

Generalized Policy Management

Introduction

The past two decades, we have witnessed the inception and maturation of open authentication and authorization frameworks for the Web, going beyond HTTP's `Authorization` header and its initial `WWW-Authenticate` schemes (i.e., `Basic`, `Digest`) [`@rfc1945`; `@rfc2069`; `@rfc2616`; `@rfc2617`]. If the infancy of these frameworks ran from the baptism of OASIS's [^Organization for the Advancement of Structured Information.] SAML 1.0 and XACML 1.0, and Internet2's Shibboleth 1.0 (2002–2003), to their confirmation in SAML 2.0, XACML 2.0, and Shibboleth 2.0 (2005–2008), then adolescence led them through the troublesome years of OpenID 2.0 and OAuth 1.0 (2007–2010), towards the confidence of early adulthood radiated by XACML 3.0, OAuth 2.0 and OpenID Connect (2012–2014). Since then, however, opposing academic, economic, and political incentives seem to have resulted in a long-stretched mid-life crisis: the misappropriation of OAuth authorization primitives in OpenID's authentication framework; the explosion of unofficial and inactive but de facto standard OAuth extensions; the simultaneous development of almost identical yet incompatible evolution paths (OAuth 2.1 versus GNAP; OpenID's Digital Credential Protocols versus OpenID Connect with Identity Assurance, Claims Aggregation, and Self-Issued OpenID Provider; multiple Data Spaces authorization protocols) ... It is no wonder that interoperability remains limited to ...

...

In this paper ...

1 Information in a request

Starting from the OAuth 2.x token request – the archetypal yet somewhat ill-named authorization request – we discern a number of functions the contents of the message serve.

1.1 Targets and actions

1.1.1 OAuth & G NAP

Central to the purpose of an authorization request is its **scope**: the extent to which an eventual grant applies, i.e., *what* one requests – and is subsequently allowed or denied – to do. Initially crammed in the eponymous **scope** parameter, a space-delimited list of opaque strings, the heterogeneity of this aspect requires a much more flexible model. While “scope is typically about *what [type of] access* is being requested rather than where that access will be redeemed” [rfc8707], the parameter is often overloaded to convey the *identity*, *location*, or *type* of the protected resource, as well as the *extent* of authorized *actions*. A good example is the custom scope format of the OAuth 2.0 profile for Fast Healthcare Interoperability Resources (FHIR), which is of the form `[type]/[identity].[action]`.¹

One underappreciated upgrade was therefore the addition of *Resource Indicators* under the **resource** parameter: one or more (absolute) URIs representing (the identity of) “the **target** service or resource to which access is being requested” [rfc8707]. In UMA, we see this division reflected in the *ticket request*: the **resource_id** parameter identifies “a resource to which the resource server is requesting a permission on behalf of the client,” while the **resource_scopes** parameter references “[the] scopes to which the resource server is requesting access for this resource” [uma:fed].

Another important progression came in the form of *Rich Authorization Requests*: an array of freely typed JSON structures to specify fine-grained authorization requirements under the **authorization_details** parameter [rfc9396]. The G NAP specification also cements this flexible pattern firmly in its access request model, but places the array under the JSON path `access_token.access`, thereby losing compatibility with the original RAR messages [gnap].

The details that are allowed in each RAR structure are indicated by a **type** parameter. The specification also proposes a number of common details. On the one hand, the target resource can be indicated by its **identifier** and/or its (server) **location**. On the other hand, the scope can be expressed as **privileges** with respect to the resource, **actions** to be taken at the resource, and/or **datatypes** in which the resource should be represented [rfc9396].

1.1.2 ODRL & Data Spaces

A similar distinction between resource and scope has been made from the start in the ODRL Information Model (2016), which also provided the foundation for the message format of the Data Spaces Contract Negotiation Protocol. Resources – called ‘assets’ – can be referenced with any kind of relation, the most prominent of which is `odrl:target`. The scope with respect to an asset is then indicated with `odrl:action`. Both of these aspects can be specified further through

¹https://openid.net/specs/openid-heart-fhir-oauth2-1_0.html

an elaborate system of *constraints*. Taken together, this allows for a far-reaching flexibility in expressing requested and granted permissions – called ‘offers’ and ‘policies’. Furthermore, ODRL extends the same model to other types of rules as well, e.g., prohibitions, or obligations.

1.1.3 OpenID

1.1.3.1 OpenID Connect

While OpenID Connect extends the OAuth architecture, and allows identity claims to be requested via the OAuth **scope** parameter, semantically this information is the target of the request, rather than the scope. After all, the action to be performed on this information is always “read this info about the user who is logged in at the identity provider” – a scope which is already indicated by mandatory value **openid**.

It is therefore more semantically correct to use OpenID Connect’s alternative and more elaborate syntax for expressing the targeted information, using the **claims** parameter. Each desired claim is then specified as a key in one (or both) of the top-level JSON objects **claims.id_token** – to receive them in the identity token – and **claims.userinfo** – to retrieve them from the UserInfo endpoint. The corresponding values can be **null** – the equivalent of requesting the claim via the **scope** parameter (‘Voluntary Claims’) – or a JSON object detailing the request. Within such objects, the boolean parameter **essential** (default **false**) indicates that the claim will ensure a smooth authorization for the specific task requested by the End-User (‘Essential Claims’). The array **values** – or shorthand **value** for a single one – lists the set of acceptable values of the claim (in order of preference); if the actual identify information has another value, it will not be returned in the response.

1.1.3.2 OpenID for Verifiable Credential Issuance

Indicating each target claim with a key name, however, does not suffice to query Verifiable Credentials, which contain a deeper structure. OpenID4VCI therefore introduces another mechanism to request specific claims within a credential. In OAuth RAR objects of the type **openid_credential** – within the **authorization_details** array – the targeted credential is referenced using the **credential_configuration_id** parameter, as it occurs in the Credential Issuer Metadata. Each desired claim is then listed as a Claim Description in the **claims** array, specifying its **path** within the credential – as an array of segments (called a Claims Path Pointer) – and whether this claim is **mandatory** for the Wallet to accept the credential (default **false**).

1.1.3.3 OpenID for Verifiable Presentations

Where the OpenID4VCI flow defines how Wallets request Credentials from issuers, OpenID4VP specifies how Verifiers in turn request Presentations of these Credentials from the Wallet. While semantically almost identical, OpenID nevertheless introduces yet another syntax for

this. Instead of in OAuth RAR objects, the details of a request are now found under the `dcql_query` parameter, an object that contains an array of requested `credentials`. Each of these credential descriptions in turn contains an array of `claims`, specified by a `path` array – like Claim Descriptions – and optionally constrained to an array of expected `values`.

OpenID4VP also adds the ability to suggest several `claim_sets` and `credential_sets`, listing the `id` of claims or credentials that together would suffice for the requester’s use case. Meta-requirements about the credential exchange can also be expressed, such as the desired `format` – and format-specific `meta`-properties – and a list of trust framework conditions constraining the `trusted_authorities`.

Abstract syntax

The syntax for specifying the target and scope of an access request, as discussed above, is summarized in Table 1. Based on this information, we distinguish the following semantic aspects that can be communicated by (the union of) these formats.

- The **target** of the request:
 - as a resource indicator (RI);
 - as a resource identifier (UMA, RAR, GNAP);
 - as a resource server location (RAR, GNAP);
 - as other type of resource description (RAR, GNAP);
 - as (sets of) claim descriptions:
 - * by key (OIDC);
 - * by path (OpenID4VCI, OpenID4VP);
 - * constrained by existence (OIDC, OpenID4VCI, OpenID4VP);
 - * constrained by value (OIDC, OpenID4VP);
 - constrained by boolean expressions (ODRL).
- The **scope** of the request:
 - as a scope indicator (OAuth, UMA);
 - as required privileges on the target (RAR, GNAP);
 - as required data types of the target (RAR, GNAP);
 - as action(s) to be done with the target (RAR, GNAP, ODRL, Data Spaces);
 - as other type of scope description (RAR, GNAP);
 - constrained by boolean expressions (ODRL).
- Other relations to resources (ODRL).

We now propose an abstract syntax that captures these aspects in a uniform way, mapping each element of the concrete formats onto the abstract model.

1.1.3.1 Operations

Operation That to which a rule applies.

An operation has the following information.

@id RECOMMENDED IRI[]. An identifier of the operation.

Mappings:

- Data Spaces: `iri("offer.@id")` (absent if derived from dataset; opt. unique)
- ODRL (Policy): `iri("uid")` (req. unique)

rdf:type RECOMMENDED IRI[]. The type of the operation.

Mappings:

- G NAP: `iri("access_token.access[].type")` (req.)
- OAuth, RAR: `iri("authorization_details[*].type")` (req.)

target RECOMMENDED Asset[] . The asset(s) targeted by the operation.

Mappings:

- RI: `Assets("resource")` (req. unique)
- UMA: `Assets("resource_id")` (req. unique; in ticket)
- RAR: `Assets("authorization_details[*].identifier")` (opt. unique)
- G NAP: `Assets("access_token.access[*].identifier")` (opt. unique)
- Data Spaces (absent if derived from dataset):
 - `Assets("offer.target")` (shared; opt. unique)
- ODRL (Policy):
 - `Assets("target")` (shared short; opt. unique)
 - `Assets("target.uid")` (shared; opt. unique)
 - `Assets("<rule>[*].target")` (short; opt. unique)
 - `Assets("<rule>[*].target.uid")` (req. unique)
- OIDC:
 - `Assets("claims.id_token")` (opt.)
 - `Assets("claims.userinfo")` (opt.)
- OpenID4VCI:
 - `Assets("credential_configuration_id")` (req. unique)
 - `Assets("authorization_details[*].claims[*]")` (opt.)
- OpenID4VP:
 - `Assets("dcql_query.credentials[*]")` (opt.)
 - `Assets("dcql_query.credentials[*].claims[*]")` (opt.)

action RECOMMENDED Action[]. The action(s) that the operation would perform.

Mappings:

- OAuth: Actions("scope") (opt. unique)
- UMA: Actions("resource_scopes") (req. unique; in ticket)
- RAR:
 - Actions("authorization_details[*].actions") (opt. unique)
 - Actions("authorization_details[*].privileges") (opt. unique)
 - Actions("authorization_details[*].datatypes") (opt. unique)
- GNAP:
 - Actions("access_token.access[*].actions") (opt. unique)
 - Actions("access_token.access[*].privileges") (opt. unique)
 - Actions("access_token.access[*].datatypes") (opt. unique)
- Data Spaces (absent if derived from dataset):
 - Actions("offer.<rule>[*].action") (opt. unique)
- ODRL (Policy):
 - Actions("action") (shared short; opt. unique)
 - Actions("action.uid") (shared; opt. unique)
 - Actions("<rule>.action") (short; opt. unique)
 - Actions("<rule>.action.uid") (req. unique)
- OIDC, OpenID4VCI, OpenID4VP: static({ @id: "openid:read" })

Table 1: Syntax for specifying the target and scope of an authorization request.

Spec.	Target	Scope
OAuth 2.x	/	scope
~ RI	resource	/
~ UMA	resource_id (ticket)	resource_scopes (ticket)
~ RAR	authorization_details[] .identifier .location	authorization_details[] .actions .privileges .datatypes
GNAP	access_token.access[] .identifier .location	access_token.access[] .actions .privileges .datatypes
ODRL	permission[].target	permission[].action
Data Spaces	offer.permission[].target	offer.permission[].action
OIDC	claims.id_token.[key] claims.userinfo.[key] .essential .values[] .value	/
OpenID4VCI	authorization_details[] .cred..._conf..._id .claims[] .path[] .mandatory	/

Spec.	Target	Scope
OpenID4VP	dcql_query .credentials[] .claims[] .path[] .values[]	/

A Mappings

str(path) Reads JSON value at **path**; if the value is not a string, the mapping fails. Otherwise, returns the string.

str'(path)(term) Writes the term **term** to **path** as a JSON string.

iri(path) Reads JSON value at **path**; if the value is not a string, the mapping fails. If the value represents an IRI, returns the IRI. Otherwise, prepends the *generic domain* to the string, then returns it as an IRI.

iri'(path)(term) Writes the IRI **term** to **path** as a JSON string. If the IRI begins with the *generic domain*, the prefix is stripped.

node(class)(path) Reads JSON value at **path**. Returns a node of type **class**. If the value is a JSON string, sets the node's identifier to **iri(path)**.

node'(class)(path)(term) If the node **term** has a single identifier as its only attribute apart from **@type**, writes the identifier to **path** as a JSON string. Otherwise calls **node''(class)(path)(term)**.

node''(class)(path)(term) Writes the node **term** to **path** as a JSON object, with the field **@type** set to **class**.

array(map)(path) Reads JSON value at **path**. If the value is not a JSON array, interprets it as a singleton array. Maps each element of the resulting array with **map(path)**. Returns the mapped array.

array'(map')(path)(terms) If **terms** is a singleton, writes the only element **term** to **path** as a JSON value, with **map'(path)(term)**. Otherwise calls **array''(map')(path)(terms)**.

array''(map')(path)(terms) Writes the terms **terms** to **path** as a JSON array, mapping each element **term** to a value with **map'(path)(term)**.

We further define the following aliases:

- **Party** = **node("Party")**
- **Parties** = **array(node("Party"))**
- **Agents** = **array(node("Agent"))**
- **Actions** = **array(node("Action"))**
- **Assets** = **array(node("Asset"))**