Workgroup: OpenID Digital Credentials Protocols
Published: 9 July 2025
Status: Final
Authors: O. Terbu *MATTR*, T. Lodderstedt *SPRIND*, K. Yasuda *SPRIND*, D. Fett *Authlete*, J. Heenan *Authlete*

OpenID for Verifiable Presentations 1.0

Abstract

This specification defines a protocol for requesting and presenting Credentials.

Table of Contents

1. Introduction

This specification defines a mechanism on top of OAuth 2.0 [RFC6749] for requesting and delivering Presentations of Credentials. Credentials and Presentations can be of any format, including, but not limited to W3C Verifiable Credentials Data Model [VC_DATA], ISO mdoc [ISO.18013-5], and IETF SD-JWT VC [I-D.ietf-oauth-sd-jwt-vc].

OAuth 2.0 [RFC6749] is used as a base protocol as it provides the required rails to build a simple, secure, and developer-friendly Credential presentation layer on top of it. Moreover, implementers can, in a single interface, support Credential presentation and the issuance of Access Tokens for access to APIs based on Credentials in the Wallet. OpenID Connect [OpenID.Core] deployments can also extend their implementations using this specification with the ability to transport Credential Presentations.

This specification can also be combined with [SIOPv2], if implementers require OpenID Connect features, such as the issuance of Self-Issued ID Tokens [SIOPv2].

Additionally, it defines how to use OpenID4VP in conjunction with the Digital Credentials API (DC API) [W3C.Digital_Credentials_API]. See section Appendix A for all requirements applicable to implementers of OpenID4VP over the DC API. Except where it explicitly references other sections of this specification, that section is self-contained, and its implementers can ignore the rest of the specification.

1.1. Additional Authors

- Tobias Looker (MATTR)
- Adam Lemmon (MATTR)

1.2. Errata revisions

The latest revision of this specification, incorporating any errata updates, is published at openid-4-verifiable-presentations-1_0. The text of the final specification as approved will always be available at openid-4-verifiable-presentations-1_0-final. When referring to this specification from other documents, it is recommended to

reference openid-4-verifiable-presentations-1_0.

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

2. Terminology

This specification uses the terms "Access Token", "Authorization Request", "Authorization Response", "Client", "Client Authentication", "Client Identifier", "Grant Type", "Response Type", "Token Request" and "Token Response" defined by OAuth 2.0 [RFC6749], the terms "End-User" and "Entity" as defined by OpenID Connect Core [OpenID.Core], the terms "Request Object" and "Request URI" as defined by [RFC9101], the term "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [RFC7519], the term "JOSE Header" defined by JSON Web Signature (JWS) [RFC7515], the term "JSON Web Encryption (JWE)" defined by [RFC7516], and the term "Response Mode" defined by OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses].

Base64url-encoded denotes the URL-safe base64 encoding without padding defined in Section 2 of [RFC7515].

This specification also defines the following terms. In the case where a term has a definition that differs, the definition below is authoritative.

Biometrics-based Holder Binding: Ability of the Holder to prove legitimate possession of a Credential by demonstrating a certain biometric trait, such as a fingerprint or face. One example of a Credential with biometric Holder Binding is a mobile driving license [ISO.18013-5], which contains a portrait of the Holder.

Claims-based Holder Binding: Ability of the Holder to prove legitimate possession of a Credential by proving certain claims, e.g., name and date of birth, for example by presenting another Credential. Claims-based Holder Binding allows long-term, cross-device use of a Credential as it does not depend on cryptographic key material stored on a certain device. One example of such a Credential could be a diploma.

Credential: A set of one or more claims about a subject made by a Credential Issuer. In this specification, Credentials are usually Verifiable Credentials (defined below). Note that the definition of the term "Credential" in this specification is different from that in [OpenID.Core].

Credential Format Identifier: An identifier to denote a specific Credential Format in the context of this specification. This identifier implies the use of parameters specific to the respective Credential Format.

Credential Issuer: An entity that issues Credentials. Also called Issuer.

Cryptographic Holder Binding: Ability of the Holder to prove legitimate possession of a Credential by proving control over the same private key during the issuance and presentation. Mechanism might depend on the Credential Format. For example, in jwt_vc_json Credential Format, a Credential with Cryptographic Holder Binding contains a public key or a reference to a public key that matches to the private key controlled by the Holder.

Digital Credentials API: The Digital Credentials API (DC API) refers to the W3C Digital Credentials API [W3C.Digital_Credentials_API] on the Web Platform and its equivalent native APIs on App Platforms (such as Credential Manager on Android).

Holder: An entity that receives Credentials and has control over them to present them to the Verifiers as Presentations.

Holder Binding or Key Binding: Ability of the Holder to prove legitimate possession of a Credential.

Issuer-Holder-Verifier Model: A model for exchanging claims, where claims are issued in the form of Credentials independent of the process of presenting them as Presentations to the Verifiers. An issued Credential may be used multiple times.

Origin: An identifier for the calling website or native application, asserted by the web or app platform. A web origin is the combination of a scheme/protocol, host, and port, with port being omitted when it matches the default port of the scheme. An app platform may use a linked web origin, or use a platform-specific URI for the

app origin. For example, the Verifier for the organization MyExampleOrg is served from https://verify.example.com. The web origin is `https://verify.example.com` with `https` being the scheme, `verify.example.com` being the host, and the port is not explicitly included as 443 is the default port for the protocol `https`. The native applications origin on some platforms will also be `https://verify.example.com` and on other platforms, may be `platform:pkg-key-hash:Z4OFzVVSZrzTRa3eg79hUuHy12MVW0vzPDf4q4zaPs0`.

Presentation: Data that is presented to a specific Verifier, derived from a Credential. In this specification, Presentations are usually Verifiable Presentations including Holder Binding (as defined below), but may also be Presentations without Holder Binding (discussed in Section 5.3).

VP Token: An artifact containing one or more Presentations returned as a response to an Authorization Request. The structure of VP Tokens is defined in Section 8.1.

Verifier: An entity that requests, receives, and validates Presentations. The Verifier is a specific case of an OAuth 2.0 Client, just like a Relying Party (RP) in [OpenID.Core].

Verifiable Credential (VC): An Issuer-signed Credential whose authenticity can be cryptographically verified. Can be of any format used in the Issuer-Holder-Verifier Model, including, but not limited to those defined in [VC_DATA] (VCDM), [ISO.18013-5] (mdoc), and [I-D.ietf-oauth-sd-jwt-vc] (SD-JWT VC).

Verifiable Presentation (VP): A Presentation with a cryptographic proof of Holder Binding. Can be of any format used in the Issuer-Holder-Verifier Model, including, but not limited to those defined in [VC_DATA] (VCDM), [ISO.18013-5] (mdoc), and [I-D.ietf-oauth-sd-jwt-vc] (SD-JWT VC).

Wallet: An entity used by the Holder to receive, store, present, and manage Credentials and key material. There is no single deployment model of a Wallet: Credentials and keys can both be stored/managed locally, or by using a remote self-hosted service, or a remote third-party service.

3. Overview

This specification defines a mechanism to request and present Credentials. The baseline of the protocol uses HTTPS messages and redirects as defined in OAuth 2.0. Additionally, the specification defines a separate mechanism where OpenID4VP messages are sent and received over the Digital Credentials API (DC API) [W3C.Digital_Credentials_API] instead of HTTPS messages and redirects.

As the primary extension, OpenID for Verifiable Presentations introduces the new response type `vp_token`, which allows a Verifier to request and receive Verifiable Presentations and Presentations in a container designated as VP Token. A VP Token contains one or more Verifiable Presentations and/or Presentations in the same or different Credential formats. Consequently, the result of an OpenID4VP interaction is one or more Verifiable Presentations and/or Presentations instead of an Access Token.

This specification supports any Credential format used in the Issuer-Holder-Verifier Model, including, but not limited to those defined in [VC_DATA] (VCDM), [ISO.18013-5] (mdoc), and [I-D.ietf-oauth-sd-jwt-vc] (SD-JWT VC). Credentials of multiple formats can be presented in the same transaction. The examples given in the main part of this specification use W3C Verifiable Credentials, while examples in other Credential formats are given in Appendix B.

OpenID for Verifiable Presentations supports scenarios where the Authorization Request is sent both when the Verifier is interacting with the End-User using the device that is the same or different from the device on which requested Credential(s) are stored.

This specification supports the response being sent using a redirect but also using an HTTP POST request. This enables the response to be sent across devices, or when the response size exceeds the redirect URL character size limitation.

In summary, OpenID for Verifiable Presentations is a framework that requires profiling to achieve interoperability. Profiling means defining:

- what optional features are used or mandatory to implement, e.g., response encryption;
- which values are permitted for parameters, e.g., Credential Format Identifiers;
- optionally, extensions for new features.

3.1. Same Device Flow

Figure 1 is a diagram of a flow where the End-User presents a Credential to a Verifier interacting with the End-User on the same device that the device the Wallet resides on.

The flow utilizes simple redirects to pass Authorization Request and Response between the Verifier and the Wallet. The Presentations are returned to the Verifier in the fragment part of the redirect URI, when Response Mode is `fragment`.

Note: The diagram does not illustrate all the optional features of this specification.

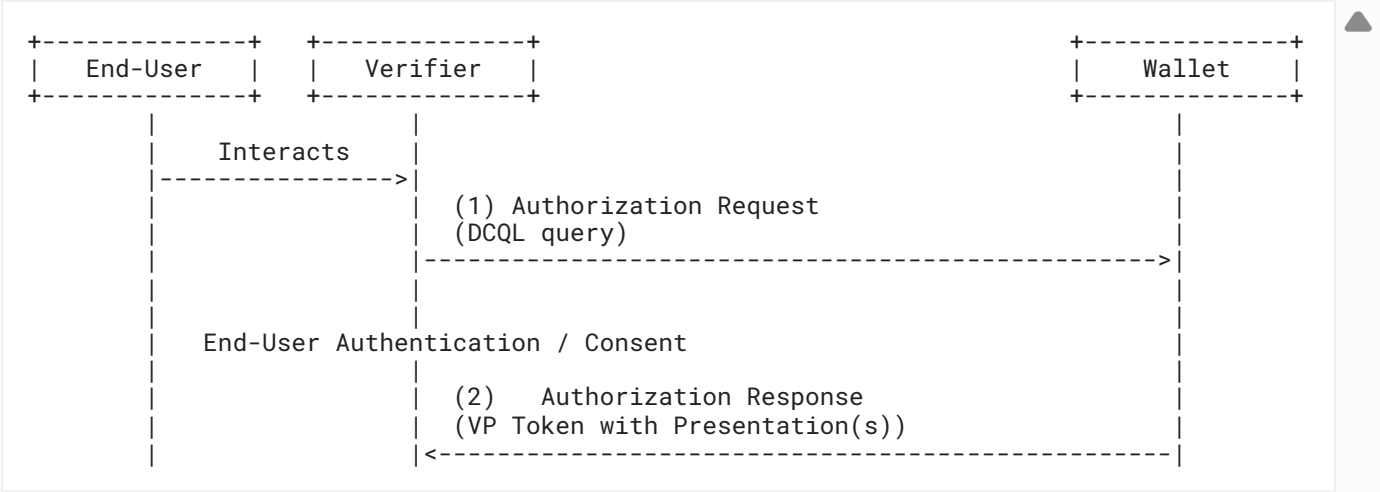

Figure 1: Same Device Flow

(1) The Verifier sends an Authorization Request to the Wallet. It contains a Digital Credentials Query Language (DCQL, see Section 6) query that describes the requirements of the Credential(s) that the Verifier is requesting to be presented. Such requirements could include what type of Credential(s), in what format(s), which individual Claims within those Credential(s) (Selective Disclosure), etc. The Wallet processes the Authorization Request and determines what Credentials are available matching the Verifier's request. The Wallet also authenticates the End-User and gathers consent to present the requested Credentials.

(2) The Wallet prepares the Presentation(s) of the Credential(s) that the End-User has consented to. It then sends to the Verifier an Authorization Response where the Presentation(s) are contained in the `vp_token` parameter.

3.2. Cross Device Flow

Figure 2 is a diagram of a flow where the End-User presents a Credential to a Verifier interacting with the End-User on a different device as the device the Wallet resides on.

In this flow, the Verifier prepares an Authorization Request and renders it as a QR Code. The End-User then uses the Wallet to scan the QR Code. The Presentations are sent to the Verifier in a direct HTTP POST request to a URL controlled by the Verifier. The flow uses the Response Type `vp_token` in conjunction with the Response Mode

`direct_post`, both defined in this specification. In order to keep the size of the QR Code small and be able to sign and optionally encrypt the Request Object, the actual Authorization Request contains only the Client Identifier and Request URI (as required by [RFC9101]), which the Wallet uses to retrieve the actual Authorization Request data.

Note: The diagram illustrates neither all parameters nor all optional features of this specification.

Note: The usage of the Request URI as defined in [RFC9101] does not depend on any other choices made in the protocol extensibility points, i.e., it can be used in the Same Device Flow, too.

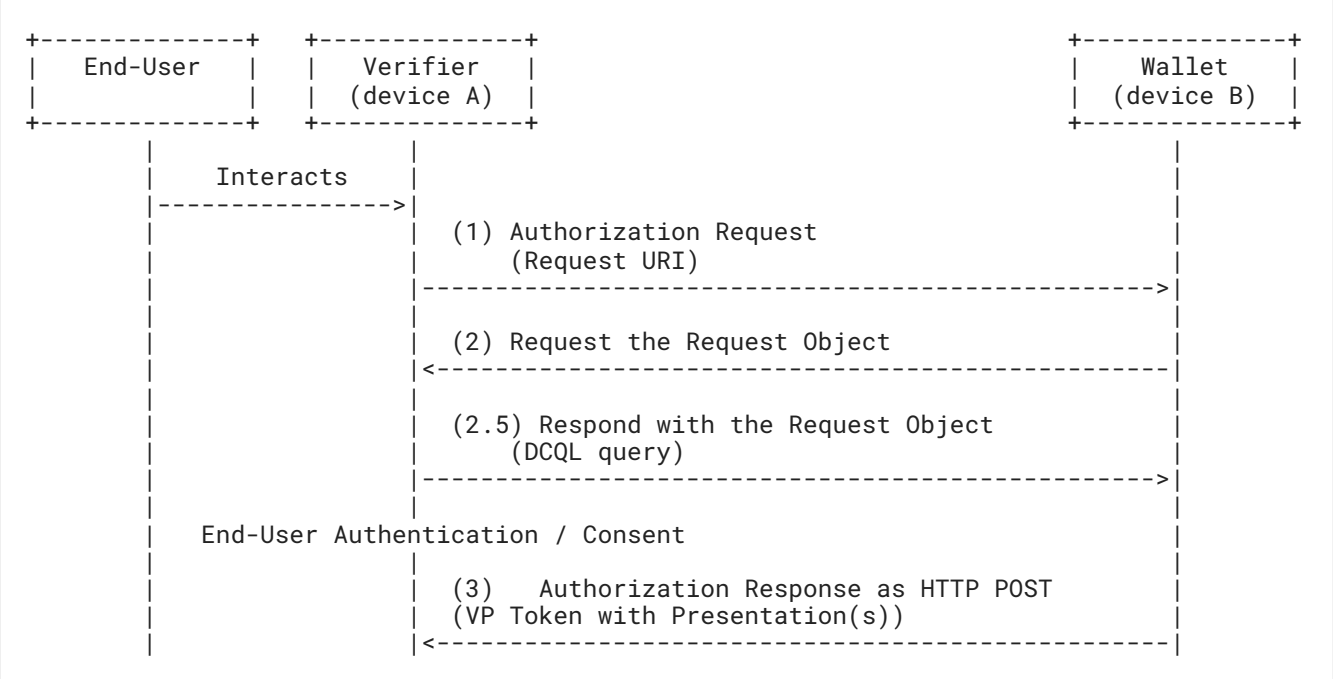

Figure 2: Cross Device Flow

(1) The Verifier sends to the Wallet an Authorization Request that contains a Request URI from where to obtain the Request Object containing Authorization Request parameters.

(2) The Wallet sends an HTTP GET request to the Request URI to retrieve the Request Object.

(2.5) The HTTP GET response returns the Request Object containing Authorization Request parameters. It contains a DCQL query that describes the requirements of the Credential(s) that the Verifier is requesting to be presented. Such requirements could include what type of Credential(s), in what format(s), which individual Claims within those Credential(s) (Selective Disclosure), etc. The Wallet processes the Request Object and determines what Credentials are available matching the Verifier's request. The Wallet also authenticates the End-User and gathers their consent to present the requested Credentials.

(3) The Wallet prepares the Presentation(s) of the Credential(s) that the End-User has consented to. It then sends to the Verifier an Authorization Response where the Presentation(s) are contained in the `vp_token` parameter.

4. Scope

OpenID for Verifiable Presentations extends existing OAuth 2.0 mechanisms in the following ways:

- A new query language, the Digital Credentials Query Language (DCQL), is defined to enable requesting Presentations in an easier and more flexible way. See Section 6 for more details.

- A new `dcql_query` Authorization Request parameter is defined to request Presentation of Credentials in the JSON-encoded DCQL format. See Section 5 for more details.
- A new `vp_token` response parameter is defined to return Presentations with or without Holder Binding to the Verifier in either Authorization or Token Response depending on the Response Type. See Section 8 for more details.
- New Response Types `vp_token` and `vp_token id_token` are defined to request Credentials to be returned in the Authorization Response (standalone or along with a Self-Issued ID Token [SIOPv2]). See Section 8 for more details.
- A new OAuth 2.0 Response Mode `direct_post` is defined to support sending the response across devices, or when the size of the response exceeds the redirect URL character size limitation. See Section 8.2 for more details.
- The `format` parameter is used throughout the protocol in order to enable customization according to the specific needs of a particular Credential format. Examples in Appendix B are given for Credential formats as specified in [VC_DATA], [ISO.18013-5], and [I-D.ietf-oauth-sd-jwt-vc].
- The concept of a Client Identifier Prefix to enable deployments of this specification to use different mechanisms to obtain and validate metadata of the Verifier beyond the scope of [RFC6749].
- A mechanism specifying the use of OpenID4VP with the Digital Credentials API (see Appendix A).

Presentation of Credentials using OpenID for Verifiable Presentations can be combined with the End-User authentication using [SIOPv2], and the issuance of OAuth 2.0 Access Tokens.

5. Authorization Request

The Authorization Request follows the definition given in [RFC6749] taking into account the recommendations given in [RFC9700] where applicable.

The Verifier MAY send an Authorization Request as a Request Object either by value or by reference, as defined in the JWT-Secured Authorization Request (JAR) [RFC9101]. Verifiers MUST include the `typ` Header Parameter in Request Objects with the value `oauth-authz-req+jwt`, as defined in [RFC9101]. Wallets MUST NOT process Request Objects where the `typ` Header Parameter is not present or does not have the value `oauth-authz-req+jwt`.

The `client_id` claim is required as defined below and would be redundant with a possible `iss` claim in the Request Object which is commonly used in JAR. To avoid breaking existing JAR implementations, the `iss` claim MAY be present in the Request Object. However, if it is present, the Wallet MUST ignore it.

This specification defines a new mechanism for the cases when the Wallet wants to provide to the Verifier details about its technical capabilities to allow the Verifier to generate a request that matches the technical capabilities of that Wallet. To enable this, the Authorization Request can contain a `request_uri_method` parameter with the value `post` that signals to the Wallet that it can make an HTTP POST request to the Verifier's `request_uri` endpoint with information about its capabilities as defined in Section 5.10. The Wallet MAY continue with JAR when it receives `request_uri_method` parameter with the value `post` but does not support this feature.

The Verifier articulates requirements of the Credential(s) that are requested using the `dcql_query` parameter. Wallet implementations MUST process the DCQL query and select candidate Credential(s) using the evaluation process described in Section 6.4

The Verifier communicates a Client Identifier Prefix that indicates how the Wallet is supposed to interpret the Client Identifier and associated data in the process of Client identification, authentication, and authorization as a prefix in the `client_id` parameter. This enables deployments of this specification to use different mechanisms to

obtain and validate Client metadata beyond the scope of [RFC6749]. A certain Client Identifier Prefix sets the requirements whether the Verifier needs to sign the Authorization Request as a means of authentication and/or pass additional parameters and require the Wallet to process them.

Depending on the Client Identifier Prefix, the Verifier can communicate a JSON object with its metadata using the `client_metadata` parameter which contains name/value pairs.

Additional request parameters, other than those defined in this section, MAY be defined and used, as described in [RFC6749]. The Wallet MUST ignore any unrecognized parameters, other than the `transaction_data` parameter. One exception to this rule is the `transaction_data` parameter. Wallets that do not support this parameter MUST reject requests that contain it.

5.1. New Parameters

This specification defines the following new request parameters:

`dcql_query`: A JSON object containing a DCQL query as defined in Section 6.

Either a `dcql_query` or a `scope` parameter representing a DCQL Query MUST be present in the Authorization Request, but not both.

In the context of an authorization request according to [RFC6749], parameters containing objects are transferred as JSON-serialized strings (using the application/x-www-form-urlencoded format as usual for request parameters).

`client_metadata`: OPTIONAL. A JSON object containing the Verifier metadata values. It MUST be UTF-8 encoded. The following metadata parameters MAY be used:

- `jwks`: OPTIONAL. A JSON Web Key Set, as defined in [RFC7591], that contains one or more public keys, such as those used by the Wallet as an input to a key agreement that may be used for encryption of the Authorization Response (see Section 8.3), or where the Wallet will require the public key of the Verifier to generate a Verifiable Presentation. This allows the Verifier to pass ephemeral keys specific to this Authorization Request. Public keys included in this parameter MUST NOT be used to verify the signature of signed Authorization Requests. Each JWK in the set MUST have a `kid` (Key ID) parameter that uniquely identifies the key within the context of the request.
- `encrypted_response_enc_values_supported`: OPTIONAL. Non-empty array of strings, where each string is a JWE [RFC7516] `enc` algorithm that can be used as the content encryption algorithm for encrypting the Response. When a `response_mode` requiring encryption of the Response (such as `dc_api.jwt` or `direct_post.jwt`) is specified, this MUST be present for anything other than the default single value of `A128GCM`. Otherwise, this SHOULD be absent.
- `vp_formats_supported`: REQUIRED when not available to the Wallet via another mechanism. As defined in Section 11.1.

Authoritative data the Wallet is able to obtain about the Client from other sources, for example those from an OpenID Federation Entity Statement, take precedence over the values passed in `client_metadata`.

Other metadata parameters MUST be ignored unless a profile of this specification explicitly defines them as usable in the `client_metadata` parameter.

`request_uri_method`: OPTIONAL. A string determining the HTTP method to be used when the `request_uri` parameter is included in the same request. Two case-sensitive valid values are defined in this specification: `get` and `post`. If `request_uri_method` value is `get`, the Wallet MUST send the request to retrieve the Request Object using the HTTP GET method, i.e., as defined in [RFC9101]. If `request_uri_method` value is `post`, a supporting Wallet MUST send the request using the HTTP POST method as detailed in Section 5.10. If the

`request_uri_method` parameter is not present, the Wallet MUST process the `request_uri` parameter as defined in [RFC9101]. Wallets not supporting the `post` method will send a GET request to the Request URI (default behavior as defined in [RFC9101]). `request_uri_method` parameter MUST NOT be present if a `request_uri` parameter is not present.

If the Verifier set the `request_uri_method` parameter value to `post` and there is no other means to convey its capabilities to the Wallet, it SHOULD add the `client_metadata` parameter to the Authorization Request. This enables the Wallet to assess the Verifier's capabilities, allowing it to transmit only the relevant capabilities through the `wallet_metadata` parameter in the Request URI POST request.

`transaction_data`: OPTIONAL. Non-empty array of strings, where each string is a base64url-encoded JSON object that contains a typed parameter set with details about the transaction that the Verifier is requesting the End-User to authorize. See Section 8.4 for details. The Wallet MUST return an error if a request contains even one unrecognized transaction data type or transaction data not conforming to the respective type definition. In addition to the parameters determined by the type of transaction data, each `transaction_data` object consists of the following parameters defined by this specification:

- `type`: REQUIRED. String that identifies the type of transaction data. This value determines parameters that can be included in the `transaction_data` object. The specific values are out of scope for this specification. It is RECOMMENDED to use collision-resistant names for `type` values.
- `credential_ids`: REQUIRED. Non-empty array of strings each referencing a Credential requested by the Verifier that can be used to authorize this transaction. The string matches the `id` field in the DCQL Credential Query. If there is more than one element in the array, the Wallet MUST use only one of the referenced Credentials for transaction authorization.

Each document specifying details of a transaction data type defines what Credential(s) can be used to authorize those transactions. Those Credential(s) can be issued specifically for the transaction authorization use case or re-use existing Credential(s) used for user identification. A mechanism for Credential Issuers to express that a particular Credential can be used for authorization of transaction data is out of scope for this specification.

The following is a non-normative example of a transaction data content, after base64url decoding one of the strings in the `transaction_data` parameter:

```
{
  "type": "example_type",
  "credential_ids": [ "id_card_credential" ],
  // other transaction data type specific parameters
}
```

`verifier_info`: OPTIONAL. A non-empty array of attestations about the Verifier relevant to the Credential Request. These attestations MAY include Verifier metadata, policies, trust status, or authorizations. Attestations are intended to support authorization decisions, inform Wallet policy enforcement, or enrich the End-User consent dialog. Each object has the following structure:

- `format`: REQUIRED. A string that identifies the format of the attestation and how it is encoded. Ecosystems SHOULD use collision-resistant identifiers. Further processing of the attestation is determined by the type of the attestation, which is specified in a format-specific way.
- `data`: REQUIRED. An object or string containing an attestation (e.g. a JWT). The payload structure is defined on a per format level. It is at the discretion of the Wallet whether it uses the information from `verifier_info`. Factors that influence such Wallet's decision include, but are not limited to, trust

framework the Wallet supports, specific policies defined by the Issuers or ecosystem, and profiles of this specification. If the Wallet uses information from `verifier_info`, the Wallet MUST validate the signature and ensure binding.

- `credential_ids`: OPTIONAL. A non-empty array of strings each referencing a Credential requested by the Verifier for which the attestation is relevant. Each string matches the `id` field in a DCQL Credential Query. If omitted, the attestation is relevant to all requested Credentials.

See Section 5.11 for more details.

The following is a non-normative example of an attested object:

```
{
  "format": "jwt",
  "data": "eyJhbGciOiJFUzI1...EF0RBtvPClL71TWHlIQ",
  "credential_ids": [ "id_card" ]
}
```

5.2. Existing Parameters

The following additional considerations are given for pre-existing Authorization Request parameters:

nonce: REQUIRED. A case-sensitive String representing a value to securely bind Verifiable Presentation(s) provided by the Wallet to the particular transaction. The Verifier MUST create a fresh, cryptographically random number with sufficient entropy for every Authorization Request, store it with its current session, and pass it in the nonce Authorization Request Parameter to the Wallet. See Section 14.1 for details. Values MUST only contain ASCII URL safe characters (uppercase and lowercase letters, decimal digits, hyphen, period, underscore, and tilde).

scope: OPTIONAL. Defined in [RFC6749]. The Wallet MAY allow Verifiers to request Presentations by utilizing a pre-defined scope value. See Section 5.5 for more details.

`response_mode`: REQUIRED. Defined in [OAuth.Responses]. This parameter can be used (through the new Response Mode `direct_post`) to ask the Wallet to send the response to the Verifier via an HTTPS connection (see Section 8.2 for more details). It can also be used to request that the resulting response be encrypted (see Section 8.3 for more details).

`client_id`: REQUIRED. Defined in [RFC6749]. This specification defines additional requirements to enable the use of Client Identifier Prefixes as described in Section 5.9. The Client Identifier can be created by parties other than the Wallet and it is considered unique within the context of the Wallet when used in combination with the Client Identifier Prefix.

`state`: REQUIRED under the conditions defined in Section 5.3. Otherwise, `state` is OPTIONAL. `state` values MUST only contain ASCII URL safe characters (uppercase and lowercase letters, decimal digits, hyphen, period, underscore, and tilde).

5.3. Requesting Presentations without Holder Binding Proofs

The primary use case of this specification is to request and present Verifiable Presentations, i.e., Presentations that contain a cryptographic Holder Binding proof.

However, there are use cases where the Verifier wants to request presentation of Credentials without a proof of cryptographic Holder Binding. Examples for such use cases include low-security Credentials that do not support Holder Binding (e.g., a cinema ticket), Credentials that are bound to a biometric trait, or Credentials that are bound to claims (e.g., a diploma). In some cases, Credentials may support Holder Binding, but the Verifier may not require it for the Presentation.

A Verifier that requests and accepts a Presentation of a Credential without a proof of Holder Binding accepts that the presented Credential may have been replayed. Section 14.1 contains additional considerations for this case.

To request a Credential without proof of Holder Binding, the Verifier uses the `require_cryptographic_holder_binding` parameter in the DCQL request as defined in Section 6 and Appendix B.

In this protocol, the nonce parameter serves to securely link the request and response and as a replay protection in the Holder Binding proof. Without the key binding proof, nonce is not returned in the response. To maintain the binding between request and response, the Verifier MUST

- include a `state` parameter as defined in Section 4.1.1 of [RFC6749] in the Authorization Request,
- ensure that the value is a cryptographically strong pseudo-random number with at least 128 bits of entropy,
- ensure that the value is chosen fresh for each Authorization Request,
- store it in the Verifier's session state, and
- check that the same `state` value is returned in the Authorization Response,

if at least one Presentation without Holder Binding is requested and unless the Digital Credentials API is used. The Digital Credentials API uses internal mechanisms to maintain the binding.

When using Response Mode `direct_post`, also see Section 14.3.

5.4. Examples

The Verifier MAY send an Authorization Request using either of these 3 options:

1. Passing as URL with encoded parameters
2. Passing a request object as value
3. Passing a request object by reference

The second and third options are defined in the JWT-Secured Authorization Request (JAR) [RFC9101].

The following is a non-normative example of an Authorization Request with URL-encoded parameters:

```
GET /authorize?
  response_type=vp_token
  &client_id=redirect_uri%3Ahttps%3A%2F%2Fclient.example.org%2Fcb
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &dcql_query=...
  &transaction_data=...
  &nonce=n-0S6_WzA2Mj HTTP/1.1
```

The following is a non-normative example of an Authorization Request with a Request Object passed by value:

```
GET /authorize?
  client_id=redirect_uri%3Ahttps%3A%2F%2Fclient.example.org%2Fcb
  &request=eyJrd...
```

Where the contents of the `request` query parameter consist of a base64url-encoded and signed (in the example with RS256 algorithm) Request Object. The decoded payload is:

```
{
  "iss": "redirect_uri:https://client.example.org/cb",
  "aud": "https://self-issued.me/v2",
  "response_type": "vp_token",
  "client_id": "redirect_uri:https://client.example.org/cb",
  "redirect_uri": "https//client.example.org/cb",
  "dcql_query": {
    "credentials": [
      {
        "id": "some_identity_credential",
        "format": "dc+sd-jwt",
        "meta": {
          "vct_values": [ "https://credentials.example.com/identity_credential" ]
        },
        "claims": [
            {"path": ["last_name"]},
            {"path": ["first_name"]}
        ]
      }
    ]
  },
  "nonce": "n-0S6_WzA2Mj"
}
```

The following is a non-normative example of an Authorization Request with a request object passed by reference:

```
GET /authorize?
  client_id=x509_san_dns%3Aclient.example.org
  &request_uri=https%3A%2F%2Fclient.example.org%2Frequest%2Fvapof4ql2i7m41m68uep
  &request_uri_method=post HTTP/1.1
```

To retrieve the actual request, the Wallet might send the following non-normative example HTTP request to the `request_uri`:

```
POST /request/vapof4ql2i7m41m68uep HTTP/1.1
Host: client.example.org
Content-Type: application/x-www-form-urlencoded

wallet_metadata=%7B%22vp_formats_supported%22%3A%7B%22dc%2Bsd-jwt%22%3A%7B%22sd-jwt_alg
_values%22%3A%20%5B%22ES256%22%5D%2C%22kb-jwt_alg_values%22%3A%20%5B%22ES256%22%5D%7D%7
D%7D&
wallet_nonce=qPmxiNFCR3QTm19POc8u
```

5.5. Using scope Parameter to Request Presentations

Wallets MAY support requesting Presentations using OAuth 2.0 scope values.

Such a `scope` parameter value MUST be an alias for a well-defined DCQL query. Since multiple `scope` values can be used at the same time, the identifiers for Credentials (see Section 6.1) and claims (see Section 6.3) within the DCQL queries associated with `scope` values MUST be unique. This ensures that there are no collisions between the identifiers used in the DCQL queries and that the Verifier can unambiguously identify the requested Credentials in the response.

The specific scope values, and the mapping between a certain scope value and the respective DCQL query, are out of scope of this specification.

Possible options include normative text in a separate specification defining scope values along with a description of their semantics or machine-readable definitions in the Wallet's server metadata, mapping a scope value to an equivalent DCQL request.

It is RECOMMENDED to use collision-resistant scopes values.

The following is a non-normative example of an Authorization Request using the example scope value `com.example.IDCardCredential_presentation`:

```
GET /authorize?
  response_type=vp_token
  &client_id=https%3A%2F%2Fclient.example.org%2Fcb
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &scope=com.example.healthCardCredential_presentation
  &nonce=n-0S6_WzA2Mj HTTP/1.1
```

5.6. Response Type vp_token

This specification defines the Response Type `vp_token`.

`vp_token`: When supplied as the `response_type` parameter in an Authorization Request, a successful response MUST include the `vp_token` parameter. The Wallet SHOULD NOT return an OAuth 2.0 Authorization Code, Access Token, or Access Token Type in a successful response to the grant request. The default Response Mode for this Response Type is `fragment`, i.e., the Authorization Response parameters are encoded in the fragment added to the `redirect_uri` when redirecting back to the Verifier. The Response Type `vp_token` can be used with other Response Modes as defined in [OAuth.Responses]. Both successful and error responses SHOULD be returned using the supplied Response Mode, or if none is supplied, using the default Response Mode.

See Section 8 on how the `response_type` value determines the response used to return a VP Token.

5.7. Passing Authorization Request Across Devices

There are use-cases when the Authorization Request is being displayed on a device different from a device on which the requested Credential is stored. In those cases, an Authorization Request can be passed across devices by being rendered as a QR Code.

The usage of the Response Mode `direct_post` (see Section 8.2) in conjunction with `request_uri` is RECOMMENDED, since Authorization Request size might be large and might not fit in a QR code.

5.8. aud of a Request Object

When the Verifier is sending a Request Object as defined in [RFC9101], the aud claim value depends on whether the recipient of the request can be identified by the Verifier or not:

- the aud claim MUST be equal to the `iss` (issuer) claim value, when Dynamic Discovery is performed.
- the aud claim MUST be "https://self-issued.me/v2", when Static Discovery metadata is used.

Note: "https://self-issued.me/v2" is a symbolic string and can be used as an aud claim value even when this specification is used standalone, without SIOPv2.

5.9. Client Identifier Prefix and Verifier Metadata Management

This specification defines the concept of a Client Identifier Prefix that dictates how the Wallet needs to interpret the Client Identifier and associated data in the process of Client identification, authentication, and authorization. The Client Identifier Prefix enables deployments of this specification to use different mechanisms to obtain and

validate metadata of the Verifier beyond the scope of [RFC6749]. The term Client Identifier Prefix is used since the Verifier is acting as an OAuth 2.0 Client.

The Client Identifier Prefix is a string that MAY be communicated by the Verifier in a prefix within the `client_id` parameter in the Authorization Request. A fallback to pre-registered Clients as in [RFC6749] remains in place as a default mechanism in case no Client Identifier Prefix was provided. A certain Client Identifier Prefix may require the Verifier to sign the Authorization Request as a means of authentication and/or pass additional parameters and require the Wallet to process them.

5.9.1. Syntax

In the `client_id` Authorization Request parameter and other places where the Client Identifier is used, the Client Identifier Prefixes are prefixed to the usual Client Identifier, separated by a `:` (colon) character:

```
<client_id_prefix>:<orig_client_id>
```

Here, `<client_id_prefix>` is the Client Identifier Prefix and `<orig_client_id>` is an identifier for the Client within the namespace of that prefix. See Section 5.9.3 for Client Identifier Prefixes defined by this specification.

Wallets MUST use the presence of a `:` (colon) character and the content preceding it to determine whether a Client Identifier Prefix is used. If a `:` character is present and the content preceding it is a recognized and supported Client Identifier Prefix value, the Wallet MUST interpret the Client Identifier according to the given Client Identifier Prefix. The Client Identifier Prefix is defined as the string before the (first) `:` character. Note that implementations should not assume that the presence of a `:` character implies that the entire value can be processed as a valid URI. Instead, the specific processing rules defined for the specified Client Identifier Prefix (see Section 5.9.3) should be used to parse the `client_id` value.

For example, an Authorization Request might contain `client_id=verifier_attestation:example-client` to indicate that the `verifier_attestation` Client Identifier Prefix is to be used and that within this prefix, the Verifier can be identified by the string `example-client`. The presentation would contain the full `verifier_attestation:example-client` string as the audience (intended receiver) and the same full string would be used as the Client Identifier anywhere in the OAuth flow.

Note that the Verifier needs to determine which Client Identifier Prefixes the Wallet supports prior to sending the Authorization Request in order to choose a supported prefix.

Depending on the Client Identifier Prefix, the Verifier can communicate a JSON object with its metadata using the `client_metadata` parameter which contains name/value pairs.

5.9.2. Fallback

If a `:` character is not present in the Client Identifier, the Wallet MUST treat the Client Identifier as referencing a pre-registered client. This is equivalent to the [RFC6749] default behavior, i.e., the Client Identifier needs to be known to the Wallet in advance of the Authorization Request. The Verifier metadata is obtained using [RFC7591] or through out-of-band mechanisms.

For example, if an Authorization Request contains `client_id=example-client`, the Wallet would interpret the Client Identifier as referring to a pre-registered client.

If a `:` character is present in the Client Identifier but the value preceding it is not a recognized and supported Client Identifier Prefix value, the Wallet can treat the Client Identifier as referring to a pre-registered client or it may refuse the request.

From this definition, it follows that pre-registered clients MUST NOT contain a `:` character preceded immediately by a supported Client Identifier Prefix value in the first part of their Client Identifier.

5.9.3. Defined Client Identifier Prefixes

This specification defines the following Client Identifier Prefixes, followed by the examples where applicable.

In case of using OpenID4VP over DC API, as defined in Appendix A, it is at the discretion of the Wallet whether it validates the signature on the Request Object following the processing rules defined by a relevant Client Identifier Prefix. Factors that influence the Wallet's decision include, but are not limited to, the trust framework the Wallet supports, the specific policies defined by the Issuers or ecosystem, and profiles of this specification.

- `redirect_uri`: This prefix value indicates that the original Client Identifier part (without the prefix `redirect_uri:`) is the Verifier's Redirect URI (or Response URI when Response Mode `direct_post` is used). The Verifier MAY omit the `redirect_uri` Authorization Request parameter (or `response_uri` when Response Mode `direct_post` is used). All Verifier metadata parameters MUST be passed using the `client_metadata` parameter defined in Section 5.1. An example Client Identifier value is `redirect_uri:https://client.example.org/cb`. Requests using the `redirect_uri` Client Identifier Prefix cannot be signed because there is no method for the Wallet to obtain a trusted key for verification. Therefore, implementations requiring signed requests cannot use the `redirect_uri` Client Identifier Prefix.

  The following is a non-normative example of an unsigned request with the `redirect_uri` Client Identifier Prefix:

```
HTTP/1.1 302 Found
Location: https://wallet.example.org/universal-link?
  response_type=vp_token
  &client_id=redirect_uri%3Ahttps%3A%2F%2Fclient.example.org%2Fcb
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &dcql_query=...
  &nonce=n-0S6_WzA2Mj
  &client_metadata=%7B%22vp_formats_supported%22%3A%7B%22dc%2Bsd-jwt%22%3A%7B%22sd-
jwt_
  alg_values%22%3A%20%5B%22ES256%22%5D%2C%22kb-
jwt_alg_values%22%3A%20%5B%22ES256%22%5D
  %7D%7D%7D
```

- `openid_federation`: This prefix value indicates that the original Client Identifier (the part without the prefix `openid_federation:`) is an Entity Identifier defined in OpenID Federation [OpenID.Federation]. Processing rules given in [OpenID.Federation] MUST be followed. The Authorization Request MAY also contain a `trust_chain` parameter. The final Verifier metadata is obtained from the Trust Chain after applying the policies, according to [OpenID.Federation]. The `client_metadata` parameter, if present in the Authorization Request, MUST be ignored when this Client Identifier Prefix is used. Example Client Identifier: `openid_federation:https://federation-verifier.example.com`.
- `decentralized_identifier`: This prefix value indicates that the original Client Identifier (the part without the prefix `decentralized_identifier:`) is a Decentralized Identifier as defined in [DID-Core]. The request MUST be signed with a private key associated with the DID. A public key to verify the signature MUST be obtained from the `verificationMethod` property of a DID Document. Since DID Document may include multiple public keys, a particular public key used to sign the request in question MUST be identified by the `kid` in the JOSE Header. To obtain the DID Document, the Wallet MUST use DID Resolution defined by the DID method used by the Verifier. All Verifier metadata other than the public key MUST be obtained from the `client_metadata` parameter as defined in Section 5.1. Example Client Identifier: `decentralized_identifier:did:example:123`.

  The following is a non-normative example of a header and a body of a signed Request Object when the Client Identifier Prefix is `decentralized_identifier`:

Header

```
{
  "typ": "oauth-authz-req+jwt",
  "alg": "RS256",
  "kid": "did:example:123#1"
}
```

Body

```
{
  "client_id": "decentralized_identifier:did:example:123",
  "response_type": "vp_token",
  "redirect_uri": "https://client.example.org/callback",
  "nonce": "n-0S6_WzA2Mj",
  "dcql_query": { ... },
  "client_metadata": {
    "vp_formats_supported": {
      "dc+sd-jwt": {
        "sd-jwt_alg_values": ["ES256", "ES384"],
        "kb-jwt_alg_values": ["ES256", "ES384"]
      }
    }
  }
}
```

- `verifier_attestation`: This Client Identifier Prefix allows the Verifier to authenticate using a JWT that is bound to a certain public key as defined in Section 12. When the Client Identifier Prefix is `verifier_attestation`, the original Client Identifier (the part without the `verifier_attestation:` prefix) MUST equal the `sub` claim value in the Verifier attestation JWT. The request MUST be signed with the private key corresponding to the public key in the `cnf` claim in the Verifier attestation JWT. This serves as proof of possession of this key. The Verifier attestation JWT MUST be added to the `jwt` JOSE Header of the request object (see Section 12). The Wallet MUST validate the signature on the Verifier attestation JWT. The `iss` claim value of the Verifier Attestation JWT MUST identify a party the Wallet trusts for issuing Verifier Attestation JWTs. If the Wallet cannot establish trust, it MUST refuse the request. If the issuer of the Verifier Attestation JWT adds a `redirect_uris` claim to the attestation, the Wallet MUST ensure the `redirect_uri` request parameter value exactly matches one of the `redirect_uris` claim entries. All Verifier metadata other than the public key MUST be obtained from the `client_metadata` parameter. Example Client Identifier: `verifier_attestation:verifier.example`.
- `x509_san_dns`: When the Client Identifier Prefix is `x509_san_dns`, the original Client Identifier (the part after the `x509_san_dns:` prefix) MUST be a DNS name and match a `dNSName` Subject Alternative Name (SAN) [RFC5280] entry in the leaf certificate passed with the request. The request MUST be signed with the private key corresponding to the public key in the leaf X.509 certificate of the certificate chain added to the request in the `x5c` JOSE header [RFC7515] of the signed request object. The Wallet MUST validate the signature and the trust chain of the X.509 certificate. All Verifier metadata other than the public key MUST be obtained from the `client_metadata` parameter. The following requirement applies unless the interaction is using the DC API as defined in Appendix A: If the Wallet can establish trust in the Client Identifier authenticated through the certificate, e.g. because the Client Identifier is contained in a list of trusted Client Identifiers, it may allow the client to freely choose the `redirect_uri` value. If not, the FQDN of the `redirect_uri` value MUST match the Client Identifier without the prefix `x509_san_dns:`. Example Client Identifier: `x509_san_dns:client.example.org`.
- `x509_hash`: When the Client Identifier Prefix is `x509_hash`, the original Client Identifier (the part without the `x509_hash:` prefix) MUST be a hash and match the hash of the leaf certificate passed with the request. The request MUST be signed with the private key corresponding to the public key in the leaf X.509 certificate of

the certificate chain added to the request in the x5c JOSE header parameter [RFC7515] of the signed request object. The value of x509_hash is the base64url-encoded value of the SHA-256 hash of the DER-encoded X.509 certificate. The Wallet MUST validate the signature and the trust chain of the X.509 leaf certificate. All Verifier metadata other than the public key MUST be obtained from the client_metadata parameter. Example Client Identifier: x509_hash:Uvo3HtuIxuhC92rShpgqcT3YXwrqRxWEviRiA0OZszk

- origin: This reserved Client Identifier Prefix is defined in Appendix A.2. The Wallet MUST NOT accept this Client Identifier Prefix in requests. In OpenID4VP over the Digital Credentials API, the audience of the Credential Presentation is always the origin value prefixed by origin:, for example origin:https://verifier.example.com/.

To use the Client Identifier Prefixes openid_federation, decentralized_identifier, verifier_attestation, x509_san_dns and x509_hash, Verifiers MUST be capable of securely storing private key material. This might require changes to the technical design of native apps as such apps are typically public clients.

Other specifications can define further Client Identifier Prefixes. It is RECOMMENDED to use collision-resistant names for such values.

5.10. Request URI Method post

This request is handled by the Request URI endpoint of the Verifier.

The request MUST use the HTTP POST method with the https scheme, and the content type application/x-www-form-urlencoded and the Accept header set to application/oauth-authz-req+jwt. The names and values in the body MUST be encoded using UTF-8.

The following parameters are defined to be included in the request to the Request URI Endpoint:

wallet_metadata: OPTIONAL. A string containing a JSON object containing metadata parameters as defined in Section 10.

wallet_nonce: OPTIONAL. A string value used to mitigate replay attacks of the Authorization Request. When received, the Verifier MUST use it as the wallet_nonce value in the signed authorization request object. Value can be a base64url-encoded, fresh, cryptographically random number with sufficient entropy.

If the Wallet requires the Verifier to encrypt the Request Object, it SHOULD use the jwks parameter within the wallet_metadata parameter to pass public encryption keys. If the Wallet requires an encrypted Authorization Response, it SHOULD specify supported encryption algorithms using the authorization_encryption_alg_values_supported and authorization_encryption_enc_values_supported parameters.

Additionally, if the Client Identifier Prefix permits signed Request Objects, the Wallet SHOULD list supported cryptographic algorithms for securing the Request Object through the request_object_signing_alg_values_supported parameter. Conversely, the Wallet MUST NOT include this parameter if the Client Identifier Prefix precludes signed Request Objects.

Additional parameters MAY be defined and used in the request to the Request URI Endpoint. The Verifier MUST ignore any unrecognized parameters.

The following is a non-normative example of a request:

```
POST /request HTTP/1.1
Host: client.example.org
Content-Type: application/x-www-form-urlencoded

wallet_metadata=%7B%22vp_formats_supported%22%3A%7B%22dc%2Bsd-jwt%22%3A%7B%22sd-jwt_a
lg_values%22%3A%20%5B%22ES256%22%5D%2C%22kb-jwt_alg_values%22%3A%20%5B%22ES256%22%5D%
7D%7D%7D&
wallet_nonce=qPmxiNFCR3QTm19POc8u
```

5.10.1. Request URI Response

The Request URI response MUST be an HTTP response with the content type `application/oauth-authz-req+jwt` and the body being a signed, optionally encrypted, request object as defined in [RFC9101]. The request object MUST fulfill the requirements as defined in Section 5.

The following is a non-normative example of a payload for a request object:

```
{
  "client_id": "x509_san_dns:client.example.org",
  "response_uri": "https://client.example.org/post",
  "response_type": "vp_token",
  "response_mode": "direct_post",
  "dcql_query": {...},
  "nonce": "n-0S6_WzA2Mj",
  "wallet_nonce": "qPmxiNFCR3QTm19POc8u",
  "state" : "eyJhb...6-sVA"
}
```

The Wallet MUST process the request as defined in [RFC9101]. Additionally, if the Wallet passed a `wallet_nonce` in the POST request, the Wallet MUST validate whether the request object contains the respective nonce value in a `wallet_nonce` claim. If it does not, the Wallet MUST terminate request processing.

The Wallet MUST extract the set of Authorization Request parameters from the Request Object. The Wallet MUST only use the parameters in this Request Object, even if the same parameter was provided in an Authorization Request query parameter. The Client Identifier value in the `client_id` Authorization Request parameter and the Request Object `client_id` claim value MUST be identical, including the Client Identifier Prefix. If any of these conditions are not met, the Wallet MUST terminate request processing.

The Wallet then validates the request as specified in OAuth 2.0 [RFC6749].

5.10.2. Request URI Error Response

If the Verifier responds with any HTTP error response, the Wallet MUST terminate the process.

5.11. Verifier Info

Verifier Info parameter allows the Verifier to provide additional context or metadata as part of the Authorization Request attested by a trusted third party. These inputs can support a variety of use cases, such as helping the Wallet apply policy decisions, validating eligibility, or presenting more meaningful information to the End-User during consent.

Each Verifier Info object contains a type identifier, associated data and optionally references to Credential identifiers. The format and semantics of these attestations are defined by ecosystems or profiles.

For example, a Verifier might include:

- A **registration certificate** issued by a trusted authority, to prove that the Verifier has publicly registered its intent to request certain credentials.
- A **policy statement**, such as a signed document describing acceptable use, retention periods, or access rights.
- The **confirmation of a role** of the Verifier in a certain domain, e.g. the Verifier might be a certified payment service provider under the EU's Payment Service Directive 2.

The Verifier Info parameter is optional. Wallets MAY use them to make authorization decisions or to enhance the user experience, but they SHOULD ignore any unrecognized or unsupported Verifier Info types.

5.11.1. Proof of Possession

This specification supports two models for proof of possession:

- **claim-bound attestations**: The attestation is not signed by the Verifier, but bound to it. The exact binding mechanism is defined by the type of the definition. For example for JWTs, the `sub` claim is including the distinguished name of the Certificate that was used to sign the request. The binding may also include the `client_id` parameter.
- **key-bound attestations**: The attestation's proof of possession is signed by the Verifier with a key contained or related to the attestation. To bind the signature to the presentation request, the respective signature object should include the `nonce` and `client_id` request parameters. The attestation and the proof of possession have to be passed in the attachment.

The Wallet MUST validate such proofs if defined by the profile and ignore or reject attachments that fail validation.

6. Digital Credentials Query Language (DCQL)

The Digital Credentials Query Language (DCQL, pronounced ['dak|]) is a JSON-encoded query language that allows the Verifier to request Presentations that match the query. The Verifier MAY encode constraints on the combinations of Credentials and claims that are requested. The Wallet evaluates the query against the Credentials it holds and returns Presentations matching the query.

A valid DCQL query is defined as a JSON-encoded object with the following top-level properties:

`credentials`: REQUIRED. A non-empty array of Credential Queries as defined in Section 6.1 that specify the requested Credentials.
`credential_sets`: OPTIONAL. A non-empty array of Credential Set Queries as defined in Section 6.2 that specifies additional constraints on which of the requested Credentials to return.

Note: Future extensions may define additional properties both at the top level and in the rest of the DCQL data structure. Implementations MUST ignore any unknown properties.

6.1. Credential Query

A Credential Query is an object representing a request for a presentation of one or more matching Credentials.

Each entry in `credentials` MUST be an object with the following properties:

`id`: REQUIRED. A string identifying the Credential in the response and, if provided, the constraints in `credential_sets`. The value MUST be a non-empty string consisting of alphanumeric, underscore (_), or hyphen (-) characters. Within the Authorization Request, the same `id` MUST NOT be present more than once.
`format`: REQUIRED. A string that specifies the format of the requested Credential. Valid Credential Format Identifier values are defined in Appendix B.
`multiple`: OPTIONAL. A boolean which indicates whether multiple Credentials can be returned for this Credential Query. If omitted, the default value is `false`.

`meta`: REQUIRED. An object defining additional properties requested by the Verifier that apply to the metadata and validity data of the Credential. The properties of this object are defined per Credential Format. Examples of those are in Appendix B.3.5 and Appendix B.2.3. If empty, no specific constraints are placed on the metadata or validity of the requested Credential.

`trusted_authorities`: OPTIONAL. A non-empty array of objects as defined in Section 6.1.1 that specifies expected authorities or trust frameworks that certify Issuers, that the Verifier will accept. Every Credential returned by the Wallet SHOULD match at least one of the conditions present in the corresponding `trusted_authorities` array if present.

Note that Verifiers must verify that the issuer of a received presentation is trusted on their own and this feature mainly aims to help data minimization by not revealing information that would likely be rejected.

`require_cryptographic_holder_binding`: OPTIONAL. A boolean which indicates whether the Verifier requires a Cryptographic Holder Binding proof. The default value is `true`, i.e., a Verifiable Presentation with Cryptographic Holder Binding is required. If set to `false`, the Verifier accepts a Credential without Cryptographic Holder Binding proof.

`claims`: OPTIONAL. A non-empty array of objects as defined in Section 6.3 that specifies claims in the requested Credential. Verifiers MUST NOT point to the same claim more than once in a single query. Wallets SHOULD ignore such duplicate claim queries.

`claim_sets`: OPTIONAL. A non-empty array containing arrays of identifiers for elements in `claims` that specifies which combinations of `claims` for the Credential are requested. The rules for selecting claims to send are defined in Section 6.4.1.

Multiple Credential Queries in a request MAY request a presentation of the same Credential.

6.1.1. Trusted Authorities Query

A Trusted Authorities Query is an object representing information that helps to identify an authority or the trust framework that certifies Issuers. A Credential is identified as a match to a Trusted Authorities Query if it matches with one of the provided values in one of the provided types. How exactly the matching works is defined for the different types below.

Note that direct Issuer matching can also work using claim value matching if supported (e.g., value matching the `iss` claim in an SD-JWT) if the mechanisms for `trusted_authorities` are not applicable but might be less likely to work due to the constraints on value matching (see Section 6.4.1 for more details).

Each entry in `trusted_authorities` MUST be an object with the following properties:

`type`: REQUIRED. A string uniquely identifying the type of information about the issuer trust framework. Types defined by this specification are listed below.

`values`: REQUIRED. A non-empty array of strings, where each string (value) contains information specific to the used Trusted Authorities Query type that allows the identification of an issuer, a trust framework, or a federation that an issuer belongs to.

Below are descriptions for the different Type Identifiers (string), detailing how to interpret and perform the matching logic for each provided value.

Note that depending on the trusted authorities type used, the underlying mechanisms can have different privacy implications. More detailed privacy considerations for the trusted authorities can be found in Section 15.10.

6.1.1.1. Authority Key Identifier

Type: `"aki"`

Value: Contains the KeyIdentifier of the AuthorityKeyIdentifier as defined in Section 4.2.1.1 of [RFC5280], encoded as base64url. The raw byte representation of this element MUST match with the AuthorityKeyIdentifier element of an X.509 certificate in the certificate chain present in the Credential (e.g., in the header of an mdoc or SD-JWT). Note that the chain can consist of a single certificate and the Credential can include the entire X.509 chain or parts of it.

Below is a non-normative example of such an entry of type `aki`:

```
{
  "type": "aki",
  "values": ["s9tIpPmhxdiuNkHMEWNpYim8S8Y"]
}
```

6.1.1.2. ETSI Trusted List

Type: `"etsi_tl"`
Value: The identifier of a Trusted List as specified in ETSI TS 119 612 [ETSI.TL]. An ETSI Trusted List contains references to other Trusted Lists, creating a list of trusted lists, or entries for Trust Service Providers with corresponding service description and X.509 Certificates. The trust chain of a matching Credential MUST contain at least one X.509 Certificate that matches one of the entries of the Trusted List or its cascading Trusted Lists.

Below is a non-normative example of such an entry of type `etsi_tl`:

```
{
  "type": "etsi_tl",
  "values": ["https://lotl.example.com"]
}
```

6.1.1.3. OpenID Federation

Type: `"openid_federation"`
Value: The `Entity Identifier` as defined in Section 1 of [OpenID.Federation] that is bound to an entity in a federation. While this Entity Identifier could be any entity in that ecosystem, this entity would usually have the Entity Configuration of a Trust Anchor. A valid trust path, including the given Entity Identifier, must be constructible from a matching credential.

Below is a non-normative example of such an entry of type `openid_federation`:

```
{
  "type": "openid_federation",
  "values": ["https://trustanchor.example.com"]
}
```

6.2. Credential Set Query

A Credential Set Query is an object representing a request for one or more Credentials to satisfy a particular use case with the Verifier.

Each entry in `credential_sets` MUST be an object with the following properties:

`options`: REQUIRED A non-empty array, where each value in the array is a list of Credential Query identifiers representing one set of Credentials that satisfies the use case. The value of each element in the `options` array is a non-empty array of identifiers which reference elements in `credentials`.

`required`: OPTIONAL A boolean which indicates whether this set of Credentials is required to satisfy the particular use case at the Verifier. If omitted, the default value is `true`.

Before sending the presentation request, the Verifier SHOULD display to the End-User the purpose, context, or reason for the query to the Wallet.

6.3. Claims Query

Each entry in `claims` MUST be an object with the following properties:

`id`: REQUIRED if `claim_sets` is present in the Credential Query; OPTIONAL otherwise. A string identifying the particular claim. The value MUST be a non-empty string consisting of alphanumeric, underscore (_), or hyphen (-) characters. Within the particular `claims` array, the same `id` MUST NOT be present more than once.

`path`: REQUIRED The value MUST be a non-empty array representing a claims path pointer that specifies the path to a claim within the Credential, as defined in Section 7.

`values`: OPTIONAL A non-empty array of strings, integers or boolean values that specifies the expected values of the claim. If the `values` property is present, the Wallet SHOULD return the claim only if the type and value of the claim both match exactly for at least one of the elements in the array. Details of the processing rules are defined in Section 6.4.1.

If a Wallet implements value matching and the Credential being matched is an ISO mdoc-based credential, the CBOR value used for matching MUST first be converted to JSON, following the advice given in Section 6.1 of [RFC8949]. The resulting JSON value is then used to match against the `values` property as specified above. When conversion according to these rules is not clearly defined, behavior is out of scope of this specification.

6.4. Selecting Claims and Credentials

The following section describes the logic that applies for selecting claims and for selecting credentials.

For formats supporting selective disclosure, these rules support selecting a minimal dataset to fulfill the Verifier's request in a privacy-friendly manner (see Section 15 for additional considerations). Wallets MUST NOT send selectively disclosable claims that have not been selected according to the rules below. A single Presentation of a Credential MAY contain more than the claims selected in the particular DCQL Credential Query if the same Credential is selected with the additional claims in a separate Credential Query in the same request, or the additional claims are not selectively disclosable.

6.4.1. Selecting Claims

The following rules apply for selecting claims via `claims` and `claim_sets`:

- If `claims` is absent, the Verifier is requesting no claims that are selectively disclosable; the Wallet MUST return only the claims that are mandatory to present (e.g., SD-JWT and Key Binding JWT for a Credential of format IETF SD-JWT VC).
- If `claims` is present, but `claim_sets` is absent, the Verifier requests all claims listed in `claims`.
- If both `claims` and `claim_sets` are present, the Verifier requests one combination of the claims listed in `claim_sets`. The order of the options conveyed in the `claim_sets` array expresses the Verifier's preference for what is returned; the Wallet SHOULD return the first option that it can satisfy. If the Wallet cannot satisfy any of the options, it MUST NOT return any claims.
- `claim_sets` MUST NOT be present if `claims` is absent.

When a Claims Query contains a restriction on the values of a claim, the Wallet SHOULD NOT return the claim if its value does not match according to the rules for `values` defined in Section 6.3, i.e., the claim should be treated the same as if it did not exist in the Credential. Implementing this restriction may not be possible in all cases, for example, if the Wallet does not have access to the claim value before presentation or user consent or if another component routing the request to the Wallet does not have access to the claim value. It is ultimately up to the Wallet and/or the End-User if the value matching request is followed. Therefore, Verifiers MUST treat restrictions expressed using `values` as a best-effort way to improve user privacy, but MUST NOT rely on it for security checks.

The purpose of the `claim_sets` syntax is to provide a way for a Verifier to describe alternative ways a given Credential can satisfy the request. The array ordering expresses the Verifier's preference for how to fulfill the request. The first element in the array is the most preferred and the last element in the array is the least preferred. Verifiers SHOULD use the principle of least information disclosure to influence how they order these options. For example, a proof of age request should prioritize requesting an attribute like `age_over_18` over an attribute like `birth_date`. The `claim_sets` syntax is not intended to define options the End-User can choose from, see Section 6.4.3 for more information. The Wallet is recommended to return the first option it can satisfy since that is the preferred option from the Verifier. However, there can be reasons to deviate. Non-exhaustive examples of such reasons are:

- scenarios where the Verifier did not order the options to minimize information disclosure
- operational reasons why returning a different option than the first option has UX benefits for the Wallet.

If the Wallet cannot deliver all claims requested by the Verifier according to these rules, it MUST NOT return the respective Credential.

For Credential Formats that do not support selective disclosure, the case of both `claims` and `claim_sets` being absent is interpreted as requesting a presentation of the "full credential" since all claims are mandatory to present.

6.4.2. Selecting Credentials

The following rules apply for selecting Credentials via `credentials` and `credential_sets`:

- If `credential_sets` is not provided, the Verifier requests presentations for all Credentials in `credentials` to be returned.
- Otherwise, the Verifier requests presentations of Credentials to be returned satisfying
  - all of the Credential Set Queries in the `credential_sets` array where the `required` attribute is true or omitted, and
  - optionally, any of the other Credential Set Queries.

To satisfy a Credential Set Query, the Wallet MUST return presentations of a set of Credentials that match to one of the `options` inside the Credential Set Query.

Credentials not matching the respective constraints expressed within `credentials` MUST NOT be returned, i.e., they are treated as if they would not exist in the Wallet.

If the Wallet cannot deliver all non-optional Credentials requested by the Verifier according to these rules, it MUST NOT return any Credential(s).

6.4.3. User Interface Considerations

While this specification provides the mechanisms for requesting different sets of claims and Credentials, it does not define details about the user interface of the Wallet, for example, if and how End-Users can select which combination of Credentials to present. However, it is typically expected that the Wallet presents the End-User with a choice of which Credential(s) to present if multiple of the sets of Credentials in `options` can satisfy the request.

7. Claims Path Pointer

A claims path pointer is a pointer into the Credential, identifying one or more claims. A claims path pointer MUST be a non-empty array of strings, nulls and non-negative integers. A claims path pointer can be processed, which means it is applied to a Credential. The results of processing are the referenced claims.

7.1. Semantics for JSON-based credentials

This section defines the semantics of a claims path pointer when applied to a JSON-based Credential.

A string value indicates that the respective key is to be selected, a null value indicates that all elements of the currently selected array(s) are to be selected; and a non-negative integer indicates that the respective index in an array is to be selected. The path is formed as follows:

Start with an empty array and repeat the following until the full path is formed.

- To address a particular claim within an object, append the key (claim name) to the array.
- To address an element within an array, append the index to the array (as a non-negative, 0-based integer).
- To address all elements within an array, append a null value to the array.

7.1.1. Processing

In detail, the array is processed from left to right as follows:

1. Select the root element of the Credential, i.e., the top-level JSON object.
2. Process the query of the claims path pointer array from left to right:
    1. If the component is a string, select the element in the respective key in the currently selected element(s). If any of the currently selected element(s) is not an object, abort processing and return an error. If the key does not exist in any element currently selected, remove that element from the selection.
    2. If the component is null, select all elements of the currently selected array(s). If any of the currently selected element(s) is not an array, abort processing and return an error.
    3. If the component is a non-negative integer, select the element at the respective index in the currently selected array(s). If any of the currently selected element(s) is not an array, abort processing and return an error. If the index does not exist in a selected array, remove that array from the selection.
    4. If the component is anything else, abort processing and return an error.
3. If the set of elements currently selected is empty, abort processing and return an error.

The result of the processing is the set of selected JSON elements.

7.2. Semantics for ISO mdoc-based credentials

This section defines the semantics of a claims path pointer when applied to a credential in ISO mdoc format.

A claims path pointer into an mdoc contains two elements of type string. The first element refers to a namespace and the second element refers to a data element identifier.

7.2.1. Processing

In detail, the array is processed as follows:

1. If the claims path pointer does not contain exactly two components or one of the components is not a string then abort processing and return an error.

2. Select the namespace referenced by the first component. If the namespace does not exist in the mdoc then abort processing and return an error.
3. Select the data element referenced by the second component. If the data element does not exist in the Credential then abort processing and return an error.

The result of the processing is the selected data element value as CBOR data item.

7.3. Claims Path Pointer Example

The following shows a non-normative, simplified example of a JSON-based Credential:

```
{
  "name": "Arthur Dent",
  "address": {
    "street_address": "42 Market Street",
    "locality": "Milliways",
    "postal_code": "12345"
  },
  "degrees": [
    {
      "type": "Bachelor of Science",
      "university": "University of Betelgeuse"
    },
    {
      "type": "Master of Science",
      "university": "University of Betelgeuse"
    }
  ],
  "nationalities": ["British", "Betelgeusian"]
}
```

The following shows examples of claims path pointers and the respective selected claims:

- `["name"]`: The claim `name` with the value `Arthur Dent` is selected.
- `["address"]`: The claim `address` with its sub-claims as the value is selected.
- `["address", "street_address"]`: The claim `street_address` with the value `42 Market Street` is selected.
- `["degrees", null, "type"]`: All `type` claims in the `degrees` array are selected.
- `["nationalities", 1]`: The second nationality is selected.

7.4. DCQL Examples

The following is a non-normative example of a DCQL query that requests a Credential of the format `dc+sd-jwt` with a type value of `https://credentials.example.com/identity_credential` and the claims `last_name`, `first_name`, and `address.street_address`:

```
{
  "credentials": [
    {
      "id": "my_credential",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": [ "https://credentials.example.com/identity_credential" ]
      },
      "claims": [
          {"path": ["last_name"]},
          {"path": ["first_name"]},
          {"path": ["address", "street_address"]}
      ]
    }
  ]
}
```

Additional, more complex examples can be found in Appendix D.

8. Response

A VP Token is only returned if the corresponding Authorization Request contained a `dcql_query` parameter or a `scope` parameter representing a DCQL Query Section 5.

A VP Token can be returned in the Authorization Response or the Token Response depending on the Response Type used. See Section 5.6 for more details.

If the Response Type value is `vp_token`, the VP Token is returned in the Authorization Response. When the Response Type value is `vp_token id_token` and the `scope` parameter contains `openid`, the VP Token is returned in the Authorization Response alongside a Self-Issued ID Token as defined in [SIOPv2].

If the Response Type value is `code` (Authorization Code Grant Type), the VP Token is provided in the Token Response.

The expected behavior is summarized in the following table:

`response_type` parameter value	Response containing the VP Token
`vp_token`	Authorization Response
`vp_token id_token`	Authorization Response
`code`	Token Response

Table 1: OpenID for Verifiable Presentations `response_type` values

The behavior with respect to the VP Token is unspecified for any other individual Response Type value, or a combination of Response Type values.

8.1. Response Parameters

When a VP Token is returned, the respective response includes the following parameters:

`vp_token`: REQUIRED. This is a JSON-encoded object containing entries where the key is the `id` value used for a Credential Query in the DCQL query and the value is an array of one or more Presentations that match the respective Credential Query. When `multiple` is omitted, or set to `false`, the array MUST contain only one Presentation. There MUST NOT be any entry in the JSON-encoded object for optional Credential Queries when

there are no matching Credentials for the respective Credential Query. Each Presentation is represented as a string or object, depending on the format as defined in Appendix B. The same rules as above apply for encoding the Presentations.

Other parameters, such as `code` (from [RFC6749]), or `id_token` (from [OpenID.Core]), and `iss` (from [RFC9207]) can be included in the response as defined in the respective specifications.

Additional response parameters MAY be defined and used, as described in [RFC6749]. The Client MUST ignore any unrecognized parameters.

The following is a non-normative example of an Authorization Response when the Response Type value in the Authorization Request was `vp_token`:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  vp_token=...
```

8.1.1. Examples

The following is a non-normative example of the contents of a VP Token containing a single Verifiable Presentation in the SD-JWT VC format after a request using DCQL like the one shown in Section 7.4 (shortened for brevity):

```
{
  "my_credential": ["eyJhbGci...QMA"]
}
```

The following is a non-normative example of the contents of a VP Token containing multiple Verifiable Presentations in the SD-JWT VC format when the Credential Query has `multiple` set to `true` (shortened for brevity):

```
{
  "my_credential": ["eyJhbGci...QMA", "eyJhbGci...QMA", ...]
}
```

8.2. Response Mode "direct_post"

The Response Mode `direct_post` allows the Wallet to send the Authorization Response to an endpoint controlled by the Verifier via an HTTP POST request.

It has been defined to address the following use cases:

- Verifier and Wallet are located on different devices; thus, the Wallet cannot send the Authorization Response to the Verifier using a redirect.
- The Authorization Response size exceeds the URL length limits of user agents, so flows relying only on redirects (such as Response Mode `fragment`) cannot be used. In those cases, the Response Mode `direct_post` is the way to convey the Presentations to the Verifier without the need for the Wallet to have a backend.

The Response Mode is defined in accordance with [OAuth.Responses] as follows:

`direct_post`: In this mode, the Authorization Response is sent to the Verifier using an HTTP POST request to an endpoint controlled by the Verifier. The Authorization Response MUST be encoded in the request body using the format defined by the `application/x-www-form-urlencoded` HTTP content type. The parameters in the request body MUST all be encoded using UTF-8. The Verifier can request that the Wallet redirects the End-User to the Verifier using the response as defined below.

The following new Authorization Request parameter is defined to be used in conjunction with Response Mode `direct_post`:

`response_uri`: REQUIRED when the Response Mode `direct_post` is used. The URL to which the Wallet MUST send the Authorization Response using an HTTP POST request as defined by the Response Mode `direct_post`. The Response URI receives all Authorization Response parameters as defined by the respective Response Type. When the `response_uri` parameter is present, the `redirect_uri` Authorization Request parameter MUST NOT be present. If the `redirect_uri` Authorization Request parameter is present when the Response Mode is `direct_post`, the Wallet MUST return an `invalid_request` Authorization Response error. The `response_uri` value MUST be a value that the client would be permitted to use as `redirect_uri` when following the rules defined in Section 5.9.

Note: When the specification text refers to the usage of Redirect URI in the Authorization Request, that part of the text also applies when Response URI is used in the Authorization Request with Response Mode `direct_post`.

Note: The Verifier's component providing the user interface (Frontend) and the Verifier's component providing the Response URI need to be able to map authorization requests to the respective authorization responses. The Verifier MAY use the `state` Authorization Request parameter to add appropriate data to the Authorization Response for that purpose, for details see Section 13.3.

Additional request parameters MAY be defined and used with the Response Mode `direct_post`. The Wallet MUST ignore any unrecognized parameters.

The following is a non-normative example of the payload of a Request Object with Response Mode `direct_post`:

```
{
  "client_id": "redirect_uri:https://client.example.org/post",
  "response_uri": "https://client.example.org/post",
  "response_type": "vp_token",
  "response_mode": "direct_post",
  "dcql_query": {...},
  "nonce": "n-0S6_WzA2Mj",
  "state": "eyJhb...6-sVA"
}
```

The following non-normative example of an Authorization Request refers to the Authorization Request Object from above through the `request_uri` parameter. The Authorization Request can be displayed to the End-User either directly (as a link) or as a QR Code:

```
https://wallet.example.com?
  client_id=https%3A%2F%2Fclient.example.org%2Fcb
  &request_uri=https%3A%2F%2Fclient.example.org%2F567545564
```

The following is a non-normative example of the Authorization Response that is sent via an HTTP POST request to the Verifier's Response URI:

```
POST /post HTTP/1.1
Host: client.example.org
Content-Type: application/x-www-form-urlencoded

  vp_token=...&
  state=eyJhb...6-sVA
```

The following is a non-normative example of an Authorization Error Response that is sent as an HTTP POST request to the Verifier's Response URI:

```
POST /post HTTP/1.1
Host: client.example.org
Content-Type: application/x-www-form-urlencoded

  error=invalid_request&
  error_description=unsupported%20client_id_prefix&
  state=eyJhb...6-sVA
```

If the Response URI has successfully processed the Authorization Response or Authorization Error Response, it MUST respond with an HTTP status code of 200 with `Content-Type` of `application/json` and a JSON object in the response body.

The following new parameter is defined for use in the JSON object returned from the Response Endpoint to the Wallet:

`redirect_uri`: OPTIONAL. String containing a URI. When this parameter is present the Wallet MUST redirect the user agent to this URI. This allows the Verifier to continue the interaction with the End-User on the device where the Wallet resides after the Wallet has sent the Authorization Response to the Response URI. It can be used by the Verifier to prevent session fixation (Section 14.2) attacks. The Response URI MAY return the `redirect_uri` parameter in response to successful Authorization Responses or for Error Responses.

Additional response parameters MAY be defined and used. The Wallet MUST ignore any unrecognized parameters.

Note: Response Mode `direct_post` without the `redirect_uri` could be less secure than Response Modes with redirects. For details, see (Section 14.2).

The value of the redirect URI is an absolute URI as defined by [RFC3986] Section 4.3 and is chosen by the Verifier. The Verifier MUST include a fresh, cryptographically random value in the URL. This value is used to ensure only the receiver of the redirect can fetch and process the Authorization Response. The value can be added as a path component, as a fragment or as a parameter to the URL. It is RECOMMENDED to use a cryptographic random value of 128 bits or more. For implementation considerations see Section 13.3.

The following is a non-normative example of the response from the Verifier to the Wallet upon receiving the Authorization Response at the Response URI (using a `response_code` parameter from Section 13.3):

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

{
  "redirect_uri":
"https://client.example.org/cb#response_code=091535f699ea575c7937fa5f0f454aee"
}
```

If the response does not contain the `redirect_uri` parameter, the Wallet is not required to perform any further steps.

Note: In the Response Mode `direct_post` or `direct_post.jwt`, the Wallet can change the UI based on the Verifier's callback to the Wallet following the submission of the Authorization Response.

Additional parameters MAY be defined and used in the response from the Response Endpoint to the Wallet. The Wallet MUST ignore any unrecognized parameters.

8.3. Encrypted Responses

This section defines how an Authorization Response containing a VP Token (such as when the Response Type value is `vp_token` or `vp_token id_token`) can be encrypted at the application level using [RFC7518] where the payload of the JWE is a JSON object containing the Authorization Response parameters. Encrypting the Authorization Response can, for example, prevent personal data in the Authorization Response from leaking, when the Authorization Response is returned through the front channel (e.g., the browser).

To encrypt the Authorization Response, implementations MUST use an unsigned, encrypted JWT as described in [RFC7519].

To obtain the Verifier's public key to which to encrypt the Authorization Response, the Wallet uses JWKs from client metadata (such as the `jwks` member within the `client_metadata` request parameter or other mechanisms as allowed by the given Client Identifier Prefix). Using what it supports and its preferences, the Wallet selects the public key to encrypt the Authorization Response based on information about each key, such as the `kty` (Key Type), `use` (Public Key Use), `alg` (Algorithm), and other JWK parameters. The `alg` parameter MUST be present in the JWKs. The JWE `alg` algorithm used MUST be equal to the `alg` value of the chosen `jwk`. If the selected public key contains a `kid` parameter, the JWE MUST include the same value in the `kid` JWE Header Parameter (as defined in Section 4.1.6) of the encrypted response. This enables the Verifier to easily identify the specific public key that was used to encrypt the response. The JWE `enc` content encryption algorithm used is obtained from the `encrypted_response_enc_values_supported` parameter of client metadata, such as the `client_metadata` request parameter, allowing for the default value of `A128GCM` when not explicitly set.

The payload of the encrypted JWT response MUST include the contents of the response as defined in Section 8.1 as top-level JSON members.

The following shows a non-normative example of the content of a request that is asking for an encrypted response while providing a few public keys for encryption in the `jwks` member of the `client_metadata` request parameter:

```
{
 "response_type": "vp_token",
 "response_mode": "dc_api.jwt",
 "nonce": "xyz123ltcaccescbwc777",
 "dcql_query": {
  "credentials": [
   {
    "id": "my_credential",
    "format": "dc+sd-jwt",
    "meta": {
      "vct_values": ["https://credentials.example.com/identity_credential"]
    },
    "claims": [
      {"path": ["last_name"]},
      {"path": ["first_name"]},
      {"path": ["address", "postal_code"]}
     ]
    }
   ]
 },
 "client_metadata": {
   "jwks": {
    "keys": [
    {
     "kty":"EC", "kid":"ac", "use":"enc", "crv":"P-256","alg":"ECDH-ES",
     "x":"YO4epjifD-KWeq1sL2tNmm36BhXnkJ0He-WqMYrp9Fk",
     "y":"Hekpm0zfK7C-YccH5iBjcIXgf6YdUvNUac_0At55Okk"
    },
    {
     "kty":"OKP","kid":"jc","use":"enc","crv":"X25519","alg":"ECDH-ES",
     "x":"WPX7wnwq10hFNK9aDSyG1QlLswE_CJY14LdhcFUIVVc"
    },
    {
     "kty":"EC","kid":"lc","use":"enc","crv":"P-384","alg":"ECDH-ES",
     "x":"iHytgLNtXjEyYMAIGwfgjINZRmLfObYbmjPhkaPD8OiTkJtRHjegTNdH31Mxg4nV",
     "y":"MizXWSqNB7sSt_SNjg3spvaJnmjB-LpxsPpLUaea33rvINL3Mq-gEaANErRQpbLx"
    },
    {
     "kty":"OKP","kid":"bc","use":"enc","crv":"X448","alg":"ECDH-ES",
     "x":"pK5IRpLlX-8XcsRYWHejpzkfsHoDOmAYuBzAC7aTpewWOw_QFHSa64t9p2kuommI8JQQLohS2AIA"
    }
   ]
  },
  "encrypted_response_enc_values_supported": ["A128GCM", "A128CBC-HS256"]
 }
}
```

A non-normative example response to the above request, having been encrypted to the first key, might look like the following (with added line breaks for display purposes only):

```
{
 "response" : "eyJhbGciOiJFQ0RILUVTIiwiZW5jIjoiQTEyOEdDTSIsImtpZCI6ImFjIiwiZXBrIjp7Imt
    0eSI6IkVDIiwieCI6Im5ubVZwbTNWM2piaGNhZlFhUkJrU1ZOSGx3Wkh3dC05ck9wSnVmeVlJdWsiLCJ5I
    joicjRmakRxd0p5czlxVU9QLV9iM21SNVNaRy0tQ3dPMm1pYzVWU05UWU45ZyIsImNydiI6IlAtMjU2In1
    9..uAYcHRUSSn2X0WPX.yVzlGSYG4qbg0bq18JcUiDRw56yVnbKR8E7S7YlEtzT00RqE3Pw5oTpUG3hdLN
    4taHZ9gC1kwak8JOnJgQ.1wR024_3-qtAlx1oFIUpQQ"
}
```

For illustrative purposes, the following JWK includes the private key `d` parameter value and can be used to decrypt the above encrypted Authorization Response example.

```
{
  "kty":"EC", "kid":"ac", "use":"enc", "crv":"P-256","alg":"ECDH-ES",
  "x":"YO4epjifD-KWeq1sL2tNmm36BhXnkJ0He-WqMYrp9Fk",
  "y":"Hekpm0zfK7C-YccH5iBjcIXgf6YdUvNUac_0At55Okk",
  "d":"Et-3ce0omz8_TuZ96Df9lp0GAaaDoUnDe6X-CRO7Aww"
}
```

The following shows the decoded header of the above encrypted Authorization Response example:

```
{
  "alg": "ECDH-ES",
  "enc": "A128GCM",
  "kid": "ac",
  "epk": {
    "kty": "EC",
    "x": "nnmVpm3V3jbhcafQaRBkSVNHlwZHwt-9rOpJufyYIuk",
    "y": "r4fjDqwJys9qUOP-_b3mR5SZG--CwO2mic5VSNTYN9g",
    "crv": "P-256"
  }
}
```

While this shows the payload of the above encrypted Authorization Response example:

```
{
 "vp_token": {"example_credential_id": ["eyJhb...YMetA"]}
}
```

Note that for the ECDH JWE algorithms (from Section 4.6 of [RFC7518]), the apu and apv values are inputs into the key derivation process that is used to derive the content encryption key. Regardless of the algorithm used, the values are always part of the AEAD tag computation so will still be bound to the encrypted response.

Note: For encryption, implementers have a variety of options available through JOSE, including the use of Hybrid Public Key Encryption (HPKE) as detailed in [I-D.ietf-jose-hpke-encrypt].

8.3.1. Response Mode "direct_post.jwt"

This specification also defines a new Response Mode `direct_post.jwt`, which allows for encryption to be used on top of the Response Mode `direct_post` defined in Section 8.2. The mechanisms described in Section 8.2 apply unless specified otherwise in this section.

The Response Mode `direct_post.jwt` causes the Wallet to send the Authorization Response using an HTTP POST request instead of redirecting back to the Verifier as defined in Section 8.2. The Wallet adds the `response` parameter containing the JWT as defined in Section 8.3 in the body of an HTTP POST request using the `application/x-www-form-urlencoded` content type. The names and values in the body MUST be encoded using UTF-8.

If a Wallet is unable to generate an encrypted response, it MAY send an error response without encryption as per Section 8.2.

The following is a non-normative example of a response (omitted content shown with ellipses for display purposes only):

```
POST /post HTTP/1.1
Host: client.example.org
Content-Type: application/x-www-form-urlencoded

response=eyJra...9t2LQ
```

The following is a non-normative example of the payload of the JWT used in the example above before encrypting and base64url encoding (omitted content shown with ellipses for display purposes only):

```
{
  "vp_token": {"example_jwt_vc": ["eY...QMA"]}
}
```

8.4. Transaction Data

The transaction data mechanism enables a binding between the user's identification/authentication and the user's authorization, for example to complete a payment transaction, or to sign specific document(s) using QES (Qualified Electronic Signatures). This is achieved by signing the transaction data used for user authorization with the user-controlled key used for proof of possession of the Credential being presented as a means for user identification/authentication.

The Wallet that received the `transaction_data` parameter in the request MUST include a representation or reference to the data in the respective Credential presentation. How this is done is transaction data type specific. Credential Formats can give recommendations of how to handle transaction data, such as those in Appendix B.

If the Wallet does not support `transaction_data` parameter, it MUST return an error upon receiving a request that includes it.

8.5. Error Response

The error response follows the rules as defined in [RFC6749], with the following additional clarifications:

`invalid_scope`:

- Requested scope value is invalid, unknown, or malformed.

`invalid_request`:

- The request contains both a `dcql_query` parameter and a `scope` parameter referencing a DCQL query.
- The request uses the `vp_token` Response Type but does not include a `dcql_query` parameter nor a `scope` parameter referencing a DCQL query.
- The Wallet does not support the Client Identifier Prefix passed in the Authorization Request.
- The Client Identifier passed in the request did not belong to its Client Identifier Prefix, or requirements of a certain prefix were violated, for example an unsigned request was sent with Client Identifier Prefix `https`.

`invalid_client`:

- `client_metadata` parameter defined in Section 5.1 is present, but the Wallet recognizes Client Identifier and knows metadata associated with it.
- Verifier's pre-registered metadata has been found based on the Client Identifier, but `client_metadata` parameter is also present.

`access_denied`:

- The Wallet did not have the requested Credentials to satisfy the Authorization Request.
- The End-User did not give consent to share the requested Credentials with the Verifier.
- The Wallet failed to authenticate the End-User.

This document also defines the following additional error codes and error descriptions:

`vp_formats_not_supported`:

- The Wallet does not support any of the formats requested by the Verifier, such as those included in the `vp_formats_supported` registration parameter.

`invalid_request_uri_method`:

- The value of the `request_uri_method` request parameter is neither `get` nor `post` (case-sensitive).

`invalid_transaction_data`:

- any of the following is true for at least one object in the `transaction_data` structure:
  - contains an unknown or unsupported transaction data type value,
  - is an object of a known type but containing unknown fields,
  - contains fields of the wrong type for the transaction data type,
  - contains fields with invalid values for the transaction data type,
  - is missing required fields for the transaction data type,
  - the credential_ids does not match, or
  - the referenced Credential(s) are not available in the Wallet.

`wallet_unavailable`:

- The Wallet appears to be unavailable and therefore unable to respond to the request. It can be useful in situations where the user agent cannot invoke the Wallet and another component receives the request while the End-User wishes to continue the journey on the Verifier website. For example, this applies when using claimed HTTPS URIs handled by the Wallet provider in case the platform cannot or does not translate the URI into a platform intent to invoke the Wallet. In this case, the Wallet provider would return the Authorization Error Response to the Verifier and might redirect the user agent back to the Verifier website.

8.6. VP Token Validation

Verifiers MUST validate the VP Token in the following manner:

1. Validate the format of the VP Token as defined in Section 8.1.
2. Check the individual Presentations according to the specific Credential Format requested:
   1. Validate the integrity and authenticity of the Presentation and Credential.
   2. Validate that the returned Credential(s) meet all criteria defined in the query in the Authorization Request (e.g., Claims included in the presentation).
   3. Validate that all Presentations contain a cryptographic proof of Holder Binding (i.e., that they are Verifiable Presentations), unless specifically requested otherwise.
   4. For Verifiable Presentations, validate the Holder Binding, including the checks required to prevent replay described in Section 14.1.
   5. Perform the checks required by the Verifier's policy based on the set of trust requirements such as trust frameworks it belongs to (e.g., revocation checks), if applicable.
3. Check that the set of Presentations returned satisfies all requirements defined in the Verifier's request as described in Section 6.4.

If any of the checks related to an individual Presentation fail, the effected Presentation MUST be discarded. If any of the checks pertaining to the VP Token or the overall response fails, the VP Token MUST be rejected.

9. Wallet Invocation

The Verifier can use one of the following mechanisms to invoke a Wallet:

- Custom URL scheme as an `authorization_endpoint` (for example, `openid4vp://` as defined in Section 13.1.2)
- URL (including Domain-bound Universal Links/App link) as an `authorization_endpoint`

For a cross device flow, either of the above options MAY be presented as a QR code for the End-User to scan using a Wallet or an arbitrary camera application on a user-device.

The Wallet can also be invoked from the web or a native app using the Digital Credentials API as described in Appendix A. As described in detail in Appendix A, DC API provides privacy, security (see Section 14.2), and user experience benefits (particularly in the cases where an End-User has multiple Wallets).

10. Wallet Metadata (Authorization Server Metadata)

This specification defines how the Verifier can determine Credential formats, proof types and algorithms supported by the Wallet to be used in a protocol exchange.

10.1. Additional Wallet Metadata Parameters

This specification defines new metadata parameters according to [RFC8414].

`vp_formats_supported`: REQUIRED. An object containing a list of name/value pairs, where the name is a Credential Format Identifier and the value defines format-specific parameters that a Wallet supports. For specific values that can be used, see Appendix B. Deployments can extend the formats supported, provided Issuers, Holders and Verifiers all understand the new format.

The following is a non-normative example of a `vp_formats_supported` parameter:

```
"vp_formats_supported": {
  "jwt_vc_json": {
    "alg_values": [
      "ES256K",
      "ES384"
    ]
  }
}
```

`client_id_prefixes_supported`: OPTIONAL. A non-empty array of strings containing the values of the Client Identifier Prefixes that the Wallet supports. The values defined by this specification are `pre-registered` (which represents the behavior when no Client Identifier Prefix is used), `redirect_uri`, `openid_federation`, `verifier_attestation`, `decentralized_identifier`, `x509_san_dns` and `x509_hash`. If omitted, the default value is `pre-registered`. Other values may be used when defined in the profiles or extensions of this specification.

Additional Wallet metadata parameters MAY be defined and used, as described in [RFC8414]. The Verifier MUST ignore any unrecognized parameters.

10.2. Obtaining Wallet's Metadata

A Verifier utilizing this specification has multiple options to obtain the Wallet's metadata:

- Verifier obtains the Wallet's metadata dynamically, e.g., using [RFC8414] or out-of-band mechanisms. See Section 10 for the details.
- Verifier has pre-obtained a static set of the Wallet's metadata. See Section 13.1.2 for the example.

11. Verifier Metadata (Client Metadata)

To convey Verifier metadata, Client metadata defined in Section 2 of [RFC7591] is used.

This specification defines how the Wallet can determine Credential formats, proof types and algorithms supported by the Verifier to be used in a protocol exchange.

11.1. Additional Verifier Metadata Parameters

This specification defines the following new Client metadata parameters according to [RFC7591], to be used by the Verifier:

`vp_formats_supported`: REQUIRED. An object containing a list of name/value pairs, where the name is a Credential Format Identifier and the value defines format-specific parameters that a Verifier supports. For specific values that can be used, see Appendix B. Deployments can extend the formats supported, provided Issuers, Holders and Verifiers all understand the new format.

Additional Verifier metadata parameters MAY be defined and used, as described in [RFC7591]. The Wallet MUST ignore any unrecognized parameters.

12. Verifier Attestation JWT

The Verifier Attestation JWT is a JWT especially designed to allow a Wallet to authenticate a Verifier in a secure and flexible manner. A Verifier Attestation JWT is issued to the Verifier by a party that Wallets trust for the purpose of authentication and authorization of Verifiers. The way this trust is established is out of scope of this specification. Every Verifier is bound to a public key, the Verifier MUST always present a Verifier Attestation JWT along with the proof of possession for this key. In the case of the Client Identifier Prefix `verifier_attestation`, the authorization request is signed with this key, which serves as proof of possession.

A Verifier Attestation JWT MUST contain the following claims:

- `iss`: REQUIRED. This claim identifies the issuer of the Verifier Attestation JWT. The `iss` value MAY be used to retrieve the issuer's public key. How the trust is established between Wallet and Issuer and how the public key is obtained for validating the attestation's signature is out of scope of this specification.
- `sub`: REQUIRED. The value of this claim MUST be the `client_id` of the client making the Credential request.
- `iat`: OPTIONAL. A number representing the time at which the Verifier Attestation JWT was issued using the syntax defined in [RFC7519].
- `exp`: REQUIRED. A number representing the time at which the Verifier Attestation JWT expires using the syntax defined in [RFC7519]. The Wallet MUST reject any Verifier Attestation JWT with an expiration time that has passed, subject to allowable clock skew between systems.
- `nbf`: OPTIONAL. A number representing the time before which the token MUST NOT be accepted for processing.

- `cnf`: REQUIRED. This claim contains the confirmation method as defined in [RFC7800]. It MUST contain a JSON Web Key [RFC7517] as defined in Section 3.2 of [RFC7800]. This claim determines the public key that the Verifier MUST prove possession of the corresponding private key for when presenting the Verifier Attestation JWT. This additional security measure allows the Verifier to obtain a Verifier Attestation JWT from a trusted issuer and use it for a long time independent of that issuer without the risk of an adversary impersonating the Verifier by replaying a captured attestation.

Additional claims MAY be defined and used in the Verifier Attestation JWT, as described in [RFC7519]. The Wallet MUST ignore any unrecognized claims.

Verifier Attestation JWTs compliant with this specification MUST use the media type `application/verifier-attestation+jwt` as defined in Appendix E.6.1.

A Verifier Attestation JWT MUST set the `typ` JOSE header to `verifier-attestation+jwt`.

The Verifier Attestation JWT MAY be conveyed in the header of a JWS signed object (JOSE header).

This specification introduces a JOSE header, which can be used to add a JWT to such a header as follows:

- `jwt`: This JOSE header MUST contain a JWT.

In the context of this specification, such a JWT MUST set the `typ` JOSE header to `verifier-attestation+jwt`.

13. Implementation Considerations

13.1. Static Configuration Values of the Wallets

This section lists profiles of this specification that define static configuration values for Wallets and defines one set of static configuration values that can be used by the Verifier when it is unable to perform Dynamic Discovery.

13.1.1. Profiles that Define Static Configuration Values

The following is a list of profiles that define static configuration values of Wallets:

- OpenID4VC High Assurance Interoperability Profile 1.0
- JWT VC Presentation Profile

13.1.2. A Set of Static Configuration Values bound to `openid4vp://`

The following is a non-normative example of a set of static configuration values that can be used with `vp_token` parameter as a supported Response Type, bound to a custom URL scheme `openid4vp://` as an Authorization Endpoint:

```
{
  "authorization_endpoint": "openid4vp:",
  "response_types_supported": [
    "vp_token"
  ],
  "vp_formats_supported": {
    "dc+sd-jwt": {
      "sd-jwt_alg_values": [
        "ES256"
      ],
      "kb-jwt_alg_values": [
        "ES256"
      ]
    },
    "mso_mdoc": {}
  },
  "request_object_signing_alg_values_supported": [
    "ES256"
  ]
}
```

13.2. Nested Presentations

This specification does not support presentation of a Presentation nested inside another Presentation.

13.3. Response Mode `direct_post`

The design of the interactions between the different components of the Verifier (especially Frontend and Response URI) when using Response Mode `direct_post` is at the discretion of the Verifier since it does not affect the interface between the Verifier and the Wallet.

In order to support implementers, this section outlines a possible design that fulfills the Security Considerations given in Section 14.

Figure 3 illustrates a sequence diagram of the design:

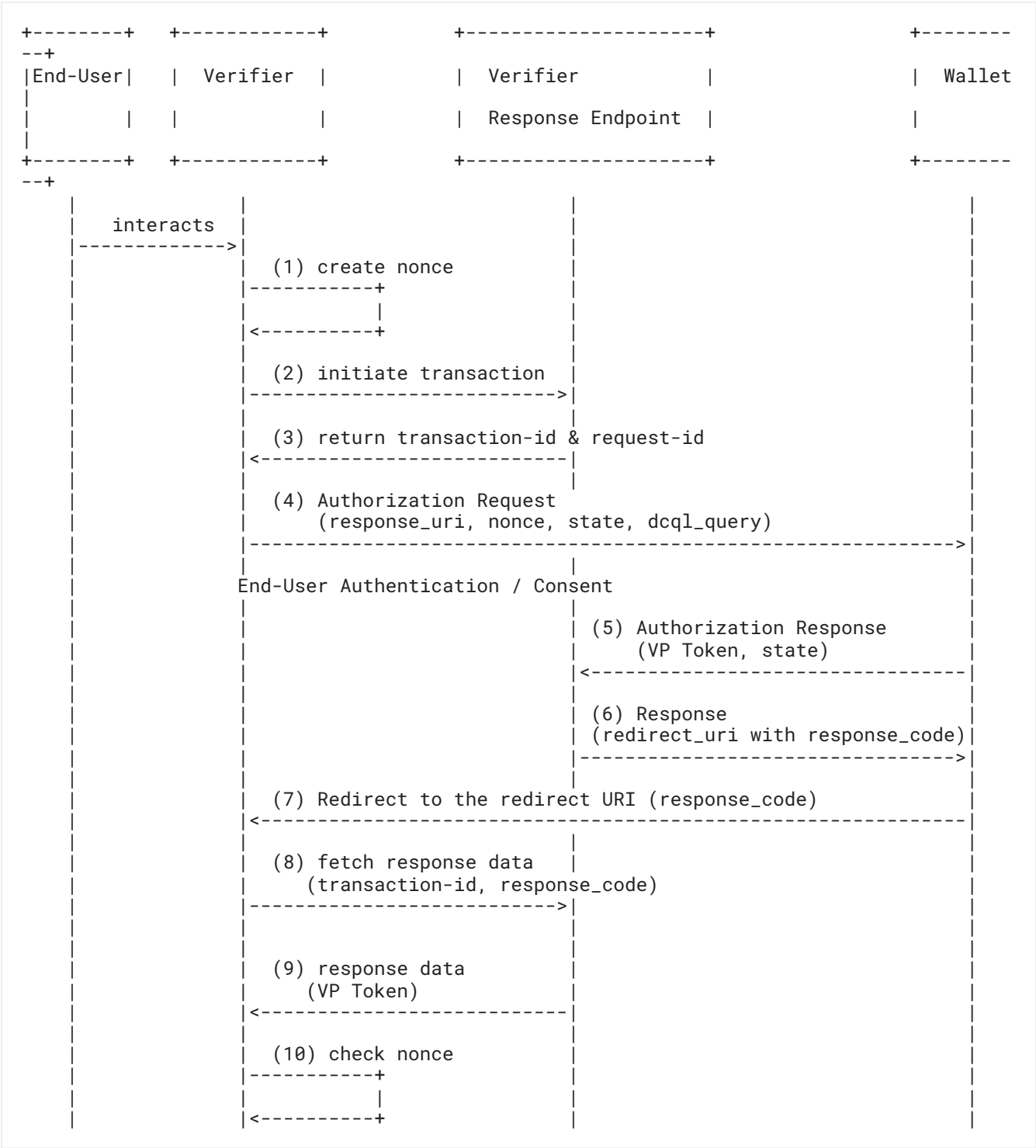

Figure 3: Reference Design for Response Mode `direct_post`

(1) The Verifier produces a nonce value by generating at least 16 fresh, cryptographically random bytes with sufficient entropy, associates it with the session and base64url encodes it.

(2) The Verifier initiates a new transaction at its Response URI.

(3) The Response URI will set up the transaction and respond with two fresh, cryptographically random numbers with sufficient entropy designated as `transaction-id` and `request-id`. Those values are used in the process to identify the authorization response (`request-id`) and to ensure only the Verifier can obtain the Authorization Response data (`transaction-id`).

(4) The Verifier then sends the Authorization Request with the `request-id` as `state` and the nonce value created in step (1) to the Wallet.

(5) After authenticating the End-User and getting their consent to share the request Credentials, the Wallet sends the Authorization Response with the parameters `vp_token` and `state` to the `response_uri` of the Verifier.

(6) The Verifier's Response URI checks whether the `state` value is a valid `request-id`. If so, it stores the Authorization Response data linked to the respective `transaction-id`. It then creates a `response_code` as fresh, cryptographically random number with sufficient entropy that it also links with the respective Authorization Response data. It then returns the `redirect_uri`, which includes the `response_code` to the Wallet.

Note: If the Verifier's Response URI does not return a `redirect_uri`, processing at the Wallet stops at that step. The Verifier is supposed to fetch the Authorization Response without waiting for a redirect (see step 8).

(7) The Wallet sends the user agent to the Verifier (`redirect_uri`). The Verifier receives the Request and extracts the `response_code` parameter.

(8) The Verifier sends the `response_code` and the `transaction-id` from its session to the Response URI.

- The Response URI uses the `transaction-id` to look the matching Authorization Response data up, which implicitly validates the `transaction-id` associated with the Verifier's session.
- If an Authorization Response is found, the Response URI checks whether the `response_code` was associated with this Authorization Response in step (6).

Note: If the Verifier's Response URI did not return a `redirect_uri` in step (6), the Verifier will periodically query the Response URI with the `transaction-id` to obtain the Authorization Response once it becomes available.

(9) The Response URI returns the VP Token for further processing to the Verifier.

(10) The Verifier checks whether the nonce received in the Credential(s) in the VP Token in step (9) corresponds to the nonce value from the session. The Verifier then consumes the VP Token and invalidates the `transaction-id`, `request-id` and nonce in the session.

13.4. Pre-Final Specifications

Implementers should be aware that this specification uses several specifications that are not yet final specifications. Those specifications are:

- OpenID Federation 1.0 draft -43 [OpenID.Federation]
- SIOPv2 draft -13 [SIOPv2]
- Selective Disclosure for JWTs (SD-JWT) draft -22 [I-D.ietf-oauth-selective-disclosure-jwt]
- SD-JWT-based Verifiable Credentials (SD-JWT VC) draft -09 [I-D.ietf-oauth-sd-jwt-vc]
- Fully-Specified Algorithms for JOSE and COSE draft -13 [I-D.ietf-jose-fully-specified-algorithms]

While breaking changes to the specifications referenced in this specification are not expected, should they occur, OpenID4VP implementations should continue to use the specifically referenced versions above in preference to the final versions, unless updated by a profile or new version of this specification.

14. Security Considerations

14.1. Preventing Replay of Verifiable Presentations

An attacker could try to inject Presentations obtained from (for example) a previous Authorization Response into another Authorization Response, thus impersonating the End-User that originally presented the respective Verifiable Presentation. Holder Binding aims to prevent such attacks.

14.1.1. Presentations without Holder Binding Proofs

By definition, Presentations without Holder Binding (see Section 5.3) do not provide protection against replay. A Verifier that consumes Presentations without Holder Binding accepts the risk that the Holder may have obtained the Credential from a third party (e.g., by playing the role of a Verifier) and that the Holder may not be the subject of the Credential.

Depending on the use case, the risk assessment of the Verifier, and external validation measures that can be taken, this risk may be acceptable.

14.1.2. Verifiable Presentations

For Verifiable Presentations, implementers of this specification MUST implement the controls as defined in this section to detect and prevent replay attacks.

The cryptographic proof of possession in a Verifiable Presentation MUST be bound by the Wallet to the intended audience (the Client Identifier of the Verifier) and the respective transaction (identified by the `nonce` parameter in the Authorization Request, as defined in Section 5.2). The Verifier MUST verify this binding.

The Wallet MUST link every Verifiable Presentation returned to the Verifier in the VP Token to the `client_id` and the `nonce` values of the respective Authentication Request.

The Verifier MUST validate every individual Verifiable Presentation in an Authorization Response and ensure that it is linked to the values of the `client_id` and the `nonce` parameter it had used for the respective Authorization Request. If any Verifiable Presentation in the response does not contain the correct `nonce` value, the response MUST be rejected.

The `client_id` is used to detect the replay of Verifiable Presentations to a party other than the one intended. This allows Verifiers to reject the Verifiable Presentation. The `nonce` value binds the Verifiable Presentation to a certain authentication transaction and allows the Verifier to detect injection of a Presentation in the flow, which is especially important in the flows where the Presentation is passed through the front-channel.

Note: Different formats for Verifiable Presentations and signature/proof schemes use different ways to represent the intended audience and the session binding. Some use claims to directly represent those values, others include the values into the calculation of cryptographic proofs. There are also different naming conventions across the different formats. The format of the respective presentation is defined by the Verifier in the request.

The following is a non-normative example of the payload of a Verifiable Presentation following a request with the Credential Format Identifier `jwt_vc_json`:

```
{
  "iss": "did:example:ebfeb1f712ebc6f1c276e12ec21",
  "jti": "urn:uuid:3978344f-8596-4c3a-a978-8fcaba3903c5",
  "aud": "s6BhdRkqt3",
  "nonce": "343s$FSFDa-",
  "nbf": 1541493724,
  "iat": 1541493724,
  "exp": 1573029723,
  "vp": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "type": ["VerifiablePresentation"],

    "verifiableCredential": [""]
  }
}
```

In the example above, the requested `nonce` value is included as the `nonce` and `client_id` as the `aud` value in the proof of the Verifiable Presentation.

The following is a non-normative example of a Verifiable Presentation following a request with the Credential Format Identifier `ldp_vc` without a `proof` property:

```
{
  "@context": [ ... ],
  "type": "VerifiablePresentation",
  "verifiableCredential": [ ... ],
  "proof": {
    "type": "DataIntegrityProof",
    "cryptosuite": "ecdsa-rdfc-2019",
    "created": "2018-09-14T21:19:10Z",
    "proofPurpose": "authentication",
    "verificationMethod": "did:example:ebfeb1f712ebc6f1c276e12ec21#keys-1",
    "challenge": "343s$FSFDa-",
    "domain": "x509_san_dns:client.example.org",
    "proofValue": "z2iAR...3oj9Q8"
  }
}
```

In the example above, the requested `nonce` value is included as the `challenge` and `client_id` as the `domain` value in the proof of the Verifiable Presentation.

14.2. Session Fixation

To perform a session fixation attack, an attacker would start the process using a Verifier on a device under their control, capture the Authorization Request, and relay it to the device of a victim. The attacker would then periodically try to conclude the process in their Verifier, which would cause the Verifier on their device to try to fetch and verify the Authorization Response.

Such an attack is impossible against flows implemented with the Response Mode `fragment` as the Wallet will always send the VP Token to the redirect endpoint on the same device where it resides. This means an attacker could extract a valid Authorization Request from a Verifier on their device and trick a Victim into performing the same Authorization Request on the victim's device. But there is usually no way for an attacker to get hold of the resulting VP Token.

However, the Response Mode `direct_post` is susceptible to such an attack as the result is sent from the Wallet out-of-band to the Verifier's Response URI.

This kind of attack can be detected if the Response Mode `direct_post` is used in conjunction with the redirect URI, which causes the Wallet to redirect the flow to the Verifier's frontend at the device where the transaction was concluded. The Verifier's Response URI MUST include a fresh secret (Response Code) into the redirect URI returned to the Wallet and the Verifier's Response URI MUST require the frontend to pass the respective Response Code when fetching the Authorization Response. That stops session fixation attacks as long as the attacker is unable to get access to the Response Code.

Note that this protection technique is not applicable to cross-device scenarios because the browser used by the Wallet will not have the original session. It is also not applicable in same-device scenarios if the Wallet uses a browser different from the one used on the presentation request (e.g. device with multiple installed browsers), because the original session will also not be available there. Appendix A provides an alternative Wallet invocation method using web/app platform APIs that avoids many of these issues.

See Section 13.3 for more implementation considerations.

When using the Response Mode `direct_post` without the further protection provided by the redirect URI, there is no session context for the Verifier to detect session fixation attempts. It is RECOMMENDED for the Verifiers to implement mechanisms to strengthen the security of the flow. For more details on possible attacks and mitigations see [I-D.ietf-oauth-cross-device-security].

14.3. Response Mode "direct_post"

14.3.1. Validation of the Response URI

The Wallet MUST ensure the data in the Authorization Response cannot leak through Response URIs. When using pre-registered Response URIs, the Wallet MUST comply with best practices for redirect URI validation as defined in [RFC9700]. The Wallet MAY also rely on a Client Identifier Prefix in conjunction with Client Authentication and integrity protection of the request to establish trust in the Response URI provided by a certain Verifier.

14.3.2. Protection of the Response URI

The Verifier SHOULD protect its Response URI from inadvertent requests by checking that the value of the received `state` parameter corresponds to a recent Authorization Request.

14.3.3. Protection of the Authorization Response Data

This specification assumes that the Verifier's Response URI offers an internal interface to other components of the Verifier to obtain (and subsequently process) Authorization Response data. An attacker could try to obtain Authorization Response Data from a Verifier's Response URI by looking up this data through the internal interface. This could lead to leakage of valid Presentations containing personally identifiable information.

Implementations of this specification MUST have security mechanisms in place to prevent inadvertent requests against this internal interface. Implementation options to fulfill this requirement include:

- Authentication between the different parts within the Verifier
- Two cryptographically random numbers. The first being used to manage state between the Wallet and Verifier. The second being used to ensure that only a legitimate component of the Verifier can obtain the Authorization Response data.

14.4. End-User Authentication using Credentials

Clients intending to authenticate the End-User utilizing a claim in a Credential MUST ensure this claim is stable for the End-User as well as locally unique and never reassigned within the Credential Issuer to another End-User. Such a claim MUST also only be used in combination with the Credential Issuer identifier to ensure global uniqueness and to prevent attacks where an attacker obtains the same claim from a different Credential Issuer and tries to impersonate the legitimate End-User.

14.5. Encrypting an Unsigned Response

Because an encrypted Authorization Response has no additional integrity protection, an attacker might be able to alter Authorization Response parameters and generate a new encrypted Authorization Response for the Verifier, as encryption is performed using the public key of the Verifier (which is likely to be widely known when not ephemeral to the request/response). Note this includes injecting a new VP Token. Since the contents of the VP Token are integrity protected, tampering with the VP Token is detectable by the Verifier. For details, see Section 14.1.

14.6. TLS Requirements

Implementations MUST follow [BCP195].

Whenever TLS is used, a TLS server certificate check MUST be performed, per [RFC6125].

14.7. Incomplete or Incorrect Implementations of the Specifications and Conformance Testing

To achieve the full security benefits, it is important that the implementation of this specification, and the underlying specifications, are both complete and correct.

The OpenID Foundation provides tools that can be used to confirm that an implementation is correct and conformant:

https://openid.net/certification/conformance-testing-for-openid-for-verifiable-presentations/

14.8. Always Use the Full Client Identifier

Confusing Verifiers using a Client Identifier Prefix with those using none can lead to attacks. Therefore, Wallets MUST always use the full Client Identifier, including the prefix if provided, within the context of the Wallet or its responses to identify the client. This refers in particular to places where the Client Identifier is used in [RFC6749] and in the presentation returned to the Verifier.

14.9. Security Checks on the Returned Credentials and Presentations

While the Verifier can specify various constraints both on the claims level and the Credential level as shown in Section 6.4, it MUST NOT rely on the Wallet to enforce these constraints. The Wallet is not controlled by the Verifier and the Verifier MUST perform its own security checks on the returned Credentials and Presentations.

15. Privacy Considerations

Many privacy considerations are specific to the Credential format and associated proof type used in a particular Presentation.

This section focuses on privacy considerations specific to the presentation protocol while also addressing cross-cutting concerns related to credential formats, Wallet behavior, and Verifier practices.

Wallet providers and Verifiers need to take into account privacy considerations in this section to mitigate the risks of data leakage, user tracking, and other privacy harms.

15.1. User Consent

Wallets SHOULD obtain explicit, informed consent from the End-User before releasing any Verifiable Credential or Presentation to a Verifier, or returning an error.

Transaction history and data within the Wallet SHOULD NOT be accessible to anyone other than the End-User, unless the End-User has given consent or there is another legal basis to do so.

15.2. Privacy Notice

Wallets SHOULD make their privacy notices readily available to the End-User.

15.3. Purpose Legitimacy

The Verifier SHOULD ensure that the purpose for collecting the information it is requesting is sufficiently specific and communicated before collection. For example, the purpose is shown to the End-User before or within the presentation request that is sent to the Wallet.

If the Wallet has indications that the Verifier is requesting data that it is not entitled to, the Wallet SHOULD warn the End-User or potentially stop processing.

15.4. Selective Disclosure

Selective disclosure is a data minimization technique that allows for sharing only the specific information needed from a Credential without revealing all of the claims contained in that Credential.

The DCQL helps facilitate selective disclosure by allowing the Verifier to specify the claims it is interested in, allowing the Wallet to disclose only the claims that are relevant to the Verifier's request.

Some Credential formats support selective disclosure and a salted-hash based approach is one common approach.

15.4.1. DCQL Value Matching

When using DCQL `values` to match the expected values of claims, the fact that a claim within a certain Credential matched a value or did not match a value might already leak information about the claim value. Therefore, Wallets MUST take precautions against leaking information about the claim value when processing `values`. This SHOULD include, in particular:

- ensuring that a Verifier, in the response, cannot distinguish between the case where an End-User did not consent to releasing the Credential and the case where the claim value did not match the expected value, and
- preventing repeated or "silent" requests leaking data to the Verifier without the user's consent by ensuring that all requests, even if no response can be sent by the Wallet due to a `values` mismatch, require some form of End-User interaction before a response is sent.

In both cases listed here, it needs to be considered that returning an error response can also leak information about the processing outcome of `values`.

15.4.2. Strictly Necessary Claims

Verifiers SHOULD use DCQL queries that request only the minimal set of claims and Credentials needed to fulfill the specified purposes.

15.5. Verifier-to-Verifier Unlinkable Presentations

Even when using selective disclosure to reveal limited claims from a Credential to a Verifier, there are ways in which a Presentation could be linked to another Presentation in another session or a Presentation to another Verifier. For example, with Credential formats such as SD-JWT and mdoc, the Issuer signature on a Credential or the public key a Credential is bound to, can provide a Verifier with a way to link the Credential across different Presentations or sessions. In order to avoid such linking, a Wallet can use multiple instances of a Credential, each with unique Issuer signatures and associated public keys to limit this:

- a Wallet can use an issued Credential instance only once in a Presentation to a specific Verifier, before discarding the Credential, thus avoiding linking on the above basis ever occurring
- a Wallet can apply a limited use policy for a specific instance of a Credential, perhaps only allowing it to be presented to the same Verifier to avoid Verifier to Verifier linkability

Considerable discourse regarding unlinkability in salted-hash based selective disclosure mechanisms is provided in Section 10.1 of [I-D.ietf-oauth-selective-disclosure-jwt]. One technique mentioned to achieve some important unlinkability properties is the use of batch issuance, which is supported in [OpenID4VCI], with individual Credentials being presented only once.

15.6. No Fingerprinting of the End-User

A Verifier SHOULD NOT attempt to fingerprint the End-User based on metadata that may be available in the interaction with the End-User's wallet.

A Wallet SHOULD implement measures that prevent fingerprinting of the End-Users during the request to resolve the Request Object URI.

A Wallet SHOULD implement measures that limit unintended additional information being disclosed through the Response URI. For example, disclosing Wallet-related information through the HTTP user agent header.

15.7. Information Security

Both Wallet providers and Verifiers SHOULD apply suitable security controls at the operational, functional, and strategic level to ensure the integrity, confidentiality and general handling of PII. Furthermore, they should consider protections against risks such as unauthorized access, destruction, use, modification, disclosure or loss throughout the whole of its life cycle.

15.8. Wallet to Verifier Communication

Wallets SHOULD only send the minimal amount of information possible, and in particular, avoid sending any additional HTTP headers identifying the software used for the request (e.g., HTTP libraries or their versions) when retrieving a `request_uri` or sending to `response_uri` to reduce the risk of fingerprinting and End-User tracking.

Wallets MUST NOT include any personally identifiable information (PII) in HTTP requests to Verifiers unless explicitly required for the flow and authorized by the End-User.

15.8.1. Establishing Trust in the Request URI

Wallets operating within a trust framework SHOULD validate that the Request URI is properly associated with the Client Identifier and authorized for the request.

Untrusted or unrecognized Request URI endpoints SHOULD be rejected or require End-User confirmation before proceeding.

15.8.2. Authorization Requests with Request URI

If the Wallet is acting within a trust framework that allows the Wallet to determine whether a Request URI belongs to a certain Client Identifier, the Wallet is RECOMMENDED to validate the Verifier's authenticity and authorization given by the Client Identifier and that the Request URI corresponds to this Verifier. If the link cannot be established in those cases, the Wallet MUST refuse the request.

15.9. Error Responses

Error responses SHOULD avoid including sensitive or detailed contextual information that could be used to infer the End-User's data.

15.9.1. `wallet_unavailable` Authorization Error Response

In the event that another component is invoked instead of the Wallet, the End-User SHOULD be informed and give consent before the invoked component returns the `wallet_unavailable` Authorization Error Response to the Verifier.

15.9.2. Digital Credential API Error Responses

Returning any OpenID4VP protocol error, regardless of content, can reveal additional information about the End-User's underlying Credentials or Wallet in a way that is unique to the Digital Credentials API since reaching the Wallet can be dependent on a Wallet's ability to satisfy the request. For example, platform implementations could only allow Wallets to be selected that satisfy the request. In this case, OpenID4VP protocol error responses can only be returned by a selected Wallet and would therefore reveal that the End-User is in possession of Credentials that satisfy the request. This is in contrast to other engagement methods, in which the Wallet receives the request before learning if it can be fulfilled. What is revealed by a Wallet in those cases depends on how each individual Wallet processes the request.

The narrower a request is, the more information is revealed:

- A request that can be fulfilled by a broad range of documents will only reveal that the End-User has a Credential from a large set of documents.
- A request for a single document type will reveal the End-User is in possession of that Credential. How sensitive this is would depend on the particular Credential.
- A request with which can only be satisfied by a single trusted authority will reveal that the End-User has a Credential from a particular authority, from which other attributes may be inferred.
- A request with value matching (as defined in Section 6.4.1) will reveal the specific value of that claim/attribute.

Wallet implementations need to balance the value of error detection to the maintenance and scaling of the Verifier ecosystem with the information that is revealed.

A Wallet SHOULD NOT return any OpenID4VP protocol errors without End-User interaction either with the platform or the Wallet. When handling errors, implementations can opt to cancel the flow (the details of which are platform specific) rather than return an OpenID4VP protocol-specific error. This will make the result indistinguishable from other platform aborts, preventing any information from being revealed.

A Wallet SHOULD NOT return any OpenID4VP protocol errors before obtaining End-User consent, when processing a request containing value matching (to avoid revealing values of claims without consent), or issuer selection (to avoid revealing that the End-User has a Credential from a particular authority). Additionally, the End-User consent protects against undetected, repeated requests to the Wallet.

15.10. Establishing Trust in the Issuers

This specification introduces an extension point that allows for a Verifier to express expected Issuers or trust frameworks that certify Issuers. It is important to understand the implications of these trust establishment mechanisms on the privacy of the overall system.

In general, two types of mechanisms can be distinguished: those that are self-contained, where the Wallet and Verifier already have all the information needed to check if a Credential satisfies the request, and those that depend on online resolution to obtain additional data. Mechanisms that require online resolution can leak information that could be used to profile the usage of the Credentials.

In particular, situations where the Wallet must fetch data before it can generate a matching presentation may expose information about individual End-Users to external parties.

Wallets SHOULD NOT access URLs included in a request from the Verifier if those URLs are unfamiliar or hosted by untrusted third parties. Privacy risks can be reduced if such URLs are treated purely as identifiers and not actually retrieved by the Wallet upon receiving the request.

Ecosystems intending to use trusted authority mechanisms SHOULD ensure that the privacy characteristics of their chosen mechanisms align with the overall privacy goals of the ecosystem.

16. Normative References

[BCP195] IETF, "BCP195", 2022, <https://www.rfc-editor.org/info/bcp195>.

[DID-Core] Sporny, M., Guy, A., Sabadello, M., and D. Reed, "Decentralized Identifiers (DIDs) v1.0", 19 July 2022, <https://www.w3.org/TR/2022/REC-did-core-20220719/>.

[I-D.ietf-jose-fully-specified-algorithms] Jones, M. B. and O. Steele, "Fully-Specified Algorithms for JOSE and COSE", Work in Progress, Internet-Draft, draft-ietf-jose-fully-specified-algorithms-13, 11 May 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-jose-fully-specified-algorithms-13>.

[I-D.ietf-oauth-sd-jwt-vc] Terbu, O., Fett, D., and B. Campbell, "SD-JWT-based Verifiable Credentials (SD-JWT VC)", Work in Progress, Internet-Draft, draft-ietf-oauth-sd-jwt-vc-10, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-sd-jwt-vc-10>.

[I-D.ietf-oauth-selective-disclosure-jwt] Fett, D., Yasuda, K., and B. Campbell, "Selective Disclosure for JWTs (SD-JWT)", Work in Progress, Internet-Draft, draft-ietf-oauth-selective-disclosure-jwt-22, 29 May 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-selective-disclosure-jwt-22>.

[JSON-LD] Kellogg, G., Champin, P., and D. Longley, "JSON-LD 1.1", 16 July 2020, <https://www.w3.org/TR/json-ld11/>.

[OAuth.Responses] de Medeiros, B., Scurtescu, M., Tarjan, P., and M. Jones, "OAuth 2.0 Multiple Response Type Encoding Practices", 25 February 2014, <https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html>.

[OpenID.Core] Sakimura, N., Bradley, J., Jones, M.B., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 2", 15 December 2023, <https://openid.net/specs/openid-connect-core-1_0.html>.

[OpenID.Federation] Ed., R. H., Jones, M. B., Solberg, A., Bradley, J., Marco, G. D., and V. Dzhuvinov, "OpenID Federation 1.0", 2 June 2025, <https://openid.net/specs/openid-federation-1_0-43.html>.

[OpenID4VCI] Lodderstedt, T., Yasuda, K., and T. Looker, "OpenID for Verifiable Credential Issuance 1.0 - draft 16", 26 June 2025, <https://openid.net/specs/openid-4-verifiable-credential-issuance-1_0.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/info/rfc6125>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[SIOPv2] Yasuda, K., Jones, M. B., and T. Lodderstedt, "Self-Issued OpenID Provider V2", 28 November 2023, <https://openid.net/specs/openid-connect-self-issued-v2-1_0-13.html>.

[W3C.Digital_Credentials_API] Caceres, M., Cappalli, T., and M. A. Yosef, "Digital Credentials API", 1 July 2025, <https://www.w3.org/TR/2025/WD-digital-credentials-20250701/>.

17. Informative References

[ETSI.TL] European Telecommunications Standards Institute (ETSI), "ETSI TS 119 612 V2.1.1 Electronic Signatures and Infrastructures (ESI); Trusted Lists", 2015, <https://www.etsi.org/deliver/etsi_ts/119600_119699/119612/02.01.01_60/ts_119612v020101p.pdf>.

[I-D.ietf-jose-hpke-encrypt] Reddy.K, T., Tschofenig, H., Banerjee, A., Steele, O., and M. B. Jones, "Use of Hybrid Public Key Encryption (HPKE) with JSON Object Signing and Encryption (JOSE)", Work in Progress, Internet-Draft, draft-ietf-jose-hpke-encrypt-11, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-jose-hpke-encrypt-11>.

[I-D.ietf-oauth-cross-device-security] Kasselman, P., Fett, D., and F. Skokan, "Cross-Device Flows: Security Best Current Practice", Work in Progress, Internet-Draft, draft-ietf-oauth-cross-device-security-10, 17 June 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-cross-device-security-10>.

[IANA.COSE] IANA, "CBOR Object Signing and Encryption (COSE)", <https://www.iana.org/assignments/cose>.

[IANA.Hash.Algorithms] IANA, "Named Information Hash Algorithm Registry", <https://www.iana.org/assignments/named-information/named-information.xhtml>.

[IANA.JOSE] IANA, "JSON Object Signing and Encryption (JOSE)", <https://www.iana.org/assignments/jose>.

[IANA.OAuth.Parameters] IANA, "OAuth Parameters", <https://www.iana.org/assignments/oauth-parameters>.

[IANA.URI.Schemes] IANA, "Uniform Resource Identifier (URI) Schemes", <https://www.iana.org/assignments/uri-schemes>.

[ISO.18013-5] ISO/IEC JTC 1/SC 17 Cards and security devices for personal identification, "ISO/IEC 18013-5:2021 Personal identification — ISO-compliant driving license — Part 5: Mobile driving license (mDL) application", 2021, <https://www.iso.org/standard/69084.html>.

[ISO.23220-2] ISO/IEC JTC 1/SC 17 Cards and security devices for personal identification, "ISO/IEC TS 23220-2 Personal identification — Building blocks for identity management via mobile devices, Part 2: Data objects and encoding rules for generic eID systems", 2024, <https://www.iso.org/standard/86782.html>.

[ISO.23220-4] ISO/IEC JTC 1/SC 17 Cards and security devices for personal identification, "ISO/IEC CD TS 23220-4 Personal identification — Building blocks for identity management via mobile devices, Part 4: Protocols and services for operational phase", 2025, <https://www.iso.org/standard/86785.html>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC9101] Sakimura, N., Bradley, J., and M. Jones, "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)", RFC 9101, DOI 10.17487/RFC9101, August 2021, <https://www.rfc-editor.org/info/rfc9101>.

[RFC9207] Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", RFC 9207, DOI 10.17487/RFC9207, March 2022, <https://www.rfc-editor.org/info/rfc9207>.

[RFC9700] Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "Best Current Practice for OAuth 2.0 Security", BCP 240, RFC 9700, DOI 10.17487/RFC9700, January 2025, <https://www.rfc-editor.org/info/rfc9700>.

[VC_DATA] Sporny, M., Noble, G., Longley, D., Burnett, D. C., Zundel, B., and D. Chadwick, "Verifiable Credentials Data Model 1.1", 3 March 2022, <https://www.w3.org/TR/2022/REC-vc-data-model-20220303/>.

[VC_DATA_INTEGRITY] Sporny, M., Jr, T. T., Herman, I., Longley, D., and G. Bernstein, "Verifiable Credential Data Integrity 1.0", 29 March 2025, <https://www.w3.org/TR/2025/PR-vc-data-integrity-20250320/>.

Appendix A. OpenID4VP over the Digital Credentials API

This section defines how to use OpenID4VP with the Digital Credentials API.

The name "Digital Credentials API" (DC API) encompasses the W3C Digital Credentials API [W3C.Digital_Credentials_API] as well as its native App Platform equivalents in operating systems (such as Credential Manager on Android). The DC API allows web sites and native apps acting as Verifiers to request the presentation of Credentials. The API itself is agnostic to the Credential exchange protocol and can be used with different protocols. The Web Platform, working in conjunction with other layers, such as the app platform/operating system, and based on the permission of the End-User, will send the request data along with the Origin of the Verifier to the End-User's chosen Wallet.

OpenID4VP over the DC API utilizes the mechanisms of the DC API while also allowing to leverage advanced security features of OpenID4VP, if needed. It also defines the OpenID4VP request parameters that MAY be used with the DC API.

The DC API offers several advantages for implementers of both Verifiers and Wallets.

Firstly, the API serves as a privacy-preserving alternative to invoking Wallets via URLs, particularly custom URL schemes. The underlying app platform will only invoke a Wallet if the End-User confirms the request based on contextual information about the Credential Request and the requestor (Verifier).

Secondly, the session with the End-User will always continue in the initial context, typically a web browser tab, when the request has been fulfilled (or aborted), which results in an improved End-User experience.

Thirdly, cross-device requests benefit from the use of secure transports with proximity checks, which are handled by the OS platform, e.g., using FIDO CTAP 2.2 with hybrid transports.

And lastly, as part of the request, the Wallet is provided with information about the Verifier's Origin as authenticated by the user agent, which is important for phishing resistance.

A.1. Protocol

To use OpenID4VP with the Digital Credentials API (DC API), the exchange protocol value has the following format: `openid4vp-v<version>-<request-type>`. The `<version>` field is a numeric value, and `<request-type>` explicitly specifies the type of request. This approach eliminates the need for Wallets to perform implicit parameter matching to accurately identify the version and the expected request and response parameters.

The value 1 MUST be used for the `<version>` field to indicate the request and response are compatible with this version of the specification. For `<request-type>`, unsigned requests, as defined in Appendix A.3.1, MUST use `unsigned`, signed requests, as defined in Appendix A.3.2.1, MUST use `signed`, and multi-signed requests, as defined in Appendix A.3.2.2, MUST use `multisigned`.

The following exchange protocol values are defined by this specification:

- Unsigned requests: `openid4vp-v1-unsigned`
- Signed requests (JWS Compact Serialization): `openid4vp-v1-signed`
- Multi-signed requests (JWS JSON Serialization): `openid4vp-v1-multisigned`

A.2. Request

The Verifier MAY send a request, as defined in Section 5, to the DC API.

The following is a non-normative example of an unsigned OpenID4VP request (when advanced security features of OpenID4VP are not used) that can be sent over the DC API:

```
{
  response_type: "vp_token",
  response_mode: "dc_api",
  nonce: "n-0S6_WzA2Mj",
  client_metadata: {...},
  dcql_query: {...}
}
```

Out of the Authorization Request parameters defined in [RFC6749] and Section 5, the following are supported with OpenID4VP over the W3C Digital Credentials API:

- `client_id`
- `response_type`
- `response_mode`
- `nonce`
- `client_metadata`
- `request`
- `transaction_data`
- `dcql_query`
- `verifier_info`

The `client_id` parameter MUST be omitted in unsigned requests defined in Appendix A.3.1. The Wallet MUST ignore any `client_id` parameter that is present in an unsigned request.

Parameters defined by a specific Client Identifier Prefix (such as the `trust_chain` parameter for the OpenID Federation Client Identifier Prefix) are also supported over the W3C Digital Credentials API.

The `client_id` parameter MUST be present in signed requests defined in Appendix A.3.2, as it communicates to the Wallet which Client Identifier Prefix and Client Identifier to use when authenticating the client through verification of the request signature or retrieving client metadata. The value of the `response_mode` parameter MUST be `dc_api` when the response is not encrypted and `dc_api.jwt` when the response is encrypted as defined in Section 8.3. The Response Mode `dc_api` causes the Wallet to send the Authorization Response via the DC API. For Response Mode `dc_api.jwt`, the Wallet includes the `response` parameter, which contains an encrypted JWT encapsulating the Authorization Response, as defined in Section 8.3.

In addition to the above-mentioned parameters, a new parameter is introduced for OpenID4VP over the W3C Digital Credentials API:

- `expected_origins`: REQUIRED when signed requests defined in Appendix A.3.2 are used with the Digital Credentials API (DC API). A non-empty array of strings, each string representing an Origin of the Verifier that is making the request. The Wallet MUST compare values in this parameter to the Origin to detect replay of the request from a malicious Verifier. If the Origin does not match any of the entries in `expected_origins`, the Wallet MUST return an error. This error SHOULD be an `invalid_request` error. This parameter is not for use in unsigned requests and therefore a Wallet MUST ignore this parameter if it is present in an unsigned request.

The transport of the request and Origin to the Wallet is platform-specific and is out of scope of OpenID4VP over the Digital Credentials API.

Additional request parameters MAY be defined and used with OpenID4VP over the DC API.

The Wallet MUST ignore any unrecognized parameters. For example, since the `state` parameter is not defined for the DC API, the Verifier cannot expect it to be included in the response.

A.3. Signed and Unsigned Requests

Any OpenID4VP request compliant to this section of this specification can be used with the Digital Credentials API (DC API). Depending on the mechanism used to identify and authenticate the Verifier, the request can be signed or unsigned. This section defines signed and unsigned OpenID4VP requests for use with the DC API.

A.3.1. Unsigned Request

The Verifier MAY send all the OpenID4VP request parameters as members in the request member passed to the API.

A.3.2. Signed Request

The Verifier MAY send a signed request, for example, when identification and authentication of the Verifier is required.

The signed request allows the Wallet to authenticate the Verifier using one or more trust framework(s) in addition to the Web PKI utilized by the browser. An example of such a trust framework is the Verifier (RP) management infrastructure set up in the context of the eIDAS regulation in the European Union, in which case, the Wallet can no longer rely only on the web origin of the Verifier. This web origin MAY still be used to further strengthen the security of the flow. The external trust framework could, for example, map the Client Identifier to registered web origins.

The signed Request Object MAY contain all the parameters listed in Appendix A.2, except `request`.

Verifiers SHOULD format signed Requests using JWS Compact Serialization but MAY use JWS JSON Serialization ([RFC7515]) to cater for the use cases described below.

A.3.2.1. JWS Compact Serialization

When the JWS Compact Serialization is used to send the request, the Verifier can convey only one Trust Framework, i.e., the Verifier knows which trust frameworks the Wallet supports. All request parameters are encoded in a request object as defined in Section 5 and the JWS object is used as the value of the `request` claim in the `data` element of the API call.

This is illustrated in the following non-normative example.

```
{ request: "eyJhbGciOiJF..." }
```

This is an example of the payload of a signed OpenID4VP request used with the W3C Digital Credentials API in conjunction with JWS Compact Serialization:

```
{
  "expected_origins": [
    "https://origin1.example.com",
    "https://origin2.example.com"
  ],
  "client_id": "x509_san_dns:rp.example.com",
  "client_metadata": {
    "jwks": {
      "keys": [
        {
          "kty": "EC",
          "crv": "P-256",
          "x": "MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
          "y": "4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
          "use": "enc",
          "kid": "1"
        }
      ]
    }
  },
  "response_type": "vp_token",
  "response_mode": "dc_api",
  "nonce": "n-0S6_WzA2Mj",
  "dcql_query": {...}
}
```

A.3.2.2. JWS JSON Serialization

The JWS JSON Serialization ([RFC7515]) allows the Verifier to use multiple Client Identifiers and corresponding key material to protect the same request. This serves use cases where the Verifier requests Credentials belonging to different trust frameworks and, therefore, needs to authenticate in the context of those trust frameworks. It also allows the Verifier to add different attestations for each Client Identifier.

In this case, the following request parameters, if used, MUST be present only in the protected header of the respective `signature` object in the `signatures` array defined in Section 7.2.1 of [RFC7515]:

- `client_id`
- `verifier_info`
- parameters that are specific to a Client Identifier Prefix, e.g., the `trust_chain` JWS header parameter for the `openid_federation` Client Identifier Prefix

All other request parameters MUST be present in the `payload` element of the JWS object.

Below is a non-normative example of such a request:

```
{
  "payload": "eyAiaXNzIjogImh0dHBzOi8...NzY4Mzc4MzYiIF0gfQ",
  "signatures": [
    {
      "protected": "eyJhbGciOiAiRVMyNT..MiLCJraWQiOiAiMSJ9XX19fQ",
      "signature": "PFwem0Ajp2Sag...T2z784h8TQqgTR9tXcif0jw"
    },
    {
      "protected": "eyJhbGciOiAiRVMyNTY...tpZCI6ICIxIn1dfX19",
      "signature": "irgtXbJGwE2wN4Lc...2TvUodsE0vaC-NXpB9G39cMXZ9A"
    }
  ]
}
```

Every object in the `signatures` structure contains the parameters and the signature specific to a particular Client Identifier. The signature is calculated as specified in section 5.1 of [RFC7515].

The following is a non-normative example of the content of a decoded protected header:

```
{
  "alg": "ES256",
  "x5c": [
    "MIICOjCCAeG...djzH7lA==",
    "MIICLTCCAdS...koAmhWVKe"
  ],
  "client_id": "x509_san_dns:rp.example.com"
}
```

The following is a non-normative example of the payload of a signed OpenID4VP request used with the W3C Digital Credentials API in conjunction with JWS JSON Serialization:

```
{
  "expected_origins": [
    "https://origin1.example.com",
    "https://origin2.example.com"
  ],
  "response_type": "vp_token",
  "response_mode": "dc_api",
  "nonce": "n-0S6_WzA2Mj",
  "dcql_query": {...},
  "client_metadata": {
    "jwks": {
      "keys": [
        {
          "kty": "EC",
          "crv": "P-256",
          "x": "MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
          "y": "4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
          "use": "enc",
          "kid": "1"
        }
      ]
    }
  }
}
```

A.4. Response

Every OpenID4VP Request results in a response being provided through the Digital Credentials API (DC API), or in a canceled flow. If a response is provided, the response is an instance of the `DigitalCredential` interface, as defined in [W3C.Digital_Credentials_API], and the OpenID4VP Response parameters as defined for the Response Type are represented as an object within the `data` property.

Protocol error responses are returned as an object within the `data` property. This object has a single property with the name `error` and a value containing the error response code as defined in Section 8.5. Note that a protocol error generated by the Wallet will still result in a fulfilled promise for the Digital Credentials API request. Privacy considerations specific to returning error responses over the Digital Credentials API can be found in Section 15.9.2.

The following is a non-normative example of a `data` object containing an error:

```
{
    "error": "invalid_request"
}
```

The security properties that are normally provided by the Client Identifier are achieved by binding the response to the Origin it was received from.

The audience for the response (for example, the `aud` value in a Key Binding JWT) MUST be the Origin, prefixed with `origin:`, for example `origin:https://verifier.example.com/`. This is the case even for signed requests. Therefore, when using OpenID4VP over the DC API, the Client Identifier is not used as the audience for the response.

A.5. Security Considerations

The following security considerations from OpenID4VP apply:

- Preventing Replay of Verifiable Presentations as described in Section 14.1, with the difference that the origin is used instead of the Client Identifier to bind the response to the Client.
- End-User Authentication using Credentials as described in Section 14.4.
- Encrypting an Unsigned Response as described in Section 14.5.
- TLS Requirements as described in Section 14.6.
- Always Use the Full Client Identifier as described in Section 14.8 for signed requests.
- Security Checks on the Returned Credentials and Presentations as described in Section 14.9.
- DCQL Value Matching as described in Section 15.4.1.

A.6. Privacy Considerations

The following privacy considerations from OpenID4VP apply:

- Selective Disclosure as described in Section 15.4.
- Privacy implications of mechanisms to establish trust in Issuers as described in Section 15.10.

Appendix B. Credential Format Specific Parameters and Rules

OpenID for Verifiable Presentations is Credential Format agnostic, i.e., it is designed to allow applications to request and receive Presentations in any Credential Format. This section defines a set of Credential Format specific parameters and rules for some of the known Credential Formats. For the Credential Formats that are not mentioned in this specification, other specifications or deployments can define their own set of Credential Format specific parameters.

B.1. W3C Verifiable Credentials

The following sections define the Credential Format specific parameters and rules for W3C Verifiable Credentials compliant to the [VC_DATA] specification and for W3C Verifiable Presentations of such Credentials.

If `require_cryptographic_holder_binding` is set to `true` in the Credential Query, the Wallet MUST return a Verifiable Presentation of a Verifiable Credential. Otherwise, a Verifiable Credential without Holder Binding MUST be returned.

B.1.1. Parameters in the `meta` parameter in Credential Query

The following is a W3C Verifiable Credentials specific parameter in the `meta` parameter in a Credential Query as defined in Section 6.1:

`type_values`: REQUIRED. A non-empty array of string arrays. The value of each element in the `type_values` array is a non-empty array specifying the fully expanded types (IRIs) that the Verifier accepts in a Presentation, after applying the `@context` to the Verifiable Credential. If a `type` value in a Verifiable Credential is not defined in any `@context`, it remains unchanged, i.e., remains a relative IRI after JSON-LD processing. For this reason, JSON-LD processing MAY be skipped in such cases and the relative IRI is considered to be the fully expanded type, as applying the `@context` would not alter the value. Implementations MAY use alternative mechanisms to obtain the fully expanded types, as long as the results are equivalent to those produced by JSON-LD processing. Each of the top-level arrays specifies one alternative to match the fully expanded `type` values of the Verifiable Credential against. Each inner array specifies a set of fully expanded types that MUST be present in the fully expanded types in the `type` property of the Verifiable Credential, regardless of order or the presence of additional types.

The following is a non-normative example of `type_values` within a DCQL query:

```
"type_values":[
  [
      "https://www.w3.org/2018/credentials#VerifiableCredential",
      "https://example.org/examples#AlumniCredential",
      "https://example.org/examples#BachelorDegree"
  ],
  [
      "https://www.w3.org/2018/credentials#VerifiableCredential",
      "https://example.org/examples#UniversityDegreeCredential"
  ],
  [
      "IdentityCredential"
  ]
]
```

The following is a non-normative example of a W3C Verifiable Credential that would match the `type_values` DCQL query above (other claims omitted for readability):

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "type": ["VerifiableCredential", "UniversityDegreeCredential"]
}
```

The following is another non-normative example of a W3C Verifiable Credential that would match the `type_values` DCQL query above (other claims omitted for readability):

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "type": ["VerifiableCredential", "BachelorDegree", "AlumniCredential"]
}
```

The following is another non-normative example of a W3C Verifiable Credential that would match the `type_values` DCQL query above (other claims omitted for readability):

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1"
  ],
  "type": ["VerifiableCredential", "IdentityCredential"]
}
```

B.1.2. Claims Matching

The `claims_path` parameter in the Credential Query as defined in Section 6.1 is used to specify the claims that the Verifier wants to receive in the Presentation. When used in the context of W3C Verifiable Credentials, the `claims_path` parameter always matches on the root of Verifiable Credential (not the Verifiable Presentation). Examples are shown in the following subsections.

B.1.3. Formats and Examples

B.1.3.1. VC signed as a JWT, not using JSON-LD

This section illustrates the presentation of a Credential conformant to [VC_DATA] that is signed using JWS, and does not use JSON-LD.

B.1.3.1.1. Format Identifier and Cipher Suites

The Credential Format Identifier is `jwt_vc_json` to request a W3C Verifiable Credential compliant to the [VC_DATA] specification or a Verifiable Presentation of such a Credential.

Cipher suites should use algorithm names defined in IANA JOSE Algorithms Registry.

B.1.3.1.2. Example Credential

The following is a non-normative example of the payload of a JWT-based W3C Verifiable Credential that will be used throughout this section:

```
{
  "iss": "https://example.gov/issuers/565049",
  "nbf": 1262304000,
  "jti": "http://example.gov/credentials/3732",
  "sub": "did:example:ebfeb1f712ebc6f1c276e12ec21",
  "vc": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "type": [
      "VerifiableCredential",
      "IDCredential"
    ],
    "credentialSubject": {
      "given_name": "Max",
      "family_name": "Mustermann",
      "birthdate": "1998-01-11",
      "address": {
        "street_address": "Sandanger 25",
        "locality": "Musterstadt",
        "postal_code": "123456",
        "country": "DE"
      }
    }
  }
}
```

B.1.3.1.3. Metadata

The `vp_formats_supported` parameter of the Verifier metadata or Wallet metadata MUST have the Credential Format Identifier as a key, and the value MUST be an object consisting of the following name/value pair:

- `alg_values`: OPTIONAL. A non-empty array containing identifiers of cryptographic algorithms supported for a JWT-secured W3C Verifiable Credential or W3C Verifiable Presentation. If present, the `alg` JOSE header (as defined in [RFC7515]) of the presented Verifiable Credential or Verifiable Presentation MUST match one of the array values.

The following is a non-normative example of `client_metadata` request parameter value in a request to present an W3C Verifiable Presentation.

```
{
  "vp_formats_supported": {
    "jwt_vc_json": {
      "alg_values": ["ES256", "ES384"]
    }
  }
}
```

B.1.3.1.4. Presentation Request

The requirements regarding the Credential to be presented are conveyed in the `dcql_query` parameter.

The following is a non-normative example of the contents of this parameter:

```
{
  "credentials": [
    {
      "id": "example_jwt_vc",
      "format": "jwt_vc_json",
      "meta": {
        "type_values": [["IDCredential"]]
      },
      "claims": [
        {"path": ["credentialSubject", "family_name"]},
        {"path": ["credentialSubject", "given_name"]}
      ]
    }
  ]
}
```

B.1.3.1.5. Presentation Response

The following requirements apply to the `nonce` and `aud` claims of the Verifiable Presentation:

- the `nonce` claim MUST be the value of `nonce` from the Authorization Request;
- the `aud` claim MUST be the value of the Client Identifier, except for requests over the DC API where it MUST be the Origin prefixed with `origin:`, as described in Appendix A.4.

The following is a non-normative example of the VP Token provided in the response (shortened for presentation):

```
{
  "example_jwt_vc": ["eY...QMA"]
}
```

The following is a non-normative example of the payload of the Verifiable Presentation in the VP Token in the last example:

```
{
  "iss": "did:example:ebfeb1f712ebc6f1c276e12ec21",
  "jti": "urn:uuid:3978344f-8596-4c3a-a978-8fcaba3903c5",
  "aud": "x509_san_dns:client.example.org",
  "nbf": 1541493724,
  "iat": 1541493724,
  "exp": 1573029723,
  "nonce": "n-0S6_WzA2Mj",
  "vp": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1"
    ],
    "type": [
      "VerifiablePresentation"
    ],
    "verifiableCredential": [
      "eyJhb...ssw5c"
    ]
  }
}
```

B.1.3.2. LDP VCs

This section illustrates presentation of a Credential conformant to [VC_DATA] that is secured using Data Integrity, using JSON-LD.

B.1.3.2.1. Format Identifier and Cipher Suites

The Credential Format Identifier is `ldp_vc` to request a W3C Verifiable Credential compliant to the [VC_DATA] specification or a Verifiable Presentation of such a Credential.

Cipher suites should use Data Integrity compatible securing mechanisms defined in Verifiable Credential Extensions.

B.1.3.2.2. Example Credential

The following is a non-normative example of the payload of a Verifiable Credential that will be used throughout this section:

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1",
    "https://w3id.org/security/data-integrity/v2"
  ],
  "id": "https://example.com/credentials/1872",
  "type": [
    "VerifiableCredential",
    "IDCredential"
  ],
  "issuer": {
    "id": "did:example:issuer"
  },
  "issuanceDate": "2025-03-19T00:00:00Z",
  "credentialSubject": {
    "given_name": "Max",
    "family_name": "Mustermann",
    "birthdate": "1998-01-11",
    "address": {
      "street_address": "Sandanger 25",
      "locality": "Musterstadt",
      "postal_code": "123456",
      "country": "DE"
    }
  },
  "proof": {
    "type": "DataIntegrityProof",
    "cryptosuite": "eddsa-rdfc-2022",
    "created": "2025-03-19T15:30:15Z",
    "proofValue": "z5C5b...EtszK",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "did:example:issuer#keys-1"
  }
}
```

B.1.3.2.3. Metadata

The `vp_formats_supported` parameter of the Verifier metadata or Wallet metadata MUST have the Credential Format Identifier as a key, and the value MUST be an object consisting of the following name/value pairs:

- `proof_type_values`: OPTIONAL. A non-empty array containing identifiers of proof types supported for a Data Integrity secured W3C Verifiable Presentation or W3C Verifiable Credential. If present, the proof `type` parameter (as defined in [VC_DATA]) of the presented Verifiable Credential or Verifiable Presentation MUST match one of the array values.

- `cryptosuite_values`: OPTIONAL. A non-empty array containing identifiers of crypto suites supported with one of the algorithms listed in `proof_type_values` for a Data Integrity secured W3C Verifiable Presentation or W3C Verifiable Credential. Note that `cryptosuite_values` MAY be used if one of the algorithms in `proof_type_values` supports multiple crypto suites. If present, the proof `cryptosuite` parameter (as defined in [VC_DATA_INTEGRITY]) of the presented Verifiable Credential or Verifiable Presentation MUST match one of the array values.

The following is a non-normative example of `client_metadata` request parameter value in a request to present an W3C Verifiable Presentation.

```
{
  "vp_formats_supported": {
    "ldp_vc": {
      "proof_type_values": [
        "DataIntegrityProof",
        "Ed25519Signature2020"
      ],
      "cryptosuite_values": [
        "ecdsa-rdfc-2019",
        "ecdsa-sd-2023",
        "ecdsa-jcs-2019",
        "bbs-2023"
      ]
    }
  }
}
```

B.1.3.2.4. Presentation Request

The requirements regarding the Credential to be presented are conveyed in the `dcql_query` parameter.

The following is a non-normative example of the contents of this parameter:

```
{
  "credentials": [
    {
      "id": "example_ldp_vc",
      "format": "ldp_vc",
      "meta": {
        "type_values": [["IDCredential"]]
      },
      "claims": [
        {"path": ["credentialSubject", "family_name"]},
        {"path": ["credentialSubject", "given_name"]},
        {"path": ["credentialSubject", "birthdate"]},
        {"path": ["credentialSubject", "address", "street_address"]},
        {"path": ["credentialSubject", "address", "locality"]},
        {"path": ["credentialSubject", "address", "postal_code"]},
        {"path": ["credentialSubject", "address", "country"]}
      ]
    }
  ]
}
```

B.1.3.2.5. Presentation Response

The following requirements apply to the `challenge` and `domain` claims within the `proof` object in the Verifiable Presentation:

- the `challenge` claim MUST be the value of `nonce` from the Authorization Request;
- the `domain` claim MUST be the value of the Client Identifier, except for requests over the DC API where it MUST be the Origin prefixed with `origin:`, as described in Appendix A.4.

The following is a non-normative example of the Verifiable Presentation in the `vp_token` parameter:


```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://w3id.org/security/data-integrity/v2"
  ],
  "type": [
    "VerifiablePresentation"
  ],
  "verifiableCredential": [
    {
      "@context": [
        "https://www.w3.org/2018/credentials/v1",
        "https://www.w3.org/2018/credentials/examples/v1",
        "https://w3id.org/security/data-integrity/v2"
      ],
      "id": "https://example.com/credentials/1872",
      "type": [
        "VerifiableCredential",
        "IDCredential"
      ],
      "issuer": {
        "id": "did:example:issuer"
      },
      "issuanceDate": "2025-03-19T00:00:00Z",
      "credentialSubject": {
        "given_name": "Max",
        "family_name": "Mustermann",
        "birthdate": "1998-01-11",
        "address": {
          "street_address": "Sandanger 25",
          "locality": "Musterstadt",
          "postal_code": "123456",
          "country": "DE"
        }
      },
      "proof": {
        "type": "DataIntegrityProof",
        "cryptosuite": "eddsa-rdfc-2022",
        "created": "2025-03-19T15:30:15Z",
        "proofValue": "z5C5b...EtszK",
        "proofPurpose": "assertionMethod",
        "verificationMethod": "did:example:issuer#keys-1"
      }
    }
  ],
  "id": "ebc6f1c2",
  "holder": "did:example:holder",
  "proof": {
    "type": "DataIntegrityProof",
    "cryptosuite": "eddsa-rdfc-2022",
    "created": "2025-04-04T10:12:15Z",
    "challenge": "n-0S6_WzA2Mj",
    "domain": "x509_san_dns:client.example.org",
    "proofValue": "z5s8c...AD3a9d",
    "proofPurpose": "authentication",
    "verificationMethod": "did:example:holder#key-1"
```

```
    }
}
```

B.2. Mobile Documents or mdocs (ISO/IEC 18013 and ISO/IEC 23220 series)

ISO/IEC 18013-5:2021 [ISO.18013-5] defines a mobile driving license (mDL) Credential in the mobile document (mdoc) format. Although ISO/IEC 18013-5:2021 [ISO.18013-5] is specific to mobile driving licenses (mDLs), the Credential format can be utilized with any type of Credential (or mdoc document types). The ISO/IEC 23220 series has extracted components from ISO/IEC 18013-5:2021 [ISO.18013-5] that are common across document types to facilitate the profiling of the specification for other document types. The core data structures are shared between ISO/IEC 18013-5:2021 [ISO.18013-5], ISO/IEC 23220-2 [ISO.23220-2], ISO/IEC 23220-4 [ISO.23220-4] which are encoded in CBOR and secured using COSE_Sign1.

The Credential Format Identifier for Credentials in the mdoc format is `mso_mdoc`.

B.2.1. Transaction Data

It is RECOMMENDED that each transaction data type defines a data element (`NameSpace`, `DataElementIdentifier`, `DataElementValue`) to be used to return the processed transaction data. Additionally, it is RECOMMENDED that it specifies the processing rules, potentially including any hash function to be applied, and the expected resulting structure.

Some document types support some transaction data (Section 8.4) to be protected using mdoc authentication, as part of the `DeviceSigned` data structure [ISO.18013-5]. In those cases, the specifications of these document types include which transaction data types are supported, and the issuer includes the relevant data elements in the `KeyAuthorizations`. If a Wallet receives a request with a `transaction_data` type whose data element is unauthorized, the Wallet MUST reject the request due to an unsupported transaction data type.

B.2.2. Metadata

The `vp_formats_supported` parameter of the Verifier metadata or Wallet metadata MUST have the Credential Format Identifier as a key, and the value MUST be an object consisting of the following name/value pairs:

- `issuerauth_alg_values`: OPTIONAL. A non-empty array containing cryptographic algorithm identifiers. The Credential MUST be considered to fulfill the requirement(s) expressed in this parameter if one of the following is true: 1) The value in the array matches the 'alg' value in the IssuerAuth COSE header. 2) The value in the array is a fully specified algorithm according to [I-D.ietf-jose-fully-specified-algorithms] and the combination of the `alg` value in the `IssuerAuth` COSE header and the curve used by the signing key of the COSE structure matches the combination of the algorithm and curve identified by the fully specified algorithm. As an example, if the `IssuerAuth` structure contains an `alg` header with value `-7` (which stands for ECDSA with SHA-256 in [IANA.COSE]) and is signed by a P-256 key, then it matches an `issuerauth_alg_values` element of `-7` and `-9` (which stands for ECDSA using P-256 curve and SHA-256 in [I-D.ietf-jose-fully-specified-algorithms]).
- `deviceauth_alg_values`: OPTIONAL. A non-empty array containing cryptographic algorithm identifiers. The Credential MUST be considered to fulfill the requirement(s) expressed in this parameter if one of the following is true: 1) The value in the array matches the 'alg' value in the `DeviceSignature` or `DeviceMac` COSE header. 2) The value in the array is a fully-specified algorithm according to [I-D.ietf-jose-fully-specified-algorithms] and the combination of the `alg` value in the `DeviceSignature` COSE header and the curve used by the signing key of the COSE structure matches the combination of the algorithm and curve identified by the fully-specified algorithm. 3) The `alg` of the `DeviceMac` COSE header is `HMAC 256/256` (as described in Section 9.1.3.5 of [ISO.18013-5]) and the curve of the device key (from Table 22 of [ISO.18013-5]) matches a value in the array using the identifiers defined in the following table:

Algorithm Name	Algorithm Value
HMAC 256/256 using ECDH with Curve P-256	-65537
HMAC 256/256 using ECDH with Curve P-384	-65538
HMAC 256/256 using ECDH with Curve P-521	-65539
HMAC 256/256 using ECDH with X25519	-65540
HMAC 256/256 using ECDH with X448	-65541
HMAC 256/256 using ECDH with brainpoolP256r1	-65542
HMAC 256/256 using ECDH with brainpoolP320r1	-65543
HMAC 256/256 using ECDH with brainpoolP384r1	-65544
HMAC 256/256 using ECDH with brainpoolP512r1	-65545

Table 2: Mapping of curves to `alg` identifiers used for the `HMAC 256/256` case

Note: These are specified in OpenID4VP only for private use in this parameter in this specification, and might be superseded by a future registration in IANA.

For clarity, the following is a couple of non-normative examples of the `deviceauth_alg_values` parameter

The example below indicates the verifier supports DeviceMac with HMAC 256/256, where the MAC key is established via ECDH using keys on the P-256 curve as per Section 9.1.3.5 of [ISO.18013-5].

```
{
  "deviceauth_alg_values": [ -65537 ]
}
```

The example below indicates the verifier supports DeviceMac with HMAC 256/256, where the MAC key is established via ECDH using keys on the P-256 curve as per Section 9.1.3.5 of [ISO.18013-5], and DeviceSignature using ECDSA with the P-256 curve.

```
{
  "deviceauth_alg_values": [ -65537, -9 ]
}
```

The following is a non-normative example of `client_metadata` request parameter value in a request to present an ISO/IEC 18013-5 mDOC.

```
{
  "vp_formats_supported": {
    "mso_mdoc": {
      "issuerauth_alg_values": [-9, -50],

      "deviceauth_alg_values": [-9, -50]
    }
  }
}
```

B.2.3. Parameter in the `meta` parameter in Credential Query

The following is an ISO mdoc specific parameter in the `meta` parameter in a Credential Query as defined in Section 6.1.

`doctype_value`: REQUIRED. String that specifies an allowed value for the doctype of the requested Verifiable Credential. It MUST be a valid doctype identifier as defined in [ISO.18013-5].

B.2.4. Parameter in the Claims Query

The following are ISO mdoc specific parameters to be used in a Claims Query as defined in Section 6.3.

`intent_to_retain` OPTIONAL. A boolean that is equivalent to `IntentToRetain` variable defined in Section 8.3.2.1.2.1 of [ISO.18013-5].

B.2.5. Presentation Response

An example DCQL query using the mdoc format is shown in Appendix D. The following is a non-normative example for a VP Token in the response:

```
{
  "my_credential": ["<base64url-encoded DeviceResponse>"]
}
```

The VP Token contains the base64url-encoded `DeviceResponse` CBOR structure as defined in ISO/IEC 18013-5 [ISO.18013-5] or ISO/IEC 23220-4 [ISO.23220-4]. Essentially, the `DeviceResponse` CBOR structure contains a signature or MAC over the `SessionTranscript` CBOR structure including the OpenID4VP-specific `Handover` CBOR structure.

B.2.6. `Handover` and `SessionTranscript` Definitions

B.2.6.1. Invocation via Redirects

If the presentation request is invoked using redirects, the `SessionTranscript` CBOR structure as defined in Section 9.1.5.1 in [ISO.18013-5] MUST be used with the following changes:

- `DeviceEngagementBytes` MUST be `null`.
- `EReaderKeyBytes` MUST be `null`.
- `Handover` MUST be the `OpenID4VPHandover` CBOR structure as defined below.

```
OpenID4VPHandover = [
  "OpenID4VPHandover", ; A fixed identifier for this handover type
  OpenID4VPHandoverInfoHash ; A cryptographic hash of OpenID4VPHandoverInfo
]

; Contains the sha-256 hash of OpenID4VPHandoverInfoBytes
OpenID4VPHandoverInfoHash = bstr

; Contains the bytes of OpenID4VPHandoverInfo encoded as CBOR
OpenID4VPHandoverInfoBytes = bstr .cbor OpenID4VPHandoverInfo

OpenID4VPHandoverInfo = [
  clientId,
  nonce,
  jwkThumbprint,
  responseUri
] ; Array containing handover parameters

clientId = tstr

nonce = tstr

jwkThumbprint = bstr

responseUri = tstr
```

The `OpenID4VPHandover` structure has the following elements:

- The first element MUST be the string `OpenID4VPHandover`. This serves as a unique identifier for the handover structure to prevent misinterpretation or confusion.
- The second element MUST be a Byte String which contains the sha-256 hash of the bytes of `OpenID4VPHandoverInfo` when encoded as CBOR.
- The `OpenID4VPHandoverInfo` has the following elements:
  - The first element MUST be the `client_id` request parameter. If applicable, this includes the Client Identifier Prefix.
  - The second element MUST be the value of the nonce request parameter.
  - If the response is encrypted, e.g., using `direct_post.jwt`, the third element MUST be the JWK SHA-256 Thumbprint as defined in [RFC7638], encoded as a Byte String, of the Verifier's public key used to encrypt the response. Otherwise, the third element MUST be `null`. See Appendix B.2.6.2 for an explanation of why this is important.
  - The fourth element MUST be either the `redirect_uri` or `response_uri` request parameter, depending on which is present, as determined by the Response Mode.

Unless otherwise stated, the values of `client_id`, nonce, `redirect_uri`, and `response_uri` request parameters referenced above MUST be obtained from the Authorization Request query parameters if the request is unsigned, or from the signed Request Object if the request is signed.

The following is a non-normative example of the input JWK for calculating the JWK Thumbprint in the context of `OpenID4VPHandoverInfo`:

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "DxiH5Q4Yx3UrukE2lWCErq8N8bqC9CHLLrAwLz5BmE0",
  "y": "XtLM4-3h5o3HUH0MHVJV0kyq0iBlrBwlh8qEDMZ4-Pc",
  "use": "enc",
  "alg": "ECDH-ES",
  "kid": "1"
}
```

The following is a non-normative example of the `OpenID4VPHandoverInfo` structure:

```
Hex:

847818783530395f73616e5f646e733a6578616d706c652e636f6d782b6578633767
426b786a7831726463397564527276654b7653734a4971383061766c58654c486847
7771744158204283ec927ae0f208daaa2d026a814f2b22dca52cf85ffa8f3f8626c6
bd669047781c68747470733a2f2f6578616d706c652e636f6d2f726573706f6e7365

CBOR diagnostic:

84                                  # array(4)
  78 18                             #   string(24)
    783530395f73616e5f646e733a6578  #     "x509_san_dns:ex"
    616d706c652e636f6d              #     "ample.com"
  78 2b                             #   string(43)
    6578633767426b786a783172646339  #     "exc7gBkxjx1rdc9"
    7564527276654b7653734a49713830  #     "udRrveKvSsJIq80"
    61766c58654c48684777717441      #     "avlXeLHhGwqtA"
  58 20                             #   bytes(32)
    4283ec927ae0f208daaa2d026a814f  #     "B\x83ì\x92zàò\x08Úª-\x02j\x81O"
    2b22dca52cf85ffa8f3f8626c6bd66  #     "+"Ü¥,ø_ú\x8f?\x86&Æ½f"
    9047                            #     "\x90G"
  78 1c                             #   string(28)
    68747470733a2f2f6578616d706c65  #     "https://example"
    2e636f6d2f726573706f6e7365      #     ".com/response"
```

The following is a non-normative example of the `OpenID4VPHandover` structure:

```
Hex:

82714f70656e494434565048616e646f7665725820048bc053c00442af9b8eed494c
efdd9d95240d254b046b11b68013722aad38ac

CBOR diagnostic:

82                                  # array(2)
  71                                #   string(17)
    4f70656e494434565048616e646f76  #     "OpenID4VPHandov"
    6572                            #     "er"
  58 20                             #   bytes(32)
    048bc053c00442af9b8eed494cefdd  #     "\x04\x8bÀSÀ\x04B¯\x9b\x8eíILïÝ"
    9d95240d254b046b11b68013722aad  #     "\x9d\x95$\x0d%K\x04k\x11¶\x80\x13r*"
    38ac                            #     "8¬"
```

The following is a non-normative example of the `SessionTranscript` structure:

```
Hex:

83f6f682714f70656e494434565048616e646f7665725820048bc053c00442af9b8e
ed494cefdd9d95240d254b046b11b68013722aad38ac

CBOR diagnostic:

83                                  # array(3)
  f6                                #   null
  f6                                #   null
  82                                #   array(2)
    71                              #     string(17)
      4f70656e494434565048616e646f  #       "OpenID4VPHando"
      766572                        #       "ver"
    58 20                           #     bytes(32)
      048bc053c00442af9b8eed494cef  #       "\x04\x8bÀSÀ\x04B¯\x9b\x8eíILï"
      dd9d95240d254b046b11b6801372  #       "Ý\x9d\x95$\x0d%K\x04k\x11¶\x80\x13r"
      2aad38ac                      #       "*8¬"
```

B.2.6.2. Invocation via the Digital Credentials API

If the presentation request is invoked using the Digital Credentials API, the `SessionTranscript` CBOR structure as defined in Section 9.1.5.1 in [ISO.18013-5] MUST be used with the following changes:

- `DeviceEngagementBytes` MUST be `null`.
- `EReaderKeyBytes` MUST be `null`.
- `Handover` MUST be the `OpenID4VPDCAPIHandover` CBOR structure as defined below.

Note: The following section contains a definition in Concise Data Definition Language (CDDL), a language used to define data structures - see [RFC8610] for more details. `bstr` refers to Byte String, defined as major type 2 in CBOR and `tstr` refers to Text String, defined as major type 3 in CBOR (encoded in utf-8) as defined in section 3.1 of [RFC8949].

```
OpenID4VPDCAPIHandover = [
  "OpenID4VPDCAPIHandover", ; A fixed identifier for this handover type
  OpenID4VPDCAPIHandoverInfoHash ; A cryptographic hash of OpenID4VPDCAPIHandoverInfo
]

; Contains the sha-256 hash of OpenID4VPDCAPIHandoverInfoBytes
OpenID4VPDCAPIHandoverInfoHash = bstr

; Contains the bytes of OpenID4VPDCAPIHandoverInfo encoded as CBOR
OpenID4VPDCAPIHandoverInfoBytes = bstr .cbor OpenID4VPDCAPIHandoverInfo

OpenID4VPDCAPIHandoverInfo = [
  origin,
  nonce,
  jwkThumbprint
] ; Array containing handover parameters

origin = tstr

nonce = tstr

jwkThumbprint = bstr
```

The `OpenID4VPDCAPIHandover` structure has the following elements:

- The first element MUST be the string `OpenID4VPDCAPIHandover`. This serves as a unique identifier for the handover structure to prevent misinterpretation or confusion.
- The second element MUST be a Byte String which contains the sha-256 hash of the bytes of `OpenID4VPDCAPIHandoverInfo` when encoded as CBOR.
- The `OpenID4VPDCAPIHandoverInfo` has the following elements:
  - The first element MUST be the string representing the Origin of the request as described in Appendix A.2. It MUST NOT be prefixed with `origin:`.
  - The second element MUST be the value of the `nonce` request parameter.
  - For the Response Mode `dc_api.jwt`, the third element MUST be the JWK SHA-256 Thumbprint as defined in [RFC7638], encoded as a Byte String, of the Verifier's public key used to encrypt the response. If the Response Mode is `dc_api`, the third element MUST be `null`. For unsigned requests, including the JWK Thumbprint in the `SessionTranscript` allows the Verifier to detect whether the response was re-encrypted by a third party, potentially leading to the leakage of sensitive information. While this does not prevent such an attack, it makes it detectable and helps preserve the confidentiality of the response.

The following is a non-normative example of the input JWK for calculating the JWK Thumbprint in the context of `OpenID4VPDCAPIHandoverInfo`:

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "DxiH5Q4Yx3UrukE2lWCErq8N8bqC9CHLLrAwLz5BmE0",
  "y": "XtLM4-3h5o3HUH0MHVJV0kyq0iBlrBwlh8qEDMZ4-Pc",
  "use": "enc",
  "alg": "ECDH-ES",
  "kid": "1"
}
```

The following is a non-normative example of the `OpenID4VPDCAPIHandoverInfo` structure:

```
Hex:

837368747470733a2f2f6578616d706c652e636f6d782b6578633767426b786a7831
726463397564527276654b7653734a4971383061766c58654c486847777174415820
4283ec927ae0f208daaa2d026a814f2b22dca52cf85ffa8f3f8626c6bd669047

CBOR diagnostic:

83                                # array(3)
  73                              #   string(19)
    68747470733a2f2f6578616d706c65 #     "https://example"
    2e636f6d                      #     ".com"
  78 2b                           #   string(43)
    6578633767426b786a783172646339 #     "exc7gBkxjx1rdc9"
    7564527276654b7653734a49713830 #     "udRrveKvSsJIq80"
    61766c58654c48684777717441     #     "avlXeLHhGwqtA"
  58 20                           #   bytes(32)
    4283ec927ae0f208daaa2d026a814f #     "B\x83i\x92zàò\x08Úª-\x02j\x81O"
    2b22dca52cf85ffa8f3f8626c6bd66 #     "+"Ü¥,ø_ú\x8f?\x86&Æ½f"
    9047                          #     "\x90G"
```

The following is a non-normative example of the `OpenID4VPDCAPIHandover` structure:

```
Hex:

82764f70656e4944345650444341504948616e646f7665725820fbece366f4212f97
62c74cfdbf83b8c69e371d5d68cea09cb4c48ca6daab761a

CBOR diagnostic:

82                                  # array(2)
  76                                #   string(22)
    4f70656e494434565044434150494 8 #     "OpenID4VPDCAPIH"
    616e646f766572                  #     "andover"
  58 20                             #   bytes(32)
    fbece366f4212f9762c74cfdbf83b8 #     "ûìãfô!/\x97bÇLý¿\x83¸"
    c69e371d5d68cea09cb4c48ca6daab #     "Æ\x9e7\x1d]hÎ\xa0\x9c´Ä\x8c¦Ú«"
    761a                            #     "v\x1a"
```

The following is a non-normative example of the `SessionTranscript` structure:

```
Hex:

83f6f682764f70656e4944345650444341504948616e646f7665725820fbece366f4
212f9762c74cfdbf83b8c69e371d5d68cea09cb4c48ca6daab761a

CBOR diagnostic:

83                                  # array(3)
  f6                                #   null
  f6                                #   null
  82                                #   array(2)
    76                              #     string(22)
      4f70656e49443456504443415049  #       "OpenID4VPDCAPI"
      48616e646f766572              #       "Handover"
    58 20                           #     bytes(32)
      fbece366f4212f9762c74cfdbf83  #       "ûìãfô!/\x97bÇLý¿\x83"
      b8c69e371d5d68cea09cb4c48ca6  #       "¸Æ\x9e7\x1d]hÎ\xa0\x9c´Ä\x8c¦"
      daab761a                      #       "Ú«v\x1a"
```

B.3. IETF SD-JWT VC

This section defines how Credentials complying with [I-D.ietf-oauth-sd-jwt-vc] can be presented to the Verifier using this specification.

If `require_cryptographic_holder_binding` is set to `true` in the Credential Query, the Wallet MUST return an SD-JWT [I-D.ietf-oauth-selective-disclosure-jwt] with a Key Binding JWT (SD-JWT+KB) as the Verifiable Presentation. SD-JWTs that do not support Holder Binding (i.e., do not have a `cnf` Claim) cannot be returned in this case. If `require_cryptographic_holder_binding` is set to `false`, an SD-JWT without the Key Binding JWT MAY be returned.

B.3.1. Format Identifier

The Credential Format Identifier is `dc+sd-jwt`.

B.3.2. Example Credential

The following is a non-normative example of the unsecured payload of an IETF SD-JWT VC that will be used throughout this section:

```
{
  "vct": "https://credentials.example.com/identity_credential",
  "given_name": "John",
  "family_name": "Doe",
  "birthdate": "1940-01-01"
}
```

The following is a non-normative example of an IETF SD-JWT VC using the unsecured payload above, containing claims that are selectively disclosable.

```
{
  "_sd": [
    "3oUCnaKt7wqDKuyh-LgQozzfhgb8gO5Ni-RCWsWW2vA",
    "8z8z9X9jUtb99gjejCwFAGz4aqlHf-sCqQ6eM_qmpUQ",
    "Cxq4872UXXngGULT_kl8fdwVFkyK6AJfPZLy7L5_0kI",
    "TGf4oLbgwd5JQaHyKVQZU9UdGE0w5rtDsrZzfUaomLo",
    "jsu9yVulwQQlhFlM_3JlzMaSFzglhQG0DpfayQwLUK4",
    "sFcViHN-JG3eTUyBmU4fkwusy5I1SLBhe1jNvKxP5xM",
    "tiTngp9_jhC389UP8_k67MXqoSfiHq3iK6o9un4we_Y",
    "xsKkGJXD1-e3I9zj0YyKNv-lU5YqhsEAF9NhOr8xga4"
  ],
  "iss": "https://example.com/issuer",
  "iat": 1683000000,
  "exp": 1883000000,
  "vct": "https://credentials.example.com/identity_credential",
  "_sd_alg": "sha-256",
  "cnf": {
    "jwk": {
      "kty": "EC",
      "crv": "P-256",
      "x": "TCAER19Zvu3OHF4j4W4vfSVoHIP1ILilDls7vCeGemc",
      "y": "ZxjiWWbZMQGHVWKVQ4hbSIirsVfuecCE6t4jT9F2HZQ"
    }
  }
}
```

The following are disclosures belonging to the claims from the example above.

Claim `given_name`:

- SHA-256 Hash: `jsu9yVulwQQlhFlM_3JlzMaSFzglhQG0DpfayQwLUK4`
- Disclosure: `WyIyR0xDNDJzS1F2ZUNmR2ZyeU5STjl3IiwgImdpdmVuX25hbWUiLCAiSm9obiJd`
- Contents: `["2GLC42sKQveCfGfryNRN9w", "given_name", "John"]`

Claim `family_name`:

- SHA-256 Hash: `TGf4oLbgwd5JQaHyKVQZU9UdGE0w5rtDsrZzfUaomLo`
- Disclosure: `WyJlbHVWNU9nM2dTTklJOEVZbnN4QV9BIiwgImZhbWlseV9uYW1lIiwgIkRvZSJd`
- Contents: `["eluV5Og3gSNII8EYnsxA_A", "family_name", "Doe"]`

Claim `birthdate`:

- SHA-256 Hash: `tiTngp9_jhC389UP8_k67MXqoSfiHq3iK6o9un4we_Y`
- Disclosure:
  `WyI2SWo3dE0tYTVpVlBHYm9TNXRtdlZBIiwgImJpcnRoZGF0ZSIsICIxOTQw
  LTAxLTAxIl0`
- Contents: `["6Ij7tM-a5iVPGboS5tmvVA", "birthdate", "1940-01-01"]`

B.3.3. Transaction Data

It is RECOMMENDED that each transaction data type defines a top-level claim parameter to be used in the Key Binding JWT to return the processed transaction data. Additionally, it is RECOMMENDED that it specifies the processing rules, potentially including any hash function to be applied, and the expected resulting structure.

The transaction data mechanism requires the use of an SD-JWT VC with Cryptographic Holder Binding. Wallets MUST reject requests with transaction data types that have the `require_cryptographic_holder_binding` parameter set to `false`.

B.3.3.1. A Profile of Transaction Data in SD-JWT VC

The following is one profile that can be included in a transaction data type specification:

- The `transaction_data` request parameter includes the following parameter, in addition to `type` and `credential_ids` from Section 5.1:
  - `transaction_data_hashes_alg`: OPTIONAL. Non-empty array of strings each representing a hash algorithm identifier, one of which MUST be used to calculate hashes in `transaction_data_hashes` response parameter. The value of the identifier MUST be a hash algorithm value from the "Hash Name String" column in the IANA "Named Information Hash Algorithm" registry [IANA.Hash.Algorithms] or a value defined in another specification and/or profile of this specification. If this parameter is not present, a default value of `sha-256` MUST be used. To promote interoperability, implementations MUST support the sha-256 hash algorithm.

- The Key Binding JWT in the response includes the following top-level parameters:
  - `transaction_data_hashes`: A non-empty array of strings where each element is a base64url-encoded hash. Each of these hashes is calculated using a hash function over the string received in the `transaction_data` request parameter (base64url decoding is not performed before hashing). Each hash value ensures the integrity of, and maps to, the respective transaction data object. If `transaction_data_hashes_alg` was specified in the request, the hash function MUST be one of its values. If `transaction_data_hashes_alg` was not specified in the request, the hash function MUST be `sha-256`.
  - `transaction_data_hashes_alg`: REQUIRED when this parameter was present in the `transaction_data` request parameter. String representing the hash algorithm identifier used to calculate hashes in `transaction_data_hashes` response parameter.

B.3.4. Metadata

The `vp_formats_supported` parameter of the Verifier metadata or Wallet metadata MUST have the Credential Format Identifier as a key, and the value MUST be an object consisting of the following name/value pairs:

- `sd-jwt_alg_values`: OPTIONAL. A non-empty array containing fully-specified identifiers of cryptographic algorithms (as defined in [I-D.ietf-jose-fully-specified-algorithms]) supported for an Issuer-signed JWT of an SD-JWT.
- `kb-jwt_alg_values`: OPTIONAL. A non-empty array containing fully-specified identifiers of cryptographic algorithms (as defined in [I-D.ietf-jose-fully-specified-algorithms]) supported for a Key Binding JWT (KB-JWT).

The following is a non-normative example of `client_metadata` request parameter value in a request to present an IETF SD-JWT VC.

```
{
  "vp_formats_supported": {
    "dc+sd-jwt": {
      "sd-jwt_alg_values": ["ES256", "ES384"],
      "kb-jwt_alg_values": ["ES256", "ES384"]
    }
  }
}
```

B.3.5. Parameter in the `meta` parameter in Credential Query

The following is an SD-JWT VC specific parameter in the `meta` parameter in a Credential Query as defined in Section 6.1.

`vct_values`: REQUIRED. A non-empty array of strings that specifies allowed values for the type of the requested Verifiable Credential. All elements in the array MUST be valid type identifiers as defined in [I-D.ietf-oauth-sd-jwt-vc]. The Wallet MAY return Credentials that inherit from any of the specified types, following the inheritance logic defined in [I-D.ietf-oauth-sd-jwt-vc].

B.3.6. Presentation Response

A non-normative example DCQL query using the SD-JWT VC format is shown in Section 7.4. The respective response is shown in Section 8.1.1.

Additional examples are shown in Appendix D.

The following requirements apply to the `nonce` and `aud` claims in the Key Binding JWT:

- the `nonce` claim MUST be the value of `nonce` from the Authorization Request;
- the `aud` claim MUST be the value of the Client Identifier, except for requests over the DC API where it MUST be the Origin prefixed with `origin:`, as described in Appendix A.4.

The following is a non-normative example of the unsecured payload of the Key Binding JWT of a Verifiable Presentation.

```
{
  "nonce": "n-0S6_WzA2Mj",
  "aud": "x509_san_dns:client.example.org",
  "iat": 1709838604,
  "sd_hash": "Dy-RYwZfaaoC3inJbLslgPvMp09bH-clYP_3qbRqtW4",
  "transaction_data_hashes": [ "fOBUSQvo46yQO-wRwXBcGqvnbKIueISEL961_Sjd4do" ]
}
```

B.3.7. SD-JWT VCLD

SD-JWT VCLD (SD-JWT Verifiable Credentials with JSON-LD) extends the IETF SD-JWT VC [I-D.ietf-oauth-sd-jwt-vc] Credential format and allows to incorporate existing data models that use Linked Data, e.g., W3C VCDM [VC_DATA], while enabling a consistent and uncomplicated approach to selective disclosure.

Information contained in SD-JWT VCLD Credentials can be processed using a JSON-LD [JSON-LD] processor after the SD-JWT VC processing.

When IETF SD-JWT VC is mentioned in this specification, SD-JWT VCLD defined in this section MAY be used.

B.3.7.1. Format

SD-JWT VCLD Credentials are valid SD-JWT VCs and all requirements from [I-D.ietf-oauth-sd-jwt-vc] apply. Additionally, the requirements listed in this section apply.

For compatibility with JWT processors, the following registered Claims from [RFC7519] and [I-D.ietf-oauth-sd-jwt-vc] MUST be used instead of any respective counterpart properties from W3C VCDM or elsewhere:

- `vct` to represent the type of the Credential.
- `exp` and `nbf` to represent the validity period of SD-JWT VCLD (i.e., cryptographic signature).
- `iss` to represent the Credential Issuer.
- `status` to represent the information to obtain the status of the Credential.

IETF SD-JWT VC is extended with the following claim:

- `ld`: OPTIONAL. Contains a JSON-LD [JSON-LD] object in compact form, e.g., [VC_DATA].

B.3.7.2. Processing

The following outlines a suggested non-normative set of processing steps for SD-JWT VCLD:

B.3.7.2.1. Step 1: SD-JWT VC Processing

- A receiver (holder or verifier) of an SD-JWT VCLD applies the processing rules outlined in Section 4 of [I-D.ietf-oauth-sd-jwt-vc], including verifying signatures, validity periods, status information, etc.
- If the `vct` value is associated with any SD-JWT VC Type Metadata, schema validation of the entire SD-JWT VCLD is performed, including the nested `ld` claim.
- Additionally, trust framework rules are applied, such as ensuring the Credential Issuer is authorized to issue SD-JWT VCLDs for the specified `vct` value.

B.3.7.2.2. Step 2: Business Logic Processing

- Once the SD-JWT VC is verified and trusted by the SD-JWT VC processor, and if the `ld` claim is present, the receiver extracts the JSON-LD object from the `ld` claim and uses this for the business logic object. If the `ld` claim is not present, the entire SD-JWT VC is considered to represent the business logic object.
- The business logic object is then passed on for further use case-specific processing and validation. The business logic assumes that all security-critical functions (e.g., signature verification, trusted issuer) have already been performed during the previous step. Additional schema validation is applied if provided in the `ld` claim, e.g., to support SHACL schemas. Note that while a `vct` claim is required, SD-JWT VC type metadata resolution and related schema validation is optional in certain cases.

B.3.7.3. Examples

The following is a non-normative example of an unsecured payload of an SD-JWT VCLD (i.e., before applying the modifications to enable selective disclosure and before adding validity claims).

```
{
  "vct": "https://credentials.example.com/example_credential",
  "ld": {
    "@context": [
      "https://www.w3.org/ns/credentials/v2",
      "https://w3id.org/citizenship/v3"
    ],
    "credentialSubject": {
      "givenName": "John",
      "familyName": "Doe",
      "birthDate": "1978-07-17"
    }
  }
}
```

The following payload would be used in the SD-JWT after encoding the payload above and enabling selective disclosure on the End-User specific claims within `credentialSubject`:

```
{
  "iss": "https://issuer.example.com",
  "iat": 1683000000,
  "exp": 1883000000,
  "vct": "https://credentials.example.com/example_credential",
  "ld": {
    "@context": [
      "https://www.w3.org/ns/credentials/v2",
      "https://w3id.org/citizenship/v3"
    ],
    "credentialSubject": {
      "_sd": [
        "6BJdQrO24ejTTMsFI-wGiJJmGrbseWc5IwCCp4NAJ0k",
        "NTDVsbVAwS9AnUVq-_YG_wv0yGD0bv2JstX-AmvN65I",
        "ts0pyPntLjD0_NcgNOI3hd_2WjbZw21p2LfqhOC0b-U"
      ]
    }
  },
  "_sd_alg": "sha-256",
  "cnf": {
    "jwk": {
      "kty": "EC",
      "crv": "P-256",
      "x": "TCAER19Zvu3OHF4j4W4vfSVoHIP1ILilDls7vCeGemc",
      "y": "ZxjiWWbZMQGHVWKVQ4hbSIirsVfuecCE6t4jT9F2HZQ"
    }
  }
}
```

Note: The decision which claims to make selectively disclosable is up to the Issuer of the Credential. Considerations can be found in Section 6 and Section 9.7 of [I-D.ietf-oauth-selective-disclosure-jwt].

Appendix C. Combining this specification with SIOPv2

This section shows how SIOP and OpenID for Verifiable Presentations can be combined to present Credentials and pseudonymously authenticate an End-User using subject controlled key material.

C.1. Request

The following is a non-normative example of a request that combines this specification and [SIOPv2].

```
GET /authorize?
  response_type=vp_token%20id_token
  &scope=openid
  &id_token_type=subject_signed
  &client_id=x509_san_dns%3Aclient.example.org
  &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
  &dcql_query=...
  &nonce=n-0S6_WzA2Mj HTTP/1.1
Host: wallet.example.com
```

The differences to the example requests in the previous sections are:

- `response_type` is set to `vp_token id_token`. This means the Wallet returns the `vp_token` parameter in the same response as the `id_token` parameter as described in Section 8.
- The request includes the `scope` parameter with value `openid` making this an OpenID Connect request. Additionally, the request also contains the parameter `id_token_type` with value `subject_signed` requesting a Self-Issuer ID Token, i.e., the request is a SIOP request.

C.2. Response

The following is a non-normative example of a response sent upon receiving a request provided in Appendix C.1:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
  id_token=
  &vp_token=...
```

In addition to the `vp_token`, it also contains an `id_token`.

The following is a non-normative example of the payload of a Self-Issued ID Token [SIOPv2] contained in the above response:

```
{
  "iss": "did:example:NzbLsXh8uDCcd6MNwXF4W7noWXFZAfHkxZsRGC9Xs",
  "sub": "did:example:NzbLsXh8uDCcd6MNwXF4W7noWXFZAfHkxZsRGC9Xs",
  "aud": "x509_san_dns:client.example.org",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970
}
```

Note: The `nonce` and `aud` are set to the `nonce` of the request and the Client Identifier of the Verifier, respectively, in the same way as for the Verifier, Verifiable Presentations to prevent replay.

Appendix D. Examples for DCQL Queries

The following is a non-normative example of a DCQL query that requests a Verifiable Credential in the format `mso_mdoc` with the claims `vehicle_holder` and `first_name`:

```
{
  "credentials": [
    {
      "id": "my_credential",
      "format": "mso_mdoc",
      "meta": {
        "doctype_value": "org.iso.7367.1.mVRC"
      },
      "claims": [
        {"path": ["org.iso.7367.1", "vehicle_holder"]},
        {"path": ["org.iso.18013.5.1", "first_name"]}
      ]
    }
  ]
}
```

The following is a non-normative example of a DCQL query that requests multiple Verifiable Credentials; all of them must be returned:

```
{
  "credentials": [
    {
      "id": "pid",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": ["https://credentials.example.com/identity_credential"]
      },
      "claims": [
        {"path": ["given_name"]},
        {"path": ["family_name"]},
        {"path": ["address", "street_address"]}
      ]
    },
    {
      "id": "mdl",
      "format": "mso_mdoc",
      "meta": {
        "doctype_value": "org.iso.7367.1.mVRC"
      },
      "claims": [
        {"path": ["org.iso.7367.1", "vehicle_holder"]},
        {"path": ["org.iso.18013.5.1", "first_name"]}
      ]
    }
]
}
```

The following shows a complex query where the Wallet is requested to deliver the `pid` Credential, or the `other_pid` Credential, or both `pid_reduced_cred_1` and `pid_reduced_cred_2`. Additionally, the `nice_to_have` Credential may optionally be delivered.

```
{
  "credentials": [
    {
      "id": "pid",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": ["https://credentials.example.com/identity_credential"]
      },
```

```
      "claims": [
        {"path": ["given_name"]},
        {"path": ["family_name"]},
        {"path": ["address", "street_address"]}
      ]
    },
    {
      "id": "other_pid",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": ["https://othercredentials.example/pid"]
      },
      "claims": [
        {"path": ["given_name"]},
        {"path": ["family_name"]},
        {"path": ["address", "street_address"]}
      ]
    },
    {
      "id": "pid_reduced_cred_1",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": ["https://credentials.example.com/reduced_identity_credential"]
      },
      "claims": [
        {"path": ["family_name"]},
        {"path": ["given_name"]}
      ]
    },
    {
      "id": "pid_reduced_cred_2",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": ["https://cred.example/residence_credential"]
      },
      "claims": [
        {"path": ["postal_code"]},
        {"path": ["locality"]},
        {"path": ["region"]}
      ]
    },
    {
      "id": "nice_to_have",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": ["https://company.example/company_rewards"]
      },
      "claims": [
        {"path": ["rewards_number"]}
      ]
    }
  ],
  "credential_sets": [
    {
      "options": [
        [ "pid" ],
        [ "other_pid" ],
        [ "pid_reduced_cred_1", "pid_reduced_cred_2" ]
      ]
    },
    {
      "required": false,
      "options": [
        [ "nice_to_have" ]
      ]
```

```
    }
  ]
}
```

The following shows a query where an ID and an address are requested; either can come from an mDL or a photoid Credential.

```
{
  "credentials": [
    {
      "id": "mdl-id",
      "format": "mso_mdoc",
      "meta": {
        "doctype_value": "org.iso.18013.5.1.mDL"
      },
      "claims": [
        {
          "id": "given_name",
          "path": ["org.iso.18013.5.1", "given_name"]
        },
        {
          "id": "family_name",
          "path": ["org.iso.18013.5.1", "family_name"]
        },
        {
          "id": "portrait",
          "path": ["org.iso.18013.5.1", "portrait"]
        }
      ]
    },
    {
      "id": "mdl-address",
      "format": "mso_mdoc",
      "meta": {
        "doctype_value": "org.iso.18013.5.1.mDL"
      },
      "claims": [
        {
          "id": "resident_address",
          "path": ["org.iso.18013.5.1", "resident_address"]
        },
        {
          "id": "resident_country",
          "path": ["org.iso.18013.5.1", "resident_country"]
        }
      ]
    },
    {
      "id": "photo_card-id",
      "format": "mso_mdoc",
      "meta": {
        "doctype_value": "org.iso.23220.photoid.1"
      },
      "claims": [
        {
          "id": "given_name",
          "path": ["org.iso.18013.5.1", "given_name"]
        },
        {
          "id": "family_name",
          "path": ["org.iso.18013.5.1", "family_name"]
        },
```

```
        {
          "id": "portrait",
          "path": ["org.iso.18013.5.1", "portrait"]
        }
      ]
    },
    {
      "id": "photo_card-address",
      "format": "mso_mdoc",
      "meta": {
        "doctype_value": "org.iso.23220.photoid.1"
      },
      "claims": [
        {
          "id": "resident_address",
          "path": ["org.iso.18013.5.1", "resident_address"]
        },
        {
          "id": "resident_country",
          "path": ["org.iso.18013.5.1", "resident_country"]
        }
      ]
    }
  ],
  "credential_sets": [
    {
      "options": [
        [ "mdl-id" ],
        [ "photo_card-id" ]
      ]
    },
    {
      "required": false,
      "options": [
        [ "mdl-address" ],
        [ "photo_card-address" ]
      ]
    }
  ]
}
```

The following is a non-normative example of a DCQL query that requests

- the mandatory claims `last_name` and `date_of_birth`, and
- either the claim `postal_code`, or, if that is not available, both of the claims `locality` and `region`.

```
{
  "credentials": [
    {
      "id": "pid",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": [ "https://credentials.example.com/identity_credential" ]
      },
      "claims": [
        {"id": "a", "path": ["last_name"]},
        {"id": "b", "path": ["postal_code"]},
        {"id": "c", "path": ["locality"]},
        {"id": "d", "path": ["region"]},
        {"id": "e", "path": ["date_of_birth"]}
      ],
      "claim_sets": [
        ["a", "c", "d", "e"],
        ["a", "b", "e"]
      ]
    }
  ]
}
```

The following example shows a query that uses the `values` constraints to request a Credential with specific values for the `last_name` and `postal_code` claims:

```
{
  "credentials": [
    {
      "id": "my_credential",
      "format": "dc+sd-jwt",
      "meta": {
        "vct_values": [ "https://credentials.example.com/identity_credential" ]
      },
      "claims": [
          {
            "path": ["last_name"],
            "values": ["Doe"]
          },
          {"path": ["first_name"]},
          {"path": ["address", "street_address"]},
          {
            "path": ["postal_code"],
            "values": ["90210", "90211"]
          }
      ]
    }
  ]
}
```

Appendix E. IANA Considerations

E.1. OAuth Authorization Endpoint Response Types Registry

This specification registers the following `response_type` values in the IANA "OAuth Authorization Endpoint Response Types" registry [IANA.OAuth.Parameters] established by [RFC6749].

E.1.1. vp_token

- Response Type Name: `vp_token`
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Specification Document(s): Section 8 of this specification

E.1.2. vp_token id_token

- Response Type Name: `vp_token id_token`
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Specification Document(s): Section 8 of this specification

E.2. OAuth Parameters Registry

This specification registers the following OAuth parameters in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749].

E.2.1. dcql_query

- Name: `dcql_query`
- Parameter Usage Location: authorization request
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 5.1 of this specification

E.2.2. client_metadata

- Name: `client_metadata`
- Parameter Usage Location: authorization request
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 5.1 of this specification

E.2.3. request_uri_method

- Name: `request_uri_method`
- Parameter Usage Location: authorization request
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 5.1 of this specification

E.2.4. transaction_data

- Name: `transaction_data`
- Parameter Usage Location: authorization request
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 5.1 of this specification

E.2.5. wallet_nonce

- Name: `wallet_nonce`

- Parameter Usage Location: authorization request, token response
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 5.10 of this specification

E.2.6. response_uri

- Name: `response_uri`
- Parameter Usage Location: authorization request
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 8.2 of this specification

E.2.7. vp_token

- Name: `vp_token`
- Parameter Usage Location: authorization response, token response
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 8.1 of this specification

E.2.8. verifier_info

- Name: `verifier_info`
- Parameter Usage Location: authorization request
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 5.1 of this specification

E.2.9. expected_origins

- Name: `expected_origins`
- Parameter Usage Location: authorization request
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Appendix A.2 of this specification

E.3. OAuth Extensions Error Registry

This specification registers the following errors in the IANA "OAuth Extensions Error" registry [IANA.OAuth.Parameters] established by [RFC6749].

E.3.1. vp_formats_not_supported

- Name: `vp_formats_not_supported`
- Usage Location: authorization endpoint, token endpoint
- Protocol Extension: OpenID for Verifiable Presentations
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 8.5 of this specification

E.3.2. invalid_request_uri_method

- Name: `invalid_request_uri_method`

- Usage Location: authorization endpoint
- Protocol Extension: OpenID for Verifiable Presentations
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 8.5 of this specification

E.3.3. wallet_unavailable

- Name: `wallet_unavailable`
- Usage Location: authorization endpoint, token endpoint
- Protocol Extension: OpenID for Verifiable Presentations
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 8.5 of this specification

E.4. OAuth Authorization Server Metadata Registry

This specification registers the following authorization server metadata parameters in the IANA "OAuth Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [RFC8414].

E.4.1. vp_formats_supported

- Metadata Name: `vp_formats_supported`
- Metadata Description: An object containing a list of name/value pairs, where the name is a string identifying a Credential format supported by the Wallet
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 10 of this specification

E.5. OAuth Dynamic Client Registration Metadata Registry

This specification registers the following client metadata parameters in the IANA "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Parameters] established by [RFC7591].

E.5.1. encrypted_response_enc_values_supported

- Client Metadata Name: `encrypted_response_enc_values_supported`
- Client Metadata Description: Non-empty array of strings, where each string is a JWE [RFC7516] enc algorithm that can be used as the content encryption algorithm for encrypting the Response
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 5.1 of this specification

E.5.2. vp_formats_supported

- Client Metadata Name: `vp_formats_supported`
- Client Metadata Description: An object containing a list of name/value pairs, where the name is a string identifying a Credential format supported by the Verifier
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 11.1 of this specification

E.6. Media Types Registry

This section registers the following media type [RFC2046] in the IANA "Media Types" registry <xref target="IANA.MediaTypes"/> in the manner described in [RFC6838].

E.6.1. application/verifier-attestation+jwt

The media type for a Verifier Attestation JWT is `application/verifier-attestation+jwt`.

- Type name: `application`
- Subtype name: `verifier-attestation+jwt`
- Required parameters: n/a
- Optional parameters: n/a
- Encoding considerations: Uses JWS Compact Serialization as defined in [RFC7515].
- Security considerations: See Security Considerations in in [RFC7519].
- Interoperability considerations: n/a
- Published specification: Section 12 of this specification
- Applications that use this media type: Applications that issue, present, verify Verifier attestation VCs
- Additional information:
  - Magic number(s): n/a
  - File extension(s): n/a
  - Macintosh file type code(s): n/a
- Person & email address to contact for further information: TBD
- Intended usage: COMMON
- Restrictions on usage: none
- Author: Oliver Terbu, oliver.terbu@mattr.global
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net

E.7. JSON Web Signature and Encryption Header Parameters Registry

This specification registers the following JWS header parameter in the IANA "JSON Web Signature and Encryption Header Parameters" registry [IANA.JOSE] established by [RFC7515].

E.7.1. jwt

- Header Parameter Name: `jwt`
- Header Parameter Description: This header contains a JWT. Processing rules MAY depend on the `typ` header value of the respective JWT.
- Header Parameter Usage Location: JWS
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Specification Document(s): Section 12 of this specification

E.7.2. client_id

- Header Parameter Name: `client_id`
- Header Parameter Description: This header contains a Client Identifier. A Client Identifier is used in OAuth to identify a certain client. It is defined in [RFC6749], section 2.2.

- Header Parameter Usage Location: JWS
- Change Controller: IETF
- Specification Document(s): [RFC6749]

E.8. Uniform Resource Identifier (URI) Schemes Registry

This specification registers the following URI scheme in the IANA "Uniform Resource Identifier (URI) Schemes" registry [IANA.URI.Schemes].

E.8.1. openid4vp

- URI Scheme: `openid4vp`
- Description: Custom scheme used for Wallet invocation
- Status: Provisional
- Well-Known URI Support: -
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Section 13.1.2 of this specification

E.9. JSON Web Token Claims Registration

- Claim Name: "ld"
- Claim Description: JSON-LD object in compact form
- Change Controller: OpenID Foundation Digital Credentials Protocols Working Group - openid-specs-digital-credentials-protocols@lists.openid.net
- Reference: Appendix B.3.7 of this specification

Appendix F. Acknowledgements

We would like to thank Richard Barnes, Paul Bastian, Vittorio Bertocci, Christian Bormann, John Bradley, Marcos Caceres, Kim Cameron, Brian Campbell, Lee Campbell, Tim Cappalli, David Chadwick, Stefan Charsley, Gabe Cohen, Andrii Deinega, Rajvardhan Deshmukh, Giuseppe De Marco, Sander Dijkhuis, Mark Dobrinic, Pedro Felix, Daniel Fett, George Fletcher, Ryan Galluzzo, Timo Glastra, Sam Goto, Mark Haine, Martijn Haring, Fabian Hauck, Roland Hedberg, Joseph Heenan, Bjorn Hjelm, Alen Horvat, Andrew Hughes, Jacob Ideskog, Łukasz Jaromin, Edmund Jay, Michael B. Jones, Tom Jones, Judith Kahrer, Takahiko Kawasaki, Gaurav Khot, Niels Klomp, Ronald Koenig, Markus Kreusch, Adam Lemmon, Hicham Lozi, Daniel McGrogan, Jeremie Miller, Matthew Miller, Mirko Mollik, Kenichi Nakamura, Andres Olave, Gareth Oliver, Aaron Parecki, Nemanja Patrnogic, Joel Posti, Andreea Prian, Rolson Quadras, Javier Ruiz, Nat Sakimura, Dimitar Atanasov Stoikov, Arjen van Veen, Steve Venema, Jan Vereecken, David Waite, Jacob Ward, David Zeuthen, and Michael Zischg for their valuable feedback and contributions to this specification.

Appendix G. Notices

Authors' Addresses

Oliver Terbu
MATTR
Email: oliver.terbu@mattr.global

Torsten Lodderstedt
SPRIND
Email: torsten@lodderstedt.net

Kristina Yasuda
SPRIND
Email: kristina.yasuda@sprind.org

Daniel Fett
Authlete
Email: mail@danielfett.de

Joseph Heenan
Authlete
Email: joseph@heenan.me.uk