Illustration created by Sherman Fuchs.

OPEN STANDARDS

Learn about refresh tokens and how they fit in the modern web. Get a working sample of how to implement it with NodeJS

Sebastian Peyrott
Software Developer

Last Updated On: February 08, 2020

In this post we will explore the concept of refresh tokens as defined by OAuth2. We will learn why they came to be and how they compare to other types of tokens. We will also learn how to use them with a simple example. Read on!

Update: at the moment this article was written Auth0 had not gone through OpenID Connect certification. Some of the terms used in this article such as `access token` do not conform to this spec but do conform to the OAuth2 specification. OpenID Connect establishes a clear distinction between `access tokens` (used by resource servers to authorize or deny requests) and the `id token` (used by client applications to identify users).

Introduction

Modern authentication and/or authorization solutions have introduced the concept of *tokens* into their protocols. Tokens are specially crafted pieces of data that carry just enough information to either **authorize the user to perform an action**, or allow a client to **get additional information about the authorization** process (to then complete it). In other words, tokens are pieces of information that allow the authorization process to be performed. Whether this information is readable or parsable by the client (or any party other than the authorization server) is defined by the implementation. The important thing is: the client gets this information, and then uses it to **get access to a resource**. The JSON Web Token (JWT) spec defines a way in which common token information may be represented by an implementation.

A short JWT recap

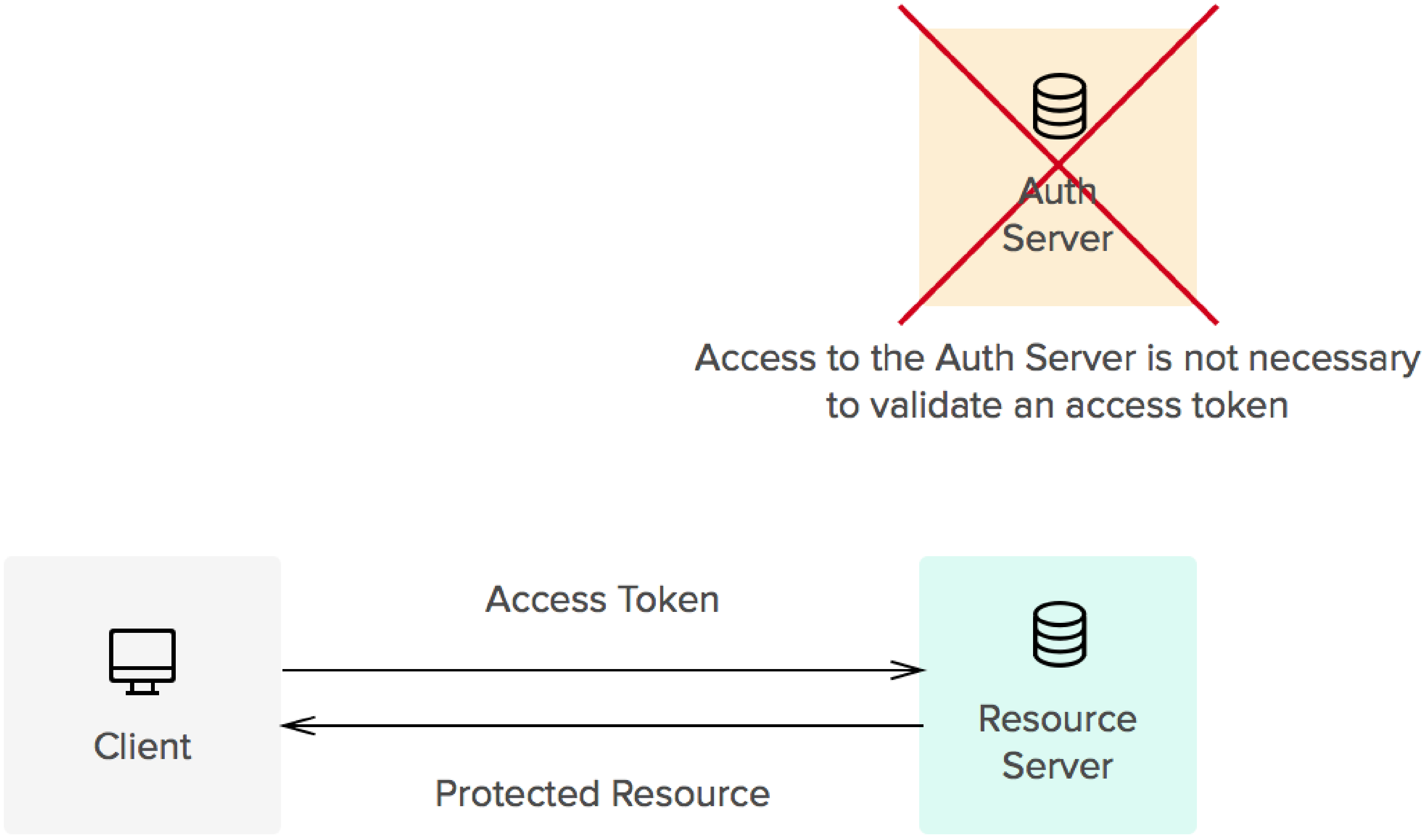
JWT defines a way in which certain common information pertaining to the process of authentication/authorization may be **represented**. As the name implies, the data format is **JSON**. JWTs carry certain **common fields** such as subject, issuer, expiration time, etc. JWTs become really useful when combined with other specs such as JSON Web Signature (JWS) and JSON Web Encryption (JWE). Together these specs provide not only all the information usually needed for an authorization token, but also a means to **validate the content** of the token so that it cannot be tampered with (JWS) and a way to **encrypt information** so that it remains **opaque** to the client (JWE). The simplicity of the data format (and its other virtues) have helped JWTs become one of the most common types of tokens.

If you're interested in learning more about how to implement JWTs, click the link below and we'll email you our in-depth **JWT Handbook** for free!

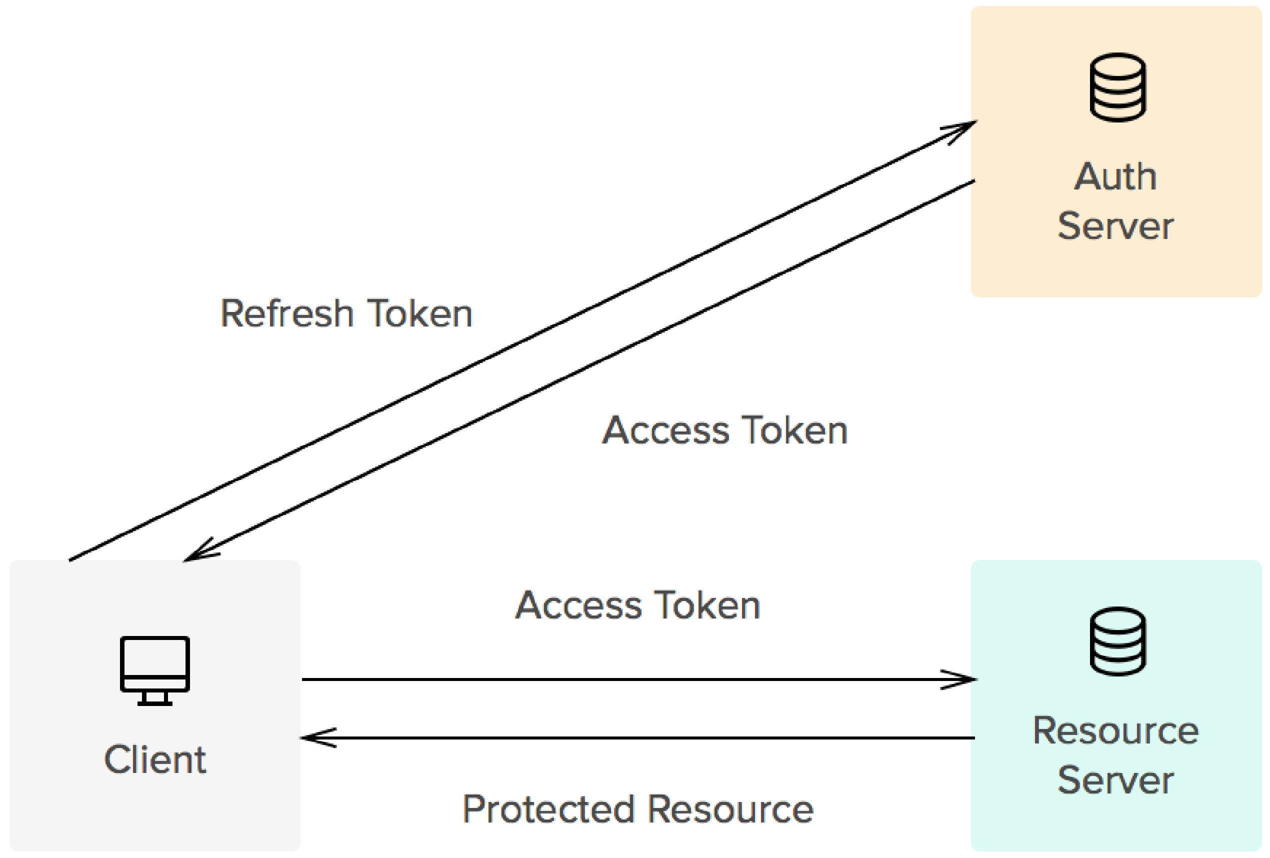
Token types

For the purposes of this post, we will focus on the two most common types of tokens: *access tokens* and *refresh tokens*.

- **Access tokens** carry the necessary information to access a resource directly. In other words, when a client passes an access token to a server managing a resource, that server can use the information contained in the token to decide whether the client is authorized or not. Access tokens usually have an expiration date and are short-lived.


Auth Server
Access to the Auth Server is not necessary to validate an access token
Client
Access Token
Protected Resource
Resource Server

- **Refresh tokens** carry the information necessary to get a new access token. In other words, whenever an access token is required to access a specific resource, a client may use a refresh token to get a new access token issued by the authentication server. Common use cases include getting new access tokens after old ones have expired, or getting access to a new resource for the first time. Refresh tokens can also expire but are rather long-lived. Refresh tokens are usually subject to strict storage requirements to ensure they are not leaked. They can also be blacklisted by the authorization server.



Whether tokens are opaque or not is usually defined by the implementation. Common implementations allow for **direct authorization checks against an access token**. That is, when an access token is passed to a server managing a resource, the server can read the information contained in the token and decide itself whether the user is authorized or not (no checks against an authorization server are needed). This is one of the reasons tokens must be signed (using JWS, for instance). On the other hand, refresh tokens usually require a check against the authorization server. This split way of handling authorization checks allows for three things:

1. Improved access patterns against the authorization server (lower load, faster checks)
2. Shorter windows of access for leaked access tokens (these expire quickly, reducing the chance of a leaked token allowing access to a protected resource)
3. Sliding-sessions (see below)

Sliding-sessions

Sliding-sessions are sessions that expire after a **period of inactivity**. As you can imagine, this is easily implemented using access tokens and refresh tokens. When a user performs an action, a new access token is issued. If the user uses an expired access token, the session is considered inactive and a new access token is required. Whether this token can be obtained with a refresh token or a new authentication round is required is defined by the requirements of the development team.

Security considerations

Refresh tokens are **long-lived**. This means when a client gets a refresh token from a server, this token must be **stored securely** to keep it from being used by potential attackers. If a refresh token is leaked, it may be used to obtain new access tokens (and access protected resources) until it is either blacklisted or it expires (which may take a long time). Refresh tokens must be issued to a single authenticated client to prevent use of leaked tokens by other parties. Access tokens must be kept secret, but as you may imagine, security considerations are less strict due to their shorter life.

"Access tokens must be kept secret, security considerations are less strict due to their shorter life."

Tweet This

Use Refresh Tokens in Your Auth0 Apps

At Auth0 we do the hard part of authentication for you. Refresh tokens are not an exception. Once you have setup your app with us, follow the docs here to learn how to get a refresh token.

Conclusion

Refresh tokens improve security and allow for reduced latency and better access patterns to authorization servers. Implementations can be simple using tools such as JWT + JWS. If you are interested in learning more about tokens (and cookies), check our article here.

> You can also check the Refresh Tokens landing page for more information.

Get the free ebook

Sebastian Peyrott

SOFTWARE DEVELOPER

I am software developer with a keen interest in open-source technologies, Linux, and native development. I've worked on many different platforms Android, iOS, Win32, Linux, FreeRTOS, the Web, and others. I've gone through the whole stack and I enjoy learning and using the latest technologies. I now work as a full-stack developer at Auth0.

VIEW PROFILE ▸

More like this

IONIC

Ionic 2 Authentication: How to Secure Your Mobile App with JWT

Follow the conversation

69 Comments   Auth0 Blog   Disqus' Privacy Policy   1 Login

Recommend 9   Tweet   Share   Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Sumit Kumar • 4 years ago
Is it OK to use sliding sessions when client app is making ajax polls and getting new tokens every time. I think in this case token will not expire till window is not closed for at least a duration of refresh token expiry time.
18 ^ | ˅ 2 • Reply • Share ›

Vladimir Stanković • 5 years ago
Let's say that client application that is interacting with API is web application.

Is there any reason why would we need Refresh token in this use case?

12 ^ | ˅ 1 • Reply • Share ›

Sebastian Peyrott ➦ Vladimir Stanković • 5 years ago

Yes, for various reasons:
- Keeping the access token short-lived will minimize damage if transport security is compromised.
- Access tokens can be requested for specific resources. A leaked token may not allow for full access to a system.
- Access tokens usually need less hits to the database, thus reducing latency (usually important for Web APIs).

2 ^ | ˅ 1 • Reply • Share ›

Vladimir Stanković ➦ Sebastian Peyrott • 5 years ago

I understand the benefits of having short-lived access tokens.

Still, I have trouble understanding how we benefit from long-lived refresh token in case of web application.

If transport security is compromised, like you say, someone would easily steal the refresh token as well, right? Damage would be even greater than stealing access token.

In case that short-lived access token is expired, I use that same expired short-lived token to ask for a new one. Additional constraint is that last valid short-lived access token can be used to ask new access token only within specified time-frame (example 30 minutes).

So, to conclude, I have short-lived access token that expire after 5 minutes. If that token expire, the same token is issued to ask for a new one. If 30 minutes time-frame for token refresh is expired user is redirected to login page again.

16 ^ | ˅ 1 • Reply • Share ›

oreqizer ➦ Vladimir Stanković • 4 years ago

refresh tokens can be revoked server-side. if your refresh token is leaked, simply set is as invalid in your auth server

1 ^ | ˅ • Reply • Share ›

Vladimir Stanković ➦ oreqizer • 4 years ago • edited

@oreqizer Completely agree with you.

Token revocation can be achieved, even without refresh token, for example by versioning of access tokens itself.

Refresh token can provide us fine grained token revocation policies. Like, for example, having opportunity to issue/revoke refresh token for each of the user devices.

However if we want more security features added - our server is not stateless any more. And stateless tokens is something that is often an argument in JWT vs Sessions/Cookies debate (even in this article).

3 ^ | ⌄ • Reply • Share ›

Sebastian Peyrott ➦ Vladimir Stanković • 4 years ago

It is, as long as you rely on the access_token as the stateless element. Refresh tokens should always be checked against a revocation list.

2 ^ | ⌄ 1 • Reply • Share ›

Marcello Kad Cadoni ➦ Vladimir Stanković • 4 years ago

If you handle Sessions like a resource the system will be still stateless.

^ | ⌄ • Reply • Share ›

Kerem Baydoğan ➦ Marcello Kad Cadoni • 4 years ago

stateless but slow. hitting the database on every JWT validation is not something you would want.

2 ^ | ⌄ • Reply • Share ›

mikestead ➦ Kerem Baydoğan • 4 years ago

I'm going to call this one out as FUD. Redis or the like is hardly slow. Security should be the main consideration here.

A refresh token is like a password, would you store a password in the browser?

Here's a more correct answer from Auth0 which recommends not issuing a refresh token to a browser, but having a long expiry on the access token. https://stackoverflow.com/q...

Do you see the security implication of this approach? A JWT that can live for a week and can't be revoked, unless you start storing state?

Another point, go look at FB, Netflix etc and tell me how many of these big sites use JWTs and refresh tokens vs session cookies and opaque tokens? There's a reason for this.

http://cryto.net/~joepie91/

http://cryto.net/~joepie91/...

4 ^ | ⌄ • Reply • Share ›

Martín ➦ Sebastian Peyrott • 3 years ago • edited

I think you don't need refresh tokens at all. You can create a JWT access token and save it into the database with a flag "refreshed" in false. You set expired time for example to 30 minutes. After 30 minutes you get 401 [Unauthorized], so you call a RefreshToken method with the expired access token. The RefreshToken method check if the access token exists in the database and if the flag "refreshed" is set to false. If this is OK, you set the flag to true and you create a new access token.
On the other hand, if you call the RefreshToken method, you check the access token in the database and the flag "refreshed" is set to true (the token has been refreshed), you get 401 [Unauthorized] and you need to login the user again.

3 ^ | ⌄ 3 • Reply • Share ›

Marcel Német ➦ Martín • 3 years ago

is that not the same as having a refresh token? but you use access token as an access token and as refresh token at the same time.

1 ^ | ⌄ • Reply • Share ›

Leo Lei ➦ Marcel Német • 3 years ago • edited

Not the same, because now all tokens have short-lived expiration. (just pointing out, not saying that it's good or bad)

^ | ⌄ • Reply • Share ›

Leo Lei ➦ Leo Lei • 3 years ago

I think it can be thought of as a combination of the tokens like **@Marcel Német** said.

Before it expires, it serves as an access token.
After it expires, it serves as a refresh token.
However, this refresh token is different from the article because the exp doesn't matter as long as the 'refreshed' flag is false.

So the difference is that the refresh token in this article has an expiration (long-lived), but can be used as many times as possible before exp, and the refresh token suggested by **@Martín** is only valid once, but it could be potentially longer-lived.

This is kind of interesting. I'm curious what other security implications it might have.
1 ^ | ⌄ • Reply • Share ›

Josephat Waweru ➦ Vladimir Stanković • 3 years ago
yea, imagine the session token lasts for 5 hours and a user is still logged in, do you have to forcefully log them out??
^ | ⌄ • Reply • Share ›

Michael Yuen • 4 years ago
Hi,
"Refresh tokens are usually subject to strict storage requirements to ensure they are not leaked."

Does this mean to store the token into Cookies? or Encrypted in local Storage?

Michael
5 ^ | ⌄ • Reply • Share ›

Adam Reis ➦ Michael Yuen • 4 years ago • edited
Probably one of the safer ways to deal with refresh tokens is to store them in a secure https only cookie with a restricted path, e.g. `/oauth/token`, and have the server that you make requests to (or a proxy server) handle the obtaining of new access tokens. This way, the refresh token can not be leaked via the client app.

You can find an example implementation for Express here https://github.com/meanie/e...
3 ^ | ⌄ • Reply • Share ›

Cameron Moss • 5 years ago • edited
Great post! For React-Native, how would you implement the sliding session? If I'm following you, you need to store both tokens in local and then on open, check to see if the access token works and, if not, refresh both? Is there some middleware that exists in RN-auth0 that can detect a failed call and do this automatically?
2 ^ | ⌄ • Reply • Share ›

Sebastian Peyrott ➦ Cameron Moss • 5 years ago
As I'm not familiar with react-native, I've asked our local react-native guru. He pointed me in the direction of our react-native-lock library (essentially a pluggable log-in screen for your apps). It should handle access_token refreshing automatically:

- https://github.com/auth0/re...
- https://github.com/auth0/re...

Reply • Share ›

Gagana Hg • 4 years ago

how can i refresh tokens?
I have got an error "oauth2client.client.HttpAccessTokenRefreshError:invalid_grant:Invalid JWT:token must be short-lived token and in a reasonable timeframe"

1 • Reply • Share ›

Nguyen Ha Duy Henry • 4 years ago

Thanks for your great post.
Regarding Sliding-sessions: When a user performs an action, a new access token is issued.
Do you mean that every new action from the user will have to re-create a new access token, and client need to update his current access token with the new one?

From your example, I guess you did not perform Sliding-sessions, since
curl 'localhost:3000/secret?
access_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiVlx1MDAxNcKbwoNUwoonbFPCu8KhwrYiLCJpYXQiOjE

Secret area

did not return you any new access token?

thank you :)

1 • Reply • Share ›

Mind ➜ Nguyen Ha Duy Henry • 4 months ago

The call to access secret area only consumes access token and will not produce a new one.
You have to use refresh token to generate new access token explicitly.

Reply • Share ›

jamesrgrinter • 4 years ago

The grant_type=refresh_token step - why would you be wanting/needing to resubmit the original username/password in an Authorization header? (This same mistake/confusing example is in the "Refresh Tokens landing page" too).

1 • Reply • Share ›

Sebastian Peyrott ➜ jamesrgrinter • 4 years ago

For some types of operations, reauthentication is necessary. When requesting a refresh token, it is wise to reauthenticate for higher security.

1 ^ | ⌄ • Reply • Share ›

Chaim Lando ➜ Sebastian Peyrott • 4 years ago

but then why use a refresh token when you have the username and password?

1 ^ | ⌄ • Reply • Share ›

Sebastian Peyrott ➜ Chaim Lando • 4 years ago

The refresh token is meant to prevent reauthentication by reentering credentials. Credentials should never be cached, and should only be managed by the owner (user).

^ | ⌄ • Reply • Share ›

daniel ➜ Sebastian Peyrott • 4 years ago

I think this is a recursive answer.
If "refresh token" is already proving identity of the client for a "long period of time", sending the Basic Authorization header again should not be necessary... or why is it?
Shouldn't the "refresh token" prevent reauthentication per se?
I mean, if the "refresh token" was indeed revoked/expired before making the request, the user *should* enter his credentials to get a new one, right?

^ | ⌄ • Reply • Share ›

Sebastian Peyrott ➜ daniel • 4 years ago • edited

Well, the previous comment said "why use a refresh token when you have the username and password?". The point I was trying to make is that you _don't_ have the username and password, the user has them.

It is also important to note that refresh tokens are not meant to be a replacement for credentials. In fact, refresh tokens can be confined to specific scopes or limits. Credentials, on the other hand, always give whomever holds them the same level of access as the user in question.

1 ^ | ⌄ • Reply • Share ›

Aaron Wright ➜ Sebastian Peyrott • 10 months ago

The confusing part is that the article above uses the username and password and refresh token when getting a new access token. So the username and password must have been stored somewhere to accomplish this, which doesn't seem right.

^ | ˅ • Reply • Share ›

daniel ➦ Sebastian Peyrott • 4 years ago

Ok, this reads more clearly now!
I got stuck on the original question, "why would you be wanting/needing to resubmit the original username/password in an Authorization header".
Indeed it is confusing and needed some clarifications (which i got from this answer).

I understand from Your last answer that the difference for a new "access token" requested with BOTH "user:pass" AND "refresh token" makes it possible to assign different scopes and limits to the newly created "access token", than to a request with only a VALID "refresh token", correct?

^ | ˅ • Reply • Share ›

Sebastian Peyrott ➦ daniel • 4 years ago

Yes, credentials can be used to request any scopes available to the user. Refresh tokens can be confined to issue access tokens for specific scopes. This is up to the developers of the system.

1 ^ | ˅ • Reply • Share ›

Cameron Moss • 5 years ago • edited

For web, it's not as safe to assume a unique user-to-computer relationship like with mobile so login after expiration (10hrs) it makes sense for re-login to be handled through the browser (saved password). So the refresh token is just so that mobile can automatically re-login after 10 hrs is how I see it.

1 ^ | ˅ • Reply • Share ›

Sebastian Peyrott ➦ Cameron Moss • 5 years ago

Indeed, there are pros and cons that need to be weighted for each scenario. IMO, re-authentication is more secure than keeping a refresh token lingering client-side, but this may probe inconvenient from a UX PoV.

^ | ˅ • Reply • Share ›

Cameron Moss ➦ Sebastian Peyrott • 5 years ago

Great, I see that for the convenience you have to be willing to take on the capacity to blacklist the Tokens. It's up to developers to implement the refresh though? as per my first question.

56 ^ | ˅ • Reply • Share ›

Sebastian Peyrott ➦ Cameron Moss • 5 years ago • edited

In general, yes, as long as the authorization server provides an endpoint for direct access_token request. Otherwise you do need to request a refresh token and then use that to get the access token. It is also up to each user to determine what to blacklist. In practice, it is common to blacklist refresh tokens and keep access tokens short lived.

^ | ˅ • Reply • Share ›

Bernhard Schmid • a year ago

Who is the managing part dealing with id Token? In an Oauth connetion I unterstand that a request is sent. If you are authenticated correctly you get an access token and a refresh Token. From the resource Server you will get Access with the access token. If the access token expires the refresh Token will go to the authorization Server to fetch a new access token. Whatif the id token is also expired. Who is responsible for that Workflow? And how? This is not clear to me.

^ | ˅ • Reply • Share ›

Sérgio Danilo • a year ago

Hi,
What's the best way to keep sliding-session active?
Is it refresh token for every call to a resource server endpoint?

^ | ˅ • Reply • Share ›

Javier G. Visiedo • 4 years ago • edited

For some implementations, when asking for a refresh, you also have to provide your client id and secret, again in the body, right?

Is that according to the spec? I assumed that the reason of having a refresh token was to avoid sending your secret again, or?

^ | ˅ • Reply • Share ›

Sebastian Peyrott ➦ Javier G. Visiedo • 4 years ago

This is up to the authentication server but it is valid and required by the spec in certain scenarios:
https://tools.ietf.org/html...

The rationale is that refresh tokens are issued to specific clients. This prevents the use of leaked refresh tokens by different clients. This also means that the client must usually authenticate with the authorization server before being granted a new access token. Do note that client credentials are not user credentials, therefore sending a "client id" and a "secret" is not equivalent to the user inputting their username and password. It is up to the authentication server to choose an authentication method between the client and itself.

Reply • Share ›

Javier G. Visiedo Sebastian Peyrott • 4 years ago

Thanks a lot for the explanation Sebastian, all clear now ;-)

Reply • Share ›

Tal Ben Simhon • 4 years ago

Hi, I would like to get some help, with several issues:
1. In all of this article id-Token is not mentioned. When it is used? it looks like this is the access-Token mentioned here encoded with JWT?!

2. I'm implementing a (proxy)-api-router, which suppose to expose same interfaces to a mobile-IOS and web application as the Auth0-SDK. The IOS app used to work with the id-token and the refresh-token, when I'm doing auth0.oauth.signIn, I do not get the refresh-token back. When I'm using postman /oauth/ro, to do the equivalent to signIn (I believe), I do get it. Why? Are these 2 methods equivalent?

3. What is exactly the flow in native auth0, to refresh a token? is this the /oauth/token? Not working for me, the example you gave here is different from https://auth0.com/docs/api/...

Thanks,
Tal

Reply • Share ›

Sebastian Peyrott Tal Ben Simhon • 4 years ago

1. As Kim mentioned, the id_token is a different type of token not related to either access tokens or refresh tokens. ID tokens encode user information and are usually only meant to be used for display purposes on client-side apps.

2. The Auth0 API has been updated to conform to the OpenID Connect specification. Check the new documentation for the authentication API.

3. You probably want the authorization code flow. With it you can first call the /authorize endpoint to get a code and then the /token endpoint to retrieve a refresh token (plus the associated id_token and access_token).

This article was written before the Auth0 API was updated to the new version, so there may be inconsistencies.

2 ^ | ˅ • Reply • Share ›

Kim Maida ➦ Tal Ben Simhon • 4 years ago
An ID token is also a JWT, but it is not the same as an access token. The ID token is intended for the client side use, whereas the access token is opaque to the client and should only be used to secure a backend API (never decoded on the client). Check out this doc to understand more about this: https://auth0.com/docs/api-...

This blog article is from 2015. Please check out the docs for current flows: https://auth0.com/docs/toke...
1 ^ | ˅ • Reply • Share ›

Marcel • 4 years ago
I like the idea of pushing stateful DB lookups to the Auth server so the Resource server can avoid hitting the auth DB on every single request.

But I want to confirm that since there is complete trust in the Access Token by the Resource server, there is no way to *immediately* do the following:

- Suspend a user account or otherwise revoke access to a resource
- Log out a user from all devices (by user or admin, or password changed)
- Log out securely (an attacker who recovers the Access Token can still access Resources)

The shorter the TTL for the Access Token the less time there is for malicious activity. For many websites and apps this would be an acceptable tradeoff. But for many devs at least one of those bullets would be a deal breaker. If that's true then it would be really helpful to add another disclaimer or at least footnote in this article. A lot of resources link to here.
^ | ˅ • Reply • Share ›

Sebastian Peyrott ➦ Marcel • 4 years ago • edited
There are two ways to mitigate what you describe, with different tradeoffs:
- Keep a revocation list for access tokens and keep the access tokens short lived. This way the revocation list can be cleared quickly (after access token expiration) and might even be able to live in-memory for all services.
- Change the signing or decryption key. This will make all tokens invalid and require reauthentication (or at least refresh token use) of all users, and not just the ones that were confirmed to be compromised.
1 ^ | ˅ • Reply • Share ›

Marcel ➦ Sebastian Peyrott • 4 years ago
Thanks. Changing the signing key feels like a "nuclear option", but it's good to know.

^ | ˅ • Reply • Share ›

codeboy2k ➦ Marcel • 2 years ago • edited

I'm not an Auth0 user but this page is linked to from many places on the web. I thought I would share an option supported by the OAUTH JWT standard.

When an access_token is granted the auth server can use a unique public and private keypair per client_id for signing the JWT, and use the `kid` property (key id) of the JWT header to let the resource server know which public key to verify the JWT against. These public keys would be available for lookup by the resource server using the `kid`. Using this technique any JWT can be immediately and individually revoked by revoking the public /private keypair associated with that `kid`. These are called JSON Web Keys (JWK) and the list of public /private keypairs are called JSON Web Key Sets (JWKS). RFC-7517 covers this spec.

Since it's 2019 now Auth0 seems to have support for RFC-7517 but I only took a cursory look and I am not sure if they allow generating unique public /private keypairs and its kid per client.

^ | ˅ • Reply • Share ›

adobot ➦ Marcel • 4 years ago

Yes, changing the signing key should only be done as a last resort and is the "nuclear option"

^ | ˅ • Reply • Share ›

Kerem Baydoğan ➦ Marcel • 4 years ago

DB lookups are not stateful. If you store something on your server/memory instance that you have to distribute via something like hazelcast then it is stateful.

^ | ˅ • Reply • Share ›

Aun Ali Motani • 4 years ago

Hi, Thanks for great post. I understand what steps are usually needed to re-authenticate with resource server, but its not clear to me what steps are performed to renew the refresh token. if it goes through the same steps isn't it still a re-authentication?

^ | ˅ • Reply • Share ›

Load more comments

Secure access for everyone. But not just anyone.

TRY AUTH0 FOR FREE

TALK TO SALES

BLOG

Developers
Identity & Security
Business
Culture
Engineering
Announcements

COMPANY

About Us
Customers
Security
Careers
Partners
Press

PRODUCT

Single Sign-On
Password Detection
Guardian
M2M
Universal Login
Passwordless

MORE

Auth0.com
Ambassador Program
Guest Author Program
Auth0 Community
Resources