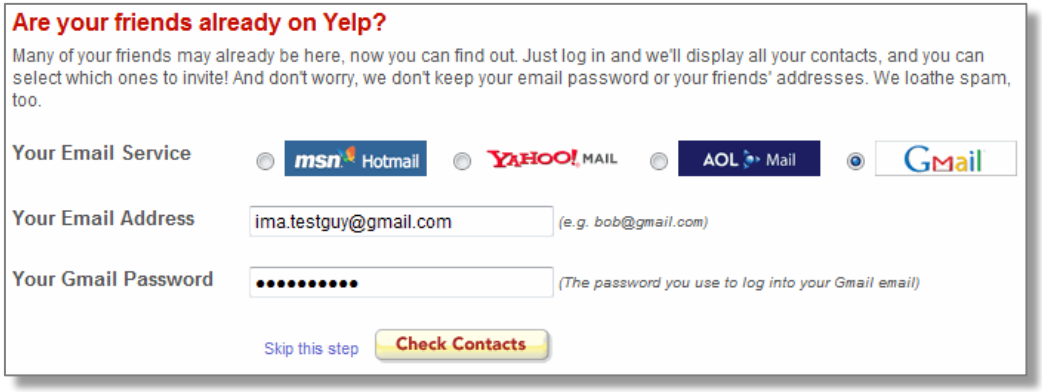
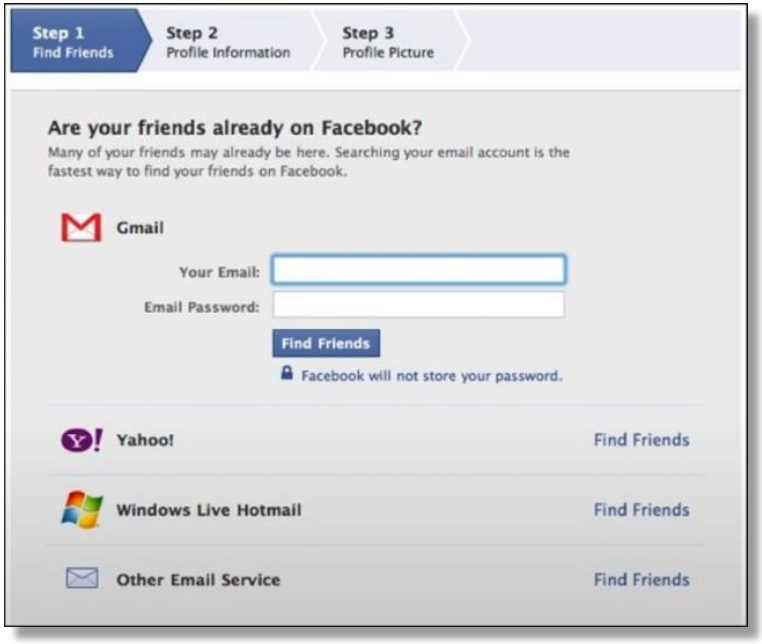
# Exploring OAuth 2.0

## A practical guide to securing your APIs

Pieter Philippaerts

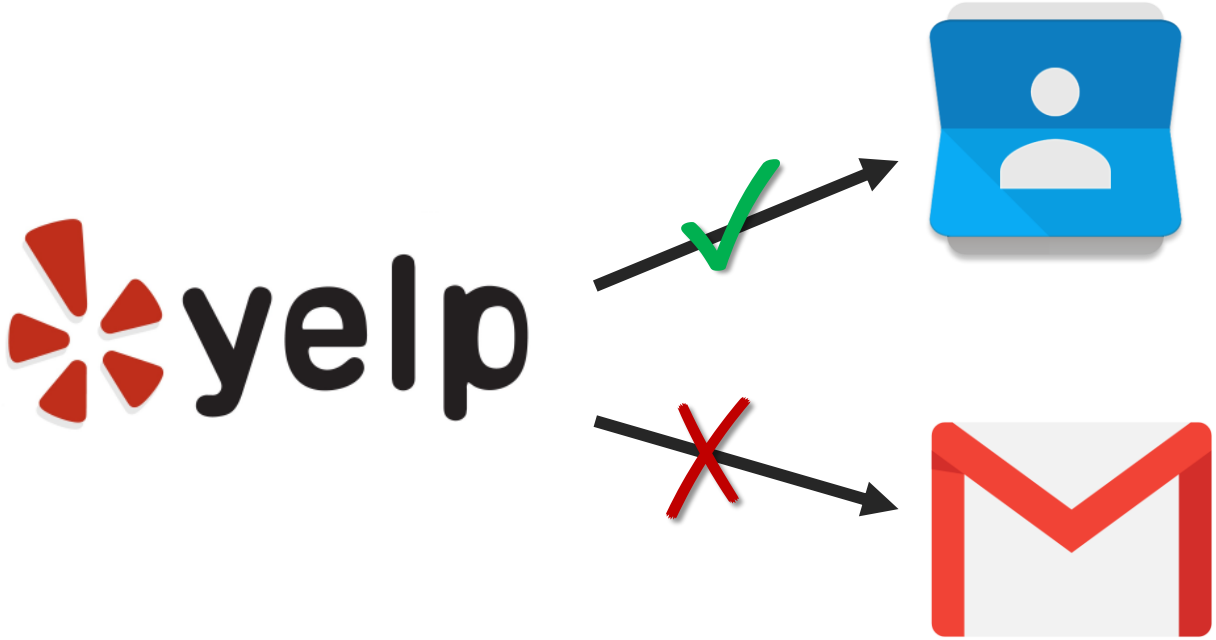

KU LEUVEN DistriNet



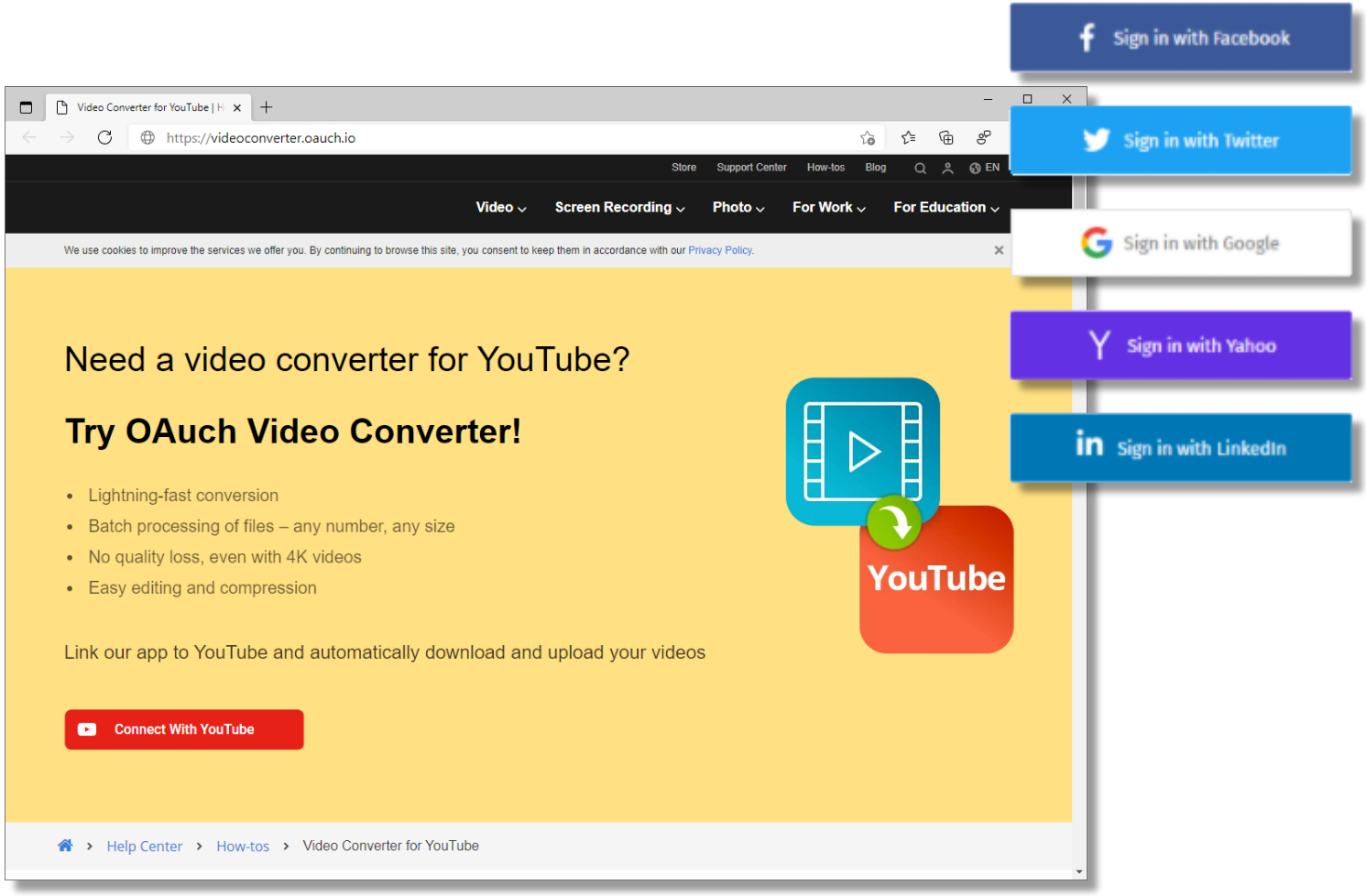



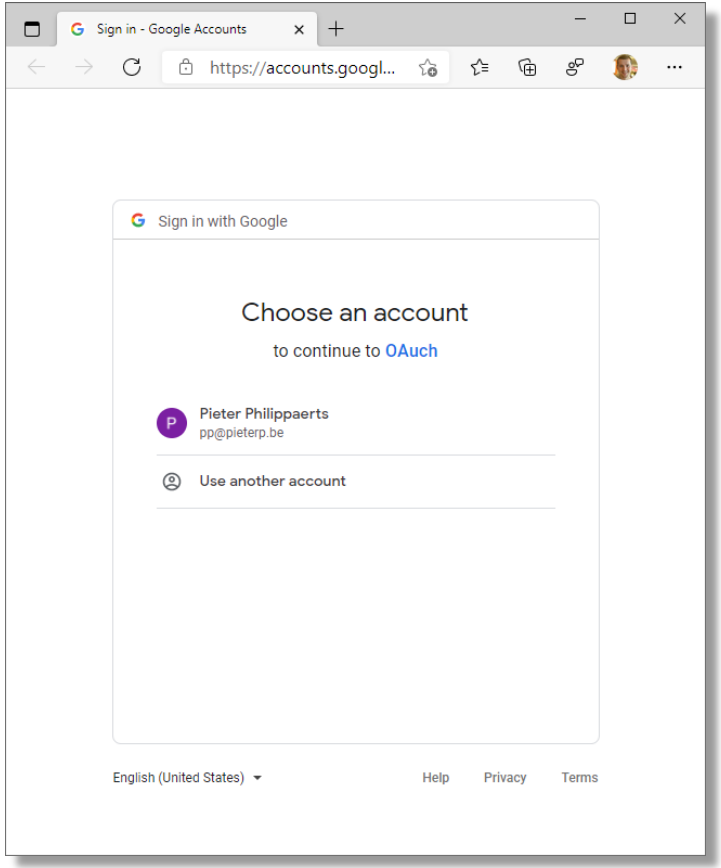

If a third party wanted access to an account,

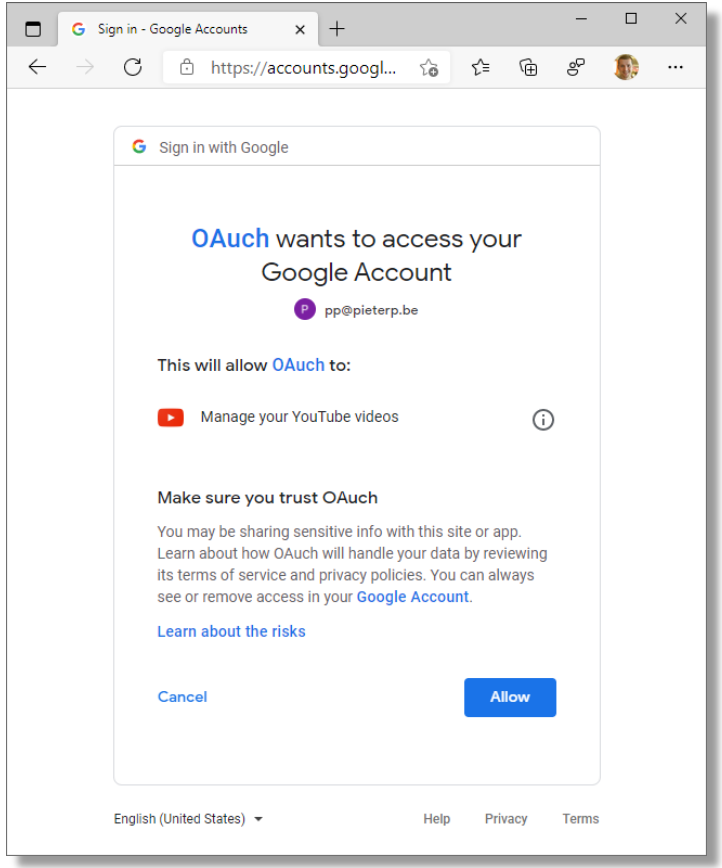
you'de give them your password


yelp

So…
how can I let an app
**access my data**
without giving it my password?


Sign in with Facebook
Sign in with Twitter
Sign in with Google
Sign in with Yahoo
Sign in with LinkedIn
Video Converter for YouTube | H
https://videoconverter.oauch.io
Store
Support Center
How-tos
Blog
EN
Video
Screen Recording
Photo
For Work
For Education
We use cookies to improve the services we offer you. By continuing to browse this site, you consent to keep them in accordance with our Privacy Policy.
Need a video converter for YouTube?
Try OAuch Video Converter!
Lightning-fast conversion
Batch processing of files – any number, any size
No quality loss, even with 4K videos
Easy editing and compression
Link our app to YouTube and automatically download and upload your videos
Connect With YouTube
YouTube
Help Center
How-tos
Video Converter for YouTube


Sign in - Google Accounts
https://accounts.googl...
Sign in with Google
Choose an account
to continue to OAuch
Pieter Philippaerts
pp@pieterp.be
Use another account
English (United States)
Help
Privacy
Terms

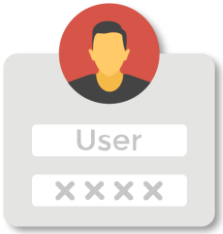

Sign in - Google Accounts
https://accounts.googl...
Sign in with Google
OAuch wants to access your Google Account
pp@pieterp.be
This will allow OAuch to:
Manage your YouTube videos
Make sure you trust OAuch
You may be sharing sensitive info with this site or app. Learn about how OAuch will handle your data by reviewing its terms of service and privacy policies. You can always see or remove access in your Google Account.
Learn about the risks
Cancel
Allow
English (United States)
Help
Privacy
Terms

# Use Cases

Web-server apps

Browser-based apps

Mobile apps




Username/Password access

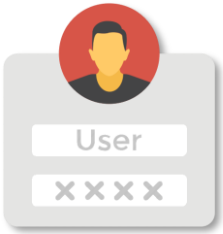
Application access

# Use Cases – Grant Types


**Web-server apps**
`authorization code`

**Browser-based apps**
`implicit`

**Mobile apps**
`implicit`



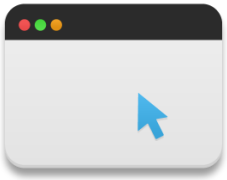

**Username/Password access**
`password`

**Application access**
`client credentials`

# OAuth 2.0 Roles

**Resource Owner**
“the user”

**User-Agent**
“the browser”



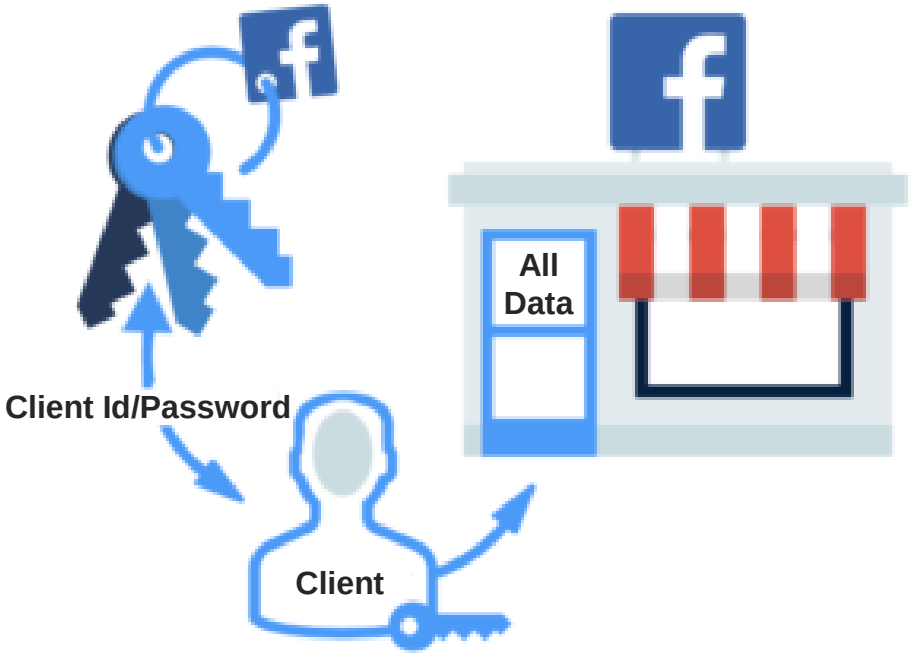
**Client**
“the app”


**Resource Server**
“the API”

**Authorization Server**

# OAuth 2.0 Grant Types

# Client Credentials Grant




https://blog.oauth.io/oauth2-flow-grant-types-in-pictures/

# Client Credentials Grant

**REQUEST**

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
```

Client ID & Password

**RESPONSE**

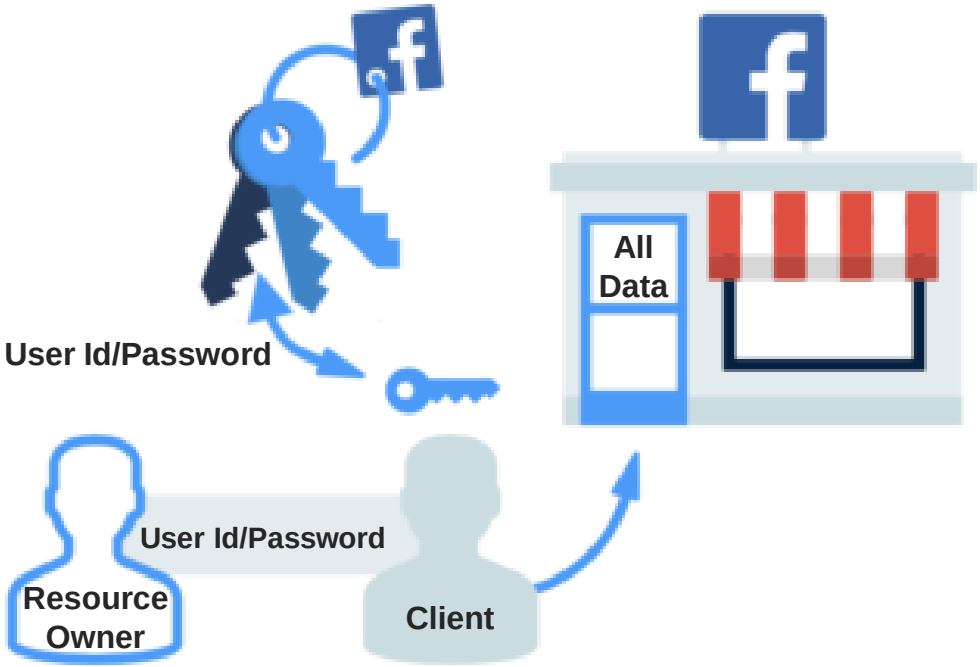
```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"Bearer",
  "expires_in":3600
}
```

# Client Credentials Grant

- Easy ✓
- Secure ✓
- Wide use case support ✗


# Use the Client Credentials flow for Machine-to-machine authorization

# Password Grant




https://blog.oauth.io/oauth2-flow-grant-types-in-pictures/

# Password Grant

**REQUEST**

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=johndoe&password=A3ddj3w
```

Client ID & Password

Resource Owner Username & Password

**RESPONSE**

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
}
```

# Password Grant

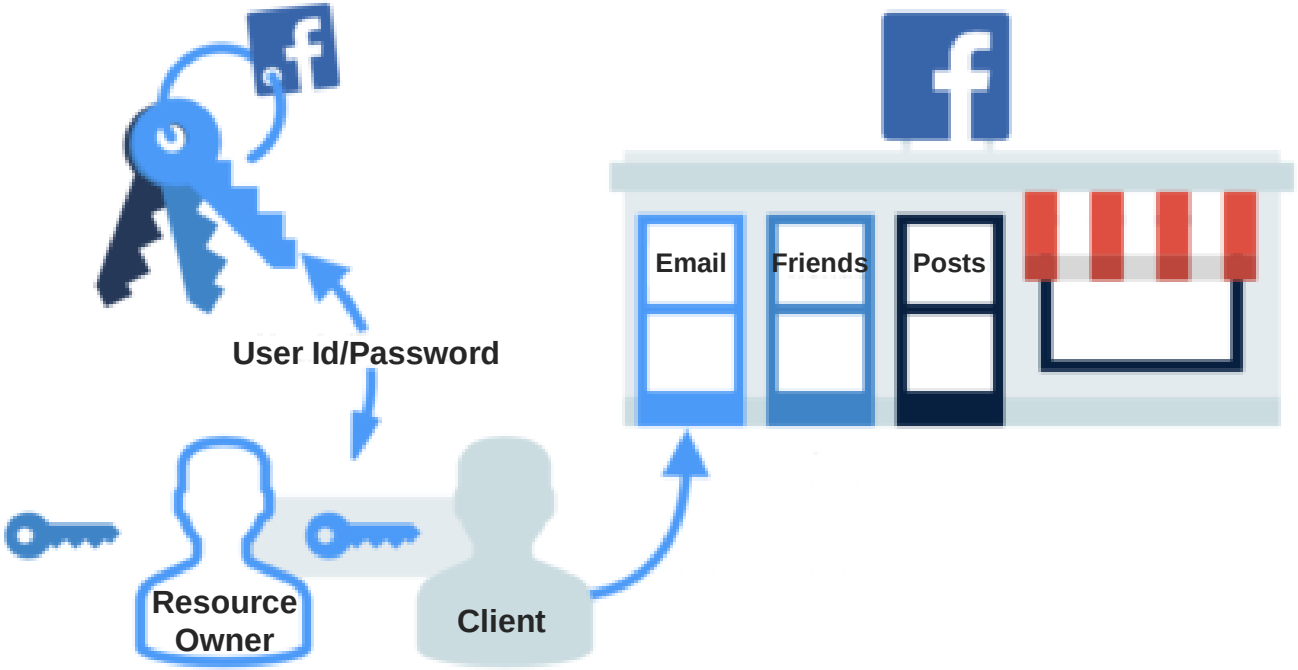
- Easy ✓
- Wide use case support ✓
- Secure ✗

# Password Grant Threats

- Threat #1: Exposes the username and password
- Threat #2: No mechanism to limit scope
- Threat #3: Trains users that it's okay to enter password in more than one place
- Threat #4: Difficult (or impossible) to add multifactor or passwordless authentication (WebCrypto, WebAuthn)

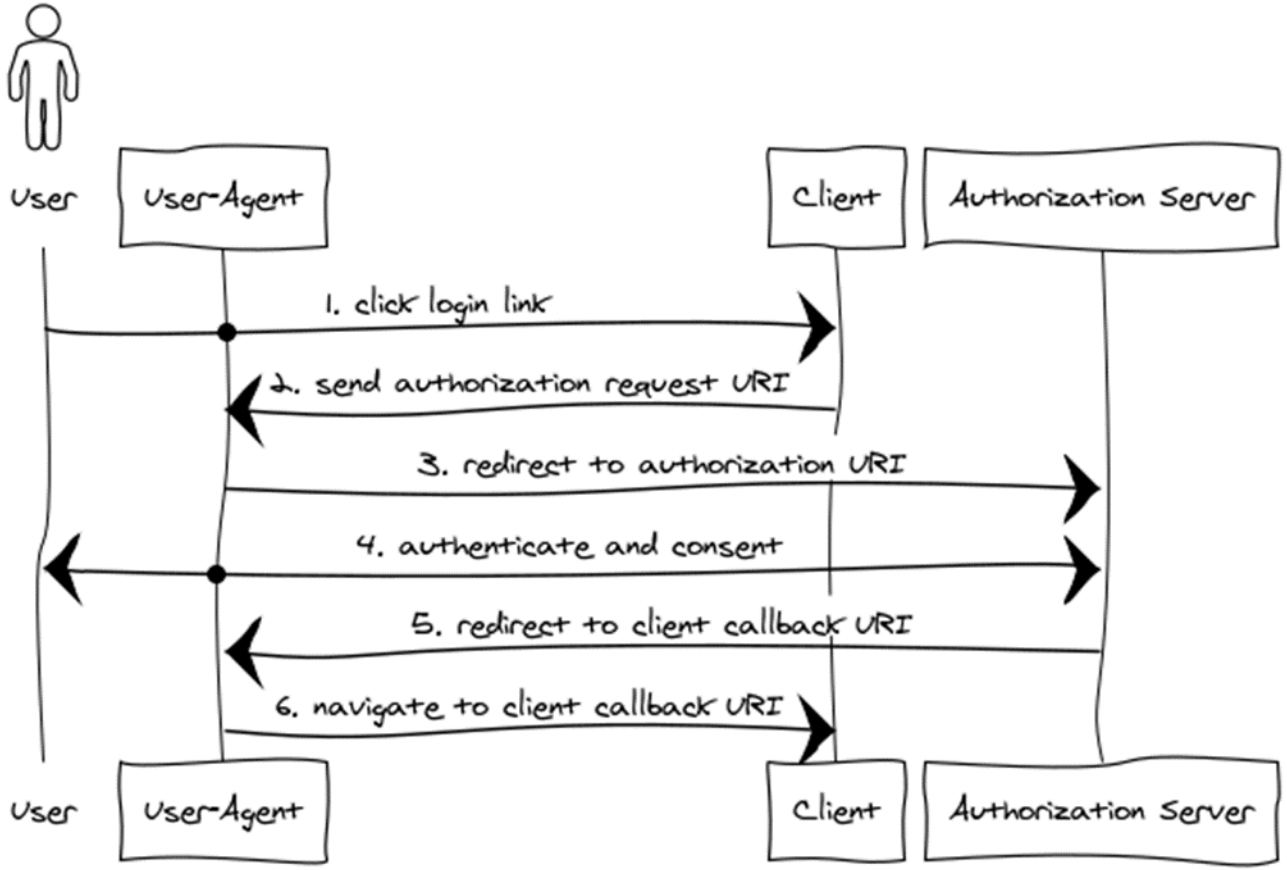

# Do not use the Password grant

# Implicit Grant




https://blog.oauth.io/oauth2-flow-grant-types-in-pictures/

# Implicit Grant




Created with websequencediagrams.com

# Implicit Grant

**REQUEST**

```
GET /authorize?response_type=token&client_id=s6BhdRkqt3&state=xyz
      &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

- response_type **token**
- client_id **s6BhdRkqt3**
- state **xyz**
- redirect_uri **https://client.example.com/cb**

# Implicit Grant

**REQUEST**

```
GET /authorize?response_type=token&client_id=s6BhdRkqt3&state=xyz
      &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

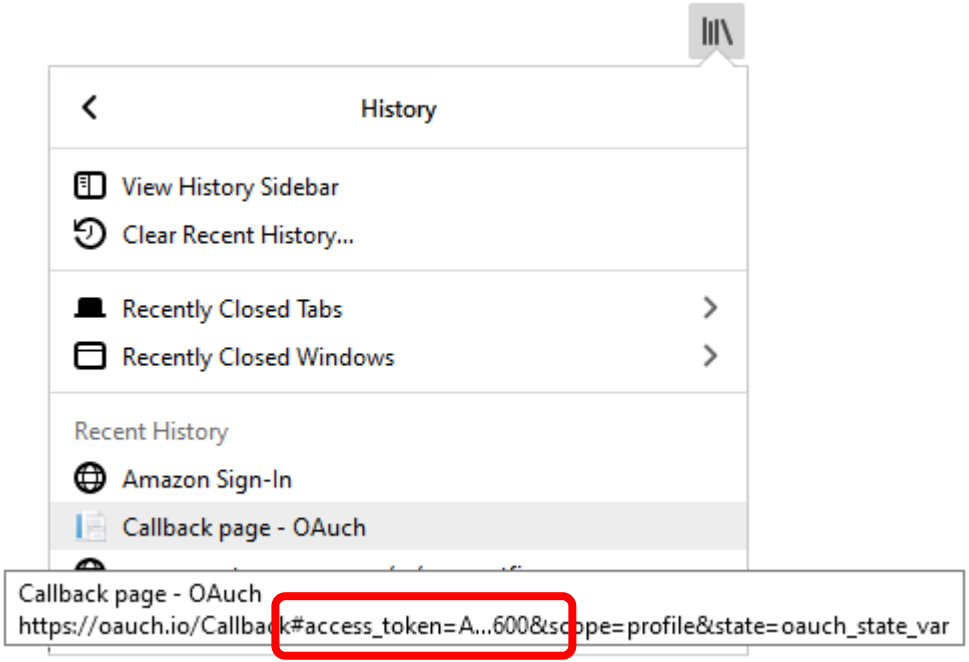
**RESPONSE**

```
HTTP/1.1 302 Found
Location: http://example.com/cb#access_token=2YotnFZFEjr1zCsicMWpAA
          &state=xyz&token_type=bearer&expires_in=3600
```

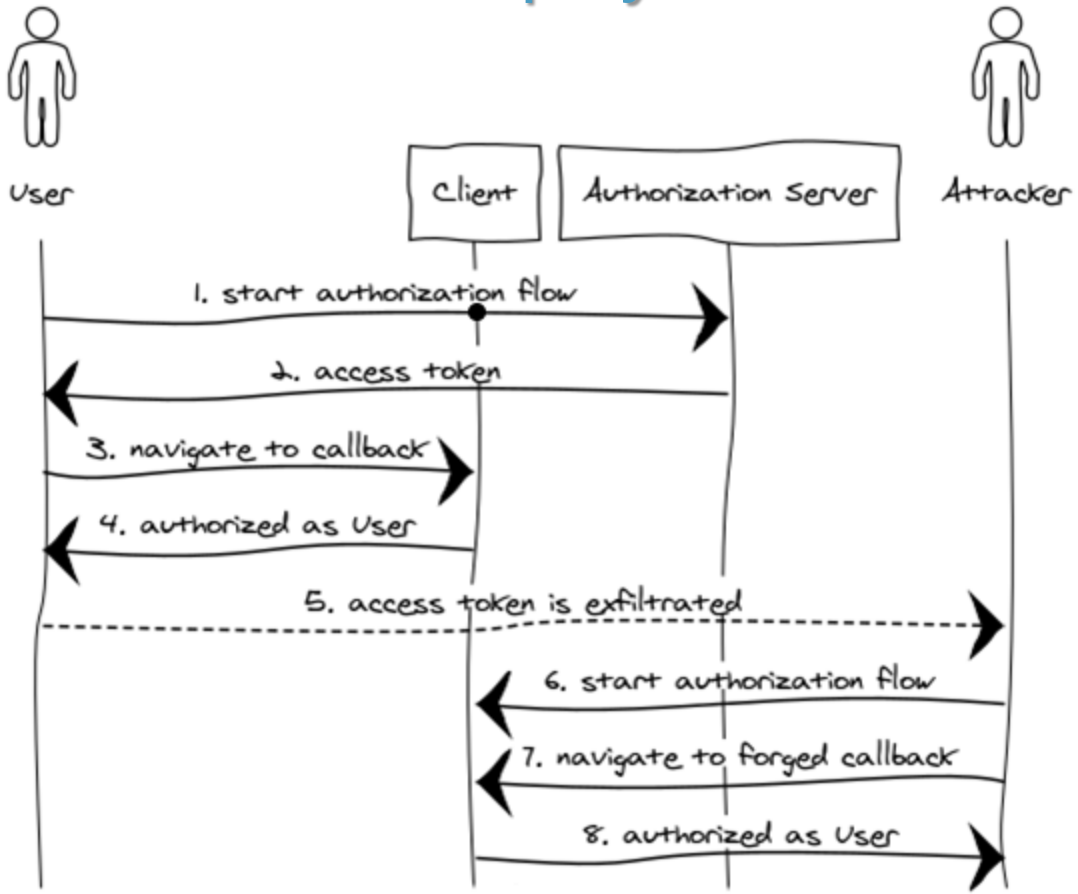
# Implicit Grant

- Easy ?
- Wide use case support ✓
- Secure
  - Username and password are not exposed ✓
  - Scope can be limited ✓
  - User always uses official authorization page ✓
  - Possible to add multi-factor or passwordless authentication ✓
  - But…

# Threat #1: Access token leakage



# Threat #2: Access token replay



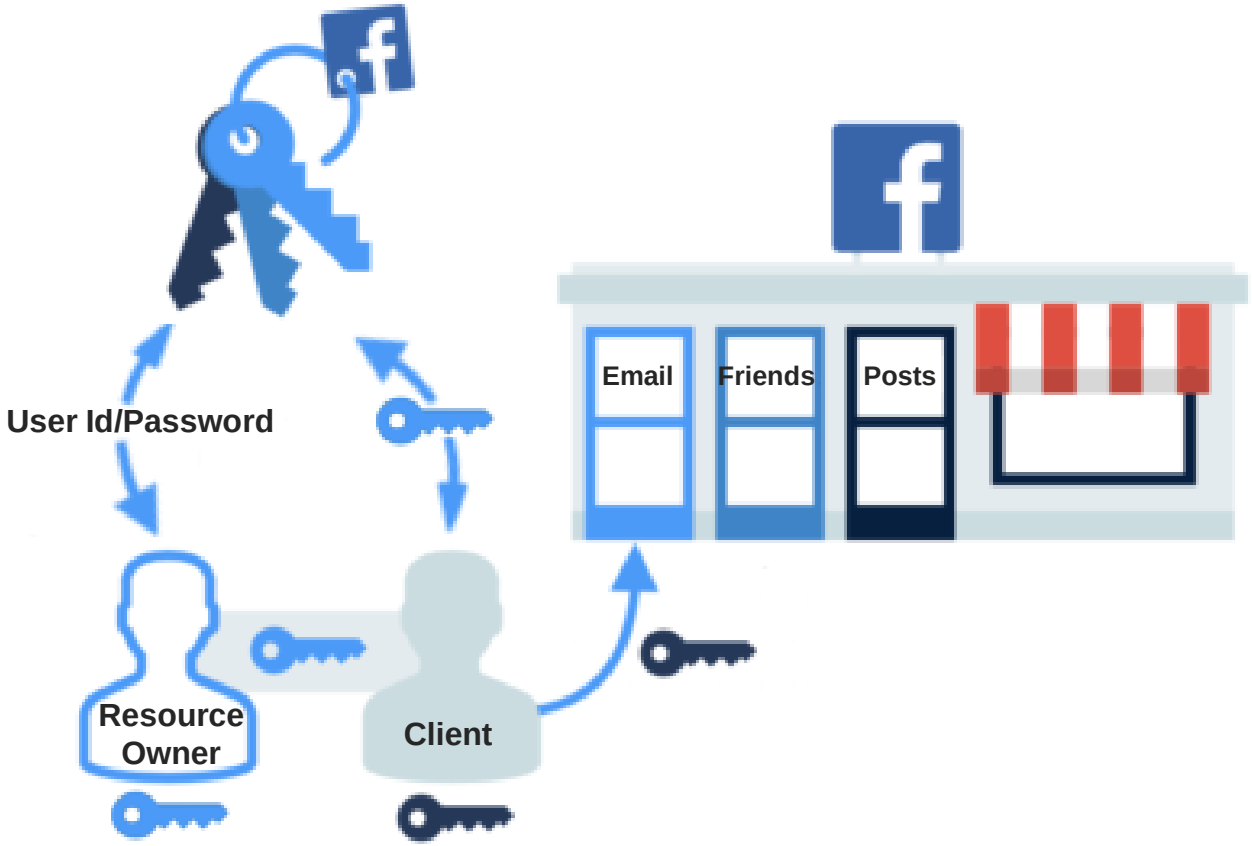

Created with websequencediagrams.com

# Additional Shortcoming

- Tokens cannot be (cryptographically) bound to a client
  - Clients are not authenticated

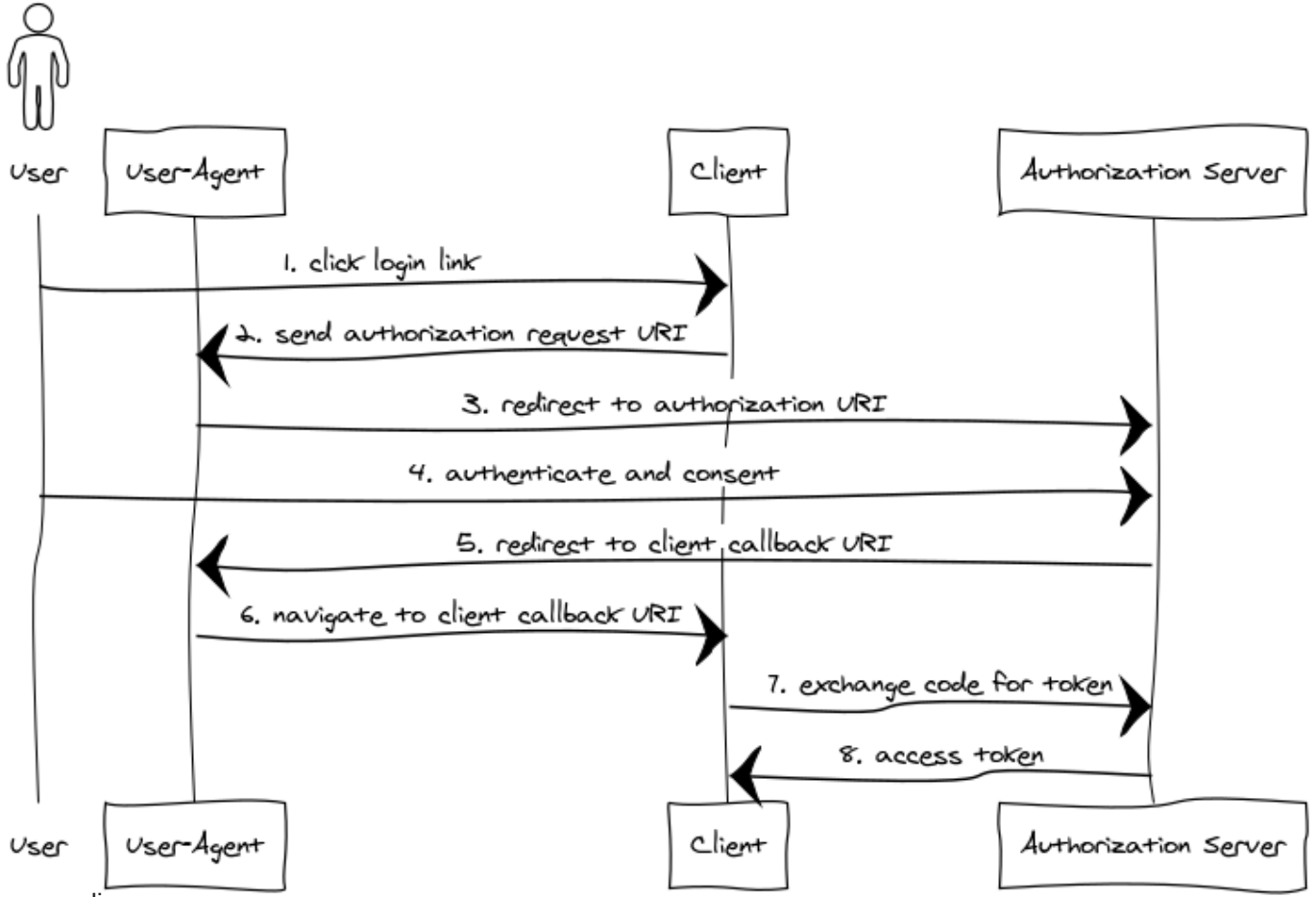

# Do not use the Implicit grant

# Authorization Code Grant




https://blog.oauth.io/oauth2-flow-grant-types-in-pictures/

# Authorization Code Grant




Created with websequencediagrams.com

# Authorization Code Grant

**REQUEST**

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

**RESPONSE**

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
&state=xyz
```

# Authorization Code Grant

**REQUEST**

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

**RESPONSE**

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"bearer",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
}
```

# Authorization Code Grant

- Easy ✗
- Wide use case support ✓
- Secure
  - All the benefits of the implicit flow ✓
  - Access tokens are not leaked ✓
  - Authorization codes cannot be replayed ✓
  - But…

# Threat #1: Insufficient Redirect URI Validation

- Some implementations allow redirect URI patterns
  - `https://*.benign.site/*`
  - Matches with `https://attacker.site/.benign.site/`

**REQUEST**

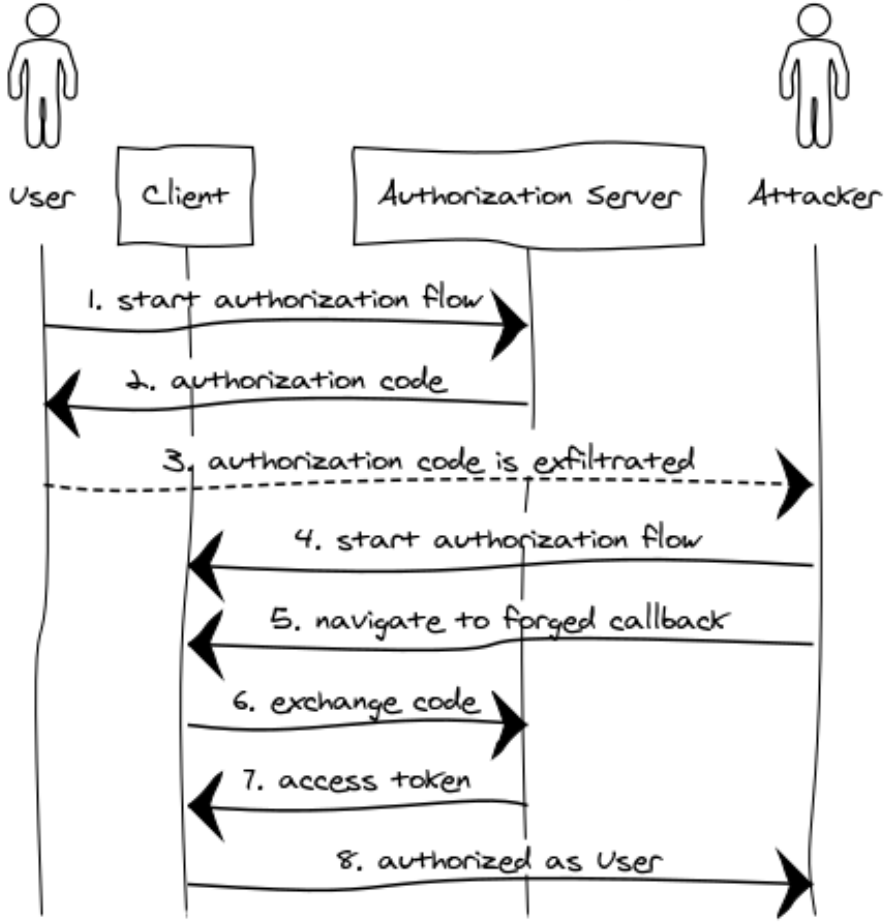
```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=9ad67f13
     &redirect_uri=https%3A%2F%2Fattacker.site%2F.benign.site%2F
     HTTP/1.1
Host: server.somesite.example
```

# Threat #1: Insufficient Redirect URI Validation

- Other problems exist (e.g. open redirectors, …)

- Always exactly match Redirect URIs with the registered values

# Threat #2: Authorization Code Injection



# 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

# Proof Key for Code Exchange (PKCE)

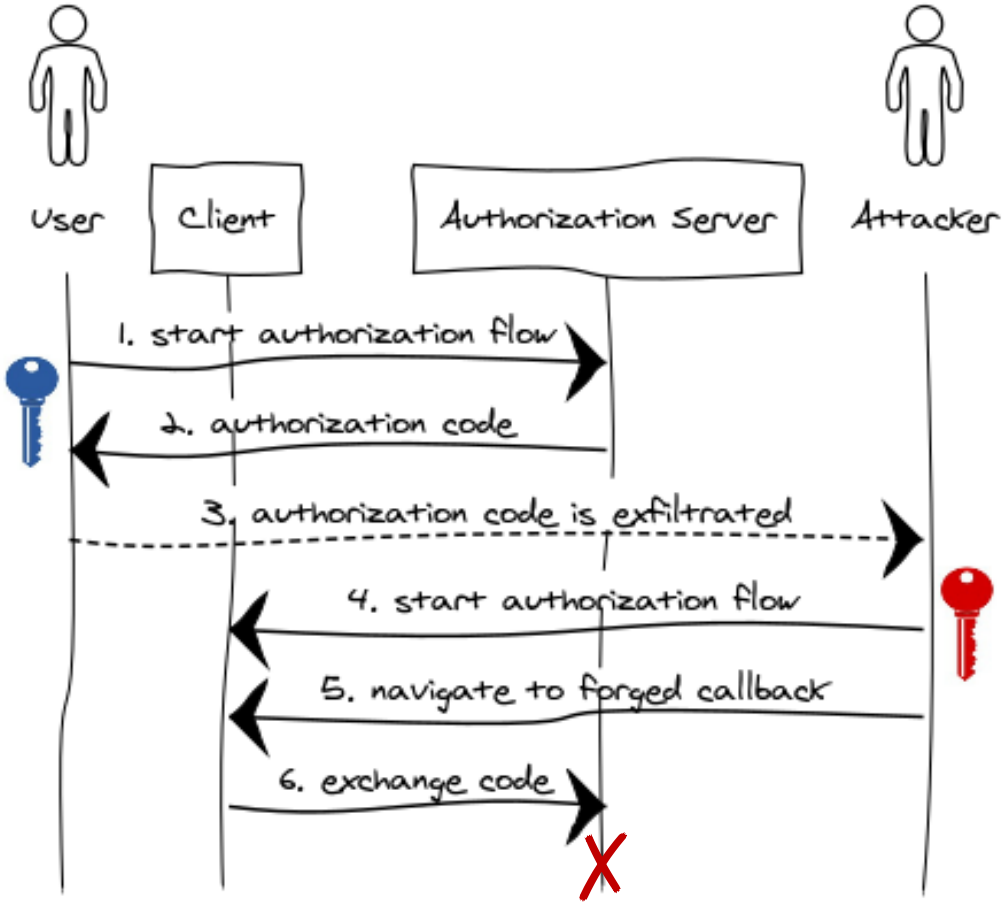
**REQUEST**

```
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
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
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
```

# Threat: Authorization Code Injection



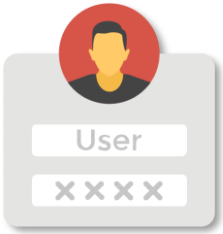

# Use Authorization Code grant + PKCE when a user is involved

# Use Cases – Grant Types

**Web-server apps**
`authorization code + PKCE`

**Browser-based apps**
~~`implicit`~~ `authorization code + PKCE`

**Mobile apps**
~~`implicit`~~ `authorization code + PKCE`




~~**Username/Password access**~~
~~`password`~~

**Application access**
`client credentials`

# More Best Practices


- Clients *should* use Mutual TLS
  - Replaces *client secret*
  - Enables sender-constrained tokens

# More Best Practices


- Clients *must not* pass access tokens in a URI query parameter
  - `https://myapi.com/posts/all?access_token=avGt23F8fWb`

# More Best Practices


- Refresh tokens must either be sender-constrained or one-time use
  - Use refresh token rotation

# Conclusion

- OAuth 2.0 is about **delegation**
  - Clients can ask permission to access protected resources on a resource owner's (user's) behalf
- OAuth 2.0 is a secure protocol ***if used correctly***
  - Most servers and clients do not follow the best practices


# Thank you!

https://distrinet.cs.kuleuven.be/

Pieter.Philippaerts@kuleuven.be