一文读懂 DTLS 协议 转载

mb6004f7ec10a08 2021-01-19 23:30:03

文章标签 DTLS 阅读数 2038

在 WebRTC 中，为了保证媒体传输的安全性，引入了 DTLS 来对通信过程进行加密。DTLS 的作用、原理与 SSL/TLS 类似，都是为了使得原本不安全的通信过程变得安全。它们的区别点是 DTLS 适用于加密 UDP 通信过程，SSL/TLS 适用于加密 TCP 通信过程，正是由于使用的传输层协议不同，造成了它们实现上面的一些差异。

基本概念

对称密钥加密技术

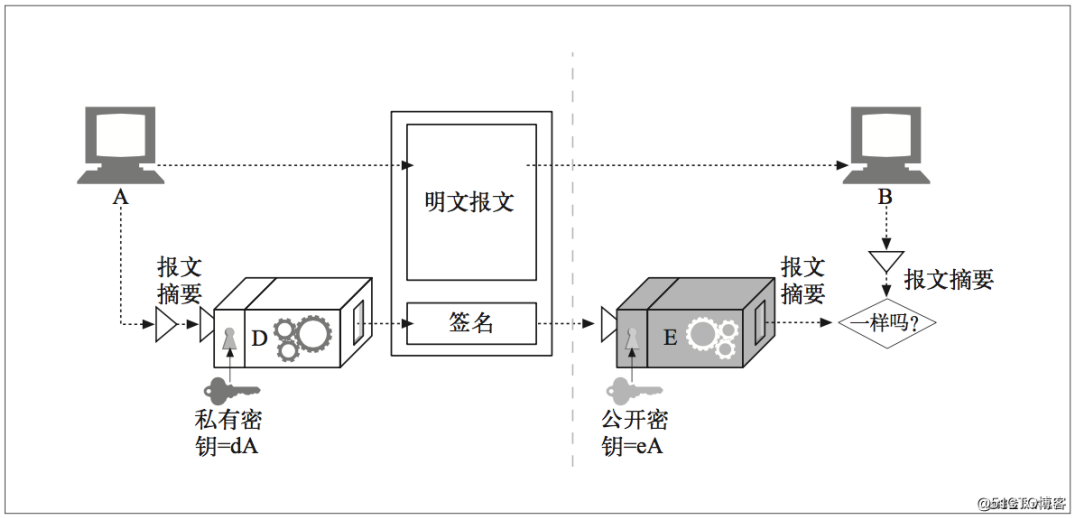
对称密钥加密的含义是加密过程和解密过程使用的是同一个密钥。常见的对称加密算法有 DES、3DES、AES、Blowfish、IDEA、RC5、RC6。

非对称加密密钥加密技术

非对称密钥加密的含义是加密过程和解密过程使用了不同的密钥，分别称为公开密钥和私有密钥。公钥是众所周知的，但是密钥只能为报文的目标主机所持有。相比对称加密技术，它的优点是不用担心密钥在通信过程中被他人窃取；缺点是需要解码速度慢，消耗更多的 CPU 资源。常见的非对称加密算法有 RSA、DSA、DH 等。

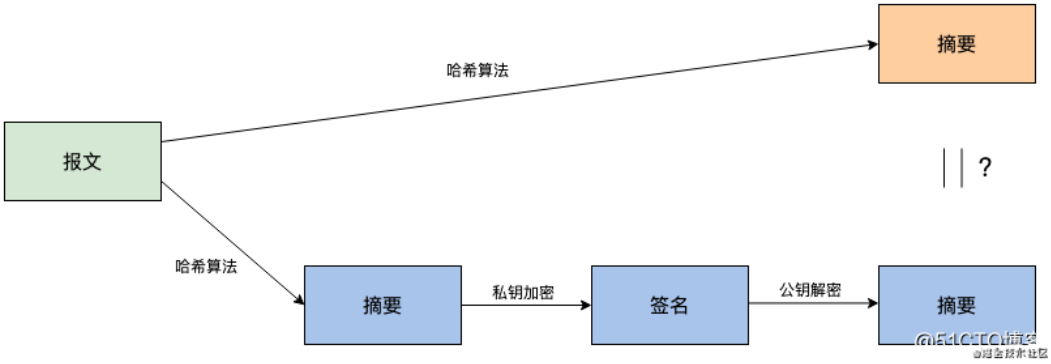
数字签名

数字签名是附加在报文上的特殊加密校验码，即所谓的校验和，其中利用了非对称加密密钥加密技术。数字签名的主要作用是防止报文被篡改，一旦报文被攻击者篡改，通过将其与校验和进行匹配，可以立刻被接收者发现。数字签名的过程如下图所示：



发送者 A 将报文摘要（报文通过 SHA-1 等哈希算法生成摘要）通过私有密钥加密生成签名，与明文报文一起发给接收者 B，接收者 B 可以通过对收到的信息进行计算后得到两份报文摘要，比较这两份报文摘要是否相等可以验证报文是否被篡改：

1. 明文报文通过使用与发送端相同的哈希算法生成摘要 1；
2. 签名通过公开密钥解密后生成摘要 2。



数字证书

数字证书是由一些公认可信的证书颁发机构签发的，不易伪造。包括如下内容：

- 证书序列号
- 证书签名算法
- 证书颁发者
- 有效期
- 公开密钥
- 证书签发机构的数字签名

数字证书可以用于接收者验证对端的身份。接收者（例如浏览器）收到某个对端（例如 Web 服务器）的证书时，会对签名颁发机构的数字签名进行检查，一般来说，接收者事先就会预先安装很多常用的签名颁发机构的证书（含有公开密钥），利用预先的公开密钥可以对签名进行验证。

参考资料

zhangbuhuai.com/post/https-…

SSL/TLS 协议

简介

要了解 DTLS，首先从我们比较熟悉的 SSL/TLS 开始讲起。SSL（Secure Socket Layer） 和 TLS（Transport Layer Security） 简单理解就是同一件东西的两个演进阶段，同样都是在应用层和传输层之间加入的安全层，最早的时候这个安全层叫做 SSL，由 Netscape 公司推出，后来被 IETF 组织标准化并称之为 TLS。SSL/TLS 的作用是为了解决互联网通信中存在的三种风险：

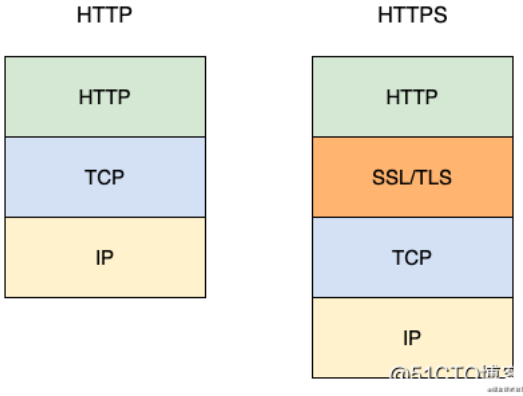
1. 窃听风险：第三方可以获知通信内容；
2. 篡改风险：第三方可以修改通信内容；
3. 冒充风险：第三方可以冒充他人身份参与通信。

SSL/TLS 协议能够做到以下这几点，从而解决上述的三种风险：

1. 所有信息通过加密传播，第三方无法窃听；
2. 具有数据签名及校验机制，一旦被篡改，通信双方立刻可以发现；
3. 具有身份证书，防止其他人冒充。

协议栈

SSL/TLS 建立在 TCP 传输层上，最常使用 SSL/TLS 的场景是在 HTTPS 中，通过在 HTTP 和 TCP 中加了一层 SSL/TLS，使得不安全的 HTTP 成为了安全的 HTTPS。协议栈如下图所示：

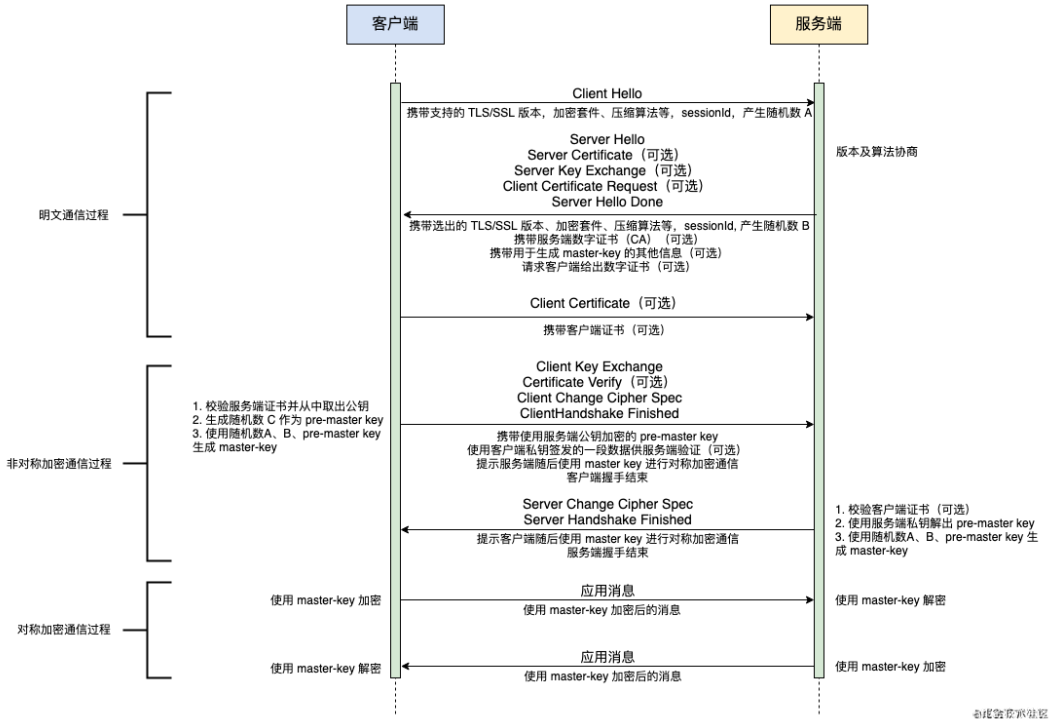


SSL/TLS 对于 TCP 传输层的加密是通过动态密钥对数据进行加密实现的，而动态密钥通过握手流程协商制定。因此在 SSL/TLS 握手过程中需要协商的信息包括：

1. 协议版本号；
2. 加密算法，包括非对称加密算法、动态密钥算法；
3. 数字证书，传输双方通过交换证书及签名校验来验证对方身份；
4. 动态密钥，由于非对称加密对性能消耗较大，因此主要的通信过程都是使用动态密钥进行对称加密的；对称加密使用的动态密钥则在握手过程中通过非对称加密来传输。

握手过程

以 TLS 1.2 为例：



握手过程如上图所示，大体来说分成三个过程：明文通信过程、非对成加密通信过程、对称加密通信过程；

1. 明文通信过程：在通信两端首次向对方发送 Hello 消息时，由于双方都没有协商好要使用哪种加密方式，因此这个过程中的消息都是使用明文进行发送的。
   a. Client Hello：客户端首先向服务端发起握手，在握手消息中告诉对方自己支持的 SSL/TLS 版本、加密套件（包括非对称加密时使用的算法与、非对称加密时使用的算法、产生密钥的伪随机函数 PRF）与数据压缩算法（TLS1.3之后就已经没有这个字段）等；还会携带一个 Session ID，因为握手流程的开销比较大，使用 Session ID 可以在下一次与 TLS 握手的过程跳过后续繁琐的握手流程，重用之前的握手结果（如版本号、加密算法套件、master-key 等）；并产生一个随机数 A，也告诉给对方；
   b. Server Hello：服务端响应一个 Server Hello 消息，携带协商出来的 TLS/SSL 版本号、加密套件和数据压缩算法，如果服务端同意客户端重用上次的会话，就返回一个相同的 Session ID，否则就填入一个全新的 Session ID；

c. Server Certificate（可选）：携带服务端数字证书（CA）以验证服务端身份，里面携带了服务端非对称加密所使用的公钥；这步虽然是可选的，但是一般来说客户端都会要求验证服务端的身份，在大多数情况下这步都会执行；

d. Server Key Exchange（可选）：在使用某些非对称加密算法（例如 DH 算法）的情况下，Server Certificate 里的信息是不足够的，或者 Server Certificate 在某些通信过程中直接被省略了（没有验证服务端身份），需要 Server Key Exchange 里的额外信息来帮助客户端生成 pre-master key；

e. Client Sertificate Request（可选）：在有些安全性要求高的场景，例如银行支付等，不仅需要验证服务端的身份，还需要验证客户端的身份，这时候服务端就会要求客户端提供客户端的身份证书；

f. Server Hello Done：表明 Server Hello 结束；

g. Client Certificate（可选）：如果服务端要求客户端提供数字证书以验证身份，则客户端发送自己的身份证书给服务端；

2. 非对称加密通信过程：由于非对称加密通信的性能较差，在实际的通信过程中其实使用的是对称加密通信，为了保证对称加密通信过程的安全性，也就是需要避免对称加密密钥被窃取，这个密钥在协商过程中使用非对称加密来进行加密。

   a. Client Key Exchange：客户端在验证服务端的身份证书后，会取出其中的服务端公钥，产生一个随机数 C，作为 pre-master key，在本地使用之前的随机数 A、B 和这次生成的 C 共同生成对称加密密钥 master-key；使用服务端公钥对 pre-master key 加密后发送给服务端；

   b. Certificate Verify（可选）：如果服务端要求客户端提供客户端证书，那么客户端在发送 Client Key Exchange 之后必须马上发送 Certificate Verify，其中的内容是客户端使用自己的私钥加密的一段数据，提供给服务端用客户端的公钥来进行解密验证。之所以需要这一步是为了确保客户端发送的证书确实是它自己的证书；

   c. Client Change Cipher Spec：提示服务端随后使用 master key 来进行对称加密通信；

   d. Client Handshake Finished: 表明客户端侧 SSL/TLS 握手结束；

   e. Server Change Cipher Spec：提示客户端随后使用 master key 来进行对称加密通信；

   f. Server Handshake Finished：表明服务端侧 SSL/TLS 握手结束；

3. 对称加密通信过程：通过上述握手过程协商出对称加密算法及使用的对称加密密钥之后，随后的通信过程，也就是实际的应用通信过程，都使用的是对称加密。

握手消息格式

SSL/TLS 的握手消息格式如下所示，消息类型已经在上文握手过程中分别进行了解释；结构体中包含 消息类型、消息长度、消息体。

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    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;
     select (HandshakeType) {
        case hello_request:       HelloRequest;
        case client_hello:        ClientHello;
        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;复制代码
```

每种消息都会有自己独立的消息体，消息体中的内容就是握手过程中提到的消息中需要携带的一些信息，以 Client Hello 为例，其中需要包含 SSL/TLS 版本号、随机数、Session ID、可选的加密套件、可选的压缩算法：

```
struct {
        ProtocolVersion client_version;
```

```
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;复制代码
```

参考资料

RFC：

tools.ietf.org/html/rfc224… TLS 1.0

tools.ietf.org/html/rfc434… TLS 1.1

tools.ietf.org/html/rfc524… TLS 1.2（本文主要参考）

tools.ietf.org/html/rfc844… TLS 1.3

论坛资料：

segmentfault.com/a/119000002…

halfrost.com/https_tls1-…

DTLS 协议

简介

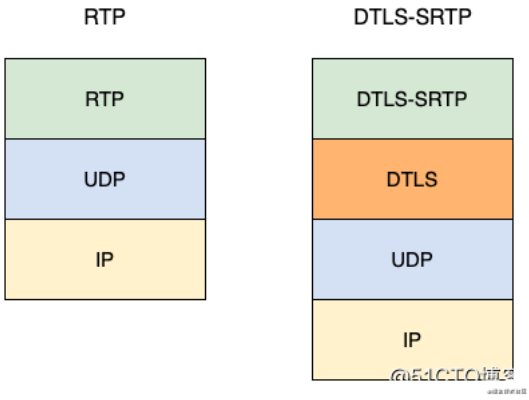
DTLS 的全称为 Datagram Transport Layer Security，从名字上就可以看出它和 TLS 的区别就在于多了一个“Datagram”，因为我们把使用 UDP 传输的报文叫做 “Datagram”，也就是 DTLS 是适用于 UDP 传输过程的加密协议。 DTLS 在设计上尽可能复用 TLS 现有的代码，并做一些小的修改来适配 UDP 传输。DTLS 与 TLS 具备了同样的安全机制和防护等级，同样能够防止消息窃听、篡改，以及身份冒充等问题。在版本上，DTLS 和 TLS 也有一定的对应关系，如下：

- DTLS 1.0 对应 TLS 1.1
- DTLS 1.2 对应 TLS 1.2
- DTLS 1.3 对应 TLS 1.3

没有 DTLS 1.1 应当是为了和 TLS 版本号相一致。

协议栈

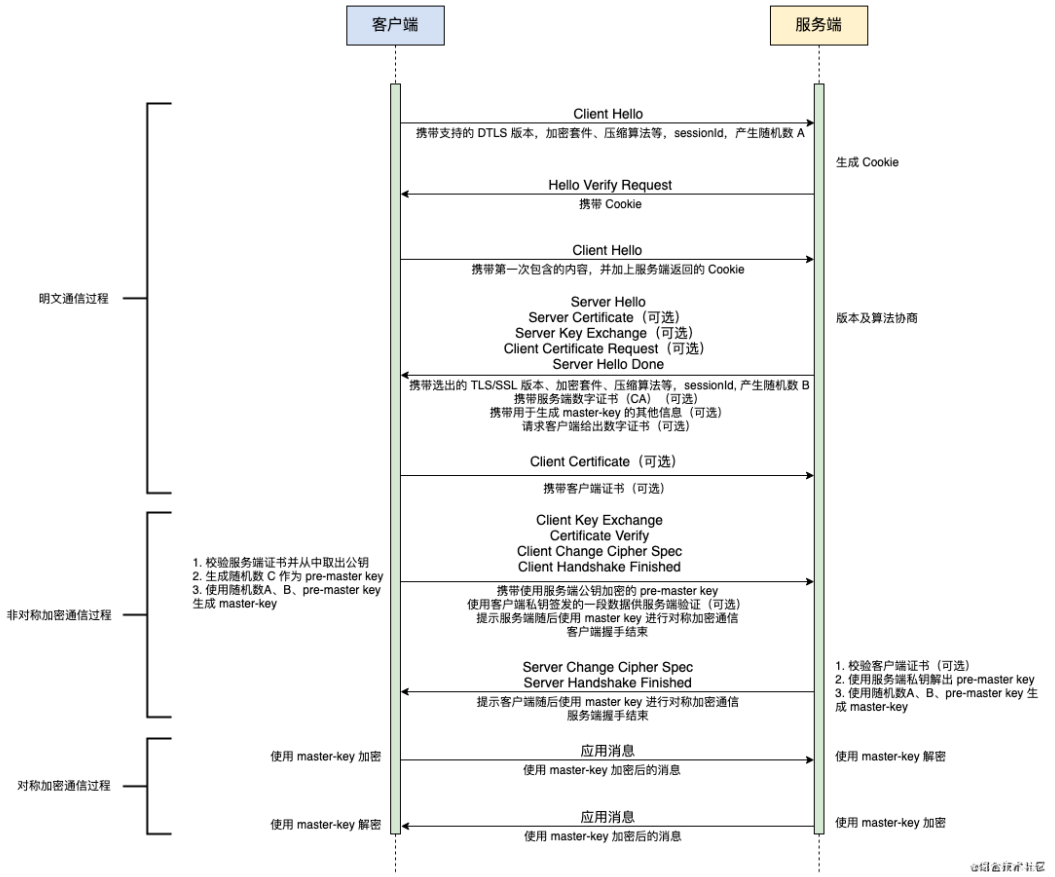
在 WebRTC 中，通过引入 DTLS 对 RTP 进行加密，使得媒体通信变得安全。通过 DTLS 协商出加密密钥之后，RTP 也需要升级为 SRTP，通过密钥加密后进行通信。协议栈如下图所示：



握手过程

TLS 1.2 及之前都没有尝试解决 DoS 攻击的问题，直到 TLS 1.3 才通过加入了 HelloRetryRequest 和 Cookie 来解决 DoS 攻击的问题（并且在 TLS 1.3 的 RFC 中提到主要用于非面向连接的通道，也就是 UDP 连接）。相对 TCP 来说，UDP 连接对 DoS 攻击更加敏感，因此 DTLS 在 1.0 版本就加入了 HelloVerifyRequest 和 Cookie，用于服务端对客户端的二次校验，避免 DoS 攻击。

同样以 DTLS 1.2 举例（TLS 1.3 和 DTLS 1.3 的流程已经很接近了），相比 TLS 1.2，DTLS 1.2 大部分步骤都是一样的，只是在服务端多了一步 HelloVerifyRequest，客户端因此也多了第二次的 ClientHello，如下图所示：



服务端在首次收到客户端发送的 Client Hello 之后，只会生成一个 Cookie，不进行任何其他的操作，并给客户端发送 HelloVerifyRequest 消息，带上这个 Cookie。只有当客户端重新发送一次 Client Hello，并带上服务端发送的这个 Cookie 后，服务端才会继续握手过程。

握手消息格式

DTLS 的握手消息格式如下所示，可以看到相比 SSL/TLS 的握手协议，DTLS 在消息类型中多了 HelloVerifyRequest 这种消息（在握手过程中介绍过），在结构体中多了 message_seq、fragment_offset、fragment_length 三个字段。

SSL/TLS 基于 TCP，因此不需要操心重放、乱序、丢包的问题，可靠传输由 TCP 做了保证；而 DTLS 基于 UDP，UDP 是一种尽力而为的协议，因此 DTLS 需要自己处理重放、乱序、丢包的问题。DTLS 在复用大部分 TLS 的基础上做了一些小改动，在握手消息中增加了三个字段 message_seq、fragment_offset、fragment_length 三个字段，具体的功能在下一节中讲述。

```
    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;复制代码
```

每种消息同样都会有自己独立的消息体，消息体中的内容就是握手过程中提到的消息中需要携带的一些信息，大部分的消息体内容与 SSL/TLS 相同，区别有两点：

1. 多了 HelloVerifyRequest 消息，需要携带 Cookie，消息体如下所示：

```
struct {
      ProtocolVersion server_version;
      opaque cookie<0..32>;
    } HelloVerifyRequest;复制代码
```

1. ClientHello 多了 Cookie 字段，因为第二次 ClientHello 需要携带 Cookie 信息：

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

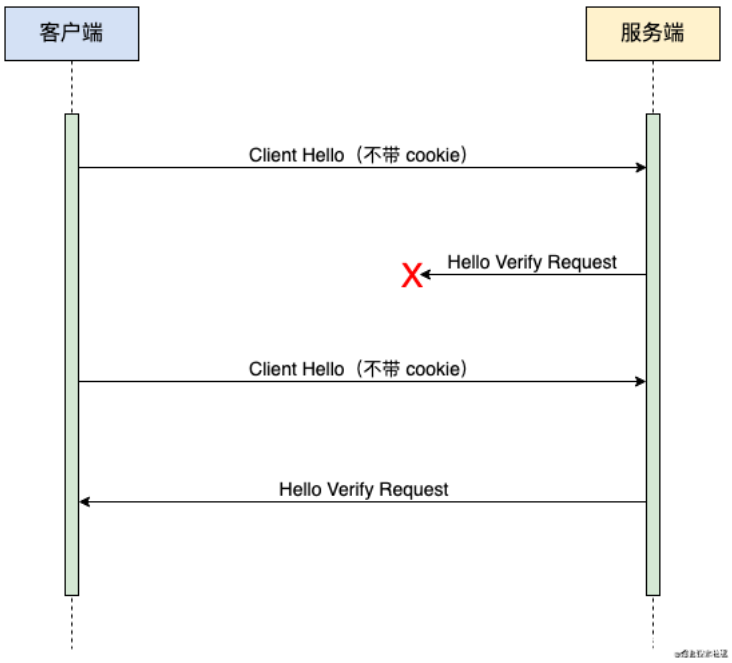
与 SSL/TLS 在实现上的区别

在上文的握手过程和握手协议中已经讲述了一些 DTLS 与 SSL/TLS 的区别。此外，DTLS 针对重复、乱序、丢包等问题增加了一些防护机制。

握手防护机制

重传

TCP 天然的重传机制保证了消息不会丢失，而 UDP 对此没有任何保证。因此 DTLS 额外增加了超时重传机制来确定握手消息到达，流程如下：



以握手的第一阶段举例，客户端发送 Client Hello（不带 Cookie，区别于握手流程中的第二次 Client Hello）之后，启动一个定时器，等待服务端返回 HelloVerifyRequest，如果超过了定时器时间客户端还没有收到 HelloVerifyRequest，那么客户端就会知道要么是 Client Hello 消息丢了要么是 Hello Verify Request 消息丢了，客户端就会再次发

送相同的 Client Hello 消息，即使服务端确实发送了 Hello Verify Request 还是收到了 Client Hello 消息，它也知道是需要重传，并再次发送 Hello Verify Request 消息，同样地，服务端也会启动定时器来等待吓一条消息。

序列号

TCP 消息中自带了序列号 seq，并处理了乱序的问题，而 UDP 对乱序问题也没有任何保证。为了保证握手消息的有序性，DTLS 在握手报文中增加了 message_seq 字段便于接收方处理乱序消息。接收方直接处理属于当前步骤的消息，提供一个缓存队列来缓存提前到达的消息。

消息重放检测

消息重放也是 DoS 攻击的一种，攻击者可以截取发送者发送的数据，并直接原封不动地发给接收方，来达到欺骗接收方的目的。 DTLS 增加了类似 IPsec AH/ESP 的消息重放检测，使用一个 bitmap 滑动窗口来接收消息，结合消息本身的序号，bitmap 可以判断该消息是否是太老的消息，是的话则直接抛弃。这个功能在 DTLS 中是可选的，因为某些 UDP 报文的重复只是单纯因为网络错误。

报文大小限制

TCP 面向字节流，而 UDP 面向报文。因此 TCP 会自动将报文进行拆分和组装，无需上层操心；而 UDP 报文如果超过了 MTU（链路层的最大传输单元） 限制，会在 IP 层被强制分片，使得每一片大小都小于 MTU，接收方 IP 层需要进行数据报的重组，这样就会多做许多工作，更麻烦的地方在于只要其中一片丢失就会造成重组失败，造成整个 UDP 报文丢失。 因此 DTLS 直接在 UDP 之上就对握手消息做了分段，握手报文中的 fragment_offset 和 fragment_length 就是为了这个目的，分别代表这段报文相对消息起始的偏移量以及这段报文的长度。

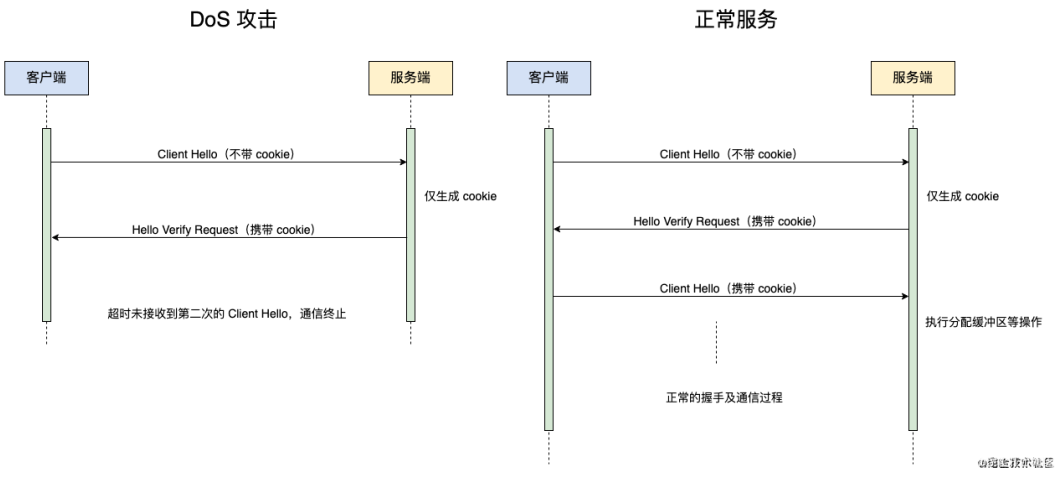
Cookie

在上文的握手过程中 TLS 1.2 及之前都没有尝试解决 DoS 攻击的问题，直到 TLS1.3 才通过加入了 HelloRetryRequest 和 Cookie 来解决 DoS 攻击的问题（并且在 TLS 1.3 的 RFC 中提到主要用于非面向连接的通道，也就是 UDP 连接）。相对 TCP 来说，UDP 连接对 DoS 攻击更加敏感，因为 TCP 可以使用 SYN Cookie 的机制来防范 DoS 攻击。 如果在 UDP 上直接使用 TLS 的握手方式，就可能发生以下两种情况：

1. 攻击者可以通过发送一系列握手启动请求来消耗服务器上的资源，例如服务端可能会为此分配缓冲区，并且加密过程也是一个非常消耗 CPU 资源的操作；
2. 攻击者可以伪造受害客户端的 IP 地址向服务端发起 DTLS 握手，迫使服务端发送 Certificate 消息给受害客户端，上文提到过 Certificate 携带了很多信息，包括数字证书等，所以这个消息会非常大，使得受害客户端不得不接受大量无用消息。

因此 DTLS 在 1.0 版本就加入了 HelloVerifyRequest 和 Cookie，用于服务端对客户端的二次校验，避免 DoS 攻击。具体实现方式如下：

1. 当客户端首次给服务端发送 Client Hello 时，服务端只会生成一个 Cookie 并通过 HelloVerifyRequest 发送给客户端，不会执行分配缓冲区等操作，直到收到带上相同 Cookie 的 Client Hello 才会继续握手，可以使得伪造 IP 的攻击难以实现（使用真实 IP 的 DoS 攻击无能为力）；
2. HelloVerifyRequest 足够小，即使服务端被攻击者当枪使来攻击其他机器，也不会造成大量数据发送。



加密方式

由于 SSL/TLS 依赖 TCP，所以 SSL/TLS 有如下几个特性：

1. SSL/TLS 不能独立解密单个封包，SSL/TLS 对于封包的认证需要序号作为输入，在 SSL/TLS 中并未直接传递序号，因为 TCP 是可靠的，所以 TLS 的两端各自维护自身的收发序号；
2. SSL/TLS 的某些加密算法不是独立解密的，需要依赖上个封包，典型的就是 RC4 流加密算法。

由于 UDP 本身的不可靠，为了解决上面提到的第一个问题，DTLS 在每条记录中显式携带的序号作为解码的输入；而对于第二个问题，DTLS 的现状是不能支持 RC4 等流式加密算法。