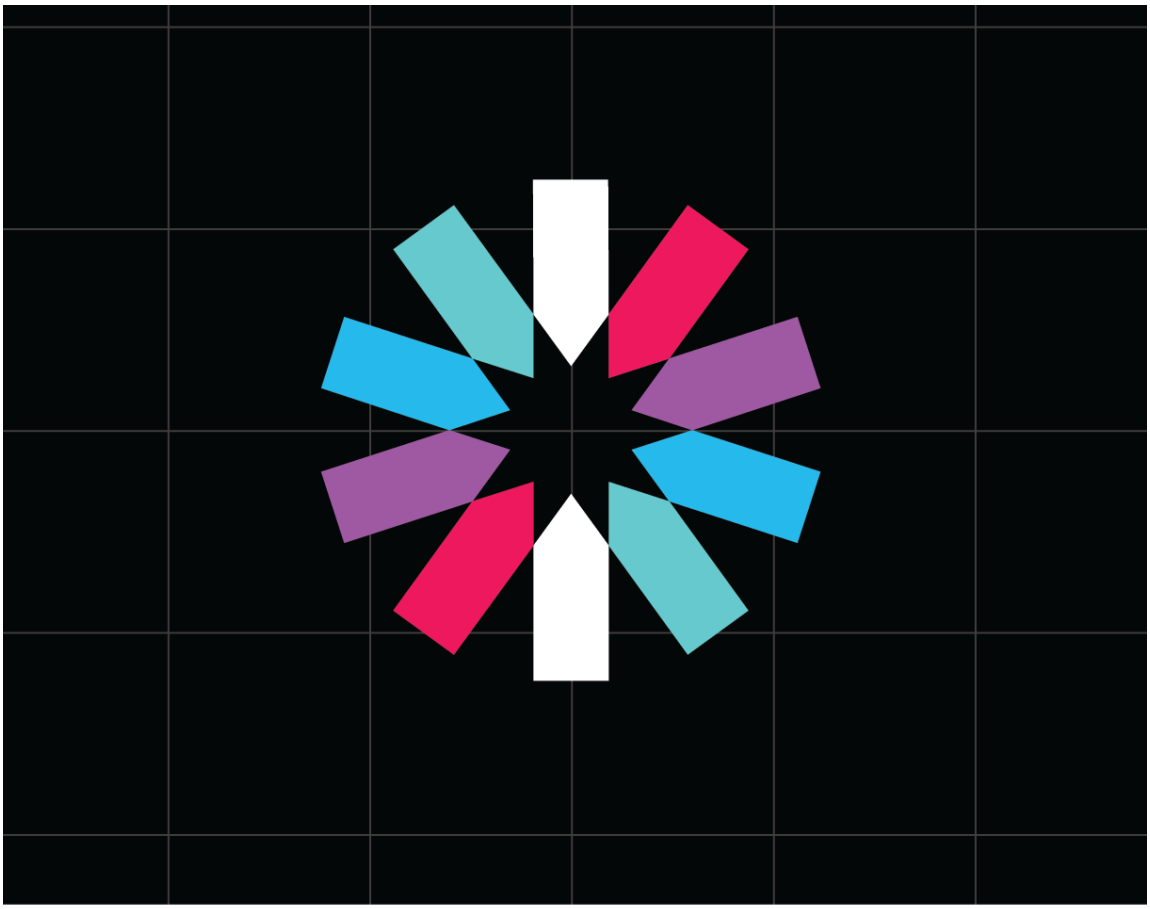

JWT HANDBOOK

By Sebastián Peyrott

Auth0

The JWT Handbook

Sebastián E. Peyrott, Auth0 Inc.

Version 0.14.1, 2016-2018

Contents

2.2.1 Access and Refresh Tokens

Access and refresh tokens are two types of tokens you will see a lot when analyzing different federated identity solutions. We will briefly explain what they are and how they help in the context of authentication and authorization.

Both concepts are usually implemented in the context of the OAuth2 specification[10]. The OAuth2 spec defines a series of steps necessary to provide access to resources by separating access from ownership (in other words, it allows several parties with different access levels to access the same resource). Several parts of these steps are *implementation defined*. That is, competing OAuth2 implementations may not be interoperable. For instance, the actual binary format of the tokens is *not specified*. Their purpose and functionality is.

Access tokens are tokens that give those who have them access to protected resources. These tokens are usually short-lived and may have an expiration date embedded in them. They may also carry or be associated with additional information (for instance, an access token may carry the IP address from which requests are allowed). This additional data is implementation defined.

Refresh tokens, on the other hand, allow clients to request new access tokens. For instance, after an access token has expired, a client may perform a request for a new access token to the authorization server. For this request to be satisfied, a refresh token is required. In contrast to access tokens, refresh tokens are usually long-lived.

[10] https://tools.ietf.org/html/rfc6749#section-1.4

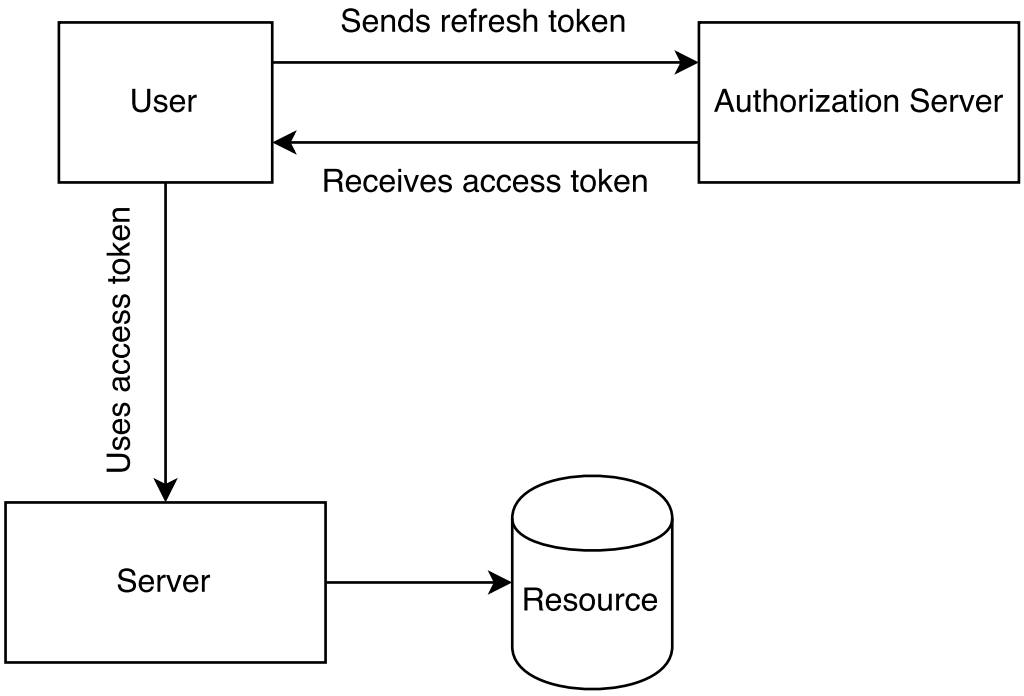

Figure 2.7: Refresh and access tokens

The key aspect of the separation between access and refresh tokens lies in the possibility of making access tokens easy to validate. An access token that carries a signature (such as a signed JWT) may be validated by the resource server on its own. There is no need to contact the authorization server for this purpose.

Refresh tokens, on the other hand, require access to the authorization server. By keeping validation separate from queries to the authorization server, better latency and less complex access patterns are possible. Appropriate security in case of token leaks is achieved by making access tokens as short-lived as possible and embedding additional checks (such as client checks) into them.

Refresh tokens, by virtue of being long-lived, must be protected from leaks. In the event of a leak, blacklisting may be necessary in the server (short-lived access tokens force refresh tokens to be used eventually, thus protecting the resource after it gets blacklisted and all access tokens are expired).

> Note: the concepts of access token and refresh token were introduced in OAuth2. OAuth 1.0 and 1.0a use the word *token* differently.

Chapter 3

JSON Web Tokens in Detail

As described in chapter 1, all JWTs are constructed from three different elements: the header, the payload, and the signature/encryption data. The first two elements are JSON objects of a certain structure. The third is dependent on the algorithm used for signing or encryption, and, in the case of *unencrypted* JWTs it is omitted. JWTs can be encoded in a *compact representation* known as *JWS/JWE Compact Serialization*.

> The JWS and JWE specifications define a third serialization format known as *JSON Serialization*, a non-compact representation that allows for multiple signatures or recipients in the same JWT. Is is explained in detail in chapters 4 and 5.

The compact serialization is a Base64[1] URL-safe encoding of the UTF-8[2] bytes of the first two JSON elements (the header and the payload) and the data, as required, for signing or encryption (which is not a JSON object itself). This data is Base64-URL encoded as well. These three elements are separated by dots (".").

> JWT uses a variant of Base64 encoding that is safe for URLs. This encoding basically substitutes the "+" and "/" characters for the "-" and "_" characters, respectively. Padding is removed as well. This variant is known as base64url[3]. Note that all references to Base64 encoding in this document refer to this variant.

The resulting sequence is a printable string like the following (newlines inserted for readability):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Notice the dots separating the three elements of the JWT (in order: the header, the payload, and the signature).

In this example the decoded header is:

[1] https://en.wikipedia.org/wiki/Base64
[2] https://en.wikipedia.org/wiki/UTF-8
[3] https://tools.ietf.org/html/rfc4648#section-5

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

The decoded payload is:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

And the secret required for verifying the signature is `secret`.

> JWT.io[4] is an interactive playground for learning more about JWTs. Copy the token from above and see what happens when you edit it.

3.1 The Header

Every JWT carries a header (also known as the *JOSE header*) with claims about itself. These claims establish the algorithms used, whether the JWT is signed or encrypted, and in general, how to parse the rest of the JWT.

According to the type of JWT in question, more fields may be mandatory in the header. For instance, encrypted JWTs carry information about the cryptographic algorithms used for key encryption and content encryption. These fields are not present for unencrypted JWTs.

The only mandatory claim for an *unencrypted* JWT header is the `alg` claim:

- **alg**: the main algorithm in use for signing and/or decrypting this JWT.

For unencrypted JWTs this claim must be set to the value `none`.

Optional header claims include the `typ` and `cty` claims:

- **typ**: the media type[5] of the JWT itself. This parameter is only meant to be used as a help for uses where JWTs may be mixed with other objects carrying a JOSE header. In practice, this rarely happens. When present, this claim should be set to the value `JWT`.
- **cty**: the content type. Most JWTs carry specific claims plus arbitrary data as part of their payload. For this case, the content type claim *must not* be set. For instances where the payload is a JWT itself (a nested JWT), this claim *must* be present and carry the value `JWT`. This tells the implementation that further processing of the nested JWT is required. Nested JWTs are rare, so the `cty` claim is rarely present in headers.

So, for unencrypted JWTs, the header is simply:

[4] https://jwt.io
[5] http://www.iana.org/assignments/media-types/media-types.xhtml

```
{
    "alg": "none"
}
```

which gets encoded to:

```
eyJhbGciOiJub25lIn0
```

> It is possible to add additional, user-defined claims to the header. This is generally of limited use, unless certain user-specific metadata is required in the case of encrypted JWTs before decryption.

3.2 The Payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

The payload is the element where all the interesting user data is usually added. In addition, certain claims defined in the spec may also be present. Just like the header, the payload is a JSON object. No claims are mandatory, although specific claims have a definite meaning. The JWT spec specifies that claims that are not understood by an implementation should be ignored. The claims with specific meanings attached to them are known as *registered claims*.

3.2.1 Registered Claims

- **iss**: from the word *issuer*. A case-sensitive string or URI that uniquely identifies the party that issued the JWT. Its interpretation is application specific (there is no central authority managing issuers).
- **sub**: from the word *subject*. A case-sensitive string or URI that uniquely identifies the party that this JWT carries information about. In other words, the claims contained in this JWT are statements about this party. The JWT spec specifies that this claim must be unique in the context of the issuer or, in cases where that is not possible, globally unique. Handling of this claim is application specific.
- **aud**: from the word *audience*. Either a single case-sensitive string or URI or an array of such values that uniquely identify the intended recipients of this JWT. In other words, when this claim is present, the party reading the data in this JWT must find itself in the *aud* claim or disregard the data contained in the JWT. As in the case of the *iss* and *sub* claims, this claim is application specific.
- **exp**: from the word *expiration* (time). A number representing a specific date and time in the format "seconds since epoch" as defined by POSIX[6]. This claims sets the exact moment from

[6] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15

which this JWT is considered *invalid*. Some implementations may allow for a certain skew between clocks (by considering this JWT to be valid for a few minutes after the expiration date).

- **nbf**: from *not before* (time). The opposite of the *exp* claim. A number representing a specific date and time in the format "seconds since epoch" as defined by POSIX[7]. This claim sets the exact moment from which this JWT is considered *valid*. The current time and date must be equal to or later than this date and time. Some implementations may allow for a certain skew.
- **iat**: from *issued at* (time). A number representing a specific date and time (in the same format as *exp* and *nbf*) at which this JWT was issued.
- **jti**: from *JWT ID*. A string representing a unique identifier for this JWT. This claim may be used to differentiate JWTs with other similar content (preventing replays, for instance). It is up to the implementation to guarantee uniqueness.

As you may have noticed, all names are short. This complies with one of the design requirements: to keep JWTs as small as possible.

String or URI: according to the JWT spec, a URI is interpreted as any string containing a `:` character. It is up to the implementation to provide valid values.

3.2.2 Public and Private Claims

All claims that are not part of the *registered claims* section are either **private** or **public** claims.

- **Private** claims: are those that are defined by *users* (consumers and producers) of the JWTs. In other words, these are ad hoc claims used for a particular case. As such, care must be taken to prevent collisions.
- **Public** claims: are claims that are either *registered* with the IANA JSON Web Token Claims registry[8] (a registry where users can register their claims and thus prevent collisions), or named using a collision resistant name (for instance, by prepending a namespace to its name).

In practice, most claims are either registered claims or private claims. In general, most JWTs are issued with a specific purpose and a clear set of potential users in mind. This makes the matter of picking collision resistant names simple.

Just as in the JSON parsing rules, duplicate claims (duplicate JSON keys) are handled by keeping only the last occurrence as the valid one. The JWT spec also makes it possible for implementations to consider JWTs with duplicate claims as *invalid*. In practice, if you are not sure about the implementation that will handle your JWTs, take care to avoid duplicate claims.

[7] http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15
[8] https://tools.ietf.org/html/rfc7519#section-10.1

3.3 Unsecured JWTs

With what we have learned so far, it is possible to construct unsecured JWTs. These are the simplest JWTs, formed by a simple (usually static) header:

```
{
    "alg": "none"
}
```

and a user defined payload. For instance:

```
{
  "sub": "user123",
  "session": "ch72gsb320000udocl363eofy",
  "name": "Pretty Name",
  "lastpage": "/views/settings"
}
```

As there is no signature or encryption, this JWT is encoded as simply two elements (newlines inserted for readability):

```
eyJhbGciOiJub25lIn0.
eyJzdWIiOiJ1c2VyMTIzIiwic2Vzc2lvbiI6ImNoNzJnc2IzMjAwMDB1ZG9jbDM2M
2VvZnkiLCJuYW1lIjoiUHJldHR5IE5hbWUiLCJsYXN0cGFnZSI6Ii92aWV3cy9zZXR0aW5ncyJ9.
```

An unsecured JWT like the one shown above may be fit for client-side use. For instance, if the session ID is a hard-to-guess number, and the rest of the data is only used by the client for constructing a view, the use of a signature is superfluous. This data can be used by a single-page web application to construct a view with the "pretty" name for the user without hitting the backend while he gets redirected to his last visited page. Even if a malicious user were to modify this data he or she would gain nothing.

> Note the trailing dot (`.`) in the compact representation. As there is no signature, it is simply an empty string. The dot is still added, though.

In practice, however, unsecured JWTs are rare.

3.4 Creating an Unsecured JWT

To arrive at the compact representation from the JSON versions of the header and the payload, perform the following steps:

1. Take the header as a byte array of its UTF-8 representation. The JWT spec *does not* require the JSON to be minified or stripped of meaningless characters (such as whitespace) before encoding.
2. Encode the byte array using the Base64-URL algorithm, removing trailing equal signs (`=`).
3. Take the payload as a byte array of its UTF-8 representation. The JWT spec *does not* require the JSON to be minified or stripped of meaningless characters (such as whitespace) before encoding.

4. Encode the byte array using the Base64-URL algorithm, removing trailing equal signs (=).
5. Concatenate the resulting strings, putting first the header, followed by a "." character, followed by the payload.

Validation of both the header and the payload (with respect to the presence of required claims and the correct use of each claim) must be performed before encoding.

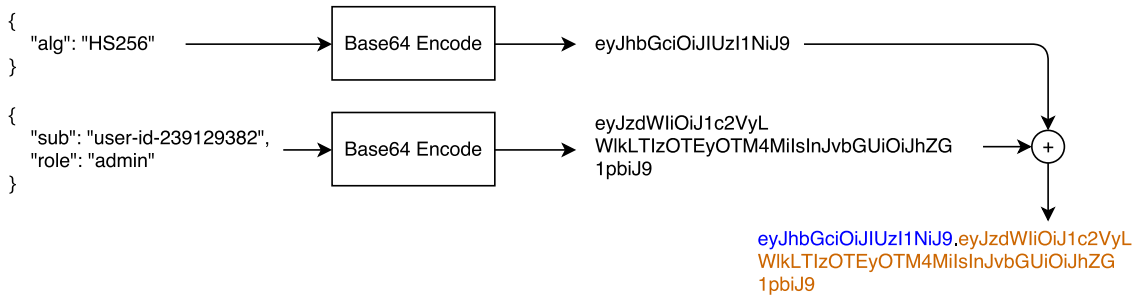

Figure 3.1: Compact Unsecured JWT Generation

3.4.1 Sample Code

```
// URL-safe variant of Base64
function b64(str) {
    return new Buffer(str).toString('base64')
                          .replace(/=/g, '')
                          .replace(/\+/g, '-')
                          .replace(/\//g, '_');
}

function encode(h, p) {
    const headerEnc = b64(JSON.stringify(h));
    const payloadEnc = b64(JSON.stringify(p));
    return `${headerEnc}.${payloadEnc}`;
}
```

The full example is in file `coding.js` of the accompanying sample code.

3.5 Parsing an Unsecured JWT

To arrive at the JSON representation from the compact serialization form, perform the following steps:

1. Find the first period "." character. Take the string before it (not including it.)

2. Decode the string using the Base64-URL algorithm. The result is the JWT header.
3. Take the string after the period from step 1.
4. Decode the string using the Base64-URL algorithm. The result is the JWT payload.

The resulting JSON strings may be "prettified" by adding whitespace as necessary.

3.5.1 Sample Code

```
function decode(jwt) {
    const [headerB64, payloadB64] = jwt.split('.');
    // These supports parsing the URL safe variant of Base64 as well.
    const headerStr = new Buffer(headerB64, 'base64').toString();
    const payloadStr = new Buffer(payloadB64, 'base64').toString();
    return {
        header: JSON.parse(headerStr),
        payload: JSON.parse(payloadStr)
    };
}
```

The full example is in file `coding.js` of the accompanying sample code.