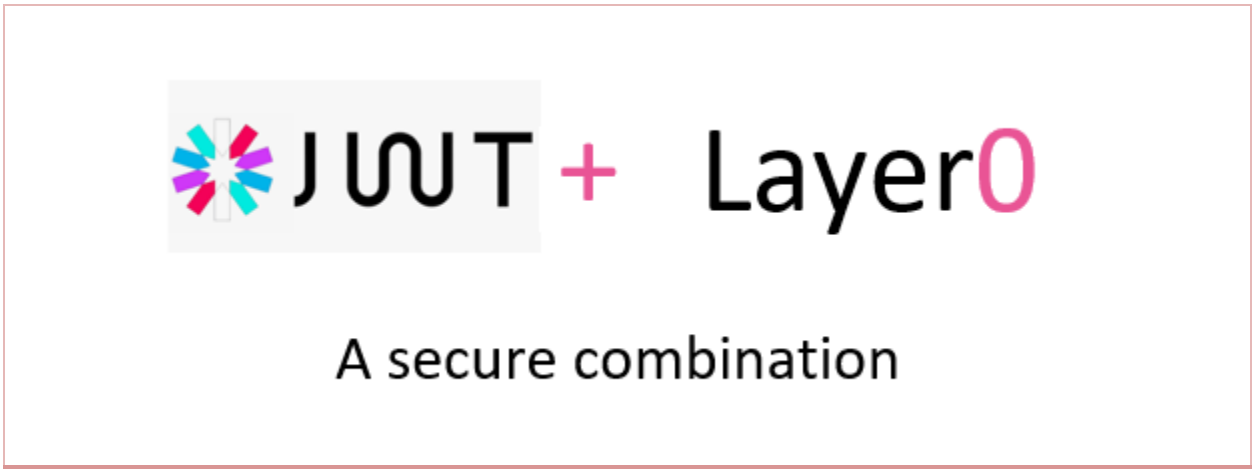

Securing access to your content is essential in today's world of sophisticated hackers. Layer0's JWT capabilities allow you to safeguard your content like never before.

Layer0's JSON Web Tokens (JWT) capability provides an easy-to-use means for safeguarding access to your content. We'll give you a brief background on JWT, then explain how you can use our capabilities to safeguard access to your content.

So, what is JWT anyway?

Well, let's start unpeeling that onion. JWT is an open standard (rfc7519) message **format** for authorization and information exchange.

Note: JWT is not a protocol, nor does it provide content encryption.

Tokens are:

- Compact and self-contained: A token is represented in JSON, a terse message format.
- URL-safe: Because a token may contain non-web safe characters, it is URL-encoded.
- Secure: The token is signed either asymmetrically with a public/private key, or symmetrically with a secret. This guaranties the token was not modified in transit.

Although tokens can also be encrypted, they seldom are because web clients need to directly read and interpret the token. Also, browsers would need to store the encryption secret but don't have a secure local storage capability.

Note: Layer0 requires that the signature be signed with a secret.

Tokens contain all user data in the payload part. Tokens are sessionless and not stored on the server, only in the client.

JWT Structure

Let's dive a little deeper and discuss the structure of a JWT. JWT contains three parts separated by a "." character: a header, payload, and signature. Each part is a base-64 encoded string of characters and numbers.

- Header: Metadata that describes the token type and the algorithm used to generate or validate the signature.
- Payload: Contains the token's claims or claim set. The word claim is a general security term that refers to assertions the token-provider makes about a person or other entity that requests access to your resources.
- Signature: Formed by signing the encoded header and encoded payload using the algorithm in the header.

Note: The three types of claims are:

- Registered: Defined by the JWT specification. See rfc7519 for a complete list.
- Public: Custom claims that anyone can define.
- Private: Claims, neither registered, not public that are exclusive to parties that agree to use them and have unique meaning to the parties.

So, all that was a lot to unpack. We'll provide a sample to help you understand.

Example Structure

Here's a JWT sample we will discuss. We've broken the content over three lines for readability. You will also note that each part ends with a "."

```
eyJhbGciOiJIMjU2IiwgInR5cCI6IkpXVCJ9.
eyJzdWIiOiI1YTg5OTEyZDdlZiIsICJuYW1lIjoiTmF5YSBTaGFybWEiLCAiaWF0
IjoiMTY1MzMyNjk3NSJ9.
b6abb1b360590a1b897ffc388d40141bca287a3fd1f984ddef0ec135ce5efffc
```

If you were to base-64 decode a token's parts, you would see something like the following.

Header

The header contains the signing algorithm and the token type:

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

Payload

The payload contains the claims:

```
{"sub":"5a89912d7ef", "name":"Naya Sharma", "iat":"1653326975"}
```

In this case, the claims are:

- `sub`: The subject of the token. It often represents a user with email, user ID, and so on.
- `name`, `iat`: The subject's name and the UNIX epoch time when the token was created.

Signature

If you try to decode the signature, you will get undisplayable characters because it is simply binary data, so we won't present a sample.

Tokens, Cookies, and Headers

Depending on the authentication provider, tokens are returned either in a cookie or in the `Bearer` token of the `Authorization` header. If returned in a cookie, the cookie name usually varies depending on the provider.

What Can Layer0 Do for You?

If your server uses JWT, you can use Layer0's capability for checking JWT tokens from entities that want to authorize with your server and request your content.

Far from simply blocking or allowing access, you can combine our JWT verification with our routing capabilities and do some amazing things. For example, if a user attempts to access resource A but the user's token has expired, you can route the user to your login page, then after successful login, automatically direct the user back to resource A.

Awesome!

An Example

Now that we know what JWT is and how Layer0 can help, let's dive into an example.

Layer0 exposes the `verifyJwt` method, which combined with our routing capabilities, allows you to accept and redirect content requests depending on the state of the token. Here is a basic example:

```
new Router()
  .match('/protected', ({ verifyJWT }) => {
    verifyJwt({
      algo: 'HS256',
      secret: superSecret,
      header: 'Authorization',
      cookie: 'next-auth.session-token',
      redirectExpiredAbsent: '/expired',
      redirectInvalid: '/invalid',
      returnUrlParamName: 'productListings'
    })
  )
```

Let's break down that example. First, we will describe some of the high-level items, then do a deep-dive into `verifyJwt`and its properties.

High-Level Items

- Router: A Layer0 class and capability that allows you to act upon requests for your content based on the content path. The Router class has quite a few methods, but we will focus just on the `match` method and how to use it for JWT verification. (If you want to learn more, see our API Router documentation.)
- match: A method on the Router class that allows you to define a route and how to handle requests for its content. Contains two arguments: the path and a handler; in this case the handler is the `verifyJWT` method.

  You'll notice the sample uses ES6 arrow function notation.

Deep-Dive Inspection

The `verifyJwt` method takes a configuration object with several properties.

Property	Description
algo	The algorithm that was used to sign the token.
secret	The secret that was used to sign the token.
header and cookie	The header or cookie to extract the token from. Note that `header` and `cookie` are mutually exclusive, we've included them for a sample usage. If you include both, `verifyJwt` will non deterministically set the precedence of one over the other.
redirectExpiredAbsent	The redirect URL to use when the JWT is expired or absent. In this case, requests are redirected to `/expired`.
redirectInvalid	The redirect URL to use when the JWT is invalid (bad signature, invalid claim-matching criteria). In this case, requests are redirected to `/invalid`.
returnUrlParamName	When redirecting to the `redirectExpiredAbsent` or `redirectInvalid` URLs, the value of this option is added as a query/search parameter to the redirect URL as the original URL, for example to return the user to the page that triggered the redirect to login. In this case, the redirects will be: `/expired?returnUrlParamName=productListings` and `/invalid?returnUrlParamName=productListings`

The descriptions here are just to get you started. For additional details about `verifyJWT` options, please see our API Interface VerifyJwtOptions docs.