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.

* In order to recall potentially comprised tokens, it makes sense to statefully store jwt tokens.

TLS – Checks that the data hasn’t been tampered with at rest

JWT – Signature ensures that the message isn’t tampered with at rest

Authentication -> jwtToken issued comprising of a short dated authentication token and long dated refresh token -> jwtToken encodes claims about the user -> stored in cookies and attached to every https request.

Advantage:

Stateless

Signature ensures authenticity of message

Disadvantage:

Stateless

In order to assess the risk of an authentication system, I need to think about the system holistically.

Browser

-Unauthorised access by the frontEnd

Frontend

- Any user input field can potentially access system resources in undesirable ways

- For example – an injected JavaScript script into a public comment section. Another user loads the page, the script is executed whilst loading the comment section, the script accesses cookies stored in browser and sends them to a private server (usually cookies contain information required for authentication).

- For example – an injected SQL script into a username input field. Depending on the architecture. SQL scripts can be run directly on the database.

- For example – CRSF. Whereby an evil website runs a script in the background to use your cookies to authenticate a unauthorised request to another company (e.g bank).

Backend

- Unauthorised access to API endpoints

Database

- Credentials are stolen

**🔒 Browser-Level Risks (Client-Side)**

**1. Clickjacking**

* Malicious site embeds your site in an invisible iframe to trick users into clicking buttons they can’t see.
* **Prevention**: X-Frame-Options: DENY or Content-Security-Policy: frame-ancestors 'none'.

**2. Man-in-the-Middle (MITM) Attacks**

* Unencrypted HTTP traffic can be intercepted (e.g. on public Wi-Fi).
* **Prevention**: Enforce HTTPS (use HSTS), secure cookies (Secure flag), and avoid mixed content.

**3. Local Storage Vulnerabilities**

* Sensitive data stored in localStorage or sessionStorage is accessible via JavaScript.
* If XSS occurs, attackers can steal it.
* **Prevention**: Don't store tokens or credentials in local storage unless absolutely necessary.

**🎨 Frontend Risks (User Input & UI)**

**4. DOM-based XSS**

* XSS attacks that occur **entirely on the client side**, through dynamic DOM manipulation.
* **Prevention**: Sanitize DOM inputs and avoid using innerHTML with untrusted data.

**5. Open Redirects**

* Malicious links redirect users to phishing sites through your domain.
* **Prevention**: Validate and restrict redirect URLs on the server.

**🔧 Backend / Server Risks**

**6. Insecure Direct Object References (IDOR)**

* Attackers change a parameter like ?user\_id=123 to ?user\_id=124 to access other users' data.
* **Prevention**: Always enforce **authorization** checks on the server side.

**7. Broken Authentication**

* Weak password policies, no rate limiting, and poorly managed session tokens.
* **Prevention**: Use MFA, strong password policies, rate limiting, and short session lifetimes.

**8. Insufficient Logging & Monitoring**

* Attacks go undetected due to poor observability.
* **Prevention**: Implement proper logging, monitoring, and alerting systems.

**🗄️ Database Risks**

**9. No Encryption at Rest**

* If your database is stolen, unencrypted data is immediately readable.
* **Prevention**: Encrypt sensitive data (PII, passwords, tokens) at rest.

**10. Improper Permissions / Overprivileged Accounts**

* Backend or app components have unnecessary access to all DB tables or servers.
* **Prevention**: Apply the principle of least privilege.

**🌐 API-Level Risks**

**11. Broken Access Control**

* Authenticated users can access endpoints or resources they shouldn’t.
* **Prevention**: Implement strict role-based access controls (RBAC).

**12. Mass Assignment**

* Users can submit fields like isAdmin=true if the server doesn’t sanitize input payloads.
* **Prevention**: Whitelist allowed fields in requests.

**🛠️ DevOps / Infrastructure**

**13. Exposed Environment Variables**

* .env files or secrets exposed via misconfigured servers or version control.
* **Prevention**: Never commit .env files, and restrict access to config servers.

**14. Outdated Libraries / Dependencies**

* Using vulnerable packages in frontend/backend.
* **Prevention**: Use dependency scanners (e.g., Snyk, npm audit) and update regularly.

**🔐 Authentication & Session Risks**

**15. JWT Token Misuse**

* Tokens that never expire or are stored in the browser insecurely.
* **Prevention**: Set short expirations and use HttpOnly cookies for storage.

**16. Session Fixation**

* An attacker sets a session ID for a user before login, then hijacks the session afterward.
* **Prevention**: Regenerate session IDs after login.

Triple authentication considerations

Design considerations

* User is only able to see the part of the process they contribute to

Triple authentication considerations

* Are triple authentication and payments made on an order level or on an item level ? Item level gives more flexibility if say only certain items are delivered but introduces more complexity in handling partially fulfilled purchase orders
* To create two tables, one called threeWayMatch pending and approved or to create just one table called threeWaymatch and have a column to indicate if they have been approved or not

Log

24/06/2025

* BPMN diagram still evolving, different software and 5 swim lanes
* Redesign purchaseOrder data models
* Continue with authentication clarification
* Tidy relations relationships on database diagrams

curl -X POST http://localhost:8080/api/auth -H "Content-Type: application/json" -d '{"key": "JohnProductionAnalyst@cookfood.com", "password": "Analyst12345"}'