

02 | 为HTTP穿上盔甲：HTTPS

2019-09-13 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 19:11 大小 13.18M



你好，我是四火。

在上一讲中，我介绍了互联网最重要的 HTTP 协议。可是随着互联网的发展，你会发现 HTTP 越来越无法满足复杂的需求，比如数据加密传输的安全性需求，再比如服务器消息即时推送的交互模式的需求，而这些不适性是由 HTTP 的基本特性所造成的。

因此，我们需要在传统 HTTP 领域以外开疆拓土，这就包括要引入其它的网络协议，或增强、或填补 HTTP 协议所不擅长的空白领域，这也是今天这一讲和下一讲的核心内容。今天我们重点学习 SSL/TLS，看看它是如何让 HTTP 传输变得安全可靠的。

HTTP, SSL/TLS 和 HTTPS

在一开始的时候，HTTP 的设计者并没有把专门的加密安全传输放进协议设计里面。因此单独使用 HTTP 进行明文的数据传输，一定存在着许多的安全问题。比方说，现在有一份数据需要使用 HTTP 协议从客户端 A 发送到服务端 B，而第三方 C 尝试来做点坏事，于是就可能产生如下四大类安全问题：

Interception：拦截。传输的消息可以被中间人 C 截获，并泄露数据。

Spoofing：伪装。A 和 B 都可能被 C 冒名顶替，因此消息传输变成了 C 发送到 B，或者 A 发送到 C。

Falsification：篡改。C 改写了传输的消息，因此 B 收到了一条被篡改的消息而不知情。

Repudiation：否认。这一类没有 C 什么事，而是由于 A 或 B 不安好心。A 把消息成功发送了，但之后 A 否认这件事发生过；或者 B 其实收到了消息，但是否认他收到过。

但是，与其重新设计一套安全传输方案，倒不如发挥一点拿来主义的精神，把已有的和成熟的安全协议直接拿过来套用，最好它位于呈现层（Presentation Layer），因此正好加塞在 HTTP 所在的应用层（Application Layer）下面，**这样这个过程对于 HTTP 本身透明，也不影响原本 HTTP 以下的协议（例如 TCP）。**

这个协议就是 SSL/TLS，它使得上面四大问题中，和传输本身密切相关的前三大问题都可以得到解决（第四个问题还需要引入数字签名来解决）。于是，HTTP 摇身一变成了 HTTPS：

HTTP + SSL/TLS = HTTPS


这里涉及到的两个安全协议，SSL 和 TLS，下面简要说明下二者关系。

SSL 指的是 Secure Socket Layer，而 TLS 指的是 Transport Layer Security，事实上，一开始只有 SSL，但是在 3.0 版本之后，SSL 被标准化并通过 [RFC 2246](#) 以 SSL 为基础建立了 TLS 的第一个版本，因此可以简单地认为 SSL 和 TLS 是具备父子衍生关系的同一类安全协议。

动手捕获 TLS 报文


介绍了最基本的概念，我们再来看看 HTTPS 是怎样安全工作，让客户端和服务端相互信任的，TLS 连接又是怎样建立起来的。还记得上一讲的选修课堂吗？我们学了怎样抓包。今天我们就能让所学派上用场！自己动手，我们抓 TLS 连接握手的报文来分析。

命令行执行抓包命令，指明要抓 <https://www.google.com> 的包（当然，你也可以使用其他 HTTPS 网站地址），注意 HTTPS 的默认端口是 443（-i 指定的 interface 可能因为不同的操作系统有所区别，在我的 Mac 上是 en0）：

 复制代码


```
1 sudo tcpdump -i en0 -v 'host www.google.com and port 443' -w https.cap
```

再新建一个命令行窗口，使用 curl 命令来访问 Google 主页：

 复制代码

```
1 curl https://www.google.com
```

于是在看到类似如下抓包后 CTRL + C 停止：

 复制代码

```
1 tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
2 ^C49 packets captured
3 719 packets received by filter
4 0 packets dropped by kernel
```

接着使用 Wireshark 打开刚才抓的 https.cap，在 filter 中输入 tls，得到如下请求和响应报文：

tls						
No.	Time	Source	Destination	Protocol	Length	Info
4	0.026639	2601:600:...	2607:f8b0:...	TLSv1.2	603	Client Hello
6	0.048770	2607:f8b0:...	2601:600:...	TLSv1.2	1294	Server Hello
7	0.049066	2607:f8b0:...	2601:600:...	TLSv1.2	1258	Certificate, Server Key Exchange, Server Hello Done
9	0.060918	2601:600:...	2607:f8b0:...	TLSv1.2	212	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
10	0.071869	2607:f8b0:...	2601:600:...	TLSv1.2	137	Change Cipher Spec, Encrypted Handshake Message

可以看到，这里有五个重要的握手消息，在它们之后的所有消息都是用于承载实际数据的“Application Data”了。握手的过程略复杂，接下来我会尽可能用通俗的语言把最主要的流程讲清楚。

对称性和非对称性加密

这里我先介绍两个概念，“对称性加密”和“非对称性加密”，这是学习后面内容的重要基础。

对称性加密（Symmetric Cryptography），指的是加密和解密使用相同的密钥。这种方式相对简单，加密解密速度快，但是由于加密和解密需要使用相同的密钥，如何安全地传递密钥，往往成为一个难题。

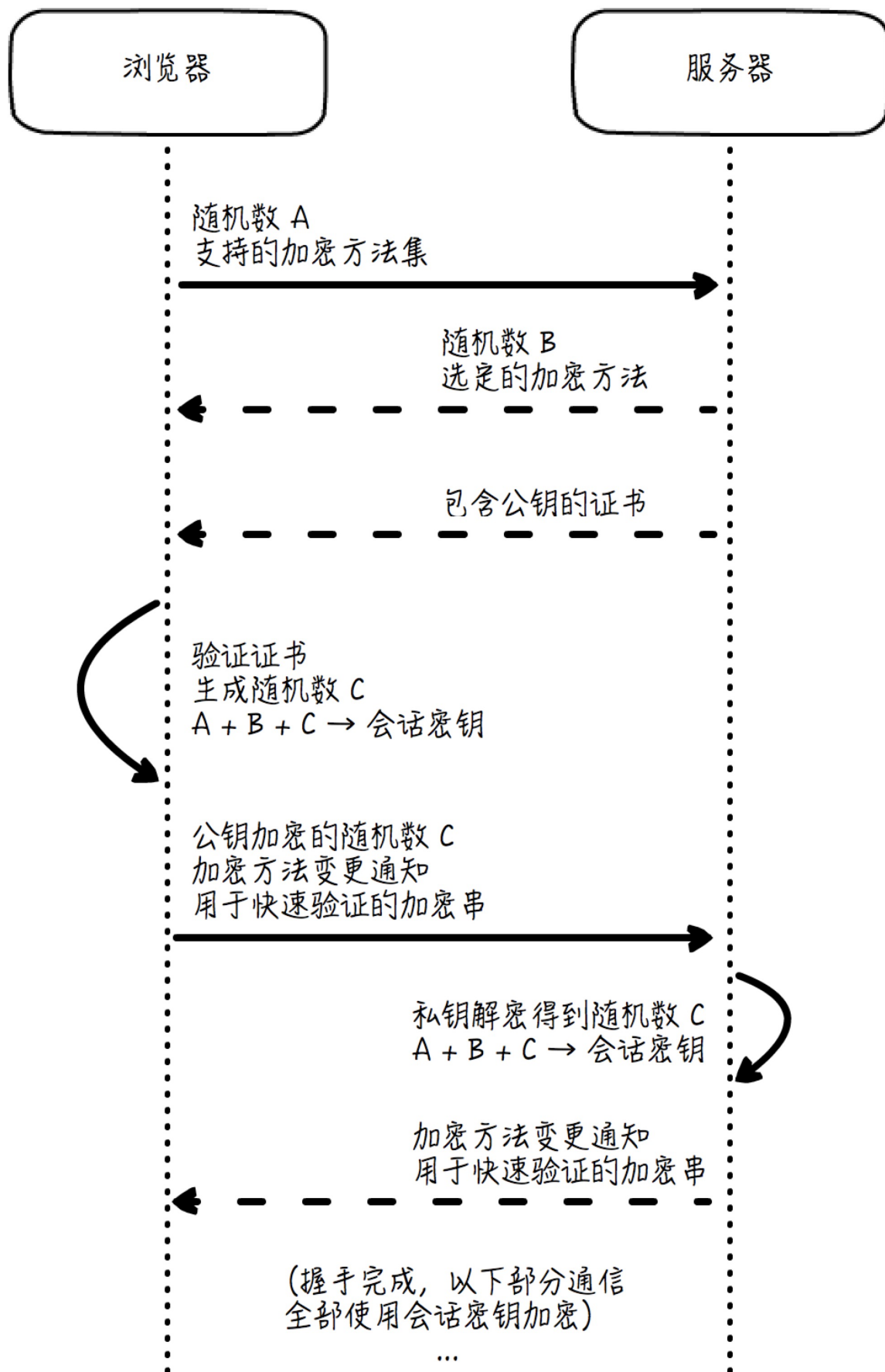
非对称性加密（Asymmetric Cryptography），指的是数据加密和解密需要使用不同的密钥。通常一个被称为公钥（Public Key），另一个被称为私钥（Private Key），二者一般同时生成，但是**公钥往往可以公开和传播，而私钥不能。经过公钥加密的数据，需要用私钥才能解密**；反之亦然。这种方法较为复杂，且性能较差，好处就是由于加密和解密的密钥具有相对独立性，公钥可以放心地传播出去，不用担心安全性问题。

原始数据 + 公钥 → 加密数据

加密数据 + 私钥 → 原始数据

TLS 连接建立原理

有了上述基础，下面我们就可以结合图示，看看整个连接建立的握手过程了。



Step 1: Client Hello. 客户端很有礼貌，先向服务端打了个招呼，并携带以下信息：

客户端产生的随机数 A;

客户端支持的加密方法列表。

Step 2: Server Hello. 服务端也很有礼貌，向客户端回了个招呼：

服务端产生的随机数 B;

服务端根据客户端的支持情况确定出的加密方法组合（Cipher Suite）。

Step 3: Certificate, Server Key Exchange, Server Hello Done. 服务端在招呼之后也紧跟着告知：

Certificate，证书信息，证书包含了服务端生成的公钥。这个公钥有什么用呢？别急，后面会说到。

客户端收到消息后，验证确认证书真实有效，那么这个证书里面的公钥也就是可信的了。

接着客户端再生成一个随机数 C（Pre-master Secret），于是现在共有随机数 A、B 和 C，根据约好的加密方法组合，三者可生成新的密钥 X（Master Secret），而由 X 可继续生成真正用于后续数据进行加密和解密的对称密钥。因为它是在本次 TLS 会话中生成的，所以也被称为会话密钥（Session Secret）。简言之：

客户端随机数 A + 服务端随机数 B + 客户端 Pre-master Secret C → 会话密钥

需要注意的是，实际这个 Pre-master Secret 的生成方法不是固定的，而会根据加密的具体算法不同而不同：

上述我介绍的是传统 RSA 方式，即 Pre-master Secret 由客户端独立生成，加密后再通过 Client Key Exchange 发回服务端。

还有一种是 [ECDHE](#) 方式，这种方式下无论在客户端还是服务端，Pre-master Secret 需要通过 Client Key Exchange 和 Server Key Exchange 两者承载的参数联合生成。

Step 4: Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message. 接着客户端告诉服务端：

Client Key Exchange，本质上它就是上面说的这个 C，但使用了服务端通过证书发来的公钥加密；

Change Cipher Spec，客户端同意正式启用约好的加密方法和密钥了，后面的数据传输全部都使用密钥 X 来加密；

Encrypted Handshake Message，快速验证：这是客户端对于整个对话进行摘要并加密得到的串，如果经过服务端解密，和原串相等，就证明整个握手过程是成功的。

服务端收到消息后，用自己私钥解密上面的 Client Key Exchange，得到了 C，这样它和客户端一样，也得到了 A、B 和 C，继而到 X，以及最终的会话密钥。

于是，客户端和服务端都得到了能够加密解密传输数据的对称密钥——会话密钥。

因此，我们可以看到：**TLS 是通过非对称加密技术来保证握手过程中的可靠性（公钥加密，私钥解密），再通过对称加密技术来保证数据传输过程中的可靠性的。**

这种通过较严格、较复杂的方式建立起消息交换渠道，再通过相对简单且性能更高的方式来完成主体的数据传输，并且前者具有长效性（即公钥和私钥相对稳定），后者具有一过性（密钥是临时生成的），这样的模式，我们还将在全栈的知识体系中，继续见到。

Step 5: Change Cipher Spec, Encrypted Handshake Message. 服务端最后也告知客户端：

Change Cipher Spec，服务端也同意要正式启用约好的加密方法和密钥，后面的数据传输全部都使用 X 来加密。

Encrypted Handshake Message，快速验证：这是服务端对于整个对话进行摘要并加密得到的串，如果经过客户端解密，和原串相等，就证明整个握手过程是成功的。

总结思考

今天我们了解了关于数据传输的四大类安全问题，了解了 HTTPS 和 SSL/TLS 的概念和它们之间的关系，还通过自己动手抓包的方式，详细学习了 TLS 连接建立的步骤。

TLS 连接的步骤是今天的重点，也是比较难理解的部分，希望你能牢牢地掌握它。现在就来检验一下今天的学习成果吧！请你思考这样两个问题：

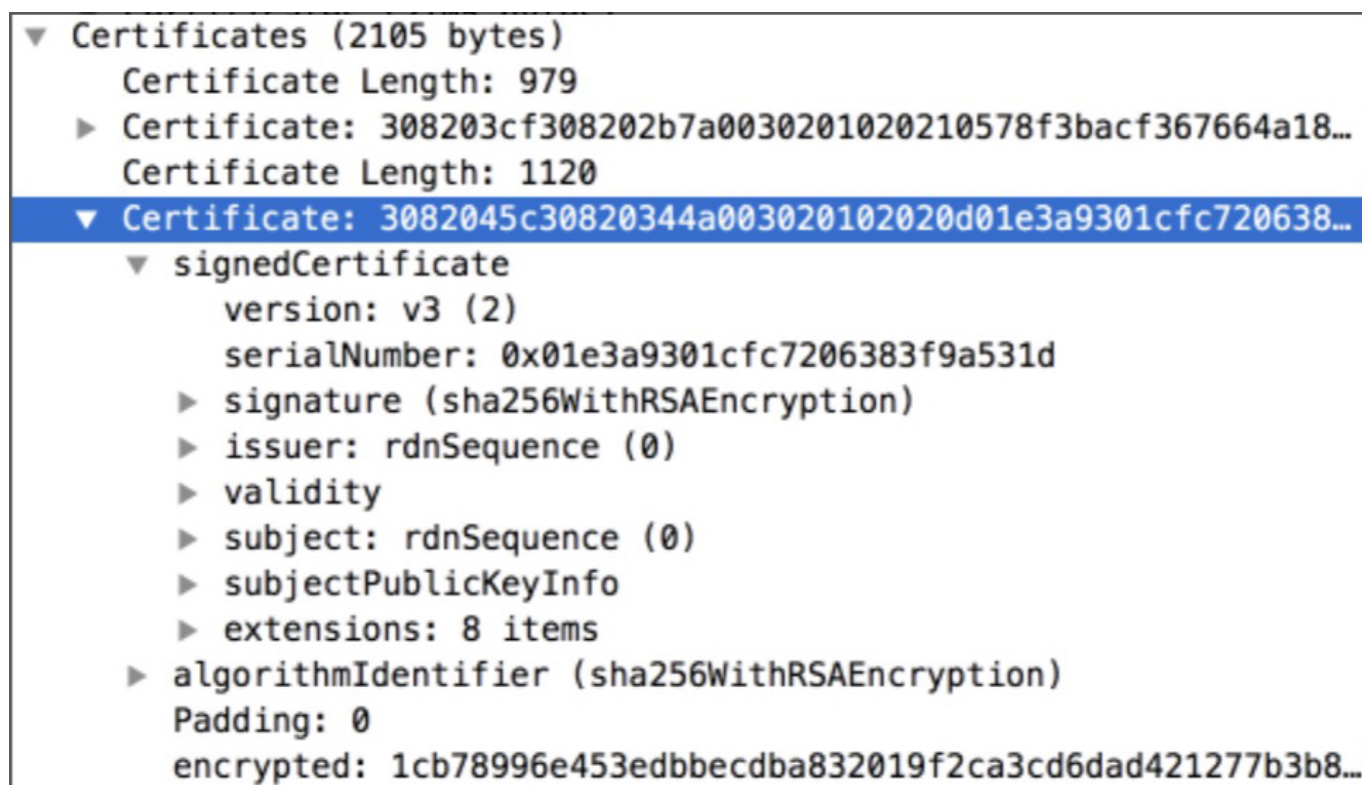
有位程序员朋友注意到，自己在使用在线支付功能时，网站访问是使用 HTTPS 加密的，因此他觉得，支付的过程中是不可能出现安全问题的，你觉得这种想法对吗？

在介绍 TLS/SSL 连接建立的过程当中，我提到了，握手过程是使用非对称加密实现的，而真正后续的数据传输部分却是由对称加密实现的。为什么要这么麻烦，全部都使用对称或非对称加密一种不好吗？

你能回答上面的问题吗？如果可以，我相信你已经理解了 HTTPS 安全机制建立的原理。

选修课堂：证书有效验证的原理

在讲解“握手过程”的 step 3 时，我提到了客户端在收到服务端发送过来的证书时，需要校证书的有效性。这个过程其实也是至关重要的，因为只有确认了证书的有效性，客户端才能放心地使用其中的公钥。如果你对它的理解比较模糊，那就一定要看看今天的选修课堂了。



这就是我们抓包中，服务器发来的证书部分的截图。我们可以看到，这不是单个证书，而是一个证书链，包含了两个证书，每个证书都包含版本、发布机构、有效期、数字签名等基本内容，以及一个公钥。实际上，这两个服务端传回来的证书，和浏览器内置的根证书联合起来，组成了一个单向、完整的证书链：



上图中的第三行，就是携带着服务器公钥的证书，它是从证书发布机构（CA, Certificate Authority）申请得来的，也就是图中第二行的 GTS CA 101。证书在申请的时候，我们提到的服务器公钥就已经是该证书的一部分了，因此我们才说，如果证书是有效的，那么它携带的公钥就是有效的。

在当时申请的时候，**证书发布机构对证书做摘要生成指纹，并使用它自己的私钥为该指纹加密，生成数字签名（Digital Signature）**，而这个数字签名也随证书一起发布。这个发布机构的私钥是它内部自己管理的，不会外泄。

指纹 + 私钥 → 数字签名

验证过程则正好是发布过程的反向，即在客户端要对这个被检测证书做两件事：

对它用指定算法进行摘要，得到指纹 P1；

使用证书发布机构的公钥对它的数字签名进行解密，得到指纹 P2。

数字签名 + 公钥 → 指纹

如果 P1 和 P2 一致，就说明证书未被篡改过，也说明这个服务端发来的证书是真实有效的，而不是仿冒的。

好，问题来了，证书发布机构使用非对称性加密和数字签名保证了证书的有效性，那么谁来保证证书发布机构的有效性？

答案就是它的上一级证书发布机构。

CA 是分级管理的，每一级 CA 都根据上述同样的原理，由它的上一级 CA 来加密证书和生成数字签名，来保证其真实性，从而形成一个单向的信任链。同时，标志着最高级别 CA 的根证书数量非常少，且一般在浏览器或操作系统安装的时候就被预置在里面了，因此它们是被我们完全信任的，这就使得真实性的鉴别递归有了最终出口。也就是说，递归自下而上验

证的过程，如果一直正确，直至抵达了顶端——浏览器内置的根证书，就说明服务端送过来的证书是安全有效的。

总结一下今天选修课堂的内容。证书有效性的验证，需要使用依赖于证书发布机构的公钥去解密被检测证书的数字签名，如果顺利解密，并且得到的指纹和被检测证书做摘要得到的指纹一致，就说明证书真实有效。

扩展阅读

[HOW HTTPS WORKS](#)：漫画版介绍 HTTPS 前前后后，很有趣。

[The First Few Milliseconds of an HTTPS Connection](#)：如果你想深究你抓到的 TLS 连接建立的包中每一段报文的意思，这篇文章是一个很好的参考。

文中介绍了两种生成 Pre-master Secret 的方法，其中第二种的方法是 Diffie-Hellman 密钥交换的变种，这里蕴含的数学原理很有意思，如果你感兴趣，请参阅[维基百科链接](#)。



全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 | 网络互联的昨天、今天和明天：HTTP 协议的演化

下一篇 03 | 换个角度解决问题：服务端推送技术

精选留言 (3)

写留言



饭团 置顶

2019-09-13

回答老师问题

1)不能 因为虽然https是安全的，但前提是你的访问对象是安全的，归根到底你要保证真实是真实的，安全的！是你想访问对象！因为证书也是可以自己生成的！

2)为了性能，非对称加密算法性能不好！对称算法性能高！

展开 ∨

作者回复: 1) 结论正确，但是解释不太妥当。HTTPS 可以达到数据在网络传输过程中的可靠性，但是支付过程是一个复杂和综合性的行为，涉及到的过程和角色远不只有 HTTPS 连接和它的客户端、服务端，因此 HTTPS 的安全性结论无法推广到整个支付过程和支付行为的安全性结论。

2) 性能是一个非常重要的因素，说得很好，因为非对称性加密的性能要比对称性加密的性能差很多，特别是在被加密数据量比较大的时候，但它的问题在于无法把密钥传递到对端，因此我们才使用了非对称加密的方式来帮助做到这一点。但是，还有其它原因，比如说，对称性密钥是每次会话生成的，会话以外自动失效，这就像武功唯快不破一样，通常很短的时间就更换掉了；如果使用非对称性加密方式来传输实际数据，因为它只在最开始的时候生成一次，而不是每次会话都生成，因此在传输中同一个公钥会被发给多个不同的客户端，因此第三方的中间人可以使用这个公开的公钥解密服务端发给其它客户端的数据，这显然不具备安全性。



4



Kada

2019-09-15

请问下关于 Repudiation，否认这项；

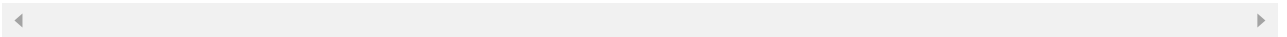
HTTPS如何保证不被否认的？因为有TLS建立过程，有双方公私钥的参与？

作者回复: 好问题！

目前的 TLS 机制无法单独保证 Repudiation（否认），请注意我在文中也写到“第四个问题还需要引入数字签名来解决”。

Repudiation 的解决，必须要靠类似数字签名这样的机制：A 必须收到 B 经过对方私钥加密的消息，这样 A 再使用 B 的公钥解密。而公钥是公开的，也是通过证书等权威形式认证的，所有人都是可以拿到的，公证方一使用 B 的公钥解密，就可以证明这条消息确实来自 B，那么 B 也就无法

否认事实了。当然，实际操作的过程中私钥加密的未必是原始信息，也可以是原始信息的散列值，从而保证效率，但效果是一样的，这也是我们看到的数字签名的原理。



💬 1



anginiit

2019-09-13

老师 我想问一下，那个快速验证，加密的内容是什么，服务器端解密后 是怎么知道内容是正确的呢？

作者回复: 加密的内容包含了前面连接过程所有传输的数据，并进行摘要，得到的这个数据串，发送到对端；而对端也拥有所有的数据，经过同样的摘要，那么它应该和对端传过来的摘要相等。这个过程对于客户端和服务端来说都是类似的。

