Web 2.0
Lecture 7: 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
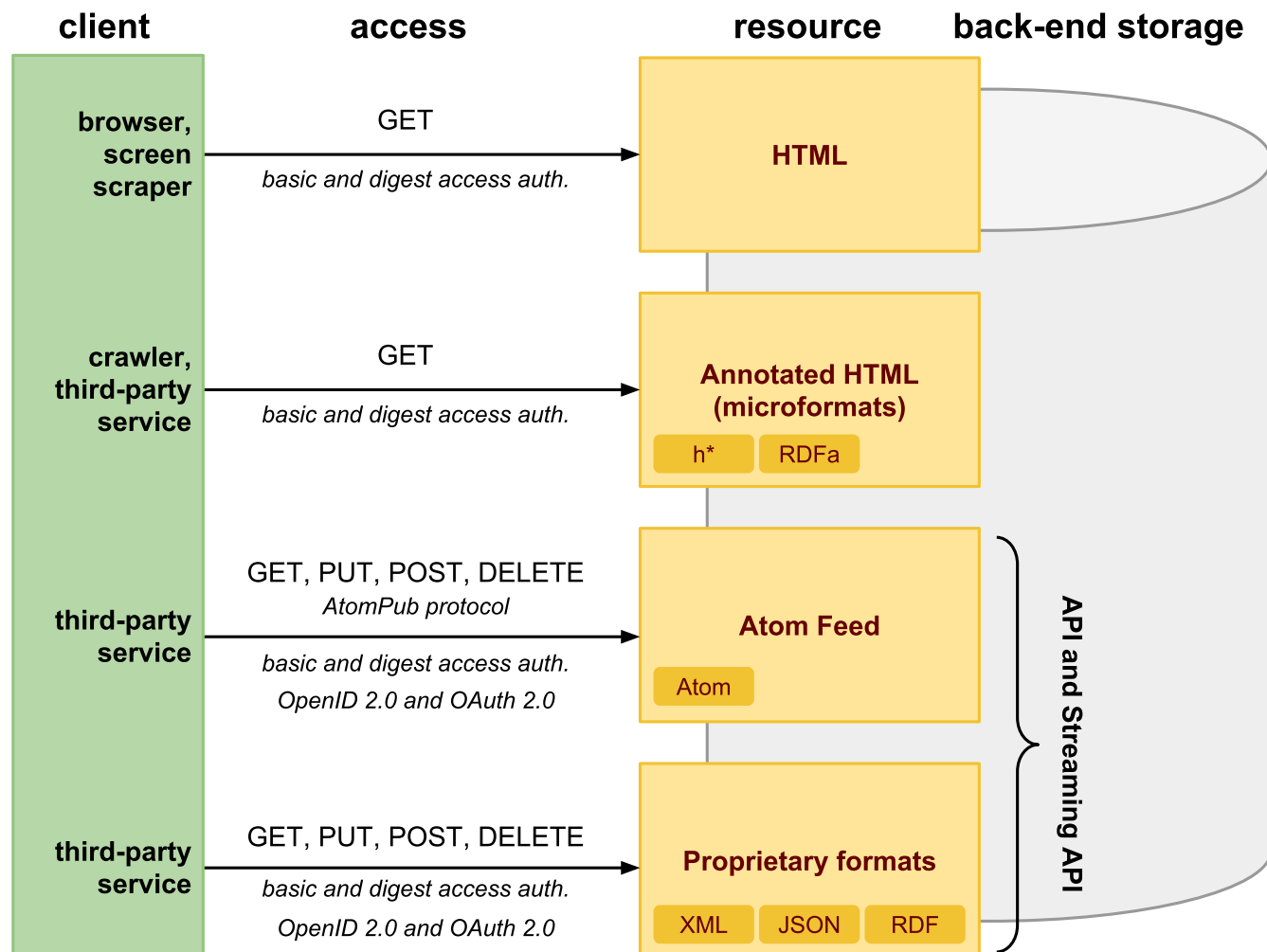
Overview

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

Web Service Security Concepts

- Securing the client-server communcation
  - *Message-level security*
  - *Transport-level security*

- Ensure
  - *Authentication – verify a client's identity*
  - *Authorizaton – 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*

"Integrity" and "Confidentiality"
XML Request (signed, encrypted)

Other security policies: throttling, etc.

"Authentication"
Who?

"Authentication" and "Authorization"

Client

Web Service

Response

Data on the Web

client	access	resource	back-end storage

browser, screen scraper

GET

basic and digest access auth.

HTML

---

crawler, third-party service

GET

basic and digest access auth.

Annotated HTML (microformats)

h* RDFa

---

third-party service

GET, PUT, POST, DELETE
AtomPub protocol

basic and digest access auth.
OpenID 2.0 and OAuth 2.0

Atom Feed

Atom

---

third-party service

GET, PUT, POST, DELETE

basic and digest access auth.
OpenID 2.0 and OAuth 2.0

Proprietary formats

XML JSON RDF
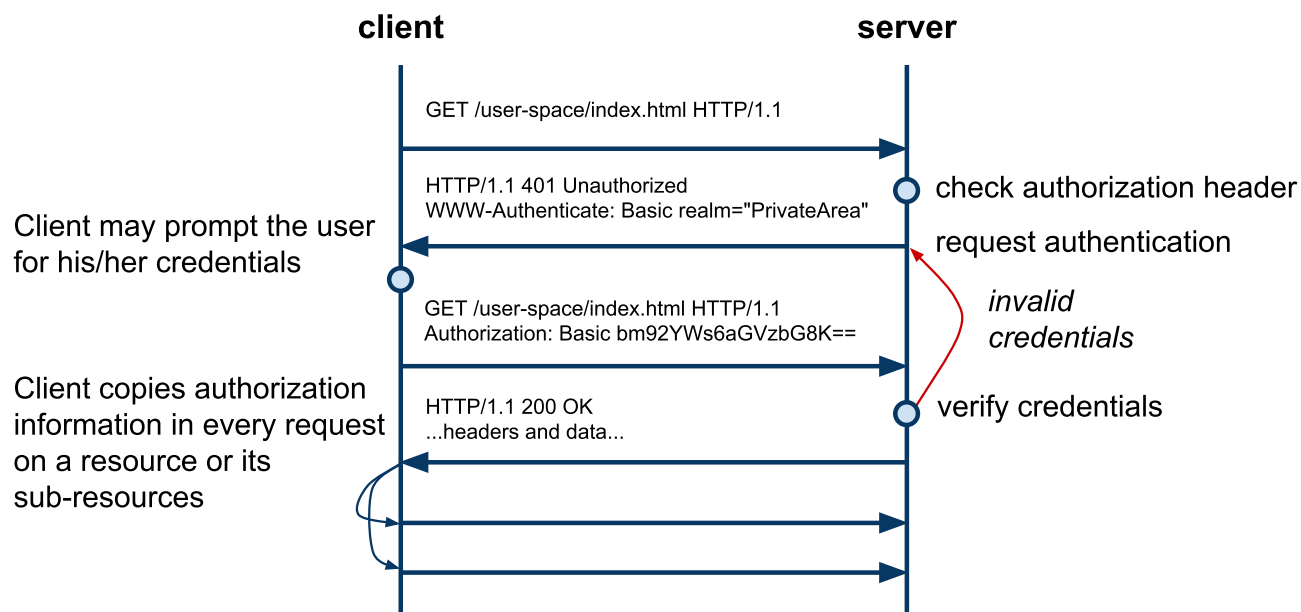
API and Streaming API

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

client   server

GET /user-space/index.html HTTP/1.1

HTTP/1.1 401 Unauthorized   check authorization header
WWW-Authenticate: Basic realm="PrivateArea"

Client may prompt the user   request authentication
for his/her credentials
                                                                                          invalid
                                                                                          credentials
GET /user-space/index.html HTTP/1.1
Authorization: Basic bm92YWs6aGVzbG8K==

Client copies authorization   HTTP/1.1 200 OK   verify credentials
information in every request   ...headers and data...
on a resource or its
sub-resources

- 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 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"
     ```
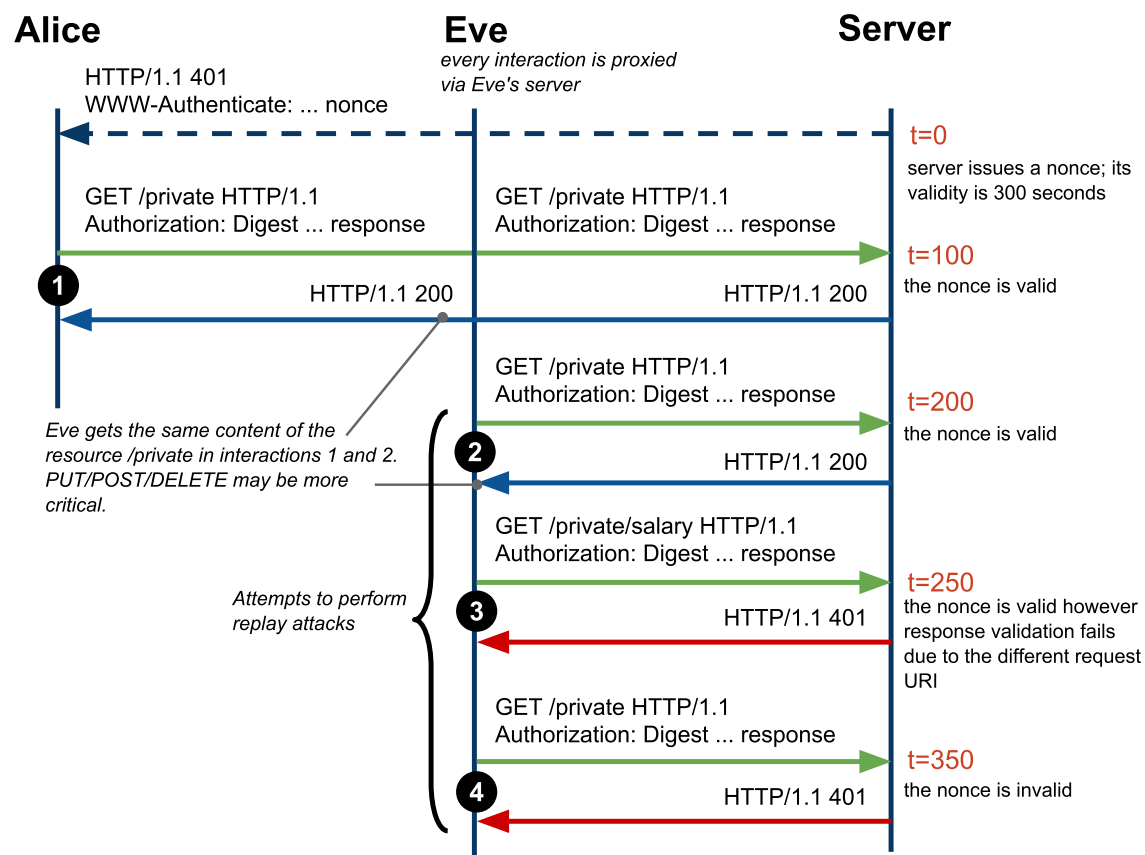
Nonce and QoP

- 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*

    ```
    1   HA1 = MD5(username:realm:password)
    2   HA2 = MD5(method:digestURI)
    3   response = MD5(HA1:nonce:HA2)
    ```

  - *with quality of protection (*`qop=auth`*)*

    ```
    1   HA1 = MD5(username:realm:password)
    2   HA2 = MD5(method:digestURI)
    3   response = MD5(HA1:nonce:nonceCount:cnonce:qop:HA2)
    ```

  - *with quality of protection for message integrity (*`qop=auth-int`*)*
    - → *enables non-repudiation (i.e., proof of integrity and origin of data)*

    ```
    1   HA1 = MD5(username:realm:password)
    2   HA2 = MD5(method:digestURI:MD5(payload))
    3   response = MD5(HA1:nonce:nonceCount:cnonce:qop:HA2)
    ```
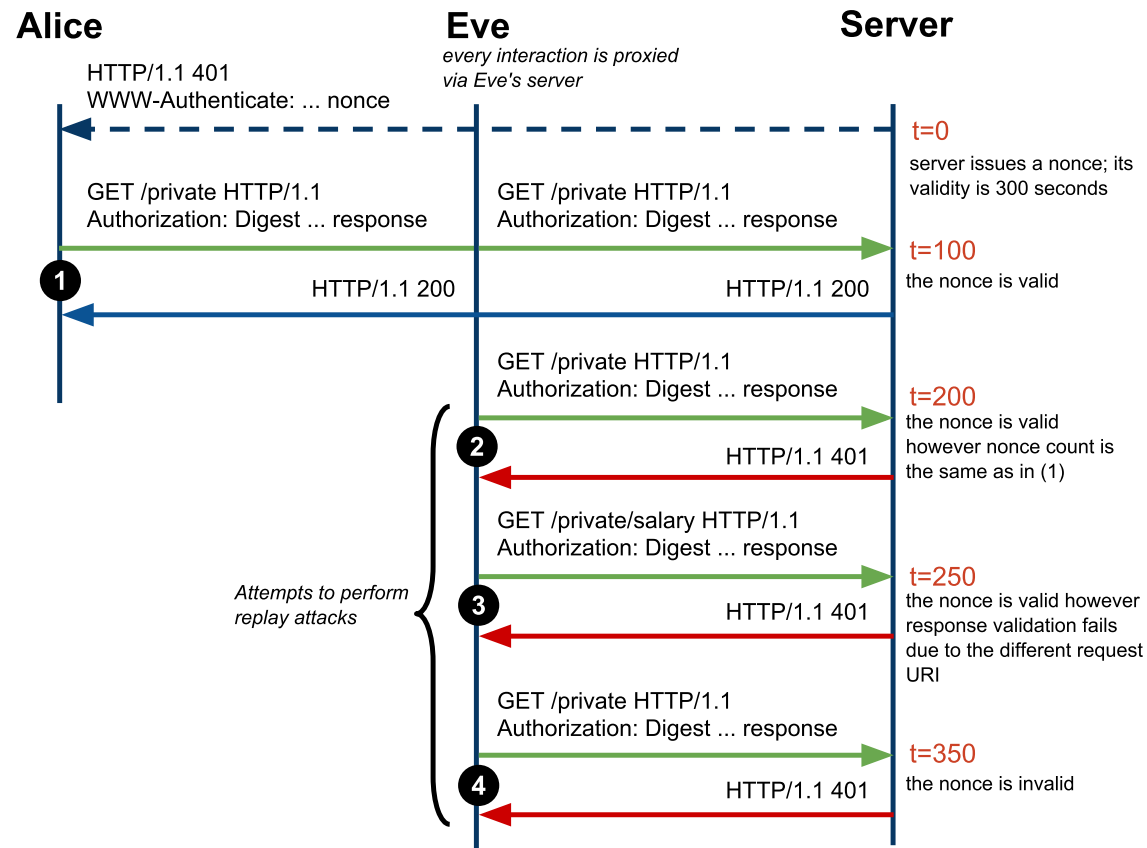
Replay Attack

- 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*

Alice

Eve
*every interaction is proxied
via Eve's server*

Server

HTTP/1.1 401
WWW-Authenticate: ... nonce

t=0
server issues a nonce; its
validity is 300 seconds

GET /private HTTP/1.1
Authorization: Digest ... response

GET /private HTTP/1.1
Authorization: Digest ... response

t=100
the nonce is valid

1   HTTP/1.1 200   HTTP/1.1 200

GET /private HTTP/1.1
Authorization: Digest ... response

t=200
the nonce is valid

*Eve gets the same content of the
resource /private in interactions 1 and 2.
PUT/POST/DELETE may be more
critical.*

2   HTTP/1.1 200

GET /private/salary HTTP/1.1
Authorization: Digest ... response

t=250
the nonce is valid however
response validation fails
due to the different request
URI

*Attempts to perform
replay attacks*

3   HTTP/1.1 401

GET /private HTTP/1.1
Authorization: Digest ... response

t=350
the nonce is invalid

4   HTTP/1.1 401

Replay Attack (Cont.)

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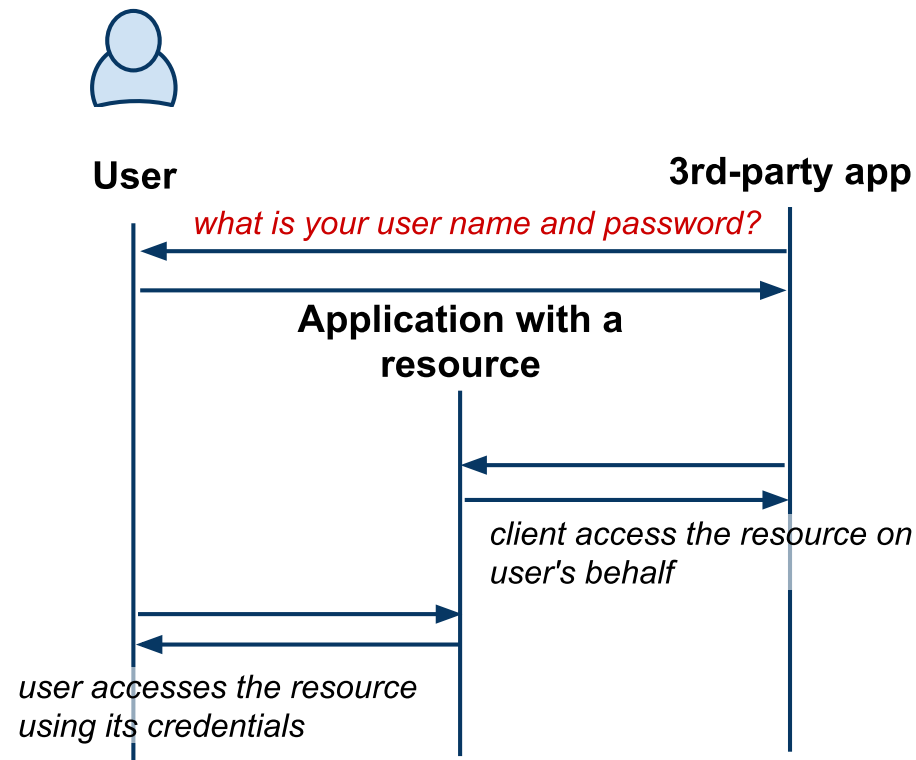 - *`nonceCount` should be incremented in every request to a response of the nonce value from the server*

Alice   **Eve**   **Server**

Eve: *every interaction is proxied via Eve's server*

HTTP/1.1 401
WWW-Authenticate: ... nonce

t=0
server issues a nonce; its validity is 300 seconds

GET /private HTTP/1.1
Authorization: Digest ... response

GET /private HTTP/1.1
Authorization: Digest ... response

t=100
the nonce is valid

1 HTTP/1.1 200   HTTP/1.1 200

GET /private HTTP/1.1
Authorization: Digest ... response

t=200
the nonce is valid however nonce count is the same as in (1)

2 HTTP/1.1 401

GET /private/salary HTTP/1.1
Authorization: Digest ... response

t=250
the nonce is valid however response validation fails due to the different request URI

Attempts to perform replay attacks

3 HTTP/1.1 401

GET /private HTTP/1.1
Authorization: Digest ... response

t=350
the nonce is invalid

4 HTTP/1.1 401

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

User   3rd-party app

what is your user name and password?

Application with a resource

client access the resource on user's behalf

user accesses the resource using its credentials

- 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 compatibile 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

**Authorization Endpoint
(authorization server)**

Client
code running in
Resource Owner's UA

*dialog is usually presented to user only
once (not in subsequent requests)*

dialog
window

1 *client initiates the flow by pointing the resource
owner's user agent to authorization endpoint.
user agent makes the redirection to the
authorization endpoint*

2 *authorization server presents a dialog to user
to grant access to specified resources.*

3 *auth. server informs resource server to open access with
access_token and expires_in*

4

*authorization server redirects the user agent to
redirect_url specified in the request
the request contains results about the authorization*

5 *user agent extracts the access token*

Resource Server

6 *user agent can access the resource within time
period specified in expires_in*

7 *resource server responds with HTTP 401
Unauthorized when access_token expires*

8 *user agent must request new access_token by
repeating the process*

Redirection – Step 1

- Methods and Parameters
  - *Methods:* `GET` *or* `POST`
  - *example authorazation 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    hhttp://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

client creator

① app (client) registration, add origin http://vitvar.com

http://vitvar.com

Client with JavaScript code

② user authorizes access to his/her resources

Authorization Server

③ allow this domain: vitvar.com

client accesses the resource
xhr.open("GET", "http://www.google.com/...")

④

Resource Server

server allows access through
Access-Control-Allow-Origin

http://www.google.com

— see Same Origin and Cross-Origin for details
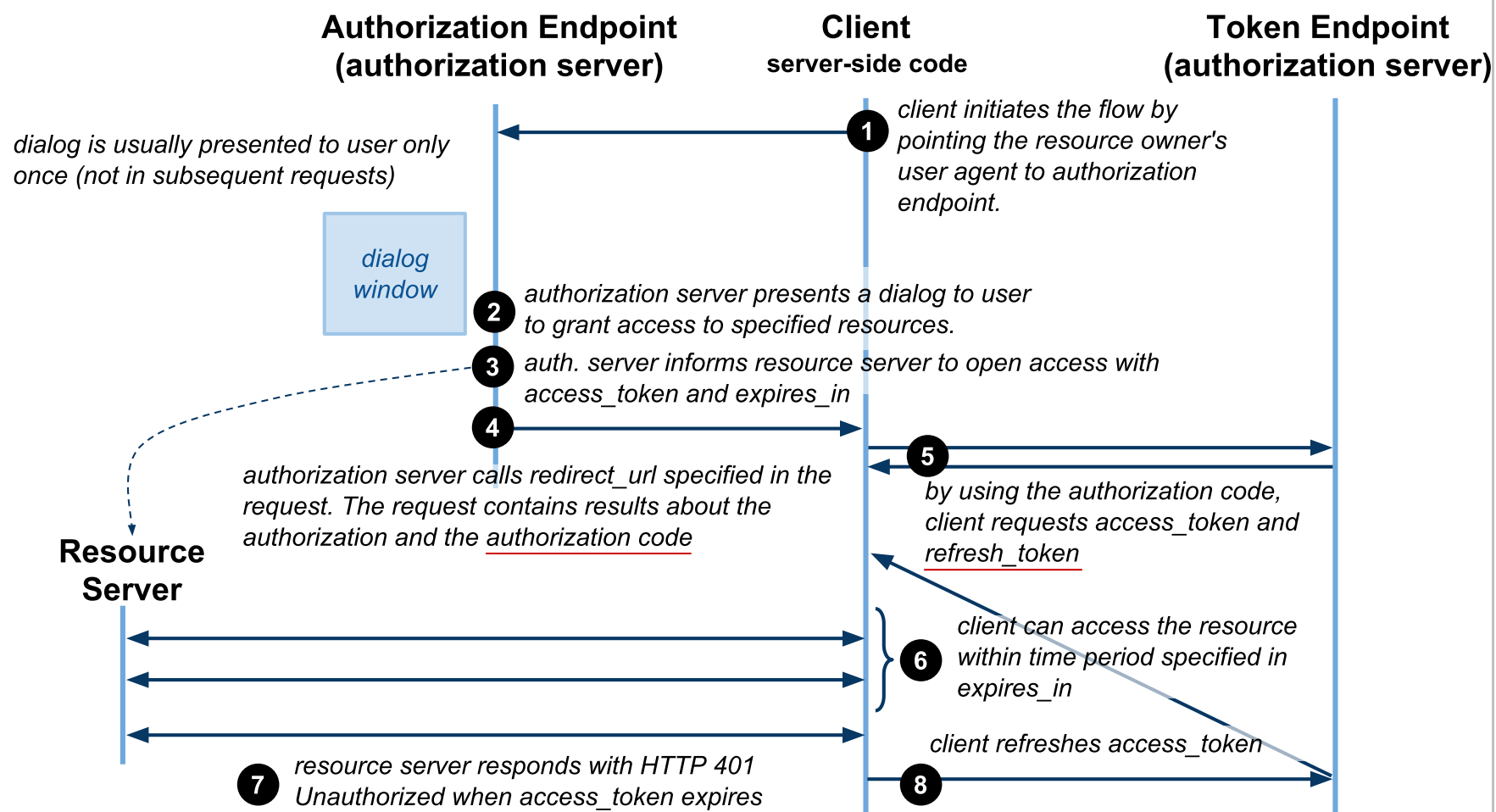
Example Application Registration

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

Authorization Endpoint (authorization server)   **Client** server-side code   **Token Endpoint (authorization server)**

dialog is usually presented to user only once (not in subsequent requests)

dialog window

1 *client initiates the flow by pointing the resource owner's user agent to authorization endpoint.*

2 *authorization server presents a dialog to user to grant access to specified resources.*

3 *auth. server informs resource server to open access with access_token and expires_in*

4

5

authorization server calls redirect_url specified in the request. The request contains results about the authorization and the <u>authorization code</u>

by using the authorization code, client requests access_token and <u>refresh_token</u>

Resource Server

6 *client can access the resource within time period specified in expires_in*

client refreshes access_token

7 *resource server responds with HTTP 401 Unauthorized when access_token expires*

8

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
- 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*
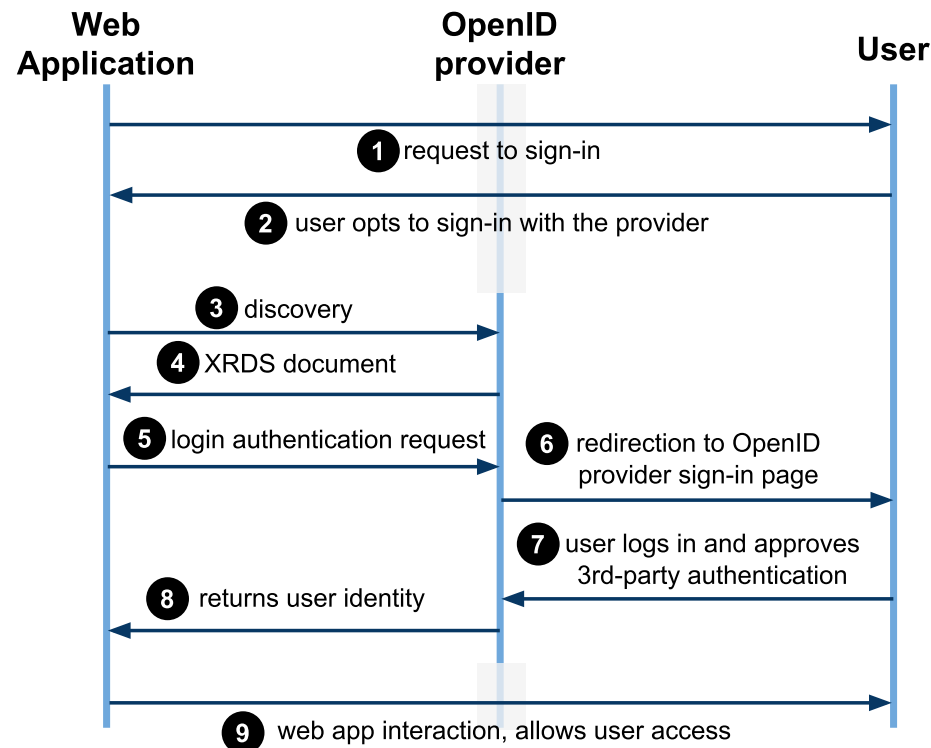 - *supports various clients including JavaScript and mobile*

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

Sequence diagram between Web Application, OpenID provider, and User:

1. request to sign-in
2. user opts to sign-in with the provider
3. discovery
4. XRDS document
5. login authentication request
6. redirection to OpenID provider sign-in page
7. user logs in and approves 3rd-party authentication
8. returns user identity
9. web app interaction, allows user access

- 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 logins 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_U
```

    – *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