19 | TLS的各种特性：TLS握手为什么会失败？

2022-03-04 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉
时长 25:22 大小 23.23M

你好，我是胜辉。

在前面三节课里，我带你排查了 HTTP 协议相关的问题。不知你有没有注意到，这三个案例里的 HTTP 都没有做加密，这就使得我们的排查工作省去了不少的麻烦，在抓包文件里直接就可以看清楚应用层的信息了。但在现实场景下，越来越多的站点已经做了 HTTPS 加密，所以像前面的三讲那样，在 Wireshark 里直接看到应用层信息的情况，已经越来越少了。

根据 w3techs.com 的调查数据，目前 Internet 上 78% 以上的站点，都默认使用了 HTTPS。显而易见，要对 Internet 上的问题做应用层方面的分析，TLS 是一道绕不开的坎。

那你可能会问了：“我主要处理内网的问题，应该不用关心太多 HTTPS 的事了吧？”

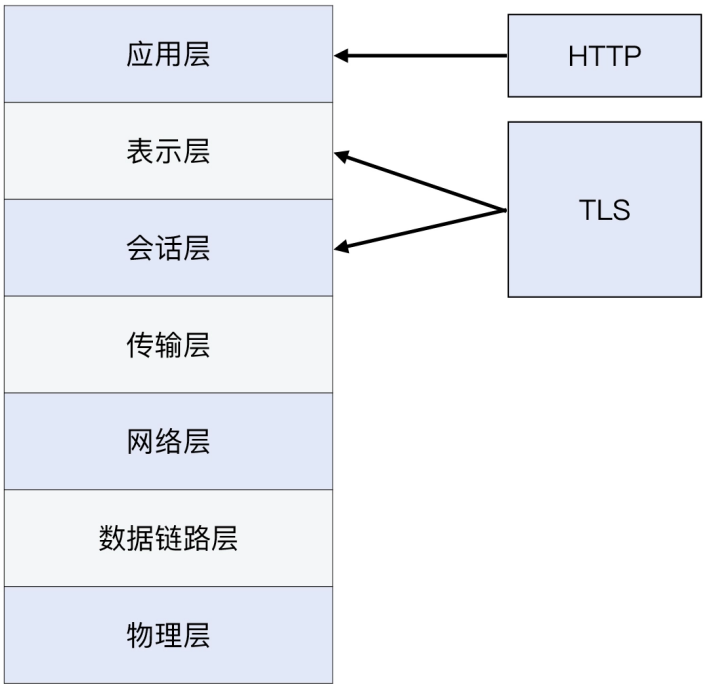
这句话也许目前还勉强算对，但是随着各大企业不断推进零信任（Zero Trust）安全策略，越来越多的内网流量也终将运行在 HTTPS 上，内网和公网将没有区别。

所以说，掌握 HTTPS/TLS 的相关知识和排查技巧，对于我们开展网络排查来说，是一项必备的技能了。

那么接下来的两节课，我们会集中到 HTTPS/TLS 这个主题上，来全面学习一下它的工作原理、常见问题和排查思路。这样以后面临 HTTPS/TLS 的问题时，你就可以运用这两讲里学到的知识和方法，展开排查工作了。

什么是 HTTPS？

首先我们要认识一下 HTTPS。它其实不是某个独立的协议，而是 HTTP over TLS，也就是把 HTTP 消息用 TLS 进行加密传输。两者相互协同又各自独立，依然遵循了网络分层模型的思想：



> 补充：这也就是我们在第 1 讲学习网络分层模型时候的图。

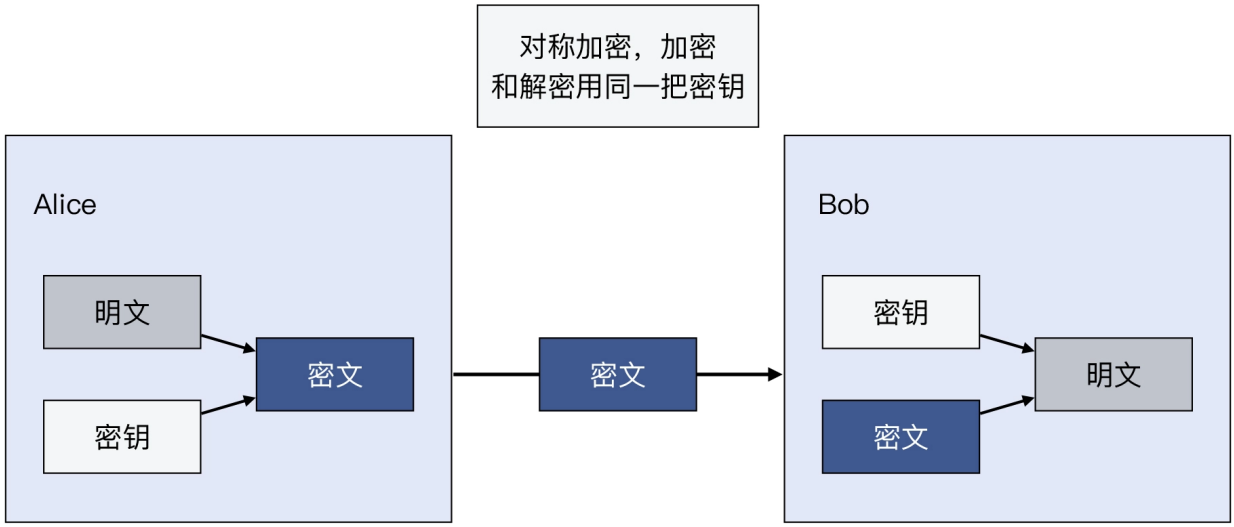
为了更好地理解 HTTPS，我们也来简单学习一下加密技术，因为它是 HTTPS 的核心。

加密技术基础

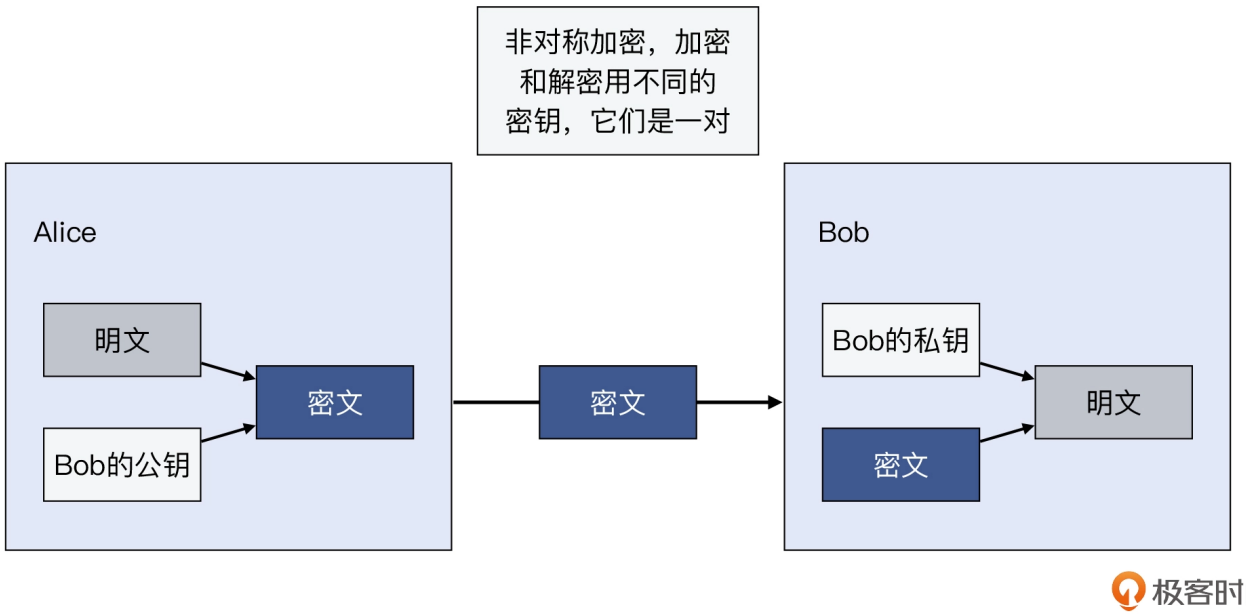
加密技术其实也是一个古老的话题。早在公元前 400 年，斯巴达人就创造了密码棒加密法。就是把纸条缠绕在一根木棒上，然后在纸上写字，这张纸条离开这根木棒后，就无法正确读取了。要“破解”它，就得找到同样粗细的木棒，然后把纸条绕上去后，才能解读。

那么在这里，纸条就相当于密文，而木棒，就相当于密钥了。而因为加密和解密用的木棒是相同的，所以它属于**对称加密算法**。

时间推进到现代，密码专家们已经开发出了众多优秀的对称加密算法，比如 AES、DES。与木棒加密法类似，Alice 和 Bob 都知道同一把密钥，Alice 用这个密钥做加密，Bob 收到密文后，也是用这把密钥做解密，就得到了明文。如下图所示：



另外一类算法就是**非对称算法**，这也是 PKI（公开密钥架构）的基础。在非对称算法中，加密和解密用了不同的密钥，这两个密钥形成了密钥对。比如 Bob 和 Alice 都各自生成了密钥对，然后互相交换了公钥。Alice 用 Bob 的公钥对明文做了加密，变成密文传给 Bob。Bob 收到后，用自己的私钥解密，就还原出了明文。如下图所示：



TLS 基础

那么 TLS 跟加密技术的关系具体是怎样的呢？实际上，**TLS 同时使用了对称算法和非对称算法**。TLS 的整个过程大致可以分为两个主要阶段：

握手阶段，完成验证，协商出密码套件，进而生成对称密钥，用于后续的加密通信。

加密通信阶段，数据由对称加密算法来加解密。

TLS 综合利用了对称算法和非对称算法的优点，因为对称算法的效率高，而非对称算法的安全性高，所以两者结合，就兼顾到了效率和安全性。不得不说，TLS 确实是一个很精妙的设计。

那么同样地，我们对 TLS 相关问题的排查，也就面临着**两类问题**：一类是 TLS 握手阶段的问题，一类是 TLS 通信过程中的问题。

在 TLS 握手阶段，真正的加密还没开始，所以依托于明文形式的握手信息，我们还有可能找到握手失败的原因。在这一阶段，我们需要掌握 TLS 握手的原理和技术细节，这样才能指导我们展开排查工作。

而在 TLS 数据交互阶段，加密已经开始，所有的数据已经是密文了。假如应用层发生了什么，而我们又看不到，那如何做排查呢？

这个时候，我们需要**把密文解密**，才能找到根因。不过你可能会问：“TLS 要是能随便解密，是不是说明这个协议还有漏洞啊？”

放心，TLS 是很安全的。我说的解密，当然是有前提条件的，跟数据安全性并不冲突。具体的细节，我到下节课会给你详细展开。

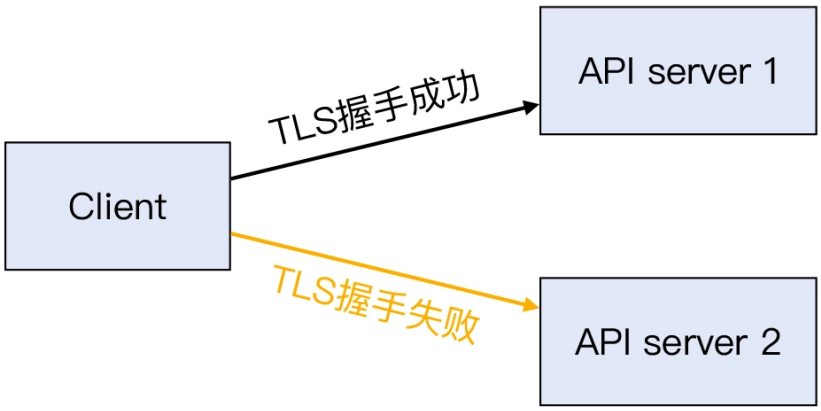
下面我们来看看案例，一起来学习下 TLS 握手失败的问题排查思路。

案例 1：TLS 握手失败

TLS 握手失败，估计你也遇到过。引起这个问题的原因还是比较多的，比如域名不匹配、证书过期等等。不过，这些问题一般都可以通过“忽略验证”这个简单的操作来跳过。比如，在浏览器的警告弹窗里点击“忽略”，就可以让整个 TLS 的过程继续下去。

而还有一些问题，就无法跳过了。

我们曾经遇到的一个例子就是这样。当时，我们有一个应用需要访问 Kubernetes 集群的 API server。因为我们有很多个集群，所以相应的 API server 也有很多个。这个问题是，从同一台客户端去访问 API server 1 是可以的，但访问 API server 2 就不行。进而发现，失败原因就是 TLS 握手失败。



在客户端的应用日志里，报告的是这段错误：

```
1 javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure
```

这段日志有没有告诉我们有价值的信息呢？好像并不多，只是告诉我们握手失败了。这也是我反复提及的，网络排查中的两大鸿沟之一的“**应用现象跟网络现象之间的鸿沟**”：你可能看得懂应用层的日志，但是不知道网络上具体发生了什么。

> 补充：我在第 4 讲里有介绍这两大鸿沟，我们要在网络排查方面取得实质性的进步，关键在于突破这两个鸿沟。

同样的，这里的日志也无法告诉我们：到底 TLS 握手哪里出了问题。所以我们需要做点别的事情。

排除服务端问题

首先，我们用另外一个趁手的小工具 curl，从这台客户端发起对 API server 2（也就是握手失败的那个）的 TLS 握手，发现其实是可以成功的。这说明，API server 2 至少在某些条件下是可以正常工作的。我们来看一下当时的输出：

```
1 curl -vk https://api.server.777.abcd.io
2 * Rebuilt URL to: https://api.server.777.abcd.io/
3 * Trying 10.100.20.200...
4 * Connected to api.server.777.abcd.io (10.100.20.200) port 443 (#0)
```

```
5  * found 153 certificates in /etc/ssl/certs/ca-certificates.crt
6  * found 617 certificates in /etc/ssl/certs
7  * ALPN, offering http/1.1
8  * SSL connection using TLS1.2 / ECDHE_RSA_AES_128_GCM_SHA256
9  * server certificate verification SKIPPED
10 * server certificate status verification SKIPPED
11 * common name: server (does not match 'api.server.777.abcd.io')
12 * server certificate expiration date OK
13 * server certificate activation date OK
14 * certificate public key: RSA
15 * certificate version: #3
16 * subject: CN=server
17 * start date: Thu, 24 Sep 2020 21:42:00 GMT
18 * expire date: Tue, 23 Sep 2025 21:42:00 GMT
19 * issuer: C=US,ST=San Francisco,L=CA,O=My Company Name,OU=Org Unit 2,CN=kubern
20 * compression: NULL
```

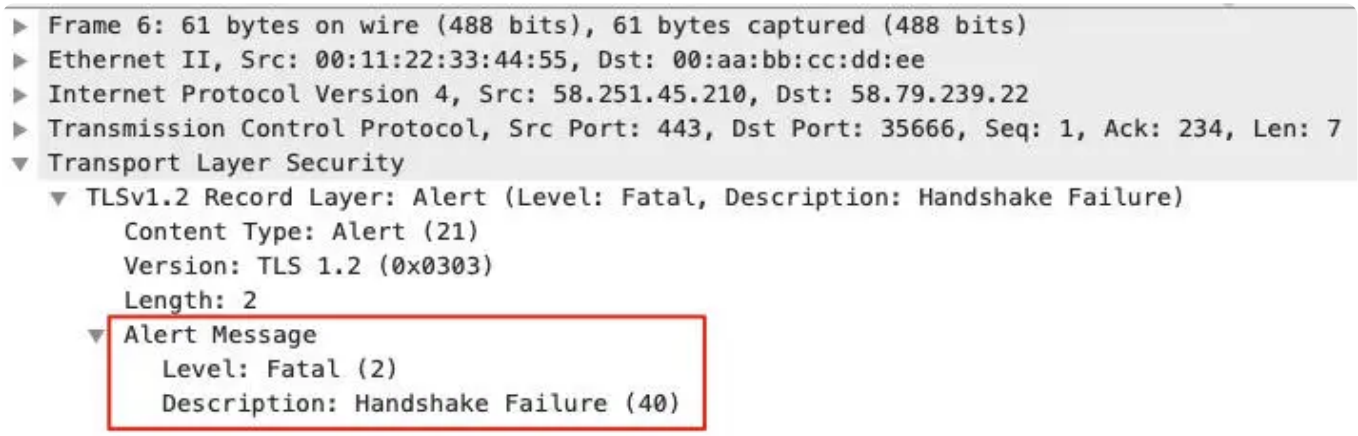
> 补充：在第 8 行可以看到协商出的密码套件 `* SSL connection using TLS1.2 / ECDHE_RSA_AES_128_GCM_SHA256`。

既然 curl 是可以 TLS 握手成功的，那是不是客户端程序本身有点问题呢？我们就进行了“问题复现”。在上节课里我们讨论了偶发性问题的“复现 + 抓包”的策略，而这里的问题是必现的，所以只要发起一次请求，同时做好抓包就可以了。

我们来看一下抓包文件：

No.	Time	Source	Destination	SrcPort	DstPort	TTL	TCP Seglen	Info
1	0.000000	58.79.239.22	58.251.45.210	35666	443	64	0	35666 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_P
2	0.000127	58.251.45.210	58.79.239.22	443	35666	247	0	443 → 35666 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0 MSS=1
3	0.000012	58.79.239.22	58.251.45.210	35666	443	64	0	35666 → 443 [ACK] Seq=1 Ack=1 Win=64256 Len=0
4	0.000757	58.79.239.22	58.251.45.210	35666	443	64	233	Client Hello
5	0.000195	58.251.45.210	58.79.239.22	443	35666	247	0	443 → 35666 [ACK] Seq=1 Ack=234 Win=16776960 Len=0
6	0.000124	58.251.45.210	58.79.239.22	443	35666	53	7	Alert (Level: Fatal, Description: Handshake Failure)
7	0.000005	58.79.239.22	58.251.45.210	35666	443	64	0	35666 → 443 [ACK] Seq=234 Ack=8 Win=64256 Len=0
8	0.000040	58.251.45.210	58.79.239.22	443	35666	53	0	443 → 35666 [FIN, ACK] Seq=8 Ack=234 Win=16776960 Len=0
9	0.000020	58.79.239.22	58.251.45.210	35666	443	64	0	35666 → 443 [FIN, ACK] Seq=234 Ack=9 Win=64256 Len=0
10	0.000115	58.251.45.210	58.79.239.22	443	35666	247	0	443 → 35666 [ACK] Seq=9 Ack=235 Win=16776960 Len=0

还真是“话不投机半句多”，客户端也就发了一个 Client Hello 报文，服务端就回复 TLS Alert 报文，结束了这次对话。那为啥聊不起来呢？我们看一下这个 Alert 报文：



这个 TLS Alert 报文显示，它的编号是 40，指代的是 Handshake Failure 这个错误类型。到这一步，我们需要去了解这个错误类型的具体定义。**正确的做法是：去 RFC 里寻找答案**，而不是随意地去网络上搜索，因为很可能你会被一些似是而非的信息误导。

因为这次握手用的是 TLS1.2 协议，我们就来看它的RFC5246。在这个 RFC 里，找到 Alert Protocol 部分，我们看看它是怎么说的：

复制代码

```
   handshake_failure
      Reception of a handshake_failure alert message indicates that the
      sender was unable to negotiate an acceptable set of security
      parameters given the options available.  This is a fatal error.
```

结合这里的实际场景，这段话的意思就是：“基于已经收到的 Client Hello 报文中的选项，TLS 服务端无法协商出一个可以接受的安全参数集”。而这个所谓的安全参数集，在这里具体指的就是加密算法套件 Cipher Suite。我们来认识一下它。

> 补充：这里的 suite 读音是 sweet 而不是 suit，我也错读过很多年。另外，suite 还有旅馆套房的意思。

Cipher Suite

前面提到过，在 TLS 中，真正的数据传输用的加密方式是**对称加密**；而对称密钥的交换，才是使用了**非对称加密**。实际上，TLS 的握手阶段需要在下面四个环节里实现不同类型的安全性，它们可以说是 TLS 的“四大护法”。

密钥交换算法：保证对称密钥的交换是安全的，典型算法包括 DHE、ECDHE。

身份验证和签名算法：确认服务端的身份，其实就是对证书的验证，非对称算法就用在这里。典型算法包括 RSA、ECDSA。

> 补充：如果是双向验证（mTLS），服务端会验证客户端的证书。

对称加密算法：对应用层数据进行加密，典型算法包括 AES、DES。

消息完整性校验算法：确保消息不被篡改，典型算法包括 SHA1、SHA256。

每一个类型都有很多不同的具体算法实现，它们的组合，就是密码套件 Cipher Suite。你可能以前也见过它，这次咱们来拆解认识一下它的组成结构。

先看一个典型的密码套件：

TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA（0xc013）

TLS 不用多说，代表了 TLS 协议。

ECDHE 是密钥交换算法，双方通过它就不用直接传输对称密钥，而只需通过交换双方生成的随机数等信息，就可以各自计算出对称密钥。

RSA 是身份验证和签名算法，主要是客户端来验证服务端证书的有效性，确保服务端是本尊，非假冒。

AES128_CBC 是对称加密算法，应用层的数据就是用这个算法来加解密的。这里的 CBC 属于块式加密模式，另外一类模式是流式加密。

SHA 就是最后的完整性校验算法（哈希算法）了，它用来保证密文不被篡改。

0xc013 呢，是这个密码套件的编号，每种密码套件都有独立的编号。完整的编号列表在 IANA 的网站上可以找到。

另外，在不同的客户端和服务端软件上，这些密码套件也各不相同。所以，TLS 握手的重要任务之一，就是**找到双方共同支持的那个密码套件**，也就是找到大家的“共同语言”，否则握手就必定会失败。

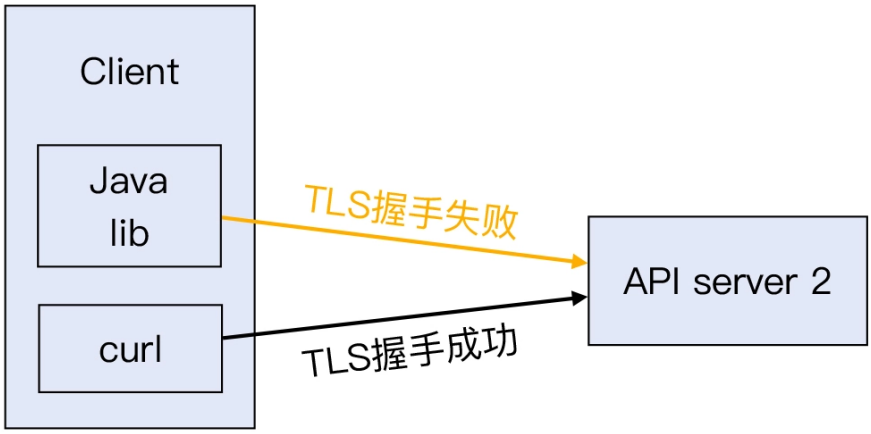
所以这个案例排查的下一步，就是要搞清楚，客户端和服务端到底都支持了哪些 Cipher Suite。

那么客户端的密码套件有哪些呢？你可能很快想到了前面 curl 命令里的输出。确实，那里就明确显示，双方协商出来的是 **ECDHE_RSA_AES_128_GCM_SHA256**。但是，这里有两个问题：

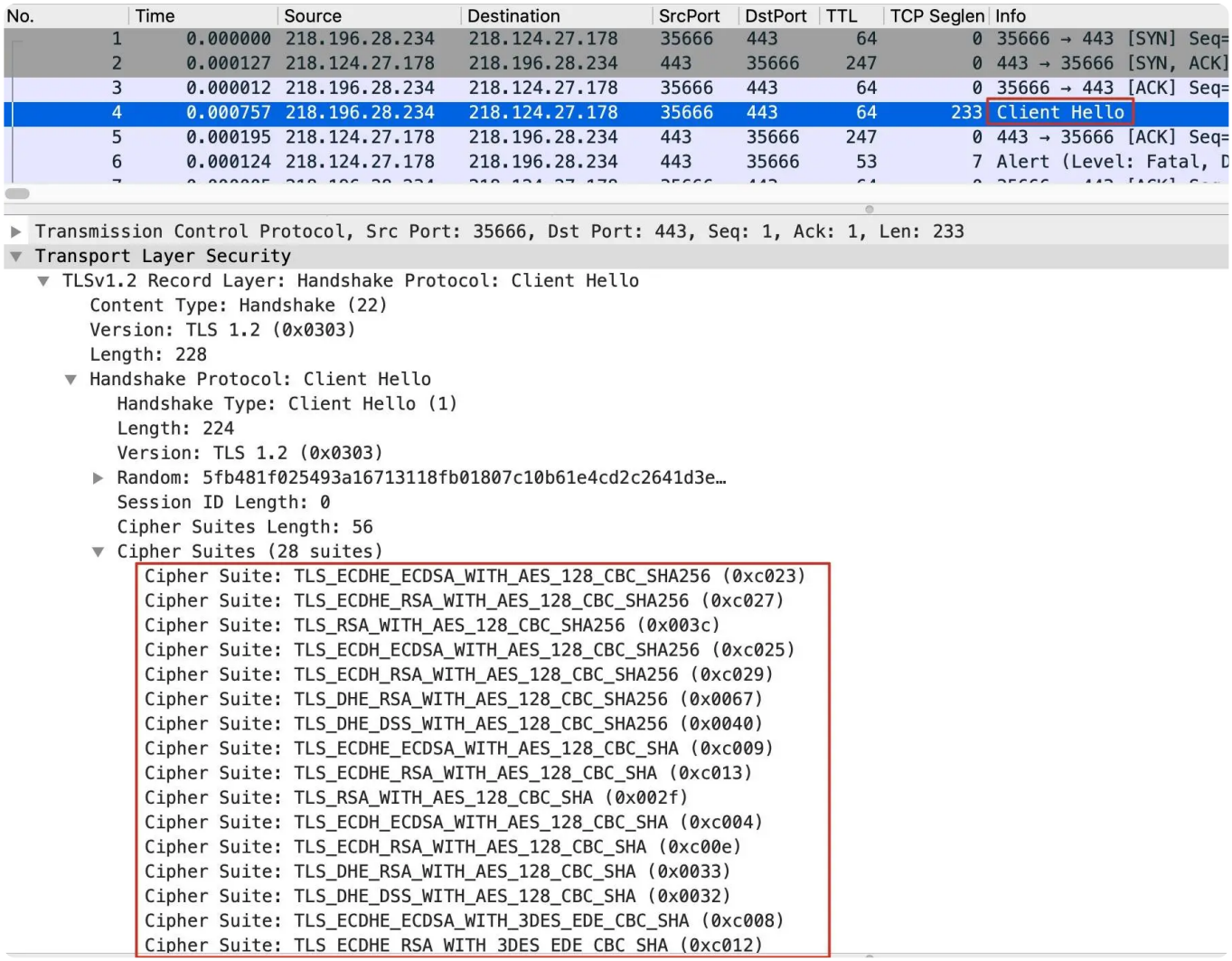
这个是协商后达成的结果，只是一个套件，而不是套件列表。

更加关键的是，这个密码套件是 curl 这个客户端的，而不是出问题的客户端。

所谓出问题的客户端，就是实际的业务代码去连接 API server 时候用的客户端，它是一个 Java 库，而不是 curl，这一点一定要分清。



那么，我们怎么获得这个 Java 库能支持的密码套件列表呢？其实最直接的办法，还是用**抓包分析**。我们回到前面那个抓包文件，检查一下 Client Hello 报文。在那里，就有 Java 库支持的密码套件列表。

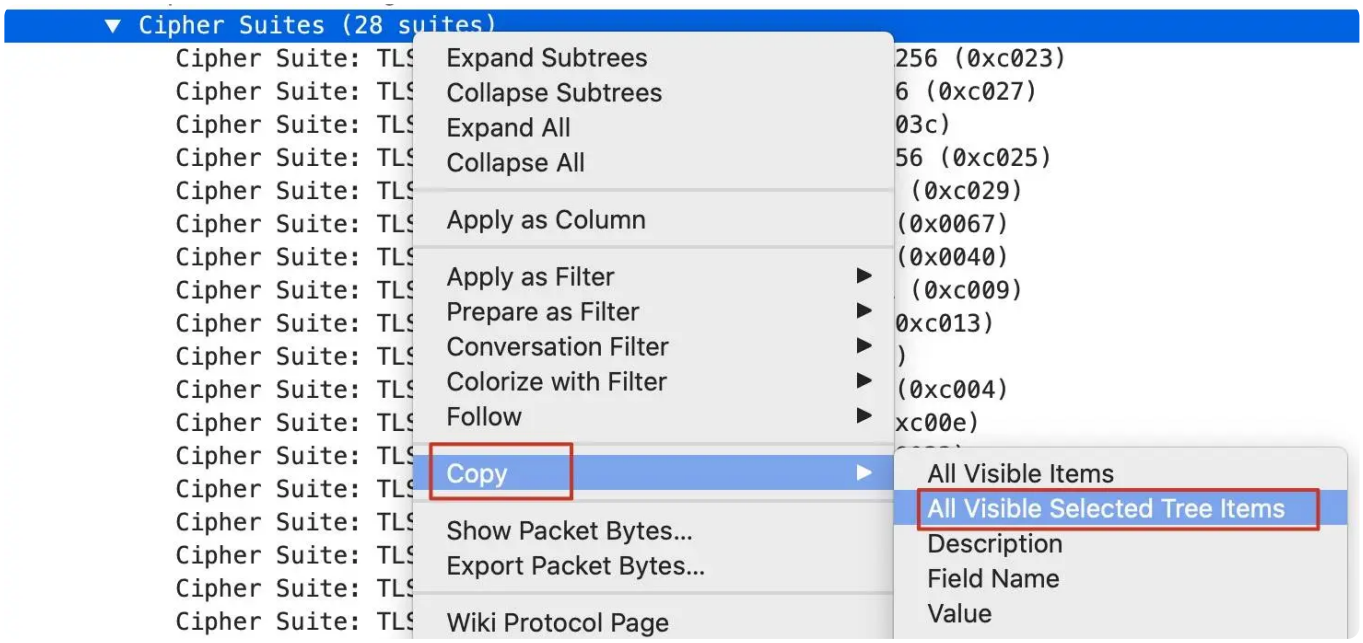


> 补充：这个列表往下还有。因为屏幕小，我没有全部展示。

找到了客户端的密码套件列表，接下来是不是就去找服务端的密码套件的列表呢？不过，这个抓包里，服务端直接回复了 Alert 消息，并没有提供它支持的密码套件列表。那我们的排查如何继续推进呢？

其实，可以换个思路：看看服务端在 TLS 握手成功后用了哪个密码套件，而不是去拿到它的全部列表。前面 curl 已经成功了，**我们来看下 curl 那次协商出来的套件是哪个，看它是否被 Java 库支持，就可以判定了。**

我们要导出这次 Client Hello 里面的密码套件列表，可以这样做：选中 Cipher Suite，右单击，选中 Copy，在次级菜单中选中 All Visible Selected Tree Items：



这样，我们就得到了下面这个列表：

```
Cipher Suites (28 suites)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256 (0xc025)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256 (0xc029)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0x0067)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA256 (0x0040)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc004)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc00e)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc008)
    Cipher Suite: TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012)
    Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc003)
    Cipher Suite: TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA (0xc00d)
    Cipher Suite: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x0016)
    Cipher Suite: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x0013)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
    Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
    Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_RC4_128_SHA (0xc002)
    Cipher Suite: TLS_ECDH_RSA_WITH_RC4_128_SHA (0xc00c)
    Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
    Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)
```

可见，里面确实没有 **ECDHE_RSA_AES_128_GCM_SHA256** 这个套件。所以到这里，我们可以确认问题根因了：因为这个 Java 库和 API server 2 之间，没有找到共同的密码套件，所以 TLS 握手失败。

根因找到了，下一步就是升级 Java 库，让双方能够协商成功。

> 补充：API server 1 能兼容这个相对旧的 Java 库，所以没有问题。

你觉得这个问题难吗？其实还好，对吧？这是因为我们一旦对协议本身有准确的理解，那么很多问题就容易被“看穿”。这也说明了理论知识的重要性。

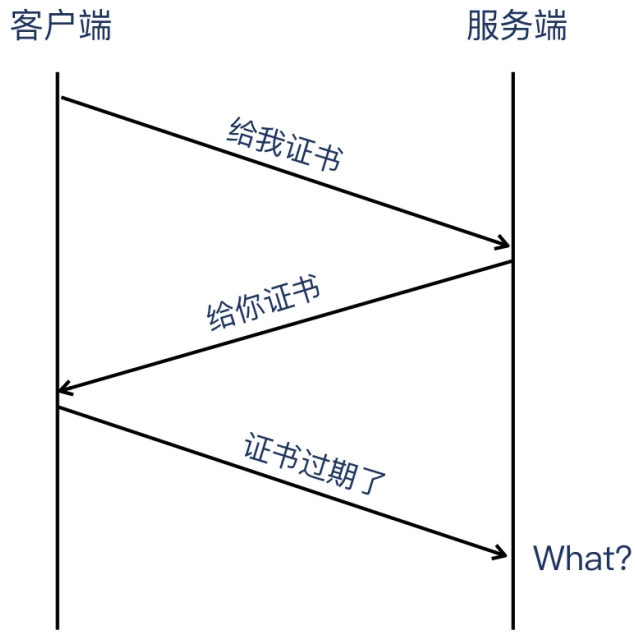
好，我们再来看一个复杂一点的案例。

案例 2：有效期内的证书为什么报告无效？

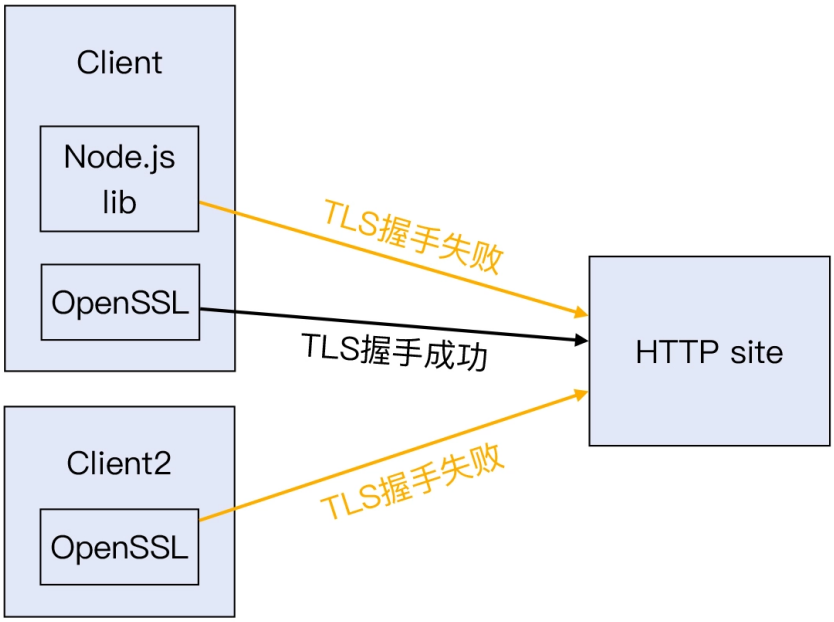
有一次，一个产品开发团队向我们运维团队报告了一个问题：他们的应用在做了代码发布后，就无法正常访问一个内部的 HTTPS 站点了，报错信息是：certificate has expired。

这就很奇怪了，我们日常对证书都做了自动更新处理，不会有“漏网之鱼”。然后我们也手工检查了这个 HTTPS 站点的证书，确定是在有效期内的，这就使得这个报错显得尤其古怪。

既然是代码发布后新出现的问题，那我们自然认为问题是跟发布有关。我们了解到：这次确实有个变更，会在客户端打开服务端证书校验的特性，而这个特性在以前是不打开的。但这还是无法解释，为什么客户端居然会认为，一个明明在有效期内的证书是过期的。



真是“秀才遇到兵”，感觉“讲理”是行不通了，于是我们换了个思路，不纠结在有效期的问题上。跟前一个案例类似，我们用交叉验证的方式来推进排查。具体做法是：在这台客户端和另一台客户端上，用 OpenSSL 向这个 HTTPS 站点发起 TLS 握手。



结果我们发现了更有意思的情况：从另外一台客户端的 OpenSSL 去连接这个 HTTPS 站点，也报告 certificate has expired。

这给了我们很大的信心：既然 OpenSSL 可以复现这个问题，那我们就可以做进一步的检查了！因为 OpenSSL 属于 OS 上的命令，虽然我们不了解如何在 Node.js 上做 debug，

但是我们对如何在 OS 上做排查是很有经验的。

于是，我们在 OpenSSL 命令前面加上 **strace**，以便于追踪 OpenSSL 在执行过程中，特别是在报告 certificate has expired 之前，具体发生了什么。执行这个命令：

```
1 strace openssl s_client -tlsextdebug -showcerts -connect abc.ebay.com:443
```

输出的关键部分如下：

```
1  stat("/usr/lib/ssl/certs/a1b2c3d4.1", {st_mode=S_IFREG|0644, st_size=2816, ...
2  openat(AT_FDCWD, "/usr/lib/ssl/certs/a1b2c3d4.1", O_RDONLY) = 6
3  ......
4  write(2, "verify return:1\n", 16verify return:1
5  )       = 16
6  .......
7  write(2, "verify error:num=10:certificate "..., 44verify error:num=10:certific
8  ) = 44
9  write(2, "notAfter=", 9notAfter=)                 = 9
10 write(2, "Oct 14 18:45:33 2020 GMT", 24Oct 14 18:45:33 2020 GMT) = 24
```

这里的关键信息是：

> OpenSSL 读取了`/usr/lib/ssl/certs`目录下的文件 `a1b2c3d4.1`。
>
> 接着，OpenSSL 就报告了 certificate has expired 的错误，expire 的日期是 2020 年 10 月 24 日（输出中的“24Oct 14”）。

这又是一个明显的进展：很可能就是这个文件导致了错误。这是个什么文件，为什么会导致错误呢？

其实，它就是 TLS 客户端本地的 Trust store 里，存放的中间证书文件。Trust store 一般用来存放根证书和中间证书文件，你可能对这几个名词还不太熟悉，我给你介绍一下 TLS 证书校验的原理。

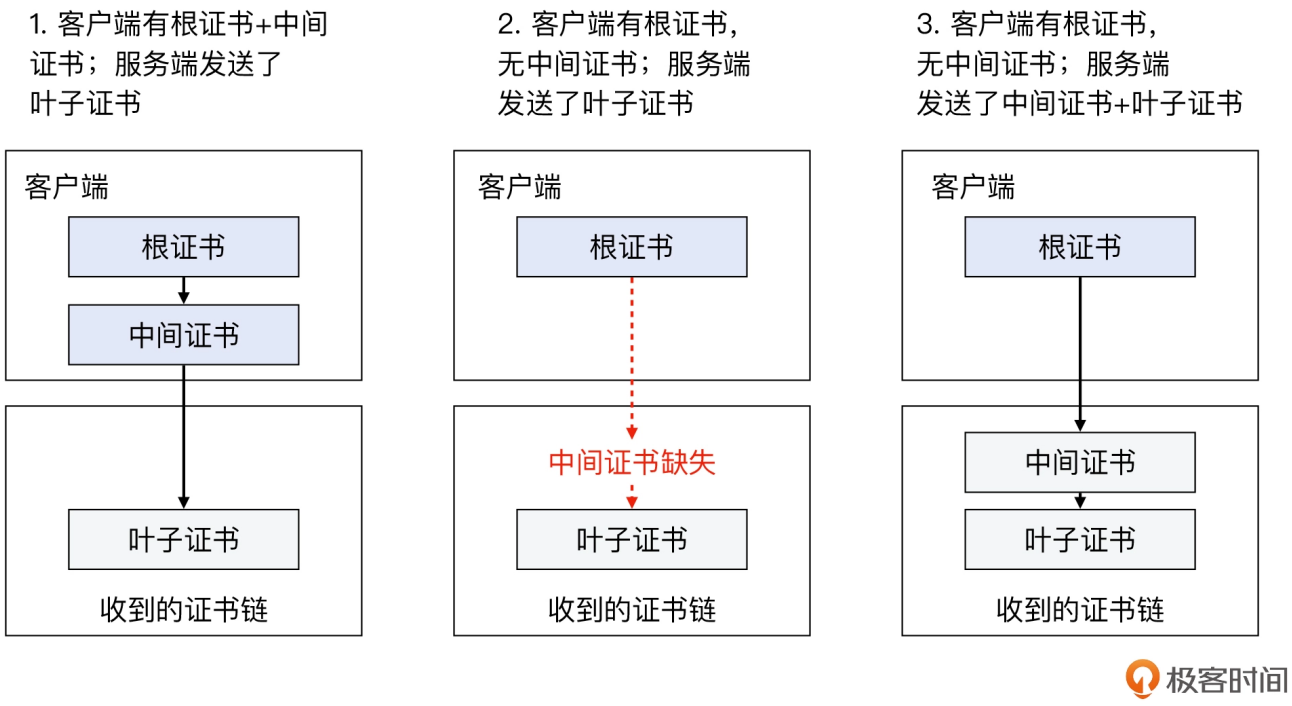
> 补充：一般来说，证书先存入文件系统，然后通过命令或者代码，导入到应用的 Trust store。

TLS 证书链

TLS 证书验证是“链式”的机制。比如，客户端存有根证书和它签发的中间证书，那么由中间证书签发的叶子证书，就可以被客户端信任了，也就是这样一条信任链：

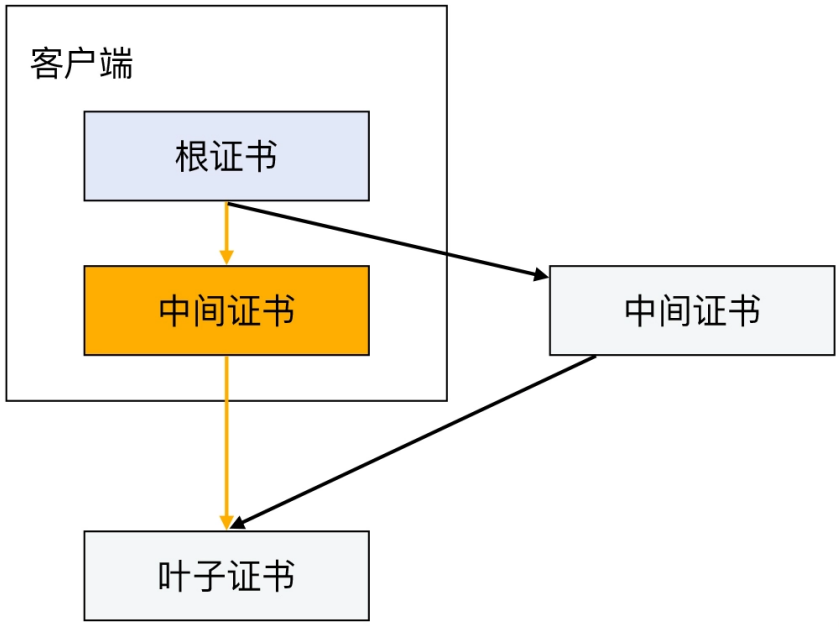
信任根证书 -> 信任中间证书 -> 信任叶子证书

我画了三种不同情况下的信任链的示意图，供你参考：



场景 1 和 3 中，信任链是完整的，证书验证就可以通过。场景 2 中，由于中间证书既不在客户端的 Trust store 里，也不在服务端回复的证书链中，这就导致信任链断裂，验证就会失败。

而我们发现在这个案例里，服务端发送的证书链中包含了正确的中间证书，那为什么还会失败呢？其实这是因为，从前面 strace openssl 的输出里已经发现，客户端本地也有一张中间证书，而且是**过期的**，示意图如下：



这两张中间证书，签发机构是同一个 CA，证书名称也相同，这就导致了 OpenSSL 在做信任链校验时，优先用了本地的中间证书，进而因为这张本地的中间证书确实已经过期，导致 OpenSSL 抛出了 certificate has expired 的错误！

这个结论你看明白了吗？你也许觉得还是有哪里不对，比如你可能会问：“照理说叶子证书是新的中间证书签发的，那用老的中间证书去验证叶子证书的签名的时候，应该会失败啊？”

你说得没错。这里最烧脑的地方在于：这两张中间证书，不仅签发机构一样，名称一样，而且**私钥也一样**！

如果你对 TLS 不熟悉，学到这里可能已经觉得有点“爆炸”了。先别急，下面有详细的解释。

这里的核心秘密在于：每次证书在更新的时候，**它对应的私钥不是必须要更新的，而是可以保持不变的。**

我们把本地的已经过期的中间证书，称为 old_cert，新的中间证书称为 new_cert。整个故事就是这样：

> 几年前，old_cert 被根证书签发了出来，名称为 inter-CA，并被保存在这台客户端的 Trust store 里。

在 2020 年，old_cert 到期，根证书机构重新签发了一张新的中间证书 new_cert，它用了新的有效期，而**证书名称 inter-CA** 和对应的**私钥**都保持不变。

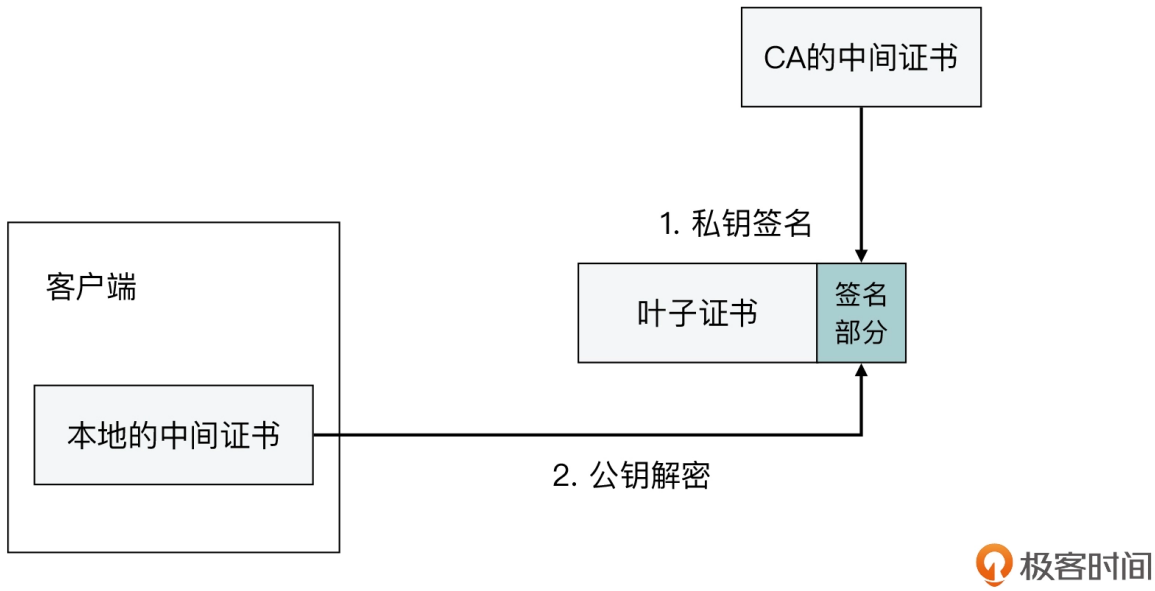
CA 用这张 new_cert 签发了这次的叶子证书。因为客户端程序没有打开证书校验机制，所以没有报错。

这一天，新的代码发布上去，证书校验机制被打开了，于是客户端开始做校验。它发现这张叶子证书的签发者名称是 inter-CA，而自己本地就有一张也叫 inter-CA 的证书，于是尝试用这张证书的公钥去解开叶子证书的签名部分，可以成功解开，于是确认 old_cert 就是对应的中间证书（而没有用收到的 new_cert，这很关键）。但是由于 old_cert 已经过期了，结果客户端抛出 certificate has expired 的错误！

如果你还没有完全看明白，说明你真的在思考了，因为我确实还没有讲完。接下来介绍核心知识点：TLS 证书签名。

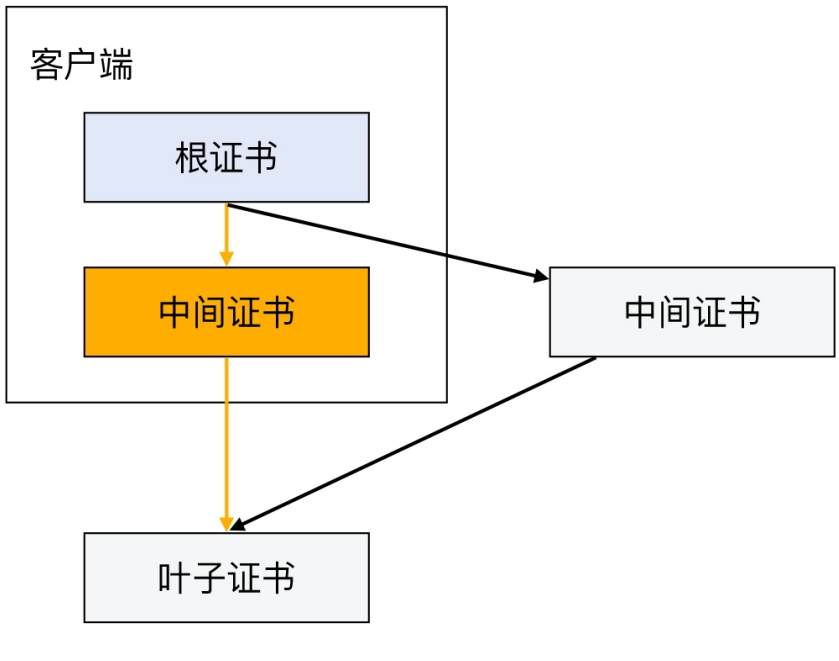
TLS 证书签名

你应该也知道，TLS 证书都有签名部分，这个签名就是用签发者的私钥加密的。客户端为什么会相信叶子证书真的是这个 CA 签发的呢？就是因为，客户端的 Trust store 里就有这个 CA 的公钥（在 CA 证书里），它用这个公钥去尝试解开签名，能成功的话，就说明这张叶子证书确实是这个 CA 签发的。



这里最关键的部分在于，新老中间证书用的私钥是同一把，所以这张叶子证书的签名部分，**用老的中间证书的公钥也能解开**，这就使得下图中的橙色的验证链条得以“打通”，不过，谁也没料到打通的是一条“死胡同”。

> 补充：PKI 里有交叉签名的技术，就是新老根证书对同一个新的中间证书进行签名，但并不适用于这个案例。



OpenSSL 报错的原因找到了，根据这个发现，我们也确认了 Node.js 的 Trust store 也存在同样的问题。我们把它的 Trust store 里的过期证书全部删除后，问题就被解决了。

另外在排查过程中，我们偶然发现 Stack Overflow 上也有人报告了类似的问题。于是我在 Stack Overflow 上也做了回复，期望可以对遇到类似问题的人提供帮助。

> 补充：我的留言在三楼，署名 VictorYang。

小结

这节课，我们通过两个典型案例，学习了 TLS 相关的知识，你可以重点关注和掌握以下知识点。

加密算法的类型

对称加密算法：加密和解密用同一个密钥，典型算法有 AES、DES。

非对称加密算法：加密和解密用不同的密钥，典型的非对称加密算法有 RSA、ECDSA。

TLS 基础

TLS 是先完成握手，然后进行加密通信。非对称算法用于交换随机数等信息，以便生成对称密钥；对称算法用于信息的加解密。

Cipher Suite

在握手阶段，TLS 需要四类算法的参与，分别是：密钥交换算法、身份验证和签名算法、对称加密算法、消息完整性校验算法。这四类算法的组合，就形成了密码套件，英文叫 Cipher Suite。这是 TLS 握手中的重要内容，我们的案例 1 就是因为无法协商出公用的密码套件，所以 TLS 握手失败了。

TLS 证书链

TLS 的信任是通过对证书链的验证：

信任根证书 -> 信任中间证书 -> 信任叶子证书

本地证书加上收到的证书，就形成了证书链，如果其中有问题，那么证书校验将会失败。我们的案例 2，就是因为一些极端情况交织在一起，造成了信任链过期的问题，导致证书验证失败了。

Trust store

它是客户端使用的本地 CA 证书存储，其中的文件过期的话可能导致一些问题，在排查时可以重点关注。

排查技巧

在排查技巧方面，你要知道使用 **curl 命令**，检查 HTTPS 交互过程的方法：

复制代码

```
1 curl -vk https://站点名
```

以及使用 **OpenSSL 命令**来检查证书的方法，也就是：

```
1 openssl s_client -tlsextdebug -showcerts -connect 站点名:443
```

复制代码

另外在需要分析 OpenSSL 为什么报错的时候，你可以在前面加上 **strace**，这对于排查根因有不少的帮助。

然后，我也带你学习了**如何在 Wireshark 里导出 Cipher Suite 的方法**，就是在 TLS 详情中选中 Cipher Suite，右单击，选中 Copy，在次级菜单中选中 All Visible Selected Tree Items。这时，列表就被复制出来了。

除此之外，我们还在排查 TLS Alert 40 这个信息时，通过查阅RFC5246得到了答案。所以，在遇到一些协议类型、定义相关的问题时，**最好查阅权威的 RFC 文档，这样可以获得最准确的信息。**

思考题

最后还是给你留两道思考题：

> 我们知道 TCP 是三次握手，那么 TLS 握手是几次呢？
>
> 假设服务端返回的证书链是根证书 + 中间证书 + 叶子证书，客户端没有这个根证书，但是有这个中间证书。你认为客户端会信任这个证书链吗？

欢迎在留言区分享你的答案，也欢迎你把今天的内容分享给更多的朋友。

上一篇 18 | 偶发性问题如何排查？

精选留言 (6)

写留言

Realm
2022-03-04

问题1 ：
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):...

展开

作者回复: 问题1：很详细，依次是OUT, IN, OUT, IN，是四次握手： ）
问题2：对，这个会验证失败。

taochao_zs
2022-03-04

杨老师有微信群吗，后面还想多和杨老师多多请教。

作者回复: 嗯我让助教拉一下~

共 2 条评论 >

那时刻
2022-03-04

我们之前也遇到过tls偶尔握手失败的案例，抓包来看是，客户端发出client hello之后，收到服务器的rst消息，且这个rst消息的ttl的值是64，我们猜想是被防火墙阻止了。不知老师是否遇到这样的case？以及分析原因呢？

作者回复: 多半是防火墙发出了RST，而且因为这个RST报文的TTL是64，就是跟你的网关了，因为这是TTL原始值的一种（其他常见的是128和255）。


那时刻

2022-03-04

请问老师，PKI 里有交叉签名的技术，就是新老根证书对同一个新的中间证书进行签名，但并不适用于这个案例。 此时，对于这个中间证书签名的叶子证书，验证流程是如何呢？新老根证书都会验证么？

作者回复: 嗯 我想表达的意思是，叶子证书不会是新老两个中间证书做交叉签名的。之所以能用老的中间证书的公钥也能解开，只是因为新的中间证书用的私钥还是老的那个。
客户端拼接处的证书链是"根证书（未过期）->中间证书（过期）->叶子证书（未过期）"，因为其中的中间证书过期了，所以报告了certificate has expired。也就是最让人困惑的地方其实是：我们以为这里说的certificate是叶子证书，实际上是中间证书~
我这样说是否更清晰一些？：）


Geek6561

2022-03-04

问题1比较肤浅的理解：TLS应该是4次握手（client hello， serverHello、cert share，finish），TLS RSA 和TLS1.2 的ECDHE都是这种，如果是TLS 1.3的 Diffie-Hellman，4次还是5次不太确定了。TLS太复杂了，这一块的东西，老师有分享的打算吗？

问题1：如果客户端没有这个根证书，是不会信任服务端返回的证书链的。因为PKI的基...
展开

作者回复: 问题1你的答案是对的。TLS一般来说是4次握手，如果没有session reuse等情况的话。当然这也不是必然的，比如TLS1.2也可以做到1RTT，甚至TLS1.3可以是0RTT，这些都是协议做的优化，为了节省在RTT上花费的时间。在下一讲里会有更多的TLS的内容，你可以期待一下。
问题2的答案也正确。因为客户端没有这个根证书，那么这个证书链的源头无法验证通过，就失去了信任链的根基。
common name: server (does not match 'api.server.777.abcd.io')这个在正常验证中确实会报错的，但是关闭了证书校验就不会了。这次握手失败，原因不是名称不符，是无法找到共用的密码套件，所以无法开展密钥协商。
能看出来你都认真思考了，很棒：）


taochao_zs

2022-03-04

1 TLS是四次握手，前面2次是hello握手，后面2次是交换密钥握手
2 会信任，只要中间证书是根证书签发出来的，可以验证证书的关系链是否一致（证书包含相同密钥和签名算法，完整性算法这些）。

作者回复: 1. 是正确的
2. 题目里提到客户端并没有服务端返回的根证书，那么会验证失败，因为信任的起点是根证书~