They say when you hack, do it your way, in your style. This is my style, like an arrow, a surgeon, targeting something specific, using a scapel, not a baazooka

When hacking there are two categories of attack surface, the Visible and the Hidden. It is always best to start with the visible and then branch out into the Hidden surfaces. The visible is easily found found normal interaction with the target so we shall begin there.

**Postman is not used in this due to the annoying feature of a request body not being saved and a stable Internet connection being required.*

STAGE ONE - ARMING

Required:

Burp Suite (any edition)
Network Tab - Dev Tools
mitmproxy2swagger
JWT Editor Extension (Burp)
JQ - Json CLI
Your favourite text editor

This stage involves using the API as a normal user in order to understand how it functions and get a feel for it. It will allow for the creation of rogue documentation, which personally I prefer over the official as it is always 100% up-to-date and if you are thorough, you can hit all almost all the endpoints.

The above tool setup

- gets you current API documentation
- an understanding of how the API in question is supposed to work
- An easy way to look for Excessive Data Exposure
- summarized endpoints and methods in an easily readable format
- how requests and responses look like along with associated headers, parameters, control & session mechanisms

The process of creating rogue documentation using Dev Tools is covered beautifully by Dan so I will not go into it.

1. Have your browser and the Network Tab in Developer Tools open. Burp should be configured and ready to proxy traffic through (intercept should be off) and the proxy history should be clear (it will get messy soon)

Use the application as a normal user and exhaust all the functionality. From the network tab, using your capture create your documentation which you can verify using swagger-editor if there are any issues. Great, we have a current API Documentation.

2. Use the power of jq (a command line JSON parser) in order to extract clean information from the documentation file. (It helps to convert the yaml file into JSON use swgger-editor)

Extract clean, uncluttered endpoints

```
> jq -r '.paths | keys[]' documentation.json > endpoints.txt
```

Extract the description, method and endpoint

* Better to save the command found at https://gist.github.com/DanaEpp/ca4b612734e73b4cf46ae6eeab6626b6 and read it using jq

```
jq -r -f endpoints-dump.jq $documentation.json > endpoint-description.txt
```

This information can be complied into a your note taking. The power of grep can be further used to grep for specific methods and group the requests all together. E.g Grep for only GET, or POST, or DELETE. (Pay attention to user-case) This will help you focus on particular requests that meet a vulnerability requirement. E.g Broken Authorization using GET methods.

```
cat endpoint-description.txt | grep GET > get-endpoints.txt
```

|__ will create a text file of the all the GET requests along with their description (if available; can be added if required).

There should now be a nice documentation that has the important information extracted out from it. An idea of how endpoints are structured should be in process and any interesting endpoints should be standing out by now.

3. In Burp, with the traffic in scope captured, arrange the request from the 1st to last. Highlight all of them with a simple color so that you know these were the original requests made. This is beneficial for later when you start testing properly. Keeping the originals around can be helpful and highlighting is simple way of identifying the originals.

4. Go through the requests and responses in order to

- understand naming schemes
- observe used methods
- find objects in use (should be noted in order to fuzz for later)
- look for Excessive Data Exposure
- find specific headers
- understand Authorization & Authentication mechanisms (session cookies or JWTs)

While going through the requests/ responses, make notes on any endpoints that may be candidates for vulnerabilities -BOLA, Mass Assignment etc.

5. For the sake of focus and not mindlessly barging our way through the vulnerable API - that the developers have asked us to check if it is possible to

- access another users details
- access mechanic reports
- reset another users password
- find any sensitive information being leaked
- If a DoS attack is possible?
- alter another users profile
- Purchase items for free or for lower price
- alter the default balance
- carry out SSRF
- issue free coupons
- redeem used coupons
- Access any endpoints as an unauthenticated user.
- alter a JWT in any way
- find anything else that may be a issue.

STAGE TWO - BREACH

Required Tools :

Browser - Incognito and Normal
Burp Suit (any edition)
JWT Editor Extension
Autorize Extension

6. Now that we have our scope of focus lets work on testing for Broken Authorization. An idea of how the API functions along with any endpoints of interest should be set, control and authentication mechanisms should be identified. Any excessive data exposure identified in the previous steps should be kept in mind so that any BOLA / BLFA vulnerabilities found could potentially be escalated.

The testing for Broken Authorization with regards to Objects and Functions can be automated using the Burp Extension, Autorize. Having the JWT EDitor extension loaded simply allows for reading the contents of the JWT to make sure the detected bypass was not a false positive.

Dan explains beautifully how to make use of Autorize https://danaepp.com/automate-your-api-hacking-with-autorize so I won't go into it in any detail.

(The way Autorize works by replacing the tokens with another users and also trying to send requests without tokens. A simple but effective way to automate testing for broken access controls.)

Apply this to crAPI to test for Broken Object Level Authorization and Broken Function Level Authorization.

Once the analysis has been completed, it is advised to export the results independently of each other - Bypassed, Enforced, Is Enforced? - so that endpoints of interest can be examined and arranged easily. Exporting can be as HTML or as CSV.

4 Reports should be generated
1 : all statuses
2 : Bypassed
3 : Is Enforced?
4 : Enforced

This will help answer the clients scope queries; namely is it possible to

- alter another users profile
- access another users details
- reset another users password
- Access any endpoints as an unauthenticated user.

Reports that are categorized as Bypassed are of the most interest. Endpoints that are of value and that can have potentially damaging consequences should be noted. It is best to manually confirm the endpoints.

Manual testing involves using the token from the Incognito browser user - token/ session cookie can be extracted from Dev Tools - and another token for the victim from Burp. The two should be exchanged in a request to repeater (any token - the vulnerability if present is not picky). Endpoints that are confirmed manually to be vulnerable to BOLA / BLFA are noted.

Applying this method to crAPI and we find that the endpoints

- /identity/api/v2/vehicle/3471092d-3604-429b-91db-f5a57c5fed02/location
- /dashboard
- /identity/api/v2/user/reset-password
- /identity/api/v2/user/change-email

are vulnerable to BOLA and BLFA respectively (two by two). This is not all the end-points, I will leave others for your curiosity.

Remembering that we have Excessive Data Exposure in the community endpoints that disclose

- vehicle UIDs
- user emails
- users full name

We can chain these vulnerabilities for better impact.

1. Enumerate the others users vehicle UUIDs and run them through Intruder or your favorite fuzzing tool in order to access other users vehicle location. This vulnerability has critical impact especially if high level clients will be making use of the application.

Broken Object Level Authorization + Excessive Data Exposure

2. User emails can be enumerated from endpoints. The design of the API means that verification is sent to the new email address and not the previous email. This works in the favor of the attacker because the email to which the verification is sent to is under their control. All they need is to have a list of valid email addresses which they can change. The JWT does not validate the scope of the user or enforce it. Resetting the email is a two step process. Until both parts are complete the email address is not changed. The presence of the vulnerability can be verified by fuzzing a list of email addresses to one new address. The response shows that the reset token was sent to the email domain under our control. The impact is locking users out of their account.

Broken Level Function Authorization + Excessive Data Exposure

3. When another users JWT is obtained, it is possible to change their password thus effectively locking them out of their account. The catch is, a token needs to be obtained...unless there was a way to tamper with the JWT as it incorporates the email address in its header.

4. By replacing the current JWT with another users JWT, it is possible to view another users dashboard and see their credit. This is a breach of privacy and should not be a functionality that is allowed.

STAGE 3 - WEIRDNESS

In the Autorize test, the endpoints

- PUT /identity/api/v2/user/videos/47
- POST /identity/api/v2/user/videos

come up as bypassed but when manually tested and verified in the browser there were no changes in either context regardless of the server returning a 200 OK response.

Using OPTIONS on the /videos endpoint did not bring anything interesting but running OPTIONS on /videos/47 revealed that the endpoint also accepted DELETE methods.

Running DELETE gave an interesting message. "This is an admin function. Try to access the admin API". So

1. We have a verbose error message (Sensitive Information Disclosure) and

2. There may be an entire admin API hidden below the surface.

Time for some fuzzing.

Focusing specifically on the PUT request, changing the /user/ object to admin results and the HTTP method to DELETE results successful deletion of the video.

A sensitive information disclosure that resulted in privilege escalation (BLFA)

Analyzing the categorized endpoints that were obtained and picking out endpoints that may belong to admin functionality was unsuccessful.

STAGE 4 : HERE I COME

Time for some fuzzing action. cat out the clean endpoints.txt file and lets look for requests suitable for object (O) and action (A)fuzzing. THe endpoints selected are those whose naming scheme can relate to other endpoints. if done correctly, found endpoints should be re-discovered during the fuzz.

- /commmunity/api/v2/community/posts/recent
    - /v2/FUZZ1/FUZZ2/FUZZ3 : both are objects : suitable clusterbomb attack

to be ignored due to the different combinations resulting in a 600+ million request which is not feasible; especially as the community endpoint is not a critical endpoint.

- identity/api/auth/login
    - /auth/login : both objects : cluster bomb attack

- /identity/api/v2/user/change-email
    - /user/change-email : object and action respectively : suitable attack is clusterbomb.

2 : fuzzing attacks
|__ one for both objects
|__ one object and actions

- /workshop/api/shop/orders
  - shop/orders : both are objects : clusterbomb attack

- workshop/api/shop/orders/return_order
  2 atacks:
  |__ /return_order : sniper attack with actions wordlist
  |__ /orders/return_order : clusterbomb attack with object and action respectively.

- workshop/api/shop/return_qr_code
  - /return_qr_code : sniper attack with actions wordlist

In Burp, the Intruder tab can be used to launch this attack. If the Community version of Burp is in use, then Ffuf can be used to get over the request-limiting barrier.

Define the objects to be fuzzed. E.g fuzz1, fuzz2 and copy and paste the whole request into a text file. Have the object and action wordlists available from Github https://github.com/danielmiessler/SecLists/tree/master/Discovery/Web-Content/api ready. Then for each endpoint that is a clusterbomb attack using Ffuf

`ffuf -request saved-endpoint-from-burp.txt -request-proto http -mode clusterbomb -w objects.txt:1FUZZ -w objects.txt:2FUZZ`

The process is fully explained : https://www.hackingarticles.in/comprehensive-guide-on-ffuf/

These fuzzing attempts will take time so use the time for some coffee or examine other possible existing vulnerabilities.

STAGE 5 : BUA in the HOUSE

Broken User Authentication. Number 2 on the OWASP API Top 10.

Time to use our special tool specifically for JWTs; jwt_tool. Installation guide available: <link>

Obtain a valid JWT and run it through

- `./jwt_tool $token`

in order to easily extract information about the JWT and what possible security is in place. The JWT used is relatively simple in the sense that it does not contain many headers. It mainly defines who it is intended for under the sub header. From our previous BOLA and BLFA we know that resources are not verified with the token. You can easily access other users data.

jwt_tool allows some automated attacks which makes analysis of the token easier. None attacks, unverified signatures, secret cracking and algorithm confusion attacks can be carried out by jwt_tool.

We could run jwt_tool as a script kiddie using

```
jwt_tool -t http://localhost:8888/identity/api/v2/user/dashboard -rh "Authorization: Bearer $token" -M pb
```

-rh : request headers
-M : attack mode, playbook tests

but lets break things down, lets use the tool like someone who is moving intentionally. The above command is great if you know what the tool is doing

Using jwt_tool + Burp lets replay a request as a baseline in order to make sure everything is working well

* Note: requests from jwt_tool can be viewed in Burp Proxy

```
jwt_tool -t http://localhost:8888/identity/api/v2/user/dashboard -rh "Authorization: Bearer $token" -cv "$user-email"
```

-cv : canary token. This is a "unique" string present in the response that will be present upon a successful request.

Lets check if the signature is being verified correctly.

None: Algroithm attack

```
jwt_tool -t http://localhost:8888/identity/api/v2/user/dashboard -rh "Authorization: Bearer $token" -cv "$user-email" -v -X a
```

-X a : Exploit, algorithm none.
-v : verbose. Should show some obfuscation attempts

Null signatrue

```
jwt_tool -t http://localhost:8888/identity/api/v2/user/dashboard -rh "Authorization: Bearer $token" -cv "$user-email" -v -X n
```

The signature is being verified correctly. The JWT is using the HS512 algorithm which is symmetric so lets try and brute-force the secret. Running the playbook test shows that the default wordlist in jwt_tool did not work in cracking. There are some suggestions on rules that we can apply but lets try and create our own custom password list.

Cewl - Custom Wordlist Generator, available on Github https://github.com/digininja/CeWL This will crawl crapi and discover any words that will be of intreset. It will be the foundation of our wordlist and hopefully, the secret token will be found.

```
cewl http://localhost:8888/dashboard -w crawled-words.txt -H "Authorization: Bearer $token"
```

We can run our basic list against the token using jwt_tool cracking features

- `./jwt_tool $token -C -d crawled-list.txt`

No luck. Lets improve this wordlist slowly. The Mentalist is a custom wordlist generator and can be used to mangle our raw list.

Github: https://github.com/sc0tfree/mentalist
cd /mentalist

sudo python3 setup.py install

Lets first attempt to change case and make all words lowercase.

- `./jwt_tool $token -C -d crawled-list-lowercase.txt`

Bingo.

That was not so bad. Of course if that didn't work, we would have gone on advancing and expanding the wordlist using upper, numbers, special characters etc.

We now have the "secret" passphrase being used to generate tokens. We can make use of the JWT Editor, https://jwt.io in order to create our own valid tokens.

- Copy a valid token into jwt.io and modify the sub to an arbitrary valid user while adding the found "secret" into the signature header. It should not be base-64 encoded as the server has not been configured in that way. Copy the generated JWT into a Burp request and viola, we can access another users information using our token.

Remember that we had a BLFA where we could reset another users password but the stumbling block was we needed to have their token. No need anymore. Remember the Excessive Data Exposure that allowed user emails to be enumerated, this can be chained to escalate. It can also be made to work with the change email functionality.

Broken User Authentication = Broken Function Level Authorization + Excessive Data + Broken Object Level Authorization

-- THE EXTRA MILE

As an extra mileage, seeing as we have the keys to the kingdom, can leverage the Excessive Data Exposure on the contact-mechanic page - that reveals the mechanic email addresses - to change a mechanics password in order to access the mechanic API?

Lets try and access the dashboard and see if the endpoint is the same as a normal user

`jwt_tool -t http://localhost:8888/identity/api/v2/user/dashboard -I -pc sub -pv $mechanic-email -S hs512 -p $secret`

Great and the response shows the role parameter. Can this be a parameter for Mass Assignment? Lets keep that for later.

Copy the modified mechanic token from Burp Repeater and paste it into a normal request that resets the password. Change it and viola, log in to the API as a mechanic is successful. We can effectively look the mechanic out of the account by changing the email. We have accessed a whole other functionality in the API.

*NOTE: I am aware there is another way to leverage this but this is a more impactful way of exploiting the Broken User Authentication Vulnerability.

MASS ASSIGNMENT

Required:
OWASP ZAP Proxy

*It is advised to make a whole different folder for Mass Assignment resources.
*Authorization issue: Use the Replacer function in OWASP ZAP to allow all requests to be authenticated. Simply add a valid token each time.

Using the text file that contained the endpoints in the clean, organized from *jq* output, let us identify requests that

1. Have the ability to modify objects using POST/PUT
2. That accept user Input

Using the *cat* command, extract endpoints that meet the above criteria and direct the output into a another file.

```
cat post-endpoints.txt put-endpoints.txt > Mass-Assignment-Requests.txt
```

For organizational purposes, have this data in a spreadsheet, marking the endpoint, the method,the request parameters,the response parameter, what objects the endpoint modifies and what it can be abused for (e.g privilege escalation).

The documentation can assist in this, especially if you included examples while creating the Swagger file (sample request and responses will be shown.

An example spreadsheet:

ID	METHOD	REQUEST	REQUEST PARAM	RESPONSE PARAMS	NOTES	AFFECTS	WHY OF INTEREST
1	POST	identity/api/auth/signup	name, email, number, password		Sign up request | User registered sucessfully	The users database / object	Adds a user into a database/ object that has different records
2	POST	/identity/api/v2/vehicle/add_vehicle	vin, pincode		The response is "Vehicle registered	The vehicle database	
3	POST	/community/api/v2/community/posts/qzEog3wwfVBvZ2eCx4yKuk/comment	id, title, nickname, email, vehicleid		Purpose is to post on an individual post	The comments public page	Modifies the public content page. May have author related information
4	POST	/workshop/api/merchant/contact_mechanic	mechanic_code, problem_details, vin, mechanic_api,	id, sent, report_link	Send a report to a designted mechanic	The mechanic reports object/ datbase	Modified the mechanic object reports
5	POST	/workshop/api/shop/orders	product_id, quantity	id, message=order sent..., credit	Purpose is to buy an item		Modifies the shop order quantity
6	POST	/workshop/api/shop/orders/return_order	name, in (query), required	status, id, price,	Returning an order		Can try and go thorugh th ebusiness
7	POST	/community/api/v2/community/posts	title, content		Make a whole new post		Modifies the public content page. May have author related information
8	POST	/identity/api/v2/user/change-email	old_email, new_email	message returned confirming successful change	Endpoint to change email address		changes the object releted to a user, can add information that modifies other name:value pairs.
9	POST	/identity/api/v2/user/pictures	id, name, status, available_credit, user, picture		Purpose is to change/ update profile picture		returns a full object, a post request can affect other fields
10	POST	/identity/api/v2/user/videos	id, video_name, conversion_params, profilevideo	Id, video_name, conversion_params, profilevideo	Purpose is to upload a video on the community page		Privilege Escalation
11	PUT	/identity/api/v2/user/videos/{id}	videoName	video-name	Purpose is to re-name the vidoe		Potentially Privilege Escalation

ZAP Proxy has a beautiful feature called Contexts (scope in Burp Suite) where the interesting requests can be added into and the Site map will only show those requests. Burpe has this feature, but I prefer ZAP, especially if you do not have the Pro edition of Burp.

Add the URL endpoints into the Contexts. On ZAP the text file containing the URLs can be imported into the Context.

(Note: remember to include the prefix http://localhost:8888 . I love using Sublime Text due to its ability for programming and editing all strings simultaneously that match a pattern. The prefix can be set and all the requests can be imported into a context. Requests that are not vulnerable to Mass Assignment can be removed from the context in order to keep the tree organized and focused)

The endpoints should all be tested by adding any parameters that can result in

possible escalation to Admin or Mechanic role. I will focus on a particular group of requests to the Shop feature.

Extract all the associated requests that interact with the Shop regardless of their method.

cat endpoints.txt | grep shop | grep -v mechanic > Mass-assignments-Shop-requests.txt

*Above command will extract any request with the *shop* pattern in it. The mechanic endpoint is also in it and this needs to be removed so the -v flag remvos any string pattern that matches *mechanic.*

On ZAP, create a new context that puts the Shop endpoints into scope. Notice the business logic flow.

One interest in particular is interesting.

http://localhost:8888/workshop/api/shop/products.

The response is a GET but in the response, the Allow header states that the POST method is also allowed. This should be interesting.

Changing the request method to POST and sending an empty request body results in a response stating "{"name":["This field is required."],"price":["This field is required."],"image_url":["This field is required."]}". This looks like the ability to upload our own products. Could this be a BLFA.

Sending a test request with the POST Data

{"name":"Ferrari","price": 1,"image_url": "image.png"}

*You may need to add the 'Content-Type: application/json' Header.

it is successful. The change is also reflected when visiting the products page. If purchased the credit also goes down.

The math here is a simple subtract. What if we can post a negative price. Mathematically, this will result in an addition on our credit.

Sending another request with the POST data

{"name":"Ferrari","price": -1000,"image_url": "http://localhost:5000/image.png"}

Creates the product...and purchasing the product results in an increase in our credit.

Prefect. We can increase our credit indefinitely and purchase valid items.

This is a combination of Broken Function Level Authorization and Mass Assignment in a different context.

Lets see what other requests are vulnerable...

HOSPITAL VISITS WITH THE DOCTOR AS A HACKER = INJECTIONS

Injection vulnerabilities can be present within headers, query string parameters and requests that accept user input.

NOSQL

Lets extract requests from our clean endpoint-description.txt list. There are a lot of requests that can be possible candidates, to simplify things lets refer back to our clients scope,

- Can free coupons be issued?
- Can used coupons be redeemed?

One request in particular stands out.

POST /community/api/v2/coupon/validate-coupon

This request takes a coupon and checks if it is valid. The logic implies that there is a database of coupons stored and a comparison is made to evalute. If the coupon is present then it is valid.

*It is best to have some basics on NoSQL syntax and how it works. Some resources are
TryHackMe : NoSQL Basics https://tryhackme.com/room/nosqlinjectiontutorial
MongoDB operators : https://www.mongodb.com/docs/manual/reference/operator/query/

The tool **nosqli** can be used to automate testing and will reveal a host of different injections that can be verified using ZAP/ Burp. A note when using ZAP raw requests, to remove the http://domain as it seems to mess with how the tool reads the file. It appends the host into the URL by itself.

Exploiting

Using the $ne meaning Not equal to operator lets send the JSON body as

{“coupon”:{“$ne”: 4444}}

This should evaluate as true and will return a document (NoSQL term) ... which it does.

Now we have a valid coupon. Checking that it works proves successful and our balance increases.

1. Other tokens can be harvested by playing around with NoSQL operators such as $eq and following the format TRAC$ and fuzzing from 10-99 and and then 100 to 999 to get around the 0 padding issue.

or an alternative

2. An alternative way is to use the $nin (not in) operator and add found tokens as the payload

{"coupon_code": {"$nin": ["TRAC075","TRAC065","TRAC125"]}}

We do find some more coupons this way.

Our client had a second query, can a used coupon be re-used? Looking around and try

t manipulate the coupon endpoint with different methods and body formats doesn’t yield and results.

SSRF

SSRF vulnerabilities are present in request that take parameters, and user input. It is important to look at the context that a URL executes in, to pick out the correct endpoints. Looking at our endpoints, these seem to peek our interest.

*http://localhost:8888/community/api/v2/community/posts/qzEog3wwfVBvZ2eCx4yKuk.**
*http://localhost:8888/community/api/v2/community/posts/recent.**
*http://localhost:8888/workshop/api/merchant/contact_mechanic.**
*http://localhost:8888/workshop/api/shop/products.**

The products endpoint, had a Mass Assignment + BLFA vulnerability allowing us to add products to the shop. The POST request took an image as a input. Lets check if we can abuse this with SSRF

Have a netcat or python HTTP server listening on a port and set the image url to localhost

```
nc -lnvp 6000
```

{<snip>..”image_url”:”http:127.1:6000”,..<snip>}

Upon reloading the products page, we should see a connection attempt was made. We can test again using OOB techniques (I made use of interactsh) and a interaction request was made. More importantly, a HTTP interaction was noted. We can try to test if it is possible to have a web shell that is fetched and executed on this endpoint in order to achieve RCE. I wont get into that.

Another endpoint that seems interesting

/workshop/api/merchant/contact_mechanic

does not seem to be vulnerable to local SSRF - even with various obfuscation techniques applied but it is vulnerable to Remote SSRF. When tested using Interactsh, a DNS and HTTP request was made and the API response contained the testing string. This may mean the endpoint is rendering the HTML ... which could mean we could server a HTML file that contained malicious JavaScript (among other things) and executed ... a bonus if it works.

The /community.* endpoint does not seem to be vulnerable to any SSRF as the input is simply reflect in the public site.

A bonus place to test is the Referrer header.(though due to the fact that that analytical software likely won’t be present, this wont work with this API)

KEEP PROPER STOCK

Improper Asset Management is Number 9 on the OWASP API Top 10. In order to test for Improper Asset Management, all the requests that have versioning need to be selected. For crAPI, the version is in the URL. We need a series of endpoints that we can feed into OWASP ZAPs contexts.

The baseline for a request is v2 so we look for all the requests that contain the string v2 in its URL path.

```
cat endpoints.txt | grep -v2 > Versioned-assests.txt
```

If we are dealing with a large group of endpoints, it first helps to separate the requests based on what is most likely critical. This should help to focus our testing.

*It can help to have *tmux* (terminal emulator) having the endpoints-description text in another window in order to aid with picking out important endpoints.

Our critical endpoints imported into the OWASP ZAP Versioning Context should be

http://localhost:8888/community/api/v2/coupon/validate-coupon
http://localhost:8888/identity/api/v2/user/change-email
http://localhost:8888/identity/api/v2/user/dashboard
http://localhost:8888/identity/api/v2/user/reset-password
http://localhost:8888/identity/api/v2/user/verify-email-token

Send each request individually to the Requester and manually change the versioning from v2 to v1 and v3 and monitor any discrepancies.

** I admit, in Postman this would have been easier but not a fan of Postman with its limitations that are tailored to poor network locations.*

For some unknown reason, I seem to have a different version of crAPI when running this (installed locally) and my instance does not have a OTP-CODE endpoint which theoretically is supposed to be vulnerable to Improper Asset Management which exploits the lack of rate-limiting allowing for a brute-force attack. The token sent is a string of characters. Potentially they can be harvested in order to look for any predictability. I will leave that to you...

meaning that this a wrap.