Logius Standard - Draft

NL GOV Assurance profile for OAuth 2.0

Logius Standard
Draft October 24, 2025

This version:

https://logius-standaarden.github.io/OAuth-NL-profiel/

Latest published version:

https://gitdocumentatie.logius.nl/publicatie/api/oauth/

Latest editor's draft:

https://logius-standaarden.github.io/OAuth-NL-profiel/

Previous version:

https://gitdocumentatie.logius.nl/publicatie/api/oauth/v1.1.0/

Editors:

Frank Terpstra (Geonovum)

Jan van Gelder (Geonovum)

Alexander Green (Logius)

Martin van der Plas (Logius)

Authors:

Jaron Azaria (Logius)

Martin Borgman (Kadaster)

Marc Fleischeuers (Kennisnet)

Alexander Green (Logius)

Peter Haasnoot (Logius)

Heiko Hudig (Logius)

Martin van der Plas (Logius)

Stas Mironov (Logius)

Leon van der Ree (Logius)

Bob te Riele (RvIG)

Remco Schaar (Logius)

Frank Terpstra (Geonovum)

Jan Jaap Zoutendijk (Rijkswaterstaat)

Participate:

GitHub Logius-standaarden/OAuth-NL-profiel

File an issue

Commit history

Pull requests

This document is also available in these non-normative format: PDF

Status of This Document

This is a draft that could be altered, removed or replaced by other documents. It is not a recommendation approved by TO.

Table of Contents

§ Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *NOT RECOMMENDED*, *OPTIONAL*, *RECOMMENDED*, *REQUIRED*, *SHALL*, *SHALL NOT*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [*RFC2119*] [*RFC8174*] when, and only when, they appear in all capitals, as shown here.

§ Introduction

This document profiles the OAuth 2.0 web authorization framework for use in the context of securing web-facing application programming interfaces (APIs), particularly Representational State Transfer (RESTful) APIs. The OAuth 2.0 specifications accommodate a wide range of implementations with varying security and usability considerations, across different types of software clients. The OAuth 2.0 client, protected resource, and authorization server profiles defined in this document serve two purposes:

1. Define a mandatory baseline set of security controls suitable for a wide range of government use cases, while maintaining reasonable ease of implementation and functionality
2. Identify optional, advanced security controls for sensitive use cases where increased risk justifies more stringent controls.

This OAuth profile is intended to be shared broadly, and has been ~~greatly influenced by the [HEART OAuth2 Profile][*HEART.OAuth2*].~~ derived from the [iGov OAuth2 profile] [*iGOV.OAuth2*].

§ Requirements Notation and Conventions

The key words "*MUST*", "*MUST NOT*", "*REQUIRED*", "*SHALL*", "*SHALL NOT*", "*SHOULD*", "*SHOULD NOT*", "*RECOMMENDED*", "*NOT RECOMMENDED*", "*MAY*", and "*OPTIONAL*" in this document are to be interpreted as described in [*rfc2119*] .

All uses of [JSON Web Signature (JWS)] [*rfc7515*] and [JSON Web Encryption (JWE)] [*rfc7516*] data structures in this specification utilize the JWS Compact Serialization or the JWE Compact Serialization; the JWS JSON Serialization and the JWE JSON Serialization are not used.

§ Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Grant", "Authorization Server", "Client", "Client Authentication", "Client Identifier", "Client Secret", "Grant Type", "Protected Resource", "Redirection URI", "Refresh Token", "Resource Owner", "Resource Server", "Response Type", and "Token Endpoint" defined by [OAuth 2.0] [*rfc6749*] , the terms "Claim Name", "Claim Value", and "JSON Web Token (JWT)" defined by [JSON Web Token (JWT)] [*rfc7519*] , and the terms defined by [OpenID Connect Core 1.0] [*OpenID.Core*] .

§ Conformance

This specification defines requirements for the following components:

- OAuth 2.0 clients.
- OAuth 2.0 authorization servers.
- OAuth 2.0 protected resources.

The specification also defines features for interaction between these components:

- Client to authorization server.
- Protected resource to authorization server.

> **iGov-NL : Additional content**
> This profile is based upon the international government assurance profile for OAuth 2.0 (iGov) [*iGOV.OAuth2*] as published by the OpenID Foundation (https://openid.net/foundation/). It should be considered a fork of this profile as the iGov profile is geared more towards the American situtation and in the Netherlands we have to deal with an European Union context.

When an ~~iGov~~ iGov-NL-compliant component is interacting with other ~~iGov~~ iGov-NL-compliant components, in any valid combination, all components *MUST* fully conform to the features and requirements of this specification. All interaction with non-~~iGov~~ iGov-NL components is outside the scope of this specification.

An ~~iGov~~ iGov-NL-compliant OAuth 2.0 authorization server *MUST* support all features as described in this specification. A general-purpose authorization server *MAY* support additional features for use with non-~~iGov~~ iGov-NL clients and protected resources.

An ~~iGov~~ iGov-NL-compliant OAuth 2.0 client *MUST* use all functions as described in this specification. A general-purpose client library *MAY* support additional features for use with non-iGov authorization servers and protected resources.

An ~~iGov~~ iGov-NL-compliant OAuth 2.0 protected resource *MUST* use all functions as described in this specification. A general-purpose protected resource library *MAY* support additional features for use with non-~~iGov~~ iGov-NL authorization servers and clients.
This document is an adaptation of the 'International Government Assurance Profile (iGov) for OAuth 2.0 - Draft 03' (hereinafter: the iGov-profile) of the OpenID Foundation. This does not indicate an endorsement by the OpenID Foundation. In as far as the iGov-profile is incorporated in this document, the OpenID Copyright License applies.

Abstract

The OAuth 2.0 protocol framework defines a mechanism to allow a resource owner to delegate access to a protected resource for a client application.

This specification profiles the OAuth 2.0 protocol framework to increase baseline security, provide greater interoperability, and structure deployments in a manner specifically applicable, but not limited to consumer-to-government deployments in the Netherlands.

Organization / Committee	Version number	Official status	Date
Forum Standaardisatie	-	reported	16-03-2016
Working group	1.0	definitive approved version	15-07-2019
Forum Standaardisatie	1.0	'comply of explain' standard (mandatory open standard)	09-07-2020
KP-API Working group	1.1.0-rc.1	working version / final draft by 'Working Group'	13-05-2024
OAuth-NL Working group	1.1.0-rc.2	proposed version - updated after public consultation	10-07-2024
MIDO programmeringstafel	1.1.0	definitive approved version	03-12-2024
Forum Standaardisatie	-	-	tbd

Dutch government Assurance profile for OAuth 2.0

This profile is based upon the *International Government Assurance Profile (iGov) for OAuth 2.0* as published by the OpenID Foundation. It should be considered a fork of this profile as the iGov profile is geared more towards the American situation and in the Netherlands we have to deal with an European Union context.

We have added the chapter Use cases to illustrate the specific use case the iGov-NL profile is aimed at. Starting with chapter Introduction we follow the structure of the iGov profile. Where we do not use content from iGov we use ~~strikethrough~~ to indicate it is not part of iGov-NL.

> **iGov-NL : Additional content**
>
> Content added for the iGov-NL profile is indicated like this.

The Governance of this standard is described by the API-Standaarden beheermodel in a separate repository and published by Logius (api@logius.nl).

Usecases

There are two use cases: The *client credentials* flow and the *authorization code flow*. In two sections below we will elaborate on these, first we will introduce some common concepts.

Introduction

For the Client credentials flow and Authorization code flow usecases to work properly the following application building blocks need to be in place:

1. the **Resource Server** (usually described as the API)
2. the **Authorization Server**
3. the **Client** (application)

Resource Server

The service is provided by a public/governmental organization. Assumed is the Resource Server is known (by the Authorization Server) prior to actual authorization of the User. A Resource Server is assumed to possess a means for identification of the Resource Server and/or encrypted information, optionally by using a PKI certificate. Furthermore, a Resource Server is assumed to be provided over HTTP using TLS, other protocols are out of scope for this profile.

Authorization Server

An Authorization Server is available, operated by either an independent trusted third-party or the service provider itself. Only a single Authorization Server is in use. The Authorization Server is trusted by the Resource Server. The Authorization Server can identify and authorize the User. In case the User has no direct relationship to the Authorization Server, it can forward the User to an IDP trusted by both the Authorization Server as well as the User. Alternatively, the Authorization Server can otherwise identify and authorize the User and is trusted by that User.

§ Client

The User uses a client, which can be any arbitrary application decided upon by the User. Assumed is that the User trusts this client for interaction with the service. The authorization server has at least low trust in the client when the client is either public or semi-confidential. Assumptions is that the Client is aware of the specifications of the API and authorization is required. The Client is either using a user-agent, typically a browser, or the relevant parts are integrated into the Client application.

Note: Web-applications by default use the system-browser on a User's device as user-agent. Typically a native application (*"mobile app"*) either starts a system browser as user-agent or uses an *in-app* browser. See RFC 8252 for more information on implementation of native applications. Clients can also be 'machine clients' types.

§ Use case: Client credentials flow

The client credentials flow can be used in usecases where there is an Client to Resource server connection where no user information is needed by the resource server. Two examples are:

- An application does a system API call. For instance a ping service to see if an API is available. The user does not need to be logged in for this and there is no relation to the identity of the end user.
- A batch application processes a large number of transactions asynchronously at at later scheduled time. The original access_tokens of the preceding synchronous proces is no longer available. The flow for such a machine to machine interaction is shown in the figure below.

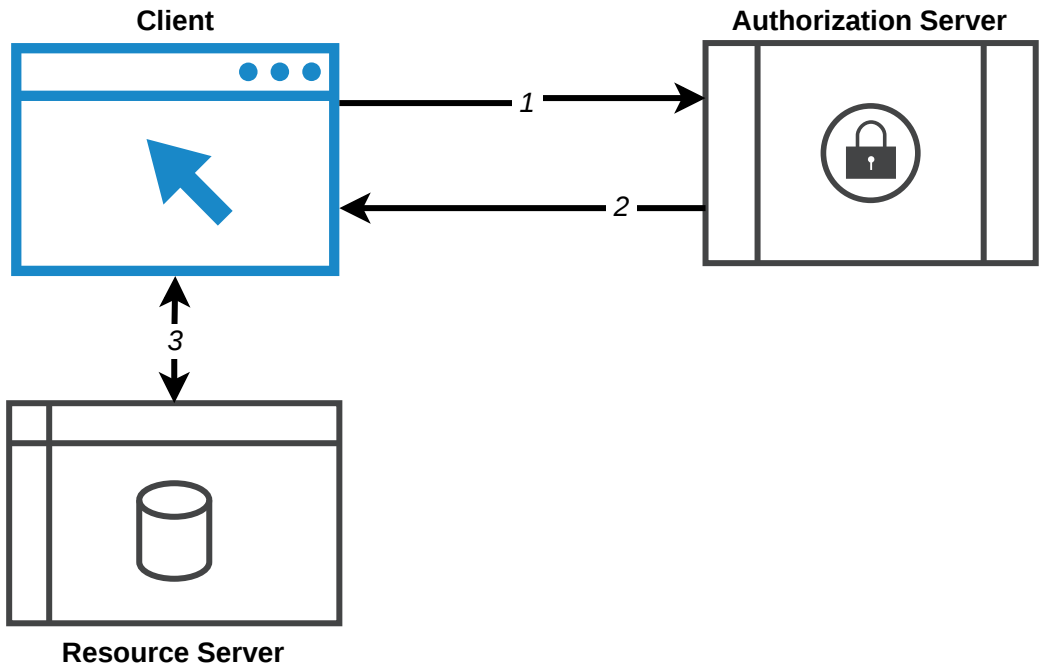

Figure 1 Use case Client credentials flow

§ **Step 1. Client Authentication**

Using the client credentials, the client sends a Authentication Request to the Authorization Server's token Endpoint. It does so using the Client authentication as pre-registered. The Authorization Server receives and validates the Authentication Request.

§ **Step 2. Access Token Response**

The Authorization Server authenticates the client and if valid responds to the client with an Access Token Response. The Authorization server issues an Access Token, specific to the requested authorization. The client receives the Access Token and can use the Access Token to send requests to the Service API.

§ **Step 3. Resource interaction**

The Client can now send (a) request(s) to the Service, on behalf of itself. It does so by sending requests to the Resource Server, along with the Access Token. The Resource Server uses the Access Token for its access control decision. The Resource Server responds based on these decisions to the

Client. The contents and protocol of the Resource Request and Resource Response are out of scope of this profile.

> Direct access clients that are using the client credentials grant type and are not using OpenIDConnect are also allowed to use an X.509 certificate to authenticate with the authorization server's token endpoint. This flow is compatible with OAuth 2.0 due to section 2.3.2 of [*rfc6749*].

§ Use case: Authorization code flow

In this use case a (public/governmental) service is offered via an API. The service will be consumed by the User using a client, that can be any arbitrary, non-trusted application. For provisioning the service, the service provider requires an identifier of the User. The identifier of the User can be either an arbitrary (self-registered) identifier or a formal identifier (citizen number or other restricted, registered ID). Upon service provisioning, the service uses the identifier of the User for access control within the service.

A Client wishes to send a request to an API, on behalf of the User. The API requires to have a trusted identification and *authorization* of the User, before providing the Service. A Client has pre-registered with the Authorization Endpoint and has been assigned a client_id.

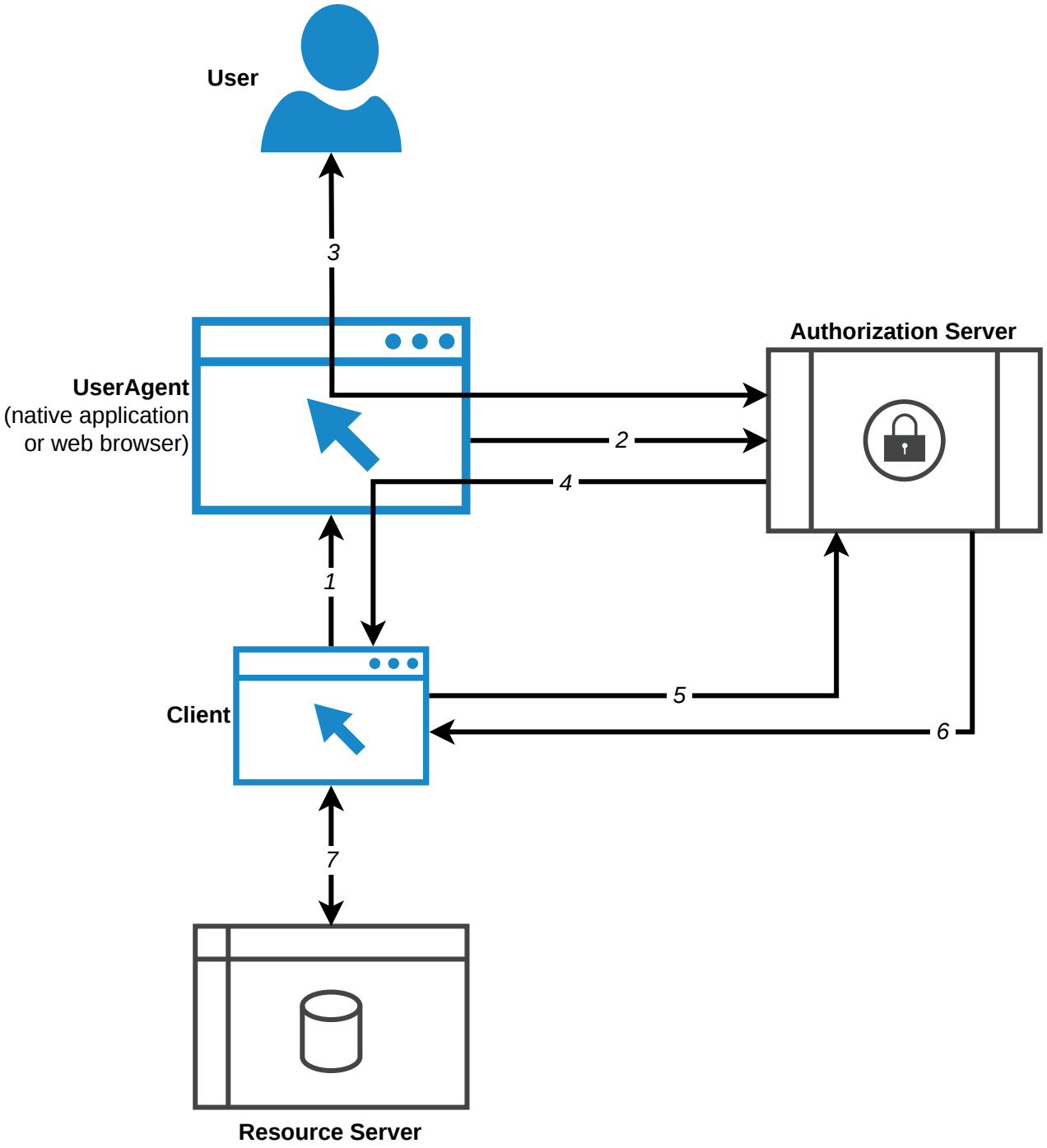

Figure 2 Use case Authorization code flow

The normal flow, that is without any error handling, is described below.

§ Step 1. Authorization initiation

As the client does not yet have a (valid) access token for this Service, it's first step is to obtain one. Therefore it sends an Authorization Request to the Authorization Server's Authorization Endpoint. It does so by redirecting / initiating the user-agent with the Authorization Request to the Authorization Endpoint. The Authorization request holds further details, as specified in this profile.

> NOTE: Extra security consideration
>
> When the Authorization Server supports *OAuth 2.0 Pushed Authorization Requests* (PAR), the client may first use PAR (or is required to use it, see `require_pushed_authorization_requests` in Authorization Server Metadata). The client can initiate the flow by pushing a POST request with the parameters to the `pushed_authorization_request_endpoint`. The Authorization Server responds to the client with a `request_uri` containing a reference. The client will then use this `request_uri` as the redirect.

§ Step 2. Authorization Request

The user-agent sends the Authorization request to the Authorization Endpoint. The Authorization Server receives and validates the request.

§ Step 3. User Authorization and consent

The Authorization Server identifies the Resource Owner (often, but not necessarily, the User) and obtains authorization and consent from the Resource Owner for using the client to access the Service. The method and means for identification, as well as how to obtain authorization and consent from the Resource Owner for the request, are implementation specific and explicitly left out of scope of this profile. Note that if the User and Resource Owner are one and the same, the Autorization Server will have to authenticate the User in order to reliably identify the User as Resource Owner before obtaining the authorization and consent.

§ Step 4. Authorization Grant

Note: applicable to the Authorization Code Flow only. The Authorization Server redirects the user-agent back to the Client, with a Authorization Response. This Authorization Response holds an Authorization Grant and is send to the `redirect_uri` endpoint from the Authorization request.

- **Step 5. Access Token Request**

  The Client receives the Authorization Response from the user-agent. Using the Authorization Grant from the response, the client sends a Token Request to the Authorization Server's token Endpoint. It does so using the Client authentication as pre-registered. The Authorization Server receives and validates the Token Request.

- **Step 6. Access Token Response**

  The Authorization Server responds to the client with an Access Token Response. This response contains an Access Token, specific to the requested authorization. The client receives and validates the Access Token and can use the Access Token to send requests to the Service API.

- **Step 7. Resource interaction**

  The Client can now send (a) request(s) to the Service, on behalf of its User. It does so by sending requests to the Resource Server, along with the Access Token. The Resource Server uses the Access Token for its access control decision and any customization of the service or data for the User, if applicable. The Resource Server responds based on these decisions to the Client. The Client can inform and interact with the User based on the information received from the Resource Server. The contents and protocol of the Resource Request and Resource Response are out of scope of this profile.

Use case: Token exchange

Token exchange is useful when the resource server requires a different token than the one(s) the client originally received during a prior interaction. To obtain this different token, the client can use token exchange [rfc8693]. It can be used for both impersonation and delegation, as specified in that RFC.

The flow for such a machine to machine interaction is shown in the figure below.

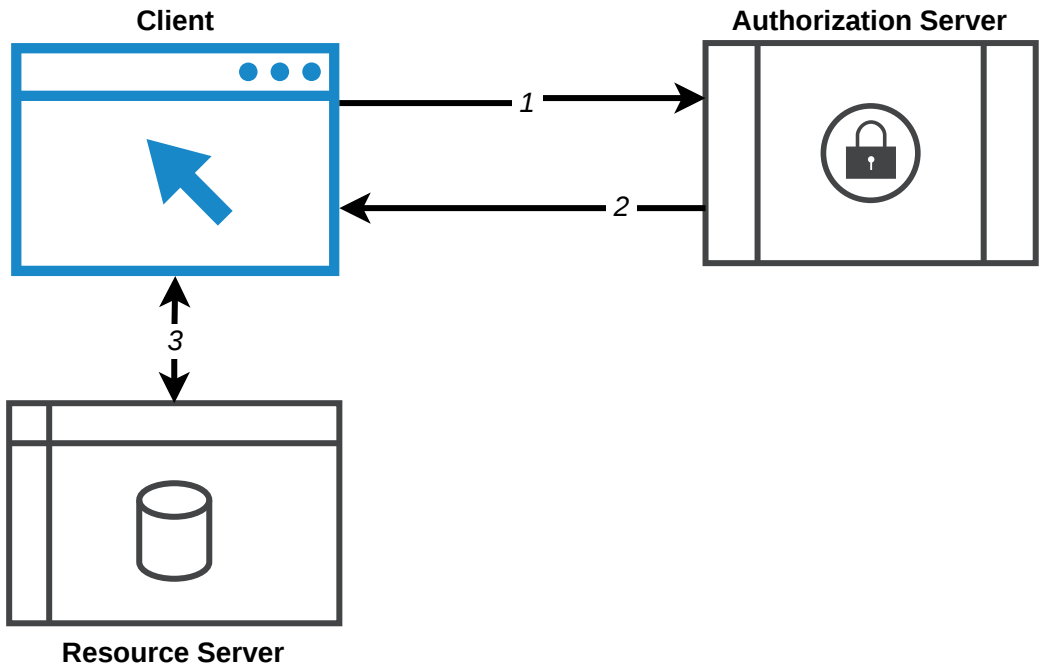

Figure 3 Use case token exchange

Note that the method by which the client received the original token(s)—either directly from an OAuth or SAML token server, or indirectly via other client applications—is not part of the token exchange process itself and, therefore, is not depicted in the diagram.

§ **Step 1: Token Exchange Request**

Using client credentials, the client sends a token exchange request to the Authorization Server's token endpoint. At a minimum, this request includes the subject token and the token exchange grant type. Optionally, the request may also include an actor token, audience, resource, and scopes.

§ **Step 2: Access Token Response**

The Authorization Server authenticates the client and validates the tokens and requested access, according to established policies. If valid, it issues a new token (typically an access token). This new token contains an "act" or "may_act" claim, linking the new token to the original subject token(s) as outlined in RFC 8693, as well as other claims relevant to the token type (e.g., an access token). The response may also include a refresh token, expiration details, and scopes. The client receives the new (access) token and can use it to make requests to the service API.

Step 3: Resource Interaction

The client can now use the new token to send requests to the service. These requests are directed to the Resource Server, which evaluates the access token as part of its access control decisions (e.g., by using the subject in the access token). The Resource Server responds based on these access control decisions. The specifics of the Resource Request and Resource Response are beyond the scope of this document.

1. Client Profiles

1.1 Client Types

The following profile descriptions give patterns of deployment for use in different types of client applications based on the OAuth grant type. Additional grant types, such as assertions, chained tokens, or other mechanisms, are out of scope of this profile and must be covered separately by appropriate profile documents.

1.1.1 Full Client with User Delegation

This client type applies to clients that act on behalf of a particular resource owner and require delegation of that user's authority to access the protected resource. Furthermore, these clients are capable of interacting with a separate web browser application to facilitate the resource owner's interaction with the authentication endpoint of the authorization server.

These clients *MUST* use the authorization code flow of OAuth 2 by sending the resource owner to the authorization endpoint to obtain authorization. The user *MUST* authenticate to the authorization endpoint. The user's web browser is then redirected back to a URI hosted by the client service, from which the client can obtain an authorization code passed as a query parameter. The client then presents that authorization code along with its own credentials (`private_key_jwt`) to the authorization server's token endpoint to obtain an access token.

> **iGov-NL : Additional content**
> In addition to `private_key_jwt`, the client authentication method `tls_client_auth` [*rfc8705*] *MAY* also be used.

These clients *MUST* be associated with a unique public key, as described in Section 2.3.4.

This client type *MAY* request and be issued a refresh token if the security parameters of the access request allow for it.

§ 1.1.2 Native Client with User Delegation

This client type applies to clients that act on behalf of a particular resource owner, such as an app on a mobile platform, and require delegation of that user's authority to access the protected resource. Furthermore, these clients are capable of interacting with a separate web browser application to facilitate the resource owner's interaction with the authentication endpoint of the authorization server. In particular, this client type runs natively on the resource owner's device, often leading to many identical instances of a piece of software operating in different environments and running simultaneously for different end users.

These clients *MUST* use the authorization code flow of OAuth 2 by sending the resource owner to the authorization endpoint to obtain authorization. The user *MUST* authenticate to the authorization endpoint. The user is then redirected back to a URI hosted by the client, from which the client can obtain an authorization code passed as a query parameter. The client then presents that authorization code along to the authorization server's token endpoint to obtain an access token.

Native clients *MUST* either:

- use dynamic client registration to obtain a separate client id for each instance, or
- act as a public client by using a common client id and use PKCE [*rfc7636*] to protect calls to the token endpoint.

Native applications using dynamic registration *SHOULD* generate a unique public and private key pair on the device and register that public key value with the authorization server. Alternatively, an authorization server *MAY* issue a public and private key pair to the client as part of the registration process. In such cases, the authorization server *MUST* discard its copy of the private key. Client credentials *MUST NOT* be shared among instances of client software.

Dynamically registered native applications *MAY* use PKCE.

Native applications not registering a separate public key for each instance are considered Public Clients, and *MUST* use PKCE [*rfc7636*] with the S256 code challenge mechanism. Public Clients do not authenticate with the Token Endpoint in any other way.

§ 1.1.3 Direct Access Client

This client type *MUST NOT* request or be issued a refresh token.

This profile applies to clients that connect directly to protected resources and do not act on behalf of a particular resource owner, such as those clients that facilitate bulk transfers.

These clients use the client credentials flow of OAuth 2 by sending a request to the token endpoint with the client's credentials and obtaining an access token in the response. Since this profile does not involve an authenticated user, this flow is appropriate only for trusted applications, such as those that would traditionally use a developer key. For example, a partner system that performs bulk data transfers between two systems would be considered a direct access client.

§ 1.2 Client Registration

All clients *MUST* register with the authorization server. For client software that may be installed on multiple client instances, such as native applications or web application software, each client instance *MAY* receive a unique client identifier from the authorization server. Clients that share client identifiers are considered public clients.

Client registration *MAY* be completed by either static configuration (out-of-band, through an administrator, etc...) or dynamically.

§ 1.2.1 Redirect URI

Clients using the authorization code grant type *MUST* register their full redirect URIs. The Authorization Server *MUST* validate the redirect URI given by the client at the authorization endpoint using strict string comparison.

A client *MUST* protect the values passed back to its redirect URI by ensuring that the redirect URI is one of the following:

- Hosted on a website with Transport Layer Security (TLS) protection (a Hypertext Transfer Protocol – Secure (HTTPS) URI)
- Hosted on a client-specific non-remote-protocol URI scheme (e.g., `myapp://`)
- Hosted on the local domain of the client (e.g., `http://localhost/`).

Clients *MUST NOT* allow the redirecting to the local domain.

Clients *SHOULD NOT* have multiple redirect URIs on different domains.

Clients *MUST NOT* forward values passed back to their redirect URIs to other arbitrary or user-provided URIs (a practice known as an "open redirector").

§ 1.3 Connection to the Authorization Server

§ 1.3.1 Requests to the Authorization Endpoint

Full clients and browser-embedded clients making a request to the authorization endpoint *MUST* use an unpredictable value for the state parameter with at least 128 bits of entropy. Clients *MUST* validate the value of the `state` parameter upon return to the redirect URI and *MUST* ensure that the state value is securely tied to the user's current session (e.g., by relating the state value to a session identifier issued by the client software to the browser).

Clients *MUST* include their full redirect URI in the authorization request. To prevent open redirection and other injection attacks, the authorization server *MUST* match the entire redirect URI using a direct string comparison against registered values and *MUST* reject requests with an invalid or missing redirect URI.

iGov-NL : Additional content

When the Authorization Server supports *OAuth 2.0 Pushed Authorization Requests* (PAR), the client may first use PAR (or is required to use it, see `require_pushed_authorization_requests` in Authorization Server Metadata). The client can initiate the flow by pushing a POST request with the parameters to the `pushed_authorization_request_endpoint`. The Authorization Server responds to the client with a `request_uri` containing a reference. The client will then use this `request_uri` as the redirect.

Public clients *MUST* apply PKCE, as per RFC7636. As `code_challenge` the S256 method *MUST* be applied. Effectively this means that browser based and native clients *MUST* include a cryptographic random `code_verifier` of at least 128 bits of entropy and the `code_challenge_method` with the value `S256`.

Request fields:

client_id
: Mandatory. *MUST* have the value as obtained during registration.

scope
: Optional.

response_type
: Mandatory. *MUST* have value `code` for the Authorization Code Flow.

redirect_uri
: Mandatory. *MUST* be an absolute HTTPS URL, pre-registered with the Authorization Server.

state
: Mandatory, see above. Do not use the SessionID secure cookie for this.

code_challenge
: In case of using a native app as user-agent mandatory. (Eg. an UUID [*rfc4122*])

code_challenge_method
: In case `code_challenge` is used with a native app, mandatory. *MUST* use the value `S256`.

EXAMPLE 1

The following is a sample response from a web-based client to the end user's browser for the purpose of redirecting the end user to the authorization server's authorization endpoint:

```
HTTP/1.2 302 Found
Cache-Control: no-cache
Connection: close
Content-Type: text/plain; charset=UTF-8
Date: Wed, 07 Jan 2015 20:24:15 GMT
Location: https://idp-p.example.com/authorize?client_id=55f9f559-
2496-49d4-b6c3-
351a586b7484&nonce=cd567ed4d958042f721a7cdca557c30d&response_type=c
ode&scope=openid+email&redirect_uri=https%3A%2F%2Fclient.example.or
g%2Fcb
Status: 302 Found
```

This causes the browser to send the following (non-normative) request to the authorization endpoint:

```
GET /authorize?   client_id=55f9f559-2496-49d4-b6c3-
351a586b7484&nonce=cd567ed4d958042f721a7cdca557c30d&response_type=c
ode&scope=openid+email&redirect_uri=https%3A%2F%2Fclient.example.or
g%2Fcb HTTP/1.1
Host: idp-p.example.com
```

§ 1.3.2 Response from the Authorization Endpoint

iGov-NL : Additional content

Response parameters

code
: Mandatory. *MUST* be a cryptographic random value, using an unpredictable value with at least 128 bits of entropy.

state
: Mandatory. *MUST* be a verbatim copy of the value of the `state` parameter in the Authorization Request.

§ 1.3.3 Requests to the Token Endpoint

Full clients, native clients with dynamically registered keys, and direct access clients as defined above *MUST* authenticate to the authorization server's token endpoint using a JWT assertion as defined by the [JWT Profile for OAuth 2.0 Client Authentication and Authorization Grants] [*rfc7523*] using only the `private_key_jwt` method defined in [OpenID Connect Core] [*OpenID.Core*]. ~~The assertion *MUST* use the claims as follows:~~

> **iGov-NL : Additional content**
>
> When using the JWT assertion, the assertion *MUST* use the claims as follows:

iss
: the client ID of the client creating the token

sub
: the client ID of the client creating the token

aud
: the URL of the authorization server's token endpoint

iat
: the time that the token was created by the client

exp
: the expiration time, after which the token *MUST* be considered invalid

jti
: a unique identifier generated by the client for this authentication. This identifier *MUST* contain at least 128 bits of entropy and *MUST NOT* be re-used by any subsequent authentication token.

> **iGov-NL : Additional content**
> In addition to `private_key_jwt`, the client authentication method `tls_client_auth` [*rfc8705*] *MAY* also be used. Examples of this method can be found in the related documentation of the specific standards.
>
> > Private Key JWT is a method of client authentication where the client creates and signs a JWT using its own private key. This method is described in a combination of RFC 7521 (Assertion Framework) and RFC 7523 (JWT Profile for Client Authentication), and referenced by OpenID Connect and FAPI 2.0 Security Profile.

The JWT assertion *MUST* be signed by the client using the client's private key. See Section 2.3.4 for mechanisms by which the client can make its public key known to the server. The authorization

server *MUST* support the RS256 signature method (the Rivest, Shamir, and Adleman (RSA) signature algorithm with a 256-bit hash) and *MAY* use other asymmetric signature methods listed in the JSON Web Algorithms ([JWA] [*rfc7518*]) specification.

> **iGov-NL : Additional content**
>
> In addition to above signing methods, the Authorization server *SHOULD* support PS256 signing algorithm [*rfc7518*] for the signing of the private_key_jwt.

EXAMPLE 2

The following sample JWT contains the above claims and has been signed using the RS256 JWS algorithm and the client's own private key (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.ew0KICAgImlzcyI6ICI1NWY5ZjU1OS0yN
DQtYjZjMy0zNTFhNTg2Yjc0ODQiLA0KICAgInN1YiI6ICI1NWY5ZjU1OS0yNDk2LTQ5ZDQ
zNTFhNTg2Yjc0ODQiLA0KICAgImF1ZCI6ICJodHRwczovL2lkcC1wLmV4YW1wbGUuY29tL
iwNCiAgICJpYXQiOiAxNDE4Njk4Nzg4LA0KICAgImV4cCI6IDE0MTg2OTg4NDgsDQogICA
gIjE0MTg2OTg3ODgvMTA3YzRkYTUxOTRkZjQ2M2U1MmI1Njg2NWM1YWYzNGU1NTk1Ig0Kf
JQGq3G2OEc2kUCQ8zVj7pqff87Sua5nktLIHj28l5onO5VpsL4sRHIGOvrpo7XO6jgtPWy
Lyo1TWHbtErQEGpmf7nKiNxVCXlGYJXSDJB6shP3OfvdUc24urPJNUGBEDptIgT7-Lhf6B
bNeOPRFDqQoLWqe7UxuI06dKX3SEQRMqcxYSIAfP7CQZ4WLuKXb6oEbaqz6gL4l6p83G7w
THszt-ZjKR38v4F_MnSrx8e0iIqgZwurW0RtetEWvynOCJXk-p166T7qZR45xuCxgOotXY
7GtgspMgOEKj3b_WpCiuNEwQ
```

This is sent in the request to the token endpoint as in the following example:

```
POST /token HTTP/1.1
Content-Type: application/x-www-form-urlencoded
User-Agent: Rack::OAuth2 (1.0.8.7) (2.5.3.2, ruby 2.1.3 (2014-09-19))
Accept: */*
Date: Tue, 16 Dec 2014 02:59:48 GMT
Content-Length: 884
Host: idp-p.example.com

grant_type=authorization_code
&code=sedaFh
&scope=openid+email
&client_id=55f9f559-2496-49d4-b6c3-351a586b7484
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-
&client_assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.ew0KICAgImlzcyI
5ZjU1OS0yNDk2LTQ5ZDQtYjZjMy0zNTFhNTg2Yjc0ODQiLA0KICAgInN1YiI6ICI1NWY5Z
Dk2LTQ5ZDQtYjZjMy0zNTFhNTg2Yjc0ODQiLA0KICAgImF1ZCI6ICJodHRwczovL2lkcC1
wbGUuY29tL3Rva2VuIiwNCiAgICJpYXQiOiAxNDE4Njk4Nzg4LA0KICAgImV4cCI6IDE0M
DgsDQogICAianRpIjogIjE0MTg2OTg3ODgvMTA3YzRkYTUxOTRkZjQ2M2U1MmI1Njg2NWM
1NTk1Ig0KfQ.t-_gX8JQGq3G2OEc2kUCQ8zVj7pqff87Sua5nktLIHj28l5onO5VpsL4sR
7XO6jgtPWy3iLXv3-NLyo1TWHbtErQEGpmf7nKiNxVCXlGYJXSDJB6shP3OfvdUc24urPJ
IgT7-Lhf6BbwQNlMQubNeOPRFDqQoLWqe7UxuI06dKX3SEQRMqcxYSIAfP7CQZ4WLuKXb6
L4l6p83G7wKGDeLETOTHszt-ZjKR38v4F_MnSrx8e0iIqgZwurW0RtetEWvynOCJXk-p16
xuCxgOotXY6O3et4n77GtgspMgOEKj3b_WpCiuNEwQ
```

§ 1.3.4 Client Keys

Clients using the authorization code grant type or direct access clients using the client credentials grant type *MUST* have a public and private key pair for use in authentication to the token endpoint. These clients *MUST* register their public keys in their client registration metadata by either sending the public key directly in the `jwks` field or by registering a `jwks_uri` that *MUST* be reachable by the authorization server. It is *RECOMMENDED* that clients use a `jwks_uri` if possible as this allows for key rotation more easily. This applies to both dynamic and static (out-of-band) client registration.

The `jwks` field or the content available from the `jwks_uri` of a client *MUST* contain a public key in [JSON Web Key Set (JWK Set)] [*rfc7517*] format. The authorization server *MUST* validate the content of the client's registered jwks_uri document and verify that it contains a JWK Set. The following example is of a 2048-bit RSA key:

EXAMPLE 3


```
{
   "keys": [
     {
       "alg": "RS256",
       "e": "AQAB",
       "n": "kAMYD62n_f2rUcR4awJX4uccDt0zcXRssq_mDch5-
ifcShx9aTtTVza23PTn3KaKrsBXwWcfioXR6zQn5eYdZQVGNBfOR4rxF5i7t3hfb4Wk
S50EK1gBYk2lO9NSrQ-xG9QsUsAnN6RHksXqsdOqv-nxjLexDfIJlgbcCN9h6TB-
C66ZXv7PVhl19gIYVifSU7liHkLe0l0fw7jUI6rHLHf4d96_neR1HrNIK_xssr99Xpv
1EM_ubxpktX0T925-qej9fMEpzzQ5HLmcNt1H2_VQ_Ww1JOLn9vRn-
H48FDj7TxlIT74XdTZgTv31w_GRPAOfyxEw_ZUmxhz5Z-gTlQ",
       "kty": "RSA",
       "kid": "oauth-client"
     }
   ]
}
```

iGov-NL : Additional content

In case the Authorization Server, Resource Server and client are not operated under responsibility of the same organisation, each party *MUST* use PKIoverheid certificates with OIN. The PKIoverheid certificate *MUST* be included either as a `x5c` or as `x5u` parameter, as per [*rfc7517*] §4.6 and 4.7. Parties *SHOULD* at least support the inclusion of the certificate as `x5c` parameter, for maximum interoperability. Parties *MAY* agree to use `x5u`, for instance for communication within specific environments.

EXAMPLE 4

For reference, the corresponding public/private key pair for this public key is the following (in JWK format):


```
{
   "alg": "RS256",
   "d": "PjIX4i2NsBQuOVIw74ZDjqthYsoFvaoah9GP-
cPrai5s5VUIlLoadEAdGbBrss_6dR58x_pRlPHWh04vLQsFBuwQNc9SN3O6TAaai9Jg
5TlCi6V0d4O6lUoTYpMR0cxFIU-xFMwII--_OZRgmAxiYiAXQj7TKMKvgSvVO7-9-
YdhMwHoD-
UrJkfnZckMKSs0BoAbjReTski3IV9f1wVJ53_pmr9NBpiZeHYmmG_1QDSbBuY35Zumm
ut4QShF-fey2gSALdp9h9hRk1p1fsTZtH2lwpvmOcjwDkSDv-zO-
4Pt8NuOyqNVPFahROBPlsMVxc_zjPck8ltblalBHPo6AQ",
   "e": "AQAB",
   "n": "kAMYD62n_f2rUcR4awJX4uccDt0zcXRssq_mDch5-
ifcShx9aTtTVza23PTn3KaKrsBXwWcfioXR6zQn5eYdZQVGNBfOR4rxF5i7t3hfb4Wk
S50EK1gBYk2lO9NSrQ-xG9QsUsAnN6RHksXqsdOqv-nxjLexDfIJlgbcCN9h6TB-
C66ZXv7PVhl19gIYVifSU7liHkLe0l0fw7jUI6rHLHf4d96_neR1HrNIK_xssr99Xpv
1EM_ubxpktX0T925-qej9fMEpzzQ5HLmcNt1H2_VQ_Ww1JOLn9vRn-
H48FDj7TxlIT74XdTZgTv31w_GRPAOfyxEw_ZUmxhz5Z-gTlQ",
   "kty": "RSA",
   "kid": "oauth-client"
}
```


Note that the second example contains both the public and private keys, while the first example contains the public key only.

§ 1.4 Connection to the Protected Resource

1.4.1 Requests to the Protected Resource

Clients *SHOULD* send bearer tokens passed in the Authentication header as defined by [*rfc6750*] . Clients *MAY* use the form-parameter ~~or query-parameter~~ method~~s~~ in [*rfc6750*] . Authorized requests *MUST* be made over TLS, and clients *MUST* validate the protected resource server's certificate.

EXAMPLE 5

An example of an OAuth-protected call to the OpenID Connect UserInfo endpoint, sending the token in the Authorization header, follows:

```
GET /userinfo HTTP/1.1
Authorization: Bearer
eyJhbGciOiJSUzI1NiJ9.eyJleHAiOjE0MTg3MDI0MTIsImF1ZCI6WyJjMWJjODRlNC
00N2VlLTRiNjQtYmI1Mi01Y2RhNmM4MWY3ODgiXSwiaXNzIjoiaHR0cHM6XC9cL2lkc
C1wLmV4YW1wbGUuY29tXC8iLCJqdGkiOiJkM2Y3YjQ4Zi1iYzgxLTQwZWMtYTE0MC05
NzRhZjc0YzRkZTMiLCJpYXQiOjE0MTg2OTg4MTJ9.iHMz_tzZ90_b0QZS-
AXtQtvclZ7M4uDAs1WxCFxpgBfBanolW37X8h1ECrUJexbXMD6rrj_uuWEqPD738oWR
o0rOnoKJAgbF1GhXPAYnN5pZRygWSD1a6RcmN85SxUig0H0e7drmdmRkPQgbl2wMhu-
6h2Oqw-ize4dKmykN9UX_2drXrooSxpRZqFVYX8PkCvCCBuFy2O-
HPRov_SwtJMk5qjUWMyn2I4Nu2s-R20aCA-
7T5dunr0iWCkLQnVnaXMfA22RlRiU87nl21zappYb1_EHF9ePyq3Q353cDUY7vje8m2
kKXYTgc_bUAYuW-W3SMSw5UlKaHtSZ6PQICoA
Accept: text/plain, application/json, application/*+json, */*
Host: idp-p.example.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.3 (java 1.5)
```

2. Authorization Server Profile

All servers *MUST* conform to applicable recommendations found in the Security Considerations sections of [*rfc6749*] and those found in the "OAuth Threat Model Document" [*rfc6819*] .

The authorization server *MUST* protect all communications to and from its OAuth endpoints using TLS.

§ 2.1 Connections with clients

§ 2.1.1 Grant types

The authorization server *MUST* support the `authorization_code` , and *MAY* support the `client_credentials` grant types as described in Section 2. The authorization server *MUST* limit each registered client (identified by a client ID) to a single grant type only, since a single piece of software will be functioning at runtime in only one of the modes described in Section 2. Clients that have multiple modes of operation *MUST* have a separate client ID for each mode.

> **iGov-NL : Additional content**
>
> Token exchange grant type [*rfc8693*] *SHOULD* be supported by the authorization server. This is used to translate a third party token (OAuth or SAML). For example, exchanging a DigiD or eHerkenning SAML token for an OAuth token.
>
> When token exchange is not supported, SAML bearer grant [*rfc7522*] *MAY* be used as alternative.

§ 2.1.2 Client authentication

The authorization server *MUST* enforce client authentication as described above for the authorization code and client credentials grant types.

The authorization server *MUST* validate all redirect URIs for authorization code ~~and implicit grant types~~.

§ 2.1.3 Dynamic Registration

Dynamic Registration allows for authorized Clients to on-board programmatically without administrative intervention. This is particularly important in ecosystems with many potential Clients, including Mobile Apps acting as independent Clients. Authorization servers *MUST* support dynamic client registration, and clients *MAY* register using the [Dynamic Client Registration

Protocol] [*rfc7591*] for authorization code grant types.S Authorization servers *MAY* limit the scopes available to dynamically registered clients.

Authorization servers *MAY* protect their Dynamic Registration endpoints by requiring clients to present credentials that the authorization server would recognize as authorized participants. Authorization servers *MAY* accept signed software statements as described in [*rfc7591*] [*rfc7591*] issued to client software developers from a trusted registration entity. The software statement can be used to tie together many instances of the same client software that will be run, dynamically registered, and authorized separately at runtime. The software statement *MUST* include the following client metadata parameters:

redirect_uris
: array of redirect URIs used by the client; subject to the requirements listed in [Section 2.2.1] (#redirect-uri)

grant_types
: grant type used by the client; must be "authorization_code” or "client_credentials”

jwks_uri or jwks
: client's public key in JWK Set format; if jwks_uri is used it *MUST* be reachable by the Authorization Server and point to the client's public key set

client_name
: human-readable name of the client

client_uri
: URL of a web page containing further information about the client

> **iGov-NL : Additional content**
> In this version of iGov-NL we recommend that the Authorization servers *SHOULD* support dynamic client registration. However depending on how the future authentication architecture of the Dutch government develops in regards to OAuth we may revisit this in a future revision. The current requirement fits an architecture where there is a limited number of widely used authorization servers. However if in practice we start seeing a very large number of authorization servers with limited use this requirement can become a recommendation in a future version of this profile. For these authorization servers with limited use we consider mandatory support for dynamic client registration a large burden.

§ 2.1.4 Client Approval

When prompting the end user with an interactive approval page, the authorization server *MUST* indicate to the user:

- Whether the client was dynamically registered, or else statically registered by a trusted administrator, or a public client.
- Whether the client is associated with a software statement, and in which case provide information about the trusted issuer of the software statement.
- What kind of access the client is requesting, including scope, protected resources (if applicable beyond scopes), and access duration.

For example, for native clients a message indicating a new App installation has been registered as a client can help users determine if this is the expected behaviour. This signal helps users protect themselves from potentially rogue clients.

§ 2.1.5 Discovery

The authorization server *MUST* provide an [OpenID Connect service discovery] [*OpenID.Discovery*] endpoint listing the components relevant to the OAuth protocol:

issuer
: *REQUIRED*. The fully qualified issuer URL of the server

authorization_endpoint
: *REQUIRED*. The fully qualified URL of the server's authorization endpoint defined by [OAuth 2.0] [*rfc6749*]

token_endpoint
: *REQUIRED*. The fully qualified URL of the server's token endpoint defined by [OAuth 2.0] [*rfc6749*]

introspection_endpoint
: *OPTIONAL*. The fully qualified URL of the server's introspection endpoint defined by [OAuth Token Introspection] [*rfc7662*]

revocation_endpoint
: *OPTIONAL*. The fully qualified URL of the server's revocation endpoint defined by [OAuth 2.0 Token Revocation] [*rfc7009*]

jwks_uri
: *REQUIRED*. The fully qualified URI of the server's public key in [JWK Set] [*rfc7517*] format

If the authorization server is also an OpenID Connect Provider, it *MUST* provide a discovery endpoint meeting the requirements listed in Section 3.6 of the iGov OpenID Connect profile.

EXAMPLE 6

The following example shows the JSON document found at a discovery endpoint for an authorization server:

> **iGov-NL : Additional content**
> Added `tls_client_auth`

```
{
   "request_parameter_supported": true,
   "registration_endpoint": "https://idp-p.example.com/register",
   "userinfo_signing_alg_values_supported": [
     "HS256", "HS384", "HS512", "RS256", "RS384", "RS512"
   ],
   "token_endpoint": "https://idp-p.example.com/token",
   "request_uri_parameter_supported": false,
   "request_object_encryption_enc_values_supported": [
     "A192CBC-HS384", "A192GCM", "A256CBC+HS512",
     "A128CBC+HS256", "A256CBC-HS512",
     "A128CBC-HS256", "A128GCM", "A256GCM"
   ],
   "token_endpoint_auth_methods_supported": [
     "private_key_jwt",
   ],
   "jwks_uri": "https://idp-p.example.com/jwk",
   "authorization_endpoint": "https://idp-p.example.com/authorize",
   "require_request_uri_registration": false,
   "introspection_endpoint": "https://idp-p.example.com/introspect",
   "request_object_encryption_alg_values_supported": [
     "RSA-OAEP", ?RSA1_5", "RSA-OAEP-256"
   ],
   "service_documentation": "https://idp-p.example.com/about",
   "response_types_supported": [
     "code", "token"
   ],
   "token_endpoint_auth_signing_alg_values_supported": [
     "HS256", "HS384", "HS512", "RS256", "RS384", "RS512"
   ],
   "revocation_endpoint": "https://idp-p.example.com/revoke",
   "request_object_signing_alg_values_supported": [
     "HS256", "HS384", "HS512", "RS256", "RS384", "RS512"
   ],
   "grant_types_supported": [
     "authorization_code",
     "client_credentials"
   ],
```

```
  "scopes_supported": [
    "profile", "openid", "email", "address", "phone", "offline_access
  ],
  "op_tos_uri": "https://idp-p.example.com/about",
  "issuer": "https://idp-p.example.com/",
  "op_policy_uri": "https://idp-p.example.com/about"
  "tls_client_certificate_bound_access_tokens": "true"
  "dpop_signing_alg_values_supported": ["PS256", "ES256"]
}
```

Clients and protected resources *SHOULD* cache this discovery information. It is *RECOMMENDED* that servers provide cache information through HTTP headers and make the cache valid for at least one week.

The server *MUST* provide its public key in JWK Set format. The key *MUST* contain the following fields:

kid
: The key ID of the key pair used to sign this token

kty
: The key type

alg
: The default algorithm used for this key

EXAMPLE 7

The following is an example of a 2048-bit RSA public key:

```
{
  "keys": [
    {
      "alg": "RS256",
      "e": "AQAB",
      "n": "o80vbR0ZfMhjZWfqwPUGNkcIeUcweFyzB2S2T-
hje83IOVct8gVg9FxvHPK1ReEW3-p7-A8GNcLAuFP_8jPhiL6LyJC3F10aV9KPQFF-
w6Eq6VtpEgYSfzvFegNiPtpMWd7C43EDwjQ-GrXMVCLrBYxZC-
P1ShyxVBOzeR_5MTC0JGiDTecr_2YT6o_3aE2SIJu4iNPgGh9MnyxdBo0Uf0TmrqEIa
bquXA1-V8iUihwfI8qjf3EujkYi7gXXelIo4_gipQYNjr4DBNlE0__RI0kDU-
27mb6esswnP2WgHZQPsk779fTcNDBIcYgyLujlcUATEqfCaPDNp00J6AbY6w",
      "kty": "RSA",
      "kid": "rsa1"
    }
  ]
}
```

Clients and protected resources *SHOULD* cache this key. It is *RECOMMENDED* that servers provide cache information through HTTP headers and make the cache valid for at least one week.

iGov-NL : Additional content

iGov requires that the authorization server provides an OpenIDConnect service discovery endpoint. Recently OAuth 2.0 Authorization Server Metadata [*rfc8414*] has been finalized, this provide the same functionality in a more generic way and could replace this requirement in a future version of the iGov-NL profile.

§ 2.1.6 Revocation

Token revocation allows a client to signal to an authorization server that a given token will no longer be used.

An authorization server *MUST* revoke the token if the client requesting the revocation is the client to which the token was issued, the client has permission to revoke tokens, and the token is revocable.

A client *MUST* immediately discard the token and not use it again after revoking it.

§ 2.1.7 PKCE

An authorization server *MUST* support the Proof Key for Code Exchange (PKCE [*rfc7636*]) extension to the authorization code flow, including support for the S256 code challenge method. The authorization server *MUST NOT* allow an ~~iGov~~ iGov-NL client to use the plain code challenge method.

§ 2.1.8 Redirect URIs

The authorization server *MUST* compare a client's registered redirect URIs with the redirect URI presented during an authorization request using an exact string match.

§ 2.1.9 RefreshTokens

Authorization Servers *MAY* issue refresh tokens to clients under the following context:

Clients *MUST* be registered with the Authorization Server.

Clients *MUST* present a valid client_id. Confidential clients *MUST* present a signed client_assertion with their associated keypair.

Clients using the Direct Credentials method *MUST NOT* be issued refresh_tokens. These clients *MUST* present their client credentials with a new access_token request and the desired scope.

> **iGov-NL : Additional content**
> Refresh tokens for public clients must either be sender-constrained or one-time use. From *The OAuth 2.1 Authorization Framework: Refresh Token Grant*

§ 2.1.10 Token Response

iGov-NL : Additional content

The Token Response has the following contents:

access_token
: Mandatory. Structured access token a.k.a. a JWT Bearer token. The JWT *MUST* be signed.

token_type
: Mandatory. The type for a JWT Bearer token is `Bearer`, as per [*rfc6750*]

refresh_token
: Under this profile, refresh tokens are supported.

expires_in
: Optional. Lifetime of the access token, in seconds.

scope
: Optional. Scope(s) of the access (token) granted, multiple scopes are separated by whitespace. The scope *MAY* be omitted if it is identical to the scope requested.

For best practices on token lifetime see section Token Lifetimes.

§ 2.2 Connections between authorization servers and protected resources

Unlike the core OAuth protocol, the ~~iGov~~ iGov-NL profile defines interoperability requirements between authorization servers and resource servers.

[/]: Dit is §3.3 geworden in de nieuwe versie van iGov. De tekst is ook (qua zinsbouw) aangepast.

§ 2.2.1 JSON Web Tokens (JWT)

In order to facilitate interoperability with multiple protected resources, all ~~iGov~~ iGov-NL-compliant authorization servers *MUST* issue cryptographically signed tokens in the JSON Web Token (JWT) format as defined in *JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens* . The information carried in the JWT is intended to allow a protected resource to quickly test the integrity of the token without additional network calls, and to allow the protected resource to determine which authorization server issued the token. The protected resource *MAY* use the

authorization server token introspection service, which is in turn used for conveying service to retrieve additional security information about the token.

The server *MUST* issue tokens as JWTs with, at minimum, the following claims:

iss
: The issuer URL of the server that issued the token

client_id
: The client id of the client to whom this token was issued

exp
: The expiration time (integer number of seconds since from 1970-01-01T00:00:00Z UTC), after which the token *MUST* be considered invalid

jti
: A unique JWT Token ID value with at least 128 bits of entropy. This value *MUST NOT* be re-used in another token. Clients *MUST* check for reuse of jti values and reject all tokens issued with duplicate jti values.

sub
: The identifier of the end-user that authorized this client, or the client id of a client acting on its own behalf (such as a bulk transfer). Since this information could potentially leak private user information, it should be used only when needed. End-user identifiers *SHOULD* be pairwise anonymous identifiers unless the audiance requires otherwise.

aud
: The audience of the token, an array containing the identifier(s) of protected resource(s) for which the token is valid, if this information is known. The aud claim may contain multiple values if the token is valid for multiple protected resources. Note that at runtime, the authorization server may not know the identifiers of all possible protected resources at which a token may be used.

iat
: The "iat" (issued at) claim identifies the time at which the JWT was issued. This claim can be used to determine the age of the JWT. Its value *MUST* be a number containing a NumericDate value.

EXAMPLE 8

The following sample claim set illustrates the use of the required claims for an access token as defined in this profile; additional claims *MAY* be included in the claim set:

```
{
   "exp": 1418702388,
   "client_id": "55f9f559-2496-49d4-b6c3-351a586b7484",
   "iss": "https://idp-p.example.com/",
   "sub" : "93ff28e3-3982-c34b-f2a4-98bb3d42b277",
   "aud": "api.example.com"
   "jti": "2402f87c-b6ce-45c4-95b0-7a3f2904997f",
   "iat": 1418698788,
   "acr": "myACR"
}
```

The access tokens *MUST* be signed with [JWS] [*RFC7515*] . If private_key_jwt is used, the authorization server *MUST* support the RS256 signature method for tokens and *MAY* use other asymmetric signing methods as defined in the [IANA JSON Web Signatures and Encryption Algorithms registry] [*JWS.JWE.Algs*] . The JWS header *MUST* contain the following fields:

iGov-NL : Additional content

In addition to above signing methods, the Authorization server *SHOULD* support PS256 signing algorithm [*rfc7518*] for the signing of the JWT Bearer Tokens.

kid

The key ID of the key pair used to sign this token

EXAMPLE 9

This example access token has been signed with the server's private key using RS256:

```
eyJhbGciOiJSUzI1NiJ9.ew0KICAgImV4cCI6IDE0MTg3MDIzODgsDQogICAiYXpwIjo
gIjU1ZjlmNTU5LTI0OTYtNDlkNC1iNmMzLTM1MWE1ODZiNzQ4NCIsDQogICAiaXNzIjo
gImh0dHBzOi8vaWRwLXAuZXhhbXBsZS5jb20vIiwNCiAgICJqdGkiOiAiMjQwMmY4N2M
tYjZjZS00NWM0LTk1YjAtN2EzZjI5MDQ5OTdmIiwNCiAgICJpYXQiOiAxNDE4Njk4Nzg
4LA0KICAgImtpZCI6ICJyc2ExIg0KfQ.iB6Ix8Xeg-L-nMStgE1X75w7zgXabzw7znWU
ECOsXpHfnYYqb-CET9Ah5IQyXIDZ20qEyN98UydgsTpiO1YJDDcZV4f4DgY0ZdG3yBW3
XqwUQwbgF7Gwza9Z4AdhjHjzQx-lChXAyfL1xz0SBDkVbJdDjtXbvaSIyfF7ueWF3M1C
M70-GXuRY4iucKbuytz9e7eW4Egkk4Aagl3iTk9-l5V-tvL6dYu8IlR93GKsaKE8bng0
EZ04xcnq8s4V5Yykuc_NARBJENiKTJM8w3wh7xWP2gvMp39Y0XnuCOLyIx-J1ttX83xm
pXDaLyyY-4HT9XHT9V73fKF8rLWJu9grrA
```

Refresh tokens *SHOULD* be signed with [JWS] [*rfc7515*] using the same private key and contain the same set of claims as the access tokens.

The authorization server *MAY* encrypt access tokens and refresh tokens using [JWE] [*rfc7516*] . Encrypted access tokens *MUST* be encrypted using the public key of the protected resource. Encrypted refresh tokens *MUST* be encrypted using the authorization server's public key.

> **iGov-NL : Additional content**
>
> How to select or obtain the key to be used for encryption of an access token is out of scope of this profile. An early draft of "Resource Indicators for OAuth 2.0" exist and COULD be used. This draft describes usage of the resource parameter to indicate the applicable resource server.
>
> In case the Authorization Server and Resource Server are not operated under responsibility of the same organization, the bearer token *MUST* be signed with the use of a PKIoverheid certificates with OIN.
>
> If the bearer token is also encrypted the bearer token *MUST* be encrypted with the use of a PKIoverheid certificates with OIN.

§ 2.2.2 Introspection

Token introspection allows a protected resource to query the authorization server for metadata about a token. The protected resource makes a request like the following to the token introspection endpoint:

EXAMPLE 10

```
POST /introspect HTTP/1.1
User-Agent: Faraday v0.9.0
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip;q=1.0,deflate;q=0.6,identity;q=0.3
Accept: */*
Connection: close
Host: as-va.example.com
Content-Length: 1412



client_assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3M
iOiJhMmMzNjkxOS0wMWZmLTQ4MTAtYTgyOS00MDBmYWQzNTczNTEiLCJzdWIi
OiJhMmMzNjkxOS0wMWZmLTQ4MTAtYTgyOS00MDBmYWQzNTczNTEiLCJhdWQiO
iJodHRwczovL2FzLXZhLmV4YW1wbGUuY29tL3Rva2VuIiwiaWF0IjoxNDE4Nj
k4ODE0LCJleHAiOjE0MTg2OTg4NzQsImp0aSI6IjE0MTg2OTg4MTQvZmNmNDQ
2OGY2MDVjNjE1NjliOWYyNGY5ODJlMTZhZWY2OTU4In0.md7mFdNBaGhiJfE_
pFkAAWA5S-JBvDw9Dk7pOOJEWcL08JGgDFoi9UDbg3sHeA5DrrCYGC_zw7fCG
c9ovpfMB7W6YN53hGU19LtzzFN3tv9FNRq4KIzhK15pns9jckKtui3HZ25L_B
-BnxHe7xNo3kA1M-p51uYYIM0hw1SRi2pfwBKG5O8WntybLjuJ0R3X97zvqHn
2Q7xdVyKlInyNPA8gIZK0HVssXxHOI6yRrAqvdMn_sneDTWPrqVpaR_c7rt8D
dd7drf_nTD1QxESVhYqKTax5Qfd-aq8gZz8gJCzS0yyfQh6DmdhmwgrSCCRC6
BUQkeFNvjMVEYHQ9fr0NA
&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-
&client_id=a2c36919-01ff-4810-a829-400fad357351
&token=eyJhbGciOiJSUzI1NiJ9.eyJleHAiOjE0MTg3MDI0MTQsImF1ZCI6W
yJlNzFmYjcyYS05NzRmLTQwMDEtYmNiNy1lNjdjMmJjMDAzN2YiXSwiaXNzIj
oiaHR0cHM6XC9cL2FzLXZhLmV4YW1wbGUuY29tXC8iLCJqdGkiOiIyMWIxNTk
2ZC04NWQzLTQzN2MtYWQ4My1iM2YyY2UyNDcyNDQiLCJpYXQiOjE0MTg2OTg4
MTR9.FXDtEzDLbTHzFNroW7w27RLk5m0wprFfFH7h4bdFw5fR3pwiqejKmdfA
bJvN3_yfAokBv06we5RARJUbdjmFFfRRW23cMbpGQCIk7Nq4L012X_1J4IewO
QXXMLTyWQQ_BcBMjcW3MtPrY1AoOcfBOJPx1k2jwRkYtyVTLWlff6S5gK-ciY
f3b0bAdjoQEHd_IvssIPH3xuBJkmtkrTlfWR0Q0pdpeyVePkMSI28XZvDaGnx
A4j7QI5loZYeyzGR9h70xQLVzqwwl1P0-F_0JaDFMJFO1yl4IexfpoZZsB3Hh
F2vFdL6D_lLeHRy-H2g2OzF59eMIsM_Ccs4G47862w
```

The client assertion parameter is structured as described in Section 2.3.3 .

The server responds to an introspection request with a JSON object representing the token containing the following fields as defined in the token introspection specification:

active
: Boolean value indicating whether or not this token is currently active at this authorization server. Tokens that have been revoked, have expired, or were not issued by this authorization server are considered non-active.

scope
: Space-separated list of OAuth 2.0 scope values represented as a single string.

exp
: Timestamp of when this token expires (integer number of seconds since from 1970-01-01T00:00:00Z UTC)

sub
: An opaque string that uniquely identifies the user who authorized this token at this authorization server (if applicable). This string *MAY* be diversified per client.

client_id
: An opaque string that uniquely identifies the OAuth 2.0 client that requested this token

EXAMPLE 11

The following example is a response from the introspection endpoint:

```
HTTP/1.1 200 OK
Date: Tue, 16 Dec 2014 03:00:14 GMT
Access-Control-Allow-Origin: *
Content-Type: application/json;charset=ISO-8859-1
Content-Language: en-US
Content-Length: 266
Connection: close
```

```
{
   "active": true,
   "scope": "file search visa",
   "exp": 1418702414,
   "sub": "{sub\u003d6WZQPpnQxV, iss\u003dhttps://idp-p.example.com/}"
   "client_id": "e71fb72a-974f-4001-bcb7-e67c2bc0037f",
   "token_type": "Bearer"
}
```

The authorization server *MUST* require authentication for both the revocation and introspection endpoints as described in Section 2.3.2 . Protected resources calling the introspection endpoint *MUST* use credentials distinct from any other OAuth client registered at the server.

A protected resource *MAY* cache the response from the introspection endpoint for a period of time no greater than half the lifetime of the token. A protected resource *MUST NOT* accept a token that is not active according to the response from the introspection endpoint.

§ 2.3 Response to Authorization Requests

The following data will be sent as an Authorization Response to the Authorization Code Flow as described above. The authentication response is sent via HTTP redirect to the redirect URI specified in the request.

The following fields *MUST* be included in the response:

state
: *REQUIRED*. The value of the state parameter passed in in the authentication request. This value *MUST* match exactly.

code
: *REQUIRED*. The authorization code, a random string issued by the IdP to be used in the request to the token endpoint.

PKCE parameters *MUST* be associated with the "code" as per Section 4.4 of [Proof Key for Code Exchange by OAuth Public Clients (PKCE)] [*rfc7636*]

> EXAMPLE 12
> The following is an example response:
>
> ```
> https://client.example.org/cb?state=2ca3359dfbfd0&code=gOIFJ1hV6Rb1sxUd
> ```

§ 2.4 Token Lifetimes

This profile provides *RECOMMENDED* lifetimes for different types of tokens issued to different types of clients. Specific applications *MAY* issue tokens with different lifetimes. Any active token *MAY* be revoked at any time.

For clients using the authorization code grant type, access tokens *SHOULD* have a valid lifetime no greater than one hour, and refresh tokens (if issued) *SHOULD* have a valid lifetime no greater than twenty-four hours.

For public clients access tokens *SHOULD* have a valid lifetime no greater than fifteen minutes.

For clients using the client credentials grant type, access tokens *SHOULD* have a valid lifetime no greater than six hours.

§ 2.5 Scopes

Scopes define individual pieces of authority that can be requested by clients, granted by resource owners, and enforced by protected resources. Specific scope values will be highly dependent on the specific types of resources being protected in a given interface. OpenID Connect, for example, defines scope values to enable access to different attributes of user profiles.

Authorization servers *SHOULD* define and document default scope values that will be used if an authorization request does not specify a requested set of scopes.

To facilitate general use across a wide variety of protected resources, authorization servers *SHOULD* allow for the use of arbitrary scope values at runtime, such as allowing clients or protected resources to use arbitrary scope strings upon registration. Authorization servers *MAY* restrict certain scopes from use by dynamically registered systems or public clients.

§ 2.5.1 Claims for Authorization Outside of Delegation Scenarios

> **iGov-NL : Additional content**
>
> If there is a need to include resource owner memberships in roles and groups that are relevant to the resource being accessed, entitlements assigned to the resource owner for the targeted resource that the authorization server knows about. The authorization server *SHOULD* include such attributes as claims in a JWT access token as defined in section 2.2.3.1 of [*rfc9068*].
>
> In cases where the default scopes provided by Authorization Server are not descriptive enough, one can make use of the `authorization_details` claim (see 4.3 Rich Authorization Requests) which provide extra details and thus greater flexibility for the Resource Server to handle the request.

§ 3. Protected Resource Profile

§ 3.1 Protecting Resources

Protected Resources grant access to clients if they present a valid `access_token` with the appropriate `scope`. Resource servers trust the authorization server to authenticate the end user and client appropriately for the importance, risk, and value level of the protected resource scope.

Protected resources that require a higher end-user authentication trust level to access certain resources *MUST* associate those resources with a unique scope.

Clients wishing access to these higher level resources *MUST* include the higher level scope in their authorization request to the authorization server.

Authorization servers *MUST* authenticate the end-user with the appropriate trust level before providing an `authorization_code` or associated `access_token` to the client.

Authorization servers *MUST* only grant access to higher level scope resources to clients that have permission to request these scope levels. Client authorization requests containing scopes that are outside their permission *MUST* be rejected.

Authorization servers *MAY* set the expiry time (`exp`) of access_tokens associated with higher level resources to be shorter than access_tokens for less sensitive resources.

Authorization servers *MAY* allow a `refresh_token` issued at a higher level to be used to obtain an access_token for a lower level resource scope with an extended expiry time. The client *MUST* request both the higher level scope and lower level scope in the original authorization request. This allows clients to continue accessing lower level resources after the higher level resource access has expired -- without requiring an additional user authentication/authorization.

EXAMPLE 13

A resource server has resources classified as "public" and "sensitive". "Sensitive" resources require the user to perform a two-factor authentication, and those access grants are short-lived: 15 minutes. For a client to obtain access to both "public" and "sensitive" resources, it makes an authorization request to the authorization server with `scope=public+sensitive`. The authorization server authenticates the end-user as required to meet the required trust level (two-factor authentication or some approved equivalent) and issues an `access_token` for the "public" and "sensitive" scopes with an expiry time of 15mins, and a `refresh_token` for the "public" scope with an expiry time of 24 hrs. The client can access both "public" and "sensitive" resources for 15mins with the access_token. When the access_token expires, the client will NOT be able to access "public" or "sensitive" resources any longer as the access_token has expired, and must obtain a new access_token. The client makes a access grant request (as described in [OAuth 2.0] [*rfc6749*] section 6) with the refresh_token, and the reduced scope of just "public". The token endpoint validates the refresh_token, and issues a new access_token for just the "public" scopewith an expiry time set to 24hrs. An access grant request for a new access_token with the "sensitive" scope would be rejected, and require the client to get the end-user to re-authenticate/authorize the "sensitive" scope request.

In this manner, protected resources and authorization servers work together to meet risk tolerance levels for sensitive resources and end-user authentication.

§ 3.2 Connections with Clients

A protected resource *MUST* accept bearer tokens passed in the authorization header as described in [*rfc6750*] . A protected resource *MAY* also accept bearer tokens passed in the form parameter ~~or query parameter~~ method~~s~~.

iGov-NL : Additional content

A Protected Resource under this profile *MUST NOT* accept access tokens passed using the query parameter method.

A Protected Resource under this profile *SHOULD* verify if the client is the Authorized party (azp) when client authentication is used. See section Advanced OAuth Security Options as well.

Protected resources *MUST* define and document which scopes are required for access to the resource.

§ 3.3 Connections with Authorization Servers

Protected resources *MUST* interpret access tokens using either JWT, token introspection, or a combination of the two.

The protected resource *MUST* check the `aud` (audience) claim, if it exists in the token, to ensure that it includes the protected resource's identifier. The protected resource *MUST* ensure that the rights associated with the token are sufficient to grant access to the resource. For example, this can be accomplished by querying the scopes and acr associated with the token from the authorization server's token introspection endpoint.

> **iGov-NL : Additional content**
>
> In case these (optional) attributes are already provided within the token, no introspection is needed. For further details we encourage to read the *OpenID NLGov* specification.

A protected resource *MUST* limit which authorization servers it will accept valid tokens from. A resource server *MAY* accomplish this using a whitelist of trusted servers, a dynamic policy engine, or other means.

§ 4. Advanced OAuth Security Options

The preceding portions of this OAuth profile provide a level of security adequate for a wide range of use cases, while still maintaining relative ease of implementation and usability for developers, system administrators, and end users. The following are some additional security measures that can be employed for use cases where elevated risks justify the use of additional controls at the expense of implementation effort and usability. This section also addresses future security capabilities, currently in the early draft stages, being added to the OAuth standard suite.

§ 4.1 Pushed Authorization Requests (PAR)

iGov-NL : Additional content

Traditionally, OAuth 2.0 authorization requests are sent via front-channel communication (e.g., browser redirects), which exposes sensitive parameters to potential tampering or interception. PAR [*rfc9126*] addresses these vulnerabilities by allowing clients to push authorization requests directly to the authorization server over a secure back-channel. *FAPI 2.0 Security Profile* also includes this feature as of version 2.0 . Below are some of the issues it alleviates:

- **Lack of Integrity and Authenticity:** Authorization request parameters sent as URI query parameters are vulnerable to tampering. Attackers can modify values like scope or redirect_uri, potentially altering the context of transactions or access permissions. Such sensitive data in front-channel requests can be intercepted or phished, compromising client credentials or authorization codes. Attackers can exploit the request_uri parameter by injecting malicious URIs, leading to unauthorized access or token leakage.
- **Lack of Confidentiality:** Although HTTPS protects the authorization endpoint, request parameters pass through the user agent in the clear, risking exposure via browser logs, referrer headers, or other leaks. This is particularly problematic for sensitive data like personally identifiable information (PII).
- **Size Limitations:** Large authorization requests with fine-grained permissions can exceed URL size limits, causing processing errors.
- **Delayed Client Authentication:** Traditional flows delay client authentication until after user interaction, making it harder to detect and reject illegitimate requests early.

To combat this PAR allows clients to push authorization requests directly to the authorization server over a secure back-channel (HTTPS), effectively preventing tampering. For higher security, PAR can be combined with JWT-based Request Objects for cryptographic signing and optional encryption. Furthermore, by moving sensitive data from the front-channel (user agent) to the back-channel, PAR ensures that request parameters are not exposed to the browser or third parties, mitigating leakage risks. Also, PAR enables the authorization server to authenticate the client before any user interaction, allowing early detection and rejection of illegitimate requests, such as spoofing or tampering attempts.

4.2 Proof of Possession Tokens (PoP)

~~OAuth proof of possession tokens are currently defined in a set of drafts under active development in the Internet Engineering Task Force (IETF) OAuth Working Group.~~
While a bearer token can be used by anyone in possession of the token, a proof of possession token is bound to a particular symmetric or asymmetric key issued to, or already possessed by, the client. The association of the key to the token is also communicated to the protected resource.
~~a variety of mechanisms for doing this are outlined in the draft [OAuth 2.0 Proof-of-Possession (PoP) Security Architecture] [*I-D.ietf-oauth-pop-architecture*] .~~
When the client presents the token to the protected resource, it is also required to demonstrate possession of the corresponding key ~~(e.g., by creating a cryptographic hash or signature of the request)~~.

> **iGov-NL : Additional content**
> *OAuth 2.0 Demonstrating Proof of Possession (DPoP)* is an extension that describes a technique to cryptographically bind access tokens to a particular client when they are issued.

~~Proof of Possession tokens are somewhat analogous to the Security Assertion Markup Language's (SAML's) Holder-of-Key mechanism for binding assertions to user identities.~~
Proof of possession could prevent a number of attacks on OAuth that entail the interception of access tokens by unauthorized parties. The attacker would need to obtain the legitimate client's cryptographic key along with the access token to gain access to protected resources.
~~Additionally, portions of the HTTP request could be protected by the same signature used in presentation of the token.~~
Proof of possession tokens may not provide all of the same protections as PKI authentication, but they are far less challenging to implement on a distributed scale.

> **iGov-NL : Additional content**
> Another implementation of PoP is using TLS with mutual authentication, where the client is using a PKI authentication. The authorized party (`azp`) can then be verified with the client certificate to match the authorized party. As an alternative, the authorization server can include a `cnf` parameter in the JWT by the authorization server, see [*rfc7800*]. The key referenced in `cnf` can be validated using a form of client authentication, e.g. using an `private_key_jwt` or `tls_client_auth`[*rfc8705*].

iGov-NL : Additional content

More detailed information about securely implementing PoP are described in FAPI 2.0 Security Profile.

§ 4.3 Rich Authorization Requests

iGov-NL : Additional content

OAuth 2.0 Rich Authorization Requests is an extension that provides a way for clients to request and obtain fine-grained authorization from resource owners such as end users during the Authorization Code Flow. In traditional OAuth flows, clients typically request access to a set of scopes from a Resource Server. The Resource Owner then grants access to the resources to the client. However, this approach allows limited granular control over the access granted to a client and can lead to over-provisioning of access, which poses various security risks. With RAR, clients can pass an `authorization_details` claim with additional details that allow for more fine-grained authorization. This also allows for the Resource Server to implement fine-grained authorization for specific requests. Think of one-time payment approvals, document signing or requesting or approving (access to a) specific cases or documents.

According to the RFC, `authorization_details` requires just one field, `type`, which determines the allowable contents of the `authorization_details`. The value is unique for the described API in the context of the Authorization Server. The underlying RFC defines a set of common data fields that are designed to be usable across different types of APIs. Fields like `locations`, `datatypes`, `identifier` and `privileges` can be added in the `authorization_details` parameter.

As mentioned in [*RFC9396*]:

> In case of authorization requests as defined in [*RFC6749*], implementers *MAY* consider using pushed authorization requests [*RFC9126*] to improve the security, privacy, and reliability of the flow.

In case more data, or more recent data, is required for fine-grained authorization then one *MAY* include the OpenID AuthZEN standard in the `authorization_details`.

5. Security Considerations

All transactions *MUST* be protected in transit by TLS as described in [*BCP195*] .

Authorization Servers *SHOULD* take into account device postures when dealing with native apps if possible. Device postures include characteristics such as a user's lock screen setting, or if the app has 'root access' (meaning the device OS may be compromised to gain additional privilages not intended by the vendor), or if there is a device attestation for the app for its validity. Specific policies or capabilities are outside the scope of this specification.

All clients *MUST* conform to applicable recommendations found in the Security Considerations sections of [*rfc6749*] and those found in the [OAuth 2.0 Threat Model and Security Considerations document] [*rfc6819*] .

6. Security Considerations

All transactions *MUST* be protected in transit by TLS as described in [*BCP195*] .

Authorization Servers *SHOULD* take into account device postures when dealing with native apps if possible. Device postures include characteristics such as a user's lock screen setting, or if the app has 'root access' (meaning the device OS may be compromised to gain additional privilages not intended by the vendor), or if there is a device attestation for the app for its validity. Specific policies or capabilities are outside the scope of this specification.

All clients *MUST* conform to applicable recommendations found in the Security Considerations sections of [*rfc6749*] and those found in the [OAuth 2.0 Threat Model and Security Considerations document] [*rfc6819*] .

A. Multi-Actor Authorization in OAuth

1. 2.1 vanuit OIDC
2. Erbij: Korte uitleg rich authorization request en verwijn NL GOV OAuth
3. 5.2.4 Delegation Relationships
4. https://gitdocumentatie.logius.nl/publicatie/api/oidc/#example-2
5. glossary

A.1 Introduction

The OAuth and OpenID Connect (OIDC) NL GOV profiles support multiple use cases where one entity acts on behalf of another. These occur when an End-User consumes an online service on behalf of a Person (natural or juridical) or an Organisation (the service consumer). In these cases, both authentication and authorization must express not only who the End-User is, but also whom they represent or act for.

Such relationships — called Representation Relationships — must be formally established. They can arise:

- voluntarily (e.g., power of attorney),
- by legal mandate (e.g., guardian, court-appointed administrator),
- or by corporate capacity (e.g., director, statutory signatory).

The formalization of these relationships is out of scope of this profile; this document focuses on how these relationships are conveyed within OAuth and OIDC tokens.

A.1.1 Delegation vs Representation

Concept	Description	Token claim Pattern
Delegation	The End-User or service grants another party the right to act on their behalf, while the delegated party remains clearly identifiable as a separate actor.	`act` claim (delegated actor identified)
Representation	The actor presents themselves *as* another principal, and is treated as that entity by relying parties.	`represents` claim (represented party identified)

So in terms of examples:

- Delegation: A service or application acts on behalf of a user (e.g., backend API call with act claim).
- Representation: A natural person acts as an organization (e.g., director signing an electronic transaction).

A.2 OAuth tokens

In traditional OAuth, an access token represents a single subject (the resource owner). However, many governmental and organizational use cases require multi-actor authorization, where multiple identities participate in or influence the authorization decision. Examples include:

- an End-User acting as a legal entity,
- an application acting for a user,
- a chain of representation, such as citizen → intermediary → government service.

To model these scenarios, OAuth provides a mechanism for expressing actor relationships through specific claims:

- [*rfc8693*] OAuth 2.0 Token Exchange — defines `act` and `may_act` claims for representing delegation chains.
- [*rfc9396*] Rich Authorization Requests (RAR) — defines how specific authorization details can be requested and conveyed between parties.
- profile-specific claims such as represents introduced in this document.

A.2.1 Application in NL GOV Profiles

Token Exchange ([*rfc8693*]) will be included in an future release of the OAuth 2.0 NL GOV Profile. See the draft version: Logius OAuth Profile – Grant Types

A.3 Delegation Relationships

In Use Cases that involve Delegation Relationships, as specified in [*rfc8693*], an actor (for example, an intermediary or system) is authorized to perform actions on behalf of another principal, while remaining a distinct identity.

Resources *SHOULD* process the `act` claim when identification of an intermediary or other acting party is applicable in the context of the OpenID Client and Resource Server.

A.3.1 claim Structure

This profile specifies delegation relationships in ID Tokens as follows:

- The `sub` claim identifies the represented user (the subject of the authorization).
- The `act` (or `may_act`) claim identifies the intermediary or acting party.
- In case of a delegation chain (multiple parties acting on behalf of each other), these can be nested via multiple `act` objects.
- Each `act` claim:
  - *MUST* contain `sub` and `iss` claims to uniquely identify the acting party.
  - *SHOULD* include a `subject_type` claim to indicate the identifier type if multiple types are supported.
  - *MAY* include additional identity-related claims (e.g., `email`) to convey useful information about the acting party.

§ A.3.2 Example: Nested Delegation

EXAMPLE 14

[Example 3](https://gitdocumentatie.logius.nl/publicatie/api/oidc/#example-3)

A sample delegation chain may look like this (note: the requested scope also includes the required `openid` scope and a fictional `brp_sensitief` scope; claims not essential to the example are omitted for readability):

```
{
  "scope": "openid brp_sensitief",
  /* Represented party – user with access rights */
  "sub": "RKyLpEVr1L",
  "subject_type": "public",
  "sub_id_type": "urn:nl-eid-gdi:1.0:id:pseudonym",
  "aud": ???,
  "act": {
    /* Intermediary in reperesentation chain - an organization oacting
    "sub": "492099595",
    "subject_type": "public",
    "aud": ???,
    "sub_id_type": "urn:nl-eid-gdi:1.0:id:RSIN",
    "act": {
      /* Individual acting on behalf of the intermediary organization
      "sub": "4Yg8u72NxR",
      "subject_type": "pairwise",
      "aud": "urn:nl-eid-gdi:1.0:id:pseudonym" // klopt dit?
    }
  }
}
```

§ A.3.3 Integration with Rich Authorization Requests (RAR)

If more specific authorization information is required — for example, identifying the organization (e.g., RSIN or KvK number), data categories, or access levels — such details *MAY* be included using the `authorization_details` claim, as defined in [*rfc9396*].

EXAMPLE 15

RAR example:

```
"authorization_details": {
  "type": "party_authorization_example",
  /* represented party - an organization in this example */
  "represented_party": {
    "sub": "492099595",
    "subject_type": "public",
    "sub_id_type": "urn:nl-eid-gdi:1.0:id:RSIN"
  }
}
```

A.4 Representation Relationships in NL GOV Context

In Use Cases that involve Representation Relationships, or other situations where a token must convey context about who is represented within a specific scope, *OAuth 2.0 Rich Authorization Requests* *MAY* be used to include explicit representation details within the `authorization_details` claim.

A representation relationship expresses that a principal (for example, an intermediary system or individual) acts as another principal (for example, a legal entity or organization). In contrast to delegation, where the actor remains a distinct identity, representation implies that the actor is treated as the represented party for the duration of the transaction.

The RAR object allows these relationships to be expressed in a structured, scope-specific way.

EXAMPLE 16

A sample chain representation for a requested scope `urn:uuid:a9e17a2e-d358-406d-9d5f-ad6045f712ba` may look like (note: the requested scope also includes the required `openid` scope; claims that do not add to the example are omitted for readability):

```
{
  "scope": "openid brp_sensitief",
  /* Intermediary in representation chain - a system (client) in
  "sub": "example-client-id",
  "subject_type": "public",
  "iss": "example.as,
  "authorization_details": {
    “type”: “party_authorization_example"
     /* represented party - an organization in this example */
     “represented_party”:{
          "sub": "492099595",
        "subject_type": "public",
        "iss": "urn:nl-eid-gdi:1.0:id:RSIN",
        /* person acting on behalf of the represented organisati
        "responsible_person": {
          "sub": "4Yg8u72NxR",
          "subject_type": "pairwise",
          "iss": "urn:"
            }
     }
}
```

§ A.4.1 Implementation Guidance

- The `authorization_details` object *MAY* include one or more `represented_party` elements to indicate the party or organization on whose behalf the client or user acts.
- Nested relationships (for example, `organization → representative → sub-representative`) *MAY* be expressed through nested objects inside `represented_party`.
- Each object *SHOULD* include the following claims:
  - `sub` and `iss` — to uniquely identify the represented party.
  - `subject_type` — to indicate the type of identifier used (for example, `public`, `pairwise`, RSIN, KvK).

Additional contextual claims (e.g., `responsible_person`, `role`, or `mandate_type`) *MAY* be included to convey the legal or organizational basis of the representation.

Claims unrelated to identity (such as `exp`, `nbf`, or `aud`) *MUST NOT* appear within these objects.

§ A.5 Glossary

Term	Definition
End-User	The natural or juridical person who authenticates and initiates an authorization flow.
Service Consumer	The entity (person or organization) for which the service is ultimately consumed.
Representation Relationship	A formally defined relationship where one actor represents another.
Delegation	Authorization for an actor to act on behalf of another while retaining its own identity.
Representation	Acting as another entity, where the system treats the actor as the represented party.
Multi-Actor Authorization	Authorization scenario involving more than one principal influencing the decision.

✧ ✧ ✧

Hieronder is een stukje van oude tekst voor opmaak en kan genegeerd worden.

§ A.6 Representation Relationships

In Use Cases that involve Representation Relationships and other situations where a token is meant to be used in the context of a specific scope, Rich Authorization requests *MAY* be used Example 4 E.g. an intermediary party acting on behalf of a government party.

```
{
  "scope": "openid brp_sensitief",
  /* Intermediary in representation chain - a system (client) in this */
  "sub": " example-client-id",
  "subject_type": "public",
  "iss": "example.as",
```

```
  "authorization_details": {
        “type”: “party_authorization_example
        “represented_party”:{
                "sub": "492099595",
               /* represented party - an organization in this example */
               "subject_type": " public",
               "iss": " urn:nl-eid-gdi:1.0:id:RSIN ",
               "responsible_person": {
                        /* person acting on behalf of the repreented org
                        "sub": "4Yg8u72NxR",
                       "subject_type": "pairwise",
                        "iss": " urn:<..eherkenning…>. "
                }
          }
}
}
```

B. References

B.1 Normative references

[BCP195]

Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). Y. Sheffer; R. Holz; P. Saint-Andre. IETF. May 2015. URL: https://tools.ietf.org/html/bcp195

[I-D.ietf-oauth-pop-architecture]

OAuth 2.0 Proof-of-Possession (PoP) Security Architecture. P. Hunt, J. Richer, W. Mills, P. Mishra, H. Tschofenig. IETF. July 8, 2016. URL: https://tools.ietf.org/html/draft-ietf-oauth-pop-architecture-08

[ietf-oauth-v2-1-10-refresh-token-grant]

The OAuth 2.1 Authorization Framework: Refresh Token Grant. Dick Hardt, Aaron Parecki, Torsten Lodderstedt. IETF. January 9, 2024. URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-10#name-refresh-token-grant

[JWS.JWE.Algs]

IANA JSON Web Signatures and Encryption Algorithms registry. Jim Schaad, Jeff Hodges, Joe Hildebrand, Sean Turner. IANA. URL: https://www.iana.org/assignments/jose/jose.xhtml#web-signature-encryption-algorithms

[NLGOV.OpenID]

OpenID NLGov. Remco Schaar, Frank van Es, Joris Joosten, Jan Geert Koops. Logius. URL: https://gitdocumentatie.logius.nl/publicatie/api/oidc/

[OpenID.Core]

OpenID Connect Core 1.0. N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, C. Mortimore. OpenID foundation. November 8 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html

[OpenID.Discovery]

OpenID Connect Discovery 1.0. N. Sakimura, J. Bradley, M. Jones, E. Jay. OpenID foundation. November 8 2014. URL: https://openid.net/specs/openid-connect-discovery-1_0.html

[OpenID.FAPI2.0]

FAPI 2.0 Security Profile. D. Fett, D. Tonge, J. Heenan. OpenID foundation. February 22 2025. URL: https://openid.net/specs/fapi-security-profile-2_0-final.html

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc2119

[rfc4122]

A Universally Unique IDentifier (UUID) URN Namespace. P. Leach; M. Mealling; R. Salz. IETF. July 2005. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc4122

[rfc6749]

The OAuth 2.0 Authorization Framework. D. Hardt, Ed. IETF. October 2012. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6749

[rfc6750]

The OAuth 2.0 Authorization Framework: Bearer Token Usage. M. Jones; D. Hardt. IETF. October 2012. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6750

[rfc6819]

OAuth 2.0 Threat Model and Security Considerations. T. Lodderstedt, Ed.; M. McGloin; P. Hunt. IETF. January 2013. Informational. URL: https://www.rfc-editor.org/rfc/rfc6819

[rfc7009]

OAuth 2.0 Token Revocation. T. Lodderstedt, Ed.; S. Dronia; M. Scurtescu. IETF. August 2013. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7009

[RFC7515]

JSON Web Signature (JWS). M. Jones; J. Bradley; N. Sakimura. IETF. May 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7515

[rfc7516]

JSON Web Encryption (JWE). M. Jones; J. Hildebrand. IETF. May 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7516

[rfc7517]

JSON Web Key (JWK). M. Jones. IETF. May 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7517

[rfc7518]

JSON Web Algorithms (JWA). M. Jones. IETF. May 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7518

[rfc7522]

Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants. B. Campbell; C. Mortimore; M. Jones. IETF. May 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7522

[rfc7523]

JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants. M. Jones; B. Campbell; C. Mortimore. IETF. May 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7523

[rfc7591]

OAuth 2.0 Dynamic Client Registration Protocol. J. Richer, Ed.; M. Jones; J. Bradley; M. Machulak; P. Hunt. IETF. July 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7591

[rfc7636]

Proof Key for Code Exchange by OAuth Public Clients. N. Sakimura, Ed.; J. Bradley; N. Agarwal. IETF. September 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7636

[rfc7662]

OAuth 2.0 Token Introspection. J. Richer, Ed. IETF. October 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7662

[rfc7800]

Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs). M. Jones; J. Bradley; H. Tschofenig. IETF. April 2016. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7800

[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc8174

[rfc8414]

OAuth 2.0 Authorization Server Metadata. M. Jones; N. Sakimura; J. Bradley. IETF. June 2018. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8414

[rfc8693]

OAuth 2.0 Token Exchange. M. Jones; A. Nadalin; B. Campbell, Ed.; J. Bradley; C. Mortimore. IETF. January 2020. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8693

[rfc8705]

OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens. B. Campbell; J. Bradley; N. Sakimura; T. Lodderstedt. IETF. February 2020. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8705

[rfc9068]

JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens. V. Bertocci. IETF. October 2021. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9068

[rfc9126]

OAuth 2.0 Pushed Authorization Requests. T. Lodderstedt; B. Campbell; N. Sakimura; D. Tonge; F. Skokan. IETF. September 2021. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9126

[RFC9396]

OAuth 2.0 Rich Authorization Requests. T. Lodderstedt; J. Richer; B. Campbell. IETF. May 2023. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9396

[RFC9449]

OAuth 2.0 Demonstrating Proof of Possession (DPoP). D. Fett; B. Campbell; J. Bradley; T. Lodderstedt; M. Jones; D. Waite. IETF. September 2023. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc9449

§ B.2 Informative references

[HEART.OAuth2]

Health Relationship Trust Profile for OAuth 2.0. J. Richer. OpenID foundation. April 25, 2017. URL: https://openid.net/specs/openid-heart-oauth2-1_0.html

[iGOV.OAuth2]

International Government Assurance Profile (iGov) for OAuth 2.0. J. Richer, M. Varley, P. Grassi. OpenID foundation. October 5 2018. URL: https://openid.net/specs/openid-igov-oauth2-1_0-03.html

[rfc7519]

JSON Web Token (JWT). M. Jones; J. Bradley; N. Sakimura. IETF. May 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7519

↥