Using OA4MP as a Dedicated Token Issuer

Overview

In this case, you have a possibly existing service that you want to start issuing tokens. The contract for your DTI (Dedicated Token Issuer) is that it gets requests from clients and it issues the tokens from OA4MP, which then have the entire abilities of any other OA4MP flow, such as refreshing or exchanging tokens, querying user information etc. OA4MP may be completely hidden from users if the DTI service so desires, by simply intercepting all requests and forwarding them. This can give enormous flexibility in use. A typical case is that a service requires tokens from a trusted issuer to access resources (e.g. computational, data) and, while running on behalf of a user, the user is not available. Think Condor with 10,000 submit nodes that needs many files for a single job from a file server. The user has started the job, but absolutely cannot manually approve each token. The DTI then services the requests.

How it works

The DTI (or others) can create standard OA4MP clients with credentials. Clients *must* be managed by an administrative client. The clients will make subsequent requests (which may involve using their credentials or signing them, however it is desired) to the DTI. The service will make the request *on behalf* of the user (and client) for a token. This a signed request the administrative client makes which returns the tokens which in turn are returned to the requester in the response. The client then takes over the flow.

Note there is a specific trust relation that allows the admin client using a signed RFC 7523 request to do this *without* having the keys or credentials for the client. The client, however, must authenticate all subsequent requests as per the specification. This means a client can set its own keys or credentials and its admin client does not have to manage them. All of OA4MP is still there and institutions can have admin clients with their own virtual issuers and clients as well so the topology of such a system may be arbitrarily complex.

Do I need the authorize and device endpoints?

No. The most basic version of this has those completely taken over by the DTI. However, the entire token request is done securely and is independent of all other forms of authentication. It is therefore possible to have a completely standard OA4MP server in addition to a DTI.

Prerequisites

In order to use this, you must set up your own DTI Service which will handle requests from users or clients. Typically they request will contain scopes, audience requests etc and your service must handle these.

1. The DTI itself.

2. The OA4MP server, which will function as the dedicated token service.
3. An administrative clients on the OA4MP server. These must have RFC 7523 keys and be set so that they can initialize flows. By *initialize a flow* we mean they make a signed request using RFC7523 §2.2 which makes the token request as per RFC7523 §2.1. This returns tokens. Note that depending upon the client configuration there are up to 3 tokens for ID, access and refresh. The admin client **must** be the administrator of the requesting client or the request will be rejected.
4. The client then uses the token and its credentials to make all of the standard requests to the OA4MP service for refresh, exchanges, user info, token introspection etc.

The essential idea is really simple. Create a dedicated client for your service, called a ***service client*** which uses RFC 7523 to communicate to OA4MP via a back channel. This client only communicates with signed requests.

Ways to show or hide OA4MP.

You have a DTI. It may either broker all requests to OA4MP, effectively hiding it.

Example: Hiding OA4MP with a reverse proxy lookup (Apache)

You are running, e.g. Apache. Your DTI, which is written in the language of your choice, resides at https://my_service.xyz.org/authorize Requests to Apache call this.

OA4MP is on the same machine running under Tomcat, and is restricted to localhost access only. It has endpoints

https://localhost:8080/oauth2/token
https://localhost:8080/oauth2/refresh
…

All requests go to Apache, which are rewritten to access Tomcat. The initial request is serviced by the AS and after that, requests for OA4MP endpoints are simply done by URL rewriting, so

https://my_service.xyz.org/refresh?…

would be converted to

https://localhost:8080/oauth2/refresh?...

and the response returned. This does give you the option of processing each request if needed.

Example: Coexisting with OA4MP

In this case, all standard OA4MP endpoints are available, you simply deploy your DTI (which must run under Tomcat, usually Java, though if you are really slick, Tomcat can run CGI and no, we don't cover how to do that here) to the standard endpoint, e.g.

https://my_service.xyz.org/dti ← your service
https://my_service.xyz.org/oauth2/authorize ← OA4MP
https://my_service.xyz.org/oauth2/device “ “
https://my_service.xyz.org/oauth2/token “ “
… “ “

(Remember that the token endpoint issues tokens, but is also used for *e.g.* refresh requests, so you still need it.)

Creating the token request to OA4MP

The details for using RFC7523 are here: https://oa4mp.org/pdf/rfc7523_intro.pdf. The authorization grant is for the administrative client and is signed with its key. The token request is *unsigned* and for the client ID, hence it has the client identifier set as the issuer (**`iss`** claim). This is because the server should not be storing the client's credentials and impersonating it, but rather there is a secure trust relationship that allows for initializing a flow.

Example

Create the authentication request for the admin client. In this case OA4MP is hidden and allows only localhost access with address **`https://localhost:9443/oauth2/token`**. The DTI has gotten a client request (whatever your service wants) which contains the information to make this token request. This request requires the private key for the admin client. Here the admin client has id **`admin:test/vo_1`** and is making the token request on behalf of the client **`localhost:test/initialize_flow`**.

Header:

```
{"kid":"563054FD9C2E418A","typ":"JWT","alg":"ES256"}
```

This uses the private key with kid 563054FD9C2E418A to sign the request.

Payload:

```
{
  "sub": "admin:test/vo_1",
  "aud": "https://localhost:9443/oauth2/token",
  "iss": "admin:test/vo_1",
  "exp": 1756158978,
  "iat": 1756158078,
  "jti": "admin:test/vo_1/rfc7523/n9b6SlQPoauQ0h_4DmU3UpFptoiglnVJKN7bQxObgwA"
}
```

which results in the signed JWT (the **`client_assertion`** parameter):

```
eyJraWQiOiI1NjMwNTRGRDlDMkU0MThBIiwidHlwIjoiSldUIiwiYWxnIjoiR...
```

Create the token request for the client:

Header:

```
{"typ":"JWT","alg":"none"}
```

Payload:

```
{
 "iss": "localhost:test/initialize_flow",
 "sub": "jeff",
 "jti": "localhost:test/initialize_flow/rfc7523/puGJcUBxSe6HawwCxyDRukGM",
 "exp": 1756158978,
 "iat": 1756158078,
 "nonce": "_0IyVynIJWys3TI1qmiCtaJF70u6X9rpgCxD8WjpwnI",
 "scope": [
 "read:",
 "write:",
 "org.cilogon.userinfo",
 "openid",
 "profile",
 "email"
  ]
}
```

The JTI is created/managed by the client and should be unique so it can be used to track the flow. This yields an unsigned JWT (the **`assertion`** parameter) :

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0. ...
```

And the full request (line breaks added) before encoding:

```
https://localhost:9443/oauth2/token?
client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer&
client_assertion=eyJraWQiOiI1NjMwNTRGRDlDMkU0MThBIiwi...&
grant_type=urn:ietf:params:oauth:grant-type:jwt-bearer&
assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lI…
```

The typical response then is

```
{
  "access_token": "eyJraWQiOiIyOTc4RkY1NDhBNTVBNzM5N...,
  "refresh_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0….,
  "scope": "read:/home/public/data/cern
write:/home/jeff/grant_76536789/cern/data email profile org.cilogon.userinfo
openid",
  "refresh_token_lifetime": 3600,
  "id_token": "eyJraWQiOiIyOTc4RkY1NDhBNTVBNzM5NTAyRTNCQzY0QTU4RTJC...,
  "token_type": "Bearer",
  "expires_in": 900,
  "refresh_token_iat": 1756206421
}
```

In this case the returned tokens and scopes simply follow the policy for the client. Remember that the scope is a blank delimited string. This is returned to the client as the body of the response and the client then uses it as needed.