

# NL GOV Assurance profile for OAuth 2.0 v1.0



Logius Standard

Definitive version 09 juli 2020

[Overzicht standaarden](#)

**This version:**

<https://publicatie.centrumvoorstandaarden.nl/api/oauth/v1.0>

**Latest published version:**

<https://publicatie.centrumvoorstandaarden.nl/api/oauth/>

**Latest editor's draft:**

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

**Editors:**

Frank Terpstra ([Geonovum](#))

Jan van Gelder ([Geonovum](#))

**Authors:**

Jaron Azaria ([Logius](#))

Martin Borgman ([Kadaster](#))

Marc Fleischeuers ([Kennisnet](#))

Peter Haasnoot ([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 licensed under a [Creative Commons Attribution 4.0 License](#).

---

*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.

## Status of This Document

This is the definitive version of the standard. Edits resulting from consultations have been applied.

Het OBDO heeft op advies van het Forum Standaardisatie deze versie vastgesteld.

## Table of Contents

### **Abstract**

### **Status of This Document**

## **1. Introduction**

- 1.1 Requirements Notation and Conventions
- 1.2 Terminology
- 1.3 Conformance

## **2. Client Profiles**

- 2.1 Client Types
  - 2.1.1 Full Client with User Delegation
  - 2.1.2 Native Client with User Delegation
  - 2.1.3 Direct Access Client
- 2.2 Client Registration
  - 2.2.1 Redirect URI
- 2.3 Connection to the Authorization Server
  - 2.3.1 Requests to the Authorization Endpoint
  - 2.3.2 Response from the Authorization Endpoint
  - 2.3.3 Requests to the Token Endpoint
  - 2.3.4 Client Keys
- 2.4 Connection to the Protected Resource
  - 2.4.1 Requests to the Protected Resource

## **3. Authorization Server Profile**

- 3.1 Connections with clients
  - 3.1.1 Grant types
  - 3.1.2 Client authentication
  - 3.1.3 Dynamic Registration
  - 3.1.4 Client Approval
  - 3.1.5 Discovery
  - 3.1.6 Revocation
  - 3.1.7 PKCE
  - 3.1.8 Redirect URIs
  - 3.1.9 RefreshTokens
  - 3.1.10 Token Response
- 3.2 Connections with protected resources
  - 3.2.1 JWT Bearer Tokens
  - 3.2.2 Introspection
- 3.3 Response to Authorization Requests
- 3.4 Token Lifetimes
- 3.5 Scopes

## **4. Protected Resource Profile**

- 4.1 Protecting Resources
  - 4.1.1 Example
    - 4.1.1.1 Request
    - 4.1.1.2 Response:
- 4.2 Connections with Clients
- 4.3 Connections with Authorization Servers

## 5. Advanced OAuth Security Options

### 5.1 Proof of Possession Tokens

## 6. Security Considerations

### A. References

#### A.1 Informative references

## Dutch government Assurance profile for OAuth 2.0

This profile is based upon the international government assurance profile for OAuth 2.0 (iGov) [\[GOV.OAuth2\]](#) 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 situation and in the Netherlands we have to deal with an European Union context.

We have added the chapter [Usecases](#) to illustrate the specific usecase 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 ~~strike through~~ to indicate it is not part of iGov-NL. Where we have added more specific requirements for the Dutch situation this is indicated with **iGov-NL** tags.

## Usecases

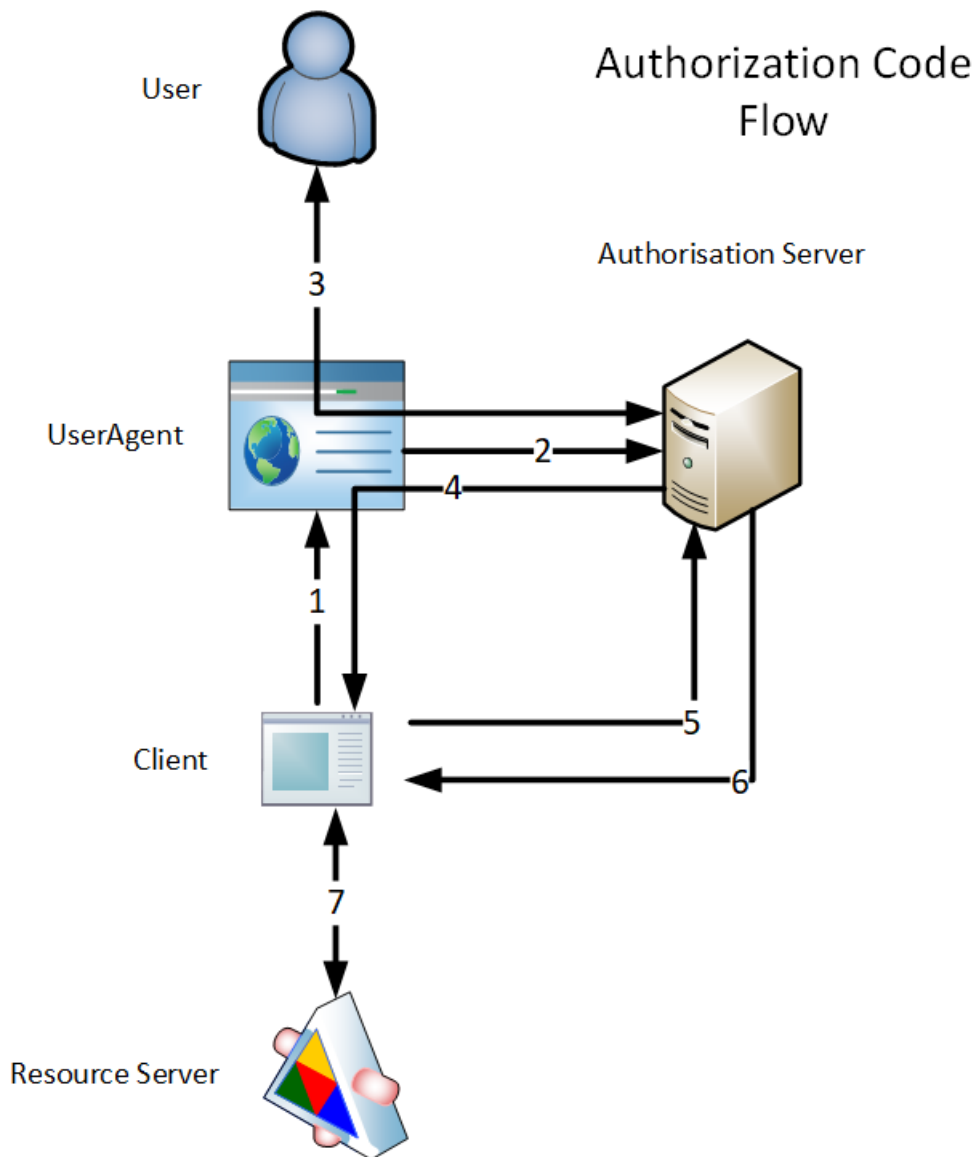


Figure 1 Use case Authorization code flow

## Introduction

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.

## Context

### 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.

## Flow for authorization

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`.

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.

### 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 Authorization 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

Note: applicable to the Authorization Code Flow only. 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 an 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.

## 1. 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] [\[GOV.OAuth2\]](#).

### 1.1 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)] [\[fc7515\]](#) and [JSON Web Encryption (JWE)] [\[fc7516\]](#) 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.

## 1.2 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].

## 1.3 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

This profile is based upon the international government assurance profile for OAuth 2.0 (iGov) [GOV.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 situation and in the Netherlands we have to deal with an European Union context.

### /iGov-NL

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

An **iGoviGov-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-**iGoviGov-NL** clients and protected resources.

An **iGoviGov-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 **iGoviGov-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-**iGoviGov-NL** authorization servers and clients.

## 2. Client Profiles

### 2.1 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.

### 2.1.1 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.

These clients *MUST* be associated with a unique public key, as described in [Section 2.2](#).

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

### 2.1.2 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.

### 2.1.3 Direct Access Client

#### iGov-NL

Direct Access Clients are out of scope in this version of iGov-NL.



~~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.~~

## 2.2 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.

### 2.2.1 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").

## 2.3 Connection to the Authorization Server

### 2.3.1 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

Native clients *MUST* apply PKCE, as per RFC7636. As `code_verifier` the S256 method *MUST* be applied. Effectively this means that a Native Client *MUST* include a cryptographic random `code_challenge` 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 `ffc4122`)

### code\_challenge\_method

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

## /iGov-NL

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=code&scope=openid+email&redirect_u
https%3A%2F%2Fclient.example.org%2Fcb
Status: 302 Found
```

This causes the browser to send the following (non-normative) request to the authorization endpoint (inline wraps for display purposes only):

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

### 2.3.2 Response from the Authorization Endpoint

## iGov-NL

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.

**iGov-NL****2.3.3 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:

**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.

The following sample claim set illustrates the use of the required claims for a client authentication JWT as defined in this profile; additional claims *MAY* be included in the claim set.

```
{
  "iss": "55f9f559-2496-49d4-b6c3-351a586b7484",
  "sub": "55f9f559-2496-49d4-b6c3-351a586b7484",
  "aud": "https://idp-p.example.com/token",
  "iat": 1418698788,
  "exp": 1418698848,
  "jti": "1418698788/107c4da5194df463e52b56865c5af34e5595"
}
```

The JWT assertion *MUST* be signed by the client using the client's private key. See [Section 2.2](#) 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**

In addition to above signing methods, the Authorization server *SHOULD* support PS256 signing algorithm [[RFC7518](#)] for the signing of the **private\_key\_jwt**.

Effectively, the Token Request has the following content:

**grant\_type**

Mandatory. *MUST* contain the value **authorization\_code**

**code**

### 2.3.4 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 `jwt` field or by registering a `jwt_uri` that *MUST* be reachable by the authorization server. It is *RECOMMENDED* that clients use a `jwt_uri` if possible as this allows for key rotation more easily. This applies to both dynamic and static (out-of-band) client registration.

The `jwt` field or the content available from the `jwt_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 `jwt_uri` document and verify that it contains a JWK Set. The following example is of a 2048-bit RSA key:

```
{
  "keys": [
    {
      "alg": "RS256",
      "e": "AQAB",
      "n": "kAMYD62n_f2rUcR4awJX4uccDt0zcXRssq_mDch5-ifcShx9aTtTVza23PTn3KaKrsBXwWcfioXR6zQn5eYdZQVGNBf0R4rxF5i7t3hfb4Wks50EK1gBYk2l09NSrQ-xG9QsUsAnN6RHksXqsd0qv-nxjLexDfIJlgbcCN9h6TB-C66ZXv7PVhl19gIYVifSU7liHkLe0l0fw7jUI6rHLHf4d96_neR1HrNIK_xssr99Xpv1EM_ubxpktX0T925-qej9fMEpzzQ5HLMcNt1H2_VQ_Ww1J0Ln9vRn-H48FDj7TxlIT74XdTZgTv31w_GRPAP0fyxEw_ZUmxhz5Z-gTlQ",
      "kty": "RSA",
      "kid": "oauth-client"
    }
  ]
}
```

## iGov-NL

In case the Authorization Server, Resource Server and client are not operated under responsibility of the same organisation, each party *MUST* use PKI-overhead certificates with OIN. The PKI-overhead 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.

## /iGov-NL

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

```
{
  "alg": "RS256",
  "d": "PjIX4i2NsBQu0VIw74ZDjqthYsoFvaoah9GP-cPrai5s5VUIlLoadEAdGbBrss_6dR58x_pRlPHWh04vLQsFBuWQnc9SN306TAaai9Jg5TlCi6V0d406lUoTYpMR0cxFIU-xFMwII--_OZRgmAxiYiAXQj7TKMKvgSvV07-9-YdhMwHoD-UrJkfnZckMKSS0BoAbjReTski3IV9f1wVJ53_pmr9NBpiZeHYmmG_1QDSbBuY35ZummUt4QShF-fey2gSALdp9h9hRk1p1fsTZtH2lwpvm0cjwDkSDv-z0-4Pt8Nu0yqNVPFahR0BPlsMVxc_zjPck8ltblalBHPo6AQ",
  "e": "AQAB",
  "n": "kAMYD62n_f2rUcR4awJX4uccDt0zcXRssq_mDch5-ifcShx9aTtTVza23PTn3KaKrsBXwWcfioXR6zQn5eYdZQVGNBf0R4rxF5i7t3hfb4Wks50EK1gBYk2l09NSrQ-xG9QsUsAnN6RHksXqsd0qv-nxjLexDfIJlgbcCN9h6TB-C66ZXv7PVhl19gIYVifSU7liHkLe0l0fw7jUI6rHLHf4d96_neR1HrNIK_xssr99Xpv1EM_ubxpktX0T925-qej9fMEpzzQ5HLMcNt1H2_VQ_Ww1J0Ln9vRn-H48FDj7TxlIT74XdTZgTv31w_GRPAP0fyxEw_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.

## 2.4 Connection to the Protected Resource

### 2.4.1 Requests to the Protected Resource

Clients *SHOULD* send bearer tokens passed in the Authentication header as defined by [\[fc6750\]](#) . Clients *MAY* use the form-parameter ~~or query-parameter~~ methods in [\[rfc6750\]](#) . Authorized requests *MUST* be made over TLS, and clients *MUST* validate the protected resource server's certificate.

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.eyJleHAiOjE0MTg3MDI0MTIsImF1ZCI6WyJjMWJjODRlNC00N2VlLTRiNjQ0YmI1Mi01Y2RhNmM4MwY3ODgiXSwiaXNzIjoiaHR0cHM6XC9cL2lkC1wLmV4YW1wbGUuY29tXC8iLCJqdGkiOiJkM2Y3YjQ4Zi1iYzgxLTQwZWMTYTE0MC05NzRhZjc0YzRkZTMiLCJpYXQiOjE0MTg2OTg4MTJ9.LmZ_tzZ90_b0QZS-AXtQtvcLZ7M4uDas1WxCfXpgBfBanolW37X8h1ECrUJexbXMD6rrj_uuWEqPD738oWRo0r0noKJAgbF1GhXPAYnN5pZRYgWSD1a6RcmN85SxUig0H0e7drmdmRkPQgb12wMhu-6h20qw-ize4dKmykN9UX_2drXrooSxpRZqFVYX8PkCvCCBuFy20-HPRov_SwtJmK5qjUwMyn2I4Nu2s-R20aCA-7T5dunr0iWckLQnVnaXMfA22RlRiU87nl21zappYb1_EHF9ePyq3Q353cDUY7vje8m2kKXYTgc_bUAYuW-W3SMSw5U1KaHtSZ6PQICoA
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)
```

## 3. 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" [\[fc6819\]](#) .

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

### 3.1 Connections with clients

#### 3.1.1 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.~~~~

#### 3.1.2 Client authentication

The authorization server *MUST* enforce client authentication as described above for the `authorization_code` ~~and `client_credentials` grant types~~ . Public client cannot authenticate to the authorization server.

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

#### 3.1.3 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. ~~Clients *MUST NOT* dynamically register for the client credentials grant type~~ 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"

**jwtks\_uri or jwtks**

client's public key in JWK Set format; if jwtks\_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**

In this version of iGov-NL we follow iGov for the requirement that the Authorization servers *MUST* 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.

**/iGov-NL**

### 3.1.4 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.

### 3.1.5 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.

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



```

{
  "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",
    "implicit",
    "urn:ietf:params:oauth:grant-type:jwt-bearer",
    "client_credentials",
    "urn:ietf:params:oauth:grant-type:redelegate"
  ],
  "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"
}

```

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

**key**

The key type

**alg**

The default algorithm used for this key

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

```
{
  "keys": [
    {
      "alg": "RS256",
      "e": "AQAB",
      "n": "o80vbR0ZFmhjZWfqwPUGNkcIeUcweFyzB2S2T-hje83IOVct8gVg9FvHPK1R
eEW3-p7-A8GNcLAuFP_8jPhiL6LyJC3F10aV9KPQFF-w6Eq6VtpEgYSfzvFegNiPtpMwd7C43
EDwjQ-GrXMVCLrBYxZC-P1ShyxVB0zeR_5MTC0JGiDTecr_2YT6o_3aE2SIJu4iNPgGh9Mnyx
dBo0Uf0TmrqEIabquXA1-V8iUihwfI8qjf3EujkYi7gXXelIo4_gipQYNjr4DBNlE0__RI0kD
U-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

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.

## /iGov-NL

### 3.1.6 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.

### 3.1.7 PKCE

An authorization server *MUST* support the Proof Key for Code Exchange ([PKCE] [\[fc7636\]](#)) 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.

### 3.1.8 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.

### 3.1.9 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.

### 3.1.10 Token Response

#### iGov-NL

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](#). **iGov-NL**

## 3.2 Connections with protected resources

Unlike the core OAuth protocol, the **iGoviGov-NL** profile intends to allow compliant protected resources to connect to compliant authorization servers.

### 3.2.1 JWT Bearer Tokens

In order to facilitate interoperability with multiple protected resources, all **iGoviGov-NL**-compliant authorization servers issue cryptographically signed tokens in the JSON Web Token (JWT) format. 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. When combined with discovery, this information is sufficient to programmatically locate the token introspection service, which is in turn used for conveying 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

##### azp

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.

The server *MAY* issue tokens with additional fields, including the following as defined here:

**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 audience requires otherwise.

**iGov-NL**

In iGov-NL the sub claim *MUST* be present.

## /iGov-NL

**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.

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,
  "azp": "55f9f559-2496-49d4-b6c3-351a586b7484",
  "iss": "https://idp-p.example.com/",
  "jti": "2402f87c-b6ce-45c4-95b0-7a3f2904997f",
  "iat": 1418698788
}
```

The access tokens *MUST* be signed with [JWS] [\[rfc7515\]](#) . 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

In addition to above signing methods, the Authorization server *SHOULD* support PS256 signing algorithm [RFC7518] for the signing of the JWT Bearer Tokens.

## /iGov-NL

**kid**

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

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

eyJhbGciOiJSUzI1NiJ9.ew0KICAgImV4cCI6IDE0MTg3MDIzODgsDQogICAiYXpwIjogZjIuUzJlbnNTU5LTI0OTYtNDlkNC1iNmMzLTM1MWE1ODZiZnQ4NCIsDQogICAiaXNzIjogZlthbDhBZ0i8vaWRwLXAuZXhhbXB5ZS5jb20vIiwNCiAgICJqdGkiOiAiMjQwMmY4N2MtYjZjZS00NW00LTK1YjAtN2EzZjI5MDQ50TdmIiwNCiAgICJpYXQ0IiAxNDE4NjQ4Nzg0LTA0KICAgImtpZCI6ICJyc2ExIgowKfQ. iB6Ix8Xeg-L-nMStgE1X75w7zgXabzw7znWUEC0sXpHfNYYqb-CET9Ah5IQyX1DZ20qEyN98UydgSTpi01YJDDcZV4f4DgY0ZdG3yBW3XqwUQwbgF7Gwza9Z4AdhjHjzQx-lChXayfL1xz0SBDkVbJdJtXbvaSIyff7ueWF3M1CM70-GXuRY4iucKbuytz9e7eW4Egkk4Aag13iTk9-l5V-tvL6dYu8I1R93GKsaKE8bng0EZ04xcnq8s4V5Yykuc\_NARBjENiKTJM8w3wh7xWP2gvMp39Y0XnuC0LyIx-J1ttX83xmpXDaLvvy-4HT9XHT9V73fKF8rLWJu9qrrA

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

**iGov-NL**

/iGov-NL

**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

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.

### 3.3 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](#)]

The following is an example response:

```
https://client.example.org/cb?
state=2ca3359dfbfd0
&code=g0IFJ1hV6Rb1sxUdFhZGACWwR1sMhYbJJcQbVJN0wHA
```

### 3.4 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.

### 3.5 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.

## 4. Protected Resource Profile

### 4.1 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.

For example: 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.

## iGov-NL

### 4.1.1 Example

#### 4.1.1.1 Request

```
GET /resource HTTP/1.1
Authorization: Bearer 4f626847-91b1-3417-a91e-c5627f377ae1
Accept: text/plain, application/json, application/*+json, */*
Host: resource.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.3 (java 1.5)
```

#### 4.1.1.2 Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "sub": "248289761001",
  "name": "Jane Doe",
  "given_name": "Jane",
  "family_name": "Doe",
  "preferred_username": "j.doe",
  "email": "janedoe@example.com",
  "picture": "http://example.com/janedoe/me.jpg"
}
```



## 4.2 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~~ methods.

### **iGov-NL**

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.

### **/iGov-NL**

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

## 4.3 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.

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.

## 5. 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.

### 5.1 Proof of Possession Tokens

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).

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

Proof of possession can be implemented using various methods. An example of such an implementation is using TLS with mutual authentication, where the client is using a PKI-overheid certificate. 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 [fc7800]. The key referenced in cnf can be validated using a form of client authentication, e.g. using an private\_key\_jwt.

#### /iGov-NL

## 6. Security Considerations

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

#### iGov-NL

In addition to the Best Current Practice for TLS, it is highly *RECOMMENDED* for all conforming implementations to incorporate the TLS guidelines from the Dutch NCSC into their implementations. If these guidelines are applied:

- For back-channel communication, the guidelines categorized as "good" *MUST* be applied.
- For front-channel communication, the guidelines for "good" *MUST* be applied and the guidelines for "sufficient" *MAY* be applied, depending on target audience and support requirements.
- Guidelines categorized as "insufficient" *MUST NOT* be applied and those categorized as "phase out" *SHOULD NOT* be used.

#### /iGov-NL

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 privileges 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] [fc6819] .

## A. References

### A.1 Informative 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>

#### [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](https://openid.net/specs/openid-heart-oauth2-1_0.html)

#### [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>

**[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.html](https://openid.net/specs/openid-igov-oauth2-1_0.html)

**[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>

**[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](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](https://openid.net/specs/openid-connect-discovery-1_0.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>

**[rfc7519]**

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

**[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>

**[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>