                                                        E. Rescorla
                                                         RTFM, Inc.
                                                        N. Modadugu
INTERNET-DRAFT   Stanford University
<draft-rescorla-dtls-02.txt>   December 2003 (Expires June 2004)

                    Datagram Transport Layer Security

Status of this Memo

By submitting this Internet-Draft, I certify that any applicable
patent or other IPR claims of which I am aware have been disclosed,
and any of which I become aware will be disclosed, in accordance with
RFC 3668.

Internet-Drafts are working documents of the Internet Engineering
Task Force (IETF), its areas, and its working groups. Note that other
groups may also distribute working documents as Internet-Drafts.

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 a "work in progress."

The list of current Internet-Drafts can be accessed at
http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at
http://www.ietf.org/shadow.html

Copyright Notice

Contents

Abstract

   This document specifies Version 1.0 of the Datagram Transport Layer
   Security (DTLS) protocol. The DTLS protocol provides communications
   privacy for datagram protocols. The protocol allows client/server
   applications to communicate in a way that is designed to prevent
   eavesdropping, tampering, or message forgery. The DTLS protocol is
   based on the TLS protocol and provides equivalent security
   guarantees. Datagram semantics of the underlying transport are
   preserved by the DTLS protocol.

[1]. Introduction

   TLS [TLS] is the most widely deployed protocol for securing network
   traffic. It is widely used for protecting Web traffic and for e-mail
   protocols such as IMAP [IMAP] and POP [POP]. The primary advantage of
   TLS is that it provides a transparent channel. Thus, it is easy to
   secure an application protocol by inserting TLS between the
   application layer and the network layer. However, TLS must run over a
   reliable transport channel--typically TCP [TCP]. It therefore cannot
   be used to secure unreliable datagram traffic.

   However, over the past few years an increasing number of application
   layer protocols have been designed which UDP transport. In particular
   such protocols as the Session Initiation Protocol (SIP) [SIP], and
   electronic gaming protocols are increasingly popular. (Note that SIP
   can run over both TCP and UDP, but that there are situations in which
   UDP is preferable). Currently, designers these applications are faced
   with a number of unsatisfactory choices. First, they can use IPsec
   [RFC2401]. However, for a number of reasons detailed in [WHYIPSEC],
   this is only suitable for some applications. Second, they can design
   a custom application layer security protocol. SIP, for instance, uses
   a variant of S/MIME to secure its traffic. Unfortunately, application
   layer security protocols typically require a large amount of effort
   to design--by contrast to the relatively small amount of effort
   required to run the protocol over TLS.

   In many cases, the most desirable way to secure client/server
   applications would be to use TLS, however the requirement for
   datagram semantics automatically prohibits use of TLS. Thus, a
   datagram-compatible variant of TLS would be very desirable. This memo
   describes such a protocol: Datagram Transport Layer Security (DTLS).

DTLS is deliberately designed to be as similar to to TLS as possible,
both to minimize new security invention and to maximize the amount of
code and infrastructure reuse.

2. Usage Model

The DTLS protocol is designed to secure data between communicating
applications. It is designed to run in application space, without
requiring any kernel modifications. While the design of the DTLS
protocol does not preclude its use in securing arbitrary datagram
traffic, it is primarily expected to secure communication based on
datagram sockets.

Datagram transport does not guarantee reliable or in-order delivery
of data. The DTLS protocol preserves this property for payload data.
Applications such as media streaming, Internet telephony and online
gaming use datagram transport for communication due to the delay-
sensitive nature of transported data. The behavior of such
applications is unchanged when the DTLS protocol is used to secure
communication, since the DTLS protocol does not compensate for lost
or re-ordered data traffic.

3. Overview of DTLS

The basic design philosophy of DTLS is to construct "TLS over
datagram". The reason that TLS cannot be used directly in datagram
environments is simply that packets may be lost or reordered. TLS has
no internal facilities to handle this kind of unreliability and
therefore TLS implementations break when rehosted on datagram
transport. The purpose of DTLS is to make only the minimal changes to
TLS required to fix this problem. To the greatest extent possible,
DTLS is identical to TLS. Whenever we need to invent new mechanisms,
we attempt to do so in such a way that it preserves the style of TLS.

Unreliability creates problems for TLS at two levels:

   1. TLS's traffic encryption layer does not allow independent
   decryption of individual records. If record N is not received,
   then record N+1 cannot be decrypted.

   2. The TLS handshake layer assumes that handshake messages are
   delivered reliably and breaks if those messages are lost.

The rest of this section describes the approach that DTLS uses to
solve these problems.

3.1. Loss-insensitive messaging

   In TLS's traffic encryption layer (called the TLS Record Layer),
   records are not independent. There are two kinds of inter-record
   dependency:

      1. Cryptographic context (CBC state, stream cipher key stream) is
      chained between records.

      2. Anti-replay and message reordering protection are provided by a
      MAC which includes a sequence number, but the sequence numbers are
      implicit in the records.

   The fix for both of these problems is straightforward and well-known
   from IPsec ESP [ESP]: add explicit state to the records. TLS 1.1
   [TLS11] is already adding explicit CBC state to TLS records. DTLS
   borrows that mechanism and adds explicit sequence numbers.

3.2. Providing Reliability for Handshake

   The TLS handshake is a lockstep cryptographic handshake. Messages
   must be transmitted and received in a defined order and any other
   order is an error. Clearly, this is incompatible with reordering and
   message loss. In addition, TLS handshake messages are potentially
   larger than any given datagram, thus creating the problem of
   fragmentation. DTLS must provide fixes for both these problems.

3.2.1. Packet Loss

   DTLS uses a simple retransmission timer to handle packet loss. The
   following figure demonstrates the basic concept using the first phase
   of the DTLS handshake:

```
      Client                                    Server
      ------                                    ------
      ClientHello              ------>

                                     X<-- HelloVerifyRequest
                                                 (lost)

      [Timer Expires]

      ClientHello              ------>
      (retransmit)
```

   Once the client has transmitted the ClientHello message, it expects
   to see a HelloVerifyRequest from the server. However, if the server's
   message is lost the client knows that either the ClientHello or the

HelloVerifyRequest has been lost and retransmits. When the server
receives the retransmission, it knows to retransmit. The server also
maintains a retransmission timer and retransmits when that timer
expires.

3.2.2. Reordering

In DTLS, each handshake message is assigned a specific sequence
number within that handshake. When a peer receives a handshake
message, it can quickly determine whether that message is the next
message it expects. If it is, then it processes it. If not, it queues
it up for future handling once all previous messages have been
received.

3.3. Message Size

TLS and DTLS handshake messages can be quite large (in theory up to
$2^{24}-1$ bytes, in practice many kilobytes). By contrast, UDP datagrams
are often limited to <1500 bytes. In order to compensate for this
limitation, each DTLS handshake message may be fragmented over
several DTLS records. Each DTLS handshake message contains both a
fragment offset and a fragment length. Thus, a recipient in
possession of all bytes of a handshake message can reassemble the
original unfragmented message.
DTLS optionally supports record replay detection. The technique used
is the same as in IPsec AH/ESP, by maintaining a bitmap window of
received records. Records that are too old to fit in the window and
records that have been previously received are silently discarded.
The replay detection feature is optional, since packet duplication is
not always malicious, but can also occur due to routing errors.
Applications may conceivably detect duplicate packets and accordingly
modify their data transmission strategy.

4. Differences from TLS

As mentioned in Section , DTLS is intentionally very similar to TLS.
Therefore, instead of presenting DTLS as a new protocol, we instead
present it as a series of deltas from TLS 1.1 [TLS11]. Where we do
not explicitly call out differences, DTLS is the same as TLS.

4.1. Record Layer

The DTLS record layer is extremely similar to that of TLS 1.1. The
only change is the inclusion of an explicit sequence number in the
record. This sequence number allows the recipient to correctly verify
the TLS MAC. The DTLS record format is shown below:

```
   struct {
    ContentType type;
    ProtocolVersion version;
    uint16 epoch;
    uint48 sequence_number;
    uint16 length;
    opaque fragment[DTLSPlaintext.length];
   } DTLSPlaintext;
```

   type
    Equivalent to the type field in a TLS 1.1 record.

   version
    The version of the protocol being employed. This document
    describes DTLS Version 1.0, which uses the version { 254, 255
    }. The version value of 254.255 is the 1's complement of DTLS
    Version 1.0. This maximal spacing between TLS and DTLS version
    numbers ensures that records from the two protocols can be
    easily distinguished.

   epoch
    A counter value that is incremented on every cipher state
    change.

   sequence_number
    The sequence number for this record.

   length
    Identical to the length field in a TLS 1.1 record. As in TLS
    1.1, the length should not exceed 2^{14}.

   fragment
    Identical to the fragment field of a TLS 1.1 record.

DTLS uses an explicit rather than implicit sequence number, carried
in the sequence_number field of the record. As with TLS, the sequence
number is set to zero after each ChangeCipherSpec message is sent.

If several handshakes are performed in close succession, there might
be multiple records on the wire with the same sequence number but
from different cipher states. The epoch field allows recipients to
distinguish such packets. The epoch number is initially zero and is
incremented each time the ChangeCipherSpec messages is sent. In order
to ensure that any given sequence/epoch pair is unique,
implementations MUST NOT allow the same epoch value to be reused
within two times the maximum segment lifetime. In practice, TLS
implementations rehandshake rarely and we therefore do not expect
this to be a problem.

4.1.1. Transport Layer Mapping

   Each DTLS record MUST fit within a single datagram. In order to avoid
   IP fragmentation [MOGUL], DTLS implementations SHOULD determine the
   MTU and send records smaller than the MTU. DTLS implementations
   SHOULD provide a way for applications to determine the value of the
   MTU (optimally the maximum application datagram size, which is the
   PMTU minus the DTLS per-record overhead). If the application attempts
   to send a record larger than the MTU, the DTLS implementation MUST
   either generate an error or fragment the packet.

4.1.1.1. **PMTU Discovery**

   The PMTU SHOULD be initialized from the interface MTU that will be
   used to send packets.

   To perform PMTU discovery, the DTLS sender sets the IP Don't Fragment
   (DF) bit. As specified in [RFC 1191], when a router receives a packet
   with DF set that is larger than the next link's MTU, it sends an ICMP
   Destination Unreachable message to the source of the datagram with
   the Code indicating "fragmentation needed and DF set" (also known as
   a "Datagram Too Big" message). When a DTLS implementation receives a
   Datagram Too Big message, it decreases its PMTU to the Next-Hop MTU
   value given in the ICMP message. If the MTU given in the message is
   zero, the sender chooses a value for PMTU using the algorithm
   described in Section 7 of [RFC 1191]. If the MTU given in the message
   is greater than the current PMTU, the Datagram Too Big message is
   ignored, as described in [RFC 1191].

   A DTLS implementation may allow the application to occasionally
   request that PMTU discovery be performed again. This will reset the
   PMTU to the outgoing interface's MTU. Such requests SHOULD be rate
   limited, to one per two seconds, for example.

   Because some firewalls and routers screen out ICMP messages, it is
   difficult to distinguish packet loss from a large PMTU estimate. In
   order to allow connections under these circumstances, DTLS
   implementations MAY choose to back off their PMTU estimate during the
   retransmit backoff described in Section . For instance, if a large
   packet is being sent, after 3 retransmits a sender might choose to
   fragment the packet.

4.1.2. **Record payload protection**

4.1.2.1. **MAC**

   The DTLS MAC is the same as that of TLS 1.1. However, rather than
   using TLS's implicit sequence number, the sequence number used to

compute the MAC is the 64-bit value formed by concatenating the epoch
and the sequence number in the order they appear on the wire. Note
that the DTLS epoch + sequence number is the same length as the TLS
sequence number.

4.1.2.2. Null or standard stream cipher

The DTLS NULL cipher is performed exactly as the TLS 1.1 NULL cipher.

The only stream cipher described in TLS 1.1 is RC4, which cannot be
randomly accessed. RC4 MUST NOT be used with DTLS.

4.1.2.3. Block Cipher

DTLS block cipher encryption and decryption are performed exactly as
with TLS 1.1.

4.1.2.4. Anti-Replay

DTLS records contain a sequence number to provide replay protection.
Sequence number verification SHOULD be performed using the following
sliding, window procedure, borrowed from Section 3.4.3 of [RFC 2402]

The receiver packet counter for this session MUST be initialized to
zero when the session is established. For each received record, the
receiver MUST verify that the record contains a Sequence Number that
does not duplicate the Sequence Number of any other record received
during the life of this session. This SHOULD be the first check
applied to a packet after it has been matched to a session, to speed
rejection of duplicate records.

Duplicates are rejected through the use of a sliding receive window.
(How the window is implemented is a local matter, but the following
text describes the functionality that the implementation must
exhibit.) A MINIMUM window size of 32 MUST be supported; but a window
size of 64 is preferred and SHOULD be employed as the default.
Another window size (larger than the MINIMUM) MAY be chosen by the
receiver. (The receiver does NOT notify the sender of the window
size.)

The "right" edge of the window represents the highest, validated
Sequence Number value received on this session. Records that contain
Sequence Numbers lower than the "left" edge of the window are
rejected. Packets falling within the window are checked against a
list of received packets within the window. An efficient means for
performing this check, based on the use of a bit mask, is described
in [RFC 2401].

If the received record falls within the window and is new, or if the packet is to the right of the window, then the receiver proceeds to MAC verification. If the MAC validation fails, the receiver MUST discard the received record as invalid. The receive window is updated only if the MAC verification succeeds.

4.2. The DTLS Handshake Protocol

DTLS uses all of the same handshake messages and flows as TLS, with three principal changes:

   1. A stateless cookie exchange to prevent denial of service attacks.

   2. Modifications to the handshake header to handle message loss, reordering and fragmentation.

   3. Retransmission timers to handle message loss.

With these exceptions, the DTLS message formats, flows, and logic are the same as those of TLS 1.1.

4.2.1. Denial of Service Countermeasures

Datagram security protocols are extremely susceptible to a variety of denial of service (DoS) attacks. Two attacks are of particular concern:

   1. An attacker can consume excessive resources on the server by transmitting a series of handshake initiation requests, causing the server to allocate state and potentially perform expensive cryptographic operations.

   2. An attacker can use the server as an amplifier by sending connection initiation messages with a forged source of the victim. The server then sends its next message (in DTLS, a Certificate message, which can be quite large) to the victim machine, thus flooding it.

In order to prevent both of these attacks, DTLS borrows the stateless cookie technique used by Photuris [PHOTURIS] and IKEv2 [IKE]. When the client sends its ClientHello message to the server, the server MAY respond with a HelloVerifyRequest message. This message contains a stateless cookie generated using the technique of [PHOTURIS]. The client MUST retransmit the ClientHello with the cookie added. The server then verifies the cookie and proceeds with the handshake only if it is valid.

The exchange is shown below:

```
Client                                      Server
------                                      ------
ClientHello             ------>

                        <----- HelloVerifyRequest
                                 (contains cookie)

ClientHello             ------>
(with cookie)

[Rest of handshake]
```

DTLS therefore modifies the ClientHello message to add the cookie value.

```
struct {
  ProtocolVersion client_version;
  Random random;
  SessionID session_id;
  Cookie cookie<0..32>;                  // New field
  CipherSuite cipher_suites<2..2^16-1>;
  CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

If the client does not have a cookie for a given server, it should use a zero-length cookie.

The definition of HelloVerifyRequest is as follows:

```
struct {
  Cookie cookie<0..32>;
} HelloVerifyRequest;
```

The HelloVerifyRequest message type is hello_verify_request(3).

When responding to a HelloVerifyRequest the client MUST use the same parameter values (version, random, session_id, cipher_suites, compression_method) as in the original ClientHello. The server SHOULD use those values to generate its cookie and verify that they are correct upon cookie receipt.

Although DTLS servers are not required to do a cookie exchange, they SHOULD do so whenever a new handshake is performed in order to avoid being used as amplifiers. If the server is being operated in an environment where amplification is not a problem, the server MAY choose not to perform a cookie exchange. In addition, the server MAY

choose not do to a cookie exchange when a session is resumed. Clients
MUST be prepared to do a cookie exchange with every handshake.

4.2.2. Handshake Message Format

In order to support message loss, reordering, and fragmentation DTLS
modifies the TLS 1.1 handshake header:

```
   struct {
     HandshakeType msg_type;
     uint24 length;
     uint16 message_seq;                              // New field
     uint24 fragment_offset;                          // New field
     uint24 fragment_length;                          // New field
     select (HandshakeType) {
   case hello_request: HelloRequest;
   case client_hello:  ClientHello;
   case hello_verify_request: HelloVerifyRequest;     // New message type
   case server_hello:  ServerHello;
   case certificate:Certificate;
   case server_key_exchange: ServerKeyExchange;
   case certificate_request: CertificateRequest;
   case server_hello_done:ServerHelloDone;
   case certificate_verify:  CertificateVerify;
   case client_key_exchange: ClientKeyExchange;
   case finished:Finished;
     } body;
   } Handshake;
```

The first message each side transmits in each handshake always has
message_seq = 0. Whenever each new message is generated, the
message_seq value is incremented by one. When a message is
retransmitted, the same message_seq value is used. For example.

```
   Client                            Server
   ------                            ------
   ClientHello (seq=0)  ------>

                           X<-- HelloVerifyRequest (seq=0)
                                       (lost)


   [Timer Expires]

   ClientHello (seq=0)  ------>
   (retransmit)

                         <------ HelloVerifyRequest (seq=0)
```

```
    ClientHello (seq=1)  ------>
    (with cookie)

                        <------            ServerHello (seq=1)
                        <------            Certificate (seq=2)
                        <------       ServerHelloDone (seq=3)


    [Rest of handshake]
```

DTLS implementations maintain (at least notionally) a
next_receive_seq counter. This counter is initially set to zero. When
a message is received, if its sequence number matches
next_receive_seq, next_receive_seq is incremented and the message is
processed. If the sequence number is less than next_receive_seq the
message MUST be discarded. If the sequence number is greater than
next_receive_seq, the implementation SHOULD queue the message but MAY
discard it. (This is a simple space/bandwidth tradeoff).

4.2.3. Message Fragmentation and Reassembly

As noted in Section , each DTLS message MUST fit within a single
transport layer datagram. However, handshake messages are potentially
bigger than the maximum record size. Therefore DTLS provides a
mechanism for fragmenting a handshake message over a number of
records.

When transmitting the handshake message, the sender divides the
message into a series of N contiguous data ranges. These range must
be no larger than the maximum handshake fragment size and MUST
jointly contain the entire handshake message. The ranges SHOULD NOT
overlap. The sender then creates N handshake messages, all with the
same message_seq value as the original handshake message. Each new
message is labelled with the fragment_offset (the number of bytes
contained in previous fragments) and the fragment_length (the length
of this fragment). The length field in all messages is the same as
the length field of the original message. An unfragmented message is
a degenerate case with fragment_offset=0 and fragment_length=length.

When a DTLS implementation receives a handshake message fragment, it
MUST buffer it until it has the entire handshake message. DTLS
implementations MUST be able to handle overlapping fragment ranges.
This allows senders to retransmit handshake messages with smaller
fragment sizes during path MTU discovery.

4.2.4. Timeout and Retransmission

DTLS messages are grouped into a series of message flights, according
the diagrams below. Although each flight of messages may consist of a

number of messages, they should be viewed as monolithic for the
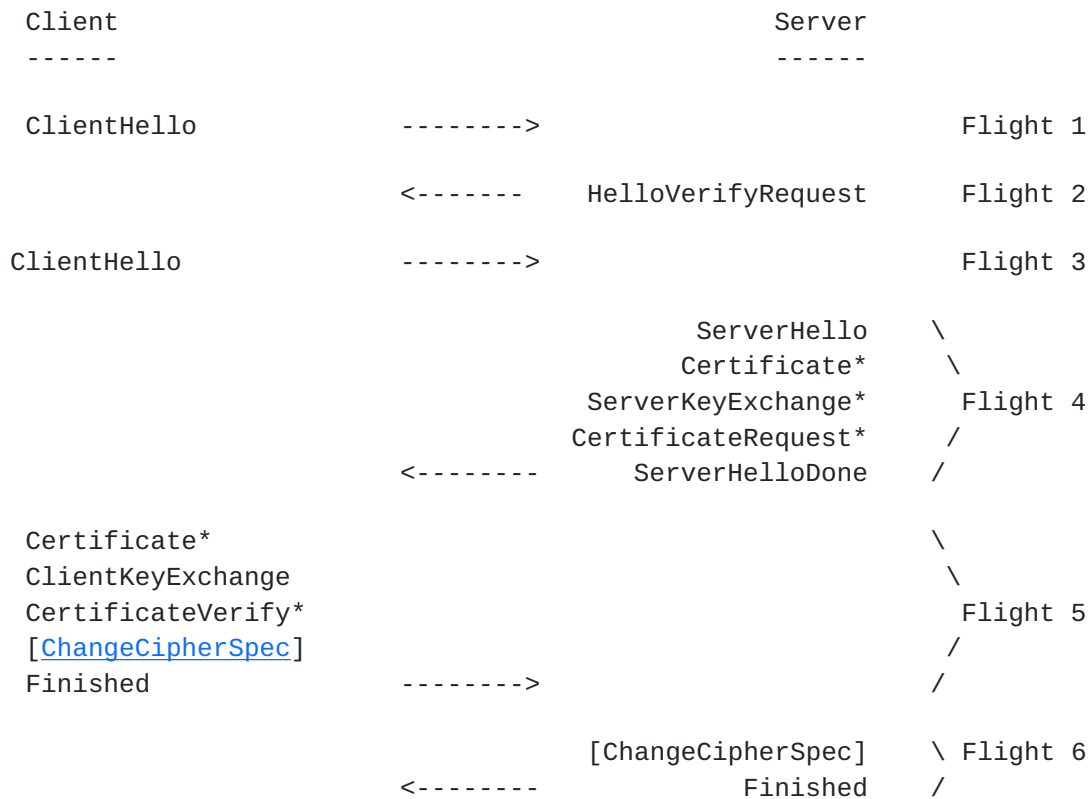purpose of timeout and retransmission.

```
   Client                                           Server
   ------                                           ------

   ClientHello            -------->                           Flight 1

                          <-------    HelloVerifyRequest      Flight 2

   ClientHello            -------->                           Flight 3

                                          ServerHello    \
                                         Certificate*     \
                                   ServerKeyExchange*      Flight 4
                                   CertificateRequest*     /
                          <--------      ServerHelloDone   /

   Certificate*                                           \
   ClientKeyExchange                                       \
   CertificateVerify*                                       Flight 5
   [ChangeCipherSpec]                                      /
   Finished               -------->                       /

                                       [ChangeCipherSpec]   \ Flight 6
                          <--------              Finished    /
           Figure 1: Message flights for full handshake

   Client                                           Server
   ------                                           ------

   ClientHello            -------->                           Flight 1

                                          ServerHello    \
                                    [ChangeCipherSpec]     Flight 2
                          <--------              Finished   /

   [ChangeCipherSpec]                                      \Flight 3
   Finished               -------->                       /
   Figure 2: Message flights for session resuming handshake (no cookie
exchange)
```
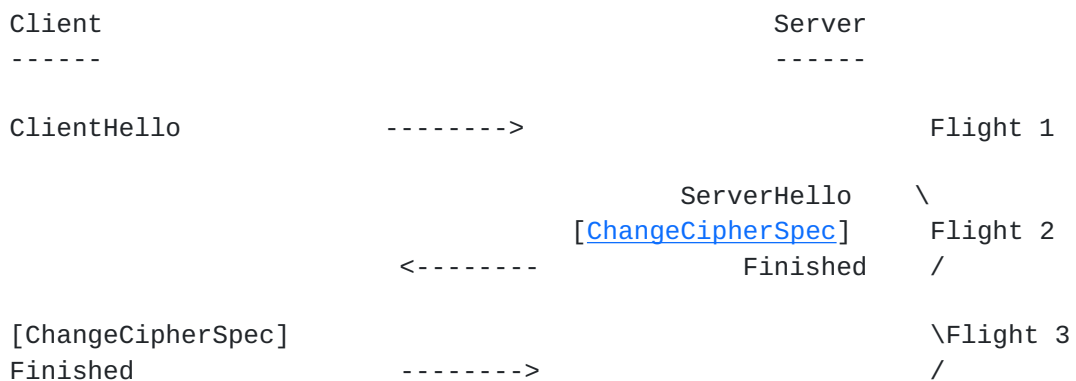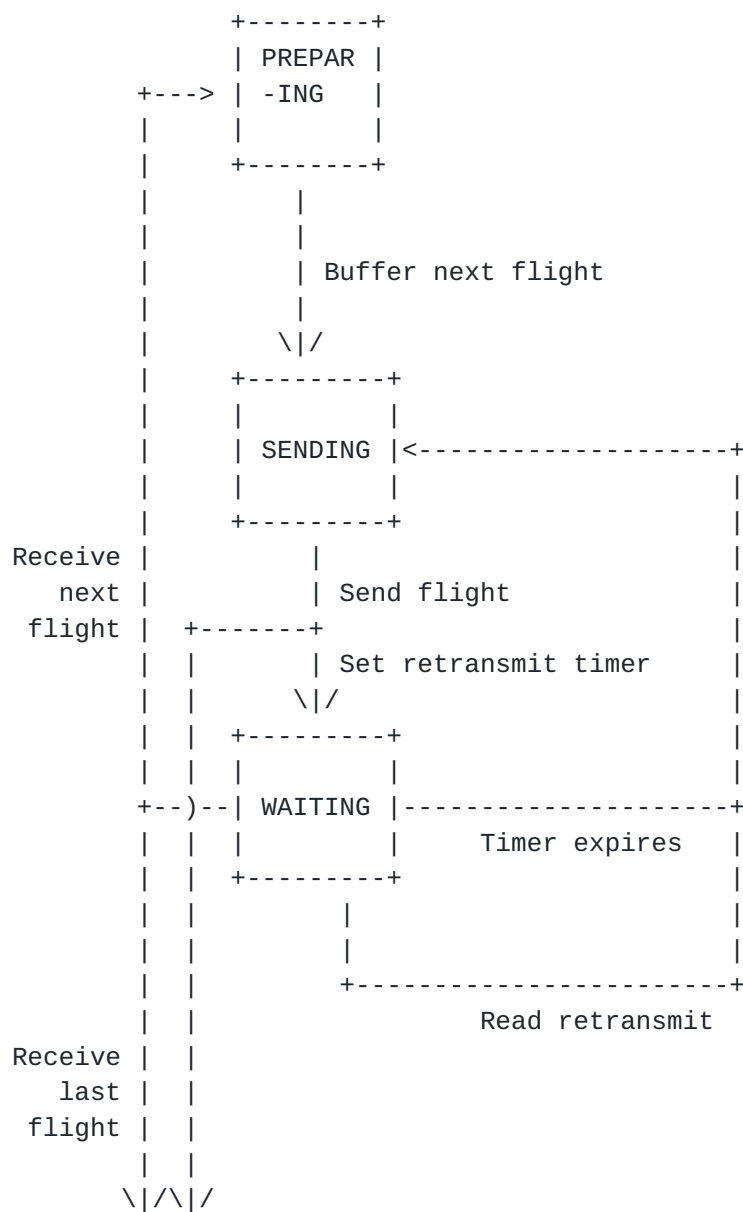
   DTLS uses a simple timeout and retransmission scheme with the
   following state machine.

```
                    +--------+
                    | PREPAR |
          +--->  |  -ING    |
          |       |          |
          |       +--------+
          |            |
          |            |
          |            | Buffer next flight
          |            |
          |           \|/
          |       +---------+
          |       |         |
          |       | SENDING |<--------------------+
          |       |         |                     |
          |       +---------+                     |
  Receive |            |                          |
     next |            | Send flight              |
   flight |   +-------+                           |
          |   |        | Set retransmit timer     |
          |   |       \|/                         |
          |   |   +---------+                     |
          |   |   |         |                     |
      +--)--| WAITING |---------------------+
          |   |   |         |     Timer expires   |
          |   |   +---------+                     |
          |   |        |                          |
          |   |        |                          |
          |   |        +------------------------+
          |   |                 Read retransmit
  Receive |   |
     last |   |
   flight |   |
          |   |
        \|/\|/

          FINISH
         Figure 3: DTLS timeout and retransmission state machine
```

The state machine has three basic states.

In the PREPARING state the implementation does whatever computations
are necessary to prepare the next flight of messages. It then buffers
them up for transmission (emptying the buffer first) and enters the
SENDING state.

In the SENDING state, the implementation transmits the buffered
flight of messages. Once the messages have been sent, the
implementation then enters the FINISH state if this is the last
flight in the handshake, or, if the implementation expects to receive
more messages, sets a retransmit timer and then enters the WAITING
state.

There are three ways to exit the WAITING state:

   1. The retransmit timer expires: the implementation transitions to
   the SENDING state, where it retransmits the flight, resets the
   retransmit timer, and returns to the WAITING state.

   2. The implementation reads a retransmitted flight from the peer:
   the implementation transitions to the SENDING state, where it
   retransmits the flight, resets the retransmit timer, and returns
   to the WAITING state. The rationale here is that the receipt of a
   duplicate message is the likely result of timer expiry on the peer
   and therefore suggests that part of one's previous flight was
   lost.

   3. The implementation receives the next flight of messages: if
   this is the final flight of messages the implementation
   transitions to FINISHED. If the implementation needs to send a new
   flight, it transitions to the PREPARING state. Partial reads
   (whether partial messages or only some of the messages in the
   flight) do not cause state transitions or timer resets.

Because DTLS clients send the first message (ClientHello) they start
in the PREPARING state. DTLS servers start in the WAITING state, but
with empty buffers and no retransmit timer.

4.2.4.1. Timer Values

Timer value choices are a local matter. We recommend that
implementations use an initial timer value of 500 ms and double the
value at each retransmission, up to 2MSL. Implementations SHOULD
start the timer value at the initial value with each new flight of
messages.

4.2.5. ChangeCipherSpec

As with TLS, the ChangeCipherSpec message is not technically a
handshake message but MUST be treated as part of the same flight as
the associated Finished message for the purposes of timeout and
retransmission.

4.2.6. Finished messages

   Finished messages have the same format as in TLS. However, in order
   to remove sensitivity to fragmentation, the Finished MAC MUST be
   computed as if each handshake message had been sent as a single
   fragment. Note that in cases where the cookie exchange is used, the
   initial ClientHello and HelloVerifyRequest ARE included in the
   Finished MAC.

A.1 Summary of new syntax

   This section includes specifications for the data structures that
   have changed between TLS 1.1 and DTLS.

4.2. Record Layer
   struct {
      ContentType type;
      ProtocolVersion version;
      uint16 epoch;   // NEW
      uint48 sequence_number;   // NEW
      uint16 length;
      opaque fragment[DTLSPlaintext.length];
   } DTLSPlaintext;

   struct {
      ContentType type;
      ProtocolVersion version;
      uint16 epoch;   // NEW
      uint48 sequence_number;   // NEW
      uint16 length;
      opaque fragment[DTLSCompressed.length];
   } DTLSCompressed;

   struct {
      ContentType type;
      ProtocolVersion version;
      uint16 epoch;   // NEW
      uint48 sequence_number;   // NEW
      uint16 length;
      select (CipherSpec.cipher_type) {
   case block: GenericBlockCipher;
      } fragment;
   } DTLSCiphertext;

4.3. Handshake Protocol

```
enum {
  hello_request(0), client_hello(1), server_hello(2),
  hello_verify_request(3),                        // NEW
  certificate(11), server_key_exchange (12),
  certificate_request(13), server_hello_done(14),
  certificate_verify(15), client_key_exchange(16),
  finished(20), (255)
} HandshakeType;

struct {
  HandshakeType msg_type;
  uint24 length;
  uint16 message_seq;                             // NEW
  uint24 fragment_offset;                         // NEW
  uint24 fragment_length;                         // NEW
  select (HandshakeType) {
case hello_request: HelloRequest;
case client_hello:  ClientHello;
case server_hello:  ServerHello;
case hello_verify_request: HelloVerifyRequest;    // NEW
case certificate:Certificate;
case server_key_exchange: ServerKeyExchange;
case certificate_request: CertificateRequest;
case server_hello_done:ServerHelloDone;
case certificate_verify:  CertificateVerify;
case client_key_exchange: ClientKeyExchange;
case finished:Finished;
  } body;
} Handshake;

struct {
  Cookie cookie<H0..32>;
} HelloVerifyRequest;
```

5. Security Considerations

This document describes a variant of TLS 1.1 and therefore most of
the security considerations are the same as TLS 1.1.

The primary additional security consideration raised by DTLS is that
of denial of service. DTLS includes a cookie exchange designed to
protect against denial of service. However, implementations which do
not use this cookie exchange are still vulnerable to DoS. In
particular, DTLS servers which do not use the cookie exchange may be
used as attack amplifiers even if they themselves are not
experiencing DoS. Therefore DTLS servers SHOULD use the cookie

exchange unless there is good reason to believe that amplification is
not a threat in their environment.

6. IANA Considerations

This document uses the same identifier space as does TLS [TLS11], so
no IANA registries are required beyond those for TLS. Identifiers MAY
NOT be assigned for DTLS that conflict with TLS.

References

Normative References

   [PHOTURIS] Karn, P., Simpson, W., "Photuris: Session-Key Management
               Protocol", RFC 2521, March 1999.

   [RFC1191]   Mogul, J. C., Deering, S.E., "Path MTU Discovery",
               RFC 1191, November 1990.

   [RFC2401]   Kent, S., Atkinson, R., "Security Architecture for the
               Internet Protocol", RFC2401, November 1998.

   [TLS]   Dierks, T., and Allen, C., "The TLS Protocol Version 1.0",
               RFC 2246, January 1999.

   [TLS11]   Dierks, T., Rescorla, E., "The TLS Protocol Version 1.1",
               draft-ietf-tls-rfc2246-bis-05.txt, July 2003.

Informative References

   [AH]   Kent, S., and Atkinson, R., "IP Authentication Header",
               RFC 2402, November 1998.

   [DCCP]   Kohler, E., Handley, M., Floyd, S., Padhye, J., "Datagram
               Congestion Control Protocol", draft-ietf-dccp-spec-05.txt,
               October 2003

   [DTLS]   Modadugu, N., Rescorla, E., "The Design and Implementation
               of Datagram TLS", in Proceedings of ISOC NDSS 2004,
               February 2004.

   [ESP]   Kent, S., and Atkinson, R., "IP Encapsulating Security
               Payload (ESP)", RFC 2406, November 1998.

   [IKE]   Harkins, D., Carrel, D., "The Internet Key Exchange (IKE)",
               RFC 2409, November 1998.

   [IMAP]   Crispin, M., "Internet Message Access Protocol - Version
               4rev1", RFC 3501, March 2003.

   [POP]   Myers, J., and Rose, M., "Post Office Protocol -
               Version 3", RFC 1939, May 1996.

   [SIP]   Rosenberg, J., Schulzrinne, Camarillo, G., Johnston, A.,
               Peterson, J., Sparks, R., Handley, M., Schooler, E.,
               "SIP: Session Initiation Protocol", RFC 3261,
               June 2002.

   [TCP]   Postel, J., "Transmission Control Protocol",
               RFC 793, September 1981.

   [WHYIPSEC] Bellovin, S., "Guidelines for Mandating the Use of IPsec",
               draft-bellovin-useipsec-02.txt, October 2003

Authors' Address

   Eric Rescorla <ekr@rtfm.com>
   RTFM, Inc.
   2064 Edgewood Drive
   Palo Alto, CA 94303

   Nagendra Modadugu <nagendra@cs.stanford.edu>
   Computer Science Department
   353 Serra Mall
   Stanford University
   Stanford, CA 94305

Full Copyright Statement