Why Can't I Just Send JWTs Without OAuth?

POSTED BY KRISTOPHER SANDOVAL | NOVEMBER 14, 2017 SECURITY

A **JSON Web Token** or **JWT** is an extremely powerful standard. It's a signed JSON object; a compact token format often exchanged in HTTP headers to encrypt web communications.

Because of its power, JWTs can be found driving some of the largest modern API implementations. For many, the JWT represents a great solution that balances weight with efficiency, and as such, it's often a very attractive standard to adopt for **API security**.

However, **a JWT should not be viewed as a complete solution**. Unfortunately, it seems that there are some significant misunderstandings as to what a JWT is, and how exactly it functions. In many situations, depending on JWTs alone can be extremely dangerous.

One of the most common questions about using JWTs is: *Why can't I send JWTs without OAuth?* Today, we're going to answer that very question. We'll define what a JWT actually is, how it functions, and why adopting it in isolation is dangerous.

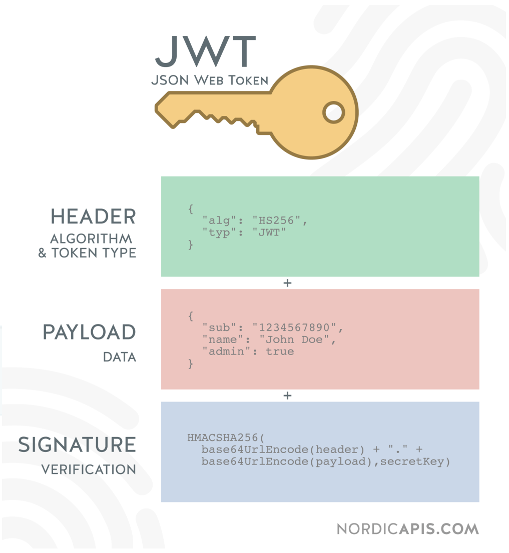
What is a JWT?

Before we address why utilizing JWTs alone is insecure, we must define **what a JWT actually is**. This is because JWTs are often conflated with the additional protocols and systems surrounding them, meaning that the JWT design concept has been bolstered beyond the actual definition of the object itself.

JWT is an open standard defined by RFC 7519. The JWT is considered by its authors to be a "compact and self-contained way for securely transmitting information between parties as a JSON object." The JWT itself is composed of a **Header**, a **Payload**, and a **signature** that proves the integrity of the message to the receiving server.

Content encoded inside a JWT is **digitally signed**, either using a secret utilizing the HMAC algorithm, or by leveraging the Public Key Infrastructure (**PKI**) model with a private/public RSA configuration. While this does lend a certain amount of integrity protection, it does not specifically guarantee *security* — we will discuss this at greater length in just a moment, but it should be understood that **a JWT is an encoding format, and only an encoding format**.

JWTs are loved because they are small, and lend themselves to efficient transport as part of a URL, as part of the POST parameter, or even within the HTTP header. More lightweight transport options exist, and further extensions of the concept exist; **CWT** is a great example, utilizing **CBOR**, or Concise Binary Object Representation, to even further reduce the size of the package and improve efficiency.


JWT
JSON Web Token
HEADER
ALGORITHM & TOKEN TYPE
{ "alg": "HS256", "typ": "JWT" }
+
PAYLOAD
DATA
{ "sub": "1234567890", "name": "John Doe", "admin": true }
+
SIGNATURE
VERIFICATION
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload),secretKey)
NORDICAPIS.COM

The main benefit (and perhaps the main drawback from a security standpoint) of the JWT standard is that the encoded package is self-contained. The JWT package contains everything the system would need to know about the user, and as such, can be delivered as a singular object.

Also Read: Using Open Standards Is Critical to API Longevity

The Dangers of a Lone Solution

JWTs are powerful — there's simply no denying that. Unfortunately, many developers seem to think that the JWT is more than an encoding methodology, but a complete and secure implementation. This is often because JWTs are typically paired with a proper protocol and encryption standard in the wild — but this is a conscious choice, not the result of an automatic security due to the structure of the JWT itself.

A JWT is only secure when it's used in tandem with encryption and transport security methodologies. JWT is a great encoding methodology, but it's not a holistic security measure. Without additional protocols backing it up, a JWT is nothing more than an admittedly lightweight and slightly more secure API key.

For an example of this insecurity, let's look at a common use case. A web API serves as the backend to a web application, and when the user generates a JWT, it is stored as an HTML5 data storage element. This is done so as to aid in the utilization of the API over multiple gateways and functions.

In this common situation, the issue is that the JWT is essentially exposed for common use. The JWT is digitally signed, which assures a certain amount of guaranteed integrity. The server itself is also set to reject any JWT with a manipulated Header, Payload, or Signature component, and as such, can reject a modified JWT token. That being said, the token doesn't need to be modified in order to breach security. In theory, an attacker could take that token and use it in a sort of replay attack, getting resources that they do not have authorization to have.

While this type of attack can be somewhat mitigated through the use of expiration dates, this does nothing for **man-in-the-middle attacks**. In the MITM attack scheme, the expiration does not matter, as the attack is initiated live as a middleman.

These issues all arise from the simple fact that JWTs are a mechanism for transferring data — not for securing it.

Read Our Deep Dive Into OAuth and OpenID Connect

Securing JWTs

JWTs are self-contained solutions containing everything the server needs to know about **who** the user is, **what** they need, and what they're **authorized** to do. Accordingly, they're great for stateless authentication, and work well with such methods geared for stateless environments.

While there are a number of third party solutions and implementations of stateless authentication, the fact is that what you'd essentially be creating is a **bearer token** or, alternately, an **access token**.

That's ok, and in fact, what we want to do, but this raises a simple question — if we are indeed creating such a bearer token, why not use the built-in functionality of the **OAuth** schema designed specifically to work with JWTs? There's already a great deal of built-in security functionality in the OAuth specification that's specifically engineered to support the JWT, so using **external solutions** — often the second question after *why can't I just sent JWTs without OAuth* — is somewhat nonsensical.

If we utilize the OAuth 2.0 Bearer Token Usage standard under RFC 6750, which incorporates authorization headers, we can essentially create JWTs that would be recognized and specially treated by a wide variety of devices, from HTTP proxies to servers. We would thereby reduce data leakage, unintended storage of requests (as displayed above), and enable transport over something as simple as HTTPS.

Also Learn: How to Handle Batch Processing With OAuth 2.0

Proper JWT Utilization

While it's important to secure your JWTs, it's also important to state what the proper utilization of a JWT within the OAuth schema would look like. While a JWT can serve many functions, let's take a look at a common use case in the form of the **access token**. Both OAuth 2.0 and OpenID Connect are vague on the type of `access_token`, allowing for a wide range of functions and formats. That being said, the utilization of a JWT as that token is quite ubiquitous, for the benefits in efficiency and size already noted.

An access token is, in simple terms, is a token that is used by the API to make requests on behalf of the user who requested the token. It is part of the fundamental authorization mechanism within OAuth, and as such, **confidentiality** and **integrity** are extremely important. In order to generate an access token, an **authorization code** is required. All of the elements of this code are also extremely important to keep confidential and secure.

Accordingly, a JWT fits this role almost perfectly. Because of the aforementioned standards that allow for transmission over HTTPS, the JWT can contain all of the information needed to generate the access token. Once the token is generated, it can likewise be kept in JWT form as what is called a **self-encoded access token**.

The key benefit of handling the encoding of the access token in this way in the OAuth 2.0 schema is that applications don't have to understand your access token schema — all of the information is encoded within the token itself, meaning that the schema can change fundamentally without requiring the clients to be aware, or even affected, by such changes.

Additionally, the JWT is great for this application because of the wide range of libraries that offer functionality such as expiration. A good example of this would be the Firebase PHP-JWT library, which offers such expiration functionality.

The following code is taken from the official OAuth documentation, and demonstrates what such an implementation would look like during encoding:

```php
<?php
use \Firebase\JWT\JWT;

# Define the secret key used to create and verify the signature
$jwt_key = 'secret';

# Set the user ID of the user this token is for
$user_id = 1000;

# Set the client ID of the app that is generating this token
$client_id = 'https://example-app.com';

# Provide the list of scopes this token is valid for
$scope = 'read write';

$token_data = array(

  # Subject (The user ID)
  'sub' => $user_id,

  # Issuer (the token endpoint)
  'iss' => 'https://' . $_SERVER['PHP_SELF'],

  # Audience (intended for use by the client that generated the token)
  'aud' => $client_id,

  # Issued At
  'iat' => time(),

  # Expires At
  'exp' => time()+7200, // Valid for 2 hours

  # The list of OAuth scopes this token includes
  'scope' => $scope
);
$token_string = JWT::encode($token_data, $jwt_key);
```

Such a mechanism would result in an **encoded string** that looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.eyJzdWIiOjEwMDAsImlzcyI6Imh0dHBzOi8
vYXV0aG9yaXphdGlvbi1zZXJ2ZXIuY29tIiw
iYXVkIjoiaHR0cHM6Ly9leGFtcGxlLWFwcC5
jb20iLCJpYXQiOjE0NzAwMDI3MDMsImV4cCI
6MTQ3MDAwOTkwMywic2NvcGUiOiJyZWFkIHd
yaXRlIn0.zhVmPMfS3_Ty4qUl5ZMh4TipXsU
CSH0mHzb4P_Ijhxs
```

Caveats

Of course, as with any security implementation, there are caveats to consider. In the case of the JWT as a self-encoded authorization solution, replay attacks should be considered. While adopting proper encryption methodologies should negate many of those issues, the fact is that the issue is still fundamental to the concept as a whole, and should probably be addressed as a possibility rather than an impossible threat.

Accordingly, **caching** the authorization code for the lifetime of the code is the suggested solution from OAuth itself. By doing this, code can be verified against the known cached code for validity and integrity, and once the expiration date is reached, automatically rejected for date reasons.

It should also be noted that, due to the nature of the JWT, once an authorization code is issued, the JWT is self-contained — as such, it cannot technically be invalidated, at least in its most basic configuration. The JWT is designed to not hit the database for every verification, and when using a global secret, the JWT is valid until expiration.

There are a few ways around this, such as adding a **counter** in the JWT that increments upon certain events (such as role change, user data change, etc). This of course results in database pollingfor each request, but the amount of data being checked is miniscule enough to make any processing increase somewhat negligible.

Additionally, at least in theory, you could use sections for specific functions, domains, and scopes, and change that secret when a breach is discovered. While this would affect more users than admins would like, it does have the effect of instituting revocation.

That being said, proper utilization of the JWT should make this largely a non-issue, as the user still has to provide a certain amount of secret information over an encrypted channel, and as such, should already be "vetted" or controlled.

How to Control User Identity Within Microservices

Don't Leave JWT All Alone

The simple fact is that JWTs are a great solution, especially when used in tandem with something like OAuth. Those benefits quickly disappear when used alone, and in many cases can result in **worse overall security**.

That being said, adopting the proper solutions can mitigate many of these threats, resulting in a more secure, efficient system. The first step to securing your JWT is to understand what it's not — a JWT is an encoding method, not an encryption or transport security method, and as such, is only part of the puzzle.

What is your best solution for securing JWTs? Let us know in the comment section below.