20 | TLS加解密：如何解密HTTPS流量？

2022-03-07 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉

时长 24:47 大小 22.71M

你好，我是胜辉。

在上节课里，我们对 TLS 的整体的知识体系做了总览性的介绍，然后回顾了两个实际的案例，从中领略了 TLS 握手的奥妙。我们也知道了，TLS 握手的信息量还是很大的，稍有差池就可能引发问题。我们只有对这些知识有深刻的理解，才能更准确地展开排查。

不过，也正因为这种种严苛的条件，TLS 才足够安全，因为满足了这些前提条件后，真正的数据传送就令人十分放心了。除非你能调动超级计算机或者拥有三体人的智慧，要不然一个 TLS 连接里面的加密数据，你是真的没有办法破解的。

可话说回来，**如果排查工作确实需要我们解开密文，查看应用层信息，那我们又该如何做到呢？**

所以在这节课里，我会带你学习 TLS 解密的技术要点，以及背后的技术原理，最后进行实战演练，让加密不再神秘。好了，让我们开始吧。

TLS 加密原理

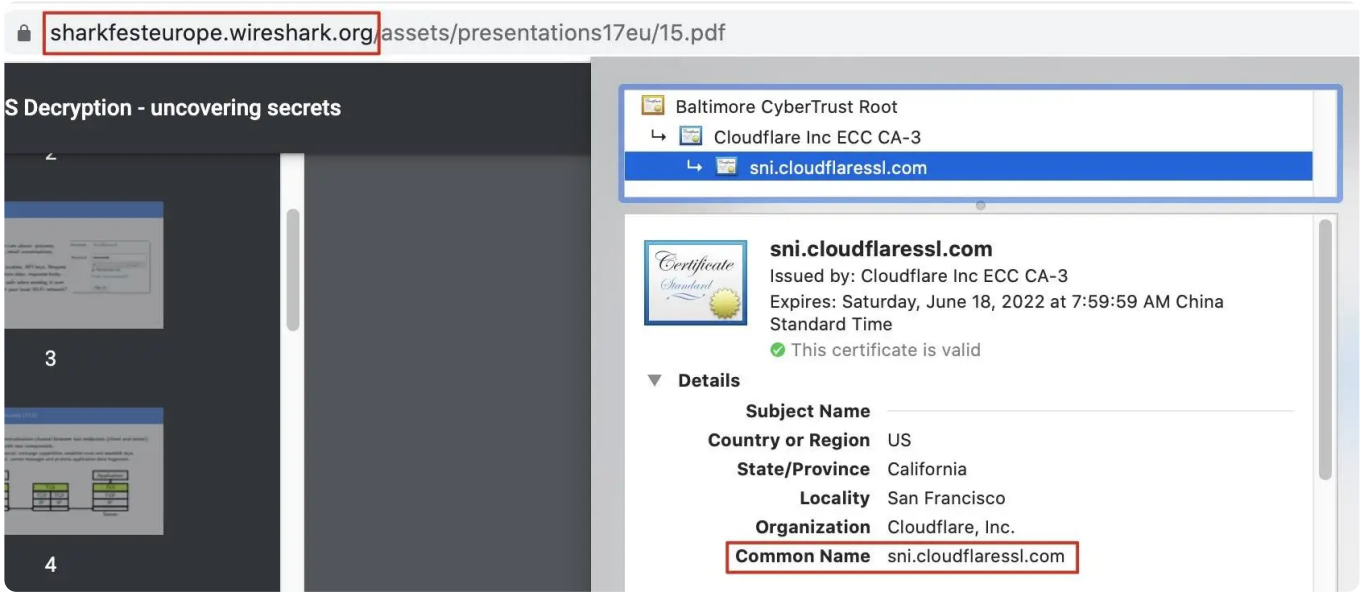
在上节课里我们已经了解到，TLS 是结合了对称加密和非对称加密这两大类算法的优点，而密码套件是四种主要加密算法的组合。那么这些概念，跟我们的日常工作又有着什么样的交集呢？

解读 TLS 证书

下面这个证书，是我在访问站点 https://sharkfesteurope.wireshark.org 的时候获取到的，我们来仔细读一下这里面的内容，看看哪些是跟我们学过的 TLS 知识相关的。

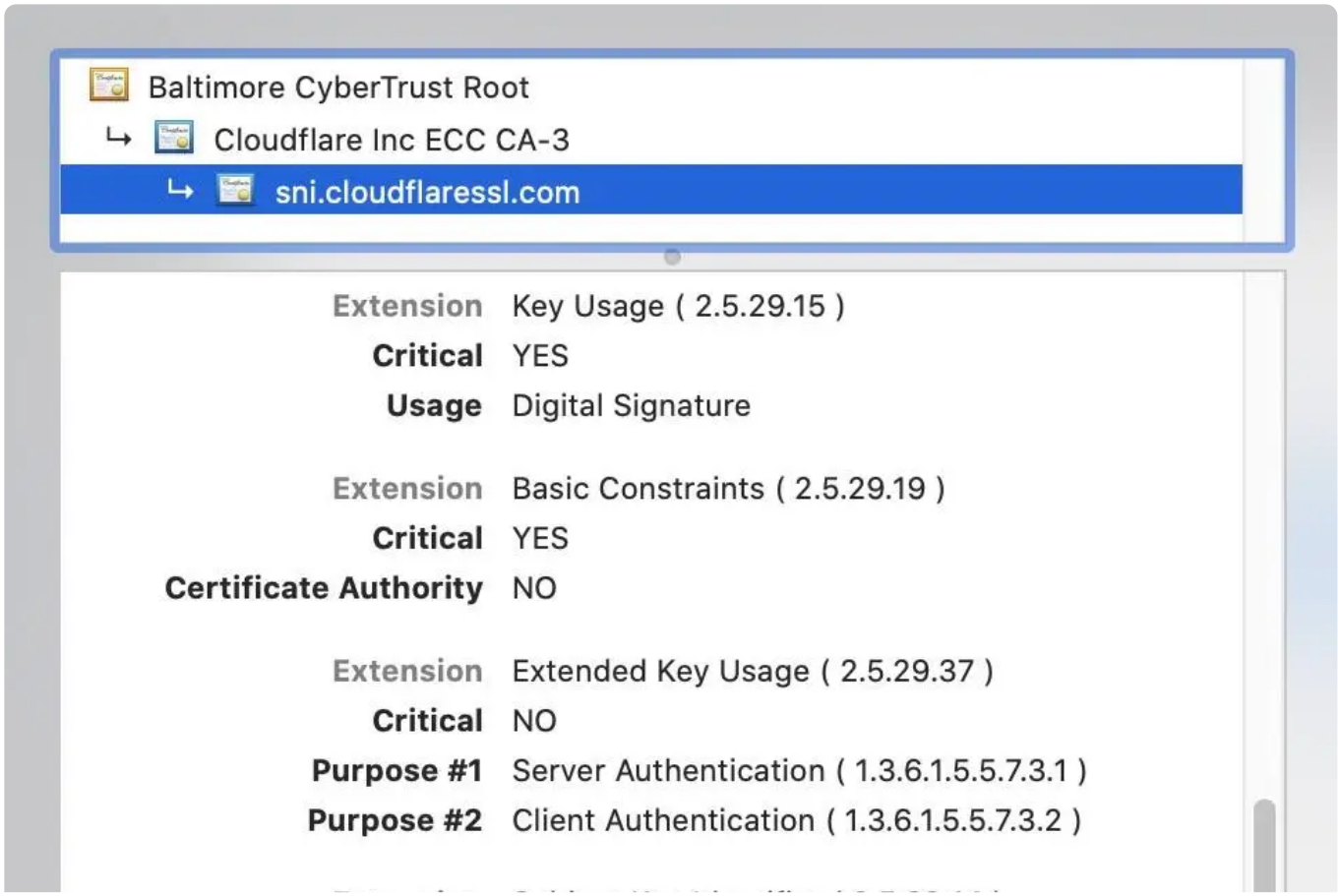
我把图中的很多关键信息做了标记，希望可以帮助你更好地理解。

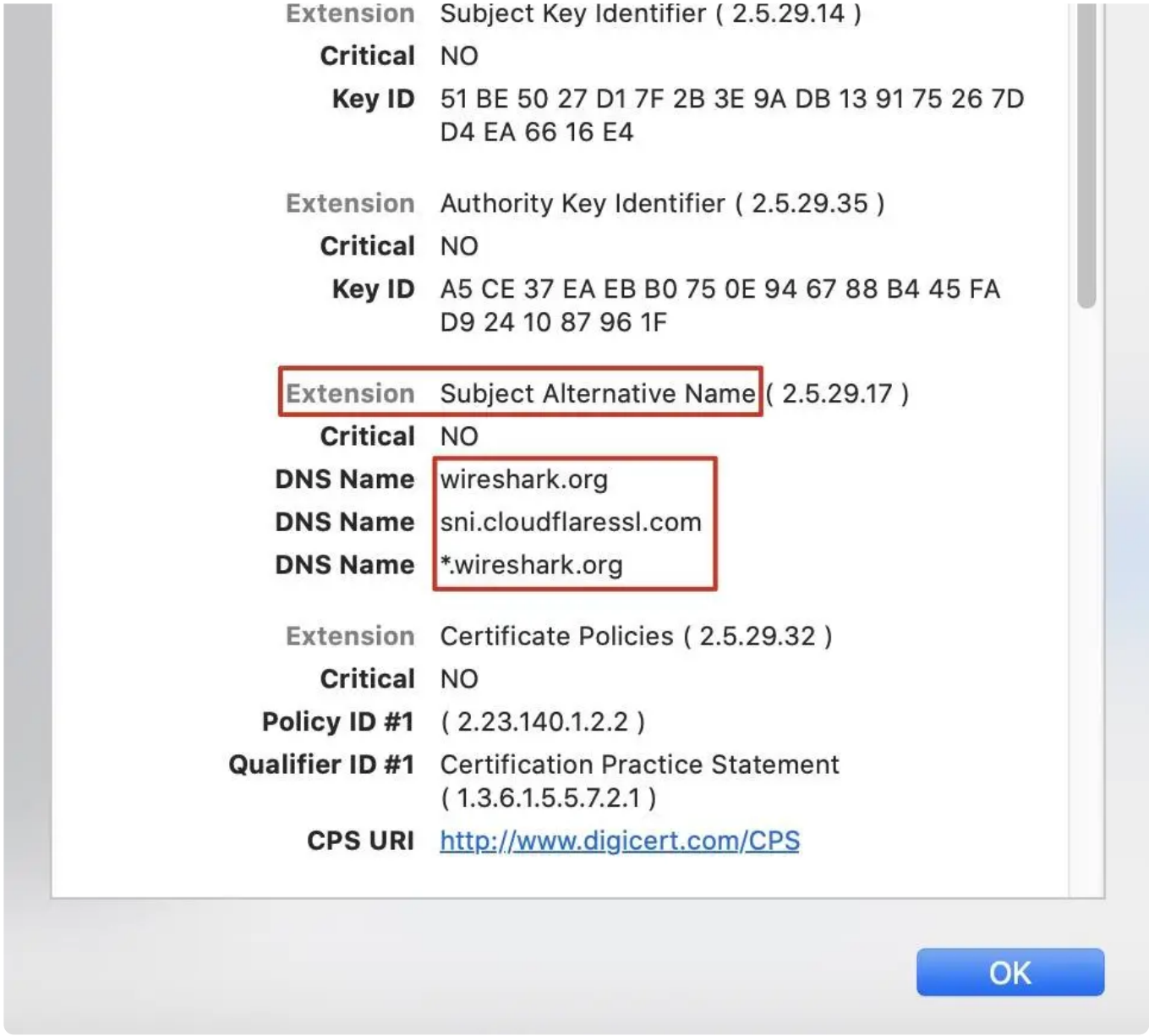
从上到下，我们了解了这张证书所在的证书链，然后是证书名称、身份验证和签名算法、有效期。不过，看完这个证书，你可能也发现了一个小问题：站点名称跟证书名称不一致？这两个不匹配，浏览器为啥不报错呢？



其实，这里的站点名称跟证书实际上是匹配的，但它匹配的不是 Common Name，而是另外一个概念：SAN。

TLS 证书为了支持更多的域名，设计了一个扩展选项 Subject Alternative Name，简称 **SAN**，它就包含有多个域名。比如还是这张证书，它的 SAN 里的域名里就有 wireshark.org、sni.cloudflaressl.com，还有跟这次访问的站点名直接相关的 *.wireshark.org。这个是通配符域名，就意味着 sharkfesteurope.wireshark.org 也被支持了。SAN 列表如下：





这里也有一个小的注意点：通配符证书只能支持一级域名，比如 *.wireshark.org 证书可以支持以下域名：

a.wireshark.org

b.wireshark.org

但不支持这样的域名：

a.b.wireshark.org

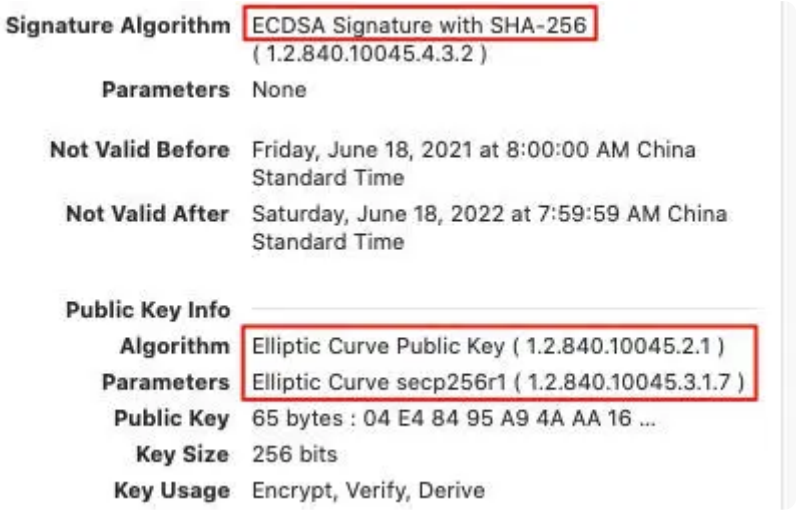
a.b.c.wireshark.org

然后我们再来温习一下**密码套件**。在这张证书里，我们能看出它用到的密码套件是什么了吗？下面我们来解读一下。

密钥交换算法是什么呢？这在证书里看不出来，需要根据握手协商的结果来判定。不过，我们也可以有个初步的判断。如果这次通信用的 TLS 版本是 1.3，那么就是 DHE 或者 ECDHE 这样的“前向加密”的密钥交换算法了。结尾的 E 是 Ephemeral，意思是“短时间的”，也就是密钥是每次会话临时生成的。

> 补充：稍后我会介绍什么是前向加密。

身份验证和签名算法呢？就是证书里明确写着的 ECDSA，其中 EC 就是 Elliptic Curve 的缩写，也就是椭圆曲线算法，它可以用更短的密钥达到跟 RSA 同样的密码强度。后面跟着的 SHA-256 是哈希摘要算法，证书内容用这个 SHA-256 算法做了哈希摘要，然后用 ECDSA 算法对摘要值做了签名，这样的话，客户端就可以验证这张证书的内容有没有被篡改了。



对称加密算法又是什么呢？在证书这里看不出来，因为它也是通过握手协商出来的。当然，用 OpenSSL 或者 curl 命令就可以观察到，我们稍后演示。

最后是完整性校验算法了。其实在 2 里面已经提过了，是 SHA-256。

我们用 OpenSSL 命令，可以直接观测到这次 TLS 里协商出来的密码套件：

复制代码

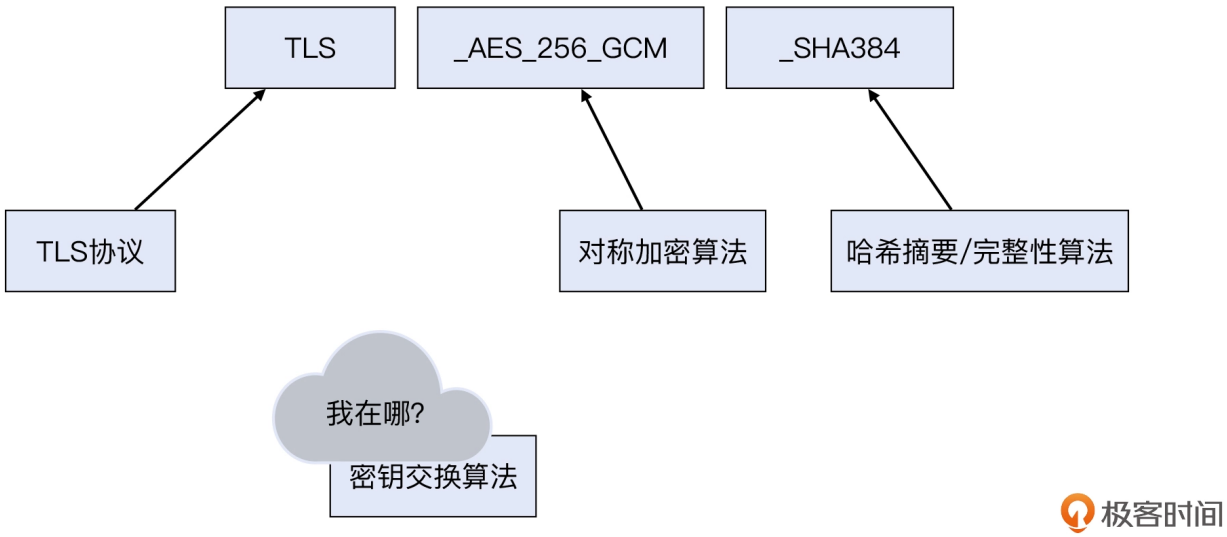
```
1 $ openssl s_client -connect sharkfesteurope.wireshark.org:443
2 ......
```

```
3 New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
4 ......
```

原来，这里用的就是最新的 TLS1.3 版本，密码套件是 TLS_AES_256_GCM_SHA384。你有没有发现它跟 TLS1.2 的那些密码套件相比还有一个区别呢？比如跟下面这个 TLS1.2 的密码套件比较一下：

TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

相比之下，TLS1.3 的套件 **TLS_AES_256_GCM_SHA384** 是不是少了两个算法：身份验证和签名算法，还有密钥交换算法。不过，身份验证和签名算法倒是可以从证书里看到，它是 ECDSA。那密钥交换算法又到哪里去了呢？



其实，这是因为 TLS1.3 只允许前向加密（PFS）的密钥交换算法了，所以使用静态密钥的 RSA 已经被排除了，它默认使用的是 DHE 和 ECDHE，所以就不写在密码套件名称里了。

那么在这里，就又涉及了一个新的概念：前向加密。

前向加密（PFS）

前向加密又称为“完美前向加密”，它的英文就是 Forward Secrecy 和 Perfect Forward Secrecy。

虽然我前面刚提到 TLS1.3 去掉了 RSA，不过你不要误会了，TLS1.3 只是把 RSA 从密钥交换算法中排除了，但证书签名算法还是可以用 RSA 的。

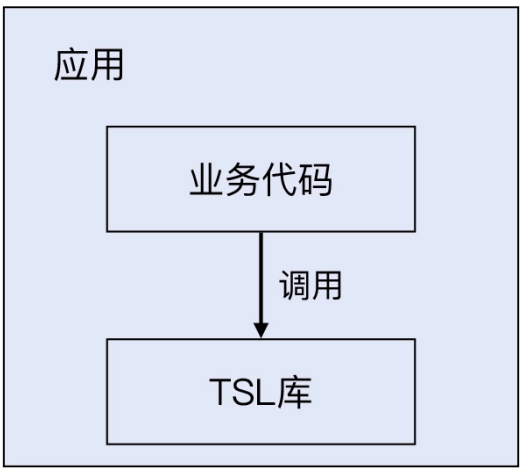
为什么 TLS1.3 强制要求前向加密呢？这是因为，如果在密钥交换的时候用非前向加密的算法（比如 RSA），那么一旦黑客取得了服务端私钥，并且抓取了历史上的 TLS 密文，他就可以用这个私钥和抓包文件，把这些 TLS 会话的对称密钥给还原出来，从而破解所有这些密文。因为可以把之前的密文都破解，RSA 就不属于“前向”加密。

人们发现，要解决这个问题的关键，就要做到：**每次参与协商对称密钥时的非对称密钥都不一样**。这样的话，即使黑客破解了其中一次会话的密钥，也无法用这个密钥破解其他会话。

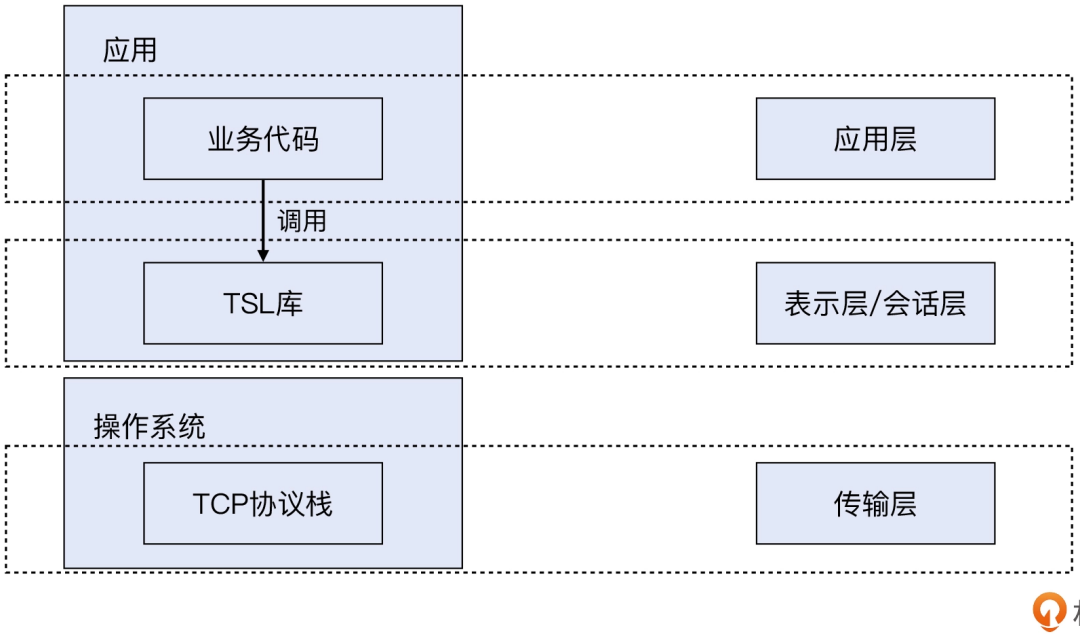
我们可以用一个例子来帮助理解“前向加密”。假设我们不断地生成一个个的保险箱，相当于一个个的 TLS 加密报文，如果每个箱子用同样的锁，那么一旦其中一把锁被破解，所有的保险箱都可以被打开了。用上“前向加密”锁之后，每次新的保险箱都用不同的锁，那么即使一把锁被破解，损失的只是一个保险箱，其他的箱子依旧安全。

TLS 的软件实现

TLS 只是一套协议，主要是“动动嘴皮子”，具体的活当然还是代码来干。目前应用最为广泛的 SSL/TLS 实现可能就是 OpenSSL 了，它既是一个开发库，也是一个命令行工具的名称。另外，NSS 和 GnuTLS 也是开源的 TLS 实现。应用程序会基于这些 TLS 库来实现 TLS 加解密功能。



有没有觉得这个很像 OSI 的分层模型？业务代码工作在应用层，TLS 库工作在表示层和会话层，两层之间有交互也有解耦，起到了很好的协同的效果。



学习完 TLS 加密原理，我们就要进入动手环节了，也就是期待已久的 TLS 抓包解密，让秘密不再是秘密。

客户端如何做 TLS 解密？

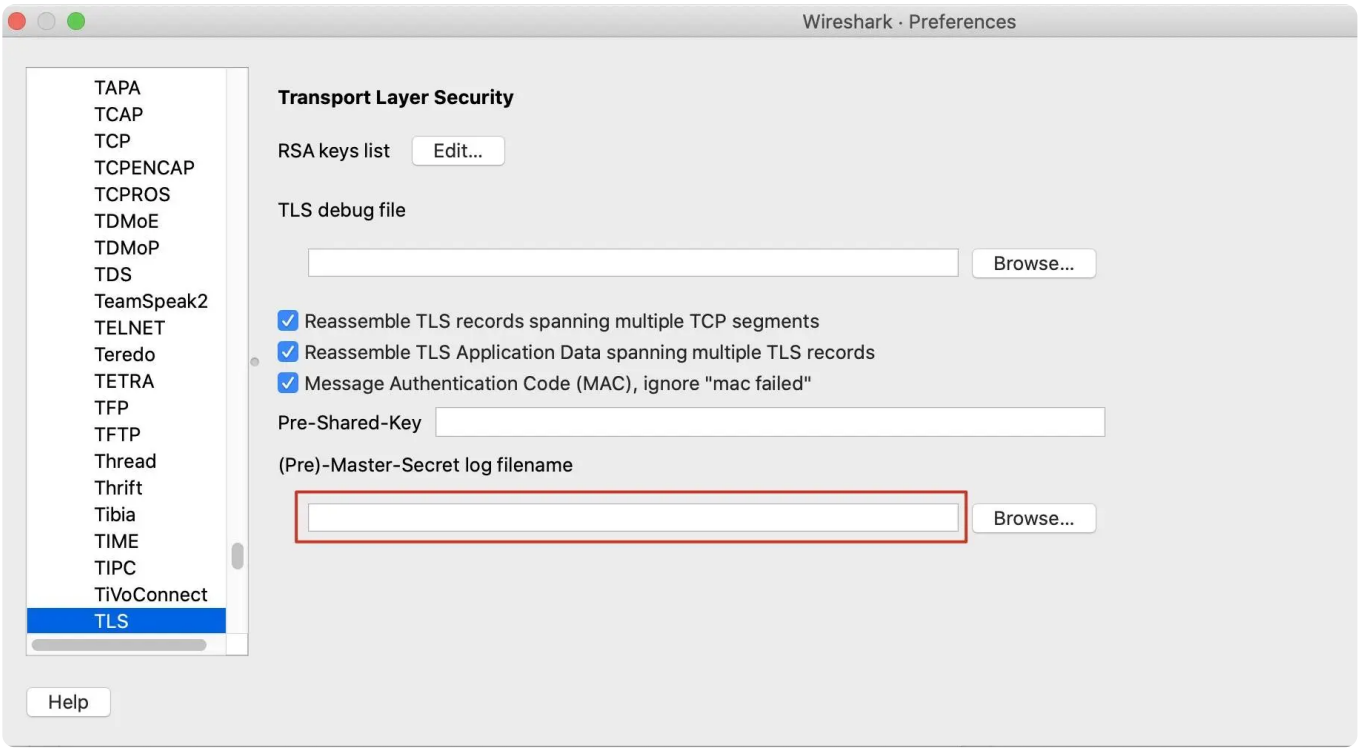
这里说的客户端，包括了 Chrome、Firefox 等浏览器，也包括 curl 这样的命令行工具。我在上节课里提过，为了把 TLS 解密，我们需要完成几个前提条件。其实这些前提条件就是下面这三件事：

创建一个用来存放 key 信息的日志文件，然后在系统里**配置一个环境变量 SSLKEYLOGFILE**，它的值就是这个文件的路径。

重启浏览器，启动抓包程序，然后访问 HTTPS 站点，此时 TLS 密钥信息将会导出到这个日志文件，而加密报文也会随着抓包，被保存到抓包文件中。

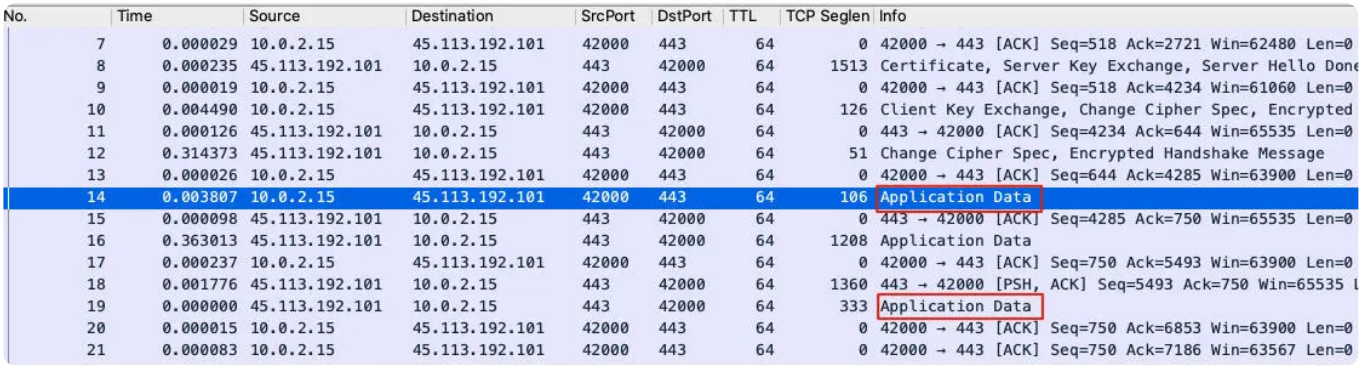
> 补充：如果是 Mac 又不想改动全局配置，那么你可以在 terminal 中的 `export SSLKEYLOGFILE=路径`，然后执行 `open "/Applications/Google\ Chrome.app"`，这时 Chrome 就继承了这个 shell 父进程的环境变量，而 terminal 退出后，这个环境变量就自动卸除了。

在 Wireshark 里，打开 Preferences 菜单，在 Protocol 列表里找到 TLS，然后把 **(Pre)-Master-Secret log filename 配置为那个文件的路径。**



在做完这三件事之后，我们用 Wireshark 打开抓包文件，就能看到解密后的报文了，比如 HTTP 请求和响应，还有 TLS 的控制信息，都会展示为明文。

比如，在默认情况下，我们看到的会是密文：



No.	Time	Source	Destination	SrcPort	DstPort	TTL	TCP Seglen	Info
7	0.000029	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=518 Ack=2721 Win=62480 Len=0
8	0.000235	45.113.192.101	10.0.2.15	443	42000	64	1513	Certificate, Server Key Exchange, Server Hello Done
9	0.000019	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=518 Ack=4234 Win=61060 Len=0
10	0.004490	10.0.2.15	45.113.192.101	42000	443	64	126	Client Key Exchange, Change Cipher Spec, Encrypted
11	0.000126	45.113.192.101	10.0.2.15	443	42000	64	0	443 → 42000 [ACK] Seq=4234 Ack=644 Win=65535 Len=0
12	0.314373	45.113.192.101	10.0.2.15	443	42000	64	51	Change Cipher Spec, Encrypted Handshake Message
13	0.000026	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=644 Ack=4285 Win=63900 Len=0
14	0.003807	10.0.2.15	45.113.192.101	42000	443	64	106	Application Data
15	0.000098	45.113.192.101	10.0.2.15	443	42000	64	0	443 → 42000 [ACK] Seq=4285 Ack=750 Win=65535 Len=0
16	0.363013	45.113.192.101	10.0.2.15	443	42000	64	1208	Application Data
17	0.000237	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=750 Ack=5493 Win=63900 Len=0
18	0.001776	45.113.192.101	10.0.2.15	443	42000	64	1360	443 → 42000 [PSH, ACK] Seq=5493 Ack=750 Win=65535 L
19	0.000000	45.113.192.101	10.0.2.15	443	42000	64	333	Application Data
20	0.000015	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=750 Ack=6853 Win=63900 Len=0
21	0.000083	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=750 Ack=7186 Win=63567 Len=0

> 补充：抓包示例文件已经上传至 Gitee，建议结合抓包文件和文稿一起学习。

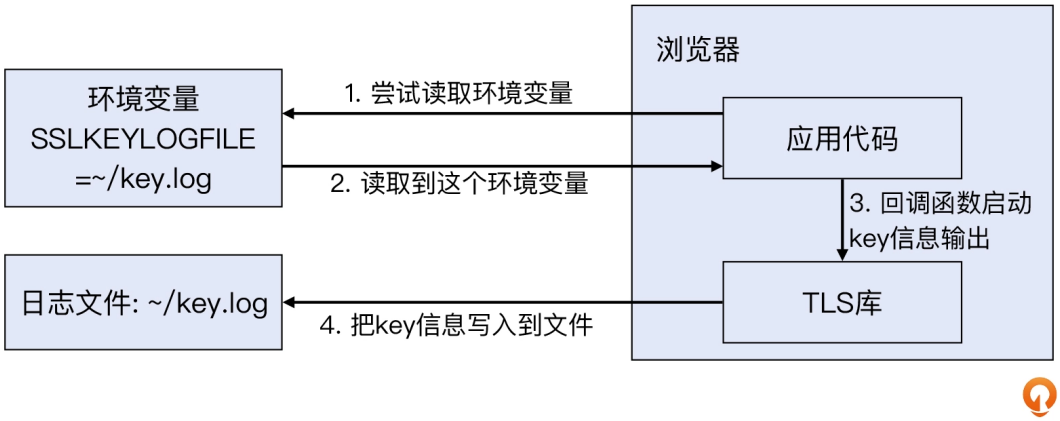
而配置解密步骤之后，看到的就是明文了：

No.	Time	Source	Destination	SrcPort	DstPort	TTL	TCP Seglen	Info
7	0.000029	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=518 Ack=2721 Win=62480 Len=0
8	0.000235	45.113.192.101	10.0.2.15	443	42000	64	1513	Certificate, Server Key Exchange, Server Hello Don
9	0.000019	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=518 Ack=4234 Win=61060 Len=0
10	0.004490	10.0.2.15	45.113.192.101	42000	443	64	126	Client Key Exchange, Change Cipher Spec, Finished
11	0.000126	45.113.192.101	10.0.2.15	443	42000	64	0	443 → 42000 [ACK] Seq=4234 Ack=644 Win=65535 Len=0
12	0.314373	45.113.192.101	10.0.2.15	443	42000	64	51	Change Cipher Spec, Finished
13	0.000026	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=644 Ack=4285 Win=63900 Len=0
14	0.003807	10.0.2.15	45.113.192.101	42000	443	64	106	GET / HTTP/1.1
15	0.000098	45.113.192.101	10.0.2.15	443	42000	64	0	443 → 42000 [ACK] Seq=4285 Ack=750 Win=65535 Len=0
16	0.363013	45.113.192.101	10.0.2.15	443	42000	64	1208	[TLS segment of a reassembled PDU]
17	0.000237	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=750 Ack=5493 Win=63900 Len=0
18	0.001776	45.113.192.101	10.0.2.15	443	42000	64	1360	443 → 42000 [PSH, ACK] Seq=5493 Ack=750 Win=65535
19	0.000000	45.113.192.101	10.0.2.15	443	42000	64	333	HTTP/1.1 200 OK (text/html)
20	0.000015	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=750 Ack=6853 Win=63900 Len=0
21	0.000083	10.0.2.15	45.113.192.101	42000	443	64	0	42000 → 443 [ACK] Seq=750 Ack=7186 Win=63567 Len=0

好了，你可以看到明文了，很多应用层的信息都可以辅助你做排查了，是不是有点小小的激动？

那么这背后的原理是什么呢？

其实是这样的：浏览器在启动过程中会尝试读取 SSLKEYLOGFILE 这个环境变量。如果存在这个变量，而它指向的又是一个有效的文件，那么浏览器就会做最为关键的事情了：它去调用 TLS 库，让 TLS 库把访问 HTTPS 站点的过程中的 key 信息导出到 SSLKEYLOGFILE 文件中。我画了一张示意图供你参考：



整个过程倒是不难理解，不过你可能会好奇：为什么这个日志文件有这么强大的能力，能解密 TLS？然后又不免担心，**如果这个文件“被坏人利用了”，该怎么办？**

所以我们还需要近距离地认识一下 SSLKEYLOGFILE。

SSLKEYLOGFILE

这个文件之所以能够解密 TLS，最关键的是，TLS 库把密钥交换阶段的核心信息 Master secret 导出到了这个文件中。基于这个信息，Wireshark 就可以还原出当时的对称密钥，

从而破解密文。

我们先来认识一下 SSLKEYLOGFILE 的格式。它是由很多条记录组成的，对于 TLS1.2 来说，每一行是一条记录，每一条记录由 3 个部分组成，中间用空格分隔，也就是下面这样：

<Label1> <ClientRandom1> <Secret1>

<Label2> <ClientRandom2> <Secret2>

……

这三个部分的具体含义是这样的。

Label：是对这条记录的描述，对于 TLS1.2 来说，这个值一般是 CLIENT_RANDOM。另外，RSA 也是一个可能的值，但是前面说过，因为 RSA 算法在密钥交换方面不是前向加密的，所以已经不推荐使用了。所以如果你在日志文件里看到 RSA，可能要小心一点，说明你的 TLS 不是前向加密的，所以并不是很安全。

ClientRandom：这个部分就是客户端生成的随机数，随机数原始长度为 32 字节，但在这里是用 16 进制表示的，每个字节用 16 进制表示就会成为 2 个字符，所以就变成了 64 个字符的字符串。我们在抓包文件里也能看到它，因为在密钥交换算法的设计中，ClientRandom 就是要在网络上公开传输的。

Secret：这就是 Master secret，也就是通过它可以生成对称密钥。Master secret 固定是 48 字节，也是十六进制表示的关系，成为 96 个字节的字符串。你应该明白了，这个 Master secret 就是最为关键的信息了，也正是黑客苦苦寻求的东西。它是万万不能在网络上传输的，自然也不可能在抓包文件里看到它，只有 TLS 库才能导出它。

> 补充：TLS1.3 的格式会很不一样，具体细节你可以参考这里的链接。

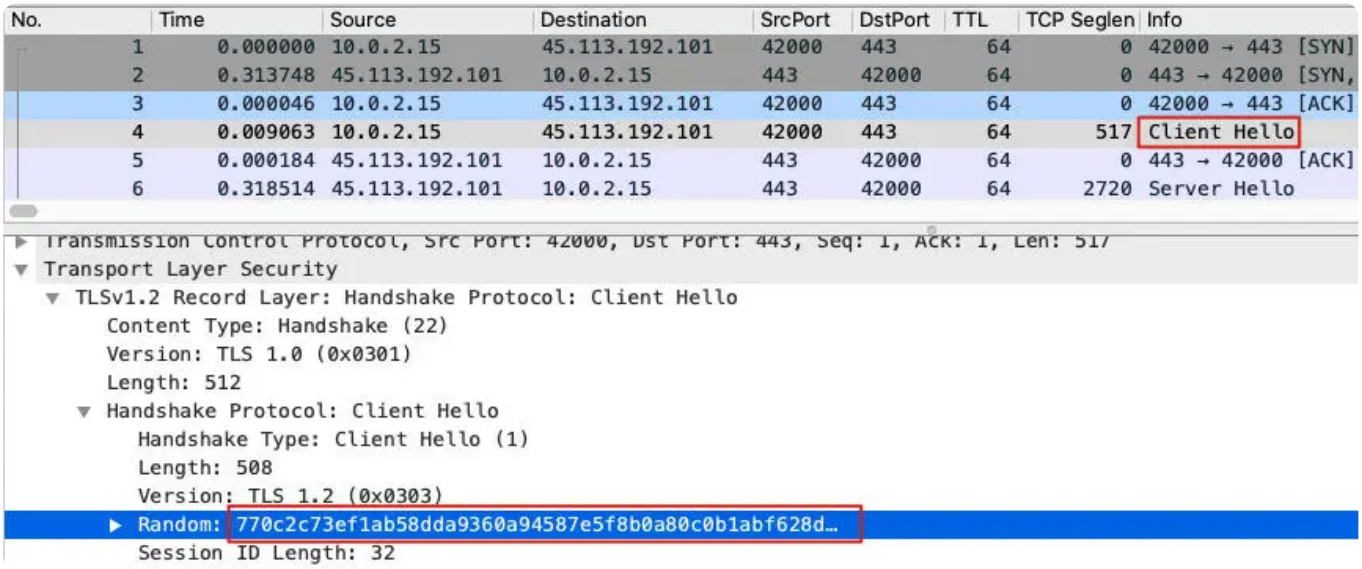
我们来看一个 TLS1.2 的 KEYLOGFILE 的具体的例子：

复制代码

```
1 CLIENT_RANDOM 770c2c73ef1ab58dda9360a94587e5f8b0a80c0b1abf628ddd7b55a118ec18ec
```

> 补充：这个日志文件也已经上传至Gitee，你可以按照前面介绍的 3 个步骤，结合上面的抓包文件和这个日志文件，自己来观察解密前后的区别。

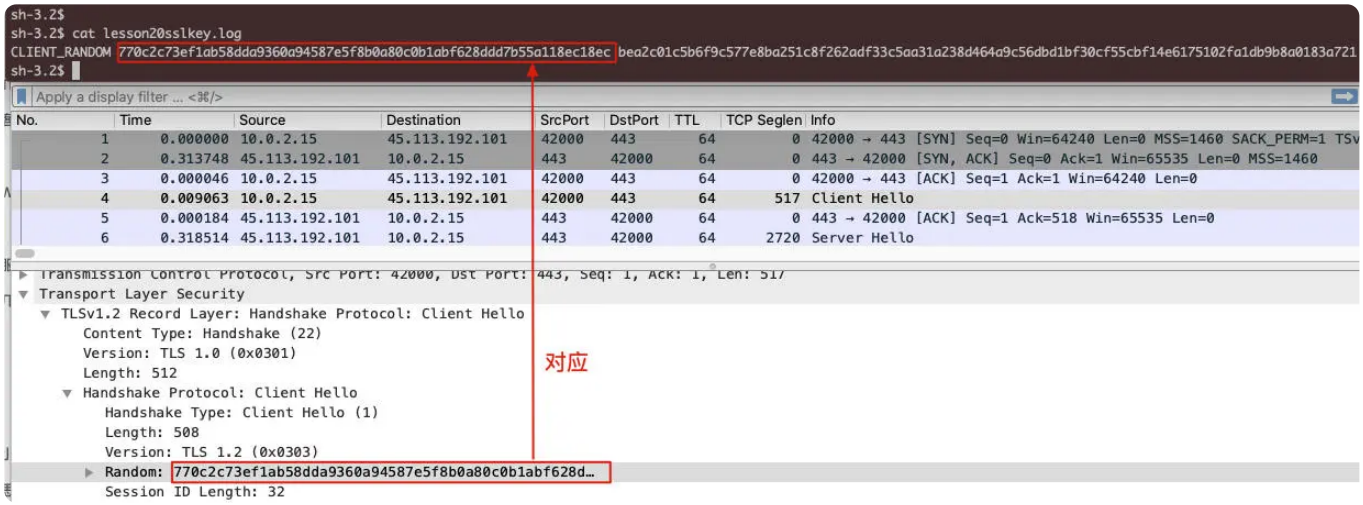
在输出这个 key 信息的时候，我们也做了对应的抓包，现在看一下抓包文件。我们选中 Client Hello 报文，点开 TLS 详情部分，继续点开 TLSv1.2 Record Layer -> Handshake Protocol -> Random，你看看它是不是就是前面日志文件里，第二列的值（开头是 770c）？



SSLKEYLOGFILE 日志文件格式我们了解了，接下来了解 Wireshark 是怎么跟它协同工作，解开密文的。

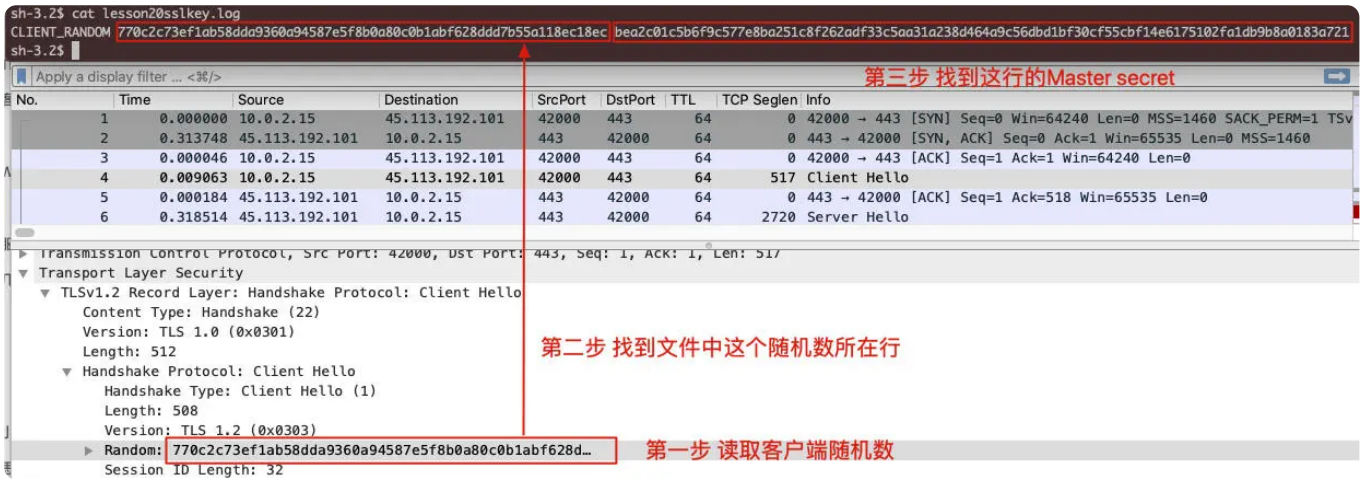
Wireshark 是怎么解开密文的？

在 TLS1.2 的 SSLKEYLOGFILE 中，每条记录的第一列是 CLIENT_RANDOM 这个字符串，第二列是这个 client random 的值，Wireshark 就是通过它，找到对应的 TLS 会话（你可以理解为是 TCP 流）。就像上图所示，通过这个随机数，就找到了这条 KEYLOG 记录对应的 TLS 会话了。



那么接下来，Wireshark 就知道真正的 Master secret 在哪里了：它就是前面匹配了这个客户端随机数的记录的第三列，也就是那 96 个字节的字符串。

由于在抓包文件里就有 ECDHE 密钥交换算法所需要的各种参数，结合这里的 Master secret，Wireshark 就可以解析出对称密钥，从而把密文解密了！



SYSKEYLOGFILE 的安全性

现在回答一下前面的问题：如果这个文件“被坏人利用了”，怎么办？

通过前面的学习，我们知道了：要想破解密文，既要有抓包文件，也要有 SSLKEYLOGFILE 日志文件，两者结合才能解密。而且不要忘了，即使你有抓包文件和日志文件，要是没抓到 TLS 握手阶段的报文，也还是不能解密，因为缺少了客户端随机数、加密算法参数等信息，Master secret 对你也是无法下嘴的美食。

服务端如何做 TLS 解密？

其实，上面的客户端做解密的过程，网络上已经有很多资料了，但是接下来要介绍的“服务端做 TLS 解密”这个话题，却鲜有人讨论。要知道，TLS 是双方的加密任务，但是我们一边倒地关心客户端如何解密，却对服务端的解密不闻不问，这又是什么道理呢？

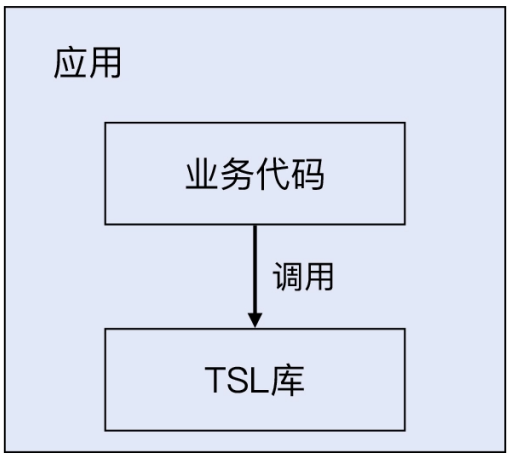
我觉得可能有这么几个原因：

多数人接触的还是以客户端为主，能在客户端解密已经满足了大多数的需求。而服务端只有一部分专职运维或者开发工程师在维护，关注度少了很多。

服务端一般有详细的日志，比如 Nginx 可以向日志里输出 HTTP 头部和性能数据，还可以输出 TLS 选择的密码套件等信息，这些在一般场景下也够用了。

很多服务端程序并没有提供 TLS 解密的功能，也就是想做抓包解密也做不了。而要自己实现这个特性，难度跟简单的参数配置，不在一个等级。难度高，可能是更加关键的原因。

其实，在软件架构上，服务端和客户端也是类似的，也是基于 TLS 库来构建 TLS 加解密的能力的。



极客时间

就我们 eBay 的情况来说，在我们的软件负载均衡方案中，第七层的部分是基于Envoy实现的。我们在把一套新系统搬上生产环境之前有一系列的考量，就像体检的检查表一样逐一校验。我们发现 Envoy 的体检就有一项“不合格”：Envoy 不提供对 TLS 流量进行抓包解密的功能。

> 补充：Envoy 是硅谷的共享出行公司 Lyft 于 2016 年发起的开源项目，可以认为是云原生时代的 Nginx。

在不少情况下，这个抓包解密特性对排查很有用。像一些商业产品比如 Netscaler 就是可以一边抓包，一边导出 TLS key。就跟我们在客户端做解密类似，我们结合这两个文件，就可以在 Wireshark 中既观察到 TCP 行为，也读到应用层信息了。

你可能会问：“Envoy 可以输出 Web 日志呀，把各种 HTTP 性能指标、访问头部，甚至包括用了什么 Cipher，都输出到文件，这样还不香吗？”

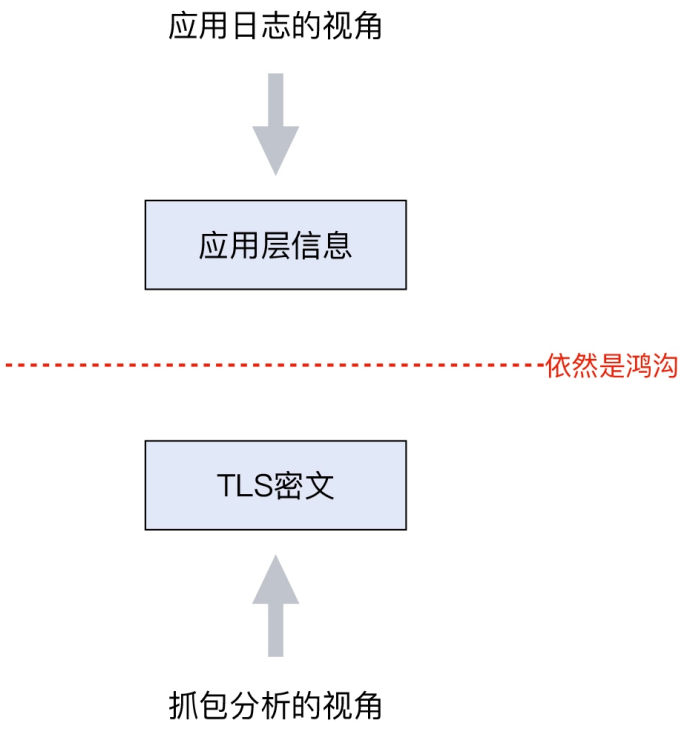
其实，我在开篇词和第 4 讲中都提到过网络排查的两大鸿沟。

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。

工具提示跟协议理解之间的鸿沟：你看得懂 Wireshark、tcpdump 这类工具的输出信息的含义，但就是无法真正地把它们跟你对协议的理解对应起来。

而包括 Envoy 在内的反向代理和 LB 软件，虽然也都提供了应用层日志，但跟实际的网络行为还有距离，这就是“应用现象跟网络现象之间的鸿沟”：日志是日志，网络是网络。如果日志里说某个 HTTP 请求耗时很长，你是无法知道网络上到底什么问题导致了这么长的耗时，是丢包引起了重传？还是没有丢包，纯粹是传输速度慢呢？

为了跨越第一个鸿沟，我们选择做 tcpdump 抓包。但是，如果抓取到的 TLS 密文无法被解密，就无法知道这些究竟是应用层的什么信息，这个鸿沟依然没有被跨越。



当然，我们可以选择“妥协”，采用一些灵活的策略来开展抓包分析。比如，可以把抓包分析的重心转移到 TCP 和 TLS 本身的层面，而不再关心其承载的应用层信息。但是“隔靴搔痒”总让排查工作不是特别“痛快”，我们犹豫许久之后，还是决定把这个解密的特性实现！

这并不是一件容易的事情。不过通过调研，我们发现，其实在服务端启用跟客户端类似的 TLS 解密功能，技术上是可行的，其中最为关键的信息就在 2017 年一位 Wireshark 开发工程师的演讲中：

> Applications using OpenSSL 1.1.1 or BoringSSL d28f59c27bac (2015-11-19) can be configured to dump keys: void SSLCTXsetkeylogcallback(SSL CTX *ctx, void (*cb)(const SSL *ssl, const char *line));

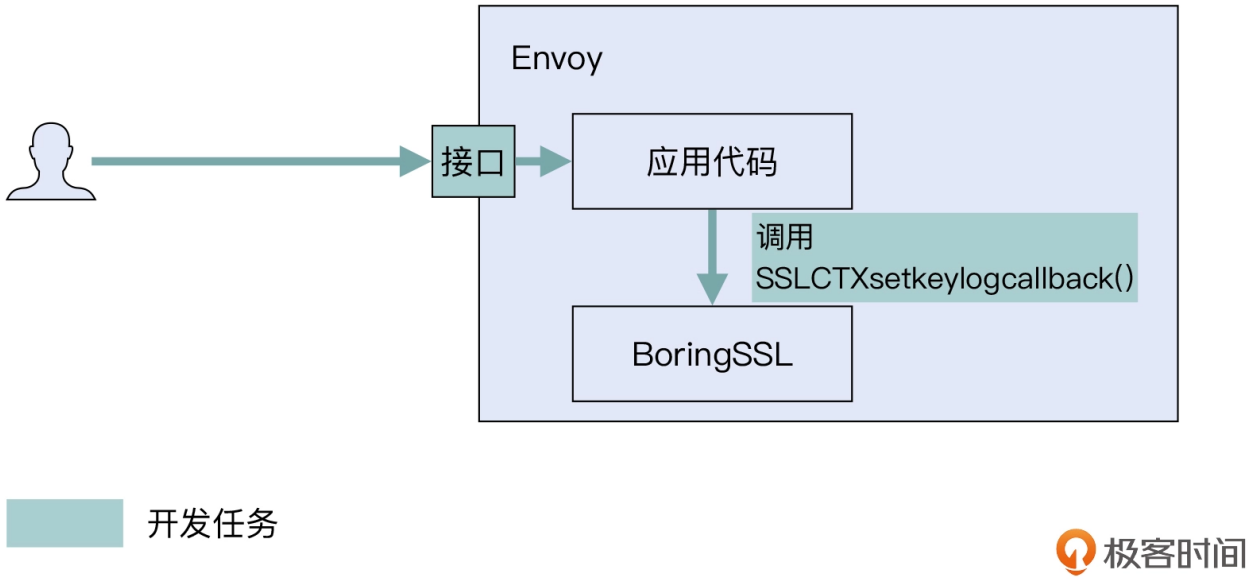
也就是说，只要我们使用这个 BoringSSL（是谷歌 Fork 自 OpenSSL 的项目）的 **SSLCTXsetkeylogcallback()** 回调函数，就可以把 TLS 信息导出来，于是我们信心大增。核心突破就是要把这个 BoringSSL 的回调函数给用起来。

具体来说，我们需要做这样的几件事：

在 Envoy 代码中增加调用 SSLCTXsetkeylogcallback() 函数的逻辑。

增加了对外的接口，使得用户可以通过某种方式让 Envoy 知道，它需要去使用这个调用逻辑。

第二点，其实就是一种接口方式，比如 SSLKEYLOGFILE 环境变量就是一种，我们也可以选择 API 接口，或者某种别的接口。总之，只要让程序（这里是 Envoy）知道：它需要去叫 BoringSSL 这个小弟去办点事情，整个功能就可以运作起来了。你可以参考这张示意图：



“体检通过”！我们在服务端 Envoy 上也可以做到方便的 TLS 解密了。其实，不仅实现了这个具体的需求本身，也实现了我们作为技术工作者，对“自我实现”的需求。

> 这个特性的主要开发者张博已经提交了 PR，相信不久之后我们就能在正式版的 Envoy 里用上这个特性了。

实现 Envoy 的 TLS 抓包解密的具体的做法跟客户端解密的步骤差不多：

调用 Envoy 接口，启用 SSL KEY 导出功能。

做 tcpdump 抓包，然后把抓包文件和 KEY 文件复制出来。

在 Wireshark 里同样配置好 **TLS 协议的 (Pre)-Master-Secret log filename**，打开抓包文件后，就可以跟在客户端类似，直接看到明文了。

几个问题

然后到这里，这里我们还需要搞清楚几个问题。

问题 1：我想实时查看解密信息行不行？

一个字的答案：行。只要设置好前面提及的 3 件事：

SSLKEYLOGFILE 环境变量；

之后再启动浏览器，然后直接在 Wireshark 里开始抓包；

设置 Wireshark 的 TLS 协议，配置 (Pre-)Master secret logfile。

这时候你访问 HTTPS 站点时，在 Wireshark 里看到的就直接是解密好的信息了！因为 Wireshark 已经能从 SSLKEYLOGFILE 里读取到密钥信息，同时又在实时地抓取到 TLS 密文，这种解密工作是可以实时进行的。

当然，这里还有一个小的注意点，我们在第二个问题里展开。

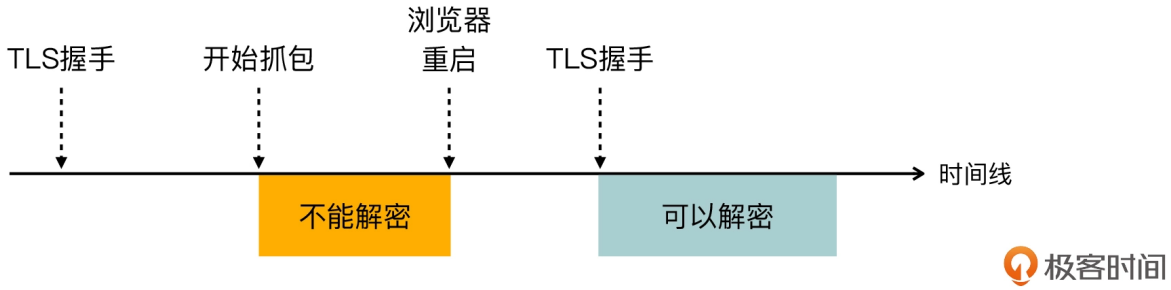
问题 2：为什么停止抓包后再启动抓包，抓包文件又变成密文了？

有同学就遇到这个问题：重启浏览器后，在 Wireshark 里马上就能看到 HTTP 数据包，确实能解密。但是停止抓包之后，再启动抓包，看到的又变成了 TLS 密文了。必须得重启浏览器才行。这是为何呢？

表面上看，这似乎又是一个“重启大法”的问题，但本质上呢？

我们知道，密文是用对称密钥加密的。而对称密钥的生成，是在 TLS 握手阶段完成的。我们前面提到过，Wireshark（也包括其他需要读取 SSLKEYLOGFILE 的程序）正是根据第二列的客户端随机数，来找到抓包文件中的 TLS session，然后运用第三列的 Master secret 来获取到对称密钥的。

抓包停止后，新的 HTTPS 请求所触发的 TLS 握手就不会被抓取到。这也就意味着，Wireshark 没有抓取到客户端随机数这个关键信息，尽管 SSLKEYLOGFILE 里依然在输出着一行行的 key 信息，但是 Wireshark 已经不知道用哪个 Master secret 了。自然，解密就无从做起。



而在浏览器重启后，事实上造成了 TLS 的重新握手，此时就又可以抓取到客户端随机数了，这样，解密工作就可以恢复。你看，这其实跟第 3 讲中，没抓到 TCP 握手报文就无法知道 Window Scale 参数这个问题差不多，也是关于握手的，只不过这次是 TLS 握手。“技术是相通的”，这句话真不是随便说说。

小结

这节课，我们通过对一张真实的 TLS 证书的解读，复习了各个加密算法在现实场景中的实现。你也需要重点掌握以下知识点：

证书中的 SAN 列表包括了它所支持的站点域名，所以只要被访问的站点名称在这个列表里，名称匹配就不是问题了。

证书中的域名通配符只支持一级域名，而不支持二级或者更多级的域名。

在 TLS1.3 中，密钥交换算法被强制要求是前向加密算法，所以默认采用 DHE 和 ECDHE，而 RSA 已经弃用。

RSA 依然可以作为可靠的身份验证和签名算法来使用。另外一种验证和签名算法是 ECDSA，它可以用更短的密钥实现跟 RSA 同样的密码强度。

前向加密可以防止黑客破解发生在过去的加密流量，提供了更好的安全性。

之后就是这节课的核心了：**如何做到对抓包文件进行解密**。这里又分客户端和服务端两个不同场景，你也需要重点关注。

首先，在客户端做抓包解密，需要做三件事：

创建一个文件，并设置为 SSLKEYLOGFILE 这个环境变量的值；

重启浏览器，开始做抓包，此时 key 信息被浏览器自动导入到日志文件；

在 Wireshark 里把该日志文件配置为 TLS 的 (Pre)-Mater-Secret log filename。

这样，我们就能在 Wireshark 里直接读取到应用层信息了。

而在服务端抓包解密，就要依托于软件实现了，但是有些软件并没有提供这种功能，比如 Envoy。借助底层 BoringSSL 库的接口，eBay 流量管理团队实现了对这个接口的调用，我们也可以在 Envoy 上完成抓包解密了。

另外，你还要知道 **Wireshark 能解读出密文的原理**：

从抓包文件中定位到 client random；

从日志文件中找到同样这个 client random，然后找到紧跟着的 Master secret；

用这个 Master secret 导出对称密钥，最后把密文解密。

思考题

最后，给你留两道思考题：

DH、DHE、ECDHE，这三者的联系和区别是什么呢？

浏览器会根据 SSLKEYLOGFILE 这个环境变量，把 key 信息导出到相应的文件，那么 curl 也会读取这个变量并导出 key 信息吗？

欢迎你把答案分享到留言区，我们一起交流、进步。

上一篇 19 | TLS的各种特性：TLS握手为什么会失败？

下一篇 21 | 为什么用了负载均衡更加不均衡？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。