OAuth 2.0 Protected Resource Metadata

now with WWW-Authenticate

draft-jones-oauth-resource-metadata

Mike Jones & Aaron Parecki
IETF 117, San Francisco
July 25, 2023
Progress Since IETF 116

- In Yokohama, Aaron and Mike were asked to merge their drafts
- Result adds WWW-Authenticate to Resource Metadata draft
  - Resource returns its Resource Identifier and other WWW-Authenticate info
  - Aaron and Mike are both happy with the result
What is this for?

- Enables clients to dynamically learn about and use protected resources they may have no prior knowledge of
- e.g., calendar / email apps that work with many resource servers, and
- authorization servers with no prior relationship to the calendar / email app

User configures client with a particular resource server
Step 0: Trigger WWW-Authenticate Response from Resource (optional)

OPTIONS / HTTP/1.1
Host: calendar.example.com

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer resource="https://calendar.example.com/" scope="read"

Resource returns its resource identifier and other WWW-Authenticate info
Step 1: Protected Resource Metadata Request

GET /.well-known/oauth-protected-resource HTTP/1.1
Host: resource.example.com
Step 1: Protected Resource Metadata Response

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

{
    "resource":
        "https://resource.example.com",
    "authorization_servers":
        ["https://as1.example.com/",
        "https://as2.example.net/"]
    "bearer_methods_supported":
        ["header", "body"],
    "resource_documentation":
        "http://resource.example.com/resource_documentation.html"
}

This is metadata element tells Clients what Authorization Server issuer URLs they can use with this Protected Resource.
Step 1: Client Discovers Resource Metadata

GET /._well-known/oauth-protected-resource HTTP/1.1
Host: calendar.example.com

HTTP/1.1 200 Ok
Content-Type: application/json

{  
  "resource": "https://calendar.example.com/",
  "authorization_servers": ["https://authorization-server.com/"]
  "bearer_methods_supported": ["header", "body"],
  ...
}
Step 2: Client Discovers AS Metadata

GET https://authorization-server.com/.well-known/oauth-authorization-server HTTP/1.1

HTTP/1.1 200 Ok
Content-Type: application/json

{
    "issuer": "https://authorization-server.com/",
    "authorization_endpoint": "https://authorization-server.com/authorize",
    "token_endpoint": "https://authorization-server.com/oauth/token",
    "registration_endpoint": "https://authorization-server.com/oauth/clients",
    "response_types_supported": "code",
    ...
}

Where to open the browser to
Where to get the tokens from
Step 3: Initiate OAuth Flow

Client launches a browser to initiate the OAuth flow...


Normal OAuth flow proceeds, enabling strong MFA and passwordless, as well as SSO

Note: The client_id could be:
- Pre-registered out of band
- Registered dynamically via RFC7591
- Provided as a URI according to a new specification

Note: The redirect_uri could be:
- Custom URL scheme
- localhost:port
- “out-of-band”
Step 4: OAuth Flow is Complete

OAuth flow completes, authorization server redirects to redirect_uri with authorization code, client exchanges code for an access token

POST /oauth/token HTTP/1.1
Host: authorization-server.com
Content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&client_id=***
&code_verifier=XXXX

HTTP/1.1 200 OK
Content-type: application/json

{
  "token_type": "Bearer",
  "expires_in": 86400,
  "access_token": "XXXXXXXX",
  "refresh_token": "YYYYYYYYY",
  "scope": "read"
}

Note: Refresh token is up to the discretion of the AS, but can be used to get a new token when the current one expires if the AS doesn't need the user to re-authenticate themselves.

RFC6749

OAuth flow completes, authorization server redirects to redirect_uri with authorization code, client exchanges code for an access token

POST /oauth/token HTTP/1.1
Host: authorization-server.com
Content-type: application/x-www-form-urlencoded

grant_type=authorization_code
&client_id=***
&code_verifier=XXXX

HTTP/1.1 200 OK
Content-type: application/json

{
  "token_type": "Bearer",
  "expires_in": 86400,
  "access_token": "XXXXXXXX",
  "refresh_token": "YYYYYYYYY",
  "scope": "read"
}

Note: Refresh token is up to the discretion of the AS, but can be used to get a new token when the current one expires if the AS doesn't need the user to re-authenticate themselves.
Step 5: Resource Request

Client uses access token to fetch data

GET / HTTP/1.1
Host: calendar.example.com
Authorization: Bearer Xxxxxxxx

CALENDAR DATA RESPONSE

Note: There are opportunities here to also leverage the new step-up OAuth draft as well, if the RS wants the user to come back with a new or different access token
May WWW-Authenticate ever return a different Resource Identifier hostname than the one to which the request was made?
  o When would that be desirable?
  o What would the security implications of doing so be?
History

- 2016: Protected Resource Metadata draft created in parallel with Authorization Server Metadata draft (which became RFC 8414)
  - AS Metadata was in use at the time and progressed by the WG
  - Protected Resource Metadata was not in use, and was not adopted
- 2022: Protected Resource Metadata reference added to OpenID Connect Federation specification
- Nov 2022: Aaron and Benjamin propose AS Discovery at Resource
- Early 2023: PR Metadata in production use in Italian Federation deployments
- July 2023: Merged drafts, adding WWW-Authenticate Functionality
Next Steps

● Additional reviews by working group members
  ● Particularly of newly added WWW-Authenticate functionality

● Possible working group adoption of draft-jones-oauth-resource-metadata?
  ○ *Because it’s in use*