Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 30, 2011

M. Badra
CNRS/LIMOS Laboratory
I. Hajjeh
INEOVATION
April 28, 2011

Multiplexing Single-Application Multiple-Connection over TLS
draft-badra-tls-multiplexing-01.txt

Abstract

The Transport Layer Security (TLS) is the most widely deployed protocol for securing network traffic. It provides mutual authentication, data confidentiality and integrity, key generation and distribution, and security parameters negotiation. However, missing from the protocol is a way to multiplex single-application multiple-stream applications that commonly use parallel connections to the same logical and/or physical server application data.

This document describes a mechanism to multiplex single-application multiple-stream over TLS. It extends TLS to multiplex parallel connections of a given application over a single TLS session, avoiding additional delay related to the TLS/TCP session/connection setup.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 30, 2011.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

A single-application multiple-stream/-thread/-connection commonly uses parallel connections to the same logical and/or physical server application data. When that application is run over TCP, a TCP connection must be established for each stream/thread/connection (channel). This incurs a significant additional delay due to the TCP slow-start and to the duplication of an existing TLS session as well. Having a single TCP connection and opening additional channels over that single TCP connection can benefit of a high TCP congestion window and throughput instantaneously via statistical multiplexing, and raising the throughput further (incrementally) from an already high level can be achieved much faster in TCP.

TLS does not multiplex single-application multiple-channel; it must instead duplicate the existing TLS session for each channel. This document defines Multiplexing Multi-Threaded Application over Transport Layer Security (MTA-TLS). MTA-TLS extends TLS to multiplex a single-application multiple-channel over a single TLS session, avoiding additional delay related to the TLS/TCP session/connection setup.

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. TLS multiplexing Overview and Considerations

This document makes use of TLS extensions described in [RFC5246] and defines an extension of type data_multiplexing(TBD). The "extension_data" field of this extension SHALL be empty (zero-length).

2.1. Handshake

The client and the server can, in an ordinary TLS handshake, negotiate the future use of MTA-TLS. The negotiation usually starts with the client that indicates its support of MTA-TLS by including an extension of type 'data_multiplexing' in its hello message.

If the client does attempt to initiate a TLS session using MTA-TLS with a server that does not support it, it will be automatically alerted. For servers aware of MTA-TLS but not wishing to use it, it will gracefully revert to an ordinary TLS handshake or stop the negotiation.

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, both the client and the server can start 'channel multiplexing' using a set of requests/responses defined below. All requests/responses will pass through MTA-TLS layer and are formatted into MTA-TLS packets, depending on each request/response.

2.1.1. Opening Channels

The sender MAY request the opening of many channels. For each channel, the MTA-TLS 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 MTA-TLS packet type (types are summarized below).

length

The "length" field indicates the length, in octets, of the current MTA-TLS 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 MTA-TLS 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 MTA-TLS 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. MTA-TLS Flow Control

The structure of the MTA-TLS 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 MTATLSPlaintext packets.

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

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 TLS Record Layer receives data from MTA-TLS, supposes it as uninterpreted data and applies the fragmentation and the cryptographic operations on it, as defined in [RFC5246].

Note: multiple MTA-TLS fragments MAY be coalesced into a single TLSPlaintext record.

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

2.3. MTA-TLS Message Types

This section defines the initial set of MTA-TLS Message Types used in Request/Response exchanges. The Message Type field is one octet and identifies the structure of an MTA-TLS 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
MTATLSPlaintext	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 MTA-TLS channels. However, if an error occurs at the MTA-TLS layer, both parties immediately close the related channel, 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 data_multiplexing extension and the MTA-TLS 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

TBC

7. Normative References

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

[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.

Authors' Addresses

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

Email: badra@isima.fr

Ibrahim Hajjeh
INEOVATION
Paris
France

Email: ibrahim.hajjeh@ineovation.fr

James Blaisdell
Mocana
San Francisco
USA

Email: JBlaisdell@mocana.com