UMA Implementer's Guide

Abstract

This document is a non-normative set of auxiliary material produced by the User-Managed Access Work Group. It provides advice to, and discussions relevant to, developers and deployers of UMA-enabled software systems, services, and applications.

Status

This document will have material added to it as warranted. It focuses primarily on UMA V2.0; every attempt will be made to mark content that applies to specific versions.

Editors

Eve Maler, Maciej Machulak

Intellectual Property Notice

The User-Managed Access Work Group operates under Kantara IPR Policy - Option Patent & Copyright: Reciprocal Royalty Free with Opt-Out to Reasonable And Non discriminatory (RAND) (HTML version) and the publication of this document is governed by the policies outlined in this option.

The content of this document is copyright of Kantara Initiative. © 2018 Kantara Initiative

---

Table of Contents

- 1 Introduction
- 2 Security Considerations Regarding Interactive Claims Gathering Flows
- 3 Motivations for the Design of the PAT
- 4 Considerations Regarding Resource Server Permission Requests
  - 4.1 REST-Style API Controlled by Scopes
  - 4.2 Ambiguous Role-Based Scopes
  - 4.3 Scopes for Ambiguous API Calls
  - 4.4 Resource Owner-Influenced Scopes
- 5 Interpreting Authorization Assessment Set Math
- 6 Resource Server Error Handling When the PAT Is Invalid
- 7 Considerations Regarding Resource Owners and Requesting Parties
- 8 Considerations Regarding Resource Server API Constraints
- 9 Considerations Regarding Resource Registration Timing and Mechanism
- 10 Considerations Regarding Scope Discovery
- 11 Extension Opportunities
  - 11.1 Using Alternate Communications Protocols
  - 11.2 Resource Registration for OAuth and OpenID Connect
  - 11.3 OpenAPI Format for Resource Registration
  - 11.4 Facilitating Chatter Reduction at the Resource Server
  - 11.5 Informing the Authorization Server About Protected Resource Locations
  - 11.6 Cascading Authorization Servers
  - 11.7 Hashed Claims Discovery
  - 11.8 Resource Baskets
  - 11.9 Notification Endpoint
- 12 Managing Resource Registration Revisions
- 13 Understanding Authorization Server Response Options From the Token Endpoint
- 14 Handling Optional and Extension Properties (V1.0.1)
- 15 Change History

Introduction

This document is a non-normative set of auxiliary material produced by the User-Managed Access Work Group. It provides advice to, and discussions relevant to, developers and deployers of UMA-enabled software systems, services, and applications.

This document uses terms and abbreviations defined in the UMA V2.0 specifications, and presumes understanding of UMA concepts.

Security Considerations Regarding Interactive Claims Gathering Flows

When the requesting party is redirected to the authorization server for interactive claims gathering, there are several possible attacks identified by the OAuth 2.0 Security Best Current Practice. The OAuth 2.0 authorization code flow is substantially similar to UMA interactive claims gathering: the `claims_redirect_uri` parameter is similar to the OAuth `redirect_uri` parameter, the incoming ticket is similar to OAuth scopes, and the returned ticket is similar to the OAuth authorization code; both flows require a client_id and recommend a state parameter. Therefore, these attacks can be mitigated through the countermeasures described in the OAuth 2.0 Security Best Current Practice. Two such attacks are Cross Site Request Forgery (Section 4.1), recommending application of PKCE, and the Mix-Up attack (Section 4.4), which has several possible mitigations.

One consideration specific to UMA is the possibility for the client to repeatedly invoke interactive claims gathering redirection before use of the token endpoint. This is possible since each ICG cycle results in a new UMA ticket to be issued to the client. This possibility requires additional security analysis and profiling of PKCE to ensure it still effectively provides the desired client authentication outcome. Another potential mechanism to mitigate these attacks is the use of client DPOP. Instead of a PKCE code challenge/verifier, the client is registered with a public key, possibly through DCR, and dynamic client authentication is the required at the token endpoint. Another direct resolution is for an AS profile to explicitly require a call to the token endpoint with the UMA ticket received through ICG redirection. This effectively ensures the PKCE flow is performed as designed.

Motivations for the Design of the PAT

Some developers and deployers studying UMA 2.0 have asked questions about the protection API access token, known as the PAT. What is it? Why does it exist? When does it need to be used?

The PAT is simply an OAuth access token with a particular scope, standardized by UMA with the name `uma_protection`. It represents the resource owner's (Alice's) authorization for the resource server, acting as an OAuth client, to use the protection API presented by the authorization server.

This API is defined by the optional UMA Federated Authorization specification that is referenced by the UMA Grant specification; these sequence diagrams may be of interest. This API is what allows the resource server, on resource owner Alice's behalf, to outsource resource protection to the authorization server in a formally defined fashion. In the FedAuthz specification, OAuth and the PAT are being used just as they would be for any other OAuth-protected API -- Facebook, Feedly, whatever. (This writeup assumes FedAuthz is in use. Otherwise there is no point talking about the PAT.)

Keep in mind that Alice could need resource protection from more than one resource server at a single authorization server, or could be using more than one authorization server -- or both. OAuth is a classic choice for ensuring both security and Alice's authorization for these trusted app-to-service connections. It's just that, in this case, the "app" is a resource server and the "service" is an authorization server.

To achieve resource protection, the protection API offers three endpoints. Alice's permission is required for the resource server to make calls to any of these; hence the resource server must present a valid PAT for any of these calls to be allowed:

- **Resource registration**, so that the authorization server knows what resources to protect and how to let Alice give it the policy conditions under which to grant access (various CRUD operations for managing registration and deregistration).
- **Permission requests** on behalf of the client app used by a requesting party (such as Bob), so that the authorization server can correlate the originally attempted resource request with the client's later request for an access token (the resource server gets back a permission ticket that it hands right over to the client).
- **Access token introspection**, so when Bob's client finally makes a resource request toting an RPT (requesting party token), the resource server can look up at the authorization server what resource and scope access was actually granted and enforce that grant.

The protection API is an "offline" type of API, meaning the resource server generally needs to make API calls to the authorization server when Alice does not currently have a session. In practice, this typically requires a way for the resource server to store a longer-lived refresh token persistently so that it can refresh a shorter-lived PAT on an ongoing basis. (For some thoughts on resource server error handling when the PAT is invalid, see this section.)

Here are "offline" use cases for each of the endpoints. None of these situations require Alice's current availability, vs. some other condition for the resource server to make the API call to the protection API.

- **Resource registration endpoint:** The resource server needs to update all of Alice's resources *when its API is versioned*. For example, the API couldn't tag photos before, but now it's able to, and this corresponds to a new available scope.
- **Permission endpoint:** The resource server needs to request one or more permissions on Bob's client's behalf *at (tokenless) resource request time*.
- Token introspection: The resource server needs to introspect the RPT that Bob's client brings it *at (token-carrying) resource request time.*

The question has been posed: Why doesn't the design of UMA2 call for the resource server (as an OAuth client) to switch to using its own client credentials, rather than the PAT, for the permission and token introspection endpoints of the protection API? It's possible for the authorization server to derive the resource server's intended resource owner from a resource ID passed in the request if client credentials were used. (see UMA GitHub issue #352)

The Work Group chose not to switch to a different construct for the following reasons:

- **It would be inefficient.** What if Alice the resource owner had revoked her PAT before Bob's client made a tokenless resource request (requiring the resource server to use the permission endpoint) or an RPT-bearing resource request (requiring the resource server to use the token introspection endpoint), say, because Alice doesn't "like" the resource server anymore? Now the authorization server would have to keep track of whenever she did this and then prevent any client credentials flows from happening. It wouldn't get this tracking "for free" from an invalid PAT.
- **It would disrupt the higher-order trust relationship.** Along with this loss of tracking comes the loss of the resource owner's trust relationship with and delegation to the authorization server (properly stated, to the Authorization Server Operator) of its authorization function. The work by the UMA Legal subgroup to develop an UMA legal framework/business model includes the mapping of legal devices, such as contracts (including trust frameworks) and licenses, to the various UMA technical artifacts and messages, including the PAT. So this mapping capability might be lost if the PAT gets overtaken by a different mechanism.
- **It would make the design inelegant.** It's weird for the authorization server to have to derive resource owner context this way. Both the permission request message and the token introspection response can contain multiple resource IDs. The spec then might need a new error condition to check whether they all match the same resource owner.

Thanks to UMAnitarians James Philpotts, Domenico Catalano, and Andi Hindle for contributing to this writeup.

Considerations Regarding Resource Server Permission Requests

Because access attempts on resources by clients are resource identifier-unaware, the process of making a permission request also requires interpretation by the resource server in order to establish a suitable resource identifier, resource owner, and authorization server. It is recommended for the resource server to document its intended pattern of permission requests in order to assist the client in pre-registering for and requesting appropriate scopes at the authorization server. Following are some scenarios.

REST-Style API Controlled by Scopes

For example, the FHIR API has a sophisticated set of resource types, with each resource (say, of `Condition`, `Medication`, and `Observation` types) having these operational options:

- Create = POST /*path*/*resourceType*
- Read = GET /*path*/*resourceType*/*id*
- Update = PUT /*path*/*resourceType*/*id*
- Delete = DELETE /*path*/*resourceType*/*id*
- Search = GET /*path*/*resourceType*?*parameters*...
- History = GET /*path*/*resourceType*/*id*/_history
- Transaction = POST/*path*/ (POST a transaction bundle to the system)
- Operation = GET /*path*/*resourceType*/*id*/$*opname*

As of this writing, there are two scopes, mapping to a subset of the options:

- `read` scope (for Read and Search)
- `write` scope (for Delete, Create, and Update)

Since there is a many-to-one relationship between API calls and the scopes they map to, the resource server can distinguish the desired scope from the client's access attempt and request it that scope if it sees fit.

Ambiguous Role-Based Scopes

An API uses scopes to manage access per role as follows (assume the API calls themselves are unambiguous for this example):

- Read and write access: `user` scope
- Read, write, execute, superuser access: `admin` scope

If a client attempts read or write access, it is ambiguous whether `user` scope or `admin` scope is sought.

This is a good example of where the client would want to exercise its option to pre-register for and then dynamically request `admin` scope if the requesting party's immediate need were only for the two actions also available within `user` scope. It's also a good example of where the resource server would potentially want to have a strategy of "parsimonious" rather than "generous" permission requests (either requesting `user` scope rather than `admin` scope and not requesting additional resource identifiers, or even requesting no scopes at all and let the client take on the burden of expressing its needs).

Scopes for Ambiguous API Calls

An API for photo retrieval and usage uses scopes to manage access for a single API call that has two different functions as follows:

- GET low-resolution version for the purpose of viewing: `view` scope
- GET high-resolution version for the purpose of downloading: `download` scope

A GET call doesn't distinguish between the two possible functions. The resource server can request zero scopes, which may be the wisest choice for (say) a paid service or just in the name of least privilege and minimal disclosure.

Resource Owner-Influenced Scopes

Assume the same example as above. Because the resource owner-resource server interface is private (beyond the issuance of the PAT), and the details of which resources are centrally protected and how are allowed to be variable, the resource server could make an option available to the resource owner to keep the the `download` scope "private" – that is, never registered for any of the resource owner's resources (meaning, never advertised by the resource server as part of the universe of possible scopes on this resource).

In this scenario, the resource server should first look at the resource owner-designated scopes before requesting permissions.

This would mean that the authorization server would fail the client with an `invalid_scope` error at the token endpoint if the client requested it (and presumably pre-registered for it). The resource server, too, would receive an `invalid_scope` error if it tried to request a permission it hadn't first registered as part of a resource.

Interpreting Authorization Assessment Set Math

Although authorization assessment is an internal process performed by the authorization server, in UMA V2.0 it gains a large degree of normative precision. This section explains, using symbolic set math.

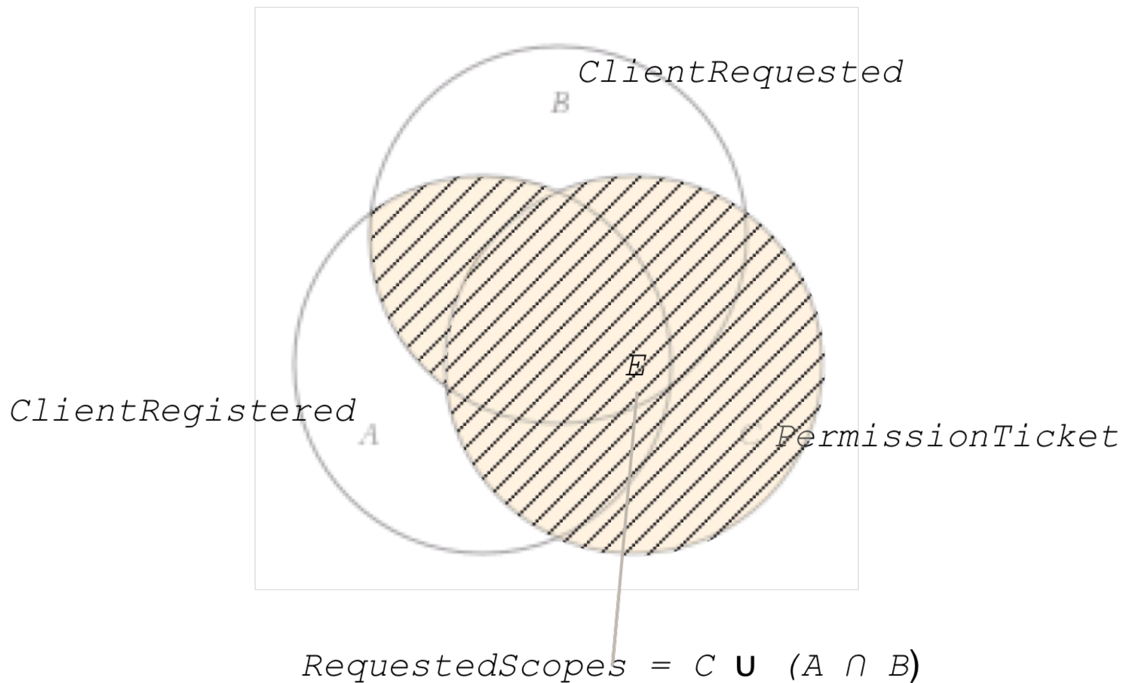
Define a superset ***S*** of all possible assignable scopes to protected resources in a UMA context.

Let *s* be an element of ***S*** (*s* ***S***). Define the following subsets of ***S***:

- *A* = *ClientRegistered* = {*s*, scopes pre-registered at AS by client , *s.t. s* ***S***}, *A* ***S***;
- *B* = *ClientRequested* = {*s*, scopes requested at AS by client, *s.t. s* ***S***}, *B* ***S***;
- *C* = *PermissionTicket* = {*s*, scopes requested at AS by RS on behalf of client, *s.t. s* ***S***}, *C* ***S***;
- *D* = *RSRegistered* = {*s*, scopes registered at AS by RS with a protected resource, *s.t. s* ***S***}, *D* ***S***;

Calculate the set *RequestedScopes* (***E***) as follows:

- ***E*** = *RequestedScopes* = *PermissionTicket* (*ClientRegistered* *ClientRequested*);
- ***E*** = *C* (*A* *B*);



Define the set *SatisfiedPolicyConditions* (***F***) as the set of all scopes for which the client satisfies all relevant policy conditions at the AS.

- ***F*** = *SatisfiedPolicyConditions* = {*s* requesting side satisfies policy conditions *s* *D*};

Calculate the set CandidateGrantedScopes (***G***) as follows:

- ***G*** = *CandidateGrantedScopes* = *RequestedScopes* *SatisfiedPolicyConditions*;
- ***G*** = ***E*** ***F***;

Proceed with the authorization results calculation based on *CandidateGrantedScopes*.

Resource Server Error Handling When the PAT Is Invalid

If the resource server can't get a permission ticket, it issues a 403 Forbidden HTTP error and `Warning: 199 - "UMA Authorization Server Unreachable"`. One reason for being unable to get a permission ticket is that the resource owner's PAT has expired or is otherwise invalid, and it has no way of refreshing the PAT. In this case, the resource server could take this opportunity to initiate some refreshing action such as send a notification to the resource owner and ask them to re-consent to the pairing with the authorization server as required.

Considerations Regarding Resource Owners and Requesting Parties

A resource owner can be a human end-user (natural person) or an organization (legal person). The same is true of a requesting party. Using a client credentials grant to issue the PAT is appropriate when the resource owner is an organization and policy conditions are set either by an administrator or autonomously in some fashion. If the requesting party is an organization, then the client is typically an autonomously running web service, service account, or similar.

See the UMA Legal page for information on ongoing work to connect the technical UMA entities to contracts and trust frameworks that define "access federations" (in contrast to identity federations).

Considerations Regarding Resource Server API Constraints

An API that is designed as follows, counting on an OAuth access token to give all the necessary context, would be problematic. This is because UMA has a client-to-resource-first flow, with permission ticket passing, in order to enable party-to-party delegation:

```
POST /doctors/me HTTP/1.1
```

In the UMA grant flow, the client first attempts access to a protected resource with no token, and the resource server next requests permissions on behalf of that client at the authorization server. In order for the resource server to know which authorization server to approach and which PAT (representing a resource owner) and resource identifier to supply in that request, the API being accessed by the client needs to be structured in such a way that the resource server can derive this information from the client's token-free access attempt. Commonly, this information can be passed through the URI, headers, or body of the client's request. Alternatively, the entire interface could be dedicated to the use of a single resource owner and protected by a single authorization server.

Resource orientation, that is, an API design that uses resource-specific endpoints rather than a single endpoint for all calls of widely differing sorts (level 1 on the Richardson Maturity Model), is a classic way of achieving sufficient context. For example, the following call by a client would be sufficient to indicate that the operation was targeted at a child resource `mjones` (presumably a specific resource owner) associated with a parent resource `doctors`, and if the resource server had previously registered a resource set corresponding to this child resource at authorization server `ABC` and received back a resource set ID of `ABC123`, it could unambiguously select the correct authorization server, PAT, and resource set ID in order to register a requested permission for that client:

```
POST /doctors/mjones HTTP/1.1
```

However, if a resource server has an API that is completely generic per resource owner, such as a singular endpoint that if OAuth-protected would have depended entirely on an OAuth token to convey the user context, a different approach would be needed. It is possible for the resource server to register the singular resource set over and over for each of its many users (each using a different PAT) at an authorization server, getting a unique resource set ID for each in turn. However, the resource server must be prepared to associate some query attribute, HTTP header value, body field, or other artifact coming from the client in a call to the otherwise generic endpoint that can be matched up with the PAT. And the resource server must, of course, provision the necessary API context cue method and the specific resource owner context needed out of band, just as it would have had to provision (or make discoverable) the same information in a more "resource-oriented" form.

It is possible for the resource server to seed discoverability of the resource owner context by populating the `uri` property of the resource set description with a network location that includes, say, a query parameter identifying the resource owner in some fashion. Then the authorization server application would need to either transmit the parameter value to a discovery service, or function as a discovery service itself, or perform some other mapping. If the authorization server application is able to map the network location to a substitute value, such as a one-time code or equivalent, and then report that value back to the resource server application, then they each can provision the code to the requesting party (say, in an email message) or client (say, in an error message) for it to be returned somewhere in the initial access attempt.

Considerations Regarding Resource Registration Timing and Mechanism

No specific timing of initial resource registration is mandated. Three stages suggest themselves as natural resource registration times:

1. On initial resource creation (say, the resource owner uploads a photo to the resource server)
2. On need for policy creation (say, the resource owner wants to apply policy constraints to the photo)
3. On resource access attempt (say, the client attempts to view the photo)

The first stage may result in registering more resources than need to be managed by the authorization server in practical terms. The third stage may forbid the use of certain flows. For example, it would not allow "Alice-to-Bob" sharing flows where Alice is able to put proactive policy conditions in place before Bob attempts access. Thus, the second stage may provide the greatest utility for the greatest number of use cases if it is necessary to pick one choice only. However, any of the stages is viable for different use cases.

Note that in current versions of UMA, the registration mechanism is limited to individual rather than bulk registration. It is possible to imagine use cases at all three stages outlined above where bulk registration could be helpful. However, in the interest of avoiding overly complex design and premature optimization for very large numbers of resources as opposed to manageably small numbers, the Work Group has currently decided to keep only the individual mechanism. Sample use cases include the following:

- The resource server treats a "wildcarded" URI as being a single complex resource for authorization server purposes; this translates to individual registration.
- To enable a human resource owner to share out resources one at a time using a Share button, the resource server would probably need individual registration at stage 2. But to enable "relationship-driven sharing" of (say) multiple smart device resources at once, the resource server

might want to register as many resources as are available in a household. For industrial IoT use cases, the number of resources to register could climb

- In discussions about the FHIR API for healthcare, resource registrations might "pair" with patterns of permission requests that anticipate a need for the requesting side to gain access to certain clusters of resources, say, three related resources if ever a client attempts to access one of them. (E.g., each of the use cases in the GDoc imagining resource server permission requests for various APIs gives us a way of imagining finite numbers.)

Considerations Regarding Scope Discovery

Rather than the resource description document pointing to a series of scope URIs that *must* be dereferenced (as was the case in UMA V1.0.x), the authorization server in UMA V2.0 can instead make use of the OpenID Connect discovery document and its `scopes_supported` metadata item, which, when filled with (the same) scope description document URIs, allows for development-time discovery of the necessary scope information.

Extension Opportunities

UMA presents a number of opportunities for extension. This section discusses some that have arisen in UMA design discussions that generally have not been taken advantage of, but which may be of interest to third parties.

Using Alternate Communications Protocols

In some circumstances, it may be desirable to couple UMA software entity roles tightly. For example, an authorization server application might also need to act as a client application in order to retrieve protected resources so that it can present to resource owners a dashboard-like user interface that accurately guides the setting of policy; it might need to access itself-as-authorization server for that purpose. For another example, the same organization might operate both an authorization server and a resource server that communicate only with each other behind a firewall, and it might seek more efficient communication methods between them.

In other circumstances, it may be desirable to bind UMA flows to transport mechanisms other than HTTP even if entities remain loosely coupled. For example, in Internet of Things scenarios, Constrained Application Protocol (CoAP) may be preferred over HTTP.

In such cases, parts of UMA's flows may require profiling or extension because it is only defined over HTTP. Where appropriate, use the `uma_profiles_supported` configuration property to flag usage of a documented profile or extension.

(See issue #267.)

Resource Registration for OAuth and OpenID Connect

UMA is defined by two specifications. User-Managed Access 2.0 ("Core") makes use of OAuth 2.0 Resource Registration ("RReg"). The latter is meant to be applicable not just to the UMA extension grant of OAuth but also to the other OAuth grants and to OpenID Connect as well, as explained in the introduction to that specification. This extensibility has been designed in to RReg, but it needs to be fully tested.

(TBS - add diagrams for the use cases)

(See issue #273.)

OpenAPI Format for Resource Registration

Currently, a special-purpose data format is used for registering resources and their scopes. The notion of making use of the Swagger-based OpenAPI format has been discussed. (See issue #288.)

Facilitating Chatter Reduction at the Resource Server

Enhancing the information returned by the authorization server to the resource server could enable the latter to respond more efficiently in the case of too-frequent client access attempts. (See issue #282.)

Informing the Authorization Server About Protected Resource Locations

In UMA V2.0, the `uri` property of the resource registration document was removed. Those wishing to use the resource server's communications channel with the authorization server to communicate information about a protected resource's location may be interested to look at this area. (See issue #270.)

Cascading Authorization Servers

A proposal has been made for enabling a cascading series of authorization servers to contributed to the contents of a requesting party token (RPT). (See issue #260.)

Hashed Claims Discovery

A proposal has been made for enabling an authorization server to convey its desired value for a pushed claim to a client in a privacy-sensitive way using a hashed value in a `need_info` response. (See issue #254.)

Resource Baskets

Currently, it is only possible for resource servers to register "flat" resources for protection, and for authorization servers to have no sophisticated understanding of their structure. It would be possible for an extension to resource description documents to description greater structure and relationships. (See issue #31.)

Notification Endpoint

In concert with the technical capabilities of UMA, it would be powerful to require the resource server to notify the authorization server on the resource owner's behalf of certain actions, or actions not taken, either as part of UMA flows (such as refusing to give access even if an RPT grants permission) or outside of UMA flows (such as giving access due to a court order). The Kantara consent receipt standard is one important format that could be made use of together with an endpoint dedicated to such notification. (See the "shoebox issues" (a shoebox being where one might keep all one's receipts).)

Managing Resource Registration Revisions

Regarding the resource registration API, it is common practice when using NoSQL databases to replicate entity tag (ETag HTTP header) revision information in the body of the response message as well, in a `_rev` property. The API does not mandate this property (though an early pre-V1.0 draft did include this property).

Understanding Authorization Server Response Options From the Token Endpoint

When the client requests an RPT from the token endpoint, the authorization server is able to issue the token as requested, deny the request definitively, and so on. You can think of the responses as mapping to well-understood access control actions (for example, in XACML) as follows. (These are non-normative descriptions; see Grant Sec 3.3.5 and 3.3.6 for normative wording.)

Response (e.g., error code)	HTTP status code	Conditions	Meaning	Permission ticket issued?
Issue an RPT	200 OK	Requesting side has met policy conditions	Permit	No
`invalid_grant`	400 Bad Request	Permission ticket in request not found at authorization server, or was expired, or other RFC 6749 conditions	(syntactic error)	No
`invalid_scope`	400 Bad Request	At least one requested scope didn't match any scope on any permissions on permission ticket in request, or at least one requested scope didn't match any scope the client was pre-registered for.	(syntactic error)	No
`need_info`	403 Forbidden	The authorization server needs additional information in order for a request to succeed.	Indeterminate	Yes
`request_denied`	403 Forbidden	The client is not authorized.	Deny	No
`request_submitted`	403 Forbidden	The authorization server requires intervention by the resource owner to determine whether the client is authorized.	NotApplicable	Yes

If the authorization server does not issue a permission ticket with an error, the client must start anew in a fresh authorization process. If the authorization server does issue a permission ticket, the client has a choice whether to continue and use it, or start anew.

Handling Optional and Extension Properties (V1.0.1)

This section is specific to UMA V1.0.1.

Any entity receiving or retrieving a JSON data structure is supposed to ignore extension properties it is unable to understand, and manage property namespaces on its own to avoid collisions. Properties defined in the specifications that are optional to supply, however, are nonetheless required to be handled by the receiving entity.

This section recommends how to deal with optional and extension properties. It is helpful for handling behavior to be consistent because UMA flows involve loosely coupled entities. Typically, an extension property would appear if one of the entities has implemented some agreed extension to the specification that might not apply to this particular transaction.

In the event that an unrecognized property is received, it's a good idea to log the property and its value, taking normal precautions regarding safe methods of logging potentially dangerous properties in order to avoid injection attacks or similar. This will help with any troubleshooting or auditing that may be required, while allowing normal processing to continue. Finally, it's also recommended to log any property that is malformed (for example, where a Boolean value is expected, but a text value is received), taking the same precautions regarding safe methods of logging.

Following are specific comments on optional properties defined in the specifications.

Property (V1.0.1 references)	**Recommendations**
Core Sec 1.4: Authorization server configuration data	
claim_token_profiles_supported	Provided as a hint; no significant impact if ignored by any party. Should be logged if ignored to help with troubleshooting.
uma_profiles_supported	Provided as a hint; no significant impact if ignored by any party. Should be logged if ignored to help with troubleshooting.
dynamic_client_endpoint	**Authorization Server**: implementations should take care to provide this parameter if support for the dynamic client registration feature is provided. Failing to provide it (or providing it erroneously) can induce incorrect handling by clients or resource servers. **Clients, Resource Servers**: if the parameter is **not** provided, clients and resources servers must assume that dynamic client registration is not possible, and should therefore not attempt such registration.
requesting_party_claims_endpoint	**Authorization Server:** to avoid confusion, should provide this parameter if end-user RP claims gathering capability exists.
Core Sec 3.4.2: RPT "Bearer" profile	
exp	**Authorization Server**: since not providing this property implicitly means that the permission does not expire, the AS should take care only to ignore the parameter if a non-expiring permission is desired. It may be sensible to consider always providing a value, even if far in the future, to avoid inadvertently granting permanent permissions. **Resource Server**: if the parameter is provided, it must be adhered to. If it is not, it may be sensible to consider applying an expiry date anyway, to avoid inadvertently allowing permanent access to a given resources. It may also be sensible to log if this parameter is not provided (and, hence, long or permanent permission is given) for audit purposes.
iat	**Authorization Server**: Not providing the issued-at time introduces the potential for confusion at the RS about whether the token is valid or not. **Resource Server**: RS should consider whether the issued-at time is reasonable (allowing for potential clock skew). Ignoring the parameter, if provided, could introduce a risk of incorrectly processing a 'bad' token.
nbf	**Authorization Server**: failure to provide this parameter might result in access to a resource being granted earlier than intended. The AS should consider providing a value to avoid any potential confusion. **Resource Server**: if the parameter is provided, it must be adhered to. If a value is not provided, the RS should assume 'now' as the nbf, and log accordingly for audit purposes.
Core Sec 3.5.4.2: Error Details About Claims	
name	**Authorization Server**: not providing a value might cause processing confusion later. The AS should consider providing this. **Client**: the client should consider using this value when returning any eventual results to the AS, in order to avoid confusion.
friendly_name	**Authorization Server**: no significant impact; although not providing a value means that the client will have to make assumptions about how to present the claim requirement to the user. **Client**: no significant impact; although the client should consider using this value to help provide improved communication to the user.
claim_type	**Authorization Server**: no significant impact. **Client**: no significant impact.
claim_token_format	**Authorization Server**: failing to provide this parameter might result in a token format being return that the AS cannot then process. AS should consider providing this parameter to avoid confusion at the Client. **Client**: if provided, client should take account of the acceptable token formats when it returns a token to the AS. Ignoring this parameter might result in a token being returned in a format which the AS cannot process.
issuer	**Authorization Server**: no significant impact. **Client**: ignoring this parameter, if provided, might result in a token being returned from an issuing authority which is not acceptable to the AS (and so lead to a poor user experience).
Core Sec 3.6.3: Client Redirects Requesting Party to Authorization Server for Claims-Gathering	
claims_redirect_uri	**Authorization Server**: it is recommended to include this parameter to avoid confusion or unexpected results at the AS. **Client**: ignoring this parameter is not recommended.
state	As noted in the spec, it is highly recommended that this parameter be included in order to avoid cross-site request forgery.
ticket (response)	There are no circumstances in which this parameter can reasonably be ignored.

state (response)	There are no circumstances in which this parameter can reasonably be ignored.
Core Sec 4.2: UMA error responses	
error_description	No significant impact.
error_uri	No significant impact.
RSR Sec 2.1: Scope descriptions	
icon_uri	**Resource Server**: no significant impact **Authorization Server**: should log that it is ignoring for troubleshooting purposes.
RSR Sec 2.2: Resource set descriptions	
uri	**Resource Server**: in many deployments, the network location for the resource set being registered will be provided by (or inferable from) the 'scope' parameter (which is required). If not, however, the resource server will most likely use the 'uri' parameter to provide the network location. **Authorization Server**: if the parameter is ignored, this should be logged for troubleshooting. It is unlikely to be ignored in most common scenarios.
type	**Resource Server**: this can be a helpful hint to provide to the AS. **Authorization Server**: should log that it is ignoring for troubleshooting purposes.
icon_uri	**Resource Server**: no significant impact. **Authorization Server**: should log that it is ignoring for troubleshooting purposes.
RSR Sec 3: Error messages	
error_description	**Resource Server**: no significant impact. **Authorization Server**: should log that it is ignoring for troubleshooting purposes.
error_uri	**Resource Server**: no significant impact. **Authorization Server**: should log that it is ignoring for troubleshooting purposes.

Change History

Version	Date	Comment
Current Version (v. 52)	**Aug 28, 2020 16:00**	**Alec L**
v. 51	Jan 29, 2018 19:12	**Eve Maler**
v. 50	Jan 29, 2018 19:11	**Eve Maler**
v. 49	Jan 09, 2018 23:08	**Eve Maler**: Added the Why The PAT? section
v. 48	Jan 09, 2018 20:56	**Eve Maler**: Corrected PDF version of IPR doc link and rationalized IPR header content so it's the same as in Release Notes
v. 47	Jan 09, 2018 18:54	**Eve Maler**
v. 46	Sep 17, 2017 15:43	**Eve Maler**
v. 45	Sep 17, 2017 15:42	**Eve Maler**
v. 44	Aug 20, 2017 18:49	**Eve Maler**
v. 43	Jun 06, 2017 17:08	**Eve Maler**
v. 42	Jun 06, 2017 17:08	**Eve Maler**
v. 41	Mar 13, 2017 00:46	**Eve Maler**

v. 40	Mar 13, 2017 00:28	**Eve Maler**
v. 39	Mar 12, 2017 20:30	**Eve Maler**
v. 38	Mar 12, 2017 20:29	**Eve Maler**
v. 37	Mar 12, 2017 20:01	**Eve Maler**
v. 36	Mar 09, 2017 11:11	**Eve Maler**
v. 35	Mar 09, 2017 11:01	**Eve Maler**
v. 34	Mar 08, 2017 18:35	**Eve Maler**
v. 33	Mar 08, 2017 18:30	**Eve Maler**
v. 32	Mar 08, 2017 18:21	**Eve Maler**
v. 31	Mar 08, 2017 18:10	**Eve Maler**
v. 30	Mar 08, 2017 15:01	**Eve Maler**
v. 29	Mar 08, 2017 14:03	**Eve Maler**
v. 28	Mar 08, 2017 13:19	**Eve Maler**
v. 27	Mar 08, 2017 12:58	**Eve Maler**
v. 26	Mar 08, 2017 09:13	**Eve Maler**
v. 25	Mar 08, 2017 09:11	**Eve Maler**
v. 24	Mar 08, 2017 08:58	**Eve Maler**
v. 23	Mar 07, 2017 23:38	**Eve Maler**
v. 22	Mar 07, 2017 22:38	**Eve Maler**
v. 21	Mar 07, 2017 21:46	**Eve Maler**
v. 20	Mar 07, 2017 21:08	**Eve Maler**
v. 19	Feb 22, 2017 22:06	**Eve Maler**
v. 18	Aug 29, 2016 13:57	**Eve Maler**
v. 17	Oct 13, 2015 20:19	**Eve Maler**
v. 16	Oct 01, 2015 17:23	**Eve Maler**
v. 15	Sep 22, 2015 11:53	**Eve Maler**
v. 14	Sep 22, 2015 11:52	**Eve Maler**
v. 13	Sep 22, 2015 11:52	**Eve Maler**
v. 12	Sep 22, 2015 11:41	**Eve Maler**
v. 11	Sep 21, 2015 22:41	**Eve Maler**
v. 10	Jun 07, 2015 13:48	**Eve Maler**
v. 9	Jun 07, 2015 13:27	**Eve Maler**
v. 8	May 12, 2015 13:29	**Eve Maler**
v. 7	May 12, 2015 13:25	**Eve Maler**

v. 6	Jan 01, 2015 02:43	**Eve Maler**
v. 5	Dec 24, 2014 19:27	**Eve Maler**
v. 4	Dec 20, 2014 17:59	**Eve Maler**
v. 3	Dec 20, 2014 14:21	**Eve Maler**
v. 2	Dec 20, 2014 13:47	**Eve Maler**
v. 1	Dec 20, 2014 13:41	**Eve Maler**