MEF

Working Draft

MEF W128.1 Draft 0.2

LSO API Security Profile

March 2023

This draft represents MEF work in progress and is subject to change.

Disclaimer

Table of Contents

List of Figures

List of Tables

1 List of Contributing Members

The following members of the MEF participated in the development of this document and have requested to be included in this list.

- To be filled out before Letter Ballot

2 Abstract

This document defines the security profile, security approaches and security architecture for LSO API security using OAuth2 and OIDC within either a centralized or federated identity provider framework.

The intended audience of this document is senior IT security professionals, in particular identity and security architects and compliance specialists implementing LSO APIs. This document is not a general reference on API security, but an LSO API-specific standard.

The document first defines the LSO API security architecture and conformance requirements to that architecture. The standard then defines the following security components:

- JWT Best Practices for LSO API Security
- JWKS Endpoints for cryptographic signatures and their verifications
- Structure and conformance requirements for JWSs and JWEs
- LSO API Payload Authenticity

3 Terminology and Abbreviations

This section defines the terms used in this document. In many cases, the normative definitions to terms are found in other documents. In these cases, the third column is used to provide the reference that is controlling, in other MEF or external documents.

Term	Definition	Reference
Account Information Service Providers	Account Information Service Providers are authorized entities to retrieve account data provided by service providers.	Open Banking [21]
AISP	Account Information Service Provider	Open Banking [21]
API	Application Program Interface	MEF 55.1 [19]
Application Program Interface	A software intermediary that allows two applications to talk to each other.	MEF 55.1 [19]
Buyer	Buyer may be a customer, or a Service Provider who is buying from a Partner	MEF 55.1 [19]
DID	Decentralized Identifier	W3C DID [27]
Decentralized Identifier	A globally unique persistent identifier that does not require a centralized registration authority and is often generated and/or registered cryptographically	W3C DID [27]
FAPI	Financial-grade API	OpenID FAPI [26]
Financial-grade API	An industry-led specification of JSON data schemas, security, and privacy protocols to support use cases for commercial and investment banking accounts as well as insurance and credit card accounts.	OpenID FAPI [26]
Intent_id	A special claim defined by Open Banking for OIDC Connect Core	OpenID Connect Core [23]
JavaScript Object Notation	A lightweight data-interchange format.	ECMA JSON [2]
JOSE	JSON Object Signing and Encryption	IANA JOSE [4]
JSON	JavaScript Object Notation	ECMA JSON [2]
JSON Web Encryption	Encrypted content represented using JSON-based data structures.	IETF RFC 7516 [11]
JSON Web Key Set	A set of keys containing the public keys used to verify any JSON Web Token (JWT) issued by the authorization server and signed using an approved signing algorithm such as the recommended RS256 (RSA signature with sha-256 hashing).	IETF RFC 7517 [12]
JSON Web Signature	Represents content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures.	IETF RFC 7515 [9]
JSON Web Token	An open, industry standard method for representing claims securely between two parties.	IETF RFC 7519 [14]
JWE	JSON Web Encryption	IETF RFC 7516 [11]
JWS	JSON Web Signature	IETF RFC 7515 [9]
JWT	JSON Web Token	IETF RFC 7519 [14]
LSO	Lifecycle Service Orchestration	MEF 55.1 [19]

Term	Definition	Reference
OAuth2	OAuth 2.0 focuses on client developer simplicity while providing specific authorization flows for web applications. The OAuth2.0 Framework is defined in RFC 6749	IETF RFC 6749 [7]
OIDC	OpenID Connect	OpenID Connect Core [23]
OpenID Connect	A simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.	OpenID Connect Core [23]
Relying Party	An OAuth 2.0 Client application that requires user authentication and claims from an OpenID Connect Provider.	OpenID Connect Core [23]
Representational State Transfer	An architectural style for distributed hypermedia systems	Fielding 2000 [3]
REST	Representational State Transfer	Fielding 2000 [3]
RP	Relying Party	OpenID Connect Core [23]
Software Statement Assertion	A JSON Web Token (JWT) containing client metadata about an instance of client software. This is used for OpenID Dynamic Client Registration.	IETF 7591 [15]
Security Domain	A domain that implements a security policy and is administered by a single authority.	CNSSI 4009 [1]
Seller	Seller may be a Service Provider or a Partner who is providing service to a Buyer	MEF 55.1 [19]
SSA	Software Statement Assertion	IETF 7591 [15]
Third Party Provider	Account Information Service Providers	Open Banking [21]
TPP	Third Party Provider	Open Banking [21]
Trust Domain	Security Domain	This document
VC	Verifiable Credential	W3C VCDM [27]
Verifiable Credential	A tamper-evident credential that has authorship that can be cryptographically verified.	W3C VCDM [27]

Table 1 – Terminology and Abbreviations

4 Compliance Levels

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 (IETF, 2017) when, and only when, they appear in all capitals, as shown here. All key words must be in bold text.

Items that are **REQUIRED** (contain the words **MUST** or **MUST NOT**) are labeled as **[Rx]** for required. Items that are **RECOMMENDED** (contain the words **SHOULD** or **SHOULD NOT**) are labeled as **[Dx]** for desirable. Items that are **OPTIONAL** (contain the words **MAY** or **OPTIONAL**) are labeled as **[Ox]** for optional**.**

A paragraph preceded by **[CRa]<** specifies a conditional mandatory requirement that **MUST** be followed if the condition(s) following the “<” have been met. For example, “**[CR1]<**[D38]” indicates that Conditional Mandatory Requirement 1 must be followed if Desirable Requirement 38 has been met. A paragraph preceded by **[CDb]<** specifies a Conditional Desirable Requirement that **SHOULD** be followed if the condition(s) following the “<” have been met. A paragraph preceded by **[COc]<** specifies a Conditional Optional Requirement that **MAY** be followed if the condition(s) following the “<” have been met.

5 Introduction

The current B2B business automation standards as expressed through the LSO APIs are lacking basic cybersecurity standards – cybersecurity “blocking and tackling” – and advanced threat protection.

One key prerequisite for a Zero Trust Framework is the implementation of cybersecurity “blocking and tackling” standards such as authentication and authorization as foundational building blocks to provide security and assurance across enterprise trust boundaries.

This standard sets out to provide such context-specific cybersecurity “blocking and tackling” by providing specific cybersecurity functional requirements and mechanisms that help to produce consistently secure LSO API-based communications between entities across Trust Domains. This standard’s aim is to gain alignment on the detailed LSO API security mechanisms for interface reference points including Sonata, Interlude, Cantata and Allegro.

For simplicity, this document will use the term entity as a stand-in for Buyer, Seller, enterprise customer, and Third-Party Provider (TPP). Where required, for disambiguation the document will use the terms Buyer, Seller, enterprise customer and TPP.

This document provides a baseline for authentication (verifying the identity of a service requester) and authorization (verifying the allowed scope of access to Buyer/Seller resources of a service requester) across Trust Domains and a list of supported Identity frameworks that will integrate with access policies.

The scope of this document is to address the following security areas for LSO APIs:

- Authentication Frameworks
- Identity Authentication
- Access Claims Requirements
- Authorization Framework
- Access Claims Processing

This standard covers OpenAPI/REST APIs. RestConf and NetConf APIs are out of scope.

Furthermore, this standard will not address the lifecycle (provisioning/removal/updates) of identities and claims (access control policies).

First, and by way to set context, accessing, requesting, and delivering a service and its notifications between entities via LSO APIs always follows the request-response pattern. This document assumes that entities are in different Trust Domains and, therefore, must apply the LSO API Security Framework to all services crossing Trust Domains. A Trust Domain in the context of this document is equivalent to a Security Domain as defined in CNSSI 4009 [1].

A Trust Domain is a security domain that implements a security Policy and is administered by a single authority. An example of a Trust Domain is a LSO API endpoint host.

Second, there are three levels of LSO API security across Trust Domains :

1. Transport layer security through HTTPS as described in OAuth2 using OAuth2's Open-API definitions – establishes a secure communication channel between entities.
2. LSO API access security through the endpoint providing LSO API authentication and authorization – answering the question: Am I allowed to access a specific environment?
3. Entity LSO API security through function-specific scopes and associated authentication and authorization policies – Answering the question: Am I allowed to access specific functions/resources in a specific environment and do specific things with that function/resource?

Transport security is considered the 1st level of security and is aligned with the minimum requirements of the standards referenced in this document – OAuth2, OpenID Connect (OIDC), UK Open Banking and W3C Verifiable Credentials – and not further discussed in this document.

This document will provide MEF-specific standards for the 2nd and 3rd level of security.

To provide further context for the subsequent discussions, the document provides concrete examples of what is meant by the 2nd and 3rd level of security as defined in this section in Figures 1 and 2. Since the 1st level is out of scope for this document, this document does not provide an example.

Figure 1 outlines an example of LSO API Authentication, the 2nd level of security.

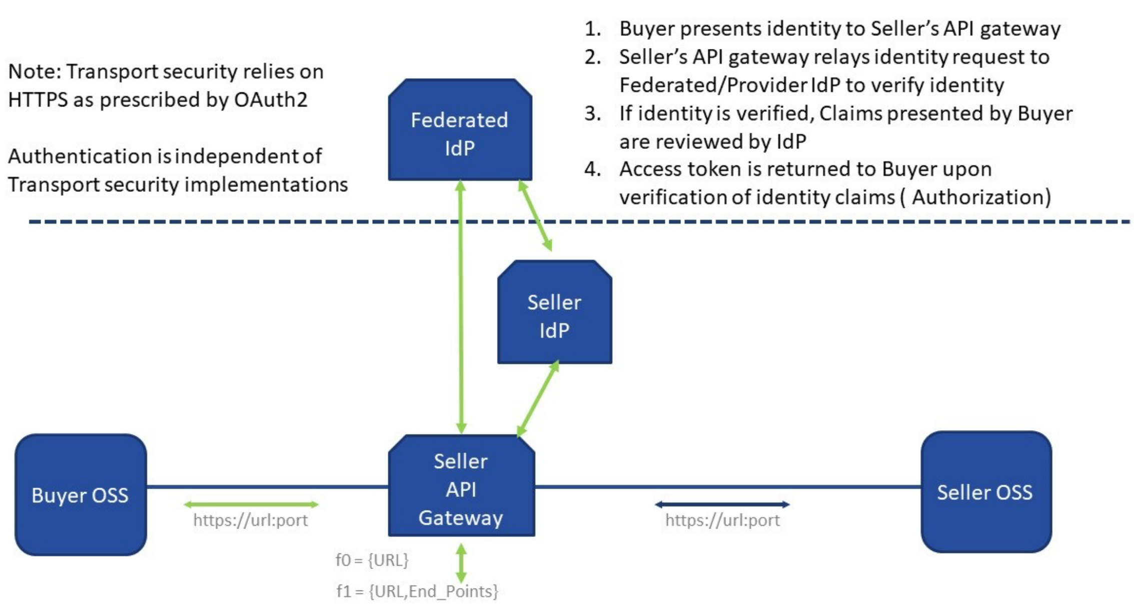

Figure 1 – Example Authentication Flow

The dataflow in Figure 1 is composed of the following steps:

- Buyer's client application presents its identity to the Seller API Gateway
- The Seller's API gateway consults with its internal or federated Identity providers to verify the identity and claims presented by the Buyers' application
- Upon verification of claims and identity, a token is provided to the Buyer's application.

Figure 2 outlines an example of Buyer–Seller LSO API security through function-specific scopes and associated authentication and authorization policies, 3rd level of security.

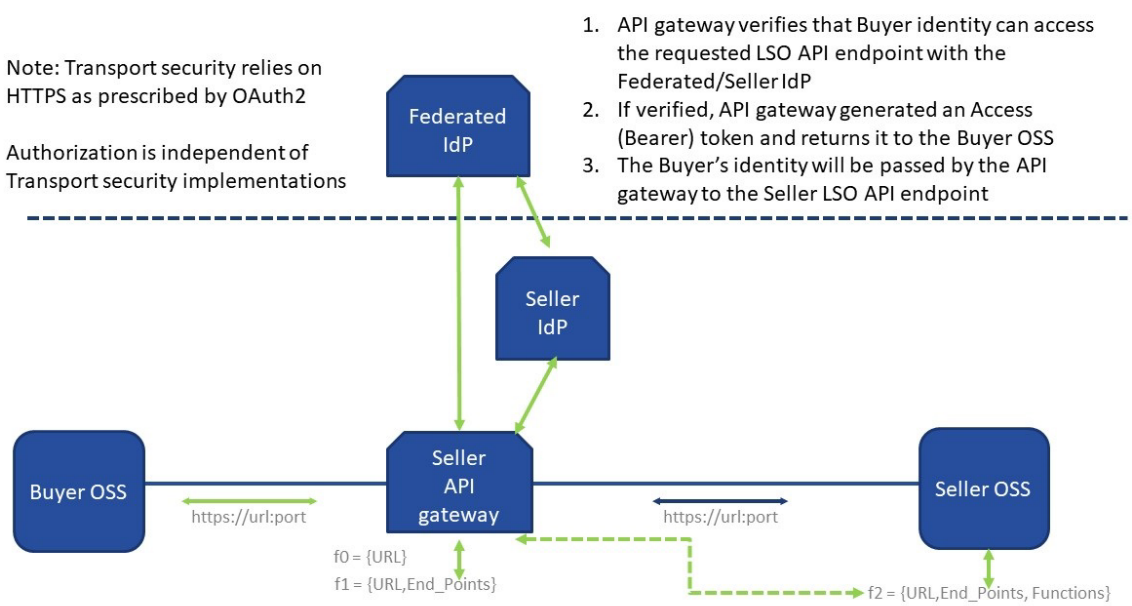

Figure 2 – Example Authorization Framework with Federation

The dataflow in Figure 2 is composed of the following steps:

- Seller's API Gateway verifies whether the endpoint access request is permitted for the Buyer's identity presented in the request
- If the request is allowed, the API gateway generates a bearer token and provides it to the Buyer's application
- The Buyer's identity is passed through the API gateway to the Seller's LSO API endpoint

The document's scope is limited to the definition of the schema of the JSON Web Token (JWT) used to perform authentication of a Buyer and the authorization that said Buyer has to the LSO API endpoint the Buyer is interacting with.

Payload security is out of scope. It should be implemented to ensure both parties use verifiable means to protect the integrity of data being exchanged.

Figure 2 depicts the data flows between Buyer and Seller to obtain an Access (Bearer) token, and how the Bearer token is used to access protected resources.

The document is structured in the following way:

1. MEF LSO Security Architecture in Section 6 with
    a. A discussion on MEF LSO API Security Architecture Prerequisites
    b. The delineation of Supported Authentication Frameworks and their threat models
    c. An outline of how to consume Service Provider (SP) owned Resources from another Service Provider (SP)
    d. A detailed discussion of the Hybrid Grant Flow Request with Intent Id
    e. A discussion of the Hybrid Grant Flow Parameters
2. JWT Security Suite Information v1.0 in section 7 with
    a. General Guidance for JWT Best Practice
    b. A brief discussion of JSON Web Key Sets (JWKS) Endpoints.
    c. General outline for creating a JSON Web Signature Token (JWS) to be used in LSO API Security Architecture.
    d. General Outline for creating a JSON Web Encryption Token (JWE), as an alternative to a JWS, to be used in LSO API Security Architecture.
3. An informative Implementation Guide in Appendix Bwith
    a. Specified and Non-specified Authentication and Authorization behavior
    b. Detailed Success Flows and examples for LSO API Authentication and Authorization
    c. Common Implementation Edge Cases

6 MEF LSO Security Architecture

This section details the MEF LSO Security Architecture. This document discusses the following aspects in sequence:

1. Prerequisites for utilizing the MEF LSO security
2. Supported authentication frameworks
3. MEF LSO API security architecture workflows, data models and JSON security information
4. MEF LSO API security model examples & exceptions

6.1 MEF LSO API Security Architecture Prerequisites

Uniqueness and security of identifiers utilized in LSO APIs is particularly important to unambiguously identify Service Providers (SPs) and TPPs as their delegates interacting with and through LSO APIs and to keep those interactions secure. Furthermore, and to facilitate automation and real time interactions within and through LSO APIs, discovery of identifiers and an ability to resolve them to the underlying public keys that secure them without having to rely on a trusted 3rd party is also critical.

This document assumes several capabilities must be in place before the MEF LSO API endpoint can be fully operational. We express them in this minimal set of prerequisites.

[R1] To open an API workflow, the Enterprise, SP or TPP **MUST** at the very least have an agreed mechanism to onboard and validate the trustworthiness of new IdPs from which they are willing to accept an identifier. This mechanism could be procedural but could also include additional technical controls. The exact implementation is left to the implementer.

[R2] Any Enterprise, SPs or TTPs wishing to enable OpenAPI access using the MEF LSO API security endpoints **MUST** also have the means to validate a requesting's identity at the time of the request and to ensure that the requesting entity is been properly granted access to the requested resource.

Conversely:

[R3] The entity requesting access to an LSO API **MUST** have a unique identifier.

[R4] Any unique identifier **MUST** be associated with a set of public keys.

[D1] Any unique identifier **SHOULD** follow the W3C DID Core specification.

These requirements allow an entity to prove that it controls, and can, thus, authenticate the unique identifier utilized in the LSO API Security context of this document without a verifying third party.

[R5] Any unique identifier **MUST** be resolvable to its associated public keys used for cryptographic authentication of the unique identifier.

This requirement ([R5]) allows an entity to access the public keys used in the unique identifier authentication independently of the entity requesting access or any other third party.

This requirement ([R5]) supports the self-issuance of unique identifiers that allow for cryptographically verifiable non-repudiation. Note that the usage of commonly used public key infrastructure (PKI) based on X.509 digital certificates is permissible. Threat models to traditional PKI are outlined in Appendix A.

After having discussed the minimal set of requirements on identifiers utilized in LSO APIs, it is important to discuss how these relate to identity and claims about facts relevant to entities, also called credentials.

[R6] A unique identifier utilized with LSO APIs **MUST** be linked to a Legal Entity of the service-requesting entity or its TPP through a cryptographically signed, cryptographically verifiable, and cryptographically revocable credential based on the public keys associated with the unique identifier of the credential issuer.

In the context of this document, a Legal Entity is an individual, organization or company that has legal rights and obligations.

This document makes no assumptions as to how a legal identity establishing credential is created, which identity credential issuers are mutually acceptable between Buyer and Seller and how these identity credentials are exchanged to establish mutual trust across enterprise trust boundaries to perform authentication and authorization operations for LSO APIs between Buyer and Seller.

Note that credentials utilized with LSO APIs may be self-issued. The acceptance of self-issued credentials is up to the Buyer/Seller that need to rely on the claim(s) within a self-issued credential.

[R7] The unique identifier of the Legal Entity of the TPP/SP **MUST** be the subject of the credential.

[R8] The unique identifier of the issuer of the Legal Entity credential utilized in LSO APIs **MUST** have a credential linking the unique identifier of the issuer to an Entity accepted by the SPs.

[R9] A credential utilized with an LSO API **MUST** itself have a unique and resolvable identifier.

Note that the unique and resolvable identifier of a credential does not have to be associated with any cryptographic keys.

[R10] If present, the status of a credential utilized within an LSO API **MUST** be discoverable by a party verifying the credential, the credential verifier.

In the context of this document, a credential status signals if a credential has been revoked or not, and a credential verifier is defined per the W3C Verifiable Credential Standard [27].

[D2] A credential utilized with an LSO API **SHOULD** be discoverable by either SP.

[R11] The presentation of a credential utilized with a LSO API **MUST** be cryptographically signed by the presenter of the credential, also known as the credential holder.

See the W3C Verifiable Credential Standard for a definition of credential holder.

[R12] If a credential holder is a SP, the holder **MUST** have a unique identifier that has been established within the LSO API security context the holder operates in.

This document makes no assumptions about existing business relationships between SPs. It is in the purview of the relying party whether these prerequisites are sufficient or whether additional requirements need to be fulfilled. An (OIDC) Relying Party is an OAuth 2.0 Client application that requires user authentication and claims from an OpenID Connect Provider.

Appendix A includes details on the scope of the threat model associated with these requirements and additional good practice steps that may be undertaken by each party to address these.

6.2 Supported Authentication Frameworks

In this standard, OAuth 2.0 is the primary framework for API Security for MEF LSO APIs augmented by both centralized and federated Identity Provider frameworks utilizing JSON Web Tokens (JWTs) [14] for authentication and resource authorization claims following the OpenID Connect standard framework (OIDC) [23]. OAuth 2.0 itself is a framework which can be deployed in many ways. Therefore, and to securely use the OAuth 2.0 framework, a security profile must exist by which Service Providers (SPs) or their Third Party Service Providers (TPPs) are certified to have correctly configured their clients and servers. TPPs act as a SP authentication service provider when the SP has outsourced its authentication services to a vendor.

6.3 Consuming Service Provider (SP)-owned Resources from another SP

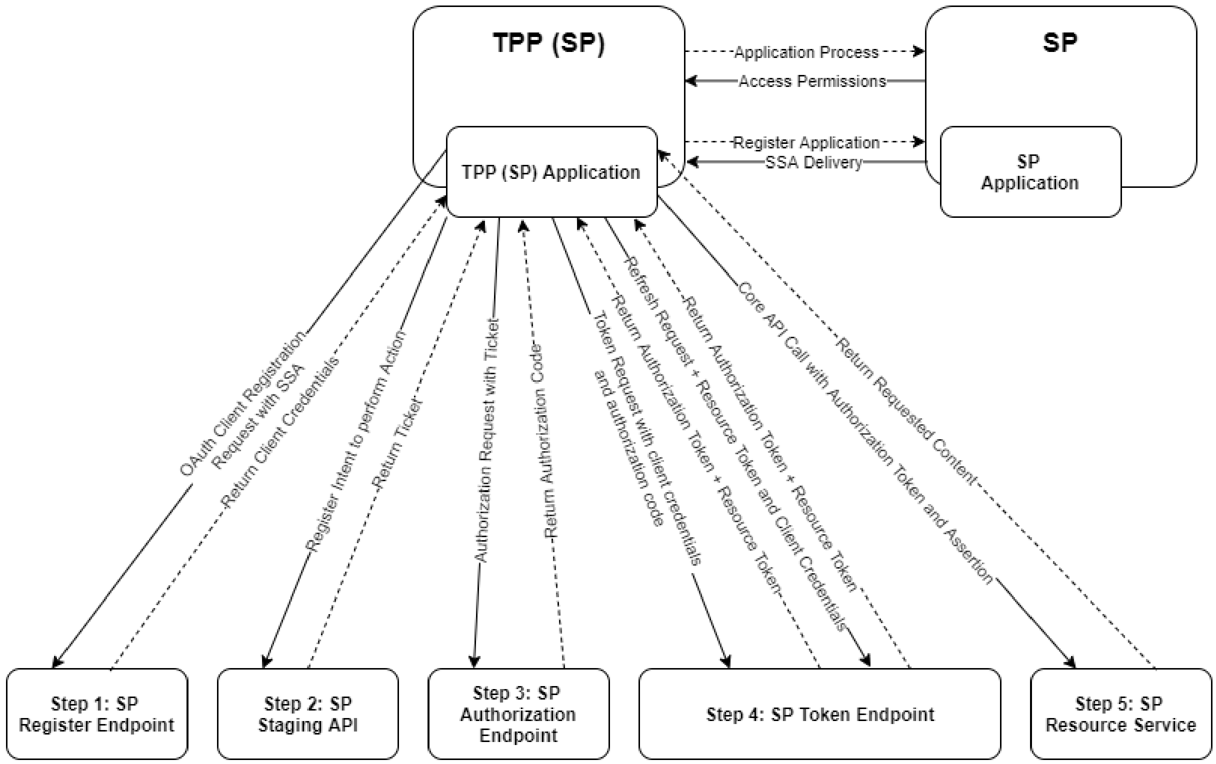

Figure 3 – MEF LSO Security Architecture

For context setting and completeness this document reiterates the typical OAuth2 authentication and authorization process for SP resources such as LSO APIs incorporating OpenID Connect Request Objects as JWTs containing relevant Identity Provider Information as depicted in 3.

Step 1: SP Register Endpoint

A TPP/SP submits a SSA through an OAuth2 client registration request to a known API endpoint of a SP that controls client registration for an LSO API as a resource to be accessed by the TPP/SP. A Software Statement Assertion (SSA) [15] is a JWT containing client metadata about an instance of TPP/SP client software. This is used for OpenID Connect Dynamic Client Registration. The SSA is used by an OAuth client to provide both informational and OAuth protocol-related assertions that aid OAuth infrastructure to both recognize client software, e.g., signed release hash and determine a client's expected requirements when accessing an OAuth-protected resource, e.g., required cryptographic algorithms to be used.

If the SSA meets the OAuth2 requirements of the target SP, either Buyer or Seller, the target SP issues client credentials.

Step 2: SP Staging API

When a TPP/SP wants to access an LSO API either once or repeatedly. the TPP/SP submits an intent to perform a specific LSO API action and why the client wants to perform such an action to a known API endpoint of a SP. If the request is authenticated, the client will receive a ticket back which is necessary to be presented in the next step. A ticket could for example be simply an Id such as an Intent Id. This step is recommended to provide very specific authorizations which might be required for regulatory reasons such as for payment. A ticket functions just like a queue number. Details of a ticket object and its definition are given in the Open Banking standard [22] and will not be repeated here.

Step 3: SP Authorization Endpoint

To receive an authorization token for the LSO API (not the specific function), the TPP/SP submits the ticket from step 2 in an authorization request to a known API endpoint of a SP. And if the TPP/SP is both authenticated and the ticket validated, the SP providing the LSO API will return an authorization code. This authorization code is used to obtain the fine-grained authorization to the desired function.

Step 4: SP Token Endpoint

Once an authorization code to access the domain of the LSO API has been obtained by the TPP/SP, the TPP submits a token request to a known API endpoint of a SP containing the client credential and the authorization token. If there is an existing authorization policy for the LSO API associated with the client credential at the token endpoint, an authorization token – that the TPP/SP can access a very specific LSO API functional endpoint and may or may not include specific fine-grained authorizations and cryptographic material – and a resource token – that the TPP/SP can access a specific resource, typically a specific server or specific serverless function and may or may not include specific resource metadata and cryptographic material – are issued to the TPP/SP. Note that if the original intent was to access the LSO API repeatedly the authorization and resource tokens are time bound and need to be refreshed. Otherwise, they are typically single use only.

Step 5: SP Resource Server

The TPP/SP can now finally access the detailed LSO API function on the resource server through a known API endpoint of a SP, by calling a single function LSO API endpoint on the resource server in a request containing the authorization and resource tokens and the LSO API endpoint payload. If the resource server validates the authorization token and the resource token, the LSO API request is executed, and the function specific response is generated and sent to the TPP/SP.

There are two possible operating models that this document needs to accommodate based on Figure 3:

- **Model 1:** An SP, as Buyer or Seller, is operating its own authentication and resource infrastructure. In this model the TPP is the SP.
- **Model 2:** An SP, as Buyer or Seller, outsourced/delegated either its authentication or resource infrastructure or both to a 3rd party, a TPP. In this model the TPP is different from the SP owning the resource.

Note that as a prerequisite to **Step 1: SP Register Endpoint**, the SP receiving the registration request needs to have a notion of the TPP/SP and its identity submitting the request.

Furthermore, since SPs TPP/SP client requirements are SP specific, these requirements are out of scope of this document as well. This means that for Step 1, this document simply refers to the OpenID Connect Dynamic Client Registration standard, and there in particular Section 3.1: Client Registration Request [23]. It is recommended that SPs follow the OpenID Connect Discovery standard [25] to publish their OAuth2 client requirements.

Model 2 is discussed because it is more general, and, where required, this document will highlight any adjustments to Model 2 to accommodate Model 1.

See the OpenID Connect Core standard, section 6 [23] for necessary OIDC flow details not discussed in this section.

The OpenID Connect Request object in Figure 3 uses the same claims' object for specifying claim names, priorities, and values. However, if the request object is used, the claims object becomes a member in an assertion that can be signed and encrypted, allowing the SP to authenticate the request directly (Model 1) or from its TPP (Model 2) and ensure it has not been tampered with. The OpenID Connect request object can either be passed as a query string parameter, a JWS or a JWE or can be referenced at a protected endpoint.

In addition to specifying a ticket, the TPP (SP) can optionally require a minimum strength of authentication context or request to know how long ago the requesting SP was authenticated. Multiple tickets could be passed, if necessary. Note, this feature is fully specified in the OpenID Connect standard, therefore, there is no need for any proprietary implementations.

Full accountability is available as required by all participants. Not only can the SP prove that they received the original request from the TPP (Model 2) or the other SP (Model 1), but the TPP (Model 2) or SP (Model 1) can prove that the access token that comes back was the token that was intended to be affiliated to this specific request.

6.4 Hybrid Flow Request with Intent Id

Within the OpenID Connect Framework there are three types of authentication flows:

1. Authentication Code Flow
2. Implicit Flow
3. Hybrid Flow

These flows are combined with OpenID Connect claims to integrate authorization within authentication flows.

The Hybrid Flow incorporating an Intent is the recommended approach because it not only addresses the attacks outlined in IETF RFC 6819 [8] but also Identity Provider Mix Up attacks. A so called 'cut and pasted code attack' where the attacker exchanges the 'code' in the authorization response with the victim's 'code' obtained by the attacker through another attack. The attacker uses the 'code' in a session to feed to the client to obtain an access token with the victim's privileges. Furthermore, registering an intent simplifies audit reporting when the API accesses sensitive data or triggers sensitive operations. This flow has also been adopted by the Open Banking consortium. Since authorization claims are included in the flow after authentication, it is called Hybrid Grant Flow.

This section describes parameters that should be used with a hybrid grant flow request such that an intent id can be passed from the TPP/SP to a SP.

Prior to this step:

- The TPP/SP (Buyer) would have been granted a credential by another SP (Seller)
- The Seller would have applied an authorization policy to the Buyer credential
- The TPP/SP would have registered a client application (Step 1 from section 6.3)
- The TPP/SP would have already registered an intent with a SP (Step 2 from section 6.3)
- The SP would have responded with an intent id (Step 2 from section 6.3).

6.5 Hybrid Grant Flow Parameters

This subsection covers the minimum requirements for the exchange of information in the hybrid grant flow and the issuance of an Id Token by the Seller to the Buyer.

Minimum Conformance Requirements

This section describes the minimal set of authorization request parameters that an SP must support. The **technical definitive reference** is specified in OpenID Connect Core Errata 1 Section 6.1 (Request Object) [23]. The requirements are listed in Table 2.

[R13] All standards and guidance **MUST** be followed as per the OpenID Connect (OIDC) specification.

[R14] A SP **MUST** support the issuance of OIDC ID Tokens as defined in the OIDC specification.

[O1] A TPP/SP **MAY** request that an ID token is issued.

Parameter	MEF LSO	Notes
response_type	Required	OAuth2 specification requires that this parameter is provided in an OAuth2 authentication workflow. The value is set to 'code id_token', 'code id_token token' or 'code'. **[R15]** TPPs/SPs **MUST** provide this parameter and set its value to one of three ('code id_token', 'code id_token token' or 'code') depending on what the SP supports as described in its well-known configuration endpoint. See definition of the well-known configuration endpoint in the OpenID Connect Discovery 1.0 specification [25]. **[R16]** The values for these configuration parameters **MUST** match those in the OIDC Request Object if present. Note: Risks have been identified with the "code" flow that can be mitigated with the hybrid flow. The MEF LSO API Profile allows SPs to indicate what grant types are supported using the standard well-known configuration endpoint. **[R17]** (OIDC) Relying Parties (RPs) **MUST** take care in validating that code swap attacks have not been attempted. An (OIDC) Relying Party is an OAuth 2.0 Client application that requires user authentication and claims from an OpenID Connect Provider.
client_id	Required	**[R18]** TPPs/SPs **MUST** provide this value and set it to the client id issued to them by the SP to which the authorization code grant request is being made. **[D3]**The client_id **SHOULD** be self-issued by the TPP as per the W3C DID standard, if it has been linked to either directly or indirectly through a verifiable credential as per the W3C Verifiable Credential standard
redirect_uri	Required	**[R19]** TPPs/SPs **MUST** provide the URI to which they want the resource owner's user agent to be redirected to after authorization. **[R20]** This URI **MUST** be a valid, absolute URL or resolvable URI that was registered during Client Registration with the SP **[R21]** In case the client_id is a DID, the URI **MUST** be a Service Endpoint in the DID document of the registering client_id
Scope	Required	**[R22]** TPPs/SPs **MUST** specify the scope that is being requested. **[R23]** At a minimum, the scope parameter **MUST** contain openid **[R24]** The scopes **MUST** be a sub-set of the scopes that were registered during client registration with the SP.

State	Recom-mended	**[O2]** TPPs/SPs **MAY** provide a state parameter. The state parameter may be of any format, and is opaque to the SP. **[CR1]<**[O1] If the state parameter is provided, the SP **MUST** play-back the value in the redirect to the TPP/SP. **[D4]**SPs **SHOULD** include the s_hash – the hash of the state as the state parameter.
Request	Required	**[R25]** The TPP **MUST** provide a value for this parameter. **[R26]** The parameter **MUST** contain a JWS or JWE that is signed by the TPP. **[R27]** The JWS/JWE payload **MUST** consist of a JSON object containing an OIDC request object as per OIDC Core specification 6.1. **[R28]** The OIDC request object **MUST** contain a claims section that includes an ID Token having as a minimum the following element: • **meflso_intent_id:** that identifies the intent id for which this authorization is requested **[R29]** The intent id **MUST** be the identifier for an intent returned by the SP to TPP that is initiating the authorization request. **[O3]** **acr_values:** TPPs **MAY** provide a space-separated string that specifies the acr values that the Authorization Server is being requested to use for processing this Authentication Request, with the values appearing in order of preference. **[R30]** The **acr_values MUST** be one of: • **urn:mef:lso:security:oidc:acr:sca:** To indicate that secure client authentication must be carried out • **urn:mef:lso:security:oidc:acr:ca:** To request that the client is authenticated without using a SCA, if permitted **[O4]** The OIDC request object **MAY** contain claims to be retrieved via the UserInfo endpoint only if the endpoint is made available and listed on the well-known configuration endpoint on the authorization server. **[O5]** The OIDC request object **MAY** contain additional claims to be requested should the SPs authorization server support them; these claims are listed on the OID well-known configuration endpoint.

Table 2 – Minimum Conformance

6.5.1 Example for minimum conformance hybrid grant flow profiles: HTTP Request and id_token returned

The HTTP request in Figure 4 depicts the fields and sample possible values defined in Table 2. The structure of id_token returned upon a successful request is shown in Figure 6. Figure 7 shows the structure of the id_token when the subject is a user.

6.5.1.1 HTTP Request JWS/JWE

```
GET /authorize?
response_type=code%20id_token
&client_id=s6BhdRkqt3
&state=af0ifjsldkj&
&scope=openid
&nonce=n-0S6_WzA2Mj
&redirect_uri=https://api.mytpp.com/cb
&request=CJleHAiOjE0OTUxOTk1ODd.....JjVqsDuushgpwp0E.5leGFtcGxlIiwianRpIjoiM....J
leHAiOjE0.olnx_YKAm2J1rbpOP8wGhi1BDNHJjVqsDuushgpwp0E
```

Figure 4 – HTTP Request – Hybrid Grant Flow

Note that the example shown in Figure 4 is without Base64 encoding. Also note that "essential" is an optional property. It indicates whether the Claim being requested is an Essential Claim. If the value is true, this indicates that the Claim is an Essential Claim. For instance, the Claim request:

```
"auth_time": {"essential": true}
```

can be used to specify that it is Essential to return an auth_time Claim Value. If the value is false, it indicates that it is a Voluntary Claim. The default is false.

By requesting Claims as Essential Claims, the RP indicates to the SP that releasing these Claims will ensure a smooth authorization for the specific task requested by a SP.

Note that even if the Claims are not available because the SP did not authorize their release or they are not present, the authorization server must not generate an error when Claims are not returned, whether they are Essential or Voluntary, unless otherwise specified in the description of the specific claim, see the OIDC Core Specification.

```
{
    "alg": "RS256",
    "kid": "GxlIiwianVqsDuushgjE0OTUxOTk"
}
.
{
    "aud": "https://api.acme.com",
    "iss": "s6BhdRkqt3",
    "response_type": "code id_token",
    "client_id": "s6BhdRkqt3",
    "redirect_uri": "https://api.mytpp.com/cb",

    "state": "af0ifjsldkj",
```

```
      "nonce": "n-0S6_WzA2Mj",
      "max_age": 86400,
      "claims":
      {
        "userinfo":
        {
         "meflso_intent_id": {"value": "urn:acme-intent-58923", "essential": true}
        },
        "id_token":
        {
         "meflso_intent_id": {"value": "urn-acme-intent-58923", "essential": true},
         "acr": {"essential": true,
                  "values": ["urn:mef:lso:security:oidc:acr:sca",
                       "urn:mef:lso:security:oidc:acr:ca"]}}
        }
      }
}
.
<<signature>>
```

Figure 5 – Request JWS/JWE

HTTP Request: id_token returned

Figure 5 shows the content of a JWS with the id_token that is being returned.

Note that Sub is being populated with an EphemeralId of the IntentId.

```
{
  "alg": "RS256",
  "kid": "12345",
  "typ": "JWT"
}
.
{
   "iss": "https://api.acme.com",
   "iat": 1234569795,
   "sub": "urn-acme-quote-58923",
   "acr": "urn:mef:lso:security:oidc:acr:ca",
   "meflso_intent_id": "urn-acme-quote-58923",
   "aud": "s6BhdRkqt3",
   "nonce": "n-0S6_WzA2Mj",
   "exp": 1311281970,
   "s_hash": "76sa5dd",
   "c_hash": "asd097d"
  }
.
{
<<Signature>>
}
```

Figure 6 – id_token Return

6.5.1.2 id_token returned

Figure 7 shows Identity Claims and IntentId with sub being populated with an UserIdentifier

```
{
  "alg": "RS256",
  "kid": "12345",
  "typ": "JWT"
}
.
{
   "iss": "https://api.acme.com",
   "iat": 1234569795,
   "sub": "ralph.bragg@raidiam.com",
   "acr": "urn:mef:lso:security:oidc:acr:sca",
   "address": "2 Thomas More Square",
   "phone": "+447890130559",
   "meflso_intent_id": "urn-acme-quote-58923",
   "aud": "s6BhdRkqt3",
   "nonce": "n-0S6_WzA2Mj",
   "exp": 1311281970,
   "s_hash": "76sa5dd",
   "c_hash": "asd097d"
  }
.
{
<<Signature>>
}
```

Figure 7 – id_token return with UserIdentifier

Implementers should note that ID Token Claims details should follow the JWT Best Current Practices [18] section 3.1.

The different token data properties are listed in the Table 3. The last column describes what the value of the field means.

Field	Definition	Notes	Value(s)
iss	Issuer of the token	Token issuer is specific to the business. **[R31]** The iss **MUST** be JSON string that represents the issuer identifier of the authorization server as defined in RFC 7519 [14]. When OAuth 2.0 is used, the value is the redirection URI. When OpenID Connect is used, the value is the issuer value of the authorization server.	A resolvable URI such as a URL or a DID

sub	Token subject identifier	**[R32]** Sub **MUST** be a unique and non-repeating identifier for the subject, i.e. the Buyer. **[R33]** The sub identifier **MUST** be the same when created by the Authorization and Token endpoints during the Hybrid flow.	**Non-Identity Services Providers** will use the Intent/Consent ID for this field. **Identity Services Providers** will choose a value at the discretion of the SP's.
meflso_intent_id	Intent ID of the originating request	**[R34]** meflso_intent_id **MUST** be a unique and non-repeating identifier containing the intent_id. **[O6]** This field **MAY** duplicate the value in "sub" for many providers.	Use the Intent/Consent ID for this field.
aud	Audience that the ID token is intended for	**[R35]** OpenID Connect protocol mandates aud **MUST** include the client ID of the TPP/SP. See also the FAPI Read Write / OpenID Standard [26].	See requirement
exp	Token expiration date/time	**[R36]** Exp **MUST** be included in the Claim ID token The validity length is set at the discretion of the SPs such that it does not impact the functionality of the APIs. For example, an expiry time of 1 second is insufficient for all Resource Requests.	Expressed as an epoch, i.e., number of seconds from 1970-01-01T0:0:0Z as measured in UTC. RFC 7519 [14]
iat	Token issuance date/time	**[R37]** The iat property **MUST** be included in the Claim ID token	Expressed as an epoch, i.e., number of seconds from 1970-01-01T0:0:0Z as measured in UTC.
auth_time	Date/time when End User was authorised	**[O7]** The max_age property **MAY** be requested in the Claim ID Token. **[CR2]< [O2]** If the max_age request is made or max_age is included as an essential claim, auth_time **MUST** be supported by the SP.	Expressed as an epoch, i.e., number of seconds from 1970-01-01T0:0:0Z as measured in UTC.

nonce	Used to help mitigate against replay attacks	**[R38]** The nonce property **MUST** be in the Claim ID Token The nonce value is passed in as a Request parameter. **[R39]** The nonce **MUST** be replayed in the ID token when the token is utilized in a subsequent access request.	
acr	Authentication Context Class Reference	**[R40]** The acr property MUST be included in the Claim ID Token The acr is an identifier that qualifies what conditions were satisfied when the authentication was performed. **[D5]** The acr **SHOULD** correspond to one of the values requested by the acr_values field on the request. However, even if not present on the request, the SP should populate the acr with a value that attests that the SP performed or NOT performed an appropriate level of authentication such that the SP believes it has met the requirement for "Strong Customer Authentication" (SCA). SPs that do not wish to provide this as a claim should remove it from the well-known configuration endpoint. As per OIDC Core, marking a claim as "essential" and a SP cannot fulfil it, then an error should not be generated.	The values to be provided are **urn:mef:lso:security:oidc:acr:ca** or **urn:mef:lso:security:oidc:acr:sca**.
amr	Authentication Methods References	The amr property specifies the methods that are used in the authentication. For example, this field might contain indicators that a password was supplied. Note that the industry direction is to consolidate on Vectors of Trust: RFC 8485 [17]. Hence, this field may be replaced shortly. Also note that amr does not give the flexibility to address all the actual particulars of both the authentication and the identity that is utilized.	

azp	Authorized party	The azp property is the authorized party to which the ID Token was issued. **[O8]** The azp property MAY be present in the Claim ID Token. **[CR3]<**[O3] If the azp property is present, it **MUST** contain the OAuth 2.0 Client ID of this party. This Claim is only needed when the ID Token has a single audience value, and that audience is different than the authorized party. It may be included even when the authorized party is the same as the sole audience.	A resolvable URI such as a URL or a DID
s_hash	State Hash Value	**[D6]**The s_hash property **SHOULD** be present in the Claim ID Token The state hash, s_hash, in the ID Token is to protect the state value.	Its value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the state value, where the hash algorithm used is the hash algorithm used in the algHeader Parameter of the ID Token's JOSE Header. For instance, if the alg is HS512, hash the code value with SHA-512, then take the left-most 256 bits and base64url encode them. The s_hashvalue is a case sensitive string.

at_hash	Access Token Hash Value	**[O9]** The Claim ID Token MAY be issued from the Authorization Endpoint with an access_token value. **[CR4]<**[O4] The at_hash property **MUST** be included in the Claim ID Token	Its value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the access_token value, where the hash algorithm used is the hash algorithm used in the alg Header Parameter of the ID Token's JOSE Header. For instance, if the alg is RS256, hash the access_token value with SHA-256, then take the left-most 128 bits and base64url encode them. The at_hash value is a case sensitive string.
c_hash	Code hash value.	**[O10]** The Claim ID Token MAY be issued from the Authorization Endpoint with a code. **[CR5]<**[O5] The c_hash property **MUST** be included in the Claim ID Token	Its value is the base64url encoding of the left-most half of the hash of the octets of the ASCII representation of the code value, where the hash algorithm used is the hash algorithm used in the alg Header Parameter of the ID Token's JOSE Header.

Table 3 – ID Token Claims Details

7 JWT Security Suite Information v1.0

This document utilizes, and where required concretizes for the usage with this standard, the JOSE standard v1.0 [9].

[R41] All JOSE standard v1.0 requirements **MUST** be implemented unless otherwise explicitly indicated in this document.

7.1 General Guidance for JWT Best Practice

See RFC 8725 [18] for the recommended JWT approach.

7.2 JSON Web Key Set (JWKS) Endpoints

Upon issuance of a certificate from a JWKS [12] hosting service, a JWK Set is created or updated for a given TPP/SP.

[D7] All participants **SHOULD** include the "kid" and "jku" properties of the key that was used to sign the payloads in the JWKS issuance request.

[D8] The JKU property **SHOULD** be considered a hint only and relying parties should derive and then validate wherever possible the appropriate JWKS endpoint for the message signer.

See [12], section 4.

Note that as certificates are added and removed the JWKS endpoint is updated automatically.

7.3 General outline for creating a JWS

There are 5 steps that must be followed to create a JWS. These steps are detailed in sections 7.3.1 to 7.3.5.

7.3.1 Step 1: Select the certificate and private key to sign the JWS

[R42] As the JWS is used for non-repudiation, it **MUST** be signed using one of JWS issuer's private keys.

[R43] The private key **MUST** have been used by the issuer to get a signing certificate issued from an identity provider.

[R44] The signing certificate **MUST** be verifiably valid at the time of creating the JWS.

7.3.2 Step 2: Form the JOSE Header

[R45] The JWS JOSE header is a JSON object which **MUST** consist of minimally two fields, also called the claims, as specified in Table 4:

Claim	Description
alg	The algorithm to use for signing the JWS. **[R46]** The alg property **MUST** be taken from the list of valid JOSE algorithms can be found in IANA JOSE [4], registry JSON Web Signature and Encryption Algorithms. In addition, this document recommends the following algorithms: **[D9]**ED25519, also as a JWK, with sha3-256 as the hashing algorithm **SHOULD** also be used as an algorithm for JWS signing
kid	The “kid” (key ID) Header Parameter is a hint indicating which key was used to secure the JWS. **[R47]** The kid property **MUST** match the certificate id of the certificate selected in step 1. **[D10]**The receiver **SHOULD** use this value to identify the certificate to be used for verifying the JWS.

Table 4 – Forming the JOSE Header

7.3.3 Step 3: Form the payload to be signed

The JSON payload to be signed must have the following claims:

Claim	Description
iss	The issuer of the JWS. **[R48]** The iss property **MUST** match the dn of the certificate selected in step 1.

Table 5 – Signing the JSON Payload

The payload to be signed is computed as:

```
payload = base64Encode (JOSEHeader) + “.” + base64Encode(json)
```

Where:

- **JOSEHeader:** is the header created in Step 2 and
- **json:** is the message for the original data to be sent

7.3.4 Step 4: Sign and encode the payload

The signed payload is computed as follows:

```
signedAndEncodedPayload = base64Encode (encrypt(privateKey, payload))
```

Where:

- **privateKey:** is the private key selected in step 1
- **payload:** is the payload computed in Step 3
- **encrypt:** Is an encryption function that implements the `alg` identified in Step 2.

7.3.5 Step 5: Assemble the JWS

The JWS is computed as follows:

```
JWS = payload + "." + signedAndEncodedPayloadWhere:
```

- **payload:** is the payload computed in Step 3
- **signedAndEncodedPayload:** is the signed element computed in Step 5.

7.4 General Outline for creating a JWE

The implementation guide is based on RFC 7516 [11]**.**

JSON Web Encryption (JWE) represents encrypted content using JSON data structures and base64url encoding. These JSON data structures may contain whitespace and/or line breaks before or after any JSON values or structural characters, in accordance with Section 2 of RFC 7516 [11]. A JWE represents these logical values:

- JOSE Header
- JWE Encrypted Key
- JWE Initialization Vector
- JWE AAD (Additional Authenticated Data)
- JWE Ciphertext
- JWE Authentication Tag

For a JWE, the JOSE Header members are the union of the members of these values:

- JWE Protected Header
- JWE Shared Unprotected Header
- JWE Per-Recipient Unprotected Header

JWE utilizes authenticated encryption to ensure the confidentiality and integrity of the plaintext and the integrity of the JWE Protected Header and the JWE AAD.

This document recommends the following for the JWE Compact Serialization as a representation:

[D11]JWE Shared Unprotected Header or JWE Per-Recipient Unprotected Header **SHOULD** not be used.

In this case, the JOSE Header and the JWE Protected Header are the same.

In this serialization, the JWE is represented as the following concatenation:

```
BASE64URL(UTF8(JWE Protected Header)) || '.' ||
BASE64URL(JWE Encrypted Key) || '.' ||
BASE64URL(JWE Initialization Vector) || '.' ||
BASE64URL(JWE Ciphertext) || '.' ||
BASE64URL(JWE Authentication Tag)
```

7.4.1 Step 1: Select the certificate and private key to sign the JWE

[R49] As the JWS is used for non-repudiation, it **MUST** be signed using one of JWS issuer's private keys.

[R50] The private key **MUST** have been used by the issuer to get a signing certificate issued from an identity provider.

[R51] The signing certificate **MUST** be verifiably valid at the time of creating the JWE.

7.4.2 Step 2: Form the JOSE Header of the JWE

[R52] The JWE JOSE header is a JSON object which **MUST** consist of minimally four fields, also called the claims, as specified in Table 6:

Claim	Description
alg	The algorithm to use for signing the JWS. **[R53]** The alg property **MUST** be taken from the list of valid JOSE algorithms in RFC 7518 [13], section 3.1 **[R54]** The NULL cipher **MUST NOT** be used as an alg value in JWTs. In addition, this document recommends the following algorithms: **[D12]**ED25519, also as a JWK, with sha3-256 as the hashing algorithm **SHOULD** be used.
kid	The "kid" (key ID) Header Parameter is a hint indicating which key was used to secure the JWS. **[R55]** The kid property **MUST** match the certificate id of the certificate selected in step 1. **[D13]**The receiver **SHOULD** use this value to identify the certificate to be used for verifying the JWS.

enc	The “enc” (encryption algorithm) Header Parameter identifies the content encryption algorithm used to perform authenticated encryption on the plaintext to produce the ciphertext and the Authentication Tag. **[R56]** The selected encryption algorithm **MUST** be an AEAD algorithm with a specified key length. The encrypted content is not usable if the “enc” value does not represent a supported algorithm. **[D14]**“enc” values **SHOULD** either be registered in the IANA “JSON Web Signature and Encryption Algorithms” registry established by [4] or be a value that contains a Collision-Resistant Name. The “enc” value is a case-sensitive ASCII string containing a String Or URI value. **[R57]** The “enc” property **MUST** be present **[R58]** The “enc” property **MUST** be understood and processed by implementations. A list of defined "enc" values for this use can be found in the IANA registry established in IANA JOSE [4], with the initial contents of this registry are the values defined in registry “JSON Web Signature and Encryption Algorithms”.
access-jwk	This parameter has the same meaning, syntax, and processing rules as the “jwk” Header Parameter defined in Section 7.1.3 of RFC 7516 [11], except that the key is the public key to which the JWE was encrypted with; this can be used to determine the private key needed to decrypt the JWE.

Table 6 – Forming the JOSE Header of the JWE

7.4.3 Step 3: Form the encryption key, initialization vector and AAD

1. Determine the Key Management Mode employed by the algorithm used to determine the Content Encryption Key value (set in “alg”).
2. When Key Wrapping, Key Encryption, or Key Agreement with Key Wrapping are employed, generate a random CEK value. See RFC 4086 [6] for considerations on generating random values.

[R59] The CEK **MUST** have a length equal to that required for the content encryption algorithm.

3. When Direct Key Agreement or Key Agreement with Key Wrapping are employed, use the key agreement algorithm to compute the value of the agreed upon key. When Direct Key Agreement is employed, let the CEK be the agreed upon key. When Key Agreement with Key Wrapping is employed, the agreed upon key is used to wrap the CEK.
4. When Key Wrapping, Key Encryption, or Key Agreement with Key Wrapping are employed, encrypt the CEK to the recipient and let the result be the JWE Encrypted Key.
5. When Direct Key Agreement or Direct Encryption are employed, let the JWE Encrypted Key be the empty octet sequence.

6. When Direct Encryption is employed, let the CEK be the shared symmetric key.
7. Compute the encoded key value BASE64URL(JWE Encrypted Key).
8. Generate a random JWE Initialization Vector of the correct size for the content encryption algorithm (if required for the algorithm); otherwise, let the JWE Initialization Vector be the empty octet sequence.
9. Compute the encoded Initialization Vector value BASE64URL(JWE Initialization Vector).
10. Create the JSON object(s) containing the desired set of Header Parameters, which together comprise the JOSE Header: one or more of the JWE Protected Header. There are no unprotected headers in the JWE compact serialization representation.
11. Compute the Encoded Protected Header value BASE64URL(UTF8(JWE Protected Header)).
12. Let the Additional Authenticated Data encryption parameter be ASCII(Encoded Protected Header).

7.4.4 Step 4: Form the JWE Ciphertext and final JWE

The JSON payload to be encrypted must have the claims defined in Table 7.

Claim	Description
iss	The issuer of the JWS. **[R60]** The iss property **MUST** match the dn of the certificate selected in Step 1, section 7.4.1.

Table 7 – JWS /JWE issuer property

1. Encrypt the BASE64URL (JSON message) using the CEK, the JWE Initialization Vector, and the Additional Authenticated Data value using the specified content encryption algorithm to create the JWE Ciphertext value and the JWE Authentication Tag (which is the Authentication Tag output from the encryption operation).
2. Compute the encoded ciphertext value BASE64URL(JWE Ciphertext).
3. Compute the encoded Authentication Tag value BASE64URL(JWE Authentication Tag).
4. If a JWE AAD value is present, compute the encoded AAD value BASE64URL(JWE AAD).
5. Create the desired serialized output. The Compact Serialization of this result is the string BASE64URL(UTF8(JWE Protected Header)) || '.' || BASE64URL(JWE Encrypted Key) || '.' ||BASE64URL(JWE Initialization Vector) || '.' || BASE64URL(JWE Ciphertext) || '.' || BASE64URL(JWE Authentication Tag).

8 LSO API Payload Authenticity

Up to this point we have only discussed security of the LSO API payload and LSO API response as described in the previous section. However, of equal importance is LSO API payload and LSO API response authenticity since the LSO API payload and LSO API response may be constructed by an entity other than Buyer or Seller. Therefore, this document only focuses on the authenticity of the LSO API payload and LSO API response since the authenticity of the Subject and Seller have already been established before an LSO API payload and LSO API response is authenticated.

LSO API payload / response authenticity is a special case of Message Authenticity which is defined as the outcome of message authentication, which is defined in NIST SP 800-152 [20] as a process that provides assurance of the integrity of messages, documents, or stored data. The following requirements are focused on authenticity and privacy.

[R61] Delegation of Trust **MUST NOT** be permitted if Buyer / Seller and their intended delegates are not in the same Trust Domain

Delegation of Trust refers to the process whereby a Buyer / Seller imparts their inherent level of trust within their Trust Domain to another Buyer / Seller.

Message Authenticity, and therefore, LSO API payload / response authenticity, in the context of this document specifies how a Message Payload needs to be structured such that it can be authenticated independent of the authentication of a Buyer or Seller.

[D15]To ensure Message Authenticity for a request from the Buyer to the Seller, the semantics of a Message Payload **SHOULD** contain the elements of Table 8.

Element	Example
A previously established shared secret between Subject and Seller	An alphanumeric string such as “ABC1234X7CV5”
A new shared secret between Buyer and Seller	An alphanumeric string such as “CBA1234X7CV5”
A domain identifier for the next response from Seller to Buyer, if the Buyer’s domain identifier changes compared to the domain identifier of the Buyer’s request	google.com
An endpoint identifier for the next response from Seller to Buyer, if the Buyer’s domain identifier changes compared to the domain identifier of the Buyer’s request	/quotemanagement/notification

Table 8 – Message Payload Request Required Elements

[D16]To ensure Message Authenticity for a response from the Seller to the Buyer, the semantics of a Message Payload **SHOULD** contain the elements of Table 8 where the roles of Buyer and Seller are reversed.

[D17]All Policies in a Buyer's or Seller's Trust Domain **SHOULD** enforce [D15] and [D16]

9 References

[1] CNSSI 4009, *Committee on National Security Systems Glossary,* April 2015

[2] ECMA JSON, *The JSON Data Interchange Syntax, 2nd Edition,* December 2017

[3] Fielding, Roy Thomas, *Architectural Styles and the Design of Network-based Software Architectures,* 2000

[4] IANA JOSE, *JSON Object Signing and Encryption (JOSE),* November 2020

[5] IETF RFC 2119, *Key words for use in RFCs to Indicate Requirement Levels,* March 1997

[6] IETF RFC 4086, *Randomness Requirements for Security,* June 2005

[7] IETF RFC 6749, *The OAuth 2.0 Authorization Framework,* October 2012

[8] IETF RFC 6819, *OAuth 2.0 Threat Model and Security Considerations,* January 2013

[9] IETF RFC 7165, *Use Cases and Requirements for JSON Object Signing and Encryption (JOSE)*, April 2014

[10] IETF RFC 7515, *JSON Web Signature (JWS),* May 2015

[11] IETF RFC 7516, *JSON Web Encryption (JWE),* May 2015

[12] IETF RFC 7517, *JSON Web Key (JWK)*, May 2015

[13] IETF RFC 7518, *JSON Web Algorithms (JWA),* March 2015

[14] IETF RFC 7519, *JSON Web Token (JWT),* May 2015

[15] IETF RFC 7591, *OAuth 2.0 Dynamic Client Registration Protocol*, July 2015

[16] IETF RFC 8174, *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words,* May 2017

[17] IETF RFC 8485, *Vectors of Trust,* October 2018

[18] IETF RFC 8725, *JSON Web Token Best Current Practices,* February 2020

[19] MEF 55.1, *Lifecycle Service Orchestration (LSO): Reference Architecture and Framework,* January 2021

[20] NIST SP 800-152, *A Profile for U.S. Federal Cryptographic Key Management Systems,* October 2015

[21] Open Banking, *Read/Write Data API Specification v3.1.9,* May 2019

[22] Open Banking, *Security Profile Implementer's Draft v1.1.2,* February 2018

[23] OpenID, *OpenID Connect Core 1.0,* November 2014

[24] OpenID, *OpenID Connect Registration 1.0,* November 2014

[25] OpenID, *OpenID Connect Discovery 1.0,* November 2014

[26] OpenID, *Financial-grade API Security Profile 1.0 – Part 1: Baseline,* March 2021

[27] W3C DID, *Decentralized Identifiers (DIDs) v1.0,* July 2022

[28] W3C VCDM, *Verifiable Credentials Data Model 1.1,* November 2021

Appendix A Why Decentralized Public Key Infrastructure? (Informative)

Currently 3rd parties such as Domain Name Services (DNS) registrars, the Internet Corporation for Assigned Names and Numbers (ICANN), X.509 Certificate Authorities (CAs), or social media companies are responsible for the creation and management of online identifiers and the secure communication between them.

As evidenced over the last 20+ years, this design has demonstrated serious usability and security shortcomings.

When DNS and X.509 Public Key Infrastructure (PKIX) as described in NIST publication SP 800-32 was designed, the internet did not have a way to agree upon the state of a registry (or database) in a reliable manner with no trust assumptions. Consequently, standard bodies designated trusted 3rd parties (TTPs) to manage identifiers and public keys. Today, virtually all Internet software relies on these authorities. These trusted 3rd parties, however, are central points of failure, where each could compromise the integrity and security of large portions of the Internet. Therefore, once a TTP has been compromised, the usability of the identifiers it manages is also compromised.

As a result, companies spend significant resources fighting security breaches caused by CAs, and public internet communications that are both truly secure and user-friendly are still out of reach for most.

Therefore, this standard suggests an identity approach where every identity is controlled by its Principal Owner and not by a 3rd party, unless the Principal Owner has delegated control to a 3rd party. A Principal Owner is defined as the entity controlling the public key(s) which control the identity and its identifiers upon inception of the identity.

Identity in the context of this document is to mean the following:

```
Identity = <Identifier(s)> + <associated data>
```

where associated data refers to data describing the characteristics of the identity that is associated with the identifier(s). An example of such associated data could be an X.509 issues by a CA.

Such an approach suggests a decentralized, or at least strongly federated, infrastructure. Decentralized in this context means that there is no single point of failure in the PKI where possibly no participants are known to one another. And strongly federated in this context means that there is a known, finite number of participants, without a single point of failure in the PKI. However, a collusion of a limited number of participants in the federated infrastructure may still lead to a compromised PKI. The consensus thresholds required for a change in the infrastructure needs to be defined by each identity federation.

For a LSO APIs to properly operate, communication must be trusted and secure. Communications are secured through the safe delivery of public keys tied to identities. The Principal Owner of the identity uses a corresponding secret private key to both decrypt messages sent to them, and to prove they sent a message by signing it with its private key.

PKI systems are responsible for the secure delivery of public keys. However, the commonly used X.509 PKI (PKIX) undermines both the creation and the secure delivery of these keys.

In PKIX services are secured through the creation of keys signed by CAs. However, the complexity of generating and managing keys and certificates in PKIX have caused companies to manage the creation and signing of these keys themselves, rather than leaving it to their clients. This creates major security concerns from the outset, as it results in the accumulation of private keys at a central point of failure, making it possible for anyone with access to that repository of keys to compromise the security of connections in a way that is virtually undetectable.

The design of X.509 PKIX also permits any of the thousands of CAs to impersonate any website or web service. Therefore, entities cannot be certain that their communications are not being compromised by a fraudulent certificate allowing a PITM (Person-in-the-Middle) attack. While workarounds have been proposed, good ones do not exist yet.

Decentralized Public Key Infrastructure (DPKI) has been proposed as a secure alternative. The goal of DPKI is to ensure that, unlike PKIX, no single third-party can compromise the integrity and security of a system employing DPKI as a whole.

Within DPKI, a Principal Owner can be given direct control and ownership of a globally readable identifier by registering the identifier for example in a Distributed Ledger, often referred to as a Blockchain, or other system that guarantees data integrity without a central point of failure. Simultaneously, Distributed Ledgers allow for the assignment of arbitrary data such as public keys to these identifiers and permit those values to be globally readable in a secure manner that is not vulnerable to the PITM attacks that are possible in PKIX. This is done by linking an identifier's lookup value to the latest and most correct public keys for that identifier. In this design, control over the identifier is returned to the Principal Owner.

Therefore, it is no longer trivial for any one entity to undermine the security of the entire DKPI system or to compromise an identifier that is not theirs overcoming the challenges of typical PKI.

Furthermore, DPKI requires a public registry of identifiers and their associated public keys that can be read by anyone but cannot be compromised. As long as this registration remains valid, and the Principal Owner is able to maintain control of their private key, no 3rd party can take ownership of that identifier without resorting to direct coercion of the Principal Owner. Any Principal Owner in a DPKI system must be able to broadcast a message if it is well-formed within the context of the DPKI. Other peers in the system do not require admission control. This implies a decentralized consensus mechanism naturally leading to the utilization of systems such as distributed ledgers. Therefore, given two or more histories of updates, any Principal Owner must be able to determine which one is preferred due to security by inspection. This implies the existence of a method of ascertaining the level of resources backing a DPKI history such as the hash power in the Bitcoin blockchain based on difficulty level and nonce.

Requirements of identifier registration in DPKI is handled differently from DNS. Although registrars may exist in DPKI, these registrars must adhere to several requirements that ensure that identities belong to the entities they represent. This is achieved the following way:

- Private keys must be generated in a manner that ensures they remain under the Principal Owner's control.
- Generating key pairs on behalf of Principal Owner must not be allowed.
- Principals Owners must always be in control of their identifiers and the corresponding public keys. However, Principal Owners may extend control of their identifier to third parties, if they prefer, for example for public key recovery purposes.
- Extension of control of identifiers to 3rd parties must be an explicit, informed decision by the Principal Owner of such identifier.
- Private keys must be stored and/or transmitted in a secure manner.
- No mechanism should exist that would allow a single entity to deprive a Principal Owner of their identifier without their consent. This implies that:
  - Once a namespace for an identity is created it must not be possible to destroy it.
  - Namespaces in a DPKI must not contain blacklisting mechanisms that would allow anyone to invalidate identifiers that do not belong to them.
  - Once set, namespace rules within a DPKI must not be altered to introduce any new restrictions for renewing or updating identifiers. Otherwise, it would be possible to take control of identifiers away from Principals Owners without their consent.
- The rules for registering and renewing identifiers in a DPKI must be transparent and expressed in simple terms.

Note that if registration is used as security to an expiration or other policy, the Principal Owner must be explicitly and timely warned that failure to renew the registration on time could result in the Principal Owner losing control of the identifier.

- Also, within a DPKI, processes for renewing or updating identifiers must not be modified to introduce new restrictions for updating or renewing an identifier, once issued.
- Finally, within a DPKI all network communications for creating, updating, renewing, or deleting identifiers must be sent via a non-centralized mechanism. This is necessary to ensure that a single entity cannot prevent identifiers from being updated or renewed.

While it might not yet be common practice to implement DPKI, DPKI mitigates the PKIX threat model, and is either already in use as with the state government of British Columbia in Canada, or under active development and regulatory consideration as within EU countries such as Germany to meet the EU's General Data Privacy Regulation directive or with the Department of Homeland Security in the US.

Appendix B Developers' Guide (Informative)

Overview

This section provides an implementation perspective of the MEF LSO API Security Profile. For generality, this document will use an abstracted API model. Any application to a specific API is simply a swapping out of the relevant API data model. The Appendix will cover:

1. Specified Behavior
2. Non-Specified Behavior
3. Success Flows
4. Edge Cases

Specified Behavior

The implementation of the abstracted API is based on the known configurations listed in this section and subsections.

Client Types

As per the OAuth 2.0 specification [7], section 2.1, the Confidential Client Type is illustrated in the sample API as it can maintain its own credentials.

Grant Types

OIDC Hybrid Flow (response_type = code id_token)

- The sample API illustrates the use of the *request_type = code id_token* for the OIDC Hybrid Flow implementation.

The SP may optionally choose to return Refresh Tokens for the Hybrid Grant Flow when issuing an access token.

Client Credentials Grant Type using multiple scopes (scope = specific functions)

- The Client Credentials Grant Type (RFC 6749 [7], section 4.4) is only used when the TPP/SP requires an access token (on behalf of itself) to access an API resource e.g.
    - Quotes:

```
POST /quote
GET /quote-submissions/{QuoteSubmissionId}
```

Figure 8 – Client Credential Type Using Multiple Scopes

- In this example, an SP enables the same Confidential Client (ClientId) access to an API called Quote. A TPP/SP may, therefore, choose to request either a single scope or multiple scope(s) as the TPP/SP may want to use the *same* access token across multiple API e.g., Quote and Order.
- Only valid API scopes are accepted when generating an access token, for example *POST /quote* or *GET/quote-submissions*.
- Access tokens generated by a Client Credentials grant may not return any refresh tokens (as per the OAuth 2.0 specification [7]).
- Scopes are delimited by using a comma, for example POST /quote, GET /hub.

Access Tokens

- For one or more APIs, the access token must be obtained within a secure, server-side context between the TPP/SP and the SP.
- Access Tokens must be validated by the TPP/SP as outlined within RFC 6749 [7].

Refresh Tokens

- SPs may optionally return a refresh token [24] when an authorization request is successfully processed at the token endpoint. The Hybrid Grant Flow supports the provisioning of refresh tokens.
- The sample API implementation cites an example for SPs requesting a Refresh Token to refresh an expired access token prior to invoking the /quote resource.
- Refresh Tokens must be validated as outlined in OpenID Registration [24].

ID Tokens

- ID Tokens must be validated by the TPP/SP as outlined in OpenID Registration [24].
- TPPs/SPs must use the *meflso_intent_id* claim to populate and retrieve the IntentID, e.g., QuoteID in our example, for any required validation.
- The full set of claims that can be represented within an ID Token are documented in the Request Object and ID Token Section of this document.

Authorization Codes

- Authorization codes must be validated by the TPP/SP as outlined in RFC 6749 [7].

Non-Specified Behavior

The current MEF LSO APIs are not specified for the following configurations:

Client Types

- As per the OAuth 2.0 specification [7], section 2.1, the Public Client Type has not been defined for MEF LSO APIs.

Grant Types

OIDC Hybrid Flow (response_type = code id_token token or response_type = code token)

- Forces an access token to be returned from the SP authorization endpoint (instead of a token endpoint).

OIDC Implicit Flow (response_type=id_token token or response_type=id_token)

- The Implicit Flow does not authenticate the Client that is invoking the request.

Client Credentials Grant Type (scope=openid email profile address phone)

- Requesting OIDC specific scopes or any non-specified scopes when using the Client Credentials grant.

Validity Lengths (Authorization Code, Access Token, ID Token, Refresh Token)

Each SP's authorization / resource server is configured independently to comply with internal SP security policies and guidelines. The LSO API specifications do not mandate validity lengths.

Authorization Code

- The OAuth 2.0 Specification [7] suggests an authorization code should be short lived to a maximum of 10 minutes. Any codes exceeding this limit are to be rejected.

ID Token

- ID Token claims (*exp* and *iat)* determine its validity.
- Returned with the authorization code when the Hybrid Grant Flow (code id_token) is initiated.

Access Token

- The *expires_in* attribute returned by the authorization server when an access token is generated determines its validity.
- Access tokens are generally short lived, and when they expire, are then exchanged for another using a longer-lived refresh token.
- Refer to Section 16.18 of OpenID Connect Core [23], Lifetimes of Access Tokens and Refresh Tokens.

Refresh Token

- The *expires_in* attribute returned by the authorization server when a refresh token is generated determines its validity.
- Refresh tokens are generally longer lived in comparison to access tokens.
- Refer to Section 16.18 of OpenID Connect Core [23], Lifetimes of Access Tokens and Refresh Tokens.

Success Flows

This subsection describes the success flow path of proper client application authentication and authorization using the sample API.

Quote API Specification

The sequence diagram in Figure 9 highlights the standard OAuth 2.0 Client Credentials Grant and OIDC Hybrid Grant flow with intent that are used by the sample API.

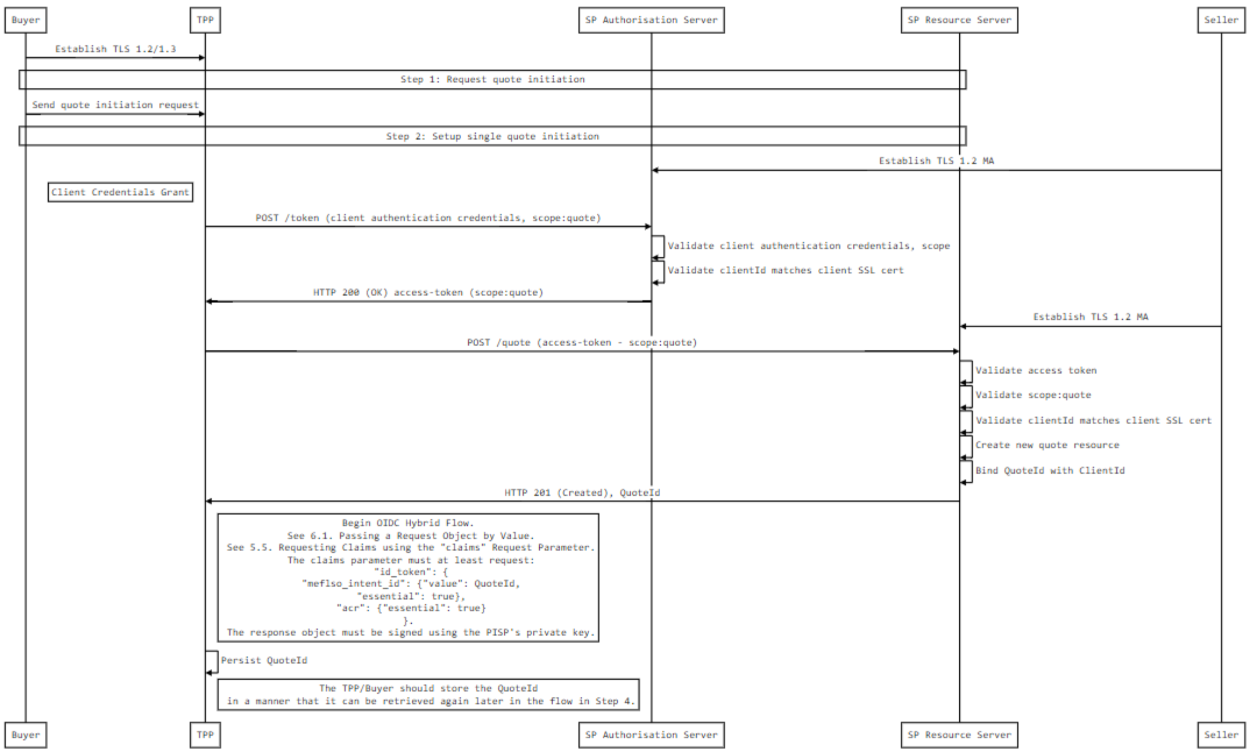

Figure 9 – Sample Quote API OAuth2/OIDC Authentication/Authorization Flow

Client Credentials Grant Type (OAuth 2.0)

Summary

This grant type is used by the Buyer (through the TPP) in Step 2 to setup a single quote with the Seller (SP).

1. The TPP initiates an authorization request using valid Client Credentials Grant (RFC 6749 [7], section 4.4) type and scope(s).
2. The SP authorization server validates the Client Authentication request from the TPP and generates an access token response when the request is valid.
3. The TPP uses the access token to create a new Quote resource against the SP resource server.
4. The SP resource server responds with the QuoteId for the resource it has created.

5. The Client Credentials Grant may optionally be used by the TPP in Step 5 to retrieve the status of a Quote or Quote-Submission where no active access token is available.

OIDC Hybrid Flow

Summary

- The Hybrid Grant flow [22] is the recommendation from the MEF LSO Security Profile and the FAPI Specification [26] for FAPI Read/Write.
- This is initiated at the end of Step 2 by the TPP after the QuoteId is generated by the SP and returned to the TPP.
- This is used in a redirect across the Buyer and Seller (SP) in Step 3 for the Buyer to authorize consent with the SP – for the TPP to proceed with the Quote.
- This is used across the TPP and SP in Step 4 by exchanging the authorization code for an access token to create the Quote-Submission resource.

HTTP Request and Response Examples

Step 1 – Request Quote Initiation

There are no requests and responses against the sample Quote API in this Step for the Buyer, TPP and Seller/SP.

Step 2 – Setup Single Quote Initiation

TPP obtains an access token using a Client Credentials Grant Type. The scope *quote* must be used. When an access token expires, the TPP will need to re-request for another access token using the same request shown in Table 9 as an example.

Request: Client Credentials using private_key_jwt	Response: Client Credentials

POST /as/token.oauth2 HTTP/1.1
Host: https://authn.acme.com
Content-Type: application/x-www-form-urlencoded
Accept: application/json
grant_type=client_credentials
&scope=quote
&client_assertion_type=
    urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRw
czovL2p3dC1pZHAuZXhhbXBsZS5jb20iLCJzdWIiOiJtYWlsdG86bWlrZUBleGFtcGxlLmN
vbSIsIm5iZiI6MTQ5OTE4MzYwMSwiZXhwIjoxNDk5MTg3MjAxLCJpYXQiOjE0OTkxODM2MD
EsImp0aSI6ImlkMTIzNDU2IiwidHlwIjoiaHR0cHM6Ly9leGFtcGxlLmNvbS9yZWdpc3Rlc
iJ9.SAxPMaJK_wYl_W2idTQASjiEZ4UoI7-P2SbmnHKr6LvP8ZJZX6JlnpK_xClJswAni1T
p1UnHJslc08JrexctaeEIBrqwHG18iBcWKjhHK2Tv5m4nbTsSi1MFQOlMUTRFq3_LQiHqV2
M8Hf1v9q9YaQqxDa4MK0asDUtE_zYMHz8kKDb-jj-Vh4mVDeM4_FPiffd2C5ckjkrZBNOK0
01Xktm7xTqX6fk56KTrejeA4x6D_1ygJcGfjZCv6Knki7Jl-6MfwUKb9ZoZ9LiwHf5lLXPuy
_QrOyM0pONWKj9K4Mj7I4GPGvzyVqpaZUgjcOaZY_rlu_p9tnSlE781dDLuw
 	
{
  "alg": "RS256",
  "kid": "12345",
  "typ": "JWT"
}
.
{
  "iss": "s6BhdRkqt3",
  "sub": "s6BhdRkqt3",
  "exp": 1499187201,
  "iat": 1499183601,
  "jti": "id123456",
  "aud": "https://authn.acme.com/as/token.oauth2"
}
.
<<signature>>
	HTTP/1.1 200 Success
	Content-Length: 1103
	Content-Type:
	application/json
	Date: Mon, 26 Jun 2022
	15:18:28 GMT
	{
	  "alg": "RS256",
	  "kid": "12347",
	  "typ": "JWT"
	}
	.
	{
	    "access_token":
	"2YotnFZFEjr1zCsicMWpAA",
	    "expires_in": 3600,
	    "token_type":
	"bearer",
	    "scope":"quote"
	}
	.
	<<signature>>

Table 9 – Non-Base64 JWT client_assertion

Then the TPP uses the access token (with *quote* scope) from the SP to invoke the sample Quote API.

Request: Quote API	**Response: Quote API**

POST /quote HTTP/1.1
Authorization: Bearer 2YotnFZFEjr1zCsicMWpAA
x-idempotency-key: FRESCO.21302.GFX.20
x-fapi-mef-id: mef/2021/011
x-fapi-buyer-last-logged-time: 2021-06-13T11:36:09
x-fapi-buyer-ip-address: 104.25.212.99
x-fapi-interaction-id: 93bac548-d2de-4546-b106-880a5018460d
Content-Type: application/json
Accept: application/json
 	
{
"alg": "RS256",
"kid": "12345",
"typ": "JWT"
}
.
{
"Data": {…}
}
.
<<signature>>
	HTTP/1.1 201 Created
	Content-Type: application/json
	x-fapi-interaction-id: 93bac548-d2de-4546-b106-880a5018460d
	
	{
	"alg": "RS256",
	"kid": "12347",
	"typ": "JWT"
	}
	.
	
	{
	"Data": {…}
	}
	.
	<<signature>>

Table 10 – Single Quote Initiation

Step 3 - Authorize Consent

Then the TPP receives a QuoteId from the SP (Seller). The TPP then creates an authorization request (using a signed, and possibly encrypted, JWT request containing the QuoteId as a claim) for the Buyer/TPP to consent to the Quote directly with their Seller/SP. The request is an OIDC Hybrid Grant flow (requesting for code and id_token).

Request: OIDC Hybrid Grant Flow	**Response: OIDC Hybrid Grant Flow**

Sourced from the MEF LSO Security Profile Request Object section
 	
GET /authorize?
response_type=code id_token
&client_id=s6BhdRkqt3
&state=af0ifjsldkj
&scope=openid quote
&nonce=n-0S6_WzA2Mj
&redirect_uri=https://api.mytpp.com/cb
&request=CJleHAiOjE0OTUxOTk1ODd.....JjVqsDuushgpwp0E.5leGFtcGxlI iwianRpIjoiM....JleHAiOjE0.olnx_YKAm2J1rbpOP8wGhi1BDNHJjVqsDuushgp wp0E
 	
{
"alg": "",
"kid": "GxlIiwianVqsDuushgjE0OTUxOTk"
}
.
{
"iss": "https://api.acme.com",
"aud": "s6BhdRkqt3",
"response_type": "code id_token",
"client_id": "s6BhdRkqt3",
"redirect_uri": "https://api.mytpp.com/cb",
"scope": "openid , POST /quote, GET /quote",
	After the Buyer has consented directly with the SP the SP validates the authorization request and generates an auth code and ID token
	
	HTTP/1.1 302 Found
	Location:
	https://api.mytpp.com/cb#
	
	code=SplxlOBeZQQYbYS6WxSbIA
	&id_token=eyJ0 ...
	NiJ9.eyJ1c ...
	I6IjIifX0.DeWt4Qu ... ZXso
	&state=af0ifjsldkj

```
   "state": "af0ifjsldkj",
   "nonce": "n-0S6_WzA2Mj",
   "max_age": 86400,
   "claims":
    {
     "userinfo":
      {
       "meflso_intent_id": {"value": "urn:acme:intent:58923",
"essential": true}
      },
     "id_token":
      {
       "meflso_intent_id": {"value": "urn:acme:intent:58923",
"essential": true},
       "acr": {"essential": true,
                "values": ["urn:mef:lso:security:oidc:acr:sca"
                  ]}}}
      }
    }
}
.
<<signature>>
```

Table 11 – Non-Base64-encoded Example of the Request Parameter Object

Then, the Buyer is redirected to the TPP. The TPP will now possess the Authorization Code and ID Token from the SP (Seller). Note at this point, there is no access token. The TPP will now introspect the ID Token and use it as a detached signature to check:

- The hash of the authorization code to prove it has not been tampered with during redirect (comparing the hash value against the c_hash attribute in ID Token)
- The hash of the state to prove it has not been tampered with during redirect (comparing the state hash value against the s_hash attribute in the ID Token)

Example: ID Token

```
{
  "alg": "RS256",
  "kid": "12345",
  "typ": "JWT"
}
.
{
   "iss": "https://api.acme.com",
   "iat": 1234569795,
   "sub": "urn:acme:quote:58923",
   "acr": "urn:mef:lso:security:oidc:acr:ca",
   "meflso_intent_id": "urn:acme:quote:58923",
   "aud": "s6BhdRkqt3",
   "nonce": "n-0S6_WzA2Mj",
   "exp": 1311281970,
   "s_hash": "76sa5dd",
   "c_hash": "asd097d"
  }
.
<<signature>>
```

Table 12 – ID Token Example

Once the state and code validations have been confirmed as successful, the TPP will proceed to obtain an access token from the SP/Seller using the authorization code it now possesses. The TPP will present its authorization code together with the private_key_jwt. The access token is required by the TPP to submit the Quote on behalf of the Buyer. The *quote* scope should already be associated with the authorization code generated in the previous step.

Request: Access Token Request using Authorization Code and private_key_jwt	**Response: Access Token**

POST /as/token.oauth2 HTTP/1.1
Host: https://authn.acme.com
Content-Type: application/x-www-form-urlencoded
Accept: application/json
grant_type=authorization_code
&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https://api.mytpp.com/cb
&client_assertion_type=
urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
&client_assertion=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRw
czovL2p3dC1pZHAuZXhhbXBsZS5jb20iLCJzdWIiOiJtYWlsdG86bWlrZUBleGFtcGxlLmN
vbSIsIm5iZiI6MTQ5OTE4MzYwMSwiZXhwIjoxNDk5MTg3MjAxLCJpYXQiOjE0OTkxODM2MD
EsImp0aSI6ImlkMTIzNDU2IiwidHlwIjoiaHR0cHM6Ly9leGFtcGxlLmNvbS9yZWdpc3Rlc
iJ9.SAxPMaJK_wYl_W2idTQASjiEZ4UoI7-P2SbmnHKr6LvP8ZJZX6JlnpK_xClJswAni1T
p1UnHJslc08JrexctaeEIBrqwHG18iBcWKjhHK2Tv5m4nbTsSi1MFQOlMUTRFq3_LQiHqV2
M8Hf1v9q9YaQqxDa4MK0asDUtE_zYMHz8kKDb-jj-Vh4mVDeM4_FPiffd2C5ckjkrZBNOK0
01Xktm7xTqX6fk56KTrejeA4x6D_1ygJcGfjZCv6Knki7Jl-6MfwUKb9ZoZ9LiwHf5lLXPuy
_QrOyM0pONWKj9K4Mj7I4GPGvzyVqpaZUgjcOaZY_rlu_p9tnSlE781dDLuw
 	
{
"alg": "RS256",
"kid": "12345",
"typ": "JWT"
}
.
{
"iss": "s6BhdRkqt3",
"sub": "s6BhdRkqt3",
"exp": 1499187201,
"iat": 1499183601,
"jti": "id123456",
"aud": "https://authn.acme.com/as/token.oauth2"
}
.
<<signature>>
	HTTP/1.1 200 OK
	Content-Type:
	application/json
	Cache-Control: no-store
	Pragma: no-cache
	
	{
	"access_token":
	"SlAV32hkKG",
	"token_type": "Bearer",
	"expires_in": 3600
	}

Table 13 – Non-Base64 JWT Client Assertion

Step 4 – Create Quote-Submission

The TPP has an access token which can be used to create a Quote-Submission (Step 4). The TPP must obtain the QuoteId (Intent ID) so that the Quote request is associated with the correct QuoteId. This is sourced from the QuoteId claim from the signed ID Token (default). The TPP will need to decode the ID Token JWT and locate the claim attribute associated with the QuoteId.

Once the previous step is completed, the TPP can now invoke the /quote-submissions API endpoint to commit the Quote using the access token and QuoteId in the payload of the request.

<table>
<tr><th>Request: quote-submissions</th><th>Response: quote-submissions</th></tr>
<tr><td><pre>POST /quote-submissions HTTP/1.1
Authorization: Bearer SlAV32hkKG
x-idempotency-key: FRESNO.1317.GFX.22
x-fapi mef-id: mef/2021/011
x-fapi-buyer-last-logged-time: 2020-06-13T11:36:09
x-fapi-buyer-ip-address: 104.25.212.99
x-fapi-interaction-id: 93bac548-d2de-4546-b106-880a5018460de9699
Content-Type: application/json
Accept: application/json

{
  "alg": "RS256",
  "kid": "12345",
  "typ": "JWT"
}
.
{
  "Data": {…}
}
.
<<signature>></pre></td><td><pre>HTTP/1.1 201 Created
x-fapi-interaction-id: 93bac548-
d2de-4546-b106-880a5018460d
Content-Type: application/json

{
  "alg": "RS256",
  "kid": "12347",
  "typ": "JWT"
}
.
{
  "Data": {…}
}
.
<<signature>></pre></td></tr>
</table>

Table 14 – Non-Base64 JWT Quote Submission

Step 5 – Get Quote-Submission Status

The TPP can query for the status of a Quote-Submission by invoking the /quote-submissions API endpoint using the known QuoteSubmissionId. This can use an existing access token with *quote* scope or the TPP/SP can obtain a fresh access token by replaying the client credentials grant request as per Step 2 – Setup Single Quote Initiation.

<table>
<tr><th>Request: quote-submissions/{QuoteSubmissionId}</th><th>Response: quote-submissions</th></tr>
<tr><td><pre>GET /quote-submissions/58923-001 HTTP/1.1
Authorization: Bearer SlAV32hkKG
x-fapi mef-id: mef/2021/011
x-fapi-buyer-last-logged-time: 2020-06-13T11:36:09
x-fapi-buyer-ip-address: 104.25.212.99
x-fapi-interaction-id: 93bac548-d2de-4546-b106-
880a5018460d
Accept: application/json</pre></td><td><pre>HTTP/1.1 200 OK
x-fapi-interaction-id: 93bac548-d2de-4546-
b106-880a5018460d
Content-Type: application/json

{
  "alg": "RS256",
  "kid": "12347",
  "typ": "JWT"
}
.
{
  "Data": {…}
}
.
<<signature>></pre></td></tr>
</table>

Table 15 – Non-Base64 JWT Quote Submission Status

Afterwards, a TPP can also optionally query for the status of a Quote resource by invoking /quote/{QuoteId} API endpoint. This can use an existing access token with *quote* scope or the TPP can obtain a fresh access token by replaying the client credentials grant request as per Step 2 – Setup Single Quote Initiation.

Edge Cases

This section provides further information on potential, common edge cases that may arise during the implementation of this standard. The document continues to use the Quote API example for specificity. However, the edge cases are general in nature, and not constrained to said API.

Buyer Consent Authorization Interrupt with Seller

API	Scenario	Workflow Step	Impact	Solution Options
Any	Due to an interruption, the Buyer does not complete the Authorization of the API request with the Entity when redirected by the TPP (for Quote API after creating a QuoteId)	Step 3: Authorize Consent	Resource Status, in the example Quote, remains as Pending	The TPP may choose to implement a separate follow up process which reminds the Buyer to complete their authorization consent steps with the Entity. This would imply re-using the assigned unique resource ID, e.g., the QuoteId, that has a status and re-issuing another Hybrid Grant Flow request to the Entity. The implementation of how the follow up process is initiated is in the competitive space for the TPPs/Entities to decide.

Table 16 – Buyer Consent Authorization Interruption

Example hybrid grant flow request/response

The HTTP request in Figure 10 depicts the fields and sample possible values defined in Table 2. The structure of id_token returned upon a successful request is shown in Figure11. Figure 12 shows the structure of the id_token when the subject is a user. In this flow, the Buyer present an Intent Id and the Seller returns an Id token after validation of the Intent Id and scope.

HTTP Request JWS/JWE

```
GET /authorize?
response_type=code%20id_token
&client_id=s6BhdRkqt3
&state=af0ifjsldkj&
&scope=openid
&nonce=n-0S6_WzA2Mj
&redirect_uri=https://api.mytpp.com/cb
&request=CJleHAiOjE0OTUxOTk1ODd.....JjVqsDuushgpwp0E.5leGFtcGxlIiwianRpIjoiM....J
leHAiOjE0.olnx_YKAm2J1rbpOP8wGhi1BDNHJjVqsDuushgpwp0E
```

Figure 10 – HTTP Request for Id Token

Note that the example shown in Figure 5 is without Base64 encoding. Also note that "essential" is an optional property. It indicates whether the Claim being requested is an Essential Claim. If the value is true, this indicates that the Claim is an Essential Claim. For instance, the Claim request:

```
"auth_time": {"essential": true}
```

can be used to specify that it is Essential to return an auth_time Claim Value. If the value is false, it indicates that it is a Voluntary Claim. The default is false.

By requesting Claims as Essential Claims, the RP indicates to the Seller that releasing these Claims will ensure a smooth authorization for the specific task requested by the Buyer.

Note that even if the Claims are not available because the Seller did not authorize their release or they are not present, the authorization server must not generate an error when Claims are not returned, whether they are Essential or Voluntary, unless otherwise specified in the description of the specific claim. See the OIDC Core Specification.

The request object in Figure 10 is expanded in Figure11.

```
{
    "alg": "RS256",
    "kid": "GxlIiwianVqsDuushgjE0OTUxOTk"
}
.
{
    "aud": "https://api.acme.com",
    "iss": "s6BhdRkqt3",
    "response_type": "code id_token",
    "client_id": "s6BhdRkqt3",
    "redirect_uri": "https://api.mytpp.com/cb",

    "state": "af0ifjsldkj",
    "nonce": "n-0S6_WzA2Mj",
    "max_age": 86400,
    "claims":
    {
      "userinfo":
      {
       "meflso_intent_id": {"value": "urn:acme-intent-58923", "essential": true}
      },
      "id_token":
      {
       "meflso_intent_id": {"value": "urn-acme-intent-58923", "essential": true},
       "acr": {"essential": true,
                 "values": ["urn:mef:lso:security:oidc:acr:sca",
                      "urn:mef:lso:security:oidc:acr:ca"]}}
      }
    }
}
.
<<signature>>
```

Figure 11 –Request JWS/JWE (expanded)

HTTP Response: id_token returned

Figure 12 shows the content of a JWS with the id_token being returned to the Buyer after authorization is successful, based on the request shown in Figure 10.

Note that Sub is being populated with an EphemeralId of the IntentId.

```
{
  "alg": "RS256",
  "kid": "12345",
  "typ": "JWT"
}
.
{
   "iss": "https://api.acme.com",
   "iat": 1234569795,
   "sub": "urn-acme-quote-58923",
   "acr": "urn:mef:lso:security:oidc:acr:ca",
   "meflso_intent_id": "urn-acme-intent-58923",
   "aud": "s6BhdRkqt3",
   "nonce": "n-0S6_WzA2Mj",
   "exp": 1311281970,
   "s_hash": "76sa5dd",
   "c_hash": "asd097d"
  }
.
{
<<Signature>>
}
```

Figure 12 – id_token Return

Id_token returned

Figure 13 shows Identity Claims and IntentId with sub being populated with an UserIdentifier. This reply is just an example of additional data that may be returned to the Buyer in an Id token.

```
{
  "alg": "RS256",
  "kid": "12345",
  "typ": "JWT"
}
.
{
   "iss": "https://api.acme.com",
   "iat": 1234569795,
   "sub": "ralph.bragg@raidiam.com",
   "acr": "urn:mef:lso:security:oidc:acr:sca",
   "address": "2 Thomas More Square",
   "phone": "+447890130559",
   "meflso_intent_id": "urn-acme-quote-58923",
```

```
    "aud": "s6BhdRkqt3",
    "nonce": "n-0S6_WzA2Mj",
    "exp": 1311281970,
    "s_hash": "76sa5dd",
    "c_hash": "asd097d"
   }
 .
 {
 <<Signature>>
 }
```


Figure 13 –id_token return with UserIdentifier

Implementers should note that ID Token Claims details should follow the JWT Best Current Practices [18] section 3.1.