

Web 2.0

Lecture 7: Security in REST

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>



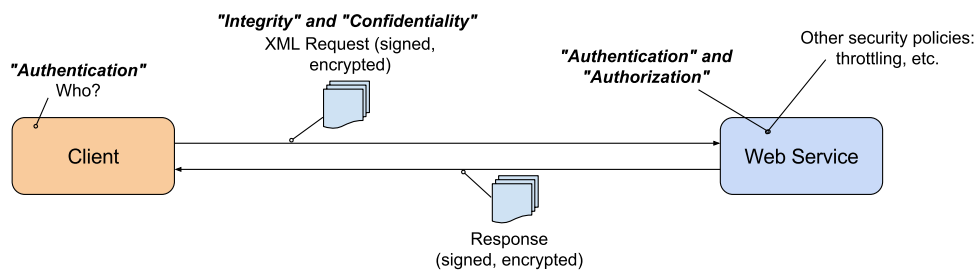
Modified: Mon Apr 07 2014, 20:27:18
Humla v0.3

Overview

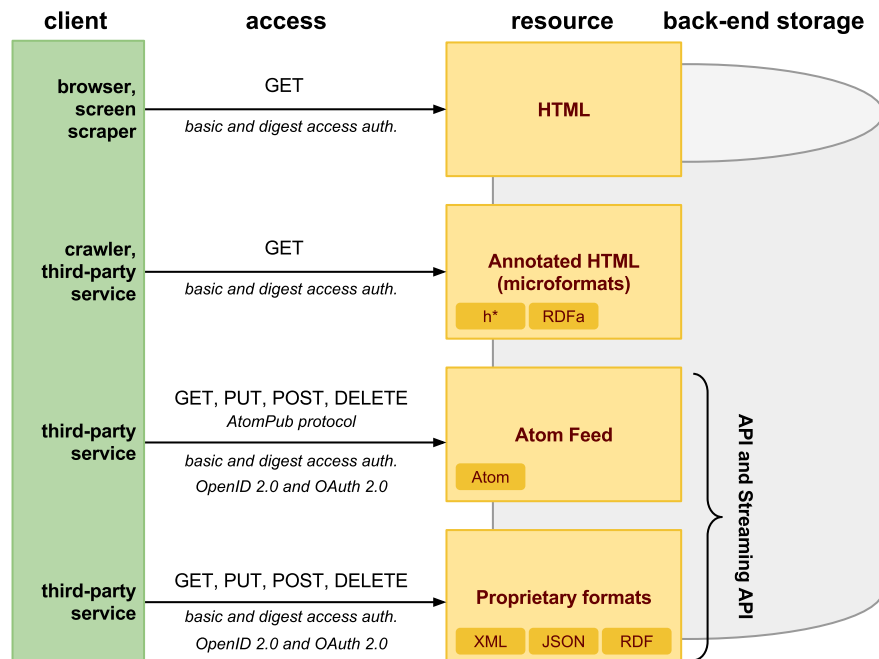
- **Security Concepts**
- Authentication and Authorization
- OAuth 2.0
- OpenID

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



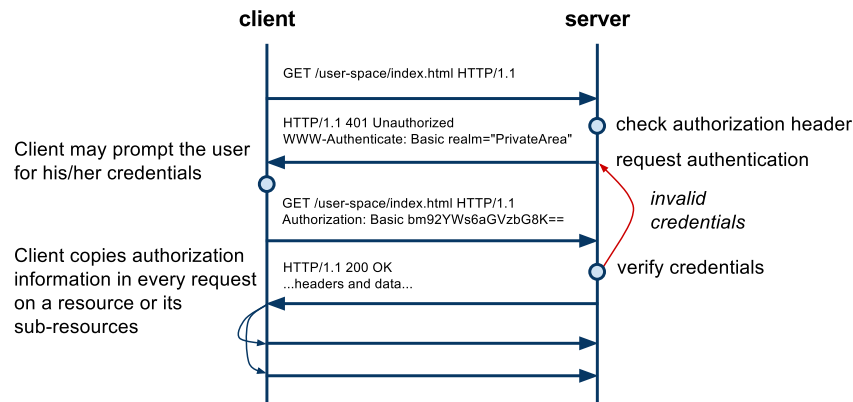
Overview

- Security Concepts
- **Authentication and Authorization**
- 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***
- **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*
 2. *Server requests authentication with WWW-Authenticate*
 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*

Nonce and QoP

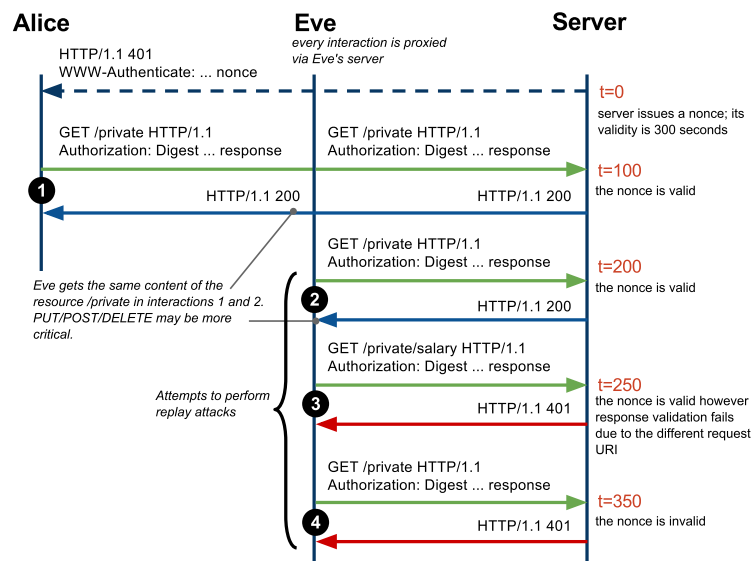
- Nonce
 - *A value to identify an interaction that should occur only once*
 - **nonce** – *generated by the server*
 - *may have a time period for which the nonce is valid*
 - *may be computed using client IP, ETag of the resource, etc.*
 - *this limits chances for the replay attack.*
 - **cnonce** – *generated by the client*
- QoP – quality of protection
 - *Further improvements to prevent replay attacks and enables non-repudiation*

Algorithms

- Algorithm for **response** value of **authorization** header
 - No quality of protection (**qop** is missing or **qop=none**)
 - limits chances of replay-attacks
 - with quality of protection (**qop=auth**)
 - with quality of protection for message integrity (**qop=auth-int**)
 - enables non-repudiation (i.e., proof of integrity and origin of data)

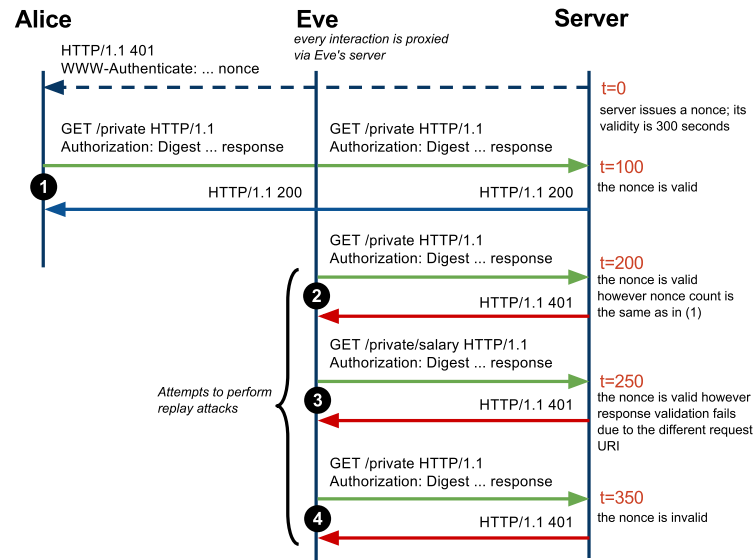
Replay Attack

- Replay Attack Scenario (quality of protection is **none**)
 - The communication is not encrypted (i.e., no use of HTTPS)
 - Eve listens to the Alice's communication (e.g. on a proxy server)
 - Eve resends requests with headers from Alice's requests



Replay Attack (Cont.)

- Replay Attack Scenario (quality of protection is **auth** or **auth-int**)
 - **nonceCount** should be incremented in every request to a response of the nonce value from the server



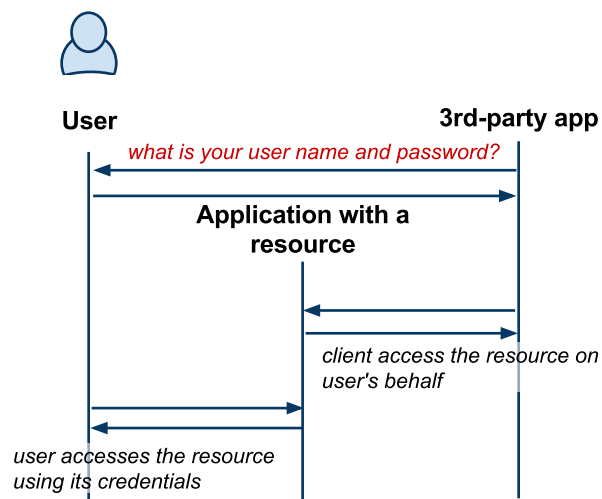
Overview

- Security Concepts
- Authentication and Authorization
- **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
- OAuth 2.0
 - *Client-side Web Apps*
 - *Server-side Web Apps*
 - *OAuth 2.0 vs. OAuth 1.0*
- OpenID

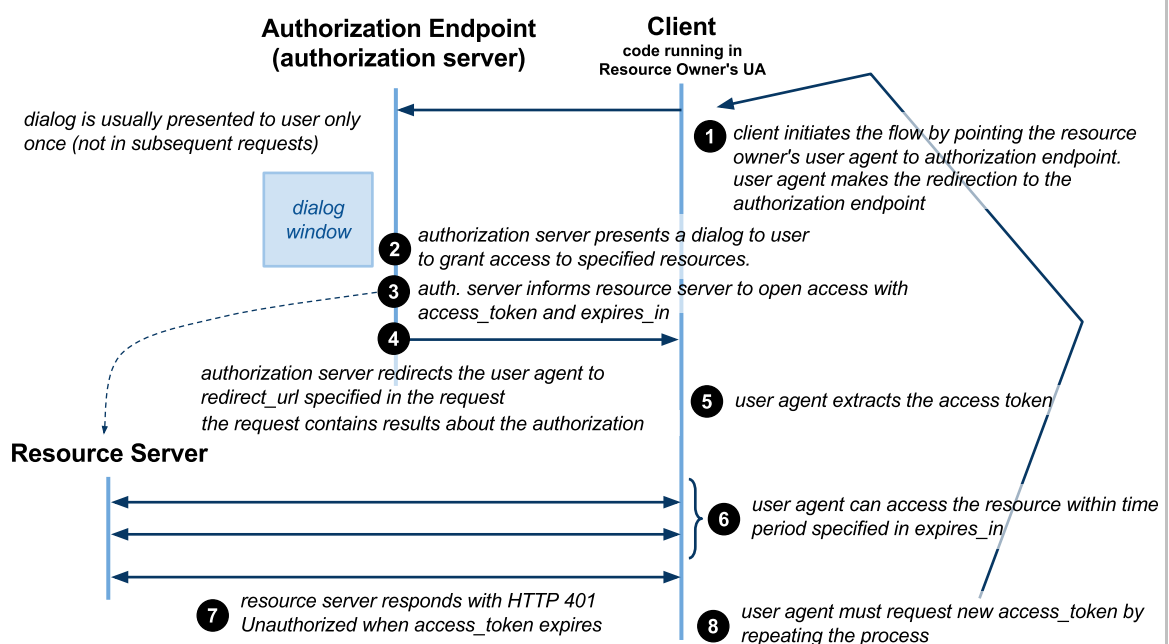
Client-side Web Apps

- 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

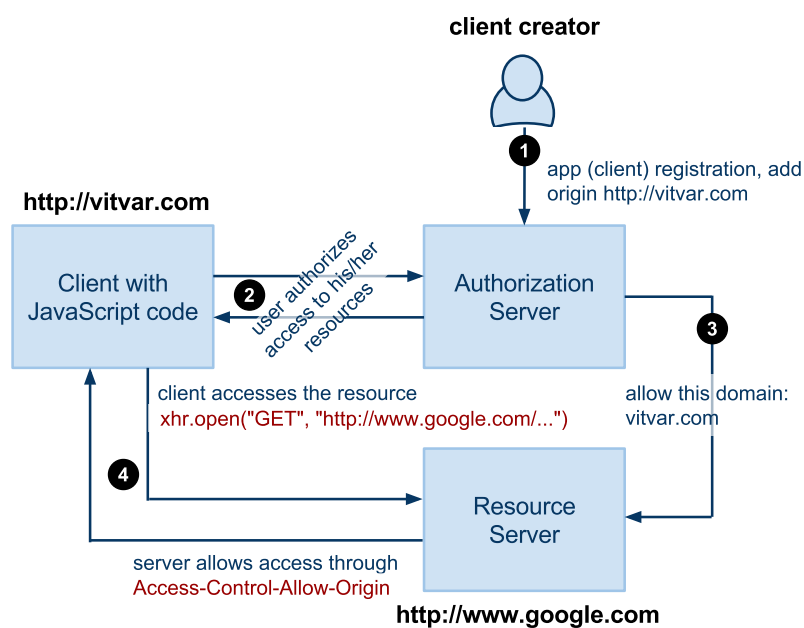
Callback – steps 4 and 5

- 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:
- Resource owner rejects the access
 - authorization server calls back **redirect_uri** with query string parameter **error=access_denied**
 - Example:

Accessing Resources – Step 6

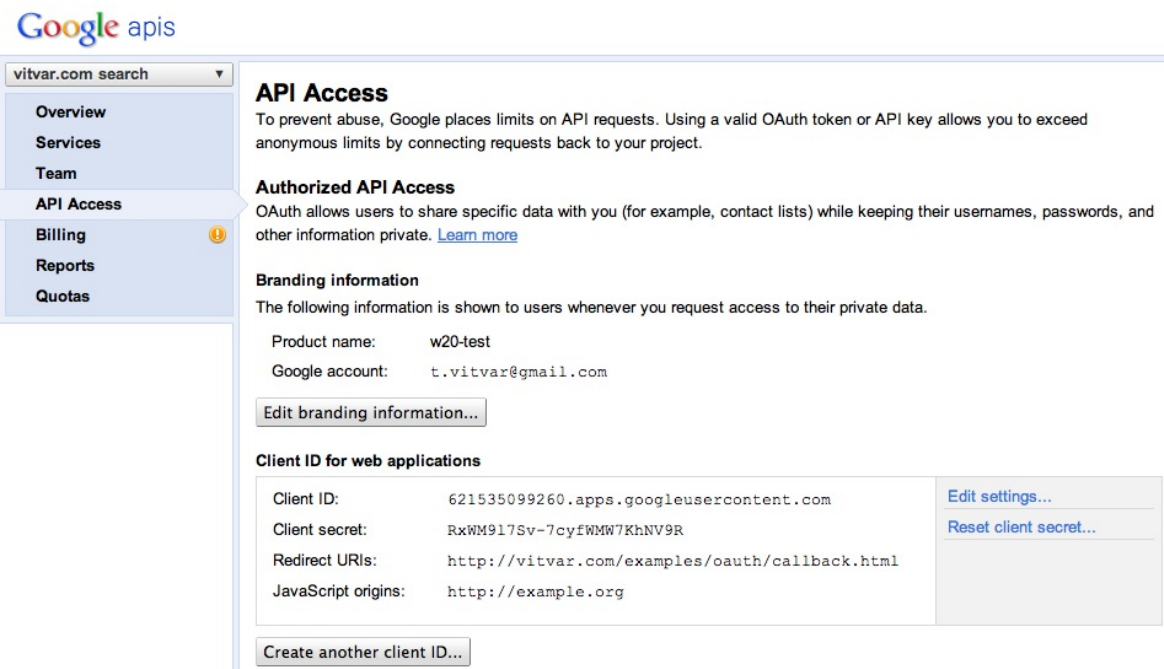
- Request
 - client can access resources defined by **scope**
 - resources' URIs defined in a particular documentation
 - Example Google Contacts
 - to access all users' contacts stored in Google
 - **scope** is **`https://www.google.com/m8/feeds`**
 - Query string parameter **oauth_token**
 - HTTP Header **Authorization**
 - 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, a table lists:
 - Product name: w20-test
 - Google account: t.vitvar@gmail.comA button 'Edit branding information...' is located below the table.
- Client ID for web applications**: A table with the following information:
 - Client ID: 621535099260.apps.googleusercontent.com
 - Client secret: RxWM917Sv-7cyfWMW7KhNV9R
 - Redirect URIs: http://vitvar.com/examples/oauth/callback.html
 - JavaScript origins: http://example.orgButtons 'Edit settings...' and 'Reset client secret...' are to the right of the table. A button 'Create another client ID...' is at the bottom.

At the bottom of the console, there is a footer with the text 'Lecture 7: Security in REST, CTU Summer Semester 2016/2017, @TomasVitvar' and a page number '- 27 -'.

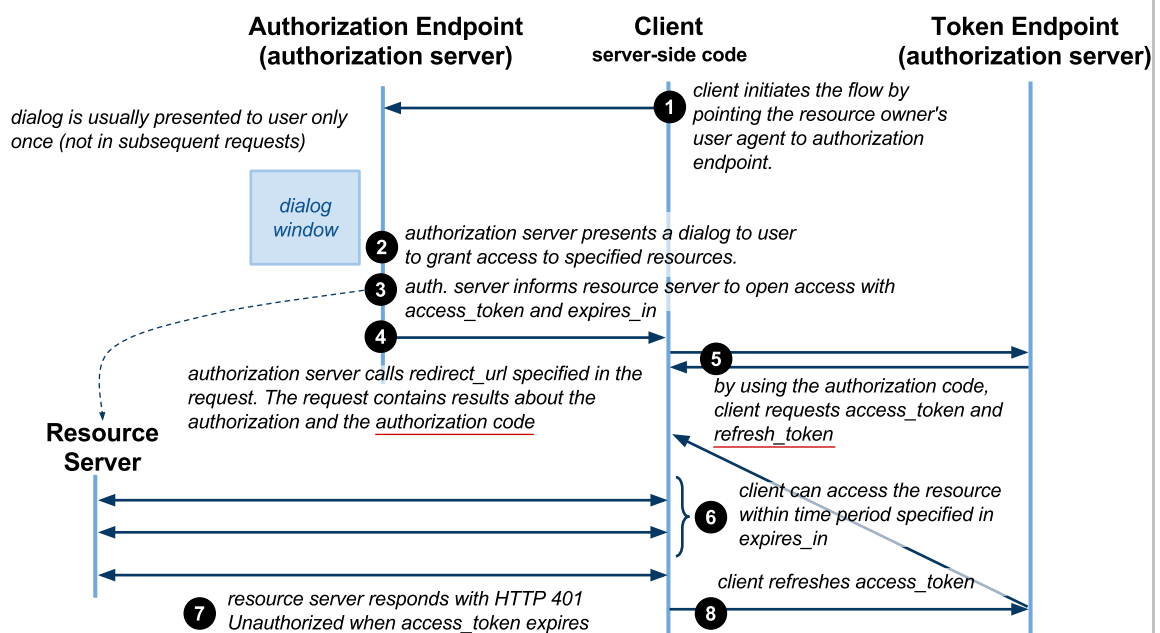
Overview

- Security Concepts
- Authentication and Authorization
- 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

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://humla.vitvar.com/slides/w20/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`

Access Token (cont.)

- Access token response
 - *Token endpoint responds with **access_token** and **refresh_token***
- Refreshing a token
 - **POST** request to the token endpoint with **grant_type=refresh_token** and the previously obtained value of **refresh_token**
- Accessing a resource is the same as in the client-side app

Overview

- Security Concepts
- Authentication and Authorization
- 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*

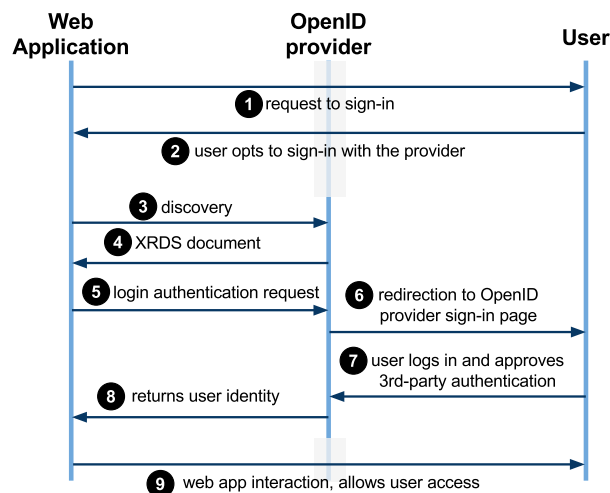
Overview

- Security Concepts
- Authentication and Authorization
- 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
 - 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
 - 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
- * Not all fields shown, check the OpenID spec for the full list of fields and their values*