```
Network Working Group                                           M. Badra
Internet-Draft                                   CNRS/LIMOS Laboratory
Intended status: Standards Track                               I. Hajjeh
Expires: October 30, 2011                                     INEOVATION
                                                          April 28, 2011
```

MTLS: (D)TLS Multiplexing
draft-badra-hajjeh-mtls-06.txt

Abstract

The (Datagram) Transport Layer Security ((D)TLS) standard provides connection security with mutual authentication, data confidentiality and integrity, key generation and distribution, and security parameters negotiation. However, missing from the protocol is a way to multiplex several application data over a single (D)TLS.

This document defines MTLS, an application-level protocol running over (D)TLS Record protocol. The MTLS design provides application multiplexing over a single (D)TLS session. Therefore, instead of associating a (D)TLS session with each application, MTLS allows several applications to protect their exchanges over a single (D)TLS session.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

(D)TLS ([RFC5246], ([I-D.ietf-tls-rfc4347-bis]) is the most deployed security protocol for securing exchanges, for authenticating entities and for generating and distributing cryptographic keys. However, what is missing from the protocol is the way to multiplex application data over the same (D)TLS session.

Actually, (D)TLS clients and servers MUST establish a (D)TLS session for each application they want to run over a transport layer. The client and the server MUST also duplicate the existing TLS/DTLS session for each application's stream/thread/connection (channel). However, some applications may agree or be configured to use the same security policies or parameters (e.g. authentication method and cipher_suite) and then to share a single TLS session to protect their exchanges. In this way, this document describes a way to allow application multiplexing over TLS/DTLS.

The document motivations included:

o TLS is application protocol-independent. Higher-level protocol can operate on top of the TLS protocol transparently.

o (D)TLS is a protocol of a modular nature. Since TLS is developed in four independent protocols, the approach defined in this document can be used with a total reuse of pre-existing (D)TLS infrastructures and implementations.

o It provides a secure VPN tunnel over a transport layer. Unlike "ssh-connection" [RFC4254], MTLS can run over unreliable transport protocols, such as UDP.

o Establishing a single (D)TLS session for a number of applications -instead of establishing a (D)TLS session per one of those applications- reduces resource consumption, latency and messages flow that are associated with executing simultaneous (D)TLS sessions.

o (D)TLS can not forbid an intruder to analyze the traffic and cannot protect data from inference. Thus, the intruder can know the type of application data transmitted through the (D)TLS sessions. However, the approach defined in this document allows, by its design, data protection against inference.

1.1. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. (D)TLS Multiplexing Overview and Considerations

This document defines an application-level protocol called (D)TLS Multiplexing (MTLS) to handle data multiplexing.

2.1. MTLS over TLS

If the client is willing to run MTLS over TLS, it MUST connect to the server that passively listens for the incoming TLS connection on the IANA-to-be-assigned TCP port (TBA). The client MUST therefore send the TLS ClientHello to begin the TLS handshake. Once the Handshake is complete, the client and the server can establish and manage many applications' channels using the MTLS requests/responses defined below.

2.1.1. Opening Channels

The sender MAY request the opening of many channels. For each channel, the MTLS layer generates and sends the following request:

```
   struct {
       uint8  type;
       uint16 length;
       opaque sender_channel_id[2];
       uint32 sender_window_length;
       uint32 sender_max_packet_length;
       opaque source_address_machine<1..2^16-1>;
       opaque source_port[2];
       opaque destination_address_machine<1..2^16-1>;
       opaque destination_port[2];
   } ChannelEstablishmentRequest;
```

type

The "type" field specifies the MTLS packet type (types are summarized below).

length

The "length" field indicates the length, in octets, of the current MTLS packet.

sender_channel_id

The "sender_channel_id" is the first half of the channel identifier. The second half is generated by the receiver of that MTLS packet.

sender_window_length

The "sender_window_length" field conveys the data length (in octets), specifying how many bytes the receiver of the packet can maximally send on the channel before receiving a new window length (available free space). Each end of the channel establishes a "receive buffer" and a "send buffer".

sender_max_packet_length

The "sender_max_packet_length" field conveys the data length (in octets), specifying the maximal packet's length in octets the receiver of that packet can send on the channel. The sender_max_packet_length is always smaller than the free space of the sender_window_length (the sender's "receive buffer").

source_address_machine and source_port

The "source_address_machine" MAY carry either the numeric IP address or the domain name of the host from where the application originates the data multiplexing request and the "source_port" is the port on the host from where the connection originated.

destination_address_machine and destination_port

The "destination_address_machine" and "destination_port" specify the TCP/IP host and port where the recipient should connect the channel. The "destination_address_machine" MAY be either a domain name or a numeric IP address.

The receiver decides whether it can open the channel, and replies with one of the following messages:

```
   struct {
       uint8  type;
       uint16 length;
       opaque sender_channel_id[2];
       opaque receiver_channel_id[2];
       uint32 receiver_window_length;
       uint32 receiver_max_packet_length;
   } ChannelEstablishmentSuccess;

   struct {
       uint8  type;
       uint16 length;
       opaque sender_channel_id[2];
       opaque error<0..2^16-6>;
   } ChannelRequestEchec;
```

The "sender_channel_id" and "receiver_channel_id" are the same generated during the channel establishment. The length conveys the data length of the current packet.

The field "error" conveys a description of the error.

Each MTLS channel has its identifier computed as:

```
       channel_id = sender_channel_id + receiver_channel_id
```

Where "+" indicates concatenation.

Note: channel_id may be susceptible to collisions. The receiver needs to take care not to choose a "receiver_channel_id" to avoid any collide with any of the established channel identifiers.

2.1.2. Closing Channels

The following packet MAY be sent to notify the receiver that the sender will not send any more data on this channel and that any data received after a closure request will be ignored. The sender of the closure request MAY close its "receive buffer" without waiting for the receiver's response. However, the receiver MUST respond with a confirmation of the closure and close down the channel immediately, discarding any pending writes.

```
   struct {
       uint8  type;
       uint16 length;
       opaque channel_id[4];
   } ChannelCloseRequest;

   struct {
       uint8  type;
       uint16 length;
       opaque channel_id[4];
   } ChannelCloseConfirmation;
```

The above two packets can be sent even if no window space is available.

2.2. MTLS Flow Control

The structure of the MTLS data packet is described below.

Each entity maintains its "max_packet_length" (that is originally initialized during the channel establishment) to a value not bigger than the free space of its "receive buffer". For each received

packet, the receiver MUST subtract the packet's length from the free space of its "receive buffer". For each transmitted packet, the sender MUST subtract the packet's length from the free space of its "send buffer". In any case, the result is always positive.

If the entity is willing to notify the other side about any change in the "max_packet_length", the entity MUST send a NewMaxPacketLength conveying the new "max_packet_length" that MUST be smaller than the free space of the entity's "receive buffer"

The free space of the "receive buffer" of the sender (resp. the receiver) MAY increase in length. The sender SHOULD send an Acknowledgment packet to inform the receiver about this increase, allowing this latter to send more packets but with length smaller or equal than the minimum of the "max_packet_length" and the "receive buffer" of the sender.

If the length of the "receive buffer" does not change, the Acknowledgment packet will never be sent.

In the case where the "receive buffer" of an entity fills up, the other entity MUST wait for an Acknowledgment packet before sending any more MTLSPlaintext packets.

```
struct {
    uint8  type;
    uint32 length;
    opaque channel_id[4];
    opaque data[MTLSPlaintext.length];
} MTLSPlaintext;

struct {
    uint8  type;
    uint16 length;
    opaque channel_id[4];
    uint32 max_packet_length;
    /* the max_packet_length of the sender of that packet */
} NewMaxPacketLength;

struct {
    uint8  type;
    uint16 length;
    opaque channel_id[4];
    uint32 free_space;
} Acknowledgment;
```

The Acknowledgment and NewMaxPacketLength packets can be sent even if no window space is available.

The (D)TLS Record Layer receives data from MTLS, supposes it as uninterpreted data and applies the fragmentation and the cryptographic operations on it, as defined in [RFC5246].

Note: multiple MTLS fragments MAY be coalesced into a single TLSPlaintext record.

Received data is decrypted, verified, decompressed, and reassembled, then delivered to MTLS layer. Next, the MTLS sends data to the appropriate application using the channel identifier and the length value.

2.3. MTLS over DTLS

To run MTLS over DTLS, we MUST provide reliability for all MTLS messages, except the MTLSPlaintext message that will be handled by DTLS record.

If the client is willing to run MTLS over DTLS, it MUST connect to the server that passively listens for the incoming DTLS connection on the IANA-to-be-assigned UDP port (TBA). The client MUST therefore send the TLS ClientHello to begin the DTLS handshake. Once the Handshake is complete, the client and the server cache the session ID and the master_secret.

Next, the client and the server start a TLS-PSK handshake [RFC4279]. The client only includes pre-shared key based cipher suites to the ClientHello message. The psk_identity is the session ID generated during the DTLS handshake and the psk is the master_secret. Using the cached session ID will help the server and the client to establish a local mapping between both TLS and DTLS sessions.

Once the TLS handshake is complete, both the client and the server can start multiplexing applications' channels using the set of requests/responses defined above. Excepting MTLSPlaintext, all requests/responses will be conveyed using TLS record.

MTLSPlaintext will be conveyed using DTLS record. The same Transport Layer Mapping defined by DTLS MUST be used here. In particular, the maximum record size. Hence, MTLSPlaintext MUST be smaller than the maximum record size - 9.

It is REQUIRED to support the cipher suite TLS_PSK_WITH_AES_128_CBC_SHA.

2.4. MTLS Message Types

This section defines the initial set of MTLS Message Types used in Request/Response exchanges. The Message Type field is one octet and identifies the structure of an MTLS Request or Response message.

The messages defined in this document are listed below. More Message Types may be defined in future documents. The list of Message Types, as defined through this document, is maintained by the Internet Assigned Numbers Authority (IANA). Thus, an application needs to be made to the IANA in order to obtain a new Message Type value. Since there are subtle (and not-so-subtle) interactions that may occur in this protocol between new features and existing features that may result in a significant reduction in overall security, new values SHALL be defined only through the IETF Review process specified in [RFC5226].

ChannelEstablishmentRequest	1
ChannelEstablishmentSuccess	2
ChannelRequestEchec	3
ChannelCloseRequest	4
ChannelCloseConfirmation	5
MTLSPlaintext	6
NewMaxPacketLength	7
Acknowledgment	8

3. Security Considerations

Security issues are discussed throughout this document and in [RFC5246].

If a fatal error related to any channel of an arbitrary application occurs, the secure session MUST NOT be resumed. This is logic since the Record protocol does not distinguish between the MTLS channels. However, if an error occurs at the MTLS layer, both parties immediately close the related channel, but not the (D)TLS session (no alert of any type is sent by the (D)TLS Record).

4. IANA Considerations

This section provides guidance to the IANA regarding registration of values related to the TLS protocol.

IANA is requested to assign a TCP and UDP port numbers that will be the default port for MTLS sessions as defined in this document. There is one name space in MTLS that requires registration: Message Types.

Message Types have a range from 1 to 255, of which 1-8 are to be allocated for this document. Because a new Message Type has considerable impact on interoperability, a new Message Type SHALL be defined only through the IETF Review process specified in [RFC5226].

5. Acknowledgments

The authors appreciate Alfred Hoenes for his detailed review.

6. Contributors

James Blaisdell (Mocana, USA)

7. References

7.1. Normative References

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC4279] Eronen, P. and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, December 2005.

[I-D.ietf-tls-rfc4347-bis] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security version 1.2", draft-ietf-tls-rfc4347-bis-02 (work in progress), March 2009.

7.2. Informative References

[RFC4254] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Connection Protocol", RFC 4254, January 2006.

Authors' Addresses

Mohamad Badra
CNRS/LIMOS Laboratory
Campus de cezeaux, Bat. ISIMA
Aubiere 63170
France

Email: badra@isima.fr

Ibrahim Hajjeh
INEOVATION
Paris
France

Email: ibrahim.hajjeh@ineovation.fr