Token handler configuration

Introduction

OA4MP supports configuration of token handlers. These will create various types of tokens, id, access, refresh and in various flavors too, such as WLCG access tokens.

Do you need one?

These handlers add functionality and support for various enhancements. If you did not configure any other, you would still get a basic, functioning client. An id token will be generated and access and refresh tokens as well. The id token is always a JWT, but to get JWTs for the other tokens, you need a handler to tell it which format and what basic information to put in.

Using QDL

If you need to extend your handler, then QDL, the policy language for OA4MP, can be used to do pretty much anything you need. Note that you should read the qdl server scripts document for more, since this is not a small topic.

Where do these live?

These are the contents of the **cfg** attribute in the client configuration. The contents are JSON, but the command line interface for client management accepts HOCON (which is a simplified superset of JSON and intended to be human-friendly).

On scope terminology

Some terminology needs to be centralized here relating to scopes. This is probably the most over-used word in OAuth. In requests, scopes are actually requests for specific claims in various tokens. In tokens, scope refers to

- meta data (such as the user's eppn in the id token).
- assertions of permissions or capabilities (such as a path in a file system)

I propose they replace it with the more vivid term, "kitchen-sink". In any case there are two broad types:

uri-scopes: Scopes of the form **scheme : path**. E.g. `storage.create:/home/users/bob`

non-uri-scopes: Scopes without a path. E.g. `compute.create`

Uri-scopes then are further broken down relative to another as super-scopes and sub-scopes:

scope = `read:/home/jeff`

superscope = `read:` or `read:/home`
subscope = `read:/home/jeff/other_stuff`

Uri-scopes are also used to fully qualify requests for metadata since they can effectively be mini-requests, *e.g.*, `cilogon:/user/eppn` might be a request to assert the user's EPPN (EduPerson Principal Name) in the id token. Many people also refer to subscopes as *scope reduction*. They do not have a good term for superscopes, so ours is a bit more accurate and we use it here.

Matching on uri-scopes is done with path components, so `read:/home/jeff1` and `read:/home/jeff` are considered unrelated – not super or sub scopes of each other.

Format

The format for these is given in HOCON, which is ingested directly in the CLI command line interface) or via the client management API. Generally these are converted internally to JSON.

```
{"tokens" :
   HANDLER+
   }

HANDLER:
IDENTITY | ACCESS | REFRESH


IDENTITY:
identity {
  BLOCK
  QDL?
 }

TOKEN_TYPE:
default | identity


ACCESS:
access {
  BLOCK
  QDL?
  TEMPLATES?
 }

TOKEN_TYPE:
default | wlcg | sci_token | access

REFRESH:
refresh {
  BLOCK
  QDL?
 }

TOKEN_TYPE:
default | refresh


BLOCK:
      "type":TOKEN_TYPE,
```

```
    "issuer":issuer
  "audience": audience
  "lifetime": lifetime (seconds)
       "id" : id
  "versions":[VERSION*]


QDL see qdl server scripts

TEMPLATES:[TEMPLATE+]

TEMPLATE:
{
   "aud": string | uri,
   PATHS?
}

PATHS:
[PATH+]

PATH:
{
   "op": string,
 "path": PATH_COMPONENT+
}

VERSION:
  "1.0"

PATH_COMPONENT:
 string  | ${claim}
```

Every type

The following are common to all handler configurations.

Attribute	Req?	Default	Description
create_ts	N	-	ISO 8601 format when this was created. Note that this is for you to help you track this configuration.
id	N	-	An identifier or description. This is ignored by the system but allows you to name or describe this for your own reference.
lifetime	N	-	The lifetime of the resulting token **in ms**. If this is not set, system defaults determined this. Note that server policies always are applied, so it is not possible to, for instance, have a lifetime for a token larger than the max allowed on the system. This lifetime supercedes any other in the configuration.
qdl	N	-	Block for QDL, OA4MP's policy language. The client must be granted permission to run/load QDL or this is ignored.
type	Y	-	The actual type the tokens created.
versions	N	-	A list of versions for which this handler applies. This is for the future if there are ever multiple versions since at this point there is only one, ["1.0"] and it may be ignored.

QDL scripts may be loaded, so see the documentation for what goes into the attribute.

Identity tokens

Supported token types: **default, identity**

What it does

This handler is charged with the creation of the id token. The lifetime attribute determines that **iat** claim. The only requirement is that the type be set to **identity** or **default**.

There are no attributes other than the default for the identity token handler

Do I need one?

Generally you only need to include this block if you want to set the lifetime to something specific or if you want/need other claims to be asserted, which can be done only in QDL, either as a code block (set the claims directly) or as a script. Again, see the documentation for QDL cited above which has several examples.

E.g.

Here is a complete configuration that only uses the id token

```
tokens{
 identity{
    type=identity
    lifetime = 3600000
    } //end identity token
  } //end tokens
```

In this case, the lifetime is set to 3600 seconds. All of the standard accounting claims will be added, such as **nbf**, **iat**, **exp** etc. You cannot add custom claims here. Use QDL for that.

E.g. Invoking QDL

```
tokens{
 identity{
    type=identity
    lifetime = 3600000
    qdl{
        load="fnal/fnal-idtoken.qdl"
        xmd={exec_phase="post_token"}
      }// end qdl
    } //end identity token
  } //end tokens
```

In this case, a QDL script is invoked in the post_token execution phase.

Access Tokens

Supported token types: **default, access, sci_token, wlcg**

What it does

If this handler is present, then a corresponding JWT is created for the access token. If this handler is missing, the default token (which is simply an opaque string) will be used with all server defaults.

Specific attributes

The following are specific to all access token handlers

Attribute	Req?	Default	Description
audience	-	-	The audience for this token. It may be a list or a single value
issuer	-	-	The issuer for this token.
resource	-	-	Resources for this token. Either a list or singleton
subject	-	-	Set the subject (templates allowed from user claims) for the access token
templates	-	-	Templates to be used to resolve scope requests.

Example of just an access token

```
tokens{
    access{
       type=wlcg
       issuer="https:cilogon.org"
       audience="https://wlcg.cern.ch/jwt/v1/any"
       lifetime=3600000
    } // end access token
  } //end tokens
```

Templates

A template is a pattern used to create a scope. These are aggregated by audience, so for a given.audience, a set of templates is available.

The `path` is a path to *e.g.* a file or other resource. These may include claims from the id token. Claims are accessed using template notation, so `${claim_name}` will substitute the given name.

There is a very special case of templates. If the claim name is an aggregate (such as a bunch of group memberships) then the template is resolved if the scope contains a component in one of the aggregates

Example

```
"templates": [ {
  "aud": "https://fnal.gov/serverA",
  "paths":   [
      {"op": "read","path": "/home/${sub}"},
```

```
      {"op": "write","path": "/home/${isMemberOf}/${sub}"}
   ]
  }
]
```

In this case, the read operation expects a scope like `/home/bob` and if the `sub` claim is `bob` then the claim will be asserted. In the write example, `isMemberOf` is a list of groups like

```
isMemberOf = ["bsu_all", "admin","staff"]
```

So a scope of `/home/admin/bob` would be asserted (since `admin` is on the list of groups and `bob` is the subject claim). A scope of `/home/students/bob` would not be asserted, since `students` is not in the list of groups for this person.

How are scopes converted to permissions?

This takes a bit of work, but in the client request, scopes may be anything as long as they are separated by spaces. So requests in the access token for reading a location like `/home/public/ncsa_all/bob` for a given resource would come through in a scope as

```
openid profile read:/home/public/ncsa_all/bob ... other scopes
```

for a resource of

```
https://ncsa.illinois.edu/access
```

In the configuration, a typical template might be

```
"templates": [ {
  "aud": "https://ncsa.illinois.edu/access",
  "paths":   [
      {"op": "read","path": "/home/${isMemberOf}/${sub}"}
   ]
  }
]
```

First, the audience (which may be any string, but generally uris are less ambiguous) is use to check what templates are there. Then the operation (before the colon) is found. There may be multiple templates for an operation. Now that a template has been found, the requested scope with have two matches done on it based on user claims. `isMemberOf` is normally a list of groups. As long as the person is in the group `ncsa_all` and has subject `bob` then the template matches the given scope and will be asserted. Any standard claim could be used.

Note that you do need an audience for your template in the configuration. If there is a single audience for your client though, you do not have to request it explicitly. If there are multiple audiences though and none is specified, an error is raised.

Edge case: no path in the template

What if you wanted to request a scopes like the following? These are properly speaking called *capabilities*.

```
compute.execute compute.cancel
```

In this case, there is not resource path. In that case just use a template like

```
"templates": [ {
  "aud": "https://ncsa.illinois.edu/access",
  "paths":   [
      {"op": "compute.path"},
      {"op": "compute.cancel"}
   ]
  }
]
```

QDL and templates

You may use QDL with templates. QDL will be processed (in case you need to update your claims) then templates will be processed. You may turn on or off template processing inside QDL by setting the do_templates flag on the flow to false. See the QDL scripting blurb for more details.

A common question

“Hey, I’d love it if we could have a template the uses part of a claim – like the first part of their EPPN, how do I do that?”

You cannot. That requires QDL. You can, however execute QDL statements (if your client is allowed to). Such a block of QDL to do this is

```
"qdl":{
     "code":"claims.my_id:=head(claims.eppn,'@');",
      "xmd":{"exec_phase":["post_token"]}
   } //end QDL
```

This creates a custom claim called my_id. Use that in your template. Generally though, if you really need to start using QDL, you should not just execute individual statements, since this makes the configuration very messy and hard to maintain.

The refresh token handler

Refresh tokens may also be issued as JWTs.

Supported token types: **default, refresh**

There is only the default handler at this point. Note that refresh tokens a *not* signed. There is simply the header and payload. This seems to be the standard that is most widely used.

```
  "refresh":  {
     "audience": "https://wlcg.cern.ch/jwt/refresh",
     "issuer": "https://refresh.cilogon.org",
     "lifetime": 3600000,
     "type": "default"
    }
   }}
```

A full example

This is an example from the test server that has a handler for each type.

The templates are intended to be small so that it is easy to see what is happening. Most real world examples of scopes tend to get very long.

```
{"tokens": {
   "access":  {
      "audience": "https://wlcg.cern.ch/jwt/v1/access",
      "issuer": "https://access.cilogon.org",
      "lifetime": 750019,
      "templates": [
         {"aud": "https://wlcg.cern.ch/jwt/v1/access",
          "paths":[
            {"op": "read","path": "/home/${sub}"},
            {"op": "read","path": "/public/lsst/${sub}"},
            {"op": "x.y","path": "/abc/def"},
            {"op": "x.z"},
            {"op": "write","path": "/data/cluster"}
           ]
         }],
      "type": "wlcg"
    },
    "identity": {
       "type": "identity"
       "lifetime": 2400000,
    },
    "refresh":  {
      "audience": "https://wlcg.cern.ch/jwt/refresh",
      "issuer": "https://refresh.cilogon.org",
      "lifetime": 3600000,
      "type": "default"
    }
   }}
```

Here is table of inputs (requested scopes) and outputs (returned scopes) for the above configuration. **E** in the table below refers to the endpoint used.

T = token endpoint

R = refresh endpoint

TX = token exchange endpoint.

Scopes are space-delimited. To make this all readable, each scope is on a separate line in the table below and rows are the correspondences between request and response.

#	requested scope	E	returned scope

1	`read:`
	
	`x.y:`
	`x.z`
	`write:`
		T	`read:/home/jeff`
			`read:/public/lsst/jeff`
			`x.y:/abc/def`
			`x.z`
			`write:/data/cluster`

2	`read:/home/jeff/data`
	`x.y:`
	`x.z`
	`write:/data/cluster/ligo`
		T	`read:/home/jeff/data`
			`x.y:/abc/def`
			`x.z`
			`write:/data/cluster/ligo`

3	`read:`
	`x.y:`
	`x.z`
	`write:`
		R/TX	
			
			`x.z`

4	`read:/home/jeff/data`
	`x.y:`
	`x.z`
	`write:/data/cluster/ligo`
		R/TX	`read:/home/jeff/data`
			
			`x.z`
			`write:/data/cluster/ligo`

5	`read:/home/jeffy`
	`x.y:/abc/def/ghi`
	`write:/data/cluster1`
	`x.z:/etc/certs`
		R/TX	`x.y:/abc/def/ghi`

6	`read:/home/bob`	T	An error, since no scopes can be asserted. Also means R and TX are impossible.

At all times, the templates are resolved, so in the above, references to `${sub}` are replaced in all endpoints.

The token endpoint will treat superscopes as requests for the corresponding scope. So in the first example, a superscopes of `write:` is sent and the server responds with `write:/data/cluster`.

The refresh and token exchange endpoints, however, generally will not treat superscopes as requests, but will try to match. The use pattern is the initial request can query for possible scopes and refreshes and exchanges may reduce the scope. In all endpoints, subscopes will be resolved.

Commentary

1. Only requests are made. Templates are resolved and the values returned.
2. Two specific subscopes are requested plus a query.
3. Same request to the exchange endpoint as 1, but requests are not serviced, hence only a single scope can be asserted
4. Same request as 2 to the exchange endpoint. The single query is ignored.
5. Request to the exchange endpoint with a single valid subscope. Since scopes are resolved by path, a scope of `read:/home/jeff` does not give access to `read:/home/jeffy`, but it would give access to `read:/home/jeff/y`
6. Attempt to get a permission that is not granted to the user. Such things are not asserted (so no error) unless *nothing* can be asserted, in which case, an error is raised.