关于HTTPS的那些事

https://mp.weixin.qq.com/s?__biz=MzA5MzI3NjE2MA==&mid=2650242840&idx=1&sn=8c0baf32761c3caca218a5b33b07a1c1&chksm=88638e77bf140761aa0e63a4be3429a017317e700743ccd8e498ef959bb71265bd95a7500398&scene=38#wechat_redirect

关于HTTPS的那些事

原创： Smilyyy 郭霖 4月16日

今日科技快讯

近日，腾讯微信团队在2018中国“互联网+”数字经济峰会透露，微信小程序将在商业化方面继续挖掘，包括正在内测的小程序广告组件下一步将继续开放，未来更多小程序开发者将变身“流量主”，享有广告收益。此外，小程序还将提供更多促成交易的能力，包括电商工具等，支持电商和零售商，实现交易闭环。

作者简介

周一早上好，春暖花开精神爽，新的一周继续努力！

本篇来自 **Smilyyy** 的投稿，分享了他对 Retrofit中https 的理解 ，一起来看看！希望大家喜欢。

Smilyyy 的博客地址：

> https://blog.csdn.net/qq_20521573

前言

由于前不久苹果公司已经强制IOS应用必须使用HTTPS协议开发，虽然Google没有强制开发者使用HTTPS，但相信不久的将来Android也会跟随IOS全面转向HTTPS。因此，HTTPS的学习也是相当重要。本篇文章涉及到的代码不多，主要内容是对HTTPS协议的讲解，最后将结合Retrofit实现HTTPS的单双向认证。

HTTPS概述

什么是HTTPS？

我们看维基百科给HTTPS的定义：

> HTTPS（Hypertext Transfer Protocol Secure）是一种通过计算机网络进行安全通信的传输协议。HTTPS经由HTTP进行通信，但利用TLS来加密数据包。HTTPS开发的主要目的，是提供对网站服务器的身份认证，保护交换数据的隐私与完整性。

原来HTTPS就是在HTTP协议的基础上加入了TLS协议。目的是保证我们的数据在网络上传输的安全性。

> TLS是传输层加密协议，前身是SSL协议。由网景公司于1995年发布。后改名为TLS。常用的 TLS 协议版本有：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。TLS1.0 也存在部分安全漏洞，比如 RC4 和 BEAST 攻击。

由于HTTP协议采用明文传输，我们可以通过抓包很轻松的获取到HTTP所传输的数据。因此，采用HTTP协议是不安全的。这才催生了HTTPS的诞生。HTTPS相对HTTP提供了更安全的数据传输保障。主要体现在三个方面：

1. 内容加密。客户端到服务器的内容都是以加密形式传输，中间者无法直接查看明文内容。
2. 身份认证。通过校验保证客户端访问的是自己的服务器。
3. 数据完整性。防止内容被第三方冒充或者篡改。

HTTPS实现原理

在学习HTTPS原理之前我们先了解一下两种加密方式： 对称加密和非对称加密。 对称加密 即加密和解密使用同一个密钥，虽然对称加密破解难度很大，但由于对称加密需要在网络上传输密钥和密文，一旦被黑客截取很容就能被破解，因此对称加密并不是一个较好的选择。
非对称加密 即加密和解密使用不同的密钥，分别称为公钥和私钥。我们可以用公钥对数据进行加密，但必须要用私钥才能解密。在网络上只需要传送公钥，私钥保存在服务端用于解密公钥加密后的密文。但是非对称加密消耗的CPU资源非常大，效率很低，严重影响HTTPS的性能和速度。因此非对称加密也不是HTTPS的理想选择。
那么HTTPS采用了怎样的加密方式呢？其实为了提高安全性和效率HTTPS结合了对称和非对称两种加密方式。即客户端使用对称加密生成密钥（key）对传输数据进行加密，然后使用非对称加密的公钥再对key进行加密。因此网络上传输的数据是被key加密的密文和用公钥加密后的密文key，因此即使被黑客截取，由于没有私钥，无法获取到明文key，便无法获取到明文数据。所以HTTPS的加密方式是安全的。
接下来我们以TLS1.2为例来认识HTTPS的握手过程。

- 客户端发送 client_hello，包含一个随机数 random1。
- 服务端回复 server_hello，包含一个随机数 random2，携带了证书公钥 P。
- 客户端接收到 random2 之后就能够生成 premaster_secrect （对称加密的密钥）以及 master_secrect（用 premaster_secret加密后的数据）。
- 客户端使用证书公钥 P 将 premaster_secrect 加密后发送给服务器 (用公钥P对premaster_secret加密)。
- 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。

HTTPS的握手过程如下图：

数字证书

我们上面提到了HTTPS的工作原理，通过对称加密和非对称加密实现数据的安全传输。我们也知道非对称加密过程需要用到公钥进行加密。那么公钥从何而来？其实公钥就被包含在数字证书中。数字证书通常来说是由受信任的数字证书颁发机构CA，在验证服务器身份后颁发，证书中包含了一个密钥对（公钥和私钥）和所有者识别信息。数字证书被放到服务端，具有服务器身份验证和数据传输加密功能。
除了CA机构颁发的证书之外，还有非CA机构颁发的证书和自签名证书。

- 非CA机构即是不受信任的机构颁发的证书，理所当然这样的证书是不受信任的。
- 自签名证书，就是自己给自己颁发的证书。当然自签名证书也是不受信任的。

例如大(chou)名(ming)鼎(zhao)鼎(zhu)的12306网站使用的就是非CA机构颁发的证书（最近发现12306购票页面已经改为CA证书了），12306的证书是由SRCA颁发，SRCA中文名叫中铁数字证书认证中心，简称中铁CA。这是个铁道部自己搞的机构，相当于是自己给自己颁发证书。因此我们访问12306时通常会看到如下情景：
说了这么多，我们来总结一下数字证书的两个作用：

- 分发公钥。每个数字证书都包含了注册者生成的公钥。在 TLS握手时会通过 certificate 消息传输给客户端。
- 身份授权。确保客户端访问的网站是经过 CA 验证的可信任的网站。（在自签名证书的情况下可以验证是否是我们自己的服务器）

最后我们从别处搬来一个中间人攻击的例子，来认识证书是如何保证我们的数据安全的。
对于一个正常的网络请求，其流程通常如下：
但是，如果有黑客在通信过程中拦截了这个请求。试想在客户端和服务端中间有一个中间人，两者之间的传输对中间人来说是透明的，那么中间人完全可以获取两端之间的任何数据并加以修改，然后转发给两端。其流程如下图：
此时恶意服务端完全可以发起双向攻击：对上可以欺骗服务端，对下可以欺骗客户端，更严重的是客户端段和服务端完全感知不到已经被攻击了。这就是所谓的中间人攻击。

中间人攻击（MITM攻击）是指，黑客拦截并篡改网络中的通信数据。又分为被动MITM和主动MITM，被动MITM只窃取通信数据而不修改，而主动MITM不但能窃取数据，还会篡改通信数据。最常见的中间人攻击常常发生在公共wifi或者公共路由上。

现在可以看看使用证书是怎么样提高安全性，避免中间人攻击的，用一张简单的流程图来说明：

HTTPS单项认证

所谓单项认证只要服务端配置证书，客户端在请求服务端时验证服务器的证书即可。我们上述讲到的内容其实都是说的HTTPS单项认证。通常来说对于安全性要求不高的网站单项认证就可以满足我们的需求了。因此我们访问的HTTPS网站大部分都是单项认证。

关于HTTPS的使用存在的误区

由于我们对安全性的认识不够重视，通常对于HTTPS存在一些误区，这些误区可能直接给我们带来一些安全隐患。

误区（1）：对于CA机构颁发的证书客户端无须内置

上面提到访问HTTPS服务器是需要在客户端配置服务器证书的。有些小伙伴可能就纳闷了，说我们用的就是HTTPS但是并没有在客户端配置证书呢？比如请求百度的网站https://www.baidu.com/，和请求HTTP服务器没什么区别。其实这是因为在Android系统中已经内置了所有CA机构的根证书，也就是只要是CA机构颁发的证书，Android是直接信任的。对于此种情况，虽然可以正常访问到服务器，但是仍然存在安全隐患。假如黑客自家搭建了一个服务器并申请到了CA证书，由于我们客户端没有内置服务器证书，默认信任所有CA证书（客户端可以访问所有持有由CA机构颁发的证书的服务器），那么黑客仍然可以发起中间人攻击劫持我们的请求到黑客的服务器，实际上就成了我们的客户端和黑客的服务器建立起了连接。

误区（2）：对于非CA机构颁发的证书和自签名证书，可以忽略证书校验

另外一种情况，如果我们服务器的证书是非认证机构颁发的 (例如12306)或者自签名证书，那么我们是无法直接访问到服务器的，直接访问通常会抛出如下异常：

网上很多解决SSLHandshakeException异常的方案是自定义TrustManager忽略证书校验。代码如下：

```
javax.net.ssl.SSLHandshakeException:
    java.security.cert.CertPathValidatorException:
        Trust anchor for certification path not found.
```

网上很多解决SSLHandshakeException异常的方案是自定义TrustManager忽略证书校验。代码如下：

```
public static SSLSocketFactory getSSLSocketFactory() throws Exception {
        //创建一个不验证证书链的证书信任管理器。
        final TrustManager[] trustAllCerts = new TrustManager[]{new X509TrustManager() {
            @Override
            public void checkClientTrusted(
                    java.security.cert.X509Certificate[] chain,
                    String authType) throws CertificateException {
            }

            @Override
            public void checkServerTrusted(
                    java.security.cert.X509Certificate[] chain,
                    String authType) throws CertificateException {
            }

            @Override
            public java.security.cert.X509Certificate[] getAcceptedIssuers() {
                return new java.security.cert.X509Certificate[0];
            }
        }};

        // Install the all-trusting trust manager
        final SSLContext sslContext = SSLContext.getInstance("TLS");
        sslContext.init(null, trustAllCerts,
                new java.security.SecureRandom());
        // Create an ssl socket factory with our all-trusting manager
        return sslContext
                .getSocketFactory();
    }
```

```
    //使用自定义SSLSocketFactory
    private void onHttps(OkHttpClient.Builder builder) {
        try {
            builder.sslSocketFactory(getSSLSocketFactory()).hostnameVerifier(org.apache.http.conn.ssl.SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
```

对于这样的处理方式虽然解决了SSLHandshakeException异常，但是却存在更大的安全隐患。因为此种做法直接使我们的客户端信任了所有证书（包括CA机构颁发的证书和非CA机构颁发的证书以及自签名证书），因此，这样配置将比第一种情况危害更大。

Retrofit绑定证书实现HTTPS单项认证

对于上述两种情况中存在的安全隐患，我们应该如何应对？最简单的解决方案就是在客户端内置服务器的证书，我们在校验服务端证书的时候只比对和App内置的证书是否完全相同，如果不同则断开连接。那么此时再遭遇中间人攻击劫持我们的请求时由于黑客服务器没有相应的证书，此时HTTPS请求校验不通过，则无法与黑客的服务器建立起连接。

那么接下来我们就结合Retrofit以访问12306为例来实现HTTPS的单项认证。

首先从12306网站下载签名证书，并放置到我们项目资源目录raw下。然后根据证书构造SSLSocketFactory，代码如下：

```
    /**
     * 单项认证
     */
    public static SSLSocketFactory getSSLSocketFactoryForOneWay(InputStream... certificates) {
        try {
            CertificateFactory certificateFactory = CertificateFactory.getInstance(CLIENT_TRUST_MANAGER, CLIENT_TRUST_PROVIDER);
            KeyStore keyStore = KeyStore.getInstance(CLIENT_TRUST_KEYSTORE);
            keyStore.load(null);
            int index = 0;
            for (InputStream certificate : certificates) {
                String certificateAlias = Integer.toString(index++);
                keyStore.setCertificateEntry(certificateAlias, certificateFactory.generateCertificate(certificate));
                try {
                    if (certificate != null)
                        certificate.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            SSLContext sslContext = SSLContext.getInstance(CLIENT_AGREEMENT);

            TrustManagerFactory trustManagerFactory =
                    TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());

            trustManagerFactory.init(keyStore);
            sslContext.init(null, trustManagerFactory.getTrustManagers(), new SecureRandom());
            return sslContext.getSocketFactory();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
```

接下来为OKHttpClient设置SslSocketFactory以及hostnameVerifier，代码如下：

```
InputStream certificate12306 = Utils.getContext().getResources().openRawResource(R.raw.srca);
        OkHttpClient okHttpClient = new OkHttpClient.Builder()
                .readTimeout(Constants.DEFAULT_TIMEOUT, TimeUnit.MILLISECONDS)
                .connectTimeout(Constants.DEFAULT_TIMEOUT, TimeUnit.MILLISECONDS)
```

```
                .addInterceptor(interceptor)
                .addInterceptor(new HttpHeaderInterceptor())
                .addNetworkInterceptor(new HttpCacheInterceptor())
                .sslSocketFactory(SslContextFactory.getSSLSocketFactoryForOneWay(certificate12306))
                .hostnameVerifier(new SafeHostnameVerifier())
                .cache(cache)
                .build();
```

上述代码中hostnameVerifier是对服务器的校验，SafeHostnameVerifier代码如下：

```
private class SafeHostnameVerifier implements HostnameVerifier {
        @Override
        public boolean verify(String hostname, SSLSession session) {
            if (Constants.IP.equals(hostname)) {//校验hostname是否正确，如果正确则建立连接
                return true;
            }
            return false;
        }
    }
```

verify方法中对比了请求的IP和服务器的IP是否一致，一致则返回true表示校验通过，否则返回false，检验不通过，断开连接。对于网上有些处理是直接返回true，即不对请求的服务器IP做校验，我们不推荐这样使用。而且现在谷歌应用商店已经对此种做法做了限制，禁止在verify方法中直接返回true的App上线。

HTTPS双项认证

对于HTTPS双向认证，用到的情况不多。但是对于像金融行业等对安全性要求较高的企业，通常都会使用双向认证。所谓双向认证就是客户端校验服务器证书，同时服务器也需要校验客户端的证书。因此，双向认证就另需一张证书放到客户端待服务端去验证。

> 单项认证保证了我们自己的客户端只能访问我们自己的服务器，但并不能保证我们自己的服务器只能被我们自己的客户端访问（第三方客户端忽略证书校验即可）。那么双向认证则保证了我们的客户端只能访问我们自己的服务器，同时我们的服务器也只能被我们自己的客户端访问。因此双向认证可以说相比单项认证安全性足足提高一个等级。

双向认证流程

接下来我们来了解下双向认证的流程，以加深对双向认证的理解：

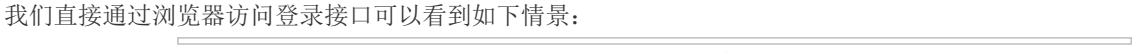
- 客户端发送一个连接请求给服务器。
- 服务器将自己的证书，以及同证书相关的信息发送给客户端。
- 客户端检查服务器送过来的证书是否和App内置证书相同。如果是，就继续执行协议；如果不是则终止此次请求。
- 接着客户端比较证书里的消息，例如域名和公钥，与服务器刚刚发送的相关消息是否一致，如果是一致的，客户端认可这个服务器的合法身份。
- 服务器要求客户发送客户自己的证书。收到后，服务器验证客户端的证书，如果没有通过验证，拒绝连接；如果通过验证，服务器获得用户的公钥。
- 客户端告诉服务器自己所能够支持的通讯对称密码方案。
- 服务器从客户发送过来的密码方案中，选择一种加密程度最高的密码方案，用客户的公钥加过密后通知客户端。
- 客户端针对这个密码方案，选择一个通话密钥，接着用服务器的公钥加过密后发送给服务器。
- 服务器接收到客户端送过来的消息，用自己的私钥解密，获得通话密钥。
- 服务器通过密钥解密客户端发送的被加密数据，得到明文数据。

Retrofit实现HTTPS双向认证

对于双向认证，我们以华为北向平台登录接口为例来进行学习。地址如下：

> http://developer.huawei.com/ict/cn/doc/site-oceanconnect-northbound_api_reference-zh/index.html/zh-cn_topic_0103199657

我们直接通过浏览器访问登录接口可以看到如下情景：



哈，惊喜不？直接被拒绝了！这就是双向认证，没有证书想访问服务器门都没有。那么对于双向认证我们应该做怎样的配置？我们可以参考华为开源出来的代码，源码中由两个证书文件ca.jks和outgoing.CertwithKey.pkcs12，其中ca.jks是在客

户端配置的证书，outgoing.CertwithKey.pkcs12是在服务端配置的证书。因为我们当前客户端是Android系统，由于Android系统不支持jks格式的证书，因此需要把jks转成Android支持的bks格式。转换方式不再贴出，可自行查阅。

有了证书，接下来看获取SSLSocketFactory的代码：

```
  /**
     * 双向认证
     *
     * @return SSLSocketFactory
     */
    public static SSLSocketFactory getSSLSocketFactoryForTwoWay() {
        try {
            InputStream certificate = Utils.getContext().getResources().openRawResource(R.raw.capk);
            //  CertificateFactory certificateFactory = CertificateFactory.getInstance("X.509", "BC");
            KeyStore keyStore = KeyStore.getInstance(CLIENT_TRUST_KEY);
            keyStore.load(certificate, SELF_CERT_PWD.toCharArray());
            KeyManagerFactory kmf =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
            kmf.init(keyStore, SELF_CERT_PWD.toCharArray());

            try {
                if (certificate != null)
                    certificate.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

            //初始化keystore
            KeyStore clientKeyStore = KeyStore.getInstance(CLIENT_TRUST_KEYSTORE);
            clientKeyStore.load(Utils.getContext().getResources().openRawResource(R.raw.cabks),
TRUST_CA_PWD.toCharArray());

            SSLContext sslContext = SSLContext.getInstance(CLIENT_AGREEMENT);
            TrustManagerFactory trustManagerFactory = TrustManagerFactory.
                    getInstance(TrustManagerFactory.getDefaultAlgorithm());

            trustManagerFactory.init(clientKeyStore);

            KeyManagerFactory keyManagerFactory =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
            keyManagerFactory.init(clientKeyStore, SELF_CERT_PWD.toCharArray());

            sslContext.init(kmf.getKeyManagers(), trustManagerFactory.getTrustManagers(), new
SecureRandom());
            return sslContext.getSocketFactory();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
```

接下来同样需要配置OKHttpClient，代码如下：

```
OkHttpClient okHttpClient = new OkHttpClient.Builder()
                .readTimeout(Constants.DEFAULT_TIMEOUT, TimeUnit.MILLISECONDS)
                .connectTimeout(Constants.DEFAULT_TIMEOUT, TimeUnit.MILLISECONDS)
                .addInterceptor(interceptor)
                .addInterceptor(new HttpHeaderInterceptor())
                .addNetworkInterceptor(new HttpCacheInterceptor())
                .sslSocketFactory(SslContextFactory.getSSLSocketFactoryForTwoWay())
                .hostnameVerifier(new SafeHostnameVerifier())
                .cache(cache)
                .build();
```

这样就完成了HTTPS的配置，接下来就可以愉快的访问HTTPS 双向认证的接口了。由于北向登录接口中需要appId和secret两个参数，因此，登录相关代码就不再贴出。