Draft ETSI TS 103 523-2 V0.2.0 (2020-05)

TECHNICAL SPECIFICATION

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

Important notice

Copyright Notification

ETSI

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 the Middlebox Security Protocol, as identified below:

Part 1: "Capability Requirements";

Part 2: "Transport layer MSP, profile for fine g[illegible]ntrol";

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 [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 [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 [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 [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 them 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 does not 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 is 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 a modified version of the mcTLS protocol [i.1]. Whilst a large portion is unchanged 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.

This 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)".

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) (2[illegible], May 2[illegible], San Francisco, United States.

[i.4] IETF RFC 8404: "Effects of Pe[illegible]ve Encryption [illegible] Operations".

[i.5] ETSI TS 103 523-1: "CYBER; [illegible]ddlebox Securit[illegible]otoc[illegible] Part 1: Capabilit[illegible]quireme[illegible]".

[i.6] D. McGrew, D. Wing, Y. Nir, a[illegible] Gladstone[illegible]S Proxy[illegible]erver Ex[illegible]tion", d[illegible]t-mcgre[illegible]tls-proxy-server-01, IETF.

[i.7] "TLMSP reference implementation", available at 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 privileges: TLMSP provides four types of context access rights: "none", "read" , "delete", and "write", ordered by increasing privilege level. Here, a lower privilege level is defined as a strict subset of a higher:

- "none" implies no access,
- "read" implies read access only,
- "delete" implies read and delete privilege only, and,
- "write" implies full privileges – the ability to read, delete, modify (re-write), and insert.

These access privileges shall be mutually exclusive and each middlebox shall have precisely one of the above privileges per context. The terms "reader", "deleter", and "writer" shall refer to an entity having the associated privilege. Delete or write access implies read access. Write access implies delete access.

downstream entity: when sending a TLMSP message in a certain direction, a downstream entity is 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. 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 writer author corresponds to the entity with write privilege that was the most recent entity to process and forward the message. The deleter author, when defined, corresponds to the entity with at least delete privilege that was the most recent entity to process and forward the message. Deleter author is considered undefined for contexts when there does not exist any middlebox with explicitly granted delete privelige. A writer author is always defined and is considered to be the entpoint, if no middlebox with write access exists.

NOTE 1: TLMSP messages corresponding to context 0 never has a deleter author since this context never has explicitly granted delete access. Further, 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) who sent new message that 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.

(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, an upstream entity is any entity located topologically, relative to the receiver, in the direction from which the message is received, including the endpoint in that direction.

3.2 Symbols

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

A || B	concatenation of binary strings A and B

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

B_TID	GBA-defined B-TID value (obtained during GBA bootstrapping)
CBC	Cipher Block Chaining
CTR	Counter (mode)
CTXT_ID	Container Context Identifier
DH	Diffie-Hellman
DHE_DSS	Ephemeral Diffie Hellman Digital Signature Standard
DNS	Domain Name System
EV	Extended Validation
FIPS	Federal Information Processing Standard
FLAGS	TLMSP container flag field
GBA	Generic Bootstrapping Architecture
GCM	Galois Counter Mode
GMAC	Galois Message Authentication Code
HM	Hop-by-hop MAC
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
Ks_NAF	Network Access Function Key
LEN	Length
MAC	Message Authentication Code
MC	Middlebox key Confirmation message
mcTLS	Multi-Context TLS
ml_d	Middlebox list, extended by dynamically discovered middleboxes
MITM	Man In The Middle
MK	Middlebox Key material message
ml_i	Middlebox list (initial)
MNO	Mobile Network Operator
MSP	Middlebox Security Protocol
NAF	Network Application Function
NAF-Id	Network Application Function Identifier
NAF-key	Network Application Function key
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
Uint	Unsigned integer
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.


```
+--------+ +----+ +----+                                     +----+ +--------+
| Client | <--> | M1 | <--> | M2 | <--> ... <--> | Mn | <--> | Server |
+--------+ +----+ +----+                                     +----+ +--------+
```

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*. 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 message types, but not for the Handshake and ChangeCipherSpec protocols, which are associated with a default context called *context zero (0)*.

4.2.1.3 Record and container construction and processing overview

```
+------+---------+------------+~~~~~~~~~~~~~~~~~~+
| type | version | tot_length |s_id | fragment |
+------+---------+------------+~~~~~~~~~~~~~~~~~~+
  <-------- TLMSP header -------->
```

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

```
+------+---------+------------+-----+---~+~~~~+~~~+~~~~+
| type | version | tot_length |s_id | C1 | C2 |...| Cn |
```

```
+------+---------+------------+-----+---~+~~~~+~~~+~~~~+
 <-------- TLMSP header -------> <- container(s) ->
```

Figure 2b: TLMSP record format using containers (as used by Application and Alert protocols after server confirmation of TLMSP support). C1, C2, … Cn represents containers, whose format is defined in Figure 3

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.

Next may follow the s_id field which is a four octet identifier for the TLMSP session. The s_id shall not be present in the initial ClientHello. It shall be chosen by the server in a TLMSP extension in the ServerHello and shall be

added to all TLMSP records occurring during the lifetime of the session after the initial ServerHello. Middleboxes shall not modify the s_id.

tot_length shall define the total (octet) length of the record following the tot_length field itself, i.e. including the length of s_id. 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, i.e. those associated with context zero, a single fragment shall follow (see clause 4.2.7.1 for details).

For the Application message type (type = 0x17 [2]), each TLMSP container is included in a payload (that would correspond to the payload part of a standard TLS record).

```
+---------+-----+~~~~~~~~~~~~~~~~~~~+----+---------~+-~+--+--+
| ctxt_id |flags|m_info (OPTIONAL)|len |fragment |dm|wm|hm|
+---------+-----+~~~~~~~~~~~~~~~~~~~+----+---------~+-~+--+--+
  <----- container header ----->
```

Figure 3: TLMSP container format

A container consists of a header, a (data) fragment (including a reader MAC) and two or three additional MAC values, dm (conditionally optional), wm and hm. 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 len 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).

```
+---+---+---+-------------+
| I | D | A | RESERVED    |
+---+---+---+-------------+
Bit 15 14 13 12           0
```

Figure 4: flags field of the container header

The flags field is used for signalling purposes, 3bits 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" at the transmitting endpoint, but may be changed by an authorized middlebox to signal insertions or deletions, as defined in clause 4.2.3.1.2. The A-bit is used to signal that the current container has auditing content, as defined in clause 4.2.3.1.3.2.

If, and only if, at least one of the I- or D-bits are set to 1, the middlebox information field (m_info) shall be present, the format of which is defined in clause 4.2.3.1.2. 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 pre-pending it (to the length and data of the fragment) when computing the MAC.

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, and one hop-by-hop MAC (denoted hm) as defined by clause 4.2.7.2.3, where the two latter MACs shall be present. The deleter MAC is used to signal that a deleter middlebox has inspected the container and decided wether to forward it or not. The writer MAC field, similarly, is used to authenticate changes (or absence of changes) to containers in the form of deletions, modifications, and insertions. To distinguish which type of change (if any) has been made, the I and D bit of the flags field is used. The key used to generate the deleter and writer MAC shall depend only on the access privilege assigned to the entity that makes the change. The deleter MAC

shall be present if, and only if, at least one middlebox has delete privilege for the associated context.

EXAMPLE 2: If only read access is granted for a particular context, then only reader, writer and hop-by-hop 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 Record processing: cryptographic state and synchronization

4.2.2.1 General

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 required, the s_id field of the record header.

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, except for the s_id which is valid end-to-end. However, since the s_id is chosen by the server, the s_id alone is not guaranteed to uniquely identify a session locally at a middlebox or client. The s_id can still be of use in those cases where there are several TLMSP sessions following exactly the same path.

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 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. Clause 4.2.2.3 describes how to determine the entity identity of the message originator and author (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 Processing order

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).

When generating a new message unit, the reader MAC shall first be computed, taking the current sequence number defined in clause 4.2.2.3, and the header and plaintext data as input, as defined in clause 4.2.7.2.2. The plaintext data and the reader MAC shall then be encrypted and placed together with the initialization vector (IV) in the fragment component of the record or container.

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

For protocols using containers, a hop-by-hop MAC shall also be computed and added as defined in clause 4.2.7.2.3, and shall be recomputed by each entity participating in the session when forwarding a received container (whether modified or not). Entities which also have deleter access shall additionally (re-)compute and add a deleter MAC as defined in

clause 4.2.7.2.2, except for message units associated with context zero where the deleter MAC shall be omitted. Entities with delete, but not write privilege, shall copy any writer MAC from an incoming message to the forwarded, outgoing message. Entities which also have write access shall additionally (re-)compute and add the writer MAC as defined in clause 4.2.7.2.2, except for message units associated with context zero where the writer MAC shall be omitted.

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 4.2.2.3.3, verification of the reader MAC requires decryption to first take place, whereas verification of the hop-by-hop, writer, and deleter MAC shall be done based on the encrypted data.

MAC verification by a middlebox shall only be done for those MACs for which the middlebox possesses the corresponding key. The key used for the hop-by-hop MAC is known by both adjacent parties and shall be used for robust sequence number handling by middleboxes lacking any read or write privileges 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 shall be aborted. On verification failure, a corresponding bad_reader_mac, bad_deleter_mac, bad_writer_mac, or (for hop-by-hop MAC or messages not using containers) bad_record_mac alert should be raised and an application-dependent action shall be taken. 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 and terminating the session.

4.2.2.3 Sequence numbers

4.2.2.3.1 General

Each entity involved in a TLMSP session shall use two arrays of 64-bit, locally maintained sequence numbers seq_client_to_server[j] and seq_server_to_client[j]for security processing, for purpose of replay protection.

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, record, or deletion attacks.

At a given TLMSP entity, there shall be one such sequence number pair maintained for each value among the set of upstream TLMSP entity identities, including the upstream endpoint, and the entity itself. No maintenance of sequence numbers for the downstream entities is necessary. The first array shall be used for messages in client-to-server direction and the second array shall be used for server-to-client direction.

NOTE 2: In the remainder of the present document, a generic description assuming one of the two directions is presented, where handling in the other direction is completely analogous. Thus, the sequence number notation is abbreviated to the generic seq[j]. For simplicity, it is assumed that seq[0] is associated with the sending endpoint, seq[j], 1 <= j < n, is associated with the n-1 middleboxes, and SEQ[n] is associated with the receiving endpoint. A given entity, i, maintains values of seq[j] for j <= i.

NOTE 3: A message unit is associated with a TLMSP protocol message undergoing its own separate security processing. There is in general no one-to-one correspondence between TLMSP protocol messages, records and sequence numbers. For example, a single TLMSP record can contain more than one TLMSP Handshake message, all being protected as a single chunk and all being associated with the same sequence number. Finally, a TLMSP Application protocol message may be sent in one single TLMSP record, but may comprise multiple containers, each to be processed and protected with a distinct sequence number.

Intuitively, the value seq[j] corresponds to the sequence number of the next anticipated message unit generated or processed/forwarded by entity j. Sequence numbers shall increase monotonically by one (1) for each new message unit, except when deletions have occured, see clause 4.2.3.1.4.

Usage of sequence numbers shall generally start after issuing the ChangeCipherSpec. When state thus changes immediately after this message the pending state in the associated direction of communication shall become the active state, setting all associated seq[j] values to 0, and the selected cipher suite shall start to be applied to all subsequent messages. There are two messages occurring before ChangeCipherSpec which require sequence numbers, the TLMSPKeyMaterial and TLMSPKeyConf, and these shall use the reserved sequence number 2_{64}-1, as defined in clause 4.3.7.

NOTE 4: This is a difference to TLS 1.2, [1], which uses sequence numbers also before ChangeCipherSpec. Messages occurring before ChangeCipherSpec are still protected against modification and reordering

by their inclusion in the Finished and MboxFinished verification hash. 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.

NOTE 5: TLMSP, like TLS, has only one type of Handshake messages occurring after ChangeCipherSpec, namely the Finished and MboxFinished messages. TLMSP records containing these messages are thus the only Handshake messages that are protected by the record layer and dependent on the correct usage of sequence numbers.

When generating a new message unit by entity j, when processing an incoming message unit at entity j (either destined for j or to be forwarded by j after processing), or, when not deleting a message, the seq[] values used for processing shall be determined as in clauses 4.2.2.3.2, 4.2.2.3.3, and 4.2.3.1.4.2. When piggy-backing of Handshake messages are used, the handling of sequence numbers as described in clause 4.3.1.2 shall also be applied.

4.2.2.3.2 Generating new or modified outgoing message, or forwarding delete indication

When the entity j is an originator of a new message unit (an endpoint generating, or a middlebox inserting a completely new message), j shall use the current value seq[j] to be included in the security processing as defined in clauses 4.2.3, 4.2.7 and 4.3.7. The selected encryption IV shall, for the pre-defined cipher suites, follow the definitions of annex A. Immediately after sending the message unit (before generating, forwarding, or deleting any further message unit), the originator entity shall update its own sequence number: seq[j]= seq[j] + 1.

EXAMPLE 1: A single record with three containers (corresponding to three message units) will be processed with sequence numbers seq[j] = n, n+1, and n+2, respectively, for some n.

EXAMPLE 2: A single record comprising three Handshake messages (corresponding to a single message unit) will be processed with the single number seq[j] = n, for some n'.

When a middlebox, j, generates an outgoing message unit by making modifications to an incoming one, the middlebox shall first determine the author entity of the message unit, i, and shall then use the sequence number seq[i]to process the incoming message unit and update the sequence number array seq[]as defined in clause 4.2.2.3.3.1. If the outgoing message unit is to be a modified one, the middlebox shall use the same seq[i]that was used when processing the incoming message unit, when generating the reader MAC, writer MAC (and if applicable, deleter MAC) of the modified outgoing message.

A writer middlebox that does not perform any modification may still choose to re-calculate reader MAC and re-encrypt the unit message before forwarding it, thereby setting itself as author. This may be decided according to policy, independently by each writer middlebox. Alternatively the encrypted message and reader MAC are maintained as-is when no modification is done. Deleter middleboxes who does not delete a message shall keep the reader MAC and encrypted part of the message unit intact.

For writer middleboxes, if the middlebox changed the message, it shall encrypt the modified message unit, and add the reader MAC, using its own entity identity and sequence number in the explicit IV, following the IV format requirements of the selected cipher suite as defined in annex A. Regardless of whether the middlebox made a modification or not, for the purpose of generating the forwarded message unit, the middlebox shall re-compute and add a new writer MAC, again using its own entity identity and sequence number as an input to the writer MAC, see clause 4.2.7.2.2. Similarly, a deleter or writer middlebox that chooses to not delete a specific container shall still re-compute and add a new deleter MAC, using its own entity ID and sequence number as input, while keeping any reader and writer MAC present unchanged. When a middlebox with delete or write privilege deletes containers, the middlebox shall replace the container with a delete indication and determine sequence numbers as defined in clause 4.2.3.1.4.

NOTE: Re-processing of otherwise unmodified messages as above is necessary to prevent reader middleboxes or unauthorized 3rd parties from "undoing" changes and deletetion done by upstream writer or deleter middleboxes, see annex E.

4.2.2.3.3 Processing incoming message

4.2.2.3.3.1 Pre-processing

When an entity k (a middlebox or an endpoint) receives an incoming message unit to undergo security processing (MAC verification and decryption), j shall first determine the author(s) of the message, as well as the originator, i.

Generally, the author , j, shall be determined from the e_id field of the explicit IV.

Additionally, for containers, there may also exist distinct deleter author and writer author, whose identity determination shall be done as follows.

Entities with deleter (or writer) privileges shall also determine the deleter author (and writer author) for the purpose of verifying the deleter (and writer) MACs. The deleter author is always the upstream closest entity with at least delete privilege and the writer author is always that upstream closest entity with write privilege (in both cases, for the

associated context).

To determine the originator, i, if the container does not have the D- or I-bits set, the originator i shall be set to the upstream endpoint. Otherwise, the originator shall be determined from the m_info field.

NOTE 1: When entities are numbered starting from 0 at the upstream endpoint, it will always be the case at the current entity k that originator (i), author (j), writer author (w), and deleter author (d) (when they are all defined for the context) satisfy: $0 \leq i \leq j \leq w \leq d < k$.

When the message unit does not use containers, the originator i shall be set according to the message type, as follows:

- if the message unit is a Handshake message, as defined in clauses 4.3.5 to 4.3.9, then the entity identity, i, of the originator is derivable as follows:

  i) if the message is encrypted (which can only be the case for the Finished and MboxFinished message), first determine the message unit author j using the entity ID of the other-side specific IV and then perform message decryption (and other security processing);

  ii) the originator shall now be extracted from the LMSP Handshake message header (if such header is present, otherwise i is the upstream endpoint using standard TLS header);

- and, for all other messages, the identity i shall be set to that of the originator endpoint.

For the Handshake protocol, if the piggy-backing of clause 4.3.1.2 is used, then a single record being processed according to the above may contain plural individual Handshake messages from different originators, which subsequently may need to be processed one by one. If the Handshake record is unencrypted, it is trivial to extract the originators of the different piggy-backed Handshake messages from the message headers and there is no need to update sequence numbers since they are not used until ChangeCipherSpec is issued. If the Handshake record message is encrypted, determining individual originators, as mentioned, first requires decryption using crypto parameters of the author whose identity can be extracted from the IV (but whose message will be the last in the record). The sequence numbers of entities topologically between the author and the most upstream originator (whose message is the first in the record) shall be updated as described in clause 4.3.1.2.

NOTE: When the Handshake message unit is encrypted, use of piggy-backing can not be detected until after the decryption of step (i) above.

If the receiving entity, k, has read, delete, or write privileges for the received message unit, the current value of seq[j] shall then be used to security-process the fragment (including reader MAC) of the message/container as defined in clauses 4.2.3 and 4.2.7. When k has delete or write privilege, the deleter and/or writer MACs, as applicable, shall also be verified using the entity identity(s) and sequence number(s) of the determined deleter and/or writer authors, before any other security processing. If integrity verification of any MAC fails, further processing of the message shall be aborted, the appropriate bad_reader_mac, bad_deleter_mac, bad_writer_mac, or,

(for hop-by-hop MAC and messages not using containers) bad_record_mac alert shall be generated, and an application dependent action may be taken. If integrity verification is successful, entityk shall, before further message unit handling (e.g. modification, buffering, forwarding), update the sequence number array seq[]as follows.

- If the message unit is not a deletion indication: set seq[t]= seq[t] + 1, t = i, …, k-1, where i

  shall be the originator of the message as determined above. Immediately after forwarding the message, entity k updates its own sequence number seq[k] as described in clause 4.2.2.3.2.

- If the message unit is a deletion indication, seq[] shall be updated as in clause 4.2.3.1.4.2.

When the message unit does not use containers there is no determined deleter author or writer author since there are no corresponding MAC values present in these message units.

For entities lacking all of read, delete, and write privileges to the associated context, sequence numbers shall be updated in the same way, but shall additionally take into account the processing details of clause 4.2.2.3.3.2.

4.2.2.3.3.2 Middleboxes without privileges

Middleboxes can be present in a connection where the middleboxes lack both read and write access for a specific context (other than context zero). However, the sequence number space is shared between all contexts. Such middleboxes shall process containers as they are being forwarded, to keep the sequence numbers updated to the correct value when they eventually receive a container for which they do have read and/or write access. Therefore, the text of clauses 4.2.2.3.3.1 shall apply also to such middleboxes as far as seq[] determination and update is concerned.

This would not be robust, however, unless middleboxes without read/delete/write privileges have means to verify the authenticity of forwarded containers. To this end, the hop-by-hop MAC of clause 4.2.7.2.3 shall be used and the sequence number updates of clause 4.2.2.3.3.1 shall only be performed if the verification of this MAC is successful.

4.2.3 Processing of specific message types

4.2.3.1 Protection of application message type

4.2.3.1.1 Container usage

The following applies only for TLMSP records having the application message type (type = 0x17 [2]).

Containers may be re-distributed between records. A single container shall not 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 for 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 applies also to the sending endpoint: the sender may buffer fragments corresponding to several containers received from the application layer and place them in one or more records before submitting them to the transport layer.

A deleter middlebox may delete a container. A writer middlebox may replace an original container by a modified one, delete a container, or insert a new container. The present document specifies insertions and deletions of containers in clauses 4.2.3.1.2 to 4.2.3.1.4.

Whenever there is a new or modified message generated by a middlebox (change/re-write, insert, or delete), not being a dedicated audit information container:

- 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.

- The resulting container shall include up to four MAC fields:

  1) a reader MAC field (using the key shared with the other readers);

  2) a deleter MAC (using the key shared with the other deleters);

  3) a writer MAC field (using the key shared with the other writers); and

  4) a hop-by-hop MAC field using the key shared with the next-hop entity.

For audit containers, the IV and MACs shall be as above, except the MAC key used in (2) shall be the key shared only with the destination endpoint, see clause 4.2.3.1.3.2.

NOTE: If several middleboxes make modifications to the same message, a receiving entity can always identify at least the originator and the author. The author will be identifiable from the e_id part of the IV which is always explicitly included. The originator will be that as indicated by the m_info field, if this field is present. (If no m_info field is present, the originator is always the upstream endpoint).

4.2.3.1.2 Insertions and deletions: general

Only deleter and writer middleboxes may delete containers, and only writer middleboxes may insert new Application protocol containers. Any middlebox may insert special audit containers as defined in clause 4.2.3.1.3.2.

NOTE: 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.

A middlebox that inserts/deletes containers shall always insert an m_info field to the container header, used for synchronization and replay protection purposes. The m_info shall have the structure shown in Figure 5.

```
+------+---------+~~~~+~~~~+~~~~+~~~~+~~~~~+~~~~~~~~~~+~~~~~~~~~~+
| e_id | n_pairs |e[1]|d[1]|e[2]|d[2]|... |e[n_pairs]|d[n_pairs]|
+------+---------+~~~~+~~~~+~~~~+~~~~+~~~~~+~~~~~~~~~~+~~~~~~~~~~+
```

Figure 5: m_info field

e_id shall be a one-octet subfield and shall contain the entity identity of the middlebox that performed the insertion or deletion and the octet-field n_pairs shall indicate the number of e[i],e[i] pairs that follow. The value pairs e[i],d[i] are present only when a deletion is indicated and n_pairs shall also have the only value in which non-zero e[i] as described in clause 4.2.3.1.4. d[i] shall indicate how many deletions that have been performed (by middlebox e_id) on messages with e[i] as source, since middlebox e_id last reported such a value. The m_info field shall comprise at most one (e[i],d[i]) value for any given e[i].

4.2.3.1.3 Insertions

4.2.3.1.3.1 General

When inserting a container associated with a specific context, the middlebox shall set the I-bit of the flags container header-field of the inserted container to 1, and add an m_info-field as defined in clause 4.2.3.1.2.

4.2.3.1.3.2 Audit trail

Request for audit trail of middlebox processing may be configured on a per-context basis during the handshake (defined in clauses 4.2.8 and 4.3.5). If a context is configured to request audit trail middleboxes shall insert special-purpose audit containers, at least on all containers on which the middlebox performed a delete or modification. Depending on application needs, the middlebox may insert audit containers also relating to inserted or just forwarded containers. The audit containers, which shall follow the general container format, carry audit information in the payload (fragment) part. Audit containers shall be inserted as logically belonging to the same context with which the audit container is associated.

EXAMPLE: An audit container could be inserted immediately after a modified (or inserted) container or immediately after a delete indication relating to a deleted container.

As long as the information in the audit container contains enough information so that the container(s) to which the audit information is related can be identified, audit containers may be inserted at any point by authorized middleboxes.

NOTE 1: The format of audit information is application specific and outside the scope of the present document.

When a middlebox inserts an audit container it shall set both the I- and A-bit to 1 in the flags field of the inserted audit container. The middlebox shall also add the m_info field. The middlebox inserts as many audit containers as needed, setting the I and A flags. In all inserted audit containers, the middlebox shall add reader and writer MACs as defined in clause 4.2.7.2. No deleter MAC shall be present of audit containers. A hop-by-hop MAC shall also be added. Regarding placement of the audit container, if the audit pertains to a deleted container, the audit container should be inserted after the delete indication pertaining to the deleted container. In all other cases, the audit container should normally be inserted immediately after the container associated with the audit info.

NOTE 2: An exception to the rule of inserting audit containers after the container associated with the audit information could be when desiering to provide processing hints to downstream entities, by instering the audit onformation ahead of the container.

An endpoint may also insert containers with audit information. If an endpoint inserts containers with audit information, it shall not add an m_info field.

For audit containers, the writer MAC shall be generated with the key shared only with the destination end-point, while the reader MAC (and encryption) shall use the context specific keys, shared by all entitities having access to the context. Inserted audit containers shall not be modified or deleted by other middleboxes.

4.2.3.1.4 Deletions

4.2.3.1.4.1 General

Only deleter and writer middleboxes may delete containers. If one or more contiguous containers are to be deleted, they shall be replaced by at least one special delete indication container, signalling the deletion to other middleboxes and the receiving endpoint. The transmission of this deletion indication may however be postponed, but shall occur at the latest immediately before the first non-deleted container to be forwarded to the destination endpoint. If a deletion is not performed, the deleter (and writer MAC, for writer middleboxes) fields shall instead be recalculated on the forwarded container.

The following approach should be used when signalling deletions. If n consecutive containers are deleted, a single delete indication container can be transmitted after the nth deletion. Alternatively, a sequence of delete indication containers can be 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. In this case, ordering is important and middlebox j shall send its own delete indication before forwarding that of middlebox i. The m_info container header field shall be set to the value corresponding to the context that the deleted containers belong to and the container shall be protected by the corresponding context-specific keys. Deletion indications shall have deleter MAC using the deleter key associated with the context of the deleted container but shall have no writer MAC field. In more detail, when a delete indication container is to be generated, the middlebox shall set the D-bit of the container header to 1 and an m_info field shall be generated as follows. The value e_id shall be set to the entity identity of the middlebox that generates the deletion indication. Further, each pair e[i],d[i] shall be generated as follows. d[i] shall be a two-octet value corresponding to the number of new deletions that has been performed since the middlebox last sent a delete indication relating to containers that where generated or inserted by the (one-octet) entity Ei (an endpoint or another middlebox). If no deletions have been made on containers with entity e[i] as origin, no e[i],d[i] shall be generated. Finally, the value n_pairs shall be the one-octet encoding of the number of e[i],d[i] pairs added.

EXAMPLE 1: 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 with e[i] as author, and

which it has not yet reported. At this point a container is received from Ei relating to context 3. The middlebox reports all outstanding deletions from context 1, before forwarding the container relating to context 3.

Alternatively, the following example approaches may be used to signal deletions.

EXAMPLE 2: Assume entity with e_id=j has first generated 5 containers and that 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 containers 3, 4 or 5. Middlebox k then generates a first delete indication replacing the 5_{th} container, containing a value pair e[i]=j, d[i]=3. Then, middlebox k forwards containers 6-10, but does not forward container 11. Then after the 12_{th} received container from entity j, a second delete indication will be generated by middlebox k, now containing a value-pair e[i]=j, d[i]=2. Therefore a total of two delete indication containers are produced by middlebox k.

EXAMPLE 3: In the same scenario as above, middlebox k could alternatively replace each deleted container by exactly 1 delete indication container, each having e[i]=j, d[i]=1, resulting in a total of 5 delete indication containers.

EXAMPLE 4: Aggregated deletion. Assume middlebox k receives a delete indication from middlebox j, indicating that j has deleted m containers originating from entity i, i < j < k. After that, middlebox k receives n containers originating from entity i that middlebox j chose not to delete, but which k wishes to delete. Middlebox k can now choose one of the following:

1) send its own delete indication (having e_id=k and comprising the pair e[k]=i, d[k]=n), following a forward of the delete indication from middlebox j (having e_id=j and comprising the pair e[j]=i, d[j]=m); or

2) send a single delete indication with itself as source (i.e. e_id=k), comprising the pair e[k]=i, d[k]=m+n.

Alternative 2 of example 3 can be seen as replacement of one middlebox's delete indication by another aggregated delete indication. A middlebox shall not delete a container with another delete indication unless it replaces it with one of its own, as exemplified above.

NOTE 1: It could be tempting to conceptually view a delete as a modification, rewriting an original message as a delete indication. A delete is as defined handled as a delete of an original message, followed by an insert of a delete indication.A deletion indication container may contain payload. It is application dependent and out of scope of the present document how to create such payloads and so is the action taken by an endpoint receiving a container with [illegible] set to 1 that contains a payload. An example may be the following.

EXAMPLE 5: The payload comprises the human readable string: "Malicious content removed". The endpoint acts on this by terminating the session.

The deletion indication container shall have a reader, delete and hop-by-hop MAC fields generated as specified in clause 4.2.7.2 and 4.2.7.3, but shall not have a writer MAC.

The sequence number to be used when generating or processing a deletion indication container shall be as defined in clause 4.2.3.1.4.2.

NOTE 2: It is within the assumed trust model that middleboxes obey the rule to not delete containers for contexts for which they have only read access or no access.

4.2.3.1.4.2 Sequence number handling for deletion indications

Let j be the entity identity of a middlebox performing deletions. There are two cases of sequence number handling depending on whether the incoming container is already a deletion indication and when it is not.

When an incoming container that has entity i as origin and the container is not a deletion indication, is received by middlebox j, and where the container is to be deleted by j, j shall increase a local value d[i] by 1 (each d[i] shall initially have been set to 0): d[i] = d[i] + 1. Furthermore, j shall increase the sequence number array: seq[k] = seq[k] + 1, k = i, …, j-1.

Additional inbound containers may be deleted, possibly having some other source i' in which case Di' shall also be increased by 1 for each container to be deleted and similarly then update seq[k] = seq[k] + 1, k = i', …, j-1.

Eventually, middlebox j generates a deletion indication container as discussed in clause 4.2.3.1.4. The middlebox j shall then retrieve its own local sequence number seq[j] and shall use it when generating the delete indication. Finally, middlebox j shall replace seq[j]by seq[j]+1, preparing for the next message. Middlebox j shall also reset all Di included in the deletion indication to 0.

Whenever an entity j (an endpoint or a middlebox) receives an incoming deletion indication container from entity i, comprising the pairs (e[0],d'[0]), (e[1],d'[1]), …, (e[d],d'[d]), for some d < i, entity j shall, after successful security processing of the deletion indication, adjust its local values seq[] values as follows:

- Let #del[k] = d'[0] + d'[1] + … + d'[k], where d'[k] = 0 if k > d;
- seq[k] = seq[k] + #del[k], k = 0, 1, ..., i-1; and
- seq[k] = seq[k] + #del[i-1] + 1, k = i, ..., j.

If the receiving entity is a reader middlebox, it shall then forward the deletion indication. A writer middlebox shall choose one of the following actions:

1) send any possible delete indication pertaining to own deletions, followed by forwarding the deletion indication from entity i; or

2) aggregate the additional deletions into its own set of deletions.

In the latter case, the middlebox shall increase its own local d[] values as follows:

- d[k] = d[k] + d'[k], k = 0, ..., i-1; and
- d[i] = d[i] + 1.

After sending (or forwarding) the delete indication, entity j shall increase its sequence number seq[j] = seq[j] + 1 and shall also reset all local d[k] values to 0.

4.2.3.1.5 Changes (write)

Only writer middleboxes may modify the content of a container. A writer middlebox shall leave the A-, I- and D-bits unchanged. If the e_id of the m_info field is present, it shall be left unmodified. The middlebox shall further generate the reader, writer, and hop-by-hop MAC fields in the same way as specified in clause 4.2.2. If applicable, a deleter MAC shall also be added.

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

4.2.3.2 Protection of control and alert message types

4.2.3.2.1 Handshake

Middleboxes shall not delete Handshake messages except under the message-specific conditions stated in clauses 4.3.6 and 4.3.7 and shall not modify parts of Handshake messages added by other entities, except as defined for middlebox discovery in clause 4.3.2.3. Two Handshake messages are protected by the Handshake protocol layer itself (before ChangeCipherSpec has been issued): the TLMSPKeyMaterial and the TLMSPKeyConf (Key Confirmation) messages (see clause 4.3.7).

When additionally protecting Handshake messages by the Record layer, the Record layer protection shall use only the first (reader) MAC field, and use the keys associated with context zero.

4.2.3.2.2 ChangeCipherSpec messages

TLMSP impose some changes on the TLS ChangeCipherSpec protocol. First, as in TLS, it consists of one single message as defined in [1]. Since ChangeCipherSpec is also logically associated with context zero, it does not use the container format. The difference compared to TLS lies in the timing of activation of security for the different TLMSP contexts, and the start of sequence number usage, which shall be done as defined in clause 4.5.

4.2.3.2.3 Alert messages

Any middlebox (including readers) may insert Alert protocol messages (type = 0x15 [2]) to signal error conditions (e.g. unsupported cipher suites) into the TLMSP session. A middlebox shall insert Alert protocol messages only for contexts to which it has at least read access (which will include at least context zero). For the Alert protocol, containers shall be used. The context ID in the container header shall be set to the context for which the alert applies.

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

The I-bit of the header shall be set, etc. as discussed in clauses 4.2.3.1. Two middleboxes inserting alerts pertaining to the same context to the same record shall do this using two separate containers. Both of these containers shall have the same context ID.

When protected alerts (i.e. alerts generated after ChangeCipherSpec) are used:

- Alert messages shall always use a reader MAC a, writer MAC, and a hop-by-hop MAC. The writer MAC shall be generated with the key shared only with the destination end-point, wheras the reader MAC shall be generated with the context specific key.

This allows all middleboxes, with any level of granted context access, to verify the integrity of an alert and also allows the endpoint(s) to verify the authenticity. Middleboxes shall not perform deletion or modification of Alert messages generated by other entities.

Insertions shall only be made into a record of the same protocol-type as the inserted message.

EXAMPLE: If an error condition occurs during the Handshake, any possible Alert message is transferred as a new, separate record rather than piggy-backed as an additional message into a Handshake protocol record.

Audit containers may be inserted also for the Alert protocol (containing e.g. additional information on an alert), following the same procedures as described in clause 4.2.3.1.3.2.

4.2.3.3 Processing summary

A middlebox shall never insert, delete, or modify messages of other protocols than those described in clauses 4.2.3.1 and 4.2.3.2. The table below summarizes how containers shall be used and which operations that are allowed.

Protocol	Use of Containers	Middlebox Modifications 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.	Yes, as a new separate message or piggy-backed in a message originating from an endpoint, see clause 4.3.1.
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, but generally by writer middleboxes only. Special audit containers may additionally be inserted by any middlebox.

Table 1: Processing summary.

4.2.3.4 MAC usage summary

The table below 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, denotes 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, denotes the the set of entities with read/delete/write access to context c.

Below, an input data to the MAC 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 used as an input to the MAC, but 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 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.

MAC type	MAC key-usage per TLMSP sub-protocol					MAC calculations[1]		Explicit data coverage						Implicit data coverage		
	Application			Alert	Hand-shake	Computation or re-calculation	Verification									
	Container type							Record	Container info					Author	MAC author	
	Normal	Audit	Delete-ind					Header	Header	Data fragm[2]	MACs			SEQ	ID	SEQ
											RM	DM	WM			
Reader MAC	RK(c)	RK(c)	RK(c)	RK(c)	RK(0)	W(c), but only when creating, inserting, or modifying message	R(c)	Y	Y	Y[3]	N	N	N	Y	n/a	n/a
Deleter MAC	DK(c)	n/a	DK(c)	n/a	n/a	D(c)	D(c)	Y	Y	Y[4]	Y[4]	N	N	Y[5]	Y	Y
Writer MAC	WK(c)	PK(i,dest)	n/a	PK(i, dest)	n/a	W(c)	W(c)	Y	Y	Y[4]	Y[4]	N	N	Y[5]	Y	Y
HBH MAC	PK(i,i+1)	PK(i,i+1)	PK(i,i+1)	PK(i,i+1)	n/a	All entities	All entities	Y	Y	Y[4]	Y[4]	Y	Y	Y	N[6]	Y

Table2: Summary of MAC usage.

Remarks:

1) We only consider entities that are currently participating in the session, see clause 4.3.8.
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.

3) Covers the unencrypted plaintext of the payload, before encryption was applied.
4) Covers the encrypted value, after encryption was applied.
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.
6) While the entity ID of the MAC author is neither explicitly or implicitly included, the MAC key used is unique to MAC author.

4.2.4 Generic TLMSP 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 (((Container.flags & 0xC000) > 0) { /* Check if I, or D bit = 1 */
                case true: m_info mInfo;
                case false: struct { };                                  /* empty */
        }
        Uint16 length;
        select (TLMSP_internal_layer) {
            case TLMSPPlainText: opaque;
            case TLMSPCompressed: opaque;
            case TLMSPCipherText: ContaineredFragment;
        } fragment;
} Container;

struct {
        Uint8 source_entity_id;
        uint16 delete_count;
} DeleteIndicator;


struct {
        uint8 e_id;
        uint8 n;                                                 /* number of delete indicators, may be zero */
        DeleteIndicator delete_indicators[3*n];
} M_Info;
```

The value of length shall be the integer representation of the octet length of the fragment.

4.2.5 Plaintext record format

The plaintext record shall be defined as in the present clau

```
struct {
        ContentType type;
        ProtocolVersion version;
        Uint16 tot_length;
        select (tlmsp_server_support_confirmed) {
            case true: uint32 s_id;
            case false: { }; /* empty */
      };
      select (type) {
          case 0x15, 0x17:
                    Container containers[TLMSPPlaintext.tot_length-4]; /* Application, Alert */
          case 0x14, 0x16:
                    opaque fragment[TLMSPPlaintext.tot_length-4]; /* ChangeCipherSpec, Handshake */
      }
} TLMSPPlainText;
```

For the Application and Alert protocols, totLength is the total length of all the containers.

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

The value TLMSPPlainText.tot_length can, for Application and Alert protocols, be calculated as

$$4 + \sum_{c} [5 + \text{TLMSPPlainText.container.length} + (\text{TLMSPPlainText.container.flags} \,\&\, 0xC000)\ ?\ (2+3*\text{TLMSPPlainText.container.m\_info.n}) : 0]$$

where the sum is taken over all the containers, c, in the TLMSPPlainText.container structure.

For ChangeCipherSpec and Handshake protocols, the value is simply 4 + the length of TLMSPPlainText.fragment.

4.2.6 Compressed record format

The current TLMSP does not make 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 totLength;
        select (tlmsp_server_support_confirmed) {
              case true: Uint32 s_id;
              case false: { }; /* empty */
            };
      select (type) {
        case 0x15, 0x17:
                Container containers[TLMSPCompressed.tot_length-4]; /* Application, Alert */
        case 0x14, 0x16:
                opaque fragment[TLMSPCompressed.tot_length-4]; /* ChangeCipherSpec, Handshake */
      }
} TLMSPCompressed;
```

where the containers field shall be identical to the corresponding TLMSPPlaintext.containers, after compression has been applied to all of its fragment TLMSPPlaintext.containers.fragment sub-fields. Hence the container length field is the TLMSPPlaintext structure, as defined as in clause 4.2.5, but based on the respective fragment lengths after they have been compressed.

The totLength can be computed using the same formula as above for the TLMSPPlaintext structure, but replacing TLMSPPlainText.container.length by TLMSPCompressed.container.length.

4.2.7 Applying 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 supported 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 three sub-layers: reader layer, writer layer, and forwarding layer applied in that order, using different keys. The writer 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 existing integrity protection, creating up to four MAC values in total. The exact details of which MACs to add and how to compute them are defined in clause 4.2.7.2.

The protected record format shall be:

```
struct {
        ContentType type;
        ProtocolVersion version;
        Uint16 tot_length;
        Uint32 s_id;
        select (type) {
            case 0x15, 0x17:
                    Container containers[TLMSPCiphertext.tot_length-4]; /* Application, Alert */
            case 0x14, 0x16:
                    Fragment fragment[TLMSPCiphertext.tot_length-4]; /* ChangeCipherSpec, Handshake */
      }
} 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. readerMAC, IV, padding */
        [opaque deleter_mac[SecurityParameters.mac_length];]
        [opaque writer_mac[SecurityParameters.mac_length];]
        opaque hop_by_hop_mac[SecurityParameters.mac_length];
} ContaineredFragment;
```

The value of TLMSPCiphertext.tot_length, shall be computed by adding the following terms:

- The length of s_id (4 bytes).
- The sum of the per-container values:
  - i) TLMSPCipherText.container.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 readerMAC).
  - ii) The sum of the sizes of: writer MAC (see clause 4.2.7.2.2), and of the hop-by-hop MACs (clause 4.2.7.2.3).

4.2.7.2 MAC generation

4.2.7.2.1 General

The first MAC shall be the built-in integrity protection of the chosen cipher suite and is referred to as the reader MAC. This is used for the detection of changes made by unauthorized third parties (which includes middleboxes that have not granted access to a particular context). For AEAD ciphersuites, 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].

Up to three additional MACs may be added; the deleter MAC, the writer MAC, and the hop-by-hop MAC. The deleter MAC serves to detect unauthorized deletions (or tampering with deletions) by unauthorized parties (including middelboxes and third parties) with less that delete access. The writer MAC similarly serves to detect unauthorized changes by parties with less that write access. The hop-by-hop MAC allows verification that a message has followed the correct path and maintenance of connection-state information (sequence numbers). Generation of reader, deleter, and writer MACs is defined in clause 4.2.7.2.2. Calculation of the hop-by-hop MAC value is defined in clause 4.2.7.2.3. Whenever a record is forwarded by a deleter or writer middlebox, the reader MAC is recalculated only if a change/deletion is made, whereas the deleter and/or writer MAC (both, if applicable) shall always be recalculated. The hop-by-hop MAC shall be calculated first by the origin of the container, and then by each each middlebox, regardless of whether that middlebox performed any modification or not. When only the reader MAC value is present, only the reader MAC value shall be recalculated. This exception corresponds to protecting the Handshake protocol and the cases defined in clauses 4.3.7.

4.2.7.2.2 Reader, deleter, and writer MAC generation

4.2.7.2.2.1 Protocols using containers

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 = TLMSPCompressed.type || TLMSPCompressed.version ||

TLMSPCompressed.s_id || seq_num ||
TLMSPCompressed.container.flags ||
[TLMSPCompressed.container.m_info] || length || data || [e_id]

where in turn:

- **MAC** shall be the message authentication algorithm of the selected cipher suite.
- **mac _key** shall be the reader key for the reader MAC. For the deleter and writer MAC, 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 a container related to context i in the client-to-server direction, computed by an entity with write access.

For Application protocol audit containers, and, [illegible] Alert protocol containers, the key for the reader MAC shall be as above, but the key for the writer MAC shall be the MAC key shared only with the destination end-point, i.e. corresponding to e1_to_e2_mac_key, as defined in clause 4.3.10.5 if the destination end-point. There shall be no deleter MAC for this type of messages.

- **TLMSPCompressed.type**, **TLMSPCompressed.version** and **TLMSPCompressed.s_id** shall be as defined by the TLMSP record header, see Figure 2.
- **seq_num** shall be as determined in clause 4.2.2.3, specifically:

  i) when entity k either generates a reader MAC for a new message with k acting as originator, or, on a reader MAC for a modified message having another originator/author, or, any of deleter or writer MAC: the local TLMSP (64-bit) sequence number seq[k];

  ii) when verifying any of: reader MAC of an incoming message having entity j as author, or, a deleter or writer MAC having entity j as deleter/writer author: the value seq[i].

- **TLMSPCompressed.container.flags** shall be taken from the container header, as in Figure 3.
- **TLMSPCompressed.container.mInfo** shall be taken from the container header, as in Figure 3, and is present only for containers with A-, I- or D-bits set.
- **length** shall be a 16-bit unsigned integer and:

  i) when computing a readerMAC value, it shall be assigned the value LR = TLMSPCompressed.container.length;

  ii) 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 + 1
with LR as above.

- **data** shall be:

  i) when computing the reader MAC: TLMSPCompressed.containers.fragment.fragment;

  ii) when computing the deleter MAC: TLMSPCipherText.containers.fragment.fragment, (which includes the IV, padding and the reader MAC).

  iii) when computing the writer MAC: TLMSPCipherText.containers.fragment.fragment (which includes the same as the deleter MAC, but not the deleter MAC itself).

- the optional entity id, **e_id,** shall be the entity id of the entity generating the MAC and shall be present only when computing 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.container.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 shall be used as input to the deleter and writer MAC. However, the value of TLMSPCipherText.container.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 and the so-called Additional Authenticated Data, AAD, (not to be encrypted) shall in the case of reader MAC consist of:

AAD = TLMSPCompressed.type || TLMSPCompressed.version || TLMSPCompressed.s_id
|| seq_num || TLMSPCompressed.container.flags ||
[TLMSPCompressed.cont[illegible].m_lined_length,

and for the writer MAC, AAD shall be the entire MAC input as above.

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 annex A.2.2 for the case-defined AEAD transform.

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. A failed reader MAC verification should thus result in issuing a bad_reader_mac alert.

A failure of the writer MAC verification while the reader MAC passes verification can only happen if a middlebox with read-only permission has modified the data in transit, or, if an entity with no access has modified the writer MAC. This shall always result in a bad_writer_mac alert being raised.

4.2.7.2.2.2 Protocols not using containers

When the container format is not used, only reader MAC values shall be computed. The details of clause 4.2.7.2.2.1 shall apply with the following changes for protocols that do not use containers (Handshake and ChangeCipherSpec). The input shall be

MAC_INPUT = seq_num || TLMSPCompressed.type || TLMSPCompressed.version ||
TLMSPCompressed.tot_length || TLMSPCompressed.s_id || data

where data shall be taken as TLMSPCompressed.fragment.

NOTE: For protocols not using containers, seq_num is the first input rather that the fourth. This difference is to simplify processing. For protocols that do not use containers, the seq_num is common to the entire record.

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

AAD = seq_num || TLMSPCompressed.type || TLMSPCompressed.version ||
TLMSPCompressed.tot_length || TLMSPCompressed.s_id.

4.2.7.2.3 Hop-by-hop MAC generation

This MAC shall be present only for protocols using containers.

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. Likewise, that entity's downstream neighbour is the next middlebox downstream and 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 container that the middlebox is sending or forwarding, an entity shall use the pairwise key it shares only with its current downstream neighbour, as defined above, and derived via the definitions of clause 4.3.10.3 and 4.3.10.4 (for non-AEAD transforms, the e1_to_e2_mac_key shall be used between entity e1 and downstream entity e2) .

Thus, the processing shall be as clause 4.2.7.2.2.1 with seq_num set to the local value of the entity generating the MAC and

data = TLMSPCipherText.containers.fragment.fragment ||
[TLMSPCipherText.containers.deleter_mac] ||
[TLMSPCipherText.containers.writer_mac]

and

length = LW + 2*SecurityParameters.mac_length + 8

(if both deleter and writer MAC are present). If the middlebox is participating in the session, the hop-by-hop MAC shall be computed and added, even if the middlebox did not modify the message.

When verifying a hop-by-hop MAC for a container it has received, an entity shall similarly always use the pairwise key it shares with its current upstream neighbour and the sequence number of that entity.

Hop-by-hop MACs shall not be used for protocols not using containers.

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 most recent entity identity of the middlebox that generated or most recently modified the message.

4.2.7.3.2 Null or stream cipher

Stream (or NULL) ciphers convert TLMSPCompressed.containers.fragment structures to and from stream TLMSPCipherText.containers.fragment structures.

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[TLMSPCompressed.container.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].

If the cipher suite is TLMSP_NULL_WITH_NULL_NULL, then this 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.

The length of the resulting fragment is:

SecurityParameters.record_iv_length + TLMSPCompressed.container.length + SecurityParameters.mac_length.

For protocols not using containers (Handshake and ChangeCipherSpec), the resulting fragment length shall be obtained replacing TLMSPCompressed.container.length in the above by TLMSPCompressed.fragment.length.

4.2.7.3.3 Generic block cipher

For block ciphers (such as 3DES or AES), the encryption and MAC functions convert TLMSPCompressed.containers.fragment structures to and from block TLMSPCiphertext.containers.fragment.c_fragment structures.

```
struct {
        opaque IV[SecurityParameters.record_iv_length];
        block-ciphered struct {
              opaque content[TLMSPCompressed.container.length];
              opaque reader_mac[SecurityParameters.mac_length];
              uint8 padding[GenericBlockCipher.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 the resulting fragment is

SecurityParameters.record_iv_length + TLMSPCompressed.container.length + SecurityParameters.mac_length + GenericBlockCipher.padding_length + 1

For protocols not using containers (Handshake and ChangeCipherSpec), the resulting fragment length shall be obtained replacing TLMSPCompressed.container.length in the above by TLMSPCompressed.fragment.length.

4.2.7.3.4 AEAD ciphers

For AEAD ciphers, the AEAD function converts TLMSPCompressed.containers.fragment structures to and from AEAD TLMSPCiphertext.containers.fragment.c_fragment structures.

The AEAD transform defined in the present document use a combination of explicitly signalled and locally derived

values to form the IV. Care has been taken to allow that the same IV can be used for all three MACs (reader, writer and hop-by-hop), see annex E for security considerations.

```
struct {
        opaque nonce_explicit[SecurityParameters.record_iv_length];
        aead-ciphered struct {
                opaque content[TLSCompressed.length + D + SecurityParameters.mac_length];
```



```
        };
} GenericAEADCipher;
```

The reader MAC is included in the content field, directly by the AEAD transform, see TLS 1.2 [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 the resulting fragment is:

SecurityParameters.record_iv_length + TLMSPCompressed.container.length + D + SecurityParameters.mac_length.

For protocols not using containers (Handshake and ChangeCipherSpec), the resulting fragment length shall be obtained replacing TLMSPCompressed.container.length in the above by TLMSPCompressed.fragment.length.

4.3 The TLMSP 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 level of access, 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 7 may be omitted if the information is already known):

- Exchange of:

  i) Hello messages to establish which contexts to use, propose algorithms and middleboxes, random values, authentication methods, and possible [illegible] of session resumption.

  ii) Certificates (or other credentials) and crypto[illegible]phic informat[illegible] to allow the client, server and middleboxes to authenticate themselves.

  iii) Necessary cryptographic parameters. The s[illegible]r chooses one [illegible]er su[illegible] that lies in the inte[illegible]ction of those supported by the client and the se[illegible] Since, exce[illegible] manip[illegible]ations of [illegible]ensions to the ClientHello, middleboxes shall typicall[illegible] engage [illegible] handsha[illegible]efore ob[illegible]ing [illegible] 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 privileges 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 to a middlebox by sending the necessary contribution for that middlebox to derive the cryptographic keys.

Each middlebox shall receive such a contribution from both client and server to authorize access 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. The contribution shall not be present in the message destined to a particular middlebox if the endpoint

wishes to withhold access permission to the context from the middlebox. Each middleboxes 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.

NOTE: These last features (TLMSPKeyConf and MboxFinished) are not present in the original mcTLS specification.

Until the first ChangeCipherSpec message, there shall only be the single context with the reserved m_info = 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. Application data shall not be transmitted in context zero. All middleboxes shall have full access to 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 annex B.1 or B.2 may be used.

CertificateVerify*


CertificateVerify2Mbox[M1]*
CertificateVerify2Mbox[…]*
Client2MboxKeyExchange[Mn]*
CertificateVerify2Mbox[Mn]*
TLMSPKeyMaterial[C,M1]
TLMSPKeyMaterial[C,…]
TLMSPKeyMaterial[C,MN]
TLMSPKeyMaterial[C,S]
TLMSPKeyConf[M1]
TLMSPKeyConf[…]
TLMSPKeyConf[MN]
TLMSPKeyMaterial[S,M1]
TLMSPKeyMaterial[S,…]
TLMSPKeyMaterial[S,MN]
TLMSPKeyMaterial[S,C]
TLMSPKeyConf[MN]
TLMSPKeyConf[…]
TLMSPKeyConf[M1]
ChangeCipherSpec
Finished
MboxFinished[C,M1]
MboxFinished[C,...]
MboxFinished[C,Mn]
MboxFinished[M1,M2]
MboxFinished[M1,…]
MboxFinished[M1,Mn]
MboxFinished[M1,S]
MboxFinished[…,…]
MboxFinished[…,Mn]
MboxFinished[…,S]
MboxFinished[MN,S]
ChangeCipherSpec
Finished
MboxFinished[S,M1]
MboxFinished[S,...]
MboxFinished[S,Mn]
MboxFinished[Mn,…]
MboxFinished[Mn,M1]

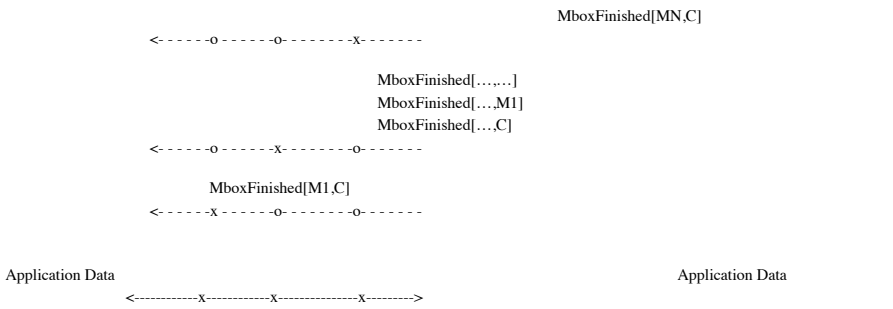

MboxFinished[MN,C]
MboxFinished[…,…]
MboxFinished[…,M1]
MboxFinished[…,C]
MboxFinished[M1,C]
Application Data
Application Data

Figure 7: Handshake, optional messages are suffixed by *

In Figure 7, 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. Lines that are "solid" indicate an explicit message, whereas the spaced dashed lines indicate a message which may be sent either as a standalone TLMSP record, or may be sent piggy-backed in a TLMSP record originating from an endpoint, according to clause 4.3.1.2. Two spaced dashed lines after each other thus indicates aggregated/recursive piggy-backing.

For middleboxes, their MboxHello, MboxCertificate, MboxKeyExchange and MboxHelloDone messages may be sent piggy-backed toward the client, but shall be sent as separate messages 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 different from that provided to the server, the client shall also send (or piggy-back) a CertificateVerify2Mbox message as defined in clause 4.3.6.7.

NOTE: From Figure 7, 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 may also be piggy-backed (and aggregated) into forwarded TLMSPKeyMaterial messages. MboxFinished[ei,ej] is a verification message of the handshake exchanges dependent on those messages previously exchanged between (or available to) both entities ei and ej. 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 in clauses 4.3.4 to 4.3.10 shall be as defined in and unchanged from structures of the same name in clause 7.3 and 7.4 of IETF RFC 5246 [1].

The signalling flow of Figure 7 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 annex B.1 or B.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, annex 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 annex 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 clauses 4.3.2 and annex B.4. This is safe since the server ignores unknown extensions.

4.3.1.2 Piggy-back of handshake messages

4.3.1.2.1 General

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 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 | ... | 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 straight forward for a middlebox MBa to piggy-back further messages into R (appending them at the end of the record). Also, it is straight-forward 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).

4.3.1.2.2 Piggy-backing principles

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

a) With the exception of TLMSPKeyMaterial and TLMSPKeyConfirmation messages, piggy-backing is generally only recommended to be applied to unprotected messages, occurring before ChangeCipherSpec.
b) Piggy-backing (before or after ChangeCipherSpec) shall not be performed if it results in a record that needs to be split into two or more records. Instead, a new, separate record aligned with the start of the message boundary shall be generated.
c) Piggy-backing shall only affect the record into which piggy-backing is performed.
d) Piggy-backing shall generally be append-only as laid out above. The only case when a middlebox may piggy-back information in the middle of a record is when replacing a TLMSPKeyMaterial message by a corresponding TLMSPKeyConfirmation message, see clause 4.3.7. In this case, the TLMSPKeyConfirmation shall be inserted in the same place as the original, replaced TLMSPKeyMaterial message.
e) Any incoming Handshake record or sequence of records into which the middlebox chooses to not piggy-back its own messages shall be forwarded toward the destination completely unmodified and respecting inter-record order. In particular, re-encryption shall not be performed.

4.3.1.2.3 Sequence number handling

Following the principles above, there will be a one-to-one correspondence between protected TLMSP records and TLMSP Handshake messages, i.e. the application of protection to a TLMSP record, even if containing several piggy-backed Handshake messages, can be unqiuely attributed to a single upstream author. The following approach may be used to determine the correct originator, author and sequence number by which to perform processing of TLMSP Handshake protocol records when they are protected. When records are not protected, which applies to all Handshake messages except the Finished and MboxFinished messages, the processing shall be omitted.

At a receiving entity k, the author, j, of a received, protected Handshake record (the upstream entity last to apply transport protection to the record) is deducible from the e_id field of the explicit IV, and thus the sequence number by which to process (decrypt, etc) the received record is therefore the next anticipated seq[j]value of that entity. All Handshake messages are associated with context zero and therefore there is no ambiguity in determining the correct cryptographic keys to use. The record may contain piggy-backed Handshake messages from other originator entities that are located further upstream from the author, but not from entities that are located topologically between entity k and j (though the record has however also passed those latter entities, k+1, k+2,… j-1, increasing their sequence numbers). The topologically most upstream entity identified as originator of any message included piggy-backed can, after decryption, be deduced from the entity ID, i, in the first message header contained in the record. Thus, in total, having received this record, seq[t] shall be increased by one for t = k+1, …, i in the same way as described in clause 4.2.2.3 and the own sequence number seq[k] shall be increased by 1 when forwarding the record (after

possibly piggy-backing information) and may be increased further, if entity k needs to generate further records or if it chooses to not piggy-back.

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 rights 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.[illegible]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 [illegible] middlebox is missing from the MiddleboxList, the server shall return a ServerHello, including the accepted middleboxes from the list in the ClientHello, extended by those non-transparent middleboxes that the server wishes to add. The 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 the 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 s_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 s_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 s_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 s_id and an additional middlebox list (ml_d) in the TLMSP extension as explained earlier in this clause. (The s_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 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 its topological order. The order should be deduced by observing the current value of previous_entity in the middlebox list, indicating the logical 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 specification.

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/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 s_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 8 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 8, 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.

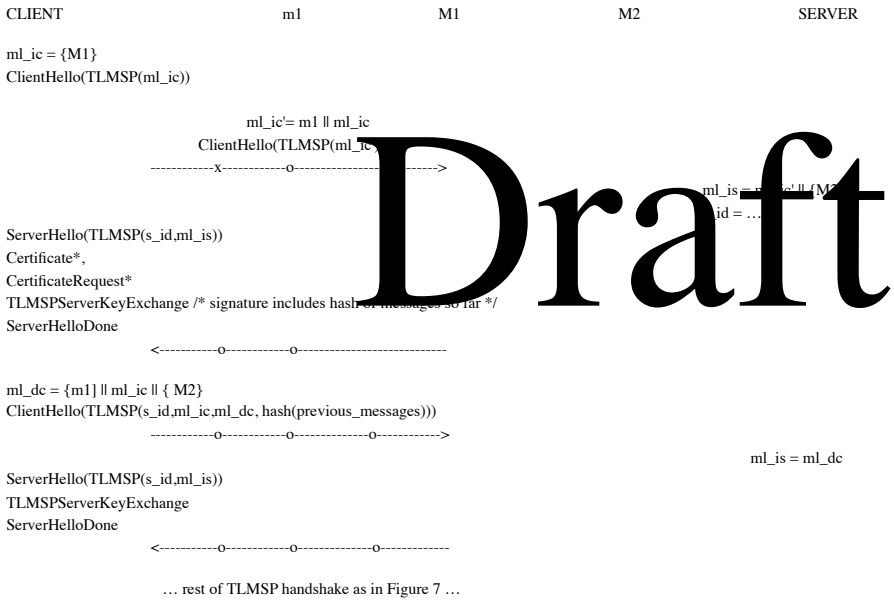


Figure 8: Dynamic discovery example

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

The optional alerts and MboxHelloRequest are not shown in Figure 8. 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 alertmiddlebox_suspend_notify (clause 4.4), or, the MiddleboxLeaveNotify handshake message (clause 4.3.8). These messages shall not be sent prior that Handshake completion (all Finished 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, include 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 s_id used in the TLMSP headers.) 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 1: 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.

NOTE 2: This is a difference to TLS [2] which allows the server to choose a new fresh session ID.

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 clause 4.3.10.4 and 4.3.10.45.

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 annex E.7. A TLMSP endpoint receiving an indication to perform renegotiation shall issue an unexpected_message alert and should abort the connection. Further, a TLMSP entity shall terminate an established session strictly before sequence numbers reach $2_{64} - 1$.

4.3.4 Handshake message types

The new TLMSP Handshake messages defined by the current specification shall be assigned IDs as follows:

```
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), (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 7.4.9 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), mbox_leave_ack(55).

The tlmsp_delegate message and usage is described in annex C.2.3.2. 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 = "to be assigned by IANA registration". The other extensions are defined in annex C.2.3. The extension shall contain a version indication according to [2] and a 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. The 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 defined as follows:

```
struct {
       enum { url(0), fqdn(1), ipv4_adr(2), ipv6_adr(3), mac_adr(4), (255) } address_type;
       opaque adr<1..65497>; /* identification string, according to address_type */
} Address;


struct {
        struct {
            uint8 major;
            uint8 minor;
        } tlmsp_version;
        CipherSuite tlmsp_cipher_suites<2..65497>;
        enum { false(0) , true(1), (255) } server_anon;
        select (is_server_hello) {
                case true: {
                                Uint32 s_id;
                                SignatureAndHashAlgorithms supported_sig_algs<2.. 65497>;
                            };
                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: Uint32 s_id;
                case false:                                     ContextList;
        }
        MiddleboxList ml_i;
        enum { false(0) , true(1), (255) } is_discovery_acknowledged_by_client;
        select (is_discovery_acknowledged_by_client) {
                case true: struct {
                                Uint32 s_id;
                                opaque pre_discovery<1..255>;
                                MiddleboxList ml_d;
                           };
                case false: struct { };
        };
} 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: 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 annex A and B.

When a TLMSP connection is first attempted, 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, 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 and shall leave ml_d empty. 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 4.3.9.3 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, a list of contexts for which the middlebox has read access, and lists of the contexts for which the middlebox has delete and write access. If a middlebox has delete or write access to a context, then read access is implicit. 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; /* type of access */
} 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;

struct {
        Uint8 previous_entity_id;
        MiddleboxInfo m<3.. 65497>;
} MiddleboxList;
```

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. 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.

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 upstream of the middlebox (in the direction of the server, including the server itself), or, to only be implicitly authenticated. Implicit authentication means that such upstream 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 upstream 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 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: 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: If, for example, the client instructs the first middlebox to not present its certificate to upstream entities, this implies that no upstream 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 upstream entities. A converse scenario applies when the server instructs a middlebox to not authenticate itself toward downstream entities.

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 session ID, s_id, and the server's supported signature algorithms. 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: A purpose string could have the value "malware removal service" for a middlebox performing malware removal.

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

struct {
      ContextDescription descriptions<4..65497>;
} ContextList;
```

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), i.e. an audit trail.

Additional values of audit are intended to be defined in the future, specifying that middleboxes add information on their processing to the audit containers. Currently only the additional value audit_trail has been specified but its usage is outside the scope of the present specification.

The third and fourth extensions are related to the TLMSP proxying and their usage is described in annex B.1 and B.2 of the present document.

TLMSP puts no restrictions on which port number to use.

Further (optional) TLMSP-related extensions are defined in annex 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 7, this message shall be sent from middlebox to client when the middlebox has received the ServerHelloDone. The message shall be simultaneously sent from the middlebox back to the server.

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;
```

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 structu


```
struct {
        Uint8 mbox_entity_id;
        ServerHelloDone hd;
} MboxHelloDone;
```

This message 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 m_info; /* information about the to-be-added middlebox */
} MboxHelloRequest;
```


The sub-fields of field m_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.

4.3.7 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 7, 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]. These messages shall use the sequence number 2_{64}-1, which thus shall not be reused for processing of any other message.

If record size extensions of 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 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:

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


Below, the value n_rctxt shall equal the number of contexts with granted read access, n_dctxt shall equal the number of contexts with granted delete access, and n_wctxt similarly shall equal the number of contexts with granted write access (including context zero). Finally, n_ctxt shall equal the number of contexts with either type of access. The entity_id shall be the entity identity of the entity (middlebox or endpoint) to which the message is directed and shall be left unencrypted. The maximum value of key_len supported by TLMSP shall be 2_{16}-1 octets.

NOTE: When a deleter_contribution is present, also a reader_contribution will be present, and when writer_contribution is present both a reader_contribution and a deleter_contribution will be present.

```
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 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 = middlebox_id || nonce_explicit.

A contribution received from the server granting read (delete or write) access to context_id = i is hereafter 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.middlebox_id ||
KeyMaterialContribution.IV ||
KeyMaterialContribution.contributions)

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 a partial secret for each context to which a middlebox has access. At least one message (for context zero) shall always 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. 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 keyLen 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. Precisely one of the delete_contrib or writer_contrib shall be present if, and only if, delete or write access is granted to the middlebox. The deleter_contrib and writer_contrib shall be cryptographically independent from each other and from the reader_contrib.

NOTE: 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 7, the TLMSPKeyConf (Key Confirmation) message shall be generated and sent by the middleboxes as they receive TLMSPKeyMaterial signalling from the client towards the server. 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 TLMSPKeyConf message also explicitly proves that the client's own key material contributions were correctly received (the server obtains this confirmation implicitly, see below).

Middleboxes shall also generate and send back TLMSPKeyConf messages as they receive TLMSPKeyMaterial signalling from the server toward the client. The TLMSPKeyConf message shall be structured as a 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.

To allow an entity to distinguish whether a message contains TLMSPKeyConf or TLMSPKeyMaterial, the message type may be used (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.

When a middlebox (entity E2) generates the protected TLMSPKeyConf message, MC1, it shall use the symmetric key shared with the destination endpoint (entity E1), using key_block_e1e2 according to clause 4.3.10.4 as basis for deriving the protection key(s).

The newly generated TLMSPKeyConf message MC1 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 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 (server or client) 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 secret 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' Finished messages, see clause 4.3.9.1.

The endpoints may use the "piggy-backing" mechanism defined in clause 4.3.2 to transmit TLMSPKeyConf information elements directed to several middleboxes in the same TLMSP record.

4.3.8 MiddleboxLeaveNotify and MiddleboxLeaveAck

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, 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, in which 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 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 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 Finished Message and handshake hashes

4.3.9.1 MboxFinished

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

```
struct {
      Uint8 src_entity_id, dest_entity_id;
      opaque verify_data[verify_data_length];
} MboxFinished;
```

src_entity_id and dest_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 4.3.9.2.2.

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

4.3.9.2 Hash computation in (endpoint) Finished message

4.3.9.2.1 Client-server Finished

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 following differences.

The first input to the hash shall be the initial ClientHello, except middlebox list entries with attribute inserted set to "dynamic". 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 4.3.9.3; and
- the client's inclusion of messages from the discovery phase into the pre_discovery_hash field of second ClientHello.

The ClientHello received at the server will contain the entity identity of the last-hop entity (a middlebox or the client) in the field previous_entity_id. This value will be identical to the entity identity of the last middlebox before the server, and thus, the value shall always (without loss of security) be replaced by the octet value 0 when computing the hash. (This value is identical to the last-hop middlebox entity identity in the middlebox list.)

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 4.3.9.1) messages.

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

- the MboxHello (clause 4.3.6.1), MboxKeyExc e (clause 4.3. and oxHelloDo (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 RFC 5246, [1], clause 7.4.9.

4.3.9.2.2 Hash computation in MboxFinished message

4.3.9.2.2.1 Case 1: one of the entities is an endpoint

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, 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.

Following a discovery, each middlebox could have received a forwarded ClientHello with different values for the field previous_entity_id. For reasons discussed in the previous clause, this value shall be replaced by 0 when calculating the hash.

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):

  iv) a list of received key material contributions, $L_{contrib}$, as defined [...] in the present clause;

  v) the client's Finished message with the server;

  vi) in the MboxFinished from the server to middlebox (only), also the middlebox's MboxFinished;
- for the MboxFinished message between middlebox and the *client* (only):

  vii) MboxCertificateRequest

  viii) the client's Finished message;

  ix) 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_certifcate = 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 or writer contributions ($C_{cd}(i_j) \,||\, C_{sd}(i_j)$), ($C_{cw}(i_j) \,||\, C_{sw}(i_j)$) are included only if the middlebox has delete or write access to the context.

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

4.3.9.2.2.2 Case 2: Both entities are middleboxes

The verification hash shall be computed over the same messages as described in clause 4.3.9.2.2.1, but with the following differences:

The client's Certificate2Mbox response to the middlebox's request and corresponding CertificateVerify2Mbox messages shall be omitted.

The MboxFinished sent from the middlebox to the server shall only be included in the MboxFinished sent in the direction of the server.

The MboxFinished sent from the middlebox to the client shall only be included in the MboxFinished sent in the direction of the client.

The MboxFinished directed from a source middlebox to a destination middlebox in the client-to-server direction shall be included in the MboxFinished sent in the other direction, between the same middleboxes.

The label "mbox to mbox finished" shall be used to prefix the hash input to the PRF and the PRF shall use the established master secret shared between the two corresponding middleboxes.

4.3.9.3 Hash in ClientHello 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, except for those entries in the middlebox list ml_i with inserted attribute set to dynamic.
- ServerHello (with extensions).
- Server's Certificate and CertificateRequest (if present).
- TLMSPServerKeyExchange.

- ServerHelloDone.

4.3.9.4 Hash in TLMSPServerKeyExchange

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.

When a middlebox generates a TLMSPServerKeyExchange (as defined in clause 4.3.6.6, 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 {
                                case (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: case (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 4.3.9.4 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: 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.

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... SHA256 ...re, id_li... all b... 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, in the same order as in the final agreed middlebox list, followed by;

3) the Server Certificate, when available.

If a certificate of some entity is not available to both entities e1 and e2, e.g. when the client and/or server has set the attribute use_certifcate 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.

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];
```

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 1: 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 2: These keys, known only to one endpoint and [illegible] box, are used in the protection of the TLMSPKeyMaterial and TLMSPKeyCo[illegible]essages co[illegible]ing the contribution. The MAC key is also used when a middlebox modifies or [illegible]rts new conta[illegible] or authenticates it via the hop-by-hop MAC.

When E1 and E2 are topologically adjacent middleboxes, [illegible] for hop-by-h[illegible]MACs [illegible]ll also be [illegible]nerate[illegible]n the same way, now identifying the entity e1 with the entity top[illegible]gically close[illegible]o the clie[illegible] and e2 th[illegible]ntity c[illegible]est to th[illegible] server, and now setting T = m.

NOTE 3: 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 that level of 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 4 or 6 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).

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, or, using client_writer_contrib_i and server_writer_contrib_i for those entities that have delete or 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 jth 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_eriter_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 associated client_to_server_reader_enc_key_i, client_to_server_fixed_IV_0 (for context 0), client_to_server_reader_mac_key_i, server_to_client_reader_enc_key_i, server_to_client_fixed_IV_0 (for context 0), server_to_client_reader_mac_key_i. Further, each deleter_key_block_i shall be partitioned into client_to_server_deleter_mac_key_i and server_ to_client_deleter_mac_key_i and each writer_key_block_i shall be partitioned into client_to_server_writer_mac_key_i and server_ to_client_writer_mac_key_i.

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 TLMSP Alert protocol

4.4.1 Alert message types

The set of Alert messages extend RFC 5246 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),                                     /* read    AC failed to verify                                  */
        bad_deleter_mac(176),                                    /* ditto,    MAC                                                */
        bad_writer_mac(177),                                     /* ditto, fo    er MAC                                          */
        middlebox_key_confirmation_fault(178), /* fail to verify key-s                                                            */
        authentication_required(179), /* middlebox requires client auth    re proceeding */
        middlebox_suspend_notify(180)                            /* middleb    aves th    ssion                                  */

        …,
        (255)
} AlertDescripton;
```


For existing Alert messages, clause 7.2 of IETF RFC 5246 [1] shall apply. Verification failure of a hop-by-hop MAC, or, of a MAC on messages not using containers shall be reported using the generic "bad_record_mac".

All alerts before the ServerHello has been observed shall follow and be limited to the definitions in [1]. Alerts following an observed server indication of support for TLMSP shall use the containered format, and will thus indicate the entity ID of the entity originating the alert. TLMSP-specific alerts occurring after the initial ServerHello shall have zero-length MACs if they occur before the ChangeCipherSpec message. Then, after the first ChangeCipherSpec, they will have non-empty MACs according to the selected cipher suite, which shall then be used until the session terminates.

The middlebox_suspend_notify alert message is a softer version of the MiddleBoxLeaveNotify. The messages signals that the middlebox will remain on-path, but only to verify and generate hop-by-hop MACs, it will not perform any message inspection.

The Alert levels of the additional messages shall be assigned as follows:

- authentication_required, middlebox_suspend_notify: AlertLevel warning;

- middlebox_route_failure, middlebox_authorization_failure, unknown_context, unsupported_context, middlebox_key_verify_failure, bad_reader_mac, bad_writer_mac, middlebox_key_confirmation_fault: AlertLevel fatal.

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 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.

Middlebox processing (read, write) of application data fragments is not possible until the middleboxes have received their read/write keys for the applicable contexts-

- The negotiated keys and cipher suite as well as any negotiated record size extention as per 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 shall come into effect, starting from zero, for messages immediately following this message.
- The negotiated keys and selected cipher suite shall start to be applied at the record layer immediately following the ChangeCipherSpec message (i.e. starting from the Finished message) for messages flowing in the same direction as the ChangeCipherSpec message itself . The ChangeCipherSpec

message shall also take any negotiated record size extension into general effect for all the following records.

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 clause C, standard TLS 1.2 cipher suites shall be used.

A.2 Key Exchange

The cipher suites defined below in annex 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 annex 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) TLMSP sequence number SEQ of the author shall be left-padded with the 1-one octet entity id of the author and three zero-octets to form a 12-octet value
   IV' = e_id || 0x00 || 0x00 || 0x00 || seq.
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.

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 and the 96 most significant bits in the counter.

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 verifying a received deleter (or writer) MAC, the IV shall use the entity id and sequence number of the deleter (or writer) author.

When verifying a received hop-by-hop MAC value, the IV shall use the entity id and sequence number of the upstream closest neighbour.

When computing deleter, writer or hop-by-hop MAC of an outbound message, the IV shall always use the own entity identity and own sequence number.

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 annex 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) << 56) XOR write_IV where:

- e_id shall be the one-octet entity identity for the originator,
- seq shall be the 64-bit 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.

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

This cipher suite consists of the counter-mode encryption part of AES_GCM (see annex 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 shall be generated as defined in annex A.3 of the present document.

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

A.6 Additional cipher suites

The cipher suite TLMSP_NULL_WITH_NULL_NULL must be used only for testing purposes, providing no security.

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 annex A.5 and should not be used without careful consideration.

A.7 Summary of security parameters

Cipher suite (Annex ref.)	parameter length				
	enc_key	mac_key	fixed_iv	Block	record_iv
GCM (A.3)	16 or 32	= enc_key_length (see note)	12	16	1

CBC (A.4)	16 or 32	= enc_key_length	16	16	1
CTR (A.5)	16 or 32	= enc_key_length	16	16	1

NOTE: For AEAD transforms, a separate MAC key is only needed for the additional writer and hop-by-hop MAC.

Table 3: Summary of security parameters

A.8 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}
TLMSP_ DHE_DSS _WITH AES_256_CBC_SHA256	{0x00,0x0C}
TLMSP_ ECDHE_ECDSA_WITH AES_256_CBC_SHA384	0,0x0D}
TLMSP_ DHE_DSS _WITH AES_256_CBC_SHA384	0,0x0D}
TLMSP_ ECDHE_ECDSA_WITH AES_128_CTR_SHA256	0,0x0F}
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}

Table 4: TLMSP cipher suite identifiers

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. 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.

Requirement A.2 may be implemented by including (e_id, seq[e_id]) in the IV using a mapping which is one-to-one with respect to (e_id, seq[e_id]).The predefined cipher suites have been designed to allow the same IV to be used for reader MAC, writer MAC, and hop-by-hop MAC. 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 two MACs, or, specify the use of three 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 annex 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 annex 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". Annex 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, it 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 must 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.

This annex 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 annex.

B.2.3.2.2 ClientHelloTo indicated 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 annex 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.

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 annex B.3.2.6.

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 key instead).

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 annex 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 annex 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.

CLIENT MIDDLEBOX 1 ... MIDDLEBOX N SERVER

ClientHello(TLMSP(ml_i))
----------o----------o----------o---------->

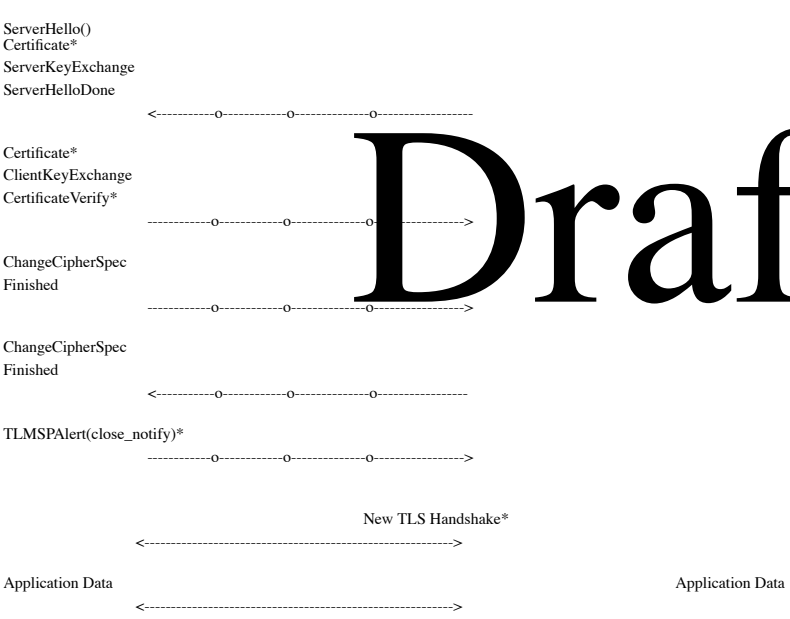


Figure B.1: Handshake for TLS 1.2 fallback

The symbols *, o and x are defined in the same way as in Figure 7.

After the handshake above is complete, the client may send a TLS close_notify Alert message 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 message to both endpoints, even if the client does not, to start a new, end-to-end TLS Handshake.

C.2 Fallback to TLMSP-proxying

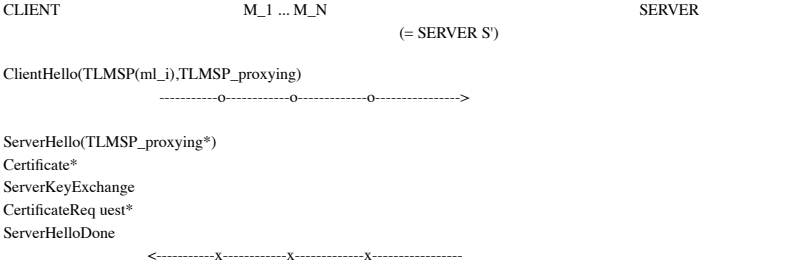
C.2.1 General

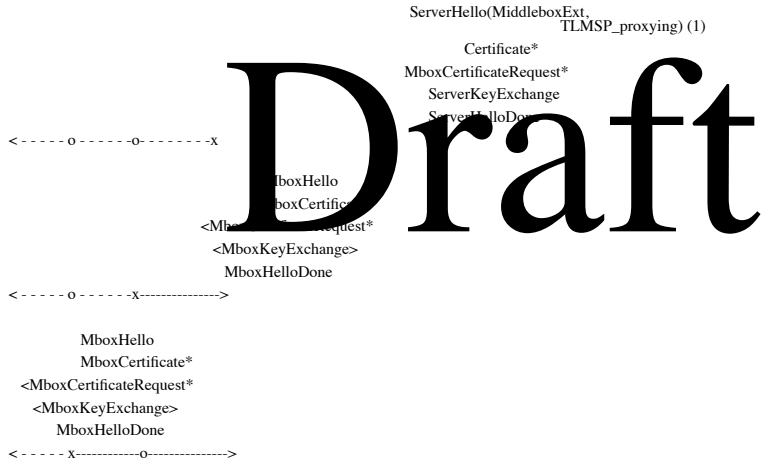
The procedure defined in annex C.2.2 of the present document may be supported by TLMSP clients and may be supported by middleboxes. The procedure shall not be used when the ClientHello does not contain a TLMSP proxying extension, as defined in clause 4.3.5. The server's lack of support for full TLMSP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




ServerHello(MiddleboxExt,
TLMSP_proxying) (1)
Certificate*
MboxCertificateRequest*
ServerKeyExchange
< - - - - - o - - - - - -o- - - - - - - -x
MboxHello
<MboxKeyExchange>
MboxHelloDone
< - - - - - o - - - - - -x---------->
MboxHello
MboxCertificate*
<MboxCertificateRequest*
<MboxKeyExchange>
MboxHelloDone
< - - - - - x----------o---------->
Draft


Certificate2Mbox[M1]*
Certificate2Mbox[…]*
Certificate2Mbox[Mn=S']*
Certificate*
ClientKeyExchange (2)
CertificateVerify*
----------o----------o----------o---------->
Certificate2Mbox[M1]*
Certificate2Mbox[…]*
Certificate2Mbox[Mn=S']*
CertificateVerify2Mbox[M1]*
CertificateVerify2Mbox[…]*
CertificateVerify2Mbox[Mn=S]*
TLMSPKeyMaterial[C,M1]
TLMSPKeyMaterial[C,…]
TLMSPKeyMaterial[C,Mn=S']
- - - - -x- - - - - - x- - - - - - - ->
TLMSPKeyConf[M1]
- - - - -x- - - - - - o- - - - - - - ->
TLMSPKeyConf[…,Mn]
- - - - o- - - - - - x- - - - - - - ->
TLMSPKeyMaterial[Mn=S',…]
TLMSPKeyMaterial[Mn=S',M1]
TLMSPKeyMaterial[Mn=S',C]
<---------x----------x----------
TLMSPKeyConf[…,C]
<- - - - -o - - - - - -x- - - - - -
TLMSPKeyConf[M1,C]
<- - - - -x - - - - - -o- - - - - -
ChangeCipherSpec /* To Mn */
ChangeCipherSpec /* To S */ (3)
Finished
----------o----------x----------o---------->
MboxFinished[C,M1]
MboxFinished[C,…]
Finished[Mn=S']
- - - - - - x- - - - - - x- - - - - - - ->
MboxFinished[M1, M2]
MboxFinished[M1, …]
MboxFinished[M1, Mn=S']
- - - - - - x- - - - - - o- - - - - - - ->
MboxFinished[…,Mn=S']
- - - - - - o - - - - - x- - - - - - - ->
ChangeCipherSpec /* From S */ (4)
Draft

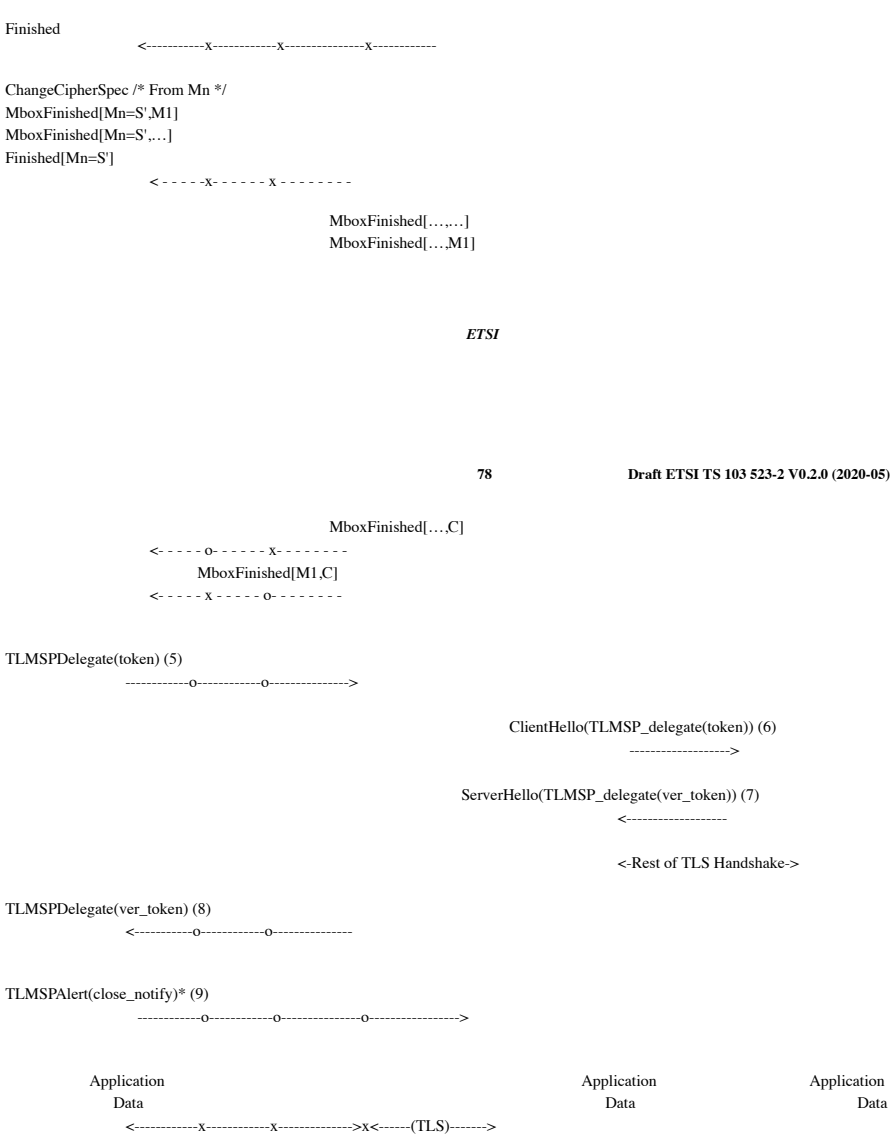


Figure B.2: Handshake for TLMSP proxying

The symbols *, o and x are defined in the same way as in Figure 7.

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 Server Hello and recognize 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 itself 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 includes the cipher suites proposed by the client for use in step 6.

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

   x) perform a standard TLS session handshake with the original server, and,

   xi) 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 annex 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 annex 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 = “to be assigned by IANA registration” and the extension_data shall consist of the session ID: uint32 s_id.

The TLMSP_delegate extension (used in step 6 and 7) shall have extension_type =“to be assigned by IANA registration” and shall have as extension_data

```
DelegateToken token;
```

where DelegateToken is defined as

```
struct {
        uint32 s_id;
        uint8 token_length;
        opaque token_value[token_length];
} DelegateToken;
```

The TLMSPDelegate message (step 5 and 8 in annex 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 annex C.2.2 shall be generated as defined in Eq. 1 and Eq.2 in the present annex.

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

The middlebox M_N (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 annex 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 annex 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 s_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

A middlebox (typically the first middlebox after the client) can enforce a security policy with respect to allowing outbound connections.

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

NOTE: The present document only defines usage of this mechanism when the policy enforcement function is placed in the first middlebox.

In such situations, it is undesirable to allow any outbound 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 7) is used, the client authenticity is not established with certainty by any middlebox until 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, the handshake with the first- middlebox shall be complete before forwarding the ClientHello to external network(s). This may be done by using the signalling flow of Figure B.2

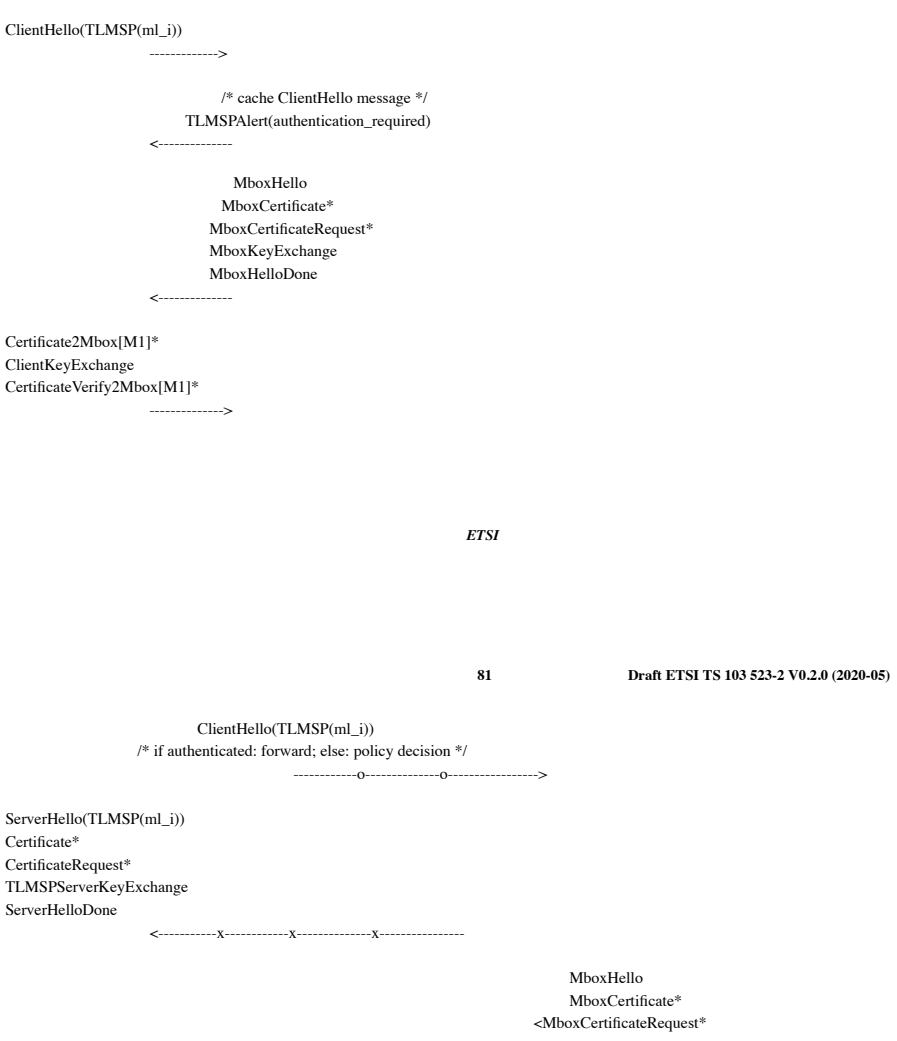


Figure C.3: Handshake, alternative policy enforcement flow

The symbols *, o and x are defined in the same way as in Figure 7.

The authentication_required message is an alert to the client to complete authentication with M1 first. Authentication of the client is done by verifying the client's signature in the CertificateVerify2Mbox.

NOTE: While the signature proves possession of the secret signing key, complete authentication of the whole handshake (by verifying all messages and security parameters provided by the client to M1) will not be done by M1 until a later point when the MboxFinished messages are verified. It is not possible to complete the whole handshake with M1 in advance, since there is no guarantee that the server and other middleboxes will support the same cipher suites.

After the client has been authenticated, M1 forwards the cached ClientHello and the protocol completes as in Figure 7, except that the key exchange with M1 is already done and omitting local information exchanged between the client and M1 when computing the Finished verification hash.

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

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.

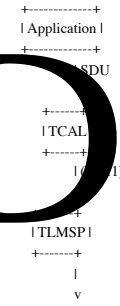


Figure D.1: TCAL concept

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 annex.

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

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 they have been granted (read and/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 privilege 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 annex E.3. This can limit the trust assumptions to only the last middlebox in the path and would also allow detection of "cheating".

Similarly, parties are assumed to participate in the protocol honestly; 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).

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. Annex 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 in 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 (as used in TLS cipher suites), but the IV formation has been changed to use partly explicit IVs. First, the sequence or cryptographically random 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 in the IV. It is impossible for these two values to both collide for two different messages. 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. While the deleter, writer MACs could in some cases happen to re-use the IV of the reader MAC, the reader, deleter, and writer MAC keys are different and for containers belonging to different contexts, they use different keys. The only case where the same key 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: in both these cases, the key used is independent of the context, but unique to the author. Thus, the only possible collision to consider is if the same entity would generate any of these MACs using the same key and IV. But again, due to the inclusion of the sequence number in the IV, any two such messages will by necessity using distinct sequence numbers and thus collisions can be avoided also here.

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 (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. The sending endpoint choses the initial nonce. There is therefore a potential threat that a malicious end-point could seek to bypass middlebox operations by reusing nonces, leading to ability to modify traffic after it has passed a middlebox. This threat is mitigated in TLMSP since all MACs (including hop-by-hop MACs) are dependent also on the TLMSP sequence number which is maintained loically by each entity.

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, or, in two middleboxes. In particular, for any pair of adjacent middleboxes, the verification provided by the MboxFinished message occurs at the end of handshake, allowing them to detect any modifications of handshake messages prior to the start of the communication session.

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 attack

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.5].

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.

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 annex 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 a 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 annex 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 annex 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 pairwise Finished messages, 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 7. 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.

EXAMPLE 1: A middlebox inserts cached content, avoiding need to fetch it 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 the attack afterwards. Eventually, some additional packets will be sent by the other endpoint or a close_notify message. 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.

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 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 16kB 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 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, insert messages into the session.

EXAMPLE 1: There is a chain of middleboxes e[1], e[2], … between server S (identified as e[0]) and client C (identified as e[n]). 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 sequence numbers used naively by e[i] only defines the inter-message order for messages inserted by e[i]. It does not uniquely define the inter-message order between two messages, one inserted by e[i] and the other inserted by e[j]. The same applies to sequence numbers used by e[j].

To address this, the sequence number used by e[j] when generating m[j] is dependent on whether the message m[i], generated by e[i], has already been received at e[j]. The sequence number used by a middlebox e when inserting a new message reflects the total number of messages that has passed e at the time of sequence number generation.

All middleboxes located downstream in the forwarding chain are able to predict the sequence number used by upstream middleboxes when they insert new messages.

One complication is that TLMSP middleboxes can also delete messages; therefore, deletions are notified to downstream middleboxes to allow them to adjust their local sequence numbers.

EXAMPLE 2: Five messages are deleted by middlebox e[i]. Middlebox e[j] obtains information about these deletions, since a future message that is not deleted will have been generated by a sequence

number which is at least 5 higher than e[j] would otherwise expect.

The sequence number handling is defined in full in clause 4.2.3.1.4.2 of the present document and the design comes from considering the following use case.

Let S' be a TLMSP session in which deletions occur. S is an identical session except that there are no deletions (any message produced by S' is allowed to reach the destination). Let m be any message of S that is generated by entity e[i]. If the sequence number used for m in S' is the same as it would have been in S, then the occurrence of deletions does not affect the (security) properties of the protocol.

Whenever e[j] (an endpoint or a middlebox) receives an authenticated deletion indication container from e[i], comprising the pairs (e[0],d'[0]), (e[1],d'[1]), …, (e[k],d'[k]), e[j] adjusts its local seq[] values as follows:

- seq[k] = seq[k] + (d'[0] + d'[1] + … + d'[k]), k = 0, 1, ..., i-1; and
- seq[k] = seq[k] + (d'[0] + d'[1] + … + d'[k]) + 1, k = i, ..., j.

Each message passing an entity, or being generated by that entity, increases the sequence number of that entity by 1. Now, the value d'[0] tells e[j] that d'[0] messages that originally had e[0] as source have been deleted; therefore, the local estimate of the sequence number for a next message coming from e[0] should be increased by d'[0]. Further, messages with e[0] as originator should also have passed e[1]. In addition, d'[1] messages having e[1] as source have also been deleted. Thus, the next expected sequence number for a message with e[1] as source should be increased by d'[0]+ d'[1], and so on; therefore a similar adjustment is made for all entities topologically upstream from e[i].

For e[i] itself, had there been no deletions, the sequence number of e[i] would have increased for each received message among the complete set of messages deleted. In addition, e[i] has also generated one additional message, namely the deletion indication itself, implying that the sequence number of e[i] should be increased by the total number of messages deleted, plus one: (d'[0] + d'[1] + … + d'[k]) + 1. The same holds for all entities between e[i] and e[j], including e[i] and e[j] themselves.

If e[j] chooses to forward the deletion indication immediately, no further action is needed; processing the forwarded deletion indication is already accounted for by the "+1" term used to update seq[j]. Alternatively, e[j] can merge the deletion counts / deletion indications it receives with its own deletions.

Where e[j] merges deletions, if e[j] has already deleted d[k] messages generated by some entity e[k], those messages were obviously not deleted by entity e[i]. Thus, when e[j] is now informed of d'[k] deletions made by e[i], of messages having e[k] as source, this means that in total d[k] + d'[k] messages are now deleted. In addition, since e[j] has chosen not to forward the deletion indication from e[i], entity e[j] has effectively deleted one message having e[i] as source. Therefore, entity e[j] updates the local counters for the deletion indications as follows:

- d[k] = d[k] + d'[k], k = 0, .., i-1; and
- d[i] = d[i] + 1.

Values d[k], k = i+1, …, j are not affected.

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 container that M has access to is received from the endpoint generating the message will process it with a sequence number s+d (or lower), M will use sequence number s-1+d (or lower).

If M does increase the sequence number to s+1, there is no way for M to know if the message is 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, but as discussed in annex F.3, this is not a workable solution for TLMSP; it creates 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.

Annex G (informative): Mapping MSP desired capabilities to TLMSP

G.1 General

The TLMSP profile is defined as a 2-sided authorization, fine-grained context MSP profile. This means that both client and server explicitly authorize middleboxes' access, and that different access rights can be granted to different parts of the data stream.

NOTE: Using the fallback mode of annex C.2, TLMSP can also support a 1-sided authorization profile, though no such mapping is explicitly provided.

The TLMSP profile meets the mandatory capabilities for a 2-sided authorization, fine-grained context MSP profile. MSP capabilities, defined for the MSP standards, are split into three groupings:

- **Audit:** capabilities that relate to the ability for a middlebox to be audited using MSP.
- **Access:** capabilities that relate to the access granted to a middlebox that is using MSP.
- **Visibility:** capabilities that relate to the ability for a middlebox to be discovered using MSP.

G.2 Audit capabilities

The table below outlines the required audit capabilities that TLMSP meets as well as which mechanisms and or concepts provide that capability.

Audit capability	Mechanism(s)
Destination endpoint able to detect if an unauthorized change to the data has occurred.	MACs (reader, writer, hop-by-hop).
Middleboxes with read-only access able to detect if a network adversary has made a change to the data.	der MAC.
Middleboxes with permission to modify content able to detect if an unauthorized change to the data has occurred.	Re MA ri AC
Middleboxes modifying content able to validate that no unauthorized changes have occurred prior to receipt by middleboxes or 3rd parties.	r MAC riter MA
Destination endpoints able to determine the middlebox or endpoint responsible for the most recent authorized change.	Meet if the hop-by-hopMAC concept is implemented.
Middleboxes able to determine the middlebox or endpoint responsible for the most recent authorized change.	Meet if the hop-by-hopMAC concept is implemented together with the audit trail/containers.
Destination endpoint able to determine all middleboxes that have performed authorized changes.	Meet if the hop-by-hopMAC concept is implemented.
Destination endpoint able to determine all authorized changes that have occurred and the parties responsible.	Meet if the hop-by-hopMAC concept is implemented together with the audit trail/containers.
Destination endpoint able to determine which middleboxes have inspected content.	Meet if the hop-by-hopMAC concept is implemented.

G.3 Access capabilities

The table below outlines the required access capabilities that TLMSP meets as well as which mechanisms and or concepts provide that capability.

Access capability	Mechanism(s)
Client and server able to grant middlebox access permissions by mutual agreement.	Key-contributions.
The protocol to provide mechanisms for fine grained control of access to portions content.	Per-context keys.
Access permission is granted on a per context basis by the granting endpoint; a single middlebox may be granted different permissions (such as read-only, read-write, append, and delete permissions) for different contexts; different middleboxes may be granted different access permissions to the same context.	Key-contributions, Per-context keys.

G.4 Visibility capabilities

The table below outlines the required visibility capabilities that TLMSP meets as well as which mechanisms and or concepts provide that capability.

Visibility capability	Mechanism(s)
Client able to learn the owner of all middleboxes.	Middlebox certificates.
Client able to learn the identity and function of all third-party middleboxes (i.e. middleboxes not under ownership of client or server endpoints).	Middlebox authentication and certificates and the purpose field of the contexts. To verify purpose, external attestation mechanism needs to be used.
Client able to learn the identity and function of all middleboxes.	Middlebox authentication and certificates and the purpose field of the contexts. To verify purpose, external attestation mechanism is however sed.
Server able to learn the owner of all middleboxes.	Middlebox certificates.
Server able to learn the identity and function of all third-party middleboxes.	Middlebox authentication and certificates and the purpose field of the contexts. To verify purpose, external attestation mechanism is however sed.
Server able to learn the identity and function of all middleboxes.	Middlebox authentication and certificates and the purpose field of the contexts. To verify purpose, external attestation mechanism is however used.
The client and server able to receive, if requested, validation of identity by each middlebox.	Middlebox authentication and certificates and the purpose field of the contexts. To verify purpose, external attestation mechanism is however used.
Client able to learn the long-term identity of the server.	Server authentication and certificate.
The client able to receive, if requested, validation of the server long-term identity.	Server authentication and certificate.
Server able to learn an identity (which may include "anonymous" or similar), for the client.	Client authentication and certificate.
Server able to learn a long-term identity for the client.	Client authentication and certificate.
Middleboxes able to offer anonymity services to protect clients' long-term identity.	Not natively supported, though the middlebox closest to the client could provide such services, acting as proxy on behalf of the client.
Server able to receive, if requested, validation of the client long-term identity (see note).	Client authentication and certificate or other credential.
Server able to reject anonymous clients (if anonymous clients are supported).	Client authentication and certificate or other credential.

NOTE: This implies the ability to perform mutual authentication.

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

IANA has been requested to assign 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 (xx), TLSMP_proxying (xx), and TLMSP_delegate (xx). See clause 4.3.5 and C.2.3 for more information.

History

Document history

V0.1.0	June 2019	First stable draft
V0.1.0a	August 2019	Fixed comments from various reviews. See CYBER-0027-2v010a version with change history for more information
V0.1.0b	Sept	Review from test/demonstrator implementer and meeting cyber#17
V0.1.1	Sept	Fixed comments from test/demonstrator implementer
V0.1.2	Sept	Some "must" removed and hanging paragraphs fixed with introduction of sub-headers
V0.1.3	October 2019	Pre-processing done before TB approval E-mail: mailto:edithelp@etsi.org
(draft) V0.1.4 October 2019		• Various corrections and editorial fixes. • Added how alert may be used when, during dynamic middlebox discovery, it turns out the server does not support TLMSP. • Clarified that when using piggybacking on handshake messages, a middlebox must first decrypt to determine the piggybacker(s). • Added a new cipher suite field to the TLMSP extension client/server now optionally containing TLMSP data from the TLS • Separated the writer MAC and forwarding MAC from the rest of the container into an "auxiliary container data" structure. "Length" of container now excludes this "auxiliary container data". • Changed notation for derived keys, hopefully making it easier to read and more consistent. • In the process above, noted that definition of MboxKeyExchange was wrong, so was the usage of certain "random" values.
(draft) V0.1.5 November 2019		• Clarified sequence number handling: for protocols that do not use containers, sequence numbers corresponf to individual records, while for containers they correspond to individual containers. Introduced "message unit" for this purpose and elaborated the handling of piggy.backed handshake messages. Also moved misplaced text in 4.2.2.3 that suggested update of sequence number after forwarding message, instead of after receiving. Also fixed typo on SEQ handling in 4.2.3.1.4.2. • Added "upstream" and "downstream" to the terms glossary. • Clarified port usage in 4.2.2.1. • Added guideance on when/where to insert audit containers. • In 4.2.8.3, added note 2 on forwarding MACs with order = 255. • Figure 7, moved ChangeCipherSpec so that it is now consistent with TLS. This required some further explanantion in clause 4.5. • Removed support for re-negotiation in 4.3.3.2 as it appears to create security issues. Added annex E.7 explaining the issues. • Clause 4.3.8, corrected and extended MboxHandshakeDeleteInd and

		removed from inclusion in Finished hash, see 4.3.9.2.2.
		• Misc editorial fixes throughout document.
(draft) V0.1.6	December 2019	• Various edits by NFR accepted.
		• Pointed out that since renegotiation is not supported, session needs to be terminated before sequence numbers wrap.
		• Corrected receiver processing of additional forwarding MACs
		• Clarified middlebox use of labels in PRF / "Finished" hash.
		• Remo Client2MboxKeyExchange, it is not needed now that R key tr port support ha en removed
		• Made verUnsupport a ha ake message rathe han alert his e ed to removed specif LMSPAlert altog er
		• Remo the "Del dication" handsha essag
		• Removed all usage of sequence numbers from Handshake before ChangeCipherSpec.
		o The only Handshake messages which may require security processing and sequence number determination is now the Finished/MboxFinished messages.
		• Clarified piggy-backing, in particular sequence number handling
		• Removed the inclusion of "inboundReaderCheckMAC" in input to MAC when only single forwarding MAC is present and a middlebox makes a change.
		• Clarified processing of "audit trail"
		• Added identifiers for ciphersuites in annex A.7, including "NULL"
		• Fixed some missing messages in the flow of annex B.3.
		• Updated annex E with the new security aspects.

0.1.7	Jan 2020	• Editorial fixes by NFR
		• Now allow only middleboxes to occur in topological order.
		• Removed forwarding MACs and replaced by hop-by-hop MACs. As a consequence, "confirmed delivery" was removed and an explcit synchronization MAC was added. "Audit trail" is still supported but without the "additional forwarding MACs".
		• Added "MiddleboxLeave" and "Suspend" possibility.
		• Further estricted the options for piggy-backing.
0.1.8(c)	March 2020	• New IV formation which avoids IV collisions between hop_by_hop MACs and also between writer MACs for audit containers.
		• Changed so that hop-by-hop MAC is now added also by entities which lack read/write access to the context. This allowed to remove synch MAC, since the hop-by-hop MAC now serves the same purpose.
		• Added pairwise MboxFinished verification hashes.
		• Simplified the description of handshake message parsing (finding author, originator, etc)
		• Added 4.3.10.6, key extraction
		• Added annex A.9 on adding new cipher suites
		• Added "delete" as a separate access right, distinct from "read" and "write".
		• Added "alternative cipher suites". This includes changing the GBA option from being a separate extension to include it in the general purpose cipher suite signalling. See (the new) annex B and clause 4.3.5

• Swapped order of annex B and C,

0.1.9	May 2020	• NFR review comments on 0.1.8 integrated (and no longer “tracked” as changes) • Added explanation why forwarding MACs do not work (seems best to document this so that we do not “rediscover” the problems again) • Restructured the TLMSP extension – some conditional fields where present based on the wrong condition • It is now mandatory for writers to compute new writer MACs on forwarded messages, even if no modification is done. This is to prevent readers from undoing writer operations. • For the same reasons, writer MAC of delete indications are now always recomputed when forwarding delet indications. • Now allow readers to insert audit containers. • Removed restriction to place audit containers **after** the container they are associated with. • Made use of “annex” and “clause” consistent. • Tried to make notation of field names and types consistent. • Added deleter MAC, but also made presence of deleter/writer MAC dependent on whether there is any middlebox with that access right o While re-computing the writer MAC at each writer solves the issue of a reader that “undos” a modification, there is still the issue that a reader could undo a delete. This requires that deleter MACs are also re-calculated, even when no delete is done. o When a delete is actually performed, the “trace” back to any possible modifications done between the originator and the point of deletion are lost. (Which is unavoidable.) • Fixed issues due to that middleboxes may not present cerificates to both endpoints • Fixed issues due to that middleboxes could be requested to use alternative cipher suites in case for both of the two directions (changed format of middleboxKeyExchange). • Added clause 4.2.3.4, MAC usage summary and the table therein.

0.2.0	May 2020	• Accepted all changes made by NFR during review • Fixed following additional NFR comments: o Added that KeyMaterial and KeyConf messages use seq = 2^64-1 as. o Clarified that all MACs except reader that encrypted data as inpout, not cleartext o Clarified when deleter/writer MACs are present and when/how they are recomputed o Clarified IV handling in annex A o Removed dummy writer MAC on Alerts • A writer may now choose freely whether or not to re-encrypt a message that it does not modify. •