RESO Web API Security v1.0.1

Copyright 2014 RESO

Section 1 - Five Minute OAuth2 . . . 3
1.1 HTTP Client Requirements . . . 5
1.1 - OAuth2 API Consumer . . . 5
1.1.1 OAuth2 Client Toolkits . . . 5
1.1.2 Step 1 - Authorize . . . 6
1.1.3 Step 2 - Callback . . . 7
1.1.4 Step 3 - DATA! . . . 9
1.1.5 Step 6 - Refresh . . . 11
1.2 - OAuth2 RETS API Server . . . 12
1.2.1 OAuth2 Server Toolkits . . . 13
1.2.2 Register New API Consumers . . . 13
1.2.3 Authorize Endpoint . . . 14
1.2.4 Grant Endpoint . . . 15
1.2.5 Verify Access Tokens . . . 16
1.2.6 Extra Security Measures . . . 17
Section 2 - OAuth2 Implementation Recommendations . . . 17
2.1 Client Password Credentials . . . 18
2.2 Token Expirations . . . 18
2.3 Format and Construction of Tokens . . . 18
2.4 Redirect_uri Enforcement . . . 19
2.5 Refreshing an Access Token . . . 20
2.5.1 An expired access token returns HTTP 401 . . . 20
2.5.2 API Consumer makes a request to the RETS Server's authorize endpoint . . . 20
2.5.3 API Consumer saves the access and refresh tokens . . . 20
Section 4 - Authors . . . 20
Section 5 - Revision List . . . 21
Section 6 - Appendices . . . 22
6.1 Intended Use Cases . . . 23
6.1.1 SP to IdP to SP Typical three-way authorization . . . 23
6.2 Unsuppported Use Cases . . . 23
6.2.1 SP (Service Provider) to SP/IdP (Identity Provider) . . . 23
6.2.2 SP to SP/IdP Transparent three-way authorization . . . 24
6.2.3 SP to SP/IdP Transparent, recurring "on behalf of" authorization . . . 25
6.3 Explicitly Disallowed Use Cases . . . 26
6.3.1 2-legged Client-Server Auth . . . 26
6.3.2 4-legged Federated Identities . . . 27

Copyright 2014 RESO

RESO Web API Security v1.0.1

Copyright 2014 RESO. By using this document you agree to the RESO End User License Agreement (EULA) posted here.
(https://reso.memberclicks.net/assets/docs/reso%20eula.pdf)

Please review the document, RETS Web API Security - RESO Position, as a preamble to this RETS Web API Security v1.0.1 document.

Section 1 - Five Minute OAuth2

- 1.1 HTTP Client Requirements
- 1.1 - OAuth2 API Consumer
  - 1.1.1 OAuth2 Client Toolkits
  - 1.1.2 Step 1 - Authorize
  - 1.1.3 Step 2 - Callback
  - 1.1.4 Step 3 - DATA!
  - 1.1.5 Step 6 - Refresh
- 1.2 - OAuth2 RETS API Server
  - 1.2.1 OAuth2 Server Toolkits
  - 1.2.2 Register New API Consumers
  - 1.2.3 Authorize Endpoint
  - 1.2.4 Grant Endpoint
  - 1.2.5 Verify Access Tokens
  - 1.2.6 Extra Security Measures

Section 2 - OAuth2 Implementation Recommendations

- 2.1 Client Password Credentials
- 2.2 Token Expirations
- 2.3 Format and Construction of Tokens
- 2.4 Redirect_uri Enforcement
- 2.5 Refreshing an Access Token
  - 2.5.1 An expired access token returns HTTP 401
  - 2.5.2 API Consumer makes a request to the RETS Server's authorize endpoint
  - 2.5.3 API Consumer saves the access and refresh tokens

Section 4 - Authors

Section 5 - Revision List

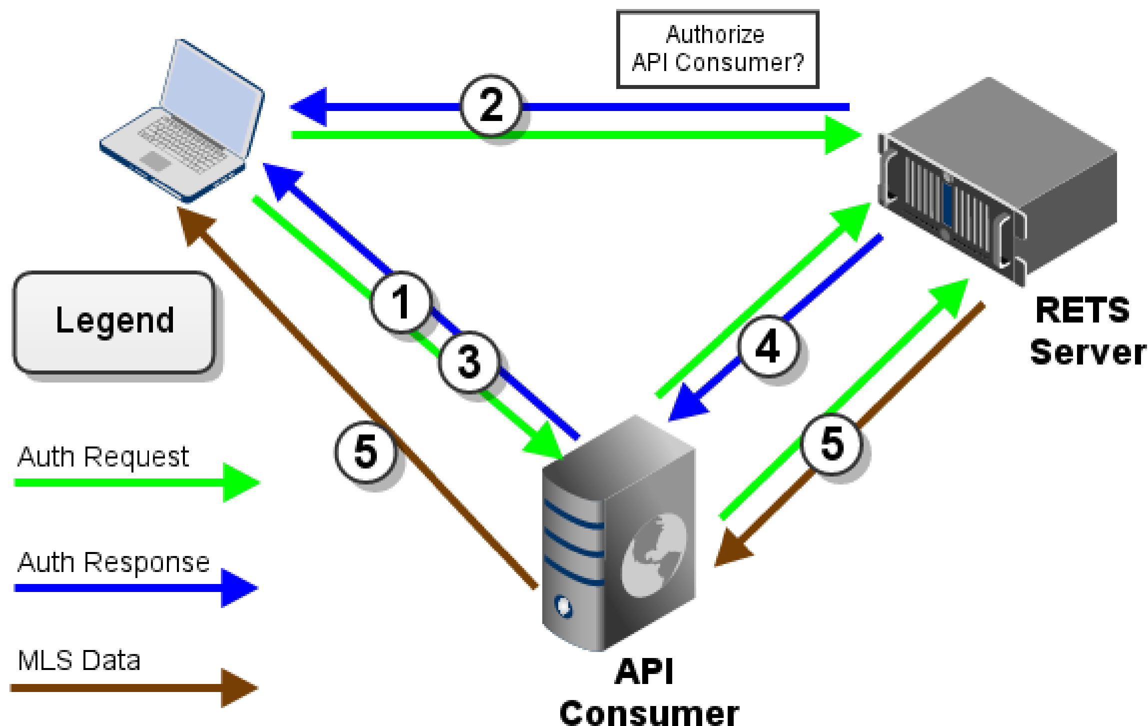
Section 6 - Appendices

- 6.1 Intended Use Cases
  - 6.1.1 SP to IdP to SP Typical three-way authorization
- 6.2 Unsuppported Use Cases
  - 6.2.1 SP (Service Provider) to SP/IdP (Identity Provider)
  - 6.2.2 SP to SP/IdP Transparent three-way authorization
  - 6.2.3 SP to SP/IdP Transparent, recurring "on behalf of" authorization
- 6.3 Explicitly Disallowed Use Cases
  - 6.3.1 2-legged Client-Server Auth
  - 6.3.2 4-legged Federated Identities

Section 1 - Five Minute OAuth2

OAuth2: too long; didn't read (TL;DR)

OAuth2 uses a 3-legged authentication scheme. The three legs are the **Client Browser**, **API Consumer**, and **RETS Server**. The Client can be an MLS Member, or VOW consumer. The API Consumer is a server-side application that consumes MLS Data and acts as a middle-man between the Client and the RETS Server. The RETS Server is operated by the Site/MLS. It services the OData endpoints, MLS Data, and operates as the identity provider.



The general flow in this process is:

1) An unauthenticated MLS Member requests access to the API Consumer. The API Consumer responds with a redirect to the RETS Server's OAuth2 `authorize` endpoint.

2) The MLS member passes a username and password to the RETS Server. They also agree to authorize the API Consumer access to their Site/MLS data. A `redirect_uri` parameter in this request is compared with a previously stored value on the RETS Server as a security check.

3) The Client browser is redirected back to the `redirect_uri` that was given in the request from Step 2. The API Consumer is also given an authorization `code` parameter for use in Step 4.

4) The API Consumer makes a request to the RETS Server's token exchange service. The authorization `code` received in Step 3 is exchanged for access and refresh tokens.

5) The API Consumer uses the access token to request Site/MLS data. The data is processed and presented to the Client browser.

Continue with the overview: Which leg are you writing?

- Section 1.1 - API Consumer
- Section 1.2 - RETS Server

For more in depth reading:

OAuth2 Resources: http://oauth.net/2/

OAuth2 RFC 6749: http://tools.ietf.org/html/rfc6749

OAuth2 Threat Model and Security Considerations RFC 6819: https://tools.ietf.org/html/rfc6819

Copyright 2014 RESO

1.1 HTTP Client Requirements

The HTTP Client Browser, and HTTP Client Library used in the API Consumer role **MUST** support these features:

- Ability to submit POST requests with a JSON message body
- Ability to submit custom headers, specifically the Authorization header
- Follow HTTP 302 redirects automatically
- Support SSL
- Display HTML content to a user

These requirements should be standard in every modern tool, and this is the bare minimum required to use OAuth2.

1.1 - OAuth2 API Consumer

The API Consumer accepts client browsers and uses an access token to retrieve data from the RETS API server. This access token is an obfuscated transient key, and represents the identity of an MLS Member or VOW consumer. An API Consumer is typically a standard MVC web application and is strictly a server-side middle-man between a Client and the RETS Server. This middle-man approach adds extra security features when compared to the standard client-server model.

An API Consumer must first apply for a registration process from the Site/MLS. The Site/MLS will provide a **`client_id`** and **`client_secret`**. The `client_id` is registered with a **`redirect_uri`**, which is a URL that points back to the API Consumer's callback controller.

The general algorithm an API Consumer must implement is:

Step 1: Accept an unauthenticated Client, and redirect to the RETS Server's **`authorize`** endpoint

Step 2: Accept a Client at the callback URL (registered `redirect_uri`). The RETS Server appends a **`code`** parameter to this call. Exchange the `code` for access and refresh tokens at the RETS Server's **`grant`** endpoint

Step 3: Use the access token in the `Authorization` HTTP header to request data from the API

Step 4: If an access token expires, use the refresh token at the RETS Server's `grant` endpoint to attain a fresh one

Four steps, that's not too bad! There are also many OAuth2 client libraries available which make the programming process quicker and easier.

A few security guidelines:

1. An API Consumer **MUST NEVER** give out access tokens, refresh tokens, or client_secrets. Treat these like a password!
2. The API Consumer **MUST** use SSL for the callback URL
3. Do not mix up client sessions with different access tokens. They are a 1:1 identity relationship

- 1.1.1 OAuth2 Client Toolkits
- 1.1.2 Step 1 - Authorize
- 1.1.3 Step 2 - Callback
- 1.1.4 Step 3 - DATA!
- 1.1.5 Step 6 - Refresh

1.1.1 OAuth2 Client Toolkits

The RESO role of an **API Consumer** may use an existing OAuth2 client library. OAuth2 client libraries are not required, they just make the programming task a little easier. Read 1.1 HTTP Client Requirements for the basic requirements.

Try to not confuse "OAuth2 client" with the RESO term for "Client" which represents the client web browser. The API Consumer is both a client and server. A client to the RETS API Server, and a server to the MLS Member.

Most of the client libraries listed here have been taken from the OAuth2 Implementations page here: http://oauth.net/2/ If your language is not listed here, please refer to the OAuth2 list.

PHP

PHP OAuth 2.0 "Authorization Code Grant" client
https://github.com/fkooman/php-oauth-client

Zend OpenID Connect (And OAuth2) Client Library
https://github.com/ivan-novakov/php-openid-connect-client

PHP OAuth API
http://www.phpclasses.org/package/7700-PHP-Authorize-and-access-APIs-using-OAuth.html

Java

Spring Security OAuth2
https://github.com/spring-projects/spring-security-oauth/wiki

Apache Oltu
http://oltu.apache.org/download.html
https://cwiki.apache.org/confluence/display/OLTU/Documentation

.NET

DotNetOpenAuth
http://dotnetopenauth.net/

Spring Social for .NET
http://www.springframework.net/social/

Ruby

Intridia OAuth2 gem
https://github.com/intridea/oauth2

Tiabas OAuth2 client
https://github.com/tiabas/oauth2-client

nov's rack-oauth2 client
https://github.com/nov/rack-oauth2

Python

rauth
https://github.com/litl/rauth

sanction
https://github.com/demianbrecht/sanction

Google APIs Client Library for Python
https://developers.google.com/api-client-library/python/guide/aaa_oauth

iOS

Client library for OAuth2
https://github.com/nxtbgthng/OAuth2Client

Android

In practice, the Java clients listed in this section should also work on Android.

Scribe

https://github.com/fernandezpablo85/scribe-java

Copyright 2014 RESO

1.1.2 Step 1 - Authorize

Accept an unauthenticated client request:

Client Request

```
GET / HTTP/1.1
Host: app.example.com
```

Respond with a redirect to the RETS Server's `authorize` endpoint. Provide the `client_id` and `redirect_uri` parameters associated with the API Consumer. A `state` parameter is an extra security measure, and is a unique session ID to assist in preventing cross-site forgery attacks.

Response

```
HTTP/1.1 302 Found
Location: https://rets.example.com/authorize?client_id=7d1wp67gl1oo8wsc8ks4csgsk&
          &state=o5n9ki8kpil86vl9j11uujbn41
          &redirect_uri=https://app.example.com/callback.php
```

PHP Example

A quick snippet in everyone's favorite language

index.php

```
<?php
$client_id = "7d1wp67gl1oo8wsc8ks4csgsk";
$callback = "https://app.example.com/callback.php";
if ( session_id() === "" && $_COOKIE[session_name()] == NULL )
{
  session_start();
  header("Location: https://rets.example.com/authorize?"
         . "client_id=$client_id"
         . "&state=" . session_id()
         . "&redirect_uri=$redirect_uri");
}
?>
```

1.1.3 Step 2 - Callback

Accept a client request that has been redirected from the RETS Server. The MLS Member has logged in, and is being redirected back to the API Consumer.

Client Request

```
GET /callback.php?code=5i46ka0uur7soktiyca6lcczt?state=o5n9ki8kpil86vl9j11uujbn41
HTTP/1.1
Host: app.example.com
Referer: https://rets.example.com/authorize
Cookie: PHPSESSID=o5n9ki8kpil86vl9j11uujbn41;     (Set from the example index.php)
```

Copyright 2014 RESO

Verify that the `state` parameter is the same as the current session ID to prevent cross-site forgery attacks.

Before responding to this client, open up a new server-side HTTP request to the RETS Server's token exchange service. Provide the **`client_id`**, **`client_secret`**, **`redirect_uri`**, and authorization **`code`**.

RETS Server Request

```
POST /grant HTTP/1.1
Host: rets.example.com
Content-Type: application/json

{"code":"5i46ka0uur7soktiyca6lcczt",
 "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
 "client_secret":"6pphytzx8qklfa2wi23wgiyil",
 "redirect_uri":"https://app.example.com/callback.php",
 "grant_type":"authorization_code"}
```

RETS Server Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{"access_token":"2w9wc3b8565ajpj4i9v68ivlv",
 "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
 "expires_in":3600}
```

- Save the access and refresh tokens in a protected storage space, referenced by the client's session ID. (`o5n9ki8kpil86vl9j11uujbn41`)
- Optionally, save the `expires_in` timestamp to know in advance when a refresh will be needed.
- If supported by the RETS Server, use the access token at this time to query an MLS Member's information for display purposes. (Name, email, etc)

Respond to the client with a redirect to the API Consumer's "logged in" landing location. SSL is not required after this step!

Client Response

```
HTTP/1.1 302 Found
Location: http://app.example.com/dashboard.php
```

PHP Example

callback.php

```
<?php
session_start();
$code = $_REQUEST["code"];
$state = $_REQUEST["state"];
if ( $state != session_id() )
{
   # Cross site forgery detection!
}
$ch = curl_init(); curl_setopt($ch, CURLOPT_URL, "https://rets.example.com/grant");
curl_setopt($ch, CURLOPT_HTTPHEADER, array('Content-Type: application/json'));
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode(array(
  "code":"5i46ka0uur7soktiyca6lcczt",
  "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
  "client_secret":"6pphytzx8qklfa2wi23wgiyil",
  "grant_type":"authorization_code"
)));

$response = curl_exec($ch);
curl_close($ch);

$response = json_decode($response);
$access_token = $response["access_token"];
$refresh_token = $response["refresh_token"];

# Calculate the timestamp a refresh will be needed at
$expires_at = strftime("%Y-%m-%d %H:%M:%S", time() + $response["expires_in"]);

# Insert something useful into a database
$sql = "insert into keys (session_id, access_token, refresh_token, expires_at) "
  . " values ('" . session_id() . "', '$access_token', '$refresh_token',
'$expires_at')";

# Redirect to our landing page
header("Location: http://app.example.com/dashboard.php"
?>
```

1.1.4 Step 3 - DATA!

I can haz data?

Copyright 2014 RESO

The access token can now be used in the `Authorization` HTTP header to request data from the RETS Server API on behalf of the MLS Member. Make sure to follow these security guidelines:

1. An API Consumer **MUST NEVER** give out access tokens, refresh tokens, or client_secrets. Treat these like a password!
2. Do not mix up client sessions with different access tokens. They are a one-to-one identity relationship

Client Request

```
GET /dashboard.php HTTP/1.1
Referer: https://app.example.com/callback.php
Cookie: PHPSESSID=o5n9ki8kpil86vl9j11uujbn41;
```

The request to the RETS Server API might look something like this:

RETS API Request

```
GET /RESO/OData/Properties.svc/Properties('ListingId3') HTTP/1.1
Host: rets.example.com
Authorization: Bearer 2w9wc3b8565ajpj4i9v68ivlv
```

PHP Example

Copyright 2014 RESO

dashboard.php

```
session_start();
# Retrieve this client's access token.  expires_at condition optional,
# the RETS Server will tell us when an access token is expired
$sql = "select access_token from keys where session_id='" . session_id() . "' and
now() < expires_at";

curl_setopt($ch, CURLOPT_URL,
"https://rets.example.com/RESO/OData/Properties.svc/Properties('ListingId3')");
curl_setopt($ch, CURLOPT_HTTPHEADER, array("Authorization: Bearer $access_token"));
$response = curl_exec($ch);
$status = curl_getinfo($ch, CURLINFO_HTTP_CODE);
if ( $status == 401 )
{
   // See Section 1.1.5 Step 4 - Refresh
}

# present $response to the client
```

1.1.5 Step 6 - Refresh

If a RETS Server responds from an API request with an HTTP 401 response, the access token is invalid and must be refreshed.

RETS API Response

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm='RETS Server', error='expired_token'
Content-Type: application/json

{ "message": "Access token has expired" }
```

The API Consumer uses the `client_id`, `client_secret`, `refresh_token`, and `redirect_uri` to retrieve a fresh set of access and refresh tokens for the MLS Member.

RETS Server Request

```
POST /grant HTTP/1.1
Host: rets.example.com
Content-Type: application/json

{"client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
"client_secret":"6pphytzx8qklfa2wi23wgiyil",
"refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
"redirect_uri":"https://app.example.com/callback.php",
"grant_type":"refresh_token",
}
```

Copyright 2014 RESO

RETS Server Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{ "access_token":"645nhg6ofaxunp2hfj0pou8r0",
"refresh_token":"3o0iipzrpiknijyxtjrugkt29",
"expires_in":3600
}
```

The new pair should be saved as a reference to the current session ID. Any old access and refresh tokens are invalid, and should be deleted.

PHP Example

Snippet of dashboard.php

```
$status = curl_getinfo($ch, CURLINFO_HTTP_CODE);
if ( $status == 401 )
{
  $ch = curl_init();
  curl_setopt($ch, CURLOPT_URL, "https://rets.example.com/grant");
  curl_setopt($ch, CURLOPT_HTTPHEADER, array('Content-Type: application/json'));
  curl_setopt($ch, CURLOPT_POST, 1);
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
  curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode(array(
    "refresh_token":$refresh_token,
    "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
    "client_secret":"6pphytzx8qklfa2wi23wgiyil",
    "redirect_uri": "https://app.example.com/callback.php",
    "grant_type":"refresh_token"
  )));
  $response = curl_exec($ch);
  curl_close($ch);
  $response = json_decode($response);
  $access_token = $response["access_token"];
  $refresh_token = $response["refresh_token"];
  # Calculate the timestamp a refresh will be needed at
  $expires_at = strftime("%Y-%m-%d %H:%M:%S", time() + $response["expires_in"]);
  $sql = "delete from keys where session_id = '" . session_id() . "'";
  $sql = "insert into keys (session_id, access_token, refresh_token, expires_at) "
    . " values ('" . session_id() . "', '$access_token', '$refresh_token',
'$expires_at')";
}
```

1.2 - OAuth2 RETS API Server

The RETS Server must implement four basic features:

1. Register new API Consumers
2. Authorize endpoint
3. Grant endpoint
4. Verify access tokens

A few security guidelines

Copyright 2014 RESO

1. All requests **MUST** be over SSL
2. API Consumers **MUST** be registered with a `redirect_uri` callback, and verified at the `authorize` endpoint

- 1.2.1 OAuth2 Server Toolkits
- 1.2.2 Register New API Consumers
- 1.2.3 Authorize Endpoint
- 1.2.4 Grant Endpoint
- 1.2.5 Verify Access Tokens
- 1.2.6 Extra Security Measures

1.2.1 OAuth2 Server Toolkits

Most of the server libraries listed here have been taken from the OAuth2 Implementations page here: http://oauth.net/2/ An OAuth2 server toolkit is not required, they just provide some helper methods to make an OAuth2 implementation quicker and easier.

If your language is not listed here, please refer to the OAuth2 list.

PHP

PHP OAuth2 Server
https://github.com/bshaffer/oauth2-server-php

PHP OAuth 2.0 Auth and Resource Server
https://github.com/php-loep/oauth2-server

PHP OAuth 2.0 Authorization Server (with SAML/BrowserID AuthN, with management REST API)
https://github.com/fkooman/php-oauth

Java

Apache Oltu
http://oltu.apache.org/download.html
https://cwiki.apache.org/confluence/display/OLTU/Documentation

Spring Security OAuth2
https://github.com/spring-projects/spring-security-oauth/wiki/oAuth2

JavaScript (Node.js)

Mark Lesswing’s OAuth 2.0 authorization code authentication strategy for Passport.

https://www.npmjs.org/package/passport-oauth2-code

Python

OAuth 2.0 Client + Server Library
https://github.com/NateFerrero/oauth2lib

Ruby

nov’s rack-oauth2 client
https://github.com/nov/rack-oauth2

.NET

DotNetOpenAuth
http://dotnetopenauth.net/

1.2.2 Register New API Consumers

A RETS Server **MUST** register an API Consumer `redirect_uri` callback with new `client_ids`. Usually this is a one-to-one relationship. One `client_id` represents one API Consumer's callback URL. This **MAY** be a one-to-many relationship, with restrictions documented in Section 2.4 Redirect_uri Enforcement. Each `client_id` has a `client_secret`, which is effectively a password in the same token format.

This **MUST NOT** be an automated process. The RETS Server **MUST** implement a method to require a human action by the Site/MLS to authorize a new API Consumer product. *No client_id vending machines!*

1.2.3 Authorize Endpoint

The `authorize` endpoint can be any URL name chosen by the RETS Server, and should be documented for the API Consumer. (This URL can be hidden from public view if desired) This URL **MUST** use SSL. The API Consumer will redirect a request from the Client browser to this endpoint:

Client Request

```
GET /authorize?client_id=7d1wp67gl1oo8wsc8ks4csgsk
    &state=o5n9ki8kpil86vl9j11uujbn41
    &redirect_uri=https://app.example.com/callback.php HTTP/1.1
Host: rets.example.com
```

The RETS Server must match a pre-registered `client_id` and `redirect_uri` pair with the request parameters. If the security check passes, generate a new `Authorization code`. (See Section 2.3 Format and Construction of Tokens) This `code` **SHOULD** expire in ten minutes. Read more about expirations in Section 2.2 Token Expirations.

Hint
Database systems that have automatic TTL expirations work great for this. MongoDB and Redis are good examples.

Redirect the Client browser to the `redirect_uri` with the `code` parameter appended to the URL. Relay the same `state` parameter the API Consumer provided to prevent cross site forgery attacks.

Copyright 2014 RESO

Client Response

```
HTTP/1.1 302 Found
Location:
http://app.example.com/callback?code=5i46ka0uur7soktiyca6lcczt&state=o5n9ki8kpil86vl9j
11uujbn41
```

PHP Example

authorize.php

```
$sql = "select redirect_uri from consumers where client_id='"  .
$_REQUEST["client_id"] . "'";
if ( $redirect_uri != $_REQUEST["redirect_uri"] )
{
   # Error response defined in OAuth2 RFC 6749 Section 4.1.2.1
}
$new_code = generate_token();
$sql = "insert into codes (client_id, code, expires_at) "
  . "values('". $_REQUEST['client_id'] . "', '$new_code', now() + 600)";
header("Location: $redirect_uri?code=$new_code&state=$_REQUEST['state']");
```

1.2.4 Grant Endpoint

The RETS Server's **`grant`** endpoint is responsible for authorization code key exchanges and refreshing access tokens. The `grant` endpoint can be any URL name chosen by the RETS Server, and should be documented for the API Consumer. (This URL can be hidden from public view if desired) This URL **MUST** use SSL.

All requests to the `grant` endpoint require a `client_id`, `client_secret`, and `redirect_uri` at a minimum.

Authorization Codes

API Consumer Request

```
POST /grant HTTP/1.1
Host: rets.example.com
Content-Type: application/json

{"code":"5i46ka0uur7soktiyca6lcczt",
 "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
 "client_secret":"6pphytzx8qklfa2wi23wgiyil",
 "redirect_uri":"http://app.example.com/callback.php",
 "grant_type":"authorization_code"}
```

First the RETS Server must validate the `client_id`, `client_secret`, and `redirect_uri`. Then verify that the `code` parameter matches with the `client_id`, and that it has not expired. If the verification is successful, generate new access and refresh tokens in a JSON response. The RETS Server **MUST** delete or invalidate authorization codes after an API Consumer uses them.

Copyright 2014 RESO

API Consumer Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{"access_token":"2w9wc3b8565ajpj4i9v68ivlv",
 "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
 "expires_in":3600}
```

Refreshing Access Tokens

API Consumer Request

```
POST /grant HTTP/1.1
Host: rets.example.com
Content-Type: application/json

{"refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
 "client_id":"7d1wp67gl1oo8wsc8ks4csgsk",
 "client_secret":"6pphytzx8qklfa2wi23wgiyil",
 "redirect_uri":"http://app.example.com/callback.php",
 "grant_type":"refresh_token"}
```

First the RETS Server must validate the `client_id`, `client_secret`, and `redirect_uri`. Then, verify the `refresh_token` is valid and pairs with the `client_id`. Although refresh tokens do not carry an expiration, they can be manually removed from the RETS Server if security tripwires are triggered. This effectively blocks the API Consumer from accessing data on behalf of the specific MLS Member.

If the verification is successful, generate new access and refresh tokens in a JSON response. The RETS Server **MUST** delete or invalidate old access and refresh tokens after an API Consumer uses them.

API Consumer Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{"access_token":"645nhg6ofaxunp2hfj0pou8r0",
 "refresh_token":"3o0iipzrpiknijyxtjrugkt29",
 "expires_in":3600}
```

1.2.5 Verify Access Tokens

On every API request, the RETS Server must verify that the access token has not expired.

API Consumer Request

```
GET /RESO/OData/Properties.svc/Properties('ListingId3') HTTP/1.1
Host: rets.example.com
Authorization: Bearer 2w9wc3b8565ajpj4i9v68ivlv
```

On success, return the OData response using the identity tied to this `access_token`

Copyright 2014 RESO

On failure, return an HTTP 401 to tell the API Consumer that the access token is invalid

Failure Response

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm='RETS Server', error='expired_token'
Content-Type: application/json

{ "message": "Access token has expired" }
```

1.2.6 Extra Security Measures

Although not defined in the OAuth2 RFC, there are a few extra security measures a RETS Server may implement for extra security

IP Address Accounting

Keep a history of all IP addresses an API Consumer uses. Most server-side applications should not cycle through IPs very often. If there is a sudden influx of many IP addresses seen from a given client_id or `access_token`, invalidate them.

User-Agent Verification

Along with a `redirect_uri` , register an API Consumer's User-Agent. On each API request, verify the requested User-Agent is the same. Return an HTTP-401, or possibly invalidate the client_id and access tokens. A less invasive approach would be to keep a history of User-Agents, and perform a similar algorithm to the IP address accounting.

Rate Limiting

Keep track of the request rate at the `grant` and `authorize` endpoints. Brute force attacks can be easily caught and disallowed with these two services. Rate limit API requests by IP address, access_token, and client_id.

Lower Access Token Expirations

Section 2.2 Token Expirations suggests an access token expiration time of less than 24 hours for production traffic. The lower the expiration time, the quicker access tokens must be discarded, which results in less time for an attacker to use a stolen access token.

OAuth 2.0 Threat Model and Security Considerations

Read RFC 6819.

Copyright 2014 RESO

Section 2 - OAuth2 Implementation Recommendations

2.1 Client Password Credentials

Client Password Credentials (OAuth2 Section 2.3)

The OAuth2 spec defines the option to use HTTP Basic authentication, or Client Credentials using a client_id and client_secret pair. RETS Server **MUST** use the Client Password Credentials. HTTP Basic is certainly easier to use, but it removes the fine grained access control that a client_id represents. Additionally, both methods use the same Authorization header. A RETS Server can only support one method with any single endpoint. Thus, the RESO standard is to use the Client Password Credentials.

2.2 Token Expirations

Access Token Expirations and Refresh Tokens

In OAuth2 RFC Section 1.4, an access token “may self-contain the authorization information in a verifiable manner (*i.e., a token string consisting of some data and a signature*).” In this use case, the access token behaves similar to a cookie that is self-signed. Altering the OAuth2 RFC, the RETS Server ***MUST NOT*** use self-authorizing access tokens. The access token ***MUST*** be stored server side, and verified during each client request. This protects against brute forcing a signature algorithm, and makes it easy to invalidate access tokens by removing the server-side record.

In OAuth2 RFC Section 10.3, an access token "***MUST*** *be kept confidential in transit and storage, and only shared among the authorization server, the resource servers the access token is valid for, and the client to whom the access token is issued*." In practice, there are circumstances where an access token might be leaked to an attacker. This is improbable in web applications, but is a rare possibility in native OS applications like mobile or desktop applications.

If an access token is stolen, the attacker could only use it for a limited amount of time until it expires. Keeping a short expiration is a balance between security, convenience, and speed. It is convenient to be able to use a single access token during development or testing for an extended period of time. And, an API Consumer application is a faster user experience if it doesn't have acquire a new access token frequently.

On a typical OAuth2 web application, refresh tokens are used less often than access tokens, and are therefore subject to fewer intrusion situations. If an access token expires once every 24 hours, then the means to steal a refresh token is a small time window once every 24 hours via sniffing, program tracing, or other real-time attacks. If a refresh token is compromised by an attacker, it is still only valid for the access token's expiry length of time. Each time the API Consumer requests a new access token, a replacement refresh token is created as well. This effectively revokes access to any previous refresh tokens.

Considering the details of the security implications of expiration times, the RETS Server ***SHOULD*** enforce an expiration time of less than 24 hours for mainstream production traffic. The expiration time ***SHOULD*** be no less than two hours to avoid unnecessary communication overhead. A few exceptions to this rule are development, testing and native applications, which could have longer expirations.

Additionally, the RETS Server ***SHOULD*** revoke access to the attacker's tokens prematurely if any sort of security tripwires have been broken. (Detecting source IP address anomalies, request rate soft-limit overages, User-Agent header anomalies, etc.) From a system administrator's standpoint, revoking tokens is much easier than attempting to block IP addresses. From the end-user's standpoint, it is more convenient than changing passwords or other security increases. The security problem is handled server-side and completely transparent to the user.

Authorization Codes

Following the OAuth2 RFC recommendation in Section 4.1.2, all authorization code tokens ***SHOULD*** have an expiration of 10 minutes. This code ***MUST NOT*** be used more than once. The authorization code grant is meant to be a short lived process. The client logs in, agrees to the approval step, and then the code is generated. After a single redirect, the code is relayed to the API Consumer, who then uses the code to exchange for access and refresh tokens. In most cases, the authorization code should live for only a few seconds.

Hint
Database systems that have automatic TTL expirations work great for this. MongoDB and Redis are good examples. RDBMSs can use triggers or cron jobs to remove expired tokens.

2.3 Format and Construction of Tokens

Format and Construction of Tokens

The format of access and refresh tokens are left fairly vague in the OAuth2 specification. In OAuth2 Section 4.2.2, "*The access token string size is left undefined by this specification. The client should avoid making assumptions about value sizes. The authorization server SHOULD document the size of any value it issues.*"

Here is the RESO standard for character set, length, and cryptographic entropy:

For a character set, base 36 **SHOULD** be used as the encoding alphabet. The case insensitivity of the alpha characters makes it easier to type or

verbally communicate when keys are manually generated for end users. Typing on a mobile device might also capitalize the first character, and base 36 protects against confusing problems associated with that use case. The length needs to be a balance between entropy and convenience. In psychology, chunking refers to the short term memory retention for bits of information in small groups. The magic number is about 5 to 9 bits of information that a human can remember in short term memory for about 20 to 30 seconds.

A good middle ground of cryptographic entropy is 128 bits (16 bytes). Encoded to base 36 results in a 25 character string. That would be 5 chunks, of 5 bits of information each. This is also the exact same length and encoding that Microsoft uses for product keys.

In terms of coding ease, here is an example in everyone's favorite language, PHP:

```
function generate_token()
{
  # Generate 128 bits of cryptographically strong
  # random binary data
  $rand_bytes = openssl_random_pseudo_bytes ( 16 );

  # Convert to hexadecimal
  $rand_hex = bin2hex($rand_bytes);

  # Convert to base36
  return base_convert( $rand_hex, 16, 36);
}
```

This returns a token string that looks like `c4xwv032sfks8so8s800scgo8`. To make that even nicer in print, it could be upper cased and chunked into 5 groups of 5 characters:

`C4XWV-032SF-KS8SO-8S800-SCGO8.`

2.4 Redirect_uri Enforcement

The OAuth2 RFC Section 3.1.2.2 states that "*the authorization server **SHOULD** require all clients to register their redirection endpoint prior to utilizing the authorization endpoint.*" It applies a **MUST** to public clients or any clients using the implicit grant type. This is a solid security method to protect against man-in-the-middle or phishing attacks. The RESO standard is changing this to a **MUST** for all requests to the authorize endpoint.

The only inconvenience to this might be that a separate client_id needs to be registered for every redirect_uri. This means a new registration for different API Consumer environments. (*Development, Alpha, Beta, Staging, Production, etc.*) It might require a little extra time on the RETS Server's behalf. But the benefit to this is that different access roles and expirations could be applied to the individual client_ids on the RETS Server. For example, a Development environment client_id might set the access token's expiration to 1 week to make it easier to experiment with. But the production client_id might set the expiration to something more strict, such as 2 hours.

To make this process more convenient, the RETS Server ***MAY*** allow multiple redirect_uris to be registered for each client_id. The RETS Server **MAY** also register the base URL containing AT LEAST the domain name of the API Consumer, and pattern match multiple redirect_uri parameters against the registered pattern. In OAuth2 Section 3.1.2.3, the RFC states that "*if multiple redirection URIs have been registered, or if only part of the redirection URI has been registered, the client **MUST** include a redirection URI with the authorization request.*"

For example, a base URI could be registered on the RETS Server, such as: "https://staging.test_app.example.com". Any redirect_uri parameters that contain that base domain name could pass the verification test.

Then URIs like this would match:

https://staging.test_app.example.com/test/callback

https://staging.test_app.example.com/test/callback2

But a phishing attempt like this would still fail the verification:

https://naughty_phisher.com/callback

`state` Parameter

OAuth2 RFC Section 4.1.1 states that the `state` parameter on the `authorization` endpoint is "*RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to*

Copyright 2014 RESO

the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12."

The RESO Web API Security document is placing the `state` parameter as a **MUST**, since it's simple to implement, and is a great security measure.

2.5 Refreshing an Access Token

2.5.1 An expired access token returns HTTP 401

An expired access token returns HTTP 401 Unauthorized on a given API request.

```
GET /my/listings HTTP/1.1

Host: rets.example.com

Authorization: Bearer 2w9wc3b8565ajpj4i9v68ivlv
```

Response:

```
HTTP/1.1 401 Unauthorized

WWW-Authenticate: Bearer realm='RETS Server', error='expired_token'

Content-Type: application/json

       { "message": "Access token has expired" }
```

2.5.2 API Consumer makes a request to the RETS Server's authorize endpoint

API Consumer makes a request to the RETS Server's authorize endpoint

The previously saved refresh token is used to request another access token. The client_id and client_secret pair are required (see Section UC 1.3 - Client Credentials) and redirect_uri (see ??) are validated before granting access.

Request:

```
POST /authorize HTTP/1.1
Host: rets.example.com
Content-Type: application/json


{"client_id":"1234",
"client_secret":"dedyhcrynzeza6vljncfn5mxj",
"refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
"redirect_uri":"https://test_app.example.com/callback",
"grant_type":"refresh_token",
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json


{ "access_token":"645nhg6ofaxunp2hfj0pou8r0",
"refresh_token":"3o0iipzrpiknijyxtjrugkt29",
"expires_in":3600
}
```

2.5.3 API Consumer saves the access and refresh tokens

API Consumer saves the access and refresh tokens for future use.

The new access and refresh tokens are saved in a database and used in subsequent RETS Server requests. Any old access and refresh tokens should be discarded.

Copyright 2014 RESO

Section 4 - Authors

Author	Company	Email
Matt Cohen	Clareity Consulting	matt.cohen@clareity.com
Cal Heldenbrand	FBS	cal@fbsdata.com
Mark Lesswing	NAR	
Matt McGuire	Corelogic	

Copyright 2014 RESO

Section 5 - Revision List

- First working draft: RETS Web API Security Group Google Document
- First RESO document: RETS Web API Security v1.0
- Current Version: RESO Web API Security v1.0.1

Copyright 2014 RESO

Section 6 - Appendices

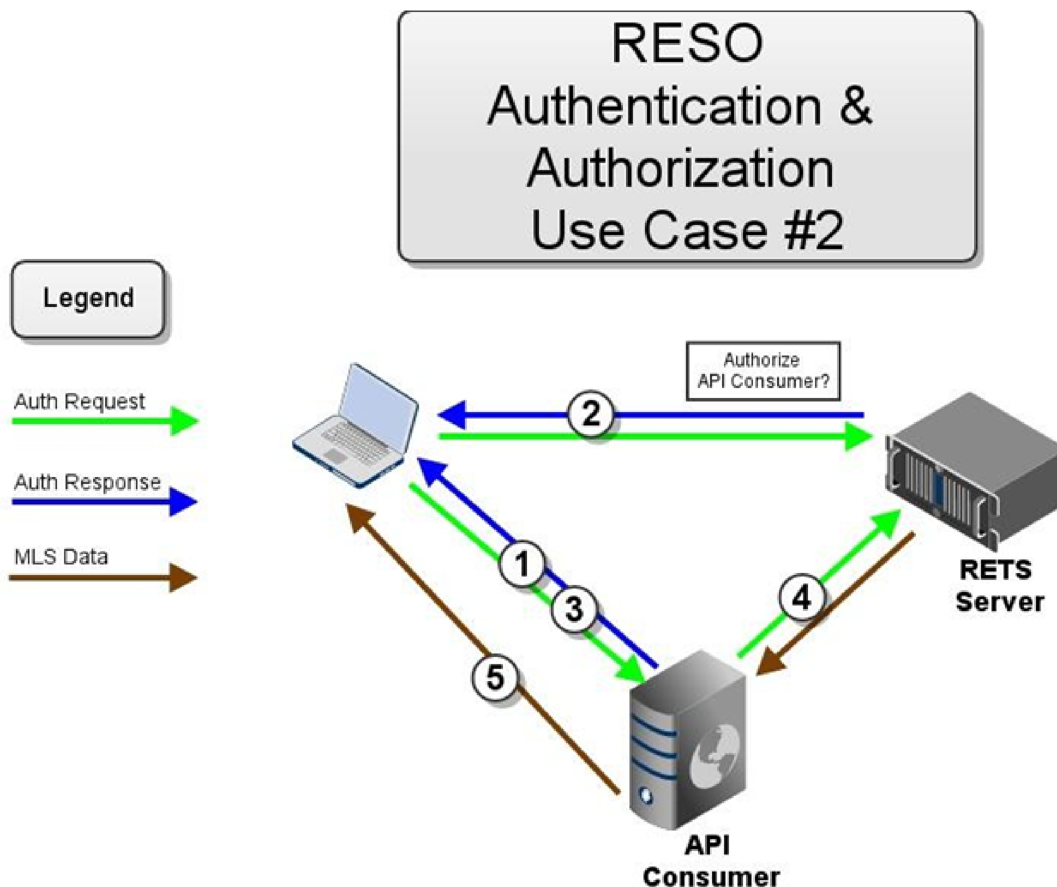
6.1 Intended Use Cases

The only case supported in this first version of the RESO RETS Web API Security standard is a real time application that works on behalf of a user against the OData Web API. The applications could be mobile, desktop, or web applications. The main attribute to this use case is that an end user requesting data will result in a real time query against the Web API, and present the data to the end user.

6.1.1 SP to IdP to SP Typical three-way authorization

Typical three-way authorization of a user (*Transient authentication of an API Consumer on behalf of an MLS member*).

Example: A web application that interacts with the MLS on behalf of a user, e.g., a real-time CMA.



1. An unauthenticated MLS member requests access to the API consumer. The API consumer responds with a failure redirect to the RETS server.

2. The MLS member enters a username / password at the RETS IdP. They also agree to authorize the API consumer product to access Site/MLS data..

3. The authenticated MLS member makes another request to the API consumer with valid authentication.

4. The API consumer makes a data request to the RETS server with the authentication supplied by the MLS member. The API consumer processes the data for presentation to the MLS member.

5. The API consumer responds with Site/MLS data to the MLS member.

6.2 Unsuppported Use Cases

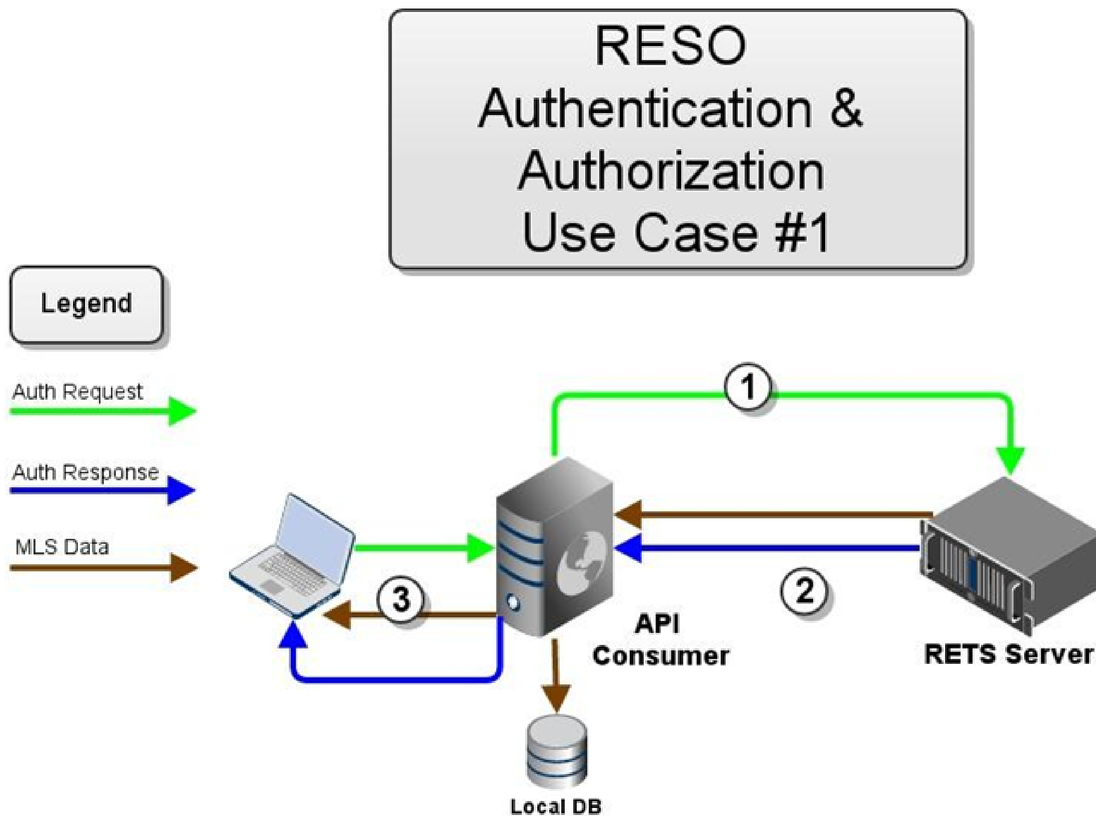
Unsupported use cases are possible to implement with OAuth2, but will be left out of the intended use case for the first version of the RESO API. As the API standard progresses, these edge cases will be worked back in.

Copyright 2014 RESO

6.2.1 SP (Service Provider) to SP/IdP (Identity Provider)

Server or Client to Server authorization (*without human intervention*)

Example: RETS 1.x style flow. A syndicator's recurring bulk download of listing data.



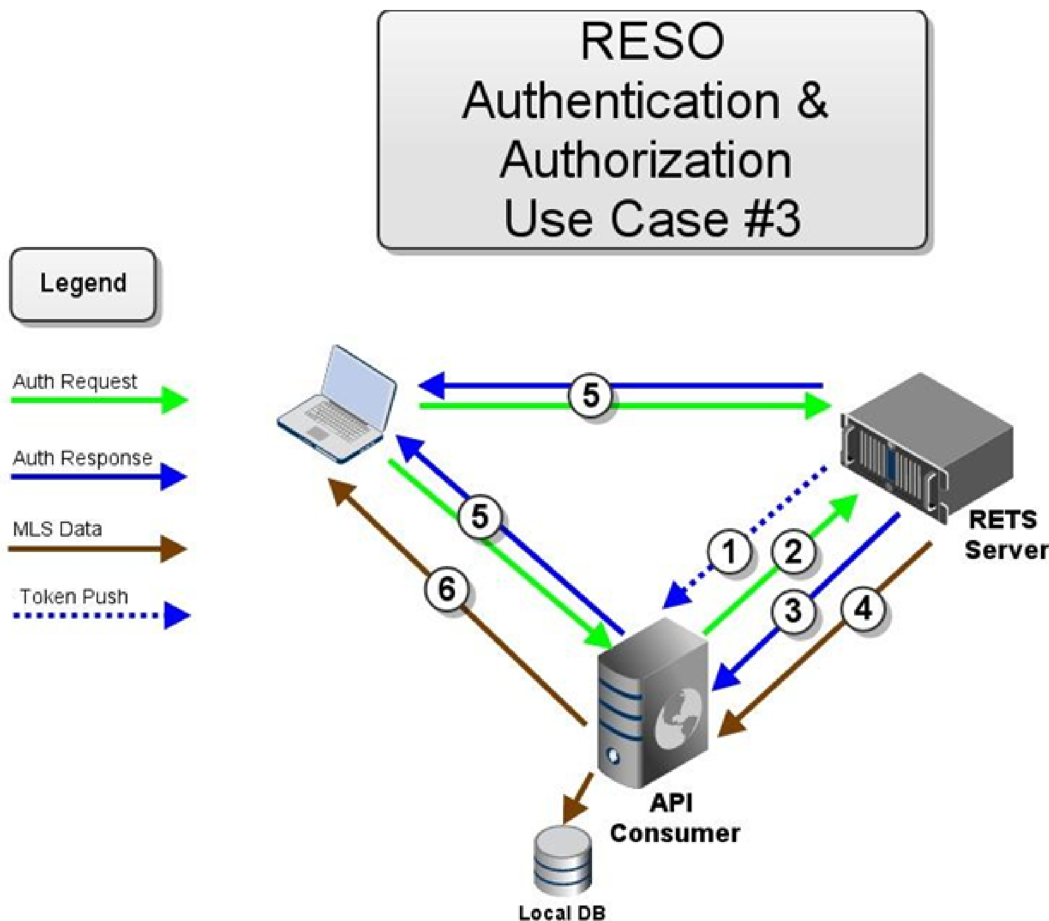
1. An API consumer submits a request for authentication to the RETS server. The API consumer declares its own identity to the RETS server (*Not on behalf of an MLS member*). No human interaction takes place.
2. The RETS server responds with an authentication success message along with any extra authentication session information. It may also respond with MLS data in the same response.
3. A web browser client requests MLS data directly from the API consumer. They may perform authentication using locally stored credentials, which may be independent of the MLS vendor's credentials. The request might also be unauthenticated for IDX sites.

6.2.2 SP to SP/IdP Transparent three-way authorization

Transparent three-way authorization of a user.
(Transient authentication of an API consumer on behalf of a user without human intervention)

Example: A VOW provider's validation of eligibility for an existing customer.

Copyright 2014 RESO



1. The Site/MLS (*or an MLS member*) gives an authentication token to the API consumer. This could be a manual process, or a batch process of tokens for many users. This token ***may*** have an expired lifetime.

2. The API consumer uses the token to request authorization from the RETS server. The API consumer could also request VOW authorization of a customer on behalf of an MLS member using that token.

3. The RETS server checks the token for authorization and expiration. They respond with a success, and possibly another (updated) token for this member. For VOW authorizations, this could respond with a token that identifies the MLS member's customer.

4. The API consumer requests Site/MLS data, same as Use Case 6.1.1. (*Use Case: SP to IdP to SP Typical three-way authorization*).

5. At some point in the future, the MLS member authenticates against the MLS, similar as Use Case 6.1.1. (*Use Case: SP to IdP to SP Typical three-way authorization*). The RETS server does not need to ask for authorization again, since this happened in step 1 (*VOW customers would have a similar authentication experience*).

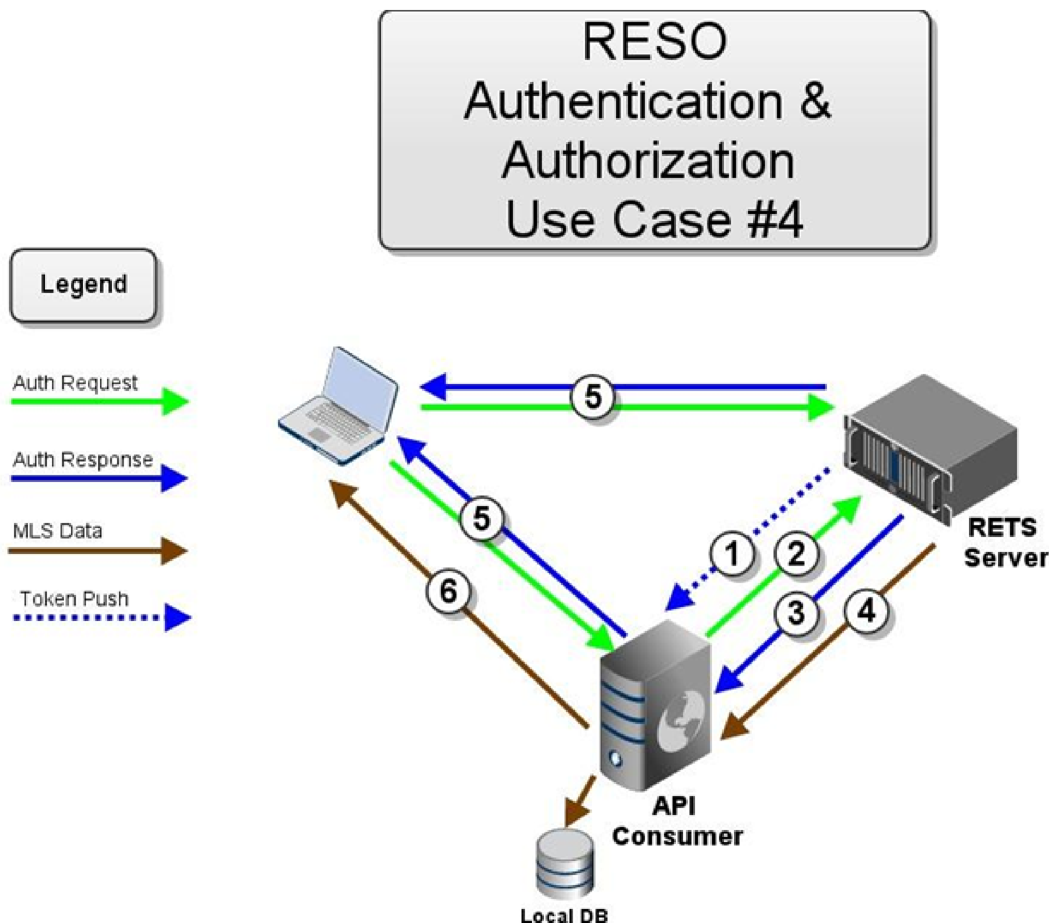
6. The MLS member or VOW customer has access to previously loaded Site/MLS data.

6.2.3 SP to SP/IdP Transparent, recurring "on behalf of" authorization

Transparent, recurring ""on behalf of"" authorization of a user.
(Persistent, transient authentication of an API consumer on behalf of a user without human intervention)

Example: Lead Management software that pulls leads from multiple sources for a given customer.

Copyright 2014 RESO



1. The Site/MLS (*or an MLS member*) gives an authentication token to the API consumer. This could be a manual process, or a batch process of tokens for many users. This token must have an infinite lifetime (*Or perhaps very long*).

2. The API consumer uses the token to request authorization from the RETS server. The API consumer could also request VOW authorization of a customer on behalf of an MLS member using that token.

3. The RETS server verifies the token. They respond with a success or failure. For VOW authorizations, this could respond with a token that identifies the MLS member's customer.

4. The API consumer requests Site/MLS data, same as Use Case #1.2.2 (*Use Case: SP to IdP to SP Typical three-way authorization*).

5. At some point in the future, the MLS member authenticates against the Site/MLS, similar as Use Case #1.2.2 (*Use Case: SP to IdP to SP Typical three-way authorization*). The RETS server does not need to ask for authorization again, since this happened in step 1 (*VOW customers would have a similar authentication experience*).

6. The MLS member or VOW customer has access to previously loaded Site/MLS data.

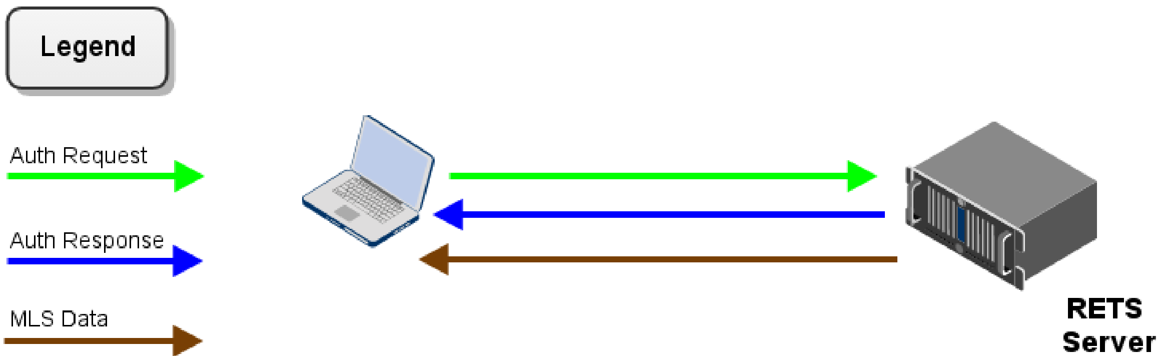
Note: The last two Use Cases are actually very similar - in terms of what standards may be called upon to service them, they are effectively the same.

6.3 Explicitly Disallowed Use Cases

6.3.1 2-legged Client-Server Auth

A client browser **MUST NOT** connect directly to a RETS API Server. While this was typical behavior in RETS 1.x, this authentication flow solicits too many security vulnerabilities with a modern API. OAuth2's Implicit Grant is a workaround to attempt to solve this problem. RESO is explicitly disallowing the OAuth2 Implicit Grant for the same reasons. The client browser should never have direct access to a client_id, secret, access token, or refresh token.

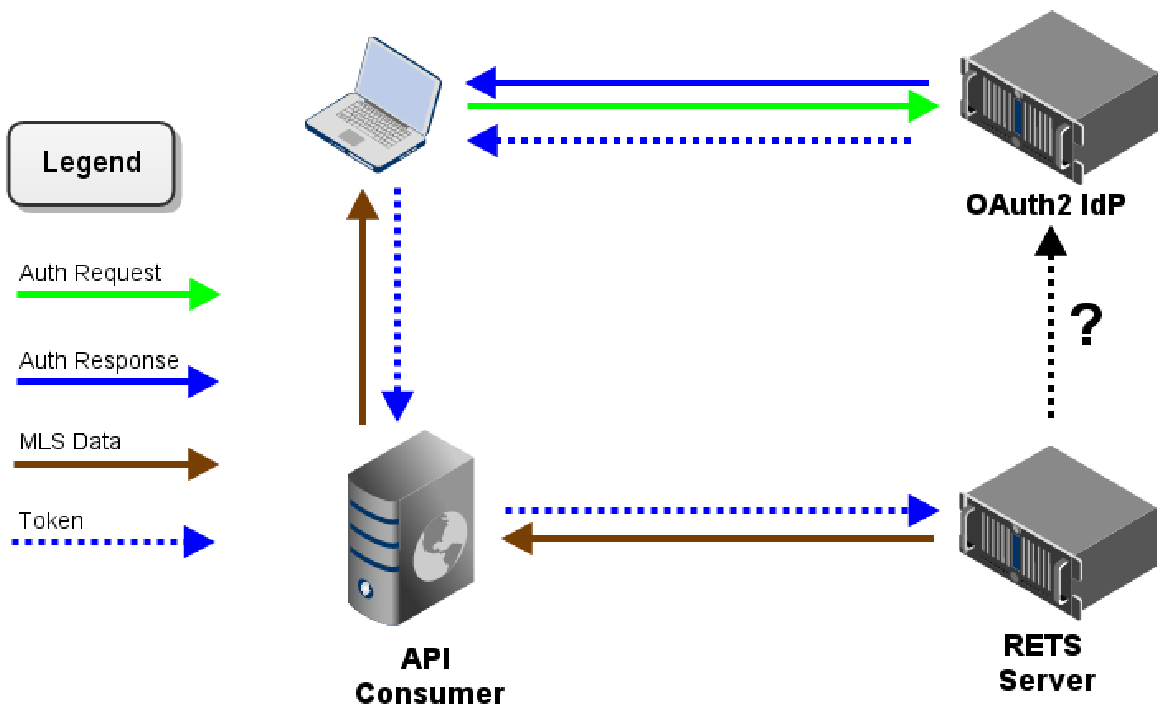
Copyright 2014 RESO



6.3.2 4-legged Federated Identities

Federated identity systems are out of the scope for this version of RESO Web API Security. From the very beginning, RESO decided that federated systems are too complex to tackle for the average developer. The complexity is introduced when an IdP is separated from the MLS's RETS Server. The two systems must maintain a constant communication in order to validate and invalidate identity sessions. Other solutions involve self-signed tokens and access control systems. Both of these solutions create a disconnect between the RETS Server and IdP, which results in the Site/MLS losing active control over Member's sessions.

Stay tuned in, as RESO may include support for OpenID Connect after this standard is adopted by the IETF as a published RFC.



Copyright 2014 RESO