

# Middleware Architectures 2

## Lecture 5: Security

**doc. Ing. Tomáš Vitvar, Ph.D.**

tomas@vitvar.com • @TomasVitvar • <https://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <https://vitvar.com/lectures>



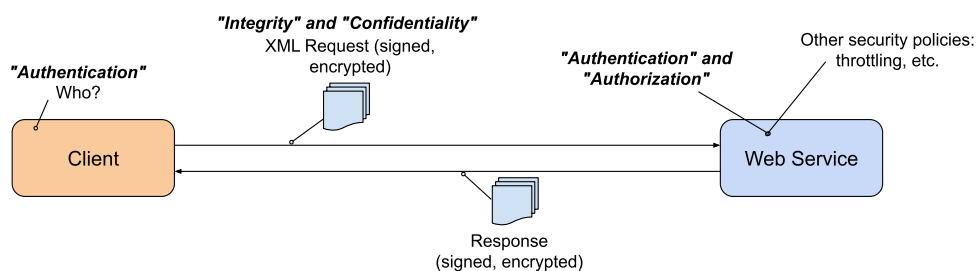
Modified: Mon May 01 2023, 21:46:41  
Humla v1.0

## Overview

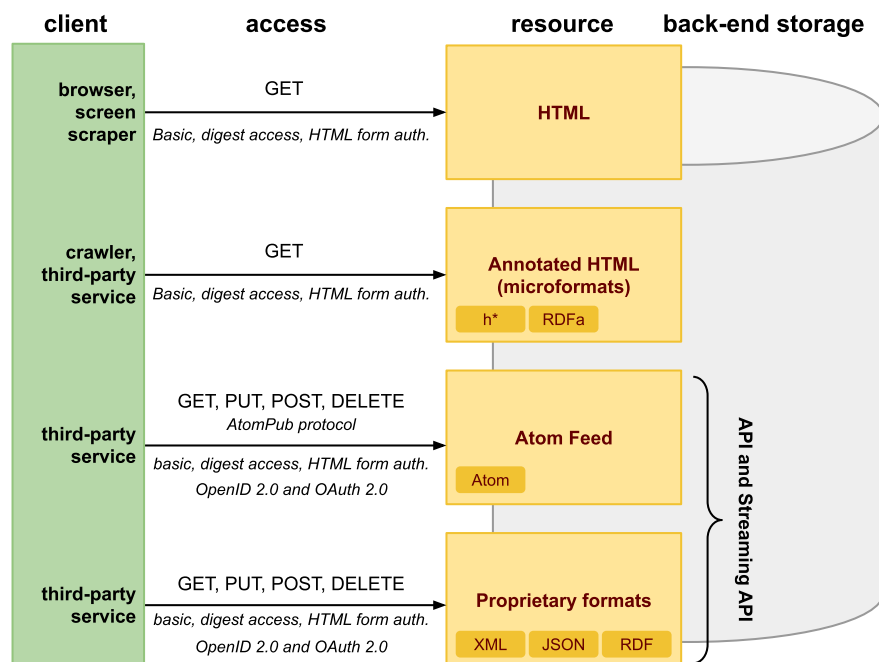
- **Security Concepts**
- Transport Level Security
- JSON Web Token
- OAuth 2.0
- OpenID

## Web Service Security Concepts

- Securing the client-server communication
  - *Message-level security*
  - *Transport-level security*
- Ensure
  - *Authentication* – *verify a client's identity*
  - *Authorization* – *rights to access resources*
  - *Message Confidentiality* – *keep message content secret*
  - *Message Integrity* – *message content does not change during transmission*
  - *Non-repudiation* – *proof of integrity and origin of data*



## Data on the Web



# Authentication and Authorization

- Authentication
  - *verification of user's identity*
- Authorization
  - *verification that a user has rights to access a resource*
- Standard: HTTP authentication
  - *HTTP defines two options*
    - *Basic Access Authentication*
    - *Digest Access Authentication*
  - *They are defined in*
    - *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*
    - *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication*
- Custom/proprietary: use of cookies

## Basic Access Authentication



- Realm
  - *an identifier of the space on the server (~ a collection of resources and their sub-resources)*
  - *A client may associate a valid credentials with realms such that it copies authorization information in requests for which server requires authentication (by **WWW-Authenticate** header)*

## Basic Access Authentication – Credentials

- Credentials

- *credentials are base64 encoded*
- *the format is: **username:password***

```
1 # to encode in linux
2 echo "novak:heslo" | base64
3 > bm92YWs6aGVzbG8K
4
5 # and to decode
6 echo "bm92YWs6aGVzbG8K" | base64 -d # use capital "D" in OS X
7 > novak:heslo
```

- Comments

- *When TLS is not used, the password can be read*
- *An attacker can repeat interactions*

## Digest Access Authentication

- RFC 2617 – Basic and Digest Access Authentication

- *No password between a client and a server but a hash value*
- *Simple and advanced mechanisms (only server-generated nonce value – replay-attacks or with client-generated nonce value)*

- Basic Steps

1. *Client accesses a protected area*

```
1 | > GET / HTTP/1.1
```

2. *Server requests authentication with **WWW-Authenticate***

```
1 < HTTP/1.1 401 Unauthorized
2 < WWW-Authenticate: Digest realm="ProtectedArea",
3   nonce="BbdQof3DBAA=a293ff3d724989371610f03015f2d23f3cd2c045",
4   algorithm=MD5, domain="/", qop="auth"
```

3. *Client calculates a response hash by using the realm, his/her username, the password, and the quality of protection (QoP) and requests the resource with **authorization** header*

```
1 > GET / HTTP/1.1
2 > Authorization: Digest username="novak", realm="ProtectedArea",
3   nonce="BbdQof3DBAA=a293ff3d724989371610f03015f2d23f3cd2c045", uri="/",
4   algorithm=MD5, response="c4ea2293aeb318826d1e533f363efd90", qop=auth,
5   nc=00000001, cnonce="531ee8ba7f2a8fd1"
```

## Overview

- Security Concepts
- **Transport Level Security**
- JSON Web Token
- OAuth 2.0
- OpenID

## Overview

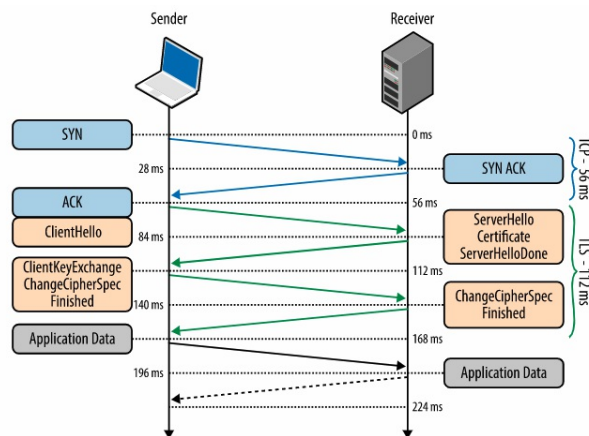
- SSL and TLS
  - *SSL and TLS is used interchangeably*
  - *SSL 3.0 developed by Netscape*
  - *IETF standardization of SSL 3.0 is TLS 1.0*
    - *TLS 1.0 is upgrade of SSL 3.0*
  - *Due to security flaws in TLS 1.0, TLS 1.1 and TLS 1.2 were created*
- TLS layer



# TLS Services

- Encryption
  - Peers must agree on ciphersuite and keys
  - This is achieved by **TLS handshake**
- Authentication
  - Peers can authenticate their identity
    - The client can verify that the server is who it is claimed to be
    - Achieved by "Chain of Trust and Certificate Authorities"
    - The server can also verify the client
- Integrity
  - TLS provides message framing mechanism
  - Every message is signed with Message Authentication Code (MAC)
    - MAC hashes data in a message and combines the resulting hash with a key (negotiated during the TLS handshake)
    - The result is a message authentication code sent with the message

# TLS Handshake Protocol



- TLS Handshake
  - 56 ms: ClientHello, TLS protocol version, list of ciphersuites, TLS options
  - 84 ms: ServerHello, TLS protocol version, ciphersuite, certificate
  - 112 ms: RSA or Diffie-Hellman key exchange
  - 140 ms: Message integrity checks, sends encrypted "Finished" message
  - 168 ms: Decrypts the message, app data can be sent

## Key Exchange

- RSA key exchange(Rivest–Shamir–Adleman)
  - The client generates a symmetric key
  - The client encrypts the key with the server's public key
  - The client sends the encrypted key to the server
  - The server uses its private key to decrypt the symmetric key
- RSA critical weakness
  - The same public-private key pair is used to:
    - authenticate the server (the server's private key is used to sign and verify the handshake)
    - encrypt the symmetric key
  - When an attacker gets hold of the server private key
    - It can decrypt the entire session
- Diffie-Hellman key exchange
  - Client and server can negotiate shared secret without its explicit communication
    - Attacker cannot get the key
  - Reduction of risk of compromising of the past communications
    - New key can be generated as part of every key exchange
    - Old keys can be discarded

## TLS and Proxy Servers

- TLS Offloading
  - Inbound TLS connection, plain outbound connection
  - Proxy can inspect messages
- TLS Bridging
  - Inbound TLS connection, new outbound TLS connection
  - Proxy can inspect messages
- End-to-End TLS (TLS pass-through)
  - TLS connection is passed-through the proxy
  - Proxy cannot inspect messages
- Load balancer
  - Can use TLS offloading or TLS bridging
  - Can use TLS pass-through with help of Server Name Indication (SNI)

## Overview

- Security Concepts
- Transport Level Security
- **JSON Web Token**
- OAuth 2.0
- OpenID

## Overview

- JSON Web Token (JWT)
  - *Open standard (RFC 7519)*
  - *Mechanism to securely transmit information between parties as a JSON object.*
  - *Can be **verified** and **trusted** as it is **digitally signed**.*
- Basic concepts
  - *Compact*
    - *has a small size*
    - *can be transmitted via a URL, POST, HTTP header.*
  - *Self-contained*
    - *payload contains all required user information.*



## Use of JWT

- Authentication
  - After user logs in, following requests contain JWT token.
  - Single Sign On widely uses JWT nowadays
- Information Exchange
  - Signature ensures senders are who they say they are.
  - Message integrity – signature calculated using the header and the payload.

## JWT Structure

**<header>.<payload>.<signature>**

- Header
  - Contains two parts, the type of the token (JWT) and the hashing algorithm being used (e.g. HMAC, SHA256, RSA).

```
{  "alg": "HS256",  "typ": "JWT"}
```
- Payload
  - Contains the claims, i.e. statements about an entity (e.g. user).
  - Can be registered, public and private
  - Registered and public should be defined in [IANA JSON Web Token Registry](#)

```
{  "sub": "1234567890",  "name": "John Doe",  "admin": true}
```

## JWT Structure (Cont.)

- Signature

- Signed encoded header, encoded payload and a secret.
- For example, signature using HMAC SHA256 algorithm

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret)
```

- Example

- JWT is a three Base64-URL strings separated by dots

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
rRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

## How to use JWT



1. User sends username and password
2. Server verifies user, creates JWT token with a secret and a expiration time
3. Server sends JWT token back to the Browser
4. Browser sends JWT token on subsequent iterations

### Notes

- Authorization header does not behave the same as cookies!
- JWT should not contain secrets (passwords) as it can be read (on the client or if non-https connection is used)

## Overview

- Security Concepts
- Transport Level Security
- JSON Web Token
- **OAuth 2.0**
  - *Client-side Web Apps*
  - *Server-side Web Apps*
- OpenID

## Motivation

- Cloud Computing – Software as a Service
  - *Users utilize apps in clouds*
    - *they access **resources** via Web browsers*
    - *they store their data in the cloud*
    - *Google Docs, Contacts, etc.*
  - *The trend is that SaaS are open*
    - *can be extended by 3rd-party developers through APIs*
    - *attract more users ⇒ increases value of apps*
  - *Apps extensions need to have an access to users' data*
- Need for a new mechanism to access resources
  - *Users can grant access to third-party apps without exposing their users' credentials*

## When there is no OAuth



- Users must share their credentials with the 3rd-party app
- Users cannot control what and how long the app can access resources
- Users must trust the app
  - In case of misuse, users can only change their passwords

## OAuth 2.0 Protocol

- OAuth Objectives
  - users can grant access to third-party applications
  - users can revoke access any time
  - supports:
    - client-side web apps (implicit grant),
    - server-side apps (authorization code), and
    - native (desktop) apps (authorization code)
- History
  - Initiated by Google, Twitter, Yahoo!
  - Different, non-standard protocols first: ClientLogin, AuthSub
  - OAuth 1.0 – first standard, security problems, quite complex
  - OAuth 2.0 – new version, not backward compatible with 1.0
- Specifications and adoption
  - OAuth 2.0 Protocol [🔗](#)
  - OAuth 2.0 Google Support [🔗](#)

## Terminology

- **Client**
  - a *third-party app* accessing resources owned by **resource owner**
- **Resource Owner** (also user)
  - a person that owns a resource stored in the **resource server**
- **Authorization and Token Endpoints**
  - endpoints provided by an **authorization server** through which a **resource owner** authorizes requests.
- **Resource Server**
  - an app that stores resources owned by a **resource owner**
  - For example, contacts in Google Contacts
- **Authorization Code**
  - a code that a **client** uses to request **access tokens** to access resources
- **Access Token**
  - a code that a **client** uses to access resources

## Overview

- Security Concepts
- Transport Level Security
- JSON Web Token
- OAuth 2.0
  - *Client-side Web Apps*
  - *Server-side Web Apps*
- OpenID

## Client-side Web Apps

- Simplified version of OAuth 2.0 protocol
  - JavaScript/AJAX apps running in a browser
  - Apps that cannot easily "remember" app state
  - limited number of interactions
- Architecture
  - User-agent processes a javascript/HTML code from the client
  - No need of authorization code
- Basic Steps
  - A client redirects a user agent to the authorization endpoint
  - A resource owner grants an access to the client
    - or he/she rejects the request
  - Authorization server provides an **access\_token** to the client
  - Client access the resource with the **access\_token**
  - When the token expires, client requests a new token

## Demo – List of Contacts

- Display your Google contacts
  - this demo requests authorization from you to access your Google contacts using client-side OAuth 2.0 protocol and then displays the contacts below. In order to transfer **access\_token** from authorization window, it stores the **access\_token** in a cookie.
  - **access\_token**
  - *Show contacts or revoke access*

# Client-side Web Apps Protocol



## Redirection – Step 1

- **Methods and Parameters**
  - **Methods:** GET or POST
  - **example authorization endpoint url (Google):**  
`https://accounts.google.com/o/oauth2/auth`
  - **query string parameters or application/x-www-form-urlencoded**
    - **client\_id** – id of the client that was previously registered
    - **redirect\_uri** – an URI that auth. server will redirect to when user grants/rejects
    - **scope** – string identifying resources/services to be accessed
    - **response\_type** – type of the response (token or code)
    - **state** (optional) – state between request and redirect
  - **Example**

```

1 | https://accounts.google.com/o/oauth2/auth?
2 | client_id=621535099260.apps.googleusercontent.com&
3 | redirect_uri=http://w20.vitvar.com/examples/oauth/callback.html&
4 | scope=https://www.google.com/m8/feeds&
5 | response_type=token
                
```

## Callback – steps 4 and 5

- Resource owner grants the access
  - *authorization server calls back **redirect\_uri***
  - *client parses URL in JavaScript (Step 5)*
    - *extracts **access\_token** and **expires\_in** (by using **window.location.hash**)*
  - *Example:*

```
1 | https://w20.vitvar.com/examples/oauth/callback.html#
2 | access_token=1/QbZfgDNsnd&
3 | expires_in=4301
```
- Resource owner rejects the access
  - *authorization server calls back **redirect\_uri** with query string parameter **error=access\_denied***
  - *Example:*

```
1 | http://w20.vitvar.com/examples/oauth/callback.html?
2 | error=access_denied
```

## Accessing Resources – Step 6

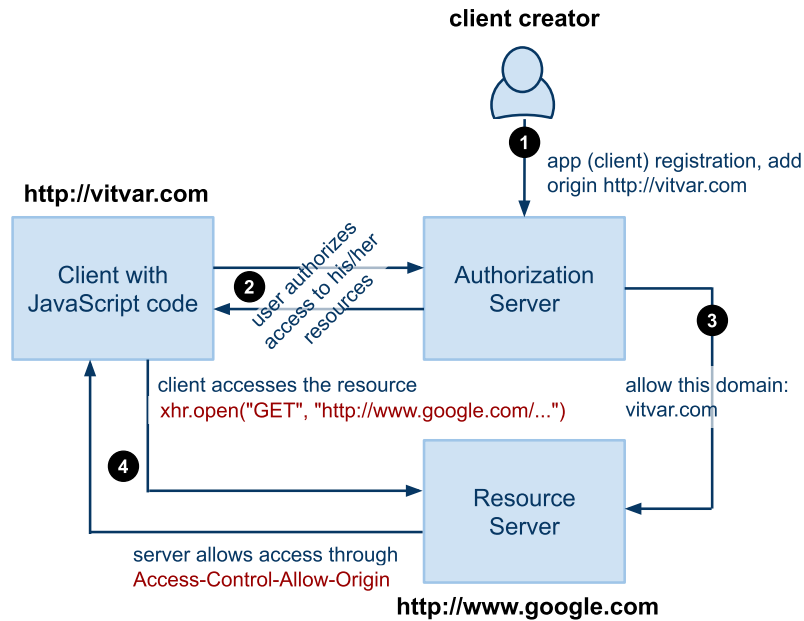
- Request
  - *client can access resources defined by **scope***
  - *resources' URIs defined in a particular documentation*
  - *Example Google Contacts*
    - *to access all users' contacts stored in Google*
    - ***scope** is **https://www.google.com/m8/feeds***
  - *Query string parameter **oauth\_token***

```
1 | curl https://www.google.com/m8/feeds/contacts/default/full?
2 |   oauth_token=1/dERFd34Sf
```
  - *HTTP Header **Authorization***

```
1 | curl -H "Authorization: OAuth 1/dERFd34Sf"
2 |   https://www.google.com/m8/feeds/contacts/default/full
```
  - *The client can do any allowed operations on the resource*
- Response
  - *Success – **200 OK***
  - *Error – **401 Unauthorized** when token expires or the client hasn't performed the authorization request.*



# Cross-Origin Resource Sharing



— see *Same Origin and Cross-Origin* for details

# Example Application Registration

The screenshot shows the Google APIs console interface for the application "vitvar.com". The left sidebar contains navigation links: Overview, Services, Team, API Access (selected), Billing, Reports, and Quotas.

**API Access**

To prevent abuse, Google places limits on API requests. Using a valid OAuth token or API key allows you to exceed anonymous limits by connecting requests back to your project.

**Authorized API Access**

OAuth allows users to share specific data with you (for example, contact lists) while keeping their usernames, passwords, and other information private. [Learn more](#)

**Branding information**

The following information is shown to users whenever you request access to their private data.

Product name: w20-test  
Google account: t.vitvar@gmail.com

[Edit branding information...](#)

**Client ID for web applications**

Client ID:	621535099260.apps.googleusercontent.com	<a href="#">Edit settings...</a>
Client secret:	RxWM917Sv-7cyfWMW7KhNV9R	<a href="#">Reset client secret...</a>
Redirect URIs:	<code>http://vitvar.com/examples/oauth/callback.html</code>	
JavaScript origins:	<code>http://example.org</code>	

[Create another client ID...](#)

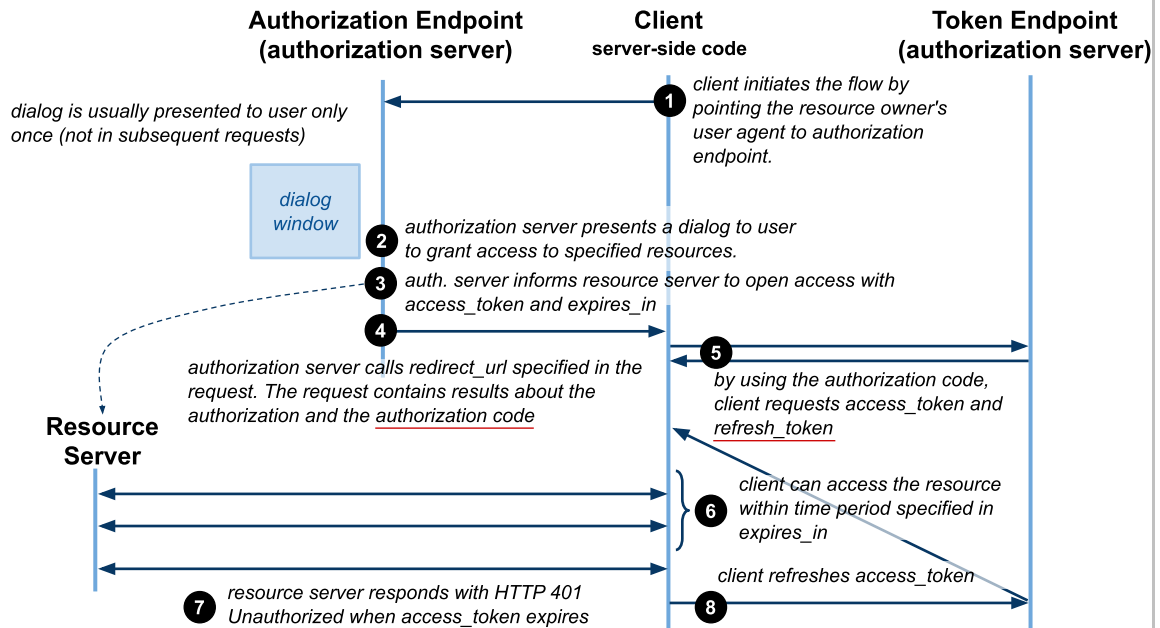
## Overview

- Security Concepts
- Transport Level Security
- JSON Web Token
- OAuth 2.0
  - *Client-side Web Apps*
  - *Server-side Web Apps*
- OpenID

## Server-side Web Apps

- Additional interactions
  - *server-side code (any language), the app can maintain the state*
  - *additional interactions, authorization code*
- Architecture
  - *Client at a server requests, remembers and refresh access tokens*
- Basic steps
  - *Client redirects user agent to the authorization endpoint*
  - *Resource owner grants access to the client or rejects the request*
  - *Authorization server provides **authorization code** to the client*
  - *Client requests **access and refresh tokens** from the auth. server*
  - *Client access the resource with the access token*
  - *When the token expires, client refreshes a token with refresh token*
- Advantages
  - *Access tokens not visible to clients, they are stored at the server*
  - *more secure, clients need to authenticate before they can get tokens*

# Server-side Web Apps Protocol



## Redirection – Step 1

- Methods and Parameters
  - same as for client-side app, except **response\_type** must be **code**
- Example

```
1 | https://accounts.google.com/o/oauth2/auth?
2 | client_id=621535099260.apps.googleusercontent.com&
3 | redirect_uri=http://w20.vitvar.com/examples/oauth/callback.html&
4 | scope=https://www.google.com/m8/feeds&
5 | response_type=code
```

## Callback + Access Token Request – steps 4, 5

- Callback
  - authorization server calls back **redirect\_uri**
  - client gets the **code** and requests **access\_token**
  - example (resource owner grants access):  
**`http://w20.vitvar.com/examples/oauth/callback.html?code=4/P7...`**
  - when user rejects → same as client-side access
- Access token request
  - **POST** request to token endpoint  
→ example Google token endpoint:  
**`https://accounts.google.com/o/oauth2/token`**
  - ```
1 POST /o/oauth2/token HTTP/1.1
2 Host: accounts.google.com
3 Content-Type: application/x-www-form-urlencoded
4
5 code=4/P7q7W91a-oMsCeLvIaQm6bTrgtp6&
6 client_id=621535099260.apps.googleusercontent.com&
7 client_secret=XTHhXh1S2UggvyWgWdK1EjXB&
8 redirect_uri=http://w20.vitvar.com/examples/oauth/callback.html&
9 grant_type=authorization_code
```

## Access Token (cont.)

- Access token response
  - Token endpoint responds with **access\_token** and **refresh\_token**
  - ```
1 { "access_token" : "1/fFAGRnJru1FTz70BzhT3Zg",
2   "expires_in"   : 3920,
3   "refresh_token" : "1/6BMfW9j53gdGimsixUH6kU5RsR4zwI9lUVX-tqf8JXQ" }
```
- Refreshing a token
  - **POST** request to the token endpoint with **grant\_type=refresh\_token** and the previously obtained value of **refresh\_token**
  - ```
1 POST /o/oauth2/token HTTP/1.1
2 Host: accounts.google.com
3 Content-Type: application/x-www-form-urlencoded
4
5 client_id=21302922996.apps.googleusercontent.com&
6 client_secret=XTHhXh1S1UNGvyWgWdK1EjXB&
7 refresh_token=1/6BMfW9j53gdGimsixUH6kU5RsR4zwI9lUVX-tqf8JXQ&
8 grant_type=refresh_token
```
- Accessing a resource is the same as in the client-side app

## Overview

- Security Concepts
- Transport Level Security
- JSON Web Token
- OAuth 2.0
- **OpenID**
  - *OpenID Connect*

## OpenID Protocol

- Motivation – many user accounts
  - *users need to maintain many accounts to access various services*
  - *multiple passwords problem*
- Objectives
  - *allows apps to utilize an OpenID provider*
    - *a third-party authentication service*
    - *federated login*
  - *users have one account with the OpenID provider and use it for apps that support the provider*
- OpenID providers
  - *it is a protocol, anybody can build a provider*
  - *Google, Yahoo!, Seznam.cz, etc.*
- Specification
  - *OpenID Protocol* [🔗](#)

## Interaction Sequence



- Discovery – discovery of a service associated with a resource
- XRDS – eXtensible Resource Descriptor Sequence
  - *format for discovery result*
  - *developed to serve resource discovery for OpenID*
  - *Web app retrieves endpoint to send login authentication requests*

## Login Authentication Request – Step 5

- Example Google OpenID provider

```
1 https://www.google.com/accounts/o8/id
2 ?openid.ns=http://specs.openid.net/auth/2.0
3 &openid.return_to=https://www.example.com/checkauth
4 &openid.realm=http://www.example.com/
5 &openid.assoc_handle=ABSmpf6DNMw
6 &openid.mode=checkid_setup
```

- Parameters

- **ns** – *protocol version (obtained from the XRDS)*
- **mode** – *type of message or additional semantics (**checkid\_setup** indicates that interaction between the provider and the user is allowed during authentication)*
- **return\_to** – *callback page the provider sends the result*
- **realm** – *domain the user will trust, consistent with **return\_to***
- **assoc\_handle** – *"log in" for web app with openid provider*

*\* Not all fields shown, check the OpenID spec for the full list of fields and their values*

## Login Authentication Response – Step 8

- User logs in successfully

```
1 http://www.example.com/checkauth
2 ?openid.ns=http://specs.openid.net/auth/2.0
3 &openid.mode=id_res
4 &openid.return_to=http://www.example.com:8080/checkauth
5 &openid.assoc_handle=ABSmpf6DNMw
6 &openid.identity=https://www.google.com/accounts/o8/id?id=ACyQatiscWwqs4UQV_L
```

- Web app will use **identity** to identify user in the application
- response is also signed using a list of fields in the response (not shown in the listing)

- User cancels

```
1 http://www.example.com/checkauth
2 ?openid.mode=cancel
3 &openid.ns=http://specs.openid.net/auth/2.0
```

*\* Not all fields shown, check the OpenID spec for the full list of fields and their values*

## Overview

- Security Concepts
- Transport Level Security
- JSON Web Token
- OAuth 2.0
- OpenID
  - *OpenID Connect*

# OpenID Connect (OIDC)

- Simple identity layer on top of the OAuth 2.0 protocol
  - *Authorization Server to verify identity of users*
  - *Clients can obtain basic profile information about users*
- OIDC vs OpenID
  - *OIDC does many of the same tasks as OpenID 2.0*
  - *API-friendly*
    - *can be used by native and mobile applications*
  - *Robust signing and encryption mechanisms*
  - *Native integration with OAuth 2.0.*
- Defined by OpenID open standard
  - *OpenID Connect*

## Interaction sequence

