Network Working Group
Request for Comments: 4347
Category: Standards Track

E. Rescorla
RTFM, Inc.
N. Modadugu
Stanford University
April 2006

数据报传输层安全（DTLS）

摘要

•此文档旨在阐明 1.0 版数据报传输层安全(DTLS)协议。此协议为数据报协议提供了交互隐私性。它允许客户端/服务端应用在交互过程中免遭窃听、篡改以及信息伪造的威胁。DTLS 协议是在传输层安全的基础上进行修改同时提供与其相同的安全保障。底层传输的数据报语义由 DTLS 协议保留。

目录

1.引言

在安全网络传输所使用的协议之中，TLS 被最为广泛地使用。在网页流量以及电子邮箱协议（例如 IMAP 和 POP）中，常常使用 TLS 来保证其安全性。在最初，TLS 的优势在于其提供了一条面向连接的透明传输通道。因此，在传输层和应用层之间引入 TLS 来确保应用层安全是较为容易的。然而，由于必须运行在一条值得信赖的传输通道上（通常为 TCP），使得 TLS 无法被用于保障不可靠的数据报传输的安全。

但是过去几年，越来越多的应用层协议，尤其是会话初始化协议和一些电子游戏协议在使用 UDP 传输(需要注意的是虽然 SIP 可以同时在 TCP 和 UDP 上运行，但在某些场景中人们更倾向于使用 UDP)。如今，这些应用的设计者们面临着一系列选择，而这些选择总是不尽如人意。第一种选择，是他们可以使用 IPsec [RFC2401]。然而，由于有太多的原因详见[WHYIPSEC]，使得其只能在部分应用场景里适用。第二种选择是，这些设计者们可以自行设计一个客户端应用层协议，例如 SIP 就使用了 S/MIME 中的一个子集来保证其安全性。 不幸的是，虽然应用层安全协议为其提供了极高的安全属性（例如 S/MIME 确保了端到端的安全）,但相比起运行在 TLS 上，设计一个应用层安全协议所需的工作量显然太大了。

大多数情况下，保证客户端/服务端应用安全的理想方式是使用 TLS; TLS 然而在数据报语义的要求下，TLS 的使用是被自动禁止的。因此，一种可取的方式是设计一个兼容数据报的 TLS 变体即本文描述的这个协议：数据报传输安全（DTLS）。 DTLS 被故意设计得与 TLS 尽可能相似，在最大程度地减少新的安全发明同时，减少代码和基础结构的重用量。

1.1 术语要求

此文档中，关键词 "必须"（MUST），"绝不能"（MUST NOT），"要求"（REQUIRED），"应当"（SHOULD），"不应该"（SHOULD NOT）以及 "可能"（MAY） 将在 RFC 2119 [REQ]中给予解释。

2.使用模式

DTLS 协议被设计于用来保护应用交互中的数据安全。它是在应用空间里运行而不涉及到内核的修改。数据报传输不要求提供可靠的亦或有序的数据发送。DTLS 协议保存了数据载荷的这样一种属性。一些应用，例如多媒体流，网络通话以及在线游戏由于其在数据传输过程中对于传输延迟较为敏感的特性，使用数据报传输作为交互方式。由于 DTLS 协议不会要求补偿丢失或重新排序的数据流量，因此当使用 DTLS 来保证交互通信安全时，上述应用的行为是不会受到改变的。

3. DTLS 概述

DTLS 基本的设计哲学是构建一个“数据报上的 TLS”。TLS 无法直接在数据报环境上运行，原因十分简单：数据包有可能会丢失或者乱序。TLS 没有内部功能来处理这样一种不可靠性，因此，在数据报传输上重新托管时，TLS 实现会中断。DTLS 的目的便是仅仅通过对 TLS 最小的改动来修补这个问题。DTLS 在最大程度上与 TLS 相同。每当我们需要发明新特性时，我们都会尝试以保留 TLS 风格的方式进行。不可靠性会在两个级别上给 TLS 带来问题：1. TLS 的流量加密层不允许对单个记录进行独立解密。如果未收到记录 N，则无法解密记录 N+1。2. TLS 握手层假定握手消息可靠传递，如果这些消息丢失，则会中断。本节的其余部分介绍 DTLS 用于解决这些问题的方法。

3.1.对于丢失不敏感的消息传递

在 TLS 的流量加密层（称为 TLS 记录层）中，记录不是独立的。记录间依赖关系有两种：

1.依赖上下文加密 (CBS 模式，流密码密钥流)在记录间是绑定的。

2.反重放和消息重新排序保护由 MAC 提供包含序列号但序列号隐含在记录中。

在 IPsec ESP [ESP]中这两个问题的解决方法很简单且众所周知：向记录添加显式状态。TLS 1.1[TLS11] 已将显式 CBC 状态添加到 TLS 记录。DTLS 借用该机制并添加显式序列号。

3.2.为握手提供可靠性

TLS 的握手是锁步加密握手。信息的发送和接收必须依照规定的顺序，否则被视为错误。显然，这样的做法与丢失和重新排序不符合。除此之外，由于 TLS 握手信息隐式的比任意数据报大，因此会有分片的问题。DTLS 必须解决这两个难题。

3.2.1.丢包

DTLS 使用简单的重传计时器来处理丢包。下图展示了 DTLS 握手第一阶段的基本概念。

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

```
ClientHello          ------>

                              X<-- HelloVerifyRequest
                                                (lost)

[Timer Expires]

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

一旦客户端发送了一个 ClientHello 信息，它便等待看到来自服务器的 HelloVerifyRequest。然而，如果服务器信息丢失，客户端便知 ClientHello 或 HelloVerifyRequest 已经丢失，于是客户端便重新发送信息。当服务器收到了重传信息后，它便知道重传。服务器还维护一个重传计时器，并在该计时器到期时重新传输。

注意：超时和重传不会应用在 HelloVerifyRequest 中，因为这需要在服务器上创建状态。

3.2.2.重排序

在 DTLS 中，每个握手消息在该握手中分配一个特定的序列号。当对等方收到握手消息时，它可以快速确定该消息是否是它期望的下一条消息。如果是，那么它会处理它。如果没有，它会在收到所有以前的消息后将其排队以供将来处理。

3.2.3.消息大小

TLS 和 DTLS 握手消息可能非常大（理论上高达 2^24-1 字节，实际上为千字节）。相反，UDP 数据报在没有分片的情况下通常限制在 1500 字节以内。为了弥补这一限制，每个 DTLS 握手消息可能会分段在多个 DTLS 记录中。每个 DTLS 握手消息同时包含片段偏移量和片段长度。因此，拥有握手所有字节的接收者消息可以重新组合原始未分段的消息。

3.3.延迟检测

DTLS 可选择支持记录重播检测。使用的技术与 IPsec AH/ESP 中的相同，通过维护位图窗口收到的记录。太旧而无法放入窗口的记录，以及以前收到的记录将以静默方式丢弃。重放检测功能是可选的，因为数据包重复并不总是恶意的，也可能是由于路由错误而发生的。可以想象，应用程序可能会检测到重复的数据包，并相应地修改其数据传输策略。

4.与 TLS 的区别

如第 3 节所述，DTLS 有意与 TLS 非常相似。因此，我们没有将 DTLS 视为新协议，而是把它看作 TLS 1.1 [TLS11]的一系列增量。在我们没有明确指出差异的地方，DTLS 与[TLS11]相同。

4.1.记录层

DTLS 记录层与 TLS 1.1 非常相似。唯一的更改是在记录中包含显式序列号。此序列号允许接收者正确验证 TLS MAC。DTLS 记录格式如下所示：

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

type
与 TLS 1.1 记录中的 type 域一样。

version
正在使用的协议的版本。 本文档描述了 DTLS 版本 1.0，它使用版本 { 254， 255}.版本值 254.255 是 DTLS 1.0 的补充。TLS 和 DTLS 版本号之间的最大间距可确保来自两个协议的记录可以很容易区分。 应该注意的是，未来的在线 DTLS 的版本号值正在下降（而真正的版本号的值正在增加。）

epoch
每次密码状态更改时递增的计数器值。

sequence_number
此记录的序列号。

length
与 TLS 1.1 记录中的长度字段相同。与 TLS1.1 一样，长度不应超过 2^14。

fragment
与 TLS 1.1 记录的片段字段相同。

DTLS 在记录的 sequence_number 字段中使用显式序列号，而不是隐式序列号。 与 TLS 一样，在每发送一条 ChangeCipherSpec 消息后，序列号设置为零。

如果连续执行多次握手，则可能会有是线路上具有序列号相同但密码状态不同的多个记录。epoch 字段允许收件人区分此类数据包。epoch 数最初为零，为每次发送更改密码规范消息时递增。在为了确保任何给定的 sequence/epoch 对是唯一的，在 TCP 最大段生存期的两倍内，**绝不能**允许相同 epoch 值的复用。在实践中，TLS 实现很少重新握手，因此我们不认为这是一个问题。

4.1.1.传输层匹配

每个 DTLS 记录**必须**适合单个数据报。为了避免 IP 分段 [MOGUL]，DTLS 实现**应当**确定 MTU 并发送小于 MTU 的记录。DTLS **应当**为应用程序提供一种确定值的方法 PMTU（或是最大应用程序数据报大小，即 PMTU 减去每条记录开销的 DTLS）。如果应用程序尝试发送大于 MTU 的记录，DTLS 实现**应当**生成一个错误，从而避免发送将被分段的数据包。

需要注意的是，与 IPsec 不同，DTLS 记录不包含任何关联标识符。应用程序必须安排在关联之间多路复用。对于 UDP，这大概是通过主机/端口号完成的。

多个 DTLS 记录可以放置在单个数据报中。它们只是连续编码。DTLS 记录成帧足以确定边界。但请注意，数据报有效负载的第一个字节必须是记录的开头。记录不能跨越数据报。

某些传输协议（如 DCCP [DCCP]）提供自己的序列号。当通过这些传输时，DTLS 和传输序列号都将存在。虽然这会带来少量的低效率，但传输层和 DTLS 序列号用于不同的目的，因此为了概念上的简单性，最好同时使用这两个序列号。将来，可能会指定 DTLS 的扩展，以仅允许使用一组序列号从而在受限环境中进行部署。

一些传输，例如 DCCP，为通过它们传输的流量提供拥塞控制。如果拥塞窗口足够窄，则可能会保留 DTLS 握手重传，而不是立即传输，这可能会导致超时和杂散重传。在此类传输上使用 DTLS 时，应注意不要超出可能的拥塞窗口。将来，可以指定 DTLS-DCCP 映射，以便为此交互提供最佳行为。

4.1.1.1. PMTU 发现

一般来说，DTLS 的理念是避免处理 PMTU 问题。一般策略是从保守的 MTU 开始，然后在握手或实际应用程序数据传输阶段的事件需要它时更新它。

PMTU 应从将用于发送数据包的接口 MTU 初始化。如果 DTLS 实现收到带有“需要分段和 DF 集”代码（也称为数据报太大）的 ICMP 目标不可达“消息 RFC 1191 [RFC1191]，则应将其 PMTU 估计值降低到 ICMP 消息中给出的值。DTLS 实现**应当**允许应用程序偶尔重置其 PMTU 估计值。DTLS 实现还**应当**允许应用程序控制 DF 位的状态。这些控件允许应用程序执行 PMTU 发现。IPv6 **应当**遵循 RFC 1981 [RFC1981] 过程。

一个特例是 DTLS 握手系统。握手消息应设置为 DF。由于某些防火墙和路由器会屏蔽 ICMP 消息，因此握手层很难将数据包丢失与过大的 PMTU 估计区分开来。为了在这些情况下允许连接，DTLS 实现**应当**在第 4.2.4 节中所述的重新传输退避期间回退握手数据包大小。例如，如果发送一个大数据包，则在重传 3 次后，握手层可能会选择在重新传输时对握手消息进行分段。通常，选择保守的初始 MTU 可以避免此问题。

4.1.2.记录载荷保护

与 TLS 一样，DTLS 将数据作为一系列受保护的记录传输。本节的其余部分介绍该格式的详细信息。

4.1.2.1. MAC

DTLS MAC 与 TLS 1.1 相同。但是，用于计算 MAC 的序列号不是使用 TLS 的隐式序列号，而是通过将 epoch 和序列号按它们在网络上出现的顺序连接起来而形成的 64 位值。请注意，DTLS epoch + 序列号与 TLS 序列号的长度相同。

TLS MAC 计算在协议版本号上参数化，对于 DTLS，该版本号是在线版本，即 DTLS 1.0 的 {254，255 }。

请注意，DTLS 和 TLS MAC 处理之间的一个重要区别是，在 TLS 中，MAC 错误必须导致连接终止。在 DTLS 中，接收实现可以简单地丢弃有问题的记录并继续连接。此更改是可能的，因为 DTLS 记录不像 TLS 记录那样相互依赖。

通常，DTLS 实现应静默丢弃具有错误 MAC 的数据。如果 DTLS 实现在收到具有无效 MAC 的消息时选择生成警报，则**必须**生成级别为致命 bad_record_mac 警报并终止其连接状态。

4.1.2.2. 空或标准流密码

DTLS NULL 密码的执行方式与 TLS 1.1 NULL 密码完全相同。

TLS 1.1 中描述的唯一流密码是 RC4，它不能随机访问。RC4 **绝不能与** DTLS 一起使用。

4.1.2.3. 分组密码

DTLS 分组密码加密和解密与 TLS 1.1 完全一致。

4.1.2.4. 新的密码套件

注册后，新的 TLS 密码套件必须表明它们是否适合 DTLS 使用，以及必须进行哪些调整（如果有的话）。

4.1.2.5. 反重放

DTLS 记录包含序列号以提供重播保护。序列号验证**应当**使用以下滑动窗口过程执行，该过程借用自 [RFC 2402] 的第 3.4.3 节。

建立会话时，此会话的接收方数据包计数器**必须**初始化为零。对于每个收到的记录，接收方**必须**验证该记录是否包含不重复在此会话生命周期内收到的任何其他记录的序列号的序列号。这**应该**是数据包与会话匹配后应用于数据包的第一次检查，以加快拒绝重复记录的速度。

通过使用滑动接收窗口拒绝重复项。（窗口的实现方式是局部问题，但下文描述了实现必须展示的功能。必须支持最小窗口大小 32，但首选窗口大小 64，并且应用作默认值。接收器可以选择另一个窗口大小（大于最小值）。（接收方不会通知发送方窗口大小。）

窗口的“右”边缘表示在此会话中收到的最高验证序列号值。包含低于窗口“左”边缘的序列号的记录将被拒绝。根据窗口中收到的数据包列表检查落在窗口内的数据包。[RFC 2401] 的附录 C 中描述了一种基于位掩码使用执行此检查的有效方法。

如果收到的记录落在窗口内并且是新的，或者数据包位于窗口右侧，则接收方继续进行 MAC 验证。如果 MAC 验证失败，接收方必须丢弃收到的无效记录。仅当 MAC 验证成功时，才会更新接收窗口。

4.2.DTLS 握手协议

DTLS 使用的所有握手消息和流与 TLS 相同，但有三个主要更改：

1. 添加了无状态 cookie 交换以防止拒绝服务攻击。
2. 修改握手标头以处理消息丢失、重新排序和分段。
3.重传计数器处理消息丢失。

除这些例外情况外，DTLS 消息格式、流和逻辑与 TLS 1.1 相同。

4.2.1.防止拒绝服务攻击的策略

数据报安全协议极易受到各种拒绝服务 （DoS） 攻击。有两次攻击特别值得关注：

1. 攻击者可以通过发送一系列握手发起请求来消耗服务器上的过多资源，从而导致服务器不断分配状态并可能执行大开销的加密操作。

2. 攻击者可以通过发送带有受害者伪造来源的连接启动消息来将服务器用作傀儡机。然后，服务器将其下一条消息（在 DTLS 中为证书消息，可能非常大）发送到受害计算机，从而形成洪泛。

为了应对这两种攻击，DTLS 借用了 Photuris [PHOTURIS] 和 IKE [IKE] 使用的无状态 cookie 技术。当客户端将其 ClientHello 消息发送到服务器时，服务器**可能**会使用 HelloVerifyRequest 消息进行响应。此消息包含使用 [PHOTURIS] 技术生成的无状态 Cookie。客户端**必须**重新传输添加了 cookie 的 ClientHello。然后，服务器验证 cookie 并仅在握手有效时才继续握手。这种机制强制攻击者/客户端能够接收 cookie，这使得使用欺骗性 IP 地址的 DoS 攻击变得困难。此机制不对从有效 IP 地址发起的 DoS 攻击提供任何防御。

交换如下图所示：

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

                            <------ HelloVerifyRequest
                                      (包含 cookie)

      ClientHello           ------>
      (携带 cookie)

      [Rest of handshake]
```

因此，DTLS 修改了 ClientHello 消息并增加 cookie 值。

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

发送第一个客户端的 Hello 时，客户端还没有 cookie;在这种情况下，Cookie 字段留空（字段长度为零）。

HelloVerifyRequest 的定义如下：

```
struct {
```

```
        ProtocolVersion server_version;
        opaque cookie<0..32>;
      } HelloVerifyRequest;
```

HelloVerifyRequest 消息类型为 hello_verify_request（3）。

server_version 字段的定义与 TLS 中相同。

在响应 HelloVerifyRequest 时，客户端**必须**使用与原始 ClientHello 中相同的参数值（version、random、session_id、cipher_suites compression_method）。服务器**应当**使用这些值来生成其 cookie，并在收到 cookie 时验证它们是否正确。服务器**必须**在 HelloVerifyRequest 中使用与发送 ServerHello 时相同的版本号。收到 ServerHello 后，客户端**必须**验证服务器版本值是否匹配。

DTLS 服务器**应当**以一种可以在服务器上保留任何每客户端状态的情况下对其进行验证的方式生成 Cookie。一种技术是拥有一个随机生成的密钥并生成 cookie：Cookie = HMAC（Secret，Client-IP，Client-Parameters）

当收到第二个 ClientHello 时，服务器可以验证 Cookie 是否有效，以及客户端是否可以在给定的 IP 地址接收数据包。

对这种方案的一种潜在攻击是攻击者从不同的地址收集大量 cookie，然后重复使用它们来攻击服务器。服务器可以通过频繁更改机密值来防御此攻击，从而使这些 cookie 失效。如果服务器希望合法客户端能够在转换过程中实现握手（例如，他们收到一个包含密钥 1 的 cookie，然后在服务器更改为密钥 2 后发送第二个 ClientHello），则服务器可以有一个有限的窗口，在此期间它接受两个机密 [IKEv2] 建议向 cookie 添加版本号以检测这种情况。另一种方法是尝试使用这两个机密进行验证。

每当执行新的握手时，DTLS 服务器都**应当**执行 cookie 交换。如果服务器在放大攻击不是问题的环境中运行，则服务器可能配置为不执行 cookie 交换。但是，默认值**应当**是执行交换。此外，服务器可能会选择在会话恢复时不进行 cookie 交换。客户**必须**准备好在每次握手时进行 cookie 交换。

如果使用 HelloVerifyRequest，则初始 ClientHello 和 HelloVerifyRequest 不包括在 Finished 消息的 verify_data 计算中。

4.2.2.握手信息格式

为了支持消息丢失、重新排序和分段，DTLS 修改了 TLS 1.1 握手标头：

```
struct {
        HandshakeType msg_type;
        uint24 length;
        uint16 message_seq;                                  // 新字段
        uint24 fragment_offset;                              // 新字段
        uint24 fragment_length;                              // 新字段
        select (HandshakeType) {
          case hello_request: HelloRequest;
          case client_hello:  ClientHello;
      case hello_verify_request: HelloVerifyRequest;  // 新类型
          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;
```

双方在每次握手中传输的第一条消息始终为 message_seq = 0。 每当生成一条新消息时，message_seq 值都会递增 1。 重新传输消息时，将使用相同的 message_seq 值。 例如：

```
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]
```

注意，从 DTLS 记录层的角度来看，重新传输是一条新记录。此记录将具有新的 DTLS Plaintext.sequence_number 值。

DTLS 维护（至少在名义上）一个 next_receive_seq 计数器。此计数器最初设置为零。收到消息时，如果其序列号与 next_receive_seq 匹配，则 next_receive_seq 递增并处理消息。如果序列号小于 next_receive_seq，则必须丢弃该消息。如果序列号大于 next_receive_seq，则应将消息排队，但可能会丢弃它。（取决于空间和带宽之间的权衡）。

4.2.3.消息分段和重组

如第 4.1.1 节所述，每个 DTLS 消息必须适合单个传输层数据报。 但是，握手消息可能大于最大记录大小。因此，DTLS 提供了一种机制，用于对多个记录上的握手消息进行分段。

传输握手消息时，发送方将消息划分为一系列 N 个连续的数据范围。 这些范围**绝不能**大于最大握手片段大小，并且**必须**共同包含整个握手消息。 范围**不应该**重叠。 然后，发送方创建 N 条握手消息，所有消息的 message_seq 值都与原始握手消息相同。 每条新消息都标有 fragment_offset（先前片段中包含的字节数）和 fragment_length（此片段的长度）。

所有消息中的长度字段与原始消息的长度字段相同。 未分段的消息是 fragment_offset=0 且 fragment_length=length 的一种退化情况。

当 DTLS 收到握手消息片段时，它**必须**对它进行缓冲存储，直至获取整个握手消息。DTLS 必须能够处理重叠的片段范围。这允许发送方在路径 MTU 发现期间重新传输具有较小片段大小的握手消息。

请注意，与 TLS 一样，多个握手消息可以放在同一个 DTLS 记录中，前提是有空间并且它们是同一批次的一部分。 因此，有两种可接受的方法可以将两个 DTLS 消息打包到同一个数据报中：在同一记录中或在不同的记录中。

4.2.4.超时和重传

根据下图，DTLS 消息被分组到一系列消息批次中。 尽管每一批消息可能包含许多消息，但出于超时和重新传输的目的，应将它们视为整体。

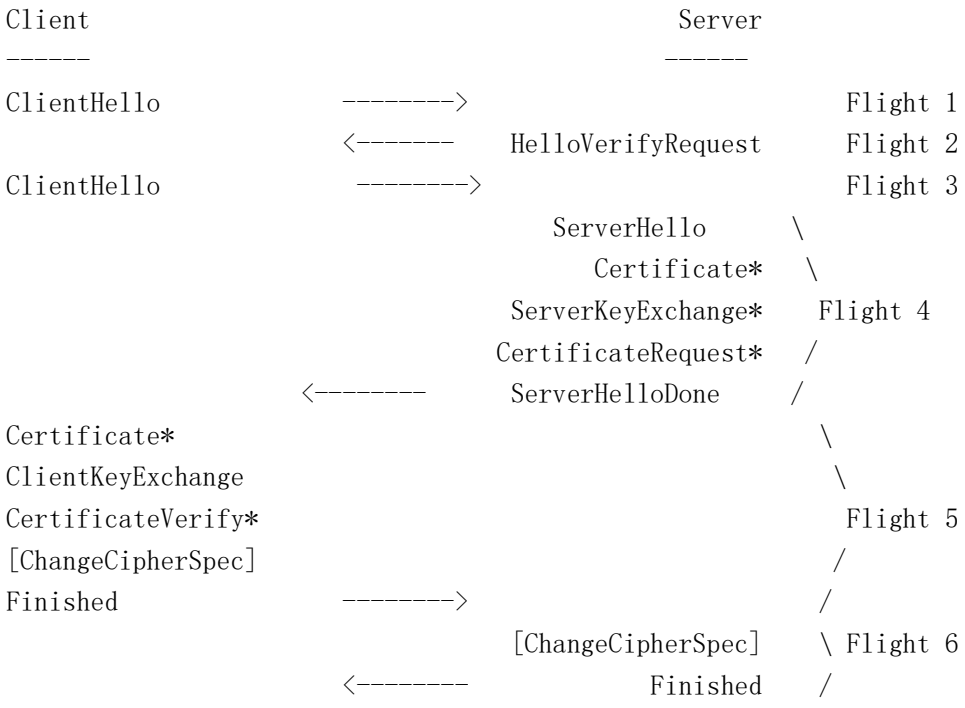

Figure 1. 有完整握手的信息交换流

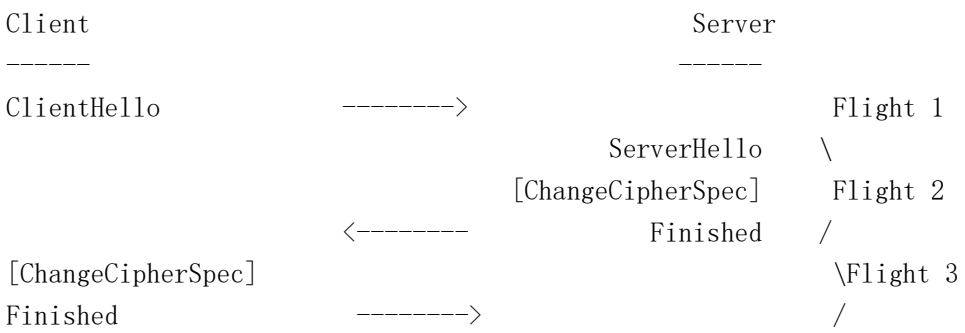

Figure 2. 会话恢复后握手的消息交换流
(没有 cookie 交换)

DTLS 对以下状态机使用简单的超时和重新传输方案。 由于 DTLS 客户端发送第一条消息 （ClientHello），因此它们以“正在准备”状态启动。 DTLS 服务器以 WAIT 状态启动，但缓冲区为空，没有重新传输计时器。

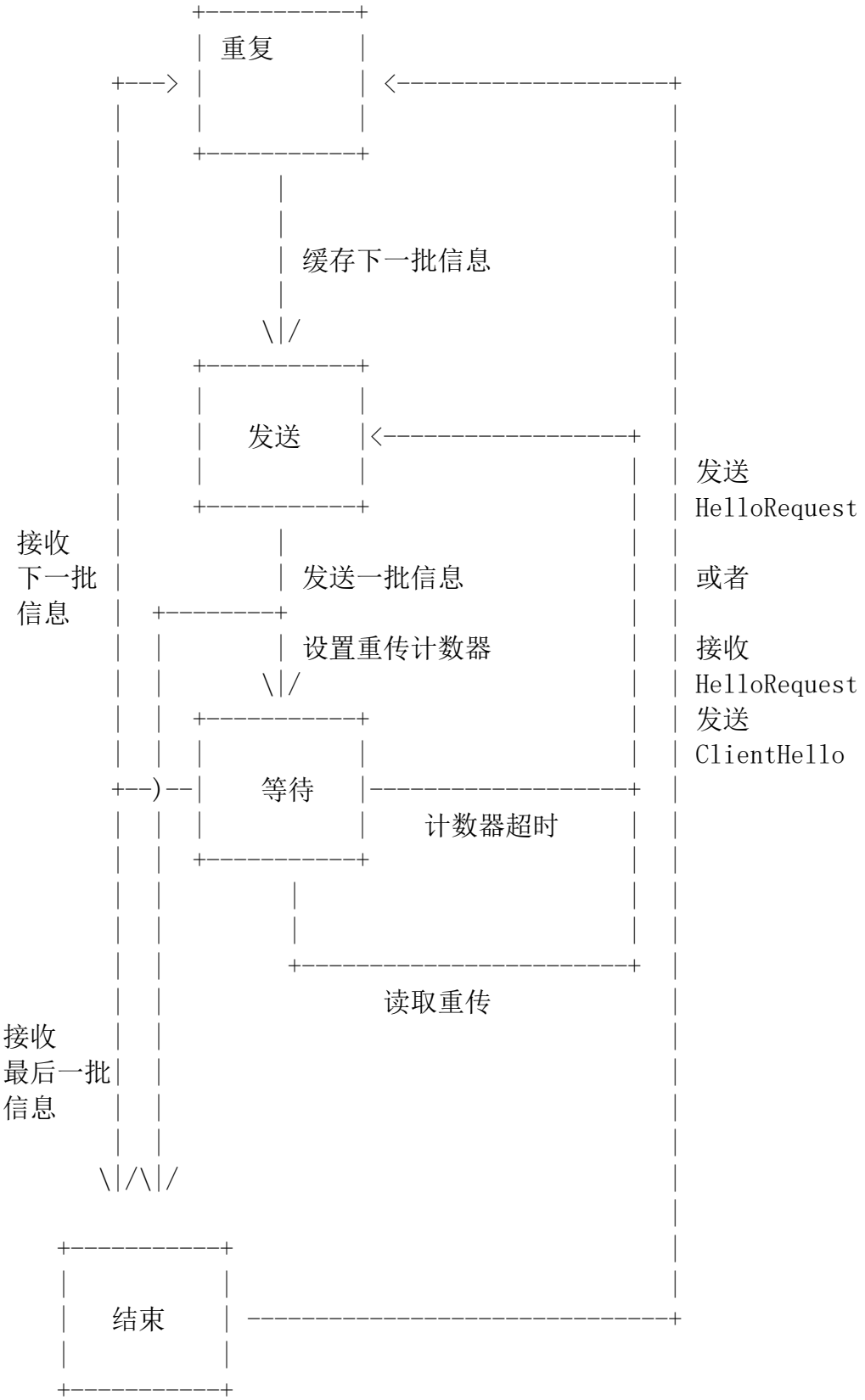

Figure 3. DTLS 超时重传状态机

状态机有三种基本状态。

在 PREPAREING 状态下，实现执行准备下一个消息所需的任何计算。 然后，它会缓冲它们以方便进行传输（首先清空缓冲区）并进入 SENDING 状态。

在 SENDING 状态下，实现传输消息的缓冲。 发送消息后，如果这是握手中的最后一批，则实现将进入 FINISHED 状态。 或者，如果实现希望接收更多消息，它会设置重新传输计时器，然后进入 WAITING 状态。

有三种方式终止 WAITING 状态:

1. 重传计数器过期：实现转换到 SENDING 状态，重传信息，重置重传计数器，返回 WAITING 状态。

2. 从对等方读取重新传输的信息：实现转换到 SENDING 状态，在那里它重新传输信息，重置重传计时器，并返回到 WAITING 状态。因为收到重复的消息可能是对等体计时器过期的结果，表明一个之前信息的一部分丢失了。

3. 接收下一批消息：如果这是最后一批消息，则将转换为 FINISHED。 如果需要发送新的一批信息,它将转换为 PREPAREING 状态。部分读取(无论是部分消息还是信息的部分）不会导致状态转换或计时器重置。

由于 DTLS 客户端发送第一条消息 （ClientHello），因此它们以 PREPAREING 状态启动。 DTLS 服务器以 WAITING 状态启动，但缓冲区为空并且没有重新传输计时器。

当服务器需要重新握手时，它会从 FINISHED 状态转换为 PREPAREING 状态以传输 HelloRequest。当客户端收到 HelloRequest 时，它会从 FINISHED 转换为 PREPAREING 以传输 ClientHello。

4.2.4.1. 计数器值

尽管计时器值是实现的选择，但计时器处理不当可能会导致严重的拥塞问题;例如，如果 DTLS 的许多实例提前超时，并且在拥塞链路上重新传输得太快。 实现应使用 1 秒的初始计时器值（RFC 2988 [RFC2988] 中定义的最小值），并在每次重新传输时将值加倍，最多不低于 RFC 2988 最大值 60 秒。 请注意，我们建议使用 1 秒计时器而不是 RFC 2988 的 3 秒默认值，以改善时间敏感型应用程序的延迟。 由于 DTLS 仅使用重新传输进行握手，而不使用数据流，因此对拥塞的影响应最小。

实现应保留当前计时器值，直到发生无丢失的传输，此时该值可能会重置为初始值。 经过长时间的空闲，不少于当前定时器值的 10 倍，实现可以将定时器重置为初始值。 可能会发生这种情况的一种情况是在大量数据传输后使用重新握手。

4.2.5.更改密码规范

与 TLS 一样，ChangeCipherSpec 消息在技术上不是握手消息，但**必须**被视为与关联的完成消息相同的外部测试的一部分，以便超时和重新传输。

4.2.6.完成消息

完成的消息具有与 TLS 中相同的格式。 但是，为了消除对分段的敏感性，**必须**将完成的 MAC 计算为每个握手消息都作为单个片段发送。请注意,在使用 cookie 交换的情况下，初始的 ClientHello 和 HelloVerifyRequest **绝不能**包含在完成的 MAC 中。

4.2.7.警报消息

请注意，警报消息根本不会重新传输，即使它们发生在握手上下文中也是如此。 但是，如果再次收到违规记录（例如，作为重新传输的握手消息），DTLS **应当**生成新的警报消息。**应当**检测对等方何时持续发送错误消息，并在检测到此类不当行为后终止本地连接状态。

4.3.新语法总结

本节包括在 TLS 1.1 和 DTLS 之间更改的数据结构的规范。

4.3.1.记录层

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

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

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

4.3.2.握手协议

```
enum {
        hello_request(0), client_hello(1), server_hello(2),
        hello_verify_request(3),                          // 新字段
        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;                               // 新字段
        uint24 fragment_offset;                           // 新字段
        uint24 fragment_length;                           // 新字段
  select (HandshakeType) {
          case hello_request: HelloRequest;
          case client_hello:  ClientHello;
          case server_hello:  ServerHello;
          case hello_verify_request: HelloVerifyRequest;  // 新字段
          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..32>;                             // 新字段
        CipherSuite cipher_suites<2..2^16-1>;
        CompressionMethod compression_methods<1..2^8-1>;
      } ClientHello;

      struct {
        ProtocolVersion server_version;
        opaque cookie<0..32>;
```

} HelloVerifyRequest;

5.安全注意事项

本文档介绍 TLS 1.1 的变体，因此大多数安全注意事项与附录 D、E 和 F 中所述的 TLS 1.1 [TLS11] 相同。

DTLS 提出的其他主要安全注意事项是拒绝服务。 DTLS 包括旨在防止拒绝服务的 Cookie 交换。 但是，不使用这种 cookie 交换的实现仍然容易受到 DoS 的攻击。 特别是,不使用 cookie 交换的 DTLS 服务器可能被用作放大攻击,即使它们本身没有遇到 DoS。因此，DTLS 服务器应使用 cookie 交换，除非有充分的理由相信放大在其环境中不会构成威胁。 客户必须准备好在每次握手时进行 cookie 交换。

6.致谢

作者要感谢 Dan Boneh，Eu-Jin Goh，Russ Housley，Constantine Sapuntzakis 和 Hovav Shacham 对 DTLS 设计的讨论和评论。 感谢 NDSS 匿名审稿人对我们关于 DTLS [DTLS]的原始 NDSS 论文的评论。 另外，感谢 Steve Kent 的反馈，这些反馈有助于澄清许多观点。 关于 PMTU 的部分是从 DCCP 规范[DCCP]中摘录的。 Pasi Eronen 对此规范进行了详细的审查。 Mark Allman，Jari Arkko，Joel Halpern，Ted Hardie 和 Allison Mankin 也对该文件提出了有用的评论。

7. IANA 注意事项

本文档使用与 TLS [TLS11] 相同的标识符空间，因此不需要新的 IANA 注册管理机构。 为 TLS 分配新标识符时，作者**必须**指定它们是否适合 DTLS。

本文档定义一个新的握手消息 hello_verify_request，其值已从 [TLS11] 中定义的 TLS 握手类型注册表中分配。 值“3”已由 IANA 分配。

8.参考资料

8.1.标准参考

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

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, August 1996.

[RFC2401] Kent, S. and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.

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

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

[TLS11] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006.

8.2. 内容参考

[AESCACHE] Bernstein, D.J., "Cache-timing attacks on AES" http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

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

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

[DNS] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987.

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

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

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

Kaufman, C., "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, "ICMP Security Failures Messages", RFC 2521, 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.

[SCTP] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, October 2000.

[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.

[WHYIPSEC] Bellovin, S., "Guidelines for Mandating the Use of IPsec", Work in Progress, October 2003.