ONVIF™
Security Service Specification

Version 25.06

June 2025

CONTENTS

1 Scope 6

2 Normative References 6

3 Terms and Definitions 8

3.1 Definitions 8

3.2 Abbreviations 8

3.3 Namespace 9

4 Overview 9

4.1 General Structure 9

4.2 Certificate-based Client Authentication 10

4.3 Certificate Path Verification 10
4.3.1 Certification Path Validation Algorithm Parameters 10
4.3.2 Revocation Status Checking 10
4.3.3 Trust Anchors 11
4.3.4 Certificate Repository for constructing Certification Paths 11
4.3.5 Specific certification path validation parameters 11

4.4 JWT-based client authorization 11
4.4.1 General 11
4.4.2 Configuration 12
4.4.3 Usage of JWT-based client authentication over HTTP 12
4.4.4 Usage of JWT-based client authentication over HTTPS 12
4.4.5 Usage of JWT-based client authentication over SCTP 13
4.4.6 Usage of JWT-based client authentication over RTSP 13
4.4.7 Usage of JWT-based client authentication over RTSPS 13

4.5 IEEE 802.1X 13

4.6 Authorization Servers 13
4.6.1 Device authentication and authorization 13
4.6.2 User authentication and authorization 14
4.6.3 Authorization server configuration 15

5 Security Configuration Service 16

5.1 General Structure 16

5.2 Keystore 17
5.2.1 Elements of the Keystore 17
5.2.2 Unique Identifiers 17
5.2.3 Uniqueness of Objects in the Keystore 18
5.2.4 Referential Integrity 18
5.2.5 Key Status 19
5.2.6 Keystore Operations 19
5.2.6.1 Passphrase Management 19
5.2.6.2 Key Management 21
5.2.6.3 Certificate Management 26
5.2.6.4 CRL Management 39
5.2.6.5 Certification Path Validation Policy Management 41

5.3 TLS Server 44
5.3.1 Elements of the TLS Server 44
5.3.2 Authorization of TLS authenticated connections 44
5.3.3 TLS Server Operations 45
5.3.3.1 AddServerCertificateAssignment 45

5.3.3.2 RemoveServerCertificateAssignment 46
5.3.3.3 ReplaceServerCertificateAssignment 47
5.3.3.4 GetAssignedServerCertificates 48
5.3.3.5 SetClientAuthenticationRequired 48
5.3.3.6 GetClientAuthenticationRequired 48
5.3.3.7 SetCnMapsToUser 49
5.3.3.8 GetCnMapsToUser 49
5.3.3.9 AddCertPathValidationPolicyAssignment 49
5.3.3.10 RemoveCertPathValidationPolicyAssignment 50
5.3.3.11 ReplaceCertPathValidationPolicyAssignment 50
5.3.3.12 GetAssignedCertPathValidationPolicies 51
5.3.3.13 SetEnabledTLSVersions 51
5.3.3.14 GetEnabledTLSVersions 52

5.4 IEEE 802.1X 52
5.4.1 AddDot1XConfiguration 52
5.4.2 GetAllDot1XConfigurations 54
5.4.3 GetDot1XConfiguration 54
5.4.4 DeleteDot1XConfiguration 54
5.4.5 SetNetworkInterfaceDot1XConfiguration 55
5.4.6 GetNetworkInterfaceDot1XConfiguration 56
5.4.7 DeleteNetworkInterfaceDot1XConfiguration 56

5.5 Media Signing 57
5.5.1 Overview 57
5.5.2 AddMediaSigningCertificateAssignment 57
5.5.3 RemoveMediaSigningCertificateAssignment 58
5.5.4 GetAssignedMediaSigningCertificates 58

5.6 Autorization Server Configuration 59
5.6.1 GetAuthorizationServerConfigurations 59
5.6.2 CreateAuthorizationServerConfiguration 59
5.6.3 SetAuthorizationServerConfiguration 59
5.6.4 DeleteAuthorizationServerConfiguration 60

5.7 JWT-based authentication 60
5.7.1 GetJWTConfiguration 60
5.7.2 SetJWTConfiguration 61

5.8 Capabilities 61
5.8.1 GetServiceCapabilities 61
5.8.2 Keystore Capabilities 62
5.8.3 TLS Server Capabilities 63
5.8.4 IEEE 802.1X Capabilities 64
5.8.5 Media Signing Capabilities 64
5.8.6 Authorization Server Capabilities 64
5.8.7 Capability-implied Requirements 65

5.9 Events 70
5.9.1 Key Status 70

5.10 Service specific data types 70

6 Security Considerations 70

7 Design Rationale 72

7.1 General Design Goals 72

7.2 Keystore 72

7.3 TLS Server 73

Annex A JWT Content example 74

Annex B JWT over SCTP example 75

Annex C JWT over HTTPS example 76

Annex D JWT over RTSPS example 77

Annex E Cipher Reference (Informative) 78

Annex F Revision History 79

1 Scope

This document defines the web service interface for ONVIF security configuration features such as a keystore and a TLS server on an ONVIF device.

Web service usage is outside of the scope of this document. Please refer to the ONVIF core specification.

2 Normative References

Basic Security Profile Version 1.1 Committee Specification 01, OASIS Standard, 22 October 2004

<https://docs.oasis-open.org/ws-brsp/BasicSecurityProfile/v1.1/cs01/BasicSecurityProfile-v1.1-cs01.pdf>

IANA TLS Supported Groups, Elliptic curve groups

<https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-8>

IEEE 802.1X, Port-Based Network Access Control

<http://standards.ieee.org/getieee802/download/802.1X-2004.pdf>

ONVIF Core Specification

<http://www.onvif.org/specs/core/ONVIF-Core-Specification.pdf>

ONVIF Media Signing Specification

<http://www.onvif.org/specs/stream/ONVIF-MediaSigning-Specification.pdf [http://www.onvif.org/specs/core/ONVIF-Core-Specification.pdf]>

IETF RFC 2246 The TLS Protocol Version 1.0

<http://www.ietf.org/rfc/rfc2246.txt>

IETF RFC 2898 PKCS#5 Password-based Cryptography Specification v2.0

<http://www.ietf.org/rfc/rfc2898.txt>

IETF RFC 2986 PKCS #10: Certification RequestSyntaxSpecification Version 1.7

<http://www.ietf.org/rfc/rfc2986.txt>

IETF RFC 3279 Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

<http://www.ietf.org/rfc/rfc3279.txt>

IETF RFC 3447 Public Key Cryptography Standards #1: RSA Cryptography Specifications Version 2.1

<http://www.ietf.org/rfc/rfc3447.txt>

IETF RFC 4055 Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

<http://www.ietf.org/rfc/rfc4055.txt>

IETF RFC 4346 The Transport Layer Security (TLS) Protocol Version 1.1

<http://www.ietf.org/rfc/rfc4346.txt>

IETF RFC 5208 Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax Specification v1.2

<http://www.ietf.org/rfc/rfc5208.txt>

IETF RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2

<http://www.ietf.org/rfc/rfc5246.txt>

IETF RFC 8422 Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier

<https://www.ietf.org/rfc/rfc8422.txt>

IETF RFC 5280 Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

<http://www.ietf.org/rfc/rfc5280.txt>

IETF RFC 5958 Asymmetric Key Packages

<http://www.ietf.org/rfc/rfc5958.txt>

IETF RFC 5959 Algorithms for Asymmetric Key Package Content Type

<http://www.ietf.org/rfc/rfc5959.txt>

IETF RFC 6749 The OAuth 2.0 Authorization Framework

<http://www.ietf.org/rfc/rfc6749.txt>

IETF RFC 6750 The OAuth 2.0 Authorization Framework: Bearer Token Usage

<http://www.ietf.org/rfc/rfc6750.txt>

IETF RFC 7517 JSON Web Key (JWK)

<http://www.ietf.org/rfc/rfc7517.txt>

IETF RFC 7518 JSON Web Algorithms (JWA)

<http://www.ietf.org/rfc/rfc7518.txt>

IETF RFC 7519 JSON Web Token (JWT)

<http://www.ietf.org/rfc/rfc7519.txt>

IETF RFC 7643 System for Cross-domain Identity Management: Core Schema

<http://www.ietf.org/rfc/rfc7643.txt>

IETF RFC 8414 OAuth 2.0 Authorization Server Metadata

<http://www.ietf.org/rfc/rfc8414.txt>

IETF RFC 8705 OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens

<http://www.ietf.org/rfc/rfc8705.txt>

Unified Modeling Language (UML)

<http://www.omg.org/spec/UML>

OpenID Connect Core

<https://openid.net/specs/openid-connect-core-1_0.html>

OpenID Connect Discovery

<https://openid.net/specs/openid-connect-discovery-1_0.html>

PKCS#5 Password-based Encryption Standard v1.5, RSA Laboratories, 1993

PKCS#12: Personal Information Exchange Syntax v1.0, RSA Laboratories, 1999

3 Terms and Definitions

3.1 Definitions

Alias	An alias is a name for an object on the device that is chosen by the client and treated transparently by the device.
Certificate	A certificate as used in this specification binds a public key to a subject entity. The certificate is digitally signed by the certificate issuer (the certification authority) to allow for verifying its authenticity.
Certification Authority	A certification authority is an entity that issues certificates to subject entities.
Certification Path	A certification path is a sequence of certificates in which the signature of each certificate except for the last certificate can be verified with the subject public key in the next certificate in the sequence.
Certificate Revocation List	A certificate revocation list is a digitally signed list of IDs of certificates that have been revoked by the issuing CA.
Digital Signature	A digital signature for an object allows to verify the object's authenticity, i.e., to check whether the object has in fact been created by the signer and has not been modified afterwards. A digital signature is based on a key pair, where the private key is used to create the signature and the public key is used for verification of the signature.
ECC key pair	A key pair that is accepted as input by the ECC algorithm.
Inner	The authentication protocol used after having established a secure connection using the outer authentication protocol. Also called Stage 2.
Key	A key is an input to a cryptographic algorithm. Sufficient randomness of the key is usually a necessary condition for the security of the algorithm. This specification supports RSA key pairs as keys.
Key Pair	A key that consists of a public key and (optionally) a private key.
Outer	The initial authentication protocol used to establish a secure connection between the device and the IEEE 802.1X authentication server. Also called Stage 1.
Passphrase	A secret string that is shared between two or more parties. A passphrase may be used to derive keys.
RSA key pair	A key pair that is accepted as input by the RSA algorithm.
TLS Server	TLS-enabled HTTP Server (HTTPS)

3.2 Abbreviations

CA	Certification Authority
CN	Common Name
CRL	Certificate Revocation List
CSR	Certificate Signing Request (also called Certification Request)

EAP	Extensible Authentication Protocol
ECC	Elliptic Curve Cryptography
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
JWS	JSON Web Signature
JWT	JSON Web Token
MAC	Message Authentication Code
MD5	Message Digest Algorithm 5
MSCHAPv2	Microsoft's Challenge Handshake Authentication Protocol version 2
PEAP	Protected EAP
SCTP	Stream Control Transmission Protocol
SHA	Secure Hashing Algorithm
SSL	Secure Socket Layer
TLS	Transport Layer Security
TTLS	Tunneled TLS
WS	Web Services

3.3 Namespace

Table 1 lists the namespaces references by this document.

Table 1: Referenced namespaces (with prefix)

Prefix	Namespace URI
env	http://www.w3.org/2003/05/soap-envelope
tas	http://www.onvif.org/ver10/advancedsecurity/wsdl
ter	http://www.onvif.org/ver10/error
tt	http://www.onvif.org/ver10/schema
xs	http://www.w3.org/2001/XMLSchema

4 Overview

4.1 General Structure

This specification covers the following security configuration features:

- Keys and certificates management interface (keystore)
- TLS server configuration interface
- IEEE 802.1X

Basic security features such as user authentication based on WS UsernameToken and HTTP Authentication as well as a default access policy are specified in the ONVIF Core Specification as part of the device management service.

WSDL for the Security Configuration Service is specified in <http://www.onvif.org/ver10/advancedsecurity/wsdl/security.wsdl>.

All sections in this specification are normative unless explicitly marked as informative.

4.2 Certificate-based Client Authentication

Devices can request the connecting client to include a so called client certificate by providing a list of accepted certificate issuers. During the initial TLS handshake the client proves with a signature that it posesses the private key of the certificate. The TLS server needs to verify the authenticity of the certificate by using certification path validation according to RFC 5280 as detailed out in the next sections.

4.3 Certificate Path Verification

4.3.1 Certification Path Validation Algorithm Parameters

By default, a device shall use the values defined in Table 2 for the algorithm parameters defined in RFC 5280, Sect. 6.1.1.

Table 2: Default parameter values for the certification path validation algorithm

Parameter	Default	Default Value Semantics
User-initial-policy-set	Any-policy	The device is not concerned about certificate policy.
Initial-policy-mapping-inhibit	0	Policy mapping is not inhibited.
Initial-explicit-policy	0	The prospective certification path does not have to be valid for at least one certificate policy in the user-initial-policy-set
Initial-any-policy-inhibit	0	The any-policy identifier, if asserted in a certificate, does not have to be ignored
Initial-permitted-subtrees	(not specified)	No restrictions on the subtree within which all subject names in every certificate in the prospective certification path must fall.
Initial-excluded-subtrees	(not specified)	No restrictions on the subtree within which no subject names in any certificate in the prospective certification path may fall.
CA-information-source-for-v1-and-v2	None	The device shall consider X.509 version 1 and version 2 certificates as non-CA certificates.

4.3.2 Revocation Status Checking

By default, a device shall use the parameter values defined in Table 3 for the parameters specified in RFC 5280, section 6.3.1. For CRL processing see section 6.3.3 of RFC 5280.

Table 3: Default parameter values for the revocation status checking algorithm

Parameter	Default	Default Value Semantics
Use-deltas	False	Delta CRLs, if available, are applied to CRLs.
Relevant-reason-codes	All-reasons	The device considers a certificate revoked if it has been revoked for any reason defined in RFC 5280.

By default, a device shall consider all trusted sources of revocation information that it has access to when determining the revocation status of a certificate. The device shall consider the certificate in question revoked if and only if at least one such source indicates that the certificate in question is revoked. If one such source is unavailable, the device shall behave as if this source had provided the reply UNDETERMINED.

A device shall consider at least the CRLs that are present in the keystore of the device.

By default, certificates that are considered revoked shall not be included in prospective certification paths.

4.3.3 Trust Anchors

The trust anchors assigned to the certification path validation policy shall be used as trust anchor input to the certification path validation algorithm specified in RFC 5280.

4.3.4 Certificate Repository for constructing Certification Paths

By default, the certification path validation shall consider all certificates in the keystore on the device when constructing prospective certification paths.

4.3.5 Specific certification path validation parameters

Table 4 defines additional certification path validation parameters.

Table 4: Specific certification path validation parameters

Parameter	Default	Default Value Semantics
RequireTLSWWWClientAuthExtendedKeyUsage	False	If true, a TLS server shall only allow TLS clients to connect that present a client certificate containing the Require WWW client auth extended key usage extension as specified in RFC 5280, Sect. 4.2.1.12.

4.4 JWT-based client authorization

4.4.1 General

Unlike with HTTP Digest and WS-UsernameToken authentication, this specification defines how to associate clients with the different User Levels as defined in the ONVIF Core Specification. Devices are not expected to have the user credentials stored in their memory, instead they will verify that the user has been correctly authenticated by a trusted service based on OpenID Connect and authorize accessing different functions, based on the claims presented in the supplied JWT. JWTs can be presented as headers in case of HTTPS or RTSP, or they can be embedded in SOAP messages, in the form of binary security tokens. In case JWTs are embedded in binary security tokens, their content is base64url-encoded according to RFC 7519.

JWTs consists of three elements:

- JOSE header
- Payload
- Signature

The JOSE header shall declare that the encoded object is a JSON Web Token by setting the *typ* parameter to *JWT*, and the JWT is a JWS that is signed using the algorithm identified by the *alg* claim with a value defined in RFC7518. Since the keystore does not support symmetric encryption, this specification only mandates asymmetric encryption.

The JWT payload shall include the following standard claims:

- *iss*: Issuer, the URI of the server that generated the JWT.

- *aud*: Audience, a list of strings representing the recipients that the JWT is intended for. Each principal intended to process the JWT shall identify itself with a value in the audience claim. If the principal processing the claim does not identify itself with a value in the audience claim when this claim is present, then the JWT shall be rejected.
- *exp*: Expiration Time.
- *nbf*: Not before.

For the *exp*, *nbf* claims, a device shall reject a token when the current time is not within the range of claims nbf and exp. It shall reject a token when at least one of the claims is missing.

The JWT payload shall include the *roles* claim, as defined within RFC 7643:

- *roles*: Access class. One of the authenticated classes of the default access policy prefixed with the string *onvif:*, i.e. "onvif:Administrator", "onvif:Operator" or "onvif:User" as defined also within the ONVIF Core Specification. This access level will be used to authorize access to the required function.

The signature is evaluated by applying ECDSA using secp256r1 and SHA-256 hashing to the base64url-encoded header and payload, concatenated by a '.'

```
ECDSA-SHA256 (base64UrlEncode(header) + "." +
              base64UrlEncode(payload),
              secp256r1))
```

The evaluated signature is further appended to the encoded header and payload with a '.' separator, to get the final JWT. Appendix Annex B demonstrates the construction or a JWT.

The signature is used to protect the JWT data integrity and JWT source authenticity.

4.4.2 Configuration

The folowing parameters are defined:

Audiences	One or more audience strings. A device shall reject a token if none of the provided strings matches.
TrustedIssuer	Optional list of trusted issuer metadata URIs. A device shall reject a token if the list is non-empty and the token does not match the information provided via one of the issuer metadata URIs.
KeyID	Optional list of keys. A device shall reject a token if this list is non-empty and none of the provided keys can verify the signature.
ValidationPolicy	Optional cert path validation policies to verify trusted issuers. If this list is non-empty a device shall only accept trusted issuers with matching TLS server certificates.
CustomClaims	Optional list of custom claims. A device shall reject a token if it does not contain a claim or if none of the provided claim values match.

4.4.3 Usage of JWT-based client authentication over HTTP

Since authentication tokens contain sensitive information that could be easily used to perform replay attacks, JWT-based client authentication shall not be used over unencrypted HTTP connections.

4.4.4 Usage of JWT-based client authentication over HTTPS

Usage JWT-based client authentication over HTTPS shall adhere to the Authorization Request header Field, as specified by RFC 6750. In case of authentication failure, ONVIF-compliant devices shall respond with the error codes specified by RFC 6750.

4.4.5 Usage of JWT-based client authentication over SCTP

In scenarios where a SCTP channel is used to exchange commands and responses between the device and the client, it is not possible to embed a JWT within a header. In this case, the JWT can be embedded in the Security Token of the SOAP message, as defined in the WS-Security Basic Security Profile.

A client shall use both ValueType and EncodingType. The device shall reject any Binary Security Token not using both *ValueType* and *EncodingType*.

The EncodingType attribute shall have the value of

```
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#BinarySecurityToke
```

The ValueType attribute shall have the value of

```
urn:ietf:params:oauth:token-type:jwt
```

4.4.6 Usage of JWT-based client authentication over RTSP

Since authentication tokens contain sensitive information that could be easily used to perform replay attacks, JWT-based client authentication shall be used with RTSP connections only when HTTPS transport is selected.

4.4.7 Usage of JWT-based client authentication over RTSPS

Since the RTSPS protocol is similar to HTTPS, JWT-based client authentication over RTSPS behaves in the same way as JWT-based client authentication over HTTPS.

4.5 IEEE 802.1X

IEEE 802.1X is an IEEE standard for port based network access control for the purpose of providing authentication and authorization of the devices attached to LAN ports. It allows access to the LAN port to devices that are configured for access, and prevents access to the LAN port to devices that are not correctly configured.

This specification recommends the adoption of IEEE 802.1X for port based authentication for wireless networks.

This specification defines a set of commands to configure and manage a device’s IEEE 802.1X configurations, both for wireless and hardwired network interfaces. It assumes that IEEE 802.1X configuration and reconfiguration is performed outside of the IEEE 802.1X-secured network.

Many schema elements in this specification include Dot1X as shorthand for IEEE 802.1X. This convention increases the readability of source code generated from the WSDL.

4.6 Authorization Servers

This section describes use of external authorization servers for authenticating users and controlling access to resources. It can be used for granting access to both the device itself and to external services, like cloud storage.

Different approaches are explained for granting access to human users and for machine to machine authorization. Both approaches base on OAuth 2.0 as defined in RFC6749. OpenID Connect has been developed as an extension of OAuth 2.0 especially for granting access to human users.

OAuth 2.0 defines a web API called "authorization endpoint" to retrieve tokens. Clients control the purpose via *scope* parameters.

4.6.1 Device authentication and authorization

Devices typically use the Oauth2 Client Credentials Grant Flow as specified by RFC6749 to gain access to resources. The device (called a client in the OAuth2 Spec) authenticates itself at the authorization server according to one of the methods listed in Table 7. In case of success the authorization server responds to the request with an access token that authorizes access to the actual resource.

The Client Credentials Flow is summarized in Figure 1.

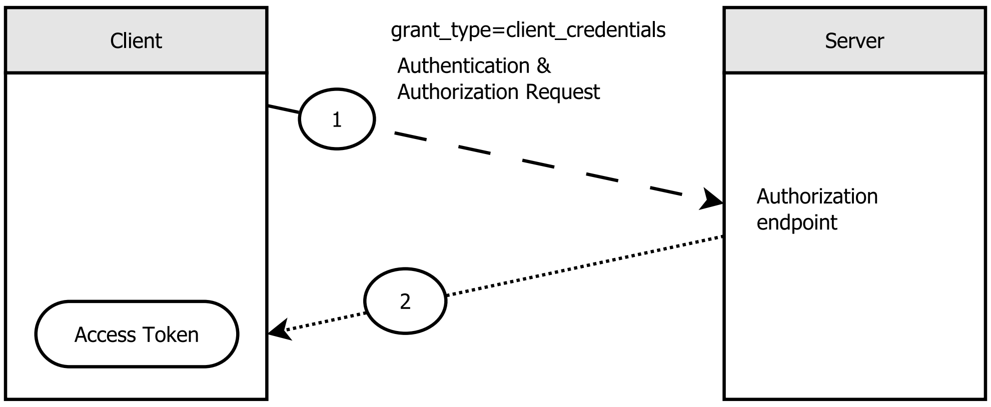
Client
Server
grant_type=client_credentials
Authentication & Authorization Request
1
Authorization endpoint
2
Access Token

Figure 1: OpenID Connect client credentials flow for devices

OAuth 2.0 client registration requires configuration of ClientId, ClientSecret, an authentication method according to Table 7plus server dependent supplemental parameters before the client can use the OAuth2.0 and OpenID Connect flow.

4.6.2 User authentication and authorization

This flow separates retrieval of access token in two steps to allow separation of authentication and authorization. Users can retrieve JWTs from an OpenID Connect server by authenticating and authorizing access to resources by following the OAuth2 Authorization Code Flow.

The Authorization Code Flow is summarized in Figure 2.

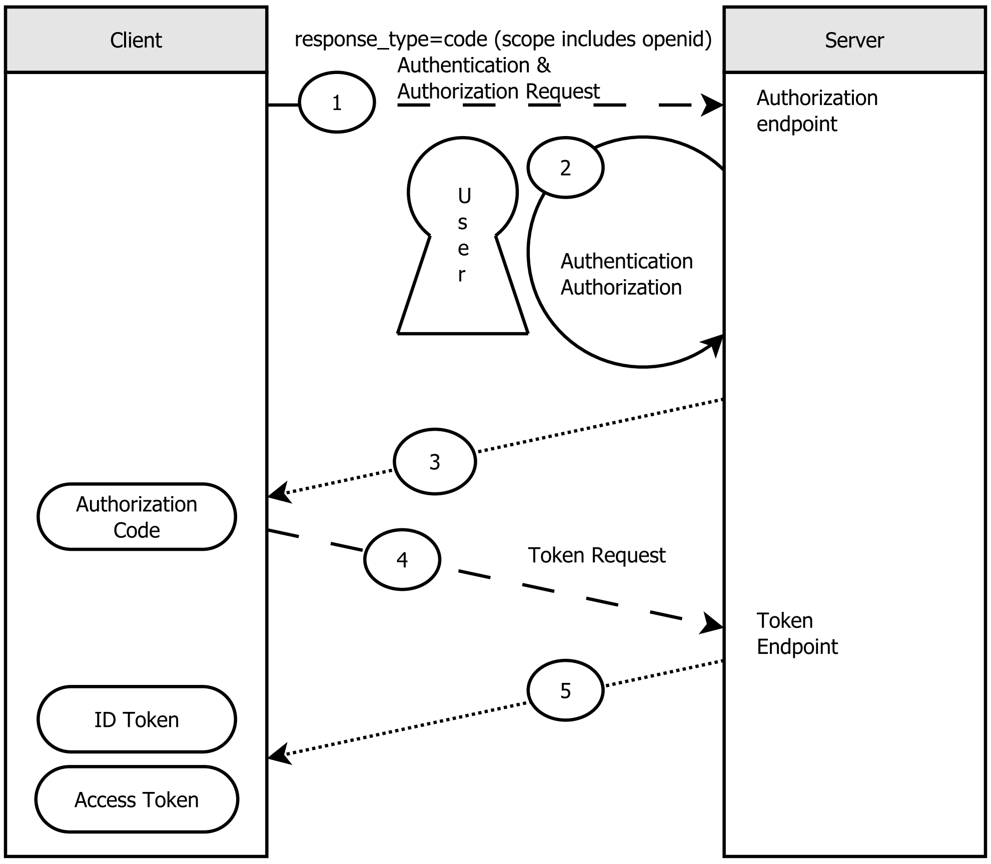
Client
Server
response_type=code (scope includes openid)
Authentication & Authorization Request
1
Authorization endpoint
User
2
Authentication Authorization
3
Authorization Code
4
Token Request
Token Endpoint
5
ID Token
Access Token

Figure 2: OpenID Connect authorization code flow for users

A device can be setup to use JWTs obtained with this flow to allow access to its web interface for externally authenticated users. In that case an authorization server configuration with type OAuthAuthorizationCode or OIDC2AuthorizationCode should be created.

Access to the device can be granted to users bearing either an ID Token or an Access Token, as long as the claims specified in 4.4 are present.

Any request for ID tokens must include *openid* in the *scope* request parameter.

4.6.3 Authorization server configuration

The following parameters are used when creating a configuration for an external authorization server.

Table 5: Authorization server settings

Setting	Description
ServerUri	Authorization server address
ClientId	Client identifier issued by the authorization server
ClientSecret	Client secret used to authenticate with the authorization server
Scope	The requested access scope(s)
KeyID	Key identifier for the *private_key_jwt* authentication method
CertificateID	Certificate identifier for the *self_signed_tls_client_auth* and *tls_client_auth* authentication methods.
Type	The type of configuration, see Table 6 for possible values.
ClientAuth	The type of client authentication method to use, see Table 7 for possible values.
CertPathValidationPolicyID	The unique identifier of the certification path validation policy to be used for validating the server certificate. If not configured, server certificate validation behavior is undefined and the device may either apply a vendor specific default validation policy or skip validation at all.

The following types of authorization server configurations are defined.

Table 6: Authorization server configuration types

Method	Description
OAuthAuthorizationCode	OAuth2 authorization code flow per RFC 6749.
OAuthClientCredentials	OAuth2 client credentials grant flow per RFC 6749.
OIDC2AuthorizationCode	OpenID Connect authorization code flow per Open ID Connect Core. The ServerUri is used as metadata URI, where you can retrieve the endpoint URIs for authorization, token and JWKS.

The following client authentication methods are defined.

Table 7: Client authentication methods

Method	Description
client_secret_basic	Use HTTP Authorization header to specify client_secret, see RFC 6749
client_secret_post	Use HTTP POST body to specify client_secret, see RFC 6749
client_secret_jwt	Use a HMAC signed JWT using client_secret as shared secret during client registration, see OpenID Connect Core

Method	Description
private_key_jwt	Use PKI signed JWT using private key, see OpenID Connect Core, public key shared with authorization server during client registration
tls_client_auth	Use PKI certificate to authenticate, public key shared with authorization server during client registration and X.509 certificate used to setup mTLS with authorization server, see RFC 8705
self_signed_tls_client_auth	Use self-signed certificate to authenticate, public key shared with authorization server during client registration and self signed certificate used to setup mTLS with authorization server,see RFC 8705

A device signaling support for tls_client_auth shall signal support for PKCS10.

A device signaling support for self_signed_tls_client_auth shall signal support for SelfSignedCertificateCreation.

A device signaling support for private_key_jwt shall signal support for ECCKeyPairGeneration.

5 Security Configuration Service

5.1 General Structure

This section covers the security features

- Keystore
- TLS server
- IEEE 802.1X

The design and data model of the ONVIF Security Configuration Service is reflected in Figure 3.

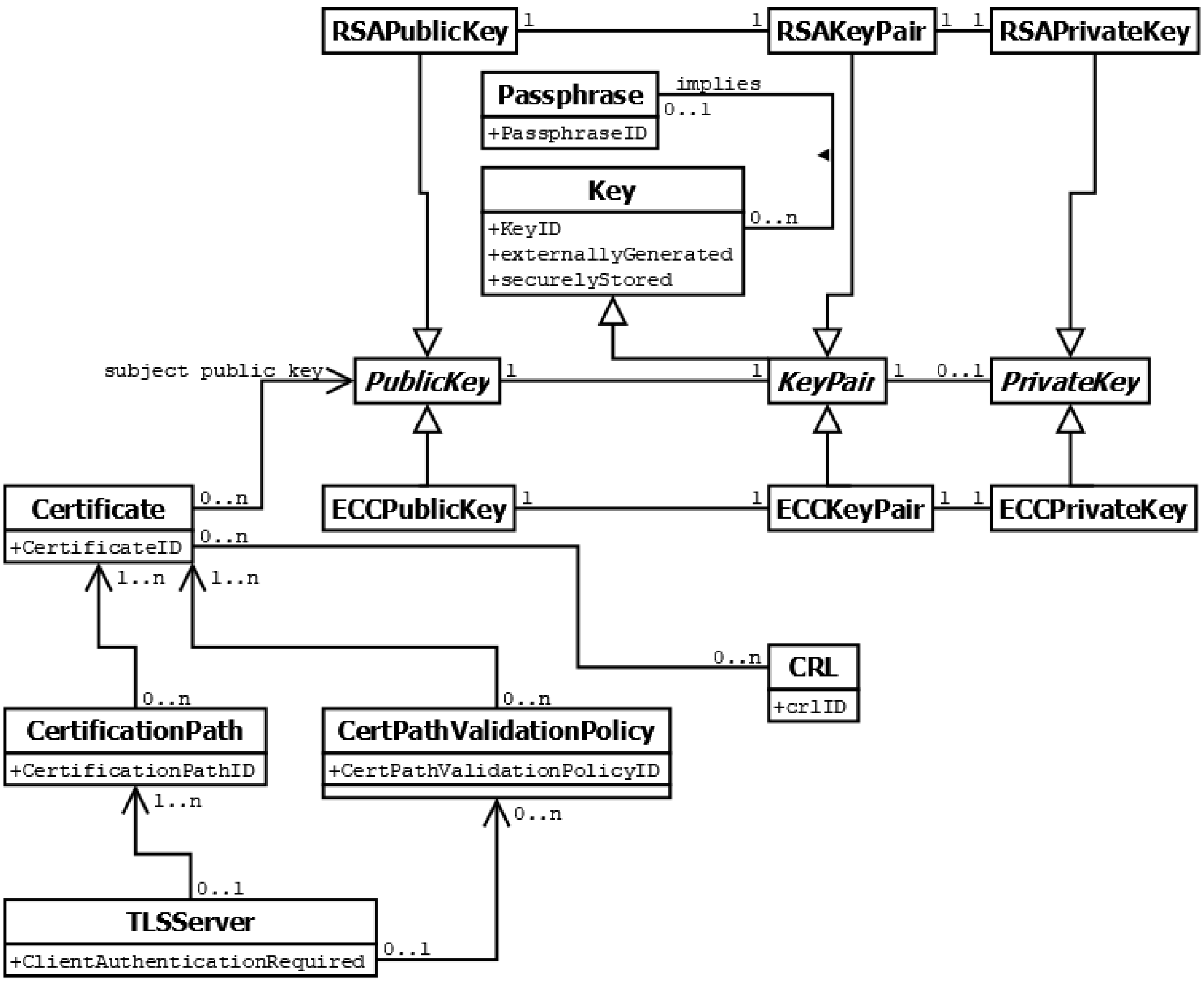

Figure 3: ONVIF Security Configuration Service UML Class Diagram

5.2 Keystore

5.2.1 Elements of the Keystore

The keystore security feature handles the storage and management of passphrases, keys, and certificates on an ONVIF device.

The keystore specified in this document supports passphrases, keys, key pairs, RSA and ECC key pairs, which are particular types of key pairs, certificates, certification paths, certificate revocation lists, and certification path validation policies.

The boolean attribute *externallyGenerated* of a key shall be true if and only if the key was generated outside the device.

The boolean attribute *securelyStored* of a key shall be true if and only if the key is stored in a specially protected hardware component (e.g., a trusted platform module) inside the device.

5.2.2 Unique Identifiers

An ID is used to uniquely identify objects of a particular type in the keystore on a device, i.e., no two objects of the same type shall have the same ID at any time.

Passphrases in the keystore shall be uniquely identified by passphrase IDs, keys shall be uniquely identified by key IDs, certificates shall be uniquely identified by certificate IDs, certification paths in the keystore shall be uniquely identified by certification path IDs, certificate revocation lists shall be uniquely identified by certificate revocation list IDs, certification path validation policies shall be uniquely identified by certification path validation policy IDs, and IEEE 802.1X configurations shall be uniquely identified by IEEE 802.1X configuration IDs.

It shall be noted that while IDs within a specific type shall be unique, no requirement exists for the uniqueness of IDs across different types. For example, there may be a key and a certificate in the keystore that share the same ID.

Devices may assign the ID of a deleted identified object to another, subsequently generated object. However, devices should avoid re-using IDs as long as possible to avoid race conditions on the client side.

A client may supply an alias for passphrases, keys, certificates, certification paths, certificate revocation lists, certification path validation policies and IEEE 802.1X configurations upon creation, e.g., to facilitate recognizing the created object at a later time. The device shall treat such aliases as unstructured data.

5.2.3 Uniqueness of Objects in the Keystore

A device shall allow multiple copies of the same passphrase to be present in the keystore under different IDs simultaneously.

A device shall allow multiple copies of the same certificate revocation list to be present in the keystore under different IDs, respectively.

A device shall allow multiple copies of the same certification path validation policy to be present in the keystore under different IDs, respectively.

A device shall allow multiple copies of the same IEEE 802.1X configuration to be present in the keystore under different IDs simultaneously.

A device shall not allow multiple copies of the same key to be present in the keystore simultaneously.

5.2.4 Referential Integrity

The keystore design relies on associations between

- Keys, especially key pairs, and certificates
- Public keys and private keys in key pairs
- Certificates and certification paths
- Keys and security features
- Certification paths and IEEE 802.1X configurations
- Passphrases and IEEE 802.1X configurations
- IEEE 802.1X configurations and security features
- Certificates and security features
- Certification path validation policies and certificates
- Certificate revocation lists and certificates
- Certification path validation policies and security features

A device shall enforce the following referential integrity rules for delete operations:

- A key shall not be deleted if it is referenced by a certificate or a security feature.
- A certificate shall not be deleted if it is referenced by a certification path, a certificate revocation list, a certification path validation policy, or a security feature.

- A certification path shall not be deleted if it is referenced by an IEEE 802.1X configuration or a security feature.
- A passphrase shall not be deleted if it is referenced by an IEEE 802.1X configuration.
- A certification path validation policy shall not be deleted if it is referenced by a security feature.
- An IEEE 802.1X configuration shall not be deleted if it is referenced by a security feature.

This integrity rule may be enforced by the following mechanism. Reference counters are maintained for keys, certificates, certification paths, passphrases, and IEEE 802.1X configurations. Each time a reference to an object of these types is added, e.g., by associating a certificate to a key pair or assigning a key pair or certificate to a security feature, the reference counter of the object is incremented. Conversely, if a reference to an object is deleted, the reference counter of the referenced object is decremented. Deleting a key, certificate, or certification path is only permitted if the corresponding reference counter is equal to zero.

A device shall enforce the following referential integrity rules for update operations:

- A key shall not be updated if it is referenced by a certificate or a security feature. However, a private key may be added to an existing key pair if the private key matches the public key in the key pair. If a private key is about to be added to a key pair that already contains the private key to be added, the adding operation shall have no effect.
- A certificate shall not be updated if it is referenced by a certification path, a certificate revocation list, a certification path validation policy, or a security feature.
- A certification path validation policy shall not be updated if it is referenced by a security feature.

This specification omits APIs for modifying keys or certificates. If a key or certificate is to be updated, it has to be deleted and newly generated with the updated information. If other API exists that allows for modification of keys or certificates, special care shall be taken in order not to break the referential integrity rule.

A device shall enforce the following invariants:

- The private key and the public key in an asymmetric key pair in the keystore shall always match, i.e., the asymmetric operation under the public key is the inverse of the corresponding operation under the private key.
- The public key in a certificate in the keystore and the public key in an associated key pair in the keystore shall always be equal for all associated key pairs.

5.2.5 Key Status

A key in the keystore is always in exactly one of the following states:

- *ok* (The key is ready to be used)
- *generating* (The key is being generated and not yet ready for use)
- *corrupt* (The key is corrupt and shall not be used, e.g., because it was not properly generated or a hardware fault corrupted a key that was ready to be used)

5.2.6 Keystore Operations

5.2.6.1 Passphrase Management

5.2.6.1.1 UploadPassphrase

This operation uploads a passphrase to the keystore of the device.

Passphrases are uniquely identified using passphrase IDs. The device shall generate a new passphrase ID for the uploaded passphrase.

If the command was successful, the device shall return the ID of the uploaded passphrase.

If the device does not have enough storage capacity for storing the passphrase to be uploaded, the device shall produce a maximum number of passphrases reached fault and shall not upload the supplied passphrase.

If the device cannot process the passphrase to be uploaded, the device shall produce a BadPassphrase fault and shall not upload a passphrase.

REQUEST:

- **Passphrase - [xs:string]**
  The passphrase to upload.
- **PassphraseAlias - optional [xs:string]**
  The alias for the passphrase to upload.

RESPONSE:

- **PassphraseID - [tas:PassphraseID]**
  The PassphraseID of the uploaded passphrase.

FAULTS:

- **env:Receiver - ter:Action - ter:MaximumNumberOfPassphrasesReached**
  The device does not have enough storage space to store the passphrase to be uploaded.
- **env:Sender - ter:InvalidArgVal - ter:BadPassphrase**
  The provided passphrase cannot be processed by the device.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.1.2 GetAllPassphrases

This operation returns information about all passphrases that are stored in the keystore of the device.

This operation may be used, e.g., if a client lost track of which passphrases are present on the device.

If no passphrase is stored on the device, the device shall return an empty list.

REQUEST:

This message is empty.

RESPONSE:

- **PassphraseAttribute - optional, unbounded [tas:PassphraseAttribute]**
  List of passphrase attributes. Each attribute contains information about a passphrase in the keystore.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.1.3 DeletePassphrase

This operation deletes a passphrase from the keystore of the device.

Passphrases are uniquely identified using passphrase IDs. If no passphrase is stored under the requested passphrase ID in the keystore, a device shall produce an invalid passphrase ID fault. If there is a passphrase under the requested passphrase ID stored in the keystore and the passphrase could not be deleted, a device shall produce a passphrase deletion failed fault.

After a passphrase is successfully deleted, the device may assign its former ID to other passphrases.

REQUEST:

- **PassphraseID - [tas:PassphraseID]**
  The ID of the passphrase that is to be deleted from the keystore.

RESPONSE:

This message is empty.

FAULTS:

- **env:Receiver - ter:Action - ter:PassphraseDeletionFailed**
  Deleting the passphrase with the requested PassphraseID failed.
- **env:Sender - ter:InvalidArgVal - ter:PassphraseID**
  No passphrase is stored under the requested PassphraseID.

ACCESS CLASS:

UNRECOVERABLE

5.2.6.2 Key Management

5.2.6.2.1 CreateRSAKeyPair

This operation triggers the asynchronous generation of an RSA key pair of a particular keylength (specified as the number of bits) as specified in RFC 3447, with a suitable key generation mechanism on the device. Keys, especially RSA key pairs, are uniquely identified using key IDs.

If the device does not have enough storage capacity for storing the key pair to be created, the maximum number of keys reached fault shall be produced and no key pair shall be generated. Otherwise, the operation generates a keyID for the new key and associates the *generating* status to it. Immediately after key generation has started, the device shall return the keyID to the client and continue to generate the key pair. The client may query the device with the GetKeyStatus operation (see Sect. 5.2.6.2.4) whether the generation has finished. The client may also subscribe to Key Status events (see Sect. 5.9.1) to be notified about key status changes.

The device also returns a best-effort estimate of how much time it requires to create the key pair.[1] A client may use this information as an indication how long to wait before querying the device whether key generation is completed.

After the key has been successfully created, the device shall assign it the *ok* status. If the key generation fails, the device shall assign the key the *corrupt* status.

REQUEST:

- **KeyLength - [xs:nonNegativeInteger]**
  The length of the key to be created.
- **Alias - optional [xs:string]**
  The client-defined alias of the key.

RESPONSE:

- **KeyID - [tas:KeyID]**
  The key ID of the key pair being generated.

[1]Implementors may estimate the key generation time for a fixed key length as the average elapsed time of a number of key generation operations for this key length.

- **EstimatedCreationTime - [xs:duration]**
  Best-effort estimation of how long the key generation will take.

FAULTS:

- **env:Receiver - ter:Action - ter:MaximumNumberOfKeysReached**
  The keystore does not have enough storage space to store the key pair that has to be generated.
- **env:Sender - ter:InvalidArgVal - ter:KeyLength**
  The specified key length is not supported by the device.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.2.2 CreateECCKeyPair

This operation triggers the asynchronous generation of an ECC key pair using a particular elliptic curve as specified in RFC 8422, with a suitable key generation mechanism on the device. Keys, especially ECC key pairs, are uniquely identified using key IDs.

If the device does not have enough storage capacity for storing the key pair to be created, the maximum number of keys reached fault shall be produced and no key pair shall be generated. Otherwise, the operation generates a keyID for the new key and associates the *generating* status to it. Immediately after key generation has started, the device shall return the keyID to the client and continue to generate the key pair. The client may query the device with the GetKeyStatus operation (see Sect. 5.2.6.2.4) whether the generation has finished. The client may also subscribe to Key Status events (see Sect. 5.9.1) to be notified about key status changes.

The device also returns a best-effort estimate of how much time it requires to create the key pair.[2] A client may use this information as an indication how long to wait before querying the device whether key generation is completed.

After the key has been successfully created, the device shall assign it the *ok* status. If the key generation fails, the device shall assign the key the *corrupt* status.

REQUEST:

- **EllipticCurve - [xs:string]**
  The name of the elliptic curve to be used for generating the ECC keypair. Supported curve names can be found in the IANA TLS Supported Groups section, under the *Description* field.
- **Alias - optional [xs:string]**
  The client-defined alias of the key.

RESPONSE:

- **KeyID - [tas:KeyID]**
  The key ID of the key pair being generated.
- **EstimatedCreationTime - [xs:duration]**
  Best-effort estimation of how long the key generation will take.

FAULTS:

- **env:Receiver - ter:Action - ter:MaximumNumberOfKeysReached**
  The keystore does not have enough storage space to store the key pair that has to be generated.
- **env:Sender - ter:InvalidArgVal - ter:UnsupportedEllipticCurve**
  The specified elliptic curve is not supported by the device.

[2]Implementors may estimate the key generation time for a fixed key length as the average elapsed time of a number of key generation operations for this key length.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.2.3 UploadKeyPairInPKCS8

This operation uploads a key pair in a PKCS#8 data structure as specified in [RFC 5958, RFC 5959].

If a passphrase is either directly provided or as ID reference to a previously uploaded passphrase, the device shall assume that the KeyPair parameter contains an EncryptedPrivateKeyInfo ASN.1 structure that is encrypted with the given passphrase. In case neither a passphrase nor a passphrase ID is provided the device shall assume that the KeyPair parameter contains a OneAsymmetricKey ASN.1 structure which contains both the private key and the corresponding public key.

If the supplied key pair cannot be processed by the device, the device shall produce an UnsupportedPublicKeyAlgorithm fault and shall not store the uploaded key pair in the keystore.

Key pairs are uniquely identified using key IDs. If a key pair exists in the keystore with the public key equal to the public key in the request and this key pair does not contain a private key, the device shall add the supplied private key to the existing key pair and return the ID of this key pair.

If a key pair exists in the keystore with the public key equal to the public key in the request and this key pair contains a private key, the device shall leave the key pair unchanged and return the ID of this key pair.

If the existing key pair does not have status *ok*, the device shall produce an InvalidKeyStatus fault and shall not modify the existing key pair.

If no key pair exists in the keystore with the public key equal to the public key in the request, the device shall generate a new key pair with the supplied private key and the supplied public key, status *ok* and the externally generated attribute set to *true*. Furthermore, the device shall return the ID of this key pair.

If a new key pair is created, the device shall assign the supplied alias to it. Otherwise, the device shall ignore an eventually supplied alias.

If decryption of the EncryptedPrivateKeyInfo failed, the device shall produce a DecryptionFailed fault and shall not store the uploaded key pair in the keystore.

If the device does not have enough storage capacity for storing the key pair that eventually has to be created, the device shall generate a maximum number of keys reached fault. Furthermore the device shall not generate a key pair.

If no passphrase exists under the ID specified by EncryptionPassphraseID, the device shall produce an invalid passphrase ID fault and shall not store the uploaded key pair in the keystore.

If the supplied PKCS#8 data structure cannot be processed by the device, the device shall produce a BadPKCS8File fault and shall not store the uploaded key pair in the keystore.

If the public key in the uploaded key pair does not match the uploaded private key, the device shall produce a PublicPrivateKeyMismatch fault and shall not store the uploaded key pair in the keystore.

If the command was successful, the device shall return the ID of the key pair in the keystore that contains the supplied public and private key.

REQUEST:

- **KeyPair - [tas:Base64DERencodedASN1Value]**
  The key pair to be uploaded in a PKCS#8 data structure.
- **Alias - optional [xs:string]**
  The client-defined alias of the key pair.
- **EncryptionPassphraseID - optional [tas:PassphraseID]**
  The ID of the passphrase to use for decrypting the uploaded key pair.

- **EncryptionPassphrase - optional [xs:string]**
  The passphrase to use for decrypting the uploaded key pair.

RESPONSE:

- **KeyID - [tas:KeyID]**
  The key ID of the uploaded key pair.

FAULTS:

- **env:Receiver - ter:Action - ter:MaximumNumberOfKeysReached**
  The device does not have enough storage space to store the key pair to be uploaded.
- **env:Sender - ter:InvalidArgVal - ter:PassphraseID**
  No passphrase is stored under the requested PassphraseID.
- **env:Sender - ter:InvalidArgVal - ter:DecryptionFailed**
  The given date could not be decrypted.
- **env:Sender - ter:InvalidArgVal - ter:UnsupportedPublicKeyAlgorithm**
  The public key algorithm of the supplied key pair is not supported by the device.
- **env:Sender - ter:InvalidArgVal - ter:InvalidKeyStatus**
  The key with the requested KeyID has an inappropriate status.
- **env:Sender - ter:InvalidArgVal - ter:BadPKCS8File**
  The PKCS#8 data structure cannot be processed by the device.
- **env:Sender - ter:InvalidArgVal - ter:PublicPrivateKeyMismatch**
  The supplied private key does not match the supplied public key.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.2.4 GetKeyStatus

This operation returns the status of a key as defined in Sect. 5.2.5.

Keys are uniquely identified using key IDs. If no key is stored under the requested key ID in the keystore, an InvalidKeyID fault is produced. Otherwise, the status of the key is returned.

REQUEST:

- **KeyID - [tas:KeyID]**
  The ID of the key for which to return the status.

RESPONSE:

- **KeyStatus - [xs:string]**
  Status of the requested key. The value should be one of the values in the tas:KeyStatus enumeration.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:KeyID**
  No key is stored under the requested KeyID.

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.2.5 GetPrivateKeyStatus (deprecated)

This operation returns whether a key pair contains a private key.

Keys are uniquely identified using key IDs. If no key is stored under the requested key ID in the keystore, an invalid key ID fault shall be produced. If a key is stored under the requested key ID in the keystore, but this key is not a key pair, an invalid key type fault shall be produced.

Otherwise, this operation returns *true* if the key pair identified by the key ID contains a private key, and *false* otherwise.

This command is deprecated. Use GetAllKeys (see Sect. 5.2.6.2.6) instead.

REQUEST:

- **KeyID - [tas:KeyID]**
  The ID of the key pair for which to return whether it contains a private key.

RESPONSE:

- **hasPrivateKey - [xs:boolean]**
  True if and only if the key pair contains a private key.

FAULTS:

- **ter:Sender - ter:InvalidArgVal - ter:KeyID**
  No key is stored under the requested KeyID.
- **ter:Sender - ter:InvalidArgVal - ter:InvalidKeyType**
  The key stored in the keystore under the requested KeyID is of an invalid type.

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.2.6 GetAllKeys

This operation returns information about all keys that are stored in the device's keystore.

This operation may be used, e.g., if a client lost track of which keys are present on the device.

If no key is stored on the device, an empty list is returned.

REQUEST:

This message is empty.

RESPONSE:

- **KeyAttribute - optional, unbounded [tas:KeyAttribute]**
  List of key attributes. Each attribute contains information about a key in the keystore.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.2.7 DeleteKey

This operation deletes a key from the device's keystore.

Keys are uniquely identified using key IDs. If no key is stored under the requested key ID in the keystore, a device shall produce an InvalidArgVal fault. If a reference exists for the specified key, a device shall produce the corresponding fault and shall not delete the key. If there is a key under the requested key ID stored in the keystore and the key could not be deleted, a device shall produce a KeyDeletion fault. If the key has the status *generating*, a device shall abort the generation of the key and delete from the keystore all data generated for this key.

After a key is successfully deleted, the device may assign its former ID to other keys.

REQUEST:

- **KeyID - [tas:KeyID]**
  The ID of the key that is to be deleted from the keystore.

RESPONSE:

This message is empty.

FAULTS:

- **ter:Receiver - ter:Action - ter:KeyDeletionFailed**
  Deleting the key with the requested KeyID failed.
- **ter:Sender - ter:InvalidArgVal - ter:KeyID**
  No key is stored under the requested KeyID.
- **ter:Sender - ter:InvalidArgVal - ter:ReferenceExists**
  A reference exists for the object that is to be deleted.

ACCESS CLASS:

UNRECOVERABLE

5.2.6.3 Certificate Management

5.2.6.3.1 CreatePKCS10CSR

This operation generates a DER-encoded PKCS#10 v1.7 certification request (sometimes also called certificate signing request or CSR) as specified in RFC 2986 for a public key on the device.

The key pair that contains the public key for which a certification request shall be produced is specified by its key ID. If no key is stored under the requested KeyID or the key specified by the requested KeyID is not an asymmetric key pair, an invalid key ID fault shall be produced and no CSR shall be generated.

The subject parameter describes the entity that the public key belongs to. Additional attributes can be included in the attribute parameter.

Distinguished name attribute values shall be supplied either in UTF-8 or in hexadecimal form as specified in RFC 4514.

If the distinguished name attribute value is supplied in hexadecimal form, the device shall encode the attribute in the format given in the hexadecimal format.

If the distinguished name attribute value is supplied in UTF-8 and the attribute value has a uniquely defined encoding (e.g., CountryName is defined as PrintableString), the device shall encode the attribute as the defined encoding. Otherwise, the device shall encode the attribute value as UTF-8.

The signature algorithm parameter determines which signature algorithm shall be used for signing the certification request with the public key specified by the key ID parameter. If the specified signature algorithm is not supported by the device, an UnsupportedSignatureAlgorithm fault shall be produced and no CSR shall be generated. If the public key identified by the requested Key ID is an invalid input to the specified signature algorithm, a KeySignatureAlgorithmMismatch fault shall be produced and no CSR shall be generated. If the specified subject is invalid or incomplete, a Subject invalid fault shall be produced and no CSR shall be created. If an attribute is invalid or incomplete, an Attribute invalid fault shall be produced and no CSR shall be generated.

If the key pair does not have status *ok*, a device shall produce an InvalidKeyStatus fault and no CSR shall be generated.

REQUEST:

- **Subject - [tas:DistinguishedName]**
  The subject to be included in the CSR.
- **KeyID - [tas:KeyID]**
  The ID of the key for which the CSR shall be created.
- **CSRAttribute - optional, unbounded [tas:CSRAttribute]**
  List of CSR attributes. Each attribute contains an attribute to be included in the CSR.
- **SignatureAlgorithm - [tas:AlgorithmIdentifier]**
  The signature algorithm to be used to sign the CSR.

RESPONSE:

- **PKCS10CSR - [tas:Base64DERencodedASN1Value]**
  The DER encoded PKCS#10 certification request.

FAULTS:

- **ter:Receiver - ter:Action - CSRCreationFailed**
  The generation of the PKCS#10 certification request failed.
- **ter:Sender - ter:InvalidArgVal - ter:KeyID**
  No key is stored under the requested KeyID.
- **ter:Sender - ter:InvalidArgVal - ter:UnsupportedSignatureAlgorithm**
  The specified signature algorithm is not supported by the device.
- **ter:Sender - ter:InvalidArgVal - ter:KeySignatureAlgorithmMismatch**
  The specified public key is an invalid input to the specified signature algorithm.
- **ter:Sender - ter:InvalidArgVal - ter:InvalidKeyStatus**
  The key with the requested KeyID has an inappropriate status.
- **ter:Sender - ter:InvalidArgVal - ter:InvalidSubject**
  The specified subject is invalid or incomplete.
- **ter:Sender - ter:InvalidArgVal - ter:InvalidAttribute**
  The specified attribute is invalid or incomplete.

ACCESS CLASS:

READ_SYSTEM

5.2.6.3.2 CreateSelfSignedCertificate

This operation generates for a public key on the device a self-signed X.509 certificate that complies to RFC 5280.

The X509Version parameter specifies the version of X.509 that the generated certificate shall comply to. A device that supports this command shall support the generation of X.509v3 certificates as specified in RFC 5280 and may additionally be able to handle other X.509 certificate formats as indicated by the X.509Versions capability. If no X509Version is specified in the request, the device shall produce an X.509v3 certificate.

The key pair that contains the public key for which a self-signed certificate shall be produced is specified by its key pair ID. The subject parameter describes the entity that the public key belongs to.

If the key pair does not have status *ok*, a device shall produce an InvalidKeyStatus fault and no certificate shall be generated.

If the specified subject is invalid or incomplete, an InvalidSubject fault shall be produced and no certificate shall be created.

The notValidBefore parameter specifies at which point in time the validity period of the generated certificate shall begin. If this parameter is not specified in the request, the device shall use its current time or a time before its current time as starting point of the validity period. The notValidAfter parameter specifies at which point in time the validity period of the generated certificate shall end. If this parameter is not specified in the request, the device shall assign the GeneralizedTime value of 99991231235959Z as specified in RFC 5280 to the notValidAfter parameter. If the notValidBefore parameter is invalid, an invalid DateTime fault shall be produced and no certificate shall be generated. If the notValidAfter parameter is invalid, an invalid DateTime fault shall be produced and no certificate shall be generated.

The signature algorithm parameter determines which signature algorithm shall be used for signing the certification request with the public key specified by the key ID parameter.

The Extensions parameter specifies potential X509v3 extensions that shall be contained in the certificate. A device that supports this command shall support the extensions that are defined in RFC5280, Sect. 4.2 as mandatory for CAs that issue self-signed certificates.

Distinguished name attribute values shall be supplied either in UTF-8 or in hexadecimal form as specified in RFC 4514.

If the distinguished name attribute value is supplied in hexadecimal form, the device shall encode the attribute in the format given in the hexadecimal format.

If the distinguished name attribute value is supplied in UTF-8 and the attribute value has a uniquely defined encoding (e.g., CountryName is defined as PrintableString), the device shall encode the attribute as the defined encoding. Otherwise, the device shall encode the attribute value as UTF-8.

RFC 5280, Sect. 4.1.2.2 mandates that the certificate serial numbers be unique for each certificate issued by a given issuer (a CA). Since the subject is equal to the issuer in a self-signed certificate, the serial number shall be unique for each self-signed certificate that the device issues for a given subject.

The generated certificate shall not contain a unique identifier as specified in RFC 5280, Sect. 4.1.2.8. The device shall not mark the generated certificate as trusted.

Certificates are uniquely identified using certificate IDs. If the command was successful, the device generates a new ID for the generated certificate and returns this ID.

If the device does not have enough storage capacity for storing the certificate to be created, the maximum number of certificates reached fault shall be produced and no certificate shall be generated.

REQUEST:

- **X509Version - optional [xs:positiveInteger]**
  The X.509 version that the generated certificate shall comply to.
- **Subject - [tas:DistinguishedName]**
  Distinguished name of the entity that the certificate shall belong to.
- **KeyID - [tas:KeyID]**
  The ID of the key for which the certificate shall be created.
- **Alias - optional [xs:string]**
  The client-defined alias of the certificate to be created.
- **notValidBefore - optional [xs:dateTime]**
  The X.509 not valid before information to be included in the certificate. Defaults to a time equal to or before the device's current time.
- **notValidAfter - optional [xs:dateTime]**
  The X.509 not valid after information to be included in the certificate. Defaults to the time 99991231235959Z as specified in RFC 5280.
- **SignatureAlgorithm - [tas:AlgorithmIdentifier]**
  The signature algorithm to be used for signing the certificate.

- **Extension - optional, unbounded [tas:X509v3Extension]**
  List of X.509v3 extensions to be included in the certificate.

RESPONSE:

- **CertificateID - [tas:CertificateID]**
  The ID of the generated certificate.

FAULTS:

- **ter:Receiver - ter:Action - ter:CertificateCreationFailed**
  The generation of the self-signed certificate failed.
- **ter:Receiver - ter:Action - ter:MaximumNumberOfCertificatesReached**
  The device does not have enough storage space to store the certificate to be created.
- **ter:Sender - ter:InvalidArgVal - ter:UnsupportedX509Version**
  The specified X.509 version is not supported by the device.
- **ter:Sender - ter:InvalidArgVal - ter:KeyID**
  No key is stored under the requested KeyID.
- **ter:Sender - ter:InvalidArgVal - ter:UnsupportedSignatureAlgorithm**
  The specified signature algorithm is not supported by the device.
- **ter:Sender - ter:InvalidArgVal - ter:KeySignatureAlgorithmMismatch**
  The specified public key is an invalid input to the specified signature algorithm.
- **ter:Sender - ter:InvalidArgVal - ter:X509VersionExtensionsMismatch**
  The request contains extensions which are not supported by the X509Version in the request.
- **ter:Sender - ter:InvalidArgVal - ter:InvalidKeyStatus**
  The key with the requested KeyID has an inappropriate status.
- **ter:Sender - ter:InvalidArgVal - ter:InvalidSubject**
  The specified subject is invalid or incomplete.
- **ter:Sender - ter:InvalidArgVal - ter:InvalidDateTime**
  A specified date Time is invalid.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.3.3 UploadCertificate

This operation uploads an X.509 certificate as specified by RFC 5280 in DER encoding and the public key in the certificate to a device's keystore. A device that supports this command shall be able to handle X.509v3 certificates as specified in RFC 5280 and may additionally be able to handle other X.509 certificate formats as indicated by the X.509Versions capability.

Certificates are uniquely identified using certificate IDs, and key pairs are uniquely identified using key IDs. The device shall generate a new certificate ID for the uploaded certificate.

Certain certificate usages, e.g. TLS server authentication, require the private key that corresponds to the public key in the certificate to be present in the keystore. In such cases, the client may indicate that it expects the device to produce a fault if the matching private key for the uploaded certificate is not present in the keystore by setting the PrivateKeyRequired argument in the upload request to *true*.

The uploaded certificate has to be linked to a key pair in the keystore.

If no private key is required for the public key in the certificate and a key pair exists in the keystore with a public key equal to the public key in the certificate, the uploaded certificate is linked to the key pair identified by the supplied key ID by adding a reference from the certificate to the key pair.

If no private key is required for the public key in the certificate and no key pair exists with the public key equal to the public key in the certificate, a new key pair with status *ok* is created with the public key from the certificate, and this key pair is linked to the uploaded certificate by adding a reference from the certificate to the key pair.

If a private key is required for the public key in the certificate, and a key pair exists in the keystore with a private key that matches the public key in the certificate, the uploaded certificate is linked to this key pair by adding a reference from the certificate to the key pair. If a private key is required for the public key and no such keypair exists in the keystore, then NoMatchingPrivateKey fault shall be produced and the certificate shall not be stored in the keystore.

The device shall assign the supplied Alias to the uploaded certificate.

If a new key pair is generated, the device shall assign the supplied KeyAlias to it. Otherwise, the device shall ignore an eventually supplied KeyAlias.

How the link between the uploaded certificate and a key pair is established is illustrated in Figure 4.

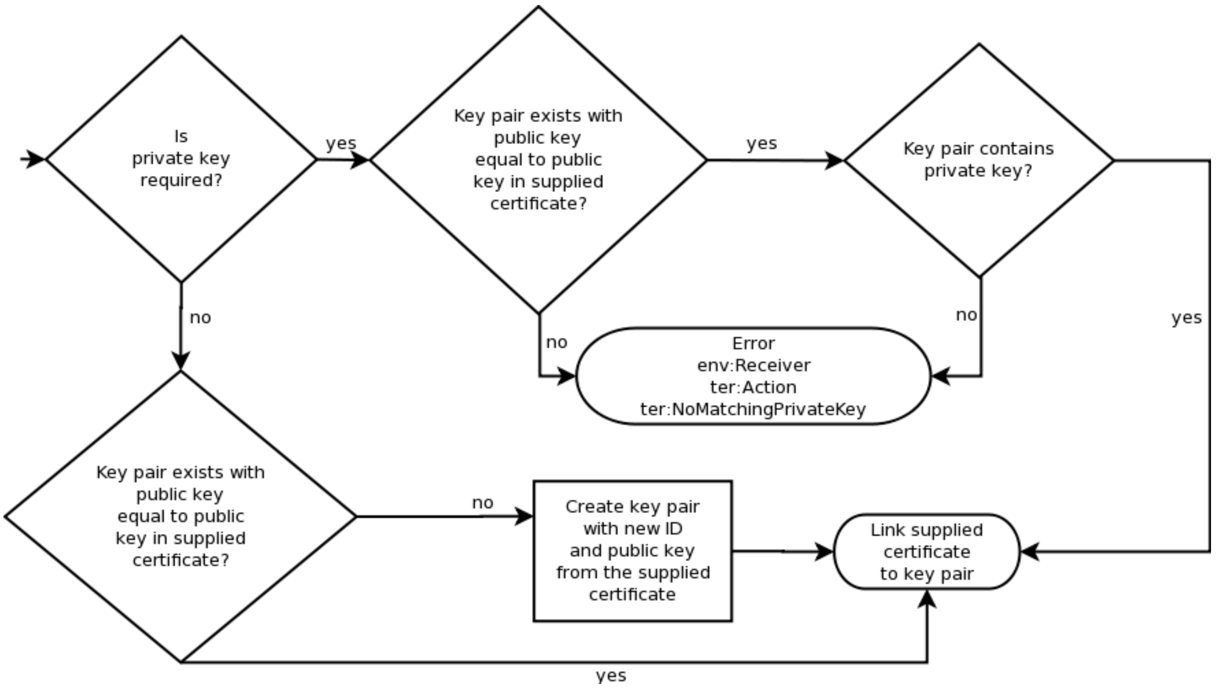
Is private key required?
yes
Key pair exists with public key equal to public key in supplied certificate?
yes
Key pair contains private key?
no
no
no
yes
Error env:Receiver ter:Action ter:NoMatchingPrivateKey
Key pair exists with public key equal to public key in supplied certificate?
no
Create key pair with new ID and public key from the supplied certificate
Link supplied certificate to key pair
yes

Figure 4: Link establishment between certificate and key pair for Upload Certificate

If the key pair that the certificate shall be linked to does not have status *ok*, an InvalidKeyStatus fault is produced, and the uploaded certificate is not stored in the keystore.

If the signature algorithm that the signature of the supplied certificate is based on is not supported by the device, the device shall generate an UnsupportedSignatureAlgorithm fault and shall not store the uploaded certificate nor the contained public key in the keystore.

If the device cannot process the uploaded certificate, a BadCertificate fault is produced and neither the uploaded certificate nor the public key are stored in the device's keystore. The BadCertificate fault shall not be produced based on the mere fact that the device's current time lies outside the interval defined by the notBefore and notAfter fields as specified by RFC 5280, Sect. 4.1.

The device shall not mark the uploaded certificate as trusted.

If the device does not have enough storage capacity for storing the certificate to be uploaded, the maximum number of certificates reached fault shall be produced and no certificate shall be uploaded.

If the device does not have enough storage capacity for storing the key pair that eventually has to be created, the device shall generate a maximum number of keys reached fault. Furthermore the device shall not generate a key pair and no certificate shall be stored.

If the command was successful, the device returns the ID of the uploaded certificate and the ID of the key pair that contains the public key in the certificate.

REQUEST:

- **Certificate - [tas:Base64DERencodedASN1Value]**
  The base64-encoded DER representation of the X.509 certificate to be uploaded.
- **Alias - optional [xs:string]**
  The client-defined alias of the certificate.
- **KeyAlias - optional [xs:string]**
  The client-defined alias of the key pair.
- **PrivateKeyRequired - optional [xs:boolean]**
  Indicates if the device shall verify that a matching key pair with a private key exists in the keystore.

RESPONSE:

- **CertificateID - [tas:CertificateID]**
  The ID of the uploaded certificate.
- **KeyID - [tas:KeyID]**
  The ID of the key that the uploaded certificate certifies.

FAULTS:

- **ter:Receiver - ter:Action - ter:MaximumNumberOfCertificatesReached**
  The device does not have enough storage space to store the certificate to be uploaded.
- **ter:Receiver - ter:Action - ter:MaximumNumberOfKeysReached**
  The device does not have enough storage space to store the key pair to be uploaded.
- **ter:Receiver - ter:Action - ter:NoMatchingPrivateKey**
  The keystore does not contain a key pair with a private key that matches the public key in the uploaded certificate.
- **ter:Sender - ter:InvalidArgVal - ter:BadCertificate**
  The supplied certificate file cannot be processed by the device.
- **ter:Sender - ter:InvalidArgVal - ter:UnsupportedPublicKeyAlgorithm**
  The public key algorithm of the public key in the certificate is not supported by the device.
- **ter:Sender - ter:InvalidArgVal - ter:UnsupportedSignatureAlgorithm**
  The specified signature algorithm is not supported by the device.
- **ter:Sender - ter:InvalidArgVal - ter:InvalidKeyStatus**
  The key pair has an inappropriate status.
- **ter:Sender - ter:InvalidArgVal - ter:Duplicate**
  The certificate already exists.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.3.4 UploadCertificateWithPrivateKeyInPKCS12

This operation uploads a certification path consisting of X.509 certificates as specified by RFC 5280 in DER encoding along with a private key to a device's keystore. Certificates and private key are supplied in the form of a PKCS#12 file as specified in PKCS#12.

The device shall support PKCS#12 files that contain the following safe bags:

- one or more instances of CertBag PKCS#12, Sect. 4.2.3
- either exactly one instance of KeyBag PKCS#12, Sect. 4.3.1 or exactly one instance of PKCS8ShroudedKeyBag PKCS#12, Sect. 4.2.2.

If the IgnoreAdditionalCertificates parameter has the value *true*, the device shall behave as if the client had supplied only the first CertBag in the sequence of CertBag instances.

The device shall support PKCS#12 passphrase integrity mode for integrity protection of the PKCS#12 PFX as specified in PKCS#12, Sect. 4. The device shall support PKCS8ShroudedKeyBags that are encrypted with the same passphrase as the CertBag instances.

If an integrity passphrase ID is supplied, the device shall use the corresponding passphrase in the keystore to check the integrity of the supplied PKCS#12 PFX. If an integrity passphrase ID is supplied, but the supplied PKCS#12 PFX has no integrity protection, the device shall produce a BadPKCS12File fault and shall not store the uploaded certificates nor the uploaded key pair in the keystore.

If an encryption passphrase ID is supplied, the device shall use the corresponding passphrase in the keystore to decrypt the PKCS8ShroudedKeyBag and the CertBag instances.

If an EncryptionPassphraseID is supplied, but a CertBag is not encrypted, the device shall ignore the supplied EncryptionPassphraseID when processing this CertBag. If an EncryptionPassphraseID is supplied, but a KeyBag is provided instead of a PKCS8ShroudedKeyBag, the device shall ignore the supplied EncryptionPassphraseID when processing the KeyBag.

If a passphrase is supplied, the device shall ignore an eventually supplied integrity passphrase ID and an eventually supplied encryption passphrase ID, and the device shall use the supplied passphrase to check the integrity of the PKCS#12 PFX and to decrypt the PKCS8ShroudedKeyBag and the CertBag instances. If a passphrase is supplied, but a CertBag is not encrypted, the device shall ignore the supplied passphrase when processing this CertBag. If a passphrase is supplied, but a KeyBag is supplied instead of a PKCS8ShroudedKeyBag, the device shall ignore the supplied passphrase when processing the KeyBag.

If decryption of either the PKCS8ShroudedKeyBag or an encrypted CertBag failed, the device shall produce a DecryptionFailed fault and shall not store the uploaded certificates nor key pair in the keystore.

If the signature algorithm of a supplied certificate is not supported by the device, the device shall produce an UnsupportedSignatureAlgorithm fault and shall not upload a certificate nor key pair.

If the supplied key pair cannot be processed by the device, the device shall produce an UnsupportedPublicKeyAlgorithm fault and shall not store the uploaded key pair nor the uploaded certificates in the keystore.

Certificates are uniquely identified using certificate IDs. The device shall store the uploaded certificates in the keystore and generate a new certificate ID for each of the uploaded certificates.

Certification paths are uniquely identified using certification path IDs. The device shall create a certification path from the uploaded certificates. In this certification path, the certificates shall appear in the same order as in the PKCS#12 file. The device shall generate a new certification path ID for the created certification path and assign the eventually supplied CertificationPathAlias to the created certification path.

The signature of each certificate in the sequence of uploaded certificates except for the last one shall be verifiable with the public key contained in the next certificate in the sequence. If there is a certificate in the request other than the last certificate for which the signature cannot be verified with the public key in the next certificate, the device shall produce an invalid certification path fault and shall not store the uploaded certificates nor uploaded private key in the keystore.

If the device cannot process one of the uploaded certificates, it shall produce a BadCertificate fault and neither store the uploaded certificates nor private key in the keystore. The BadCertificate fault shall not be produced based on the mere fact that the device’s current time lies outside the interval defined by the notBefore and notAfter fields as specified by RFC 5280, Sect. 4.1.

The device shall not mark the uploaded certificates as trusted.

The uploaded certificates have to be linked to key pairs in the keystore. Key pairs are uniquely identified using key IDs.

If a key pair exists in the keystore with the public key equal to the public key in a certificate in the request, the device shall link the uploaded certificate to the key pair in the keystore by adding a reference from the certificate to the key pair. If the key pair in the keystore does not contain a private key and the private key contained in the KeyBag or PKCS8ShroudedKeyBag that matches the public key in the key pair, the device shall add the private key contained in the KeyBag or PKCS8ShroudedKeyBag to the key pair.

If no key pair exists in the keystore with the public key equal to the public key in a certificate in the request, the device shall create a new key pair with status ok, externally generated attribute set to *true*, and the public and private keys from the request, and shall link this key pair to the uploaded certificate by adding a reference from the certificate to the key pair.

If a new key pair is created for the uploaded private key, the device shall assign the supplied KeyAlias to it. Otherwise, the device shall ignore an eventually supplied KeyAlias.

How the link between an uploaded certificate and a key pair is established is illustrated in Figure 5.

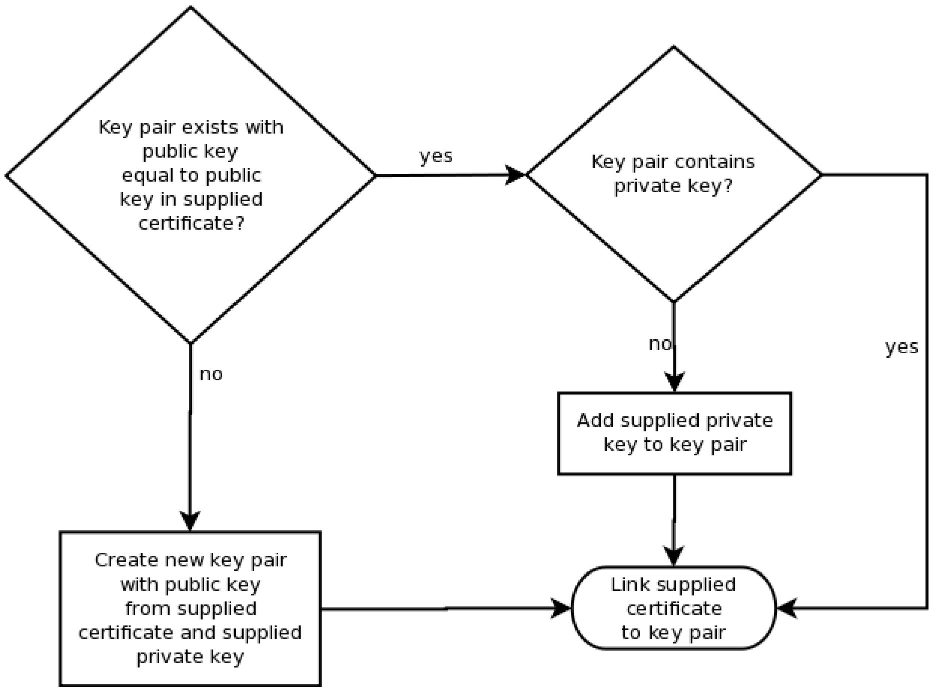
Key pair exists with public key equal to public key in supplied certificate?
yes
Key pair contains private key?
no
yes
no
Add supplied private key to key pair
Create new key pair with public key from supplied certificate and supplied private key
Link supplied certificate to key pair

Figure 5: Link establishment between certificates and key pair for Upload Certificate with Private Key in PKCS#12

If the key pair that a certificate shall be linked to does not have status *ok*, the device shall produce an invalid key status fault and shall not store the uploaded certificates nor the uploaded key pair in the keystore.

If the device does not have enough storage capacity for storing the certificates to be uploaded, the device shall produce a maximum number of certificates reached fault and shall not store the uploaded certificates nor the uploaded key pair in the keystore.

If the device does not have enough storage capacity for storing the key pair that eventually has to be created, the device shall generate a maximum number of keys reached fault. Furthermore the device shall not store a key pair and shall not store the uploaded certificates in the keystore.

If the device does not have enough storage capacity for storing the certification path to be created, the device shall produce a maximum number of certification paths reached fault and shall not store the uploaded certificates nor the uploaded key pair in the keystore.

If no passphrase exists under the ID specified by IntegrityPassphraseID, the device shall produce an invalid passphrase ID fault and shall not store the uploaded certificates nor the uploaded key pair in the keystore.

If no passphrase exists under the ID specified by EncryptionPassphraseID, the device shall produce an invalid passphrase ID fault and shall not store the uploaded certificates nor the uploaded key pair in the keystore.

If the supplied PKCS#12 data structure cannot be processed by the device, the device shall produce a BadPKCS12File fault and shall not store the uploaded certificates nor the uploaded key pair in the keystore.

If the public key in the first uploaded certificate does not match the uploaded private key, the device shall produce a PublicPrivateKeyMismatch fault and shall not store the uploaded certificates nor the uploaded key pair in the keystore.

If the command was successful, the device shall return the ID of the created certification path and the ID of the key pair that contains the public key in the certificate.

REQUEST:

- **CertWithPrivateKey - [tas:Base64DERencodedASN1Value]**
  The certifcates and key pair to be uploaded in a PKCS#12 data structure.
- **CertificationPathAlias - optional [xs:string]**
  The client-defined alias of the certification path.
- **KeyAlias - optional [xs:string]**
  The client-defined alias of the key pair.
- **IgnoreAdditionalCertificates - optional [xs:boolean]**
  True if and only if the device shall behave as if the client had only supplied the first certificate in the sequence of certificates.
- **IntegrityPassphraseID - optional [tas:PassphraseID]**
  The ID of the passphrase to use for integrity checking of the uploaded PKCS#12 data structure.
- **EncryptionPassphraseID - optional [tas:PassphraseID]**
  The ID of the passphrase to use for decrypting the uploaded PKCS#12 data structure.
- **Passphrase - optional [xs:string]**
  The passphrase to use for integrity checking and decrypting the uploaded PKCS#12 data structure.

RESPONSE:

- **CertificationPathID - [tas:CertificationPathID]**
  The certification path ID of the uploaded certification path.
- **KeyID - [tas:KeyID]**
  The key ID of the uploaded key pair.

FAULTS:

- **ter:Receiver - ter:Action - ter:**
  MaximumNumberOfCertificatesReached The device does not have enough storage space to store the certificate to be uploaded.
- **ter:Receiver - ter:Action - ter:**
  MaximumNumberOfKeysReached The device does not have enough storage space to store the key pair that has to be generated.
- **ter:Receiver - ter:Action - ter:**
  MaximumNumberOfCertificationPathsReached The device does not have enough storage space to store the certification path to be uploaded.
- **ter:Sender - ter:InvalidArgVal - ter:PassphraseID**
  No passphrase is stored under the requested PassphraseID.

- **env:Sender - ter:InvalidArgVal - ter:DecryptionFailed**
  The given data could not be decrypted.
- **env:Sender - ter:InvalidArgVal - ter:BadCertificate**
  The supplied certificate file cannot be processed by the device.
- **env:Sender - ter:InvalidArgVal - ter:UnsupportedPublicKeyAlgorithm**
  The public key algorithm of the public key in the certificate is not supported by the device.
- **env:Sender - ter:InvalidArgVal - ter:UnsupportedSignatureAlgorithm**
  The signature algorithm that the signature of the supplied certificate is based on is not supported by the device.
- **env:Sender - ter:InvalidArgVal - ter:InvalidKeyStatus**
  The key with the requested KeyID has an inappropriate status.
- **env:Sender - ter:InvalidArgVal - ter:BadPKCS12File**
  The PKCS#12 data structure cannot be processed by the device.
- **env:Sender - ter:InvalidArgVal - ter:PublicPrivateKeyMismatch**
  The supplied private key does not match the supplied public key.
- **env:Sender - ter: InvalidArgVal - ter:InvalidCertificationPath**
  At least one certificate in the certification path is not correctly signed with the public key in the next certificate in the path.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.3.5 GetCertificate

This operation returns a specific certificate from the device's keystore.

Certificates are uniquely identified using certificate IDs. If no certificate is stored under the requested certificate ID in the keystore, an InvalidArgVal fault is produced.

The certificate shall be returned in DER encoding.

It shall be noted that this command does not return the private key that is associated with the public key in the certificate.

REQUEST:

- **Certificateid - [tas:CertificateID]**
  The ID of the certificate to retrieve.

RESPONSE:

- **Certificate - [tas:X509Certificate]**
  The DER representation of the certificate.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertificateID**
  No certificate is stored under the requested CertificateID.

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.3.6 GetAllCertificates

This operation returns all certificates that are stored in the device's keystore.

This operation may be used, e.g., if a client lost track of which certificates are present on the device.

The certificates shall be returned in DER encoding.

If no certificate is stored in the device's keystore, an empty list is returned.

REQUEST:

This message is empty.

RESPONSE:

- **Certificate - optional, unbounded [tas:X509Certificate]**
  List of DER representation of certificates stored in the keystore.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.3.7 DeleteCertificate

This operation deletes a certificate from the device's keystore.

The operation shall not delete the public key that is contained in the certificate from the keystore.

Certificates are uniquely identified using certificate IDs. If no certificate is stored under the requested certificate ID in the keystore, an InvalidArgVal fault is produced. If there is a certificate under the requested certificate ID stored in the keystore and the certificate could not be deleted, a CertificateDeletion fault is produced.

If a reference exists for the specified certificate, the certificate shall not be deleted and the corresponding fault shall be produced.

After a certificate has been successfully deleted, the device may assign its former ID to other certificates.

REQUEST:

- **CertificateID - [tas:CertificateID]**
  The ID of the certificate to delete.

RESPONSE:

This message is empty.

FAULTS:

- **ter:Receiver - ter:Action - ter:CertificateDeletionFailed**
  Deleting the certificate with the requested CertificateID failed.
- **ter:Sender - ter:InvalidArgVal - ter:CertificateID**
  No certificate is stored under the requested CertificateID.
- **ter:Sender - ter:InvalidArgVal - ter:ReferenceExists**
  A reference exists for the specified certificate.

ACCESS CLASS:

UNRECOVERABLE

5.2.6.3.8 CreateCertificationPath

This operation creates a sequence of certificates that may be used, e.g., for certification path validation or for TLS server authentication.

Certification paths are uniquely identified using certification path IDs. Certificates are uniquely identified using certificate IDs. A certification path contains a sequence of certificate IDs.

If there is a certificate ID in the sequence of supplied certificate IDs for which no certificate exists in the device's keystore, the corresponding fault shall be produced and no certification path shall be created.

The signature of each certificate in the certification path except for the last one shall be verifiable with the public key contained in the next certificate in the path. If there is a certificate ID in the request other than the last ID for which the corresponding certificate cannot be verified with the public key in the certificate identified by the next certificate ID, an InvalidCertificateChain fault shall be produced and no certification path shall be created.

REQUEST:

- **CertificateIDs - [tas:CertificateIDs]**
  The IDs of the certificates to include in the certification path, where each certificate signature except for the last one in the path must be verifiable with the public key certified by the next certificate in the path.
- **Alias - optional [xs:string]**
  The client-defined alias of the certification path.

RESPONSE:

- **CertificationPathID - [tas:CertificationPathID]**
  The ID of the generated certification path.

FAULTS:

- **env:Receiver - ter:Action - ter:MaximumNumberOfCertificationPathsReached**
  The maximum number of certification paths that may be assigned to the TLS server simultaneously is reached.
- **env:Sender - ter:InvalidArgVal - ter:CertificateID**
  No certificate is stored under the requested CertificateID.
- **env:Sender - ter:InvalidArgVal - ter:InvalidCertificationPath**
  At least one certificate in the certification path is not correctly signed with the public key in the next certificate in the path.
- **env:Receiver - ter:Action - ter:CertificationPathCreationFailed**
  Creating the certification path failed.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.3.9 GetCertificationPath

This operation returns a specific certification path from the device's keystore.

Certification paths are uniquely identified using certification path IDs. If no certification path is stored under the requested ID in the keystore, an InvalidArgVal fault is produced.

REQUEST:

- **CertificationPathID - [tas:CertificationPathID]**
  The ID of the certification path to retrieve.

RESPONSE:

- **CertificationPath - [tas:CertificationPath]**
  The certification path that is stored under the given ID in the keystore.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertificationPathID**
  No certification path is stored under the requested certification path ID.

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.3.10 GetAllCertificationPaths

This operation returns the IDs of all certification paths that are stored in the device's keystore.

This operation may be used, e.g., if a client lost track of which certificates are present on the device.

If no certification path is stored on the device, an empty list is returned.

REQUEST:

This message is empty.

RESPONSE:

- **CertificationPathID - optional, unbounded [tas:CertificationPathID]**
  List of IDs of certification paths stored in the keystore.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.3.11 SetCertificationPath

This operation allows to modify a certification path. A device shall support this method if support for SetCertPath is signaled via its capabilities.

REQUEST:

- **CertificationPathID - [tas:CertificationPathID]**
  The ID of the certification path to modify.
- **CertificateIDs - [tas:CertificateIDs]**
  The IDs of the certificates to include in the certification path, where each certificate signature except for the last one in the path must be verifiable with the public key certified by the next certificate in the path.
- **Alias - optional [xs:string]**
  The client-defined alias of the certification path.

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertificationPathID**
  No certification path is stored under the requested certification path ID.
- **env:Sender - ter:InvalidArgVal - ter:CertificateID**
  No certificate is stored under the requested CertificateID.

- **env:Sender - ter:InvalidArgVal - ter:InvalidCertificationPath**
  At least one certificate in the certification path is not correctly signed with the public key in the next certificate in the path.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.3.12 DeleteCertificationPath

This operation deletes a certification path from the device's keystore.

This operation shall not delete the certificates that are referenced by the certification path.

Certification paths are uniquely identified using certification path IDs. If no certification path is stored under the requested certification path ID in the keystore, an InvalidArgVal fault is produced. If there is a certification path under the requested certification path ID stored in the keystore and the certification path could not be deleted, a CertificationPathDeletion fault is produced.

If a reference exists for the specified certification path, the certification path shall not be deleted and the corresponding fault shall be produced.

After a certification path is successfully deleted, the device may assign its former ID to other certification paths.

REQUEST:

- **CertificationPathID - [tas:CertificationPathID]**
  The ID of the certification path to delete.

RESPONSE:

This message is empty.

FAULTS:

- **ter:Receiver - ter:Action - ter:CertificationPathDeletionFailed**
  Deleting the certification path with the requested certification path ID failed.
- **ter:Sender - ter:InvalidArgVal - ter:CertificationPathID**
  No certification path is stored under the requested certification path ID.
- **ter:Sender - ter:InvalidArgVal - ter:ReferenceExists**
  A reference exists for the object that is to be deleted.

ACCESS CLASS:

UNRECOVERABLE

5.2.6.4 CRL Management

5.2.6.4.1 UploadCRL

This operation uploads a certificate revocation list (CRL) as specified in RFC 5280 to the keystore on the device.

If the device does not have enough storage space to store the CRL to be uploaded, the device shall produce a MaximumNumberOfCRLsReached fault and shall not store the supplied CRL.

If the device is not able to process the supplied CRL, the device shall produce a BadCRL fault and shall not store the supplied CRL.

If the device does not support the signature algorithm that was used to sign the supplied CRL, the device shall produce an UnsupportedSignatureAlgorithm fault and shall not store the supplied CRL.

REQUEST:

- **Crl - [tas:Base64DERencodedASN1Value]**
  The CRL to be uploaded to the device.
- **Alias - optional [xs:string]**
  The alias to assign to the uploaded CRL.
- **anyParameters - optional, unbounded [xs:any]**

RESPONSE:

- **CrlID - [tas:CRLID]**
  The ID of the uploaded CRL.

FAULTS:

- **env:Receiver - ter:Action - ter:MaximumNumberOfCRLsReached**
  The device does not have enough storage space to store the CRL to be uploaded.
- **env:Sender - ter:InvalidArgVal - ter:BadCRL**
  The supplied CRL cannot be processed by the device.
- **env:Sender - ter:InvalidArgVal - ter:UnsupportedSignatureAlgorithm**
  The specified signature algorithm is not supported by the device.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.4.2 GetCRL

This operation returns a specific certificate revocation list (CRL) from the keystore on the device.

Certification revocation lists are uniquely identified using CRLIDs. If no CRL is stored under the requested CRLID, the device shall produce a CRLID fault.

REQUEST:

- **CrlID - [tas:CRLID]**
  The ID of the CRL to be returned.

RESPONSE:

- **Crl - [tas:CRL]**
  The CRL with the requested ID.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CRLID**
  No CRL is stored under the requested CRL ID.

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.4.3 GetAllCRLs

This operation returns all certificate revocation lists (CRLs) that are stored in the keystore on the device.

If no certificate revocation list is stored in the device’s keystore, an empty list is returned.

REQUEST:

This message is empty.

RESPONSE:

- **Crl - optional, unbounded [tas:CRL]**
  A list of all CRLs that are stored in the keystore on the device.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.4.4 DeleteCRL

This operation deletes a certificate revocation list (CRL) from the keystore on the device.

Certification revocation lists are uniquely identified using CRLIDs. If no CRL is stored under the requested CRLID, the device shall produce a CRLID fault.

If a reference exists for the specified CRL, the device shall produce a ReferenceExists fault and shall not delete the CRL.

After a CRL has been successfully deleted, a device may assign its former ID to other CRLs.

REQUEST:

- **CrlID - [tas:CRLID]**
  The ID of the CRL to be deleted.

RESPONSE:

This message is empty.

FAULTS:

- **ter:Sender - ter:InvalidArgVal - ter:CRLID**
  No CRL is stored under the requested CRL ID.
- **ter:Sender - ter:InvalidArgVal - ter:ReferenceExists**
  A reference exists for the object that is to be deleted.

ACCESS CLASS:

UNRECOVERABLE

5.2.6.5 Certification Path Validation Policy Management

5.2.6.5.1 CreateCertPathValidationPolicy

This operation creates a certification path validation policy.

Certification path validation policies are uniquely identified using certification path validation policy IDs. The device shall generate a new certification path validation policy ID for the created certification path validation policy.

If the device does not have enough storage capacity for storing the certification path validation policy to be created, the device shall produce a maximum number of certification path validation policies reached fault and shall not create a certification path validation policy.

If there is at least one trust anchor certificate ID in the request for which there exists no certificate in the device's keystore, the device shall produce a CertificateID fault and shall not create a certification path validation policy.

If the device cannot process the supplied certification path validation parameters, the device shall produce a CertPathValidationParameters fault and shall not create a certification path validation policy.

REQUEST:

- **Alias - optional [xs:string]**
  The alias to assign to the created certification path validation policy.
- **Parameters - [tas:CertPathValidationParameters]**
  The parameters of the certification path validation policy to be created.
- **TrustAnchor - unbounded [tas:TrustAnchor]**
  The trust anchors of the certification path validation policy to be created.
- **anyParameters - optional [xs:any]**

RESPONSE:

- **CertPathValidationPolicyID - [tas:CertPathValidationPolicyID]**
  The ID of the created certification path validation policy.

FAULTS:

- **env:Receiver - ter:Action - ter:MaximumNumberOfCertPathValidationPoliciesReached**
  The device does not have enough storage to store the certification path validation policy to be created.
- **env:Sender - ter:InvalidArgVal - ter:CertificateID**
  No certificate is stored under the requested CertificateID.
- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationParameters**
  The specified certification path validation parameters are invalid.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.5.2 GetCertPathValidationPolicy

This operation returns a certification path validation policy from the keystore on the device.

Certification path validation policies are uniquely identified using certification path validation policy IDs. If no certification path validation policy is stored under the requested certification path validation policy ID, the device shall produce a CertPathValidationPolicyID fault.

REQUEST:

- **CertPathValidationPolicyID - [tas:CertPathValidationPolicyID]**
  The ID of the certification path validation policy to be created.

RESPONSE:

- **CertPathValidationPolicy - [tas:CertPathValidationPolicy]**
  The certification path validation policy that is stored under the requested ID.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationPolicyID**
  No certification path validation policy is stored under the requested certification path validation policy ID.

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.5.3 GetAllCertPathValidationPolicies

This operation returns all certification path validation policies that are stored in the keystore on the device.

If no certification path validation policy is stored in the device's keystore, an empty list is returned.

REQUEST:

This message is empty.

RESPONSE:

- **CertPathValidationPolicy - optional, unbounded - [tas:CertPathValidationPolicy]**
  A list of all certification path validation policies that are stored in the keystore on the device.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.2.6.5.4 SetCertPathValidationPolicy

This operation allows to modify a certification path validation policy in the keystore on the device. A device shall support this method if support for SetCertPath is signaled via its capabilities.

REQUEST:

- **CertPathValidationPolicyID - [tas:CertPathValidationPolicyID]**
  The ID of the certification path validation policy to be modified.
- **Parameters - [tas:CertPathValidationParameters]**
  The parameters of the certification path validation policy.
- **TrustAnchor - unbounded [tas:TrustAnchor]**
  The trust anchors of the certification path validation policy.

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationPolicyID**
  No certification path validation policy is stored under the requested certification path validation policy ID.
- **env:Sender - ter:InvalidArgVal - ter:CertificateID**
  No certificate is stored under the requested CertificateID.
- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationParameters**
  The specified certification path validation parameters are invalid.

ACCESS CLASS:

WRITE_SYSTEM

5.2.6.5.5 DeleteCertPathValidationPolicy

This operation deletes a certification path validation policy from the keystore on the device.

Certification path validation policies are uniquely identified using certification path validation policy IDs. If no certification path validation policy is stored under the requested certification path validation policy ID, the device shall produce an CertPathValidationPolicyID fault.

If a reference exists for the requested certification path validation policy, the device shall produce a ReferenceExists fault and shall not delete the certification path validation policy.

After the certification path validation policy has been deleted, the device may assign its former ID to other certification path validation policies.

REQUEST:

- **CertPathValidationPolicyID - [tas:CertPathValidationPolicyID]**
  The ID of the certification path validation policy to be deleted.

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationPolicyID**
  No certification path validation policy is stored under the requested certification path validation policy ID.
- **ter:Sender - ter:InvalidArgVal - ter:ReferenceExists**
  A reference exists for the object that is to be deleted.

ACCESS CLASS:

UNRECOVERABLE

5.3 TLS Server

5.3.1 Elements of the TLS Server

The TLS server security feature implements a TLS server as specified in RFC 2246 and subsequent specifications.

This specification defines how to manage the associations between certification paths and the TLS server. All other TLS server configuration actions are outside the scope of this specification. In particular, enabling and disabling the TLS server on the device shall be performed using the device management service specified in the ONVIF Core Specification.

5.3.2 Authorization of TLS authenticated connections

If TLS client authentication is enabled, connections shall be authenticated as specified in RFC 2246, and the device shall before all validate the client TLS certificate. In case of invalid certificate the TLS connection shall be terminated and the device shall ignore any other credentials received on HTTP or WS layer.

Once a service request is authenticated on the TLS layer, the device shall decide based on its access policy whether the requestor is authorized to receive the service. In order to authorize the requestor, additional information for the device is required.

If CnMapsToUser is true, the name of the user requiring access to the device shall be presented in the Common Name (CN) attribute of the certificate presented by the client to the device. The device shall validate the provided username against its set of credentials, and grant access to the requested function in case of success. If the user is not allowed to access the function, the device shall return a 403 Forbidden.

If CnMapsToUser is false, from this point forward the authorization procedure follows what is specified in the ONVIF Core Specification as part of the security service.

The authentication and authorization process and falling back to the ONVIF Core Specification is illustrated in Figure 6.

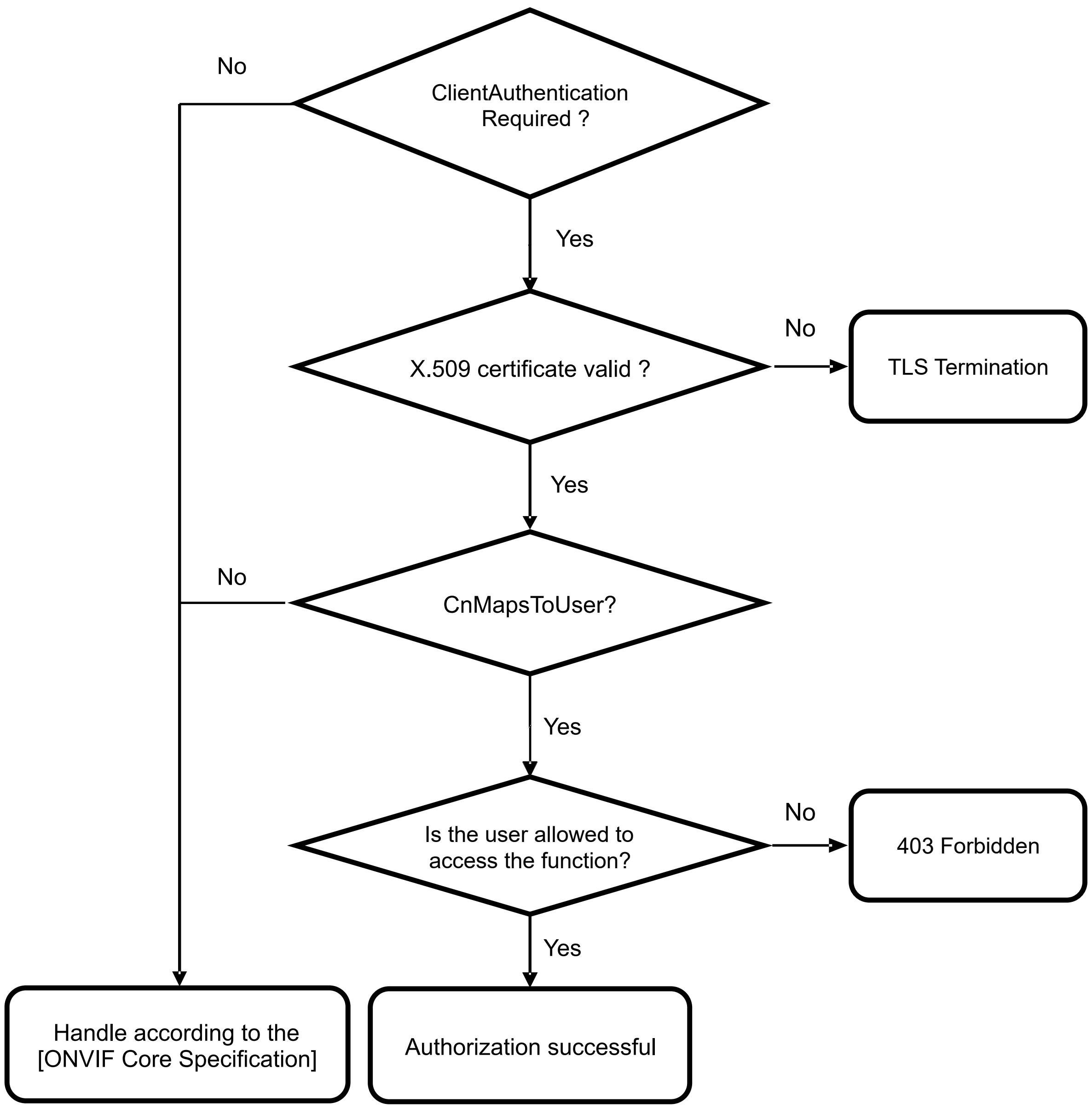
ClientAuthentication Required ?
No
Yes
X.509 certificate valid ?
No
TLS Termination
Yes
CnMapsToUser?
No
Yes
Is the user allowed to access the function?
No
403 Forbidden
Yes
Handle according to the [ONVIF Core Specification]
Authorization successful

Figure 6: Authentication and authorization flow chart and fallback mechanism to the ONVIF Core Specification.

5.3.3 TLS Server Operations

5.3.3.1 AddServerCertificateAssignment

This operation assigns a key pair and certificate along with a certification path (certificate chain) to the TLS server on the device. The TLS server shall use this information for key exchange during the TLS handshake, particularly for constructing server certificate messages as specified in RFC 4346, RFC 2246.

Certification paths are identified by their certification path IDs in the keystore. The first certificate in the certification path shall be the TLS server certificate.

Since each certificate has exactly one associated key pair, a reference to the key pair that is associated with the server certificate is not supplied explicitly. Devices shall obtain the private key or results of operations under the private key by suitable internal interaction with the keystore.

If a device chooses to perform a TLS key exchange based on the supplied certification path, it shall use the key pair that is associated with the server certificate for key exchange and transmit the certification path to TLS clients as-is, i.e., the device shall not check conformance of the certification path to RFC 4346, RFC 2246.

In order to use the server certificate during the TLS handshake, the corresponding private key is required. Therefore, if the key pair that is associated with the server certificate, i.e., the first certificate in the certification path, does not have an associated private key, the NoPrivateKey fault is produced and the certification path is not associated with the TLS server.

A TLS server may present different certification paths to different clients during the TLS handshake instead of presenting the same certification path to all clients. Therefore more than one certification path may be assigned to the TLS server. If the maximum number of certification paths that may be assigned to the TLS server simultaneously is reached, the device shall generate a MaximumNumberOfTLSCertificationPathsReached fault and the requested certification path shall not be assigned to the TLS server.

If the certification path identified by the supplied certification path ID is already assigned to the TLS server, this command shall have no effect.

REQUEST:

- **CertificationPathID - [tas:CertificationPathID]**
  The ID of the certification path to assign to the TLS server.

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertificationPathID**
  No certification path is stored in the keystore under the given certification path ID.
- **env:Sender - ter:InvalidArgVal - ter:NoPrivateKey**
  The key pair that is associated with the first certificate in the certificate chain does not have an associated private key.
- **env: Receiver - ter: Action - ter:MaximumNumberOfTLSCertificationPathsReached**
  The maximum number of certification paths that may be assigned to the TLS server simultaneously is reached.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.2 RemoveServerCertificateAssignment

This operation removes a key pair and certificate assignment (including certification path) to the TLS server on the device.

Certification paths are identified using certification path IDs. If the supplied certification path ID is not associated with the TLS server, an InvalidArgVal fault is produced.

If the TLS server on the device is enabled, the device shall produce a ReferenceExists fault and shall not remove the server certificate assignment.

REQUEST:

- **CertificationPathID - [tas:CertificationPathID] The ID of the certification path to remove from the TLS server.**

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:OldCertificationPathID**
  No certification path under the given certification path ID is associated with the TLS server.

- **env:Sender - ter:InvalidArgVal - ter:ReferenceExists**
  A reference exists for the object that is to be deleted.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.3 ReplaceServerCertificateAssignment

This operation replaces an existing key pair and certificate assignment to the TLS server on the device by a new key pair and certificate assignment (including certification paths).

After the replacement, the TLS server shall use the new certificate and certification path exactly in those cases in which it would have used the old certificate and certification path. Therefore, especially in the case that several server certificates are assigned to the TLS server, clients that wish to replace an old certificate assignment by a new assignment should use this operation instead of a combination of the Add TLS Server Certificate Assignment and the Remove TLS Server Certificate Assignment operations.

Certification paths are identified using certification path IDs. If the supplied old certification path ID is not associated with the TLS server, or no certification path exists under the new certification path ID, the corresponding InvalidArgVal faults are produced and the associations are unchanged.

The first certificate in the new certification path shall be the TLS server certificate.

Since each certificate has exactly one associated key pair, a reference to the key pair that is associated with the new server certificate is not supplied explicitly. Devices shall obtain the private key or results of operations under the private key by suitable internal interaction with the keystore.

If a device chooses to perform a TLS key exchange based on the new certification path, it shall use the key pair that is associated with the server certificate for key exchange and transmit the certification path to TLS clients as-is, i.e., the device shall not check conformance of the certification path to RFC 4346, RFC 2246.

In order to use the server certificate during the TLS handshake, the corresponding private key is required. Therefore, if the key pair that is associated with the server certificate, i.e., the first certificate in the certification path, does not have an associated private key, the NoPrivateKey fault is produced and the certification path is not associated with the TLS server.

REQUEST:

- **OldCertificationPathID - [tas:CertificationPathID]**
  The ID of the certification path to remove from the TLS server.
- **NewCertificationPathID - [tas:CertificationPathID]**
  The ID of the certification path to assign to the TLS server.

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:OldCertificationPathID**
  No certification path under the given certification path ID is associated with the TLS server.
- **env:Sender - ter:InvalidArgVal - ter:NewCertificationPathID**
  No certification path is stored in the keystore under the given certification path ID.
- **env:Sender - ter:InvalidArgVal - ter:NoPrivateKey**
  The key pair that is associated with the leaf certificate in the certificate chain does not have an associated private key.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.4 GetAssignedServerCertificates

This operation returns the IDs of all certification paths that are assigned to the TLS server on the device.

This operation may be used, e.g., if a client lost track of the certification path assignments on the device.

If no certification path is assigned to the TLS server, an empty list is returned.

REQUEST:

This message is empty.

RESPONSE:

- **CertificationPathID - optional, unbounded [tas:CertificationPathID]**
  List of certification path IDs assigned to the TLS server.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.3.3.5 SetClientAuthenticationRequired

This operation activates or deactivates TLS client authentication for the TLS server on the device.

The TLS server on the device shall require client authentication if and only if clientAuthenticationRequired is set to *true*.

If TLS client authentication is requested to be enabled and no certification path validation policy is assigned to the TLS server, the device shall return an EnablingClientAuthenticationFailed fault and shall not enable TLS client authentication.

The device shall execute this command regardless of the TLS enabled/disabled state configured in the ONVIF Device Management Service.

REQUEST:

- **clientAuthenticationRequired - [xs:boolean]**
  Define whether TLS client authentication is active on the device.

RESPONSE:

This message is empty.

FAULTS:

- **env:Receiver - ter:ActionNotSupported - ter:EnablingClientAuthenticationFailed**
  The device does not support TLS client authentication, or TLS client authentication is not configured appropriately.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.6 GetClientAuthenticationRequired

This operation returns whether TLS client authentication is active.

REQUEST:

This message is empty.

RESPONSE:

- **clientAuthenticationRequired - [xs:boolean]**
  Report whether TLS client authentication is active on the device.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM

5.3.3.7 SetCnMapsToUser

This operation enables or disables mapping of the Common Name present in the TLS client certificate to an existing user name in the device.

The TLS server on the device shall perform mapping if parameter clientAuthenticationRequired is set to *true*.

REQUEST:

- **cnMapsToUser - [xs:boolean]**
  A request for the device to enable or disable Common Name Mapping to User.

RESPONSE:

This message is empty.

FAULTS:

- **env:Receiver - ter:ActionNotSupported - ter:CnMapsToUserFailed**
  The device does not support TLS client authentication, or TLS client authentication is not configured appropriately.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.8 GetCnMapsToUser

This operation returns whether the Common Name Mapping to User is enabled.

REQUEST:

This message is empty.

RESPONSE:

- **cnMapsToUser - [xs:boolean]**
  Whether cnMapsToUser is enabled.

FAULTS:

- **None**

ACCESS CLASS:

READ_SYSTEM

5.3.3.9 AddCertPathValidationPolicyAssignment

This operation assigns a certification path validation policy to the TLS server on the device. The TLS server shall enforce the policy when authenticating TLS clients and consider a client authentic if Certificaton Path Validation according to section 6 of RFC 5280 succeeds.

If no certification path validation policy is stored under the requested CertPathValidationPolicyID, the device shall produce a CertPathValidationPolicyID fault.

A TLS server may use different certification path validation policies to authenticate clients. Therefore more than one certification path validation policy may be assigned to the TLS server. If the maximum number of certification path validation policies that may be assigned to the TLS server simultaneously is reached, the device shall produce a MaximumNumberOfTLSCertPathValidationPoliciesReached fault and shall not assign the requested certification path validation policy to the TLS server.

REQUEST:

- **CertPathValidationPolicyID - [tas:CertPathValidationPolicyID]**
  The ID of the certification path validation policy to assign to the TLS server.

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationPolicyID**
  No certification path validation policy is stored under the requested CertPathValidationPolicyID.
- **env:Receiver - ter:Action - ter:MaximumNumberOfTLSCertPathValidationPoliciesReached**
  The maximum number of certification path validation policies that may be assigned to the TLS server simultaneously is reached.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.10 RemoveCertPathValidationPolicyAssignment

This operation removes a certification path validation policy assignment from the TLS server on the device.

If the certification path validation policy identified by the requested CertPathValidationPolicyID is not associated to the TLS server, the device shall produce a CertPathValidationPolicyID fault.

REQUEST:

- **CertPathValidationPolicyID - [tas:CertPathValidationPolicyID]**
  The ID of the certification path validation policy to remove from the TLS server.

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationPolicyID**
  No certification path validation policy is stored under the requested CertPathValidationPolicyID.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.11 ReplaceCertPathValidationPolicyAssignment

This operation replaces a certification path validation policy assignment to the TLS server on the device with another certification path validation policy assignment.

If the certification path validation policy identified by the requested OldCertPathValidationPolicyID is not associated to the TLS server, the device shall produce an OldCertPathValidationPolicyID fault and shall not associate the certification path validation policy identified by the NewCertPathValidationPolicyID to the TLS server.

If no certification path validation policy exists under the requested NewCertPathValidationPolicyID in the device's keystore, the device shall produce a NewCertPathValidationPolicyID fault and shall not remove the association of the old certification path validation policy to the TLS server.

REQUEST:

- **OldCertPathValidationPolicyID - [tas:CertPathValidationPolicyID]**
  The ID of the certification path validation policy to remove from the TLS server.
- **NewCertPathValidationPolicyID - [tas:CertPathValidationPolicyID]**
  The ID of the certification path validation policy to assign to the TLS server.
- **RESPONSE:**
  This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:OldCertPathValidationPolicyID**
  No certification path validation policy under the given OldCertPathValidationPolicyID is associated with the TLS server.
- **env:Sender - ter:InvalidArgVal - ter:NewCertPathValidationPolicyID**
  No certification path validation policy under the given NewCertPathValidationPolicyID is stored in the device's keystore.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.12 GetAssignedCertPathValidationPolicies

This operation returns the IDs of all certification path validation policies that are assigned to the TLS server on the device.

REQUEST:

This message is empty.

RESPONSE:

- **CertPathValidationPolicyID - optional, unbounded [tas:CertPathValidationPolicyID]**
  List of certification path validation policy IDs assigned to the TLS server.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.3.3.13 SetEnabledTLSVersions

This operation sets the version(s) of TLS which the device shall use. Valid values are taken from the TLSServerSupported capability.

A client initiates a TLS session by sending a ClientHello with the highest TLS version it supports. This suggests to the server that the client can accept any TLS version up to and including that version.

The server then chooses the TLS version to use. This is generally the highest TLS version the server supports that is within the range of the client. For example, if a ClientHello indicates TLS version 1.1, the server can proceed with TLS 1.0 or TLS 1.1.

In the event that an ONVIF installation wishes to disable certain version(s) of TLS, it may do so with this operation. For example, to disable TLS 1.0 on a device signaling support for TLS versions 1.0, 1.1, and 1.2, the enabled version list may be set to "1.1 1.2", omitting 1.0. If a client then attempts to connect with a ClientHello containing TLS 1.0, the server shall send a "protocol_version" alert message and close the connection. This handshake indicates to the client that TLS 1.0 is not supported by the server. The client must try again with a higher TLS version suggestion.

An empty version list is not permitted. Disabling all versions of TLS is not the intent of this operation. See AddServerCertificateAssignment and RemoveServerCertificateAssignment.

A device signalling support for TLS version enabling with the EnabledVersionsSupported capability shall support this command.

REQUEST:

- **Versions - [tas:TLSVersions]**
  Space-delimited list of TLS versions to allow.

RESPONSE:

This is an empty message.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:EmptyList**
  The version list is empty.
- **env:Sender - ter:InvalidArgVal - ter:TLSVersion**
  A version is not recognized.

ACCESS CLASS:

WRITE_SYSTEM

5.3.3.14 GetEnabledTLSVersions

This operation retrieves the version(s) of TLS which are currently enabled on the device. A device signalling support for TLS version enabling with the EnabledVersionsSupported capability shall support this command.

REQUEST:

This is an empty message.

RESPONSE:

- **Versions - [tas:TLSVersions]**
  Space-delimited list of enabled TLS versions.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM

5.4 IEEE 802.1X

5.4.1 AddDot1XConfiguration

This operation adds an IEEE 802.1X configuration to the device.

Configurations are uniquely identified using IEEE 802.1X configuration IDs. The device shall ignore Dot1XID in the request, if present, and shall generate a unique configuration ID for the added configuration.

If the command was successful, the device shall return the ID of the configuration.

If the device does not have capacity for the configuration, the device shall produce a MaximumNumberOfDot1XConfigurationsReached fault and shall not add the configuration.

If Identity is used as an anonymous identity for the corresponding authentication method, the device shall ignore an eventually supplied passphrase ID in the same Dot1XStage. Otherwise, if the device cannot process a passphrase ID included in the configuration to be added, the device shall produce a PassphraseID fault and shall not add the configuration.

If the device cannot process a certification path ID included in the configuration to be added, the device shall produce a CertificationPathID fault and shall not add the configuration.

If no certification path validation policy is stored under the requested certification path validation policy ID, the device shall produce a CertPathValidationPolicyID fault.

If no certification path validation policy configured, authorization server certificate validation behavior is undefined and the device may either apply a vendor specific default validation policy or skip validation at all.

If the device cannot process an authentication method included in the configuration to be added (e.g., unrecognized method or missing configuration parameter), the device shall produce a Dot1XMethod fault and shall not add the configuration.

A device signalling support for IEEE 802.1X configuration with the MaximumNumberOfDot1XConfigurations capability shall support this command.

REQUEST:

- **Dot1XConfiguration - [tas:Dot1XConfiguration]**
  The desired 802.1X configuration.
- **Alias - optional [xs:string]**
  The client-defined alias of the 802.1X configuration.

RESPONSE:

- **Dot1XID - [tas:Dot1XID]**
  The unique identifier of the created 802.1X configuration.

FAULTS:

- **env:Receiver - ter:Action - ter:MaximumNumberOfDot1XConfigurationsReached**
  The device already has the number of configurations specified by MaximumNumberOfDot1XConfigurations.
- **env:Sender - ter:InvalidArgVal - ter:PassphraseID**
  A supplied passphrase ID cannot be processed by the device.
- **env:Sender - ter:InvalidArgVal - ter:CertificationPathID**
  A supplied certification path ID cannot be processed by the device.
- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationPolicyID**
  No certification path validation policy is stored under the requested certification path validation policy ID.
- **env:Sender - ter:InvalidArgVal - ter:Dot1XMethod**
  A supplied IEEE 802.1X authentication method cannot be processed by the device.
- **env:Sender - ter:InvalidArgVal - ter:Dot1XMethodCombination**
  The combination of IEEE 802.1X authentication methods cannot be processed by the device.

ACCESS CLASS:

WRITE_SYSTEM

5.4.2 GetAllDot1XConfigurations

This operation returns details of all IEEE 802.1X configurations that are on the device. This operation may be used, e.g., if a client lost track of which IEEE 802.1X configurations are present on the device.

If no IEEE 802.1X configurations exist on the device, an empty list is returned.

A device signalling support for IEEE 802.1X configuration with the MaximumNumberOfDot1XConfigurations capability shall support this command.

REQUEST:

This is an empty message.

RESPONSE:

- **Configuration - optional, unbounded [tas:Dot1XConfiguration]**
  The list of 802.1X configurations on the device.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.4.3 GetDot1XConfiguration

This operation returns details of a specific IEEE 802.1X configuration on the device.

If the device cannot process the provided IEEE 802.1X configuration ID, the device shall produce a Dot1XConfigurationID fault.

A device signalling support for IEEE 802.1X configuration with the MaximumNumberOfDot1XConfigurations capability shall support this command.

REQUEST:

- **Dot1XID - [tas:Dot1XID]**
  The unique identifier of the desired 802.1X configuration.

RESPONSE:

- **Dot1XConfiguration - [tas:Dot1XConfiguration]**
  The 802.1X configuration, without password information.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:Dot1XConfigurationID**
  The supplied IEEE 802.1X configuration ID cannot be processed by the device.

ACCESS CLASS:

READ_SYSTEM_SECRET

5.4.4 DeleteDot1XConfiguration

This operation deletes an IEEE 802.1X configuration from the device.

If the device cannot process the provided IEEE 802.1X configuration ID, the device shall produce a Dot1XConfigurationID fault.

If a reference exists for the specified IEEE 802.1X configuration, the device shall produce a ReferenceExists fault and shall not delete the configuration.

After an IEEE 802.1X configuration has been successfully deleted, the device may assign its former ID to a new configuration.

A device signalling support for IEEE 802.1X configuration with the MaximumNumberOfDot1XConfigurations capability shall support this command.

REQUEST:

- **Dot1XID - [tas:Dot1XID]**
  The unique identifier of the 802.1X configuration to be deleted.

RESPONSE:

This is an empty message.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:Dot1XConfigurationID**
  The supplied IEEE 802.1X configuration ID cannot be processed by the device.
- **env:Sender - ter:InvalidArgVal - ter:ReferenceExists**
  A network interface reference exists for the specified IEEE 802.1X configuration.

ACCESS CLASS:

UNRECOVERABLE

5.4.5 SetNetworkInterfaceDot1XConfiguration

This operation binds an IEEE 802.1X configuration to a network interface on the device. This operation shall either create a new binding or replace an existing binding. On failure when an existing binding already exists, the existing binding shall remain.

The Device Management SetNetworkInterface operation provides a method of binding an IEEE 802.1X configuration to an IEEE 802.11 (wireless) interface, and that operation may still be used. But there is no ability for SetNetworkInterface to bind an IEEE 802.1X configuration to a hardwired interface. This operation is provided to bind an IEEE 802.1X configuration to either type of interface.

If SetNetworkInterfaceDot1XConfiguration is used to bind an IEEE 802.1X configuration to an IEEE 802.11 (wireless) interface, then the DeviceManagement GetNetworkInterfaces operation shall return the IEEE 802.1X configuration ID along with the rest of that interface's configuration information.

If the device cannot process the provided network interface token, the device shall produce an InvalidNetworkInterface fault.

If the device cannot process the provided IEEE 802.1X configuration ID, the device shall produce a Dot1XConfigurationID fault.

A device signalling support for IEEE 802.1X configuration with the MaximumNumberOfDot1XConfigurations capability shall support this command.

REQUEST:

- **token - [xs:string]**
  The unique identifier of the Network Interface on which the 802.1X configuration is to be set. (NOTE: the network interface token is defined in devicemgmt.wsdl as tt:ReferenceToken, which is a derived type of xs:string. To avoid importing all of common.xsd for this single type, the base type is used here.)
- **Dot1XID - [tas:Dot1XID]**
  The unique identifier of the 802.1X configuration to be set.

RESPONSE:

- **RebootNeeded - [xs:boolean]**
  Indicates whether or not a reboot is required after configuration updates.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:InvalidNetworkInterface**
  The supplied network interface token does not exist.
- **env:Sender - ter:InvalidArgVal - ter:Dot1XConfigurationID**
  The supplied IEEE 802.1X configuration ID cannot be processed by the device.

ACCESS CLASS:

WRITE_SYSTEM

5.4.6 GetNetworkInterfaceDot1XConfiguration

This operation returns the IEEE 802.1X ID and configuration associated with a network interface on the device. If there is no IEEE 802.1X configuration associated with the specified network interface, then the response shall be empty.

If the Device Management SetNetworkInterface operation was used to bind an IEEE 802.1X configuration to an IEEE 802.11 (wireless) interface, then this operation shall return the IEEE 802.1X configuration information as if the SetNetworkInterfaceDot1XConfiguration operation had been used.

If the device cannot process the provided network interface token, the device shall produce an InvalidNetworkInterface fault.

A device signalling support for IEEE 802.1X configuration with the MaximumNumberOfDot1XConfigurations capability shall support this command.

REQUEST:

- **token - [xs:string]**
  The unique identifier of the Network Interface for which the 802.1X configuration is to be retrieved. (NOTE: the network interface token is defined in devicemgmt.wsdl as tt:ReferenceToken, which is a derived type of xs:string. To avoid importing all of common.xsd for this single type, the base type is used here.)

RESPONSE:

- **Dot1XID - optional [tas:Dot1XID]**
  The unique identifier of 802.1X configuration assigned to the Network Interface.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:InvalidNetworkInterface**
  The supplied network interface token does not exist.

ACCESS CLASS:

READ_SYSTEM_SECRET

5.4.7 DeleteNetworkInterfaceDot1XConfiguration

This operation unbinds the IEEE 802.1X configuration associated with a network interface on the device. If there is no IEEE 802.1X configuration associated with the specified network interface, then the operation does nothing.

The Device Management SetNetworkInterface operation provides a method of unbinding an IEEE 802.1X configuration from an IEEE 802.11 (wireless) interface by omitting the configuration ID, and that operation may still be used. But there is no ability for SetNetworkInterface to unbind an IEEE 802.1X configuration from a hardwired interface. This operation is provided to unbind an IEEE 802.1X configuration from either type of interface.

If the Device Management SetNetworkInterface operation was used to bind an IEEE 802.1X configuration to an IEEE 802.11 (wireless) interface, then this operation shall unbind the IEEE 802.1X configuration information as if the SetNetworkInterfaceDot1XConfiguration operation had been used.

If the device cannot process the provided network interface token, the device shall produce an InvalidNetworkInterface fault.

A device signalling support for IEEE 802.1X configuration with the MaximumNumberOfDot1XConfigurations capability shall support this command.

REQUEST:

- **token - [xs:string]**
  The unique identifier of the Network Interface for which the 802.1X configuration is to be deleted. (NOTE: the network interface token is defined in devicemgmt.wsdl as tt:ReferenceToken, which is a derived type of xs:string. To avoid importing all of common.xsd for this single type, the base type is used here.)

RESPONSE:

- **RebootNeeded - [xs:boolean]**
  Indicates whether or not a reboot is required after configuration updates.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:InvalidNetworkInterface**
  The supplied network interface token does not exist.

ACCESS CLASS:

WRITE_SYSTEM

5.5 Media Signing

5.5.1 Overview

Signing of media that is generated by the device is described in the [Media Signing Specification]. Media is signed using a private key that is provisioned during factory production that is stored in a specially protected hardware component (e.g., a trusted platform module). This private key is associated with a certificate that holds the public key. In addition to the factory provisioned key one additional private key can be used to sign media.

5.5.2 AddMediaSigningCertificateAssignment

This operation assigns certification path (certificate chain) to use for media signing, replacing the one that is provisioned during factory production. The leaf certificate in the chain and its associated private key shall be used for signing media as described in the [Media Signing Specification]. This key and certificate are referred to as user provisioned key and certificate in that specification.

If this operation is called when there is already a user provisioned certification path configured, the existing certification path shall be replaced with the new certification path.

A device shall support this command if the UserMediaSigningKeySupported capability is true.

REQUEST:

- **CertificationPathID - [tas:CertificationPathID] The ID of the certification path to assign for media signing.**

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:CertificationPathID No certification path is stored in the keystore under the given certification path ID.**
- **env:Sender - ter:InvalidArgVal - ter:NoPrivateKey The key pair that is associated with the leaf certificate in the certificate chain does not have an associated private key.**

ACCESS CLASS:

WRITE_SYSTEM

5.5.3 RemoveMediaSigningCertificateAssignment

This operation removes a certificate assignment (including certification path) on the device that has been added by AddMediaSigningCertificateAssignment. The factory provisioned certification path cannot be removed.

If media signing on the device is enabled, the device shall produce a ReferenceExists fault and shall not remove the certificate assignment.

A device shall support this command if the UserMediaSigningKeySupported capability is true.

REQUEST:

- **CertificationPathID - [tas:CertificationPathID] The ID of the certification path to remove.**

RESPONSE:

This message is empty.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:OldCertificationPathID**
  No certification path under the given certification path ID is associated with media signing.
- **env:Sender - ter:InvalidArgVal - ter:ReferenceExists**
  A reference exists for the object that is to be deleted.

ACCESS CLASS:

WRITE_SYSTEM

5.5.4 GetAssignedMediaSigningCertificates

This operation returns the IDs of the certification paths that are assigned for media signing on the device. This operation will always return the factory provisioned certification path and can additionally return a certification path that has been added by AddMediaSigningCertificateAssignment.

A device shall support this command if the MediaSigningSupported capability is true.

REQUEST:

This message is empty.

RESPONSE:

- **CertificationPathID - optional, max 2 [tas:CertificationPathID]**
  List of certification path IDs assigned for media signing. At least one certification path will be returned, the factory provisioned one. At most two certification paths will be returned.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.6 Autorization Server Configuration

This chapter describes configuration of external authorization servers. For an overview of this see 4.6.

5.6.1 GetAuthorizationServerConfigurations

This operation retrieves an existing authorization server configuration, or all existing authorization server configurations if Token is not specified. The device shall support this command if MaxConfigurations capability is greater than zero.

REQUEST:

- **Token - optional [tt:ReferenceToken]**
  Optional configuration token to get.

RESPONSE:

- **Configuration - optional, unbounded [tas:AuthorizationServerConfiguration]**
  List of authorization server configurations.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:NoConfig**
  The requested configuration does not exist.

ACCESS CLASS:

READ_SYSTEM_SECRET

For security reasons a device shall not provide the parameter ClientSecret in the response.

5.6.2 CreateAuthorizationServerConfiguration

This operation creates a new authorization server configuration. The configuration data shall be created in the device and shall be persistent (remain after reboot). The device shall support this command if MaxConfigurations capability is greater than zero.

REQUEST:

- **Configuration - [tas:AuthorizationServerConfigurationData]**
  Details of the configuration that shall be created.

RESPONSE:

- **Token - [tt:ReferenceToken]**
  Token assigned to the newly configuration.

FAULTS:

- **env:Receiver - ter:Action - ter:MaxAuthorizationServers**
  The maximum number of configurations supported by the device has been reached.
- **env:Sender - ter:InvalidArgVal - ter:InvalidConfig**
  The configuration parameters are not possible to set.

ACCESS CLASS:

WRITE_SYSTEM

5.6.3 SetAuthorizationServerConfiguration

This operation modifies an existing authorization server configuration. The device shall support this command if MaxConfigurations capability is greater than zero.

REQUEST:

- **Configuration - [tas:AuthorizationServerConfiguration]**
  The modified authorization server configuration.

RESPONSE:

This is an empty message

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:NoConfig**
  The requested configuration does not exist.
- **env:Sender - ter:InvalidArgVal - ter:InvalidConfig**
  The configuration parameters are not possible to set.

ACCESS CLASS:

WRITE_SYSTEM

5.6.4 DeleteAuthorizationServerConfiguration

This operation deletes the given authorization server configuration and configuration change shall always be persistent. The device shall support this command if MaxConfigurations capability is greater than zero.

REQUEST:

- **Token - [tt:ReferenceToken]**
  Token of the configuration to delete.

RESPONSE:

This is an empty message

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:NoConfig**
  The requested configuration does not exist.

ACCESS CLASS:

WRITE_SYSTEM

5.7 JWT-based authentication

The following functions are defined to control the set of accepted *aud* claims. See also 4.4.

5.7.1 GetJWTConfiguration

This operation returns the parameters of the JWT authorization used by the device. A device shall support this command if the capability JsonWebToken in the device service capabilities is set.

REQUEST:

This is an empty message.

RESPONSE:

- **Configuration - [tas:JWTConfiguration]**
  The configuration parameters for JWT authorization used by the device.

FAULTS:

None

ACCESS CLASS:

READ_SYSTEM_SECRET

5.7.2 SetJWTConfiguration

This operation sets the parameters of the JWT authorization used by the device. A device shall support this command if the capability JsonWebToken in the device service capabilities is set.

JWT client authorization of the device is enabled when at least one key or trusted issuer URI is provided.

REQUEST:

- **Configuration - [tas:JWTConfiguration]**
  The configuration parameters for JWT authorization used by the device.

RESPONSE:

This is an empty message.

FAULTS:

- **env:Sender - ter:InvalidArgVal - ter:TooManyAudiences**
  The list of audiences is too big to be saved.
- **ter:Sender - ter:InvalidArgVal - ter:KeyID**
  No key is stored under the requested KeyID.
- **ter:Sender - ter:InvalidArgVal - ter:MaxIssuersExceeded**
  Too many trusted issuers provided.
- **env:Sender - ter:InvalidArgVal - ter:CertPathValidationPolicyID**
  No certification path validation policy is stored under the requested certification path validation policy ID.

ACCESS CLASS:

WRITE_SYSTEM_SECRET

A device shall support assignment of as many keys as its key store can hold. A device shall support at least two trusted issuers.

5.8 Capabilities

5.8.1 GetServiceCapabilities

The capabilities reflect optional functions and functionality of the different features in the security configuration service. The service capabilities consist of keystore capabilities and TLS server capabilities. The information is static and does not change during device operation.

A device shall support this command.

REQUEST:

This is an empty message.

RESPONSE:

- **Capabilities - [tas:Capabilities]**
  The capabilities for the security configuration service.

FAULTS:

None

ACCESS CLASS:

PRE_AUTH

5.8.2 Keystore Capabilities

The keystore capabilities reflect optional functions and functionality of the keystore on a device. The following capabilities are available:

Table 8: Keystore Capabilities

Capability Name	Capability Semantics
MaximumNumberOfPassphrases	Indicates the maximum number of passphrases that the device is able to store simultaneously.
MaximumNumberOfKeys	Indicates the maximum number of keys that the device is able store simultaneously.
MaximumNumberOfCertificates	Indicates the maximum number of certificates that the device is able to store simultaneously.
MaximumNumberOfCertificationPaths	Indicates the maximum number of certificate paths that the device is able to store simultaneously.
SetCertPath	Indicates support for modifying an existing certification path or certification path validation policy.
RSAKeyPairGeneration	Indicates support for on-board RSA key pair generation.
ECCKeyPairGeneration	Indicates support for on-board ECC key pair generation.
RSAKeyLengths	Indicates which RSA key lengths are supported by the device.
EllipticCurves	Indicates which elliptic curves are supported by the device.
PKCS8RSAKeyPairUpload	Indicates support for uploading an RSA key pair in a PKCS#8 data structure.
PKCS8	Indicates support for uploading supported key pair in a PKCS#8 data structure.
PKCS12CertificateWithRSAPrivateKeyUpload	Indicates support for uploading a certificate along with an RSA private key in a PKCS#12 data structure.
PKCS12	Indicates support for uploading a certificate along with a private key in a PKCS#12 data structure.
PKCS10ExternalCertificationWithRSA	Indicates support for creating PKCS#10 requests for RSA keys and uploading the certificate obtained from a CA.
PKCS10	Indicates support for creating PKCS#10 requests for asymetric key pair and

Capability Name	Capability Semantics
	uploading the certificate obtained from a CA.
SelfSignedCertificateCreationWithRSA	Indicates support for creating self-signed certificates for RSA keys.
SelfSignedCertificateCreation	Indicates support for creating self-signed certificates.
SignatureAlgorithms	Indicates which signature algorithms are supported by the device.
PasswordBasedEncryptionAlgorithms	Indicates which password-based encryption algorithms are supported by the device.
PasswordBasedMACAlgorithms	Indicates which password-based MAC algorithms are supported by the device.
X.509Versions	Indicates which X.509 versions are supported by the device.[a] X.509 versions shall be encoded as version numbers, e.g., 1, 2, 3.
MaximumNumberOfCRLs	Indicates the maximum number of CRLs that the device is able to store simultaneously.
MaximumNumberOfCertificationPathValidationPolicies	Indicates the maximum number of certification path validation policies that the device is able to store simultaneously.
EnforceTLSWebClientAuthExtKeyUsage	Indicates whether a device supports checking for the TLS WWW client auth extended key usage extension while validating certification paths.
NoPrivateKeySharing	Indicates the device requires that each certificate with private key has its own unique key.
SetCertPath	The device supports modification of certificate path and certificate validation path.

[a]If a device supports X.509v3 certificates, this fact shall also be signalled by this capability.

5.8.3 TLS Server Capabilities

The TLS server capabilities reflect optional functions and functionality of the TLS server. The information is static and does not change during device operation. The following capabilities are available:

Table 9: TLS Server Capabilities

TLSServerSupported	Indicates which TLS server versions are supported by the device. Server versions shall be encoded as version numbers, e.g., “1.0 1.1 1.2”.
MaximumNumberOfTLSCertificationPaths	Indicates the maximum number of certification paths that may be assigned to the TLS server simultaneously.

TLSClientAuthSupported	Indicates whether the device supports TLS client authentication as defined in this specification.
CnMapsToUserSupported	Indicates whether the device supports TLS client authorization using common name to local user mapping as defined in this specification
MaximumNumberOfTLSCertificationPathValidationPolicies	Indicates the maximum number of certification path validation policies that may be assigned to the TLS server simultaneously
EnabledVersionsSupported	Indicates whether the device supports enabling and disabling specific TLS versions.

5.8.4 IEEE 802.1X Capabilities

The IEEE 802.1X configuration capabilities reflect optional functions and functionality of IEEE 802.1X configuration on a device. The following additional capabilites are defined:

Table 10: Additional IEEE 802.1X Configuration Capabilities

Capability Name	Capability Semantics
MaximumNumberOfDot1XConfigurations	Indicates the maximum number of IEEE 802.1X configurations that the device is able to configure simultaneously.
Dot1XMethods	A list of authentication method outer/inner (phase1/ phase2) combinations supported by the device.

5.8.5 Media Signing Capabilities

The Media Signing capabilities reflects optional functionality that is described in the [Media Signing Specification].

Table 11: Media Signing Capabilities

Capability Name	Capability Semantics
MediaSigningSupported	Indicates whether the device supports signing of media according to the [Media Signing Specification].
UserMediaSigningKeySupported	Indicates whether the device supports signing of media using a user provisioned key.

5.8.6 Authorization Server Capabilities

The authorization server capabilities reflect optional functionality regarding configuration of external authorization servers. The following capabilities are defined:

Table 12: Authorization Server Configuration Capabilities

Capability Name	Capability Semantics
MaxConfigurations	Indicates the maximum number of authorization server configurations that the device is able to configure simultaneously.

Capability Name	Capability Semantics
ConfigurationTypesSupported	A list of authorization server configuration types that is supported by the device, see Table 6 for possible values.
ClientAuthenticationMethodsSupported	A list of authentication methods that is supported by the device, see Table 7 for possible values.

5.8.7 Capability-implied Requirements

Table 13 summarizes for each capability the minimum requirements that a device signaling this capability shall satisfy; it should not be seen as a recommendation.

Table 13: Requirements implied by Capabilities

Capability	Implied Requirements
MaximumNumberOfPassphrases	If greater than zero, the following commands shall be supported: • UploadPassphrase • GetAllPassphrases • DeletePassphrase If greater than zero, the device shall support passphrases that consist of characters from the ASCII character set and that have a length of up to 40 characters.
MaximumNumberOfKeys	If greater than zero, then the following commands shall be supported: • GetKeyStatus • GetAllKeys • DeleteKey
MaximumNumberOfCertificates	If greater than zero, then MaximumNumberOfKeys>0 shall hold.
MaximumNumberOfCertificationPaths	If greater than zero, MaximumNumberOfCertificates>=2 shall hold.
RSAKeyPairGeneration	If true, the following commands shall be supported: • CreateRSAKeyPair If true, the list of supported RSA key lengths as indicated by the RSAKeyLengths capability shall not be empty. If true, MaximumNumberOfKeys>0 shall hold.
ECCKeyPairGeneration	If true, the following commands shall be supported: • CreateECCKeyPair If true, the list of supported elliptic curves indicated by the EllipticCurves capability shall not be empty. If true, MaximumNumberOfKeys>0 shall hold.
PKCS8	If true, the following commands shall be supported: • UploadKeyPairInPKCS8

Capability	Implied Requirements
	If true, MaximumNumberOfKeys > 0 shall hold. If true, the list of supported password-based encryption algorithms as indicated by the PasswordBasedEncryptionAlgorithms capability shall contain at least the algorithm pbeWithSHAAnd3-KeyTripleDES-CBC. If true, at least one capability between EllipticCurves and RSAKeyLengths shall be specified.
PKCS8RSAKeyPairUpload	If true, the conditions of PKCS8 shall be supported: If true, the list of supported RSA key lengths as indicated by the RSAKeyLengths capability shall not be empty.
PKCS12	If true, the following commands shall be supported: • UploadCertificateWithPrivateKeyInPKCS12 • GetCertificate • GetAllCertificates • DeleteCertificate • GetCertificationPath • GetAllCertificationPaths • DeleteCertificationPath If true, MaximumNumberOfKeys >=2 shall hold. If true, MaximumNumberOfCertificates >=2 shall hold. If true, MaximumNumberOfCertificattionPaths >0 shall hold. If true, the list of supported password-based encryption algorithms as indicated by the PasswordBasedEncryptionAlgorithms capability shall contain at least the algorithm pbeWithSHAAnd3-KeyTripleDES-CBC. If true, the list of supported password-based MAC algorithms as indicated by the PasswordBasedMACAlgorithms capability shall contain at least the algorithm hmacWithSHA256. If true, the list of supported X.509 versions as indicated by the X.509Versions capability shall contain at least the value 3.
PKCS12CertificateWithRSAPrivateKeyUpload	If true, the conditions of capability PKCS12 shall be fulfilled. If true, the list of supported RSA key lengths as indicated by the RSAKeyLengths capability shall not be empty.
PKCS10	If true, the following operations shall be supported: • Creating a CSR with the CreatePKCS10CSR command. • GetCertificate • GetAllCertificates

Capability	Implied Requirements
	• DeleteCertificate • Uploading the certificate created for the CSR as well as the certificate of the created certificate's signer with the UploadCertificate command. If true, MaximumNumberOfCertificates>=2 and MaximumNumberOfCertificationPaths>0 shall hold. If true, MaximumNumberOfKeys>=2 shall hold. If true and the capability RSAKeyLengths is not empty the following condition shall be fulfilled: • The capability RSAKeyPairGeneration shall be specified . If true and the capability EllipticCurves is provided, the following conditions shall be fulfilled: • The capability ECCKeyPairGeneration shall be specified. • The list of supported signature algorithms as indicated by the SignatureAlgorithms capability shall contain at least the algorithms ECDSA-With-SHA1 and ECDSA-With-SHA256.
PKCS10ExternalCertificationWithRSA	If true, the conditions of capability PKCS10 shall be fulfilled. If true, the list of supported RSA key lengths as indicated by the RSAKeyLengths capability shall not be empty.
SelfSignedCertificateCreation	If true, the following commands shall be supported: • CreateSelfSignedCertificate • GetCertificate • GetAllCertificates • DeleteCertificate If true, MaximumNumberOfCertificates> 0 shall hold. If true, the following operations shall be supported: • Key pair generation as signaled by the RSAKeyPairGeneration or ECCKeyPairGeneration capability or key pair upload as signaled by the PKCS8 capability or key pair upload as signaled by the PKCS12CertificateWithRSAPrivateKeyUpload capability If true and the capability RSAKeyLengths is not empty the following conditions shall be fulfilled: • The capability RSAKeyPairGeneration shall be specified . • The list of supported signature algorithms as indicated by the SignatureAlgorithms capability shall contain at least the algorithms sha1-WithRSAEncryption and sha256WithRSAEncryption.

Capability	**Implied Requirements**
	If true and the capability EllipticCurves is provided, the following condition shall be fulfilled: • The capability ECCKeyPairGeneration shall be specified.
SelfSignedCertificateCreationWithRSA	If true, the conditions of the SelfSignedCertificateCreation capability shall be supported. If true, the list of supported RSA key lengths as indicated by the RSAKeyLengths capability shall not be empty.
TLSServerSupported	If not empty, PKCS10ExternalCertificationWithRSA or SelfSignedCertificateCreationWithRSA or PKCS10 or SelfSignedCertificateCreation shall be true. If not empty, the following commands shall be supported: • CreateCertificationPath • GetCertificationPath • GetAllCertificationPaths • DeleteCertificationPath • AddServerCertificateAssignment • RemoveServerCertificateAssignment • ReplaceServerCertificateAssignment • GetAssignedServerCertificates If not empty, MaximumNumberOfCertificationPaths>=2 and MaximumNumberOfTLSCertificationPaths>0 shall hold.
TLSServerSupported and PKCS10ExternalCertificationWithRSA	If TLSServerSupported is non-empty and PKCS10ExternalCertificationWithRSA is true, MaximumNumberOfCertificates>=3 shall hold.
TLSServerSupported and PKCS10	If TLSServerSupported is non-empty and PKCS10 is true, MaximumNumberOfCertificates>=3 shall hold.
MaximumNumberOfTLSCertificationPaths	If greater than zero, MaximumNumberOfCertificationPaths>0 shall hold.
EnabledVersionsSupported	If true, the following commands are supported: • SetEnabledTLSVersions • GetEnabledTLSVersions
X.509Versions	If X.509v3 is supported, the device shall support the distinguished name attribute types *country*, *organization*, *organizational unit*, *distinguished name qualifier*, *state or province name*, *common name*, and *serial number*.
MaximumNumberOfCRLs	If greater than zero, then the following commands shall be supported • UploadCRL

Capability	Implied Requirements
	• GetCRL • GetAllCRLs • DeleteCRL
MaximumNumberOfCertificationPathValidationPolicies	If greater than zero, then the following command shall be supported • CreateCertPathValidationPolicy • GetCertPathValidationPolicy • GetAllCertPathValidationPolicies • DeleteCertPathValidationPolicy If greater than zero, PKCS12CertificateWithRSAPrivateKeyUpload, PKCS10ExternalCertificationWithRSA or SelfSignedCertificateCreationWithRSA or PKCS12 or PKCS10 or SelfSignedCertificateCreation shall be true.
TLSClientAuthSupported	If true, the following commands shall be supported • SetClientAuthenticationRequired • GetClientAuthenticationRequired • AddCertPathValidationPolicyAssignment • RemoveCertPathValidationPolicyAssignment • ReplaceCertPathValidationPolicyAssignment • GetAssignedCertPathValidationPolicies If true, TLSServerSupported shall not be empty. If true, MaximumNumberOfCertificationPathValidationPolicies>=2 and MaximumNumberOfTLSCertificationPathValidationPolicies>0 shall hold. If true, the device shall support • validating certification paths containing X.509v3 certificates that are signed with signatures of type sha1-WithRSAEncryption or sha256WithRSAEncrpytion • processing X.509 CRLs that are compliant to the CRL profile defined in RFC 5280, Sect. 5 and that ○ are signed with signatures of type sha1-WithRSAEncryption or, sha256WithRSAEncryption and ○ are complete direct CRLs as defined in RFC 5280 that are signed with the same signature key as the signature key that the CA uses to sign issued certificates

Capability	Implied Requirements
MaximumNumberOfTLSCertificationPathValidationPolicies	If greater than zero, MaximumNumberOfCertificationPathValidationPolicies >0 shall hold.
MaximumNumberOfDot1XConfigurations	If greater than zero, the following commands shall be supported: • AddDot1XConfiguration • GetDot1XConfigurations • GetDot1XConfiguration • DeleteDot1XConfiguration • SetNetworkInterfaceDot1XConfiguration • GetNetworkInterfaceDot1XConfiguration • DeleteNetworkInterfaceDot1XConfiguration If greater than zero, the Dot1XMethods capability shall be present and shall contain at least the EAP-PEAP MSCHAPv2 method.
Dot1XMethods	If not empty, shall contain at least the "EAP-PEAP/MSCHAPv2" method, and MaximumNumberOfDot1XConfigurations shall be greater than zero.
MediaSigningSupported	If true, GetAssignedMediaSigningCertificates shall be supported.
UserMediaSigningKeySupported	If true, AddMediaSigningCertificateAssignment and RemoveMediaSigningCertificateAssignment shall be supported.
EllipticCurves	The list of supported elliptic curves.
AuthorizationServer.MaxConfigurations	If >0, MaximumNumberOfCertificationPathValidationPolicies>=2

5.9 Events

5.9.1 Key Status

A device that indicates support for key handling via the MaximumNumberOfKeys capability shall provide information about key status changes through key status events.

A device shall include optional item OldStatus unless NewStatus is generating.

```
Topic: tns1:Advancedsecurity/Keystore/KeyStatus
<tt:MessageDescription>
  <tt:Source>
    <tt:SimpleItemDescription Name="KeyID" Type="tas:KeyID"/>
  </tt:Source>
  <tt:Data>
    <tt:SimpleItemDescription Name="OldStatus" Type="tas:KeyStatus"/>
    <tt:SimpleItemDescription Name="NewStatus" Type="tas:KeyStatus"/>
  </tt:Data>
</tt:MessageDescription>
```

5.10 Service specific data types

The service specific data types are defined in security.wsdl.

6 Security Considerations

This section is informative.

- Faults and their types shall not disclose sensitive information to an attacker that he could not obtain otherwise.
- Secure up to date signature algorithm should be implemented by devices and clients. Devices signal the supported algorithm via their capabilities and clients should select the algorithm with highest security strength supported by both parties. As common baseline this specification requires device side support for SHA-2, particularly sha256WithRSAEncryption as specified in RFC 4055. Note that for backward compatibility of some usages this specification currently mandates device side support for sha1WithRSAEncryption as specified in RFC 3279.
- Operations with arguments that need protection against eavesdropping or manipulation shall only be executed over sufficiently protected communication channels.
- It is good practice not to use the same key for different purposes. In order to prevent the device from using the same key for different purposes unnoticedly, this specification mandates that all keys in the keystore be distinct.
- Private keys must be protected against disclosure to unauthorized parties. If a private key is uploaded in an encrypted PKCS#8 or PKCS#12 structure, the passphrase that is used to encrypt the structure must be uploaded to the device over a communication channel that is protected against eavesdropping in order to preserve the confidentiality of the private key. Moreover, the confidentiality of the uploaded private key depends on the strength of the encryption passphrase. It is therefore strongly recommended to use random passwords with sufficient length.
- In general, externally generated keys must be regarded less trustworthy than keys that are generated by the device because the probability of being disclosed to an attacker is higher for an externally generated key than for an internally generated key. A client may determine whether a key was generated by the device from the *externallyGenerated* attribute of the key.
- While new specifications should be based on PKCS#5 v2.0 or higher, adoption of this standard is still limited. Therefore, this specification intends to balance security and interoperability by mandating cryptographic algorithms based on PKCS#5 v1.5 as interoperability baseline while strongly encouraging the use of PKCS#5 v2.0 or higher. Future versions of this specification or specifications referring to this specification may mandate additional cryptographic algorithms.
- Although PKCS#8 RFC 5208 is widely used for exchanging cryptographic keys, this specification is based on the successor standard RFC 5958, particularly in order to incorporate both private key and public key in the same data structure.
- The default certification path validation policy is designed as a permissive interoperability baseline based on the certification path validation algorithm defined in RFC 5280.
- CRLs can be expected to be available from virtually any CA as a source of revocation information. The benefit of OCSP RFC 6960 as a means to obtain revocation information is increasingly under question, since a man-in-the-middle attacker blocking OCSP traffic combined with a permissive validator that silently accepts certificates for which no revocation is available limits the effective security gain of using OCSP. Therefore, this specification mandates support for CRLs as interoperability baseline and leaves other revocation information sources to future versions.
- Devices may be required to use different trust anchors for different security features. Therefore, trust in a certificate is indicated as part of a certification path validation policy rather than globally, e.g., with an attribute of the X509Certificate data type.
- RFC 5280, Sect. 6.1.4 (k) mandates that every certificate in a certification path except for the end entity certificate must be verified to be a CA certificate. For X.509 version 3 certificates, this is verified through the CA attribute in the basic constraints extension. For X.509 version 1 and 2 certificates, this information must be supplied by out-of-band mechanisms. Within the scope of this specification, the only means to obtain this information is the trust anchor information contained in the certification path validation algorithm.

- When configuring IEEE 802.1X, it is usually necessary to upload a password to the device's keystore. This should be done either on a private network (e.g., using a direct network connection between a laptop and a device) or using TLS (SSL) on the device to encrypt client / device traffic.

7 Design Rationale

This section is informative.

7.1 General Design Goals

The Security Configuration Service is designed for modularity and extensibility. Therefore, each security feature is encapsulated in a separate port type within the service. Later revisions of this specification may add port types to enhance the Security Configuration Service by additional security features.

Within a security feature, capabilities indicate support for sub-features and configuration options. Later revisions of this specification may add additional sub-features to existing features and identify them by additional capabilities.

Port types and capabilities enable devices to support well-defined subsets of this specification and to communicate this information to clients effectively.

7.2 Keystore

The keystore design assumes that passphrases are chosen by clients. Therefore, an operation for retrieving passphrases from a device is deliberately omitted. If client loses a previously uploaded passphrase, the client should create a new passphrase, upload the new passphrase to the device, and delete the old passphrase from the device.

This specification deliberately deviates from the terminology in PKCS#8 and PKCS#12 by using the term 'passphrase' instead of 'password' in order to avoid confusion with the password that is assigned to ONVIF device users and the corresponding API in the ONVIF Device Management Service.

The keystore design is based on the rationale that an RSA key pair is a special type of key pair and a key pair is a special type of key. Therefore, key-related operations in the keystore deliberately refer to the most generic possible type in this hierarchy. For example, the DeleteKey operation (see Sect. 5.2.6.2.7) refers to a key instead of a key pair or even an RSAKeyPair because it is applicable to all keys. On the other hand, the GetPrivateKeyStatus command refers to a key pair instead of a key, since this command is not meaningful for a key that is not a key pair, e.g., a symmetric key.

While this revision of the keystore specification only supports RSA key pairs as key pairs, later revisions of this specification may add other types of key pairs or symmetric keys as special types of keys.

Some interactions with the keystore, e.g., retrieving the private key for a public key that is contained in a certificate, are required device-internally, but need not be accessible to clients and may even, as in the above example, imply a security risk when made available outside the device. Such operations are therefore deliberately omitted from this specification.

The certificate-based client authentication specification intends to balance security concerns, interoperability, and implementation effort in order to facilitate adoption. Therefore, the certification path validation algorithm defined in RFC 5280 serves as interoperability baseline. The parameter values in the default certification path validation policy have been selected such that widely used implementations of the certification path validation algorithm can be used in their default configurations as much as permitted by the objective to provide an acceptable security baseline.

At the same time, more fine grained customization of the default certification path validation behavior in future versions of this specification is enabled by an extensible CertValidationPolicyParameters data type and capabilities that indicate which configuration options a device supports. As an example, checking for the TLS WWW client authentication key usage extension in client certificates is included in this specification, which can be implemented with moderate effort on the device side (e.g., with the OpenSSL option –purpose sslclient).

Customization options for other parameters of the certification path validation algorithm are deliberately left to future versions of this specification in order to limit the required initial implementation effort.

In order to facilitate future extensions of this specification, the number of certification path validation policies that may be assigned to the TLS server simultaneously is not limited, but the certification path validation behavior is unspecified if more than one policy is assigned to the TLS server at the same time. Therefore, devices implementing this specification should limit the number of simultaneously assigned policies to one.

7.3 TLS Server

This revision of the Security Configuration Service Specification allows to manage assignments of certification paths to the TLS server on a device. It is permitted that a TLS server presents different certification paths to different clients, therefore more than one certification path may be assigned simultaneously to the TLS server to use as a server certificate.

All other configuration of the TLS server on a device is outside the scope of this specification revision and may be addressed by later revisions of this document.

Annex A. JWT Content example

Here is an example of JWT components to be encoded into a JWT Token

```
{
    "typ":"JWT",
    "alg":"ES256",
}.{
    "iss":"server.path.com", // FQDN of the server that issued the JWT
    "nbf":1670482800, // not before Thursday, December 8, 2022 7:00:00 AM (GMT)
    "exp":1670572800, // expire at Friday, December 9, 2022 8:00:00 AM (GMT)
    "aud":"target.audience",
    "roles":["onvif:Operator"]
}
```

which is base64url-encoded in three parts according to RFC 7519 as

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NiJ9.eyJpc3MiOiJzZXJ2ZXIucGF0aC5jb20iLCJuYmYiOjE2NzA0ODI4MDAsImV
4cCI6MTY3MDU3MjgwMCwiYXVkIjoidGFyZ2V0LmF1ZGllbmNlIiwicm9sZXMiOlsib252aWY6T3BlcmF0b3IiXX0.SflKxw
RJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Annex B. JWT over SCTP example

A JWT, whose content example is shown in Annex A, can be used in the WS-Security header of the SOAP request. To conform to the BinarySecurityToken format, the JWT itself must be base64-encoded before being embedded in the request:

```
ZZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKRlV6STFOaUo5LmV5SnBjM01pT2lKelpYSjJaWEl1Y0dGMGFDNW
piMjBpTENKdVltWWlPakUyTnpBME9ESTRNREFzSW1WNGNDSTZNVFkzTURVM01qZ3dNQ3dpWVhWa0lqb2lkR0Z5
WjJWMExtRjFaR2xsYm1ObElpd2ljbTlzWlhNaU9sc2liMjUyYVdZNlQzQmxjbUYwYjNJaVhYMC5TZmxLeHdSSl
NNZUtLRjJRVDRmd3BNZUpmMzZQT2s2eUpwWX2FkUXNzdzVj
```

This encoded token can then be added to the SOAP request in a BinarySecurityToken element.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:enc="http://www.w3.org/2003/05/soap-encoding"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:xop="http://www.w3.org/2004/08/xop/include"
              xmlns:tds="http://www.onvif.org/ver10/device/wsdl"
              xmlns:tt="http://www.onvif.org/ver10/schema">
  <env:Header>
    <wsse:Security>
      <wsse:BinarySecurityToken ValueType="urn:ietf:params:oauth:token-type:jwt"
EncodingType=”http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-sec
urity-1.0#BinarySecurityToken”>
ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKRlV6STFOaUo5LmV5SnBjM01pT2lKelpYSjJaWEl1Y0dGMGFDNW
piMjBpTENKdVltWWlPakUyTnpBME9ESTRNREFzSW1WNGNDSTZNVFkzTURVM01qZ3dNQ3dpWVhWa0lqb2lkR0Z5
WjJWMExtRjFaR2xsYm1ObElpd2ljbTlzWlhNaU9sc2liMjUyYVdZNlQzQmxjbUYwYjNJaVhYMD0uU2ZsS3h3Uk
pTTWVLS0YyUVQ0ZndwTWVKZjM2UE9rNnlKVl9hZFFzc3c1Yw==
      </wsse:BinarySecurityToken>
    </wsse:Security>
  </env:Header>
  <env:Body>
    <tds:GetDeviceInformation/>
  <env:Body>
</env:Envelope>
```

Annex C. JWT over HTTPS example

The following exchange between client and device over HTTPS demonstrates a device supporting MD5 and SHA-256 for digest authentication as well as JWT-based client authentication.

Unauthenticated request from the client:

```
POST /onvif/device_service HTTP/1.1
Host: 10.XX.XX.XX
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299

<?xml version="1.0" encoding="utf-8"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
<s:Body xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<GetDeviceInformation xmlns="http://www.onvif.org/ver10/device/wsdl"/>
</s:Body></s:Envelope>
```

Response from the device challenging for authentication

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="Silvan_http_digest"
WWW-Authenticate: Digest algorithm=MD5, realm="Silvan_http_digest", qop="auth",
nonce="62d82aa9ca59e3a04cd1", opaque="5b6ea228"
WWW-Authenticate: Digest algorithm=SHA-256, realm="Silvan_http_digest", qop="auth",
nonce="62d82aa9ca59e3a04cd1", opaque="5b6ea228"
X-Frame-Options: SAMEORIGIN
```

Authenticated request from the client, by using JWT-based client authentication whose JWT content looks like the example is shown in Annex A

```
POST /onvif/device_service HTTP/1.1
Host: 10.XX.XX.XX
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NiJ9.eyJpc3MiOiJzZXJ2ZXIucGF0aC5
jb20iLCJuYmYiOjE2NzA0ODI4MDAsImV4cCI6MTY3MDU3MjgwMCwiYXVkIjoidGFyZ2V0LmF1ZGllbmNlIiwic
m9sZXMiOlsib252aWY6T3BlcmF0b3IiXX0.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299

<?xml version="1.0" encoding="utf-8"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
<s:Body xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<GetDeviceInformation xmlns="http://www.onvif.org/ver10/device/wsdl"/>
</s:Body></s:Envelope>
```

Response from the device, rejecting the expired token

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="Silvan_http_digest", error="invalid_token",
error_description="The access token expired"
WWW-Authenticate: Digest algorithm=MD5, realm="Silvan_http_digest", qop="auth",
nonce="62d82aa9ca59e3a04cd1", opaque="5b6ea228"
WWW-Authenticate: Digest algorithm=SHA-256, realm="Silvan_http_digest", qop="auth",
nonce="62d82aa9ca59e3a04cd1", opaque="5b6ea228"
X-Frame-Options: SAMEORIGIN
```

Annex D. JWT over RTSPS example

The following exchange between client and device over RTSPS demonstrates a device supporting MD5 and SHA-256 for digest authentication as well as JWT-based client authentication.

Unauthenticated request from the client:

```
DESCRIBE rtsp://10.XX.XX.XX/stream RTSP/1.0
CSeq: 1
User-Agent: ./onvifClient
Accept: application/sdp
```

Response from the device challenging for authentication

```
RTSP/1.0 401 Unauthorized
CSeq: 1
WWW-Authenticate: Bearer realm="Silvan_rtsp_digest"
WWW-Authenticate: Digest algorithm=MD5, realm="Silvan_rtsp_digest", qop="auth",
nonce="62d82aa9ca59e3a04cd1", opaque="5b6ea228"
WWW-Authenticate: Digest algorithm=SHA-256, realm="Silvan_rtsp_digest", qop="auth",
nonce="62d82aa9ca59e3a04cd1", opaque="5b6ea228"
```

Authenticated request from the client, by using JWT-based client authentication whose JWT content looks like the example is shown in Annex A

```
DESCRIBE rtsp://10.XX.XX.XX/stream RTSP/1.0
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NiJ9.eyJpc3MiOiJzZXJ2ZXIucGF0aC5
jb20iLCJuYmYiOjE2NzA0ODI4MDAsImV4cCI6MTY3MDU3MjgwMCwiYXVkIjoidGFyZ2V0LmF1ZGllbmNlIiwic
m9sZXMiOlsib252aWY6T3BlcmF0b3IiXX0.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
CSeq: 2
User-Agent: ./onvifClient
Accept: application/sdp
```

Annex E. Cipher Reference (Informative)

This Annex is advocating a small subset of Ciphers as part of the ONVIF standard that readers of this specification can use as an informative guide. The following small subset of ciphers covering TLS 1.2 / 1.3 are common recommendations issued as guidance by the bodies Cloudflare, IETF (TLS 1.2, TLS1.3), Mozilla, and ciphersuite.info (TLS 1.2, TLS 1.3).

While TLS 1.2 and 1.3 are both currently viable, we would suggest that TLS 1.3 is preferred. Do note that the table is not ordered by the strength of the cipher.

Table E.1: TLS 1.3 / 1.2 Cipher list

Minimum Protocol	IANA Name
TLS 1.3	TLS_AES_256_GCM_SHA384
TLS 1.3	TLS_CHACHA20_POLY1305_SHA256
TLS 1.2	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS 1.2	TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
TLS 1.2	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS 1.2	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
TLS 1.2	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

Annex F. Revision History

Rev.	Date	Editor	Changes
1.0	Aug - 2013	Dirk Stegemann	Initial version
1.0.1	Dec - 2013	Michio Hirai, , Dirk Stegemann	Change Request 1219, 1220, 1222, 1267, 1271, 1272, 1277
1.0.2	June - 2014	Dirk Stegemann, Stefan Andersson	Change Request 1268, 1276, 1349, 1350, 1351, 1352, 1376, 1377, 1378, 1379, 1380, 1381, 1382, 1390
1.1	Dec - 2014	Dirk Stegemann	Change Request 1528, 1529, 1530, 1531, 1532, 1533, 1534, 1535, 1536, 1543, 1554
1.2	Jun - 2015	Dirk Stegemann	Change Request 1552, 1555, 1565, 1580, 1583, 1590, 1615, 1616, 1617, 1618, 1619. Added certificate-based client authentication
1.3	Feb-2016	Stefan Andersson Steve Wolf	Added IEEE 802.1X configuration
	Mar-2017	Hans Busch	Change Request 1843
18.06	Jun-2018	Hiroyuki Sano	Change Request 2240, 2259
18.12	Dec-2018	Steve Wolf	Change Request 2262, 2308
19.12	Dec-2019	Davide Cristanelli	Added Authorization of TLS authenticated connections
21.12	Dec-2021	Oksana Tyushkina, Rick Boer	Fix typo in commands names. Fix inconsistency in Security NoMatchingPrivateKey
22.06	June-2022	Hans Busch	Fix fault namespace prefix and remove requirement to accept missing MAC when passphrase is present.
22.12	Dec-2022	Hans Busch	Do not require to support multiple identical pathes. Remove CRL requirement on client authentication.
23.06	June-2023	Hans Busch	Remove requirement on passphrase support.
23.12	Dec-2023	Ottavio Campana, Fredrik Svensson	Added support for ECC cryptography, JSON Web Tokens, Authorization Servers using OAuth2 and OpenID Connect.
24.06	Jun-2024	Sriram Bhetanabottla, Hans Busch	Improve readability of section on authorization server. Move JWT example to annex.
24.12	Dec-2024	Fredrik Svensson, Sriram Bhetanabottla, Hans Busch, Jose Melancon, Sujith Raman, Kieran McCartan	Add media signing capabilities and operations. Clarify JWT and OAuth sections. Add CertPathValidationPolicyId to AuthorizationServer & StorageConfiguration. Recommend strong signature selection. Update CreateEccKeyPair and Curve name references. Add annex on TLS ciphers.
25.06	Jun-2025	Tomasz Zajac, Tomasz Chmiel	Add custom claims to JWTConfiguration. Add validation policy to Dot1X configuration.