Scopes

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 requests for claims in various tokens***. People use the term scope (for the request) interchangeably with the response to the scope (properly called a claim). It does not help that the claim itself is labeled as **`scope`** in many places (such as in a SciToken), In tokens, scope requests refer to

- **metadata**
  E.g. **`email, profile`** for a user's email or eppn in the id token.
- access **permissions**
  E.g. **`read:/igwn/data/2021`** for access to a resource. These are usually of the form **`action:path_to_resource`**
- **capabilities**
  E.g. **`compute.modify`**, **`mysql.read`** for use of a component (which may have its own access policies too). These are effectively flags – they are there or not.
- processing **directives** which will be parsed as requests for specific actions, but are generally not extensible (see below for definition).
  E.g. **`wlcg.capabilityset:/duneana`** which is a directive to a claim source about which permissions, capabilities etc. to assert for the user.
  E.g. **`igwn.robot:ligorobot`** which is a directive that the request should be processed as if handled by a robot. This means the user starts the flow to be run as the robot, and the robot name (here **`ligorobot`**) has its "user" information retrieved and returned in the tokens.

Directives and capabilities may also have side-effects such as adding claims to the identity token and refresh tokens as well.

I propose they replace "scope" with the more vivid term, "kitchen-sink," but this does not seem to be catching on.

Basic categories of templates

In any case there are two broad categories of scope templates,

- ***fixed***. These include all directives, capabilities and user meta data. These do not change and may not be down/up-scoped (see below for definition of this term).
- ***extensible***. generally these are uris. These include all permissions. These have paths and may have additional path components added.

Semantics of scopes

Aside from the function, there is the practical issue of what the semantics are.

- ***uri-scopes***: Scopes of the form **scheme : path**. E.g. **`storage.create:/home/users/bob`**
- ***simple scopes***: Scopes that are not uris. E.g. **`compute.create`**. These are effectively opaque strings.[1]

Uri-scopes *may* then be extensible. It is policy as to what is fixed or not and there is no *a priori* method to determine which is which. Since URIs have a regular format and are easily parsed, many institutions use them for everything, which is a good idea.

Handy table relating these concepts:

	fixed	**extensible**
uri	Y	Y
simple	Y	N

Subscope, downscope, scope reduction

Many people also refer to subscopes as *scope reduction* and refer to super scopes "upscopes" and subscopes as "downscopes". The semantics for dealing with subscopes is by path component, so **`read:/home/jeff1`** and **`read:/home/jeff`** are considered unrelated – not super or sub scopes of each other, but **`read:/home/jeff/data`** is a sub-scope of the latter. Subscopes policies are often applied in certain types of tokens if at all possible.

Initial requests

In the authorization code flow, requests for scopes are allowed. This means that a superscope is sent and the server resolves it to the actual scope that may be requested.

Example

The initial request for a service include the scopes

`read:`

`write:`

The response includes (for instance) the scopes

`read:/home/user/bob`
`read:/home/lsst/data`
`read:/home/ligo/data`

`write:/home/user/bob`

1 Properly they should be a type of URN, in that they name something and are immutable, but they do not follow that syntax. Think of house numbers. A house number is a number, but nobody is going to start adding them because it is understood they are not for computation. So the *policy* regarding these is that they are immutable. This adds another layer of confusion since they look as if you should do something with them but you will never figure that out without context.

Indicating that these are the configured templates for these users. In the next leg of OAuth, the user can subscope these.

E.g. of an extensible uri-scope:

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

E.g. of fixed uri-scope

`mysql:/read`

Attempts to subscope it should probably be flagged as an error.

A complete 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
```

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);
```