Creating Claim Sources

Version 1.3.1

Introduction

This document tells you how to create claim sources for your OIDC client. This uses the QDL scripting language.

Lifecycle of claims

A claim source is defined by its configuration. You create a list of these and they are executed sequentially, the output of one being the input of the next. Furthermore claims may be run either at authorization or right before the access token is issued. You may also execute scripts before any processing is done and before and after both the authorization and access token stages. Note that you probably won't need that much massaging. Frequently the needs are extremely modest for such processing.

Scripts

Scripts are executed in one of 6 execution phases. There are pre- and post- phases for each of authorization, token and refresh. If you specify that the phase is pre_*X* then it is run before any claims are gotten. This is typically where you set a claim source(s). The post_*X* phase allows you to change your claims

- pre_auth – executed before any claims are obtained in the authorization phase.
- post_auth – executed after the initial set of claims is obtained in the authorization phase.
- pre_token – executed before the final set of claims is obtained in the token phase
- post_token – excuted after the final set of claims is obtained in the token phase
- pre_refresh – executed before the claims are refreshed
- post_refresh – executed after the claims are refreshed.
- pre_exchange - executed before the token exchange
- post_exchange - executed after the token exchange

Certain claims can only be gotten at certain times. For instance, claims that rely on the http headers from the identity provider are only available during the authorization phases, so these claims are gotten and stored. These can never be re-gotten until the user logs in again.

In the case of the token exchange (RFC 8693), there are no user claims that are gotten because none are to be returned. Mostly this supplies an entry point for custom code to process the request (for instance, if scopes are requested that require a check in LDAP to verify). Usually the requirements for the excahnge are the same as for the token phases, so the most common use pattern is to just specify exchange, refresh and access phases for a single script. Note that, as always, the current phase is set in the state, so your scripts can check.

An important example

Another very common situation is that the authorization phase is done by some other piece of software (*e.g.* CILogon) in which case, OA4MP is notified that the authorization is in progress. Note especially in that case, that OA4MP gets a request to start the flow (this is where the pre_auth script would be run) and responds, and in the response is the authorization grant. The server then parses the response, takes the grant and makes various calls to populate the current OAuth transaction state with user information for later. All of this is happening before the user gets a response from the service. Only at the very last exchange between the OAuth server and the authorization server is the post_auth script run.

Since it is possible to have each phase of the OAuth flow do out of band transactions, there are pre and post phase calls. (The scripting language is called QDL which comes from aviation, where Q-D-L is a moniker that relates to taking repeated navigational bearing at specific times as vital course corrections, especially if the aircraft is flying blind. In much the same way, whatever happens out of band, the flow is set in the right direction with each call.)

Types

There are 5 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 intial 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. Custom. 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.

General attributes for any claim source

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	n 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 so it is mostly to help you.
notify_on_fail	B	N	T	If there is an error, notify the system administrators.
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
N=Not required
S = String
Y = Required

Code

type = ‘code’

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

This is for a claim source that you write in Java. Note that the basic types of claim sources you are apt to need are already written and unless you have extremely specific requirements, you should not be writing custom extensions. This is included for completeness.

The first step in this is to write a custom extension to BasicClaimSourceImpl. The contract is to have a no-argument constructor. If that is missing, a runtime exception will be raised.

Once you write your extension, you must put it in the classpath for the Java VM, since the lifecycle is to create an instance using reflection, then inject the state. The configuration object will have defaults set (see above) which may be overridden. The configuration object will be turned into a JSON object and handed to your class too.

E.g.

There is a test class for this `edu.uiuc.ncsa.myproxy.oa4mp.oauth2.claims.TestClaimSource` which will simply return a claims object that echos the parameters. Here's how to invoke it. Note that all you need to do is change the java class to your class (make sure its in the classpath!) and pass in your arguments. See the documentation for BasicClaimSourceImpl. In this example, we invoke this test claim source with two custom parameters, foo and bar:

```
  cfg. := new_template('code')
  cfg.java_class := 'edu.uiuc.ncsa.myproxy.oa4mp.oauth2.claims.TestClaimSource'
  cfg.foo := 'foo-test'
  cfg.baz := 'baz-test'
  get_claims(create_source(cfg.), 'jeff')
{
 config param "type"=code,
 config param "foo"=foo-test,
 config param "baz"=baz-test,
 config param "enabled"=true,
 config param "fail_on_error"=false,
 config param
"java_class"=edu.uiuc.ncsa.myproxy.oa4mp.oauth2.claims.TestClaimSource,
 info=Echoing configuration parameters.,
 config param "id"=qdl_claim_source,
 config param "notify_on_fail"=true
}
```

The create_source function adds some properties, such as notify_on_fail and sets a dummy id. You can ignore all of that (unless for some reason you don't want to).

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.
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.

name	type	req?	default	description
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

The file is a simple JSON object of the form

```
{
  “username0”:{“key0”:”value0”, “key1”:”value1”,…},
  “username1”:{…}
}
```

Where each username is a login name and the list of key/value pairs is returned as part of the claims. For instance If the file contains

```
{“bob”:{
   “eppn”:”bob@bigstate.edu”,
   ”affiliation”:”staff”,
   "isMemberOf":  [
        {"name": "all_employees","id": 1097},
        {"name": "jira-users"},
        {"name": "lsst_users","id": 1363},
      ],
 }
}
```

then the resulting claims object when bob logs in would contain the given eppn and affiliation as well as the given groups.

Example.

```
f. := new_template('file');
f.file_path := '/home/ncsa/dev/ncsa-git/oa4mp/oa4mp-server-test-oauth2/src/main/resources/
test-claims.json';
file_claims. := get_claims(f. , 'jeff');
say('***** File Test');
say(file_claims., true);
```

NOTE. A default claim can be returned if the user is not found. This gives the same set of claims for all users who do not have an explicit entry. This is useful if, e.g., all users in an organization are to be some member of a default group and have the same affiliation.

NCSA

type = ‘ncsa’

Mostly this is created for you.

```
cfg. := new_template(‘ncsa’);
// any overrides you may need.
cfg. := create_source(cfg.);
```

If you look at the created claim source, you will a ton of stuff there. This is all fixed to talk to the NCSA LDAP server. Note especially, that this only works through the NCSA internal network, so this is not a generally useful claim source. This is just a customization of the LDAP claim source.

LDAP

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
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!
group_names.	N	The names of attributes in LDAP that represent groups
rename.	N	Rename the LDAP attributes as claims
list.	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.

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

```
cfg2. := new_template('ldap');
cfg2.address := 'ldap1.ncsa.illinois.edu,ldap2.ncsa.illinois.edu';
cfg2.auth_type := 'none';
cfg2.search_base := 'ou=People,dc=ncsa,dc=illinois,dc=edu';

// So the next two lines are key:
```

```
// This is the name of the attribute in LDAP to search on
cfg2.ldap_name := 'uid';

// There is a claim with this name, pass the value to the search engine
cfg2.claim_name := 'uid';

// The search in LDAP then is ldap_name == claim_name
// The next commands specify what attributes to get back from
// the server (rather than all of them)
cfg2.search_attributes.0 := 'email';
cfg2.search_attributes.1 := 'uid';
cfg2.search_attributes.2 := 'uin';
cfg2.search_attributes.3 := 'memberOf';
// Since memberOf is a group, we want it parsed rather than being a long messy string
cfg2.groups.0 := 'memberOf';

// Finally, the name in LDAP is 'memberOf' and the expected claim name is 'isMemberOf'
// So we set the rename of
cfg2.rename.memberOf := 'memberOf';

claims. := get_claims(create_source(cfg2.), 'jgaynor');
say('***** LDAP Test');
say(claims., true);
```

This will return *all* the claims in the server. This is usually far too much and what's more, there is no canonical way the LDAP should organize the attributes for a person. As such the result, while certainly complete, is apt to be a mess to deal with. Fortunately, you can restrict what attributes you can and can do things like flag them as groups (so as to turn them in to more digestible objects) or rename them all in one step so you don't have to do so later.

Next is another example.

```
{
 auth_type=none,
 address=ldap.ligo.org,
 port=636,
 groups.= {
  0=isMemberOf
 },
 claim_name=uid,
 search_base=ou=people,dc=ligo,dc=org,
 search_attributes.= {
  0=email,
  1=uid,
  2=uin,
  3=isMemberOf
 },
 type=ldap,
 ldap_name=ligo
}
```

HTTP headers

You may also harvest claims from the http headers. The usual reason for doing this is because some IDPs prefer doing this (Satosa is a framework that prefers it, for instance). Another reason is if you are chaining together delegation through a sequence of OIDC servers.

```
h. := new_template('http');
h.prefix := 'OIDC__';
h2. := create_source(h.);
h2.headers. := headers.;
h_claims. := get_claims(h2., 'jgaynor');
say('***** HTTP Headers Test');


headers.OIDC__sub := 'http://cilogon.org/serverT/users/173048';
headers.OIDC__idp_name := 'National Center for Supercomputing Applications';
headers.OIDC__eppn := 'jgaynor@ncsa.illinois.edu';
headers.OIDC__cert_subject_dn := '/DC=org/DC=cilogon/C=US/O=National Center for
Supercomputing Applications/CN=Jeffrey Gaynor T173053';
headers.OIDC__eptid := 'https://idp.ncsa.illinois.edu/idp/shibboleth!https://cilogon.org/
shibboleth!i65P3o9qFNjrpS4z6+WI7Dir/4I=';
headers.OIDC__iss := 'https://cilogon.org';
headers.OIDC__given_name := 'Jeffrey';
headers.OIDC__aud := 'myproxy:oa4mp,2012:/client_id/7e864a3c5200ca701df4900d490257f3';
headers.OIDC__acr := 'https://refeds.org/profile/mfa';
headers.OIDC__idp := 'https://idp.ncsa.illinois.edu/idp/shibboleth';
headers.OIDC__affiliation :=
'staff@ncsa.illinois.edu;employee@ncsa.illinois.edu;member@ncsa.illinois.edu';
headers.OIDC__name := 'Jeffrey Gaynor';
headers.OIDC__family_name := 'Gaynor';
headers.OIDC__email := 'gaynor@illinois.edu';
headers.OIDC__auth_time := '1581461958';
headers.OIDC__exp := '1581462859';
headers.OIDC__iat := '1581461959';
headers.OIDC__nonce := 'stW3-mkHk70ERlIx2vwasSWXkjXjLEsTUqTrouxcJZY';
```

Alternate

Send up a JSON object as the configuration. For scripts, this is a JSON serialization of a stem.

E.g.

```
{“qdl”: {“script”:”comanage/ldap.qdl”,
    “args”:{
   “password”:”changeme!”,
   “principal”:”uid=oa4mp_user,ou=system,o=MESS,o=CO,dc=cilogon,dc=org”
   "auth_type":"simple",
   "address":"ldap.cilogon.org,ldap2.cilogon.org",
   "claim_name":"username",
   "search_base":"ou=People,dc=ncsa,o=MESS, dc=illinois,dc=edu",
   "search_attributes":["email","uid","uin","memberOf"],
   "groups":["memberOf"],
   "rename":{"memberOf":"isMemberOf"},
   "type":"ldap",
   "ldap_name":"username"
   }
}
```

This is a pretty complete example. Generally the user should send up only what is necessary.

```
{“qdl”: {“script”:”comanage/ldap.qdl”,
    “args”:{
```

```
   “password”:”changeme!”,
   “principal”:”uid=oa4mp_user,ou=system,o=MESS,o=CO,dc=cilogon,dc=org”
   "search_base":"ou=People,dc=ncsa,o=MESS, dc=illinois,dc=edu",
   }
}
```

So the default configuration for this script is created, the json object named “args” is turned in to a stem and the values there are set in the configuration that is then sent for processing.

This should be contrasted with uploading actual QDL code.

```
{“qdl”:{“code”:[lines]}}
```

or perhaps if there are several things to upload

```
{“qdl”:[
   {“code”:[lines0],”phase”:”...”},
   {“code”:[lines1], “phase”:”...”},
   …
  ]
}
```

Handlers

It is possible that you want to complete custom create your own token. You may do this if the handler module is loaded.