MatrixDTLS Developer's Guide

inside
secure
DRIVING TRUST

TABLE OF CONTENTS

1 DIFFERENCES FROM MATRIXSSL

MatrixDTLS is very similar to the MatrixSSL product. The DTLS specification is simply an extension of TLS for use in applications that utilize a datagram transport mechanism such as UDP. DTLS and TLS are not compatible. The DTLS protocol differences allow the library to account for out-of-order message delivery and fragmentation of large records. This section highlights the differences between the MatrixDTLS product and MatrixSSL.

1.1 API

There are only two differences in the public API between the two products. In MatrixDTLS the function **matrixDtlsGetOutdata** is used instead of matrixSslGetOutdata and the function **matrixDtlsSentData** is used instead of matrixSslSentData. The prototypes for these functions are identical to their SSL counterparts. More information on the DTLS APIs can be found in the MatrixSSL API documentation.

1.2 Functionality

The only functional difference between the two protocols is that stream ciphers cannot be used in DTLS. ARC4 is the only stream cipher available in the Matrix libraries and is already disabled by default.

1.3 Application Responsibilities

The application has additional responsibilities in a DTLS implementation in comparison with SSL. Because the transport protocol will be unreliable, the application must manage a timeout mechanism that determines when to resend DTLS handshake messages. If the application has not received an expected response from the peer it is assumed that the peer did not receive the last sent message. The DTLS protocol specification offers some guidelines on this topic and is discussed in more detail in the Timeouts and Resends section below. In addition, the example dtlsServer and dtlsClient applications implement a retransmit timeout mechanism.

inside
secure
DRIVING TRUST

2 APPLICATION INTEGRATION TOPICS

2.1 Fragmentation and Maximum Transmission Unit (MTU)

Datagram transport protocols do not support the delivery or receipt of partial datagrams. The entire datagram is either delivered or lost. Therefore it is important that the maximum transmission unit (MTU) size is agreed upon between the two peers that will be communicating. The DTLS specification does not deal with how the two sides agree on the MTU.

A MTU value of 1500 is recommended as a starting point and should be set as the default by the DTLS_PMTU define in *matrixsslConfig.h*. MatrixDTLS does not support an MTU value smaller than 256. The functions matrixDtlsSetPmtu and matrixDtlsGetPmtu are the run-time mechanisms for setting and getting the global value.

If a single message is too large to fit within the maximum datagram it must be fragmented at the record level. The only SSL handshake messages that supports fragmentation in MatrixDTLS are the CERTIFICATE and CERTIFICATE_REQUEST messages. These are the only messages in which the length could reasonably be expected to exceed some maximum size (especially if certificate chaining is used).

Internal fragmentation is not supported on application data records. Application data lengths are constrained by the MTU setting and large records must be divided by the user and individually encoded. The call to matrixSslGetWritebuf will always return an output buffer that enables the encoding of a valid datagram length. So the user must make sure that if matrixSslGetWritebuf returns an available length that is less than the requested length that the plaintext is divided on that boundary.

2.2 Timeouts and Resends

Because the transport mechanism is unreliable, applications must implement a retransmission timer to account for lost datagrams. Section 4.2.4 of RFC 4347 explains the timeout and retransmission state machine in detail but the concept is simply to resend the previous flight of handshake messages if the expected reply from the peer does not arrive before the timer expires. The specification recommends that the timer be initialized to 1 second and doubled after each unresponsive resend to a maximum of 64 seconds. The client and server examples provided in the MatrixDTLS package implement these recommendations. Also see the discussion on the Handshake Endgame that follows for details on how to handle the transition from handshake records to application data records.

To rebuild the previous flight the user simply calls matrixDtlsGetOutdata after the timer expires.

NOTE: If handshake messages are being received individually in separate datagrams (rather than the entire flight of handshake messages being sent in a single datagram) the call to matrixDtlsGetOutdata will not rebuild the previous flight if the application has already parsed some individual handshake records from the expected flight. This is because the application can be certain the other side has received our previous flight since we have received a partial response to it. For example, if a server application has received the CLIENT_KEY_EXCHANGE message but has not received the expected CHANGE_CIPHER_SPEC message before the timeout, the server will not resend the SERVER_HELLO flight because it is certain that flight has already been seen. Rather, the server should continue to wait because the client will timeout and resend the CLIENT_KEY_EXCHANGE flight because it never received the expected FINISHED flight from the server.

By default, MatrixDTLS will attempt to send entire flights in a single datagram. The debug option DTLS_SEND_RECORDS_INDIVIDUALLY will force matrixDtlsGetOutdata to return single handshake messages if desired. See the **SSL Handshaking** section of the MatrixSSL Developer's Guide for details on the handshake messages that comprise the specific flights.

inside
secure
DRIVING TRUST

2.3 The Handshake Endgame

One topic that is not well covered by the DTLS specification is how a peer can be confident the handshake has successfully completed before transitioning to the sending of application data. If the FINISHED messages have not been received by both peers, the application data will not decrypt properly.

For the majority of connections this handshake endgame is not a practical concern because in a standard handshake the client is last to receive a handshake message but is also typically the first to send an application request. In this standard case the client can be confident the handshake is complete before sending data.

The problematic cases arise when the peer that is last to send handshake data is also the first to send application data. This scenario occurs during resumed handshakes when the client uses a previous session ID for fast connection. How can the client be certain the server has received the FINISHED message in the handshake protocol to be able to decrypt the application data that follows?

The only way to truly be certain that the other side has received the FINISHED message and is in a secure state is if the application receives an encrypted data record. So, **ideally the application protocol implementation should contain an initial data exchange to occur that is initiated by the peer that is last to receive a handshake message.**

In the absence of this protocol control, the application has a couple other options in this scenario;

1. The side that is last to send could wait a timeout period to see if the peer is resending its previous flight. If a resend is received, this is an indication that this application must resend the final handshake flight. The client example provided in the MatrixDTLS package uses this solution in the resumed handshake scenarios but this is not a good option in a production environment because the additional timeout will probably take much longer than then standard handshake protocol would.

2. Avoiding the DTLS handshake scenarios in which the endgame can be a problem is probably the best solution. In the standard case, this would mean not performing resumed handshakes (don't send a session id when creating a session).

Below is a table that shows where in the MatrixDTLS API a server and client should add logic to handle the handshake endgame problem.

	Standard Handshake	Resumed Handshake
Client	Last to receive handshake messages. Handshake is known to be complete when MATRIXSSL_HANDSHAKE_COMPLETE is returned from matrixSslReceivedData. Client can immediately send application data knowing the handshake is complete.	Last to send handshake messages. The final handshake flight is known to have been sent when MATRIXSSL_HANDSHAKE_COMPLETE is returned from matrixDtlsSentData but the client can't be sure the server received this message until an encrypted application data record is received.
Server	Last to send handshake messages. The final handshake flight is known to be sent when MATRIXSSL_HANDSHAKE_COMPLETE is returned from matrixDtlsSentData but the server can't be sure the client received this message until an encrypted application data record is received. Server should resend flight if timeout occurs.	Last to receive handshake messages. Handshake is known to be complete when MATRIXSSL_HANDSHAKE_COMPLETE is returned from matrixSslReceivedData. Server might choose to send application data to let the client know the connection is complete or may just choose to wait for application data record from client.

inside secure
DRIVING TRUST