Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 6, 2015

T. Hardjono, Ed.
MIT
E. Maler
ForgeRock
M. Machulak
Cloud Identity
D. Catalano
Oracle

April 4, 2015

User-Managed Access (UMA) Profile of OAuth 2.0
draft-hardjono-oauth-umacore-13

Abstract

User-Managed Access (UMA) is a profile of OAuth 2.0. UMA defines how resource owners can control protected-resource access by clients operated by arbitrary requesting parties, where the resources reside on any number of resource servers, and where a centralized authorization server governs access based on resource owner policies.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 6, 2015.

Copyright Notice

Table of Contents

1. Introduction

User-Managed Access (UMA) is a profile of OAuth 2.0 [OAuth2]. UMA defines how resource owners can control protected-resource access by clients operated by arbitrary requesting parties, where the resources reside on any number of resource servers, and where a centralized authorization server governs access based on resource owner policies. Resource owners configure authorization servers with access policies that serve as asynchronous authorization grants.

UMA serves numerous use cases where a resource owner uses a dedicated service to manage authorization for access to their resources, potentially even without the run-time presence of the resource owner. A typical example is the following: a web user (an end-user resource owner) can authorize a web or native app (a client) to gain one-time or ongoing access to a protected resource containing his home address stored at a "personal data store" service (a resource server), by telling the resource server to respect access entitlements issued by his chosen cloud-based authorization service (an authorization server). The requesting party operating the client might be the resource owner, where the app is run by an e-commerce company that needs to know where to ship a purchased item, or the requesting party might be resource owner's friend who is using an online address book service to collect contact information, or the requesting party might be a survey company that uses an autonomous web service to compile population demographics. A variety of use cases can be found in [UMA-usecases] and [UMA-casestudies].

Practical control of access among loosely coupled parties requires more than just messaging protocols. This specification defines only the "technical contract" between UMA-conforming entities; a companion specification, [UMA-obligations], additionally discusses expected behaviors of parties operating and using these entities. Parties operating entities that claim to be UMA-conforming should provide documentation of any rights and obligations between and among them,

especially as they pertain the concepts and clauses discussed in this companion specification.

In enterprise settings, application access management sometimes involves letting back-office applications serve only as policy enforcement points (PEPs), depending entirely on access decisions coming from a central policy decision point (PDP) to govern the access they give to requesters. This separation eases auditing and allows policy administration to scale in several dimensions. UMA makes use of a separation similar to this, letting the resource owner serve as a policy administrator crafting authorization strategies for resources under their control.

In order to increase interoperable communication among the authorization server, resource server, and client, UMA defines two purpose-built APIs related to the outsourcing of authorization, themselves protected by OAuth (or an OAuth-based authentication protocol) in embedded fashion.

The UMA protocol has three broad phases, as shown in Figure 1.

The Three Phases of the UMA Profile of OAuth

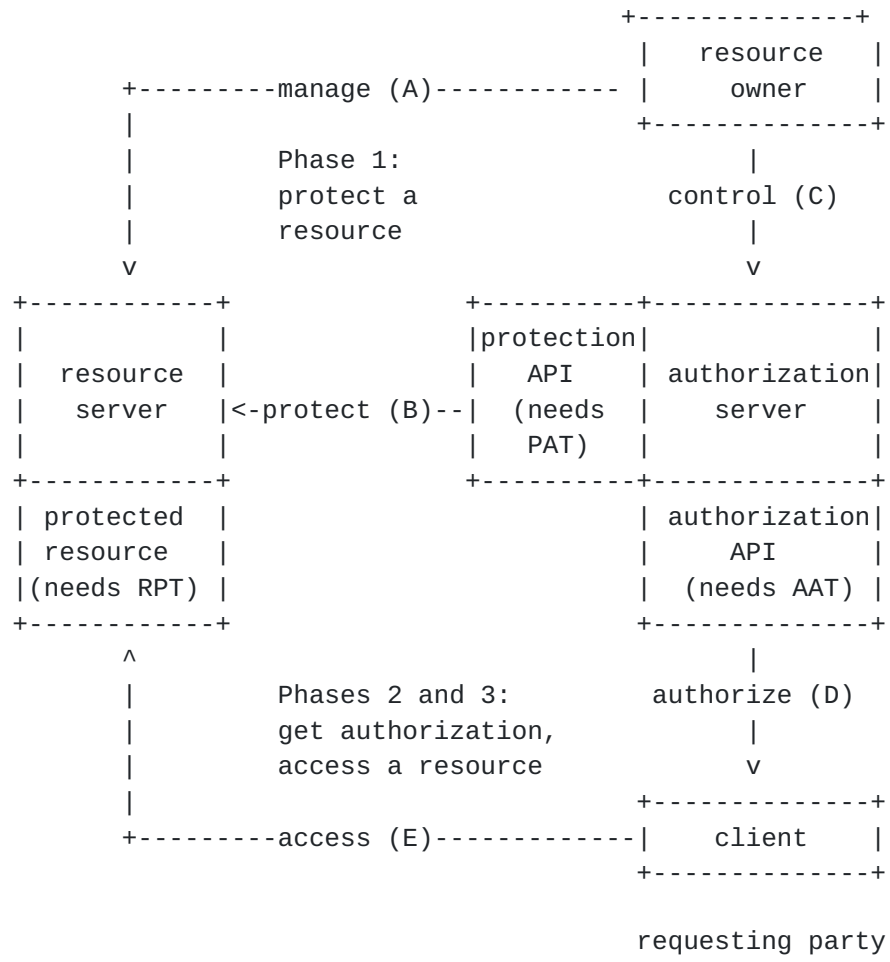

Figure 1

The phases work as follows:

Protect a resource (Described in Section 2.) The resource owner, who manages online resources at the resource server ("A"), introduces it to the authorization server so that the latter can begin protecting the resources. To accomplish this, the authorization server presents a protection API ("B") to the resource server. This API is protected by OAuth (or an OAuth-based authentication protocol) and requires a protection API token (PAT) for access. Out of band, the resource owner configures the authorization server with policies associated with the resource sets ("C") that the resource registers for protection.

Get authorization (Described in Section 3.) The client approaches the resource server seeking access to an UMA-protected resource. In order to access it successfully, the client must first use the

authorization server's authorization API ("D") to obtain authorization data and a requesting party token (RPT) on behalf of its requesting party, and the requesting party may need to supply identity claims. The API is protected by OAuth (or an OAuth-based authentication protocol) and requires an authorization API token (AAT) for access.

Access a resource (Described in Section 3.) The client successfully presents to the resource server an RPT that has sufficient authorization data associated with it, gaining access to the desired resource ("E"). Phase 3 is effectively the "success path" embedded within phase 2.

Implementers have the opportunity to develop profiles (see Section 6) that specify and restrict various UMA protocol, RPT, and identity claim format options, according to deployment and usage conditions.

1.1. Notational Conventions

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this document are to be interpreted as described in [RFC2119].

Unless otherwise noted, all protocol properties and values are case sensitive. JSON [JSON] data structures defined by this specification MAY contain extension properties that are not defined in this specification. Any entity receiving or retrieving a JSON data structure SHOULD ignore extension properties it is unable to understand. Extension names that are unprotected from collisions are outside the scope of this specification.

1.2. Terminology

UMA introduces the following new terms and enhancements of OAuth term definitions.

resource owner
: An OAuth resource owner that is the "user" in User-Managed Access. This is typically an end-user (a natural person) but it can also be a corporation or other legal person.

policy
: The configuration parameters of an authorization server that effect resource access management. Authorization policies typically include elements similar to parts of speech; for example, "subjects" describe those seeking access (requesting parties and clients), "verbs" describe operational scopes of access, and "objects" describe targeted resource sets. Policy

configuration takes place between the resource owner and the authorization server, and thus is out of band of UMA.

requesting party
: An end-user, or a corporation or other legal person, that uses a client to seek access to a protected resource. The requesting party may or may not be the same party as the resource owner.

client
: An application making protected resource requests with the resource owner's authorization and on the requesting party's behalf.

claim
: A statement of the value or values of one or more identity attributes of a requesting party. A requesting party may need to provide claims to an authorization server in order to gain permission for access to a protected resource.

resource set
: One or more protected resources that a resource server manages as a set, abstractly. In authorization policy terminology, a resource set is the "object" being protected. This term derives from [OAuth-resource-reg].

scope
: A bounded extent of access that is possible to perform on a resource set. In authorization policy terminology, a scope is one of the potentially many "verbs" that can logically apply to a resource set ("object"). UMA associates scopes with labeled resource sets.

authorization data
: Data associated with an RPT that enables some combination of the authorization server and resource server to determine the correct extent of access to allow to a client. Authorization data is a key part of the definition of an RPT profile.

authorization server
: A server that issues authorization data and RPTs to a client and protects resources managed at a resource server.

permission
: A scope of access over a particular resource set at a particular resource server that is being requested by, or granted to, a requesting party. In authorization policy terminology, a permission is an entitlement that includes a "subject" (requesting party), "verbs" (one or more scopes of access), and an "object" (resource set). A permission is one

example of authorization data that an authorization server may add to a requesting party token.

permission ticket A correlation handle that is conveyed from an authorization server to a resource server, from a resource server to a client, and ultimately from a client back to an authorization server, to enable the authorization server to assess the correct policies to apply to a request for authorization data.

token A packaged collection of data meant to be transmitted to another entity. A token could be used for authorized access (an "access token" such as an UMA RPT, PAT, or AAT), or could be used to exchange information about a subject (a "claim token" such as one that is conveyed by a client to an authorization server while seeking authorization data).

1.3. Achieving Distributed Access Control

The software components that fill the roles of UMA authorization servers, resource servers, and clients respectively are intended to work in an interoperable fashion when each is operated by an independent party (for example, different organizations). For this reason, UMA specifies communications channels that the authorization server MUST implement as HTTP-based APIs that MUST use TLS and OAuth (or OAuth-based authentication protocol) protection, and that the resource server MUST implement as an HTTP-based interface. UMA's use of TLS is governed by Section 1.6 of [OAuth2], which discusses deployment and adoption characteristics of different TLS versions.

For those OAuth protection use cases where an identity token is desired in addition to an access token, it is RECOMMENDED that an OAuth-based authentication protocol such as OpenID Connect be used.

It is also REQUIRED, in turn, for resource servers and clients on the requesting side of UMA interactions to use these channels, unless a profile is being used that enables API extensibility. The profiles that enable such alternatives are provided in Section 5.

1.3.1. Protection API

The authorization server MUST present an HTTP-based protection API, protected by TLS and OAuth (or an OAuth-based authentication protocol), for use by resource servers. The authorization server thus has an OAuth token endpoint and authorization endpoint. The authorization server MUST declare all of its protection API endpoints in its configuration data (see Section 1.4).

The protection API consists of three endpoints:

- Resource set registration endpoint as defined by [OAuth-resource-reg]
- Permission registration endpoint as defined by Section 3.2
- Token introspection endpoint as defined by [OAuth-introspection] and Section 3.3.1

An entity seeking protection API access MUST have the scope "uma_protection". An access token with at least this scope is called a protection API token (PAT) and an entity that can acquire an access token with this scope is by definition a resource server. A single entity can serve in both resource server and client roles if it has access tokens with the appropriate OAuth scopes. If a request to an endpoint fails due to an invalid, missing, or expired PAT, or requires higher privileges at this endpoint than provided by the PAT, the authorization server responds with an OAuth error.

The authorization server MUST support the OAuth bearer token profile for PAT issuance, and MAY support other OAuth token profiles. It MUST declare all supported token profiles and grant types for PAT issuance in its configuration data. Any OAuth authorization grant type might be appropriate depending on circumstances; for example, the client credentials grant is useful in the case of an organization acting as a resource owner. [UMA-Impl] discusses grant options further.

A PAT binds a resource owner, a resource server the owner uses for resource management, and an authorization server the owner uses for protection of resources at this resource server. It is not specific to any client or requesting party. The issuance of a PAT represents the approval of the resource owner for this resource server to use this authorization server for protecting some or all of the resources belonging to this resource owner.

1.3.2. Authorization API

The authorization server MUST present an HTTP-based authorization API, protected by TLS and OAuth (or an OAuth-based authentication protocol), for use by clients. The authorization server thus has an OAuth token endpoint and authorization endpoint. The authorization server MUST declare its authorization API endpoint in its configuration data (see Section 1.4).

The authorization API consists of one endpoint:

- RPT endpoint as defined in Section 3.4.1

An entity seeking authorization API access MUST have the scope "uma_authorization". An access token with at least this scope is called an authorization API token (AAT) and an entity that can acquire an access token with this scope is by definition a client. A single entity can serve in both resource server and client roles if it has access tokens with the appropriate OAuth scopes. If a request to an endpoint fails due to an invalid, missing, or expired AAT, or requires higher privileges at this endpoint than provided by the AAT, the authorization server responds with an OAuth error.

The authorization server MUST support the OAuth bearer token profile for AAT issuance, and MAY support other OAuth token profiles. It MUST declare all supported token profiles and grant types for AAT issuance in its configuration data. Any OAuth authorization grant type might be appropriate depending on circumstances; for example, the client credentials grant is useful in the case of an organization acting as a requesting party. [UMA-Impl] discusses grant options further.

An AAT binds a requesting party, a client being used by that party, and an authorization server that protects resources this client is seeking access to on this requesting party's behalf. It is not specific to any resource server or resource owner. The issuance of an AAT represents the approval of this requesting party for this client to engage with this authorization server to supply claims, ask for authorization, and perform any other tasks needed for obtaining authorization for access to resources at all resource servers that use this authorization server. The authorization server is able to manage future processes of authorization and claims-caching efficiently for this client/requesting party pair across all resource servers they try to access; however, these management processes are outside the scope of this specification.

1.3.3. Protected Resource Interface

The resource server MAY present to clients whatever HTTP-based APIs or endpoints it wishes. To protect any of its resources available in this fashion using UMA, it MUST require a requesting party token (RPT) with sufficient authorization data for access.

This specification defines one RPT profile, call "bearer" (see Section 3.3.2), which the authorization server MUST support. It MAY support additional RPT profiles, and MUST declare all supported RPT profiles in its configuration data (see Section 1.4).

An RPT binds a requesting party, the client being used by that party, the resource server at which protected resources of interest reside, and the authorization server that protects those resources. It is not specific to a single resource owner, though its internal components are likely to be bound in practice to individual resource owners, depending on the RPT profile in use.

1.3.4. Time-to-Live Considerations

The authorization server has the opportunity to manage the validity periods of access tokens that it issues, their corresponding refresh tokens where applicable, the individual authorization data components associated with RPTs where applicable, and even the client credentials that it issues. Different time-to-live strategies may be suitable for different resource sets and scopes of access, and the authorization server has the opportunity to give the resource owner control over lifetimes of tokens and authorization data issued on their behalf through policy. These options are all outside the scope of this specification.

1.4. Authorization Server Configuration Data

The authorization server MUST provide configuration data in a JSON document that resides in an /uma-configuration directory at its host-meta [hostmeta] location. The configuration data documents conformance options and endpoints supported by the authorization server.

The configuration data has the following properties.

version
: REQUIRED. The version of the UMA core protocol to which this authorization server conforms. The value MUST be the string "1.0".

issuer
: REQUIRED. A URI with no query or fragment component that the authorization server asserts as its issuer identifier. This value MUST be identical to the web location of the configuration data minus the host-meta [hostmeta] and /uma-configuration path components

pat_profiles_supported
: REQUIRED. OAuth access token types supported by this authorization server for PAT issuance. The property value is an array of string values, where each string value (which MAY be a URI) is a token type. Non-URI token type strings defined by OAuth token-defining specifications are privileged. For

example, the type "bearer" stands for the OAuth bearer token type defined in [OAuth-bearer]. The authorization server is REQUIRED to support "bearer", and to supply this value explicitly. The authorization server MAY declare its support for additional PAT profiles.

aat_profiles_supported
: REQUIRED. OAuth access token types supported by this authorization server for AAT issuance. The property value is an array of string values, where each string value (which MAY be a URI) is a token type. Non-URI token type strings defined by OAuth token-defining specifications are privileged. For example, the type "bearer" stands for the OAuth bearer token type defined in [OAuth-bearer]. The authorization server is REQUIRED to support "bearer", and to supply this value explicitly. The authorization server MAY declare its support for additional AAT profiles.

rpt_profiles_supported
: REQUIRED. Profiles supported by this authorization server for RPT issuance. The property value is an array of string values, where each string value is a URI identifying an RPT profile. The authorization server is REQUIRED to support the "bearer" RPT profile defined in Section 3.3.2, and to supply its identifying URI explicitly. The authorization server MAY declare its support for additional RPT profiles.

pat_grant_types_supported
: REQUIRED. OAuth grant types supported by this authorization server in issuing PATs. The property value is an array of string values, where each string value (which MAY be a URI) is a grant type. Non-URI token type strings defined by OAuth grant type-defining specifications are privileged. For example, the type "authorization_code" stands for the OAuth authorization code grant type defined in [OAuth2].

aat_grant_types_supported
: REQUIRED. OAuth grant types supported by this authorization server in issuing AATs. The property value is an array of string values, where each string value (which MAY be a URI) is a grant type. Non-URI token type strings defined by OAuth grant type-defining specifications are privileged. For example, the type "authorization_code" stands for the OAuth authorization code grant type defined in [OAuth2].

claim_token_profiles_supported

OPTIONAL. Claim token format profiles supported by this authorization server. The property value is an array of string values, where each string value MAY be a URI.

uma_profiles_supported
OPTIONAL. UMA profiles supported by this authorization server. The property value is an array of string values, where each string value is a URI identifying an UMA profile. Examples of UMA profiles are the API extensibility profiles defined in Section 5.

dynamic_client_endpoint
OPTIONAL. The endpoint to use for performing dynamic client registration in the case of the use of [DynClientReg], or alternatively the reserved string "openid" in the case of the use of [OIDCDynClientReg]. In the latter case, it is presumed that the resource server or client will discover the dynamic client registration endpoint from the authorization server's published OpenID Provider Configuration Information. The presence of this property indicates authorization server support for dynamic client registration feature; its absence indicates a lack of support.

token_endpoint
REQUIRED. The endpoint URI at which the resource server or client asks the authorization server for a PAT or AAT. A requested scope of "uma_protection" results in a PAT. A requested scope of "uma_authorization" results in an AAT. Usage of this endpoint is defined by [OAuth2].

authorization_endpoint
REQUIRED. The endpoint URI at which the resource server gathers the consent of the end-user resource owner or the client gathers the consent of the end-user requesting party for issuance of a PAT or AAT respectively, if the "authorization_code" grant type is used. Usage of this endpoint is defined by [OAuth2].

requesting_party_claims_endpoint
OPTIONAL. The endpoint URI at which the authorization server interacts with the end-user requesting party to gather claims. If this property is absent, the authorization server does not interact with the end-user requesting party for claims gathering.

introspection_endpoint
REQUIRED. The endpoint URI at which the resource server introspects an RPT presented to it by a client. Usage of this

endpoint is defined by [OAuth-introspection] and Section 3.3.1. A valid PAT MUST accompany requests to this protected endpoint.

resource_set_registration_endpoint
: REQUIRED. The endpoint URI at which the resource server registers resource sets to put them under authorization manager protection. Usage of this endpoint is defined by [OAuth-resource-reg] and Section 2. A valid PAT MUST accompany requests to this protected endpoint.

permission_registration_endpoint
: REQUIRED. The endpoint URI at which the resource server registers a requested permission that would suffice for a client's access attempt. Usage of this endpoint is defined by Section 3.2. A valid PAT MUST accompany requests to this protected endpoint.

rpt_endpoint
: REQUIRED. The endpoint URI at which the client asks for authorization data. Usage of this endpoint is defined in Section 3.4. A valid AAT and a permission ticket MUST, and an RPT MAY, accompany requests to this protected endpoint.

Example of authorization server configuration data that resides at https://example.com/.well-known/uma-configuration (note the use of https: for endpoints throughout):


```
{
"version":"1.0",
"issuer":"https://example.com",
"pat_profiles_supported":["bearer"],
"aat_profiles_supported":["bearer"],
"rpt_profiles_supported":
["https://docs.kantarainitiative.org/uma/profiles/uma-token-bearer-1.0"],
"pat_grant_types_supported":["authorization_code"],
"aat_grant_types_supported":["authorization_code"],
"claim_token_profiles_supported":["https://example.com/claims/formats/token1"],
"dynamic_client_endpoint":"https://as.example.com/dyn_client_reg_uri",
"token_endpoint":"https://as.example.com/token_uri",
"authorization_endpoint":"https://as.example.com/authz_uri",
"requesting_party_claims_endpoint":"https://as.example.com/rqp_claims_uri",
"resource_set_registration_endpoint":"https://as.example.com/rs/rsrc_uri",
"introspection_endpoint":"https://as.example.com/rs/status_uri",
"permission_registration_endpoint":"https://as.example.com/rs/perm_uri",
"rpt_endpoint":"https://as.example.com/client/rpt_uri"
}
```

Where this specification does not already require optional features to be documented, it is RECOMMENDED that authorization server deployers document any profiled or extended features explicitly and use configuration data to indicate their usage.

2. Protecting a Resource

The resource owner, resource server, and authorization server perform the following actions to put resources under protection. This list assumes that the resource server has discovered the authorization server's configuration data and endpoints as needed.

1. The authorization server issues client credentials to the resource server. It is OPTIONAL for the client credentials to be provided dynamically through [DynClientReg] or [OIDCDynClientReg]; alternatively, they MAY use a static process.

2. The resource server acquires a PAT from the authorization server. It is OPTIONAL for the resource owner to introduce the resource server to the authorization server dynamically (for example, through a "NASCAR"-style user interface where the resource owner selects a chosen authorization server); alternatively, they MAY use a static process that may or may not directly involve the resource owner at introduction time.

3. In an ongoing fashion, the resource server registers any resource sets with the authorization server for which it intends to outsource protection, using the resource set registration endpoint of the protection API (see [OAuth-resource-reg]).

Note: The resource server is free to offer the option to protect any subset of the resource owner's resources using different authorization servers or other means entirely, or to protect some resources and not others. Additionally, the choice of protection regimes can be made explicitly by the resource owner or implicitly by the resource server. Any such partitioning by the resource server or owner is outside the scope of this specification.

Once a resource set has been placed under authorization server protection through the registration of a resource set description for it, and until such a description's deletion by the resource server, the resource server MUST limit access to corresponding resources, requiring sufficient authorization data associated with client-presented RPTs by the authorization server (see Section 3.1.2).

3. Getting Authorization and Accessing a Resource

An authorization server orchestrates and controls clients' access (on their requesting parties' behalf) to a resource owner's protected resources at a resource server, under conditions dictated by that resource owner.

The process of getting authorization and accessing a resource always begins with the client attempting access at a protected resource endpoint at the resource server. How the client came to learn about this endpoint is out of scope for this specification. The resource owner might, for example, have advertised its availability publicly on a blog or other website, listed it in a discovery service, or emailed a link to a particular intended requesting party.

The resource server responds to the client's access request with whatever its application-specific resource interface defines as a success response, either immediately if the client has sufficient authorization, or having first performed one or more embedded interactions with the authorization server and client in the case of a failed access attempt.

A high-level summary of the interactions is as follows. The recipient of each request message SHOULD respond unless it detects a security concern, such as a suspected denial of service attack that can be mitigated by rate limiting.

- The client attempts to access a protected resource.
  - If the access attempt is unaccompanied by an RPT, the resource server registers a requested permission at the authorization server that would suffice for the access attempt, and then responds with an HTTP 403 (Forbidden) response, a permission ticket, and instructions on where to go to obtain an RPT and authorization data.
  - If the access attempt is accompanied by an RPT, the resource server checks the RPT's status.
    - If the RPT is invalid, or if the RPT is valid but has insufficient authorization data, the resource server registers a requested permission at the authorization server that would suffice for the access attempt, and then responds with an HTTP 403 (Forbidden) response, a permission ticket, and instructions on where to go to obtain a valid RPT and authorization data for it.

    + If the RPT is valid, and if the authorization data associated with the token is sufficient for allowing access, the resource server responds with an HTTP 2xx (Success) response.

o If the client received a 403 response and a permission ticket, it asks the authorization server for authorization data that matches the ticket using the RPT endpoint of the authorization API. If the authorization server needs requesting party claims in order to assess this client's authorization, it engages in a claims-gathering flow.

    * If the client does not already have an AAT at the appropriate authorization server to be able to use its authorization API, it first obtains one.

The interactions are described in detail in the following sections.

3.1. Client Attempts to Access Protected Resource

This interaction assumes that the resource server has previously registered one or more resource sets that correspond to the resource the client is attempting to access.

The client attempts to access a protected resource (for example, when an end-user requesting party clicks on a thumbnail representation of the resource to retrieve a larger version). It is expected to discover, or be provisioned or configured with, knowledge of the protected resource and its location out of band. Further, the client is expected to acquire its own knowledge about the application-specific methods made available by the resource server for operating on this protected resource (such as viewing it with a GET method, or transforming it with some complex API call).

The access attempt either is or is not accompanied by an RPT.

3.1.1. Client Presents No RPT

Example of a request carrying no RPT:

```
GET /album/photo.jpg HTTP/1.1
Host: photoz.example.com
...
```

If the client does not present an RPT with the request, the resource server uses the protection API to register a requested permission with the authorization server that would suffice for the access attempt (see Section 3.2), and receives a permission ticket back in

response. It then responds to the client. It SHOULD respond with the HTTP 403 (Forbidden) status code, providing the authorization server's URI in an "as_uri" property in the header, along with the just-received permission ticket in the body in a JSON-encoded "ticket" property. Responses that use any code other than 403 are undefined by this specification; any common or best practices for returning other status codes will be documented in the [UMA-Impl].

For example:

```
HTTP/1.1 403 Forbidden
WWW-Authenticate: UMA realm="example",
  as_uri="https://as.example.com"

{
"ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
}
   ...
```

3.1.2. Client Presents RPT

Example of a request carrying an RPT using the UMA "bearer" RPT profile:

```
GET /album/photo.jpg HTTP/1.1
Authorization: Bearer vF9dft4qmT
Host: photoz.example.com
...
```

If the client presents an RPT with its request, the resource server MUST determine the RPT's status (see Section 3.3) before responding.

If the RPT is invalid, or if the RPT is valid but has insufficient authorization data for the type of access sought, the resource server uses the protection API to register a requested permission with the authorization server that would suffice for the access attempt (see Section 3.2), and receives a permission ticket back in response. It then responds to the client with the HTTP 403 (Forbidden) status code, providing the authorization server's URI in an "as_uri" property in the header, along with the just-received permission ticket in the body in a JSON-encoded "ticket" property.

Example of the resource server's response after having registered a requested permission and received a ticket:

```
HTTP/1.1 403 Forbidden
WWW-Authenticate: UMA realm="example",
  as_uri="https://as.example.com"
  error="insufficient_scope"

{
"ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
}
```

If the RPT's status is associated with authorization data that is sufficient for the access sought by the client, the resource server MUST give access to the desired resource.

Example of the resource server's response after having determined that the RPT is valid and associated with sufficient authorization data:

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
...

/9j/4AAQSkZJRgABAgAAZABkAAD/7AARRHVja
3kAAQAEAAAAPAAA/+4ADkFkb2JlAGTAAAAAAf
/bAIQABgQEBAUEBgUFBgkGBQYJCwgGBggLDAo
KCwoKDBAMDAwMDAwQDA4PEA8ODBMTFBQTExwb
```

The resource server MUST NOT give access where the token's status is not associated with sufficient authorization data for the attempted scope of access.

3.2. Resource Server Registers Requested Permission With Authorization Server

The resource server uses the protection API's permission registration endpoint to register a requested permission with the authorization server that would suffice for the client's access attempt. The authorization server returns a permission ticket for the resource server to give to the client in its response. The PAT provided in the API request implicitly identifies the resource owner ("subject") to which the permission applies.

Note: The resource server is free to choose the extent of the requested permission that it registers, as long as it minimally suffices for the access attempted by the client. For example, it can choose to register a permission that covers several scopes or a

resource set that is greater in extent than the specific resource that the client attempted to access. Likewise, the authorization server is ultimately free to choose to partially fulfill the elements of a permission request based on incomplete satisfaction of policy criteria, or not to fulfill the request.

The resource server uses the POST method at the endpoint. The body of the HTTP request message contains a JSON object providing the requested permission, using a format derived from the scope description format specified in [OAuth-resource-reg], as follows. The object has the following properties:

resource_set_id REQUIRED. The identifier for a resource set to which this client is seeking access. The identifier MUST correspond to a resource set that was previously registered.

scopes REQUIRED. An array referencing one or more identifiers of scopes to which access is needed for this resource set. Each scope identifier MUST correspond to a scope that was registered by this resource server for the referenced resource set.

Example of an HTTP request that registers a requested permission at the authorization server's permission registration endpoint, with a PAT in the header:

```
POST /host/scope_reg_uri/photoz.example.com HTTP/1.1
Content-Type: application/json
Host: as.example.com
Authorization: Bearer 204c69636b6c69

{
  "resource_set_id": "112210f47de98100",
  "scopes": [
      "http://photoz.example.com/dev/actions/view",
      "http://photoz.example.com/dev/actions/all"
  ]
}
```

If the registration request is successful, the authorization server responds with an HTTP 201 (Created) status code and includes the "ticket" property in the JSON-formatted body.

The permission ticket is a short-lived opaque structure whose form is determined by the authorization server. The ticket value MUST be securely random (for example, not merely part of a predictable sequential series), to avoid denial-of-service attacks. Since the ticket is an opaque structure from the point of view of the client, the authorization server is free to include information regarding

expiration time or any other information within the opaque ticket for its own consumption. When the client subsequently uses the authorization API to ask the authorization server for authorization data to be associated with its RPT, it will submit this ticket to the authorization server.

For example:

```
HTTP/1.1 201 Created
Content-Type: application/json
...

{
"ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
}
```

If the registration request is authenticated properly but fails due to other reasons, the authorization server responds with an HTTP 400 (Bad Request) status code and includes one of the following UMA error codes (see Section 4.2):

invalid_resource_set_id The provided resource set identifier was not found at the authorization server.

invalid_scope At least one of the scopes included in the request was not registered previously by this resource server.

3.3. Resource Server Determines RPT's Status

The resource server MUST determine a received RPT's status, including both whether it is active and, if so, its associated authorization data, before giving or refusing access to the client. An RPT is associated with a set of authorization data that governs whether the client is authorized for access. The token's nature and format are dictated by its profile; the profile might allow it to be self-contained, such that the resource server is able to determine its status locally, or might require or allow the resource server to make a run-time introspection request of the authorization server that issued the token.

This specification makes one type of RPT REQUIRED for the authorization server to support: the UMA bearer token profile, as defined in Section 3.3.2. Implementers MAY define and use other RPT profiles.

3.3.1. Token Introspection

Within any RPT profile, when a resource server needs to introspect a token in a non-self-contained way to determine its status, it MAY require, allow, or prohibit use of the OAuth token introspection endpoint (defined by [OAuth-introspection]) that is part of the protection API, and MAY profile its usage. The resource server MUST use the POST method in interacting with the endpoint, not the GET method also defined by [OAuth-introspection].

3.3.2. RPT Profile: Bearer

This section defines the UMA bearer token profile. Following is a summary:

- Identifying URI: https://docs.kantarainitiative.org/uma/profiles/uma-token-bearer-1.0
- Profile author and contact information: Thomas Hardjono (hardjono@mit.edu)
- Updates or obsoletes: None; this profile is new.
- Keyword in HTTP Authorization header: "Bearer".
- Syntax and semantics of token data: As defined below; an opaque string value, resolving to an extended JSON Web Token (JWT) [JWT] format on introspection at the authorization server.
- Token data association: The on-the-wire token is opaque; it is introspected at run time by the resource server through profiled use of the OAuth token introspection endpoint [OAuth-introspection], as defined below.
- Token data processing: As defined in this section and throughout Section 3 of this specification.
- Grant type restrictions: None.
- Error states: As defined below.
- Security and privacy considerations: As defined in this section, throughout Section 3, and in Section 8.

An example of a client making a request with an RPT using the "Bearer" scheme appears in Section 3.1.2.

On receiving an RPT with the "Bearer" scheme in an authorization header from a client making an access attempt, the resource server introspects the token by using the token introspection endpoint of the protection API. The PAT used by the resource server to make the introspection request provides resource-owner context to the authorization server.

The authorization server responds with a JSON object with the structure dictated by [OAuth-introspection]. If the "active" property has a Boolean value of true, then the JSON object MUST NOT contain a "scope" claim, and MUST contain an extension property with the name "permissions" that contains an array of zero or more values, each of which is an object consisting of these properties:

resource_set_id REQUIRED. A string that uniquely identifies the resource set, access to which has been granted to this client on behalf of this requesting party. The identifier MUST correspond to a resource set that was previously registered as protected.

scopes REQUIRED. An array referencing one or more URIs of scopes to which access was granted for this resource set. Each scope MUST correspond to a scope that was registered by this resource server for the referenced resource set.

exp OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this permission will expire. If the property is absent, the permission does not expire. If the token-level "exp" value pre-dates a permission-level "exp" value, the former overrides the latter.

iat OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this permission was originally issued. If the token-level "iat" value post-dates a permission-level "iat" value, the former overrides the latter.

nbf OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating the time before which this permission is not valid. If the token-level "nbf" value post-dates a permission-level "nbf" value, the former overrides the latter.

Example:

```
HTTP/1.1 200 OK
   Content-Type: application/json
   Cache-Control: no-store

   {
    "active": true,
    "exp": 1256953732,
    "iat": 1256912345,
    "permissions": [
      {
        "resource_set_id": "112210f47de98100",
        "scopes": [
          "http://photoz.example.com/dev/actions/view",
          "http://photoz.example.com/dev/actions/all"
         ],
        "exp" : 1256953732
      }
    ]
   }
```

3.4. Client Seeks Authorization for Access

In order to access a protected resource successfully, a client needs to present a valid RPT with sufficient authorization data for access. To get to this stage requires a number of previously successful steps:

1. The authorization server issues client credentials to the client. It is OPTIONAL for the client credentials to be provided dynamically through [DynClientReg] or [OIDCDynClientReg]; alternatively, they MAY use a static process.

2. The client acquires an AAT.

3. The client uses the authorization API to acquire an RPT and to ask for authorization data, providing the permission ticket it got from the resource server. The authorization server associates authorization data with the RPT based on the permission ticket, the resource owner's operative policies, and the results of any claims-gathering flows.

3.4.1. Client Requests Authorization Data

Once in possession of a permission ticket and an AAT for this authorization server, the client asks the authorization server to give it authorization data corresponding to that permission ticket.

It performs a POST on the RPT endpoint, supplying its own AAT in the header and a JSON object in the body with a "ticket" property containing the ticket as its value.

If the client had included an RPT in its failed access attempt, It MAY also provide that RPT in an "rpt" property in its request to the authorization server.

In circumstances where the client needs to provide requesting party claims to the authorization server, it MAY also include a "claim_tokens" property in its request; see Section 3.4.1.2.1 for more information.

Example of a request message containing an AAT, an RPT, and a permission ticket:

```
POST /authz_request HTTP/1.1
Host: as.example.com
Authorization: Bearer jwfLG53^sad$#f
...

{
 "rpt": "sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv",
 "ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
}
```

The authorization server uses the ticket to look up the details of the previously registered requested permission, maps the requested permission to operative resource owner policies based on the resource set identifier and scopes associated with it, potentially requests additional information, and ultimately responds positively or negatively to the request for authorization data.

The authorization server bases the issuing of authorization data on resource owner policies. These policies thus amount to an asynchronous OAuth authorization grant. The authorization server is also free to enable the resource owner to set policies that require the owner to interact with the server in near-real time to provide consent subsequent to an access attempt. All such processes are outside the scope of this specification.

Once the authorization server adds the requested authorization data, it returns an HTTP 200 (OK) status code with a response body containing the RPT with which it associates the requested authorization data. If the client did not present an RPT in the request for authorization data, the authorization server creates and returns a new RPT. If the client did present an RPT in the request, the authorization server returns the RPT with which it associated the

requested authorization data, which MAY be either the RPT that was in the request or a new one. Note: It is entirely an implementation issue whether the returned RPT is the same one that appeared in the request or a new RPT, and it is also an implementation issue whether the AS chooses to invalidate or retain the validity of the original RPT or any authorization data that was previously added to that RPT; to assist in client interoperability and token caching expectations, it is RECOMMENDED that authorization servers document their practices. [UMA-Impl] discusses the implications.

Example:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "rpt": "sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv"
}
```

If the authorization server does not add the requested authorization data, it responds using one of the following UMA error codes and corresponding HTTP status codes (see Section 4.2):

invalid_ticket The provided ticket was not found at the authorization server. The authorization server responds with the HTTP 400 (Bad Request) status code.

expired_ticket The provided ticket has expired. The authorization server responds with the HTTP 400 (Bad Request) status code.

not_authorized The client is not authorized to have this authorization data added. The authorization server responds with the HTTP 403 (Forbidden) status code.

need_info The authorization server needs additional information in order to determine whether the client is authorized to have this authorization data. The authorization server responds with the HTTP 403 (Forbidden) status code. It MAY also respond with an "error_details" object that contains one or more sub-properties with hints about the nature of further required information. The client then has the opportunity to engage in follow-on flows to continue seeking authorization, in a process sometimes referred as "trust elevation". This specification defines two nonexclusive "error_details" sub-properties: "authentication_context", described in Section 3.4.1.1, and "requesting_party_claims", described in Section 3.4.1.2.

request_submitted The authorization server requires intervention by the resource owner to determine whether the client is authorized to have this authorization data. Further immediate interaction between the client and authorization server is out of scope of this specification.

Example when the ticket has expired:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
...

{
  "error": "expired_ticket"
}
```

Example of a "need_info" response with a full set of "error_details" hints:

```
HTTP/1.1 403 Forbidden
Content-Type: application/json
Cache-Control: no-store
...

{
 "error": "need_info",
 "error_details": {
   "authentication_context": {
     "required_acr": ["https://example.com/acrs/LOA3.14159"]
   },
   "requesting_party_claims": {
     "required_claims": [
       {
         "name": "email23423453ou453",
         "friendly_name": "email",
         "claim_type": "urn:oid:0.9.2342.19200300.100.1.3",
         "claim_token_format":
["http://openid.net/specs/openid-connect-core-1_0.html#HybridIDToken"],
         "issuer": ["https://example.com/idp"]
       }
     ],
     "redirect_user": true,
     "ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
   }
 }
}
```

3.4.1.1. Authentication Context Flows

The "authentication_context" sub-property provides hints about additional requirements regarding the requesting party's authentication that underlies the issuance of the currently valid AAT. On receiving such hints, the client has the opportunity to redirect the requesting party to the authorization server to reauthenticate in a manner anticipated to be more successful for gaining access. Such an action is sometimes referred to as "step-up" authentication. The "authentication_context" sub-property contains the following parameter:

required_acr REQUIRED. An array of strings specifying a set of acceptable authentication context class reference values. Any one of the referenced authentication context classes (sets of authentication methods or procedures considered to be equivalent in a particular context) would satisfy the requesting party authentication requirements. Each string MAY be a URI, including one that has been registered through [RFC6711].

3.4.1.2. Claims-Gathering Flows

The "requesting_party_claims" sub-property provides hints about additional requirements regarding information the authorization server needs about the requesting party. On receiving such hints, the client has the opportunity to engage in claims-gathering flows of various types. The "requesting_party_claims" sub-property MAY contain the following parameters, where at least one of "required_claims" or "redirect_user" MUST be supplied:

required_claims An array containing objects that describe characteristics of the required claims, with the following properties:

name OPTIONAL. A string (which MAY be a URI) representing the name of the claim; the "key" in a key-value pair.

friendly_name OPTIONAL. A string that provides a more human-readable form of the attribute's name, which may be useful as a "display name" for use in user interfaces in cases where the actual name is complex or opaque, such as an OID or a UUID.

claim_type OPTIONAL. A string, indicating the expected interpretation of the provided claim value. The string MAY be a URI.

claim_token_format OPTIONAL. An array of strings specifying a set of acceptable formats for a token pushed by the client

containing this claim (see Section 3.4.1.2.1). Any one of the referenced formats would satisfy the authorization server's requirements. Each string MAY be a URI.

issuer OPTIONAL. An array of strings specifying a set of acceptable issuing authorities for the claim. Any one of the referenced authorities would satisfy the authorization server's requirements. Each string MAY be a URI.

redirect_user A Boolean value indicating whether the requesting party's presence at the authorization server is required for the process of claims gathering. For example, the authorization server may require the requesting party to fill out a CAPTCHA to help prove humanness. The default is false if this parameter is not present. See Section 1.4 for how the authorization server declares the requesting party claims endpoint to which the client has the opportunity to redirect the requesting party. Note that the word "user" implies a human requesting party; if the requesting party is not an end-user, then no client action would be possible on receiving the hint.

ticket The permission ticket that was in the client's request for authorization data. If the authorization server provides the "redirect_user" property, it MUST also provide the "ticket" property. This helps the client avoid maintaining this state information after the redirect.

An example of the use of these properties appears in Section 3.4.1.

The authorization server has many options for gathering requesting party claims. For example, it could interact with an end-user requesting party directly, or accept claims delivered by a client, or perform a lookup in some external system. The process is extensible and can have dependencies on the type of requesting party (for example, natural person or legal person) or client (for example, browser, native app, or autonomously running web service).

The client and authorization server have two nonexclusive claims-gathering interaction patterns: push and redirect.

3.4.1.2.1. Client Pushes Claim Tokens to Authorization Server

If the client is ?claims-aware? and the authorization server can accept pushed claims (for example, as it might have indicated by providing "requesting_party_claims" hints described in Section 3.4.1), the client has the option to _push_ claim tokens to the RPT endpoint. The claim token can reflect the client's role as a

federated identity provider, a federated relying party, or an application integrated with a native identity repository.

If the client is aware of the authorization server's requirements for claims through an out-of-band relationship, the client MAY push claim tokens in an initial interaction with the RPT endpoint.

The client supplies claim tokens in the body of the authorization data request message by providing, in addition to the "rpt" and "ticket" properties, the following property:

claim_tokens REQUIRED. An array of objects with the following properties:

format REQUIRED. A string specifying the format of the accompanying claim tokens. The string MAY be a URI.

token REQUIRED. A string containing the claim information in the indicated format, base64url encoded. If claim token format features are included that require special interpretation, the client and authorization server are assumed to have a prior relationship that establishes how to interpret these features. For example, if the referenced format equates to SAML 2.0 assertions and the claim token contains audience restrictions, it is the joint responsibility of the client and authorization server to determine the proper audience values that enable successful token consumption.

Example:

```
POST /rpt_authorization HTTP/1.1
Host: www.example.com
Authorization: Bearer jwfLG53^sad$#f
...
{
    "rpt": "sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv",
    "ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de",
    "claim_tokens": [
      {
        "format":
"http://openid.net/specs/openid-connect-core-1_0.html#HybridIDToken",
        "token": "..."
      }
    ]
}
```

This specification provides a framework for extensibility through claim token format profiling. The authorization server MAY support

any number of claim token profiles, and SHOULD document the claim token profiles it supports in its configuration data.

3.4.1.2.2. Client Redirects Requesting Party to Authorization Server

If the client is ?claims-unaware? and the authorization server has declared a requesting party claims endpoint in its configuration data, or if the authorization server requires direct interaction with the requesting party as part of its claims-gathering process (for example, as it might have indicated through the "redirect_user" hint described in Section 3.4.1), the client has the option to _redirect_ an end-user requesting party to the requesting party claims endpoint. In this case, the authorization server might be a relying party in a federated identity interaction, or it might connect to a directory or other user repository, or even interact with the user in other ways, such as presenting a questionnaire in a web form. After this process completes, the authorization server redirects the end-user requesting party back to the client.

The client constructs the request URI by adding the following parameters to the query component of the requesting party claims endpoint URI using the "application/x-www-form-urlencoded" format:

client_id REQUIRED. The client's identifier issued by the authorization server.

redirect_uri OPTIONAL. The URI to which the client wishes the authorization server to direct the requesting party's user agent after completing its interaction. The URI MUST be absolute, MAY contain an "application/x-www-form-urlencoded" formatted query parameter component that MUST be retained when adding addition parameters, and MUST NOT contain a fragment component. The authorization server SHOULD require all clients to register their redirection endpoint prior to utilizing the authorization endpoint. If the URI is pre-registered, this URI MUST exactly match one of the pre-registered redirection URIs, with the matching performed as described in Section 6.2.1 of [RFC3986] (Simple String Comparison).

ticket REQUIRED. The permission ticket associated with the client's current request for authorization data for this requesting party. The authorization server MUST return this parameter back to when the authorization_state is need_info.

state OPTIONAL. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user agent back to the

client. The use of this parameter is STRONGLY RECOMMENDED for preventing cross-site request forgery.

Example of a request issued by a client application (line breaks are shown only for display convenience):

```
GET /rqp_claims?client_id=some_client_id&state=abc
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fredirect HTTP/1.1
Host: as.example.com
```

At the conclusion of its interaction with the requesting party, the authorization server returns the user agent to the client adding the following parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded" format:

authorization_state REQUIRED. Indicates that the authorization server completed its claims-gathering interaction with the requesting party with the indicated state:

claims_submitted The client is free to return to the RPT endpoint to seek authorization data once again.

not_authorized The client is not authorized to have the desired authorization data added.

need_info The authorization server needs additional information in order to determine whether the client is authorized to have this authorization data. This response directs the client to return to the RPT endpoint, where it might be provided with error_details hints about additional information needed.

request_submitted The authorization server requires intervention by the resource owner to determine whether authorization data can be added. Further immediate interaction between the client, requesting party, and authorization server is out of scope of this specification.

ticket OPTIONAL. The same permission ticket value that the client provided in the request. It MUST be present if and only if the authorization_state is need_info.

state OPTIONAL. The same state value that the client provided in the request. It MUST be present if and only if the client provided it.

The client MUST ignore unrecognized response parameters. If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the

authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user agent to the invalid redirection URI. If the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding an "error" parameter to the query component of the redirection URI using the "application/x-www-form-urlencoded" format, containing one of the following ASCII error codes:

invalid_request The request is missing a required parameter, includes an invalid parameter value (such as an invalid or expired ticket), includes a parameter more than once, or is otherwise malformed.

server_error The authorization server encountered an unexpected condition that prevented it from fulfilling the request. (This error code is needed because an HTTP 500 (Internal Server Error) status code cannot be returned to the client via an HTTP redirect.)

temporarily_unavailable The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server. (This error code is needed because an HTTP 503 (Service Unavailable) status code cannot be returned to the client via an HTTP redirect.)

4. Error Messages

Ultimately the resource server is responsible for either granting the access the client attempted, or returning an error response to the client with a reason for the failure. [OAuth2] defines several error responses for a resource server to return. UMA makes use of these error responses, but requires the resource server to "outsource" the determination of some error conditions to the authorization server. This specification defines additional UMA-specific error responses that the authorization server may give to the resource server and client as they interact with it, and that the resource server may give to the client.

4.1. OAuth Error Responses

When a resource server or client attempts to access one of the authorization server endpoints or a client attempts to access a protected resource at the resource server, it has to make an authenticated request by including an OAuth access token in the HTTP request as described in [OAuth2] Section 7.2.

If the request failed authentication, the authorization server or the resource server responds with an OAuth error message as described in this specification in Section 3.

4.2. UMA Error Responses

When a resource server or client attempts to access one of the authorization server endpoints or a client attempts to access a protected resource at the resource server, if the request is successfully authenticated by OAuth means, but is invalid for another reason, the authorization server or resource server responds with an UMA error response by adding the following properties to the entity body of the HTTP response:

error REQUIRED. A single error code. Values for this property are defined throughout this specification.

error_description OPTIONAL. Human-readable text providing additional information.

error_uri OPTIONAL. A URI identifying a human-readable web page with information about the error.

The following is a common error code that applies to several UMA-specified request messages:

invalid_request The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed. The authorization server MUST respond with the HTTP 400 (Bad Request) status code.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
...

{
  "error": "invalid_request",
  "error_description": "There is already a resource with this identifier.",
  "error_uri": "https://as.example.com/errors/resource_exists"
}
```

5. Profiles for API Extensibility

In some circumstances, it may be desirable to couple UMA 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.

This section defines profiles that allow inter-role communications channels and methods to vary in these circumstances. This specification still REQUIRES authorization servers to issue PATs, AATs, and RPTs and associate authorization data with RPTs, and REQUIRES resource servers to give clients access only when RPTs are associated with sufficient authorization data. This is because, although tokens might not always appear on the wire in the normal fashion, the tokens may represent binding obligations that involve additional parties unable to take part in these optimization opportunities (see [UMA-obligations]).

Where alternate communications channels are being used between independently implemented system entities, it is RECOMMENDED, for reasons of implementation interoperability, to define concrete extension profiles that build on these extensibility profiles (see Section 6.1).

5.1. Protection API Extensibility Profile

This section defines a profile for UMA where the authorization server and resource server roles either reside in the same system entity or otherwise have a privileged or specialized communications channel between them. Following is a summary:

- Identifying URI: https://docs.kantarainitiative.org/uma/profiles/prot-ext-1.0
- Profile author and contact information: Mark Dobrinic (mdobrinic@cozmanova.com)

- Updates or obsoletes: None; this profile is new.
- Security considerations: See below.
- Privacy considerations: See below.
- Error states: None additional.

Using this profile, the resource server MAY use means other than the HTTP-based protection API that is protected by TLS and OAuth (or an OAuth-based authentication protocol) to communicate with the authorization server in all respects, including using software interfaces and methods rather than network interfaces and APIs. The authorization server MUST still issue PATs, AATs, and RPTs and associate authorization data with RPTs, and the resource server MUST still give clients access only when RPTs are associated with sufficient authorization data. Interactions with entities other than the authorization server or resource server MUST be preserved exactly as they would have if either of them were using standardized UMA APIs, unless other extensibility profiles are also in use.

An authorization server using any of the opportunities afforded by this profile MUST declare use of this profile by supplying its identifying URI for one of its "uma_profiles_supported" values in its configuration data (see Section 1.4).

Same-entity communication or a tight integration of entities has the opportunity to make deployments more secure by reducing possible attack vectors. However, if the entities do not use TLS but communicate across a transport layer, it is RECOMMENDED to use an alternate means of transport-layer security, for example, using DTLS in the case of a CoAP-based UMA profile.

Same-entity communication or a tight integration of entities has the potential to compromise privacy by promoting the freer exchange of personal information within a deployment ecosystem. It is RECOMMENDED to account for privacy impacts in each deployment scenario.

5.2. Authorization API Extensibility Profile

This section defines a profile for UMA where the authorization server and client roles either reside in the same system entity or otherwise have a privileged or specialized communications channel between them. Following is a summary:

- Identifying URI: https://docs.kantarainitiative.org/uma/profiles/authz-ext-1.0

- Profile author and contact information: Mark Dobrinic (mdobrinic@cozmanova.com)
- Updates or obsoletes: None; this profile is new.
- Security considerations: See below.
- Privacy considerations: See below.
- Error states: None additional.

Using this profile, the client MAY use means other than the HTTP-based authorization API that is protected by TLS and OAuth (or an OAuth-based authentication protocol) to communicate with the authorization server in all respects, including using software interfaces and methods rather than network interfaces and APIs. The authorization server MUST still issue PATs, AATs, and RPTs and associate authorization data with RPTs, and the resource server MUST still give clients access only when RPTs are associated with sufficient authorization data. Interactions with entities other than the authorization server or client MUST be preserved exactly as they would have if either of them were using standardized UMA APIs, unless other extensibility profiles are also in use.

An authorization server using any of the opportunities afforded by this profile MUST declare use of this profile by supplying its identifying URI for one of its "uma_profiles_supported" values in its configuration data (see Section 1.4).

Same-entity communication or a tight integration of entities has the opportunity to make deployments more secure by reducing possible attack vectors. However, if the entities do not use TLS but communicate across a transport layer, it is RECOMMENDED to use an alternate means of transport-layer security, for example, using DTLS in the case of a CoAP-based UMA profile.

Same-entity communication or a tight integration of entities has the potential to compromise privacy by promoting the freer exchange of personal information within a deployment ecosystem. It is RECOMMENDED to account for privacy impacts in each deployment scenario.

5.3. Resource Interface Extensibility Profile

This section defines a profile for UMA where the resource server and client roles either reside in the same system entity or otherwise have a privileged or specialized communications channel between them. Following is a summary:

- Identifying URI: https://docs.kantarainitiative.org/uma/profiles/rsrc-ext-1.0

- Profile author and contact information: Mark Dobrinic (mdobrinic@cozmanova.com)

- Updates or obsoletes: None; this profile is new.

- Security considerations: See below.

- Privacy considerations: See below.

- Error states: None additional.

Using this profile, the resource server MAY use means other than an HTTP-based resource interface to communicate with the authorization server in all respects, including using software interfaces and methods rather than network interfaces and APIs. The resource server MUST still give clients access only when RPTs are associated with sufficient authorization data. Interactions with entities other than the resource server or client MUST be preserved exactly as they would have if either of them were using standardized UMA APIs, unless other extensibility profiles are also in use.

An authorization server involved in deployments where resource servers and clients are known to be using opportunities afforded by the resource interface extensibility profile MAY declare use of this profile by supplying its identifying URI for one of its "uma_profiles_supported" values in its configuration data (see Section 1.4).

Same-entity communication or a tight integration of entities has the opportunity to make deployments more secure by reducing possible attack vectors. However, if the entities do not use TLS but communicate across a transport layer, it is RECOMMENDED to use an alternate means of transport-layer security, for example, using DTLS in the case of a CoAP-based UMA profile.

Same-entity communication or a tight integration of entities has the potential to compromise privacy by promoting the freer exchange of personal information within a deployment ecosystem. It is RECOMMENDED to account for privacy impacts in each deployment scenario.

6. Specifying Additional Profiles

This specification defines a protocol that has optional features. For implementation interoperability and to serve particular deployment scenarios, including sector-specific ones such as healthcare or e-government, third parties may want to define profiles of UMA that restrict these options.

Further, this specification creates extensibility points for RPT profiles and claim token profiles, and third parties may likewise want to define their own. Different RPT profiles could be used, for example, to change the dividing line between authorization server and resource server responsibilities in controlling access. Different claim token profiles could be used to customize sector-specific or population-specific (such as individual vs. employee) claim types that drive the types of policies resource owners could set.

It is not practical for this specification to standardize all desired profiles. However, to serve overall interoperability goals, this section provides guidelines for third parties that wish to specify UMA-related profiles. In all cases, it is RECOMMENDED that profiles document the following information:

- Specify a URI that uniquely identifies the profile.
- Identify the responsible author and provide postal or electronic contact information.
- Supply references to any previously defined profiles that the profile updates or obsoletes.
- Define any additional or changed error states.
- Specify any conformance and interoperability considerations.
- Supply any additional security and privacy considerations.

6.1. Specifying Profiles of UMA

It is RECOMMENDED that profiles of UMA additionally document the following information:

- Specify the set of interactions between endpoint entities involved in the profile, calling out any restrictions on ordinary UMA-conformant operations and any extension properties used in message formats.

See Section 5 for examples.

6.2. Specifying RPT Profiles

It is RECOMMENDED that RPT profiles additionally document the following information:

- Specify the keyword to be used in HTTP Authorization headers with tokens conforming to this profile.
- Specify the syntax and semantics of the data that the authorization server associates with the token.
- Specify how the token data is associated with, contained within, and/or retrieved by means of, the on-the-wire token string.
- Specify processing rules for token data.
- Identify any restrictions on grant types to be used with the token profile.

See Section 3.3.2 for an example.

6.3. Specifying Claim Token Format Profiles

It is RECOMMENDED that claim token format profiles additionally document the following information:

- Specify any related or additional error_details hints.
- Specify any constraints on the claim token format vs. a standard definition for it in a specification.
- Specify any mutual interpretation details of claim token formats by authorization servers and clients.

7. Compatibility Notes

Implementers should heed the following compatibility notes.

- This specification uses a specific draft of a specification that is not yet final: [OAuth-introspection] (draft 04); the reference will be updated until this "UMA V1.0 candidate" specification is finalized. While every effort will be made to prevent breaking changes to this specification, should they occur, UMA implementations should continue to use the specifically referenced draft version in preference to the final versions, unless using a possible future UMA profile or specification that updates the relevant references.

- In cases where this specification is not prescriptive regarding conformance or interoperability, any common or best practices for implementation will be documented in the [UMA-Impl] over time.

8. Security Considerations

As a profile of OAuth, this specification relies mainly on OAuth security mechanisms as well as transport-level encryption. Thus, implementers are strongly advised to read the security considerations in [OAuth2] (Section 10) and [OAuth-bearer] (Section 5) along with the security considerations of any other OAuth token-defining specifications in use, along with the entire [OAuth-threat] specification, and apply the countermeasures described therein. As well, since this specification builds on [OAuth-resource-reg], implementers should also take into account the security considerations in that specification.

The following sections describe additional security considerations.

8.1. Redirection and Impersonation Threats

This section discusses threats related to UMA's nature as an protocol enabling autonomous (non-resource-owner) requesting parties to gain authorized access to sensitive resources, including through the process of claims-gathering redirection.

Like ordinary OAuth redirection, UMA redirection for the purpose of gathering claims from an end-user requesting party (described in Section 3.4.1.2.2) creates the potential for cross-site request forgery (CSRF) through an open redirect if the authorization server does not force the client to pre-register its redirection endpoint, and server-side artifact tampering if the client does not avail itself of the state parameter. The client SHOULD check that the ticket value returned by an authorization server after a redirect is completed has not been maliciously changed, for example by a man in the browser (MITB), by using the state parameter. (See the [UMA-Impl] for advice on ways to accomplish this.) Sections 4.4.1.8, 4.4.2.5, and 5.3.5 of [OAuth-threat] are apropos for the UMA claims-gathering redirection flow as well.

When a client redirects an end-user requesting party to the requesting party claims endpoint, the client provides no a priori context to the authorization server about which user is appearing at the endpoint, other than implicitly through the permission ticket. Since the authorization server is free to gather any claims it wishes, the effect is to "late-bind" them to the permission ticket and the state string provided by the client, with the effect of enabling the authorization server not to trust client-asserted

claims. This is a desirable result and reflects one reason why the authorization server might choose to demand use of the redirect flow over the push flow. However, the client has the opportunity to switch end-users -- say, enabling malicious end-user Carlos to impersonate the original end-user Bob who approved the minting of of the AAT -- after the redirect completes and before it returns to the RPT endpoint to seek authorization data.

Another issue concerns the exposure of the RPT to an autonomous requesting party, which could maliciously pass the token to an unauthorized party.

To mitigate requesting-party switching and RPT exposure threats, consider the following strategies.

- Require that the Requesting Party (as defined in [UMA-obligations], meaning this party is able to take on legal obligations) legitimately represent the wielder of the bearer token. This solution is based on a legal or contractual approach, and therefore does not reduce the risk from the technical perspective.

- The authorization server, possibly with input from the resource owner, can implement tighter time-to-live strategies around the authorization data in RPTs. This is a classic approach with bearer tokens that helps to limit a malicious party's ability to intercept and use the bearer token. In the same vein, the authorization server could require claims to have a reasonable degree of freshness (which would require a custom claims profile).

- The strongest strategy is to disallow bearer-type RPTs within the UMA profile being deployed, by providing or requiring an RPT profile that requires use of a holder-of-key approach. In this way, the wielder of the token must engage in a live session for proof-of-possession. A less complex version of this strategy is to "elevate trust" in the requesting party by requiring a stronger authentication context, forcing step-up authentication by the requesting party at run time.

8.2. Client Authentication

Along with TLS, UMA requires OAuth, or any OAuth-based authentication protocol, as the security mechanism for its standardized APIs. The UMA resource server acts in the role of an OAuth client at the authorization server's protection API, and the UMA client acts in the role of an OAuth client at the authorization server's authorization API. While it is possible to use any profile of OAuth for this protection, it is RECOMMENDED for the authorization server to use

OpenID Connect, and to use its mechanisms for stronger client authentication at the token endpoint, in order to strengthen the authentication of OAuth clients. Section 16 of [OIDCCore] provides more information on OpenID Connect security considerations.

Clients using the OAuth implicit grant type carry particular vulnerabilities in OAuth, and OpenID Connect doesn't help because of the nature of the implicit grant flow. UMA scenarios are vulnerable as well. For example, an "implicit client" might require the retrieval of AATs more frequently, for each browser on each platform. An attacker can initiate a spear phishing attack on the requesting party with a link to a malicious website, relying on the requesting party to authenticate to the authorization server through an email-based identity provider in order to receive the AAT. The site can impersonate the requesting party using the browser client's client ID in an OpenID Connect implicit request to the UMA authorization server. If the requesting party had previously given consent for an AAT to be issued, this attempt will likely succeed. The subsequently issued AAT and permission ticket for an attempted resource access could potentially be used for RPT retrieval and authorization data issuance.

A number of mitigation strategies are possible.

- The authorization server could penalize or disallow use of the implicit grant flow. This could be done at a variety of levels:
  - Enabling resource owners to define policies controlling the use of such clients
  - Setting system-default policies controlling their use
  - Participating in mutual agreements with other parties that admit only suitably secure client applications to interact with service operators
- The authorization server could support dynamic client registration at the client instance level, such that each instance receives a unique client_id and secret. The client can then use the authorization code flow and have at least some form of client authentication. However, this is easier for a mobile app than for a browser-based HTML app.

8.3. JSON Usage

This specification defines a number of data formats based on [JSON]. As a subset of the JavaScript scripting language, JSON data SHOULD be consumed through a process that does not dynamically execute it as

code, to avoid malicious code execution. One way to achieve this is to use a JavaScript interpreter rather than the built-in JavaScript eval() function.

8.4. Profiles, Binding Obligations, and Trust Establishment

Parties operating and using UMA software entities have opportunities to establish agreements about mutual rights, responsibilities, and common interpretations of UMA constructs for consistent and expected software behavior. These agreements can be used to improve the parties' respective security postures, and written profiles are a key mechanism for conveying and enforcing these agreements. Section 6 discusses profiling. Section 5 discusses profiling for extensibility. [UMA-obligations] discusses the development of binding obligations.

9. Privacy Considerations

The authorization server comes to be in possession of resource set information that may reveal information about the resource owner, which the authorization server's trust relationship with the resource server is assumed to accommodate. However, the client is a less-trusted party -- in fact, entirely untrustworthy until authorization data is associated with its RPT. The more information about a resource set that is registered, the more risk of privacy compromise there is through a less-trusted authorization server.

The primary privacy duty of UMA's design is to the resource owner. However, privacy considerations affect the requesting party as well. This can be seen in the issuance of an AAT, which represents the approval of a requesting party for a client to engage with an authorization server to perform tasks needed for obtaining authorization, possibly including pushing claim tokens.

Parties operating and using UMA software entities have opportunities to establish agreements about mutual rights, responsibilities, and common interpretations of UMA constructs for consistent and expected software behavior. These agreements can be used to improve the parties' respective privacy postures. For information about the additional technical, operational, and legal elements of trust establishment, see [UMA-obligations]. Additional considerations related to Privacy by Design concepts are discussed in [UMA-PbD].

10. IANA Considerations

This document makes the following requests of IANA.

10.1. JSON Web Token Claims Registration

This specification registers the claim defined in Section 3.3.2.

10.1.1. Registry Contents

- Claim name: permissions
- Claim description: Array of objects, each describing a set of scoped, time-limitable entitlements to a resource set
- Change controller: Kantara Initiative User-Managed Access Work Group - wg-uma@kantarainitiative.org
- Specification document: Section 3.3.2 in this document

10.2. Well-Known URI Registration

This specification registers the well-known URI defined in Section 1.4.

10.2.1. Registry Contents

- URI suffix: uma-configuration
- Change controller: Kantara Initiative User-Managed Access Work Group - wg-uma@kantarainitiative.org
- Specification document: Section 1.4 in this document

11. Acknowledgments

The following people made significant text contributions to the specification:

- Paul C. Bryan, ForgeRock US, Inc. (former editor)
- Mark Dobrinic, Cozmanova
- George Fletcher, AOL
- Lukasz Moren, Cloud Identity Ltd
- Christian Scholz, COMlounge GmbH (former editor)
- Mike Schwartz, Gluu
- Jacek Szpot, Newcastle University

Additional contributors to this specification include the Kantara UMA Work Group participants, a list of whom can be found at [UMAnitarians].

12. References

12.1. Normative References

[DynClientReg]
Richer, J., "OAuth 2.0 Core Dynamic Client Registration", December 2014, <http://tools.ietf.org/html/draft-ietf-oauth-dyn-reg>.

[JSON] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", March 2014, <https://tools.ietf.org/html/rfc7159>.

[JWT] Jones, M., "JSON Web Token (JWT)", December 2014, <http://datatracker.ietf.org/doc/draft-ietf-oauth-json-web-token/>.

[OAuth-bearer]
"The OAuth 2.0 Authorization Framework: Bearer Token Usage", October 2012, <http://tools.ietf.org/html/rfc6750>.

[OAuth-introspection]
Richer, J., "OAuth Token Introspection", December 2014, <http://tools.ietf.org/html/draft-ietf-oauth-introspection-04>.

[OAuth-resource-reg]
Hardjono, T., "OAuth 2.0 Resource Set Registration", February 2015, <https://tools.ietf.org/html/draft-hardjono-oauth-resource-reg>.

[OAuth-threat]
Lodderstedt, T., "OAuth 2.0 Threat Model and Security Considerations", January 2013, <http://tools.ietf.org/html/rfc6819>.

[OAuth2] Hardt, D., "The OAuth 2.0 Authorization Framework", October 2012, <http://tools.ietf.org/html/rfc6749>.

[OIDCCore]
Sakimura, N., "OpenID Connect Core 1.0 incorporating errata set 1", November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.

[OIDCDynClientReg]
Sakimura, N., "OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1", November 2014, <http://openid.net/specs/openid-connect-registration-1_0.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3986] Berners-Lee, T., "Uniform Resource Identifier (URI): Generic Syntax", January 2005, <http://www.ietf.org/rfc/rfc3986.txt>.

[RFC6711] Johansson, L., "An IANA Registry for Level of Assurance (LoA) Profiles", August 2012, <https://tools.ietf.org/html/rfc6711>.

[hostmeta]
Hammer-Lahav, E., "Web Host Metadata", October 2011, <http://tools.ietf.org/html/rfc6415>.

12.2. Informative References

[UMA-Impl]
Maler, E., "UMA Implementer's Guide", December 2014, <http://kantarainitiative.org/confluence/display/uma/UMA+Implementer%27s+Guide>.

[UMA-PbD] Maler, E., "Privacy by Design Implications of UMA", December 2013, <http://kantarainitiative.org/confluence/display/uma/Privacy+by+Design+Implications+of+UMA>.

[UMA-casestudies]
Maler, E., "UMA Case Studies", April 2014, <http://kantarainitiative.org/confluence/display/uma/Case+Studies>.

[UMA-obligations]
Maler, E., "Binding Obligations on UMA Participants", January 2013, <http://docs.kantarainitiative.org/uma/draft-uma-trust.html>.

[UMA-usecases]
Maler, E., "UMA Scenarios and Use Cases", October 2010, <http://kantarainitiative.org/confluence/display/uma/UMA+Scenarios+and+Use+Cases>.

[UMAnitarians]
Maler, E., "UMA Participant Roster", December 2014,
<http://kantarainitiative.org/confluence/display/uma/Participant+Roster>.

Authors' Addresses

Thomas Hardjono (editor)
MIT

Email: hardjono@mit.edu

Eve Maler
ForgeRock

Email: eve.maler@forgerock.com

Maciej Machulak
Cloud Identity

Email: maciej.machulak@cloudidentity.co.uk

Domenico Catalano
Oracle

Email: domenico.catalano@oracle.com