



OWAPS API TOP10 Workshop @AppSecIL 2023

May 2023

Lab Instructions & Setup.....	2
Lab 01: Recon.....	4
Lab 02: Authenticated Recon.....	7
Lab 03: Broken Object Level Authorization (BOLA)	9
Lab 04: Broken Authentication	12
Lab 05: Excessive Data Exposure	15
Lab 06: Lack of Resources and Rate Limiting	17
Lab 07: Broken Function Level Authorization	18
Lab 08: Mass Assignment.....	20
Lab 09: Security Misconfiguration	23
Lab 10: Injection.....	25
Lab 11: Improper Assets Management.....	27
Lab 12: Insufficient Logging and Monitoring	29

Lab Instructions & Setup

To set up the lab, you will need to comply with the following requirements:

- Windows / Linux machine with administrator or root privileges
- At least 500MB of free space available (for the container)
- An account in Docker Hub (hub.docker.com)
- A DockerHub GUI utility installed on your machine
- OWASP Zap Proxy Installed

Please keep notice of the following:

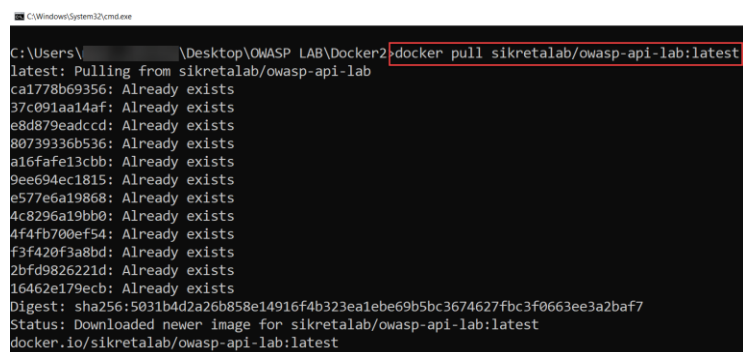
- During the class and the lab manual might use a different container version that is being installed on your local machine. We highly recommend that you keep the container up-to-date before starting the exercises.
- No need to execute denial of service or disruptive attacks on the application. Focus on the application area and not the infrastructure.
- Unless specified otherwise, some exercises have several ways to solve and sometimes even more than one option to resolve. Keep that in mind during the study process.
- The class is based on OWASP API TOP10 2019 to easy for new students in API security 😊

SETUP THE LAB:

1. Open an account / Login in DockerHub: <https://hub.docker.com>
2. Install the DockerHub client for your operating system as described here: <https://docs.docker.com/engine/install/>

NOTICE: This step may require local administrator/root privileges.

3. On your student machine, run the following command:
`docker pull sikretalab/owasp-api-lab:latest`



```
C:\Windows\System32\cmd.exe
C:\Users\...\Desktop\OWASP LAB\Docker2>docker pull sikretalab/owasp-api-lab:latest
latest: Pulling from sikretalab/owasp-api-lab
ca1778b69356: Already exists
37c091aa14af: Already exists
e8d879eadccd: Already exists
80739336b536: Already exists
a16fafa13cbb: Already exists
9ee694ec1815: Already exists
e577e6a19868: Already exists
4c8296a19bb0: Already exists
4f4fb70ef54: Already exists
f3f420f3a8bd: Already exists
2bfd9826221d: Already exists
16462e179ecb: Already exists
Digest: sha256:5031b4d2a26b858e14916f4b323ea1ebe69b5bc3674627fbc3f0663ee3a2baf7
Status: Downloaded newer image for sikretalab/owasp-api-lab:latest
docker.io/sikretalab/owasp-api-lab:latest
```

NOTE: Our machine already has the image but it is being updated instead.

- Wait for the completion of pulling the container.
- Run the following command to launch the container, loading all test data and the success message:

```
docker run -p 3000:3000 sikretalab/owasp-api-lab
```

```
C:\Windows\System32\cmd.exe - docker run -p 3000:3000 sikretalab/owasp-api-lab
C:\Users\...\Desktop\OWASP LAB\Docker2>docker run -p 3000:3000 sikretalab/owasp-api-lab
* Starting database mongodb
...done.
2023-04-24T21:04:44.433+0000    connected to: 0.0.0.0
2023-04-24T21:04:44.497+0000    imported 13 documents
2023-04-24T21:04:44.511+0000    connected to: 0.0.0.0
2023-04-24T21:04:44.528+0000    imported 4 documents
2023-04-24T21:04:44.540+0000    connected to: 0.0.0.0
2023-04-24T21:04:44.550+0000    imported 1 document
2023-04-24T21:04:44.559+0000    connected to: 0.0.0.0
2023-04-24T21:04:44.568+0000    imported 7 documents

> vulnspalab@1.0.0 start
> nodemon --quiet index.js

Starting lab...
Application listening in http://localhost:3000
Loading DB...
Database Connected! Enjoy!
```

- Go to your browser to <http://localhost:3000>. You should see the application up and running.

Troubleshooting:

- “docker daemon is not running.”

```
C:\Users\...>docker pull sikretalab/owasp-api-lab:latest
error during connect: This error may indicate that the docker daemon is not running.: Pos
er_engine/v1.24/images/create?fromImage=sikretalab%2Fowasp-api-lab&tag=latest": open //.
cannot find the file specified.
```

Solution:

Rerun the docker utility / GUI as administrator or root account.

- “Docker Desktop requires a newer WSL kernel version.”



Solution:

Install and download the Linux Kernel update package (WSL2) from the Microsoft website for your operating system, then restart the Docker GUI.

Lab 01: Recon

Scenario:

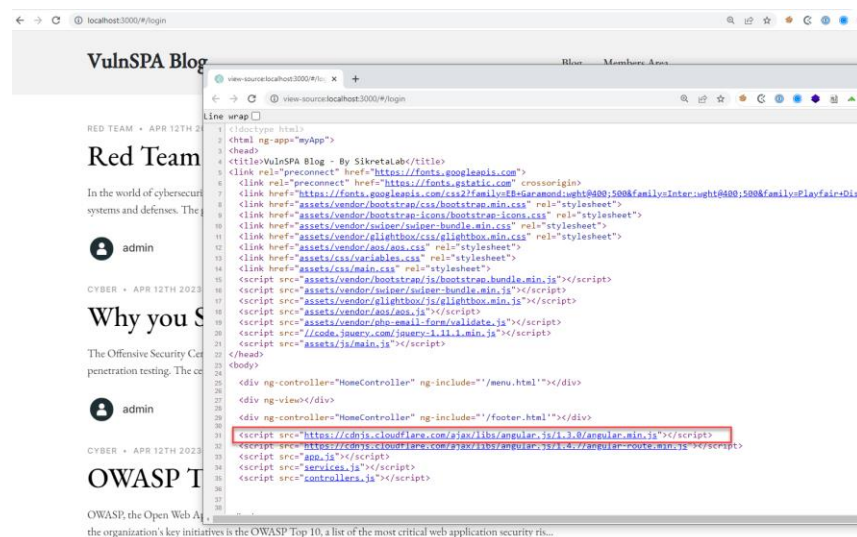
You must find information about the API, such as the technology stack and ,documentation to help you find remote endpoints or extend the attack surface.

Goals:

- Find the technology used in the VulnSPA application
- Can you find any supported documentation which allows you to extend the attack surface?

Walkthrough:

In our web application, we will use F12 in the browser. Right-click -> “View source” to see any libraries the application use. In our example, we can see that application implements a JavaScript library called AngularJS:



What is AngularJS? AngularJS is a framework for building Single Page Applications (SPA) using HTML and JavaScript. The application is based on MVC architecture which divides the app into three layers:

- Model – The way application manages data (the API implementations)
- View – The GUI (“user interface”) you see (for example, the blog design)
- Controller – The link between service to view (our exciting part)

As the app is written in JavaScript, we can read the Controller/View/Model source code to understand the logic, and any hidden gems (such as passwords, API endpoints, etc.) it would make our life much easier in our task. We do not dive into the technicalities of AngularJS (or any other JS framework, for that matter) . We need to understand the basic concept for our testing.

Now we continue looking for more ways to learn about the API endpoints.

There are many ways to enumerate possible API endpoints from GitHub repositories leak, Pastebin, and Google Dork. But we will focus on two forms – a trial and error technique or by enumeration of potential API documentation.

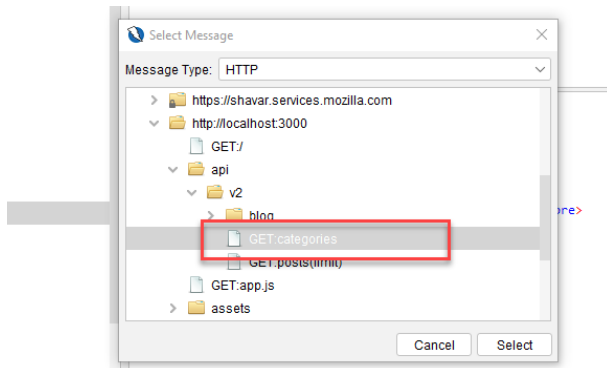
In the first method, we will use Zap Proxy to send an extensive crafted list of possible API endpoints, analyze their response, and look for remote endpoints. Alternatively, we may enumerate standard API endpoints to see if the application exposed any sensitive development documentation of application API endpoints.

You can use our list from here:

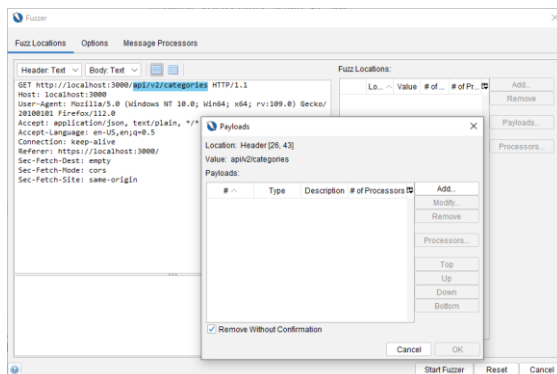
<https://github.com/SikretaLabs/workshops/tree/main/AppSecL2023/wordlists/endpoints.txt>

STEPS:

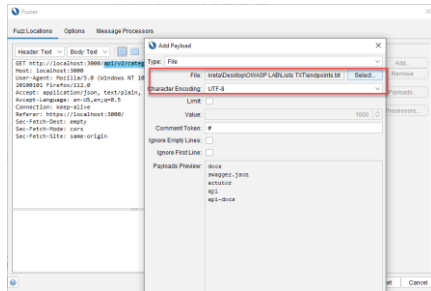
1. Open Zap Proxy -> Open the application using “Manual Inspection.”
2. Go to Tools -> Fuzz -> Choose the localhost site and click on some API endpoint:



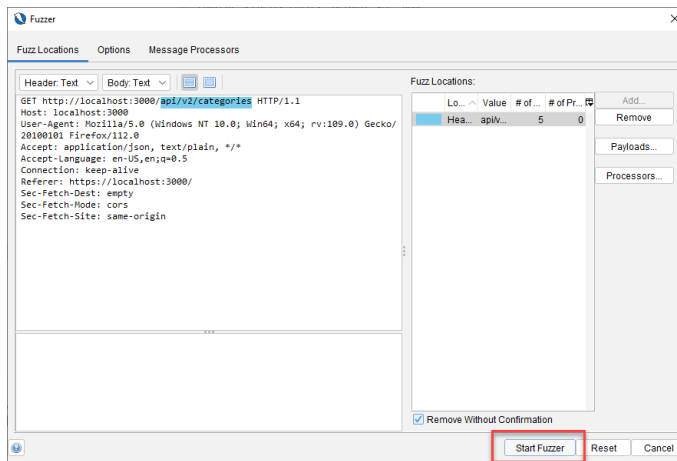
3. After clicking “Select,” then mark the “api/v2/categories” so we can start fuzzing for the top-level path as shown below, then click “Add..” from the left side menu:



4. Click on “Add..” -> Type: File -> click on “Select..” to load our file and click “Add” below:



5. Finally, click on “Start Fuzzer”:



You should find the remote API documentation endpoint by now.

Lab 02: Authenticated Recon

Scenario:

As the application allows external users or guests to sign up. We should identify how the authorization works and if there is any exciting endpoint we need to focus on.

Goals:

- Find what type of authentication & authorization the application uses
- What technology application is used to store the user's session?

Walkthrough:

In our web application, we can create an account quickly. We can sign up using the link below the form by navigating the member's area page. For the workshop, we can open a user account with the following details:

- Username: test
- Email: test@local.lab <or any email you would like>
- Password: test

Upon successful signup, we will be able to login into our account:

My Profile

Username	Email	Subscribers	Level
test	test@gmail.com	0	user

My Billing Profile

No Billing Information updated. [Click here](#) to update.



Let us inspect the requests inside Zap Proxy. We can identify that the API requests are using JSON Web Token (JWT) technology for authentication & authorization as the user's JWT token is sent over Authorization Header as shown below:

```
Header: Text Body: Text
GET http://localhost:3000/api/v2/me/profile HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/112.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiI2NDQ3YjE4YVh0bGVhbnQyZDNIYTciLCJ1c2VybmFtZSI6IHR5cGU6ImV4cCI6MTcxNDM0MTg3OH0.d0FdzWYhZ3F5LN8aNP3FT3iIN53qTqB1-uN5tEU_gU
Connection: keep-alive
Referer: https://localhost:3000/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
```

The browser should store this token as the server has generated it somehow. Back to the browser and clicking F12, we can see (in Chrome) that our application utilizes HTML5 Local Storage to keep the token in the user's browser:

SPA Blog

My Profile

Username	Email	Subscribers	Level
test	test@gmail.com	0	user

My Billing Profile

No Billing Information updated. [Click here](#) to update.

Change PasswordUpdate ProfileLogout

Application

Filter

Key

Value

uid

6447b18a0f5405c5d3d3ee7

token

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiI2NDQ3YjE4YVh0bGVhbnQyZDNIYTciLCJ1c2VybmFtZSI6IHR5cGU6ImV4cCI6MTcxNDM0MTg3OH0.d0FdzWYhZ3F5LN8aNP3FT3iIN53qTqB1-uN5tEU_gU

A screenshot of a web browser's developer tools window, specifically the Network tab. The top pane shows headers for a GET request to http://localhost:3000/api/v2/me/profile/card/6447b18aa065405c5d2d3ea7 HTTP/1.1. The second pane displays various header values including Host, User-Agent, Accept, Accept-Language, Authorization (Bearer token), Connection, Referer, Sec-Fetch-Dest, Sec-Fetch-Mode, and Sec-Fetch-Site.

Header

GET http://localhost:3000/api/v2/me/profile/card/6447b18aa065405c5d2d3ea7 HTTP/1.1

Host: localhost:3000

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/112.0

Accept: application/json, text/plain, */*

Accept-Language: en-US,en;q=0.5

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiJlNDQ3YjE4YWwWJmVudGVzZWZDNDYxOTk1LjC1c3VybmFtZSI6ImRlc3Q1LjC3ZXZlbiBicnVzXiIjLCJpcyYXQioJe2ODI4NzYzMDOqImV4CC16MTcxNDQxMjMwNH0.qLnQSJ6sMQEI0DV1vvgpAC0BJxN46GvyIA8j_gH3o9g

Connection: keep-alive

Referer: https://localhost:3000/

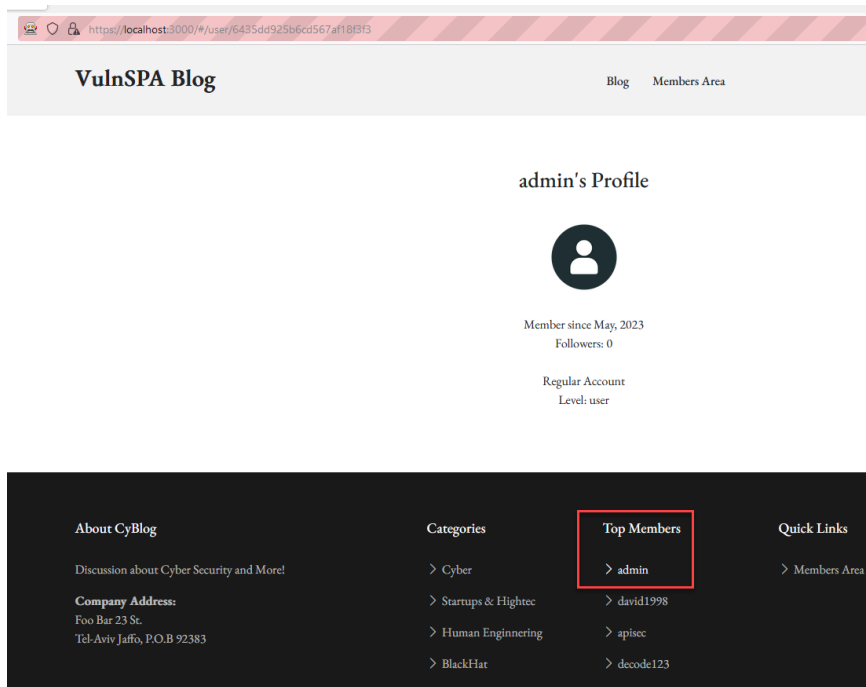
Sec-Fetch-Dest: empty

Sec-Fetch-Mode: cors

Sec-Fetch-Site: same-origin

So how can we generate other users' IDs?

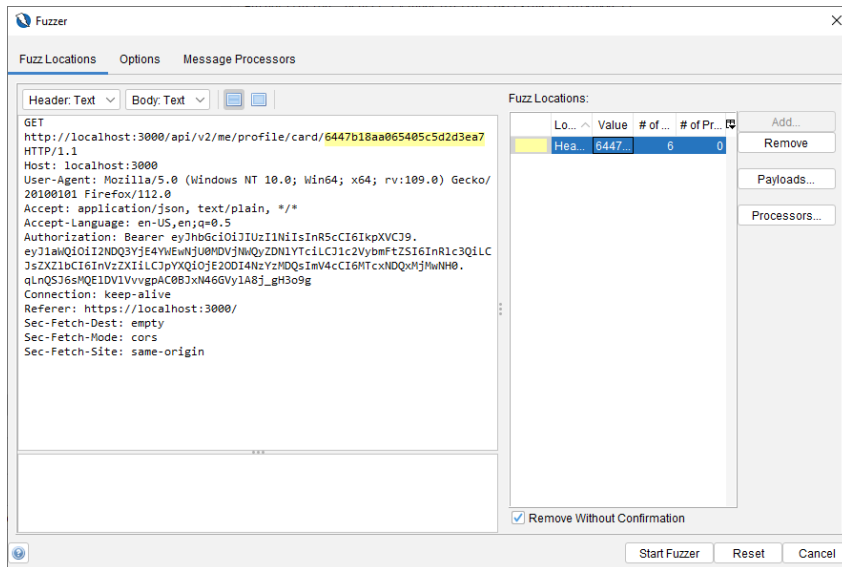
Many techniques may involve cryptographic implementations, but sometimes the solution is simple. In our case, we can find that we can fetch other users' IDs by navigating the main page:



Now it may seem tedious to go over each one manually, but in the end, it will create a decent list for fuzzing:

```
users.txt - Notepad
File Edit Format View Help
6435dd925b6cd567af18f3f3
6435e0575b6cd567af18f448
6435e06c5b6cd567af18f44b
6435e0885b6cd567af18f44e
6435e0955b6cd567af18f451
6435e0a15b6cd567af18f454
```

Based on the 'profile' request we found earlier, we can use the Zap Fuzzer again but on the user UID this time, as shown below:



Starting the Fuzzer, we should find another user credit card.

Lab 04: Broken Authentication

Scenario:

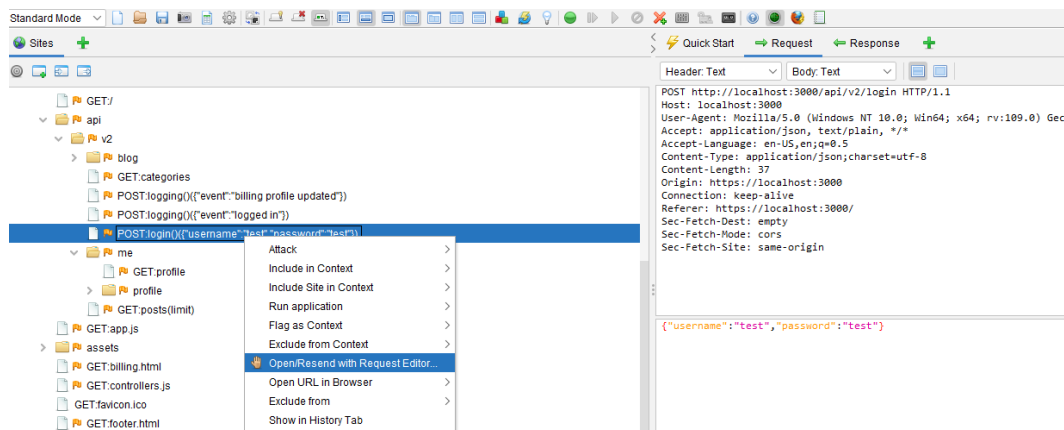
We need to find a way for non-authenticated users to assume other users' accounts using brute force attacks and analyze the application authentication login logical flow.

Goals:

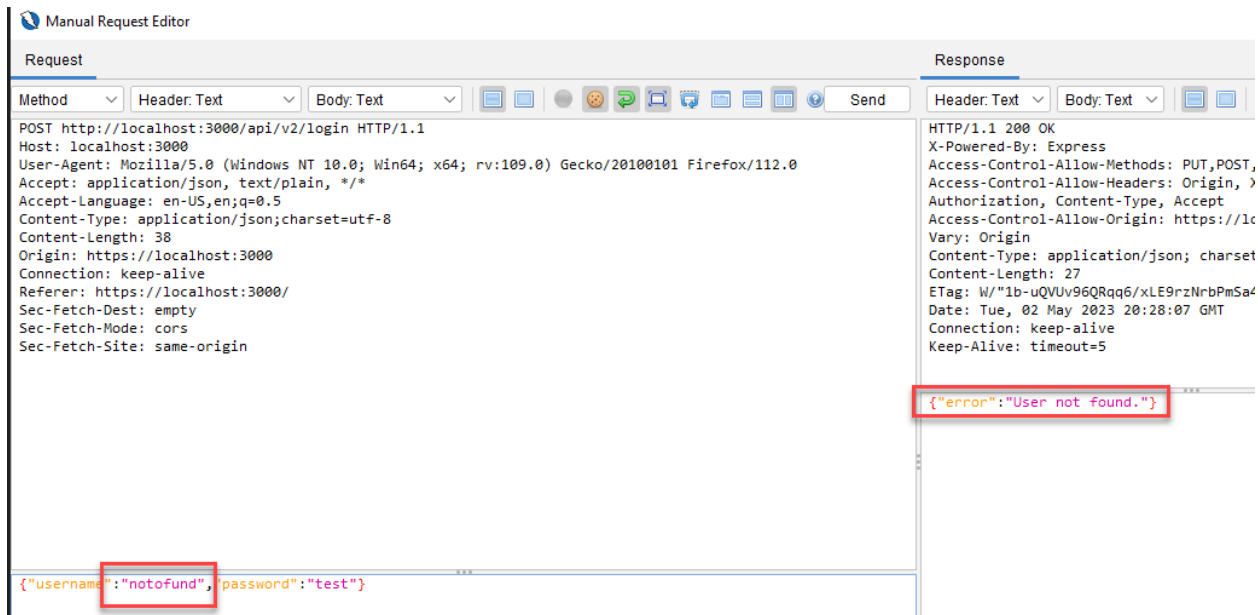
- Can you find a logical flow that makes authentication susceptible to brute force attacks?
- Can you abuse such flow to find credentials for other users?

Walkthrough:

Log out from our account; we can inspect the login flow by providing invalid credentials in the login API endpoint. Right-click on the login API method, then “Open/Resend with Request Editor...”:

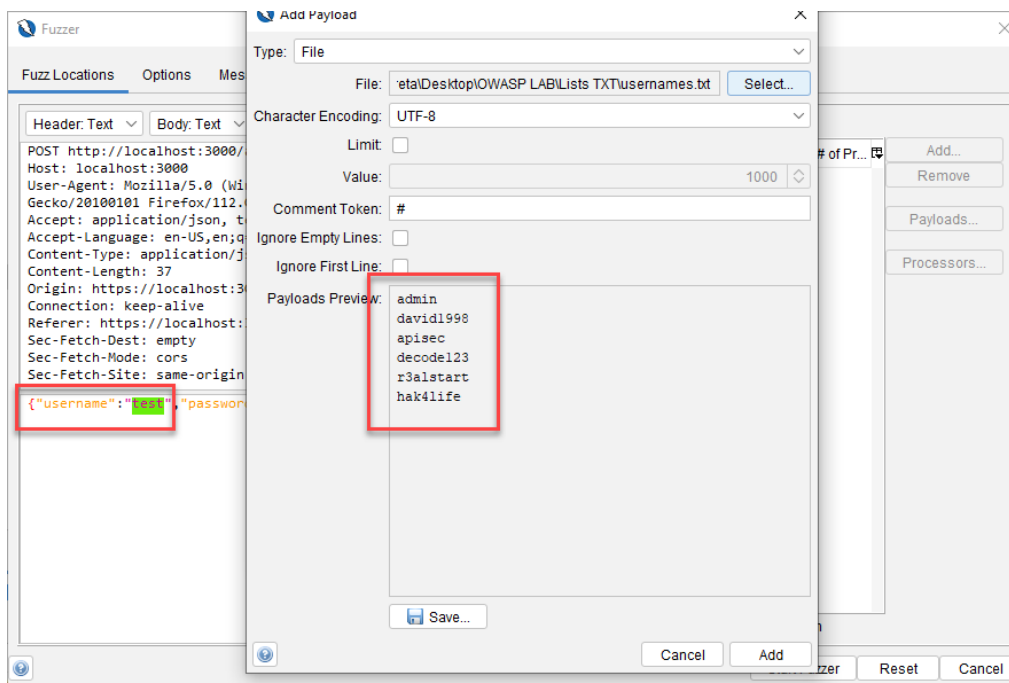


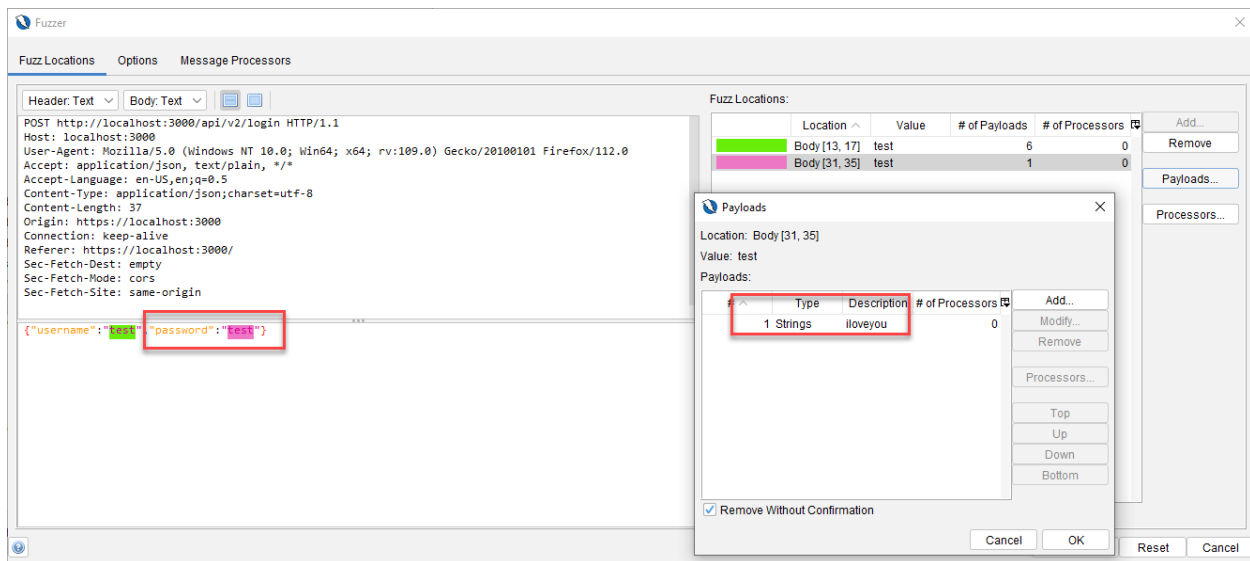
On the new windows, modify the username to a fake one (i.e., not found) and click submit:



In our case, the server returned a message disclosing the non-existence of the requested account in the database; therefore, we could guess other users' passwords based on the error message.

For that purpose, let us use the Fuzzer tool to see if we can find any user with a weak password (i.e., 123456, iloveyou, etc.). Right-click on the login API method, then "Fuzz" to place fuzz location as username with pre-defined password:





Now click on the “Start Fuzzer” button and see if you can find the weak user:)

Lab 05: Excessive Data Exposure

Scenario:

The application is used to display some information to users. But as we know, the API response may filter out some value to the client. As a hacker, we need to see if we can detect engaging API endpoints that may expose us to the juicy stuff.

Goals:

- Find any API endpoint which may return excessive information but not display it on our web application GUI.

Walkthrough:

As vulnerabilities can mostly find on authenticated API endpoints. Therefore, we need to log in with our account to test it.

By navigating the member's area page, we can see several API calls being triggered:

Id	Source	Req. Timestamp	Method	URL	Code	Reason
503	Proxy	4/30/23, 9:31:14 PM	GET	http://localhost:3000/api/v2/posts?limit=5	200	OK
502	Proxy	4/30/23, 9:31:14 PM	GET	http://localhost:3000/api/v2/blog/users	200	OK
501	Proxy	4/30/23, 9:31:14 PM	GET	http://localhost:3000/api/v2/categories	200	OK
506	Proxy	4/30/23, 9:31:22 PM	GET	http://localhost:3000/api/v2/blog/users	304	Not Modified
504	Proxy	4/30/23, 9:31:22 PM	GET	http://localhost:3000/api/v2/categories	304	Not Modified
505	Proxy	4/30/23, 9:31:22 PM	GET	http://localhost:3000/api/v2/posts?limit=5	304	Not Modified
507	Proxy	4/30/23, 9:31:26 PM	POST	http://localhost:3000/api/v2/login	200	OK
508	Proxy	4/30/23, 9:31:26 PM	POST	http://localhost:3000/api/v2/logging	200	OK
510	Proxy	4/30/23, 9:31:26 PM	GET	http://localhost:3000/api/v2/me/profile	200	OK
509	Proxy	4/30/23, 9:31:26 PM	GET	http://localhost:3000/profile.html	200	OK
512	Proxy	4/30/23, 9:31:27 PM	GET	http://localhost:3000/api/v2/blog/users	304	Not Modified
511	Proxy	4/30/23, 9:31:26 PM	GET	http://localhost:3000/api/v2/me/profile/card/6447b18aa06...	304	Not Modified
513	Proxy	4/30/23, 9:31:27 PM	GET	http://localhost:3000/api/v2/posts?limit=5	304	Not Modified
514	Proxy	4/30/23, 9:31:27 PM	GET	http://localhost:3000/api/v2/categories	304	Not Modified

If we look at our profile page, we can see that the Card Number value is masked:

My Profile

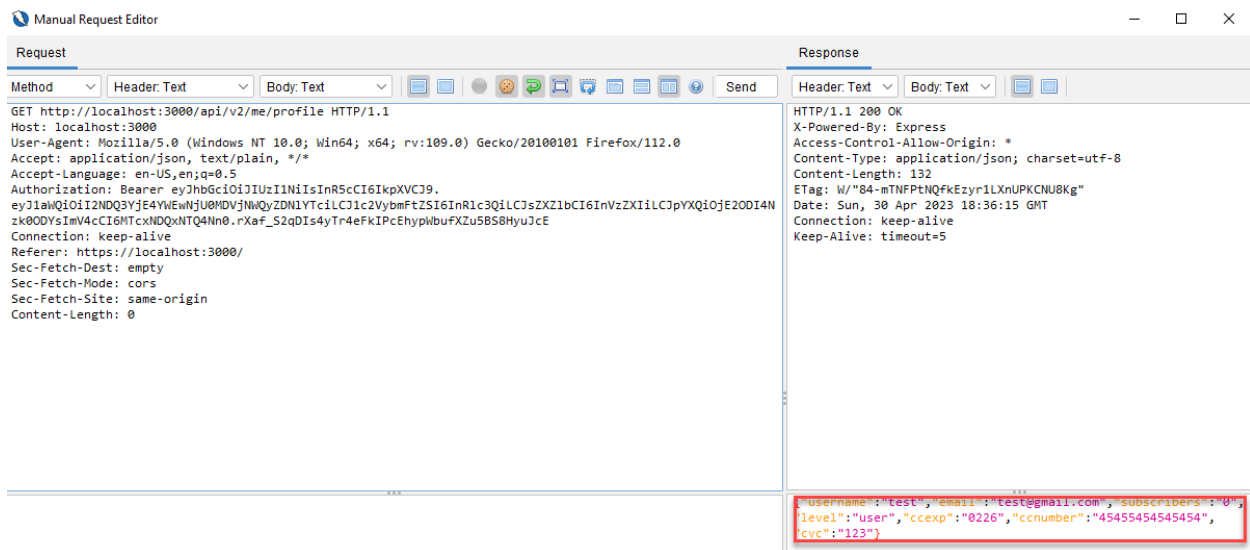
Username	Email	Subscribers	Level
test	test@gmail.com	0	user

My Billing Profile

Card Number	Exp. Date	Issuer
****5454	0226	CreditCompany Ltd.

[Change Password](#) [Update Profile](#) [Logout](#)

We may discover the total number if the client is masking the data. Checking the profile API endpoint, we can see that all credit card values are returned as part of the API response:



Now we found our excessive API endpoint.

Later in class, we would see how this vulnerability can be found elsewhere.

Lab 06: Lack of Resources and Rate Limiting

Scenario:

The application uses specific process resources to retrieve data from the database in our application. Some of these operations may be memory or CPU consume more resources. The developers claimed they already cared about most of the application's error and memory handling. You will need to see if any hidden or existing API endpoint can be abused to consume some resources.

Goals:

- Find an API endpoint running some operation against the database, abuse it, and get the unique flag without crashing the app.

Walkthrough:

CRUD (Create, Read, Update, Delete) operations are used in many web applications. Usually, this operation communicates with the database and might consume some resources (i.e., execute a query to fetch data).

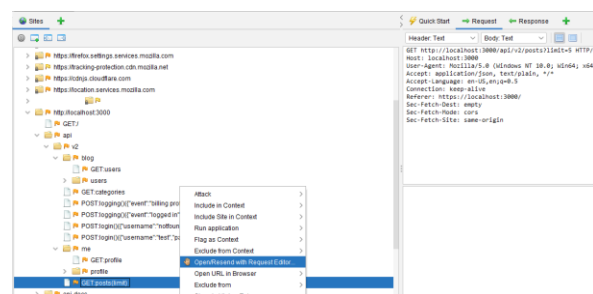
An everyday thing that consumes resources is a large SQL or database query to fetch such data. If we look at our API endpoints again, we can see that Post API has a limitation of 5 records to fetch:

Id	Source	Req. Timestamp	Method	URL	Code	Reason	RTT	Size Resp
502	Proxy	4/30/23, 9:31:14 PM	GET	http://localhost:3000/api/v2/blog/users	200	OK	63 ms	2,199 bytes
501	Proxy	4/30/23, 9:31:14 PM	GET	http://localhost:3000/api/v2/categories	200	OK	33 ms	400 bytes
506	Proxy	4/30/23, 9:31:22 PM	GET	http://localhost:3000/api/v2/blog/users	304	Not Modified	36 ms	0 bytes
504	Proxy	4/30/23, 9:31:22 PM	GET	http://localhost:3000/api/v2/categories	304	Not Modified	35 ms	0 bytes
505	Proxy	4/30/23, 9:31:22 PM	GET	http://localhost:3000/api/v2/posts?limit=5	304	Not Modified	48 ms	0 bytes
507	Proxy	4/30/23, 9:31:26 PM	POST	http://localhost:3000/api/v2/login	200	OK	20 ms	280 bytes
508	Proxy	4/30/23, 9:31:26 PM	POST	http://localhost:3000/api/v2/logging	200	OK	25 ms	26 bytes

Which may look like that in SQL:

```
SELECT * FROM `post_table` LIMIT 5
```

Theoretically, forcing the database to fetch more than five records may lead the database to consume memory or CPU resources in the background. Right-click on the login API me, then then “Open/Resend with Request Editor...”:



All left is to fetch more resources than five and get the flag.

Lab 07: Broken Function Level Authorization

Scenario:

The application uses both user-level and admin-level API endpoints. As attackers, we need to find such endpoints and try to invoke them, whether by finding exposed API documentation or manual inspection.

Goals:

- Can you find any administrative API endpoints that can be abused?
- Can you find a way to remove content as a non-privileged account?

Walkthrough:

As we found the API documentation, we already have a lead for the first use case. If we investigate the Swagger documentation, we may be able to see some admin API endpoints, as shown below:

Backoffice Only			^
POST	/api/v2/admin/categories	Add Category	▼
DELETE	/api/v2/admin/posts/comments/641f67ed125c310e475af005	Delete Post Comments	▼
DELETE	/api/v2/admin/posts/641f67ed125c310e475af005	Delete Blog Post	▼

Now we can test each one to operate some administrative operations.

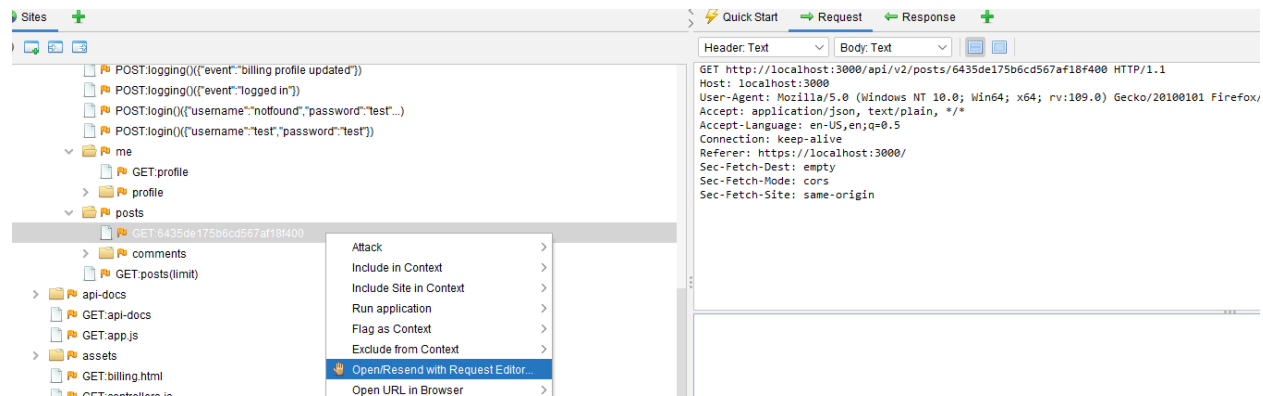
However, in (most) cases, that could be more obvious. Sometimes we need to tamper with existing API endpoints that need to be documented to perform sensitive actions (e.g., creation, modification, or erasure) that we should not have access to.

If we look carefully at API documentation, we can see many API endpoints are related to posts content (i.e., publish or comment):

Posts			^
GET	/api/v2/posts/comments/6435deb15b6cd567af18f414	Single Post Comments	▼
GET	/api/v2/categories	Categories Blog	▼
GET	/api/v2/posts	Blog Posts	▼
GET	/api/v2/posts/641feed3a0d451199495bc96	Single Post	▼
POST	/api/v2/posts/642f3e95d66bb610bd78618c/replay	Post Comment	▼
POST	/api/v2/logging	Log Event	▼
POST	/api/v2/publish	Create Post	▼

We can perform an HTTP method not listed here, such as DELETE, to remove content from posts. For example, we can use the single post API to delete the center instead of fetching it.

Right-click on the login API method, then “Open/Resend with Request Editor...”:



Modify the “GET” HTTP method to “DELETE”:

Manual Request Editor



Then we can delete such posts as non-administrators.

Lab 08: Mass Assignment

Scenario:

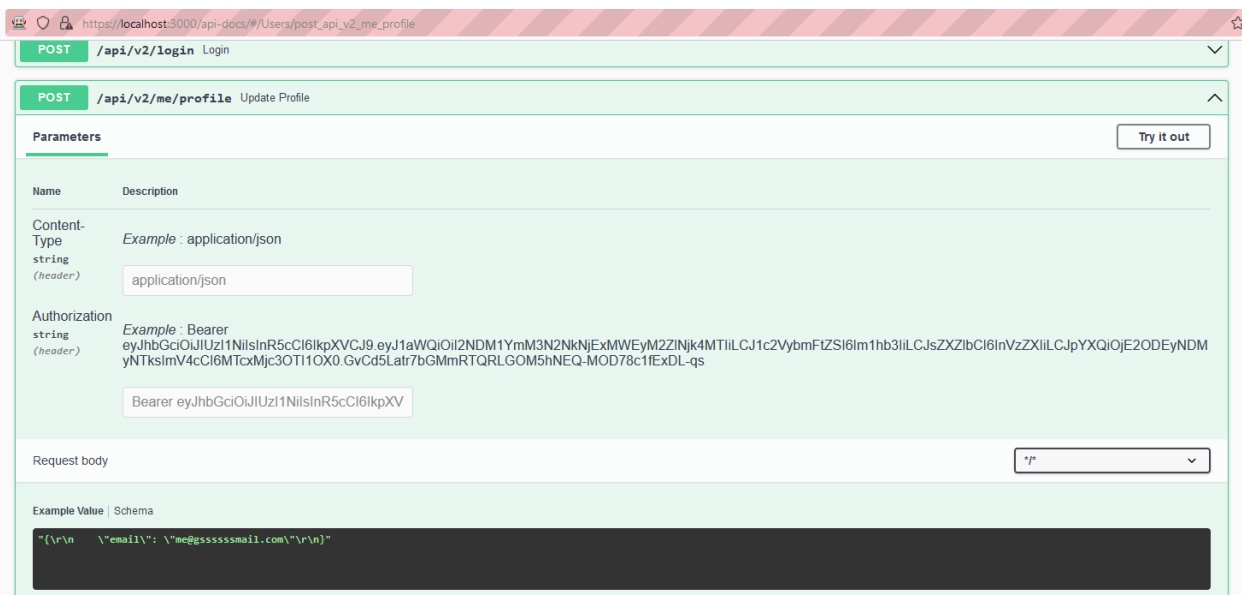
The application uses MongoDB and NodeJS technology to perform actions based on the client's input—the API under networks work by receiving any user's input as payload to execute CRUD operations. As attackers, we must abuse the incoming data as part of the application's object to modify stored information in the database.

Goals:

- Find an API endpoint that performs an update operation and abuse it to modify some Restricted information about your user.

Walkthrough:

To exploit mass assignment, we need to find an API endpoint that takes data from the client and processes it as an object without filtering the incoming properties. We may see them usually in update operations. One of the update operations is to update the user's profile information.



But to test mass assignment, we need to identify our object's properties (in our case, we assume it is called a user object). So, we have two approaches— guessing or relying on GET requests to return such object properties.

Assume we want to start effectively; we can begin to inspect the GET response:

GET

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Headers ☐ 6 hidden

Key	Value	Description
<input checked="" type="checkbox"/> Content-Type	application/json	
<input checked="" type="checkbox"/> Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiI1NDQ3YjE4YWE	
Key	Value	Description

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "username": "test",
3   "email": "test@gmail.com",
4   "subscribers": "0",
5   "level": "user",
6   "ccexp": "0226",
7   "ccnumber": "45455454545454",
8   "cvc": "123"
9 }
```

We can see there are the following properties:

- Email
- Username
- Subscribes
- Level

There is also the credit card information, but we can skip it now.

To use mass assignment, we could try to use “level” properties in our request. Right-click on the login API method, then “Open/Resend with Request Editor...”:

Sites

Quick Start Request Response

Header Text Body Text

POST http://localhost:3000/api/v2/me/profile HTTP/1.1

Host: localhost:3000

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/97.0

Accept: application/json, text/plain, */*

Accept-Language: en-US,en;q=0.5

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOiI1NDQ3YjE4YWE

Content-Type: application/json; charset=utf-8

Content-Length: 26

Origin: https://localhost:3000

Connection: keep-alive

Referer: https://localhost:3000/

Sec-Fetch-Dest: empty

Sec-Fetch-Mode: cors

Sec-Fetch-Site: same-origin

Attack

Include in Context

Include Site in Context

Run application

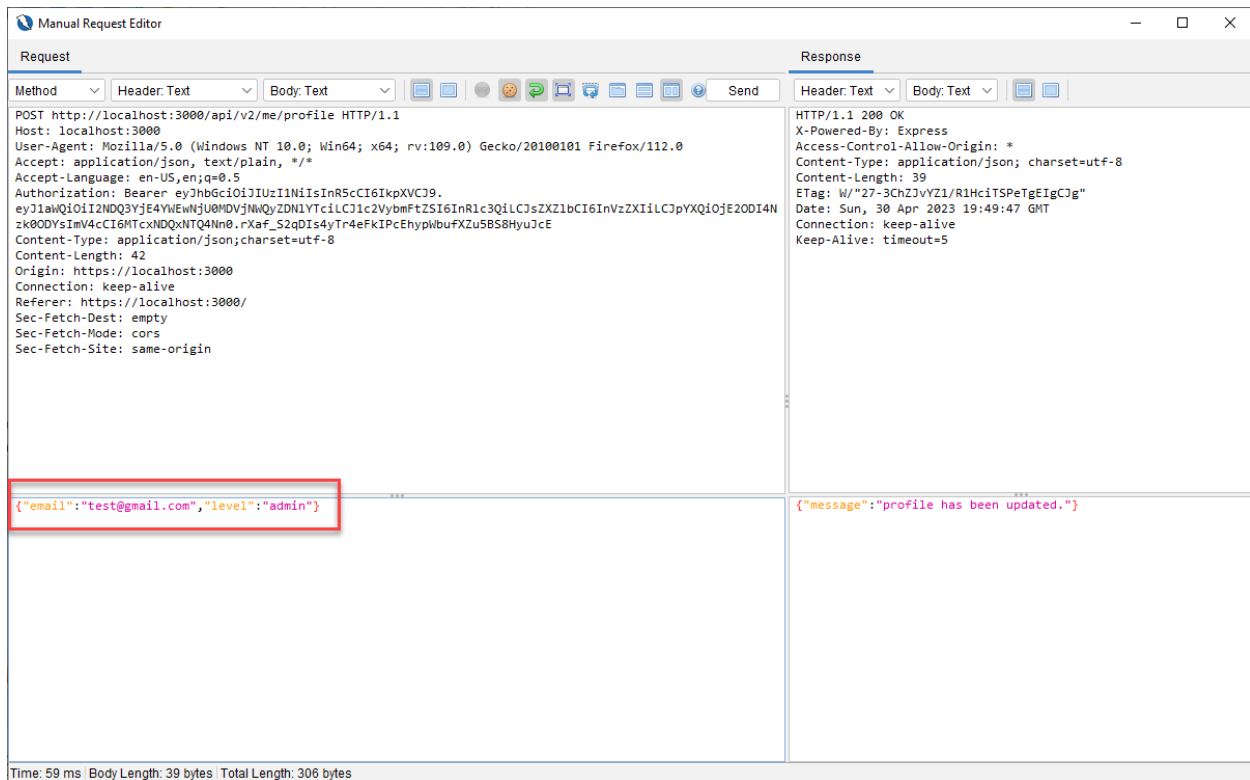
Flag as Context

Exclude from Context

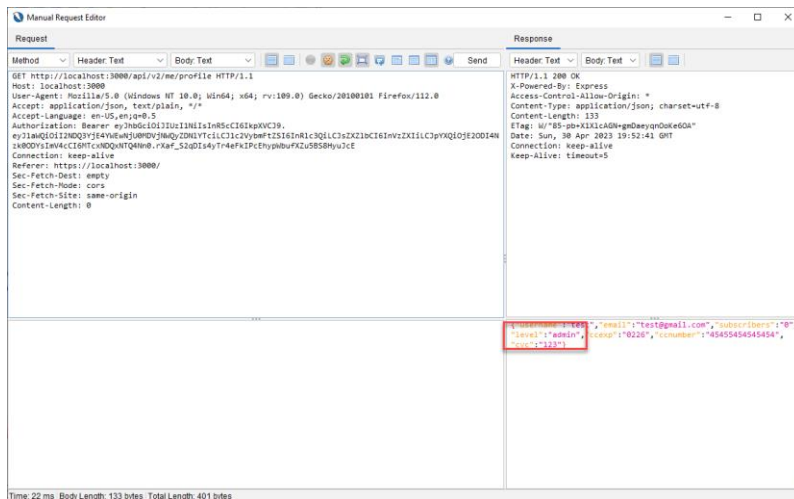
Open/Resend with Request Editor...

Open URL in Browser

Adding to the request our new property:



If we recheck the GET method, we will see that our level is being updated to the admin level, as the backend does not filter the input by request properly, allows to overview object properties in the database:



Can you do it with another property? Give it a try!

Lab 09: Security Misconfiguration

Scenario:

As in every web application, some poor or insufficient API endpoints or infrastructure configuration allows attackers to exploit them (i.e., unpatched server version or default configuration). As hackers, we need to find leads to locate any misconfiguration that may lead to application exposure.

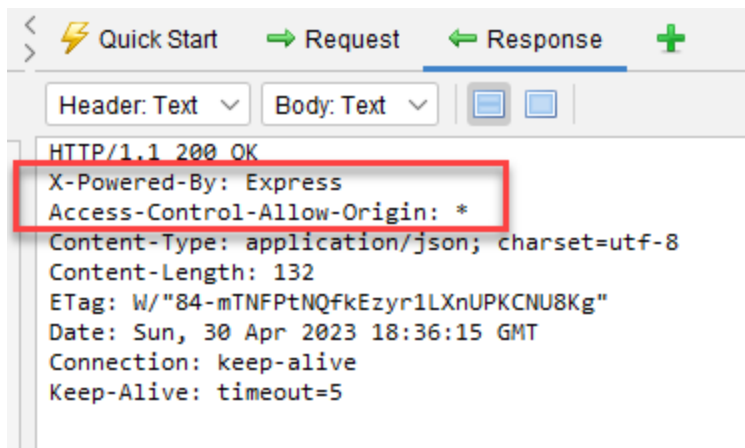
Goals:

- Find a very common misconfiguration about the server
- Try to see if you can find any missing configuration in the API webserver which allows you to execute remote API calls

Walkthrough:

First, to look up any misconfiguration of the application webserver, we could focus on Low Hanging Fruit (LHF) by analyzing the webserver response headers. Though many methods exist to abuse this vulnerability (i.e., error messages, exposed files, weak or default app configuration, etc.), we will focus on a simple yet effective way of examining the response headers and looking for leads.

As shown in the screenshot below, the server responds with two interesting headers:



First, we can identify the server technology stack via the “Server” header – ExpressJS. From our perspective, we can google it or use tools such as Synk to identify potential vulnerabilities. However, for practice’s sake, we do not need to execute ZeroDay or exploit. But you can if you wish 😊

The 2nd header uses the form Cross-Origin-Resource-Share (CORS) policy which is very common in such SPA application as the API web server need to allow JavaScript application to communicate with the server.

One of the common misconfigurations in CORS Policy is known as “CORS Origin Reflected,” which means, under some conditions, enabling controlled access to resources located outside of a given domain (i.e., Origin). We can test it: By sending a request from an untrusted or external domain via the Origin header, the server will reflect our part in the Access-Control-Allow-Origin response header back to us. Therefore, we can craft an exploit to request resources from the API web server.

To test it in our case, we can choose one of the API requests:



Then add the 'Origin' header to our request and get the value reflected via server response:



And finally, we abuse CORS misconfiguration! Can you see the flag?

Lab 10: Injection

Scenario:

The VulnSPA application uses a database to store, track and update information about content and user information. Every API endpoint starts from view post to log in, and the member's section is communicated with the database. You need to see if the API endpoint is prone to any database injection attack.

Goals:

- Can you find a bypass for the login API endpoint with Injection?

Walkthrough:

In our application, we can easily detect the login API endpoint:

```
POST http://localhost:3000/api/v2/login HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/112.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Content-Type: application/json;charset=utf-8
Content-Length: 38
Origin: https://localhost:3000
Connection: keep-alive
Referer: https://localhost:3000/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
```


```
{"username": "test", "password": "dssds"}
```

Now you probably wonder – “How can I detect the technology underneath? Is that MySQL, SQL Server, MongoDB, etc.?”. Usually, before a real engagement, we ask the customer for the technology stack. But if we are on black box methodology, we can use the fuzzing technique to identify the technology.



Use the list from:

<https://github.com/SikretaLabs/workshops/tree/main/AppSecL2023/wordlists/injection.txt>

Right-click on the login API method, then “Fuzz” to place two fuzz locations (username and password field):

 Fuzzer


Fuzz Locations Options Message Processors

Header: Text Body: Text  

```
POST http://localhost:3000/api/v2/login HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0)
Gecko/20100101 Firefox/112.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Content-Type: application/json;charset=utf-8
Content-Length: 38
Origin: https://localhost:3000
Connection: keep-alive
Referer: https://localhost:3000/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{"username":"test","password":"dssds"}
```

Fuzz Locations:

	Lo... ^	Value	# of ...	# of Pr...
	Body...	test	10	0
	Body...	dssds	10	0

Now run the fuzzer and see the result. Can you identify the technology behind it now?

Lab 11: Improper Assets Management

Scenario:

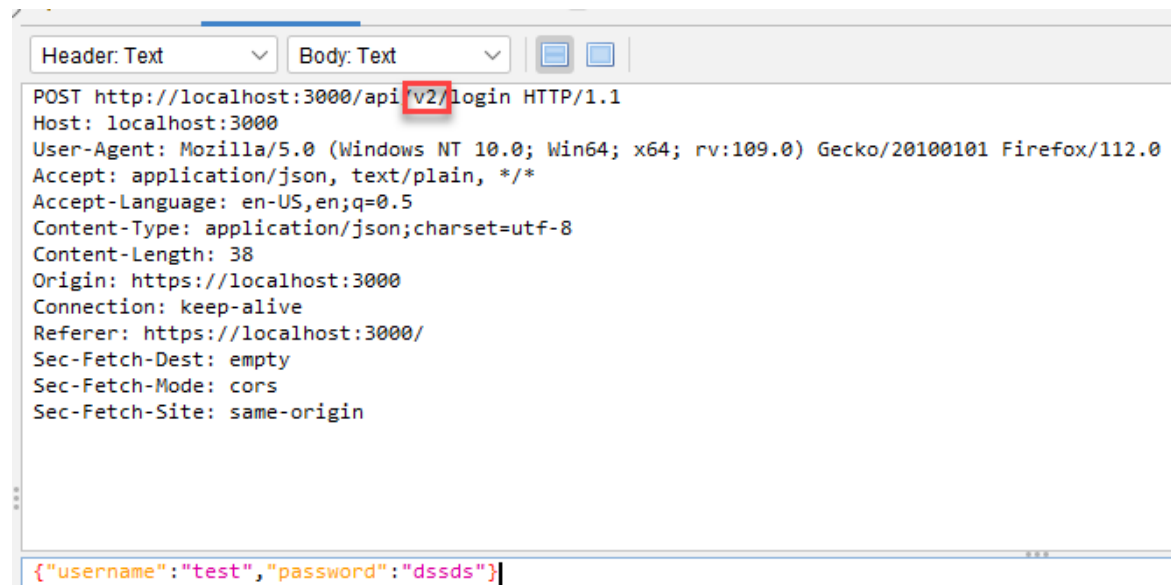
In the application's development cycle, the developers use versioning for different application development stages such as staging, beta, or earlier. Sometimes, these endpoints are not well-protected and are being deployed on production (known as "Shadow API"). You will need to ensure that developers did not make such a mistake.

Goals:

- Try to enumerate a legacy or older version of the API endpoint. Can you find a legacy API endpoint that exposed the flag?

Walkthrough:

In our API endpoints, we can see that most of the APIs are running under version 2, as shown below:



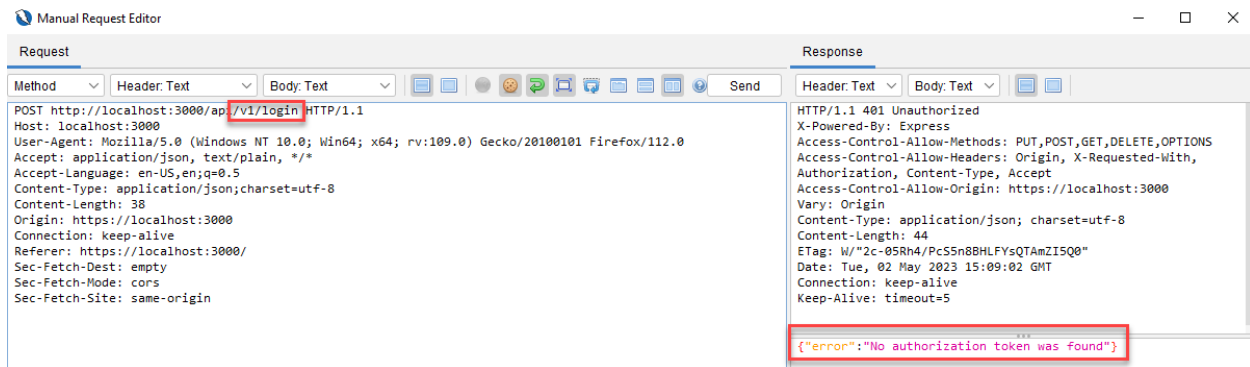
The screenshot shows a web browser's developer console with the 'Network' tab selected. A request to 'POST http://localhost:3000/api/v2/login HTTP/1.1' is highlighted. The 'Headers' pane is open, showing the request details. The URL 'api/v2/login' is highlighted with a red box. The request body is visible at the bottom of the console.

```
POST http://localhost:3000/api/v2/login HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/112.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Content-Type: application/json;charset=utf-8
Content-Length: 38
Origin: https://localhost:3000
Connection: keep-alive
Referer: https://localhost:3000/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{"username":"test","password":"dssds"}
```

To test it, we need to modify the version into an earlier one, such as v,1, and see if there is any exciting response.

For example, in login API:



Now try to do the same for authenticated API endpoints, and you should find one API that exposed you to the flag!

Lab 12: Insufficient Logging and Monitoring

Scenario:

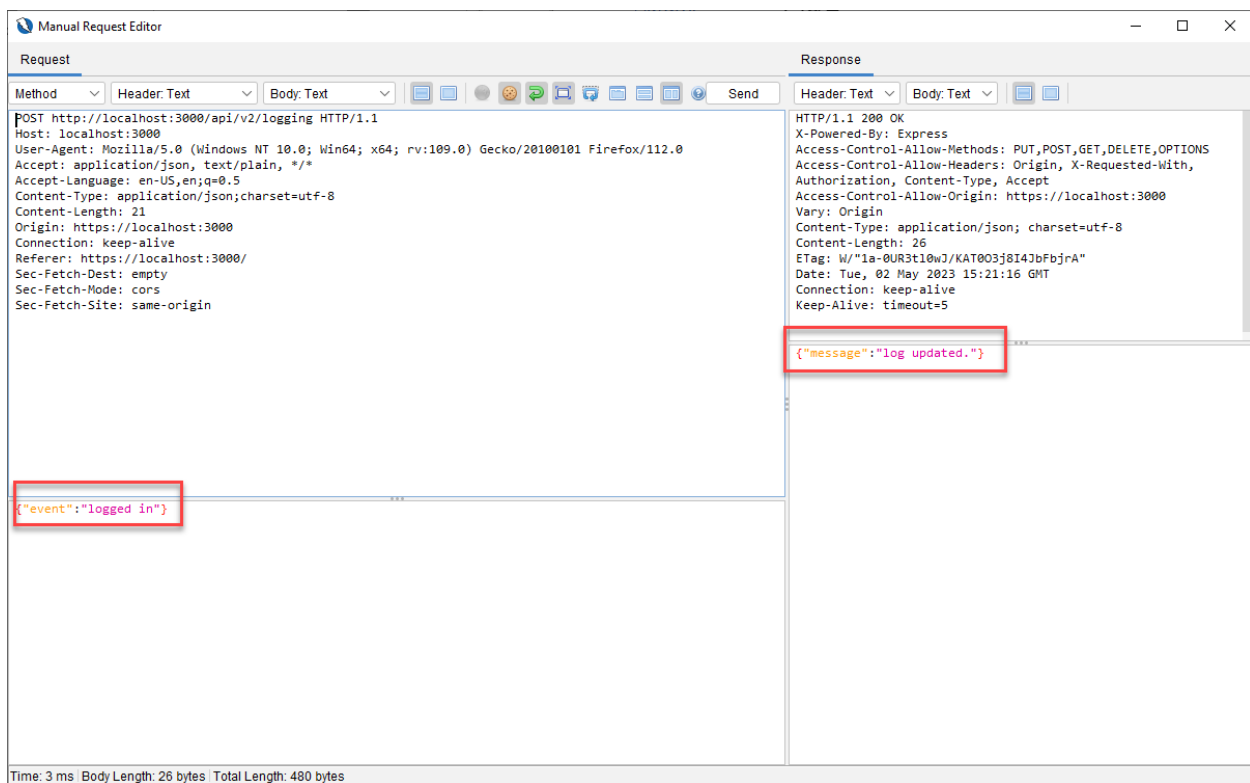
An essential part of logging and monitoring is ensuring that every log is protected for its integrity so the record will provide value in case of software failure or security incidents. It would be best to ensure the logging mechanism is resilient against abuse.

Goals:

- Find the logging API and try to make an invalid logging request to retrieve the flag

Walkthrough:

In our application, we can see there is an API endpoint for the logging data, as shown below:



To test whether the API endpoint performs a proper checking for the logs, we can modify the record into something else (i.e., AAAAAA...) to make the web server log insufficient event data and retrieve the flag!