JSON WEB TOKENS (JWT)

JSON Web Tokens (JWTs) have become extremely popular. JWTs seem deceivingly simple. However, to ensure their security properties, they depend on complex and often misunderstood concepts. This cheat sheet focuses on the underlying concepts. The cheat sheet covers essential knowledge for every developer producing or consuming JWTs.

INTRODUCTION

A JWT is a convenient way to represent claims securely. A claim is nothing more than a key/value pair. One common use case is a set of claims representing the user's identity. The claims are the payload of a JWT. Two other parts are the header and the signature.

- JWTs should always use the appropriate signature scheme
- If a JWT contains sensitive data, it should be encrypted
- JWTs require proper cryptographic key management
- Using JWTs for sessions introduces certain risks

JWT INTEGRITY VERIFICATION

Claims in a JWT are often used for security-sensitive operations. Preventing tampering with previously generated claims is essential. The issuer of a JWT signs the token, allowing the receiver to verify its integrity. These signatures are crucial for security.

SYMMETRIC SIGNATURES

Symmetric signatures use an HMAC function. They are easy to setup, but rely on the same secret for generating and verifying signatures. Symmetric signatures only work well within a single application.

ASYMMETRIC SIGNATURES

Asymmetric signatures rely on a public/private key pair. The private key is used for signing, and is kept secret. The public key is used for verification, and can be widely known. Asymmetric signatures are ideal for distributed scenarios

BEST PRACTICES

- [ ] Always verify the signature of JWT tokens
- [ ] Avoid library functions that do not verify signatures

  Example: The decode function of the auth0 Java JWT library
- [ ] Check that the secret of symmetric signatures is not shared
- [ ] A distributed setup should only use asymmetric signatures

JWT Encryption is a complex topic. It is out of scope for this cheat sheet.

VALIDATING JWTS

Apart from the signature, a JWT contains other security properties. These properties help enforce a lifetime on a JWT. They also identify the issuer and the intended target audience. The receiver of a JWT should always check these properties before using any of the claims.

- [ ] Check the **`exp`** claim to ensure the JWT is not expired

  *Alternatively verify lifetime using creation time in the **`iat`** claim*
- [ ] Check the **`nbf`** claim to ensure the JWT can already be used
- [ ] Check the **`iss`** claim against your list of trusted issuers
- [ ] Check the **`aud`** claim to see if the JWT is meant for you

Some libraries offer support for checking these properties. Verify which properties are covered, and complement these checks with your own.

CRYPTOGRAPHIC KEY MANAGEMENT

The use of keys for signatures and encryption requires careful management. Keys should be stored in a secure location. Keys also need to be rotated frequently. As a result, multiple keys can be in use simultaneously. The application has to foresee a way to manage the JWT key material.

- [ ] Store key material in a dedicated key vault service

  Keys should be fetched dynamically, instead of being hardcoded
- [ ] Use the **`kid`** claim in the header to identify a specific key

  Keys should be fetched dynamically, instead of being hardcoded
- [ ] Public keys can be embedded in the header of a JWT

  *The **`jwk`** claim can hold a JSON Web Key-formatted public key*
  *The **`x5c`** claim can hold a public key and X509-certificate*
- [ ] Validate an embedded public key with a list of known keys

  Failure to restrict keys causes an attacker's JWT to be accepted
- [ ] The header can also contain a URL pointing to public keys

  *The **`jku`** claim can point to a file containing JSON Web Keys*
  *The **`x5u`** claim can point to a certificate containing a public key*
- [ ] Validate a key URL against a safe list of URLs / domains

  Failure to restrict keys causes an attacker's JWT to be accepted

USING JWTS FOR AUTHORIZATION STATE

Many modern applications use JWTs to push authorization state to the client. Such an architecture benefits from a stateless backend, often at the cost of security. These JWTs are typically bearer tokens, which can be used or abused by whoever obtains them.

- It is hard to revoke a self-contained JWT before it expires
- [ ] JWTs with authorization data should have a short lifetime
- [ ] Combine short-lived JWTs with a long-lived session