# App安全之网络传输安全

移动端App安全如果按CS结构来划分的话，主要涉及客户端本身数据安全，Client到Server网络传输的安全，客户端本身安全又包括代码安全和数据存储安全。所以当我们谈论App安全问题的时候一般来说在以下三类范畴当中。

- App代码安全，包括代码混淆，加密或者app加壳。
- App数据存储安全，主要指在磁盘做数据持久化的时候所做的加密。
- App网络传输安全，指对数据从客户端传输到Server中间过程的加密，防止网络世界当中其他节点对数据的窃听。

这一篇我们先聊下网络传输的安全。

## 安全相关的基础概念

网络安全相关的概念非常之多，要在广度和深度上都有所造诣很困难。但如果只是站在保障App通信基本安全这个维度上，做到合理使用安全算法，比大部分人所预期的都要简单很多。以下这些基础概念是必备知识：

- 对称加密算法，代表算法AES
- 非对称加密算法，代表算法RSA，ECC
- 电子签名，用于确认消息发送方的身份
- 消息摘要生成算法，MD5，SHA，用于检测消息是否被第三方修改过

## 怎么样算安全?

安全问题说白了是信任问题，谈到信任一定要有一个可被信任的实体。假设某天A收到了一条“Message”，如果这个消息确实是来自B，消息就可以被信任，那么B就这安全问题当中可被信任的实体。

对于A来说只要满足三点就说明收到“Message”是安全的：

- Message有B的电子签名，表明消息确实是来自B。
- Message没有被篡改过。
- Message被某种加密算法加密过，只有A和B知道如何解密。

A和B的交谈如果放到网络世界当中，就是典型的Client－Server模式。和现实世界当中聊天不同的是，A和B所说的任何话都可以轻而易举的被其他人听到，隔墙随处有耳。

# 怎么去保证App网络传输的安全?

为了保证传输数据的安全，需要使用加密算法。在决定什么样的业务场景使用什么样的加密算法之前，先要了解我们的工具箱里有哪些可用工具。加密算法的工具箱里其实就两种工具：“对称”加密算法和“非对称”加密算法。清楚这两个分类是合理设计加密算法使用场景的大前提，两个分类里我们各挑选一个代表算法来研究，“对称”加密挑选AES，“非对称”加密选择RSA。了解AES和RSA之后我们再针对一些特定业务场景设计安全模型。

# 对称加密之AES

要深入了解像AES这种加密算法，对大部分初学的同学来说可能有些费时费力，但对算法掌握可以是个循序渐进的过程。简单来说可以分为以下几个阶段：

**阶段一：** 在AES诞生之前（1975年之前），早起的加密算法十分简单，是一种类似“暗号”的机制。通信的双方通过事先约定一种“加密机制”对明文进行处理，只要“加密机制”不被泄漏，信息就算安全。后来无数的经验表明算法总是会被第三方得知，对算法本身进行保密管理也不方便。

**阶段二：** 到上世纪70年代，公众和政府意识到加密算法的重要性，由美国政府机构NBS牵头组织业界专家设计了DES算法。DES算法本身公开，但算法的安全全依赖于一个密钥。后来DES统治加密算法长达二十多年。不过DES从诞生到逐步退出历史舞台，一直都伴随着阴谋的论调。DES原本是由IBM提出，针对已有算法Lucifer改造而成，但DES制定的过程一直都有美国另一政府机构NSA的影子。

DES的两大特点是短密钥（short key）和结构复杂的s-box。有研究表明，是NSA当时劝说IMB短密钥足够安全，而且NSA也直接参与了s-box的设计。后来就一直有传言说NSA似乎有办法能轻易破解DES加密后的秘文。

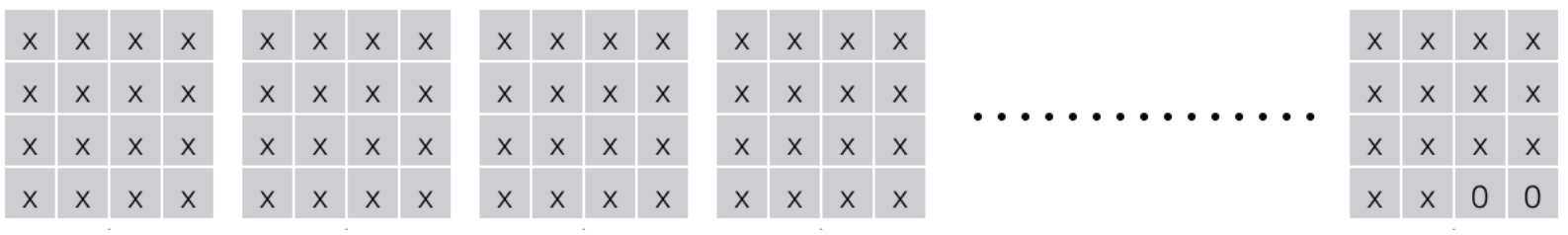
到1990年，剧情出现反转。第三方独立研究者Eli Biham和Adi Shamir发布了破解块加密方式的通用破解方法differential cryptanalysis。出人意料的是按照NSA建议设计的s-box对这种破解方法有更好的免疫性，NSA的参与似乎更加提高了DES的安全性。

到1994年，剧情进一步升华。公开文件披露，早在1974年IBM的研究者就已经发现了differential cryptanalysis这种破解方式，不过这项研究被NSA禁止公开，理由是会威胁到国家安全，对，就是现在美剧里经常提到的national security。不仅如此，NSA还针对DES做了一些改造加强安全性。至此，NSA已经全身都挂满了节操，被业界质疑忍辱负重二十多年一声没吭，比24小时男主角Jack Bauer形象更加高大光辉。

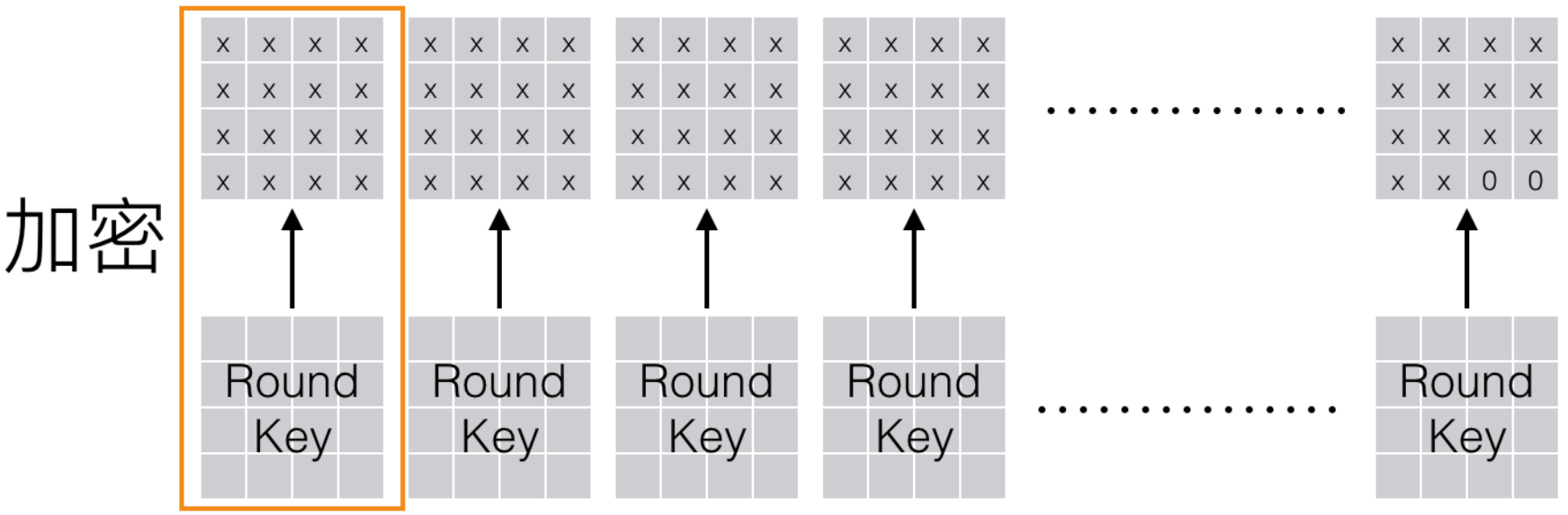
**阶段三：** 后来陆陆续续有一些针对DES破解的攻防战，DES最终演化成Triple DES的形态，不过在性能上已经出现明显问题，催生了今天对称加密的王者算法AES。AES在安全性，实现难度，运算性能上都有更好的表现。在全面了解AES之前，首先明确下好的加密算法必须符合哪些条件，这些条件是经过数十年加密破解攻防战所总结出来的。

- 条件一，Confusion。对原文以最小的粒度(按字节)进行混淆生成秘文。最简单的confusion可以是A+3=D.
- 条件二，Deffusion。对原文所包含的位置信息进行打乱排布,比如将第一个字节与第四个字节的位置对换.
- 条件三，Secret Key。最终的秘文与Secret Key相关，只要持有Secret Key就可以对数据解密，将安全性的管理归于一个密钥的管理。

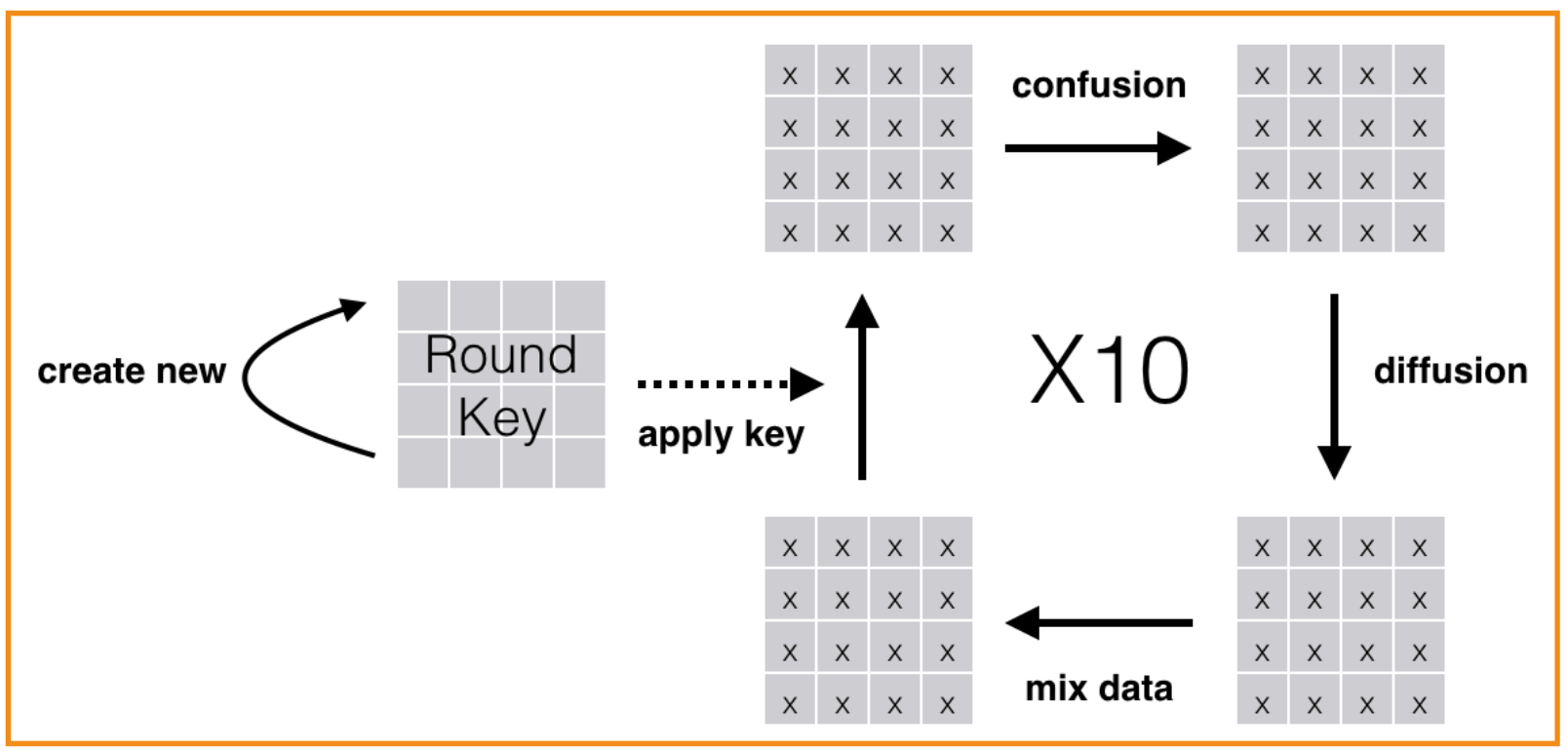
AES也是属于Cipher Block的一种，对于数据的加密都是已block为单位进行处理。针对需要加密的数据AES会先把数据分成一个个的block，每个block为16字节，可以排布成4x4的表格（又名矩阵）。如下图所示：




如果最后一块不足16字节，用0补齐（padding）。后续的加密行为都是以block为单位进行处理，这个处理过程必须满足上述所说三个条件。加密的流程如下图：



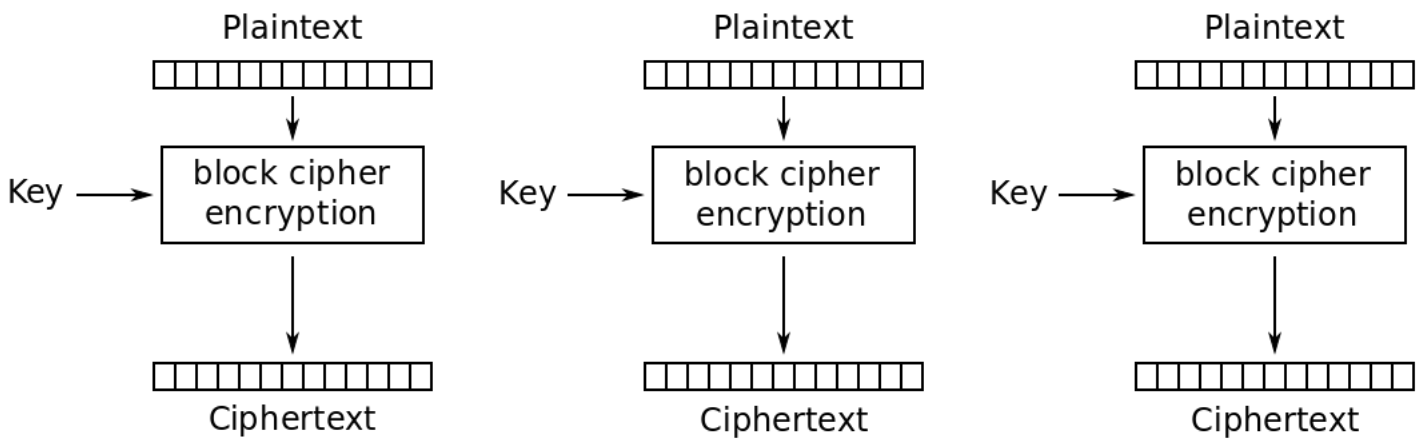

每一个block都会被单独依次处理，橙色方框表示加密的过程，这个过程包括两个方面，一方面是block本身数据的运算处理，二是与round key（也就是我们所说的对称加密算法密钥）进行运算处理。流程示意图如下：




每一个block会依次经过Confusion（条件一），Difussion（条件二），Mix Data（可以看作再一次的Confusion）。最后一步是和我们的round key进行位运算得到最后的block秘文，这样一个完整的来回称为一个round，重复10次round就完成了block加密，当然每个round当中所使用的round key也会发生变化，以保证每次block加密所使用的key都不同。

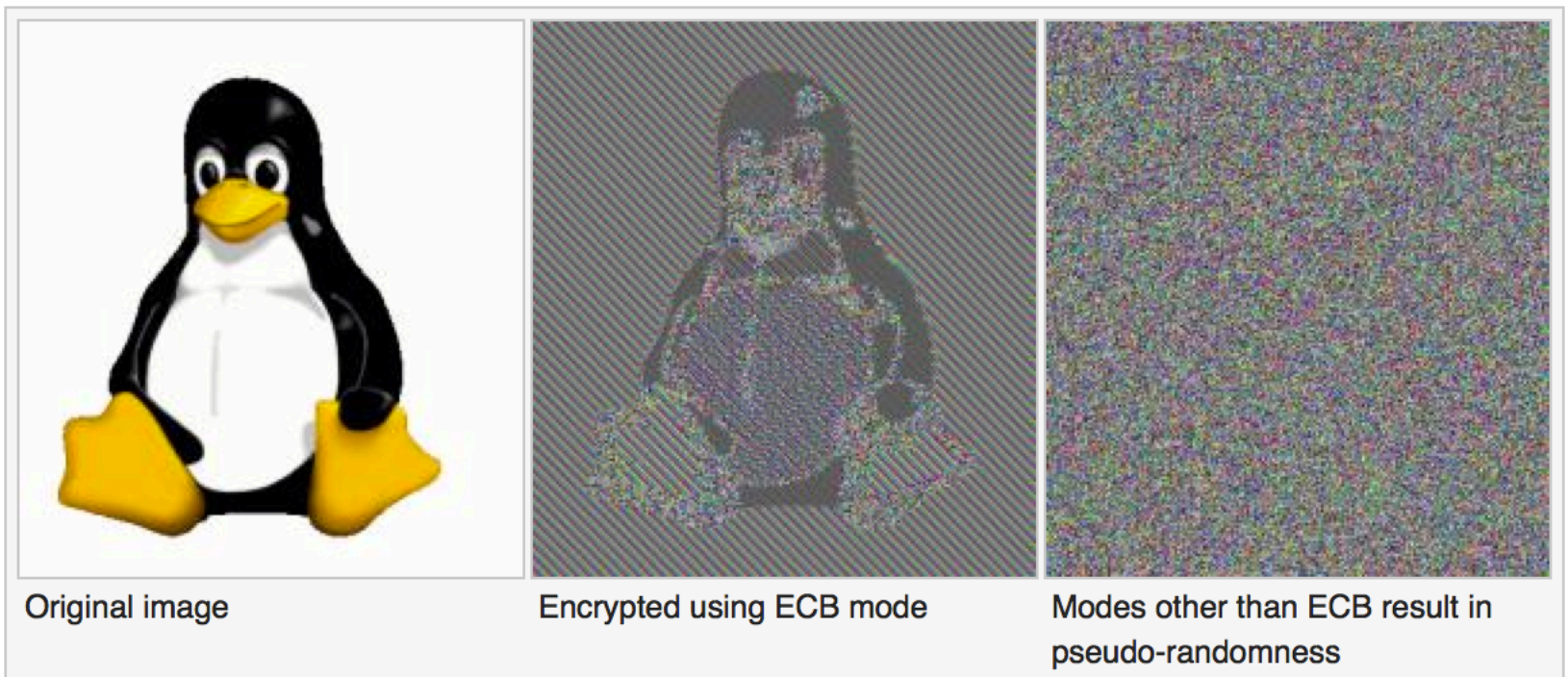
针对每个block都重复上述处理过程就完成了AES加密过程，是不是so easy。当然这是被我高度抽象简化后的流程示意图，当中每一步都有很多的细节可以深入。密钥越长，每个block处理的round次数越多，AES就越安全，不过安全性和计算性能不可兼得，一般来说，我们使用128bit的Key就可以保证算法的安全性了。对了，解密的过程就是把上面加密的过程反过来做一遍。。

**阶段四：** 明白了AES的流程，还需要了解正确的使用姿势。AES有两个经典的使用姿势，ECB模式和CBC模式。ECB模式可以用下图表示：




Electronic Codebook (ECB) mode encryption

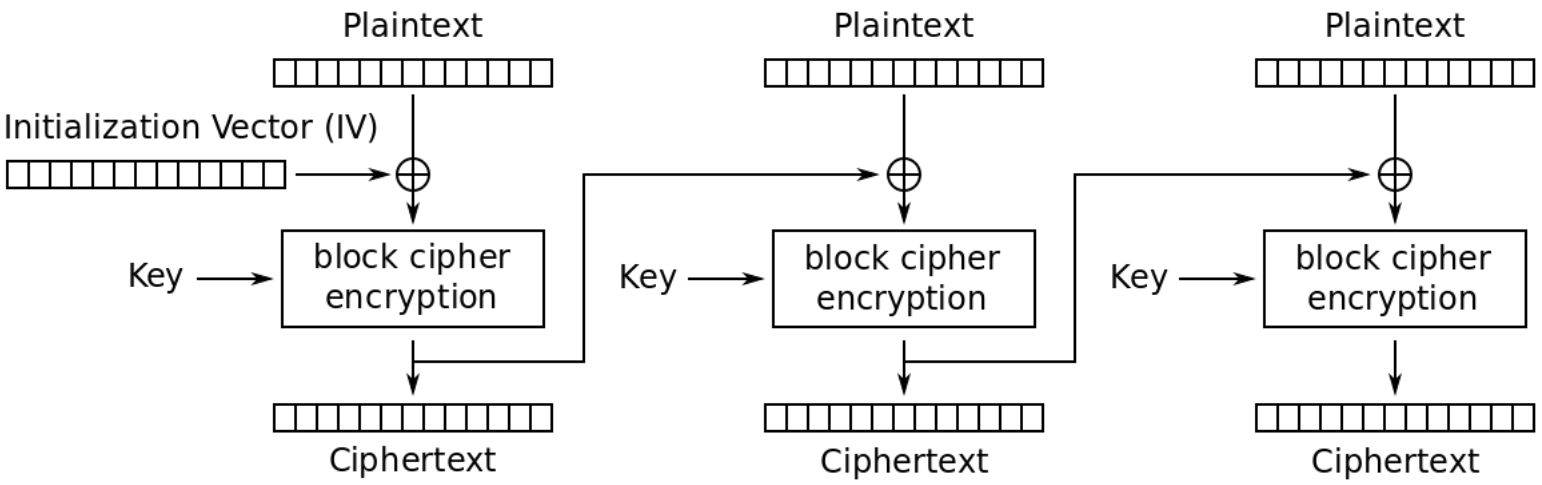
ECB模式很简单，就是针对每一个block单独进行AES运算，每个block的处理结果之间没有任何关系。使用这种方式单个block都是安全的，但对包含大量block的数据来说，没有能够隐藏data pattern，因为相同的原文会产生相同的秘文，这对图片文件来说比较致命：



Original image　Encrypted using ECB mode　Modes other than ECB result in pseudo-randomness

相同的色块产生相同的秘文，这样相同颜色在图片当中出现的规律就和原始数据一样一目了然。

CBC模式针对ECB模式的缺陷做了处理，使得每个block的AES运算结果都依赖于之前的block秘文。如下图所示：




Cipher Block Chaining (CBC) mode encryption

每一个block在加密之前都会与上个block产生的秘文进行抑或运算，这样相同的数据也会产生不同的秘文，data pattern得以隐藏。第一个block没有刻可以或处理的秘文，就传入一个IV（初始化向量）。很明显，IV不同，AES运算的结果也不同。

# 非对称加密之RSA

对称加密的安全性全系于加密密钥的管理，在非对称加密算法出现之前，如何动态的协商密钥一直是个难题，大部分的应用场景都是采用通信双方通过其他手段预先交流密钥的方式。一旦密钥泄漏，就会导致严重的安全事故。直到1976年Diffie Hellman算法出现解决了密钥协商的问题，1977年RSA诞生同时提供了密钥协商方案和电子签名方案。

RSA的使用已经相当广泛，也有很多优秀的教程解释其原理，推荐其中一篇(http://www.ruanyifeng.com/blog/2013/06/rsa_algorithm_part_one.html)。

关于RSA这种非对称加密算法，在App的使用当中，需要明白其主要作用有2个：

- **信息加密**：通信双方可以在公开的网络环境下，“安全”的商量对称加密算法所使用的密钥。
- **电子签名**：为了防止中间人攻击，通信双方在商量密钥之前可以通过签名算法确认对方的身份。

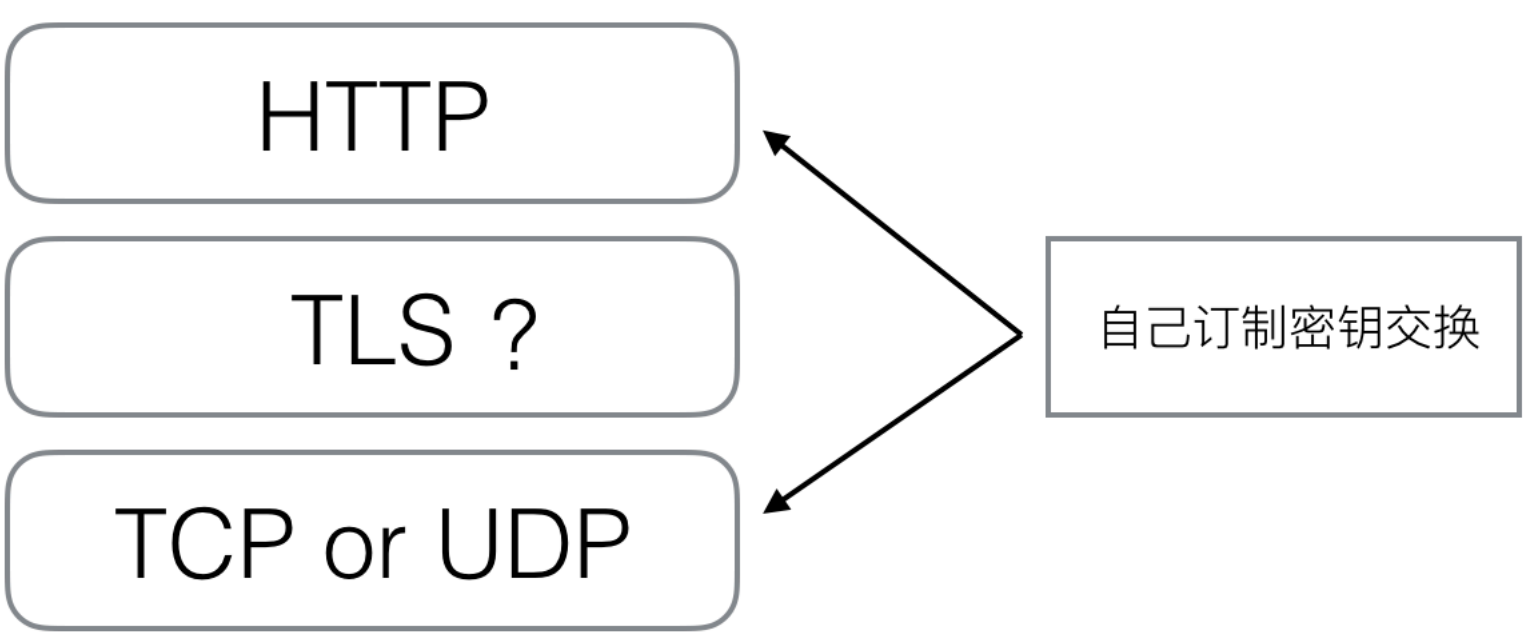
非对称加密算法本身是一种加密算法，但由于RSA本身加解密的性能在现在的计算机硬件条件下存在一定瓶颈，同时对加密数据的“安全长度”也有限制，被加密数据的长度一般要求不超过公钥的长度。所以RSA更多的是被用来商量一个密钥，如果密钥是安全的，那么后续的通信都可以使用上面提到的AES来完成，AES在性能上不存在瓶颈。

RSA算法最经典也是最广泛的应用场景是HTTPS，HTTPS的安全握手流程完整的阐释了“加密”和“签名”这两个概念。推荐一篇文章 (http://www.moserware.com/2009/06/first-few-milliseconds-of-https.html)详细的分析HTTPS的握手流程。

RSA有另一个竞争者ECC，ECC现在使用也越来越广泛。二者在安全性上都不存在问题。不过ECC额外的优势，公钥私钥的生成速度快于RSA，在需要大量生产密钥对的业务场景下ECC会是更好的选择。ECC的最短安全公钥也比RSA要短的多，224bits的ECC公钥就已经足够安全，而同等级别的RSA公钥需要长达2048bits。RSA由于实现简单，出现较早，可以预见在很长一段时间内都将和ECC共存。

# App网络应用场景

现在绝大部分的App都在使用http和https，少部分会有自己的tcp长连接通道，更少部分的app搭配udp通道或者类似QUIC这种reliable UDP协议来提升体验。不管是什么协议，只要涉及客户端和服务器的通信，就必然要实现类似https安全握手的流程，部分或者全部，开发者总是在性能和安全性之间取舍。有实力的大厂可以鱼与熊掌兼得，初创型企业往往会避开性能优化，直接跳过安全问题。




使用http，不做任何加密相当于裸奔，初级工程师都可以轻易窥探你全部的业务数据。

使用http，但所有的流量都通过预埋在客户端的key进行AES加密，流量基本安全，不过一旦客户端代码被反编译窃取key，又会回到裸奔状态。

使用http，但AES使用的key通过客户端以GUID的方式临时生成，但为了保证key能安全的送达服务器，势必要使用服务器的公钥进行加密，所以要预埋服务器证书，又涉及到证书过期更新机制。而且无法动态协商使用的对称加密算法，安全性还是有暇疵。

所以要做到真正的安全，最后还是会回归到https的流程上来，https在身份验证，密钥协商，解密算法选择，证书更新等方面都已经做了最合适的选择。

对于App开发者来说，到底选择什么样的安全策略，是在全盘了解现有安全模型的前提下，在投入，产出，风险三者之间去平衡而做出最优的选择。

# App网络安全实战

在App安全上的投入再多也不会过，不过安全问题上所投入的开发资源应该根据开发团队技术积累，产品发布deadline，用户规模及产品关注度等综合因素考量。结合这些因素我把App分为三类，各类App对安全级别的要求不同，投入产出也不同。

## 第一类，作坊式创业App

这些年伴随着移动互联网的创业潮，各式各样的app出现在用户的手机端。对于创业初期的团队来说，能把业务模型尽快实现上线当然是重中之重。但很多创业团队在安全上的投入几乎为零，所导致的安全问题比想象中的要严重。我见过不少使用http明文传输用户名密码的app，其中甚至包括一些知名传统企业。其实只要照顾到一些基础方面就能过滤掉大部分的安全漏洞了。这里提供一些小tip供创业初期团队参考：

**Tip 1:尽量使用https**

https可以过滤掉大部分的安全问题。https在证书申请，服务器配置，性能优化，客户端配置上都需要投入精力，所以缺乏安全意识的开发人员容易跳过https，或者拖到以后遇到问题再优化。https除了性能优化麻烦一些以外其他都比想象中的简单，如果没精力优化性能，至少在注册登录模块需要启用https，这部分业务对性能要求比较低。

**Tip 2:不要传输密码**

不知道现在还有多少app后台是明文存储密码的。无论客户端，server还是网络传输都要避免明文密码，要使用hash值。客户端不要做任何密码相关的存储，hash值也不行。存储token进行下一次的认证，而且token需要设置有效期，使用refresh token去申请新的token。

**Tip 3:Post并不比Get安全**

事实上，Post和Get一样不安全，都是明文。参数放在QueryString或者Body没任何安全上的差别。在Http的环境下，使用Post或者Get都需要做加密和签名处理。

**Tip 4:不要使用301跳转**

301跳转很容易被Http劫持攻击。移动端http使用301比桌面端更危险，用户看不到浏览器地址，无法察觉到被重定向到了其他地址。如果一定要使用，确保跳转发生在https的环境下，而且https做了证书绑定校验。

**Tip 5:http请求都带上MAC**

所有客户端发出的请求，无论是查询还是写操作，都带上MAC（Message Authentication Code）。MAC不但能保证请求没有被篡改（Integrity），还能保证请求确实来自你的合法客户端（Signing）。当然前提是你客户端的key没有被泄漏，如何保证客户端key的安全是另一个话题。MAC值的计算可以简单的处理为hash（request params＋key）。带上MAC之后，服务器就可以过滤掉绝大部分的非法请求。MAC虽然带有签名的功能，和RSA证书的电子签名方式却不一样，原因是MAC签名和签名验证使用的是同一个key，而RSA是使用私钥签名，公钥验证，MAC的签名并不具备法律效应。

**Tip 6:http请求使用临时密钥**

高延迟的网络环境下，不经优化https的体验确实会明显不如http。在不具备https条件或对网络性能要求较高且缺乏https优化经验的场景下，http的流量也应该使用AES进行加密。AES的密钥可以由客户端来临时生成，不过这个临时的AES key需要使用服务器的公钥进行加密，确保只有自己的服务器才能解开这个请求的信息，当然服务器的response也需要使用同样的AES key进行加密。由于http的应用场景都是由客户端发起，服务器响应，所以这种由客户端单方生成密钥的方式可以一定程度上便捷的保证通信安全。

**Tip 7:AES使用CBC模式**

不要使用ECB模式，原因前面已经分析过，记得设置初始化向量，每个block加密之前要和上个block的秘文进行运算。

# 第二类，正规军App

**All Traffic HTTPS**

全站使用HTTPS，而且是强制使用。baidu到今天（2016.04.13）还没有强制使用HTTPS。所有的流量都应该在HTTPS上产生，没有人可以决定哪些流量是可以不用考虑安全问题的。如果自建长连接使用tcp，udp或者其他网络协议，也应该实现类似HTTPS的密钥协商流程。

**Certificate Pinning**

RSA的签名机制虽然看着安全，一旦出现上游证书颁发机构私钥泄漏，或者签名流程发现漏洞等情况，中间人攻击还是会导致数据被第三方破解甚至被钓鱼。Certificate Pinning是一种与服务器证书强绑定的机制，要么绑定证书本身，需要证书更新机制配合加强安全性，要么使用公钥绑定，这样更新证书的时候只要保证私钥不变即可。现在流行的HTTP framework，iOS端如AFNetworking，Android端如OKHttp都支持Certificate Pinning。

**Perfect Forward Secrecy**

很多人会觉得非对称加密算法足够安全，只要使用了RSA或者AES，加密过后的数据就认为安全。但没有绝对的安全，无论是RSA或者AES算法本身都有可能在未来某一天被破解，正如当年的DES，甚至有传言NSA正如当年掌握了differential cryptanalysis一样，现在已经获取了某种方法来破解当前互联网当中的部分网络流量，至于到底是RSA还是AES就不得而知了。

未来计算机的计算能力是个未知数，或许某一天brute force能够暴力破解的密钥长度会远超128bits（现阶段上限应该在80bits）。

2014年1月3日，美国国家安全局（NSA）正在研发一款用于破解加密技术的量子计算机，希望破解几乎所有类型的加密技术。

即使算法本身没有被破解，密钥也有可能被泄漏，技术上的原因或者政策上的因素都可能导致RSA或者ECC的私钥被泄漏。所以尽可能针对不同的session使用不同的key能够使的我们的数据更佳安全。

Forward Secrecy就是为了避免某个私钥的泄漏或者被破解而导致历史数据一起泄漏。现在google的https配置所使用的是TLS_ECDHE_RSA算法，每次对称密钥的协商都是使用ECC生成临时的公钥私钥对（之前提到过ECC在快速生成密钥对上有优势），身份验证使用RSA算法进行签名。

**每天跟踪信息安全动态**

安全的攻防战不会有穷尽的一天，算法的更替会伴随着人类对知识的无尽渴望延绵至不可预知的未来。AES说不定哪天被破解了，openSSL可能又出现新的漏洞了，google又提倡新的安全模型了，NSA的量子计算机说不准已经在悄悄解密google的流量了，每天跟踪八卦最新业界动态才是码农避免因bug而背黑锅的不二法宝。

# 第三类，带节操正规军App

现在互联网早已渗入每个人的平常生活当中，当我们的行为越来越多的迁移到互联网这个媒介当中之后，行为本身及所产生的关联数据都将被滴水不漏的记录起来，特别是在大数据研究兴起的当下，服务提供商总是希望尽可能多的记录用户所有的行为数据。每个互联网产品的使用者都成了样本，你的购物记录，商品浏览历史，搜索引擎搜索记录，打车记录，租房记录，股票记录，甚至聊天记录等等都是样本，毫不夸张的说，如果将淘宝，微信，支付宝，快滴，美团等等高频次产品数据统一分析，基本上可以将你的身高，性别，年龄，三维，家庭住址，恋爱史，家庭成员，甚至是个人喜好，性格等等完美的呈现出来，其后果远不是一个骚扰电话带来的隐私泄漏那么简单。

移动互联网的大部分使用者还不具备强烈的安全意识，当你用手机号作为登录id方便记忆的同时，骚扰电话就可能随时来临，你在百度输入租房关键字，下一秒中介就已经电话打上门。当你允许app上传通讯录匹配可能认识的好友同时，你认识哪些人就变得一清二楚，你p2p借贷未及时归还时，你的亲朋好友第二天就收到了催债电话。我们在享受移动互联网的便利同时，付出的是个人隐私这种隐形成本。下一次，当我们感叹新app好用便利的同时，静思三分钟，好好想想我们的哪些隐私又被当白菜卖了。

在互联网受众的安全意识普遍觉醒之前，只能靠app开发商，服务提供商的节操来保证用户信息隐私安全。

带节操的App在打算记录用户行为或者数据之前会考虑下是不是真的有需要，用户的确会有需要查询历史购买记录，但有多少人会在意自己几年前花几个小时浏览了杜蕾斯的产品。

服务器作为数据存储或者转发的媒介是不是真的需要了解真实的数据为何？现在WhatsApp，Telegram都已经支持端到端的加密聊天方式，服务器本身看到的都是秘文，只做秘文转发处理，带着这样的节操设计产品，用户才会觉得安全。

WhatsApp的端到端加密安全模型是怎么样实现的呢？非常值得学习。

简单来说是严格遵循forward secrecy。每个用户在注册成功之后会在服务器存一对永久的Identity Key，一对临时的Signed Pre Key（Signed Pre Key由Identity Key签名，每隔一段时间变化一次），n对临时的One-Time Pre Key（每次建立session消耗一个）。

每次session开始建立的时候使用Identity Key，Signed Pre Key， One Time Key生成Master Secret。Master Secret再通过HKDF算法生成对称加密使用的Root Key，Chain Key，Message Key。

Forward Secrecy体现在每次sender发送的消息被ack后，都会交换新的临时ECC Key对，并更新Root Key，Chain Key，Message Key。这样网络中的流量即使被第三方缓存起来，而且某一天某个Key Pair的私钥被破解，也不会对之前的流量产生安全影响。ECC Key对会随着消息的发送不停的“Ratcheting”。这是属于非对称加密的Forward Secrecy。

在sender的消息被ack之前，也就是新的ECC Key对交换成功之前，Message Key也会通过HKDF算法不停的“Ratcheting”，确保每条消息所使用的对称密钥也不相同。这是属于对称加密的Forward Secrecy。

有兴趣深入了解的同学可以自己google：WhatsApp Security WhitePaper。

---


**上一篇**
搭建数据驱动型Android架构 (**/blog/dda-android/**)

**下一篇**
Android线程的正确使用姿势 (**/blog/android-threading/**)