April 10, 2018

What is the OAuth 2.0 Authorization Code Grant Type?

Aaron Parecki

The Authorization Code Grant Type is probably the most common of the OAuth 2.0 grant types that you'll encounter. It is used by both web apps and native apps to get an access token after a user authorizes an app.

This post is the first part of a series where we explore frequently used OAuth 2.0 grant types. If you want to back up a bit and learn more about OAuth 2.0 before we dive in, check out What the Heck is OAuth?, also on the Okta developer blog.

What is an OAuth 2.0 Grant Type?

In OAuth 2.0, the term "grant type" refers to the way an application gets an access token. OAuth 2.0 defines several grant types, including the authorization code flow. OAuth 2.0 extensions can also define new grant types.

Each grant type is optimized for a particular use case, whether that's a web app, a native app, a device without the ability to launch a web browser, or server-to-server applications.

The Authorization Code Flow

The Authorization Code grant type is used by web and mobile apps. It differs from most of the other grant types by first requiring the app launch a browser to begin the flow. At a high level, the flow has the following steps:

- The application opens a browser to send the user to the OAuth server
- The user sees the authorization prompt and approves the app's request
- The user is redirected back to the application with an authorization code in the query string
- The application exchanges the authorization code for an access token

Get the User's Permission

OAuth is all about enabling users to grant limited access to applications. The application first needs to decide which permissions it is requesting, then send the user to a browser to get their permission. To begin the authorization flow, the application constructs a URL like the following and opens a browser to that URL.

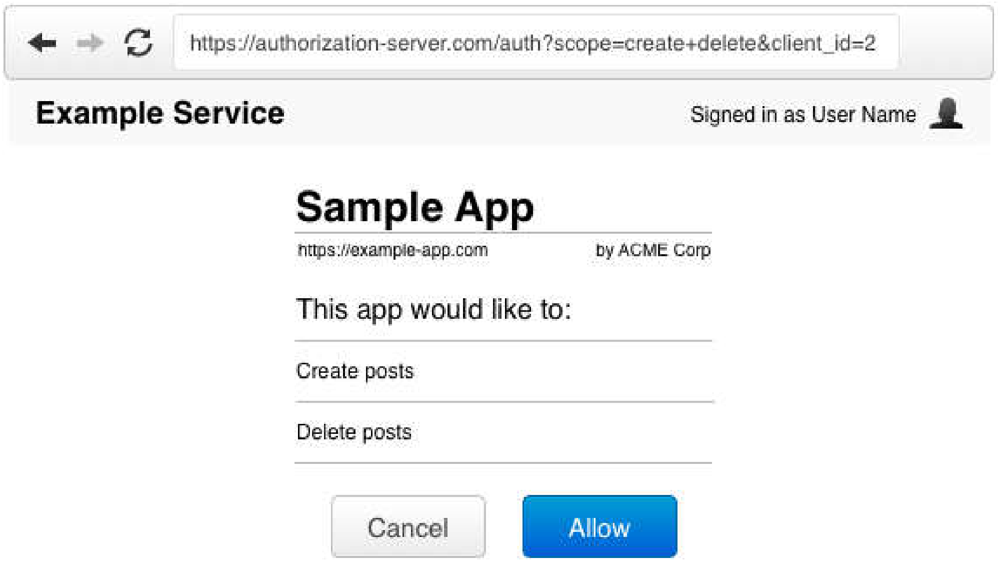
```
https://authorization-server.com/auth
 ?response_type=code
 &client_id=29352915982374239857
 &redirect_uri=https%3A%2F%2Fexample-app.com%2Fcallback
 &scope=create+delete
 &state=xcoiv98y2kd22vusuye3kch
```

Here's each query parameter explained:

- `response_type=code` - This tells the authorization server that the application is initiating the authorization code flow.
- `client_id` - The public identifier for the application, obtained when the developer first registered the application.
- `redirect_uri` - Tells the authorization server where to send the user back to after they approve the request.
- `scope` - One or more space-separated strings indicating which permissions the application is requesting. The specific OAuth API you're using will define the scopes that it supports.

- `state` - The application generates a random string and includes it in the request. It should then check that the same value is returned after the user authorizes the app. This is used to prevent CSRF attacks.

When the user visits this URL, the authorization server will present them with a prompt asking if they would like to authorize this application's request.


https://authorization-server.com/auth?scope=create+delete&client_id=2

Example Service

Signed in as User Name

Sample App

https://example-app.com

by ACME Corp

This app would like to:

Create posts

Delete posts

Redirect Back to the Application

If the user approves the request, the authorization server will redirect the browser back to the `redirect_uri` specified by the application, adding a `code` and `state` to the query string.

For example, the user will be redirected back to a URL such as

```
https://example-app.com/redirect
  ?code=g0ZGZmNjVmOWIjNTk2NTk4ZTYyZGI3
  &state=xcoiv98y2kd22vusuye3kch
```

The `state` value will be the same value that the application initially set in the request. The application is expected to check that the state in the redirect matches the state it originally set. This protects against CSRF and other related attacks.

The `code` is the authorization code generated by the authorization server. This code is relatively short-lived, typically lasting between 1 to 10 minutes depending on the OAuth service.

Exchange the Authorization Code for an Access Token

We're about ready to wrap up the flow. Now that the application has the authorization code, it can use that to get an access token.

The application makes a POST request to the service's token endpoint with the following parameters:

- `grant_type=authorization_code` - This tells the token endpoint that the application is using the Authorization Code grant type.
- `code` - The application includes the authorization code it was given in the redirect.
- `redirect_uri` - The same redirect URI that was used when requesting the code. Some APIs don't require this parameter, so you'll need to double check the documentation of the particular API you're accessing.
- `client_id` - The application's client ID.
- `client_secret` - The application's client secret. This ensures that the request to get the access token is made only from the application, and not from a potential attacker that may have intercepted the authorization code.

The token endpoint will verify all the parameters in the request, ensuring the code hasn't expired and that the client ID and secret match. If everything checks out, it will generate an access token and return it in the response!

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"MTQ0NjJkZmQ5OTM2NDE1ZTZjNGZmZjI3",
  "token_type":"bearer",
  "expires_in":3600,
  "refresh_token":"IwOGYzYTlmM2YxOTQ5MGE3YmNmMDFkNTVk",
  "scope":"create delete"
}
```

The Authorization Code flow is complete! The application now has an access token it can use when making API requests.

When to use the Authorization Code Flow

The Authorization Code flow is best used in web and mobile apps. Since the Authorization Code grant has the extra step of exchanging the authorization code for the access token, it provides an additional layer of security not present in the Implicit grant type.

If you're using the Authorization Code flow in a mobile app, or any other type of application that can't store a client secret, then you should also use the PKCE extension, which provides protections against other attacks where the authorization code may be intercepted.

The code exchange step ensures that an attacker isn't able to intercept the access token, since the access token is always sent via a secure backchannel between the application and the OAuth server.

Learn More About OAuth and Okta

You can learn more about OAuth 2.0 on OAuth.com, or check out any of these resources to get started building!

- Get Started with Spring Boot, OAuth 2.0, and Okta
- Token Authentication in ASP.NET Core 2.0 - A Complete Guide
- Secure your SPA with Spring Boot and OAuth

Or hit up Okta's OIDC/OAuth 2.0 API for specific information on how we support OAuth. And as always, follow us on Twitter @oktadev for more great content.

PS: We recently built a new security site where we're publishing lots of other security-focused articles (like this one). You should cehck it out!

Aaron Parecki

Aaron Parecki is a Senior Security Architect at Okta. He is the author of OAuth 2.0 Simplified, and maintains oauth.net. He regularly writes and gives talks about OAuth and

TERMS & CONDITIONS

PRIVACY POLICY

online security. He is an editor of several internet specs, and is the co-founder of IndieWebCamp, a conference focusing on data ownership and online identity. Aaron has spoken at conferences around the world about OAuth, data ownership, quantified self, and home automation, and his work has been featured in Wired, Fast Company and more.

Okta Developer Blog Comment Policy

We welcome relevant and respectful comments. Off-topic comments may be removed.

23 Comments **Okta Developer Blog** **Disqus' Privacy Policy** 1 **Login**

Recommend 16 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Luís de Sousa • a year ago

Nice and simple, thank you very much

1 • Reply • Share ›

Peter Cimring • 10 days ago

Thanks, Aaron

There's one point that confuses me:

To fetch the Access Token, the OAuth spec implies (or *seems* to imply) that it's sufficient to pass a client id (*without* a client secret) - see: https://tools.ietf.org/html...

However, in practice, Authorization Servers seem to require *both* the client id and the client secret - as you mentioned in the above article.

Have I missed something? (Or is this what the spec means by the "client type" being "confidential"?)

Thanks!

• Reply • Share ›

Aaron Parecki Peter Cimring • 9 days ago

It's true that the spec allows the authorization code flow without a client secret for clients that were not issued a client secret. This is vulnerable to a few different attacks, so in practice most servers never allowed that from the beginning, instead making the Implicit flow available to public clients (which has its own problems too).

Some time after RFC6749 was published, PKCE was developed as way to enable the authorization code flow securely without using a client secret. But of course since RFCs don't get updated, if you're only reading RFC6749 you may not necessarily find your way to PKCE.

This is one of the motivations for OAuth 2.1, which is to consolidate all the current best practices into a new spec so that this kind of thing doesn't confuse people.

^ | ˅ • Reply • Share ›

Peter Cimring ➦ Aaron Parecki • 9 days ago

Thanks, Aaron!

One more question, if I might - since you used the word 'public' :)

I'm trying to characterize the following 4 flows.

Would the following accurately summarize their use cases?

1. Client Credentials - 'private', server-to-server
2. Password Credentials - 'private', app-to-server (generally not recommended, because the resource owner passes his credentials)
3. Implicit - 'public', app-to-server (used when 'Authorization Code' is not an option)
4. Authorization Code - 'private', app-to-server

Sorry - but of an involved question... :)

^ | ˅ • Reply • Share ›

Aaron Parecki ➦ Peter Cimring • 9 days ago

This would be a good summary of RFC6749, but doesn't reflect the current best practice. Implicit serves no purpose anymore and is being removed, Password is also being removed because it's dangerous and inflexible, and the authorization code flow is being extended to include PKCE by default. Web apps, mobile apps, and single-page apps should all use authorization code + PKCE now. Take a look at https://oauth.net/2.1/ and https://oauth.net/2/oauth-best-practice/ for some more links and background.

^ | ˅ • Reply • Share ›

Peter Cimring ➦ Aaron Parecki • 9 days ago

Great - thanks!

^ | ˅ • Reply • Share ›

Gautam Nagpal • 16 days ago

Is there any option to sent this grant_type as request Body using VBscript or any other script or language ?

^ | ˅ • Reply • Share ›

toerktumlare • 2 months ago • edited

"The application opens a browser to send the user to the OAuth server"

this is a quite a fuzzy explanation. The application itself will not do anything without the initiation of something by the user, and what the user actually does is not described, is it access the api and getting rejected, or is it them clicking the "login" button. Also, what if the application is already in a browser (say website application) does it open another browser, another tab, another page?

Quite fuzzy

• Reply • Share ›

Matt Raible Mod toerktumlare • 2 months ago

It's them clicking on a login button. If the application is already in a browser, it'll usually do a redirect in the same tab, but I've also seen some applications use a popup window.

• Reply • Share ›

Ivan Starkov • 3 months ago

The thing I dont understand well.
What if attacker intercept authorisation code from callback url (on os level for example using extension or screen reader), then force its own auth process to get valid state parameter for its own session, then just call callback url in its own session using authorisation code from step 1 and valid state from step 2, finally attacker is able to finish auth process as state is not used in exchange of authorisation code and access token.

• Reply • Share ›

aaronpk Mod Ivan Starkov • 3 months ago

Yes you're right, that exact attack is why PKCE exists! PKCE prevents that by requiring the use of an additional secret that the app generates on each request.

• Reply • Share ›

Ivan Starkov aaronpk • 3 months ago

Thank you!!! I thought that PKCE is just replacement for implicit flow.

• Reply • Share ›

aaronpk Mod Ivan Starkov • 3 months ago

I should clarify. This problem doesn't exist when the client can use a client secret. It's only a problem for public clients that don't have a secret. In that case, PKCE solves the problem, and the Implicit flow has that problem and can't be solved.

• Reply • Share ›

Ivan Starkov aaronpk • 3 months ago • edited

For me it looks like the problem exists for clients with secrets.

This url "*example - app . com/redirect?*

code=g0ZGZmNjVmOWIjNTk2NTk4ZTYyZGI3&state=xcoiv98y2k
can be intercepted at user OS level before this request reaches client with secret code (finally its user browser do redirect its not server 2 server communication).

Attacker take code=g0ZGZmNjVmOWIjNTk2NTk4ZTYyZGI3, initiate auth in other browser, get state for his session, and use code above with his own state.

I don't see this attack can be prevented, as state is not a parameter of exchange code to token.

^ | ˅ • Reply • Share ›

Aaron Parecki ➦ Ivan Starkov • 3 months ago

When there's a client secret involved, the authorization server requires the client secret in order to use the authorization code. So even if someone did steal the authorization code, they wouldn't be able to use it, because they can't have also stolen the client secret.

^ | ˅ • Reply • Share ›

Ivan Starkov ➦ Aaron Parecki • 3 months ago

They will.
Imagine client is a web server with callback url like.

`mysite.com/google/callback`

So secret is known only for web server and then browser get from google "signIn" redirect to `mysite.com/google/callback?code=GOOD_CLIENT_CODE&state=GOOD_CLIENT_STATE` web server can exchange code for access token.

But before browser get redirected and send request to api, nothing prevent "malicious software" to intercept "code" from browser as its shown in browser url.

Then "malicious software" can from other browser just force new auth request, get MALICIOUS_CLIENT_STATE, and then just browse to
`mysite.com/google/callback?code=GOOD_CLIENT_CODE&state=MALICIOUS_CLIENT_STATE`

web server will compare MALICIOUS_CLIENT_STATE with session.state - all will be ok, then
exchange GOOD_CLIENT_CODE using SECRET and then authorize wrong user.

^ | ˅ • Reply • Share ›

Aaron Parecki ➦ Ivan Starkov • 3 months ago

ah I misunderstood what you were describing at first. You're describing the authorization code injection attack https://tools.ietf.org/html... and PKCE solves that!

• Reply • Share ›

Ivan Starkov Aaron Parecki • 3 months ago

Super! Thank you!

• Reply • Share ›

PingPong • 9 months ago

What will happen if the user has not signed on?

For internal company, is it possible disable the prompt that user approves app request as mentioned below:

[illegible]