Tester: Falilat Owolabi (CSF/API/25B/1095)

Fundamental Of Api Security Course Summary.

As the week progressed, I was able to understand the OWASP API Security Top 10, threat model, pillars of API and the lifecycle of every API, and I will summarize each as I understand them:

1) BOLA: Broken Object Level Authorization. If a user with a certain clearance level has access to data outside their level in a system or organization, it means the data are being exposed to those who should not have access, which is why it is called Broken Object Level Authorization.

How this can be prevented: Every user in the system should be explicitly checked to confirm they have authorization to access the data they are about to access at every given point in time.

2) Broken Authentication: A broken authentication vulnerability can exist in a system if user credentials are not checked thoroughly before allowing access, or if the credentials used by the system are too weak and can be guessed to gain access.

This can be prevented if the system implements a very strong credential system that cannot be guessed or brute-forced easily, and every user who needs to gain access into the system will have to prove to the core that they are who they claim to be.

NOTE: The difference between Authentication and Authorization:
Authentication uses a user's credentials to determine if they can access the system—that is, verifying the information needed to decide if the user can access anything in the system/organization.
Authorization determines what the user is allowed to access within the system/organization after being authenticated.

For example: Successful authentication means valid login details that result in a successful login to the system. Authorization is the role you have in the system that determines what you can read and write after you have logged in successfully.

3) Excessive Data Exposure (Broken Object Property Level Authorization): This vulnerability discloses more than necessary to users, which a malicious user can use against the system, potentially causing loss—for example, disclosing system configuration when there is an error from users.

This kind of vulnerability can be prevented if the amount of information displayed on every request is only the required information. If errors need to be displayed, use error codes or brief notifications to users about what caused the error from their input, not what caused the error from the system.

4) Unrestricted Resource Consumption: This occurs when there is no limit to the amount of requests that can be made at a point in time, which can lead to a malicious user causing DoS to the system or using that opportunity to brute-force login credentials until access is gained.

To avoid this type of issue, there should be rate limiting for every call that can be made.

5) Broken Function Level Authorization: I do not relate to this very well, but I understand it as a situation where a function was able to modify what it is not meant to modify, which means that function has been abused because an attacker is able to use it for what it is not meant to do.

To prevent this kind of issue, I am thinking by making sure every necessary measure is put in place to avoid a function level going beyond what they are meant to do.

6) Unrestricted Access to Sensitive Business Flow: This is an issue if a system allows a lot of access to the main and sensitive logic that holds the system together and makes it work properly. It can be abused by attackers who get their hands on the API.

To prevent this, the core logic of a system/organization should be restricted from every other person who is not in the highest hierarchy or anyone who does not really need it to make decisions on the system.

7) Server Side Request Forgery (SSRF): This is an issue if an attacker is able to send a request that will make the API server call another or unprotected third-party server in order to exploit the information the API server has.

8) Server Misconfiguration: This is an issue if there is any discrepancy, regardless of how small it is, in the way the server is configured, which can lead to unpredicted results that can be exploited.

Server/security should be configured correctly to avoid any unpredicted outcome.

9) Improper Inventory Management: This is due to the mismanagement of APIs in the system. For every API being used, its state should be properly documented, and unused APIs should not be accessed anymore.

Recommendation: Know the state of every API asset in the system and make sure they are in the right state.

10) Unsafe Consumption of APIs: APIs that are going to be used by a system should be trusted and verified APIs. If proper testing can be done by the organization before being used, it is good for the betterment of having confidence in their API.

It should be noted that while it might take more than 4 steps before an exploit is done in traditional cybersecurity, it only takes two steps for damage to be done in API security—from discovering vulnerability to attacking straight.

The threat modeling of API security involves:

- Identity

- Assess

- Probability

- Impact

- Mitigation

The 3 pillars of API security are:

- Governance

- Monitoring

- Testing

These pillars are very important to making sure every API in use is up to standard and free from any vulnerability.

Lastly, the lifecycle of API security involves the following order:

1. API Definition

2. API Development

3. API Testing

4. API Deployment

5. API Retirement.

End of course summary report!

VamPI- lab Report

Environment: analysis of OpenAPI 3.0.1 specification for VAmPI (Vulnerable API); API base URL at http://localhost:5000.
Analysis performed via OpenAPI specification review, Postman testing,

Report date: 2025-11-23

Format: Combined technical & business-focused report, presented chronologically in the order vulnerabilities were discovered, with mapping to OWASP API Security Top 10 Includes remediation guidance, and PoC evidence from OpenAPI specification analysis.

Executive Summary

Objective:

bjective:

The goal of this engagement was to perform a comprehensive security audit of the VAmPI (Vulnerable API) application through manual review and Dynamic analysis of its OpenAPI 3.0.1 specification. The assessment aimed to:

1. Review the complete API specification for security misconfigurations and design flaws.
2. Identify missing authentication and authorization controls documented in the specification.
4. Identify improper asset management practices (debug endpoints, administrative functions).

Approach:

Testing was performed from a security Tester perspective using:

- Dynamic analysis of OpenAPI 3.0.1 using Postman.
- Manual review of all endpoints, security schemes, request/response schemas.
- Identification of missing security: sections indicating unauthenticated endpoints.
- Analysis of path parameters and authorization patterns for IDOR vulnerabilities.
- Review of response schemas for excessive data exposure (PII, passwords, admin flags).
- Documentation of improper assets (debug endpoints) in production specifications.

Key high-level findings (prioritized):

1. **Broken Authentication:** debug endpoint (/users/v1/_debug) exposes all user credentials including passwords and admin flags without authentication. (OWASP: API2);
2. **Broken Object Level Authorization (BOLA)**: authenticated users can modify any user's password by changing path parameters; unauthenticated users can enumerate and access user profiles. (OWASP: API1).
3. **Improper Assets Management:** debug

endpoint (/users/v1/_debug) exists in production API specification, exposing development/testing functionality. (OWASP: API9)

Business impact: combined findings allow complete authentication bypass, account takeover, credential harvesting, user enumeration and unauthorized administrative actions — all of which would be material in production: regulatory exposure (GDPR/CCPA/HIPAA), financial loss, severe reputational damage, and potential legal liability.

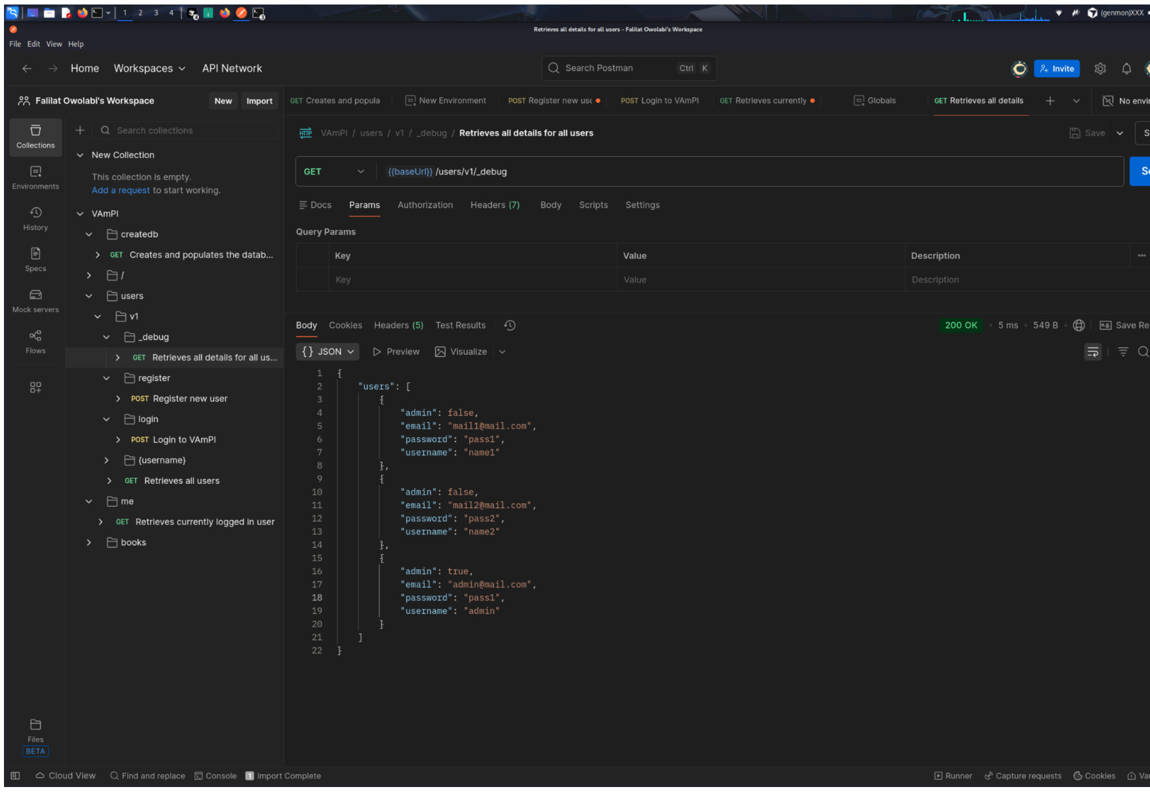
Immediate priorities: remove debug endpoints from production, implement authentication on all sensitive endpoints, enforce object-level and authorization checks,

1 Broken Authentication: unauthenticated access to user credentials via debug endpoint

How found

- Continued reviewing paths section and identified /users/v1/_debug endpoint (lines 88-119). Endpoint name contains "_debug" indicator. Reviewed response schema and discovered it returns user objects with password field in plaintext (line 114-116), admin boolean flags (line 108-110), and email addresses. Critical finding: no security: section present despite API defining bearerAuth security scheme (lines 9-14). This completely bypasses authentication system.

Evidence / PoC:



Retrieves all details for all users - Falilat Owolabi's Workspace

Falilat Owolabi's Workspace

VAmPI / users / v1 / _debug / Retrieves all details for all users

GET {{baseUrl}} /users/v1/_debug

200 OK • 5 ms • 549 B

```
{
    "users": [
        {
            "admin": false,
            "email": "mail1@mail.com",
            "password": "pass1",
            "username": "name1"
        },
        {
            "admin": false,
            "email": "mail2@mail.com",
            "password": "pass2",
            "username": "name2"
        },
        {
            "admin": true,
            "email": "admin@mail.com",
            "password": "pass1",
            "username": "admin"
        }
    ]
}
```

Impact: Complete authentication bypass allowing unauthenticated attackers to retrieve all user credentials including passwords in plaintext, email addresses, usernames, and admin status flags. This enables immediate account takeover, privilege escalation to admin accounts, and credential harvesting for all users in the system.

OWASP mapping: API2: Broken Authentication.

authentication system completely ineffective, complete credential exposure.

Remediation:

1. Immediate action: remove /users/v1/_debug endpoint from production API specification and implementation immediately.
2. Add authentication: require bearerAuth security scheme if debug functionality must exist.
3. Add authorization: ensure only authenticated administrators can access (if absolutely necessary).
4. Secure password storage: verify passwords are hashed and never returned in any API responses.
5. Rotate credentials: force password reset for all users after patching.

Verify fix: GET request to /users/v1/_debug should return 404 Not Found or 401 Unauthorized if endpoint exists. No credentials should be returned in any response

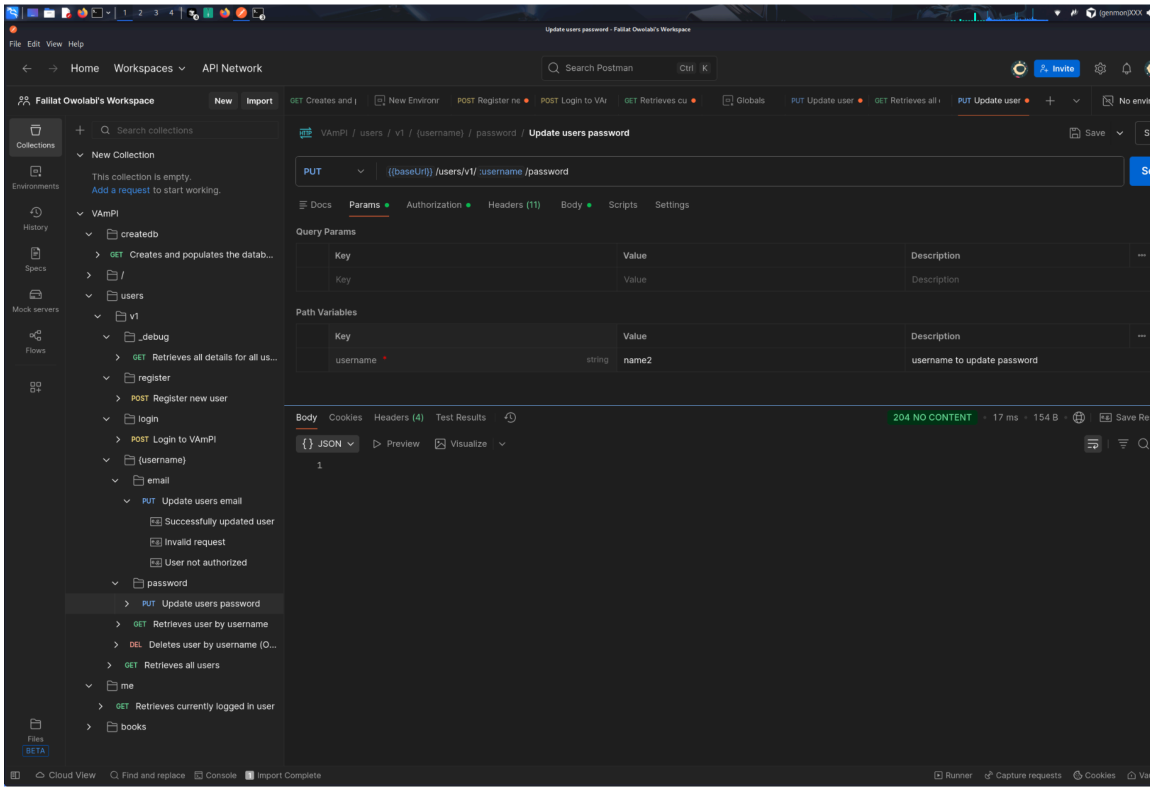
2 .Broken Object Level Authorization: unauthorized email modification

How found

- Reviewed PUT /users/v1/{username}/password endpoint (lines 426-485). authentication required (line 428-429) but username path parameter (lines 435-442) allows targeting any user. Request body accepts password field (lines 443-453). No authorization verification documented in specification.

Evidence / PoC:

Using the token gotten from name1 during the login end point give acces to changing other users password



```
PUT   {{baseUrl}} /users/v1/ :username /password
```

Query Params

Key	Value	Description
Key	Value	Description

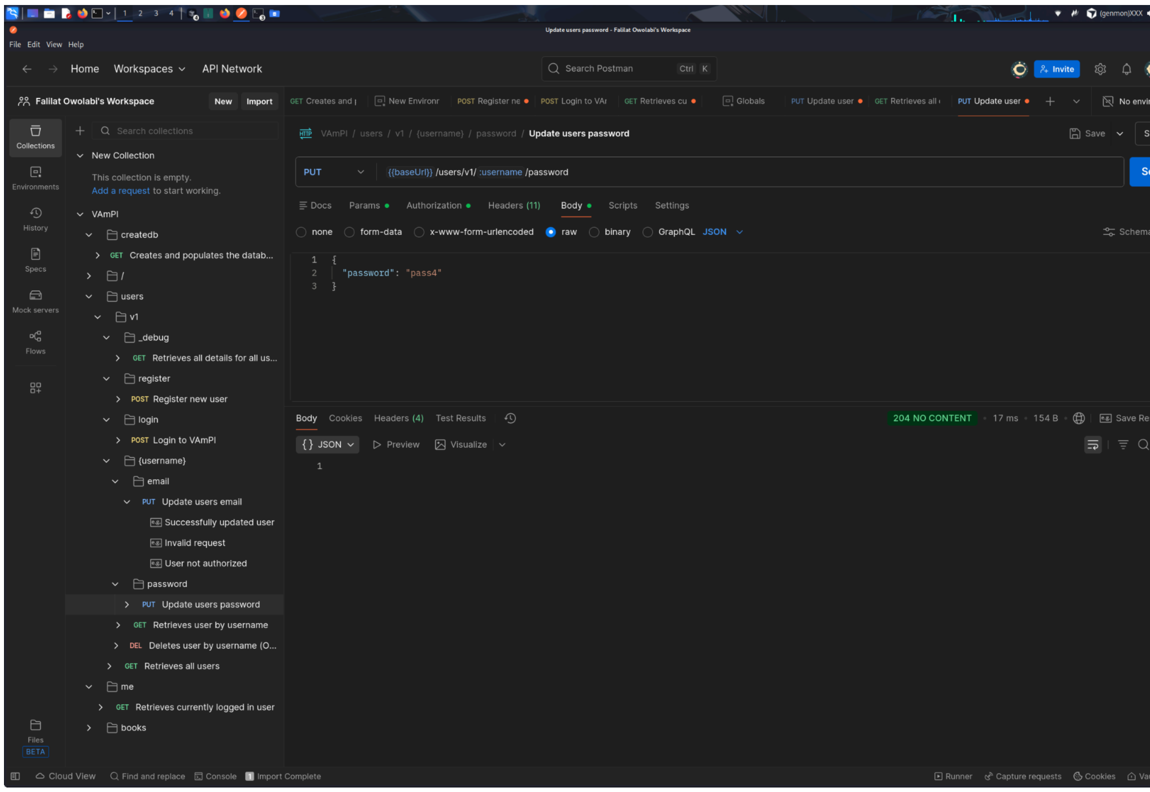
Path Variables

Key		Value	Description
username *	string	name2	username to update password

204 NO CONTENT · 17 ms · 154 B

{ } JSON

```
1
```



```
PUT   {{baseUrl}} /users/v1/ :username /password
```

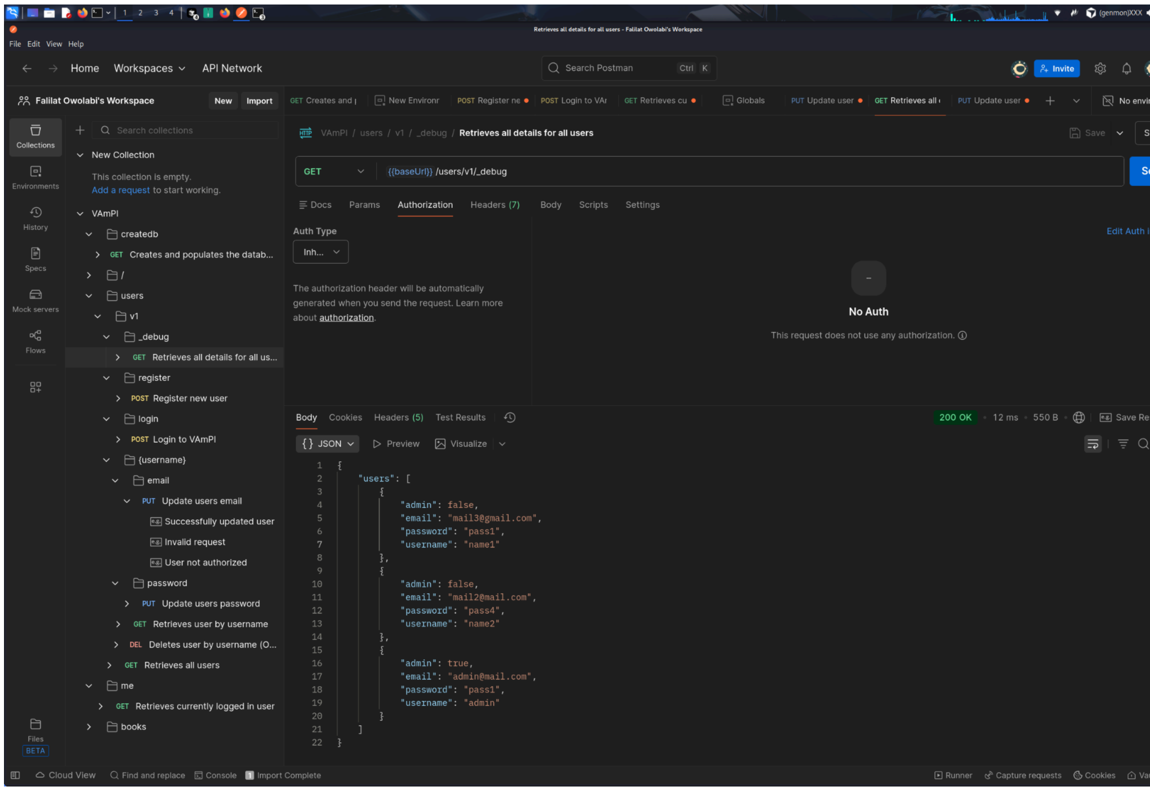
none · form-data · x-www-form-urlencoded · [✓ raw] · binary · GraphQL · JSON

```
{
  "password": "pass4"
}
```

204 NO CONTENT · 17 ms · 154 B

{ } JSON

```
1
```



GET {{baseUrl}} /users/v1/_debug

The authorization header will be automatically generated when you send the request. Learn more about authorization.

No Auth

This request does not use any authorization.

200 OK • 12 ms • 550 B

```
{
    "users": [
        {
            "admin": false,
            "email": "mail3@gmail.com",
            "password": "pass1",
            "username": "name1"
        },
        {
            "admin": false,
            "email": "mail2@mail.com",
            "password": "pass4",
            "username": "name2"
        },
        {
            "admin": true,
            "email": "admin@mail.com",
            "password": "pass1",
            "username": "admin"
        }
    ]
}
```

Impact: Critical account takeover vulnerability. Authenticated users can change any user's password including administrators, enabling complete account hijacking, privilege escalation, and unauthorized system access. Most severe form of IDOR vulnerability.

OWASP mapping: API1: Broken Object Level Authorization (BOLA).

Remediation:

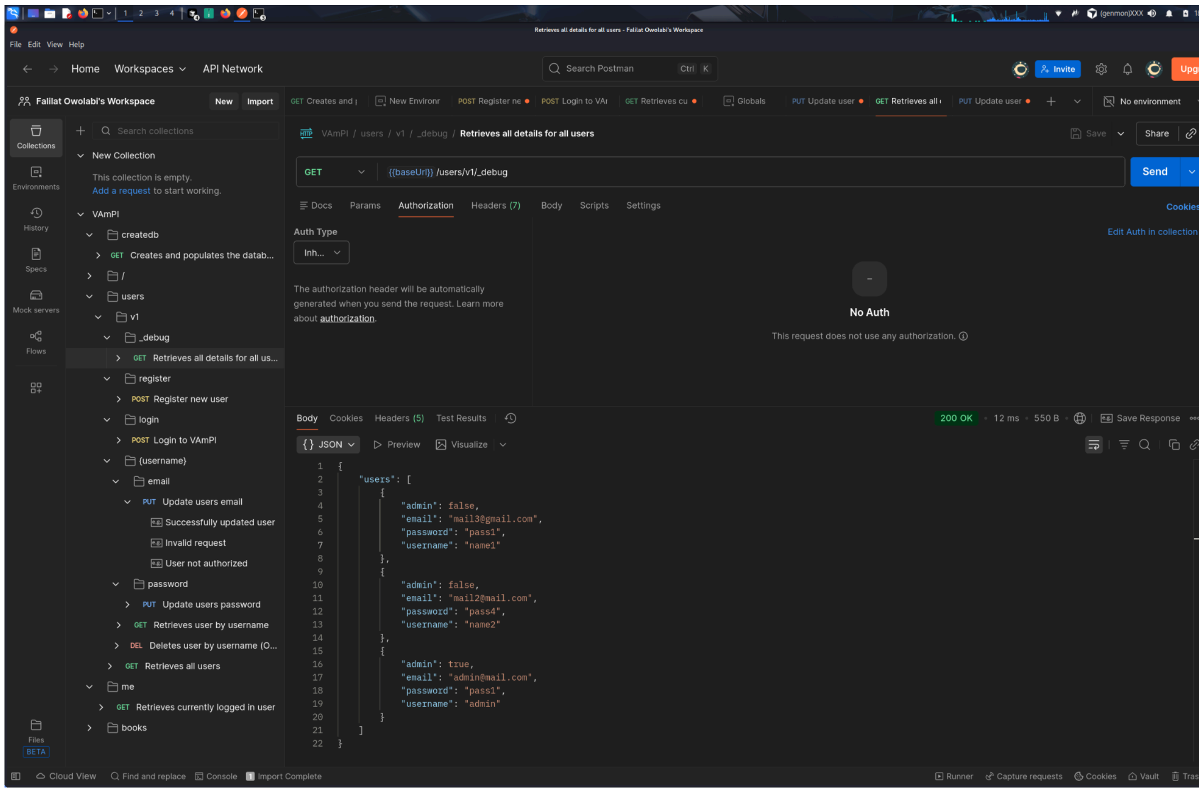
1. Implement ownership verification: verify authenticated_user.username == requested_username.
2. Require current password: for security, require current password before allowing change.
3. Add authorization check: validate JWT token subject matches path parameter.
4. Add proper response codes: include 403 Forbidden for unauthorized attempts.
5. Password policy: enforce strong password requirements.
6. Security monitoring: log all password change

3. Improper Assets Management: debug endpoint in production

How found

- Identified /users/v1/_debug endpoint during initial review (lines 88-119). Multiple indicators this is a debug/development endpoint: endpoint name contains "_debug" (line 88), operationId contains "debug" (line 94: api_views.users.debug), functionality exposes sensitive debug data (passwords, all user details). This represents a development/testing asset that should not exist in production API specifications.

Evidence / PoC:



VAmPI / users / v1 / _debug / Retrieves all details for all users

GET {{baseUrl}} /users/v1/_debug

Auth Type: Inh...

The authorization header will be automatically generated when you send the request. Learn more about authorization.

No Auth

This request does not use any authorization.

Body | Cookies | Headers (5) | Test Results — 200 OK • 12 ms • 550 B

```
{
    "users": [
        {
            "admin": false,
            "email": "mail3@gmail.com",
            "password": "pass1",
            "username": "name1"
        },
        {
            "admin": false,
            "email": "mail2@mail.com",
            "password": "pass4",
            "username": "name2"
        },
        {
            "admin": true,
            "email": "admin@mail.com",
            "password": "pass1",
            "username": "admin"
        }
    ]
}
```

Impact: Debug endpoints in production expose internal system information, development artifacts, and sensitive data. They often have weaker security controls and are intended for development/testing only. Their presence in production indicates poor asset management and increases attack surface significantly.

OWASP mapping: API9: Improper Assets Management.

Remediation:

1. Remove from production: immediately remove debug endpoints from production API specifications and implementations.
2. Separate environments: maintain separate API specifications for development, staging, and production.
3. Environment validation: implement checks to ensure debug endpoints disabled in production.
4. API versioning: use separate versions or paths for development vs production.
5. Configuration management: use environment-based configuration to disable debug endpoints.

Verify fix: GET request to /users/v1/_debug should return 404 Not Found in production. Endpoint should not exist in production API specification.

Challenges & Tester’s reflection (chronological narrative)

Being new to API security testing and Postman, I initially spent time learning the interface and how to construct requests, configure authentication headers, and manage environment variables.

I also needed to understand OpenAPI YAML structure—indentation, path definitions, security schemes, and request/response schemas. This learning curve delayed the start of the analysis.

Note on evidence: screenshots in this report are unedited. I intended to crop and annotate, but could not install suitable image editing tools on the Kali environment within the submission timeline. To meet the deadline, they were included as captured. Future reports will use proper editing tools for clearer, annotated evidence.

Despite these challenges, the assessment provided valuable experience in API security testing methodology and OpenAPI specification review

Conclusion & Final Action Plan

Overall posture: This report reveals three critical security weaknesses that align with the OWASP API Security Top 10. These represent fundamental flaws in authentication, authorization, and asset management. In production, these issues would pose significant risk.

The API shows:

Complete authentication bypass through exposed debug endpoints

Critical authorization failures allowing unauthorized password changes

Poor asset management with development endpoints in production specifications

While the specification defines authentication mechanisms (bearerAuth security scheme), they are inconsistently applied. The debug endpoint completely bypasses authentication, and the password modification endpoint lacks proper authorization controls despite requiring authentication.

Top 5 Immediate Actions (Executive Checklist):

Remove debug endpoint immediately: Remove /users/v1/_debug from production API specification and implementation. Force password rotation for all users due to credential exposure. Verify with GET request—should return 404 Not Found.

Implement object-level authorization: Add ownership verification on /users/v1/{username}/password. Ensure authenticated users can only change their own passwords. Verify by attempting to change another user's password—should return 403 Forbidden.

Fix authentication enforcement: Apply the defined bearerAuth security scheme consistently across all sensitive endpoints. The debug endpoint should never be accessible without authentication in any environment.

Separate development from production: Implement environment-specific API specifications. Debug endpoints must not exist in production specifications or configurations. Add automated checks to prevent debug endpoints in production builds.

Add authorization documentation: Update the OpenAPI specification to document authorization requirements for password modification endpoints. Add 403 Forbidden responses to indicate when users are authenticated but not authorized.

The specification review methodology effectively identified design-level vulnerabilities that would likely manifest in production implementations. Regular security reviews of API specifications should be integrated into the development lifecycle to catch these issues before deployment.

Priority ranking:

Critical (Immediate): Finding #1 and #3 (debug endpoint exposure) — requires immediate removal and credential rotation

Critical (Urgent): Finding #2 (unauthorized password modification) — requires authorization implementation to prevent account takeover

These three findings, if left unaddressed, could lead to complete system compromise, account takeover attacks, and unauthorized administrative access.

End of Report.