Technical Integration: AltID

Contents

1 Introduction 5

1.1 Terminology 6

2 Functional description 7

2.1 AltID – The Danish Digital Identity Wallet 7

2.2 Attestations in AltID 7
2.2.1 Proof of Identity (PID) 8
2.2.2 Proof of Age (PoA) 10

2.3 Relying Party Registry 10

3 Signed QR 11

3.1 About Signed QR 11
3.1.1 Sharing attestations with Signed QR 11

4 OpenID for Verifiable Presentations (OID4VP) 14

4.1 About OID4VP 14
4.1.1 Presenting attestations using OID4VP 14

4.2 OID4VP Authorization Requests supported by AltID 15
4.2.1 Request Object 16
4.2.2 Digital Credentials Query Language (DCQL) 17
4.2.3 Client Metadata 20
4.2.4 Verifier info 21

4.3 OID4VP flows supported by AltID 22
4.3.1 OID4VP presentation flow following [HAIP] 23
4.3.2 OID4VP age verification flow following [AVP] 28

5 Appendix A: Attestation data structures 31

5.1 Document 32

5.2 IssuerSigned 32

5.3 Issuer signature 32

5.4 Device signature 34

5.5 SessionTranscript for signed QR codes 34
5.5.1 mdocGeneratedNonce 35

5.6 SessionTranscript for OID4VP 35

5.7 Token Status Lists 36
5.7.1 Validation of attestation status using Token Status List 37

6 Appendix B: Validation of Signed QR presentations 39

6.1 QRPayload 39

6.2 Steps for validating a Signed QR presentation 39

7 Appendix C: Signed QR code example **41**

7.1 Multiple QR codes 41

7.2 Device signature validation 46

7.3 Attribute validation 47

8 Appendix D: Validation of OID4VP Authorization Responses **49**

8.1 Error Responses 50

9 Appendix E: OID4VP example for [HAIP] **51**

9.1 Registered URLs for AltID 51

9.2 Authorization Request 51

9.3 Authorization Response 54

10 Appendix F: OID4VP example for [AVP] **59**

10.1 Authorization Request 59

10.2 Authorization Response 60

11 Appendix G: Issuer Auth certificate chain and issuer signature validation **63**

11.1 Extracting the signing certificate 63

11.2 Trust 63

11.3 Trust lists 63

11.3.1 Trust lists for EU PoA attestations 63

11.3.2 Trust lists for AltID attestations 64

12 References **66**

Version	Description	Responsible	Date
0.9	First draft.	The Danish Agency for Digital Government	2025-10-28
1.0	- Added descriptions and examples for the HAIP compliant OID4VP-flow. - Updated data format for Signed QR. - Updated existing descriptions and examples to reflect the current state of the solution.	TNY & ANBRU	2026-04-24

1 Introduction

By Q2 2026 the Danish Agency for Digital Government will launch the Danish Digital Identity Wallet (DKTB), hereafter AltID. This document describes how a relying party, i.e. a Verifier, may utilize AltID for receiving attestations in online (remote) and proximity usage scenarios. The intention is to provide the technical information necessary for Verifiers to implement support for requesting and receiving attestations from AltID. The target audience for this document is therefore IT professionals that are interested in knowing how an attestation from AltID is requested and presented to an IT system.

Because this document focuses on receiving attestations it does not include a general introduction to the Danish Digital Identity Wallet project. If you as a reader are looking for a general introduction, we recommend reading the following paper before engaging with this document (albeit in Danish): Notat om den nationale digitale identitetstegnebog[1]. Should you have any questions about the documentation, they can be sent to altid@digst.dk.

The development of AltID stems from the eIDAS2-regulation. AltID is therefore, to a large degree built on standards and specifications agreed upon in the EU, enabling the use of AltID across the EU, among other things for age verification. If you are interested in knowing more about the related work in EU, see [ARF] and [AVP].

The technical documentation is structured as follows - The first section is the present introduction. The introduction also contains a terminology list. The second section presents a short description of the AltID app itself as well as a description of the two first attestations – Proof of Identity (PID) and Proof of Age (PoA) and the verification mechanisms underpinning the issuance and validation of these attestations. Section 3 and 4 present the two transmission protocols that can be used for receiving an attestation from AltID and implementation guidelines for these. The Appendices contain the data structures, guidelines for validation of the attestations as well as practical examples for each of the protocols. Figure 1 illustrates the contents in scope of this document:

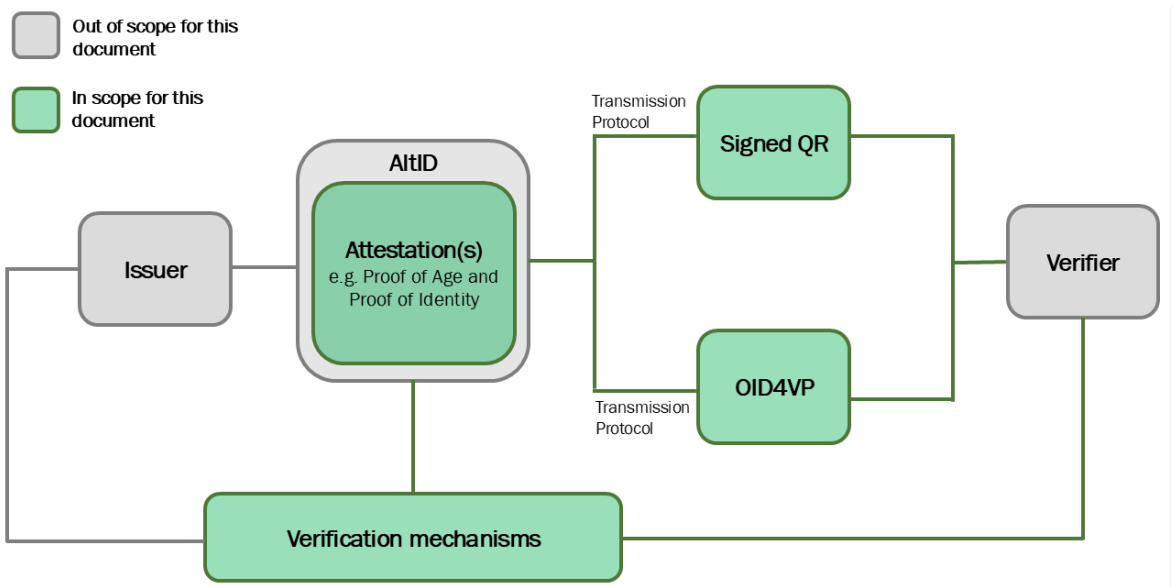

Figure 1: Overview of scope for this document

[1] Notat om AltID

1.1 Terminology

AltID: The Danish Digital Identity Wallet

Attestation (or **credential**): A set of one or more claims about a subject made by an Issuer

Issuer: A public entity that issues attestations to a User.

Wallet: An entity used by the User to receive, store, present, and manage Attestations and key material, i.e. the AltID app.

User: A natural person to whom a Wallet (AltID) has been issued and who uses it to receive, store, manage and present attestations.

Verifier: An entity that requests and verifies information presented by a User through the AltID system. In eIDAS2 terminology a Verifier is referred to as a relying party.

Presentation: Data that is presented to a Verifier, derived from an Attestation. I.e. Verifiers receive presentations rather than actual attestations.

OID4VP: OpenID for Verifiable Presentations 1.0 [OID4VP] – a specification from OpenID Foundation defining a protocol for requesting and presenting credentials/attestations over the internet.

Signed QR: A bespoke format and protocol for transmitting an attestation from the AltID where all information required for verification is contained within the signed payload of a multipart QR code.

Online scenarios: Situations where a Verifier interacts with the User through a digital service, such as a website or mobile application.

Proximity scenarios: Situations where the Verifier and the User are physically present in the same location.

Same-device flow: Situations where the Verifier is interacting with the User on the device where AltID is installed, e.g. a mobile browser or native app.

Cross-device flow: Situations where the Verifier is interacting with the User on a different device than the one where AltID is installed, e.g. a web browser (desktop) or Point-of-Sale system.

Status list: The Token Status List [TSL] provides a mechanism for representing attestation status, i.e. whether an attestation has been revoked, in a privacy-preserving manner while enabling highly efficient lookups.

2 Functional description

2.1 AltID – The Danish Digital Identity Wallet

AltID will function as a personal container that citizens can use to store and share digital attestations in a secure and privacy-preserving manner. Users can initially acquire AltID on their smartphone or tablet via Google Play or the Apple App store.

AltID is a personal app. It is therefore presumed that the User will not share it with other people, and that only the User can access and control their personal AltID. Ultimately, this means that all attestations in AltID are expected to pertain to and only be presented by the same User. This is enforced by requiring the User to authenticate using biometrics or PIN-code when using the app and only allowing one AltID application to be active per user.

2.2 Attestations in AltID

The attestations issued to AltID are "verifiable" in themselves — meaning that they contain the information required to create a verifiable presentation of one or more claims about the subject, which a Verifier can confirm as genuine and valid without conferring with a central AltID service.

The integrity and authenticity of the attestations is protected through Public Key Infrastructure (PKI) and the use of cryptographic mechanisms and digital certificates as specified in the [ISO18013-5] standard for mobile driving licenses (mDL). This means that all attestations in AltID are issued, stored and presented in the ISO mdoc format and encoded using Concise Binary Object Representation (CBOR). In the following a general overview is provided. For additional information about the ISO mdoc format we refer to [ISO18013-5] and Appendix A: Attestation data structures.

In the ISO mdoc format, attestations (mdocs) are identified by a document type (doctype). A doctype contains one or more namespaces, each of which contain one or more data elements (attributes presented as key-value pairs). In an online scenario, a verifier needs to specify both the doctype, the namespace(s) and the identifier of the specific attributes when making a request for a presentation. While the attestations stored in AltID always contain the full set of attributes, e.g. multiple age limits, it is possible, and advisable, to request a presentation of only necessary information from the attestation, whether that be a single attribute or a subset of the available attributes. With the Users' consent, AltID will generate a device-signed presentation derived from the stored attestation, containing only the attributes requested, i.e. perform what is known as *selective disclosure* of attributes.

Finally, the mdoc contains a Mobile Security Object (MSO), signed by the Issuer to prevent tampering, this information allows the verifier to verify the integrity, temporal validity and, depending on the type of attestation, the revocation status of the attestation. See Figure 2 for a graphical representation of an attestation in the ISO mdoc format.

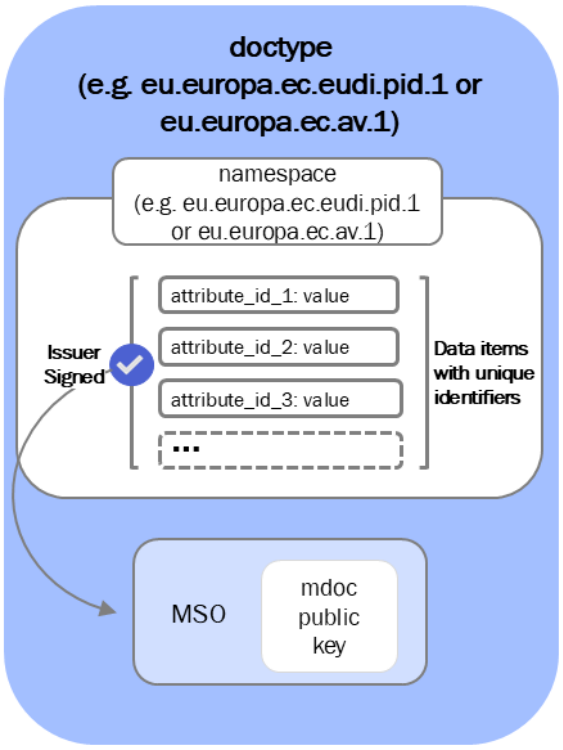

Figure 2: mdoc data model

At launch, AltID will include two different attestations: 1) Proof of Identity attestation which contains Personal Identification Data (PID) and 2) Proof of Age attestation (PoA) which contains attributes with different age limits. The following sections will elaborate further on the two attestations.

2.2.1 Proof of Identity (PID)

To activate AltID, the User is required to obtain a Proof of Identity attestation by getting a PID issued as part of the onboarding process[2]. The purpose of the PID is to enable the User to prove their identity in everyday situations. This is similar to how passports and driver's licenses are commonly used as identification in contexts other than travel and driving.

In this context, a distinction can be made between identification and legitimation. Identification refers to the act of presenting or claiming an identity, typically by providing identifying information or attributes. Legitimation refers to the act of demonstrating that the claimed identity is valid by presenting a trusted attestation.

The PID attestation functions as such an attestation by enabling the User to present verified identity attributes. When presented to a Verifier, these attributes can be used to legitimate the User's identity in a verifiable manner.

The attributes contained in a PID attestation is shown in Table 1 below. For more detailed information on the contents and data structure see Appendix A: Attestation data structures.

[2] Note: If a user uploads their passport photo, the attestation will be labelled 'Photo ID card' enabling AltID to be used in proximity scenarios.

Identifier	Description	Format (CBOR)	Example value
`full_name`	Full name of the User Note: the namespace for this specific attribute is "eu.europa.ec.eudi.pid.dk.1"	`tstr`	`Anders Monrad Carlsen`
`family_name`	Last name(s) of the User	`tstr`	`Carlsen`
`given_name`	First and middle name(s) of the User	`tstr`	`Anders Monrad`
`birth_date`	The User's date of birth	`full-date`	`1992-07-01`
`birth_place`	The area where the User's birth was registered. Typically, either a parish or a hospital, but can be any type of location. Note: while the format deviates from the current EU specification[3] it still uses the common PID namespace.	`tstr`	`Sundkirkens,Kbhvn`
`nationality`	An array of Alpha-2 country codes representing the User's nationalities[4]. "DK" will be listed first, if present.	`[+ tstr]`	`["DK"]`
`resident_address`[5]	Full address of the User	`tstr`	`Wildersgade 39,` `1408 København`
`personal_administrative_number`[5]	The User's CPR-number	`tstr`	`010792-0000`
`expiry_date`[6]	Specifies the expiry date of the document/data the credential is based on, not the credential itself. The credentials expiration is denoted by the `validUntil` attribute of the MSO.	`tdate`	`2027-01-01T00:00:00Z`
`issuing_authority`[6]	The Issuer of the PID attestation, always "Digitaliseringsstyrelsen" in AltID.	`tstr`	`Digitaliseringsstyrelsen`
`issuing_country`[6]	Country of the Issuer, always DK in AltID	`tstr`	`DK`

Table 1: PID attestation attributes

[3] The PID Rulebook: https://github.com/eu-digital-identity-wallet/eudi-doc-attestation-rulebooks-catalog/blob/main/rulebooks/pid/pid-rulebook.md
[4] Values are based on the "statsborgerskab"-attribute from CPR.
[5] Non-mandatory attributes, i.e. not always included in a PID. Will be included if available from CPR; address will only be included if the user has a Danish address.
[6] Classified as metadata, i.e. these attributes say something about the credential itself. Cannot be shared via Signed QR.

2.2.2 Proof of Age (PoA)

After having the PID issued, the User is able to request the issuance of a Proof of Age (PoA) attestation.

The purpose of the PoA attestation is that a user can prove that they are over or, by inference, under a specific age without sharing any additional personal information or metadata that can be linked to the specific user. The PoA attestation contains `age_over_NN` claims for the following age limits: 13, 15, 16, 18, 21, 23, 25, 27, and 67. The value of the attribute is a Boolean (true/false) specifying whether the User's age is greater than or equal to (true) or less than (false) the given age limit. For example, in the case of a 21-year-old user the attributes in the attestation would have following values:

```
{
    "age_over_13": true,
    "age_over_15": true,
    "age_over_16": true,
    "age_over_18": true,
    "age_over_21": true,
    "age_over_23": false,
    "age_over_25": false,
    "age_over_27": false,
    "age_over_67": false
}
```

The different age limits in the PoA attestation are intended for, but not limited to, different use cases:

- The attributes `age_over_13` to `age_over_18` are intended to be used to verify eligibility for age-restricted products or services, such as the sale of tobacco, alcohol or online content with an age limit.
- The attributes `age_over_21` to `age_over_27` are intended to be used to verify eligibility for age-restricted locations, such as nightclubs, bars, or events with specific age entry requirements.
- The attribute `age_over_67` is intended to be used to verify eligibility for seniors such as discounts or special services.

To prevent tracking of the User, a batch of individual one-time use PoA attestations is always issued to the User. Each attestation is only used to generate a single presentation and is deleted after use. This means that each PoA presentation is unique, and that colluding verifiers therefore are unable to conclude whether different presentations originate from the same User. PoA attestations do not make use of status (revocation) lists.

For more detailed information on the contents and data structure of the PoA attestation see Appendix A: Attestation data structuresAppendix A: Attestation data structures

2.3 Relying Party Registry

Organisations that intend to request attestations, **other than Proof of Age** (PoA), must register in the designated Relying Party Registry [RPR]. When the register opens for registration, this document will be updated with a link to the Relying Party Registry. For now, a link to the test environment for the Relying Party Registry version is included.

Registration enables AltID to display verified information about your organisation to Users, allowing them to make an informed consent decision before sharing their attestations. It also provides transparency by allowing everyone to see which organisations are authorised to request and receive attestation data from AltID, helping ensure that organisations only request the information they are entitled to.

After registering, organisations will receive a token that must be presented to AltID in order to prove that they have registered. The then contents of the token and the process of sharing it is described in greater detail in section 4.2.4.

3 Signed QR

3.1 About Signed QR

Signed QR is a bespoke protocol and format developed for AltID. It allows presentation and verification of attestations to take place through a signed QR code exchange between the AltID application and the Verifier.

Signed QR is designed for use in proximity scenarios, where a User presents a QR code containing a presentation to a Verifier. In this way, the Signed QR protocol allows both the user and the verifier to be offline during this process, as all information required for verification is contained within the signed payload of the QR code. However, the Verifier must be online to obtain status information for attestations where such information is present. It is recommended to always check the status of an attestation; however, it is left up to the Verifier to decide if the business process can be completed without validating the status of the attestation. See section 5.7 for more information about the validation of attestation statuses.

3.1.1 Sharing attestations with Signed QR

Sharing an attestation via the signed QR protocol is a four-step process (Figure 3):

1. **Presentation** of the attestation in the form of a QR code from the Users' AltID application
2. **Verifier scans** the displayed multipart QR code in its entirety
3. **Verifier assembles** the parts into a single, signed payload
4. **Verifier validates** the presentation and decides if the business process can continue or not

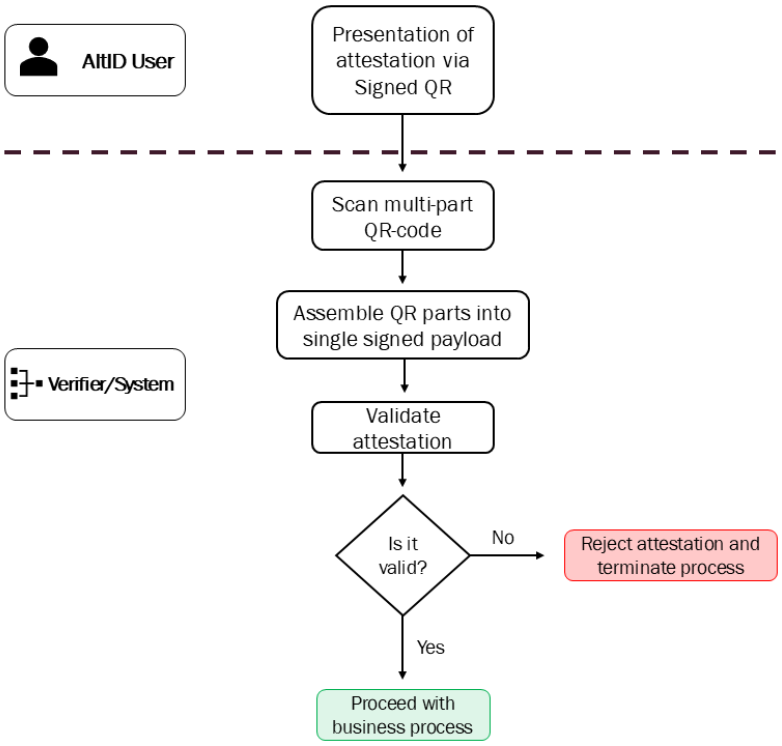

Figure 3: Signed QR verification flow

Step 1: Presentation of the QR code

A Verifier must request the User to present the relevant attestation from AltID. Upon this request, the User opens their AltID application and selects the desired attestation. The contents of the QR code are defined by the User who has the option to configure which attributes to disclose by selecting the button “Vælg, hvad du deler” (“Choose what you share”). Once selected, the signed multipart QR code is immediately generated and displayed on the User’s device (Figure 4).

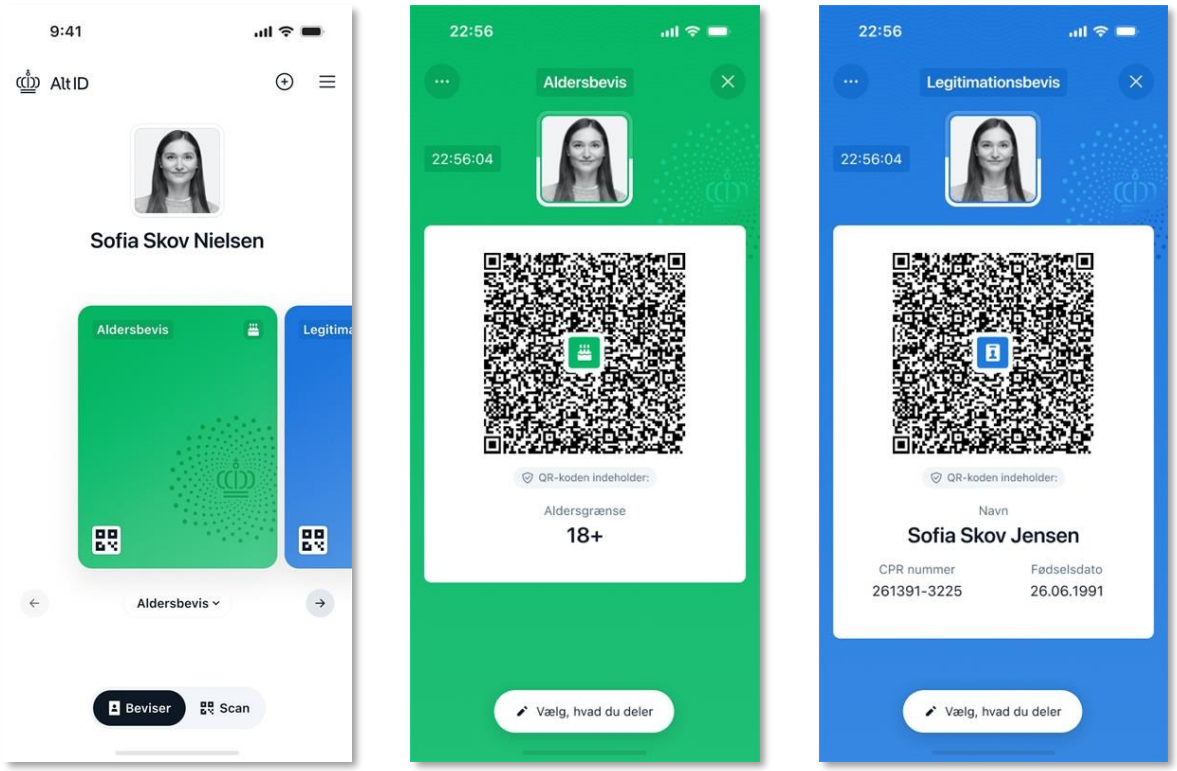

Figure 4: Mock-up of AltID dashboard with attestations and the display of signed QR codes

Steps 2 and 3: Scanning and assembling multipart QR codes

The presentation is split into multiple parts each represented by a QR code. These individual QR codes are shown in quick succession and looped. By making use of multipart QR codes, the size of a single QR code can be kept constant, while the only variable is the total amount of QR codes – parts – that make up the presentation. Figure 5shows an example of a multipart QR code consisting of four parts.

Figure 5: Multipart QR code

Each part contains a binary CBOR payload. The payload of each QR code part is structured as an ordered CBOR map as shown below:

```
PartialQRPayload = {
  "typ": tstr,          ; Type of QR code, will be "AltID-<version>", e.g. "AltID-1.0"
  "txn": tstr,          ; Transaction identifier, i.e. identifier for the resulting QRPayload
  "idx": uint,          ; Index of this part
  "cnt": uint,          ; Number of parts in total
  "part": bstr          ; Partial payload, i.e. part_idx
}
```

A camera or other QR code reader is used to scan all parts belonging to `txn`. When all parts have been scanned, i.e. the parts with `idx` values ranging from 0 to `cnt-1`, the full binary payload can be assembled. If the value of `txn` is altered at any point, the Verifier should discard any parts already scanned and start over.

Step 4: Validate the attestation

To validate the contents of the presentation, the steps outlined in Appendix B: Validation of Signed QR must be followed.

4 OpenID for Verifiable Presentations (OID4VP)

4.1 About OID4VP

OID4VP is a specification developed by the OpenID Foundation that defines a protocol for a secure and privacy-preserving request mechanism and delivery of so-called "Presentations of Credentials". Credentials (called attestations in this document) and Presentations can be of any format, including, but not limited to [ISO18013-5] mdoc, which is currently the only format supported by AltID. As such, OID4VP specifies how to request and present an attestation between a Wallet (AltID) and a Verifier.

The full specification is available at [OID4VP] and contains a wide range of options with details left up to implementers. To ensure interoperability, several profiles that specify exactly how to implement OID4VP are available. AltID supports [HAIP] for all use cases and [AVP] exclusively for age verification.

4.1.1 Presenting attestations using OID4VP

Overall, the OID4VP protocol is a mechanism that builds on top of OAuth 2.0. Thus, it follows a similar flow. In contrast to Signed QR, where the contents of the presentation are defined by the User, the Verifier must define the format and contents of the requested presentation when using OID4VP.

The OID4VP presentation flow begins when a verifier sends an Authorization Request to AltID, including a specification for the attestation(s) and attribute(s) that are requested. The URL containing the Authorization Request can be shown as a QR code that the user must scan to initiate the presentation flow, or if the interaction takes place on the same device as AltID is installed on, the User can instead follow a direct link containing the Authorization Request. Regardless of the method, interacting with the URL invokes the AltID app, and the presentation flow is initiated.

The flow is shown in Figure 6 where the User scans a QR code containing a presentation request from the Verifier using AltID. The app validates the request and presents the contents to the user; in the example the Verifier has specifically requested `age_over_16` from the PoA. The User then has to consent to, or reject, sharing a presentation containing the requested data with the Verifier. If the User consents, the requested presentation is generated and sent to the Verifier.

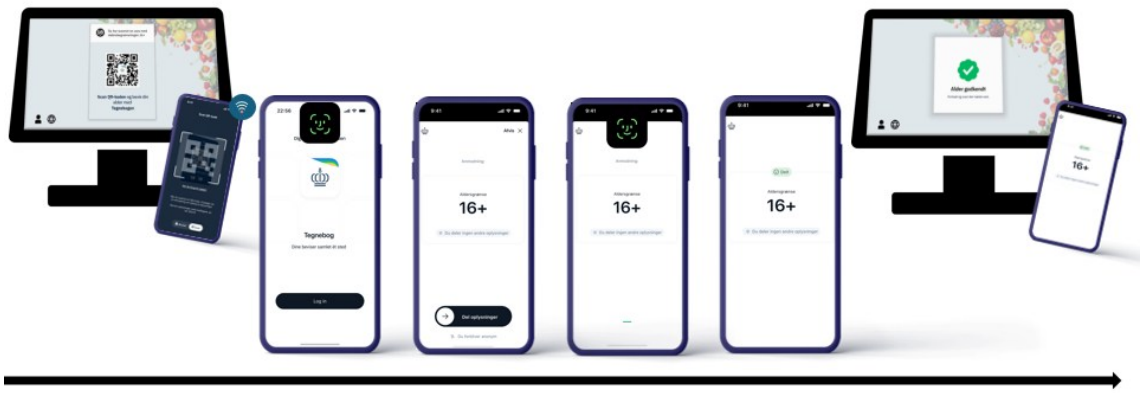

Figure 6: User flow for sharing a PoA via OID4VP

4.2 OID4VP Authorization Requests supported by AltID

As mentioned, [OID4VP] specifies a range of options, and as a result, offers support for several Authorization Request parameters and values. While [AVP] and [HAIP] narrows those options down, further specifications apply in the context of AltID.

For requests adhering to [HAIP], the bulk of the Authorization Request is encoded as a JWT Request Object (see section 4.2.1) and included by reference. The initial URL-encoded Authorization Request contains only `client_id` and a reference to the Request Object (`request_uri`) as specified in

Parameter	Description
`client_id`	MUST adhere to the [OID4VP] client identifier scheme `x509_hash`. This scheme defines the following format "x509_hash:<SHA-256(DER({Verifier certificate}))>", e.g. `x509_hash:Uvo3HtuIxuhC92rShpgqcT3YXwrqRxWEviRiA0OZszk`.
`request_uri`	MUST be the URI where the Request Object containing additional Authorization Request parameters can be fetched by AltID. See 4.2.1.

Table 2. AltID MUST be invoked via a platform-specific application link (i.e. Universal Link/App Link). See Appendix E: OID4VP example for [HAIP].

Parameter	Description
`client_id`	MUST adhere to the [OID4VP] client identifier scheme `x509_hash`. This scheme defines the following format "x509_hash:<SHA-256(DER({Verifier certificate}))>", e.g. `x509_hash:Uvo3HtuIxuhC92rShpgqcT3YXwrqRxWEviRiA0OZszk`.
`request_uri`	MUST be the URI where the Request Object containing additional Authorization Request parameters can be fetched by AltID. See 4.2.1.

Table 2: [HAIP] Authorization Request parameters

For requests adhering to [AVP], the entire Authorization Request is encoded as URL parameters, as specified in Table 3, and AltID MAY be invoked via the custom URL scheme "av://". See Appendix F: OID4VP example for [AVP]. Note that the custom URL scheme might invoke other applications installed on the User's device, as it is not reserved. Verifiers SHOULD use the application link if they want to ensure that the request is handled by AltID.

Parameter	Description
`client_id`	MUST adhere to the [OID4VP] client identifier scheme `redirect_uri`. This scheme defines the following format "redirect_uri:<response_uri>", e.g. `redirect_uri:https://verifier.example.com`.
`response_type`	MUST have the value `vp_token`, indicating that a successful Authorization Response must include a VP token including the requested presentation(s).
`response_mode`	MUST have the value `direct_post`, indicating that a successful Authorization Response is returned via an HTTP POST request (to the `response_uri`).

Parameter	Description
`response_uri`	MUST be the URI, where AltID should POST the Authorization Response, i.e. the requested presentation(s).
`dcql_query`	MUST be a JSON object specifying the attestations and attributes being requested in the form of a DCQL query. See 4.2.2.
`nonce`	MUST be unique and have a high amount of entropy (e.g. 128 bit). The device signature includes the `nonce`, and it serves the purpose of binding the device response cryptographically to the request, thereby preventing a class of replay attacks. See the flows in section 4.3 for examples of application.
`state`	OPTIONAL. If included the response by AltID will also include the `state` parameter value. This allows the verifier to connect a response to a specific internal transaction. See the flows in section 4.3 for examples of application.

Table 3: [AVP] Authorization Request parameters

4.2.1 Request Object

A Request Object is a JWT that contains the Authorization Request parameters, as defined in [JAR]. This JWT-Secured Authorization Request enables the transmission of larger and more complex Authorization Requests and provides integrity protection. The Verifier MUST sign the Authorization Request according to [JAR]. The structure of a Request Object in AltID is shown below, followed by a description of the claims in Table 4.

Header

```
{
  "x5c": ["…"],
  "typ": "oauth-authz-req+jwt",
  "alg": "RS256"
}
```

Body

```
{
  "aud": "https://self-issued.me/v2",
  "iss": "x509_hash:EPD6X85fbUtA5uMgM4XyVU3FHJ0rTfQVW_vUEGwtETY",
  "client_id": "x509_hash:EPD6X85fbUtA5uMgM4XyVU3FHJ0rTfQVW_vUEGwtETY",
  "response_type": "vp_token",
  "response_mode": "direct_post.jwt",
  "response_uri": "https://test-tool.test.tegnebog.dk/verifier/wallet/direct_post.jwt",
  "dcql_query": {…},
  "client_metadata": {…},
  "verifier_info": [{…}],
  "nonce": "b9b8b8b6-dfc2-4cc4-88e6-5507dca7bde4",
  "state": "88MaEDEsUeF9zjJZmJqOYGwbD14S9G70pavEGJi_ASI"
}
```

Claim	Description
`x5c`	MUST include the Verifier's signing certificate and any intermediate CA certificates. The root certificate SHALL NOT be included.
`typ`	MUST have the value `oauth-authz-req+jwt`, i.e. a [JAR] compliant Request Object.

Claim	Description
`alg`	MUST have the value `RS256`. Note that this is a deviation from [HAIP], which mandates support for `ES256`, however OCES does not currently support elliptic curves.
`aud`	MUST have the value `https://self-issued.me/v2` as specified by [OID4VP]
`iss`	MUST have the same value as `client_id`
`client_id`	MUST adhere to the [OID4VP] client identifier scheme `x509_hash`. This scheme defines the following format "x509_hash:<SHA-256(DER({Verifier certificate}))>", e.g. `x509_hash:Uvo3HtuIxuhC92rShpgqcT3YXwrqRxWEviRiA0OZszk`. MUST have the same value as the `client_id` provided in the Authorization Request. The corresponding private key MUST be used to sign the JWT.
`response_type`	MUST have the value `vp_token`, indicating that a successful Authorization Response must include a VP token including the requested presentation(s).
`response_mode`	MUST have the value `direct_post.jwt`, indicating that a successful Authorization Response must be an encrypted JWT and returned via an HTTP POST request.
`response_uri`	MUST be the URI, where AltID should POST the Authorization Response, i.e. the requested presentation(s).
`dcql_query`	MUST be a JSON object specifying the attestations and attributes being requested in the form of a DCQL query. See 4.2.2.
`client_metadata`	MUST be a JSON object including the key(s) to be used for encrypting the Authorization Response. See 4.2.3.
`verifier_info`	MUST be an array of objects and MUST at least include the signed JWT issued by [RPR] attesting to the Verifiers identity. See 4.2.4.
`nonce`	MUST be unique and have a high amount of entropy (e.g. 128 bit). The device signature includes the `nonce`, and it serves the purpose of binding the device response cryptographically to the request, thereby preventing a class of replay attacks. See the flows in section 4.3 for examples of application.
`state`	OPTIONAL. If included the response by AltID will also include the `state` parameter value. This allows the Verifier to connect a response to a specific request/transaction. See the flows in section 4.3 for examples of application.

Table 4: Request Object claims

4.2.2 Digital Credentials Query Language (DCQL)

Section 6 of [OID4VP] introduces the Digital Credentials Query Language (DCQL) which is a JSON-encoded query language that allows a Verifier to request presentations that match a given query. The language additionally allows the Verifier to apply a number of constraints to the request. Initially, AltID only supports the simplest form of

queries: requesting one or more attributes from a single attestation. Support for more advanced queries is planned, and this section will be expanded to reflect that once implemented. For now, DCQL queries MUST follow the structure presented below.

The following DCQL queries request presentations of all the attributes contained in the PoA and PID respectively. This serves only to showcase the structure of a DCQL query. In practice, Verifiers MUST limit their requests to only include the attributes necessary for their business operation.

```
// DCQL query requesting a PoA in its entirety
"dcql_query": {
    "credentials": [
       {
            "id": "K60LEvDmTYZI7P6wN-VQmCDucewt5IJdqJO_FJ8nxjM",  // Arbitrary string used to identify the
                                                                  // credential in the response.
            "format": "mso_mdoc",
            "meta": {
                "doctype_value": "eu.europa.ec.av.1"
            },
            "claims": [
               {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_13"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_15"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_16"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_18"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_21"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_23"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_25"
```

```
                    ]
                },
                {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_27"
                    ]
                },
                {
                    "path": [
                        "eu.europa.ec.av.1",
                        "age_over_67"
                    ]
                }
            ]
        }
    ]
}
// DCQL query requesting a PID in its entirety
"dcql_query": {
    "credentials": [
       {
            "id": "SKcHnGuV87FQOVhI13VcMRFkMsaP3Ez6Hwu404-eg1g",
            "format": "mso_mdoc",
            "meta": {
                "doctype_value": "eu.europa.ec.eudi.pid.1"
            },
            "claims": [
               {
                    "path": [
                        "eu.europa.ec.eudi.pid.dk.1",
                        "full_name"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "given_name"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "family_name"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "birth_date"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "birth_place"
                    ]
                },
               {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "resident_address"
```

```
                    ]
                },
                {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "personal_administrative_number"
                    ]
                },
                {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "nationality"
                    ]
                },
                {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "expiry_date"
                    ]
                },
                {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "issuing_authority"
                    ]
                },
                {
                    "path": [
                        "eu.europa.ec.eudi.pid.1",
                        "issuing_country"
                    ]
                }
            ]
        }
    ]
}
```

4.2.3 Client Metadata

[HAIP] requires responses to be encrypted. To enable this, the Verifier MUST include a `client_metadata` JSON object in the Request Object with the following structure:

```
"client_metadata": {
        "encrypted_response_enc_values_supported": [
            "A128GCM",
            "A256GCM"
        ],
        "jwks": {
            "keys": [
                {
                    "kty": "EC",
                    "use": "enc",
                    "crv": "P-256",
                    "kid": "kmOcIdb8CKIJ6LL-ehjvnpvGkqw8reHYFMqew2vgJjo",
                    "x": "3-oqRGJ1IpXI_2IRbcz6015kZ2HXMaUOl6eMPwoaOUA",
                    "y": "TvYPWB-r9qDHcefxRMRRrUc8DMSD36M7xJ9WRAOuco4",
                    "alg": "ECDH-ES"
                }
            ]
        },
        "vp_formats_supported": {
```

```
            "mso_mdoc": {
                "issuerauth_alg_values": [
                    -7
                ],
                "deviceauth_alg_values": [
                    -7
                ]
            }
        }
    }
```

`jwks` MUST contain exactly one ephemeral key in JWK format to be used for encrypting the response. The encryption key MUST include a `kid` parameter which the Verifier MUST be able to use to correlate the response. The parameters `kty`, `use`, `crv`, and `alg` MUST contain exactly the values provided above.

`encrypted_response_enc_values_supported` and `vp_formats_supported` MUST contain exactly the values provided above as specified by [HAIP].

4.2.4 Verifier info

The `verifier_info` parameter included in the Authorization Request is a list of objects, each containing a `format` parameter, specifying the type of object, and a `data` parameter containing the object itself, e.g. a JWT.

In the context of AltID, `verifier_info` allows the Verifier to prove to AltID that it is a registered relying party in the form of a signed data-structure issued by [RPR]. Upon registration, the Verifier is issued a JWT, which MUST be included as the `data` in a Verifier Info object where `format` is `dktb-rpr+jwt`.

A Verifier MAY additionally include an object where `format` is `dktb-info+json` and `data` is a JSON object containing a single claim, `name`, i.e. `"data": {"name": "My AltID Verifier"}`. This can be used to provide a name, which will be shown to the User in AltID instead of the organisations primary name in CVR. Note that the Authorization Request will be rejected if the provided `name` is not present in `allowed_names` (see below).

The Verifier Info object with `format` `dktb-rpr+jwt` provides a JWT signed by [RPR] with the following structure:

Header

```
{
  "x5c": […],
  "typ": "dktb-rpr+jwt",
  "alg": "ES256"
}
```

Body

```
{
  "sub": "x509_hash:EPD6X85fbUtA5uMgM4XyVU3FHJ0rTfQVW_vUEGwtETY",
  "iss": "https://relying-party-registry.test.tegnebog.dk",
  "allowed_names": [
    "Digitaliseringsstyrelsen"
  ],
  "exp": 1797443329,
  "iat": 1773726826,
  "jti": "0e6e4156-d895-43d8-b558-7ed8dabb1352",
  "status": {
    "status_list": {
      "idx": 25296,
      "uri": "https://dktb-rpr-cdn.test.myracdn.com/status-lists/2e95bda0-769a-4092-9cc5-
a43e5dbb7a79"
    }
  }
}
```

`x5c` includes the [RPR] signing certificate, trusted by the AltID app.

`sub` includes the `client_id` for the relying party (Verifier) that the JWT is issued to. AltID verifies that `sub` claim matches `client_id` provided in the Authorization Request.

`allowed_names` provides a list of approved names, that the Verifier may present itself by. The list includes the organisation's primary and secondary names ("binavne") collected from CVR. By default, the organisation's primary name will be the first entry in the list and shown to the User in AltID. It is possible to specify that another approved name should be treated as the primary name via the [RPR] GUI. Alternatively, another name can be specified by including the `dktb-info+json` object in the request, allowing for adjusting it per request.

`status` provides information allowing AltID to verify that the JWT has not been revoked by [RPR]. See 5.7.

The remaining claims adhere to the common [JWT] ruleset.

4.3 OID4VP flows supported by AltID

While the overall OID4VP flow is the same, this section describes the specifics of flows adhering to either the "High Assurance Interoperability Profile" or the "Age Verification Profile".

The [HAIP] flow can be used to request all types of attestations (including PoA) and should be seen as the default implementation of OID4VP in the context of AltID.

The [AVP] flow can only be used for requesting PoA. It is supported to allow for a simplified, use-case specific Verifier implementation, and to ensure alignment with ongoing EU-initiatives related to online age-verification. The main simplification of [AVP] compared to [OID4VP] is that Verifiers are not required to register in the [RPR]. Instead, Verifiers are authenticated simply by a URI. For an elaboration on this see section 10.1.

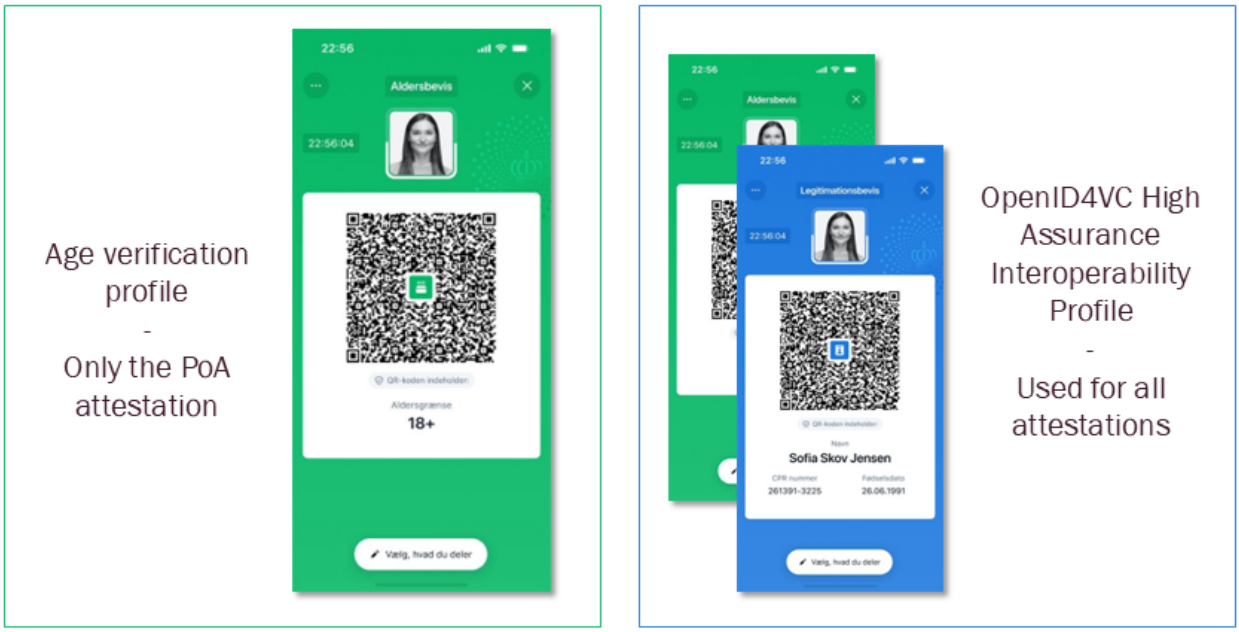

Figure 6: Difference between Age Verification Profile and High Assurance Interoperability Profile

4.3.1 OID4VP presentation flow following [HAIP]

Figure 7 below shows a non-normative sequence diagram for an exchange following the OID4VP [HAIP] flow. The figure assumes that the Verifier application consists of a frontend part (Client) and a REST-oriented Backend, however other design patterns can be applied based on the business needs and existing infrastructure of the Verifier.

OID4VP, and by extension [HAIP], defines the exchange of Authorization Request and Authorization Response between a Verifier and a Wallet (AltID), thus the steps directly related to these exchanges are mandatory. The specifics of the interaction between Verifier Client and Verifier Backend, and between the Verifier and the User, is not specified by the OID4VP protocol and the description in Table 5 merely suggests an implementation strategy.

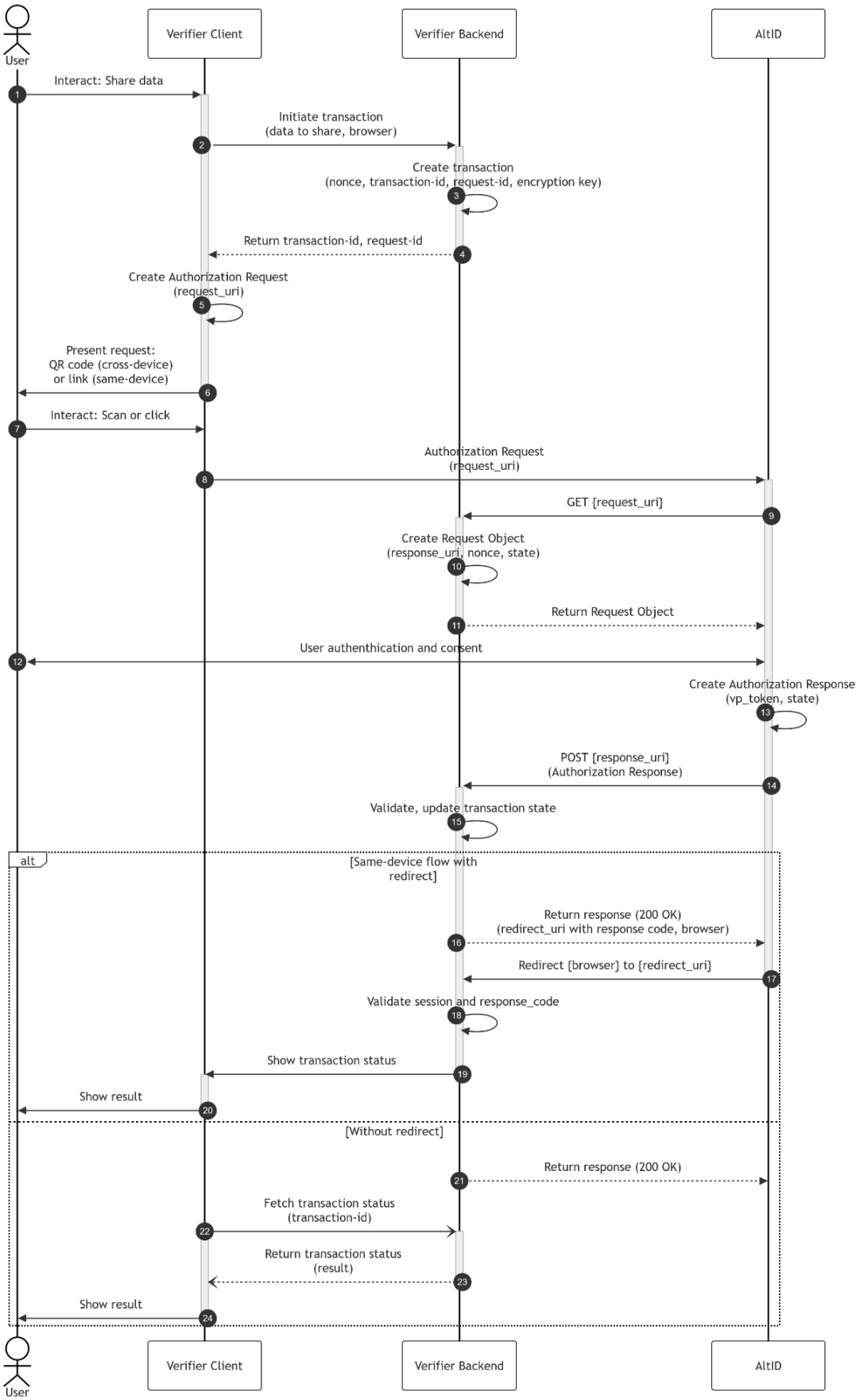

Figure 7: Example of OID4VP credential presentation flow following [HAIP].

Step	Description
1	User interacts with a Verifier, typically through a web browser, a native app or another type of user-facing system, e.g. a POS-system, in some way, that requires them to share data from an attestation.
2	Verifier Client initiates a new transaction at a dedicated Verifier Backend application, passing information on the data to share and, if applicable, the used browser application, the latter detected e.g. from User Agent header by conventional means (JavaScript library or similar).
3	Verifier Backend creates a transaction record with an associated random `nonce`. Two cryptographically random identifiers are also associated with the transaction: `transaction-id`, used by the Verifier Client to eventually obtain the response, and `request-id`, used to link the transaction with the corresponding Authorization Response. An ephemeral encryption key is also generated. This key is used by AltID to encrypt the response. Verifier Backend must associate a unique key identifier with the encryption key to allow the transaction record to be retrieved using the key identifier. It is recommended to use `request-id` as key identifier. Finally, for same-device (redirect) flows, Verifier Backend creates an HTTP session (cookie based) and stores the `transaction-id` in the session.
4	Verifier Backend returns `transaction-id` and `request_uri` containing the `request-id` to Verifier Client.
5	Verifier Client forms an Authorization Request, containing the `request_uri` as request parameter.
6	Verifier Client decides how to present the Authorization Request to the User: If the client detects that the User is using a mobile device, it is likely that AltID is installed on that same device (same-device flow). In that case, the client should present the Authorization Request as a button/link ("Share data") in the UI. It is recommended to display another button ("Share data using other device"), that causes a QR code to be displayed. See [UX] for UI implementation guidelines. If not, the client should encode the Authorization Request in a QR code[7] and display this in the UI. This allows the User to scan the QR code using their mobile device with AltID installed (cross-device flow).
7	User interacts with UI – clicks button/link (same-device) or scans QR code (cross-device).
8	Authorization Request, containing the `request_uri`, is sent to AltID.
9	AltID requests the Request Object at the provided `request_uri`.
10	Verifier Backend then creates a corresponding [JAR] Request Object, including the `response_uri`, `nonce`, encryption key and `state` (using `request-id` as value) in addition to the specification of the desired content of the presentation in the form of a DCQL query (see section 4.2.2).
11	Verifier Backend returns the Request Object to AltID which validates the request and ensures that it conforms to [HAIP].
12	User interacts with AltID and consents to present the requested attributes to the Verifier.

[7] This can be done by utilizing one of the many available open-source libraries for QR-code generation, such as https://github.com/nayuki/QR-Code-generator/

13	AltID creates the Authorization Response (`vp_token`, `state`), where `vp_token` is a base64-url encoded `DeviceResponse`. This involves constructing the `SessionTranscript` and `DeviceAuthenticationBytes`, and creating the device signature, using the key that was bound to the attestation during issuance. Finally, AltID forms an encrypted JWT with Authorization Response as payload, using the ephemeral encryption key. The unique key identifier is included as header parameter in the encrypted JWT. For details, see Appendix A: Attestation data structures.
14	AltID POSTs the encrypted JWT to the `response_uri` provided by the Verifier in the Request Object.
15	Verifier Backend receives the response, performs validation and updates the transaction record state correspondingly to prevent replay. See Appendix D: Validation of OID4VP Authorization Responses. If validation fails, the transaction should be marked as failed, and a (user friendly) error description should be added to the transaction record. Further, Verifier Backend may respond with an error response (HTTP 400) only if the failure is technical in nature, i.e. if the user's data does not meet the business requirements, but was received correctly, the Verifier Backend should still return a 200 OK.
16	**Same-device**: Verifier Backend returns a 200 OK response containing a `redirect_uri` (including a cryptographically random `response_code`) and the `browser`-value passed by Verifier Client in step 2.
17	AltID redirects the specified `browser` to the specified `redirect_uri`.
18	Verifier Backend receives the redirect request, containing the `response_code`. Verifier Backend looks up the `transaction-id` in the HTTP session and uses that to look up the corresponding transaction record. It then verifies that the transaction is in correct state, and that the `response_code` in the redirect request matches the response code stored in the transaction record. If that is the case, the transaction is marked as successful. If any step fails, the transaction is marked as failed and the received VP Token is deleted. Further, Verifier Backend uses the `response_code` received in the redirect request to look up the intended transaction record and marks the found transaction as failed. The last step ensures that the response code is invalidated even if no transaction id was found in the session.
19 & 20	As response to the redirect request, Verifier Backend displays page to Verifier Client/User, Verifier shares the result with the User, usually by simply continuing the business operation for successful presentations and by displaying an error message in case of error.
21	**Alternative (cross-device):** Verifier Backend returns a 200 OK response with an empty JSON-body.
22	Verifier Client retrieves transaction status using `transaction-id` as response of long-polling.
23	Verifier Backend looks up transaction record using `transaction-id` and returns a result record (status/error message, values of requested attributes).
24	Verifier shares the result with the User, usually by simply continuing the business operation for successful presentations and by displaying an error message in case of error.

Table 5: Description of each step in the OID4VP flow following [HAIP].

See Appendix A: Attestation data structures for a normative example highlighting the exchanged data elements.

4.3.1.1 Real-time phishing protection

The use of `redirect_uri` and `response_code` allows the Verifier to protect presentation flows against real-time phishing attacks. The controls, as described above, require that the user's browser holds the session cookie for the original browser initiating the request and therefore prevents the user from authorizing an attacker's browser. Note that private/incognito tabs do not support this, as the redirect will invoke a "normal" tab without access to the session cookie. If Verifiers deem such protection necessary, based on the risk associated with their specific business operation, they should only allow same-device flows, mandating response code validation, and not allowing cross-device (QR code) flows, as these cannot be protected by this mechanism.

Verifiers MUST use the same-device flow with redirect for presentations containing PID, unless the Verifier does not rely on session binding for phishing resistance, e.g. in a proximity scenario. In general, Verifiers SHOULD NOT support the cross-device flow for presentations containing sensitive data.

Verifiers using flows without redirect SHOULD limit the lifetime of their Authorization Requests (e.g. by assigning an expiration time to the transaction record) to reduce the risk of replay or relay attacks.

4.3.1.2 Browser detection

When AltID receives the Authorization Request in same-device flows, AltID has no way of knowing which browser made the request. For the redirect flow to be successful on the AltID device, the Verifier must detect which browser is used and convey that information to AltID (step 2) so AltID can open the `redirect_uri` in the `browser` (step 18) containing the session cookie. Supported User Agents and corresponding values for `browser` (step 2) are shown in Table 6 below.

User Agent	Value for "browser"-parameter
Default browser. To be used for app-switching.	`"app"`
Apple Safari	`"safari"`
Brave	`"brave"`
DuckDuckGo	`"duck"`
Google Chrome	`"chrome"`
Microsoft Edge	`"edge"`
Mozilla Firefox	`"firefox"`
Opera	`"opera"`
Samsung Internet	`"samsung"`

Table 6: Supported "browser"-parameter values

4.3.1.3 Redirect and termination of same-device flows

Although it is possible to use a long-polling approach for same-device flows, and thereby aligning the two flows, it is recommended to use the outlined approach. The reason is, that it is difficult to seamlessly navigate the user back to the initial browser pane from where the verification was initiated in step 8.

By continuing the flow from the `redirect_uri` page, a better user-experience can be obtained. The obvious drawback of leaving the initial page/tab open may be mitigated by displaying a message to the user on that page either immediately after user has invoked the app-switch link and opened AltID app or when receiving the response from a long-polling request. In the latter case, the response handler for the long-polling request has different purposes for same-device and cross-device flows, being responsible for continuing the flow correctly for cross-device flows (step 24) and merely explaining to the user, that the (initial) tab may be closed (same-device).

4.3.2 OID4VP age verification flow following [AVP]

A non-normative example of an interaction between a Verifier and AltID in an OID4VP age verification flow following [AVP] is shown below in Figure 8. Similar to the [HAIP] example, the Verifier application is assumed to be split into a frontend (Client) and a Backend, and the description in Table 7 is only a suggested implementation strategy.

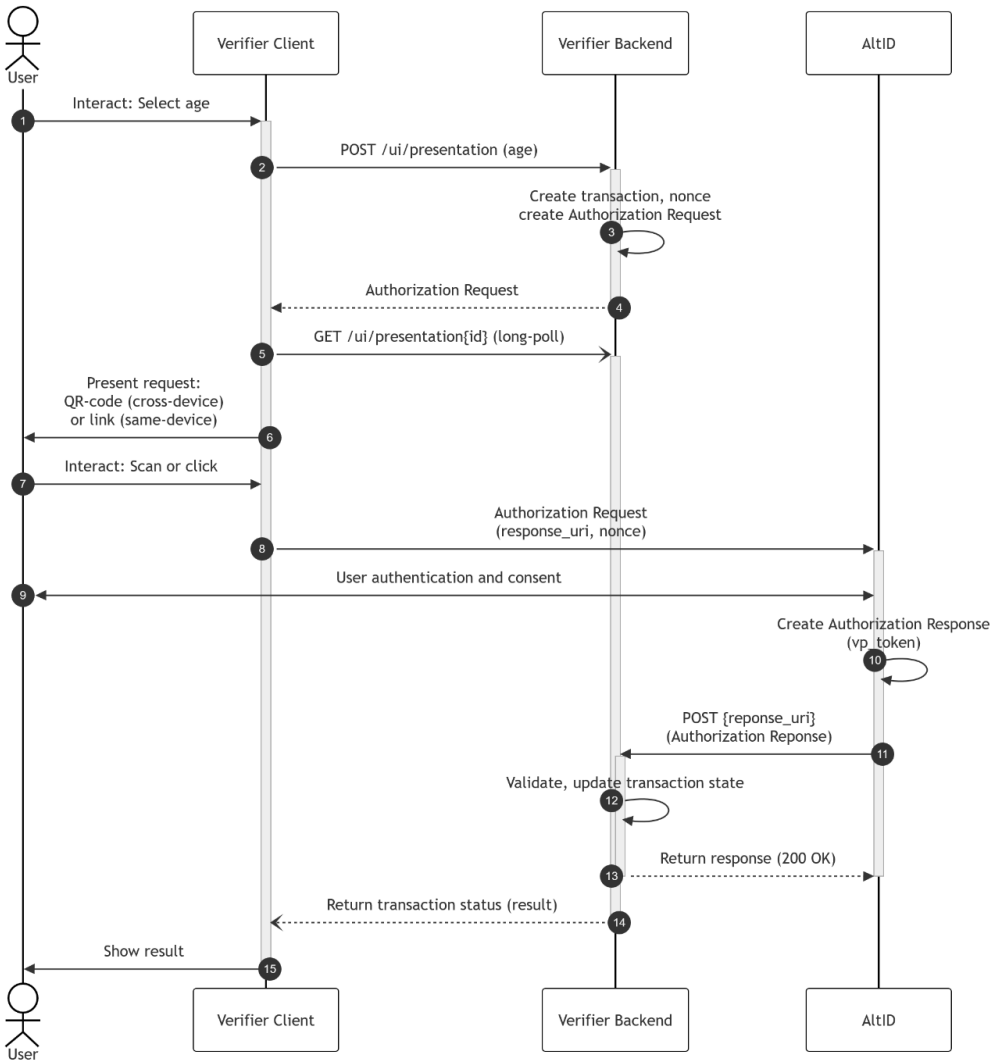

Figure 8: Example of OID4VP age verification flow following [AVP].

Step	Description
1	User interacts with Verifier Client, in some way, that requires their age to be verified. The diagram suggests that the User selects the age to be verified, usually this is a choice made by the Verifier.
2	Verifier Client sends the age(s) to be verified to Verifier Backend, initiating a new transaction.
3	Verifier Backend creates a transaction record with a unique identifier (id). A random `nonce` is generated and associated to the transaction, and the transaction record is stored. Verifier Backend then creates a corresponding OID4VP Authorization Request (as URL encoded values), including a specification of the desired age(s) to verify in the form of a DCQL query (see section 4.2.2).
4	The Authorization Request is returned to Verifier Client as response to step 2.
5	Verifier Client starts a (long)-polling request, awaiting the AltID response. Long-polling is suggested for optimization reasons.
6	Verifier Client now decides how to present the Authorization Request to the User: If the client detects that the User is using a mobile device, it is likely that AltID is installed on that same device (same-device flow). In that case, the client should present the Authorization Request as a button/link ("Verify age") in the UI. It is recommended to display another button ("Verify age using other device"), that causes a QR code to be displayed. See [UX] for GUI implementation guidelines. If not, the client should encode the Authorization Request in a QR code[8] and display this in the UI. This allows the User to scan the QR code using their mobile device with AltID installed (cross-device flow).
7	User interacts with UI – clicks button/link (same-device) or scans QR code (cross-device).
8	Authorization Request is sent to AltID which validates the request and ensures that it conforms to [AVP].
9	User interacts with AltID and accepts to present the requested attributes of the PoA to the Verifier.
10	AltID creates the Authorization Response (`vp_token`), which is an encoded `DeviceResponse`. This involves constructing the `SessionTranscript` and `DeviceAuthenticationBytes`, and creating the device signature, using the key that was bound to the PoA during issuance.
11	AltID POSTs the Authorization Reponse to the `response_uri` provided by the Verifier in the Authorization Request.
12	Verifier Backend receives the response, performs validation and updates the transaction record state correspondingly to prevent replay. If validation fails, transaction should be marked as failed, and a (user friendly) error description should be added to the transaction record. For details, see Appendix D: Validation of OID4VP Authorization Responses

[8] This can be done by utilising one of the many available open-source libraries for QR-code generation, such as https://github.com/nayuki/QR-Code-generator/

Step	Description
13	Verifier Backend responds 200 OK to AltID POST. Verifier Backend may respond with an error response (HTTP 400) only if the technical validation of the response failed, i.e. if the user's data does not meet the business requirements, but was received correctly, the Verifier Backend should still return a 200.
14	With the transaction now being completed the (long)-polling request can be replied to, in accordance with the status determined by Verifier Backend validation in step 12. The response should include ok/not ok and a suitable error message in case of errors.
15	Verifier Client shares the result with User, usually by simply continuing the business operation for successful age verification and by displaying an error message in case of error.

Table 7: Description of each step in the OID4VP age verification flow following [AVP].

See Appendix A: Attestation data structures for a normative example highlighting the exchanged data elements.

5 Appendix A: Attestation data structures

This appendix will present and describe the data structure(s) related to the presentation of attestations from AltID.

Attestations in AltID adhere to the [ISO18013-5] mobile document format (mdoc), which encodes data using CBOR (Concise Binary Object Representation), a binary alternative to JSON optimised for size and speed. CBOR preserves data types precisely and is suitable for signing and verification but is not human-readable; every field and structure must be decoded to inspect its content. When working with mdoc data, you can use decoding tools or libraries such as [CBORZ] to convert it to diagnostic notation for inspection.

To make the specification of mdoc easier to understand, it can be viewed as hierarchical binary tree, where each node represents a CBOR map or array, and each leaf contains a concrete data value (string, number, or byte string). In the context of presentations, the `DeviceResponse` structure is the root, which branches out into `version`, `documents` and `status`, where `documents` contain one or more `Document` structures, which again branches into `docType`, `issuerSigned` and `deviceSigned`. Each of these nodes can in turn contain nested maps, for example `issuerSigned` → `nameSpaces` → "`eu.europa.ec.av.1`" →...→ "`age_over_18`" & "`true`".

A simplified diagram of the relevant elements of the `DeviceResponse` structure, based on CDDL notation, is provided in Figure 9 below. Note that while OID4VP presentations contain a full `DeviceResponse` structure, Signed QR presentations contain only the `Document` structure. However, in the former case, the value of `version` is always "1.0" while the value of `status` is always 0.

Thus, the point of interest is the `Document` structure, which is presented in detail in the following sections.

```
DeviceResponse
├── version : tstr
├── ? documents : [+ Document]
│   └── Document
│       ├── docType : DocType
│       ├── issuerSigned
│       │   ├── ? nameSpaces : IssuerNameSpaces => [+ IssuerSignedItemBytes]
│       │   │   └── IssuerSignedItemBytes = #6.24(bstr .cbor IssuerSignedItem)
│       │   │       └── IssuerSignedItem
│       │   │           ├── digestID : uint
│       │   │           ├── random : bstr
│       │   │           ├── elementIdentifier : DataElementIdentifier
│       │   │           └── elementValue : DataElementValue
│       │   └── issuerAuth : IssuerAuth
│       └── deviceSigned
│           ├── nameSpaces : DeviceNameSpacesBytes
│           │   └── DeviceNameSpacesBytes = #6.24(bstr .cbor DeviceNameSpaces)
│           └── deviceAuth
│               └── deviceSignature : DeviceSignature
└── status : uint
```

Figure 9: ISO mdoc `DeviceResponse` structure

5.1 Document

`Document` is the attestation itself and has the following CBOR syntax:

```
Document = {
   "docType" : DocType,           ; Document type returned, e.g. "eu.europa.ec.av.1" for a PoA
   "issuerSigned" : IssuerSigned, ; Returned data elements signed by the issuer
   "deviceSigned" : DeviceSigned, ; The Wallet (AltID) signature
}
```

5.2 IssuerSigned

Here `IssuerSigned` contains the attestation including the attribute(s) disclosed by the user:

```
IssuerSigned = {
  ? "nameSpaces" : IssuerNameSpaces,  ; Returned data elements
  "issuerAuth" : IssuerAuth           ; Contains the Mobile Security Object (MSO)
}
```

Here `IssuerNameSpaces` contains the attributes included in the presentation:

```
IssuerNameSpaces = {                            ; Returned data elements for each namespace
   + NameSpace => [+ IssuerSignedItemBytes]
}

IssuerSignedItemBytes = #6.24(bstr .cbor IssuerSignedItem) ; Tagged bstr[9]

IssuerSignedItem = {
   "digestID" : uint,                           ; Digest ID for issuer data authentication
   "random" : bstr,                             ; Random value for issuer data authentication
   "elementIdentifier" : DataElementIdentifier, ; Data element identifier, e.g. "age_over_18"
   "elementValue" : DataElementValue            ; Data element value, e.g. "true"
}
```

5.3 Issuer signature

`IssuerAuth` contains the `MobileSecurityObject` (MSO) which is a [COSE] (COSE_Sign1) signature:

```
IssuerAuth = COSE_Sign1                ; As defined in section 4.2 of [COSE]

COSE_Sign1 = [                         ; Contents of the COSE_Sign1 structure of IssuerAuth
   ProtectedHeaderBytes : bstr,        ; Encoded map with header elements included in signature
   UnprotectedHeader,                  ; Map with header elements not included in signature
   Payload : MobileSecurityObjectBytes,; CBOR encoded MSO bytes
   Signature: bstr                     ; The signature
}

ProtectedHeader = {
   1 : int/tstr                        ; Algorithm identifier. Value is always "-7" for AltID
                                       ; Corresponds to ES256 (ECDSA w/ SHA-256)
}
```

[9] Syntax #6.24 means a CBOR tag (major type 6), tag number 24 (binary data), see https://www.rfc-editor.org/rfc/rfc8949.html#name-encoded-cbor-data-item

```
UnprotectedHeader = {
   33 : [+ bstr] / bstr
              ; Array with X.509 certificate chain (33) or single bstr with issuer certificate
              ; If given as chain (array), chain begins with issuer certificate
              ; and ends with root-certificate.
              ; If given as single certificate, only issuer certificate with public key
              ; needed to validate MSO is provided.
}

MobileSecurityObjectBytes = #6.24(bstr .cbor MobileSecurityObject)
```

For AltID, the protected header always contains a single element, stating that the signature algorithm used is ECDSA with SHA-256 ("`1: -7`"), and the unprotected header always contains a single element, with the issuer certificate and corresponding certificate chain.

The `Payload` is the CBOR encoded Mobile Security Object:

```
MobileSecurityObject = {
   "version" : tstr,                ; Version of the MSO, always "1.0"
   "digestAlgorithm" : tstr,        ; Message digest algorithm used, always "SHA-256"
   "valueDigests" : ValueDigests,   ; Digests of all data elements per namespace
   "deviceKeyInfo" : DeviceKeyInfo, ; Device key to use for presentation
   "docType" : tstr,                ; Same as Document DocType, e.g. "eu.europa.ec.av.1"
   "validityInfo" : ValidityInfo,   ; Temporal validity information
   ? "status" : Status              ; Status (revocation) information, not included in PoA
}

ValueDigests = {
  + NameSpace => DigestIDs   ; DigestIDs for a given NameSpace, PoA always contains only a
}                            ; single NameSpace: "eu.europa.ec.av.1" while PID can contain both
                             ; "eu.europa.ec.eudi.pid.dk.1" and "eu.europa.ec.eudi.pid.1".

DigestIDs = {
  + DigestID => Digest       ; Map from DigestID to Digest
}

DigestID = uint              ; DigestID corresponding to the one(s) in IssuerSignedItem
Digest = bstr                ; Digest of the IssuerSignedItems used for data authentication

DeviceKeyInfo = {
   "deviceKey" : DeviceKey,
   ? "keyAuthorizations" : KeyAuthorizations ; Element included but not used in AltID.
}

DeviceKey = COSE_Key         ; Public key of the device key-pair used to sign the presentation

ValidityInfo = {
   "signed" : tdate,         ; Timestamp at which the MSO signature was created
   "validFrom" : tdate,      ; Timestamp before which the MSO is not yet valid
   "validUntil" : tdate,     ; Timestamp after which the MSO is no longer valid
}

Status = {
   "status_list" : StatusListInfo
```

```
}
StatusListInfo = {
   "idx" : uint,        ; The index to check for the presented attestation's status information
   "uri" : tstr         ; Identifies the status list containing the status information for the
                        ; attestation. See section 5.7 for more information on status lists
}
```

The MSO signature (`IssuerAuth`) can be verified using the public key from the issuer certificate. This is obtained by the Verifier as an unprotected header element (X.509 certificate chain, `x5chain`, label 33).

5.4 Device signature

`DeviceSigned` holds the signature created by AltID using the device key, for which the public key is included in the Mobile Security Object.

```
DeviceSigned = {
   "nameSpaces" : tagged empty bstr,    ; Returned data elements, always empty for AltID
   "deviceAuth" : DeviceAuth            ; Contains the device signature
}

DeviceAuth = { ;
   "deviceSignature" : DeviceSignature  ; Signature used for mdoc (device) authentication
}
```

Where `DeviceSignature` is a [COSE] (COSE_Sign1) signature over `DeviceAuthentication` data:

```
DeviceSignature = COSE_Sign1

COSE_Sign1 = [                          ; Contents of the DeviceSignature
   ProtectedHeaderBytes : bstr,         ; Encoded map with header elements included in signature
   UnprotectedHeader,                   ; Map with header elements not included in signature
                                        ; This map is empty for DeviceSignature
   Payload : DeviceAuthenticationBytes,; Tagged and encoded DeviceAuthentication
                                        ; This is null when transmitted
   Signature : bstr                     ; The signature
}

DeviceAuthenticationBytes = #6.24(bstr .cbor DeviceAuthentication)

DeviceAuthentication = [
   "DeviceAuthentication",
   SessionTranscript,                ; See section 7.2
   DocType,                          ; Same as Document DocType, e.g. "eu.europa.ec.av.1"
   DeviceNameSpacesBytes             ; Always tagged bstr of empty map: #6.24(bstr .cbor {})
]
```

Note that `DeviceSignature` is *detached*, meaning that the `COSE_Sign1` data contains a null value for the `Payload` element when transmitted. As a result, the payload bytes (`DeviceAuthenticationBytes`) must be **reconstructed** prior to validating the signature. Also note, that the only variable data in `DeviceAuthentication` is `SessionTranscript`.

5.5 SessionTranscript for signed QR codes

The `SessionTranscript` takes a special form for `Document` presented as a signed QR code:

```
SessionTranscript = [
  DeviceEngagementBytes : bstr,             ; Always null for signed QR presentations
```

```
  EReaderKeyBytes       : bstr,             ; Always null for signed QR presentations
  Handover              : SignedQRHandover
]

SignedQRHandover = [
  mdocGeneratedNonce    : tstr,             ; 16 random bytes, Base64URL-encoded
  validFrom             : uint,             ; Unix timestamp the QR code was generated at
  validTo               : uint              ; Unix timestamp the QR code is no longer valid at
]
```

Note that the `SessionTranscript` binds the presented `Document` to the temporal validity of the attestation (`validFrom` and `validTo`, i.e. `nbf` and `exp` in section 6.1) and random data generated by AltID (`mdocGeneratedNonce`, i.e. `mnonce` in section 6.1).

5.5.1 mdocGeneratedNonce

The `mdocGeneratedNonce` is a random value created by AltID for Signed QR flows to ensure that each presentation is unique and cannot be reused in later sessions. By including this nonce in the signed response, the mdoc provides proof that the data was generated specifically for the current transaction, thereby protecting against some forms of replay attacks.

5.6 SessionTranscript for OID4VP

For [OID4VP], the `SessionTranscript` takes a different form:

```
OpenID4VPHandover = [
  "OpenID4VPHandover" : tstr,      ; Fixed identifier for this handover type
  OpenID4VPHandoverInfoHash : bstr ; The SHA-256 hash of OpenID4VPHandoverInfoBytes
]

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

OpenID4VPHandoverInfo = [    ; Array containing handover parameters
  clientId : tstr,           ; The Verifier's client_id as defined in the Authorization Request
  nonce : tstr,              ; The nonce passed by the Verifier in Authorization Request
  jwkThumbprint : bstr,      ; JWK SHA-256 thumbprint for the public encryption key provided by
                             ; the Verifier in the client_metadata of the Authorization Request
                             ; null if none was provided
  responseUri : tstr         ; The Verifier's response URI from the Authorization Request
]
```

The `jwkThumbprint` parameter is mandatory in [HAIP] flows and must be null in [AVP] flows. The Verifier MUST compute the JWK SHA-256 thumbprint of the transaction's ephemeral encryption key according to [JWK].

Note, that the `DeviceAuthentication` is not included in the response. This means that Verifier must reconstruct the payload bytes (`DeviceAuthenticationBytes`) prior to validating the device signature. This process is illustrated in section 7.2.

5.7 Token Status Lists

AltID uses Token Status Lists in JWT format as defined in [TSL] to convey information about the status of attestations after issuance, specifically whether the attestation has been revoked. Note that the [TSL] specification is currently a draft and changes to the implementation should be expected once it is finalised.

A Token Status List in JWT format is a single JWT that provides status information for multiple attestations in a compact structure. The Issuer publishes and periodically updates a signed JWT that includes a JSON-encoded Status List. The Status List consists of a compressed byte array containing the status values and an indication of the number of bits used to represent the status of a single attestation.

Presentations of attestations may include a `status` element of the `IssuerSigned` MSO (see section 5.3), which contains a reference to the Status List Token (as an absolute URL) and the bit index corresponding to the status of the specific attestation. When `status` is present in a returned `document`, Verifiers MUST fetch and validate the Status List and check the relevant bit(s) to determine the validity of the attestation. Verifiers SHOULD implement appropriate precautions in case the Status List is unavailable, based on business needs and risk considerations.

The structure of a Status List in JWT format is shown below. Descriptions of the individual claims are provided in Table 8.

Header

```
{
  "typ": "statuslist+jwt",
  "alg": "ES256",
  "x5c": "[…]"
}
```

Body

```
{
  "sub": "https://dktb-issuer-cdn.test.myracdn.com/status-lists/...
  "exp": 1775128620,
  "iat": 1775042220,
  "ttl": 900,
  "status_list": {
    "lst": " eNo76fITAAPfAgc",
    "bits": 2
  }
}
```

Verifiers MUST validate the Status List JWT signature using the Issuer certificate from `x5c` claim and MUST ensure that the signing certificate is a trusted Issuer certificate.

Claim	Description
`typ`	JWT type. Always `statuslist+jwt` for AltID.
`alg`	Cryptographic algorithm used to sign the JWT. Always `ES256` for AltID.
`x5c`	Contains the Issuer's signing certificate.
`sub`	MUST be equal to the URI of the Status List Token.

Claim	Description
`exp`	Expiration time of the Status List Token, after which it MUST no longer be used for validating attestation status.
`iat`	Issuance time of the Status List Token. MUST not be in the future (allowing for clock skew).
`ttl`	The maximum amount of time (in seconds) the Status List Token can be cached, after which a new one SHOULD be fetched.
`status_list`	JSON-encoded Status List structure.
├── `lst`	A Base64url-encoded compressed byte array containing the status information. See 5.7.1.
└── `bits`	Specifies the number of bits used to describe a single status in the `lst`. For example, a value of 2 means that each attestation's status is represented by the combined value of two bits. This would allow for four different status types based on their value (0-3), and for a single byte to contain four statuses.

Table 8: Status List Token claims

AltID makes use of the following status values and corresponding status types:

Value	Status type
0x00 (0)	The attestation is valid.
0x01 (1)	The attestation is invalid, i.e. revoked, and MUST be rejected by the Verifier.
0x03 (3)	The attestation is valid, but an updated version is available. One or more attributes are no longer accurate. When the User is issued an updated attestation, the current attestation is revoked. Verifiers MAY reject attestations with this status based on their business requirements and risk tolerance. Verifiers that rely on attributes that are subject to change (e.g., `resident_address`) and require current values SHOULD reject attestations with this status. Note that this status is specific to AltID.

Table 9: Status types used by AltID

5.7.1 Validation of attestation status using Token Status List

Given an attestation with a `status` where `idx` is 6 and a TSL where `lst` is "eNo76fITAAPfAgc" and `bits` is 2:

1. Base64-url decode `lst` to get the compressed bytes: 78da3be9f2130003df0207
2. Decompress to get the byte array: `bytes = [0xC9, 0x44, 0xF9]`
3. Compute the position of the first relevant bit and the index of the byte containing it:
   `bit_pos = idx * bits = 6 * 2 = 12`
   `byte_index = bit_pos / 8 = 1`
4. Extract the status value from the byte using a shift and mask operation:
   `offset = bit_pos % 8 = 4`

```
value = (bytes[byte_index] >> offset) & ((1 << bits) - 1)
      = (0x44 >> 4) & 0b11
      = 0b00 = 0
```

5. The resulting `value` of 0 corresponds to the status type valid, i.e. the attestation has not been revoked.

The byte is shifted by the remainder (`offset`) because indexing is counted LSB-first, i.e. index 0 is the least significant (rightmost) bit. The shifting, combined with the masking, effectively results in the value of the byte being transformed to the status value. Thus, the status of the attestation can be validated directly against the value of the byte. See [TSL] for more information and additional examples.

6 Appendix B: Validation of Signed QR presentations

6.1 QRPayload

The byte array Q is a binary CBOR encoding of a CBOR map with the following syntax:

```
QRPayload = {
  "typ": tstr,          ; Type of QR code, will be "AltID-<version>", e.g. "AltID-1.0"
  "txn": tstr,          ; Transaction identifier, i.e. an identifier for the QRPayload
  "mnonce": tstr        ; mdocGeneratedNonce – random data generated by AltID
  "nbf": uint,          ; validFrom - Unix timestamp
  "exp": uint,          ; validTo - Unix timestamp
  "doc": bstr           ; CBOR encoding of Document
}
```

6.2 Steps for validating a Signed QR presentation

With Q in hand, parse it into the `QRPayload` CBOR-map, and validate the presented attestation according to the steps shown in Table 10 below:

Step	Description
1	Validate that the CBOR map contains six parts named `typ`, `txn`, `mnonce`, `nbf`, `exp`, and `doc`.
2	Validate that `typ` contains the value expected for an AltID Signed QR: "`AltID-1.0`".
3	Validate that `nbf` and `exp` are integers and construct the corresponding Unix UTC timestamps `validFrom` and `validTo`.
4	Validate that the current timestamp is equal to or later than `validFrom` (cf. 3), allowing for clock skew.
5	Validate that `validTo` (cf. 3) is equal to or later than the current timestamp, allowing for clock skew.
6	Validate that `mnonce` *(`mdocGeneratedNonce`)* is Base64URL encoding of 16 bytes.
7	Extract the `x5c` header in the unprotected header of the `MobileSecurityObject` (MSO), validate the certificate chain, and that the chain terminates in a root certificate registered in your Issuer trust list.
8	Validate the Issuer signature *(`IssuerAuth`)* using the public key from the Issuer certificate from step 7.
9	Calculate the digest value for every `IssuerSignedItem` in the `Document/IssuerNameSpaces` structure and verify that these calculated digests equal the corresponding digest values in the signed MSO (`DigestIDs`), see section 7.3 for more info and an example.
10	Verify that the `DocType` in the MSO and the `DocType` in the `Document` structure are equal.

Step	Description
11	Validate the elements in the `ValidityInfo` structure, i.e. verify: a. that the `signed` date is within the validity period of the certificate in the MSO header and b. that the `validFrom` element shall be earlier than or equal to the current timestamp, and c. that the `validUntil` element shall be equal to or later than the current timestamp
12	If the MSO contains a `status` element, validate the attestation status as described in section 5.7.
13	Validate the device signature (`DeviceAuth`) as described in section 7.2: a. Construct the `SessionTranscript`. b. Use `SessionTranscript` to construct `DeviceAuthentication`. c. Validate the `DeviceAuth` COSE_Sign1 signature using CBOR encoding of `DeviceAuthentication` as external payload and the public key from the MSO (`DeviceKeyInfo/DeviceKey`).
13	Validate, that `mdocGeneratedNonce` is not replayed: a. Maintain a set of all presented `mdocGeneratedNonce` values, that are valid according to `validTo` cf. 3. b. Verify that `mdocGeneratedNonce` is not in this set. c. Add `mdocGeneratedNonce` to the set.
14	Validate that the disclosed attribute(s) satisfy the requirements of the application.

Table 10: Steps for validating an attestation presented using Signed QR

7 Appendix C: Signed QR code example

The included example only discloses the attribute `age_over_18` from the PoA attestation for brevity. In practice, attestations can disclose multiple attributes. If multiple attributes are be disclosed, they will be included as additional `IssuerSignedItems`.

All examples use the issuer certificate and corresponding certificate chain described in Appendix G: Issuer Auth certificate chain and issuer signature validation.

When working with the examples, it is very convenient to use online tools such as [CBORZ] to parse binary data and examine the corresponding CBOR data structures.

7.1 Multiple QR codes

The four QR codes shown in Figure 5 contain the following binary data:

idx	$part_{idx}$ as Base64Url
0	pWN0eXBpQWx0SUQtMS4wY3R4bngkMzEyZjcxNzAtYjZjYy00NDQ2LTkyMjMtYWViZDc3ODI2MjE1Y2lkeABjY250BGRwYXJ0WQG2pmN0eXBpQWx0SUQtMS4wY3R4bngkMzEyZjcxNzAtYjZjYy00NDQ2LTkyMjMtYWViZDc3ODI2MjE1Zm1ub25jZXZvVDdDSE9rS2trY2ptMEVzRldEME9nY25iZhpp0TFqY2V4cBpp0THiY2RvY1kGaaNnZG9jVHlwZXFldS5ldXJvcGEuZWMuYXYuMWxpc3N1ZXJTaWduZWSiam5hbWVTcGFjZXOhcWV1LmV1cm9wYS5lYy5hdi4xgdgYWE-kaGRpZ2VzdElEBGZyYW5kb21QC1KaYFj6a9ngS35beyuH2nFlbGVtZW50SWRlbnRpZmllcmthZ2Vfb3Zlcl8xOGxlbGVtZW50VmFsdWX1amlzc3VlckF1dGiEQ6EBJqEYIVkCTjCCAkowggHwoAMCAQICFEJHyJBiXLqU6mv0sc1tAZeDbL3PMAoGCCqGSM49BAMCMG0xCzAJBgNVBAYTAkRLMRMwEQYDVQQHDApLw7hiZW5oYXZuMSEwHwYDVQQKDBhEaWdpdGFsaXNlcmluZ3NzdHlyZWxzZW4xDDAKBgNVBAsMA0tFQTEYMBYGA1UEAwwPREtUQiBJ
1	pWN0eXBpQWx0SUQtMS4wY3R4bngkMzEyZjcxNzAtYjZjYy00NDQ2LTkyMjMtYWViZDc3ODI2MjE1Y2lkeAFjY250BGRwYXJ0WQG2c3N1aW5nIENBMB4XDTI1MDYxODE0MjM1MVoXDTI2MDYxODE0MjM1MVowdDELMAkGA1UEBhMCREsxEzARBgNVBAcMCkvDuGJlbmhhdm4xITAfBgNVBAoMGERpZ2l0YWxpc2VyaW5nc3N0eXJlbHNlbjEMMAoGA1UECwwDS0VBMR8wHQYDVQQDDBZES1RCIENyZWRlbnRpYWwgSXNzdWVyMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEpHBVn6kQu5ZKnzSFrPLLfHXEX6ZxWlhfES5MUbsOMxPeFEz0zPOQLvJU278M-mBHcJF1ZUy294qYs7D2PacIH6NnMGUwDgYDVR0PAQH_BAQDAgeAMB8GA1UdIwQYMBaAFKIpVen1SlQ2mm7eWxDXsBDglv3iMB0GA1UdDgQWBBQwUoAEIiXsQfN2bcL3c_uKFUnytDATBgNVHSUEDDAKBggrBgEFBQcDAzAKBggqhkjOPQQDAgNIADBFAiEAz6cGyitgDHQYBfn87kW2Ww4sJyJagTP2iElqYc7QyawCIBUqQWHPNFRx_pqFYJXkVQxeozlVWvTlOP_MnD3n
2	pWN0eXBpQWx0SUQtMS4wY3R4bngkMzEyZjcxNzAtYjZjYy00NDQ2LTkyMjMtYWViZDc3ODI2MjE1Y2lkeAJjY250BGRwYXJ0WQG2U5KwWQKj2BhZAp6mZ3ZlcnNpb25jMS4wb2RpZ2VzdEFsZ29yaXRobWdTSEEtMjU2bHZhbHVlRGlnZXN0c6FxZXUuZXVyb3BhLmVjLmF2LjGpAVggtcBRNo9LzEoKU9qBJxuhEVsDEWKA-fMwsx93n74SHzQCWCCgZ_DeQ5RTRxHLRpmK1sDiBeiW3Q-GAv6Or_Kv1g7tpQNYIHOHRQe6KsXplpla4O_yUxIIx1hUBEsJrcJ0YyF5krm1BFgg8Yo5VlXMwEj3TLC_SDA5JHL3W6VszYEuZqGhXyj6paUFWCB2B_RYml-_NQPjaPqwXM0HajPQZgFevdcN1YGgUCAdKwZYIMJPHkVhf9EQpjaQytctdgfqqYKFDFTLIguvJiuJXl3fB1ggL0V537rvxKIugQqTNzlJ5RhBTG9ywgg-A9qXYkVtUQEIWCDZIGgUpA4llNcKUcB65LCEH-eD_73bwM01t_8aVnrI9AlYIErtcDCNa6Qe6h3IznhBDo5-FFetaQEEuQ8WT86U5MZYbWRldmljZUtleUluZm-iaWRldmljZUtleaUBAiABIVggEW4ccBJWWL9z

idx	$part_{idx}$ as Base64Url
3	pWN0eXBpQWx0SUQtMS4wY3R4bngkMzEyZjcxNzAtYjZjYy00NDQ2LTkyMjMtYWViZDc3ODI2MjE1Y2lkeANjY250BGRwYXJ0WQG33x7_-sPGkfBpCi8pgR4SS8fkcQjbBNUiWCBBc8SnDVlBoxTV8B7t_d1cuzBB61lZGxIPJK1fT5aO2wMmcWtleUF1dGhvcml6YXRpb25zoWpuYW1lU3BhY2VzgXFldS5ldXJvcGEuZWMuYXYuMWdkb2NUeXBlcWV1LmV1cm9wYS5lYy5hdi4xbHZhbGlkaXR5SW5mb6Nmc2lnbmVkwHQyMDI2LTA0LTA0VDAwOjAwOjAwWml2YWxpZEZyb23AdDIwMjYtMDQtMDRUMDA6MDA6MDBaanZhbGlkVW50aWzAdDIwMjYtMDUtMDRUMDA6MDA6MDBaWECXDhVgCMTpXCNkMDRMJ9wAofDN497xyFqEbFW_54GmKlBSIxYLJ-xTzN9fErXNsGQa1gdTSWoYxKkfv71CRQZ5bGRldmljZVNpZ25lZKJqbmFtZVNwYWNlc9gYQaBqZGV2aWNlQXV0aKFvZGV2aWNlU2lnbmF0dXJlhEOhASag9lhA-CMfj2RQusYPy04vhjZaPwgP6RXY_0i4_yGmEhd4kO9M7hmMxrXkDugiSG0Tb3AunA9oQi554v-ExlOBWdMWEw

For `idx` = 1 (the second part), the data represents this CBOR data structure:

```
{
  "typ": "AltID-1.0",
  "txn": "312f7170-b6cc-4446-9223-aebd77826215",
  "idx": 1,
  "cnt": 4,
  "part":
h'737375696e67204341301e170d323530363138313432333531 5a170d323630363138313432333531 5a3074310b300
906035504061302444b311330110603550407 0c0a4bc3b862656e6861766e3121301f060355040a0c18446967697461
6c69736572696e677373747972656c73656e310c300a060355040b0c034b4541311f301d06035504030c16444b54422
043726564656e7469616c2049737375657230593013060 72a8648ce3d020106082a8648ce3d030107034200 04a47055
9fa910bb964a9f3485acf2cb7c75c45fa6715a585f112e4c51bb0e3313de144cf4ccf3902ef254dbbf0cfa604770917
5654cb6f78a98b3b0f63da7081fa3673065300e0603551d0f0101ff040403020780301f0603551d23041830168014a2
2955e9f54a54369a6ede5b10d7b010e096fde2301d0603551d0e04160414305280042225ec41f3766dc2f773fb8a154
9f2b430130603551d25040c300a06082b06010505070303300a06082a8648ce3d040302034800304502210 0cfa706ca
2b600c741805f9fcee45b65b0e2c27225a8133f688496a61ced0c9ac0220152a4161cf345471fe9a856095e4550c5ea
339555af4e538ffcc9c3de7'
}
```

The four binary strings (`part` for `idx` 0, 1, 2, and 3) are concatenated into a single binary string (`Q`), which represents this CBOR data structure:

```
{
  "typ": "AltID-1.0",
  "txn": "312f7170-b6cc-4446-9223-aebd77826215",
  "mnonce": "oT7CHOkKkkcjm0EsFWD0Og",
  "nbf": 1775317354,
  "exp": 1775317474,
  "doc":
h'a367646f635479706571657 65752e6575726f70612e65632e61762e316c6973737565725369676e6564a26a6e616d655
37061636573a17165752e6575726f70612e65632e61762e3181d818584fa4686469676573744944046672616e646f6d
500b529a6058fa6bd9e04b7e5b7b2b87da71656c656d656e744964656e7469666965726b6167655f6f7665725f31386
c656c656d656e7456616c7565f56a697373756572417574688443a10126a1182159024e3082024a308201f0a0030201
```

```
0202144247c890625cba94ea6bf4b1cd6d0197836cbdcf300a06082a8648ce3d040302306d310b30090603550406130
2444b311330110603550407 0c0a4bc3b862656e6861766e3121301f060355040a0c184469676974616c69736572696e
677373747972656c73656e310c300a060355040b0c034b4541311830160603550403 0c0f444b54422049737375696e6
72043413 01e170d323530363138313432333531 5a170d323630363138313432333531 5a3074310b3009060355040613
02444b311330110603550407 0c0a4bc3b862656e6861766e3121301f060355040a0c184469676974616c69736572696
e677373747972656c73656e310c300a060355040b0c034b4541311f301d06035504030c16444b54422043726564656e
7469616c20497373756572305930130607 2a8648ce3d020106082a8648ce3d03010703420004a470559fa910bb964a9
f3485acf2cb7c75c45fa6715a585f112e4c51bb0e3313de144cf4ccf3902ef254dbbf0cfa6047709175654cb6f78a98
b3b0f63da7081fa3673065300e0603551d0f0101ff040403020780301f0603551d23041830168014a22955e9f54a543
69a6ede5b10d7b010e096fde2301d0603551d0e04160414305280042225ec41f3766dc2f773fb8a1549f2b430130603
551d25040c300a06082b060105050703033 00a06082a8648ce3d0403020348003045022100cfa706ca2b600c741805f
9fcee45b65b0e2c27225a8133f688496a61ced0c9ac0220152a4161cf345471fe9a856095e4550c5ea339555af4e538
ffcc9c3de75392b05902a3d81859029ea66776657273696f6e63312e306f646967657374416c676f726974686d67534
8412d3235366c76616c756544696765737473a17165752e6575726f70612e65632e61762e31a9015820b5c051368f4b
cc4a0a53da81271ba1115b03116280f9f330b31f779fbe121f34025820a067f0de4394534711cb46998ad6c0e205e89
6dd0f8602fe8eaff2afd60eeda503582073874507ba2ac5e996995ae0eff2531208c75854044b09adc27463217992b9
b5045820f18a395655ccc048f74cb0bf4830392472f75ba56ccd812e66a1a15f28faa5a50558207607f4589a5fbf350
3e368fab05ccd076a33d066015ebdd70dd581a050201d2b065820c24f1e45617fd110a63690cad72d7607eaa982850c
54cb220baf262b895e5ddf0758202f4579dfbaefc4a22e810a93373949e518414c6f72c2083e03da9762456d5101085
820d9206814a40e2594d70a51c07ae4b0841fe783ffbddbc0cd35b7ff1a567ac8f40958204aed70308d6ba41eea1dc8
ce78410e8e7e1457ad690104b90f164fce94e4c6586d6465766963654b6579496e666fa2696465766963654b6579a50
1022001215820116e1c70125658bf73df1efffac3c691f0690a2f29811e124bc7e47108db04d52258204173c4a70d59
41a314d5f01eedfddd5cbb3041eb59591b120f24ad5f4f968edb0326716b6579417574686f72697a6174696f6e73a16
a6e616d6553706163657 3817165752e6575726f70612e65632e61762e3167646f63547970657165752e6575726f7061
2e65632e61762e316c76616c6964697479496e666fa3667369676e6564c074323032362d30342d30345430303a30303
a30305a6976616c696446726f6dc074323032362d30342d30345430303a30303a30305a6a76616c6964556e74696cc0
74323032362d30352d30345430303a30303a30305a5840970e156008c4e95c236430344c27dc00a1f0cde3def1c85a8
46c55bfe781a62a505223160b27ec53ccdf5f12b5cdb0641ad60753496a18c4a91fbfbd424506796c64657669636553
69676e6564a26a6e616d65537061636573d81841a06a6465766963654 1757468a16f6465766963655369676e6174757
2658443a10126a0f65840f8231f8f6450bac60fcb4e2f86365a3f080fe915d8ff48b8ff21a612177890ef4cee198cc6
b5e40ee822486d136f702e9c0f68422e79e2ff84c6538159d31613'
}
```

The value doc represents this CBOR Document:

```
{
  "docType": "eu.europa.ec.av.1",
  "issuerSigned": {
    "nameSpaces": {
      "eu.europa.ec.av.1": [
        24(<<
          {
            "digestID": 4,
            "random": h'0b529a6058fa6bd9e04b7e5b7b2b87da',
            "elementIdentifier": "age_over_18",
            "elementValue": true
          }
        >>)
```

```
      ]
    },
    "issuerAuth": [// COSE signature
      h'a10126',   // COSE signature protected header:
                   // CBOR encoding of {1: -7}, i.e. alg_id = ECDSA with SHA-256 (ES256)
      {            // COSE signature unprotected header:
        33:        // COSE header 33 is X.509 certificate chain,
                   // the chain contains only the leaf certificate described in section 11.
            [ h'3082024a308201f0...']
      },
   // The object below is the signature payload, i.e. the Mobile Security Object
h'd81859029ea66776657273696f6e63312e306f646967657374416c676f726974686d675348412d3235366c76616c7
565446967657374473a17165752e6575726f70612e65632e61762e31a9015820b5c051368f4bcc4a0a53da81271ba111
5b03116280f9f330b31f779fbe121f34025820a067f0de4394534711cb46998ad6c0e205e896dd0f8602fe8eaff2afd
60eeda503582073874507ba2ac5e996995ae0eff2531208c75854044b09adc27463217992b9b5045820f18a395655cc
c048f74cb0bf4830392472f75ba56ccd812e66a1a15f28faa5a5055207607f4589a5fbf3503e368fab05ccd076a33d
066015ebdd70dd581a050201d2b065820c24f1e45617fd110a63690cad72d7607eaa982850c54cb220baf262b895e5d
df0758202f4579dfbaefc4a22e810a93373949e518414c6f72c2083e03da9762456d5101085820d9206814a40e2594d
70a51c07ae4b0841fe783ffbddbc0cd35b7ff1a567ac8f40958204aed70308d6ba41eea1dc8ce78410e8e7e1457ad69
0104b90f164fce94e4c6586d6465766963654b6579496e666fa2696465766963654b6579a501022001215820116e1c7
0125658bf73df1efffac3c691f0690a2f29811e124bc7e47108db04d52258204173c4a70d5941a314d5f01eedfddd5c
bb3041eb59591b120f24ad5f4f968edb0326716b6579417574686f72697a6174696f6e73a16a6e616d6553706163657
3817165752e6575726f70612e65632e61762e3167646f63547970657165752e6575726f70612e65632e61762e316c76
616c6964697479496e666fa3667369676e6564c074323032362d30342d30345430303a30303a30305a6976616c69644
6726f6dc074323032362d30342d30345430303a30303a30305a6a76616c6964556e74696cc074323032362d30352d30
345430303a30303a30305a',
     // And finally, the COSE binary signature value, formed using the issuer private key:
h'970e156008c4e95c236430344c27dc00a1f0cde3def1c85a846c55bfe781a62a505223160b27ec53ccdf5f12b5cdb
0641ad60753496a18c4a91fbfbd42450679'
    ]
  },
  "deviceSigned": {
    "nameSpaces": 24(<<{}>>), // empty map: all disclosed attributes are incl. in issuerSigned
    "deviceAuth": {
      "deviceSignature": (/ COSE_Sign1 / [
        / protected / <<{ 1: -7 }>>,   // alg_id = ECDSA with SHA-256 (ES256)
        / unprotected / { },           // no header elements in unprotected header
        null,                          // Payload not included – 'detached' signature
h'f8231f8f6450bac60fcb4e2f86365a3f080fe915d8ff48b8ff21a612177890ef4cee198cc6b5e40ee822486d136f7
02e9c0f68422e79e2ff84c6538159d31613' // ECDSA Signature, P-256, 64 bytes
      ])
    }
  }
}
```

The included Mobile Security Object (h'd818...') is:

```
24(<<
  {
    "version": "1.0",
    "digestAlgorithm": "SHA-256",
    "valueDigests": {
      "eu.europa.ec.av.1": {
        1: h'b5c051368f4bcc4a0a53da81271ba1115b03116280f9f330b31f779fbe121f34',
        2: h'a067f0de4394534711cb46998ad6c0e205e896dd0f8602fe8eaff2afd60eeda5',
        3: h'73874507ba2ac5e996995ae0eff2531208c75854044b09adc27463217992b9b5',
        4: h'f18a395655ccc048f74cb0bf4830392472f75ba56ccd812e66a1a15f28faa5a5',
        5: h'7607f4589a5fbf3503e368fab05ccd076a33d066015ebdd70dd581a050201d2b',
        6: h'c24f1e45617fd110a63690cad72d7607eaa982850c54cb220baf262b895e5ddf',
        7: h'2f4579dfbaefc4a22e810a93373949e518414c6f72c2083e03da9762456d5101',
        8: h'd9206814a40e2594d70a51c07ae4b0841fe783ffbddbc0cd35b7ff1a567ac8f4',
        9: h'4aed70308d6ba41eea1dc8ce78410e8e7e1457ad690104b90f164fce94e4c658'
      }
    },
    "deviceKeyInfo": {
      "deviceKey": {
        1: 2,
        -1: 1,
        -2: h'116e1c70125658bf73df1efffac3c691f0690a2f29811e124bc7e47108db04d5',
        -3: h'4173c4a70d5941a314d5f01eedfddd5cbb3041eb59591b120f24ad5f4f968edb',
        3: -7
      },
      "keyAuthorizations": {
        "nameSpaces": [
          "eu.europa.ec.av.1"
        ]
      }
    },
    "docType": "eu.europa.ec.av.1",
    "validityInfo": {
      "signed": 0("2026-04-04T00:00:00Z"),
      "validFrom": 0("2026-04-04T00:00:00Z"),
      "validUntil": 0("2026-05-04T00:00:00Z")
    }
  }
>>)
```

Here the nine digests represent nine different `age_over_NN` attributes, that can be disclosed for this PoA. As can be seen from `IssuerSigned`, digest with id = 4 refers to the `age_over_18` attribute, which is the only `age_over_NN` attribute disclosed in this QR code.

The COSE signature in `IssuerSigned` was created when the PoA was issued by the Issuer service (DIGST). The signature can be verified with the public key from the PoA Issuer's certificate (see Appendix G: Issuer Auth

certificate chain and issuer signature validation under section 11). The validity of this signature proves, that no data in Mobile Security Object was changed or tampered with.

7.2 Device signature validation

The device signature can be verified with the public key included in the Mobile Security Object of the `IssuerSigned` object (as `DeviceKey`). This key is bound to the device on which the AltID app is installed.

To verify the device signature, as a Verifier we first need to construct the `SessionTranscript` from `mnonce`, `nbf`, and `exp` from `QRPayload` and, in turn, `DeviceAuthenticationBytes`, the payload for the device signature.

Remembering the `SessionTranscript` form for Signed QR:

```
SessionTranscript = [
  DeviceEngagementBytes : bstr,             ; Always null for Signed QR presentations
  EReaderKeyBytes       : bstr,             ; Always null for Signed QR presentations
  Handover              : SignedQRHandover
]

SignedQRHandover = [
  mdocGeneratedNonce    : tstr,             ; 16 random bytes, Base64URL-encoded
  validFrom             : uint,             ; Unix timestamp the QR code was generated at
  validTo               : uint              ; Unix timestamp the QR code is no longer valid at
]
```

Using the values for `mnonce`, `nbf`, and `exp` (as provided in section 7.1) for `mdocGeneratedNonce`, `validFrom`, and `validTo`, respectively, the handover becomes:

```
SignedQRHandover = [
  "oT7CHOkKkkcjm0EsFWD0Og",
  1775317354,
  1775317474
]
```

Using this, gives the following transcript:

```
SessionTranscript = [
  null,
  null,
  ["oT7CHOkKkkcjm0EsFWD0Og", 1775317354, 1775317474]
]
```

CBOR encoding this array, and then Base64Url-encoding the output gives the following value:

```
g_b2g3ZvVDdDSE9rS2trY2ptMEVzRldEME9nGmnRMWoaadEx4g
```

This can be verified at [CBORZ].

Using the `SessionTranscript` constructed above and remembering `DeviceAuthentication`:

```
DeviceAuthentication = [
   "DeviceAuthentication",
   SessionTranscript,
   DocType,                ; Same as Document DocType, e.g. "eu.europa.ec.av.1"
   DeviceNameSpacesBytes   ; Always tagged bstr of empty map: #6.24(bstr .cbor {} )
```

```
]
```

We can now construct the CBOR representation of `DeviceAuthentication` as:

```
DeviceAuthentication = [
   "DeviceAuthentication",
   [ null, null, ["oT7CHOkKkkcjm0EsFWD0Og", 1775317354, 1775317474]],
   "eu.europa.ec.av.1",
   #6.24(bstr .cbor {} )
]
```

Remember the meaning of #6.24(bstr .cbor {}): A tag (CBOR major type 6) with number 24 (meaning bstr) over CBOR encoding of an empty map.

We can now construct `DeviceAuthenticationBytes` as #6.24(bstr .cbor `DeviceAuthentication`), giving the following Base64-url encoded value:

```
2BhYUYR0RGV2aWNlQXV0aGVudGljYXRpb26D9vaDdm9UN0NIT2tLa2tjam0wRXNGV0QwT2caadExahpp0THicWV1LmV1cm9
wYS5lYy5hdi4x2BhBoA
```

Testing with [CBORZ] (by converting from "base64url" to "diagnostic") gives:

```
24(<<
  [
    "DeviceAuthentication",
    [
      null,
      null,
      [
        "oT7CHOkKkkcjm0EsFWD0Og",
        1775317354,
        1775317474
      ]
    ],
    "eu.europa.ec.av.1",
    24(<<
      {
      }
    >>)
  ]
>>)
```

The device COSE signature can now be verified using the public key from MSO `DeviceKeyInfo` and the constructed `DeviceAuthenticationBytes` as payload.

7.3 Attribute validation

In practice, validate disclosed mdoc attributes, e.g. the `age_over_18` attribute as in the example above, by doing the following for each disclosed attribute, i.e. `IssuerSignedItemBytes` in the `IssuerNameSpaces` map:

1. Use the `digestID` as index to lookup the corresponding signed digest value in the `ValueDigests` map of the Mobile Security Object.
2. Compute the digest over `IssuerSignedItemBytes` using the hash algorithm given by `digestAlgorithm`, which is always SHA-256 for AltID.
3. Compare the digest from the `ValueDigests` map with the computed digest (byte-array comparison).

4. If the two digests are not equal, reject the attestation.

It is recommended to compute digest directly over `IssuerSignedItemBytes` taken from the attestation, and not to parse the data, and attempt to reconstruct `IssuerSignedItem` from its components. If the latter approach is used, make sure to remember adding the tag, to use the correct ordering of map elements, and not to use a CBOR indefinite-length map before computing `IssuerSignedItemBytes`.

In the example, only `age_over_18` is disclosed, with the value true. This is represented by the following part (`IssuerSignedItem`) of the CBOR Document:

```
24(<<
  {
    "digestID": 4,
    "random": h'0b529a6058fa6bd9e04b7e5b7b2b87da',
    "elementIdentifier": "age_over_18",
    "elementValue": true
  }
>>)
```

The corresponding `IssuerSignedItemBytes`, which can be obtained by Base64Url-encoding the CBOR encoding of syntax above, are:

```
2BhYT6RoZGlnZXN0SUQEZnJhbmRvbVALUppgWPpr2eBLflt7K4facWVsZW1lbnRJZGVudGlmaWVya2FnZV9vdmVyXzE4bGVsZW1lbnRWYWx1ZfU
```

Computing the digest, by calculating the SHA-256 hash over the Base64Url-encoded string, yields the following hex value:

```
f18a395655ccc048f74cb0bf4830392472f75ba56ccd812e66a1a15f28faa5a5
```

We see that this value matches that of the fourth (as denoted by the `digestID`) entry in the `ValueDigests` map of the MSO, thus confirming the validity of the `age_over_18` attribute.

8 Appendix D: Validation of OID4VP Authorization Responses

Successful OID4VP Authorization Responses contain the `vp_token` parameter. Depending on the response mode, it is included either directly in the body of the HTTP POST request (when using `direct_post`) or as a claim in an encrypted JWT (when using `direct_post.jwt`). In either case, the `vp_token` MUST be validated as follows:

Step	Description
1	Validate that the HTTP POST message contains a `vp_token` parameter or a `response` parameter containing an encrypted JWT (which has the `vp_token`), which MUST be decrypted using the key provided in the Authorization Request.
2	Construct `DeviceResponse` (see 9.2 and 10.2 for examples) by Base64Url decoding `vp_token` and CBOR parsing the resulting byte stream.
3	Validate that `DeviceResponse` parses into CBOR map with members `version`, `documents`, and `status`.
4	Validate that `status` equals 0 (OK) and that `documents` array contains exactly one `Document`.
5	Extract the x5chain from `Document/IssuerSigned/IssuerAuth` unprotected header and validate the certificate chain (see section 11). Validate that the last certificate in the chain is a self-signed certificate present in the issuer trust list. Validate that the chain is ordered, and that the issuer certificate is first certificate in the chain.
6	Validate the issuer signature (`IssuerAuth`) using the public key from the issuer certificate from step 5.
7	Calculate the digest value for every `IssuerSignedItem` in the `Document/IssuerNameSpaces` structure and verify that these calculated digests equal the corresponding digest values in the signed MSO (`DigestIDs`), see section 7.3 for more info and an example.
8	Validate the issuer signature (`IssuerAuth`) using the public key from the issuer certificate from step 7.
9	Verify that the `DocType` in the MSO and `DocType` in the `Document` structure are equal, e.g. "eu.europa.ec.av.1" for PoA.
10	Validate the elements in the `ValidityInfo` structure, i.e. verify: a. that the `signed` date is within the validity period of the certificate in the MSO header and b. that the `validFrom` element shall be earlier than or equal to the current timestamp, and c. that the `validUntil` element shall be equal to or later than the current timestamp
11	If the MSO contains a `status` element, validate the attestation status as described in section 5.7.

Step	Description
12	Validate the device signature (`DeviceAuth`) as follows: a. Construct the `SessionTranscript` as described in (section 5.6). b. Use `SessionTranscript` to construct `DeviceAuthentication` as described in section 5.4. c. Validate the `DeviceAuth` COSE_Sign1 signature using CBOR encoding of `DeviceAuthentication` as external payload and public key from MSO (`DeviceKeyInfo/DeviceKey`).
13	Validate that `nonce` is not replayed, i.e. that the internal transaction associated with the `nonce` is not already marked as completed, then mark it as completed.
14	Validate that the disclosed attribute(s) satisfy the requirements of the application.

Table 11: Steps for validating an OID4VP Authorization Response

If the Verifier has successfully validated the Authorization Response, it MUST respond with an HTTP 200 status code with Content-Type of `application/json` and an empty JSON object in the response body. If the same-device flow is used, the `redirect_uri` and `browser` claims MUST be included in the JSON object as described in section 4.3.1.1.

If any errors are encountered when performing the validation, except for step 14, the Verifier MUST respond with an HTTP 400 status code and provide and error response including a user-friendly error description.

8.1 Error Responses

Verifiers MUST be able to handle error responses as defined in section 8.5 of [OID4VP]. If the user rejects to share the credential or if AltID experiences other errors when handling the request, AltID will post an (unencrypted) error response to `response_uri`. An example:

```
POST /wallet/direct_post/K7GtmcQt4CnEm[...] HTTP/1.1
Host: verifier-backend.ageverification.dev
Content-Type: application/x-www-form-urlencoded

error=invalid_request
&error_description="Failed to fetch request object from request_uri"
&state=K7GtmcQt4CnEmCyvEQe5aOiOXH5O6VJIyGudl4pMQb1xbJNrUcNeMPgd6svMrxTDyKCpKC6Wz7tZh22YxvJRDg
```

In some cases, including an `error_description` may leak information about the User. For example, a description stating that the user has rejected the request reveals that the user in fact satisfies the query. To prevent this, AltID MAY omit the `error_description`. Error code (`error`) and `state` will always be included in error responses.

9 Appendix E: OID4VP example for [HAIP]

This section will give a detailed example of presenting PID over the OID4VP protocol according to [HAIP].

The following section is based on and aligned with the OpenID4VC High Assurance Interoperability Profile 1.0. This includes specific requirements for the implementation of the OID4VP protocol. [HAIP] is intended for use cases demanding strong security and adds security elements compared to [AVP]. The most prominent of these are:

- Verifiers (Relying Parties) MUST authenticate requests to AltID: Verifiers MUST register an OCES certificate to use for this authentication at [RPR] and use this certificate to sign their Authorization Requests.
- Responses MUST be encrypted using ephemeral (one-time) encryption keys supplied by the Verifier.
- Verifiers MUST support same-device flows using `redirect_uri` to enable phishing resistance by session binding.

This section will clarify these elements by providing detailed examples.

9.1 Registered URLs for AltID

AltID registers the following iOS Universal Links/Android App Links:

Environment	Registered link
Production	https://app.tegnebog.dk/
Test	https://app.test.tegnebog.dk/

All URLs under these schemes and domains are handled by AltID. [HAIP] flows MUST use these links for communication with AltID, specifically links of the form <registered_link>/oid4vp are treated as OID4VP Authorization Requests by AltID.

9.2 Authorization Request

The example shown here passes the Authorization Request by reference and implements the Pushed Authorization Request [PAR] model enforced by [HAIP]. The Authorization Request below should be presented to the User either as a (navigable) link or rendered as a QR code that can be scanned by the User's device. The example has been indented and URL decoded for readability.

```
https://app.test.tegnebog.dk/oid4vp?
    client_id=x509_hash:EPD6X85fbUtA5uMgM4XyVU3FHJ0rTfQVW_vUEGwtETY&
    request_uri=https://test-tool.test.tegnebog.dk/verifier/wallet/request.jwt/2GgzSSRf8m8N4e_
12eDVqb1E1ppgyJGUvpxF0QGzo0c
```

The provided `client_id` identifies the Verifier by a SHA-256 hash value of the certificate the Verifier has registered at [RPR]. The `request_uri` parameter identifies the location of the Authorization Request (Request Object). The Request Object is signed by the Verifier according to [JAR], and its contents are shown below. A description of the included claims can be found in section 4.2.

Header


```
{
    "x5c": [
"MIIGfTCCBLGgAwIBAgIUJQXSOpd7jUH+KFu/UpjmIEIds84wQQYJKoZIhvcNAQEKMDSgDzANBglghkgBZQMEAgEFAKEcMB
oGCSqGSIb3DQEBCDANBglghkgBZQMEAgEFAKIDAgEgMGsxLTArBgNVBAMMJERlbiBEYW5za2UgU3RhdCBPQ0VTIHVkc3RlZ
GVuZGUtQ0EgMTETMBEGA1UECwwKVGVzdCAtIGN0aTEYMBYGA1UECgwPRGVuIERhbnNrZSBTdGF0MQswCQYDVQQGEwJESzAe
Fw0yNjAxMDkwOTQ3MjdaFw0yOTAxMDgwOTQ3MjZaMIGSMR8wHQYDVQQDDBZES1RCIFRlc3RUb29sIFZlcmlmaWVyMTcwNQY
DVQQFEy5VSTpESy1POkc6MGFhNTA4NjItMmVhNC00NzExLWFiN2ItZGIzNWQzZWRhODAxMRAwDgYDVQQKDAdpZDIgQS9TMR
cwFQYDVQRhDA5OVFJESy00NDE2MDMxNTELMAkGA1UEBhMCREswggGiMA0GCSqGSIb3DQEBAQUAA4IBjwAwggGKAoIBgQCvR
kEUYgCci59cNLvfhLbNqROYx/Chi9MgVxQev7wXBfPymGPEdJ4C8SdvkymR8taScCvTrE+AhOPY8LSIP6pWpnNGRSeKGlkI
xLREr+ZNUi2e0LQ7HBie2NNe4/NTEdZaCHACc8adT164jinqBAmd1xF75OoaKOjbX9Cy1GJe3jfl9UOWen6WuhJp5dsLjyp
RG5Z9KuRLlkyJfuCcBbIdtBcpsfubueJvh7VSc+cJyquaQd5XzIQOOofb6jn+sUpQl0fpy8rwIfdXg/lIewJQXzeF88ECF6
StYKYrmF2hmOUtto5A4LvakyYPXxtnhPyMyWSIJ4XpyPAHb/8cevv+r9gU5u6+NLSurV/sMeygciN7DHLNs99yNhv6G88Ru
Im0pYrJOqqaHGZEEFeV/OpA6Ot2PF74QkmlSlfoJdyfmpWF3ym9h8zEFt/HBg2Anf/7hrgrTPQN21qmsiTIU9pJ6aI21PNy
xrKpTUDRPbeKO0CYxJhP/QCp4Veeq6ZvniUCAwEAAaOCAYcwggGDMAwGA1UdEwEB/wQCMAAwHwYDVR0jBBgwFoAUfyif2XG
ZQuJ159c1di5NCCVtdl4wewYIKwYBBQUHAQEEbzBtMEMGCCsGAQUFBzAChjdodHRwOi8vY2ExLmN0aS1nb3YuZGsvb2Nlcy
9pc3N1aW5nLzEvY2FjZXJ0L2lzc3VpbmcuY2VyMCYGCCsGAQUFBzABhhpodHRwOi8vY2ExLmN0aS1nb3YuZGsvb2NzcDAiB
gNVHSAEGzAZMAgGBgQAj3oBATANBgsqgVCBKQEBAQMHATA7BggrBgEFBQcBAwQvMC0wKwYIKwYBBQUHCwIwHwYHBACL7EkB
AjAUhhJodHRwczovL3VpZC5nb3YuZGswRQYDVR0fBD4wPDA6oDigNoY0aHR0cDovL2NhMS5jdGktZ292LmRrL29jZXMvaXN
zdWluZy8xL2NybC9pc3N1aW5nLmNybDAdBgNVHQ4EFgQUJyWN4HAtaS6vXq4zbWeCm3hy5rUwDgYDVR0PAQH/BAQDAgWgME
EGCSqGSIb3DQEBCjA0oA8wDQYJYIZIAWUDBAIBBQChHDAaBgkqhkiG9w0BAQgwDQYJYIZIAWUDBAIBBQCiAwIBIAOCAYEAA
LomYKh1DmtlHk20FJvoGBXaeeQ/Qrcly0OTglVu/GaQz5pUJcq4DUegiyFH/KvIwok42bHkRSgLrnu23oo9ed4kCzY1ytS4
A1p3Q4OPArZ03Jn5lARa0Fu+Gsveo/jzYfIaKs6esodOh4SYz2OqimeFkgqY6LVOXsLe+sLWDQ0619g4OnB7iUkTRJbhG/U
ICdchkYUv6vcPyeku7ISEyepLFgfqJHd3oqtv2/9XV6yQJ49M1IElimPoxCA7okhdN5Hcs2zRVJlEEgviy4ZceNCVcUCRwL
5JkcbbUPKq1fDC2siVAuHeHUp21kA1Q+auilzh1Cp12qRzGgOVJzYu01GnY7V1Ycq8EJgjMs7muGWgfsNT0U1FtTkZaeUvx
4l86XEbuCS2KipWS1Hx/uD3CYu+4p/7nhnHFEWFSX+CJJvSObAKRrmOC3Zw2XmSoWSUBBXQD/PavkgGO7N9lK2yt/PeGDvf
epGJyPg8pPGwiY6etJHe+sXdBufTTegI+kK2",
"MIIGYTCCBBWgAwIBAgIUc01wpi5SHNF8ZEyZoJAOHVngYa4wQQYJKoZIhvcNAQEKMDSgDzANBglghkgBZQMEAgMFAKEcMB
oGCSqGSIb3DQEBCDANBglghkgBZQMEAgMFAKIDAgFAMGIxJDAiBgNVBAMMG0RlbiBEYW5za2UgU3RhdCBPQ0VTIHJvZC1DQ
TETMBEGA1UECwwKVGVzdCAtIGN0aTEYMBYGA1UECgwPRGVuIERhbnNrZSBTdGF0MQswCQYDVQQGEwJESzAeFw0yMTAzMTIw
NzQyNTlaFw0zMTAzMTAwNzQyNThaMGsxLTArBgNVBAMMJERlbiBEYW5za2UgU3RhdCBPQ0VTIHVkc3RlZGVuZGUtQ0EgMTE
TMBEGA1UECwwKVGVzdCAtIGN0aTEYMBYGA1UECgwPRGVuIERhbnNrZSBTdGF0MQswCQYDVQQGEwJESzCCAaIwDQYJKoZIhv
cNAQEBBQADggGPADCCAYoCggGBANOLofW5cEL4GKGtDfVb75BI+UXI5nl/gnoi1CJcsirR+07jFP4x7mSZe/Olr0z1M3boT
VaLCtdZurqR6AysVI9z9ymzE8zCLxE5U1x3ay/x/0J0iTxpqHqDhSF0OH+Ymb1vgp+9ta6kPweSdG+WjJL+5xkeynGP2Zd8
0O2RusLAjx7s7b8sR73ImevflENJXNAJwL6MQrx4wiTZYdPOVOSyWKTMHnBJ0MTinPyFxIv8zfbpJ2w8I+/YNTzI9EsXDDd
BpQ4+asKxA4qZY/1+OQbVTGgMTNQKl7GireR2R+cTobGiD4zzd/7mt12/gQDkjOFtSVCuOfCU82EKilEysrnKhqFgtyGzUg
RQ0PFePdI8DS50popRVtNlKKo17eE3P2GlFr9SIm9U48GN10Eb1q+X+K1OjmNJBUVYVHF3fxC2iPySLbwkS2DSj/A6fBznj
/FkIGL1BzcG+DQPimsqKDZoIBm5SBOmi4ujoTxi+4w7h6uLyUt3D3vsPL1yUkl+qQIDAQABo4IBHDCCARgwEgYDVR0TAQH/
BAgwBgEB/wIBADAfBgNVHSMEGDAWgBQ53N7e0JAmR6Dgxm5JfybyqS/2GzBzBggrBgEFBQcBAQRnMGUwOwYIKwYBBQUHMAK
GL2h0dHA6Ly9jYTEuY3RpLWdvdi5kay9vY2VzL3Jvb3QvY2FjZXJ0L3Jvb3QuY2VyMCYGCCsGAQUFBzABhhpodHRwOi8vY2
ExLmN0aS1nb3YuZGsvb2NzcDA9BgNVHR8ENjA0MDKgMKAuhixodHRwOi8vY2ExLmN0aS1nb3YuZGsvb2Nlcy9yb290L2Nyb
C9yb290LmNybDAdBgNVHQ4EFgQUfyif2XGZQuJ159c1di5NCCVtdl4wDgYDVR0PAQH/BAQDAgEGMEEGCSqGSIb3DQEBCjA0
oA8wDQYJYIZIAWUDBAIDBQChHDAaBgkqhkiG9w0BAQgwDQYJYIZIAWUDBAIDBQCiAwIBQAOCAgEAuJcPSd3E6SqABGEckwv
dF0W0TrScGglvOyyX5Sx7jChPy1FFJS8vxgoqd8CYoXhzzRzuaTLLWE0P7sz51sZJKGMMA4p+f7/h4ftzfEON1359fckE/S
K5RA9IDdzVmiKxDoi8bAO/CyO/mXnZm0WFdtsU7OGgHCwAItIjNLanSnRijazcwsNfxNJoh99S6dAV/GGzo9NMPIRrLNG3I
ZboPbQe07MT/DFeHP7/xA5KxB2SdnPTfFevulJOARq4Fv+rThaSp+j67AhmzI6qrIj8r4GlCyr3+EI474ieZeiiClMtT+Db
HpPPfDMvNk1DYOqWNRi4warsDXhVkhXdGZrAP6OE3hnVbS94q5B7FhwugFl5XzSu06F3s3BLHrEbMCff2QZuGtf2xZIMRlW
O8tivguxM2KLJYewGbOuMuk1SQcB22fF0NQH2yzqnW6DFTGORDgYfy8IJItv82IQxKiS7/FMaoMp3mLyqb9dNrgRyvnxwMu
zLiVBDSJX0Hf78Nx/MwFvdbLawzdi9Fro9EWJ1hzjfJMZTiCP7my/E5UJMj/Sjce3jLep25c4pu8qrY7PC6o4OqICOkLJAV
NL1TU+OVpKnUd7flWBOA4rHaR0dZY+OC/ZQhymMgT6Zg+Yj6d3cNwh8tiJTlCUYSlX27NP2HGhjZ5OlyHcOXtvQ3VlJhI4=
"
    ],
    "typ": "oauth-authz-req+jwt",
    "alg": "RS256"
}
```


Body


```
{
    "response_uri": "https://test-tool.test.tegnebog.dk/verifier/wallet/direct_post.jwt",
    "aud": "https://self-issued.me/v2",
    "iss": "x509_hash:EPD6X85fbUtA5uMgM4XyVU3FHJ0rTfQVW_vUEGwtETY",
    "response_type": "vp_token",
    "state": "2GgzSSRf8m8N4e_12eDVqb1E1ppgyJGUvpxF0QGzo0c",
    "dcql_query": {
        "credentials": [
```

```
            {
                 "id": "2GgzSSRf8m8N4e_12eDVqb1E1ppgyJGUvpxF0QGzo0c",
                 "format": "mso_mdoc",
                 "meta": {
                     "doctype_value": "eu.europa.ec.eudi.pid.1"
                 },
                 "claims": [
                    {
                         "path": [
                             "eu.europa.ec.eudi.pid.dk.1",
                             "full_name"
                         ]
                     },
                    {
                         "path": [
                             "eu.europa.ec.eudi.pid.1",
                             "birth_date"
                         ]
                     }
                 ]
             }
         ]
    },
    "verifier_info": [
       {
            "format": "dktb-rpr+jwt",
            "data":
"eyJ4NWMiOlsiTUlJQ1NqQ0NBZkdnQXdJQkFnSVVkSjNhUEI5em5LaEZoME5DQ3RzQm15aEhjZGd3Q2dZSUtvWkl6ajBFQX
dJd2JURUxNQWtHQTFVRUJoTUNSRXN4RXpBUkJnTlZCQWNNQ2t2RHVHSmxibWhoZG00eElUQWZCZ05WQkFvTUdFUnBaMmwwW
Vd4cGMyVnlhVzVuYzNOMGVYSmxiSE5sYmpFTU1Bb0dBMVVFQ3d3RFMwVkJNUmd3RmdZRFZRUUREQTlFUzFSQ0lFbHpjM1Zw
Ym1jZ1EwRXdIaGNOTWpVeE1qRTJNVGMwT0RRNVdoY05Nall4TWpFMk1UYzBPRFE1V2pCNU1Rc3dDUVlEVlFRR0V3SkVTekV
UTUJFR0ExVUVCd3dLUzhPNFltVnVhR0YyYmpFaE1COEdBMVVFQ2d3WVJHbG5hWFJoYkdselpYSnBibWR6YzNSNWNtVnNjMl
Z1TVF3d0NnWURWUVFMREFOTFJVRXhKREFpQmdOVkJBTU1HMFJMVkVJZ1VtVnNlV2x1WnlCUVlYSjBlU0JTWldkcGMzUnllV
EJaTUJNR0J5cUdTTTQ5QWdFR0NDcUdTTTQ5QXdFSEEwSUFCSFpva2xlTlFKemdOajc2M2NXWk1pUGFpWFAvb1gxUTZIQjc2
TXNTTTNDM1Bhd3pmQ2RLWWFqa1lDSW9jWHBaOWx0RFJrVTZSdkNOMXhyZ1U2R0VXVE9qWXpCaE1BOEdBMVVkRXdFQi93UUZ
NQU1CQWY4d0RnWURWUjBQQVFIL0JBUURBZ0VHTUI4R0ExVWRJd1FZTUJhQUZLSXBWZW4xU2xRMm1tN2VXeERYc0JEZ2x2M2
lNQjBHQTFVZERnUVdCQlNRNEs3YVgrSGhZQ1Z3RUtsZlRkMERZNk50dURBS0JnZ3Foa2pPUFFRREFnTkhBREJFQWlCYTdTL
1ZNU3lmbmdTQ2NKZ3N3M1Y4VzQzdXhuWVNYZXY4WnhNZUsvcWJpUUlnZkdDdkFzZGtqUlNxRHYvOHZLbEJaOEZ3YVUwL09Y
ZW9MYUxnY3FUdzBYWT0iXSwidHlwIjoiZGt0Yi1ycHIrand0IiwiYWxnIjoiRVMyNTYifQ.eyJzdWIiOiJ4NTA5X2hhc2g6
RVBENlg4NWZiVXRBNXVNZ000WHlWVTNGSEowclRmUVZXX3ZVRUd3dEVUWSIsImlzcyI6Imh0dHBzOi8vcmVseWluZy1wYXJ
0eS1yZWdpc3RyeS50ZXN0LnRlZ25lYm9nLmRrIiwiYWxsb3dlZF9uYW1lcyI6WyJNYXJyaW90dCJdLCJleHAiOjE3OTc0ND
MzMjksImlhdCI6MTc3MzcyNjgyNiwianRpIjoiMGU2ZTQxNTYtZDg5NS00M2Q4LWI1NTgtN2VkOGRhYmIxMzUyIiwic3Rhd
HVzIjp7InN0YXR1c19saXN0Ijp7ImlkeCI6MjUyOTYsInVyaSI6Imh0dHBzOi8vZGt0Yi1ycHItY2RuLnRlc3QubXlyYWNk
bi5jb20vc3RhdHVzLWxpc3RzLzJlOTViZGEwLTc2OWEtNDA5Mi05Y2M1LWE0M2U1ZGJiN2E3OSJ9fX0.G_LVIEXILr6lLpf
CznXB9YSenTBzT-7AAyCO_nvW6DHr-7dwteWcXqqYhhJKBC0XaoExZj1hkMGkQqqE8xybDQ"
       },
       {
            "format": "dktb-info+json",
            "data":
                {
                   "name": "Marriott"
                }
        }
    ],
    "nonce": "75275cf5-5c6b-46f2-9db3-a8e02c102b92",
    "client_id": "x509_hash:EPD6X85fbUtA5uMgM4XyVU3FHJ0rTfQVW_vUEGwtETY",
    "client_metadata": {
        "encrypted_response_enc_values_supported": [
            "A128GCM",
            "A256GCM"
        ],
        "jwks": {
```

```
                    "keys": [
                       {
                            "kty": "EC",
                            "use": "enc",
                            "crv": "P-256",
                            "kid": "2GgzSSRf8m8N4e_12eDVqb1E1ppgyJGUvpxF0QGzo0c",
                            "x": "1u6_wMZTR8SchrS5xkQdxfGBb39FonHjcS74Uswjjt0",
                            "y": "Vp8Y4PPADKORsPNvoc8ITBXKVT0hefvMH5DylCKotlU",
                            "alg": "ECDH-ES"
                        }
                    ]
                },
                "vp_formats_supported": {
                    "mso_mdoc": {
                        "issuerauth_alg_values": [
                            -7
                        ],
                        "deviceauth_alg_values": [
                            -7
                        ]
                    }
                }
            },
            "response_mode": "direct_post.jwt"
        }
```

9.3 Authorization Response

[HAIP] Authorization Responses are encrypted. In response to the example Authorization Request above, AltID responds with the following:

```
POST /verifier/wallet/direct_post.jwt HTTP/1.1

Host: test-tool.dev.tegnebog.dk
Content-Type: application/x-www-form-urlencoded

response=eyJhbGciOiJFQ0RI…
```

Here the `response` parameter holds the encrypted JWT containing the `vp_token`. The following response may be decrypted using this key:

```
{
    "kty": "EC",
    "use": "enc",
    "crv": "P-256",
    "kid": "2GgzSSRf8m8N4e_12eDVqb1E1ppgyJGUvpxF0QGzo0c",
    "x": "1u6_wMZTR8SchrS5xkQdxfGBb39FonHjcS74Uswjjt0",
    "y": "Vp8Y4PPADKORsPNvoc8ITBXKVT0hefvMH5DylCKotlU",
    "d": "KsXmyJdXPsAht2Q0O42Kg7BDZ77acywI_iwFpP7Kvzo",
    "alg": "ECDH-ES"
}
```

Encrypted `response`:

```
eyJhbGciOiJFQ0RILUVTIiwiZW5jIjoiQTI1NkdDTSIsImVwayI6eyJjcnYiOiJQLTI1NiIsImtpZCI6IkU3NzM0QTBELTRFRjQtNDZEOC1CO
EM4LTQ1ODUxMUJCMzQ0QiIsImt0eSI6IkVDIiwieCI6ImZhRnZLWmpVMXNUVF9ISlJleDFPcFV2SXpmUkJPVXZEUXg2M1BSLU1PZFkiLCJ5Ij
oiczdKdzV4dWNGbWxKZ1FWTFh0aTQzTnIzdlgzcmZzYlZ0V3BEbzZ5Y185YyJ9LCJraWQiOiIyR2d6U1NSZjhtOE40ZV8xMmVEVnFiMUUxcHB
neUpHVXZweEYwUUd6bzBjIn0..26o_Dyx00sQTHLyz.mvE4JD73al-
```

mVbXu0kLVPm0mgF9r1HznPgdFWL8vNtOQK3d4uxMPS3_8AnYEnOEMAtkyajPTmBH2WelZ5DulWrUfatT_YQyQPP7Bb-d3h5s7JLVm-
OdDFZB6duLfFgHBW4I2oncR2ZwJO5-PuxursEZ6yQHezFbM6CwYTsPN405tMGzK20-
6eYgFThS1zt4NZjwS7vF7sNKlNYIMDRTS31jiLAcgatPGAXMTHHu_KATn4upPyoXLxjgW4p8vMiLJVXo3Pq-
YrgfxiGR9NxaGqSAKanpgdvWVhP4rfc7ikB2-Rn9oP8MrMdf6Kow1q0mCKe1ON4G3dsabCt0cm1ezHPQn-KtB0d3iV-cv1XoLUI8Tclj-
Q9CDhuXHR_gjmGi59L-NTPIBftrRuEkLWvbvXbOkOF53XPabmstn4um7YbWHSE2JCvY3ie3721TK8eRrYiAst752suY-
BPLIC9BjTLP9LRW4cTpjtmyCZRivyn6nkeg4LupjIzbbVbLYuI782YGF-
MFW67GlOIphf9K_EfoE4xw79k0079ElaoFOc25x7OubJc0bHyY8aLGO09rw3ZUYBic6gGBFhT6EqETeWz3Lv1Fq2j4XMAAsMzY2quEtuv9M3j
Vlb5wQO0DXi-
tRChDC7MWFrC0fOmT0jb1PO6NJOWygJKzu9RzAddvmhWAVp4U1UJNCujqAeZv4sq_nqlY64go4cWpWNYTjY1XLHZkDkuTVsFphzjaOEowJRts
zZ2iwGRJmG4fNWXCpTrXSkJm-jAK8d0h2g26g-stglskzIqi5De4-WLINx4ofqWbKUJRwuopyExYSl3Y8CqvirYajg1EEiECdLlOBB-
WFAJtZ-Xu-BPmlIrzIlqxEOzvp6y79SpqYNJIvklO2skW7tjFy9XKHKzfW90IWMtAXhDwBSgt1kd6qn6YEPYefXm-
XKVCnwV4vi7IJSyCYehPxTHi3YLdt_fibT2OcTudMMW2QoxixiciK_ZugRFUR_rhv-
VkxsXYyakdwJZWislfbEpS8rlMkkqR9tccRRY8klD9BA6qdNuI1fV7V3jR17WtcRG5jWPTdSyIzLC4H-0Xq7YcTyKpTW8SMtlRjixUkv-
YjIplnfj9rxB2pE5-SrLHef0PHODqBhdbRvnushX6xtJ9jkI4g-INlTJAJNYMQVRUGFRk3CealPGoD8j76ZdmV2XaP-
Qz48tteeikFhw_lJ0O3ph5AtvuDTlQjok5AQ8pIxebWywIfwZ5udhrCn3XZ2d5WmYWa-I0zHAttyGlnPt3Qk6PwruY2XMS04WiRFGms0-
DW1_Zgc2LLHXV4imvba1rIiT2uxdGvseRhUVb2rYo7_CKEJUGO1a6W_zOkdosL37pJmwm0me0TI_JuQ2Lv6Nri6-YbBTBOXN62yaxJ3Cf-
K9X_sk3kHfcAHC5xUNjTzhNqrLPLmmYkt6qrlOVffHAafltjObK-KB4JNG93qdU2qDxkJ7XHD40fqnpzynVo_hq8DJNkCUP-
Xpn4khUG0Ve_5N5taVrFh48FEjP8birt6D5hejOd9hK1u5FOPOhe_w4cO9oBdUvw-
98bEnzbC_EPBB6Eh5Rs3tLNCoMHxKKWlAR5geQQPqFudJAKqNPorJ4UIAyTbJVEBGJb5a2N80mlUVd9IbYjvMynsM66bwoVnrJNF4z_F9imte
iMZXQhYOnsDEnd6mCHqyUFyL0GtiwPLgruPsQaCAVG4cTixITJz95jt8Y5pmgeh-slHd-fQTiraxHvOoL_hPaaJh6EMJ34iCoHNcmYq-
Zmjxqh7tuySacpiwZanWFaqga1LU9fNlgAExRQcoyuSP1TCW2ygMnb-1P-
bpJRPCc9O0D0ko1BC3QrLjNvL5VWnqDZwavjfxtysfaDCj5jLoFj-gFhL1q_wV5TYFV7N-
aGWViqIyVYmYzqVedgWc5lPfxzS2NWOGdCjod7AFrq13rZhNyx0tZlHV6aiEOgDJUWXfV6RsSeC_VdVpi0pCOxuAIMxx04NULoIlPvT68buIG
NeZN0QMk6EXPwL4Ud6IyKm51n2ew-ndceXwBLTPFcVuo_vYCeSsFqIqgNIeGGOpdleFjhHm7J-
6lIdbpxGg5xrA_c8M1MXZcHsZhXAzm_niW0yL_P5DWze5MKGz4QxRlNcAd841vwp6VEf03GDp7EsOTc_iP7D82_Z8FtiFuKJnPEepidyUgcL7
kW0AcKMOJ2vmKR7sbyORVPjWBboaNJNOOnKtVmiYvvp2PZ1-ZawtbbrZKP6V4yuWMBQoNh-fN-
K1wcotfpAULgAC_r4dmYpFs_QwrpVVLnslG53PEkLdSvtPs6OoXUuvntlZdOOoWmcfvdI4zo-
D0iqmYLySM0l3XMcQpxdxKUksxcF3U4j59sacjWT6eqLr5-vz74QZk2V-
4JdEzBoo6f2n3v7XZyWrMqDiIyehS3PBW75_OCTST087hxaMKELal6uyhQ7Nf6N8Su9tE0iiKfwb6s1iVrFl0u9yoPgLLofkmdVYAGPzfavDk
D0C1WACN97sPpuZvg5n-cB0VWT-
Og16GYXL0SuYHkmYV3ZGG_d1z32E0Yr3241rC8aAXX9MpbR4R3x9KU8p9dS18qzAQ4nrCQ1qGBZ3dfd07OxDDsRPEVimw2qmHBAmjyfwas4z-
6uSZHXvkvXj4R-aiwmMIDx8fXZhbyX1KewXgvq8enzGJPa6e0o02j8Nfv-oyT-1D6tV3Z45uXK3GT-
6XW4BxfaBnZ0hSdXWMnQaCfzVH3TvW6h9TSiH5Y1caQUQX9sS_LoAmbBnPMzl5b0S4OOQ7msCby0lmBnI2TQdQYxJy2Gbg1S_E1FpXdFFp-
myCumWFaB_8gXOYwgo-6Je_qyW4ekc8zsYOEES6l9RVLyUHoa_Wefl6tuSldabCgCXZ8Ch_ClcOzLSjCq4Ii2O04e4nqXAepcfHSD-W-
k8CKMRpkdIbw33O_ohIW2ecsvFS3TkwP-_lNO4xEP_U4O9BXncqu3QeJBmu_DkXCkZTbYHco5ZB7Jn-
nohPC_oOtJUokKDbJIo0TU0YrJ7IMbZOLYiWOkM4AHIMCn9PiNF1EG6RmNGEe9KCHkmU-TbYG0gV6GZtPeQhlcsuCNk7AQy_b53p5n1G-
eYl4rYjR6oSwG6eiCLflItkH3JtBpRy-DiwQeQjt-
0jHyffHvDyIDwBEQRUW_f44GiMkfIkZcmoJY4Ef0KsbWosYMOJRU_yH5RJ2ww03ysQ9jlFuOLtlG742MQBflBghSCaxMn3VbYiNzoKoaGUQnx
eyKiu3zhZbdAEbKMzBJlQ9tbFps5OBS9MLfG4zri9BaHBqWoTwae3ifCqLtd0obEoFmvc0iVGMCb3IOekqIKrJAxEcSgSsjZ7WyIzvpFkBryS
38Yb1QXcPboT5jy2jyoFxj2MqVlJJ45mym-
RCAkRLOykEV1G3fkpz2NBYeI6mUmDEpiZo1a6PohXdBiNNlJmbiI7XO6ZYwCKqdbD9yQOMd2EtAu0rU8cZ88cn1D4qw052HC91P78mQSrNz5p
cjTfbaHWg7qBakRH68njBTowmJUIyPbtas1e5HEmhMMmREeJp-ET1WSIuSwm-cK0afkN9tt-F4jTsGZeq9VA4L4lH8-
LacOzte2i1tNm9HJmYzF2iePKygyry8Q-
z0F0ROLJhCApUU4p2e01WS5e8CTe7dyuXYzZRoTd1wFKLbHUZBF7ruF8UIrWPXU6ID3Iv9LG4OD5o1Kn8wmUtPLFlpriHalVXzuzCtdrcy8Bh
8YV24mQYJ8KKh1TfO6X_yIZPkv6Gq-
JLeLQqFMUWttBVYv7Vp9yRSlifFuGhCfarwYoWcRcwfGqJle4S8TKz8yWcOwjqV5_FhUud1_im9A.zEyD1L_yWS0iovmYTx4e_Q

The response includes this header:

```
{
    "kid": "2GgzSSRf8m8N4e_12eDVqb1E1ppgyJGUvpxF0QGzo0c",
    "enc": "A256GCM",
    "alg": "ECDH-ES"
}
```

(The response in the example also includes `epk` claim specifying public key, which should be ignored by Verifier). The Verifier MUST use `kid` value to retrieve corresponding private key and decrypt the response.

The decrypted payload is shown below:


```
{
    "state": "2GgzSSRf8m8N4e_12eDVqb1E1ppgyJGUvpxF0QGzo0c",
    "vp_token": {
        "2GgzSSRf8m8N4e_12eDVqb1E1ppgyJGUvpxF0QGzo0c": [
"o2d2ZXJzaW9uYzEuMGlkb2N1bWVudHOBo2dkb2NUeXBld2V1LmV1cm9wYS5lYy5ldWRpLnBpZC4xbGlzc3VlclNpZ25lZKJqbmFtZVNwYWNl
c6J4GmV1LmV1cm9wYS5lYy5ldWRpLnBpZC5kay4xgdgYWFukaGRpZ2VzdElEAWZyYW5kb21QJyeRcILpTQZuN-
9L2vEMzXFlbGVtZW50SWRlbnRpZmllcmlmdWxsX25hbWVsZWxlbWVudFZhbHVlbkFuZGVycyBDYXJsc2Vud2V1LmV1cm9wYS5lYy5ldWRpLnB
pZC4xgdgYWFikaGRpZ2VzdElEBGZyYW5kb21QDCSPg2aQ-
RYd70xujXBWZXFlbGVtZW50SWRlbnRpZmllcmpiaXJ0aF9kYXRlbGVsZW1lbnRWYWx1ZWoxOTc3LTAxLTAxamlzc3VlckF1dGiEQ6EBJqEYIV
kCTjCCAkowggHwoAMCAQICFEJHyJBiXLqU6mv0sc1tAZeDbL3PMAoGCCqGSM49BAMCMG0xCzAJBgNVBAYTAkRLMRMwEQYDVQQHDApLw7hiZW5
oYXZuMSEwHwYDVQQKDBhEaWdpdGFsaXNlcmluZ3NzdHlyZWxzZW4xDDAKBgNVBAsMA0tFQTEYMBYGA1UEAwwPREtUQiBJc3N1aW5nIENBMB4X
DTI1MDYxODE0MjM1MVoXDTI2MDYxODE0MjM1MVowdDELMAkGA1UEBhMCREsxEzARBgNVBAcMCkvDuGJlbmhhdm4xITAfBgNVBAoMGERpZ2l0Y
Wxpc2VyaW5nc3N0eXJlbHNlbjEMMAoGA1UECwwDS0VBMR8wHQYDVQQDDBZES1RCIENyZWRlbnRpYWwgSXNzdWVyMFkwEwYHKoZIzj0CAQYIKo
ZIzj0DAQcDQgAEpHBVn6kQu5ZKnzSFrPLLfHXEX6ZxWlhfES5MUbsOMxPeFEz0zPOQLvJU278M-
mBHcJF1ZUy294qYs7D2PacIH6NnMGUwDgYDVR0PAQH_BAQDAgeAMB8GA1UdIwQYMBaAFKIpVen1SlQ2mm7eWxDXsBDglv3iMB0GA1UdDgQWBB
QwUoAEIiXsQfN2bcL3c_uKFUnytDATBgNVHSUEDDAKBggrBgEFBQcDAzAKBggqhkjOPQQDAgNIADBFAiEAz6cGyitgDHQYBfn87kW2Ww4sJyJ
agTP2iElqYc7QyawCIBUqQWHPNFRx_pqFYJXkVQxeozlVWvTlOP_MnD3nU5KwWQOw2BhZA6unZ3ZlcnNpb25jMS4wb2RpZ2VzdEFsZ29yaXRo
bWdTSEEtMjU2bHZhbHVlRGlnZXN0c6J4GmV1LmV1cm9wYS5lYy5ldWRpLnBpZC5kay4xoQFYIP2McMds6jqy2yLS32eRbqEwWbEL2DVJr9A9F
Vm_O_Cnd2V1LmV1cm9wYS5lYy5ldWRpLnBpZC4xqgJYIAhrnL_yCGhhD5VcVlekgi14DSrrX7io7Y_mw_Q4Yel9A1ggPI5ZA1beTf_WAgtCcf
0LxA-
LXXcpTre8VlPDoaXo6NcEWCCZBlJt3xlrXy_NfjDuscA3ikndRW8M3Pp8xPuORJ4kLQVYIFRPB48zdYo_CNg9rcDsf7mwKMrkgiDJJuIaY1PQ
yWFCBlggguIRl5dfbEaSo50fHWUJ35hoMCYMeEfuv8GzB2dL69wHWCA0-0pvsf-
KOL79pAYXXjXDkUNpBW7KitGERRjCw0F26QhYIFHwMxPXN4-
7NI4lRjfhQdQfmGgfHHE6wa_UbJ_etGMPCVggDbn4eqwxuEyMTclVVpqLbpoPpx1fzfRhohCijcRbsVkKWCBkoVpoktRh7vlYFImLdnsVZBTW
BaO8JoRemk7CN1BaHgtYICyg-Qg60qeIuche0Hc22l59dsPf9My8TPbhtdyl1OdtbWRldmljZUtleUluZm-
iaWRldmljZUtleaUBAiABIVgg3ZPxpQElWtMjGevCw1yxfNkBMnDejre1LBC_cP6VUzYiWCCQAv-
r1UIbDbzthzdT7TenBMBHxBGPMIDDksDbM-
zPCgMmcWtleUF1dGhvcml6YXRpb25zoWpuYW1lU3BhY2VzgngaZXUuZXVyb3BhLmVjLmV1ZGkucGlkLmRrLjF3ZXUuZXVyb3BhLmVjLmV1ZGk
ucGlkLjFnZG9jVHlwZXdldS5ldXJvcGEuZWMuZXVkaS5waWQuMWx2YWxpZGl0eUluZm-
jZnNpZ25lZMB0MjAyNi0wMy0xMFQwMDowMDowMFppdmFsaWRGcm9twHQyMDI2LTAzLTEwVDAwOjAwOjAwWmp2YWxpZFVudGlswHQyMDI3LTAz
LTEwVDAwOjAwOjAwWmZzdGF0dXOha3N0YXR1c19saXN0omNpZHgZGGRjdXJpeFpodHRwczovL2RrdGItaXNzdWVyLWNkbi50ZXN0Lm15cmFjZ
G4uY29tL3N0YXR1cy1saXN0cy82MmI1YmQwZi1mY2U1LTQwZmUtODc3ZC1hZDEwMjU3NjNlOTNYQCOsQFssGedmNIdFvgB1ks5LW8-
KDfq79R18dsUxkdwCtJoFGTE1QJX9K1nCJ-
FX8MSzCfOvJ1yq1xgO_FoLDBlsZGV2aWNlU2lnbmVkompuYW1lU3BhY2Vz2BhBoGpkZXZpY2VBdXRooW9kZXZpY2VTaWduYXR1cmWEQ6EBJqD
2WECgMgmrrgDZ_9tV5wO8mSU1EleEHjzz4_1iE2BJqQK78Xj8HL8W7RMvMpMi6vpS1aJnKYgK1F1OU5zTjuRyu15PZnN0YXR1cwA"
        ]
    }
}
```


The VP Token, which contains the requested PID, is a Base64Url-encoding of a mdoc `DeviceResponse` (as CBOR). The contents are shown in CBOR annotation below.

```
 DeviceResponse = {
    "version" : tstr,                    ; Version, always "1.0" for AltID
  ? "documents" : [+Document],           ; Always exactly one document for AltID
    "status" : uint                      ; Status code
 }
```

```
{
  "version": "1.0",
  "documents": [
    {
      "docType": "eu.europa.ec.eudi.pid.1",
      "issuerSigned": {
        "nameSpaces": {
          "eu.europa.ec.eudi.pid.dk.1": [
            24(<<
              {
                "digestID": 1,
                "random": h'2727917082e94d066e37ef4bdaf10ccd',
                "elementIdentifier": "full_name",
                "elementValue": "Anders Carlsen"
              }
            >>)
          ],
          "eu.europa.ec.eudi.pid.1": [
            24(<<
              {
                "digestID": 4,
                "random": h'0c248f836690f9161def4c6e8d705665',
                "elementIdentifier": "birth_date",
                "elementValue": "1977-01-01"
              }
            >>)
          ]
        },
        "issuerAuth": [ // COSE_Sign1 signature
          h'a10126',
          {
            33:  // X.509 certificate chain
h'3082024a308201f0a003020102021442247c890625cba94ea6bf4b1cd6d0197836cbdcf300a06082a8648ce3d04030
2306d310b30090603550406130244b3113301106035504070c0a4bc3b862656e6861766e3121301f060355040a0c18
4469676974616c69736572696e677373747972656c73656e310c300a060355040b0c034b4541311830160603550403 0c
f444b54422049737375696e67204341301e170d323530363138313432333531 5a3074310b30090603550406130244b3113301106035504070c0a4bc3b862656e6861766e3121301f060355040a0c1
84469676974616c69736572696e677373747972656c73656e310c300a060355040b0c034b4541311f301d0603550403
0c16444b54422043726564656e7469616c204973737565723059301306072a8648ce3d020106082a8648ce3d0301070
3420004a470559fa910bb964a9f3485acf2cb7c75c45fa6715a585f112e4c51bb0e3313de144cf4ccf3902ef254dbbf
0cfa6047709175654cb6f78a98b3b0f63da7081fa3673065300e0603551d0f0101ff040403020780301f0603551d230
41830168014a22955e9f54a54369a6ede5b10d7b010e096fde2301d0603551d0e041604143052800422 25ec41f3766d
c2f773fb8a1549f2b430130603551d25040c300a06082b06010505070303300a06082a8648ce3d04030203480030450
22100cfa706ca2b600c741805f9fcee45b65b0e2c27225a8133f688496a61ced0c9ac0220152a4161cf345471fe9a85
6095e4550c5ea339555af4e538ffcc9c3de75392b0'
          },  // Payload, i.e. Mobile Security Object
h'd8185903aba76776657273696f6e63312e306f646967657374416c676f726974686d675348412d3235366c76616c7
5654469676573747 3a2781a65752e6575726f70612e65632e657564692e7069642e646b2e31a1015820fd8c70c76cea
3ab2db22d2df67916ea13059b10bd83549afd03d1559bf3bf0a77765752e6575726f70612e65632e657564692e70696
42e31aa025820086b9cbff20868610f955c5657a4822d780d2aeb5fb8a8ed8fe6c3f43861e97d0358203c8e590356de
4dffd6020b4271fd0bc40f8b5d77294eb7bc5653c3a1a5e8e8d70458209906526ddf196b5f2fcd7e30eeb1c0378a49d
d456f0cdcfa7cc4fb8e449e242d055820544f078f33758a3f08d83dadc0ec7fb9b028cae48220c926e21a6353d0c961
4206582082e21197975f6c4692a39d1f1d6509df986830260c7847eebfc1b307674bebdc07582034fb4a6fb1ff8a38b
efda406175e35c3914369056eca8ad1844518c2c34176e908582051f03313d7378fbb348e254637e141d41f98681f1c
```

```
713ac1afd46c9fdeb4630f0958200db9f87aac31b84c8c4dc955569a8b6e9a0fa71d5fcdf461a210a28dc45bb1590a5
82064a15a6892d461eef95814898b767b156414d605a3bc26845e9a4ec237505a1e0b58202ca0f9083ad2a788b9c85e
d07736da5e7d76c3dff4ccbc4cf6e1b5dca5d4e76d6d6465766963654b6579496e666fa2696465766963654b6579a50
1022001215820dd93f1a501255ad32319ebc2c35cb17cd9013270de8eb7b52c10bf70fe9553362258209002ffabd542
1b0dbced873753ed37a704c047c4118f3080c392c0db33eccf0a0326716b6579417574686f72697a6174696f6e73a16
a6e616d655370616365738278 1a65752e6575726f70612e65632e657564692e7069642e646b2e317765752e6575726f
70612e65632e657564692e7069642e3167646f63547970657765752e6575726f70612e65632e657564692e7069642e3
16c76616c6964697479496e666fa3667369676e6564c074323032362d30332d31305430303a30303a30305a6976616c
696446726f6dc074323032362d30332d31305430303a30303a30305a6a76616c6964556e74696cc074323032372d303
32d31305430303a30303a30305a66737461747573a16b7374617475735f6c697374a2636964781918646375726978 5a
68747470733a2f2f646b74622d6973737565722d63646e2e746573742e6d79726163646e2e636f6d2f7374617475732
d6c697374732f36326235626430662d666365352d343066652d383737642d6164313032353733633653933',
          //Signature
h'23ac405b2c19e766348745be007592ce4b5bcf8a0dfabbf51d7c76c53191dc02b49a051931354095fd2b59c227e15
7f0c4b309f3af275caad7180efc5a0b0c19'
        ]
      },
      "deviceSigned": {
        "nameSpaces": 24(<< { } >>),
        "deviceAuth": {
          "deviceSignature": [6           h'a10126', // Protected header: CBOR encoding of {1:
-7}, i.e. alg_id = ES256
            { },       // Unprotected header, empty
            null,
h'a03209abae00d9ffdb55e703bc9925351257841e3cf3e3fd62136049a902bbf178fc1cbf16ed132f329322eafa52d
5a26729880ad45d4e539cd38ee472bb5e4f'
          ]
        }
      }
    }
  ],
  "status": 0 // Device response status: OK
}
```

If the Verifier has successfully processed the Authorization Response (according to Appendix D: Validation of OID4VP Authorization Responses), it MUST respond with an HTTP 200 status code with Content-Type of `application/json` and a JSON object in the response body. If the same-device flow is used, the `redirect_uri` and `browser` claims MUST be included in the JSON-object as described in 4.3.1.1. In case of validation error(s), the Verifier MUST instead respond with an error response (HTTP 400) and include a user-friendly error description.

10 Appendix F: OID4VP example for [AVP]

This section will give a detailed example of presenting PoA over the OID4VP protocol according to [AVP], using the sample Verifier application provided by the European Commission [AVV].

The following section is based on and aligned with the EU Age Verification Profile [AVP], ensuring consistency with the EU approach to online age verification and the interoperability profile developed in this context. This includes specific requirements for the implementation of the OID4VP protocol.

10.1 Authorization Request

A Verifier can request a PoA by making an Authorization Request as shown below. Specifically, the following requests the user to present the `age_over_18` attribute from their PoA in the ISO mdoc format (as specified by the contents of `dcql_query`). The Authorization Request below should be presented to the User either as a (navigable) link or rendered as a QR code that can be scanned by the User's device. The example has been indented and URL decoded for readability.

```
av://?
   response_type=vp_token
   &response_mode=direct_post
   &client_id= redirect_uri:https://verifier-backend.ageverification.dev/wallet/direct_post/
ktTY6nFkVdv2oAmkqt6pAEQQAUiPY8PlCjRrhHW36AQrrIYtOfREVxIOKJrPw0JTEAP9H_0WJ3xhw2-qyrKUng
   &response_uri= https://verifier-backend.ageverification.dev/wallet/direct_post/
ktTY6nFkVdv2oAmkqt6pAEQQAUiPY8PlCjRrhHW36AQrrIYtOfREVxIOKJrPw0JTEAP9H_0WJ3xhw2-qyrKUng
   &dcql_query={
     "credentials": [
       {
         "id": "proof_of_age",
         "format": "mso_mdoc",
         "meta": {
           "doctype_value": "eu.europa.ec.av.1"
         },
         "claims": [
           {
             "path": [
               "eu.europa.ec.av.1",
               "age_over_18"
             ]
           }
         ]
       }
     ]
   }
   &nonce=0493288b-478e-4aef-beb2-f71931fc2603
   &state=ktTY6nFkVdv2oAmkqt6pAEQQAUiPY8PlCjRrhHW36AQrrIYtOfREVxIOKJrPw0JTEAP9H_0WJ3xhw2-qyrKUng
```

See section 4.2 for a description of the included parameters. Note that in this example correlation to a specific request/transaction is obtained by including an identifier ("ktTY6...") in the response URI, which is a valid alternative to using `state`. Both are included in the example for completeness.

10.2 Authorization Response

In response to the example Authorization Request above, AltID responds with the following:

```
POST /wallet/direct_post/ktTY6... HTTP/1.1
Host: verifier-backend.ageverification.dev
Content-Type: application/x-www-form-urlencoded

  state=ktTY6nFkVdv2oAmkqt6pAEQQAUiPY8PlCjRrhHW36AQrrIYtOfREVxIOKJrPw0JTEAP9H_0WJ3xhw2-qyrKUng
  &vp_token=
o2d2ZXJzaW9uYzEuMGlkb2N1bWVudHOBo2dkb2NUeXBlcWV1LmV1cm9wYS5lYy5hdi4xbGlzc3VlclNpZ25lZKJqbmFtZVNwYWNlc6FxZXU
uZXVyb3BhLmVjLmF2LjGB2BhYT6RoZGlnZXN0SUQEZnJhbmRvbVBxl2rgO0ACxMK-
K4RbPc7RcWVsZW1lbnRJZGVudGlmaWVya2FnZV9vdmVyXzE4bGVsZW1lbnRWYWx1ZfVqaXNzdWVyQXV0aIRDoQEmoRghWQJOMIICSjCCAfC
gAwIBAgIUQkfIkGJcupTqa_SxzW0Bl4Nsvc8wCgYIKoZIzj0EAwIwbTELMAkGA1UEBhMCREsxEzARBgNVBAcMCkvDuGJlbmhhdm4xITAfBg
NVBAoMGERpZ2l0YWxpc2VyaW5nc3N0eXJlbHNlbjEMMAoGA1UECwwDS0VBMRgwFgYDVQQDDA9ES1RCIElzc3VpbmcgQ0EwHhcNMjUwNjE4M
TQyMzUxWhcNMjYwNjE4MTQyMzUxWjB0MQswCQYDVQQGEwJESzETMBEGA1UEBwwKS8O4YmVuaGF2bjEhMB8GA1UECgwYRGlnaXRhbGlzZXJp
bmdzc3R5cmVsc2VuMQwwCgYDVQQLDANLRUExHzAdBgNVBAMMFkRLVEIgQ3JlZGVudGlhbCBJc3N1ZXIwWTATBgcqhkjOPQIBBggqhkjOPQM
BBwNCAASkcFWfqRC7lkqfNIWs8st8dcRfpnFaWF8RLkxRuw4zE94UTPTM85Au8lTbvwz6YEdwkXVlTLb3ipizsPY9pwgfo2cwZTAOBgNVHQ
8BAf8EBAMCB4AwHwYDVR0jBBgwFoAUoilV6fVKVDaabt5bENewEOCW_eIwHQYDVR0OBBYEFDBSgAQiJexB83Ztwvdz-
4oVSfK0MBMGA1UdJQQMMAoGCCsGAQUFBwMDMAoGCCqGSM49BAMCA0gAMEUCIQDPpwbKK2AMdBgF-
fzuRbZbDiwnIlqBM_aISWphztDJrAIgFSpBYc80VHH-
moVgleRVDF6jOVVa9OU4_8ycPedTkrBZAqPYGFkCnqZndmVyc2lvbmMxLjBvZGlnZXN0QWxnb3JpdGhtZ1NIQS0yNTZsdmFsdWVEaWdlc3R
zoXFldS5ldXJvcGEuZWMuYXYuMakBWCBm93skX28Yj3O_G93i9T3WMfdigWUueNhAp2exRgB7cAJYIK73eOP-
kfwNPGjHugLX1NgmFbAZthjpRL-1PQO0DbLBA1ggmxaiVxiXJINyFhbYVKNgZYmB9PD_CmG9-ggv3-
RRvgMEWCCfY8jdWZtLgBwFTdrt2LUH2q8kn2tpNTaSKvuJxQteIwVYINxhzP4pshcAYZnT42NohAczvs2exLnvLULmejhjymY2BlggOnERd
9s2V9bAQRy0xdh-
_8Vgma9fne_hQvDUChiC0a0HWCC1eUMeOd1px2LRevJ9C1b2sLYyYAs9t88AbmeWR5ItyQhYIOmQTZxZ_hGuinCLhoq6QNyP2OVGj5oIPcn
WSJJfvFd9CVggAjn1XsFaSwetb-2Rg_Bu3NDV_-
BtdPPz0pgICNtK4QdtZGV2aWNlS2V5SW5mb6JpZGV2aWNlS2V5pQECIAEhWCC04B2AigIvEauRoxhNAhDNBt5Q9h0SVQpbc6HrQ39WcSJYI
GkAg8x6uoRNaL9Ph7nRvcTTWcWO5-z6ZDE3ozA-
U_tPAyZxa2V5QXV0aG9yaXphdGlvbnOham5hbWVTcGFjZXOBcWV1LmV1cm9wYS5lYy5hdi4xZ2RvY1R5cGVxZXUuZXVyb3BhLmVjLmF2LjF
sdmFsaWRpdHlJbmZvo2ZzaWduZWTAdDIwMjYtMDQtMDdUMDA6MDA6MDBaaXZhbGlkRnJvbcB0MjAyNi0wNC0wN1QwMDowMDowMFpqdmFsaW
RVbnRpbMB0MjAyNi0wNS0wN1QwMDowMDowMFpYQJH-7SFWi3EH6sYKvzpFGsiBSs6Op7gPyqiSiSRRINr31hgMtKaUSaR-
g1llHayUXdD7li5ffiQ2aAwv1R_K_4ZsZGV2aWNlU2lnbmVkompuYW1lU3BhY2Vz2BhBoGpkZXZpY2VBdXRooW9kZXZpY2VTaWduYXR1cmW
EQ6EBJqD2WED8l-
IxbUGKtxWlL3bLlGwrvgcs_SyS8vMjmDuCSrGG84ftLLyVePwkEafrhsDOAZpu_24rrXXxiO2XgEEYYq_9ZnN0YXR1cwA
```

The VP Token, which contains the requested PoA, is a Base64Url-encoding of a mdoc `DeviceResponse` (as CBOR). The contents are shown in CBOR annotation below.

```
DeviceResponse = {
   "version" : tstr,                    ; Version, always "1.0" for AltID
 ? "documents" : [+Document],           ; Always exactly one document for AltID
   "status" : uint                      ; Status code
}

{
  "version": "1.0",
  "documents": [
    {
      "docType": "eu.europa.ec.av.1",
      "issuerSigned": {
        "nameSpaces": {
          "eu.europa.ec.av.1": [
            24(<<
```

```
              {
                "digestID": 4,
                "random": h'71976ae03b4002c4c2be2b845b3dced1',
                "elementIdentifier": "age_over_18",
                "elementValue": true
              }
            >>)
          ]
        },
        "issuerAuth": [ //COSE_Sign1 signature
          h'a10126',
          {
            33: //X.509 certificate chain
h'3082024a308201f0a003020102021442247c890625cba94ea6bf4b1cd6d0197836cbdcf300a06082a8648ce3d040302306d310b300
906035504061302444b311330110603550407 0c0a4bc3b862656e6861766e3121301f060355040a0c184469676974616c6973657269
6e677373747972656c73656e310c300a060355040b0c034b4541311830160603550403 0c0f444b54422049737375696e67204341301
e170d3235303631383134323335315a170d3236303631383134323335315a3074310b3009060355040610 6035504
070c0a4bc3b862656e6861766e3121301f060355040a0c184469676974616c69736572696e677373747972656c73656e310c300a060
355040b0c034b4541311f301d06035504030c16444b54422043726564656e7469616c204973737565723059301306072a8648ce3d02
0106082a8648ce3d03010703420004a470559fa910bb964a9f3485acf2cb7c75c45fa6715a585f112e4c51bb0e3313de144cf4ccf39
02ef254dbbf0cfa6047709175654cb6f78a98b3b0f63da7081fa3673065300e0603551d0f0101ff040403020780301f0603551d2304
1830168014a22955e9f54a54369a6ede5b10d7b010e096fde2301d0603551d0e041604143052800422 25ec41f3766dc2f773fb8a154
9f2b430130603551d25040c300a06082b06010505070303300a06082a8648ce3d040302034800304502 2100cfa706ca2b600c741805
f9fcee45b65b0e2c27225a8133f688496a61ced0c9ac0220152a4161cf345471fe9a856095e4550c5ea339555af4e538ffcc9c3de75
392b0'
          }, // Payload, i.e. Mobile Security Object
h'd81859029ea66776657273696f6e63312e306f646967657374416c676f726974686d675348412d3235366c76616c7565446967657
37473a17165752e6575726f70612e65632e61762e31a901582066f77b245f6f188f73bf1bdde2f53dd631f76281652e78d840a767b1
46007b70025820aef778e3fe91fc0d3c68c7ba02d7d4d82615b019b618e944bfb53d03b40db2c10358209b16a25718972483721616d
854a360658981f4f0ff0a61bdfa082fdfe451be030458209f63c8dd599b4b801c054ddaedd8b507daaf249f6b693536922afb89c50b
5e23055820dc61ccfe29b217006199d3e36368840733becd9ec4b9ef2d42e67a3863ca66360658203a711177db3657d6c0411cb4c5d
87effc56099af5f9defe142f0d40a1882d1ad075820b579431e39dd69c762d17af27d0b56f6b0b632600b3db7cf006e679647922dc9
085820e9904d9c59fe11ae8a708b868aba40dc8fd8e5468f9a083dc9d648925fbc577d0958200239f55ec15a4b07ad6fed9183f06ed
cd0d5ffe06d74f3f3d2980808db4ae1076d6465766963654b6579496e666fa2696465766963654b6579a501022001215820b4e01d80
8a022f11ab91a3184d0210cd06de50f61d12550a5b73a1eb437f5671225820690083cc7aba844d68bf4f87b9d1bdc4d359c58ee7ecf
a643137a3303e53fb4f0326716b6579417574686f72697a6174696f6e73a16a6e616d6553706163657381716 5752e6575726f70612e
65632e61762e3167646f63547970657165752e6575726f70612e65632e61762e316c76616c6964697479496e666fa3667369676e656
4c074323032362d30342d30375430303a30303a30305a6976616c696446726f6dc074323032362d30342d30375430303a30303a3030
5a6a76616c6964556e74696cc074323032362d30352d30375430303a30303a30305a',
            // Signature
h'91feed21568b7107eac60abf3a451ac8814ace8ea7b80fcaa89289245120daf7d6180cb4a69449a47e8359651dac945dd0fb962e5
f7e2436680c2fd51fcaff86'
          ]
        },
        "deviceSigned": {
          "nameSpaces": 24(<< { } >>),
          "deviceAuth": {
            "deviceSignature": [
              h'a10126', // Protected header: CBOR encoding of {1: -7}, i.e. alg_id = ES256
              { },       // Unprotected header, empty
              null,
h'fc97e2316d418ab715a52f76cb946c2bbe072cfd2c92f2f323983b824ab186f387ed2cbc9578fc2411a7eb86c0ce019a6eff6e2ba
d75f188ed9780411862affd'
            ]
          }
        }
      }
    ],
    "status": 0 // Device response status: OK
}
```

If the Verifier has successfully processed the Authorization Response (according to Appendix D: Validation of OID4VP Authorization Responses), it MUST respond with an HTTP 200 status code with Content-Type of `application/json` and a JSON object in the response body. If the same-device flow is used, the `redirect_uri` and `browser` claims MUST be included in the JSON-object as described in 4.3.1.1. In case of validation error(s), the Verifier MUST instead respond with an error response (HTTP 400) and include a user-friendly error description.

11 Appendix G: Issuer Auth certificate chain and issuer signature validation

When receiving an attestation, Verifiers MUST validate the signature (`IssuerAuth`) using the public key in the first certificate included in `IssuerAuth`, which is the attestation signing certificate.

11.1 Extracting the signing certificate

The signing certificate is included in the unprotected header in the COSE signature (`IssuerAuth`), in parameter *x5chain* (label 33).

In general, this header contains this CBOR structure:

```
x5chain (33): bstr (single cert) or [+ bstr] (chain)
```

Verifiers MUST support both single certificate (bstr) and chain (array of bstr) formats.

11.2 Trust

In addition to validating the signature, Verifiers MUST ensure that the signing certificate is trusted. The method for deciding trust depends on the use case:

A. Verifiers that need to validate PoA attestations from EU citizens
B. Verifiers that only need to validate AltID attestations

In both cases Verifiers must build a set of trusted certificates (the *trust list)* on which the trust validation is based. This trust list may in general contain both root, intermediate, and leaf certificates. A signing certificate is trusted if:

- The signing certificate is included as **leaf** certificate in the trust list
- A valid certificate chain including the signing certificate as leaf can be built that terminates with either
    - an **intermediate** CA certificate in the trust list OR
    - a **root** CA certificate in the trust list

Building such certificate chains correctly is complicated and Verifiers SHOULD use standard software for this purpose, such as system packages System.Security.Cryptography.X509Certificates (.NET) or java.security/javax.security (Java) or dedicated libraries such as BouncyCastle [BC] to ensure that chain building and validation conforms to [RFC5280].

11.3 Trust lists

As mentioned, trust lists may be constructed from different sources, depending on the Verifier's use case.

11.3.1 Trust lists for EU PoA attestations

When Verifiers build the trust list artefact in their implementation and wishes to support attestations issued from multiple EU member states (use case A), it MUST be based on the trust list signed XML-files published by EU.

The EU Age Verification Trusted List can be found at the locations provided in Table 12 below.

Environment	EU trust list source
Test and development	https://acceptance.eidas.ec.europa.eu/efda/trust-services/browse/av-tl
Production	https://eidas.ec.europa.eu/efda/trust-services/browse/av-tl

Table 12: EU Age Verification Trusted List locations

When Verifiers retrieve the XML-file, they MUST verify its signature, and extract certificates to add to internal trust list from the data structure, only including certificates for the age verification service type identifier `http://ec.europa.eu/tools/lotl/av/TrstSvc/Svctype/PAA`.

Note, that the description here is not a complete description of controls that MUST be performed by Verifiers that implement EU XML trust list handling. We refer to guidelines published by EU, see links in Table 12.

11.3.2 Trust lists for AltID attestations

When Verifiers build the trust list artefact in their implementation and wishes only to support AltID attestations, they MAY also use the EU published trust lists, and restrict included certificates to those valid, and from TrustServiceProvider Digitaliseringsstyrelsen by only considering TrustServiceProvider elements with TSPTradeName VATDK-34051178. Note, that handling of EU published trust lists also in this case MUST conform to guidelines published by EU, see links in Table 12.

Verifiers MAY also directly include certificates listed in in Table 13 below.

Environment	Certificates to trust
Test and development	Issuer: C=DK, L=København, O=Digitaliseringsstyrelsen, OU=KEA, CN=DKTB Issuing CA Validity Not Before: Jun 18 14:23:51 2025 GMT Not After : Jun 18 14:23:51 2026 GMT Subject: C=DK, L=København, O=Digitaliseringsstyrelsen, OU=KEA, CN=DKTB Credential Issuer -----BEGIN CERTIFICATE----- MIICSjCCAfCgAwIBAgIUQkfIkGJcupTqa/SxzW0Bl4Nsvc8wCgYIKoZIzj0EAwIw bTELMAkGA1UEBhMCREsxEzARBgNVBAcMCkvDuGJlbmhhdm4xITAfBgNVBAoMGERp Z2l0YWxpc2VyaW5nc3N0eXJlbHNlbjEMMAoGA1UECwwDS0VBMRgwFgYDVQQDDA9E S1RCIElzc3VpbmcgQ0EwHhcNMjUwNjE4MTQyMzUxWhcNMjYwNjE4MTQyMzUxWjB0 MQswCQYDVQQGEwJESzETMBEGA1UEBwwKS8O4YmVuaGF2bjEhMB8GA1UECgwYRGln aXRhbGlzZXJpbmdzc3R5cmVsc2VuMQwwCgYDVQQLDANLRUExHzAdBgNVBAMMFkRL VEIgQ3JlZGVudGlhbCBJc3N1ZXIwWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAASk cFWfqRC7lkqfNIWs8st8dcRfpnFaWF8RLkxRuw4zE94UTPTM85Au8lTbvwz6YEdw kXVlTLb3ipizsPY9pwgfo2cwZTAOBgNVHQ8BAf8EBAMCB4AwHwYDVR0jBBgwFoAU

	oilV6fVKVDaabt5bENewEOCW/eIwHQYDVR0OBBYEFDBSgAQiJexB83Ztwvdz+4oV SfK0MBMGA1UdJQQMMAoGCCsGAQUFBwMDMAoGCCqGSM49BAMCA0gAMEUCIQDPpwbK K2AMdBgF+fzuRbZbDiwnIlqBM/aISWphztDJrAIgFSpBYc80VHH+moVgleRVDF6j OVVa9OU4/8ycPedTkrA= -----END CERTIFICATE-----
Production	<Not yet determined>

Table 13: Trust list certificates for AltID attestations

Since issuer certificates published for AltID are leaf certificates, Verifiers SHOULD monitor publication of new certificates either in the EU trust lists (Table 12) or in updates of this document. Verifiers MUST prepare their implementation to handle multiple certificates in the trust list. Digitaliseringsstyrelsen will publish new issuer certificates at least 3 months before AltID will begin using it for issuing attestations.

Information about the certificates used in the production environment will be included in this section at a later point.

12 References

[ARF]	Architecture Reference Framework for European Identity Wallets https://github.com/eu-digital-identity-wallet/eudi-doc-architecture-and-reference-framework
[AV]	European Age Verification Solution, Technical Specifications https://ageverification.dev/av-doc-technical-specification/docs/architecture-and-technical-specifications/
[AVP]	European Age Verification Solution, Age Verification Profile https://ageverification.dev/av-doc-technical-specification/docs/annexes/annex-A/annex-A-av-profile/
[AVV]	European Sample Age Verifier web application https://verifier.ageverification.dev/
[BC]	Bouncy Castle – Open-source cryptographic APIs https://www.bouncycastle.org
[CBOR]	RFC 8949: Concise Binary Object Representation (CBOR) https://www.rfc-editor.org/rfc/rfc8949.html
[CBORZ]	CBOR zone website https://cbor.zone/
[CDDL]	RFC 8610: Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) JSON Data Structures https://www.rfc-editor.org/rfc/rfc8610.html
[COSE]	RFC 8152: CBOR Object Signing and Encryption (COSE) https://www.rfc-editor.org/rfc/rfc8152.html
[JAR]	The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR) https://www.rfc-editor.org/rfc/rfc9101.html
[JWK]	RFC 7638: JSON Web Key (JWK) Thumbprint https://www.rfc-editor.org/rfc/rfc7638.html
[JWT]	RFC 7519: JSON Web Token (JWT) https://www.rfc-editor.org/rfc/rfc7519.html
[HAIP]	OpenID4VC High Assurance Interoperability Profile https://openid.net/specs/openid4vc-high-assurance-interoperability-profile-1_0-final.html

[ARF]	Architecture Reference Framework for European Identity Wallets https://github.com/eu-digital-identity-wallet/eudi-doc-architecture-and-reference-framework
[ISO18013-5]	ISO/IEC 18013-5, Personal identification, ISO-compliant driving license - Part 5: Mobile driving license (mDL) application. https://www.iso.org/standard/69084.html
[OID4VP]	OpenID for Verifiable Presentations https://openid.net/specs/openid-4-verifiable-presentations-1_0.html
[PAR]	RFC 9126: OAuth 2.0 Pushed Authorization Requests https://www.rfc-editor.org/rfc/rfc9126.html
[RFC5280]	RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile https://www.rfc-editor.org/rfc/rfc5280.html
[RPR]	AltID Relying Party Registry https://relying-party-registry.test.tegnebog.dk/
[TSL]	Token Status List (draft version 13) https://www.ietf.org/archive/id/draft-ietf-oauth-status-list-13.html
[UX]	AltID UX Scheme Coming soon.