

FIDO Voucher Transfer Protocol

Review Draft, October 01, 2026

FIDO
ALLIANCE
REVIEW DRAFT

This version:

<https://fidoalliance.org/specs/fidoiot/voucher-transfer-v1.0-rd-20261001.html>

Issue Tracking:

[GitHub](#)

Editors:

[Brad Goodman](#) (Dell Technologies)

[Gerardo Diaz Cuellar](#) (Microsoft)

[Geof Cooper](#) (FIDO Alliance)

Contributors:

[Sven Uthe](#) (DEVITY)

Version:

1.0

Copyright © 2026 [FIDO Alliance](#). All Rights Reserved.

Abstract

Standard REST/JSON protocol for secure transfer of FIDO ownership vouchers between manufacturing systems and owner systems via HTTPS. All APIs use standard HTTP methods, JSON payloads, and bearer token authentication — designed to integrate with existing business systems (ERP, order management, CI/CD) using standard tooling. FIDO-specific cryptographic authentication is available as an optional, self-contained shim; it is not required for API integration.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the [FIDO Alliance specifications index](#) at <https://fidoalliance.org/specifications/>.

This document was published by the [FIDO Alliance](#) as a Review Draft Specification. This document is intended to become a FIDO Alliance Proposed Standard. If you wish to make comments regarding this document, please [Contact Us](#). All comments are welcome.

This is a Review Draft Specification and is not intended to be a basis for any implementations as the Specification may change. Permission is hereby granted to use the Specification solely for the purpose of reviewing the Specification. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this Specification for other uses must contact the FIDO Alliance to determine whether an appropriate license for such use is available.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1	Introduction
2	Terminology
3	Voucher Management Overview
3.1	The Problem: Transferring Vouchers
3.2	The Solution: Enroll Once, Transfer Securely
3.3	Enrollment: Exchanging Credentials
3.3.1	How Enrollment Works in Practice
3.3.2	Supply Chains and Multi-Hop Transfer
3.4	Transfer: Push, Pull, or Out-of-Band
3.4.1	Push Transfer (Supplier-Initiated)
3.4.2	Pull Transfer (End-User-Initiated)
3.4.3	Out-of-Band Transfer
3.5	Protocol-Native Authentication
3.6	Reading Guide
4	Use Cases and Requirements
4.1	Real-World Deployment Scenarios (Examples)
4.1.1	Cloud Service Push
4.1.2	On-Premises Pull
4.1.3	On-Premises Push
4.1.4	Batch Transfer
4.1.5	Reconciliation and Recovery
4.1.6	Pagination
4.2	Security Requirements
4.2.1	Authentication is Mandatory
4.2.2	DDoS Considerations
4.2.3	Defense-in-Depth
5	Transfer Models
5.1	Push Model (Issuer-Initiated)
5.1.1	Flow
5.1.2	Characteristics
5.2	Pull Model (Recipient-Initiated)
5.2.1	Flow
5.2.2	Characteristics
5.3	Hybrid Approaches
6	Voucher File Format
6.1	File Extension
6.2	MIME Type
6.3	File Contents
6.3.1	PEM Encoding
6.4	Size Constraints
7	Voucher Archive Format
7.1	File Extension
7.2	Format
7.3	MIME Type
7.4	Archive Structure
7.5	Size Constraints
8	Voucher Identification
8.1	Voucher Digest
8.1.1	Rationale

- 8.1.2 Security Considerations
- 8.1.3 String Representation
- 8.2 Voucher Fingerprint
 - 8.2.1 Computation
 - 8.2.2 Format
 - 8.2.3 Use Cases
 - 8.2.4 Security Note
- 8.3 Comparison with GUID

9 Service Root URLs

- 9.1 Overview
- 9.2 Service Types
 - 9.2.1 Push Service Root
 - 9.2.2 Pull Service Root
- 9.3 Endpoint Notation
- 9.4 Versioning

10 Push API Specification

- 10.1 Submit Voucher
 - 10.1.1 POST {root}
 - 10.1.1.1 Request
 - 10.1.1.2 Response
- 10.2 Query Voucher Status
 - 10.2.1 GET {root}/status/{identifier}
 - 10.2.1.1 Request
 - 10.2.1.2 Response
 - 10.2.1.3 Status Values
 - 10.2.1.4 Response Fields

11 Pull API Specification

- 11.1 Overview
- 11.2 API Endpoints
 - 11.2.1 List Vouchers
 - 11.2.1.1 GET {root}
 - 11.2.1.2 Request
 - 11.2.1.3 Response
 - 11.2.1.4 Voucher Fields
 - 11.2.1.5 Pagination
 - 11.2.2 Download Voucher
 - 11.2.2.1 GET {root}/{voucher_id}/download
 - 11.2.2.2 Request
 - 11.2.2.3 Response
 - 11.2.3 Subscription and Notification (OPTIONAL)
 - 11.2.3.1 Long-Polling
 - 11.2.3.1.1 GET {root}/subscribe
 - 11.2.3.2 Server-Sent Events
 - 11.2.3.2.1 GET {root}/stream
- 11.3 Assignment (Pull Without Sign-Over)

12 Voucher Assignment

- 12.1 Overview
- 12.2 When Assignment Applies
 - 12.2.1 Build-to-Stock (Assignment)
 - 12.2.2 Build-to-Order (Not Assignment)
- 12.3 Cryptographic Constraint
- 12.4 Authentication
- 12.5 Assign Endpoint
 - 12.5.1 POST {root}/assign
 - 12.5.1.1 Request

12.5.1.2	Server Processing
12.5.1.3	Response
12.6	Credential Relay
12.7	Assignment Immutability
12.7.1	Rationale
12.7.2	Error Recovery (Non-Normative)
13	Authentication Framework
13.1	The Three Layers
13.1.1	Layer 1: Voucher Operations
13.1.2	Layer 2: Request Authentication
13.1.3	Layer 3: Token Establishment
13.2	Authentication Paths
13.2.1	Owner with FDO Key
13.2.2	Owner with Enterprise Identity
13.2.3	Custodian with mTLS
13.2.4	Custodian with Portal Token
13.3	Design Rationale: Architectural Separation
13.4	Owner vs Custodian Authentication
14	Request Authentication
14.1	Bearer Tokens
14.1.1	Token Format
14.1.2	Token Scoping
14.1.3	Token Lifetime
14.1.4	Token Refresh
14.2	DPoP Tokens (Sender-Constrained)
14.2.1	Why DPoP?
14.2.2	Request Format
14.2.3	DPoP Proof Structure (per RFC9449)
14.2.4	Token Binding
14.2.5	Server Validation
14.2.6	Nonce Refresh
14.2.7	Obtaining DPoP Tokens
14.3	mTLS Authentication
15	Token Establishment
15.1	OAuth/OIDC Integration (Non-Normative)
15.1.1	Token Exchange Pattern
15.1.2	Server Configuration
15.1.3	DPoP with OAuth
15.2	Portal/Out-of-Band Tokens (Non-Normative)
15.2.1	Partner Portal Pattern
15.2.2	API Key Pattern
15.2.3	Security Considerations
16	FDOKeyAuth Protocol
16.1	Overview
16.1.1	The Authentication Problem
16.1.2	TO2-Inspired Challenge-Response
16.1.3	Protocol Summary
16.2	Protocol Messages
16.2.1	Protocol Messages
16.2.2	Common Data Types
16.2.2.1	Nonces
16.2.2.2	Hash Continuity
16.2.2.3	Public Keys and Certificate Chains
16.2.3	Message 1: FDOKeyAuth.Hello

16.2.3.1	POST {root}/auth/hello
16.2.3.2	Request Body (application/cbor)
16.2.3.3	Server Processing
16.2.3.4	Response: FDOKeyAuth.Challenge (application/cbor)
16.2.4	Message 2: FDOKeyAuth.Prove
16.2.4.1	POST {root}/auth/prove
16.2.4.2	Request Body (application/cbor)
16.2.4.3	Server Verification
16.2.4.4	Response: Authentication Result (application/cbor)
16.3	Token Lifecycle
16.3.1	Token Lifetime
16.3.2	Token Expiry and Re-Authentication
16.3.3	Token Scoping
16.3.4	API Gateway Compatibility
16.3.5	Alternative Token Sources (Non-Normative)
16.3.5.1	DPoP with Alternative Token Sources
16.4	DPoP: Sender-Constrained Tokens
16.4.1	Overview
16.4.2	DPoP with FDOKeyAuth
16.4.3	DPoP Key Registration Endpoint
16.4.3.1	POST {root}/auth/dpop-keys
16.4.3.2	GET {root}/auth/dpop-keys
16.4.3.3	DELETE {root}/auth/dpop-keys
16.4.4	Using DPoP Tokens in API Requests
16.4.5	Server Validation of DPoP Requests
16.4.6	DPoP Nonce Refresh
16.4.7	Server DPoP Policy
16.4.8	HTTP Header Discovery
16.4.9	DID Document Discovery
16.4.10	COSE_Key to JWK Conversion
16.5	mTLS Authentication
16.5.1	Overview
16.5.2	When to Use mTLS
16.5.3	Certificate Enrollment
16.5.4	Server Configuration
16.5.5	Request Processing
16.5.6	Custodian Authentication via mTLS
16.5.7	Combining mTLS with Token Authentication
16.5.8	Relationship to DID Enrollment
16.6	Wire Format: CBOR Encoding
16.7	Delegation Support
16.7.1	Delegate Certificate Requirements
16.7.2	Extended Key Usage Permissions
16.7.3	Delegate Authentication Flow
16.8	Continuation Token Security
16.8.1	Overview
16.8.2	Threat: Continuation Rollback
16.8.3	Mitigation Requirements
16.8.4	Recommended Token Structure
16.8.5	Informative Note on Continuation Rollback Risk
16.9	Implementation Notes
16.9.1	Signing Oracle Protection
16.9.2	Session State Management
16.9.3	Server Signature Verification by Caller

17 Security Framework

17.1	How Authentication Works
17.1.1	Security Properties of FDOKeyAuth

17.1.2	Summary
17.2	Threat Model
17.3	DDoS Considerations for Push Endpoints
18	Voucher Sequestering
18.1	Overview
18.2	Process
18.3	Risk-Based Sequestering
19	DID Integration
19.1	Trust Foundation: Mutual DID Exchange
19.2	DID-Based Security Model
19.2.1	Case 1: Provider Pushes Only to a Provably-Known Owner
19.2.2	Case 2: Recipient Accepts Pushes Only from Known-Trusted Providers
19.2.3	Case 3: Recipient Pulls from a Known Provider
19.2.4	Case 4: Provider Allows Pulls Only from Authorized Owners
19.2.5	Summary
19.3	Endpoint Discovery
19.3.1	Push Discovery
19.3.2	Pull Discovery
19.4	FDO JSON-LD Context
19.5	DID Document Service Types
19.5.1	FDOVoucherRecipient
19.5.2	FDOVoucherHolder
19.5.3	Optional TLS Certificate Authority
19.5.4	Example DID Document
19.6	Defense-in-Depth Layers (Informative)
20	Key Formats and Enrollment Considerations
20.1	Key Independence After Enrollment
20.2	Enrollment Mechanisms
20.3	Key Format Comparison
20.3.1	did:web
20.3.2	did:key
20.3.3	X.509 Certificates
20.3.4	Raw PEM Public Keys
20.3.5	Summary Comparison
21	Error Handling and Retry Logic
21.1	Retry Strategy
21.2	Error Categories
21.2.1	Transient Errors (Retry)
21.2.2	Permanent Errors (Do Not Retry)
21.2.3	Configuration Errors (Manual Intervention)
22	Security Considerations
22.1	Transport Security
22.2	Error Response Security
22.3	Rate Limiting
22.4	Security Monitoring
22.5	FDOKeyAuth Security Considerations
22.5.1	Signing Oracle Attacks
22.5.2	Voucher Existence Disclosure
22.5.3	Device GUID Entropy
22.5.4	Session Exhaustion
22.5.5	Delegate Certificate Revocation
22.5.6	Relationship to Transport Security
22.5.7	Bearer Token Security Analysis

- 22.5.8 DPOP Security Analysis
 - 22.5.8.1 Threat Model
 - 22.5.8.2 Security Properties
 - 22.5.8.3 Limitations
 - 22.5.8.4 Recommendations
- 22.5.9 Application-Layer Protections: Comparison with FDO TO2
 - 22.5.9.1 Why TO2 Needs Application-Layer Protections
 - 22.5.9.2 Why Voucher Management Does Not
 - 22.5.9.3 TLS Trust Models

23 Implementation Guidelines

- 23.1 Versioning
- 23.2 Protocol Extensibility
 - 23.2.1 General Principle
 - 23.2.2 JSON Extensibility (REST APIs)
 - 23.2.3 CBOR Extensibility (FDOKeyAuth Messages)
 - 23.2.3.1 CBOR Arrays
 - 23.2.3.2 CBOR Maps
 - 23.2.3.3 Signed Payloads
 - 23.2.3.4 Version Negotiation
 - 23.2.4 CBOR Deterministic Encoding
- 23.3 Content Negotiation
- 23.4 Idempotency
- 23.5 Pagination Best Practices

24 Future Enhancements

25 Conclusion

Appendix A: Optional Security Layers (Informative)

Token-Based Authorization

- Overview
- Token Format
- Token Types
- Validation

Mutual TLS (mTLS) Authentication

- Overview
- Considerations
- Enrollment Process

Business Logic Validation

- Overview
- Validation Types

Token vs mTLS Comparison

Purchase Process Integration

- Token Generation During Purchase
- Purchase-Tied Token Format

Appendix B: Voucher Recovery When the Ownership Voucher Is Not Forwarded (Informative)

The Problem

Relationship to Voucher Assignment

Design Principles

- Minimize Manufacturer Burden
- Don't Prescribe the Mechanism
- Proof of Possession, Not Proof of Ownership
- At-Most-Once

The Claim Code

- Requirements
- Recommended Approach: Random Claim Code

- What About OV-Derived Claim Codes?
- Conveying the Claim Code to the Device Recipient
 - QR Code on a Sticker (Recommended)
 - Static QR Code with Manual Identifier Entry
 - Printed Card or Documentation
 - On-Screen Display at Boot (Not Recommended)
- The Claim Process
 - Minimum Viable Claim Flow
 - Optional Additional Verification
- Security Considerations for Voucher Recovery
 - Claim Code Entropy
 - Rate Limiting
 - Claim Notification
 - Revocation
- Rationale: Why Not the Earlier Proposal?
 - Prescribed Proof-of-Possession Computation
 - Transmission via Email or Fax
 - Mandatory On-Screen Display
 - Single Prescribed Flow

References

Non-Normative References

1. Introduction§

The FIDO Device Onboard (FDO) specification [\[FDO-Specification\]](#) defines mechanisms for secure device provisioning and ownership transfer. A critical step in this process is the transfer of ownership vouchers from manufacturing systems (where devices are provisioned) to owner systems (where devices will be onboarded). Today, this transfer is handled through proprietary, implementation-specific mechanisms.

This App Note defines a proposed Standard for an HTTP-based protocol for voucher transfer that addresses the following challenges (see [§ 3 Voucher Management Overview](#) for a conceptual walkthrough before diving into protocol details):

- **Integration Complexity:** Each manufacturer-owner pair currently requires custom integration work
- **Security Inconsistencies:** Varying security models across implementations create gaps and confusion
- **Scalability Issues:** No standard approach exists for high-volume or complex deployments
- **Operational Overhead:** Lack of common monitoring, error handling, and debugging approaches

The central design principle of this specification is **business logic integration**: every API defined here is a standard REST/JSON endpoint that works with standard tools — curl, Postman, Python requests, API gateways, CI/CD pipelines, and enterprise integration platforms. Organizations should not need to reinvent their integration practices or adopt specialized cryptographic libraries to transfer vouchers. FDO-specific cryptographic authentication (FDOKeyAuth) is defined as an optional, self-contained shim that produces a standard bearer token; once that token exists, FDOKeyAuth's job is done and the rest of the interaction is indistinguishable from any other bearer-authenticated REST API. Organizations that already have an identity provider (OAuth2, OIDC) or partner portal can skip FDOKeyAuth entirely and use their existing token infrastructure. The result is a protocol that meets FDO's security requirements while fitting naturally into the IT systems and operational practices that organizations already have.

The protocol defined here supports both **push** (manufacturer-initiated) and **pull** (owner-initiated) transfer models, multiple security mechanisms, pagination for large datasets, and integration with Decentralized Identifiers (DIDs) [\[DID-Core\]](#) for endpoint discovery.

2. Terminology§

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [\[RFC2119\]](#).

Voucher: Throughout this document, "voucher" refers to an FDO Ownership Voucher as defined in [\[FDO-Specification\]](#), unless otherwise qualified.

Voucher Issuer: A manufacturing system that generates FDO ownership vouchers during device provisioning. Also referred to as the manufacturer.

Voucher Recipient: An Owner system that receives and processes ownership vouchers for device onboarding. The Voucher Recipient holds the Owner Key to which vouchers are signed over. Note that the Voucher Recipient (an Owner in the cryptographic sense) may or may not be the same entity as the device's current business owner (see the distinction between Owner and owner under *Owner Key*).

Voucher Transfer: The act of securely delivering an ownership voucher from a Voucher Issuer to a Voucher Recipient.

Push Transfer: A transfer initiated by the Voucher Issuer, who sends the voucher to the Voucher Recipient's endpoint.

Pull Transfer: A transfer initiated by the Voucher Recipient, who retrieves vouchers from the Voucher Issuer's endpoint.

Continuation Token: An opaque, server-generated token used for paginating through large sets of vouchers.

Voucher Sequestering: A process where received vouchers are held in quarantine pending additional validation before being released for use.

Holder: A Voucher Issuer or intermediary management service that holds vouchers on behalf of a Voucher Recipient. In the pull model, the Holder is the server from which the Recipient retrieves vouchers.

Recipient: A Voucher Recipient that initiates a pull transfer to retrieve vouchers from a Holder. The Recipient proves ownership of the key to which vouchers have been signed over.

Owner Key: The public key appearing as the final key in the Ownership Voucher's entry chain. Vouchers are "signed over to" this key. Possession of the corresponding private key proves the right to claim those vouchers.

NOTE: This specification uses **Owner** (capitalized) to refer specifically to the holder of an Owner Key — an entity that possesses the private key corresponding to the public key at the tip of a voucher's ownership chain. This is a cryptographic role: the Owner can prove key possession via FDOKeyAuth and can further extend (sign over) the voucher chain to another party. In contrast, "owner" (lowercase) refers to the entity that owns a device in the ordinary business sense — the organization that purchased the device, is responsible for it, and will ultimately onboard it. In many deployments these are the same entity, but they need not be. A reseller may own thousands of devices without holding any FDO Owner Key. The distinction matters because business ownership (which entity controls what happens to a device) and cryptographic ownership (which key is at the voucher chain tip) can diverge in multi-hop supply chains.

Custodian: An entity in the supply chain that has business authority over devices but does not hold an FDO Owner Key. A Custodian is the current business owner of a device — they purchased it, are responsible for it, and have the right to direct where it goes next — but they do not appear in the voucher's cryptographic ownership chain. Custodians are known to the Key Holder (the entity at the voucher chain tip) through out-of-band business relationships: purchase orders, contracts, portal accounts, mTLS certificates, or other means.

Custodians can perform voucher management operations (push, pull, assign, query status) by authenticating through non-FDO mechanisms — mTLS client certificates, JWT tokens from an identity provider, API keys, or portal credentials. They cannot be sign-over targets (because they have no Owner Key to sign to), but they can direct the Key Holder to sign vouchers over to a designated party (see *Voucher Assignment*).

The Custodian role is common in supply chains: a reseller who buys devices from a manufacturer and sells them to end customers is a Custodian. They need to manage vouchers (retrieve them from the manufacturer, assign them to customers) but they never take cryptographic ownership. The APIs in this specification are designed to support Custodians as first-class participants, not just Owners.

Voucher Assignment: The act of a Custodian directing the Key Holder to extend a voucher's ownership chain to a designated party's public key. The Custodian authenticates to the Key Holder through whatever mechanism their business relationship provides (mTLS certificate, API token, web portal, manual process, etc.). The Key Holder performs the cryptographic sign-over on the Custodian's behalf. The Custodian never appears in the voucher chain. Assignment is an at-most-once operation per Custodian per voucher: once a Custodian has directed an assignment, the same Custodian **MUST NOT** direct a different assignment for the same voucher (this is termed *reassignment* and is prohibited). Note: If a party wishes to retain the ability to further extend a voucher, they should take cryptographic ownership (hold an Owner Key) rather than acting as a Custodian.

Delegate Certificate: An X.509 certificate (or certificate chain) signed by the Owner Key that authorizes a Delegate to perform specific operations on behalf of the Owner. The voucher-transfer delegation profile is defined in this specification (see [§ 16.7 Delegation Support](#)).

Delegate Key: A private key held by a Delegate (see *Delegate Certificate*) that has been authorized by the Owner to act on the Owner's behalf. The Delegate Key is identified by a Delegate Certificate signed by the Owner Key.

Pull Authentication: A cryptographic challenge-response protocol by which a Recipient proves possession of an Owner Key (or authorized Delegate Key) to a Holder, in order to retrieve vouchers signed over to that Owner Key.

Challenge Nonce: A random value issued by one party to another, to be included in a signed response, proving freshness and preventing replay attacks. Follows the nonce patterns defined in [\[FDO-Specification\]](#).

Supplier: Any party from whom a Voucher Recipient acquires devices (and therefore vouchers). A Supplier may be the device manufacturer who ran Device Initialize, or a reseller who received vouchers from an upstream source and signs them over to the next party. In all cases, the Supplier holds vouchers that must be signed over and delivered to the Recipient.

Enrollment: The one-time, out-of-band exchange of identity information (typically DIDs) between a Supplier and a Recipient, establishing the trust relationship and cryptographic credentials required for subsequent voucher transfers.

Bearer Token: An opaque access token carried in the `Authorization: Bearer <token>` HTTP header. Possession of the token alone grants access; no additional proof is required. See [§ 16.3 Token Lifecycle](#).

DPoP (Demonstrating Proof of Possession): A mechanism defined in [\[RFC9449\]](#) that binds access tokens to a specific cryptographic key. Each API request using a DPoP token must include a signed proof demonstrating possession of the bound key. This prevents token theft attacks where an attacker who obtains a token cannot use it without also possessing the private key.

DPoP Token: An access token that contains a `cnf` (confirmation) claim binding it to a specific public key via a JWK Thumbprint [\[RFC7638\]](#). DPoP tokens are carried in the `Authorization: DPoP <token>` header and require an accompanying DPoP header containing a signed proof.

DPoP Proof: A signed JWT included in the DPoP HTTP header that proves the caller possesses the private key bound to a DPoP token. The proof includes the HTTP method, URI, and a unique identifier to prevent replay attacks. See [\[RFC9449\]](#).

JWK Thumbprint: A hash of a JSON Web Key (JWK) computed according to [\[RFC7638\]](#). Used to bind DPoP tokens to specific keys via the `jkt` (JWK Thumbprint) confirmation method.

Sender-Constrained Token: A token that can only be used by a specific sender who can prove possession of a bound key. DPoP tokens are sender-constrained tokens. Contrast with bearer tokens, which grant access to

anyone who possesses them.

mTLS (Mutual TLS): A TLS configuration where both the server and client present certificates during the TLS handshake. When mTLS is used, the server can identify the client based on the client certificate without requiring an Authorization header. mTLS provides both encryption and authentication at the transport layer. See [§ 16.5 mTLS Authentication](#).

Client Certificate: An X.509 certificate presented by the client during an mTLS handshake. The server uses the client certificate to identify the caller. Client certificates are enrolled out-of-band and associated with a caller identity (Owner or Custodian) in the server's configuration.

Voucher Operations: The core APIs that perform voucher management work: Push (submit vouchers to a recipient), Pull (retrieve vouchers from a holder), and Assign (associate vouchers with a new owner key). All Voucher Operations require authentication. The authentication method is independent of the operation itself — the same Push API works whether the caller authenticates via Bearer token, DPoP token, or mTLS.

Request Authentication: The mechanism by which a caller proves identity on each API request to a Voucher Operation endpoint. This specification supports three methods: Bearer tokens (Authorization: Bearer <token>), DPoP tokens (Authorization: DPoP <token> with accompanying proof), and mTLS (client certificate in TLS handshake). Request Authentication is distinct from Token Establishment — it is about *using* credentials, not *obtaining* them.

Token Establishment: The process by which a caller obtains a token for use in Request Authentication. Methods include FDOKeyAuth (the FDO-specific challenge-response protocol defined in this specification), OAuth/OIDC (external identity providers), or out-of-band provisioning (portal tokens, API keys). Token Establishment produces Bearer or DPoP tokens that are then used for Request Authentication. Note that mTLS bypasses Token Establishment entirely — the client certificate *is* the credential.

FDOKeyAuth: The FDO-specific token establishment protocol defined in this specification. FDOKeyAuth is a cryptographic challenge-response handshake inspired by the TO2 protocol in [\[FDO-Specification\]](#). The caller proves possession of an enrolled key (Owner Key or Delegate Key), and the server issues a Bearer or DPoP token. FDOKeyAuth uses CBOR encoding and COSE signatures. See [§ 16 FDOKeyAuth Protocol](#).

3. Voucher Management Overview

This section provides a high-level introduction to voucher management: what problem it solves, what setup is required before vouchers can flow, and how the rest of this specification fits together. Readers new to this specification should start here.

In brief: Exchange credentials once, optionally run a small FDOKeyAuth shim to get a bearer token (or use your own OAuth2/OIDC), then push, pull, and assign vouchers over plain REST/JSON — no specialized crypto in the business logic path.

3.1. The Problem: Transferring Vouchers

When an FDO-capable device is manufactured, the factory creates an **ownership voucher** — a cryptographic document that proves who is authorized to onboard the device. The voucher starts out "owned" by the manufacturer's key. Before the device's eventual owner can onboard it, the voucher must be **signed over** to that owner's key (by a key owned-by, or accessible to, or delegated from, that manufacturer) and **delivered** to the owner's management service.

This sounds simple, but consider what the manufacturer actually needs:

1. **The owner's public key** — The manufacturer must sign the voucher over to a specific cryptographic key. Without the owner's key, the manufacturer cannot produce a voucher the owner can use. Once signed over, only the holder of the target key's private key can further extend the voucher chain or onboard the device.

2. **A way to deliver the voucher** — Once signed, the voucher must reach the owner's management service. The manufacturer needs either a URL to push the voucher to, or needs to make the voucher available for the owner to pull.
3. **Proof of manufacturer identity** — When the owner's management service receives a voucher, it needs to verify that the voucher actually came from a trusted manufacturer — not from a malicious actor injecting fraudulent vouchers.

These three requirements — **key exchange**, **endpoint discovery**, and **mutual authentication** — are what this specification addresses.

3.2. The Solution: Enroll Once, Transfer Securely§

This specification solves all three problems with a single mechanism: **mutual DID exchange**. Each party gives the other a short Decentralized Identifier (DID) string that encodes a public key and service URLs. From that one-time exchange, the protocol derives everything it needs — keys for signing and verification, endpoints for push or pull delivery, and cryptographic authentication that requires no additional credentials.

The diagram below shows the complete lifecycle: enrollment (exchanging DIDs), followed by voucher transfer via push or pull.

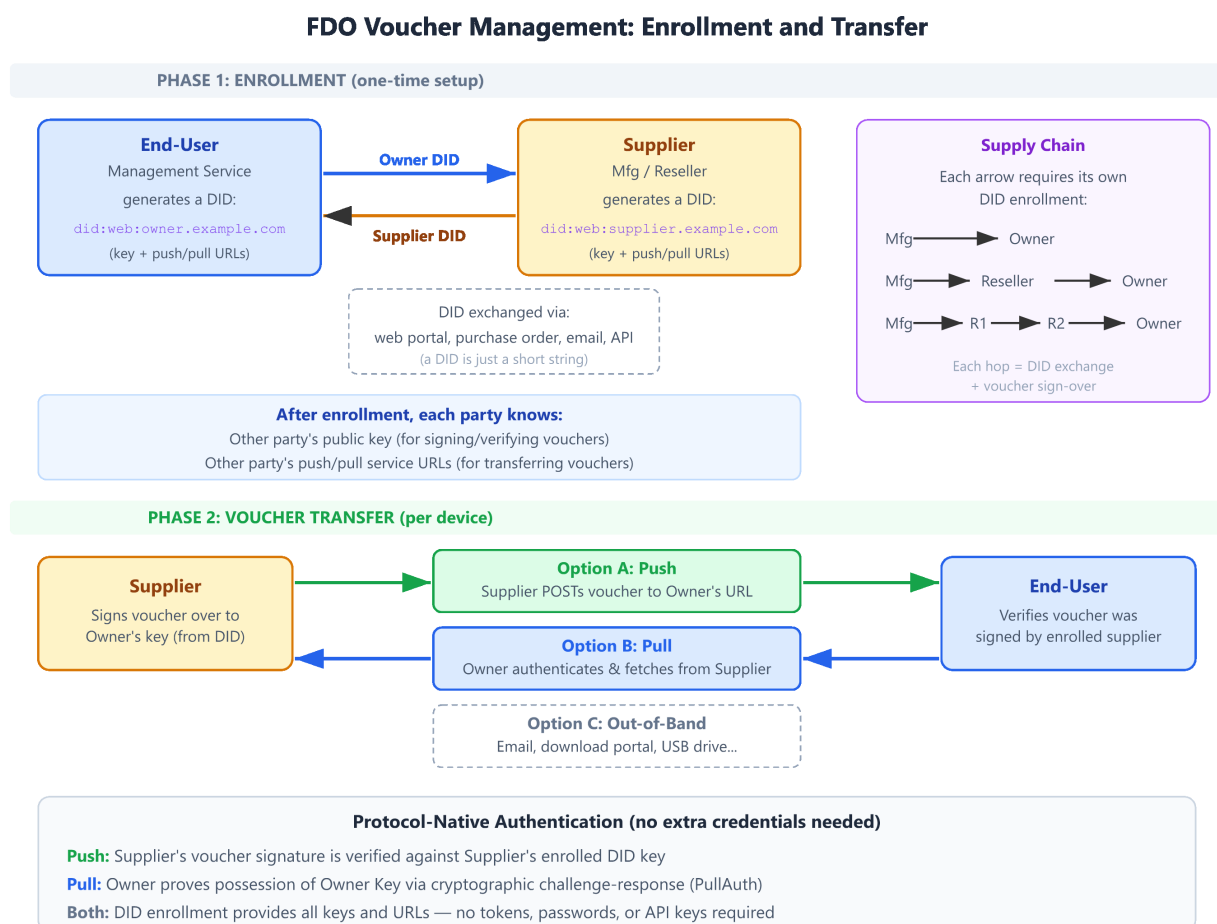


Figure 1 Credentials Enrollment and Voucher Exchange

The rest of this section walks through each phase in detail.

3.3. Enrollment: Exchanging Credentials§

Before any vouchers can flow, the two parties must establish a trust relationship. This specification calls this

enrollment — the one-time, mutual exchange of identity credentials (a public key and, optionally, service endpoint URLs) between a Supplier and a Recipient.

The most convenient way to perform this exchange is through **Decentralized Identifiers (DIDs)**. A DID is a short string — something like `did:web:acme-mfg.example.com` — that encodes everything each party needs to know about the other:

- **A public key** (`verificationMethod`) — for signing and verifying vouchers
- **Service endpoints** (`service`) — URLs for push and/or pull voucher transfer
- **An implicit trust declaration** — enrolling a DID is an explicit statement: "I trust this party for voucher operations"

The examples in this section use DIDs because they illustrate the enrollment concept most concisely. However, DIDs are not the only option — enrollment can be performed with any key format that conveys the necessary public key material. See [§ 20 Key Formats and Enrollment Considerations](#) for a detailed comparison of `did:web`, `did:key`, X.509 certificates, raw PEM keys, and their respective tradeoffs.

3.3.1. How Enrollment Works in Practice

Consider a typical scenario: an enterprise (the **end-user**) purchases FDO-capable devices from a manufacturer (the **supplier**).

1. **The end-user sets up a management service** — whether cloud-hosted, on-premises, or managed by a third party. During setup, the service generates a DID. This DID is unique to the end-user's service and encodes the owner key plus any push/pull URLs the service offers.
2. **The end-user places an order** — When ordering devices from the supplier (via web portal, purchase order, or any other mechanism), the end-user provides their DID as part of the purchase or onboarding process — for example, as a field on the order form or partner portal. A DID is just a short string, so it can be pasted into a web form, written on a purchase order, or sent in an email.
3. **The supplier provides their DID in return** — The supplier gives the end-user a DID that identifies the supplier's signing key. The end-user enters this DID into their management service, telling it: "accept vouchers signed by this supplier."

That is the entire enrollment process. After this one-time exchange:

- The **supplier** can resolve the end-user's DID to obtain the owner key (to sign vouchers over to) and the push URL (to deliver vouchers to)
- The **end-user's service** can verify incoming vouchers against the supplier's enrolled key, and can resolve the supplier's DID to discover a pull URL if pull-based transfer is preferred
- **No additional credentials** — no tokens, passwords, API keys, or certificates — need to be provisioned or exchanged

NOTE: The example above uses `did:web` DIDs, but the underlying requirement is simply the exchange of a public key and (for push) a service URL. If the end-user provides an X.509 certificate or a `did:key` string instead, the supplier obtains the same key material — the only difference is that service URLs must be conveyed separately. Once enrollment is complete, the format in which the key was originally provided is no longer relevant to protocol operations. See [§ 20 Key Formats and Enrollment Considerations](#) for details.

3.3.2. Supply Chains and Multi-Hop Transfer

The examples above describe a direct manufacturer-to-end-user relationship, but real supply chains often involve intermediaries:

Manufacturer → Reseller → End-User
Manufacturer → Distributor → Reseller → End-User

The enrollment and transfer mechanisms in this specification work identically at **every hop** in the supply chain. Each arrow (→) represents an independent trust relationship that requires:

- **Its own DID enrollment** — the upstream party and downstream party exchange DIDs
- **Its own voucher sign-over** — the upstream party signs the voucher over to the downstream party's key
- **Its own transfer** — the voucher is pushed or pulled between those two parties

A reseller, for example, enrolls with the manufacturer (to receive vouchers) and separately enrolls with the end-user (to forward vouchers). The protocol does not need to know or care about the overall chain — each hop is a self-contained enrollment + transfer between two parties. The term **Supplier** in this specification refers to whichever party is upstream in any given hop.

3.4. Transfer: Push, Pull, or Out-of-Band§

Once enrollment is complete, vouchers can flow. This specification defines two standard transfer protocols, but does not mandate either:

3.4.1. Push Transfer (Supplier-Initiated)§

The supplier sends vouchers directly to the end-user's management service via HTTP POST. The supplier resolves the end-user's DID to discover the push URL, signs the voucher over to the end-user's key, and delivers it. The management service verifies the voucher was signed by an enrolled supplier before accepting it.

Push is ideal when the end-user's service is reachable over the network — for example, a cloud-hosted management platform.

3.4.2. Pull Transfer (End-User-Initiated)§

The end-user's management service reaches out to the supplier's service to fetch vouchers. The end-user resolves the supplier's DID to discover the pull URL, then authenticates using a cryptographic challenge-response protocol (FDOKeyAuth, see [§ 16 FDOKeyAuth Protocol](#)) that proves possession of the owner key the vouchers were signed over to. The supplier releases only vouchers belonging to that authenticated key.

Pull is ideal when the end-user's service cannot accept inbound connections — for example, an on-premises deployment behind a corporate firewall that blocks inbound traffic but permits outbound connections.

3.4.3. Out-of-Band Transfer§

This specification does **not** require the use of its push or pull protocols for voucher delivery. Suppliers and end-users MAY transfer vouchers through any mechanism that works for their deployment:

- **Email** — Supplier emails .fdoov voucher files as attachments. Note: Email transfer is viable because the voucher is cryptographically signed to the recipient's Owner Key — only the key holder can use it, regardless of who handles the file in transit.
- **Download portal** — Supplier hosts a web portal where end-users log in and download vouchers
- **USB drive or physical media** — For truly air-gapped or "dark" environments with no network connectivity
- **Custom API** — Integration with existing procurement or supply chain systems

These out-of-band methods are particularly important for **disconnected management services** that cannot participate in push or pull protocols at all.

Regardless of the conveyance mechanism, services that provide vouchers for out-of-band retrieval (e.g., download portals, email systems, file shares) and services that receive vouchers via out-of-band import (e.g., management plane upload interfaces) **MUST** support the following normalized file formats:

- **Individual voucher files** — A single `.fdoov` PEM-encoded ownership voucher as defined in [§ 6 Voucher File Format](#).
- **Voucher archives** — An `.fdozip` file containing one or more individual `.fdoov` files at the top level of the archive. The `.fdozip` format is a standard ZIP archive with an FDO-specific extension; it **MAY** be renamed to `.zip` for extraction with standard tools. Each entry in the archive **MUST** be a valid `.fdoov` file. Implementations **SHOULD** ignore non-`.fdoov` entries in the archive (e.g., `__MACOSX/` metadata or README files) rather than rejecting the entire archive. See [§ 7 Voucher Archive Format](#) for details.

These two formats — single `.fdoov` and `.fdozip` archive — are the **baseline interchange formats** for all out-of-band voucher transfer. Implementations **MAY** support additional formats, but **MUST** support at least these two.

However, even when vouchers are transferred out-of-band, **DID enrollment is still required for keying**. The supplier still needs the end-user's public key to sign vouchers over to, and the end-user still needs the supplier's key to verify voucher authenticity. The DID exchange provides these keys regardless of how the voucher file itself is delivered.

3.5. Protocol-Native Authentication§

A key design principle of this specification is that **enrollment provides all the credentials needed for secure transfer**. No additional tokens, passwords, or API keys are required:

- **Push authentication:** When a voucher arrives at an end-user's service, the service verifies the voucher's cryptographic signature against the supplier's enrolled DID key. The voucher is a self-authenticating artifact — it carries proof of who created it.
- **Pull authentication:** When an end-user's service connects to a supplier's pull endpoint, it proves possession of the owner key via the same FDOKeyAuth handshake. The owner key is the same key the vouchers were signed over to — proving you hold the key proves you are authorized to retrieve those vouchers.

Both push and pull use the same cryptographic handshake (FDOKeyAuth, see [§ 16 FDOKeyAuth Protocol](#)); only the key differs. These mechanisms are described in detail in [§ 19.2 DID-Based Security Model](#). The Security Framework ([§ 17 Security Framework](#)) describes optional, additive layers (tokens, mTLS, business logic) that deployments **MAY** add for defense-in-depth, but these are not required for protocol security.

3.6. Reading Guide§

With this context, the rest of the specification is organized as follows:

- ****[§ 19 DID Integration](#) (DID Integration)**** — Start here to understand the trust model. DID exchange is the foundation for all security in this protocol.
- ****[§ 20 Key Formats and Enrollment Considerations](#) (Key Formats and Enrollment)**** — How keys are enrolled, why format doesn't matter after enrollment, and a comparison of DID and non-DID key formats.
- ****[§ 5 Transfer Models](#) – [§ 9 Service Root URLs](#) (Transfer Models, File Format, Root URLs)**** — The building blocks: how vouchers are structured, how push and pull work, and how endpoints are addressed.
- ****[§ 10 Push API Specification](#) and [§ 11 Pull API Specification](#) (Push and Pull APIs)**** — The protocol details: HTTP endpoints, request/response formats, and bearer token usage. Both require authorization first.
- ****[§ 12 Voucher Assignment](#) (Voucher Assignment)**** — How intermediaries (resellers, distributors) who lack FDO infrastructure can redirect vouchers to the end customer without taking cryptographic ownership.
- ****[§ 16 FDOKeyAuth Protocol](#) (Owner Key Authorization)**** — The FDOKeyAuth cryptographic handshake that produces bearer tokens for both Push and Pull. Same protocol, different key depending on direction.

- [**§ 17 Security Framework – § 18 Voucher Sequestering, § 21 Error Handling and Retry Logic – § 23 Implementation Guidelines**](#) — Reference material for implementers: optional security layers, error handling, and operational guidance. See in particular [§ 22.5.9 Application-Layer Protections: Comparison with FDO TO2](#) for why the REST APIs do not need TO2-style nonces, [§ 22.5.9.3 TLS Trust Models](#) for WebPKI vs. pinned CA tradeoffs, and [§ 23.2 Protocol Extensibility](#) for forward/backward compatibility rules for both JSON and CBOR structures.

One-liner: Exchange credentials once, optionally run a small FDOKeyAuth shim to get a bearer token (or use your own OAuth2/OIDC), then push, pull, and assign vouchers over plain REST/JSON — no specialized crypto in the business logic path.

4. Use Cases and Requirements§

4.1. Real-World Deployment Scenarios (Examples)§

4.1.1. Cloud Service Push§

A Voucher Issuer manufactures devices and immediately pushes vouchers to a cloud-based Voucher Recipient service. This scenario requires real-time delivery, high availability, and global accessibility. The Voucher Recipient's endpoint is always reachable over the public internet.

4.1.2. On-Premises Pull§

A Voucher Recipient operates in an isolated network (air-gapped, local data center) without inbound internet access. The Voucher Recipient periodically initiates outbound connections to retrieve vouchers from the Voucher Issuer. This scenario requires secure outbound connections, batch processing, and offline operation support.

4.1.3. On-Premises Push§

Other on-premises networks may allow periodic, or specific services to connect inward. This may allow vouchers to be pushed inward on a limited basis, i.e. only at certain times, or from services at specific IP addresses, or via gateways only with specific authorization.

4.1.4. Batch Transfer§

Manufacturing sites may go offline due to network outages or maintenance. Vouchers generated during offline periods SHOULD be queued and batch-uploaded when connectivity is restored. Similarly, Voucher Recipients MAY need to retrieve large volumes of vouchers efficiently during maintenance windows.

4.1.5. Reconciliation and Recovery§

System failures, data corruption, or disaster recovery scenarios require voucher re-synchronization. A Voucher Recipient MUST be able to request "all vouchers since a given checkpoint" to restore state after a failure. Checkpoints are typically timestamp-based; see the `since` query parameter in [§ 11.2 API Endpoints](#) for the Pull API implementation. Duplicate detection ensures that re-transferred vouchers do not cause errors.

4.1.6. Pagination§

Pull transfers involving large datasets MUST support pagination. Push transfers MAY support pagination for bulk uploads or rate-limited environments. Pagination tokens SHOULD be opaque to the client and managed entirely

by the server. See [§ 11.2 API Endpoints](#) for pagination details including continuation tokens and the `limit` parameter.

4.2. Security Requirements§

4.2.1. Authentication is Mandatory§

Voucher transfer endpoints MUST NOT be exposed without authentication and authorization. Both push and pull endpoints require bearer tokens obtained through the FDOKeyAuth cryptographic handshake (see [§ 16 FDOKeyAuth Protocol](#)). The protocol's primary security model is **DID-based mutual authentication** (see [§ 19.2 DID-Based Security Model](#)), where the one-time exchange of DIDs between trusted parties provides all necessary credentials. For push transfers, the Supplier authenticates with its enrolled key; for pull transfers, the Owner authenticates with the Owner Key. In both cases, voucher signature verification against enrolled manufacturer DID keys provides additional defense-in-depth.

4.2.2. DDoS Considerations§

Relying solely on voucher signature validation to reject unauthorized push requests places a computational burden on the Voucher Recipient, since the entire voucher must be parsed and cryptographically verified before rejection. Deployments exposed to high-volume or adversarial traffic SHOULD add a lightweight authentication layer (e.g., bearer tokens, mTLS) that enables early rejection at the edge before reaching the application. These are **additive defense-in-depth layers** (see [Appendix A: Optional Security Layers \(Informative\)](#)) — they are not required for protocol security but improve DDoS resilience.

4.2.3. Defense-in-Depth§

Deployments SHOULD use a layered approach:

Layer 1: Transport Security (REQUIRED)

- HTTPS with TLS 1.2+ [\[RFC8446\]](#)
- Certificate validation (WebPKI or DID-pinned CA, see [§ 19.5.3 Optional TLS Certificate Authority](#))

Layer 2: Protocol Authentication (REQUIRED)

- Push: FDOKeyAuth proving Supplier Key + voucher signature verification
- Pull: FDOKeyAuth proving Owner Key possession

Layer 3: Edge Protection (RECOMMENDED for internet-facing deployments)

- Token-based authorization (JWT) for fast edge/WAF rejection
- Mutual TLS with enrolled manufacturer certificates
- Rate limiting per source identity

Layer 4: Business Logic (OPTIONAL)

- Order validation, device registration checks, customer verification

5. Transfer Models§

5.1. Push Model (Issuer-Initiated)§

In the push model, the Voucher Issuer sends vouchers to the Voucher Recipient as soon as they are generated, or in response to a sales event (and voucher signover).

5.1.1. Flow§

1. Voucher Issuer generates voucher during device provisioning, signed over to the Recipient's Owner Key (obtained from the Recipient's DID)
2. Voucher Issuer resolves Voucher Recipient endpoint (configured or via DID, see [§ 19.3 Endpoint Discovery](#))
3. Voucher Issuer sends voucher via HTTP POST to the Push Service Root
4. Voucher Recipient verifies the voucher signature against enrolled manufacturer DID keys
5. Voucher Recipient accepts or rejects the voucher and returns status

5.1.2. Characteristics§

- **Latency:** Minimal — vouchers available immediately after generation
- **Availability:** Voucher Recipient endpoint MUST be reachable
- **DDoS Exposure:** Voucher Recipient is exposed to unsolicited requests
- **Complexity:** Simple retry logic on the issuer side

5.2. Pull Model (Recipient-Initiated)§

In the pull model, the Voucher Recipient (Recipient) retrieves vouchers from the Voucher Issuer or intermediary service (Holder) when ready.

5.2.1. Flow§

1. Holder generates and stores vouchers, signed over to the Recipient's Owner Key
2. Recipient authenticates to Holder endpoint using FDOKeyAuth (cryptographic proof of Owner Key possession; see [§ 16 FDOKeyAuth Protocol](#))
3. Recipient requests vouchers (optionally filtered by time, status, etc.)
4. Holder returns voucher metadata with pagination, scoped to the authenticated Owner Key
5. Recipient downloads individual voucher files
6. Recipient acknowledges receipt (optional)

The Recipient proves possession of the private key corresponding to the Owner Key to which vouchers have been signed over. This eliminates the need for pre-shared tokens — the Owner Key itself serves as the authorization credential. The Holder returns only vouchers whose final Owner Key matches the authenticated key.

5.2.2. Characteristics§

- **Latency:** Depends on polling interval
- **Availability:** Holder endpoint MUST be reachable; Recipient controls timing
- **DDoS Exposure:** Reduced — Recipient initiates all connections
- **Complexity:** Requires pagination, polling, or subscription infrastructure. FDOKeyAuth adds a multi-message handshake but eliminates token provisioning overhead

5.3. Hybrid Approaches§

Deployments MAY combine push and pull models:

- **Push with Pull Fallback:** Primary push delivery with pull capability for missed or failed vouchers
- **Pull with Notifications:** Pull model with webhook or other notification when new vouchers are available
- **Conditional Transfer:** Push for high-priority devices, pull for bulk processing

6. Voucher File Format§

6.1. File Extension§

Voucher files SHALL use the `.fdoov` file extension.

6.2. MIME Type§

The MIME type for FDO voucher files SHALL be `application/x-fdo-voucher`.

Implementations MAY also accept `application/octet-stream` as a fallback. When accepting `application/octet-stream`, implementations SHOULD verify the content is a valid ownership voucher by checking the CBOR structure and protocol version before further processing.

6.3. File Contents§

A `.fdoov` file is a binary-encoded FDO ownership voucher as defined in [\[FDO-Specification\]](#). The file contains:

1. Device information (serial number, model, GUID)
2. Manufacturer signature
3. Owner public key
4. Device attestation data

6.3.1. PEM Encoding§

Implementations SHALL wrap voucher CBOR in a PEM block to ensure interoperability with existing tooling:

- The PEM block type MUST be `OWNERSHIP VOUCHER`, using the literal header/footer strings `-----BEGIN OWNERSHIP VOUCHER-----` and `-----END OWNERSHIP VOUCHER-----`.
- Base64 output MUST be line-wrapped at 64 characters per line (per [\[RFC7468\]](#) Section 2) to avoid extremely long lines and to match the formatting emitted by the `go-fdo` CLI utilities.
- Files SHOULD terminate with a newline after the footer so downstream parsers that expect text files continue to work.

Voucher payloads MAY still be stored or transmitted as raw CBOR bytes when both parties explicitly agree, but the PEM-wrapped form is REQUIRED for interchange via this specification.

Support for PEM-encoded vouchers is REQUIRED for all implementations. Support for raw binary (non-PEM-encoded) vouchers MAY be supported by mutual agreement between parties.

6.4. Size Constraints§

Voucher Recipient implementations SHOULD accept voucher files up to 10 MB. Typical voucher files are expected to be well under 1 MB. The 10 MB size limit applies to the decompressed voucher content; compressed transfer sizes may be smaller.

Voucher files MAY be compressed using gzip. When compressed, the `Content-Encoding: gzip` header MUST be present.

7. Voucher Archive Format§

7.1. File Extension§

Voucher archive files SHALL use the `.fdozip` file extension.

7.2. Format§

The `.fdozip` format is a standard ZIP archive ([RFC1951](#), [RFC6713](#)) containing one or more `.fdoov` voucher files. This follows the pattern established by other domain-specific ZIP-based formats such as `.docx`, `.jar`, and `.epub` — the format is structurally a ZIP file but uses a distinct extension to convey semantic meaning.

Implementations that need to extract contents using standard ZIP tools MAY rename `.fdozip` files to `.zip`. Conversely, ZIP files containing `.fdoov` entries MAY be renamed to `.fdozip` for clarity.

7.3. MIME Type§

The MIME type for `.fdozip` files SHALL be `application/x-fdo-voucher-archive`.

Implementations MAY also accept `application/zip` as a fallback.

7.4. Archive Structure§

- All `.fdoov` files MUST be at the top level of the archive (not in subdirectories)
- Each entry MUST be a valid `.fdoov` PEM-encoded voucher
- Implementations SHOULD ignore non-`.fdoov` entries (e.g., `__MACOSX/` metadata, `README` files) rather than rejecting the archive
- Archive compression (DEFLATE) is OPTIONAL; implementations MUST support both compressed and stored entries

7.5. Size Constraints§

Implementations SHOULD accept `.fdozip` archives up to 100 MB uncompressed. The per-voucher size limits from [§ 6 Voucher File Format](#) apply to each contained `.fdoov` file.

8. Voucher Identification§

This section defines standard methods for computing voucher identifiers that are stable across the voucher's lifecycle. These identifiers enable consistent tracking of vouchers across manufacturing, supply chain, and owner systems without requiring access to the full voucher contents.

8.1. Voucher Digest§

The **Voucher Digest** is a cryptographic hash that uniquely identifies a voucher. It is computed as follows:

1. CBOR-encode the `0VHeader` (Ownership Voucher Header) structure using deterministic CBOR encoding ([RFC8949](#) Section 4.2)

2. CBOR-encode the `OVHeaderHMac` structure using deterministic CBOR encoding
3. Concatenate the two encoded byte strings: `DigestInput = CBOR(OVHeader) || CBOR(OVHeaderHMac)`
4. Compute: `VoucherDigest = SHA-256(DigestInput)`

The result is a 32-byte (256-bit) value.

NOTE: The digest input is a raw byte concatenation of two independently CBOR-encoded structures, NOT a CBOR array containing both structures. Implementations MUST concatenate the encoded bytes directly.

8.1.1. Rationale§

The digest is computed from the Header and HMAC only, deliberately excluding the certificate chain (`OVDevCertChain`) and ownership entries (`OVEntryArray`). This design provides two critical properties:

1. **Stability across extensions:** When a voucher is extended (signed over to a new owner), only the `OVEntryArray` changes. The Header and HMAC remain constant. Therefore, the Voucher Digest remains constant across all extensions of the same voucher.
2. **Device binding:** The HMAC is computed using the device's secret key during Device Initialize. Including the HMAC in the digest ensures the identifier is cryptographically bound to the specific device, not just to the header contents.

8.1.2. Security Considerations§

The concatenation order (Header first, HMAC second) is chosen deliberately. SHA-256 is vulnerable to length extension attacks where an attacker who knows $\text{SHA-256}(M)$ can compute $\text{SHA-256}(M || \text{padding} || \text{extension})$ without knowing M . By placing the HMAC (which contains unpredictable keyed material) at the end of the digest input, any length extension would append data after the HMAC, which has no security impact on the voucher digest use case. The digest is used solely for identification, not for authentication — the HMAC itself provides the cryptographic binding to the device secret.

8.1.3. String Representation§

When represented as a string, the Voucher Digest SHALL be encoded as lowercase hexadecimal (64 characters):

```
5b0527b7158dce7ca6dd6a96aec60e1df1cfbfd2229eb45bb5c734990d944d8b
```

Implementations MAY prefix the string with `sha256:` for clarity in contexts where multiple hash algorithms might be used:

```
sha256:5b0527b7158dce7ca6dd6a96aec60e1df1cfbfd2229eb45bb5c734990d944d8b
```

8.2. Voucher Fingerprint§

The **Voucher Fingerprint** is a shortened, human-readable form of the Voucher Digest, suitable for display in user interfaces, logs, and verbal communication.

8.2.1. Computation§

1. Compute the Voucher Digest as defined in [§ 8.1 Voucher Digest](#)
2. Take the first 16 bytes (128 bits) of the digest

3. Format as colon-separated lowercase hexadecimal pairs

8.2.2. Format§

The fingerprint is formatted as 16 hexadecimal byte pairs separated by colons:

```
5b:05:27:b7:15:8d:ce:7c:a6:dd:6a:96:ae:c6:0e:1d
```

This format (47 characters total) is inspired by the traditional SSH MD5 fingerprint format, providing familiarity for operators and easy visual comparison.

8.2.3. Use Cases§

- **CLI output:** Display fingerprints when listing or inspecting vouchers
- **Logging:** Include fingerprints in audit logs for voucher operations
- **Verbal verification:** Operators can read fingerprints aloud to verify voucher identity
- **Database indexing:** Use fingerprints as human-readable secondary identifiers alongside GUIDs

8.2.4. Security Note§

The 128-bit fingerprint provides sufficient collision resistance for identification purposes but **SHOULD NOT** be used as the sole basis for security-critical decisions. For cryptographic verification, use the full 256-bit Voucher Digest.

8.3. Comparison with GUID§

The voucher GUID (from `OVHeader.OVGuid`) and the Voucher Digest/Fingerprint serve different purposes:

Property	GUID	Voucher Digest/Fingerprint
Source	Assigned during Device Initialize	Computed from voucher contents
Uniqueness	Unique per device	Unique per voucher instance
Stability	Constant	Constant across extensions
Cryptographic binding	None (arbitrary identifier)	Bound to device via HMAC
Forgery resistance	Can be guessed or duplicated	Requires device secret to forge

Both identifiers are useful: the GUID provides a simple lookup key, while the Digest/Fingerprint provides cryptographic assurance that a voucher has not been tampered with and is bound to a specific device.

9. Service Root URLs§

9.1. Overview§

All API endpoints in this specification are defined as **path suffixes relative to a Service Root URL**. The Service Root URL is the base URL of a particular service instance, and all endpoints for that service are constructed by appending the defined suffix to the root.

This design allows the same protocol to operate in diverse deployment scenarios:

- **Sub-service integration:** The voucher transfer service can be mounted at any path within a larger application that already provides other REST APIs (e.g.,

`https://app.example.com/onboarding/vouchers).`

- **Reverse proxy compatibility:** Operators can define routing rules that steer traffic to the voucher service based on the root URL prefix, without requiring fixed absolute paths.
- **Multi-instance deployments:** Multiple independent voucher services can coexist on the same host at different root paths.

9.2. Service Types

This specification defines two service roles, each with its own Service Root URL:

9.2.1. Push Service Root

The **Push Service Root** is the base URL of a Voucher Recipient's push endpoint. It is the URL to which Voucher Issuers submit vouchers.

The Push Service Root is conveyed to Voucher Issuers through one of:

- The `voucherRecipientURL` field in a DID Document's `FD0VoucherRecipient` service entry (see [§ 19 DID Integration](#))
- Static configuration (e.g., a configured URL in the Voucher Issuer's settings)
- Out-of-band agreement between parties

Example: If the Push Service Root is `https://owner.example.com/fdo/v1/vouchers`, then the submit endpoint is `POST https://owner.example.com/fdo/v1/vouchers` and the status endpoint is `GET https://owner.example.com/fdo/v1/vouchers/{serial}/status`.

9.2.2. Pull Service Root

The **Pull Service Root** is the base URL of a Voucher Holder's pull endpoint. It is the URL from which Voucher Recipients retrieve vouchers.

The Pull Service Root is conveyed to Voucher Recipients through one of:

- The `voucherHolderURL` field in a DID Document's `FD0VoucherHolder` service entry (see [§ 19 DID Integration](#))
- Static configuration
- Out-of-band agreement between parties

Example: If the Pull Service Root is `https://holder.example.com/partner/vouchers`, then the `FDOKeyAuth hello` endpoint is `POST https://holder.example.com/partner/vouchers/auth/hello` and the list endpoint is `GET https://holder.example.com/partner/vouchers`.

9.3. Endpoint Notation

Throughout this specification, endpoint definitions use the notation `{root}` to refer to the applicable Service Root URL. For example:

- `POST {root}` means an HTTP POST to the Service Root URL itself
- `GET {root}/{id}/download` means an HTTP GET to the root URL with `{id}/download` appended

Implementations **MUST NOT** assume any particular absolute path structure. The path component of the Service Root URL is entirely deployment-specific.

9.4. Versioning§

The Service Root URL MAY include a version component (e.g., /api/v1/vouchers), but this is a deployment choice, not a protocol requirement. The protocol itself is versioned through the X-FD0-Version header and the ProtocolVersion field in FD0KeyAuth.Hello (see [§ 16.2.3 Message 1: FD0KeyAuth.Hello](#)). Servers that wish to support multiple protocol versions simultaneously MAY use distinct Service Root URLs for each version.

10. Push API Specification§

Both the Push API and Pull API require a bearer token obtained through the Owner Key Authorization protocol (see [§ 16 FD0KeyAuth Protocol](#)). A single authentication handshake produces a token that authorizes multiple push or pull operations within the token's lifetime.

Before calling any Push API endpoint, the Supplier MUST obtain a bearer token by completing the Owner Key Authorization handshake (see [§ 16 FD0KeyAuth Protocol](#)) with the Recipient's push endpoint. The Supplier presents its enrolled key (the key the Recipient knows from DID enrollment); the Recipient issues a bearer token scoped to that key. The bearer token is then included as Authorization: Bearer <token> in all Push API requests.

10.1. Submit Voucher§

10.1.1. POST {root}§

Submits a single voucher to the Voucher Recipient.

10.1.1.1. Request§

Content-Type: multipart/form-data [\[RFC7578\]](#)

Form Parameters:

- voucher (file, REQUIRED): The .fdoov voucher file
- serial (string, OPTIONAL): Device serial number
- model (string, OPTIONAL): Device model identifier
- manufacturer (string, OPTIONAL): Voucher Issuer identifier
- timestamp (string, OPTIONAL): Voucher generation timestamp (ISO 8601)

Headers:

- Content-Type: multipart/form-data; boundary=... (REQUIRED)
- Authorization: Bearer <token> or DPoP <token> (REQUIRED unless using mTLS) — obtained via Owner Key Authorization (see [§ 16 FD0KeyAuth Protocol](#)) or alternative token source. When using DPoP, include the DPoP header with a signed proof (see [§ 16.4.4 Using DPoP Tokens in API Requests](#)). When using mTLS authentication (see [§ 16.5 mTLS Authentication](#)), this header MAY be omitted.
- DPoP: Signed DPoP proof JWT (REQUIRED when using Authorization: DPoP)
- X-FD0-Version: Protocol version, e.g., 1.0 (RECOMMENDED)
- X-FD0-Client-ID: Client identifier for tracking (OPTIONAL)

10.1.1.2. Response§

Success Codes:

- 200 OK: Voucher accepted and processed synchronously
- 202 Accepted: Voucher accepted for asynchronous processing

Error Codes:

- 400 Bad Request: Invalid voucher format or missing required data
- 401 Unauthorized: Authentication required or failed
- 403 Forbidden: Authenticated but not authorized to submit vouchers
- 409 Conflict: Duplicate voucher for same device
- 413 Payload Too Large: Voucher file exceeds size limits
- 429 Too Many Requests: Rate limit exceeded;Retry-After header SHOULD be present
- 500 Internal Server Error: Server processing error
- 503 Service Unavailable: Temporarily unable to process;Retry-After header SHOULD be present

Response Body (JSON [\[RFC8259\]](#)):

```
{
  "status": "accepted",
  "voucher_id": "550e8400-e29b-41d4-a716-446655440000",
  "message": "Voucher accepted for processing",
  "timestamp": "2024-01-15T12:00:00Z"
}
```

The status field SHALL be one of: accepted, processed, pending, error.

10.2. Query Voucher Status

Implementations SHOULD provide an endpoint for querying the processing status of a specific voucher. This endpoint is shared across all transfer models — it is used by Voucher Recipients checking on pushed vouchers, by owners checking on vouchers available for pull, and by resellers checking on vouchers they have assigned (see [§ 12 Voucher Assignment](#)).

This capability is particularly valuable for diagnosing missing-voucher scenarios. Consider a deployment where a batch of devices is delivered and powered on, but one device fails to onboard. The error logs (or the device itself) report that no voucher is available. The device owner needs to determine whether the voucher was never sent, was sent but failed processing, was assigned to the wrong party, or was lost in transit — and potentially request re-delivery of that specific voucher. Without a status query endpoint, the only options are to request re-delivery of *all* vouchers (wasteful) or to manually search through logs.

10.2.1. GET {root}/status/{identifier}

Returns the processing status of a voucher, looked up by either the device serial number or the voucher GUID. Results are scoped to the caller's identity (see [§ 16.3.3 Token Scoping](#)) — a caller can only query status for vouchers associated with their identity.

The {identifier} path parameter SHALL be interpreted as follows:

- If it matches the format of a GUID (hex-encoded, 32 characters or with hyphens), look up by voucher GUID
- Otherwise, treat it as a device serial number

Implementations MAY support an explicit query parameter to disambiguate:GET {root}/status/{identifier}?type=serial or GET {root}/status/{identifier}?type=guid.

10.2.1.1. Request

Headers:

- Authorization: Bearer <token> or DPoP <token> (REQUIRED unless using mTLS) — obtained via Owner Key Authorization (see § 16 FDOKeyAuth Protocol) or an alternative token source (see § 16.3.5 Alternative Token Sources (Non-Normative)). When using DPoP, include the DPoP header. When using mTLS authentication (see § 16.5 mTLS Authentication), this header MAY be omitted.
- DPoP: Signed DPoP proof JWT (REQUIRED when using Authorization: DPoP)

10.2.1.2. Response

```
{
  "voucher_id": "550e8400-e29b-41d4-a716-446655440000",
  "serial": "ACME-2024-001234",
  "status": "pushed",
  "owner_key_fingerprint": "sha256:abcdef1234567890...",

  "created_at": "2024-01-10T08:00:00Z",
  "signed_over_at": "2024-01-15T12:00:00Z",

  "pushed_at": "2024-01-15T12:01:30Z",
  "pushed_to": "https://customer.example.com/fdo/vouchers",
  "push_attempts": 1,
  "last_push_error": null,

  "pulled_at": null,
  "pulled_by_fingerprint": null,

  "assigned_at": null,
  "assigned_to_fingerprint": null,
  "assigned_to_did": null,
  "assigned_by_fingerprint": null,

  "error_message": null
}
```

10.2.1.3. Status Values

The status field SHALL be one of the following values, representing the current state of the voucher at the Holder:

Status	Description
held	Voucher is stored at the Holder, not yet signed over or delivered. This is the initial state for vouchers awaiting sign-over.
signed_over	Voucher has been signed over to an owner key but not yet delivered (neither pushed nor pulled).
pushed	Voucher was successfully pushed to the owner’s endpoint.
pulled	Voucher was downloaded by the owner via the Pull API.
assigned	Voucher was assigned to a new owner (see § 12 Voucher Assignment). The assigned_to_* fields identify the new owner.

Status	Description
failed	An error occurred during processing. The error_message field provides details.
unknown	The identifier format is recognized but the server cannot determine the voucher's state (e.g., the voucher was received but not yet indexed).

10.2.1.4. Response Fields

The only REQUIRED fields are voucher_id, serial, and status. All other fields are OPTIONAL — implementations return them when available, and clients MUST NOT assume any field other than these three will be present.

Field	Type	Requirement	Description
voucher_id	string	REQUIRED	Voucher GUID (hex-encoded)
serial	string	REQUIRED	Device serial number
status	string	REQUIRED	Current voucher state (see Status Values above)
owner_key_fingerprint	string	OPTIONAL	SHA-256 fingerprint of the current owner key the voucher is signed over to (or will be signed over to)
created_at	string	OPTIONAL	ISO 8601 timestamp when the voucher was created
signed_over_at	string	OPTIONAL	ISO 8601 timestamp when the voucher was signed over to the owner key
pushed_at	string	OPTIONAL	ISO 8601 timestamp of the most recent successful push
pushed_to	string	OPTIONAL	URL to which the voucher was pushed. Implementations MAY omit this field for privacy reasons.
push_attempts	integer	OPTIONAL	Number of push attempts (successful and failed)
last_push_error	string	OPTIONAL	Error message from the most recent failed push attempt, or null
pulled_at	string	OPTIONAL	ISO 8601 timestamp when the voucher was downloaded via pull
pulled_by_fingerprint	string	OPTIONAL	SHA-256 fingerprint of the key that authenticated the pull download
assigned_at	string	OPTIONAL	ISO 8601 timestamp when the voucher was assigned
assigned_to_fingerprint	string	OPTIONAL	SHA-256 fingerprint of the new owner key the voucher was assigned to
assigned_to_did	string	OPTIONAL	DID of the new owner, if the new owner's credentials were provided as a DID
assigned_by_fingerprint	string	OPTIONAL	SHA-256 fingerprint of the key (or identity) that authorized the assignment
error_message	string	OPTIONAL	Human-readable error description when status is failed

If the identifier does not match any known voucher within the caller's scope, the server SHALL return 404 Not Found.

NOTE: This endpoint is RECOMMENDED, not REQUIRED. Implementations that do not support status queries SHOULD return 501 Not Implemented.

11. Pull API Specification

Both the Push API and Pull API require a bearer token obtained through the Owner Key Authorization protocol (see [§ 16 FDOKeyAuth Protocol](#)). A single authentication handshake produces a token that authorizes multiple push or pull operations within the token's lifetime.

11.1. Overview§

The Pull API allows a Voucher Recipient (the **owner**) to retrieve vouchers that have already been signed over to the owner's key but are held by a Supplier's voucher management service (the **Holder**). This is the complement to the Push API: instead of the Supplier delivering vouchers to the owner, the owner reaches out to the Supplier to fetch them.

Pull is the natural choice when the owner's management service cannot accept inbound connections — for example, an on-premises deployment behind a corporate firewall, or any setup where the owner prefers to initiate connections rather than expose an endpoint.

Before calling any Pull API endpoint, the Recipient **MUST** obtain a bearer token by completing the Owner Key Authorization handshake (see [§ 16 FDOKeyAuth Protocol](#)) with the Holder. The Recipient presents the Owner Key that vouchers were signed over to; the Holder issues a bearer token scoped to that key. The bearer token is then included as `Authorization: Bearer <token>` in all Pull API requests.

11.2. API Endpoints§

All Pull API endpoints require a bearer token obtained through the Owner Key Authorization protocol (see [§ 16 FDOKeyAuth Protocol](#)). The `SessionToken` from the authentication result is used as `Authorization: Bearer <SessionToken>` in all requests below.

11.2.1. List Vouchers§

11.2.1.1. GET {root}§

Returns voucher metadata for the authenticated owner. Results are paginated and scoped to only return vouchers whose final Owner Key matches the authenticated key.

11.2.1.2. Request§

Query Parameters:

- `since` (timestamp, OPTIONAL): Return vouchers created after this time (ISO 8601)
- `until` (timestamp, OPTIONAL): Return vouchers created before this time
- `status` (string, OPTIONAL): Filter by status (pending, downloaded, all). Default: pending
- `limit` (integer, OPTIONAL): Maximum vouchers to return. Default: 100. Maximum: 1000
- `continuation` (string, OPTIONAL): Opaque continuation token from a previous response. See [§ 16.8 Continuation Token Security](#) for security considerations regarding continuation tokens.
- `fields` (string, OPTIONAL): Comma-separated list of voucher fields to include in the response (e.g., `fields=voucher_id,serial,created_at`). If omitted, the server returns all fields it supports. Servers **MUST** always include `voucher_id` regardless of this parameter. Servers **SHOULD** silently ignore field names they do not recognize or support.

Headers:

- `Authorization: Bearer <token>` or `DPoP <token>` (REQUIRED unless using mTLS). When using mTLS authentication (see [§ 16.5 mTLS Authentication](#)), this header **MAY** be omitted.
- `DPoP`: Signed DPoP proof JWT (REQUIRED when using `Authorization: DPoP`)

11.2.1.3. Response§

```

{
  "vouchers": [
    {
      "voucher_id": "550e8400-e29b-41d4-a716-446655440000",
      "serial": "ABC123",
      "model": "model-a",
      "manufacturer_id": "mfg-abc123",
      "status": "pending",
      "created_at": "2024-01-15T12:00:00Z",
      "size_bytes": 1024,
      "checksum": "sha256:abcdef1234567890..."
    }
  ],
  "continuation": "opaque-token-abc123",
  "has_more": true,
  "total_count": 1500
}

```

11.2.1.4. Voucher Fields

The only REQUIRED field in each voucher entry is `voucher_id`. All other fields are OPTIONAL — implementations return them when available, and clients MUST NOT assume any field other than `voucher_id` will be present.

Field	Type	Requirement	Description
<code>voucher_id</code>	string	REQUIRED	Voucher GUID (hex-encoded)
<code>serial</code>	string	OPTIONAL	Device serial number
<code>model</code>	string	OPTIONAL	Device model identifier
<code>manufacturer_id</code>	string	OPTIONAL	Voucher Issuer identifier
<code>status</code>	string	OPTIONAL	pending, downloaded, etc.
<code>created_at</code>	string	OPTIONAL	ISO 8601 timestamp
<code>size_bytes</code>	integer	OPTIONAL	Voucher file size
<code>checksum</code>	string	OPTIONAL	sha256:{hash} of voucher file. Checksum values MUST include the algorithm prefix (e.g., sha256:) to allow for future hash algorithm support.

Not all implementations will have all fields available. For example, an implementation that stores vouchers as opaque CBOR blobs may not extract `serial` or `model` at ingest time. Clients using the `fields` parameter can reduce response size by requesting only the fields they need.

11.2.1.5. Pagination

The `continuation` field SHALL be present when `has_more` is `true`. Clients SHALL pass this value unchanged in the `continuation` query parameter of the next request to retrieve the next page.

Continuation tokens are opaque to the client. Servers MAY encode position, filters, expiry, or other state within the token. Continuation tokens SHOULD expire after a reasonable period (e.g., 1 hour).

The `total_count` field is OPTIONAL. Computing a total count may require a full table scan or separate counting query, which can be expensive on large datasets. Servers that cannot efficiently compute a total count SHOULD omit this field rather than return an inaccurate value. Clients MUST NOT depend on `total_count` being present — the `has_more` flag is the authoritative signal for whether more pages exist.

11.2.2. Download Voucher§

11.2.2.1. GET {root}/{voucher_id}/download§

Downloads the actual voucher file.

11.2.2.2. Request§

Headers:

- Authorization: Bearer <token> or DPoP <token> (REQUIRED unless using mTLS). When using mTLS authentication (see [§ 16.5 mTLS Authentication](#)), this header MAY be omitted.
- DPoP: Signed DPoP proof JWT (REQUIRED when using Authorization: DPoP)

11.2.2.3. Response§

Headers:

- Content-Type: application/x-fdo-voucher
- Content-Disposition: attachment; filename="{voucher_id}.fdoov"
- Content-Length: File size in bytes
- X-FD0-Checksum: sha256:{hash}

Body: Raw .fdoov file content.

11.2.3. Subscription and Notification (OPTIONAL)§

Implementations MAY provide one or both of the following notification mechanisms to allow clients to efficiently detect new voucher availability without polling the List endpoint. These endpoints are OPTIONAL — implementations that do not support them SHOULD return 501 Not Implemented.

11.2.3.1. Long-Polling§

11.2.3.1.1. GET {root}/SUBSCRIBE§

Returns immediately if vouchers are available; otherwise blocks up to the specified timeout.

Query Parameters:

- timeout (integer, OPTIONAL): Maximum wait time in seconds. Default: 30. Maximum: 120
- since (timestamp, OPTIONAL): Only return vouchers created after this time

Response: Same format as the List Vouchers endpoint.

11.2.3.2. Server-Sent Events§

11.2.3.2.1. GET {root}/STREAM§

Returns a Server-Sent Events stream of voucher availability notifications.

Response:

Content-Type: text/event-stream
Cache-Control: no-cache

```
retry: 5000
event: voucher_available
data: {"voucher_id": "uuid-1", "serial": "ABC123", "created_at": "2024-01-15T12:00:00Z"}

event: voucher_available
data: {"voucher_id": "uuid-2", "serial": "DEF456", "created_at": "2024-01-15T12:01:00Z"}
```

Clients receiving these events SHOULD use the Download Voucher endpoint to retrieve the actual voucher file.

11.3. Assignment (Pull Without Sign-Over)[§](#)

In some supply chains, the party associated with vouchers at the Holder does not intend to take delivery of those vouchers themselves. Instead, they wish to redirect the vouchers to a different party — for example, a reseller who purchased devices from a manufacturer but is reselling them to an end customer.

This operation — **assignment** — is a variant of the pull model where the caller interacts with the Holder's pull endpoint but, instead of downloading vouchers, instructs the Holder to sign them over to a different owner and deliver them to that owner.

Assignment is defined in its own section (see [§ 12 Voucher Assignment](#)) because it has distinct authentication requirements (the caller may not have an FDO owner key) and distinct semantics (the caller is not the sign-over target). However, assignment operates on the same set of vouchers as the Pull API and uses the same Service Root URL — the Assign endpoint (POST {root}/assign) is an additional endpoint alongside the Pull API's List and Download endpoints.

12. Voucher Assignment[§](#)

12.1. Overview[§](#)

In many supply chains, devices pass through intermediaries — resellers, distributors, or procurement partners — who purchase devices from a manufacturer but do not operate FDO infrastructure themselves. These intermediaries are **commercial participants** in the supply chain but not **cryptographic participants** in the voucher chain.

Consider a typical scenario: Reseller B purchases devices from Manufacturer A. Customer C then purchases those devices from B. B does not run a voucher management service — B may not even have an FDO owner key. But C cares about FDO and needs the vouchers signed over to C's key and delivered to C's management service.

Without assignment, B would need to: receive the vouchers from A (requiring FDO infrastructure), sign them over to C's key (requiring B's private key and signing capability), and push them to C (requiring integration with C's endpoint). This is a significant burden for a party that is simply reselling equipment.

The **Assign API** allows B to instruct A: "sign these vouchers over to C instead of me, and deliver them to C." B authenticates to A using whatever business credentials B has (portal login, API token, OAuth — see [§ 16.3.3 Token Scoping](#) and [§ 16.3.5 Alternative Token Sources \(Non-Normative\)](#)), provides C's credentials (key and optionally a push URL), and A handles the sign-over and delivery directly.

12.2. When Assignment Applies[§](#)

12.2.1. Build-to-Stock (Assignment)[§](#)

Assignment is designed for the **build-to-stock** scenario:

1. B purchases devices from A. A manufactures the devices and holds the vouchers, pending sign-over.
2. Time passes. C purchases devices from B's inventory.
3. B calls A's Assign API, providing C's credentials and the serial numbers of the devices C purchased.
4. A signs the vouchers over to C's key and delivers them — by pushing to C's endpoint (if C provided a push URL via DID), by holding them for C to pull from A, or by making them available for out-of-band retrieval.

12.2.2. Build-to-Order (Not Assignment)§

In a **build-to-order** scenario, the flow is different and does not require the Assign API:

1. C orders devices from B, providing C's DID or credentials during the purchase.
2. B orders devices from A, providing **C's credentials** (not B's) as the owner to sign vouchers over to.
3. A manufactures the devices, signs vouchers directly to C's key, and delivers to C.

This is standard enrollment through an intermediary — B simply passes C's credentials to A during ordering. No assignment is needed because the vouchers were never associated with B.

However, this flow introduces a **credential relay** requirement: B must also provide C with A's credentials (A's DID or signing key) so that C can verify vouchers from A and, if pull-based transfer is used, discover A's pull endpoint. The timing of this relay may be awkward — B may not know A's credentials until B places the order with A, which happens after C places the order with B. This relay is an out-of-band process and its mechanism is outside the scope of this specification.

12.3. Cryptographic Constraint§

Assignment is only possible for vouchers that the Holder has **not yet signed over** to the current purchaser. In the FDO ownership voucher model, extending the voucher chain requires the current owner's private key. Once a voucher has been signed over to B's key, only B's private key can sign the next OVerEntry to extend the chain to C — and B, as a non-FDO-aware reseller, does not have the capability to perform this operation.

Therefore, the Assign API operates on vouchers that the Holder (A) is **holding on behalf of** the purchaser (B) but has not yet committed to the voucher chain. The Holder signs the voucher directly to the new owner's (C's) key, resulting in a voucher chain that goes A → C — B never appears in the cryptographic chain.

Implementations **MUST** reject assignment requests for vouchers that have already been signed over to the purchaser's key. The appropriate error response is 409 `Conflict` with an error code indicating the voucher has already been signed over (see [§ 12.5 Assign Endpoint](#)).

12.4. Authentication§

The Assign API requires a bearer token in the `Authorization` header, consistent with all other APIs in this specification. However, the expected authentication path for assignment differs from push and pull:

- **Push and Pull:** The caller typically authenticates via `FDOKeyAuth` (see [§ 16 FDOKeyAuth Protocol](#)), proving possession of an enrolled FDO key.
- **Assignment:** The caller (B) may not have an FDO owner key at all. B is a reseller who has a business relationship with A — an account on A's partner portal, an API key, OAuth credentials — but no FDO-specific cryptographic material.

The Assign API therefore relies on **alternative token sources** (see [§ 16.3.5 Alternative Token Sources \(Non-Normative\)](#)) as the primary authentication path. Implementations **MUST** accept bearer tokens that identify the caller and scope access to vouchers associated with that caller's identity. How the token is obtained — `FDOKeyAuth`, OAuth, partner portal, API key — is an implementation concern.

If the caller does have an FDO owner key and authenticates via FDOKeyAuth, the Assign API works identically — the token still maps to the caller's identity, and vouchers are scoped accordingly.

12.5. Assign Endpoint

12.5.1. POST {root}/assign

Assigns one or more vouchers to a new owner. The caller specifies vouchers by serial number and provides the new owner's credentials.

12.5.1.1. Request

Headers:

- Authorization: Bearer <token> (REQUIRED)
- Content-Type: application/json (REQUIRED)

Body (JSON):

```
{
  "serials": ["SN-001", "SN-002", "SN-003"],
  "new_owner_did": "did:web:customer-c.example.com"
}
```

Or, when the new owner's credentials are not expressed as a DID:

```
{
  "serials": ["SN-001", "SN-002", "SN-003"],
  "new_owner_key": "<PEM-encoded public key or base64-encoded COSE_Key>",
  "new_owner_key_format": "pem",
  "push_url": "https://customer-c.example.com/fdo/vouchers"
}
```

Fields:

- serials (array of strings, REQUIRED): Device serial numbers identifying the vouchers to assign. Each serial MUST correspond to a voucher associated with the caller's identity. Callers SHOULD specify exact serial numbers rather than broad patterns to prevent accidental mass-assignment.
- new_owner_did (string, CONDITIONAL): DID of the new owner. When present, the Holder resolves the DID to obtain the new owner's public key and, optionally, push/pull service endpoint URLs. Either new_owner_did or new_owner_key MUST be present, but not both.
- new_owner_key (string, CONDITIONAL): The new owner's public key, PEM-encoded or base64-encoded. Required when new_owner_did is not provided.
- new_owner_key_format (string, CONDITIONAL): Format of new_owner_key. SHALL be one of: pem, cose, x509. Required when new_owner_key is present.
- push_url (string, OPTIONAL): URL to which the Holder SHOULD push assigned vouchers. Only applicable when new_owner_key is used (DIDs convey push URLs via their service entries). Ignored if the Holder does not support push delivery.

12.5.1.2. Server Processing

1. The Server verifies the bearer token and identifies the caller.

2. For each serial number in `serials`, the Server:

- Verifies the serial corresponds to a voucher associated with the caller's identity
- Verifies the voucher has NOT already been signed over or assigned (see [§ 12.3 Cryptographic Constraint](#))
- If `new_owner_did` is present, resolves the DID to obtain the new owner's public key and service endpoints
- Signs the voucher over to the new owner's public key
- Records the assignment (for status queries)

3. If the new owner's credentials include a push URL (from the DID document or the `push_url` field), the Server SHOULD push the signed voucher to that URL. The timing of push delivery is an implementation decision — it MAY happen immediately, on a schedule, or according to external arrangements (see [§ 3.4 Transfer: Push, Pull, or Out-of-Band](#)).

4. If no push URL is available, the Server holds the signed voucher. The new owner (C) MAY retrieve it via the Pull API (authenticating with the owner key the voucher was signed to via `FDOKeyAuth`) or via out-of-band transfer.

12.5.1.3. Response

Success (HTTP 200 OK):

```
{
  "results": [
    {
      "serial": "SN-001",
      "status": "ok",
      "voucher_id": "550e8400-e29b-41d4-a716-446655440000"
    },
    {
      "serial": "SN-002",
      "status": "pending",
      "voucher_id": "660e8400-e29b-41d4-a716-446655440001"
    },
    {
      "serial": "SN-003",
      "status": "error",
      "error_code": "not_found",
      "error_message": "No voucher found for serial SN-003"
    }
  ]
}
```

The status field for each result SHALL be one of:

- `ok`: Voucher was successfully assigned (and signed over to the new owner's key)
- `pending`: Assignment accepted but sign-over or delivery is not yet complete. The caller SHOULD query voucher status (see [§ 10.2 Query Voucher Status](#)) for updates.
- `error`: Assignment failed for this voucher. The `error_code` and `error_message` fields provide details.

Error Codes (per-voucher):

- `not_found`: No voucher matches this serial number within the caller's scope
- `already_signed_over`: Voucher has already been signed over and cannot be assigned (see [§ 12.3 Cryptographic Constraint](#))

- `already_assigned`: Voucher has already been assigned to another party
- `invalid_key`: The new owner's key could not be parsed or is in an unsupported format
- `did_resolution_failed`: The new owner's DID could not be resolved

HTTP-Level Errors:

- 400 Bad Request: Malformed JSON, missing required fields, or both `new_owner_did` and `new_owner_key` present
- 401 Unauthorized: Authentication required or failed
- 403 Forbidden: Authenticated but not authorized to assign vouchers
- 429 Too Many Requests: Rate limit exceeded

12.6. Credential Relay

After assignment, the new owner (C) needs the manufacturer's (A's) credentials in order to:

- **Verify pushed vouchers**: C's management service needs A's signing key to verify that incoming vouchers are authentic
- **Discover A's pull endpoint**: If C will pull vouchers from A, C needs A's Pull Service Root URL
- **Authenticate to A for pull**: C authenticates to A via FDOKeyAuth using C's owner key — this is self-proving and requires no prior enrollment of C at A. However, C still needs to know A's endpoint URL.

The reseller (B) SHOULD provide A's credentials (DID or key + URLs) to C when fulfilling C's order. The mechanism for this relay is out of scope for this specification — it is part of the business relationship between B and C. In the build-to-stock scenario, B already knows A's credentials from B's original purchase, so B can provide them to C at order fulfillment time.

Note that C does **not** need to enroll with A for pull authentication. FDOKeyAuth proves possession of the owner key, and A scopes the response to vouchers signed to that key. Since A signed the vouchers to C's key during assignment, C's FDOKeyAuth handshake will match those vouchers. The only information C needs from B is how to reach A's service (the endpoint URL).

12.7. Assignment Immutability

Once a voucher has been assigned, the Assign API MUST reject subsequent assignment attempts for that voucher with the `already_assigned` error code. A voucher can be assigned at most once through this API.

12.7.1. Rationale

This restriction is deliberate. While the Holder could technically sign a voucher over to a different key (the voucher has not left the Holder's custody), allowing programmatic re-assignment would create supply chain integrity risks:

- **Accidental double-assignment**: A scripting error could assign vouchers intended for customer C to customer D
- **Audit trail ambiguity**: Multiple assignments make it difficult to determine the authoritative owner of a voucher
- **Security risk**: An attacker who compromises B's credentials could redirect vouchers that B has already assigned to C

The immutability rule ensures that mistakes require deliberate human intervention to correct, which is appropriate for operations that affect the cryptographic ownership of physical devices.

12.7.2. Error Recovery (Non-Normative)

In real-world supply chains, mistakes happen — serial numbers may be assigned to the wrong customer, orders may be cancelled after vouchers are assigned, or testing devices may need to be redirected. This specification intentionally does **not** provide an API for reversing or re-doing an assignment.

When an assignment must be reversed, the reseller (B) **MUST** contact the Holder (A) through an out-of-band process — for example, a support ticket, a phone call, or a dedicated operations interface. The Holder’s administrator manually reverses the assignment in their internal systems. After reversal, the voucher returns to a held state from the API’s perspective and can be assigned again through the normal Assign API.

This out-of-band reversal process is deliberately implementation-specific. The friction of requiring human intervention is a feature — it prevents casual re-assignment and ensures that ownership changes to physical devices receive appropriate scrutiny.

NOTE: The FDO ecosystem has precedent for this "lock once, manual unlock" pattern. The FDO Rendezvous Service locks the first RV entry, requiring it to expire before another party can update it. The assignment immutability rule follows the same principle — once committed, reversal requires a more privileged operation than the original assignment.

13. Authentication Framework

All Voucher Operations (Push, Pull, Assign) require authentication. This section provides an overview of the authentication architecture, which is organized into three distinct layers:

- 1. **Voucher Operations** (Layer 1) — The APIs that do the work: Push, Pull, Assign
- 2. **Request Authentication** (Layer 2) — How callers prove identity on each request: Bearer tokens, DPOP tokens, or mTLS
- 3. **Token Establishment** (Layer 3) — How callers obtain tokens: FDOKeyAuth, OAuth/OIDC, or out-of-band provisioning

This layered model allows flexibility: a caller can use FDOKeyAuth to obtain a DPOP token, or OAuth to obtain a Bearer token, or mTLS to bypass tokens entirely. The Voucher Operations APIs are agnostic to which combination is used.

13.1. The Three Layers

13.1.1. Layer 1: Voucher Operations

The Voucher Operations are the core APIs that perform voucher management work:

Operation	HTTP Method	Purpose
Push	POST {root}	Submit vouchers to a recipient
Pull	GET {root}, GET {root}/{id}/download	List and retrieve vouchers from a holder
Assign	POST {root}/assign	Associate vouchers with a new owner key

These APIs are standard REST/JSON — no CBOR, no COSE, no FDO-specific encoding. Any HTTP client can call them. Authentication is provided via Layer 2.

13.1.2. Layer 2: Request Authentication

Request Authentication is how the caller proves identity on each API request. This specification supports three methods:

Method	HTTP Mechanism	Token Required	Best For
Bearer Token	Authorization: Bearer <token>	Yes	Simple integrations, standard tooling
DPoP Token	Authorization: DPoP <token> + DPoP: <proof>	Yes	High-security, token theft mitigation
mTLS	Client certificate in TLS handshake	No	Point-to-point, enterprise PKI

- **Bearer tokens** are the simplest: possession of the token grants access. See [§ 14.1 Bearer Tokens](#).
- **DPoP tokens** are sender-constrained: the token is bound to a key, and each request includes a signed proof. See [§ 14.2 DPoP Tokens \(Sender-Constrained\)](#).
- **mTLS** authenticates via client certificate at the transport layer — no token is needed. See [§ 14.3 mTLS Authentication](#).

Layer 2 is about *using* credentials. How those credentials are *obtained* is Layer 3.

13.1.3. Layer 3: Token Establishment

Token Establishment is how callers obtain the tokens used in Layer 2. Methods include:

Method	Normative Status	Produces	Best For
FDOKeyAuth	Normative (this spec)	Bearer or DPoP token	Owners with FDO keys
OAuth/OIDC	Non-normative (external specs)	Bearer or DPoP token	Enterprise identity integration
Portal/Out-of-Band	Non-normative	Bearer token	Partner portals, API keys

- **FDOKeyAuth** is the FDO-specific protocol defined in [§ 16 FDOKeyAuth Protocol](#). It uses Owner Keys and a cryptographic challenge-response handshake.
- **OAuth/OIDC** uses external identity providers. See [§ 15.1 OAuth/OIDC Integration \(Non-Normative\)](#).
- **Portal/Out-of-Band** covers pre-provisioned tokens from partner portals or manual processes. See [§ 15.2 Portal/Out-of-Band Tokens \(Non-Normative\)](#).

NOTE: **mTLS can bypass Layer 3** when used alone — the client certificate is the credential. However, modern infrastructure often terminates TLS at load balancers or gateways, breaking end-to-end mTLS. For high-security deployments, combining mTLS (Layer 2) with token authentication (Layers 2+3) provides defense-in-depth: mTLS secures the transport while tokens provide application-level identity that survives TLS termination. See [§ 16.5 mTLS Authentication](#).

13.2. Authentication Paths

Different callers take different paths through the layers:

13.2.1. Owner with FDO Key§

An Owner who possesses an FDO Owner Key (the key at the tip of voucher ownership chains):

FDOKeyAuth (Layer 3) → Bearer or DPoP token → Push/Pull/Assign APIs (Layer 1)

The Owner runs the FDOKeyAuth protocol, receives a token, and uses it for API calls.

13.2.2. Owner with Enterprise Identity§

An Owner whose organization uses OAuth/OIDC for identity:

OAuth/OIDC (Layer 3) → Bearer or DPoP token → Push/Pull/Assign APIs (Layer 1)

The Owner authenticates via their enterprise identity provider, receives a token, and uses it for API calls. The server maps the OAuth identity to an Owner Key via configuration.

13.2.3. Custodian with mTLS§

A Custodian (business authority but no Owner Key) using mTLS:

mTLS (Layer 2) → Push/Pull/Assign APIs (Layer 1)

The Custodian presents a client certificate during TLS handshake. No token is needed. The server maps the certificate to a Custodian identity via configuration.

13.2.4. Custodian with Portal Token§

A Custodian using a partner portal:

Portal login (Layer 3) → Bearer token → Push/Pull/Assign APIs (Layer 1)

The Custodian logs into a web portal, obtains a token, and uses it for API calls.

13.3. Design Rationale: Architectural Separation§

A fundamental architectural principle of this specification is the **clean separation between cryptographic authentication and business-logic APIs**. The Push, Pull, Assign, and Status APIs are designed to be **standard REST/JSON APIs with standard authentication headers**:

1. All request and response bodies are **JSON** — no CBOR encoding
2. All endpoints follow **standard REST conventions** — GET for retrieval, POST for submission
3. All authentication uses **standard HTTP mechanisms** — Authorization header or mTLS

Any IT engineer with standard tools — `curl`, Postman, Python requests, shell scripts, CI/CD pipelines — can interact with these APIs without specialized knowledge of FDO, CBOR, COSE, or custom cryptographic protocols. This is essential because voucher transfer must integrate with business logic systems (order management, inventory, procurement, ERP) that are built on standard REST/JSON tooling.

The **FDOKeyAuth protocol** (see [§ 16 FDOKeyAuth Protocol](#)) is the sole exception. It uses CBOR encoding, FDO owner keys, a custom challenge-response protocol, and COSE cryptographic operations. Implementing FDOKeyAuth requires specialized knowledge and libraries that a typical IT integration engineer would not have.

This separation is deliberate, and the token is the bridge between the two worlds. In practice, this means:

1. **IT teams integrate the REST APIs** into their business logic using standard tools and practices. Pushing

vouchers, pulling voucher lists, checking status, assigning — all standard REST with a bearer token. The integration code is the same Java, Python, PHP, shell script, or any other language the team already uses for REST/JSON APIs.

2. **If they use FDOKeyAuth**, they use a **standalone tool or utility** — a standard, open-source command-line program that takes an owner key as input, runs the FDOKeyAuth protocol, and outputs a bearer token. That is the tool's entire purpose. The integration team does not need to understand CBOR, COSE, or the challenge-response protocol — they run the tool, receive a token, and use it with the REST APIs they already know how to call.
3. **Or they skip FDOKeyAuth entirely** and obtain a token from an existing identity provider (OAuth2, OIDC), a partner portal, or any other token source their organization already uses. The REST APIs do not care how the token was obtained.

To illustrate, here is a complete terminal session — authenticate with an owner key, then retrieve recent vouchers using standard curl:

```
# Step 1: Run the FDOKeyAuth tool. It handles CBOR/COSE internally,
#           outputs a plain bearer token.
$ TOKEN=$(fdo-auth --key owner-key.pem --url https://mfg.example.com/api/v1)
Authenticated as key af3b...c8d1 – token expires in 3600s

# Step 2: Use the token with ordinary curl. No FDO libraries needed.
$ curl -s -H "Authorization: Bearer $TOKEN" \
    "https://mfg.example.com/api/v1/vouchers?since=2026-03-01" | jq .
{
  "vouchers": [
    { "serial": "DMC-48291", "guid": "a1b2c3...", "status": "ready" },
    { "serial": "DMC-48292", "guid": "d4e5f6...", "status": "ready" }
  ],
  "total_count": 2
}
```

The fdo-auth tool is the only component that touches CBOR, COSE, or owner keys. Everything after it — the curl call, the JSON response, the bearer token header — is standard REST that works with any HTTP client, scripting language, or API gateway.

13.4. Owner vs Custodian Authentication

The authentication framework distinguishes between two types of callers:

Caller Type	Has Owner Key?	Can be Sign-Over Target?	Typical Auth Methods
Owner	Yes	Yes	FDOKeyAuth, OAuth (mapped to key)
Custodian	No	No	mTLS, OAuth, Portal tokens

Owners possess an FDO Owner Key — the private key corresponding to a public key that appears (or will appear) in voucher ownership chains. Owners can:

- Authenticate via FDOKeyAuth (proving key possession)
- Be the target of voucher sign-over operations
- Pull vouchers signed to their key

Custodians have business authority over devices but no Owner Key. They are known to the server through out-of-band relationships (contracts, portal accounts, mTLS certificates). Custodians can:

- Push vouchers on behalf of a supplier
- Pull vouchers they have been granted access to
- Direct the server to assign vouchers to a designated Owner
- Query status of vouchers they have access to

Custodians **cannot** be sign-over targets (they have no key to sign to). When a Custodian needs vouchers signed to a specific party, they use the Assign API to direct the server to perform the sign-over.

14. Request Authentication§

This section defines the three Request Authentication methods: how callers prove identity on each API request to Voucher Operation endpoints.

14.1. Bearer Tokens§

Bearer tokens are the simplest authentication method. The caller includes the token in the `Authorization` header:

```
GET /api/v1/vouchers HTTP/1.1
Host: holder.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

Possession of the token grants access — no additional proof is required. This is the standard mechanism used by virtually every modern REST API.

14.1.1. Token Format§

This specification does not mandate a specific token format. Tokens **MAY** be:

- Opaque strings (server stores token → identity mapping)
- JWTs (self-contained, signed claims)
- Any other format the server can validate

The token is obtained via Token Establishment (Layer 3) — see [§ 15 Token Establishment](#).

14.1.2. Token Scoping§

Servers **SHOULD** scope tokens to limit what operations they authorize:

Scope	Description
Owner-key scoped	Token authorizes access only to vouchers associated with a specific Owner Key
Custodian scoped	Token authorizes access only to vouchers the Custodian has been granted access to
Admin scoped	Token authorizes broad access (use sparingly)

Scoping is a server-side policy decision. The token itself may or may not encode the scope (JWTs can include scope claims; opaque tokens rely on server-side lookup).

14.1.3. Token Lifetime§

Tokens SHOULD have limited lifetimes. Recommended defaults:

Use Case	Recommended Lifetime
Interactive sessions	1 hour
Automated pipelines	8-24 hours
High-security environments	5-15 minutes

Servers MUST reject expired tokens with 401 Unauthorized. Callers that receive 401 SHOULD re-authenticate via their Token Establishment method.

14.1.4. Token Refresh§

For long-running operations, callers MAY need to refresh tokens before expiry. Refresh mechanisms depend on the Token Establishment method:

- **FDOKeyAuth**: Re-run the protocol to obtain a new token
- **OAuth/OIDC**: Use refresh tokens per OAuth specification
- **Portal**: Re-authenticate via the portal

For detailed token lifecycle guidance including expiry handling, scoping rules, and API gateway compatibility, see § 16.3 Token Lifecycle.

14.2. DPoP Tokens (Sender-Constrained)§

DPoP (Demonstrating Proof of Possession) tokens provide stronger security than bearer tokens by binding the token to a specific cryptographic key. Each request must include a signed proof demonstrating possession of the bound key.

14.2.1. Why DPoP?§

Bearer tokens are vulnerable to theft: if an attacker obtains the token (via log exposure, memory dump, or network interception), they can use it. DPoP mitigates this by requiring proof of key possession on every request. A stolen DPoP token is useless without the corresponding private key.

14.2.2. Request Format§

DPoP requests include two headers:

```
POST /api/v1/vouchers HTTP/1.1
Host: recipient.example.com
Authorization: DPoP eyJhbGciOiJFUzI1NiIs...
DPoP: eyJ0eXAiOiJkcG9wK2p3dCI6ImFsZyI6IkVMTmJuU2IiwiaWandr...
Content-Type: multipart/form-data
```

- **Authorization**: DPoP <token> — The DPoP-bound access token
- **DPoP**: <proof> — A signed JWT proving key possession

14.2.3. DPoP Proof Structure (per RFC9449)

The DPoP header contains a JWT with:

Claim	Description
typ	MUST be "dpop+jwt"
alg	Signature algorithm (ES256, ES384, RS256)
jwk	The public key (for signature verification)
jti	Unique identifier (replay protection)
htm	HTTP method (e.g., "POST")
htu	HTTP URI (e.g., "https://recipient.example.com/api/v1/vouchers")
iat	Issued-at timestamp
nonce	Server-provided nonce (if required)

14.2.4. Token Binding

DPoP tokens contain a cnf (confirmation) claim with a jkt (JWK Thumbprint) value:

```
{
  "sub": "owner-key-fingerprint",
  "exp": 1699999999,
  "cnf": {
    "jkt": "0Zc0C0RZNYy-DWpqq30jZyJGHTN0d2HglBV3uiguA4I"
  }
}
```

The server validates that the jkt in the token matches the thumbprint of the public key in the DPoP proof.

14.2.5. Server Validation

Servers validating DPoP requests MUST:

- 1. Verify the DPoP proof signature using the embedded jwk
- 2. Verify typ is "dpop+jwt"
- 3. Verify htm matches the request HTTP method
- 4. Verify htu matches the request URI
- 5. Verify iat is recent (within acceptable clock skew)
- 6. Verify jti has not been seen before (replay protection)
- 7. Verify the token's cnf.jkt matches the JWK Thumbprint of the proof's jwk
- 8. If server requires nonces, verify nonce matches a server-issued value

14.2.6. Nonce Refresh

Servers MAY require nonces for additional replay protection. When a server requires a nonce but the request lacks one (or has an invalid one), it responds:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP error="use_dpop_nonce"
DPoP-Nonce: eyJ7S_zG.eyJH0-Z.HX4w-7v
```

The caller retries with the provided nonce in the DPoP proof's nonce claim.

14.2.7. Obtaining DPoP Tokens§

DPoP tokens can be obtained via:

1. **FDOKeyAuth with DPoP:** Request a DPoP token during FDOKeyAuth by including `TokenPreferences` in the Hello message. See [§ 16.4.2 DPoP with FDOKeyAuth](#).
2. **DPoP Key Registration:** If you have a bearer token, upgrade to DPoP by registering a key via `POST {root}/auth/dpop-keys`. See [§ 16.4.3 DPoP Key Registration Endpoint](#).
3. **OAuth with DPoP:** Use OAuth 2.0 with DPoP extension per [\[RFC9449\]](#).

For additional DPoP details including server policy configuration and discovery mechanisms, see [§ 16.4 DPoP: Sender-Constrained Tokens](#).

14.3. mTLS Authentication§

Mutual TLS (mTLS) authenticates the caller via client certificate during the TLS handshake. When used alone, no `Authorization` header is needed — the TLS layer provides authentication.

mTLS is well-suited for point-to-point integrations, enterprise environments with existing PKI, and Custodian authentication. However, modern infrastructure often terminates TLS at load balancers or API gateways, breaking end-to-end mTLS. For high-security deployments, **combining mTLS with token authentication** provides defense-in-depth: mTLS secures the transport while tokens provide application-level identity that survives TLS termination.

For detailed mTLS guidance including certificate enrollment, server configuration, and combining mTLS with tokens, see [§ 16.5 mTLS Authentication](#).

15. Token Establishment§

This section describes how callers obtain tokens for use in Request Authentication. The primary method is FDOKeyAuth (defined in [§ 16 FDOKeyAuth Protocol](#)). This section covers alternative methods.

15.1. OAuth/OIDC Integration (Non-Normative)§

Organizations with existing OAuth 2.0 or OpenID Connect infrastructure can use their identity provider to issue tokens for Voucher Operations.

15.1.1. Token Exchange Pattern§

1. Caller authenticates to their organization's identity provider (IdP)
2. IdP issues an access token (Bearer or DPoP)
3. Caller presents the token to Voucher Operation endpoints
4. Server validates the token with the IdP (or via JWT signature verification)
5. Server maps the token's identity claims to an Owner Key or Custodian identity

15.1.2. Server Configuration§

The server must be configured to trust the IdP and map identities:

```
oauth_providers:
- issuer: "https://idp.example.com"
  jwks_uri: "https://idp.example.com/.well-known/jwks.json"
  identity_claim: "sub"
  identity_mappings:
    - claim_value: "user@example.com"
      owner_key_fingerprint: "sha256:af3b...c8d1"
    - claim_value: "reseller-service"
      custodian_id: "reseller-001"
```

15.1.3. DPoP with OAuth§

OAuth 2.0 supports DPoP via [\[RFC9449\]](#). When the IdP issues a DPoP-bound token, the caller must include DPoP proofs on each request, exactly as described in [§ 14.2 DPoP Tokens \(Sender-Constrained\)](#).

15.2. Portal/Out-of-Band Tokens (Non-Normative)§

Partner portals and manual processes can provision tokens without requiring FDOKeyAuth or OAuth infrastructure.

15.2.1. Partner Portal Pattern§

1. Custodian logs into a partner web portal
2. Portal authenticates the Custodian (username/password, SSO, etc.)
3. Portal issues a bearer token with appropriate scopes
4. Custodian uses the token for API calls

15.2.2. API Key Pattern§

For automated systems, long-lived API keys may be provisioned:

1. Administrator creates an API key in the server's management interface
2. API key is securely delivered to the caller (out-of-band)
3. Caller uses the API key as a bearer token

API keys SHOULD:

- Be scoped to specific operations and voucher sets
- Have expiration dates
- Be rotatable without service interruption
- Be auditable (who created, when used)

15.2.3. Security Considerations§

Out-of-band tokens lack the cryptographic proof-of-possession that FDOKeyAuth provides. They are appropriate for:

- Custodians (who have no Owner Key to prove)
- Low-security environments
- Bootstrap scenarios before FDOKeyAuth is available

For high-security deployments, prefer FDOKeyAuth or OAuth with DPoP.

16. FDOKeyAuth Protocol§

FDOKeyAuth is the FDO-specific token establishment protocol. It is a cryptographic challenge-response handshake in which the caller proves possession of an enrolled key (Owner Key or Delegate Key). Upon successful authentication, the server issues a Bearer or DPoP token.

16.1. Overview§

16.1.1. The Authentication Problem§

When a party connects to a voucher transfer endpoint, the server needs to answer **is this caller who they claim to be?** The caller's identity is bound to a specific cryptographic key exchanged during enrollment. The most natural proof is: **prove you hold the corresponding private key.**

The keys exchanged during enrollment serve double duty: they were originally provided for voucher signing and verification, and now they also serve as credentials for accessing Voucher Operation APIs.

16.1.2. TO2-Inspired Challenge-Response§

Simply claiming "I have this key" is not sufficient — the caller must *prove* possession of the private key without revealing it. FDOKeyAuth is modeled after the TO2 protocol in [\[FDO-Specification\]](#):

1. The **caller** presents the key it claims to hold and a fresh nonce
2. The **server** issues a signed challenge containing both nonces
3. The **caller** signs a response proving possession of the private key
4. The **server** verifies the signature and issues a token

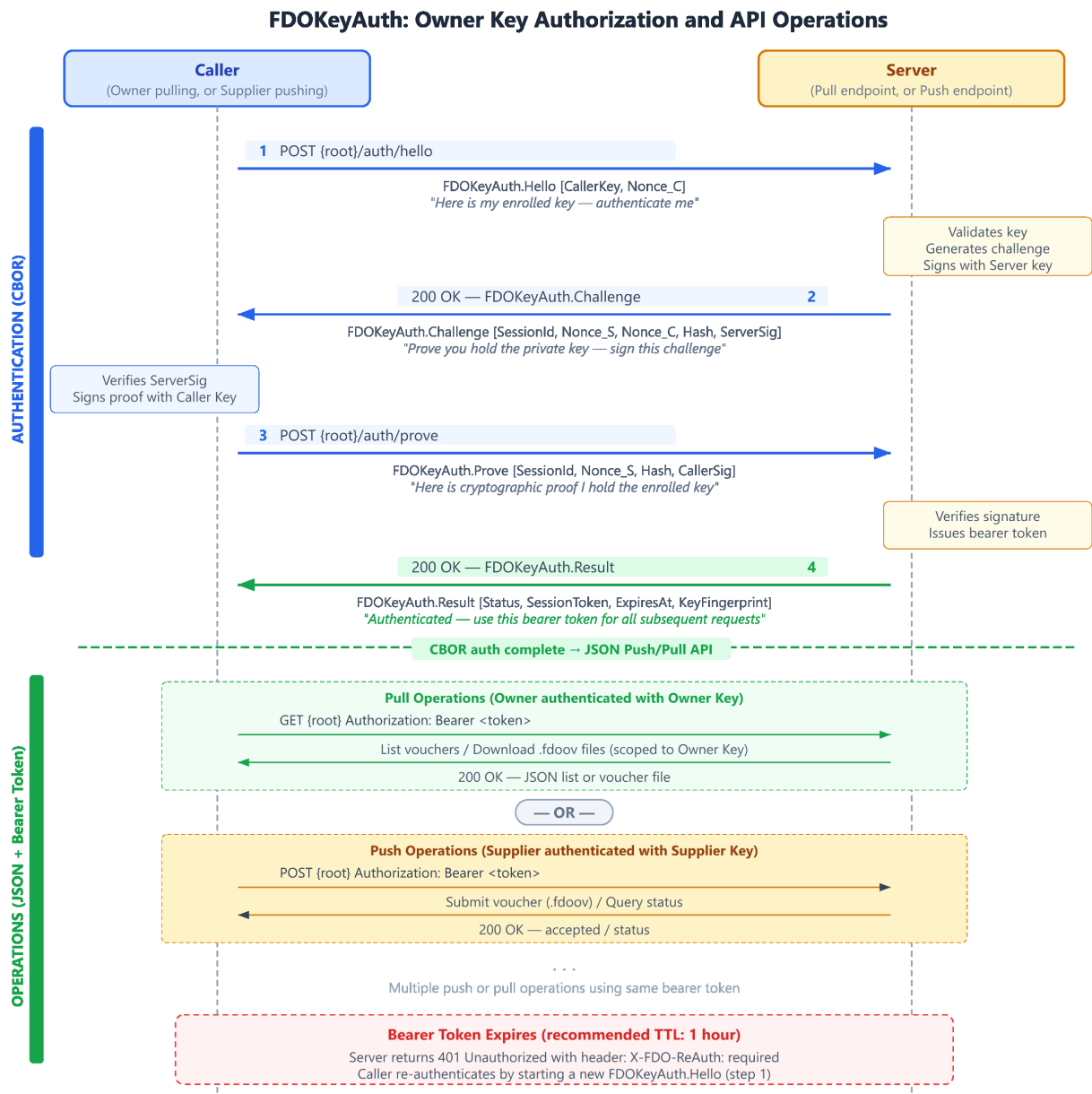


Figure 2 FDOKeyAuth Handshake and Subsequent Voucher Operations

16.1.3. Protocol Summary

Message	Direction	Purpose
FDOKeyAuth.Hello	Caller → Server	Present key and caller nonce
FDOKeyAuth.Challenge	Server → Caller	Server nonce, signed challenge
FDOKeyAuth.Prove	Caller → Server	Signed proof of key possession
FDOKeyAuth.Result	Server → Caller	Token (Bearer or DPoP)

16.2. Protocol Messages

This section defines the FDOKeyAuth cryptographic handshake. Upon successful authentication, the server issues a bearer token (the SessionToken) that the caller uses for all subsequent API requests.

All FDOKeyAuth messages are carried over HTTP as request and response bodies with Content-Type

application/cbor. Implementations MUST set Content-Type: application/cbor on all FDOKeyAuth requests and responses. Implementations MUST reject FDOKeyAuth messages that do not carry this Content-Type.

16.2.1. Protocol Messages

The protocol consists of three messages exchanged over two HTTP round trips:

Message	Direction	Content-Type	Purpose
FDOKeyAuth.Hello	Caller → Server	application/cbor	Caller presents key (w/ optional Delegate Chain) and a caller nonce
FDOKeyAuth.Challenge	Server → Caller	application/cbor	Server issues a signed challenge containing both nonces and a hash of the Hello message
FDOKeyAuth.Prove	Caller → Server	application/cbor	Caller signs a response proving key possession; Server returns a session token

16.2.2. Common Data Types

All data types in this section are defined in CDDL[RFC8610] and encoded as CBOR[RFC8949].

16.2.2.1. Nonces

Following the nonce conventions of[FDO-Specification], nonces are 16-byte (128-bit) random values. Each nonce is generated by one party and echoed back in a signed message by the other party to prove freshness.

```
;; CDDL
Nonce = bstr .size 16

NonceAuthCaller_Prep = Nonce    ;; generated by Caller, sent in FDOKeyAuth.Hello
NonceAuthServer_Prep = Nonce    ;; generated by Server, sent in FDOKeyAuth.Challenge

NonceAuthCaller      = Nonce    ;; echoed by Server in FDOKeyAuth.Challenge (signed)
NonceAuthServer      = Nonce    ;; echoed by Caller in FDOKeyAuth.Prove (signed)
```

Following [FDO-Specification] convention, the _Prep suffix indicates a nonce being transmitted for use in a future message. The suffix is removed when the nonce appears in its target signed message.

16.2.2.2. Hash Continuity

Each message after the first includes a hash of the preceding message’s CBOR-encoded body, providing transaction integrity. Because CBOR encoding is deterministic, both parties compute identical hashes over the same logical message content. This prevents an attacker from altering messages between protocol steps.

```
;; CDDL
Hash = [
    hashType: int,    ;; hash algorithm identifier per [[FDO-Specification]]
    hashValue: bstr   ;; hash output
]
```

Hash algorithms MUST be one of the algorithms defined in[FDO-Specification].

16.2.2.3. Public Keys and Certificate Chains

```
;; CDDL
PublicKey = COSE_Key          ;; as defined in [[RFC9052]]
CertChainOrNull = X5CHAIN / null ;; X.509 certificate chain per [[FDO-Specification]]
X5CHAIN = [ + bstr ]          ;; array of DER-encoded X.509 certificates,
                                ;; root first, leaf last
```

16.2.3. Message 1: FDOKeyAuth.Hello

16.2.3.1. POST {root}/auth/hello

The Caller initiates the authentication handshake by presenting its claimed key, an optional Delegate Chain, and a freshly generated nonce.

From Caller to Server.

16.2.3.2. Request Body (application/cbor)

```
;; CDDL
FDOKeyAuth.Hello = [
  CallerKey:          PublicKey,          ;; Caller's public key (COSE_Key)
  DelegateChain:      CertChainOrNull,    ;; optional Delegate X5CHAIN
  NonceAuthCaller_Prep,          ;; 16-byte random nonce
  ProtocolVersion:    uint,              ;; protocol version; current value: 1
  ? TokenPreferences:  TokenPrefs,        ;; token type preferences (see [[#auth-dpop]])
  * any                  ;; future extensions (see [[#cbor-extensibility]])
]

;; Token preferences for DPoP support (see [[#auth-dpop]])
TokenPrefs = {
  ? "token_types":    [ + tstr ],          ;; e.g., ["dpop", "bearer"] - preference order
  ? "dpop_jwk":       COSE_Key,            ;; DPoP public key (required if "dpop" requested)
}

  ? "dpop_alg":       tstr,                ;; DPoP signature algorithm: "ES256", "ES384", "
6"
  * tstr => any          ;; future extensions
}
```

Fields:

- **CallerKey** (REQUIRED): The public key the caller claims to hold, encoded as a COSE_Key[RFC9052]. For pull operations, this is the Owner Key (the key vouchers were signed over to). For push operations, this is the Supplier Key (the key the recipient enrolled via DID).
- **DelegateChain** (REQUIRED, may be null): An X.509 certificate chain encoded using the X5CHAIN format defined in [FDO-Specification] section 3.3.4. The voucher-transfer delegation requirements for this chain are defined in this specification. When non-null, indicates that the Caller is a Delegate acting on behalf of the key owner. The chain MUST be rooted by the key specified in CallerKey and MUST terminate with the Delegate's leaf certificate. Set to CBOR null when Delegation is not in use.
- **NonceAuthCaller_Prep** (REQUIRED): A freshly generated 16-byte random nonce. The Server MUST echo this nonce in its signed challenge response.
- **ProtocolVersion** (REQUIRED): Protocol version as an unsigned integer. Current version:1.
- **TokenPreferences** (OPTIONAL): A map specifying the Caller's preferred token types and DPoP key material. See § 16.4 DPoP: Sender-Constrained Tokens for details. When omitted, the Server issues a bearer token (legacy behavior).

TokenPrefs Fields:

- token_types (OPTIONAL): An ordered list of acceptable token types. Valid values: "bearer", "dpop". The first entry is most preferred. If omitted, defaults to ["bearer"].
- dpop_jwk (OPTIONAL): The public key the Caller will use for DPoP proofs, encoded as a COSE_Key. MUST be provided when "dpop" is in token_types. The Server computes the JWK Thumbprint[RFC7638] of this key for token binding.
- dpop_alg (OPTIONAL): The signature algorithm the Caller will use for DPoP proofs. Valid values: "ES256", "ES384", "RS256". SHOULD be provided when "dpop" is in token_types.

16.2.3.3. Server Processing

1. The Server MUST validate that CallerKey is a well-formed COSE_Key in a supported key type.
2. The Server MUST verify that CallerKey matches an enrolled key. For pull endpoints, the Server SHOULD check whether any vouchers exist that are signed over to the presented key. For push endpoints, the Server SHOULD verify the key matches an enrolled Supplier DID. The Server MAY reject the request early with HTTP 404 Not Found or MAY proceed with the challenge to avoid revealing information (see Security Considerations).
3. If DelegateChain is non-null, the Server MUST verify:
 - The chain is well-formed and all certificates are valid (not expired, not revoked)
 - The root of the chain is signed by the key in CallerKey
 - The chain includes the appropriate ECU permission (see § 16.7.2 Extended Key Usage Permissions)
 - The Server MUST apply standard X.509 chain validation rules
4. If TokenPreferences is present and contains "dpop" in token_types:
 - The Server MUST validate that dpop_jwk is present and well-formed
 - The Server MUST validate that dpop_alg (if present) is a supported algorithm
 - The Server computes the JWK Thumbprint[RFC7638] of the dpop_jwk (converting from COSE_Key to JWK format internally)
 - The Server stores the DPoP key material with the session for later token issuance
 - If the Server does not support DPoP, it ignores these fields and issues a bearer token
5. The Server generates a fresh nonce (NonceAuthServer_Prep) and a session identifier.
6. The Server computes HashAuthHello = Hash(CBOR-encoded FDOKeyAuth.Hello request body). Because CBOR encoding is deterministic, this hash is unambiguous.
7. The Server stores the session state: {session_id, CallerKey, DelegateChain, NonceAuthCaller_Prep, NonceAuthServer_Prep, HashAuthHello, TokenPreferences, created_at}.

16.2.3.4. Response: FDOKeyAuth.Challenge (application/cbor)

Success (HTTP 200 OK):

```

;; CDDL
FD0KeyAuth.Challenge = [
  SessionId:          bstr,          ;; opaque session identifier
  NonceAuthServer_Prep,          ;; 16-byte random nonce
  NonceAuthCaller,          ;; echo of Caller's nonce from Hello
  HashAuthHello:      Hash,          ;; hash of FD0KeyAuth.Hello CBOR body
  ServerSignature:     CoSeSignature, ;; COSE_Sign1 over challenge payload
  ServerInfo:          ServerInfoOrNull, ;; optional metadata
  * any                ;; future extensions (see [[#cbor-extensibility]])
]

ServerInfoOrNull = ServerInfo / null
ServerInfo = {
  ? "server_id":      tstr,          ;; Server identifier (e.g., hostname)
  ? "voucher_count":  uint,          ;; approximate vouchers available (pull only)
  ? "algorithms":     [ + int ],     ;; COSE algorithm identifiers Server can verify
  ? "dpop_supported": bool,          ;; true if Server supports DPoP tokens
  ? "dpop_algs":      [ + tstr ],    ;; supported DPoP algorithms: "ES256", "ES384",
56"
  ? "dpop_nonce":     tstr,          ;; Server nonce for DPoP proofs (if required)
  * tstr => any                ;; future extensions (see [[#cbor-extensibility]])
}

```

Fields:

- **SessionId (REQUIRED):** An opaque, server-generated session identifier (bstr). The Caller **MUST** include this in the subsequent FD0KeyAuth.Prove message. Session identifiers **SHOULD** expire after a short period (e.g., 60 seconds).
- **NonceAuthServer_Prep (REQUIRED):** A freshly generated 16-byte random nonce. The Caller **MUST** echo this nonce in its signed proof.
- **NonceAuthCaller (REQUIRED):** The Caller's nonce from the Hello message, echoed back.
- **HashAuthHello (REQUIRED):** Hash of the CBOR-encoded FD0KeyAuth.Hello request body, providing message continuity.
- **ServerSignature (REQUIRED):** A COSE_Sign1 [\[RFC9052\]](#) signature over the challenge payload (see below). This allows the Caller to verify the Server is a live participant and that the challenge has not been tampered with.
- **ServerInfo (REQUIRED, may be null):** Informational metadata about the Server. Set to CBORnull when not provided. The voucher_count field, if present, indicates the approximate number of vouchers available for the presented key (relevant for pull operations). The algorithms field lists COSE algorithm identifiers (e.g., -7 for ES256, -35 for ES384) that the Server can verify. The DPoP-related fields (dpop_supported, dpop_algs, dpop_nonce) advertise the Server's DPoP capabilities; see [§ 16.4 DPoP: Sender-Constrained Tokens](#) for details.

Server Signature Payload:

The ServerSignature is a COSE_Sign1 [\[RFC9052\]](#) signature over a well-defined CBOR structure:

```

;; CDDL
FD0KeyAuthChallengeSignedPayload = [
  "FD0KeyAuth.Challenge",          ;; protocol message type tag (tstr)
  NonceAuthCaller,                  ;; echo of Caller's nonce
  NonceAuthServer_Prep,             ;; Server's nonce for next message
  HashAuthHello: Hash,              ;; hash of FD0KeyAuth.Hello CBOR body
  CallerKey: PublicKey,             ;; the key being authenticated
  * any                             ;; future extensions (see [[#cbor-extensibility]])
]

```

NOTE: The explicit "FD0KeyAuth.Challenge" text string tag ensures this signed payload is structurally distinct from any other FDO protocol message. Because the payload is a CBOR array with a fixed type tag, it cannot be confused with an 0VEntry, T02Prove0VHdrPayload, or any other COSE-signed structure. This is a critical defense against signing oracle attacks.

Error Responses:

Error responses MAY be returned as `application/cbor` or `application/json` at the Server's discretion. Implementations SHOULD check the Content-Type header to determine the encoding.

- 400 Bad Request: Malformed CBOR, invalid key format, or invalid delegate chain
- 404 Not Found: Key not recognized or no matching resources (OPTIONAL; see Security Considerations)
- 429 Too Many Requests: Rate limit exceeded
- 500 Internal Server Error: Server error

16.2.4. Message 2: FDOKeyAuth.Prove^S

16.2.4.1. POST {root}/auth/prove^S

From Caller to Server.

The Caller proves possession of the claimed key (or Delegate Key) by signing a well-defined response that includes the Server's challenge nonce.

16.2.4.2. Request Body (`application/cbor`)^S

```
;; CDDL
FD0KeyAuth.Prove = [
  SessionId:          bstr,          ;; session identifier from FD0KeyAuth.Challenge
  NonceAuthServer,    ;; echo of Server's nonce from Challenge
  HashAuthChallenge:  Hash,          ;; hash of FD0KeyAuth.Challenge CBOR body
  CallerSignature:    CoseSignature, ;; COSE_Sign1 proving key possession
  * any              ;; future extensions (see [[#cbor-extensibility]
)
]
```

Fields:

- SessionId (REQUIRED): The session identifier from the `FD0KeyAuth.Challenge` response.
- NonceAuthServer (REQUIRED): The Server's nonce from the Challenge message, echoed back.
- HashAuthChallenge (REQUIRED): Hash of the CBOR-encoded `FD0KeyAuth.Challenge` response body, providing message continuity from the previous step.
- CallerSignature (REQUIRED): A COSE_Sign1 [\[RFC9052\]](#) signature proving key possession (see below).

Caller Signature Payload:

The CallerSignature is a COSE_Sign1 signature over a well-defined CBOR structure:

```
;; CDDL
FD0KeyAuthProveSignedPayload = [
  "FD0KeyAuth.Prove",          ;; protocol message type tag (tstr)
  NonceAuthServer,             ;; echo of Server's nonce (proves freshness)
  NonceAuthCaller,             ;; Caller's own nonce (proves continuity)
  HashAuthChallenge: Hash,     ;; hash of FD0KeyAuth.Challenge CBOR body
  CallerKey: PublicKey,        ;; the key being authenticated
  * any                        ;; future extensions (see [[#cbor-extensibility]])
]
```

Signing Key:

- If no Delegate Chain was presented in `FD0KeyAuth.Hello`: The Caller signs with the **private key** corresponding to the `CallerKey` presented in the Hello message.
- If a Delegate Chain was presented: The Caller signs with the **Delegate private key** corresponding to the leaf certificate of the Delegate Chain.

NOTE: The explicit `"FD0KeyAuth.Prove"` text string tag ensures this signed payload is structurally distinct from any other FDO protocol message. In particular, this CBOR array structure is intentionally distinct from an `OVEntry` or `T02ProveOVHdrPayload`, preventing a malicious Server from using this protocol as a signing oracle to forge Ownership Voucher extensions or TO2 messages.

16.2.4.3. Server Verification

The Server MUST perform the following verification steps:

1. **Session Lookup:** Retrieve the session state for `SessionId`. If the session has expired or does not exist, return HTTP 401 Unauthorized.
2. **Hash Continuity:** Verify that `HashAuthChallenge` matches the hash of the CBOR-encoded `FD0KeyAuth.Challenge` response body that the Server sent. This confirms the Caller received the unmodified challenge.
3. **Nonce Verification:** Verify that `NonceAuthServer` matches the `NonceAuthServer_Prep` stored in the session. This proves the signature is fresh and not replayed.
4. **Signature Verification:**
 - If no Delegate Chain: Verify `CallerSignature` using the `CallerKey` from the session.
 - If Delegate Chain is present: Verify `CallerSignature` using the public key from the **leaf certificate** of the Delegate Chain. Then verify the Delegate Chain is rooted by `CallerKey` and contains the required ECU permission.
5. **Payload Verification:** Decode the `COSE_Sign1` payload and verify:
 - The message type tag is `"FD0KeyAuth.Prove"`
 - `NonceAuthServer` matches the stored nonce
 - `NonceAuthCaller` matches the Caller's original nonce from the Hello message
 - `CallerKey` matches the `CallerKey` from the session
6. **Session Cleanup:** Invalidate the session to prevent reuse.

16.2.4.4. Response: Authentication Result (*application/cbor*)

Success (HTTP 200 OK):

```

;; CDDL
FDOKeyAuth.Result = [
    Status:          tstr,                ;; "authenticated"
    SessionToken:    tstr / null,         ;; bearer token (null when TokenInfo present)
    TokenExpiresAt:  uint,                ;; expiration as Unix timestamp (seconds)
    KeyFingerprint:  bstr,                ;; SHA-256 hash of the authenticated key
    ? VoucherCount:  uint,                ;; approximate vouchers available (pull only)
    ? TokenInfo:     TokenInfo,           ;; structured token info (see [[#auth-dpop]])
    * any            ;; future extensions (see [[#cbor-extensibility]])
]

;; Structured token information for DPoP support (see [[#auth-dpop]])
TokenInfo = {
    ? "bearer": BearerTokenInfo,         ;; bearer token option
    ? "dpop":   DPoPTokenInfo,           ;; DPoP token option
    * tstr => any                         ;; future extensions
}

BearerTokenInfo = {
    "token":      tstr,                  ;; opaque bearer token
    "expires_at": uint                  ;; expiration timestamp
}

DPoPTokenInfo = {
    "token":      tstr,                  ;; JWT with cnf.jkt claim
    "expires_at": uint,                  ;; expiration timestamp
    "jkt":        tstr,                  ;; JWK thumbprint of bound key
    "alg":        tstr,                  ;; algorithm for DPoP proofs
    ? "nonce":    tstr,                  ;; server nonce (if required)
    ? "nonce_ttl": uint                  ;; nonce validity in seconds
}

```

Fields:

- Status (REQUIRED): SHALL be "authenticated" on success.
- SessionToken (REQUIRED, may be null): An opaque bearer token (tstr) for backward compatibility. When TokenInfo is present, this field SHOULD be null and Callers SHOULD use TokenInfo instead. When TokenInfo is absent (legacy Server), this field contains the bearer token to use asAuthorization: Bearer <SessionToken>.
- TokenExpiresAt (REQUIRED): Token expiration as a Unix timestamp (seconds since epoch). See [§ 16.3 Token Lifecycle](#) for recommended token lifetimes and re-authentication behavior.
- KeyFingerprint (REQUIRED): SHA-256 hash of the CBOR-encoded authenticated key (bstr), allowing the Caller to confirm which key was authenticated. Implementations MUST normalize the key to its canonical form before computing the fingerprint: extract the raw public key material, re-encode it as a protocol.PublicKey with pkEnc = X509 (DER encoding), then CBOR-encode and hash. This ensures the fingerprint depends only on the key material, not on the wire encoding (X509, X5CHAIN, or COSEKEY) used during transmission.
- VoucherCount (OPTIONAL): The number of vouchers currently available for the authenticated key (relevant for pull operations).
- TokenInfo (OPTIONAL): Structured token information containing bearer and/or DPoP tokens. See [§ 16.4 DPoP: Sender-Constrained Tokens](#) for details. When present, Callers SHOULD prefer this over the legacy SessionToken field.

TokenInfo Fields:

- bearer (OPTIONAL): Bearer token information. Contains token (the opaque bearer token) and expires_at (expiration timestamp).

- **dpop (OPTIONAL):** DPoP token information. Contains `token` (a JWT with `cnf.jkt` claim binding it to the DPoP key), `expires_at`, `jkt` (the JWK Thumbprint for verification), `alg` (the algorithm for DPoP proofs), and optionally `nonce` and `nonce_ttl` if the Server requires nonces for DPoP proofs.

Token Issuance Logic:

The Server issues tokens based on the Caller's `TokenPreferences` and the Server's policy:

Server Policy	Caller Requests	Result
<code>bearer_only</code>	<code>["dpop", "bearer"]</code>	<code>TokenInfo.bearer</code> only
<code>bearer_only</code>	<code>["bearer"]</code>	<code>TokenInfo.bearer</code> only
<code>dpop_required</code>	<code>["dpop", "bearer"]</code>	<code>TokenInfo.dpop</code> only
<code>dpop_required</code>	<code>["bearer"]</code>	Error 403 Forbidden
<code>flexible</code>	<code>["dpop", "bearer"]</code>	Both <code>TokenInfo.bearer</code> and <code>TokenInfo.dpop</code>
<code>flexible</code>	<code>["bearer"]</code>	<code>TokenInfo.bearer</code> only
<code>flexible</code>	<code>["dpop"]</code>	<code>TokenInfo.dpop</code> only
(any)	(no <code>TokenPreferences</code>)	<code>SessionToken</code> field (legacy)

Error Responses:

Error responses MAY be returned as `application/cbor` or `application/json` at the Server's discretion.

- **401 Unauthorized:** Signature verification failed, nonce mismatch, session expired, or session not found
- **403 Forbidden:** Delegate Chain does not include required permissions
- **400 Bad Request:** Malformed CBOR or invalid hash continuity
- **500 Internal Server Error:** Server error

16.3. Token Lifecycle

After successful authentication, the Caller uses the issued token for all subsequent Push or Pull API requests. The token type determines the authorization header format:

- **Bearer token:** Authorization: Bearer <token>
- **DPoP token:** Authorization: DPoP <token> with accompanying DPoP: <proof> header (see [§ 16.4 DPoP: Sender-Constrained Tokens](#))

A single token authorizes multiple operations within the token's lifetime — listing vouchers, downloading voucher files, submitting vouchers, querying status, etc.

16.3.1. Token Lifetime

Implementations SHOULD issue tokens with a lifetime of **1 hour** (3600 seconds). This balances security (limiting the window of token misuse) against usability (allowing a reasonable batch of operations without re-authenticating).

Implementations MAY use shorter lifetimes (e.g., 15 minutes) for high-security deployments or longer lifetimes (e.g., 4 hours) for environments where re-authentication overhead is a concern. The `TokenExpiresAt` field in `FDKeyAuth.Result` (or `expires_at` in `TokenInfo`) communicates the exact expiration to the Caller.

NOTE: DPOP tokens provide additional security even with longer lifetimes, since stolen tokens cannot be used without the private key. Deployments using DPOP MAY consider longer token lifetimes to reduce re-authentication overhead.

16.3.2. Token Expiry and Re-Authentication§

When a token expires, the Server MUST reject subsequent requests with:

- ****HTTP 401 Unauthorized****
- ****Header X-FDO-ReAuth: required**** — signals the Caller that re-authentication (a new FDOKeyAuth handshake) is needed, as opposed to a permanent authorization failure

The Caller SHOULD handle this gracefully by initiating a newFDOKeyAuth.Hello handshake to obtain a fresh bearer token, then retrying the failed request. For pull operations, implementations SHOULD NOT require the Caller to restart any pagination sequence from the beginning — continuation tokens obtained before re-authentication SHOULD remain valid if they have not independently expired.

16.3.3. Token Scoping§

All authentication methods — bearer tokens, DPOP tokens, and mTLS client certificates — MUST be scoped to the **identity** of the authenticated caller. The server maintains associations between caller identities and vouchers — these associations are established through business logic (purchase orders, account records, enrollment) and are implementation-specific.

Callers fall into two categories:

- **Owners:** Entities that possess an FDO Owner Key. They authenticate via FDOKeyAuth (proving key possession) and can be sign-over targets.
- **Custodians:** Entities with business authority but no Owner Key. They authenticate via alternative mechanisms (mTLS, OAuth, portal tokens) and can manage vouchers but cannot be sign-over targets.

When a token is issued via FDOKeyAuth, the caller is an Owner and their identity is bound to a specific cryptographic key. When authentication is via mTLS or alternative token sources (OAuth, API key, partner portal), the caller may be an Owner or a Custodian — the server maps the authenticated identity to a set of vouchers and scopes API responses accordingly.

The distinction matters for one specific operation: **voucher sign-over**. Signing a voucher over to a party requires that party's public key — it is a cryptographic operation, not an access control decision. A Custodian (e.g., a reseller using mTLS or a portal API token) can list, download, query status, and assign vouchers associated with their identity. They simply cannot be a sign-over target, because there is no key to sign to.

Operation	Requires identity	Requires Owner Key	Custodian can perform?
Query voucher status	Yes (bearer, DPOP, or mTLS)	No	Yes
List vouchers	Yes (bearer, DPOP, or mTLS)	No	Yes
Download voucher file	Yes (bearer, DPOP, or mTLS)	No	Yes
Submit voucher (push)	Yes (bearer, DPOP, or mTLS)	No (but voucher signature is verified)	Yes
Assign voucher (see § 12 Voucher Assignment)	Yes (bearer, DPOP, or mTLS)	No	Yes

Have vouchers signed over to caller Operation	— Requires identity	Yes Requires Owner Key	Custodian can perform?
---	-------------------------------	----------------------------------	-------------------------------

For pull operations where sign-over is expected, the Server MUST verify that the authenticated caller is an Owner with a corresponding Owner Key. For all other operations, the Server MUST scope responses to vouchers associated with the caller's identity, regardless of whether the caller is an Owner or Custodian, and regardless of authentication method (FDOKeyAuth, mTLS, or alternative tokens).

16.3.4. API Gateway Compatibility

The bearer token is carried in the standard `HTTPAuthorization: Bearer <token>` header, which is the conventional location inspected by API gateways, reverse proxies, and web application firewalls (WAFs). This means:

- **Standard API gateways** (Kong, Envoy, AWS API Gateway, Azure APIM, etc.) can inspect, validate, or forward the bearer token without custom plugin development.
- **Rate limiting and access control** policies in API gateways can be applied based on the bearer token or its decoded contents.
- **The token format is opaque** to this specification. Implementations MAY issue tokens in any format (e.g., JWT [\[RFC7519\]](#), opaque random strings, signed cookies). Implementations that wish to maximize API gateway compatibility SHOULD consider issuing JWTs with standard claims (sub for key fingerprint, exp for expiry, iss for server identity, aud for the service root URL), as most API gateways have built-in JWT validation support.

This design ensures that the FDOKeyAuth protocol integrates naturally with existing API infrastructure. The authentication handshake is the "hard part" — once the bearer token is issued, the rest of the protocol looks like any standard REST API with bearer token authentication.

16.3.5. Alternative Token Sources (Non-Normative)

This specification defines the FDOKeyAuth handshake as the normative way to obtain a bearer token using enrolled keys. However, implementations MAY accept bearer tokens obtained through other means, provided the token satisfies the same scoping and lifetime requirements defined above. For example:

- An organization's **existing Identity Provider** (OAuth2, OIDC) could issue tokens that the Push or Pull API accepts
- A **pre-shared token** could be provisioned out-of-band during enrollment (see [Appendix A: Optional Security Layers \(Informative\)](#) for token format guidance)
- A **hardware security module** or token service could issue tokens via a proprietary protocol
- A **partner portal or ordering system** could issue tokens to resellers or intermediaries who may not have FDO-specific credentials (see [§ 12 Voucher Assignment](#) for this use case)

When accepting tokens from alternative sources, the Server MUST still enforce identity-scoped authorization: the token MUST identify the caller, and the Server MUST scope API responses to vouchers associated with that caller's identity (see [§ 16.3.3 Token Scoping](#)). Note that tokens from alternative sources may identify a caller who has no FDO owner key — such callers can perform identity-scoped operations (status queries, assignment) but cannot be sign-over targets.

This flexibility means that all APIs in this specification are **token-agnostic** — they care only that a valid `Authorization: Bearer <token>` or `Authorization: DPoP <token>` header is present. How the token was obtained is an implementation concern. The FDOKeyAuth protocol provides a standard, interoperable way to obtain such a token using only the keys from enrollment, but deployments with existing identity infrastructure MAY substitute their own token issuance mechanism.

16.3.5.1. DPoP with Alternative Token Sources

Alternative token sources (OAuth2, OIDC, etc.) MAY also issue DPoP-bound tokens. RFC9449 defines how OAuth2 authorization servers issue DPoP tokens — the client includes a DPoP header in the token request, and the server binds the issued token to the key in that proof.

When using DPoP tokens from an alternative source:

1. **Native RFC9449 support:** If the token source natively supports RFC9449, the DPoP key is registered during the OAuth flow itself. The FDO server simply validates the `cnf.jkt` claim against the DPoP proof header — no separate registration needed.
2. **DPoP key registration endpoint:** If the token source issues bearer tokens only, the caller can upgrade to DPoP by registering a key via `POST {root}/auth/dpop-keys` (see [§ 16.4.3 DPoP Key Registration Endpoint](#)). The server issues a new DPoP-bound token that the caller can use for subsequent requests.

This design ensures that DPoP is available regardless of how the initial token was obtained — FDOKeyAuth, OAuth2, OIDC, or portal.

16.4. DPoP: Sender-Constrained Tokens

This section defines optional support for DPoP (Demonstrating Proof of Possession) [RFC9449], which binds tokens to a specific cryptographic key. DPoP tokens provide stronger security than bearer tokens by requiring the caller to prove possession of a private key on every API request.

16.4.1. Overview

Bearer tokens are vulnerable to theft: if an attacker obtains a bearer token (from logs, memory, or network interception), they can use it directly. DPoP mitigates this by binding the token to a key pair held by the legitimate caller. Each API request must include a signed proof demonstrating possession of the bound key.

Key benefits of DPoP:

- **Token theft mitigation:** Stolen tokens are useless without the private key
- **Replay protection:** Each proof is bound to a specific HTTP method and URI
- **Hardware binding pathway:** DPoP keys can be stored in TPMs or secure enclaves
- **Standard protocol:** RFC9449 libraries are widely available

When to use DPoP:

- High-security deployments where token theft is a significant concern
- Environments where tokens may be logged or cached by intermediaries
- When hardware-bound credentials are desired (future enhancement)

When bearer tokens suffice:

- Internal deployments with strong network security
- Short-lived tokens with frequent re-authentication
- Simple integrations where DPoP library overhead is undesirable

16.4.2. DPoP with FDOKeyAuth

When authenticating via FDOKeyAuth, the Caller requests DPoP tokens by including `TokenPreferences` in the Hello message:

```

; Example: Caller requests DPoP with ES256, falling back to bearer
TokenPreferences = {
    "token_types": ["dpop", "bearer"],
    "dpop_jwk": <C0SE_Key>,
    "dpop_alg": "ES256"
}

```

The Server responds with `TokenInfo` containing the appropriate tokens based on its policy (see [§ 16.2.4 Message 2: FDOKeyAuth.Prove](#) for the full response format).

16.4.3. DPoP Key Registration Endpoint

For callers who obtain tokens via OAuth2, OIDC, or portal (skipping FDOKeyAuth), a dedicated endpoint allows DPOP key registration.

16.4.3.1. POST {root}/auth/dpop-keys

Registers a DPoP public key for the authenticated caller.

Request:

```
POST /fdo/v1/vouchers/auth/dpop-keys HTTP/1.1
Host: holder.example.com
Authorization: Bearer <existing_token>
Content-Type: application/json
```

```
{
  "jwk": {
    "kty": "EC",
    "crv": "P-256",
    "x": "WbbXwISbIHkjPwgKFUVHMQ...",
    "y": "5Qd2Rq3bKyeFNCgt0wPY3A..."
  },
  "alg": "ES256"
}
```

Response (201 Created):

```
{
  "status": "registered",
  "jkt": "0Zc0C0RZNYy-DWpqQ30jZyJGHTN0d2HglBV3uiguA4I",
  "expires_at": 1640995200,
  "dpop_token": "eyJhbGciOiJIUzI1NiIs... ",
  "nonce": "server-nonce-value"
}
```

Fields:

- **jwk (REQUIRED):** The DPoP public key in JWK format[\[RFC7638\]](#).
- **alg (REQUIRED):** The signature algorithm for DPoP proofs. Valid values: "ES256", "ES384", "RS256".

Response Fields:

- status: "registered" on success.
- jkt: The JWK Thumbprint [\[RFC7638\]](#) of the registered key.
- expires_at: Expiration timestamp for the DPoP key registration.

- `dpop_token`: A new DPoP-bound token (with `cnf.jkt` claim) that the caller can use immediately.
- `nonce` (OPTIONAL): Server nonce for DPoP proofs, if the server requires nonces.

Error Responses:

- 400 Bad Request: Invalid JWK format or unsupported algorithm
- 401 Unauthorized: Invalid or expired bearer token
- 409 Conflict: A DPoP key is already registered for this identity

16.4.3.2. GET {root}/auth/dpop-keys

Returns information about the registered DPoP key for the authenticated caller.

Request:

```
GET /fdo/v1/vouchers/auth/dpop-keys HTTP/1.1
Host: holder.example.com
Authorization: Bearer <token>
```

Response (200 OK):

```
{
  "registered": true,
  "jkt": "0ZcOCORZNYy-DWpqq30jZyJGHTN0d2HglBV3uiguA4I",
  "alg": "ES256",
  "registered_at": 1640908800,
  "expires_at": 1640995200
}
```

Response (200 OK, no key registered):

```
{
  "registered": false
}
```

16.4.3.3. DELETE {root}/auth/dpop-keys

Removes the registered DPoP key for the authenticated caller.

Request:

```
DELETE /fdo/v1/vouchers/auth/dpop-keys HTTP/1.1
Host: holder.example.com
Authorization: Bearer <token>
```

Response: 204 No Content on success.

16.4.4. Using DPoP Tokens in API Requests

When using a DPoP token, the caller includes two headers:

1. Authorization: DPoP <token> — the DPoP-bound access token

2. DPoP: <proof> — a signed JWT proving key possession

Example request:

```
GET /fdo/v1/vouchers HTTP/1.1
Host: holder.example.com
Authorization: DPoP eyJhbGciOiJIJFuzI1NiIsInR5cCI6ImlF0K2p3dCJ9...
DPoP: eyJ0eXAiOiJKcG9wK2p3dCIsImFsZyI6IkdVTMjU2IiwiaWandrIjp7Imt0eSI6IkVDIi4uLn19...
```

DPoP Proof Structure (per RFC9449):

The DPoP header contains a JWT with:

- **Header:**
 - typ: "dpop+jwt"
 - alg: The signature algorithm (e.g., "ES256")
 - jwk: The public key (for signature verification)
- **Payload:**
 - jti: Unique identifier (for replay protection)
 - htm: HTTP method (e.g., "GET", "POST")
 - htu: HTTP URI (e.g., "https://holder.example.com/fdo/v1/vouchers")
 - iat: Issued-at timestamp
 - nonce (OPTIONAL): Server-provided nonce

Example DPoP proof payload:

```
{
  "jti": "e1j3V_bKic8-LAEB_lccDA",
  "htm": "GET",
  "htu": "https://holder.example.com/fdo/v1/vouchers",
  "iat": 1640908800,
  "nonce": "server-provided-nonce"
}
```

16.4.5. Server Validation of DPoP Requests

When a server receives a request with `Authorization: DPoP`:

1. **Extract the DPoP proof** from the DPoP header
2. **Validate the proof JWT:**
 - Verify the signature using the embedded jwk
 - Verify typ is "dpop+jwt"
 - Verify htm matches the request method
 - Verify htu matches the request URI
 - Verify iat is recent (within acceptable clock skew, e.g., 60 seconds)
 - Verify jti has not been seen before (replay protection)
 - If server requires nonces: verify nonce matches an issued nonce
3. **Compute the JWK Thumbprint** of the key in the proof
4. **Validate the access token:**

- Verify the token signature and standard claims
- Extract the `cnf.jkt` claim
- Verify `cnf.jkt` matches the computed thumbprint from step 3

5. **Proceed with the request** if all validations pass

Error responses:

- 401 Unauthorized with WWW-Authenticate: DPoP error="invalid_dpop_proof" — proof validation failed
- 401 Unauthorized with WWW-Authenticate: DPoP error="use_dpop_nonce" and DPoP-Nonce: <new_nonce> — nonce required or expired

16.4.6. DPoP Nonce Refresh

When a server requires nonces and the client's nonce is missing or expired:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP error="use_dpop_nonce"
DPoP-Nonce: eyJ7S_zG.eyJH0-Z.HX4w-7v
```

The client **MUST** retry the request with the new nonce in the DPoP proof's nonce claim.

16.4.7. Server DPoP Policy

Servers configure their DPoP policy based on security requirements:

****bearer_only**** (Legacy/Simple)

- Server only accepts bearer tokens
- DPoP requests are rejected or the DPoP proof is ignored
- Simplest deployment; no DPoP infrastructure needed

****flexible**** (Recommended for Transition)

- Server accepts both bearer and DPoP tokens
- Clients choose their security level
- Enables gradual migration to DPoP

****dpop_required**** (High Security)

- Server only accepts DPoP tokens
- Bearer-only clients are rejected with 401 Unauthorized
- Maximum security; requires all clients to support DPoP

Servers **MAY** apply different policies to different operations:

```
# Example server configuration
dpop:
  default_policy: "flexible"
  operations:
    voucher_download: "flexible"      # Reading is lower risk
    voucher_submit: "dpop_required"  # Writing requires proof-of-possession
    voucher_assign: "dpop_required"  # Assignment is sensitive
```

16.4.8. HTTP Header Discovery

Servers advertise DPoP support via HTTP headers:

On any API response (informational):

```
HTTP/1.1 200 OK
DPoP-Supported: true
DPoP-Algs: ES256, ES384
```

On 401 responses when DPoP is required:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP algs="ES256 ES384"
DPoP-Nonce: eyJ7S_zG.eyJH0-Z.HX4w-7v
```

16.4.9. DID Document Discovery

For servers discovered via DID, the service entry MAY include DPoP capability:

```
{
  "id": "did:web:holder.example.com#voucher-service",
  "type": "FDOVoucherHolder",
  "serviceEndpoint": {
    "voucherHolderURL": "https://holder.example.com/fdo/v1/vouchers",
    "dpopSupported": true,
    "dpopAlgorithms": ["ES256", "ES384"],
    "dpopNonceRequired": false
  }
}
```

16.4.10. COSE_Key to JWK Conversion

When a DPoP key is provided as a COSE_Key in FDOKeyAuth.Hello, the server converts it to JWK format for thumbprint computation:

1. Extract the key type and curve parameters from the COSE_Key
2. Convert coordinates from CBOR bstr to base64url encoding
3. Construct the JWK JSON object with required members in lexicographic order
4. Compute the SHA-256 hash of the UTF-8 encoding
5. Base64url-encode the hash to produce the JWK Thumbprint

Example conversion (EC P-256 key):

COSE_Key parameters:

- kty (1): EC2 (2)
- crv (-1): P-256 (1)
- x (-2): <32 bytes>
- y (-3): <32 bytes>

Resulting JWK (normalized for thumbprint):

```
{"crv": "P-256", "kty": "EC", "x": "<base64url>", "y": "<base64url>"}
```

16.5. mTLS Authentication

Implementations MAY support mutual TLS (mTLS) as an alternative authentication mechanism. When mTLS is used, the client presents a certificate during the TLS handshake, and the server identifies the caller based on that certificate without requiring an Authorization header.

16.5.1. Overview

mTLS provides authentication at the transport layer rather than the application layer. This has several implications:

Advantages:

- **Strongest binding:** The client's identity is proven by possession of a private key that never leaves the client, verified during the TLS handshake itself
- **No token to steal:** There is no bearer token that could be logged, cached, or intercepted
- **Continuous authentication:** Every byte of the connection is authenticated, not just the initial request
- **Familiar to enterprises:** Many organizations already have PKI infrastructure and certificate management processes

Disadvantages:

- **TLS termination breaks it:** If a load balancer, CDN, or reverse proxy terminates TLS, the client certificate information must be forwarded via headers (e.g., X-Client-Cert), which reintroduces trust assumptions
- **Certificate management overhead:** Issuing, distributing, rotating, and revoking client certificates is operationally complex
- **Not compatible with all clients:** Some HTTP client libraries and tools have limited mTLS support
- **Point-to-point only:** Works best for direct connections; multi-hop architectures require careful design

16.5.2. When to Use mTLS

mTLS is well-suited for:

- **Direct server-to-server connections** where TLS is not terminated by intermediaries
- **High-security deployments** where the operational overhead of certificate management is acceptable
- **Custodian authentication** where the Custodian has no FDO Owner Key but has been issued a client certificate
- **Environments with existing PKI** where client certificates are already in use for other purposes

mTLS is less suitable for:

- **Cloud deployments** where TLS is typically terminated at a load balancer or CDN
- **Integrations with third-party services** that may not support mTLS
- **Lightweight clients** (scripts, CLI tools) where certificate management is burdensome

16.5.3. Certificate Enrollment

Client certificates for mTLS are enrolled **out-of-band** — this specification does not define a certificate enrollment

protocol. Typical enrollment methods include:

- **Manual exchange:** The client generates a certificate signing request (CSR), sends it to the server operator, and receives a signed certificate
- **Enterprise PKI:** The client obtains a certificate from an organizational CA that the server trusts
- **Partner portal:** The server operator provides a certificate through a web portal or API
- **ACME with client extensions:** Automated certificate issuance (less common for client certificates)

During enrollment, the server operator associates the client certificate (or its issuing CA) with a caller identity — either an Owner (with a corresponding Owner Key) or a Custodian (with business authority but no Owner Key).

16.5.4. Server Configuration

Servers that support mTLS MUST:

1. **Configure TLS to request client certificates** (ClientAuth in Go, ssl_verify_client in nginx, etc.)
2. **Maintain a mapping** from client certificate identities (subject DN, fingerprint, or issuing CA) to caller identities (Owner or Custodian)
3. **Scope API responses** to vouchers associated with the authenticated caller identity, just as with bearer or DPoP tokens

16.5.5. Request Processing

When a request arrives over an mTLS connection:

1. **TLS handshake completes** with client certificate verification
2. **Server extracts client identity** from the verified certificate
3. **Server looks up caller identity** in its certificate-to-identity mapping
4. ****If Authorization header is absent**:** The caller is authenticated via mTLS; proceed with the request scoped to the mapped identity
5. ****If Authorization header is present**:** The server MAY validate both (defense-in-depth) or MAY ignore the header and use mTLS identity alone — this is implementation-specific

16.5.6. Custodian Authentication via mTLS

mTLS is particularly useful for Custodians — entities with business authority over vouchers but no FDO Owner Key. A Custodian can:

1. **Enroll a client certificate** with the server operator during business onboarding
2. **Connect via mTLS** to push, pull, assign, or query vouchers
3. **Perform all Custodian operations** without needing FDOKeyAuth, bearer tokens, or DPoP

This provides a clean separation: Custodians use mTLS (or other out-of-band credentials), while Owners use FDOKeyAuth to prove possession of their Owner Key.

16.5.7. Combining mTLS with Token Authentication

Modern infrastructure often terminates TLS at load balancers, API gateways, CDNs, or service meshes — meaning the "mutual" authentication in mTLS doesn't actually reach the application server. The client certificate

is verified at the edge, but the connection between the edge and the application is a separate TLS session (or even plaintext within a trusted network). This breaks the end-to-end security model that mTLS promises.

The solution: use mTLS for transport security and tokens for application-level authentication.

This combination provides advantages that neither method achieves alone:

1. **mTLS secures the transport** — Even if TLS is terminated at a load balancer, the edge-to-client connection is authenticated. The client proves possession of a certificate, and the connection is encrypted with keys bound to that certificate.
2. **Tokens survive TLS termination** — The Authorization header passes through proxies, load balancers, and service meshes unchanged. The application server validates the token regardless of how many TLS sessions were involved in the request path.
3. **Defense-in-depth** — An attacker who compromises a token still cannot use it without also compromising the mTLS certificate (or bypassing the edge). An attacker who compromises a certificate still cannot access resources without a valid token.
4. **DPoP adds proof-of-possession** — When combined with DPoP tokens, each request proves possession of both the mTLS certificate (at the edge) and the DPoP key (at the application). This is stronger than either mechanism alone.

Deployment patterns:

Servers MAY accept multiple authentication methods simultaneously:

- **mTLS only:** Client certificate is sufficient; Authorization header is ignored. Suitable for point-to-point connections where TLS is not terminated.
- **Token only:** No mTLS required; standard TLS (server-authenticated) is sufficient. Suitable for cloud deployments, browser clients, or environments without PKI.
- **mTLS + token (recommended for high security):** Both must be valid. mTLS secures the transport; the token provides application-level identity that survives TLS termination.
- **mTLS at edge, token at application:** The edge (load balancer, API gateway) validates the client certificate and may forward certificate info in headers. The application validates the token. This is the most common pattern in modern cloud infrastructure.

The choice is deployment-specific and SHOULD be documented in the server's API documentation.

16.5.8. Relationship to DID Enrollment

When using DIDs for enrollment (see [§ 19 DID Integration](#)), the DID document MAY include a `tlsClientCertificateAuthority` field specifying a CA whose certificates the server should accept for mTLS:

```
{
  "id": "did:web:reseller.example.com",
  "verificationMethod": [...],
  "service": [{
    "id": "did:web:reseller.example.com#voucher-client",
    "type": "FDOVoucherClient",
    "serviceEndpoint": {
      "tlsClientCertificateAuthority": "-----BEGIN CERTIFICATE-----\n..."
    }
  }]
}
```

This allows the server to automatically trust client certificates issued by the enrolled party's CA, reducing manual certificate enrollment overhead.

16.6. Wire Format: CBOR Encoding§

The FDOKeyAuth protocol uses CBOR [RFC8949] encoding for all authentication messages, even though the subsequent Push and Pull APIs use JSON. This is a deliberate design choice:

- **Deterministic hashing:** The protocol relies on hash continuity between messages. CBOR provides deterministic encoding (see [RFC8949] Section 4.2), unlike JSON which has canonicalization problems (key ordering, whitespace, number formatting).
- **Native binary data:** Nonces, public keys, hashes, and COSE signatures are binary values — `nativebstr` in CBOR, but requiring `base64url` encoding in JSON.
- **COSE integration:** COSE_Sign1 [RFC9052] objects are defined in CBOR. Nesting COSE inside CBOR is natural; inside JSON it requires an additional serialization boundary.
- **Consistency with FDO:** The FDO protocol ([FDO-Specification]) uses CBOR/COSE throughout. Using the same wire format for FDOKeyAuth reduces implementation friction.
- **Scoped complexity:** CBOR is limited to the three authentication messages. Once authenticated, the bearer token is used with the JSON-based Push and Pull APIs.

16.7. Delegation Support§

16.7.1. Delegate Certificate Requirements§

When a Caller is a Delegate (i.e., it does not possess the enrolled key directly but has been authorized by the key owner), the Delegate Chain presented in `FD0KeyAuth.Hello` MUST satisfy the following requirements:

1. **Chain Structure:** The Delegate Chain is an X5CHAIN as defined in [FDO-Specification]. It MAY contain a single certificate (signed directly by the key owner) or multiple certificates forming a chain.
2. **Root Trust:** The root (top) certificate of the chain MUST be signed by the key specified in the `CallerKey` field of `FD0KeyAuth.Hello`.
3. **Leaf Identity:** The leaf (bottom) certificate identifies the Delegate. The Caller signs with the private key corresponding to this leaf certificate's public key.
4. **Required Permission:** At least one certificate in the chain MUST include the appropriate Extended Key Usage (EKU) permission (see § 16.7.2 Extended Key Usage Permissions).
5. **Permission Subsetting:** For delegation under this specification, when permissions are present in a certificate, a subset of these permissions MUST be specified in certificates closer to the leaf. This allows hierarchical delegation with progressively narrower permissions.
6. **Standard X.509 Validation:** All certificates MUST be valid (not expired, not revoked), non-leaf certificates MUST set the `IsCA` flag, and a single-certificate chain MUST NOT set the `IsCA` flag.

16.7.2. Extended Key Usage Permissions§

This specification defines the following EKU permissions for use in Delegate Certificates:

Identifier	OID	Description
<code>fdo-ekt-permit-voucher-claim</code>	1.3.6.1.4.1.45724.3.1.5	Authorizes the Delegate to claim (pull/download) vouchers signed over to the delegating key. Required for pull authentication.
<code>fdo-ekt-permit-voucher-upload</code>	1.3.6.1.4.1.45724.3.1.6	Authorizes the Delegate to upload (push) vouchers on behalf of the delegating key. Required for push authentication.

NOTE: The OIDs follow the FDO permission OID `arc.fdo-ekt-permit-voucher-claim` (PERM.5) was discussed during [\[FDO-Specification\]](#) development. `fdo-ekt-permit-voucher-upload` (PERM.6) is newly defined by this specification for push delegation.

16.7.3. Delegate Authentication Flow

When Delegation is in use, the protocol flow is identical to the direct key flow with the following differences:

1. In `FD0KeyAuth.Hello`, the Caller includes both `CallerKey` (the key owner's public key as a `COSE_Key`) and `DelegateChain` (the X5CHAIN authorizing the Delegate, as an array of DER-encoded certificates).
2. The Server verifies the Delegate Chain in addition to the standard Hello processing.
3. In `FD0KeyAuth.Prove`, the Caller signs with the **Delegate private key** (corresponding to the leaf certificate) instead of the key owner's private key.
4. The Server verifies the signature against the leaf certificate's public key, and verifies the chain is rooted by `CallerKey`.

16.8. Continuation Token Security

16.8.1. Overview

When pulling large numbers of vouchers, the Pull API uses continuation tokens for pagination (see [§ 11.2 API Endpoints](#)). After authentication, the Caller receives a `SessionToken` scoped to the authenticated key. This token is used as a standard `Authorization: Bearer` header with the JSON-based Pull API. Continuation tokens are used within this authenticated session.

16.8.2. Threat: Continuation Rollback

An attacker who can observe or intercept continuation tokens could potentially:

- **Replay** a previous continuation token to re-download vouchers already retrieved
- **Forge** a continuation token to skip ahead or access vouchers outside the intended page sequence

While the impact of such attacks is limited (the attacker would only receive vouchers already authorized for the authenticated key), it could cause operational issues such as duplicate processing or missed vouchers.

16.8.3. Mitigation Requirements

Implementations SHOULD protect continuation tokens against manipulation:

1. **Cryptographic Binding:** Continuation tokens SHOULD be cryptographically bound to the session. The Server SHOULD include a MAC or signature over the token contents using a server-side secret, ensuring that tokens cannot be forged by a client.
2. **Session Scoping:** Continuation tokens MUST be valid only within the session that created them. A continuation token from one authenticated session MUST NOT be accepted in a different session.
3. **Monotonic Progression:** Implementations MAY enforce that continuation tokens are used in sequence (i.e., the Caller cannot "go back" to a previous page). This prevents rollback attacks but may complicate error recovery.
4. **Expiration:** Continuation tokens SHOULD have a short expiration time (e.g., matching the session token lifetime). Expired continuation tokens MUST be rejected.

5. **Opacity:** Continuation tokens **MUST** remain opaque to the client. Implementations **SHOULD NOT** encode plaintext page numbers or offsets that could be manipulated.

16.8.4. Recommended Token Structure

Implementations **MAY** use the following internal structure for continuation tokens (opaque to the client):

```
ContinuationToken =  
    MAC(server_secret,  
        session_id || page_offset || filter_hash || expiry)  
    || payload
```

Where payload contains the encoded pagination state and MAC ensures integrity. The `filter_hash` binds the token to the specific query parameters used when the token was issued, preventing parameter substitution attacks.

16.8.5. Informative Note on Continuation Rollback Risk

If continuation tokens are **NOT** cryptographically protected, the practical risk is limited: an attacker could cause the Caller to re-download vouchers it was already authorized to receive, or skip pages of vouchers. This does **NOT** allow access to vouchers belonging to other keys (which is prevented by the session token scoping). Deployments where this risk is acceptable **MAY** use simpler continuation token implementations, but **SHOULD** document this decision.

16.9. Implementation Notes

16.9.1. Signing Oracle Protection

A critical security requirement of this protocol is that the Caller **MUST NOT** be tricked into signing data that could be used for purposes other than `FDOKeyAuth`. This is achieved through:

1. **Explicit Message Type Tags:** Every signed payload begins with a string literal identifying the protocol message type (e.g., `"FDOKeyAuth.Prove"`). This tag is structurally distinct from any other FDO protocol message.
2. **Structured Payloads:** Signed payloads are CDDL-defined CBOR arrays with a fixed structure, not opaque byte strings. A Server cannot construct a payload that simultaneously satisfies the `FDOKeyAuthProveSignedPayload` structure and an `OVEntry` or `T02ProveOVHdrPayload` structure.
3. **Nonce Binding:** The inclusion of both parties' nonces in every signed message ensures that signatures are bound to a specific protocol session and cannot be lifted for use elsewhere.

16.9.2. Session State Management

Servers **MUST** manage authentication session state carefully:

- Sessions **SHOULD** expire within 60 seconds of creation if `FDOKeyAuth.Prove` is not received
- Sessions **MUST** be single-use: once `FDOKeyAuth.Prove` is processed (success or failure), the session **MUST** be invalidated
- Implementations **SHOULD** limit the number of concurrent pending sessions per source IP to prevent resource exhaustion
- Session identifiers **MUST** be cryptographically random and sufficiently long (at least 128 bits) to prevent guessing

16.9.3. Server Signature Verification by Caller§

While the Server's signature in `FDOKeyAuth.Challenge` is OPTIONAL to verify (the Caller trusts the TLS connection), Caller implementations SHOULD verify it when the Server's signing key is known (e.g., from a DID document or pre-configured trust store). This provides defense-in-depth against man-in-the-middle attacks at the TLS termination point.

17. Security Framework§

17.1. How Authentication Works§

The voucher transfer protocol authenticates using **credentials that already exist from enrollment** (see [§ 3.3 Enrollment: Exchanging Credentials](#)). No additional tokens, passwords, or API keys are needed.

Both push and pull operations use the same authentication mechanism: the **FDOKeyAuth** cryptographic challenge-response handshake (see [§ 16 FDOKeyAuth Protocol](#)). The caller proves possession of an enrolled key and receives a bearer token. The key differs by direction:

- **Push:** The Supplier proves possession of the Supplier Key (the key the Recipient enrolled via DID exchange)
- **Pull:** The Owner proves possession of the Owner Key (the key vouchers were signed over to)

In addition to FDOKeyAuth bearer-token authentication, the Recipient SHOULD also verify the voucher's cryptographic signature against enrolled Supplier DID keys. This provides **defense-in-depth**: the bearer token proves the Supplier is authorized to push, and the voucher signature proves the voucher was actually created by that Supplier.

17.1.1. Security Properties of FDOKeyAuth§

- **Mutual Authentication:** Both parties contribute nonces, ensuring both are live participants
- **Replay Prevention:** Nonces bind each signature to a specific session
- **Signing Oracle Protection:** Signed payloads use explicit type tags ("`FDOKeyAuth.Prove`") that are structurally distinct from all other FDO messages
- **Message Integrity:** Hash chaining detects tampering between protocol steps
- **Delegation Support:** Authorized delegates can authenticate on behalf of a key owner using ECU permissions (see [§ 16.7.2 Extended Key Usage Permissions](#))

17.1.2. Summary§

Transfer Direction	Authentication Mechanism	Credential Source	Extra Setup?
Push (Supplier → Recipient)	FDOKeyAuth proving Supplier Key + voucher signature verification	Supplier DID enrolled during business setup	No
Pull (Recipient ← Holder)	FDOKeyAuth proving Owner Key possession	Owner Key enrolled during business setup (signed into voucher)	No

17.2. Threat Model§

The following threats apply to voucher transfer endpoints:

- **Unauthorized Voucher Submission:** Malicious actors sending fraudulent vouchers to a Voucher Recipient

- **Voucher Injection:** Attackers submitting vouchers for devices they do not own
- **Denial of Service:** Overwhelming Voucher Recipient systems with invalid vouchers or requests
- **Data Harvesting:** Using voucher endpoints to discover device information
- **Manufacturer Impersonation:** Fake Voucher Issuers submitting counterfeit vouchers

FDOKeyAuth addresses all of these threats: push authentication rejects unauthenticated suppliers before voucher processing begins; pull authentication rejects unauthenticated owners at the first message. Voucher signature verification provides an additional layer of defense for push operations. See [§ 19.2 DID-Based Security Model](#) for the complete trust analysis.

17.3. DDoS Considerations for Push Endpoints§

With FDOKeyAuth authentication required for push operations, unauthenticated requests are rejected before any voucher processing occurs. This significantly reduces the DDoS attack surface compared to a model where the voucher itself is the only authentication artifact.

Deployments exposed to high-volume or adversarial traffic MAY add additional **edge protection layers** (see [Appendix A: Optional Security Layers \(Informative\)](#)) such as mTLS or API gateway rate limiting for further resilience.

18. Voucher Sequestering§

18.1. Overview§

Voucher sequestering holds submitted vouchers in quarantine until additional validation or approval is completed. This provides an additional layer of protection against unauthorized or suspicious voucher submissions.

18.2. Process§

1. **Quarantine Reception:** Voucher is accepted but held in quarantine status
2. **Additional Validation:** Business logic, order verification, or manual review is performed
3. **Approval:** Automatic or manual approval based on risk assessment rules
4. **Release or Reject:** Approved vouchers are released for use; rejected vouchers are deleted or archived

18.3. Risk-Based Sequestering§

Risk Level Auto-Approval		Manual Review	Quarantine Duration
Low	Yes	No	1 hour
Medium	Conditional	Yes	4 hours
High	No	Required	24 hours
Critical	No	Executive	72 hours

Implementations SHOULD define risk assessment rules based on:

- Whether the Voucher Issuer is a known, trusted manufacturer
- Whether the voucher matches a verified purchase order
- Whether the device count is within expected ranges

- Whether the submission pattern is anomalous

19. DID Integration§

19.1. Trust Foundation: Mutual DID Exchange§

The security model for voucher transfer is rooted in a single prerequisite:**the mutual exchange of DIDs between trusted parties**. A DID is a short identifier string (e.g.,`did:web:acme-mfg.com`) that can be pasted into a purchase order, typed into a web portal, or included in an ordering system — no file exchange or document upload is required. This exchange happens once during business relationship setup (e.g., when a device owner enrolls a manufacturer, or when a manufacturer registers a customer). The exchange is an explicit, human-verified act — the parties are responsible for ensuring they are exchanging DIDs with the correct counterpart.

Once DIDs are exchanged, each party can resolve the other's DID document to obtain:

- **The other party's public key**(`verificationMethod`) — enabling cryptographic verification of identity
- **The other party's service endpoints**(`service`) — enabling endpoint discovery without further configuration
- **An implicit trust relationship** — the act of enrolling a DID is a declaration that "I trust this party for voucher operations"

From this single exchange, **all subsequent authentication, authorization, and endpoint discovery is handled by the protocol itself**. No additional credentials (tokens, passwords, API keys) need to be provisioned or exchanged. The protocol's cryptographic mechanisms — voucher signatures, `FDOKeyAuth` (see [§ 16 FDOKeyAuth Protocol](#)), and TLS — provide all necessary security assurances.

19.2. DID-Based Security Model§

The following analysis demonstrates that mutual DID exchange is sufficient for all four trust relationships in the push and pull transfer models. In each case, the security assurance comes from cryptographic properties already present in the protocol, not from pre-shared secrets or bearer tokens.

19.2.1. Case 1: Provider Pushes Only to a Provably-Known Owner§

Trust requirement: The Voucher Issuer (Provider) must ensure it is sending vouchers to the legitimate Voucher Recipient (Owner), not an impersonator.

What the Provider has: The Recipient's DID, obtained during business setup.

How it works:

1. Provider resolves the Recipient's DID to obtain the Push Service Root URL (`FDOVoucherRecipient` service entry)
2. Provider connects to the endpoint over HTTPS — standard TLS or DID-pinned TLS (see [§ 19.5.3 Optional TLS Certificate Authority](#)) validates the endpoint
3. Provider signs the voucher over to the Recipient's Owner Key (from `verificationMethod` in the Recipient's DID)

Even in the unlikely event of endpoint compromise, the voucher is cryptographically bound to the Recipient's Owner Key. Only the holder of the corresponding private key can use the voucher for device onboarding. **No token or credential exchange is needed.**

19.2.2. Case 2: Recipient Accepts Pushes Only from Known-Trusted Providers§

Trust requirement: The Voucher Recipient must verify that a received voucher was issued by a trusted manufacturer, not a malicious actor.

What the Recipient has: The Provider's DID, enrolled during business setup.

How it works:

1. Recipient receives a voucher via its Push Service Root endpoint
2. Recipient extracts the manufacturer's signing key from the voucher's signature chain
3. Recipient verifies the voucher signature against the manufacturer's public key from the enrolled DID (verificationMethod)
4. If the signature is valid and the signing key matches an enrolled manufacturer DID, the voucher is accepted

The voucher itself is a self-authenticating artifact — it carries cryptographic proof of its issuer. **No token or credential exchange is needed.**

19.2.3. Case 3: Recipient Pulls from a Known Provider

Trust requirement: The Voucher Recipient must know it is pulling from the legitimate Holder, not a man-in-the-middle serving fabricated vouchers.

What the Recipient has: The Holder's DID, obtained during business setup.

How it works:

1. Recipient resolves the Holder's DID to obtain the Pull Service Root URL (FD0VoucherHolder service entry)
2. Recipient connects over HTTPS to the Pull Service Root
3. Recipient initiates Owner Key Authorization (FDOKeyAuth, see [§ 16 FDOKeyAuth Protocol](#)), which provides mutual authentication: the Recipient proves it holds the Owner Key, and the Server signs its challenge with a key verifiable against the Holder's DID
4. Downloaded vouchers can be further verified against the manufacturer's DID (as in Case 2)

No token or credential exchange is needed. FDOKeyAuth provides mutual cryptographic authentication using keys already conveyed via DID exchange.

19.2.4. Case 4: Provider Allows Pulls Only from Authorized Owners

Trust requirement: The Holder (Provider) must release vouchers only to the party whose Owner Key the vouchers were signed over to — not to any authenticated party.

What the Holder has: The vouchers themselves, which contain the Owner Key they were signed over to.

How it works:

1. Recipient authenticates via FDOKeyAuth (see [§ 16 FDOKeyAuth Protocol](#)), proving possession of its Owner Key (or an authorized Delegate Key)
2. The Holder computes the fingerprint of the authenticated key
3. The Holder returns **only** vouchers whose final Owner Key matches the authenticated fingerprint
4. Delegation support (see [§ 16.7 Delegation Support](#)) allows authorized delegates to pull on behalf of an Owner using the fdo-ekt-permit-voucher-claim permission

The Owner Key IS the authorization credential — it is the same key the vouchers were signed over to. **No token or credential exchange is needed.**

19.2.5. Summary§

Case Direction	Trust Question	Security Mechanism	Token Needed?
1 Push → Recipient	Am I sending to the right owner?	Voucher signed to Recipient's Owner Key (from DID)	No
2 Push ← Provider	Did this come from a trusted manufacturer?	Voucher signature verified against Provider's DID key	No
3 Pull ← Holder	Am I pulling from the real holder?	FDOKeyAuth mutual authentication + DID-pinned TLS	No
4 Pull → Recipient	Is this puller authorized for these vouchers?	FDOKeyAuth Owner Key proof + fingerprint scoping	No

In all four cases, the only prerequisite is the one-time, human-verified exchange of DIDs. The protocol's cryptographic mechanisms handle all subsequent authentication and authorization.

19.3. Endpoint Discovery§

DID documents [\[DID-Core\]](#) convey the Service Root URLs (see [§ 9 Service Root URLs](#)) that allow parties to discover each other's voucher transfer endpoints without out-of-band configuration.

19.3.1. Push Discovery§

1. Voucher Issuer resolves Voucher Recipient's DID URI
2. Voucher Issuer extracts Push Service Root URL from the `FD0VoucherRecipient` service entry
3. Voucher Issuer POSTs voucher directly to the extracted root URL
4. If DID resolution fails, Voucher Issuer falls back to a statically configured endpoint

19.3.2. Pull Discovery§

1. Voucher Recipient resolves the Holder's DID URI
2. Voucher Recipient extracts Pull Service Root URL from the `FD0VoucherHolder` service entry
3. Voucher Recipient initiates FDOKeyAuth handshake at `{pull_root}/auth/hello`
4. If DID resolution fails, Voucher Recipient falls back to a statically configured endpoint

19.4. FDO JSON-LD Context§

DID documents are JSON-LD [\[JSON-LD\]](#) documents. Terms not defined in a `loaded@context` are ignored by conformant JSON-LD processors. Because this protocol introduces custom service types (`FD0VoucherRecipient`, `FD0VoucherHolder`) and custom properties (`tlsCertificateAuthority`, `fido-device-onboarding`), DID documents SHOULD reference the FDO JSON-LD context to ensure these terms are formally defined:

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://fidoalliance.org/ns/fdo/v1"
  ]
}
```

The FDO context document at <https://fidoalliance.org/ns/fdo/v1> defines the following terms:

Term	IRI	Description
FD0VoucherRecipient	https://fidoalliance.org/ns/fdo#FD0VoucherRecipient	Service type for push endpoints
FD0VoucherHolder	https://fidoalliance.org/ns/fdo#FD0VoucherHolder	Service type for pull endpoints
tlsCertificateAuthority	https://fidoalliance.org/ns/fdo#tlsCertificateAuthority	Optional PEM CA cert for TLS pinning
fido-device-onboarding	https://fidoalliance.org/ns/fdo#fido-device-onboarding	Informational extension block

Implementations that do not perform JSON-LD processing (i.e., those that parse DID documents as plain JSON) MAY omit the FDO context reference. The service type strings, property names, and data formats defined in this specification are self-describing and can be parsed without JSON-LD expansion. The FDO context exists for correctness under JSON-LD processing rules and for interoperability with generic DID tooling, not as a runtime dependency.

NOTE: The FIDO Alliance can publish this context document at the canonical URL. Until it is available there, implementations can use an inline context definition or a self-hosted URL. If published, the context URL is intended to be a stable identifier, and its contents MUST NOT change in backward-incompatible ways.

19.5. DID Document Service Types

DID documents for voucher transfer participants use standard DID service entries to advertise Service Root URLs.

19.5.1. FD0VoucherRecipient

The FD0VoucherRecipient service type advertises the Push Service Root URL. Its `serviceEndpoint` is the full URL to which Voucher Issuers submit vouchers (i.e., the Push Service Root).

19.5.2. FD0VoucherHolder

The FD0VoucherHolder service type advertises the Pull Service Root URL. Its `serviceEndpoint` is the full URL from which Voucher Recipients retrieve vouchers (i.e., the Pull Service Root). This allows a party to give the same DID to an upstream Voucher Issuer (for push) and to downstream partners (for pull), each service type conveying the appropriate root.

19.5.3. Optional TLS Certificate Authority

Each service entry MAY include a `tlsCertificateAuthority` field containing a PEM-encoded CA certificate (or certificate chain) that the connecting party SHOULD use to validate the TLS connection to that service endpoint. This enables TLS pinning to a private CA without relying solely on the public WebPKI.

This field is OPTIONAL. When absent, standard WebPKI certificate validation applies. When present, the connecting party SHOULD treat it as the trust anchor for that specific endpoint's TLS connection.

Push and pull service entries MAY specify different CA certificates, since they MAY be hosted on different infrastructure with different certificate chains.

19.5.4. Example DID Document

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://fidoalliance.org/ns/fdo/v1"
  ],
  "id": "did:web:owner.example.com",
  "verificationMethod": [...],
  "service": [
    {
      "id": "did:web:owner.example.com#voucher-recipient",
      "type": "FDOVoucherRecipient",
      "serviceEndpoint": "https://owner.example.com/fdo/v1/push/vouchers",
      "tlsCertificateAuthority": "-----BEGIN CERTIFICATE-----\nMIIBxTC...push-ca...\n-----END CERTIFICATE-----"
    },
    {
      "id": "did:web:owner.example.com#voucher-holder",
      "type": "FDOVoucherHolder",
      "serviceEndpoint": "https://owner.example.com/fdo/v1/pull/vouchers",
      "tlsCertificateAuthority": "-----BEGIN CERTIFICATE-----\nMIIBxTC...pull-ca...\n-----END CERTIFICATE-----"
    }
  ]
}
```

Note that the serviceEndpoint URLs are entirely deployment-specific. They MAY point to the same host or to different hosts. The path components are chosen by the operator and do not need to follow any particular convention — they are the Service Root URLs from which all API endpoints are derived (see [§ 9 Service Root URLs](#)). The tlsCertificateAuthority field is entirely OPTIONAL and MAY be omitted from either or both service entries.

19.6. Defense-in-Depth Layers (Informative)

The DID-based security model described above is sufficient for all voucher transfer operations. However, deployments that operate behind API gateways, web application firewalls (WAFs), or other layered security infrastructure MAY choose to add additional authentication mechanisms on top of the DID-based model. These are **additive layers** — they do not replace the core cryptographic assurances, and their setup requires provisioning beyond the simple exchange of DIDs between trusted parties. See [Appendix A: Optional Security Layers \(Informative\)](#) for detailed descriptions of these mechanisms.

Examples of additive security layers include:

- **Bearer tokens:** JWT tokens issued by the Recipient to authorized Issuers, enabling fast rejection at edge/WAF infrastructure before reaching the application. Useful for rate limiting and quota enforcement.
- **Mutual TLS (mTLS):** Client certificates enrolled by the Recipient for manufacturer authentication at the TLS layer. Useful when TLS termination happens at the application rather than at a reverse proxy.
- **Business logic validation:** Application-level checks against purchase orders, device registries, or CRM systems.

These mechanisms are out of scope for the core protocol design. They MAY be configured between parties through deployment-specific means, but are not required for secure voucher transfer. DID documents MAY advertise supported additive mechanisms via an fido-device-onboarding extension:

```
{
  "fido-device-onboarding": {
    "supportedFormats": ["application/x-fdo-voucher"],
    "additionalAuthentication": ["bearer", "mtls"],
    "trusted_manufacturers": ["did:web:mfg1.com", "did:web:mfg2.com"]
  }
}
```

This extension is purely informational. Its presence does not impose requirements on connecting parties beyond what is specified by the core protocol.

20. Key Formats and Enrollment Considerations§

20.1. Key Independence After Enrollment§

Throughout this specification, the term **enrollment** refers to the one-time exchange of identity information — primarily a public key and optionally a service endpoint URL — between a Supplier and a Recipient. Enrollment establishes the trust relationship and cryptographic credentials required for all subsequent voucher operations: signing vouchers over to the Recipient’s key, pushing vouchers to the Recipient’s endpoint, pulling vouchers from the Supplier’s endpoint, and authenticating via FDOKeyAuth.

A critical design principle is that **once a key is enrolled, the format in which it was originally conveyed is irrelevant**. The enrollment mechanism imports the key material into the party’s trust store. From that point forward, the key is used for voucher sign-over, signature verification, and FDOKeyAuth authentication — all of which operate on the key itself, not on its enrollment encoding. Whether the key arrived as a `did:web` URI, a `did:key` string, a PEM-encoded public key, or an X.509 certificate does not affect any subsequent protocol operation.

This means that the choice of key format is primarily an **enrollment-time concern** — it affects the user experience of onboarding a new trust relationship, not the security or functionality of the protocol itself.

20.2. Enrollment Mechanisms§

The specific mechanism by which parties enroll each other’s credentials is largely out of scope for this specification, but is clearly important to any deployment. Enrollment is a business process that accompanies the establishment of a commercial relationship — the purchase of devices, the onboarding of a new supplier, or the registration of a new customer.

In practice, enrollment may take many forms:

- **Purchase order** — The Recipient includes a DID or public key as a line item on a purchase order when buying devices
- **Supplier web portal** — The Recipient pastes a DID or uploads a key file into the supplier’s ordering or provisioning portal
- **Email exchange** — Parties exchange DIDs or key files via email during business setup
- **API integration** — Automated enrollment via the supplier’s partner onboarding API
- **Sales engineering** — A supplier’s technical team manually configures the Recipient’s key during deployment planning

In all cases, the fundamental information exchanged during enrollment is:

1. **A public key (REQUIRED)** — The Recipient’s Owner Key, to which vouchers will be signed over. The Supplier’s signing key, against which the Recipient will verify voucher signatures.
2. **A service endpoint URL (CONDITIONAL)** — Required only if vouchers will be pushed to the Recipient via

the Push API, or if the Recipient will pull from the Supplier via the Pull API. Not needed if vouchers will be transferred exclusively out-of-band (see [§ 3.4 Transfer: Push, Pull, or Out-of-Band](#)).

3. **A TLS root CA certificate** (OPTIONAL) — A PEM-encoded CA certificate (or certificate chain) that the connecting party SHOULD use to validate the TLS connection to the service endpoint. Required only when the endpoint uses a private CA, self-signed certificate, or other non-WebPKI trust anchor. When omitted, standard WebPKI certificate validation applies. DID documents convey this via the `tlsCertificateAuthority` field in each service entry (see [§ 19.5.3 Optional TLS Certificate Authority](#)); for non-DID enrollment, it MUST be conveyed alongside the URL through whatever mechanism is used for enrollment (see [§ 22.5.9.3 TLS Trust Models](#) for when and why this is needed).

DID URIs are emphasized throughout this specification because they conveniently encode all three pieces — key, service URLs, and optional CA certificates — in a single short string. However, the protocol does not require DIDs — any format that conveys the key material (and optionally a URL and CA certificate) is sufficient.

20.3. Key Format Comparison

This section compares the key formats that enrollment mechanisms MAY accept. The choice of format affects the enrollment experience and the operational properties of the trust relationship, but does not affect the security of the voucher transfer protocol itself.

20.3.1. did:web

A `did:web` identifier (e.g., `did:web:acme-mfg.example.com`) resolves to a DID document hosted at a well-known HTTPS URL on the identified domain. The DID document contains the public key (`verificationMethod`) and optionally service endpoints for push and pull transfer.

Advantages:

- **Single string encodes key and URLs** — One short, human-readable string conveys everything needed for enrollment: the public key and any push/pull service endpoints
- **Easy to convey** — A cleartext string that can be pasted into a web form, written on a purchase order, or included in an email
- **External key management** — The key is hosted at a URL, so the domain operator (or a third-party service) can manage key rotation, expiration, and revocation without requiring the enrolling party to re-enroll
- **Service endpoint discovery** — Push and pull URLs are embedded in the DID document, eliminating the need to convey them separately

Disadvantages:

- **Relies on DNS and web infrastructure** — Resolving the DID requires DNS lookup and HTTPS fetch, introducing dependencies on DNS integrity and web server availability. A compromised DNS record or web server could serve a fraudulent key
- **Not self-contained** — The key material is not in the identifier itself; it must be fetched. This creates a time-of-use vs. time-of-enrollment consistency concern
- **Attack surface** — DNS hijacking, BGP hijacking, or compromise of the hosting web server are all vectors that could result in the wrong key being returned

20.3.2. did:key

A `did:key` identifier (e.g., `did:key:z6Mkf5rGMoatrSj1f...`) encodes the public key directly in the identifier string itself. No resolution step is required — the key material is self-contained in the DID.

Advantages:

- **Self-contained and prescriptive** — The key is encoded directly in the string. The Recipient gives the Supplier exactly the key to use, with no ambiguity and no external resolution
- **No DNS or web dependency** — No network fetch is required to obtain the key material, eliminating an entire class of attack vectors
- **Cleartext string** — Like `did:web`, it is a string that can be pasted into forms or written on purchase orders (though it is longer and less human-readable)

Disadvantages:

- **No service endpoint encoding** — `did:key` does not support service entries. Push and pull URLs must be conveyed separately through another mechanism
- **No key management** — There is no mechanism for key rotation, expiration, or revocation. To change the key, the parties must re-enroll with a new `did:key` string
- **Longer and less readable** — The base-encoded key material makes the string significantly longer than a `did:web` identifier

20.3.3. X.509 Certificates

An X.509 certificate (PEM or DER encoded) conveys a public key along with metadata including validity period, issuer, and optional revocation information (CRL/OCSP).

Advantages:

- **Prescriptive** — Like `did:key`, the certificate contains the exact key to use
- **Expiration and revocation** — X.509 certificates natively support validity periods and revocation via CRL Distribution Points or OCSP, providing lifecycle management that `did:key` and raw PEM keys lack
- **Widely supported** — X.509 is a mature, universally supported standard with extensive tooling
- **HSM and CA compatibility** — Commercial Hardware Security Modules (HSMs) and Certificate Authorities (CAs) operate natively with X.509 certificates. In enterprise environments where key management policy requires keys to be generated and stored in an HSM, or where certificates must be issued by an organizational CA, X.509 is often the only viable format. Extracting keys from X.509 certificates and re-encoding them in other formats (PEM, COSE_Key, DID) would lose the management properties (revocation, expiration tracking) and may violate HSM security policies where private keys are non-extractable

Disadvantages:

- **No automatic rotation** — While certificates expire, there is no built-in mechanism to discover a replacement. The parties must re-enroll when a certificate expires
- **No service endpoint encoding** — Push and pull URLs must be conveyed separately
- **Not a simple string** — Certificates are binary (DER) or multi-line (PEM) objects, making them harder to include in purchase orders or paste into simple web forms. Typically conveyed as file uploads or attachments

20.3.4. Raw PEM Public Keys

A PEM-encoded public key (e.g., `-----BEGIN PUBLIC KEY-----`) conveys the raw key material without any surrounding metadata.

Advantages:

- **Prescriptive** — The key is exactly what is provided, with no ambiguity
- **Widely used today** — Many existing FDO implementations use PEM keys for enrollment

- **Simple** — No additional metadata, certificates, or resolution required

Disadvantages:

- **No key management** — No expiration, revocation, or rotation mechanism. Same limitations as `did:key`
- **No service endpoint encoding** — Push and pull URLs must be conveyed separately
- **Not a simple string** — Multi-line PEM encoding is awkward for purchase orders or simple text fields, though it works well for file uploads and API exchanges

20.3.5. Summary Comparison

Property	<code>did:web</code>	<code>did:key</code>	X.509 Certificate	Raw PEM Key
Encodes public key	Yes (via resolution)	Yes (self-contained)	Yes (self-contained)	Yes (self-contained)
Encodes service URLs	Yes	No	No	No
Easy to paste / convey as text	Yes (short string)	Moderate (long string)	No (binary or multi-line)	No (multi-line)
Key expiration	Yes (document update)	No	Yes (native)	No
Key revocation	Yes (document update)	No	Yes (CRL/OCSP)	No
Key rotation	Yes (document update)	No (re-enroll)	No (re-enroll)	No (re-enroll)
External key management	Yes	No	No	No
DNS/web dependency	Yes	No	No	No
Requires separate URL conveyance	No	Yes	Yes	Yes

Implementations that accept alternative key formats SHOULD document which formats they support and how endpoint URLs are conveyed alongside the key.

21. Error Handling and Retry Logic

21.1. Retry Strategy

Voucher Issuers MUST implement retry logic for transient failures:

1. **Exponential Backoff**: Base delay of 1 second, doubling on each retry (1s, 2s, 4s, 8s, 16s, 32s)
2. **Maximum Attempts**: At least 5 retries (configurable)
3. **Jitter**: Random delay of $\pm 25\%$ to prevent thundering herd
4. **Circuit Breaker**: Temporarily stop sending to endpoints that are consistently failing
5. **Dead Letter Queue**: Failed transfers SHOULD be stored for manual review

21.2. Error Categories

21.2.1. Transient Errors (Retry)

- Network timeouts
- HTTP 5xx server errors
- HTTP 429 Too Many Requests (honor `Retry-After` header)

- HTTP 503 Service Unavailable (honor Retry-After header)

21.2.2. Permanent Errors (Do Not Retry)§

- HTTP 400 Bad Request (invalid voucher format)
- HTTP 401 Unauthorized (authentication failure)
- HTTP 403 Forbidden (authorization failure)
- HTTP 409 Conflict (duplicate voucher)
- HTTP 413 Payload Too Large

21.2.3. Configuration Errors (Manual Intervention)§

- Invalid recipient URLs
- Missing or expired authentication credentials
- DID resolution failures
- Certificate validation errors

22. Security Considerations§

22.1. Transport Security§

All voucher transfer endpoints MUST use HTTPS with TLS 1.2 or later[RFC8446]. Certificate validation MUST be performed. HTTP Strict Transport Security (HSTS) headers SHOULD be set.

22.2. Error Response Security§

Error responses MUST NOT reveal internal system details, stack traces, or other information that could aid an attacker. Error responses SHOULD include a request identifier for correlation with server-side logs.

```
{
  "error": "authentication_failed",
  "message": "Invalid or expired token",
  "error_code": "AUTH_001",
  "timestamp": "2024-01-15T12:00:00Z",
  "request_id": "req-abc123"
}
```

22.3. Rate Limiting§

Voucher Recipient implementations SHOULD implement rate limiting:

- Per-manufacturer rate limits based on token or certificate identity
- Per-IP rate limits as a fallback
- Burst limits to handle legitimate spikes
- 429 Too Many Requests response with Retry-After header

22.4. Security Monitoring§

Implementations SHOULD monitor for:

- Authentication failures by manufacturer
- Invalid signature attempts
- Token usage anomalies
- Unusual geographic patterns
- Rate limit violations
- Certificate expiration warnings

22.5. FDOKeyAuth Security Considerations§

22.5.1. Signing Oracle Attacks§

The FDOKeyAuth protocol (see [§ 16 FDOKeyAuth Protocol](#)) requires the Caller to sign a payload provided in the context of a challenge from the Server. Care MUST be taken that this signed payload cannot be repurposed for other operations. The protocol addresses this through:

- **Explicit message type tags** ("FD0KeyAuth.Prove") that are structurally distinct from all other FDO protocol messages, including 0VEntry payloads and T02.Prove0VHdr12 payloads
- **CDDL-defined array structures** that cannot be confused with other protocol message formats
- **Nonce binding** that ties each signature to a specific, ephemeral protocol session

Implementations MUST NOT modify the signed payload structure in ways that could create ambiguity with other FDO message types.

22.5.2. Voucher Existence Disclosure§

When processing FD0KeyAuth.Hello, the Server learns which key the Caller is claiming. If the Server returns 404 Not Found when the key is not recognized, this reveals information about which keys are enrolled. In high-security deployments, the Server SHOULD proceed with the full challenge-response protocol regardless of whether the key is recognized, returning an empty result only after successful authentication. In lower-security deployments, early rejection MAY be acceptable to conserve resources.

22.5.3. Device GUID Entropy§

Device GUIDs are generated per [\[FDO-Specification\]](#) requirements and SHOULD have sufficient entropy to prevent enumeration attacks. Implementations MUST NOT use predictable or sequential GUIDs that would allow attackers to guess valid device identifiers.

22.5.4. Session Exhaustion§

The FDOKeyAuth protocol requires the Server to maintain session state between FD0KeyAuth.Hello and FD0KeyAuth.Prove. An attacker could attempt to exhaust server resources by initiating many authentication sessions without completing them. Implementations MUST:

- Enforce aggressive session timeouts (RECOMMENDED: <https://fidoalliance.org/specs/fido-IoT-specs-v1.0-rd-20261001/voucher-transfer-v1.0-rd-20261001.html>)
- Limit concurrent pending sessions per source IP
- Limit concurrent pending sessions per claimed key
- Use constant-space session storage where possible (e.g., encrypted session state in the `session_id` itself)

22.5.5. Delegate Certificate Revocation

When Delegation is used in FDOKeyAuth, the Server MUST verify that Delegate Certificates have not been revoked. Revocation checking SHOULD use OCSP or CRL mechanisms as appropriate. If revocation status cannot be determined, the Server SHOULD reject the authentication attempt in high-security deployments and MAY accept it with a warning in lower-security deployments.

22.5.6. Relationship to Transport Security

FDOKeyAuth provides application-layer authentication that is independent of, and complementary to, transport-layer security (TLS). Even when TLS is in use (as REQUIRED by this specification), the FDOKeyAuth protocol provides:

- **Authorization scoping:** TLS authenticates the transport peer but does not determine which Owner Key's vouchers should be returned
- **Defense-in-depth:** Application-layer cryptographic proof survives TLS termination at load balancers or reverse proxies
- **Audit trail:** Cryptographic proof of key possession can be logged for non-repudiation

22.5.7. Bearer Token Security Analysis

The bearer token issued by FDOKeyAuth (or by an alternative token source) is a session credential. Like all session credentials, its security depends on the confidentiality of the transport layer. This section analyzes the threat and compares it to alternative designs.

Threat: Token interception. If an attacker obtains a valid bearer token, they can use it to make API calls as the authenticated identity until the token expires. This is the session hijack threat.

Mitigation: TLS. All operations in this specification MUST use HTTPS with TLS 1.2 or later. Within a TLS session, the bearer token is encrypted in transit and cannot be observed by a network attacker. Token interception requires compromising the TLS session itself (e.g., compromising the server's private key, exploiting a TLS implementation vulnerability, or compromising an endpoint). These are catastrophic failures that would equally compromise a per-request challenge-response protocol — if an attacker can observe or modify traffic within the TLS session, they can intercept and replay any authentication evidence, not just bearer tokens.

Mitigation: Short token lifetime. Bearer tokens SHOULD have a lifetime of 1 hour or less (see [§ 16.3.1 Token Lifetime](#)). This limits the window during which an intercepted token is useful. Deployments that require a shorter exposure window MAY issue tokens with lifetimes as short as 5 minutes.

Comparison to per-request challenge-response. A design that requires a full cryptographic challenge-response for every API request would eliminate the bearer token as an interception target. However, it would introduce equivalent session state: the challenge nonces, session identifiers, and signed responses exchanged in each request would themselves be session credentials vulnerable to the same transport-level interception. The FDO TO2 protocol illustrates this — after the initial challenge-response, subsequent TO2 messages rely on session state (signed nonces, encrypted channels) that are no more resistant to transport compromise than a bearer token. The security difference between "bearer token over TLS" and "per-request signatures over TLS" is marginal; the operational and integration costs of the latter are substantial (see [§ 13.3 Design Rationale: Architectural Separation](#)).

Alternative: DPOP tokens. For deployments where bearer token theft is a significant concern, this specification supports DPOP (Demonstrating Proof of Possession) tokens as defined in [\[RFC9449\]](#). See [§ 16.4 DPOP: Sender-Constrained Tokens](#) for details. DPOP provides a middle ground between bearer tokens and full per-request challenge-response: the token itself is bound to a key, and each request includes a lightweight signed proof. This mitigates token theft without the complexity of full challenge-response on every request.

22.5.8. DPoP Security Analysis

DPoP [RFC9449] provides sender-constrained tokens that mitigate the token theft threat described above. This section analyzes the security properties of DPoP in the context of this specification.

22.5.8.1. Threat Model

DPoP addresses the following threats:

1. **Token theft from logs or caches.** Bearer tokens may be logged by intermediaries (proxies, API gateways, debugging tools) or cached in memory. An attacker who obtains a logged or cached bearer token can use it directly. DPoP tokens are useless without the corresponding private key.
2. **Token theft from compromised endpoints.** If an attacker compromises a client system, they may be able to extract bearer tokens from memory or storage. With DPoP, the attacker must also extract the DPoP private key — which may be stored in a hardware security module (HSM), TPM, or secure enclave that resists extraction.
3. **Token replay across different requests.** Each DPoP proof is bound to a specific HTTP method and URI. A proof captured from one request cannot be replayed against a different endpoint.

22.5.8.2. Security Properties

Proof-of-possession. Each API request with a DPoP token requires a fresh DPoP proof signed by the bound key. The server verifies that the proof's JWK thumbprint matches the token's `cnf.jkt` claim. An attacker who obtains only the token cannot generate valid proofs.

Replay protection. DPoP proofs include:

- `jti` (unique identifier): Servers SHOULD track recently-seen `jti` values and reject duplicates
- `iat` (issued-at timestamp): Servers SHOULD reject proofs with `iat` outside an acceptable window (e.g., ± 60 seconds)
- `nonce` (server nonce): Servers MAY require a server-issued nonce in proofs for additional replay protection

Method and URI binding. The `htm` (HTTP method) and `htu` (HTTP URI) claims in the proof are verified by the server. A proof for `GET /vouchers` cannot be used for `POST /vouchers` or for any other URI.

22.5.8.3. Limitations

DPoP does not protect against:

1. **Compromise of the DPoP private key.** If an attacker obtains both the token and the private key, they can generate valid proofs. Hardware-bound keys (TPM, secure enclave) mitigate this.
2. **Man-in-the-middle within the TLS session.** If an attacker can observe and modify traffic within the TLS session (e.g., by compromising the TLS termination point), they can intercept DPoP proofs and tokens together. TLS is still the primary defense.
3. **Compromised server.** If the server is compromised, the attacker can accept any proof or issue tokens bound to attacker-controlled keys.

22.5.8.4. Recommendations

- **Use DPoP for high-security deployments** where token theft is a significant concern
- **Use hardware-bound DPoP keys** (TPM, secure enclave) when available

- **Enable server nonces** (dpop_nonce) for additional replay protection in sensitive environments
- **Continue to rely on TLS** as the primary transport security mechanism — DPOP is defense-in-depth, not a replacement for TLS

22.5.9. Application-Layer Protections: Comparison with FDO TO2

Other FDO protocols — notably TO2 — include application-layer protections such as nonce chaining and hash continuity in every protocol message. This section explains why the Push, Pull, and Assign REST APIs do not require equivalent protections, and why the security properties of these APIs are not diminished by their absence.

22.5.9.1. Why TO2 Needs Application-Layer Protections

The TO2 protocol operates in a **fundamentally hostile environment**. A device may be connecting to a network for the first time. It does not know whether it is talking to the real Rendezvous Server or Owner — or to a man-in-the-middle. DNS may not be trustworthy. The device may not have a trusted CA store. The transport layer itself may be compromised or absent. In this context, application-layer nonces and hash chaining are essential: they are the device's only assurance that it is communicating with the correct party and that messages have not been tampered with in transit.

22.5.9.2. Why Voucher Management Does Not

The voucher management APIs in this specification operate in a **trusted-transport environment**. These are server-to-server connections between established business partners on production networks. Three conditions — all mandated by this specification — collectively eliminate the threats that TO2's application-layer protections address:

1. **We know who we are talking to.** TLS endpoint authentication verifies the server's identity before any application traffic is exchanged. This is established either through standard WebPKI certificate validation (the default) or through a pinned root CA provided during enrollment (see [§ 22.5.9.3 TLS Trust Models](#)). In both cases, the connecting party has cryptographic assurance of the server's identity at the transport layer — a man-in-the-middle cannot insert itself without being detected.
2. **They know who we are.** The FDOKeyAuth handshake (or alternative authentication) has proven the caller's identity, and the resulting bearer token scopes all subsequent operations to that identity. The server knows exactly who is making each request.
3. **The session is encrypted.** TLS provides confidentiality and integrity for all traffic. An attacker cannot observe, modify, or replay application-layer messages within the TLS session.

With all three conditions satisfied:

- **Man-in-the-middle** is prevented by TLS endpoint authentication — the attacker cannot impersonate the server or intercept the encrypted session
- **Replay** is prevented at the transport layer by TLS sequence numbers, and at the application layer by idempotency: duplicate voucher submissions return 409 Conflict, assignments are immutable (see [§ 12.7 Assignment Immutability](#)), downloads are idempotent, and status queries are read-only
- **Message injection or modification** is prevented by TLS record-layer integrity (MAC verification on every TLS record)
- **Session hijack** requires stealing the bearer token, which requires breaking TLS confidentiality or compromising an endpoint — failures against which no application-layer protocol can defend

Adding nonces or hash chaining to the REST API messages would not improve any of these properties. It would, however, destroy the architectural separation between cryptographic authentication and standard REST

integration (see [§ 13.3 Design Rationale: Architectural Separation](#)) — every API consumer would need to implement nonce generation, hash computation, and chaining logic, eliminating the ability to use standard REST tooling.

22.5.9.3. TLS Trust Models

The security argument above depends on TLS providing authentic endpoint identification. This specification supports two TLS trust models, each with distinct tradeoffs:

WebPKI (default). When no root CA is provided during enrollment, the connecting party validates the server's TLS certificate against the standard set of publicly-trusted Certificate Authorities. This is the same trust model used by web browsers and the vast majority of HTTPS clients.

- *Advantages:* Zero configuration — works immediately with any publicly-accessible endpoint. Universally supported. Certificate issuance and renewal can be automated (e.g., Let's Encrypt / ACME).
- *Disadvantages:* Relies on DNS integrity (the domain must resolve correctly) and the public CA ecosystem (any publicly-trusted CA can issue a certificate for any domain). Cannot authenticate endpoints on private networks, behind NAT, or using self-signed certificates. Vulnerable to DNS hijacking or CA compromise, though both are rare and detectable.

Pinned CA (via DID or separate enrollment). When a root CA certificate is provided during enrollment — either in the DID document's `tlsCertificateAuthority` field (see [§ 19.5.3 Optional TLS Certificate Authority](#)) or conveyed separately for non-DID enrollment — the connecting party validates the server's TLS certificate against that specific CA only.

- *Advantages:* Stronger assurance — the connecting party trusts exactly the CA the enrollment partner specified, not the entire public CA ecosystem. Works for private networks, self-signed certificates, and environments that do not participate in the public WebPKI. Eliminates DNS hijacking and public CA compromise as attack vectors for the TLS connection.
- *Disadvantages:* Requires conveying and managing the CA certificate as part of enrollment. CA certificate rotation requires re-enrollment or an out-of-band update. More operational burden than WebPKI.

For typical supply chain voucher management — cloud-hosted services, partner portals, internet-facing endpoints — **WebPKI is appropriate and sufficient**. For on-premises deployments, private infrastructure, or environments with heightened security requirements, **pinned CA provides additional assurance** at the cost of operational complexity. Both models satisfy the TLS endpoint authentication requirement that underpins the security of the REST APIs.

23. Implementation Guidelines

23.1. Versioning

Protocol versioning is handled through Service Root URLs (see [§ 9 Service Root URLs](#)), the `X-FD0-Version` header, and the `ProtocolVersion` field in `FD0KeyAuth.Hello` (see [§ 16.2.3 Message 1: FD0KeyAuth.Hello](#)). The Service Root URL MAY include a version component (e.g., `/api/v1/vouchers`) at the operator's discretion. Servers that wish to support multiple protocol versions simultaneously MAY use distinct Service Root URLs for each version. Servers SHOULD support at least one previous version during transition periods.

23.2. Protocol Extensibility

This specification is designed for forward and backward compatibility. Future versions of this protocol may add new fields to existing data structures. Both senders and receivers MUST follow the extensibility rules in this section to ensure interoperability across protocol versions.

23.2.1. General Principle

Implementations MUST follow Postel's Law: **be conservative in what you send, and liberal in what you accept**. When sending, implementations MUST conform to the structure defined in this specification and MUST NOT include fields not defined for the protocol version they are implementing. When receiving, implementations MUST tolerate the presence of fields they do not recognize and MUST NOT treat unrecognized fields as an error.

23.2.2. JSON Extensibility (REST APIs)

All JSON request and response bodies in the Push, Pull, Assign, and Status APIs are JSON objects (key-value maps). The following rules apply:

1. **Unknown fields MUST be ignored.** Implementations that receive a JSON object containing fields not defined in this specification MUST ignore those fields and MUST NOT treat their presence as an error. This allows future protocol versions to add new fields without breaking existing implementations.
2. **Field order is not significant.** JSON object fields MAY appear in any order. Implementations MUST NOT depend on a particular field ordering.
3. **Missing optional fields.** When a field defined as OPTIONAL in this specification is absent from a JSON object, implementations MUST interpret its absence according to the default specified for that field. When no default is specified, the field's absence means "not provided."
4. **New required fields in future versions.** A future protocol version that adds a required field MUST define a default value that previous-version receivers can safely assume when the field is absent, so that the new message remains intelligible to older implementations.

These rules apply equally to request bodies sent by clients and response bodies sent by servers.

23.2.3. CBOR Extensibility (FDOKeyAuth Messages)

The FDOKeyAuth authentication messages use CBOR arrays (positional structures) and one CBOR map (ServerInfo). CBOR arrays are inherently fragile with respect to extensibility — a strict parser that expects exactly N elements will reject an array of $N+1$ elements. The following rules close this gap:

23.2.3.1. CBOR Arrays

1. **Trailing elements MUST be ignored.** When a receiver decodes a CBOR array and finds more elements than defined for the message type in this specification, the receiver MUST ignore the trailing elements and MUST NOT treat their presence as an error. The receiver MUST process only the elements at positions it understands.
2. **Missing optional trailing elements.** When a CBOR array has fewer elements than the maximum defined for the message type, the receiver MUST treat the absent trailing elements as absent (not provided). Specifically, optional trailing fields (indicated by ? in the CDDL) are encoded by their absence — the array is simply shorter. Senders MUST NOT encode absent optional trailing elements as CBOR `null`; they MUST omit them entirely (producing a shorter array). For example, `FDOKeyAuth.Result` without `VoucherCount` is a 4-element array, not a 5-element array with `null` in the fifth position.
3. **Required elements MUST be present.** All elements not marked ? in the CDDL MUST be present at their defined positions. A receiver that cannot find a required element at its expected position MUST reject the message.
4. **Future versions append fields.** Future versions of this protocol MUST add new fields only at the end of existing CBOR arrays, after all fields defined in the current version. Future versions MUST NOT insert fields

between existing positions or change the meaning of existing positions. This ensures that a current-version receiver can parse the fields it understands by position, ignoring any trailing additions.

5. **Future versions and defaults.** When a future version adds a trailing field, it SHOULD specify the default interpretation for receivers that do not recognize the field (i.e., receivers implementing an earlier version of the protocol). The default MUST be chosen so that the older receiver's behavior remains correct or safely degraded.

23.2.3.2. CBOR Maps

1. **Unknown keys MUST be ignored.** When a receiver decodes a CBOR map (e.g., `ServerInfo`) and finds keys not defined in this specification, the receiver MUST ignore those key-value pairs and MUST NOT treat their presence as an error.
2. **Key order is not significant.** CBOR map entries MAY appear in any order. Implementations MUST NOT depend on a particular key ordering for map semantics. (Note: for deterministic encoding purposes, keys MUST be sorted per [RFC8949](#) Section 4.2, but this is an encoding constraint, not a semantic one.)

23.2.3.3. Signed Payloads

The extensibility rules above apply to the **outer message structures** (`FD0KeyAuth.Hello`, `FD0KeyAuth.Challenge`, `FD0KeyAuth.Prove`, `FD0KeyAuth.Result`). The **signed payloads** (`FD0KeyAuthChallengeSignedPayload`, `FD0KeyAuthProveSignedPayload`) are subject to the same array extensibility rules — trailing elements beyond those defined in this version MUST be ignored by verifiers. Critically, signature verification and hash continuity operate on **raw CBOR bytes**, not on parsed structures: the signer serializes the payload (which may include fields unknown to the verifier), signs the serialized bytes via `COSE_Sign1`, and the verifier validates the signature over the same bytes. The verifier then parses the verified payload and extracts the fields it understands. This means additional fields in signed payloads do not break signature verification.

23.2.3.4. Version Negotiation

The `ProtocolVersion` field in `FD0KeyAuth.Hello` identifies the protocol version the Caller supports. When a Server receives a `ProtocolVersion` value it does not support, it MUST respond with `HTTP400 Bad Request` and SHOULD include a response body (JSON or CBOR) indicating the supported version(s). A Server MUST NOT silently downgrade to a lower protocol version — version mismatches MUST be reported as errors so the Caller can retry with an appropriate version or implementation.

23.2.4. CBOR Deterministic Encoding

The `FD0KeyAuth` protocol relies on hash continuity between messages: each message includes a hash of the preceding message's CBOR-encoded body. This mechanism requires that the same logical message always produces the same byte sequence. Implementations MUST use CBOR Preferred Serialization as defined in [RFC8949](#) Section 4.2.1 for all `FD0KeyAuth` messages. In particular:

- Integers MUST use the shortest encoding
- Lengths (for `bstr`, `tstr`, arrays, maps) MUST use the shortest definite-length encoding
- Map keys MUST be sorted in bitwise lexicographic order of their encoded forms
- Floating-point values (if any) MUST use the shortest encoding that preserves the value

Senders that do not use deterministic encoding risk hash mismatches and protocol failure. Receivers SHOULD verify that received CBOR is deterministically encoded and MAY reject messages that are not.

23.3. Content Negotiation§

Servers MUST return `application/json` [\[RFC8259\]](#) for all non-file responses. Servers MUST return `application/x-fdo-voucher` for voucher file downloads.

23.4. Idempotency§

The POST endpoint MAY return 409 Conflict for duplicate voucher submissions. Implementations SHOULD use the voucher GUID as the deduplication key.

23.5. Pagination Best Practices§

- Continuation tokens SHOULD be opaque to clients
- Continuation tokens SHOULD expire after a reasonable period
- Servers SHOULD support `since` parameter for checkpoint-based retrieval
- Servers SHOULD return `total_count` when feasible

24. Future Enhancements§

- **Batch Transfer:** Submit multiple vouchers in a single request
- **WebSocket Support:** Real-time transfer status updates
- **Event-Driven Architecture:** Message queue integration for high-volume deployments
- **Standardized MIME Type Registration:** Formal IANA registration of `application/x-fdo-voucher`
- **Additional Delegate ECU Permissions:** Formal registration of `fdo-ekt-permit-voucher-upload` (PERM.6) for push-side delegate authorization, complementing the `fdo-ekt-permit-voucher-claim` (PERM.5) defined in this specification
- **Pull Authentication Token Refresh:** Protocol extension to allow session token renewal without full re-authentication, reducing handshake overhead for long-running pull sessions
- **Selective Voucher Claim:** Allow Delegates with scoped permissions to claim only vouchers matching specific criteria (e.g., device model, serial prefix) rather than all vouchers for an Owner Key
- **Push Challenge-Response Authentication:** ~~Now addressed by the unified FDOKeyAuth protocol (see [§ 16 FDOKeyAuth Protocol](#)), which provides the same cryptographic challenge-response handshake for both push and pull endpoints.~~

25. Conclusion§

The FDO Voucher Transfer Protocol provides a standardized, secure method for transferring ownership vouchers between manufacturing systems and owner systems. Its security model is founded on a single prerequisite — the mutual exchange of DIDs (short identifier strings) between trusted parties — from which all subsequent authentication, authorization, and endpoint discovery follows without additional credential provisioning (see [§ 19.1 Trust Foundation: Mutual DID Exchange](#)).

The protocol defines three components: the **Push API** for supplier-initiated voucher delivery, the **Pull API** for owner-initiated voucher retrieval, and the **Owner Key Authorization** protocol (FDOKeyAuth) that provides cryptographic authentication for both. FDOKeyAuth is modeled after the TO2 protocol handshake in [\[FDO-Specification\]](#) and uses the same enrolled keys from DID exchange — the Supplier Key for push, the Owner Key for pull — eliminating the need for separate credential provisioning. The resulting bearer token integrates naturally with standard API gateways and infrastructure. Support for Delegate Certificates allows authorized delegates to act on

behalf of a key owner using the `fdo-ekt-permit-voucher-claim` and `fdo-ekt-permit-voucher-upload` ECU permissions, enabling enterprise-scale deployments where multiple sites push or pull vouchers independently.

Deployments that require additional protection beyond the DID-based model — such as API gateway rate limiting, WAF integration, or purchase-order-tied quotas — MAY layer mutual TLS or business logic validation on top of the core protocol (see [Appendix A: Optional Security Layers \(Informative\)](#)). These additive mechanisms are out of scope for the core design and require provisioning beyond the simple exchange of DIDs.

Appendix A: Optional Security Layers (Informative)

The protocol-native authentication described in [§ 17.1 How Authentication Works](#) is sufficient for secure voucher transfer. The mechanisms in this appendix are **additive, defense-in-depth layers** that deployments MAY use to address operational concerns such as DDoS mitigation at API gateways, WAF integration, rate limiting, or business-process enforcement. They are not required for protocol security and require provisioning beyond the simple exchange of DIDs.

Token-Based Authorization

Overview

The Voucher Recipient issues scoped tokens to authorized Voucher Issuers. Tokens are validated at the HTTP layer before any voucher processing occurs. This is useful for fast rejection at edge infrastructure (CDN, WAF) and for per-Supplier rate limiting.

Token Format

Tokens SHOULD use the JWT format [\[RFC7519\]](#). The following claims are defined:

```
{
  "iss": "https://recipient.example.com",
  "sub": "manufacturer-123",
  "aud": "https://recipient.example.com/fdo/v1/vouchers",
  "exp": 1640995200,
  "iat": 1640908800,
  "scope": "voucher:submit",
  "limits": {
    "max_vouchers": 100,
    "allowed_models": ["model-a", "model-b"],
    "allowed_serials": ["prefix-*"]
  },
  "manufacturer_id": "mfg-abc123"
}
```

Token Types

- **Time-Limited Tokens:** Valid for a specified duration (e.g., 24 hours, 30 days)
- **Quota-Limited Tokens:** Maximum number of vouchers allowed
- **One-Time Use Tokens:** Single voucher submission, auto-expire after use
- **Device-Specific Tokens:** Valid only for specific device serial numbers or models
- **Purchase-Tied Tokens:** Generated during the purchase process, tied to a specific order

Validation§

1. Client includes Authorization: Bearer <token> header
2. Server (or WAF) validates JWT signature and claims
3. Server enforces token limits and scope
4. Token quota is decremented (for quota-limited tokens)

Mutual TLS (mTLS) Authentication§

Overview§

The Voucher Recipient enrolls manufacturer certificates for mutual TLS authentication. All voucher submissions require a successful mTLS handshake.

Considerations§

- **Medium Resource Usage:** TLS handshake requires CPU but less than voucher parsing
- **Connection Filtering:** Invalid certificates are rejected at the TLS layer
- **WAF Limitations:** mTLS validation typically requires TLS termination at the application, limiting edge filtering

Enrollment Process§

1. Voucher Issuer provides public key or CA certificate to Voucher Recipient
2. Voucher Recipient adds certificate to trusted manufacturer store
3. Voucher Issuer configures client with corresponding private key
4. All voucher submissions require mTLS handshake

Business Logic Validation§

Overview§

Application-level validation based on business rules and order data. This provides the deepest validation but has the highest resource cost and depends on external systems (CRM, order management, device registries).

Validation Types§

- **Order Number Validation:** Voucher must reference a valid purchase order
- **Device Registration:** Device must be pre-registered in the Voucher Recipient system
- **Customer Validation:** Voucher must be for a registered customer
- **Quantity Limits:** Enforce order quantities and limits

Token vs mTLS Comparison§

Aspect	JWT Tokens	mTLS Certificates
Validation Speed	Very fast (signature check)	Medium (TLS handshake)

Aspect	JWT Tokens	mTLS Certificates
WAF Support	Excellent (edge validation)	Limited (requires TLS termination)
Revocation	Immediate (token blacklist)	Delayed (CRL/OCSP)
Scope Flexibility	High (claims-based)	Low (certificate-based)
Key Rotation	Easy (new tokens)	Complex (certificate reissuance)
Third-Party Integration	Excellent	Poor

Purchase Process Integration§

Deployments that use token-based authorization as an additive layer MAY integrate token generation with their purchasing workflow.

Token Generation During Purchase§

1. Customer purchases devices through ordering system
2. Ordering system generates manufacturer-specific JWT tokens
3. Tokens include purchase order details, device counts, and expiration
4. Tokens are provided to the Voucher Issuer along with DID information
5. Voucher Issuer includes tokens when submitting vouchers

Purchase-Tied Token Format§

```
{
  "iss": "https://ordering.example.com",
  "sub": "manufacturer-123",
  "aud": "https://owner.example.com/fdo/v1/vouchers",
  "exp": 1640995200,
  "iat": 1640908800,
  "purchase_order": "P0-12345",
  "customer_id": "cust-abc123",
  "device_counts": {
    "model-a": 100,
    "model-b": 50
  },
  "manufacturer_id": "mfg-abc123"
}
```

Appendix B: Voucher Recovery When the Ownership Voucher Is Not Forwarded (Informative)

The Problem§

The main body of this specification assumes that every entity in the supply chain is an FDO participant — each hop has a party that receives the voucher, signs it over to the next party, and delivers it onward. In practice, this assumption frequently does not hold.

Consider a common scenario: a factory manufactures an FDO-enabled device and holds the ownership voucher. The factory sells the device to Distributor D, who sells it to Reseller R, who sells it to End-User E. Neither D nor R

is an FDO participant. They have no FDO keys, no voucher management infrastructure, and no awareness that the device carries an FDO identity at all. They are simply moving boxes through a supply chain.

When E unpacks the device and attempts FDO onboarding, E discovers there is no ownership voucher — nobody signed it over. The voucher is still held by the original manufacturer (or possibly by an FDO-aware intermediary early in the chain). E needs a way to go back to the voucher's current Key Holder and claim the device.

This appendix describes a **voucher claim** process: a lightweight mechanism by which a party in physical possession of an unclaimed device can request that the Key Holder sign the voucher over to them. This process is an at-most-once exception recovery path — it is not intended to replace the normal enrollment and transfer protocols defined in the main specification.

Relationship to Voucher Assignment§

The Voucher Assignment mechanism ([§ 12 Voucher Assignment](#)) addresses a related but distinct scenario: a Custodian (reseller B) who has a business relationship with the Key Holder (manufacturer A) directs A to sign a voucher over to a designated party (customer C). Assignment requires B to have an authenticated relationship with A and to know C's identity.

Voucher claim, by contrast, addresses the case where **no intermediary acts at all**. E arrives at the Key Holder's door with nothing but a device in hand and says: "I have this device; please give me the voucher." The Key Holder needs some mechanism to verify that E actually possesses the device before releasing the voucher.

If a Custodian relationship exists between an intermediary and the Key Holder, the Assign API [§ 12.5 Assign Endpoint](#) is the appropriate mechanism. This appendix applies when that relationship does not exist or was never exercised.

Design Principles§

The voucher claim process is guided by the following principles:

Minimize Manufacturer Burden§

Device manufacturers operate established, often inflexible manufacturing processes. Many are general-purpose hardware OEMs — they assemble boards into chassis, solder components, flash firmware images, and apply labels. They typically do not control what operating system or application software runs on the device after it leaves the factory. A voucher claim process that requires manufacturers to modify boot sequences, install interactive software, or generate cryptographic artifacts at claim-time would impose impractical burdens on many manufacturers.

The claim process **SHOULD** require no more from the manufacturer than what they already do: apply a label or sticker to the device during manufacturing.

Don't Prescribe the Mechanism§

Different manufacturers have different capabilities, different business relationships with their customers, and different security postures. A manufacturer that already operates a customer portal with authenticated accounts has different options than a small OEM that sells through distributors and has no direct relationship with end users.

This appendix describes the **goals** of a claim process and provides **recommended approaches**, but it does not mandate a specific mechanism. The Key Holder is ultimately responsible for deciding how much verification they require before releasing a voucher, and what user experience they provide.

Proof of Possession, Not Proof of Ownership§

The claim process establishes that the claimant has **physical possession** of the device. It does not — and cannot — establish that the claimant is the device's rightful owner. Proof of purchase, proof of ownership, and other business-level verifications are the Key Holder's prerogative to require or waive. This specification concerns itself only with the minimum: giving the Key Holder enough confidence that the claimant holds the device.

At-Most-Once§

A voucher claim is an at-most-once operation. If a voucher has already been signed over via the normal transfer protocol, or has already been claimed, or has been assigned via the Assign API, the claim **MUST** be rejected. This prevents a second party from claiming a device whose voucher has already been issued.

The Claim Code§

The central element of the claim process is a **claim code**: a value that is associated with a specific device, is known to the Key Holder, and is available to someone in physical possession of the device. The claim code serves as proof of physical possession.

Requirements§

A claim code **MUST** satisfy two properties:

1. **Device-specific**: The claim code uniquely identifies a device (or maps to a device identifier that the Key Holder can look up).
2. **Not easily guessable**: The claim code has sufficient entropy that an attacker cannot enumerate valid claim codes through brute force or inference.

The second property is critical. If the claim code is simply a device serial number, and serial numbers are sequential (e.g., SN-001, SN-002, SN-003), an attacker can trivially enumerate all devices and claim vouchers for devices they do not possess. Serial numbers are typically designed for inventory tracking and human readability, not for security.

Recommended Approach: Random Claim Code§

Manufacturers **SHOULD** generate a random claim code for each device — for example, an alphanumeric string of at least 20 characters or a UUID — and record the mapping between the claim code and the device's identity (serial number, GUID, etc.) in their systems.

If the manufacturer's existing serial numbers already have sufficient entropy and unpredictability (e.g., cryptographic random identifiers), they **MAY** serve as claim codes without a separate value. However, most real-world serial number schemes do not meet this bar, and manufacturers are strongly advised to use a dedicated claim code.

What About OV-Derived Claim Codes?§

An earlier proposal for this mechanism suggested computing the claim code from Ownership Voucher data — for example, hashing the device's public key together with a signed nonce. While this approach has attractive anti-guessing properties (the claim code is cryptographically bound to the specific device identity), it imposes a significant manufacturing burden:

- **Just-in-time printing**: The claim code depends on data generated during the Device Initialize (DI) protocol,

which means stickers or labels containing the claim code must be printed on the manufacturing line *during or after* the DI process. Many manufacturers pre-print label rolls or use generic sticker stock — requiring on-demand label generation tied to DI output may require new equipment, new software integration, and changes to manufacturing line sequencing.

- **Software integration complexity:** The claim code computation requires access to the device's private key (for signing) and the ownership voucher data. This means the label printing system must integrate with the DI infrastructure, creating a coupling between physical labeling and cryptographic provisioning that does not otherwise exist.

While manufacturers with sophisticated integrated manufacturing systems MAY choose to derive claim codes from OV data, this specification does not require it. A random claim code, recorded in the manufacturer's database alongside the device record, provides equivalent anti-guessing properties without coupling label production to the DI process.

Conveying the Claim Code to the Device Recipient

The claim code must be available to someone in physical possession of the device. This appendix describes several conveyance mechanisms, ordered from most recommended to least.

QR Code on a Sticker (Recommended)

The manufacturer affixes a sticker bearing a QR code to the device (or its packaging, or a card included in the box). The QR code encodes a URL that directs the user to the Key Holder's claim portal and includes the claim code as a URL parameter.

Example QR code URL:

```
https://devices.acme-mfg.example.com/claim?code=K7xM2pQr9vNw4TjL8sFh
```

This is the recommended approach because:

- **Minimal manufacturing change:** If the manufacturer already applies per-device serial number stickers, adding a QR code sticker is a minor incremental change. The QR code can share the same label as the serial number.
- **Frictionless user experience:** The end user scans the QR code with a phone, is taken to a web page, enters their DID (or other credentials), and receives their voucher. No command-line tools, no reading hex strings, no special software.
- **No device power-on required:** The user does not need to connect the device to power, attach a monitor, or boot an operating system. This preserves the "zero-touch" property — especially important for headless edge devices, network equipment, and IoT sensors that may have no display or human interface at all.

Static QR Code with Manual Identifier Entry

If per-device QR code stickers are impractical (e.g., the manufacturer uses generic packaging), a static QR code — identical on every device — can direct the user to the claim portal without encoding a device-specific claim code. The user then manually enters a device identifier (serial number, claim code printed elsewhere on the device, etc.) on the web page.

Example static QR URL:

```
https://devices.acme-mfg.example.com/claim
```

This approach requires the user to type in an identifier, which is slightly more effort, but it allows the manufacturer to use stock stickers or packaging that do not vary per device. The per-device claim code (or serial number plus separate claim code) would still need to appear somewhere on the device or its documentation.

Printed Card or Documentation§

A card, insert, or page in the device's documentation can contain the claim code (and optionally a QR code). This is less secure than a sticker on the device itself because cards can be separated from the device — a card found loose does not prove possession of the device. However, for many deployment scenarios, this is acceptable.

On-Screen Display at Boot (Not Recommended)§

An earlier proposal suggested that the device display a proof-of-possession string on a screen when first booted. This approach has significant practical problems:

- **Requires physical interaction:** The user must connect the device to power, attach a display (and possibly a keyboard), and boot the device. For headless edge devices, IoT sensors, or rack-mount servers without video output, this may be physically impossible without special equipment.
- **Negates "zero-touch":** FDO's value proposition is that devices can be onboarded without manual configuration at the deployment site. Requiring the user to boot a device, read a code off a screen, and type it into a portal undermines this value.
- **Manufacturer control:** The device manufacturer typically does not control the software that runs on the device. A general-purpose server shipping with a Linux distribution (RHEL, Ubuntu, etc.) has a boot sequence controlled by the OS vendor, not the hardware OEM. Inserting a "pause and display claim code" step into an arbitrary OS boot process is a non-trivial software integration that most hardware manufacturers cannot perform.
- **One-time boot state complexity:** If the code is only displayed on first boot, the software must reliably detect "first boot" and manage state transitions. If the user misses the code or the display is not connected, recovery becomes problematic.

Manufacturers MAY offer on-screen display as an **additional** conveyance mechanism for devices that have a display, but it SHOULD NOT be the sole or primary mechanism.

The Claim Process§

The claim process is intentionally described at a high level. The Key Holder's implementation of the claim portal, the specific user experience, and any additional verification steps are at the Key Holder's discretion.

Minimum Viable Claim Flow§

1. **Claimant obtains claim code:** The claimant scans a QR code on the device (or reads the claim code from a sticker, card, or other source).
2. **Claimant presents claim code and DID:** The claimant navigates to the Key Holder's claim portal (via the QR URL or a known URL) and provides:
 - The claim code (or device identifier, if the QR code did not encode it directly)
 - The claimant's DID (or public key in another supported format — see [§ 20 Key Formats and Enrollment Considerations](#))
3. **Key Holder verifies the claim code:** The Key Holder looks up the claim code in its records and verifies:
 - The claim code is valid and corresponds to a known device
 - The voucher for that device has not already been signed over, assigned, or claimed

4. **Key Holder extends the voucher.** The Key Holder signs the voucher over to the claimant's public key (obtained from the DID) using the standard voucher extension mechanism.
5. **Key Holder delivers the voucher.** The voucher is delivered to the claimant via any suitable mechanism — pushed to the claimant's endpoint (if the DID includes a push URL), made available for pull, provided for download on the portal, or sent via email.

Optional Additional Verification§

The Key Holder MAY require additional verification before releasing a voucher. This is entirely at the Key Holder's discretion and may include:

- **Account authentication:** Requiring the claimant to log into an account on the Key Holder's portal before claiming. This is particularly useful for manufacturers that sell direct-to-customer and maintain customer accounts.
- **Email or phone verification:** Sending a one-time code to an email address or phone number provided by the claimant.
- **Purchase order or invoice number:** Asking the claimant to provide a purchase reference that the Key Holder can validate against its sales records.
- **Identity verification:** For high-value or security-sensitive devices, the Key Holder may require additional identity proofs.

These additional verification steps improve confidence that the claimant is the legitimate owner of the device, not merely someone who has momentary physical access or who guessed the claim code. However, they add friction to the process and require the Key Holder to maintain additional infrastructure (customer accounts, email verification systems, order databases, etc.). The appropriate level of verification is a business decision that depends on the value of the devices, the Key Holder's risk tolerance, and the Key Holder's existing infrastructure.

Security Considerations for Voucher Recovery§

Claim Code Entropy§

The claim code is the primary defense against unauthorized voucher claims. If an attacker can guess or enumerate valid claim codes, they can claim vouchers for devices they do not possess. Key Holders SHOULD ensure that claim codes have at least 128 bits of entropy. A 22-character base62 string, a UUID v4, or a 32-character hex string all satisfy this requirement.

Key Holders MUST NOT use sequential serial numbers, MAC addresses, or other easily enumerable values as the sole claim code.

Rate Limiting§

Key Holders SHOULD implement rate limiting on the claim endpoint to prevent brute-force enumeration of claim codes. Standard measures include:

- Per-IP rate limiting
- Per-session rate limiting
- CAPTCHA or proof-of-work challenges after repeated failures
- Exponential backoff on failed attempts

Claim Notification§

Key Holders MAY notify known contacts (the original purchaser, if known) when a voucher is claimed. This provides an out-of-band detection mechanism for fraudulent claims.

Revocation§

This specification does not define a voucher revocation mechanism. If a voucher is claimed fraudulently, the Key Holder's recourse is the same as for any mis-issued voucher: the legitimate owner must contact the Key Holder through an out-of-band process to resolve the situation. The at-most-once claim constraint (see [At-Most-Once](#)) limits the blast radius — a fraudulently claimed voucher blocks the legitimate owner from claiming, which serves as a detection signal.

Rationale: Why Not the Earlier Proposal?§

An earlier draft of this appendix proposed a more prescriptive claim process. The following discusses why certain elements of that proposal were modified or removed.

Prescribed Proof-of-Possession Computation§

The earlier proposal specified a specific PoP string computed as `asSHA256<DevicePublicKey, Enc{DevicePrivateKey, <Guid, nonce>}>`. While cryptographically sound, this couples the claim process to the device's DI-time cryptographic identity and requires:

- The labeling system to have access to the device's private key for signing
- The claim code to be generated during or after DI, precluding pre-printed labels
- The claimant (or intermediary) to relay a 64-character hex string, which is error-prone for manual entry or phone-based support

A random claim code achieves the same goal — proving possession of a device-associated secret — without coupling to the DI process or requiring specific cryptographic computation.

Transmission via Email or Fax§

The earlier proposal suggested the claimant send claim information to the Key Holder via "email or fax." While the intent (any out-of-band channel) was sound, specifying particular communication media is unnecessary and dated. A web-based claim portal provides a better user experience, integrates directly with the Key Holder's systems, and eliminates manual processing. The claim process described in this appendix is transport-agnostic — the Key Holder provides whatever interface makes sense for their operations.

Mandatory On-Screen Display§

The earlier proposal listed device display as the primary method for conveying the PoP string (with serial device, LCD, and shipped card as alternatives). As discussed in [On-Screen Display at Boot \(Not Recommended\)](#), on-screen display is problematic for the majority of FDO-targeted devices. A sticker or QR code is far more practical and does not require powering on the device or controlling its boot sequence.

Single Prescribed Flow§

The earlier proposal described one specific flow: device displays PoP, user copies it, user sends to manufacturer, manufacturer verifies, manufacturer extends voucher. This appendix instead describes the *goals* (prove possession, present DID, receive voucher) and lets the Key Holder implement whatever flow achieves them. This flexibility is essential because:

- Different manufacturers have vastly different capabilities and customer relationships
- The "right" flow depends on factors this specification cannot anticipate (manufacturer's existing systems, device type, deployment scenario, security requirements)
- Over-prescribing the flow risks making it impractical for manufacturers whose processes don't fit the prescribed model, which would result in the mechanism being ignored entirely

References

Non-Normative References

[DID-Core]

W3C. *Decentralized Identifiers (DIDs) v1.0*. URL: <https://www.w3.org/TR/did-core/>

[FDO-Specification]

FIDO Alliance. *FIDO Device Onboard Specification, Version 2.0* URL: <https://fidoalliance.org/specs/FDO/FIDO-Device-Onboard-PS-v2.0-20260402/FIDO-Device-Onboard-PS-v2.0-20260402.html>

[JSON-LD]

Manu Sporny; Gregg Kellogg; Markus Lanthaler. *JSON-LD 1.0*. 3 November 2020. REC. URL: <https://www.w3.org/TR/json-ld/>

[RFC1951]

P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. May 1996. Informational. URL: <https://www.rfc-editor.org/info/rfc1951/>

[RFC2119]

S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels* URL: <https://tools.ietf.org/html/rfc2119>

[RFC6713]

J. Levine. *The 'application/zlib' and 'application/gzip' Media Types*. August 2012. Informational. URL: <https://www.rfc-editor.org/info/rfc6713/>

[RFC7468]

S. Josefsson. *Textual Encodings of PKIX, PKCS, and CMS Structures* April 2015. Standards Track. URL: <https://tools.ietf.org/html/rfc7468.html>

[RFC7519]

M. Jones; J. Bradley; N. Sakimura. *JSON Web Token (JWT)*. URL: <https://tools.ietf.org/html/rfc7519>

[RFC7578]

L. Masinter. *Returning Values from Forms: multipart/form-data*. URL: <https://tools.ietf.org/html/rfc7578>

[RFC7638]

M. Jones; N. Sakimura. *JSON Web Key (JWK) Thumbprint*. URL: <https://tools.ietf.org/html/rfc7638>

[RFC8259]

T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format* URL: <https://tools.ietf.org/html/rfc8259>

[RFC8446]

E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3* URL: <https://tools.ietf.org/html/rfc8446>

[RFC8610]

H. Birkholz; C. Viganò; C. Bormann. *Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures*. URL: <https://tools.ietf.org/html/rfc8610>

[RFC8949]

C. Bormann; P. Hoffman. *Concise Binary Object Representation (CBOR)*. URL: <https://tools.ietf.org/html/rfc8949>

[RFC9052]

J. Schaad. *CBOR Object Signing and Encryption (COSE): Structures and Process* URL: <https://tools.ietf.org/html/rfc9052>

[RFC9449]

D. Fett; et al. *OAuth 2.0 Demonstrating Proof of Possession (DPoP)* URL: <https://tools.ietf.org/html/rfc9449>

