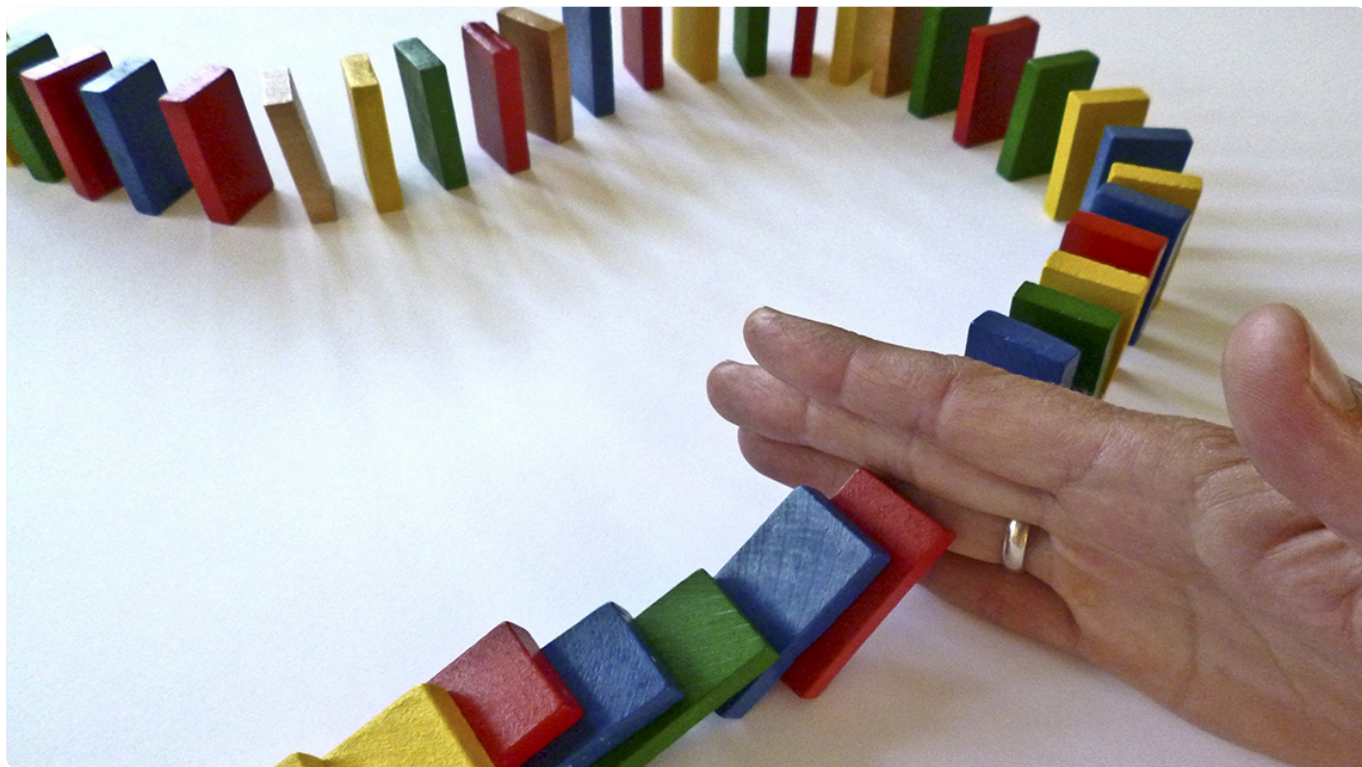


16 | 安全管理：数字世界的守护

2019-06-07 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 17:48 大小 16.31M



你好，我是七牛云许式伟。今天我们要聊的话题是操作系统的最后一个子系统：安全管理。

数字世界是高效的，但数字世界也是脆弱的。在越来越多的日常生活被数字化的今天，安全问题也越来越凸显出了它的重要性。

有经验的安全工程师都知道，做好安全的基本逻辑是：不要开太多的门和窗，最好所有人都在同一道门进出，安全检查工作就可以非常便利地进行。

要想构建一个安全可靠的环境，从最底层就开始设计显然是最好的。所以安全管理是一个基础架构问题。现代操作系统必然会越来越关注安全性相关的问题。因为一旦安全问题严重到触及人们的心里防线，整个数字世界都有可能随之崩塌。

让我们从头回顾一下操作系统安全能力的演进。

病毒与木马

首先是实模式的操作系统，以微软的 DOS 系统为代表。实模式的操作系统进程都运行在物理地址空间下。

这意味着，每个软件进程都可以访问到其它软件进程（包括操作系统）的内存数据，也可以随意地修改它。所以这个时期的计算机是非常脆弱的，它选择的是信任模式：我相信你不会搞破坏。

不过，好在这个时期网络还并不发达，所以一个单机版本的恶意软件，能够干的真正恶意的事情也很有限。这一时期恶意软件以计算机病毒为主，其特征主要是繁衍自己（复制自己），对计算机系统本身做某种程度的破坏。

现代操作系统基本上都是保护模式的操作系统。保护模式就是让软件运行的内存地址空间隔离，进程之间相互不能访问（除非基于共享内存技术，那也是进程自己主动选择，与被动无感知的情况下被人窥视不同）。

这从安全角度来说，是很重要的进步。不管怎么说，内存数据是最为敏感的，因为它无所不包。况且，从 Windows 开始，互联网逐步进入人们的视野。计算机的联网，一下子让安全问题变得严峻起来。

恶意软件目的开始变得不单纯。它不再只是黑客的技术炫耀，而是切切实实的黑色产业链的关键依赖。

这一时期恶意软件开始以木马为主。木马和病毒一样会去繁衍自己（复制自己），但是它较少以破坏计算机的运行为目的，相反它默默隐藏起来，窃取着你的隐私。然后，它再通过互联网把窃取的信息默默地传递出去（比如通过电子邮件）。

哪些信息是木马感兴趣的？有很多。比如以下这些信息：

键盘按键；

剪贴板的内容；

内存数据；

文件系统中关键文件的内容；

.....

你可能奇怪，前面不是说保护模式已经把内存数据隔离了么，为什么木马还是能够取到内存数据？

其实这一点不难想明白，虽然跨进程已经无法取得数据了，但是木马本来就是靠复制自己，把自己伪装成正常软件的一部分。这样，木马程序和正常的软件代码同属于一个进程内，所有信息对其仍然一览无余。

为了彻底阻止木马程序篡改正常的应用程序，聪明的操作系统创造者们想到了好方法：数字签名。

这本质上是白名单技术。所有正常发布的软件都到操作系统厂商那里登记一下。这样，一旦木马去修改软件，把自己附加上去，这个软件的签名验证就通不过，也就直接暴露了。

其实 Windows 操作系统已经引入了数字签名的概念，可以用以鉴别软件的可信度。但是考虑到从开放转向封闭有极大的历史负担，所以无论是 Windows 还是 Mac，都没有完全杜绝无签名的软件，最多当你运行无数字签名的软件时，会给个不可信的警告。

第一个大规模把软件发布变成一个封闭环境的是苹果的 iOS 操作系统。苹果通过引入 App Store，要求所有应用发布都必须通过 App Store 进行。今天无论是 Android 还是 iOS 操作系统都基于应用市场这样的封闭软件发布的形态。

这样一来，软件无法被非法修改，木马基本上就无所遁形了。当然，这并不代表木马在这些平台上就消失了。虽然不容易，但是通过感染开发人员的软件开发环境，还是可以在软件编译或其它环节中把木马注入到要发布的软件中。

要发现这种异常，iOS 和 Android 系统的厂商对软件进行数字签名前，往往会对其进行安全扫描，以发现各种潜在的安全风险。一旦某个软件被鉴定为恶意软件，就无法通过数字签名，也无法发布到应用市场上。

通过这些机制，木马很难再有机会得到传播。

软件的信息安全

但是，这意味着我们没有安全风险了么？当然不是。在移动设备上，安全问题的大环境发生了巨大的变化。

首先，移动时代随着我们数字世界对现实生活影响的加深，我们越来越多的敏感信息更加容易被软件触及。有很多新增的敏感信息是 PC 时代所不具备的，例如：

通讯录和通话记录；

短信；

个人照片和视频；

个人地理位置（GPS）信息；

移动支付的支付密码、支付验证码；

录像和录音权限；

通话权限；

.....

正因为如此，尽管操作系统正变得越来越安全，但我们面临的安全威胁却也在日趋严重。

其实，iOS 操作系统在安全管理上的考虑不可谓不周全。

首先，在软件隔离机制上，除了基于 CPU 的保护模式，确保软件之间的内存隔离外，iOS 还引入了沙盒系统（Sandbox），确保软件之间文件系统隔离，相互之间不能访问对方保存在磁盘上的文件。

其次，通过上面我们已经提及的数字签名机制，防止了软件被恶意篡改，让病毒和木马无法传播繁衍。

最后，对涉及敏感信息的系统权限进行管控。各类敏感信息的授予均是在应用程序使用的过程中进行提示，提醒用户注意潜在的安全风险。

在这一点上，Android 操作系统往往则是在安装软件时索要权限。这两者看似只是时机不同，但是从安全管理角度来说，iOS 强很多。

还没有见到软件真身就让用户判断要不要给权限，用户往往只能无脑选择接受。而如果是在软件运行到特定场景时再索要权限，那么权限给不给就有合理的场景支持决策。

但是，在利益面前，软件厂商们是很难抵御住诱惑的。所以不仅仅是恶意软件会去过度索要系统权限，很多我们耳熟能详的常规软件也会索要运行该软件所不需要的权限。

移动时代，恶意软件的形态已经再一次发生变化。它既不是病毒也不是木马，而是“具备实用功能，但背地却通过获取用户的敏感信息来获利”的应用软件。

它通过诱导用户下载，然后在软件安装或者使用时索要敏感信息的获取权限。

一个软件到底是正常的还是恶意的？边界已经越来越模糊了。

以前病毒和木马都有复制和繁衍自己，这样一个显著的特征，但如今病毒和木马的复制繁衍能力已经被操作系统的安全机制所阻止，所以恶意软件和普通软件一样，都是通过某种手段吸引用户下载安装。

怎么保护好用户的隐私信息？道高一尺，魔高一丈。攻防之间的斗争仍将继续下去。

网络环境的信息安全

如果我们不轻易尝试不可信的软件，就可以一切安全无虞？并不然，我们还要考虑我们的计算机所处的网络环境安全问题。

我们上网过程需要经过一系列的中间节点，有交换机，有路由器。我们的上网产生的所有数据包，都经由这些中间节点，**这意味着我们有以下三个级别的安全风险。**

被窃听的风险。可能会有人在这些节点上监听你访问和提交的内容。

被篡改的风险。可能会有人在这些节点上截获并修改你访问的内容。

被钓鱼的风险。可能会有人冒充你要访问的服务提供方和你通讯。

虽然大部分的中间节点由网络运营商提供，我们刨除这些节点被黑客所黑的情形，基本上认为可信。但这并不绝对，至少在中国，运营商修改中转的数据包这样的事情是干得出来的，常见的手法有：

在正常的 HTML 页面插入广告；

修改用户下载的 apk 文件，替换成自己想分发的 apk 文件；

修改 404 类型的 HTML 页面，替换成自己的搜索引擎的搜索页；

.....

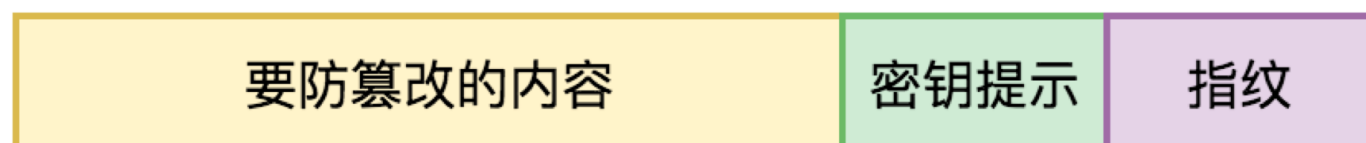
其次是 WiFi 路由器。WiFi 路由器因为其提供方鱼龙混杂，天生是安全问题的大户。运营商能够干的事情它全都可以干，甚至可以更加肆无忌惮，以李鬼替换李逵，钓鱼的风险并不低。

比如你以为登录的是交通银行官网，它可能给你一个一模一样外观的网站，但是一旦你输入用户名和密码就会被它偷偷记录下来。

怎么解决中间人问题？

首先是怎么防篡改。应用场景是电子合同 / 公章、网络请求授权（例如你要用七牛的云服务，需要确认这个请求的确是你，而不是别人发出的）等。这类场景的特征是不在乎内容是否有人看到，在乎的是内容是不是真的是某个人写的。

解决方法是数字签名技术。一般来说，一个受数字签名保护的文档可示意如下：



其中，“要防篡改的内容”是信息原文。“密钥提示”是在数字签名的“密钥”有多个的情况下，通过“密钥提示”找到对应的“密钥”。如果用于保护信息的“密钥”只有一个，那么可以没有“密钥提示”。“指纹”则是对信息使用特定“密钥”和信息摘要算法生成的信息摘要。

大部分情况下，数字签名的信息摘要算法会选择 HMAC MD5 或者 HMAC SHA1。在 Go 语言中，使用上示意如下：

复制代码

```
1 import "crypto/hmac"
2 import "crypto/sha1"
```



```
3 import "encoding/base64"
4
5 textToProtected := " 要防篡改的内容 "
6 keyHint := "123"
7 key := findKey(keyHint) // 根据 keyHint 查找到 key []byte
8
9 h := hmac.New(sha1.New, key) // 这里用 sha1, 也可以改成别的
10 h.Write([]byte(textToProtected))
11 textDigest := base64.URLEncoding.EncodeToString(h.Sum(nil))
12 textResult := textToProtected + ":" + keyHint + ":" + textDigest
```

得到的 textResult 就是我们期望的不可篡改信息。验证信息是否被篡改和以上这个过程相反。

首先根据 textResult 分解得到 textToProtected、keyHint、textDigest, 然后根据 keyHint 查找到 key; 再根据 textToProtected 和 key 算一次我们期望的信息摘要 textDigestExp。

如果 textDigestExp 和 textDigest 相同, 表示没被篡改, 否则则表示信息不可信, 应丢弃。

如果我们希望更彻底的隐私保护, 避免被窃听、被篡改、被钓鱼, 那么数字签名就不顶用了, 而需要对内容进行加密。

加密算法上, 一般分为对称加密和非对称加密。对称加密是指用什么样的密钥 (key) 加密, 就用什么样的密钥解密, 这比较符合大家惯常的思维。

非对称加密非常有趣。它有一对钥匙, 分私钥 (private key) 和公钥 (public key)。私钥自己拿着, 永远不要给别人知道。公钥顾名思义是可以公开的, 任何人都允许拿。

那么公私钥怎么配合? 首先, 通过公钥加密的文本, 只有私钥才能解得开。这就解决了定向发送的问题。网络中间人看到加密后的信息是没有用的, 因为没有私钥解不开。

另外, 私钥拥有人可以用私钥对信息进行数字签名 (防止篡改), 所有有公钥的人都可以验证签名, 以确认信息的确来自私钥的拥有者, 这就解决了请求来源验证的问题。

那么 A、B 两个人怎么才能进行安全通讯呢？首先 A、B 两人都要有自己的公私钥，并把公钥发给对方。这样 A 就有 A-private-key、B-public-key，B 就有 B-private-key、A-public-key。通讯过程如下所示。

A 向 B 发信息 R。具体来说，A 首先用 A-private-key 对 R 进行签名，得到 (R, R-digest)；然后用 B-public-key 对 (R, R-digest) 加密，得到 encoded (R, R-digest)；然后把最终的加密信息发出去。

B 收到 encoded (R, R-digest)，用 B-private-key 解密得到 (R, R-digest)，然后再用 A-public-key 验证信息的确来自 A。

B 理解了 R 后，回复信息给 A。这时两人的角色互换，其他同上。

非对称加密机制非常有效地解决了在不可信的网络环境下的安全通讯问题。但是它也有一个缺点，那就是慢。相比之下，它的速度比对称加密慢很多。

所以，一个改善思路是结合两者。非对称加密仅用于传输关键信息，比如对称加密所需的密码。完整的通讯过程如下所示。

A 生成一个临时用的随机密码 random-key。

A 向 B 发送 random-key，机制用的就是上面的非对称加密，基于 B-public-key。

B 收到 A 发送的 random-key，把它记录下来，并回复 A 成功。回复的信息可以基于 random-key 做对称加密。

此后，A 向 B 发、B 向 A 发信息，都用 random-key 作对称加密，直到本次会话结束。

你可能发现，整个过程中 A 自己已经不再需要非对称的公私钥对了。只要 A 事先有 B 的公钥 (B-public-key) 就可以。

当然，上面我们的讨论，没有涉及 B 如何把自己的 B-public-key 交给对方的。在假设网络不可信的前提下，这似乎是个难题。

我觉得有两个可能性。一个是 A 和 B 很熟悉，平常都经常一起玩。那么他们交换 public-key 完全可以不依赖任何现代通讯设备，包括电话和互联网，而是写在一张纸上，某天聚会的时候交换给对方。

另一个是更为常见的互联网世界场景：我要访问一个网站。我怎么才能避免被窃听、被篡改、被钓鱼？

通常我们用 HTTPS 协议。

在 HTTPS 协议中，第一步是 A 作为客户端（Client）去获取 B 作为网站的公钥（B-public-key）。

怎么获取？如果我们认为网络不可信，那么我们就需要找一个可信的中间人，第三方权威机构 G，向它获取我们要访问的网站的公钥（B-public-key）。

当然这里就有一个前提，我们已经提前拥有第三方权威机构 G 的公钥（G-public-key）了。过程如下：

A 用 G-public-key 加密信息，向 G 发了求公钥请求，参数为 B 网站的域名（domain）。

G 查到域名（domain）对应的公钥（B-public-key），用 G-private-key 做数字签名，得到（B-public-key, B-public-key-digest），并返回它。注意这里并不需要加密，因为没有保密需求。

A 收到（B-public-key, B-public-key-digest），用 G-public-key 验证信息的确来自于权威机构 G，于是选择相信 B-public-key。

有了 B-public-key，客户端 A 就可以愉快地上网，不必担心网络通讯的安全了。

但是，HTTPS 并不能完全解决钓鱼问题。它假设用户对要访问的网站域名（domain）可靠性有自己的判断力。

这当然并不是事实。所以，高级一点的浏览器（例如 Google Chrome），它会建立不靠谱网站域名的数据库，在用户访问这些网站时进行风险提示。

更多的信息安全话题

上面我们更多从服务终端用户角度，操作系统和浏览器以及我们的应用程序需要考虑的信息安全问题。有以下这些信息安全问题没有涉及：

服务器的安全问题（DDOS 攻击、漏洞与入侵）；

企业信息安全；

社会工程学的安全问题；

.....

结语

总结一下，我们今天聊了软件安全态势的演变过程，从最早的病毒和木马，演化到今天敏感信息如通讯录等内容的窃取，正常软件与恶意软件的判断边界越来越模糊。

我们也聊了网络环境带来的安全问题。今天主流的假设是网络链路是不可信的，在不可信的网络之上如何去做安全的通讯，可以做到防窃听、防篡改、防钓鱼。这也是苹果前几年强制要求 iOS App 必须走 HTTPS 协议的原因。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。本章关于操作系统的话题到此就结束了。下一节我们结合前面的内容，讨论并实战架构第一步，怎么做需求分析。

如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。




许式伟的架构课

从源头出发，带你重新理解架构设计

许式伟
七牛云 CEO



新版升级：点击「请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 15 | 可编程的互联网世界

精选留言 (2)

写留言



pawhrmyki

2019-06-07



看过一个讲rsa原理的文章，非常清楚，推荐给大家~~

http://www.ruanyifeng.com/blog/2013/06/rsa_algorithm_part_one.html



勇闯天涯

2019-06-07



凌晨4点多醒来，看到许老师更新，一口气读完，大脑出奇的清醒，再读前面几篇。