DIGITALISERINGSSTYRELSEN

OIO Open ID Connect Profiles
Version 0.3

Status: Draft
Date: 07.08.2020

1 Introduction

1.1 Preface

The Danish Agency for Digitisation is planning to establish new infrastructure in NemLog-in3 to support native apps and manage their access to external APIs based on OAuth and Open ID Connect. Web apps will also be allowed to use the new functionality as a more modern alternative to the current SAML implementation.

A central goal for NemLog-in3 will be to provide a modern authentication and authorization infrastructure that can be reused across a large number of business applications, clients and APIs. The new infrastructure will among other things include an Authorization Server, a Token Server and web portals for registration and management of clients and APIs. The infrastructure will ensure authentication of the end-user based on [NSIS] levels of assurance and subsequent authorization of the client (e.g. app) based on end-user consent followed by issuance and management of security tokens – similar to how the existing NemLog-in solution currently supports web applications and SOAP-services based on SAML and WS-Trust.

The first step in realizing a new infrastructure is to establish the necessary specifications and profiles that ensure interoperability and a high level of security. This document contains deployment profiles of OpenID Connect [OIDC] and OAuth 2.0 detailing protocols for the interaction between a client and an Authorization Server, Token Server and REST APIs. The specifications are written with NemLog-in3 in mind but can freely be used elsewhere.

The profiles can be used with three types of clients: native apps, web-applications with a backend, and Javascript applications with no backend.

1.2 Audience

The document is written for a technical audience including architects, security professionals and developers already familiar with OAuth 2.0, Open ID Connect, JWT, REST, TLS and other related technologies and specifications.

1.3 Usage Scenarios

This profile is intended for use within Danish public sector federations where information about authenticated identities is federated across service providers. The goal is to achieve standardization, interoperability, security and privacy, while enabling re-use of common implementations.

The current version focuses on the most common scenarios involving native apps and web apps. More advanced use cases may however be added later – including scenarios with federated Authorization Servers or APIs exchanging incoming tokens for downstream invocation of other APIs.

1.4 Overview of Specifications

A set of specifications and documents will be developed, covering various aspects of App scenarios:

- This document covers protocols for the interaction between a client and an Authorization Server, Token Server and REST APIs. The goal is to get Access Tokens issued to a client which then be used to gain access to an external API.
- The OIO JWT Token Profile [OIO JWT] specifies formats for JWT tokens used with this profile, including claims, privileges and signatures. It is inspired by the OIO SAML 3.0 Web SSO profile [OIOSAML] and OIO Basic Privilege Profile [OIO-BPP].
- Terms and conditions for using the NemLog-in infrastructure solution will be defined, including expected behavior of Service Providers, terms of use, responsibilities etc.
- A guide to the registration portal in NemLog-in will describe how clients and APIs are registered and governed including relevant metadata, certificates, scopes, approval processes etc.

The first two specifications in the above list are independent of NemLog-in and can be used everywhere. In particular, early local implementations of Authorization and Token Servers can use them in order to pave the way for a smooth transition from a local to a central implementation provided by NemLog-in. This approach thus minimizes the risk of redoing the client or API implementation at a later stage.

The last two documents are specific to NemLog-in's future implementation and are not on critical path for early, local implementations.

2 Notation and terminology

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 BCP 14 [RFC2119][RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification uses the following typographical conventions in text: `<ns:Element>`, `Attribute`, **Datatype**, `OtherCode`. The normative requirements of this specification are individually labeled with a unique identifier in the following form: **[OIO-EXAMPLE-01]**. All information within these requirements should be considered normative unless it is set in *italic* type. Italicized text is non-normative and is intended to provide additional information that may be helpful in implementing the normative requirements.

2.1 Terminology

This specification describes flows involving the following actors:

- **Client** – a native app or browser app acting in the role of client in OAuth and OpenID Connect sense. It provides application services to the enduser, requests access tokens and consumes one or more external (REST) APIs e.g. for retrieving or updating data about the end-user.
- **SP API** – Service Provider API. An API offered by a Service Provider which is protected by a trusted Authorization- and Token Server – i.e. all API access requires a signed token from these. The API Service Provider can be the same or a different organization providing the client.
- **End-user** – a person authorizing client access on his/her behalf regarding defined scopes, and in case the client as a native app installs the app on his/her personal mobile device.
- **Authorization Server** – a central OAuth 2.0 infrastructure component (in the future delivered by NemLog-in).
- **Token Server** – OIDC/OAuth 2.0 infrastructure component (in the future delivered by NemLog-in) that issues tokens which provide access to external APIs.

Three types of clients are supported by this profile:

a) **Native apps** installed on the end-user device which consume external REST APIs. These will be considered public clients as defined in OAuth 2.0. It is assumed that the device is personal and that access and refresh tokens for the user can be securely stored on the device; if this is not the case, one of the other variants below should be used.
b) **Web Applications with a backend**. In these applications, Javascript code is loaded from a dynamic Application Server that also has the ability to execute code itself. It is assumed that the Application Server performs the OAuth and

OIDC interactions itself and keeps tokens stored internally, creating a separate session with the browser via a traditional session cookie - see [BBA] for additional details. The Application Server (backend) will be considered a confidential client for the purposes of its OAuth interactions.

c) **Javascript Applications** without a backend (also known as Single Page Applications). Here the entire application runs in the browser, and the client should therefore be considered a public client. Note that to be able interact with Authorization Servers and Token Servers from different domains, these must support the necessary CORS headers in order to avoid same-origin restrictions imposed by browsers.

The vast majority of the flows and requirements in this profile apply to all three types of clients, and where requirements do depend on the client type, it will be stated explicitly. The illustrations and drawings primarily show examples of clients being native apps, since this has been the main reason for writing the specification.

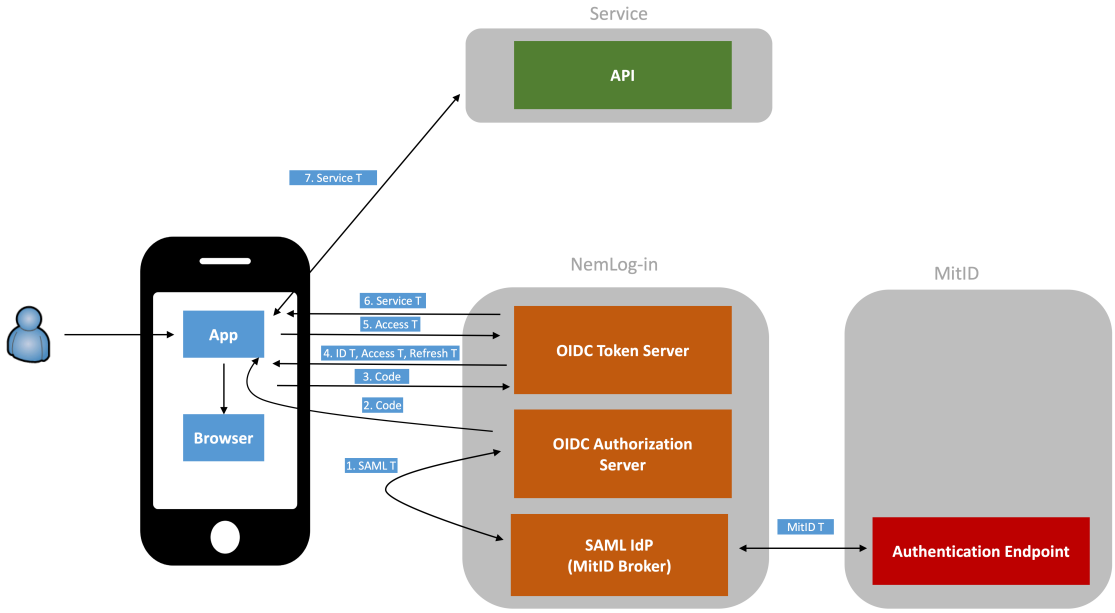
The table below shows important properties of the three client types which will be explained in subsequent chapters:

Client type	Native app	Web/JS app with a Backend	JS app without a Backend
Allowed flows	OAuth 2.0 Authorization code flow	OAuth 2.0 Authorization code flow	OAuth 2.0 Authorization code flow
PKCE	Mandatory	Mandatory	Mandatory
Token storage	Secure device storage	In backend or encrypted cookie in browser	Browser APIs
Authorization Server Requirements	Redirect URI registered and exact match required; no wildcards allowed	Redirect URI registered and exact match required; no wildcards allowed	CORS headers enabled; Redirect URI registered and exact match required; no wildcards allowed
Refresh Token Policy	Long-lived refresh tokens allowed (not expiring) if revocation mechanism implemented	Refresh tokens allowed up to 8 hrs; must be invalidated on logout	Refresh tokens up to 60 min and ONLY with rotation on each use.
Client Authentication	Not possible (public client); use re-direct URI as proof	Backend should register credential with Authorization Server (confidential client)	Not possible (public client); use re-direct URI as proof
Other security req			Limit Javascript execution to set of defined origins
Logout handling	No session (only refresh token revocation).	Backend should send/receive logout request via SAML or	App should poll Authorization Server via frame to detect session

		OIDC and invalidate session cookie and tokens.	termination and invalidate tokens.

3 Scenario overview

The figure below illustrates the main components and their interactions when a client being a native app is authorized by the user:



The main principles are:

- The end-user authenticates to a central Authorization Server via a separate user agent (e.g. browser); the Authorization Server may invoke an external SAML Identity Provider which provides authentication of the end-user (e.g. based on MitID as shown in the figure or something else).
- The Authorization server issues an authorization code (according to the OAuth authorization code flow).
- The Authorization code is first exchanged for a set of tokens: an ID Token to be consumed within the client App, an Access Token to be used with the Token Server and a Refresh Token also used with the Token Server.
- The client can subsequently exchange the Access Token for a Service Token with the Token Server– the Service Token is simply an Access Token for an external API protected by the infrastructure.
- Client access to external APIs is obtained by presenting a valid security token issued by a trusted Token Server. The security token provides the claims necessary for fulfilling the API's access control policy.

All tokens except the Refresh token are short-lived which ensures that they are renewed often and therefore get updated frequently. More details on this is described in chapter 6.

3.1 Overview of protocol flows

This specification defines a number of protocol flows where a client interacts with an external Authorization and Token Server using OAuth 2.0 and OpenID Connect. Requirements for the individual steps are profiled in order to ensure interoperability, narrow implementation choices, mandate best practice, and achieve a high security level and compatibility with existing models and infrastructure.

The main flows are:

- Client Authorization where the client is installed/loaded and authorized by the user, described in chapter 4.
- Issuance of an Access Token to the client providing access to an external SP API (STS flow), described in chapter 5.
- Renewal of expired Access Tokens is described in chapter 6. Here, also the timeout policies and revocation of tokens is described.
- Using an Access Token to get access to an SP API is described in chapter 7.

Note also that generic security requirements (e.g. for transport security) described in chapter 8 apply to all profiles.

3.2 Pre-requisites

A number of pre-requisites are assumed to be in place before the above flows can be executed:

- The end-user has obtained relevant credentials needed for authentication (e.g. NemID, MitID or credential from local IdP).
- In case the client is a native app, the end-user has a personal device[1]. Other flows should be used for non-personal devices where security tokens for a specific user cannot be persisted on the device. The native app has been installed on the end-user device (e.g. from a public app store or a closed app store).
- The client and SP API have been registered and configured with the central Authorization Server including relevant parameters / metadata:
    - Relevant identifiers (e.g. EntityIDs) have been assigned for client and API instances such that they can be referenced in tokens and protocol messages.
    - The type of client has been registered (see section 2.1 for details).
    - The client must have registered a unique **redirect URI** for returning the authorization response to the client, and the URI scheme should be based on a domain name that is under the control of the service provider of the client. More details are described in chapter 4.

[1] The profiles assume that security tokens can be persisted in Apps including a long-lived Refresh-token. The risk profile of the App and user terms may define whether it is acceptable to do this on a device shared in a family or shared among colleagues in workplace environment.

- Necessary privileges and scopes have been defined for the client and SP API such that they can be requested by the client and issued in tokens. See section 3.3 for details.
- Claims sets have been defined during registration. The OIO JWT profile [OIO JWT] defines mandatory claims but any optional claims needed by an app or API should be registered.
- Metadata and certificates have been exchanged in advance as part of trust establishment.
  - Native app clients have been configured with (pinned) TLS server certificates.
  - Clients have been configured with trusted token signing certificates, such token signatures can be securely validated.
  - Confidential clients have registered their secret or certificate for client authentication.
  - Clients have registered certificates if they support encrypted ID tokens.
  - SP APIs have registered certificates if they support encrypted Service Tokens.
- Web clients with a backend supporting HTTP-based (front-channel) logout must register a logout URI as part of the client registration process.

The actual registration process which establishes these pre-conditions will not be described in this document as it is highly implementation specific. For example, in NemLog-in the registration will likely be based on the existing administration portal.

3.3 Authorization models

This section describes the different authorization models. A central design goal has been to reuse the existing mechanisms for web applications and SOAP web services (in particular the OIO Basic Privilege Profile) such that API providers can reuse existing logic and access control policies. Other authorization models can be added later if needed.

3.3.1 Scopes granted by end-user

A fundamental design principle in OAuth (and hence this profile) is that the end-user should authorize access granted to the client by authenticating to the Authorization Server and providing explicit consent for the client to act on his behalf. This is accomplished by including a set of OAuth scopes in the authorization request from the client, which allows the Authorization Server to obtain the necessary consent from the end-user and reflect it in issued tokens. This consent both covers the authorization to obtain an OIDC ID Token as well as authorization to invoke external APIs (SP API).

Therefore, the initial scope values provided by the client in the authorization request (see next section) has to contain sufficient scopes to cover all APIs and scopes which the client needs to invoke on the end-user's behalf. If the client at a later stage (e.g. a

later version of an app) needs further access (new API or scope), a fresh authorization request is required with the additional scopes added – which the user can then consent to.

As specified in the JWT Token Profile [OIO JWT], Access Tokens for SP APIs will contain privileges according to the model defined in OIO Basic Privilege Profile. Privileges are URIs defined by a Service Provider representing a specific access with that Service Provider. Thus, the meaning, granularity and consent text of privileges is defined entirely by the Service Provider.

It is assumed that privileges to be requested and asserted in tokens will be registered in advance (by the Service Provider) with the Authorization and Token Server in as indicated in the example below:

Privilege info	**Example values of privilege metadata registered**
SP EntityID	https://ngdp.digst.dk
Privilege URI	https://ngdp.digst.dk/priv/read_mail
OAuth scope shorthand[2]	xq7j
Description	This privilege grants access to read mail from a citizen inbox in the NGdP solution.
UI Context text (DK)	"Vil du give samtykke til, at denne App tilgår din Digitale Post fra det offentlige?"

The Authorization Server registration process ensures uniqueness of EntityIDs, privileges URIs, scope shorthands etc. and ensures proper ownership (e.g. an admin can only administer relevant Apps and APIs).

Thus, if the client includes the `xq7j` shorthand in the scope parameter (see chapter 4), the Authorization Server will prompt the end-user for consent to authorize the client to access his mail in the NGdP solution, and if consent is granted, the client will subsequently be able to obtain an Access Token for the relevant SP API, where the associated privilege URI `https://ngdp.digst.dk/priv/read_mail` is included with scope of the citizen.

Below is given an example of the resulting JSON structure within the Access Token based on the OIO JWT Profile [OIO JWT], where the privilege is included with scope[3] of "1202801024" (CPR number of citizen):

[2] In order to keep requests small enough to fit in HTTP headers used with the OIDC authentication request, privileges are suggested to have a unique short-hand such that the entire URI is not necessary.
[3] Note that the scope in OIO Basic Privilege Profile should not be confused with scope in OAuth / OIDC. The first is the context of a privilege (e.g. person or organization the privilege applies to) and the latter corresponds to a given access requested (similar to a privilege in OIO BPP).

```
{
    "privilegegroups" : [
       {
           "privilege" : "https://ngdp.digst.dk/priv/read_mail",
           "scope" : "urn:dk:gov:saml:cprNumberIdentifier:1202801024"
       }
    ]
}
```

A similar model can be used for other scopes and for data restrictions; see the OIO JWT Profile [OIO JWT] for details.

3.3.2 Scopes granted by API provider (WSP)

In addition to scopes granted by the end-user, the Authorization Server may allow the API provider to grant privileges/scopes to certain client independent of the user.

This is similar to the current mechanism in the NemLog-in STS, where a WSP (Web Service Provider) can define a number of privileges, which can then be granted to certain Web Service Consumers – i.e. clients of the WSP. The assignment of privileges is done by the WSP administrator in the NemLog-in administration portal.

A similar model can be used with clients and APIs – acting as WSC and WSP respectively. It can be used to grant specific access only to certain clients.

This model requires that the Authorization Server is able to securely authenticate the client. Since a public native apps cannot have secrets embedded in their installation package (they would be trivial to extract by others), the authentication of the native apps must be performed using a claimed “https” scheme URI redirection described in [RFC8252]. This prevents other App instances from claiming URIs from domains they don’t control.

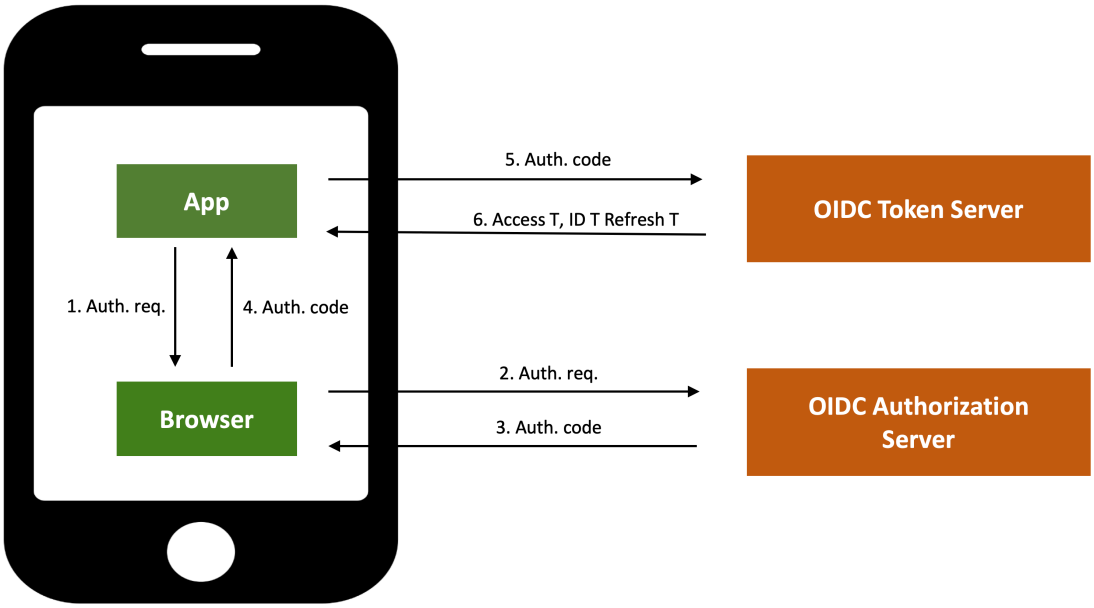
Privileges granted by the API provider will be represented in the same way as user-granted privileges except they have a different scope being the client ID instead of the end-user CPR-number:

```
{
    "privilegegroups" : [
       {
           "privilege" : "https://ngdp.digst.dk/priv/read_mail",
           "scope" : "https://digst.dk/ngdp/apps/borger_dk_client",
       }
    ]
}
```

4 Client Authorization Profile

This chapter specifies a profile of OpenID Connect (using the [OAuth] 2.0 authorization code grant) used for the initial authorization of the client. The profile is primarily based on [OIDC], [OAuth] and [RFC8252].

The figure below illustrates the main steps where the client is a native app – the same flow is used by all client types:



The main steps of the flow are:

1. The client opens/redirects a browser with an authorization[4] request.
2. The Authorization endpoint receives the authorization request, authenticates the user, and obtains end-user consent for the requested scopes. Authenticating the user may involve chaining to other authentication systems (e.g. an existing SAML IdP).
3. The Authorization server issues an authorization code to the redirect URI.
4. The client receives the authorization code from the redirect URI.
5. The client presents the authorization code at the token endpoint.
6. The token endpoint validates the authorization code and issues the tokens requested.
7. The client validates the response (not shown on figure above).

[OIDC-01]

The client authorization protocol MUST follow the [OAuth] 2.0 authorization code grant type as defined in section 4.1 of [OAuth]. Unless otherwise stated

[4] Note that in [OIDC] this is called an authentication request, whereas OAuth calls it an authorization request.

explicitly, the requirements from this specification apply directly.

4.1.1 Step 1: Client opens/redirects browser with authorization[5] request

[OIDC-02]

The request parameters in the authorization request MUST follow the requirements specified in the table below:

Parameter	Mandatory	Usage
scope	Y	MUST contain the 'openid' scope value as well as scopes for external APIs which the user should authorize. See section 3.3 for a description of the authorization model.
response_type	Y	MUST be set to 'code'
client_id	Y	MUST be set to the client identifier (Entity ID) pre-registered with the Authorization Server.
redirect-uri	Y	The client MUST use a claimed "https" scheme URI redirection when supported by the client platform6 (e.g. "https://app.example.com/oauth2redirect/example-provider") and the URI MUST NOT contain any wildcards. This ensures the identity of the destination client to the authorization server by the operating system. If the mechanism is not supported, the client SHOULD instead uses a "custom URL scheme" for URI redirection, and it MUST be URI scheme based on a domain name under control of the client developer as described in RFC7595. It is REQUIRED that a unique redirect URI is used for each authorization server used by the client.
state	Y	To mitigate CSRF-style attacks over inter-app URI communication channels (so called "cross-app request forgery"), it is REQUIRED that the client includes a high-entropy secure random number (>=128 bit) in the "state" parameter of the authorization request.
code_challenge	Y	The client MUST use the Proof Key for Code Exchange ([PKCE], RFC7636) extension to OAuth and include a code_challenge being a high-entropy cryptographic random STRING containing 128 characters. The code_challenge is a obtained as hash of the secret code_verifier: code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
code_ challenge_method	Y	MUST be 'S256' (see [PKCE]).
nonce	Y	MUST include a high-entropy secure random number (>=128 bit) in order to prevent ID token replay.

[5] Actually, it is both an authentication and authorization request.
[6] This is supported both on iOS and Android 6.0 and above.

amr_values	N	String that specifies the amr value that the Authorization Server is being requested to use for processing this Authentication Request. In this profile NSIS levels are used, and the minimum NSIS level is specified as one of the below NSIS levels: https://data.gov.dk/concept/core/nsis/loa/Low https://data.gov.dk/concept/core/nsis/loa/Substantial https://data.gov.dk/concept/core/nsis/loa/High The Authorization Server SHOULD ensure that the end-user is authenticated at least to the specified NSIS level.

Note: PKCE is a proof-of-possession extension to OAuth 2.0 that protects the authorization code from being used if it is intercepted. The extension has the client generate a secret verifier; it passes a hash of this verifier in the initial authorization request and must present the unhashed verifier when redeeming the authorization code. An attacker that intercepted the authorization code would not be in possession of this secret, rendering the code useless.

[OIDC-03]

The following request parameters SHOULD NOT be used with this profile: `display`, `acr_values`, `prompt`, `response_mode`, `max_age`[7], and `id_token_hint`[8].

Other request parameters defined in [OIDC] and [OAuth] and not mentioned here are all OPTIONAL.

[OIDC-04]

OAuth 2.0 authorization requests from the client MUST be sent through external user agents (i.e. not embedded web views in a native app). Otherwise, the native app may be able to copy user credentials and cookies. In-app browser tabs MAY be used if they separate security context from the native app.

Example authentication request:

The following is the non-normative example request that would be sent by the User Agent to the Authorization Server in response to the HTTP 302 redirect response by the client (with line wraps within values for display purposes only):

```
GET /authorize?
  response_type=code
  &scope=openid
  &client_id=https%3A%2F%2Fclient.example.org%2Fcb
```

[7] This parameter is not necessary since the profile requires 'fresh' user authentication.
[8] The `login_hint` parameter may possibly be used to facilitate MitID App switch. This has to be clarified further when MitID specifications are available.

```
    &state=af0ifjsldkj
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb HTTP/1.1
    &code_challenge=qjrzSW9gMiUgpUvqgEPE4_-8swvyCtfOVvg55o5S_es
    &code_challenge_method=S256

  Host: server.example.com
```

4.1.2 Step 2: Authorization endpoint receives authentication request

[OIDC-05]

The Authorization Server MUST validate the request as specified in section 3.1.2.2 (Authentication Request Validation) of [OIDC] including that all required parameters mentioned above (section 4.1.1) are present. Hence, the `scope` parameter MUST contain the `openid` scope value.

As specified in [OAuth], Authorization Servers SHOULD ignore unrecognized request parameters.

[OIDC-06]

If the client is a confidential client (e.g. a web application with a backend), it MUST be authenticated by the Authorization Server using the registered credential. Other (public) client types SHOULD NOT be authenticated.

[OIDC-07]

The Authorization Server MUST reject a `redirect_uri` in requests that doesn't *exactly* match the one that was previously registered.

Note: As mentioned under prerequisites, the client must register its complete redirect URI with the Authorization Server and it must be unique per Authorization server.

[OIDC-08]

The Authorization Server MUST record the [PKCE] challenge and method in the request and reject requests not containing these parameters.

[OIDC-09]

If the request is valid, the Authorization Server SHOULD authenticate the end-user at the NSIS level of assurance defined in the request[9]. If the client type is a native app, it MUST be a fresh authentication of the end-user (e.g. SSO is not permitted here).

[9] As mentioned previously, an external authentication server (e.g. SAML IdP) may be used for this purpose.

[OIDC-10]

After successful authentication of the end-user, the Authorization Server MUST obtain (and securely store) user consent to the scopes defined in the request.

The user SHOULD be able to decide/grant consent individually per scope in the request (such that it is not all or nothing).

4.1.3 Step 3: Authorization server issues an authorization code

[OIDC-11]

After successful authentication of the end-user and granted consent, the Authorization Server MUST issue an authorization code, and the Authorization Response MUST return the parameters defined in Section 4.1.2 of [OAuth] by adding them as query parameters to the `redirect_uri` specified in the Authorization Request using the `application/x-www-form-urlencoded` format.

For web clients, the Authorization Server MUST include a `session_state` parameter in order to enable session management (see chapter 6) as described in the OIDC Session Management Specification.

The following is a non-normative example successful response using this flow (with line wraps within values for display purposes only):

```
HTTP/1.1 302 Found
  Location: https://client.example.org/cb?
    code=SplxlOBeZQQYbYS6WxSbIA
    &state=af0ifjsldkj
    &nonce=a8jf0dfjslkai
    &session_state=aa2i4jslkdu
```

4.1.4 Step 4: Client receives the authorization code

[OIDC-12]

The client MUST validate the response according to [OAuth] especially Sections 4.1.2 and 10.12.

[OIDC-13]

The client MUST validate `state` and `nonce` in responses and MUST reject responses if they do not match a pending outgoing authorization request. The client MUST further compare the redirect URI in the response to the value

used in the authorization request (step 1) and MUST verify that the URI on which the authorization response was received exactly matches it.

4.1.5 Step 5: Client presents the authorization code at the token endpoint

[OIDC-14]

The client MUST send a token request to the token endpoint to obtain a token response as described in Section 3.2 of [OAuth], using the `grant_type` value `authorization_code`.

[OIDC-15]

The client MUST include the PKCE `code_verifier` secret matching the `code_challenge` sent in step 1.

The following is a non-normative example of a Token Request (with line wraps within values for display purposes only):

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded


grant_type=authorization_code&
code=SplxlOBeZQQYbYS6WxSbI&
redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb&
code_verifier=JDu29BkSH99283aS
client_id=https%3A%2F%2Fclient.example.org
```

4.1.6 Step 6: Token endpoint validates the authorization code and issues the tokens requested

[OIDC-16]

The Token endpoint MUST validate the Token Request as described in OpenID Connect Core section 3.1.3.2 including the presented authorization code, PKCE `code_verifier` and value of `redirect_uri` parameter.

[OIDC-17]

The Token endpoint MUST issue an ID Token, an Accces Token and MAY issue a Refresh Token according to section 3.1.3.3 of OpenID Connect Core.

[OIDC-18]

The ID Token MUST be a JWT token according to the OIO JWT Profile [OIO JWT] and include an `at_hash` claim. Further, the ID Token MUST be encrypted, if the client has registered a certificate with the Authorization Server for this purpose. The Access Token and Refresh Tokens are opaque and SHOULD include at least 128 bit of entropy.

4.1.7 Step 7: Client validates response

[OIDC-19]

The client MUST validate the response according to section 3.1.3.5 (Token Response Validation) of OIDC Core. In addition, the `at_hash` value MUST be validated as specified in section 3.1.3.6 of OIDC to ensure binding between Access Token and ID Token.

[OIDC-20]

The client MUST validate that the signature of the ID Token is valid, that is uses an allowed signing algorithm defined in the OIO JWT Profile [OIO JWT], and with a pinned[10] token signing certificate of the Authorization Server. It MUST also verify, that it is the audience (`aud` claim) of the token.

[OIDC-21]

The client MUST check that the resulting NSIS assurance level in the ID Token lives up to its requirements as well as other claims in the ID token required by the client.

The following is a non-normative example of a successful Token Response[11]:

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

  {
   "access_token": "SlAV32hkKG",
   "token_type": "Bearer",
   "refresh_token": "8xLOxBtZp8",
   "expires_in": 3600,
   "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzc
     yI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4Mjg5
     NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAibi0wUzZ
     fV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5Nz
     AKfQ.ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr_P6q
     Jp6IcmD3HP99Obi1PRs-cwh3LO-p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJ
     NqeGpe-gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7Tpd
```

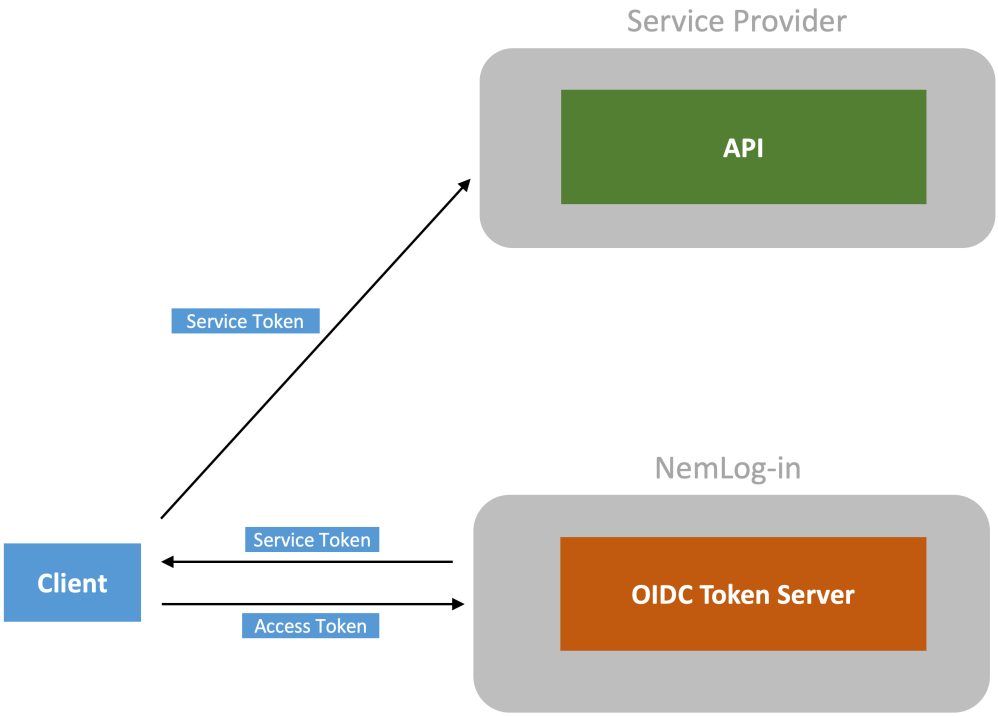
[10] E.g. part of the App configuration.
[11] Example from OpenID ID Connect Core – the shown id_token is not OIO JWT token compliant if decoded.

```
    QyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoS
    K5hoDalrcvRYLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4
    XUVrWOLrLl0nx7RkKU8NXNHq-rvKMzqg"
}
```

5 Token Request Profile

This chapter describes a token request profile building on the OAuth 2.0 client credentials grant flow. In the profile, the client requests a Service Token[12] (for a specific SP API and with scopes covering this API) and authorizes the request using an Access Token for the Token Service issued during the Client Authorization Profile described in chapter 4. If multiple APIs are to be accessed by the client, multiple tokens have to be requested.

The Token Server verifies that the presented Access Token is valid, and that the end-user has previously authorized the client instance (holding the Access Token) to use the requested scopes (via the consent gathered in step 3 of the Client Authorization Profile). If the request is successful, the Token Server issues a new token (Service Token) for the API according to the [OIO JWT] Profile, where the scopes are encoded as privileges in a JSON structure. See also section 3.3 for further detail.



5.1.1 Step 1: Client sends Token Request

[OIDC-51]

The token request MUST use the [OAuth] 2.0 client credentials grant type as defined in section 4.4 of [OAuth]. Unless otherwise stated explicitly, the requirements from the [OAuth] specification apply directly.

[12] The Service Token is an Access Token, but has a different name to distinguish it from the Access Token issued in the App Authorization Profile described in chapter 4.

[OIDC-52]

The request parameters in the token request MUST fulfill the requirements specified in the table below:

Parameter	Man-da-tory	Usage
client_id	Y	Value MUST be set to the client identifier (Entity ID) pre-registered with the Authorization Server.
sub	Y	Value MUST be the `sub` field (user identifier) from an ID Token issued in the App Authorization Flow described in chapter 4.
grant_type	Y	Value MUST be `client_credentials`
scope	Y	Value MUST contain list of scope values belonging to at most one external SP API registered with the Authorization Server.

[OIDC-53]

The token request MUST be authorized using an Access Token obtained in previous Client Authorization Flow and provided via the HTTP `Authorization` header using the `Bearer` authentication scheme defined in [RFC6750]. See chapter 4 for details.

A sample Token Request is shown below:

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Bearer czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

client_id=https%3A%2F%2Fclient.example.org%2Fcb
&sub=https%3A%2F%2Fdata.gov.dk%2Fmodel%2Fcore%2Feid%2Fperson%2Fuuid%2F
123e4567-e89b-12d3-66554400
&grant_type=client_credentials
&scope=xq7j%20uq2ja%20sdh34
```

5.1.2 Step 2: Token Server validates request and returns token

[OIDC-54]

The Token Server MUST verify that the Access Token is valid and is issued to a client instance with the stated `client_id` type.

[OIDC-55]

The Token Server MUST verify the request, including that requested `scope` values have been previously been consented by the end-user before the Access Token was issued. Further, is MUST be verified, that requested scopes only belong to one SP API.

[OIDC-56]

If the request is successful, the Token Server MUST issue a Service Token according to [OIO JWT] Profile with the requested scopes converted to privileges. This includes both user-granted scopes and Service Provider granted scopes (see section 3.3 for details.)

[OIDC-57]

The Service Token SHOULD have a validity period of maximum 1 hour.

[OIDC-58]

The Service Token MUST be encrypted if the SP API provider has registered a certificate for this purpose with the Authorization Server.

A sample Token Response is shown below[13] - note that the Service Token is returned in the `access_token` field according to OIDC conventions:

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

  {
   "token_type": "Bearer",
   "expires_in": 3600,
   "access_token":
   "eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzc
     yI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4Mjg5
     NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAibi0wUzZ
     fV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5Nz
     AKfQ.ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr_P6q
     Jp6IcmD3HP99Obi1PRs-cwh3LO-p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJ
     NqeGpe-gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7Tpd
     QyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoS
     K5hoDalrcvRYLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4
     XUVrWOLrLl0nx7RkKU8NXNHq-rvKMzqg"
  }
```

[13] Example from Open ID Connect – the shown Access Token is not OIO JWT token compliant if decoded.

6 Token renewal and session management

Refresh Tokens are credentials used to obtain Access Tokens. Refresh tokens are issued to the client by the Authorization Server and are used to obtain a new Access Token when the current Access Token becomes invalid or expires.

The profiles described in this document rely on the principle that issued Access and Service Tokens are relatively short-lived (e.g. one hour or less) such that have to be refreshed often. This approach has several benefits:

- A short validity period reduces attack windows.
- Continuously refreshing tokens means that their content can/will be updated – e.g. if the user has withdrawn their consent to an app or if the app has been revoked.
- Recipients of Access Tokens (e.g. API providers) are not burdened with having to check for token revocation by calling external services.
- Token revocation functionality can be focused on Refresh Tokens, which have a potentially longer validity period. It can be handled internally in the Authorization and Token Servers.

When all Access Tokens and the Refresh Token have expired, the client has to obtain new tokens using the Client Authorization Profile described in chapter 4. User interaction can be avoided as long as the Refresh Token remains valid.

6.1 Using a Refresh Token

A client with a valid Refresh Token can use it to obtain a new Access Token for the Token Server, and then use this Access Token to obtain new Service Tokens for SP APIs using the profiles described previously.

Refresh tokens are used with the `refresh_token` grant type as described in section 12 of [OIDC]; below is shown an example:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

client_id=s6BhdRkqt3
&grant_type=refresh_token
&refresh_token=8xLOxBtZp8
&scope=openid%20profile
```

6.2 Token revocation

[OIDC-61]

Authorization Servers that issue long-lived (>=8 hours) Refresh Tokens MUST provide a mechanism to revoke those tokens including a token revocation endpoint compliant with [RFC7009]. Thus, tokens can be revoked by making an HTTP POST request to the token endpoint URL as specified in this RFC.

This profile does not define the specific circumstances or policies where Refresh Tokens have to be revoked – only the capability to revoke them is required. Examples of circumstances that could lead to revocation in specific implementations are:

- A user interface could be provided to the end-users allowing the them to revoke Refresh tokens for individual App instances or all instances running on a particular device (e.g. when the user has lost his/her device). This might also be used by support personnel in case the end-user has temporarily lost their ability to authenticate (e.g. because their credential is on the same device).
- Refresh Tokens that are not frequently used could be set to be revoked automatically (e.g. 3 months of inactivity).
- Client being native apps can be programmed to request revocation when certain criteria are met on the end-user device – for example if the user has changed biometry on their phone, if a wrong user-pin is entered a certain number of times, or an indication of compromise is detected.
- External events (e.g. the user revoking their user credential such as NemID or MitID) could be configured to automatically revoke Refresh Tokens enrolled and authorized with the credential. Other events could be that the user identity is revoked (e.g. an employee identity no longer being associated with a company), or an API provider revoking all access to a client.

Revocation policies are internal to the implementation of the Authorization Server and do not affect the wire protocol; they are therefore left to implementations to decide.

6.3 Refresh token rotation

[OIDC-62]

For clients that are Javascript applications with no backend (i.e. SPAs), the Authorization Server MUST rotate Refresh Tokens on each use and ensure mechanisms to detect token replay (as described in [OSBP] section 4.12). Further, the lifetime of the new refresh token MUST NOT extend the lifetime of the initial refresh token.

6.4 Token lifetime

[OIDC-63]

The lifetime of tokens SHOULD NOT exceed the values in the table below:

Client type	Native app	Web/JS app with a Backend	JS app without a Backend
Refresh Token	Long-lived refresh tokens allowed (not expiring) if revocation mechanism implemented	8 hours	1 hour (only with rotation)
ID Token	1 hour	1 hour	1 hour
Service Token	1 hour	1 hour	1 hour
Access Token	1 hour	1 hour	1 hour

6.5 Session management

Clients that are native apps are not considered to have a (web) session based on the initial end-user authentication. For web clients, the session management requirements are stated below based on the client type.

6.5.1 Session management for web apps with backend

[OIDC-64]

Web clients with a backend MUST be able to receive logout events from the Authorization Server using OIDC Front Channel Logout or OIDC Back Channel Logout.

When terminating a session, in response to a logout request, both session cookies and tokens MUST be discarded by the app backend.

[OIDC-65]

The Authorization Server MUST propagate logout events to/from any federated authentication server (i.e. SAML IdP) used to authenticate the end-user in addition to own relying parties involved in current session.

6.5.2 Session management for web apps without backend

[OIDC-66]

Web clients with no backend (i.e. SPAs) MUST continuously[14] poll the Authorization server for changes to the user session via the hidden iframe mechanism defined in the OIDC Session Management Specification.

When a session change is detected, all tokens and HTML5 local storage MUST be discarded by the app.

[14] Every 10 seconds or less.

7 API Access Profile

This chapter describes how a client can invoke an external SP API using a Service Token obtained via the mechanisms described in the Token Request profile in chapter 5.

Note that this profile is currently restricted to 'bearer' tokens which is sufficient for most scenarios in combination with strong transport security (see chapter 8) and short token lifetime. This means a security token with the property that any party in possession of the token (a "bearer") can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material (proof-of-possession).

Other mechanisms may be considered during the further development of the profile including:

- HTTP Mac Authentication (https://tools.ietf.org/id/draft-ietf-oauth-v2-http-mac-00.html)
- Holder-of-key tokens as defined in the OIO IDWS REST Profile 1.0 (https://digitaliser.dk/resource/526486).
- Using the "OpenID Connect Token Bound authentication" draft specification, where a hash of the token binding ID from the TLS sessions is included in the token's confirmation claim (cnf). This mechanism is based on TLS extensions which may be problematic to support in Apps where the TLS stack is fixed.

However, none of the above mechanisms seem to be widely implemented and requiring their use in this profile could severely hurt interoperability and use of standard client libraries. Further, they require the establishment of either a shared symmetric key between the client and the server, a public/private keypar or use TLS client certificates which can be problematic with public clients.

[OIDC-71]

The client MUST pass the Service Token[15] in an `Authorization` HTTP header with token type `Bearer` as described in [RFC6750].

Example[16]:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer 7Fjfp0ZBr1H8JgaJs97Jb.8shJgaJs97Jb.asd&DSasdaJs97Jb
```

[15] The Service Token is just an Access Token in OAuth sense.

[16] Note that the token in the example above is not a real token and cannot be decoded.

[OIDC-72]

The SP API MUST validate the received Service Token including (as a minimum) that it is not expired, that it is signed by a trusted Token Server using an allowed algorithm, that the SP API is the intended audience of the token (`aud` field), and that required privileges are included (`priv` field). See the [OIO JWT] profile for details.

[OIDC-73]

The SP API MUST validate that NSIS assurance level (for the end-user authentication) asserted in the Access Token (`amr` field) is sufficient according to local access policy. The SP API MAY also consider the authentication time of the end-user, (`auth_time` field) before access is granted.

8 Security Requirements

The security requirements below apply to all protocol profiles in this document.

[OIDC-81]

All transport communication between the client and the authorization infrastructure MUST use TLS 1.2 or higher and SHOULD only use cipher suites supporting perfect forward secrecy. Servers MUST reject negotiation of insecure TLS connections. The document [NIST 800-52] (section "Minimum Requirements for TLS Servers") or subsequent revision may serve as reference for an acceptable level of transport security.

[OIDC-82]

Native app clients MUST pin server TLS certificates (i.e. maintain a list of trusted TLS server certificates as part of their configuration).

[OIDC-83]

JWT Access Tokens and Service Tokens MUST follow the OIO JWT Profile [OIO JWT].

[OIDC-84]

Implementations MUST follow requirements in [RFC8252].

[OIDC-85]

Client being native apps or Javascript applications without a backend MUST be treated as public native clients and MUST NOT have any secrets embedded in their script or installation package.

9 References

- [JWA] Jones, M., "JSON Web Algorithms (JWA)," draft-ietf-jose-json-web-algorithms (work in progress), July 2014.
- [JWE] Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)," draft-ietf-jose-json-web-encryption (work in progress).
- [JWK] Jones, M., "JSON Web Key (JWK)," draft-ietf-jose-json-web-key (work in progress), July 2014.
- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)," draft-ietf-jose-json-web-signature (work in progress), July 2014.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)," draft-ietf-oauth-json-web-token (work in progress), July 2014.
- [BBA] Parecki, Waite: "OAuth 2.0 for Browser-Based Apps", IETF draft.
- [NSIS] "National Standard for Identiteters Sikringsniveauer 2.0.1". https://digst.dk/it-loesninger/nemlog-in/det-kommende-nemlog-in/vejledninger-og-standarder/nsis-standarden/
- [OIOSAML] "OIOSAML Web SSO Profile 3.0". https://digst.dk/it-loesninger/nemlog-in/det-kommende-nemlog-in/vejledninger-og-standarder/oiosaml-30/
- [OIO-BPP] "OIO Basic Privilege Profile 1.1". https://www.digitaliser.dk/resource/2377872
- [RFC6819] "OAuth 2.0 Threat Model and Security Considerations", IETF. https://tools.ietf.org/html/rfc6819
- [RFC8252] "OAuth 2.0 for Native apps", IETF.
- [RFC6750] "The OAuth 2.0 Authorization Framework: Bearer Token Usage", IETF, https://tools.ietf.org/html/rfc6750
- [RFC7009] "OAuth 2.0 Token Revocation", IETF.
- [OIDC] "OpenID Connect Core 1.0 incorporating errata set 1, November 2014", OpenID.Net.
- [OAuth] "The OAuth 2.0 Authorization Framework", RFC6749, IETF, October 2012.
- [OIO JWT] "OIO JWT Token Profile", Danish Agency for Digitisation.
- [NSIS] "National Standard for Identiteters Sikringsniveauer 2.0.1", Digitaliseringsstyrelsen. https://digst.dk/it-loesninger/nemlog-in/det-kommende-nemlog-in/vejledninger-og-standarder/nsis-standarden/
- [OSBP] "OAuth 2.0 Security Best Current Practice", IEFT. https://data-tracker.ietf.org/doc/draft-ietf-oauth-security-topics/