Jwt security considerations

* Json web tokens(jwts) are comprised of a short dated token, used for authentication in API calls, and a long dated token to refresh the short dated token upon its expiration. The short dated token contains information about the user they were issued to that the server can verify.
* A surface level explanation as to why the first token is short dated ( normally 15 mins or less). is so that if it is stolen, the amount of potential damage caused by unwanted authorised API activity is limited time Once the token has expired, if there were no refresh token, the session would expire and the user would be logged out. However if the long dated token is present, the long dated refresh token is then used to reissue a short dated token – allowing the session to continue seamlessly.
* If you dig a little deeper you then realise that this approach doesn’t actually mitigate the risk of an attacker having authorised access to an API, it only transfers the risk to the long dated token. In my design, both tokens would be stored as a cookie attached to each http request and hence if you’re able to access one token, you’ll be able to access the other. If you can access the long dated token, you can potentially create a new short dated token used for authentication.
* Below is a workflow to mitigate the risk of having long dated tokens, full stop.

Mitigating long dated workflow

User logs in → gets access token (15m) + refresh token (abc123)

 Access token expires → browser silently calls /refresh

 Server:

* Verifies refresh token abc123
* Issues **new** access token + **new** refresh token def456
* **Invalidates abc123**

 If someone later tries to use abc123 → token reuse detected → block or alert

* 1. This way, long dated tokens are refreshed alongside the short dated token. Specifically, the long dated property is used in a scenario when, let’s say, the user is reading something on the webpage, the short dated token expires, and then 20 minutes later they submit some new data. In this instance, because the refresh token is long dated, the tokens can be refreshed and the session can continue uninterrupted.
  2. An additional way to reduce the risk posed by jwt interception is the state full storage of tokens in the database. Previously, I implemented a stateless approach whereby once issued, the only way to deny access to a call with a valid jwt was to wait until it had expired or reboot the backend with an amended security key. With a state full solution, you can store the token as well as the IP address to which it was issued. Furthermore, if a request using the token comes from an unrecognised IP address the request can be denied and the token can be invalidated.

Jwt.header.payload.signature

* encoded in base64 for ease of transportation and handling.
* Signature is used to maintain integrity of the information inside the jwt .

If the payload or header change then the signature will be different.

* Jwt issued by authentication service if credentials are correct. Jwt contains ‘claims’ - information about the user i.e roles, email. Jwt has an expiration. Jwt is checked in every server request to see if it has expired. Furthermore the signature is checked to see if the data has been tampered with. If both of these check out, the API call will go ahead.

Key points. You need a Jwt to access an API as a user-client. It must not have expired. And it has to have the same signature as the signature created when the back-end validates it – header and payload taken from incoming request and then signature created at the server using the secret using

signature = HMACSHA256(base64url(header) + "." + base64url(payload), secret). The signature sent in the Jwt and the signature created by the server need to match.