Attributes New Generation

This page describe a new attribute engine for authentic having simpler articutaled concepts and being more powerful.

It's largely based on a reading of Shibboleth IdPAddAttribute wiki page but with a larger setting than an only SAML 1.0 / SAML 2.0 compliant identity provider.

There are 3 functional concepts: the attribute sources, the attribute engine and the attribute encoders.

Attribute sources:

They contain generic informations common to all kind of attribute sources (hasA relationship in a class diagram):

- a description name,
- a slug (an identifier without spaces),
- dependencies upon other sources, i.e. this source can only be called after another source has been called, for example a script source working upong LDAP attributes must be executed after the LDAP attributes have been retrieved,
- a boolean parameter if the source should always produce its result or only if one of its attributes is requested (present in wanted_attributes attribute describe later in get_attributes() method of the attribute engine)
- source type related informations:
  - location of the source (LDAP url, SCIM REST endpoint, SQL alchemy URL),
  - how to map existing attributes to a search request (LDAP search or SQL select statements, with variables),
  - the list of attributes to extracts. They can be defined through settings ou models, or any other methods.

List of type of sources to have

- an LDAP source using python-ldap
- an SQL source using python-sqlalchemy
- a SCIM source using python-scim
- implicit sources based on authentication backends:
  - LDAP authentication provides an LDAP source
  - OAuth 2.0 authentication provides an user-info web-service source
  - SAML 2.0 authentication provides an assertion attributes source
- a python expression source: it contains two fields:
  - a name for the attribute
  - a python expression, a variable named attributes is available as a local variable

Configuration

Each source is responsible to finding its configuration, there is no global definition of how the configuration should look like, where it is stored, etc... Available attribute source types are found using the list of installed applications by looking for a contained module named attribute_backend containing a class named AttributeBackend, i.e. if the saml application want to define an attribute source there should be file named saml/attribute_backend.py which contains a classe named AttributeBackend.

Interface

- get_instances(ctx)

  Return a list of (source, instance) for this source. It's used by the attribute engine to list all attribute source instances.

- get_dependencies(instance, ctx)

  Return the list of attributes this instance requires. It's used by the attribute engine to order the attribute source instances before running them.

- get_attribute_names(instance, ctx)

  Return the list of attribute names produced by this instance.

- get_attributes(instance, ctx)

  Return a dictionnary of the attributes produced for this instance given the context ctx.

Attribute engine

A singleton object, there is one instance for the global application. The default implementation is found using the A2_ATTRIBUTE_ENGINE which should be a Python class path. A default one is provided.

Interface

- get_attribute_names(ctx)

  It calls every sources and returns the aggregated list of defined attributes, defined does not mean that the attributes are available. For example if you have two ldap sources ldap1 and ldap2 against which people can be authenticated. If an user is authenticated against the ldap1 sources, the attributes from ldap1 will be defined and available (not all of them as an LDAP search can return a partial response), but the attributes from ldap2 will be defined but not available at all.

- get_attributes(ctx):

  It is responsible for calling every sources in order to aggregate attributes. The ctx argument is used to initialize the ctx argument passed to the first source called. Each source instance decides each time to produce or not some attributes, i.e if its dependencies have not produced the needed attributes, it can decide to do nothing. The engine is authorized to cache the list of instances between calls and the topological sort done based on the dependencies attributes. If a cyclical dependency is found an exception is raised. Each source is called only one time. If you want a source to be run again (if there is an implicit cyclical dependency), then call get_attributes() again.

  Some attribute names have special meaning:

  - wanted_attributes:
    It must be a list of attribute names, it list the attributes expected by the requestor. For example a service provider could list the attributes referenced by some of its attribute encoders.
  - session:
    The request.session object, if available.
  - request:
    The Django request object, if available.
  - user:
    The Django request.user object, if available.

TBD

- How to remember that a source has already been called ? Maybe it's the job of each to store an attributes identifer.last_call with a timestamp, to determine previous call and freshness.
- Maybe linking get_attributes to the request object is too limiting. Maybe we should just pass request.session.pop('attributes', {}) then overwrite it with the return value, which would permit to use the engine in other contexts than a Django HTTP request.
- What to do on source errors ? i.e the LDAP raise an error or the script source fails on an undefined variable.

Attribute encoders

They are objects linked to attribute recipients, usually service providers in the Authentic2 setting. They define which and how attributes should be transmitted. They can be defined through the setting file or ORM models, this is undefined. Their interface is also undefined as it depends upon the identity provider calling them. They are not really part of the attribute engine but clients of it. Each identity provider backend is responsible for calling them. They are the clients of the attribute engine interfaces, and are here only to see how the attribute engine interfaces will be used.

Examples

- The SAML 2.0 identity provider could define the following encoders:
  - SAML2SimpleAttributeEncoder:
    - Its definition contains:
      - an attribute name from the list given by @Engine.get_attribute_names(),
      - parameters to build a saml:Attribute element, i.e a Name, NameFormat and FriendlyName
    - For ease of use the IHM interface could provide templates, using the current mapping.py file.

    - SAML2PersistentNameID: generate the NameID from a persistent storage,
    - SAML2TargetedIdEncoder: takes the NameID and create an attribute from it.
    - SAML2XMLAttributeEncoder: take the value of the attribute, try to cast it as an XML fragment and use it as the content of the attribute.

- The OAuth 2.0 identity provider could define the following encoders:
    - UserInfoWebServiceEncoder: it contains
        - the URL name of the user_info rest endpoint
        - a Django template which should produce a Django document. It can refers to attributes from the attribute engine.