TLS Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 27, 2008

M. Badra
LIMOS Laboratory
I. Hajjeh
ESRGroups

October 25, 2007

MTLS: TLS Multiplexing
<draft-badra-hajjeh-mtls-03.txt>

Status of this Memo

Copyright Notice

Abstract

The Transport Layer Security (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 application data over a single TLS session.

This document defines MTLS, a new TLS sub-protocol running over TLS (or DTLS) Record protocol. The MTLS design provides application multiplexing over a single TLS (or DTLS) session. Therefore, instead

of associating a TLS connection with each application, MTLS allows

several applications to protect their exchanges over a single TLS session.

1. Introduction

HTTP over TLS [HTTPTLS], POP over TLS and IMAP over TLS [POPTLS] are examples of securing, respectively HTTP, POP and IMAP data exchanges using the TLS protocol [TLS].

TLS ([TLS], [DTLS]) 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 TLS session.

Actually, TLS (or DTLS) clients and servers MUST establish a TLS (or DTLS) session for each application they want to run over a transport layer. 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 extends TLS to allow application multiplexing over TLS.

The document motivations included:

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

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

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

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

o 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 TLS session. However, the extension defined in this document allows, by its design, data protection against inference.

1.2. Requirements language

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 [KEYWORDS].

2. TLS multiplexing overview and considerations

This document defines a new TLS sub-protocol called Multiplexing TLS (MTLS) to handle data multiplexing, and it specifies the content type mtls(TBA). It extends also TLS with a new extension type (TBA) allowing the negotiation of data multiplexing features.

2.1. Handshake

This document defines an extension of type "data_multiplexing". The "extension_data" field of this extension is zero-length.

Based on the TLS Extensions [TLSEXT], a client and a server can, in an ordinary TLS handshake, negotiate the future use of MTLS. If the client does attempt to initiate a TLS connection using MTLS with a server that does not support it, it will be automatically alerted. For servers aware of MTLS but not wishing to use it, it will gracefully revert to an ordinary TLS handshake or stop the negotiation.

The negotiation usually starts with the client determining whether the server is capable of and willing to use MTLS or not. In order to allow a TLS client to negotiate the application multiplexing functionality, a new extension type SHOULD be added to the Extended Client and Extended Server Hello messages.

If the server is able of and willing to use the "data_multiplexing" extension, it MUST reply with an empty extension of the same type. Once the Handshake is complete, the client and the server SHOULD establish and manage many application channels using the requests/responses defined below.

2.1.1. Opening and closing connections

Once the Handshake is complete, both the client and the server can start data multiplexing using a set of requests/responses defined below. All requests/requests will pass through MTLS layer and are formatted into MTLS packets, depending on each request/response.

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


```
    struct {
           uint8 type;
           opaque sender_channel_id[2];
           uint32 sender_window_length;
           uint32 sender_max_packet_length;
           opaque source_address_machine<4..7>;
           opaque source_port[2];
           opaque destination_address_machine<4..7>;
           opaque destination_port[2];
        } RequestEstablishmentChannel;
```

The field "type" specifies the MTLS packet type (types are summarized below), the "max_packet_length" and the "sender_channel_id" are used as described below. 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 "port" is the port on the host from where the connection originated.

The sender initializes its "window_length" with the data length (in octets), specifying how many bytes the receiver 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".

The sender initializes its "max_packet_length" with the data length (in octets), specifying the maximal packet's length in octets the receiver can send on the channel.

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;
       opaque sender_channel_id[2];
       opaque receiver_channel_id[2];
       uint32 receiver_window_length;
       uint32 max_packet_length;
     } RequestEstablishmentSuccess;

struct {
       uint8 type;
       opaque sender_channel_id[2];
       opaque error<0..2^16>;
     } RequestEchecChannel;
```


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

If an error occurs at the MTLS layer, the established secure session is still valid and no alert of any type is sent by the TLS Record.

Each MTLS channel has its identifier computed as:

```
channel_id = sender_channel_id" + "receiver_channel_id
```

Where "+" indicates concatenation.

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;
```

```
            opaque channel_id[4];
        } CloseChannel;

      struct {
            uint8 type;
            opaque channel_id[4];
        } ConfirmationCloseChannel;
```

2.2. MTLS sub-protocol

The structure of the MTLS packet is described below. The "sender_channel_id" and "receiver_channel_id" are the same gererated during the connection establishment. The length conveys the data length of the current packet.

Each entity maintains its "max_packet_length" (that is originally initialized during the connection establishment) to a value not bigger than the maximum size of this entity's "receive buffer". For each received packet, the entity MUST subtract the packet's length from the "max_packet_length". The result is always positive since the packet's length is always less than or equal to the current "max_packet_length".

The free space of the "receive buffer" MAY increase in length. Consequently, the entity MUST inform the other end about this increase, allowing the other entity to send packet with length bigger than the old "max_packet_length" but smaller or equal than the new value.

The entity MAY indicate this increase by sending an Acknowledgment packet. The Acknowledgment packet carries the available free space ("free_space" field in octets) the receiver of that packet can send on the channel before receiving a new window length.

If the length of the "receive buffer" does not change, 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;
```

```
            opaque channel_id[4];
            uint32 length;
            opaque data[MTLSPlaintext.length];
          } MTLSPlaintext;

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

The 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 [TLS]. The type is set to mtls(TBA).

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

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

```
      enum {
          change_cipher_spec(20), alert(21), handshake(22),
          application_data(23), mtls(TBA), (255)
        } ContentType;
```

2.3. MTLS Message Types

Additional message types can be supported by MTLS.

```
                RequestEstablishmentChannel        0x01
                RequestEstablishmentSuccess        0x02
                RequestEchecChannel                0x03
                CloseChannel                       0x04
```



```
                ConfirmationCloseChannel           0x05
                MTLSPlaintext                      0x06
                Acknowledgment                     0x07
```

3. Security Considerations

Security issues are discussed throughout this document, and in

[TLS], [DTLS] and [TLSEXT] documents.

If a fatal error related to any channel or a connection 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 channels, but not the TLS session (no alert of any type is sent by the TLS Record).

4. IANA Considerations

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

There are name spaces that require registration: the mtls content type, the data_multiplexing extension, and the MTLS message types.

5. References

5.1. Normative References

[TLS] Dierks, T., Rescorla, E., "The TLS Protocol Version 1.1", RFC 4346, April 200P.

[TLSEXT] Blake-Wilson, S., et. al., "Transport Layer Security (TLS) Extensions", RFC 4346, April 2006.

[DTLS] Rescorla, E., Modadugu, N., "Datagram Transport Layer Security", RFC 4347, April 2006.

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

5.2. Informative References

[HTTPTLS] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[POPTLS] Newman, C., "Using TLS with IMAP, POP3 and ACAP", RFC 2595, June 1999.

[SSHCON] Lonvick, C., "SSH Connection Protocol", RFC 4254, January 2005.

Author's Addresses

Mohamad Badra
LIMOS Laboratory - UMR6158, CNRS
France   Email: badra@isima.fr

Ibrahim Hajjeh
ESRGroups, Security WG
France   Email: Ibrahim.Hajjeh@esrgroups.org

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).