                 Datagram Transport Layer Security version 1.2
                      draft-ietf-tls-rfc4347-bis-05.txt

Abstract

   This document specifies Version 1.2 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 Transport Layer Security (TLS) protocol and provides
   equivalent security guarantees. Datagram semantics of the underlying
   transport are preserved by the DTLS protocol. This document
   updates DTLS 1.0 to work with TLS version 1.2.

   This Internet-Draft will expire on September 14, 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.

   This document may contain material from IETF Documents or IETF
   Contributions published or made publicly available before November
   10, 2008. The person(s) controlling the copyright in some of this
   material may not have granted the IETF Trust the right to allow
   modifications of such material outside the IETF Standards Process.
   Without obtaining an adequate license from the person(s) controlling
   the copyright in such materials, this document may not be modified
   outside the IETF Standards Process, and derivative works of it may
   not be created outside the IETF Standards Process, except to format
   it for publication as an RFC or to translate it into languages other
   than English.

Table of Contents

[1](1). Introduction

   TLS [[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](POP)]. The primary advantage
   of TLS is that it provides a transparent connection-oriented channel.
   Thus, it is easy to secure an application protocol by inserting TLS
   between the application layer and the transport layer. However, TLS
   must run over a reliable transport channel -- typically TCP [[TCP](TCP)].
   It therefore cannot be used to secure unreliable datagram traffic.

However, an increasing number of application layer protocols have
been designed that use UDP transport. In particular protocols such
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 of these applications are faced
with a number of unsatisfactory choices. First, they can use IPsec
[RFC4301]. 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. Unfortunately,
although application layer security protocols generally provide

superior security properties (e.g., end-to-end security in the case
of S/MIME), they typically require a large amount of effort to design
-- in 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. This memo
describes a protocol for this purpose: Datagram Transport Layer
Security (DTLS). DTLS is deliberately designed to be as similar to
TLS as possible, both to minimize new security invention and to
maximize the amount of code and infrastructure reuse.

DTLS 1.0 [DTLS1] was originally defined as a delta from [TLS11]. This
document introduces a new version of DTLS, DTLS 1.2, which is defined
as a series of deltas to TLS 1.2 [TLS12] There is no DTLS 1.1. That
version number was skipped in order to harmonize version numbers with
TLS. This version also clarifies some confusing points in the DTLS
1.0 specification.

Implementations that speak both DTLS 1.2 and DTLS 1.0 can
interoperate with those that speak only DTLS 1.0 (using DTLS 1.0 of
course), just as TLS 1.2 implementations can interoperate with
previous versions (See Appendix E.1 of [TLS12] for details) with the
exception that there is no DTLS version of SSLv2 or SSLv3 so that the
backward compatibility issues for those protocols do not apply.

1.1. Requirements Terminology

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 RFC 2119 [REQ].

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.

Datagram transport does not require or provide 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 transport." 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 preserves the style of TLS.

Unreliability creates problems for TLS at two levels:

   1. TLS does not allow independent decryption of individual records. Because the integrity check depends on the sequence number, if record N is not received, then the integrity check on record N+1 will be based on the wrong sequence number and thus will fail. [Note that prior to TLS 1.1, there was no explicit IV and so decryption would also fail.]

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 (stream cipher key stream) is retained
   between records.

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

DTLS solves the first problem by banning stream ciphers. DTLS solves
the second problem by adding 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 IP
fragmentation. DTLS must provide fixes for both of 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.

   Note: timeout and retransmission do not apply to the
   HelloVerifyRequest, because this requires creating state on the
   server.

   Note: The HelloVerifyRequest is designed to be small enough that it
   will not itself be fragmented, thus avoiding concerns about
   interleaving multiple HelloVerifyRequests.

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.

3.2.3. 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 if IP fragmentation is not
   desired. In order to compensate for this limitation, each DTLS
   handshake message may be fragmented over several DTLS records, each
   of which is intended to fit in a single IP datagram. 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.

3.3. Replay Detection

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 previously been 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 3, DTLS is intentionally very similar to TLS.
Therefore, instead of presenting DTLS as a new protocol, we present
it as a series of deltas from TLS 1.2 [TLS12]. Where we do not
explicitly call out differences, DTLS is the same as in [TLS12].

4.1. Record Layer

The DTLS record layer is extremely similar to that of TLS 1.2. 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;                                    // New field
    uint48 sequence_number;                          // New field
    uint16 length;
    opaque fragment[DTLSPlaintext.length];
  } DTLSPlaintext;

  type
   Equivalent to the type field in a TLS 1.2 record.
```

```
    version
```

The version of the protocol being employed. This document
describes DTLS Version 1.2, which uses the version { 254, 253
}. The version value of 254.253 is the 1's complement of DTLS
Version 1.2. This maximal spacing between TLS and DTLS version
numbers ensures that records from the two protocols can be
easily distinguished. It should be noted that future on-the-wire
version numbers of DTLS are decreasing in value (while the true
version number is increasing in value.)

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 1.2 record. As in TLS
 1.2, the length should not exceed 2^{14}.

fragment
 Identical to the fragment field of a TLS 1.2 record.

DTLS uses an explicit sequence number, rather than an implicit one,
carried in the sequence_number field of the record. Sequence numbers
are maintained separately for each epoch, with each sequence_number
initially being 0 for each epoch. For instance, if a handshake
message from epoch 0 is retransmitted, it might have a sequence
number after a message from epoch 1, even if the message from epoch 1
was transmitted first. Note that some care needs to be taken during
the handshake to ensure that retransmitted messages use the right
epoch and keying material.

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 TCP maximum segment lifetime. In practice, TLS
implementations rarely rehandshake and we therefore do not expect
this to be a problem.

Note that because DTLS records may be reordered, a record from epoch
1 may be received after epoch 2 has begun. In general,
implementations SHOULD discard packets from earlier epochs, but if

packet loss causes noticeable problems MAY choose to retain keying
material from previous epochs for up to 120 seconds (the default TCP
MSL) to allow for packet reordering. Until the handshake has
completed, implementations MUST accept packets from the old epoch.

Conversely, it is possible for records that are protected by the
newly negotiated context to be received prior to the completion of a
handshake. For instance, the server may send its Finished and then
start transmitting data. Implementations MAY either buffer or
discard such packets, though when DTLS is used over reliable
transports (e.g., SCTP), they SHOULD be buffered and processed once
the handshake completes. Note that TLS's restrictions on when
packets may be sent still apply, and the receiver treats the packets
as if they were sent in the right order. In particular, it is still
impermissible to send data prior to completion of the first
handshake.

Note that in the special case of a rehandshake on an existing
association, it is safe to process a data packet immediately even if
the ChangeCipherSpec or Finished has not yet been received provided
that either the rehandshake resumes the existing session or that it
uses exactly the same security parameters as the existing
association. In any other case, the implementation MUST wait for the
receipt of the Finished to prevent downgrade attack.

As in TLS, implementations MUST either abandon an association or
rehandshake prior to allowing the sequence number to wrap. Similarly,
implementations MUST NOT allow the epoch to wrap, but instead MUST
establish a new association, terminating the old association as
described in Section 4.2.8. In practice, implementations rarely
rehandshake repeatedly on the same channel, so this is not likely to
be an issue.

4.1.1. Transport Layer Mapping

Each DTLS record MUST fit within a single datagram. In order to
avoid IP fragmentation, clients of the DTLS record layer SHOULD
attempt to size records so that they fit within any PMTU estimates
obtained from the record layer.

Note that unlike IPsec, DTLS records do not contain any association
identifiers. Applications must arrange to multiplex between
associations. With UDP, this is presumably done with host/port
number.

Multiple DTLS records may be placed in a single datagram. They are
simply encoded consecutively. The DTLS record framing is sufficient

to determine the boundaries. Note, however, that the first byte of
the datagram payload must be the beginning of a record. Records may
not span datagrams.

Some transports, such as DCCP [DCCP] provide their own sequence
numbers. When carried over those transports, both the DTLS and the
transport sequence numbers will be present. Although this introduces
a small amount of inefficiency, the transport layer and DTLS sequence
numbers serve different purposes, and therefore for conceptual
simplicity it is superior to use both sequence numbers. In the
future, extensions to DTLS may be specified that allow the use of
only one set of sequence numbers for deployment in constrained
environments.

Some transports, such as DCCP, provide congestion control for traffic
carried over them. If the congestion window is sufficiently narrow,
DTLS handshake retransmissions may be held rather than transmitted
immediately, potentially leading to timeouts and spurious
retransmission. When DTLS is used over such transports, care should
be taken not to overrun the likely congestion window. [DCCPDTLS]
defines a mapping of DTLS to DCCP that takes these issues into
account.

4.1.1.1. PMTU Issues

In general, DTLS's philosophy is to leave PMTU discovery to the
application. However, DTLS cannot completely ignore PMTU for three
reasons:

  - The DTLS record framing expands the datagram size,
    thus lowering the effective PMTU from the application's
    perspective.

  - In some implementations the application may not directly
    talk to the network, in which case the DTLS stack may
    absorb ICMP [RFC1191] Datagram Too Big indications
    or ICMPv6 [RFC1885] Packet Too Big indications.

  - The DTLS handshake messages can exceed the PMTU.

In order to deal with the first two issues, the DTLS record layer
SHOULD behave as described below.

If PMTU estimates are available from the underlying transport
protocol, they should be made available to upper layer protocols. In
particular:

- For DTLS over UDP, the upper layer protocol SHOULD be allowed

   to obtain the PMTU estimate maintained in the IP layer.

- For DTLS over DCCP, the upper layer protocol
  SHOULD be allowed to obtain the current estimate of the
  PMTU.

- For DTLS over TCP or SCTP, which automatically fragment
  and reassemble datagrams, there is no PMTU limitation.
  However, the upper layer protocol MUST NOT write any
  record that exceeds the maximum record size of 2^14 bytes.

The DTLS record layer SHOULD allow the upper layer protocol to
discover the amount of record expansion expected by the DTLS
processing. Note that this number is only an estimate because of
block padding and the potential use of DTLS compression.

If there is a transport protocol indication (either via ICMP or via a
refusal to send the datagram as in DCCP Section 14), then the DTLS
record layer MUST inform the upper layer protocol of the error.

The DTLS record layer SHOULD NOT interfere with upper layer protocols
performing PMTU discovery, whether via [RFC1191] or [RFC4821]
mechanisms. In particular:

- Where allowed by the underlying transport protocol,
  the upper layer protocol SHOULD be allowed to set
  the state of the DF bit (in IPv4) or prohibit local
  fragmentation (in IPv6).

- If the underlying transport protocol allows the application
  to request PMTU probing (e.g., DCCP), the DTLS record
  layer should honor this request.

The final issue is the DTLS handshake protocol. From the perspective
of the DTLS record layer, this is merely another upper layer
protocol. However, DTLS handshakes occur infrequently and involve
only a few round trips, and therefore the handshake protocol PMTU
handling places a premium on rapid completion over accurate PMTU
discovery. In order to allow connections under these circumstances,
DTLS implementations SHOULD follow the following rules:

- If the DTLS record layer informs the DTLS handshake layer
  that a message is too big, it SHOULD immediately attempt
  to fragment it, using any existing information about the
  PMTU.

- If repeated retransmissions do not result in a response, and the
  PMTU is unknown, subsequent retransmissions SHOULD back off to a

smaller record size, fragmenting the handshake message as
appropriate. This standard does not specify an exact number
of retransmits to attempt before backing off, but 2-3 seems
appropriate.

[4.1.2](#). Record Payload Protection

Like TLS, DTLS transmits data as a series of protected records. The
rest of this section describes the details of that format.

[4.1.2.1](#). MAC

The DTLS MAC is the same as that of TLS 1.2. 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.

TLS MAC calculation is parameterized on the protocol version number,
which, in the case of DTLS, is the on-the-wire version, i.e., {254,
253} for DTLS 1.2.

Note that one important difference between DTLS and TLS MAC handling
is that in TLS MAC errors must result in connection termination. In

DTLS, the receiving implementation MAY simply discard the offending
record and continue with the connection. This change is possible
because DTLS records are not dependent on each other in the way that
TLS records are.

In general, DTLS implementations SHOULD silently discard records with
bad MACs or that are otherwise invalid. They MAY log an error. If a
DTLS implementation chooses to generate an alert when it receives a
message with an invalid MAC, it MUST generate a bad_record_mac alert
with level fatal and terminate its connection state.

4.1.2.2. Null or Standard Stream Cipher

The DTLS NULL cipher is performed exactly as the TLS 1.2 NULL cipher.

The only stream cipher described in TLS 1.2 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 1.2.

4.1.2.3. AEAD Ciphers

TLS 1.2 introduced authenticated encryption with additional data
(AEAD) cipher suites. The existing AEAD cipher suites, defined in
[ECCGCM] and [RSAGCM] can be used with DTLS exactly as with TLS 1.2.

4.1.2.5. New Cipher Suites

Upon registration, new TLS cipher suites MUST indicate whether they
are suitable for DTLS usage and what, if any, adaptations must be
made (See Section 7 for IANA considerations).

4.1.2.6. 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 [ESP].

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 Section 3.4.3 of [ESP].

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.

4.1.2.7. Handling Invalid Records

Unlike TLS, DTLS is resilient in the face of invalid records (e.g.,
invalid formatting, length, MAC, etc.) In general, invalid records
SHOULD be silently discarded, thus preserving the association,
however an error MAY be logged for diagnostic purposes.
Implementations which choose to generate an alert instead, MUST
generate fatal level alerts to avoid attacks where the attacker
repeatedly probes the implementation to see how it responds to

various types of error. Note that if DTLS is run over UDP, then any
implementation which does this will be extremely susceptible to DoS
attacks because UDP forgery is so easy. Thus, this practice is NOT
RECOMMENDED for such transports.

If DTLS is being carried over a transport which is resistant to
forgery (e.g., SCTP with SCTP-AUTH), then it is safer to send alerts
because an attacker will have difficulty forging a datagram which
will not be rejected by the transport layer.

[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 has been added to prevent denial of
    service attacks.

    2. Modifications to the handshake header to handle message loss,
    reordering, and DTLS message fragmentation (in order to avoid IP
    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 1.2.

[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 to 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 counter both of these attacks, DTLS borrows the stateless
cookie technique used by Photuris [PHOTURIS] and IKE [IKEv2]. 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. This mechanism forces the attacker/client to be able
to receive the cookie, which makes DoS attacks with spoofed IP
addresses difficult. This mechanism does not provide any defense
against DoS attacks mounted from valid IP addresses.

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;
      opaque cookie<0..2^8-1>;                        // New field
      CipherSuite cipher_suites<2..2^16-1>;
      CompressionMethod compression_methods<1..2^8-1>;
    } ClientHello;
```

When sending the first ClientHello, the client does not have a cookie

yet; in this case, the Cookie field is left empty (zero length).

The definition of HelloVerifyRequest is as follows:

```
struct {
  ProtocolVersion server_version;
  opaque cookie<0..2^8-1>;
} HelloVerifyRequest;
```

The HelloVerifyRequest message type is hello_verify_request(3).

The server_version field is defined as in TLS.

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

Note: this specification increases the cookie size limit to 255 bytes
for greater future flexibility. The limit remains 32 for previous
versions of DTLS.

The DTLS server SHOULD generate cookies in such a way that they can
be verified without retaining any per-client state on the server.
One technique is to have a randomly generated secret and generate
cookies as: Cookie = HMAC(Secret, Client-IP, Client-Parameters)

When the second ClientHello is received, the server can verify that
the Cookie is valid and that the client can receive packets at the
given IP address.

One potential attack on this scheme is for the attacker to collect a
number of cookies from different addresses and then reuse them to
attack the server. The server can defend against this attack by
changing the Secret value frequently, thus invalidating those
cookies. If the server wishes that legitimate clients be able to
handshake through the transition (e.g., they received a cookie with
Secret 1 and then sent the second ClientHello after the server has
changed to Secret 2), the server can have a limited window during
which it accepts both secrets. [IKEv2] suggests adding a version
number to cookies to detect this case. An alternative approach is
simply to try verifying with both secrets.

DTLS servers SHOULD perform a cookie exchange whenever a new

handshake is being performed. If the server is being operated in an
environment where amplification is not a problem, the server MAY be
configured not to perform a cookie exchange. The default SHOULD be
that the exchange is performed, however. In addition, the server MAY
choose not to do a cookie exchange when a session is resumed.
Clients MUST be prepared to do a cookie exchange with every
handshake.

If HelloVerifyRequest is used, the initial ClientHello and
HelloVerifyRequest are not included in the calculation of the
handshake_messages (for the CertificateVerify message) and
verify_data (for the Finished message).

If a server receives a ClientHello with an invalid cookie, it SHOULD
treat it the same as a ClientHello with no cookie. This avoids
race/deadlock conditions if the client somehow gets a bad cookie
(e.g., because the server changes its cookie signing key).

Note to implementors: this may results in clients receiving multiple
HelloVerifyRequest messages with different cookies. Clients SHOULD
handle this by sending a new ClientHello with a cookie in response to
the new HelloVerifyRequest. Note that this message should have the
same handshake sequence number (0) and record sequence number (0),
thus avoiding the need to create state on the server. This also
implies that the client MUST NOT do replay suppression during the
initial handshake (this is safe as handshake messages have their own
sequence numbers). [OPEN ISSUE: It's not clear that this is the best
choice. Michael Tuexen suggested mimicing the client's record
sequence number instead.]

[4.2.2](4.2.2). Handshake Message Format

In order to support message loss, reordering, and message
fragmentation, DTLS modifies the TLS 1.2 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 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]

Note, however, that from the perspective of the DTLS record layer,
the retransmission is a new record.  This record will have a new
DTLSPlaintext.sequence_number value.

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. Handshake Message Fragmentation and Reassembly

As noted in Section 4.1.1, 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, each of which can be transmitted separately, thus
avoiding IP fragmentation.

When transmitting the handshake message, the sender divides the
message into a series of N contiguous data ranges. These ranges MUST
NOT be 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 if the PMTU estimate changes.

Note that as with TLS, multiple handshake messages may be placed in
the same DTLS record, provided that there is room and that they are
part of the same flight. Thus, there are two acceptable ways to pack
two DTLS messages into the same datagram: in the same record or in
separate records.

4.2.4. Timeout and Retransmission

DTLS messages are grouped into a series of message flights, according
to 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.

draft-ietf-tls-rfc4347-bis-00   DTLS   March 2011

```
      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. 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.

```
                       +-----------+
                       | PREPARING |
               +---> |           | <-------------------+
               |     |           |                     |
               |     +-----------+                     |
               |           |                           |
               |           |                           |
               |           | Buffer next flight        |
               |           |                           |
               |          \|/                          |
               |     +-----------+                     |
               |     |           |                     |
```

```
           |        | SENDING  |<-------------------+  |
           |        |          |                    |  | Send
           |        +----------+                    |  | HelloRequest
   Receive |          |                             |  |
      next |          | Send flight                 |  | or
    flight |  +-------+                             |  |
           |  |       | Set retransmit timer        |  | Receive
           |  |      \|/                            |  | HelloRequest
           |  |  +----------+                       |  | Send
           |  |  |          |                       |  | ClientHello
      +--)--|  | WAITING  |-------------------+  |
           |  |  |          |    Timer expires  |  |
           |  |  +----------+                   |  |
           |  |       |                          |  |
           |  |       |                          |  |
           |  |       +-----------------------+  |
           |  |                Read retransmit    |
           |  |                                   |
   Receive |  |                                   |
      last |  |                                   |
    flight |  |                                   |
           |  |                                   |
         \|/\|/                                  |
                                                  |
      +----------+                                |
      |          |                                |
      | FINISHED | ------------------------------+
      |          |
      +----------+
          |  /|\
          |   |
          |   |
        +---+
```

```
       Read retransmit
     Retransmit last flight
```

        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 FINISHED state if this is the last
flight in the handshake. Or, if the implementation expects to
receive more messages, it 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.

When the server desires a rehandshake, it transitions from the
FINISHED state to the PREPARING state to transmit the HelloRequest.
When the client receives a HelloRequest it transitions from FINISHED
to PREPARING to transmit the ClientHello.

In addition, for at least 2MSL, when in the FINISHED state, the node
which transmits the last flight (the server in an ordinary handshake

or the client in a resumed handshake) MUST respond to a retransmit of
the peer's last flight with a retransmit of the last flight. This
avoids deadlock conditions if the last flight gets lost. This
requirement applies to DTLS 1.0 as well, and though not explicit in
[DTLS1] but was always required for the state machine to function
correctly. To see why this is necessary, consider what happens in an
ordinary handshake if the server's Finished is lost: the server
believes the handshake is complete but it actually is not. As the
client is waiting for the Finished, the client's retransmit timer
will fire and it will retransmit the client Finished, causing the
server to respond with its own Finished, completing the handshake.
The same logic applies on the server side for the resumed handshake.

Note that because of packet loss it is possible for one side to be
sending application data even though the other side has not received
the first side's Finished. Implementations MUST either discard or
buffer all application data packets for the new epoch until they have
received the Finished for that epoch. Implementations MAY treat
receipt of application data with a new epoch prior to receipt of the
corresponding Finished as evidence of reordering or packet loss and
retransmit their final flight immediately, shortcutting the
retransmission timer.

4.2.4.1. Timer Values

Though timer values are the choice of the implementation, mishandling
of the timer can lead to serious congestion problems; for example, if
many instances of a DTLS time out early and retransmit too quickly on
a congested link. Implementations SHOULD use an initial timer value
of 1 second (the minimum defined in RFC 2988 [RFC2988]) and double
the value at each retransmission, up to no less than the RFC 2988
maximum of 60 seconds. Note that we recommend a 1-second timer
rather than the 3-second RFC 2988 default in order to improve latency
for time-sensitive applications. Because DTLS only uses
retransmission for handshake and not dataflow, the effect on
congestion should be minimal.

Implementations SHOULD retain the current timer value until a
transmission without loss occurs, at which time the value may be
reset to the initial value. After a long period of idleness, no less
than 10 times the current timer value, implementations may reset the
timer to the initial value. One situation where this might occur is
when a rehandshake is used after substantial data transfer.

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. CertificateVerify and Finished Messages

CertificateVerify and Finished messages have the same format as in
TLS. Hash calculations include entire handshake messages, including
DTLS specific fields: message_seq, fragment_offset and
fragment_length. However, in order to remove sensitivity to
handshake message 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 MUST NOT be included in the
CertificateVerify or Finished MAC computations.

4.2.7. Alert Messages

Note that Alert messages are not retransmitted at all, even when they
occur in the context of a handshake. However, a DTLS implementation
SHOULD generate a new alert message if the offending record is
received again (e.g., as a retransmitted handshake message).
Implementations SHOULD detect when a peer is persistently sending bad
messages and terminate the local connection state after such
misbehavior is detected.

4.2.8. Establishing New Associations With Existing Parameters

If a DTLS client-server pair are configured in such a way that
repeated connections happen on the same host/port quartet, then it is
possible that a client will silently abandon one connection and then
initiate another with the same parameters (e.g., after a reboot).
This will appear to the server as a new handshake with epoch=0. In
cases where a server believes it has an existing association on a
given host/port quartet and it receives an epoch=0 ClientHello, it
SHOULD proceed with a new handshake but MUST NOT destroy the existing
association until the client has demonstrated reachability either by
completing a cookie exchange or by completing a complete handshake
including delivering a verifiable Finished message. After a correct
Finished is received, the server MUST abandon the previous
association to avoid confusion between two valid associations with
overlapping epochs. The reachability requirement prevents off-
path/blind attackers from destroying associations merely by sending

forged ClientHellos.

4.3. Summary of new syntax

   This section includes specifications for the data structures that

   have changed between TLS 1.2 and DTLS 1.2. See [TLS12] for the
   definition of this syntax.

4.3.1. Record Layer

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

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

struct {
  ContentType type;
  ProtocolVersion version;
  uint16 epoch;                                  // New field
  uint48 sequence_number;                        // New field
  uint16 length;
  select (CipherSpec.cipher_type) {
    case block:  GenericBlockCipher;
```

```
            case aead:   GenericAEADCipher;                  // New field
          } fragment;
        } DTLSCiphertext;

4.3.2. Handshake Protocol

        enum {
          hello_request(0), client_hello(1), server_hello(2),
          hello_verify_request(3),                           // New field
          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 field
          uint24 fragment_offset;                            // New field
```

```
          uint24 fragment_length;                            // New field
          select (HandshakeType) {
            case hello_request: HelloRequest;
            case client_hello:  ClientHello;
            case server_hello:  ServerHello;
            case hello_verify_request: HelloVerifyRequest;  // New field
            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 {
          ProtocolVersion client_version;
          Random random;
          SessionID session_id;
          opaque cookie<0..2^8-1>;                           // New field
```

```
         CipherSuite cipher_suites<2..2^16-1>;
         CompressionMethod compression_methods<1..2^8-1>;
       } ClientHello;

       struct {
         ProtocolVersion server_version;
         opaque cookie<0..2^8-1>;
       } HelloVerifyRequest;
```

5. Security Considerations

   This document describes a variant of TLS 1.2 and therefore most of
   the security considerations are the same as those of TLS 1.2 [TLS12],
   described in Appendices D, E, and F.

   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. Clients MUST be prepared to do a
   cookie exchange with every handshake.

   Unlike TLS implementations DTLS implementations SHOULD NOT respond to

   invalid records by terminating the connection. See Section 4.1.2.7
   for details on this.

6. Acknowledgements

The authors would like to thank Dan Boneh, Eu-Jin Goh, Russ Housley,
Constantine Sapuntzakis, and Hovav Shacham for discussions and
comments on the design of DTLS. Thanks to the anonymous NDSS
reviewers of our original NDSS paper on DTLS [DTLS] for their
comments. Also, thanks to Steve Kent for feedback that helped

clarify many points. The section on PMTU was cribbed from the DCCP
specification [DCCP]. Pasi Eronen provided a detailed review of this
specification. Peter Saint-Andre provided the changes list in Section
8. elpful comments on the document were also received from Mark
Allman, Jari Arkko, Mohamed Badra, Michael D'Errico, Adrian Farrell,
Joel Halpern, Ted Hardie, Charlia Kaufman, Pekka Savola, Allison
Mankin, Nikos Mavrogiannopoulos, Alexey Melnikov, Robin Seggelman,
Michael Tuexen, Juho Vaha-Herttua, and Florian Weimer.

7. IANA Considerations

This document uses the same identifier space as TLS [TLS12], so no
new IANA registries are required. When new identifiers are assigned
for TLS, authors MUST specify whether they are suitable for DTLS.
IANA [should modify/has modified] all TLS parameter registries to add
a DTLS-OK flag, indicating whether the specification may be used with
DTLS.

This document defines a new handshake message, hello_verify_request,
whose value has been allocated from the TLS HandshakeType registry
defined in [TLS12]. The value "3" has been assigned by the IANA.

8. Changes Since DTLS 1.0

This document reflects the following changes since DTLS 1.0 [DTLS1].

   - Updated to match TLS 1.2 [TLS12].

   - Addition of AEAD Ciphers in Section 4.1.2.3 (tracking changes in
     TLS 1.2.

   - Clarifications regarding sequence numbers and epochs in Section 4.1
     and a clear procedure for dealing with state loss in Section 4.2.8.

   - Clarifications and more detailed rules regarding Path MTU issues in
     Section 4.1.1.1. Clarification of the fragmentation text throughout.

   - Clarifications regarding handling of invalid records in Section 4.1.2.7.

   - A new paragraph describing handling of invalid cookies at the end of

Section 4.2.1.

- Some new text describing how to avoid handshake deadlock conditions at the end of Section 4.2.4.

- Some new text about CertificateVerify messages in Section 4.2.6.

- A prohibition on epoch wrapping in Section 4.1.

- Clarification of the IANA requirements and the explicit requirement for a new IANA registration flag for each parameter.

- Numerous editorial changes.

9. References

9.1. Normative References

   [RFC1191]   Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191,
               November 1990.

   [RFC1885]   Conta, A., and S. Deering, "Internet Control Message
               Protocol (ICMPv6) for the Internet Protocol Version 6
               (IPv6) Specification", RFC 1885, December 1995.

   [RFC4301]   Kent, S. and K. Seo, "Security Architecture for the
               Internet Protocol", RFC 4301, December 2005.

   [RFC2988]   Paxson, V. and M. Allman, "Computing TCP's Retransmission
               Timer", RFC 2988, November 2000.

   [RFC4821]   Mathis, M., and J. Heffner, "Packetization Layer Path MTU
               Discovery", RFC 4821, March 2007.

   [RSAGCM]   Salowey, J., Choudhury, A., and D. McGrew, "AES-GCM Cipher
               Suites for TLS", RFC 5288, August 2008.

   [TCP]   Postel, J., "Transmission Control Protocol", STD 7, RFC
               793, September 1981.

   [TLS12]   Dierks, T. and E. Rescorla, "The Transport Layer Security
               (TLS) Protocol Version 1.2", RFC 5246, May 2008.

9.2. Informative References

    [DCCP]   Kohler, E., Handley, M., Floyd, S., Padhye, J., "Datagram
                Congestion Control Protocol", Work in Progress, 10 March
                2005.

    [DCCPDTLS] T. Phelan, "Datagram Transport Layer Security (DTLS) over
                the Datagram Congestion Control Protocol (DCCP)", RFC
                5238, May 2008.

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

    [DTLS1]   Rescorla, E., and N. Modadugu, "Datagram Transport Layer
                Security", RFC 4347, April 2006.

    [ECCGCM]   E. Rescorla, "TLS Elliptic Curve Cipher Suites with
                SHA-256/384 and AES Galois Counter Mode", RFC 5289, August
                2008.

    [ESP]   S. Kent "IP Encapsulating Security Payload (ESP)", RFC
                4303, December 2005.

    [IKEv2]   C. Kaufman (ed), "Internet Key Exchange (IKEv2) Protocol",
                RFC 4306, December 2005.

    [IMAP]   Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - VERSION
                4rev1", RFC 3501, March 2003.

    [PHOTURIS] Karn, P. and W. Simpson, "Photuris: Session-Key Management
                Protocol", RFC 2522, March 1999.

    [POP]   Myers, J. and M. Rose, "Post Office Protocol - Version 3",
                STD 53, RFC 1939, May 1996.

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

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

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

   [TLS11]   Dierks, T. and E. Rescorla, "The Transport Layer Security

---

              (TLS) Protocol Version 1.1", RFC 4346, April 2006.

   [WHYIPSEC] Bellovin, S., "Guidelines for Mandating the Use of IPsec",
              RFC 5406, October 2003.

Authors' Addresses

   Eric Rescorla
   RTFM, Inc.
   2064 Edgewood Drive
   Palo Alto, CA 94303

   EMail: ekr@rtfm.com

   Nagendra Modadugu
   Computer Science Department
   Stanford University
   353 Serra Mall
   Stanford, CA 94305

   EMail: nagendra@cs.stanford.edu