Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 28, 2012

T. Hardjono, Ed.
MIT
M. Machulak
Newcastle University
E. Maler
XMLgrrl.com
C. Scholz
COM.lounge GmbH
April 26, 2012

OAuth Dynamic Client Registration Protocol
draft-hardjono-oauth-dynreg-03

Abstract

This specification proposes an OAuth Dynamic Client Registration protocol.

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 28, 2012.

Copyright Notice

Table of Contents

1. Introduction 3
1.1. Notational Conventions 3
1.2. Terminology 3
2. Use Cases 4
3. Requirements 5
3.1. The client needs to be uniquely identifiable by the authorization server 5
3.2. The authorization server must collect metadata about a client for later user interaction 5
3.3. The authorization server must have the option of strongly authenticating the client and its metadata . . . 5
3.4. Dynamic client registration must be possible from both web-server applications and applications with other capabilities and limitations, such as native applications 6
3.5. Transaction integrity must be ensured in large deployments where data propagation can be an issue 6
3.6. Use of standardized discovery protocol 6
3.7. UMA design principles and requirements 7
4. Analysis of Registration Flow Options 7
5. Client Registration with Pushed Metadata 8
5.1. Client Registration Request 9
5.2. Client Registration Response 10
5.3. Error Response 11
6. Client Registration with Pushed URL and Pulled Metadata . . . 12
6.1. Client Registration Request 13
6.2. Client Discovery 13
6.3. Client Registration Response 13
6.4. Error Response 14
7. Native Application Client Registration 15
8. Security Considerations 16
9. Acknowledgments 17
10. Document History 17
11. References 17
11.1. Normative References 17
11.2. Non-Normative References 18
Authors' Addresses 18

1. Introduction

This draft discusses a number of requirements for and approaches to automatic registration of clients with an OAuth authorization server, with special emphasis on the needs of the OAuth-based User-Managed Access protocol [UMA-Core]. This draft also proposes a dynamic registration protocol for an OAuth authorization server.

In some use-case scenarios it is desirable or necessary to allow OAuth clients to obtain authorization from an OAuth authorization server without the two parties having previously interacted. Nevertheless, in order for the authorization server to accurately represent to end-users which client is seeking authorization to access the end-user's resources, a method for automatic and unique registration of clients is needed.

The goal of this proposed registration protocol is for an authorization server to provide a client with a client identifier and optionally a client secret in a dynamic fashion. To accomplish this, the authorization server must first be provided with information about the client, with the client-name being the minimal information provided. In practice, additional information will need to be furnished to the authorization server, such as the client's homepage, icon, description, and so on.

The dynamic registration protocol proposed here is envisioned to be an additional task to be performed by the OAuth authorization server, namely registration of a new client identifier and optional secret and the issuance of this information to the client. This task would occur prior to the point at which the client wields its identifier and secret at the authorization server in order to obtain an access token in normal OAuth fashion.

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 the protocol parameter names and values are case sensitive.

1.2. Terminology

resource server
: A server capable of accepting and responding to protected resource requests.

resource owner
: An entity capable of granting access to a protected resource.

client
: An application obtaining authorization and making protected resource requests.

authorization server
: A server capable of issuing tokens after successfully authenticating the resource owner and obtaining authorization. The authorization server may be the same server as the resource server, or a separate entity.

authorization manager
: An UMA-defined variant of an authorization server that carries out an authorizing user's policies governing access to a protected resource.

end-user authorization endpoint
: The authorization server's HTTP endpoint capable of authenticating the end-user and obtaining authorization.

token endpoint
: The authorization server's HTTP endpoint capable of issuing tokens and refreshing expired tokens.

client identifier
: An unique identifier issued to the client to identify itself to the authorization server. Client identifiers may have a matching secret.

client registration endpoint The authorization server's HTTP endpoint capable of issuing client identifiers and optional client secrets.

2. Use Cases

The UMA protocol involves two instances of OAuth flows. In the first, an end-user introduces a host (essentially an enhanced OAuth resource server) to an authorization manager (an enhanced OAuth authorization server) as a client of it, possibly without that host having obtained client identification information from that server previously. In the second, a requester (an enhanced OAuth client)

approaches a host and authorization manager to get and use an access token in approximately the normal OAuth fashion, again possibly without that client having obtained client identification information from that server previously. Both the host-as-client and the requester-as-client thus may need dynamic client registration in order for the UMA protocol flow to proceed.

The needs for inter-party trust vary in different UMA use cases. In lightweight Web circumstances such as person-to-person calendar sharing, dynamic registration is entirely appropriate. In cases where high-sensitivity information is being protected or where a regulatory environment puts constraints on the building of trust relationships, such as sharing health records with medical professionals or giving access to tax records to outsourced bookkeeping staff, static means of provisioning client identifiers may be imposed.

More information about UMA use cases is available at [UMA-UC].

3. Requirements

Following are proposed requirements for dynamic client registration.

3.1. The client needs to be uniquely identifiable by the authorization server

In order for an authorization server to do proper user-delegated authorization and prevent unauthorized access it must be able to identify clients uniquely. As is done today in OAuth, the client identifier (and optional secret) should thus be issued by the authorization server and not simply accepted as proposed by the client.

3.2. The authorization server must collect metadata about a client for later user interaction

In order for the authorization server to describe a client to an end-user in an authorization step it needs information about the client. This can be the client name at a minimum, but today servers usually request at least a description, a homepage URL, and an icon when doing manual registration.

3.3. The authorization server must have the option of strongly authenticating the client and its metadata

In order to prevent spoofing of clients and enable dynamic building of strong trust relationships, the authorization server should have

the option to verify the provided information. This might be solved using message signature verification; relatively weaker authentication might be achieved in a simpler way by pulling metadata from a trusted client URL.

3.4. Dynamic client registration must be possible from both web-server applications and applications with other capabilities and limitations, such as native applications

In the UMA context, alternative types of applications might serve as both hosts (for example, as a device-based personal data store) and requesters (for example, to subscribe to a calendar or view a photo). Such applications, particularly native applications, may have special limitations, so new solutions to meeting the set of requirements presented here may be needed. We anticipate that each instance of a native application (that is, the specific instance running on each device) that is installed and run by the same user may need the option of getting a unique client identifier. In this case, there are implications around gathering and displaying enough information to ensure that the end-user is delegating authorization to the intended application.

3.5. Transaction integrity must be ensured in large deployments where data propagation can be an issue

When a client sends information to a server endpoint, it might take time for this data to propagate through big server installations that spread across various data centers. Care needs to be taken that subsequent interactions with the user after the registration process, such as an authorization request, show the correct data.

In the UMA context, dynamic registration of a host at an AM is almost certain to take place in the middle of an introduction and authorization process mediated by the end-user; even though the host needs a client identifier from the AM no matter which end-user caused the registration process to take place, the end-user may need to wait for the registration sub-process to finish in order to continue with the overall process. It may be necessary to ensure that the host interacts with the same AM server throughout.

3.6. Use of standardized discovery protocol

Regardless of flow option, the client needs to discover the authorization server’s client registration endpoint. The client MUST use the [RFC5785] and [hostmeta] discovery mechanisms to learn the URI of the client registration endpoint at the authorization server. The authorization server MUST provide a host-meta document that clearly defines the registration end-point at the server.

3.7. UMA design principles and requirements

In addition to general requirements for dynamic client registration, UMA seeks to optimize for the design principles and requirements found in the UMA Requirements document [UMA-Reqs], most particularly:

- DP1: Simple to understand, implement in an interoperable fashion, and deploy on an Internet-wide scale
- DP6: Able to be combined and extended to support a variety of use cases and emerging application functionality
- DP8: Avoid adding crypto requirements beyond what existing web app implementations do today
- DP10: Complexity should be borne by the authorization endpoint vs. other endpoints

4. Analysis of Registration Flow Options

This section analyzes some options for exchanging client metadata for a client identifier and optional secret.

It currently seems impossible to specify a single registration flow that will satisfy all requirements, deployment needs, and client types. This document, therefore, presents as small a variety of options as possible. If it is possible to construct a single unified flow in the ultimate design, all other things being equal this would be preferred.

Client provides metadata on every request
: In this approach, the client passes all necessary metadata such as its name and icon on every request to the authorization server, and the client doesn't wield a client identifier as such. This option makes it more difficult (though not impossible) to meet the first and second requirements since different clients could theoretically represent themselves to an authorization server with the same metadata and the same client could represent itself on subsequent visits with different metadata. Also, today's OAuth protocol requires the use of a client identifier. Because of the UMA simplicity principle we do not recommend this flow option and and have not provided a candidate solution.

Client pushes metadata
: In this approach, the client discovers the registration endpoint of the authorization server and sends its metadata directly to that endpoint in a standard format. The authorization server answers with a client identifier and optional secret in the response. This approach may be necessary in cases where the client is behind a firewall, but strong authentication of the client metadata may be more difficult or costly with this approach than with a "pull" approach, discussed just below. Further, this approach is problematic in the case of applications that can't function as POST-capable web servers. A proposal for "push" is presented in this document.

Client pushes URL, server pulls metadata from it
: In this approach, the client sends only a URL to the authorization server, which then uses that URL to pull metadata about the client in some standard format, returning identification information in the response to the initial request. This approach more easily allows for strong authentication of clients because the metadata can be statically signed. (The message containing the URL could be signed as well.) However, caution should be exercised around the propagation issue if the initial URL push is made to a server different from the one the end-user is interacting with. Further, this approach is problematic in the case of applications that cannot themselves serve as "pull-able" metadata repositories. A proposal for "pull" is presented in this document.

Native-app client collaborates with home-base web app to provide metadata
: An instance of a native application (for example, on a mobile device) may have difficulty directly conveying trustworthy metadata but may also have difficulty providing a trustworthy third-party source from which a server can pull metadata. This document explores one option for meeting the requirements, but does not present a full-fledged proposal.

5. Client Registration with Pushed Metadata

This registration flow works as follows:

1. The client sends its metadata in JSON form to the client registration endpoint. The client MUST send its name, description, and redirection URI and MAY send a URI for its icon. The client MAY sign the metadata as a JSON Token issuer, using

the mechanisms defined in [OAuth-Sig].

2. The authorization server checks the data, verifying the signature as necessary, and returns a client identifier and an optional client secret.

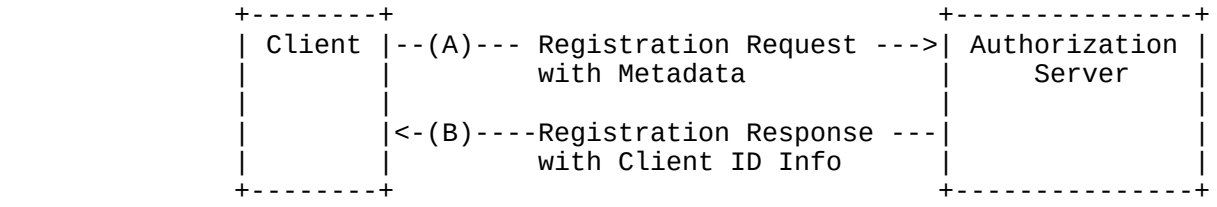

Figure 1: Client Registration Flow with Pushed Metadata

5.1. Client Registration Request

The client sends a JSON formatted document to the client registration endpoint. The client includes the following parameters in the request:

type
: REQUIRED. This parameter must be set to "push".

client_name
: REQUIRED. This field contains a human-readable name of the client.

client_url
: REQUIRED. This field contains the URL of the homepage of the client.

client_description
: REQUIRED. This field contains a text description of the client.

client_icon
: OPTIONAL. This field contains a URL for an icon for the client.

redirect_url
: REQUIRED. This field contains the URL to which the authorization server should send its response.

The client MAY include additional metadata in the request and the authorization server MAY ignore this additional information.

For example, the client might send the following request:

```
POST /register HTTP/1.1
Host: server.example.com
Content-Type: application/json

{
  type: "push",
  client_name: "Online Photo Gallery",
  client_url:  "http://onlinephotogallery.com",
  client_description: "Uploading and also editing capabilities!",
  client_icon: "http://onlinephotogallery.com/icon.png",
  redirect_url: "https://onlinephotogallery.com/client_reg"
}
```

The parameters are included in the entity body of the HTTP request using the "application/json" media type as defined by [JSON]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings.

5.2. Client Registration Response

After receiving and verifying information received from the client, the authorization server issues a client identifier and an optional client secret, and constructs the response by adding the following parameters to the entity body of the HTTP response with a 200 status code (OK):

client_id
: REQUIRED.

client_secret
: OPTIONAL.

issued_at
: OPTIONAL. Specifies the timestamp when the identifier was issued. The timestamp value MUST be a positive integer. The value is expressed in the number of seconds since January 1, 1970 00:00:00 GMT.

expires_in
: OPTIONAL; if supplied, the "issued_at" parameter is REQUIRED. Specifies the valid lifetime, in seconds, of the identifier. The value is represented in base 10 ASCII.

The parameters are included in the entity body of the HTTP response using the "application/json" media type as defined by [JSON]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings.

The authorization server MUST include the HTTP "Cache-Control" response header field with a value of "no-store" in any response containing "client_secret".

For example, the authorization server might return the following response:

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

{
  client_id: "5UO9XcL4TQTa",
  client_secret: "WdRKN3zeTc20"
}
```

5.3. Error Response

If the request for registration is invalid or unauthorized, the authorization server constructs the response by adding the following parameters to the entity body of the HTTP response with a 400 status code (Bad Request) using the "application/json" media type:

- "error" (REQUIRED).
- "error_description" (OPTIONAL). Human-readable text providing additional information, used to assist in the understanding and resolution of the error occurred.
- "error_uri" (OPTIONAL). A URI identifying a human-readable web page with information about the error, used to provide the end-user with additional information about the error.

An example error response (with line breaks for readability):

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

{
"error": "unauthorized_client",
"description": "This client is not on the
  white list of this Authorization Server."
}
```

6. Client Registration with Pushed URL and Pulled Metadata

This registration flow works as follows:

1. The client sends its metadata URI to the client registration endpoint. The client MAY sign the metadata as a JSON Token issuer, using the mechanisms defined in [OAuth-Sig].

2. The authorization server verifies the signature as necessary, and uses the [RFC5785] and [hostmeta] discovery mechanisms on this URI to retrieve the host-meta document describing the client. The host-meta document MUST contain the client name, description, and redirection URI, and MAY contain a URI for the client icon.

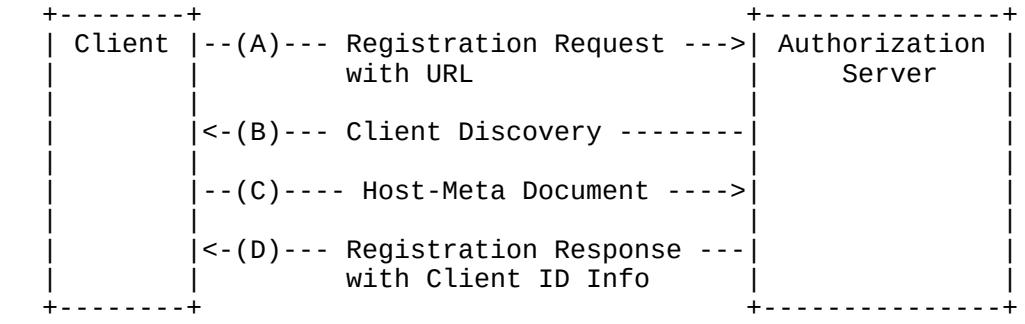

Figure 2: Client Registration Flow with Pushed URL and Pulled Metadata

6.1. Client Registration Request

The client sends a JSON formatted document to the client registration endpoint. The client includes the following parameters in the request:

type
: REQUIRED. This parameter must be set to "pull".

client_url
: REQUIRED. This field contains the URL of the homepage of the client.

The client MUST NOT include other metadata parameters, such as those defined in the pushed-metadata scenario.

For example, the client might send the following request:

```
POST /register HTTP/1.1
Host: server.example.com
Content-Type: application/json

{
  type: "pull",
  url: "http://onlinephotogallery.com"
}
```

The parameters are included in the entity body of the HTTP request using the "application/json" media type as defined by [JSON]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings.

6.2. Client Discovery

The authorization server evaluates this request and MAY perform a [RFC5785] and [hostmeta] discovery mechanism on the provided URL to the host-meta document for the client.

6.3. Client Registration Response

After receiving and verifying information retrieved from the client, the authorization server issues the client identifier and an optional client secret, and constructs the response by adding the following parameters to the entity body of the HTTP response with a 200 status

code (OK):

- "client_id" (REQUIRED)

- "client_secret" (OPTIONAL)

The parameters are included in the entity body of the HTTP response using the "application/json" media type as defined by [JSON]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings.

The authorization server MUST include the HTTP "Cache-Control" response header field with a value of "no-store" in any response containing the "client_secret".

For example the authorization server might return the following response:

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

{
  "client_id":"5UO9XcL4TQTa",
  "client_secret":"WdRKN3zeTc20"
}
```

6.4. Error Response

If the request for registration is invalid or unauthorized, the authorization server constructs the response by adding the following parameters to the entity body of the HTTP response with a 400 status code (Bad Request) using the "application/json" media type:

- "error" (REQUIRED). A single error code.

- "error_description" (OPTIONAL). Human-readable text providing additional information, used to assist in the understanding and resolution of the error occurred.

- "error_uri" (OPTIONAL). A URI identifying a human-readable web page with information about the error, used to provide the end-user with additional information about the error.

An example error response (with line breaks for readability):

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

{
"error": "unauthorized_client",
"description": "This client is not on the
  white list of this Authorization Server."
}
```

If the host-meta discovery was not successful, the authorization server MUST use the error code "hostmeta_error".

An example error response (with line breaks for readability):

```
HTTP/1.1 404 Not Found
Content-Type: application/json
Cache-Control: no-store

{
"error": "hostmeta_error",
"description": "The hostmeta document could
  not be retrieved from the URL."
}
```

7. Native Application Client Registration

For a native application serving as an UMA host, we anticipate that the need for dynamic client registration to introduce this app to an UMA authorization manager may typically happen only once (or very infrequently), likely to a single authorization manager, and registration could usefully take place at the time the app is provisioned onto a device. By contrast, for a native app serving as an UMA requester, it may need to register at multiple authorization managers over time when seeking access tokens, at moments much later than the original provisioning of the app onto the device.

When a native application is provisioned on a device, such as through an app store model, often it has an associated "home base" web server application component with which it registers (outside of any UMA-related or OAuth-related interactions). This pairwise relationship

can be exploited in a number of ways to allow trustable, unique metadata to be conveyed to an OAuth server and for this instance of the app to receive a client identifier and optional secret. We have discussed "device-initiated" and "home base-initiated" pattern options for OAuth dynamic client registration in these circumstances. Device-initiated flows seem more generically applicable (for example, for both UMA host and UMA requester needs). However, a home base-initiated flow may be preferable in case it is necessary to pre-determine a trust level towards an OAuth server. In this case, the home base server could initiate the registration process if and only if there exists a trust relationship between the two parties.

Following is one option for a device-initiated flow:

1. User provisions native app on device and registers with and authenticates to app’s home-base web application.

2. Home base provisions native app with home base-signed metadata.

3. Whenever user tries to use native app to access a protected resource, native app provides home base-provided metadata to server.

4. Server verifies home base signature by pulling public key from home base URL and generates client identifier and secret for native app.

5. Server returns client identifier and secret to native app.

8. Security Considerations

Following are some security considerations:

- No client authentication: The server should treat unsigned pushed client metadata as self-asserted.

- Weak client authentication: The server should treat unsigned pulled client metadata as self-asserted unless the the domain of the client matches the client metadata URL and the URL is well-known and trusted.

- Strong client authentication: The server should treat signed client metadata (pushed or pulled) and a signed metadata URL as self-asserted unless it can verify the signature as being from a trusted source.

9. Acknowledgments

The authors thank the User-Managed Access Work Group participants, particularly the following, for their input to this document:

- Domenico Catalano
- George Fletcher
- Nat Sakimura

10. Document History

[[to be removed by RFC editor before publication as an RFC]]

11. References

11.1. Normative References

[JSON] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", 2006, <http://tools.ietf.org/html/rfc4627>.

[OAuth-Sig] Balfanz, D., "OAuth Signature proposals", 2010, <http://www.ietf.org/mail-archive/web/oauth/current/msg03893.html>.

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

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, April 2010.

[hostmeta] Hammer-Lahav, E., "Web Host Metadata", 2010, <http://xml.resource.org/public/rfc/bibxml3/reference.I-D.draft-hammer-hostmeta-13.xml>.

11.2. Non-Normative References

[UMA-Core] Hardjono, T., "UMA Core Specification", 2012, <http://tools.ietf.org/id/draft-hardjono-oauth-umacore-04.txt>.

[UMA-Reqs] Maler, E., "UMA Requirements", 2010, <http://kantarainitiative.org/confluence/display/uma/UMA+Requirements>.

[UMA-UC] Akram, H., "UMA Explained", 2010, <http://kantarainitiative.org/confluence/display/uma/UMA+Scenarios+and+Use+Cases>.

Authors' Addresses

Thomas Hardjono (editor)
MIT

Phone:
Fax:
Email: hardjono@mit.edu
URI:

Maciej Machulak
Newcastle University

Email: m.p.machulak@ncl.ac.uk
URI: http://ncl.ac.uk/

Eve Maler
XMLgrrl.com

Email: eve@xmlgrrl.com
URI: http://www.xmlgrrl.com

Christian Scholz
COM.lounge GmbH

Phone:
Fax:
Email:
URI: