Creating claim sources

What is a claim source?

A *claim source* is a place to get metadata (called claims in the OIDC world) about a user. There are many ways of doing this, though the most common is to access an LDAP server. Each of the types of sources below gives a way to get user metadata.

Lifecyle

The server may (and very probably does) have a default set of claim sources. These will be accessed every time a request is made to the server. You may add your claims sources to this list say in the pre_auth phase and never have to worry about it being called. The downside is that everything returned by your claim source will always be added to the claims returned to the user.

You can also directly get claims and manage them. This is a common occurrence in some cases, such as user metadata including groups is gotten, then another set of claims is requested based on those.

Handler types

Types

There are 6 general types of claim sources allowed.

1. LDAP. You may specify any LDAP server and supply credentials for it.
2. HTTP headers. If your server passes claims in the headers of the initial request, you may harvest them.
3. File. If your server stores information about, say, users in a file system
4. NCSA default. Since this was created at the *National Center for Supercomputing Applications* there is a specific source for that. Since you must be in the NCSA's VPN to access it, this is of limited utility outside of the organization.
5. Code. You *may* write any Java class that extends basic claim source and reference that. Generally you do not need to do this except in very, very specific cases and should use any built-in claim sources if at all possible.
6. RESTful calls

Basic construction of a claim source

The basic steps are

1. Create a script to hold whatever you write. Everything goes in there. Of course, this being QDL and all, you can run everything
2. Create a configuration stem. You probably want to use the **`claims#new_template(type)`** function.
3. Populate specific attributes you need. Each type below has the specifics
4. Issue a **`claims#create_source`** call. This will tidy up any missing essential information based on the type
5. Either add it to the system claims, or invoke it with **`claims#get_claims(config., username)`** and manage it yourself.

Development tools

Be sure you have the following modules loaded in your QDL workspace. This should be in the configuration:

```
<modules>
            <module type="java"
                    import_on_start="true">
                <class_name>edu.uiuc.ncsa.myproxy.oa4mp.qdl.OA2QDLLoader</class_name>
            </module>
            <module type="java"
                    import_on_start="true">
                <class_name>edu.uiuc.ncsa.oa2.qdl.QDLToolsLoader</class_name>
            </module>
</modules>
```

What does this do to the workspace? Once QDL is up, you'll see it added several test variables:

```
   )vars -m
jwt#test_audience.  jwt#test_claims.  jwt#test_scopes.  jwt#test_xas.
```

These are samples for audience, claims, scopes and extended attributes.

There are also several modules added

```
  )modules
oa2:/qdl/acl [acl]
oa2:/qdl/jwt [jwt]
oa2:/qdl/oidc/claims [claims]
oa2:/qdl/oidc/client/manage [cm]
oa2:/qdl/oidc/token [tokens]
```

(If you have other modules, they will show up here too.)

You can print out help for one of the modules e.g. as

```
   )module  claims -help
// tons of stuff
```

And from the modules, there are a lot of functions. To list all of the functions (fully qualified by module, limiting display with to 72 chars)

```
     )funcs -fq -m -w 72
// tons of stugg
```

Will spit out the lot of them, but here is the breakdown.

ACL functions – not needed, but loaded if you are doing access control

```
acl#acl_add(1)
acl#acl_check(0)
acl#acl_reject(1)
```

Claims specific functions. These are the work horses of this document

```
claims#create_source(1)
claims#get_claims(2)
claims#in_group(2)
claims#new_template(1)
claims#resolve_templates(3)
claims#template_substitution(2)
claims#template_substitution(3)
```

You need (in order) **`new_template, create_source, get_claims`**

Java Web Token utilities. These will allow you to create JWTs on the fly.

```
jwt#create_jwt(1)
jwt#create_jwt(2)
jwt#create_keys(0)
jwt#create_keys(1)
jwt#create_keys(2)
jwt#create_skeys(0)
jwt#create_skeys(1)
jwt#create_skeys(2)
jwt#create_uuid(0)
jwt#default_key(0)
jwt#default_key(1)
jwt#get_header(1)
jwt#get_payload(1)
jwt#key_info(0)
jwt#load_keys(1)
jwt#save_keys(1)
jwt#verify_jwt(1)
jwt#verify_jwt(2)
```

Token utilities. These are very low level and essentially allow you to completely replace any built-in machinery in OA4MP. These should have their own tutorial but generally are only for real pros with a burning need.

```
tokens#at_finish(2)
tokens#at_finish(3)
tokens#at_init(2)
tokens#at_refresh(2)
```

```
tokens#id_check_claim(0)
tokens#id_finish(1)
tokens#id_init(1)
tokens#id_refresh(1)
tokens#rt_finish(1)
tokens#rt_init(1)
tokens#rt_refresh(1)
```

Extremely useful tool

Once you have everything debugged and in your script, you should use the **`check_syntax(string)`** function. The argument is a string! This will run it through the parser looking for syntax errors. So if you have a QDL script at **`/home/me/qdl/scripts/my_script.qdl`** then issue

```
check_syntax(file_read('/home/me/qdl/scripts/my_script.qdl'))
```

and either it will return nothing (means everything is ok) or it will return the error string from the parser. Not quite lint but boy is it helpful at times.

Next step: Open up the examples and have this document for cross-reference. There are many, many examples and comments.

Reference for Claim Source attributes

General attributes for all claim source objects

name	type	req?	default	Description
enabled	B	N	T	if this component is enabled. Enabled means it will be processed,
fail_on_error	B	N	F	in the case of some error (such as the underlying service is unavailable), fail. This means that the entire transaction is aborted and the request is rejected. This is a drastic move in most cases. If this is set to false, then the effect is that the claims from this source will not be available
id	S	N	-	a unique identifier for this that may be referenced.
name	S	N	-	A name for this. This is not used by the system

name	type	req?	default	Description
				so it is mostly to help you.
notify_on_fail	B	N	T	If there is an error, notify the system administrators via email.
retry_count	I	N	1	The number of times the claim source should retry to get the claims before failing.
retry_wait	I	N	0	The number of milliseconds in between retries to wait.
type	S	Y	-	This is the type of the claim source. Note that this must be specified or the entire claim source is rejected as invalid. Each of the given sources lists its type below.

key:
B = Boolean
I = integer
N=No
S = String
Y = Yes

For certain claim sources, the retry count and wait will be honored, but that is up to the implementation. It is used, for instance, in the LDAP claim source for particularly pesky servers.

The function **`create_source`** will examine the type of the configuration and assert defaults for all of these as indicated.

Java Code

Type = 'code'

name	type	req?	default	description
java_class	S	Y	--	The full path to the java class

This is the only required attribute.

HTTP Header filter

Type = 'header'

name	type	req?	default	description
prefix	S	?	oidc_claim_	Filter prefix for all claims. Any header that starts with this prefix will be returned without the prefix.
regex	S	?	-	If included, this matches the names of the header as a regular expression.

Note that this can only be called reasonably in the pre or post auth phase, since it reads the headers from the authorization server. This will take the headers and filter them based on the prefix. That means that if a header starts with the prefix, the prefix is removed and the value is asserted.

Claims from the HTTP headers directly.

There is a variable in the QDL runtime workspace named **`auth_headers.`** which in the pre_ or post_auth phases will contain the headers from the authorization service request. Since as per RFC 7230 Section 3.2 HTTP header names are case insensitive, these are normalized to lower case (though their values are not changed, of course).

In other phases, this variable exists, but is empty. Note that if you are using a proxy, the headers from the proxy server are *not* available to you. You need to manage the values on that server with scripting there.

To get the functionality of the previous filter claim source, you might try using the **`pick`** function

```
// filter header w/ regex:
   a. := pick((k,v)-> '^OIDC_CLAIM_.*' =~ k, auth_headers.);
// a. now has only claims that start with OIDC_CLAIM_.
// To rename the claims, one way is to remove the OIDC_CLAIM_ string
   rename_keys(a., keys(a.)- 'OIDC_CLAIM_'); // This alters a.
   claims. := claims. ~ a.; // add them to your claims.
```

Tip on using the header claim source inside a QDL script.

If you want to create a header claim source to be used in QDL, you should load the claims module using **share** mode:

```
          c2 := j_load('oa4mp.util.claims', 'share');
         cfg. := c2#new_template('header')
 cfg.'prefix' := 'x_';
auth_headers. := auth_headers. ~ {'x_foo' :'bar', 'x_arf':'woof', 'y_oink':'oink'};
         cfg. := c2#create_source(cfg.);
     headers. := c2#get_claims(cfg., claims.'sub');
      claims. := claims. ~ headers.;
```

In this (rather artificial) example, values are added (in the client's stored script) to the header which are then processed. In practice, the values in the **`auth_headers.`** would have been sent by the client in the request. So you can either process them directly using QDL (previous example) or if you want to use the claim source for whatever reason, you can load the module in shared mode and use that.

The net result of this example is that the claims **`{'foo':'bar','arf':'woof'}`** are added to the returned claims.

File System

Type = 'file'

name	type	req?	default	description
file_path	S	Y	--	This is the local absolute path on the server to a file containing attributes about a user. The file must be accessible to the system at runtime.
claims	Stem	N	--	A stem of the claims. You may either specify this directly or specify a file_path.
claim_key	S	N	sub	The name of the claim that will be used for fetching information. This defaults to the sub claim – the name the user used when authenticating.
use_default	B	N	--	If a user is not found, return a default set of claims
default_claim	S	N	--	The name for the default set of claims

Claim file format

The file is a simple JSON object of the form

```
{
  “username0":{“key0":"value0", “key1":"value1", … [,"comment":["line0","line1",
…]]},
  “username1":{…}
}
```

Where each username is a login name and the list of key/value pairs is returned as part of the claims. Comments may be embedded in the file (as a JSON array of strings) or for each entry. Comments are not returned as part of the claims. See the sample file at test-claims.json.

Caveat. In QDL. If you get the claims directly each time, you may use virtual paths in your file claim source. If, however, you create the claim source and add it to the claim sources OA4MP has (effectively handing off the claim source to something outside of QDL) then getting claims will fail since OA4MP does not understand virtual file paths. Therefore, if you are going to add a file system source to the claim sources, you must either set the claims property with the claims you want (which can therefore live in the VFS) *or* set the file path to the absolute path on the systehm.

NCSA

Type = 'ncsa'

You really do not need to so more than create one of these from the template.

LDAP

This is probably the most complex and flexible of them. This is also why the **`create_source`** function exists: You can set just a few of the attributes you need, and that will fill in all the other default values for you.

Type = 'ldap'

Name	req?	Description
auth_type	Y	The authorization type. Values are **none**, **simple** or **strong**
address	Y	The address of the server
port	N	The port. Default is 636
claim_name	N	The value of the claim to pass in the search. Default is the *username* claim, *i.e.*, the name the user used to log in.
search_base	Y	The path in LDAP where to start this search.
ldap_name	N	The name of the attribute in LDAP to search on. This defaults to *uid*.
password	?	Only if the authorization type is simple
filter	N	The search filter.
username	?	Only if the authorization type is simple.
search_attributes	N	The names in LDAP of the attributes to return. Omitting returns *all* LDAP attributes!
groups	N	The names of attributes in LDAP that represent groups
rename	N	Rename the LDAP attributes as claims. Form is rename.**old_name** := **new_name** for each claim you want to rename.
lists	N	The names of attributes in LDAP that should be treated as lists (so multi-valued)
context	N	the name of the LDAP context or object to search. Very necessary when you need it, but can be ignored most of the time.

address is a comma separated list of addresses. These will be tried consecutively until one of them works or all of them fail. This allows for fail over servers to be specified and treated as a unit.

Caveat: If you want to rename attributes, you must explicitly list all attributes in **`search_attributes.`** You cannot (at this point) omit the attributes (to just get everything for an entry) and rename some of them.

Filters

There is a complete syntax for filters which use NPN (Normal Polish Notation). There are two main ways to create them.

1. Have a very basic match constructed. If you set **`claim_name`** and **`ldap_name`** then a basic search filter consisting of a single criterion is created and used

   `(ldap_name=claims.claim_name)`

   i.e. the `claim_name` is taken from the `claims.`

   E.g.

   ```
   ldap.'claim_name' := 'eppn'
    ldap.'ldap_name' := 'voPersonExternalID';
   ```

   constructs the query

   `(voPersonExternalID=bob@bsu.edu)`

   Assuming that `claims.'eppn' == 'bob@bsu.edu';`

2. Setting the **`filter`** attribute uses that *exactly* and will ignore creating them as in step 1, even if those attributes are set.

Example 1

This defines a function that makes a set of simple criteria that can be stuck together with logical connectors (& or |) to make more or less any filter you want. Then it is used in an example to construct the specific filter

```
   // concatenate the arguments, which can be stems:
   criteria(key, op, value) -> reduce(@+, '(' + key + op + value + ')');
     op. := {*:'='}; // sets default value
    op.2 := '<='; // element 2 is inequality, not equality
   keys. := ['uid', 'voPersonExternalID', 'storageQuota'];
 values. := ['user_547', 'bob@bgsu.edu', 100000];
       s := criteria(keys., op., values.);
   s; // display the result
(uid=user_547)(voPersonExternalID=bob@bgsu.edu)(storageQuota<=100000)
```

So this would be fine

```
      ldap_cfg. := new_template('ldap');
ldap_cfg.filter :=  '(|' +  s + ')';
```

Example 2

Create the following

```
(|(uid=http://cilogon.org/serverT/users/2604273)(voPersonExternalID=http://
cilogon.org/serverT/users/2604273))
```

Solution

Define a function to slap together (um, concatenate with +) attributes and values then put `(| )` around them:

```
filter_or(attr, value)->'(|' + reduce(@+, '(' + attr + '=' + value + ')') + ')';
```

In context you would use this (assuming that `claim.sub` contained the identifier) as

```
filter_or(attr, value)->'(|' + reduce(@+, '(' + attr + '=' + value + ')') + ')';
        cfg. := new_template('ldap');
    cfg.port := 636;
  cfg.filter := filter_or(['uid','voPersonExternalID'], claims.'sub');
  // etc., etc., etc.
```