Why You Should Always Use Access Tokens to Secure an API

By Maria Paktiti

Archived: 2026-04-06 00:03:50 UTC

TL;DR: There is much confusion on the Web about the differences between the OpenID Connect and OAuth 2.0 specifications, and their respective tokens. As a result many developers publish insecure applications, compromising their users security. The contradicting implementations between identity providers do not help either.

This article is an attempt to clear what is what and explain why you should always use an access token to secure an API, and never an ID token.

Two complementary specifications

OAuth 2.0 is used to **grant authorization**. It enables you to authorize the Web App A to access your information from Web App B, without sharing your credentials. It was built with *only* authorization in mind and doesn't include any authentication mechanisms (in other words, it doesn't give the Authorization Server any way of verifying who the user is).

OpenID Connect builds on OAuth 2.0. It enables you, the user, to verify your identity and give some basic profile information, without sharing your credentials.

An example is a to-do application, that lets you log in using your Google account, and can push your to-do items as calendar entries, at your Google Calendar. The part where you authenticate your identity is implemented via OpenID Connect, while the part where you authorize the to-do application to modify your calendar by adding entries, is implemented via OAuth 2.0.

> OpenID Connect is about who someone is. OAuth 2.0 is about what they are allowed to do.

You may notice the *"without sharing your credentials"* part, at our definitions of the two specifications earlier. What you do share in both cases, is tokens.

OpenID Connect issues an identity token, known as `id_token`, while OAuth 2.0 issues an `access_token`. Learn more about OIDC with the free OpenID Connect Handbook:

Learn about the de facto standard for handling authentication in the modern world.

Download the free ebook

How to use each token

The `id_token` is a JWT and is meant for the client only. In the example we used earlier, when you authenticate using Google, an `id_token` is sent from Google to the to-do application, that says who you are. The to-do application can parse the token's contents and use this information, like your name and your profile picture, to customize the user experience.

Note: You must never use the info in an `id_token` unless you have validated it! For more information refer to: How to validate an ID token. For a list of libraries you can use to verify a JWT refer to JWT.io.

The `access_token` can be any type of token (not necessarily a JWT) and is meant for the API. Its purpose is to inform the API that the bearer of this token has been authorized to access the API and perform specific actions (as specified by the `scope` that has been granted). In the example we used earlier, after you authenticate, and provide your consent that the to-do application can have read/write access to your calendar, an `access_token` is sent from Google to the to-do application. Each time the to-do application wants to access your Google Calendar it will make a request to the Google Calendar API, using this `access_token` in an HTTP `Authorization` header.

Note: Access Tokens should be treated as opaque strings by clients. They are only meant for the API. Your client should not attempt to decode them or depend on a particular `access_token` format.

How NOT to use each token

Now that we saw what these tokens can be used for, let's see what they cannot be used for.

- **An `access_token` cannot be used for authentication.** It holds no information about the user. It cannot tell us if the user has authenticated and when.
- **An `id_token` cannot be used for API access.** Each token contains information on the intended audience (recipient). According to the OpenID Connect specification, the audience (claim `aud`) of each `id_token` must be the `client_id` of the client making the authentication request. If it isn't you shouldn't trust the token. An API, on the other hand, expects a token with the audience set to the API's unique identifier. So unless you are in control of both the client and the API, sending an `id_token` to an API will not work.

> Furthermore, the `id_token` is signed with a secret that is known to the client (since it is issued to a particular client). This means that if an API were to accept such token, it would have no way of knowing if the client has modified the token (to add more scopes) and then signed it again.

Compare Tokens

To better understand what we just read, let's look at the contents of example tokens.

The (decoded) contents of an `id_token` look like the following:

```
 {
  "iss": "http://${account.namespace}/",
  "sub": "auth0|123456",
  "aud": "${account.clientId}",
  "exp": 1311281970,
  "iat": 1311280970,
  "name": "Jane Doe",
  "given_name": "Jane",
  "family_name": "Doe",
  "gender": "female",
  "birthdate": "0000-10-31",
  "email": "janedoe@example.com",
  "picture": "http://example.com/janedoe/me.jpg"
}
```

This token is meant for **authenticating the user to the client**. Note that the audience (`aud` claim) of the token is set to the client's identifier, which means that only this specific client should consume this token.

For comparison, let's look at the contents of an `access_token` :

```
 {
  "iss": "https://${account.namespace}/",
  "sub": "auth0|123456",
  "aud": [
    "my-api-identifier",
    "https://${account.namespace}/userinfo"
  ],
  "azp": "${account.clientId}",
  "exp": 1489179954,
  "iat": 1489143954,
  "scope": "openid profile email address phone read:appointments email"
}
```

This token is meant for **authorizing the user to the API**. As such, it is completely opaque to clients, meaning that a client should not care about the contents of this token, decode it or depend on a particular token format. Note that the token does not contain any information about the user itself besides their ID (`sub` claim), it only contains authorization information about which actions the client is allowed to perform at the API (`scope` claim).

Since in many cases it is desirable to retrieve additional user information at the API, this token is also valid for calling the `/userinfo` API, which will return the user's profile information. So the intended audience (`aud` claim) of the token is either the API (`my-api-identifier`) or the `/userinfo` endpoint (`https://${account.namespace}/userinfo`).

Recommended resources for further reading

- Access Token
- ID Token
- The problem with OAuth for Authentication
- The OAuth 2.0 Authorization Framework
- OAuth 2.0 Overview
- OpenID Connect Overview

Securing Applications with Auth0

Are you building a B2C, B2B, or B2E tool? Auth0, can help you focus on what matters the most to you, the special features of your product. Auth0 can improve your product's security with state-of-the-art features like passwordless, breached password surveillance, and multifactor authentication.

We offer a generous **free tier** so you can get started with modern authentication.
