作者 StanOz (/users/ce38a30b09c6) 2016.08.14 01:52*
写了15678字，被32人关注，获得了44个喜欢
(/users/ce38a30b09c6)

iOS 中对 HTTPS 证书链的验证

这篇文章是我一边学习证书验证一边记录的内容，
稍微整理了下，共扯了三部分内容：

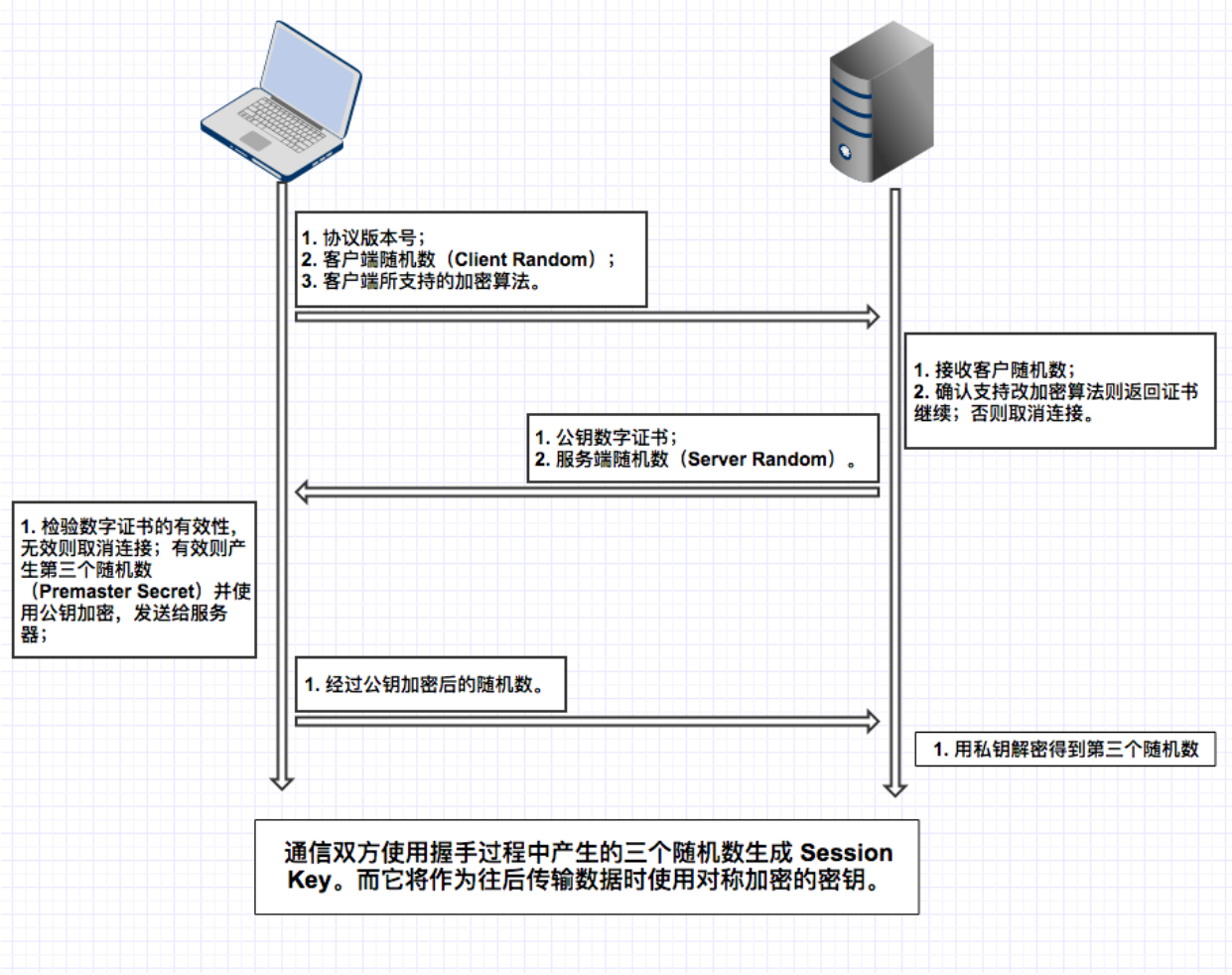
1. HTTPS 简要原理；
2. 数字证书的内容、生成及验证；
3. iOS 上对证书链的验证。

HTTPS 概要

HTTPS 是运行在 TLS/SSL 之上的 HTTP，与普通的 HTTP 相比，在数据传输的安全性上有很大的提升。
要了解它安全性的巧妙之处，需要先简单地了解**对称加密**和**非对称加密**的区别：

- 对称加密只有一个密钥，加密和解密都用这个密钥；
- 非对称加密有公钥和私钥，私钥加密后的内容只有公钥才能解密，公钥加密的内容只有私钥才能解密。

为了提高安全性，我们常用的做法是使用对称加密的手段加密数据。可是只使用对称加密的话，双方通信的开始总会以明文的方式传输密钥。那么从一开始这个密钥就泄露了，谈不上什么安全。所以 TLS/SSL 在握手的阶段，结合非对称加密的手段，保证只有通信双方才知道对称加密的密钥。大概的流程如下：



TSL:SSL_handshake.png

所以，HTTPS 实现传输安全的关键是：在 TLS/SSL 握手阶段保证仅有通信双方得到 Session Key！

数字证书的内容

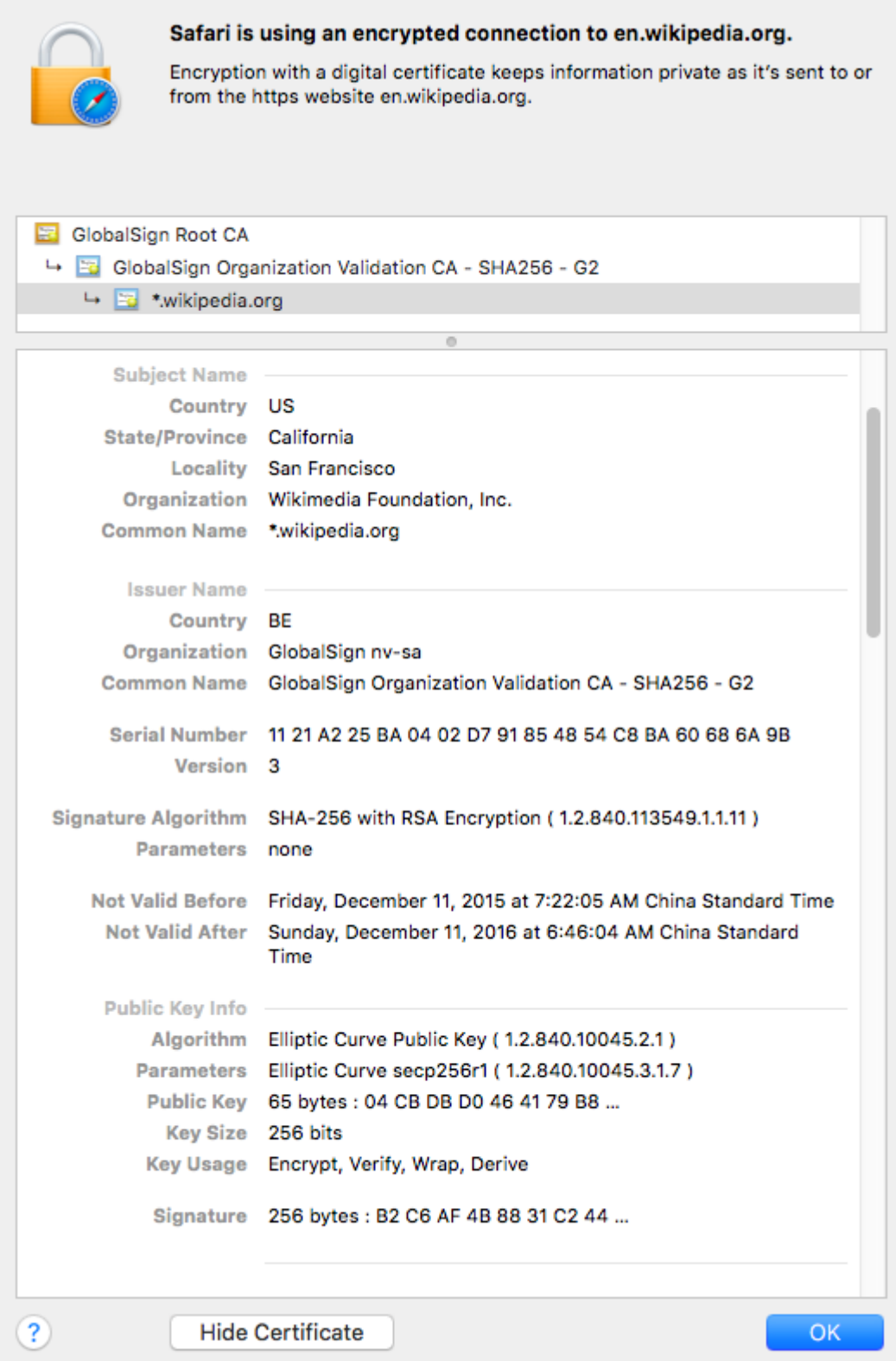
X.509 应该是比较流行的 SSL 数字证书标准，包含（但不限于）以下的字段：

字段	值说明
对象名称（Subject Name）	用于识别该数字证书的信息
共有名称（Common Name）	对于客户证书，通常是相应的域名
证书颁发者（Issuer Name）	发布并签署该证书的实体的信息
签名算法（Signature	签名所使用的算法

Algorithm）	
序列号（Serial Number）	数字证书机构（Certificate Authority， CA）给证书的唯一整数，一个数字证书一个序列号
生效期（Not Valid Before）	(｀・ω・´)
失效期（Not Valid After）	(╯°口°)╯(┻━┻
公钥（Public Key）	可公开的密钥
签名（Signature）	通过签名算法计算证书内容后得到的数据，用于验证证书是否被篡改

除了上述所列的字段，还有很多拓展字段，在此不一一详述。

下图为 Wikipedia 的公钥证书：



wikipedia_cer.png

数字证书的生成及验证

数字证书的生成是分层级的，下一级的证书需要其上一级证书的**私钥**签名。

所以后者是前者的证书颁发者，也就是说上一级证书的 Subject Name 是其下一级证书的 Issuer Name。

在得到证书申请者的一些必要信息（对象名称，公钥私钥）之后，证书颁发者通过 SHA-256 哈希得到证书内容的摘要，再用自己的私钥给这份摘要加密，得到数字签名。综合已有的信息，生成分别包含公钥和私钥的两个证书。

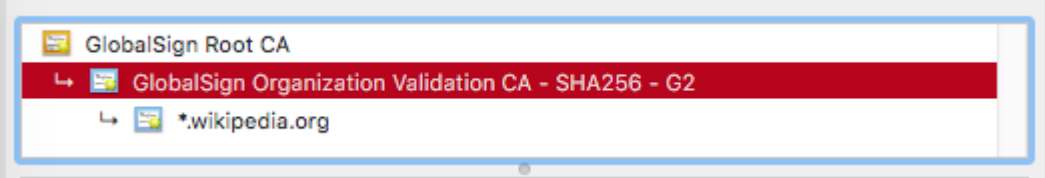
扯到这里，就有几个问题：

问：如果说发布一个数字证书必须要有上一级证书的私钥加密，那么最顶端的证书——根证书怎么来的?

> 根证书是自签名的，即用自己的私钥签名，不需要其他证书的私钥来生成签名。

问：怎么验证证书是有没被篡改?

> 当客户端走 HTTPS 访问站点时，服务器会返回整个证书链。以下图的证书链为例：



chain_hierarchy.png

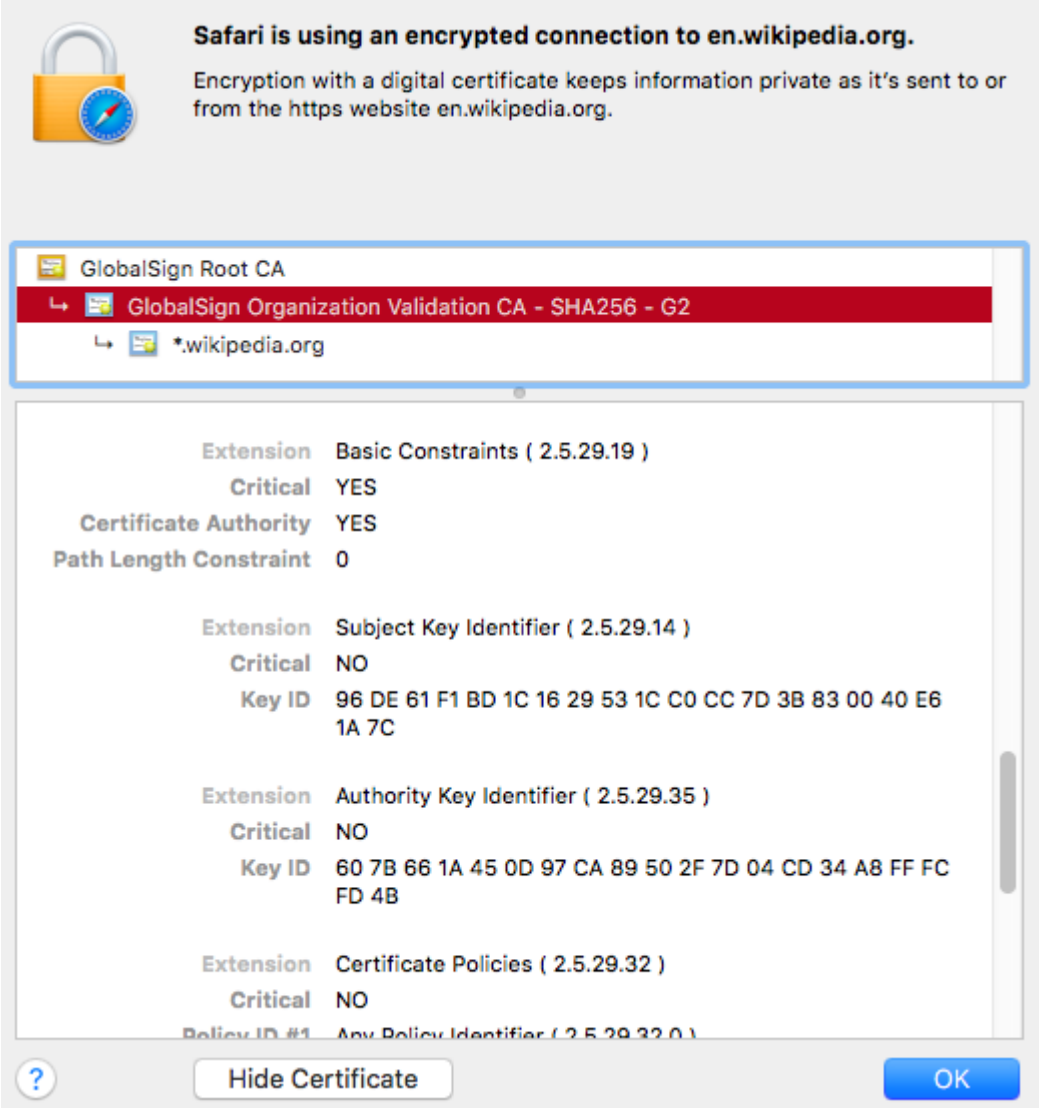
> 要验证 `*.wikipedia.org` 这个证书有没被篡改，就要用到 `GlobalSign Organization Validation CA - SHA256 - G2` 提供的公钥解密前者的签名得到摘要 Digest1，我们的客户端也计算前者证书的内容得到摘要 Digest2。对比这两个摘要就能知道前者是否被篡改。后者同理，使用 `GlobalSign Root CA` 提供的公钥验证。当验证到**到受信任的根证书**时，就能确定 `*.wikipedia.org` 这个证书是可信的。

问：为什么上面那个根证书 `GlobalSign Root CA` 是**受信任的**?

> 数字证书认证机构（Certificate Authority, CA）签署和管理的 **CA 根证书**，会被纳入到你的浏览器和操作系统的可信证书列表中，并由这个列表判断根证书是否可信。所以不要随便导入奇奇怪怪的根证书到你的操作系统中。

问：生成的数字证书（如 `*.wikipedia.org` ）都可用来签署新的证书吗?

> 不一定。如下图，拓展字段里面有个叫 Basic Constraints 的数据结构，里面有个字段叫路径长度约束（Path Length Constraint），表明了该证书能继续签署 CA 子证书的深度，这里为0，说明这个 `GlobalSign Organization Validation CA – SHA256 – G2` 只能签署客户端证书，而客户端证书不能用于签署新的证书，CA 子证书才能这么做。



path_length_constraint.png

iOS 上对证书链的验证

在 Overriding TLS Chain Validation Correctly (https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/NetworkingTopics/Articles/OverridingSSLChainValidationCorrectly.html) 中提到：

> When a TLS certificate is verified, the operating system verifies its chain of trust. If that chain of trust contains only valid certificates and ends at a known (trusted) anchor certificate, then the certificate is considered valid.

所以在 iOS 中，证书是否有效的标准是：

信任链中如果只含有有效证书并且以可信锚点（trusted anchor）结尾，那么这个证书就被认为是有效的。

其中可信锚点指的是系统隐式信任的证书，通常是包括在系统中的 CA 根证书。不过你也可以在验证证书链时，设置自定义的证书作为可信的锚点。

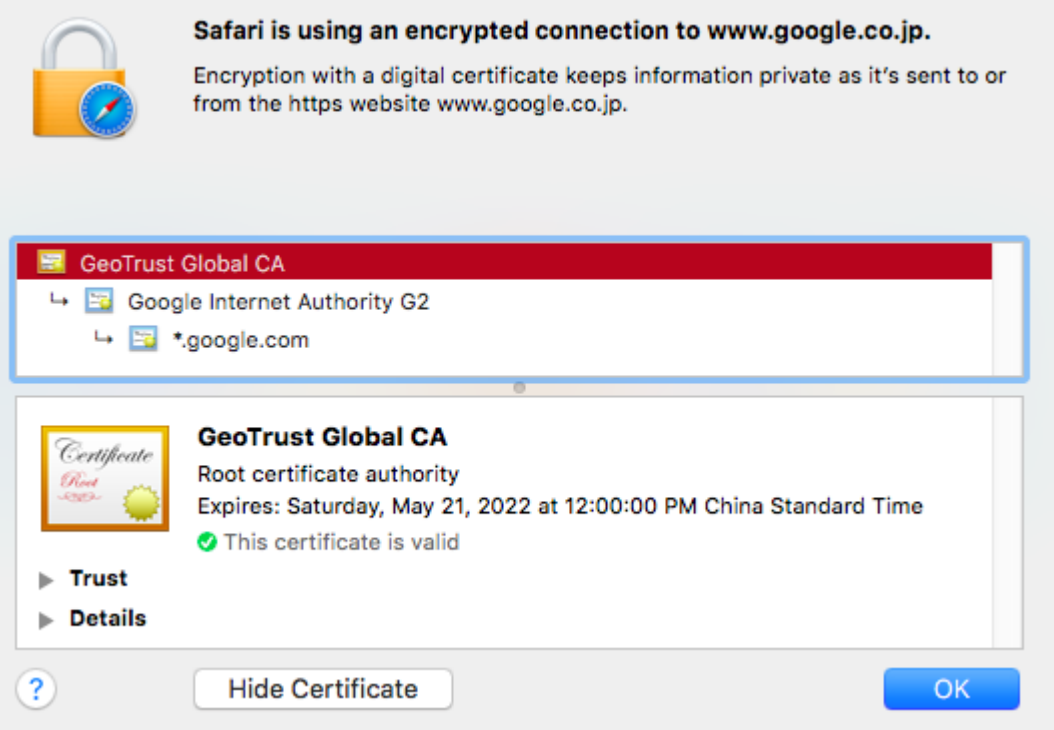
NSURLSession 实现 HTTPS

具体到使用 NSURLSession 走 HTTPS 访问网站，-`URLSession:didReceiveChallenge:completionHandler:` 回调中会收到一个 challenge，也就是质询，需要你提供认证信息才能完成连接。这时候可以通过 `challenge.protectionSpace.authenticationMethod` 取得保护空间要求我们认证的方式，如果这个值是 `NSURLAuthenticationMethodServerTrust` 的话，我们就可以插手 TLS 握手中“验证数字证书有效性”这一步。

默认的实现

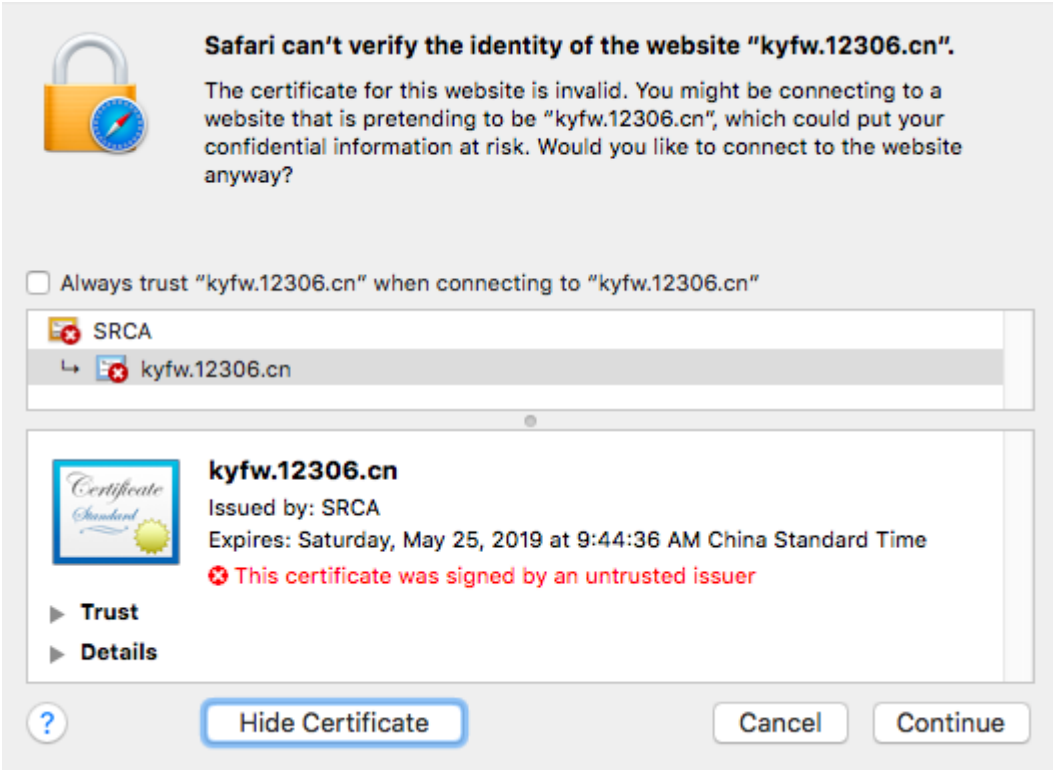
系统的默认实现（也即代理不实现这个方法）是验证这个信任链，结果是有效的话则根据 serverTrust 创建 credential 用于同服务端确立 SSL 连接。否则会得到 “The certificate for this server is invalid...” 这样的错误而无法访问。

比如在访问 https://www.google.com (https://www.google.com) 的时候咧，我们不实现这个方法也能访问成功的。系统对 Google 服务器返回来的证书链，从叶节点证书往根证书层层验证（有效期、签名等等），遇到根证书时，发现作为可信锚点的它存在与可信证书列表中，那么验证就通过，允许与服务端建立连接。



google.png

而当我们访问 https://www.12306.cn (https://www.12306.cn) 时，就会出现 "The certificate for this server is invalid. You might be connecting to a server that is pretending to be “www.12306.cn” which could put your confidential information at risk." 的错误。原因就是系统在验证到根证书时，发现它是自签名、不可信的。



12306.png

自定义实现

如果我们要实现这个代理方法的话，需要提供 NSURLSessionAuthChallengeDisposition（处置方式）和 NSURLCredential（资格认证）这两个参数给 completionHandler 这个 block：

```
-(void)URLSession:(NSURLSession *)session
        didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge
        completionHandler:(void (^)(NSURLSessionAuthChallengeDisposition,
                    NSURLCredential * _Nullable))completionHandler {

    // 如果使用默认的处置方式，那么 credential 就会被忽略
    NSURLSessionAuthChallengeDisposition disposition = NSURLSessionAuthChallengePerf
    NSURLCredential *credential = nil;

    if ([challenge.protectionSpace.authenticationMethod
            isEqualToString:
            NSURLAuthenticationMethodServerTrust]) {

        /* 调用自定义的验证过程 */
        if ([self myCustomValidation:challenge]) {
            credential = [NSURLCredential credentialForTrust:challenge.protectionSpa
            if (credential) {
                disposition = NSURLSessionAuthChallengeUseCredential;
            }
        } else {
            /* 无效的话，取消 */
            disposition = NSURLSessionAuthChallengeCancelAuthenticationChallenge
        }
    }
    if (completionHandler) {
        completionHandler(disposition, credential);
    }
}
```

在 `[self myCustomValidation:challenge]` 调用自定义验证过程，结果是有效的话才创建 credential 确立连接。

自定义的验证过程，需要先拿出一个 SecTrustRef 对象，它是一种执行信任链验证的抽象实体，包含着**验证策略（SecPolicyRef）**以及一系列**受信任的锚点证书**，而我们能做的也是修改这两样东西而已。

```
SecTrustRef trust = challenge.protectionSpace.serverTrust;
```

拿到 trust 对象之后，可以用下面这个函数对它进行验证。

```
static BOOL serverTrustIsVaild(SecTrustRef trust) {
    BOOL allowConnection = NO;

// 假设验证结果是无效的
SecTrustResultType trustResult = kSecTrustResultInvalid;

// 函数的内部递归地从叶节点证书到根证书的验证
OSStatus statue = SecTrustEvaluate(trust, &trustResult);

    if (statue == noErr) {
    // kSecTrustResultUnspecified: 系统隐式地信任这个证书
    // kSecTrustResultProceed: 用户加入自己的信任锚点，显式地告诉系统这个证书是值得信任的

    allowConnection = (trustResult == kSecTrustResultProceed
                                || trustResult == kSecTrustResultUnspecified);
    }
    return allowConnection;
}
```

这个函数什么时候调用完全取决于你的需求，如果你不想对验证策略做修改而直接调用的话，那你居然还看到这里！？ (╯‵□′)╯ ┻━┻

域名验证

可以通过以下的代码获得当前的验证策略：

```
CFArrayRef policiesRef;
SecTrustCopyPolicies(trust, &policiesRef);
```

打印 policiesRef 后，你会发现默认的验证策略就包含了域名验证，即“服务器证书上的域名和请求域名是否匹配”。如果你的一个证书需要用来连接不同域名的主机，或者你直接用 IP 地址去连接，那么你可以重设验证策略以忽略域名验证:

```
NSMutableArray *policies = [NSMutableArray array];

// BasicX509 不验证域名是否相同
SecPolicyRef policy = SecPolicyCreateBasicX509();
[policies addObject:(__bridge_transfer id)policy];
SecTrustSetPolicies(trust, (__bridge CFArrayRef)policies);
```

然后再调用 `serverTrustIsVaild()` 验证。

但是如果不验证域名的话，安全性就会大打折扣。拿浏览器举🌰：

试想你要传输报文到 https://www.real-website.com (https://www.real-website.com)，然而由于域名劫持，把你带到了 https://www.real-website.cn (https://www.real-website.cn) 这个🎧网站，大概有以下两种结果：

1. 这个伪造网站的证书是非 CA 颁布的伪造证书的话，那么浏览器会提醒你这个证书不可信；
2. 这个伪造网站也使用了 CA 颁布的证书，由于我们不做域名验证，你的浏览器不会有任何的警告。

> 你可能会问：公钥证书是每个人都能得到的，钓鱼网站能不能返回真正的公钥证书给我们呢?

我觉得是可以的，然而这并没有什么卵用。没有私钥的钓鱼服务器无法获得第三个随机数，无法生成 Session Key，也就不能对我们传给它的数据进行解密了。

自签名的证书链验证

在 App 中想要防止上面提到的**中间人公鸡攻击**，比较好的做法是将公钥证书打包进 App 中，然后在收到服务端证书链的时候，能够有效地验证服务端是否可信，这也是验证自签名的证书链所必须做的。

假设你的服务器返回：**[你的自签名的根证书]** -- **[你的二级证书]** -- **[你的客户端证书]**，系统是不信任这个三个证书的。
所以你在验证的时候需要将这三个的其中一个设置为锚点证书，当然，多个也行。

比如将 **[你的二级证书]** 作为锚点后， `SecTrustEvaluate()` 函数只要验证到 **[你的客户端证书]** 确实是由 **[你的二级证书]** 签署的，那么验证结果为 `kSecTrustResultUnspecified` ，表明了 **[你的客户端证书]** 是可信的。下面是设置锚点证书的做法：

```
NSMutableArray *certificates = [NSMutableArray array];

NSDate *cerData = /* 在 App Bundle 中你用来做锚点的证书数据，证书是 CER 编码的，常见扩展名有:

SecCertificateRef cerRef = SecCertificateCreateWithData(NULL, (__bridge CFDataRef)ce

[certificates addObject:(__bridge_transfer id)cerRef];

// 设置锚点证书。
SecTrustSetAnchorCertificates(trust, (__bridge CFArrayRef)certificates);
```

只调用 `SecTrustSetAnchorCertificates ()` 这个函数的话，那么就**只有**作为参数被传入的证书作为锚点证书，连系统本身信任的 CA 证书不能作为锚点验证证书链。要想恢复系统中 CA 证书作为锚点的功能，还要再调用下面这个函数：

```
// true 代表仅被传入的证书作为锚点，false 允许系统 CA 证书也作为锚点
SecTrustSetAnchorCertificatesOnly(trust, false);
```

这样，再调用 `serverTrustIsVaild()` 验证证书有效性就能成功了。

CA 证书链的验证

上面说的是没经过 CA 认证的自签证书的验证，而 CA 的证书链的验证方式也是一样，不同点在不可信锚点的证书类型不一样而已：前者的锚点是自签的需要被打包进 App 用于验证，后者的锚点可能本来就存在系统之中了。不过我脑补了这么的一个坑：

假如我们使用的是 CA 根证书签署的数字证书，而且只用这个 CA 根证书作为锚点，在不验证域名的情况下，是不是就会在握手阶段信任被同一个 CA 根证书签名的伪造证书呢?

参考阅读

iOS安全系列之一：HTTPS (http://oncenote.com/2014/10/21/Security-1-HTTPS/)

iOS 安全系列之二： HTTPS 进阶 (http://oncenote.com/2015/09/16/Security-2-HTTPS2/#verify_safely)

Overriding TLS Chain Validation Correctly (https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/NetworkingTopics/Articles/OverridingSSLChainValidationCorrectly.html)

HTTPS Server Trust Evaluation (https://developer.apple.com/library/ios/technotes/tn2232/_index.html#//apple_ref/doc/uid/DTS40012884-CH1-SECGLOSSARY)

上文有什么我理解得不正确、或表达不准确的地方，烦请指教。

如果你是程序员，或者有一颗喜欢写程序的心，喜欢分享技术干货、项目经验、程序员日常囧事等等，欢迎投稿《程序员》专题。 专题主编：小...

(/collection/NEt52a)

2478篇文章 (/collection/NEt52a) · 185851人关注

添加关注 (/sign_in)

iOS Developer (/collection/3233d1a249ca)

分享 iOS 开发的知识，解决大家遇到的问题，讨论iOS开发的前沿，欢迎大家投稿

(/collection/3233d1a249ca)

13738篇文章 (/collection/3233d1a249ca) · 26995人关注

添加关注 (/sign_in)

iOS 大神班 (/collection/5aac963ca52d)

收集进阶知识、编程干货，每天会更新四篇。对于接收的文章，伯乐在线（@iOS大[illegible]）小编会联系您进行转载推广。如果您的投稿（私信投稿）...

(/collection/5aac963ca52d)

863篇文章 (/collection/5aac963ca52d) · 2673人关注

添加关注 (/sign_in)