ETSI TS 104 145 V1.1.1 (2026-06)

TECHNICAL SPECIFICATION

Cyber Security (CYBER);
Quantum-Safe Cryptography (QSC);
Quantum-Safe Enterprise Transport Security (QSETS)

Reference
DTS/CYBER-QSC-0029

Keywords
quantum safe cryptography

ETSI

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

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

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

Contents

Intellectual Property Rights

Essential patents

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

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

Trademarks

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

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

Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee Cyber Security (CYBER).

Modal verbs terminology

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

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

Executive summary

Requirements - such as legal mandates and service agreements - exist for enterprise network and data centre operators and service providers, organizations, and small businesses to be able to observe and audit the content and metadata of encrypted sessions transported across their infrastructures. Enterprise Transport Security (ETS) [2] provides these capabilities for TLS 1.3 [13] when used with traditional Diffie-Hellman key exchanges. However, Diffie-Hellman based ETS is not quantum safe.

The present document specifies an enhanced Quantum-Safe Enterprise Transport Security (QSETS) profile that allows TLS visibility in enterprise networks and data centres while protecting against quantum threats. It also meets the same mandatory requirements for the Middlebox Security Profile (MSP) [1] as ETS.

Introduction

Enterprise Transport Security (ETS), specified in ETSI TS 103 523-3 [2], is an implementation variant of Transport Layer Security (TLS) protocol version 1.3 [13] that provides visibility of TLS sessions in enterprise network and data centre domains.

TLS 1.3 was originally designed to use ephemeral Diffie-Hellman as its primary key exchange mechanism. Ephemeral Diffie-Hellman prevents passive decryption of TLS 1.3 sessions at any scale. However, there are operational circumstances where passive decryption of TLS sessions by authorized entities is required, either in real-time or by storing packets and decrypting them later.

ETS relies on long-lived Diffie-Hellman server keys that are re-used across different TLS sessions. The keys can be distributed to real-time decryption middleboxes in advance or can be requested by middleboxes to decrypt sessions post-capture. This greatly reduces the number of keys to be stored and correlated with packet storage systems compared to exporting the ephemeral keys for each TLS session.

Traditional Diffie-Hellman key exchanges are known to be vulnerable to attackers with access to a cryptographically-relevant quantum computer. Consequently, support for post-quantum [i.3] and hybrid [i.5] and [i.4] Key Encapsulation Mechanisms (KEMs) is being added to TLS 1.3.

The present document specifies an enhanced Quantum-Safe Enterprise Transport Security (QSETS) profile that allows visibility of TLS sessions protected by post-quantum or hybrid KEMs.

QSETS relies on long-lived server encapsulation seeds that are used together with session-specific data from the TLS handshake to derive KEM shared secrets and ciphertexts deterministically. The seeds can be distributed to authorized decryption middleboxes allowing them to recover the KEM shared secret and recompute the traffic keys for a TLS session. This only impacts the per-session forward secrecy of the existing TLS 1.3 security architecture, thereby ensuring interoperability and operational effectiveness.

QSETS also requires the server to report visibility information in its certificate, to indicate to the client that long-lived encapsulation seeds are in use for a particular KEM, and to describe the entities that are authorized to receive the seed and decrypt the TLS session.

There are circumstances in which visibility information is not suitable and in which the client has been informed by other means that the connections can be inspected. In such circumstances, annex A can be used. Annex A, which is optional, specifies a variant of QSETS where the visibility information is not sent.

QSETS is compatible with any TLS 1.3 compliant client. Annex B, which is optional, defines the concept of a QSETS aware client whereby a TLS 1.3 client provides additional capabilities in relation to QSETS visibility.

QSETS modifies KEM encapsulation so that it is uses a long-lived encapsulation seed. This modification is not suitable in circumstances where the KEM is required to conform to a standard that specifies randomized encapsulation. Annex C briefly describes how to provide TLS session visibility in such circumstances by exporting the ephemeral TLS traffic keys.

1 Scope

The present document specifies the enhanced Quantum Safe Enterprise Transport Security (QSETS) profile to provide quantum-safe secure communication sessions between network endpoints whilst enabling network operations. This allows the use of Transport Layer Security (TLS) version 1.3 [13] with post-quantum [i.3] or hybrid [i.5], [i.4] Key Encapsulation Mechanisms (KEMs) in, for example, compliance constrained environments.

The present document describes three QSETS architectures:

- In the first architecture, both the TLS 1.3 client and the QSETS server are located inside the enterprise.
- In the second architecture, the server is a QSETS server inside the enterprise and the TLS 1.3 client is external to the enterprise. TLS 1.3 is terminated at the enterprise edge such that QSETS is used only inside the enterprise.
- In the third architecture, the TLS 1.3 server is external to the enterprise and the TLS 1.3 client is internal to the enterprise. TLS 1.3 is again terminated at the network edge such that QSETS is used only inside the enterprise.

The present document specifies a modified KEM encapsulation process in which the randomized shared secret and ciphertext are replaced with deterministic values derived from a static encapsulation seed and bound to the TLS session. Annex C describes an alternative approach to providing TLS 1.3 visibility for circumstances in which it is not possible to modify the KEM encapsulation process.

The present document specifies visibility information for indicating the QSETS profile setup. The actions of the client on receiving the visibility information are not normatively defined; however, capabilities for a QSETS aware client are defined in annex B, which is optional. The means by which the QSETS endpoints share the static encapsulation seed are specified. Annex A describes a variant of the QSETS profile for circumstances in which visibility information is not suitable and in which the client has been informed by other means that sessions can be inspected. The means by which the client is informed is out of scope.

The present document also discusses the Middlebox Security Profile (MSP) mapping for QSETS, based on the mapping for Enterprise Transport Security (ETS) from ETSI TS 103 523-3 [2]. Annex D provides a broader comparison between ETS and QSETS.

2 References

2.1 Normative references

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

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

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

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

[1] ETSI TS 103 523-1: "CYBER; Middlebox Security Protocol; Part 1: MSP Framework and Template Requirements".

[2] ETSI TS 103 523-3: "CYBER; Middlebox Security Protocol; Part 3: Enterprise Transport Security".

[3] IANA: "Transport Layer Security (TLS) Parameters".

NOTE: Registry for TLS 1.3 and later SignatureScheme values: TLS Signature Scheme registry and TLS Supported Groups registry.

[4] IETF RFC 2818: "HTTP Over TLS".

[5] IETF RFC 4073: "Protecting Multiple Contents with the Cryptographic Message Syntax (CMS)".

[6] IETF RFC 5280: "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile".

[7] IETF RFC 5869: "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)".

[8] IETF RFC 5958: "Asymmetric Key Packages".

[9] IETF RFC 6031: "Cryptographic Message Syntax (CMS) Symmetric Key Package Content Type".

[10] IETF RFC 7193: "The application/cms Media Type".

[11] IRTF RFC 7748: "Elliptic Curves for Security".

[12] IETF RFC 7906: "NSA's Cryptographic Message Syntax (CMS) Key Management Attributes".

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

[14] Recommendation X.509 (10/19) | ISO/IEC 9594-8: "Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks".

[15] FIPS 203: "Module-Lattice-Based Key-Encapsulation Mechanism Standard".

[16] NIST SP 800-56A Rev. 3: "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography".

[17] NIST SP 800-185: "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash".

2.2 Informative references

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

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

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

[i.1] IETF RFC 5083: "Cryptographic Message Syntax (CMS) Authenticated-Enveloped-Data Content Type".

[i.2] IETF RFC 5652: "Cryptographic Message Syntax (CMS)".

[i.3] IETF draft-ietf-tls-mlkem: "ML-KEM Post-Quantum Key Agreement for TLS 1.3", IETF Internet-Draft (work in progress), D. Connolly.

[i.4] IETF draft-ietf-tls-ecdhe-mlkem: "Post-quantum hybrid ECDHE-MLKEM Key Agreement for TLSv1.3", IETF Internet-Draft (work in progress), K. Kwiatkowski, P. Kampanakis, B. Westerbaan and D. Stebila.

[i.5] IETF draft-ietf-tls-hybrid-design: "Hybrid key exchange in TLS 1.3", IETF Internet-Draft (work in progress), D. Stebila, S. Fluhrer and S. Gueron.

3 Definition of terms, symbols and abbreviations

3.1 Terms

Void.

3.2 Symbols

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

$a \| \| b$	Concatenation of the byte strings a and b
$a[i : j]$	Sub-string $a_i \ldots a_{j-1}$ of the byte string $a = a_0 \ldots a_{n-1}$
c	Ciphertext
K	Shared secret
pk	Public key
sk	Private key
m	Key Encapsulation Mechanism (KEM) plaintext

3.3 Abbreviations

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

AEAD	Authenticated Encryption with Associated Data
ASN	Abstract Syntax Notation
CMS	Cryptographic Message Syntax
Decaps	Decapsulation
DER	Distinguished Encoding Rules
ECDH	Elliptic Curve Diffie-Hellman
Encaps	Encapsulation
ETS	Enterprise Transport Security
FFDH	Finite Field Diffie-Hellman
FIPS	Federal Information Processing Standards
HKDF	HMAC-based Key Derivation Function
HMAC	Hashed Message Authentication Code
HSM	Hardware Security Module
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
KEM	Key Encapsulation Mechanism
KeyGen	Key Generation
ML-KEM	Module-Lattice-based Key Encapsulation Mechanism
MSP	Middlebox Security Protocol
PFS	Perfect Forward Secrecy
QSETS	Quantum-Safe Enterprise Transport Security
TLS	Transport Layer Security

4 Quantum-Safe Enterprise Transport Security

4.1 Middlebox Security Protocol requirements mapping

ETSI TS 103 523-1 [1] defines a set of Middlebox Security Protocol (MSP) Template Requirements that underpin all MSP profile specifications. These are based around the principles of Data Protection, Transparency, Access Control and Good Citizen.

An MSP profile assigns one of the following Profile Requirement labels to each MSP Template Requirement:

- MSP-Mandatory;
- Profile-Mandatory;
- Profile-Optional;
- Profile-Not-Applicable; or
- Profile-Rejected.

For a profile specification to be included in the MSP series, a Conformance Analysis is required to demonstrate that all MSP Template Requirements labelled MSP-Mandatory, Profile-Mandatory or Profile-Optional are satisfied. Optionally, a Conformance Analysis can also provide justifications for labelling any MSP Template Requirements as Profile-Not-Applicable or Profile-Rejected.

Annex B of ETSI TS 103 523-1 [1] gives a Profile Requirement labelling for the Diffie-Hellman-based Enterprise Transport Security (ETS) profile from ETSI TS 103 523-3 [2] and performs a full Conformance Analysis for all MSP Template Requirements.

The Quantum-Safe Enterprise Transport Security (QSETS) profile specified in the present document uses the same Profile Requirement labelling as Diffie-Hellman-based ETS. The Conformance Analysis for QSETS is identical to the Conformance Analysis for Diffie-Hellman-based ETS under the same threat model and assumptions.

4.2 The Quantum-Safe Enterprise Transport Security profile

4.2.1 Key Encapsulation Mechanisms

IETF RFC 8446 [13] only specified Diffie-Hellman-based key exchanges for TLS 1.3. Traditional key exchanges such as Finite Field Diffie-Hellman (FFDH) and Elliptic Curve Diffie-Hellman (ECDH) are vulnerable to adversaries with access to a Cryptographically Relevant Quantum Computer due to Shor's algorithm.

Post-quantum key exchanges are being added to TLS 1.3 to mitigate the quantum threat [i.3], [i.4] and [i.5]. However, these are Key Encapsulation Mechanisms (KEMs) which do not have the same interface as Diffie-Hellman.

A KEM consists of three algorithms:

- **KeyGen() → (*pk*, *sk*):** Takes no input. Returns a public key *pk* and the corresponding private key *sk* as output.
- **Encaps(*pk*) → (*K*, *c*):** Takes a public key *pk* as input. Returns a shared secret *K* and a ciphertext *c* that encapsulates the shared secret as output.
- **Decaps(*sk*, *c*) → *K*:** Takes a private key *sk* and ciphertext *c* as input. Returns the shared secret *K* encapsulated by *c* as output.

4.2.2 KEM-based TLS 1.3 key exchange

For reference, a description of the KEM-based TLS 1.3 key exchange is included. Unless a pre-shared key is in use, the TLS 1.3 key exchange mechanism proceeds at the start of a new session as follows [i.5]:

1) The client generates an ephemeral KEM public *pk* and private key *sk*. The public key *pk* is transmitted to the server in a "key_share" message with a random client nonce *client_random*.
2) The server uses the client public key *pk* to generate a shared secret *K* and a ciphertext *c* that encapsulates *K*. The ciphertext *c* is transmitted to the client in a "key_share" message with a random server nonce *server_random*.
3) The client uses their private key *sk* and the ciphertext *c* from the server to recover the shared secret *K*.
4) The client and server then use the shared secret *K* along with the initial handshake messages, which include the nonces *client_random* and *server_random*, to generate a set of handshake traffic keys for encryption of the remainder of the handshake.

5) During the remainder of the handshake, the server sends its certificate encrypted using a handshake traffic key.

6) Upon completion of the handshake, the client and server then use the shared secret along with various elements of the handshake messages, which again include the nonces *client_random* and *server_random*, to generate a set of application traffic keys for the session.

7) The application traffic keys are used to encrypt further data exchanged between the client and server.

4.2.3 KEM-based QSETS key exchange

The KEM-based QSETS key exchange shall use exactly the same messages and procedures to establish a set of session keys as a KEM-based TLS 1.3 exchange, except for three differences:

- the server shall overwrite the first 8 bytes of the random server nonce *server_random* with the identifier *encaps_seed_id* for a static encapsulation seed *encaps_seed* (see clause 5.2.3);

NOTE 1: This does not conflict with the downgrade protection for TLS 1.3 which overwrites the last 8 bytes of the random server nonce to indicate a downgrade to TLS 1.2 or TLS 1.1.

- the server shall derive the shared secret *K* and ciphertext *c* at Step 2 in clause 4.2.2 from the static encapsulation seed *encaps_seed*, the random client nonce *client_random*, the random server nonce *server_random* and the client's public key *pk* (see clause 5); and

- the server's certificate at Step 5 in clause 4.2.2 shall contain visibility information to indicate to the client that QSETS is in use (see clause 4.2.5).

NOTE 2: Neither the static encapsulation seed nor the visibility information affects the operation of a TLS1.3 compliant client, so a QSETS server is therefore fully interoperable with TLS 1.3 clients.

The QSETS server shall be provisioned with a different static encapsulation seed for each KEM supported by the server. These static encapsulation seeds may be shared with middleboxes that are authorized to decrypt sessions from the server.

The static encapsulation seed shall be:

- installed directly on the server, or generated on an associated Hardware Security Module (HSM), and rotated as necessary, described in clause 4.2.6; or

- downloaded from a central key manager and updated as necessary, described in clause 4.2.7.

4.2.4 QSETS middlebox key recovery

A middlebox provisioned with the static encapsulation seed *encaps_seed* for the KEM shall recover the handshake and application traffic keys sessions as follows:

1) The middlebox obtains the client's public key *pk* and random client nonce *client_random* from the client's "key_share" message.

2) The middlebox obtains the random server nonce *server_random* from the server's "key_share" message.

3) The middlebox derives the shared secret *K* from the static encapsulation seed *encaps_seed*, the random client nonce *client_random*, the random server nonce *server_random*, and the client's public key *pk* (see clause 5).

4) The middlebox uses the shared secret *K* along with the initial handshake messages to derive the handshake traffic keys used to encrypt the remainder of the handshake.

5) The middlebox uses the shared secret *K* along with various elements of the handshake messages to derive the application traffic keys used to encrypt the session.

4.2.5 Visibility information

With the QSETS profile, the server certificate, as defined in Recommendation ITU-T X.509 [14], shall include visibility information which:

- indicates that QSETS is being used; and
- identifies, either generally or specifically, the controlling or authorizing entities or roles or domains, or any combination of these, of any middleboxes that may be allowed access to the static encapsulation seed described in clause 4.2.3.

The QSETS profile shall reuse the Visibility Information field from ETSI TS 103 523-3 [2], which is defined by the following ASN.1 type:

```
VisibilityInformation ::= SEQUENCE {
     fingerprint         OCTET STRING (SIZE(10)),
     accessDescription   UTF8String }
```

The `fingerprint` field shall be set to *fingerprint = encaps_seed_id || kem_id* where:

- *encaps_seed_id* is the 8-byte identifier for the seed (see clause 5.2.3); and
- *kem_id* is the 2-byte identifier for the KEM in the TLS Supported Groups registry in Transport Layer Security (TLS) Parameters [3].

EXAMPLE 1: The 2-byte identifier for ML-KEM-768 is `0x0201`.

The `accessDescription` field shall be a human-readable text string that identifies, either generally or specifically, the controlling or authorizing entities or roles or domains, or any combination of these, of any middleboxes that may be allowed access to the QSETS static encapsulation seed and therefore be given the ability to decrypt data protected by the corresponding QSETS key exchange.

The Visibility Information field shall be sent as an `otherName` field entry of the `subjectAltName` as defined in clause 4.2.1.6 Subject Alternative Name of IETF RFC 5280 [6]. The `type-id` shall be set to the Object Identifier 0.4.0.3523.3.1 {itu-t(0) identified-organization(4) etsi(0) msp(3523) ets(3) visibility(1)} and the contents of `value` shall be set to the DER encoding of `VisibilityInformation`.

The server may include Visibility Information for multiple QSETS static encapsulation seeds or ETS static Diffie-Hellman keys in a single certificate by including a `subjectAltName` containing `VisibilityInformation` for each seed or key.

The recommended actions of the client on receiving the visibility information can be found in annex B, which is optional.

An optional variant of the Enterprise Transport Security profile, where `VisibilityInformation` is not sent, is described in annex A. This variant may be used when visibility information is not suitable. This variant shall not be used unless the client operator has been informed by some means that the connection can be inspected. The means by which the client operator is informed is out of scope for the present document.

EXAMPLE 2: A client operator is informed through an acceptable use policy for a client on a private enterprise network.

If operation according to annex A is supported, it shall be explicitly enabled; otherwise, the server shall not establish QSETS sessions using certificates that do not include the visibility information defined in the present clause.

4.2.6 Directly installed seeds

The means by which a QSETS implementation shares the static encapsulation seed of clause 4.2.3 between a QSETS server and a middlebox is not specified, however the current clause and subsequent clauses 4.2.7 to 4.2.10 provide possible means without being exhaustive. The current clause describes perhaps the simplest means, which is that the static encapsulation seed is directly installed in both the QSETS server and permitted middleboxes.

4.2.7 Centrally managed seeds

Figure 1 shows the basic architecture for using a central key manager to deploy the QSETS static encapsulation seed to a key consumer.

EXAMPLE 1: A QSETS server is a key consumer.

EXAMPLE 2: A passive decryption middlebox is a key consumer.

The key manager generates the static encapsulation seed. The key manager can actively push seeds to the key consumer or the key consumer can request seeds. If a key consumer, such as a middlebox, is required to decrypt communications without delaying message processing to retrieve seeds from the key manager, it shall pre-fetch all anticipated static encapsulation seeds in advance. The key package is specified in clause 4.2.8. Protection of the key package is specified in clause 4.2.9 and the transfer mechanism is specified in clause 4.2.10.

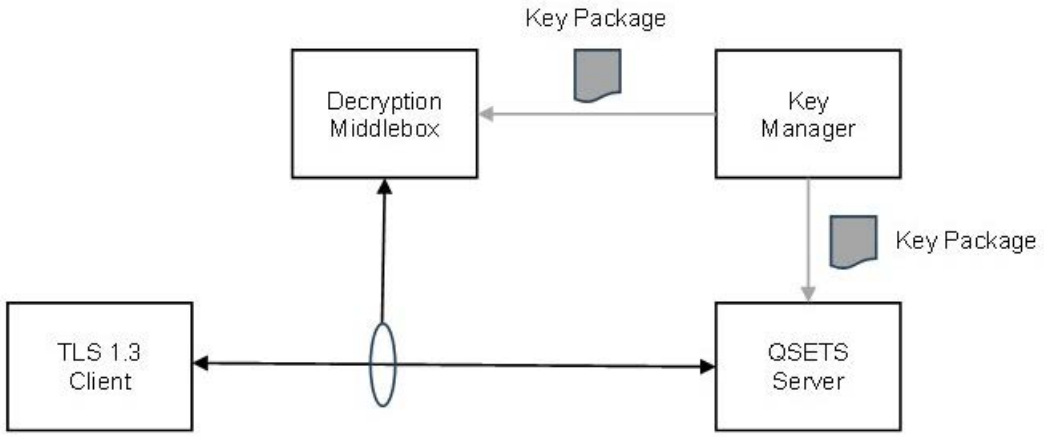

Figure 1: Architecture for centrally managed static encapsulation seeds

NOTE: In the scenario where a key manager generates the static encapsulation seed for installation on both the server and any decryption appliances, it would be natural for the key manager to generate the server certificate at the same time for each seed.

4.2.8 Key package

When a static encapsulation seed is sent from the key manager to a key consumer, it shall be packaged using the Symmetric Key Package defined in IETF RFC 6031 [9]. Each Symmetric Key Package shall contain one or more OneSymmetricKey elements to store the seeds.

The Symmetric Key Package version shall be set to v1.

The OneSymmetricKey element used to store each seed shall have the following fields:

- sKeyAttrs shall include:
  - An algorithm identifier for the KEM encoded as a UTF-8 string using the attribute defined in clause 3.2.2 of IETF RFC 6031 [9].
  - A validity period for the static encapsulation seed using the attribute defined in clause 15 of IETF RFC 7906 [12].
- sKey shall be set to the static encapsulation seed encoded as an octet string.

The algorithm identifier included in the OneSymmetricKey shall be "QSETS-" followed by the 4-character ASCII representation of the hexadecimal identifier for the KEM in the TLS Supported Groups registry in Transport Layer Security (TLS) Parameters [3].

EXAMPLE: The algorithm identifier for a static ML-KEM-768 encapsulation seed is "QSETS-0201".

When the key manager also sends the key consumer one or more static key pairs for traditional Diffie-Hellman key exchanges and/or a private signing key and certificate, these shall be packaged in an Asymmetric Key Package [8] as specified in clause 4.3.6 of ETSI TS 103 523-3 [2]. The Symmetric Key Package and Asymmetric Key Package shall be contained in a Content Collection as defined in clause 2 of IETF RFC 4073 [5].

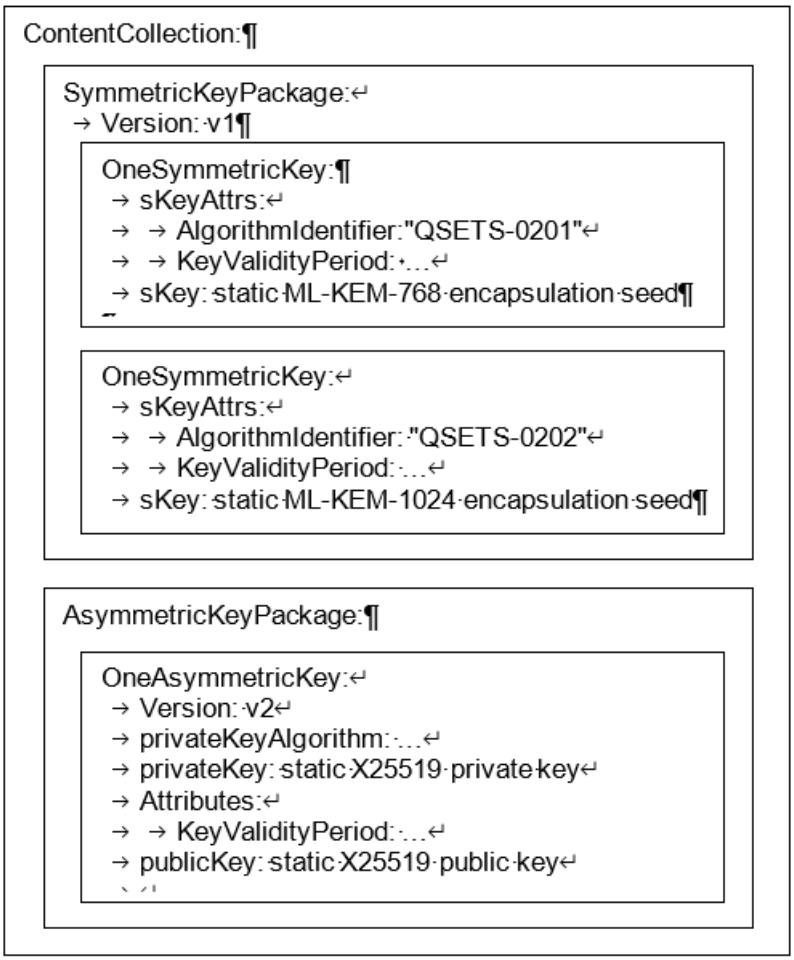

Figure 2: Key package structure

Figure 2 illustrates the structure of a key package that contains two static encapsulation seeds, one for ML-KEM-768 and one for ML-KEM-1024, and a static Diffie-Hellman key pair for X25519.

NOTE: When a key manager only sends static encapsulation seeds, it is not necessary to use a Content Collection to contain the Symmetric Key Package but the key manager can choose to do so.

4.2.9 Protecting the key package

The Cryptographic Message Syntax (CMS) defined in IETF RFC 5652 [i.2] and IETF RFC 5083 [i.1] may be used to provide authentication, integrity and/or confidentiality to the Symmetric Key Package or Content Collection transported between the key manager and the key consumer.

4.2.10 Transferring keys

4.2.10.1 Protocol overview

Enterprise Transport Security Asymmetric Key Packages shall be transferred between the key manager and key consumer using HTTPS [4]. The HTTP messages containing key packages shall comprise a suitable HTTP header followed by the Symmetric Key Package or Content Collection encoded using Distinguished Encoding Rules (DER). The HTTP Content-Type header shall be set to application/cms, as defined in IETF RFC 7193 [10].

4.2.10.2 Transfer initiated by the key manager

Key consumers may support key transfers initiated by the key manager via an HTTPS key installation service, whereby the key manager initiates an HTTPS connection to the key consumer, which acts as an HTTP server.

Key manager initiated key transfers shall follow clause 4.3.8.2 of ETSI TS 103 523-3 [2].

It is the responsibility of the key consumer to determine that the key manager is authorized to provide keys, and it is the responsibility of the key manager to determine that the key consumer is authorized to receive these keys.

4.2.10.3 Transfer initiated by the key consumer

Key managers may support key transfers initiated by the key consumer via an HTTPS key retrieval service, whereby the key consumer initiates an HTTPS connection to the key manager, which acts as an HTTP server.

Key consumer initiated key transfers shall follow clause 4.3.8.3 of ETSI TS 103 523-3 [2] with the following changes:

- For key package requests via `fingerprints=[fingerprints]`:
  - `[fingerprints]` shall be a comma-separated list where each entry in the list is the hexadecimal representation of either the fingerprint of a static encapsulation seed, as defined in clause 4.2.5 of the present document, or the fingerprint of a static Diffie-Hellman key pair, as defined in clause 4.3.3 of ETSI TS 103 523-3 [2].
- For key package requests via groups=[groups]&certs=[sigalgs]&context=contextstr:
  - `[groups]` shall be a comma-separated list where each entry in the list is the hexadecimal representation of the identifier for either a KEM or a Diffie-Hellman group from the TLS Supported Groups registry in Transport Layer Security (TLS) Parameters [3].
  - If `certs` is included, its value, `[sigalgs]`, shall be a comma-separated list where each entry is a colon-separated pair of hexadecimal representations of identifiers for algorithms from the TLS Signature Scheme registry in Transport Layer Security (TLS) Parameters [3].
- The key consumer shall only request key packages in the `application/cms` format.

EXAMPLE 1: The key consumer requests the static ML-KEM-768 encapsulation seed with fingerprint `0x0123 4567 89ab cdef 0201` and the static Diffie-Hellman key pair with fingerprint `0x0908 0706 0504 0302 0100`.

The HTTP GET request would be:

GET /.well-known/enterprise-transport-security/keys?fingerprints =0123456789abcdef0201,
09080706050403020100
Accept: application/cms

EXAMPLE 2: The key consumer requests a static Diffie-Hellman key pair for X25519 and a static encapsulation seed for ML-KEM-768.

The key consumer also requests a certificate that includes fingerprints for the static X25519 key and static ML-KEM-768 seed, where the certificate is signed with ecdsa_secp384r1_sha384 and is associated with an ecdsa_secp256r1_sha256 subject signing key.

The HTTP GET request would be:

GET /.well-known/enterprise-transport-security/keys?groups=0x001d,
0x0201&certs=0x0503:0x0403
Accept: application/cms

It is the responsibility of the key consumer to determine that the key manager is authorized to provide keys, and it is the responsibility of the key manager to determine that the key consumer is authorized to receive these keys.

4.3 QSETS Architecture

4.3.1 QSETS with enterprise clients and servers

Figure 3 illustrates the QSETS architecture when both the client and server are located in the enterprise network.

Middlebox A is authorized to inspect the traffic between the firewall and the web server. It receives a copy of the static encapsulation seed (A) used by the web server.

EXAMPLE 1: Middlebox A passively decrypts the traffic in real-time to perform intrusion detection.

The web server acts as a client in its connection to the application server. In this case, middlebox B is authorized to inspect the traffic between the two servers. It receives a copy of the static encapsulation seed (B) used by the application server.

EXAMPLE 2: Middlebox B stores copies of the encrypted packets so that they can be decrypted at a later date for compliance and auditing purposes.

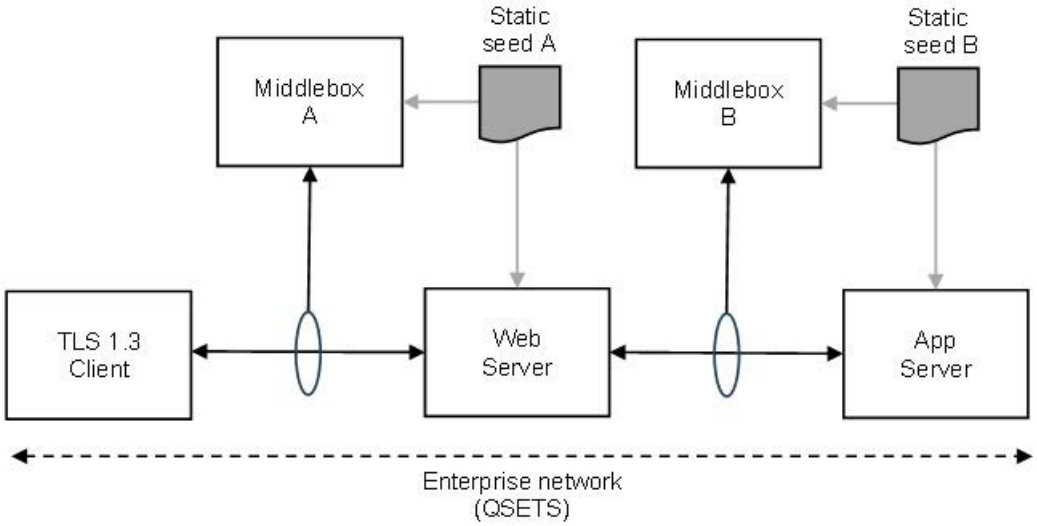

Figure 3: QSETS architecture with enterprise client and server

4.3.2 QSETS with enterprise servers

Figure 4 illustrates the QSETS architecture when the client is outside the enterprise and the server is located inside the enterprise. The firewall terminates the TLS 1.3 session from the external client and uses the QSETS profile between the firewall and the web server, with the firewall acting as the client.

Middlebox A is authorized to inspect the traffic between the firewall and the web server. It receives a copy of the static encapsulation seed (A) used by the web server.

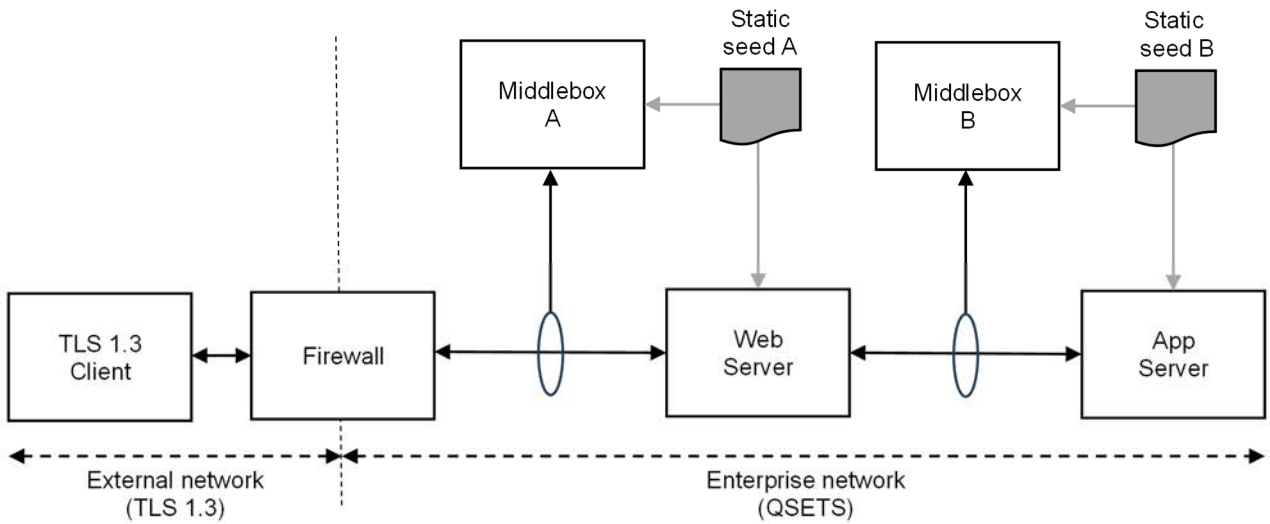

Figure 4: QSETS architecture with enterprise servers

4.3.3 QSETS with enterprise clients

Figure 5 illustrates the QSETS architecture when enterprise clients are used with a server outside the enterprise. The firewall terminates the enterprise network sessions that are using QSETS, with the firewall acting as the QSETS server. The firewall then acts as a TLS 1.3 client to the TLS 1.3 server on the external network.

Middlebox A is authorized to inspect the traffic between the client and the firewall. It receives a copy of the static encapsulation seed (A) used by the firewall.

NOTE: Although the client is participating in a QSETS connection, it is a TLS 1.3 compliant client.

EXAMPLE: Middlebox A passively decrypts the traffic in real-time to perform data loss prevention.

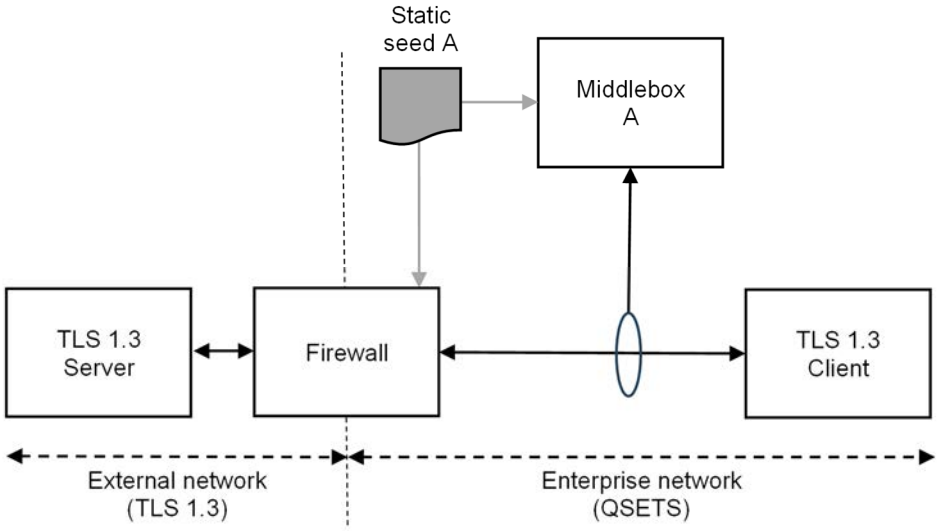

Figure 5: QSETS architecture with enterprise server

5 QSETS key encapsulation operations

5.1 Introduction

The present clause specifies the procedure for deriving the KEM ciphertext and shared secret for use in the server's encapsulation operation in clause 4.2.3, and in the middlebox's key recovery operation in clause 4.2.4.

5.2 Supporting functions

5.2.1 Encoding

To prevent ambiguities when combining multiple data fields, this protocol employs a length-prefixed encoding scheme as defined in [17].

- **encode_string(*S*):** Encodes a string *S* by prepending the length of the binary representation of the string as a 4-byte big-endian integer. That is:

  encode_string(S) = len_S || S

  where *len_S* is the length of the binary representation of *S* in bytes, represented as a 4-byte big-endian integer.

5.2.2 HKDF

The HMAC-based Key Derivation Function (HKDF) shall be as defined in IETF RFC 5869 [7]. It is parameterized by a cryptographic hash function *H* and has the interface:

- key = HKDF(ikm, salt, info, len)

where the inputs and output are as follows:

- Inputs:
  - *ikm* - a byte string representing the input keying material;
  - *salt* - a byte string representing the salt value;
  - *info* - a byte string containing context specific information; and
  - *len* - an integer specifying the length of the derived keying material in bytes.
- Output:
  - *key* - a byte string of length *len* bytes representing the derived keying material.

5.2.3 Encapsulation seed identifier

The identifier for a static encapsulation seed *encaps_seed* should be the 8-byte value:

- *encaps_seed_id* = trunc(SHA-256(*encaps_seed* || "QSETS-identifier"), 8)

where trunc(*input*, *n*) returns the first *n* bytes of the byte array *input*.

NOTE: While this method is recommended, implementations may use other mechanisms to determine encapsulation seed identifiers; for example, to avoid identifier collisions in large-scale deployments.

5.3 ML-KEM

5.3.1 ML-KEM internal encapsulation

The ML-KEM encapsulation function (Algorithm 20, FIPS 203 [15]) samples a 32-byte value *m* and calls an internal encapsulation function (Algorithm 17, FIPS 203 [15]). This internal encapsulation function has the interface:

- (*K*, *c*) = ML-KEM.Encaps_internal(*pk*, *m*)

where the inputs and outputs are as follows:

- Input:
  - *pk* - a byte string representing the public encapsulation key; and
  - *m* - a 32-byte value used to derive the shared secret and ciphertext randomness.
- Outputs:
  - *K* - a byte string representing the shared secret
  - *c* - a byte string representing the ciphertext

5.3.2 Server encapsulation

When ML-KEM is used in QSETS, the server derives the shared secret and ciphertext from a static encapsulation seed, the random client nonce, the random server nonce and the client's public key *pk* using HKDF with the hash function from the negotiated TLS 1.3 cipher suite.

The server shall check that *pk* is a byte string of the correct length for the ML-KEM parameter set before it is used (see clause 7.2 from FIPS 203 [15]). If the public key length check fails, the server shall abort the handshake with an "illegal_parameter" alert (see clause 6.2 in IETF RFC 8446 [13]).

NOTE: The server can also perform the public key modulus check specified in clause 7.2 from FIPS 203 [15], but this is not required.

The inputs, outputs and process for server encapsulation shall be as follows:

- Inputs:
  - *encaps_seed* - a 32-byte static encapsulation seed;
  - *client_random* - the 32-byte random nonce from the TLS 1.3 ClientHello message;
  - *server_random* - the 32-byte random nonce from the TLS 1.3 ServerHello message; and
  - *pk* - a byte string representing the client public key from the TLS 1.3 ClientHello key share.
- Outputs:
  - *K* - a 32-byte shared secret; and
  - *c* - a byte string representing the ciphertext.
- Process:
  1) Set ikm = encaps_seed.
  2) Set salt = client_random || server_random.
  3) Set *info* = encode(*label*) || encode(*pk*) where:
     - *label* = "QSETS-0200" for ML-KEM-512;
     - *label* = "QSETS-0201" for ML-KEM-768; and

- *label* = "QSETS-0202" for ML-KEM-1024.

4) Derive *m* = HKDF(*ikm*, *salt*, *info*, 32) using the hash function *H* specified by the cipher suite.

5) Derive (*K*, *c*) = ML-KEM.Encaps_internal(*pk*, *m*).

6) Return (*K*, *c*).

5.3.3 Middlebox key recovery

The middlebox derives the shared secret from the static encapsulation seed, the random client nonce, the random server nonce, and the client's public key *pk.*

The middlebox should check that *pk* is a byte string of the correct length for the ML-KEM parameter set before it is used (see clause 7.2 from FIPS 203 [15]). If the public key length check fails, the middlebox may still choose to use it to attempt to recover the shared secret.

NOTE: A server that follows clause 5.3.2 would be expected to abort a handshake when it receives a ClientHello containing a client public key that is not the correct length.

The inputs, outputs and process for middlebox key recovery shall be as follows:

- Inputs:
  - *encaps_seed* - a 32-byte static encapsulation seed;
  - *client_random* - the 32-byte random nonce from the TLS 1.3 ClientHello message;
  - *server_random* - the 32-byte random nonce from the TLS 1.3 ServerHello message; and
  - *pk* - a byte string representing the client public key from the TLS 1.3 ClientHello message.
- Output:
  - *K* - a 32-byte shared secret.
- Process:
  - This is identical to server encapsulation in clause 5.3.2 except that the ciphertext *c* output by ML-KEM.Encaps_internal in step 5 can be ignored and step 6 only returns the shared secret *K*.

5.4 X25519 with ML-KEM-768

5.4.1 Hybrid key encapsulation

X25519-MLKEM768 is a hybrid KEM [i.4] that combines the post-quantum ML-KEM-768 [15] with a traditional X25519 key agreement specified in IRTF RFC 7748 [11]:

- The hybrid public key *pk* is the concatenation of the public key *pk_mlkem* for ML-KEM-768 and the client public key *pk_x25519* for X25519.
- The hybrid ciphertext *c* is the concatenation of the ciphertext *c_mlkem* for ML-KEM-768 and the server public key *c_x25519* for X25519.
- The hybrid shared secret *K* is the concatenation of the shared secret *k_mlkem* for ML-KEM-768 and the shared secret *K_x25519* for X25519.

NOTE: The ordering of the ML-KEM and X25519 components for the X25519-MLKEM768 hybrid is not the same as the ordering of the ECDH and ML-KEM components for the ECDH-MLKEM hybrids in clause 5.5.

For more details on the use of hybrid KEMs in TLS 1.3, see [i.5].

5.4.2 Server encapsulation

When X25519-MLKEM768 is used in QSETS, the server derives the shared secret and ciphertext from a static encapsulation seed, the random client nonce, the random server nonce and the client's public key *pk* using HKDF with the hash function from the negotiated TLS 1.3 cipher suite.

The server shall check that *pk* is a byte string of length 1 216 bytes before it is used. If the public length check fails, the server shall abort the handshake with an "illegal_parameter" alert (see clause 6.2 in IETF RFC 8446 [13]).

NOTE 1: The server can also perform the modulus check on the ML-KEM component public key specified in clause 7.2 from FIPS 203 [15], but this is not required.

The inputs, outputs and process for server encapsulation shall be as follows:

- Inputs:
  - *encaps_seed* - a 32-byte static encapsulation seed;
  - *client_random* - the 32-byte random nonce from the TLS 1.3 ClientHello message;
  - *server_random* - the 32-byte random nonce from the TLS 1.3 ServerHello message; and
  - *pk* - a byte string representing the client public key from the TLS 1.3 ClientHello key share.
- Outputs:
  - *K* - a 32-byte shared secret; and
  - *c* - a byte string representing the ciphertext.
- Process:
  1) Set ikm = encaps_seed.
  2) Set salt = client_random || server_random.
  3) Set *info* = encode(*label*) || encode(*pk*) where *label* = "QSETS-11EC".
  4) Derive *key* = HKDF(*ikm*, *salt*, *info*, 64) using the hash function *H* specified by the cipher suite.
  5) Set *m* = *key*[0 : 32].
  6) Set *pk_mlkem* = *pk*[0 : 1184].
  7) Derive (*K_mlkem*, *c_mlkem*) = ML-KEM.Encaps_internal(*pk_mlkem*, *m*).
  8) Set *sk_x25519* = *key*[32 : 64].
  9) Set *pk_x25519* = *pk*[1184 : 1216].
  10) Derive *c_x25519* = X25519(*sk_x25519*, 9).
  11) Derive K_x25519 = X25519(sk_x25519, pk_x25519).
  12) Set K = K_mlkem || K_x25519.
  13) Set c = c_mlkem || c_x25519.
  14) Return (*K*, *c*).

NOTE 2: The X25519 function used in steps 10 and 11 is specified in clause 5 of IRTF RFC 7748 [11].

5.4.3 Middlebox key recovery

The middlebox derives the shared secret from the static encapsulation seed, the random client nonce, the random server nonce, and the client's public key *pk*.

The middlebox should check that *pk* is a byte string of length 1 216 bytes before it is used. If the public key length check fails, the middlebox may still choose to use it to attempt to recover the shared secret.

The inputs, outputs and process for middlebox key recovery shall be as follows:

- Inputs:
  - *encaps_seed* - a 32-byte static encapsulation seed;
  - *client_random* - the 32-byte random nonce from the TLS 1.3 ClientHello message;
  - *server_random* - the 32-byte random nonce from the TLS 1.3 ServerHello message; and
  - *pk* - a byte string representing the client public key from the TLS 1.3 ClientHello key share.
- Outputs:
  - *K* - a 32-byte shared secret.
- Process:
  - This is identical to server encapsulation in clause 5.4.2 except that the ML-KEM component ciphertext *c_mlkem* output by ML-KEM.Encaps_internal in step 7 can be ignored, steps 10 and 13 can be omitted completely, and step 14 only returns the shared secret *K*.

5.5 ECDH with ML-KEM

5.5.1 Hybrid encapsulation

ECDH-MLKEM is a hybrid KEM [i.4] that combines a traditional ECDH key agreement [16] with the post-quantum ML-KEM [15]:

- The hybrid public key *pk* is the concatenation of the client ECDH public key *pk_ecdh* and the ML-KEM public key *pk_mlkem*.
- The hybrid ciphertext *c* is the concatenation of the server ECDH public key *c_ecdh* and the ML-KEM ciphertext *c_mlkem*.
- The hybrid shared secret *K* is the concatenation of the ECDH shared secret *K_ecdh* and the ML-KEM shared secret *k_mlkem*.

NOTE: The ordering of the ECDH and ML-KEM components for the ECDH-MLKEM hybrids is not the same as the ordering of the ML-KEM-768 and X25519 components for the X25519-ML-KEM-768 hybrid in clause 5.4.

For more details on the use of hybrid KEMs in TLS 1.3, see [i.5].

5.5.2 Server encapsulation

When ECDH-MLKEM is used in QSETS, the server derives the shared secret and ciphertext from a static encapsulation seed, the random client nonce, the random server nonce and the client's public key *pk* using HKDF with the hash function from the negotiated TLS 1.3 cipher suite.

The server shall check that *pk* is a byte string of the correct length for the ECDH-MLKEM parameter set before it is used. If the public key length check fails, the server shall abort the handshake with an "illegal_parameter" alert (see clause 6.2 in IETF RFC 8446 [13]).

NOTE: The server can also perform the modulus check on the ML-KEM component public key specified in clause 7.2 from FIPS 203 [15] and the validity checks on the ECDH component public key specified in clause 5.6.2.2.2 of NIST SP 800-56A Rev. 3 [16], but these are not required.

The inputs, outputs and process for server encapsulation shall be as follows:

- Inputs:
  - *encaps_seed* - a 32-byte static encapsulation seed;
  - *client_random* - the 32-byte random nonce from the TLS 1.3 ClientHello message;
  - *server_random* - the 32-byte random nonce from the TLS 1.3 ServerHello message; and
  - *pk* - a byte string representing the client public key from the TLS 1.3 ClientHello key share.
- Outputs:
  - *K* - a 32-byte shared secret; and
  - *c* - a byte string representing the ciphertext.
- Process:
  1) Set ikm = encaps_seed.
  2) Set salt = client_random || server_random.
  3) Set *info* = encode(*label*) || encode(*pk*) where:
     - *label* = "QSETS-11EB" for P256-MLKEM768; and
     - *label* = "QSETS-11ED" for P384-MLKEM1024.
  4) Derive *key* = HKDF(*ikm*, *salt*, *info*, *len*) using the hash function *H* specified by the cipher suite where:
     - *len* = 72 for P256-MLKEM768; and
     - *len* = 88 for P384-MLKEM1024.
  5) Set *m* = *key*[0 : 32].
  6) Set pk_mlkem = pk[pk_ecdh_len : pk_len] where:
     - *pk_ecdh_len* = *65* and *pk_len* = 1249 for P256-MLKEM768; and
     - *pk_ecdh_len* = *97* and *pk_len* = 1665 for P384-MLKEM1024.
  7) Derive (*K_mlkem*, *c_mlkem*) = ML-KEM.Encaps_internal(*pk_mlkem*, *m*).
  8) Set *bits* = *key*[32 : *len*].
  9) Set pk_ecdh = pk[0 : pk_ecdh_len].
  10) Decode *pk_ecdh* as a point *P_ecdh* on the elliptic curve.
  11) Derive (*d_ecdh*, *Q_ecdh*) as in clause 5.6.1.2.1 of NIST SP 800-56A Rev. 3 [16] using *bits* as the random bits.
  12) Derive the ECDH shared secret *K_ecdh* from the elliptic curve point *P_ecdh* and scalar *d_ecdh* as in clause 7.4.2 of IETF RFC 8446 [13].
  13) Encode the elliptic curve point *Q_ecdh* as a byte string *c_ecdh* using the uncompressed point representation as in clause 4.2.8.2 of IETF RFC 8446 [13].
  14) Set K = K_ecdh || K_mlkem.
  15) Set c = c_ecdh || c_mlkem.
  16) Return (*K*, *c*).

5.5.3 Middlebox key recovery

The middlebox derives the shared secret from the static encapsulation seed, the random client nonce, the random server nonce, and the client's public key *pk*.

The middlebox should check that *pk* is a byte string of the correct length for the ECDH-MLKEM parameter set before it is used. If the public key length check fails, the middlebox may still choose to use it to attempt to recover the shared secret.

The inputs, outputs and process for middlebox key recovery shall be as follows:

- Inputs:
  - *encaps_seed* - a 32-byte static encapsulation seed;
  - *client_random* - the 32-byte random nonce from the TLS 1.3 ClientHello message;
  - *server_random* - the 32-byte random nonce from the TLS 1.3 ServerHello message; and
  - *pk* - a byte string representing the client public key from the TLS 1.3 ClientHello key share.
- Outputs:
  - *K* - a 32-byte shared secret.
- Process:
  - This is identical to server encapsulation in clause 5.4.2 except that the ML-KEM component ciphertext *c_mlkem* output by ML-KEM.Encaps_internal in step 7 and the elliptic curve point *Q_ecdh* derived in step 11 can be ignored, steps 13 and 15 can be omitted completely, and step 14 only returns the shared secret *K*.

6 Security operations

6.1 Security and conformance analysis

6.1.1 Ambiguity resistance

The use of encode_string for all components of the HKDF info parameter ensures that the concatenated inputs to HKDF are unambiguous. This mitigates a class of attacks that exploit parsing ambiguities in cryptographic protocols and provides robustness should the protocol evolve to include variable-length fields.

6.1.2 Cryptographic binding

The use of a structured salt and info containing personalization labels, session randomness, and the client's public key cryptographically binds the derived shared secrets to a specific protocol context, a unique TLS session, and a specific client key share.

6.1.3 Forward secrecy

This construction intentionally sacrifices Perfect Forward Secrecy (PFS) in exchange for passive middlebox access.

6.1.4 FIPS compliance

The mechanisms presented in clause 5 are non-compliant with FIPS 203 [15] which requires that the input *m* to ML-KEM internal encapsulation function should be a fresh string of random bytes generated by an approved RBG. The deterministic derivation defined in the present document is necessary to allow middlebox access.

Systems requiring strict FIPS compliance should consider alternative solutions. This can include ephemeral key export as described in annex C.

Annex A (normative): Middlebox visibility information variant

The present annex is optional. It establishes a variant of the Quantum-Safe Enterprise Transport Security profile that was defined in clause 4. The profile described in the present annex is the same as the profile in clause 4 in all respects, except that the server sends no visibility information and thus clause 4.2.5 is not carried out.

The variant of the Enterprise Transport Security profile described in the present annex may be implemented. If it is implemented, the client operator shall be informed by some means that the connection can be inspected. The means by which the client operator is informed is out of scope for the present document.

EXAMPLE 1: The client and server are wholly within a private enterprise network and the client operator has already been notified by alternative means, such as a condition of access to the network, that connections can be inspected.

EXAMPLE 2: The client operator agrees to an acceptable use policy in order to access a private enterprise network. This acceptable use policy indicates that connections can be inspected.

Annex B (normative): Requirements for a Quantum-Safe Enterprise Transport Security aware client

The present annex is optional.

A TLS 1.3 client that satisfies the requirements specified in the present annex can be designated as "Quantum-Safe Enterprise Transport Security aware":

a) The client shall provide configuration means to accept all QSETS connections or deny all QSETS connections as indicated by the certificate extension defined in clause 4.2.5.

b) If the client is a browser, it shall be possible to control the configuration in requirement (a) above by means of policy provided by a centralized policy controller.

c) If the client provides a means of viewing a server's TLS certificate, it shall correctly parse and display the contents of the certificate extension defined in clause 4.2.5.

d) If the client is a browser, the user of the browser shall be able to see the QSETS policy that is in force, even if the policy is not under their direct control.

e) The client may provide a means to only accept QSETS connections that match an allowlist.

f) The client may provide a means to only deny QSETS connections that match a blocklist.

The client may offer a user prompt to accept QSETS connections.

Annex C (informative): Middlebox ephemeral keys export

Alternatively, to maintain consistency with the requirements in FIPS 203 [15] where is necessary, and taking into account the mechanism outlined in ETSI TS 103 523-3 [2], passive decryption of TLS 1.3 sessions could be achieved by exporting the ephemeral session keys generated for each session to an authorized middlebox.

Moreover in some cases, an enterprise might not be able to use the mechanism described in clause 4 to provide middlebox access. The alternative is for the server to export the keys used in each TLS session.

Specifically, the server can export an ephemeral key package containing:

- client_random
- cipher_suite
- client_handshake_traffic_secret
- server_handshake_traffic_secret
- client_application_traffic_secret_0
- server_application_traffic_secret_0
- (optional) client_early_traffic_secret

The 32-byte ClientHello.random is sent unencrypted by the client so can be used by the middlebox to match the ephemeral key package to a specific TLS session.

The 2-byte cipher_suite indicates the Authenticated Encryption with Associated Data (AEAD) algorithm used to encrypt data and the hash function used to derive keys. Middleboxes can identify the selected cipher suite from the ServerHello message.

The client_handshake_traffic_secret and server_handshake_traffic_secret allow middleboxes access to the encrypted portions of the TLS handshake which include the encrypted extensions and certificates.

The client_application_traffic_secret_0 and server_handshake_traffic_secret_0 allow middleboxes access to the application data sent following the handshake.

NOTE: The middlebox can derive client_application_traffic_secret_N and server_application_traffic_secret_N following a KeyUpdate message as in section 7.2 of IETF RFC 8446 [13].

The ephemeral key package can optionally include the client_early_traffic_secret to allow middleboxes access to any early data sent during the handshake.

Annex D (informative): ETS and QSETS comparison

D.1 Differences between ETS and QSETS

Enterprise Transport Security (ETS), specified in ETSI TS 103 523-3 [2], and Quantum-Safe Enterprise Transport Security (QSETS), specified in the present document, share the same enterprise visibility model but differ in the underlying cryptographic mechanisms.

In particular, the following main differences can be identified:

- Key establishment:
  - ETS is based on static Diffie-Hellman key exchange (FFDH/ECDH).
  - QSETS is based on post-quantum or hybrid Key Encapsulation Mechanisms (KEMs), with deterministic derivation of encapsulation inputs as described in clause 5.
- Quantum resistance:
  - ETS is vulnerable to adversaries equipped with a Cryptographically Relevant Quantum Computer.
  - QSETS is designed to support post-quantum secure key establishment (e.g. ML-KEM).
- Enterprise long-term secret material:
  - ETS relies on long-term server Diffie-Hellman private key.
  - QSETS relies on a long-term server seed used as input keying material to deterministically derive the KEM encapsulation input per session clause 5.
- Passive decryption by authorized middleboxes:
  - ETS middleboxes use the static server Diffie-Hellman key and observed handshake values to recover session secrets.
  - QSETS middleboxes re-derive the deterministic KEM encapsulation input from the static server seed and observed handshake values and recover the same shared secret as the server.
- Visibility information fingerprint:
  - ETS uses a fingerprint derived from the static Diffie-Hellman public key.
  - QSETS uses a fingerprint that identifies the supported group in clause 4.2.5, as no static public key exists.
- Security trade-offs:
  - Both ETS and QSETS intentionally trade Perfect Forward Secrecy for enterprise visibility: compromise of the long-term enterprise secret compromises all sessions protected under that configuration.

D.2 ETS and QSETS interoperability

ETS and QSETS are designed to coexist and to support incremental deployment and migration. In particular, in terms of interoperability the following properties hold:

- TLS 1.3 wire compatibility:
  - QSETS is fully compatible with TLS 1.3. Clients that support the negotiated key exchange complete the handshake without modification.

  - Visibility information is carried in the server certificate and can be ignored by clients that do not understand it.

- Client awareness:
  - Clients are not required to be ETS- or QSETS-aware to interoperate.
  - QSETS-aware clients may parse and act on visibility information according to local or enterprise policy, as described in annex B.
- Parallel operation:
  - A deployment may support ETS and QSETS in parallel, including via server certificates carrying multiple Visibility Information entries.
  - This enables gradual migration from Diffie Hellman-based ETS to KEM-based QSETS without disrupting existing clients.
- Negotiation and policy:
  - Use of ETS or QSETS for a given connection depends on negotiated TLS 1.3 key exchange groups, server configuration, and enterprise policy.
  - Where required cryptographic support is unavailable, deployments may fall back to TLS 1.3 operation according to policy.

Annex E (informative): Bibliography

- FIPS 180-4: "Secure Hash Standard (SHS)".

History

Version	Date	Status
V1.1.1	June 2026	Publication