TASK

Authentication with JWT

Visit our website

Introduction

MAKING SURE YOUR USER IS WHO THEY SAY THEY ARE

Up until now, you've been exposed to web development languages, frameworks, libraries, and paradigms. One crucial (and final) component to web dev that you haven't yet made contact with is authentication. That is, how to ensure your users only have access to the facets that they're supposed to. In this task, we'll look at one popular way of doing so.

THE HISTORY OF REST AND EARLY AUTHENTICATION

In the early days of the internet, when the idea of RESTful requests was first being designed, software engineers of the time decided that such requests should be *stateless*. That is, any REST request need not rely on any prior request having been made. This meant that everything a user wanted to do had to happen in one main request — e.g. to buy a product online the REST request needed to contain enough information to authenticate the user, select the products to buy, and provide payment information.

Even though this meant really long-running requests, there were good reasons for this decision. One is that with the early internet infrastructure, making a request already took a long time, and many of them were bound to fail anyway because of physical networking issues. Another is that the early hardware had a hard time dealing with many requests at a time, so cutting down on the number of user requests per transaction was the top priority.

The first web authentication method, called basic authentication, was created under these limitations. It very simply has the username and password in the header of every request. The disadvantage here is that if you're using http (not https) then that password is being sent in plaintext and is rather easily interceptable by methods such as **man-in-the-middle attacks**. It is, however, the simplest and thus the quickest method to implement as a programmer. Basic authentication is still used today for the odd occasion where only a single REST request needs to be made, and the endpoint supports it. For example, getting a weather forecast, stock market history, or version control repo data. However, it is only ever used over https, and usually not in browser situations (only via APIs in apps, where the user doesn't have access to the requests themselves).

CONTEMPORARY AUTHENTICATION

Today, technology has improved exponentially to the point where you can make 10 requests and with reasonable confidence expect them all to complete before a second has passed — even on mediocre hardware and internet connection by today's standard. Through multiple iterations, many different authentication techniques came about. Most of them that are still used today work something like this:

1. The client sends the username and password to an authentication endpoint.
2. The auth (authentication) endpoint checks the data and, if legit, generates an authentication token which is relevant to the requesting user's session.
3. The client stores the token and adds it to the header of further requests.
4. The server checks the token every time it receives a request and uses it to determine which user is making the request.

There are many different ways to implement this process. Some have multiple-step authentication, and some require extra internal authentication steps with regular requests. All of them expend a great deal of energy to ensure this generated token is secure and hacker-proof.

WHAT ARE JSON WEB TOKENS (JWT)?

JSON web tokens are a URL-safe method of transferring information between two parties. JWTs are mainly used to transfer information securely between two endpoints – server and client (web browser). Their purpose is to authenticate and authorise processes and often help in the stateless exchange of information.

COMPONENTS OF A JWT

The components of JWT consist of mainly three things:

- Header
- Payload
- Signature

Let's look at all three of these constituents of JWTs:

Headers

Headers provide information on how a JWT is encoded. It consists of cryptic algorithms that help in securing it. It is a JSON object with two main properties:

- **alg**: Short for 'algorithm', this property specifies the algorithm used to sign the JWT. Some of the most popular algorithms are **HMAC-SHA256**, RSA, and ESDSA. The choice of algorithm alters the security and integrity of the token.

- **typ**: This property indicates the type of tokens. For JWT, this is specifically set to JWT.

An example of a JWT header is given below:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Base64 encoding is a way to represent binary data in a textual format that is safe to be transmitted over various protocols. When a Base64 header is encoded, it is converted from binary data to ASCII characters. To encode any header content into Base64 encoding, you can use any programming language that supports the use of Base64 libraries/modules. The original header above is converted into a Base64 encoded header using the **JWT platform**.

The Base64 encoded header used for a JWT is given below:

```
eyJhbGciOiAiSFMyNTYiLCAidHlwIjogIkpXVCJ9
```

Payload

The payload of a JWT consists of actual claims or statements about the subject (user) or device. It also contains additional data. Claims are assertions made about the actual subject or device. They have three types:

- **Reserved claims:** These are pre-defined claims which have several special meanings, such as "iss" which stands for issuer, "sub," which means subject, "exp," which stands for expiration; and "iat," which means issues at time, etc. These claims convey standard information about the token.

- **Public claims:** These are customised claims made by the parties involved in the token exchange. They also provide extra information regarding a specific use case.
- **Private claims:** These claims are used to share private information between parties in a private manner. These claims are not registered in any official claims namespaces.

An example of a payload is given below:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

The Base64 encoded payload is given below:

```
eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1lIjogIkpvaG4gRG9lIiwgImlhdCI6IDE1MTYyMzkw
MjJ9
```

To produce the signature, we use the algorithm specified in the header. In our case, HMAC-SHA256 is a complicated name representing a kind of hashing function. You may recall hashing from the data structures task. It is the same idea: an object (message or index) is one-way encoded to a seemingly random hash. However, it's not completely random because the hash will reliably be computed the same given the same input. Our HS256 algorithm takes this further by requiring a secret key during encoding. In this case, you need the message (the header and the payload) and a secret key to produce the signature. Combining the three produces a hash (the signature) that will always reliably compute the same given the same header, payload, and key.

Signature

Creating a signature for a token is to take a header, a payload, and a secret key and apply a specific algorithm to it. The key can be private or public, depending on the algorithm used for the payloads. The signatures ensure that the tokens are authentic and their integrity is maintained. Only the parties who know the key can generate a valid signature if the key is private. The process of creating a signature is, mentioned below in the steps:

1. Base64 URL encoded header

2. Base64 URL encoded payload
3. Concatenating the header and the payload with a full stop.
4. Applying a specific algorithm to the concatenated string and secret key.

An example of the resulting signature is given below:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret key
)

header = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9'
payload = 'eyJpZCI6MTIzNCwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9'
msg = header + '.' + payload
sig = HS256('secret-key', msg).digestBase64()
```

Combining all three of these components creates a token. These tokens ensure that information is passed securely between two parties. An example of a signature after encoding is given below:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

The final JWT looks something like below:

```
eyJhbGciOiAiSFMyNTYiLCAidHlwIjogIkpXVCJ9.eyJzdWIiOiAiMTIzNDU2Nzg5MCIsICJuYW1
lIjogIkpvaG4gRG9lIiwgImlhdCI6IDE1MTYyMzkwMjJ9.dBjftJeZ4CVP-mB92K27uhbUJU1p1r
_wW1gFWFOEjXk
```

This token may look useless, but it contains valuable data like the user's ID, name, and admin level. Base64 can read this by decoding the middle part of the token (the payload). Note that Base64 encoding is two-way and does not require a secret key. That means that anyone with the token can read its contents. You should never put sensitive data like passwords or credit card numbers in JWTs. Permission and user data are fine, though.

The purpose of JWTs is not to hide data but to ensure its integrity. That is, to ensure it cannot be tampered with. This is achieved with the signature, which requires a secret key to produce. Anyone who alters the key's payload can only produce the correct signature with the secret key.

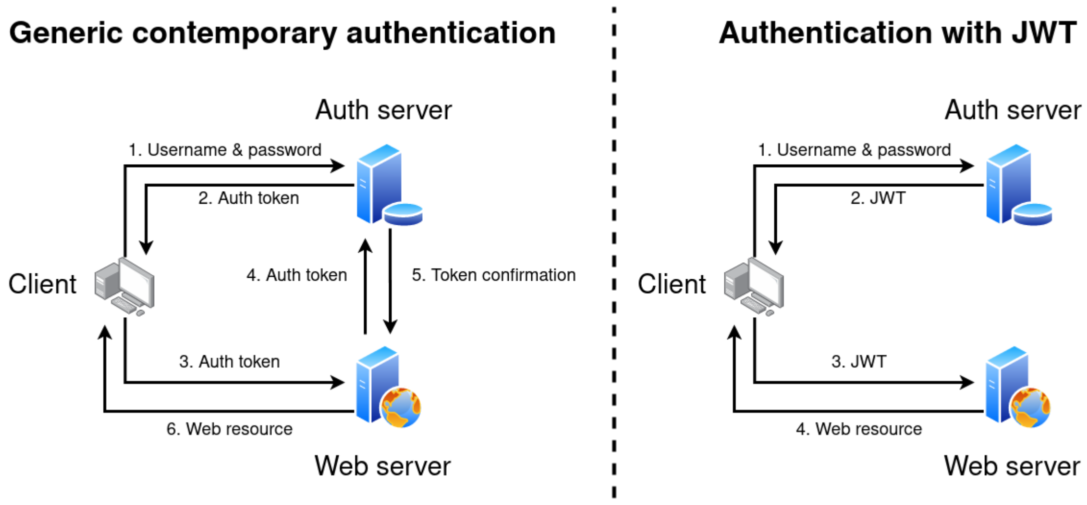
This JWT is ready to be transmitted. When the user receives it, they can split it into three parts: header, payload, and signature, and then use the claims in the payload for their intended purpose.

JWT OVER TRADITIONAL AUTHENTICATION?

So far we have learned about the most commonly used authentication method using JWTs. However, other such methods have been traditionally used for decades. JWT has only recently made its mark in the authentication world. JWTs are a way of representing data that does not allow tampering during transit and that can be validated using encryption keys. When the server receives a JWT, it has a surefire way of determining whether the data it contains is legitimate.

The purpose of authenticating a user is to ensure they only have access to the resources they should. For example, in a WordPress site, an admin should be able to delete and re-categorise posts, while authors should only create and edit posts. If you're using a generic authentication method from above, the server would have to check the auth token against the database to ensure it is valid and then do another lookup to check what resources the user has access to. This is a very time-consuming procedure, especially if your auth endpoint is on another server entirely (which they usually are on big projects).

JWT circumvents many of these issues by allowing the auth endpoint to put all the relevant permission data inside the token. When the server receives it, it can validate and decode the token (without DB lookups) and immediately grant or deny access. Compare the two authentication methods below:



With JWT, there is no direct communication between the web and auth servers. JWT contains all the necessary authorisation data, and its legitimacy can be validated independently.

IMPLEMENTATION OF JWT

Let's put JWT into practice by creating an Express app. As you may have created many Express.js apps in your Node.js journey, let's create another which starts your server. Use your knowledge from the earlier Express tasks to create a new Express app with the main file: **index.js.** Place the following code to get a basic app running:

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
const PORT = 8000

// Allows us to parse the body of a request
app.use(bodyParser.json())

// User login
app.post('/login', (req, res) => {
  // Req.body is sent by the client
  const usr = req.body.username
  const pwd = req.body.password
  res.send(`Username: ${usr}\n Password: ${pwd}`)
})

// Start the server
app.listen(PORT, () => console.log(
  `Now listening at http://localhost:${PORT}`))
```

You must install Express and further dependencies to run your file smoothly. Installing Express and body-parser by typing the following command in your terminal:

```
npm install express body-parser
```

You can now start your application by running the following command. Your command may differ if you are using nodemon instead:

```
node index.js
```

As you are well aware of Postman by now, you can use Postman to create POST requests to `http://localhost:8000/login` with the following body:

```
{
 "username": "zama",
 "password": "abcdef"
}
```

Be sure to include the header `Content-Type: application/json` to describe the body's content type accurately. This is standard practice in RESTful APIs.

If all goes well, your Express app server should respond with the following, confirming that it understands your POST request:

```
Username: zama
Password: abcdef
```

Now that we have a hello-world Express app working, let's discuss the design. We will have two endpoints: an authentication endpoint and a resource endpoint. The authentication endpoint will validate passwords and respond with a JWT if successful. The resource endpoint will respond with an arbitrary resource if the JWT is legitimate. We'll get into specifics later.

Let us craft the authentication endpoint. The first thing to do is to check the username and password and respond with HTTP 403 if it is wrong — this is the standard response code for auth failure in RESTful APIs. Remove the `res.send()` statement from before and replace it with the following code:

```
if (usr === 'zama' && pwd === 'secret') {

   // To-do

} else {
  res.status(403).send({ 'err': 'Incorrect login!' })
}
```

In a production application, you will not hardcode the username and password but look it up from a database. We are just doing it for demonstrative purposes. Note that the error response is in JSON format. This is unnecessary (it could have just been a simple string), but keeping the request and response content type consistent throughout your application is good practice.

Restart your app if you are not using Nodemon. You should see the incorrect login error message when making the same request.

The most popular JWT library for node is simply called **jsonwebtoken**. Install it using the following command: **`npm install jsonwebtoken`**

Then, add the following line to the top of your index.js file.

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

Now replace the to-do comment from above with the following code that generates a proper JWT:

```
payload = {
  'name': usr,
  'admin': false
}
const token = jwt.sign(JSON.stringify(payload), 'jwt-secret',
  { algorithm: 'HS256' })
res.send({ 'token': token })
```

See how easy the library makes the JWT generation process? We didn't have to code up any sticky algorithms mentioned above.

You can now restart your application if required. You can change your request body to have the correct password and make the request again. You should now receive your token like the following:

```
{
"token":"eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiemFtYSIsImFkbWluIjpmYWxzZX0.KTo5xX
D2ecZWEaABPny6Y2Z6dM0AxPcdtWAAW_TcKTM"
}
```

Now we will code up the resource endpoint. Add the following code to your **index.js**.

```
app.get('/resource', (req, res) => {
  const auth = req.headers['authorization']
  const token = auth.split(' ')[1]
  try {
    const decoded = jwt.verify(token, 'jwt-secret')
    res.send({
      'msg':
        `Hello, ${decoded.name}! Your JSON Web Token has been verified.`
    })
  } catch (err) {
    res.status(401).send({ 'err': 'Bad JWT!' })
  }
})
```

- JWTs are usually added to the authorisation header of requests in the format:
  `Authorization: Bearer <token>`
  This is the REST standard. The first two lines take care of extracting said token. Note that because of the way Express processes headers, the auth header is accessed with a lowercase “a”, even though the header itself has an uppercase “A”.
- The `jwt.verify()` method verifies a given token using the specified secret key. An error is thrown if it produces an unexpected signature (due to a bad key or tampered token) or if the token is malformed. We don’t have to specify the algorithm here because it is already in the header of the token.
- The token can be trusted if verification is successful and its payload is decoded. We then use this to construct a personalised message for the user.
- If the verification fails, an `HTTP 401` status is returned. This is the REST standard for bad authorisation.

Now in your REST tester, make a GET request to `http://localhost:8000/resource`, and add the following header to it (as one line):

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiemFtYSIsImFkbWluIjpmYWxzZX0.KTo5xXD2ecZWEaA
BPny6Y2Z6dM0AxPcdtWAAW_TcKTM
```

Your request should succeed. Notice how the request fails if you change merely a single character in the token. This is due to the strictness of the signature algorithm.

Finally, let us make use of user permissions. Add an admin resource and verify the user like so:

```
app.get('/admin_resource', (req, res) => {
  const token = req.headers['authorization'].split(' ')[1]
  try {
    const decoded = jwt.verify(token, 'jwt-secret')
    if (decoded.admin) {
      res.send({ 'msg': 'Success!' })
    } else {
      res.status(403).send(
        { 'msg': 'Your JWT was verified, but you are not an admin.' })
    }
  } catch (e) {
    res.sendStatus(401)
  }
})
```

Requesting this endpoint with your previous token will result in an HTTP 403 because your user is not an admin, even though the token was verified. You'll note that attempts to change the payload's admin-attribute to true directly (using base64 decode and encode) will result in a token verification failure. This is because the signature will be incorrect. The only way to get an admin token is to have your auth endpoint dispense it as such. To see this in action, change your login endpoint to have its payload contain `admin: true`, and use the resulting token to request this URL: [`http://localhost:8000/admin_resource`](http://localhost:8000/admin_resource).

Compulsory Task 1

The objective of this task is for you to understand how JSON Web Tokens are encoded and decoded.

Follow the steps below to complete this task:

1. Navigate to **jwt.io**.

2. Generate a JWT using jwt.io. Enter HS256 as your algorithm and enter your secret key **(“mysecretkey”)**.

3. Replace the payload with the following:

```
{
 "username": "zama",
 "password": "abcdef"
}
```

4. Take a screenshot of the decoded input (payload and secret key) and the encoded output (generated token).

5. Upload the screenshot to this task’s folder on your Dropbox

Compulsory Task 2

Create an Express app by following the steps in the ‘IMPLEMENTATION OF JWT’ section. It should have the following endpoints:

- `/login` - checks a POSTed username and password and produces a JWT.
- `/resource` - checks the JWT in the auth header and displays a message with the username.
- `/admin_resource` - checks the JWT and displays a message if the token is verified and the token holder is an admin.

Once again, when you are ready to have your code reviewed, **delete** the **node_modules** folder (this folder typically contains hundreds of files which, if you're working directly from Dropbox, has the potential to **slow down** Dropbox sync and, possibly, **your computer**), compress your project folder, and add it to the relevant task folder in Dropbox.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.