ETSI EN 319 432 V1.3.3 (2026-07)

EUROPEAN STANDARD

DRAFT

Electronic Signatures and Trust Infrastructures (ESI); Protocols for remote digital signature creation

This DRAFT is a working document of ETSI. It is provided for information only and is for future development work within ETSI. DRAFTS may be updated, deleted, replaced, or obsoleted by other documents at any time.

ETSI and/or its Members have no liability for any current or further use/implementation of the present DRAFT.

Do not use as reference material.

Do not cite this document other than as "work in progress."

Any draft approved and PUBLISHED shall be obtained exclusively as a deliverables via the ETSI Standards search page at:

http://www.etsi.org/standards-search

Call for comments:

Any comments are requested by 30 August 2026 EOB.

Comments should be submitted using ETSI commenting template.

Comments can be submitted using e-signatures comments@list.etsi.org email list.

Reference
RTS/ESI-0019432v141

Keywords
authentication, authorization, electronic signature, protocol, remote, security, trust services

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Important notice

The present document can be downloaded from the ETSI Search & Browse Standards application.

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format on ETSI deliver repository.

Users should be aware that the present document may be revised or have its status changed, this information is available in the Milestones listing.

If you find errors in the present document, please send your comments to the relevant service listed under Committee Support Staff.

If you find a security vulnerability in the present document, please report it through our Coordinated Vulnerability Disclosure (CVD) program.

Notice of disclaimer & limitation of liability

The information provided in the present deliverable is directed solely to professionals who have the appropriate degree of experience to understand and interpret its content in accordance with generally accepted engineering or other professional standard and applicable regulations.
No recommendation as to products and services or vendors is made or should be implied.
No representation or warranty is made that this deliverable is technically accurate or sufficient or conforms to any law and/or governmental rule and/or regulation and further, no representation or warranty is made of merchantability or fitness for any particular purpose or against infringement of intellectual property rights.
In no event shall ETSI be held liable for loss of profits or any other incidental or consequential damages.

Any software contained in this deliverable is provided "AS IS" with no warranties, express or implied, including but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights and ETSI shall not be held liable in any event for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption, loss of information, or any other pecuniary loss) arising out of or related to the use of or inability to use the software.

Copyright Notification

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.
The content of the PDF version shall not be modified without the written authorization of ETSI.
The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2026.
All rights reserved.

DRAFT

Contents

List of figures 8
Intellectual Property Rights 10
Foreword 10
Modal verbs terminology 10
Executive summary 11
Introduction 11
1 Scope 12
2 References 12
2.1 Normative references 12
2.2 Informative references 14
3 Definition of terms, symbols and abbreviations 15
3.1 Terms 15
3.2 Symbols 16
3.3 Abbreviations 16
4 Signature creation process, service decomposition 18
4.1 Signature creation process steps and data elements 18
4.2 Service main components and interfaces 19
4.3 Signature Creation Application 21
4.3.1 Signer's document and hashing 21
4.3.2 DTBS composition and formatting 22
4.3.3 DTBS preparation 22
4.3.4 SDO composer 22
4.4 Server Signing Application 22
4.4.1 Signature creation 22
4.4.1.1 Introduction 22
4.4.1.2 Signature activation 22
4.4.1.3 Signature creation by SCDev 23
5 Authentication and authorization 23
5.1 Introduction 23
5.2 Service authorization 23
5.2.1 Overview 23
5.2.2 HTTP Basic Authentication and HTTP Digest Authentication 24
5.2.3 OAuth 2.0 25
5.2.4 Other authentication mechanisms 27
5.3 Signatures creation authorization 28
5.3.1 Overview 28
5.3.2 Creating a remote signature with a credential protected by signature creation service-managed authorization 28
5.3.3 Create a remote signature with a credential protected by OAuth2 30
5.4 Credential creation authorization 31
5.5 Credential deletion authorization 33
6 Architectures and use cases for server signing 33
6.1 Introduction 33
6.2 Architectures for creating remote signatures with a credential protected by signature creation service-managed authorization 34
6.3 Architectures for creating remote signatures with a credential protected by OAuth2 36
6.4 Architectures for creating remote signatures using the European Digital Identity Wallet 37
6.4.1 Overview 37
6.4.2 EUDIW as authentication and authorization layer for creating remote signatures 38
6.4.3 EUDIW participating in the signature creation process 47
6.4.4 EUDIW registration and authentication to authorization servers 50

DRAFT

6.4.4.1 Introduction 50
6.4.4.2 EUDIW registration 50
6.4.4.3 EUDIW authentication 53
6.4.4.4 Security requirements 53
6.4.4.5 Examples 53
6.4.4.5.1 Dynamic Client Registration request payload (JSON) 53
6.4.4.5.2 Authorization server response 54
6.4.4.5.3 EUDIW authorization request payload (JSON) 54
6.4.4.5.4 PAR Response (from AS to EUDIW) 55
6.4.4.5.5 Authorization Request (from EUDIW to authorization endpoint) 55
6.4.4.5.6 Authorization server redirects back with authorization code 56
6.4.5 DA / SCA registration and authentication to authorization servers 56
6.4.5.1 Introduction 56
6.4.5.2 client registration 56

7 Remote signatures creation service API 58
7.1 Introduction 58
7.2 Service information API 58
7.2.1 Description 58
7.2.2 Request message 58
7.2.2.1 Component for requesting service information 58
7.2.3 Response message 58
7.2.3.1 Component for responding to service information requests 58
7.3 Credentials list API 59
7.3.1 Description 59
7.3.2 Request message 59
7.3.2.1 Component for requesting credentials list 59
7.3.3 Response message 59
7.3.3.1 Component for responding to credentials list requests 59
7.4 Credentials info API 59
7.4.1 Description 59
7.4.2 Request message 59
7.4.2.1 Component for requesting credential information 59
7.4.3 Response message 60
7.4.3.1 Component for responding to credential information requests 60
7.5 Hash(es) signing API 60
7.5.1 Description 60
7.5.2 Request message 60
7.5.2.1 Component for requesting hash(es) signature creation 60
7.5.3 Response message 60
7.5.3.1 Component for responding to hash(es) signature creation requests 60
7.6 Document(s) signing API 60
7.6.1 Description 60
7.6.2 Request message 60
7.6.2.1 Component for requesting documents signature creation 60
7.6.3 Response message 61
7.6.3.1 Component for responding to documents signature creation requests 61
7.7 Signature(s) creation polling API 61
7.7.1 Description 61
7.7.2 Request message 61
7.7.2.1 Component for polling previous signature creation requests 61
7.7.3 Response message 61
7.7.3.1 Component for responding to polling previous signature creation requests 61
7.8 Credentials creation API 61
7.8.1 Description 61
7.8.2 Request message 61
7.8.2.1 Component for requesting credential creation 61
7.8.3 Response message 62
7.8.3.1 Component for responding to credential creation requests 62
7.8.4 Subject distinguished name of the new credential certificate 62
7.9 Credentials deletion API 62
7.9.1 Description 62

DRAFT

7.9.2 Request message ... 62
7.9.2.1 Component for requesting credential creation ... 62
7.9.3 Response message ... 62
7.9.3.1 Component for responding to credential deletion requests ... 62

8 Remote signatures creation service API based on OASIS DSS-X ... 63
8.1 Introduction and General Provisions ... 63
8.2 Service information API ... 63
8.2.1 Description ... 63
8.3 Credentials list API ... 64
8.3.1 Description ... 64
8.4 Credentials info API ... 64
8.4.1 Description ... 64
8.5 Hash(es) signing API ... 64
8.5.1 Description ... 64
8.6 Document(s) signing API ... 64
8.6.1 Description ... 64
8.7 Signature(s) creation polling API ... 65
8.7.1 Description ... 65
8.8 Credentials creation API ... 65
8.8.1 Description ... 65

Annex A (normative): OpenID4VP EUDIW-centric signing flow profile ... 66
A.1 Overview ... 66
A.2 Roles & Components ... 66
A.3 Profile scope & transport ... 67
A.4 Transaction Data Types and Flows ... 67
A.4.1 Overview ... 67
A.4.2 Creating a certificate and signing credential ... 68
A.4.2.1 Purpose 68
A.4.2.2 transaction_data ... 68
A.4.2.3 Transaction flow ... 68
A.4.3 Creating a QES using existing certificate and signing credential ... 70
A.4.3.1 Purpose 70
A.4.3.2 transaction_data ... 70
A.4.3.3 Transaction flow ... 70
A.4.4 Creating a QES on-the-fly ... 71
A.4.4.1 Purpose 71
A.4.4.2 transaction_data ... 71
A.4.4.3 Transaction flow ... 71
A.4.5 List credentials ... 73
A.4.5.1 Purpose 73
A.4.5.2 transaction_data ... 73
A.4.5.3 Transaction flow ... 73
A.5 High-level interaction models ... 74
A.5.1 EUDIW: RP ↔ EUDIW, EUDIW ↔ QTSP ... 74
A.5.2 QTSP: QTSP ↔ EUDIW ... 74
A.6 Request construction (RP → EUDIW) ... 74
A.6.1 Introduction ... 74
A.6.2 Required parameters (OpenID4VP layer) ... 74
A.6.3 dcql_query (selecting an acceptable signing certificate) ... 75
A.6.4 transaction_data (the QES transaction) ... 76
A.7 Response handling (EUDIW → WRP) ... 77
A.7.1 QES response in VP Token ... 77
A.7.2 Out-of-band (HTTP POST to responseURI) ... 78
A.8 EUDIW processing ... 78
A.9 Complementary approval flow ... 79
A.10 Security & privacy requirements ... 79
A.11 Conformance checklist (RP) ... 79
A.12 Examples ... 79
A.12.1 Authorization request (core parameters) ... 79
A.12.2 Example 'qesRequest' (transaction_data before base64url encoding) ... 79

DRAFT

A.12.3 VP Token (inline) - PAdES result ... 80
A.12.4 Out-of-band POST to responseURI ... 81

Annex B (normative): EUDIW QES creation approval profile ... 81
B.1 Overview ... 81
B.2 Roles and architecture ... 81
B.3 Signature authorization attestation ... 82
B.4 Profile scope & transport ... 82
B.5 Profile identifiers ... 82
B.6 Data model for approval ... 83
B.6.1 Introduction ... 83
B.6.2 qesApprovalRequest ... 83
B.6.3 Computing qesApproval (EUDIW) ... 84
B.7 OpenID4VP request (QTSP → Wallet) ... 84
B.8 EUDIW response (EUDIW → QTSP) ... 85
B.8.1 EUDIW validation ... 85
B.8.2 VP Token / DC API response ... 86
B.8.3 Using qesApproval at the AS/SCSP ... 86
B.9 Security & privacy ... 86
B.10 Conformance checklist (SCSP) ... 86
B.11 Working sequence ... 86

Annex C (normative): Specific recommendations for qualified signatures and/or seals creation ... 88
C.1 Introduction ... 88
C.2 Separation of Responsibilities ... 88
C.3 Authentication and Authorization Layer ... 88
C.4 Signature Creation Service ... 88
C.5 Strong Binding Between Consent and DTBS ... 88
C.6 Use of FAPI 2.0 for EUDIW-Based Clients ... 88
C.7 SCS API Security Recommendations ... 88

Annex D (normative): Signing flow profile involving a generic signature initiation component ... 89
D.1 Overview ... 89
D.1.1 General ... 89
D.1.2 Signing flows ... 89
D.2 Roles and components ... 89
D.3 Signature request model ... 90
D.3.1 Requesting signatures ... 90
D.4 Restricting access ... 92
D.4.1 General ... 92
D.4.2 OAuth2 access control ... 92
D.4.3 PID Based Access Control ... 92
D.5 Transport of the requestToSign ... 92
D.5.1 Engagement Protocols ... 93
D.5.1.1 General ... 93
D.5.1.2 Transfer Mechanisms Data Structure ... 93
D.5.1.3 Deep Link ... 94
D.5.1.4 URI ... 95
D.5.1.5 QR Code ... 95
D.5.1.6 NFC Tag ... 95
D.5.2 Transfer mechanisms ... 96
D.5.2.1 Proximity Transfer Mechanisms ... 96
D.5.2.1.1 Introduction ... 96
D.5.2.1.2 NFC ... 96
D.5.2.1.3 BLE ... 97
D.5.2.1.2 Wi-Fi Aware ... 97
D.5.2.2 Remote Transfer Mechanisms ... 98
D.5.2.2.1 HTTP ... 98
D.5.2.2.2 Messaging ... 98
D.6 Security considerations ... 98
D.6.1 Integrity ... 98
D.6.2 Authentication ... 99

DRAFT

D.6.3 Confidentiality 99
D.7 Conformance requirements 99

Annex E (informative): Change history 100

History 101

DRAFT

List of figures

Figure 1: Process Steps and Data Elements in Signature Creation 19

Figure 1a: main components of a remote signature creation service 21

Figure 2: Messages between a driving application and a signature creation service in a common authentication scheme 24

Figure 3: Authorization Code Grant between a driving application and a signature creation service 26

Figure 4: A general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by SCSP-managed authorization 29

Figure 5: A general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server 30

Figure 6: A general flow involving a driving application requesting signing credential creation to a signature creation service with an authorization process managed by an OAuth2 authorization server 32

Figure 7: Remote hashes signature creation by using a signing credential whose usage is protected by authorization mechanisms managed by the signature creation service 34

Figure 8: Remote hashes signature creation by using a signing credential whose usage is protected by a PIN and an OTP provided through an agreed communication channel managed by the signature creation service (i.e. SMS, app, email) 35

Figure 9: Multiple remote signatures creation from a list of hash values by using a signing credential whose usage is protected by PIN 35

Figure 10: Sequence diagram illustrating an authorization code flow between a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server 36

Figure 11: Sequence diagram illustrating an authorization code flow between a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server by a pushed and rich authorization request 37

Figure 12: Architecture with SCA at QTSP and DA at RP 39

Figure 13: Architecture with SCA and DA at different parties and with EUDIW and user agent in the same user device 40

Figure 14: Architecture with SCA and DA at different parties 41

Figure 15: Architecture with DA at RP and SCA at independent party 42

Figure 16: Architecture with DA and SCA at RP 43

Figure 17: Architecture with authorization server at QTSP and with SCA and DA at different parties 44

Figure 18: Generic simplified flow with EUDIW as authentication and authorization layer 45

Figure 19: Remote signature creation with OAuth2 Authorization Code flow and Pushed and Rich Authorization Request with EUDIW as authentication and authorization layer 46

Figure 20: Architecture with EUDIW integrating SCA and SIC and DA running at independent intermediary 47

Figure 21: Architecture with SCA running in the same user device as the EUDIW and DA running at independent intermediary 48

Figure 22: Generic simplified signature model flow with EUDIW integrating SCA and SIC 50

Figure 23: Transaction flow for creating a certificate and signing credential 68

Figure 24: Transaction flow for creating a QES with signing credential 70

DRAFT

Figure 25: Transaction flow for creating a QES on-the-fly 72

Figure 26: Transaction flow for listing of available signing credentials 73

DRAFT

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM®** and the GSM logo are trademarks registered and owned by the GSM Association.

BLUETOOTH® is a trademark registered and owned by Bluetooth SIG, Inc.

Foreword

This draft European Standard (EN) has been produced by ETSI Technical Committee Electronic Signatures and Trust Infrastructures (ESI) and is now submitted for the combined Public Enquiry and Vote phase of the ETSI EN Approval Procedure.

Proposed national transposition dates	
Date of latest announcement of this EN (doa):	3 months after ETSI publication
Date of latest publication of new National Standard or endorsement of this EN (dop/e):	6 months after doa
Date of withdrawal of any conflicting National Standard (dow):	6 months after doa

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

DRAFT

Executive summary

The present document specifies protocols and interfaces for components providing specific functionalities as part of a process for remote digital signatures creation and construction of AdES formats.

Introduction

Standards for digital signatures have generally been developed for a long-time considering solutions tailored to the characteristics of devices such as desktop computers and laptops where all signature processing was done in one system locally to the user. These traditional signature solutions assume that the signer uses smart cards or tokens to create any required digital signatures. Given developments in distributed systems, cloud computing, mobile equipment and related technologies, solutions have been emerging in the last few years where the process of digital signature creation and construction of AdES format is done in a distributed way with different steps of the process carried out by different systems/services that may be controlled by different actors.

The present document aims at supporting electronic signatures and electronic seals, including qualified electronic signatures and qualified electronic seals according to the Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC [i.1] (the eIDAS Regulation).

DRAFT

1 Scope

The present document specifies protocols and interfaces applicable when the process of creating AdES digital signatures as defined by ETSI EN 319 102-1 [i.12] and/or digital signature values, as result of Data To Be Signed Representations signatures, is carried out by a distributed solution comprised of two or more systems/services/components.

The present document is limited to remote server signing, i.e. the signing key is held in a remote shared service.

NOTE: Remote signature creation with local signing, i.e. the signing key is held with the signer's personal device but other steps in the signature creation are carried out by means of networked services, is a possible solution but protocols for such architecture are not covered in the present document.

Finally, the present document does not specify any mandatory binding.

As far as it has been possible and suitable, references to constructs of CSC API [1] JSON specifications and OASIS DSS-X [4] XML specifications are specified in the present document. When this has not been possible the present document specifies new components semantically and also syntactically.

The authorized signer's use of its key for signing requires users to provide multiple proofs of their claimed identity before being granted access to the needed set of resources. The present document considers some ways in which the user identity verification process and the user signatures authorization process can be carried out by the service provider.

2 References

2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found in the ETSI docbox.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents are necessary for the application of the present document.

[1] Cloud Signature Consortium Standard (Version 2.2.0.0): "Architectures and protocols for remote signature applications".

[2] Cloud Signature Consortium Standard (Version 1.0.0): "Data model for remote signature applications".

[3] Cloud Signature Consortium Standard (Version 1.0.0): "CSC data model bindings".

[4] OASIS: "Digital Signature Service Core Protocols, Elements, and Bindings Version 2.0", comprises JSON and XML Bindings.

[5] OASIS: "Digital Signature Service Metadata Version 1.0", comprises JSON and XML Bindings.

[6] EN 419 241-1: "Trustworthy Systems Supporting Server Signing - Part 1: General System Security Requirements", produced by CEN.

[7] IETF RFC 8446: "The Transport Layer Security (TLS) Protocol Version 1.3".

[8] IETF RFC 7515: "JSON Web Signature (JWS)".

[9] W3C®: "XML Signature Syntax and Processing Version 1".

DRAFT

[10] OpenID Foundation: "FAPI 2.0 Security Profile".

[11] IETF RFC 7591: "OAuth 2.0 Dynamic Client Registration Protocol".

[12] IETF RFC 7517: "JSON Web Key (JWK)".

[13] OpenID Connect Relying Party Metadata Choices 1.0.

[14] IETF RFC 7636: "Proof Key for Code Exchange by OAuth Public Clients".

[15] IETF RFC 8414: "OAuth 2.0 Authorization Server Metadata".

[16] IETF RFC 9110: " HTTP Semantics".

[17] IETF RFC 6749: "The OAuth 2.0 Authorization Framework".

[18] IETF RFC 9126: "OAuth 2.0 Pushed Authorization Requests".

[19] IETF RFC 6750: "The OAuth 2.0 Authorization Framework: Bearer Token Usage".

[20] FIDO Alliance: "Client to Authenticator Protocol (CTAP)".

[21] "OpenID Connect Core 1.0 incorporating errata set 2".

[22] OpenID for Verifiable Presentations 1.0.

[23] W3C®: "Digital Credentials, W3C Working Draft", 18 December 2025.

[24] IETF RFC 9207: "OAuth 2.0 Authorization Server Issuer Identification".

[25] IETF RFC 9449: "OAuth 2.0 Demonstrating Proof of Possession (DPoP)".

[26] IETF RFC 8725: "JSON Web Token Best Current Practices".

[27] OASIS: "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) 2.0".

[28] OASIS: "Profiles for the OASIS Security Assertion Markup Language (SAML) 2.0".

[29] IETF RFC 7522: "Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants".

[30] W3C®: "Web Authentication - An API for accessing Public Key Credentials Level 3".

[31] IETF RFC 9101: "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)".

[32] IETF RFC 9728: "OAuth 2.0 Protected Resource Metadata".

[33] IETF RFC 8705: "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens".

[34] IETF RFC 7521: "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants".

[35] IETF RFC 9396: "OAuth 2.0 Rich Authorization Requests".

[36] ETSI TS 119 476-3: "Electronic Signatures and Trust Infrastructures (ESI); Selective disclosure and zero-knowledge proofs applied to Electronic Attestation of Attributes; Part 3: EUDI Wallet Unit Attestation (WUA)".

[37] ETSI EN 319 411-1: "Policy and security requirements for Trust Service Providers issuing certificates; Part 1: General requirements".

[32] IETF RFC 9112: "HTTP/1.1".

[33] IETF RFC 8785: "JSON Canonicalization Scheme (JCS)".

DRAFT

[40] IETF RFC 8949: "Concise Binary Object Representation (CBOR)".

[41] IETF RFC 4648: "The Base16, Base32, and Base64 Data Encodings".

[42] NFC Forum Type 4 Tag Specification (Version 1.2)

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding but are not required for conformance to the present document.

[i.1] Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC, amended by Regulation (EU) 2024/1183.

[i.2] ETSI TS 119 312: "Electronic Signatures and Trust Infrastructures (ESI); Cryptographic Suites".

[i.3] ETSI TR 119 001: "Electronic Signatures and Infrastructures (ESI); The framework for standardization of signatures; Definitions and abbreviations".

[i.4] ETSI TS 103 171: "Electronic Signatures and Infrastructures (ESI); XAdES Baseline Profile".

[i.5] ETSI TS 103 172: "Electronic Signatures and Infrastructures (ESI); PAdES Baseline Profile".

[i.6] ETSI TS 119 431-1: "Electronic Signatures and Trust Infrastructures (ESI); Policy and security requirements for trust service providers; Part 1: TSP services operating a remote QSCD / SCDev".

[i.7] ETSI TS 119 431-2: "Electronic signatures and infrastructures (ESI); Policy and security requirements for trust service providers; Part 2: TSP service components supporting AdES digital signature creation".

[i.8] IETF RFC 7662: "OAuth 2.0 Token Introspection".

[i.9] IETF RFC 8259: "The JavaScript Object Notation (JSON) Data Interchange Format".

[i.10] IETF RFC 7617: "The 'Basic' HTTP Authentication Scheme".

[i.11] Architecture and Reference Framework.

[i.12] ETSI EN 319 102-1: "Electronic Signatures and Trust Infrastructures (ESI); Procedures for Creation and Validation of AdES Digital Signatures; Part 1: Creation and Validation".

[i.13] IETF draft-ietf-oauth-sd-jwt-vc-15: "SD-JWT-based Verifiable Digital Credentials (SD-JWT VC)".

[i.14] OpenID for Verifiable Credential Issuance 1.0.

[i.15] IETF RFC 7616: "HTTP Digest Access Authentication".

[i.16] IETF RFC 9068: "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens".

[i.17] IETF RFC 7519: "JSON Web Token (JWT)".

[i.18] IETF RFC 7523: "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants".

[i.19] IETF RFC 8017: "PKCS #1: RSA Cryptography Specifications Version 2.2".

[i.20] ETSI TS 103 173: "Electronic Signatures and Infrastructures (ESI); CAdES Baseline Profile".

DRAFT

[i.21] ISO/IEC 18013-5: "Personal identification — ISO-compliant driving licence — Part 5: Mobile driving licence (mDL) application".

[i.22] EN 419 241-2: "Trustworthy Systems Supporting Server Signing - Part 2: Protection profile for QSCD for Server Signing.

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the terms given in ETSI TR 119 001 [i.3] and the following apply:

AdES (digital) signature: digital signature that is either a CAdES signature, or a PAdES signature or a XAdES signature

client application: application that requests authorization from an Authorization Server on behalf of a resource owner (signer in the present document), in order to access protected resources at a Resource Server (signer's private key usage in the present document)

digital signature: data appended to, or a cryptographic transformation of a data unit that allows a recipient of the data unit to prove the source and integrity of the data unit and protect against forgery (e.g. by the recipient)

digital signature value: result of the cryptographic transformation of a data unit that allows a recipient of the data unit to prove the source and integrity of the data unit and protect against forgery (e.g. by the recipient)

driving application: application that uses a signature creation system to create a signature, or a signature validation application to validate digital signatures, or a signature augmentation application to augment digital signatures.

NOTE: in the present document, the driving application is the component initiating or orchestrating remote signature creation requests.

European Digital Identity Wallet / EUDI Wallet / EUDIW: electronic identification means under Regulation (EU) 2024/1183 enabling users to store, manage, validate and present person identification data and electronic attestations of attributes, and to create or support qualified electronic signatures or seals where applicable.

FAPI: an OpenID Foundation working group, where FAPI was previously known as the Financial-grade API but where consensus was reached within the working group to update the name to just FAPI to reflect that the specification was appropriate for many high-value use-cases requiring a more secure model beyond just financial services

intermediary: entity acting on behalf of a relying party for the purpose of transmitting or facilitating requests to the EUDIW

relying party: entity that relies upon the output of a trust service, in particular upon a certificate, an electronic signature/seal, or the result of its validation; in a remote-signature architecture, it can be the entity that requests remote signature creation and then relies on the result and/or the recipient that verifies and then relies upon the AdES/QES generated by the remote signing service

remote signature creation device: signature creation device used remotely from signer perspective and providing control of signing operation on the signer's behalf

remote signing server: server-side component hosting or invoking the SSA and possibly exposing authorization endpoints;

NOTE: It doesn’t not necessarily include the Authorization Server.

secure shell protocol: cryptographic network protocol for operating network services securely over an unsecured network

server signing application: application using a remote signature creation device to create a digital signature value on behalf of a signer

DRAFT

NOTE: The term Server Signing Application is used consistently with EN 419 241-1 [6] and EN 419 241-2 [i.22].

service authorization: process of verifying and enforcing that a client (such as an application, user, or system component) has the necessary permissions and rights to access and invoke specific operations or endpoints exposed by a service API, according to predefined policies or access control rules

signature activation data: set of data used to control with a high level of confidence a given signature operation, performed by a cryptographic module on behalf of the signer, that is under sole control of the signer

signature activation module: configured software that uses the SAD in order to guarantee with a high level of confidence that the signing keys are used under sole control of the signer

NOTE: As defined in EN 419 241-1 [6].

signature creation application: application within the signature creation system that creates the AdES digital signature and relies on the SCDev to create a digital signature value

NOTE: The SCDev can be managed by the SSA.

signature creation device: configured software or hardware used to implement the signature creation data and to create a digital signature value

signature creation policy: set of signature creation constraints processed or to be processed by the SCASC or the SSA

signature creation service: TSP service implementing a signature creation application and/or a server signing application

signature creation service provider: service provider offering a signature creation service

signing credential: set of the signing key and the corresponding signing certificate

user agent: software component acting on behalf of the resource owner (the signer in the context of the present document) that interacts with the authorization server and the driving application

NOTE: Typically, this is the web browser or application through which the user accesses the driving application. The user agent is responsible for presenting the authorization interface to the user and forwarding requests and responses between the driving application and the authorization server.

wallet-relying party : relying party requesting data, attestations, transaction approval, or signature creation through an EUDI Wallet.

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

API	Application Program Interface
AS	Authorization Server
AS/SAM	Authorization Server / Signature Activation Module
BLE	Bluetooth Low Energy
CA	Certification Authority
CBOR	Concise Binary Object Representation
CEN	European Committee for Standardization
CMS	Cryptographic Message Syntax
CR	Change Request
CRL	Certificate Revocation List
CSC	Cloud Signature Consortium
CSC API	CSC Standard: "Architectures and protocols for remote signature applications"
CSC DM	CSC Standard: "Data model for remote signature applications"

DRAFT

CTAP	Client to Authenticator Protocol
DA	Driving Application
DC	Digital Credential
DCQL	Digital Credentials Query Language
DPoP	Demonstrating Proof-of-Possession
DSS-X	Digital Signature Services eXtended
DSV	Digital Signature Value
DTBS	Data To Be Signed
DTBSF	Data To Be Signed Formatted
DTBSR	Data To Be Signed Representation

NOTE: DTBSR denotes the representation of the complete Data To Be Signed, including the document representation and signed attributes/properties as applicable.

ECDSA	Elliptic Curve Digital Signature Algorithm
eIDAS	electronic Identification And trust Services
EN	European Norm
EUDI	European Digital Identity
EUDIW	European Digital Identity Wallet
FIDO2	Fast IDentity Online 2
GATT	Generic ATTribute Profile
HMAC	Hashed Message Authentication Code
HTTP	Hyper Text Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ISO	International Organization for Standardization
JAR	JWT-Secured Authorization Request
JSON	Java Script Object Notation
JWS	JSON Web Signature
JWT	JSON Web Token
LT	Long Term
LTA	Long Term Archival
mTLS	mutual Transport Layer Security
NAN	Neighbor Awareness Networking
NFC	Near Field Communication
NDEF	NFC Data Exchange Format
OASIS	Organization for the Advancement of Structured Information Standards
OCSP	Online Certificate Status Protocol
OID	Object IDentifier
OIDC	OpenID Connect
OpenID	OpenID foundation
OpenID4VP	OpenID for Verifiable Presentations
OTP	One-Time Password
PAR	Pushed Authorization Requests
PID	Person Identification Data
PIN	Personal Identification Number
PKCE	Proof Key for Code Exchange
PKI	Public Key Infrastructure
PMK	Pairwise Master Key
QES	Qualified Electronic Signature
QR code	Quick Response code

NOTE: Two-dimensional machine-readable optical barcode that contains information about the item to which it is attached.

QSCD	Qualified electronic Signature Creation Device
QTSP	Qualified Trust Service Provider
RAR	Rich Authorization Request
RP	Relying Party
RSA	Rivest, Shamir, & Adleman
RSSP	Remote Signing Server Provider
SAD	Signature Activation Data
SAM	Signature Activation Module
SAML	Security Access Markup Language

DRAFT

SCA Signature Creation Application
SCAL Sole Control Assurance Level
SCAL1 Sole Control Assurance Level 1

NOTE: As defined in EN 419 241-1 [6].

SCAL2 Sole Control Assurance Level 2

NOTE: As defined in EN 419 241-1 [6].

SCDev Signature Creation Device
SCS Signature Creation Service
SCSP Signature Creation Service Provider
SD Signer's Document
SD-JWT VC Selective Disclosure based JSON Web Token Verifiable Credentials
SDO Signed Data Object
SDOC Signed Data Object Composer
SDR Signer's Document Representation

NOTE: SDR denotes a representation of the signer's document before incorporation into the Data To Be Signed as defined in EN 319 102-1 [i.12].

SHA Secure Hash Algorithm
SIC Signature Interaction Component

NOTE: As defined in EN 419 241-1 [6].

SMS Short Message Service
SODR Signer's Original Document Representation (i.e. hash of signer's original document).

NOTE: As defined in CSC API [1].

SRI SubResource Integrity
SSA Server Signing Application
SSH Secure Shell Protocol
TLS Transport Layer Security
TR Technical Report
TSP Trust Service Provider
URI Uniform Resource Identifier
URL Uniform Resource Locator
URN Uniform Resource Name
UTF-8 Unicode Transformation Format - 8 bit
VP Verifiable Presentation
WIA Wallet Instance Attestation
WRP Wallet-Relying Party
WUA Wallet Unit Attestation
XML eXtended Markup Language
XSD XML Schema Definition

4 Signature creation process, service decomposition

4.1 Signature creation process steps and data elements

Figure 1 (derived from ETSI EN 319 102-1 [i.12], clause 4.2.1) shows the various steps and the related data elements for a signature creation process. For remote signature creation, different steps of this process are carried out according to a decomposition into several components, which will have access to or make available the corresponding data elements. The process illustrated in the figure below is limited to the buildings blocks and information needed for creating a signature without specifying possible solutions for tasks such as signer authentication, authorization to the signing key usage or signing certificate availability. The signature activation module in the tamper protected area is needed only when the Signature Creation Service (SCS) complies to the Sole Control Assurance Level 2 (SCAL2) signature activation mechanism as defined in EN 419 241-1 [6].

DRAFT

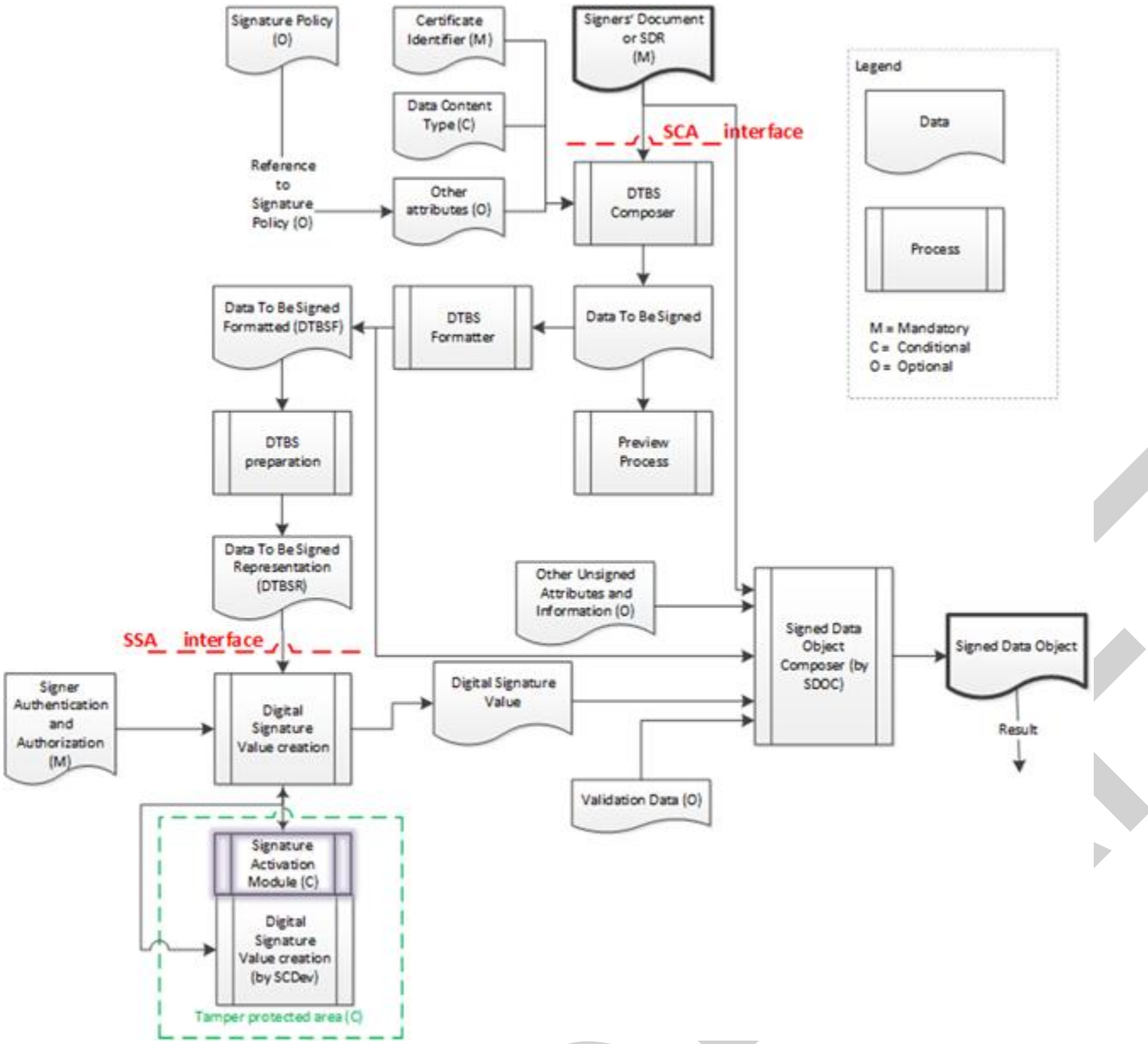

Figure 1: Process Steps and Data Elements in Signature Creation

4.2 Service main components and interfaces

The above process points out scenarios where the AdES and/or Digital Signature Value (DSV) are created using a signing key held within a cryptographic security module named Signature Creation Device (SCDev) operated by a Signature Creation Service Provider (SCSP).

Based on the different types of data managed in requests and responses, two main components can be identified in the above schema providing different interfaces for signing management: the Server Signing Application Service Component (SSA) and the Signature Creation Application Service Component (SCA) defined below.

The SSA is the component supporting digital signature values creation. The SSA can interact with the SCDev holding the signer's private key. When the SSA uses the SCDev, the authorized signer controls the signing key with a certain level of confidence.

The SSA interface has the Data To Be Signed Representation (DTBSR) and other parameters as main input and the digital signature value as main output.

The SCA is the component supporting AdES digital signature creation and carries out several specific parts of the signature creation process. The SCA interacts with the SSA for requesting digital signature values creation.

The SCA interface has the document(s) to be signed (SD) or its (their) representation (SDR) and other parameters as main input and the signed document(s) or the digital signature(s) as main output.

SCS denotes a TSP service implementing a Signature Creation Application (SCA) and/or a Server Signing Application (SSA).

Some variants of these interfaces are possible depending on the functional split between the SCS and the signer's local system.

The next figure identifies the main logical components involved in a remote signature creation service. The Driving Application initiates or orchestrates the signature creation request. The Signer Interaction Component supports signer-facing authentication, transaction review and approval. The Signature Creation Application prepares the signature according to the requested signature format and invokes the Server Signing Application to obtain the digital signature value. The Signature Activation Module enforces authorization and activation requirements before the Signature Creation Device uses the signing key. Credential and certificate management functions may be invoked when a credential has to be created, selected, updated or deleted as part of the transaction.

DRAFT

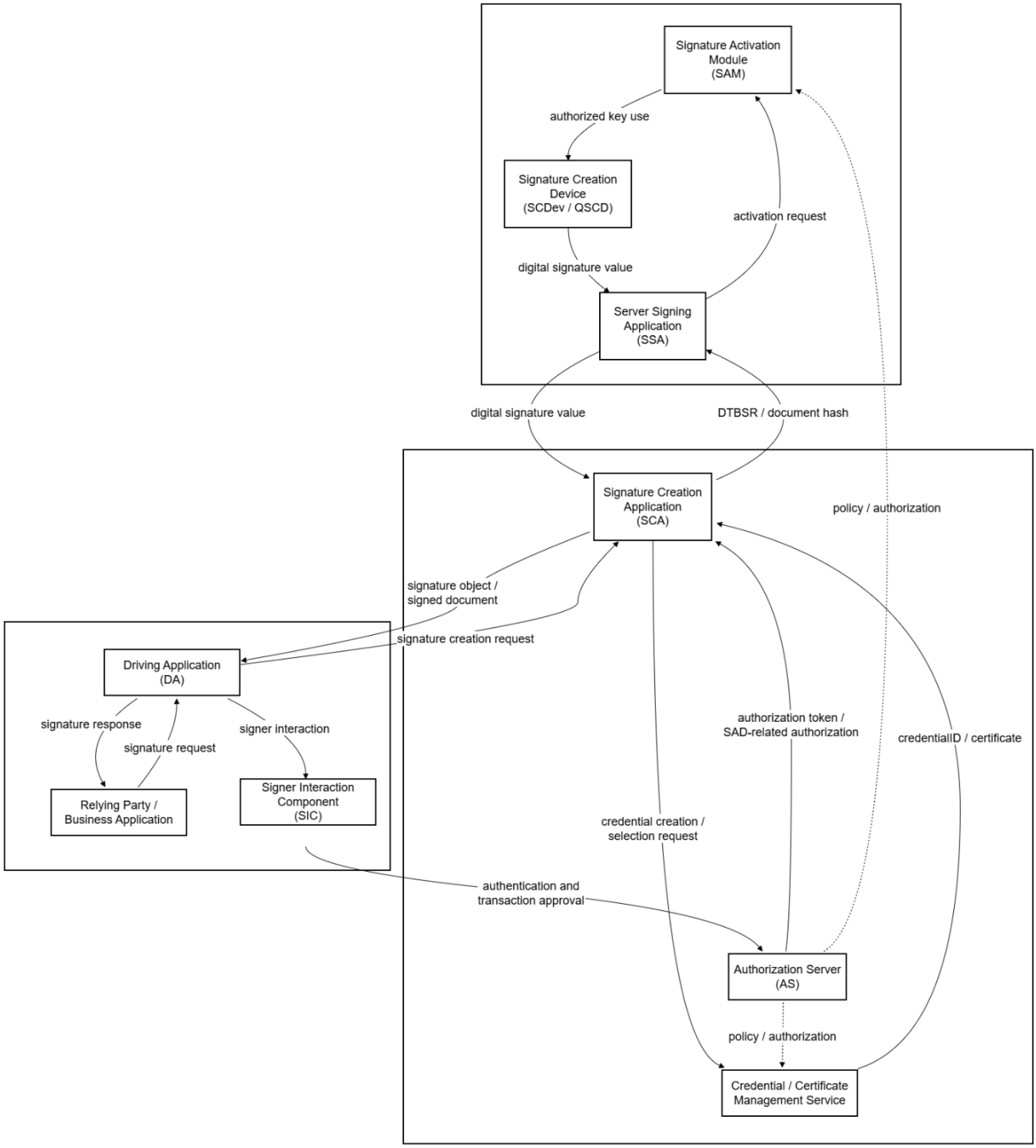

Figure 2a: main components of a remote signature creation service

The following clauses specify main information objects and processes in SCA and SSA.

4.3 Signature Creation Application

4.3.1 Signer's document and hashing

The signature creation process starts with the signer's document (SD), which is to be signed. The SD is represented (SDR) by a hash value in the Data To Be Signed (DTBS). The following observations are made:

- The creation of the SDR (the hashing) can be done where the SD is stored or by the SCA. In the former case, the SDR is transferred to the SCA while in the latter case, the SD is transferred to the SCA.
- The SD is part of the final Signed Data Object (SDO). Part of the Signed Data Object Composer (SDOC) function (building of the final AdES format) is to relate the digital signature value to the SD.

An important design decision for remote signature creation services is where the SD, and thus its content, needs to be available. Making available only the SDR limits threats to confidentiality but may result in limitations in the functionality of the remote signature creation solution (i.e. when enveloping or enveloped signatures need to be created, or when visual representation of the signature needs to be included).

4.3.2 DTBS composition and formatting

In the two processes of DTBS composition and formatting, which in the context of the present document are considered together, the SDR (hash of the document to be signed or hash of a formatted version of the document to be signed) and hashes of signed attributes/properties are assembled into the Data To Be Signed Formatted (DTBSF). In addition to a certificate identifier (hash of signing certificate, possibly also of further certificates in a certificate chain) as indicated in the figure, further signed attributes are required or allowed by the ETSI standard signature formats (C/X/J/PAdES). For example all baseline CAdES and XAdES variants require the presence of the signed attributes "document type" (of SD) and "claimed signing time".

The signed attributes or their hash values, whose presence is needed in the DTBS, are available to the SCA when the DTBSF is created by the SCA.

4.3.3 DTBS preparation

This step consists of creating the DTBSR from the DTBSF. The SCA prepares the entire DTBSF, calculates the hash, and sends the hash value (DTBSR) as input to an SSA.

4.3.4 SDO composer

As the final step, the SDO (the AdES format) is constructed. This consists of combining the digital signature value with other parameters into the requested format. Depending on the format, the digital signature made available for the SD is named:

- Enveloped: The signature is added to the SD (e.g. PAdES signature).
- Enveloping: The signature wraps the SD (e.g. certain CAdES formats).
- Detached: The signature is a separate object linked to the SD.

The SDO composing is done by a separate service instance or integrated with other functions in the SCA.

4.4 Server Signing Application

4.4.1 Signature creation

4.4.1.1 Introduction

The purpose of the signature creation process is to take the DTBSR and create a digital signature value under the control of the signer. In the context of the present document, the creation of the digital signature value is managed by an SSA that uses a signing key, held within a cryptographic security module (SCDev), that the signers can activate by means of a secure authorization and activation process (i.e. by applying the signature activation protocol specified in EN 419 241-1 [6]).

4.4.1.2 Signature activation

The SSA uses a remote SCDev to generate, maintain and use the signing keys under the control of their authorized signers. The authorized signer remotely controls the signing key with a certain level of confidence possibly by means of

DRAFT

the Signature Activation Module (SAM). The SAM is a software component using the Signature Activation Data (SAD) for enforcing sole control in remote signatures creation from the signer confirming its authorization to activate its signing key for the purpose of signing the DTBSR. This process ensures confidence that the signing keys are under the sole control of the signer.

Two different levels of confidence of the control of the signing key, as defined in EN 419 241-1 [6], are considered in the present document:

- Sole Control Assurance Level 1 (SCAL1):
  - The signing keys are used, with a low level of confidence, under the sole control of the signer.
  - The authorized signer's use of its key for signing is enforced by the SSA which authenticates the signer. The activation of the signing key can remain for a given period and/or for a given number of signatures.

NOTE: It is not expected that such implementations would meet the requirements of sole control as it would be expected for a stand-alone QSCD as defined in the eIDAS [i.1] Regulation.

- Sole Control Assurance Level 2 (SCAL2):
  - The signing keys are used, with a high level of confidence, under the sole control of the signer.
  - The authorized signer's use of its key for signing is enforced by the signature activation module by means of signature activation data provided, by the signer, using a signature activation protocol, in order to enable the use of the corresponding signing key to sign specific documents. The signer may interact with a signer interaction component to review the transaction, authenticate, and approve the creation of one or more signatures. The SIC may be implemented by the signature creation service, by a wallet, by a trusted application, or by another component depending on the architecture.

4.4.1.3 Signature creation by SCDev

The signature creation process is performed by the SCDev. In the context of the present document, only architectures where the signature creation process is carried out by a remote SCDev are considered. According to the above sole control assurance levels the signing key can be used to generate the digital signature value creation after a successful signer authentication by the SSA (SCAL1) or after a successful SAD verification by the SAM.

5 Authentication and authorization

5.1 Introduction

The present document recommends protecting remote signature creation services from unauthorized access. Any driving application shall be authenticated when invoking a signature creation service. In order to create signatures an authorization from the user who owns and controls the signing key is required. The user may authorize the creation of one or more signatures with a signing certificate of a certain type possibly using short-lived signing keys and certificates to be created on-the-fly. The driving application, responsible for requesting signature creation, in the authentication and authorization flows described in the following clauses assumes also the client role implementing the necessary client authentication and authorization procedures.

5.2 Service authorization

5.2.1 Overview

Various types of authorization mechanisms may be supported, and the driving application shall adopt any of those available from the signature creation service.

DRAFT

5.2.2 HTTP Basic Authentication and HTTP Digest Authentication

HTTP defines a general challenge-response authentication framework. When a client requests a protected resource, the server replies with the HTTP status code 401 "Unauthorized", defined in clause 15.5.2 of IETF RFC 9110 [16], and a WWW-Authenticate header advertising one or more authentication challenges; the client retries the request including credentials in the Authorization header. Proxy variants use the HTTP status code 407 "Proxy Authentication Required", defined in clause 15.5.8 of IETF RFC 9110 [16].

The driving application may obtain an access token for service authorization directly from the signature creation service using HTTP Basic Authentication, as defined in IETF RFC 9110 [16], using authentication factors (like user and password) assigned to the signer. These authentication factors will be passed directly in the HTTP header as an authorization grant to obtain an access token to use for the subsequent requests within the same session. HTTP Basic Authentication is a simple scheme that sends credentials as base64 (username:password) in the Authorization header. Base64 is an encoding, not encryption, therefore TLS (HTTPS) should be used to protect credentials in transit.

The challenge and response flow works like this:

- The signature creation service responds to the driving application with the HTTP status code 401 "Unauthorized", defined in clause 15.5.2 of IETF RFC 9110 [16], and provides information on how to authorize with a WWW-Authenticate response header containing at least one challenge.
- A client that wants to authenticate itself with the signature creation service can do so by including an Authorization request header with the authentication factors.
- Usually, a driving application will present a password prompt to the signer and will then issue the request including the correct Authorization header.

Figure 3: Messages between a driving application and a signature creation service in a common authentication scheme

The "Basic" authentication scheme used in the diagram above sends the authentication factors encoded but not encrypted. This would be completely insecure unless the exchange is over a secure connection (HTTPS/TLS).

HTTP Basic Authentication [i.10] is an unsafe mechanism, and therefore it should not be used, especially by a driving application running as a service. It should only be used when there is a high degree of trust between the user and the driving application and when other authorization types are not available. This mechanism may also be deprecated in future releases of the present document. HTTP Basic Authentication should not be employed in processes implementing qualified electronic signature or seal creation, as it does not meet the required security assurance level.

The general message flow above is the same for most authentication schemes. The actual information in the headers and the way it is encoded may change. HTTP Digest authentication [i.15] improves on HTTP Basic Authentication by avoiding transmission of plaintext passwords. The server issues a challenge containing parameters such as realm, nonce, algorithm, and qop (quality of protection values supported by the server). The client responds with a hash that proves knowledge of the password, incorporating method and URI, and freshness parameters (nonce, cnonce, nonce count).

The above HTTP Basic Authentication scheme is implemented in CSC API [1] by invoking the API:

- *auth/login* passing the username and password assigned to the signer directly in the HTTP header as an authorization grant.
- *auth/revoke* passing the token, that was obtained from the remote service, to be revoked.

By invoking the *auth/login* API the driving application can obtain an access token for service authorization from the remote service using HTTP Basic Authentication or HTTP Digest authentication, as defined in IETF RFC 9110 [16].

The *auth/login* API identified in CSC API [1], clause 11.2 shall apply.

The message for requesting the access token for service authorization shall contain the components defined in CSC API [1], clause 11.2 section Input.

The message for returning the access token for service authorization shall contain the components defined in CSC API [1], clause 11.2 section Output.

By invoking the *auth/revoke* API the driving application can invalidate access or refresh token obtained from the remote service or an associated authorization server so that any further access by reusing the token itself is prevented.

The *auth/revoke* API identified in CSC API [1], clause 11.3 shall apply.

The message for invalidating an access or refresh token for service authorization shall contain the components defined in CSC API [1], clause 11.3 section Input.

The *auth/revoke* API has no output values, the HTTP status code 204 "No Content", defined in clause 15.3.5 of IETF RFC 9110 [16], is provided in response.

5.2.3 OAuth 2.0

When the authentication is under the control of the signature creation service, the signature creation service may adopt an indirect way of authorizing access to the API by using OAuth 2.0, defined in IETF RFC 6749 [17], that is primarily an authorization framework, but that can also be leveraged for authentication when interacting with a signature creation service API. OAuth 2.0 ensures that only authorized applications and users can access sensitive signing operations without exposing credentials directly. A common approach is to combine OAuth 2.0 with OpenID Connect (OIDC) [21] to handle user authentication and, optionally, authorization delegation. OAuth 2.0 defines a framework for token-based authorization, while OIDC adds an identity layer for authentication. In this flow, the user authenticates through an Identity Provider, which issues an access token for authorization and, in the case of OIDC, an identity token (JWT [i.4] may be used) that conveys identity information. These tokens can then be presented to the signature creation service. The Identity Provider can be an internal authentication service or an external one. A driving application will include a token in its API calls to the signature creation service, representing the user's identity and permissions. The signature creation service validates this token to ensure the driving application is authorized and allowed to access the requested resource.

Main steps:

1) Client Registration

   The driving application (client) registers with the remote signing server's Authorization Server to obtain a *client_id* and possibly a *client_secret*. Where dynamic registration is used, the client registers with the authorization server according to IETF RFC 7591 [11].

2) Authorization Flow

   The user (signatory) is redirected to the signature creation service's Authorization Server.

DRAFT

The user authenticates using their identity credentials.

The user grants consent for the driving application to access signature creation service API.

3) Token Issuance

After successful authentication and consent, the Authorization Server issues:

- Access Token, used by the driving application to invoke the signature creation service API.
- ID Token (optional, if OpenID Connect is used), that provides user identity information for authentication purposes.

4) API Access

The driving application includes the Access Token in the Authorization header:

- Authorization: Bearer <access_token>

The signature creation service validates the token before allowing operations like listing credentials or retrieving certificates.

A common token format is JSON Web Tokens (JWT), which are digitally signed tokens that can carry signer identity (subject), issued-at time, expiry, and roles or scopes as claims. They are convenient in services invocation because they are stateless - any service can verify the token with a public key without needing to call back to the issuer for each request. It shall be ensured that the JWT is signed with strong algorithms and that remote signature creation services trust the issuer's public key securely. Using JWT scopes or claims, a fine-grained authorization can be implemented.

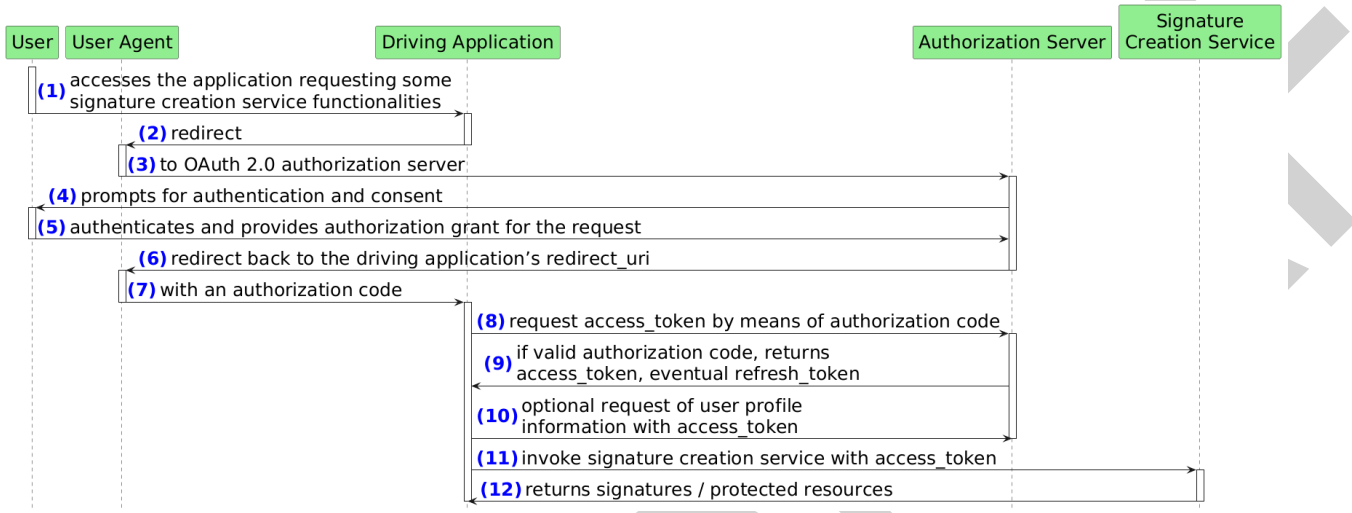

Figure 4: Authorization Code Grant between a driving application and a signature creation service

In the above diagram, if the user grants authorization, the driving application will obtain from the authorization server an access token that can be used for authorizing requests to signature creation service by using an Authorization header with *Bearer* type followed by that access token.

If the user does not grant authorization, the authorization server will return an error message and no access to authenticated API of the signature creation service will be possible.

NOTE: In order to ensure security and integrity of authorization requests, IETF RFC 9126 [18] recommends invoking first pushed_authorize to securely transmit the authorization request, immutable, and protected from tampering or leakage and then the authorize endpoint that will only need the request_uri lightweight reference, minimizing risk and mitigating injection attacks.

The above scheme is implemented in CSC API [1] by invoking the API:

- *oauth2/authorize* (and *oauth2/pushed_authorize* if needed), that manages authorization requests using the Authorization Code flow as described in Section 4.1 of IETF RFC 6749 [17].

- *oauth2/token* to obtain an OAuth 2.0 bearer access token [19] from the authorization server by passing either the client credentials (id and secret) pre-assigned by the authorization server to the driving application, or the authorization code or refresh token returned by the authorization server after a successful user authentication.

- *oauth2/revoke* passing the access or refresh token, that was obtained from the authorization server, to be revoked.

By invoking the *oauth2/authorize* API the driving application can obtain an authorization code for service authorization from the authorization server of the remote signing service.

The *oauth2/authorize* API identified in CSC API [1], clause 8.2.2 shall apply.

The message for requesting the authorization code for service authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Input considering what specified in section Service scope.

The message for returning the authorization code for service authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Output.

By invoking the *oauth2/pushed_authorize* API the driving application can push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and obtain a request URI to be used as reference to the data in a subsequent call to the authorization endpoint (*oauth2/authorize*) of the authorization server.

The *oauth2/pushed_authorize* API identified in CSC API [1], clause 8.2.3 shall apply.

The message for requesting the request URI that will be used as reference to the data in a subsequent call to the authorization endpoint shall contain the components defined in CSC API [1], clause 8.2.2 section Input apart from the request_uri parameter.

The message for returning the request URI shall contain the components defined in IETF RFC 9126 [18], clause 2.2.

By invoking the *oauth2/token* API the driving application can obtain an access token for service authorization from the authorization server of the signature creation service.

The *oauth2/token* API identified in CSC API [1], clause 8.2.4 shall apply.

The message for requesting the access token for service authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Input.

The message for returning the access token for service authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Output.

By invoking the *oauth2/revoke* API the driving application can invalidate access or refresh token obtained from the authorization server of the signature creation service so that any further access by reusing the token itself is prevented.

The *oauth2/revoke* API identified in CSC API [1], clause 8.2.5 shall apply.

The message for invalidating an access or refresh token for service authorization shall contain the components defined in CSC API [1], clause 8.2.5 section Input.

The *oauth2/revoke* API has no output values, the HTTP status code 204 "No Content", defined in clause 15.3.5 of IETF RFC 9110 [16], is provided in response.

5.2.4 Other authentication mechanisms

The underlying communication between the driving application and the signature creation service may ensure access control in some other different ways than OAuth 2.0 or HTTP Basic or Digest Authentication. The following alternatives can be considered for authenticating clients to a signature creation service:

- Mutual TLS (mTLS) with client certificates according to RFC 8705 [33]. The driving application presents an X.509 certificate during the TLS handshake. The signature creation service validates it (chain, revocation, policy) and authorizes based on the certificate's identity/attributes.

- HMAC-Signed Requests. Each driving application has an API key + secret. It computes HMAC (secret, canonical_request) and sends signature + metadata (timestamp, nonce).

DRAFT

- Signed JWT Assertions according to IETF RFC 7523 [i.18]. Driving applications self-issue a signed assertion (JWT) proving identity and authorization claims. The signature creation service validates signature (against client public key or its own CA) and standard claims (aud, exp, nbf, jti). This mechanism is a profile of OAuth 2.0 Assertion Framework, defined in IETF RFC 7521 [34].

- SAML 2.0 Assertions used within the SAML authentication framework (see Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) 2.0 [27] and Profiles for the OASIS Security Assertion Markup Language (SAML) 2.0 [28]) or alternatively within the OAuth2.0 framework (see IETF RFC 6749 [17]) like Bearer (or Holder-of-Key) (see IETF RFC 7522 [29] a profile of OAuth 2.0 Assertion Framework, defined in IETF RFC 7521.). Driving applications obtain a SAML assertion from an Identity Provider and present it to the signature creation service, with Holder-of-Key, the driving applications prove possession of a key (stronger than bearer).

- SSH Mutual Authentication via Reverse Tunnel. Establishing an SSH tunnel with mutual public-key auth.

- FIDO2/Passkey phishing-resistant strong multi-factor authentication (see Web Authentication - An API for accessing Public Key Credentials Level 3 [30] and CTAP [20]).

The above list is not exhaustive, further different solutions can be operational too.

5.3 Signatures creation authorization

5.3.1 Overview

Requesting signatures creation to a signature creation service requires an authorization from the user who owns and controls the signing key to be used for signatures creation. The user might authorize signatures creation with an already existing signing key or with a signing key to be created on-the-fly. Two different levels of confidence of the control of the signing key, as defined in EN 419 241-1 [6], are considered in the present document:

- SCAL1 - Sole Control Assurance Level 1 that ensures that the signature operation is performed on behalf of the signer, but with limited confidence. Typically relies on basic authentication mechanisms (e.g. username/password or PIN). There is no cryptographic binding between the Signature Activation Data (SAD) and the data to be signed.

- SCAL2 - Sole Control Assurance Level 2 that requires stronger authentication (e.g. multi-factor, cryptographic challenge-response) and cryptographic binding between the SAD and the data to be signed. This means the SAD is linked to the specific document or hash being signed, reducing the risk of misuse.

The signing credential use by the driving application may be protected directly by the Signature Creation Service Provider (SCSP) or by means of a framework, like OAuth2, that allows the driving applications to obtain access to the user's signing credential without needing the user's authentication data.

In the following clauses the sequence diagrams of some of the most common operations needed to request and authorize a remote signature creation are described, with particular reference to the possible use of the European Digital Identity Wallet (EUDIW), designed to operate at a Level of Assurance (LoA) High, as a signer authentication factor.

5.3.2 Creating a remote signature with a credential protected by signature creation service-managed authorization

NOTE: Explicit authorization requires the driving application to collect authentication factors (e.g. PIN, OTP) within its own environment. This introduces complexity in ensuring secure factor collection and storage. This mechanism should not be suitable to satisfy SCAL2 requirements.

Below is a sequence diagram illustrating a general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by SCSP-managed authorization.

DRAFT

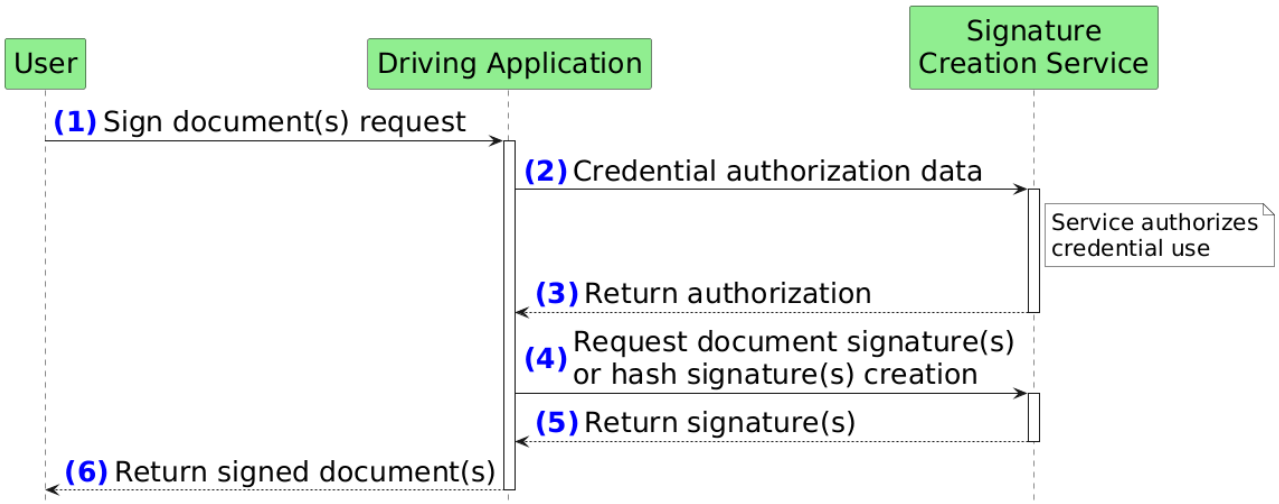

Figure 5: A general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by SCSP-managed authorization

The above scheme is implemented in CSC API by invoking the API

- *credentials/authorize*, passing, in addition to the values identifying the signing credential and the DTBSR, the authentication objects needed to complete signatures creation authorization; CSC API specification [1] in section Authentication objects of clause 11.8 defines a generic way to manage access control to signing credentials associating each signing credential with a set of authentication object types and an access rule describing the precondition to authorize the signing credential access and usage; *credentials/authorizeCheck* and *credentials/extendTransaction* may be invoked for retrieving authorization data and for extending the validity of a multi signature transaction authorization data; when the authorization process is based on a challenge response protocol, *credentials/getChallenge* may be invoked to request a challenge from the signature creation service

- *signatures/signHash* or *signatures/signDoc* to obtain the requested signature objects or signed documents as specified in clause 11.13 and 11.14 of CSC API.

By invoking the *credentials/authorize* API the driving application can obtain the authorization data for making use of a signing credential to create signatures, according to the authorization mechanisms associated to the credential itself.

The *credentials/authorize* API identified in CSC API [1], clause 11.8 shall apply.

The authentication objects and corresponding values collected from the user shall be included in the components defined in CSC API [1], clause 11.8 section Input for creating the request message.

The message for returning the authorization data for signing credential usage shall contain the components defined in CSC API [1], clause 11.8 section Output.

By invoking the *credentials/authorizeCheck* API the driving application can poll the authorization state of a previous *credentials/authorize* request terminated with HTTP status code 202 "Accepted", defined in clause 15.3.3 of IETF RFC 9110 [16], representing that some authorization process was still underway.

The *credentials/authorizeCheck* API identified in CSC API [1], clause 11.9 shall apply.

The message for polling the authorization state of a previous *credentials/authorize* request terminated with HTTP status code 202 "Accepted", defined in clause 15.3.3 of IETF RFC 9110 [16], shall contain the components defined in CSC API [1], clause 11.9 section Input.

The message for returning the authorization data for signing credential usage shall contain the components defined in CSC API [1], clause 11.9 section Output.

By invoking the *credentials/extendTransaction* API the driving application can extend the validity of a multi-signature transaction authorization by obtaining new authorization data so that, i.e. the API method *signatures/signHash* can be invoked multiple times with a single credential authorization event.

DRAFT

The *credentials/extendTransaction* API identified in CSC API [1], clause 11.11 shall apply.

The message for extending the validity of a multi-signature transaction authorization shall contain the components defined in CSC API [1], clause 11.11 section Input.

The message for returning the new authorization data for signing credential usage shall contain the components defined in CSC API [1], clause 11.11 section Output.

By invoking the *credentials/getChallenge* API the driving application can request a challenge needed to perform a challenge response protocol for the access to a credential for remote signing.

The *credentials/getChallenge* API identified in CSC API [1], clause 11.10 shall apply.

The message for requesting a challenge for an authentication object shall contain the components defined in CSC API [1], clause 11.10 section Input.

The message for returning the requested challenge or for indicating that a challenge has been sent by out-of-band means shall contain the components defined in CSC API [1], clause 11.10 section Output.

5.3.3 Create a remote signature with a credential protected by OAuth2

OAuth 2.0 authorization is a widely used framework that allows applications to obtain limited access to a user's resources on a service, without needing the user's authentication data (i.e. the user's password). OAuth 2.0 can provide client applications a secure delegated access to server resources on behalf of a resource owner. In the context of remote signatures creation, the driving application is the client application. This allows signing credentials owners to authorize third-party usage of their signing credentials without sharing their authentication and/or authorization data.

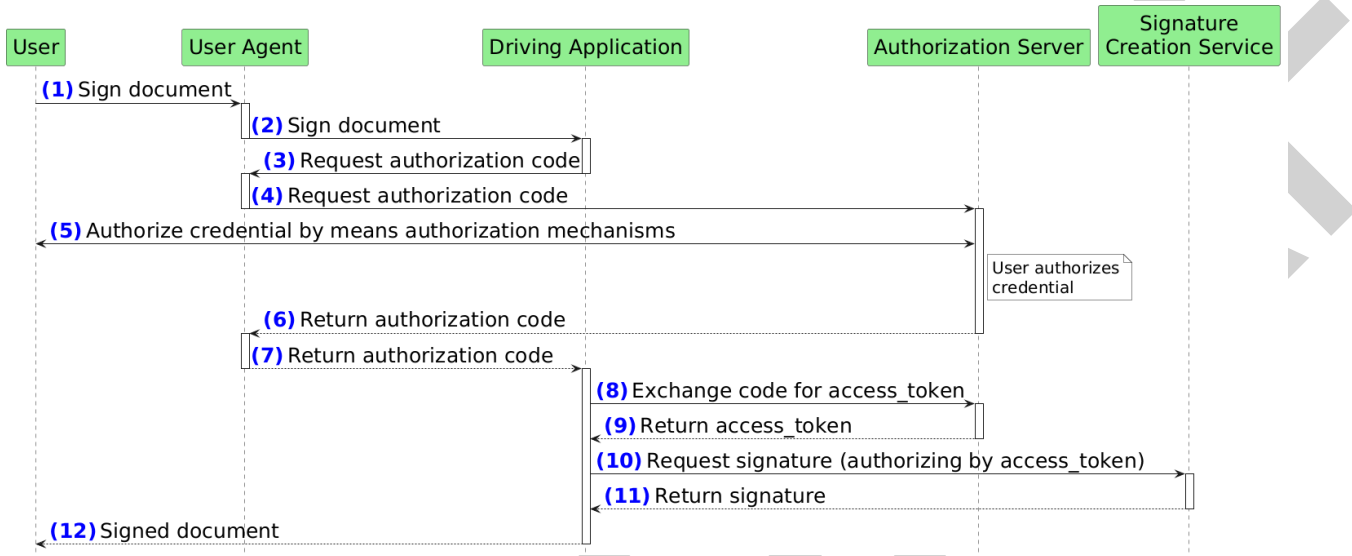

Figure 6: A general flow involving a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server

The above scheme is implemented in CSC API [1] by invoking the API:

- *oauth2/authorize* (and *oauth2/pushed_authorize* if needed), passing the needed information about the data to be signed and the signing credential to be used according to the parameters defined in clause 8.2.2 and 8.2.3 of CSC API [1], that manages authorization requests using the Authorization Code flow as described in Section 4.1 of IETF RFC 6749 [17].
- *oauth2/token* to obtain an OAuth 2.0 bearer access token from the authorization server by passing the authorization code or refresh token returned by the authorization server after a successful signatures creation authorization.
- *oauth2/revoke* passing the access or refresh token, that was obtained from the authorization server, to be revoked.

- *signatures/signHash* or *signatures/signDoc* to obtain the requested signature objects or signed documents as specified in clause 11.13 and 11.14 of CSC API [1].

By invoking the *oauth2/authorize* API, via the user agent, the driving application can obtain an authorization code for signatures creation authorization from the authorization server of the signature creation service.

The *oauth2/authorize* API identified in CSC API [1], clause 8.2.2 shall apply.

The message for requesting the authorization code for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Input considering what specified in sections Service scope and Credential scope and authorization details.

The message for returning the authorization code for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Output.

By invoking the *oauth2/pushed_authorize* API the driving application can push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and obtain a request URI to be used as reference to the data in a subsequent call to the authorization endpoint (*oauth2/authorize*) of the remote service.

The *oauth2/pushed_authorize* API identified in CSC API [1], clause 8.2.3 shall apply.

The message for requesting the request URI that will be used as reference to the data in a subsequent call to the authorization endpoint shall contain the components defined in CSC API [1], clause 8.2.2 section Input apart from the request_uri parameter.

The message for returning the request URI shall contain the components defined in IETF RFC 9126 [18], clause 2.2.

By invoking the *oauth2/token* API the driving application can obtain an access token for signatures creation authorization from the authorization server of the remote service.

The *oauth2/token* API identified in CSC API [1], clause 8.2.4 shall apply.

The message for requesting the access token for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Input.

The message for returning the access token for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Output.

By invoking the *oauth2/revoke* API the driving application can invalidate access or refresh token obtained from the authorization server of the signature creation service so that any further access by reusing the token itself is prevented.

The *oauth2/revoke* API identified in CSC API [1], clause 8.2.5 shall apply.

The message for invalidating an access or refresh token for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.5 section Input.

The *oauth2/revoke* API has no output values, the HTTP status code 204 "No Content", defined in clause 15.3.5 of IETF RFC 9110 [16], is provided in response.

5.4 Credential creation authorization

Requesting credential creation to the signature creation service requires an authorization from the user to whom the credential to be created will be bound.

CSC API [1] only supports OAuth 2.0 authorization to grant credential creation authorization.

DRAFT

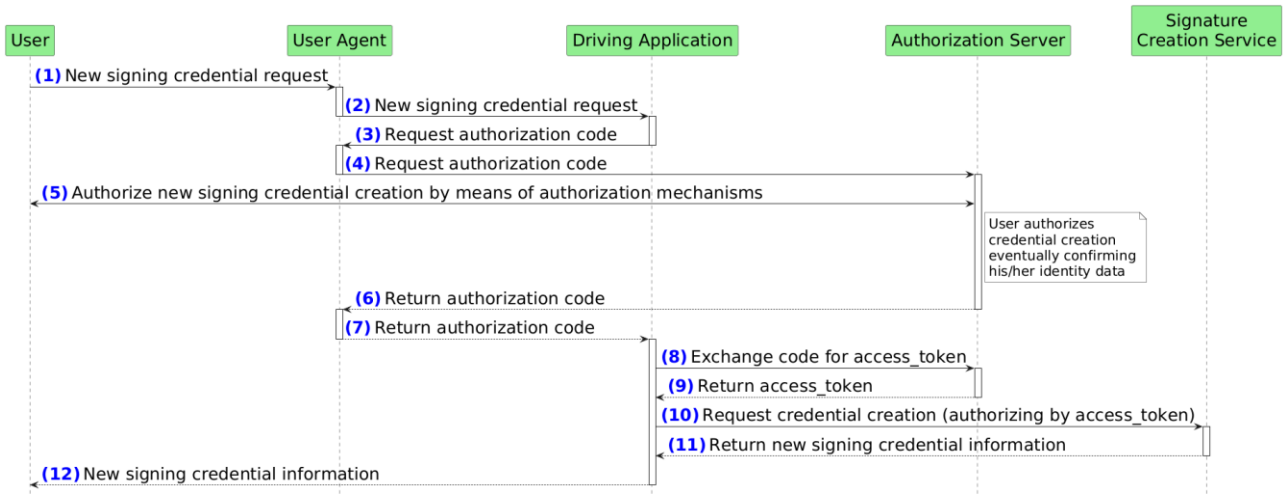

Figure 7: A general flow involving a driving application requesting signing credential creation to a signature creation service with an authorization process managed by an OAuth2 authorization server

The above scheme is implemented in CSC API [1] by invoking the API:

- *oauth2/authorize* (and *oauth2/pushed_authorize* if needed), possibly passing the subject data needed for the certificate signing request according to the parameters defined in clauses 8.2.2 and 8.2.3 of CSC API [1], that manages authorization requests using the Authorization Code flow as described in Section 4.1 of IETF RFC 6749 [17].
- *oauth2/token* to obtain an OAuth 2.0 bearer access token from the authorization server by passing the authorization code or refresh token returned by the authorization server after a successful credential creation authorization.
- *credentials/create* to requests credential creation possibly requesting information about the new signing certificate and/or about the authorization mechanisms required to authorize the usage of the new credential for signing as specified in clause 11.4 of CSC API [1].

By invoking the *oauth2/authorize* API, via the user agent, the driving application can obtain an authorization code for credential creation authorization from the authorization server of the signature creation service.

The *oauth2/authorize* API identified in CSC API [1], in clause 8.2.2 shall apply.

The message for requesting the authorization code for credential creation authorization shall contain the components defined in CSC API [1], clause 8.2.2 considering what specified in section Credential creation scope and authorization details.

The message for returning the authorization code for signatures creation authorization shall contain the components defined in CSC API [1], clause 8.2.2 section Output.

By invoking the *oauth2/pushed_authorize* API the driving application can push the payload of an OAuth 2.0 authorization request to the authorization server via a direct request and obtain a request URI to be used as reference to the data in a subsequent call to the authorization endpoint (*oauth2/authorize*) of the remote service.

The *oauth2/pushed_authorize* API identified in CSC API [1], clause 8.2.3 shall apply.

The message for requesting the request URI that will be used as reference to the data in a subsequent call to the authorization endpoint shall contain the components defined in CSC API [1], clause 8.2.2 sections Input and Credential creation scope and authorization details apart from the request_uri parameter.

The message for returning the request URI shall contain the components defined in IETF RFC 9126 [18], clause 2.2.

By invoking the *oauth2/token* API the driving application can obtain an access token for credential creation authorization from the authorization server of the remote service.

The *oauth2/token* API identified in CSC API [1], clause 8.2.4 shall apply.

DRAFT

The message for requesting the access token for credential creation authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Input.

The message for returning the access token for credential creation authorization shall contain the components defined in CSC API [1], clause 8.2.4 section Output.

Subject data needed for certificate issuance shall be obtained from, or verified through, an identity verification process appropriate to the requested certificate policy. The signature creation service may collect the subject data needed for the certificate signing request in different ways. The following is a non-exhaustive list of possible ways:

- when Rich Authorization Requests (RAR) are supported as specified in IETF RFC 9396 [35], the subject data may be provided in the authorization request;
- the subject data may be collected by the authorization server and made available to the signature creation service remote service:
  - in the authorization details of the access token;
  - as custom claims in the access token [i.16];
  - at a token introspection endpoint [i.8].

5.5 Credential deletion authorization

A credential may be deleted for various reasons:

- End of lifecycle: the credential is no longer needed or has expired.
- Security reasons: the credential may be compromised or the user requests revocation.
- Regulatory compliance: certain frameworks require deletion because of cryptographic algorithms obsolescence

Requesting credential deletion to the signature creation service requires an authorization from the user who owns and controls the credential or from a trusted driving application or trusted third-party. In cases where credential deletion authorization is not requested to the credential owner, the SCSP should inform the credential owner when a credential is deleted.

CSC API [1] only supports OAuth 2.0 authorization to grant credential deletion authorization.

The deletion process is implemented in CSC API [1] by invoking the API:

- *oauth2/authorize* (and *oauth2/pushed_authorize* if needed) according to the parameters defined in clause 8.2.2, specifically in section Credential deletion scope and authorization details, and clause 8.2.3 of CSC API [1], that manages authorization requests using the Authorization Code flow or the Client Credential flow, in case of trusted driving applications, as described respectively in Section 4.1 and 4.4 of IETF RFC 6749 [17].
- *oauth2/token*, possibly, to obtain an OAuth 2.0 bearer access token from the authorization server by passing the authorization code or refresh token returned by the authorization server after a successful credential deletion authorization.
- *credentials/delete* to requests credential deletion as specified in clause 11.5 of CSC API [1].

6 Architectures and use cases for server signing

6.1 Introduction

The following clauses describe a set of architectural models for the creation of remote electronic signatures, each distinguished by the mechanisms used to protect the signer's credential and to manage the authorization required for signature creation. The models differ in the interaction between components and in the authentication and authorization flows, but the governance and final responsibility of authentication and authorization mechanisms remain with the signature creation service provider.

DRAFT

Clause 6.2 presents architectures in which the signature creation credential is protected by an authorization mechanism fully managed by the signature creation service. Clause 6.3 introduces architectures leveraging OAuth 2.0-based protection of the signature creation credential, focusing on standardized authorization delegation and interoperability. Clause 6.4 describes architectures that employ the EUDIW as a central component for user authentication, authorization, and orchestration of signature-creation operations.

Together, these clauses provide a comprehensive overview of the main architectural models currently applicable to remote signature creation, supporting diverse deployment scenarios and regulatory requirements but other architectural models can be implemented achieving comparable levels of security.

Where CSC is referenced in the subsequent sub-clauses, the same considerations generally apply to OASIS DSS-X.

6.2 Architectures for creating remote signatures with a credential protected by signature creation service-managed authorization

NOTE: Explicit authorization requires the driving application to collect authentication factors (e.g. PIN, OTP) within its own environment. This introduces complexity in ensuring secure factor collection and storage. This mechanism should not be suitable to satisfy SCAL2 requirements.

Creating a remote signature using a signing credential protected by a signature creation service can be implemented in some different ways and involving the usage of different authorization data. Moreover, remote signatures creation under the eIDAS [i.1] Regulation involves a well-defined rationale rooted in security, compliance, and usability.

Below some examples of sequence diagrams illustrating specific different flows between a driving application and a signature creation service in the case of remote signatures creation with a signing credential protected by signature creation service-managed authorization mainly derived from CSC API [1].

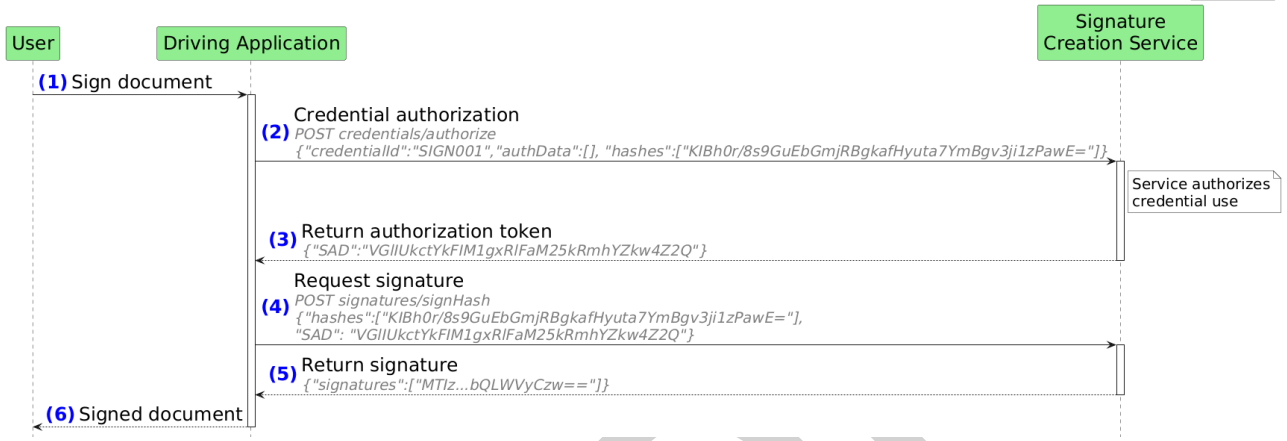

Figure 8: Remote hashes signature creation by using a signing credential whose usage is protected by authorization mechanisms managed by the signature creation service

Figure 9: Remote hashes signature creation by using a signing credential whose usage is protected by a PIN and an OTP provided through an agreed communication channel managed by the signature creation service (i.e. SMS, app, email)

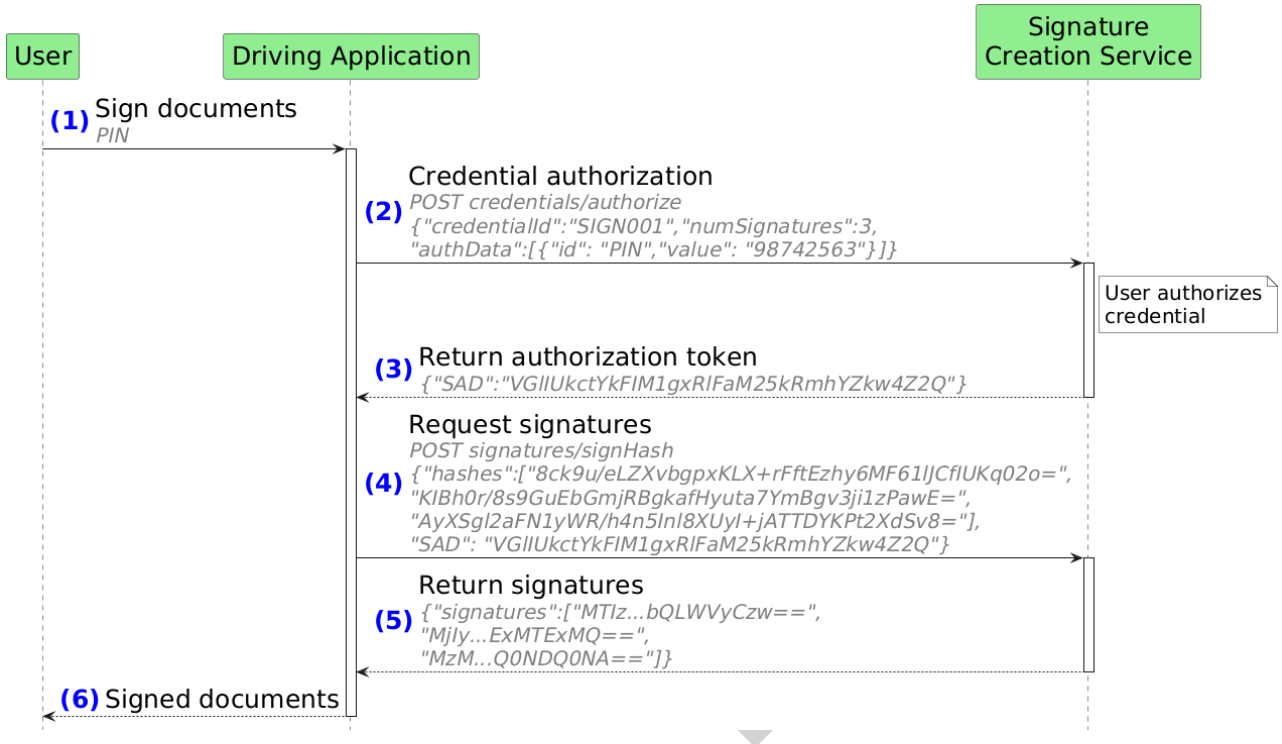

Figure 10: Multiple remote signatures creation from a list of hash values by using a signing credential whose usage is protected by PIN

6.3 Architectures for creating remote signatures with a credential protected by OAuth2

Using the OAuth 2.0 authorization scheme, the driving application, acting as OAuth client, will use the remote signature creation service's authorization server for user authentication and signing credential usage authorization. After a successful authentication and authorization, the authorization server of the signature creation service will provide the driving application with an access token that the driving application will use to authorize access to the signature creation service's resources (that's the users signing credentials).

There are some different ways in which requesting a remote signature creation authorization by means of an OAuth2 authorization server. In clauses 8.2.2 and 8.2.3 of CSC API [1] possible choices for selecting the signing credential and providing information about the data to be signed are specified.

The signing credential may be indicated in the *credentialID* parameter or selected according to the *signatureQualifier* parameter value that specifies the kind of signature to be created.

The data to be signed may be identified as data to be signed representation (hash value to be signed) or as signer's document or signer's document representation (i.e. hash of SD) together with a label containing a human-readable description of the respective document.

Moreover, the number of signatures to authorize and a list of signed attributes to be included in the signature may be provided too. Signed attributes or properties may be conveyed using the *signed_props* parameter of the CSC Data Model *documentInfo* or *adesParameters* objects, depending on whether they are part of the authorization information or the signature creation request.

Below some examples of sequence diagrams illustrating specific different flows between a driving application and a signature creation service in the case of remote signatures creation with a signing credential protected by an OAuth2 authorization server mainly derived from CSC API [1].

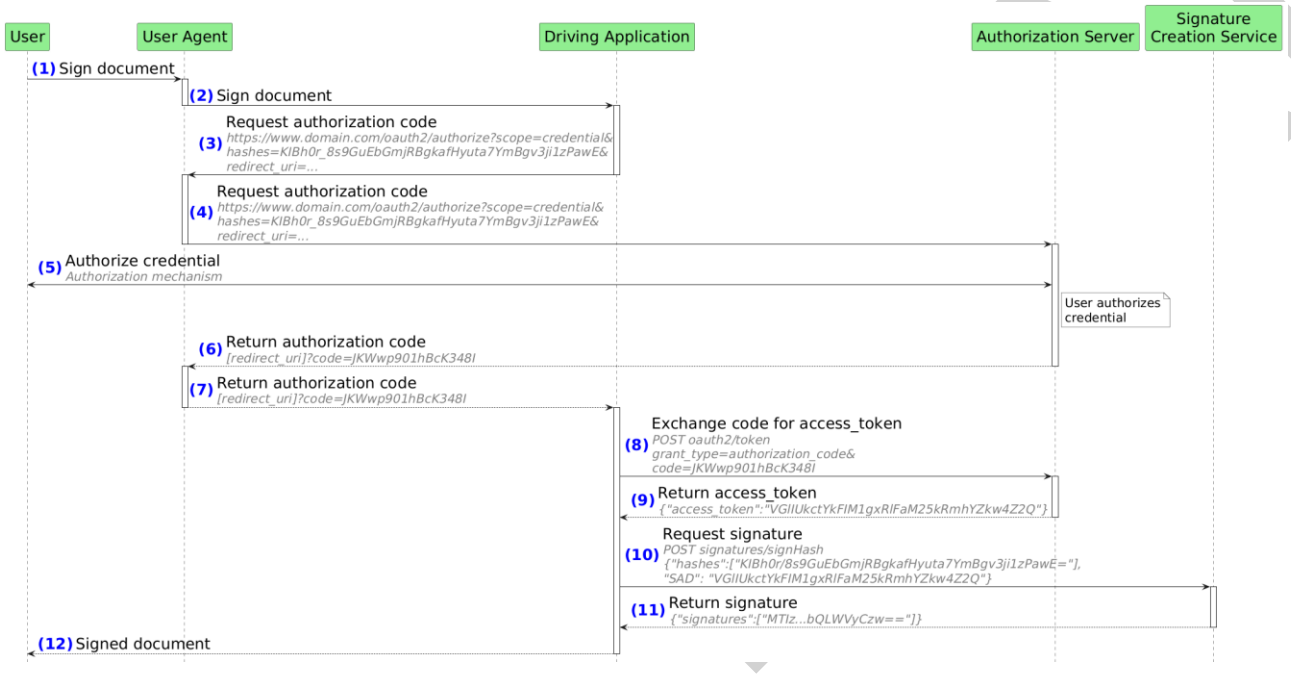

Figure 11: Sequence diagram illustrating an authorization code flow between a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server

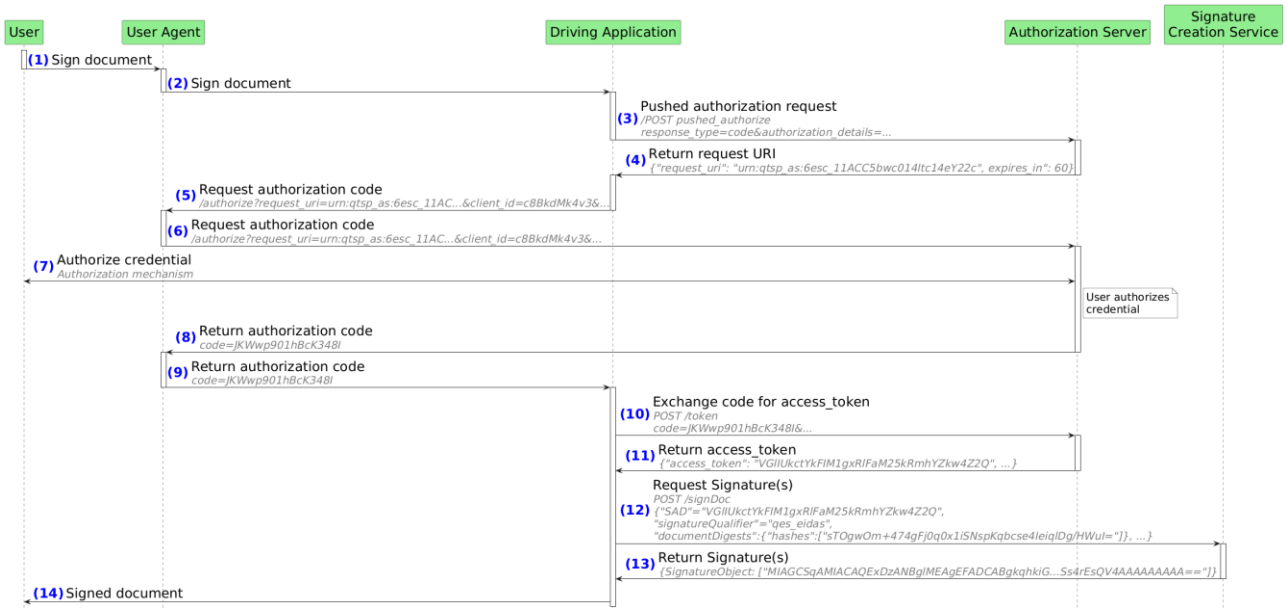

Figure 12: Sequence diagram illustrating an authorization code flow between a driving application requesting signatures creation to a signature creation service with a credential protected by OAuth2 authorization server by a pushed and rich authorization request

6.4 Architectures for creating remote signatures using the European Digital Identity Wallet

6.4.1 Overview

In the European Union, Qualified Electronic Signatures (QES) are governed by the eIDAS Regulation [i.1]. This regulation defines the European Digital Identity Wallet (EUDIW) as an electronic identification means that enables users to securely store, manage, and validate personal identification data and electronic attestations of attributes. The EUDIW serves the purpose of presenting these credentials to wallet-relying parties and other EUDIW users, and of performing qualified electronic signatures or qualified electronic seals.

EUDIWs shall allow users to sign using qualified electronic signatures and to seal using qualified electronic seals. In particular, they shall support common protocols and standardized interfaces for creating qualified electronic signatures or seals through qualified electronic signature or seal creation devices.

Several models may be implemented for remote signature creation using the EUDIW, depending on the signature creation flow and the authorization process. In these models one or more of the following roles can be played by the EUDIW:

- authentication and transaction-approval component, where the EUDIW serves as the authentication and authorization layer restricted to user authentication and consent confirmation without performing signature creation.
- Signature Creation Application, where the EUDIW prepares the signature format and invokes a remote signing service to obtain the digital signature value;
- Driving Application or OAuth client, where the EUDIW initiates or continues an OAuth/OpenID4VP-based authorization or signature creation flow.
-

DRAFT

6.4.2 EUDIW as authentication and authorization layer for creating remote signatures

In this clause, “EUDIW as authentication and authorization layer” means that the EUDIW is used to authenticate the signer, present verified attributes where needed, render the transaction data to the signer, and obtain explicit approval for the signature operation. The EUDIW does not create the AdES signature or invoke the remote signing API.

In the figures in this clause, the Wallet-Relying Party that creates the OpenID4VP Authorization Request towards the EUDI Wallet is the QTSP authorization server. Depending on the deployment, the OAuth client for the remote signature authorization flow may be the Driving Application, the EUDI Wallet, or the SCA. Each figure illustrates which component acts as OAuth client.

In Figure 12, the signer uses the EUDIW to authorize a QTSP to sign a document on his/her behalf, the QTSP and the RP play the central roles in managing the signature creation process, while the EUDIW is primarily responsible for authorizing the signature creation. The Signer Interaction Component (SIC) is represented in two distinct parts, SIC Part 1 and SIC Part 2, to reflect the separation of responsibilities between the browser-based user interaction layer and the EUDIW-resident trusted execution layer. The driving application runs in the RP environment and acts as OAuth client to request authorization for remote signature creation interacting with the authorization server via the user agent. The driving application requests AdES signatures creation to the QTSP which is integrating the SCA, i.e. by invoking CSC signDoc API. The authorization server creates the OpenID4VP authorization request to be processed by the EUDIW. The request may be forwarded to the EUDIW through an engagement mechanism selected according to the deployment scenario. In a same-device scenario, the browser or user agent may invoke the EUDIW using a custom URL scheme such as openid4vp:// or haip-vp://, or a verified application link, such as an Android App Link or Apple Universal Link, where supported by the platform. In a cross-device scenario, the browser or user agent may render a QR code containing an EUDIW invocation URL or a request_uri; the signer scans the QR code with the EUDIW on a second device. The QR code should not contain sensitive transaction data unless the data is integrity-protected and confidentiality requirements are met. The recommended approach is for the QR code or deep link to carry a short-lived request_uri referring to a signed OpenID4VP request object. The request object should be protected using JAR and, where supported, registered using PAR before being made available to the EUDIW. The EUDIW resolves the request_uri, validates the request signature, issuer, audience, expiry, state and nonce, renders the QES transaction data to the signer, and obtains explicit approval before returning the OpenID4VP response to the AS or to the configured response endpoint

Key steps

1. The DA prepares signatures creation request.
2. If no suitable certificate exists, credential creation may be authorized and performed (otherwise the signature creation shall be cancelled).
3. The DA redirects the signer’s browser/user agent to the AS for signatures creation authorization
4. The AS creates the OpenID4VP Authorization Request, including dcql_query and transaction_data.
5. The AS returns an EUDIW engagement instruction to the browser/user agent.
6. For same-device flows, the browser/user agent invokes the EUDIW directly.
7. For cross-device flows, the browser/user agent displays a QR code.
8. The EUDIW obtains the signed request object, validates it, renders the transaction to the signer and collects approval.
9. The EUDIW returns the VP / approval result to the AS, after which the AS can issue the signature authorization token or SAD-related authorization.
10. The DA invokes the SCA requesting signature creation.
11. Result is returned to DA/RP.

DRAFT

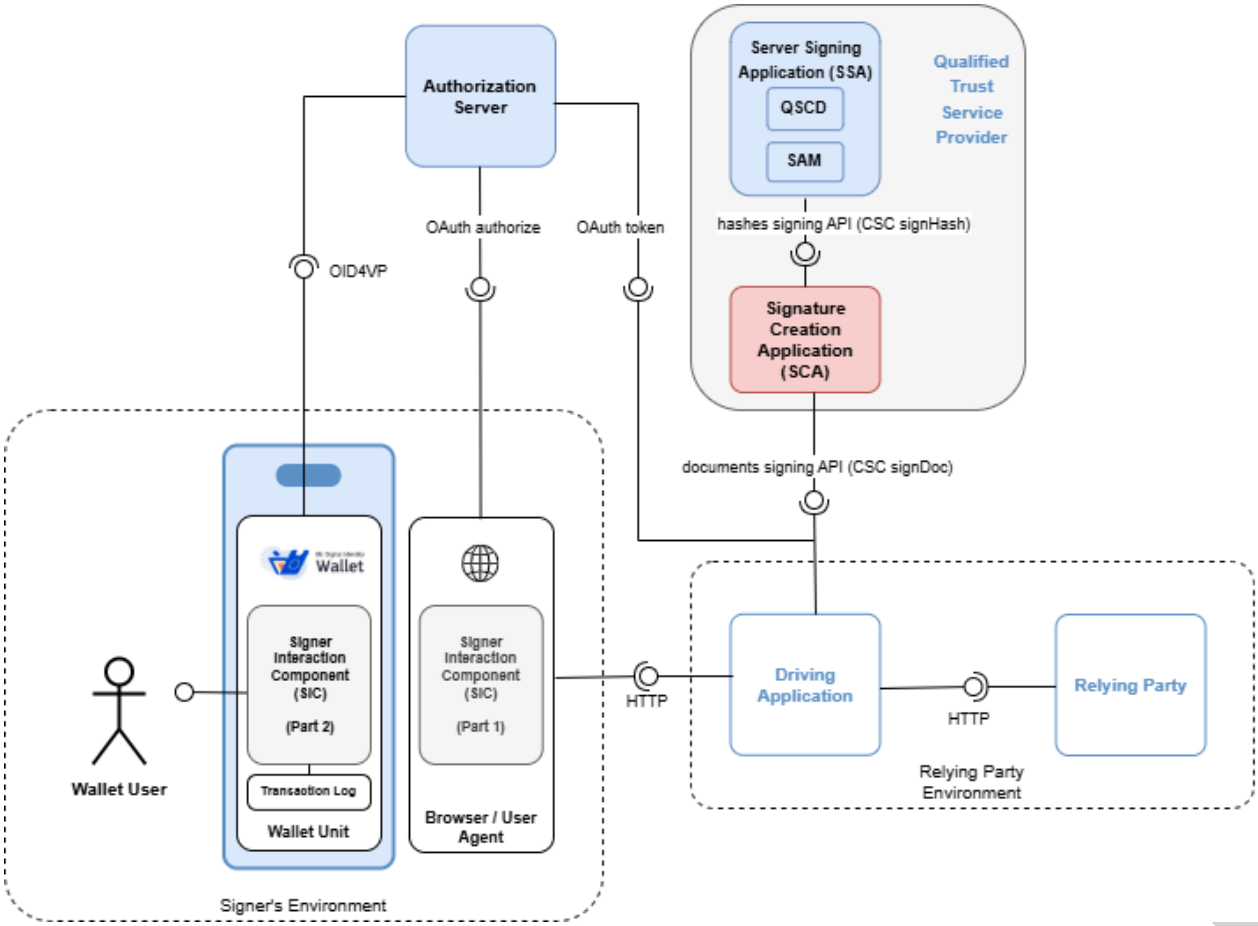

Figure 13: Architecture with SCA at QTSP and DA at RP

Different variations of the above architecture may be implemented according to the following alternatives:

- The user agent and the EUDIW may be in the same user device.
- The SCA may be implemented by an independent party or intermediary or by the RP that will operate both the driving application and the SCA.
- The driving application may be implemented by an independent party.
- The authorization server may be implemented by the QTSP operating the SSA.

In below figures the driving application acts as OAuth client to request authorization for remote signature creation.

Figure 13 shows an architecture in which the user agent and the EUDIW are in the same user device and the SCA and the driving application are implemented by different independent parties.

Figure 14 shows a slightly different architecture in which the user agent and the EUDIW are in different user devices and the SCA and the driving application are implemented by different independent parties.

Figure 15 shows an architecture in which the user agent and the EUDIW are in different user devices, the driving application is implemented at RP and the SCA is implemented by an independent party.

Figure 16 shows an architecture in which the user agent and the EUDIW are in different user devices, the driving application and the SCA are implemented at RP and the SCA interacts with authorization server requesting the tokens that authorize signatures creation.

Figure 17 shows an architecture in which the user agent and the EUDIW are in different user devices, the SCA and the driving application are implemented by different independent parties, and the authorization server is implemented by the QTSP operating the SSA.

DRAFT

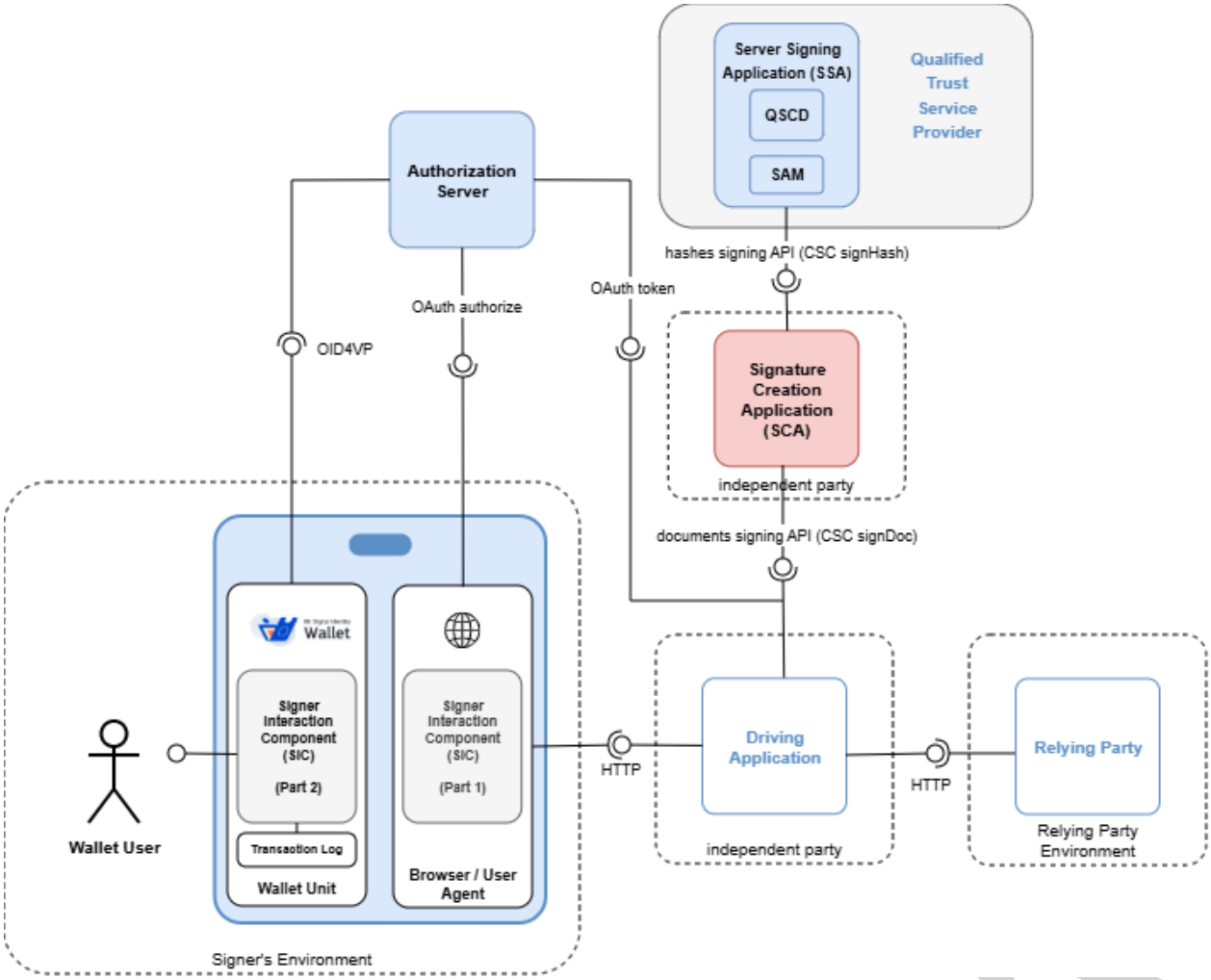

Figure 14: Architecture with SCA and DA at different parties and with EUDIW and user agent in the same user device

DRAFT

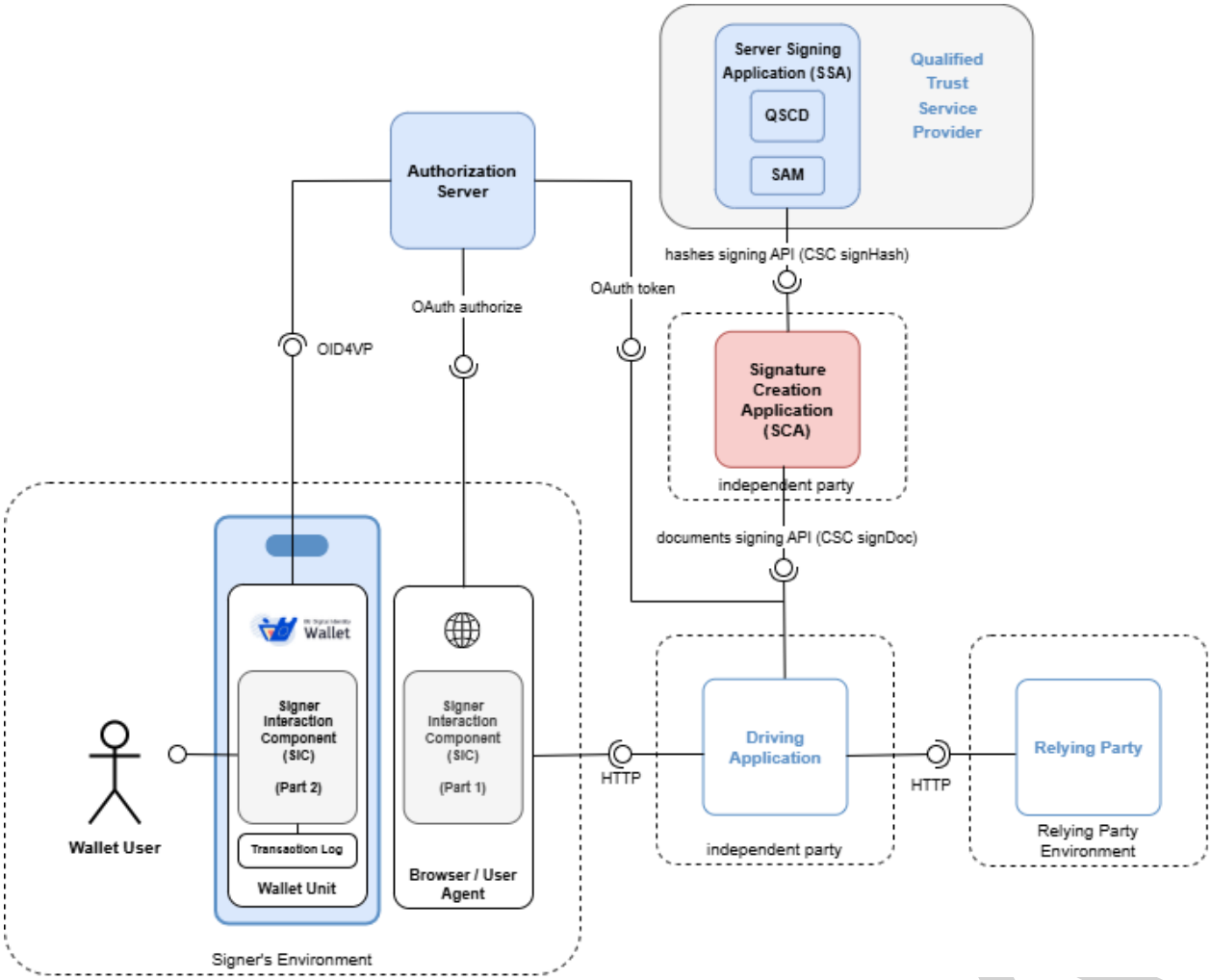

Figure 15: Architecture with SCA and DA at different parties

DRAFT

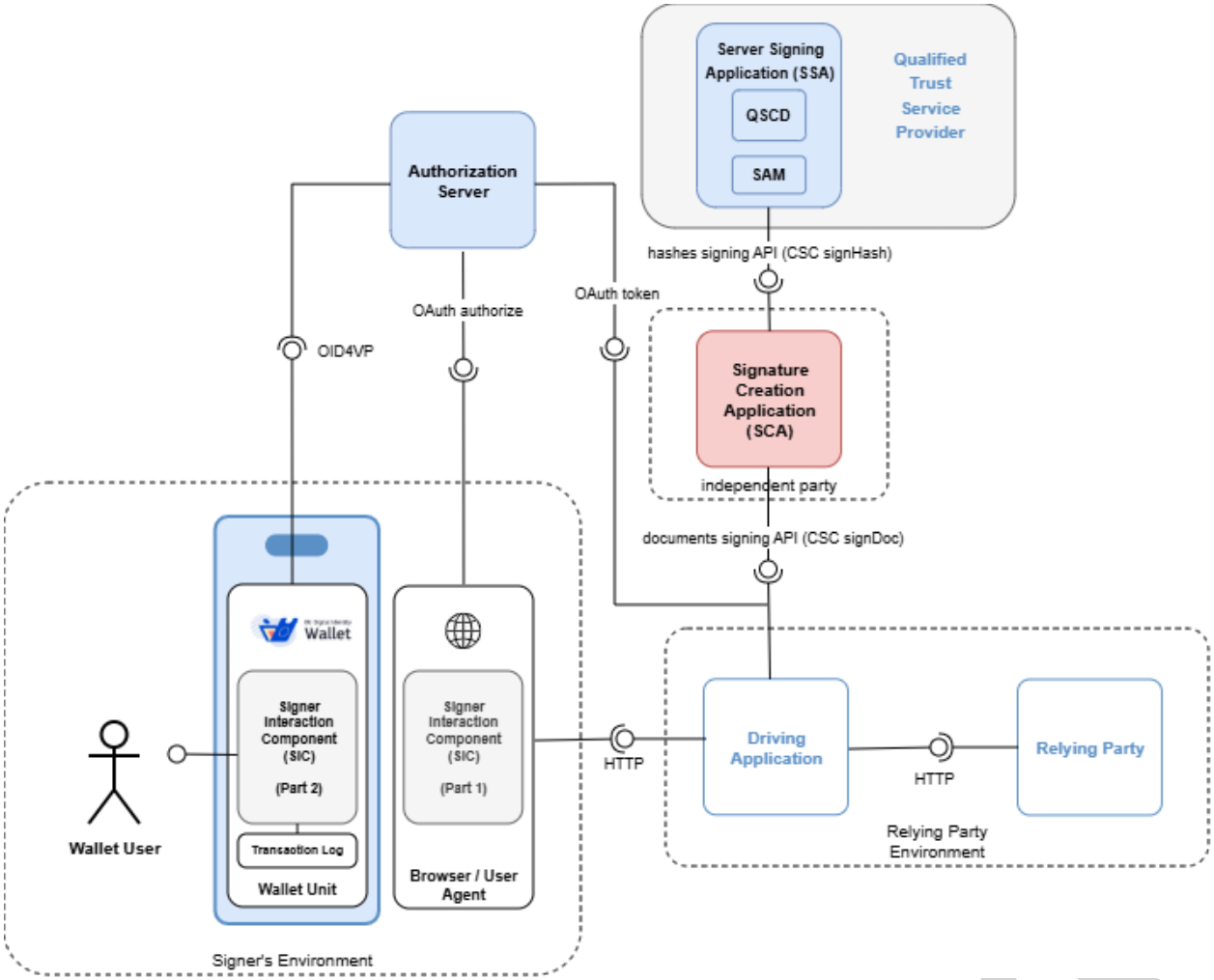

Figure 16: Architecture with DA at RP and SCA at independent party

DRAFT

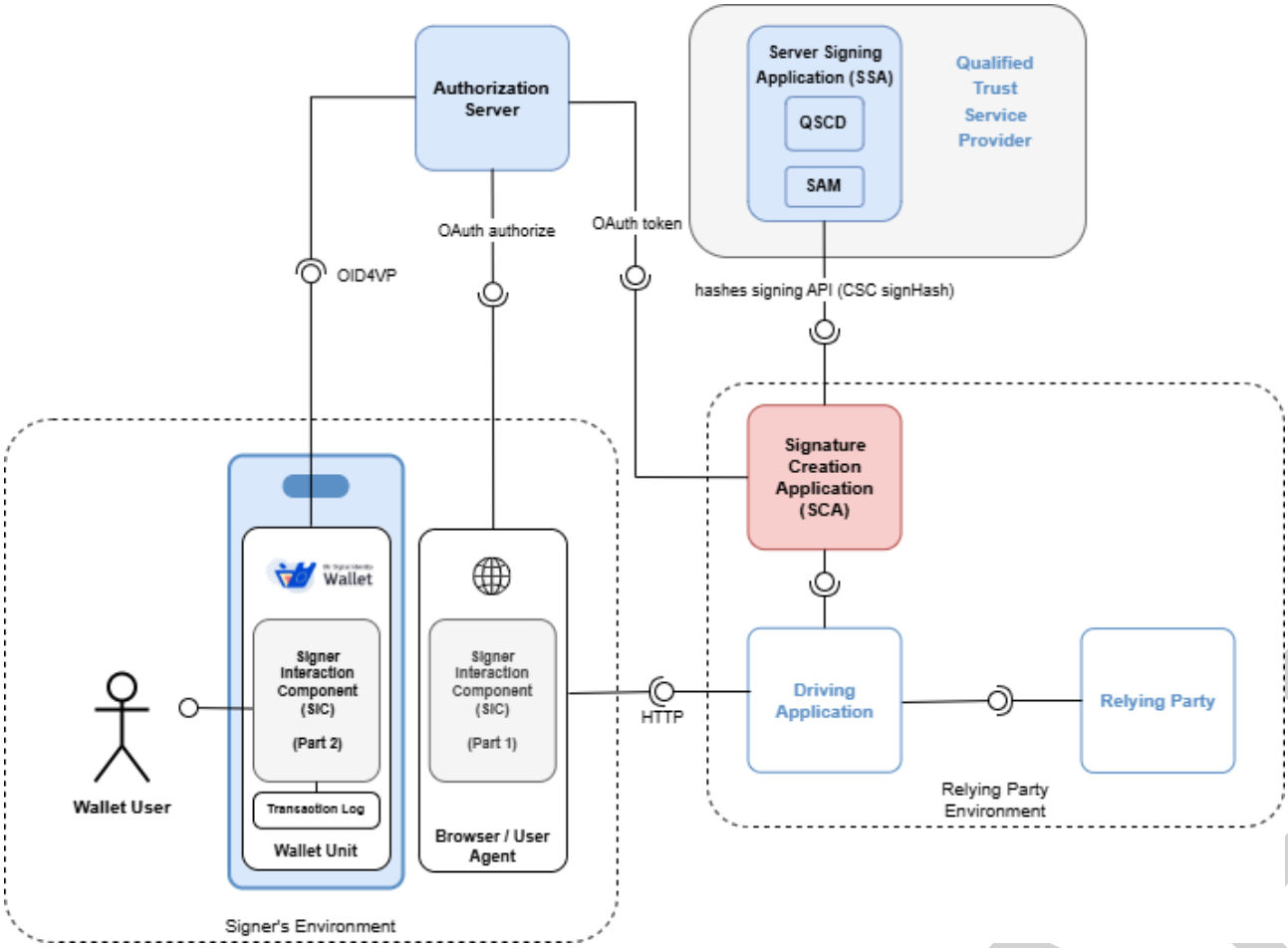

Figure 17: Architecture with DA and SCA at RP

DRAFT

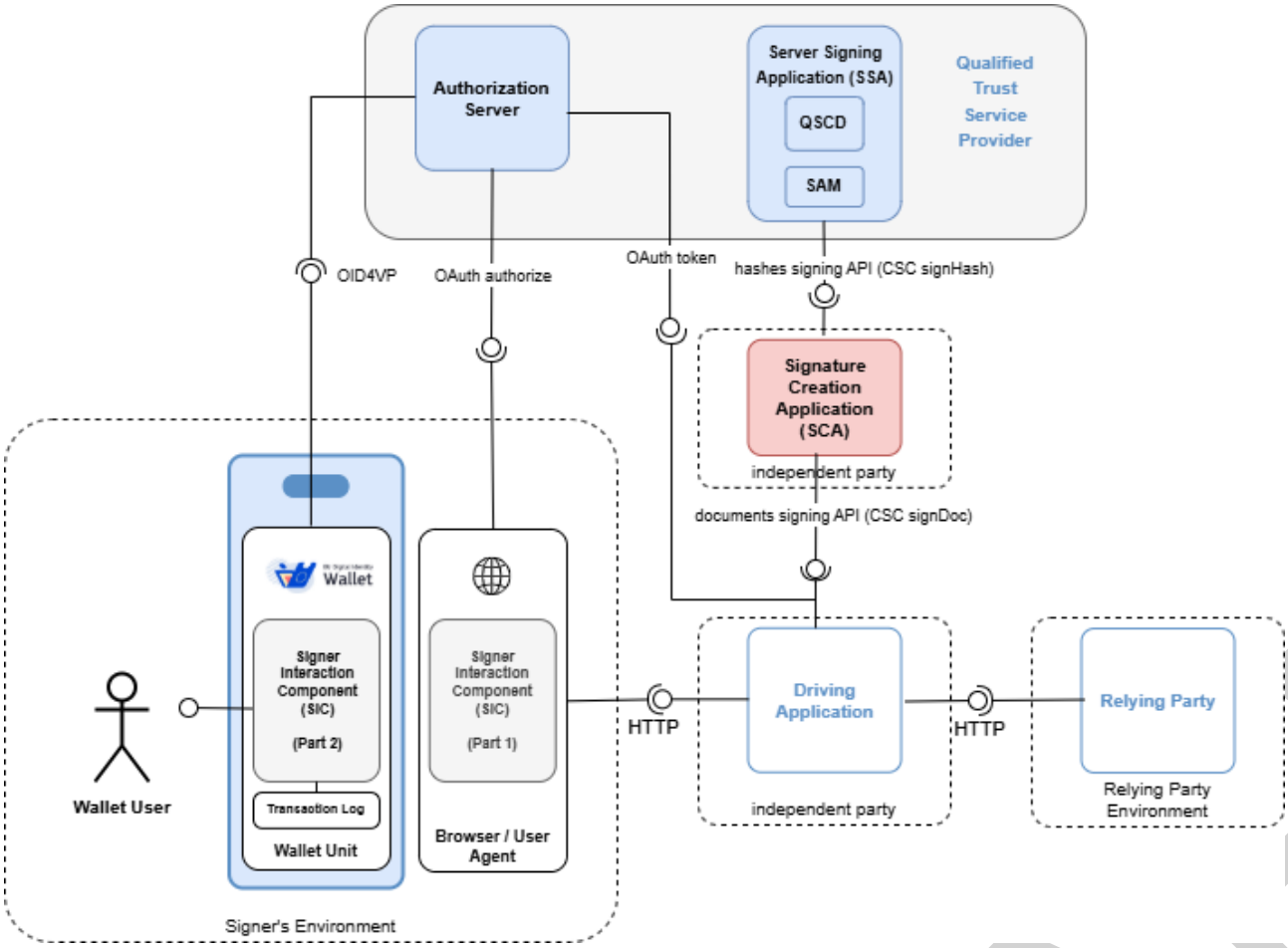

Figure 18: Architecture with authorization server at QTSP and with SCA and DA at different parties

In the above scenarios, the Authorization Server may make an OpenID4VP authorization request available to the EUDIW through a wallet engagement mechanism using the openid4vp:// or haip-vp:// custom URL scheme, as specified in Section 13.1.2 of OpenID for Verifiable Presentations 1.0 [22], or via Digital Credentials API methods [23]. The EUDIW processes the OpenID4VP request for example using PID presentation (if allowed as possible way to provide authorization to signatures creation) or transaction approval. The driving application can run in the local environment of the signer or in the environment of a relying party or in the environment of an independent intermediary. The driving application may request AdES signatures or digital signature values creation to the QTSP, i.e. by invoking CSC API [1].

Figure 18 shows the generic sequence where the EUDIW is used only for signer authentication and transaction approval. The Authorization Server creates an OpenID4VP Authorization Request and makes it available to the EUDIW through a wallet engagement mechanism. Same-device engagement may use a custom URL scheme such as openid4vp:// or haip-vp://, or a verified application link, such as an Android App Link or Apple Universal Link, where supported by the platform, or the Digital Credentials API. Cross-device engagement may use a QR code rendered by the browser or user agent and scanned by the EUDIW. The QR code or deep link should preferably contain a short-lived request_uri referencing a signed request object protected by JAR. After validating the request and obtaining signer approval, the EUDIW returns the presentation or approval result to the Authorization Server, which can then issue the authorization required for remote signature creation.

The driving application can be hosted at a relying party, at a TSP, or even an intermediary.

The signature creation application can be implemented at the driving application or at the signature creation service.

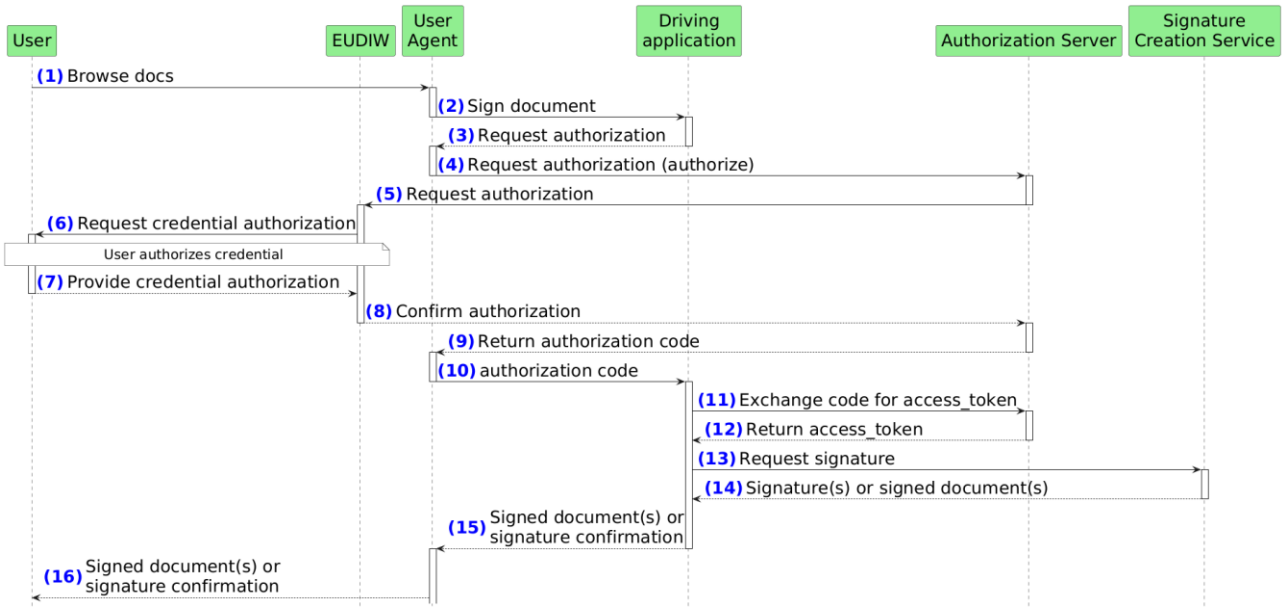

Figure 19: Generic simplified flow with EUDIW as authentication and authorization layer

Figure 19 refines the generic flow shown in Figure 18 by using the OAuth 2.0 Authorization Code flow together with Pushed Authorization Requests and Rich Authorization Requests. In this model, the Driving Application acts as OAuth client and requests authorization for remote signature creation. The authorization request may include rich authorization details describing the credential to be used or created, the number of signatures, the document digests or transaction data, and the applicable signature qualifier.

The Driving Application first sends the authorization request parameters to the Authorization Server using PAR. The request may contain, or refer to, a signed request object protected using JAR. The Authorization Server then creates an OpenID4VP Authorization Request for the EUDIW, containing the transaction data that the signer has to review and approve. The OpenID4VP request is forwarded to the EUDIW through a same-device or cross-device engagement mechanism. In the same-device case, the browser or user agent may invoke the EUDIW using a custom URL scheme such as openid4vp:// or haip-vp://, or a verified application link, such as an Android App Link or Apple Universal Link, where supported by the platform, or the Digital Credentials API. In the cross-device case, the browser or user agent may render a QR code containing a wallet invocation URL or a short-lived request_uri, which is scanned by the EUDIW.

The EUDIW retrieves and validates the OpenID4VP request, authenticates the signer, renders the transaction data, and obtains explicit approval. The transaction data may include the QES approval information needed by the Authorization Server to bind the signer's approval to the requested signature operation. After successful approval, the EUDIW returns the presentation or approval evidence to the Authorization Server. The Authorization Server validates the wallet response and, if policy requirements are met, completes the OAuth authorization process by issuing an authorization code. The Driving Application exchanges the authorization code for an access token and invokes the remote signature creation service, for example through CSC signatures/signHash or signatures/signDoc.

The OpenID4VP request forwarding step in the figure represents a logical interaction. Implementations should avoid transporting sensitive transaction data directly in QR codes or deep links. Instead, QR codes and deep links should preferably contain only a short-lived reference to a signed request object. The Authorization Server and EUDIW should enforce request integrity, expiry, audience validation, state and nonce checks, and single-use semantics for the referenced request.

The driving application can be hosted at a relying party, at a TSP, or even an intermediary.

The signature creation application can be implemented at the driving application or at the signature creation service.

DRAFT

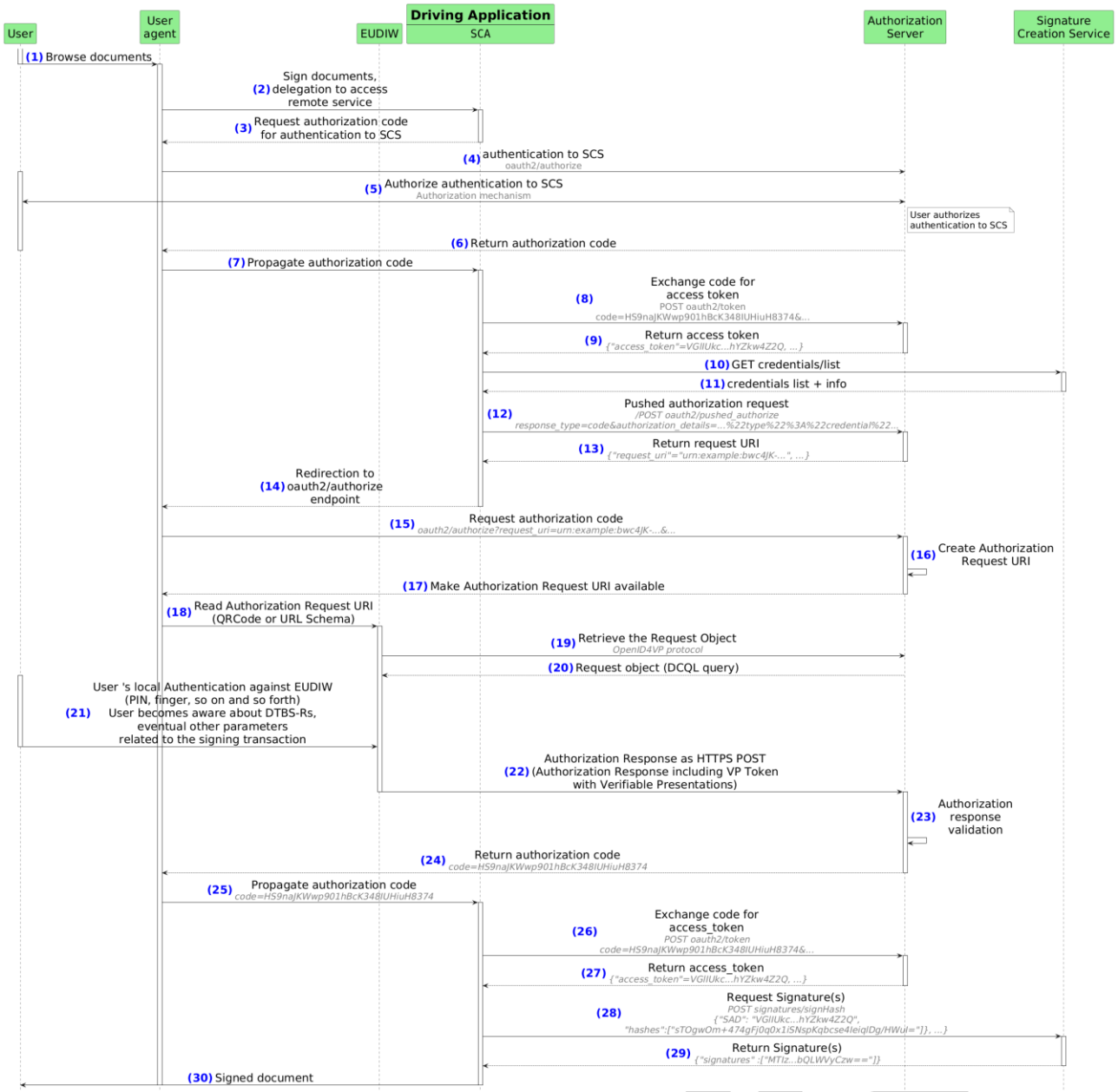

Figure 20: Remote signature creation with OAuth2 Authorization Code flow and Pushed and Rich Authorization Request with EUDIW as authentication and authorization layer

Service authentication and retrieval of the signing credential information may be omitted if the driving application knows in advance the user signing credential. The content of the request object passed to the EUDIW and of its response are specified in clause 7 of CSC data model bindings [3]. The transaction data type identifier for approving Qualified Electronic Signatures or seals (QES) creation is: "https://cloudsignatureconsortium.org/2025/qes-approval". The remote signatures creation may be performed by invoking the SSA CSC *signatures/signHash* or *signatures/signDoc* endpoints.

NOTE: In Figures 18 and 19, the arrow representing the OpenID4VP interaction between the Authorization Server and the EUDIW is logical. The actual handover to the EUDIW can be performed by same-device invocation, such as a custom URL scheme or Digital Credentials API, or by cross-device engagement, such as a QR code rendered by the browser or user agent. In both cases, implementations should use a short-lived request_uri referencing a signed request object and should apply replay protection, request integrity protection, and explicit signer consent.

6.4.3 EUDIW participating in the signature creation process

In the model outlined in this clause, the EUDI Wallet, or a wallet-controlled component, acts as the Signature Creation Application. In the model outlined in Figure 20, the EUDIW coordinates AdES signature creation through a SCA, that may be part of the EUDIW or of the EUDIW backend, interacting with the QTSP-operated SSA. The Wallet Relying Party (WRP) can pass data to be signed by the EUDIW through a structured and regulated process that ensures trust, transparency, and user control. The user's EUDIW plays the central role in managing the signature process to sign documents and to control and approve necessary transactions. The EUDIW receives or retrieves the signer's document, prepares the AdES data to be signed, computes the DTBSR, obtains signer approval, and invokes a remote SSA service, for example through CSC signatures/signHash, to obtain the digital signature value (DSV). The wallet then composes the final AdES signature or signed document and returns it to the Wallet-Relying Party.

In the model outlined in Figure 21, the SCA is implemented in the same user device where the EUDIW is implemented too but is not part of the EUDIW itself.

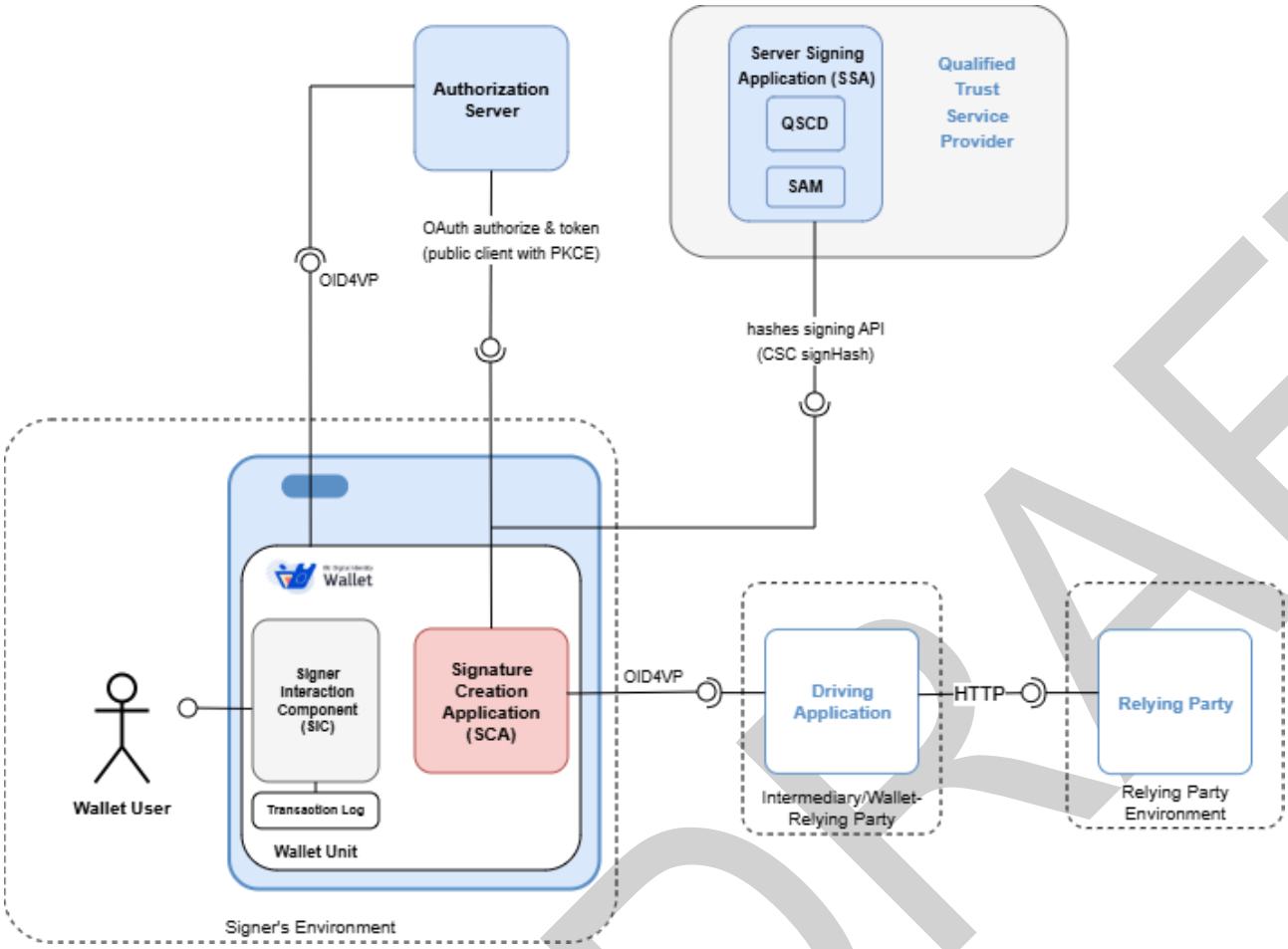

Figure 21: Architecture with EUDIW integrating SCA and SIC and DA running at independent intermediary

DRAFT

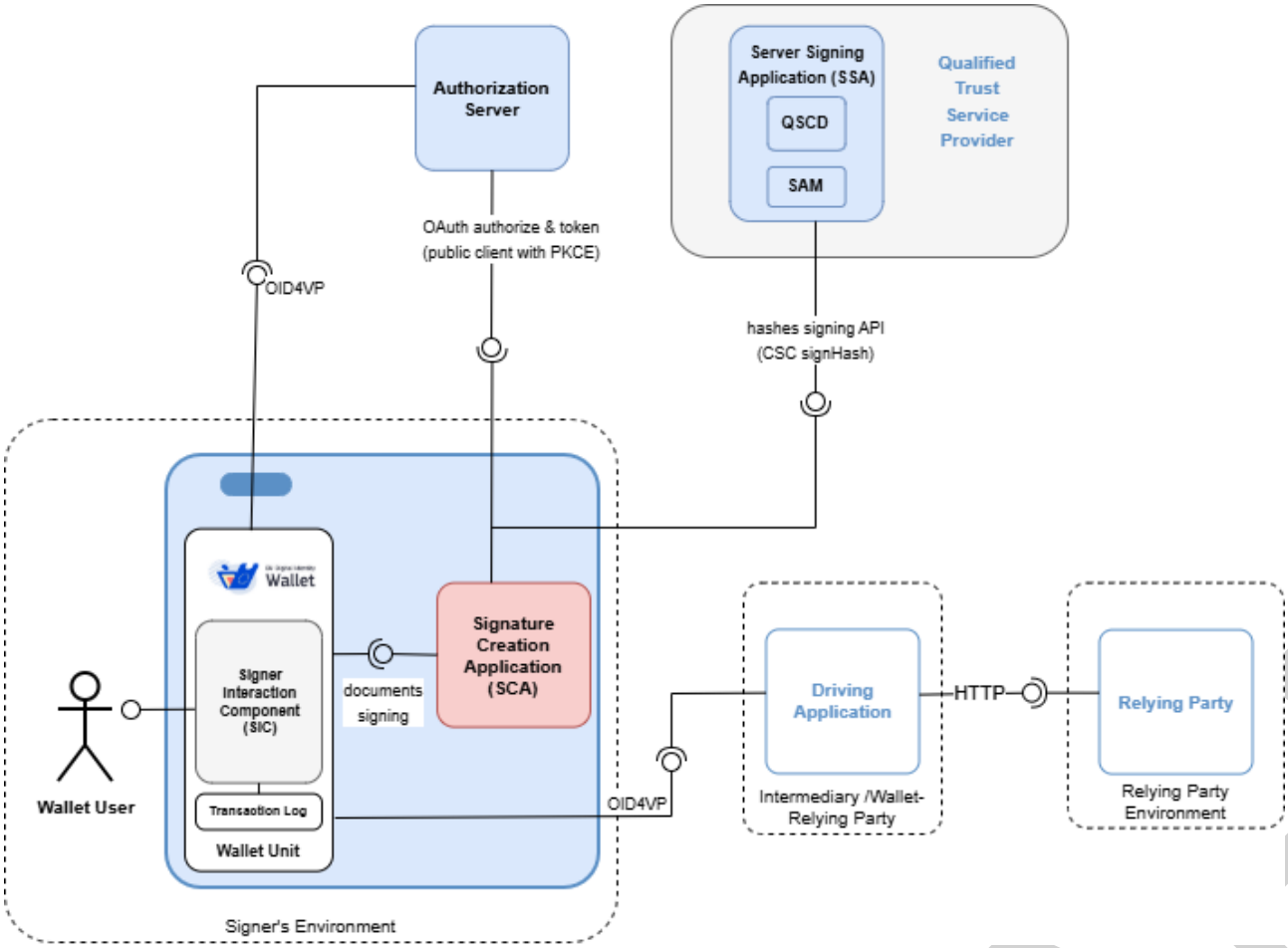

Figure 22: Architecture with SCA running in the same user device as the EUDIW and DA running at independent intermediary

Key Steps:

- Preparation of the signature transaction by the Driving Application

  The Driving Application prepares the signature transaction on behalf of the Relying Party. The transaction includes the document to be signed, or a representation of the document, the requested signature type, the requested signature format, and any information required to identify or create the signing credential. The SCA acts as OAuth client where OAuth 2.0 is used for authorization of the remote signature creation process.

- Determination of the signing credential or certificate creation need

  Before signature creation can be authorized, it shall be determined whether a suitable signing credential and corresponding signing certificate already exist. Two cases shall be distinguished:

  - Existing or long-term credential case: the signer already has a signing credential and certificate available at the QTSP or remote signature creation service. The authorization request identifies the credential directly, for example by credentialID, or indirectly, for example by signatureQualifier, certificate policy, or other credential selection criteria.

  - On-the-fly or one-shot credential case: no suitable signing credential exists, or the requested policy requires a transaction-specific credential. In this case, credential and certificate creation shall be authorized before or together with signature creation. The subject data used for certificate issuance shall result from an appropriate identity verification process and may be provided through the authorization process, through wallet-presented attributes, through access-token claims, or through token introspection.

- Construction of the authorization request by the Authorization Server

DRAFT

The Authorization Server creates, or causes to be created, the authorization request needed for the EUDIW interaction. Where OpenID4VP is used, the request shall contain transaction data describing the QES or seal transaction to be reviewed and approved by the signer. The transaction data should be carried as transaction_data and should use the CSC data model binding for QES approval, such as https://cloudsignatureconsortium.org/2025/qes-approval, where the EUDIW is used to approve the remote signature creation operation.

- Forwarding of the OpenID4VP request to the EUDIW

The OpenID4VP Authorization Request is made available to the EUDIW through a wallet engagement mechanism. In a same-device scenario, the browser or application may invoke the EUDIW using a custom URL scheme, or a verified application link, such as an Android App Link or Apple Universal Link, where supported by the platform, or the Digital Credentials API. In a cross-device scenario, the browser or application may render a QR code containing a wallet invocation URL or a short-lived request_uri, which is scanned by the EUDIW. The QR code or deep link should preferably contain only a reference to a signed request object rather than the full transaction data.

- Validation and rendering of the transaction by the EUDIW

The EUDIW validates the OpenID4VP request, including the issuer, audience, expiry, state, nonce, request integrity and, where applicable, the signature of the request object. The EUDIW then renders the transaction data to the signer. The rendered information should include at least the Wallet-Relying Party identity, the QTSP or signature creation service involved, the requested signature qualifier, the number of signatures, the document labels, the document hashes or checksums, the signature format, the conformance level, and whether an existing certificate or a newly created certificate will be used.

- Signer authentication and transaction approval through the EUDIW

The EUDIW authenticates the signer and obtains explicit approval for the transaction. Where the transaction involves certificate creation, the approval shall also cover the creation of the signing credential and certificate, including the certificate policy and the subject data to be used. Where the transaction involves only the use of an existing credential, the approval shall bind the signer's consent to the specific credential or credential selection criteria and to the specific document digests or transaction data.

- Return of wallet approval evidence to the Authorization Server

After successful signer approval, the EUDIW returns the OpenID4VP response, presentation, or transaction approval evidence to the Authorization Server or to the configured response endpoint. The approval evidence may include a qesApproval value or equivalent evidence bound to the transaction data. Such evidence supports the Authorization Server's authorization decision but is not, by itself, Signature Activation Data unless a specific profile defines how it is transformed into or cryptographically bound to SAD.

- Credential creation, where required

If the transaction requires a new signing credential or certificate, the QTSP or signature creation service creates the credential and associated certificate after successful authorization. For one-shot or short-lived credentials, the credential should be restricted to the authorized transaction, the authorized number of signatures, and the authorized document digests. For long-term credentials, the credential lifecycle and later use shall follow the applicable certificate policy and authorization requirements.

- Signature creation request to the QTSP or signature creation service

Once authorization has been completed, the SCA invokes the remote signature creation service. Depending on the deployment, this may be done through CSC signatures/signDoc where the QTSP creates the AdES signature, or CSC signatures/signHash where only the digital signature value is created remotely. The authorization data issued by the Authorization Server shall be bound to the approved transaction data, including document digests and the number of signatures.

- Signature creation using the QSCD / SCDev

The QTSP uses the remote QSCD or SCDev to create the digital signature value under the control of the signer. For SCAL2-level authorization, the signature activation process shall bind the activation to the specific data to be signed or to the approved document digests.

DRAFT

- Return of the signed document or signature object

  The resulting signed document, signature object, or digital signature value is returned to the Driving Application, Relying Party, or configured response endpoint according to the selected protocol and response mode. If the credential was created only for this transaction, the QTSP should apply the configured post-signing credential lifecycle action, such as expiry, deletion, or revocation where required by policy.

In Figure 22 representing a simplified signature model with EUDIW integrating SCA and SIC, the data flows between different components are outlined.

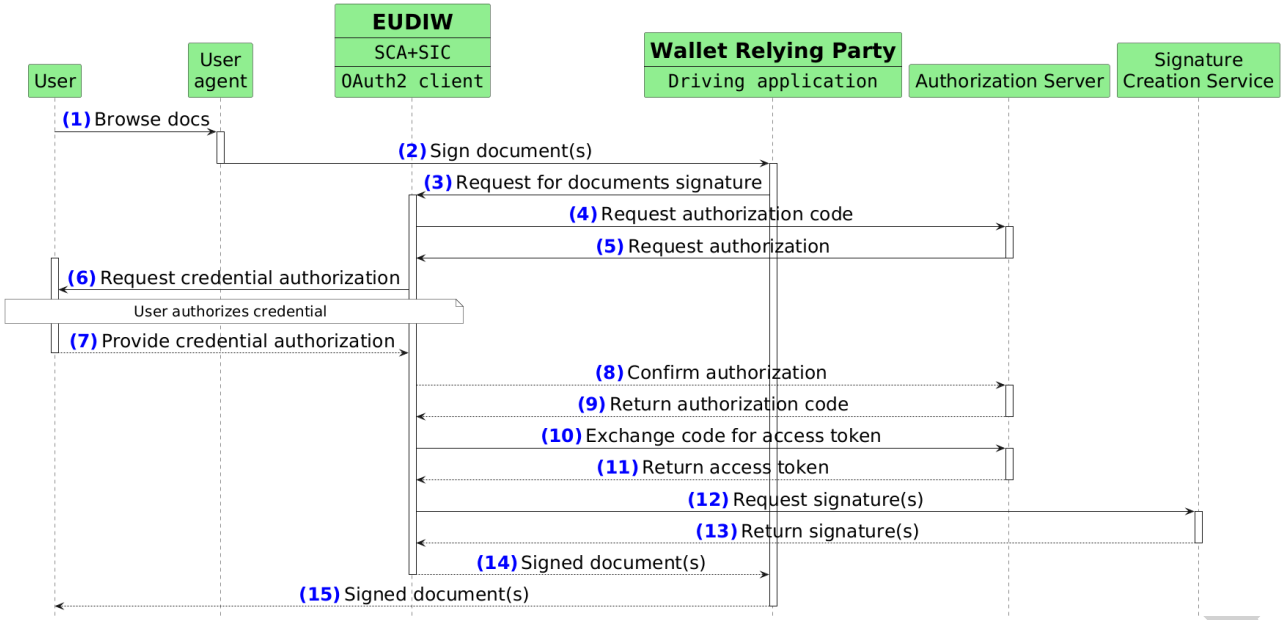

Figure 23: Generic simplified signature model flow with EUDIW integrating SCA and SIC

The content of the signature request object passed to the EUDIW and of its response are specified in clause 8 of CSC data model bindings [3]. The transaction data type identifier is https://cloudsignatureconsortium.org/2025/qes for requesting qualified electronic signatures or seals creation to EUDIW. Other more suitable request-response protocols for data other than verifiable presentations can arise in the future and could be used for this purpose. In annex A, a profile designing such a request-response protocol is defined.

6.4.4 EUDIW registration and authentication to authorization servers

6.4.4.1 Introduction

This clause defines the requirements for an EUDIW acting as an OAuth 2.0 client to register and authenticate to an authorization server using the OpenID Foundation FAPI 2.0 Security Profile [10], with explicit integration of Wallet Instance Attestation public key during dynamic client registration.

NOTE: At the time of publication of the present document, ETSI TS 119 476-3, specifying EUDI Wallet Unit Attestation, Wallet Instance Attestation and Wallet Key Attestation, was under development. A future version of the present document is expected to further align the EUDIW registration and authentication mechanisms in clause 6.4.4 with the final content of ETSI TS 119 476-3, including the use of WIA, WKA, WUA, wallet_link, eudi_wallet_info, client_status and key_storage_status.

6.4.4.2 EUDIW registration

- The authorization servers shall implement the FAPI 2.0 Security Profile and IETF RFC 7591, OAuth 2.0 Dynamic Client Registration Protocol.

- The authorization servers shall accept registration requests including client metadata values specified as claims in a software statement.

- The EUDIW shall register dynamically implementing requirements specified in IETF RFC OAuth 2.0 Dynamic Client Registration Protocol providing:
  - *redirect_uris*: the redirection URI strings for use in redirect-based flows;
  - *jwks*: EUDIW's JSON Web Key Set as specified in IETF RFC 7517 [12], which contains the EUDIW's public keys, the WIA public key used by the EUDIW shall be included;
  - *token_endpoint_auth_method*: containing the value private_key_jwt;
  - *grant_types*: the grant type strings that the EUDIW can use at the token endpoint, the value "authorization_code", representing the authorization code grant type defined in OAuth 2.0 clause 4.1, shall be included;
  - *response_types*: the OAuth 2.0 response type strings that the EUDIW can use at the authorization endpoint, the value "code", representing the authorization code response type defined in OAuth 2.0 clause 4.1, shall be included;
  - *software_statement*: a JSON Web Token (JWT), defined in RFC 7519 [i.17], that asserts metadata values about the EUDIW software as a bundle including the following claims:
    - *software_id*: a unique identifier string used by registration endpoints to identify the client software to be dynamically registered, it shall include the attribute refNum, that specifies the reference number of the wallet solution, as defined in class WalletSolution as defined in the specification of systems enabling the notification and subsequent publication of EUDIW provider information;
    - *client_name*: human-readable string name of the EUDIW to be presented to the end-user during authorization.If a WIA is provided and contains a *wallet_name* claim, *client_name* should be derived from *wallet_name*. If *wallet_name* is not present, *client_name* should be derived from the *eudi_wallet_info* object referenced by *wallet_link*, for example by combining wallet_provider_name, *wallet_solution_id* and, where useful, *wallet_solution_version*. The *client_name* value is for display and administrative purposes only and shall not be used as the sole basis for trusting the EUDIW client. If neither wallet_name nor sufficient eudi_wallet_info values are available, the value of client_name may be set to the generic value such "European Digital Identity Wallet".;
    - *client_attestation* (optional): an attestation that the authorization server may use to evaluate whether to accept the registration request. If provided for an EUDIW client, the client_attestation should include, or be bound to, a Wallet Instance Attestation (WIA) issued by the Wallet Provider. The WIA attests the integrity and authenticity of the Wallet Instance and binds the Wallet Instance to cryptographic key material through the confirmation information. Where the security properties of the Wallet Unit key storage are relevant, the registration or subsequent authorization process may also use Wallet Key Attestation (WKA) information. The Authorization Server may reject the registration if the attestation is invalid, expired, revoked, not bound to the registered key material, or not issued under an accepted trust framework.
  - other optional client metadata parameters defined in OpenID Connect Relying Party Metadata Choices 1.0 [13] that the authorization server may ignore if not supported.
- The authorization servers may validate *client_attestation* evidence and reject registration if the attestation is invalid, expired, or revoked.
- The SCS shall provide the information stating if the *client_attestation* claim is mandatory or optional or not supported.
  - The information may be provided in the service information API response message.
  - The information may be provided in authorization server metadata. In such case the authorization server shall publish metadata indicating whether EUDI Wallet client registration is supported and, where supported, whether wallet client attestation is required. This metadata may be published as part of the authorization server metadata, or through a signature-creation-service metadata document referencing the relevant Authorization Server. The metadata shall indicate whether dynamic client registration by EUDI Wallets is supported, whether a wallet client attestation is required, optional, or

DRAFT

unsupported, and which wallet client attestation formats are accepted. Proposed metadata parameters are listed below:

- *wallet_client_registration_supported*: boolean value indicating whether the authorization server supports client registration by EUDI Wallets for the purposes of signer authentication, transaction approval, credential creation authorization, or remote signature creation authorization. Presence: OPTIONAL, default value: false, if omitted. If set to true, the Authorization Server supports registration of EUDI Wallet clients, either dynamically according to RFC 7591 or through an equivalent pre-registration mechanism. If set to false or omitted, EUDI Wallets shall not assume that direct wallet client registration is supported by the Authorization Server.

- *wallet_client_attestation*: string value indicating whether the Authorization Server requires wallet client attestation during EUDI Wallet client registration or authentication. Presence: OPTIONAL, allowed values: "required", "optional", "not_supported". default value: not_supported, if omitted. Semantics

  - required: the EUDI Wallet shall provide wallet client attestation evidence during registration, authentication, or both, according to the authorization server policy. If attestation is missing, invalid, expired, revoked, or not issued under an accepted trust framework, the Authorization Server shall reject the registration or authentication attempt.

  - optional: the authorization server accepts wallet client attestation if provided, but does not require it for all EUDI Wallet clients. The authorization server may apply different policy decisions depending on whether attestation is provided.

  - not_supported: the authorization server does not process wallet client attestation evidence. EUDI Wallets shall not include wallet client attestation.

- *wallet_client_attestation_formats_supported*: Array of strings identifying the wallet client attestation formats accepted by the authorization server. Presence: REQUIRED Conditional Required if wallet_client_attestation is required or optional. This parameter lists the attestation evidence formats that the Authorization Server can validate for EUDI Wallet registration or authentication.

  - If wallet_client_attestation is required, an EUDI Wallet shall provide attestation evidence in one of the listed formats.

  - If wallet_client_attestation is optional, an EUDI Wallet may provide attestation evidence in one of the listed formats.

  - If wallet_client_attestation is not_supported, this parameter should be omitted or contain an empty array.

- The present document defines the value "eudi_wallet_instance_attestation" with the following description: attestation evidence that the client is a wallet instance conforming to the applicable EUDI Wallet trust framework and that the cryptographic key material used by the wallet client is bound to the attested wallet instance. Additional values may be defined by profiles or trust frameworks. Values not defined by the present document shall use collision-resistant identifiers. Upon a successful registration request, the authorization server shall respond with an HTTP 201 "Created status" code and a body of type "application/json" with the following fields:

  - client_id: the OAuth 2.0 client identifier string that should not be valid for any other registered client, though an authorization server may issue the same client identifier to multiple instances of a registered client at its discretion;

  - all registered metadata about the EUDIW, including the software statement used as part of the registration request whose value shall be returned unmodified in the response.

- Upon an unsuccessful registration request, the authorization server shall return an HTTP 400 status code with content type "application/json" consisting of a JSON object (IETF RFC 8259 [i.9]), describing the error in the response body, with two members:

DRAFT

- error: single ascii error code string.
- error_description: human-readable ASCII text description of the error (optional).

NOTE: The JWT representing the software_statement can be signed by an EUDIW trust authority / registry / provider authority.

6.4.4.3 EUDIW authentication

- The EUDIW shall implement OAuth 2.0 client capabilities.
- The EUDIW shall use Authorization Code Flow with PKCE (IETF RFC 7636 [14]).
- The EUDIW shall use PAR (IETF RFC 9126 [18]) with a signed JAR (IETF RFC 9101 [31]) for request integrity.
- The EUDIW authentication at PAR and token endpoints shall be supported via private_key_jwt as specified in Section 9 of OIDC.
- The EUDIW shall use state and nonce claims and validate issuer metadata (IETF RFC 8414 [15]) as specified in IETF RFC 9207 [24].

6.4.4.4 Security requirements

- The authorization server shall support issuance of sender-constrained access and refresh tokens by implementing the application-level mechanism Demonstrating Proof of Possession (DPoP), as specified in IETF RFC 9449 [25] as per FAPI.
- The EUDIW shall sign the DPoP according to JWT syntax with the private key whose public part is included in the WUA.
- The DPoP shall include the fields specified in clause 4.2 of IETF RFC 9449 [25].
- When creating the DPoP according to JWT syntax the recommendations provided in IETF RFC 8725 [26] should be followed.
- SCS should accept access tokens with DPoP proofs.
- If SCS accepts access tokens with DPoP proofs it shall validate any DPoP proof JWT.

6.4.4.5 Examples

6.4.4.5.1 Dynamic Client Registration request payload (JSON)

POST /oauth/register HTTP/1.1
Host: as.example.eu
Content-Type: application/json

```
{
  "client_name": "European Digital Identity Wallet (MemberState-X)",
  "application_type": "native",
  "redirect_uris": [
    "https://wallet.example.eu/oauth2/callback",
    "https://wallet.example.ms/callback"
  ],
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "private_key_jwt",
  "token_endpoint_auth_signing_alg": "ES256",
  "jwks": {
    "keys": [
      {
        "kty": "EC",
        "crv": "P-256",
        "use": "sig",
        "alg": "ES256",
```

DRAFT

```
        "kid": "wua-key-2026-01",
        "x": "TCAER19Zvu3OHF4j4W4vfSVoHIP1ILilDls7vCeGemc",
        "y": "ZxjiWWbZMQGHVWKVQ4hbSIirsVfuecCE6t4jT9F2HZQ"
      }
    ]
  }
  "tls_client_certificate_bound_access_tokens": true,
  "require_pushed_authorization_requests": true,
  "request_object_signing_alg": "ES256",
  "authorization_details_types": ["openid_credential_issuance"],
  "software_statement": "<signed-JWT>",
  "scope": "openid profile"
}
```

and the software statement would include these matching values:

```
{
  "iss": "https://eudi-trust-list.example.eu",
  "sub": "EU.EUDI.WS.IT.000456",
  "aud": "https://as.example.eu/register",
  "software_id": "EU.EUDI.WS.IT.000456",
  "software_version": "2.4.1",
  "client_name": "European Digital Identity Wallet",
  "application_type": "native",
  "redirect_uris": [
    "https://wallet.example.eu/oauth2/callback",
    "https://wallet.example.ms/callback"
  ],
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "none",
  "dpop_bound_access_tokens": true,
  "require_pushed_authorization_requests": true
}
```

6.4.4.5.2 Authorization server response

HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-store

```
{
  "client_id": "client-9f7e2",
  "client_id_issued_at": 1736940201,
  "registration_access_token": "eyJraWQiOiJtZy0yMDI2LTAxIn0.eyJzY29wZSI6...",
  "registration_client_uri": "https://as.example.eu/oauth/register/client-9f7e2",
  "client_name": "European Digital Identity Wallet (MemberState-X)",
  "application_type": "native",
  "redirect_uris": ["https://wallet.example.eu/oauth2/callback"],
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "private_key_jwt",
  "token_endpoint_auth_signing_alg": "ES256",
  "jwks": {
    "keys": [
      {
        "kty": "EC",
        "crv": "P-256",
        "use": "sig",
        "alg": "ES256",
        "kid": "wua-key-2026-01",
        "x": "TCAER19Zvu3OHF4j4W4vfSVoHIP1ILilDls7vCeGemc",
        "y": "ZxjiWWbZMQGHVWKVQ4hbSIirsVfuecCE6t4jT9F2HZQ"
      }
    ]
  },
  "request_object_signing_alg": "ES256",
  "require_pushed_authorization_requests": true,
  "tls_client_certificate_bound_access_tokens": true,
  "software_statement": "<signed-JWT>",
  "scope": "openid profile"
}
```

6.4.4.5.3 EUDIW authorization request payload (JSON)

FAPI 2.0 requires:

DRAFT

- PAR (Pushed Authorization Request) for all authorization requests
- JAR (JWT-Secured Authorization Request) as the request object
- response_type=code (only)
- PKCE (S256)
- sender-constrained tokens (DPoP or mTLS); for EUDI Wallet clients, sender-constrained access tokens should be based on DPoP rather than mTLS where the use of client certificates would create avoidable correlation or privacy risks
- Signed request object containing all parameters (AS ignores URL parameters except client_id).

FAPI also mandates use of request object signing and PAR.

EUDIW sends a pushed authorization request with a signed JAR (JWS).

POST /par HTTP/1.1
Host: as.example.eu
Content-Type: application/x-www-form-urlencoded
Authorization: DPoP eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3A...
DPoP: eyJhbGciOiJFUzI1NiIsInR5cCI6ImRwb3AifQ...

client_id=client-12345&
request=eyJhbGciOiJQUzI1NiIsImtpZCI6Ind1YS1rZXktMjAyNi0wMSIsInR5cCI6Im9hdXRoLXJlcXVlc3Qrand0In0.
eyJpc3MiOiJjbGllbnQtMTIzNDUiLCJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuZXUiLCJyZXNwb25zZV90eX
BlIjoiY29kZSIsImNsaWVudF9pZCI6ImNsaWVudC0xMjM0NSIsInJlZGlyZWN0X3VyaSI6ImV1ZGl3YWxsZXQ6L
y9jYiIsInNjb3BlIjoib3BlbmlkIHByb2ZpbGUiLCJzdGF0ZSI6ImFmMGlmanNsZGtqIiwibm9uY2UiOiJuLTBTNl9Xek
EyTWoiLCJjb2RlX2NoYWxsZW5nZSI6InM0UW1ab2R3SmFZRmpUZlZLbHBjRUYiLCJjb2RlX2NoYWxsZW5nZ
V9tZXRob2QiOiJTNTYiLCJleHAiOjE3MzY5NDAxMDAsIm5iZiI6MTczNjkzOTQwMH0.VERY-LONG-
SIGNATURE...

The request parameter contains a signed JAR (JSON Web Signature) as specified in IETF RFC 9101 [31].

FAPI 2.0 requires PAR so that all request parameters are integrity-protected.

Client authentication at PAR will use private_key_jwt or DPoP.

JWS header uses the WIA-attested key (kid).

6.4.4.5.4 PAR Response (from AS to EUDIW)

HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-store

```
{
  "request_uri": "https://as.example.eu/oauth2/par/request/cx298dhasd98hasd9",
  "expires_in": 90
}
```

request_uri can only be used once, an authorization server ignores all authorization request parameters except client_id and request_uri.

6.4.4.5.5 Authorization Request (from EUDIW to authorization endpoint)

GET /authorize?
client_id=client-12345&
request_uri=urn:as:par:cx298dhasd98hasd9
HTTP/1.1
Host: as.example.eu

DRAFT

This request contains almost nothing except client_id and request_uri. All security parameters are inside the signed JAR that was previously pushed via PAR. request URI points to the previously registered JAR as stated in FAPI.

6.4.4.5.6 Authorization server redirects back with authorization code

If the user authenticates successfully and consents, the authorization server redirects:

```
HTTP/1.1 302 Found
Location: eudiwallet://cb?
  code=SplxlOBeZQQYbYS6WxSbIA&
  state=af0ifjsldkj
```

state is returned exactly as provided by the client, code is bound to the client through sender-constrained token issuance with DPoP. Response mode is the default (query mode) because response_type=code only.

6.4.5 DA / SCA registration and authentication to authorization servers

6.4.5.1 Introduction

This clause defines the requirements for a DA or a SCA acting as an OAuth 2.0 client to register and authenticate to an authorization server. In the following clauses the generic term "client" will be used to refer to DA or SCA.

6.4.5.2 client registration

- The authorization server shall implement the FAPI 2.0 Security Profile and IETF RFC 7591 [11] OAuth 2.0 Dynamic Client Registration Protocol.
- The authorization server shall accept registration requests including client metadata values specified as claims in a software statement.
- The client may register dynamically implementing requirements specified in IETF RFC OAuth 2.0 Dynamic Client Registration Protocol providing:
  - *redirect_uris*: the redirection URI strings for use in redirect-based flows;
  - *jwks*: client's JSON Web Key Set as specified in IETF RFC 7517 [12], which contains the client's public keys, the client's registration certificate public key shall be included;
  - *jwks_uri*: URL string referencing the client's JSON Web Key Set as specified in IETF RFC 7517 [12], which contains the client's public keys, the client's registration certificate public key shall be included;
  - *token_endpoint_auth_method*: containing the value private_key_jwt;
  - *grant_types*: the grant type strings that the client can use at the token endpoint, the value "authorization_code", representing the authorization code grant type defined in OAuth 2.0 clause 4.1, shall be included;
  - *response_types*: the OAuth 2.0 response type strings that the client can use at the authorization endpoint, the value "code", representing the authorization code response type defined in OAuth 2.0 clause 4.1, shall be included;
  - *software_statement*: metadata values about the client software including the following claims:
    - *software_id*: a unique identifier string used by registration endpoints to identify the client software to be dynamically registered;
    - *client_name*: human-readable string name of the client to be presented to the end-user during authorization;
    - *client_attestation*: an attestation that the authorization server uses to evaluate whether accepting or not the registration request depending on the trust relationship the authorization server has with the issuer of the software statement, it shall include the client's registration certificate, allowing

DRAFT

authorization servers to trust the integrity of the client's digital identities and attributes, in particular the public key included in the claim jwks;

- other optional client metadata parameters defined in OpenID Connect Relying Party Metadata Choices 1.0 - draft 04 [13] that the authorization server may ignore if not supported.

- The authorization server should validate *client_attestation* evidence and reject registration if the attestation is invalid, expired, or revoked.

- Upon a successful registration request, the authorization server shall respond with an HTTP 201 "Created status" code and a body of type "application/json" with the following fields:

  - *client_id*: the OAuth 2.0 client identifier string that should not be valid for any other registered client, though an authorization server may issue the same client identifier to multiple instances of a registered client at its discretion;

  - all registered metadata about the client, including the software statement used as part of the registration request whose value shall be returned unmodified in the response.

- Upon an unsuccessful registration request, the authorization server shall return an HTTP 400 status code with content type "application/json" consisting of a JSON object (IETF RFC 8259 [i.9]), describing the error in the response body, with two members:

  - error: single ascii error code string;

  - error_description: human-readable ASCII text description of the error (optional).

- The client may be configured with pre-registered (static) client authentication data. Appropriate in the case of closed ecosystems, driving applications issued by a national authority, authorization server and client belonging to the same trust framework. The client is manually registered out-of-band with the authorization server operator, who issues:

  - client_id.

  - one of:

    - client_secret.

    - a public key, if using private_key_jwt.

    - a certificate, if using mTLS.

NOTE: The issuer of the JWT representing the software_statement can be one of:

- The API ecosystem operator: for example a vehicle OEM, mobility platform, insurance platform or fleet platform.
- The app provider’s developer portal: for example the app vendor registers the app manually once, then receives a software statement for automated DCR.
- A commercial trust registry: for example open mobility / open insurance / connected-car ecosystem registry.
- The app backend itself, if the AS already trusts that backend: This is weaker unless backed by mTLS, private key JWT, or prior onboarding.

The software statement should be signed by an entity that the authorization server already trusts. RFC 7591 does not define who must issue the statement; it defines the mechanism by which a signed statement can carry client registration claims.

DRAFT

7 Remote signatures creation service API

7.1 Introduction

In the present document the API represent the way by which any driving application can interact with any SCS conforming to the present document. The following clauses define the API that a driving application can invoke within an SCS. For any API the request and response messages pair is specified referring to CSC API [1], when possible, otherwise detailing new components defined in the present document, by means of a table where the following information are provided:

- the components name;
- the presence of the component;
- the data type of the component;
- a brief description of the component.

The value included in the column "Presence" of the tables has the meaning defined in table 1.

Table 1

Value	Description
M	The component shall be included in the request to or response from the SCS.
O	The component may be included in the request to or response from the SCS.
C	The component shall be included in the request or response based on the occurrence of certain conditions.

7.2 Service information API

7.2.1 Description

Returns information about the remote service and the list of the API methods it supports. The *info* API method referenced in CSC API [1], Clause 11.1 shall be applied and fully implemented to ensure compliance with the present document.

7.2.2 Request message

7.2.2.1 Component for requesting service information

The message for requesting the service information shall contain the components defined in CSC API [1], clause 11.1 section Input.

7.2.3 Response message

7.2.3.1 Component for responding to service information requests

The message for returning the service information shall contain the components defined in CSC API [1], clause 11.1 section Output with the addition of the components defined in Table 2.

DRAFT

Table 2

Parameter	Presence	Type	Description
oauth2FAPIsupport	O	boolean	This parameter shall be "true" if the SCS authorization server supports the FAPI 2.0 Security Profile and can • authenticate clients using private_key_jwt as specified in Section 9 of OIDC; • issue sender-constrained access tokens according to DPoP method as described in IETF RFC 9449 [25].

7.3 Credentials list API

7.3.1 Description

Returns the list of credentials of a certain user. A user may have one or multiple credentials hosted by a single RSSP.

If requested, it shall also return information about the signing certificate(s) and/or about the authorization mechanisms required to authorize the usage of the credentials for remote signing.

The *credentials/list* API method referenced in CSC API [1], clause 11.6 shall be applied and fully implemented to ensure compliance with the present specification.

7.3.2 Request message

7.3.2.1 Component for requesting credentials list

The message for requesting the list of credentials shall contain the components defined in CSC API [1], clause 11.6 section Input. The parameter *onlyValid* needs not be supported.

7.3.3 Response message

7.3.3.1 Component for responding to credentials list requests

The message for returning the requested list of credentials information shall contain the components defined in CSC API [1], clause 11.6 section Output. The parameter *onlyValid* needs not be supported if the input parameter of the same name is not supported.

7.4 Credentials info API

7.4.1 Description

Retrieves a credential and returns information about signing certificate and authorization mechanisms required to authorize the usage of the credential for remote signing.

The *credentials/info* API method referenced in CSC API [1], clause 11.7 shall be applied and may be implemented to ensure compliance with the present document.

7.4.2 Request message

7.4.2.1 Component for requesting credential information

The message for requesting the information about a credential shall contain the components defined in CSC API [1], clause 11.7 section Input.

DRAFT

7.4.3 Response message

7.4.3.1 Component for responding to credential information requests

The message for returning the service information shall contain the components defined in CSC API [1], clause 11.7 section Output.

7.5 Hash(es) signing API

7.5.1 Description

Computes the digital signature of one or multiple hash values. The driving application shall pass the authorization data that allows the signing credential usage.

The *signatures/signHash* API method referenced in CSC API [1], clause 11.13 shall be applied and fully implemented to ensure compliance with the present specification.

7.5.2 Request message

7.5.2.1 Component for requesting hash(es) signature creation

The message for requesting the hash(es) signature creation shall contain the components defined in CSC API [1], clause 11.13 section Input. The parameters *operationMode*, *validity_period* and *response_uri* need not be supported, so that only synchronous operation mode shall be supported by the SSA.

7.5.3 Response message

7.5.3.1 Component for responding to hash(es) signature creation requests

The message for returning the signed hash(es) shall contain the components defined in CSC API [1], clause 11.13 section Output. The parameter *responseID* needs not be supported if the input parameter *operationMode* is not supported.

7.6 Document(s) signing API

7.6.1 Description

Creates one or more AdES signatures. The driving application shall pass the authorization data that allows the signing credential usage.

The *signatures/signDoc* API method referenced in CSC API [1], clause 11.14 shall be applied and should be implemented to ensure compliance with the present specification.

7.6.2 Request message

7.6.2.1 Component for requesting documents signature creation

The message for requesting the hash(es) signature creation shall contain the components defined in CSC API [1], clause 11.14 section Input. The parameters *operationMode*, *validity_period* and *response_uri* need not be supported, so that only synchronous operation mode shall be supported by the SSA. The values "C", "X" and "J" that can be assigned to the parameter *signature_format* defined in the object *adesParameters* that is one of the objects that compose the object *signatureCreationRequest* need not be supported. The values "AdES-B", "AdES-T", "AdES-LT" and "AdES-LTA" that can be assigned to the parameter *conformance_level* defined in the object *adesParameters* that is one of the objects that compose the object *signatureCreationRequest* need not be supported.

DRAFT

7.6.3 Response message

7.6.3.1 Component for responding to documents signature creation requests

The message for returning the signed hash(es) shall contain the components defined in CSC API [1], clause 11.14 section Output. The parameter *responseID* needs not be supported if the input parameter *operationMode* is not supported.

7.7 Signature(s) creation polling API

7.7.1 Description

Returns the outcome corresponding to a previous signature(s) creation request when processed in asynchronous mode or the indication that the signing process has not yet been completed.

The *signatures/signPolling* API method referenced in CSC API [1], clause 11.15 shall be applied. The *signatures/signPolling* API method needs not be supported, unless signatures creation in asynchronous operation mode is supported by the SSA.

7.7.2 Request message

7.7.2.1 Component for polling previous signature creation requests

The message for requesting the responses corresponding to a previous signature(s) creation request shall contain the components defined in CSC API [1], clause 11.15 section Input.

7.7.3 Response message

7.7.3.1 Component for responding to polling previous signature creation requests

The message for returning the outcome of a previous signature creation request shall contain the components defined in CSC API [1], clause 11.15 section Output.

7.8 Credentials creation API

7.8.1 Description

The *credentials/create* API method identified in CSC API [1], clause 11.4 creates a new signing credential. If requested, it can also return the new signing certificate, the whole associated certificate chain, additional information about the signing certificate and/or information about the authorization mechanism required to authorize the access to the credential for remote signing.

This method requires credential creation authorization.

The remote signature creation service shall collect the subject data needed for the certificate for the new credential.

The *credentials/create* API method referenced in CSC API [1], clause 11.4, shall be applied and fully implemented to ensure compliance with the present specification.

7.8.2 Request message

7.8.2.1 Component for requesting credential creation

The message for requesting a credential creation shall contain the components defined in CSC API [1], clause 11.4 section Input.

DRAFT

7.8.3 Response message

7.8.3.1 Component for responding to credential creation requests

The message for returning the information of the newly created credential shall contain the components defined in CSC API [1], clause 11.4 section Output.

7.8.4 Subject distinguished name of the new credential certificate

The remote signature creation service shall support the parameter *subjectData* given in the *credentialCreationRequest* input parameter in order to construct the subject distinguished name of the certificate for the new credential.

When Rich Authorization Requests (RAR) are supported, the remote signature creation service shall support receiving the subject data needed to construct the subject distinguished name of the certificate for the new credential from the authorization server in the authorization details of the access token expressed using OIDC claims notation. When Rich Authorization Requests (RAR) are supported, the remote signature creation service may support receiving the subject distinguished name data of the certificate for the new credential from the authorization server as custom claims in the access token [i.16] or retrieving them at a token introspection endpoint [i.8]. The remote signature creation service may ignore or overwrite all or some of the subject distinguished name data provided by the parameter *subjectData* given in the *credentialCreationRequest* input parameter.

The following subject data expressed using standard OIDC claims shall be collected by the remote signature creation service: given_name, family_name, gender, birthdate, address:country. The following subject data expressed using standard OIDC claims should be collected by the remote signature creation service: sub (if provided it shall contain a subject unique identifier assigned by a government or civil authority). The following subject data expressed using standard OIDC claims may be collected by the remote signature creation service: address:locality, address:region, email.

7.9 Credentials deletion API

7.9.1 Description

The *credentials/delete* API identified in CSC API [1], clause 11.5 deletes a signing credential.

This method requires credential deletion authorization. The *credentials/delete* API method referenced in CSC API [1], clause 11.5, shall be applied and may be implemented to ensure compliance with the present specification.

7.9.2 Request message

7.9.2.1 Component for requesting credential creation

The message for requesting a credential deletion shall contain the components defined in CSC API [1], clause 11.5 section Input.

7.9.3 Response message

7.9.3.1 Component for responding to credential deletion requests

This API has no output values and the HTTP status code 204 "No Content", defined in clause 15.3.5 of IETF RFC 9110 [16], is provided in response as specified in CSC API [1], clause 11.5 section Output.

DRAFT

8 Remote signatures creation service API based on OASIS DSS-X

8.1 Introduction and General Provisions

In the present document, the API represent the way by which any client application can interact with any SCS conforming to the present document. The following clauses define the API that a client application can invoke within an SCS. For simplifying API mapping comparison, the subsequent sections present DSS-X API elements in the order of the above-cited CSC API elements (see clause 7 of the present document). The subsequent clauses also reference certain mandatory elements and thus implicitly define a profile of DSS-X with regard to the application context of the present document.

NOTE 1: Given the fact that OASIS DSS-X and CSC follow different protocol design paradigms, API elements may not match regarding all details of the abstract signature creation process. Nevertheless, and specifically due to the generic nature of DSS-X, numerous equivalent patterns exist for implementing the same approach using either protocol.

NOTE 2: DSS-X has an abstract syntax specification with a mapping to XML and an alternative mapping to JSON.

DSS-X can have various transport bindings as specified in section 8 of [4]. The present document recommends the HTTP Post binding (see section 8.1 of [4]), but also admits alternative transports.

DSS-X can have various security bindings to ensure authenticity, integrity and confidentiality of protocol messages as detailed in section 8.3 of [4]. Reliable DSS-X security bindings shall apply to all protocol messages in the context of the present document. In particular, TLS (see [7]) should supply the security base layer, while alternative and/or complementary security layers are admissible. For instance, a complementary security layer may consist in digitally signing DSS-X messages, with the chosen DSS-X syntax variant (see section 2.2.5 of [4]) determining an eligible signature format (see [8] and [9]).

DSS-X supports various options for transmitting signature activation data. The service metadata (see clauses 8.2 and 8.3 of the present document) shall provide details regarding the applicable mechanism embodied by one of the following variants:

- The *ClaimedIdentity* optional component (see section 4.4.9 of [4]) can identify the signatory regarding service calls described in clauses 8.3, 8.4, 8.5, 8.6 and 8.7 of the present document, with the *SupportingInfo* optionally carrying corresponding authorization data. This is the recommended approach when using the API in combination with the OAuth2.0 authorization framework (e.g. when using the API in the context of the European Digital Identity Wallet).

- The transport binding or an underlying security layer, or a combination of both can identify the signatory and carry authorization data. For instance, when using the API in combination with SAMLv2, an outer SAML envelope can provide identification and authorization data regarding the signatory of an inner DSS-X layer. Alternatively, an outer security layer can implicitly identify the signatory and provide authorization data. This latter approach is particularly suitable for an automated electronic seal creation service, when the service requestor is also the signatory.

- Finally, a sub-protocol that interleaves a DSS *SignRequest* and *SignResponse* scenario, may be used for identifying the signatory and providing authorization data.

8.2 Service information API

8.2.1 Description

Returns technical information about the service.

OASIS DSS-X [4] provides various options for conveying service-specific information as detailed in clause 4 of [5] that represents a complementary standard document of DSS-X.

The element name specification of clause 3.2 of [5] shall apply.

DRAFT

A metadata instance shall be as complete as necessary to enable integrators for obtaining accurate information regarding available feature support of the service.

For enabling metadata discovery, the mechanism described in clause 4 of [5] shall apply.

8.3 Credentials list API

8.3.1 Description

Returns the list of credentials of a certain user.

OASIS DSS-X [4] does not provide a specific API for requesting user credentials, which specifically protects privacy of the user by not exposing any credentials without explicit authentication or without user consent. By contrast, the metadata *AuthInfo* element (see clause 3.2 of [5]) can supply general information about the authentication mechanisms supported by the underlying signature service.

If the signer certificate is required upfront for preparing a signature request, e.g. for calculating a signer certificate hash as part of the DTBS or for specifying a PDF signature appearance based on the signer certificate subject Distinguished Name, the signature service may expose means for explicitly requesting the signer certificate. Such a request may consist of a *VerifyRequest* containing an *OptionalInputs* component with a *ReturnSignerIdentity* element (see clause 4.4.5 of [4]).

Alternatively, the response of the applied authentication protocol may supply the signer certificate.

Alternatively, the signer certificate may already be available on the client side by out-of-band means.

Alternatively, no signer certificate may be required upfront on the client side, e.g. when creating a signer certificate on the fly for single use, e.g. in the form of a validity-assured short-term certificate.

The metadata shall indicate via clause 3.1.5 of [5], which of the above-cited approaches the service supports.

8.4 Credentials info API

8.4.1 Description

Retrieves a credential and returns information about signing certificate and authorization mechanisms required to authorize the usage of the credential for remote signing.

See clause 8.3 of the present document.

8.5 Hash(es) signing API

8.5.1 Description

Computes the digital signature of one or multiple hash values.

For this purpose, the *SignRequest* with the *DocumentHash* variant of the *InputDocuments* component (see clause 4.5.5 of [4]) shall apply. The *KeySelector* component (see clause 4.4.12 of [4]) shall apply whenever the signing key cannot be determined implicitly.

8.6 Document(s) signing API

8.6.1 Description

Creates one or more AdES signatures.

DRAFT

For this purpose, the *SignRequest* with the *Document* variant of the *InputDocuments* component (see clause 4.5.3 of [4]) shall apply. For conveying more complex DTBS, particularly in the case of XAdES, the *TransformedData* variant (see clause 4.5.4 of [4]) may be more appropriate and is therefore an admissible alternative. The *KeySelector* component (see clause 4.4.12 of [4]) shall apply whenever the signing key cannot be determined implicitly.

8.7 Signature(s) creation polling API

8.7.1 Description

Returns the outcome corresponding to a previous signature(s) creation request when processed in asynchronous mode or the indication that the signing process has not yet completed.

For this purpose, the *PendingRequest* element (see clauses 4.3.5 and 7 of [4]) shall apply.

8.8 Credentials creation API

8.8.1 Description

The API *credentials/create* creates a new signing credential.

OASIS DSS-X does not provide a specific API for signer certificate creation. This happens implicitly when the corresponding *SignRequest* implementation creates the signer certificate on the fly for single use, e.g. in the form of a validity-assured short-term certificate upon user consent, while creation of long-term valid signer certificates is subject to a dedicated PKI process outside the scope of a signature creation process.

Consequently, a credential creation API is not applicable in the case of the present DSS-X profile.

DRAFT

Annex A (normative): OpenID4VP EUDIW-centric signing flow profile

A.1 Overview

This annex profiles an OpenID4VP flow in which a Driving Application acting as Wallet-Relying Party (WRP) can request the creation of a certificate and signing credential and the creation, or orchestration of the creation of, one or more qualified electronic signatures or seals from a European Digital Identity Wallet (EUDIW) that integrates a Signature Creation Application, using OpenID for Verifiable Presentations (OpenID4VP) as the transport/interaction protocols, and the CSC DM [2] and CSC data model bindings [3] for transaction data and results. The signatures creation request uses dcql_query to select an acceptable X.509 signing credential and transaction_data carrying a CSC qesRequest.

The profile focuses on:

- Requests: how a WRP formulates a credential creation or QES signing request towards the EUDIW (dcql_query + transaction_data) and optionally constrains acceptable X.509 credentials or signature parameters.
- Responses: how the EUDIW returns signatures inline or posted to response_uri within a suitable VP Token, or out-of-band to a responseURI.
- Transaction data: required JSON structures for requesting the creation of a certificate and signing credential or the creation of a QES and the complementary transaction authorization flow.

This profile assumes AdES signature creation processing (PAdES/XAdES/CAdES/JAdES) per ETSI EN 319 102-1 [i.12].

The following flow is considered:

- the Relying Party (RP/Verifier) asks the EUDIW to create one or more electronic signatures,
- the EUDIW (user-held wallet) orchestrates signature creation, collects user consent, obtains any required credentials, calls a remote QTSP/signature creation service to create the signature(s), and
- the EUDIW returns the produced signature(s) (or signed documents) to the WRP in the OpenID4VP authorization response which may be provided inline or posted to the response_uri provided in the authorization request according to the value of the response_mode parameter included in the authorization request.

The profile defined in the present annex covers only the WRP↔EUDIW interface (request + response). The internal EUDIW↔QTSP interaction is outside the scope of the profile defined in the present annex.

NOTE: The flow outlined here does not constitute the only possible approach for implementing a signing process in which requesting to an EUDIW the creation of QES and obtaining the corresponding QES in response.

A.2 Roles & Components

- Wallet-Relying Party (WRP): verifier initiating a signature transaction to the EUDIW. Uses OpenID4VP authorization request with dcql_query and transaction_data. Provides documents by reference with integrity protection or as data encoded with URLs. Receives signatures from the EUDIW.
- EUDIW: the Wallet Unit, used by the holder, that includes or has access to a SCA, implements OpenID4VP, holds or may obtain an X.509 signing credential, presents credentials, captures consent, verifies documents integrity before signing, interacts with a remote signature creation service via CSC API and returns signatures.
- QTSP/SAM/Authorization Server (out of scope here): QTSP components supporting authorization and activation of the remote signature operation. The Authorization Server processes signer authentication, EUDIW presentation or transaction approval evidence, and issues or validates the authorization data. The

DRAFT

SAM enforces the resulting authorization and activates the QSCD only for the authorized signing transaction. In case of a QES creation the SAM enforces sole control using Signature Activation Data (SAD).

- Holder: owns and uses the EUDIW, may be also referred to as signer in this annex.

This profile aligns with the EUDI Wallet Architecture and Reference Framework, which foresees wallet-based QES creation and interoperable presentation protocols.

A.3 Profile scope & transport

This profile is based on OpenID4VP with the following mandatory features.

- Signature creation requests shall be signed according to JWT-Secured Authorisation Request (JAR) as specified in IETF RFC 9101 [31].
- Digital Credentials Query Language (DCQL) shall be used to express the query for credentials or credential presentations requested from the EUDIW. DCQL shall not be used to define the signature creation request itself. The signature creation request, including the requested signature qualifier, document references or document digests, number of signatures, signature format and response handling, shall be expressed using the applicable CSC data model object and conveyed as OpenID4VP transaction_data.

NOTE: in this annex, DCQL is used to select or constrain credentials, for example an X.509 credential suitable for QES creation. The transaction_data parameter is used to convey the transaction that the signer is requested to approve, including the signature creation details.

- Holder Key Binding JWT (KB-JWT) for SD-JWT VCs shall be used.
- Signature creation requests using OpenID4VP shall include suitable transaction_data containing a base64url-encoded qesRequest object as defined in CSC data model bindings [3], clause 6.2.1. The qesRequest object shall contain the transaction data to be rendered to and approved by the signer, including the signature qualifier and the signatureRequests. If the OpenID4VP profile requires the transaction data to be associated with one or more credentials selected by the credential query, the qesRequest object shall include credential_ids. The credential_ids parameter contains opaque identifiers local to the OpenID4VP transaction and shall not be used as, or mapped directly to, CSC credentialID values.
- signatureQualifier parameter shall be present in the qesRequest, as specified in clause 9.4 of CSC DM.
- Document to be signed integrity shall be verified by the EUDIW when both href and checksum claims are present in the signatureRequests array entries.

In the flow WRP→EUDIW a signed Request Object (JAR) is provided at request_uri. EUDIW resolves it after the user opens a deep link or scans a QR code.

In order to invoke the EUDIW, a custom URL scheme openid4vp:// or haip-vp:// may be supported in addition to other ways to invoke the EUDIW.

The response_type parameter value shall be vp_token. The response_mode parameter value should be direct_post or direct_post.jwt. When either response mode is used, the authorization request shall include a response_uri parameter identifying the endpoint to which the EUDIW sends the authorization response by HTTP POST. After successful processing of the authorization response, the WRP may return a redirect instruction, where supported by the selected OpenID4VP response mode, enabling the EUDIW to redirect the user to a WRP-controlled HTTPS endpoint. The response_uri parameter and any subsequent user redirection URI are distinct and shall not be conflated.

The Authorization Request is provided using the request_uri parameter as defined in JWT-Secured Authorization Request (JAR) IETF RFC 9101 [31]. The DCQL query and response parameters are defined in Section 6 of OpenID for Verifiable Presentations 1.0.

In the flow EUDIW→WRP the response_mode=direct_post or direct_post.jwt values are used to return vp_token (and the signatures/results) in a signed POST body.

A.4 Transaction Data Types and Flows

A.4.1 Overview

The present document defines the following transaction data types and associated transaction flows:

DRAFT

- Transaction data type and flow for the creation of a certificate and signing credential (see Annex A.4.1)
- Transaction data type and flow for creation of a QES using existing certificate and signing credential (see Annex A.4.2)
- Transaction data type and flow for the creation of a QES on-the-fly (see Annex A.4.3)
- Transaction data type and flow for listing of credentials (see Annex A.4.4)
- Transaction data type for approval of QES (see Annex B)

A.4.2 Creating a certificate and signing credential

A.4.2.1 Purpose

The purpose of this transaction is to request the creation of a X.509 certificate and an associated signing credential at a RSSP, which can later on be used for the creation of qualified electronic signatures with the transaction data type and flow defined in Annex A.4.2.

A.4.2.2 transaction_data

The transaction_data element according to clause 5.1 of [22] is specified as follows:

- type: Shall be present and shall be equal to http://uri.etsi.org/19432/td/create-certificate-and-credential .
- credential_ids: Shall be present and shall contain the string "eu.europa.ec.eudi.pid.1" to reference the PID, which is used for identification of the user.
- certificate_policies: May be present and shall, if present, contain one or more strings, for which each string identifies an admissible certificate policy for the certificate, which is requested to be created.

A.4.2.3 Transaction flow

The transaction flow is outlined in the following sequence diagram:

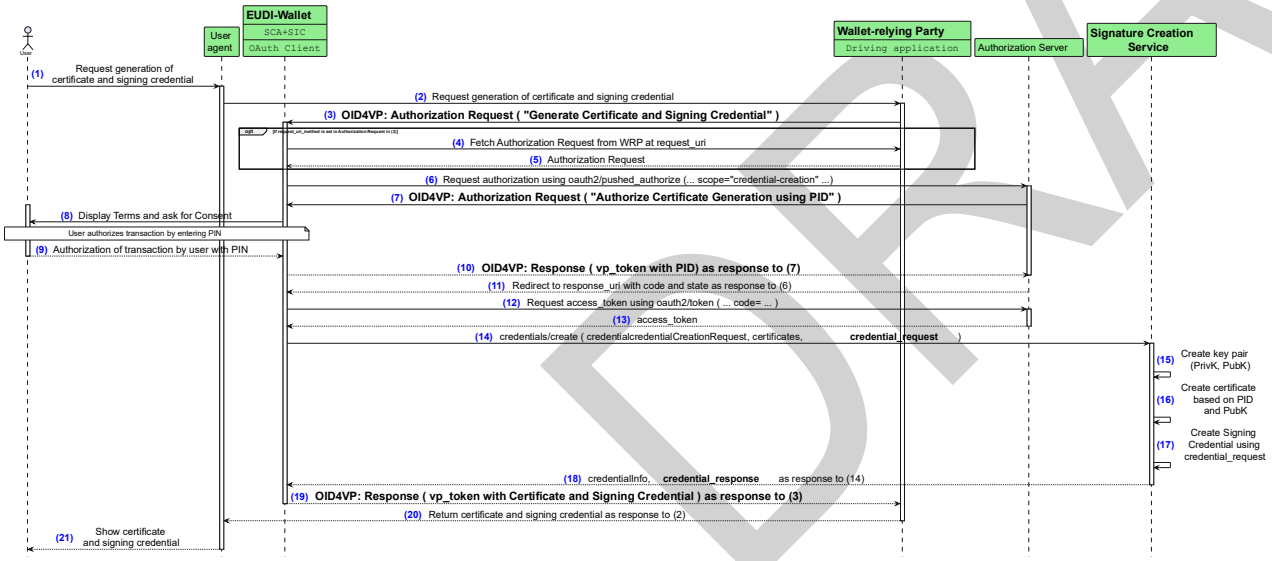

Figure 24: Transaction flow for creating a certificate and signing credential

(1) The User is requesting the generation of a certificate and signing credential.

(2) The User Agent is used to request the generation of a certificate and signing credential at the DA, which also acts as WRP.

(3) The WRP sends an Authorization Request according to [22] with the transaction_data specified in Annex A.4.1.1 above to start the transaction or a request_uri where the sealed Authorization Request can be fetched.

DRAFT

(4) If the Authorization Request sent in (3) contains a request_uri the EUDIW fetches the sealed Authorization Request.

(5) The Authorization Request is returned to the EUDIW, which can validate and process it.

(6) The EUDIW requests authorization of the transaction using the oauth2/pushed_authorize call with the following elements according to clause 8.2.2 of the CSC-API [1]:

- response_type: shall be present and the value shall be equal to “code”, as required by clause 4.1.1 of IETF RFC 6749 [17].
- client_id: shall be the client identifier according to clause 4.1.1 of IETF RFC 6749 [17], which shall be identical to the wallet_name indicated in the Wallet Instance Attestation (WIA) according to ETSI TS 119 476-3 [36].
- redirect_uri: shall be present and shall be as in of IETF RFC 6749 [17].
- scope: shall be present and shall be equal to “credential-creation”.
- code_challenge: shall be present and shall be according to IETF RFC 7636.
- code_challenge_method: shall be present and shall be as defined in IETF RFC 7636.
- state: shall be present and shall be as defined in clause 4.1.1 of IETF RFC 6749 [17].
- Account_token: shall be present and shall contain the Wallet Instance Attestation (WIA) and one or more Wallet Key Attestations according to ETSI TS 119 476-3 [36].

(7) The Authorization Server starts an Authorization Request according to [22] such that the User can Authorize the generation of the requested certificate and signing credential with the transaction_data according to Annex B containing the following element:

- terms: shall be present and shall be a documentReference object according to clause 8.3 of the CSC Data Model [2], which shall contain the child elements label, href and checksum and which points to the terms and conditions of the RSSP according to clause 6.9.4 of ETSI EN 319 411-1 [37].

(8) The EUDIW displays the terms and conditions for the certificate and requests the User to provide its consent.

(9) The User authorizes the transaction by entering the PIN.

(10) The EUDIW returns the Response with the requested PID to the Authorization Server.

(11) The Authorization Server validates the Response with the provided vp_token containing the PID, stores it and provides a redirect to the response_uri, which contains the code and state.

(12) The EUDIW requests the access_token at the token endpoint of the Authorization Server by calling oauth2/token.

(13) The Authorization Server returns the access_token to the EUDIW.

(14) The EUDIW requests the creation of a certificate and signing credential by calling credentials/create according to clause 11.4 of CSC-API [1], which contains a credential_request according to clause 8.2 of OpenID for Verifiable Credential Issuance [i.14].

(15) The SCS creates a key pair (PrivK, PubK).

(16) The SCS creates a certificate based on the PID provided in (10) and the PubK.

(17) The SCS creates a signing credential based on the credential_request provided in (14).

(18) The SCS returns the created certificate within credentialInfo and the signing credential as response to (14) within a credential_response element according to clause 8.3 of OpenID for Verifiable Credential Issuance [i.14].

(19) The EUDIW is sending the Response back to the WRP / DA with a credential_info element created from the credentialInfo element and the credential_response received in the previous step.

(20) The certificate and signing credential is returned as response to (2).

(21) The certificate and signing credential is shown to the User.

DRAFT

A.4.3 Creating a QES using existing certificate and signing credential

A.4.3.1 Purpose

The purpose of this transaction is to use a previously created X.509 certificate and an associated signing credential at a RSSP for the remote creation of a QES with the transaction data type and flow specified below.

A.4.3.2 transaction_data

The transaction_data element according to clause 5.1 of [22] is specified as follows:

- type: shall be present and shall be equal to http://uri.etsi.org/19432/td/create-qes .
- credential_ids: shall be present and shall contain the id of the previously created signing credential, which is used for authorizing the signature generation.
- signatureCreationRequests: shall be present and shall be as defined in clause 9.3 of the CSC Data Model [2].

A.4.3.3 Transaction flow

The transaction flow is outlined in the following sequence diagram:

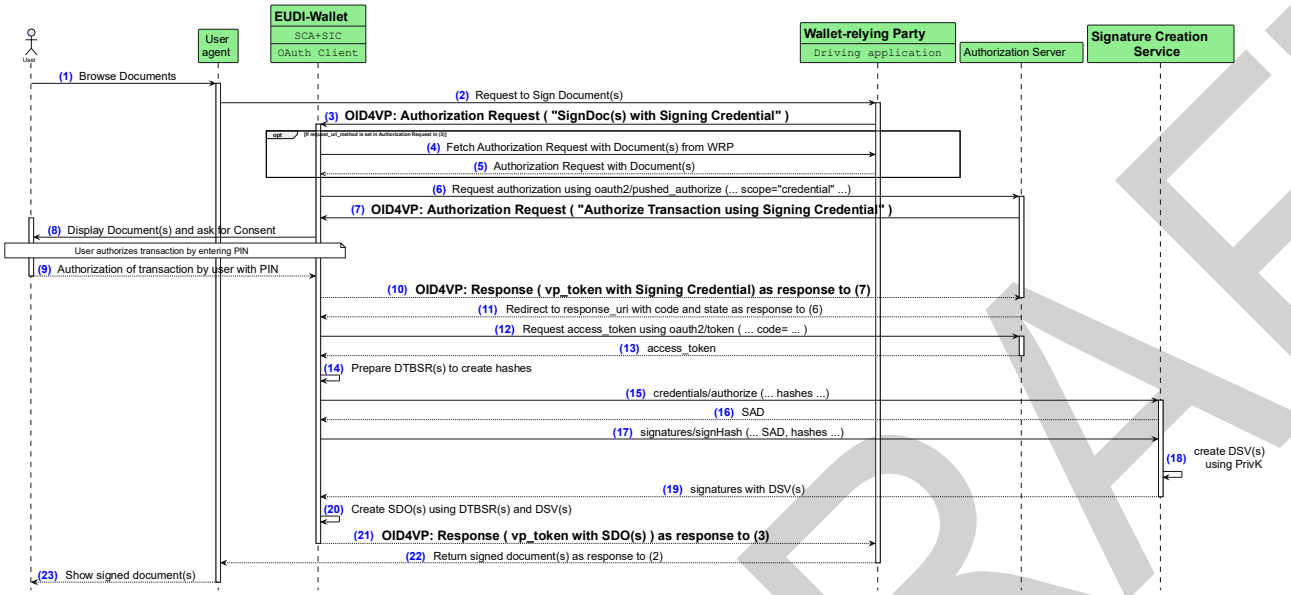

Figure 25: Transaction flow for creating a QES with signing credential

(1) The User is requesting the generation of a signature and signing credential.

(2) The User Agent is used to request the generation of qualified electronic signatures for one or more provided documents using a previously generated certificate and signing credential at the DA, which also acts as WRP.

(3) The WRP sends an Authorization Request according to [22] with the transaction_data specified in Annex A.4.2.1 above to start the transaction or a request_uri where the sealed Authorization Request can be fetched.

(4) If the Authorization Request sent in (3) contains a request_uri the EUDIW fetches the sealed Authorization Request, which contains the document(s) to be signed.

(5) The Authorization Request is returned to the EUDIW, which can validate and process it.

(6) The EUDIW requests authorization of the transaction using the oauth2/pushed_authorize call as specified in Annex A.4.1.2 step (6), whereas the scope is equal to “credential”.

(7) The Authorization Server starts an Authorization Request according to [22] such that the User can Authorize the generation of the signature with the transaction_data according to Annex B and with the essential content of the signatureCreationRequests provided in (3).

DRAFT

(8) The EUDIW displays the documents and requests the User to provide its consent.

(9) The User authorizes the signing transaction by entering the PIN.

(10) The EUDIW returns the Response to the Authorization Server with the presentation of the signing credential, which can be verified by the Authorization Server.

(11) The Authorization Server validates the Response with the provided vp_token containing the presented signature credential, stores it and provides a redirect to the response_uri, which contains the code and state.

(12) The EUDIW requests the access_token at the token endpoint of the Authorization Server by calling oauth2/token.

(13) The Authorization Server returns the access_token to the EUDIW.

(14) The EUDIW prepares the DTBSR for each document to be signed and creates the corresponding hashes.

(15) The EUDIW calls credentials/authorize with the generated hashes.

(16) The SCS returns the SAD to the EUDIW.

(17) The EUDIW sends signatures/signHash with the SAD and the hashes to the SCS.

(18) The SCS creates a DSV for each of the provided hash values provided in (15) using PrivK.

(19) The SCS returns the created DSV(s).

(20) The EUDIW is creating the SDO(s) from the DTBSR(s) and DSV(s).

(21) The EUDIW is returning the Response with the SDO(s) within the vp_token as response to (3).

(22) The signed document(s) are returned as response to (2)

(23) The signed document(s) may be shown to the User.

A.4.4 Creating a QES on-the-fly

A.4.4.1 Purpose

The purpose of this transaction is to use the PID to authorize the generation of an X.509 certificate and a QES on-the-fly within a single transaction with transaction data type and flow specified below.

A.4.4.2 transaction_data

The transaction_data element according to clause 5.1 of [22] is specified as follows:

- type: shall be present and shall be equal to http://uri.etsi.org/19432/td/create-qes-on-the-fly .
- credential_ids: shall be present and shall contain “eu.europa.ec.eudi.pid.1” to reference the PID, which is used for identification of the user, which is necessary for the creation of a certificate and the authorization of the transaction.
- signatureCreationRequests: shall be present and shall be as defined in clause 9.3 of the CSC Data Model [2].

A.4.4.3 Transaction flow

The transaction flow is outlined in the following sequence diagram:

DRAFT

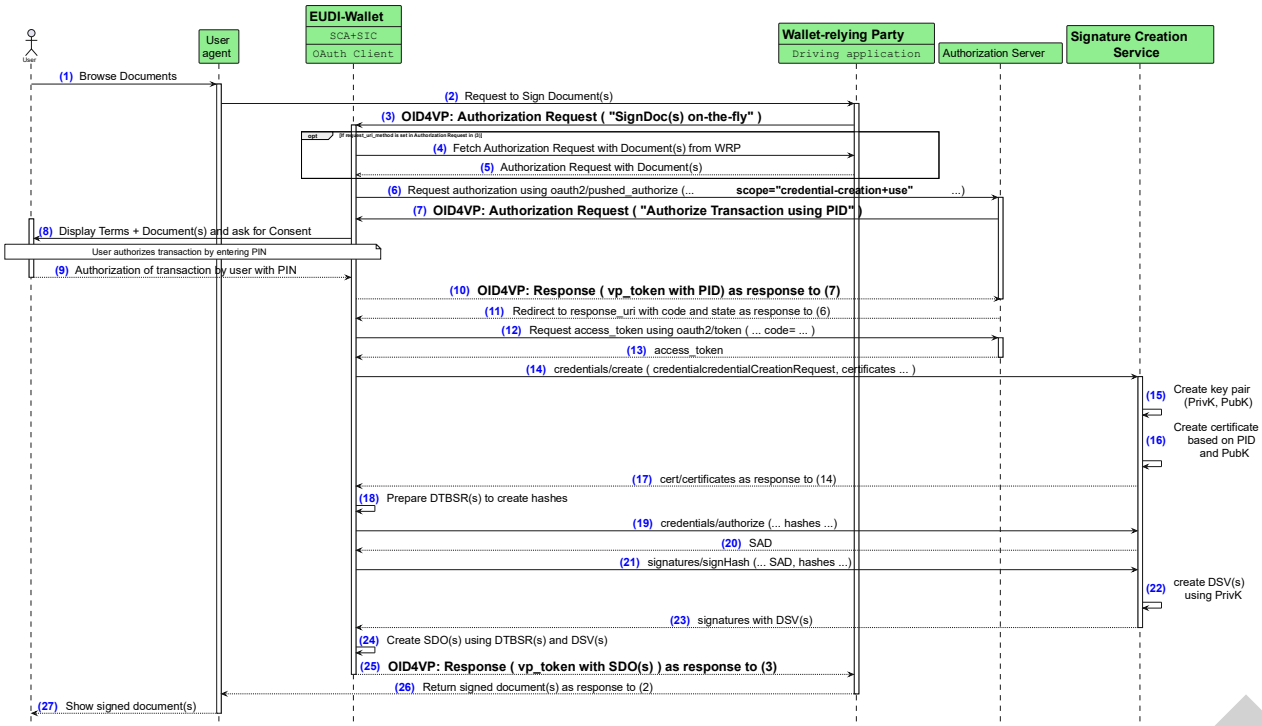

Figure 26: Transaction flow for creating a QES on-the-fly

(1) The User is browsing documents.

(2) The User Agent is used to request the generation of qualified electronic signatures on-the fly at the DA, which also acts as WRP.

(3) The WRP sends an Authorization Request according to [22] with the transaction_data specified in Annex A.4.3.1 above to start the transaction or a request_uri where the sealed Authorization Request can be fetched.

(4) If the Authorization Request sent in (3) contains a request_uri the EUDIW fetches the sealed Authorization Request, which contains the document(s) to be signed.

(5) The Authorization Request is returned to the EUDIW, which can validate and process it.

(6) The EUDIW requests authorization of the transaction using the oauth2/pushed_authorize call as specified in Annex A.4.1.2 step (6), whereas the scope is equal to “credential-creation+use”.

NOTE: The scope equal to “credential-creation+use” corresponds to the combination of the scopes “credential-create” and “credential” in the CSC-API [1].

(7) The Authorization Server starts an Authorization Request according to [22] such that the User can Authorize the generation of the signature with the transaction_data according to Annex B and with the essential content of the signatureCreationRequests provided in (3).

(8) The EUDIW displays the terms and conditions for the on-the-fly certificate and signature creation and the documents to sign and requests the User to provide its consent.

(9) The User authorizes the signing transaction by entering the PIN.

(10) The EUDIW returns the response to the Authorization Server including the PID.

(11) The Authorization Server validates the response with the provided vp_token containing the PID and the signature authorization, stores it and provides a redirect to the response_uri, which contains the code and state.

(12) The EUDIW requests the access_token at the token endpoint of the Authorization Server by calling oauth2/token.

(13) The Authorization Server returns the access_token to the EUDIW.

DRAFT

(14) The EUDIW requests the creation of a certificate by calling credentials/create according to clause 11.4 of CSC-API [1].

(15) The SCS creates a key pair (PrivK, PubK).

(16) The SCS creates a certificate based on the PID provided in (10) and the PubK.

(17) The SCS returns cert/certificates as response to (14).

(18) The EUDIW prepares the DTBSR for each document to be signed and creates the corresponding hashes.

(19) The EUDIW calls credentials/authorize with the generated hashes.

(20) The SCS returns the SAD to the EUDIW.

(21) The EUDIW sends signatures/signHash with the SAD and the hashes to the SCS.

(22) The SCS creates a DSV for each of the provided hash values provided in (19) using PrivK.

(23) The SCS returns the created DSV(s).

(24) The EUDIW is creating the SDO(s) from the DTBSR(s) and DSV(s).

(25) The EUDIW is returning the Response with the SDO(s) within the vp_token as response to (3).

(26) The signed document(s) are returned as response to (2).

(27) The signed document(s) may be shown to the User.

A.4.5 List credentials

A.4.5.1 Purpose

The purpose of this transaction is to list the available credentials with transaction data type and flow specified below.

A.4.5.2 transaction_data

The transaction_data element according to clause 5.1 of [22] is specified as follows:

- type: Shall be present and shall be equal to http://uri.etsi.org/19432/td/list-credentials
- credential_ids: Shall be present and shall contain “local” to indicate that the authorization is performed locally within the EUDIW.

A.4.5.3 Transaction flow

The transaction flow is outlined in the following sequence diagram:

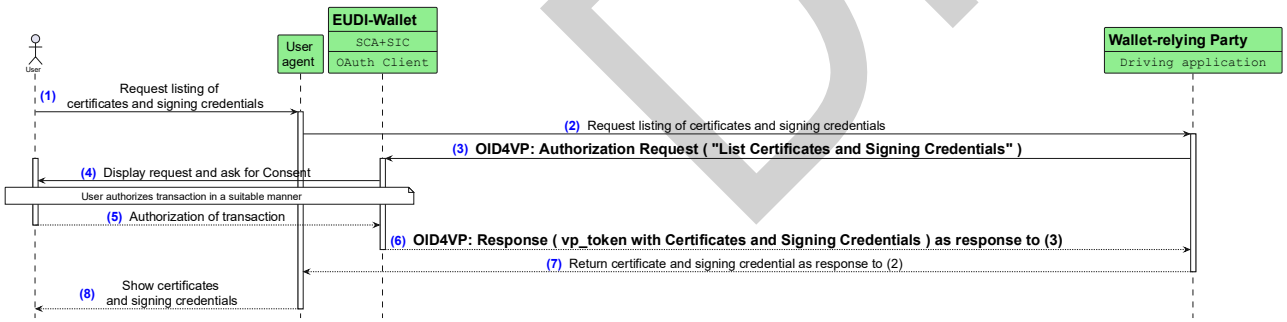

Figure 27: Transaction flow for listing of available signing credentials

(1) The User is requesting the listing of available certificates and signing credentials.

(2) The User Agent is used to request the listing of certificates and signing credentials.

(3) The WRP sends an Authorization Request according to [22] with the transaction_data specified in Annex A.4.4.1 above to start the transaction.

(4) The EUDIW displays the request and requests the User to provide its consent.

(5) The User authorizes the transaction by entering the PIN.

(6) The EUDIW returns the Response to the WRP within a vp_token which contains a credentialIDs and a credentialInfos element according to clause 11.6 of the CSC-API [1].

(7) The list of certificates and signing credentials is returned as response to (2).

(8) The list of certificates and signing credentials may be shown to the User.

A.5 High-level interaction models

A.5.1 EUDIW: RP ↔ EUDIW, EUDIW ↔ QTSP

1) WRP sends an OpenID4VP signature creation request to EUDIW that includes:

   - a dcql_query selecting acceptable X.509 credentials, and
   - transaction_data carrying qesRequest (base64url-encoded JSON).

2) EUDIW is invoked via openid4vp:// or haip-vp:// (same-device) or QR code (cross-device).

3) EUDIW renders transaction data, obtains consent, and interacts with the remote signature creation service to create the signatures.

4) EUDIW returns inline the signatures in the VP Token or posts out-of-band to the response_uri provided in the request.

A.5.2 QTSP: QTSP ↔ EUDIW

The QTSP (providing remote signature creation service) requests the EUDIW to present an attestation that includes qesApproval, cryptographically binding approval to the transaction data; the AS uses it to authorize the SCS API signing call (see Annex B).

A.6 Request construction (RP → EUDIW)

A.6.1 Introduction

The RP shall use OpenID4VP 1.0 and prepare a signed presentation request integrity-protected (JAR) with the parameters specified in the following sub-clauses. The RP shall redirect the user-agent to the EUDIW via openid4vp:// or haip-vp:// (same-device) or QR code (cross-device). The EUDIW retrieves the presentation request object (embedded or via request_uri).

A.6.2 Required parameters (OpenID4VP layer)

- response_type parameter value shall be vp_token. The response_mode parameter value should be direct_post or direct_post.jwt for the flows described in this annex. The value direct_post.jwt should be used where integrity protection of the OpenID4VP authorization response is required. The fragment response mode should not be used for QES transaction flows unless a specific profile explicitly permits it and the applicable security and privacy requirements are met. In same-device flows, the OpenID4VP authorization request may be conveyed to the EUDIW through the Digital Credentials API where supported. Use of the Digital Credentials API does not change the transaction_data requirements defined in this annex.

DRAFT

- client_id, redirect_uri, state, nonce, credential_ids, dcql_query and transaction_data parameters shall be included as specified in OpenID4VP. Credential selection should be expressed using dcql_query. Where another OpenID4VP credential query mechanism, such as query, is used, its use shall be explicitly profiled and shall provide equivalent semantics for selecting acceptable credentials. The examples in this annex use dcql_query.

A.6.3 dcql_query (selecting an acceptable signing certificate)

The WRP shall use the format identifier "https://cloudsignatureconsortium.org/2025/x509" and optionally constrain policies, keys, or fingerprints. A request for proof of possession of an X.509 certificate shall include transaction_data for creating electronic signatures or seals using the queried certificate.
A non-normative example DCQL query constraining certificate policies by using the X.509 credential format is:

```
{
   "dcql_query": {
      "credentials": [{
         "id": "xyz123",
         "format": "https://cloudsignatureconsortium.org/2025/x509",
         "meta": {
            "certificatePolicies": ["0.4.0.2042.1", "0.4.0.194112.1"]
         }
      }]
   },
   "transaction_data": [
      "ew0KICAgICJ0eXBlIjogImh0dHBzOi8vY2xvdWRzaWduYXR1cmVjb25zb3J0aXVtLm9yZy8yMDI1Lw..."
   ]
}
```

Another non-normative example DCQL query constraining certificate fingerprints by using the X.509 credential format is:

```
{
   "dcql_query": {
      "credentials": [{
         "id": "xyz123",
         "format": "https://cloudsignatureconsortium.org/2025/x509",
         "meta": {
            "certificateFingerprints": [{
               "hashValue": "Jal8JexCNec2FBjKNGNM6WSiiE44NWjbvJBHav+vCXU=",
               "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
            },
            {
               "hashValue": "ax8xsy0gdqyJtxmcw8xgZ85DbiC905oKdYBQspddxmk=",
               "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
            }]
         }
      }]
   },
   "transaction_data": [
      "ew0KICAgICJ0eXBlIjogImh0dHBzOi8vY2xvdWRzaWduYXR1cmVjb25zb3J0aXVtLm9yZy8yMDI1Lw..."
   ]
}
```

DRAFT

Requirements:

- the dcql_query shall contain a credential entry with format: "https://cloudsignatureconsortium.org/2025/x509" and optionally certificatePolicies or certificateFingerprints parameters in the meta field applying x509MetadataQuery semantics as defined in CSC data model bindings [3] clause 8.1.

A.6.4 transaction_data (the QES transaction)

transaction_data shall contain a base64url-encoded UTF-8 JSON string whose decoded content is a qesRequest object specifying type, signatureQualifier, and at least one signatureRequests entry as defined in CSC data model bindings [3] clause 6.2.1.

The transaction_data type identifier for requesting QES is "https://cloudsignatureconsortium.org/2025/qes".

The parameter signatureRequests specifying an array of signatureRequest shall be included in the transaction data.

The parameter signatureQualifier shall be included in any signatureRequest object specifying the value "eu_eidas_qes" in order to indicate the applicable trust framework.

Any signatureRequest object shall include one of the following objects:

- documentData as specified in clause 8.1 of CSC DM
- documentReference as specified in clause 8.3 of CSC DM

The parameter label should be specified in documentData and documentReference objects in order to provide a human-readable description of the document to be signed to the EUDIW holder.

The following is a non-normative example of a transaction_data after base64url decoding:

```
{
   "type": "https://cloudsignatureconsortium.org/2025/qes",
   "credential_ids": ["xyz123"],          // optional, protocol-dependent
   "signatureRequests": [{
      "label": "Example Terms of Service",
      "access": { "type": "public" },
      "href": "https://public.rp-cdn.example/terms-and-conditions.pdf",
      "checksum": {
         "value": " HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
         "algorithmOID": "2.16.840.1.101.3.4.2.1"
      },
      "signature_format": "P",
      "conformance_level": "AdES-B-B",
      "signed_envelope_property": "Certification",
      "signAlgo": "1.2.840.113549.1.1.1",
      "signatureQualifier": "eu_eidas_qes",
      "responseURI": "https://example.com/signatureResponse/123/"
   }]
}
```

Requirements:

- type shall be https://cloudsignatureconsortium.org/2025/qes;
- signatureQualifier shall be present and shall indicate eu_eidas_qes (or the appropriate variant specifically defined by the SCSP) for qualified signatures creation requests;
- each signatureRequests shall be a CSC DM signatureRequest with documentReference semantics (href, access, checksum) or documentData semantics (document, documentType) and include adesParameters semantics. Data URLs shall be supported; media types/size may be restricted;

DRAFT

- checksum parameter uses hash object syntax as defined in clause 7.4 of CSC DM;
- credential_ids may appear to bind the transaction to the credential selected in dcql_query (different from CSC API credentialID at SCSP);
- responseURI shall be included if it is requested to post the signatures to this endpoint before returning the (empty) credential presentation. Otherwise, inline response in VP Token will be expected.

Access control for document references in cross-device flows is optional and profile-specific. Where a document reference is transferred through a QR code or another cross-device engagement mechanism, implementations should not rely solely on a one-time password to mitigate shoulder-surfing or relay attacks.

The preferred approach is to use a short-lived, single-use, integrity-protected request reference, such as a request_uri referring to a signed request object, together with document integrity protection, transaction binding, and explicit user confirmation. The document reference should include an integrity value, such as a checksum or document digest, and the EUDIW shall verify the integrity of the retrieved document before presenting the transaction for approval.

An OTP-based access method may be used as an additional user-mediated release mechanism, for example:

```
{
 "access": {
  "type": "OTP",
  "oneTimePassword": "51623"
 }
}
```

Where OTP is used, the EUDIW shall render the OTP to the user and the RP shall release the protected resource only after receiving the correct OTP through an authenticated RP-controlled channel. The OTP shall be short-lived, single-use, bound to the specific transaction and document reference, and shall not replace request integrity protection, nonce validation, transaction binding, or document checksum verification.

A.7 Response handling (EUDIW → WRP)

A.7.1 QES response in VP Token

If no QES response endpoint is specified in the responseURI parameter of qesRequest, the EUDIW shall return the QES result in the OpenID4VP authorization response, inline, where supported by the selected response mode and size constraints, or posting the OpenID4VP authorization response to the OpenID4VP response_uri when response_mode is direct_post or direct_post.jwt. The QES result in the OpenID4VP authorization response shall be represented using the qesResponse object defined in CSC data model bindings [3], clause 6.2.2. The qesResponse object shall contain documentWithSignature when the signature is embedded in the signed document, for example for PAdES, or signatureObject when the signature is detached or otherwise not embedded in the document.

If response_mode=direct_post is set, the EUDIW POSTs response parameters. If, instead, response_mode=direct_post.jwt is set, the EUDIW still POSTs, but places all response parameters inside a signed JWT carried in a single form field named response.

The following is a non-normative example of a QES response from an EUDIW:

```
{
   "xyz123": {
      "qes": {
         "documentWithSignature": ["<base64-encoded document 1>", "<base64-encoded document 2>"]
      // or
      // "signatureObject": ["<base64-encoded detached signature>"]
```

DRAFT

```
        }
    }
}
```

documentWithSignature is used when the signature is embedded (e.g. PAdES), signatureObject otherwise. Values are base64 (padded) by CSC API convention (even though OpenID4VP typically uses base64url).

A.7.2 Out-of-band (HTTP POST to responseURI)

If responseURI is present in the qesRequest, the EUDIW shall POST a JSON qesResponse to that HTTPS endpoint, using HTTP/1.1 and Content-Type: application/json, then return an empty credential response in the vp_token. Example response body:

```
{
   "documentWithSignature": ["<base64-encoded document>"]
}
```

The WRP server should accept only the first successfully received response.

A.8 EUDIW processing

After receiving the `dcql_query` and the base64url-encoded `qesRequest` string inside the OpenID4VP authorization request, the EUDIW shall validate:

- signature of presentation request
- nonce freshness
- that audience matches EUDIW
- expiry validity
- credential types and disclosure constraints
- presentation request integrity
- transaction_data that shall decode to a valid qesRequest
- presence and recognition of signatureQualifier
- that the available signing credential is capable of producing a QES with the specified signatureQualifier; if not, the EUDIW shall abort.

The EUDIW shall display:

- RP identity
- requested credential types
- requested attributes or claims
- any selective disclosure details
- a clear indication that the transaction creates a qualified electronic signature or seal
- the trust framework identified by signatureQualifier
- signature format/conformance level of the requested QES
- the label of each document, or a clear indication that no label is provided
- whether document integrity has been verified
- the URI to which the vp_token including the requested QES will be sent, if response_uri is specified

The EUDIW shall authenticate the user, display a QES consent screen (with doc labels, OTPs, format/conformance, hash algorithm, etc.), and, upon explicit consent from the holder, invoke the QTSP (as specified in the present document) with the hashes (DTBSRs) and the requested signature parameters (policy/format/alg) requesting QES creation.

The EUDIW should provide a preview or rendering of the to-be-signed document(s) where technically feasible, to allow the signer to verify the content before approving and should log transaction data and the user's approval/rejection.

DRAFT

A.9 Complementary approval flow

The SCSP AS shall require a separate qesApproval bound to the transaction data. See Annex B.

A.10 Security & privacy requirements

- JWT signing and issuer validation: if an RP transports a CSC DM signatureRequest as a JWT, it shall be signed and the EUDIW shall validate signature and issuer authorization.
- Checksum: use resources integrity with SHA-256 or stronger; EUDIW verifies before signing.
- Encoding rules: OpenID4VP parameters (transaction_data) are base64url encoded; CSC API data elements inside remain base64 (padded).
- AdES: signature formats and conformance levels follow ETSI EN 319 102-1 [i.12]; server signing enforces policies, timestamps, etc., per TSP configuration.
- SAM & SAD: sole control is enforced per EN 419 241-1 [6]; SAM validates SAD provenance and integrity.

A.11 Conformance checklist (RP)

1) Use OpenID4VP authorization request with vp_token response type and include dcql_query + transaction_data (qesRequest).
2) Encode transaction_data as base64url of the JSON object; include signatureQualifier = eu_eidas_qes.
3) Provide documents by href + checksum (SRI).
4) Choose QES response in authorization response vs out-of-band by omitting or setting responseURI.
5) Constrain acceptable QCerts via dcql_query.meta (policies, keys, fingerprints).

A.12 Examples

A.12.1 Authorization request (core parameters)

```
GET /authorize?
response_type=vp_token&
client_id=https%3A%2F%2Frp.example%2Fclient&
redirect_uri=https%3A%2F%2Frp.example%2Fcb&
state=af0ifjsldkj&nonce=n-0S6_WzA2Mj&
dcql_query=eyJjcmVkZW50aWFscyI6W3siaWQiOiJ4eXoxMjMiLCJmb3JtYXQiOiJodHRwczovL2Nsb3Vkc2lnbmF0dXJlY29uc
29ydGl1bS5vcmcvMjAyNS94NTA5IiwibWV0YSI6eyJjZXJ0aWZpY2F0ZVBvbGljaWVzIjpbIjAuNC4wLjIwNDIuMSIsIjAuNC4wL
jE5NDExMi4xIl19fV19&
transaction_data=ew0KICAidHlwZSI6ICJodHRwczovL2Nsb3Vkc2lnbmF0dXJlY29uc29ydGl1bS5vcmcvMjAyNS9xZXMiLA0
KICAic2lnbmF0dXJlUmVxdWVzdHMiOiBbDQogICAgew0KICAgICAgImxhYmVsIjogIkV4YW1wbGUgVG9TIEp1bHktMjAyNSIsDQo
gICAgICAiYWNjZXNzIjogeyAidHlwZSI6ICJwdWJsaWMiIH0sDQogICAgICAiaHJlZiI6ICJodHRwczovL3B1YmxpYy5ycC1jZG4
uZXhhbXBsZS90ZXJtcy1hbmQtY29uZGl0aW9ucy5wZGYiLA0KICAgICAgImNoZWNrc3VtIjogInNoYTI1Ni1IWlF6Wm1NQUlXZWt
mR0gwL1pLVzFuc2R0MHhnM0g2YlpZenRnc01UTHcwPSIsDQogICAgICAic2lnbmF0dXJlX2Zvcm1hdCI6ICJQIiwNCiAgICAgICJ
jb25mb3JtYW5jZV9sZXZlbCI6ICJBZEVTLUItQiIsDQogICAgICAic2lnbmVkX2VudmVsb3BlX3Byb3BlcnR5IjogIkNlcnRpZml
jYXRpb24iLA0KICAgICAgInNpZ25hdHVyZVF1YWxpZmllciI6ICJldV9laWRhc19xZXMiLA0KICAgICAgInNpZ25BbGdvIjogIjE
uMi44NDAuMTEzNTQ5LjEuMS4xIg0KICAgIH0NCiAgXQ0KfQ
```

(Whitespace added for readability.)

A.12.2 Example 'qesRequest' (transaction_data before base64url encoding)

```
{
   "type": "https://cloudsignatureconsortium.org/2025/qes",
   "credential_ids": ["qes-cert-1"],
   "signatureRequests": [
```

```
    {
       "label": "Service Agreement #2025-09",
       // Integrity pinning via hash-like object checksum (base64)
       "checksum": {
          "value": "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
          "algorithmOID": "2.16.840.1.101.3.4.2.1"
       },
       // Where the wallet fetches the original document
       "href": "https://protected.rp.example/contracts/2025-09-01.pdf?token=...",
       "signature_format": "P",
       "conformance_level": "AdES-B-B",
       "signed_envelope_property": "Certification",
       "signatureQualifier": "eu_eidas_qes",
       "signAlgo": "1.2.840.113549.1.1.1"
    },
    {
       "label": "Annex A - JSON config",
       // Provide the object inline; wallet still verifies checksum
       "href": "data:application/json; base64,eyJleGFtcGxlS2V5IjoiZXhhbXBsZVZhbHVlIn0K",
       "signature_format": "J",
       "conformance_level": "AdES-B-B",
       "signed_envelope_property": "Attached",
       "signAlgo": "1.2.840.113549.1.1.1",
       "checksum": {
          "value": "cuKv8Ee9H/rQsteQ1MQZ2Ld2ERXRkkulihFh3/XOXFQ=",
          "algorithmOID": "2.16.840.1.101.3.4.2.1"
       },
       "signatureQualifier": "eu_eidas_qes"
       // OPTION A: inline delivery (omit responseURI) -> wallet returns QES inside vp_token
       // "responseURI": undefined
       // OPTION B: out-of-band delivery (recommended for large PDFs):
       // "responseURI": "https://rp.example/qes/receive"
       }]
}
```

Notes & rules:

- input mode:
  - by-reference: the EUDIW fetches the document and computes DTBSR itself; integrity is assured via sha256 over the fetched bytes;
  - inline: large payloads discouraged; if used, provide base64, mime, size.
- return mode:
  - embedded → EUDIW returns PAdES embedded PDF to the RP (by reference or inline);
  - detached → EUDIW returns a detached CMS signature over DTBSR. (PAdES/XAdES/CAdES policies follow ETSI EN 319 102-1 [i.12]);
  - when responseURI is used, the wallet POSTs the `qes` response to that endpoint, mutually authenticated TLS is recommended.

A.12.3 VP Token (inline) - PAdES result

```
{
```

```
   "xyz123": {
      "qes": {
         "documentWithSignature": ["<base64-encoded PDF with embedded PAdES>"]
      }
   }
}
```

A.12.4 Out-of-band POST to responseURI

POST /signatureResponse/123/ HTTP/1.1
Host: rp.example
Content-Type: application/json

```
{
"documentWithSignature": ["<base64-encoded PDF with embedded PAdES>"]
}
```

The EUDIW returns an empty presentation to the OpenID4VP flow.

Annex B (normative): EUDIW QES creation approval profile

B.1 Overview

This profile specifies how a QTSP's remote signature creation service can request approval for Qualified Electronic Signature/Seal creation from an EUDI Wallet (EUDIW) using the OpenID for Verifiable Presentations (OpenID4VP) protocol and the CSC data model bindings [3]. It defines requests, responses, and the transaction_data structure that binds the user's consent to concrete data to be signed and to the target credential/remote QSCD operation.

In this profile, the SCA may be located in the EUDIW, in a wallet-controlled component, in the QTSP or signature creation service, or in another component acting on behalf of the Driving Application; this annex focuses on the EUDIW approval evidence used by the Authorization Server or SAM and does not mandate a specific SCA placement.

It aligns with:

- CSC DM, for signatureCreationApproval object and document digests, and CSC data model bindings [3] defining https://cloudsignatureconsortium.org/2025/qes-approval transaction data type identifier and qesApproval data component.
- EN 419 241-1 [6] (SAM, SAD and sole control) as architectural baseline for remote QSCD; the approval is used by the AS/SAM to authorize the signature.
- ETSI EN 319 102-1 [i.12] for AdES creation/validation procedures referenced by QTSP policies.

B.2 Roles and architecture

- QTSP Remote Signing Server: runs Authorization Server (AS) + SAM in front of a QSCD. The AS is the Verifier in OpenID4VP and asks the EUDIW for a presentation of a credential containing QES Approval. The SAM uses the token obtained with the code provided by the AS after successful validation of qesApproval to authorize the key operation. The QTSP creates and issues the attestation whose presentation is requested to EUDIW to authorize signatures creation.
- EUDI Wallet: receives the approval request (OpenID4VP), renders the transaction data, collects user consent, and returns a presentation embedding qesApproval per credential format (ISO mdoc or SD-JWT VC).
- QSCD: Performs the private key operation once the SAM validates the approval and policy constraints.

DRAFT

B.3 Signature authorization attestation

The QTSP managing SCS may create and issue a “service user attestation” as an attestation of attributes in mdoc or SD-JWT VC format with qesApproval support.

Recommended minimal information:

Attribute	Purpose
iss	QTSP identifier
userName	Wallet holder pseudonymous identifier (NOT PID)
qesApproval	It should be "org.cloudsignatureconsortium.dm.1"
credentialID	Reference to the signing certificate or key to be used
valid_for	One signing operation or limited timeframe
aud	The Authorization Server / Signing Service
nonce / challenge	Prevent replay

It ensures binding to the certificate or signing key.

Two recommended bindings:

(A) Binding to the signing certificate including one of:

- signing certificate hash (SHA 256)
- SKID / AKID
- public key hash

(B) Binding to the wallet instance (optional) including a proof-of-possession element similar to:

- jwks (as used in WUA)

This prevents forwarding or misuse.

B.4 Profile scope & transport

The profile makes use of OAuth2 Authorization Request / Response as defined by OpenID for Verifiable Presentations 1.0.

In the flow SCSP →EUDIW a signed Request Object (JAR) is provided at request_uri. EUDIW resolves it after the user opens a deep link or scans a QR code.

In order to invoke the EUDIW, a custom URL scheme openid4vp:// may be supported in addition to other ways to invoke the EUDIW.

The response_type parameter value shall be vp_token. The response_mode parameter value should be direct_post or direct_post.jwt. The SCSP shall return redirect_uri in response to the HTTP POST request from the EUDIW, where the EUDIW redirects the user to, as defined in Section 8.2 of OpenID for Verifiable Presentations 1.0.

The Authorization Request is sent using the request_uri parameter as defined in JWT-Secured Authorization Request (JAR) IETF RFC 9101 [31].

B.5 Profile identifiers

- Approval transaction data type: https://cloudsignatureconsortium.org/2025/qes-approval (qesApprovalRequest).
- Approval result name: qesApproval (a base64-encoded hash digest bound to the original request).
- Credential format identifiers (for the presented credential carrying qesApproval), e.g. ISO mdoc or SD-JWT VC as defined in CSC data model bindings [3].

DRAFT

B.6 Data model for approval

B.6.1 Introduction

The SCSP shall use OpenID4VP 1.0 and send an OAuth/OIDC authorization request with the parameters specified below.

B.6.2 qesApprovalRequest

The client_id of the OpenID4VP request object shall identify the entity acting as Wallet-Relying Party for the QES approval request, typically the SCSP or QTSP Authorization Server. The client_id shall be resolvable, according to the applicable trust framework or OpenID4VP client identifier scheme, to trusted metadata enabling the EUDIW to authenticate the requester, verify the request object signature, determine the applicable trust-framework status, and identify the endpoints and keys used for the QES approval flow.

NOTE: The client_id identifies the OpenID4VP requester and not the signing credential or the CSC credentialID.

The DCQL query shall request a credential in SD-JWT VC format whose issuance supports the org.cloudsignatureconsortium.dm.1.qesApproval claim as defined in CSC data model bindings [3] Section 7.2.1.2.

The transaction_data for requesting authorization to a signature creation shall include a base64url-encoded qesApprovalRequest object as defined in CSC data model bindings [3], clause 7.1.1 specifying at minimum type, signatureQualifier, documentInfos, and hashAlgorithmOID parameters.

A signatureCreationApproval object, specified in CSC DM clause 10.1, representing information required by the signer to give consent to a signature creation, shall be included in any qesApprovalRequest object. Differently from signatureCreationApproval object specification stated in CSC DM clause 10.1 the parameter documentDigests is defined as Array of union of documentInfo and documentReference.

The transaction data type identifier for approving Qualified Electronic Signatures or seals (QES) creation is: "https://cloudsignatureconsortium.org/2025/qes-approval". The transaction data shall be associated with a credential for approval of QES creation that can be identified by an x509MetadataQuery object as specified in CSC data model bindings [3], clause 8.1.

The following is a non-normative example of a transaction_data after base64url decoding

```
{
   "type": "https://cloudsignatureconsortium.org/2025/qes-approval",
   "credential_ids": ["xyz123"],
   "credentialID": "GX0112348",
   "signatureQualifier": "eu_eidas_qes",
   "numSignatures": 2,
   "documentDigests": [{
      "label": "Example Contract",
      "hash": "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
      "hashType": "sodr",
      "access": { "type": "OTP", "oneTimePassword": "51623" },
      "href": "https://protected.example/doc-01.pdf?token=...",
      "checksum": {
         "value": " HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
         "algorithmOID": "2.16.840.1.101.3.4.2.1"
      }
   },
   {
      "label": "Terms of Service",
      "hash": "HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
      "hashType": "sodr",
      "access": { "type": "public" },
```

DRAFT

```
      "href": "https://public.example/tos.pdf",
      "checksum": {
         "value": " HZQzZmMAIWekfGH0/ZKW1nsdt0xg3H6bZYztgsMTLw0=",
         "algorithmOID": "2.16.840.1.101.3.4.2.1"
      }
   }],
   "hashAlgorithmOID": "2.16.840.1.101.3.4.2.1"
}
```

NOTE 1: credential_ids binds to an EUDIW-local identifier per OpenID4VP (distinct from the SCSP's credentialID).

NOTE 2: signatureCreationApproval, defined in CSC DM, includes the QTSP credentialID (if known) or at least signatureQualifier, numSignatures, and an array of documentDigests with the digests and optional href/checksum/access for render and integrity checks. hashAlgorithmOID identifies the digest algorithm.

NOTE 3: hashType values like sodr/dtbsr follow CSC DM semantics; integrity checksum is base64 and includes algorithm prefix per CSC DM.

B.6.3 Computing qesApproval (EUDIW)

- The EUDIW computes qesApproval = base64(hash(original UTF-8 qesApprovalRequest)); the hash algorithm is the request's hashAlgorithmOID. The encoding of hash input preserves the original UTF-8 JSON (decode base64url if transaction_data came that way).
- Embedding depends on the approval credential format:
  - ISO/IEC 18013-5 mdoc [i.21]: put qesApproval as a DeviceSigned data element under namespace org.cloudsignatureconsortium.dm.1, identifier qesApproval, value type bstr of the raw digest (no base64 inside CBOR).
  - SD-JWT VC [i.13]: include a top-level claim org.cloudsignatureconsortium.dm.1.qesApproval in the Key Binding JWT; value is a String (base64 of the digest).

B.7 OpenID4VP request (QTSP → Wallet)

The SCSP's AS, acting as Verifier, sends an OpenID4VP authorization request:

- response_type=vp_token; choose response_mode (fragment, direct_post, or direct_post.jwt) for same-device vs cross-device.
- include dcql_query selecting the approval credential to be presented (e.g. a service attestation issued to the user that supports qesApproval). The query can request specific claims/doctype needed to carry qesApproval.
- include transaction_data = base64url (JSON qesApprovalRequest as in clause B.6.2).

Example (decoded)

```
{
   "dcql_query": {
      "credentials": [{
         "id": "qscd_service_attestation",
         "format": "mso_mdoc",
         "meta": { "doctype_value": "com.example.service.1.attestation" },
         "claims": [
            { "path": ["com.example.service.1", "userName"], "values": ["willeke"] },
            { "path": ["com.example.service.1", "credentialID"], "values": ["GX0112348"] },
            { "path": ["org.cloudsignatureconsortium.dm.1", "qesApproval"] }
         ]
      }]
```


DRAFT

```
  },
  "transaction_data": [ "<base64url of qesApprovalRequest JSON>" ]
}
```

B.8 EUDIW response (EUDIW → QTSP)

B.8.1 EUDIW validation

The EUDIW shall validate:

- Signature of Presentation Request. The EUDIW shall authenticate the OpenID4VP requester before processing the qesApprovalRequest. The client_id in the request object shall identify the QTSP, SCSP Authorization Server, or other entity acting as Wallet-Relying Party for the QES approval request. The EUDIW shall resolve the client_id according to the applicable trust framework or OpenID4VP client identifier scheme to trusted metadata containing the requester's identity, trust-framework status, endpoints and signing keys. The EUDIW shall verify the request object signature against the keys obtained from such trusted metadata and shall abort the transaction if the requester cannot be authenticated or is not authorized to request QES approval.
- Nonce freshness
- Audience validation. The EUDIW shall verify that the OpenID4VP request object is intended for processing by the EUDIW. The aud value, where present or required by the selected OpenID4VP profile, shall identify the EUDIW, the wallet-facing endpoint, or another audience identifier defined by the applicable trust framework for EUDIW-bound requests. The EUDIW shall reject the request if the aud value identifies another recipient, is inconsistent with the resolved requester metadata or transaction context, or is absent where required by the applicable profile.

NOTE: The aud value identifies the intended recipient of the OpenID4VP request object. It does not identify the signer, the signing credential, the CSC credentialID, or the QTSP.

- Expiry validity
- Credential types and disclosure constraints
- Request integrity
- transaction_data that shall decode to a valid qesApprovalRequest
- signatureQualifier recognition and support

Unsigned or invalid requests shall be rejected.

The selected SD-JWT VC shall support the org.cloudsignatureconsortium.dm.1.qesApproval claim, otherwise the EUDIW shall abort.

If any documentInfos entry contains both a href and a checksum, the EUDIW shall verify document integrity; if verification fails, the EUDIW shall abort.

The EUDIW shall display:

- Verifier identity
- Requested credential types
- Requested attributes or claims
- Any selective disclosure details
- A clear indication that the transaction authorizes a qualified electronic signature or seal to be created by the QTSP's Remote Signing Service Provider.
- The identity of the QTSP operating the Remote Signing Server that will perform the signature creation.
- The trust framework identified by signatureQualifier.
- The label of each document in documentInfos, or a clear indication that no label is provided.
- The number of signatures authorized, if numSignatures is present in the qesApprovalRequest.
- Whether document integrity has been automatically verified.
- A preview or rendering of the to-be-signed document(s) where technically feasible.

EUDIW holder shall explicitly consent.

DRAFT

B.8.2 VP Token / DC API response

Upon consent, the EUDIW shall compute the qesApproval value as defined in CSC data model bindings [3] Section 7.2.1.2:

- Hash input: the base64url-encoded UTF-8 qesApprovalRequest as it appears in the Authorization Request.
- Hash algorithm: as identified by hashAlgorithmOID in the qesApprovalRequest.
- Representation: a String containing the base64-encoded digest.

The EUDIW returns a verifiable presentation containing the selected approval credential with qesApproval set. For ISO mdoc this is a DeviceResponse; for SD-JWT VC it is the presentation with a Key Binding JWT that includes the claim org.cloudsignatureconsortium.dm.1.qesApproval carrying the qesApproval value.

Example (conceptual)

```
{
   "qscd_service_attestation": [
      "<base64url-encoded DeviceResponse or SD-JWT VC presentation with qesApproval>"]
}
```

The SCSP's AS validates the presentation like a normal Verifier and extracts qesApproval.

B.8.3 Using qesApproval at the AS/SCSP

After successful validation, the AS verifies:

- The digest in qesApproval equals hash (the exact qesApprovalRequest it sent) under hashAlgorithmOID.
- The bound credentialID/user identity in the approval credential matches the targeted remote QSCD credential and policy. Then the AS/SAM authorizes the signature (e.g. CSC API signatures/signHash or signatures/signDoc).

B.9 Security & privacy

- JWT/issuer: if the SCSP transports the approval request in a JWT, it shall be signed; the EUDIW shall validate signature and issuer authorization.
- Encodings: transaction_data outer is base64url encoded; integrity checksum and any binary items defined by CSC DM remain base64 (padded).
- AS/SAM: enforce sole control via SAD and trusted path; use qesApproval only within its lifetime/purpose.

B.10 Conformance checklist (SCSP)

1) Build OpenID4VP request with dcql_query asking for an approval credential (doctype/claims supporting qesApproval).
2) Include transaction_data = base64url(JSON) of qesApprovalRequest with signatureCreationApproval (labels, hashes, hashAlgorithmOID, counts).
3) Validate the returned presentation and extract qesApproval bound to the exact bytes of the request.
4) Authorize the QSCD signing step at the AS/SAM only if qesApproval and user/credential bindings match policy (EN 419 241-1 [6]).

B.11 Working sequence

1) SCSP AS → EUDIW: OpenID4VP Authorization Request with dcql_query (service attestation) + transaction_data (qes-approval).

DRAFT

2) EUDIW: Renders data, user approves; computes qesApproval; produces presentation.

3) EUDIW → SCSP AS: VP/DeviceResponse with qesApproval.

4) AS/SAM: Validates presentation; verifies qesApproval; calls QSCD (e.g. CSC API signatures/signHash and/or signatures/signDoc) to create the QES.

DRAFT

Annex C (normative): Specific recommendations for qualified signatures and/or seals creation

C.1 Introduction

This annex provides formal recommendations for implementing Qualified Electronic Signature (QES) creation workflows, while ensuring a clear separation between the authentication/authorization layer and the signature creation API layer.

C.2 Separation of Responsibilities

Authentication and authorization should be performed outside the SCS using high-assurance mechanisms (e.g. OAuth 2.0, OIDC, FAPI 2.0). The SCS is responsible exclusively for signature creation within the remote QSCD operated by the QTSP.

C.3 Authentication and Authorization Layer

The authentication/authorization layer responsible for user identity verification should implement OAuth 2.0/OIDC with PAR and PKCE, and private_key_jwt authentication mechanisms.

C.4 Signature Creation Service

The SCS that provides standardized operations for credential listing, credential information retrieval, credential authorization, and signature creation should validate authorization tokens and DTBS binding.

C.5 Strong Binding Between Consent and DTBS

Authorization tokens should embed specific DTBS hashes, credential identifiers and signature parameters to ensure non-repudiation and prevent replay attacks.

C.6 Use of FAPI 2.0 for EUDIW-Based Clients

When the EUDIW acts as the OAuth client, FAPI 2.0 mechanisms should be adopted: private_key_jwt with WUA keys, PAR for request integrity, and DPoP for sender-constrained access tokens.

C.7 SCS API Security Recommendations

All SCS API calls should be authenticated using OAuth access tokens, possibly following DPoP constraints, validating nonces, and enforcing one-time-use authorization objects.

DRAFT

Annex D (normative): Signing flow profile involving a generic signature initiation component

D.1 Overview

D.1.1 General

The present annex defines a generic signing flow profile in which a relying party (RP) initiates a signature request independently of the specific signature creation service or driving application used by the signer.

The profile enables interoperability between:

- relying parties requesting signatures,
- signature initiation components,
- driving applications (including EUDIW),
- signature creation services.

The signing flow is based on a two-object model:

- a signatureRequest, defining what, how and where to sign;
- a requestToSign, defining how the signer retrieves and executes the request.

D.1.2 Signing flows

A generic signing flow shall include the following steps:

1. The relying party creates a signatureRequest.
2. The relying party exposes the signatureRequest via a remote URI.
3. The relying party creates a requestToSign object referencing the signatureRequest.
4. The requestToSign is transferred to the signer using one of the supported mechanisms.
5. The signer uses a driving application to:
    a. retrieve the signatureRequest,
    b. select a signing service,
    c. authorize and execute the signature.
6. The signed result is returned to the responseURI defined in the signatureRequest.

D.2 Roles and components

The following roles are defined:

- Relying Party (RP): in the context of this annex, the entity that requests the signer to sign a document.
- Signer: natural or legal person providing signature authorization.
- Driving Application: application used by the signer to execute signatures.
    - Receives signature requests from the relying party.
    - Interacts with a signature creation application or signature creation application service component to sign signer's document.
    - Returns the signed document to the relying party and/or to the signer.
- Signature Creation Service (SCS): service performing signature creation.
- Signature Initiation Component (SIC): mechanism enabling transfer of signing requests.

The driving application may be:

- embedded in an EUDIW,
- a companion application,
- a browser-based component,

DRAFT

- a remote service.

D.3 Signature request model

D.3.1 Requesting signatures

To request a signature from a *signer*, a *relying party* prepares a requestToSign and sends it to the *signer*. Note that the *relying party* may be a natural person. The requestToSign is a JSON object with the following schema:

```
{
  "type": "object",
  "title": "requestToSign",
  "properties": {
    "version": {
      "type": "string",
      "const": "1.0"
    },
    "exp": {
      "type": "string",
      "format": "date-time"
    },
    "signatureRequestLocation": {
      "type": "object",
      "properties": {
        "uri": {
          "type": "string",
          "format": "uri"
        },
        "digest": {
          "$ref": "https://cloudsignatureconsortium.org/wp-content/uploads/2025/10/csc-dm.pdf#hash",
          "description": "Digest of the signatureRequest object."
        },
        "access": {
          "$ref": "https://cloudsignatureconsortium.org/wp-content/uploads/2025/10/csc-dm.pdf#accessControlMethod"
        }
      },
      "required": [ "uri", "digest" ]
    },
    "whereToSign": {
      "type": "array",
      "items": {
        "type": "string",
        "format": "uri"
      }
    }
  },
  "required": [ "version", "signatureRequestLocation" ]
}
```

The requestToSign is not confidential. If the signatureRequest referenced in the requestToSign contains confidential information, access to the signatureRequest should be restricted.

The *relying party* shall do the following:

1. Prepare a signatureRequest object.
    - The signatureRequest object shall be as defined in CSC data model, version 1.0.0.
    - The signatureRequest shall have a responseURI.
    - The signatureRequest may have a documentReference.
        - The access property of the documentReference may specify one of the access methods described in Restricting access clause below.
2. Prepare a requestToSign object according to the schema above.
    - The requestToSign shall contain a version property and the value shall be 1.0.
    - The requestToSign should contain an exp property.

DRAFT

        - The signatureRequest should not be available after the expiration indicated by the exp property.
    - The requestToSign shall contain a signatureRequestLocation property.
        - The signatureRequest from the preceding step shall be available at the uri property of the signatureRequestLocation.
        - The digest property of the signatureRequestLocation shall contain the digest of the signatureRequest object.
            - The signatureRequest shall be canonicalized according to RFC 8785 [39] prior to digest calculation.
            - References in the signatureRequest shall not be dereferenced prior to canonicalization.
            - The digest shall be calculated over the canonicalized signatureRequest.
        - Access to the signatureRequest shall be protected as indicated by the access property of the signatureRequestLocation.
    - The requestToSign may have one or more entries in the whereToSign array.
        - Each entry in whereToSign shall be a URI that accepts POST requests with a signatureRequest in the body.
            - The response to the request shall be either an appropriate error message or *signer's document* signed by the *signer* as specified in the signatureRequest.
        - Each entry in whereToSign shall use "https" URI scheme.
            - The HTTPS service shall present a valid QWAC.

3. Sign the requestToSign.
    - The requestToSign shall be encoded according to the JSON canonicalization scheme in RFC 8785.
    - The encoded requestToSign shall be signed by the relying party.
        - The signature should be done with the relying party access certificate.
    - The signature shall be a JWS compliant with RFC 7515 [8].
        - The JWS compact serialization rules shall apply.
        - The x5c JOSE header parameter shall be present.
            - The trust anchor shall not be included in the x5c header.
        - The alg JOSE header parameter shall identify an approved cryptographic suite listed in ETSI TS 119 312 [i.2].
            - The none algorithm shall not be used.
    - The typ JOSE header parameter SHALL be vnd.etsi.rts+jws.
4. Transfer the signed requestToSign to *signer's driving application* (see clause D.5).

Signer's driving application shall respond as follows:

- Validate the requestToSign:
    - *Signer's driving application* shall reject the requestToSign if the JWS cannot be validated.
        - Signer's driving application shall validate the certificate chain in the x5c JOSE header.
        - The trust anchor shall be validated against an appropriate root of trust.
    - *Signer's driving application* shall reject JWS with alg set to none.
    - *Signer's driving application* shall reject a requestToSign if the expiration time (exp property) has been exceeded.
- If whereToSign is present and contains at least one URI:
    - *Signer's driving application* should let the *signer* choose one of the URIs.
    - *Signer's driving application* may let the *signer* choose to sign using another local or remote signature creation application.
        - The signature creation application shall provide a signing policy that supports the signatureQualifier specified in the signatureRequest.
- If the signer chooses a remote signature creation application (i.e. one of the services in whereToSign or another remote service), the following shall apply:
    - Signer's driving application may POST the signatureRequest to the URI.
    - Signer's driving application may POST the signatureRequestLocation to the URI.
    - When signatureRequest is retrieved, it's digest shall be validated against signatureRequestLocation.digest.
- The signature creation application shall submit the signed document to the responseURI of the signatureRequest with a POST request.

NOTE: The root of trust could be the EU member states trust lists.

DRAFT

D.4 Restricting access

D.4.1 General

Access to the signatureRequest and/or signer's document may require authorization from the signer. In addition to the values specified in the CSC data model, the access property may use any of the following values:

- https://uri.etsi.org/19432/2026/access/oauth2 (TODO: Are URIs conformant and should they be registered?)
- https://uri.etsi.org/19432/2026/access/pid

NOTE: If access to signatureRequest is restricted, then access to signer's document can be protected by making it available at a URI that is unpredictable.

D.4.2 OAuth2 access control

When the access property is https://uri.etsi.org/19432/2026/access/oauth2 the following shall apply:

- The accessControlMethod shall contain the following additional parameters:
  - authzServer: URI identifying the OAuth2 authorization server.
  - scope: The OAuth2 scope that is required to access the signatureRequest.

NOTE: The format of the scope is out of scope for this specification. It is the responsibility of the relying party to ensure that the scope value is bound to a specific signatureRequest. The OAuth2 client can use the scope as an opaque value.

- The OAuth2 authorization server shall support:
  - Dynamic registration (RFC 7591).
  - Well-known URI supporting the oauth-authorization-server suffix (RFC 8414).
- The OAuth2 authorization server shall authenticate the signer by the EUDI wallet of the signer.
  - The OAuth2 authorization server may request PID presentation from the wallet to authenticate the signer.
  - The OAuth2 authorization server may request an EAA presentation from the wallet to authenticate the signer.
  - The OAuth2 authorization server should use OpenID4VP to request authentication and authorization from the EUDI wallet as described in annex B.
- The OAuth2 authorization server shall require PKCE as specified in RFC 7636.
- Access to signer's document shall require an access token authorized by the signer.

NOTE: It is the responsibility of the driving application and OAuth2 authorization server to agree on scopes or other mechanisms that allow the resource server to link signer's document with the intended signer.

D.4.3 PID Based Access Control

When the access property is https://uri.etsi.org/19432/2026/access/pid the following shall apply:

- The *relying party* shall possess one or more known attributes identifying the intended *signer*.
  - The attributes shall be sufficient to identify the intended *signer* with high confidence.
  - The attributes should be sufficient to uniquely identify the *signer*.
- Access to the protected resource shall require presentation of a PID belonging to the intended *signer*.
- The *relying party* shall validate that the presented PID contains the known attributes.
- Access to the protected resource shall require successful validation of the PID presentation.
- The *relying party* may request PID presentation with OpenID4VP.
- The *relying party* may request PID presentation with ISO/IEC 18013-5.
- The *relying party* may request PID by another protocol accepted by the European Commission for use in EUDI wallets.

D.5 Transport of the requestToSign

DRAFT

D.5.1 Engagement Protocols

D.5.1.1 General

There are a number of ways that a requestToSign can be transferred to signer's driving application.

In some situations, a signing flow is triggerred when the signer interacts with the relying party outside signer's driving application/EUDI wallet. This annex covers interactions where the signer:

- opens a link on the relying party web page with a browser on a device with signer's driving application.
- scans a QR code presented by the relying party with a device with signer's driving application.
- scans an NFC tag presented by the relying party with a device with signer's driving application.

An engagement protocol should cause the device with signer's driving application to open a transferMechanisms object with signer's driving application. This specification specifies the following engagement protocols:

- Deep Link
- URI with custom URI scheme
- QR Code
- NFC tag

Other specifications may define additional engagement protocols.

D.5.1.2 Transfer Mechanisms Data Structure

In an engagement protocol the relying party proposes one or more possible transfer mechanisms to signer's driving application. Signer's driving application selects one of the transfer mechanisms and retrieves the requestToSign as described in the transferMechanisms data structure.

The possible transfer mechanisms are described in a JSON object with the following schema:

```
{
  "name": "transferMechanisms",
  "type": "object",
  "properties": {
    "ble": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Human readable name of the relying party BLE service."
        },
        "sid": {
          "type": "string",
          "format": "uuid",
          "description": "UUID of the service."
        },
        "did": {
          "type": "string",
          "format": "uuid",
          "description": "Data characteristic UUID."
        }
      },
      "required": [ "sid", "did" ]
    },
    "wifi": {
      "type": "object",
      "properties": {
        "bands": {
          "type": "array",
```

DRAFT

```
        "items": {
          "enum": [ "2.4GHz", "3.65GHz", "4.9-5.0GHz", "5GHz", "5.9GHz", "6GHz" ]
        }
      },
      "chn": {
        "type": "integer"
      },
      "cls": {
        "type": "integer"
      },
      "host": {
        "type": "string",
        "description": "IP address of the relying party.",
        "anyOf": [
          {
            "format": "ipv4"
          },
          {
            "format": "ipv6"
          }
        ]
      }
    },
    "required": [ "bands", "chn", "cls", "host" ]
  },
  "https": {
    "type": "object",
    "properties": {
      "uri": {
        "type": "string",
        "format": "uri"
      }
    },
    "required": [ "uri" ]
  }
 }
}
```

Example:

```
{
 "ble": {
   "name": "Relying Party Signing Request",
   "sid": "08032841-2b51-4686-156d-c2e09814c2a7",
   "did": "724b3ab8-0b55-b041-65d6-d72f896e0fca"
 },
 "wifi": {
   "bands": [ "5GHz" ],
   "chn": 36,
   "cls": 114,
   "host": "fe80::1234:5678:9abc:def0"
 },
 "https": {
   "uri": "https://example.com/v1/rts/b5a2d858-88c4-934a-0cc4-bef54f0b5ad5"
 }
}
```

Other specifications may extend the schema with additional transfer mechanisms.

D.5.1.3 Deep Link

When engaging with a deep link, the relying party shows a URL to the signer:

DRAFT

- The base URL shall be provided by the wallet solution.

NOTE: This specification does not address how the relying party can identify the base URL. EUDI wallets could present an EAA with the base URL or a common URL can be used by all EUDI wallets.

- The URL shall have as first query parameter the transferMechanisms.
  - The transferMechanisms object shall be CBOR encoded in accordance with RFC 8949 [40].
  - The query parameter shall be base64 encoded according to RFC 4648 [41], section 5.
- The deep link may be accessed with a browser on a device with *signer's driving application*.
  - When the URL is accessed with a browser on a device with *signer's driving application*, the browser should open *signer's driving application* with the CBOR and base64 encoded transferMechanisms object.

NOTE: This method allows signer's driving application to use Android App Links and iOS Universal Links. Alternatively, the base URL can be a landing page which opens signer's driving application. The details for using these technologies are not in scope of this specification.

NOTE: Opening the deep link with a browser only works when signer's driving application is on the same device.

D.5.1.4 URI

The *relying party* presents a URI to the *signer*:

- The URI shall have URI scheme sign.
- The URI shall have as path element the transferMechanisms.
  - The transferMechanisms object shall be CBOR encoded in accordance with RFC 8949.
  - The path element shall be base64 encoded according to RFC 4648, section 5.
- The URI may be accessed directly with *signer's driving application*.
- The URI may be accessed with a browser on a device with *signer's driving application*.
  - When the URI is accessed with a browser on a device with *signer's driving application*, the browser should open *signer's driving application* with the CBOR and base64 encoded transferMechanisms object.

D.5.1.5 QR Code

When engaging with QR code, the *relying party* prepares a QR code as follows:

- The QR code shall conform with ISO/IEC 18004.
- The QR code may encode one of:
  - A URL according to clause Deep Link.
  - A URI according to clause URI.
  - Data specified in another specification.
- The QR code may be scanned directly with *signer's driving application*.
- The QR code may be scanned with a device with *signer's driving application*.
  - When the QR code is scanned with a device with *signer's driving application*, the device should open *signer's driving application* with the CBOR encoded transferMechanisms object.

NOTE: More than one signer's driving application could be installed on the same device. This specification does not enforce or suggest any particular approach to selecting signer's driving application. This is left to the device with signer's driving application.

NOTE: This method only works when *signer's driving application* is on a different device than the one showing the QR code.

D.5.1.6 NFC Tag

When engaging with NFC, the *relying party* presents an NFC tag to the *signer*:

DRAFT

- The NFC tag shall comply with NFC Forum, Type 4 Tag Specification, version 1.2 [42].
- The NFC tag shall be a Type 4 NDEF tag.
- The TNF value shall be Well Known (0x01).
- The type shall be URI record (0x55).
- The payload may be one of:
    - A URL according to clause D.5.1.3
    - A URI according to clause D.5.1.4
    - Data specified in another specification.
- The NFC tag may be scanned directly with *signer's driving application*.
- The NFC tag may be scanned with a device with *signer's driving application*.
    - When the NFC tag is scanned with a device with *signer's driving application*, the device should open *signer's driving application* with the CBOR encoded transferMechanisms object.

NOTE: This method only works when signer's driving application is in proximity of the relying party or a device controlled by the relying party.

D.5.2 Transfer mechanisms

D.5.2.1 Proximity Transfer Mechanisms

D.5.2.1.1 Introduction

These transfer mechanisms may only be initiated by the engagement protocols above.

D.5.2.1.2 NFC

The *relying party* may make the signed requestToSign available as an NFC tag. When the *relying party* makes the signed requestToSign available as an NFC tag, the following requirements shall apply:

- The NFC tag shall comply with NFC Forum, Type 4 Tag Specification, version 1.2.
- The NFC tag shall be a Type 4 NDEF tag.
- The TNF value shall be "Media Type" (0x02).
- The Payload Type field shall be application/vnd.etsi.rts+jws.
- The payload shall be the signed requestToSign.
- The NFC tag may be scanned directly with *signer's driving application*.
- The NFC tag may be scanned with a device with *signer's driving application*.
    - When the NFC tag is scanned with a device with *signer's driving application*, the device should open *signer's driving application* with the signed reqeustToSign object.
- After successful transfer of the signed requestToSign, both parties shall terminate the NFC session.

NOTE: NFC provides proximity but does not provide authentication of the relying party. Authenticity and integrity the requestToSign are provided by the JWS signature.

DRAFT

D.5.2.1.3 BLE

The *relying party* may make the signed requestToSign available over Bluetooth. When the *relying party* makes the signed requestToSign available over Bluetooth, the following requirements shall apply:

- The *relying party* shall make the signed requestToSign available over a Bluetooth Low Energy (BLE) connection.
- The *relying party* and *signer's driving application* shall establish a BLE connection in accordance with the Bluetooth Core Specification.
    - The *relying party* shall serve as the GATT Server.
    - The *relying party* shall advertise the BLE service parameters in transferMechanisms.
    - *Signer's driving application* shall serve as the GATT Client.
- The *relying party* shall expose the signed requestToSign through a GATT characteristic identified in transferMechanisms.
    - The signed requestToSign shall be exposed as a byte stream.
    - The first four octets shall contain the byte-length of the signed requestToSign encoded as an unsigned 32-bit integer in network byte order.
    - If the signed requestToSign exceeds the size of a single GATT characteristic value, the *relying party* shall expose the byte stream through multiple characteristic values.
    - *Signer's driving application* shall retrieve the signed requestToSign by reading the corresponding GATT characteristic.
    - *Signer's driving application* shall concatenate the characteristic values in transmission order and shall continue reading until the declared byte-length has been received.
    - The requirements in BLE shall apply to the transfer of the signed requestToSign.
- After successful transfer of the signed requestToSign, both parties shall terminate the BLE connection.

D.5.2.1.2 Wi-Fi Aware

The *relying party* may make the signed requestToSign available over a Wi-Fi Aware NAN Data Path. When the *relying party* makes the signed requestToSign available over a Wi-Fi Aware NAN Data Path, the following requirements shall apply:

- *Signer's driving application* and the *relying party* shall establish a NAN Data Path in accordance with the Wi-Fi Alliance Neighbour Awareness Networking Specification.
    - The *relying party* shall serve as Service Publisher.
    - The *relying party* shall advertise the Wi-Fi parameters in transferMechanisms.
    - The Wi-Fi Aware NAN Data Path may be established without a PMK.
    - *Signer's driving application* shall serve as Service Subscriber.
- The *relying party* shall expose the signed requestToSign as an HTTPS resource at path / at the IP address specified in the host property of transferMechanisms. (Consider direct TCP transfer)
- The requirements in HTTP shall apply to the transfer of the signed requestToSign with URL /.

NOTE: Since the server is likely referenced by IP address, the HTTP connection is not expected to use TLS. Integrity and authenticity is protected by the signature on the requestToSign.

DRAFT

- After successful transfer of the signed requestToSign, the *relying party* and *signer's driving application* shall release the NAN Data Path.

D.5.2.2 Remote Transfer Mechanisms

D.5.2.2.1 HTTP

The relying party may make the signed requestToSign available at an HTTP URL. When the relying party makes the signed requestToSign available at an HTTP URL, the following requirements shall apply:

- The URL shall identify an HTTP endpoint.
    - The server shall support HTTP/1.1 as defined in RFC 9112.
    - The server may support HTTP/2 or HTTP/3.
    - The server shall respond to HTTP GET requests.
- The connection should use TLS 1.3.
- When the request is successful the signed requestToSign shall be returned.
    - The response code shall be 200.
    - The content type shall be application/vnd.etsi.rts+jws.
- The 401 HTTP response code may be returned. If so:
    - The WWW-Authenticate: Bearer header shall be present as described in RFC 6750 [19].
    - Discovery of the authorization server may be supported in accordance with section 5 of RFC 9728 [32]. The resource_metadata shall be the URI of the authorization server well-known endpoint as specified in section 3 of RFC 9728.
- Signer's driving application shall treat any status code different from 200 or 401 as a failure.
- After successful transfer of the signed requestToSign, The relying party and signer's driving application shall terminate the HTTPS session.

D.5.2.2.2 Messaging

The *relying party* may send the signed requestToSign to *signer*. When the *relying party* sends the signed requestToSign to *signer*, the following requirements shall apply:

- The file shall be sent with mime-type application/vnd.etsi.rts+jws. (should a mime type be registered?)
- The *relying party* shall send the file to the *signer*.
- *Signer's driving application* shall support that the *signer* opens the file with *signer's driving application*.

NOTE: The file could be sent via web interface, email, or electronic registered delivery service

D.6 Security considerations

D.6.1 Integrity

- The signatureRequest shall be integrity protected.
- The requestToSign does not require integrity protection.

DRAFT

D.6.2 Authentication

- Signing services shall be authenticated (e.g. QWAC validation).
- Authorization shall be enforced before signature creation.

D.6.3 Confidentiality

- The signatureRequest need not be confidential.
- Document access shall be protected when required.

D.7 Conformance requirements

An implementation conforms to this annex if it:

- supports the signatureRequest and requestToSign structures;
- implements at least one transfer mechanism;
- supports HTTPS-based signing service invocation;
- enforces authentication and authorization;
- correctly delivers signed results to the responseURI.

DRAFT

Annex E (informative): Change history

Date	Version	Information about changes
September 2026	1.4.1	Publication as ETSI EN 319 432

DRAFT

History

Version	Date	Status
V1.3.2	June 2026	1st draft
V1.3.3	July 2026	2nd draft

DRAFT