Securing online services

Application-level access control

Bert Lagaisse, 2022/10/25

Bert.Lagaisse@kuleuven.be

KU LEUVEN DistriNet

T10 OWASP API Security Top 10 - 2019

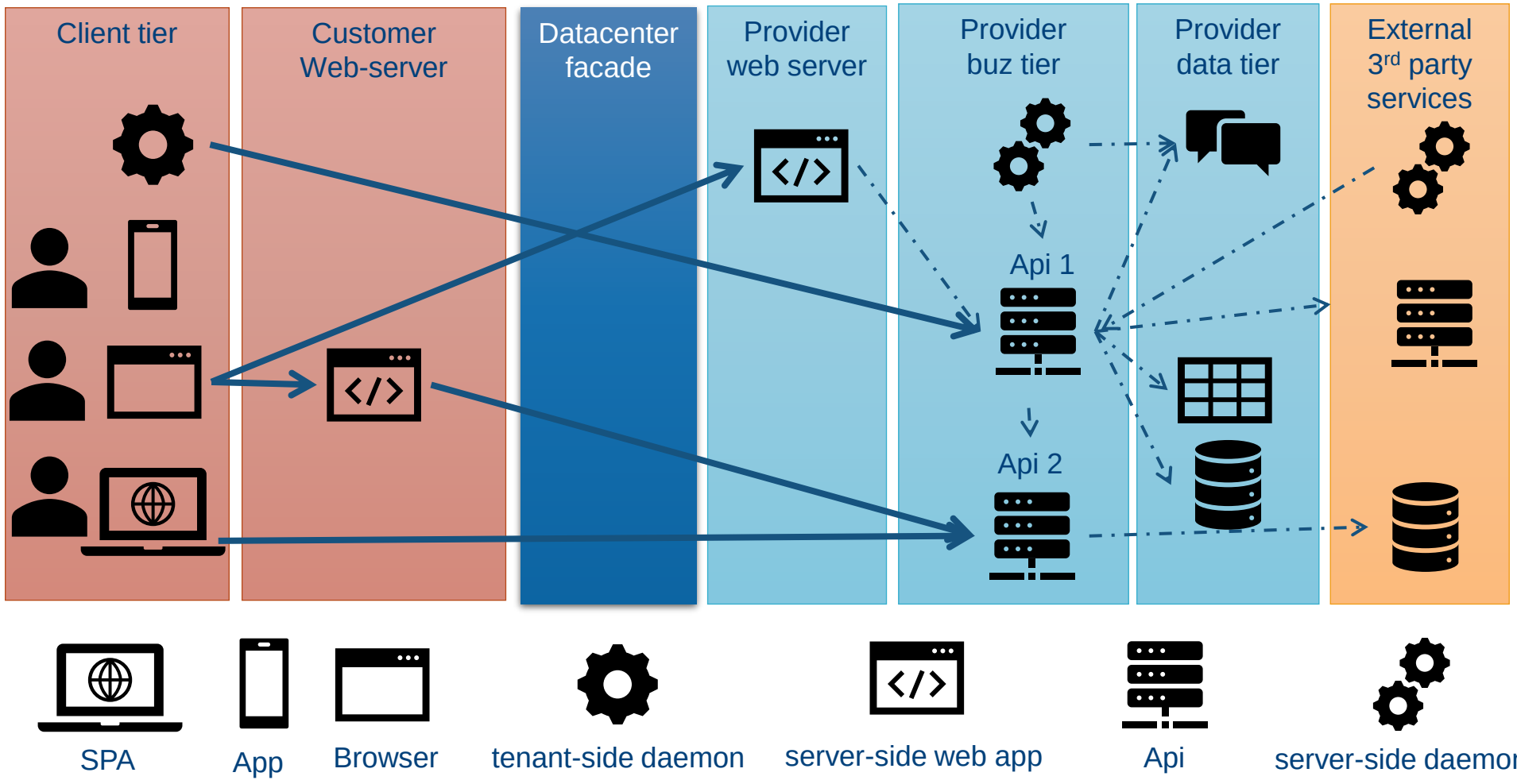
API1:2019 - Broken Object Level Authorization	APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user.
API2:2019 - Broken User Authentication	Authentication mechanisms are often implemented incorrectly, allowing attackers to compromise authentication tokens or to exploit implementation flaws to assume other user's identities temporarily or permanently. Compromising system's ability to identify the client/user, compromises API security overall.
API3:2019 - Excessive Data Exposure	Looking forward to generic implementations, developers tend to expose all object properties without considering their individual sensitivity, relying on clients to perform the data filtering before displaying it to the user.
API4:2019 - Lack of Resources & Rate Limiting	Quite often, APIs do not impose any restrictions on the size or number of resources that can be requested by the client/user. Not only can this impact the API server performance, leading to Denial of Service (DoS), but also leaves the door open to authentication flaws such as brute force.
API5:2019 - Broken Function Level Authorization	Complex access control policies with different hierarchies, groups, and roles, and an unclear separation between administrative and regular functions, tend to lead to authorization flaws. By exploiting these issues, attackers gain access to other users' resources and/or administrative functions.

Broken
Application-level
access control
(Authentication,
Authorization)
=
Root of
Many problems
In API Security

Complex online services

Example architecture

KU LEUVEN DistriNet


Client tier
Customer Web-server
Datacenter facade
Provider web server
Provider buz tier
Provider data tier
External 3rd party services
Api 1
Api 2
SPA
App
Browser
tenant-side daemon
server-side web app
Api
server-side daemon

Architectural decomposition: types of subsystems

Client-side

- Tenant-side Daemon
  - In tenant's datacenter / administration domain
  - Often used for batch operations uploads/downloads
- Mobile app with browser
- Mobile app without browser
- Desktop app
- Stateless Browser
- SPA in Browser

Server-side

- Provider-side Daemon
  - In your administration domain/data center
  - Often processes tasks in a queue
- External 3rd party Daemon
  - E.g. External job/loadtester calls your urls
- Customer web app
- Provider web app
- Façade API
- Downstream API (Provider)
- Downstream external API (3rd party)
- Storage API's (e.g. Amazon s3)

Authentication and Authorization

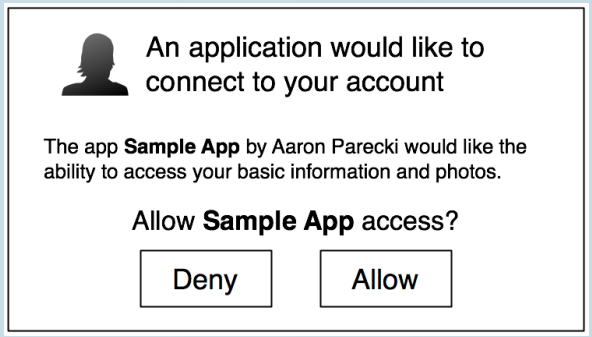
Authentication vs authorization

Authentication (AuthN)

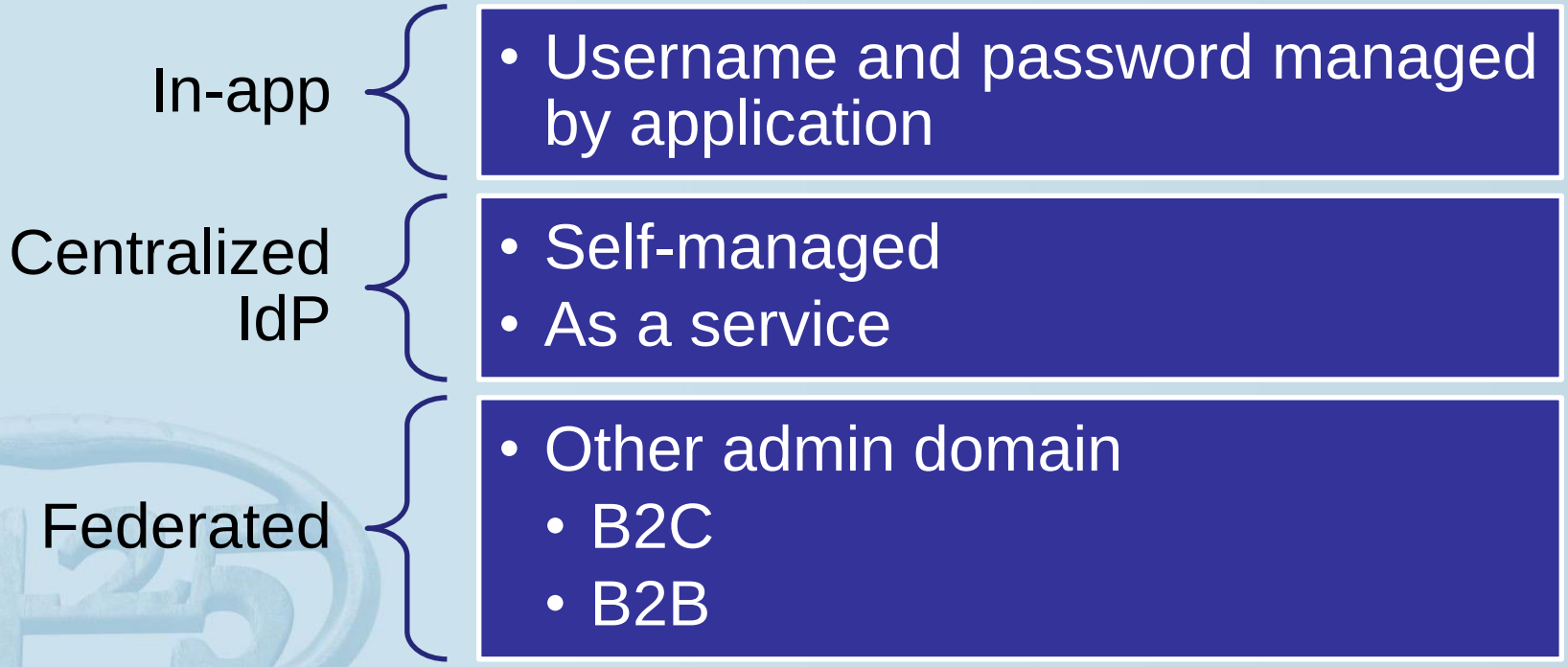
- The process of proving that someone or something is who she/he/it claims to be.
- E.g. using password or biometrics
- E.g using OpenID Connect protocol, or using SAML

Authorization (AuthZ)

- Grant an authenticated entity(someone or something) permission to do something
- E.g. “managers can edit”
- E.g. using the OAuth 2.0 protocol



Identity management provider



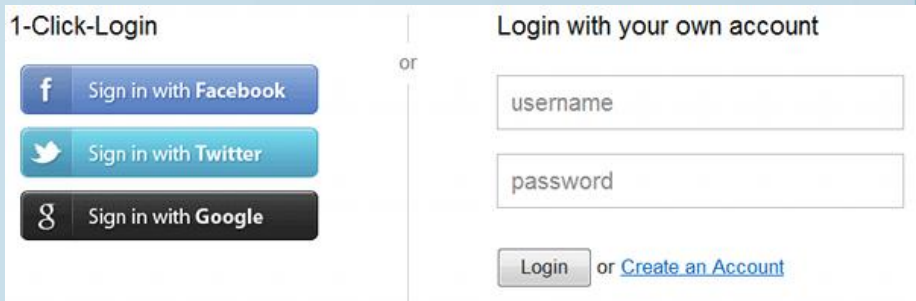
In-app vs Externalized vs Federated authentication

Centralized identity provider

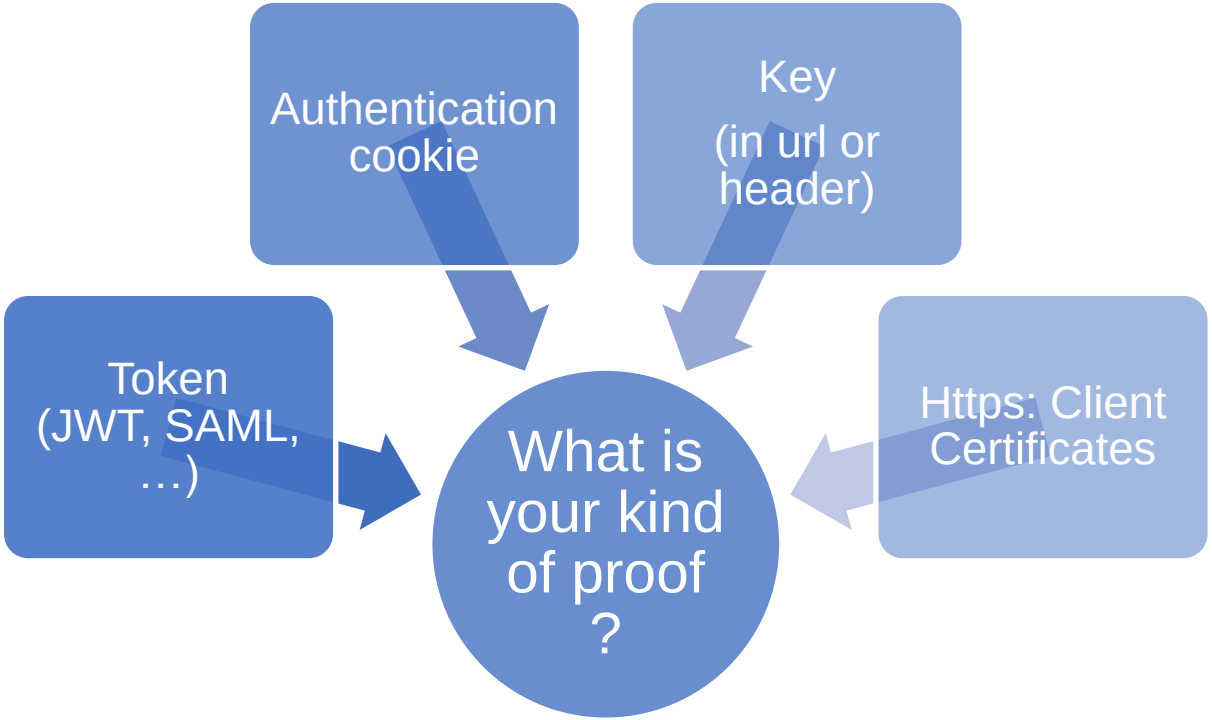
- managing username and password per app ?
  - high administrative burden
  - **Externalize** from your application
- Delegate to central service in your administration boundary
  - Self-managed in your own data center (e.g. keycloack)
  - IAM as a service (e.g. azure ad)
- You still manage and control the identities, users and password

Federated authentication

- Users authenticate to IAM system of other administration boundary
  - B2B: Organizational customer or partner IAM system
  - B2C: authentication using Facebook, Google or Microsoft account



Authenticating and authorizing incoming http calls…

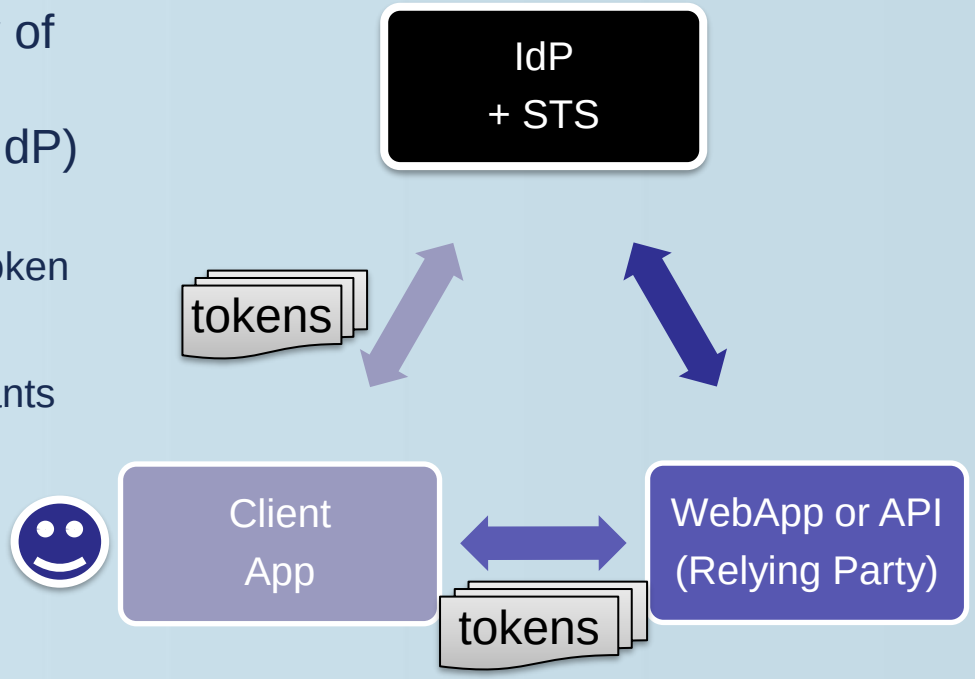


Externalizing identity and access management: Based on security tokens

Security tokens

- **Identity token:** proves identity of a user as verified by the externalized identity provider (IdP)
  - **Security token server** (STS)
  - E.g. SAML or OpenID Connect token
- **Access token:**
  - Contains user and/or app that wants access to a resource
  - Often short-lived TTL
- **Refresh token:**
  - Longer-lived
  - To get a new access token

Parties



Who creates and manages the token ?

In-app token management	• Web app or api creates tokens

Externalized token management	• Externalized IdM provides STS
	  • Self-managed
	  • As a service

Background on token management

In-app token management

- **Web app or api uses a library to verify token and extract info**
  - Verify Issuer
  - Verify Audience (target application)
  - Verify Lifetime
  - …
- *Web app or api creates tokens*
  - Often with application-specific info
  - Web app or api implements and hosts its own STS (security token service)

Externalized token management

- IdM systems often provide STS
- Standard set of token claims
  - UserID, username, groups, roles
  - Customizable with some extra claims
- Self-managed
  - IdentityServer
  - KeyCloak
  - …
- As-a-Service
  - Azure AD
  - Amazon Cognito
  - Auth0
  - …

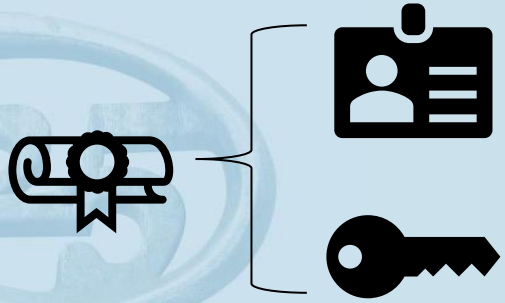
Application vs User identity
Access token vs Identity token

Application identity

- E.g. “The Twitter app TweetDeck is allowed to communicate with the Twitter API”.

User identity

- E.g. Bert Lagaisse is allowed to access the Twitter stream of Pieter Phillipaerts.



Identity token:
proofs identity of Bert Lagaisse and his attributes

Access token: identity and permissions of TweetDeck

Distinguishing between: Application vs User permission ?

User permission

- Assigned permission to a user, typically a human identity
  - Patrick has the role project manager
  - Patrick can read and write data

Application permission

- Assigned to client applications of an API
  - The daemon application of tenant234 can only write data to datastore 234
  - The public client app to view bills can only read information when using the API

Public vs Private Client Application

Public

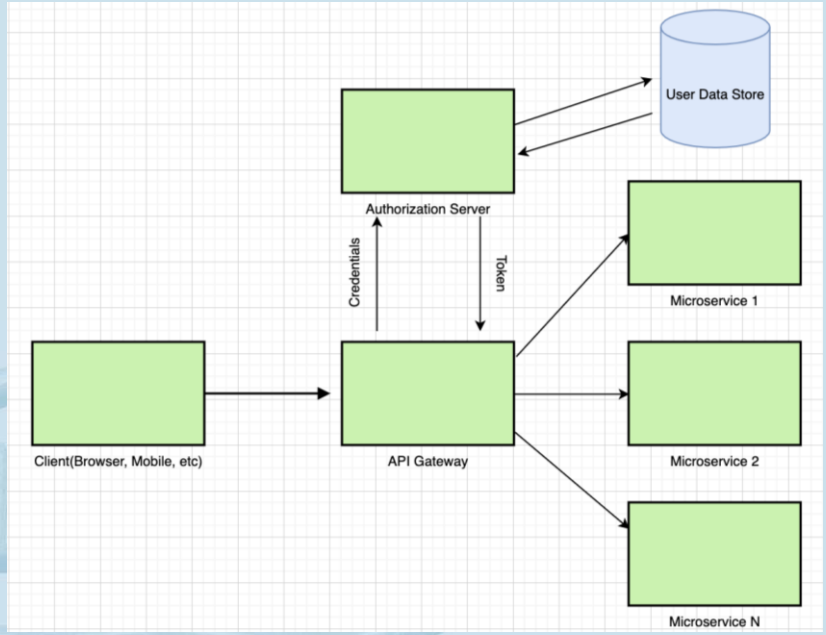
- Consuming application is on a public device
  - E.g. TweetDeck
- Can not keep a secret
  - Application password visible
  - In browser
  - In mobile app store

Private

- Consuming application of the API is deployed securely and can keep a secret.
  - E.g. web app on a web server
  - Provider-side daemon

Where to store the tokens?
Client-side vs server-side token storage

E.g Spring cloud gateway

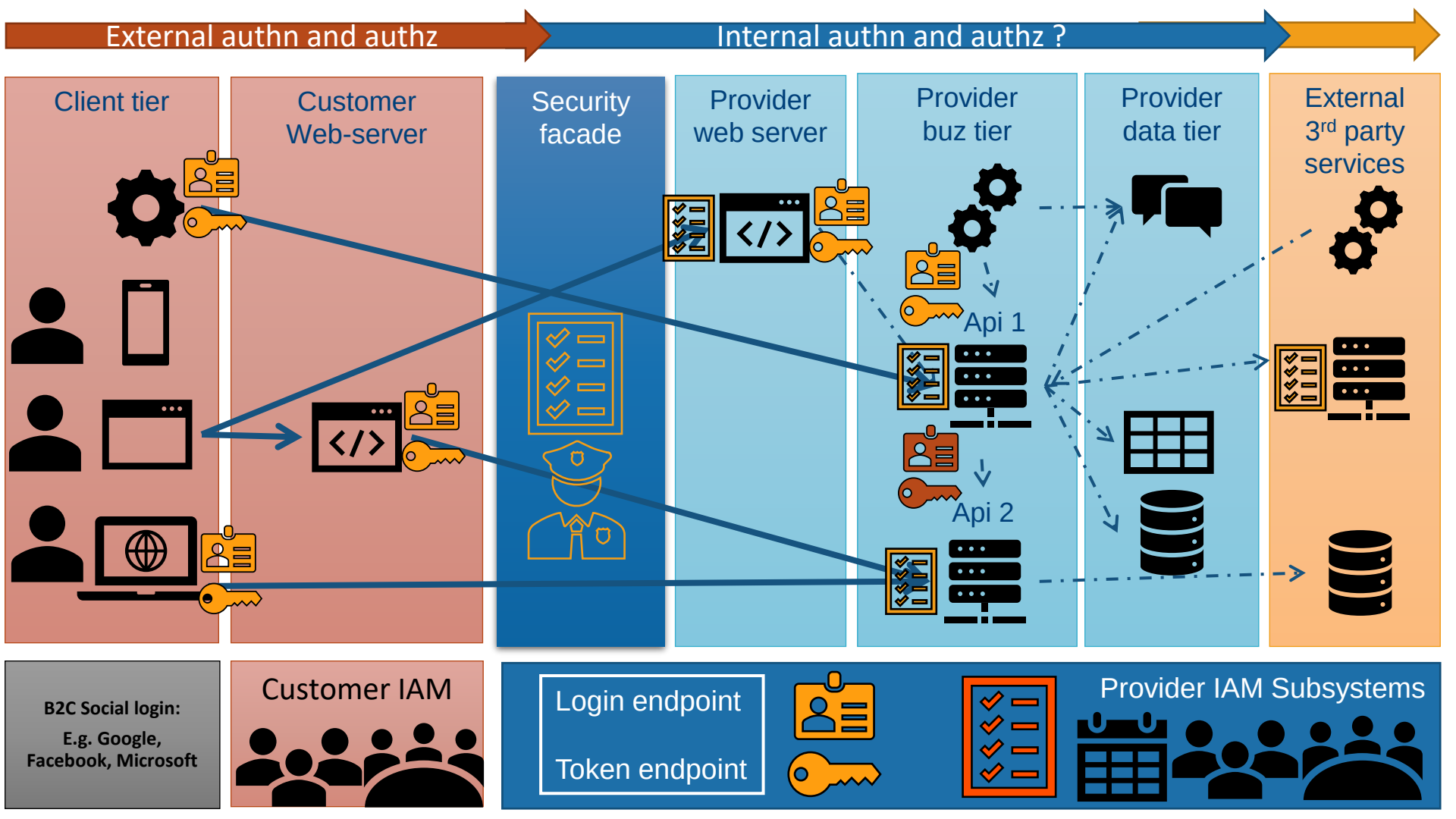


Client-side API consumers

- Single page application in Browser
  - Angular, React, Vue, Blazor, …
- Mobile app
- Lots of security considerations

Server-side API consumers

- Security token is never exposed to a browser
- Stored in
  - a user's session on a webserver
  - Server side persistent storage (!!!)
- Browser only gets a cookie


External authn and authz
Internal authn and authz ?
Client tier
Customer Web-server
Security facade
Provider web server
Provider buz tier
Provider data tier
External 3rd party services
Api 1
Api 2
B2C Social login:
E.g. Google, Facebook, Microsoft
Customer IAM
Login endpoint
Token endpoint
Provider IAM Subsystems

Token acquisition from several types of apps:

consumer tier

SPA

server-side web app

Mobile App

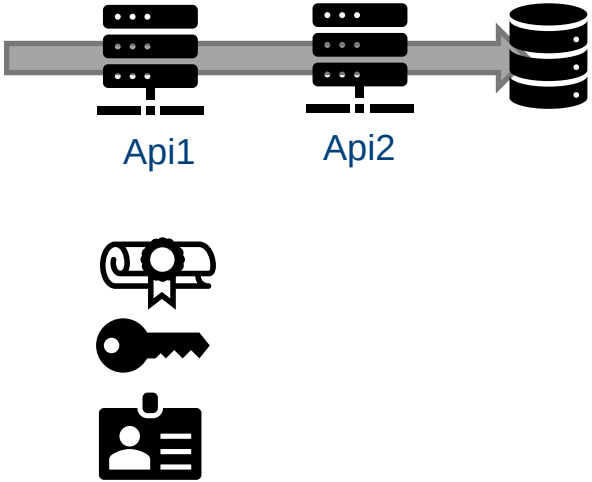
Browserless app

Desktop app

IOT device

tenant-side daemon

› With a signed-in-user



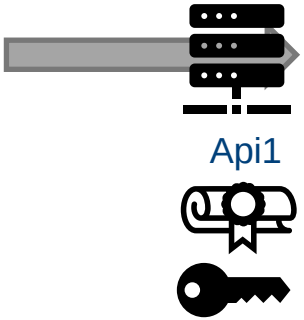
consumer tier

› As application

server-side daemon

tenant-side daemon

daemon web app

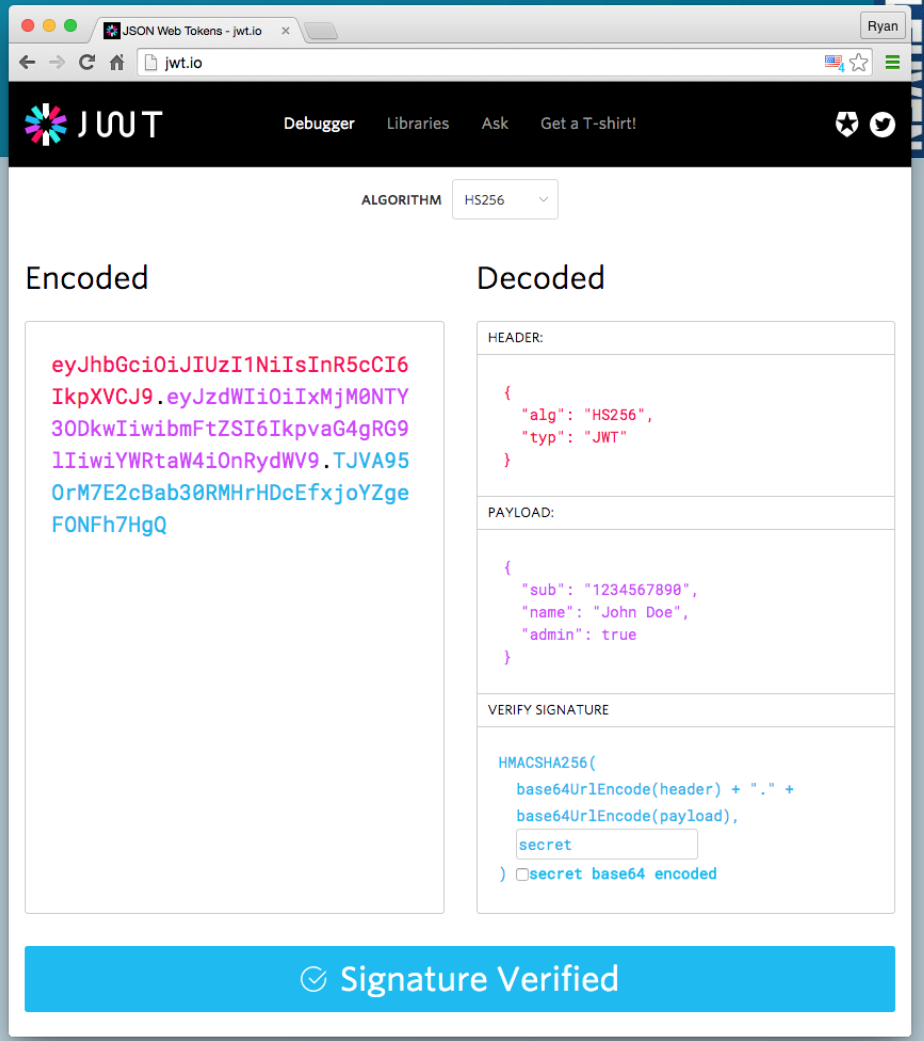


Tokens and Token acquisition: technology and protocols

The low-level carrier JWT Token Format

Structure:

- 3 parts
  - Header: token type and signing algo
  - Payload: claims (user statements)
    - Iss: issuer (e.g. facebook)
    - Exp: expiration time
    - Sub: subject
    - …
  - Signature
- Format: xxxxx.yyyyy.zzzzz



Supporting client variation with OAuth 2.0 and OpenID connect

OAuth 2.0:

Authorization protocol

- Variation per type of client: basics
  - Implicit grant flow
  - Authorization code grant (+PKCE)
  - Resource owner password credentials grant (ROPC)
  - Client credentials grant
- Flows beyond basics/standards:
  - Hybrid flow
  - On-behalf-of flow
    - Token-exchange flow
  - Device code flow

OpenID Connect:

Authentication protocol

- Securely login a user to an application
- Interactive authentication
- Without exposing password to app
- Built on OAuth 2.0
- Extends OAuth 2.0 for use as an authentication protocol

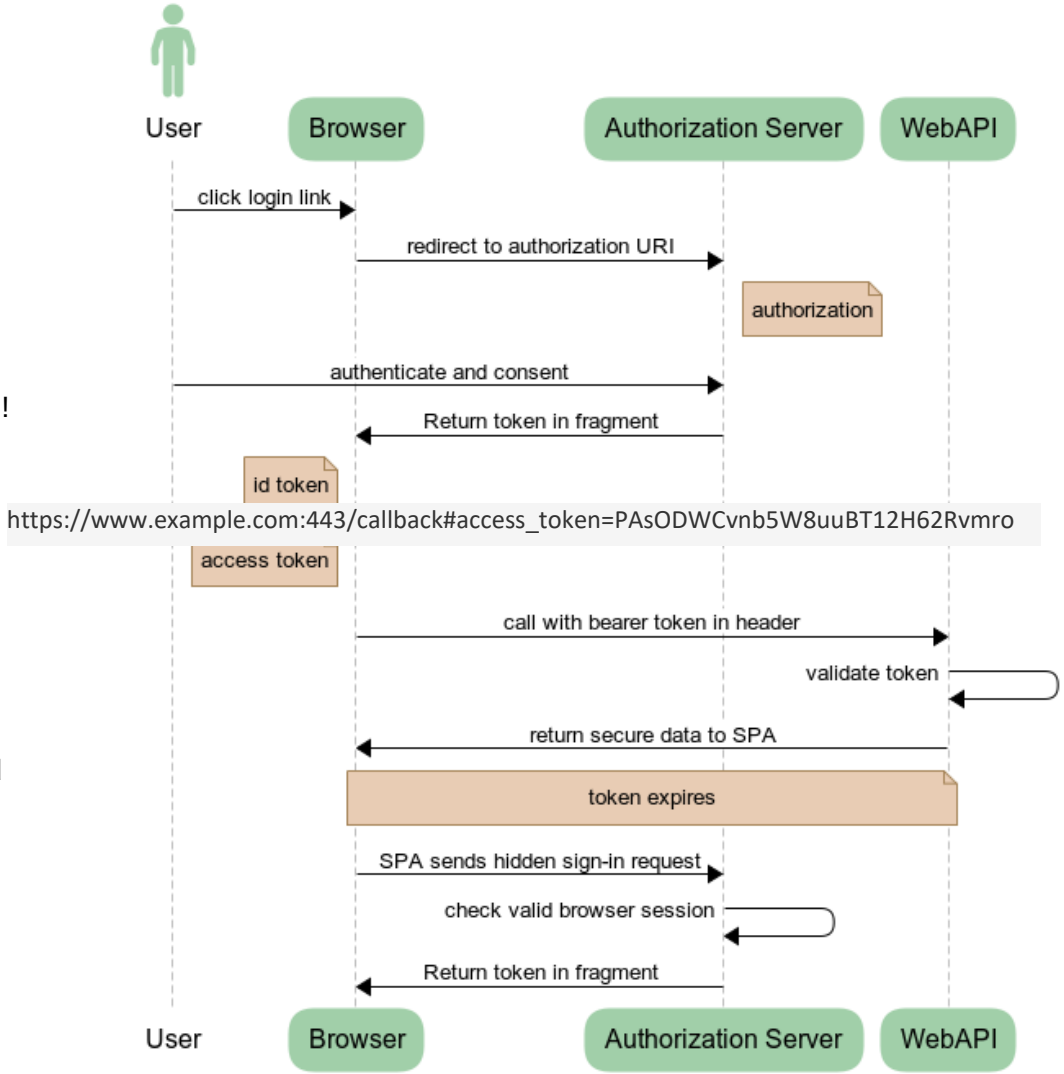
Other protocols and token formats: Oauth, OpenID, SAML, WS-Fed ?

PROTOCOL	OAuth	OpenID	SAML	WS-Fed
Introduced	2010	2005	2001	2003
Current version	2.0, released in 2012	OpenID Connect 1.0, released in 2014	2.0, released in 2005	1.2, released in 2009
Proprietary or Open	Open	Open	Open	Proprietary
Purpose	Enables delegated authorization for internet resources	Provides an authentication layer over OAuth2.0	Allows 2 web entities to exchange authentication and authorization data	The same as SAML, but not as widely used
When to Use	To provide temporary resource access to a 3rd-party application on a legitimate user's behalf	To authenticate users to your web or mobile app without requiring them to create an account	To allow a user or corporate partner to use single sign-on to access a web service	The SAML use case also applies. Most MSFT web applications like SharePoint & Azure have native support for WS-Fed
Supported protocols	HTTP	XRDS, HTTP	XML, HTTP, SOAP, and any protocols that can transport XML	XML, HTTP, SOAP, and any protocols that can transport XML

Single page app

Token acquisition variation

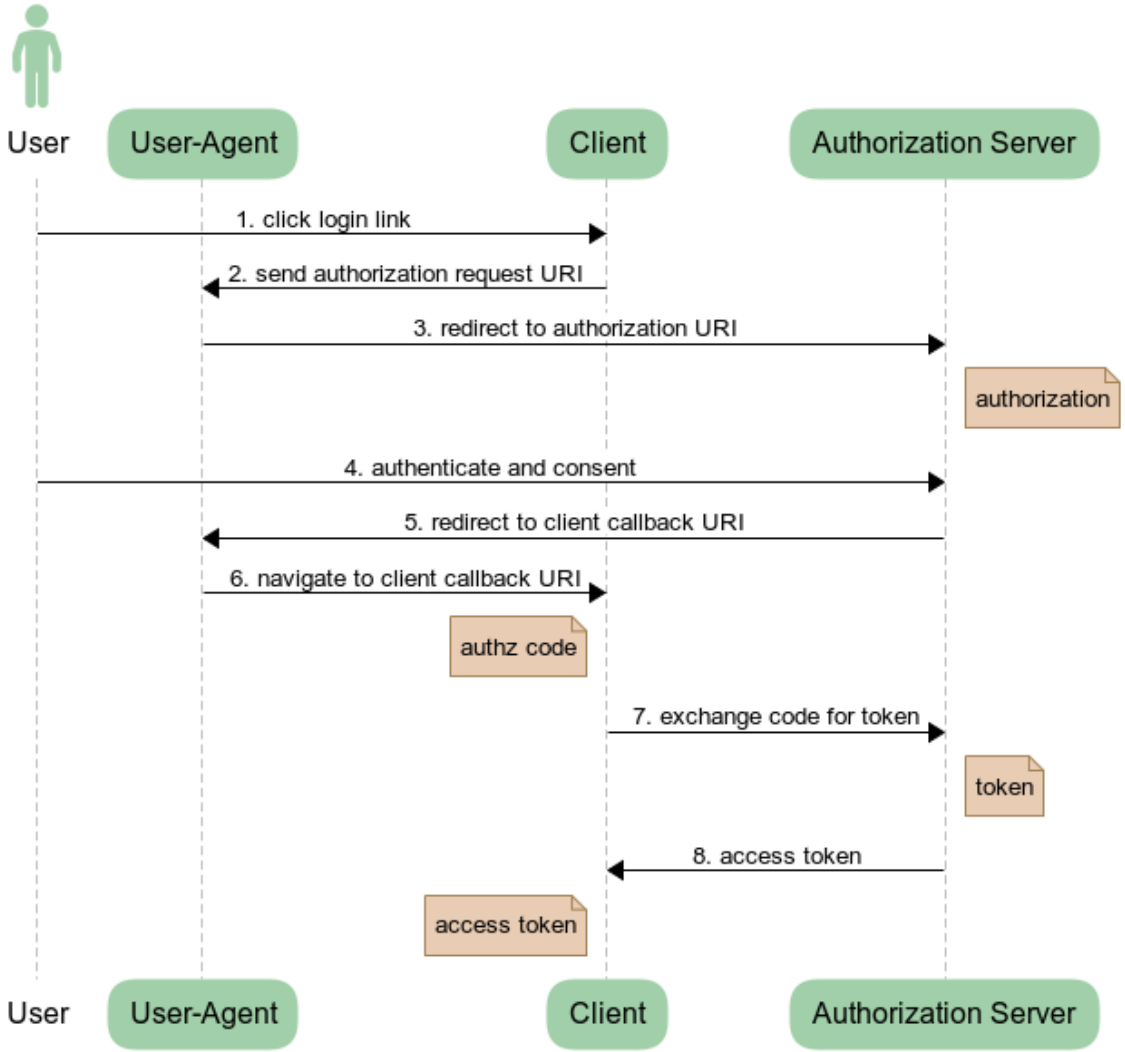
- Implicit flow
  - Get ID token and access token directly from authorization endpoint
  - Redirects to redirect_uri with token in url fragment !
  - No refresh token
- Authorization code flow
  - First get a “short” authorization code
  - Get access token with this code
  - Refresh token to renew access token
- Hybrid flow
  - ID token + authz code from authz endpoint instead of token endpoint
  - Access token from token endpoint
- PKCE ?



Authz code grant flow

e.g. client app with browser

- url fragment might be too limited for
  - All claims
  - All access tokens
  - Protecting the token
- Authorization code flow
  - First get a “short” authorization code
  - Get access token with this code
  - Refresh token to renew access token

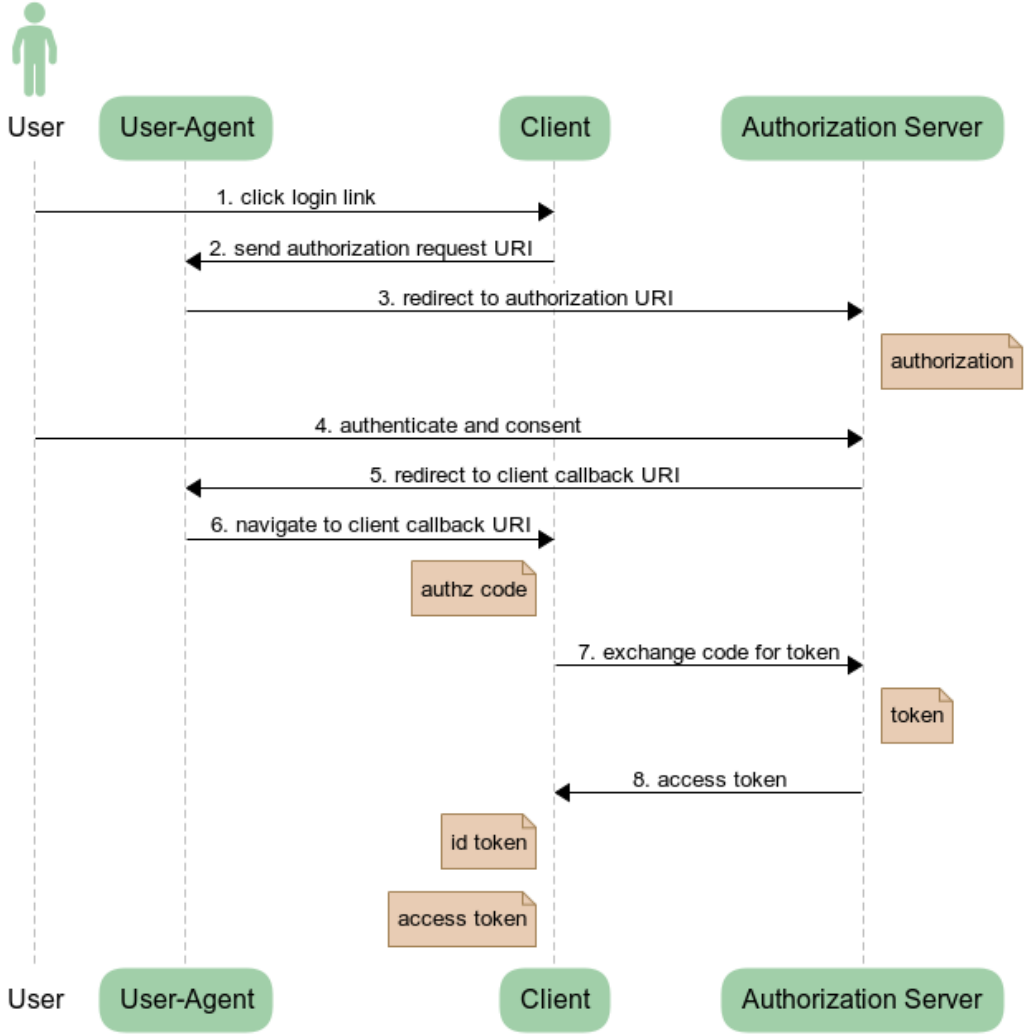


Auth code grant flow (2)

Id token also via code

- Both access token
  and id token

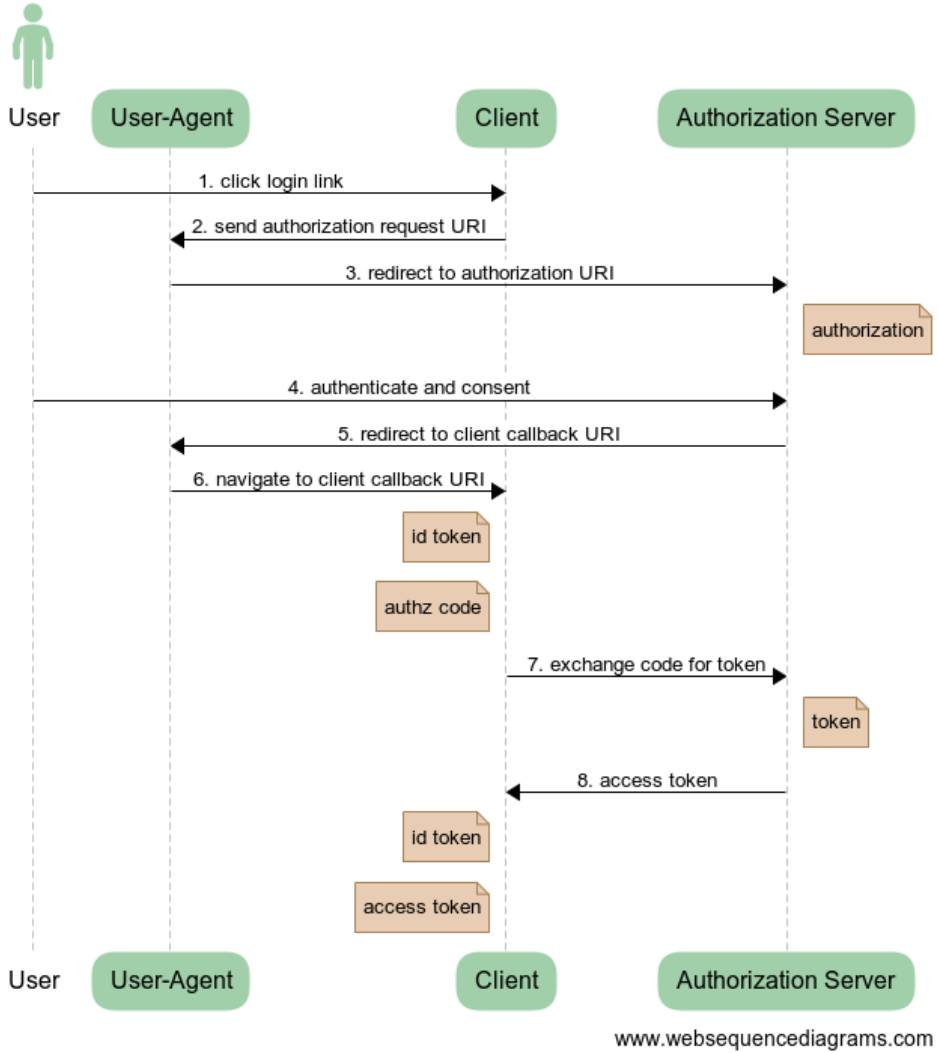
  are obtained via authz code



Hybrid flow

- ID token + authz code from authz endpoint
  - Access token from token endpoint

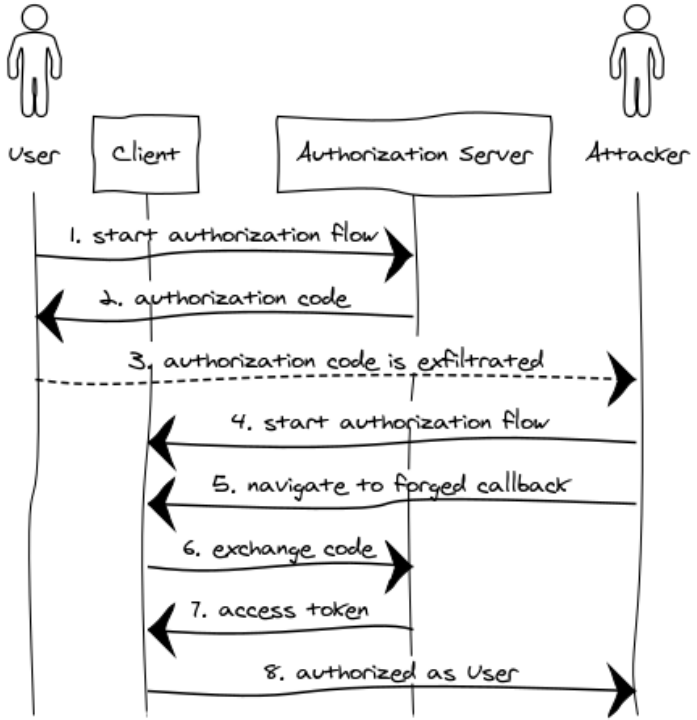
.



Security considerations: implicit, authz code, hybrid

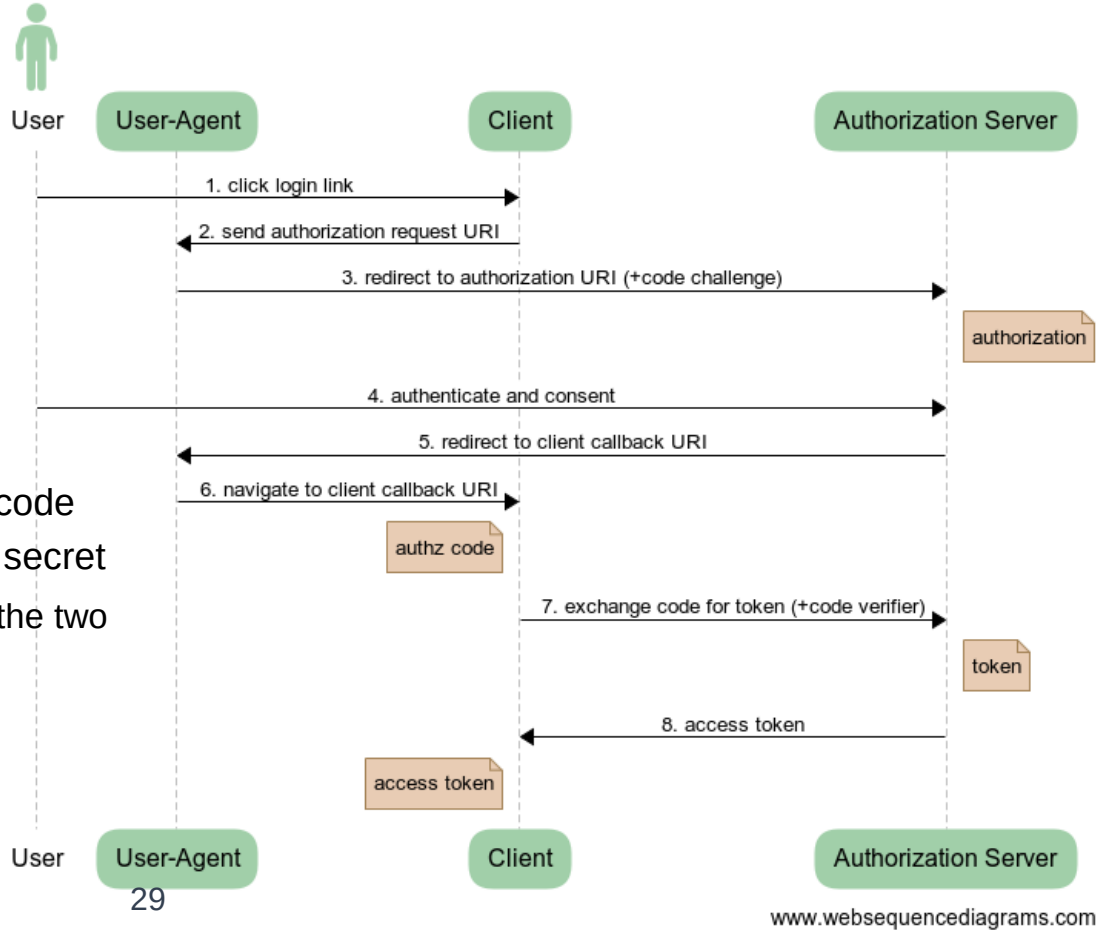
PUBLIC client-side consumers: SPA, Desktop and Mobile apps

- Avoid Implicit flow grant
  - Easier, but can't offer token protection
  - Other apps can subscribe for redirect URI and steal the access token
- Client receives the authorization code from the redirect URI.
  - App parses redirect URI in browser
  - protect against other apps on device
- use PKCE: Proof Key for Code Exchange



Proof Key for Code Exchange (PKCE)

- Bind an authorization code to a client's session
  - Client generates a random secret per authorization request
  - Client sends the hashed secret in the authorization request
  - When it exchanges the authorization code for an access token, it also sends the secret
    - The server can hash and compare the two hashes



www.websequencediagrams.com

Proof Key for Code Exchange (PKCE)

REQUEST

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
    &code_challenge=rLGaLy…5Z5Dc&code_challenge_method=S256 HTTP/1.1
Host: server.example.com
```

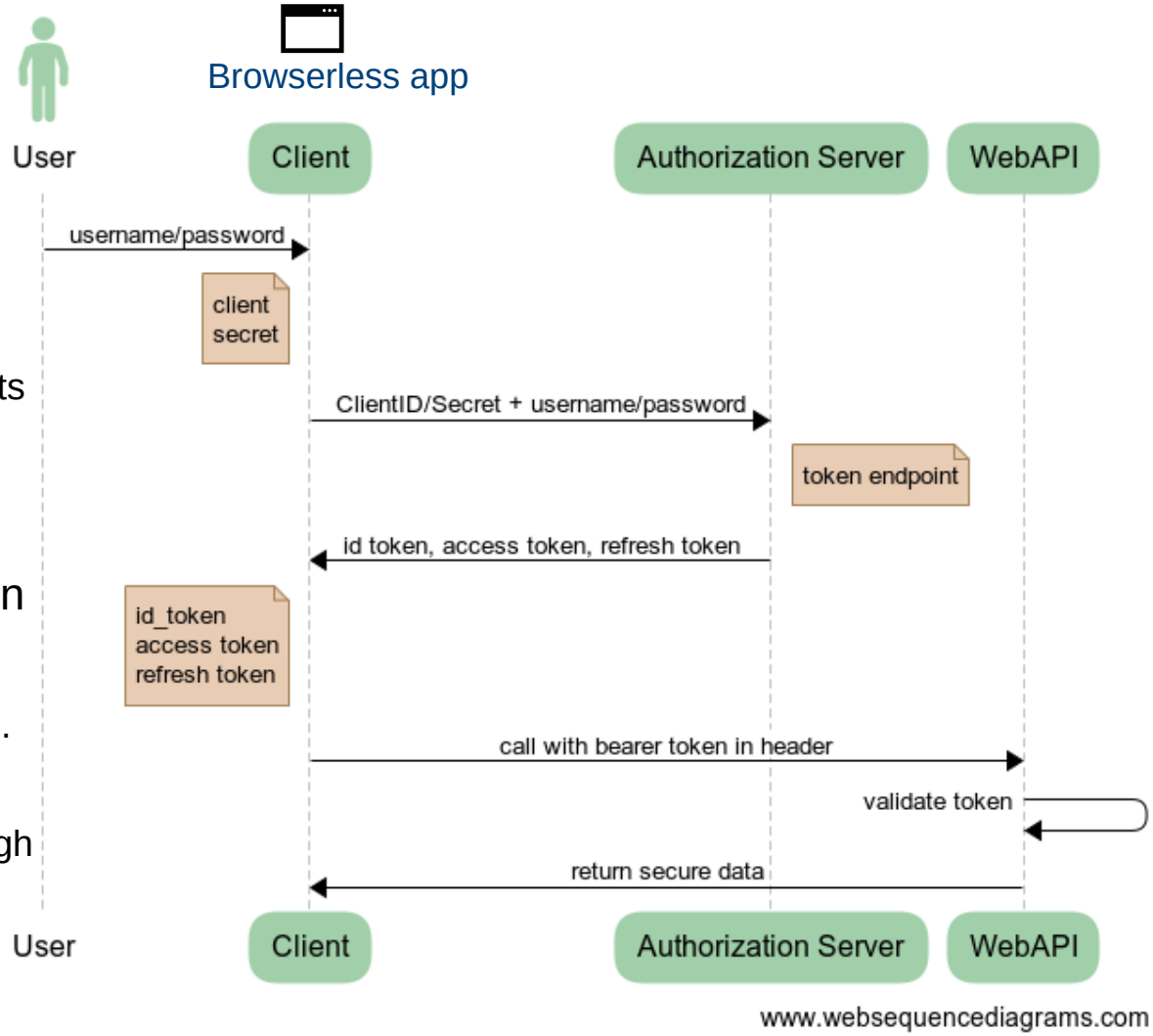
REQUEST

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&code_verifier=8WBGM8cbVT…bRzqts370
```

Resource owner password credentials

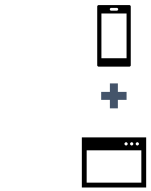
- ROPC: only for trusted clients
  - Client sees password
  - Often used for browserless clients
- Alternatives are available and recommended
- Requires high-degree of trust in client application
  - Private, trusted client application.
    - Not public one !
  - Password goes in plain text trough the client application



Device code flow

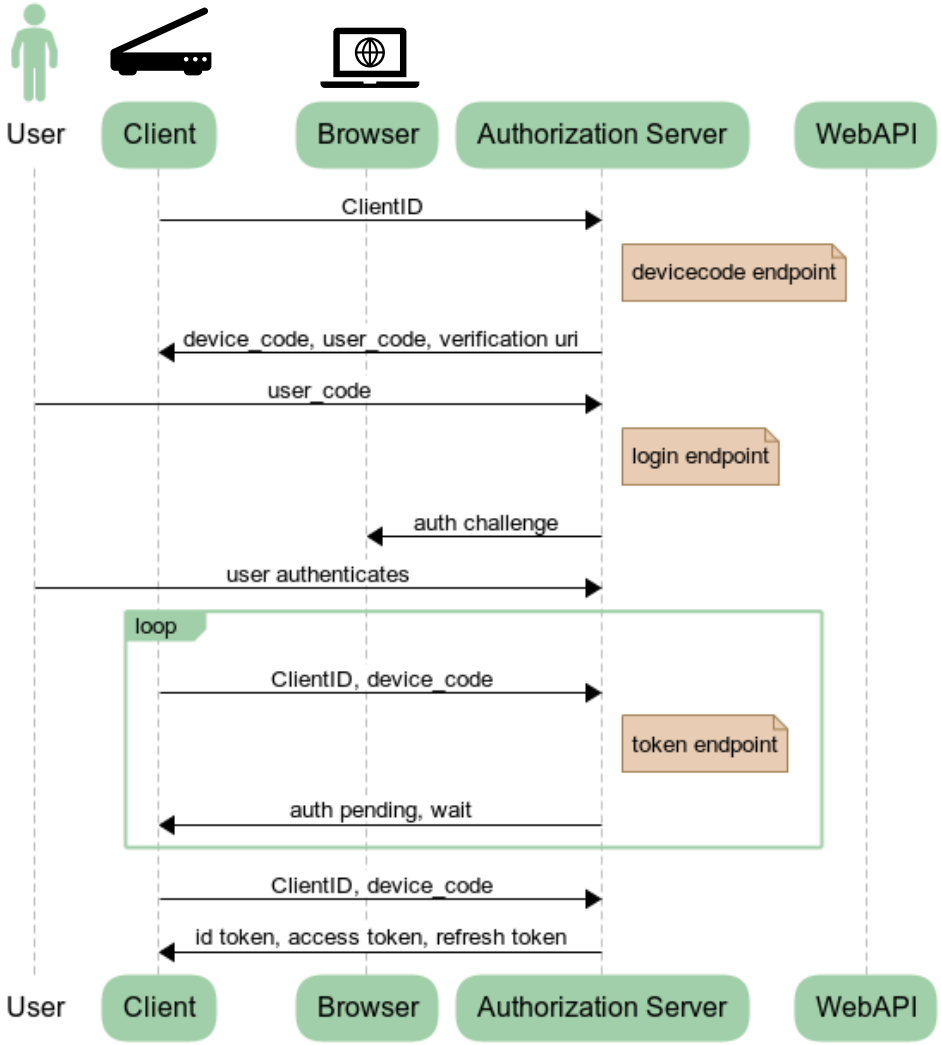
For IOT devices

- For clients
  - Browserless systems
  - Limited input device (IOT device)
- Involves 2 devices
  - User device with browser
  - Client: IOT device
- Better than ROPC



Browserless app

IOT device

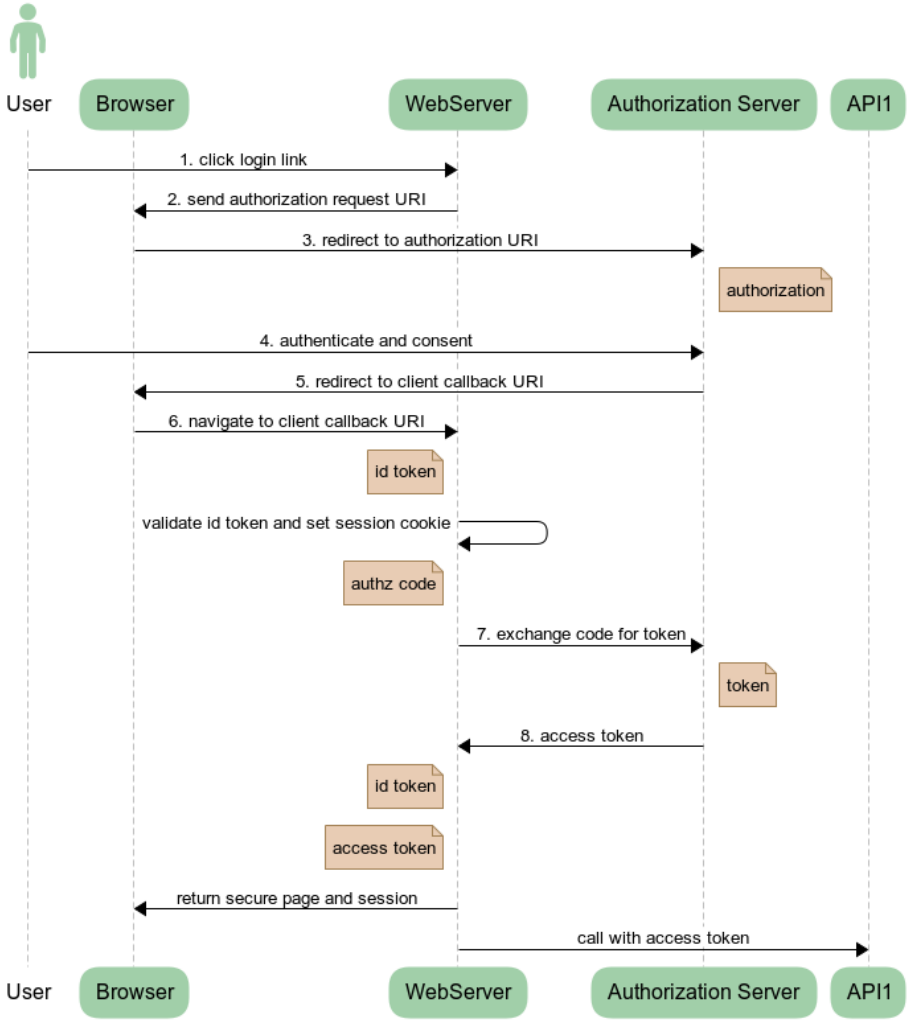


Server-side token acquisition

Server-side Web app

authz code / hybrid

- To authenticate a user
  - Protecting the web app
  - Using an externalized IdP
    - Redirect to IdP to enter credential
    - OpenID connect
  - Web app validates received id token
- To call web api on behalf of user
  - Web apps that call web APIs are confidential client applications
  - Application secret or certificate (NO PKCE NEEDED)
  - Web app authorizes itself AND acts for the user
- Also used in web application firewalls and micro-service platforms
  - WAF stores the access tokens (the real meat)
  - Browser gets a cookie



www.websequencediagrams.com

Machine 2 machine, without user: Oauth 2.0 client credentials grant flow

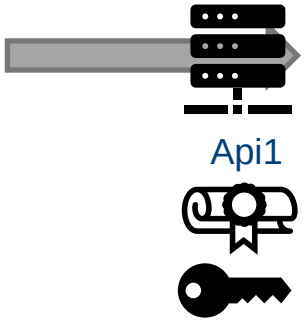
interactions in name of application accounts

consumer tier

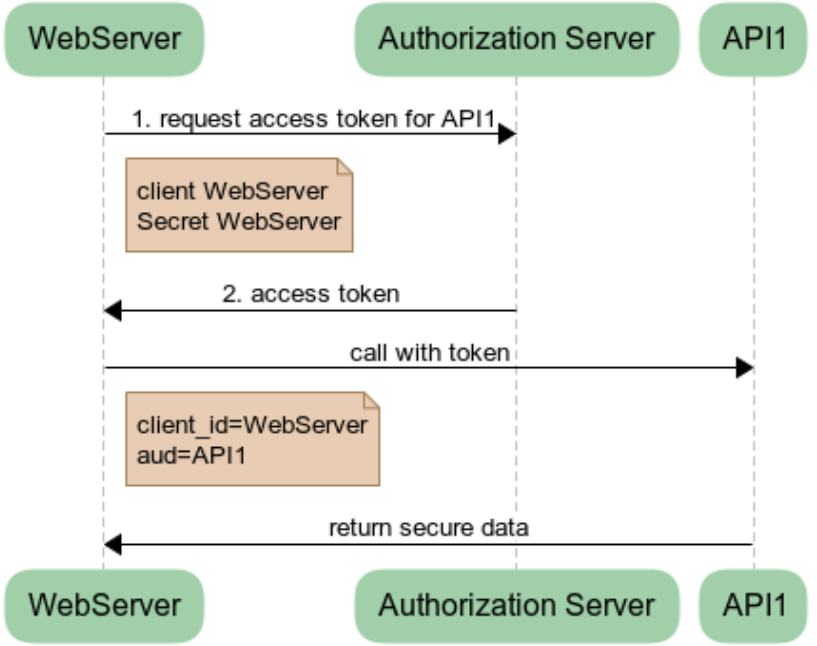
server-side daemon

tenant-side daemon

daemon web app

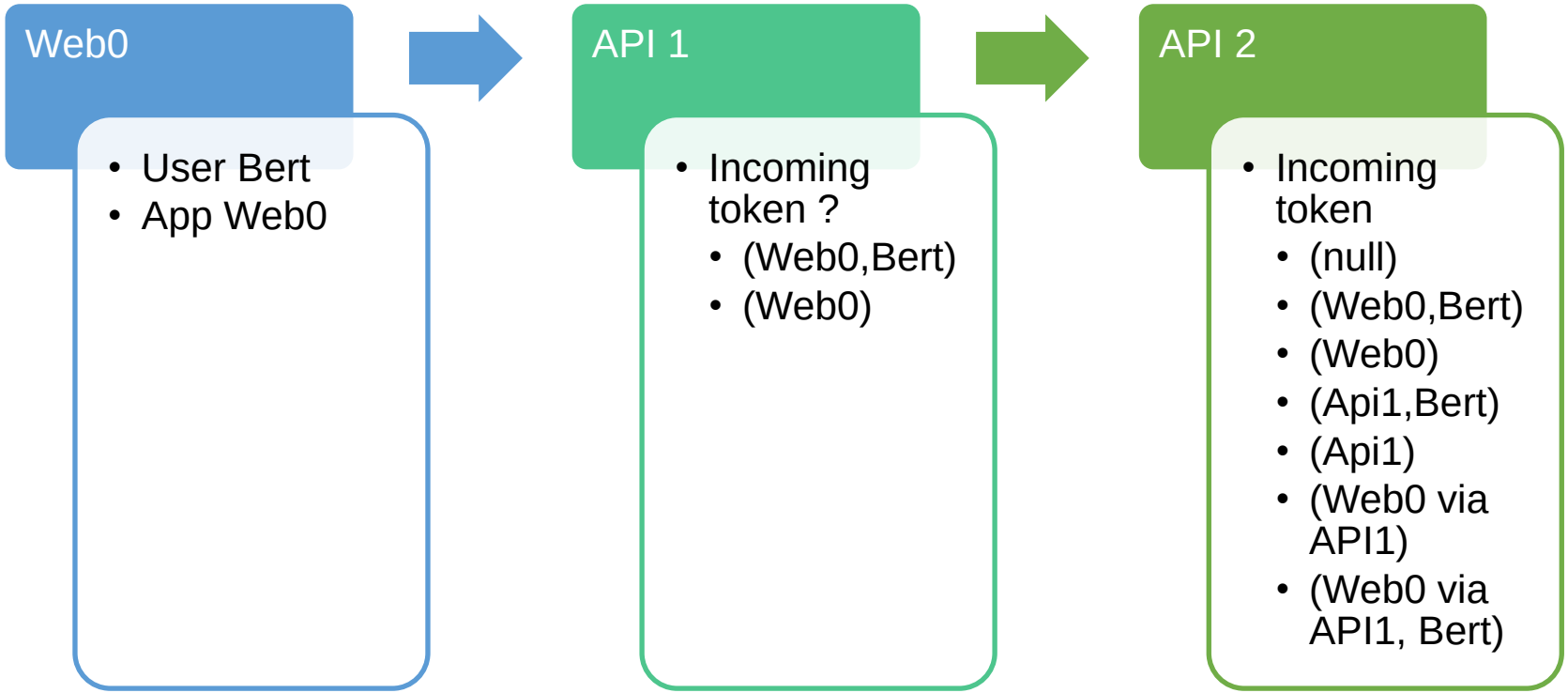


› *“two-legged Oauth”*



Calling downstream API’s
Delegation and impersonation

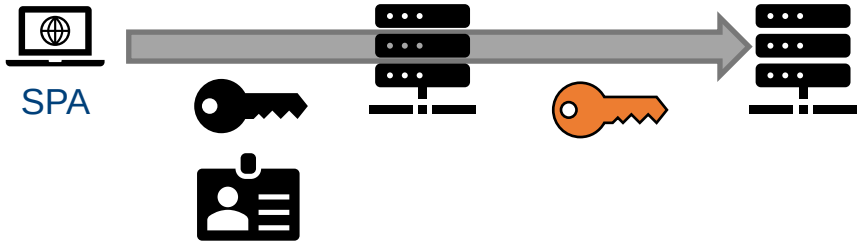
Calling a downstream API
Conceptual identity flow variations with proof

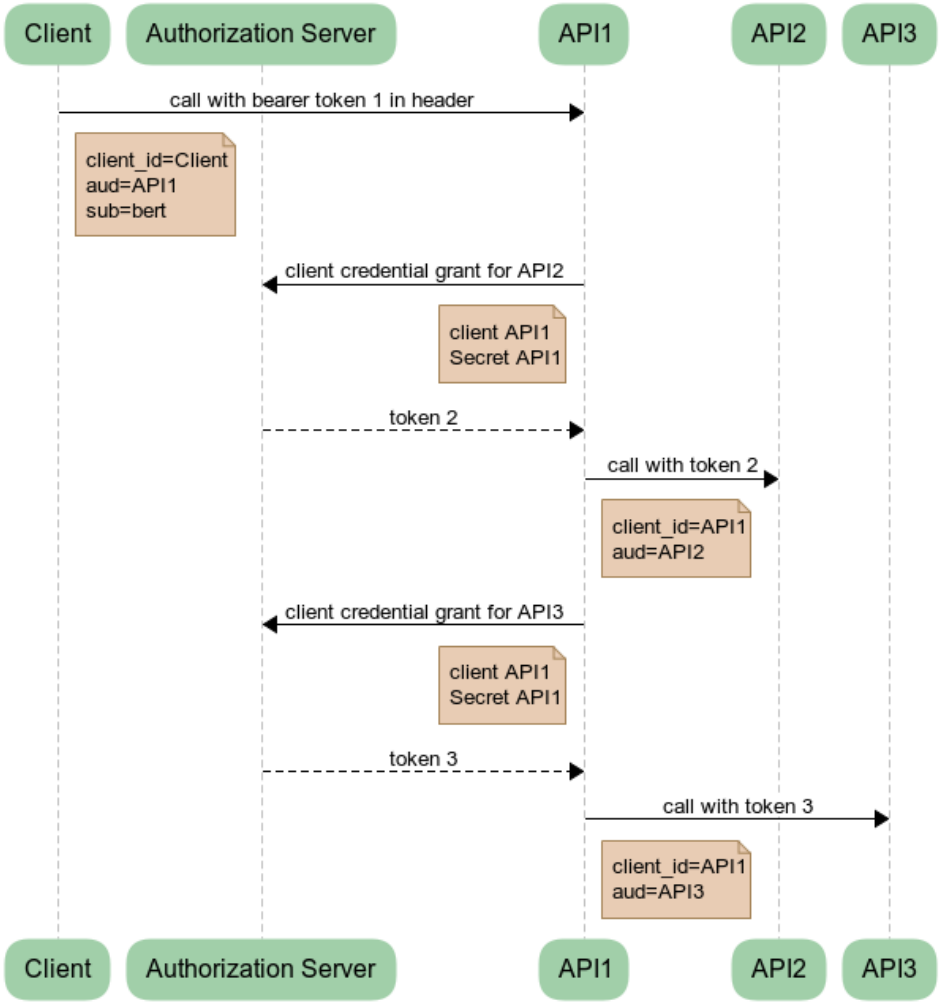


Calling a downstream API

Just call as app without user id

- Get new token from authz server
  - Specific for API1 as client
  - Specific for API2 as audience(target)
- Using client credentials grant flow
- Using client id and secret of the middle tier API1.
  - Acquire token 2 for API2
  - Acquire token 3 for API3
- Token can be reused within TTL
  - Then use refresh token.



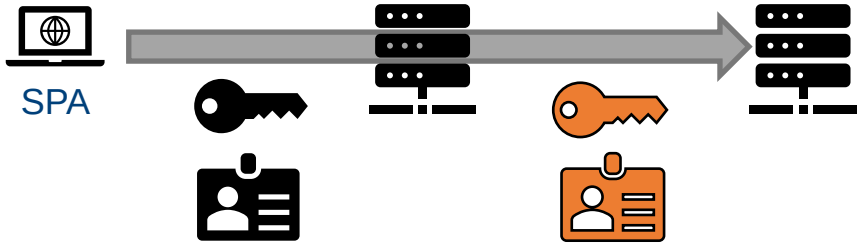


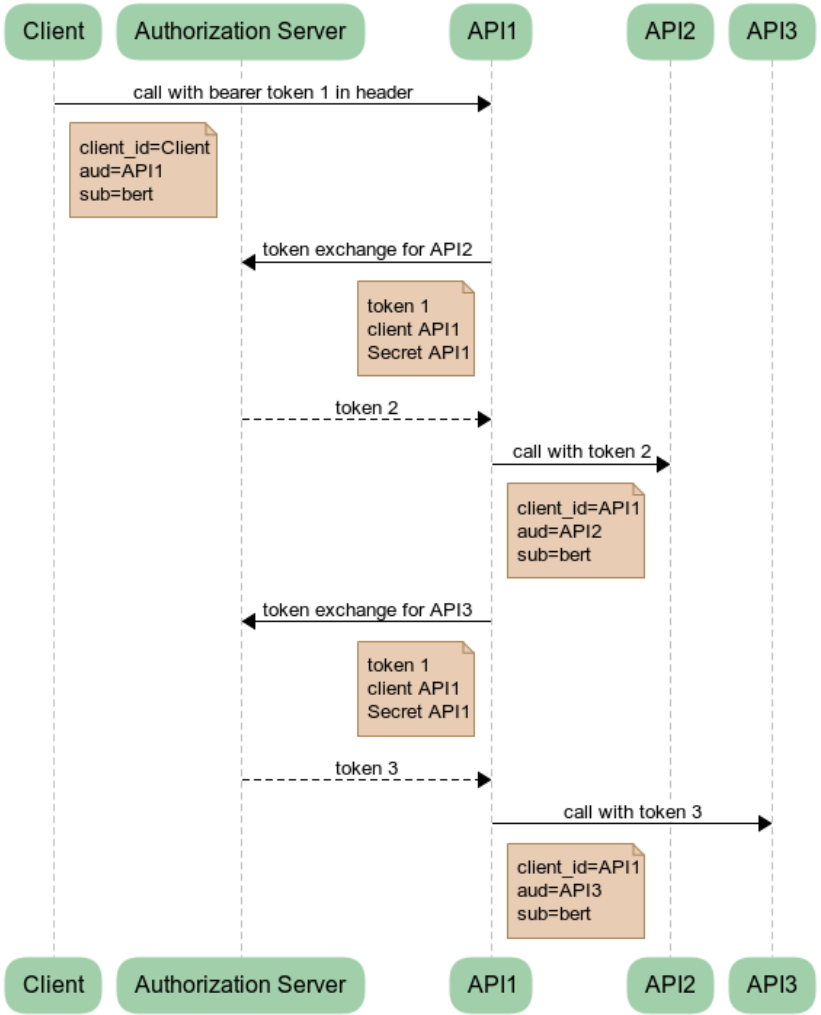
www.websequencediagrams.com

Calling a downstream API

Token exchange flow

- Get new token from authz server
  - Specific for API1 as client
  - Specific for API2 as audience(target)
  - Containing user id and user claims (id token)
- Called API 1 becomes calling client to API2
  - Uses received token to exchange for a new token containing
    - Subject (user identity)
    - New Calling Client API1
    - New Called Audience API2





Calling an external API out of your admin domain

Retrieve key or credential from a secrets store

- E.g. AWS secrets manager
  - Requires AWS credentials
- E.g. Azure Key vault
  - REST API
  - Protected by Azure AD
    - Authn and access policies
  - Additionally and optionally
    - Protected by firewall (IP)

Which flow supports which token ?

Access token and/or id token

Flow/Grant type	Access token	ID token
Implicit	V	V
Authz code	V	V
Authz code + PKCE	V	V
Hybrid	V	V
ROPC	V	V
Token exchange	V	V
Device code	V	V
Client credentials	V	X

Which flow for which client type ?

Client (Consumer)	Access token	Id token
Server-side web app	Authz code flow	
SPA	Authz code + PKCE	
Native (mobile, desktop)	Authz code + PKCE	
Fully trusted client	(ROPC)	
Daemon	Client credentials (shared) token exchange	(ROPC) (shared) token exchange
API	Client credentials, token exchange	Token exchange
IOT device	Device code	

Flow/Grant overview

And their implementations in (some) technologies and managed services

Technology	KeyCloak	IdentityServer	AzureAD	Cognito	Auth0	Okta
implicit	V	V	V	V	V	V
Authz code	V	V	V	V	V	V
Authz code+PKCE	V	V	V	V	V	V
Hybrid flow	V	V	V		V	
Client Credentials	V	V	V	V	V	V
Token Exchange	V	V	V			
ROPC	V	V	V		V	V
Device code		V	V		V	