

Web 2.0

Lecture 6: Security

doc. Ing. Tomáš Vitvar, Ph.D.

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



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <http://vitvar.com/courses/w20>



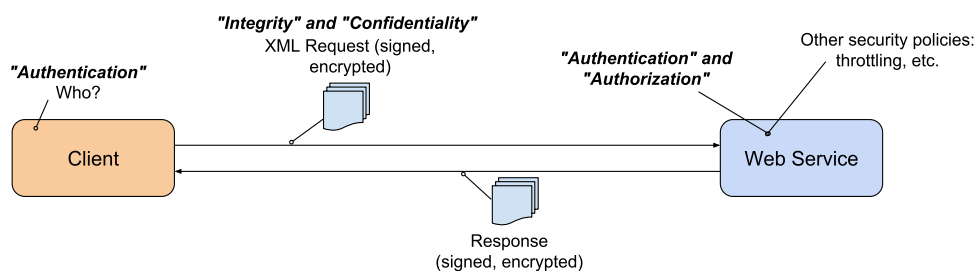
Modified: Sun May 20 2018, 12:48:20
Humla v0.3

Overview

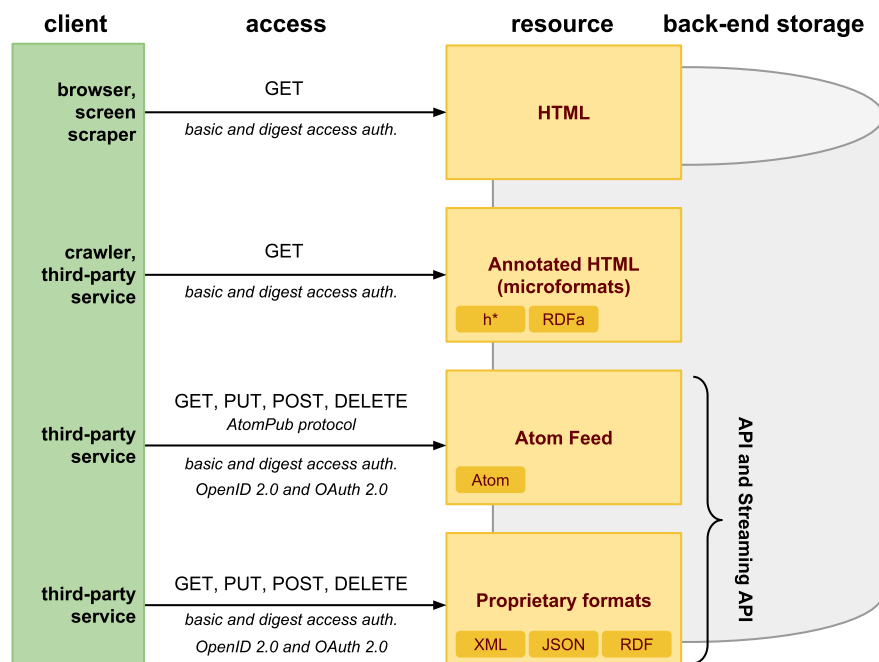
- **Security Concepts**
- Authentication and Authorization
- JSON Web Token
- OAuth 2.0
- OpenID

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



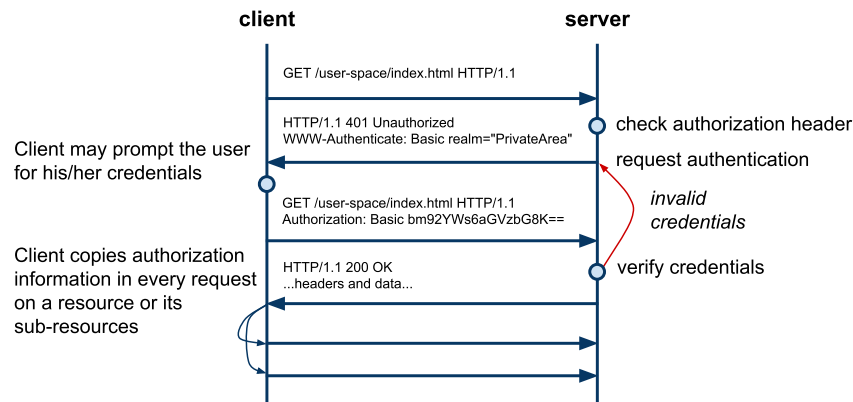
Overview

- Security Concepts
- **Authentication and Authorization**
- JSON Web Token
- OAuth 2.0
- OpenID

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 > bm92YWw6aGVzZG8K
4
5 # and to decode
6 echo "bm92YWw6aGVzZG8K" | base64 -d # use capital "D" in OS X
7 > novak:heslo
```

- Comments

- When SSL 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
- Authentication and Authorization
- **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, all 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  
gRG9lIiwiaXNtb2NpYWwiOnRydWV9.  
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 interactions

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
- Authentication and Authorization
- JSON Web Token
- **OAuth 2.0**
 - Client-side Web Apps
 - Server-side Web Apps
 - OAuth 2.0 vs. OAuth 1.0
- 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, PicasaWeb, 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** (e.g., pictures in Google PicasaWeb)*
- **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
- Authentication and Authorization
- JSON Web Token
- OAuth 2.0
 - *Client-side Web Apps*
 - *Server-side Web Apps*
 - *OAuth 2.0 vs. OAuth 1.0*
- OpenID

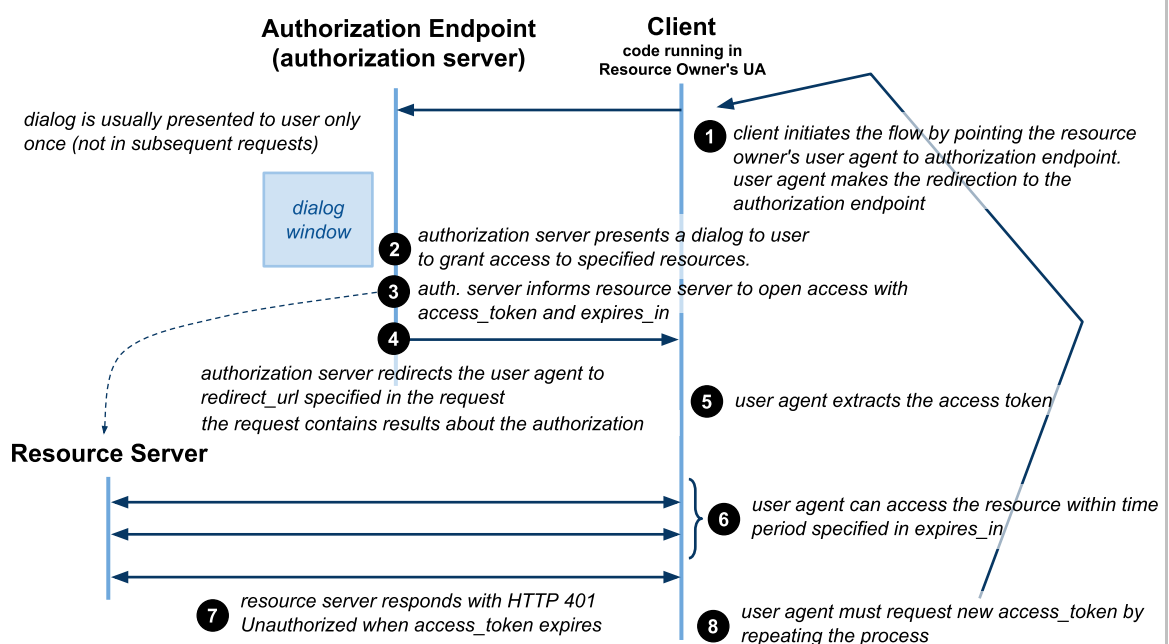
Client-side Web Apps

- 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 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 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 | http://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`**

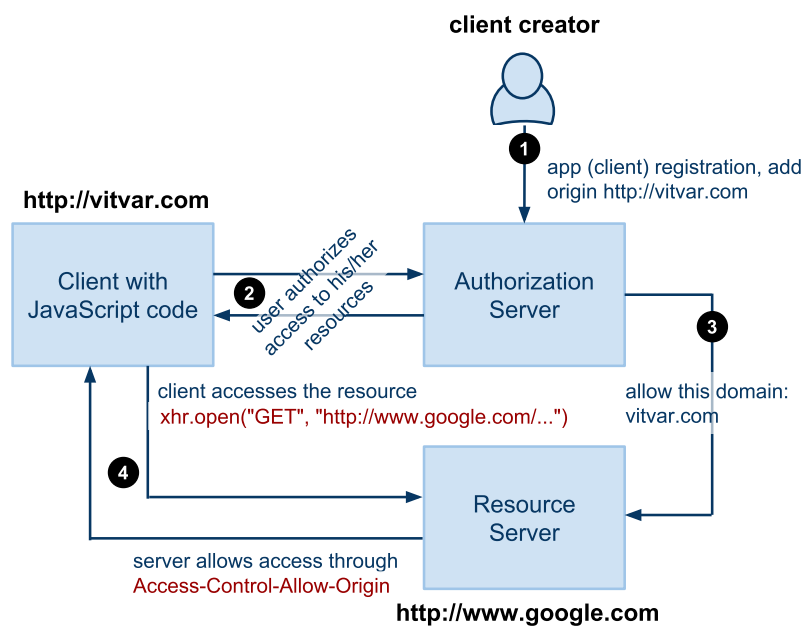
```
1 | curl https://www.google.com/m8/feeds/contacts/default/full?
2 |     oauth_token=1/dERFd34Sf
```
- HTTP Header **Authorization**

```
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

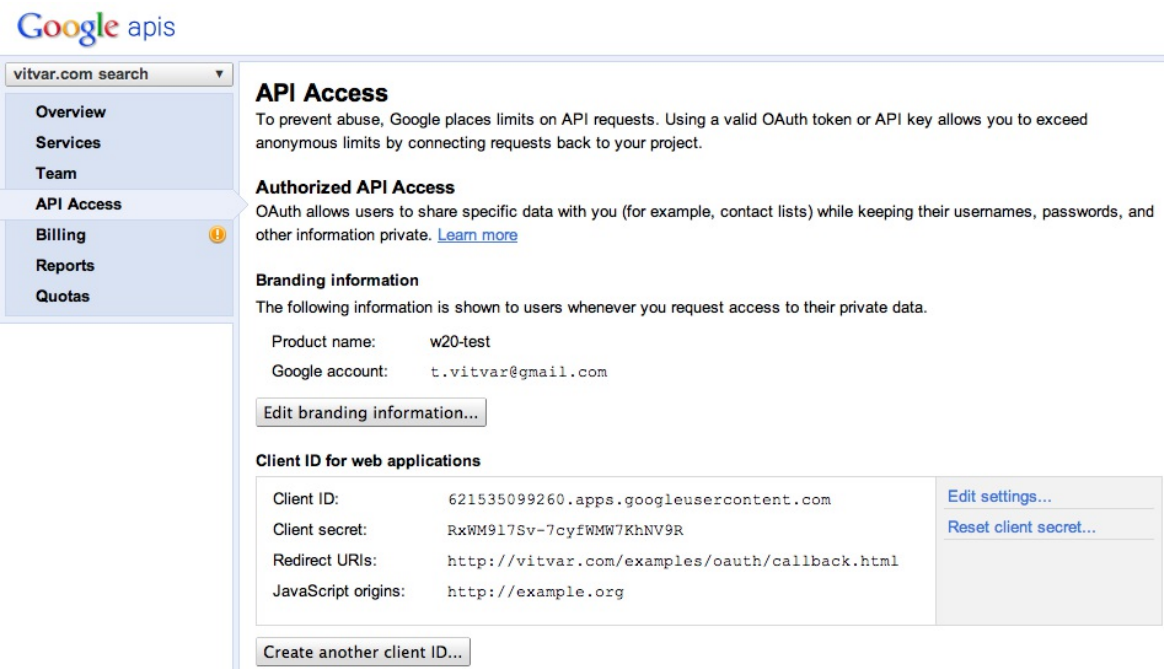
- Success – **200 OK**
- Error – **401 Unauthorized** when token expires or the client hasn't performed the authorization request.

Cross-Origin Resource Sharing



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

Example Application Registration



The screenshot shows the Google APIs console interface. On the left is a sidebar with navigation links: Overview, Services, Team, API Access (selected), Billing, Reports, and Quotas. The main content area is titled 'API Access' and contains the following sections:

- API Access**: A paragraph explaining that Google places limits on API requests and that using a valid OAuth token or API key allows exceeding these limits.
- Authorized API Access**: A paragraph explaining that OAuth allows sharing specific data while keeping usernames, passwords, and other information private, with a link to 'Learn more'.
- Branding information**: A paragraph stating that the following information is shown to users when requesting access to their private data. Below this, it lists:
 - Product name: w20-test
 - Google account: t.vitvar@gmail.comA button 'Edit branding information...' is located below the list.
- Client ID for web applications**: A table with the following data:

Client ID:	621535099260.apps.googleusercontent.com	Edit settings...
Client secret:	RxWM917Sv-7cyfWMW7KhNV9R	Reset client secret...
Redirect URIs:	http://vitvar.com/examples/oauth/callback.html	
JavaScript origins:	http://example.org	

A button 'Create another client ID...' is located below the table.

At the bottom of the slide, the footer reads: 'Lecture 6: Security, CTU Summer Semester 2017/2018, @TomasVitvar' on the left and '- 29 -' on the right.

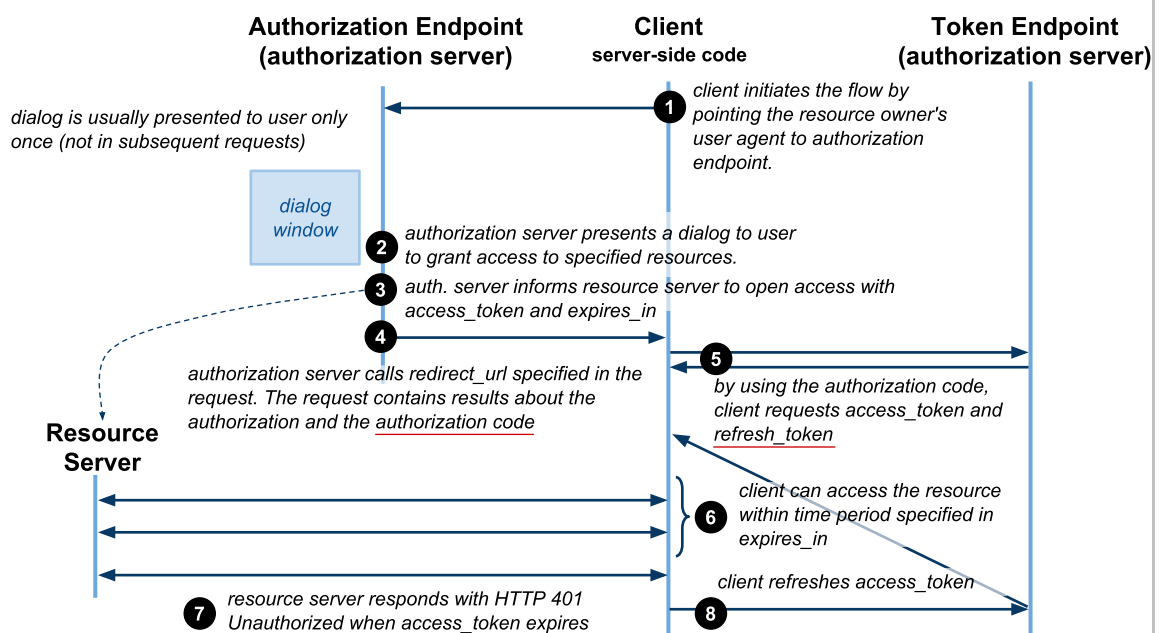
Overview

- Security Concepts
- Authentication and Authorization
- JSON Web Token
- OAuth 2.0
 - *Client-side Web Apps*
 - *Server-side Web Apps*
 - *OAuth 2.0 vs. OAuth 1.0*
- 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=XTHhXh1SlUNgvyWgWdk1EjXB&  
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
- Authentication and Authorization
- JSON Web Token
- OAuth 2.0
 - *Client-side Web Apps*
 - *Server-side Web Apps*
 - *OAuth 2.0 vs. OAuth 1.0*
- OpenID

Why new version?

- OAuth 1.0 in brief
 - *security not based on SSL*
 - *client must sign every request using a defined algorithm*
 - *e.g., public-private key signatures by RSA*
 - *More complex to be implemented by clients*
 - *although client libraries exist*
 - *not suitable for JavaScript-based clients*
- OAuth 2.0 simplifies the process
 - *SSL is required for all communications to generate the token*
 - *Signatures are not required for the actual API calls once the token has been generated*
 - *SSL is also strongly recommended here*
 - *supports various clients including JavaScript and mobile*

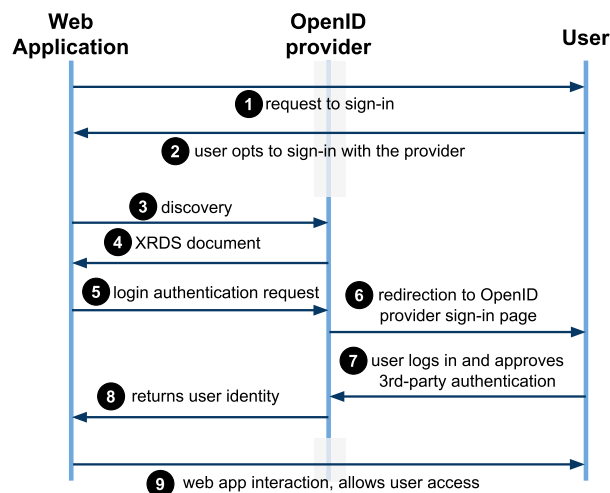
Overview

- Security Concepts
- Authentication and Authorization
- JSON Web Token
- OAuth 2.0
- **OpenID**

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=ACyQatiscWvwqs4UQV_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*