Mitigations

- Explicitly whitelist allowed input parameters (only accept username, email, password) and reject all others.
- Use Data Transfer Objects or serializers that define which fields can be set by the client.
- Never bind request data directly to sensitive model attributes like admin, is_superuser, roles, etc.
- Implement validation logic that checks the legitimacy of role or privilege-related values before saving to the database.

c) Broken Object Level Authorization

BOLA occurs when access controls are missing or improperly enforced, allowing users to access objects (books or secrets) they do not own. Simply changing an identifier or token enables unauthorized access.

Fixes and Mitigations

- Enforce object ownership checks on every data access request, ensure that the user making the request is authorized to access the specific resource.
- Do not rely solely on client-provided tokens or object identifiers; always verify ownership on the server.
- Use contextual authorization: For example, match *book.user_id == auth_user.id* before returning data.
- Implement centralized authorization logic to standardize access checks across endpoints.

5. BLOCKERS FACED AND HOW I OVERCAME THEM

Initially encountered networking issues in Postman when attempting to connect to the local API. This was resolved by mocking requests and adjusting local IP settings to ensure proper routing and connectivity.

Additionally, I faced conceptual difficulties in accurately classifying vulnerabilities within the OWASP API Security Top 10 categories, particularly because some security flaws exhibit overlapping characteristics. To resolve this, I thoroughly reviewed the course materials multiple times and supplemented my understanding through additional external research. This comprehensive approach enabled me to confidently distinguish between the categories and correctly attribute each identified vulnerability to its appropriate classification.

6. CONCLUSION

Through the systematic testing and analysis of the target API, several critical security flaws aligned with the OWASP API Security Top 10 were identified and validated. These included Excessive Data Exposure, Mass Assignment and Broken Object Level Authorization (BOLA), each representing a significant risk to the confidentiality, integrity and access control mechanisms of the API.

The report documents the vulnerabilities discovered but also offers realistic and actionable mitigation strategies for each flaw. These recommendations aim to assist developers in strengthening the security posture of their API by implementing secure coding practices, proper access controls and robust validation mechanisms.

This exercise underscores the importance of continuous security testing in API development workflows and highlights the need for security awareness throughout the software development lifecycle.

7. NEXT STEPS

Having identified some of the key vulnerabilities, I plan to continue testing and expanding my understanding of API security through the following actions:

- Investigate other potential vulnerabilities such as Security Misconfiguration and Lack of Rate Limiting using manual testing and tools like Postman or Burp Suite.
- Experiment with modifying headers, parameters and payloads to uncover hidden or undocumented behaviors, edge cases or improper validation.
- For each new vulnerability found, document the exploitation process clearly, including setup, attack steps and impact, along with the mitigation strategies.
- Applying secure design concept such as least privilege, schema validation, proper use of authentication tokens and defense-in-depth strategies.

8. REFERENCES

OWASP Foundation, *OWASP API Security Top 10 - 2023*. [Online]. Available: https://owasp.org/API-Security/editions/2023/en/0x00-introduction

OWASP Foundation, "API3:2023 - Broken Object Property Level Authorization," in *OWASP API Security Top 10 - 2023*. [Online]. Available: https://owasp.org/API-Security/editions/2023/en/0xa3-broken-object-property-level-authorization/

Postman, Inc., "Postman Learning Center – API Testing," [Online]. Available: https://learning.postman.com/docs/writing-scripts/script-references/test-examples/

OWASP Foundation, "OWASP Cheat Sheet Series – REST Security Cheat Sheet," 2023. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html