Pragmatic Web Security

Security training for developers

Introduction to OAuth 2.0 and OpenID Connect

Philippe De Ryck

@PhilippeDeRyck – philippe@PragmaticWebSecurity.com


LIGHT ST.
STOPPING ANY TIME
VALET
PARKING
VALET
4DJ2760

buffer

Learn About Buffer for Business | Help | My Account

Accounts | Content | Ana

PhilippeDeRyck
Twitter

Philippe De ...
LinkedIn

Facebook
Connect it now!

Instagram
Connect it now!

Connect More
Profiles and Pages

Posts

All Recent Posts
Your latest posts are looking good,

Recent | Most Popular | Least

Today

TOP TWEET
The slides for my talk on #angula
#SanFrancisco community are o

01:25 (CET) via Web

2 Retweets | 5 Likes

Philippe De Ryck
@PhilippeDeRyck

- Account
- Privacy and safety
- Password
- Cards and shipping
- Order history
- Mobile
- Email notifications
- Notifications
- Web notifications
- Find friends
- Muted accounts
- Muted words
- Blocked accounts
- **Apps**
- Widgets
- Your Twitter data

Applications

These are the apps that can access your Twitter account. Learn more.

Facebook Connect
Post Tweets to your Facebook profile or page.
Connect to Facebook

Having trouble? Learn more.

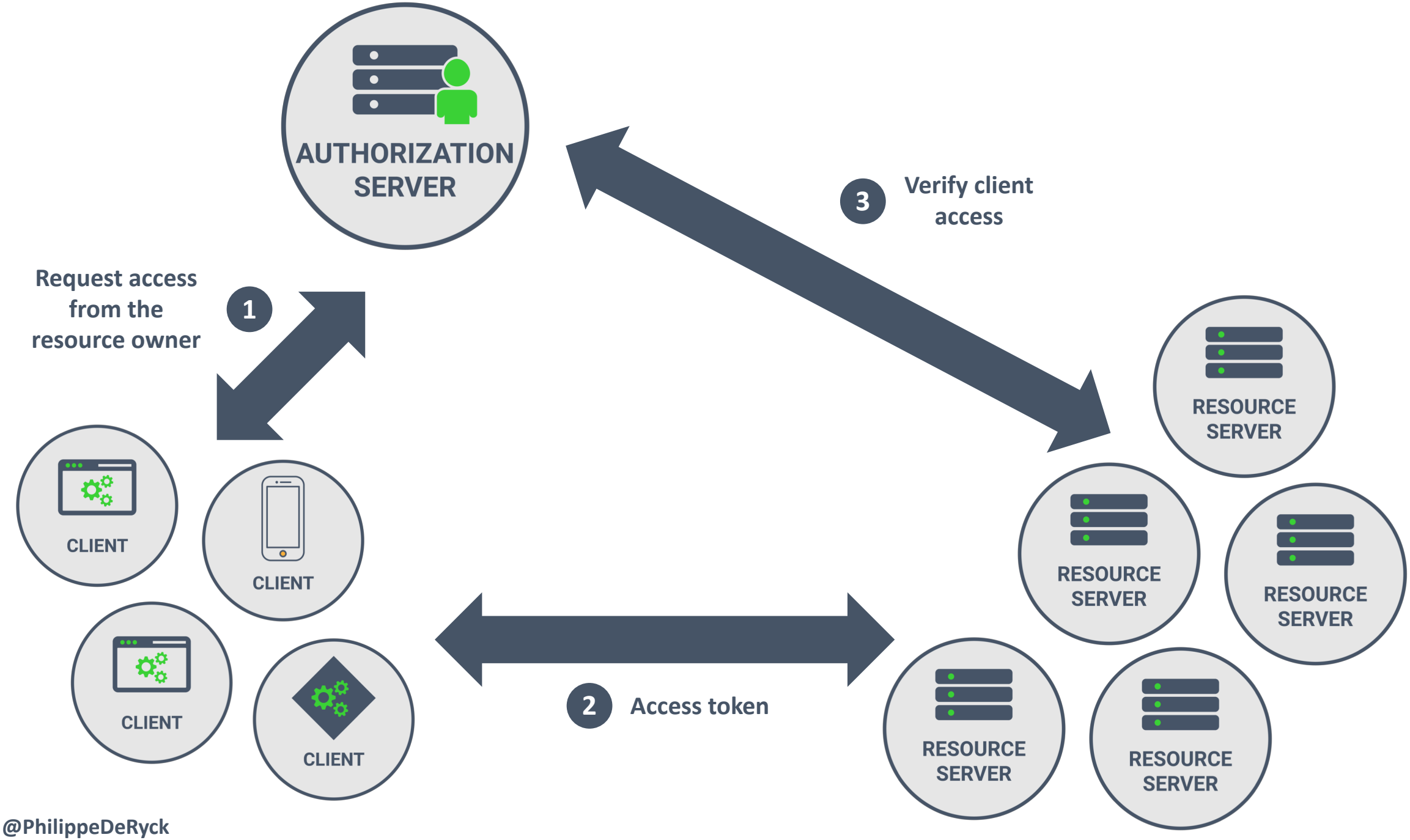
Tweepsmap by TweepsMap
intelligent publishing, communications and brand management platform. Precision segmentation actionable audience analytics. Will never Tweet without your permission http://tweepsmap.com/Info/FAQ#faq6
Permissions: read and write
Approved: Tuesday, December 27, 2016 at 10:38:06 AM
Revoke access

Twitter for Android
Twitter for Android
Permissions: read, write, and direct messages
Approved: Friday, November 6, 2015 at 9:27:28 AM
Revoke access

Twitter Web Client by Twitter, Inc.
The official client for Twitter.com
Permissions: read and write
Approved: Wednesday, August 12, 2015 at 8:18:56 AM
Revoke access

Bitly by Bitly
Save, Share and Bundle your Bitlinks
Permissions: read and write
Approved: Monday, January 23, 2017 at 7:21:02 PM
Revoke access

Buffer by Buffer
Buffer is a service to help you tweet interesting and valuable content to your Twitter followers more consistently.
Permissions: read and write
Approved: Thursday, June 9, 2016 at 12:00:52 PM
Revoke access


AUTHORIZATION SERVER
3
Verify client access
Request access from the resource owner
1
CLIENT
CLIENT
CLIENT
CLIENT
2
Access token
RESOURCE SERVER
RESOURCE SERVER
RESOURCE SERVER
RESOURCE SERVER
RESOURCE SERVER

Dr. Philippe De Ryck

- Ph.D-level understanding of the web security landscape
- Google Developer Expert (not employed by Google)
- Author of the *primer on client-side web security*
- Course curator of the SecAppDev course (https://secappdev.org)

@PhilippeDeRyck
https://PragmaticWebSecurity.com

Pragmatic Web Security

Custom training courses on web/API/JS frontend security

Technical writing, architectural security assessments and brief consultancy

Introduction to OAuth 2.0 and OpenID Connect

K

The client's perspective

The resource server's perspective

Introducing OpenID Connect

Conclusion

App details

The following app details will be visible to app users and are required to generate the API keys needed to authenticate Twitter developer products.

App name (required)

PragmaticWebSecurity

Maximum characters: **32**

Application description (required)

Share a description of your app. This description will be visible to users so this is a good place to tell them what your app does.

This is for personal use only, not for public use

Between 10 and 200 characters

Website URL (required)

https://pragmaticwebsecurity.com

Allow this application to be used to sign in with Twitter

Learn more

☐ **Enable Sign in with Twitter**

Callback URLs

OAuth 1.0a applications should specify their oauth_callback URL on the request token step, which must match the URLs provided here. To restrict your application from using callbacks, leave these blank.

https://pragmaticwebsecurity.com/twittercallback.php ✕

+ Add another

Terms of Service URL

https://

Privacy policy URL

https://

Organization name

Organization website URL

https://

Tell us how this app will be used (required)

This field is only visible to Twitter employees. Help us understand how your app will be used. What will it enable you and your customers to do?

This is for personal use only, not for public use. Will be used as a relatable example of how OAuth 2.0 works in practice.

Cancel Create

App details

The follo
to gener
products

App nan

Pragm

Applicat

Share a d
a good pla

This is

Website

https:/

Allow th
Learn m

Enab

Callback

OAuth 1.0
token step
applicatio

https://pragmaticwebsecurity.com/twittercallback.php

+ Add another

Terms of Service URL

https://

Keys and tokens

Keys, secret keys and access tokens management.

Consumer API keys

QC1wPqVwsj74TCyqmEsdxXbJB (API key)

m80Rg66HLKck1SChuRUaKGBFwqZgCwysSGZDYT8nMlzXlGBNfM (API secret key)

Regenerate

Access token & access token secret

3417260589-WdGA0zLBXyc11BrACkBANpgSzpuhuHLK8JjfZhs (Access token)

JafkKM2wJwjrMrT6GwSfJoHW42Yd0aVXG4crmX3ZmqhXq (Access token secret)

Read and write (Access level)

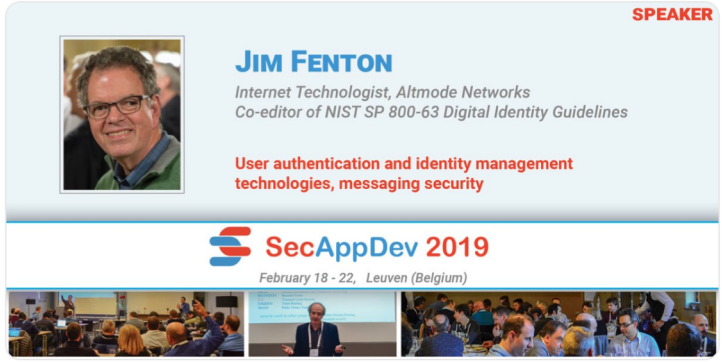
Revoke Regenerate

Don't worry, these are revoked!

Cancel Create

Scenario 1 – Show a selected number of tweets

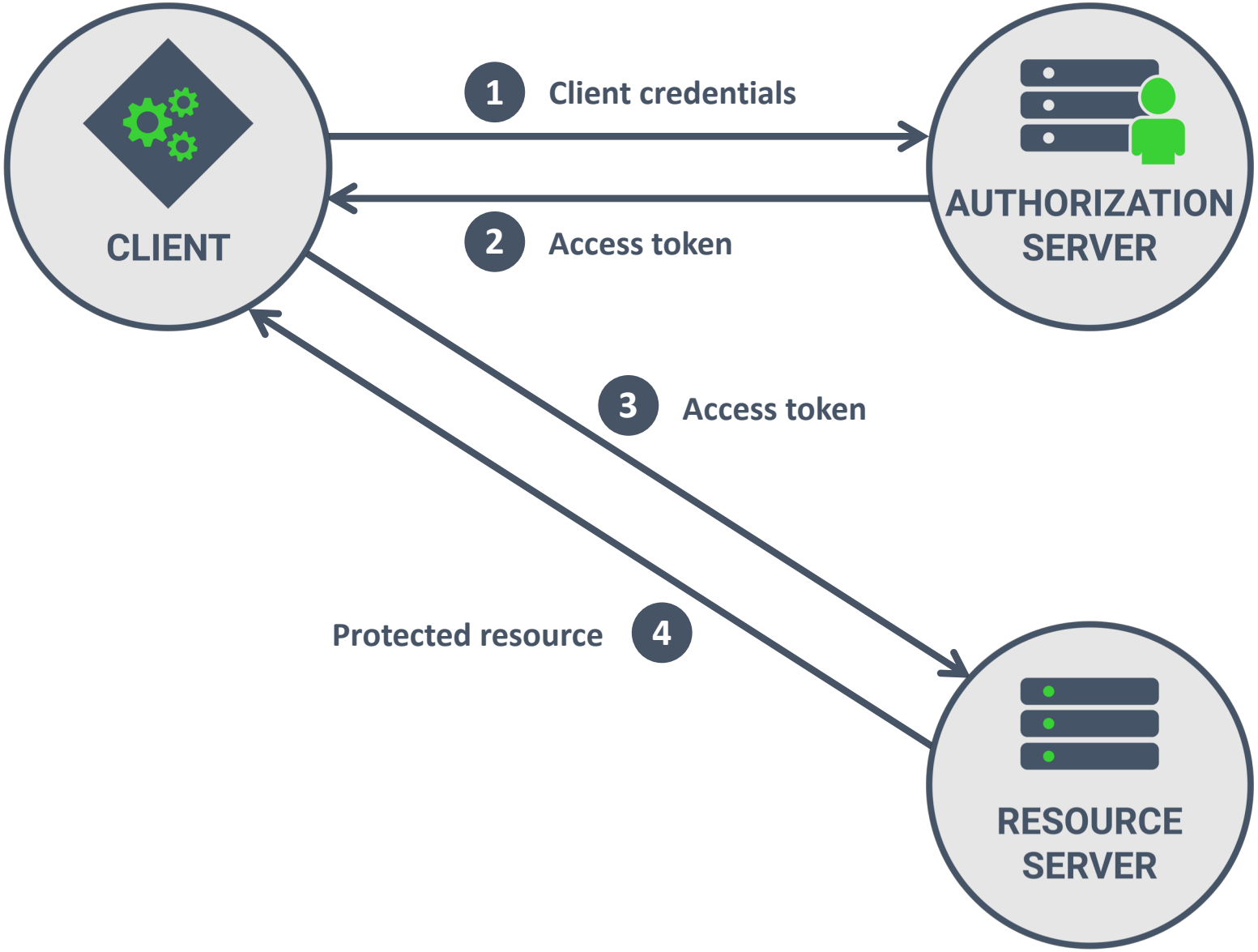
Almost every application depends on authentication, a much-debated topic. Who better to teach about it than @jimfenton, the co-editor of the #NIST SP 800-63 Digital Identity Guidelines. Proud to have Jim on board. buff.ly/1Ric8Zq



For the 15th year in a row, Bart Preneel from @CosicBe will be at SecAppDev. He is one of the world's experts on cryptography. This year, he will teach about crypto, but will also give his expert opinion on #blockchain and the current hype. buff.ly/1Ric8Zq

With a lot of excitement, we can announce @jimmesta will be part of the SecAppDev 2019 faculty. He is one of the top experts on security in DevOps environments. He also teaches one of his excellent 1-day workshops, where you dive #Kubernetes! buff.ly/1Ric8Zq

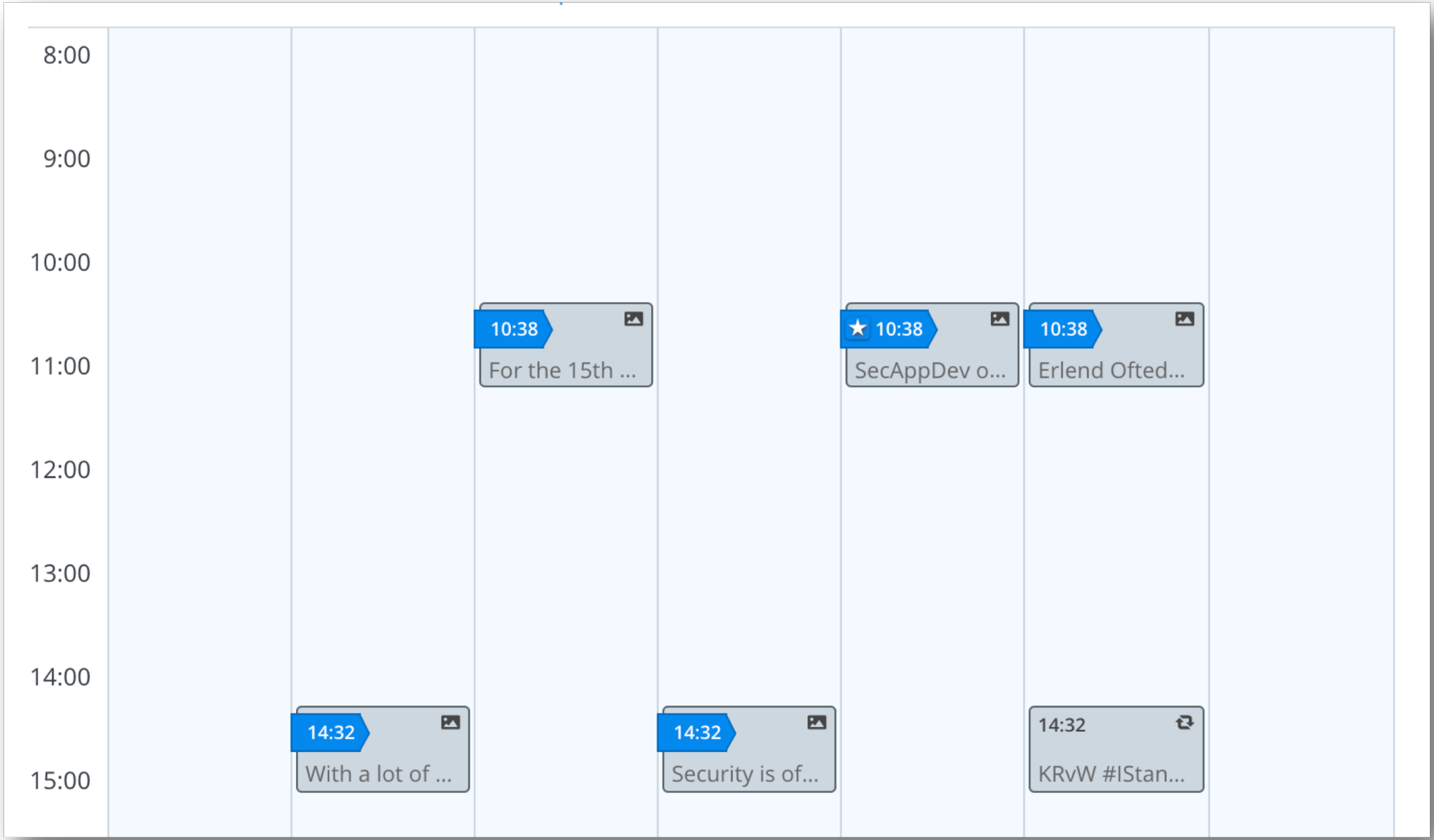
The Client Credentials Grant flow



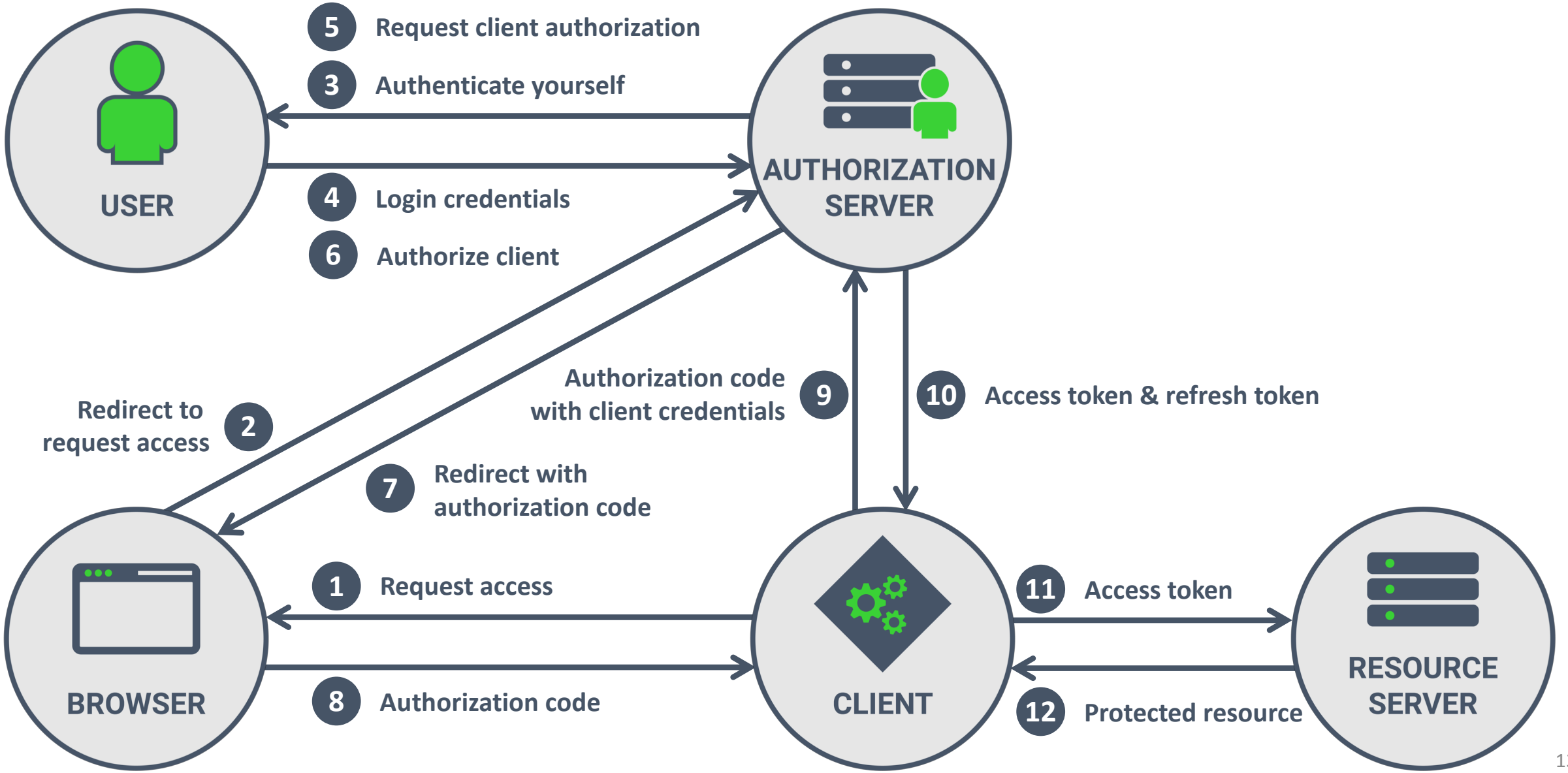
Client credentials grant

- Direct access by the client application
- Access token obtained using client credentials

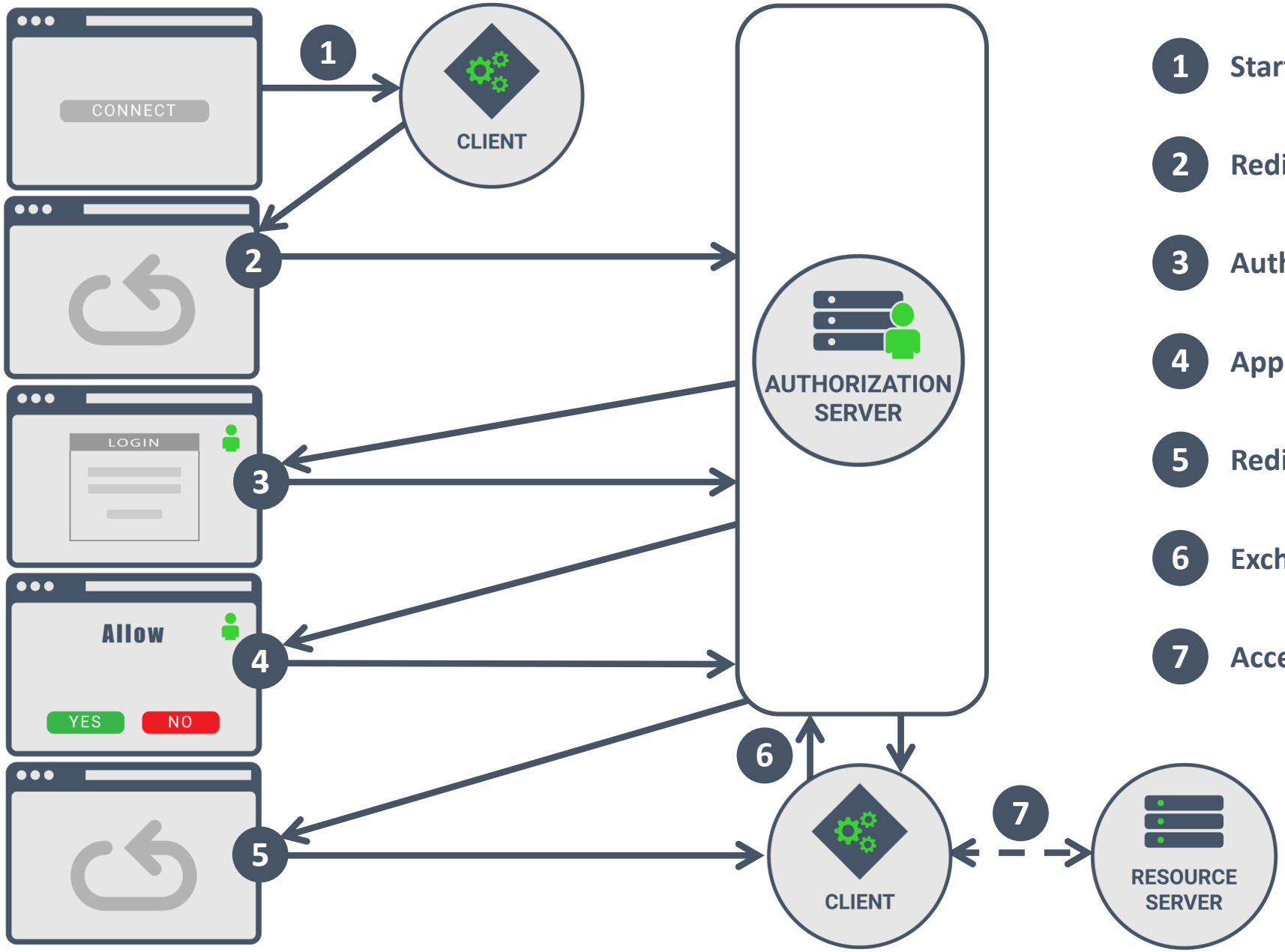
Scenario 2 – Schedule tweets on behalf of a user



The Authorization Code Grant flow



The Authorization Code Grant flow



1. Start the delegation process
2. Redirect to the authorization server
3. Authenticate the user
4. Approve access delegation
5. Redirect with authorization code
6. Exchange code for access token
7. Access protected resources

Requesting an authorization code

```
https://twitter.example.com/auth
   ?response_type=code
   &client_id=PragmaticWebSecurity
   &scope=read write
   &redirect_uri=https://pragmatic…/twittercallback.php
   &state=s0wzojm2w8c23xzprkk6
```

Response containing authorization code

```
https://pragmatic…/twittercallback.php
   ?code=eyJhb...0X4UeQ
   &state=s0wzojm2w8c23xzprkk6
```

Requesting an access token

```
POST /auth
Authorization: Basic UmFuZG9tQ2xpZW50SU...tODdlYTJmZDVhN2Rm
Host: twitter.example.com

 grant_type=authorization_code
&redirect_uri=https%3A%2F%2Fpragmaticweb…%2Ftwittercallback.php
&client_id=PragmaticWebSecurity
&code=eyJhb...0X4UeQ
```

Response containing access token

```
{
    "access_token": "eyJhbGciO...du6TY9w",
    "expires_in": 300,
    "token_type": "bearer"
    "refresh_token": "eyJhbGciO...E4wRw",
}
```

How do you USE a refresh token?

"Refreshing" an access token

```
POST /auth
Authorization: Basic UmFuZG9tQ2xpZW50SU...tODdlYTJmZDVhN2Rm
Host: twitter.example.com

 grant_type=refresh_token
&refresh_token=eyJhbGciO...E4wRw
```

Response containing new tokens

```
{
    "access_token": "eyJhbGciO...OiJC",
    "expires_in": 300,
    "token_type": "bearer"
    "refresh_token": "eyJhbGciO...OaW9uIl19",
}
```

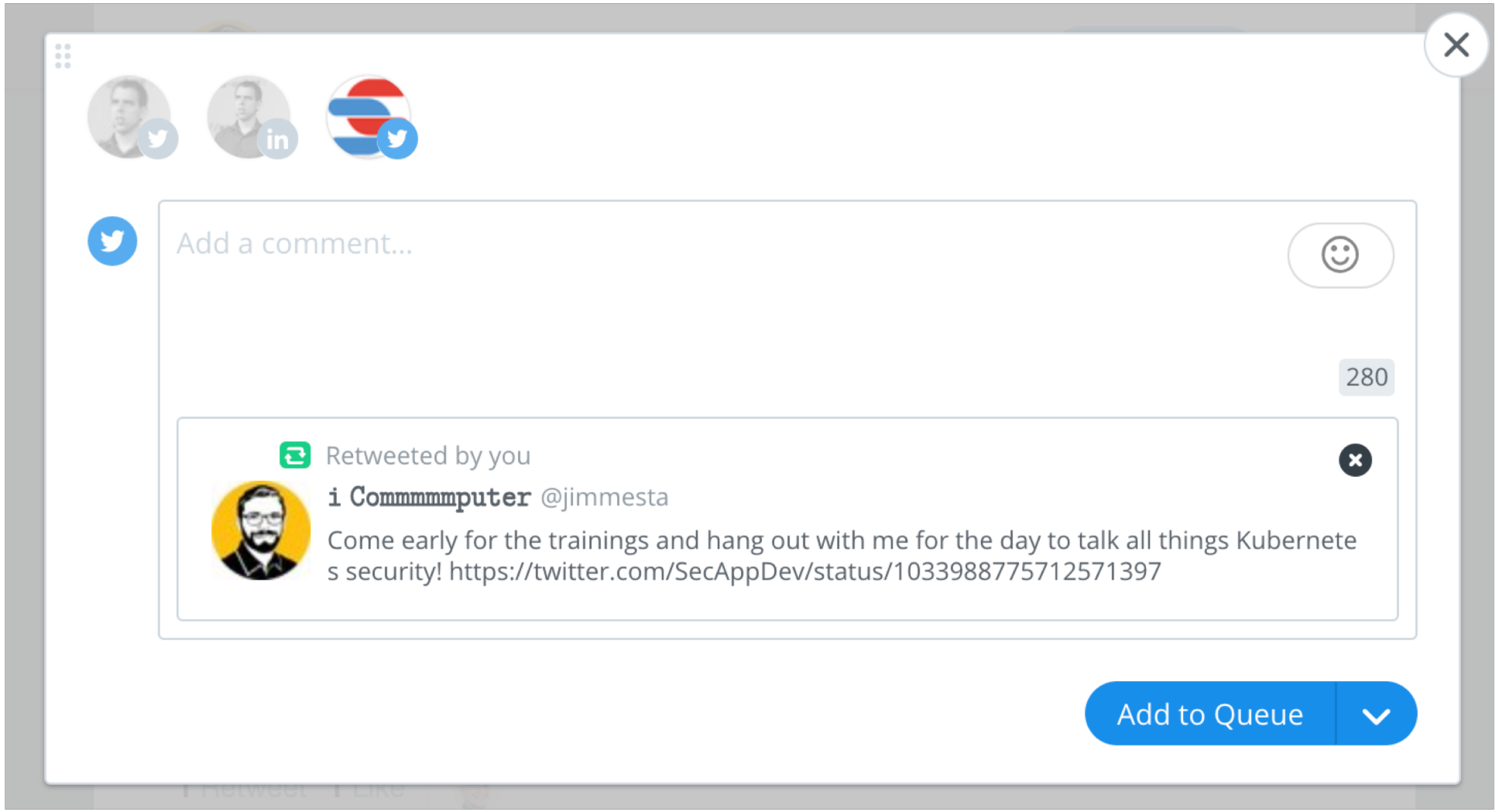
Client credentials grant

- Direct access by the client application
- Access token obtained using client credentials

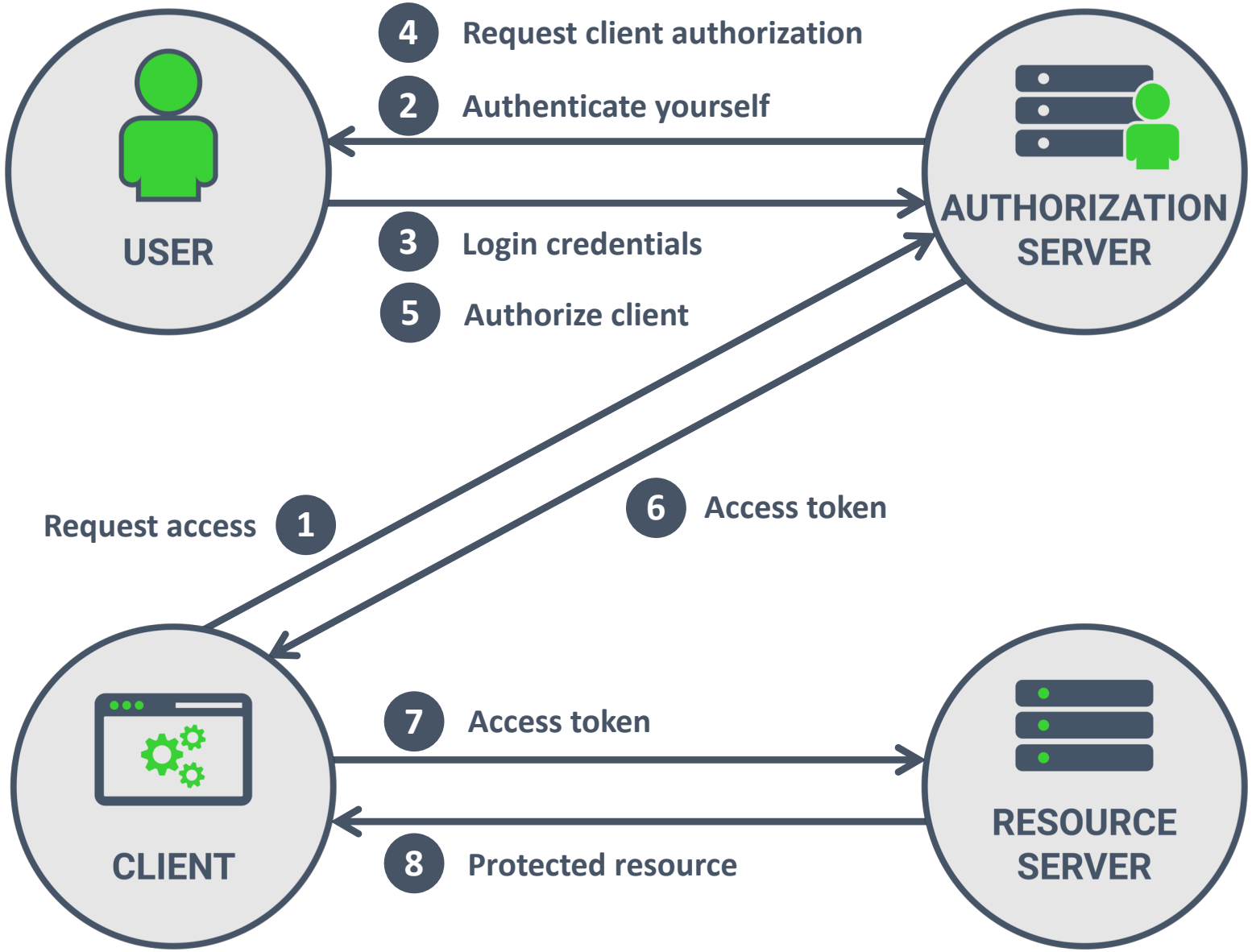
Authorization code grant

- Delegated access to a backend application
- Access token obtained by exchanging code with client credentials
- Refresh token can be used with client credentials

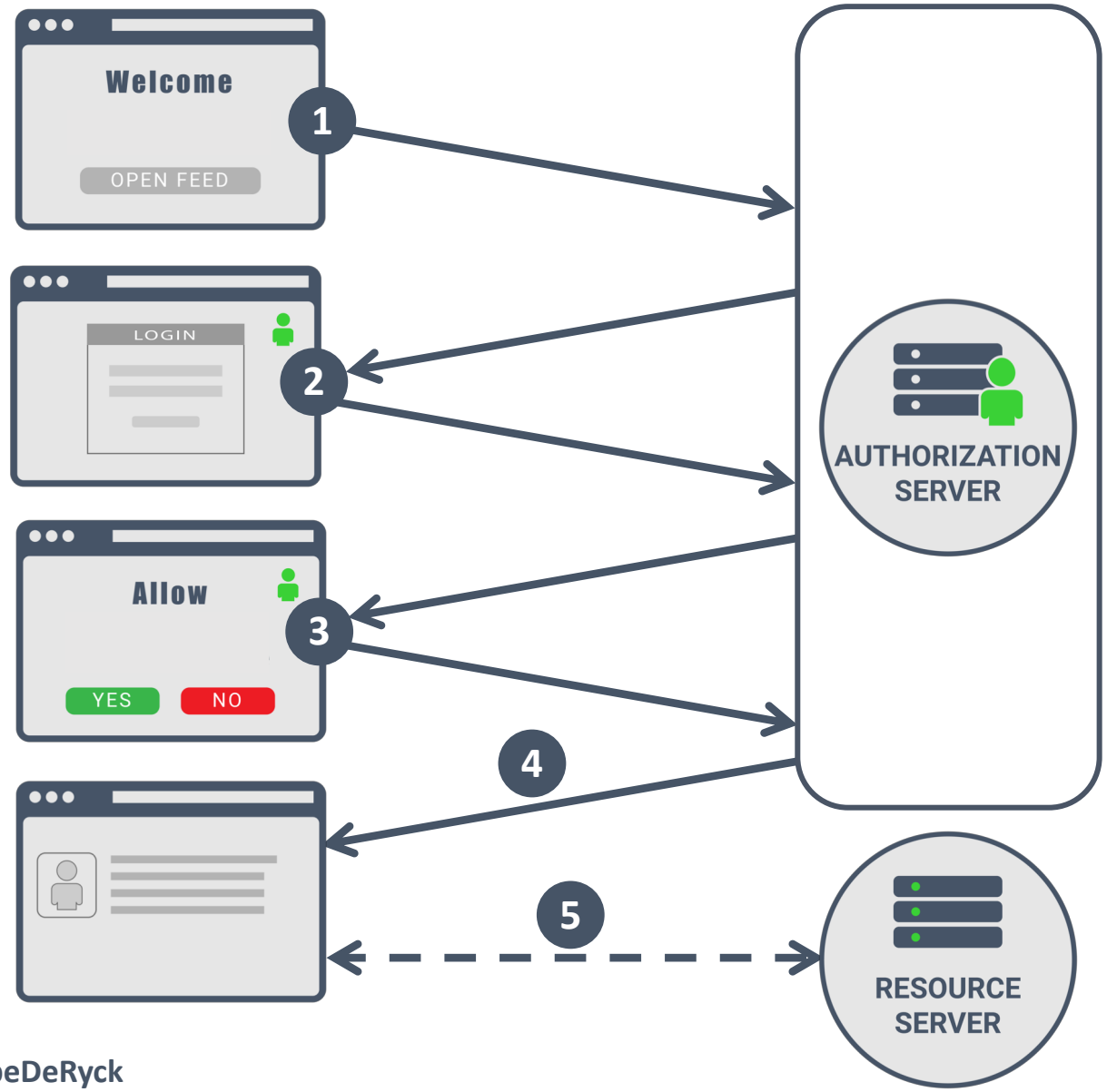
Scenario 3 – Allow live interaction on behalf of a user



The Implicit Grant flow



The Implicit Grant flow



1. Start the delegation process
2. Authenticate the user
3. Approve access delegation
4. Load callback with access token
5. Access protected resources

Requesting an access token

```
https://twitter.example.com/auth
   ?response_type=token
   &client_id=PragmaticWebSecurity
   &scope=read write
   &redirect_uri=https://pragmatic…/twittercallback.html
   &state=s0wzojm2w8c23xzprkk6
```

Response containing access token

```
https://pragmatic…/twittercallback.html
   #access_token=eyJhb...0X4UeQ
   &token_type=bearer
   &expires_in=300
   &state=s0wzojm2w8c23xzprkk6
```

Client credentials grant

- Direct access by the client application
- Access token obtained using client credentials

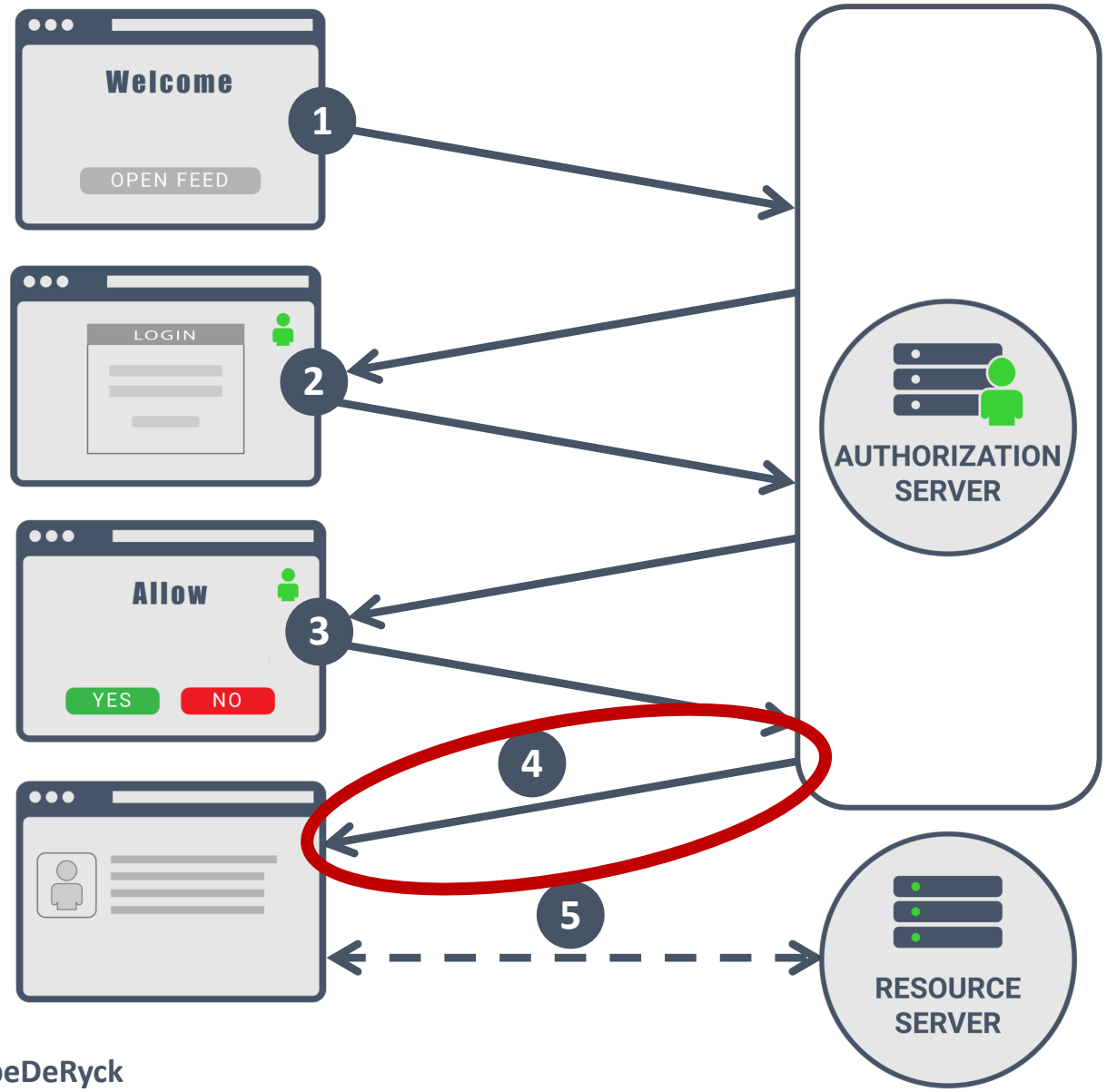
Authorization code grant

- Delegated access to a backend application
- Access token obtained by exchanging code with client credentials
- Refresh token can be used with client credentials

Implicit grant

- Delegated access to a frontend application
- Access token directly obtained through the redirect
- Not supposed to have access to refresh tokens

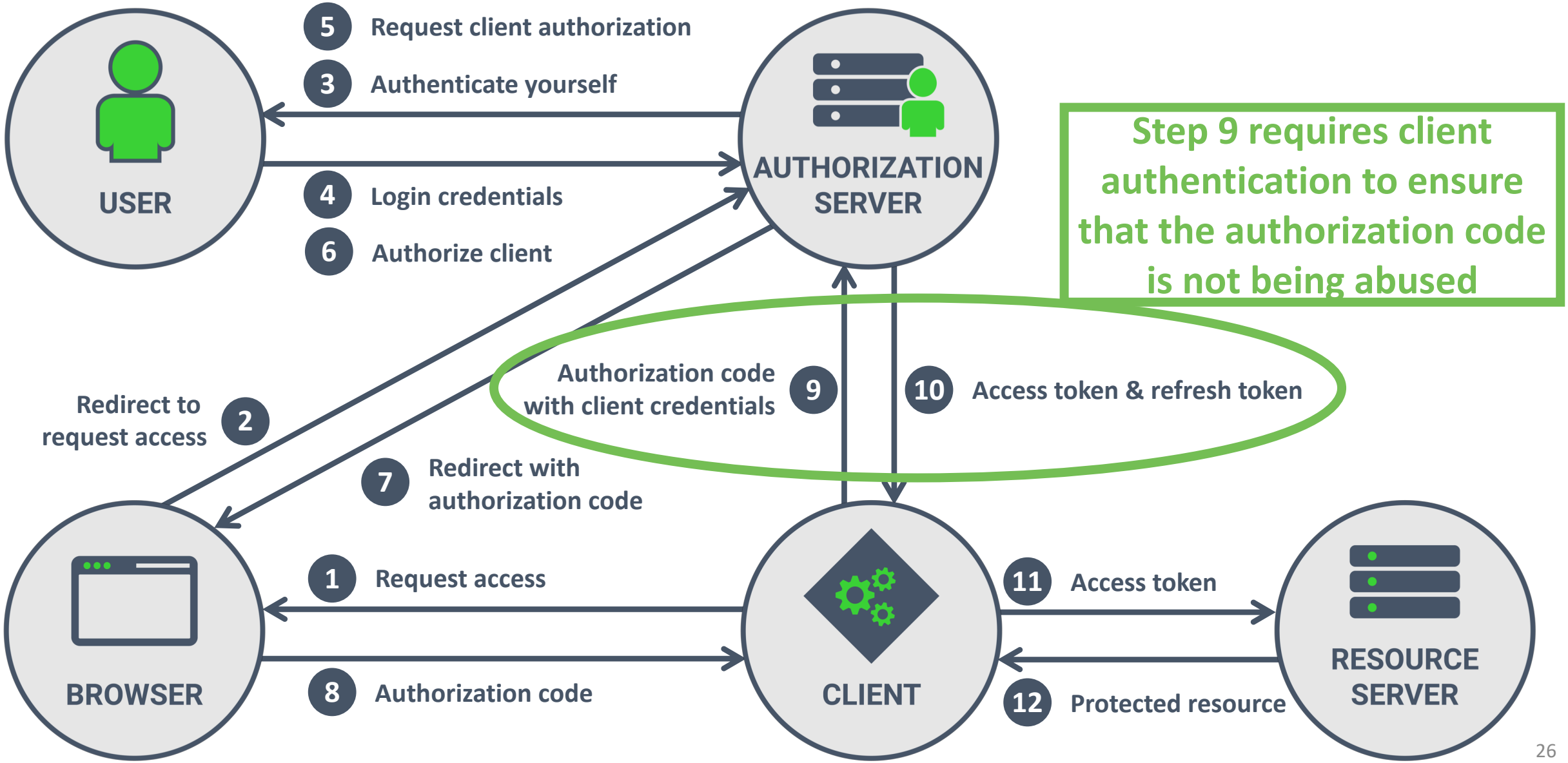
The Implicit Grant flow



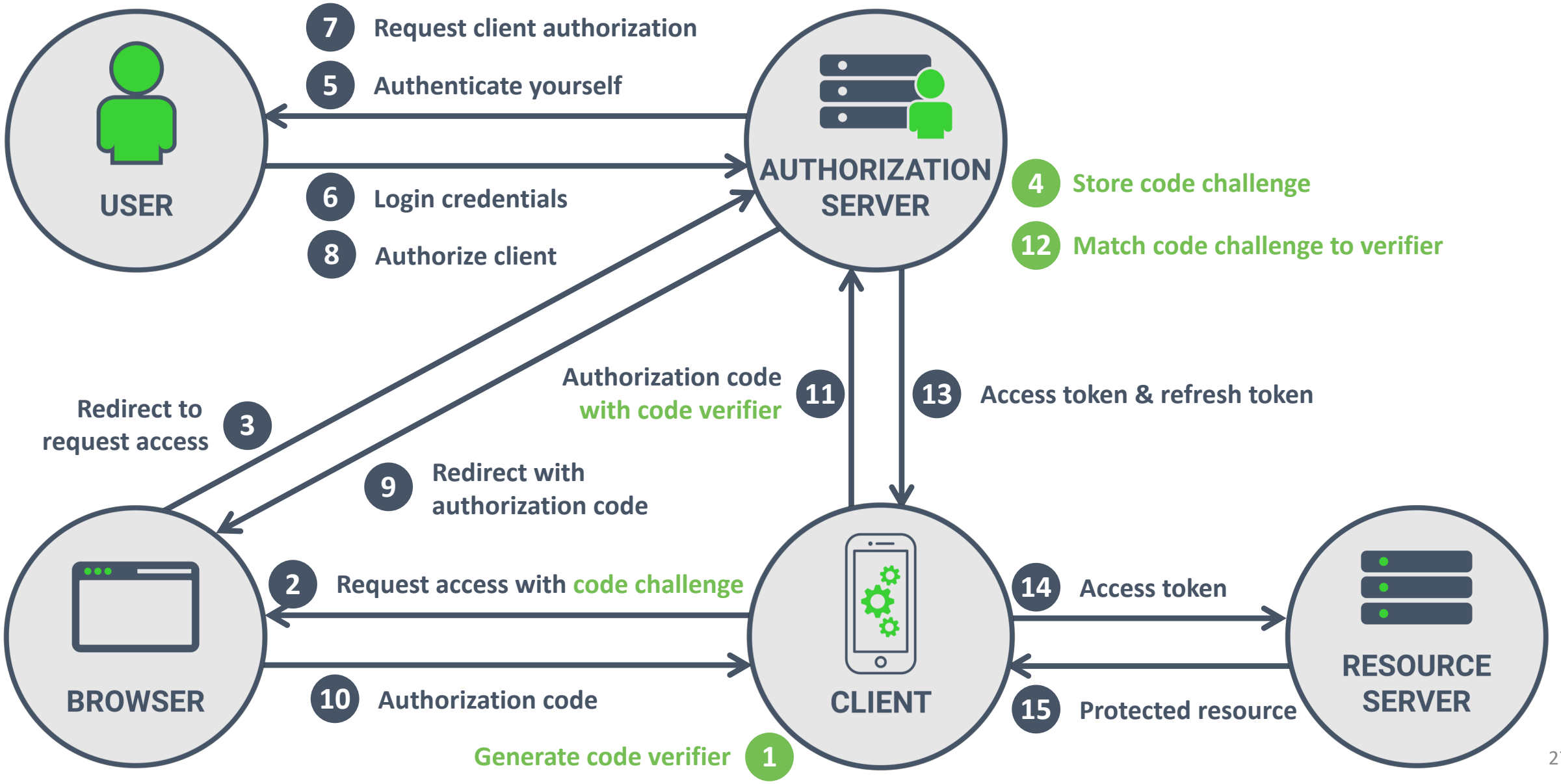
1. Start the delegation process
2. Authenticate the user
3. Approve access delegation
4. Load callback with access token
5. Access protected resources

Access token susceptible for interception (i.e., mobile inter-app communication)

The Authorization Code Grant flow



The Authorization Code Grant flow with PKCE



Details of the PKCE-based Authorization Code Grant flow

- PKCE allows public clients to run a more secure Authorization Code Grant
  - Even though the client is public, it is still expected to be a secure environment
  - Native applications were the primary target, but the same advice extends to SPAs as well
  - Refresh tokens are optional, but their use cannot be protected with the client secret

- It is crucial to ensure that the right client exchanges the authorization code
  - Instead of using client credentials, PKCE uses a ***code challenge*** and ***code verifier***
  - The code verifier is a cryptographically secure random string
    - Between 43 and 128 characters of this character set: ***[A-Z] [a-z] [0-9] - . _ ~***
  - The code challenge is a ***SHA256*** hash of the code verifier
    - The hash function uniquely connects the code challenge to the code verifier
    - The code verifier cannot be derived from the code challenge

Requesting an authorization code

```
https://twitter.example.com/auth
   ?response_type=code
   &client_id=PragmaticWebSecurity
   &scope=read write
   &redirect_uri=com.pragmaticwebsecurity:/handleToken
   &state=s0wzojm2w8c23xzprkk6
   &code_challenge=608891507E432E...0A47106CCF9504A526694
   &code_challenge_type=s256
```

Response containing authorization code

```
https://reviewer.restograde.com/index.html
   #code=eyJhb...0X4UeQ
   &state=s0wzojm2w8c23xzprkk6
```

Requesting an access token

```
POST /auth
Authorization: Basic UmFuZG9tQ2xpZW50SU...tODdlYTJmZDVhN2Rm
Host: twitter.example.com

 grant_type=authorization_code
&redirect_uri=com.pragmaticwebsecurity:%2FhandleToken
&client_id=PragmaticWebSecurity
&code=eyJhb...0X4UeQ
&code_verifier=eolom2qk30a8hv5p68o... d23k46rd4nrk6
```

Response containing access token

```
{
    "access_token": "eyJhbGciO...du6TY9w",
    "expires_in": 300,
    "token_type": "bearer"
    "refresh_token": "eyJhbGciO...E4wRw",
}
```

Client credentials grant

- Direct access by the client application
- Access token obtained using client credentials

Authorization code grant

- Delegated access to a backend application
- Access token obtained by exchanging code with client credentials
- Refresh token can be used with client credentials

Authorization code grant with PKCE

- Delegated access to a frontend application (web / mobile)
- Access token obtained by exchanging code with code verifier
- Refresh token can be used, but not protected with client credentials

Implicit grant

- More or less deprecated

Introduction to OAuth 2.0 and OpenID Connect

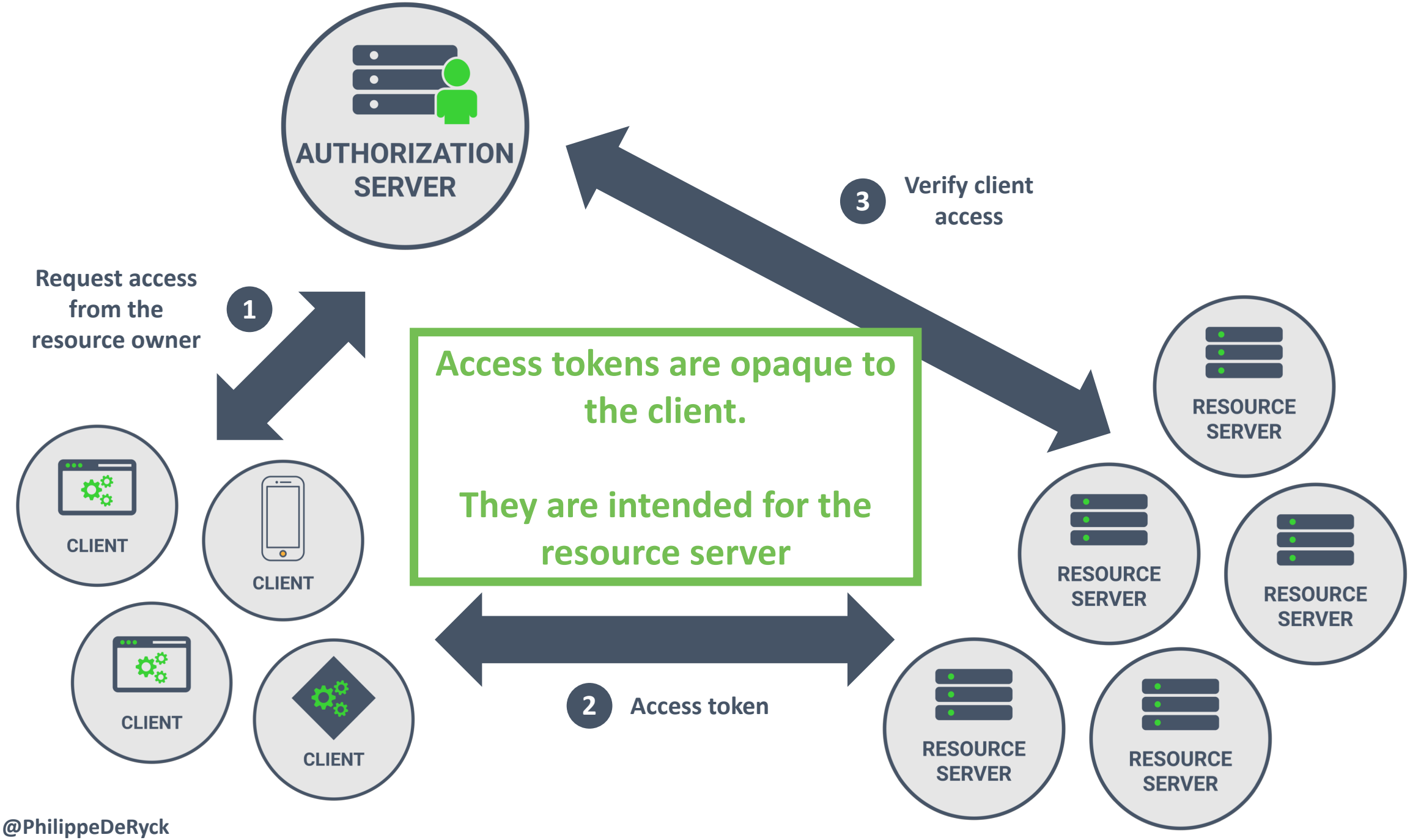
K

The client's perspective

The resource server's perspective

Introducing OpenID Connect

Conclusion


AUTHORIZATION SERVER
3
Verify client access
Request access from the resource owner
1
Access tokens are opaque to the client.
They are intended for the resource server
CLIENT
CLIENT
CLIENT
CLIENT
2
Access token
RESOURCE SERVER
RESOURCE SERVER
RESOURCE SERVER
RESOURCE SERVER
RESOURCE SERVER

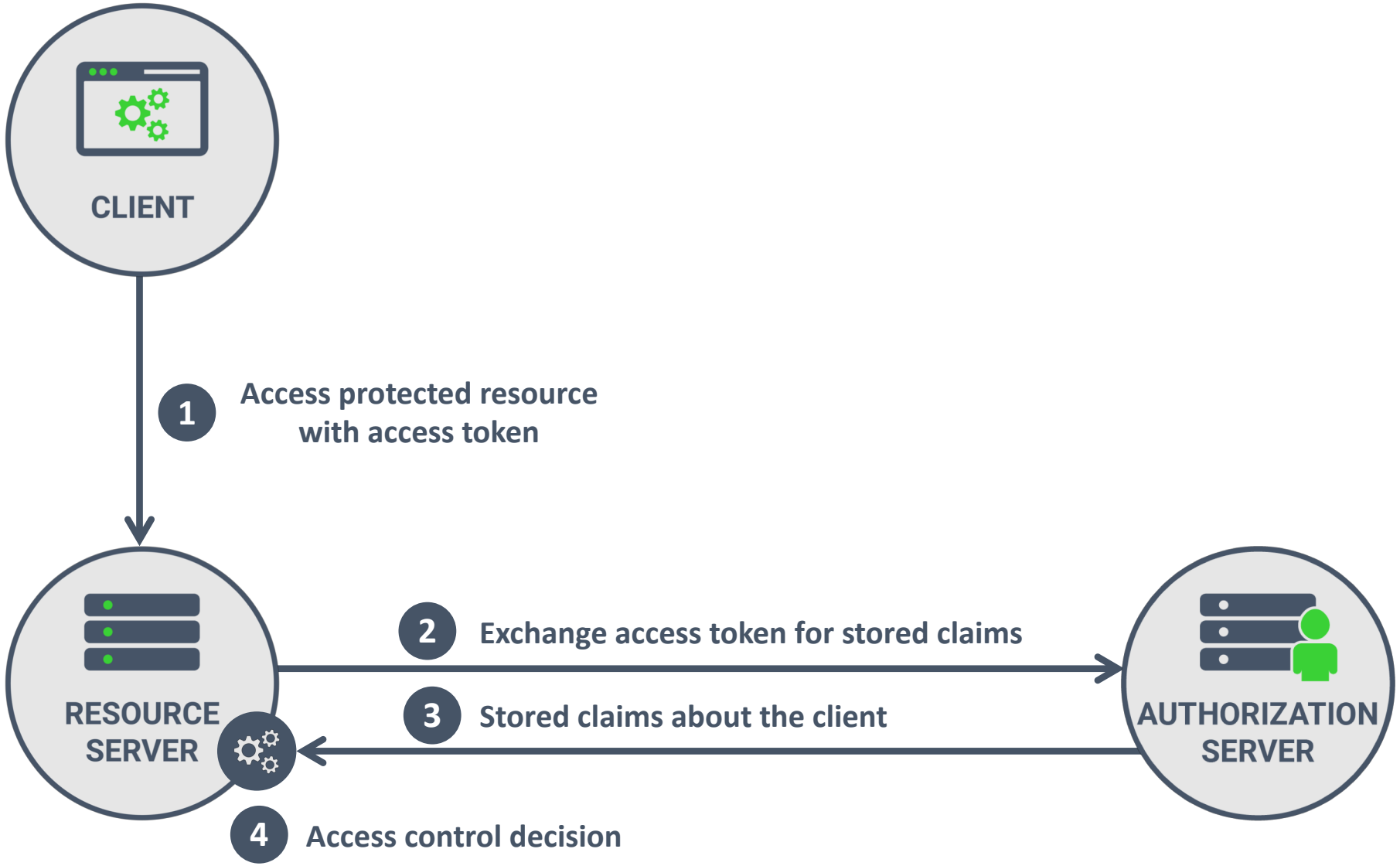
2YotnFZFEjr1zCsicMWpAA

eyJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJqZG9lQGV4YW1wbGUuY29tIiwi
YXVkIjoiaHR0cHM6Ly9hcGkuZXhhbXBsZS5jb20iLCJhenAiOiJSY
W5kb21DbGllbnRJRCIsImlzcyI6Imh0dHBzOi8vYXV0aG9yaXphdGlvbn
NlcnZlci5leGFtcGxlLmNvbS8iLCJleHAiOjE0MTkzNTYyMzgsIml
hdCI6MTQxOTM1MDIzOCwic2NvcGUiOiJyZWFkIHdyaXRlIiwianRpIjoi
NDA1YjRkNGUtODUwMS00ZTFhLWExMzgtZWQ4NDU1Y2QxZDQ3In0.
FCk3Wo8DnFEHb02JCd9BWAHQ48BBt3n2YLQV6TpLMpFvTRNCZJAA-
aEH4LrE7oVejvGd7YWGDy2Vzb7x-Bpg7yMYxozUerCkMy_F4Iw_xctgE
J3WF_TTJFhISGNoWlFXspM5d9EQvMvk0JxAovhE0HfXv5GCosGy-
0oT7ShQrwZLBIwE9d0ceUcmly42dvDZSsqHDIzPjrFzvpXwbZqq_sRFnh
6MHlmmug7tlUCs85caoLhfSweaT0z7ED8P2Tsg_HgmnaaeDapszG6Lcke
BglqYwbRHy6X6LAcJfAkkwAlqrU0Vu4azsuE8BsLPKMYzu9ZeCoH
dLHYdtz-I0yKQ

2YotnFZFEjr1zCsicMWpAA

eyJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJqZG9lQGV4YW1wbGUuY29tIiwi
YXVkIjoiaHR0cHM6Ly9hcGkuZXhhbXBsZS5jb20iLCJhenAiOiJSY
W5kb21DbGllbnRJRCIsImlzcyI6Imh0dHBzOi8vYXV0aG9yaXphdGlvbn
NlcnZlci5leGFtcGxlLmNvbS8iLCJleHAiOjE0MTkzNTYyMzgsIml
hdCI6MTQxOTM1MDIzOCwic2NvcGUiOiJyZWFkIHdyaXRlIiwianRpIjoi
NDA1YjRkNGUtODUwMS00ZTFhLWExMzgtZWQ4NDU1Y2QxZDQ3In0.
FCk3Wo8DnFEHb02JCd9BWAHQ48BBt3n2YLQV6TpLMpFvTRNCZJAA-
aEH4LrE7oVejvGd7YWGDy2Vzb7x-Bpg7yMYxozUerCkMy_F4Iw_xctgE
J3WF_TTJFhISGNoWlFXspM5d9EQvMvk0JxAovhE0HfXv5GCosGy-
0oT7ShQrwZLBIwE9d0ceUcmly42dvDZSsqHDIzPjrFzvpXwbZqq_sRFnh
6MHlmmug7t1UCs85caoLhfSweaT0z7ED8P2Tsg_HgmnaaeDapszG6Lcke
BglqYwbRHy6X6LAcJfAkkwAlqrU0Vu4azsuE8BsLPKMYzu9ZeCoH
dLHYdtz-I0yKQ

Token introspection for reference tokens



Token introspection request

```
POST /token_info
Authorization: Bearer eyJhb...N2Rm
Host: twitter.example.com

 token=2YotnFZFEjr1zCsicMWpAA
&token_type_hint=access_token
```

Token introspection response

```
{
    "active": true
    "client_id": "PragmaticWebSecurity",
    "sub": "Z5O3upPC88QrAjx00dis"
    "exp": 1419356238,
    "scope": "read write"
}
```

Reference tokens

- An identifier pointing to metadata kept by the authorization server
- Authorization server retains full control over the metadata
- Requires a backchannel request when received by the resource server
- Easy to revoke if needed

2YotnFZFEjr1zCsicMWpAA

eyJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJqZG9lQGV4YW1wbGUuY29tIiwi
YXVkIjoiaHR0cHM6Ly9hcGkuZXhhbXBsZS5jb20iLCJhenAiOiJSY
W5kb21DbGllbnRJRCIsImlzcyI6Imh0dHBzOi8vYXV0aG9yaXphdGlvbn
NlcnZlci5leGFtcGxlLmNvbS8iLCJleHAiOjE0MTkzNTYyMzgsIml
hdCI6MTQxOTM1MDIzOCwic2NvcGUiOiJyZWFkIHdyaXRlIiwianRpIjoi
NDA1YjRkNGUtODUwMS00ZTFhLWExMzgtZWQ4NDU1Y2QxZDQ3In0.
FCk3Wo8DnFEHb02JCd9BWAHQ48BBt3n2YLQV6TpLMpFvTRNCZJAA-
aEH4LrE7oVejvGd7YWGDy2Vzb7x-Bpg7yMYxozUerCkMy_F4Iw_xctgE
J3WF_TTJFhISGNoWlFXspM5d9EQvMvk0JxAovhE0HfXv5GCosGy-
0oT7ShQrwZLBIwE9d0ceUcmly42dvDZSsqHDIzPjrFzvpXwbZqq_sRFnh
6MHlmmug7tlUCs85caoLhfSweaT0z7ED8P2Tsg_HgmnaaeDapszG6Lcke
BglqYwbRHy6X6LAcJfAkkwAlqrU0Vu4azsuE8BsLPKMYzu9ZeCoH
dLHYdtz-I0yKQ

PAYLOAD: DATA


```
{
  "sub": "philippe@secappdev.org",
  "azp": "PragmaticWebSecurity",
  "iss": "https://twitter.example.com/",
  "exp": 1419356238,
  "iat": 1419350238,
  "scope": "read write",
  "jti": "405b4d4e-8501-4e1a-a138-ed8455cd1d47"
}
```

Reference tokens

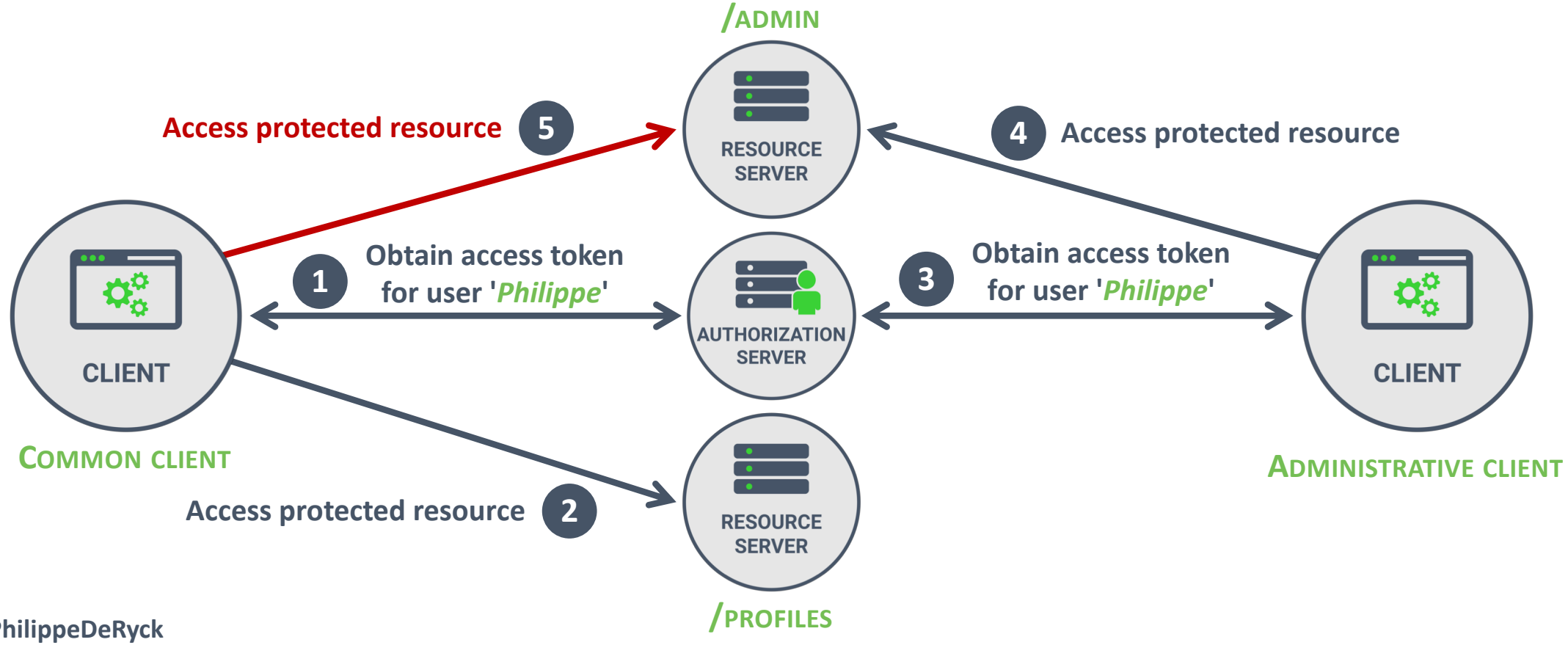
- An identifier pointing to metadata kept by the authorization server
- Authorization server retains full control over the metadata
- Requires a backchannel request when received by the resource server
- Easy to revoke if needed

Self-contained tokens

- The token itself contains the metadata used by the authorization server
- Stored on the client, so out of reach from the authorization server
- Can be used independently by the resource server after integrity check
- Hard or impossible to revoke

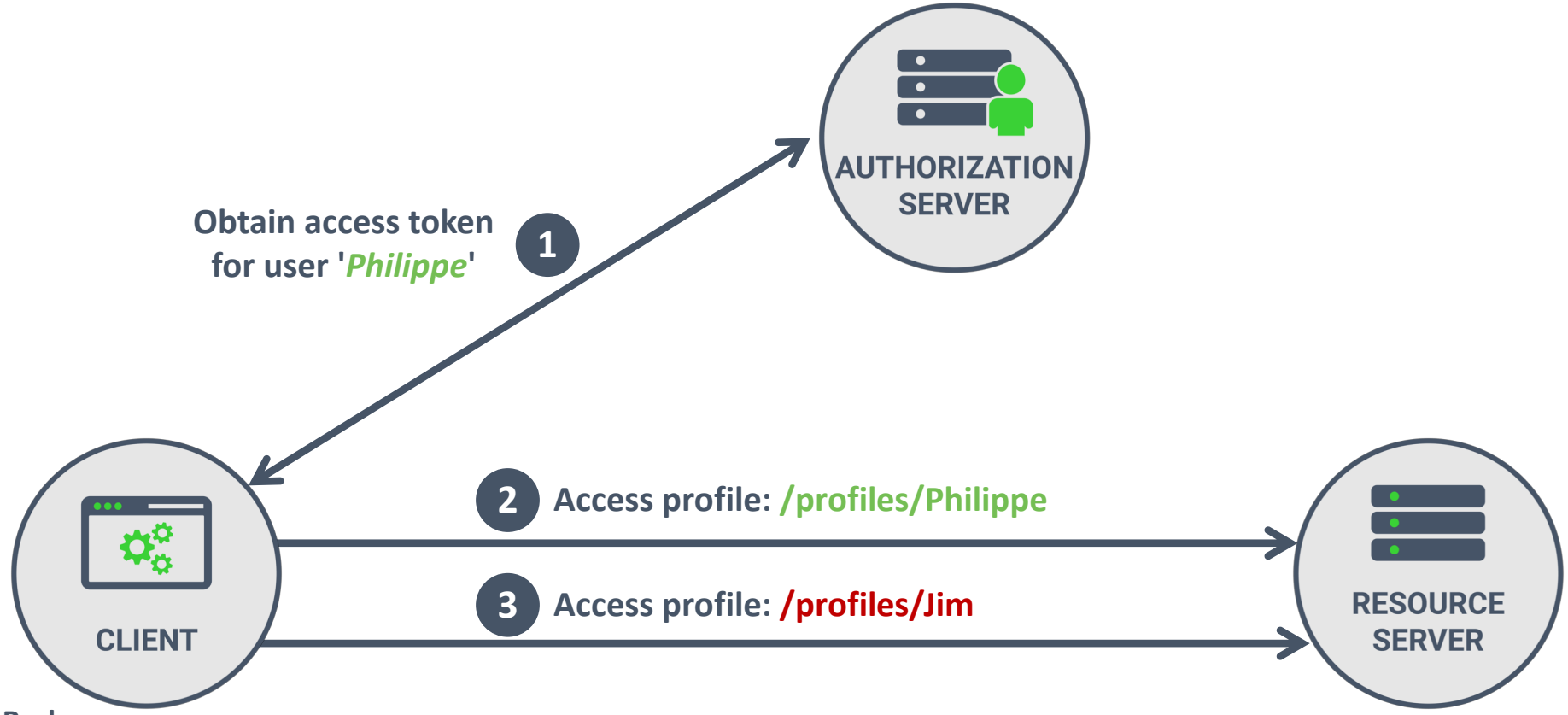
THE ESSENCE OF OAUTH 2.0 AUTHORIZATION DECISIONS

- The resource server needs to authorize both the client and the resource owner
  - The decoupling of these concepts in OAuth 2.0 causes a lot of confusion
  - Authorization of only one of these aspects creates severe security vulnerabilities



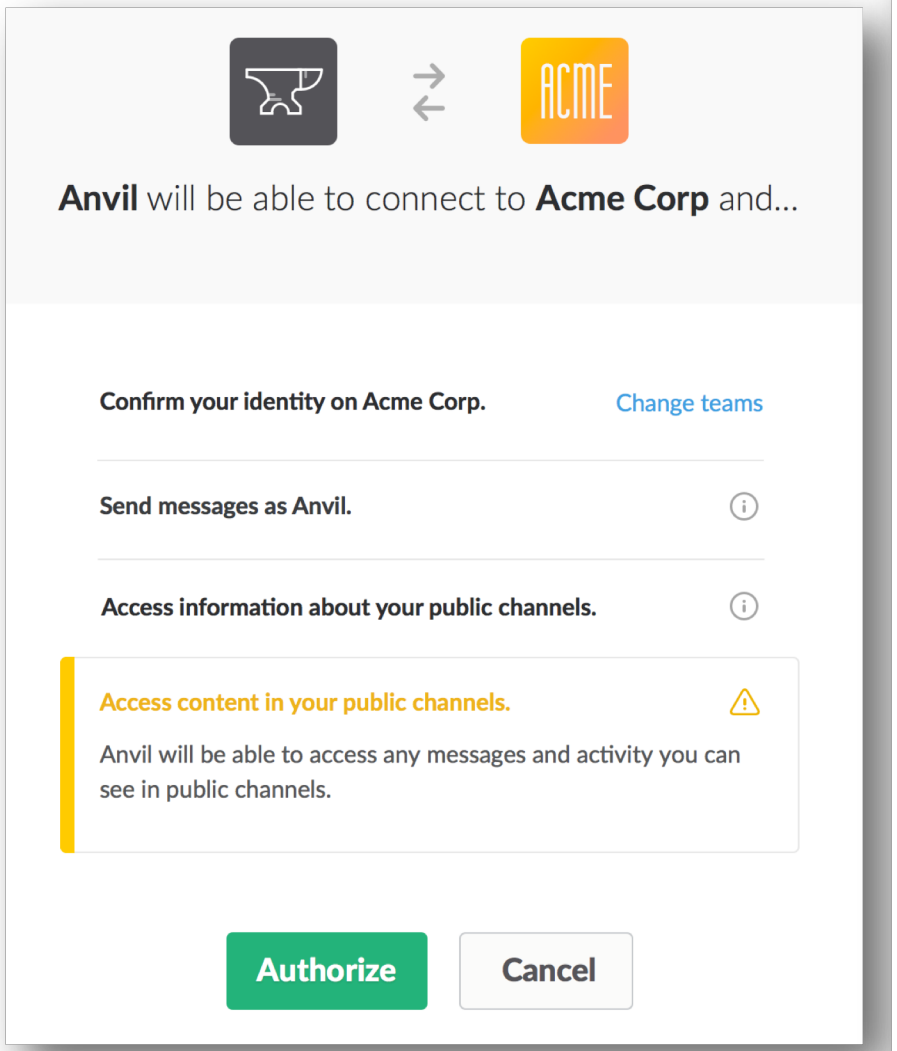
THE ESSENCE OF OAUTH 2.0 AUTHORIZATION DECISIONS

- The resource server needs to authorize both the client and the resource owner
  - The decoupling of these concepts in OAuth 2.0 causes a lot of confusion
  - Authorization of only one of these aspects creates severe security vulnerabilities



scope=read write

Scopes as used by the Slack API



OAuth Scope	Associated Methods	
`channels:history`	channels.history	channels.replies
`channels:read`	channels.info	channels.list
`channels:write`	channels.archive channels.create channels.invite channels.join channels.kick channels.leave	channels.mark channels.rename channels.setPurpose channels.setTopic channels.unarchive conversations.join
`chat:write:bot`	chat.delete chat.postEphemeral	chat.postMessage chat.update
`chat:write:user`	chat.delete chat.meMessage chat.postEphemeral	chat.postMessage chat.update

Scopes as used by the Google API

Google Analytics API, v3

Scopes	
https://www.googleapis.com/auth/analytics	View and manage your Google Analytics data
https://www.googleapis.com/auth/analytics.edit	Edit Google Analytics management entities
https://www.googleapis.com/auth/analytics.manage.users	Manage Google Analytics Account users by email address
https://www.googleapis.com/auth/analytics.manage.users.readonly	View Google Analytics user permissions
https://www.googleapis.com/auth/analytics.provision	Create a new Google Analytics account along with its default property and view
https://www.googleapis.com/auth/analytics.readonly	View your Google Analytics data
https://www.googleapis.com/auth/analytics.user.deletion	Manage Google Analytics user deletion requests

Analytics Reporting API, v4

Scopes	
https://www.googleapis.com/auth/analytics	View and manage your Google Analytics data
https://www.googleapis.com/auth/analytics.readonly	View your Google Analytics data

Best practices when using scopes

- Clients should request an access token with minimal scopes
  - A minimal scope limits the harm that can be done with a stolen access token
  - Security engineering best practice
- Granting scopes is the responsibility of the authorization server
  - Scopes can be hardcoded, but can also be approved by the user
- If needed, clients can request additional scopes afterward
  - Traditionally, this involves running a second OAuth 2.0 flow
    - A new flow issues a new authorization code or access token, which is quite noisy
  - New specification proposes ***incremental authorization*** to make this easier
    - The current access token is augmented to include the new scopes

Introduction to OAuth 2.0 and OpenID Connect

K

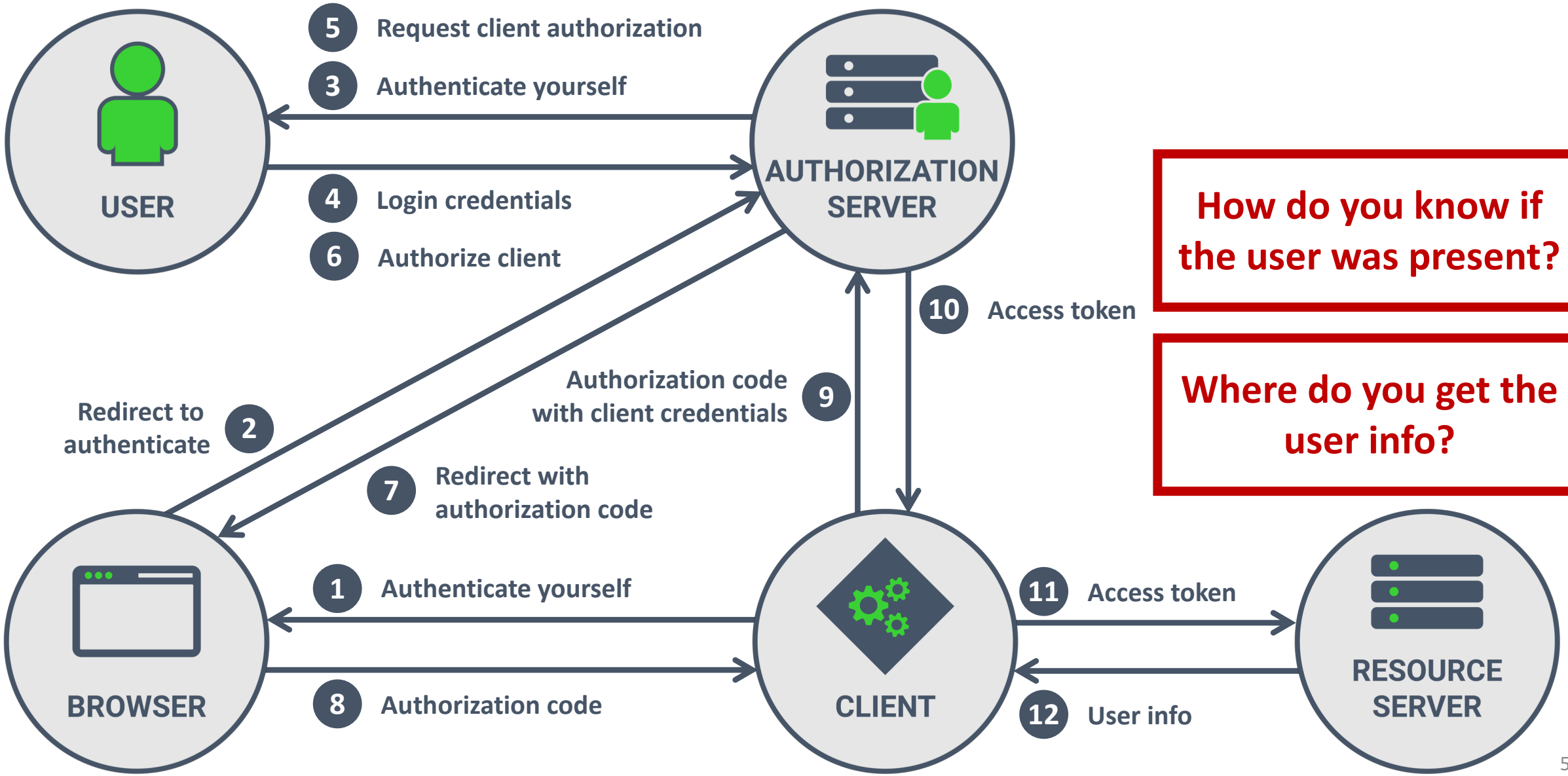
The client's perspective

The resource server's perspective

Introducing OpenID Connect

Conclusion

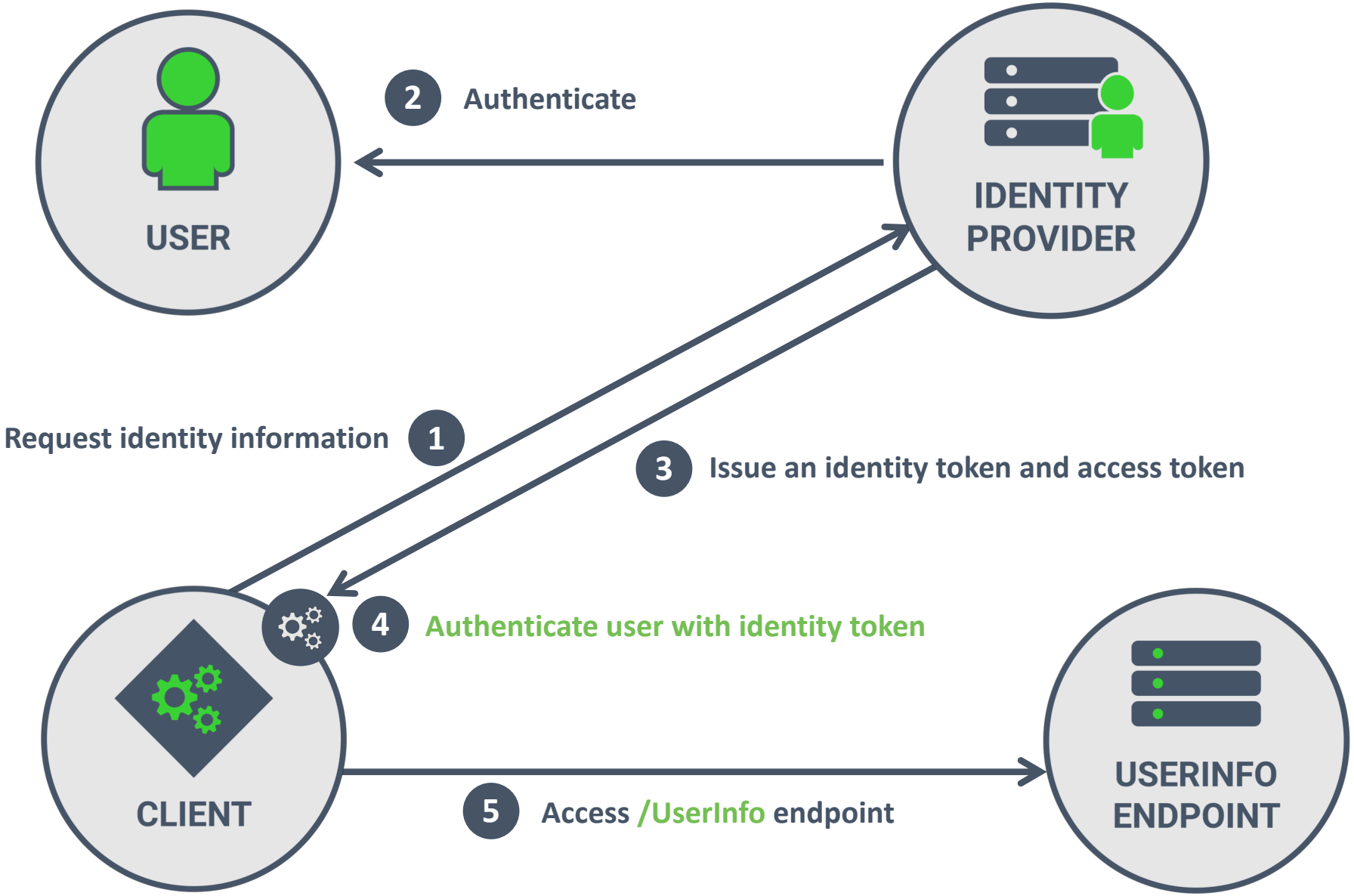
Pseudo-authentication with OAuth 2.0



How do you know if the user was present?

Where do you get the user info?

Conceptual overview of OpenID Connect



PAYLOAD: DATA

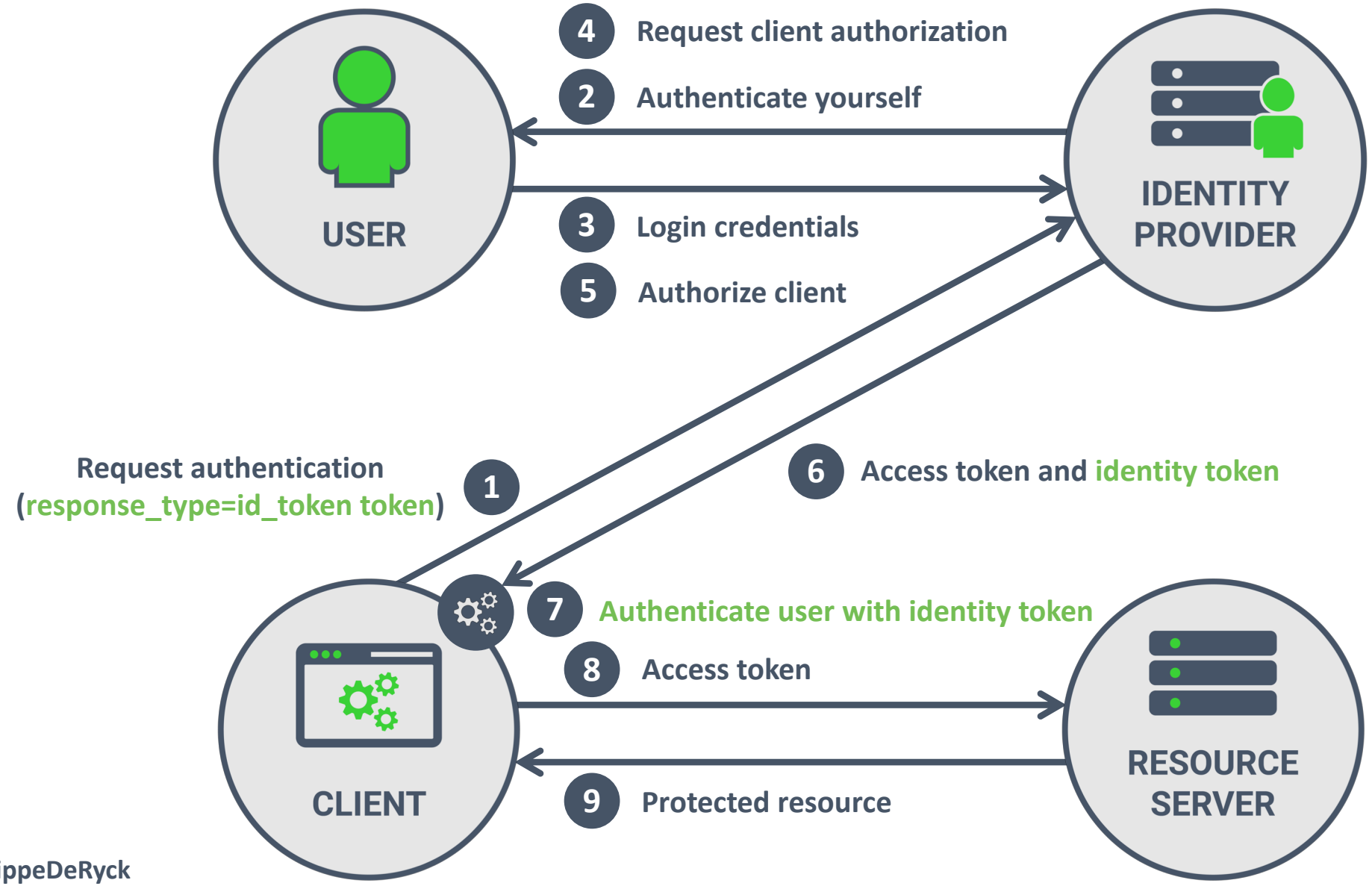

```
{
  "nickname": "philippe",
  "name": "Philippe De Ryck",
  "updated_at": "2019-02-17T05:35:37.127Z",
  "email": "philippe@pragmaticwebsecurity.com",
  "email_verified": true,
  "iss": "https://pragmaticwebsecurity.eu.auth0.com/",
  "sub": "auth0|5c4720e833d46068468cc7ca",
  "aud": "zqTuXYxr8Xnrwtnl6uNEeTgdSidR6qcZ",
  "iat": 1550400912,
  "exp": 1550422512,
  "nonce": "6fb13493044f4fdf92b587ca42c3068b"
}
```

PAYLOAD: DATA


```
{
  "nickname": "philippe",
  "name": "Philippe De Ryck",
  "updated_at": "2019-02-17T05:35:37.127Z",
  "email": "philippe@pragmaticwebsecurity.com",
  "email_verified": true,
  "iss": "https://pragmaticwebsecurity.eu.auth0.com/",
  "sub": "auth0|5c4720e833d46068468cc7ca",
  "aud": "zqTuXYxr8Xnrwtnl6uNEeTgdSidR6qcZ",
  "iat": 1550400912,
  "exp": 1550422512,
  "nonce": "6fb13493044f4fdf92b587ca42c3068b"
}
```


The "sub" claim is guaranteed to be unique for this issuer

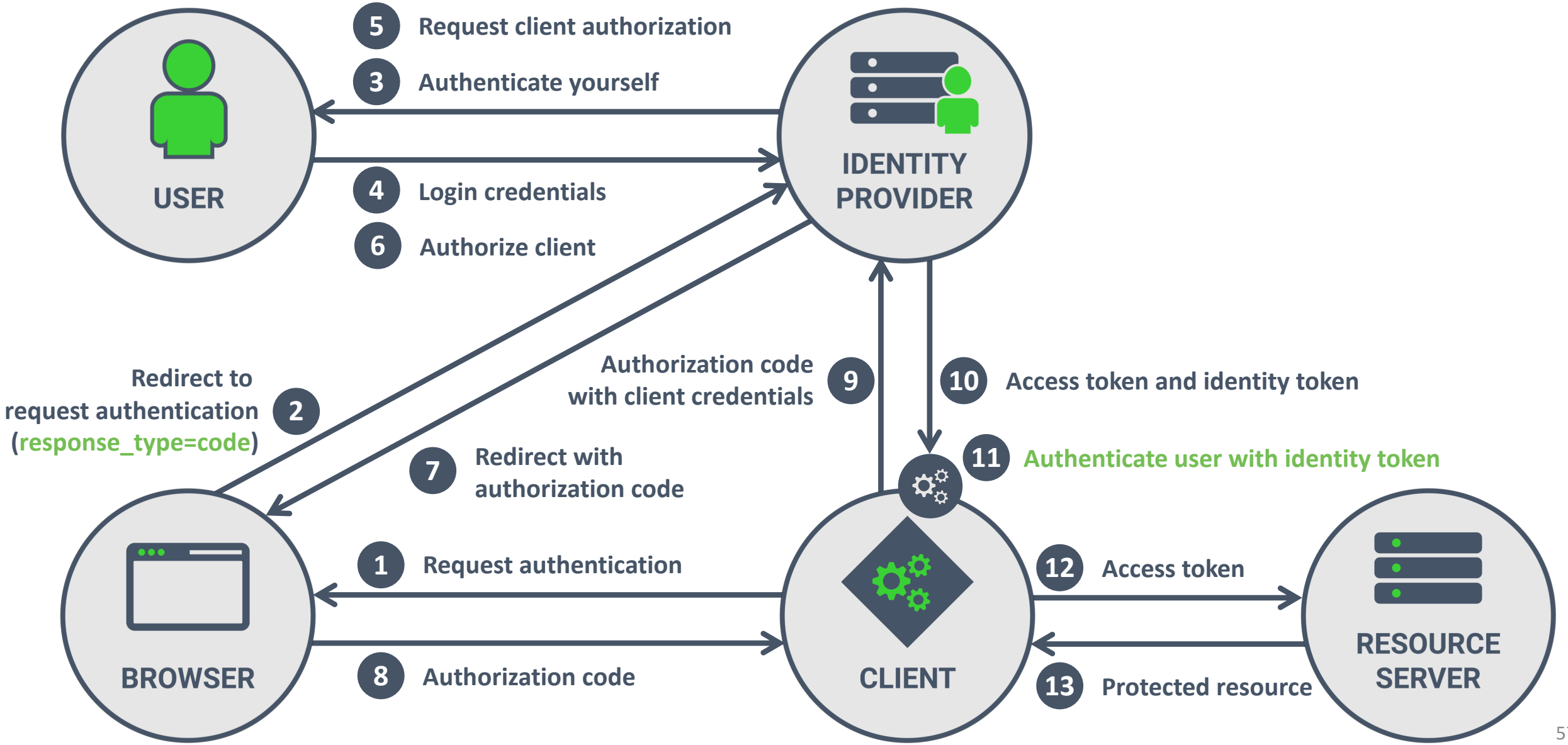
The OIDC Implicit Grant flow



Implicit grant

- Identity token is intended for the frontend application
- Allows establishing the user's identity in the frontend only

The OIDC Authorization Code Grant flow



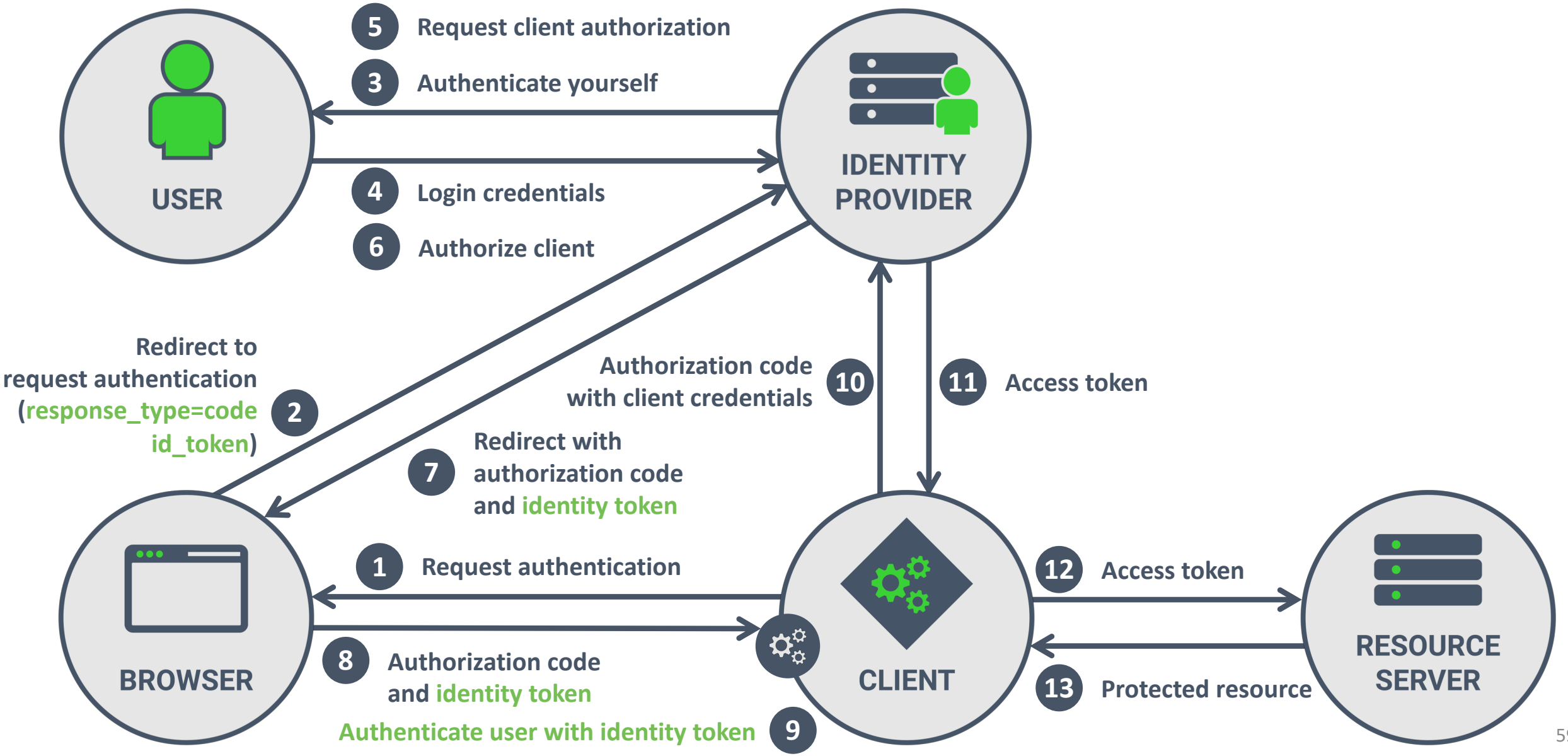
Implicit grant

- Identity token is intended for the frontend application
- Allows establishing the user's identity in the frontend only

Authorization code grant

- Identity token is intended for the backend application
- Allows connecting the identity of the user to an internal user concept

THE OIDC HYBRID FLOW



Implicit grant

- Identity token is intended for the frontend application
- Allows establishing the user's identity in the frontend only

Authorization code grant

- Identity token is intended for the backend application
- Allows connecting the identity of the user to an internal user concept

Hybrid

- Identity token is intended for the backend application
- Allows connecting the identity of the user to an internal user concept
- The backend must check if the audience of the token matches its client id

pragmaticwebsecurity.eu.auth0.com

https://pragmaticwebsecurity.eu.auth0.com/.well-known/openid-configuration

JSON | Raw Data | Headers

Save | Copy | Collapse All | Expand All | Filter JSON


```
issuer:                                  "https://pragmaticwebsecurity.eu.auth0.com/"
authorization_endpoint:                  "https://pragmaticwebsecurity.eu.auth0.com/authorize"
token_endpoint:                          "https://pragmaticwebsecurity.eu.auth0.com/oauth/token"
userinfo_endpoint:                       "https://pragmaticwebsecurity.eu.auth0.com/userinfo"
mfa_challenge_endpoint:                  "https://pragmaticwebsecurity.eu.auth0.com/mfa/challenge"
jwks_uri:                                "https://pragmaticwebsecurity.eu.auth0.com/.well-known/jwks.json"
registration_endpoint:                   "https://pragmaticwebsecurity.eu.auth0.com/oidc/register"
revocation_endpoint:                     "https://pragmaticwebsecurity.eu.auth0.com/oauth/revoke"
scopes_supported:                        […]
response_types_supported:                […]
response_modes_supported:                […]
subject_types_supported:                 […]
id_token_signing_alg_values_supported:
    0:                                   "HS256"
    1:                                   "RS256"
token_endpoint_auth_methods_supported:
    0:                                   "client_secret_basic"
    1:                                   "client_secret_post"
claims_supported:                        […]
```

K

THE CLIENT'S PERSPECTIVE

THE RESOURCE SERVER'S PERSPECTIVE

INTRODUCING OPENID CONNECT

CONCLUSION

RECAP

- OAuth 2.0 is not about authentication or authorization, but delegation
  - It allows a resource owner to delegate access to a protected resource to a client
  - The authorization server plays a central role in establishing that delegation
- OpenID Connect is about delegating authentication to a third-party
  - OIDC flows result in an identity token containing properties about the authentication
  - OIDC combines with OAuth 2.0, as the same flow can also issue access tokens
- Authorization is the responsibility of the resource server
  - It uses the access token to make authorization decisions on incoming requests
  - Making a proper authorization decision requires careful attention to detail

BEST PRACTICES

- Choose the right flow for the right scenario
  - Do not combine flows, or transfer tokens from one location to another
  - Use the proper flow for your use case, without modifications
- Do not implement custom authentication with OAuth 2.0
  - OpenID Connect went through a lot of trouble to get it right, use it
- Minimize the attack surface following from using OAuth 2.0
  - Use fine-grained scopes to limit the power of an access token
  - Only store the tokens that you need
  - Strengthen the security of token storage through encryption and isolation
  - Use strict ***redirect URIs*** to prevent token stealing attacks
  - Use the ***state*** parameter to prevent CSRF attacks against the flow's integrity

Free security cheat sheets for modern applications

Pragmatic Web Security
Security training for developers

SECURITY CHEAT SHEET
Version 2018.002

Angular and the OWASP top 10

The OWASP top 10 is one of the most influential security documents of all time. But how do these top 10 vulnerabilities resonate in a frontend JavaScript application?

This cheat sheet offers practical advice on handling the most relevant OWASP top 10 vulnerabilities in Angular applications.

DISCLAIMER: This is an opinionated interpretation of the OWASP top 10 (2017), applied to frontend Angular applications. Many backend-related issues apply to the API-side of an Angular application (e.g., SQL Injection), but are out of scope for this cheat sheet. Hence, they are omitted.

1 Using *dependencies* with known vulnerabilities

OWASP #9

- [ ] Plan for a periodical release schedule
- Use `npm audit` to scan for known vulnerabilities
- Setup automated dependency checking to receive alerts
  Github offers automatic dependency checking as a free service
- Integrate dependency checking into your build pipeline

2 Broken authentication

OWASP #2

From an Angular perspective, the most important aspect of broken authentication is maintaining state after authentication. Many alternatives exist, each with their specific security considerations.

- Decide if a stateless backend is a requirement
  Server-side state is more secure, and works well in most cases

Server-side session state

- [ ] Use long and random session identifiers with high entropy
  OWASP has a great cheat sheet offering practical advice [1]

Client-side session state

- [ ] Use signatures to protect the integrity of the session state
- Adopt the proper signature scheme for your deployment
  HMAC-based signatures only work within a single application
  Public/private key signatures work well in distributed scenarios
- [ ] Verify the integrity of inbound state data on the backend
  Explicitly avoid the use of "decode-only" functions in libraries
- [ ] Setup key management / key rotation for your signing keys
- Ensure you can handle session expiration and revocation

Cookie-based session state transport

- [ ] Enable the proper cookie security properties
  *Set the **HttpOnly** and **Secure** cookie attributes*
  *Add the **__Secure** or **__Host-** prefix on the cookie name*
- [ ] Protect the backend against Cross-Site Request Forgery
  *Same-origin APIs should use a **double submit cookie***
  *Cross-Origin APIs should force the use of CORS preflights by only accepting a non-form-based content type (e.g. **application/json**)*

Authorization header-based session state transport

- [ ] Only send the authorization header to whitelisted hosts
  *Many custom interceptors send the header to **every** host*

[1] https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

3 Cross-Site Scripting

OWASP #7

Preventing HTML/script injection in Angular

- [ ] Use interpolation with `{{}}` to automatically apply escaping
- [ ] Use binding to [innerHTML] to safely insert HTML data
- [ ] Do not use `bypassSecurityTrust*()` on untrusted data
  These functions mark data as safe, but do not apply protection

Preventing code injection outside of Angular

- [ ] Avoid direct DOM manipulation
  *E.g. through **ElementRef** or other client-side libraries*
- [ ] Do not combine Angular with server-side dynamic pages
- [ ] Use Ahead-Of-Time compilation (AOT)

4 Broken access control

OWASP #5

Authorization checks

- [ ] Implement proper authorization checks on API endpoints
  Check if the user is authenticated
  Check if the user is allowed to access the specific resources
- [ ] Do not rely on client-side authorization checks for security

Cross-Origin Resource Sharing (CORS)

- [ ] Prevent unauthorized cross-origin access with a strict policy
- [ ] Avoid whitelisting the *null* origin in your policy
- [ ] Avoid blindly reflecting back the value of the origin header
- [ ] Avoid custom CORS implementations
  Origin-matching code is error-prone, so prefer the use of libraries

5 Sensitive data exposure

OWASP #3

Data in transit

- [ ] Serve everything over HTTPS
- [ ] Ensure that all traffic is sent to the HTTPS endpoint
  Redirect HTTP to HTTPS on endpoints dealing with page loads
  Disable HTTP on endpoints that only provide an API
- [ ] Enable *Strict Transport Security* on all HTTPS endpoints

Data at rest in the browser

- [ ] Encrypt sensitive data before persisting it in the browser
- [ ] Encrypt sensitive data in JWTs using JSON Web Encryption

The key to building secure applications is security knowledge
Reach out to learn more about our in-depth training program for developers

Pragmatic Web Security
Security training for developers

SECURITY CHEAT SHEET
Version 2018.001

JSON Web Tokens (JWT)

JSON Web Tokens (JWTs) have become extremely popular. JWTs seem deceivingly simple. However, to ensure their security properties, they depend on complex and often misunderstood concepts. This cheat sheet focuses on the underlying concepts. The cheat sheet covers essential knowledge for every developer producing or consuming JWTs.

Introduction

A JWT is a convenient way to represent claims securely. A claim is nothing more than a key/value pair. One common use case is a set of claims representing the user's identity. The claims are the payload of a JWT. Two other parts are the header and the signature.

- JWTs should always use the appropriate signature scheme
- If a JWT contains sensitive data, it should be encrypted
- JWTs require proper cryptographic key management
- Using JWTs for sessions introduces certain risks

JWT integrity verification

Claims in a JWT are often used for security-sensitive operations. Preventing tampering with previously generated claims is essential. The issuer of a JWT signs the token, allowing the receiver to verify its integrity. These signatures are crucial for security.

Symmetric signatures

Symmetric signatures use an HMAC function. They are easy to setup, but rely on the same secret for generating and verifying signatures. Symmetric signatures only work well within a single application.

Asymmetric signatures

Asymmetric signatures rely on a public/private key pair. The private key is used for signing, and is kept secret. The public key is used for verification, and can be widely known. Asymmetric signatures are ideal for distributed scenarios

Best practices

- [ ] Always verify the signature of JWT tokens
- [ ] Avoid library functions that do not verify signatures
  Example: The decode function of the auth0 Java JWT library
- [ ] Check that the secret of symmetric signatures is not shared
- [ ] A distributed setup should only use asymmetric signatures

JWT Encryption is a complex topic. It is out of scope for this cheat sheet.

Validating JWTs

Apart from the signature, a JWT contains other security properties. These properties help enforce a lifetime on a JWT. They also identify the issuer and the intended target audience. The receiver of a JWT should always check these properties before using any of the claims.

- [ ] Check the `exp` claim to ensure the JWT is not expired
- [ ] Check the `nbf` claim to ensure the JWT can already be used
- [ ] Check the `iss` claim against your list of trusted issuers
- [ ] Check the `aud` claim to see if the JWT is meant for you

Some libraries offer support for checking these properties. Verify which properties are covered, and complement these checks with your own.

Cryptographic key management

The use of keys for signatures and encryption requires careful management. Keys should be stored in a secure location. Keys also need to be rotated frequently. As a result, multiple keys can be in use simultaneously. The application has to foresee a way to manage the JWT key material.

- [ ] Store key material in a dedicated key vault service
  Keys should be fetched dynamically, instead of being hardcoded
- [ ] Use the `kid` claim in the header to identify a specific key
  Keys should be fetched dynamically, instead of being hardcoded
- [ ] Public keys can be embedded in the header of a JWT
  The `jwk` claim can hold a JSON Web Key-formatted public key
  The `x5c` claim can hold a public key and X509-certificate
- [ ] Validate an embedded public key against a whitelist
  Failure to whitelist will cause an attacker's JWT to be accepted
- [ ] The header can also contain a URL pointing to public keys
  The `jku` claim can point to a file containing JSON Web Keys
  The `x5U` claim can point to a certificate containing a public key
- [ ] Validate a key URL against a whitelist of URLs / domains
  Failure to whitelist will cause an attacker's JWT to be accepted

Using JWTs for authorization state

Many modern applications use JWTs to push authorization state to the client. Such an architecture benefits from a stateless backend, often at the cost of security. These JWTs are typically bearer tokens, which can be used or abused by whoever obtains them.

- It is hard to revoke a self-contained JWT before it expires
- [ ] JWTs with authorization data should have a short lifetime
- [ ] Combine short-lived JWTs with a long-lived session

The key to building secure applications is security knowledge
Reach out to learn more about our in-depth training program for developers

https://cheatsheets.pragmaticwebsecurity.com/

Web Security Essentials

2-day training course

Modern-day best practices

Hands-on labs on a custom training application

April 25th – 26th, 2019

Leuven, Belgium

https://essentials.pragmaticwebsecurity.com

Pragmatic Web Security

Security training for developers

/in/PhilippeDeRyck

@PhilippeDeRyck

philippe@pragmaticwebsecurity.com