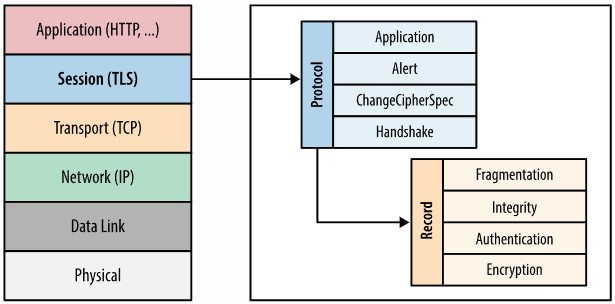
# 传输层安全 (TLS)

SSL 协议最初是在 Netscape 开发的，用于实现 Web 上的电子商务交易安全，这需要加密以保护客户的个人数据，以及身份验证和完整性保证以确保安全交易。为了实现这一点，SSL 协议在应用层实现，直接在 TCP 之上，使其之上的协议（HTTP、电子邮件、即时消息和许多其他）能够在不改变运行的同时提供通信安全性通过网络进行通信。正确使用 SSL 时，第三方观察者只能推断连接端点、加密类型以及发送数据的频率和大致数量，但无法读取或修改任何实际数据。



SSL 2.0 是该协议的第一个公开发布版本，但由于发现了许多安全漏洞，它很快被 SSL 3.0 取代。因为 SSL 协议是 Netscape 专有的，所以 IETF 努力使该协议标准化，从而产生了 RFC 2246，它于 1999 年 1 月发布并被称为 TLS 1.0。从那时起，IETF 一直在对该协议进行迭代以解决安全漏洞并扩展其功能：TLS 1.1 (RFC 4346) 于 2006 年 4 月发布，TLS 1.2 (RFC 5246) 于 2008 年 8 月发布，现在工作正在进行中正在定义 TLS 1.3。

# 加密、身份验证和完整性

TLS 协议旨在为在其上运行的所有应用程序提供三项基本服务：加密、身份验证和数据完整性。从技术上讲，您不需要在每种情况下都使用这三个。您可能决定接受证书而不验证其真实性，但您应该充分了解这样做的安全风险和影响。在实践中，安全的 Web 应用程序将利用所有三种服务。

* 加密：一种混淆从一个主机发送到另一个主机的内容的机制。
* 验证：一种验证所提供身份证明材料有效性的机制。
* 完整：一种检测消息篡改和伪造的机制。

为了建立加密安全的数据通道，连接对等方必须就将使用哪些密码套件以及用于加密数据的密钥达成一致。TLS 协议指定了一个明确定义的握手序列来执行此交换，。这种握手的巧妙部分以及 TLS 在实践中起作用的原因在于它使用了公钥加密（也称为非对称密钥加密），它允许对等方协商共享密钥而无需建立任何先验知识彼此，并通过未加密的通道这样做。

作为 TLS 握手的一部分，该协议还允许对等方验证他们的身份。当在浏览器中使用时，这种身份验证机制允许客户端验证服务器是它声称的身份（例如，您的银行），而不是简单地通过欺骗其名称或 IP 地址来假装是目的地的人。此外，服务器还可以选择性地验证客户端的身份——例如，公司代理服务器可以验证所有员工，每个员工都可以拥有自己的公司签署的唯一证书。

最后，在加密和认证到位后，TLS 协议还提供了自己的消息成帧机制，并使用消息认证码 (MAC) 对每条消息进行签名。MAC 算法是一种单向加密散列函数（实际上是校验和），其密钥由两个连接对等方协商。每当发送 TLS 记录时，都会为该消息生成并附加一个 MAC 值，然后接收方能够计算和验证发送的 MAC 值，以确保消息的完整性和真实性。

结合起来，这三种机制可作为 Web 上安全通信的基础。所有现代 Web 浏览器都支持各种密码套件，能够对客户端和服务器进行身份验证，并对每条记录透明地执行消息完整性检查。

# HTTPS

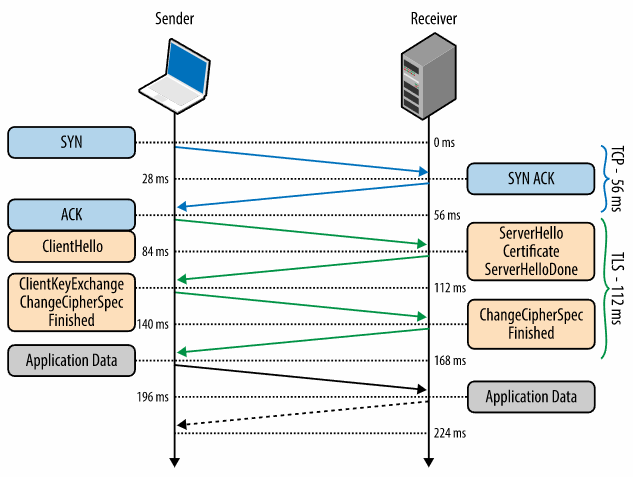
通过 HTTP 和其他协议进行的未加密通信会产生大量隐私、安全和完整性漏洞。此类交换容易受到拦截、操纵和冒充，并可能泄露用户凭据、历史记录、身份和其他敏感信息。我们的应用程序需要通过 HTTPS 传输数据来保护自己和我们的用户免受这些威胁。

* HTTPS 保护网站的完整性
* HTTPS 保护用户的隐私和安全
* HTTPS 在 Web 上启用强大的功能

# TLS 握手

在客户端和服务器开始通过 TLS 交换应用程序数据之前，必须协商加密隧道：客户端和服务器必须就 TLS 协议的版本达成一致，选择密码套件，并在必要时验证证书。不幸的是，这些步骤中的每一个都需要在客户端和服务器之间进行新的数据包往返，这会增加所有 TLS 连接的启动延迟。

* 0 ms：TLS 运行在可靠传输 (TCP) 上，这意味着我们必须首先完成 TCP 三次握手，这需要一个完整的往返。
* 56 ms：建立 TCP 连接后，客户端会以纯文本形式发送许多规范，例如它正在运行的 TLS 协议的版本、支持的密码套件列表以及它可能想要使用的其他 TLS 选项。
* 84 ms：服务器选择 TLS 协议版本进行进一步通信，从客户端提供的列表中决定密码套件，附加其证书，并将响应发送回客户端。可选地，服务器还可以发送对客户端证书和其他 TLS 扩展参数的请求。
* 112 ms：假设双方能够协商一个共同的版本和密码，并且客户端对服务器提供的证书感到满意，则客户端发起 RSA 或 Diffie-Hellman 密钥交换，用于建立对称密钥随后的会议。
* 140 ms：服务器处理客户端发送的密钥交换参数，通过验证 MAC 来检查消息的完整性，并将加密的Finished消息返回给客户端。
* 168 ms：客户端使用协商的对称密钥解密消息，验证 MAC，如果一切正常，则建立隧道，现在可以发送应用程序数据。



如上述交换所示，新的 TLS 连接需要两次往返才能“完全握手”。 ——这是个坏消息。然而，在实践中，优化部署可以做得更好，并提供一致的 1-RTT TLS 握手：

* False Start 是一种 TLS 协议扩展，它允许客户端和服务器在握手仅部分完成时开始传输加密的应用程序数据——即，一旦 ChangeCipherSpec发送Finished消息，但无需等待对方也这样做。
* 如果客户端之前已经与服务器进行过通信，则可以使用“缩写握手”，这需要一次往返，并且还允许客户端和服务器通过重用之前协商的安全会话参数来减少 CPU 开销

# RSA、Diffie-Hellman 和前向保密

RSA 握手已成为大多数 TLS 部署中的主要密钥交换机制：客户端生成对称密钥，使用服务器的公钥对其进行加密，然后将其发送到服务器以用作对称密钥已建立会话的密钥。反过来，服务器使用其私钥解密发送的对称密钥，密钥交换完成。从这一点开始，客户端和服务器使用协商的对称密钥来加密他们的会话。

RSA 握手有效，但有一个严重的弱点：相同的公私密钥对用于验证服务器和加密发送到服务器的对称会话密钥。结果，如果攻击者获得了服务器私钥的访问权并监听了交换，那么他们就可以解密整个会话。更糟糕的是，即使攻击者当前无法访问私钥，他们仍然可以记录加密会话，并在获得私钥后对其进行解密。

相比之下，Diffie-Hellman 密钥交换允许客户端和服务器协商共享密钥，而无需在握手中明确传达它：服务器的私钥用于签名和验证握手，但已建立的对称密钥永远不会离开客户端或服务器，即使他们有权访问私钥，也不能被被动攻击者拦截。

Diffie-Hellman 密钥交换可用于降低过去通信会话受到损害的风险：我们可以生成一个新的“临时”对称密钥作为每次密钥交换的一部分，并丢弃以前的密钥。因此，由于临时密钥永远不会被通信并且会为每个新会话主动重新协商，最坏的情况是攻击者可能会破坏客户端或服务器并访问当前和未来会话的会话密钥。但是，知道私钥或当前的临时密钥并不能帮助攻击者解密之前的任何会话！

结合使用 Diffie-Hellman 密钥交换和临时会话密钥可以实现“完美前向保密”（PFS）：长期密钥（例如服务器的私钥）的泄露不会泄露过去的会话密钥，并且不允许攻击者解密以前记录的会话。至少可以说，这是一个非常理想的财产！

# 应用层协议协商 (ALPN)

两个网络对等体可能希望使用自定义应用程序协议相互通信。解决此问题的一种方法是预先确定协议，为其分配一个众所周知的端口（例如，用于 HTTP 的端口 80，用于 TLS 的端口 443），并配置所有客户端和服务器以使用它。然而，在实践中，这是一个缓慢且不切实际的过程：每个端口分配都必须得到批准，更糟糕的是，防火墙和其他中介通常只允许端口 80 和 443 上的流量。

因此，为了能够轻松部署自定义协议，我们必须重用端口 80 或 443，并使用额外的机制来协商应用程序协议。端口 80 是为 HTTP 保留的，HTTP 规范Upgrade为此目的提供了一个特殊的流程。但是，使用Upgrade可能会增加额外的网络往返延迟，并且在实践中，在存在许多中介的情况下通常是不可靠的。

解决方案是使用端口 443，该端口是为通过 TLS 运行的安全 HTTPS 会话保留的。端到端加密隧道的使用混淆了来自中间代理的数据，并提供了一种快速可靠的方式来部署新的应用程序协议。但是，我们仍然需要另一种机制来协商将在 TLS 会话中使用的协议。顾名思义，应用层协议协商 (ALPN) 是解决此需求的 TLS 扩展。它扩展了 TLS 握手并允许对等方协商协议而无需额外的往返。具体流程如下：

* ProtocolNameList客户端将包含支持的应用程序协议列表的新字段 附加到ClientHello消息中。
* 服务器检查该ProtocolNameList字段并返回一个ProtocolName指示所选协议的字段作为ServerHello消息的一部分。

服务器可能只响应一个协议名称，如果它不支持任何客户端请求，那么它可能会选择中止连接。结果，一旦 TLS 握手完成，安全隧道就建立起来了，客户端和服务器就将使用哪个应用协议达成一致；客户端和服务器可以立即开始通过协商协议交换消息。

# 服务器名称指示 (SNI)

可以在任意两个 TCP 对等体之间建立加密的 TLS 隧道：客户端只需要知道另一个对等体的 IP 地址即可建立连接并执行 TLS 握手。但是，如果服务器想要在同一个 IP 地址上托管多个独立站点，每个站点都有自己的 TLS 证书，那该怎么办？为了解决上述问题，TLS 协议中引入了服务器名称指示 (SNI) 扩展，它允许客户端指示客户端尝试连接的主机名，作为 TLS 握手的一部分。反过来，服务器能够检查ClientHello消息中发送的 SNI 主机名，选择适当的证书，并完成所需主机的 TLS 握手。

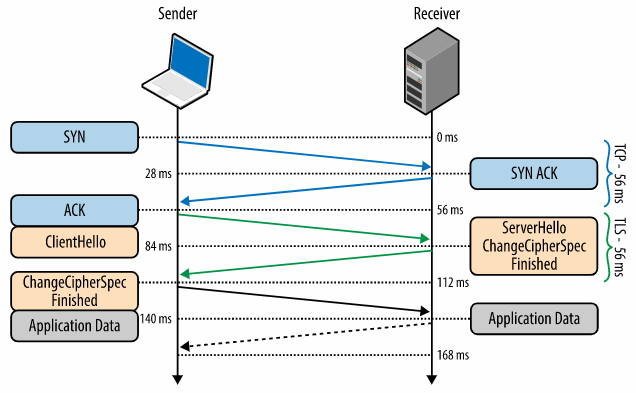
# TLS 会话恢复

完整 TLS 握手的额外延迟和计算成本会对所有需要安全通信的应用程序造成严重的性能损失。为了帮助降低一些成本，TLS 提供了一种机制来恢复或在多个连接之间共享相同的协商密钥数据。

## 会话标识符

ServerHello 第一个会话标识符 (RFC 5246) 恢复机制是在 SSL 2.0 中引入的，它允许服务器在我们之前看到的完整 TLS 协商期间创建并发送一个 32 字节的会话标识符作为其消息的一部分。有了session-ID，客户端和服务器都可以存储先前协商的会话参数（由session-ID键入），并将它们重用于后续会话。

具体来说，客户端可以在 ClientHello消息中包含session-ID，以向服务器指示它仍然记得协商的密码套件和来自先前握手的密钥，并且能够重用它们。反过来，如果服务器能够在其缓存中找到与advertised-ID 关联的会话参数，则可以进行简短的握手。否则，需要一个全新的会话协商，这将生成一个新的session-ID。



利用会话标识符，我们可以消除完整的往返，以及用于协商共享密钥的公钥加密的开销。这允许快速建立安全连接并且不会损失安全性，因为我们正在重用先前协商的会话数据。

## 会话票

解决对 TLS 会话缓存的服务器端部署的问题，引入了Session Ticket （会话票证）（RFC 5077）替换机制，它消除了服务器保持每个客户端会话状态的要求。相反，如果客户端表明它支持会话票证，则服务器可以包含一条New Session Ticket记录，该记录包括使用只有服务器知道的密钥加密的所有协商会话数据。该会话票证随后由客户端存储，并且可以包含在后续会话消息中的SessionTicket扩展中 。ClientHello因此，所有会话数据仅存储在客户端上，但票证仍然是安全的，因为它是使用只有服务器知道的密钥加密的。

会话标识符和会话票据机制通常分别称为会话缓存和无状态恢复机制。无状态恢复的主要改进是删除了服务器端会话缓存，这通过要求客户端在与服务器的每个新连接上提供会话票证来简化部署——也就是说，直到票证过期。

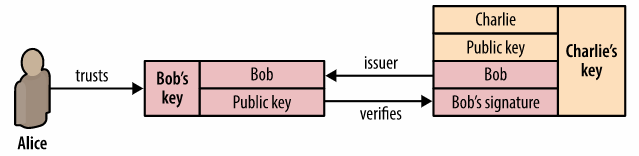
# 信任链和证书颁发机构

身份验证是建立每个 TLS 连接的一个组成部分。毕竟，可以通过加密隧道与任何对等方（包括攻击者）进行对话，除非我们可以确定我们正在与之交谈的主机是我们信任的主机，否则所有加密工作都可能用于没有。要了解我们如何验证对等方的身份，让我们检查 Alice 和 Bob 之间的简单身份验证工作流程：

* Alice 和 Bob 都生成自己的公钥和私钥。
* Alice 和 Bob 都隐藏了各自的私钥。
* Alice 与 Bob 共享她的公钥，Bob 与 Alice 共享他的公钥。
* Alice 为 Bob 生成一条新消息，并用她的私钥对其进行签名。
* Bob 使用 Alice 的公钥来验证提供的消息签名。

信任是上述交换的关键组成部分。具体来说，公钥加密允许我们使用发送者的公钥来验证消息是否使用正确的私钥签名，但批准发送者的决定仍然是基于信任的决定。在刚刚展示的交换中，Alice 和 Bob 在他们亲自见面时可能已经交换了他们的公钥，并且因为他们彼此非常了解，他们确信他们的交换没有被冒名顶替者破坏——也许他们甚至通过另一个人验证了他们的身份，他们之前建立的秘密（物理）握手！

接下来，Alice 收到来自 Charlie 的消息，她从未谋面，但自称是 Bob 的朋友。事实上，为了证明他是 Bob 的朋友，Charlie 要求 Bob 用 Bob 的私钥签署他自己的公钥，并将这个签名附加到他的消息中。在这种情况下，爱丽丝首先检查鲍勃对查理密钥的签名。她知道 Bob 的公钥，因此能够验证 Bob 确实签署了 Charlie 的密钥。因为她信任 Bob 验证 Charlie 的决定，所以她接受该消息并对 Charlie 的消息执行类似的完整性检查，以确保它确实来自 Charlie。

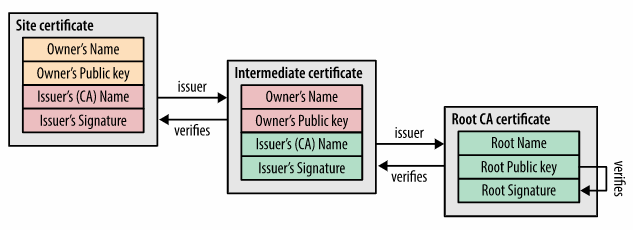


我们刚才所做的是建立了一条信任链​​：Alice 信任 Bob，Bob 信任 Charlie，通过传递信任，Alice 决定信任 Charlie。只要链中没有人受到损害，这使我们能够建立和增加受信任方的列表。

Web 和浏览器中的身份验证遵循与所示完全相同的过程。

* 手动指定的证书：每个浏览器和操作系统都为您提供了手动导入您信任的任何证书的机制。如何获取证书并验证其完整性完全取决于您。
* 证书颁发机构：证书颁发机构 (CA) 是受证书主体（所有者）和依赖证书的一方信任的受信任第三方。
* 浏览器和操作系统：每个操作系统和大多数浏览器都附带一个知名证书颁发机构的列表。因此，您还信任该软件的供应商提供和维护受信任方列表。

在实践中，存储和手动验证每个网站的每个密钥是不切实际的（尽管您可以，如果您愿意的话）。因此，最常见的解决方案是使用证书颁发机构 (CA) 为我们完成这项工作：浏览器指定信任哪些 CA（根 CA），然后由 CA 负责验证每个他们签署的网站，并审核和验证这些证书没有被滥用或泄露。如果具有 CA 证书的任何站点的安全性遭到破坏，则该 CA 也有责任吊销受损证书。



# 证书吊销列表 (CRL)

证书撤销列表 (CRL) 由 RFC 5280 定义，并指定了检查每个证书状态的简单机制：每个证书颁发机构维护并定期发布已撤销证书序列号的列表。任何试图验证证书的人都可以下载吊销列表，对其进行缓存，并检查其中是否存在特定的序列号——如果存在，则说明它已被吊销。此过程简单明了，但有许多限制：

* 越来越多的撤销意味着 CRL 列表只会变得更长，每个客户端都必须检索整个序列号列表。
* 没有即时通知证书吊销的机制——如果 CRL 在证书被吊销之前被客户端缓存，那么 CRL 将认为被吊销的证书有效，直到缓存过期。
* 从 CA 获取最新 CRL 列表的需要可能会阻止证书验证，这可能会显着增加 TLS 握手的延迟。
* 由于各种原因，CRL 提取可能会失败，在这种情况下，浏览器行为是未定义的。大多数浏览器将此类情况视为“软失败”，允许验证继续进行。

# 在线证书状态协议 (OCSP)

为了解决 CRL 机制的一些限制，RFC 2560 引入了在线证书状态协议 (OCSP)，它提供了一种对证书状态执行实时检查的机制。与包含所有已撤销序列号的 CRL 文件不同，OCSP 允许客户端在验证证书链时直接查询 CA 的证书数据库以获取相关序列号。因此，OCSP 机制消耗的带宽更少，并且能够提供实时验证。但是，执行实时 OCSP 查询的要求会产生一系列问题：

* CA 必须能够处理实时查询的负载。
* CA 必须确保服务始终处于启动状态并在全球范围内可用。
* 实时 OCSP 请求可能会损害客户端的隐私，因为 CA 知道客户端正在访问哪些站点。
* 客户端在验证证书链时必须阻止 OCSP 请求。
* 再次，浏览器行为是未定义的，如果 OCSP 获取由于网络超时或其他错误而失败，通常会导致“软失败”。

# OCSP 装订

由于上面列出的原因，CRL 或 OSCP 撤销机制都没有提供我们希望为我们的应用程序提供的安全性和性能保证。但是，不要绝望，因为 OCSP 装订（RFC 6066，“证书状态请求”扩展名）通过允许服务器执行验证并作为与客户端的 TLS 握手：

* 不是客户端发出 OCSP 请求，而是服务器定期从 CA 检索签名和时间戳的 OCSP 响应。
* 然后，服务器附加（即“装订”）签名的 OCSP 响应作为 TLS 握手的一部分，允许客户端验证 CA 签名的证书和附加的 OCSP 撤销记录。

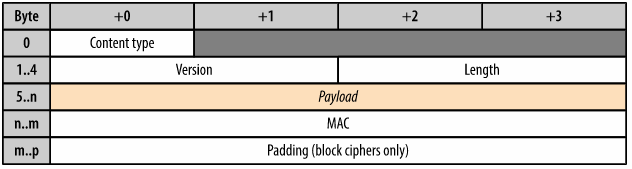
这种角色转换是安全的，因为装订记录由 CA 签名并且可以由客户端验证，并提供了许多重要的好处：

* 客户端不会泄露其导航历史。
* 客户端不必阻塞和查询 OCSP 服务器。
* 如果服务器选择加入（通过宣传 OSCP“Must-Staple”标志）并且验证失败，则客户端可能会“硬失败”撤销处理。

简而言之，要获得最佳的安全性和性能保证，请确保在您的服务器上配置和测试 OCSP 装订。

# TLS 记录协议

与它下面的 IP 或 TCP 层不同，在 TLS 会话中交换的所有数据也都使用定义良好的协议进行构建。TLS 记录协议负责识别不同类型的消息（握手、警报或通过“内容类型”字段的数据），以及保护和验证每条消息的完整性。



交付应用程序数据的典型工作流程如下：

* 记录协议接收应用程序数据。
* 接收到的数据被分成块：最多 2个 14 字节，或每条记录 16 KB。
* 消息验证码 (MAC) 或 HMAC 被添加到每条记录。
* 每条记录中的数据都使用协商的密码进行加密。

完成这些步骤后，加密数据将向下传递到 TCP 层进行传输。在接收端，对等方应用相同的工作流程，但相反：使用协商密码解密记录，验证 MAC，提取数据并将其传递给它上面的应用程序。

刚才展示的所有工作都由 TLS 层本身处理，并且对大多数应用程序完全透明。但是，记录协议确实引入了一些我们需要注意的重要含义：

* 最大 TLS 记录大小为 16 KB
* 每条记录包含一个 5 字节的标头、一个 MAC（对于 SSLv3、TLS 1.0、TLS 1.1 最多 20 个字节，对于 TLS 1.2 最多 32 个字节）和填充（如果使用块密码）。
* 要解密和验证记录，整个记录必须可用。

为您的应用程序选择正确的记录大小（如果您有能力这样做）可能是一项重要的优化。由于记录成帧和 MAC 验证，小记录会产生较大的 CPU 和字节开销，而大记录必须先由 TCP 层交付和重新组装，然后才能由 TLS 层处理并交付给您的应用程序——跳到优化TLS 记录大小以获取完整详细信息。

# TLS优化

通过 TLS 部署您的应用程序将需要一些额外的工作，包括在您的应用程序中（例如，将资源迁移到 HTTPS 以避免混合内容），以及负责通过 TLS 传递应用程序数据的基础设施的配置。一个经过良好调整的部署可以对观察到的性能、用户体验和总体运营成本产生巨大的积极影响。

## 降低计算成本

建立和维护加密通道会为双方带来额外的计算成本。具体来说，首先是在 TLS 握手期间使用的非对称（公钥）加密（解释为TLS 握手）。然后，一旦建立了共享密钥，它就会用作对称密钥来加密所有 TLS 记录。与对称密钥加密相比，公钥加密的计算成本更高，并且在 Web 的早期，通常需要额外的硬件来执行“SSL 卸载”。 当然，现在这不再是必需的。

## 启用 1-RTT TLS 握手

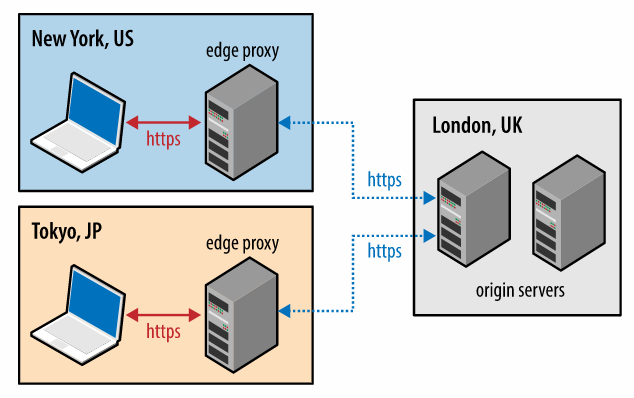
未优化的 TLS 部署很容易添加许多额外的往返并为用户引入显着延迟 - 例如多次 RTT 握手、缓慢且无效的证书撤销检查、需要多次往返的大型 TLS 记录等等。不要成为那个网站，你可以做得更好。一个经过良好调整的 TLS 部署应该最多添加一个额外的往返来协商 TLS 连接，无论它是新的还是恢复的，并避免所有其他延迟陷阱：配置会话恢复，并启用前向保密以启用 TLS 错误启动。

## 优化连接重用

最小化设置新 TCP+TLS 连接的延迟和计算开销的最佳方法是优化连接重用。这样做可以在请求之间分摊设置成本，并为用户提供更快的体验。验证您的服务器和代理配置是否设置为允许保持连接，并审核您的连接超时设置。许多流行的服务器设置了激进的连接超时（例如，某些 Apache 版本默认为 5 秒超时），这会强制进行许多不必要的重新协商。为获得最佳结果，请使用您的日志和分析来确定最佳超时值。

## 利用提前终止

正如我们在延迟和带宽入门中所讨论的，我们可能无法让我们的数据包传输得更快，但我们可以让它们传输更短的距离。通过将我们的“边缘”服务器放置在更靠近用户的位置，我们可以显着减少 TCP 和 TLS 握手的往返时间和总成本。



实现此目的的一种简单方法是利用内容交付网络 (CDN) 的服务，该网络在全球范围内维护着边缘服务器池，或者部署您自己的。通过允许用户终止与附近服务器的连接，而不是穿越海洋和大陆链接到您的原点，客户端可以获得“提前终止”和更短的往返行程的好处。这种技术对于静态和动态内容同样有用和重要：静态内容也可以由边缘服务器缓存和服务，而动态请求可以通过已建立的连接从边缘路由到源。

## 配置会话缓存和无状态恢复

在离用户更近的地方终止连接是一种优化，有助于在所有情况下减少用户的延迟，但再一次，没有比特比不发送比特更快——发送更少的比特。启用 TLS 会话缓存和无状态恢复使我们能够消除整个往返延迟并减少重复访问者的计算开销。

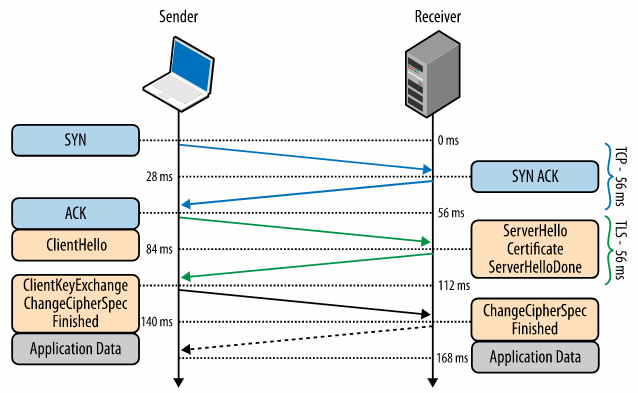
TLS 会话缓存所依赖的会话标识符是在 SSL 2.0 中引入的，并在大多数客户端和服务器中得到广泛支持。但是，如果您在服务器上配置 TLS，请不要假设默认情况下会启用会话支持。事实上，在大多数服务器上默认关闭它更为常见——但您知道得更清楚！仔细检查并验证您的服务器、代理和 CDN 配置：

* 具有多个进程或工作人员的服务器应使用共享会话缓存。
* 共享会话缓存的大小应根据您的流量级别进行调整。
* 应提供会话超时期限。
* 在多服务器设置中，将相同的客户端 IP 或相同的 TLS 会话 ID 路由到同一服务器是提供良好会话缓存利用率的一种方式。
* 在不能选择“粘性”负载平衡的情况下，应在不同服务器之间使用共享缓存以提供良好的会话缓存利用率，并且需要建立安全机制来共享和更新密钥以解密提供的会话票证。
* 检查和监控您的 TLS 会话缓存统计信息以获得最佳性能。

## 启用 TLS 错误启动

会话恢复提供了两个重要的好处：它消除了回访者的额外握手往返，并通过允许重用先前协商的会话参数来降低握手的计算成本。但是，在访问者第一次与服务器通信或前一个会话已过期的情况下，它无济于事。

为了获得两全其美——新访客和重复访客的一次往返握手，以及重复访客的计算节省——我们可以使用 TLS False Start，这是一个可选的协议扩展，允许发送者发送应用程序数据当握手仅部分完成时。



False Start 不会修改 TLS 握手协议，而只会影响发送应用程序数据的协议时序。直观地说，一旦客户端发送了 ClientKeyExchange记录，它就已经知道加密密钥并可以开始传输应用程序数据——握手的其余部分用于确认没有人篡改握手记录，并且可以并行完成。因此，False Start 允许我们将 TLS 握手保持在一次往返中，无论我们是执行完整握手还是简短握手。

## 优化 TLS 记录大小

通过 TLS 传递的所有应用程序数据都在记录协议中传输。每条记录的最大大小为 16 KB，根据选择的密码，每条记录将为标头、MAC 和可选填充添加 20 到 40 字节的开销。如果记录随后适合单个 TCP 数据包，那么我们还必须添加 IP 和 TCP 开销：IP 的 20 字节标头和没有选项的 TCP 20 字节标头。因此，每条记录可能会产生 60 到 100 字节的开销。对于线路上 1,500 字节的典型最大传输单元 (MTU) 大小，此数据包结构转换为至少 6% 的帧开销。

记录越小，帧开销就越高。但是，简单地将记录的大小增加到其最大大小 (16 KB) 不一定是一个好主意。如果记录跨越多个 TCP 数据包，则 TLS 层必须等待所有 TCP 数据包到达后才能解密数据。如果这些 TCP 数据包中的任何一个由于拥塞控制而丢失、重新排序或受到限制，则 TLS 记录的各个片段必须在解码之前进行缓冲，从而导致额外的延迟。在实践中，这些延迟会给浏览器造成严重的瓶颈，浏览器更喜欢以流方式消费数据。

小记录会产生开销，大记录会产生延迟，并且“最佳”记录大小没有一个值。相反，对于浏览器使用的 Web 应用程序，最好的策略是根据 TCP 连接的状态动态调整记录大小：

* 当连接是新的并且 TCP 拥塞窗口较低时，或者当连接空闲一段时间（参见Slow-Start Restart）时，每个 TCP 数据包应该携带一个 TLS 记录，并且 TLS 记录应该占据完整的最大段TCP 分配的大小 (MSS)。
* 当连接拥塞窗口很大并且我们正在传输大流（例如，流式视频）时，可以增加 TLS 记录的大小以跨越多个 TCP 数据包（最大 16KB），以减少客户端的帧和 CPU 开销和服务器。

使用动态策略为交互式流量提供了最佳性能：小记录大小消除了不必要的缓冲延迟并提高了第一个{HTML字节，...，视频帧}的时间，更大的记录大小通过最小化开销来优化吞吐量用于长寿命流的 TLS。

为了确定每个状态的最佳记录大小，让我们从一个新的或空闲的 TCP 连接的初始情况开始，我们希望避免 TLS 记录跨越多个 TCP 数据包：

* 为 IPv4 帧开销分配 20 个字节，为 IPv6 分配 40 个字节。
* 为 TCP 帧开销分配 20 个字节。
* 为 TCP 选项开销（时间戳、SACK）分配 40 个字节。

假设一个通用的 1,500 字节的起始 MTU，这将为通过 IPv4 传递的 TLS 记录留下 1,420 字节，为 IPv6 提供 1,400 字节。为了面向未来，请使用 IPv6 大小，这为每个 TLS 记录留下 1,400 个字节，如果您的 MTU 较低，请根据需要进行调整。

接下来，关于何时增加记录大小并在连接空闲时重置的决定，可以根据预先配置的阈值进行设置： X在传输 KB 数据后将记录大小增加到最多 16 KB，然后重置Y空闲时间毫秒 后的记录大小。通常，配置 TLS 记录大小不是我们可以在应用层控制的。相反，这通常是 TLS 服务器的设置，有时是编译时常量。有关如何配置这些值的详细信息，请查看服务器文档。

## 优化证书链

验证信任链需要浏览器遍历链，从站点证书开始，递归地验证父节点的证书，直到到达可信根。因此，提供的链包含所有中间证书至关重要。如果有任何遗漏，浏览器将被迫暂停验证过程并获取丢失的证书，在过程中添加额外的 DNS 查找、TCP 握手和 HTTP 请求。

相反，不要包含不必要的证书，例如证书链中的受信任根——它们会添加不必要的字节。回想一下，服务器证书链是作为 TLS 握手的一部分发送的，这很可能发生在处于慢启动算法早期阶段的新 TCP 连接上。如果证书链大小超过了 TCP 的初始拥塞窗口，那么我们会无意中为 TLS 握手添加额外的往返：证书长度将溢出拥塞窗口并导致服务器停止并等待客户端 ACK 再继续。

## 配置 OCSP 装订

每个新的 TLS 连接都要求浏览器必须验证发送的证书链的签名。但是，我们不能忘记还有一个更关键的步骤：浏览器还需要验证证书没有被吊销。要验证证书的状态，浏览器可以使用以下几种方法之一：证书撤销列表 (CRL)、在线证书状态协议 (OCSP)或OCSP 装订。每种方法都有其自身的局限性，但 OCSP 装订提供了迄今为止最好的安全性和性能保证。

确保将您的服务器配置为包含（装订）来自 CA 的 OCSP 响应到提供的证书链。这样做允许浏览器执行吊销检查，而无需任何额外的网络往返并具有改进的安全保证。

* OCSP 响应的大小可以从 400 到 4,000 字节不等。将此响应装订到您的证书链将增加其大小 - 请密切注意证书链的总大小，以便它不会溢出新 TCP 连接的初始拥塞窗口。
* 当前的 OCSP 装订实现只允许包含单个 OCSP 响应，这意味着如果浏览器需要验证链中的其他证书，则可能不得不回退到另一个撤销机制——减少证书链的长度。将来，OCSP Multi-Stapling 应该可以解决这个特殊问题。

## 启用 HTTP 严格传输安全 (HSTS)

HTTP Strict Transport Security 是一种重要的安全策略机制，它允许源通过简单的 HTTP 标头（例如，“ Strict-Transport-Security: max-age=31536000 ”）向兼容浏览器声明访问规则。具体来说，它指示用户代理执行以下规则：

* 对源的所有请求都应通过 HTTPS 发送。这包括导航和所有其他同源子资源请求——例如，如果用户键入一个没有 https 前缀的 URL，用户代理应该自动将其转换为 https 请求；如果页面包含对非 https 资源的引用，用户代理应自动将其转换为请求 https 版本。
* 如果无法建立安全连接，则不允许用户绕过警告并请求 HTTP 版本——即来源仅 HTTPS。
* max-age以秒为单位指定指定 HSTS 规则集的生命周期（例如，max-age=31536000等于广告策略的 365 天生命周期）。
* includeSubdomains表示该策略应适用于当前来源的所有子域。

HSTS 将源转换为仅限 HTTPS 的目标，并帮助保护应用程序免受各种被动和主动网络攻击。作为一个额外的好处，它还通过消除 HTTP 到 HTTPS 重定向的需要提供了一个很好的性能优化：客户端在分派之前自动将所有请求重写到安全源！

**HSTS 预载列表**：HSTS 机制让源的第一个请求不受主动攻击的保护——例如，恶意方可以降级客户端的请求并阻止其注册 HSTS 策略。为了解决这个问题，大多数浏览器提供了一个单独的“HSTS 预加载列表”机制，允许源请求被包含在浏览器附带的支持 HSTS 的站点列表中。

## 启用 HTTP 公钥固定 (HPKP)

当前系统的缺点之一是我们对大量受信任的证书颁发机构 (CA) 的依赖。一方面，这很方便，因为这意味着我们可以从广泛的实体池中获得有效的证书。但是，这也意味着这些实体中的任何一个也能够在未经他们明确同意的情况下为我们和任何其他来源颁发有效证书。

公钥固定使站点能够发送一个 HTTP 标头，指示浏览器记住（“固定”）其证书链中的一个或多个证书。通过这样做，它能够确定浏览器在后续访问中应接受哪些证书或颁发者：

* 来源可以固定它的叶子证书。这是最安全的策略，因为您实际上是在硬编码一小部分应该被浏览器接受的特定证书签名。
* 源可以固定证书链中的父证书之一。例如，源可以固定其 CA 的中间证书，这告诉浏览器，对于这个特定的源，它应该只信任由该特定证书颁发机构签名的证书。

## 更新为 HTTPS

为了获得最佳的安全性和性能保证，站点实际使用 HTTPS 来获取其所有资源至关重要。否则，我们会遇到一些问题，这些问题会损害两者，或者更糟的是，破坏网站：

* 混合的“活动”内容（例如通过 HTTP 传递的脚本和样式表）将被浏览器阻止，并可能破坏站点的功能。
* 将获取混合的“被动”内容（例如，通过 HTTP 传递的图像、视频、音频等），但允许攻击者观察和推断用户活动，并通过需要额外的连接和握手来降低性能。

审核您的内容并更新您的资源和链接，包括第三方内容，以使用 HTTPS。内容安全策略(CSP) 机制在这里可以提供很大帮助，既可以识别 HTTPS 违规，也可以执行所需的策略。