**HTTPS协议详解**

**1 前言**

百度已经于近日上线了全站 HTTPS 的安全搜索，默认会将 HTTP 请求跳转成 HTTPS。本文重点介绍 HTTPS 协议, 并简单介绍部署全站 HTTPS 的意义。

**2 HTTPS 协议概述**

HTTPS 可以认为是 HTTP + TLS。HTTP 协议大家耳熟能详了，目前大部分 WEB 应用和网站都是使用 HTTP 协议传输的。

TLS 是传输层加密协议，它的前身是 SSL 协议，最早由 netscape 公司于 1995 年发布，1999 年经过 IETF 讨论和规范后，改名为 TLS。如果没有特别说明，SSL 和 TLS 说的都是同一个协议。

HTTP 和 TLS 在协议层的位置以及 TLS 协议的组成如下图：

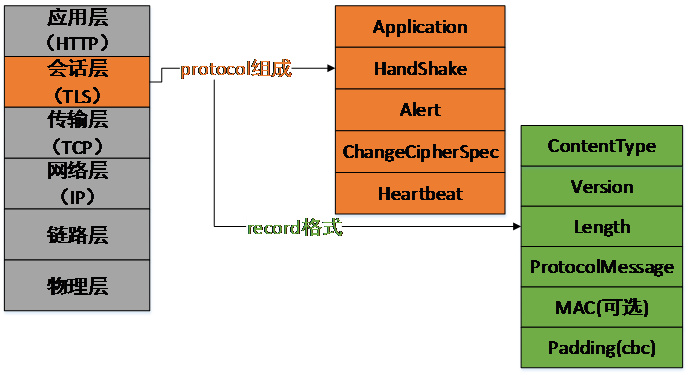


图 1 TLS 协议格式

TLS 协议主要有五部分：应用数据层协议，握手协议，报警协议，加密消息确认协议，心跳协议。

TLS 协议本身又是由 record 协议传输的，record 协议的格式如上图最右所示。

目前常用的 HTTP 协议是 HTTP1.1，常用的 TLS 协议版本有如下几个：TLS1.2, TLS1.1, TLS1.0 和 SSL3.0。其中 SSL3.0 由于 POODLE 攻击已经被证明不安全，但统计发现依然有不到 1% 的浏览器使用 SSL3.0。TLS1.0 也存在部分安全漏洞，比如 RC4 和 BEAST 攻击。

TLS1.2 和 TLS1.1 暂时没有已知的安全漏洞，比较安全，同时有大量扩展提升速度和性能，推荐大家使用。

需要关注一点的就是 TLS1.3 将会是 TLS 协议一个非常重大的改革。不管是安全性还是用户访问速度都会有质的提升。不过目前没有明确的发布时间。

同时 HTTP2 也已经正式定稿，这个由 SPDY 协议演化而来的协议相比 HTTP1.1 又是一个非常重大的变动，能够明显提升应用层数据的传输效率。

**3 HTTPS 功能介绍**

百度使用 HTTPS 协议主要是为了保护用户隐私，防止流量劫持。

HTTP 本身是明文传输的，没有经过任何安全处理。例如用户在百度搜索了一个关键字，比如“苹果手机”，中间者完全能够查看到这个信息，并且有可能打电话过来骚扰用户。也有一些用户投诉使用百度时，发现首页或者结果页面浮了一个很长很大的广告，这也肯定是中间者往页面插的广告内容。如果劫持技术比较低劣的话，用户甚至无法访问百度。

这里提到的中间者主要指一些网络节点，是用户数据在浏览器和百度服务器中间传输必须要经过的节点。比如 WIFI 热点，路由器，防火墙，反向代理，缓存服务器等。

在 HTTP 协议下，中间者可以随意嗅探用户搜索内容，窃取隐私甚至篡改网页。不过 HTTPS 是这些劫持行为的克星，能够完全有效地防御。

总体来说，HTTPS 协议提供了三个强大的功能来对抗上述的劫持行为：

1，  内容加密。浏览器到百度服务器的内容都是以加密形式传输，中间者无法直接查看原始内容。

2，  身份认证。保证用户访问的是百度服务，即使被 DNS 劫持到了第三方站点，也会提醒用户没有访问百度服务，有可能被劫持

3，  数据完整性。防止内容被第三方冒充或者篡改。

那 HTTPS 是如何做到上述三点的呢？下面从原理角度介绍一下。

**4 HTTPS 原理介绍**

**4.1 内容加密**

加密算法一般分为两种，对称加密和非对称加密。所谓对称加密（也叫密钥加密）就是指加密和解密使用的是相同的密钥。而非对称加密（也叫公钥加密）就是指加密和解密使用了不同的密钥。

图 2 对称加密



图 3 非对称加密

对称内容加密强度非常高，一般破解不了。但存在一个很大的问题就是无法安全地生成和保管密钥。假如客户端软件和服务器之间每次会话都使用固定的，相同的密钥加密和解密，肯定存在很大的安全隐患。如果有人从客户端端获取到了对称密钥，整个内容就不存在安全性了，而且管理海量的客户端密钥也是一件很复杂的事情。

非对称加密主要用于密钥交换（也叫密钥协商），能够很好地解决这个问题。浏览器和服务器每次新建会话时都使用非对称密钥交换算法协商出对称密钥，使用这些对称密钥完成应用数据的加解密和验证，整个会话过程中的密钥只在内存中生成和保存，而且每个会话的对称密钥都不相同（除非会话复用），中间者无法窃取。

非对称密钥交换很安全，但同时也是 HTTPS 性能和速度严重降低的“罪魁祸首”。想要知道 HTTPS 为什么影响速度，为什么消耗资源，就一定要理解非对称密钥交换的整个过程。

下面重点介绍一下非对称密钥交换的数学原理及在 TLS 握手过程中的应用。

**4.1.1 非对称密钥交换**

在非对称密钥交换算法出现以前，对称加密一个很大的问题就是不知道如何安全生成和保管密钥。非对称密钥交换过程主要就是为了解决这个问题，使得对称密钥的生成和使用更加安全。

密钥交换算法本身非常复杂，密钥交换过程涉及到随机数生成，模指数运算，空白补齐，加密，签名等操作。

常见的密钥交换算法有 RSA，ECDHE，DH，DHE 等算法。它们的特性如下：

* RSA：算法实现简单，诞生于 1977 年，历史悠久，经过了长时间的破解测试，安全性高。缺点就是需要比较大的素数（目前常用的是 2048 位）来保证安全强度，很消耗 CPU 运算资源。RSA 是目前唯一一个既能用于密钥交换又能用于证书签名的算法。
* DH：diffie-hellman 密钥交换算法，诞生时间比较早（1977 年），但是 1999 年才公开。缺点是比较消耗 CPU 性能。
* ECDHE：使用椭圆曲线（ECC）的 DH 算法，优点是能用较小的素数（256 位）实现 RSA 相同的安全等级。缺点是算法实现复杂，用于密钥交换的历史不长，没有经过长时间的安全攻击测试。
* ECDH：不支持 PFS，安全性低，同时无法实现 false start。
* DHE：不支持 ECC。非常消耗 CPU 资源。

建议优先支持 RSA 和 ECDH\_RSA 密钥交换算法。原因是：

1，  ECDHE 支持 ECC 加速，计算速度更快。支持 PFS，更加安全。支持 false start，用户访问速度更快。

2，  目前还有至少 20% 以上的客户端不支持 ECDHE，我们推荐使用 RSA 而不是 DH 或者 DHE，因为 DH 系列算法非常消耗 CPU（相当于要做两次 RSA 计算）。



需要注意通常所说的 ECDHE 密钥交换默认都是指 ECDHE\_RSA，使用 ECDHE 生成 DH 算法所需的公私钥，然后使用 RSA 算法进行签名最后再计算得出对称密钥。

非对称加密相比对称加密更加安全，但也存在两个明显缺点：

1，  CPU 计算资源消耗非常大。一次完全 TLS 握手，密钥交换时的非对称解密计算量占整个握手过程的 90% 以上。而对称加密的计算量只相当于非对称加密的 0.1%，如果应用层数据也使用非对称加解密，性能开销太大，无法承受。

2，  非对称加密算法对加密内容的长度有限制，不能超过公钥长度。比如现在常用的公钥长度是 2048 位，意味着待加密内容不能超过 256 个字节。

所以公钥加密目前只能用来作密钥交换或者内容签名，不适合用来做应用层传输内容的加解密。

非对称密钥交换算法是整个 HTTPS 得以安全的基石，充分理解非对称密钥交换算法是理解 HTTPS 协议和功能的关键。

下面分别通俗地介绍一下 RSA 和 ECDHE 在密钥交换过程中的应用。

**4.1.1.1 RSA 密钥协商**

**4.1.1.1.1 RSA 算法介绍**

RSA 算法的安全性是建立在乘法不可逆或者大数因子很难分解的基础上。RSA 的推导和实现涉及到了欧拉函数和费马定理及模反元素的概念，有兴趣的读者可以自行百度。

RSA 算法是统治世界的最重要算法之一，而且从目前来看，RSA 也是 HTTPS 体系中最重要的算法，没有之一。

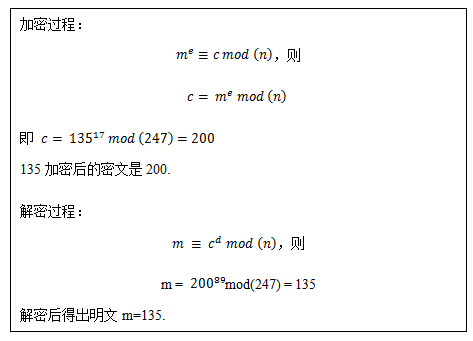
RSA 的计算步骤如下：

1，  随机挑选两个质数 p, q，假设 p = 13, q = 19。 n = p \* q = 13 \* 19 = 247;

2，  ∅(n) 表示与整数 n 互质数的个数。如果 n 等于两个质数的积，则∅(n)=(p-1)(q-1) 挑选一个数 e，满足 1< e <∅(n) 并且 e 与互质，假设 e = 17;

3，  计算 e 关于 n 的模反元素, ed=1 mod ∅(n) , 由 e = 17 ,∅(n) =216  可得 d = 89;

4，  求出了 e，和 d，假设明文 m = 135，密文用 c 表示。那么加解密计算如下：



实际应用中，(n,e) 组成了公钥对，（n,d）组成了私钥对，其中 n 和 d 都是一个接近 22048的大数。即使现在性能很强的 CPU，想要计算 m≡c^d mod(n)，也需要消耗比较大的计算资源和时间。

公钥对 (n, e) 一般都注册到了证书里，任何人都能直接查看，比如百度证书的公钥对如下图，其中最末 6 个数字（010001）换算成 10 进制就是 65537，也就是公钥对中的 e。e 取值比较小的好处有两个：

1，  由 c=m^e mod(n) 可知，e 较小，客户端 CPU 计算消耗的资源较少。

2，  加大 server 端的破解难度。e 比较小，私钥对中的 d 必然会非常大。所以 d 的取值空间也就非常大，增加了破解难度。

那为什么 (n,e) 能做为公钥公开，甚至大家都能直接从证书中查看到，这样安全吗？分析如下：

由于 ed≡1 mod ∅(n)，知道了 e 和 n，想要求出私钥 d，就必须知道∅(n)。而∅(n)=(p-1)\*(q-1)，必须计算出 p 和 q 才能确定私钥 d。但是当 n 大到一定程度时（比如接近 2^2048），即使现在最快的 CPU 也无法进行这个因式分解，即无法知道 n 是由哪个数 p 和 q 乘出来的。所以就算知道了公钥，整个加解密过程还是非常安全的。

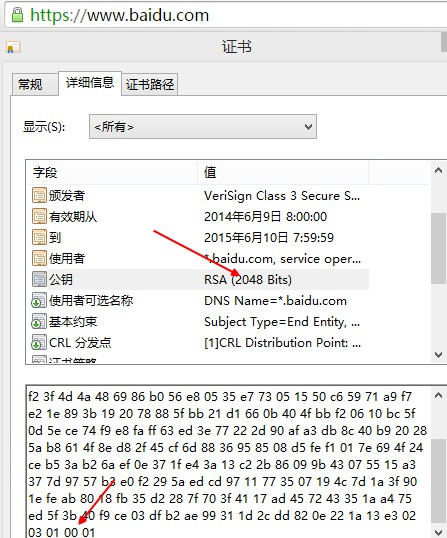


图 5 百度 HTTPS 证书公钥

**4.1.1.1.2 握手过程中的 RSA 密钥协商**

介绍完了 RSA 的原理，那最终会话所需要的对称密钥是如何生成的呢？跟 RSA 有什么关系？

以 TLS1.2 为例简单描述一下，省略跟密钥交换无关的握手消息。过程如下：

1，  浏览器发送 client\_hello，包含一个随机数 random1。

2，  服务端回复 server\_hello，包含一个随机数 random2，同时回复 certificate，携带了证书公钥 P。

3，  浏览器接收到 random2 之后就能够生成 premaster\_secrect 以及 master\_secrect。其中 premaster\_secret 长度为 48 个字节，前 2 个字节是协议版本号，剩下的 46 个字节填充一个随机数。结构如下：



|  |  |
| --- | --- |
| 1 | Struct {byte Version[2];bute random[46];} |

master secrect 的生成算法简述如下：



|  |  |
| --- | --- |
| 1 | Master\_key = PRF(premaster\_secret, “master secrect”, 随机数1+随机数2)其中 PRF 是一个随机函数，定义如下：PRF(secret, label, seed) = P\_MD5(S1, label + seed)  XOR  P\_SHA-1(S2, label + seed) |

而 master secrect 包含了六部分内容，分别是用于校验内容一致性的密钥，用于对称内容加解密的密钥，以及初始化向量（用于 CBC 模式），客户端和服务端各一份。从上式可以看出，把 premaster\_key 赋值给 secret，”master key”赋值给 label，浏览器和服务器端的两个随机数做种子就能确定地求出一个 48 位长的随机数。

至此，浏览器侧的密钥已经完成协商。

4，  浏览器使用证书公钥 P 将 premaster\_secrect 加密后发送给服务器。

5，  服务端使用私钥解密得到 premaster\_secrect。又由于服务端之前就收到了随机数 1，所以服务端根据相同的生成算法，在相同的输入参数下，求出了相同的 master secrect。

RSA 密钥协商握手过程图示如下：

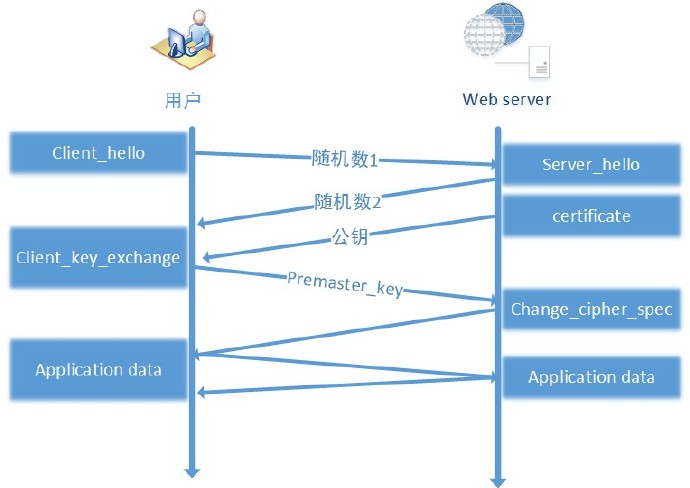


图 6 RSA 密钥协商过程

可以看出，密钥协商过程需要 2 个 RTT，这也是 HTTPS 慢的一个重要原因。而 RSA 发挥的关键作用就是对 premaster\_secrect 进行了加密和解密。中间者不可能破解 RSA 算法，也就不可能知道 premaster\_secrect，从而保证了密钥协商过程的安全性。

**4.1.1.2 ECDHE 密钥协商**

**4.1.1.2.1 DH 与 ECC 算法原理**

ECDHE 算法实现要复杂很多，主要分为两部分：diffie-hellman 算法（简称为 DH）及 ECC（椭圆曲线算术）。他们的安全性都是建立在离散对数计算很困难的基础上。

简单介绍一下 dh 算法的实现，先介绍两个基本概念：

* 本原根：如果整数 a 是素数 p 的本原根，则 a, a^2, …, a^(p-1) 在 mod p 下都不相同。
* 离散对数：对任意整数 b 和素数 p 的本原根 a，存在唯一的指数 i 满足:

b ≡ a^i mod p (0≤i≤p-1)

则称 i 是 b 的以 a 为底的模 p 的离散对数。

理解这两个概念，dh 算法就非常简单了，示例如下：

假设 client 和 server 需要协商密钥，p=2579，则本原根 a = 2。

1，  Client 选择随机数 Kc = 123 做为自己的私钥，计算 Yc = a^Kc  mod p = 2^123 mod 2579 = 2400，把 Yc 作为公钥发送给 server。

2，  Server 选择随机数 Ks = 293 作为私钥，计算 Ys = a^Ks  mod p = s^293 mod 2579 = 968，把 Ys 作为公钥发送给 client。

3，  Client 计算共享密钥：secrect =  Ys^Kc mod (p) = 968^123  mod(2579) = 434

4，  Server 计算共享密钥：secrect = Yc^Ks mod(p) =2400^293 mod(2579) =434

上述公式中的 Ys,Yc,P, a, 都是公开信息，可以被中间者查看，只有 Ks,Kc 作为私钥没有公开，当私钥较小时，通过穷举攻击能够计算出共享密钥，但是当私钥非常大时，穷举攻击肯定是不可行的。

DH 算法有一个比较大的缺陷就是需要提供足够大的私钥来保证安全性，所以比较消耗 CPU 计算资源。ECC 椭圆曲线算术能够很好的解决这个问题，224 位的密钥长度就能达到 RSA2048 位的安全强度。

ECC 的曲线公式描述的其实不是椭圆，只是跟椭圆曲线周长公式形似才叫椭圆曲线加密算术。ECC 涉及到了有限域、群等近世代数的多个概念，就不做详细介绍了。

ECC 安全性依赖于这样一个事实：



|  |  |
| --- | --- |
| 1 | P = kQ, 已知 k, Q 求出 P 相对简单，但是已知 P 和 Q 求出 k 却非常困难。 |

上式看起来非常简单，但有如下约束条件：

1，  Q 是一个非常大的质数，p, k, q 都是椭圆曲线有限域上的离散点。

2，  有限域定义了自己的加法和乘法法则，即使 kQ 的运算也非常复杂。

ECC 应用于 Diffie-Hellman 密钥交换过程如下：

1，  定义一个满足椭圆方程的有限域，即挑选 p, a, b 满足如下方程：



|  |  |
| --- | --- |
| 1 | y^2 mod p = (x^3+ax +b) mod p |

2，  挑选基点 G = (x, y)，G 的阶为 n。n 为满足 nG = 0 的最小正整数。

3，  Client 选择私钥 Kc (0 <Kc<n )，产生公钥 Yc =Kc \*G

4，  server 选择私钥 Ks 并产生公钥 Ys =Ks\*G

5，  client 计算共享密钥 K = Kc\*Ys   ，server 端计算共享密钥 Ks\*Yc ，这两者的结果是一样的，因为：



|  |  |
| --- | --- |
| 1 | Kc\*Ys = Kc\*(Ks\*G) = Ks\*(Kc\*G) = Ks\*Yc |

由上面描述可知，只要确定 p, a, b 就能确定一条有限域上的椭圆曲线，由于不是所有的椭圆曲线都能够用于加密，所以 p, a, b 的选取非常讲究，直接关系曲线的安全性和计算速度。

Openssl 实现的，也是 FIPS 推荐的 256 位素数域上的椭圆曲线参数定义如下：



|  |  |
| --- | --- |
| 1 | 质数 p = 115792089210356248762697446949407573530086143415290314195533631308867097853951 阶 n = 115792089210356248762697446949407573529996955224135760342422259061068512044369SEED = c49d3608 86e70493 6a6678e1 139d26b7 819f7e90c = 7efba166 2985be94 03cb055c 75d4f7e0 ce8d84a9 c5114abcaf317768 0104fa0d 椭圆曲线的系数 a = 0 椭圆曲线的系统 b = 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f63bce3c3e 27d2604b 基点 G x = 6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0f4a13945 d898c296 基点 G y = 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ececbb64068 37bf51f5 |

**4.1.1.2.2 握手过程中的 ECDHE 密钥协商**

简单介绍了 ECC 和 DH 算法的数学原理，我们看下 ECDHE 在 TLS 握手过程中的应用。

相比 RSA，ECDHE 需要多发送一个 server\_key\_exchange 的握手消息才能完成密钥协商。

同样以 TLS1.2 为例，简单描述一下过程：

1，  浏览器发送 client\_hello，包含一个随机数 random1，同时需要有 2 个扩展：

a)         Elliptic\_curves：客户端支持的曲线类型和有限域参数。现在使用最多的是 256 位的素数域，参数定义如上节所述。

b)        Ec\_point\_formats：支持的曲线点格式，默认都是 uncompressed。

2，  服务端回复 server\_hello，包含一个随机数 random2 及 ECC 扩展。

3，  服务端回复 certificate，携带了证书公钥。

4，  服务端生成 ECDH 临时公钥，同时回复 server\_key\_exchange，包含三部分重要内容：

a)         ECC 相关的参数。

b)        ECDH 临时公钥。

c)         ECC 参数和公钥生成的签名值，用于客户端校验。

5，  浏览器接收 server\_key\_exchange 之后，使用证书公钥进行签名解密和校验，获取服务器端的 ECDH 临时公钥，生成会话所需要的共享密钥。

至此，浏览器端完成了密钥协商。

6，  浏览器生成 ECDH 临时公钥和 client\_key\_exchange 消息，跟 RSA 密钥协商不同的是，这个消息不需要加密了。

7，  服务器处理 client\_key\_exchang 消息，获取客户端 ECDH 临时公钥。

8，  服务器生成会话所需要的共享密钥。

9，  Server 端密钥协商过程结束。

图示如下：

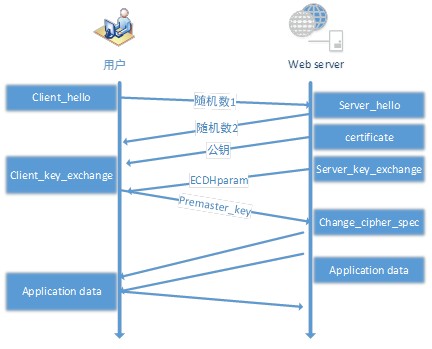


图 7 ECDHE 密钥协商过程

**4.1.2 对称内容加密**

非对称密钥交换过程结束之后就得出了本次会话需要使用的对称密钥。对称加密又分为两种模式：流式加密和分组加密。流式加密现在常用的就是 RC4，不过 RC4 已经不再安全，微软也[建议网站尽量不要使用 RC4 流式加密](http://blogs.technet.com/b/srd/archive/2013/11/12/security-advisory-2868725-recommendation-to-disable-rc4.aspx)。

一种新的替代 RC4 的流式加密算法叫 ChaCha20，它是 google 推出的速度更快，更安全的加密算法。目前已经被 android 和 chrome 采用，也编译进了 google 的开源 openssl 分支 —boring ssl，并且[nginx 1.7.4 也支持编译 boringssl](http://forum.nginx.org/read.php?27,252308,252308" \l "msg-252308)。

分组加密以前常用的模式是 AES-CBC，但是 CBC 已经被证明容易遭受[BEAST](http://resources.infosecinstitute.com/beast-vs-crime-attack/)和[LUCKY13 攻击](http://www.isg.rhul.ac.uk/tls/TLStiming.pdf)。目前建议使用的分组加密模式是 AES-GCM，不过它的缺点是计算量大，性能和电量消耗都比较高，不适用于移动电话和平板电脑。

**4.2 身份认证**

身份认证主要涉及到 PKI 和数字证书。通常来讲 PKI（公钥基础设施）包含如下部分：

* End entity：终端实体，可以是一个终端硬件或者网站。
* CA：证书签发机构。
* RA：证书注册及审核机构。比如审查申请网站或者公司的真实性。
* CRL issuer：负责证书撤销列表的发布和维护。
* Repository：负责数字证书及 CRL 内容存储和分发。

申请一个受信任的数字证书通常有如下流程：

1，  终端实体生成公私钥和证书请求。

2，  RA 检查实体的合法性。如果个人或者小网站，这一步不是必须的。

3，  CA 签发证书，发送给申请者。

4，  证书更新到 repository，终端后续从 repository 更新证书，查询证书状态等。

目前百度使用的证书是 X509v3 格式，由如下三个部分组成：

1，  tbsCertificate(to be signed certificate 待签名证书内容)，这部分包含了 10 个要素，分别是版本号，序列号，签名算法标识，发行者名称，有效期，证书主体名，证书主体公钥信息，发行商唯一标识，主体唯一标识，扩展等。

2，  signatureAlgorithm，签名算法标识，指定对 tbsCertificate 进行签名的算法。

3，  signaturValue（签名值），使用 signatureAlgorithm 对 tbsCertificate 进行计算得到签名值。

数字证书有两个作用：

1，  身份授权。确保浏览器访问的网站是经过 CA 验证的可信任的网站。

2，  分发公钥。每个数字证书都包含了注册者生成的公钥。在 SSL 握手时会通过 certificate 消息传输给客户端。比如前文提到的 RSA 证书公钥加密及 ECDHE 的签名都是使用的这个公钥。

申请者拿到 CA 的证书并部署在网站服务器端，那浏览器发起握手接收到证书后，如何确认这个证书就是 CA 签发的呢？怎样避免第三方伪造这个证书？

答案就是数字签名（digital signature）。数字签名是证书的防伪标签，目前使用最广泛的 SHA-RSA 数字签名的制作和验证过程如下：

1，  数字签名的签发。首先是使用哈希函数对待签名内容进行安全哈希，生成消息摘要，然后使用 CA 自己的私钥对消息摘要进行加密。

2，  数字签名的校验。使用 CA 的公钥解密签名，然后使用相同的签名函数对待签名证书内容进行签名并和服务端数字签名里的签名内容进行比较，如果相同就认为校验成功。

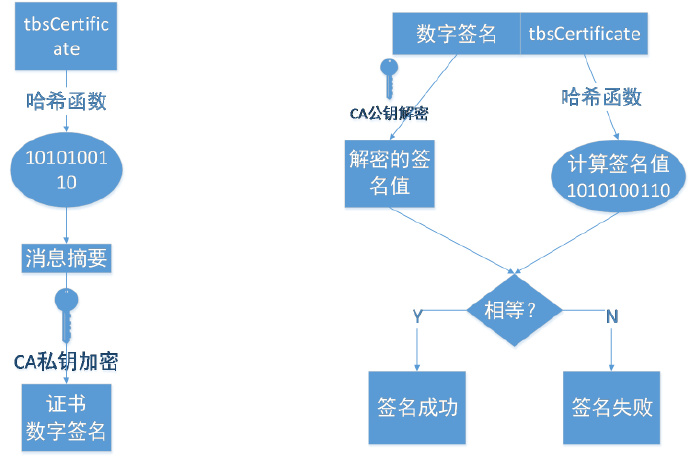


图 8 数字签名生成及校验

这里有几点需要说明：

1. 数字签名签发和校验使用的密钥对是 CA 自己的公私密钥，跟证书申请者提交的公钥没有关系。
2. 数字签名的签发过程跟公钥加密的过程刚好相反，即是用私钥加密，公钥解密。
3. 现在大的 CA 都会有证书链，证书链的好处一是安全，保持根 CA 的私钥离线使用。第二个好处是方便部署和撤销，即如果证书出现问题，只需要撤销相应级别的证书，根证书依然安全。
4. 根 CA 证书都是自签名，即用自己的公钥和私钥完成了签名的制作和验证。而证书链上的证书签名都是使用上一级证书的密钥对完成签名和验证的。
5. 怎样获取根 CA 和多级 CA 的密钥对？它们是否可信？当然可信，因为这些厂商跟浏览器和操作系统都有合作，它们的公钥都默认装到了浏览器或者操作系统环境里。比如[firefox 就自己维护了一个可信任的 CA 列表](http://mxr.mozilla.org/mozilla-central/source/security/nss/lib/ckfw/builtins/certdata.txt)，而[chrome 和 IE 使用的是操作系统的 CA 列表](http://support2.microsoft.com/kb/931125)。

**4.3 数据完整性**

这部分内容比较好理解，跟平时的 md5 签名类似，只不过安全要求要高很多。openssl 现在使用的完整性校验算法有两种：MD5 或者 SHA。由于 MD5 在实际应用中存在冲突的可能性比较大，所以尽量别采用 MD5 来验证内容一致性。SHA 也不能使用 SHA0 和 SHA1，中国山东大学的王小云教授在 2005 年就宣布[破解了 SHA-1 完整版算法](http://en.wikipedia.org/wiki/Wang_Xiaoyun)。

微软和 google 都已经宣布 16 年及 17 年之后不再支持 sha1 签名证书。

**5 HTTPS 使用成本**

HTTPS 目前唯一的问题就是它还没有得到大规模应用，受到的关注和研究都比较少。至于使用成本和额外开销，完全不用太过担心。

一般来讲，使用 HTTPS 前大家可能会非常关注如下问题：

1. 证书费用以及更新维护。大家觉得申请证书很麻烦，证书也很贵，可是证书其实一点都不贵，便宜的一年几十块钱，最多也就几百。而且现在也有了免费的证书机构，比如著名的 mozilla 发起的免费证书项目：let’s encrypt（https://letsencrypt.org/）就支持免费证书安装和自动更新。这个项目将于今年中旬投入正式使用。

数字证书的费用其实也不高，对于中小网站可以使用[便宜甚至免费的数字证书服务](http://www.startssl.com/)（可能存在安全隐患），像著名的 verisign 公司的证书一般也就几千到几万块一年不等。当然如果公司对证书的需求比较大，定制性要求高，可以建立自己的 CA 站点，比如 google，能够随意签发 google 相关证书。

1. HTTPS 降低用户访问速度。HTTPS 对速度会有一定程度的降低，但是只要经过合理优化和部署，HTTPS 对速度的影响完全可以接受。在很多场景下，HTTPS 速度完全不逊于 HTTP，如果使用 SPDY，HTTPS 的速度甚至还要比 HTTP 快。

大家现在使用百度 HTTPS 安全搜索，有感觉到慢吗？

1. HTTPS 消耗 CPU 资源，需要增加大量机器。前面介绍过非对称密钥交换，这是消耗 CPU 计算资源的大户，此外，对称加解密，也需要 CPU 的计算。

同样地，只要合理优化，HTTPS 的机器成本也不会明显增加。对于中小网站，完全不需要增加机器也能满足性能需求。

**6 后记**

国外的大型互联网公司很多已经启用了全站 HTTPS，这也是未来互联网的趋势。国内的大型互联网并没有全站部署 HTTPS，只是在一些涉及账户或者交易的子页面 / 子请求上启用了 HTTPS。百度搜索首次全站部署 HTTPS，对国内互联网的全站 HTTPS 进程必将有着巨大的推动作用。

目前互联网上关于 HTTPS 的中文资料比较少，本文就着重介绍了 HTTPS 协议涉及到的重要知识点和平时不太容易理解的盲区，希望能对大家理解 HTTPS 协议有帮助。百度 HTTPS 性能优化涉及到大量内容，从前端页面、后端架构、协议特性、加密算法、流量调度、架构和运维、安全等方面都做了大量工作。本系列的文章将一一进行介绍。