Token handler configuration

Table of Contents

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. Every value can be set in a QDL script.

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).

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+
("extensible": boolean)?

}

VERSION:
  "1.0"

PATH_COMPONENT:
 string  | ${claim}
```

Note on using a proxy server

If you are using a proxy for authorization, then be advised that the proxy may need to be configured to get the right user claims from the proxy server and pass them back. Setting up a handler for the identity token on your server might not be the right place, so you may need one for the proxy client. OA4MP will not forward your handlers to the proxy server! You must set that up separately following the policies there. E.g. if you have a minimal OIDC proxy client (only returns the **sub** claim), then that is all you will have available. You may, of course, write your own handler with its own additional claim sources to run, but a common caveat is to be sure you know where the user claims reside and have the appropriate access.

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** or units. 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 supersedes 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.
subject	-	-	Set the subject (templates allowed) for the token[1]
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.

Server Constants

You may specify a few constants in any of the issuer, resource, audience or subject entries in access and refresh tokens. These are of the form ${*name*} and are replaced *in situ.* The values allowed are

Name	Description
client_id	The current client_id
eppn	The EPPN (if present, otherwise nothing will be asserted).
eppn_2	The EPPN up to the first stop character.
host	The current host
now	The current time in milliseconds
now_iso	The current time as an ISO 8601 string
now_sec	The current time in seconds.

The aim of server constants is that certain values that cannot be known ahead of time can be specified. Note that ***all*** claims in the ID token are available to be asserted as well. So if you knew that the IDP asserted a pairwise_subject claim, you could use ${pairwise_subject} as a variable. If no such claim exists, then no substitution will be made, vs. throwing an error.

An example of and access token configuration using server constants.

```
tokens{
    access{
       type=access
       issuer="${host}/physics"
       audience=["${client_id}/v1", "${client_id}/v2"]
       subject = "${eppn_2}"
       lifetime=3600000
```

1 It may seem odd to be able to set the subject in the identity token, however not every client has a specific login. In particular, services may use the client credentials flow in which case there is no user per se. It may make sense to set the subject to something fixed, e.g. **`ncsa-robot.`**

```
    } // end access token
  } //end tokens
```

The subject will be the eppn up to the stop character (“@”), so if the eppn is `"bob@bgsu.edu"`, then eppn_2 is just `"bob".` This assumes that the eppn is known to be well-defined for this transaction. The issuer will be created from the current host, and the audience will have two claims created from the current client’s identifier.

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.

How are issuers determined?

The hierarchy in OA4MP is as follows:

1. Any value set directly in QDL.
2. This configuration
3. The issuer attribute for the virtual organization (if this client belongs to one of those)
4. The issuer attribute as set in the admin client
5. The issuer attribute in the client configuration (there is an issuer attribute in the client itself which is rarely used.)
6. The issuer attribute as set in the server configuration
7. The address of the server

So if you fail to set the issuer in this configuration, you can see where the value is determined as per above.

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 = "1 hr"
    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. Again, the lifetime is set to 3600 sec = 1 hour.

Access Tokens

Supported token types: **default, access, sci_token, wlcg, rfc9068**

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
templates	-	-	Templates to be used to resolve scope requests.

How are issuers determined?

The hierarchy is

1. Value set directly in QDL
2. This configuration attribute
3. The value in the virtual organization at_issuer attribute
4. The server default issuer
5. The address of the server

How is the audience determined?

1. Value set from QDL
2. Value set in this configuration

Note that no value set means this is not asserted. The only exception i

e of refresh tokens etc. To be frank, these tend to be a mess, but that is how most specifications that use them are written.

Templates

A *template* is a fixed pattern used to create a scope. They may model static or dynamic permissions. See the associated blurb on scopes. 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.

Which template?

If there is a single template, that is used without any further ado. If there are multiple templates and the client has an audience configured, that will be used, unless you override it by passing in the **`audience`** parameter. Using the **`audience`** parameter will also set that as the audience in the access token. You may also explicitly use a template by passing in the parameter `org.oa4mp:/template=`*`name`* .

A special case is SciTokens which allows for the audience to be specified using a scope of the form

`aud:NAME`

which you may use as an alternate to the org.oa4mp parameter. If both are sent, the org.oa4mp parameter is used. Intabulated

Name	Value	Effect
`audience`	Single, value of the audiemce	Which template to use, overrides the audience in the configuration.
`org.oa4mp:/template`	Multiple, values of audience	Which template to use. Does *not* alter the audience in the token
`aud:NAME`	(as a scope) single values, but	(Scitokens only) identical to

	multiple occurences will be aggregated.	audience parameter, except multiple values may be sent.

Group resolutions

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 templates 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.

Static permissions: no path in the template

These are called *entitlements* by some, but are simply flags that are present or not. In this case, there is not resource path. In that case just use a template like e.g.

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

Using QDL and templates

You may use QDL in conjunction 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.

You *cannot* set templates from QDL. These are in the configuration for the client.

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.

When to use templates or not

Templates are used when there is an identifiable and relatively static set of patterns to capture. If that is not viable, then QDL should be used as the policy language.

Example

Let us say that you have individual permissions set per user that will be used based on the group memberships of that user. In that case literally every user would have their own set of templates which is frankly impossible to maintain. QDL would then be used. Automatic downscope tests (applied for templates) cannot work here since the system has no idea what are supposed to be static or dynamic permissions. In such cases you should apply these yourself using the **`downscope()`** function in QDL.

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 are *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"
  }
 }}
```

Templates in scripts

A common situation is that templates are stored in an outside source (such as LDAP, Kerberos or many others) and are retrieved on a per-user basis, possibly with imposing requirements, such as group membership, institutional affiliations and many more. Discussing that is beyond the scope of this section, though QDL handles all of these. The case we want to focus on is what to do once you have your templates. Here is a quick rundown of the most useful calls in QDL.

`template_substitution(templates., claims.)` - substitute the values in claims into the templates.. Note that this returns a potentially very large list of possible values to check, especially if the template corresponds to a stem. (e.g. `${isMemberOf}`)

`downscope(allowed_scopes., requested_scopes.)` - takes the requested_scopes and checks as downscopes for the allowed scopes. This returns a list of scopes.

`query_scopes(allowed_scopes., requested_scopes.)` - Takes the requested_scopes and if they are superscopes, returns the correct resolved template and if they are subscopes, checks they are allowed.

`to_scope_string(x.)` - takes the list of values in x (such as scopes) and turns it into a blank-delimited string.

```
allowed_scopes. := resolve_templates(allowed_scopes., requested_scopes., false);

permissions. := resolve_templates(permissions., requested_scopes.,
exec_phase=='post_token');

access_token.'scope' := detokenize(unique(permissions.), ' ', 2); // turn in to
string, omit duplications, trailing space

s_rec. := template_substitution(record.EPE., claims.);
say('*** FNAL eta. post subst =' + to_string(s_rec.));
// Get any specific overrides.
eta. := resolve_templates(s_rec., scopes., true);
```

Examples.

There are two quite complete examples in the sequel. The second is longer since it is actually an only slightly modified live configuration.

Ex. #1. A full example for each handler type.

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. Throwing an exception in such cases is specific to the token type, wlcg. A token type of `default` does not do this, *e.g.*

Ex. #2. A complete template example, modeled after a real configuration.

Scopes in the initial request

So a typical set of scopes in an initial request may look like this (each one is on a separate line, but in the request itself they would be separated by blanks e.g. “openid email profile ...”

Scope	Type	Comment
`openid`	MD	Marks protocol as open id
`email`	MD	request for user email
`profile`	MD	request for user profile (such as user preferred name)
`org.cilogon.userinfo`	MD	Gets additional metadata, such as user EPPN, affiliation
`wlcg.capabilityset:/bgsu`	DIR	Request templates for this institution based on (here) user IDP, affiliation, roles and group membership
`wlcg.groups:/bgsu`	DIR	Request roles and group membership based on IDP EPPN
`compute.cancel`	CAP	Specific request
`compute.create`	CAP	“
`compute.delete`	CAP	“

storage.read:/bgsu/users/bob/data	P	“
storage.create:/bgsu/users/bob/data	P	“
storage.read:/home/lsst/data/2022-12	P	“

MD = user meta data DIR = directive CAP = capability P = permission

What does all this mean? For one thing, user metadata is used with directives to search for the groups and other capabilities for a specific user (in this case, in an LDAP over a secure connection).

Templates (from LDAP)

Based on policies set by the institution, An initial lookup for groups and roles is done, then based on that the actual set of templates is returned for the capabilities and permissions:

```
compute.cancel
compute.create
compute.modify
storage.read:/bgsu/${user}
storage.write:/bgsu/${user}
storage.create:/bgsu/${user}
storage.read:/home/lsst/data
storage.read:/home/ligo/data
```

Again, these are the complete possible set of permissions and capabilities. The scopes requested by the user then have the templates applied to them. Note that user requested **`compute.delete`** but that is not in the templates list, so it will not be asserted. (The general policy for access tokens is that a scope request that does not result in an asserted value is ignored, but if ultimately no values are asserted, then an error is raised.)

The resulting claims

ID token: Several though not necessarily all. Policy may require access to lots of user information (EPPN, affiliations), but not return any of it because of privacy concerns or anonymization requirements.

Access token: These are put into the scope claim:

```
compute.cancel
compute.create
storage.read:/bgsu/users/bob/data
storage.create:/bgsu/users/bob/data
storage.read:/home/lsst/data/2022-12
```