iOS应用安全杂谈

如何打造一个安全的App？这是每一个移动开发者必须面对的问题。在移动App开发领域，开发工程师对于安全方面的考虑普遍比较欠缺，而由于iOS平台的封闭性，遭遇到的安全问题相比于Android来说要少得多，这就导致了许多iOS开发人员对于安全性方面没有太多的深入，但对于一个合格的软件开发者来说，安全知识是必备知识之一。

对于未越狱的iOS设备来说，由于强大的沙盒和授权机制，以及Apple自己掌控的App Store， 基本上杜绝了恶意软件的入侵（非越狱）。但除系统安全之外，我们还是面临很多的安全问题：网络安全、数据安全等，每一项涉及也非常广，安全是非常大的课题，本人并非专业的安全专家，只是从开发者的角度，分析我们常遇到的各项安全问题，并提出通常的解决方法，与各位同学交流学习。

(图：iOS设备的安全体系结构)

(过渡：我们知道大部分App都是有后端服务支持的，前后端通过网络协议传输数据，其中使用最多的是HTTP协议)

网络安全

使用HTTPS

HTTPS

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比HTTP协议安全。

(图：SSL+HTTP)

HTTPS的主要思想是在不安全的网络上创建一安全信道，并可在使用适当的加密包和服务器证书可被验证且可被信任时，对窃听和中间人攻击提供合理的防护。

(图：图解HTTPS)

1. 客户端发起HTTPS请求
   这个没什么好说的，就是用户在浏览器里输入一个https网址，然后连接到server的

443端口。

2. 服务端的配置
   采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。如果对公钥和私钥不太理解，可以想象成一把钥匙和一个锁头，只是全世界只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

3. 传送证书
   这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。

4. 客户端解析证书
   这部分工作是有客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随机数。然后用证书对该随机数进行加密。就好像上面说的，把随机数用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。

5. 传送加密信息
   这部分传送的是用证书加密后的随机数，目的就是让服务端得到这个随机数，以后客户端和服务端的通信就可以通过这个随机数来进行加密解密了。

6. 服务段解密信息
   服务端用私钥解密后，得到了客户端传过来的随机数(私钥)，然后把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

7. 传输加密后的信息
   这部分信息是服务段用私钥加密后的信息，可以在客户端被还原

8. 客户端解密信息
   客户端用之前生成的私钥解密服务段传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

(过渡：真实的过程比这还要复杂，特别是握手过程，但因为不是这次分享会的重点，这里只是简单介绍一下，想深入了解的同学可以自己会后去了解)

(过渡：从上面可以看到)

HTTPS的信任继承基于预先安装在浏览器(或系统)中的证书颁发机构（如Symantec、Comodo、GoDaddy和GlobalSign等）（意即“我信任证书颁发机构告诉我应该信任的”）。因此，一个到某网站的HTTPS连接可被信任，当且仅当：

- 用户相信他们的浏览器正确实现了HTTPS且安装了正确的证书颁发机构；
- 用户相信证书颁发机构仅信任合法的网站；
- 被访问的网站提供了一个有效的证书，意即，它是由一个被信任的证书颁发机构签发的（大部分浏览器会对无效的证书发出警告）；
- 该证书正确地验证了被访问的网站（如，访问https://example.com时收到了给example.com而不是其它组织的证书）；
- 或者互联网上相关的节点是值得信任的，或者用户相信本协议的加密层（TLS或SSL）不能被窃听者破坏。

查看iOS设备中可用的受信任根证书列表

(图：查看iOS设备中可用的受信任根证书列表)

(过渡：使用了HTTPS就安全了吗？不一定，为什么不一定呢，继续往下看)

中间人攻击

在密码学和计算机安全领域中，中间人攻击（英语：Man-in-the-middle attack，缩写：MITM）是指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话，但事实上整个会话都被攻击者完全控制。在中间人攻击中，攻击者可以拦截通讯双方的通话并插入新的内容。在许多情况下这是很简单的（例如，在一个未加密的Wi-Fi 无线接入点的接受范围内的中间人攻击者，可以将自己作为一个中间人插入这个网络）。

(过渡：即使使用了HTTPS，还是有可能受到中间人攻击，下面使用Charles模拟攻击)

(过渡：Charles Mac电脑上的抓包软件，因为这电脑没有相关环境，所以我们直接看图)

(图：使用Charles模拟中间人攻击攻击)

使用Charles模拟攻击

点击 Charles 的顶部菜单，选择 “Help” -> “SSL Proxying” -> “Install Charles Root Certificate on a Mobile Device or Remote Browser”，然后就可以看到 Charles 弹出的简单的安装教程。如下图所示：

按照我们之前说的教程，在设备上设置好 Charles 为代理后，在手机浏览器中访问地址：http://charlesproxy.com/getssl，即可打开证书安装的界面，安装完证书后，就可以截取手机上的 Https 通讯内容了。不过需要注意，默认情况下 Charles 并不做截取，你还需要在要截取的网络请求上右击，选择 SSL proxy 菜单项。

打包证书校验

为什么伪造证书是可以实现中间人攻击的？答案就在于用户让系统信任了不应该信任的证书。用户设置系统信任的证书，会作为锚点证书(Anchor Certificate)来验证其他证书，当返回的服务器证书是锚点证书或者是基于该证书签发的证书（可以是多个层级）都会被信任。

这就是基于信任链校验方式的最大弱点。我们不能完全相信系统的校验，因为系统的校验依赖的证书的源很可能被污染了。这就需要选取一个节点证书，打包到App中，作为Anchor Certificate来保证证书链的唯一性和可信性。

使用AFNetworking校验证书的关键代码：

```
    //服务器端返回的证书与本地保存的证书中的PublicKey部分进行校验，如果正确，才继续进行，不校验证书有效期
    AFSecurityPolicy *policy = [AFSecurityPolicy policyWithPinningMode:AFSSLPinningModePublicKey];
    //从字节数据常量加载证书数据，从常量加载能防止因bundle的cer文件被人替换而引起的中间人攻击，提高App安全性。
    NSData *cerData = [[NSData alloc] initWithBytes:YMAPICerBytes length:sizeof(YMAPICerBytes)/sizeof(YMAPICerBytes[0])];
    //设置证书数据，不设置pinnedCertificates时默认搜索bundle的cer文件自动设置证书数据
    policy.pinnedCertificates = [NSSet setWithArray:@[cerData]];

    AFHTTPSessionManager *manager = [AFHTTPSessionManager manager];
    //设置HTTPS安全策略
    manager.securityPolicy = policy;
```

(过渡：没有绝对的安全，即使用了HTTPS且校验了证书，也有可能被黑客攻破。假设现在被黑客攻破了，那么就要考虑以下的情况了。首先是请求参数篡改问题，比如提现100元被改成10000元。)

防止参数篡改

使用参数签名，解决请求参数篡改问题。

1. 约定好secretKey和secretValue
2. 按照请求参数名的字母升序排列请求参数，使用URL键值对的格式（即key1=value1&key2=value2...）拼接成字符串stringA。
3. 在stringA最后拼接&secretKey=secretValue得到字符串stringB。
4. 对stringB进行MD5运算，得到sign值。
5. 请求携带上参数sign。

后端用同样的方法生成sign值并与接收到的请求的sign值比较，只有一样才返回正确的

数据，否则返回错误。因为只有知道secretValue和sign生成算法的，才能得到正确的sign值。所以这样就解决了参数篡改问题，即使请求参数被劫持，由于获取不到secretValue（仅作本地加密使用，不参与网络传输）及sign生成算法，无法伪造合法的请求。

防止重放攻击

虽然使用参数签名解决了请求参数被篡改的问题，但是还存在着重复使用请求参数伪造二次请求的问题。下面使用nonce(唯一随机字符串)+timestamp(时间戳)方案解决此问题。

nonce用来标识每个被签名的请求，通过为每个请求提供一个唯一的标识符，服务器能够防止请求被多次使用(记录所有用过的nonce以阻止它们被二次使用)。然而，对服务器来说永久存储所有接收到的nonce的代价是非常大的。因此使用timestamp来优化nonce的存储。

1. App端给每个请求都带上不重复的随机数nonce，和当前时间戳timestamp。
2. 后端检查携带的timestamp是否在10分钟内，如超出时间范围，则拒绝该请求，否则继续。
3. 查询携带的nonce，如在已有的nonce集合中存在，则拒绝该请求，否则继续。
4. 记录该nonce，并删除集合内时间戳大于10分钟的nonce。

数据安全

Sandbox 数据

iOS 8.3之前，不越狱的手机也可以直接用iTools等软件来查看App沙盒（系统App除外），查看里面存储的文件，如sqlite、plist、图片等。而且越狱手机可以查看任意App沙盒和系统目录等。

所以数据尽量不要本地存储或者用后即删，可能要考虑用户体验有些敏感数据一定要存储于本地，那么请将数据进行加密，数据库要添加访问限制。

(图：iTools查看App沙盒和系统目录)

Keychain 数据

Console Log 数据

在正式发布(release)环境下NSLog不要打印代码调试日志，否则Xcode等工具可以查看到打印的日志。

(图：查看到打印的代码调试日志)

在.pch文件中加下面的几行代码，在发布环境下用宏定义替换NSLog(...)为{}就可以解决。

```
//只在Debug模式下执行NSLog
#ifndef __OPTIMIZE__
//#define NSLog(fmt, ...) NSLog((@"%s" fmt), __FUNCTION__, ##__VA_ARGS_
_)
#else
#define NSLog(...) {}
#endif
```

环境安全

(过渡：环境，指App的运行环境，操作系统等)

越狱检测

iOS设备越狱(获取Root权限)后，破坏了系统原有的安全屏障，使得App置于各种风险之中。通过越狱检测，我们可以禁止用户在越狱设备上使用App，或禁用部分敏感功能，降低风险。

(图：支付宝App在越狱设备上禁用了指纹解锁及支付)

1. 检测当前设备是否已越狱。
    - 通过NSFileManager判断是否存在常见越狱文件。（如"/Applications/Cydia.app"）
    - 可能存在hook了NSFileManager方法，此处用底层C stat去检测。
    - 可能存在stat也被hook了，可以看stat是不是出自系统库，有没有被攻击者换掉。
    - 检测链接动态库，检测是否被链接了异常动态库(包含MobileSubstrate)，但动态库相关Api属于私有Api(待确认，因为微信也有用)，调用的话appStore审核会不通过。
    - 检测当前程序运行的环境变量，如果攻击者给MobileSubstrate改名，但是原理都是通过DYLD_INSERT_LIBRARIES注入动态库。
2. 在启动页面接口传输越狱状态给后台，通过jailbroken参数。
3. 若设备已越狱(jailbroken为1)，则后台返回错误及相应错误信息，让App端不能进入主页并给出相应的提醒信息。

之所以要经过后端来控制是否允许在越狱设备上使用App，是因为越狱检测可能存在误

判，让后端来控制更灵活。

包名检测

一般破解(非官方)修改过的App重新打包安装后包名(Bundle Identifier)会改变。通过检测包名，我们可以禁止用户使用破解修改过App，降低风险。

1. 获取当前App包名。
2. 在启动页面接口传输包名给后台，通过bundleId参数。
3. 若包名与后台设定好的官方包名不一致，则后台返回错误及相应错误信息，让App端不能进入主页并给出相应的提醒信息。

```
    //传bundleIdentifier，后台可根据此字段来禁止用户使用修改过的App包。
    params[@"bundleId"]  = [[NSBundle mainBundle] bundleIdentifier];
```

(图：微信App通过包名检测识别用户是否在使用非官方客户端)

代码安全

IDA反汇编

IDA 是一个反汇编工具，对于 Objective-C 代码，它可以常常可以反汇编到可以方便阅读的程度，这对于程序的安全性，也是一个很大的危害。因为通过阅读源码，黑客可以更加方便地分析出应用的通讯协议和数据加密方式。

下图分别示例了一段代码的原始内容，以及通过 IDA 反汇编之后的结果。可以看到，IDA 几乎还原了原本的逻辑，而且可读性也非常高。

原始代码：

```
if ([[VersionAgent sharedInstance] isUpgraded]) {
    UpdateMigrationAgent *agent =
            [[UpdateMigrationAgent alloc] init];
    [FileUtils clearCacheDirectory];
    [[VersionAgent sharedInstance] saveAppVersion];
}
```

反汇编后：

```
v6 = _objc_msgSend(&OBJC_CLASS___VersionAgent,
                   "sharedInstance");
```

```
v7 = objc_retainAutoreleasedReturnValue(v6);
v41 = _objc_msgSend(v7, "isUpgraded");
objc_release(v7);
if ( v41 )
{
    NSLog(CFSTR("app is upgraded"), v41);
    _objc_msgSend(&OBJC_CLASS___FileUtils,
                  "clearCacheDirectory");
    v8 = _objc_msgSend(&OBJC_CLASS___VersionAgent,
                       "sharedInstance");
    v9 = objc_retainAutoreleasedReturnValue(v8);
    _objc_msgSend(v9, "saveAppVersion");
    objc_release(v9);
}
```

反汇编的代码被获得后，由于软件内部逻辑相比汇编代码来说可读性高了很多。黑客可以用来制作软件的注册机，也可以更加方便地破解网络通讯协议，从而制作出机器人（僵尸）帐号。最极端的情况下，黑客可以将反汇编的代码稍加修改，植入木马，然后重新打包发布在一些越狱渠道上，这将对用户产生巨大的危害。

对于 IDA 这类工具，我们的应对措施就比较少了。除了可以用一些宏来简单混淆类名外，我们也可以将关键的逻辑用纯 C 实现。例如微信的 iOS 端的通讯底层，就是用 C 实现的。这样的方式除了能保证通讯协议安全外，也可以在 iOS 和 Android 等多个平台使用同一套底层通讯代码，达到复用的目的。

代码混淆

iOS App通过class-dump工具可以很方便的导出程序头文件，让攻击者了解了程序结构方便逆向。通过IDA反汇编工具，代码常常可以反汇编到可以方便阅读的程度，这对于程序的安全性，也是一个很大的危害。所以出于安全考虑，最好把代码混淆。

代码混淆的方式有几种：

- 添加无用又不影响逻辑的代码片段，迷糊逆向人员
- 对关键的类、方法，命名成与真实意图无关的名称
- 对于第二种，目前有一些自动化工具。

但是目前没有比较成熟可靠的方案，且网络上大量反馈代码混淆后不能通过App Store的审核。

(过渡：最后给大家看一张图)

(图： 没有混淆代码的微信App被逆向并被注入插件)

其它

- 账号保护，非常用设备登录需手机验证码。
- 自定义安全键盘。
- 双向证书校验。

总结