API Security Testing

Mike Morin

April 2025

Table of Contents

Executive Summary

This project consisted of a focused security testing assessment on Payatu's Damn Vulnerable API (DVAPI) to identify weaknesses aligned with the OWASP Top 10 API Security Risk. Using Postman, I performed systematic endpoint discovery, analyzed request and response patterns, and mapped the API's functional behavior. I then used Burp Suite to intercept and manipulate traffic, perform active tests, and replicate common attack techniques used against modern APIs.

The security testing process revealed several vulnerabilities, including issues related to authentication, authorization, and data exposure. All findings were validated through practical attack simulations and documented strictly based on observable behavior, without providing remediation steps. This project strengthened my hands-on skills in API vulnerability discovery, request manipulation, and structured security testing methodology.

Environment Setup

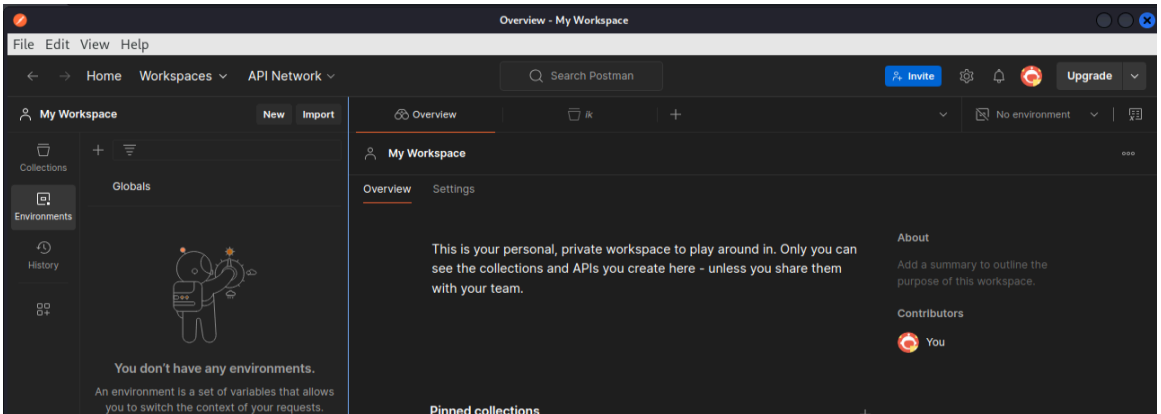
Installation and configuration of the **Damn Vulnerable API (DVAPI)** in an isolated lab environment.

```
┌──(root㉿kali)-[/home/mike/Downloads/DVAPI-main]
└─# apt install -y docker.io
Installing:
  docker.io
```

```
┌──(root㉿kali)-[/home/mike/Downloads/DVAPI-main]
└─# apt install docker-compose
Installing:
  docker-compose
```

```
┌──(root㉿kali)-[/home/mike/Downloads/DVAPI-main]
└─# docker-compose up --build
Creating network "dvapi-main_default" with the default driver
Creating volume "dvapi-main_mongodb-data" with default driver
Pulling mongodb (mongo:latest)...
```

Postman installation for API interaction and testing.



API Security Assessment Tasks

Endpoint Discovery

Endpoint	**Method**	**parameter**
http://localhost:3000/api/register	POST	username, password
http://localhost:3000/api/login	POST	username, password
http://localhost:3000/api/profile	GET	N/A
http://localhost:3000/api/profile	POST	current_passwor, password, confirm_password
http://localhost:3000/api/profile/upload	POST	file
http://localhost:3000/api/user/<user>	GET	username
http://localhost:3000/api/logout	GET	N/A
http://localhost:3000/api/addNoteWithLink	POST	url
http://localhost:3000/api/addNote	POST	note
http://localhost:3000/api/getNote?userName=<user>	GET	username
http://localhost:3000/api/flag/submit	POST	challengeNo, flag
http://localhost:3000/api/allChalanges	POST	released
http://localhost:3000/api/getSolves	GET	N/A
http://localhost:3000/api/scores	GET	N/A
http://localhost:3000/api/addTicket	POST	message
http://localhost:3000/api/getTicket	POST	ticketno

Testing for OWASP API Security Top 10 Vulnerabilities:

Broken Object Level Authorization (BOLA):

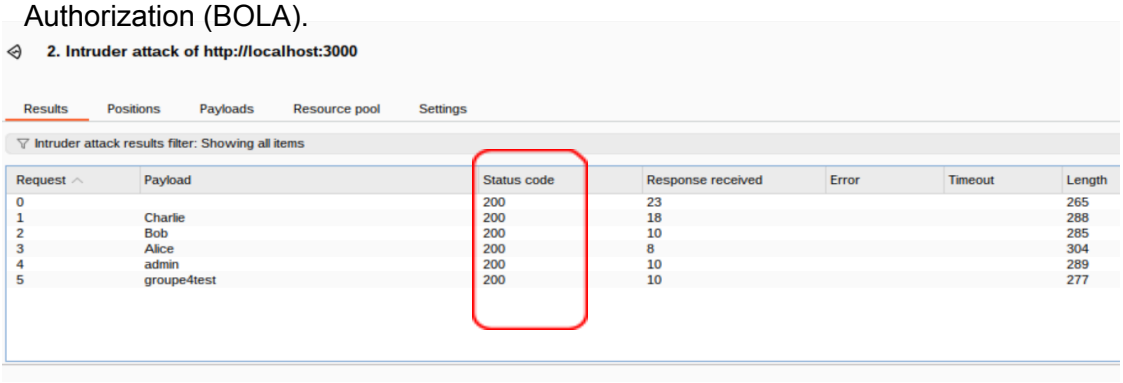
Broken Object Level Authorization (BOLA) is a security vulnerability where an application does not properly enforce authorization checks at the object level, leading to unauthorized access to sensitive data or actions.

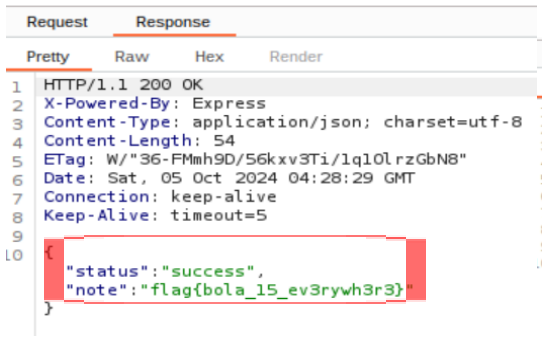
Upon reviewing the API documentation, we identified two noteworthy endpoints: one that retrieves a list of all users along with their scores, and another that might be vulnerable to Broken Object Level Authorization. (BOLA).

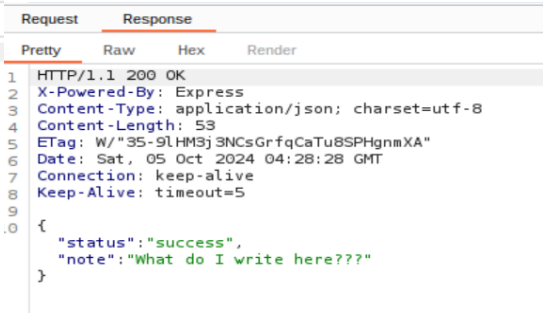
1- http://localhost:3000/api/scores: This endpoint allows a registered user to view the scores of all other competing users.

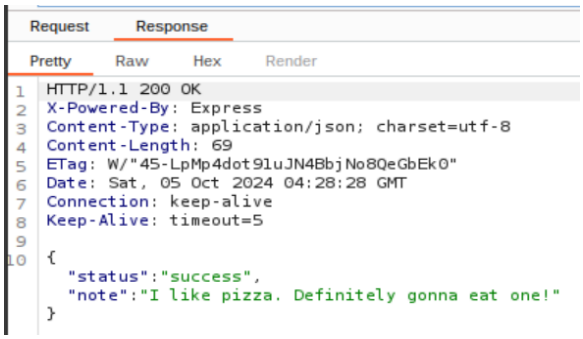

```
{
    "status": "success",
    "scores": [
        {
            "username": "Charlie",
            "score": 300
        },
        {
            "username": "Bob",
            "score": 200
        },
        {
            "username": "Alice",
            "score": 100
        },
        {
            "username": "admin",
            "score": 0
        },
        {
            "username": "groupe4test",
            "score": 0
        },
        {
            "username": "attacker",
            "score": 0
```

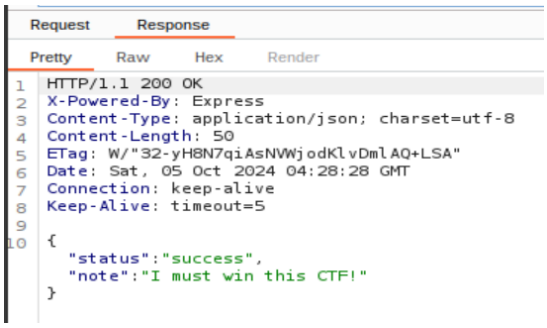

2- http://localhost:3000/api/getNote?username=user: This endpoint allows a user to read their own secret note, which is intended to be accessible only by the creator. We use Burp Suite to modify the endpoint's parameter to determine if it is vulnerable to Broken Object Level Authorization (BOLA).









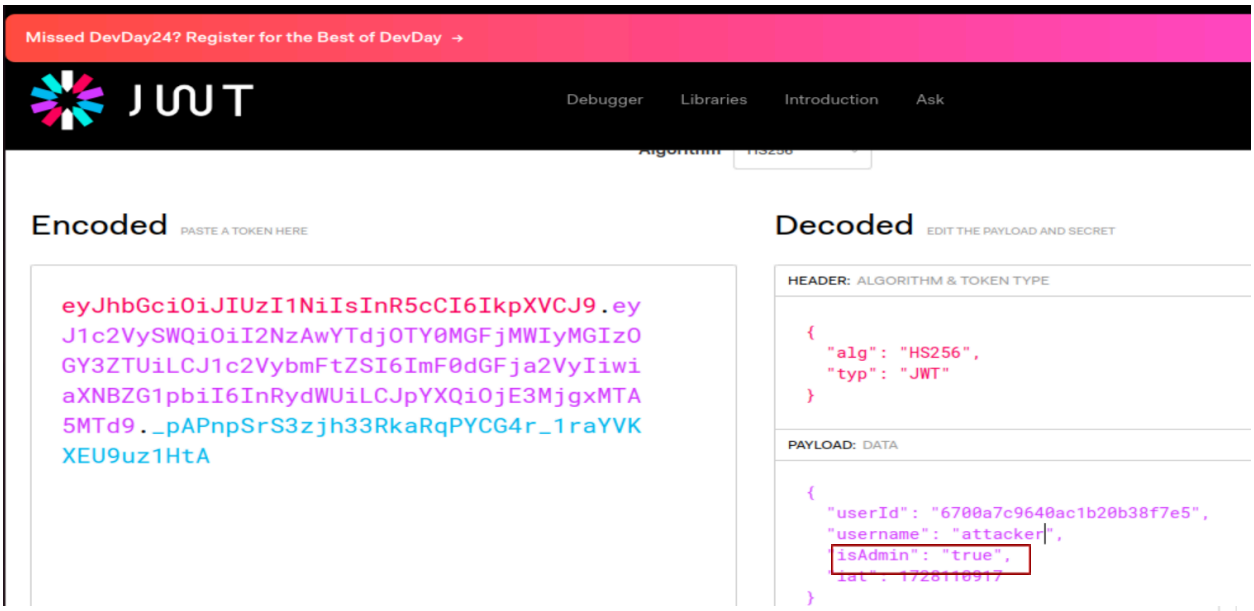


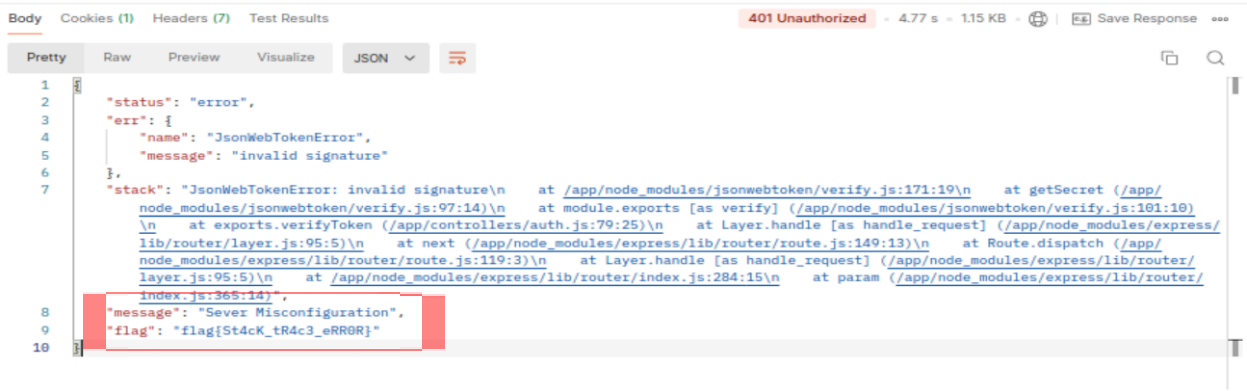
The secret notes of other users are accessible without proper authorization, and notably, we were able to retrieve the flag from the admin's note.

Excessive Data Exposure

This vulnerability refers to a situation where an API exposes more information than is necessary or intended, potentially leading to security vulnerabilities and privacy concerns.

We attempted to modify the API authentication token by altering the “isAdmin” field from false to true. After importing the token into the API in an effort to escalate privileges. The API responded with error messages, revealing some confidential information that should not have been disclosed.





Broken User Authentication And Injection Flaws

Broken User Authentication is a vulnerability where attackers can exploit flaws in the authentication process to impersonate other users, including administrators. This typically occurs due to weak, poorly implemented, or improperly enforced authentication mechanisms.

The causes of broken user authentication typically involve weak password policies, absence of multi-factor authentication (MFA), lack of rate limiting on login attempts, insecure password recovery mechanisms, and poor session management practices, which can all contribute to the ease with which attackers compromise user accounts. Additionally, improper implementation of protocols like JWT or OAuth, failure to implement account lockout mechanisms, and SQL injection vulnerabilities are significant risks. SQL injection can allow attackers to bypass authentication entirely by injecting malicious SQL code into login fields, leading to unauthorized access and exposing sensitive data.

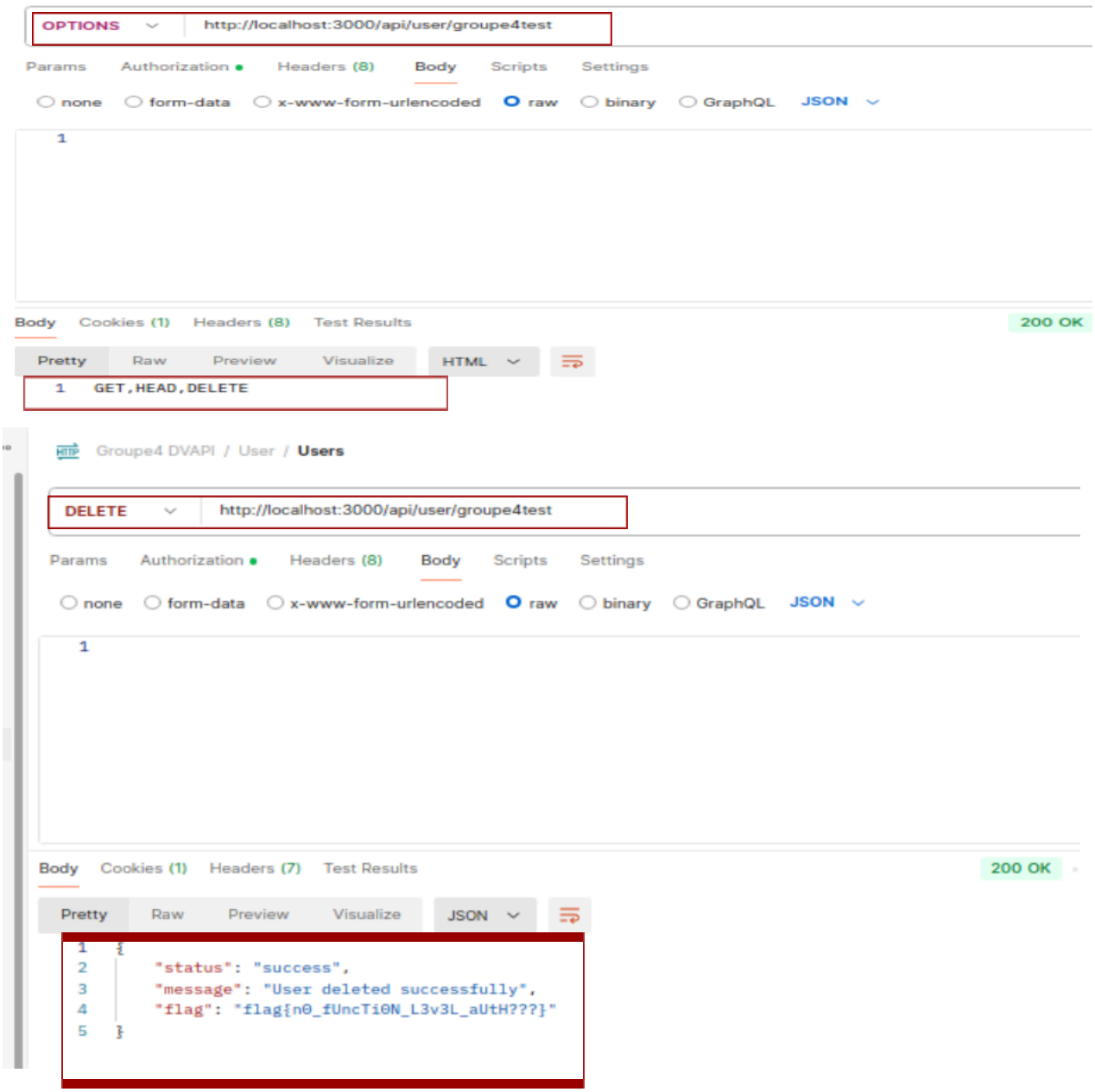
We attempted multiple methods to bypass the authentication process, including token alteration and brute-force password attacks, but none were successful. Knowing that the API uses MongoDB, a NoSQL database, we explored the possibility of a NoSQL injection. We employed the username “admin” combined with the “$ne” operator, which stands for “not equal.” This operator filters results based on the condition that the specified field should not match a given value. By using the payload { "username": "admin", "password": { "$ne": null } }, we successfully bypassed the authentication mechanism.

Broken Function Level Authorization (Broken Access Control)

Broken Function Level Authorization refers to a security vulnerability where an application does not properly enforce permissions at the function or endpoint level. This allows users to access functionalities they should not have permission to use, potentially exposing sensitive data or allowing unauthorized actions.

We utilized the OPTIONS method to identify the communication options available for the

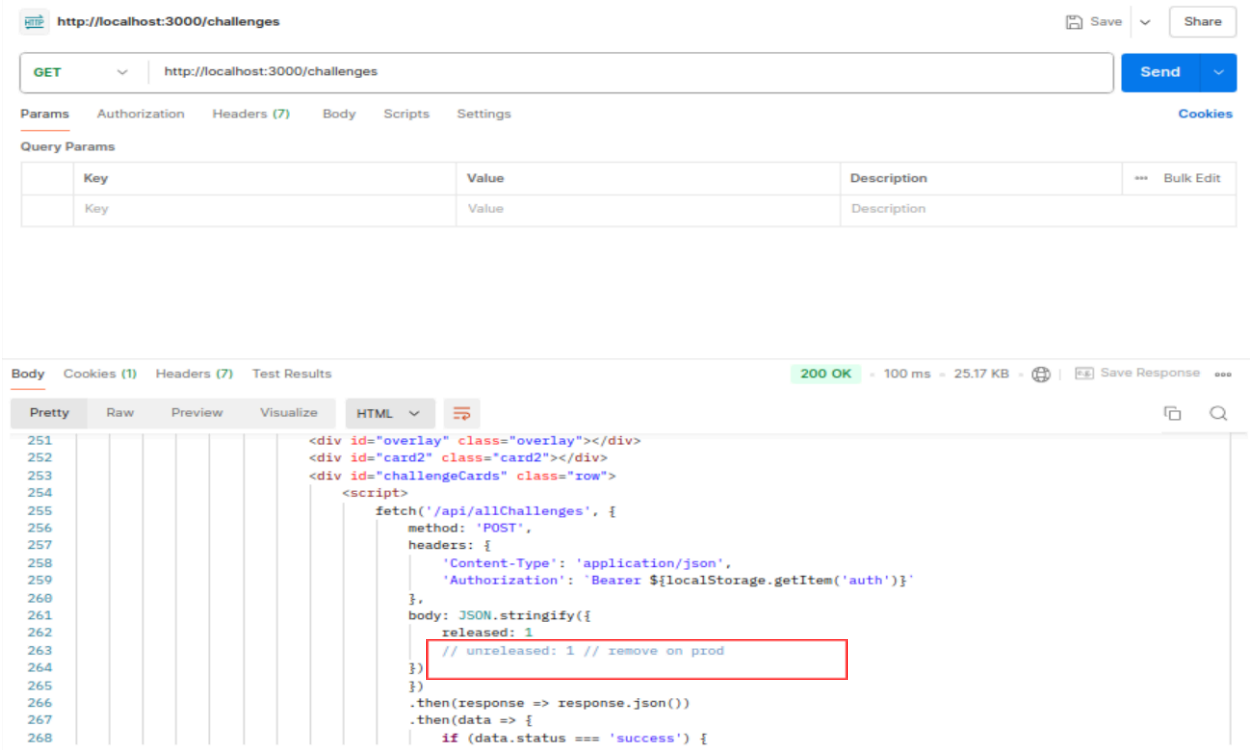
resources and compared the results with the documentation. We observed that the endpoint http://localhost:3000/api/user/<username> allows the DELETE method, which is not mentioned in the documentation. Upon investigating this method, we discovered that a user can delete another user's account without the necessary permissions.

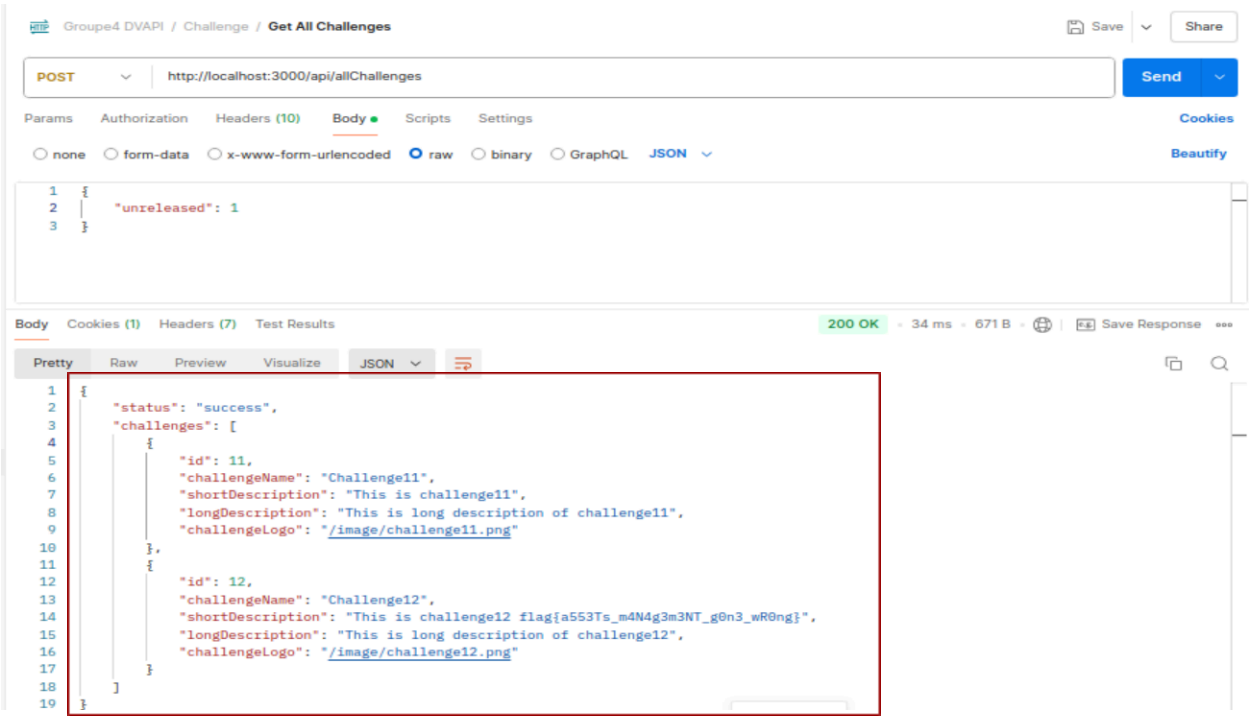


Improper Assets Management

Improper Inventory Management refers to vulnerabilities arising from poorly managed API endpoints, such as exposing undocumented or internal APIs without proper access controls, which leads to increased attack surfaces and unauthorized access. This can result in attackers discovering and exploiting hidden functionalities, gaining access to sensitive data, or performing

unauthorized actions.
The endpoint http://localhost:30000/allChallenges is intended to return all available challenges, with a parameter {released: 1} indicating which challenges to include. We tested this endpoint for improper inventory management by modifying the released parameter to values such as 0, 2, and null, and also tried different API versions like v1, v2, staging, production, and prod, but none of these worked. Finally, we replaced allChallenges with challenges, and although it did not return the expected results, the response contained valuable information. The URL provided HTML and JavaScript code used to display the challenges, revealing that the developer had commented out an unreleased parameter intended to be removed in production, which had unfortunately not been done.





Conclusion

This project involved hands-on security testing of Payatu's Damn Vulnerable API (DVAPI) to identify weaknesses aligned with the OWASP Top 10 API Security Risk. Using Postman for structured endpoint enumeration and Burp Suite for traffic interception and active testing, I was able to uncover several critical vulnerabilities, including Broken Object Level Authorization (BOLA), Excessive Data Exposure, Broken User Authentication, Injection Flaws, Broken Function Level Authorization, and Improper Asset Management. These findings demonstrate the range of potential security risks present in modern API implementations.

The assessment strengthened my practical skills in API security testing, including reconnaissance, vulnerability validation, and attack simulation. It also highlighted the importance of carefully managing authentication, authorization, and data exposure in API design. Overall, the project reinforces the need for continuous security testing to proactively identify and understand potential risks in API-driven applications.