维护SSL/TLS实践

维护SSL/TLS实践

- 开始
- 概念
  - PKI 联盟
  - 证书颁发机构 (CA)
  - SSL 证书的类型
  - SSL 证书链
  - SSL/TLS/DTLS 加解密
  - SSL/TLS证书工作原理
- 什么是TLS握手
  - TLS 握手
  - TLS 握手期间会发生什么?
  - TLS 握手有哪些步骤?
  - 密码套件
    - 密码套件命名规则
    - TLS 1.0–1.2 密码套件支持的算法
    - TLS 1.3
    - 密码套件协商
    - openssl 查看支持的密码套件
  - 配置示例
  - http2 和TLS
- 工具
  - 使用openssl 自签发证书
  - 解密TLS流量
    - RSA密钥交换算法解密
    - ECDHE密钥交换算法解密
- 常见的SSL/TLS 部署方式
  - 1.服务端直接部署SSL进行卸载(前端无负载均衡设备)
  - 2.前端负载均衡SSL卸载
  - 3.前端负载均衡SSL透传
  - 4.前端负载均衡SSL Full proxy
  - 5.前端负载均衡调用外部HSM进行SSL卸载(专用加密机)
  - 6.两层负载进行SSL卸载
- 参考:

开始

在维护和部署SSL/TLS 时需要评估应用程序需求、选择合适的加密算法和密钥管理策略、准备和配置网络基础设施、监控和更新SSL/TLS部署。同时，需要关注证书颁发机构(CA)选择、证书和私钥管理、证书签名请求(CSR)生成、SSL/TLS协议和密码套件选择等方面的具体事项。本篇文档整体梳理关于SSL/TLS相关的概念，总结了部署位置，详细描述加密套件的各项功能，最后对实际数据包交互和应用场景进行了分析。

概念

PKI 联盟

PKI联盟(Public Key Infrastructure)，即公钥基础设施联盟，是由多个组织和机构组成的合作体，也有亚洲PKI联盟（Asia PKI Consortium, APKIC），他们旨在推动公钥基础设施（PKI）的发展和应用。PKI是一种安全技术，它使用数字证书、证书颁发机构（CA）和加密技术来验证通信双方的身份，并确保数据的完整性和机密性。

证书颁发机构 (CA)

CA（Certificate Authority）是证书颁发机构的简称，它是公钥基础设施（PKI）的核心，负责签发数字证书、认证证书和管理已颁发证书的机关。CA的主要目的是确保企业组织和用户的信息数据在互联网环境下的安全性。CA通过验证申请者的域名所有权和身份信息，颁发数字证书，以建立信任并保护用户与服务器之间传输数据的安全。

CA机构如VeriSign、Thawte、GeoTrust、Comodo（现Sectigo）等，都是国际上知名的CA，它们颁发的证书被广泛信任和使用。在中国，也有中国金融认证中心（CFCA）等机构提供CA服务。

自签名证书是由个人或组织自己签发的证书，通常用于内部系统或测试环境，因为它们没有经过第三方CA的验证，所以不具有公共CA证书的可信度。自签名证书的生成和安装可以通过工具如openssl 、Java的keytool完成，但需要注意，自签名证书存在安全风险，不应在生产环境中使用。

中国金融认证中心（简称 CFCA）是经中国人民银行和国家信息安全管理机构批准成立的国家级权威安全机构。CFCA SSL 证书，是由中国数字证书认证机构自主研发的纯国产证书。该证书已通过微软根证书项目认证、Mozilla 根证书认证，谷歌（安卓）根证书认证和苹果根证书认证，且根证书已经预埋在微软系统、设备，Mozilla 相关产品，谷歌（安卓操作系统）相关产品以及苹果相关产品中。

选择一个核心业务是公钥基础设施（PKI）的CA,以PKI为核心业务的CA拥有专业知识，可以在你的证书部署出现问题时及时处理问题。

在末尾的参考中列举了各个网站机构的链接。

SSL 证书的类型

SSL证书对于网站安全至关重要，证书有多种类型，可以满足不同的安全需求。 SSL 证书的主要类型有：

域验证 (DV) 证书：这些证书提供基本级别的验证，仅验证域的所有权。它们通常用于博客和个人网站。

组织验证 (OV) 证书：这些证书需要更多验证，包括组织的身份，使其适合收集用户数据的商业网站。

扩展验证 (EV) 证书：提供最高级别的验证，可验证域名的所有权和组织的合法存在，非常适合电子商务网站和大型企业。

通配符 SSL 证书：这些证书可保护域及其无限数量的子域，对于具有许多子域的大型网站来说，这是一种经济高效的解决方案。

多域 SSL 证书 (MDC)：也称为统一通信证书 (UCC)，它们使用单个 SSL 证书保护多个域名。

每种类型的 SSL 证书都提供不同级别的安全性和信任度，选择取决于网站所有者的具体需求和预算。

SSL 证书链

证书链用于在公钥基础设施 (PKI) 的证书颁发机构 (CA) 之间建立信任。信任设置根 CA、中间 CA 和颁发的 SSL 证书之间的分层角色和关系。

根CA是证书链的绝对基础。根 CA 颁发的证书具有与根 CA 证书相同的信任级别。根 CA 通常为中间 CA 签署证书。中间 CA 的作用是为根 CA 签署最终实体证书。让中间 CA 执行此任务的目的是限制对根 CA 的暴露。

SSL 证书由中间 CA 签名，用作域特定证书。 SSL 证书安装在启用 SSL 的服务器上，并且在启动与服务器的 SSL 连接时，该证书会呈现给浏览器。浏览器将尝试通过检查证书的签名机构来确认最终实体证书的真实性。

中间CA从属于特定的根CA，并使用由根CA签名的证书。中间 CA 通常是 SSL 证书的签署者。

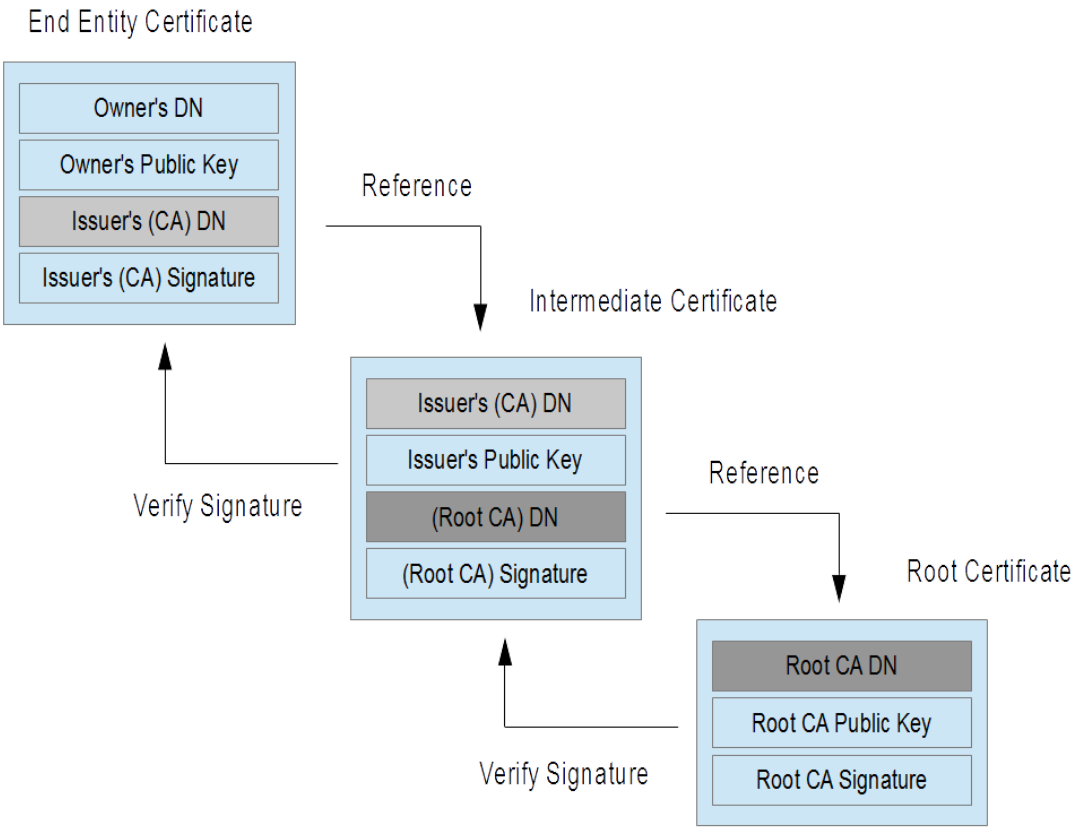
根 CA 使用基于 X.509 的公钥证书来专门标识根 CA。根CA是SSL证书链的绝对签名机构。浏览器供应商提供了一系列已知的、受信任的根 CA，这些 CA 将最终确定 SSL 证书的有效性。

为了使浏览器接受 SSL 证书，该证书必须由具有来自根 CA 的签名证书的 CA 颁发，该根 CA 包含在浏览器的已知可信根 CA 存储中。浏览器将检查每个中间 CA 的证书，以确定它是否是由已知的、受信任的根 CA 颁发的。如果中间 CA 的证书是由另一个中间 CA 签名的，则 Web 浏览器将检查该中间 CA 的证书以查看证书的颁发者是否是已知的、受信任的根 CA。逐级向上，直到找到根 CA 证书并根据浏览器的受信任根 CA 存储信息进行检查。

当根 CA 与浏览器存储中已知的、受信任的根 CA 匹配时，SSL 证书将被视为有效。当根 CA 不匹配时，SSL 证书将被视为不可信。不同的浏览器供应商以不同的方式处理不受信任的 SSL 连接，大多数浏览器供应商都会警告该连接不受信任，要求用户确认或完全不允许建立连接。

将 SSL 证书从 SSL 证书映射到签署 SSL 证书的中间 CA，最终映射到签署中间 CA 证书的根 CA，就是所谓的 SSL 证书链。该链可能包含 SSL 和根 CA 之间的多个中间 CA。

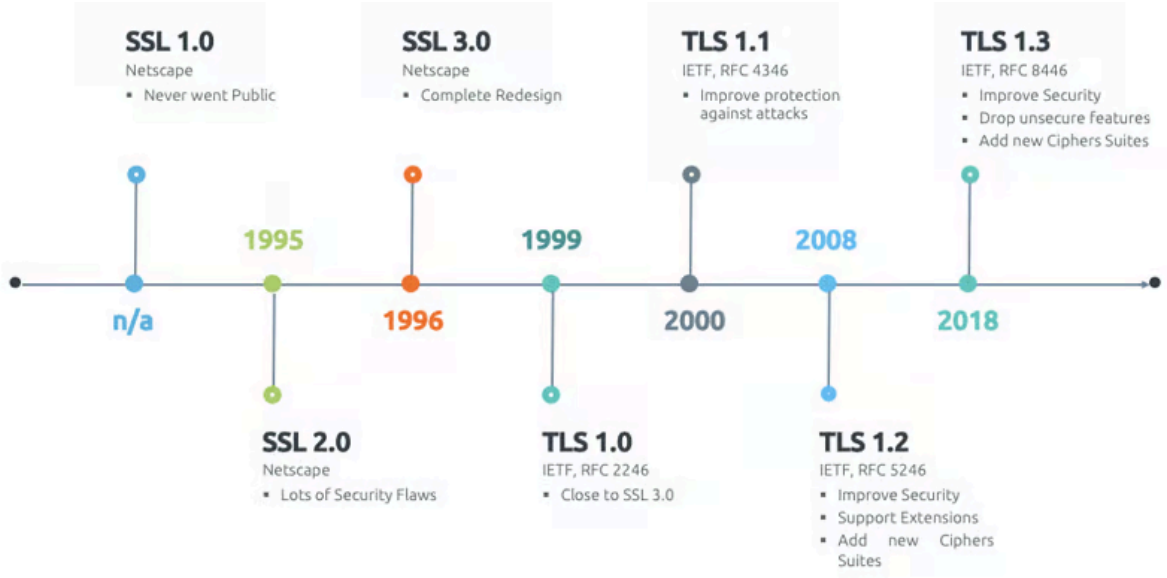
见下图：



SSL/TLS/DTLS 加解密

传输层安全性协议（英语：Transport Layer Security，缩写：TLS），前身称为安全套接层（英语：Secure Sockets Layer，缩写：SSL）是一种安全协议，目的是为互联网通信提供安全及数据完整性保障。TLS 和 SSL 是用于保护基于流的 TCP Internet 流量的标准协议。 DTLS 是基于 TLS 的协议，能够保护用户数据报协议 (UDP) 或流控制传输协议 (SCTP)。 DTLS 非常适合保护对延迟敏感的应用程序和服务、VPN 等隧道应用程序以及往往会耗尽文件描述符或套接字缓冲区的应用程序。

协议	发布时间	状态
SSL 1.0	未公布	未公布
SSL 2.0	1995年	已于2011年弃用
SSL 3.0	1996年	已于2015年弃用
TLS 1.0	1999年	于2021年弃用
TLS 1.1	2006年	于2021年弃用
TLS 1.2	2008年	
TLS 1.3	2018年	



SSL/TLS证书工作原理

SSL握手

证书主要作用是在SSL握手中，我们来看一下SSL的握手过程

1. 客户端提交https请求

务器响应客户，并把证书公钥发给客户端

3. 客户端验证证书公钥的有效性

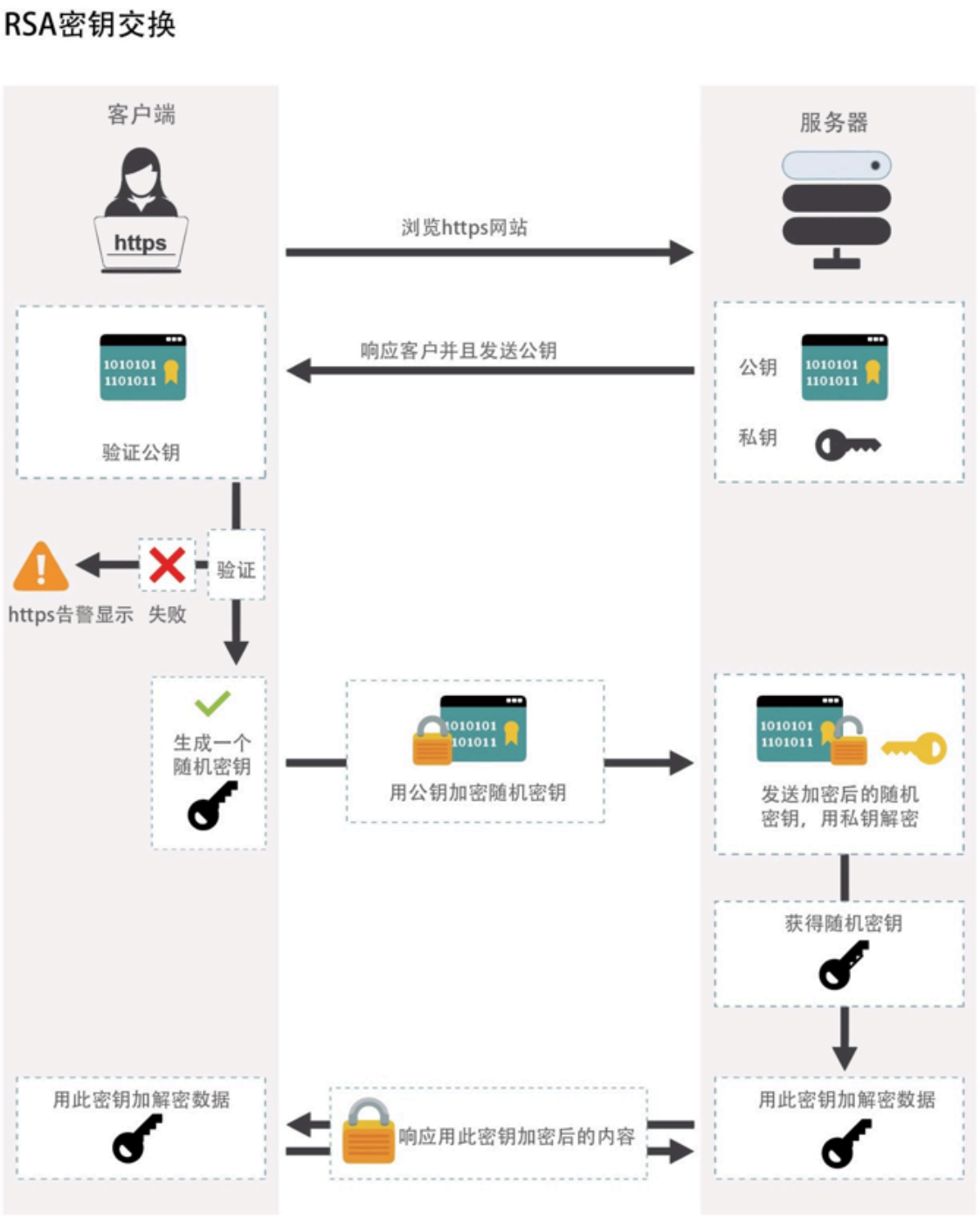
效后，会生成一个预主密钥

5. 用证书公钥加密这个预主密钥后，发送给服务器

务器收到公钥加密的预主密钥后，用私钥解密，获取预主生成会话密钥

7. 客户端与服务器双方利用这个会话密钥加密要传输的数据进行通信

见下图：



在TLS1.3 中会话密钥的生成有点区别。

什么是TLS握手

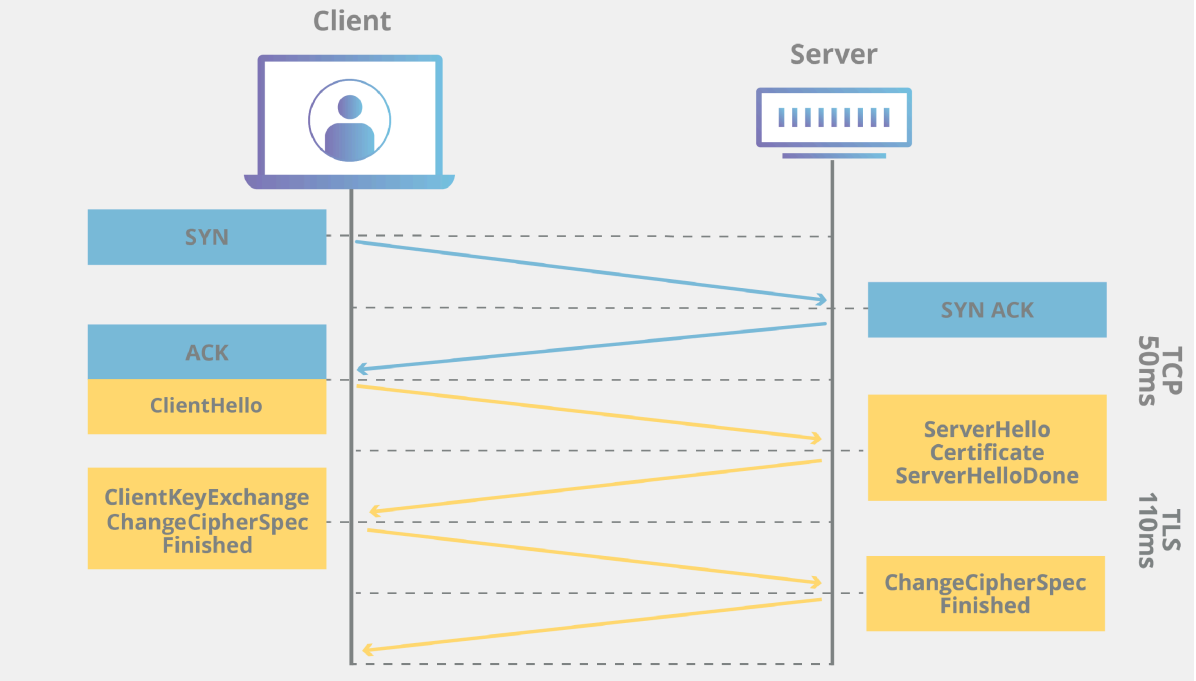
在 TLS/SSL 握手过程中，客户端和服务器交换 SSL 证书、密码套件要求以及为创建会话密钥而随机生成的数据。

TLS 是一种旨在保护互联网通信安全的加密和身份验证协议。TLS 握手是启动 TLS 通信会话的过程。在 TLS 握手过程中，通信双方交换消息以相互确认，彼此验证，确立它们将使用的加密算法，并生成一致的会话密钥。TLS 握手是 HTTPS 工作原理的基础部分。

TLS 握手

SSL（安全套接字层）是为 HTTP 开发的原始安全协议。SSL 已经被 TLS （传输层安全性）所取代。SSL 握手现在称为 TLS 握手，尽管“SSL”这个名称仍在广泛使用。

当用户访问HTTPS网站，浏览器开始工作时这时就会发生TLS握手，浏览器通过TCP握手打开TCP链接后，将发生TLS握手。



TLS 握手期间会发生什么?

在 TLS 握手过程中，客户端和服务器一同执行以下操作：

- 指定将要使用的 TLS 版本（TLS 1.0、1.2、1.3 等）
- 决定将要使用哪些密码套件
- 通过服务器的公钥和 SSL 证书颁发机构的数字签名来验证服务器的身份
- 生成会话密钥，以在握手完成后使用对称加密

TLS 握手有哪些步骤?

TLS 握手是由客户端和服务器交换的一系列数据报或消息。TLS 握手涉及多个步骤，因为客户端和服务器要交换完成握手和进行进一步对话所需的信息。

TLS 握手中的确切步骤将根据所使用的密钥交换算法的种类和双方支持的密码套件而有所不同。RSA 密钥交换算法虽然现在被认为不安全，TLS1.3 非堆成密钥交换的过程中不再使用RSA了。大致如下：

1. **“客户端问候（client hello）”消息：** 客户端通过向服务器发送“问候”消息来开始握手。该消息将包含客户端支持的 TLS 版本，支持的密码套件，以及称为一串称为“客户端随机数（client random）”的随机字节。
2. **“服务器问候（server hello）”消息：** 作为对 client hello 消息的回复，服务器发送一条消息，内含服务器的 [SSL 证书]、服务器选择的密码套件，以及“服务器随机数（server random）”，即由服务器生成的另一串随机字节。
3. **身份验证：** 客户端使用颁发该证书的证书颁发机构验证服务器的 SSL 证书。此举确认服务器是其声称的身份，且客户端正在与该域的实际所有者进行交互。

4. **预主密钥：** 客户端再发送一串随机字节，即“预主密钥（premaster secret）”。预主密钥是使用公钥加密的，只能使用服务器的私钥解密。（客户端从服务器的 SSL 证书中获得[公钥]。）
5. **私钥被使用**：服务器对预主密钥进行解密。
6. **生成会话密钥**：客户端和服务器均使用客户端随机数、服务器随机数和预主密钥生成会话密钥。双方应得到相同的结果。
7. **客户端就绪**：客户端发送一条“已完成”消息，该消息用会话密钥加密。
8. **服务器就绪**：服务器发送一条“已完成”消息，该消息用会话密钥加密。
9. **实现安全对称加密**：已完成握手，并使用会话密钥继续进行通信。

所有 TLS 握手均使用非对称加密（公钥和私钥），但并非全都会在生成会话密钥的过程中使用私钥。例如，Diffie-Hellman 握手过程如下：

1. **客户端问候（client hello）**：客户端发送客户端问候消息，内含协议版本、客户端随机数和密码套件列表。
2. **服务器问候（server hello）**：服务器以其 SSL 证书、其选定的密码套件和服务器随机数回复。与上述 RSA 握手相比，服务器在此消息中还包括以下内容（步骤 3）：
3. **服务器的数字签名**：服务器对到此为止的所有消息计算出一个数字签名。
4. **数字签名确认**：客户端验证服务器的数字签名，确认服务器是它所声称的身份。
5. **客户端 DH 参数**：客户端将其 DH 参数发送到服务器。
6. **客户端和服务器计算预主密钥**：客户端和服务器使用交换的 DH 参数分别计算匹配的预主密钥，而不像 RSA 握手那样由客户端生成预主密钥并将其发送到服务器。
7. **创建会话密钥**：与 RSA 握手中一样，客户端和服务器现在从预主密钥、客户端随机数和服务器随机数计算会话密钥。
8. **客户端就绪**：与 RSA 握手相同。
9. **服务器就绪**
10. **实现安全对称加密**

密码套件

密码套件（Cipher suite）是传输层安全（TLS）/安全套接字层（SSL）网络协议中的一个概念。在TLS 1.3之前，密码套件的名称是以协商安全设置时使用的身份验证、加密、消息认证码（MAC）和密钥交换算法组成。使用的TLS协议不通所支持的密码套件也不相同。

通常当我们需要调整密码套件参数时可能是由于漏洞和业务需求导致，如果不明白怎么调整很容易出现问题。

密码套件命名规则

每个密码套件都有一个唯一的名称，用于识别它并描述它的算法内容。密码套件名称中的每个段代表不同的算法或协议。密码套件名称示例: **TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256**

这个名字的含义是：

TLS:定义该密码套件适用的协议；通常是 TLS。

ECDHE:指示正在使用的密钥交换算法。

RSA:握手期间的身份验证机制

AES_128_GCM:批量加密使用的会话密钥算法。

SHA256:签名机制。消息认证算法，用于对消息进行认证。摘要大小256 bits

TLS 1.0–1.2 密码套件支持的算法

Algorithms supported in TLS 1.0–1.2 cipher suites

Key exchange/agreement	Authentication	Block/stream ciphers	Message authentication
RSA	RSA	RC4	Hash-based MD5
Diffie–Hellman	DSA	Triple DES	SHA hash function (SHA-1 and SHA-2)
ECDH	ECDSA	AES (128-bits and 256-bits)	
SRP		IDEA	
PSK[10]		DES	
		Camellia	
		ChaCha20	

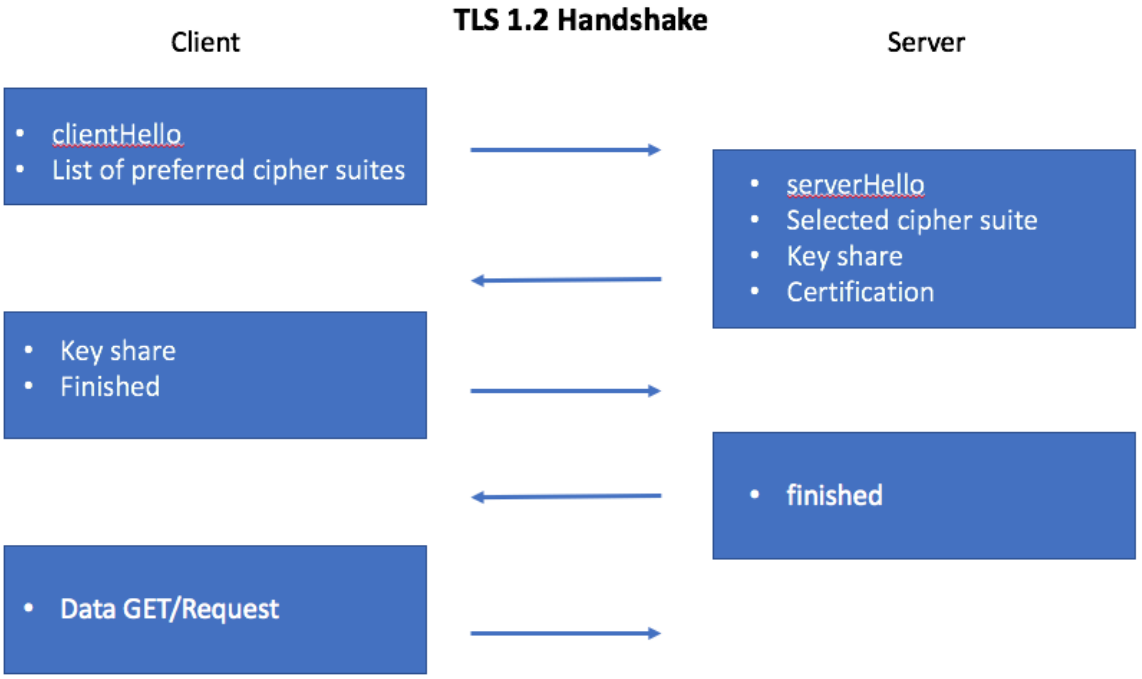
TLS 1.3

在 TLS 1.3 中，早期版本的 TLS 支持的许多旧算法已被删除，以使协议更加安全。 此外，所有加密和身份验证算法都组合在带有关联数据的身份验证加密 (AEAD) 加密算法中。此外，现在必须在基于 HMAC 的密钥派生 (HKDF) 中使用哈希算法。 由于可能出现的弱点或漏洞，所有非 AEAD 密码均已被删除，并且密码必须使用临时密钥交换算法，以便每次交换都会生成新的密钥对。在TLS1.3 中强制实现了向前保密，相反，RSA 没有前向保密；在私钥受损的情况下，攻击者可以确定过去对话的会话密钥，因为他们可以解密明文形式的预主密钥以及客户端随机数和服务器随机数。通过将这三者结合起来，攻击者可以得到任何指定的会话密钥。

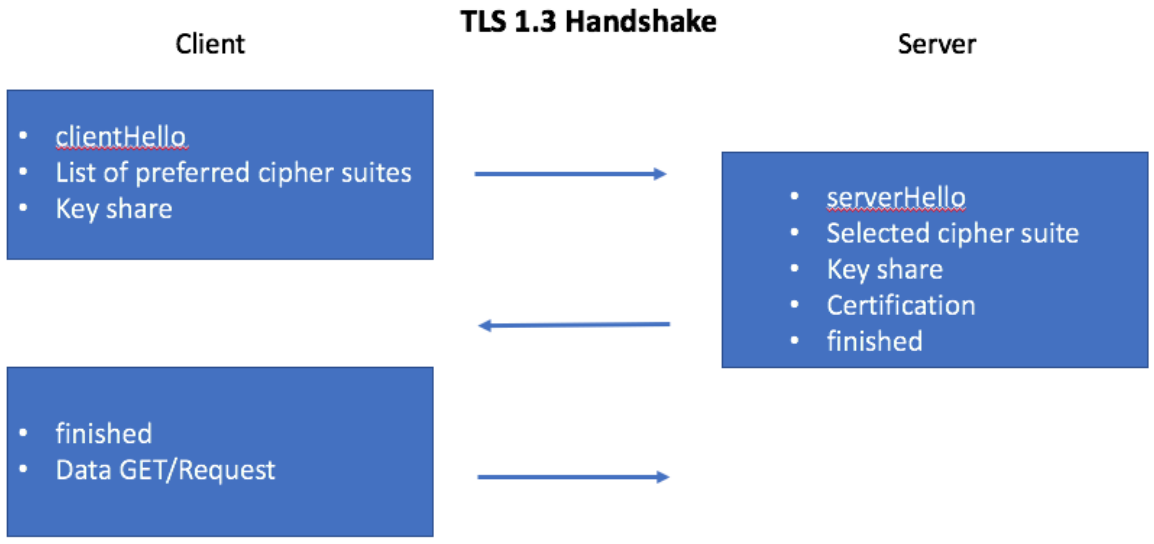
密码套件协商

在TLS 协议握手时进行密码套件协商，协商一致会建立TLS会话，如果协商不一致连接会被终止，这样情况在数据包里很常见。

TLS 1.0–1.2 handshake



TLS 1.3 handshake



openssl 查看支持的密码套件

```
[root@localhost ~]# openssl ciphers -v 'DEFAULT'
ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AESGCM(256)
Mac=AEAD
ECDHE-RSA-AES256-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  Enc=AES(256)  Mac=SHA384
ECDHE-ECDSA-AES256-SHA384 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AES(256)  Mac=SHA384
ECDHE-RSA-AES256-SHA    SSLv3 Kx=ECDH     Au=RSA  Enc=AES(256)  Mac=SHA1
ECDHE-ECDSA-AES256-SHA  SSLv3 Kx=ECDH     Au=ECDSA Enc=AES(256)  Mac=SHA1
DH-DSS-AES256-GCM-SHA384 TLSv1.2 Kx=DH/DSS   Au=DH   Enc=AESGCM(256) Mac=AEAD
DHE-DSS-AES256-GCM-SHA384 TLSv1.2 Kx=DH       Au=DSS  Enc=AESGCM(256) Mac=AEAD
DH-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=DH/RSA   Au=DH   Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=DH       Au=RSA  Enc=AESGCM(256) Mac=AEAD
DHE-RSA-AES256-SHA256   TLSv1.2 Kx=DH       Au=RSA  Enc=AES(256)  Mac=SHA256
DHE-DSS-AES256-SHA256   TLSv1.2 Kx=DH       Au=DSS  Enc=AES(256)  Mac=SHA256
DH-RSA-AES256-SHA256    TLSv1.2 Kx=DH/RSA   Au=DH   Enc=AES(256)  Mac=SHA256
DH-DSS-AES256-SHA256    TLSv1.2 Kx=DH/DSS   Au=DH   Enc=AES(256)  Mac=SHA256
```

```
DHE-RSA-AES256-SHA      SSLv3 Kx=DH       Au=RSA  Enc=AES(256)  Mac=SHA1
DHE-DSS-AES256-SHA      SSLv3 Kx=DH       Au=DSS  Enc=AES(256)  Mac=SHA1
DH-RSA-AES256-SHA       SSLv3 Kx=DH/RSA   Au=DH   Enc=AES(256)  Mac=SHA1
DH-DSS-AES256-SHA       SSLv3 Kx=DH/DSS   Au=DH   Enc=AES(256)  Mac=SHA1
DHE-RSA-CAMELLIA256-SHA SSLv3 Kx=DH       Au=RSA  Enc=Camellia(256) Mac=SHA1
DHE-DSS-CAMELLIA256-SHA SSLv3 Kx=DH       Au=DSS  Enc=Camellia(256) Mac=SHA1
DH-RSA-CAMELLIA256-SHA  SSLv3 Kx=DH/RSA   Au=DH   Enc=Camellia(256) Mac=SHA1
DH-DSS-CAMELLIA256-SHA  SSLv3 Kx=DH/DSS   Au=DH   Enc=Camellia(256) Mac=SHA1
ECDH-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH/RSA Au=ECDH Enc=AESGCM(256) Mac=AEAD
ECDH-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH/ECDSA Au=ECDH Enc=AESGCM(256)
Mac=AEAD
ECDH-RSA-AES256-SHA384  TLSv1.2 Kx=ECDH/RSA Au=ECDH Enc=AES(256)  Mac=SHA384
ECDH-ECDSA-AES256-SHA384 TLSv1.2 Kx=ECDH/ECDSA Au=ECDH Enc=AES(256)  Mac=SHA384
ECDH-RSA-AES256-SHA     SSLv3 Kx=ECDH/RSA Au=ECDH Enc=AES(256)  Mac=SHA1
ECDH-ECDSA-AES256-SHA   SSLv3 Kx=ECDH/ECDSA Au=ECDH Enc=AES(256)  Mac=SHA1
AES256-GCM-SHA384       TLSv1.2 Kx=RSA      Au=RSA  Enc=AESGCM(256) Mac=AEAD
AES256-SHA256           TLSv1.2 Kx=RSA      Au=RSA  Enc=AES(256)  Mac=SHA256
AES256-SHA              SSLv3 Kx=RSA      Au=RSA  Enc=AES(256)  Mac=SHA1
CAMELLIA256-SHA         SSLv3 Kx=RSA      Au=RSA  Enc=Camellia(256) Mac=SHA1
PSK-AES256-CBC-SHA      SSLv3 Kx=PSK      Au=PSK  Enc=AES(256)  Mac=SHA1
ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(128) Mac=AEAD
ECDHE-ECDSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AESGCM(128)
Mac=AEAD
ECDHE-RSA-AES128-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  Enc=AES(128)  Mac=SHA256
ECDHE-ECDSA-AES128-SHA256 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AES(128)  Mac=SHA256
ECDHE-RSA-AES128-SHA    SSLv3 Kx=ECDH     Au=RSA  Enc=AES(128)  Mac=SHA1
ECDHE-ECDSA-AES128-SHA  SSLv3 Kx=ECDH     Au=ECDSA Enc=AES(128)  Mac=SHA1
DH-DSS-AES128-GCM-SHA256 TLSv1.2 Kx=DH/DSS   Au=DH   Enc=AESGCM(128) Mac=AEAD
DHE-DSS-AES128-GCM-SHA256 TLSv1.2 Kx=DH       Au=DSS  Enc=AESGCM(128) Mac=AEAD
DH-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=DH/RSA   Au=DH   Enc=AESGCM(128) Mac=AEAD
DHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=DH       Au=RSA  Enc=AESGCM(128) Mac=AEAD
DHE-RSA-AES128-SHA256   TLSv1.2 Kx=DH       Au=RSA  Enc=AES(128)  Mac=SHA256
DHE-DSS-AES128-SHA256   TLSv1.2 Kx=DH       Au=DSS  Enc=AES(128)  Mac=SHA256
DH-RSA-AES128-SHA256    TLSv1.2 Kx=DH/RSA   Au=DH   Enc=AES(128)  Mac=SHA256
DH-DSS-AES128-SHA256    TLSv1.2 Kx=DH/DSS   Au=DH   Enc=AES(128)  Mac=SHA256
DHE-RSA-AES128-SHA      SSLv3 Kx=DH       Au=RSA  Enc=AES(128)  Mac=SHA1
DHE-DSS-AES128-SHA      SSLv3 Kx=DH       Au=DSS  Enc=AES(128)  Mac=SHA1
DH-RSA-AES128-SHA       SSLv3 Kx=DH/RSA   Au=DH   Enc=AES(128)  Mac=SHA1
DH-DSS-AES128-SHA       SSLv3 Kx=DH/DSS   Au=DH   Enc=AES(128)  Mac=SHA1
DHE-RSA-SEED-SHA        SSLv3 Kx=DH       Au=RSA  Enc=SEED(128) Mac=SHA1
DHE-DSS-SEED-SHA        SSLv3 Kx=DH       Au=DSS  Enc=SEED(128) Mac=SHA1
DH-RSA-SEED-SHA         SSLv3 Kx=DH/RSA   Au=DH   Enc=SEED(128) Mac=SHA1
DH-DSS-SEED-SHA         SSLv3 Kx=DH/DSS   Au=DH   Enc=SEED(128) Mac=SHA1
DHE-RSA-CAMELLIA128-SHA SSLv3 Kx=DH       Au=RSA  Enc=Camellia(128) Mac=SHA1
DHE-DSS-CAMELLIA128-SHA SSLv3 Kx=DH       Au=DSS  Enc=Camellia(128) Mac=SHA1
DH-RSA-CAMELLIA128-SHA  SSLv3 Kx=DH/RSA   Au=DH   Enc=Camellia(128) Mac=SHA1
DH-DSS-CAMELLIA128-SHA  SSLv3 Kx=DH/DSS   Au=DH   Enc=Camellia(128) Mac=SHA1
ECDH-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH/RSA Au=ECDH Enc=AESGCM(128) Mac=AEAD
ECDH-ECDSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH/ECDSA Au=ECDH Enc=AESGCM(128)
Mac=AEAD
ECDH-RSA-AES128-SHA256  TLSv1.2 Kx=ECDH/RSA Au=ECDH Enc=AES(128)  Mac=SHA256
ECDH-ECDSA-AES128-SHA256 TLSv1.2 Kx=ECDH/ECDSA Au=ECDH Enc=AES(128)  Mac=SHA256
ECDH-RSA-AES128-SHA     SSLv3 Kx=ECDH/RSA Au=ECDH Enc=AES(128)  Mac=SHA1
ECDH-ECDSA-AES128-SHA   SSLv3 Kx=ECDH/ECDSA Au=ECDH Enc=AES(128)  Mac=SHA1
AES128-GCM-SHA256       TLSv1.2 Kx=RSA      Au=RSA  Enc=AESGCM(128) Mac=AEAD
AES128-SHA256           TLSv1.2 Kx=RSA      Au=RSA  Enc=AES(128)  Mac=SHA256
```

```
AES128-SHA              SSLv3 Kx=RSA      Au=RSA  Enc=AES(128)  Mac=SHA1
SEED-SHA                SSLv3 Kx=RSA      Au=RSA  Enc=SEED(128) Mac=SHA1
CAMELLIA128-SHA         SSLv3 Kx=RSA      Au=RSA  Enc=Camellia(128) Mac=SHA1
PSK-AES128-CBC-SHA      SSLv3 Kx=PSK      Au=PSK  Enc=AES(128)  Mac=SHA1
ECDHE-RSA-DES-CBC3-SHA  SSLv3 Kx=ECDH     Au=RSA  Enc=3DES(168) Mac=SHA1
ECDHE-ECDSA-DES-CBC3-SHA SSLv3 Kx=ECDH     Au=ECDSA Enc=3DES(168) Mac=SHA1
EDH-RSA-DES-CBC3-SHA    SSLv3 Kx=DH       Au=RSA  Enc=3DES(168) Mac=SHA1
EDH-DSS-DES-CBC3-SHA    SSLv3 Kx=DH       Au=DSS  Enc=3DES(168) Mac=SHA1
DH-RSA-DES-CBC3-SHA     SSLv3 Kx=DH/RSA   Au=DH   Enc=3DES(168) Mac=SHA1
DH-DSS-DES-CBC3-SHA     SSLv3 Kx=DH/DSS   Au=DH   Enc=3DES(168) Mac=SHA1
ECDH-RSA-DES-CBC3-SHA   SSLv3 Kx=ECDH/RSA Au=ECDH Enc=3DES(168) Mac=SHA1
ECDH-ECDSA-DES-CBC3-SHA SSLv3 Kx=ECDH/ECDSA Au=ECDH Enc=3DES(168) Mac=SHA1
DES-CBC3-SHA            SSLv3 Kx=RSA      Au=RSA  Enc=3DES(168) Mac=SHA1
IDEA-CBC-SHA            SSLv3 Kx=RSA      Au=RSA  Enc=IDEA(128) Mac=SHA1
PSK-3DES-EDE-CBC-SHA    SSLv3 Kx=PSK      Au=PSK  Enc=3DES(168) Mac=SHA1
KRB5-IDEA-CBC-SHA       SSLv3 Kx=KRB5     Au=KRB5 Enc=IDEA(128) Mac=SHA1
KRB5-DES-CBC3-SHA       SSLv3 Kx=KRB5     Au=KRB5 Enc=3DES(168) Mac=SHA1
KRB5-IDEA-CBC-MD5       SSLv3 Kx=KRB5     Au=KRB5 Enc=IDEA(128) Mac=MD5
KRB5-DES-CBC3-MD5       SSLv3 Kx=KRB5     Au=KRB5 Enc=3DES(168) Mac=MD5
ECDHE-RSA-RC4-SHA       SSLv3 Kx=ECDH     Au=RSA  Enc=RC4(128)  Mac=SHA1
ECDHE-ECDSA-RC4-SHA     SSLv3 Kx=ECDH     Au=ECDSA Enc=RC4(128)  Mac=SHA1
ECDH-RSA-RC4-SHA        SSLv3 Kx=ECDH/RSA Au=ECDH Enc=RC4(128)  Mac=SHA1
ECDH-ECDSA-RC4-SHA      SSLv3 Kx=ECDH/ECDSA Au=ECDH Enc=RC4(128)  Mac=SHA1
RC4-SHA                 SSLv3 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=SHA1
RC4-MD5                 SSLv3 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=MD5
PSK-RC4-SHA             SSLv3 Kx=PSK      Au=PSK  Enc=RC4(128)  Mac=SHA1
KRB5-RC4-SHA            SSLv3 Kx=KRB5     Au=KRB5 Enc=RC4(128)  Mac=SHA1
KRB5-RC4-MD5            SSLv3 Kx=KRB5     Au=KRB5 Enc=RC4(128)  Mac=MD5
[root@localhost ~]#
```

正如您所看到的，这启用了更多的密码，包括许多不安全的密码例如:RC4 ,MD5,DES等。

随便找一个示例看下:

```
RC4-MD5                 SSLv3 Kx=RSA      Au=RSA  Enc=RC4(128)  Mac=MD5
```

这些列是:

密码套件: RC4-MD5 (**为什么这么短?** 定义中如果没指定默认就是RSA)

协议: SSLv3

密钥交换算法（Kx）: RSA

认证算法（Au）： RSA

密码/加密算法 (Enc): RC4(128)

MAC :MD5

配置示例

通常在配置SSL 时需要考虑网站服务的兼容性，像 百度这种网站SSL 安全等级是C。需要兼容IE6 这种老古董。但是对于兼容性没这么变态要求的可以参考如下配置:

```
# generated 2024-09-14, Mozilla Guideline v5.7, nginx 1.17.7, OpenSSL 1.1.1k,
intermediate configuration
```

```
# https://ssl-
config.mozilla.org/#server=nginx&version=1.17.7&config=intermediate&openssl=1.1.1
k&guideline=5.7
server {
    listen 80 default_server;
    listen [::]:80 default_server;

    location / {
        return 301 https://$host$request_uri;
    }
}

server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2;

    ssl_certificate /path/to/signed_cert_plus_intermediates;
    ssl_certificate_key /path/to/private_key;
    ssl_session_timeout 1d;
    ssl_session_cache shared:MozSSL:10m;  # about 40000 sessions
    ssl_session_tickets off;

    # curl https://ssl-config.mozilla.org/ffdhe2048.txt > /path/to/dhparam
    ssl_dhparam /path/to/dhparam;

    # intermediate configuration
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-
ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-
POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-
GCM-SHA384:DHE-RSA-CHACHA20-POLY1305;
    ssl_prefer_server_ciphers off;

    # HSTS (ngx_http_headers_module is required) (63072000 seconds)
    add_header Strict-Transport-Security "max-age=63072000" always;

    # OCSP stapling
    ssl_stapling on;
    ssl_stapling_verify on;

    # verify chain of trust of OCSP response using Root CA and Intermediate certs
    ssl_trusted_certificate /path/to/root_CA_cert_plus_intermediates;

    # replace with the IP address of your resolver
    resolver 127.0.0.1;
}
```

在配置文件中开启了http2， http2 只能基于TLS运行。

http2 和TLS

HTTP/2 和 HTTP/1.1 之间有哪些影响性能的区别?

多路复用：HTTP/1.1 依次加载各个资源，因此，如果无法加载某一资源，它将阻碍其后的所有其他资源。相比之下，HTTP/2 可以使用单个 TCP 连接来一次发送多个数据流，使得任何资源都不会会阻碍其他资源。为此，HTTP/2 将数据拆分为二进制代码消息并为这些消息编号，以便客户端知道每个二进制消息所属的流。

服务器推送：通常，服务器仅在客户端要求时才向客户端设备提供内容。但是，这种方法并不总是适用于现代网页，因为现代网页通常涉及客户端必须请求的数十个独立资源。HTTP/2 通过允许服务器在客户端请求之前向客户端"推送"内容来解决此问题。服务器还发送一条消息，让客户知道预期推送的内容是什么，就像 Bob 在发送整本书之前向 Alice 发送小说目录一样。

标头压缩：小文件的加载速度比大文件快。为了提高 Web 性能，HTTP/1.1 和 HTTP/2 都会压缩 HTTP 消息以使其更小。但是，HTTP/2 使用一种称为 HPACK 的更高级压缩方法，可以消除 HTTP 标头数据包中的多余信息。这样可以从每个 HTTP 数据包中消除几个字节。考虑到即使只加载一个网页时所涉及的 HTTP 数据包的数量，这些字节会迅速累加，从而加快了加载速度。

TLS 1.3 和TLS 1.2 的比较?

TLS 1.3 比 TLS 1.2 更快、更安全。使 TLS 1.3 更快的一处更改是对 TLS 握手工作方式的更新：TLS 1.3 中的 TLS 握手只需要一次往返（或来回通信）而不是两次，从而将过程缩短了几毫秒。如果客户端之前连接到网站，TLS 握手的往返次数为零。这使 HTTPS 连接更快，减少延迟并改善整体用户体验。

TLS 1.2中的许多主要漏洞与仍受到支持的较旧的加密算法有关。TLS 1.3放弃了对这些易受攻击的加密算法的支持，因此，它不太容易受到网络攻击。

几乎所有浏览器都支持 HTTP/2 — Chrome、Firefox、Opera、Safari（包含针对特定操作系统版本）甚至于 IE。

注意：HTTP/2 仅通过 TLS 实现。因此以后TLS只会变得越来越多，越来越重要。

工具

使用openssl 自签发证书

```
.ssl 证书加密文件

    ******************************
        建立私有CA
        openCA
        openssl
    证书申请及签署步骤
        1.生成证书申请请求
        2.RA 效验
        3.CA 签署
        4. 获取证书
    openssl  默认配置文件：/etc/pki/tls/openssl.cnf
        1.创建所需要的文件
          文件serial和index.txt分别用于存放下一个证书的序列号和证书信息数据库。
          文件serial填写第一个证书序列号（如10000001），之后每前一张证书，序列号自动加1。
            touch index.txt
            echo 01 > serial
        2.CA 自签发证书

        生成私钥：
            (umask 077; openssl genrsa -out private/cakey.pem 2048)

        生成新证书：
```

```
            openssl  req -new -x509 -key /etc/pki/CA/private/cakey.pem -days 7300
-out cacert.pem
            -new :生成新的证书签署文件
            -x509 :专用于CA 自签发证书
            -key :私钥
            -days : 有效期
            -out : 证书的保存路径
        然后根据信息填完可以生成新的证书了

        3.发证
            1.主机生成证书请求文件
            2.将请求文件发给CA
            3.CA 效验请求文件
            4.签发

                给httpd 服务器签发证书。
                创建私钥:
                (umask 077;openssl genrsa -out httpd.key 2048)
                创建证书请求文件
                openssl req -new -key httpd.key -days 365 -out httpd.csr
                签发证书
                openssl ca -in /etc/httpd/ssl/httpd.csr -out httpd.crt -days 365
                查看证书中的信息:
                openssl x509  -in /path/cert_file -noout -text
```

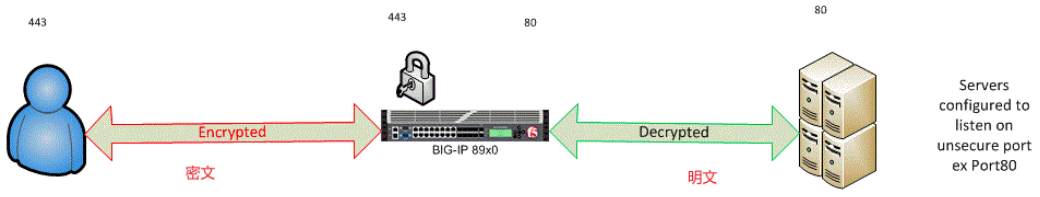
可参考

解密TLS流量

TLS 部署持续增长，加密流量也越来越多。安全设备和运维人员对于流量解密的需求也更加迫切，但是在日常运维中，手动解密对于不了解TLS 机制的人员来说不是一件很简单的事。

这里分别介绍下tls1.2解密RSA和ECDHE 密钥交换算法下的加密流量:

拓扑:

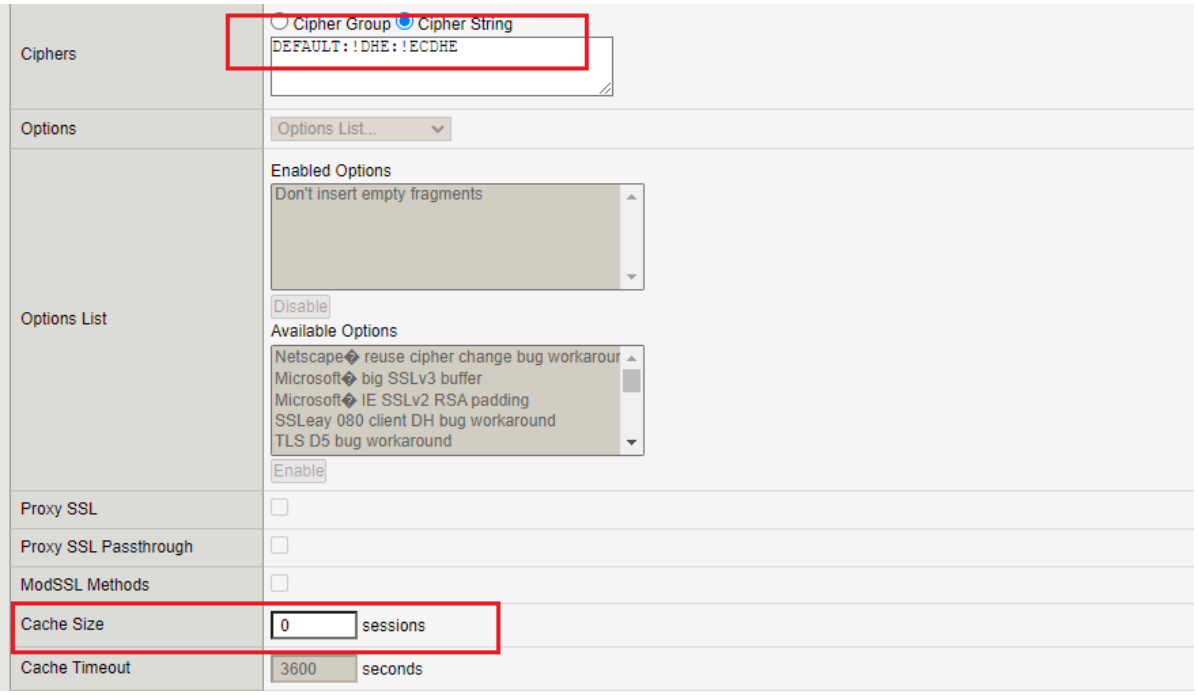


前置条件:

- 1.熟练使用wireshark ,支持tls 解密
- 2.必须捕获完整的 TLS 握手
- 3.对TLS 私钥的完全管理权限
- 4.捕获的数据包必须有完整的 TLS 握手信息

RSA密钥交换算法解密

1.禁用SSL会话缓存并优先启用 RSA 密钥交换算法



2.使用tcpdump 捕获数据包

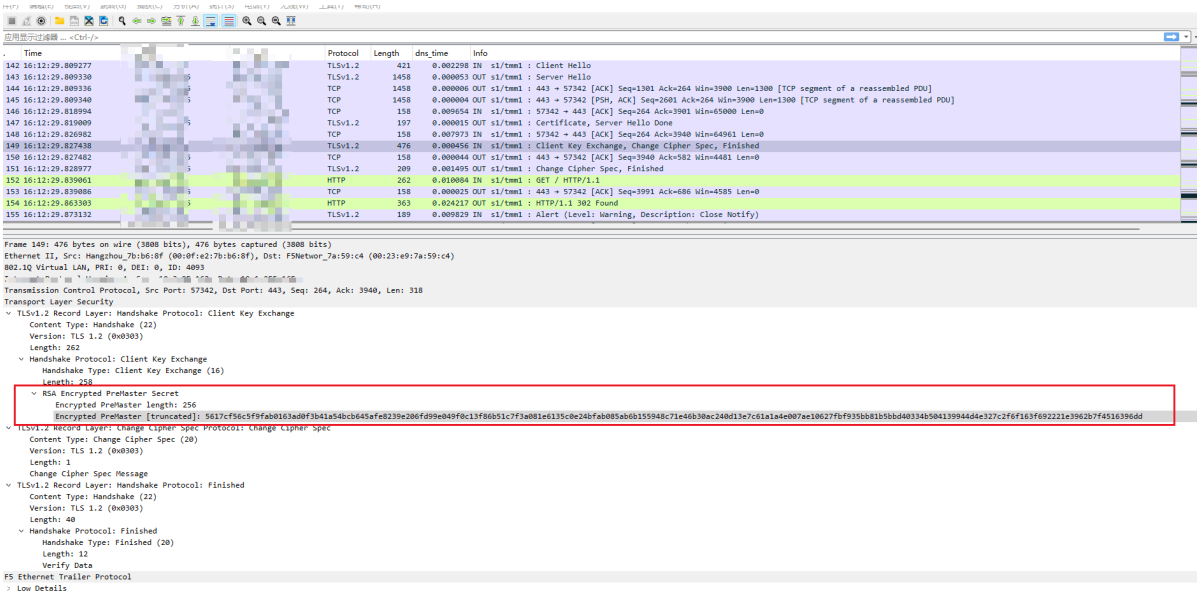
```
tcpdump -ni 0.0:nnn -s0 host x.x.x.x and port 443 -w
/root/165_443_Decrypting_RSA3.cap -vvv

curl -ik https://x.x.x.x
-i 参数告诉 curl 显示 HTTP 响应的头部信息。
-k 或 --insecure 参数让 curl 命令在与服务器通信时不进行 SSL 证书验证。
```

3.导入key 解密RSA数据包

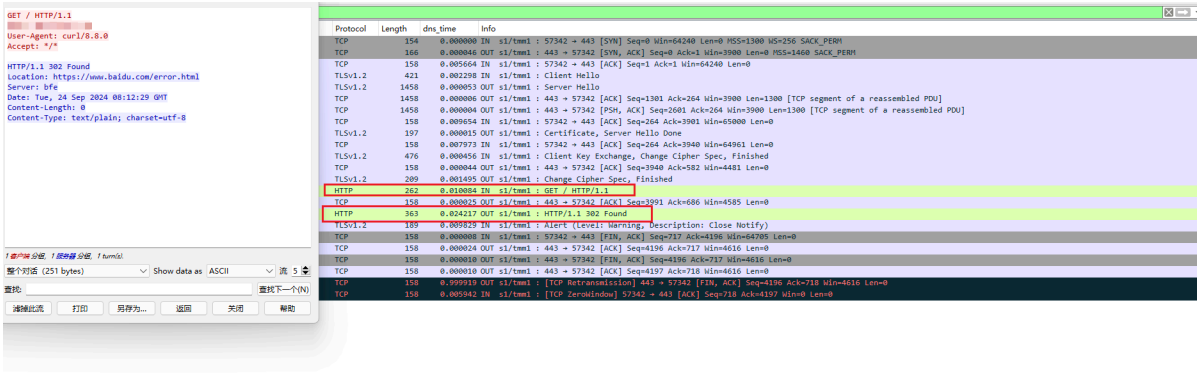
确认在密码套件协商时使用了Cipher Suite:**TLS_RSA_WITH_AES_128_GCM_SHA256**

```
Transport Layer Security
  TLSv1.2 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 53
    Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 49
        Version: TLS 1.2 (0x0303)
      > Random: 036554bb8cd25748ffb5bc0f0bc6159beada19fc2f6f30d3ad4ecb2bee3ad4d3
        Session ID Length: 0
        Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
        Compression Method: null (0)
        Extensions Length: 9
      > Extension: renegotiation_info (len=1)
          Type: renegotiation_info (65281)
          Length: 1
        > Renegotiation Info extension
      > Extension: extended_master_secret (len=0)
          Type: extended_master_secret (23)
          Length: 0
        [JA3S Fullstring: 771,156,65281-23]
        [JA3S: de23ddf1eedc35f4f513dc76eeb1824a]
    TLS segment data (1242 bytes)
F5 Ethernet Trailer Protocol
  > Low Details
  > Medium Details
```

这里直接导入私钥文件是因为RSA 密钥交换只需要私钥解密这里的**Encryted PerMaster**就可以



在wireshark 单击 Wireshark → 首选项→**Protocols → TLS → RSA keys list**我们在其中看到一个窗口，我们可以在Key File 字段中引用负载或服务器的私钥。我们可以看到 HTTP 解密流量（绿色）：

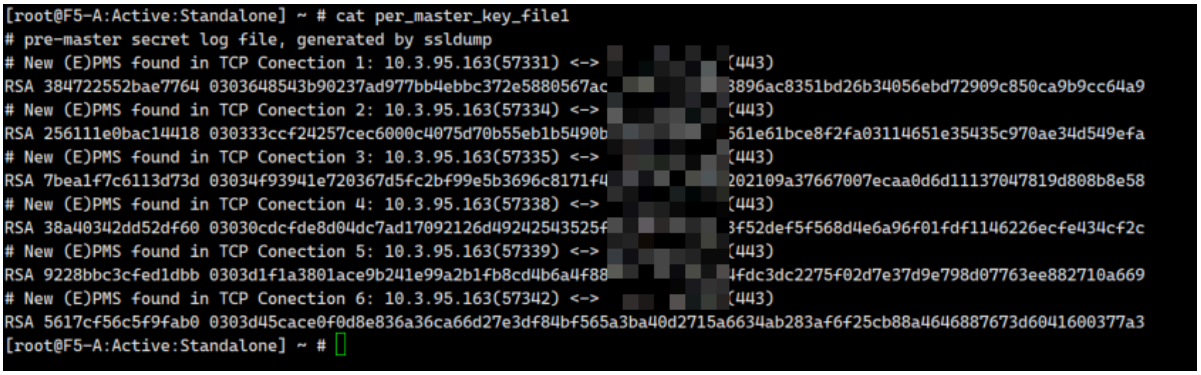


在生产环境中，我们通常会避免将私钥复制到不同的计算机，因此另一种选择是直接在我们尝试捕获的服务器上使用 ssldump 命令。

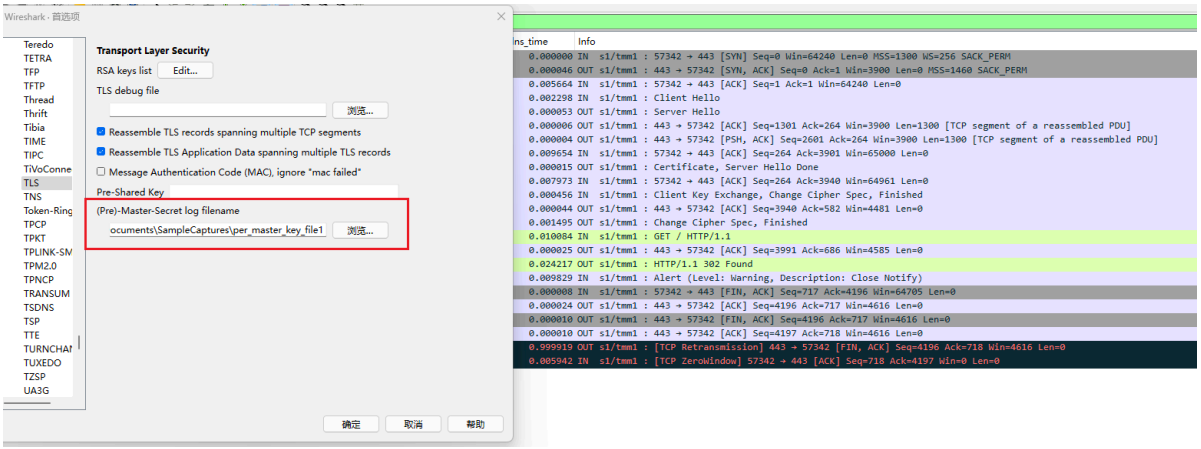
```
Syntax: ssldump -r <capture.pcap> -k <private key.key> -M <type a name for your ssldump file here.pms>
```

```
ssldump -r 165_443_Decrypting_RSA3.cap    -k cert_ssl.key -M per_master_key_file
```

导出加密数据包的预主密钥(可以将这个文件导入到wireahrk 中，就不用导入私钥文件):



Wireshark → Preferences → Protocols → TLS → Pre-Master Key logfile name导入预主密钥日志文件即可:



ECDHE密钥交换算法解密

针对ECDHE F5 的办法通过irules 打印 SSL 会话对应日志实现:

针对开启和禁用SSL 会话有两种脚本:

SSL session cache enabled

```
when CLIENTSSL_HANDSHAKE {
      if { [IP::addr [getfield [IP::client_addr] "%" 1] equals <client_IP_addr>]
} {
        log local0. "[TCP::client_port] :: RSA Session-ID:[SSL::sessionid]
Master-Key:[SSL::sessionsecret]"
      }
    }
```

SSL session cache disabled

```
when CLIENTSSL_HANDSHAKE {
 if {[IP::addr [getfield [IP::client_addr] "%" 1] equals <client_IP_addr>] } {
   log local0. "CLIENT_Side_IP:TCP source port: [IP::client_addr]:
[TCP::remote_port]"
   log local0. "CLIENT_RANDOM [SSL::clientrandom] [SSL::sessionsecret]"
   log local0. "RSA Session-ID:[SSL::sessionid] Master-Key:[SSL::sessionsecret]"
 }
}
```

后台过滤数据的日志:

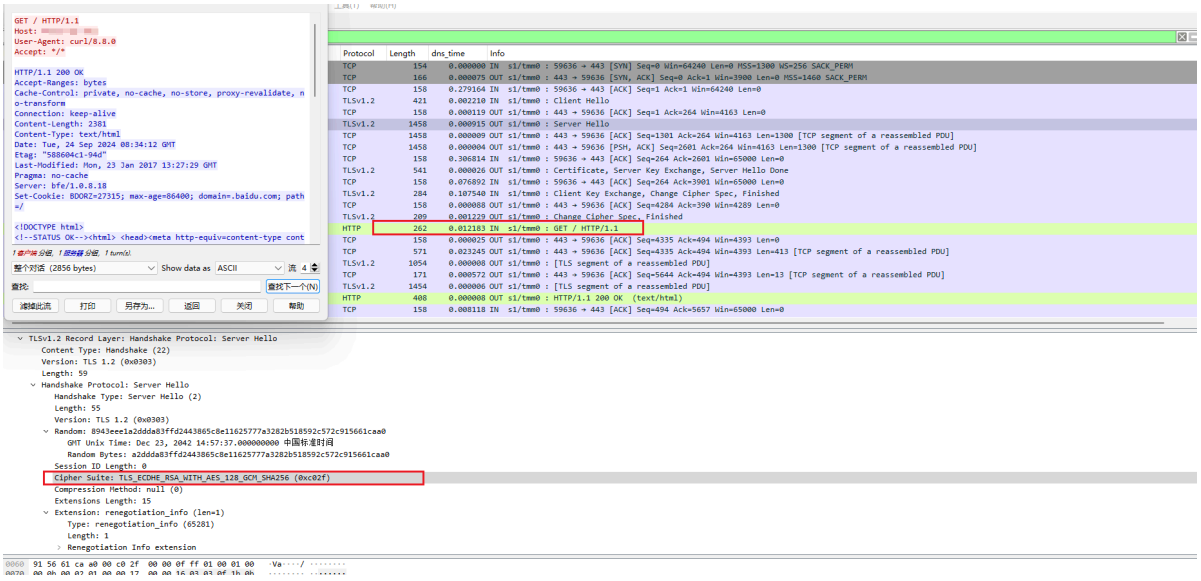
```
grep -h -o 'CLIENT_RANDOM.*' /var/log/ltm* > /var/tmp/sessionsecrets.pms
```

每个会话都生成一条预祝密钥和随机数如下:

```
CLIENT_RANDOM 6b04ae23470db1b1648279997761b97599a9cb47ccac10357a388992793b2619b ffa1d2bde14fc8e7d7a98c0b2ff0e01eb95f522703ba7fd28c4227348fc9b13484afd17db33a4b679047f2bf551b1196
CLIENT_RANDOM 05e82f00fb233b67e19761b6ab49f4ebc4e2f610cfca6f5070addcca72a5e67e b1e508793a9c5368484840db450a773e76fe26ab7b06e584df8e8acc06d1b677e9361bdcd84e2084518b957153722a07
CLIENT_RANDOM 0f2656362cb21246128c089d12cc3afe9977f2f82d3064fad965f27271a5926d 0913eb8d3801ca8c5a84944ce8ec90a19556f54df0583c6f1fbd388228358e7766013d7c2b0037a87c4cbf466bd32c62
CLIENT_RANDOM e6c4e9a8a2e9aad59782e00d9b764ce7e5324f1c9b1578b09bd3af58198cbeaa 8b05d974fdfa2ac2707f556e76be7beb5c9d65486ba66bc29be0ea0d0ea96fcadd839844ab6072b77bbb831adb377513
CLIENT_RANDOM 9b7dbe01170a86657136ed1e4c7561c6bfdfd16f29e7fd9e56bc51511305de80 600584e2288d672eec5f50798b6c8ac170a90cf7547b55bbd98ef85769a18fd1c4060d19fb8e5bd31be9228878c06ade
CLIENT_RANDOM f5fae5d5c11a70e8d4a6da508d9e71d5539fba6da8a803656cf11e8ead56ad7e 028c76cebf2db56c8032e20d313cf5e5a7d31bebe9a3c4ff9f5d396ef056f21f341931d2e4e1122d0e1fc10b8b9387f7
CLIENT_RANDOM 842949ff23370c0a7b6c85cf66d611afee181b2200bcb83d86aa66e2c855141b 8c21e6bb61b45140424b4e9510b7d55b05a1e0eaae68c2ba1d1105d8e404fc582912bfcacd89f21ce590179a0ae5d921
CLIENT_RANDOM 9f79ec1ac67165cc32239cc43500854ca730c3b317e886d752f5ea370d580d98 886c3b5b02154b89477e9ec81bfa471d23ac2bde8f400963200f3f9e4d2fb8a7d76216a6a1b9984d2d58e91a58608739
CLIENT_RANDOM 26cae6f78deea2ce8df81b16a0484c2ad82c8c349a217a61cc149da4b0ca7f42 1aa8031f4115dc2c914979491e744e65b83dafed21e638f3db7aac08bca09698c4bf51159fb030e7ec6601b24c17e793
CLIENT_RANDOM 91c906ee805d4a9215baea13a9918ce964b547798966eb832d680dd8814bf59b a06bb46f2af98a7b1d2025200dec37a36f4e6ee897f1d49977072628032e8f0ecc3f7dcb9285f235094437746ce7b130
CLIENT_RANDOM 91f5be14bd7ee3fda6e9a379f9268952cc2e07ddc6c6c644aae0a8c09fcfcc43 8a4bd82716aa79cd14da18e9d03bd495cc6cfa5f2865cdedc14a06ad9499e69601514f83085e27156f38deа8fa3a356f
CLIENT_RANDOM 0fe054a50e133eb6aa3f2c90893a61cc9a4fafaebe615862c3b8dec40f9307ba df59f85d6a39027067a53574e3609044a59786cf4e0257420ea80f1a82b4880b7ae4c7f46279a4a57d75b0028ef273c9
CLIENT_RANDOM 052bc7006333c9a81496511bd06165a91688cfea1a9736ee72782e44cdb4dd19 b6ec6bd5231bd2e22bfc4a369406108bd72e05ed22ac29a38afdc6f8d483476b7070f42fd320e78e9c603040d3861ced
CLIENT_RANDOM 484829d74c7f73c4066e524808251f3cb8f770ef8c95a98e181de7a629fcadeb 7c1959be15b869d868209ceaf9194c96d6442ec940d731f7fa123e83e8c67d7ea7f2f6216a8fe94d53afae9a722a27da
CLIENT_RANDOM cb1bde21f859d61fe4f956c2c79451df45a04eb36b4141277cdffe0d653ad8dc e5860eb1d9b92c8f7fce3ac87e5205f02a086b8d37c8e0677b15d61c75cf1c8a685fadce689d3c62a1b20f48f56fc538
CLIENT_RANDOM bc7d0ae07cb73e728186afccbb7e18edb356bdbc99d7b0f6e8c68c8b2238e4d8 5d8600aa425284cb958cea338d430c2525a2c2917e5650558280389 3f540f6f167a498e91df5385bf9bf306867b3bf4a
CLIENT_RANDOM 53210ebc8f94bc5e0548b0a0f1ae013c567b066a02db6d27baf730552cb54437 b7ed2a515401c8ae5cd1958fd884915da399eb178f3a8ce57765a62ed8fed3cb0ca20f04a3bdc44e70a9547520626588
CLIENT_RANDOM bf970a83acf2822515549f0508f0b0f023c25d9d32ac02e0de3bc6b0269da2bc a0e3848078e70d52fdc9d61b929a887cfb05014672c4c1ee0c5baa0deef3a6ff6ff88551608b3f9ce83b7ebb3d5c7b71
CLIENT_RANDOM a02c80865824124bb0b89d2a823e8b6bc31fcc5a38c68f30f1f34a86f4e2d6b6 5d8a6e2ca4b70be719f6851b27f22d7be183f7c70f2e748c75cb94cf2d98421f2a498ba57ec96b3c7a17103cd8b38a47
CLIENT_RANDOM 5ce8805a834934c3fe6c96a25877104e8c118b81c1722dfbc9bc553996b976d1 72ab14505396f2f69e511c3fd8176dbbeaffc1313044121eea01cf16d747e6fc248cd5887e9478a4fb61b7407689d2b3
CLIENT_RANDOM 02921c87966d085fae0b4a7c629f654f48004774eaa1dc973488d7f2a847c9f b770fda13e873bcee39ae840fb3362d302db2942221b85967e0db1f2872e735ac7978843e66e4b23aa9a753f54ba72f2
CLIENT_RANDOM e2aa4753273e5697e12a62fba318ace3d05420bc01cca715b31b75b8762c0459 4707107a51f74f4e46d09fe3d697edcb04cfae9da6c789b2a596b0175ed9c460fcc51a70a3324887092 8bf71d94176db
CLIENT_RANDOM 29b3503e279af60ec7c3531217ff5198317dc7041b7eca8c4c82cf14f920ce55 e27dcecaa926507e129e51c29a64fece55f6f94cfcfb6572ff2520d5ad435f0f616aeecfb1471794dcf0e0a2e8279539
CLIENT_RANDOM ae03f35e494c6c5349802d102076b16fd614b3ab2e97b3fcd02d24ff8b590094 3eb1f6d8519b0365b280faa1affd2e64a307e7fefcad935ce580c343721c8ec4b8e99bc64c992a5bb9f58ee0c429d09e
CLIENT_RANDOM 991d3250b3b5232fa6fc9d7d1e28f1090bb4fe142a259441c28349ff3ac810c9 80c2f0cfa3f7994e86adfbb0ce7d1120cba144170414b1e9d62aefa3faf1a6ce8ccb5708e037fc77d37177c166424f37
CLIENT_RANDOM bbd932e645035ac2ac81ad7e23410a90b775015f8c2deb1148aa20f81c33d2b0 87a8d703148112bc67e3d428a675cb9a442a26d1b0bc3df5518e1bf2fe690a4dc3804248f747d404707862666f168993
CLIENT_RANDOM 24d3fe9a8cf5a84bce2d601eb0c3d4e1f5d92a6f247add6cdee9017cea717cea d10efce23dcd40571e522116817560cd368257757986d0e72704dc6c8172d8b3732ac7044032b0868dc20f79c5bd948f
CLIENT_RANDOM cfba473cd164e109a7040537273225a80f5f06d56d5e350c835712fa7d2ebeff8 be444e9b8701db311857243053adad73fd2e94d4e8fdc6dae6c5ff94f85816f6fb28eaedcddc49da1647bde6c5cf19e
CLIENT_RANDOM 48af5c36ee2b46b030f57e9e4dba92f53d334ce9d3fbf90fa23a514cdd8b4d87 33d06d3b59fc0a34e76996e47cf9be80c82420abee3a39ad1f9f0449f6999011b8d8ad483f7acea6806f6065b393f705
```

在wireshark 单击 Wireshark → 首选项→**Protocols** → **TLS** → **RSA keys list** Pre)-Master-Secret log filename, **select** Browse** 导入essionsecrets.pms 文件如下:

在选择密码套件Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 情况下解密见(绿色):



注意:现在一些依赖**eBPF** 的工具也可以对ECDHE 解密。

常见的SSL/TLS 部署方式

在应用部署时我们通常有以下方式来部署SSL 证书:

- **1.服务端直接部署SSL进行卸载(前端无负载均衡设备)**
- **2.前端负载均衡SSL卸载**
- **3.前端负载均衡SSL透传**
- **4.前端负载均衡SSL Full proxy (或SSL 重新加密/SSL 桥接/SSL 终止等名称)**
- **5.前端负载均衡调用外部HSM进行SSL卸载(专用加密机)**
- **6.两层负载进行SSL卸载**

1.服务端直接部署SSL进行卸载(前端无负载均衡设备)

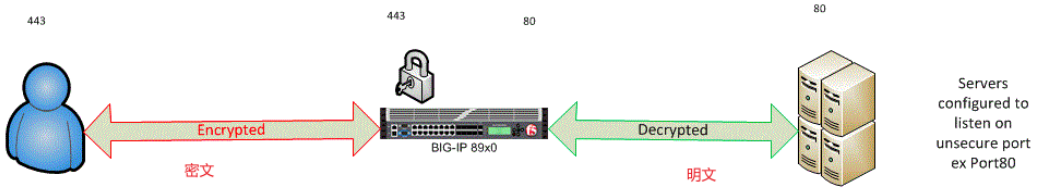
此场景中客户端和服务端直接建立SSL链接，SSL 卸载依赖服务端来处理。

Note:

- 1.客户端和服务器直接密文通信。
- 2.通常服务端使用443端口。
- 3.中间所有安全设备无法过滤分析加密流量。

2.前端负载均衡SSL卸载

在此场景中，到 负载均衡的客户端流量以加密方式发送。负载均衡将处理该部分，而不是由服务器解密和重新加密流量。因此，客户端流量由 负载均衡解密，并将解密后的流量发送到服务器。从服务器到客户端的返回数据由负载均衡加密并发送回客户端。从而减轻服务器额外的加密和解密负担。现在可以充分利用所有服务器资源来服务应用程序内容或任何其他目的。

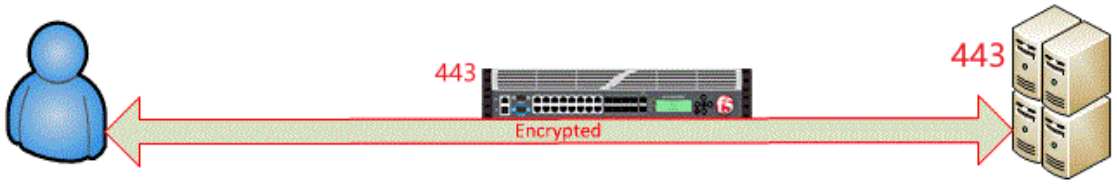


Note:

- 1.客户端和负载均衡密文通信，负载均衡和服务器明文通信。
- 2.通常负载均衡使用443端口转发到服务端80端口。
- 3.负载均衡之后的安全设备可以看到明文流量。

3.前端负载均衡SSL透传

顾名思义，负载均衡只会将流量从客户端传递到服务器，不会进行任何与 SSL 相关的工作负载。它不会直接进行SSL 握手，而是仅将客户端流量握手请求转发至服务器。



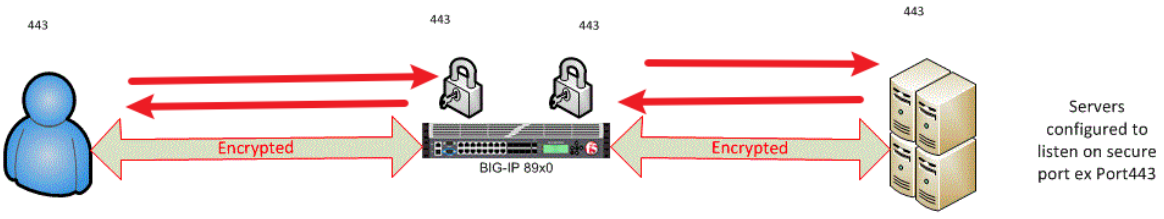
Note:

- 1.客户端和服务器密文通信。

- 2.通常负载均衡使用443端口转发到服务端443端口。
- 3.安全设备只能看到 密文流量。
- 4.负载均衡无法读取报头，这对会话保持有限制。只有数据包中的非 SSL 信息可用于保持持久性，如源 IP 地址、目标 IP 地址。

4.前端负载均衡SSL Full proxy

此方法有几个名称，例如 SSL 重新加密、SSL 桥接和 SSL 终止。在此方法中，负载均衡将在将流量发送到服务器之前重新加密流量。客户端将加密流量发送到负载均衡，负载均衡对其进行解密，并在将其发送到服务器或池成员之前再次对其进行重新加密。该方法通常用于安全级别较高的业务场景。

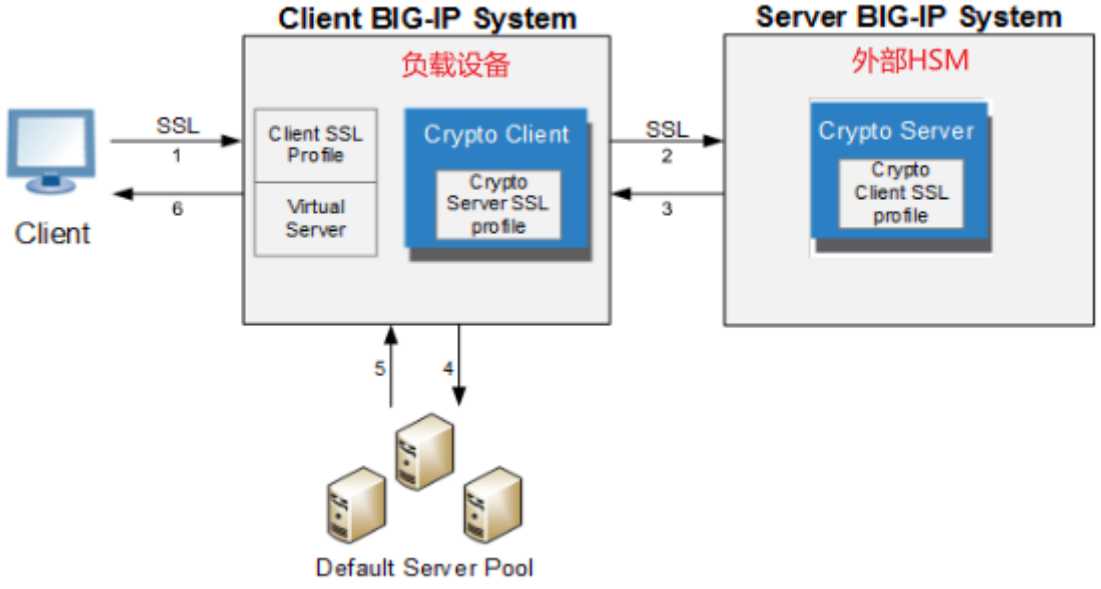


Note:

- 1.客户端和服务器全密文，只有中间负载设备内部处理时解密。
- 2.通常负载均衡使用443端口转发到服务端443端口。
- 3.安全设备只能看到 密文流量。
- 4.由于负载设备解密流量，因此它能够读取内容（报头、txt、cookie 等），并且所有会话保持选项都能应用。 （例如:源地址、目标地址、Cookie、SSL、SIP、通用、MSRDP）

5.前端负载均衡调用外部HSM进行SSL卸载(专用加密机)

此方法适用于对于SSL加解密需求特别大，安全要求特别高的场景。客户端密文请求，中间负载设备调用HSM的加解密功能进行SSL 交互和握手。解密后明文向后传输。



Note:

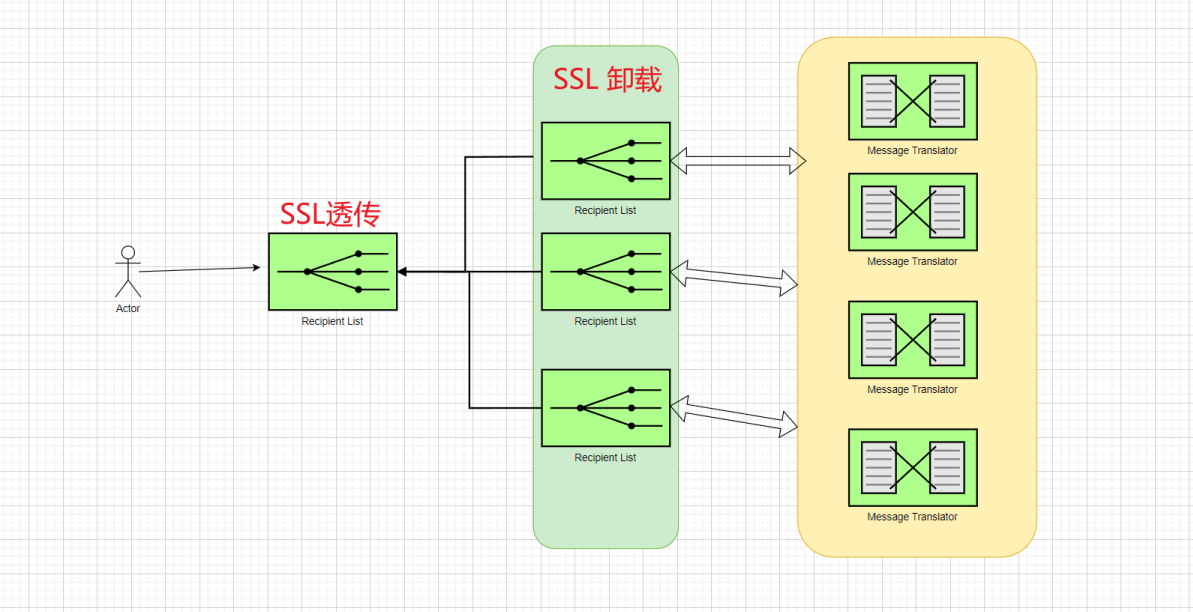
- 1.客户端和负载设备密文，负载设备和服务器明文。
- 2.通常负载均衡使用443端口转发到服务端80端口。
- 3.负载设备之下的安全设备能看到明文流量。

- 4.由于负载设备解密流量，因此它能够读取内容（报头、txt、cookie 等），并且所有会话保持选项都能应用。 （例如:源地址、目标地址、Cookie、SSL、SIP、通用、MSRDP）
- 5.HSM统一管理密钥，算法等功能，但是负载设备是否支持HSM 需要确认。

6.两层负载进行SSL卸载

此方法是两层架构，第一层进行SSL 透传将密文流量分别负载分担到不同的负载均衡设备，

第二层负载均衡设备进行SSL 卸载后将明文流量转发至服务器。



Note:

- 1.客户端到第二层负载设备密文，负载设备和服务器明文。
- 2.第一层负载443 到第二层负载443端口，服务端80端口。
- 3.第二层负载设备之下的安全设备能看到明文流量。
- 4.第一层负载均衡无法读取报头，这对会话保持有限制。只有数据包中的非 SSL 信息可用于保持持久性，如源 IP 地址、目标 IP 地址。
- 5.第二层负载设备解密流量，因此它能够读取内容（报头、txt、cookie 等），并且所有会话保持选项都能应用。 （例如:源地址、目标地址、Cookie、SSL、SIP、通用、MSRDP）

参考:

- PKI联盟
- CA/Browser Forum CA机构
- SSL Certificate Reviews CA机构评价网站
- Cipher suite - Wikipedia 密码套件
- OpenSSL Documentation
- Module ngx_http_ssl_module
- Mozilla SSL Configuration GeneratorSSL 配置推荐网站
- What is the difference between HTTP/2 and HTTP/1 (2023)
- "http2" | Can I use... Support tables for HTML5, CSS3, etc 浏览器协议支持情况
- www.baidu.com -亚数信息-SSL/TLS安全评估报告 SSL测试网站

- SSL Server Test (Powered by Qualys SSL Labs)
- 21 OpenSSL Examples to Help You in Real-World | Geekflare证书签发和格式转换
- https://my.f5.com/manage/s/article/K12783074 使用 SSL::sessionsecret iRules 命令解密 SSL 流量
- https://community.f5.com/kb/technicalarticles/tls-stateful-vs-stateless-session-resumption/281041
- https://community.f5.com/kb/technicalarticles/decrypting-tls-traffic-on-big-ip/280936 一个比较详细的流程

--道阻且长，行则将至

zy　2024-09-25