@justin _ _ richer

https://bspk.io/

Introduction to OAuth 2.0

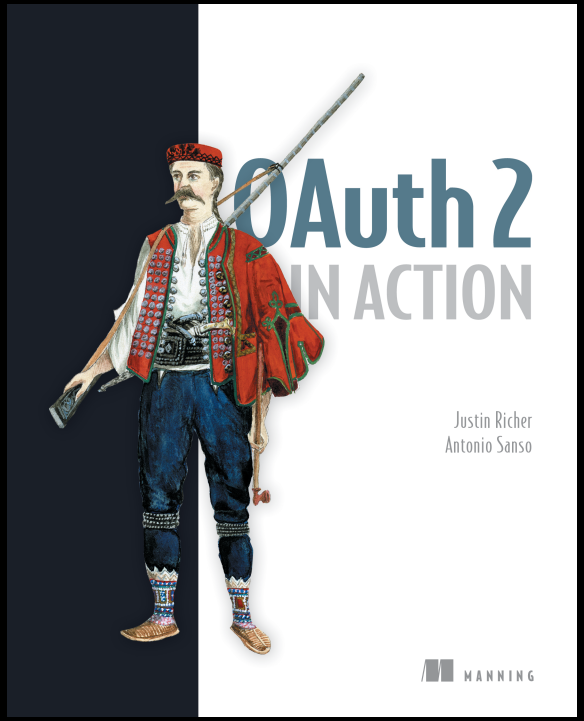
Justin Richer

Bespoke Engineering

Course Version 1.6

Feedback is welcome!

Try the home edition



- *OAuth 2 In Action*
- Code is open source
- Published March 2017

WHAT IS OAUTH 2.0?

From the spec (RFC6749)

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

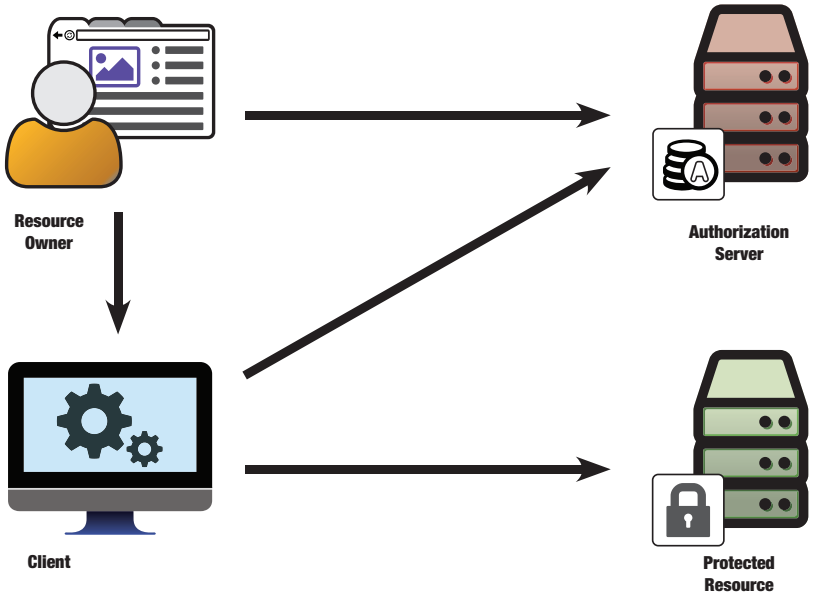
The good bits

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

In other words

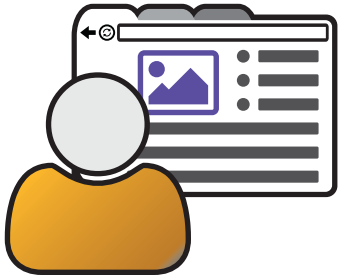
OAuth 2.0 is a delegation protocol that lets people allow applications to access things on their behalf.

Who is involved?



The resource owner

- Has access to some resource or API
- Can delegate access to that resource or API
- Usually has access to a web browser
- Usually is a person



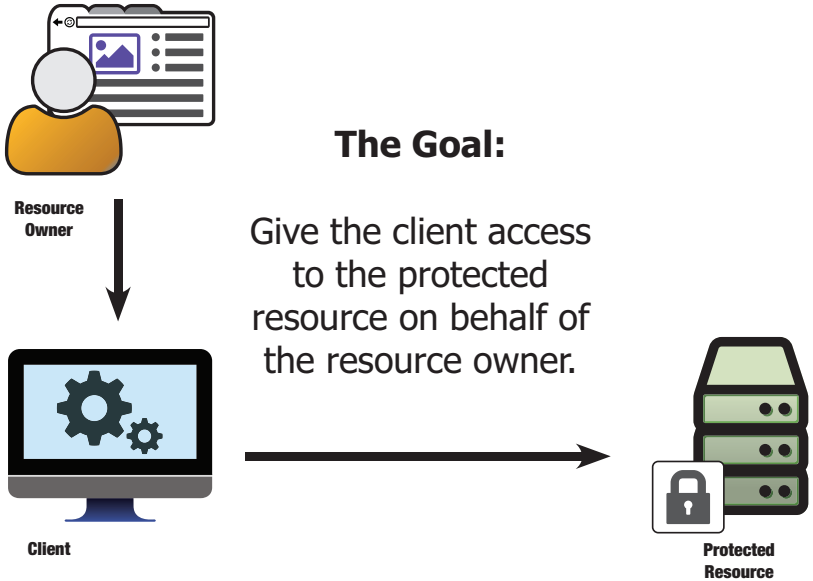
The protected resource

- Web service (API) with security controls
- Protects things for the resource owner
- Shares things on the resource owner's request

The client application

- Wants to access the protected resource
- Does things on the resource owner's behalf
- Could be a web server
  - But it's still a "client" in OAuth parlance
  - Could also be a native app or JS app

What are we trying to solve?



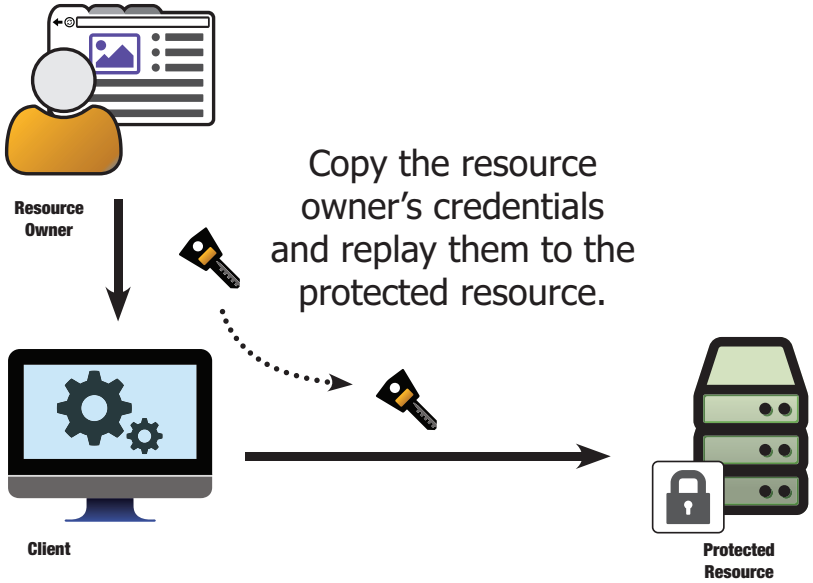
The Goal:

Give the client access to the protected resource on behalf of the resource owner.

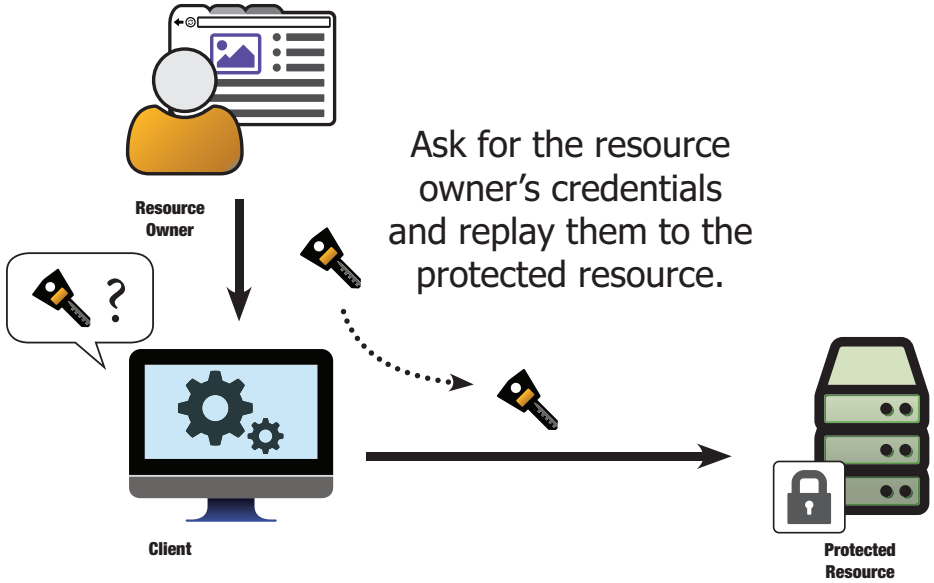
This isn't a new problem

People have been solving this for a long time

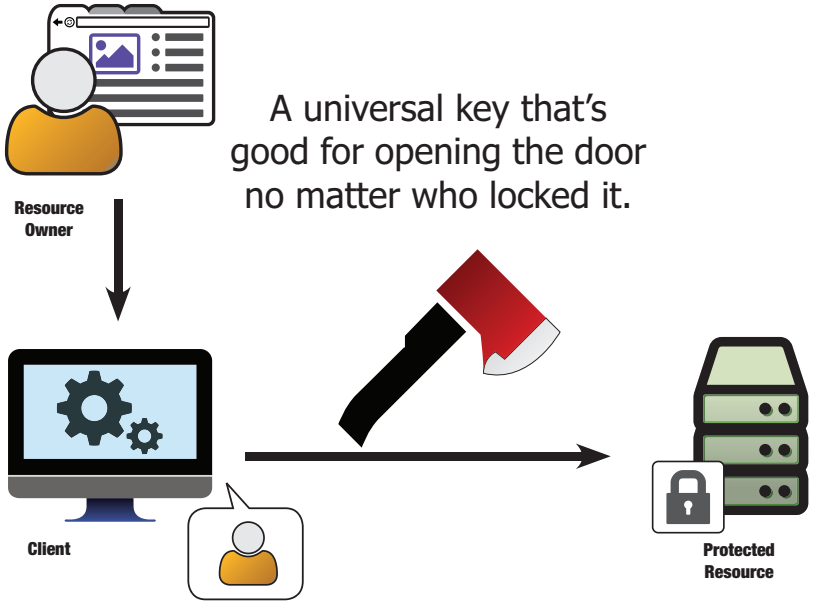
Steal the keys



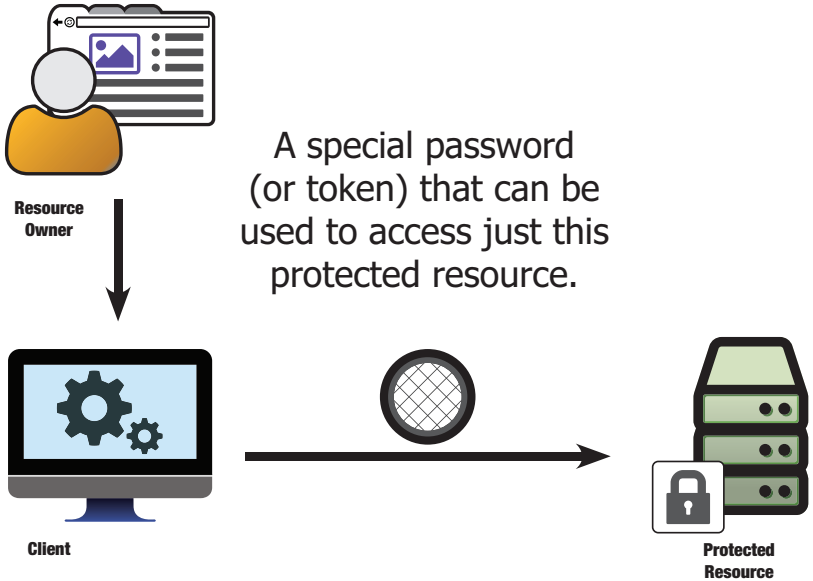
Ask for the keys



Use a universal key



Service-specific credentials



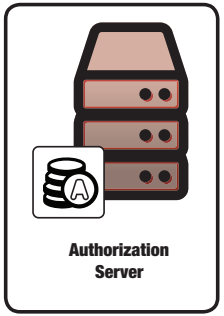
WE'RE GETTING CLOSER...

Introducing the Authorization Server (AS)

Resource Owner

Client

The Authorization Server gives us a mechanism to bridge the gap between the client and the protected resource

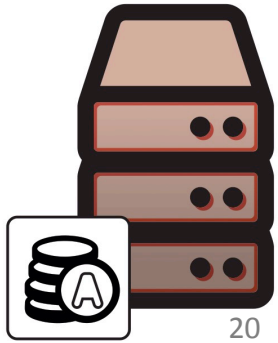


Authorization Server

Protected Resource

The Authorization Server

- Generates tokens for the client
- Authenticates resource owners (users)
- Authenticates clients
- Manages authorizations



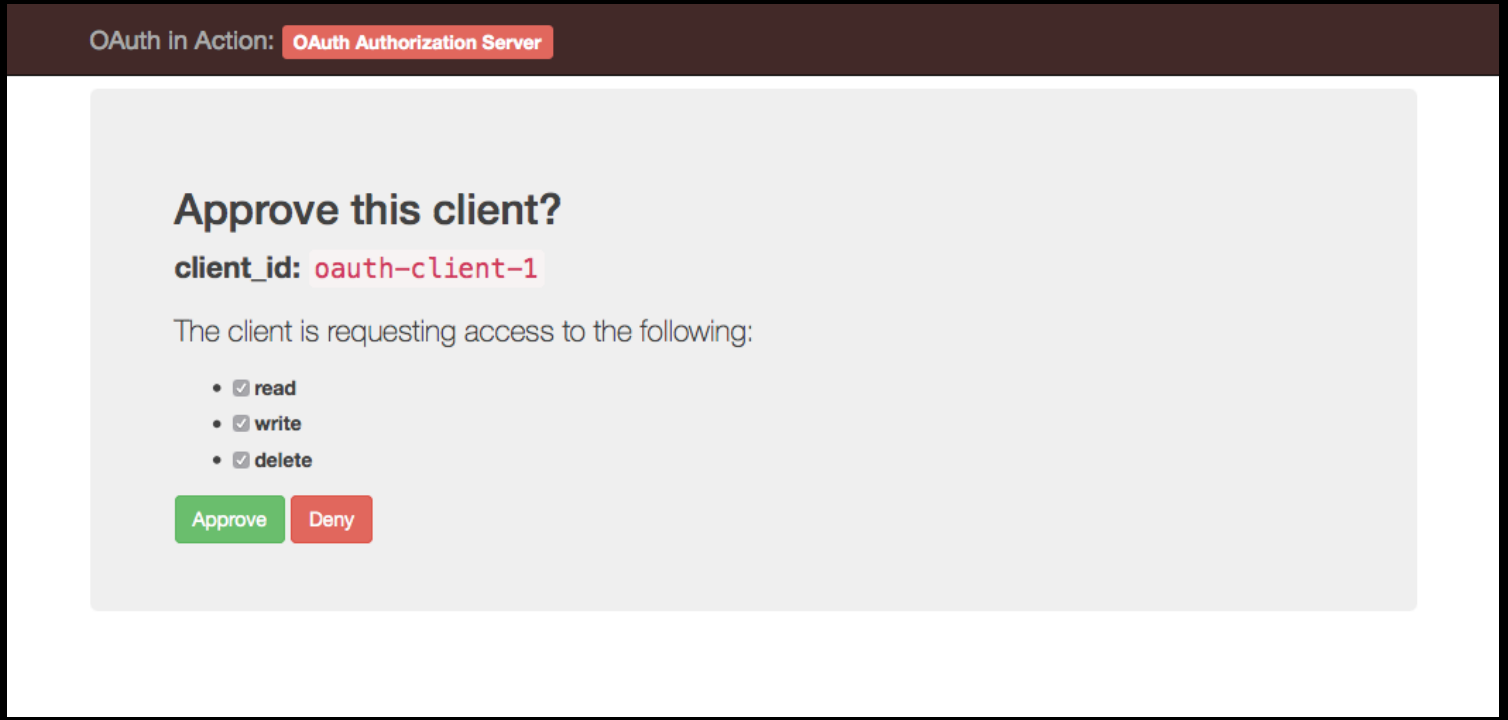
OAuth Tokens

- Represent granted delegated authorities
  - From the resource owner to the client for the protected resource
- Issued by authorization server
- Used by client
  - Format is opaque to clients
- Consumed by protected resource

Example OAuth Tokens

- 92d42038006dba95d0c501951ac5b5eb
- 2df029c6-b38d-4083-b8d9-db67c774d13f
- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
- waterbuffalo-elephant-helicopter-argument

You've used OAuth



A BRIEF HISTORY OF OAUTH 2.0

Circa 2006

- HTTP password authentication common for API access
  - “Give me your password”
- Internet companies have proprietary solutions for delegated access
  - BBAuth, AuthSub, a few others

The problem

- Two smaller sites want to connect their APIs for their users
- Both use OpenID for login
  - No username/password to pass!
- Neither wants to use a proprietary solution

A new standard is born

- OAuth 1.0 is published independently
  - No formal standards body, people just use it
- A session fixation attack is found and fixed
  - New version is called OAuth 1.0a
- This community document is standardized as RFC5849 in the IETF

People start using it

- OAuth 1.0a solves major pain points for many people in a standard and understandable way
- Google, Yahoo, and others replace their solutions with the new standard

People start abusing it

- People also decide to start using OAuth for off-label use cases
  - Native applications
  - No user in the loop
  - Distributed authorization systems

Version 2.0: The framework

- Modularized concepts
- Separated previously conflated components
- Added explicit extensibility points
- Removed pain points of implementers
- Standardized in RFC6749 and RFC6750

What does this mean?

- Instead of a single protocol, OAuth 2.0 defines common concepts and components and different ways to mix them together
- It's not a single standard, it's a set of standards for different use cases

WHAT OAUTH ISN'T

Not defined outside of HTTP

- Core protocol defined only for HTTP
- Relies on TLS for securing messages
- There are efforts to use OAuth over non-HTTP protocols
  - GSSAPI
  - CoAP

Not an authentication protocol

- Relies on authentication in several places
  - Client authentication to token endpoint
  - Resource owner authentication at authorization endpoint
- Doesn’t communicate anything about the user
- However: authentication protocols can be built using OAuth (OpenID Connect)

No user-to-user delegation

- Allows a user to delegate to a piece of software but not to another user
- However, multi-party delegation can be built using OAuth as a core component (UMA)

No authorization processing

- Tokens can represent scopes and other authorization information
- Processing of this information is up to the resource server
- However, several methods (UMA, JWT, introspection) to communicate this information

No token format

- Token is opaque to the client
- Token needs to be issued by the authorization server and understood by the resource server, but they're free to use whatever format they want
- However, JSON Web Tokens (JWT) provide a useful common format

No cryptographic methods

- Core OAuth relies on TLS for protecting information in transit
- However, other mechanisms like JSON Object Signing and Encryption (JOSE) define things that can be used with OAuth

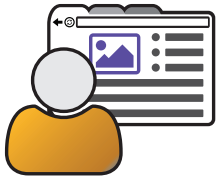
Not a single protocol

- OAuth 2.0 is a *framework*
  - Several core flows plus extensions
- Two things can “implement OAuth” but be incompatible with each other
- However, code re-use and patterns between common components makes life simpler

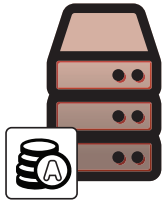
The Authorization Code Flow

A deep dive into the canonical OAuth 2.0 transaction

The pieces of OAuth



Resource Owner



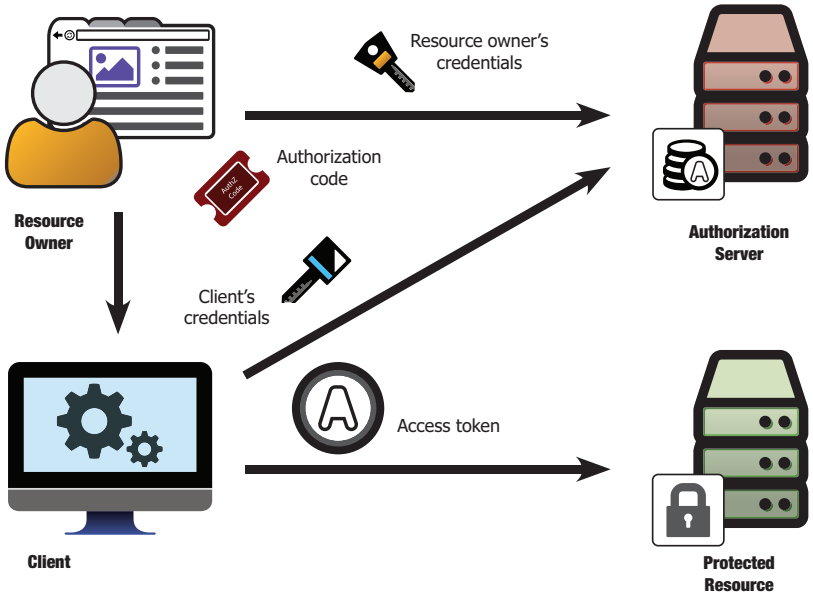
Authorization Server

Access Token

Client

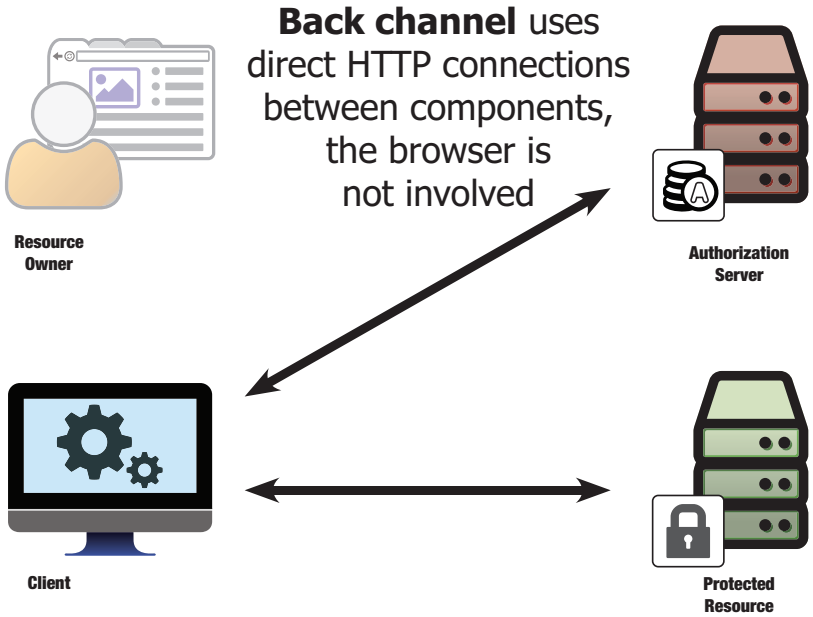
Protected Resource

The authorization code flow

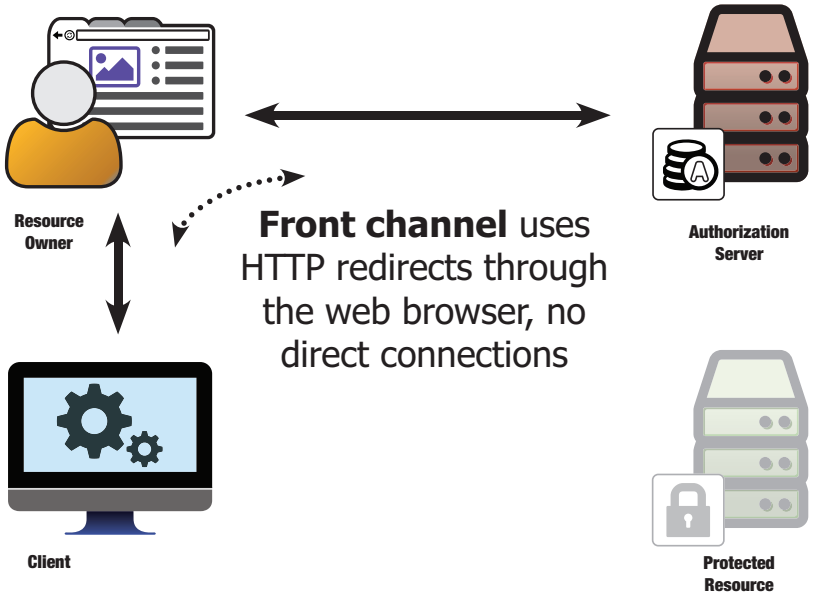


TWO FORMS OF COMMUNICATION

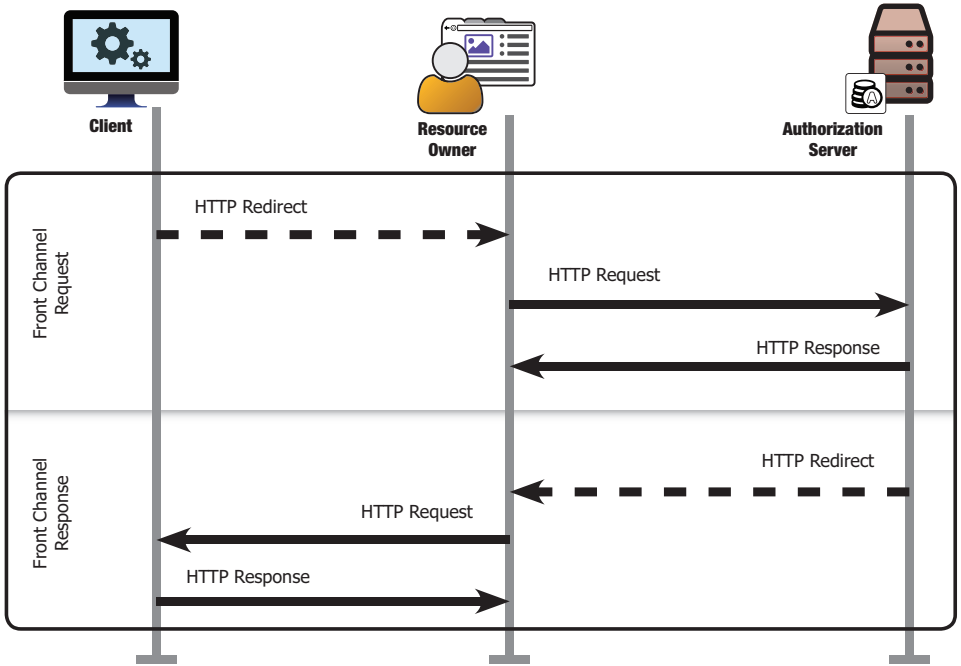
The back channel



The front channel



A front channel request/response



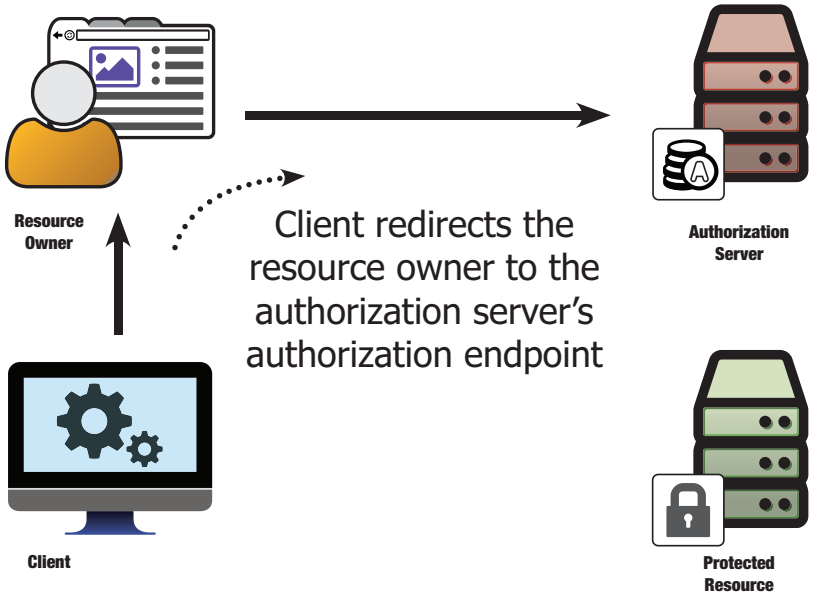
Why both?

- Separation of information
- Front channel when the user's involved
- Back channel when they're not

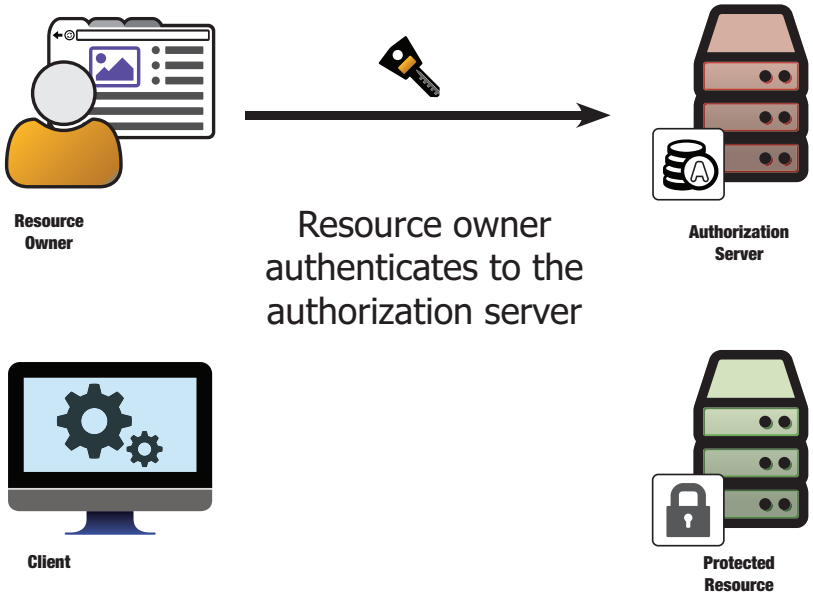
The authorization code flow

Step by step

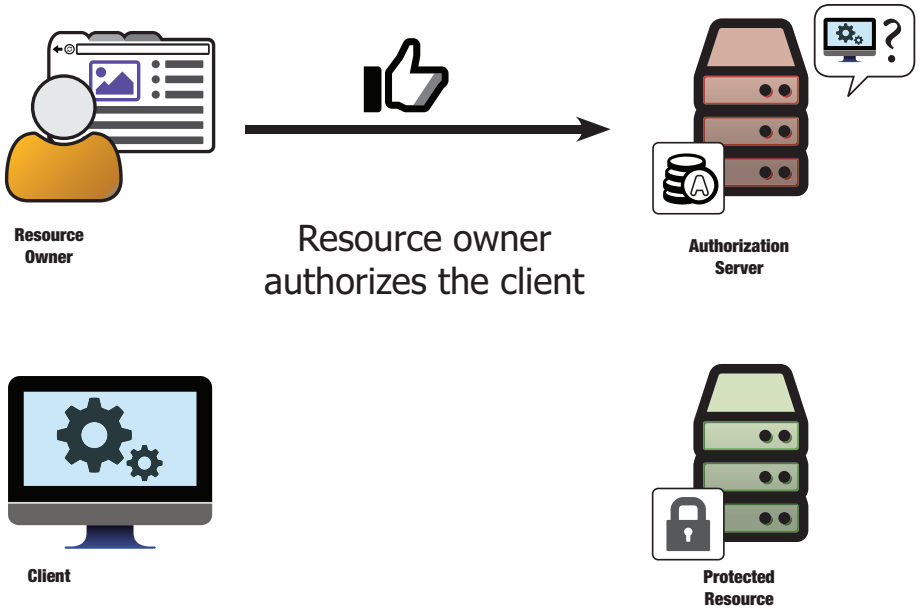
Authorization Code: Step 1



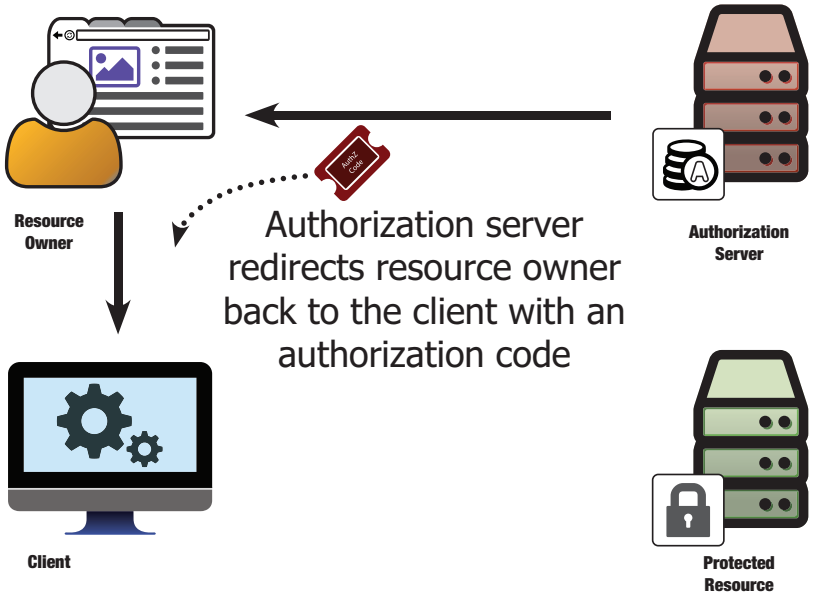
Authorization Code: Step 2



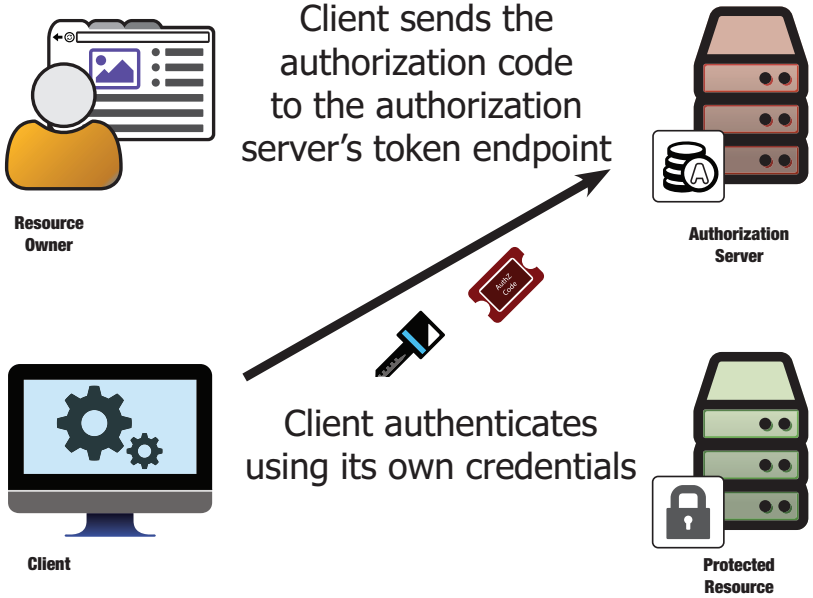
Authorization Code: Step 3



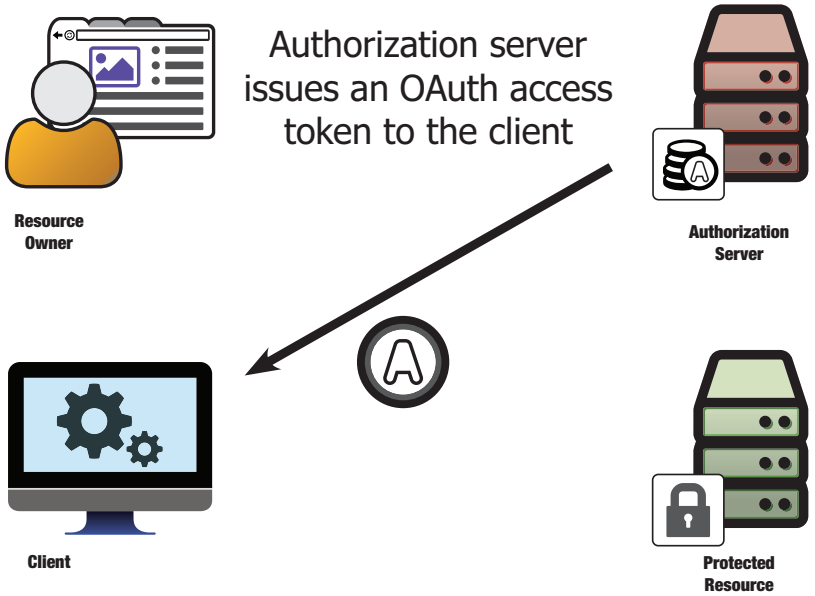
Authorization Code: Step 4



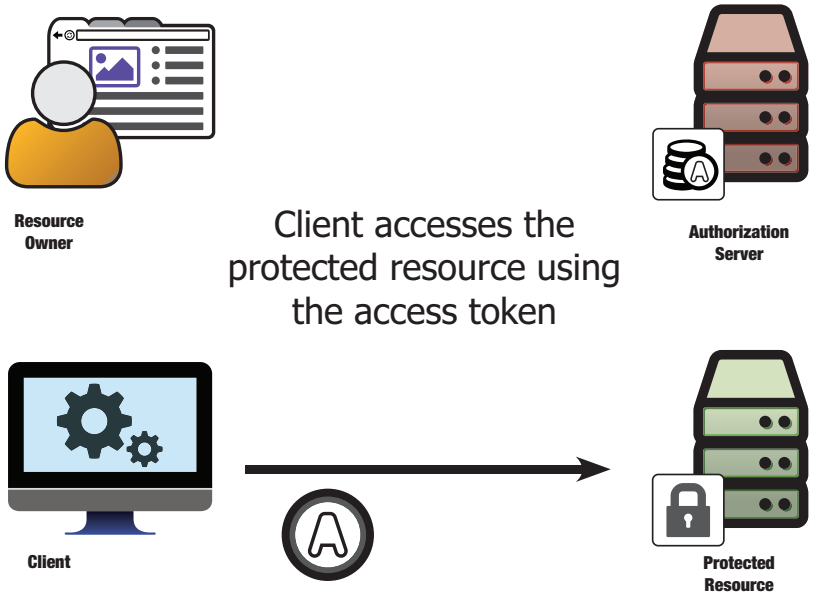
Authorization Code: Step 5



Authorization Code: Step 6



Authorization Code: Step 7



Refresh Tokens

When the user isn't there

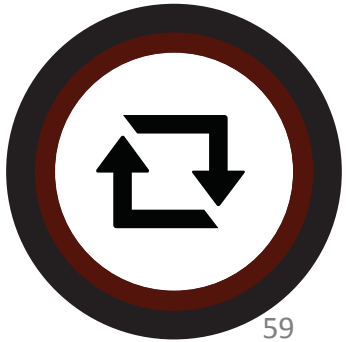
- Access tokens work after the user leaves
  - One of the original design goals of OAuth
- What does a client do when the access token stops working?
  - Expiration
  - Revocation

Getting a new token

- Repeat the process of getting a token
  - Interactive grants: send the resource owner to the authorization endpoint
- But what if the user's not there anymore?

Refresh tokens

- Issued alongside the access token
- Used for getting new access tokens
  - Presented along with client credentials
  - Not good for calling protected resources directly



SCOPES

API Design

- Naïve APIs (like what we built) allow simple yes/no access
  - If your token is good, your request is good
- Smarter APIs divide access

Limited access

- Type of action
  - Read, write, delete
- Type of resource
  - Photos, metadata, profile
- Time of access
  - User is offline, limited number of accesses

OAuth Scopes

- Strings that represent what the token can do
- Client can ask for scopes
- Resource owner approves scopes
- Access token is bound to scopes

Other ways to do OAuth 2.0

Protocol flexibility

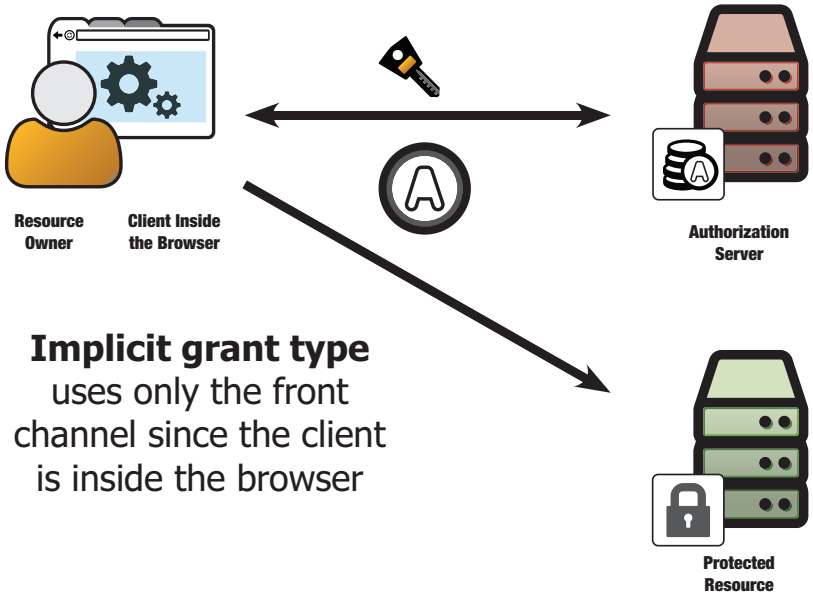
- Canonical use case: web server based application accessed through a browser
- Authorization code flow is built around this use case
- What about different kinds of clients?
- What about different kinds of delegation?

Implicit Flow

Stuff the client into the browser

- Authorization code flow keeps the token out of the browser and in the client
- But what if the client is *inside* the browser?

The implicit flow



Implicit grant type uses only the front channel since the client is inside the browser

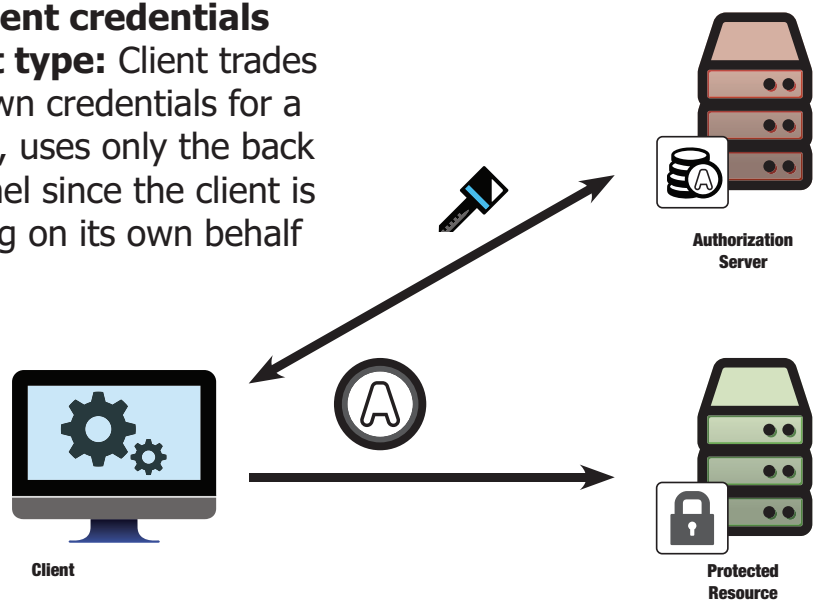
Client Credentials Flow

Client acts on its own behalf

- No explicit resource owner
- Replacement for API keys

The client credentials flow

Client credentials grant type: Client trades its own credentials for a token, uses only the back channel since the client is acting on its own behalf

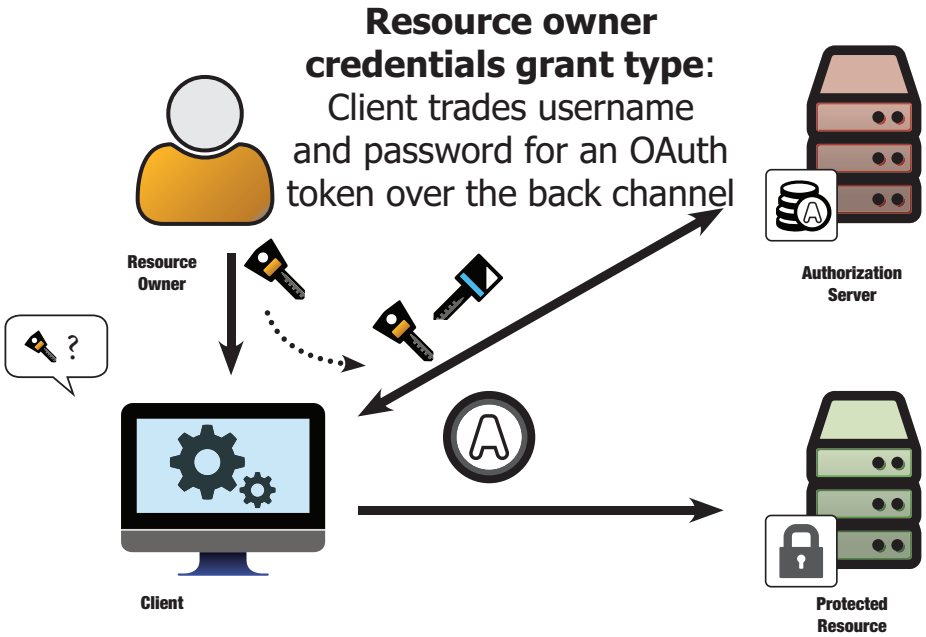


Resource Owner Password Flow

Stealing the password

- Codify the anti-pattern: ask the user for their credentials and replay them
- Instead of saving the credentials, trade for an access token

The resource owner password flow



Hold on!

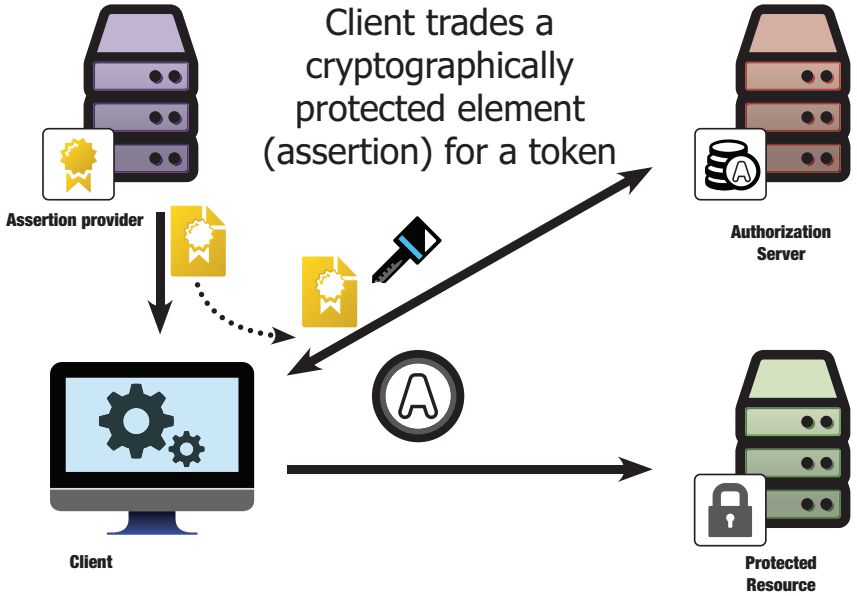
Didn’t we say it was bad to steal the credentials?

Assertion Flows

Third-party authorization

- Have a trusted third party hand authorization to the client
- Client trades that for a token

The assertions flows

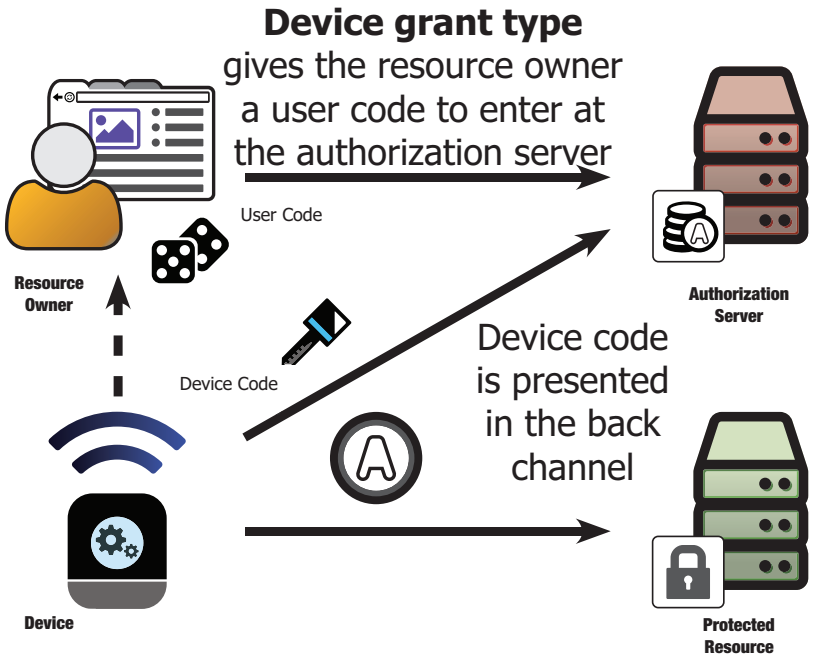


Device Flow

Limited interactivity

- Not every client has a web browser
  - Set-top boxes
  - Smart devices
- How do we get user interaction?
  - Split the pieces
  - Use the user to carry the information

The device flow



Native Clients

What's a native client?

- Runs on the end user's system
  - Not hosted on a remote web server
  - Not executed inside of a web browser
- Can be desktop or mobile
  - Local self-contained web server apps qualify

What makes a native client different?

- Functionality lives outside the browser
- Can’t keep secrets from the user
  - Especially configure-time secrets
- Requires adaptations to redirect URI to use the front channel

Dealing with secrets

- Application is copied and run many times
  - Shouldn't give each copy the same secret
- Dynamic client registration
  - Give each instance its own ID and secret
- Public clients
  - Share an ID and don't use secrets

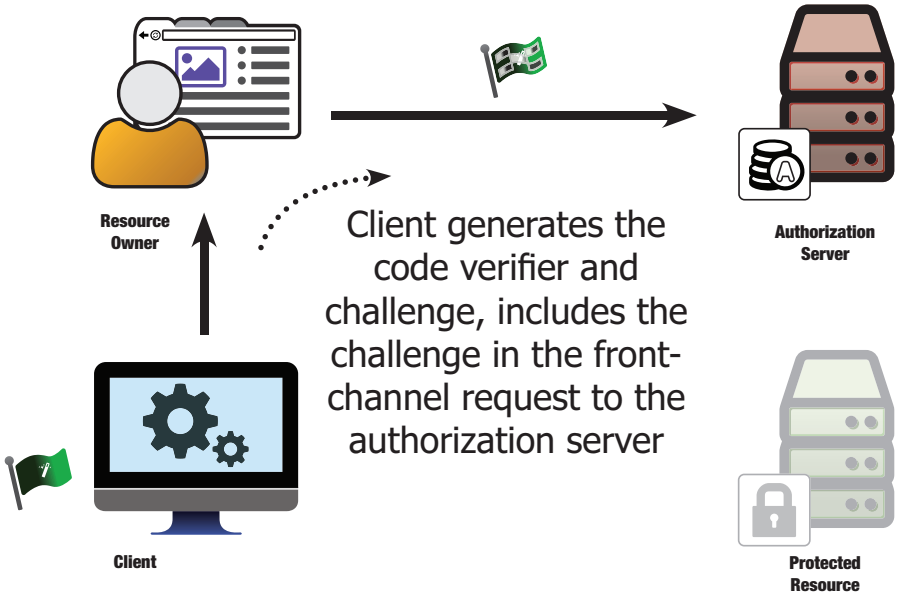
Redirect URIs

- Custom URI scheme
  - myapp:/oauth_callback?code=ABC123
- Locally hosted web server
  - http://localhost:39103/myapp?code=ABC123
- Remote host with push notification
  - https://push.example.com/app-942/code=ABC123

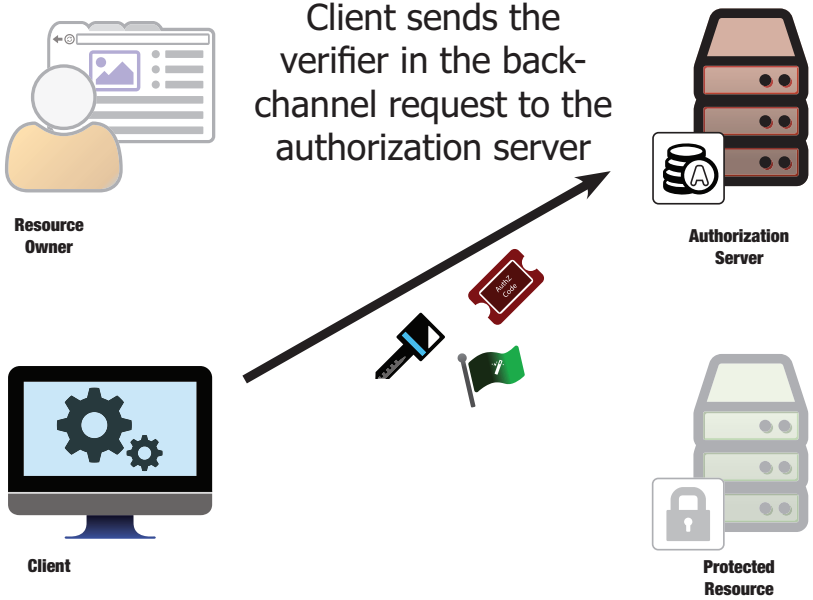
Redirect URIs with custom schemes

- Apps need to register for namespace
- Any app can take any namespace
- Malicious apps can try to grab items coming in on redirect URIs
  - Authorization codes (for code flow)
  - Tokens (for implicit flow)

PKCE: Sending the challenge



PKCE: Sending the verifier

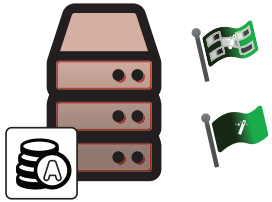


PKCE: Verifying the challenge

Resource Owner

Client

Authorization server re-generates the challenge from the verifier and compares it to the challenge previously sent



Authorization Server

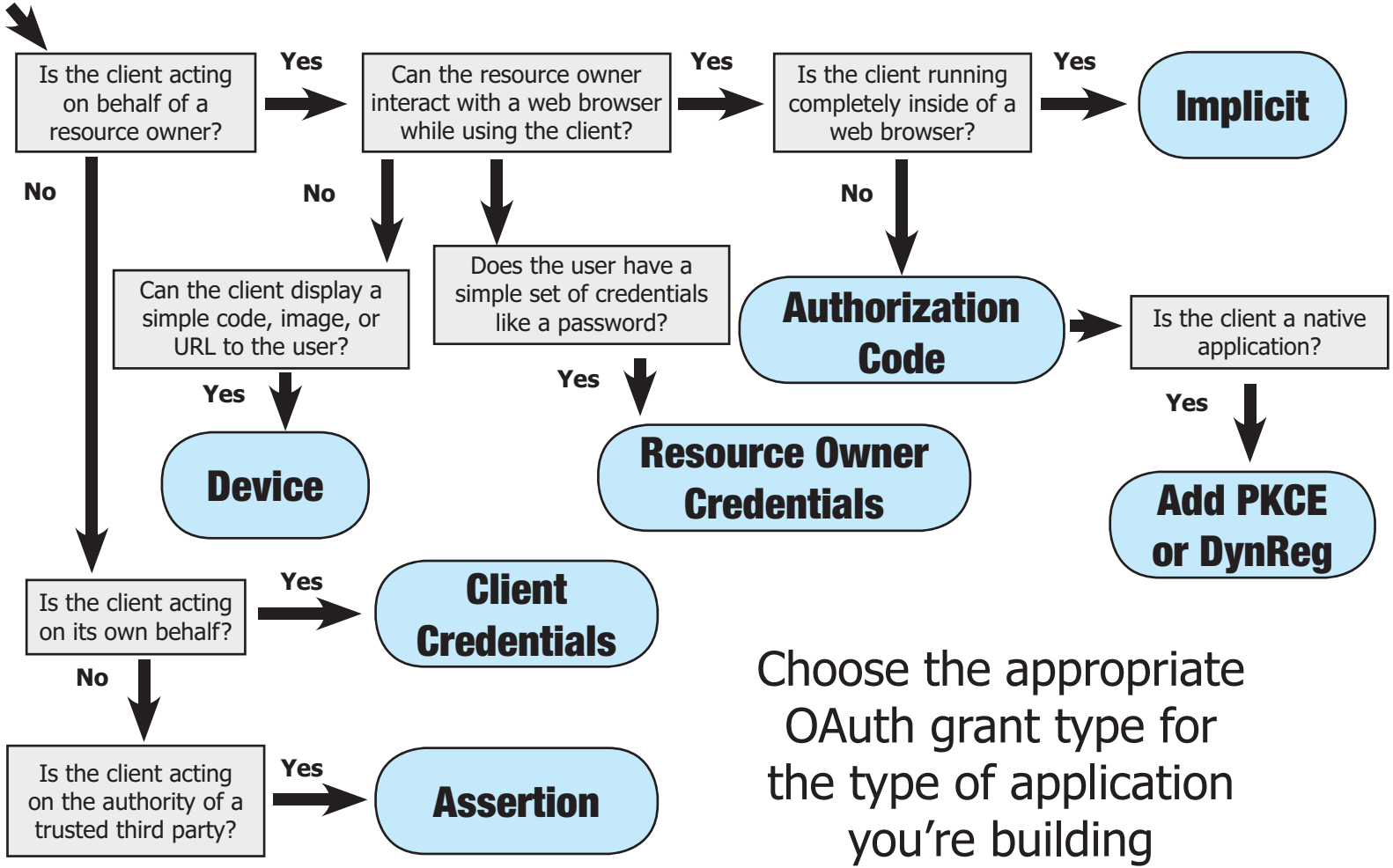
Protected Resource

Managing the Grant Types

Different use cases

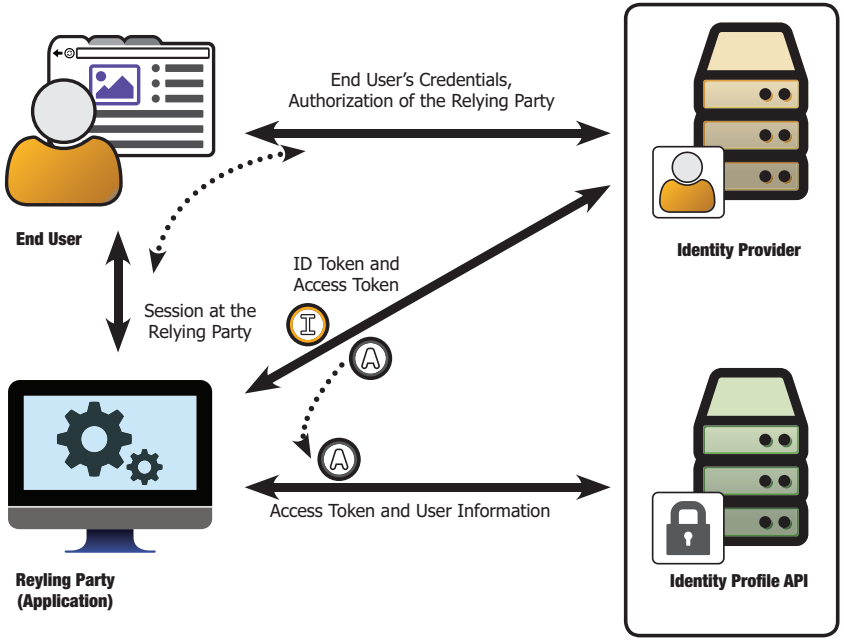
- Authorization code flow: web applications, some native applications
- Implicit flow: in-browser applications
- Client credentials flow: non-interactive
- Password flow: trusted legacy clients
- Assertion flows: trust frameworks

How to Choose a Grant Type

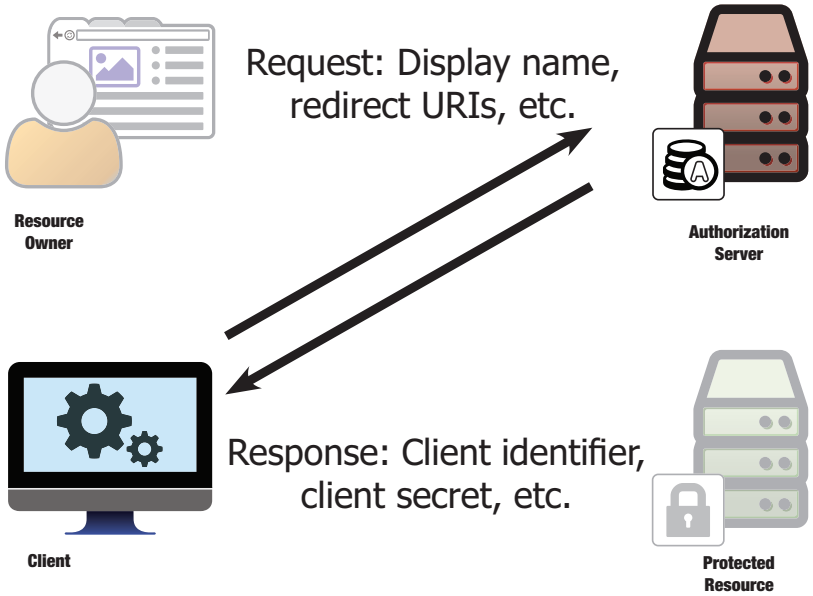


Choose the appropriate OAuth grant type for the type of application you're building

OpenID Connect



Dynamic Client Registration



Software statements

- Third party generates an assertion that contains fixed attributes of the client
  - Client can't change or override what's in the statement
- Client presents the statement alongside any variable attributes
- Server generates unique ID and secret for client

Why use a software statement?

- Many instances of a client software
  - Each instance needs its own ID/secret
  - All instances should be “recognizable”
- Allow pre-registration across domains
  - Software statement from trusted server
  - Individual AS registrations for clients

Token Introspection

OAuth tokens are opaque

- But they're only opaque *to the client*
- Protected resource needs to know the token
  - What's it good for?
  - Who issued it?
  - Is it valid?

How does the resource know?

- Database lookup
  - AS and RS are in the same box
- Pack information into the token itself
  - Remember JWT?
- Query the AS
  - Runtime lookup over the network

“What’s this token good for?”

- Protected resource queries the AS about a token it received
- AS responds with a JSON structure describing the token’s status

Introspection trade-offs

- Requires extra credentials (at the RS)
- More network traffic
- Subject to cache consistency problems
  - Introspect every time? Only on timeout?

Token Revocation

Completing the token lifecycle

- OAuth defines how to get a new token and refresh a dead token
- Revocation allows clients to proactively throw away tokens they no longer use

Why revoke tokens?

- Native application being uninstalled
- User selects “log out” or “de-authorize” from the client (not the AS)

A simple protocol

- Client POSTs to the revocation endpoint
  - Token included in body
- Server deletes the token if it finds it
- Server tells the client everything is OK
  - Even if no token was deleted, we pretend we did
  - Otherwise clients could use this to fish for token values
- Client throws out its copy of the token

PoP, mTLS, and Token Binding

Beyond bearer tokens

- Bearer tokens are sent as-is over the wire
- Anyone who has access to the token can use it
- Proof of Possession (PoP) tokens require cryptographic proof of a key
  - Token is transmitted as-is
  - Key is used to sign something, not transmitted itself

Two parts

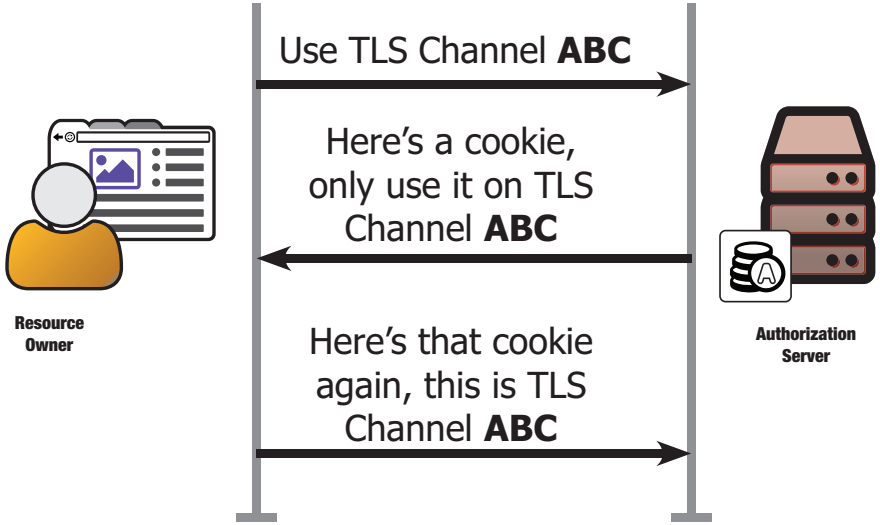
Token:
Opaque to client
Associated with scopes and RO
Sent as-is to PR

Key:
Known to client
Associated with token
Used to sign request

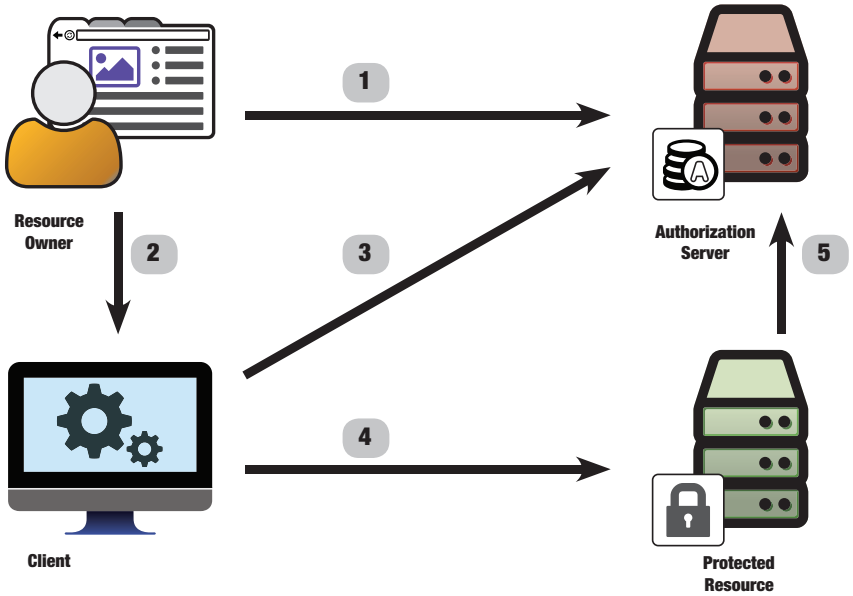
Mutual TLS

- Client presents certificate to token endpoint
- AS hashes certificate and ties it to token
- Client presents same certificate to RS
- RS hashes certificate and sees if it's the same as the one bound to the token
- Client *does not have to authenticate* with TLS

Token binding



A problem with token binding



Wrapping Up

Thank You

http://oauthinaction.com/