ETSI TS 103 523-2 V1.1.1 (2021-02)

CYBER;
Middlebox Security Protocol;
Part 2: Transport layer MSP, profile for fine grained access control

Reference
DTS/CYBER-0027-2

Keywords
cyber security

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 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is 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 Web server (https://ipr.etsi.org/).

Pursuant to the ETSI IPR Policy, no investigation, 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.

Foreword

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

The present document is part 2 of a multi-part deliverable covering Middlebox Security Protocols (MSP), defining a generic security blueprint for a family of profiles of MSP, as identified below:

Part 1: "MSP Framework and Template Requirements";

Part 2: "Transport layer MSP, profile for fine grained access control";

Part 3: "Enterprise Transport Security".

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 exist for network operators, service providers, users, enterprises, and small businesses, to be able to grant varied (fine grained) permissions and to enable visibility of middleboxes, where the middleboxes in turn gain observability of the content and metadata of encrypted sessions. Various cyber defence techniques motivate these requirements. At present, the solutions used often break security mechanisms and/or ignore the desire for explicit authorization by the endpoints. Man-In-The-Middle (MITM) proxies frequently used by enterprises prevent the use of certificate pinning and EV (Extended Validation) certificates. Where no such mechanisms exist, some encryption protocols can even be blocked altogether at the enterprise gateway, forcing users to revert to insecure protocols. As more datagram network traffic is encrypted, the problems for cyber defence will grow (IETF RFC 8404 [i.4]).

The present document is one of a series of implementation profiles to achieve these visibility and observability goals, putting the user in control of the access to their data for cyber defence purposes and protecting against unauthorized access. It sets forth a "Transport layer MSP (TLMSP), profile for fine grained access control" that meets the capability requirements found in Middlebox Security Protocol MSP Part 1 (ETSI TS 103 523-1 [i.5]).

Authorized middleboxes rarely need full read and write access to both the headers and full content of both directions of a communication session to perform their function. TLMSP provides means for classification of the communication between the endpoints into different so-called "contexts", each of which can have different read, delete, and write permissions associated with it, following the security principle of least privilege. This subdivision is for the application to determine and is under endpoint control.

TLMSP is modelled similarly to the TLS protocol (IETF RFC 5246 [1]) and composed of the TLMSP Record Protocol for the encapsulation of data from higher level protocols, and the TLMSP Handshake Protocol for the agreement of keys and the authentication of all parties with access to the communication prior to the sending of any application data. Alert and ChangeCipherSpec Protocols are also provided with similar functionalities as their TLS counterparts. These protocols: satisfy the same basic properties described in IETF RFC 5077 [2], they give visibility and control of the security of the entire communication pathway to the endpoints, and they allow the principle of least privilege to be enforced.

TLMSP is derived from mcTLS [i.1] with added features that include: additional metadata fields that allow middleboxes to perform not only read and modification operations, but also auditable insertions (of new data, originating at the middlebox) and deletions; a more flexible message format, allowing adaptation to varying network conditions; on-path middlebox discovery; improved sequence number handling; fallback to TLS; and additional security measures against recently discovered security vulnerabilities. Three normative annexes are included that contain defined cipher suites, TLS fallback mechanisms, and authentication extensions.

Introduction

There are many uses of middlebox technologies. Some examples are: providing a better user experience (content caching to reduce latency, network prefetching of content); providing user protection and cyber defence (firewalls, intrusion and malware detection, child protection); providing business protection (data loss prevention and audit).

These middlebox systems rarely require both read and write access to all communication content to function, though current security protocols necessitate an all-or-nothing approach, forcing to break the security assurances that underlying encrypted protocols are intended to provide.

EXAMPLE: Man-In-The-Middle proxies used for gateway defence do not provide any assurance of the final endpoint identity, breaking certificate pinning and violating PKI trust models. They also fail to provide assurance that the connection beyond the gateway to the endpoint is even encrypted.

On most non-enterprise networks, users generally desire control of their own data - to choose whether to grant access or not to another party. Users wishing to protect themselves from malicious software on their own systems stealing their data (or including software that harvests user data without user consent) are not currently well-positioned to insist that data is forwarded through their own cyber-defence systems or to grant access to the content. Any system that prevents this can be used as a means of stealing the user data, which is a privacy failure.

To avoid these issues, users need to layer their security architecture and not be forced to rely on endpoint defence alone, as there will be some platforms where this is not optimal, hard, or even impossible. The best defence is always expected to be a layered approach and not reliant on a single mechanism at a single location/layer. This is expected to be particularly true for those low power IoT devices that lack capability of running endpoint protection, where endpoint protection does not even exist, and where patches are slow or non-existent. Unpatched devices can be protected from vulnerabilities only by preventing malicious payloads reaching the IoT device at all; this is a requirement that can only be satisfied by network-based defence.

However, for privacy reasons, network defence ought not to require disabling of data encryption, and maintaining end-to-end encrypted data is a requirement. In the present document, a protocol profile is defined to allow endpoints in a session to authenticate, create an end-to-end encrypted session, and then authorize additional parties to access portions of the encrypted traffic. This profile provides full visibility of all additional middleboxes and their permissions to both parties prior to the sending of any application layer traffic. Additionally, no middleboxes can be added or have permissions granted by this protocol without the both endpoints agreeing to both their presence and their permission level. These requirements assure the fundamental principle that the endpoints are in control of their own data and who can have access to it.

1 Scope

The present document specifies a protocol to enable secure transparent communication sessions between network endpoints with one or more middleboxes between these endpoints, using data encryption and integrity protection, as well as authentication of the identity of the endpoints and the identity of any middlebox present. This protocol can be mapped to the abstract MSP protocol capability requirements in ETSI TS 103 523-1 [i.5].

The Middlebox Security Protocol builds on TLS 1.2 [1] and is an extensively modified version of the mcTLS protocol [i.1]. Whilst basic concepts are inherited from the mcTLS variant, the protocol specified in the present document also contains significant additional functionality and feature changes that would render it incompatible with the original version published.

The present document focuses on TLMSP usage with TCP as it is the most common usage. Usages with other transport protocols are possible but left out of scope. In the remainder of the present document, unless otherwise noted, the word TLS refers to TLS 1.2 [1].

The present document defines a set of five sub-protocols for specific purposes: Handshake (authenticating endpoints and middleboxes and negotiating cryptographic configuration among those entities); Alert (signalling errors and notifications); Application (carrying data generated by higher layers); ChangeCipherSpec (signalling the activation of the negotiated cryptographic configuration) and a Record protocol, (responsible for applying the activated security configuration to all of the other aforementioned sub-protocols).

Since TLMSP is a generic protocol, usable with a wide range of applications, issues related to mapping of application-specific security policy to explicit configurations of TLMSP is largely left out of scope. Further, out-of-band provisioning aspects relating to policies, pre-configuration of the client, details on actions in error situations are also out of scope. While some informal discussion on the security properties of TLMSP is provided, a complete (formal) security analysis of the protocol is currently left out of scope.

A reference implementation of TLMSP is being developed and can be accessed at [i.7].

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 at https://docbox.etsi.org/Reference/.

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] IETF RFC 5246: "The Transport Layer Security (TLS) Protocol Version 1.2".

[2] IETF RFC 5077: "Transport Layer Security (TLS) Session Resumption without Server-side State".

[3] IETF RFC 5116: "An Interface and Algorithms for Authenticated Encryption".

[4] IETF RFC 5746: "Transport Layer Security (TLS) Renegotiation Indication Extension".

[5] IETF RFC 7748: "Elliptic Curves for Security".

[6] IETF RFC 7919: "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)".

[7] IETF RFC 8449: "Record Size Limit Extension for TLS".

[8] IETF RFC 5288: "AES Galois Counter Mode (GCM) Cipher Suites for TLS".

[9] NIST FIPS PUB 186-4: "Digital Signature Standard (DSS)".

[10] NIST SP 800-38D: "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC".

[11] ETSI TS 133 220: "Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; Generic Authentication Architecture (GAA); Generic Bootstrapping Architecture (GBA)".

[12] IETF RFC 3986: "Uniform Resource Identifier (URI): Generic Syntax".

[13] IETF RFC 1983: "Internet Users' Glossary".

[14] IETF RFC 1123: "Requirements for Internet Hosts -- Application and Support".

[15] IETF RFC 793: "Transmission Control Protocol".

[16] IETF RFC 791: "Internet Protocol".

[17] IETF RFC 8200: "Internet Protocol, Version 6 (IPv6) Specification".

[18] IEEE 802-2014: "IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture".

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 are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] D. Naylor et al.: "Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS", SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom.

NOTE: http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p199.pdf.

[i.2] D. Naylor: "Architectural Support for Managing Privacy Tradeoffs in the Internet", Carnegie Mellon University, August 2017, PhD Thesis.

NOTE: http://reports-archive.adm.cs.cmu.edu/anon/2017/CMU-CS-17-116.pdf.

[i.3] K. Bhargavan et al.: "A Formal Treatment of Accountable Proxying over TLS", IEEE™ Symposium on Security and Privacy (SP) (2018), May 20 - 24, San Francisco, United States.

[i.4] IETF RFC 8404: "Effects of Pervasive Encryption on Operators".

[i.5] ETSI TS 103 523-1: "CYBER; Middlebox Security Protocol; Part 1: MSP Framework and Template Requirements".

[i.6] D. McGrew, D. Wing, Y. Nir, and P. Gladstone: "TLS Proxy Server Extension", draft-mcgrew-tls-proxy-server-01, IETF.

[i.7] "TLMSP reference implementation".

NOTE: Available at https://forge.etsi.org/rep/cyber.

[i.8] IETF RFC 8446: "The Transport Layer Security (TLS) Protocol Version 1.3".

[i.9] IETF RFC 8447: "IANA Registry Updates for TLS and DTLS".

3 Definition of terms, symbols and abbreviations

3.1 Terms

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

1-sided authorization: middlebox traffic observability enabled unilaterally by one endpoint such that the other endpoint is not able to reject or negotiate the traffic observability, other than by ceasing the communication

NOTE: See [i.5].

2-sided authorization: middlebox traffic observability enabled only when both endpoints agree to it

NOTE: See [i.5].

(access) privilege level: per context access rights given to an entity, amongst the four possible options:

- "none" meaning no access rights;
- "read" meaning read access rights only;
- "delete" meaning read and delete access rights only; and
- "write" meaning full access rights - the ability to read, delete, and write (including modify).

NOTE: These access privilege levels are mutually exclusive and each middlebox will have precisely one of the above privilege levels per context.

deleter: for a given context, entity having delete access privilege level with respect to that context

deleter author: for a given context, entity with at least delete access privilege that was the most recent entity to process and forward the message

NOTE 1: Deleter author is considered undefined for contexts when there does not exist any middlebox with explicitly granted delete access.

NOTE 2: TLMSP messages corresponding to context zero never has a deleter author since this context never has explicitly granted delete access.

downstream entity: when sending a TLMSP message in a certain direction, any entity located topologically, relative to the sender, in the direction of the sent message, including the enpoint in that direction

fragment: Service Data Unit (SDU), delivered from one of the higher level TLMSP protocols (Application, Alert, ChangeCipherSpec or Handshake) to the TLMSP Record protocol for protection

(message) author: entity (endpoint or middlebox) making the most recent modification to a message or part thereof

NOTE 1: In TLMSP, there can be up to three distinct authors of a given message. The term author in itself refers to the author of the (possibly encrypted) payload. The other types of authors are the "deleter author" and "writer author", see adjacent definitions. The author, deleter author, and writer author can all be the same entity, or, can all be separate, distinct entities.

NOTE 2: Modification above includes re-encrypting a message using new security parameters of the author, even if the content of the message is unchanged.

(message) originator: entity (endpoint or middlebox) where a new message was first generated and forwarded toward the destination endpoint

NOTE 1: The message originator is invariant. The message author can change as the message is being forwarded.

NOTE 2: The originator and author are only guaranteed to be the same entity at the moment when the message is transmitted by the originator.

reader: for a given context, entity having at least read access privilege level with respect to that context

(TLMSP) context: part of the fragments governed by specific, application dependent access policy

NOTE 1: Here, "part" can refer to a header, a payload, a specific implicitly or explicitly "tagged" part of the payload, or other section of the communication. A special context is defined for non-application data such as handshake and control messages.

NOTE 2: The original mcTLS specification uses the term "slice" instead of "context".

NOTE 3: A context has associated cryptographic keys, made available to those entities that are allowed certain access ("read" and possibly "delete" or "write") to the corresponding context.

(TLMSP) container: order-preserving sub-division of fragments belonging to the Application or Alert protocol, where each sub-division is associated with a specific context or part thereof

(TLMSP) entity: client, server or middlebox engaged in a TLMSP session or the negotiation of such session

(TLMSP) record: Packet Data Unit (PDU) resulting from applying TLMSP security processing directly, either to an entire fragment or to one or more containers, while preserving the inter-container ordering

NOTE: The record is delivered as SDU to lower layer (typically TCP).

upstream entity: when receiving a TLMSP message, any entity located topologically, relative to the receiver, in the direction from which the message is received, including the endpoint in that direction

writer: for a given context, entity having write access privilege level with respect to that context

writer author: for a given context, entity with write access privilege that was the most recent entity to process and forward the message

NOTE: A writer author is always defined and is considered to be the endpoint if no middlebox with write access exists for the given context.

3.2 Symbols

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

A || B	concatenation of binary strings A and B
b^n	the n-bit string consisting of the binary value b (0 or 1), repeated n times
B-TID	GBA-defined B-TID value (obtained during GBA bootstrapping)
CTXT_ID	Container Context Identifier
FLAGS	TLMSP container flag field
Ks_NAF	Network Access Function Key
LEN	Length
`ml_d`	Middlebox list, extended by dynamically discovered middleboxes
`ml_i`	Middlebox list (initial)

3.3 Abbreviations

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

3DES	Triple Data Encryption Standard
3GPP	Third Generation Partnership Project
AAD	Additional Authenticated Data
AEAD	Authenticated Encryption Additional Data
AES	Advanced Encryption Standard
AES-CBC	Advanced Encryption Standard - Cipher Blocker Chaining
AES-GCM	Advanced Encryption Standard - Galois Counter Mode
BSF	Bootstrapping Server Function
CBC	Cipher Block Chaining
CTR	Counter (mode)
DH	Diffie-Hellman
DHE_DSS	Ephemeral Diffie Hellman Digital Signature Standard

DNS	Domain Name System
EV	Extended Validation
FIPS	Federal Information Processing Standard
GBA	Generic Bootstrapping Architecture
GCM	Galois Counter Mode
GMAC	Galois Message Authentication Code
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
IEEE	Institute for Electrical and Electronic Engineers
IoT	Internet of Things
IP	Internet Protocol
IV	Initialization Vector
MAC	Message Authentication Code
MC	Middlebox key Confirmation message
mcTLS	Multi-Context TLS
MITM	Man In The Middle
MK	Middlebox Key material message
MNO	Mobile Network Operator
MSP	Middlebox Security Protocol
NAF	Network Application Function
NAF-Id	Network Application Function Identifier
NAI	Network Access Identifier
NAT	Network Adress Translation
NIST	National Institute of Standards and Technology
PDU	Packet Data Unit
PKI	Public Key Infrastructure
PRF	Pseudorandom Function
RFC	Request for Comments
RSA	Rivest-Shamir-Adleman
SDU	Service Data Unit
SHA	Secure Hash Algorithm
SP	Special Publication
TCAL	TLMSP Context Adaptation Layer
TCP	Transmission Control Protocol
TLMSP	Transport Layer Middlebox Security Protocol
TLS	Transport Layer Security
TR	Technical Report
TS	Technical Specification
USIM	Universal Subscriber Identity Module
UTF	Unicode Transformation Format

4 TLMSP specification

4.1 Introduction

The Transport Layer Middlebox Security Protocol (TLMSP) specified in the present document is derived from the published mcTLS protocol [i.1], [i.2]. The objective is to provide data privacy, data integrity and authentication controls of communication similar to that provided by TLS whilst also providing access to the content (with fine grained access control) to additional authorized and authenticated middleboxes, with visibility of these middleboxes and endpoint control over the permissions granted to middleboxes. Authorized middleboxes rarely need full read and write access to all parts of data and/or to both directions of a communication session to perform their function. TLMSP divides the communication between the endpoints into different contexts, each of which can have different permissions associated with it, following the security principle of least privilege with regards to read and write access. This division of communication is for the application to determine and under endpoint control.

EXAMPLE 1: Application-layer headers and content can be handled as two separate contexts with different associated permissions to each context, described further in annex D.

The TLMSP protocol model builds on the TLS protocol model with a similar presentation language [1]. It is composed mainly of the TLMSP Record Protocol, for the encapsulation of data from higher level TLMSP protocols, and the TLMSP Handshake Protocol, for the agreement of keys and the authentication of all parties with access to the communication prior to the sending of any application data. Alert and ChangeCipherSpec Protocols are also provided with similar functionalities as the TLS counterparts. These protocols satisfy the same basic properties described in the TLS protocol [1]; additionally allowing visibility and control of the security of the entire communication pathway to the endpoints and allowing the principle of least privilege to be enforced.

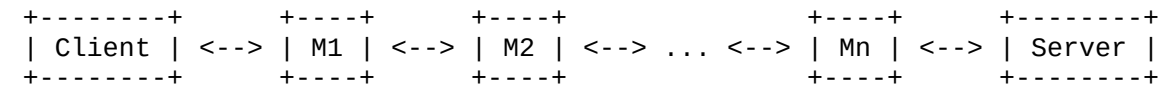


Figure 1: The TLMSP network architecture with client, server and middleboxes M1, M2, …

Unlike the original mcTLS [i.1], the protocol specified here includes:

- additional metadata fields to allow middleboxes to perform not only read and modification operations, but also auditable insertions (of new data, originating at the middlebox) and deletions;
- a more flexible message format, allowing adaptation to varying network conditions;
- on-path middlebox discovery;
- a fallback mechanism to standard TLS; and
- improved robustness of sequence number handling and additional security measures against discovered security vulnerabilities in the original mcTLS specification.

On the topic of TLS-fallback, there could be situations in which a standard TLS client initiates a TLS connection to a server supporting both TLS and TLMSP, but where this server, for whatever reason, has a policy to only allow TLMSP for this particular client. It is out of scope of the present document to specify use-cases for such policies.

EXAMPLE 2: The policy could state that additional 3rd party content filtering is necessary.

4.2 The Record protocol

4.2.1 Overview

4.2.1.1 General

Akin to TLS, the `Record` protocol is a layered protocol that fragments data from higher level protocols (e.g. `Handshake` protocol, `Application` protocol), into TLMSP records, applies the agreed data integrity checks and encryption, and then transmits the resultant records over the transport layer.

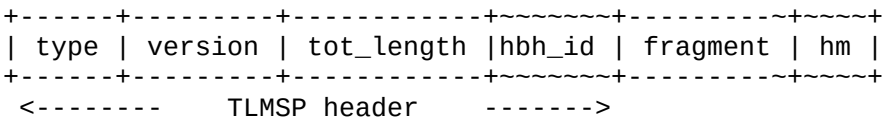
EXAMPLE: TCP can be used for transport. Each TLMSP record delivered to TCP is split across several TCP segments before transmission. Received records (after TCP re-assembly) are decrypted, integrity verified, decompressed, reassembled and then delivered to the higher protocol levels.

The current version of TLMSP does not define or make use of any (non-trivial) compression method, due to several foreseen issues as discussed in annex H. Future versions of TLMSP may specify usage of compression.

4.2.1.2 Records, containers and contexts

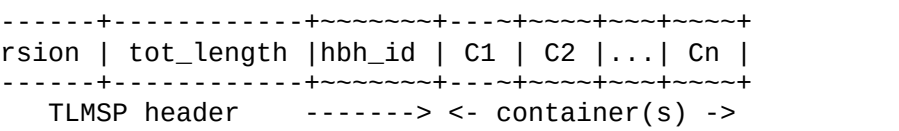
For TLMSP to allow the traffic optimizations it seeks to enable, TLMSP allows data fragments associated with multiple contexts to be "packaged" into one single TLMSP record and also allows for data associated with a single context to be split across records. Thus, a *TLMSP record* comprises protected data corresponding to one or more *TLMSP contexts*. Within a record, a (contiguous) fragment of data associated with a context is called a *TLMSP container* (or simply *container*). An explicit container format shall be used for the `Alert` and `Application` protocols, but not for the `Handshake` and `ChangeCipherSpec` protocols, both of which are associated with a default context called *context zero*.

4.2.1.3 Record and container construction and processing overview



NOTE: The field hm is the hop-by-hop MAC and is present only for Handshake records occurring after ChangeCipherSpec.

Figure 2a: TLMSP record format not using containers used by the Handshake and ChangeCipherSpec protocol



NOTE: C1, C2, … Cn represents containers, whose format is defined in Figure 3.

Figure 2b: TLMSP record format using containers (as used by Application and Alert protocols after server confirmation of TLMSP support)

The first five octets of the TLMSP header comprising `type`, `version`, and `tot_length` shall be formatted as a TLS 1.2 header as per clause 6.2.1 of IETF RFC 5246 [1].

EXAMPLE 1: `type` = 0x15 is used to signal the `Alert` protocol.

In the `ServerHello`, confirming TLMSP extension support, and in all records thereafter, there shall after the `tot_length` field follow the `hbh_id` field which is a variable length (possibly zero length) identifier for the TLMSP session, valid on a particular hop (between neighbouring entities). The `hbh_id` shall be chosen by the transmitting entity for each hop as defined in clause 4.3.5 and shall be used as defined in clauses 4.2.2.1 and 4.3.5.

The field `tot_length` shall define the total (octet) length of the record following the `tot_length` field itself, i.e. including the length indicator portion of `hbh_id` plus the indicated number of octets (which may be zero). TLMSP allows record lengths up to 2^{16} -1. However, if a TLMSP client is willing to accept lengths above the normal IETF RFC 5246 maximum of 2^{14} octets [1], this shall be signalled using the extension of IETF RFC 8449 [7]. The server and middleboxes, observing the client extension may accept or limit the length by including their corresponding maximum acceptable lengths in their extensions. The maximum length to be used shall be the minimum over the lengths occurring in all entities' extensions.

After the TLMSP record header, there shall follow the actual container(s) for those TLMSP protocols that use containers, i.e. `Alert` and `Application`. For all other TLMSP protocols, a single fragment shall follow (see clause 4.2.7.1 for details). When record protection is active, all protocols except `ChangeCipherSpec` shall then include a hop-by-hop MAC tag, denoted `hm` and computed according to clause 4.2.7.2.3, added at the end of the record in order to integrity protect the entire record (excluding `hm` itself).

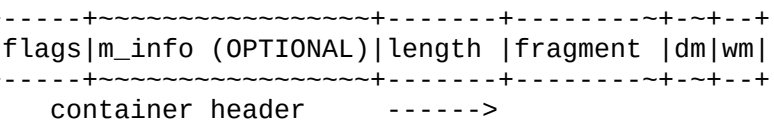


Figure 3: TLMSP container format

A container consists of a header, a (data) fragment (including a reader MAC) and one or two additional MAC values, `dm` (conditionally optional), and `wm`. Specifically, each container shall start with a container header which shall include all of the following: the associated one-octet context identifier `ctxt_id` (where `ctxt_id` = 0 is reserved), two bytes reserved for `flags`, and a 16-bit `length` field, indicating the length up to the end of the `fragment` field.

Each container shall have a maximum size of 2^{14}-1 octets, with the additional requirement that the total size of the entire TLMSP record (defined by the `tot_length` field of the TLMSP header) shall be limited to maximum default (i.e. 2^{14} octets), or, a maximum negotiated value (up to 2^{16} -1 octets).

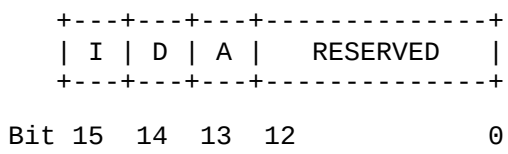


Figure 4: `flags` field of the container header

The `flags` field is used for signalling purposes, 3 bits are currently used and the remaining 13 are reserved for future versions of TLMSP. The I- and D- bits shall always be set to "0" for containers originating at the transmitting endpoint. These bits shall be set to "1" by authorized middleboxes if the container was inserted by the middlebox (only the I bit shall then be set to "1"), or, if the container is a deletion indication (both I and D bits shall be set to "1") inserted by that middlebox, as defined in clauses 4.2.3.1.3 and 4.2.3.1.4.1. The A-bit is used to additionally signal that an inserted container has auditing content, as defined in clause 4.2.3.1.3.5. Thus, the I- and D-bits always have value "10" whenever the A-bit has the value "1". Application data containers are those whose D- and A-bits are zero and that are transmitted in `Application` protocol records.

If, and only if, the I-bit is set to 1, the middlebox information field (`m_info`) shall be present, with format defined in clause 4.2.3.1.3. The container header is followed by the protected data fragment associated with the indicated context. The `m_info` field, when present, shall not be encrypted, but shall be integrity protected by including it when computing the MAC values as defined in clause 4.2.7.2.

The `fragment` field shall comprise the protected data fragment, including a reader MAC (`rm`) value as defined by clause 4.2.7.2.2 (not explicitly shown). After the `fragment`, one deleter MAC (`dm`) may be present, followed by a writer MAC (`wm`) defined by clause 4.2.7.2.2, where the writer MACs shall be present. The deleter MAC is used to signal that a deleter middlebox has inspected an application data container and decided whether to forward it or not and shall be present on such containers if, and only if, at least one middlebox has delete access to the associated context. The fact that an application data container with a valid deleter MAC is forwarded implies that the container has passed inspection by deleter middleboxes. The deleter MAC shall also be present on delete indication containers, see clause 4.2.3.1.4.1, and serves to authenticate auhorized deletions.The writer MAC field, similarly, is used to authenticate authorized changes (or absence of changes) and insertions.

EXAMPLE 2: If only read access is granted for a particular context, then only reader and writer MACs are present in containers associated with that context.

The above container format of Figure 3 shall also be used for the `Alert` protocol (`type` = 0x15), following a `ServerHello` confirming TLMSP support. Since the server's support for TLMSP can not be detected until the `ServerHello` has been received, `Alert` messages sent prior to the `ServerHello` shall be formatted as standard TLS 1.2 records, and TLMSP entities shall be implemented to be able to handle initial `Alert` messages sent by the server without the container format.

For message with `type` = 0x14 or 0x16, indicating `ChangeCipherSpec` or `Handshake`, these messages are implicitly associated with the reserved context zero as defined in clause 4.2.1.2, and containers shall not be used. That is, a single fragment without container header, carrying the protocol message content shall follow directly after the TLMSP header, as defined in clause 4.2.3.2. These messages are control messages and their semantics shall apply to all contexts associated with the TLMSP session.

EXAMPLE 3: A `ChangeCipherSpec` message is communicated as logically belonging to context zero, but the effect of `ChangeCipherSpec` will be to activate security for all contexts in use, not just context zero (see clause 4.5 for details).

4.2.2 Message unit and record processing: cryptographic state and synchronization

4.2.2.1 General

In the sequel, the term *message unit* shall denote a TLMSP record, for protocols that do not use containers (`Handshake` and `ChangeCipherSpec`), and shall denote a container, for protocols that do use containers (`Application` and `Alert`).

Each TLMSP session is associated with a state, i.e. cryptographic parameters that include a chosen PRF and cipher suite, current sequence numbers, replay protection list (e.g. window-based list of already received sequence numbers), master keys, the set of contexts and their associated key material. Further, the session is associated with non-cryptographic configuration parameters, such as the list of middleboxes and their access rights. When several TLMSP sessions are active, the correct current state and configuration can be identified at an entity (endpoint or middlebox) by the TCP socket information (IP address, port) of the local hop and, if configured, the `hbh_id` field of the record header as defined in clause 4.2.1.3.

NOTE: Since each hop of the path from sender to receiver uses a separate, locally created TCP session (defined in clause 4.3.2), the identifier for the state information is local to that hop.

Unless server support for TLMSP on a particular service port is known in advance, TLMSP should use the relevant, well-known port for TLS usage for the given application layer protocol.

A TLMSP state comprises several sub-states relating to the different entities (endpoints share certain unique parameters with different middleboxes) and certain parameters are unique to each TLMSP contexts (e.g. keys). Thus, within the state, entity identities and context identities shall be used to further retrieve relevant state information.
Clauses 4.2.2.3.3.2 and 4.2.2.2.2 describe how to determine the entity identities of the message unit originator and MAC authors, respectively, (from which relevant state information can be retrieved), and clauses 4.3.7 and 4.3.10 describe how to manage context-specific key material.

4.2.2.2 MAC overview

4.2.2.2.1 General

This clause provides an overview of the presence and processing of various MACs on message units and records. Prior to `ChangeCipherSpec`, no message unit or record (hop-by-hop) MACs shall be present. This can be viewed equivalently as either being due to no message unit or record protection yet being active, or as being due to the cipher suite `TLMSP_NULL_WITH_NULL_NULL` being active.

After security activation, message units shall include up to three MAC fields:

- a reader MAC field (using the key shared with the other readers, see clause 4.2.7.2.2);
- a deleter MAC (using the key shared with the other deleters, see clause 4.2.7.2.2);
- a writer MAC field (using the key shared with the other writers or the pairwise key shared with the downstream endpoint, see clause 4.2.7.2.2).

The reader MAC is used for the detection of changes made by unauthorized parties (which includes middleboxes that have no granted access to a particular context). Successful reader MAC verification implies that the data has not been corrupted in transit (inadvertently or maliciously). When reliable transport is used, an incorrect MAC strongly suggests adversarial attack. The deleter MAC serves to detect deletions (or tampering with deletions) by unauthorized parties (including middleboxes with less than delete privilege level). A failure of the deleter MAC verification while the reader MAC passes verification can only happen if a middlebox with read but no delete access has modified the data in transit, or, if an entity with no access has modified the deleter MAC. The writer MAC similarly serves to detect unauthorized changes by parties with less than write privilege level. A failure of the writer MAC verification while the reader MAC passes verification can only happen if a middlebox with read but no write access has modified the data in transit, or, if an entity with no access has modified the writer MAC. Generation of the reader, deleter, and writer MACs is defined in clause 4.2.7.2.2.

Records shall include a hop-by-hop MAC field (using the pairwise key shared by the sending and receiving neighbours, see clause 4.2.7.2.3). Generation of the hop-by-hop MAC is defined in clause 4.2.7.2.3.

When generating a new message unit, the reader MAC shall first be computed. The plaintext data and the reader MAC shall then be encrypted and placed together with the explicit part of the initialization vector (IV) in the `fragment` component of the message unit.

NOTE 1: If an AEAD transform is used, the MAC encryption step is typically integrated into that transform [4].

Then, if a deleter MAC is to be present according to clause 4.2.1.3, the deleter MAC shall be computed and added. The writer MAC shall then be computed and added, except for message units so indicated in Table 2.

Finally, when completing the generation of a new record carrying the message unit (for containers, after adding the last container), a hop-by-hop MAC shall be computed and added regardless of whether the record is composed of multiple containers or is a message unit itself.

When a middlebox forwards a message unit, if it contains a deleter MAC and the middlebox has delete access, the middlebox shall recompute the deleter MAC. If the middlebox has write access, it may choose to re-calculate the reader MAC and re-encrypt the message unit even if it does not perform any modifications. If the writer middlebox did modify the message, it shall recompute the reader MACs. Regardless of whether the middlebox made a modification, it shall re-compute the writer MAC.

When a middlebox forwards a record, it shall always recompute the hop-by-hop MAC regardless of whether it made modifications to the record.

When an entity computes a new MAC in preparation for transmission of a message unit, or computes a new hop-by-hop MAC in preparation for transmission of a record, it always uses its own entity identity (to indicate itself as the MAC author), and as described in clause 4.2.2.3.2, the current transmit sequence number values.

NOTE 2: Re-computation of certain MACs for otherwise unmodified message units and records as described above is necessary to prevent reader middleboxes or unauthorized 3rd parties from "undoing" changes and deletetions performed by upstream writer or deleter middleboxes, see annex E.

On the receiving side, the order of the steps described previously in this clause shall be reversed: all MAC calculation steps shall be replaced by MAC verifications and the encryption step shall be replaced by decryption. As described in clause 4.2.7.2, verification of the reader MAC requires decryption to first take place, whereas verification of the hop-by-hop, writer, and deleter MACs shall be done based on the encrypted data.

MAC verification by a middlebox shall be done for those MACs for which the middlebox possesses the corresponding key, and only for those MACs. The key used for the hop-by-hop MAC is always known by both adjacent entities, which allows for robust sequence number handling by middleboxes lacking any read or write access at all for a given context. This is described in clause 4.2.7.2.3.

If any of the performed MAC verifications fail, further processing of the received message unit or record shall be aborted. On verification failure, a corresponding `bad_reader_mac`, `bad_deleter_mac`, or `bad_writer_mac` alert shall be raised in the corresponding context. For the hop-by-hop MAC, a `bad_record_mac` alert shall be raised in context zero. An application-dependent action shall be taken in response to the alert. Defining this action is however out of scope of the present document.

EXAMPLE: In an example system, a MAC verification failure is recorded in a log and the session is terminated.

An incorrect MAC on any of the bad MAC alerts above should result in issuing a `bad_record_mac` alert in context zero and terminating the session.

4.2.2.2.2 MAC author determination

Each received record has a hop-by-hop MAC, and each received message unit has up to three additional MACs: reader, deleter, and writer MAC. The author (entity identity) of a received MAC, which in general can be distinct from the originator of the message unit and can also be distinct for different MACs, is determined as follows:

- For hop-by-hop MACs, the author is the current upstream neighbour, as defined in clause 4.2.7.3. Any indication in the received record of the author associated with the received MAC (e.g. included in an IV) is ignored.

- For writer MACs, the author is the nearest upstream entity that has write access to the message unit's context. Any indication in the received message unit of the author associated with the received MAC message unit is ignored.

- For deleter MACs, the author is the nearest upstream entity that has delete access to the message unit's context. Any indication in the received message unit of the author associated with the received MAC message unit is ignored.

- For reader MACs, the author is the entity indicated by the explicit IV associated with the received MAC.

NOTE: For an application data container in a particular context, the potential authors of each of its reader, deleter, and writer MACs are those entities with corresponding access rights to the context. It is not possible to distinguish among the potential authors of each of these MACs in an assured way as they all have access to the same keys.

4.2.2.3 Sequence numbers

4.2.2.3.1 General

Sequence numbers shall be used for security processing, for the purposes of cryptographic synchronization and replay protection. Under TLMSP, sequence numbers are not defined or maintained prior to `ChangeCipherSpec`, except for two specific `Handshake` messages as described later in this clause. Following `ChangeCipherSpec`, when a non-NULL cipher suited is selected (offering at least one of confidentiality or integrity) at each entity, each message unit is associated with unique context-independent and context-dependent sequence numbers.

NOTE 1: Even though reliable transport such as TCP is assumed, the fact that middleboxes may delete or insert message units could, without due consideration, make TLMSP vulnerable to replay, reorder, or deletion attacks.

NOTE 2: There is in general no one-to-one correspondence between TLMSP protocol messages, TLMSP records, and sequence numbers. For example, a single TLMSP Handshake protocol record can contain more than one TLMSP Handshake message, all being protected as a single message unit and thus all being associated with the same sequence number. Likewise, a TLMSP `Application` protocol record may comprise multiple containers, each to be processed and protected with a distinct sequence number.

In this section TLMSP entities are numbered starting from `e = 0` at the client, then `e = 1, 2, …, n` with `n` corresponding to the number of middleboxes and finally `e = n+1` corresponding to the server. Each entity, `i`, involved in a TLMSP session shall maintain six arrays of 64-bit sequence numbers and two individual 64-bit sequence numbers as follows:

- `seq_client_to_server_rx[j],` for `j` ranging over all entities which are topologically upstream in the client to server direction, i.e. for `0 ≤ j < i,` and `seq_server_to_client_rx[j],` for entities located upstream in the server to client direction, i.e. for `i < j ≤ n+1`. These values shall be used to record the total number (over all contexts) of valid (i.e. integrity verified) message units that have been received by, or originated at, the respective entity `j` for the direction of transmission;

- `seq_client_to_server_rx_C[j][c],` for `j` ranging over all entities which are topologically upstream in the client to server direction, i.e. for `0 ≤ j < i,` and `seq_server_to_client_rx_C[j][c],` for entities located upstream in the server to client direction, i.e. for `i < j ≤ n+1`. In both cases, `c` ranges over the set of all contexts. These values shall be used to record the total number of valid message units that have been received by, or originated at, the respective entity j in each context `c`;

- `seq_client_to_server_tx`, counting the total number of message units sent (either originated or forwarded) by entity `i` in any context in the client to server direction, and, `seq_server_to_client_tx` similarly counting the total number sent in the other direction;

- `seq_client_to_server_tx_C[c]`, counting the total number of message units sent by entity i in context c in the client to server direction, and, `seq_server_to_client_tx_C[c]` having the equivalent counting functionality, but in the other direction of transmission. In both cases, `c` ranges over the set of all contexts.

NOTE 3: It will always hold that the sum over all contexts `c` of `seq_client_to_server_rx_C[j][c]` will be equal to `seq_client_to_server_rx[j]`. Likewise, the sum over all contexts `c` of `seq_client_to_server_tx_C[c]` will equal `seq_client_to_server_tx`. Analogous relations will hold for the other direction of transmission.

The context-independent sequence numbers defined above are also referred to as global sequence numbers, as they are used to maintain a global ordering among all message units. The TLMSP session shall be terminated if any of the above defined sequence numbers is incremented to the reserved sequence number 2^{64}-1.

In the sequel, for brevity, only a specific direction of transmission from client to server is considered. Therefore, the direction aspect is omitted from notation, and only values `seq_rx[j]`, `seq_rx_C[j][c]`, `seq_tx`, and, `seq_tx_C[c]` are considered, bearing in mind that processing of messages in the other direction shall be completey analogous, but using the sequence number(s) associated with that direction. When `[]` is used as part of an array reference, it represents the set of sequence numbers referred to by all values of that array index. For example, `seq_rx_C[j][]` refers to all per-context receive sequence numbers an entity maintains for upstream entity `j` in a given direction.

When entity `i` receives a message unit in context `c`, if no tampering has occurred, `seq_rx[j]` and `seq_rx_C[j][c]`, where `j` ranges over the set of MAC authors for this message unit (different MACs can have different authors), respectively correspond to the `seq_tx` and `seq_tx_C[c]` values of each MAC author at the point when they authored the given MAC. Maintenance of these sequence numbers is essential for processing the MAC(s) of received message units.

After a message unit has been received and processed, only the applicable context-independent receive sequence numbers and the applicable receive sequence numbers associated with the specific context, `c`, of the message unit shall be updated. Likewise, after a message unit has been transmitted, only the context-independent transmit sequence number and the transmit sequence number associated with the specific context, `c`, of the message unit shall be updated.

EXAMPLE: In a TLMSP session with three contexts, `c1`, `c2`, and `c3`, when computing any of the writer or deleter MACs for an application data container associated with context `c2` that is to be transmitted, all three values `seq_tx_C[c1]`, `seq_tx_C[c2]`, and `seq_tx_C[c3]` are used as inputs to the MAC calculations (by contcatenating them, as explained below). After the message has been passed on downstream, however, only the values `seq_tx` and `seq_tx_C[c2]` are increased by one. For the reader MAC of this container, only the global value `seq_tx` is used and increased by one after the container is processed. The hop-by-hop MAC of the entire record in which this container is transmitted also uses only the global value `seq_tx`, and uses the same value as used when processing the reader MAC of the first container to be included in the record. More precise details are given below. This seemingly complex sequence number handling is needed to protect against attacks which would otherwise be possible on protocols which selectively allow insert and delete operations in multiple contexts by multiple entities. Essentially, it is necessary to use both context-specific sequence numbers (via the individual `seq_C[c]` values), as well as a session-unique identifier for each message unit (here, formed by the set of all `seq_C[c]` values). Annex E and in particular F.3 gives detailed rationale for this handling.

Usage of sequence numbers generally begins after the `ChangeCipherSpec` message. There are however two `Handshake` messages occurring before `ChangeCipherSpec`, `TLMSPKeyMaterial` and `TLMSPKeyConf`, which require sequence numbers for the security processing of their contents. These messages contain a reader MAC internally, and the reserved sequence number 2^{64}-1 shall be used in the generation and verification of that reader MAC.

NOTE 4: As all these messages contain a verification payload, computed with pairwise disitinct keys shared only between pairs of entities, it does not matter from a security point of view that the record layer processes them with the same sequence number.

When an entity transmits the `ChangeCipherSpec` message, the pending write state in the associated direction of communication shall become the active state, setting `seq_tx` to zero and `seq_tx_C[c]` to zero for all contexts `c`. When an entity receives the `ChangeCipherSpec` message, the pending read state in the associated direction of communication shall become the active state, setting `seq_rx[j]` and `seq_rx_C[j][c]` to zero for all upstream entities `j` and contexts `c`.

NOTE 5: This is a difference to TLS 1.2, [1], which uses sequence numbers also for unprotected messages, before `ChangeCipherSpec`. Messages occurring before `ChangeCipherSpec` are still protected against modification and reordering by their inclusion in the `Finished` and `MboxFinished` verification hashes. Further, since TLMSP does not support renegotiation, `Handshake` messages occuring before `ChangeCipherSpec` cannot be protected in any other way. This approach greatly simplifies message insertions/deletions by middleboxes that may occur during intial stages of the TLMSP handshake.

At this point, regular sequence number maintanenance is performed for all message units sent or received, and all records include a hop-by-hop MAC. The author of each MAC, depending on its type, includes either the context-independent transmit sequence number or a context-based sequence number formed by concatenating the transmit sequence numbers for all contexts as follows:

```
s = seq_tx_C[0] || seq_tx_C[1] || … || seq_tx_C[n_ctxt-1]
```

where `n_ctxt` is the total number of contexts in use in the TLMSP session.

NOTE 7: Since each `seq[j]` is 64 bits/8 octets, `s` is an 8*`n_ctxt` octet value.

Further details of handling of sequence number values are found in clauses 4.2.2.3.2 and 4.2.2.3.3, below.

4.2.2.3.2 Outgoing message units and records

After `ChangeCipherSpec` is sent, when an entity `j` prepares a message unit for transmission in context `c`, it uses the current values of `seq_tx` and `seq_tx[c]`, as required, for the security processing defined in clauses 4.2.3 and 4.2.7. Immediately after completing processing of the message unit (before processing any further message unit for transmission), the entity shall update its own sequence numbers as follows: `seq_tx = seq_tx + 1` and `seq_tx_C[c] = seq_tx_C[c] + 1`. If the message unit is a deletion indication container, `seq_tx` and `seq_tx[c]` shall be further updated as described in clause 4.2.3.1.4.2.

An entity preparing a message unit for transmission that does not require any security processing, for example an entity forwarding a container in a context to which is has no access rights, shall still update the transmission sequence numbers as described above.

When generating the hop-by-hop MAC (see clause 4.2.7.2.3) for a record that is composed of one or more containers, the sequence number used is the value of `seq_tx` when the first container in the record was being prepared. That is, when creating a new record for transmission that will consist of one or more containers, prior to processing the first container for that record, entity `j` temporarily records the current value of `seq_tx` and then later uses that value when computing the final MAC, the record's hop-by-hop MAC.

EXAMPLE: Consider a TLMSP session with three contexts, `c1`, `c2`, and `c3`. Entity `j` prepares a single record with three containers for transmission (corresponding to three message units for contexts `c1`, `c2`, and `c3`, in that order). Each container will be processed with particular values of seq_tx and the set of per-context sequence numbers `seq_C[] = {seq_tx_C[c1], seq_tx_C[c2], seq_tx_C[c3]}`. Given `seq_tx = s` prior to the processing of the first container, the first container will be processed with `seq_tx = s`, the second with `seq_tx = s + 1`, and the third with `seq_tx = s + 2`, with `seq_tx` having the value `seq_tx = s + 3` following the processing of the final container. Given `seq_C[] = {n1, n2, n3}` prior to the processing of the first container, the first container will be processed with `seq_C[] = {n1, n2, n3}`, the second with `seq_C[] = {n1 + 1, n2, n3}`, and the third container with `seq_C[] = {n1 + 1, n2 + 1, n3}`, with `seq_C` having the value `seq_C[] = {n1 + 1, n2 + 1, n3 + 1}` following the processing of the final container. The value `seq_tx` = `s` is used to compute the hop-by-hop MAC of the record. If there would have been addititional contexts in the session, their corresponding sequence numbers would have been included in each `seq_C[]` value, but would remain constant, throughout the processing.

4.2.2.3.3 Incoming message units and records

4.2.2.3.3.1 General

Each received record has a hop-by-hop MAC which shall be verified, and each received message unit has up to three MACs that may need to be verified, depending on access rights: reader, writer, and deleter. Verification of these MACs depends on using correct sequence numbers, which requires first determining the author of each MAC according to clause 4.2.2.2.2. For a given MAC that is to be verified, once the identity, `e_id`, of the MAC author is determined, the current value of `seq_rx[e_id]` or values of `seq_rx_C[e_id][]` are then used as required to perform the verification.

After all required MAC verifications are successfully performed for a message unit, the receive sequence number state is updated. This requires first determining the message unit originator according to clause 4.2.2.3.3.2. Once the message unit originator, `i`, is determined, the receive sequence number state of receiving entity `k` shall be updated as follows:

- `seq_rx[j] = seq_rx[j] + 1` and `seq_rx_C[j][c] = seq_rx_C[j][c] + 1`, for `j = i, i+1, …, k-1`, and where c is the context with which the message unit is associated;
- If the message unit is a deletion indication, `seq_rx[]` and `seq_rx_C[]` shall be further updated as described in clause 4.2.3.1.4.2.

NOTE: An entity that receives a message unit in a context to which is has no access rights still updates the receive sequence numbers as described above. In this case, only the hop-by-hop MAC of the associated record is possible to be verified, but this is sufficient to ensure correct modification of the receive sequence number state.

4.2.2.3.3.2 Message unit originator determination

For all containers, a receiving entity determines the originating entity by examining the container header. If the header contains the `m_info` field (see clause 4.2.3.1.3), then the container originator is indicated by the `e_id` subfield. Otherwise, the originator is the upstream endpoint.

Handshake records are the only record message unit type that require originator determination. In general, the originator of a handshake record can be determined by examining the `Handshake` message(s) within. However, there is no need to determine the originator of handshake records that are received prior to `ChangeCipherSpec`. For handshake records that may be received after `ChangeCipherSpec`:

- If the record contains at least one `Finished` or `MboxFinished` message, the originator is the upstream endpoint.
- If the record consists of an `MboxLeaveNotify` message (such messages cannot be sent any other way), the originator is the `mbox_entity_id` present in the message.
- If the record contains an `MboxLeaveAck` message, the originator is the upstream endpoint.

4.2.3 Processing of specific message unit types

4.2.3.1 Container message units

4.2.3.1.1 Container usage

The following applies to the `Application` and `Alert` protocols.

Containers may be re-distributed between records of the same content type. A single container shall never be split across more than one record. However, for traffic flow optimization purposes:

1) Middleboxes (both readers and writers) may split a single received TLMSP record comprising `C > 1` containers into `R (1 < R ≤ C)` distinct records before forwarding.
2) Middleboxes may combine TLMSP containers from `R > 1` separate TLMSP records into a single record.

In both cases, the original order between containers shall always be strictly preserved and the middlebox shall construct the TLMSP record header, specifically the `tot_length` field, to correctly reflect the total length.

This splitting and combining applies also to the sending endpoint: the sender may buffer fragments, corresponding to several containers, received from the application layer and place those containers in one or more records before submitting them to the transport layer.

The present document specifies the production and deletions of containers in clauses 4.2.3.1.3 to 4.2.3.1.6.

Whenever there is a new container originated by an entity or a modified container generated by a middlebox (change/re-write, insert or delete):

- The resulting container shall be processed with a new IV that contains the author `e_id` and is otherwise compliant with the IV format of the used cipher suite, see annex A.
- For `Application` protocol containers other than audit containers, the resulting container shall include two or three MAC fields:
    1) a mandatory reader MAC field (using the key shared with the other readers, see clause 4.2.7.2.2);
    2) a deleter MAC (using the key shared with the other deleters, see clause 4.2.7.2.2) on all containers of contexts with granted delete access; and
    3) a mandatory writer MAC field (using the key shared with the other writers, see clause 4.2.7.2.2 deleter, the deleter key).
- For audit and alert containers, the resulting container shall include exactly two MAC fields:
    1) a reader MAC field (using the key shared with the downstream endpoint, see clause 4.2.7.2.2); and
    2) a writer MAC field (using the key shared with the downstream endpoint, see clause 4.2.7.2.2).

Application data containers shall not be transmitted in context zero.

NOTE: Delete indications are considered part of `Application` protocol and processing is therefore covered by the second bullet, except that they do not have a writer MAC.

4.2.3.1.2 Modifications

In certain cases, a middlebox can modify the contents of a container that it is forwarding. The only containers that support modification of their contents are application data containers. Only writer middleboxes with access to the associated context may modify the contents of such a container. When doing so, the writer middlebox shall leave the A, I, and D flag bits unchanged. If the `m_info` field is present, the `e_id` shall be left unmodified, see clause 4.2.3.1.3. Following the modification, the middlebox shall update the container MACs as described in clause 4.2.2.2.1.

Modifying content at an endpoint is an application layer issue and is out of scope of the present document.

4.2.3.1.3 Insertions generally

Insertions are the introduction of new containers into the session by middleboxes. Containers originating at endpoints are not considered to be insertions. When inserting a container, the middlebox shall set the I-bit of the `flags` container header-field to 1.

A middlebox that inserts a container shall always add an `m_info` field to the container header, which provides information required by each downstream entity to maintain its sequence number state. The `m_info` field shall have the structure shown in Figure 5.

```
+-----+~~~~+~~~~~~~+
|e_id |src |del_c  |
+-----+~~~~+~~~~~~~+
```

Figure 5: `m_info` field

Every `m_info` shall contain the one-octet subfield `e_id`, which is the entity identity of the middlebox that performed the insertion. As is explained in clause 4.2.3.1.4.1, deletion indication containers also include the `src` and `del_c` subfields, which are one and two octets, respectively.

4.2.3.1.4 Deletion indication containers

4.2.3.1.4.1 General

Deletion indication containers are `Application` protocol message units that signal the deletion of a continguous set of application data containers, associated with a particular context, to all downstream entities. Application data containers may be deleted by middleboxes having delete access to the corresponding contexts. No other type of container may be deleted. If one or more contiguous containers originated by the same entity in a given context are deleted, they shall be replaced by at least one deletion indication container in that same context.

The transmission of a deletion indication may be postponed, but shall occur at the latest immediately before another container is forwarded to the destination endpoint. When inserting a deletion indication container, a middlebox shall:

- set the `ctxt_id` field of the container header to the context of the deleted containers;
- set the D-bit of the `flags` container header field to 1; and
- include the `src` and `del_c` subfields in the `m_info` field, setting `src` to the entity identity of the originator of the deleted container(s) and `del_c` to the number of containers deleted. The value of `del_c` shall not be zero, i.e. delete indications shall only be generated if at least one deletion has been performed.

A deletion indication container may contain a payload in the `fragment` field. It is application dependent and out of scope of the present document how to create such payloads and, as the receiving entity of such payloads, how to take action in response to them. An example may be the following.

EXAMPLE 1: The payload comprises the human readable string: "Malicious content removed". The endpoint acts on this by terminating the session.

Deletion indication containers shall have a reader and deleter MAC, but shall not have a writer MAC.

The following approach should be used when signalling deletions. A sequence of delete indication containers are sent at different points in time during the "window" of deletions. When the last deletion indicator has been sent, the normal flow of containers resumes, via the middlebox. This approach simplifies handling and is more preserving to the audit history of deletions. Alternatively, if `n` consecutive containers originated by the same entity are deleted from the same context, a single delete indication container may be transmitted after the nth deletion.

EXAMPLE 2: Assume a middlebox has write access to context 1, but has no access to context 3, and assume the middlexbox is in progress of deleting some messages relating to context 1, and which it has not yet reported. At this point a container is received relating to context 3. The middlebox reports all outstanding deletions from context 1, before forwarding the container relating to context 3.

EXAMPLE 3: Assume an entity with `e_id = j` has first generated 5 containers and that a middlebox with `e_id = k` (`k > j`) deletes the last 3 of them. Entity `j` then generates 7 additional containers, out of which the 2 last are deleted by middlebox `k`. Middlebox `k` forwards the two first containers but does not forward (i.e, "disposes of") containers 3, 4, and 5. Middlebox `k` then generates a first delete indication to replace containers 3-5, containing a value pair `src = j, del_c = 3`. Then, middlebox `k` forwards containers 6-10, but disposes of containers 11 and 12. After the 12th received container from entity `j`, a second delete indication will be generated by middlebox `k`, now containing a value-pair `src = j, del_c = 2`. Therefore a total of two delete indication containers are produced by middlebox `k`.

EXAMPLE 4: In the same scenario as above, middlebox `k` could alternatively replace each deleted container by exactly one delete indication container, each having `src = j, del_c= 1`, resulting in a total of 5 delete indication containers.

NOTE: It could be tempting to conceptually view the deletion of a single container as a modification, rewriting an original container as a delete indication. However, this view does not extend to the multiple-deletion case, which is why a delete of one or more containers is defined as the removal of those containers followed by the insertion of a deletion indication.

4.2.3.1.4.2 Sequence number handling

As deletion indication containers represent more than one container, additional steps are required, beyond those described in clause 4.2.2.3, to update the sequence number state when processing them.

When a deletion indication is prepared for transmission in context `c`, after the updates to `seq_tx` and `seq_tx_C[c]` defined in clause 4.2.2.3.2, the following additional updates are performed:

- `seq_tx = seq_tx + del_c`
- `seq_tx_C[c] = seq_tx_C[c] + del_c`

When a deletion indication is received by entity `k` in context `c`, after the updates to `seq_rx[]` and `seq_rx_C[][c]` defined in clause 4.2.2.3.3, the following additional updates are performed:

- `seq_rx[j] = seq_rx[j] + del_c, for j = src, src+1, …, k-1`
- `seq_rx_C[j][c] = seq_rx_C[j][c] + del_c, for j = src, src+1, …, k-1`

4.2.3.1.5 Audit containers

Audit containers are `Application` protocol message units that convey information, as a payload in the container `fragment` field, pertaining to the processing of application data containers or the generation of alert containers. The production of audit containers shall be configured on a per-context basis during the handshake, see clause 4.3.5. Production of audit containers shall not occur prior to completion of the handshake.

If the `audit` parameter for a context is configured with the value `audit_info`, all entities with at least read access to the context may originate audit containers pertaining to application data or alert containers in that context. While the contents of the audit payload are out of the scope of the present document, they may provide processing hints to downstream entities for downstream entities, describe actions taken by middleboxes on the associated application data containers, or provide supplemental information for an alert.

If the `audit` parameter for a context is configured with the value `audit_trail`, the meaning is the same as for `audit_info`, with the additional requirement that every middlebox with at least read access to the context shall insert audit containers for all application data containers associated with the context for which a non-trivial action was taken. Which actions to consider as non-trivial is application dependent and left oustide the scope of the present document.

EXAMPLE 1: For writer middleboxes, performing insert, modify, or delete could be considered non-trivial actions.

When originating an audit container, an entity shall:

- set the `ctxt_id` field of the container header to the context of the associated containers; and
- set the A-bit of the `flags` container header field to 1.

Audit containers shall have a reader and writer MAC, but shall not have a deleter MAC. Audit containers shall not be modified or deleted by middleboxes.

Regarding placement of the audit container, if it pertains to a deleted container, the audit container should be inserted after the delete indication corresponding to the deleted container. In all other cases, the audit container should normally be inserted immediately after the container associated with the audit information. However, as long as the audit container contains enough information to identify the application data container(s) to which the audit information is related, its may be originated in the session at any point.

EXAMPLE 2: An audit container could be placed immediately after a modified (or inserted) container or immediately after a delete indication.

EXAMPLE 3: An entity could provide processing hints to downstream entities by placing the audit container ahead of the associated application data container(s).

4.2.3.1.6 Alert containers

Alert containers are `Alert` protocol message units that signal error or warning conditions to downstream entities. They begin to be used in a session as described in clause 4.4.1. Endpoints may originate an alert container in any context, and a middlebox shall only insert an alert container in a context to which it has at least read access (which includes at least context zero). The `ctxt_id` field of the container header shall be set to the context to which the alert applies.

EXAMPLE: `ctxt_id` = 0 is used for alerts relating to the handshake itself.

Alert containers shall have a reader and writer MAC, but shall not have a deleter MAC. Alert containers shall not be modified or deleted by middleboxes.

NOTE 1: Prior to `ChangeCipherSpec`, alert containers will have zero-length MACs.

The key selection for alert container MACs described in clause 4.2.7.2.2 allows all middleboxes, with any level of granted context access, to verify the integrity of an alert container and also allows the endpoint(s) to verify the authenticity.

NOTE 2: By definition, all middleboxes have write access to context zero and are therefore always authorized to insert `Alert` protocol messages/containers associated with context zero. Refer to annex E for security considerations.

4.2.3.2 Record message units

4.2.3.2.1 Handshake message units

A handshake record is a `Handshake` protocol message unit that contains one or more `Handshake` messages. Middleboxes shall not:

- delete or replace `Handshake` messages except under the message-specific conditions stated in clauses 4.3.6 and 4.3.7;
- modify parts of `Handshake` messages added by other entities, except as defined for middlebox discovery in clause 4.3.2.3; or
- following `ChangeCipherSpec`, forward the contents of an inbound handshake record using more than one outbound record, or combine the contents of more than one inbound handshake record into one outbound record.

Following `ChangeCipherSpec`, handshake records have a reader MAC, but do not have a deleter MAC or a writer MAC, and have a hop-by-hop MAC. Handshake records are always associated with context zero.

4.2.3.2.2 ChangeCipherSpec message units

A `ChangeCipherSpec` record is a `ChangeCipherSpec` protocol message unit that contains one `ChangeCipherSpec` message. There are never any MACs present on a `ChangeCipherSpec` record.

4.2.3.3 Middlebox processing summary

A middlebox shall never insert, delete, or modify messages in other protocols than those described in clauses 4.2.3.1 and 4.2.3.2. Table 1 summarizes for each protocol whether containers shall be used and which operations on message units are allowed.

Table 1: Middlebox processing summary

Protocol	Use of Containers	Middlebox Modifications or Deletions Permitted	Middlebox Insertions Permitted
`Handshake`	No	Only under the message-specific conditions stated in clauses 4.3.2.3, 4.3.6 and 4.3.7, and via piggy-backing as described in clause 4.3.1.	Yes
`ChangeCipherSpec`	No	No	No
`Alert`	Yes	No	Yes, for contexts to which at least read access is granted.
`Application`	Yes	Yes, by deleter and writer middleboxes but only to non-audit containers. Only writer middleboxes may modifiy a container in other ways than deletions. Any middlebox may abort the session.	Yes. Application data containers may be inserted by writer middleboxes. Deletion indication containers may be inserted by deleter middleboxes. Depending on per-context audit configuration, audit containers may be inserted by reader middleboxes.

4.2.3.4 MAC usage summary

Table 2 summarizes MAC usage. The value i below refers to the entity identity where a message is currently being processed, entity i+1 then being the downstream neighbour and dest being the downstream endpoint destination. RK(c)/DK(c)/WK(c), respectively, denote the reader/deleter/writer key for context c, and PK(i,j) denotes the pairwise key shared only between entity i and j. The sets R(c)/D(c)/W(c), respectively, denote the set of entities with read/delete/write access to context c. Sequence number usage in MAC computation is indicated in the SEQ column: G means that the global, context-independent sequence number is used, and A means that the array of all context-dependent sequence numbers are used.

Below, a MAC data input is considered explicit if it is part of the information explicitly carried in the TLMSP message. A MAC data input is considered implicit if it is not carried explicitly in the message. In the last column, MAC author pertains to the author of the deleter, writer, and hop-by-hop MAC only (the author of the reader MAC is identical to the overall author of the message).

RM/DM/WM are used as abbreviations of reader/deleter/writer MAC respectively, and HBH MAC denotes the hop-by-hop MAC.

The `ChangeCipherSpec` protocol is not included since it is never protected as TLMSP does not support renegotiation.

Table 2: Summary of MAC usage

MAC type	SEQ	MAC key-usage per TLMSP sub-protocol					MAC calculations (see note 1)		Explicit data coverage						Implicit data coverage		
		Application			Alert	Hand-shake	Generation	Verification	Record Header	Container info					Author	MAC author	
		Container type								Header	Data fragm (see note 2)	MACs			SEQ	ID	SEQ
		Normal	Audit	Delete-ind								RM	DM	WM			
Reader MAC	G	RK(c)	RK(c)	RK(c)	RK(c)	RK(0)	W(c), but only when creating or modifying message unit	R(c)	Y	Y	Y (see note 3)	N	N	N	Y	n/a	n/a
Deleter MAC	A	DK(c)	n/a	DK(c)	n/a	n/a	D(c)	D(c)	Y	Y	Y (see note 4)	Y (see note 4)	N	N	Y (see note 5)	Y	Y
Writer MAC	A	WK(c)	PK(i,dest)	n/a	PK(i, dest)	n/a	W(c), for application data containers, R(c) for audit containers, and all entities for alert containers	W(c), for application data containers, dest for audit and alert containers	Y	Y	Y (see note 4)	Y (see note 4)	N	N	Y (see note 5)	Y	Y
HBH MAC	G (see note 7)	PK(i,i+1)	PK(i,i+1)	PK(i,i+1)	PK(i,i+1)	PK(i,i+1)	All entities	All entities	Y	Y	Y (see note 4)	Y (see note 4)	Y	Y	Y	N (see note 6)	Y

NOTE 1: Only entities that are currently participating in the session, see clause 4.3.8.
NOTE 2: The data fragment includes the explicit IV, which in turns always explicitly includes the author's entity ID and implicitly the author's SEQ. Thus, the author identity is always explicitly included in the reader MAC and the author SEQ is implicitly included.
NOTE 3: Covers the unencrypted plaintext of the payload, before encryption was applied.
NOTE 4: Covers the encrypted value, after encryption was applied.
NOTE 5: Since the author SEQ is input to the reader MAC (2), and the reader MAC is input to this MAC, the author SEQ is implicitly input also to this MAC.
NOTE 6: While the entity ID of the MAC author is neither explicitly nor implicitly included in all cases, the MAC key used is unique to MAC author.
NOTE 7: For records composed of one more containers, the hop-by-hop MAC uses the same global sequence number value as that used by the first container of the record.

4.2.4 Container format

For `Application` and `Alert` protocols, the container format shall be defined as in the present clause. The "payload" (or fragment) part of the container shall have a type that varies depending on whether the content is unprocessed plaintext, compressed plaintext or protected ciphertexts. This last type is ready for submission to the TCP layer.

```
struct {
    uint8 context_id;
    uint16 flags;
    select (flags & 0x8000) {   /* Check if I-bit = 1  */
       case true: struct {
                     uint8 e_id;
                     select (flags & 0x4000) {   /* Check if D-bit = 1  */
                        case true: struct {
                                      uint8 src;     /* originator of delete message units */
                                      uint16 del_c;  /* delete count */
                                   };
                        case false: struct { };    /* empty */
                     };
                  } m_info;
       case false: struct { };    /* empty */
    };
    uint16 length;
    select (TLMSP_internal_layer) {
       case TLMSPPlainText:  opaque;
       case TLMSPCompressed: opaque;
       case TLMSPCipherText: ContaineredFragment;
    } fragment;
} Container;
```

The value of `length` shall be the octet length of `fragment`.

4.2.5 Plaintext record format

The plaintext record shall be defined as in the present clause:

```
opaque HopID<0..16>;

struct {
    ContentType type;
    ProtocolVersion version;
    uint16 tot_length;
    select (tlmsp_server_support_confirmed) {
       case true:  HopID hbh_id;
       case false: struct { };    /* empty */
    };
    select (type) {
       case 0x15, 0x17:                                          /* Application, Alert */
              Container containers[tot_length - 4];
       case 0x14, 0x16:                                          /* ChangeCipherSpec, Handshake */
              opaque fragment[tot_length - tlmsp_server_support_confirmed ? 4 : 0];
   };
} TLMSPPlainText;
```

Prior to the confirmation of server support of TLMSP, `tot_length` is the length of `fragment` for all protocols.

Following the confirmation of server support of TLMSP, for the `Application` and `Alert` protocols, `tot_length` is the length of `hbh_id` plus the total length of all the containers and can be calculated as:

$$\texttt{tot\_length} = 1 + \texttt{length(hbh\_id)} + \Sigma_{\texttt{c}}[5 + \texttt{c.length} + ((\texttt{c.flags} \,\&\, 0\texttt{x8000})\ ?\ 1 : 0) + ((\texttt{c.flags} \,\&\, 0\texttt{x4000})\ ?\ 3 : 0)]$$

where `length(hbh_id)` is number of octets of `hbh_id` indicated by its encoded length, and the sum is taken over all the containers, `c`, in the `containers` vector, and for the `ChangeCipherSpec` and `Handshake` protocols, `tot_length` is one plus `length(hbh_id)` plus the length of `fragment`.

NOTE: This ensures that the header part (`type, version, tot_length`) is compatible on record-level with that of IETF RFC 5246 [1].

4.2.6 Compressed record format

The present document does not specify use of compression for reasons discussed in annex H and any proposed compression method other than `null` shall be rejected by the TLMSP version defined herein. However, for possible future extensions, a compressed record format is defined in the present clause:

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 tot_length;
    select (tlmsp_server_support_confirmed) {
       case true:  HopID hbh_id;
       case false: struct { };    /* empty */
    };
    select (type) {
       case 0x15, 0x17:                                          /* Application, Alert */
              Container containers[tot_length - 4];
       case 0x14, 0x16:                                          /* ChangeCipherSpec, Handshake */
              opaque fragment[tot_length - tlmsp_server_support_confirmed ? 4 : 0];
   };
} TLMSPCompressed;
```

This is identical in structure to `TLMSPPlaintext`. The difference is that the contents of `fragment`, or the contents of each container's `fragment`, depending on the protocol, have been been compressed, which will be reflected in their corresponding length values.

The value `tot_length` can be computed using the same approach as for the `TLMSPPlaintext` structure defined in clause 4.2.5.

4.2.7 Applying message unit and record protection

4.2.7.1 General

As in TLS 1.2 [1], the record layer of TLMSP is generally responsible for applying data protection to the sub-protocols forming the complete TLMSP protocol-suite (the `Handshake`, `ChangeCipherSpec`, `Alert`, and `Application` protocols). In TLMSP, the protection applied at the record layer can conceptually be viewed as composed of four sub-layers: reader layer, deleter layer, writer layer, and forwarding/record layer, applied in that order, using different keys. The reader layer applies encryption and integrity protection, whereas the other layers only apply integrity protection. The result of this layering is that for payload protection, up to three additional MAC values are typically added to the basic reader layer integrity protection, creating up to four MAC values in total. The exact details of which MACs to add are defined in clause 4.2.3 and how to compute them is defined in clause 4.2.7.2.

The protected record format shall be:

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 tot_length;
    HopID hbh_id;
    select (type) {
     case 0x15, 0x17: struct {                                 /* Application, Alert */
            Container containers[TLMSPCompressed.tot_length-4];
            opaque hop_by_hop_mac[SecurityParameters.mac_length];
            };
     case 0x14:                                                /* ChangeCipherSpec */
            opaque fragment[TLMSPCompressed.tot_length-4];
     case 0x16: struct {                                       /* Handshake */
            opaque fragment[TLMSPCompressed.tot_length-4];
            opaque hop_by_hop_mac[SecurityParameters.mac_length];
            };
   };
} TLMSPCipherText;
```

where the `containers` field shall be the result of applying the selected TLMSP cipher suite to the corresponding `TLMSPCompressed.containers`, on a per-fragment basis. For `ChangeCipherSpec` and `Handshake` protocols, containers shall not be used and the `fragment` data shall be of the generic type `Fragment`, which is defined in a cipher-suite dependent way, as follows:

```
select (SecurityParameters.cipher_type) {
   case stream:     GenericStreamCipher;
   case block:      GenericBlockCipher;
   case aead:       GenericAEADCipher;
} Fragment;
```

NOTE: The format of a `Fragment` is backward-compatible with the format for TLS fragments [1]. In particular, the TLMSP reader MAC can be identified with the MAC value included in the output format of one of the three generic formats.

When the fragments are part of a container (`Application` and `Alert` protocols) they shall be of type `ContaineredFragment`, defined as:

```
struct {
     Fragment c_fragment[TLMSPCipherText.container.length]; /* Incl. reader_mac, IV, padding */
     [opaque deleter_mac[SecurityParameters.mac_length];]
     [opaque writer_mac[SecurityParameters.mac_length];]
} ContaineredFragment;
```

The value of `TLMSPCipherText.tot_length` shall be computed by adding the following terms:

- The length of `hbh_id` (including its encoded length field).
- The sum of the per-container values:
  - `TLMSPCipherText.containers.length`, which shall be calculated as in clause 4.2.7.2.2 (including the container header and fragment size, in particular the sizes of the IV, any possible padding, and the reader MAC).
  - The sum of the sizes of: writer MAC (see clause 4.2.7.2.2) if present, deleter MAC (see clause 4.2.7.2.2) if present.
- The length of the hop-by-hop MAC (clause 4.2.7.2.3).

4.2.7.2 MAC generation

4.2.7.2.1 General

An overview of the MACs used for message units and records is provided in clause 4.2.2.2.

For AEAD ciphers such as GCM, the integrity mechanism included in the cipher mechanism shall be used as the reader MAC. The reader MAC value shall for `Application` and `Alert` protocols be included inside the `TLMSPCipherText.containers.fragment.c_fragment` field, or otherwise, in the `fragment` field of the message, in the same way MACs are included in TLS 1.2 [1].

4.2.7.2.2 Reader, deleter and writer MACs

4.2.7.2.2.1 Container message units

For protocols that use containers (`Alert` and `Application`), the MAC generation schemes defined in the present clause shall be used.

For generic stream and block ciphers (using a standalone MAC), for each `TLMSPCompressed.containers.fragment` the corresponding reader and writer MAC values shall be computed as follows:

```
MAC(mac_key, mac_input);
```

where:

```
mac_input = seq_num || TLMSPCompressed.containers.flags ||
            [TLMSPCompressed.containers.m_info] ||
            length || data || [e_id]
```

where in turn:

- **MAC** shall be the message authentication algorithm of the selected cipher suite.
- **mac _key** shall in all cases be the reader key for the reader MAC. For the deleter and writer MAC, there are two cases. For `Application` protocol containers that are not audit containers, the key shall be the deleter and writer key, respectively. The key used for the reader, deleter, and writer MACs shall furthermore be the one applicable for the current context (as defined by `i = TLMSPCompressed.container.ctxt_id`) and for the direction of transmission/reception.

  EXAMPLE: The `mac_key` is `client_to_server_writer_mac_key_i` (defined in clause 4.3.10.5) for a writer MAC on an application data container related to context `i` in the client-to-server direction, computed by an entity with write access.

  For `Application` protocol audit containers, and, for `Alert` protocol containers, the key for the writer MAC shall be the (context-independent) MAC key shared only with the destination end-point, i.e. corresponding to `e1_to_e2_mac_key`, as defined in clause 4.3.10.4, where `e2` is the destination end-point. There shall be no deleter MAC for these type of messages.
- **seq_num** shall be as determined in clause 4.2.2.3, specifically, with `n_ctxt` being equal to the total number of contexts in the session:
  - for a reader MAC:
    - when generating: the value `seq_tx`;
    - when verifying: the value `seq_rx[k]`, where `k` is the MAC author.
  - for a writer or deleter MAC:
    - when generating: seq_num = seq_tx_C[0] || seq_tx_C[1] || … || seq_tx_C[n_ctxt-1];
    - when verifying: seq_num = seq_rx_C[k][0] || seq_rx_C[k][1] || … || seq_rx_C[k][n_ctxt-1], where k is the MAC author.
- **length** shall be a 16-bit unsigned integer and:
  - when computing a reader MAC value, it shall be assigned the value `LR = TLMSPCompressed.containers.length`;
  - when computing a deleter or writer MAC value, it shall be assigned the value `LW = LR + SecurityParameters.record_iv_length + SecurityParameters.padding_length + SecurityParameters.mac_length` with `LR` as above.
- **data** shall be:
  - when computing the reader MAC: `TLMSPCompressed.containers.fragment`;
  - when computing the deleter MAC: `TLMSPCipherText.containers.fragment.c_fragment`, (which includes the IV, padding and the reader MAC);
  - when computing the writer MAC: `TLMSPCipherText.containers.fragment.c_fragment` (which includes the inputs same as the deleter MAC, but not the deleter MAC itself).
- the optional entity ID, **e_id,** shall be the entity ID of the MAC author and shall be present only when computing or verifying the deleter and writer MAC.

The reader MAC is calculated based on the compressed plaintext, before encryption. However, the deleter and writer MAC shall be calculated based on the result after reader security processing. Thus, the value of `TLMSPCipherText.containers.length` shall be updated after adding the reader MAC and performing other security processing to include the lengths of the IV, any possible padding, and the reader MAC itself. This updated length value is used as input, `LW`, to the deleter and writer MAC. However, the value of `TLMSPCipherText.containers.length` shall not be further updated after calculating and appending the deleter and writer MAC.

NOTE: In TLS, the sequence number is the first input to the MAC. For TLMSP sub-protocols that use containers, the sequence number varies for each container. Therefore, placing `seq_num` as the fourth input allows the three first input fields to be a fixed prefix for all containers included in the record.

For AEAD transforms, following the AEAD interface specification of [4], the plaintext input, `P` (to be encrypted and authenticated), shall consist of the `data` value as defined above. The so-called Additional Authenticated Data, AAD, (not to be encrypted) shall in the case of reader MAC consist of:

```
AAD = seq_num || TLMSPCompressed.containers.flags ||
      [TLMSPCompressed.containers.m_info] || length,
```

and for the writer and deleter MAC, `AAD` shall be the same as `mac_input.`

The actual computation of stand-alone MAC values (i.e. other than the first reader MAC) is, in the case of AEAD, transform-dependent. MACs of AEAD transforms may also require an IV. See clause A.3.2 for the pre-defined AEAD transform.

4.2.7.2.2.2 Record message units

For message units that do not use the container format, only reader MAC values shall be computed, and the details of clause 4.2.7.2.2.1 shall apply with the following changes.

The input shall be:

```
mac_input = seq_num || TLMSPCompressed.type || TLMSPCompressed.version ||
            TLMSPCompressed.tot_length || TLMSPCompressed.hbh_id ||
            TLMSPCompressed.fragment
```

NOTE: A zero-length `TLMSPCompressed.hbh_id` is present as a one-octet length-indicator (having the value zero).

When using AEAD transforms, the AAD, (not to be encrypted) shall consist of:

```
AAD = seq_num || TLMSPCompressed.type || TLMSPCompressed.version ||
      TLMSPCompressed.tot_length || TLMSPCompressed.hbh_id
```

4.2.7.2.3 Hop-by-hop MAC

This MAC shall cover the entire record excluding the hop-by-hop MAC itself.

Each entity maintains a concept of who its upstream neighbour and downstream neighbour are for each direction of communication (client-to-server and server-to-client). In a given direction, an entity's upstream neighbour is the next middlebox upstream who is participating (see clause 4.3.8.2.2 for the notion of participating). If there is no such middlebox, the upstream neighbour is the upstream endpoint (or is non-existent if the entity is the endpoint that transmits in that direction). Likewise, that entity's downstream neighbour is the next middlebox downstream whose current state is participating. If there is no such middlebox, the downstream neighbour is the downstream endpoint (or is non-existent if the entity is the endpoint that receives in that direction).

When generating a hop-by-hop MAC for a record that is prepared to be transmitted, an entity shall use the pairwise key it shares only with its current downstream neighbour (with neighbour as defined above), and derived via the definitions of clauses 4.3.10.3 and 4.3.10.4 (for non-AEAD transforms, the `e1_to_e2_mac_key` shall be used between transmitting entity `e1` and downstream entity `e2`).

Similarly, when verifying a hop-by-hop MAC for a record being received, an entity shall use the pairwise key it shares with its current upstream neighbour (for non-AEAD transforms, the `e1_to_e2_mac_key` shall be used between upstream entity `e1` and receiving entity `e2`).

The processing shall be as in clause 4.2.7.2.2.1 with the following changes.

The MAC input shall be:

```
mac_input = seq_num || TLMSPCipherText.type || TLMSPCipherText.version ||
            TLMSPCipherText.tot_length || TLMSPCipherText.hbh_id ||
            record_payload
```

NOTE: A zero-length `TLMSPCipherText.hbh_id` is present as a one-octet length-indicator (having the value zero).

where `seq_num` is chosen as for reader MACs, and where `record_payload` is `TLMSPCipherText.containers` when the record is composed of one or more containers, and `TLMSPCipherText.fragment` otherwise. The `hbh_id` is the identity chose by the sending entity.

4.2.7.3 Cipher suite specifics

4.2.7.3.1 General

All TLMSP cipher suites shall use an initialization vector explicitly carrying at least the one-octet entity identity of the middlebox that generated or most recently modified the message. The selected encryption IV shall, for the pre-defined cipher suites, follow the definitions of annex A.

For protocols using containers, the selected cipher converts `TLMSPCompressed.containers.fragment` structures to and from `TLMSPCipherText.containers.fragment.c_fragment` structures, and for protocols not using containers, converts `TLMSPCompressed.fragment` to `TLMSPCipherText.fragment`.

The structures for enciphered data are defined for each type of cipher in the following clauses. In these clauses, `content_length` refers to:

- the length of the corresponding `TLMSPCompressed.containers.fragment`, for protocols that use containers; and
- the length of the corresponding `TLMSPCompressed.fragment`, for protocols that do not use containers.

If the cipher suite is `TLMSP_NULL_WITH_NULL_NULL`, then security processing consists of the identity operation (i.e. the data is not encrypted and the MAC length is zero for reader and writer MACs). If a cipher suite of type `TLMSP_X_WITH_NULL_Y` is used, where `X` and `Y` are any non-null cryptographic transforms, then the data shall not be encrypted, but a reader MAC of non-zero length shall be present, and depending on the MAC algorithm, potentially also a nonce.

4.2.7.3.2 Null or stream cipher

In contrast to TLS, all TLMSP stream ciphers shall use an explicit IV. This allows middleboxes to modify/insert/delete containers.

```
struct {
    opaque IV[SecurityParameters.record_iv_length];
    stream-ciphered struct {
       opaque content[content_length];
       opaque reader_mac[SecurityParameters.mac_length];
    };
} GenericStreamCipher;
```

The `IV` and the `reader_mac` shall be created prior to encryption. The encryption shall then be performed, using the stream cipher to encrypt the `content` and the `reader_mac` as defined in IETF RFC 5246 [1].

The length of `GenericStreamCipher` is:

```
SecurityParameters.record_iv_length + content_length +
SecurityParameters.mac_length.
```

4.2.7.3.3 Generic block cipher

```
struct {
    opaque IV[SecurityParameters.record_iv_length];
    block-ciphered struct {
        opaque content[content_length];
        opaque reader_mac[SecurityParameters.mac_length];
        uint8 padding[padding_length];
        uint8 padding_length;
    };
} GenericBlockCipher;
```

The `padding` and `padding_length` shall be as specified in clause 6.2.3.2 of IETF RFC 5246 [1].

The length of `GenericBlockCipher` is:

```
SecurityParameters.record_iv_length + content_length +
SecurityParameters.mac_length + padding_length + 1.
```

4.2.7.3.4 AEAD ciphers

The AEAD transform defined in the present document use a combination of explicitly signalled and locally derived values to form the IV.

```
struct {
    opaque nonce_explicit[SecurityParameters.record_iv_length];
    aead-ciphered struct {
        opaque content[content_length + D + SecurityParameters.mac_length];
    };
} GenericAEADCipher;
```

The reader MAC is included in the `content` field directly by the AEAD transform, see TLS 1.2 (IETF RFC 5077 [2]), clause 6.2.3.3. The value `D` corresponds to padding and other overhead added by the AEAD transform in use.

The length of `GenericAEADCipher` is:

```
SecurityParameters.record_iv_length + content_length + D +
SecurityParameters.mac_length.
```

4.3 The Handshake protocol

4.3.1 Overview

4.3.1.1 General

The cryptographic parameters of the session state are produced by the TLMSP `Handshake` protocol, which operates on top of the TLMSP record layer. When a TLMSP client and server first communicate, they agree on a protocol version, the number of contexts and their purpose(s), the middleboxes' granted access privilege level, and the cryptographic algorithm suite to use. The TLMSP `Handshake` protocol generally involves the following steps (as in standard TLS, certain steps, marked by * in Figure 6 may be omitted if the information is already known):

- Exchange of:
  - Hello messages to establish which contexts to use, propose algorithms and middleboxes, random values, authentication methods, and possible indications of session resumption.
  - Certificates (or other credentials) and cryptographic information to allow the client, server and middleboxes to authenticate themselves.

- Necessary cryptographic parameters. The server chooses one cipher suite that lies in the intersection of those supported by the client and the server. Since, except for manipulations of extensions to the `ClientHello`, middleboxes shall typically not engage in the handshake before observing the `ServerHello`, the server should be pre-configured with knowledge of the cipher suite support of all the middleboxes in the middlebox list and choose a secure cipher suite in the intersection of those supported by also all middleboxes. Alternatively, the server may propose a secure and mandatory-to-support cipher suite.

- Agree on keys shared between the client and server endpoints and between middleboxes and endpoints.
- Mutual authorization of middlebox access privilege levels by providing key-shares from both client and server.
- Allow entities to verify that their peer(s) have calculated the same security parameters, including the list of middleboxes and their respective permissions requested, and that the handshake occurred without tampering by unauthorized parties.

The TLMSP handshake shall use a TLMSP extension added to the `Hello` messages in the TLS handshake to agree on the authorized middleboxes and the contexts. An additional `Handshake` message, `TLMSPKeyMaterial`, shall be used to grant access rights to a middlebox by sending the necessary contribution(s) for that middlebox to derive the corresponding cryptographic keys.

Each middlebox shall receive such a contribution from both client and server to grant a particular access right to a particular context; knowledge of a contribution from only one endpoint does not weaken the level of security of the end-to-end agreed session. The client and server shall send a `TLMSPKeyMaterial` message to each middlebox participating in the connection. A contribution shall not be present in the message destined to a particular middlebox if the endpoints agreed to withhold the corresponding access permission to the context from the middlebox. Each middlebox shall transform the `TLMSPKeyMaterial` message destined to it into a `TLMSPKeyConf` message before forwarding it to the next entity in order to provide the endpoints with key confirmation, i.e. providing cryptographic proof to an endpoint that all middleboxes have received their shares from the other endpoint, before the data session starts. This prevents an endpoint from unilaterally removing a priori agreed access rights from a certain middlebox. TLMSP shall also add cryptographic verification messages (`MboxFinished`) of the handshake with each middlebox.

Until the first `ChangeCipherSpec` message, there shall only be the single context with the reserved `ctxt_id = 0` in use, which at that point shall not use any protection (the cipher suite shall be `TLMSP_NULL_WITH_NULL_NULL`). Application data shall not be sent until after the associated contexts have been agreed and the handshake has fully completed. After this, a cipher suite with a non-NULL integrity algorithm shall always be selected. The currently defined cipher suites are defined in annex A. `Handshake` and `ChangeCipherSpec` messages shall not be transmitted in any other context than context zero.

The signalling diagram below assumes that the middlebox configuration and discovery of clause 4.3.2 has been completed and that the server supports TLMSP, which it shall indicate by inserting the TLMSP extension, `TLMSP` (including middlebox list, L), into its `ServerHello` as acknowledgement of the presence of the same extension in the `ClientHello`. When this extension is absent, the fallback mechanisms of clauses C.1 or C.2 may be used.

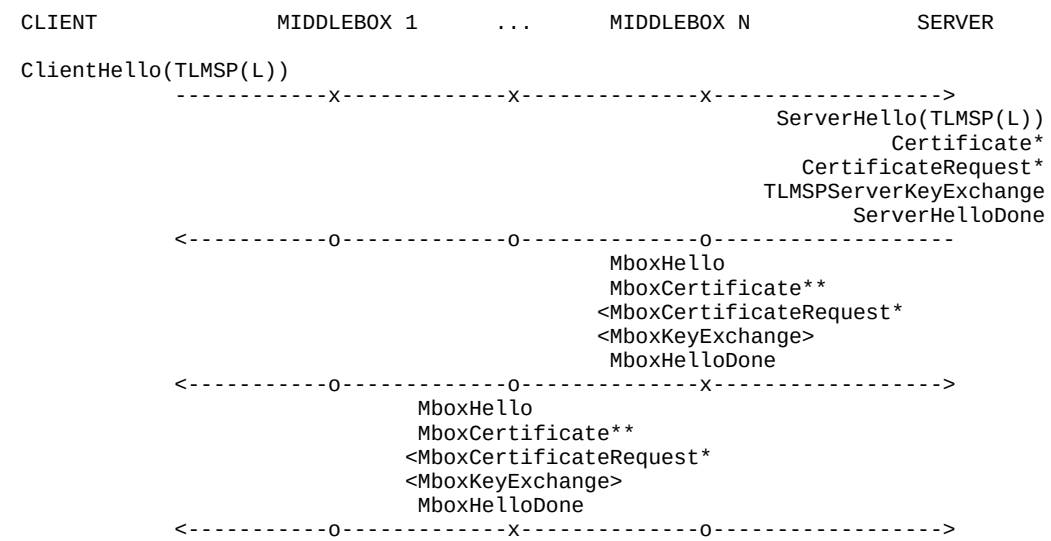


**Figure 6: Handshake, optional messages are suffixed by *,
messages which could occur in zero, one, or two directions are suffixed by ****

In Figure 6, x indicates that the middlebox inserts data and forwards the message; o indicates the middlebox is able to read/process content, but does not modify it, and then forwards the contents. A `Handshake` message may always be sent as a standalone handshake record, and where possible may instead be sent piggy-backed according to clause 4.3.1.2.

For middleboxes, their `MboxHello, MboxCertificate`, `MboxKeyExchange` and `MboxHelloDone` messages may be sent piggy-backed toward the client, but shall be sent beginning with a new record toward the server. Also, these messages shall have identical content both when sent to the server and to the client.

If sent, the `MboxCertificateRequest` shall be sent or piggy-backed only towards the client. When used, `MboxCertificateRequest` requests client authentication by a middlebox.

The MboxCertificateRequest defined in clause 4.3.6.3 can be sent independently of whether the server sends a CertificateRequest. Moreover, the client can in response provide different certificates to different middleboxes. For each middlebox to which the client sends a certificate, the client shall also send (or piggy-back) a CertificateVerify2Mbox message as defined in clause 4.3.6.7.

NOTE: From Figure 6, it can be seen that TLMSP uses a special TLMSPServerKeyExchange instead of the standard ServerKeyExchange in TLS [2]. Additionally, the order between CertificateRequest and the key exchange is reversed compared to TLS. This enables authentication of the certificate requests and protects against unauthorized harvesting of the client's certificate, see clause 4.3.10.1 for details.

The optional piggy-backing is described in more detail in clause 4.3.1.2. TLMSPKeyMaterial[Mi] denotes a message containing middlebox key shares from an endpoint directed to middleboxM and TLMSPKeyConf[Mi] denotes a middlebox's key confirmation message from middlebox ei to an endpoint. As seen, these are piggy-backed (and aggregated) into forwarded TLMSPKeyMaterial messages. MboxFinished[ei,ej] is a verification message of the handshake exchanges dependent on messages previously exchanged between (or available to) both entities ei and ej, except when both ei and ej are middleboxes, in which case no MboxFinished[ei,ej] message shall be present. For messages originating at a middlebox and potentially sent to both endpoints, messages prefixed by < (or suffixed by >) are sent only in the indicated direction. Messages embraced inside <…> are sent in both directions, but possibly with different content. Middlebox messages shown above bi-directional signalling arrows, but without any of these angle-brackets, are sent in identical copies to both endpoints.

For the definition of the Handshake protocol, message structures that are not defined in the present document shall be as defined in and unchanged from structures of the same name in clauses 7.3 and 7.4 of IETF RFC 5246 [1].

The signalling flow of Figure 6 should be followed since it has the property that no middlebox starts to send messages until after the ServerHelloDone has been observed. It is only at this point that all entities can be assured that the server really supports TLMSP so that none of the fallbacks of clause C.1 or C.2 are necessary. Also, it is only at this point that all entities know whether any additional middleboxes could enter into the session via dynamic discovery as defined in clause 4.3.2. If a middlebox has started to send messages before the above knowledge has been obtained, there is in general no guarantee that the handshake succeeds. Nevertheless, clause C.3 describes an alternative flow which is useful in some scenarios and may be used when it is known that the first on-path middlebox has certain features, see clause C.3 for details. A general exception to this rule is that middleboxes may add or manipulate TLMSP-specific extensions provided in the ClientHello, see clause 4.3.2 and clause C.2. This is safe since the server ignores unknown extensions.

4.3.1.2 Piggy-backing of handshake messages

Piggy-backing intuitively means that a middlebox appends a Handshake message with itself as origin to an already in-transit record comprising a Handshake message that originates from an upstream endpoint. More formally, the piggy-back of handshake information by middleboxes shall be done as follows.

Assume without loss of generality that a middlebox, MBa, wishes to piggy-back information in a message from server to client, such as in the server's response to the ClientHello. This server message is in current TLS implementations and typically consists of several individual messages combined into one record R:

```
+------+---------+------------+---~+---~+---~+---~+
| type | version | tot_length | M1 | M2 | M3 | M4 |
+------+---------+------------+---~+---~+---~+---~+
```

where M1 is a ServerHello, M2 is a ServerCertificate, M3 is a ServerKeyExchange and M4 is a ServerHelloDone. Type will have the value 0x16, identifying the message(s) as belonging to the Handshake protocol. The message M1 has the form:

```
+---------~+---------+-------------~+
| msg_type | length  | message_data |
+---------~+---------+-------------~+
```

where msg_type = 0x02, signifying a ServerHello. Similar sub-structures are used for M2, M3 and M4, each with a distinguishing MSG_type.

Suppose MBa wishes to piggy-back a `MboxHello` (`MH`) by appending it into R. To this end, MBa shall create a new record, R', as follows:

```
+------+---------+------------+---~+---~+---~+---~+---~+
| type | version | tot_length | M1 | M2 | M3 | M4 | MH |
+------+---------+------------+---~+---~+---~+---~+---~+
```

where `tot_length` shall have been increased by the length of `MH` and where `MH` shall follow the format of a `MboxHello` as defined in clause 4.3.6. In particular, `MH` shall have format

```
+---------~+---------+-------------~+
| msg_type | length  | message_data |
+---------~+---------+-------------~+
```

where `msg_type = 0x28` (`MboxHello`) and `length` is calculated in accordance with the total data length.

The format of `MboxHello` and other middlebox specific `Handshake` messages specifies that the first part of `message_data` is the middlebox ID. This way, identification of which middlebox that performed the piggy-backing is straightforward at the receiving endpoint. Also, the original content (from the server) is easily identified due to having distinct `msg_type` values in M1-M4 which are never re-used by a middlebox-originated `Handshake` message.

It is also straightforward for a middlebox MBa to piggy-back further messages into R (appending them at the end of the record). Also, it is straightforward for a second middlebox, MBb, to perform further piggy-backing, by appending to the record R' produced by MBa. A middlebox that piggy-backs a message part to a protected handshake record shall re-calculate the single (reader) MAC value. This MAC value shall be based on the new (increased) total record length value. The middlebox shall then re-encrypt the record, setting itself as author (via the entity ID in the IV).

Use of piggy-backing shall be optional and when used, shall be according to the following principles:

a) Piggy-backing shall not be applied to messages occurring after `ChangeCipherSpec`.

b) Piggy-backing shall not be performed if it results in data of a total length that needs to be split into two or more records. Instead, a new, separate record aligned with the start of the new message shall be generated.

c) Piggy-backing shall only affect the record into which piggy-backing is performed.

d) Piggy-backing shall be append-only as described above.

NOTE: The replacement of a `TLMSPKeyMaterial` message with a corresponding `TLMSPKeyConf` message described in clause 4.3.7.3 is not considered to be piggy-backing, nor is the replacement of an `MboxFinished` message from an endpoint to a middlebox with the `MboxFinished` message from that middlebox to the other endpoint described in clause 4.3.6.10.

4.3.2 Middlebox configuration, discovery

4.3.2.1 General

This clause describes alternatives of how to configure or establish the `MiddleboxList` with the complete set of middleboxes. There are two main cases: static pre-configuration and dynamic discovery.

Static pre-configuration shall be supported. Dynamic discovery should be supported.

For the purpose of discovery, the `MiddleboxList` in the TLMSP extension of the `ClientHello` shall contain at least one but may also contain two lists of middleboxes. The first list, denoted `ml_i`, shall always be present and shall include those middleboxes a priori known to the client: via static pre-configuration, due to dynamic discovery of middleboxes during previous TLMSP sessions, or, combinations thereof. The order of the middleboxes in `ml_i` shall be according to the overall network topological order and each middlebox shall occur only one time in the list.

NOTE 1: Nothing precludes that the same physical server hosts two or more virtual middlebox functions.

If, and only if, middleboxes are dynamically discovered (and accepted), this shall result in a new `ClientHello` as described below. The TLMSP extension of this second `ClientHello` shall contain two lists of middleboxes: an identical copy of `ml_i` as above, followed by a second list, `ml_d`, containing also middleboxes that were dynamically discovered.

NOTE 2: This creates cryptographic binding to the set of middleboxes that were initially proposed. This is obtained via inclusion of the original list in the `Finished` verification hash of the second handshake.

The lists `ml_i` and `ml_d` shall contain all middleboxes (including also dynamically discovered ones) according to the overall network topological order.

4.3.2.2 Static pre-configuration

In the case of static pre-configuration, the client shall be manually pre-configured with the complete set of middleboxes as per the `MiddleboxList` defined in clause 4.3.5. The list shall be arranged in network-topological order and each middlebox in the list shall occur only once. All the middleboxes in the initial list shall have the `inserted` field set to "static".

NOTE: It is left to the implementation to add robustness in the form of "loop avoidance" among the middleboxes, i.e. to detect if one and the same middlebox occurs in several places of the list.

The client shall have obtained the IP address of the first-hop middlebox. How this is obtained is out of scope of the present document. Each middlebox shall know or shall be able to obtain the IP address of the next-hop middlebox and the last middlebox shall also be able to obtain the IP address of the server. How this is done is out of scope of the present document.

EXAMPLE: IP address retrieved by DNS lookup of the middlebox address (name) field.

The client shall initiate the handshake by sending the `ClientHello` including the TLMSP extension (including a `MiddleboxList`) to the first middlebox. Before each entity (including the client itself) forwards the `ClientHello` to the next entity, it shall set the `previous_entity_id` field of the middlebox list to its own `entity_id`, for usage as described in clause 4.3.2.3.3. The process shall be repeated at each middlebox, setting up a transport connection with the next middlebox, until a transport connection is eventually established between the last middlebox and the server. Messages from server to client shall be handled in the reverse network-topological order, via the middleboxes.

When the server receives the client's middlebox list, it shall decide if to authorize the proposed middleboxes and also their suggested access privilege level to various contexts. If a middlebox cannot be authorized by the server, the server may reject the session, or, respond with a subset of the client's proposed middleboxes in its own middlebox list, and it is then up to the client how to proceed. Optionally, the server may return a middlebox list to the client, with the attribute `forbidden` set for this middlebox as described in clause 4.3.2.3.2, indicating that the client should not include this middlebox on future sessions.

4.3.2.3 Dynamic discovery

4.3.2.3.1 General

In this case, the client and/or server does not know all middleboxes to be potentially involved in the connection.

EXAMPLE: One of the known (pre-configured as in clause 4.3.2.2) middleboxes or the server can request that one or more additional client-unknown middleboxes are added to the `MiddleboxList`. Additionally, a transparent middlebox can request its own addition. To do this, the client uses dynamic discovery.

It is at the discretion of the endpoints whether to accept additional middleboxes that were not statically pre-configured. There are two sub-cases to consider: non-transparent and transparent middleboxes, referring to whether the middleboxes are directly visible on the IP layer.

If a dynamically discovered middlebox is rejected, it may be included in the `ml_d` list, with the `inserted` attribute set to "forbidden". This allows verification of the rejection without granting privileges to the rejected middlebox.

When an additional dynamically discovered middlebox is proposed (by the middlebox itself or the server), the corresponding entry in the middlebox list extension shall be populated by information about which contexts the middlebox is to be authorized to access.The functionality provided by the middlebox shall be populated into the `purpose` field of extension, as defined in clause 4.3.5.

Discovery of transparent and non-transparent middleboxes may be combined with each other as defined in clause 4.3.2.4.

If additional middleboxes are dynamically discovered as the `ClientHello` propagates toward the server, the list of (proposed) middleboxes received at the server will differ from the list originally included by the client. If it turns out that the server does not support TLMSP, the client may chose to accept fallback to TLS by one of the mechanisms defined in annex C. This fallback would encounter problems if the client's list of which middleboxes to include does not agree with that received at the server (for example, the computation of the `Finished` verification hash would fail). Therefore, the first middlebox to detect the server's non-support for TLMSP, i.e. the middlebox closest to the server, shall send a `Handshake` message of type `ServerUnsupport` and shall include, in the `middlebox_info` field, the complete list of middleboxes that it previously forwarded to the server, see clause 4.3.6.9. Other middleboxes shall just forward this message unless they consider it to disagree with their own view of which middleboxes that took part of the discovery, in which case such middlebox may additionally send its own `ServerUnsupport` message. This allows the client both to compute the correct `Finished` verification hash, as well as to make a decision on whether to accept the additional middleboxes to take part in a TLS fallback.

It is again left to implementation to add robustness in the form of "loop detection" during dynamic discovery.

As defined in clauses 4.3.2.3.2 and 4.3.2.3.3, dynamic discovery leads to the client restarting the handshake by sending a new (modified) `ClientHello`. If a middlebox detects that transparent middleboxes wish to join the session, or, that a non-transparent middlebox is proposed by another entity, the middlebox shall not engage in a TLMSP specific handshake until after it observes the `ServerHello` following the second `ClientHello`.

4.3.2.3.2 Non-transparent middleboxes

This clause applies to use cases where middleboxes visible on the IP layer are to be added.

EXAMPLE: An enterprise's security policy mandates traffic being routed via a data-leakage prevention function. Such middleboxes can in general not make their own presence known during the handshake since the handshake cannot be assumed to be passing through such middleboxes. The (enterprise) server is however likely to be aware of such middleboxes.

Therefore, when using TLMSP, the server may propose that an additional middlebox or middleboxes are to be added. When the server receives the `ClientHello` and finds that a IP-routable middlebox is missing from the `MiddleboxList`, the server shall return a `ServerHello`, including the acceptable middleboxes from the list in the `ClientHello`, extended by those non-transparent middleboxes that the server wishes to add. The added middleboxes shall be inserted into the server's list `ml_i` (as defined in clause 4.3.5). The server shall assign the additional middleboxes unique entity identities and shall insert them in correct topological order.

The server's proposed middlebox entries shall have the `inserted` field set to "dynamic" and the `transparency` field set to "false".

The client shall decide whether to accept the proposed middlebox(es) (in the server's middlebox list extension). If so, the client shall proceed as in clause 4.3.2.2, sending a new `ClientHello` containing `hs_id` (the handshake id) from the `ServerHello` and both an identical copy of the original middlebox list, as well as a list of all middleboxes, including also the discovered and accepted middlebox(es) into the list `ml_d` as defined in clause 4.3.5. The entries for the dynamically discovered middleboxes in the discovered list shall have the `inserted` field set to "dynamic" and the `transparency` field set to "false". The client shall now reject further middleboxes proposed for inclusion as part of the new handshake.

If the client does not accept the middlebox with the proposed access rights, it should send an `Alert` of type `middlebox_authorization_failure` and the client should close the connection. In this case, the client may choose to include the proposed middlebox in the `MiddleboxList` of the TLMSP extension in future TLMSP session initiations with the inserted field set to "forbidden".

If the server proposed an additional, client-accepted non-transparent middlebox which topologically lies between the server and the middlebox which was immediately before the server in the client's initial proposal, the server could receive the second `ClientHello` over a new TCP connection. In this case, the server should use the `hs_id` (if required, extended by `client_address, server_address` from the TLMSP extension) to associate the new TCP-connection to the same TLMSP session (used to retrieve the correct hash-context for the handshake verification as defined in clause 4.3.9).

If the second `ClientHello` is received over a new TCP-connection, the second `ClientHello` could be processed by a new physical server. To allow the new physical server to take over handling of the session, the client shall (as defined in clause 4.3.5) include the `hs_id` and a hash state of the messages sent/received before this second `ClientHello`, in the TLMSP extension of the second hello. The new server will be able to determine that this new `ClientHello` is associated with dynamic discovery of middleboxes in an earlier session setup, since the second `ClientHello` contains both the `hs_id` and an additional middlebox list (`ml_d`) in the TLMSP extension as explained earlier in this clause. (The `hs_id` and `ml_d` are not present in an initial `ClientHello`).

The two paragraphs above shall apply also when replacing "the server" by "a middlebox", and the middlebox shall then follow the recommendation of clause 4.3.1 and not generate any TLMSP-messages of their own until after the discovery phase is done.

4.3.2.3.3 Transparent middleboxes

This clause applies to use cases involving middleboxes that are not individually visible/routable on the IP layer but which are still present on the client-server network path.

EXAMPLE: A middlebox function co-located with a default gateway, a firewall, or within a mobile operator core network is not visible on the IP layer but is present in the client-server path of communication.

NOTE 1: In principle it could be possible to also pre-configure certain transparent middleboxes similar to the way described clause 4.3.2.2, if their on-path presence is always guaranteed.

The middlebox is assumed to detect initialization of TLMSP handshakes passing through it, even if the handshake is not explicitly addressed to the middlebox. Thus the middlebox has opportunity to make its presence known without assistance from the server. The middlebox can propose its own inclusion by adding itself to the `MiddleboxList` extension of the `ClientHello`. This proposal may initially be done silently towards the client; the middlebox only forwards the modified `ClientHello` toward the server. This usually allows plural transparent middleboxes to add themselves to the same `ClientHello` as it propagates toward the server.

Thus, the client will be informed about all the dynamically added transparent middleboxes as it later receives the ServerHello. Both server and client may reject any or all of the transparent middleboxes.

A transparent middlebox may intercept the `ClientHello` (either between the client and the first middlebox, between two middleboxes, or, between the last middlebox and the server). If the intercepting middlebox wishes to propose its own addition, it shall add itself to the `MiddleboxList` of the client's TLMSP extension (the `ml_i` list as described in clause 4.3.5), assigning itself a unique entity identity, setting `transparency` to "true" and setting `inserted` to "dynamic". The middlebox inserts itself in the middlebox list according to topological order. The order should be deduced by observing the current value of `previous_entity_id` in the middlebox list, indicating the logical entity identity of the previous hop. A transparent middlebox that wishes to be included may send a `MboxHelloRequest`, as defined in clause 4.3.6.8, to the client to inform the client about, for example, its provided services. How to generate and use such information is outside the scope of the present document.

NOTE 2: This avoids the need for the middlebox to perform extensive (DNS) look-ups to find the previous entity's logical identifier and thus the correct topological placement. This holds in particular when IP address is not used as `address` of the middleboxes and may also avoid NAT issues.

The middlebox shall also include information about which contexts it seeks read/delete/write access to and forward the modified `ClientHello` toward the server (addressing it to the next-hop entity/middlebox).

When the server receives the (modified) `ClientHello`, it shall authorize all middleboxes, including transparent ones that made their presence known in the modified `MiddleboxList` as described in the present clause. All middleboxes shall be included in the `MiddleboxList` of the `ServerHello` extension, but those transparent middleboxes that were not authorized by the server shall have their `inserted` attribute set to "forbidden".

When the client receives the `ServerHello` response, it will be able to tell from the attribute fields of the middlebox list which transparent middleboxes are proposed and which ones the server accepts. The client shall decide whether to authorize the middleboxes that were accepted by the server. If so, the client shall proceed as in clause 4.3.2.2, now with the server's `hs_id` and two middlebox lists in the extension; an identical copy of the client's original `ml_i` and the second list `ml_d` also including the accepted middleboxes whose entries have the `inserted` field set to "dynamic" and the transparency field set to "true", as defined in clause 4.3.5. The client and server shall now ignore and reject further middleboxes that attempt to add themselves as part of the new handshake.

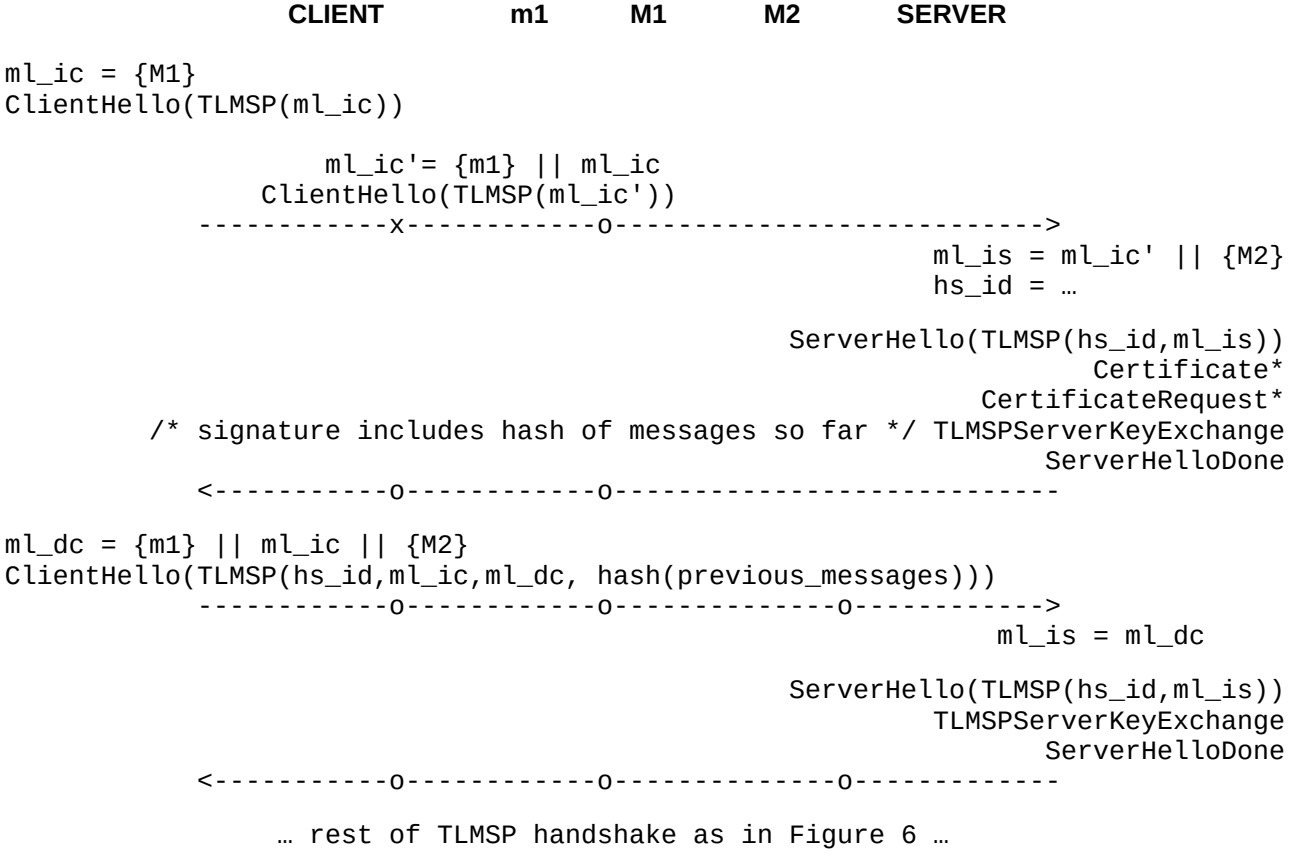
Otherwise, if the client does not accept the dynamically discovered middleboxes, it shall send an alert of type `middlebox_authorization_failure`.

NOTE 3: This way of handling additional middleboxes implies that the added middlebox remains (transparently) on-path for the duration of the session.

4.3.2.4 Combined discovery

4.3.2.4.1 Example use case

Figure 7 illustrates an example scenario with one middlebox (`M1`) pre-configured in the client, as defined in clause 4.3.2.2 and thus included in the initial `MiddleboxList`, `ml_ic`, of the TLMSP extension in the `ClientHello`. As the `ClientHello` traverses the network, a transparent middlebox, `m1`, detects the signalling and wishes to add itself to the TLMSP session. It does this adding itself to `ml_ic`, as defined in clause 4.3.2.3.3. With respect to Figure 7, `m1` adds itself to the list `ml_ic` before the middlebox `M1`. Additionally, when the server finally receives the `ClientHello`, it detects that a second, non-transparent middlebox, `M2`, is also desired, which is handled according to clause 4.3.2.3.2, i.e. `M2` is added to the list `ml_is`, just after the middlebox `M1`.



NOTE: All but the two last messages are not available to `M2`, because `M2` is not on the IP path between client and server.

Figure 7: Dynamic discovery example

The optional alerts and `MboxHelloRequest` are not shown in Figure 7. Setting attributes of the discovered middleboxes (i.e. `inserted` = "dynamic" and `transparent` = "true" or "false") is also omitted for simplicity.

Although a new key exchange by the server will become necessary in this case (since `M2` has not been in the path throughout the handshake), the server should still include certificate and key exchange, as it will give the client an opportunity to authenticate the server during the discovery.

4.3.2.4.2 Practical considerations

For dynamic discovery of transparent middleboxes to work in general, and particularly if used in combination with dynamically discovered non-transparent middleboxes, assumptions (or preferably knowledge) of the network topology are needed.

Suppose that in the example clause 4.3.2.4.1, `m1` lies topologically between `M1` and `M2`. While `m1` is transparently on path between `M1` and the server, `m1` could in general not be transparently present also on the path between `M1` and `M2`, which is the path followed on the second `ClientHello` and subsequent messages. Clearly, when `m1` attempts to add itself to the middlebox list, `m1` does not yet know that the server will change the IP routing path by adding the non-transparent middlebox `M2`. Regardless of whether `m1` is immediately after the client or immediately before the server, there exist cases when an additional non-transparent middlebox addition by the server (or by another middlebox) could remove `m1` from the subsequent signalling path.

Dynamic discovery by transparent middleboxes should therefore only take place when the middlebox has strong assurance that it will remain on path for the rest of the session. How such assurance is obtained is out of scope of the present document.

4.3.2.5 Middlebox leave and suspend

Middleboxes may find it necessary, e.g. due to processing load, to step down or step out of an ongoing session. A middlebox shall always notify other entities before doing so either by issuing one the TLMSP specific alert `middlebox_suspend_notify` (clause 4.4), or, the `MboxLeaveNotify` message (clause 4.3.8). These messages shall not be sent prior to handshake completion (all `Finished` and `MboxFinished` messages being verified).

4.3.3 Session resumption and renegotiation

4.3.3.1 Resumption

As with TLS 1.2, TLMSP provides an abbreviated handshake to resume a previously established session, refreshing the keys but keeping the previous cipher suite.

Similarly to TLS 1.2, the server may, in the initial handshake, indicate a session ID in its hello message, indicating to the client that the server may be willing to cache the session state for later resumption. (This session ID is generally not the same as the `hbh_id` which may be used in the TLMSP headers, or the `hs_id` assigned in the `ServerHello`.) In TLMSP, if resumption is enabled, the server shall allocate a session ID. Middleboxes shall obtain this session ID from the handshake signalling and associate it with the current session. This session ID could be non-unique among all the sessions that a middlebox is serving at once. Therefore, the middleboxes shall locally extend the session ID by a client identity and a server identity conditioned on that the triplets (session ID, client ID, server ID) becomes globally unique from the middlebox point of view. Any server ID, client ID that enables such unique identification may be used, and it is out of scope to specify details of the identity selection.

NOTE: Such client ID, server ID need to exist, otherwise the middlebox would confuse some TCP sessions passing through it.

When a client wishes to resume a session, the session ID is indicated by a client in its `ClientHello` with the server. If the server recognizes the provided session ID, it may choose to allow resumption. When allowing session resumption, the server shall signal the same (own) session ID back toward the client.

If a middlebox recognizes the session ID (in client's and server's hello) and is willing to resume the session, it shall indicate this by adding the same session ID in its hello toward the server and client, otherwise the middlebox's session ID shall be empty. Session resumption shall be performed if, and only if, the server and all middleboxes indicate the same session ID for resumption.

TLS also supports a (server-side) stateless resumption via session tickets, [3], if the client indicated support for session tickets via the ticket extension to the `ClientHello`. The client may therefore attempt to initiate resumption by including previously received tickets, in the handshake toward other entities. The client shall include the server-associated ticket in its `ClientHello`, whereas the middleboxes' tickets shall be included in the middlebox list extension. Each middlebox shall indicate toward the server, in the standard Hello extension, that it accepts the client's resumption proposal by copying the same ticket it received from the client when generating its hello messages toward the server. If the server received positive confirmation (tickets) from all middleboxes, the server may choose to proceed with resumption. During resumption, the client may also receive renewed tickets, which it may store for future resumptions of the same session.

Finally the client, the server and the set of middleboxes shall refresh keys as defined in clauses 4.3.10.4 and 4.3.10.5.

Middleboxes shall not be removed or added as part of resumption negotiation and resumption shall be done using the same contexts, cipher suite as the original session.

4.3.3.2 Renegotiation

In TLS 1.2, the client endpoint can initiate a renegotiation of the security parameters by sending a new `ClientHello`. A server endpoint can, in TLS 1.2, request renegotiation by sending a `HelloRequest`. The present document does not allow a corresponding renegotiation for TLMSP, for reasons laid out in clause E.7. A TLMSP endpoint receiving an indication to perform renegotiation shall issue an `unexpected_message` alert and should abort the connection.

4.3.4 Handshake message types

TLMSP employs the following `Handshake` message types:

```
enum {
  hello_request(0), client_hello(1), server_hello(2), certificate(11),
  server_key_exchange(12), certificate_request(13), server_hello_done(14),
  certificate_verify(15), client_key_exchange(16), finished(20), tlmsp_server_key_exchange(40),
  mbox_hello(41), mbox_certificate(42), mbox_certificate_request(43),
  certificate_2_mbox(44), mbox_key_exchange(45), mbox_hello_done(46),
  certificate_verify_2_mbox(47), tlmsp_key_material(48),
  tlmsp_key_conf(49), server_unsupport(50), mbox_hello_request(51), mbox_finished(52),
  tlmsp_delegate(53), mbox_leave_notify(54), mbox_leave_ack(55), mbox_auth_request(56),
  mbox_auth_response(57), (255)
} HandshakeType;
```

For the messages `hello_request(0)`, `client_hello(1)`, `server_hello(2)`, `certificate(11)`, `server_key_exchange(12)`, `certificate_request(13)`, `server_hello_done(14)`, `certificate_verify(15)`, `client_key_exchange(16)`, and `finished(20)`, clause 7.4 of IETF RFC 5246 [1] shall apply.

In addition:

- the `client_hello` and `server_hello` messages shall support the extensions defined in clause 4.3.5;
- the hash computation in the `Finished` messages shall be computed as defined in clause 4.3.9.

The present document defines the following new `Handshake` messages: `tlmsp_server_key_exchange(40)`, `mbox_hello(41)`, `mbox_certificate(42)`, `mbox_certificate_request(43)`, `certificate_2_mbox(44)`, `mbox_key_exchange(45)`, `mbox_hello_done(46)`, `certificate_verify_2_mbox(47)`, `tlmsp_key_material(48)`, `tlmsp_key_conf(49)`, `server_unsupport(50)`, `mbox_hello_request(51)`, `mbox_finished(52)`, `tlmsp_delegate(53)`, `mbox_leave_notify(54)`, and `mbox_leave_ack(55)`.

The `tlmsp_delegate` message and usage is described in clause C.2.3.2. Use of `mbox_auth_request` and `mbox_auth_response` is defined in clause C.3. Details of the other new `Handshake` messages and extensions thereto are provided in clauses 4.3.5 to 4.3.9.

4.3.5 TLMSP Handshake extensions

Recall that a TLS extension is defined in [2] as:

```
enum {
    server_name(0), … , (65535)
} ExtensionType;

struct {
   ExtensionType extention_type;
   opaque extension_data<0..2^16-1>;
} Extension;
```

TLMSP defines three new TLS handshake extensions to the `Hello` messages, the first in the form of a basic `TLMSP` extension with `extension_type = 0x24`. The other extensions are defined in clause C.2.3. The extension shall contain a version indication according to [2], a list of middleboxes, and information related to the TLMSP session being negotiated.

For the entities in the list of middleboxes, the `entity_id` values `0x00`, `0x01`, `0xfe`, and `0xff` are reserved with `0x01` reserved for the client and `0xfe` reserved for the server. Values 0x00 and 0xff are reserved for other purposes. A middlebox may be assigned any value in the range `0x02-0xfd`. The list of middleboxes shall be ordered by the network topology order of the connections established from client to server.

The format for the entries in the `TLMSP` extension and the associated `MiddleboxList` shall be as follows:

First, each entity (middlebox or endpoint) shall be identified by an Address value composed of a primary component and sometimes a secondary component, the format and use of which depend on the type as follows:

- `uri - primary` shall be a URI per [12] with a non-empty authority component. `secondary` shall not be present.
- `fqdn - primary` shall be a fully qualified domain name as defined in [13], using the syntax defined in section 2.1 of [14]. When using TCP, `secondary` shall be a two octet TCP port number per [15], otherwise it shall not be present.
- `ipv4_adr - primary` shall be a four octet destination address as defined by [16]. When using TCP, `secondary` shall be a two octet TCP port number per [15], otherwise it shall not be present.
- `ipv6_adr - primary` shall be a 16 octet destination address as defined by [17]. When using TCP, `secondary` shall be a two octet TCP port number per [15], otherwise it shall not be present.
- `mac_adr - primary` shall be a six octet MAC address as defined by [18]. `secondary` shall not be present.

```
enum { tcp } TransportProtocol;

struct {
    enum { uri(0), fqdn(1), ipv4_adr(2), ipv6_adr(3), mac_adr(4), (255) } type;
    select (type) {
       case uri: opaque primary<5..2^16-1>;
       case fqdn: struct {
                      opaque primary<3..253>;
                      select (TransportProtocol) {
                         case tcp: opaque secondary[2];
                      };
                  };
       case ipv4_adr: struct {
                          opaque primary[4];
                          select (TransportProtocol) {
                             case tcp: opaque secondary[2];
                          };
                      };
       case ipv6_adr: struct {
                          opaque primary[16];
                          select (TransportProtocol) {
                             case tcp: opaque secondary[2];
                          };
                      };
       case mac_adr: opaque primary[6];
    };
```

```
} Address;

opaque HandshakeID[4];

struct {
    struct {
      uint8 major;
      uint8 minor;
    } tlmsp_version;
    CipherSuite tlmsp_cipher_suites<2..2^16-2>;
    enum { false(0) , true(1), (255) } server_anon;
    HopID hbh_id;
    select (is_server_hello) {
       case true: struct {
                      HandshakeID hs_id;
                      SignatureAndHashAlgorithm supported_sig_algs<2..2^16-2>;
                  };
       case false: struct {
                       uint8 previous_entity_id;
                       enum { false(0) , true(1), (255) } discovery_ack;
                       select (discovery_ack) {
                          case true: struct {
                                        HandshakeID hs_id;
                                        opaque pre_discovery<1..255>;
                                        MiddleboxList ml_d;
                                     };
                          case false: struct { };
                       };
                  };
    };
    Address client_address, server address;
    enum { false(0) , true(1), (255) } is_client_resumption_req;
    select (is_client_resumption_req) {
       case true: HandshakeID hs_id;
       case false: ContextList cL;
    };
    MiddleboxList ml_i;
} TLMSP;
```

The `tlmsp_version` has no direct relation to the `version` field of the TLMSP record header of Figure 2. When initiating the handshake, the `version` field of the TLMSP record header indicates which version of TLS serves as the base specification from which the current version of TLMSP is derived, and thus also indicates which version of TLS to fallback to, in case TLMSP is not supported. The `tlmps_version` in the extension indicates the requested version of TLMSP. The value `tlmsp_version = {1, 0}` shall be used for the current version of TLMSP. The (possibly different) values of `tlmsp_version` in the extension carried in the `ClientHello` and the `ServerHello` shall be used for TLMSP version negotiation in the same way as the `version` field of the record header is used by TLS for version negotiation as defined in [2]. Since the fixed data in the TLMSP extention (including length indicators of variable length fields) consists of at least 38 octets and the maximum size of a TLS extension is 2^16-1, this leaves at most 65497 octets for any variable length fields.

The client shall include its support for TLMSP-specific cipher suites in the field `tlmsp_cipher_suites`, which shall follow the same format as defined in [2]. The value `server_anon` shall be used by the client to signal if it is willing to accept connections in which the server does not authenticate.

NOTE 1: This field applies only to the server and serves the same purpose as the set of anon cipher suites in TLS, but without the need to define a specific separate anon cipher suite for each authenticated cipher suite.

The possibility for middleboxes to skip authentication is also supported but handled via the middlebox list, as defined below.

The server shall use the `tlmsp_cipher_suites` and `server_anon` field to indicate the selected TLMSP cipher suite in the `ServerHello`. The client and server shall also include their support for standard TLS cipher suites in the normal way, as part of the hello message body (outside the extension field), to allow TLS fallback as defined in annex B. The currently defined TLMSP cipher suites are found in annexes A and B.

The value `previous_entity_id` shall be used to indicate to the next-hop-entity from which entity (client or middlebox) an inbound `ClientHello` is being forwarded, as described in clause 4.3.2.3.

When a TLMSP connection is first attempted, the client shall set the field `discovery_ack` to `false`. Only the first middlebox list `ml_i` shall be present and shall include middleboxes already known to the client. During such initial handshake, additional middleboxes may be dynamically discovered as described in clause 4.3.2.3. No `hash` value shall be included.

When the new handshake following the discovery is initiated by the client, the client shall set the field `discovery_ack` to `true` and both the original list `ml_i` and the complete list of all authorized middleboxes `ml_d` shall be included. The server shall in the `ServerHello` include the received `ml_d` list as its own `ml_i` list in its corresponding response. The value `pre_discovery` shall also be present in the second `ClientHello` following dynamically discovered middlebox(es). The `pre_discovery` field shall contain the hash of all the messages sent/received between client and server up to, but not including, this second `ClientHello`, see clause 4.3.9.4 for details. The field is defined as variable length to limit the need to maintain state at server between first and second `ClientHello` (the field size otherwise depends on the proposed cipher suite).

Each entry in the middlebox list specifies the middlebox's address, a unique ID, and a list of contexts with the corresponding access privilege level. For contexts not present in the list, the privilege level is "none". As `Application` protocol containers (including deletion indication containers) cannot appear in context zero, middleboxes shall not be assigned the "delete" access privilege level for context zero. The list also contains proposed authentication methods that the endpoints propose to use with each middlebox. The ticket shall be included if the client seeks to resume a previous session based on a previously received ticket.

```
struct {                            /* middlebox identification */
    uint8 entity_id;
    Address address;
} Middlebox;

struct {
    uint8 context_id;                                 /* ID of context */
    enum { none(0), read(1), delete(2), write(3), (255) } authorization; /* privilege level */
} ContextAccess;

struct {
     enum { anon(0), psk(1), gba(2), (255) } method_id; /* alt. key ex. method, see annex B */
     opaque credential_hint<0..2^16-1>;        /* hint to identity of the credential (psk) to use */
     enum { false(0), true(1), (255) } use_certificate;  /* true if and only if the middlebox
                                                            is expected to authenticate itself
                                                            using a cerificate to other endpoint */
} MboxAlternativeCipherSuite;

struct {
    Middlebox middlebox;                    /* middlebox identification */
    enum { static(0), dynamic(1), forbidden(2), (255) } inserted;
    enum { false(0), true(1), (255) } transparency; /* is the middlebox transparent or not */
    opaque ticket<0..2^16-1>;           /* used during session resumption with tickets */
    select (is_client_resumption_req) {
       case true: struct {};            /* resume always use the same contexts and accesses  */
       case false:
             struct {
                 uint8 n_contexts;      /* number of contexts for this middlebox */
                 ContextAccess contexts[2*n_contexts]; /* list of contexts for this middlebox */
                 enum { standard(0), alternative(1), (255) } cipher_suite_options; /* see text */
                 select (cipher_suite_options) {
                    case alternative: MboxAlternativeCipherSuite alt_cs;
                    case standard: struct { };
                 };
             };
    };
} MiddleboxInfo;

MiddleboxInfo MiddleboxList<0..2^16-1>;
```

The (possibly empty) `ticket` shall be used as defined in IETF RFC 5077 [2]. The field `inserted` is used to distinguish between middleboxes that are statically pre-configured or added dynamically during the handshake. The field may also be used to prevent "black-listed" middleboxes from being dynamically added. The truth value of `resumption_attempt` may be established based on the presence of a session `ticket`, or on the presence of a session ID in the `ClientHello`.

Using the field `cipher_suite_options`, the endpoints shall signal to each middlebox whether that middlebox should use the standard cipher suites as defined in annex A, or, whether the middlebox should use the alternative cipher suites as defined in annex B. The difference between the standard and alternative cipher suites are only related to the key exchange and authentication method.

The client may further use the field `use_certificate` of the `alt_cs` field to instruct the middlebox whether it should present and authenticate itself using a certificate to entities located downstream of the middlebox (in the direction of the server, including the server itself), or, to only be implicitly authenticated. Implicit authentication means that such downstream entities are assumed to trust the client and that the client will properly authenticate the middlebox. This implicit authentication should only be used when such trust exists, and, when the downstream entities can authenticate the client. The server or any other entity may reject such proposal and terminate the connection. Middleboxes who are not explicitly instructed to not provide their certificates shall provide them according to standard procedures. The client shall include preferences about alternative middlebox cipher suites in the list `ml_i` included in the client's TLMSP middlebox list extension.

The above shall apply, mutatis mutandi, also when the server responds and provides its own middlebox list extension toward the client, see below. If both the client and the server simultaneously signals to a specific middlebox to not use a certificate in either direction, the alternative cipher suite used with that middlebox shall provide built-in authentication of the middlebox, e.g. through the use of pre-shared keys or similar mechanism.

The values chosen by the client and server for `cipher_suite_options` and `use_certificate` are made independently by the client and server endpoints and shall apply only to the point-to-point security configuration between the endpoint and the middlebox in question. That is, the pairwise keys between pairs of middleboxes continue to use the key exchange mechanism of the standard cipher suite regardless of the value of `cipher_suite_options`. If for a given middlebox, an endpoint sets `cipher_suite_options` to `alt_cs` alternative and sets `method_id` to anon, the middlebox will not provide a certificate to that endpoint. This implies that pairwise key exchange between that middlebox and any other middleboxes between it and the other endpoint will also not be authenticated. Likewise, if for a given middlebox, an endpoint sets `use_certificate` to false, the middlebox will not provide a certificate to the other endpoint, so the pairwise key exchange between that middlebox and any other middleboxes between it and the other endpoint will not be authenticated.

EXAMPLE 1: For a specific middlebox, the client could set use_certificate = false and cipher_suite_options = alternative, while the server, for the same middlebox, sets use_certificate = true and cipher_suite_options = standard. This will not cause interoperability problems, see annex B.

NOTE 2: If, for example, the client instructs the first middlebox to not present its certificate to downstream entities, this implies that no downstream entity will be able to authenticate the first middlebox. In this case, mere trust in the client, and that the client properly authenticates the first middlebox could provide insufficient assurance unless the client authenticates itself to all downstream entities. A converse scenario applies when the server instructs a middlebox to not authenticate itself toward downstream entities (in the server-to-client direction).

The `hbh_id` in the `ServerHello` and `ClientHello` `TLMSP` extensions may be used by entities to assign a value of the `hbh_id` that the entity wishes to use in the header field of its outbound TLMSP messages, in both directions. If an entity wishes to use a `hbh_id`, it shall ensure that any non-NULL `hbh_id` selected is unique among concurrent active TLMSP sessions in which the entity is using the same transport connection.

EXAMPLE 2: An entity, `e`, participating in two TLMSP sessions s1, s2, has the same downstream entity `e'` (using the same IP address and port numbers) in the client-server direction in both sessions (i.e. `e` is using the same transport connection). Then `e` selects distinct values to include in the `hbh_id` of the `ClientHello` in the handshake exchanges of s1 and s2. Alternatively `e` can select a NULL `hbh_id` in at least one of the two sessions.

The client may originally assign any value to the `hbh_id` field. The entity on each hop shall record a `hbh_id` received in the `ClientHello` from the upstream neighbour and shall replace the value in the `ClientHello` extension with an own selected value (possibly, a NULL value, if the entity does not choose to use any `hbh_id`) as it forwards the `ClientHello` toward the server. As the `ServerHello` is then forwarded toward the client, each entity shall record `hbh_id` received from the upstream neighbour, and when forwarding the `ServerHello`, includes the same `hbh_id` in the server-client direction that it previously chose in the client-server direction (conditioned by the uniqueness requirements) by including it in the `hbh_id` field as it forwards the `ServerHello`. All entities shall support the multiplexing of multiple TLMSP sessions on a single transport connection.

The server shall include in its `ServerHello` response the list `ml_i` of all middleboxes that it received via the `ClientHello` (i.e. including additional middleboxes that have been added dynamically in-band as defined in clause 4.3.2), and furthermore extended by any middlebox requested for addition by the server (see also clause 4.3.2). The `ServerHello` shall also contain the server-assigned handshake ID, `hs_id`, and the server's supported signature algorithms. The `hs_id` is useful for identifying a new TCP connection to the server, following dynamic discovery of middleboxes as described in clause 4.3.2.3. The `supported_sig_algs` list may be used by middleboxes to deduce which certificate(s) to present: the middlebox already knows the client's support (from default values or extensions to `ClientHello`) but only knows the single server-supported algorithm indicated by the server's certificate. Thus, this information improves the likelihood of the middlebox being able to select an appropriate certificate/algorithm. If a middlebox does not support any of the client/server indicated algorithms, it shall send an alert of type `handshake_failure` at the point where the middlebox would otherwise send its certificate. The server shall also includes its preferences regarding alternative cipher suites in the list `ml_d`.

The second component of the extension to the `Hello` is a list of the context IDs and descriptions. A context description comprises a `purpose` string meaningful only to the application; TLMSP does not use it.

EXAMPLE 3: A `purpose` string could have the value "malware removal service" for a middlebox performing malware removal.

```
struct {
    uint8 context_id;
    enum { unconfirmed(0), audit_info(1), audit_trail(2), (255)} audit;
    opaque purpose<0..255>;
} ContextDescription;

ContextDescription ContextList<3..2^16-1>;
```

The `ContextList` shall not include an entry for the reserved context with `context_id = 0`.

The `audit` field is used to request confirmation from middleboxes that the containers associated with the context have traversed via them, allowing them to act on the content (if authorized). The values `audit_info` and `audit_trail` enable the production of audit containers as described in clause 4.2.3.1.5. The value `unconfirmed` indicates that all entities shall not insert any audit containers for the associated context. Additional values of `audit` are intended to be defined in the future, specifying that middleboxes add information on their processing to the audit containers.

The third and fourth extensions are related to the TLMSP proxying and their usage is described in clause C.2 of the present document.

TLMSP puts no restrictions on which port number to use.

Further (optional) TLMSP-related extensions are defined in annexes B and C.

4.3.6 Middlebox related messages

4.3.6.1 MboxHello

The `MboxHello` message shall be structured as follows:

```
struct {
    uint8 mbox_entity_id;
    ProtocolVersion client_version;
    Random client_mboxhello_random, server_mboxhello_random;
    SessionID session_id;
    select (extensions_present) {
       case false: struct {};
       case true: Extension extensions<0..2^16-1>;
     };
} MboxHello;
```

The MboxHello is identical to a TLS 1.2 Hello, except for the inclusion of the `mbox_entity_id`, the two random parameters, `client_mboxhello_random` and `server_mboxhello_random`, and the two alternative cipher suite fields. The message excludes cipher suites and compression methods (since compression is not supported and selection of cipher suite is made by the server, before the `MboxHello` is sent). Middleboxes shall provide the same content in their MboxHello directed to client and server. The two random values shall be selected (pseudo)randomly and independently. The client shall use the `client_mboxhello_random` to generate master keys shared with the middlebox, whereas the server shall use the `server_mboxhello_random` to generate master keys shared with the middlebox. If a middlebox does not support, or does not approve the proposed alternative cipher suite, it should raise an `unsupported_extension` alert.

This message shall always be forwarded by middleboxes.

4.3.6.2 MboxCertificate

As is shown in Figure 6, unless the server has set the extension field `use_certificate` to false, this message shall be sent from middlebox to client when the middlebox has received the `ServerHelloDone`. The message shall, unless the client specified had set the extension field `use_certificate` to false, be simultaneously sent from the middlebox back to the server. The `MboxCertificate` shall have identical content when sent to both the server and to the client.

NOTE: This is identical in format to a server's Certificate message, but with an added entity identity field of the middlebox.

This message shall have the following structure:

```
struct {
    uint8 mbox_entity_id;
    Certificate cert;
} MboxCertificate;
```

The field `cert` shall be formatted as the `Certificate` message in clause 7.4.2 of IETF RFC 5246 [1].

The middlebox shall set `mbox_entity_id` to the `mbox_entity_id` value found in the received `ServerHello` message. The message shall always be forwarded by other middleboxes.

4.3.6.3 MboxCertificateRequest

This message shall have the following structure:

```
struct {
    uint8 mbox_entity_id;
    CertificateRequest cr;
} MboxCertificateRequest;
```

This message shall be sent by a middlebox to the client when the middlebox wishes to authenticate the client and shall always be forwarded by other middleboxes.

4.3.6.4 Certificate2Mbox

This message shall have the following structure which is identical to the `MboxCertificate` message defined in clause 4.3.6.2:

```
struct {
    uint8 mbox_entity_id;
    Certificate cert;
} Certificate2Mbox;
```

If, and only if, the signature verification by the client as defined in clause 4.3.10.1 is successful, this message shall be sent by a client in response to a received `MboxCertificateRequest` from a middlebox with identity `mbox_entity_id`. A middlebox receiving this message shall always forward it, unless the middlebox is the intended receiver.

4.3.6.5 MboxKeyExchange

This message shall have the following structure:

```
struct {
    uint8 mbox_entity_id;
    TLMSPServerKeyExchange client_exch;  /* key exchange middlebox <-> client */
    TLMSPServerKeyExchange server_exch;  /* key exchange middlebox <-> server */
} MboxKeyExchange;
```

The security relevant parameters of `client_exch` and `server_exch` shall be generated independently, but identical copies of this message shall be sent to both server and client. If the client has not requested an alternative cipher suite, or has requested the alternative cipher suite `anon`, the client shall use only the `client_exch` element to establish the shared pre-master secret. If the client has requested alternative cipher suite other than `anon`, the middlebox shall still provide a `client_exch` component, but the client and the middlebox shall ignore it when generating their pairwise master key. In this case, the middlebox closest to the client may populate the `client_exch` field with a correctly formatted dummy value. The `client_exch` component shall however be used by other middleboxes situated between the middlebox in question and the client when generating master key between the corresponding pair of middleboxes. If the client has requested an alternative cipher suite with `method_id = anon`, or, the server has requested `use_certificate = false`, neither the client, nor any middlebox situated between the middlebox in question and the client will be able to verify the authenticity based on the `MboxKeyExchange` message itself.

The paragraph above shall apply also when substituting "client" with "server", `client_exch` with `server_exch`.

The entire message (including both elements) shall be included by both client and server when computing the `Finished` hash. `TLMSPServerKeyExchange` is defined in clause 4.3.10.1.

The `dh_p` and `dh_g` parameters of the `ServerDHParams` in the contained `ServerKeyExchange` and `TLMSPServerKeyExchange` structures shall be identical to those in `ServerKeyExchange` message received earlier from the server, and they shall be ignored by the endpoints upon receipt. This message shall always be forwarded by middleboxes.

If the middlebox, via the `MboxHello` of clause 4.3.6.1, has accepted one or both endpoint's suggested use of alternative cipher suites according to annex B, the part of the message directed to that endpoint shall be ignored by the endpoint, except for the purpose of generating the `Finished` verification message. Non-endpoint entities, e.g. other middleboxes located between the sender middlebox (generating the `MboxKeyExchange`) and the endpoint shall use the `MboxKeyExchange` information in the standard way, to generate shared keys with the sender middlebox, except that authentication of the parameters will not be possible depending on the settings of `use_certficate` requested by the endpoints.

4.3.6.6 MboxHelloDone

This is identical in format to a `ServerHelloDone` of IETF RFC 5246 [1], but with an added identity field of the middlebox. This message shall have the following structure:

```
struct {
    uint8 mbox_entity_id;
    ServerHelloDone hd;
} MboxHelloDone;
```

This message shall have identical content to the server and to the client and shall always be forwarded by middleboxes.

4.3.6.7 CertificateVerify2Mbox

This message shall have the following structure:

```
struct {
    uint8 mbox_entity_id;
    CertificateVerify cv;
} CertificateVerify2Mbox;
```

This message shall be sent following a `Certificate2Mbox` as defined in clause 4.3.6.4 that is sent to a middlebox with the stated `mbox_entity_id`. It allows that middlebox to verify the client. A middlebox receiving this message shall always forward it, unless the middlebox is the intended receiver.

4.3.6.8 MboxHelloRequest

This message may be sent in response to a `ClientHello` by a transparent middlebox that has added itself to the `MiddleboxList` in the `ClientHello`. The message format shall have the following structure:

```
struct {
    uint8 mbox_entity_id;
    MiddleboxInfo mb_info;     /* information about the to-be-added middlebox */
} MboxHelloRequest;
```

The sub-fields of field `mb_info` may be used for information about the reason for why and how the middlebox is to be added (which contexts it wants access to and the purpose). The `cipher_suite_options` field may be used to indicate preference for alternative cipher suites. A middlebox receiving this message shall always forward it.

4.3.6.9 ServerUnsupport

This message shall be used by the first middlebox to detect that the server does not support TLMSP, i.e. the middlebox located closest to the server and shall be sent from that middlebox towards the client.

```
struct {
    uint8 entity_id;               /* the identity of the middlebox originating the message */
    MiddleboxList middlebox_info;  /* list of middleboxes */
} ServerUnspport;
```

`The middlebox_info` field shall contain the complete list of middleboxes that the originating middlebox previously forwarded to the server, i.e. including any dynamically discovered middleboxes. Other middleboxes shall forward this message to the client.

4.3.6.10 MboxFinished

A special handshake verification message is defined, used only between an endpoint and a middlebox. It shall have the format:

```
struct {
    uint8 entity_id;
    opaque verify_data[verify_data_length];
} MboxFinished;
```

`entity_id` shall be the entity identity of the origin/destination middlebox.

`verify_data` shall be formatted as specified in clause 7.4.9 of IETF RFC 5246 [1] with the deviations for the hash computation as specified in clause 4.3.9.3.

NOTE: Different data is included in hash computations for the "standard" `Finished` message sent between endpoints than for the `MboxFinished` message, since not all parties have access to the complete set of messages exchanged during the handshake.

When a middlebox receives an `MboxFinished` message destined to it, after validating the included `verify_data`, it places the `verify_data` for the downstream endpoint into the record at the same location, applies the record protection, and forwards the record to the next entity downstream. No other modifications shall be made to the received record.

4.3.7 TLMSPKeyMaterial and TLMSPKeyConf

4.3.7.1 KeyMaterialContribution

The following described message generation method shall be the used when generating `TLMSPKeyMaterial` and when generating `TLMSPKeyConf` messages. The only difference between the two types of messages is that the messages shall have different `type` identifiers and the `content` field shall be generated differently as described below.

With reference to Figure 6, during an initial handshake, the `TLMSPKeyMaterial` and `TLMSPKeyConf` messages occur before the `ChangeCipherSpec` message, which will activate record protection. The content payload of each `TLMSPKeyMaterial` and `TLMSPKeyConf` message shall however still be protected as described in the present clause, using the keys established between only the endpoint and the receiving entity, and using the same encryption and integrity check mechanism that is being agreed during the ongoing handshake, see clause 4.3.1 of the present document and clause 7.4 of IETF RFC 5246 [1].

If record size extensions of IETF RFC 8449 [7] is being negotiated, the new record sizes shall be applied to the `TLMSPKeyMaterial` and `TLMSPKeyConf` messages, even though they occur before the `ChangeCipherSpec` message (which would otherwise activate usage of the new record sizes).

```
select (SecurityParameters.cipher_type) {
   case stream: StreamCipherContribution;
   case block:  BlockCipherContribution;
   case aead:   AEADCipherContribution;
} KeyMaterialContribution;
```

In the structures below, the `contributions` field shall be comprised of the sequence of contributions for all contexts to which the middlebox has at least read access (thus, a `reader_contrib` shall be present and a `writer_contrib` or `delete_contrib` may be present for each context). Specifically, a contribution shall have the following format, where `key_length` is equal to `SecurityParameters.enc_key_length`:

```
struct {
    uint8 context_id;
    opaque reader_contrib<0..key_length>;  /* zero length if no read access granted */
    opaque deleter_contrib<0..key_length>; /* zero length if no delete access granted */
    opaque writer_contrib<0..key_length>;  /* zero length if no write access granted */
} Contribution;
```

The maximum value of `SecurityParameters.enc_key_length` supported by TLMSP shall be 2^{16}-1 octets.

Below, for a given entity `i`, (the intended recipient of the message) the value `n_rctxt` shall equal the number of contexts to which `i` is granted the read access privilege level, `n_dctxt` shall equal the number of contexts to which `i` is granted the delete access privilege level, and `n_wctxt` similarly shall equal the number of contexts to which `i` is granted the write access privilege level. Finally, `n_ctxt` shall equal the number of contexts to which `i` is granted an access privilege level greater than "none". The `entity_id` shall be the entity identity, `i`, of the entity (middlebox or endpoint) to which the message is directed and shall be left unencrypted.

NOTE 1: When a `deleter_contrib` is present, also a `reader_contrib` will be present, and when `writer_contrib` is present, both a `reader_contrib` and a `deleter_contrib` will be present.

NOTE 2: Although deleter MACs are never used in context zero, as write access implies delete access generally, `deleter_contrib` is present for context zero in order to maintain a regular approach across all contexts.

```
struct {
    uint8 entity_id;
    opaque IV[SecurityParameters.record_iv_length];
    stream-ciphered struct {
        Contribution contributions[n_ctxt + (n_rctxt + 2*n_dctxt + 3*n_wctxt) * (2+key_length)];
        opaque mac[SecurityParameters.mac_length];
    };
} StreamCipherContribution;

struct {
    uint8 entity_id;
    opaque IV[SecurityParameters.record_iv_length];
```

```
    block-ciphered struct {
       Contribution contributions[n_ctxt + (n_rctxt + 2*n_dctxt + 3*n_wctxt) * (2+key_length)];
       opaque mac[SecurityParameters.mac_length];
       uint8 padding[BlockCipherContribution.padding_length];
       uint8 padding_length;
    };
} BlockCipherContribution;

struct {
    uint8 entity_id;
    opaque nonce_explicit[SecurityParameters.record_iv_length];
    aead-ciphered struct {
       Contribution contributions[n_ctxt + (n_rctxt + 2*n_wdtxt + 3*n_wctxt) * (2+key_length) +
                                  D + SecurityParameters.mac_length];
    };
} AEADCipherContribution;
```

where `key_length` is equal to `SecurityParameters.enc_key_length`, and the value `D` corresponds to padding and other overhead added by the AEAD transform in use.

For AEAD transforms, the AAD shall be defined as:

`AAD = entity_id || nonce_explicit || seq_num`,

where, as defined in clause 4.2.2.3, `seq_num = 264-1`. A contribution received from the server granting read (delete or write) access to `context_id = i` is hereinafter notationally described as `server_reader_contrib_i` (and `server_deleter_contrib_i`, `server_writer_contrib_i`). Similarly, a contribution received from the client granting read (delete or write) access to `context_id = i` is in the sequel denoted `client_reader_contrib_i` (and `client_deleter_ontrib_i`, `client_writer_contrib_i`). The contributions from client and server shall be combined according to clause 4.3.10.5 into `reader_key_block_i`, `deleter_key_block_i`, and `writer_key_block_i`. From these combined blocks, the actual data protection keys (in each of the two directions) shall be derived as defined also in clause 4.3.10.5.

For messages that contain an explicit MAC (i.e. non-AEAD contributions), the MAC of the message shall be calculated as:

```
    MAC(e1_to_e2_mac_key, KeyMaterialContribution.entity_id || seq_num ||
                          KeyMaterialContribution.IV ||
                          KeyMaterialContribution.contributions)
```

with `seq_num` as above and where `e1_to_e2_mac_key` is the MAC key shared between endpoint `e1` and middlebox (or other endpoint) `e2` derived according to clause 4.3.10.4. For encryption of the messages defined in this clause, the shared key `e1_to_e2_encryption_key` generated as in clause 4.3.10.4, shall be used.

4.3.7.2 TLMSPKeyMaterial

TLMSP introduces a new `Handshake` message for delivering context key material to the middleboxes. During the handshake, both the client and server shall send `TLMSPKeyMaterial` messages through the chain of all middleboxes, providing key shares for each middlebox (and the other endpoint). The message contains, for each context, a partial secret for each access right granted to a middlebox for that context. At least one message (for context zero) shall always be present. The final keys used to protect the context(s) can be derived only with both partial secrets (from the client and from the server); knowledge of only one partial secret in isolation does not reveal any knowledge of the context protection keys. Each `TLMSPKeyMaterial` message shall be generated by an endpoint e (server or client) using the defined data formats of clause 4.3.7.1, populated by parameters computed as defined in the sequel of the present clause.

Individual `TLMSPKeyMaterial` messages shall be formatted in the same way as `KeyMaterialContribution`, defined in clause 4.3.7.1.

NOTE 1: All `TLMSPKeyMaterial` and `TLMSPKeyConf` use the same fixed sequence number. This is not a security problem since there will be at most one such message processed by any given cryptographic key.

The entity_id field shall be set to the receiving middlebox and the context_id of each part of the contribution shall be set to the context to which the contribution pertains. The value key_length shall be identical to SecurityParameters.enc_key_length and the applicable reader_contrib, deleter_contrib, or writer_contrib field(s) shall be randomly generated using a cryptographically strong method. The deleter_contrib and writer_contrib shall be cryptographically independent from each other and from the reader_contrib.

NOTE 2: Each transferred contribution has the same size as the final desired key length. Thus, when the two parts from both client and server are combined, the resulting effective key length is sufficient for full entropy of both encryption and MAC keys.

The endpoints may use "piggy-backing" as defined in clause 4.3.1.2 to transmit TLMSPKeyMaterial information elements directed to several middleboxes in the same TLMSP record.

4.3.7.3 TLMSPKeyConf

As shown in Figure 6, the TLMSPKeyConf (Key Confirmation) message shall be generated and sent by the middleboxes as they receive TLMSPKeyMaterial signalling from the client towards the server, and likewise for the other direction. The TLMSPKeyConf message provides proof to the client and server that each middlebox has successfully obtained correct (partial) key material from the other endpoint for all contexts to which the middlebox is granted access. For the client, the receipt of one or more TLMSPKeyConf messages also explicitly proves that the client's own key material contributions were correctly received by the server (the server obtains this confirmation implicitly, see below).

The TLMSPKeyConf message shall be structured as the KeyMaterialContribution message (defined in clause 4.3.7.1). That is, the same message fields shall be used, but with different usage and semantics as defined below. An entity determines whether a message contains TLMSPKeyConf or TLMSPKeyMaterial by using the included message type (as defined in clause 4.3.4).

For each received TLMSPKeyMaterial message, MK, directed to the middlebox, exactly one TLMSPKeyConf message, MC, shall be generated as follows by the middlebox.

The middlebox shall set the entity_id field of MC to its own identity.

The middlebox shall further generate the contributions field of MC, where the to-be-protected payload is either:

- the entire decrypted contributions field of the MK message received from the client, when forwarding the message MC in client-to-server direction; or
- the entire decrypted contributions field of the MK message received from the server, when forwarding the message MC in the server-to-client direction.

The IV, MAC, and other fields as defined in clause 4.3.7.1 shall be generated according to the selected cipher suite. The symmetric key shared with the destination endpoint, as determined according to clause 4.3.10.4, shall be used.

The newly generated TLMSPKeyConf message MC shall then replace the corresponding received TLMSPKeyMaterial message MK when forwarding the message towards the destination. Any additional TLMSPKeyMaterial messages (not directed towards this middlebox, which is detectable by the entity_id field) shall be forwarded without further processing/action.

EXAMPLE: An original (complete) set of messages that was sent from an endpoint source that initially contained TLMSPKeyMaterial shares for middleboxes e[1], e[2], ..., e[N] (in network topological order) and e' (the destination endpoint, server or client), after processing by middlebox e[j] contains the TLMSPKeyMaterial for middleboxes e[j+1], e[j+2], ..., e[N], and destination e', and, in addition, TLMSPKeyConf messages from middleboxes e[1], e[2], ..., e[j], directed to e'.

When the destination endpoint ultimately receives the single `TLMSPKeyMaterial` message (from the other endpoint) and the set of `TLMSPKeyConf` messages, it shall verify that it received `TLMSPKeyConf` messages from all middleboxes, and for each context for which access to that middlebox was granted. The receiving endpoint shall then decrypt, verify integrity, and finally confirm that each of the retrieved decrypted secrets matches with the expected value. This confirmation shall be done as follows, depending on the endpoint in question:

- the *server* shall verify that all the secret(s) of all the contexts (`reader_contrib`, and `delete_contrib` or `writer_contrib`) of `MC` is equal to the corresponding values of the client's share as received directly from the client (in its own, separate `TLMSPKeyMaterial` message);
- the *client* shall verify that all the secret(s) of all the contexts (`reader_contrib`, and `delete_contrib` or `writer_contrib`) is equal to the server's share(s) as received directly from the server in the `TLMSPKeyMaterial` message.

If any of these checks fail, the endpoint shall send an alert of type `middlebox_key_confirmation_fault` and shall abort the handshake. The event should be logged.

NOTE: The above provides *explicit* confirmation to the *client* that all middleboxes received both contributions from the client itself and from the server. The *server* obtains an explicit verification of the contributions from the client and will later obtain an implicit confirmation of its own key contributions via the middleboxes' `MboxFinished` messages, see clause 4.3.9.3.

4.3.8 MboxLeaveNotify and MboxLeaveAck

4.3.8.1 Message format

These messages shall have the following structure:

```
struct {
     uint8 mbox_entity_id;
} MboxLeaveNotify;

struct {
     uint8 mbox_entity_id;
} MboxLeaveAck;
```

These messages are sent when a middlebox wishes to leave a session and shall be processed as defined in clause 4.3.2.5.

4.3.8.2 Message processing

4.3.8.2.1 General

The `MboxLeaveNotify` and `MboxLeaveAck` messages shall only be issued after completeing the handshake, and such messages occurring at earlier stages shall be discarded.

A simplified overview of the function of these two messages follows. When a middlebox wants to leave a TLMSP session, it enqueues an `MboxLeaveNotify` message to be sent in each direction. These messages are forwarded to the endpoints, who in turn each respond with a corresponding `MboxLeaveAck` message. As an `MboxLeaveAck` message travels to the other endpoint, it provides the synchronization point for:

- the entity upstream of the departing middlebox to begin computing hop-by-hop MACs using the pairwise key it shares with the entity downstream of the departing middlebox;
- the departing middlebox to stop participating in the protocol in that direction and begin simply forwarding all transport packets for the remaining lifetime of the transport connections; and
- the entity downstream of the departing middlebox to begin expecting hop-by-hop MACs computed using the pairwise key it shares with the entity upstream of the departing middlebox.

4.3.8.2.2 Detailed operation

Each entity maintains a concept of the current state of each middlebox for each direction of communication (client-to-server and server-to-client). The three states shall be: establishing, participating, and gone. The states shall have the following meaning.

Establishing: The middlebox has not yet completed the handshake in the given direction. This is the initial state.

Participating: The middlebox has completed the handshake and is fully participating in the TLMSP protocol in the given direction.

Gone: The middlebox has reduced its participation in the given direction to forwarding unmodified transport packets.

Each middlebox shall also maintain the single state variable `leave_notify_sent`, which indicates whether it has begun the departure process.

When an entity receives, or in the case of an originating endpoint, sends, an `MboxFinished` message pertaining to a given middlebox, it shall update that middlebox's current state for that direction to participating. Other state transitions are described below. Middleboxes keep track of the upstream and downstream (participating) neighbours as described in clause 4.2.7.2.3, and verify/compute the associated hop-by-hop MACs as also describe in clause 4.2.7.2.3.

When a middlebox in the participating state wishes to leave a TLMSP session, it shall set `leave_notify_sent` to "true" and send an `MboxLeaveNotify` message, with `mbox_entity_id` set to its entity identity, in each direction. An `MboxLeaveNotify` message shall not be combined in a record with any other messages. The middlebox shall ensure that the second `MboxLeaveNotify` message is sent before the `MboxLeaveAck` message corresponding to the first `MboxLeaveNotify` message is received and processed.

When an entity receives an `MboxLeaveNotify` message:

- If the originator of the message is an endpoint, or the origin of the message is a middlebox that is not in the participating state in the direction the message was received, the entity shall raise a fatal `unexpected_message` alert and stop further processing.
- If the entity is not an endpoint, it shall forward the message.
- If the entity is an endpoint, it shall respond with an `MboxLeaveAck` message bearing the same `mbox_entity_id`. The entity should send all `MboxLeaveAck` messages in the same order that the corresponding `MboxLeaveNotify` messages were received.

An endpoint sends an MboxLeaveAck message in response to an `MboxLeaveNotify` as described under `MboxLeaveNotify` processing above. An `MboxLeaveAck` message shall not be combined in a record with any other messages. Immediately after an endpoint sends an `MboxLeaveAck` message, it sets the current state of middlebox `mbox_entity_id` in that direction to gone.

When an entity receives an `MboxLeaveAck` message:

- If the originator of the message is not the upstream endpoint, or the current state of the middlebox indicated by the `mbox_entity_id` is not participating in the direction the message arrived in, the entity shall raise a fatal `unexpected_message` alert and stop further processing.
- If the entity is not an endpoint, it shall forward the message.
- If the entity is upstream of the middlebox `mbox_entity_id`, immediately after sending the message, the entity shall set the current state of the middlebox `mbox_entity_id` in that direction to gone.
- If the entity is downstream of the middlebox `mbox_entity_id`, immediately after processing the received message, the entity shall set the current state of the middlebox mbox_entity_id in that direction to gone.
- If the entity is the middlebox `mbox_entity_id`, and `leave_notify_sent` is "false", it shall raise an `unexpected_message` alert and stop further processing. Otherwise, if leave_notify_sent is "true", it shall forward all subsequent transport packets without performing any further processing.

- If the entity is an endpoint, immediately after processing the received message, the entity shall set the current state of the middlebox mbox_entity_id in that direction to gone.

Although a departing middlebox sends an `MboxLeaveNotify` message in each direction when beginning the departure process, in general, the actual transition of the middlebox's local state to gone will occur at different times in each direction. This gives rise to the possibility that the middlebox's processing in one direction encounters an error that requires an alert to be sent in the other direction, but that direction has transitioned to the gone state, so no such action is possible. In this case, the middlebox shall either send an alert only in the direction in which it is still participating or forward the message whose processing generated the error in such a way that the alert will be raised by the next downstream entity.

When the status of a writer or deleter middlebox changes to gone, the first downstream adjacent writer or deleter middlebox shall from this point on reconfigure to no longer having the leaving middlebox as the expected deleter/writer author of deleter/writer MACs, and shall instead reconfigure to now verify deleter/writer MACs having the next upstream deleter/writer middlebox as the expected author of the corresponding MACs. This could imply that the upstream endpoint enters the role of author of these MACs.

An entity that receives a message whose originator or author is a middlebox whose current state in the direction the message was received is gone shall raise a fatal `unexpected_message` alert and stop further processing.

4.3.9 Message hashes

4.3.9.1 ClientHello and ServerHello value substitutions

The `TLMSP` extension contains several fields whose values can be modified at each hop when traversing from one endpoint to the other. Entities shall use consistent values for these fields when computing verification hashes that include a `ClientHello` or `ServerHello` message, as follows.

For `ClientHello` messages:

- `MiddlexboInfo` entries in `ml_i` with the attribute `inserted` set to `dynamic` shall be omitted. The encoded size of `ml_i` shall not be adjusted.
- If present, the value of `previous_entity_id` shall be replaced by the octet value zero.
- The encoded value of `hbh_id` shall be replaced by the encoding of an empty `HopID` (that is, by a single octet having the value zero).

For `ServerHello` messages, the encoded value of `hbh_id` shall be replaced as described above for `ClientHello` messages.

Omission from the verification hashes of the specific values that each entity observes for these fields does not result in a loss of security as tampering with their values on the wire will cause session establishment to fail.

4.3.9.2 Finished hash

When computing the hash that is included in the client-server `Finished` messages, the processing of IETF RFC 5246 [1], clause 7.4.9 shall be applied, the only difference being that client and server shall omit certain information elements that were inserted by middleboxes since the client and server need not have received identical copies of these messages. Messages that were inserted by middleboxes are recognizable via the dedicated message types used to distinguish middlebox `Handshake` messages from those of client/server.

The `handshake_messages` input to the hash calculation shall be as defined in clause 7.4.9 of IETF RFC 5246 [1], but with the value substitutions described in clause 4.3.9.1 applied and with the following differences.

The first input to the hash shall be the initial `ClientHello`. Additionally if any middleboxes are dynamically discovered during the handshake, the client shall complete the ongoing hash computation, and include in the TLMSP extension of the second `ClientHello` (in the `pre_discovery` field, as defined in clause 4.3.5), a hash of the messages exchanged with the server up to, but not including, the second `ClientHello` following the discovery phase. At this point, the client and server shall reset the hash calculations to re-start with the inclusion of the second `ClientHello`, following the discovery. If there are no dynamically discovered middleboxes, the hash computation shall just proceed.

NOTE 1: The discovery phase itself is protected by:

- the server's signature on the initial messages (including the `MiddleboxList`) as defined in clause 4.3.9.4; and
- the client's inclusion of messages from the discovery phase into the `pre_discovery_hash` field of second `ClientHello`.

After a possible discovery phase, the inputs to the hash shall consist of the remaining set of `Handshake` messages in the order which they appeared, except the following, middlebox-related messages:

- `MboxCertificateRequest` (clause 4.3.6.3), `Certificate2Mbox` (clause 4.3.6.7), `Certificate` sent from client to a middlebox, `CertificateVerify2Mbox` , `ChangeCipherSpec`, and `TLMSPKeyConf` (clause 4.3.7.3) messages;
- `TLMSPKeyMaterial` (clause 4.3.7.2) message directed to a middlebox (non-endpoint); and
- `MboxFinished` (clause 4.3.6.10) messages.

The following middlebox-related messages shall be included (since they are always sent as identical copies towards both client and server):

- the `MboxHello` (clause 4.3.6.1), `MboxKeyExchange` (clause 4.3.6.5), and `MboxHelloDone` (clause 4.3.6.6);
- the `TLMSPKeyMaterial` (clause 4.3.7.2) message directed from one endpoint to the other endpoint.

NOTE 2: As in of IETF RFC 5246 [1], `HelloRequest` (including `MboxHelloRequest`) messages are not included, as they restart the handshake.

- `MboxCertificate` (clause 4.3.6.2) shall be included for middleboxes that present the certificate to both endpoints, which is the case except when at least one endpoint has requested `use_certificate = false` in the corresponding entry of the middlebox list.

Between client and server, the server's `Finished` message shall include a hash of the client's `Finished` message. The client and server shall use the same labels to prefix the hash input to the PRF as in IETF RFC 5246 [1], clause 7.4.9.

4.3.9.3 MboxFinished hash

This hash computation is used for verification between an endpoint and a middlebox and shall also be done with the prescribed processing of IETF RFC 5246 [1], clause 7.4.9, with the value substitutions described in clause 4.3.9.1 of the present document applied as well as including the following items, in the order which they were sent/received. First, if any dynamic middlebox discovery occurs, any message sent during the discovery phase shall be omitted.

NOTE 1: This is to ensure that all middleboxes, including dynamically discovered ones, observe the same value for messages included in the hash. Messages exchanged during discovery are still protected by the client including their hash in the second `ClientHello`, as noted above.

Below, the middlebox-specific messages shall be those relating to the middlebox with which the `MboxFinished` message is associated:

- All messages from `ClientHello` (the ones occurring after the discovery phase, if any, is completed) up to and including the `ServerHelloDone` message.

- `MboxHello, MboxCertificate, MboxKeyExchange, MboxHelloDone`.
- the client's `Certificate2Mbox` response to the middlebox, the `Certificate` response to the server and the corresponding `CertificateVerify` and `CertificateVerify2Mbox` messages.
- the `ClientKeyExchange`.
- the two `TLMSPKeyMaterial` messages, directed between the endpoints.
- `ChangeCipherSpec`.
- for the `MboxFinished` messages between middlebox and the *server* (only), the following items (in this order):
  - a list of received key material contributions, $L_{contrib}$, as defined later in the present clause;
  - the client's `Finished` message with the server;
  - in the `MboxFinished` from the server to middlebox (only), also the middlebox's `MboxFinished`;
- for the `MboxFinished` message between middlebox and the *client* (only):
  - `MboxCertificateRequest`
  - the client's `Finished` message;
  - in the `MboxFinished` from middlebox to client (only), also the server's `Finished` and the client's `MboxFinished`.

For specific middleboxes where at least one of the endpoints have requested `use_certificate = false` in the middlebox list extension, the `MboxCertificate` shall be omitted. Similarly, if at least one of the endpoints have requested `alt_cs` for a middlebox, the `MboxKeyExchange` for that middlebox shall be omitted.

`TLMSPKeyMaterial` messages from the client or server to a middlebox shall not be included. Similarly, `TLMSPKeyConf` messages directed to the client and related to a specific middlebox shall not be included.

NOTE 2: These messages are explicitly verified when received.

The input to the hash in the `MboxFinished` messages between a middlebox and the *server* shall additionally include the concatenated list of all the decrypted `content` fields from all `readerContributions`, `deleterContributions`, and `writerContributions` received from client and server (as part of `TLMSPKeyMaterial` messages), ordered according to their associated `context_id`.

Let $C_{cr}(i)$, $C_{sr}(i)$, $C_{cd}(i)$, $C_{sd}(i)$, $C_{cw}(i)$, and $C_{sw}(i)$, be the decrypted content fields from the client (c) and the server (s) of the `reader_contrib` (r) and the `delete_contrib` (d) or `writer_contib` (w) associated with `context_id` = `i`.

Then this list shall be:

$$L_{contrib} = C_{cr}(i_1) \,||\, C_{sr}(i_1) \,||\, [C_{cd}(i_1) \,||\, C_{sd}(i_1) \,||\, C_{cw}(i_1) \,||\, C_{sw}(i_1) \,||]$$
$$C_{cr}(i_2) \,||\, C_{sr}(i_2) \,||\, [C_{cd}(i_2) \,||\, C_{sd}(i_2) \,||\, C_{cw}(i_2) \,||\, C_{sw}(i_2) \,||]$$
$$C_{cr}(i_3) \,||\, C_{sr}(i_3) \,||\, \ldots$$

where $\{ 0 = i_1 < i_2 < \ldots < i_m \}$ is the set of contexts for which the middlebox has granted access and where the deleter contributions ($C_{cd}(i_j) \,||\, C_{sd}(i_j)$) are included only if the middlebox has delete or write access to the context, and the writer contributions ($C_{cw}(i_j) \,||\, C_{sw}(i_j)$) are included only if the middlebox has write access to the context.

NOTE 3: Since all middleboxes have both read and write access to context zero, $C_{cr}(0)$, $C_{sr}(0)$, $C_{cd}(0)$, $C_{sd}(0)$, $C_{cw}(0)$, and $C_{sw}(0)$ will always be present.

The following labels shall be used to prefix the hash input to the PRF:

- from client to middlebox, `"client to mbox finished"`

- from middlebox to client, "mbox to client finished"
- from server to middlebox, "server to mbox finished"
- from middlebox to server, "mbox to server finished"

4.3.9.4 ClientHello hash (following dynamic discovery)

This hash, included in the TLMSP extension (the pre_discovery_hash as defined in clause 4.3.5), shall be computed as the hash of the concatenation of the following messages occurring during the discovery phase:

- The initial ClientHello and its TLMSP extension, with the value substitutions described in clause 4.3.9.1 applied.
- ServerHello (with extensions, and with the value substitutions described in clause 4.3.9.1 applied).
- Server's Certificate and CertificateRequest (if present).
- TLMSPServerKeyExchange.
- ServerHelloDone.

4.3.9.5 TLMSPServerKeyExchange hash

This hash value shall be included in TLMSPServerKeyExchange messages, and shall also be included in the input to the server's and middlebox's signature related thereto, as defined in clause 4.3.10.1. The hash shall be computed as the hash of all messages from the initial ClientHello (sent prior to any possible dynamic middlebox discovery), up to, but not including, the TLMSPServerKeyExhange itself, with the value substitutions described in clause 4.3.9.1 applied.

When a middlebox generates a TLMSPServerKeyExchange (as defined in clause 4.3.10.1, it does so only directed towards the client), it shall also include in the hash, messages that it has forwarded to the client on behalf of the server, but not messages that it has forwarded on behalf of another middlebox.

4.3.10 Key generation

4.3.10.1 TLMSPServerKeyExchange

TLMSP uses a slightly modified server key exchange message format, compared to IETF RFC 5246 [1]. The message shall be used by both server and middlebox when generating a key exchange message directed to the client. The message includes a hash of previous messages in the handshake and there is further no option for RSA key transport. The message shall have the following format.

```
struct {
    select (KeyExchangeAlgorithm) {
        case dhe_dss:
        case dhe_rsa:
        case ecdhe_dss:
        case ecdhe_rsa:
            ServerDHParams params;
            select (certificate provided) {
               case true:
                  digitally-signed struct {
                      select (server_generated_message) {
                         case true: opaque hash[SecurityParameters.hash_length];
                         case false: struct { };
                      };
                      opaque client_random[32];
                      opaque server_random[32];
                      ServerDHParams params;
                  } signed_params;
               case false:
                  select (server_generated_message) {
                     case true: opaque hash[SecurityParameters.hash_length];
                     case false: struct { };
```

```
                };
            };
    };
} TLMSPServerKeyExchange;
```

The format difference to IETF RFC 5246 [1] is the additional `hash` field. This value shall be computed according to clause 4.3.9.5 and shall be included in the input to the server's or middlebox's signature. This signature serves two purposes. When used by a server, this signature verifies the value of the middlebox lists, both the one received in the `ClientHello`, as well as the list returned in the `ServerHello`, protecting from third party modification attempts during early phases of the handshake. Secondly, when used by the server or a middlebox, it further authenticates any possible `MboxCertificateRequest`, protecting the client's privacy from spoofed requests. When the client or server has requested a middlebox to not `use_certificate`, or, to use an alternative cipher suite with `method_id = anon`, verification of this message is not possible until in conjunction with the `Finished` hash verification.

Since this message is used by both the server and middleboxes, the (implicit) value of `server_generated_message` shall be construed accordingly, based on the originator of the message.

Similar to to IETF RFC 5246 [1], certificate requests shall not be allowed from entities not providing certificates.

When the client receives a `TLMSPServerKeyExchange`, it shall calculate the `hash` field and verify the signature. If the signature verification fails, this indicates the possibility of one or both of:

a) a spoofed `CertificateRequest` or `MboxCertificateRequest`, appearing to come from the sender;

b) an unauthorized modification of one of the middlebox lists (the original client list and/or the list claiming to originate in the server).

In this case, the client shall send a `handshake_failure` alert and terminate the session.

4.3.10.2 General

During the first stage of the handshake, the server and client exchange random nonces, certificates, and signed ephemeral public keys in the `Hello`, `Certificate`, and `KeyExchange` messages respectively. These are used to generate the client-server master secret (via a premaster secret) as per IETF RFC 5246 [1] that defines TLS 1.2. To generate the endpoint-middlebox premaster secret, the same endpoint ephemeral public key shall be re-used but combined with unique, per-middlebox ephemeral keys. To this end, each middlebox also sends messages (`MboxHello`, `MboxCertificate`) containing the middlebox's nonce and its certificate. Different ephemeral public keys shall be used by the middlebox for the exchange with the client and the server and both shall be included in the `MboxKeyExchange` message.

NOTE 1: This is for two reasons: so that the client and server see identical messages and can therefore include them in the hash for the confirmation of the integrity of the key exchange; so both the client and the server can verify that the middlebox has used a different key with the other endpoint.

The client-server premaster secret shall be generated as per clause 8 of IETF RFC 5246 [1].

The client-middlebox premaster secret shall be generated using the ephemeral key and nonce from the client and using the middlebox ephemeral key and nonces exchanged between middlebox and client.

The server-middlebox premaster secret shall be generated using the ephemeral key and nonce from the server and using the middlebox ephemeral key and nonces exchanged between middlebox and server.

NOTE 2: In what follows, `e1` and `e2` correspond to a pair of entities, not necessarily endpoints. When it is of importance that one of `e1` and/or `e2` is an endpint, this will be stressed.

4.3.10.3 Premaster secret and master secret generation

The `pre_master_secret_e1e2` shared between entities `e1` and `e2` is generated in a way specific to the cipher suite in use; annex A describes the predefined suites. The master key shared between precisely two entities, e1 and e2 (two endpoints, an endpoint and a middlebox, or two middleboxes), shall be generated as:

```
master_secret_e1e2 = PRF(pre_master_secret_e1e2,
                         "master secret",
                         id_list ||
```

```
                    e1_ Hello.random ||
                    e2_ Hello.random)[0..47];
```

where the PRF shall be the same as in clause 5 of IETF RFC 5246 [1] i.e. P_SHA256. Here, `id_list` shall be the hash of the concatenated list of the following identities, in the stated order:

1) a client ID, when available (e.g. via a certificate), followed by;

2) all middlebox `MboxCertificate` messages, as available, in the same order as in the final agreed middlebox list, followed by;

3) the Server `Certificate` message, when available.

For items 2 and 3, the entire messages (including `type` and `length` fields) shall be included. If a certificate of some entity is not available to both entities `e1` and `e2`, due to the client and/or server having set the attribute `use_certificate` to false for that entity, the certificate shall be replaced by the value of the `address` field in the middlebox list extension corresponding to that entity. Further, when `e1` is the client, to resolve ambiguity (e.g. when the client provides different certificates to different entities), a (certificate based) client ID shall be considered available to the entity `e1` (client) and `e2` if, and only if:

- `e2` is an entity who has made an explicit certificate request to the client (`MboxCertificateRequest,` if `e2` is a middlebox and `CertificateRequest`, if `e2` is the server), in which case the certificate (identity) to use shall be the one in the client's corresponding response (i.e. `Certificate2Mbox` or `Certificate`);

- `e2` is a middlebox who has not explicitly requested a client certificate, but the server has (via `CertificateRequest`), in which case the certificate (identity) to use shall be the one in the client's corresponding `Certificate` response.

In all other cases, `e2` shall not be considered as having any client certificate available.

EXAMPLE: If middlebox `e2'` is downstream from middlebox `e2` (in the client-server direction) and both middleboxes have requested client certificates, then although both certificate responses will have passed `e2` (making both certificates "visible" to `e2`), `e2` (and the client `e1`) will still only use the certificate included in the response directed to `e2`, since that is the only certificate which is considered as being available.

The order of `e1_ Hello.random` and `e2_ Hello.random` shall be such that `e1` is the entity topologically closest to the client and `e2` is topologically closest to the server.

NOTE: When one of the entities is the client (or server), `e1` is identified with the client (and `e2` with the server).

The same cryptographic hash as that used in the PRF defined by the selected cipher suite shall be applied.

4.3.10.4 Pairwise encryption and integrity key generation

The encryption and integrity keys for communication between entities e1 and e2 when e1 is the client and e2 is the server shall be generated from `SecurityParameters.master_secret_e1e2` derived as in clause 4.3.10.3 according to:

```
key_block_e1e2 = PRF(SecurityParameters.master_secret_e1e2,
                     "key expansion",
                     SecurityParameters.e2_random ||
                     SecurityParameters.e1_random)[0..2*T-1];
```

where `T` shall be defined as follows. Let `e = SecurityParameters.enc_key_length` and `m = SecurityParameters.mac_key_length and n = SecurityParameters.fixed_iv_length` and define `T = e+n+m`. The `key_block_e1e2` shall be partitioned into:

```
e1_to_e2_encryption_key[SecurityParameters.enc_key_length];
e2_to_e1_encryption_key[SecurityParameters.enc_key_length];
e1_to_e2_write_fixed_IV[SecurityParameters.fixed_iv_length];
e2_to_e1_write_fixed_IV[SecurityParameters.fixed_iv_length];
e1_to_e2_mac_key[SecurityParameters.mac_key_length];
e2_to_e1_mac_key[SecurityParameters.mac_key_length];
```

NOTE 1: The only messages which make use of the encryption keys are the `TLMSPKeyMaterial` and `TLMSPKeyConf messages`. The only security protection that makes use of the IV are also when protecting those two messages, and additionally, whenever computing hop-by-hop MACs.

The two last keys shall be used whenever a standalone MAC is to be computed (without encryption) between e1 and e2. When an AEAD transform is in use, this shall be done by only using the MAC-part of the transform, see annex A for the predefined cipher suites.

NOTE 2: When e1 and e2 are the endpoints, a message that is correctly authenticated with these keys will have originated at the endpoint. It has not been altered by a middlebox in transit and it will not have been accessible by anyone else. These keys are also used when the client (or server) send the `TLMSPKeyMaterial` messages between each other containing the contributions.

Keys for communication between client (or server) and each middlebox shall be generated in the above way, identifying the entity e1 with the entity topologically closest to the client and e2 the entity closest to the server.

NOTE 3: These keys, known only to one endpoint and one middlebox, are used in the protection of the `TLMSPKeyMaterial` and `TLMSPKeyConf` messages containing the contribution. The MAC key is also used when a middlebox modifies or inserts new containers or authenticates it via the hop-by-hop MAC.

When e1 and e2 are topologically adjacent middleboxes, keys for hop-by-hop MACs shall also be generated in the same way, now identifying the entity e1 with the entity topologically closest to the client and e2 the entity closest to the server, and now setting `T = m`.

NOTE 4: This implies that when e1 and e2 are both endpoints, or, when precisely one of e1 and e2 is is an endpoint, but the other is a middlebox, two pairwise encryption keys and two IVs will be always generated, and when a non-AEAD transform is used, two further pairwise MAC keys will also be generated. When e1 and e2 are both middleboxes only a single pair of MAC keys will be generated since only keys for the hop-by-hop MAC are needed.

On session resumption, the previously established `key_block_e1e2` shall be refreshed by mixing the existing key with the new client and server random values as defined here:

```
key_block_e1e2_new = PRF(key_block_e1e2,
                         "key expansion",
                         SecurityParameters.client_random_new ||
                         SecurityParameters.server_random_new)[0..2*T-1]);
```

which is then partitioned as stated in the preceding paragraph.

4.3.10.5 Context specific keys

For the context specific keys, the client and server shall generate two pseudorandom partial secrets for each context:

- the client shall generate a client read secret and a client write secret;
- the server shall generate a server read secret and a server write secret.

Partial secrets for different contexts shall be cryptographically independent.

As specified in clause 4.3.7, these partial secrets are only sent to middleboxes to which the endpoint is willing to authorize the corresponding access, encrypted and integrity protected with the keys derived in clause 4.3.10.4. Each party with authorized access to a particular context, `i`, shall derive values associated with each context as follows:

- `client_to_server_reader_enc_key_i`: Encrypt/Decrypt data in direction from the client to server;
- `server_to_client_reader_enc_key_i`: Encrypt/Decrypt data in direction from the server to client;
- for context zero only, `client_to_server_fixed_IV_0`: fixed IV for data in direction from the client to server;

- for context zero only, `server_to_client_fixed_IV_0`: fixed IV for data in direction from the server to client;
- `client_to_server_reader_mac_key_i`: Compute reader MAC for data in direction from client to server (for non-AEAD transforms only);
- `server_to_client_reader_mac_key_i`: Compute reader MAC for data in direction from server to client (for non-AEAD transforms only).

Encryption of messages of other contexts than context zero shall use the same `fixed_IV values` as those derived for context zero.

When also delete access is granted, two additional keys shall be derived:

- `client_to_server_deleter_mac_key_i`: Compute deleter MAC for data in direction from client to server;
- `server_to_client_deleter_mac_key_i`: Compute deleter MAC for data in direction from server to client.

When write access is granted, two additional keys shall be derived:

- `client_to_server_writer_mac_key_i`: Compute writer MAC for data in direction from client to server;
- `server_to_client_writer_mac_key_i`: Compute writer MAC for data in direction from server to client.

NOTE: In some cases (such as when AEAD cipher suites are used) the client/server read keys and the client/server read MAC keys are notionally the same key.

After receiving a `TLMSPKeyMaterial` message from both endpoints, for each authorized context `i`, all authorized parties shall compute the context reader keys for the contexts they can access, using `client_reader_contrib_i` and `server_reader_contrib_i` for context `i`. This shall be repeated using `client_deleter_contrib_i` and `server_deleter_contrib_i` for those entities that have delete or write access the context `i`, and using `client_writer_contrib_i` and `server_writer_contrib_i` for those entities that have write access to context `i`. In more detail, let `MRs[j]` and `MRc[j]` be the random values included in the `MboxKeyExchange` sent directed from the `j`th middlebox (in network topological order) towards the server and client, respectively, i.e. the middlebox-selected random values included in the `server_exch` and `client_exch` part of the `MboxKeyExchange` as defined in clause 4.3.6.5. Notice that by the definition in clause 4.3.6.5 these values are available to all entities in the `MboxKeyExchange`. For each context `i`, each authorized party shall use the partial secrets from client and server to compute two blocks of key material:

```
reader_key_block_i = PRF(server_reader_contrib_i || client_reader_contrib_i,
                         "reader keys",
                         i ||
                         MRs[1] || MRc[1] || MRs[2] || MRc[2] || … || MRs[N] || MRc[N] ||
                         SecurityParameters.server_random ||
                         SecurityParameters.client_random)[0..2*T-1];

deleter_key_block_i = PRF(server_deleter_contrib_i || client_deleter_contrib_i,
                          "deleter keys",
                          i ||
                          MRs[1] || MRc[1] || MRs[2] || MRc[2] || … || MRs[N] || MRc[N] ||
                          SecurityParameters.server_random ||
                          SecurityParameters.client_random)[0..2*m-1];

writer_key_block_i = PRF(server_writer_contrib_i || client_writer_contrib_i,
                         "writer keys",
                         i ||
                         MRs[1] || MRc[1] || MRs[2] || MRc[2] || … || MRs[N] || MRc[N] ||
                         SecurityParameters.server_random ||
                         SecurityParameters.client_random) )[0..2*m-1];
```

where, for context i = 0, T = e+n for AEAD transforms, and t = e+m+n otherwise and for all other contexts i, T = e for AEAD transforms, and t = e+m, where i is the octet context identifier. Each reader_key_block_i above shall be partitioned according to the values m, and T into:

```
client_to_server_reader_enc_key_i[SecurityParameters.enc_key_length];

client_to_server_reader_mac_key_i[SecurityParameters.mac_key_length];
server_to_client_reader_enc_key_i[SecurityParameters.enc_key_length];

server_to_client_reader_mac_key_i[SecurityParameters.mac_key_length];
client_to_server_fixed_IV_0[SecurityParameters.fixed_iv_length]; (for context zero)
server_to_client_fixed_IV_0[SecurityParameters.fixed_iv_length]; (for context zero).
```

Further, each deleter_key_block_i shall be partitioned into:

```
client_to_server_deleter_mac_key_i[SecurityParameters.mac_key_length];
server_to_client_deleter_mac_key_i[SecurityParameters.mac_key_length];
```

and each writer_key_block_i shall be partitioned into:

```
client_to_server_writer_mac_key_i[SecurityParameters.mac_key_length];
server_to_client_writer_mac_key_i[SecurityParameters.mac_key_length];
```

The derived fixed_IV values for context zero shall be used for all contexts, when the cryptographic transform requires a fixed IV. Due to the additional inclusion of the sequence number in the final IV, collsions are still avoided.

On session resumption, the previously established keys, the reader_key_block_i, deleter_key_block_i, and writer_key_block_i, for each context i, shall be refreshed by mixing the existing secrets with the new client and server random values as defined here:

```
reader_key_block_new_i = PRF(reader_key_block_i,
                             "reader keys",
                             i ||
                             SecurityParameters.server_random_new ||
                             SecurityParameters.client_random_new)[0..2*T-1];

deleter_key_block_new_i = PRF(deleter_key_block_i,
                              "deleter keys",
                              i ||
                              SecurityParameters.server_random_new ||
                              SecurityParameters.client_random_new)[0..2*m-1];

writer_key_block_new_i = PRF(writer_key_block_i,
                             "writer keys",
                             i ||
                             SecurityParameters.server_random_new ||
                             SecurityParameters.client_random_new)[0..2*m-1];
```

which are then partitioned as stated in the immediately preceding paragraph.

4.3.10.6 Key extraction

The functionality of this clause shall be optional to implement and use. When implemented, the functionality of this clause may be used by an application to extract key material for other purposes. Specifically, one or more additional key blocks shared uniquely between entities with a certain access right to a context i shall then be extracted as follows:

```
   extracted_reader_keyblock_i = PRF(reader_key_block_i,
                                     "TLMSP reader key extraction",
                                     SecurityParameters.server_random ||
                                     SecurityParameters.client_random ||
                                     [context_value_length || context_value])[0..N-1];

   extracted_deleter_keyblock_i = PRF(deleter_key_block_i,
                                      "TLMSP deleter key extraction",
                                      SecurityParameters.server_random ||
                                      SecurityParameters.client_random ||
                                      [context_value_length || context_value])[0..N-1];
```

```
   extracted_writer_keyblock_i = PRF(writer_key_block_i,
                                     "TLMSP writer key extraction",
                                     SecurityParameters.server_random ||
                                     SecurityParameters.client_random ||
                                     [context_value_length || context_value])[0..N-1];
```

where all parameters named as in clause 4.3.10.4 are the same, and where `context_value` shall be an optional string of length `context_value_length` octets. N is the number of desired output octets. Different applications of this function for a given writer, deleter, or reader key block shall use distinct values of `context_value`.

4.4 The Alert protocol

4.4.1 General

All alert messages before the `ServerHello` message has been observed shall follow and be limited to the definitions in [1]. After the transmission or receipt by an entity of a `ServerHello` containing a `TLMSP` extension, all alert messages that entity originates shall use the containered format described in clause 4.2.3.1.6 and will thus indicate the entity ID of the entity originating the alert.

4.4.2 Alert message types

The set of `Alert` protocol messages extend IETF RFC 5246 [1] as follows:

```
enum {
    close_notify(0), unexpected_message(10),
     … ,                                    /*    the existing TLS alert codes           */

    middlebox_route_failure(170),           /* middlebox fails to connect to next hop    */
    middlebox_authorization_failure(171),   /* endpoint does not accept middlebox        */

    unknown_context(172),          /* entity does not recognize a context or its purpose */
    unsupported_context(173), /* middlebox can not perform requested operation on context */
    middlebox_key_verify_failure(174),
    bad_reader_mac(175),                    /* reader MAC failed to verify               */
    bad_deleter_mac(176),                   /* ditto, deleter MAC                        */
    bad_writer_mac(177),                    /* ditto, for writer MAC                     */
    middlebox_key_confirmation_fault(178),  /* failure to verify key-share               */

    middlebox_suspend_notify(179)           /* middlebox leaves the session              */

    …,
    (255)
} AlertDescripton;
```

For existing `Alert` messages, clause 7.2 of IETF RFC 5246 [1] shall apply. The use of the `bad_reader_mac`, `bad_deleter_mac`, `bad_writer_mac`, and `bad_record_mac` alerts are described in clause 4.2.2.2.

The `middlebox_suspend_notify` alert is a softer version of the `MboxLeaveNotify` message. This alert signals that the middlebox will remain on-path, but only to verify and generate hop-by-hop MACs without performing any message inspection.

The `level` field of the additional messages shall be assigned an `AlertLevel` value as follows:

- `middlebox_suspend_notify`: `warning(1)`;
- `middlebox_route_failure`, `middlebox_authorization_failure`, `unknown_context`, `unsupported_context`, `middlebox_key_verify_failure`, `bad_reader_mac`, `bad_writer_mac`, and `middlebox_key_confirmation_fault`: `fatal(2)`.

The other `Alert` levels shall be as defined in clause 7.2 of IETF RFC 5246 [1].

If a middlebox encounters a fatal TLMSP or connectivity related error which leads to it closing the connection, prior to doing so, the middlebox shall send a `close_notify` alert in both directions.

4.5 The ChangeCipherSpec protocol

The single message of the `ChangeCipherSpec` protocol shall be as specified in clause 7.1 of IETF RFC 5246 [1].

In TLMSP, there are the following differences in effects (semantics) of issuing the `ChangeCipherSpec` message.

- The negotiated keys and cipher suite as well as any negotiated record size extention as per IETF RFC 8449 [7] shall be applied to the contents of the `TLMSPKeyMaterial` and `TLMSPKeyConf` messages as described in clauses 4.3.7.2 and 4.3.7.3, even though these messages occur before `ChangeCipherSpec`. No record layer protection of these message shall however be performed.
- Sequence numbers are not defined prior to `ChangeCipherSpec`, but shall come into effect in the usual way following this message.

Annex A (normative): Defined cipher suites

A.1 General

The cipher suites defined in this annex are only defined for TLMSP. When using the fallback mechanisms of annex C, standard TLS 1.2 cipher suites shall be used.

A.2 Key Exchange

The cipher suites defined below in clauses A.3 to A.5 are defined for TLMSP use. One of the following key exchange methods as defined in TLS 1.2 [1] shall be used.

- `ECDHE_ECDSA`. One of the following curves should be used: `secp256r1`, `secp384r1`, `secp512r1` (defined in FIPS 186-4 [10]), `x25519` or `x448` (defined in IETF RFC 7748 [5]). This key exchange method shall be supported.
- `DHE_DSS`. One of the following groups should be used: `ffdhe2048` or `ffdhe3072` (defined in IETF RFC 7919 [6]).

A.3 AES_{128,256}_GCM_SHA{256,384}

A.3.1 General

The cipher suite shall be `TLS_*_WITH_AES_{128,256}_GCM_SHA{256,384}`, for GCM as defined in clause 3 of IETF RFC 5288 [8] (where * shall be replaced by one of `ECDHE_ECDSA` or `DHE_DSS`, key exchange mechanisms as defined in clause A.1 of the present document) with the following exception. The IV shall instead of the format specified in [9], be calculated as follows:

1) The 64-bit (8 octet) `seq_author` context-independent sequence number of the author (i.e. the value `seq_tx` for an entity generating an outbound message unit, or, `seq_rx[e_id]`, for an entity processing a received message unit having `e_id` as its author) shall be left-padded with the one-octet entity ID of the author, a one-octet value defining the type of MAC computed, and two zero-octets to form a 12-octet value `IV' = e_id || mac_type || 0x00 || 0x00 || seq_author`.
2) Compute a 12-octet fixed IV-value according to clause 4.3.10.4 or 4.3.10.5 (depending on whether context specific keys or only pairwise keys are used), let the result be `write_IV`.
3) Form the final IV as `IV = IV' XOR write_IV`.

For the reader MAC (i.e. the MAC built in to the AES GCM AEAD transform), `mac_type` shall have the value 0x52 (ASCII code of the character "R"). Definition of `mac_type` for the other MACs shall be as defined in clause A.3.2. Only the `e_id` part of the IV shall be explicitly signalled, thus `record_iv_length` shall be 1.

NOTE 1: The IV format above is compatible with that of TLS 1.3 [i.8], except for the inclusion of `e_id`.

NOTE 2: Internally, AES-GCM will use the above `IV` as the 96 most significant bits in the counter.

NOTE 3: In step 2, pairwise keys are used only for the `TLMSPKeyMaterial` and `TLMSPKeyConf` messages, and when forming the `IV'` is done as part of stand-alone MAC computation according to clause A.3.2. In all other cases, context-specific keys are used.

All TLMSP entities shall support this cipher suite.

A.3.2 Additional MAC computations

When generating additional MAC values, i.e. the writer and hop-by-hop MAC values, only the GMAC function of AES-GCM shall be used as per NIST SP 800-38D [11] with the appropriate key (the writer key, the MAC key shared only with an endpoint, or the key shared with the next hop entity, respectively). The input (`MAC_INPUT`) shall consist of the input data as defined in clauses 4.2.7.2.2 and 4.2.7.2.3, depending on which MAC to compute.

When computing or verifying a deleter, writer, or hop-by-hop MAC, the value `mac_type` of the `IV'` shall have the values 0x44, 0x57, and 0x48, respectively. This corresponds to the ASCII codes for the letters "D", "W", and "H", respectively.When verifying a received deleter (or writer) MAC, the `IV'` shall use the entity ID and sequence number of the deleter (or writer) author. This will always be the upstream closest entity with deleter (or writer) access to the corresponding context.

When verifying a received hop-by-hop MAC value, the `IV'` shall use the entity ID and expected next global sequence number of the upstream neighbour.

When computing the deleter or writer MAC of an outbound message unit, or the hop-by-hop MAC of an outbound record, the `IV'` shall always use the author's entity identity. For the deleter MAC, writer MAC, or the hop-by-hop MAC of a record for a protocol that does not use containers, `seq_author` shall be the author's current global transmit sequence number. For the hop-by-hop MAC of a record for a protocol that uses containers, `seq_author` shall be the author's global transmit sequence number corresponding to the first container in the record.

A.4 AES_{128,256}_CBC_SHA{256,384}

This cipher suite shall be `TLS_*_WITH_AES_{128,256}_CBC_SHA{256,384}` as defined in clause A.5 of IETF RFC 5246 [1] with the following exception. The `IV` is carried partially explicitly in the protected fragment and shall have the following form: `IV = ((e_id || seq_author) << 56) XOR write_IV` where:

- `e_id` shall be the one-octet entity identity for the originator;
- `seq_author` shall be the 64-bit context-independent sequence number of the author;
- `write_IV` shall be a 16-octet fixed IV-value generated according to clause 4.3.10.4 or 4.3.10.5 (depending on whether context specific keys or only pairwise key are used).

Only the `e_id` part of the IV shall be explicitly signalled, thus `record_iv_length` shall be 1.

NOTE: In this case, no IVs for the separate deleter, writer, and hop-by-hop MACs are needed.

A.5 AES_{128,256}_CTR_SHA{256,384}

This cipher suite consists of the counter-mode encryption part of AES_GCM (see clause A.2 of the present document), in conjunction with the HMAC_SHA256 (or SHA384) MAC as defined in IETF RFC 5246 [1], for AES_CBC_SHA256. The `IV` for encryption shall be generated as defined in clause A.3 of the present document.

NOTE 1: This cipher suite has no analogue in TLS 1.2.

NOTE 2: In this case, no IVs for the separate deleter, writer, and hop-by-hop MACs are needed.

A.6 Additional cipher suites

The cipher suite `TLMSP_NULL_WITH_NULL_NULL` may be used only for testing purposes, providing no security. In this case no sequence number maintenance is needed. While it would be possible to omit the explicit IV (carrying the 1-octet entity ID oth the author), the IV shall still be present, allowing a uniform message format.

The cipher suites TLMSP_ECDHE_ECDSA_WITH_NULL_SHA256 and TLMSP_DHE_DSS_WITH_NULL_SHA256 provides only integrity protection using the integrity part of the cipher suite defined in clause A.5 and should not be used without careful consideration. These cipher suites require sequence number management.

A.7 Summary of security parameters

Table A.1: Summary of security parameters

Cipher suite (Annex ref.)	parameter length (*_length)				
	enc_key	mac_key	fixed_iv	block	record_iv
GCM (A.3)	16 or 32	= enc_key (see note)	12	16	1
CBC (A.4)	16 or 32	= enc_key	16	16	1
CTR (A.5)	16 or 32	= enc_key	16	16	1
NULL_SHA256 (A.6)	0	32	0	n/a	1
NULL_NULL (A.6)	0	0	0	n/a	1
NOTE: For AEAD transforms, a separate MAC key is only needed for the additional deleter, writer, and hop-by-hop MACs.					

By the notation *_length in the heading of Table A.1, it is to be understood that all parameter names (e.g. enc_key) is to be suffixed by _length in order to cross-reference other parts of the present document where the corresponding parameter is being used.

EXAMPLE: The parameter enc_key is in other parts of the present document denoted enc_key_length.

A.8 Cipher suite identifiers

Table A.2: TLMSP cipher suite identifiers

Name	Identifier
TLMSP_NULL_WITH_NULL_NULL	{0x00,0x00}
TLMSP_ECDHE_ECDSA_WITH_NULL_SHA256	{0x00,0x01}
TLMSP_DHE_DSS_WITH_NULL_SHA256	{0x00,0x02}
TLMSP_ECDHE_ECDSA_WITH AES_128_GCM_SHA256	{0x00,0x03}
TLMSP_DHE_DSS_WITH AES_128_GCM_SHA256	{0x00,0x04}
TLMSP_ ECDHE_ECDSA_WITH AES_256_GCM_SHA256	{0x00,0x05}
TLMSP_ DHE_DSS _WITH AES_256_GCM_SHA256	{0x00,0x06}
TLMSP_ ECDHE_ECDSA_WITH AES_256_GCM_SHA384	{0x00,0x07}
TLMSP_ DHE_DSS _WITH AES_256_GCM_SHA384	{0x00,0x08}
TLMSP_ ECDHE_ECDSA_WITH AES_128_CBC_SHA256	{0x00,0x09}
TLMSP_ DHE_DSS _WITH AES_128_CBC_SHA256	{0x00,0x0A}
TLMSP_ ECDHE_ECDSA_WITH AES_256_CBC_SHA256	{0x00,0x0B}

Name	Identifier
TLMSP_ DHE_DSS _WITH AES_256_CBC_SHA256	{0x00,0x0C}
TLMSP_ ECDHE_ECDSA_WITH AES_256_CBC_SHA384	{0x00,0x0D}
TLMSP_ DHE_DSS _WITH AES_256_CBC_SHA384	{0x00,0x0E}
TLMSP_ ECDHE_ECDSA_WITH AES_128_CTR_SHA256	{0x00,0x0F}
TLMSP_ DHE_DSS _WITH AES_128_CTR_SHA256	{0x00,0x10}
TLMSP_ ECDHE_ECDSA_WITH AES_256_CTR_SHA256	{0x00,0x11}
TLMSP_ DHE_DSS _WITH AES_256_CTR_SHA256	{0x00,0x12}
TLMSP_ ECDHE_ECDSA_WITH AES_256_CTR_SHA384	{0x00,0x13}
TLMSP_ DHE_DSS _WITH AES_256_CTR_SHA384	{0x00,0x14}

A.9 Future extensions

To provide protection against keystream reuse and vulnerabilities in AEAD transforms, any future extension to the present document in the form of additionally defined cipher suites shall comply with the following rules:

a) Any IV used to create a protected TLMSP message unit (a record or a container) during a session shall:

   1) include a per session fixed, or, per message unit variable nonce, of at least 64-bits of entropy;

   2) enable determining the message unit author by a receiving entity;

   3) never repeat, for any fixed value of (`e_id`, `key_id`) where `e_id` is entity identity of the message author and `key_id` is some unique identifier for the key used by the author.

b) For AEAD transforms, only ones that allow separation of the encryption function from the MAC-value computation shall be used in TLMSP.

Requirements a)2) and a)3) may be implemented by including (`e_id`, `seq[c]`) in the IV using a mapping which is one-to-one with respect to (`e_id`, `seq[c]`).The predefined cipher suites have been designed to allow the same IV structure to be used for reader MAC, deleter MAC, writer MAC, and hop-by-hop MAC, in a secure manner. Future extensions to the present document may instead opt to specify two separate IVs: one for the hop-by-hop MAC and another IV for the other three MACs, or, specify the use of four completely separate IVs. The IV for the reader MAC shall always be the one included in the Fragment part of the TLMSP record/container (see clause 4.2.7.1) and the location of any additional IV(s) shall then be specified.

Annex B (normative): Alternative cipher suites

B.1 General

The alternative cipher suites defined in this annex shall be identical to those of clauses A.3, A.4 and A.5, apart from the key exchange and authentication during the handshake. The use of one of the alternative cipher suites shall be signalled by using the corresponding cipher suite identifier as those defined in annex A, but additionally setting the value of `cipher_suite_options` in the middlebox list extension as defined in clause 4.3.5 to "`alternative`", and, to set the `methodID` of the `altCS` field to indicate which alternative cipher suite to use: "anon", "`psk`" or "`gba`". Clauses B.2.1, B.2.2 and B.2.3, respectively, provide normative definitions of each of the three choices.

NOTE: Since the middlebox lists contains one individual value of `cipher_suite_options` value for each middlebox, this implies that each middlebox's use of the alternative cipher suites can can be configured individually. Since the only difference lies in the key exchange and authentication mechnisms toward the endpoint, and not in the bulk data protection algorithms, this does not cause any interoperability problems. Simililary, while the endpoints (client and server) may individually choose different alternative cipher suites, since it only affects the key exchange and authentication between the middlebox and that endpoint, also this implies no interoperability issues.

Middleboxes who are not configured by an endpoint to use alternative cipher suites, shall use the key exchange and authentication mechanisms exactly as defined in annex A, together with certificates for communication with that endpoint.

A middlebox which accepts the endpoint's suggested use of the alternative cipher suite shall acknowledge this by setting the value of `client_altCS`, and/or, `server_altCS` (as apropriate) in the `MboxHello` to indicate "`alternative`".

B.2 Defined alternative cipher suites

B.2.1 Anon

The endpoint requesting this alternative cipher suite shall set the `method_id` of the `alt_cs` field of the corresponding middlebox list entry in the hello message to indicate "anon".

The key exchange corresponding to the selected cipher suites of annex A shall be used, but without authentication. Security aspects of not authenticating the endpoint shall be considered before using this alternative cipher suite.

B.2.2 Preshared keys

B.2.2.1 General

In this case, the client is assumed to have a pre-shared key with the middlebox.

B.2.2.2 Technical Details

B.2.2.2.1 ClientHello and ServerHello

The endpoint (client or server) requesting this alternative cipher suite shall set the `method_id` of the `alt_cs` field of the corresponding middlebox list entry to indicate "psk". The endpoint shall also include in the field `credentialHint` of the `alt_cs` field, of the middlebox list extension, an identifier for this key.

B.2.2.2.2 MboxKeyExchange

This message shall be generated and used as normally, except that the endpoint that requested the alternative cipher suite shall ignore the included key exchange information (since the keys will be used on a pre shared key instead).

B.2.2.2.3 TLMSPKeyMaterial

When generating keys, the preshared key, `PSK`, indicated by `credential_hint` shall take the place of the master key of clause 4.3.10.4, i.e.:

```
key_block_e1e2 = PRF(PSK,  "key expansion",
                     SecurityParameters.e2_random ||
                     SecurityParameters.e1_random).
```

From this key block encryption keys and MAC keys shall be obtained as also describe in clause 4.3.10.4. Authentication of the client toward the middlebox is then assured by successful verification of the associated MAC values.

No other messages are affected by this extension.

B.2.3 GBA

B.2.3.1 General

This alternative cipher suite shall only be used between the client and a middlebox.

The entire clause B.2.3 specifies an additional authentication and key exchange method, specific to Mobile Network Operators (MNO). The mutual authentication between middlebox and client (or server) obtained through this method provides stronger assurance that the middlebox services are only provided to clients who subscribe to those MNO services. It also, if applicable, enables more robust charging of the services.

When implementing TLMSP, clients equipped with USIM cards, such as smartphones, should implement and use the extension described whenever it wishes to receive services by middleboxes provided by a MNO (Mobile Network Operator).

EXAMPLE: An example use case is when connecting to an internet server via the MNO's network.

NOTE: The client is assumed to have prior knowledge of those middleboxes in the `MiddleboxList` (or the server) that are associated with the MNO and therefore which middleboxes can support this extension. How the client obtains this prior knowledge is outside the scope of the present document, but it can be done in conjunction to MNO configuration of the client. If the client incorrectly assumes a certain middlebox supports these extensions (or not), no adverse security issues result; an error alert will be raised or a fallback to standard (certificate based) TLMSP will occur.

B.2.3.2 Technical details

B.2.3.2.1 General

A client wishing to make use of this alternative cipher suite shall first perform GBA (Generic Bootstrapping Architecture) bootstrapping with the BSF (Bootstrapping Server Function) as defined in the GBA specification ETSI TS 133 220, clause 4.5.2 [11].

A middlebox or server supporting this extension is viewed as a NAF (Network Application Function) in GBA terminology and is assumed to follow the GBA-specified procedures, observing the details of this entire clause B.2.3.

B.2.3.2.2 ClientHello

To indicate use of the alternative cipher suite, the client shall set the `method_id` of the `alt_cs` field of the corresponding middlebox list entry to indicate "psk". The client shall also include in the field `credential_hint` of the `alt_cs` field, of the middlebox list extension, an identifier for the key to used, as follows:

```
credential_hint = B-TID,
```

where `BTID` is the B-TID value obtained during GBA bootstrapping, defined in clause C.2.1.2 of ETSI TS 133 220 [11], i.e. an encoded Network Access Identifier (NAI) of format:

```
base64encode(RAND)@BSF_servers_domain_name
```

NOTE: All strings in the GBA specification are encoded in UTF-8 format.

When this extension is present in the `ClientHello` but a non-empty BTID, a middlebox supporting this extension shall contact the BSF as indicated by the `BTID` of the extension data and (unless already available) request the NAF-key (`Ks_NAF`), as defined in clause 4.5.3 [11]. When deriving or requesting the `Ks_NAF`, the client, middlebox (NAF), and the BSF shall use the same middlebox address as the `NAF-Id` (the "`address`" field, excluding the one-octet "`middlebox_id`") as described in the middlebox extension list defined in clause 4.3.5 of the present document.

At this point, any the client and any middlebox (or server) that supports this extension will have or be able to derive a pairwise unique, shared key `Ks_NAF`. This shared key shall be used as in clause B.2.3.2.4.

B.2.3.2.3 MboxKeyExchange

This message shall be generated and used as normally, except that the client shall ignore the included key exchange information (since the keys will be used on a pre shared keyinstead).

B.2.3.2.4 TLMSPKeyMaterial

When generating keys shared between the client and the middlebox using this method, the associated `Ks_NAF` shall be used in place of the master secret of clause 4.3.10.4, i.e.:

```
key_block_e1e2 = PRF(Ks_NAF, "key expansion",
                     SecurityParameters.middlebox_random ||
                     SecurityParameters.client_random)
```

where e1 is the client and e2 the middlebox. From this key block encryption keys and MAC keys shall be obtained as also describe in clause 4.3.10.4. Authentication of the client toward the middlebox is then assured by successful verification of the associated MAC values.

GBA-produced keys have an associated lifetime that shall be respected by this TLMSP profile.

EXAMPLE: This implies that the client shall not attempt to resume a session where the underlying GBA keys (Ks_NAF) have expired.

No other messages are affected by this authentication method.

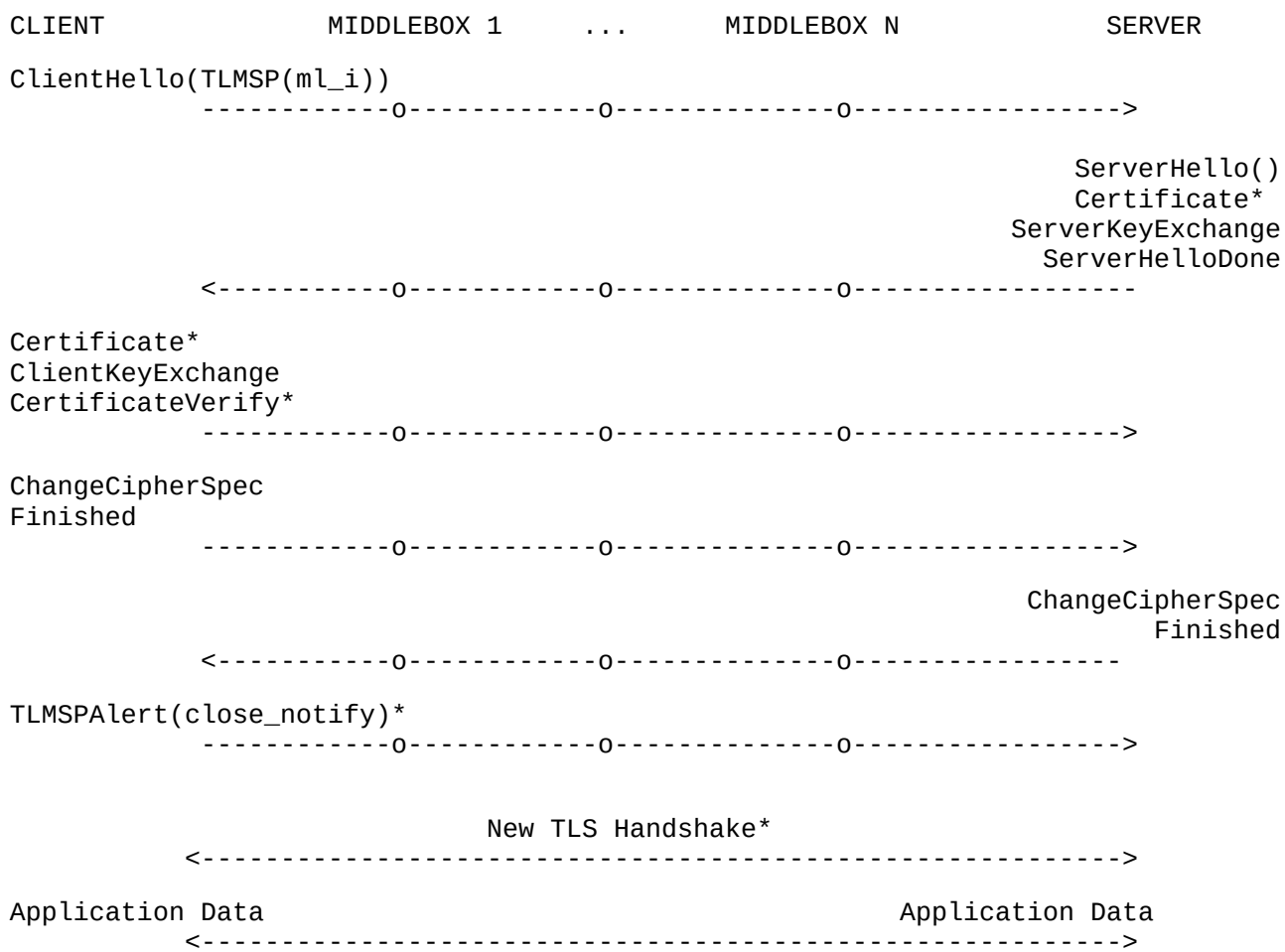
Annex C (normative): TLMSP alternative modes

C.1 Fallback to TLS 1.2

The specification of this annex shall be mandatory to support fallback for TLMSP clients and middleboxes.

Where a client does not know if the server supports TLMSP, a fallback to TLS 1.2 may be performed as follows. When a server lacks TLMSP support, another option is to fallback to TLMSP proxying, described in clause C.2 of the present document. The fallback mechanism of the current annex shall be followed if the client has not indicated that it accepts TLMSP proxying using the `Hello` extension in clause 4.3.5, whose usage is defined in clause C.2.

In the signalling below, the absence of the TLMSP middlebox list extension in the `ServerHello` indicates to client and middleboxes that the server does not support TLMSP. In this case, the middleboxes shall remain on-path, but in a passive capacity; forwarding the messages between client and server, allowing the handshake to complete, so that the connection can be established and the client can cryptographically verify that the server does not support TLMSP.



NOTE: The symbols *, o and x are defined in the same way as in Figure 6.

Figure C.1: Handshake for TLS 1.2 fallback

After the handshake above is complete, the client may send a TLS `close_notify` alert to the server and may restart the negotiation directly with the server, without the middleboxes now taking part in the session. The client may omit this Alert, to indicate it permits the middleboxes to remain on-path, continuing to forward application data. However, a middlebox on the connection may send a `close_notify` alert to both endpoints, even if the client does not, to start a new, end-to-end TLS Handshake.

C.2 Fallback to TLMSP-proxying

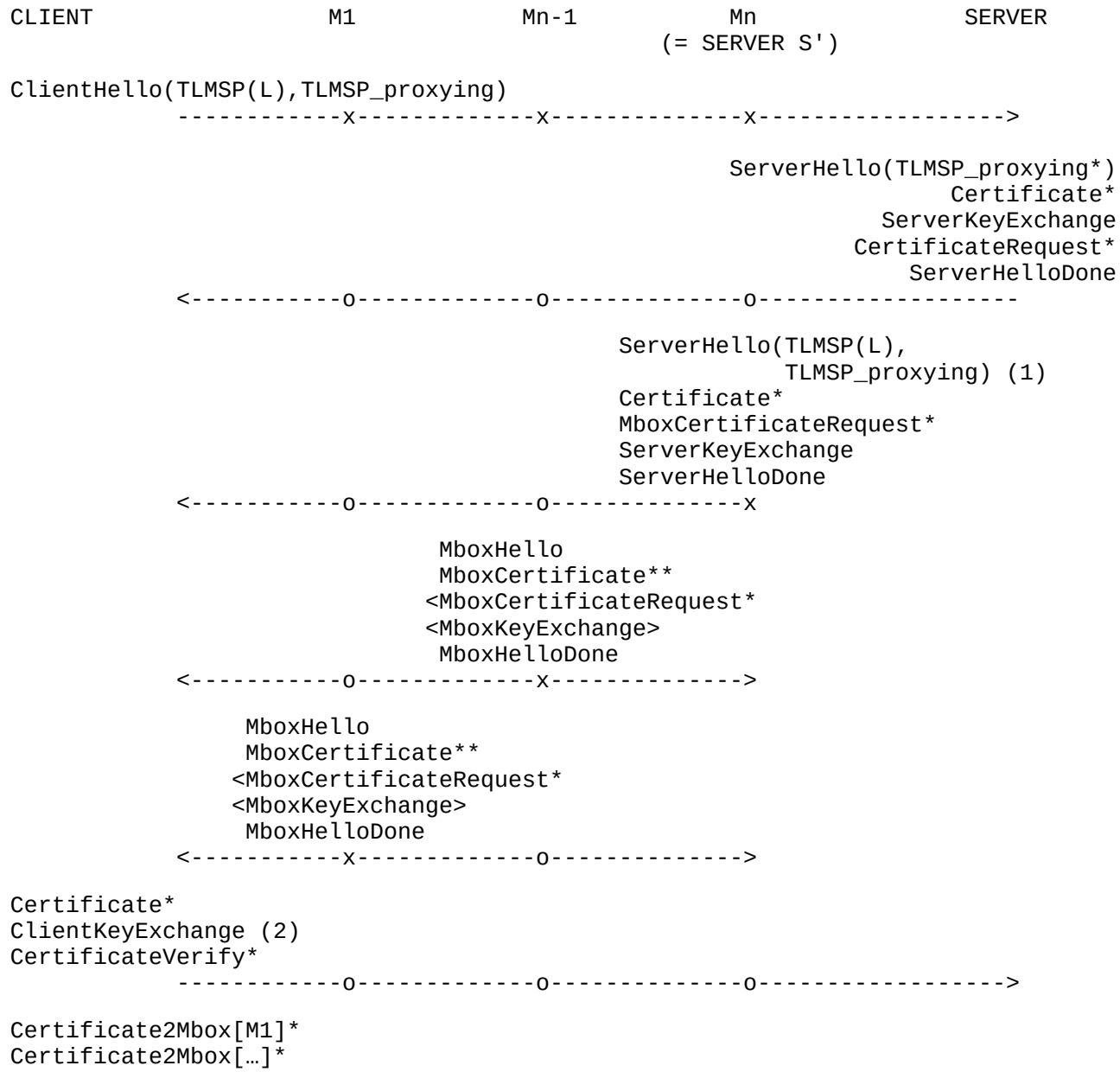
C.2.1 General

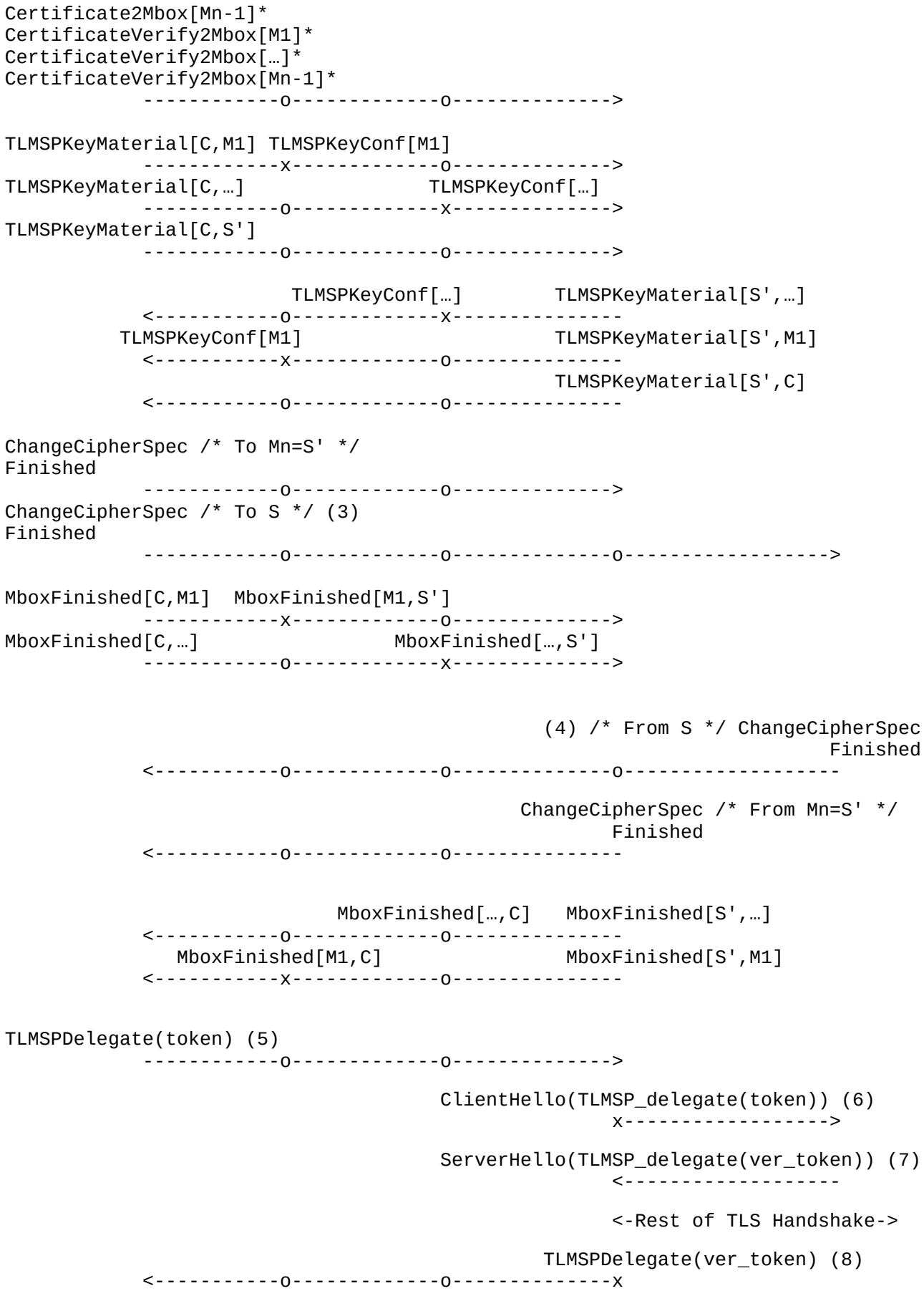
The procedure defined in clause C.2.2 of the present document may be supported by TLMSP clients and may be supported by middleboxes. The procedure shall not be used when the `ClientHello` does not contain a TLMSP proxying extension, as defined in clause 4.3.5. The server's lack of support for full TLMSP will be indicated by the absence of the TLMSP middlebox list extension in the `ServerHello`.

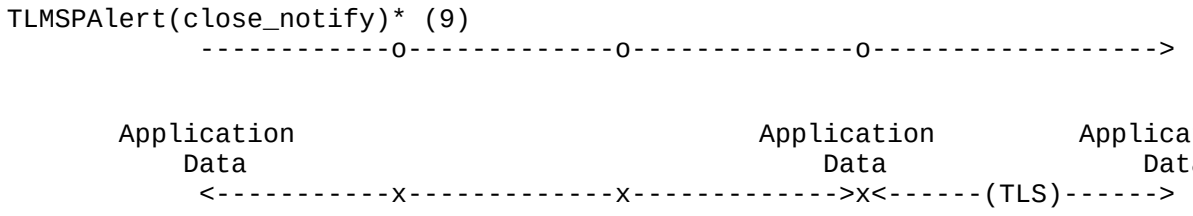
If middleboxes were dynamically discovered and the client accepts these middleboxes, the client shall include the complete list of middleboxes in the computation of the verification hash of the `Finished` message, exchanged with the server. This is enabled via the `ServerUnsupport` message from the last middlebox, see clause 4.3.2.3.1.

C.2.2 Fallback procedure

The principle behind the signalling is that the last middlebox, `N`, steps in and proposes to act as a TLMSP server, `S'` toward the client, while running TLS 1.2 with the server. Other middleboxes remain as normal TLMSP middleboxes.




Certificate2Mbox[Mn-1]*
CertificateVerify2Mbox[M1]*
CertificateVerify2Mbox[…]*
CertificateVerify2Mbox[Mn-1]*
TLMSPKeyMaterial[C,M1] TLMSPKeyConf[M1]
TLMSPKeyMaterial[C,…] TLMSPKeyConf[…]
TLMSPKeyMaterial[C,S']
TLMSPKeyConf[…] TLMSPKeyMaterial[S',…]
TLMSPKeyConf[M1] TLMSPKeyMaterial[S',M1]
TLMSPKeyMaterial[S',C]
ChangeCipherSpec /* To Mn=S' */
Finished
ChangeCipherSpec /* To S */ (3)
Finished
MboxFinished[C,M1] MboxFinished[M1,S']
MboxFinished[C,…] MboxFinished[…,S']
(4) /* From S */ ChangeCipherSpec
Finished
ChangeCipherSpec /* From Mn=S' */
Finished
MboxFinished[…,C] MboxFinished[S',…]
MboxFinished[M1,C] MboxFinished[S',M1]
TLMSPDelegate(token) (5)
ClientHello(TLMSP_delegate(token)) (6)
ServerHello(TLMSP_delegate(ver_token)) (7)
<-Rest of TLS Handshake->
TLMSPDelegate(ver_token) (8)



NOTE: The symbols *, o and x are defined in the same way as in Figure 6.

Figure C.2: Handshake for TLMSP proxying

With reference to the numerals in the signalling above, the following steps shall be taken:

1) The last middlebox is the first to receive the `ServerHello` and recognizes the absence of the TLMSP middlebox list extension. Since the client indicated acceptance for proxying (via the presence of the `TLMSP_proxying` extension in the `ClientHello`), the last middlebox offers to act as a TLMSP proxy by echoing the client's extension in a standard TLS `ServerHello`, which is sent alongside forwarding the original `ServerHello`, from the actual server. The middlebox `Mn` notes the cipher suites proposed by the client for use in step 6.

2) If the client accepts `TLMSP_proxying`, it shall now:

   - perform a standard TLS session handshake with the original server; and
   - perform a TLMSP handshake with the middlebox `Mn` acting as a TLMSP server and other middleboxes acting as standard TLMSP middleboxes.
     If the client does not accept it, it may close the connection.

3) The client-side TLS handshake with the original server is completed. The client-side TLMSP handshake is also completed.

4) Server-side also completes.

5) The client shall instruct the last middlebox to take care of proxying by setting up a TLS 1.2 session with the server. Included in this message shall be a (secured) delegation token and a verification token, defined in clause C.2.3.

6) The middlebox `Mn` shall now, acting as a TLS client, initiate TLS setup with the server. In the TLS `ClientHello of Mn`, a `TLMSP_delegation` extension shall be included if, and only if, the server's previous `ServerHello` (step 1) contained the `TLMSP_proxying` extension. The `TLMSP_delegation` extension shall comprise the token. `Mn` shall not propose any cipher suites of lower strength than those observed in step 1 and should propose TLS cipher suites that are a subset (including the full set) of the TLMSP cipher suites observed in step 1. `Mn` should preserve the order of the cipher suites observed in step 1 in its `ClientHello`.

7) If the server understands the `TLMSP_delegation` extension, the server shall include a verification token (`ver_token`) after completion of the TLS handshake between `Mn` and the server. `Mn` shall verify this acknowledgement token using the verification token sent in step 5, as defined in clause C.2.3.

8) `Mn` shall return a `TLMSPDelegate` message comprising the acknowledgement token.

9) The client shall attempt to verify the acknowledgement token. If verification is successful, it shall close the TLS session with the server.

If the server does not understand the `TLMSP_delegate` extension/message, then the extension of message (7) will not be present.

C.2.3 Message and processing details

C.2.3.1 TLMSP proxying and delegate extension and message specifications

The `TLMSP_proxying` extension shall have `extension_type = 37, "0x25"` and the `extension_data` shall consist of the handshake ID: `HandshakeID hs_id`.

The `TLMSP_delegate` extension (used in steps 6 and 7) shall have `extension_type = 38, "0x26"` and shall have as `extension_data`

```
DelegateToken token;
```

where `DelegateToken` is defined as

```
struct {
    HandshakeID hs_id;
    uint8 token_length;
    opaque token_value[token_length];
} DelegateToken;
```

The `TLMSPDelegate` message (steps 5 and 8 in clause C.2.2) shall have the following format

```
struct {
    DelegateToken token;
} TLMSPDelegate;
```

C.2.3.2 Delegate message specification

The `token_value` of the token included in message (5,6) and `ver_token` of message (7,8) of clause C.2.2 shall be generated as defined in Eq. 1 and Eq.2 in the present clause.

Let `master_secret_C_S` be the TLS master secret established between client and server. Then

```
ver_token = PRF(master_secret_C_S, "ver token", ServerCertificate)[0..31]
```
(Eq 1)

and:

```
token = PRF(ver_token, "delegate token", ServerCertificate)[0..31]
```
(Eq 2)

The value of `token_length` is 32. The PRF shall be the same as defined in clause 4.3.10 for the key derivations.

The value `ServerCertificate` shall be taken from the original server's certificate message, binding the `token` and `ver_token` to the server.

C.2.3.3 Processing

When the server receives message (6) of clause C.2.2, assuming it understands the `TLMSP_delegate` extension, the server shall verify that the received `token` has been computed as defined in clause C.2.3.2 (Eq. 1 and Eq. 2). If this is the case, the server shall return `ver_token` computed as in clause C.2.3.2 (Eq. 1). If the server does not understand the `TLMSP_delegate` extension, no token will be present as defined in clause C.2.2, step 6, and even if a token was included, the server would ignore it.

The middlebox `Mn` (acting as TLS server `S'`) shall verify that the `ver_token` received from the server, together with the `token` received from the client, satisfies relation (Eq. 2) of clause C.2.3.2. If so, it shall return `ver_token` to the client. The client shall then verify that `ver_token` satisfies (Eq. 1) of clause C.2.3.2 and if not, it shall close the connection.

NOTE: This proves to the server that middlebox M_N is authorized by the client to act as a proxy. It also proves to the client that the middlebox is connected to the correct server.

The hs_id fields of the token and TLMSP_delegate extension may be used to associate a server with the delegated session.

C.3 Middlebox security policy enforcement

C.3.1 General

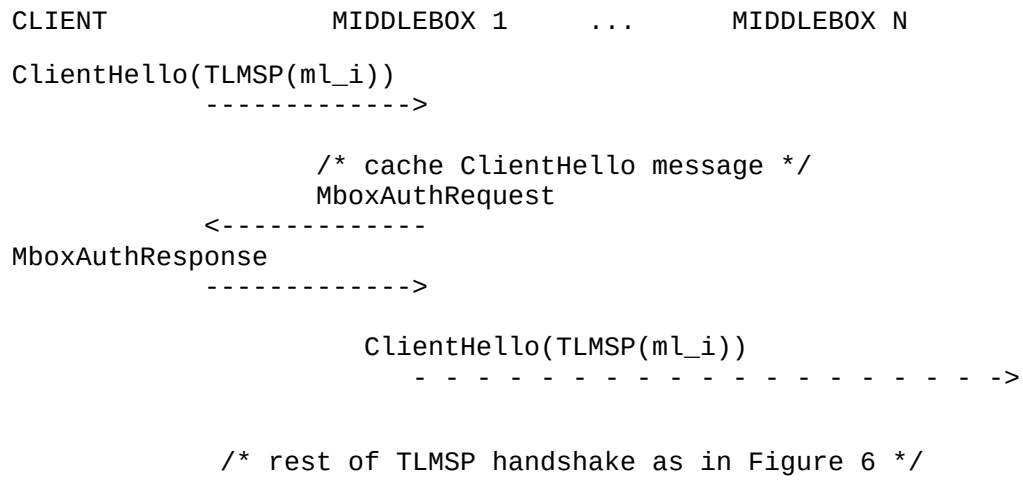
A middlebox can enforce a security policy with respect to allowing connections to traverse it.

EXAMPLE: An enterprise gateway between an enterprise intranet and the rest of the Internet, protecting against data leakage.

In such situations, it is undesirable to allow any traffic to pass the policy enforcement in the middlebox until the client authenticity has been established. When the signalling flow of clause 4.3.1 (Figure 6) is used, the client authenticity is not established with certainty by any middlebox until after the client has sent the ClientHello to the server. An unverified insider could leak information to the server by embedding the information in various information elements of the ClientHello.

To implement such policy enforcement, a verifiable handshake with the first middlebox shall be completed before forwarding the ClientHello to external network(s). This may be done by using the signalling flow in Figure C.3.

If the client proposes to use alternative (non certificate based) cipher suites according to annex B, it is still assumed that the client also supports standard, certificate based methods, which is needed in the exchange with the first middlebox.



NOTE: The symbols *, o and x are defined in the same way as in Figure 6.

Figure C.3: Handshake, alternative policy enforcement flow

On reception of the ClientHello at middlebox M1, M1 responds with an MboxAuthRequest comprising a signature over (parts of) the ClientHello (including the nonce), thereby authenticating M1 to the client. If the signature verifies, and the client wishes to proceed, the client responds with an MboxAuthResponse message, which shall contain a client certificate and a signature over M1's MboxAuthRequest message. The certificate provided at this point may differ from the certificate the client uses in later Handshake messages. M1 now verifies the signature. If it fails, it shall respond with an access_denied alert and terminate the connection.

Otherwise, after the client has been authenticated, M1 forwards the ClientHello and the protocol completes as in Figure 6, except that when computing the verification messages as part of Finished and MboxFinished, the MboxAuthRequest and MboxAuthResponse messages shall both be omitted. If M1 is transparent, it may not have been included in the client's original middlebox list. In this case, middlebox M1 may choose to not actually participate in the to-be-established TLMSP session.

NOTE: The security policy determination is bound to the `ClientHello` based on the signatures included in the two messages following the `ClientHello`, and the correct forwarding of the `ClientHello` will eventually be verified by the `Finished` messages (and thus the security policy decision is bound to the session).

As noted in clause 4.3.1, this flow could encounter problems which is to be considered before using it.

C.3.2 Message formats

The formats of shall be as follows:

```
struct {
    Certificate mbox_cert;
    CertificateRequest cr;
    digitally-signed struct {
         CertificateRequest cr;
         ClientHello client_hello;
    } verify;
} MboxAuthRequest;
```

The field `mbox_cert` is a `Certificate` message containing a middlebox certificate and the field `cr` is a `CertificateRequest`, where each shall follow the formats as defined in IETF RFC 5246 [1]. The field `verify` shall contain a signature made with respect to the public signing key of the `mbox_cert` field, over the `CertificateRequest` being sent and the received `ClientHello` (with the value substitutions described in clause 4.3.9.1 applied).

```
struct {
    Certificate client_cert;
    CertificateVerify ver;
} MboxAuthResponse;
```

The contents shall follow IETF RFC 5246 [1], with the signed payload of the `ver` field comprising a signature over the received `MboxAuthRequest`. The certificate included in `cert` may be a different one from the one used in the later `Handshake` messages.

NOTE: There are no entity IDs in the `MboxAuthRequest` and `MboxAuthResponse` messages. In consideration of all of the possible scenarios, such identities do not seem to convey any generally useful information. In particular, the middlebox `M1` implementing the security policy may be a transparent middlebox that was not in the initial middlebox list in the `ClientHello` (and thus would later invent its own ID if it participated in the session), and further, could be a middlebox that may not actually include itself to the TLMSP session. In such case the concept of entity ID is not meaningful.

Annex D (informative): Contexts and application layer interaction

D.1 Application layer interaction model

Central to TLMSP principles is the ability to partition data from the application layer into contexts associated with certain privileges, delegated to middleboxes. This requires an intelligent agent that can extract parts of the application data for which delegated access rights are granted.

EXAMPLE 1: In a simple cases, such as header vs payload distinctions, this is straightforward and a generic agent could be built into TLMSP.

In other cases, the situation is more complex and only the application itself would typically have sufficient knowledge about the sensitivity of certain parts of the data. In this case, the agent would be built into the application itself, and thus all such applications would need to be modified to make use of TLMSP.

The layered model in Figure D.1 could be used to allow support for TLMSP in a wider range of applications (rather than limited to naïve context concepts such as in example 1) without applications needing to be re-written for TLMSP usage.

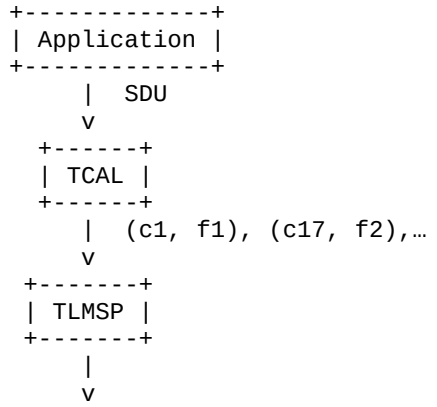


Figure D.1: TCAL concept

Here, a TLMSP Context Adaptation Layer (TCAL) is provided between the application and TLMSP. The principle of operation of TCAL is to take the application layer SDUs, match them against a suitable context model, split according to those contexts and deliver data to TLMSP as fragments, tagged by their context. On the receiving side, a mirrored TCAL layer receives decrypted fragments from TLMSP together with a context identifier, and re-assembles these into data readable by the application.

EXAMPLE 2: TCAL could use templates or plug-ins tailored to a specific application or a specific type of applications.

D.2 Example context usage

How contexts are configured and used is out of scope of the present document, as to define contexts would require knowledge of the application and use cases. However, some suggestions for how contexts can be used are given in the present clause.

Different contexts could be associated with different, distinguishable parts of the data fragments generated by the application.

EXAMPLE 1: One context is associated with headers or metadata, and another context is associated with payloads. Middlebox read access is granted to the headers and no access is granted for payloads. This is implemented through two separate contexts. If write access is granted to payloads, it is write access would also be granted for the headers, as the length of messages could change as a result of middlebox processing.

EXAMPLE 2: A highly trusted middlebox is allowed to insert application data into the server-client flow; another middlebox in the connection only has (partial) read access. One context is allocated so that only this highly trusted middlebox can perform insertions. Data inserted by the middlebox would be recognized by the corresponding context identifier in the container headers.

EXAMPLE 3: A middlebox has access to the downstream server-client flow, but not to the upstream flow from client to server. Two contexts are used: one for each direction. Generally, one middlebox is granted read access to the upstream flow and another middlebox is granted write access to the downstream flow.

Annex E (informative): Security considerations

E.1 Trust model

During the development of TLMSP, a lot of experience has been gained regarding the trust model and how different assumptions on fair-play and honesty of both middleboxes and endpoints can lead to unanticipated security issues. The present document reflects the summary of gained knowledge and the technical specification mitigates several attack vectors that were not anticipated by the original mcTLS protocol, nor by earlier versions of TLMSP.

Middleboxes have many diverse applications and it is not possible to define a one-size-fits-all trust model, as it depends on the use case.

The client and server trust each other not to attempt to include other entities beyond those middleboxes agreed in the TLMSP handshake phase. Middleboxes cannot take part in the session without obtaining the required key material, and said key material is provided in two halves, from each endpoint. Once the handshake is completed, both endpoints will know the secret contribution provided by the other endpoint. At this point, no protection can be provided against an endpoint attempting to share additional key material with additional entities after completion of the handshake. This is true for all cryptographic protocols: the protocol itself cannot ensure that the endpoints do not leak the key material. On the other hand, the `TLMSPKeyConf` mechanism of TLMSP can be used to prove the converse: that no middleboxes have been omitted from gaining intended access to keys.

In general, client and server can have varying trust levels of each other. One foreseen use case of TLMSP is for the middlebox(es) to filter out unwanted or malicious content transmitted from the server, or, from the client (e.g. DoS attacks against the server). TLMSP, as with all protocols, can provide cryptographic integrity and authentication of content, but this does not guarantee that the content in general is safe, trustworthy, or correct. Indeed, when middleboxes are provided by a trusted entity, such an enterprise or a MNO, trust assumptions can be moved from plural untrusted entities on the Internet to a smaller number of trusted entities under the control of one's own organization.

Authorized parties, including middleboxes, are trusted to handle the access privilege level they have been granted (read, delete, or write) to specific contexts. This is cryptographically assured through authentication, key management, and an assumption that endpoints and middleboxes do not leak keys to other, unauthorized entities. Middleboxes with write access can also perform insertions and deletions; it is assumed that middleboxes do not exploit this for malicious purposes, such as denial of service or replay attacks, as they are under the control of a trusted entity. If a middlebox exhibits such behaviour, using the hop-by-hop MAC could be used detect such attempts, as described in clause E.3. This can limit the trust assumptions to only the last middlebox on path and will also allow detection of "cheating".

Parties are assumed to participate in the protocol fully; they do not drop out or refuse to forward messages passing them. However, some measures are still taken to prevent certain types of "selective refusal" by middleboxes. One middlebox could attempt to selectively refuse another middlebox its granted access rights by selectively choosing to not forward `TLMSPKeyMaterial` to that specific middlebox. This would however in TLMSP be detected by the `TLMSPKeyConf` and the `MboxFinished` messages (though not revealing the identity of the refusing middlebox).

Another example of such refusal was discovered during the development of TLMSP. At a point in time, a separate delete access right was included, that falls between basic read access and the full write privilege level. It was however realized that this delete access right could, without consideration, imply that a dishonest middlebox with delete access to some message could actually use this capability to unnoticeably delete other messages, even if the middlebox has no access at all to those other messages. The same analysis would of course also have applied even if a separate delete access right had not been introduced: a malicious writer middlebox could exercise the delete access right included as part of its write privilege level to achieve the same effect. This is a form of privilege escalation attack which is more severe than mere denial-of-service if the attack would be allowed to go unnoticed, and is described in more detail (including mitigation) in clause F.3

Many middlebox solutions focus on the need for endpoints to trust middleboxes.

EXAMPLE 1: A main feature of TLMSP is to ensure endpoints can include and refuse specific middleboxes, allowing endpoints to authorize the access for each middlebox.

A common theme in discussions is that the middleboxes are by default the only potential abusers to worry about. There could, however, also be a need for middleboxes to trust and authorize the endpoints, and so authentication of the client by the middleboxes is present but optional in TLMSP. When this option is not used, the trust model changes. It now generally becomes necessary for middleboxes to trust the server fully. If the server is not fully trusted, the server could be colluding with unauthorized and untrustworthy clients, allowing clients to benefit from middleboxes' services. Omitting client-to-middlebox authentication is used only after careful consideration.

In most situations, it is not in the interest of endpoints to weaken the security on purpose, and attacks based on such principles fall outside trust models that are typically relevant.

Cases exist, such as in enterprise environments, where the endpoints only have one path, via middleboxes, and are forced to either accept the middleboxes fully or to not communicate. In such cases, it could become attractive for malicious endpoints to circumvent middleboxes, either by bypassing them or undoing effects of certain middleboxes. Clause E.3 discusses one specific case when the server is malicious.

In general, both the server and client could collude maliciously to affect the cryptographic keys obtained by a middlebox, resulting in incorrect or weak keys. In this case, the `TLMSPKeyConf` feature is not effective when both endpoints are malicious.

EXAMPLE 2: Due to the way that keys are cryptographically derived from the distributed key contributions, it would be highly unlikely that an incorrect reader MAC key verifies the packets (that were protected by a different key) as being valid; therefore such an attack is likely to be detected by middleboxes.

NOTE: It is also highly unlikely that a middlebox has obtained an incorrect reader decryption key, while the middlebox still is able to verify reader MACs (since MAC verification is done after decryption).

To mitigate risk of weak keys, the key derivation in TLMSP is modified compared to mcTLS [i.1] so that also middleboxes contribute to the entropy of the keys. Observe however, that when *both* endpoints are malicious and assumed to collude, they would be able to insert any data into the TLMSP connection without breaking any cryptographic primitives. The MACs (in any form) do not help in this case, since the destination endpoint is assumed malicious and will not care about the MAC value validity. In this extreme trust model, an extension of the TLMSP protocol with third party (or publicly) verifiable audit records would be necessary, but is out of scope of the present document.

E.2 Cryptographic primitives

E.2.1 General

In TLMSP, it is the client and server that propose and select the cipher suite. In comparison to back-to-back proxy approaches (selecting cipher suite per-hop) this has the great advantage that client and server remain in charge and mitigates the risk that one hop uses a transform that the endpoint would normally not accept. To the extent possible, TLMSP seeks to use the same cryptographic primitives as in TLS. To avoid the risk of failed connections due to lack of cipher suite support in one of the middleboxes, a mandatory-to-support AES-GCM cipher suite is defined, which is cryptographically equivalent to the TLS counterpart.

The PRF used for key derivation and the default HMAC_SHA256 based primitive is the same as used in IETF RFC 5246 [1]. The inputs to the PRF are however different TLMSP due to the inclusion of the list of the full set of entity identities. This binds the key material to a specific session, making it less probable that key material could be re-used in a future connection with different middleboxes.

The pre-defined cipher suites of annex A are based on state-of-the-art cryptography (also used in TLS cipher suites), but the IV formation has been changed to use partly explicit IVs. First, the nonce or cryptographically derived `fixed_IV` part of the IVs is expected to contain enough entropy to protect against time-memory trade-off attacks. Re-use of IV with the same key can compromise confidentiality, in particular for stream ciphers,[i.6]. To ensure IV-uniqueness for a given key, all predefined transforms include both `e_id` and the sequence number (the global, context-independent one) in the IV. It is impossible for these two values to both collide for two different messages since either `e_id` will differ (for different authors), or, the sequence numbers involved will be different (for two message units from the same author). IV reuse may also be catastrophic for the MAC in AEAD transforms. Note that the fixed part of the IV is context-independent. When a middlebox modifies or inserts a message, due to the fact that both the middlebox local sequence number and entity ID are also included in the IV, IV collisions are avoided for containers associated with the same context. The deleter and writer MACs can never re-use information from the IV of the reader MAC, and moreover, the reader, deleter, and writer MAC keys are all different. Further, for containers belonging to different contexts, their keys differ. The only case where the same key and IV could be used for two different message units is when generating the hop-by-hop MAC and/or when generating the writer MAC for an audit container or writer MAC for an alert: in all these cases, the key used is independent of the context, but unique to the (author,destination) entity pair. Thus, the only case when the same key could be used for two (or more) of these MACs is therefore on the last hop, from the last middlebox to the destination endpoint. Note that in this case, also the author may be the same for two different message units, so the fact that the author entity ID is part of the IV does not ensure IV collision avoidance. However, in this case, the IV of a hop-by-hop MAC and writer MAC can never be the same since the MAC-type is included in the IV. (Indeed, no two MACs of different type can ever have the same IV, since the MAC-type is included in the IV.) Therefore, the only remaining issue is if the IV of a writer MAC of an `Alert` message could be the same as the IV of a writer MAC on an audit container (on the last hop). But again, this is impossible: the alert and audit containers (even if associated with the same context) need be carried in two different containers, and thus the sequence number part of the IV will differ by at least one.

Use of ephemeral (standardized) Diffie-Hellman cipher suites offers forward secrecy.

When using AEAD transforms, the computation of the reader MAC value is integrated with the encryption. Computation of other MAC values (deleter, writer, and hop-by-hop MAC values) uses a separate application of the MAC-part of the AEAD as defined in annex A. As mentioned in annex A, this means that only AEAD transforms that allow such separation can be used in TLMSP. Many AEADs require nonces, both when computing the combined AEAD transform as well as when computing stand-alone MACs. These nonces are used only once for a given key. In TLMSP, the MAC-value computations all uses distinct keys, thus the same nonce can be used for all reader and writer MAC values, as long as no two MAC-value computations of the same type use the same nonce. For each key, the corresponding nonces are, for the pre-defined transform of clause A.2, formed by an exclusive-OR of a per-message unique IV-part (the uniqueness following from the discussion above) with a fixed, pseudo-random value derived from the master key. The exclusive-OR preserves this per-message unqinuess. Further, since the only real source of nonce-entropy is cryptographically derived from the master key there is no threat that a malicious end-point could somehow affect nonce reuse or nonce entropy.

TLMSP supports unauthenticated cipher suites, as well as cipher suites based on non-Diffie-Hellman key exchange mechanisms and non-signature based authentication. However, instead of defining a complete set of additional cipher suites for this purpose, TLMSP uses indicators in the Hello messages that determine whether "standard " or "alternative" cipher suites are to be used.

E.2.2 Handshake verification

The verification of the handshake (the keyed hash in the `Finished` messages) is, compared to TLS, more complex as not all entities share the same view of all the messages. Some information added by one middlebox could be unavailable to all other middleboxes. The verification has therefore been split in two stages: the first being an end-to-end verification between server and client based on those message elements that are common in both endpoints. When possible, information specific to the middleboxes is also included to create a cryptographic binding between end-to-end and middlebox specific information. Secondly, a set of pairwise `MboxFinished` messages are exchanged to verify parts of the handshake which uses local information, usually known only in one of the endpoints and one of the middleboxes. While there are no pairwise verification messages between pairs of adjacent middleboxes, modifications of such inter-middlebox `Handshake` messages (key exchange messages) will be detected at the latest when the first protected `Alert` or `Application` protocol message unit (of any context) passes the corresponding pair of adjacent middleboxes: the hop-by-hop MAC will fail as a consequence of the middleboxes having derived different keys.

An explicit verification of the middleboxes' reception of the key material contributions is provided for the client, whereas the server obtains an implicit verification via the `Finished` message as defined in clause 4.3.9.

In addition to this, by modifying the `ServerKeyExchange` as defined in clause 4.3.10.1 and changing the order of `CertificateRequest` and `ServerKeyExchange`, TLMSP protects the client against unauthorized harvesting of its certificate(s) and also enables detection of 3rd party modifications to the middlebox lists as early as possible, Modification of middlebox lists would still be detected even without these changes, but only at the end of the handshake.

E.3 Protection against mcTLS attacks

The original mcTLS proposal suffers from a vulnerability by which a malicious endpoint, either the server or client, can undo filtering operations performed by a middlebox [i.3].

EXAMPLE: A middlebox detecting malware content from a server could be ineffective at removing this malware if the server can access the remaining path between middlebox and client and re-insert the malware.

This is possible as the server generally has access to all keys used by middleboxes to authenticate their inbound/outbound containers. Similar issues exist also when considering two middleboxes with write privileges, where one of them not is malicious. In fact, the problem in the original mcTLS protocol is bigger than just the problem with a malicious server: any party, not even knowing a single cryptographic key, can copy an mcTLS message from one hop, and inject them on another hop. None of the MACs in mcTLS (incldluing the endpoint MAC) can protect against such attacks. Moreover, once a single modification has been done by any middlebox, the value of the endpoint MAC is heavily reduced since nobody is able to tie the endpoint MAC to any specific change made by any specific entity.

A previous draft version of the present document used a so called forwarding MAC, computed by middleboxes using a key known only to the middlebox and the downstream endpoint, aiming to thwart this attack. The problem with this (as well as with the end-point MAC of the original mcTLS specification) is that *at most* the destination endpoint would be able to verify such a MAC. Since no other middlebox knows the corresponding MAC key, there is no way for a middlebox to distinguish TLMSP messages that are really coming from the upstream neighbour from messages that have been copied from another hop, further upstream. Thus, such forwarding MACs do not prevent a middlebox from forwarding messages that have not been seen and processed by *all* upstream enitities. The endpoint would be able to verify the forwarding MAC, but only the forwarding MAC added in conjunction to the *last* modification to the message. This is because any subsequent modification of a message destroys the cryptographic link to a MAC that was made on an earlier version of the same message. Thus, any modification that was made further upstream, may be undone by a reader or even any third party attacker (without any knowledge of keys), and will remain unverifiable, even to the destination endpoint. Therefore, the forwarding MAC mechanism did not meet its intended purpose. Similarly, deletions (e.g. of messages that contain malware), could be undone by any party.

TLMSP instead addresses this attack by requesting that middleboxes perform an additional local check on inbound containers and that they also authenticate their outbound containers by a key only known to the next-hop endpoint, via the hop-by-hop MAC. Through this approach, each receiving entity will be able to verify that it receives containers that were unaltered from when they left the previous middlebox. The obtained end-to-end verification is implicit: it implies that each middlebox received and had opportunity to act on authenticated containers, but it does not prove that the middlebox performed the "right" action. This is left as an assumption of the trustworthiness of the middlebox.

For similar reasons, it is necessary for writer middleboxes to re-compute writer MAC values (using a new IV), even when they did not perform any modification. If not, a reader or deleter middlebox could escalate its privilege to"undo" modifications done by upstream writer middleboxes in a similar way as described above. Likewise, deleter middleboxes need to re-compute the deleter MAC on a container even if they choose not to delete it. Refer to clause F.3 for security considerations on sequence number usage.TLMSP authors have noted one additional issue and one observation on the security properties of the original mcTLS specification. The issue has to do with robustness and was a reason that led to adding the feature of a hop-by-hop MAC, as described in clause F.4.

Another observation is that mcTLS (and TLMSP) distributes key-shares to middleboxes before the handshake is complete. In particular, this distribution occurs before the verification that the selected cipher suite is not subject to an active downgrade attack. This could be argued as sub-optimal, but two arguments can be made in favour of not addressing it:

1) This attack would be detected at the later completion of the handshake, which still happens before the keys protected by the cipher suites are used.

2) If the handshake was modified, allowing complete verification of the selected cipher suite before distributing key-shares, it would no longer be possible to bind those key-shares into the handshake verification.

Therefore, TLMSP authors leave this as an observation.

E.4 Inter-session assurance

Even if the original content stored in a cache was delivered via TLMSP and was thoroughly inspected by some middlebox before it was stored, TLMSP does not propagate assurance information from one TLMSP session to another. A different client that later downloads the cached content does not automatically obtain any assurance that the content was previously inspected and is free of malware. Indeed, in a caching use case, later downloads of the same content could use TLS instead of TLMSP. On the other hand, the audit mechanism of TLMSP could be used to provide evidence that content is trustworthy. In this case, audit records would be constructed to be universally and publicly verifiable.

E.5 Use of the default context zero

All entities have read and write access to context zero, motivated by a common need to read and/or insert messages into this context. Thus context zero does not have the same separation of privileges as the other contexts. The present clause analyses potential issues caused by this lack of privilege separation.

During the initial phases of the handshake (before `ChangeCipherSpec` has been issued), context zero and all other contexts are not protected in any way. This is identical to the situation for TLS 1.2, except that in TLMSP, there is further no usage of sequence numbers at this stage. When security has been activated, both endpoints and any middlebox can generate (valid) messages with respect to context zero, but no third party to the connection can do so. The only messages protected by context zero are `Handshake` messages sent after `ChangeCipherSpec` and `Alert` messages related to context zero itself.

The alerts for context zero are, in addition to the context zero reader/writer MAC-values, also using hop-by-hop MAC values of the originator and can therefore be authenticated as originating from a specific source (endpoint or middlebox). Therefore, whether to trust and act upon the alert is purely an issue of whether the entity that generated the alert can be trusted. This is identical to the trust model required when using point-to-point TLS. (Recall that the trust model of clause E.1 assumes that middleboxes follow the specification and do not drop alerts by other entities.)

A handshake exchange, whether protected by context zero keys or not at all, always ends with a set of `Finished` messages between each of the endpoints and the set of midddleboxes, authenticating the handshake exchanges by pairwise keys (known only to two of the entities). This is therefore not dependent on the common, shared context zero keys (though the context zero security further protects from third party eavesdroppers). Recall also that the most critical part of the handshake, the transfer of (new) key material contributions to middleboxes is always protected independently of the record layer (using pairwise keys) as defined in clause 4.3.7.

Finally, where `ChangeCipherSpec` messages can occur in a handshake are only at the points of the handshake defined in Figure 6. Such commands, if spoofed by middleboxes at other points, can be ignored without issue. The message does not carry any information other than to activate the pending state. Additionally, this command is always followed by a (set of) verification `Finished` message(s), using the pairwise keys.

E.6 Removal of middlebox insertions

TLMSP adds functionality for middleboxes to insert content that does not originate from an endpoint. Under the assumption that the inserted content is there to improve security and/or improve the service experience, removal of such insertions needs to be interpreted as an attack on the protocol.

EXAMPLE 1: A middlebox inserts cached content, avoiding need to repeatedely fetch the same content from a server. When combined with the middlebox deletion feature, the middlebox replaces an outdated or infected file with an updated one.

When this feature is used, it is intended to improve security and/or service delivery; therefore the impact of blocking these insertions go beyond denial-of-service prevention. Just like normal TLS use, nothing can be done if an attacker is able to drop packets.

EXAMPLE 2: In TLMSP, an attacker starts to drop packets as soon as the first insertion by a middlebox is done. The attacker allows packet flow to resume as soon as the middlebox has done the last insertions. TLMSP can not prevent this attack, but can help to detect it. Eventually, some additional packets will be sent by the other endpoint or a `close_notify` message will be sent. When the middlebox adds a hop-by-hop MAC value to this message, it will be done with a different sequence number to that expected by the endpoint; therefore, verification of this MAC value will fail.

The general scenario behind Example 2 is in reality somewhat more complex if one considers also the possibility that the attacker could actually be another (malicious) middlebox who has been granted at least some level of privilege to the data protected by the TLMSP session. Such a scenario is discussed in more detail clause F.3. It is however stressed already here that TLMSP does provide protection also against attacks of this more advanced type.

E.7 Removal of support for renegotiation

TLMSP according to the present document does not support renegotiation due to potential threats to middlebox operations that seem to require additional mechanisms to be handled securely.

During a renegotiation handshake, application data protected by a previously established cryptographic state could possibly be interspersed with `Handshake` messages associated with the renegotiation. This could defeat the function of middleboxes: a middlebox cannot buffer `Application` protocol containers and let `Handshake` messages pass them as that would break sequence number handling. Therefore, a middlebox could be forced to a make a decision to let application protocol messages pass, while having been able to examine further application messages might have led the middlebox to block the `Application` protocol messages. Further, letting `Handshake` messages pass buffered containers could lead to problems with buffered containers winding up getting delivered only to be processed with the wrong cryptographic state. An attacker with control of an endpoint could attempt to bypass middlebox functionality this way, by interspersing payload with `Handshake` messages as required to defeat the middlebox functionality. Even without concern for such attacks, there is in general need for a cooperation mechanism between TLMSP and the application layer protocol to avoid timing a renegotiate such that it can defeat middlebox functionality. Specification of such a cooperation mechanism is however not in scope of the present document and therefore renegotiation is not supported.

Annex F (informative): TLMSP design rationale

F.1 General

This clause provides background material about design considerations when modifying mcTLS to produce the TLMSP specification.

F.2 Containers

A driver for the original mcTLS protocol was to provide fine grained access control to an encrypted session; contexts could be used to provide different levels of access control to different parts of the application data.

EXAMPLE: For website whitelisting/blacklisting, granting a middlebox access to HTTP headers without granting access to the HTTP body.

To do this, the HTTP headers and HTTP body could belong to different contexts, protected by different keys. The middlebox responsible for the whitelisting/blacklisting would only require access to the HTTP header context but not the HTTP body context. Whilst the method by which an application chooses to split the data content across different contexts/containers is not part of this protocol specification, this example does highlight some potentially undesirable features in the original mcTLS design.

The first is that data in one context can relate to data in another context (HTTP headers and body are clearly related to each other) and therefore a middlebox could need simultaneous access to data from more than one context to carry out its function. It would be desirable to have these contexts delivered in one TLMSP record, even though they correspond to different contexts. However mcTLS specified that the contexts be transmitted in separate records.

The second undesirable feature is related to another goal of TLMSP: to enable middleboxes to optimize traffic flow under varying network conditions. To that end, direct cloning of the TLS record format, as is done in mcTLS [i.1], would have drawbacks. Fragmentation could be done so that each TLMSP record contains data associated with precisely one TLMSP context, according to a specific access policy for the middleboxes. Thus, use of contexts implies a specific *maximum* fragment size; this size could be much smaller than the 16 kB maximum record size specified for standard TLS, meaning data is transmitted in smaller chunks, even when larger chunks are preferred for network performance.

It should be noted that endpoint congestion control techniques can be defeated by the presence of middleboxes, a problem which exists in general with the use of middleboxes, and particularly with the terminate-and-reoriginate approach. TLMSP explicitly adds middleboxes to the model but does not define or provide mechanisms to address interworking with congestion control methods.

F.3 Sequence numbers and re-ordering/deletion attacks

A straight-forward adaptation of TLS sequence number handling does not work in a protocol which allows the middleboxes to, independently of each other, delete or insert messages into the session.

EXAMPLE 1: There is a chain of middleboxes entities, `e[1], e[2], …` between client `C` (identified as `e[0]`) and server `S` (identified as `e[n]`). Assume an attacker can access and control the transport network somewhere after middlebox `e[j]`.

Two middleboxes, `e[i]` and `e[j]`, `j < i`, each insert a message `m[i]` and `m[j]` at times `T[i]` and `T[j]` respectively, where `T[i]` and `T[j]` are "close". At some point, `m[i]` and `m[j]` will reach the point in the network where the attacker is present; the attacker can now store/buffer `m[i]` and `m[j]` and forward them in any order it chooses without detection.

The example above shows that context-specific sequence numbers (alone) are insufficient, as they only provide a binding to the inter-message order for messages from different contexts. Thus a global sequence number is required.

On the other hand, a single global sequence number is also insufficient as illustrated by the following attack.

EXAMPLE 2: A middlebox entity, `e`, has delete access to container associated with context `c1`. It has no access whatsoever to context `c2`. It is then possible for `e` to drop (i.e. delete) containers associated with context `c2` and replace them with delete indications associated with context `c1`.

For entities located downstream from `e` in Example 2 above, the delete indications associated with context `c1` will make the total number of containers that have passed `e` appear to be consistent: the dropped containers from `c2` will not be missed. It is only if/when `e` starts to forward containers from `c2` again that the attack will be detected due to sequence number mismatch (and associated MAC failure). Arguably, the attack is non-persistent in this sense, but the TLMSP design has nevertheless added a mechanism to mitigate this (and other) attacks.

Specifically, TLMSP counters this attack by:

a) using both global, context-independent sequence numbers as well as context-dependent sequence numbers; and

b) using the global sequence numbers as input to reader and hop-by-hop MAC; and

c) using the complete set of all context-dependent sequence numbers as inputs to all writer and deleter MACs.

Feature (a) is obviously a pre-requisite for features (b,c). By feature (c), then, the attack of Example 2 will immediately be discovered as soon as the malicious entity `e` allows any container (of any context) to be forwarded. The discovery will be made by the closest downstream entity who has at least delete- or write access to the context of the forwarded container. This is the best protection possible to attain, as one cannot expect that a downstream middlebox with only read access would be able to detect insider-attacks by middleboxes of higher privilege level. By feature (b), the attack of Example 1 will be thwarted.

F.4 MAC for synchronization purposes

The mcTLS protocol [i.1] on which TLMSP is based does not specify the use of a MAC for synchronization purposes. This is problematic for maintaining synchronization between entities in a connection and maintaining sequence numbers.

EXAMPLE: A middlebox `M` has neither read- nor write-access to a particular context, `c`. The endpoint sends a record associated with context `c`, and the record is processed with sequence number `s` at that endpoint. When this message passes `M`, will `M` increase its local sequence number?

If `M` does not, then when a context that `M` has access to is processed, the endpoint generating the message will process it with a sequence number `s+d`. However, `M` will use sequence number `s-1+d` (or lower).

If `M` does increase the sequence number to `s`, there is no way for `M` to know if the message was spoofed by an attacker since `M` cannot verify the authenticity of the message. `M` will have increased the sequence number so that it is too high when a later, authentic container is accessed.

NOTE: This is a problem also for the original mcTLS specification.

A potential solution to this would be to use independent, per-context sequence numbers. This would be a viable solution for the mcTLS protocol which does not allow insertions or deletions, but as discussed in clause F.3, this is not a sufficient solution for TLMSP; it leaves open attacks related to re-ordering of containers. This is the reason for introducing the hop-by-hop MAC which, besides preventing injection of messages from one indivudal hop to another, also serves as a MAC for synchronization purposes that all middleboxes can verify.

F.5 Removal of support for renegotiation

This is motivated in detail in clause E.7.

Annex G (informative): Mapping MSP desired capabilities to TLMSP

G.1 General

Following the framework of [i.5], the clauses below state which of the MSP Requirements that have been selected for the TLMSP Profile defined in the present document and provides conformance claims how each of the requirements are met. The column MSP/Profile Type contains information regarding the status of each requirement. The value before the "/" dash denotes whether the Template Requirement is mandatory in all MSP profiles (MM) or whether it is optional to certain profiles only (MO), according to [i.5]. The value after the "/" denotes the status of the requirement in the TLMSP profile defined in the current document and can have values Profile Mandatory (PM), Profile Optional (PO), Profile Not-applicable (PNA), or Profile Rejected (PR).

EXAMPLE: The presence of "MO/PM" in the MSP/Profile Type column means that the requirement is in general optional for MSP protocols, but is mandatory in the TLMSP profile. Obviously, any of the combinations "MM/PO", "MM/PNA" or "MM/PR" would be incompatible with claiming conformance to [i.5].

For the mandatory requirements and those optional requirements that have been selected for TLMSP, a conformance claim with motivation is provided in the last column. For requirement that are profile non-applicable or have been rejected, it is in the last column stated why rationale for why the requirement is not applicable or not included in TLMSP.

G.2 MSP Requirements - Data Protection

The present clause defines the TLMSP Data Protection Requirements, based on the MSP Template Requirements in clause 6.2 of [i.5].

Ref	Data Protection Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
E.DP.1	Endpoints shall protect confidentiality of sensitive data that they send.	MM/PM	Provided through selection of a cipher suite with non-NULL encryption. The predefined transform of annex A is mandatory to support. (NULL encryption may be supported but discouraged from usage.)
E.DP.2	Endpoints may add protection to externally visible characteristics of application data to protect confidentiality of sensitive information about application activity. (This is commonly referred to as Traffic Analysis Protection.)	MO/PR	Requirement has been rejected due to the excess overhead it would create.
E.DP.3	Endpoints shall protect integrity of application data.	MM/PM	Generally provided through selection of a cipher suite with non-NULL MAC. (NULL integrity only allowed for testing.)
E.DP.3.1	Endpoints shall protect application datagrams from modification in transit between authorized participants.	MM/PM	Achieved by reader- and hop-by-hop MAC.
E.DP.3.2	Endpoints may protect application datagrams from unauthorized modification by a middlebox.	MO/PM	Achieved by assigning separate contexts to parts of data and use of separate deleter- and writer MAC, as well as hop-by-hop MAC.
E.DP.3.3	Endpoints may protect the datastream from modification in transit between authorized participants.	MO/PM	See E.DP.3.1 (Due to hop-by-hop MACs, even an authorized middlebox can only modify data when the data passes the middlebox itself.)
E.DP.3.4	Endpoints may protect the datastream from unauthorized modification by a middlebox.	MO/PM	See E.DP.3.2. In addition inclusion of sequence numbers in MACs
E.DP.4	Endpoints shall protect sensitive information about session state from unauthorized disclosure, discovery, manipulation and creation.	MM/PM	Compliance via sub-requirement fulfilment as below.
E.DP.4.1	Endpoints shall protect the sensitive cryptographic state from unauthorized disclosure, discovery, manipulation and creation.	MM/PM	Left as an implementation assumption on endpoints.
E.DP.4.2	Endpoints may protect the application state from replay and pre-play of data.	MO/PM	Supported by the inclusion of sequence numbers in MACs.
M.DP.1	Middleboxes shall protect confidentiality of sensitive data that they send.	MM/PM	Cipher suite selection is under control of endpoints, middleboxes assumed to follow that choice.
M.DP.2	Middleboxes may add protection to externally visible characteristics of application data to protect confidentiality of sensitive information about application activity. (This is commonly referred to as Traffic Analysis Protection.)	MO/PR	See E.DP.2.

Ref	Data Protection Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
M.DP.3	Middleboxes shall protect integrity of application data.	MM/PM	Compliance via sub-requirement fulfilment as below.
M.DP.3.1	Middleboxes shall protect application datagrams from modification in transit between authorized participants.	MM/PM	Supported by reader-, writer-, deleter- and hop-by hop MACs.
M.DP.3.2	Middleboxes may protect application datagrams from unauthorized modification by a middlebox.	MO/PM	Supported by assignment of data to different context, and assumption that middleboxes do not collude maliciously.
M.DP.3.3	Middleboxes may protect the datastream from modification in transit between authorized participants.	MO/PM	See M.DP.3.1. In addition inclusion of sequence numbers in MACs
M.DP.3.4	Middleboxes may protect the datastream from unauthorized modification by a middlebox.	MO/PM	See M.DP.3.2.
M.DP.4	Middleboxes shall protect sensitive information about session state from unauthorized disclosure, discovery, manipulation and creation.	MM/PM	Compliance via sub-requirement fulfilment as below.
M.DP.4.1	Middleboxes shall protect the sensitive cryptographic state from unauthorized disclosure, discovery, manipulation and creation.	MM/PM	Left as an implementation assumption on middleboxes.
M.DP.4.2	Middleboxes may protect the application state from replay and pre-play of data.	MO/PM	Supported by usage of sequence numbers in MACs.
M.DP.5	Middleboxes may protect against protocol data fields being used as covert channels by validating the contents or otherwise. (This does not eliminate covert channels from externally visible characteristics such as timings and sizes.)	MO/PR	Requirement has been reject since obtaining assurance that all forms of covert channels are avoided is deemed too difficult to verify. For example, it is clear that information could be leaked via spoofed Hello messages, information embedded in certificates, etc.

G.3 MSP Requirements - Transparency

The present clause defines the TLMSP Transparency Requirements, based on the MSP Template Requirements defined in clause 6.3 of [i.5].

Ref	Transparency Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
E.T.1	Endpoints shall receive suitable knowledge of all middlebox identities.	MM/PM	Information about identity and purpose of middleboxes is available in middlebox certificates and the mandatory middlebox list extension. It is generally required to be able to authenticate all middlebox identities. If one endpoint proposes that a middlebox ought not to present certificate to the other endpoint, it is at the discretion of the other endpoint whether to accept this.
E.T.1.1	Both endpoints shall receive suitable knowledge about the identity of all middleboxes authorized.	MM/PM	In addition to E.T.1, an authorized middlebox needs to obtain key material from both

Ref	Transparency Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
			endpoints in order to gain access.
E.T.1.2	Endpoints may receive knowledge about the identity of all refused middleboxes.	MO/PM	Rejected middleboxes (including dynamically inserted ones) are available in the middlebox list extension.
E.T.1.3	Endpoints shall be able to verify or otherwise confirm that they have the same knowledge as the peer endpoint of all middleboxes' identities that are authorized.	MM/PM	Middlebox list extension included in verification hash at end of handshake and explicit signalling verifies that middleboxes have been given access to key material.
E.T.2	Endpoints shall receive knowledge of all middlebox permissions and knowledge of all security mechanisms for data protection.	MM/PM	Endpoints choose both cipher suite and define and grant access rights on a per context basis.
E.T.3	Each endpoint shall be able to verify or otherwise confirm that they have the same knowledge (of middlebox permissions and security mechanisms for data protection) as the other endpoint.	MM/PM	Verification as stated in E.T.1.3.
E.T.4	Endpoints may receive knowledge of the peer endpoint identity.	MO/PO	Compliance via sub-requirement fulfilment as below.
E.T.4.1	The initiator endpoint may authenticate or otherwise verify the identity of the responder endpoint.	MO/PO	Server authentication is strongly recommended.
E.T.4.2	The responder endpoint may authenticate or otherwise verify the identity of the initiator endpoint.	MO/PO	Client authentication is optional but recommended.
E.T.5	Endpoints may verifiably audit activity of middleboxes.	MO/PO	Compliance via sub-requirement fulfilment as below.
E.T.5.1	The destination endpoint may verifiably audit the activity of middleboxes.	MO/PO	Endpoints have option to configure middleboxes to send special audit containers, only verifiable between a specific middlebox and destination endpoint.
E.T.5.1.1	The destination endpoint may verify that data has transited and not bypassed each middlebox.	MO/PM	Under the assumption that middleboxes follow the protocol, this is supported by per-entity sequence numbers in MACs and hop-by-hop MACs.
E.T.5.1.2	The destination endpoint may verify whether a middlebox has modified data.	MO/PO	Supported by optional use of audit containers.
E.T.5.1.3	The destination endpoint may verify the full change history of received data.	MO/PO	Supported if all middleboxes are requested to send audit containers.
E.T.5.2	The sending endpoint may verifiably audit the activity of middleboxes.	MO/PR	Would require feedback signalling channel.
E.T.5.2.1	The sending endpoint may verify that data has transited and not bypassed each middlebox.	MO/PR	See E.T.5.2
E.T.5.2.2	The sending endpoint may verify whether a middlebox has modified data.	MO/PR	See E.T.5.2
E.T.5.2.3	The sending endpoint may verify the full change history of received data.	MO/PR	See E.T.5.2
E.T.6	Endpoints may verify or otherwise confirm that middlebox access and middlebox permissions have been granted or denied.	MO/PM	Supported by key confirmation messages.

Ref	Transparency Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
M.T.1	Middleboxes may receive knowledge of all middlebox identities.	MO/PO	Middleboxes would normally be able to verify other middleboxes' signatures as part of the handshake. However, some middlebox could be requested by an endpoint to not supply a certificate.
M.T.1.1	Middleboxes may receive knowledge about the identity of all middleboxes authorized.	MO/PO	See M.T.1
M.T.1.2	Middleboxes may receive knowledge about the identity of all refused middleboxes.	MO/PM	This is supported by the middlebox list extension which passes all middleboxes.
M.T.1.3	Middleboxes may be able to verify or otherwise confirm that they have the same knowledge as other participants of all middleboxes' identities that are authorized.	MO/PO	Supported by verification at end of handshake. The inter-middlebox verifications are obtained as the application datagrams start to flow (through hop-by-hop MACs).
M.T.2	Middleboxes may receive knowledge of all middlebox permissions and knowledge of all security mechanisms for data protection.	MO/PO	Cipher suite selection available to all middleboxes. There is an exception for middleboxes' signature algorithms in case one endpoint requests a middlebox to not provide a certificate to downstream entities.
M.T.3	Middleboxes may be able to verify or otherwise confirm that they have the same knowledge (of middlebox permissions and security mechanisms for data protection) as the other participants.	MO/PM	See M.T.1.3.
M.T.4	Middleboxes may receive knowledge of either or both endpoint identities.	MO/PO	Compliance via sub-requirement fulfilment as below.
M.T.4.1	Middleboxes may receive knowledge about the identity of the responder endpoint.	MO/PO	Server authentication is strongly recommended.
M.T.4.2	Middleboxes may receive knowledge about the identity of the initiator endpoint.	MO/PO	Client authentication optional.

Ref	Transparency Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
M.T.5	Middleboxes may verifiably audit activity of other middleboxes.	MO/PR	Deemed in general to be too costly.
M.T.5.1	Middleboxes may verifiably audit activity of other participants on received data.	MO/PR	See M.T.5.
M.T.5.1.1	Middleboxes may verify that received data has transited and not bypassed each middlebox.	MO/PNA	Supported for upstream middleboxes due to usage of sequence numbers and hop-by-hop MACs, but not generally applicable due to lack of feedback channel for downstream entities.
M.T.5.1.2	Middleboxes may verify whether another middlebox has modified received data	MO/PR	See M.T.5.
M.T.5.1.3	Middleboxes may verify the full change history of received data.	MO/PR	See M.T.5.
M.T.5.2	Middleboxes may verifiably audit activity of other participants on sent data.	MO/PR	See M.T.5..
M.T.5.2.1	Middleboxes may verify that sent data has transited and not bypassed each middlebox.	MO/PNA	See M.T.5.1.1
M.T.5.2.2	Middleboxes may verify whether another middlebox has modified sent data.	MO/PR	See M.T.5.
M.T.5.2.3	Middleboxes may verify the full change history of sent data.	MO/PR	See M.T.5.

G.4 MSP Requirements - Access Control

The present clause defines the TLMSP Access Control Requirements, based on the MSP Template Requirements defined in clause 6.4 of [i.5].

Ref	Access Control Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
E.AC.1	Only endpoints shall grant or deny middlebox access and middlebox permissions.	MM/PM	Assignment of contexts and key material in control of and defined by endpoints.
E.AC.1.1	Middlebox access shall be granted by at least one endpoint.	MM/PM	See E.AC.1.
E.AC.1.2	Middlebox permissions shall be granted by the same endpoint or endpoints that granted access.	MM/PM	Both endpoints involved to define per-context access rights.
E.AC.1.3	The profile may support multiple levels for middlebox permissions.	MM/PM	Read, delete, and write/modify/insert supported.
E.AC.1.4	Endpoints may authorize middlebox permissions per context.	MM/PM	Access rights assigned per context.
E.AC.1.5	Only endpoints shall deny middlebox access or middlebox permissions. (A middlebox, such as a cyber defence gateway, can still block the entire connection between suspected malicious endpoints.)	MM/PM	Only an endpoint has possibility to reject access rights proposed by other endpoint.
E.AC.2	The endpoint(s) that grant(s) access to a middlebox shall authenticate or otherwise confirm its identity before granting access.	MM/PM	Either explicit authentication of each middlebox or, if acceptable, relies on trust in that the other endpoint authenticates middlebox (to "otherwise confirm" is understood to rely on trust in other endpoint's authentication).

Ref	Access Control Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
E.AC.3	At least one endpoint shall choose all security mechanisms for data protection.	MM/PM	Endpoints negotiate cipher suites.
E.AC.4	Endpoints may grant middlebox access and middlebox permissions only through mutual agreement with the peer endpoint.	MO/PM	Middlebox list extension includes permissions. The whole list is mutually agreed.
E.AC.5	Endpoints may authenticate or otherwise verify the identity of all middleboxes whose access is granted by the other endpoint.	MO/PM	See E.AC.2 and E.AC.4
M.AC.1	Middleboxes shall authenticate or otherwise confirm any participant identity they use for an identity-dependent action. This action is not granting or denying access to an MSP connection, which shall fall within endpoint remit only (E.AC.1). (This stops a middlebox unlocking access to data or services for an identity that has not been checked by the middlebox.)	MM/PM	At the transport layer, and after the session is established, the only applicable identity dependent action is to ensure to only accept receiving datagrams from the upstream neighbour (via hop-by-hop MAC) and to only forward to the downstream neighbour. Generally, middleboxes cannot take identity dependent actions since they cannot in general verify which entity that was the most recent to modify/insert data. Considering identity-dependent actions related to the application layer, context mappings can be used to allow only certain entities to modify the content, which makes actions identifiable at the granularity of the group of entities sharing the same access rights. Moreover, the middlebox can tell if a participant is authenticated and could fulfil the requirement, assuming the middlebox interface supports combined access to both application layer information and MSP layer information.
M.AC.2	Middleboxes may authenticate or otherwise confirm participant identities.	MO/PO	Authentication is not mandatory. In general, not desired that all middleboxes would always confirm all other middleboxes' identities.
M.AC.2.1	Middleboxes may authenticate or otherwise confirm the initiator endpoint identity	MO/PO	See M.T.4.2
M.AC.2.2	Middleboxes may authenticate or otherwise confirm the responder endpoint identity	MO/PO	See M.T.4.1
M.AC.2.3	Middleboxes may authenticate or otherwise confirm all middlebox identities	MO/PO	In general, not desired that all middleboxes would always confirm all other middleboxes' identities. It is however supported in case all middleboxes provide certificates.
M.AC.3	A middlebox may know that its access has been withheld. (Meeting this requirement implies it is not possible to deceive a middlebox into believing it has access.)	MO/PM	Can be determined from middlebox list and/or lack of received key material message.

G.5 MSP Requirements - Good Citizen

The present clause defines the TLMSP Good Citizen Requirements, based on the MSP Template Requirements defined in clause 6.5 of [i.5].

Ref	Good Citizen Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
E.GC.1	Resource attacks that use an endpoint action or request shall have some attribution to the attacker.	MM/PM	During handshake, an endpoint may request/propose that one or more other middleboxes take part in the session, but participation is decided by the proposed middlebox. After handshake, middleboxes (with appropriate access rights) can distinguish message units originating from the end point from inserted message units and decide how to handle them. Attacks to other entities participating in the same TLMSP session are further attributed via the hop-by-hop MAC. However, pure DoS attacks using malformed packets (with incorrect MACs) cannot be attributed to a source.
E.GC.1.1	Any party being asked to expend significant resource due to an endpoint request, shall have some attribution of the request to the endpoint.	MM/PM	Supported if client authentication is enforced (server authentication is as discussed strongly recommended).
E.GC.2	An MSP profile shall not provide a significant amplification factor for a resource attack that uses an endpoint action or request.	MM/PM	No sources of amplification have been identified.
E.GC.2.1	Where an endpoint sends MSP protocol messages that request a significant amplification factor on resource expenditure, then one of the following two things shall happen: either the recipient is not forced to accept the request or the requesting endpoint expends commensurately amplified resource as a consumer of the result.	MM/PM	No sources of amplification have been identified.
M.GC.1	Resource attacks that uses an endpoint action or request shall have some attribution to the attacker.	MM/PM	See E.GC.1.
M.GC.1.1	Any party being asked to expend significant resource due to a middlebox request, shall have some attribution of the request to the middlebox.	MM/PM	Middleboxes cannot request resources from other entities and cannot effect which type of processing that is needed at other entities. A middlebox can however request another middlebox to take part in the session, but participation is decided by the other middlebox.
M.GC.2	An MSP profile shall not provide a significant amplification factor for a resource attack that uses an endpoint action or request.	MM/PM	See E.GC.1.
M.GC.2.1	Where a middlebox sends MSP protocol messages that request a significant amplification factor on resource expenditure, then one of the following two things shall happen: either the recipient is not forced to accept the request or the requesting middlebox expends commensurately amplified resource as a consumer of the result.	MM/PM	A middlebox that was requested to join a session by another middlebox can decline to do so.

Ref	Good Citizen Template Requirement	MSP/Profile Type	Conformance and Selection Analysis
M.GC.3	Middleboxes may be able to drop out of a connection, without breaking or degrading the connection for other participants, to counter an attempted resource attack.	MO/PO	There is possibility to use the middlebox leave protocol for this purpose.

Annex H (informative): TLMSP compression issues

The current version of TLMSP does not support compression. If a future version of TLMSP is to support compression along the lines of TLS, a number of considerations need to be taken into account.

First, it can be noted that TLS compressed data is allowed to be 1 024 bytes greater than the uncompressed text and this could run into TLMSP container-length field limitations. Also, in TLS, plaintext is segmented into records, then compressed, with the limitation that the compressed data for each record is itself sent in a single record, and this is again allowed to grow up to 1 024 bytes.

If a TLMSP middlebox wants to edit data (insert/modify/delete), one faces the problems of breaking back-references and missing dictionary symbol redefinitions, so when modification is done, one also has to recompress the entire remainder of data for that context.

Finally, compression was removed in the recent TLS 1.3 update [i.8] because consensus was that compression belongs closer to the application layer, where relevant context can be taken into account to avoid/mitigate compression-based vulnerabilities.

Annex I (informative): IANA considerations

The TLMSP protocol has by IANA been assigned three values for new TLS extension types from the "TLS ExtensionType Values" registry defined in IETF RFC 8446 [i.8] and IETF RFC 8447 [i.9]. They are TLMSP (36), TLSMP_proxying (37), and TLMSP_delegate (38). See clauses 4.3.5 and C.2.3 for more information.

History

Document history		
V1.1.1	February 2021	Publication