ANNOUNCEMENT OF MIDDLEBOX SECURITY PROTOCOL (MSP) DRAFT PARTS

ETSI TC CYBER announces the release of two draft parts of an important new cyber security technical specification. These first two parts of a Technical Specification called the Middlebox Security Protocol address one of the most difficult security challenges today: how to enable network operators and end-users to cooperate in managing encryption security for their applications.

An exponential increase in the use of encrypted traffic is occurring at the same time as network cyber security requirements are resulting in massive numbers of intelligent systems in network infrastructures known as "middleboxes." In order to function for dozens of different essential needs, including cyber security, middleboxes need to understand the traffic being transported through the network to end-users.

The Middlebox Security Protocol enables the existence of a "smart proxy" where end-users can be potentially aware of a middlebox in their traffic stream (visibility) and control what that middlebox sees for different purposes (observability). The result allows for balancing privacy, network operations, and security for different applications. With the Protocol, both users and providers gain the ability to grant or restrict the permissions for visibility and observability.

Part 1 of the Middlebox Security Protocol specification defines the generic capabilities and security requirements. Additional parts define specific implementations in the form of profiles for different use cases that can be mapped to the Part 1 requirements. Part 2 provides a common profile for widespread network use known in the research community as mcTLS. Included with Part 2 are a patch for a known vulnerability as well as an exemplar of use by Mobile Network Operators. Other profiles will be released over the coming months – especially one for data centre access control to meet the critical needs of enterprise network communities.

These initial two draft specifications are relatively complete and stable, and derived from best-of-breed solutions drawn from extensive surveys and evaluation of the considerable published technical literature. However, this standards work is new, complex, and unique. The specifications will remain draft for a period during which widespread industry and public comments and views are sought. In addition, TC CYBER is proactively sending the drafts to other industry standards bodies as well as holding a Hot Middlebox Workshop (12 June 2018) and Hackathon (12-13 June 2018), in Sophia-Antipolis France, where the coding community can seek to implement and hack a test implementation of Part 2.

Comments may be sent until 30 June 2018 to: cybersupport@etsi.org using the template for comments:

https://docbox.etsi.org/CYBER/CYBER/Open/Latest_Drafts/Template-for-comments.doc

Draft ETSI TS 103 523-2 V0.0.8 (2018-04)

TECHNICAL SPECIFICATION

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

DRAFT PUBLICLY AVAILABLE UNTIL 30 JUNE 2018

Send comments ONLY to CYBERsupport@etsi.org

Download the template for comments:

https://docbox.etsi.org/CYBER/CYBER/Open/Latest_Drafts/Template-for-comments.doc

CAUTION: This DRAFT document is provided for information and is for future development work within the ETSI Technical Committee CYBER only. ETSI and its Members accept no liability for any further use/implementation of this Specification. Approved and published specifications and reports shall be obtained exclusively via the ETSI Documentation Service at http://www.etsi.org/standards-search

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

Draft

Important notice

The present document can be downloaded from:
http://www.etsi.org/standards-search

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the only prevailing document is the print of the Portable Document Format (PDF) version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at
https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx.

If you find errors in the present document, please send your comment to one of the following services:
https://portal.etsi.org/People/CommiteeSupportStaff.aspx

Copyright Notification

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.
The content of the PDF version shall not be modified without the written authorization of ETSI.
The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2018.
All rights reserved.

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

Contents

Intellectual Property Rights 5
Foreword 5
Modal verbs terminology 5
Executive summary 5
Introduction 6
1 Scope 7
2 References 7
2.1 Normative references 7
2.2 Informative references 7
3 Definitions, symbols and abbreviations 8
3.1 Definitions 8
3.2 Symbols 8
3.3 Abbreviations 8
4 TLMSP Protocol specification 10
4.1 Introduction 10
4.2 The Record Protocol 10
4.2.1 Overview 10
4.2.1.1 General 10
4.2.1.2 Records, containers and contexts 10
4.2.1.3 Record and Container Construction and Processing 11
4.2.1.4 Middlebox Container re-arrangements 12
4.2.2 Processing of Specific Message Types 12
4.2.2.1 Protection of Application Message Type 12
4.2.2.1.1 Insertions and Deletions: General 13
4.2.2.1.2 Insertions 13
4.2.2.1.2.1 Audit 13
4.2.2.1.3 Deletions 14
4.2.2.1.3.1 Writers 14
4.2.2.1.3.1 Readers 14
4.2.2.1.4 Changes (Write) 14
4.2.2.2 Protection of Handshake and Alert Message Type 14
4.2.2.3 Other Message Types 14
4.2.3 Generic TLMSP Container Format 15
4.2.4 Plaintext Record Format 15
4.2.5 Compressed Record Format 15
4.2.6 Applying Record Protection 16
4.2.6.1 General 16
4.2.6.2 Reader and Writer ICV Generation 17
4.2.6.3 Forwarding ICV Generation 18
4.2.6.4 Cipher Suite Specifics 19
4.2.6.4.1 Null or Stream Cipher 19
4.2.6.4.2 Generic Block Cipher 20
4.2.6.4.3 AEAD Ciphers 20
4.2.7 Middlebox Audit Trail and Forwarding MAC 20
4.2.7.1 General 20
4.2.7.2 Sender Processing 20
4.2.7.3 Middlebox Processing 20
4.2.7.4 Receiver Endpoint Processing 21
4.3 The TLMSP Handshake Protocol 21
4.3.1 Overview 21
4.3.2 Middlebox Configuration and Discovery 24
4.3.2.1 General 24
4.3.2.2 Static Pre-configuration 24
4.3.2.3 Dynamic Discovery 24

Draft

4.3.2.3.1 General ... 24
4.3.2.3.2 Non-transparent middleboxes ... 24
4.3.2.3.3 Transparent middleboxes ... 25
4.3.2.3.4 Semi-transparent middleboxes ... 25
4.3.3 Session Resumption ... 25
4.3.4 Handshake Message Types ... 26
4.3.5 Handshake Extension ... 26
4.3.6 Middlebox Hello ... 27
4.3.7 Middlebox Certificate ... 27
4.3.8 Middlebox Key Material ... 27
4.3.9 Middlebox Key Confirmation ... 29
4.3.10 Key Generation ... 30
4.3.10.1 General ... 30
4.3.10.2 Premaster and Master Secret and Key Generation ... 30
4.3.10.3 Context Specific Keys ... 31
4.3.11 MiddleboxHelloRequest ... 31
4.3.12 Finished Message ... 32
4.3.12.1 Middlebox Finished ... 32
4.3.12.2 Hash Computation in (endpoint) Finished Message ... 32
4.3.12.2.1 Hash Computation in Middlebox Finished ... 33
4.4 Alert Protocol ... 33
4.4.1 Alert Message Types ... 33
4.4.2 Change Cipher Spec Protocol ... 33
4.5 Compatibility Mode ... 34

5 Mapping to MSP Abstraction ... 35

6 Trust Model ... 35

7 Security Considerations ... 36
7.1 Protection Against mcTLS Attack ... 36
7.2 Inter-session Assurance ... 36

Annex A (normative): Defined Cipher Suites ... 37
A.1 Key Exchange ... 37
A.2 AES_{128,256}_GCM_SHA256 ... 37
A.2.1 General ... 37
A.2.2 Additional MAC computations ... 37
A.3 AES_{128,256}_CBC_SHA256 ... 37
A.4 AES_{128,256}_CTR_SHA256 ... 38
A.5 Summary of Security Parameters ... 38

Annex B (normative): Authentication Extension for MNO Provided Middleboxes ... 39
B.1 Introduction ... 39
B.2 Technical Details ... 39
B.2.1 General ... 39
B.2.2 Client Hello ... 39
B.2.3 Middlebox Hello ... 40
B.2.4 Server Hello ... 40
B.2.5 Server Key Exchange ... 40
B.2.6 Middlebox Key Material ... 40

History ... 41

Draft

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to the present document 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 the first part of a multi-part deliverable covering the Middlebox Security Protocol.

Part 1: "Profile Capability Requirements";

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

Part 3: "Transport layer MSP, Profile for data centre access control";

Part 4: "Network layer MSP, Profile for fine grained access control".

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 and service providers, users, enterprise, small business and individuals, to be able to grant varied (fine-grained) permissions to enable visibility of middleboxes and observability of the content and metadata of encrypted sessions. At present, the mechanisms used often break security mechanisms. For example, Man-In-The-Middle proxies frequently used by enterprise prevent the use of certificate pinning and EV Certificates. Where no such mechanisms exist, some encryption protocols may even be blocked altogether at the enterprise gateway, forcing their users to revert to insecure protocols. As more datagram network traffic is encrypted, the problems for cyber defence will grow. [i.5],

This technical specification is one of a series of implementation profiles that to achieve the above goals, putting the user in control of the access to their data for cyber defence and preventing unauthorised access. It sets forth a "Transport layer MSP (TLMSP), profile for fine grained access control" that meets the requirements found in Middlebox Security Protocol MSP Part 1. [1]

Authorised 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 divides the communication between the endpoints into

different contexts, each of which may have different permissions associated with it, following the security principle of least privilege. This subdivision is for the application to determine and under endpoint control.

The TLMSP protocol is modelled similar to the TLS protocol [2] 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 the TLS counterparts. These protocols satisfy the same basic properties described in [2] and additionally allow visibility and control of the security of the entire communication pathway to the endpoints and allowing the principle of least privilege to be enforced.

TLMSP is derived from mcTLS [i.1] with added features that include additional meta-data fields to allow middleboxes to perform not only read and modification, but also auditable insertions (of new data, originating at the middlebox) and deletions; a more flexible message format, allowing more adaptation to varying network conditions; on-path middlebox discovery; and additional security measures against recently discovered security vulnerabilities. Two normative Annexes are included that contain Defined Cipher Suites and Authentication Extensions.

Introduction

There are many uses of middlebox technologies that provide a better user experience (e.g. content caching to reduce latency, network prefetching of content); providing user protection and cyber defence (firewalls, malware detection); providing business protection (data loss prevention). These systems rarely require read and write access to all content sent and to all content received in order to function.

In addition, many of these systems also have to break the current security assurances that underlying encrypted protocols are intended to provide. For example, Man-In-The-Middle proxies used for gateway defence do not provide any assurance of the final endpoint identity, breaking certificate pinning and breaking PKI certificates. They also fail to provide any assurance that the connection beyond the gateway to the endpoint is even encrypted at all.

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 even supposedly legitimate software harvesting user data without user consent) should be able to insist that it is forwarded through their own cyber-defence systems and should be able to grant access to the content, any system that prevents this can be used as a means of stealing the user data with no means for them to prevent this. This is a privacy failure.

To prevent this, users should be able to layer their security architecture. They should not be forced to rely on endpoint defence alone as there will be some platforms where this is not possible. The best defence should always be a layered approach and not reliant on a single mechanism. This is expected to be particularly true of low power IoT devices which may not be capable of running endpoint protection, endpoint protection may not even exist, and where manufacturers may no longer support the platforms. This latter case leads to vulnerabilities that can only be protected by preventing malicious payloads reaching the IoT device in the first place, a requirement that can only be satisfied by network defence.

It is also not reasonable to expect users to replace devices as soon as manufacturers cease to support them. It may be a reasonable assumption but also a flawed one to expect manufacturers will patch their devices in a prompt manner. Cyber defence therefore assumes that there will be users with devices with known exploitable vulnerabilities. Protecting these devices requires a network level defence measure, endpoint defence is too late.

However, for privacy reasons, network defence should not require decryption and re-encryption of the data, data should be and remain end-to-end encrypted. In this specification a profile is presented to allow endpoints in a session to authenticate, create and end-to-end encrypted session and then authorise additional parties to access portions of the encrypted traffic. This profile provides full visibility of all the additional middleboxes and their permissions to both parties prior to the sending of any application layer traffic, 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, thereby assuring the fundamental principle that the user is in control of their own data and who may have access to it.

Draft

1 Scope

The present document specifies a protocol to enable secure communication sessions between network endpoints and one or more middleboxes between them using encryption, as well as authentication of the identity of any middleboxes present. This protocol is mapped to the abstract MSP protocol profile requirements in [1], the MSP characteristics supported are included in this specification.

The protocol specified here is a modified version of the mcTLS protocol by David Naylor et al. Whilst a significant portion is unchanged from the mcTLS variant presented, the protocol standardised here contains additional functionality and feature changes that would render it incompatible with the original version published.

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] ETSI TS 103 523-1, "Middlebox Security Protocol; Part 1: Profile Capability Requirements"

[2] IETF RFC 5246, "The Transport Layer Security (TLS) Protocol Version 1.2"

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] SIGCOMM '15, Naylor et al., Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS, August 17 - 21, 2015, London, United Kingdom. http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p199.pdf

[i.2] Carnegie Mellon University, David Naylor, Architectural Support for Managing Privacy Tradeoffs in the Internet, August 2017, PhD Thesis, http://reports-archive.adm.cs.cmu.edu/anon/2017/CMU-CS-17-116.pdf

[i.3] IEEE Symposium on Security and Privacy (SP) (2018), Bhargavan et al., A Formal Treatment of Accountable Proxying over TLS, May 23, San Francisco United States, (to appear)

[i.4] ETSI TR 103 421 Network Gateway Cyber Defence

[i.5] IETF, "Effects of Pervasive Encryption on Operators," draft-mm-wg-effect-encrypt. See also, "TLS 1.3 Impact on Network-Based Security," draft-camwinget-tls-use-cases.

Draft

3 Definitions, symbols and abbreviations

3.1 Definitions

For the purposes of the present document, the [following] terms and definitions [given in ... and the following] apply:

Fragment: A Service Data Unit (SDU) of application layer data, delivered from the higher level to TLMSP for compression and protection.

(TLMSP) Context: A specific part of the fragments governed by specific access policy. Here, “parts” can refer to headers, payload, specific “tagged” parts of the payload, etc. A context has associated cryptographic keys, made available to those entities that are allowed certain access (“read” and possibly “write”) to the corresponding context. The original mcTLS specification also uses the term “slice” here. A special context is defined for non-application data such as handshake and control messages.

(TLMSP) Container: An order-preserving sub-division of fragments, where each part is associated with a specific context or part thereof.

(TLMSP) Record: The Packet Data Unit (PDU) resulting from applying (optional compression and) TLMSP security processing to one or more containers while preserving the inter-container ordering. The record is delivered as SDU to lower layer (TCP).

Stacking: The process of grouping one or more containers into specific record.

3.2 Symbols

For the purposes of the present document, the [following] symbols [given in ... and the following] apply:

3.3 Abbreviations

For the purposes of the present document, the [following] abbreviations [given in ... and the following] apply:

3DES	Triple Data Encryption Algorithm
3GPP	Third Generation Partnership Project
AEAD	Authenticated Encryption Additional Data
AES	Advanced Encryption Standard
AES-GCM	Advanced Encryption Standard - Galois Counter Mode
AES_CBC	Advanced Encryption Standard - Cipher Blocker Chaining
B-TID	GBA-defined B-TID value (obtained during GBA bootstrapping)
BSF	Bootstrapping Server Function
BTID	GBA-defined B-TID value (obtained during GBA bootstrapping)
CBC	Cipher Block Chaining
CTR	Counter (mode)
CTXT_ID	Container Context Identifier
DH	Diffie-Hellman
DHCP	Dynamic Host Configuration Protocol
DHE_DSS	Ephemeral Diffie Hellman Digital Signature Standard
DNS	Domain Name System
FLAGS	TLMSP container flag field
FMs	TLMSP container fragment field
GBA	Generic Bootstrapping Architecture
GCM	Galois Counter Mode
GMAC	Galois Message Authentication Code
HMAC	Hash-Based Message Authentication Code
HTTP	Hypertext Transfer Protocol Version 2
ICV	Integrity Check Value
IEEE	Institute for Electrical and Electronic Engineers
IKE	Internet Key Exchange
IP	Internet Protocol

Draft

IV	Initialization Vector
IoT	Internet of Things
Ks_NAF	Network Access Function Key
LEN	Length
LFSR	Linear-feedback shift register
MAC	Message Authentication Code
MC	Middlebox Key confirmation message
mcTLS	Multi-Context TLS
MITM	Man in the Middle
MK	Middlebox Key material message
MNO	Mobile Network Operator
MSP	Middlebox Security Protocol
NAF	Network Application Function
NAF-Id	Network Application Function Identifier
NAF-key	Network Application Function key
NAI	Network Access Identifier
PDU	Packet Data Unit
PKI	Public Key Infrastructure
PRF	Pseudorandom Function
RFC	Request for Comments
RNG	Random Number Generator
RSA	Rivest–Shamir–Adleman
SDU	Service Data Unit
SHA	Secure Hash Algorithm
TCP	Transmission Control Protocol
TLMSP	Transport Layer Middlebox Security Protocol
TLS	Transport Layer Security
TR	Technical Report
TS	Technical Specification
USIM	Universal Subscriber Identity Module
UTF	Unicode Transformation Format
Uint	Unsigned integer

Draft

4 TLMSP Protocol specification

4.1 Introduction

The TLMSP protocol specified here is derived from the published mcTLS protocol. [i.1, i.2] That objective is to provide privacy, data integrity and authentication controls of communicating as provided by TLS whilst also providing access to the content (with fine-grained access control) to additional authorised and authenticated trusted middleboxes, with visibility of these middleboxes and control over the permissions granted provided to both endpoints.

Authorised 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 divides the communication between the endpoints into different contexts, each of which may have different permissions associated with it, following the security principle of least privilege. This subdivision is for the application to determine and under endpoint control. An example use case is for application-layer headers and content to be two separate contexts with different associated permissions.

The TLMSP protocol model is similar to the TLS protocol model with a similar presentation [2]. It is composed mainly 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 the TLS counterparts. These protocols satisfy the same basic properties described in the TLS protocol [2] and 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.

Unlike the original mcTLS, the protocol specified here includes

- additional meta-data fields to allow middleboxes to perform not only read and modification, but also auditable insertions (of new data, originating at the middlebox) and deletions;
- a more flexible message format, allowing more adaptation to varying network conditions;
- on-path middlebox discovery;
- additional security measures against recently discovered security vulnerabilities (see immediately below).

A recent paper [i.3] has identified vulnerabilities in the original mcTLS specification. These vulnerabilities pertain to middleboxes' assurance of endpoint identity and the possibility for a malicious server inserting (or re-insert) unwanted content after the data has passed a filtering middlebox. These issues are believed to be addressed by the TLMSP specification, though no formal proof is supplied.

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 Data Protocol), into TLMSP records, applies the agreed compression (optional), MAC integrity checks and encryption and then transmits the resultant records over TCP. Note that each record delivered to TCP can then be split across several TCP segments before transmission. Received records (after TCP re-assembly) are decrypted, verified, decompressed, reassembled and delivered to higher protocol layers.

4.2.1.2 Records, containers and contexts

A driver for the original mcTLS protocol was to provide fine grained access control to an encrypted session; different *contexts* could be used to provide different levels of access control to different parts of the application data. For example, for website whitelisting/blacklisting, one may wish to grant access to HTTP headers without granting access to the HTTP body. To do this, the HTTP headers and HTTP body can 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 may choose to split the data content across different contexts/containers is not part of this protocol specification, this example does highlight some potential undesirable features in the original mcTLS design.

The first is that data in one context may relate to data in another context (HTTP headers and body are clearly related to each other) and therefore a middlebox may need simultaneous access to data from more than one context in order to

Draft

carry out its function. It may therefore be desirable to have these delivered in one record, even though they correspond to different contexts.

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 would have drawbacks. This is because the straightforward approach would be that fragmentation is done so that each TLMSP record contains data associated with precisely one TLMSP context (i.e. according to a specific access policy for the middleboxes). Thus, the usage of contexts implies a specific *maximum* fragment size, and this size could be much smaller than the 16kB maximum record size specified for standard TLS. This would make it necessary to transmit data in smaller chunks, even when larger chunks are preferred from network performance point of view.

Therefore, in order for TLMSP not to prevent the traffic optimizations it seeks to enable, TLMSP allows data fragments associated with multiple contexts to be "packaged" into one single TLMSP record. Thus, an *TLMSP record* comprises protected data corresponding to one or more *TLMSP contexts*. A (contiguous) fragment of data associated with a particular context is called a *TLMSP container* (or simply "container"). Containers are used for the `Alert` and `Application` message types, but not for the `Handshake` and `ChangeCipherSpec` protocol/message types.

4.2.1.3 Record and Container Construction and Processing

```
+------+---------+------------+---~+~~~~+~~~+~~~~+
| TYPE | VERSION | TOT_LENGTH | C1 | C2 |...| Cn |
+------+---------+------------+---~+~~~~+~~~+~~~~+
 <-----   TLMSP header   ----> <- container(s) ->
```

Figure 1: TLMSP record format. C1, C2, … Cn represents containers, whose format is defined in Figure 2.

The first five octets of the TLMSP header is format compatible with a standard TLS 1.2 header, thus `TYPE` = 0x15 is for example used to signal the `Alert` protocol, etc. The TLMSP protocol specified herein uses `VERSION` = 1.2 (two octets {0x03, 0x03}). The `TOT_LENGTH` defines the total (octet) length of the record and is as in TLS bounded by 2^{14} octets. After the record header follows the actual container(s) For the `Application` message type (`TYPE` = 0x17), each TLMSP container is included in what would correspond to the payload part of a standard TLS record.

```
+---------+-----+~~~~~~~~~~~~~~~~~~+----+--------+-~+-~+~~~+~~~~~~~~~~~~~~~+
| CTXT_ID |FLAGS|M_INFO (OPTIONAL)|LEN |FRAGMENT|WM|FM|nAF|Additional FMs|
+---------+-----+~~~~~~~~~~~~~~~~~~+----+--------+-~+-~+~~~+~~~~~~~~~~~~~~~+
<-----       container header    ----->                <-  (OPTIONAL)  ->
```

Figure 2: TLMSP container format.

Each container starts with a container header comprising the associated one-octet context identifier, two bytes reserved for flags, and a 16-bit length field, indicating the length up to (and including) the FM field. Each container has 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 at the same time be limited to maximum 2^{14} octets.

```
+---+---+---+----------+
| I | D | A | RESERVED |
+---+---+---+----------+

Bit 15  14  13           0
```

Figure 3: `FLAGS` field of the container header.

The `FLAGS` field is used for signaling purposes, three bits are currently used and the remaining 13 are reserved for future purposes. The I and D bits are always set to "0" at the transmitting endpoint, but may be changed by an authorized middlebox to signal insertions or deletions. The A bit is used to signal that the current container has auditing content. Note that when computing any of the three MAC fields of a container as defined by TLMSP (reader, writer, and forwarding MAC), the information corresponding to length shall be based on the LEN field of the container header, it shall not be taken from the `TOT_LENGTH` field of the TLMSP header (this is described in more detail in clause 4.2.6). The forwarding MACs also contain a `FLAGS` field values used locally at a specific middlebox, see clause 4.2.6.3.

Draft

If, and only if, at least one of the I or D-bits are set to 1, there follows a three-octet middlebox info field (`M_INFO`), the format of which is discussed in clause 4.2.2.1.2. The container header is followed by the (possibly compressed and then) 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 (before the length and data of the fragment) when computing the MAC.

The fragment comprises the protected data fragment, including one MAC value (the reader MAC in mcTLS language, not explicitly shown). After the fragment there follows a writer MAC (`WM`) and at least one forwarding MAC (denoted `FM`). The (mandatory) first forwarding MAC corresponds functionally to the so-called endpoint MAC in mcTLS and is associated with the origin which is typically an endpoint, but could also be a middlebox in the case of inserted or changed containers. The (optional) additional forwarding MACs are used to obtain cryptographic verification that the container has passed all requested middleboxes. The number of additional forwarding MACs shall be indicated by the one-octet field `nAF`. When use of additional forwarding MACs have not been configured for the associated context during the handshake, the `nAF` field shall be omitted. Use of forwarding MACs is strongly recommended, see clause 4.2.7 for details. An implementation should not rely on any sequential ordering of the forwarding MACs, except when there is only one. The forwarding MAC has an explicit order field.

For other message types (e.g. `TYPE` = 0x14 or 0x16, indicating `ChangeCipherSpec or Handshake`), these are implicitly associated with context zero and containers shall not be used. That is, a single without container header, carrying the protocol message content) shall follow directly after the TLMSP header. Such messages shall apply generically to all contexts associated with the TLMSP session with one exception (see clause 4.4.2).

4.2.1.4 Middlebox Container re-arrangements

Middleboxes may combine containers from separate records in a single record containing all the containers, or may divide containers from a single record into multiple records, in order to make the most efficient use of the network bandwidth. This can be done without the middlebox needing access to the content of the context that the container belongs to, merely knowledge that the contexts exist.

This version of the protocol does not provide mechanisms to support changing of the container sizes after fragmentation. However, a middlebox with write access would be capable of performing this operation.

4.2.2 Processing of Specific Message Types

4.2.2.1 Protection of Application Message Type

4.2.2.1.1 General

For TLMSP records having of `Application` message type (`TYPE` = 0x17) only, the following applies.

A single container shall never be split across more than one record. However, for traffic flow optimization purposes middleboxes (both readers and writers) may split a single received TLMSP record comprising $C > 1$ containers into R ($1 < R \leq C$) distinct records before forwarding. Likewise, 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 as to correctly reflect the number of containers, their total length, etc.

A writer middlebox may always replace an original container by a modified one. Specific details regarding insertions and deletions of containers apply as specified below.

If requested via configuration done during the handshake (see clause 4.3.5), middleboxes shall add additional forwarding MAC-fields to containers, serving as proof that the middlebox has had access to the container and has had opportunity to act on the content (see clause 4.2.7).

Draft

4.2.2.1.2 Insertions and Deletions: General

Only writer middleboxes shall in general be allowed to modify, delete or insert containers. A middlebox that inserts/deletes containers shall always insert an M_INFO-field to a container header. The field is intended for synchronization and replay protection purposes and shall have the following structure.

```
+--------+----------------+
|   M_ID |     M_COUNT     |
+--------+----------------+
```

Figure 4: M_INFO field.

M_ID is a one-octet subfield which shall contain the identity of the middlebox that performed the insertion/deletion. M_COUNT is a two-octet subfield that shall contain a counter for the number of previous insertions and deletions performed by the middlebox during the session for the given context. Specifically, a 48-bit counter shall be used locally by the middlebox (and receiving endpoint). The sixteen least significant bits of this 48-bit counter shall be explicitly included in the M_COUNT field. The first insertion or deletion shall have M_COUNT = 0 and M_COUNT thereafter increases by one for each additional insertion/deletion. Whenever M_COUNT wraps (modulo 2^{16}), the remaining 32 most significant bits shall be incremented by one, locally by the middlebox/endpoint. A middlebox shall maintain one M_COUNT value for each context into which insertions/deletions are made.

The sequence number used to cryptographically process a newly inserted container or a container signalling a deleted container (at middleboxes and the receiving end-point) shall be the concatenation of the 48-bit M_COUNT field and the TLS-internal 64-bit counter.

Both when a middlebox inserts and re-writes a container, it shall generate a new IV compliant with the IV format of the used cipher suite. For example, any randomness or (statistical) uniqueness properties of the IV shall apply also to middleboxes' IV-generation.

When an end-point finds a gap in the M_COUNT sequence from any middlebox it should abort the session as it is a sign of malicious removal of content.

4.2.2.1.3 Insertions

4.2.2.1.3.1 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 one, and add an M_INFO-field as described above. The middlebox shall further add three MAC fields: reader and writer MAC fields using the key(s) shared with the other readers/writers and a forwarding MAC using the key shared with the destination endpoint.

When additional forwarding MACs are used (clause 4.2.6.3), a middlebox inserting a container should also set the I bit in the flag field of its forwarding MAC to one in the container before the container it is going to insert. Although this can only be cryptographically verified at the endpoint, other middleboxes may use this as a guide in their processing of containers. The middlebox can insert several containers in a row. It shall then set the I bit in the flags field of all inserted containers. It shall also set the I bit in its forwarding MAC to one in all but the last (locally) inserted container.

When more than one forwarding MAC has its I bit set, the order field of the FMs can be used, and shall be used by the endpoint, to determine the order of the insertions. The highest order should be the first to arrive.

4.2.2.1.3.1 Audit

Request for audit of middlebox processing may be configured during the handshake and middleboxes then insert special-purpose containers carrying audit information. When a middlebox inserts an audit record in a new container it shall set both the I and A bit to one in the flags field of the inserted container. The M_INFO field shall in this case also be added by the middlebox. The middlebox inserts as many containers as needed, setting the I and A flags. In all inserted containers the middlebox shall set the A bit to one in its forwarding MAC.

An endpoint may also set the A bit but shall then not add an M_INFO field.

Draft

4.2.2.1.4 Deletions

4.2.2.1.4.1 Writers

Deletions by a middlebox shall only be made if the D- and I-bits of the original container header are both zeroes. If one or more containers are to be deleted, at least the first deleted container shall be replaced by special container, signalling the deletion to other middleboxes and endpoints. This container shall have the same context identifier as the deleted one but shall set the D-bit of the container header to one and an `M_INFO` field shall be added as described above. When forwarding MACs are used, also the D bit of the flags field of the forwarding MAC shall be set to one. The actual fragment content of the container is application specific and outside the scope of this specification and so is the action taken by an endpoint receiving a container with D-bit set to one. A possible action is that the endpoint will terminate the session. The replaced container shall have a correctly generated reader MAC field and also a correctly generated writer MAC field. All existing forwarding MAC field(s) shall be copied from the original forwarding MAC field(s) and the middlebox shall add its own forwarding MAC field as follows: If forwarding audit trail is configured for the context, then the middlebox shall append its own forwarding MAC to the end of the existing set of forwarding MACs. Otherwise, the middlebox shall replace the (single) existing forwarding MAC with its own.

4.2.2.1.4.2 Readers

Readers may mark containers for deletion when forwarding MACs are used by setting the D bit to one in the flag field of its forwarding MAC. Other middleboxes shall not take action on this because they cannot verify that value cryptographically. Action, if any, shall be taken at the endpoint only. This feature may be useful to allow middleboxes with only read access to “hint” to the endpoint that the container should be discarded, without need to escalate the middlebox privileges to include full write/modify access.

4.2.2.1.5 Changes (Write)

A writer middlebox may modify the content of a container. It shall leave the I, D bits unchanged. The middlebox shall further generate the MAC fields in the same way as specified in clause 4.2.2.1.4 when generating a container replacing a deleted one. Specifically, the generated forwarding MAC is either appended to the pre-existing MACs (when audit trail is requested) and otherwise replaces the single pre-existing forwarding MAC. This enables the receiver endpoint to identify which middlebox that performed the last write modification.

4.2.2.2 Protection of Handshake and Alert Message Type

While insertions via containers could in principle be used also for the `Handshake` protocol (`TYPE` = 0x16), they are not necessary for this purpose. Standard TLS permits to insert/stack several `Handshake` messages into the same record (e.g. ServerHello, Certificate, and `ServerKeyExchange` typically are sent in the same record), using the per-message length field to identify individual messages during parsing. To implement the so called “piggy-backing” of middlebox information into the client-server handshake messages (see clause 4.3), the same approach is used in TLMSP. Note that in contrast to TLS, different parts of a `Handshake` message may thus originate from different entities. Middleboxes shall not delete `Handshake` messages.

Any middlebox (including readers) may insert `Alert` protocol messages (`TYPE` = 0x15) to signal error conditions (e.g. unsupported cipher suites) into the TLMSP session. 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, e.g. context ID zero is used for alerts relating to the handshake itself. For protected alerts, the following applies. A middlebox generating an Alert message shall add dummy MAC values for those keys it does not possess (e.g. endpoint keys and potentially writer keys). An exception is made for context zero (to which all entities have full access), where only the reader MAC value shall be present. Middleboxes shall not perform deletion or modification of `Alert` messages generated by other entities.

Note that insertions can (and shall) only be made into a record of the same protocol-type as the inserted message. For example, if an error condition occurs during the Handshake, any possible `Alert` message shall be transferred as a new, separate record rather than inserted as an additional message into a `Handshake` protocol message.

4.2.2.3 Other Message Types

A middlebox shall never insert, delete, or modify messages in other cases than those described above.

Draft

4.2.3 Generic TLMSP Container Format

For `Application` and `Alert` protocols, the container format is generically defined as follows, where the "payload" part (the `cFragment`) has a type that varies depending on whether we consider yet unprocessed plaintext received from the application, compressed plaintext, or protected ciphertexts, ready for submission to the TCP layer.

```
struct {
    Uint8 contextId;
    Uint16 flags;
    select (((Container.flags & 0xC000) > 0) { /* I and D bit  */
          case true:  M_INFO mInfo;
          case false: struct { };           /*  empty */
    }
    Uint16 length;
    select (TLMSP_internal_layer) {
       case TLMSPPlainText:  opaque;
       case TLMSPCompressed: opaque;
       case TLMSPCipherText: ContaineredFragment;
    } cFragment;
    select (more_containered_fragments) {    /* optional remaining containers */
          case false: struct { };            /*   empty  */
          case true: Container remaining_containers;
    }
} Container;

struct {
    Uint8  M_ID;
    Uint16 M_count;
} M_Info;
```

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

For other protocols, e.g. `Handshake`, that do not use containers, the payload fragment type is similarly defined as

```
select (TLMSP_internal_layer) {
   case TLMSPPlainText:  opaque;
   case TLMSPCompressed: opaque;
   case TLMSPCipherText: BasicFragment;
} Fragment;
```

4.2.4 Plaintext Record Format

```
struct {
    ContentType type;
    ProtocolVersion version;
    Uint16 totLength;
   select (type) {
     case 0x15, 0x17:

            Container container[TLMSPPlaintext.totLength]; /* Application, Alert */
     case 0x14, 0x16:
            Fragment fragment;                      /* ChangeCipherSpec, Handshake */
   }
} TLMSPPlainText;
```

Note that for the `Application` and `Alert` protocols, totLength is the total length of all the containers. This ensures that the format is compatible on record-level with that of TLS.

The value `TLMSPPlainText.totLength` shall (for `Application` and `Alert` protocols) be calculated as

$$\Sigma_c\,[5 + \texttt{TLMSPPlainText.container.length} + (\texttt{TLMSPPlainText.container.flags \& 0xC000})\,?\,3:0]$$

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

4.2.5 Compressed Record Format

Compression is defined similarly to [2], with the difference that compression is performed on per container basis:

```
struct {
    ContentType type;
    ProtocolVersion version;
    Uint16 totLength;
    select (type) {
     case 0x15, 0x17:
           Container container[TLMSPCompressed.totLength]; /* Application, Alert */
     case 0x14, 0x16:
           Fragment fragment;    /* ChangeCipherSpec, Handshake */
    }
} TLMSPCompressed;
```

where the `container` field shall be identical to the corresponding `TLMSPPlaintext.container`, after compression has been applied to all of its `cFragment` sub-fields. Compression shall be applied only to the cFragment fields of the container structure. Hence the container `length` field is defined as for `TLMSPPlaintext` structure but based on the fragment lengths after they have been compressed.

4.2.6 Applying Record Protection

4.2.6.1 General

As in TLS, the record layer of TLMSP is responsible for applying data protection to the sub-protocols forming the complete TLMSP protocol-suite (the `Handshake`, `ChangeCipherSpec`, `Alert`, and `Application` protocols). In TLMSP, the protection applied at the Record layer can conceptually be viewed as composed of three sub-layers: reader layer, writer layer, and forwarding layer, applied in that order, using different keys. The reader layer applies encryption and integrity protection, whereas the other layers only apply integrity protection.

The result of this layering is that for payload protection, two additional Integrity Check Values are typically added to the existing integrity protection, creating three Integrity Check Values in total.

The first ICV (MAC) shall be the defined TLS integrity protection and is referred to as the reader ICV. This is used for the detection of changes made by unauthorised third parties. For AEAD ciphers such as GCM, the integrity mechanism included in the cipher mechanism is considered to be the reader ICV. For block and stream ciphers, the first ICV is similar to what is specified in [2].

At least two additional ICVs are added (in most cases); the writer ICV and a forwarding ICV. The writer ICV shall use the MAC algorithm and a key known only to the endpoints and middleboxes that are authorised to have write access to the session. The forwarding ICV is computed using a key known only to the destination endpoint and the endpoint or middlebox making the change. Whenever a record is written, all three ICVs will need to be recalculated. There are exceptions when only one ICV is added, see clauses 4.3.7 and 4.3.8. In these cases, only the reader ICV (the first of the three ICV) shall be present.

More formally, the protected record format is defined by

```
struct {
    ContentType type;
    ProtocolVersion version;
    Uint16 totLength;
   select (type) {
    case 0x15, 0x17:
           Container container[TLMSPCiphertext.totLength]; /* Application, Alert */
    case 0x14, 0x16:
           Fragment fragment;    /* ChangeCipherSpec, Handshake */
   }
} TLMSPCipherText;
```

where the `container` field shall be the result of applying the selected TLMSP cipher suite to the corresponding `TLMSPCompressed.container`, on a per-fragment basis. The generic type `BasicFragment` is defined in a cipher-suite dependent way according to

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

which is then backward-compatible with the type for TLS fragments. When the fragments are part of a container we define

```
struct {
     BasicFragment fragment;
     opaque writer_mac;
     ForwardingMAC forwarding_mac;
    [[uint8 nAF;]]                          /* Number of additional forwarding MACs */
    [[opaque additional_forwarding_macs;]]} ContaineredFragment;
```

The `additional_forwarding_macs` are optional but recommended (see clause 4.2.6.3 and 4.2.7). To set the value of `TLMSPCiphertext.totLength`, note that this value will be larger than the sum of the individual `TLMSPCiphertext.container.length` fields. First, the size of the container headers needs to be added. Secondly, the security processing will add reader and writer MACs (see clause 4.2.6.2) and at least one forwarding MAC (clause 4.2.6.3). Also, when additional forwarding MACs are used, the value of `TLMSPCiphertext.totLength` will vary during the forwarding process from the endpoint via the middleboxes. It is the resposibibility of entities adding these additional forwarding MACs to adjust the value of `TLMSPCiphertext.totLength` accordingly. The source of the container shall take into account the number of forwarding MACs to be added by middleboxes and partition the data into protected fragments of sizes which, together with the to-be-added additional forwarding MACs, do not exceed the maxiumum size of a container.

4.2.6.2 Reader and Writer ICV Generation

For generic stream and block ciphers, for each fragment of the `TLMSPCiphertext.container` structure, the reader and writer different MAC fields shall be computed as follows.

MAC(MAC_write_key, MAC_INPUT);

where

MAC_INPUT = seq_num + TLMSPCompressed.type + TLMSPCompressed.version + flags + [TLMSPCompressed.container.mInfo] + length + data

where in turn

- **MAC** is the message authentication algorithm of the selected cipher suite.
- **MAC_write_key** is the writer/reader key (i.e. depending on which ICV to compute) for the current context (as defined by i = `TLMSPCompressed.container.contextId`), for example the key `client_writer_MAC_key_i` (clause 4.3.10.3) for a writer MAC on a container related to context i in client-to-server direction
- **seq_num** is
    - the 16-bit `TLMSPCompressed.container.mInfo.M_count`, first pre-pended with the local 32-bit roll-over counter, the resulting 48-bit sequence number finally being pre-pended to the local TLMSP sequence number; if at least one of the I and D bits of the flags are non-zero,
    - the local TLMSP (64-bit) sequence number; otherwise.
- **flags** is the two byte `TLMSPCompressed.container.flags` field from the container header
- **length** is
    - the unsigned integer value `TLMSPCompressed.container.length` when computing reader MAC
    - the value `TLMSPCipherText.container.length` when computing writer MAC (see note below)
- The optional **`TLMSPCompressed.container.mInfo`** field is included if, and only if, at least one of the two I and D bits of `TLMSPCompressed.container.flags` are non-zero.
- **data** is

Draft

- TLMSPCompressed.container.cFragment.fragment, except the reader MAC field itself, when computing the reader MAC,
- TLMSPCipherText.container.cFragment.fragment, including the reader MAC itself, when computing the writer MAC.

Thus note that the reader MAC is calculated based on the plaintext, before encryption (and the reader MAC may thereafter, depending on cipher suite, be encrypted). However, the writer MAC shall be calculated based on the result after the "reader sub-layer" has encrypted and integrity protected the fragment. Thus, the value of `TLMSPCipherText.container.length` shall be updated after applying the reader protection (which adds the size of the reader MAC and padding, if any). After calculating and adding the writer MAC, `TLMSPCipherText.container.length` shall be updated again to include also the writer MAC.

For generic AEAD cipher suites, the so called additional authenticated data shall consists of

seq_num + TLMSPCompressed.type + TLMSPCompressed.version + flags + [TLMSPCompressed.container.mInfo] + length

the mInfo field being included when present. Note that the computation of stand-alone MACs (other than the first reader MAC) is in the case of AEAD transform-dependent, see Annex A.2 for the pre-defined AEAD transform.

The reader ICV may be used by any endpoint or middlebox to detect that an unauthorised entity has modified the data in transit. The reader ICV passing verifies that the data has not been corrupted in transit (inadvertently or maliciously). This may be treated as an error. Some implementations may choose not to avoid possibilities for denial of service, particularly if they have alternative action they can take.

The writer ICV can be used by the endpoint or any middlebox with write access to determine if any reader middlebox has made an unauthorised change to the data. A failure of the writer ICV if the reader ICV passes can only happen if a middlebox with read-only permission has modified the data in transit. This shall always result in an error.

The server or client endpoint may choose only to check whether the data has been modified by a middlebox without subsequently verifying which middlebox made the change; a correct writer ICV will have already shown that whoever made the change was authorised to do so. However, a TLMSP implementation should indicate to the upper layer which middlebox made the last change to the container. The TLMSP protocol can thus be used by the upper layer to prevent attacks where the order of the endpoints and middleboxes is violated and used by a malicious endpoint.

4.2.6.3 Forwarding ICV Generation

The forwarding ICV shall be generated using a key only known by the endpoints, or, the destination endpoint and middlebox modifying/inserting the data. It allows the destination endpoint to detect whether or not the data has been changed by a middlebox in transit and by which middlebox.

Forwarding MACs have a special format and are generated in a special way.

```
struct {
    integrity-protected struct {
       uint8  id;
       uint8  order;
       uint8  flags;
       opaque outboundReaderCheckMAC[SecurityParameters.mac_length];
    };
    opaque checkMAC[SecurityParameters.mac_length];
} ForwardingMAC;
```

Here, `id` is the middlebox identity, and `order` field usage is defined in clause 4.2.7.

```
+---+---+---+----------+
| I | D | A | RESERVED |
+---+---+---+----------+

Bit  7   6   5           0
```

Figure 5: `FLAGS` field of the forwarding MAC.

The `FLAGS` field is used for signaling purposes, three bits are currently used and the remaining 5 are reserved for future purposes. This field basically reflects the current/local value of the FLAGS field of the container header at a particular middlebox.

As discussed earlier, when a middlebox constructs a forwarding MAC it should set the I bit to one when the next container is an inserted one and the D bit is used to signal deletions. The A bit of the forwarding MAC is used in conjunction with the corresponding A bit of the container flags to signal that the container comprises an inserted audit record, see clause 4.2.7. The use of the audit records is left largely unspecified, though examples are given in clause 4.2.7.3. The `outboundReaderCheckMAC` defined as

outboundReaderCheckMAC = MAC(MAC_write_key, HASH(MAC_INPUT))

where the MAC_write_key is as in clause 4.2.6.2 when generating the reader MAC. However, MAC_INPUT shall be the same as when generating the writer MAC. The HASH shall be the same cryptographic hash as used by the key derivation PRF.

Finally, the `checkMAC` field shall be generated as:

```
MAC(A_to_B_MAC_key, seq_num +
        TLMSPCompressed.type +
        TLMSPCompressed.version +
        TLMSPCompressed.container.flags +

        [TLMSPCompressed.container.mInfo] +
        length +
        id +
        order +
        flags +

        [inboundReaderCheckMAC] +
        outboundReaderCheckMAC);
```

where `inboundReaderCheckMAC` is the `outboundReaderCheckMAC` of the container as received from the previous sender (middlebox or endpoint). This field shall be treated as empty at the sender origin (the server or a middlebox performing an insert). The `A_to_B_MAC_key` shall here be the key shared (only) between A with the destination endpoint B (clause 4.3.10.2). The `id, order, flags` fields are as above and the remaining fields are as defined in clause 4.2.6.2. Note that besides the use of different inputs, also the cryptographic computation differs from the other MACs, by the use of the additional hash. A rationale for this computation is provided in clause 7.1. Note that adding forwarding MACs shall update the value of the container length (i.e. `TLMSPCipherText.container.length`), but only for the first (mandatory) forwarding MAC.

More than one forwarding ICV may be present in the form of additional forwarding ICVs, see clause 4.2.7 for details.

4.2.6.4 Cipher Suite Specifics

4.2.6.4.1 Null or Stream Cipher

Stream (or NULL) ciphers convert `TLMSPCompressed.container.cFragment.fragment` structures to and from stream `TLMSPCipherText.container.cFragment.fragment` structures. In contrast to TLS, TLMSP requires explicit IVs also for stream ciphers. This is necessary to allow the middleboxes to modify/insert/delete containers.

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

The two MAC's shall be created prior to encryption and are applied over identical data, "content" in the above structure, using different keys, as described above. The stream cipher shall then encrypt the entire block including the MACs.

If the cipher suite is TLS_NULL_WITH_NULL_NULL, then this consists of the identity operation (i.e. the data is not encrypted and the MAC size is zero for all two MACs).

Draft

4.2.6.4.2 Generic Block Cipher

For block ciphers (such as 3DES or AES), the encryption and MAC functions convert `TLMSPCompressed.container.cFragment.fragment` structures to and from block `TLMSPCiphertext.container.cFragment.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 [2].

4.2.6.4.3 AEAD Ciphers

For AEAD ciphers, the AEAD function converts `TLMSPCompressed.container.cFragment.fragment` structures to and from AEAD `TLMSPCiphertext.container.cFragment.fragment` structures. In contrast to TLS, TLMSP requires the entire IV to be explicit, or at least derivable from parameters included in cleartext, i.e. no implicit (key-dependent) IV shall be used.

```
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. The value `D` corresponds to padding and other overhead added by the AEAD transform in use.

4.2.7 Middlebox Audit Trail and Forwarding MAC

4.2.7.1 General

The purpose of this mechanism is to obtain an audit trail of the fact that a container passed all middleboxes and in which order it passed the middleboxes. This implies that all middleboxes had opportunity to perform their tasks, but it does not per se prove that they did those tasks correctly. This is left as an assumption on trust in the middleboxes.

When audit trail is requested, middleboxes (authorized to access the associated context) shall each generate and append a forwarding MAC at the end of the container.

4.2.7.2 Sender Processing

The original sender of the container (an endpoint or a middlebox making an insertion) generates the first forwarding MAC to be appended to the container as follows. The `order` field shall be set to zero and the `inboundReaderCheckMAC` is treated as empty. The remaining fields shall be set according to the same procedure as defined in clause 4.2.6.3 when generating the outbound ForwardingMAC.

The endpoint creating the container may insert forwarding MACs intended for verification by a specific middlebox, rather than the endpoint. The id field shall in such case be set to the id field of the destination middlebox and the order field shall have the reserved value 255. The check MAC shall be protected with the key only known to the creating endpoint and that middlebox. This could be used for special auditing purposes.

4.2.7.3 Middlebox Processing

When receiving a container, the middlebox verifies both the ordinary reader/writer MACs, as well as the `inboundReaderCheckMAC`. Only if these verifications are successful shall the middlebox proceed to process the container. When the middlebox processing of a container is finished, the middlebox shall generate the usual reader (writer) MAC values, encrypt, etc (if no changes are made, this can be done by simply copying the received MAC

Draft

fields). The middlebox shall also append all received forwarding MACs to the end of the container. Finally, the middlebox shall append its own generated `ForwardingMAC`. Note that when in use, a `ForwardingMAC` shall be appended regardless of whether the middlebox performed any changes to or operations on the container. The `order` field shall be set to n+1, where n is the highest value of the order field among the received forwarding MACs. The remaining processing to create the forwarding MAC is as defined in clause 4.2.6.3.

If the generating endpoint inserted a forwarding MAC directed to this middlebox and the data was not claimed to be modified by any other middlebox the receiving middlebox shall verify that this is the case. The middlebox shall remove - or overwrite with its own forwarding MAC - such forwarding MAC before passing the container on to the next entity.

The context may have been configured to keep a more detailed audit record of operations made by middleboxes. If this is the case, the middlebox shall add an extra audit container and use the A-bit of the flag field of the forwarding MAC for signalling purposes. Specifically, when the A bit of the forwarding MAC is set to one and the A bit of the container flags is set to zero this is signalling that the middlebox made a log audit record entry. How that log entry is accessed is out of scope of this specification. However, a session-ID together with the forwardingMAC.checkMAC value should be sufficiently unique for this purpose.

4.2.7.4 Receiver Endpoint Processing

When the receiver endpoint (client or server) receives the container it shall, besides verifying the other MACs, also perform the following verification.

- Verify that precisely one forwarding MAC field is present for all middlebox identities authorized to access the context associated with the container.
- Verify that all values of order (between zero and the total number of middleboxes involved) occur exactly once.
- For each forwarding MAC field, cryptographically verify the correctness of the `check_mac` field using the key shared (only) with the entity indicated by `id` using the `outboundReaderCheckMac` values of "adjacent" forwarding MACs (i.e. those with order values differing by one).
- Verify that the ordering of the middleboxes doing the audit was acceptable. What would be acceptable is out of scope of this specification.

Note that since there is no one-to-one association assumed between middlebox identities and the order in which they they are invoked, it allows some flexibility of the network to dynamically change the order in which middleboxes process the containers.

4.3 The TLMSP Handshake Protocol

4.3.1 Overview

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 start communicating, they agree on a protocol version, the number of contexts and their purpose, the middleboxes' level of access granted, and the cryptographic algorithm suite to use. The TLMSP Handshake Protocol involves the following steps:

- Exchange Hello messages to establish which contexts to use, agree on algorithms and middleboxes, exchange random values, and check for session resumption.

- Exchange certificates and cryptographic information to allow the client, server and middleboxes to authenticate themselves.

- Exchange the necessary cryptographic parameters. Middleboxes each insert their supported cipher suites during the handshake, and the server shall choose one suite that lies in the intersection of those supported by the client, the server, and by all the middleboxes.

- Information to authorise read-only or read-write access to the traffic if and only if both client and server agree to grant that level of access to the middlebox.

- Agree on a premaster secret and generate a master secret between the client and server endpoints and provide the security parameters to the record layer.

Draft

- Allow client and server to verify that their peer has calculated the same security parameters, including the list of middleboxes and their respective permissions requested, and that the handshake occurred without tampering by an attacker.

- The TLMSP handshake uses extensions added to the hello messages in the TLS handshake to agree on the authorised middleboxes and the contexts. An additional handshake message, `MiddleboxKeyMaterial`, is used to grant access to a middlebox by sending the necessary contribution for that middlebox to derive the cryptographic keys. Each middlebox shall receive a contribution from both client and server in order to be authorised to access 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 `MiddleboxKeyMaterial` message through the chain of every middlebox participating in the connection; the contribution will not be present in this message if the endpoint wishes to withhold access permission to the context. The middleboxes further process these `MiddleboxKeyMaterial` messages before forwarding to the next entity in such a way as 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 adds cryptographic verification messages (MiddleboxFin) of the handshake with each middlebox, a feature not present in the original mcTLS specification.

- The purpose of each of the contexts is for the application layer to determine. Until the first CipherChangeSpec message, there shall be a single context with ID zero and the cipher suite shall be TLS_NULL_WITH_NULL_NULL. Application data shall not be sent until after the contexts have been agreed and the first handshake has fully completed. After this, a cipher suite with a non-NULL integrity algorithm shall be selected and context zero shall then be reserved for future renegotiation and alert records. Key negotiation and alert messages shall not be transmitted in any other context. Application data shall not be transmitted in context zero. All middleboxes shall have full-access to context zero to take part in any renegotiation.

- Only the client endpoint may initiate a renegotiation of the security parameters of the session by sending a new `ClientHello`. A server endpoint may request renegotiation by sending a `HelloRequest`. A renegotiation may change the number of contexts and/or the permissions granted to any middlebox. Re-negotiation shall only offer cipher suites with a non-NULL integrity algorithm.

```
CLIENT                MIDDLEBOX 1    ...    MIDDLEBOX N               SERVER

ClientHello ------------x------------x--------------x------------------>

            - - - MiddleboxHello -  -o- - - - - - - o- - - - - - - - - ->

            - - - - - - o - - -MiddleboxHello- - - -o- - - - - - - - - >
  …
            - - - - - - o - - - - - - -o- - - - - MiddleboxHello - - - - >
ServerHello
Certificate
ServerKeyExchange
ServerHelloDone
            <-----------x------------x--------------x----------------

            <- - - - - -o - - - - - - -o- - - - -MiddleboxHello- - - - -

                                                  MiddleboxCert
                                               MiddleboxKeyExchange
            < - - - - - - o - - - - - - -o- - - - -MiddleboxHelloDone------->

            <- - - - - -o - - - MiddleboxHello- - - - o- - - - - - - -

                                MiddleboxCert
                            MiddleboxKeyExchange
            < - - - - - - o - - MiddleboxHelloDone------o----------------->
```

```
            <- - -MiddleboxHello- - - o- - - - - - - -o- - - - - - - -

                   MiddleboxCert
                MiddleboxKeyExchange
            < - -MiddleboxHelloDone --o---------------o----------------->

ClientKeyExchange
ChangeCipherSpec -------x------------x---------------x------------------>

MiddleboxKeyMaterial[M1]
               - - MiddleboxKC[M1]- -o- - - - - - - -o- - - - - - - - - ->

MiddleboxKeyMaterial[…]
               - - - - -o- - - MiddleboxKC[…] - - - - o- - - - - - - - - >

MiddleboxKeyMaterial[MN]
              - - - - -o - -  - - - -o- - - -  MiddleboxKC[MN] - - - - ->

MiddleboxKeyMaterial[S]
               - - - - -o - - - - - -o- - - - - - - -o- - - - - - - -- ->

ChangeCipherSpec
MiddleboxKeyMaterial[C]
              <---------o------------o---------------o---------

MiddleboxKeyMaterial[MN]
              <- - - - -o - - - - - -o- - - - - MiddleboxKC[MN]-

MiddleboxKeyMaterial[…]
              <- - - - -o- - - -MiddleboxKC[…]-  - - o- - - - - -

MiddleboxKeyMaterial[M1]
              <- - MiddleboxKC[M1]- -o- - - - - - - -o- - - - -

Finished
            ------------x------------x---------------x----------
>MiddleboxFin[C,M1]
            - -MiddleboxFin[M1,S]- -o- - - - - - - -o - - - - ->

MiddleboxFin[C,...]
           - - - - - - - - - MiddleboxFin[…,S]- - - o - - - - -->
MiddleboxFin[C,Mn]
            - - - - - - - - - - - - -o - - - MiddleboxFin[Mn,S]->

Finished
            <-----------x------------x---------------x---------
MiddleboxFin[S,Mn]
            < - - - - - - - - - - - - -o - - - - MiddleboxFin[Mn,C]-

MiddleboxFin[S,...]
            <- - - - - - - - - MiddleboxFin[C,…] - -o - - - - - -
MiddleboxFin[S,M1]
            <- MiddleboxFin[C,M1]- -o - - - - - - - o - - - - - -


Application Data                                      Application Data
            <------------x------------x---------------x--------->
```

Figure6: Handshake

In the figure, x indicates that the middlebox reads/processes and forwards the message; o indicates it just forwards it. The spaced dashed lines indicate a message which is sent piggy-backed on the previous one. Thus, two spaced dashed lines after each other indicates aggregated/recursive piggy-backing. For example, the middlebox certificate/key exchange messages are sent piggy-backed on the end-to-end `ServerKeyExchange` (towards the client). Note that

for Middleboxes, their Certificate, `KeyExchange` and `HelloDone` messages are sent piggy-backed toward the client, but sent explicitly (as separate messages) toward the server. `MiddleboxKeyMaterial[A]` denotes a message containing middlebox key shares from an endpoint directed to A and `MiddleboxKC[M]` denotes a key confirmation message from middlebox M to an endpoint. As seen, these are also piggy-backed (and aggregated) into forwarded `MiddleboxKeyMaterial` messages.

For the definition of the protocol, structures that are not defined in the document are defined in and unchanged from structures of the same name in [22].

4.3.2 Middlebox Configuration and Discovery

4.3.2.1 General

This clause describes options for how to configure or establish the middlebox list with the complete set of middleboxes. There are two main cases: static pre-configuration and dynamic discovery. The static pre-configuration shall be mandatory to support, the other is recommended to support, but shall be optional.

4.3.2.2 Static Pre-configuration

Here, it is assumed that by some means, the client is manually pre-configured with the complete set of middleboxes (their names/addresses as required in the middlebox list extension). The list is assumed to be arranged in network-topological order while the one-octet middlebox identities are not necessarily in order. All the middleboxes in the original list have the inserted field set to “false”. The client is furthermore assumed to have obtained the IP address of the first-hop middlebox, e.g. by DNS lookup of the middlebox address (name) field. It is also assumed that each middlebox knows or can obtain the IP address of the next-hop middlebox, the last middlebox being able to obtain the IP address of the server. The client now initiates the handshake by sending the `ClientHello` (including the middlebox list etc) to the first middlebox, which then forwards it to the next middlebox after inserting its own information into the handshake. The message thus flows through all middleboxes to the server and back (in reverse order of the middleboxes), etc.

4.3.2.3 Dynamic Discovery

4.3.2.3.1 General

In this case, the client may know some of the middleboxes, but not all. For example, one of the known (e.g. pre-configured as in clause 4.3.2.2) middleboxes or the server may request that one or more additional middleboxes should be added to the middlebox list. It is at the discretion of the endpoints whether to accept additional middleboxes that were not 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.

4.3.2.3.2 Non-transparent middleboxes

If the client is not aware of any a priori middlebox, one or more middleboxes may first be discovered via extensions to DHCP or DNS. If the middlebox(es) is (are) acceptable, the client shall address the first one in its `ClientHello` and proceeds as discussed above. As the `ClientHello` propagates via the middleboxes, a middlebox M1 may find reason to request addition of another middlebox, M2 say.

If M2 is to be placed between M1 and the server, M1 inserts an entry for M2 in the list (with inserted field set to “true”) and forwards the `ClientHello` toward the server (via M2). It is optional for M1 to send an Alert of type middlebox_required to the client as an advance notification. As the server will later copy the complete received list of middleboxes to the client in the `ServerHello`, both server and client will still have the opportunity to explicitly grant (or deny) access to M2.

If, on the other hand, M2 lies between the client and M1, M1 shall send an Alert of type “middlebox_required” toward the client, followed by a `MiddleboxHelloRequest` (with transparency field set to false), requesting it to restart the negotiation with the extended set of middlebox(es) in the path. If M2 is accepted the original transport layer connection to M1 shall be closed by the middlebox or client which made the connection. That middlebox shall then set up a new connection to M2 which shall in its turn set up a new connection to M1. M1 may discard the original proposal and regard the new ClientHello as the start of a new session.

The server shall inspect the list of middleboxes received in the `ClientHello` extension (including middleboxes that were dynamically inserted) and decide if the set of middleboxes is acceptable. If this is the case, the `ServerHello` response shall include the same middleboxes in the middlebox list extension of the `ServerHello`.

Draft

However, the server may also wish to add further middleboxes. If this is the case, the server shall respond to the client with an `Alert` of type middlebox_required and a `MiddleboxHelloRequest` containing the additional middleboxes. New transport layer connections shall be established at least from the end of the original chain of middleboxes which is still intact. This is similar to a middlebox inserting another middlebox before itself in the chain.

The client shall decide whether to accept the complete set of middleboxes. If required, the client shall re-start the handshake including middleboxes that were not on-path in the original handshake. If the client chooses to keep its transport layer connection at least the middlebox where the new path differs from the original one shall close the previous connection and establish a new connection to the first inserted one.

4.3.2.3.3 Transparent middleboxes

In this case, an on-path middlebox which is not explicitly visible as a separate entity on the IP layer wishes to add itself to the middlebox list. This situation can occur e.g. if the middlebox is located in a default gateway, a firewall, inside a mobile operator network, etc.

In this case, the middlebox intercepts the ClientHello, returns an Alert of type "middlebox_required", followed by a MiddleboxHelloRequest with the transparency field set to true. The source IP address in the packet is set to be equal to the destination IP address of the client's Hello message. Again, the client decides whether to accept the middlebox and proceeds as above.

Note that this way of handling additional middleboxes requires that the added middlebox remains (transparently) on-path for the duration of the session.

4.3.2.3.4 Semi-transparent middleboxes

This is the case where the middlebox M2 is on-path in a connection with M1 as endpoint, not included in the list and explicitly visible from the previous box. Then M2 will not send the `ClientHello` any further but instead send an Alert of type "middlebox_required" as it would have been sent from M1. After this the remaining will be as for the non-transparent case.

4.3.3 Session Resumption

As with standard TLS, TLMSP provides an abbreviated handshake to resume a previously established session, refreshing the keys but keeping the previous cipher suite and compression algorithm. In TLMSP, only the per-context reader/writer traffic protection keys shall be refreshed (see clause 4.3.7), the pairwise keys between client/middlebox respectively server/middlebox shall not be refreshed.

Similarly to TLS, the server may, on the initial handshake, indicate a session ID in its hello message, indicating to the client that the server may be willing to cache the session state for later resumption. In TLMSP, the client may receive one such session ID also from each of the middleboxes. The middleboxes' session IDs are likely to differ from that of the server, even though they logically are associated with the same session. The session may be resumable, but only if the server and all middleboxes have provided a session ID. From a middlebox point of view, each session is composed of one session toward the client and one toward the server, but they shall be locally identified at the middlebox by the same bi-directional session ID. If the client wishes to enable later resumption, the client shall store this list of N+1 session ID, where N is the number of middleboxes.

To later request session resumption, the client shall include the list of the N+1 session ID in its client hello message. Here, the middleboxes' N session IDs are provided via the middlebox list extension (see clause 4.3.5), whereas the corresponding value for the server is provided in the usual way, via the client hello. If a middlebox recognizes the session ID and is willing to resume the session, it shall indicate this toward the server by adding the same session ID in its hello toward the server (which is piggybacked in the same signalling message), otherwise the session ID shall be empty. The server finally receives the hello messages. If it recognizes its own session ID, and, if session IDs of all middleboxes are present and identical to those in the middlebox list extension, it may choose to allow resumption. It does so by signalling the same (own) session ID back toward the client (via all middleboxes) in its server hello response.

TLS also supports a (server-side) stateless resumption via session tickets. In TLMSP the support for this mechanism may vary between middleboxes and server and between different middleboxes. If the client indicated support for tickets in the original handshake extension, the client may have received tickets from some middleboxes and perhaps from the server, but not from all entities. The client may therefore attempt to initiate resumption by including a previous session ID in the initial handshake directed toward some entities and may at the same time include previously received tickets, plus a dummy session ID, in the handshake toward other entities. In any case, each middlebox shall indicate toward the

Draft

server that it accepts the client's resumption proposal by copying the same session ID (a real one or the dummy) that they received from client when generating their hello messages toward the server. Otherwise, the middlebox-provided session ID shall be empty. The server may proceed with resumption, but only if it received positive confirmation (matching session IDs) from all middleboxes. Thus, an entity that accepts the resumption (regardless of whether based on session ID or ticket) shall always indicate this by forwarding/responding with the same session ID as received from the client. When the client later receives positive resumption notification, the client shall discard the dummy session IDs, after verifying their presence. The client may also receive renewed tickets, which it may store for future resumptions of the same session.

This in principle concludes the resumption process. The only remaining step is for the client, the server and the set of middleboxes to refresh each reader/writer key for each context to which it has access. This is done using the previous key shares of the client and server, but now using the new client and server random values, see clause 4.3.7.

Resumption shall be done using the same cipher suite and compression algorithms as the original session.

4.3.4 Handshake Message Types

The new handshake message has been assigned IDs as follows. The `HandshakeType` enum type is therefore:

```
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),
  middlebox_hello(40), middlebox_certificate(41), middlebox_key_exchange(42),
  middlebox_hello_done(43), middlebox_key_material(44), middlebox_key_confirmation(45),
  middlebox_hello_request(46), middlebox_finished(47), (255)
} HandshakeType;
```

Details of the new handshake messages are provided below.

4.3.5 Handshake Extension

TLMSP defines a new TLS handshake extension to the Hello messages in the form of a MiddleboxList with `typeID` 0xff06. The extension contains: a list of middleboxes and optionally session IDs. Each entry specifies the middlebox's address, a unique ID, a list of contexts for which the middlebox has read access and a list of the contexts for which the middlebox has write access. If a middlebox has write access to a context/slice then read access in implicit. The session IDs are included if the client seeks to resume a previous session.

The middlebox ID values 0x00 – 0x02 are reserved with 0x01 reserved for the client and 0x02 reserved for the server. The middlebox may be assigned any value in the range 0x03-0xff. The list of middleboxes shall be ordered by the connections established from client to server.

The format for the entries in the middlebox list:

```
struct {
    struct {                            /* middlebox identification */
        Address address;
        uint8 middlebox_id;
    } middlebox;
    enum { false(0), true(1) } inserted; /* true if the middlebox is dynamically inserted */
    NewSessionTicket ticket;            /* may be empty */
    SessionID session_id;               /* may be empty */
    select (session_id_present) {
       case true: struct {};            /* resume always use the same contexts  */
       case false: struct {
                       struct {         /* one of entry for each context */
                           uint8 context_id;
                           enum { read(0), write(1), (255) } authorization;
                       } contexts<1..?>;
                   };
    };
} MiddleboxInfo;
```

where the (possibly empty) `NewSessionTicket` and `SessionID` are defined as in TLS. It is thus assumed that session resumption shall always use the same contexts for the same purposes (with the same cipher suite) as the original session.

The server shall in its ServerHello response include the list of all middleboxes that it received via the ClientHello (i.e. including additional middleboxes that may have been added dynamically in-band as discussed in clause 4.3.2.3), and furthermore extended by any middlebox requested for addition by the server (see also clause 4.3.2.3).

The second extension to the hello is a list of the context IDs and descriptions. A context description is a string meaningful only to the application; TLMSP does not use it. This extension has typeID 0x<TBD> and is defined by:

```
struct {
     uint8 context_id;
     enum { unconfirmed(0), confirmed(1), audit_trail(2), (255)} audit;
     enum { unconfirmed(0), confirmed(1), (255)} origin_verification;
     opaque purpose<0..?>;
  } context<1..?>;
```

The audit field is used to request confirmation from middleboxes that the containers associated with the context have passed via them, allowing them to act on the content (if authorized), i.e. an audit trail. Specifically, for contexts with this field set to the value confirmed, all middleboxes authorized to access these contexts shall process and add additional forwarding MAC values to the containers as described in clause 4.2.6.3 and 4.2.7

It is strongly recommended to use this feature, in particular when middleboxes are invoked for filtering purposes.

Additional values of audit may be defined in the future, specifying that middleboxes add information on their processing to the auditRecordcontainers . Currently only the additional value audit_trail has been specified for this purpose, but its usage and interpretation is left for future study.

The origin verification field tells if the original sender should add forwarding MACs directed to specific middleboxes.

The use of the purpose field is application dependent.

4.3.6 Middlebox Hello

During the initial Handshake, the middlebox shall include its supported cipher and compression suites. Also, when the ClientHello message contains a sessionID (and optionally a Ticket) indicating desire to resume a session, a middlebox shall acknowledge this through copying/inserting the session ID to their own hello. Middleboxes shall therefore insert a MiddleboxHello message of the same format as TLS ClientHello, but with an added middlebox identity field, i.e.

```
struct {
   uint8 middlebox_id;
   ClientHello mbox_hello;
} MiddleboxHello;
```

Except for the random value and extension fields, middleboxes shall provide the same content in their Middlebox Hello directed to client and server.

4.3.7 Middlebox Certificate

This is identical in format to a Server Certificate, but with an added idenity field of the middlebox

```
struct {
     Uint8 middlebox_id;
     Certificate cert;
} MiddleboxCertificate;
```

Note that this message piggy-backs on the ServerHello, which may include dynamically added middleboxes that were not present in the ClientHello. A middlebox shall therefore obtain its id to be used from the ServerHello.

4.3.8 Middlebox Key Material

TLMSP introduces a new handshake message for delivering context key material to the middleboxes. During the handshake, both the client and server shall send MiddleboxKeyMaterial messages through the chain of all middleboxes, providing key shares for each middlebox (and the other endpoint). The payload of each message shall be encrypted and integrity protected with a symmetric key shared between the endpoint sending the message and the middlebox (or other endpoint) receiving it. The message contains a partial secret for each context to which a middlebox

Draft

has access. Only with the partial secrets from both client and server can the context key be derived; knowledge of either partial secret in isolation does not reveal any knowledge of the context key.

The `writer_contribution` part shall be included only for contexts to which write access is granted. This structure is protected by the keys shared (only) between the endpoint and the receiving entity (derived as specified in clause 4.3.10.2), using the same encryption and compression algorithms and integrity check mechanism that has been agreed for the end to end encryption. The only difference is that only one integrity check value is required and the mechanism used is the same one as is used to add the reader MAC. The writer MAC and forwarding MAC shall be omitted. (Note that one `MiddleboxKeyMaterial` message will always be directed toward the other endpoint, but for such client-server messages, the reader MAC serves the same function as the forwarding MAC since a distinct key, unknown to middleboxes, is still used.)

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


integrity-protected struct {
    uint8 middleboxID;
    uint8 contextID;
    KeyMaterialContribution readerContribution;
    select (middlebox_has_write_access_to_context) {
        case false: {};
        case true: KeyMaterialContribution writerContribution;
} MiddleboxKeyMaterial;
```

Above, `middleboxID` shall be the identity of the entity (middlebox or endpoint) to which the message is directed and shall be left un-encrypted. Encryption shall be applied to the `contextID` and the contribution(s).

```
stream-ciphered struct {
    opaque IV[SecurityParameters.record_iv_length];
    opaque content[TLMSPCompressed.length];
    opaque mac[SecurityParameters.mac_length];
} StreamCipherContribution;

struct {
    opaque IV[SecurityParameters.record_iv_length];
    block-ciphered struct {
        opaque content[TLSCompressed.length];
        opaque mac[SecurityParameters.mac_length];
        uint8 padding[BlockCipherContribution.padding_length];
        uint8 padding_length;
    };
}  BlockCipherContribution;

struct {
    opaque IV[SecurityParameters.record_iv_length];
    aead-ciphered struct {
        opaque content[TLMSPCompressed.length + D + SecurityParameters.mac_length];
    };
} AEADCipherContribution;
```

The MACs above shall be calculated as

```
    MAC(A_to_B_MAC_key, MiddleboxKeyMaterial.middleboxID +
                   MiddleboxKeyMaterial.contextID +
                   MiddleboxKeyMaterial.readerContribution.IV +
                   MiddleboxKeyMaterial.readerContribution.content +
                  [MiddleboxKeyMaterial.writerContribution.IV +
                   MiddleboxKeyMaterial.writerContribution.content]
```

where `A_to_B_MAC_key` is the MAC key shared between endpoint A and middlebox B and where the writer contributions are included when present. As encryption key, the shared key `A_to_B_encryption_key` shall be used, see clause 4.3.10.2.

For efficiency reasons, the endpoints may use the aforementioned “piggy-backing” to transmit `MiddleboxKeyMaterial` information element directed to several middleboxes in the same TLMSP record.

Draft

4.3.9 Middlebox Key Confirmation

The Middlebox Key Confirmation piggy-backs on the Middlebox Key Material signalling and is intended to provide proof toward an endpoint that each middlebox has successfully obtained correct (partial) key material from the other endpoint for all contexts to which the middlebox is granted access.

The `MiddleboxKeyConfirmation` message shall be format equivalent to the `MiddleboxKeyMaterial` (defined in clause 4.3.8 above), except that the `middlebox_id` field here refers to the sending entity rather than the receiving. To allow an entity to distinguish whether a message contains `MiddleboxKeyConfirmation` or `MiddleboxKeyMaterial`, the message type is used (see below).

For each received `MiddleboxKeyMaterial` message, MK, directed to the middlebox, exactly one `MiddleboxKeyConfirmation` message, MC, shall be generated as follows by the middlebox. The middlebox shall set the `middlebox_id` field of MC to its own identity, and set the `context_id` of MC to the same value as indicated by the received MK. The middlebox shall further generate the `readerContribution` field of MC, using as the to-be-protected payload

- the key-share (client_reader_secret) received in MK from the client, when forwarding the message MC in client-to-server direction, and,
- the final combined key (the derived reader_key_block), when forwarding the message MC in the server-to-client direction.

The above shall be repeated using the `client_writer_secret` and `writer_key_block` fields if write access is granted for the context. When the middlebox (B) generates the protected `MiddleboxKeyConfirmation` message, it shall use the symmetric key shared with the destination endpoint (A, i.e. the server or client), `B_A_key_block` according to clause 4.3.10.2 as basis for protection key. Only one MAC field (the reader MAC) shall be used (also generated based on the key shared with the endpoint).

The so generated `MiddleboxKeyConfirmation` message MC shall then replace (via piggy-backing, see below) the corresponding received `MiddleboxKeyMaterial` message MK when forwarding the signalling toward the destination. Any additional `MiddleboxKeyMaterial` message (not directed toward this middlebox) shall be forwarded without further processing/action. This means that an original (complete) message that was sent from an endpoint source, i.e. that initially contained `MiddleboxKeyMaterial` shares for middleboxes M_1, M_2, .., M_N (in network topological order) and D (the destination endpoint, server or client), shall after middlebox M_j contain the `MiddleboxKeyMaterial` for middlebox M_{j+1}, M_{j+2}, …, M_n, and destination D, and, in addition, `MiddleboxKeyConfirmation` messages from middlebox M_1, M_2, …, M_j, directed to D.

Note that an entity receiving a TLMSP record containing a mix of `MiddleboxKeyMaterial` and `MiddleboxKeyConfirmation` containers can easily distinguish between them since they have different messages.

When the destination (server or client) ultimately receives the single `MiddleboxKeyMaterial` message (from the other endpoint) and the set of `MiddleboxKeyConfirmation` messages, it shall verify that it received `MiddleboxKeyConfirmation` messages from all middleboxes, and for each context for which access to that middlebox was granted. The endpoint shall then decrypt, verify integrity, and finally confirm that each of the retrieved decrypted secret agrees with the expected value. That is,

- the *server* shall verify that the secret (client_reader_secret and/or client_writer_secret) is equal to the copy of the client's share as received directly from the client (in its own, separate `MiddleboxKeyMaterial` message), and,
- the *client* shall verify that the secret (`reader_key_block` and/or `writer_key_block`) is equal to the final combined key derived from both client's and server's shares (the latter received directly from the server).

If any of these checks fail, the endpoint shall <TBD>.

As a final note, observe that since each middlebox replaces a `MiddleboxKeyMaterial` message with precisely one `MiddleboxKeyConfirmation` message, where the latter is essentially as large as the first, the total transferred data does not grow in size.

Draft

4.3.10 Key Generation

4.3.10.1 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 key exchange messages respectively. These are used to generate the client-server premaster secret as per the specification in TLS 1.2. To generate the endpoint-middlebox premaster secret, the same endpoint ephemeral public key as above shall be re-used but combined with unique, per-middlebox ephemeral keys. To this end, each middlebox will also send hello messages containing its nonce, certificate and ephemeral public key. Different ephemeral public keys are used by the middlebox for the exchange with the client and the server and both shall be included in the middlebox key exchange message. 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 (if it has not then they should abort the key exchange with a fatal error). The same middlebox nonce is used for both client-middlebox and server-middlebox exchanges.

The client-server premaster secret is generated as per TLS 1.2

The client-middlebox and server-middlebox premaster secrets are generated using the same ephemeral keys for the client and server but the corresponding middlebox ephemeral key instead.

4.3.10.2 Premaster and Master Secret and Key Generation

The `pre_master_secret_A_B` is generated in a way specific to the cipher suite in use. The master key shared between precisely two parties, A and B (two endpoints or endpoint-middlebox), is generated as:

```
master_secret_A_B = PRF(pre_master_secret_A_B,
                        "master secret" + IDList,
                        Hello{a}.random + Hello{B}.random)[0..47];
```

Here, `IDList` shall be the hash of the concatenated list of the following identities, in the stated order:

1. the client ID, when available (e.g. via a certificate), followed by
2. all middlebox (complete) certificates, in the same order as in the MiddleboxList, followed by
3. the server certificate.

The same cryptographic hash as that used in the PRF shall be applied.

The endpoint encryption integrity keys and the IV's for communication between A and B are then generated as:

```
A_B_key_block = PRF(SecurityParameters.master_secret_A_B,
                "key expansion",
                SecurityParameters.B_random +
                SecurityParameters.A_random);
```

The key block is then partitioned into

```
A_to_B_MAC_key[SecurityParameters.mac_key_length];
B_to_A_MAC_key[SecurityParameters.mac_key_length];
A_to_B_Encryption_key[SecurityParameters.enc_key_length];
B_to_A_Encryption_key[SecurityParameters.enc_key_length];
```

When A and B are endpoints, a message that is correctly MAC'd with these keys shall 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 `MiddleboxKeyMaterial` messages to each other containing the contributions.

Keys between client (or server) and middlebox N are generated in the above way for A = client (or server) and B = middlebox[n], the middlebox with (octet) identity n.

These keys, known only to one endpoint and one middlebox, are used in the encryption of the `MiddleboxKeyMaterial` message containing the contribution. The MAC key is also used when a middlebox modifies or inserts new containers and authenticates it via the forwarding MAC, and also when audit trail is requested through use of forwarding MACs, see clause 4.2.7.

Draft

4.3.10.3 Context Specific Keys

For the context keys, the client and server generate two pseudo random partial secrets for each context, a client (server) read secret and a client (server) write secret. Partial secrets for different contexts shall be cryptographically independent. These partial secrets will only be sent to middleboxes to which the endpoint is willing to authorise that level of access, encrypted and integrity protected with the keys derived above. Each party with authorised access to a particular context, i, will notionally derive 4 keys associated with each context.

- client_reader_enc_key_i: Encrypt/Decrypt data from the client to server.
- server_reader_enc_key_i: Encrypt/Decrypt data from the server to client.
- client_reader_mac_key_i: Compute reader MAC for data from client to server.
- server_reader_mac_key_i: Compute read MAC for data from server to client.

When also write access is granted, two additional keys are derived

- client_writer_mac_key_i: Compute writer MAC for data from client to server.
- server_writer_mac_key_i: Compute writer MAC for data from server to client.

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

After receiving a `MiddleboxKeyMaterial` message from each endpoint (containing client_reader_secret and server_reader_secret, respectively), all parties compute the context reader (and writer) keys for the contexts they can access. For each context i, each party uses the partial secrets (one from the client and one from the server) to compute two blocks of key material:

```
reader_key_block_context_i = PRF(client_reader_secret_i + server_reader_secret_i,
                    "reader keys" + i,
                    SecurityParameters.client_random + SecurityParameters.server_random)

writer_key_block_context_i = PRF(client_writer_secret_i + server_writer_secret_i,
                    "writer keys" + i,
                    SecurityParameters.client_random + SecurityParameters.server_random)
```

where i is the octet context identifier. Each `reader_key_block_i` is partitioned into associated `client_reader_enc_key_i`, `server_reader_enc_key_i`, `client_reader_MAC_key_i`, `server_reader_MAC_key_i`; each `writer_key_block_i` is similarly partitioned into `client_writer_MAC_key_i` and `server_writer_MAC_key_i`. A middlebox shall use these keys in dependence of the direction, i.e. keys prefixed by client shall be used for containers travelling from client to server, etc.

On session resumption, the previously established keys shall be refreshed by mixing the existing endpoint secrets for each context with the new client and server random values:

```
reader_key_block_new_context_i = PRF(client_reader_secret_i + server_reader_secret_i,
                         "reader keys",
                         i,
                         SecurityParameters.client_random_new +
                         SecurityParameters.server_random_new);

writer_key_block_new_context_i = PRF(client_writer_secret_i + server_writer_secret_i,
                          "writer keys", i,
                          SecurityParameters.client_random_new +
                          SecurityParameters.server_random_new);
```

which are then partitioned as above.

4.3.11 MiddleboxHelloRequest

This message is sent in response to a `ClientHello`, requesting the client to (re)start the handshake, but including one or more additional middleboxes. The format is a list containing entries of type

```
struct {
    struct {                                /* middlebox identification */
        Address address;
        uint8 middlebox_id;
```

```
    } middlebox;
    enum { false(0), true(1) } transparency; /* is the middlebox transparent or not */
    struct {                              /* one entry for each requested context */
         uint8 context_id;
         enum { read(0), write(1), (255) } authorization;
       } contexts<1..?>;
} MiddleboxHelloRequest;
```

4.3.12 Finished Message

4.3.12.1 Middlebox Finished

A special finished message is added, directed toward/from middleboxes. It shall have format

```
struct {
   Uint8 middlebox_id;
   opaque verify_data[verify_data_length];
} MiddleboxFinished;
```

Except the middlebox_id, which is the identity of destination/origin the middlebox, the format is identical to TLS own Finished message.

Different data sets need to be included in hash computations than the "standard" finished message sent between endpoints and the `MiddleboxFinished` message since not all parties have access to the complete set of messages exchanged during the handshake. The order of inclusion of data shall agree with the order of the handshake protocol specification.

4.3.12.2 Hash Computation in (endpoint) Finished Message

4.3.12.2.1 General

When computing the hash included for verification purpose in the client-server Finished messages, client and server shall omit information elements that were inserted ("piggy-backed") by middleboxes when the client and server may not know of or have the same view on these messages. Messages that were inserted by middleboxes are usually recognizable via the dedicated message types used to distinguish middlebox handshake messages from those of client/server. The included messages are the same as in TLS, but with the following messages or parts of messages being omitted:

- Entries in the middlebox list extension in the `ClientHello` with inserted field set to "true",
- `MiddleboxKeyExchange`,
- `MiddleboxKeyMaterial` message directed to a middlebox (non-endpoint),
- `MiddleboxKeyConfirmation messages`,
- The random value of `MiddleboxHello` and any possible extensions included in the `MiddleboxHello`.
- `MiddleboxFinished` messages.

Thus, note that the following middlebox related messages shall be included (which are always sent as identical copies toward both client and server):

- The `Certificate` messages inserted by middleboxes
- The following parts of the `MiddleboxHello`
    - The list of middlebox supported cipher suite(s)
    - The list of middlebox acknowledged session IDs, when resumption is performed.
- `MiddleboxKeyMaterial` message directed to the other endpoint.

Draft

4.3.12.2.2 Hash Computation in Middlebox Finished

This hash computation shall include

- `ClientHello`, except entries in the middlebox list extension of the `ClientHello` with inserted field set to "true",
- `ServerHello`,
- Server `Certificate`, the set of Middlebox `Certificate` messages, and (if present) Client `Certificate`,
- Client and Server `KeyExchange`,
- `ServerHelloDone`,
- `MiddleboxKeyMaterial`, directed between endpoints,
- `ChangeCipherSpec`,
- The following messages, as received from or directed to the middlebox with which the `MiddleboxFinished` is associated
    - `MiddleboxHello` (including extensions, if present),
    - `MiddleboxKeyExchange`,
    - `MiddleboxHelloDone`.

Note that `MiddleboxKeyMaterial` and `MiddleboxKeyConformation` related to a specific middlebox shall not be included as they are explicitly verified on reception.

4.4 Alert Protocol

4.4.1 Alert Message Types

TLMSP introduces some new Alert messages,

```
enum {
    close_notify(0), unexpected_message(1),
     … ,                                  /*    the existing TLS alert codes           */

    middlebox_authorization_failure(170),    /*    endpoint does not accept middlebox    */
    middlebox_required(171),                 /* one or more middleboxes missing from list */
    unknown_context(172),           /* entity does not recognize a context or its purpose */
    unsupported_context(173), /* middlebox can not perform requested operation on context */
    middlebox_key_verify_failure(174),
    bad_reader_mac(175),                     /*     reader MAC failed to verify            */
    bad_writer_mac(176), /* dito, writer MAC (end-point MAC handled via "bad_record_mac") */
    …,
    (255)
} AlertDescripton;
```

The alert levels of the additional messages are:

- `middlebox_required`: AlertLevel warning
- `middlebox_authorization_failure, unknown_context, middlebox_key_verify_failure, bad_reader_mac, bad_writer_mac`: AlertLevel fatal.

4.4.2 Change Cipher Spec Protocol

TLMSP does not impose changes on the TLS ChangeCipherSpec protocol as such: as in TLS it consists of one single message. Note however the following particulars on the effect of issuing the ChangeCipherSpec message. Since middlebox processing (read, write) of application data fragments is not possible until the middleboxes have received

Draft

their read/write keys for the applicable contexts, which occurs after the ChangeCipherSpec message, the following applies:

- The selected cipher suite shall start to be applied immediately following the ChangeCipherSpec message, but initially only for the remaining handshake messages (i.e. `MiddleboxKeymaterial`, `MiddleboxKeyConfirmation` and Done), i.e. for messages associated with context zero.
- Immediately after the `Done` handshake message, then, the selected cipher suite shall be applied to all application layer fragments (i.e. for all other contexts).

4.5 Compatibility Mode

[tbd]

Draft

5 Mapping to MSP Abstraction

[tbd]

6 Trust Model

[Editor's note: this clause will define the trust model under which TLMSP is assumed to use]

Draft

7 Security Considerations

[Editor's note: this clause will (informally) analyse the security properties of TLMSP, in particular how the recent mcTLS attacks are mitigated. Below are some initial observations.]

7.1 Protection Against mcTLS Attack

It has been observed [i.3] that the original mcTLS proposal suffers from a vulnerability by which a malicious endpoint, e.g. the server, may undo operations performed by a middlebox. For example, a middlebox detecting malware content from a server may be ineffective at removing this malware if the server can access the remaining path between middlebox and client and there re-insert the malware again. This is caused by the fact that the server generally has access to all keys used by middleboxes to authenticate their inbound/outbound containers. TLMSP addresses this issue by (as an option) 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 receiving endpoint. The receiving endpoint will be able to verify that all middleboxes received containers that were unaltered from when they left the previous middlebox. More precisely, while a middlebox cannot locally verify this property, due to a local verification done by the middlebox in combination with a final verification at the receiving end-point, this property is established at the endpoint. The obtained end-to-end verification is implicit: it implies that each middlebox had the opportunity to act on authenticated containers, but it does not per se prove that the middlebox actually performed the "right" action. The latter is left as an assumption of the trustworthiness of the middleboxes.

The cryptographic MAC transform used for the so-called forwarding MAC is slightly different from that used directly within the selected cipher suites. This is due to the following reason. The forwarding MAC generated at a middlebox is a cryptographic checksum of the inbound/outbound reader MACs. When the middlebox generates the forwarding MAC, it therefore "locks" the value of the inbound/outbound reader MACs so that that they cannot later be changed by the server (since the server does not know the MAC key of the forwarding MAC of a middlebox). However, it does not protect against a server who is able to modify the *content while leaving the two reader MACs intact*. Specifically, since the server does know the reader MAC key, the question becomes: can the server launch a "second-preimage" attack on the (reader) MAC transform using the knowledge of the key. For MACs such as HMAC, this is believed infeasible due to the cryptographic properties of the hash functions involved. It is however not true for all MACs. For "number theoretic" or "algebraic" MACs (such as that of AES GCM for example), an entity knowing the key may very well be able to generate a second message, having a MAC identical to that of a given message. For this reason, the reader MACs used as inputs to the forwarding MACs shall make use of an additional cryptographic component that adds the necessary "collision resistance". This is the reason for using an extra cryptographic hash computation in the forwarding MACs.

Note that unless forwarding MACs are used an attacker which has the possibility to modify data after a middlebox can delete all inserted containers unnoticed by the endpoint. Likewise, without forwarding MACs, if the attacker is also able to read incoming containers to a middlebox it could undo the operation of the middlebox.

The forwarding MAC mechanism of TLMSP does not provide cryptographic proof to middleboxes that any other middlebox has inspected or otherwise processed the content. A threat could be that a middlebox located after a malware detector could be fooled to act on potentially malicious content if the malware detector (for some reason) has been bypassed. However, the endpoint can still detect that the data was inserted after the malware detector and therefore the data cannot be considered to have be checked .

7.2 Inter-session Assurance

Similar considerations appear with caching middleboxes. Even if the original content stored in the cache was delivered via TLMSP and was thoroughly inspected by some middlebox before it was stored, TLMSP as such does not propagate assurance information from one TLMSP session to another. Some other client who 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, it may very well be that later downloads of the same content do not even use TLMSP but instead uses TLS. On the other hand, the audit mechanism of TLMSP could potentially be used to provide evidence that content is trustworthy. This requires that the audit records are constructed in such a way that they are universally/publicly verifiable.

Draft

Annex A (normative): Defined Cipher Suites

A.1 Key Exchange

The cipher suites defined below in Appendixes A.2 to A.4 are defined for TLMSP use. One of the following key exchange methods (as defined in TLS) shall be used.

- ECDHE_ECDSA. Recommended curves to use is TBD.
- DHE_DSS
- RSA (for authentication and premaster secret transport). In this case, the middleboxes shall act as clients toward the server, i.e. they select the premaster secret and encrypt it using the server's public key.

A.2 AES_{128,256}_GCM_SHA256

A.2.1 General

This cipher suited is defined as TLS_*_*_AES_{128,256}_GCM_SHA256 GCM for TLS1.2 (where * is to be replaced by one of the applicable key exchange mechanisms above) with the following exception. The IV shall instead have the following form: IV = M_ID || NONCE where

- M_ID is the one-octet middlebox identity for the sender ("0x00" for client and "0x01" for server). This value is not explicitly transmitted as it is either understood implicitly or available via the container header.

- NONCE is an 11-octet random value selected by the sender. This is the only part carried explicitly in the protected message.

(Note that internally, AES-GCM will use the above IV as the 96 most significant bit in the counter.) Thus, no implicit IV derived from the session master key is used. When a middlebox modifies or inserts a message, it shall generate a new NONCE and set M_ID accordingly. The inclusion of M_ID avoids the need for different entities (e.g. middleboxes) to synchronize their NONCE-generation schemes.

The additional authenticated data field shall, as already noted (refer to clause 4.2.6.1 for details), consist of

seq_num + TLMSPCompressed.type + TLMSPCompressed.version + TLMSPCompressed.container.flags + [TLMSPCompressed.container.mInfo] (*) + TLMSPCompressed.container.length

This cipher suite is mandatory to support by all TLMSP entities.

A.2.2 Additional MAC computations

When generating additional MAC values, i.e. the writer and forwarding MAC(s), only the GMAC function of AES GCM shall be used (see [NIST SP 800-38D] for a definition of GMAC) with the appropriate key (the writer key, etc). Conceptually, this consists of associating the additional authenticated data with the data (*) defined above, concatenated with the entire data fragment after it has been encrypted (with the reader key), but excluding any previously generated MAC values, already appended to the fragment. Thus, no additional encryption takes place.

A.3 AES_{128,256}_CBC_SHA256

This cipher suite is defined as TLS_*_*_AES_{128,256}_CBC_SHA256 for TLS1.2 with the following exception. The IV shall have the following form: IV = M_ID || NONCE where

- M_ID is the one-octet identity for the sender.

- NONCE is a 15-octet random value selected by the sender. This is carried explicitly in the protected message.

When a middlebox modifies or inserts a message, it shall generate a new NONCE and set M_ID accordingly.

Draft

A.4 AES_{128,256}_CTR_SHA256

This cipher suite consists of the counter-mode encryption part of AES_GCM (see Annex A.2 above), in conjunction with the HMAC_SHA256 MAC as defined for AES_CBC_SHA256. (The IV is thus to be generated in the same way as in Annex A.2).

A.5 Summary of Security Parameters

Cipher suite (Annex ref.)	enc_key_length	mac_key_length	fixed_iv_key_length	block_length
GCM (see A.2)	16 or 32	= enc_key_length	11	16
CBC (see A.3)	16 or 32	= enc_key_length	15	16
CTR (see A.4)	16 or 32	= enc_key_length	11	16

Draft

Annex B (normative): Authentication Extension for MNO Provided Middleboxes

B.1 Introduction

The mechanism described in this Annex is, similar to the support for the MiddleboxList, based on an extension of TLMSP. It is assumed that extensions to TLMSP are handled in the same way as extensions to standard TLS, i.e. that entities who do not support or even understand the extension silently discards the extension. This ensures backward compatibility and automatic fall-back to standard TLMSP (without using the extension).

The technical details of this Annex are optional to support by both client, middlebox(es) and server. It is however recommended that clients equipped with USIM cards (e.g. smartphones) implements and uses the extension described herein whenever it wishes to receive services by middleboxes provided by the MNO (Mobile Network Operator) issuing the USIM, for example, when connecting to an internet server via the MNO's network. Therefore, it is also recommended that middleboxes provided by MNOs implement this extension and uses it whenever the client, thorough the extension, makes an offer to do so. Mutual authentication between middlebox and client obtained through this extension will provide stronger assurance that the middlebox services are only provided to clients who subscribe to those services. It will therefore also, if applicable, enable more robust charging of the services.

Note 1: The client is assumed to be able to have prior knowledge of those middleboxes in the middlebox list that are associated with the MNO and which may therefore be able to support this extension. Exactly how the client obtains this knowledge is outside the scope of this specification, but may for example be done in conjunction to MNO configuration of the client. Even if the client incorrectly assumes a certain middlebox supports these extensions (or not), no adverse security issues exist; an error alert will be raised or a fall-back to standard TLMSP will occur.

Note 2: The extension may also be used by middleboxes not provided by the MNO, as long as the provider of those middleboxes have a trust relation with the MNO. Means for establishing this trust, setting up secure channels etc, are however outside the scope of this specification.

Note 3: As noted, the extension is transparent to middleboxes that do not support it. This means that when more than one middlebox is on the path between client and server, there may in general be a mix of middleboxes using this extension and middleboxes that do not.

B.2 Technical Details

B.2.1 General

A client wishing to make use of this extension shall first do GBA bootstrapping with the BSF (Bootstrapping Server Function) according to the GBA (Generic Bootstrapping Architecture) specification [3GPP TS 33.220, §4.5.2]. A middlebox supporting this extension is viewed as a NAF (Network Application Function) in GBA terminology and is assumed to follow the GBA-specified procedures, in particular observing the details below. Middleboxes not supporting this option will silently discard this extension and comply with normal TLMSP protocols.

When this extension is present in the `Hello` message from the Client (and optionally from the Server), this extension shall be included in the hash-calculation of the `Finished` message.

B.2.2 Client Hello

Recall that a TLS extension is defined as

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

In this case, the existing set of ExtensionType is augmented to include the new type

```
gba_client_authentication:
enum {
   …, gba_client_authentication(TBD), …
} ExtensionType;
```

The client shall indicate desire to use this extension (i.e. to allow middleboxes to authenticate it) by including, in the `ClientHello`, an extension with the `extension_type` value set to `gba_client_authentication` and the associated `extension_data` field shall contain a `BTID` value according to

```
opaque BTID<1..TBD>
```

where `BTID` is the GBA-defined B-TID value (obtained during GBA bootstrapping), i.e. an encoded NAI of format: base64encode(RAND)@BSF_servers_domain_name. (As noted elsewhere in this specification, all strings are encoded in UTF-8 format.)

A middlebox supporting this extension shall, when observing the presence of the above extension in the `ClientHello`, contact the BSF as indicated by the `BTID` of the extension data and request the NAF-key (Ks_NAF) according to [3GPP TS 33.220, §4.5.3]. When deriving or requesting the Ks_NAF, the client, middlebox (=NAF), and the BSF shall use as the NAF-Id the same middlebox address (the "address" field, excluding the one-octet "middlebox_id") as specified in the middlebox extension list.

At this point (assuming no errors) any middlebox that supports this extension (and the client) will have (or be able to derive), a pairwise unique, shared key Ks_NAF.

B.2.3 Middlebox Hello

A middlebox accepting to use GBA shall include the `gba_client_authentication` as extension in its MiddelboxHello toward the client.

B.2.4 Server Hello

A server that understands this extension may optionally include ("echo") the original client-side authentication extension as an extension in the `ServerHello`. Currently, such indication would not serve any purpose and shall not be acted upon by the client or middleboxes.

B.2.5 Server Key Exchange

Recall that in TLMSP, the middleboxes normally piggy-back their random values and Diffie-Hellman key- exchange parameters in the `ServerKeyExhange` response toward the client. When using this extension the `KeyExchanges` between client and middlebox serve no purpose and shall be omitted.

The middlebox's key exchange toward the server shall proceed according to standard TLMSP.

B.2.6 Middlebox Key Material

When generating (and processing) the `MiddleboxKeyMaterial` message between client and middlebox, the Ks_NAF shall be used as basis to protect those messages. Specifically, the symmetric key used to protect `MiddleboxKeyMaterial` messages shall be computed using the Ks_NAF in place of the master secret of clause 4.3.10.2, i.e.

```
client_middlebox_key_block = PRF(Ks_NAF,
        "key expansion",
        SecurityParameters.client_random +
        SecurityParameters.middlebox_random)
```

from which IV, Encryption- and MAC-keys are then obtained in the standard way. Authentication of the client toward the middlebox is then assured by successful verification of the associated MAC-values.

`MiddleboxKeyMaterial` messages from server to middlebox are not affected by this extension (nor are any other messages than those explicitly described).

History

Document history		
V0.0.1	June 2017	Created document with data from part 1
V0.0.1	June 2017	Created document with data from part 1
V0.0.2	August 2017	Changed to new template and updated with ETSI styles
V0.0.3	August 2017	Added information on discovery and key exchange
V0.0.4	January 2018	Major changes/rework to whole document
V0.0.5	January 2018	New contributions
V0.0.6	February 2018	New contributions
V0.0.7	April 2018	Major updates and review work
V0.0.8	April 2018	few updates to comply with ETSI Drafting Rules. Version made publicly available

Draft