Going Against the Flow

How to Build and Break OAuth 2.0

Dominik Holzapfel & Claudia Ully
BSides Munich 2022

nviso

Who we are

Who we are: nviso

Our Company

NVISO is a pure play **Cyber Security consulting firm** of 150+ specialized security experts and founded in 2013.

We have offices in **Belgium** and **Germany** (Frankfurt & Munich).

Our mission is to **safeguard the foundations of European society from cyber attacks.**

Our DNA

Pride: We are proud of who we are and what we do.

We care: We care about our customers and people.

Break barriers: We challenge the status quo by continuous innovation.

No BS: We keep our promises and don't fool around.

Our Research

We invest 10% of our annual revenue in innovation to ensure we stay on top of our game; innovating the things we do, the technology we use and the way we work form an essential part of this.

Follow us on :

@NVISOsecurity and @NVISO_Labs

blog.nviso.eu/

Our Services

We have a **strong track record** providing information and cyber security services to the **Financial Services**, **Government & Defense** and **Technology** sector.

NVISO can support you throughout the **entire cyber security incident lifecycle**.

Why we are here

Understand why we need OAuth 2.0

Be aware of potential implementation flaws

Know different flows and where to use them

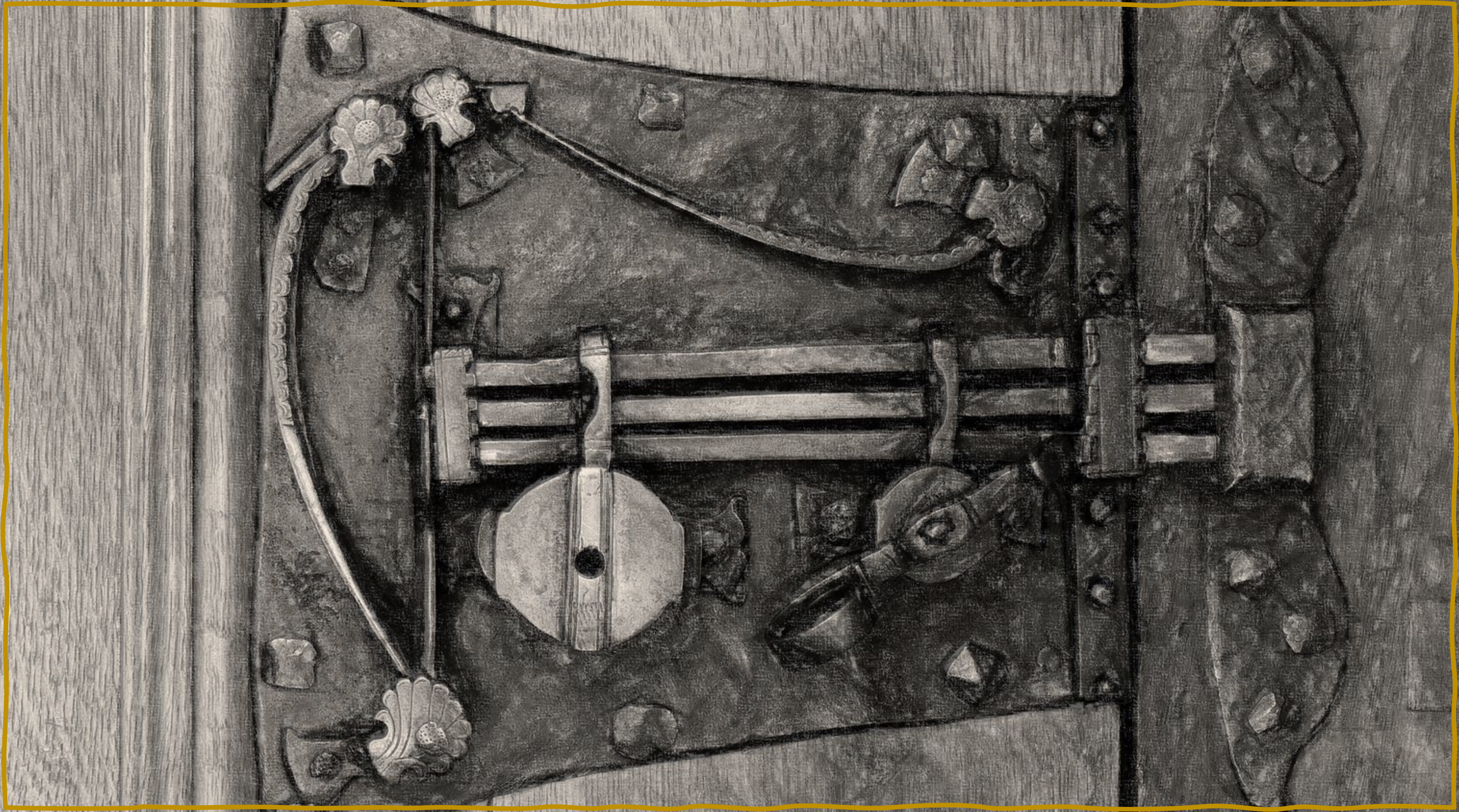
Know how to test for vulnerabilities

What we are going to do

09:00		10:25		11:50		13:00
		10 min BREAK		10 min BREAK		
	Learn the basics Implement an Implicit Flow for a Single-Page App		Implement an Authorization Code Flow for a Web App		Learn about PKCE Learn about mobile clients Wrap-up	

The Story of the Stolen Wine

How it all began



Credential Sharing is a Bad Idea

Letting a Client Store Credentials for other Services is a Bad Idea


IN GOD

Tokens are a Good Idea but Managing Them is Hard

The Basics

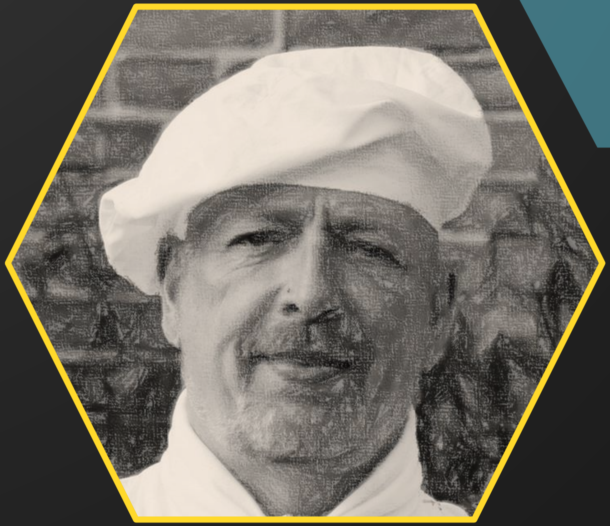
The ingredients

Resource Owner

Entity that can delegate access

Typically the end-user



Client

Third-party application that wants access to the Resource Owner's data at the Protected Resource

PUBLIC
can't keep secrets

CONFIDENTIAL
can keep secrets

Protected Resource

Server (usually an API) that holds data of the Resource Owner




Authorization Server

Central authorization point where the Resource Owner delegates access to the Client


Resource Owner
Client
Authorization Server
Protected Resource

The spices

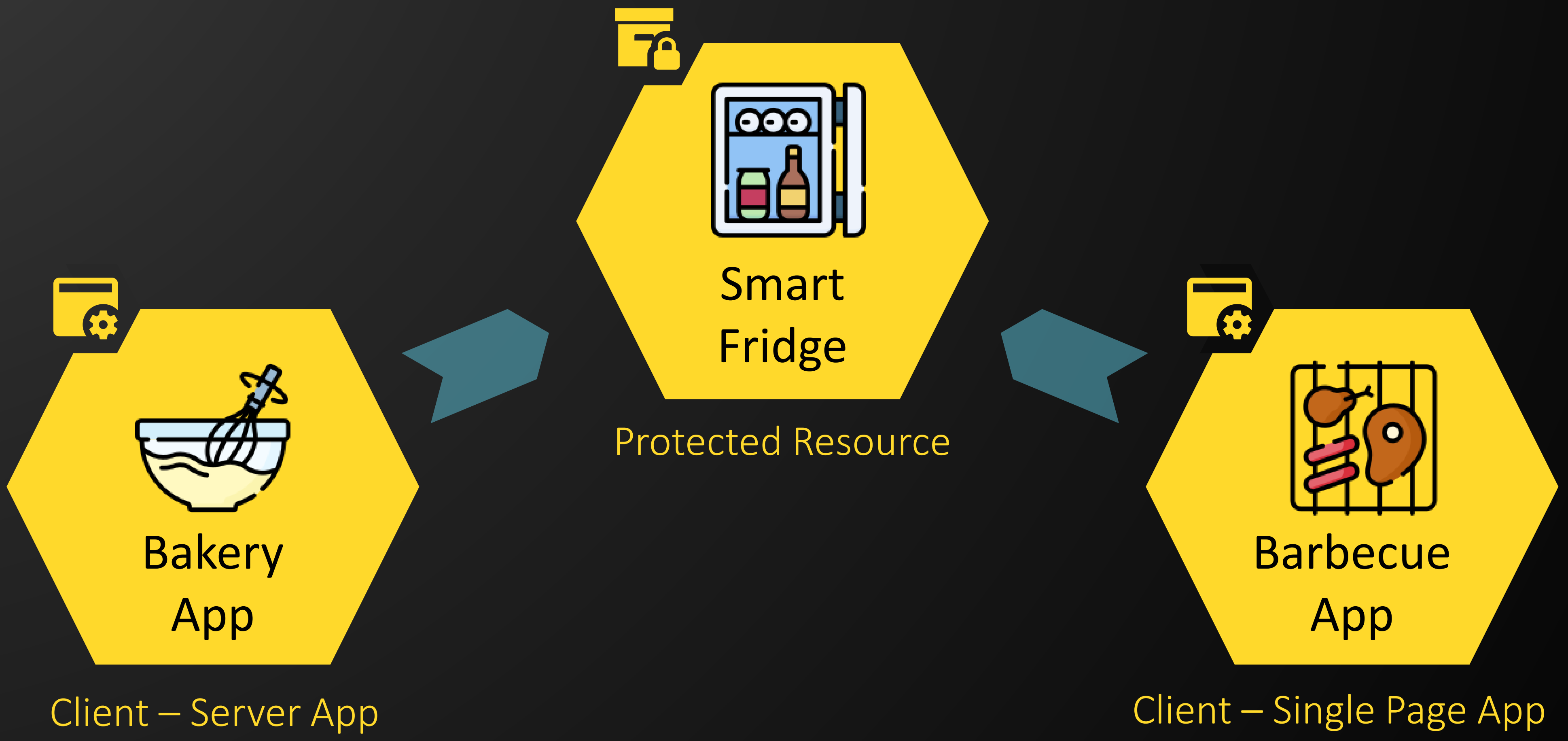
ACCESS
represents delegated access, short-lived

REFRESH
allows to reauthenticate without re-requesting Resource Owner approval, long-lived

Must be Bearer token but no specific format defined

Can be random alphanumeric value or JSON Web Token

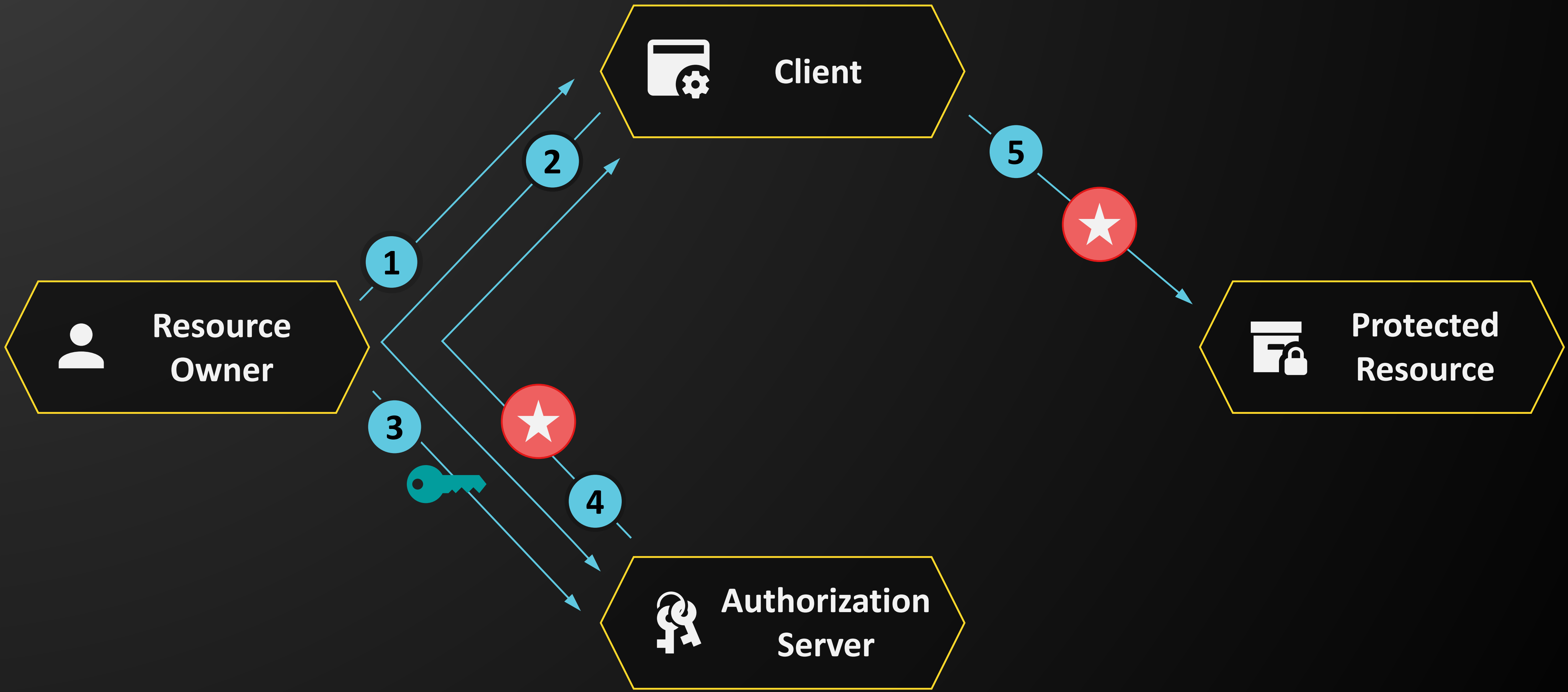
The recipe – What we are going to cook today

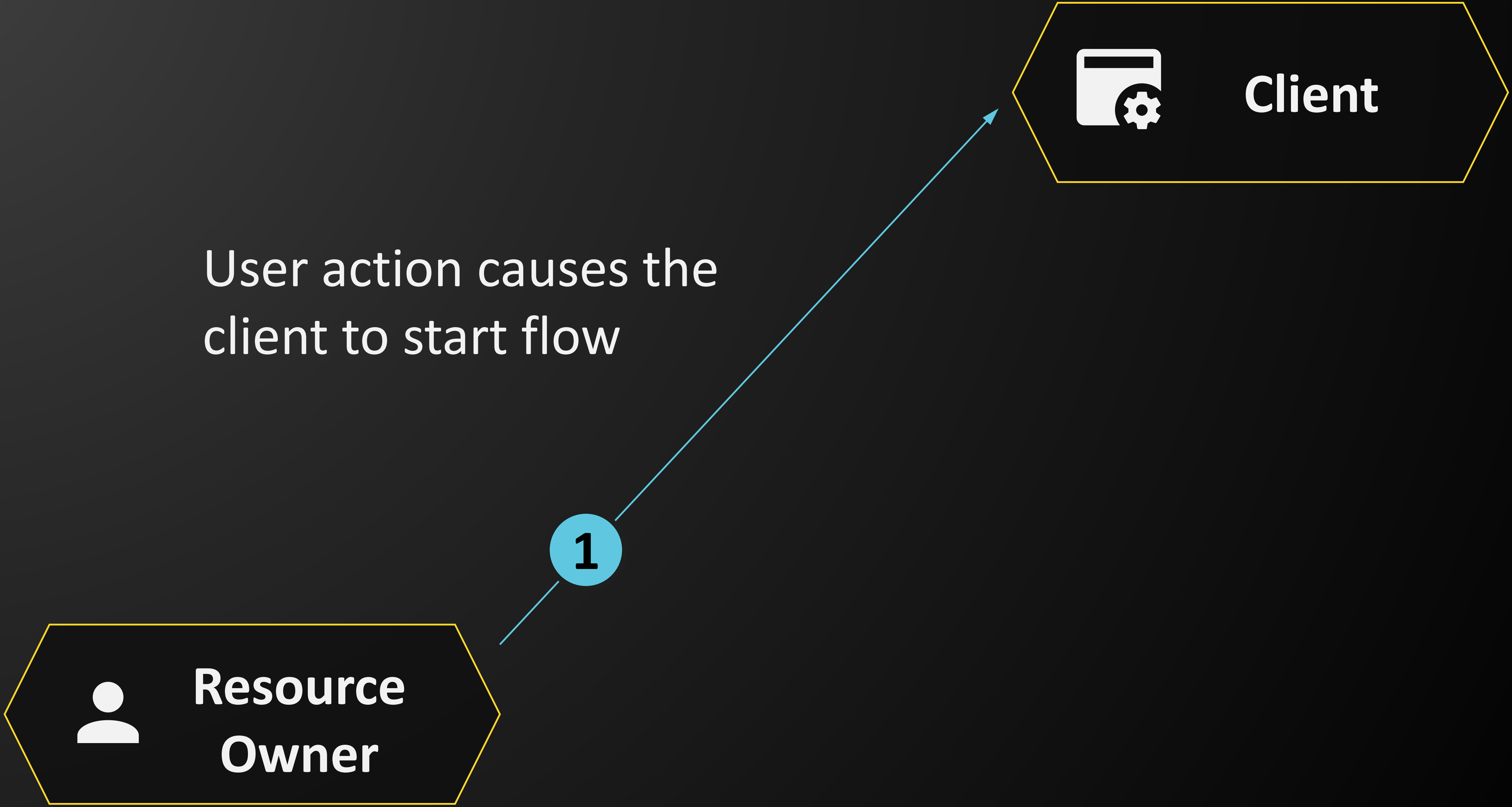


The Grant Types

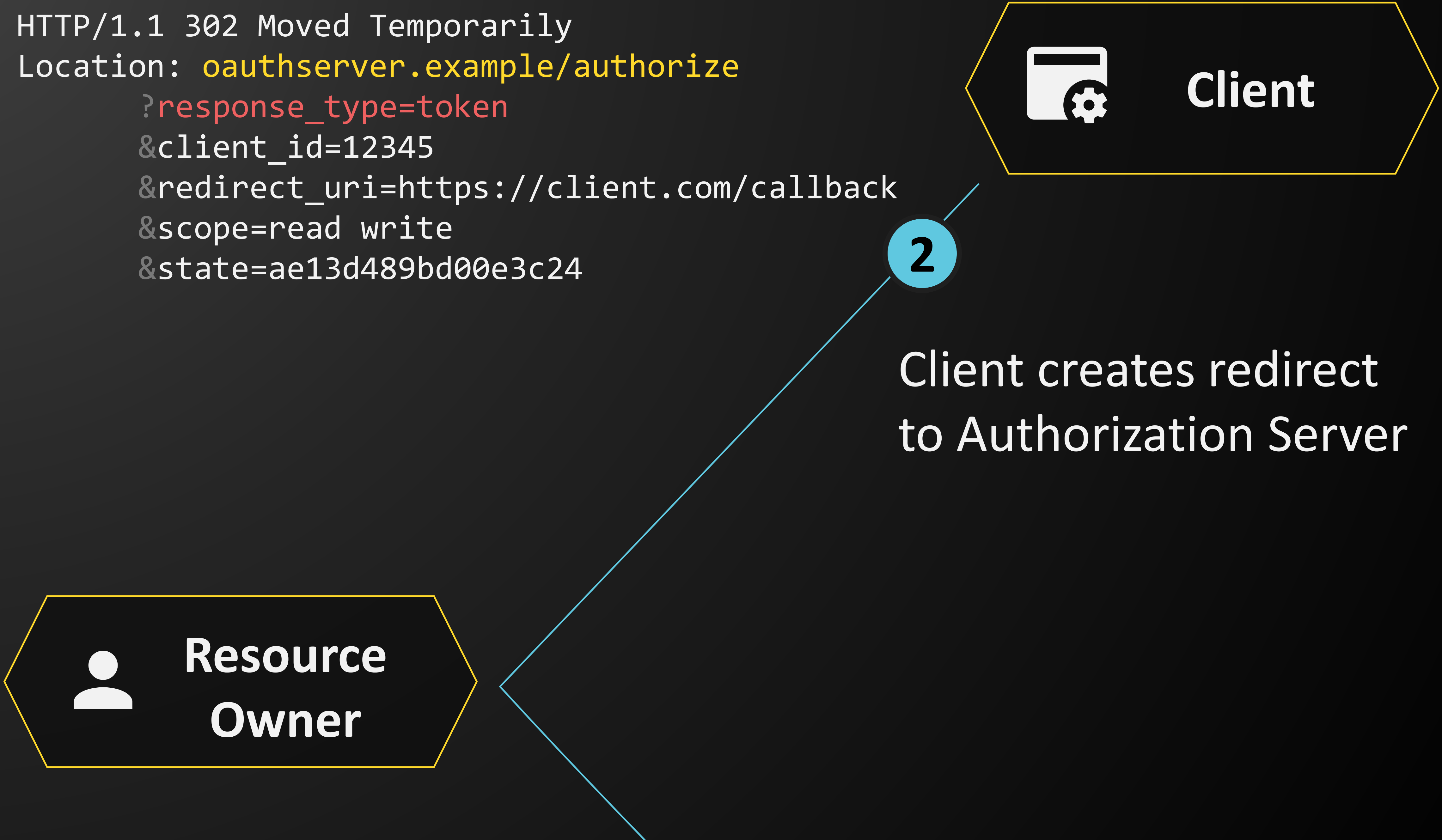
"Flows"

Appetizer – Implicit flow




Client
User action causes the client to start flow
1
Resource Owner

```
HTTP/1.1 302 Moved Temporarily
Location: oauthserver.example/authorize
      ?response_type=token
      &client_id=12345
      &redirect_uri=https://client.com/callback
      &scope=read write
      &state=ae13d489bd00e3c24
```



ID that has been issued by the Authorization Server when the Client was registered there

API-dependent

URL to Authorization Server should redirect after successful authorization

optional

set of requested permissions

recommended

random alphanumeric string as anti-CSRF protection

```
HTTP/1.1 302 Moved Temporarily
Location: oauthserver.example/authorize
     ?response_type=token
     &client_id=12345
     &redirect_uri=https://client.com/callback
     &scope=read write
     &state=ae13d489bd00e3c24
```

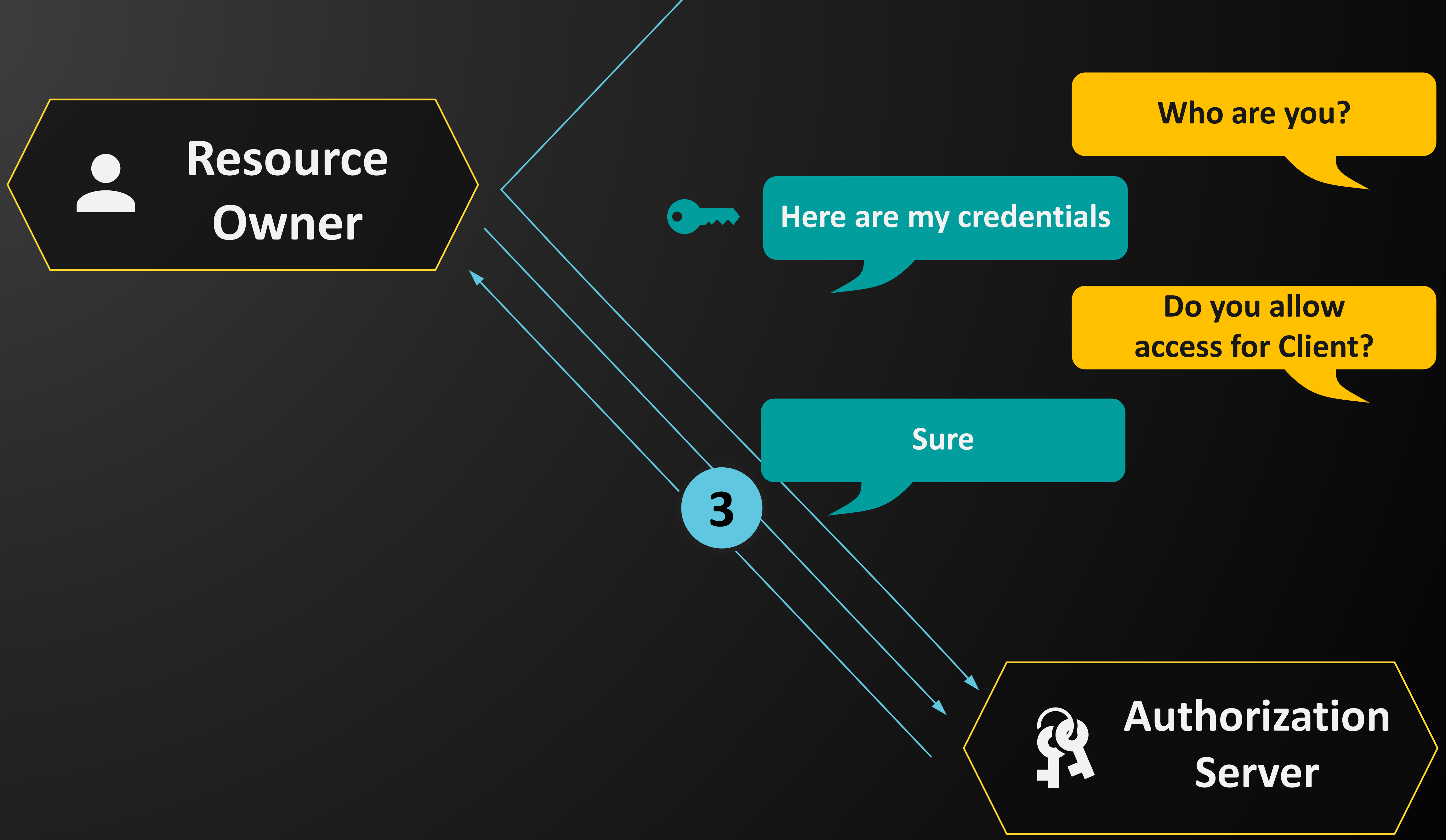
```
HTTP/1.1 302 Moved Temporarily
Location: oauthserver.example/authorize
      ?response_type=token
      &client_id=12345
      &redirect_uri=https://client.com/callback
      &scope=read write
      &state=ae13d489bd00e3c24
```

Client

2

Client creates redirect
to Authorization Server

Resource Owner


Resource
Owner
Who are you?
Here are my credentials
Do you allow
access for Client?
Sure
3
Authorization
Server

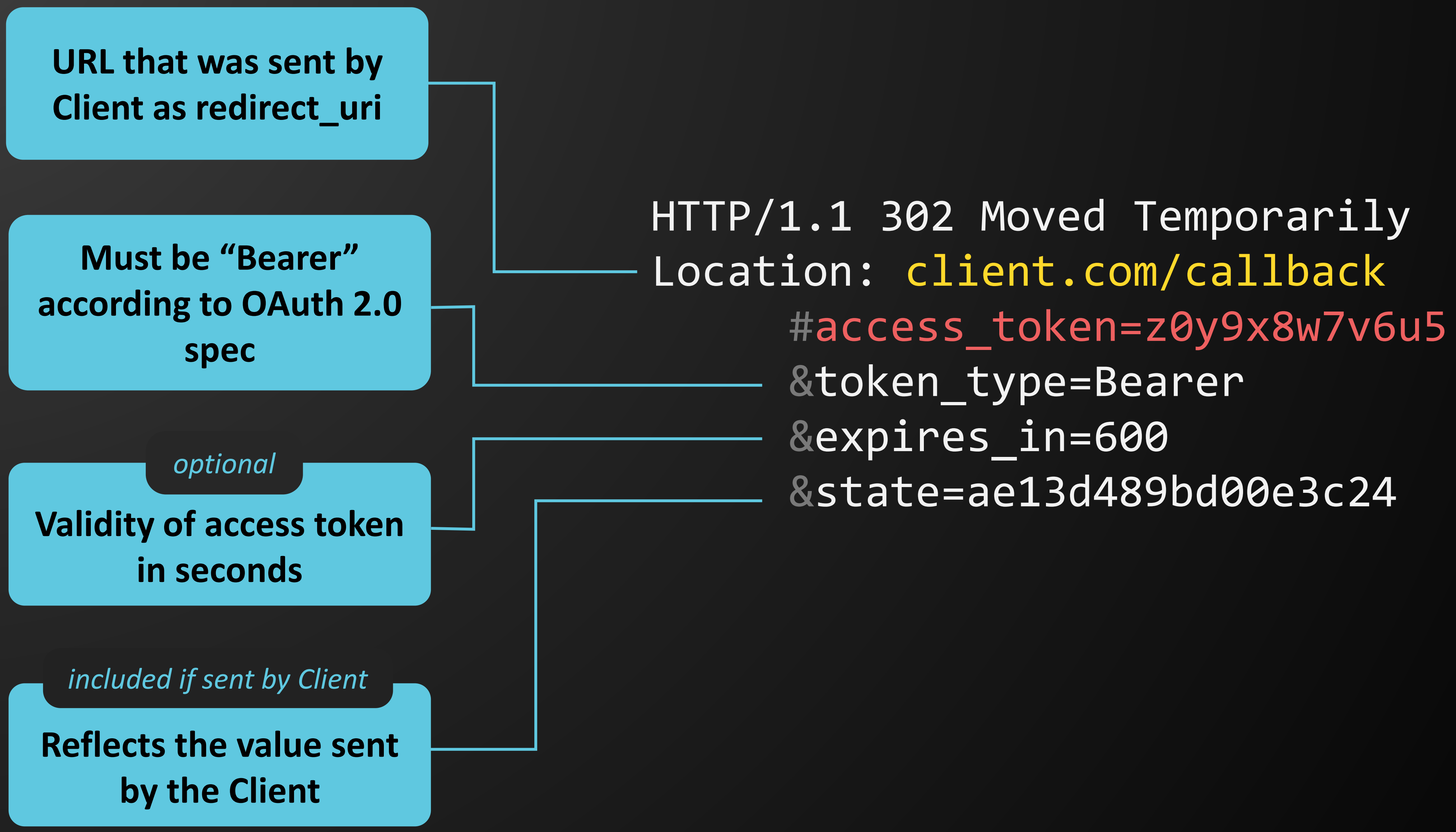
Resource Owner

Authorization Server creates redirect to Client with access token as URL fragment

```
HTTP/1.1 302 Moved Temporarily
Location: client.com/callback
      #access_token=z0y9x8w7v6u5
      &token_type=Bearer
      &expires_in=600
      &state=ae13d489bd00e3c24
```

4

Authorization Server


URL that was sent by Client as redirect_uri
Must be "Bearer" according to OAuth 2.0 spec
optional
Validity of access token in seconds
included if sent by Client
Reflects the value sent by the Client
HTTP/1.1 302 Moved Temporarily
Location: client.com/callback
#access_token=z0y9x8w7v6u5
&token_type=Bearer
&expires_in=600
&state=ae13d489bd00e3c24

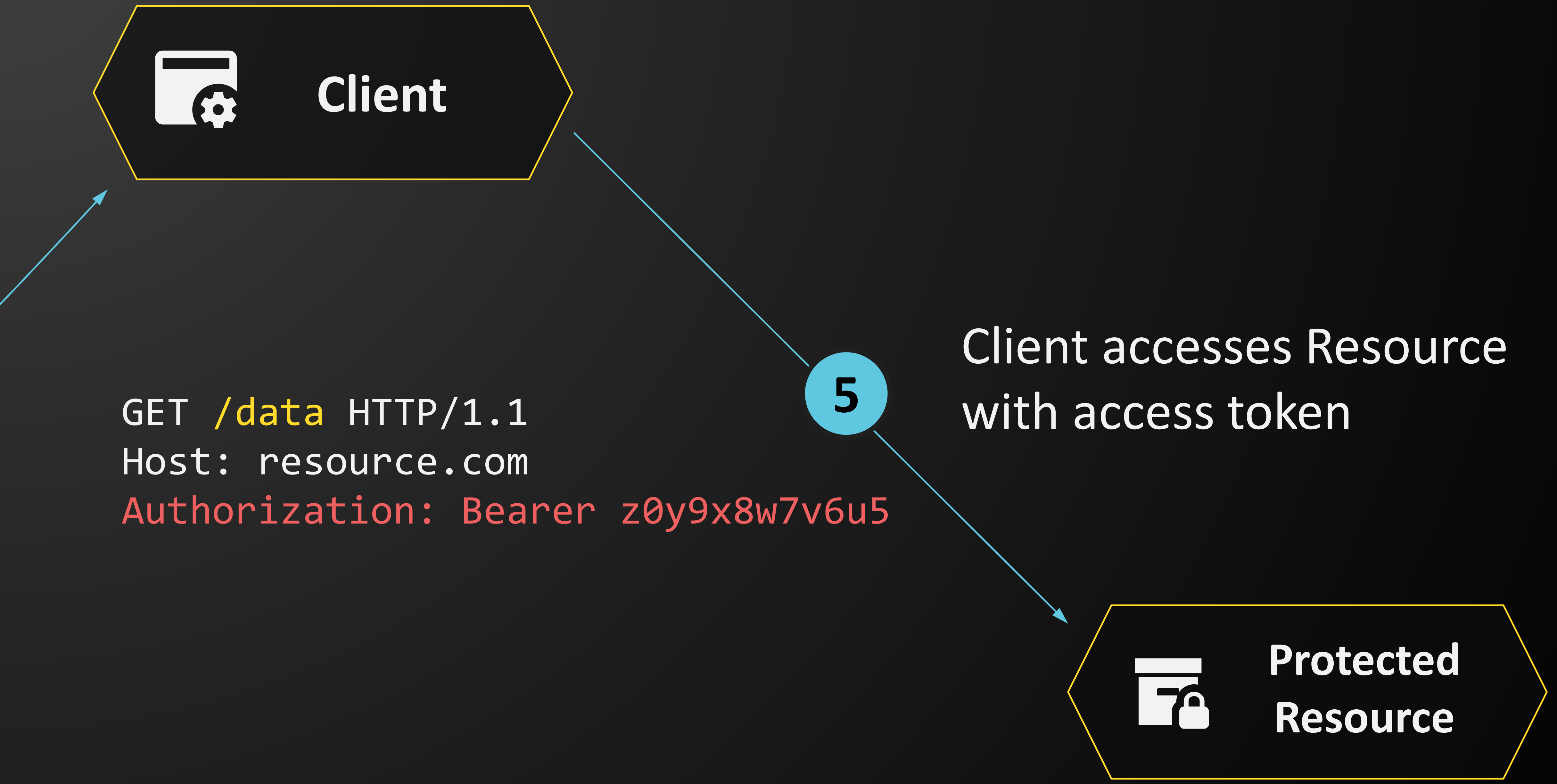
Resource Owner

Authorization Server creates redirect to Client with access token as URL fragment

```
HTTP/1.1 302 Moved Temporarily
Location: client.com/callback
        #access_token=z0y9x8w7v6u5
        &token_type=Bearer
        &expires_in=600
        &state=ae13d489bd00e3c24
```

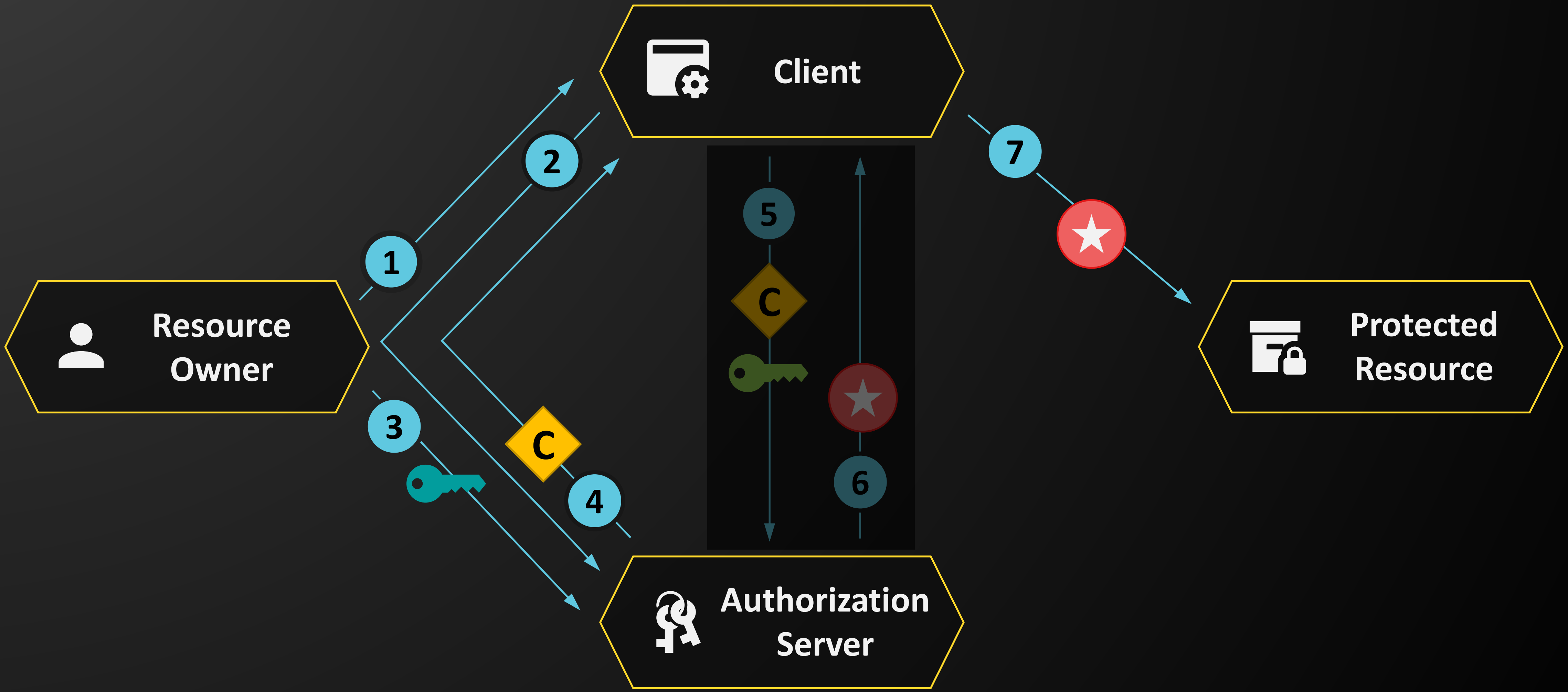
4

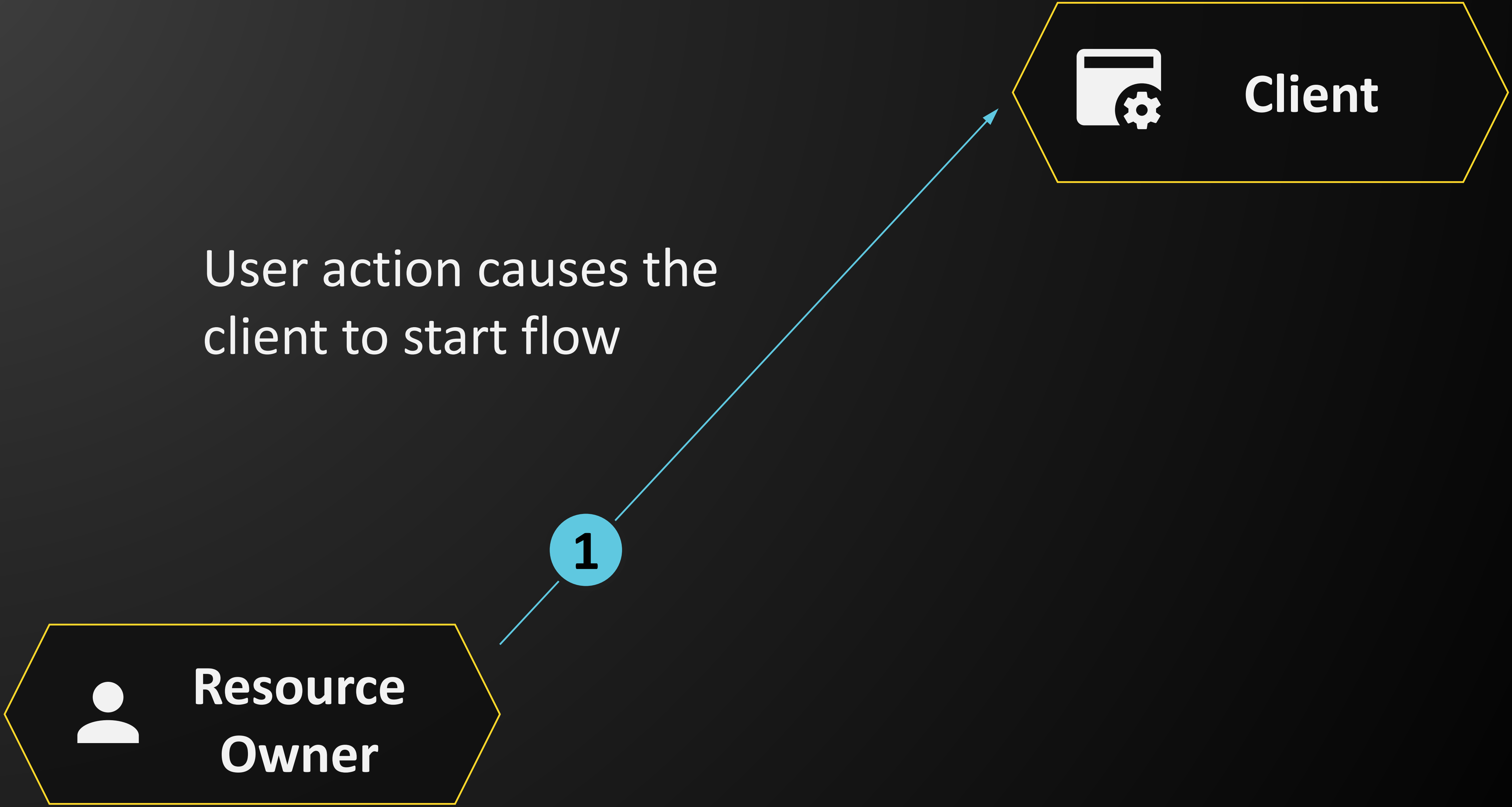
Authorization Server


Client
5
Client accesses Resource
with access token
GET /data HTTP/1.1
Host: resource.com
Authorization: Bearer z0y9x8w7v6u5
Protected
Resource


Let’s cook!

Main course – Authorization code flow




Client
User action causes the
client to start flow
1
Resource
Owner

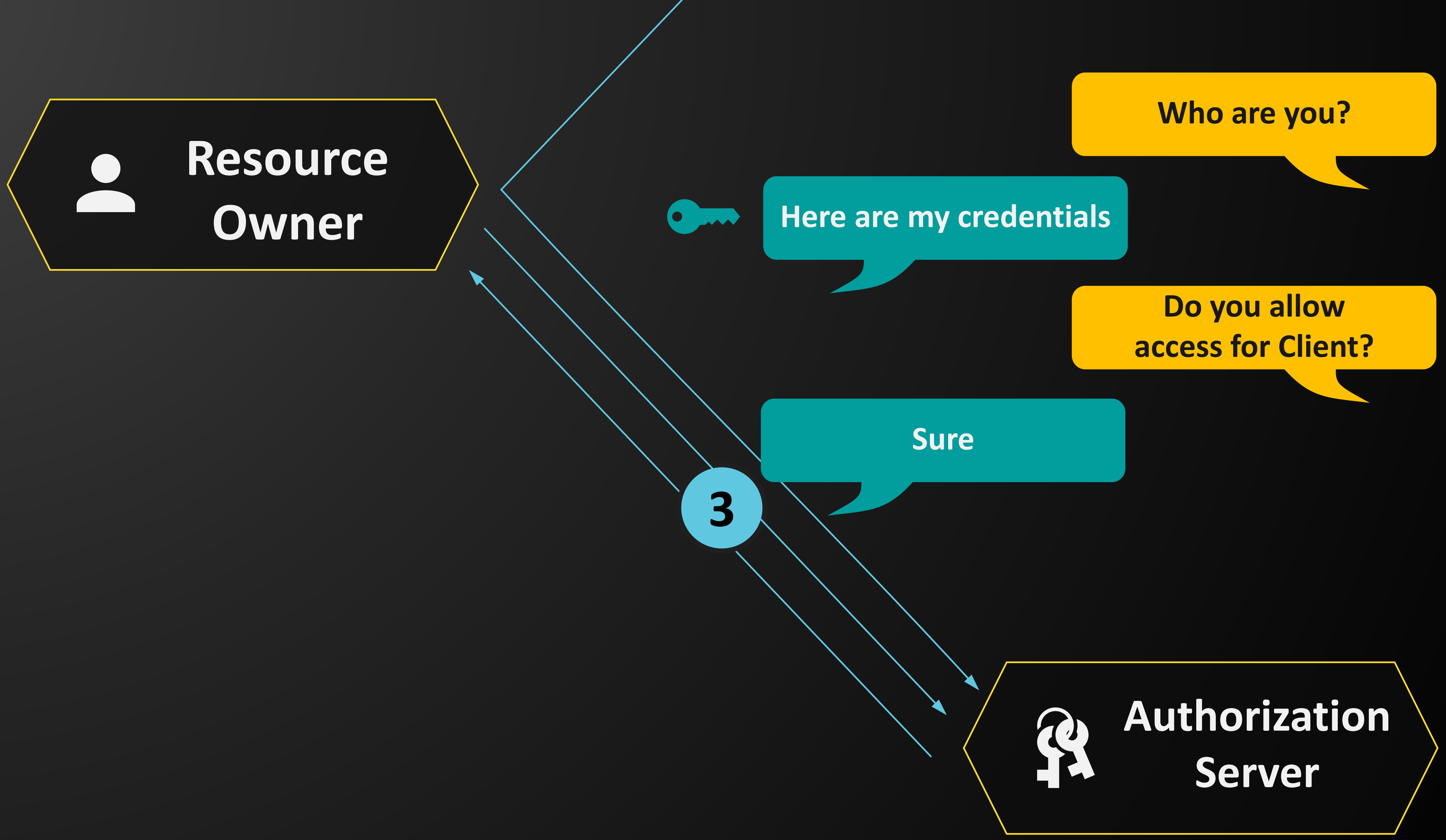
```
HTTP/1.1 302 Moved Temporarily
Location: oauthserver.example/authorize
      ?response_type=code
      &client_id=12345
      &redirect_uri=http://client.com/callback
      &scope=read write
      &state=ae13d489bd00e3c24
```

Client

2

Client creates redirect to Authorization Server

Resource Owner


Resource
Owner
Who are you?
Here are my credentials
Do you allow
access for Client?
Sure
3
Authorization
Server

Resource Owner

Authorization Server creates redirect to Client with temporary authorization code

```
HTTP/1.1 302 Moved Temporarily
Location: client.com/callback
          ?code=z0y9x8w7v6u5
          &state=ae13d489bd00e3c24
```

4

Authorization Server

Client

Client sends code to Authorization Server to retrieve token

5

```
POST /token HTTP/1.1
Host: oauthserver.com

grant_type=authorization_code
&code=z0y9x8w7v6u5
&redirect_uri=https://client.com/callback
&client_id=12345
&client_secret=xyzab
```

Can be sent as POST body parameters or via HTTP Basic Auth, depending on Authorization Server

```
POST /token HTTP/1.1
Host: oauthserver.com

grant_type=authorization_code
&code=z0y9x8w7v6u5
&redirect_uri=https://client.com/callback
&client_id=12345
&client_secret=xyzab
```

Client

5

```
POST /token HTTP/1.1
Host: oauthserver.com

grant_type=authorization_code
&code=z0y9x8w7v6u5
&redirect_uri=https://client.com/callback
&client_id=12345
&client_secret=xyzab
```

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

{
  "access_token":"MTQ0NjJkZmQ",
  "token_type":"Bearer",
  "expires_in":3600,
  "refresh_token":"IwOGYzYTlmM2Yx",
  "scope":"read write"
}
```

6

Authorization Server returns token

Authorization Server

Must be “Bearer” according to OAuth 2.0 spec

recommended

Validity of access token in seconds

optional

Token to obtain a new access token once the former expires

required if different from requested

granted permissions

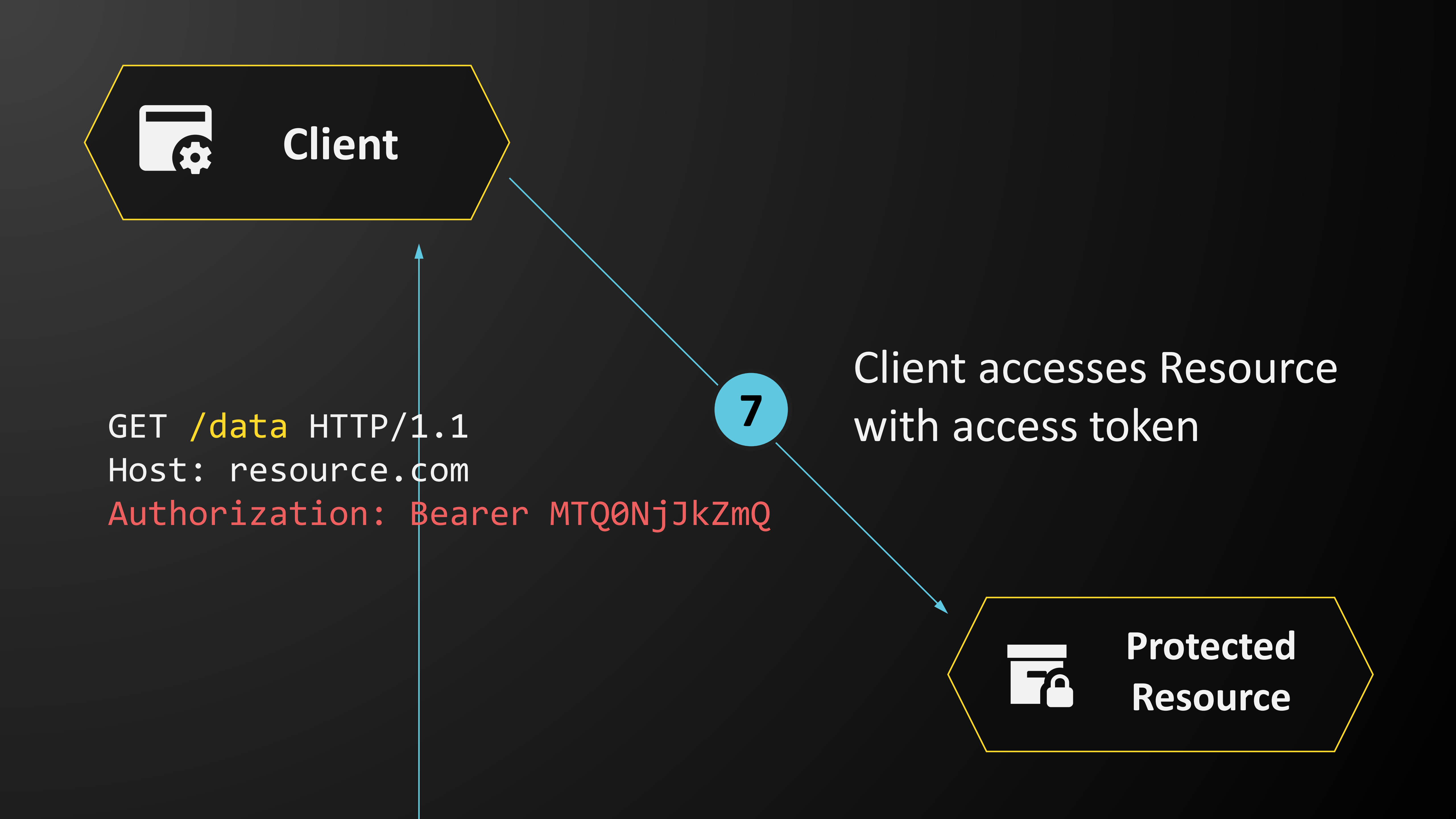
```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "access_token":"MTQ0NjJkZmQ",
  "token_type":"Bearer",
  "expires_in":3600,
  "refresh_token":"IwOGYzYTlmM2Yx",
  "scope":"read write"
}
```

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

{
  "access_token":"MTQ0NjJkZmQ",
  "token_type":"Bearer",
  "expires_in":3600,
  "refresh_token":"IwOGYzYTlmM2Yx",
  "scope":"read write"
}
```

6

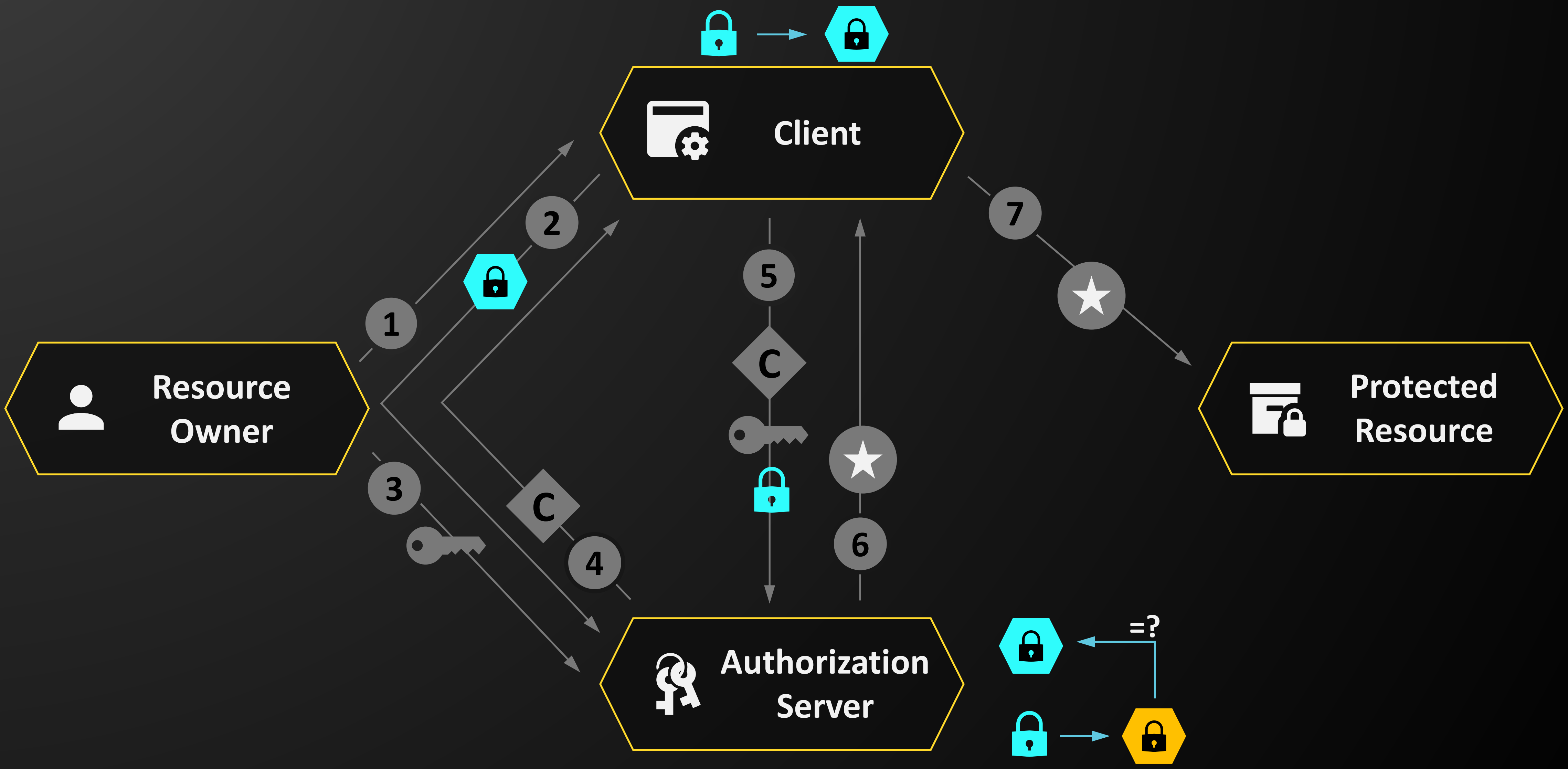
Authorization Server returns token

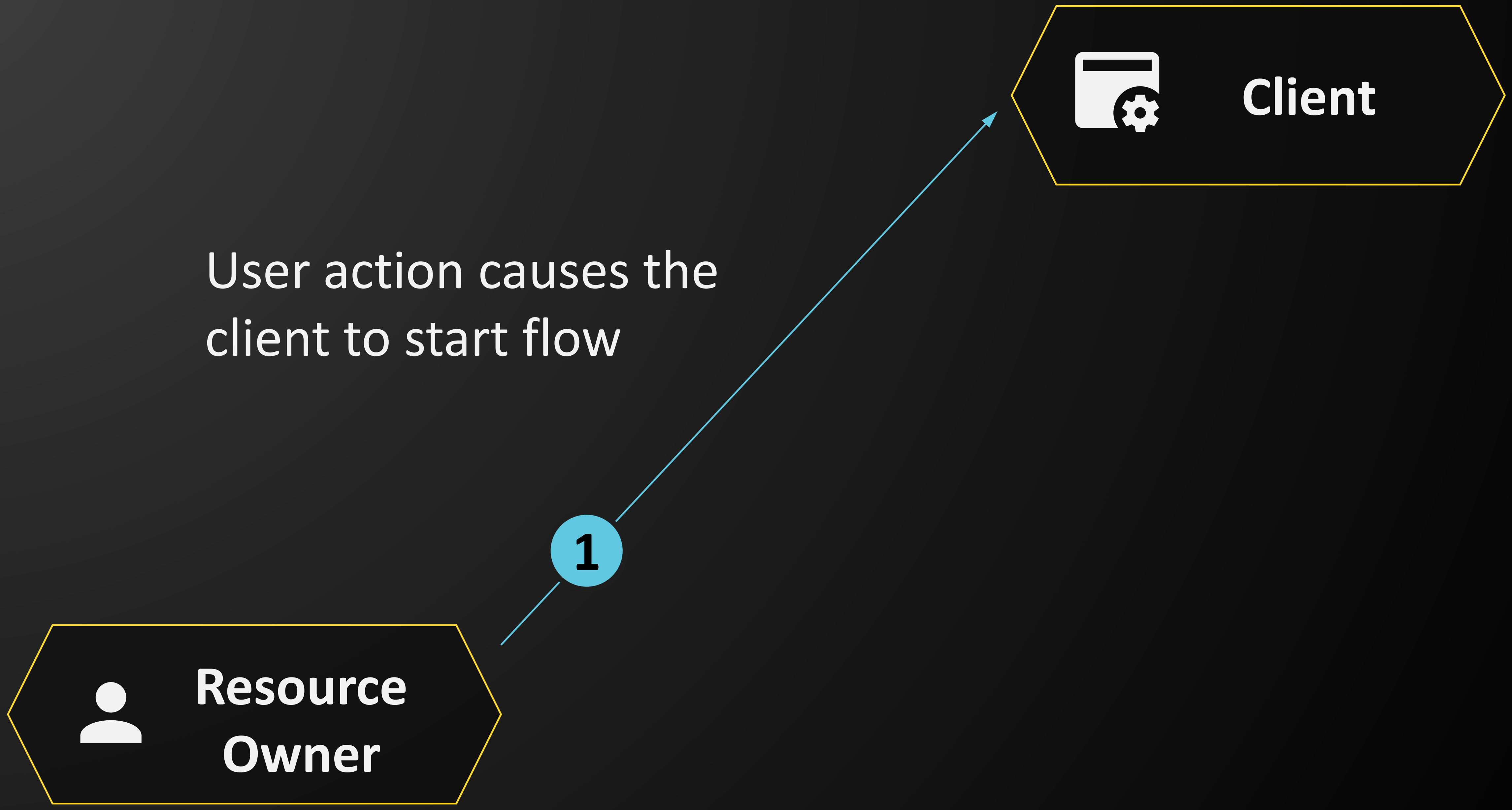
Authorization Server


Client
7
Client accesses Resource with access token
GET /data HTTP/1.1
Host: resource.com
Authorization: Bearer MTQ0NjJkZmQ
Protected Resource


Let’s cook!

Dessert – Adding some fairy dust




Client
User action causes the client to start flow
1
Resource Owner

```
HTTP/1.1 302 Moved Temporarily
Location: oauthserver.example/authorize
        ?response_type=code
        &client_id=12345
        &redirect_uri=http://client.com/callback
        &scope=read write
        &state=ae13d489bd00e3c24
        &code_challenge=9f86d081884c7
        &code_challenge_method=s256
```

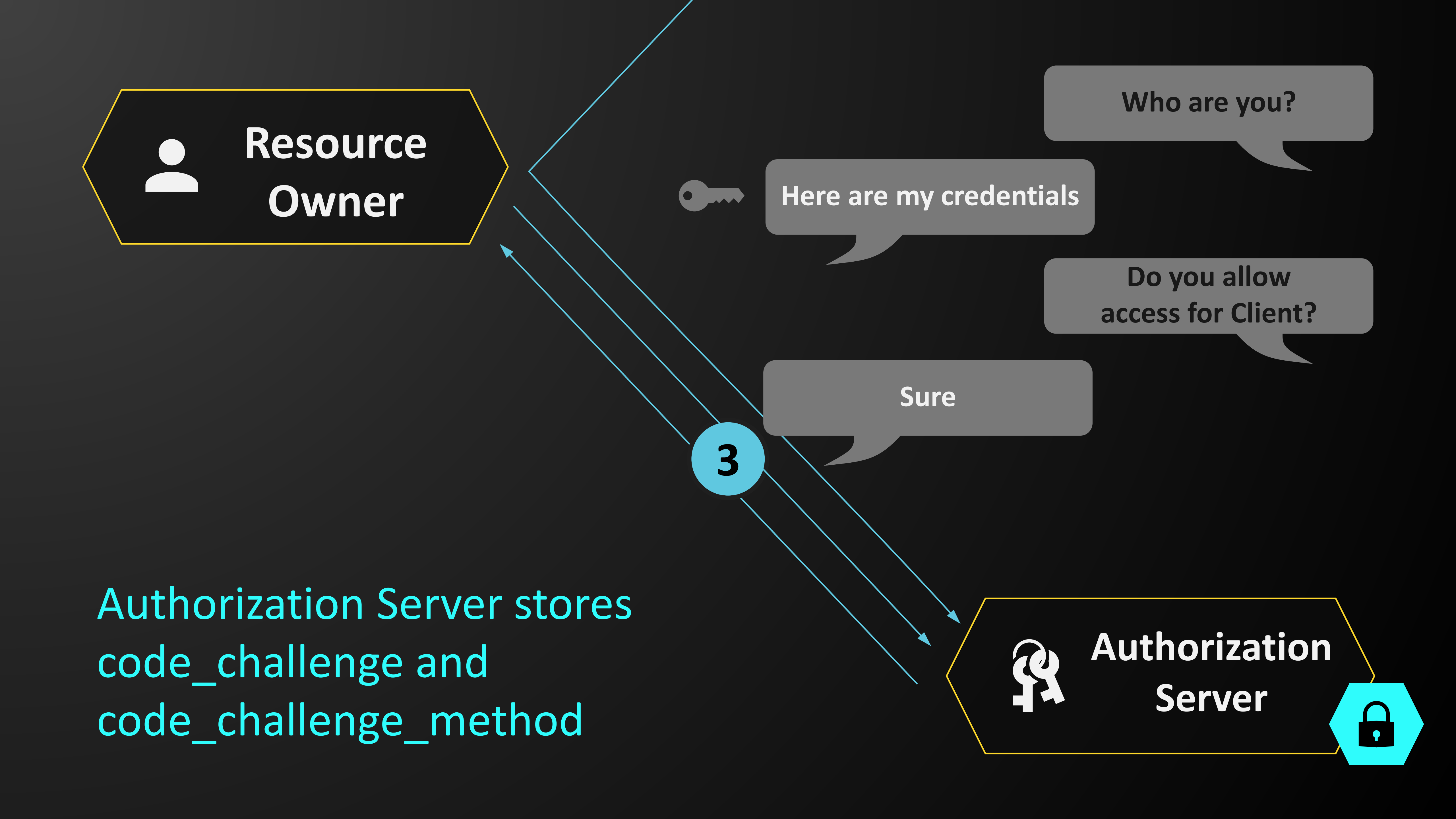
Client

2

Client creates and stores a code_verifier and creates a SHA256 hash of it as code_challenge

Client creates redirect to Authorization Server

Resource Owner


Resource Owner
Who are you?
Here are my credentials
Do you allow access for Client?
Sure
3
Authorization Server
Authorization Server stores code_challenge and code_challenge_method

Resource Owner

Authorization Server creates redirect to Client with temporary authorization code

```
HTTP/1.1 302 Moved Temporarily
Location: client.com/callback
         ?code=z0y9x8w7v6u5
         &state=ae13d489bd00e3c24
```

4

Authorization Server

5

Client sends code and code_verifier to Authorization Server to retrieve token

```
POST /token HTTP/1.1
Host: oauthserver.com

grant_type=authorization_code
&code=z0y9x8w7v6u5
&redirect_uri=https://client.com/callback
&code_verifier=a1b2c3d4
&client_id=12345
&client_secret=xyzab
```

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

{
  "access_token":"MTQ0NjJkZmQ",
  "token_type":"Bearer",
  "expires_in":3600,
  "refresh_token":"IwOGYzYTlmM2Yx",
  "scope":"read write"
}
```

Authorization Server calculates SHA256 hash of code_verifier and compares it to stored code_challenge

6

If both match, Authorization Server returns token

Authorization Server

Client

7

Client accesses Resource with access token

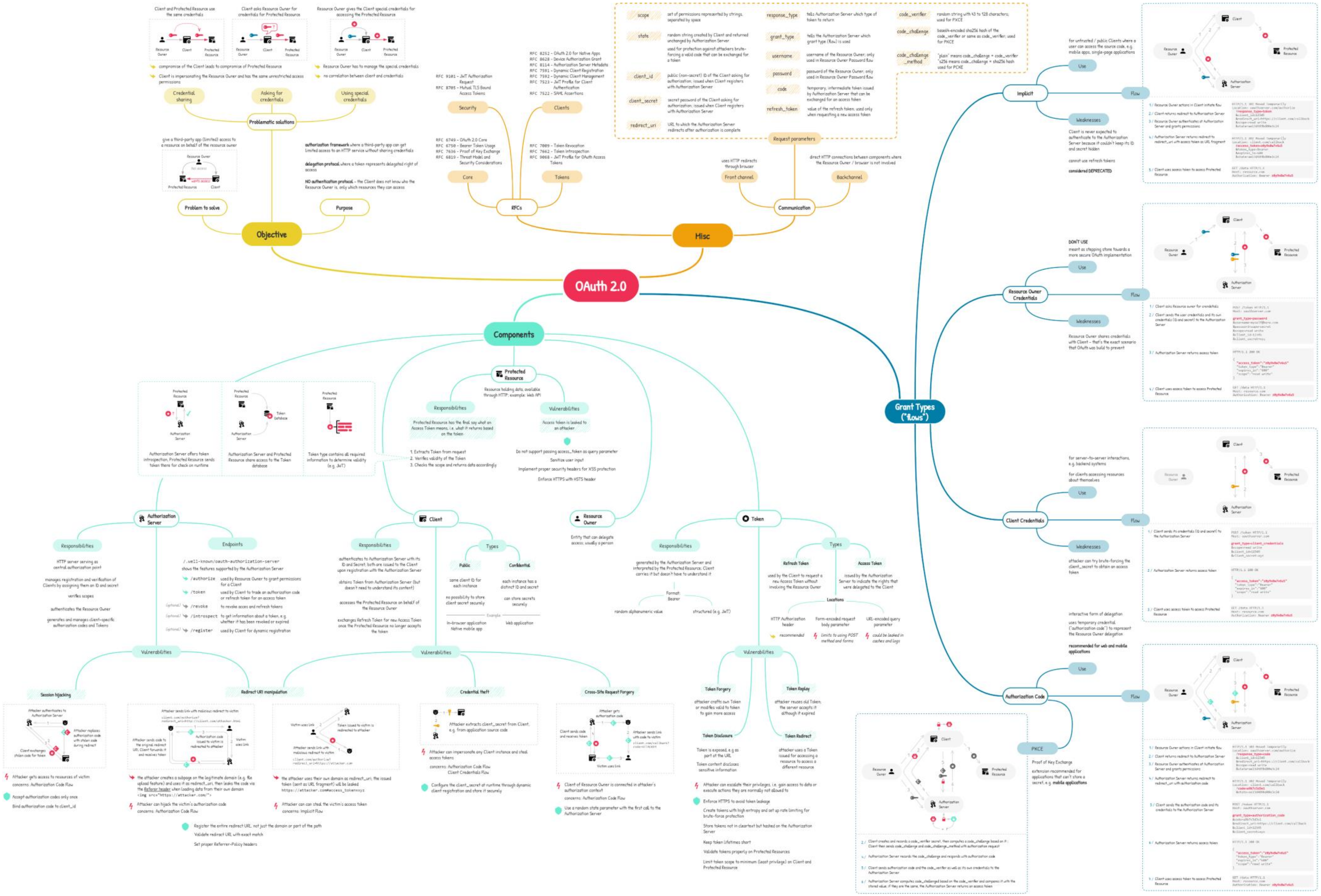
```
GET /data HTTP/1.1
Host: resource.com
Authorization: Bearer MTQ0NjJkZmQ
```

Protected Resource


Let’s cook!

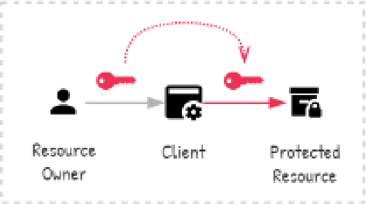
Doggy Bag

Main take-aways

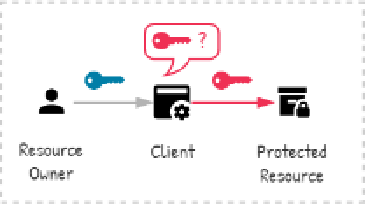


OAuth 2.0
Objective
Problem to solve
Purpose
Problematic solutions
Credential sharing
Asking for credentials
Using special credentials
Misc
RFCs
Security
Clients
Core
Tokens
Communication
Front channel
Backchannel
Request parameters
Components
Protected Resource
Responsibilities
Vulnerabilities
Authorization Server
Endpoints
Client
Types
Public
Confidential
Resource Owner
Token
Refresh Token
Access Token
Session hijacking
Redirect URI manipulation
Credential theft
Cross-Site Request Forgery
Token Forgery
Token Replay
Token Disclosure
Token Redirect
Grant Types ("flows")
Implicit
Resource Owner Credentials
Client Credentials
Authorization Code
PKCE
Use
Flow
Weaknesses

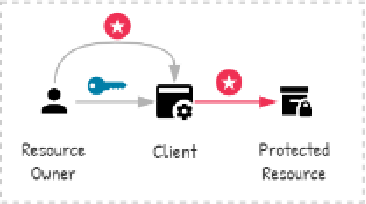
Client and Protected Resource use the same credentials



Client asks Resource Owner for credentials for Protected Resource



Resource Owner gives the Client special credentials for accessing the Protected Resource



- compromise of the Client leads to compromise of Protected Resource
- Client is impersonating the Resource Owner and has the same unrestricted access permissions

- Resource Owner has to manage the special credentials
- no correlation between client and credentials

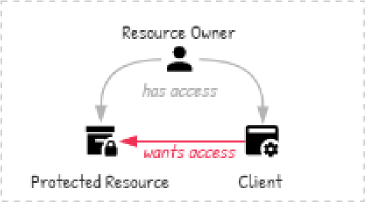
Credential sharing

Asking for credentials

Using special credentials

Problematic solutions

give a third-party app (limited) access to a resource on behalf of the resource owner



Problem to solve

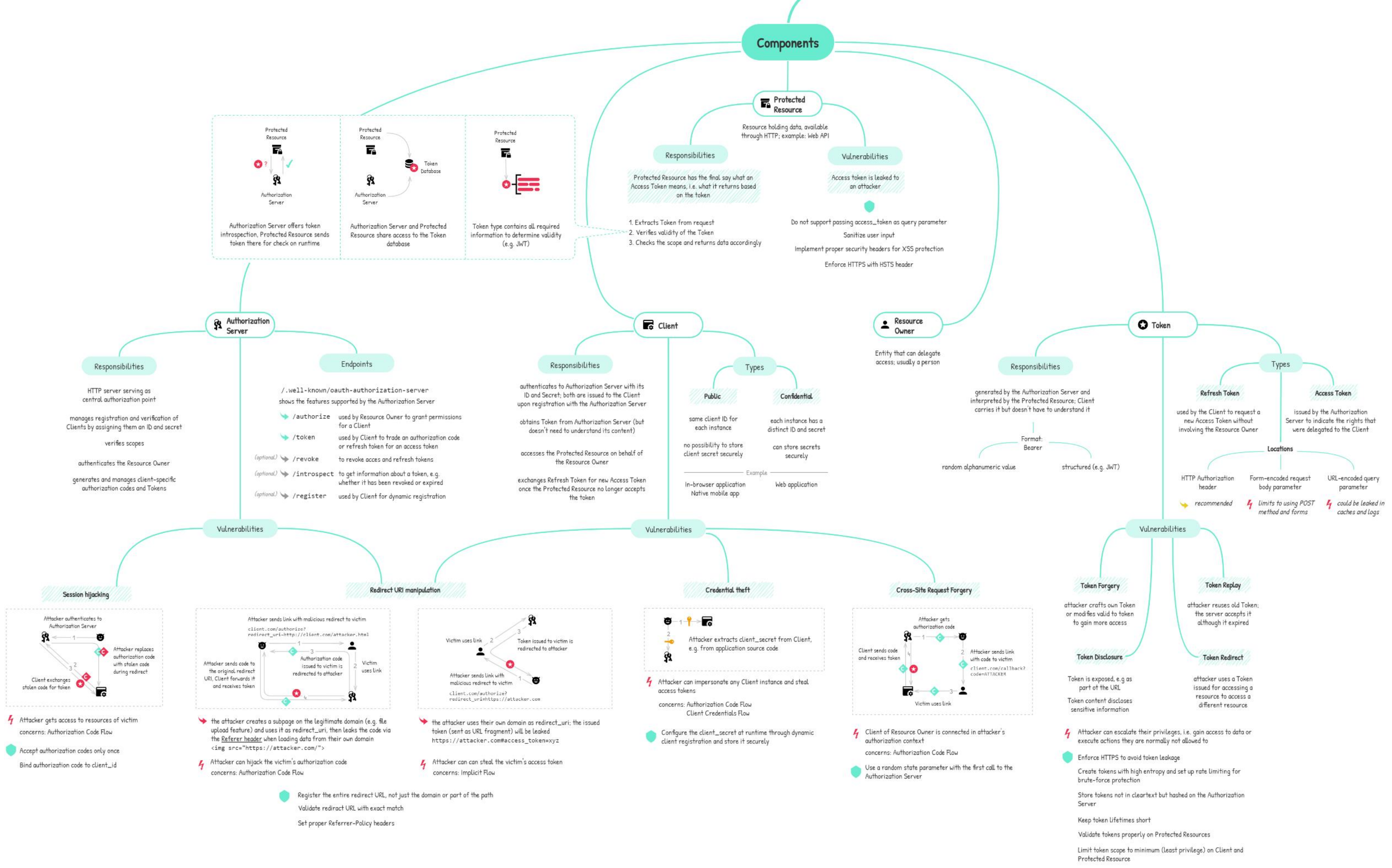
authorization framework where a third-party app can get limited access to an HTTP service without sharing credentials

delegation protocol where a token represents delegated right of access

NO authentication protocol – the Client does not know who the Resource Owner is, only which resources they can access

Purpose

Objective

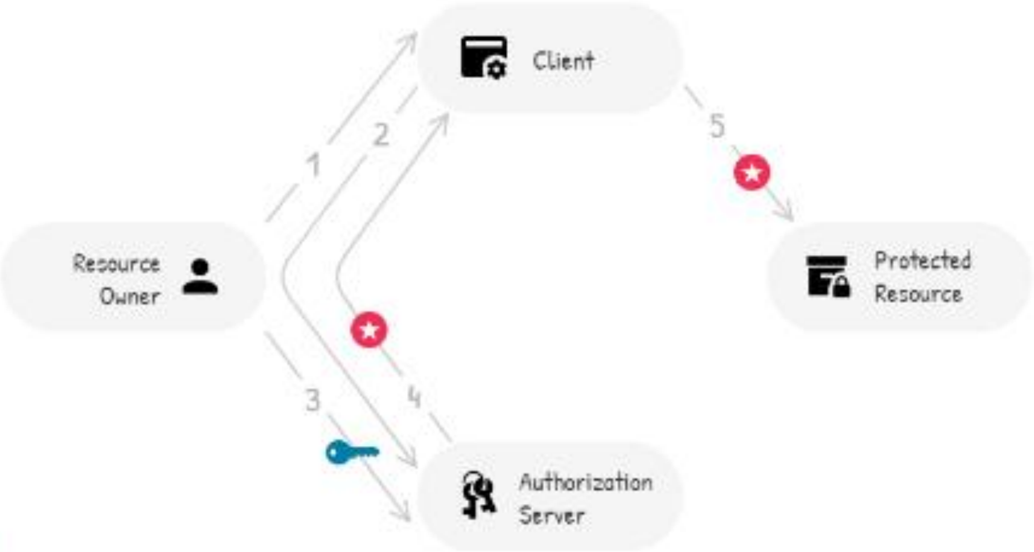

Components
Protected Resource
Resource holding data, available through HTTP; example: Web API
Responsibilities
Protected Resource has the final say what an Access Token means, i.e. what it returns based on the token
1. Extracts Token from request
2. Verifies validity of the Token
3. Checks the scope and returns data accordingly
Vulnerabilities
Access token is leaked to an attacker
Do not support passing access_token as query parameter
Sanitize user input
Implement proper security headers for XSS protection
Enforce HTTPS with HSTS header
Protected Resource
Authorization Server
Authorization Server offers token introspection, Protected Resource sends token there for check on runtime
Protected Resource
Token Database
Authorization Server
Authorization Server and Protected Resource share access to the Token database
Protected Resource
Token type contains all required information to determine validity (e.g. JWT)
Authorization Server
Responsibilities
HTTP server serving as central authorization point
manages registration and verification of Clients by assigning them an ID and secret
verifies scopes
authenticates the Resource Owner
generates and manages client-specific authorization codes and Tokens
Endpoints
/.well-known/oauth-authorization-server
shows the features supported by the Authorization Server
/authorize used by Resource Owner to grant permissions for a Client
/token used by Client to trade an authorization code or refresh token for an access token
(optional) /revoke to revoke acces and refresh tokens
(optional) /introspect to get information about a token, e.g. whether it has been revoked or expired
(optional) /register used by Client for dynamic registration
Vulnerabilities
Session hijacking
Attacker authenticates to Authorization Server
Attacker replaces authorization code with stolen code during redirect
Client exchanges stolen code for token
Attacker gets access to resources of victim
concerns: Authorization Code Flow
Accept authorization codes only once
Bind authorization code to client_id
Redirect URI manipulation
Attacker sends link with malicious redirect to victim
client.com/authorize?redirect_uri=http://client.com/attacker.html
Attacker sends code to the original redirect URI, Client forwards it and receives token
Authorization code issued to victim is redirected to attacker
Victim uses link
the attacker creates a subpage on the legitimate domain (e.g. file upload feature) and uses it as redirect_uri, then leaks the code via the Referer header when loading data from their own domain <img src="https://attacker.com/">
Attacker can hijack the victim's authorization code
concerns: Authorization Code Flow
Victim uses link
Token issued to victim is redirected to attacker
Attacker sends link with malicious redirect to victim
client.com/authorize?redirect_uri=https://attacker.com
the attacker uses their own domain as redirect_uri; the issued token (sent as URL fragment) will be leaked
https://attacker.com#access_token=xyz
Attacker can can steal the victim's access token
concerns: Implicit Flow
Register the entire redirect URL, not just the domain or part of the path
Validate redirect URL with exact match
Set proper Referrer-Policy headers
Client
Responsibilities
authenticates to Authorization Server with its ID and Secret; both are issued to the Client upon registration with the Authorization Server
obtains Token from Authorization Server (but doesn't need to understand its content)
accesses the Protected Resource on behalf of the Resource Owner
exchanges Refresh Token for new Access Token once the Protected Resource no longer accepts the token
Types
Public
same client ID for each instance
no possibility to store client secret securely
Confidential
each instance has a distinct ID and secret
can store secrets securely
Example
In-browser application
Native mobile app
Web application
Vulnerabilities
Credential theft
Attacker extracts client_secret from Client, e.g. from application source code
Attacker can impersonate any Client instance and steal access tokens
concerns: Authorization Code Flow
Client Credentials Flow
Configure the client_secret at runtime through dynamic client registration and store it securely
Cross-Site Request Forgery
Attacker gets authorization code
Client sends code and receives token
Attacker sends link with code to victim
client.com/callback?code=ATTACKER
Victim uses link
Client of Resource Owner is connected in attacker's authorization context
concerns: Authorization Code Flow
Use a random state parameter with the first call to the Authorization Server
Resource Owner
Entity that can delegate access; usually a person
Token
Responsibilities
generated by the Authorization Server and interpreted by the Protected Resource; Client carries it but doesn't have to understand it
Format: Bearer
random alphanumeric value
structured (e.g. JWT)
Types
Refresh Token
used by the Client to request a new Access Token without involving the Resource Owner
Access Token
issued by the Authorization Server to indicate the rights that were delegated to the Client
Locations
HTTP Authorization header
recommended
Form-encoded request body parameter
limits to using POST method and forms
URL-encoded query parameter
could be leaked in caches and logs
Vulnerabilities
Token Forgery
attacker crafts own Token or modifies valid to token to gain more access
Token Replay
attacker reuses old Token; the server accepts it although it expired
Token Disclosure
Token is exposed, e.g as part of the URL
Token content discloses sensitive information
Token Redirect
attacker uses a Token issued for accessing a resource to access a different resource
Attacker can escalate their privileges, i.e. gain access to data or execute actions they are normally not allowed to
Enforce HTTPS to avoid token leakage
Create tokens with high entropy and set up rate limiting for brute-force protection
Store tokens not in cleartext but hashed on the Authorization Server
Keep token lifetimes short
Validate tokens properly on Protected Resources
Limit token scope to minimum (least privilege) on Client and Protected Resource

Implicit

Use

for untrusted / public Clients where a user can access the source code, e.g. mobile apps, single-page applications

Flow



1 / Resource Owner actions in Client initiate flow

2 / Client returns redirect to Authorization Server

3 / Resource Owner authenticates at Authorization Server and grants permissions

```
HTTP/1.1 302 Moved Temporarily
Location: oauthserver.com/authorize
  ?response_type=token
  &client_id=12345
  &redirect_uri=https://client.com/callback
  &scope=read write
  &state=ae13d489bd00e3c24
```

4 / Authorization Server returns redirect to redirect_uri with access token as URL fragment

```
HTTP/1.1 302 Moved Temporarily
Location: client.com/callback
  #access_token=z0y9x8w7v6u5
  &token_type=Bearer
  &expires_in=600
  &state=ae13d489bd00e3c24
```

5 / Client uses access token to access Protected Resource

```
GET /data HTTP/1.1
Host: resource.com
Authorization: Bearer z0y9x8w7v6u5
```

Weaknesses

Client is never expected to authenticate to the Authorization Server because it couldn't keep its ID and secret hidden

cannot use refresh tokens

considered DEPRECATED

Authorization Code

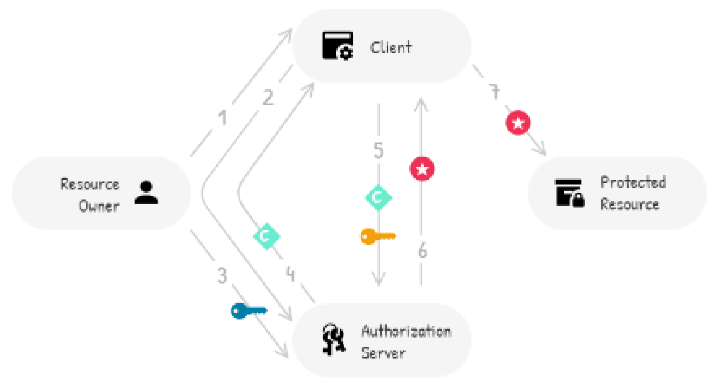
Use

interactive form of delegation

uses temporary credential ("authorization code") to represent the Resource Owner delegation

recommended for web and mobile applications

Flow



1 / Resource Owner actions in Client initiate flow

2 / Client returns redirect to Authorization Server

```
HTTP/1.1 302 Moved Temporarily
Location: oauthserver.com/authorize
  ?response_type=code
  &client_id=12345
  &redirect_uri=https://client.com/callback
  &scope=read write
  &state=ae13d489bd00e3c24
```

3 / Resource Owner authenticates at Authorization Server and grants permissions

4 / Authorization Server returns redirect to redirect_uri with authorization code

```
HTTP/1.1 302 Moved Temporarily
Location: client.com/callback
  ?code=a9b7c5d3e1
  &state=ae13d489bd00e3c24
```

5 / Client sends the authorization code and its credentials to the Authorization Server

```
POST /token HTTP/1.1
Host: oauthserver.com

grant_type=authorization_code
&code=a9b7c5d3e1
&redirect_uri=https://client.com/callback
&client_id=12345
&client_secret=xyz
```

6 / Authorization Server returns access token

```
HTTP/1.1 200 OK

{
  "access_token":"z0y9x8w7v6u5"
  "token_type":"Bearer"
  "expires_in":"600"
  "scope":"read write"
}
```

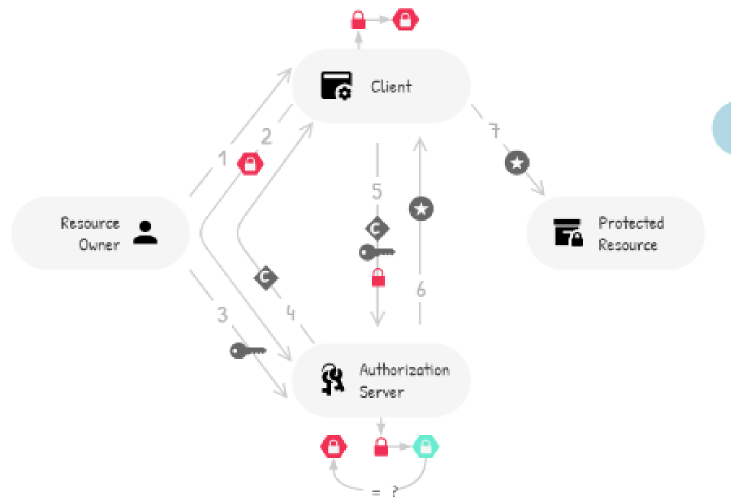
7 / Client uses access token to access Protected Resource

```
GET /data HTTP/1.1
Host: resource.com
Authorization: Bearer z0y9x8w7v6u5
```

PKCE

Proof of Key Exchange

extension recommended for applications that can't store a secret, e.g. **mobile applications**



2 / Client creates and records a code_verifier secret, then computes a code_challenge based on it ; Client then sends code_challenge and code_challenge_method with authorization request

4 / Authorization Server records the code_challenge and responds with authorization code

5 / Client sends authorization code and the code_verifier as well as its own credentials to the Authorization Server

6 / Authorization Server computes code_challenged based on the code_verifier and compares it with the stored value; if they are the same, the Authorization Server returns an access token

Grant Types ("flows")

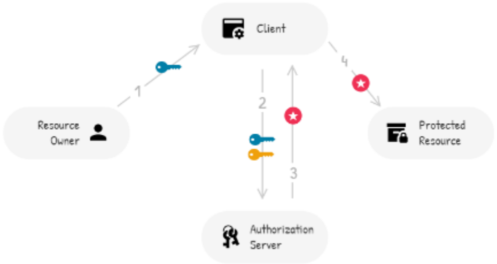
Resource Owner Credentials

Use

DON'T USE

meant as stepping stone towards a more secure OAuth implementation

Flow



1 / Client asks Resource owner for crendetials

2 / Client sends the user credentials and its own credentials (ID and secret) to the Authorization Server

```
POST /token HTTP/1.1
Host: oauthserver.com

grant_type=password
&username=myself@here.com
&password=supersecret
&scope=read write
&client_id=12345
&client_secret=xyz
```

3 / Authorization Server returns access token

```
HTTP/1.1 200 OK

{
  "access_token":"z0y9x8w7v6u5"
  "token_type":"Bearer"
  "expires_in":"600"
  "scope":"read write"
}
```

4 / Client uses access token to access Protected Resource

```
GET /data HTTP/1.1
Host: resource.com
Authorization: Bearer z0y9x8w7v6u5
```

Weaknesses

Resource Owner shares credentials with Client - that's the exact scenario that OAuth was build to prevent

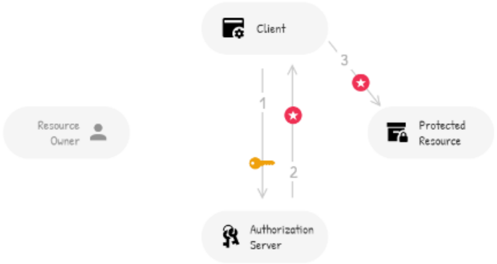
Client Credentials

Use

for server-to-server interactions, e.g. backend systems

for clients accessing resources about themselves

Flow



1 / Client sends its credentials (ID and secret) to the Authorization Server

```
POST /token HTTP/1.1
Host: oauthserver.com

grant_type=client_credentials
&scope=read write
&client_id=12345
&client_secret=xyz
```

2 / Authorization Server returns access token

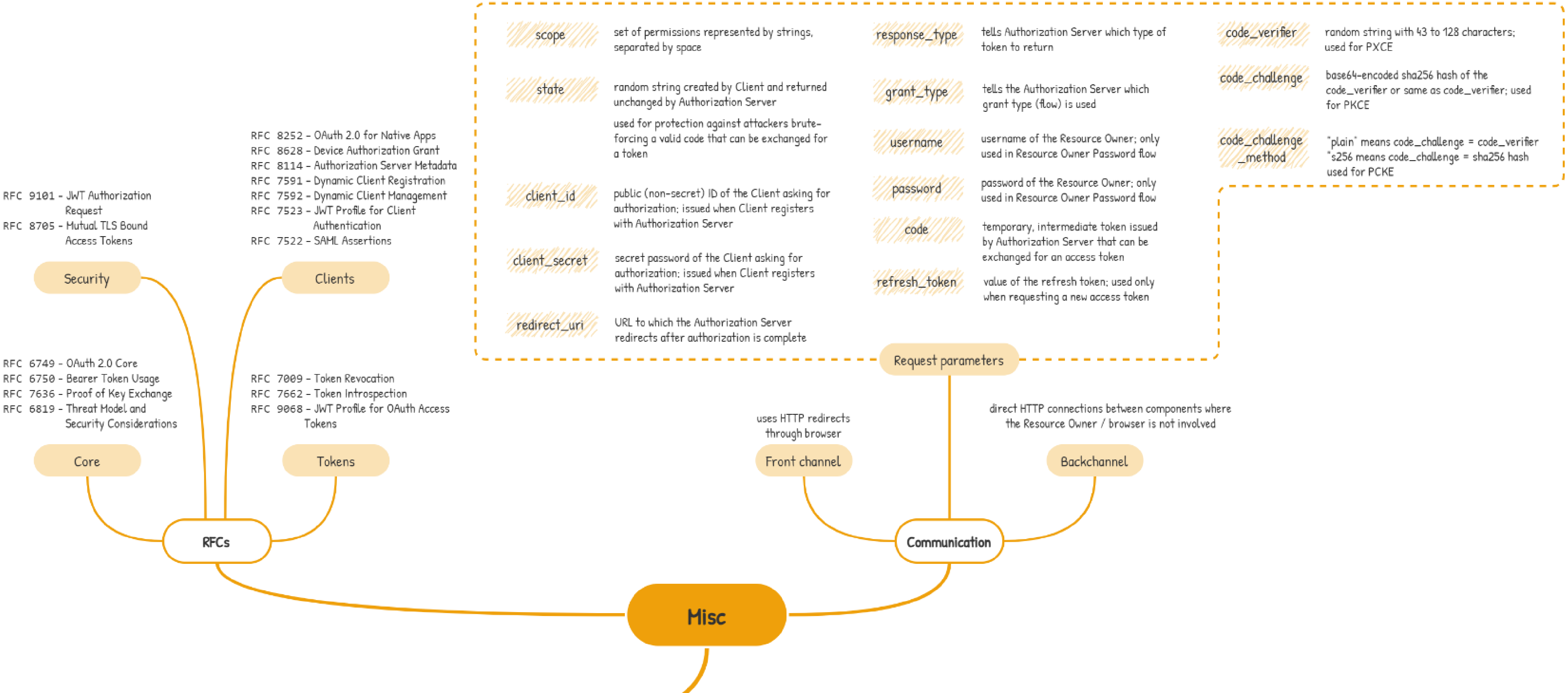
```
HTTP/1.1 200 OK

{
  "access_token":"z0y9x8w7v6u5"
  "token_type":"Bearer"
  "expires_in":"600"
  "scope":"read write"
}
```

3 / Client uses access token to access Protected Resource

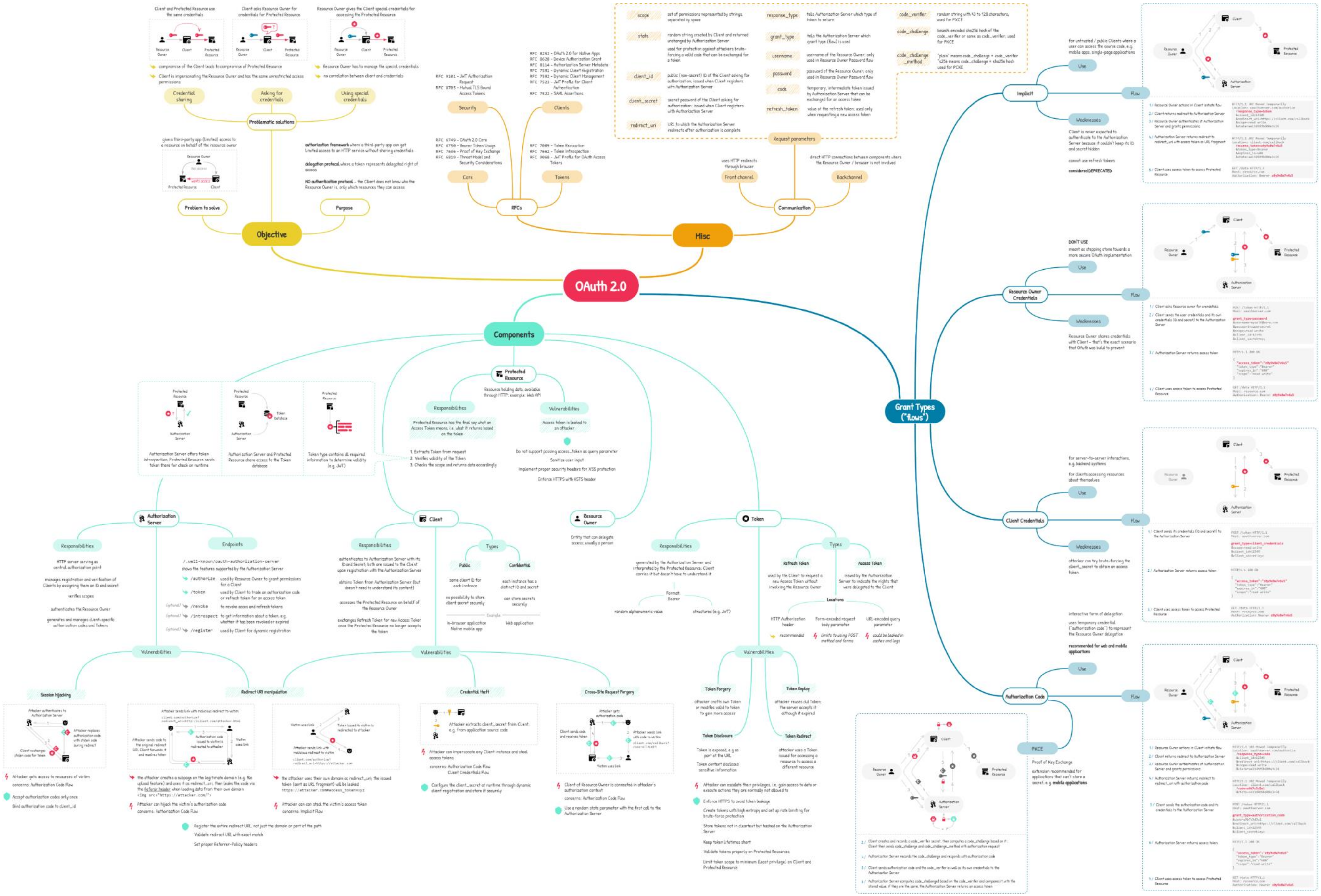
```
GET /data HTTP/1.1
Host: resource.com
Authorization: Bearer z0y9x8w7v6u5
```

Weaknesses

attacker can try brute-forcing the client_secret to obtain an access token



Misc
RFCs
Security
RFC 9101 - JWT Authorization Request
RFC 8705 - Mutual TLS Bound Access Tokens
Clients
RFC 8252 - OAuth 2.0 for Native Apps
RFC 8628 - Device Authorization Grant
RFC 8114 - Authorization Server Metadata
RFC 7591 - Dynamic Client Registration
RFC 7592 - Dynamic Client Management
RFC 7523 - JWT Profile for Client Authentication
RFC 7522 - SAML Assertions
Core
RFC 6749 - OAuth 2.0 Core
RFC 6750 - Bearer Token Usage
RFC 7636 - Proof of Key Exchange
RFC 6819 - Threat Model and Security Considerations
Tokens
RFC 7009 - Token Revocation
RFC 7662 - Token Introspection
RFC 9068 - JWT Profile for OAuth Access Tokens
Communication
Front channel
uses HTTP redirects through browser
Backchannel
direct HTTP connections between components where the Resource Owner / browser is not involved
Request parameters
scope
set of permissions represented by strings, separated by space
state
random string created by Client and returned unchanged by Authorization Server
used for protection against attackers brute-forcing a valid code that can be exchanged for a token
client_id
public (non-secret) ID of the Client asking for authorization; issued when Client registers with Authorization Server
client_secret
secret password of the Client asking for authorization; issued when Client registers with Authorization Server
redirect_uri
URL to which the Authorization Server redirects after authorization is complete
response_type
tells Authorization Server which type of token to return
grant_type
tells the Authorization Server which grant type (flow) is used
username
username of the Resource Owner; only used in Resource Owner Password flow
password
password of the Resource Owner; only used in Resource Owner Password flow
code
temporary, intermediate token issued by Authorization Server that can be exchanged for an access token
refresh_token
value of the refresh token; used only when requesting a new access token
code_verifier
random string with 43 to 128 characters; used for PXCE
code_challenge
base64-encoded sha256 hash of the code_verifier or same as code_verifier; used for PKCE
code_challenge_method
"plain" means code_challenge = code_verifier
"s256 means code_challenge = sha256 hash
used for PCKE



OAuth 2.0
Objective
Problem to solve
Purpose
Problematic solutions
Credential sharing
Asking for credentials
Using special credentials
Misc
RFCs
Security
Clients
Core
Tokens
Communication
Front channel
Backchannel
Request parameters
Components
Protected Resource
Responsibilities
Vulnerabilities
Authorization Server
Endpoints
Client
Types
Public
Confidential
Resource Owner
Token
Refresh Token
Access Token
Locations
Session hijacking
Redirect URI manipulation
Credential theft
Cross-Site Request Forgery
Token Forgery
Token Replay
Token Disclosure
Token Redirect
Grant Types ("flows")
Implicit
Resource Owner Credentials
Client Credentials
Authorization Code
PKCE
Use
Flow
Weaknesses

Blick über den Tellerrand

What else

Attacks

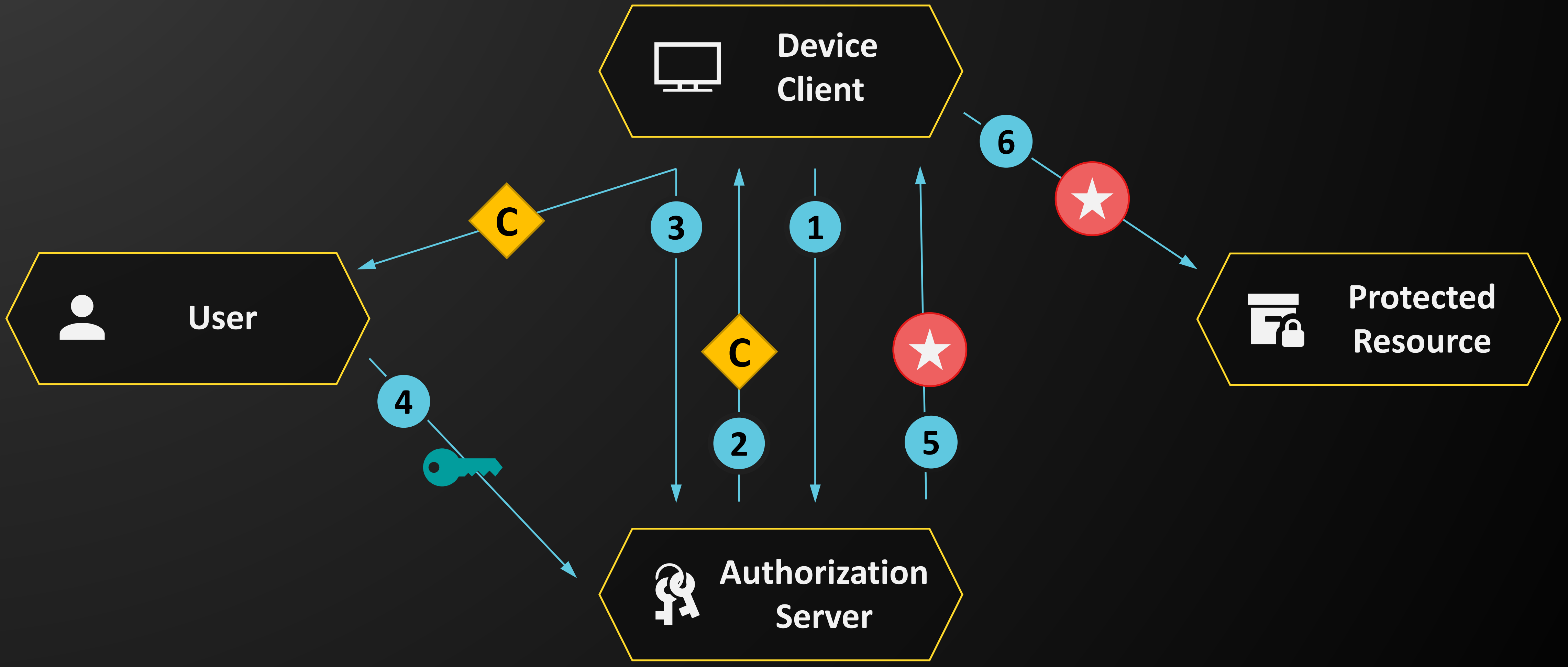
What we tested

- Open redirect through missing or flawed redirect URL checking
- Cross-Site Request Forgery
- Session hijacking through code reuse

What else there is

- Further ways for open redirect
- Exchanging code for bigger scope
- Leakage of client secret
- Leakage of token
- Token-related vulnerabilities

Device Flow


Device Client
User
Protected Resource
Authorization Server
C
C
1
2
3
4
5
6

Further Extensions

Dynamic Client Registration

- Allows registering clients at runtime
- Client receives unique ID and secret from Authorization Server
- Useful e.g. for mobile apps with multiple instances of a given Client

OpenID Connect

- Standard built upon OAuth 2.0 for adding authentication and identity
- ID token is issued in addition to an access token

OAuth 2.1

Work in progress to consolidate most commonly used OAuth 2.0 features

- PKCE obligatory for Authorization Code flow
- Obligatory exact matching of redirect URIs
- Implicit flow is deprecated
- Resource Owner Password Credentials flow deprecated
- Bearer tokens not allowed in query string of URIs
- Refresh tokens must be sender-constrained or one-time use

The End