第 3 章

移动终端漏洞类型

通过对移动终端目前发现的漏洞进行分析可以发现，移动终端的攻击面主要包括操作系统、App、固件、硬件、通信协议等方面，本章将详细讨论各部分的漏洞情况。移动终端漏洞主要参见 CVE 通用安全漏洞库、国家信息安全漏洞库、国家信息安全共享平台和 Google 安全公告。

3.1 操作系统漏洞

3.1.1 操作系统分布

产业界各方都将移动智能终端当作自己进军移动互联网领域的入口，但由于自身优势和经营理念差异，其发展模式也各不相同。按照操作系统的授权方式和应用商店的运营方式，主要可分为封闭端到端模式、半封闭模式和开放开源模式。

（1）封闭端到端模式是指终端厂商完全控制终端产品的生产，基于封闭的操作系统平台构建端到端闭合的应用生态系统，在终端中深度内置自营业务，并对第三方应用的开发、测试、上架和使用全程控制，不允许第三方应用商店存在，如苹果公司、黑莓公司等。

（2）半封闭模式是指操作系统厂商授权给 OEM 厂商或者终端设备厂商生产终端产品，但不向其开放源代码。同时，操作系统厂商构建封闭端到端的应用生态系统，在操作系统中深度内置自营业务，同时对第三方应用的开发、测试、上架和使用全程控制，不允许未经审核认证的应用在操作系统上使用，如微软公司 WindowsPhone 操作系统。

（3）开放开源模式是操作系统厂商对源代码开放开源，任何终端厂商均可针对操作系统进行定制和修改，任何硬件开发商均可为操作系统开发驱动程序，从而组成范围更大的产业联盟。同时，操作系统厂商对第三方应用的开发、传播一般不做任何限制，允许任何应用在操作系统上运行。开放开源模式以 Google 公司的 Android 系统为代表，普遍被终端领域后进入者所采用，发展极为迅猛。

统计机构 Staticsta 发布了 2019 年第二季度移动终端操作系统市场份额占比，如图 3-1

所示。其中操作系统 Android 占比最高，为 77.14%，与 2019 年第一季度占比基本一致；iOS 系统占比 22.83%，位居第二；其他（Other）移动终端操作系统相加占比仅 0.03%，完全不及 iOS 系统和 Android 系统。

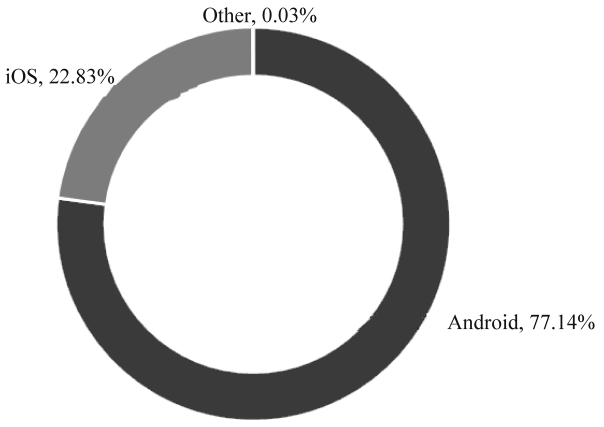

图 3-1　2019 年第二季度移动终端操作系统市场份额占比

如图 3-2 所示，在 Android 系统中，Android 8.1 为 20.03%，占比最高，较 2019 年第一季度占比上升约 2%；Android 6.0 占比为 16.16%；Android 8.0 占比为 14.96%；Android 9.0 占比为 12.18%，较 2019 年第一季度占比上升约 8%；Android 7.1 占比为 9.53%；Android 5.1 占比为 9.41%；Android 7.0 占比为 7.29%；Android 4.4 占比为 4.25%；Android 5.0 占比为 2.74%；其他版本（Other）Android 系统占比为 3.45%。

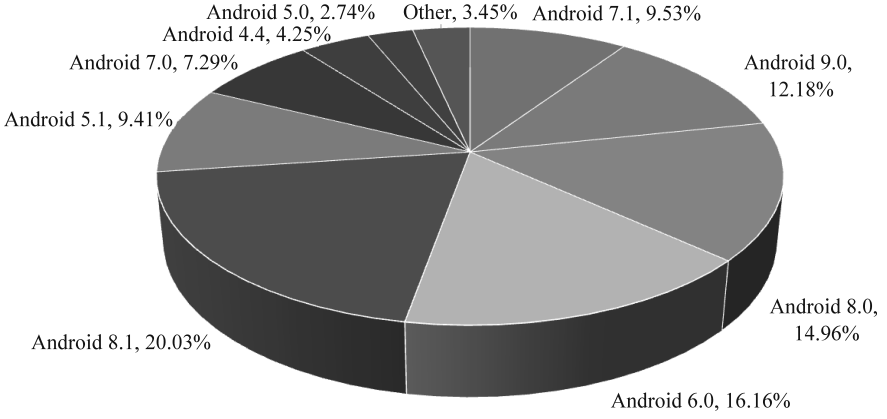

图 3-2　Android 系统各版本占比（2019 年第二季度）

如图 3-3 所示，在 iOS 系统中，iOS 12 为 56.86%，占比最高，较 2019 年第一季度占比上升约 7%；iOS 11 占比为 19.44%；iOS 10 占比为 10.43%；iOS 9 占比为 5.37%；iOS 8 占比为 2.40%；iOS 7 占比为 0.64%；其他版本（Other）iOS 系统占比为 4.86%。

不同移动智能终端的发展模式也使得终端面临的安全威胁程度不一。在封闭端到端模式和半封闭模式下，终端厂商在封闭的生态系统中占据绝对主导地位，承担一定的第三方应用管理责任，因此针对其的恶意代码和违反其经营理念的数字内容较少。但是，终端

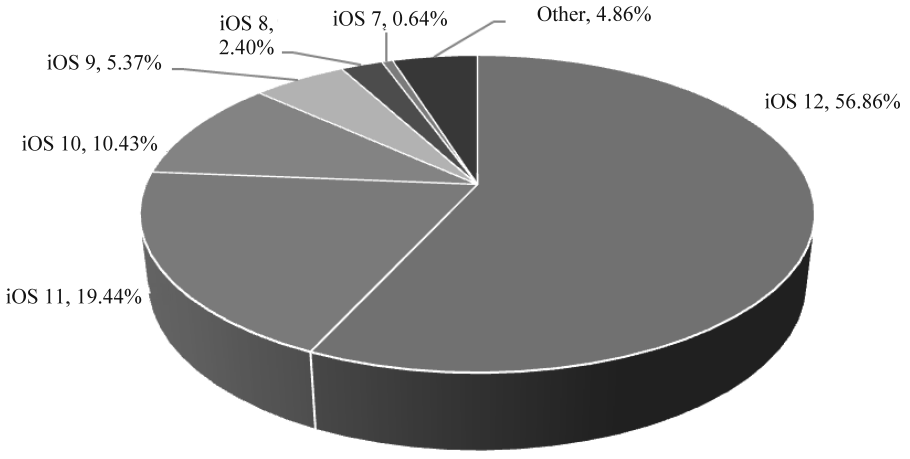

图 3-3　iOS 系统各版本占比(2019 年第二季度)

厂商自身的各种行为难以得到有效监管和制约,如苹果公司能够对其出售的所有移动终端上的应用程序进行远程安装和卸载。而在开放开源模式下,操作系统厂商基本不对应用程序进行任何控制,导致针对开源操作系统的恶意代码和不良信息呈现泛滥趋势。

3.1.2　操作系统架构

目前,移动终端常用的操作系统包括 Android、iOS、Windows Mobile、Symbian、BlackBerry OS,以及其他操作系统(如 Linux、PalmOS、WebOS、MeeGo…),其中 iOS 和 Android 系统是目前主流的移动终端操作系统,占据了绝对的主导地位。由于 iOS 系统的闭源特性,其系统特征无法进行详细研究,因此下面以 Android 系统为例进行说明。

Android 是一种基于 Linux 的自由及开放源代码的操作系统。主要使用于移动设备,如智能手机和平板计算机,由 Google 公司和开放手机联盟领导及开发。目前,Android 尚未有统一的中文名称,较多人使用“安卓”。Android 系统最初由 AndyRubin 开发,主要支持手机。2005 年 8 月,由 Google 公司收购注资。2007 年 11 月,Google 公司与 84 家硬件制造商、软件开发商及电信营运商组建开放手机联盟共同研发改良 Android 系统。随后 Google 公司以 Apache 开源许可证的授权方式,发布了 Android 的源代码。第一部 Android 智能手机发布于 2008 年 10 月。随后 Android 逐渐扩展到平板计算机及其他领域上,如电视、数码照相机、游戏机、智能手表等。

Android 系统在正式发行前,最开始拥有两个内部测试版本,并且以著名的机器人名称对其进行命名,它们分别是阿童木(Android Beta)和发条机器人(Android 1.0)。后来由于涉及版权问题,Google 公司将其命名规则变更为用甜点作为它们系统版本的代号的命名法。甜点命名法开始于 Android 1.5 发布。作为每个版本代表的甜点尺寸越变越大,然后按照 26 个英文字母表顺序如下:纸杯蛋糕(Cup Cake,Android 1.5)、甜甜圈(Donut,Android 1.6)、松饼(English Muffin,Android 2.0/2.1)、冻酸奶(Froyo,Android 2.2)、姜饼(Gingerbread,Android 2.3)、蜂巢(Hive,Android 3.0)、冰激凌三明治(Ice Cream Sandwich,Android 4.0)、果冻豆(Jelly Bean,Android 4.1 和 Android 4.2)、奇巧(KitKat,Android 4.4)、棒棒糖(Lollipop,Android 5.0)、棉花糖(Marshmallow,Android 6.0)、牛轧

糖(Nougat，Android 7.0)、奥利奥(Oreo，Android 8.0)、派(Pie，Android 9.0)和未知(Android 10.0Q)。

Android的系统整体架构和其操作系统一样，采用了分层的架构，如图3-4所示。从架构图看，Android系统分为4层，从高层到低层分别是应用程序层(Applications)、应用程序框架层(Application Framework)、Android运行库层(Android Runtime)和Linux内核层(Linux Kernel)。

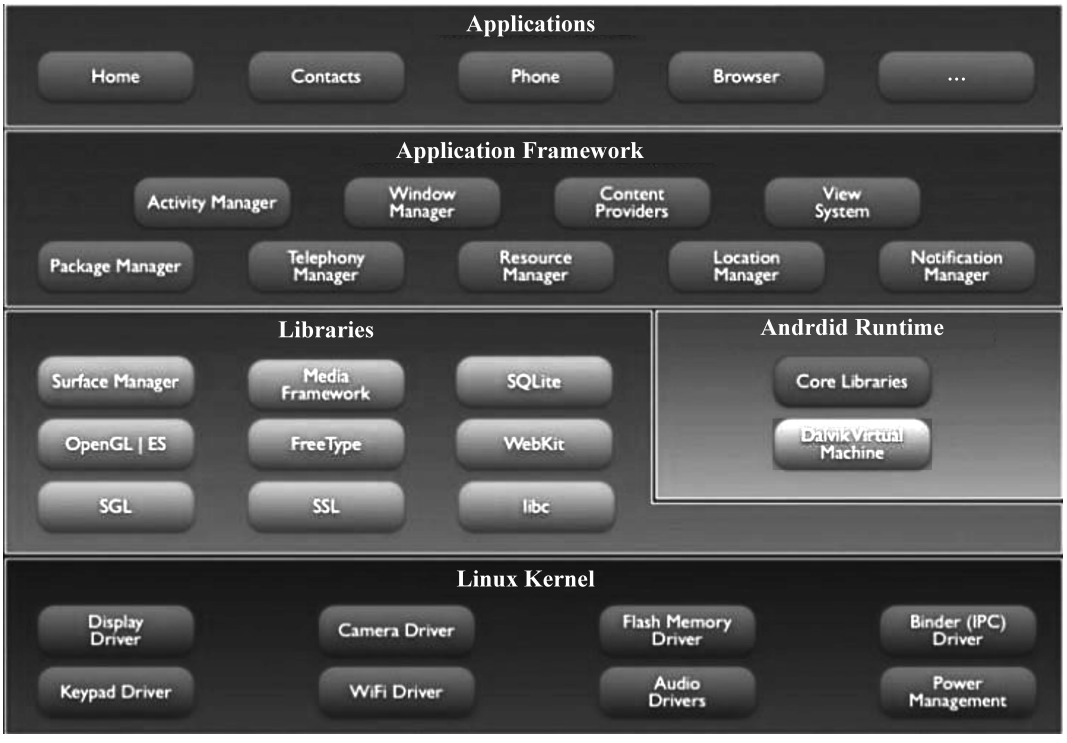

图3-4　Android系统整体架构图

1. 应用程序层

Android会同一系列核心应用程序包一起发布，该应用程序包包括客户端、短消息(SMS)程序、日历、地图、浏览器、联系人管理程序等。所有的应用程序都使用Java语言编写。

2. 应用程序框架层

开发人员也可以完全访问核心应用程序所使用的API框架。该应用程序的架构设计简化了组件的重用，任何一个应用程序都可以发布它的功能块并且任何其他的应用程序都可以使用其所发布的功能块(须遵循框架的安全性)。同样，该应用程序重用机制也使用户可以方便地替换程序组件。

隐藏在每个应用程序后面的是一系列的服务和系统，其中包括以下内容。

(1) 丰富而又可扩展的视图系统(View System)，可以用来构建应用程序，它包括列表(Lists)、网格(Grids)、文本框(Textboxes)、按钮(Buttons)，以及可嵌入的Web浏览器。

(2) 内容提供器(Content Provider)使应用程序可以访问另一个应用程序的数据(如联系人数据库),或者共享它们自己的数据。

(3) 资源管理器(Resource Manager)提供非代码资源的访问,如本地字符串、图形和布局文件(Layout Files)。

(4) 通知管理器(Notification Manager)使应用程序可以在状态栏中显示自定义的提示信息。

(5) 活动管理器(Activity Manager)用来管理应用程序生命周期,并提供常用的导航回退功能。

3. Android 运行库层

Android 系统包含一些 C/C++ 库,这些库能被 Android 系统中不同的组件使用。它们通过 Android 应用程序框架层为开发者提供服务。一些核心库如下。

(1) 系统 C 库。一个从 BSD 继承来的标准 C 系统函数库 libc,它是专门为基于 Embedded Linux 系统的设备定制的。

(2) 媒体库。基于 OpenCore(PacketVideo),该库支持多种常用的音频、视频格式回放和录制,同时支持静态图像文件。编码格式包括 MPEG4、H.264、MP3、AAC、AMR、JPG、PNG 等。

(3) SurfaceManager。对显示子系统的管理,并且为多个应用程序提供了 2D 和 3D 图层的无缝融合。

(4) LibWebCore。一个最新的 Web 浏览器引擎,支持 Android 浏览器和一个可嵌入的 Web 视图。

4. Linux 内核层

Android 系统运行于 Linux Kernel 之上,但并不是 GNU/Linux 系统。因为在一般 GNU/Linux 系统里支持的功能,Android 系统大都没有支持,包括 Cairo、X11、ALSA、FFmpeg、GTK、Pango 及 glibc 等都被移除。Android 系统又以 Bionic 取代 glibc,以 Skia 取代 Cairo,再以 OpenCore 取代 FFmpeg,等等。Android 系统为了使应用可商用,必须移除被 GNU 和 GPL 授权证所约束的部分,例如 Android 系统将驱动程序移到 Userspace,使 Linux Driver 与 Linux Kernel 彻底分开。Bionic/Libc/Kernel/并非标准的 Kernel header files。Android 系统的 Kernel header 是利用工具由 Linux Kernel header 所产生的,这样做是为了保留常数、数据结构与宏。

Android 系统的 Linux Kernel 控制包括安全(Security)、存储器管理(Memory Management)、程序管理(Process Management)、网络堆栈(Network Stack)、驱动程序模型(Driver Model)等。下载 Android 源码前,先要安装其构建工具 Repo 来初始化源码。Repo 是 Android 系统用来辅助 Git 工作的一个工具。

3.1.3　操作系统安全现状

1. Android 系统的特点

1) 开放性

Android 平台首先就是其开放性,开发的平台允许任何移动终端厂商加入 Android

联盟中。显著的开放性可以使其拥有更多的开发者，随着用户和应用的日益丰富，一个崭新的平台也将很快走向成熟。

开放性对于 Android 系统的发展而言，有利于积累人气，这里的人气包括消费者和厂商，而对于消费者，最大的受益正是丰富的软件资源。开放的平台也会带来更大的竞争，如此一来，消费者将可以用更低的价位购得心仪的手机。

2）丰富的硬件

这与 Android 平台的开放性相关，由于 Android 系统的开放性，众多的厂商会推出千奇百怪、各具特色的多种产品。功能差异和特色却不会影响数据同步，甚至软件的兼容，如同从诺基亚 Symbian 风格手机一下改用苹果 iPhone，同时还可将 Symbian 中优秀的软件带到 iPhone 上使用，联系人等资料可以方便转移。

3）方便开发

Android 平台提供给第三方开发商一个十分宽泛、自由的环境，不会受到各种条条框框的阻扰，可想而知，会有多少新颖别致的软件诞生。但也有其两面性，如何控制血腥、暴力等方面的程序和游戏正是留给 Android 系统的难题之一。

4）Google 应用

从搜索巨人到全面的互联网渗透，Google 服务（如地图、邮件、搜索等）已经成为连接用户和互联网的重要纽带。Android 从 2005 年被 Google 公司收购，在互联网时代已经走过 10 多个春秋，无缝整合了这些优秀的 Google 服务。

2. Android 系统安全机制

Android 系统安全性方面的重要设计：在默认情况下，应用程序没有权限执行对其他应用程序、操作系统或用户有危害的操作，这些操作包括读写其他应用程序的文件等。Android 系统主要提供如下安全机制。

1）进程保护

程序只在自己的进程空间，与其他进程完全隔离，从而保证进程间安全。在同一个进程内部，可以任意切换到 Activity；在不同的进程间，如进程 A 的 Activity 去启动进程 B 中的 Activity，结果通常是请求被拒绝（Permission Denial）。

2）权限模型

Android 系统要求用户在使用 API 时进行申明，称为请求（Permission）。申明在 AndroidManifest.xml 文件里进行设置。这样对一些敏感 API 的使用在安装时就可以给用户风险提示，由用户确定是否安装。下面是一些最常用的权限许可。

（1）READ_CONTACTS：读用户通讯录数据。

（2）RECEIVE_SMS：监测是否收到短信。

（3）ACCESS_COARSE_LOCATION：通过基站或者 WiFi 获取位置信息。

（4）ACCESS_FINE_LOCATION：通过 GPS 获取到更精确的位置信息。

例如，要监控是否有短信到达，需要在 AndroidManifest.xml 文件中进行如下设置：

```
<manifestxmlns:android=http://schemas.android.com/apk/res/android
Package="com.google.android.app.myapp">
<uses-permissionandroid:name="android.permission.RECEIVE_SMS"/>
</manifest>
```

同时，Permission 通过 ProtectionLevel 分为 4 个保护等级：normal、dangerous、signature、signatureorsystem。不同的保护级别代表程序要使用此权限时的认证方式。normal 的权限只要申请就可以使用；dangerous 的权限在安装时需要用户确认才可以使用；signature 的权限可以让应用程序不弹出确认提示；signatureorsystem 的权限需要开发者的应用和系统使用同一个数字证书（其实就是需要系统或者平台签名）。dangerous 是最常用的权限，在平时安装应用时都会提示应用使用了哪些权限。

例如，在 Android 系统的 API 中提供 SystemClock.setCurrentTimeMillis 函数修改系统时间，可惜调用这个函数时发现，无论是模拟器还是真机，在 logcat 中总会出现“Unable to open alarm driver: Permission denied”。这个 API 其实就是 signatureorsystem 权限等级，即调用这个函数需要系统签名，但真实手机中的系统签名密钥只有厂商知道。

3）文件访问

Android 应用程序的安装目录分为以下 3 部分。

（1）/data/data 下会有每个应用程序的私有目录。

（2）/data/app 会保存所有安装文件的 APK（Android 系统的安装包）。

（3）/data/dalvik-cache 会有每个应用程序的核心文件 DEX（DEX 文件是 Android 系统的可执行文件，包含应用程序的全部操作指令及运行时的数据）的缓存文件，主要是为了提高效率。

（4）/System/app 通常是系统自带的应用程序目录。

每个 Android 应用程序（.apk 文件）会在安装时分配一个独有的 Linux 用户 ID，这就为它建立了一个沙盒，使其不能与其他应用程序进行接触（也不会让其他应用程序接触它）。这个用户 ID 会在安装时分配给它，并在该设备上一直保持同一个数值。所有存储在应用程序中的数据都会赋予一个属性——该应用程序的用户 ID，这使得其他安装包（Package）无法访问这些数据。当通过方法 getSharedPreference、openFileOutput 等创建一个新文件时，可以通过使用 MODE_WORLD_READABLE、MODE_WORLD_WRITEABLE 标志位设置是否允许其他 Package 访问读写这个文件。当设置这些标志位时，该文件仍然属于该应用程序，但是它的全局读写权限已经被设置，使得它对于其他任何应用程序都是可见的。下面的例子给出了 Android 文件存储的 4 种方式。

（1）Context.MODE_PRIVATE：默认操作模式，代表该文件是私有数据，只能被应用本身访问，在该模式下，写入的内容会覆盖原文件的内容。

（2）Context.MODE_APPEND：该模式会检查文件是否存在，存在就往文件追加内容，否则就创建新文件。

（3）Context.MODE_WORLD_READABLE：用来控制其他应用是否有权限读该文件，存在即表示当前文件可以被其他应用读取。

（4）Context.MODE_WORLD_WRITEABLE：用来控制其他应用是否有权限写该文件，存在即表示当前文件可以被其他应用写入。

当然也有方法可以使两个程序相互访问对方的资源，即使用 sharedUserId 属性。通过使用 AndroidManifest.xml 文件的 manifest 标签中的 sharedUserId 属性，使不同的 Package 共用同一个用户 ID。通过这种方式，这两个 Package 就可以进行资源的相互访问。但共用同一个用户 ID 需要两个应用程序可被同一个签名签署才能实现。

4）应用程序签名

Android 系统的代码签名采用自签名机制，是一种适度安全策略的体现，在某种程序上保证了软件的溯源目标及完整性保护。但程序签名只是为了声明该程序是由哪个公司或个人发布的，无须权威机构签名和审核，完全由用户自行判断是否信任该程序。API 按照功能划分为多个不同的能力集，应用程序要明确声明使用的能力。应用程序在安装时提示用户所使用到的能力，用户确认后安装。

APK 安装时的验证过程如下。

（1）计算 CERT.sf 文件的哈希值。

（2）用公钥（证书）验证 CERT.rsa 文件，得到结果与上面的 CERT.sf 文件的哈希值进行比较。如果相同，则表明 CERT.sf 文件是未被篡改的。

（3）由于 CERT.sf 文件包含了 APK 中 MANIFEST.MF 文件的哈希值，而 MANIFEST.MF 文件包含了 APK 中其他文件的哈希值，因此从 CERT.sf 文件可以得到其他文件的正确哈希值。

（4）最后验证 MANIFEST.MF 中列出的 APK 中的其他文件和其对应的哈希值是否一致，从而判断 APK 的完整性。

5）系统区和用户区分离

Android 系统的核心应用都部署在系统区（system/app），该区域是只读的。用户程序通常部署在 data/app。

6）密码学服务

Android 系统支持下列算法。

（1）对称算法，包括 AES、DES、3DES、RC5、RC2、PBE。

（2）非对称算法，包括 RSA、DSA、ECC。

（3）杂凑算法，包括 SHA1、MD5、MD2。

（4）传输加密。

Android 系统的传输加密支持 L2TP/IPSEC、L2TP、PPTPVPN，支持 SSLV 2.0\3.0\3.1。

3.2 App 漏洞

Android 系统由于其开源的属性，市场上针对开源代码定制的只读存储器（Read-Only Memory，ROM）参差不齐，在系统层面的安全防范和易损性都不一样，Android 系统应用市场对 App 的审核相对 iOS 系统也比较宽泛，为很多漏洞提供了可乘之机。市场上一些主流的 App 虽然做了一些安全防范，但由于大部分 App 不涉及资金安全，所以对安全的重视程度不够；同时由于安全是门系统学科，大部分 App 层的开发人员缺乏安全技术的积累，措施相对有限。

3.2.1 App 安全概述

移动市场成熟发展的同时，不法分子利用应用安全漏洞，致使安全事件频发，黑灰产业链也向移动终端用户蔓延，App 成为国家网络安全领域的重点对象。

1. App 发展现状

1）应用数量持续增加，用户增量饱和

据中华人民共和国工信部数据显示，2019 年年底我国 App 数量达 367 万。其中，游

戏类应用以占比 30.73%排行第一；生活服务类、电子商务类分别以占比 12.07%和 9.38%排名第二和第三。规模前三的应用类型总和占比为整个 App 市场规模的 52.18%，如图 3-5 所示。

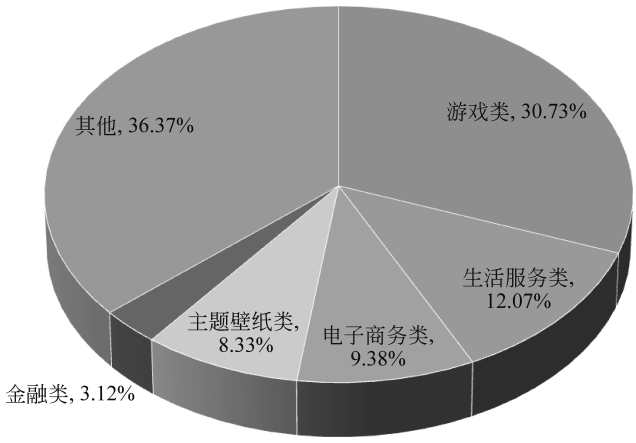

图 3-5　App 类型分布

截至 2019 年年底，我国移动互联网用户总数达 13.9 亿，我国手机上网用户数量达 12.6 亿，自 2019 下半年起维持稳定，市场用户增量基本饱和，如图 3-6 所示。

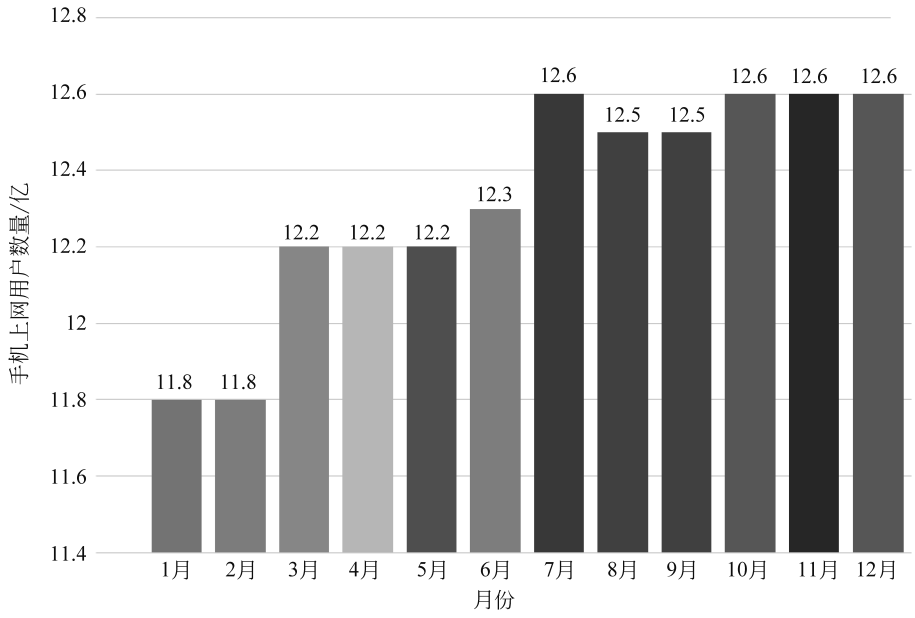

图 3-6　2019 年我国手机上网用户数量

2）App 漏洞大量催生，平均每个 App 存在 19 个漏洞

大量的用户基数及与日俱增的市场规模，为不法分子实施恶意攻击等行为提供了前提条件，权威数据表明，在 2019 年公开信息安全漏洞统计中，58%来自应用程序漏洞，而操作系统漏洞与数据库漏洞仅占 11%和 1%，应用程序漏洞大量催生。

Testin 云测安全实验室对 2019 年扫描的 573 652 款 App 分析后共计发现漏洞 10 794 512 个，平均每个 App 存在 19 个漏洞，仅有 0.3%的 App 不存在漏洞。其中，20%属于高危漏洞，39%属于中危漏洞，41%属于低危漏洞，如图 3-7 所示。

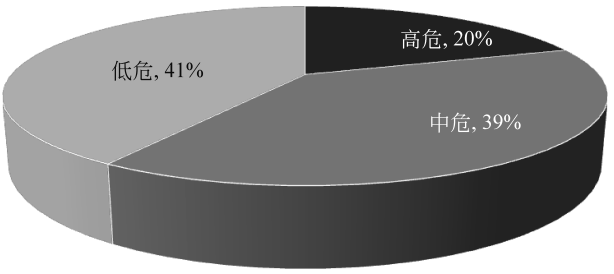

图 3-7　漏洞威胁等级分布

在应用风险类型分布中，由数据安全和代码安全因素引发的漏洞占比最多，所占比例分别为 30%和 28%。如图 3-8 所示。

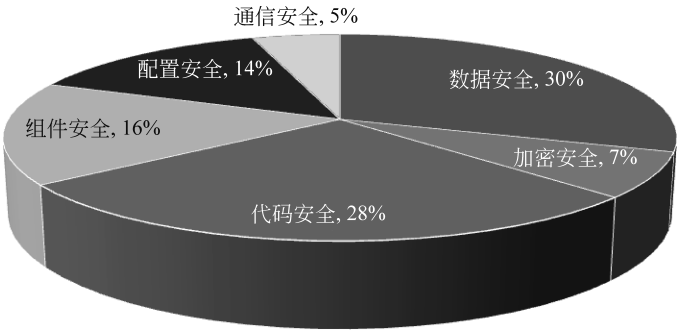

图 3-8　应用风险类型分布

3）数据保护监管愈发严格，用户信息安全意识觉醒

2019 年是信息泄露严重爆发的一年，如用户隐私信息在暗网公开售卖、上市公司窃取用户隐私牟利超千万、数十亿公民虹膜扫描和指纹信息外泄、新生婴儿信息非法倒卖等事件数见不鲜，信息泄露让数以亿计的用户毫无隐私可言且惶恐不安，也让不少企业深陷舆论泥潭，用户信息安全保护意识开始觉醒。

堪称史上最严格《通用数据保护条例》(General Data Protection Regulation，GDPR)的正式生效，定义了个人数据安全监管和保护新的要求高度，企业必须将用户个人的 IP 地址或 Cookie 数据等信息置于和其他机密数据(姓名、地址以及社会安全号码等)相同的保护等级。根据 GDPR 的要求，企业在获取用户资料时被要求使用简明语言，必须说明为什么要处理数据，谁将接收到这些数据，以及这些数据将被存储多长时间；一旦企业发生数据泄露事件，将被处以 4%的全球营业额或 2000 万欧元的罚款，同时在发现违规事件的 72 小时内，向监管当局和受到违规事件影响的个人通报数据违规行为。

而对于 App，可导致信息泄露的攻击面则在日益扩大。例如，使用不安全的通信协议，使用不安全的加密算法，应用提交数据时未对目标域名进行校验，无断网和网络异常提示等 App 漏洞是引发信息泄露的风险来源之一。

2. App 存在的共性安全问题

App 存在的共性安全问题主要包括以下 6 方面。

1）关键信息泄露

虽然 Java 代码一般要做混淆，但是 Android 系统的几大组件的创建方式是依赖注入的方式，因此不能被混淆，而且目前常用的一些反编译工具（如 ApkTool 等）能够毫不费劲地还原 Java 里的明文信息，Native 里的库信息也可以通过 objdump 或 IDA 获取。因此一旦 Java 或 Native 代码里存在明文敏感信息，基本上毫无安全可言。

2）App 重打包

App 重打包即反编译后重新加入恶意的代码逻辑，重新打包一个 APK 文件。重打包的目的一般都是和病毒结合，对正版 APK 进行解包，插入恶意病毒后重新打包并发布，因此伪装性很强。截住 App 重打包就一定程度上防止了病毒的传播。

3）进程被劫持

这个几乎是目前针对性最强的一种攻击方式，一般通过进程注入或者调试进程的方式 Hook 进程，改变程序运行的逻辑和顺序，获取程序运行的内存信息，即用户所有的行为都被监控起来，这也是盗取账号密码最常用的一种方式。

当然 Hook 行为不一定完全是恶意的，如有些安全软件会利用 Hook 的功能做主动防御（如最新的 APKProtect 线上加固产品）。一般来说，Hook 需要获取 root 权限或者与被 Hook 进程相同的权限，因此如果手机没有被 root，而且是正版 APK，被注入还是很困难的。

4）数据在传输过程中遭劫持

传输过程最常见的劫持就是中间人攻击。很多安全要求较高的 App 的所有的业务请求都要通过 HTTPS，但是 HTTPS 的中间人攻击逐渐增多，并且在实际使用中，证书交换和验证在一些非主流手机或者 ROM 上存在一些问题，让 HTTPS 的使用受到阻碍。

5）键盘输入安全隐患

支付密码一般是通过键盘输入的，键盘输入安全直接影响了密码安全。键盘输入安全隐患来自以下 3 方面。

（1）使用第三方输入法，则所有的点击事件在技术上都可以被第三方输入法截取，如果不小心使用了不合法的输入法，或者输入法把采集的信息上传并且泄露，后果是不堪设想的。

（2）截屏，该方法需要手机具有 root 权限，才能跑起截屏软件 getevent，通过读取系统驱动层 dev/input/event1 中的信息，获取手机触屏的位置坐标，再结合键盘的布局，就能算出事件与具体数字的映射关系，这也是目前比较常用的攻击方式。编者之前做过一套安全键盘的方案，就是“自定义键盘＋数字”布局随机化。但是随机化的键盘很不符合人性的操作习惯。所以之后去除了随机化。还需要说明，有一种更为安全的方式就是现在 TrustZone 的标准已经有 GlobalPlatform_Trusted_User_Interface，即在 TrustZone 里实现安全界面的一套标准，如果安全键盘在 TrustZone 里弹出，则黑客无论通过什么手段都无法拿到密码，是目前最为安全的方式，但是 TrustZone 依赖设备底层实现，如果设备不支持 TrustZone，或者 TrustZone 不支持 GlobalPlatform_Trusted_ User_Interface 标准，这种方式也是无能为力的。

6）WebView 漏洞

由于现在 Hybrid App 的盛行，WebView 在 App 的使用也是越来越多，Android 系统 WebView 存在一些漏洞，造成 JavaScript 提权。最为著名的就是传说中 JavaScript 注入漏洞和 WebKit 的 XSS 漏洞。

3. 个人信息保护面临的挑战

随着大数据时代的到来，个人信息保护问题逐渐暴露。信息泄露、信息过度收集使用、权限滥用等问题严重威胁了广大用户的切身利益。应用 API 等级低、一揽子授权、不授权不给用等现象的存在，将用户推入隐私与便利的两难选择。移动智能终端产业用户个人信息保护工作面临严峻的挑战。

1）用户信息过度收集

主要存在以下两种方式。

（1）App 在用户不知情的情况下，过度收集和使用个人信息。大量用户反映，个人信息在不知情的情况下被收集使用，搜索过的信息、说过的话、敏感的健康数据，以用户可感知的方式呈现。但信息是如何被收集，通过何种渠道共享传播？普通用户难识别、难举证。较多应用并未通过隐私政策或其他途径告知用户收集使用信息的目的、方式和范围，也未向用户提供明确的允许和拒绝的选择，这种累积性的权益侵害在日常生活中普遍存在，引发了用户的严重担忧。信息过度收集使用的乱象亟待解决。

（2）应用第三方软件开发工具包（Software Development Kit，SDK）大量收集使用个人信息。应用通常会使用第三方 SDK 快速实现业务功能，而第三方 SDK 与应用在收集用户信息方面具有同样的能力。鉴于第三方 SDK 的不开源性，应用无法完全掌控第三方 SDK 的行为。部分应用不清楚 SDK 申请权限的目的，难以准确明示第三方 SDK 所收集使用的用户信息，通常只能通过协议约束第三方 SDK 收集使用用户信息的行为。某些第三方 SDK 同时被多个 App 集成使用，收集的海量数据一旦泄露，将造成广泛的恶劣影响。

2）App 权限申请过度

权限是指为保护用户的隐私，移动终端操作系统对于应用访问敏感用户数据或使用特定系统功能的限制。为满足用户可知、可控的要求，我国终端大都具备权限管理机制，权限申请在显著位置提示，并经用户同意后方可使用。但目前权限申请过度仍是普遍现象。

（1）权限申请过度现象严重。根据对国内应用市场 Top1000 应用取样分析显示，Android 应用普遍会申请电话、定位、摄像头和录音等核心敏感权限，其中读取电话状态权限的比例为 97.37%，申请位置权限的比例为 84.15%，申请摄像头权限的比例为 66.8%，申请录音权限的比例为 59.1%，申请联系人权限的比例为 42.4%。应用过度申请权限的问题普遍存在。申请超出应用实际业务功能和场景的权限，为应用过度收集用户个人信息打开了通道，极易造成用户信息泄露。

（2）权限过度申请、滥用规范难判定。如何判定应用权限过度申请和滥用，存在易感知难判定的问题。目前尚缺乏成熟的技术规范和判定手段，难以正确引导应用开发者遵循合法正当必要原则申请权限，是智能终端产业在个人信息保护工作中面临的巨大的挑战。如图 3-9 所示。

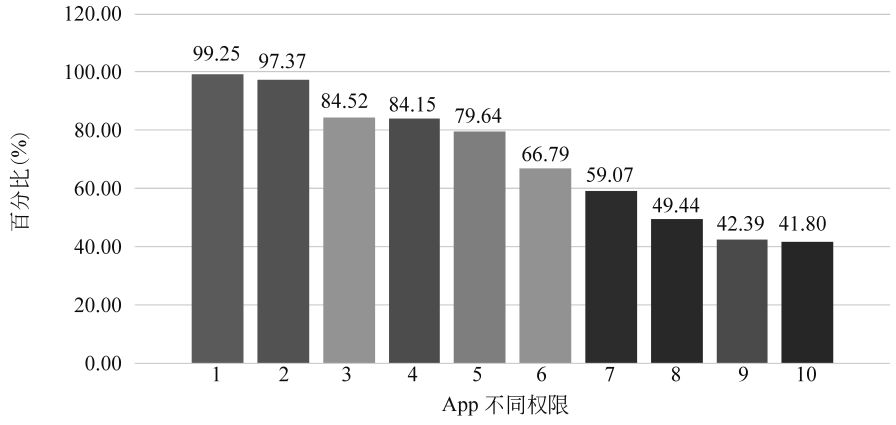

图 3-9　应用权限获取情况

3）低 API 等级应用规避 Android 系统安全机制

App 与 Android 系统的交互依赖框架 API，开发时要配置 App 的目标 API 等级以明确 App 支持的 Android 目标系统版本。低 API 等级 App 风险高、升级难度大。Android 系统在应用运行时检查目标 API 等级设置，若系统版本低于或等于 App 的目标 API 等级，系统无须进行任何兼容性处理；若系统版本高于此项配置，则系统会执行兼容性策略。低 API 等级应用运行在高版本的 Android 系统上，可绕过 Android 系统的信息保护机制。同时，Android 系统针对目标 API 等级 23 及以上的 App 执行运行时权限机制，即业务功能运行时系统才会授予 App 权限；目标 API 等级 23 以下的 App 采用一揽子授权，存在不授权无法安装使用的问题。目前，我国 App 达到目标 API 等级 26 及以上的比例大致为 10%，推动 App 开发者及时适配高版本 Android 系统，加强移动智能终端预置与分发环节对 App 高 API 等级的上架要求，是近期用户个人信息保护的重点工作。

4）常见的 Android 系统 App 漏洞

（1）AndroidManifest 配置相关的风险或漏洞。

① 程序可被任意调试。

风险详情：Android 系统 App APK 配置文件 AndroidManifest.xml 中的 android:debuggable=true，调试开关被打开。

危害情况：App 可以被调试。

修复建议：把 AndroidManifest.xml 配置文件中调试开关属性关掉，即设置 android:debugable="false"。

② 程序数据任意备份。

风险详情：Android 系统 App APK 配置文件 AndroidManifest.xml 中的 android:allowBackup=true，数据备份开关被打开。

危害情况：App 应用数据可被备份导出。

修复建议：把 AndroidManifest.xml 配置文件备份开关属性关掉，即设置 android:allowBackup="false"。

组件暴露：建议使用 android：protectionLevel＝"signature"验证调用来源。

③ Activity 组件暴露。

风险详情：Activity 组件的属性 exported 被设置为 true，或未设置 exported 值，但 IntentFilter 不为空时，Activity 被认为是导出的，可通过设置相应的 Intent 唤起 Activity。

危害情况：黑客可能构造恶意数据针对导出 Activity 组件实施越权攻击。

修复建议：如果组件不需要与其他 App 共享数据或交互，将 AndroidManifest.xml 配置文件中设置该组件为 exported＝"False"；如果组件需要与其他 App 共享数据或交互，对组件进行权限控制和参数校验。

④ Service 组件暴露。

风险详情：Service 组件的属性 exported 被设置为 true，或未设置 exported 值，但 IntentFilter 不为空时，Service 被认为是导出的，可通过设置相应的 Intent 唤起 Service。

危害情况：黑客可能构造恶意数据针对导出 Service 组件实施越权攻击。

修复建议：与 Activity 组件暴露修复建议相同。

⑤ ContentProvider 组件暴露。

风险详情：ContentProvider 组件的属性 exported 被设置为 true 或 Android API≤16 时，ContentProvider 被认为是导出的。

危害情况：黑客可能访问到 App 本身不想共享的数据或文件。

修复建议：与 Activity 组件暴露修复建议相同。

⑥ BroadcastReceiver 组件暴露。

风险详情：BroadcastReceiver 组件的属性 exported 被设置为 true 或未设置 exported 值，但 IntentFilter 不为空时，BroadcastReceiver 被认为是可导出的。

危害情况：导出的 BroadcastReceiver 可以导致数据泄露或者是越权。

修复建议：与 Activity 组件暴露修复建议相同。

⑦ Intent Scheme URLs 攻击。

风险详情：在 AndroidManifast.xml 设置 Scheme 协议后，可以通过浏览器打开对应的 Activity。

危害情况：攻击者通过访问浏览器构造 Intent 语法唤起 App 相应组件，轻则引起拒绝服务，重则可能演变对 App 进行越权调用甚至升级为提权漏洞。

修复建议：App 对外部调用过程和传输数据进行安全检查或检验，配置 category filter，添加 android.intent.category.BROWSABLE 方式规避风险。

(2) WebView 组件及与服务器通信相关的风险或漏洞

① WebView 存在本地 Java 接口。

风险详情：Android 系统的 WebView 组件有一个非常特殊的接口 addJavaScriptInterface，能实现本地 Java 与 JavaScript 之间交互。

危害情况：targetSdkVersion 使用低于 17 的版本时，攻击者利用 addJavaScriptInterface 这个接口添加的函数，可以远程执行任意代码。

修复建议：建议开发者不要使用 addJavaScriptInterface，使用注入 JavaScript 和第三方协议的替代方案。

② WebView 组件远程代码执行(调用 getClassLoader)。

风险详情：targetSdkVersion 使用低于 17 的版本，并且在 Context 子类中使用 addJavaScriptInterface 绑定 this 对象。

危害情况：通过调用 getClassLoader 可以绕过 Google 底层对 getClass 方法的限制。

修复建议：targetSdkVersion 使用高于 17 的版本。

③ WebView 忽略 SSL 证书错误。

风险详情：WebView 调用 onReceivedSslError 方法时，直接执行 handler.proceed() 忽略该证书错误。

危害情况：忽略 SSL 证书错误可能引起中间人攻击。

修复建议：不要重写 onReceivedSslError 方法，或者对于 SSL 证书错误问题按照业务场景判断，避免造成数据明文传输情况。

④ WebView 启用访问文件数据。

风险详情：WebView 中使用 setAllowFileAccess(true)，App 可通过 WebView 访问私有目录下的文件数据。

危害情况：在 Android 系统中，mWebView.setAllowFileAccess(true)为默认设置。当 setAllowFileAccess(true)时，在 File 域下，可执行任意的 JavaScript 代码，如果绕过同源策略能够对私有目录文件进行访问，导致用户隐私泄露。

修复建议：使用 WebView.getSettings().setAllowFileAccess(false)禁止访问私有文件数据。

⑤ SSL 通信服务端检测信任任意证书。

风险详情：自定义 SSL X509TrustManager，重写 checkServerTrusted 方法，方法内不做任何服务端的证书校验。

危害情况：黑客可以使用中间人攻击获取加密内容。

修复建议：严格判断服务端和客户端证书校验，对于异常事件禁止 return 空或 null。

⑥ HTTPS 关闭主机名验证。

风险详情：构造 HttpClient 时，设置 HostnameVerifier 参数使用 ALLOW_ALL_HOSTNAME_VERIFIER 或空的 HostnameVerifier。

危害情况：关闭主机名校验可以导致黑客使用中间人攻击获取加密内容。

修复建议：App 在使用 SSL 时没有对证书的主机名进行校验，信任任意主机名下的合法的证书，导致加密通信可被还原成明文通信，加密传输遭到破坏。

⑦ SSL 通信客户端检测信任任意证书。

风险详情：自定义 SSL X509TrustManager，重写 checkClientTrusted 方法，方法内不做任何客户端的证书校验。

危害情况：黑客可以使用中间人攻击获取加密内容。

修复建议：严格判断服务端和客户端证书校验，对于异常事件禁止 return 空或 null。

⑧ 开放 Socket 端口。

风险详情：App 绑定端口进行监听，建立连接后可接收外部发送的数据。

危害情况：攻击者可构造恶意数据对端口进行测试，对于绑定了 IP 0.0.0.0 的 App 可发起远程攻击。

修复建议：如无必要，只绑定本地 IP 127.0.0.1，并且对接收的数据进行过滤、验证。

(3) 数据安全风险或漏洞

① SD 卡数据被第三方程序访问。

漏洞描述：发现调用 getExternalStorageDirectory，存储内容到 SD 卡可以被任意程序访问，存在安全隐患。

安全建议：建议存储敏感信息到程序私有目录，并对敏感数据加密。

② 全局可读漏洞。

风险详情：openFileOutput(String name，int mode)方法创建内部文件时，将文件设置了全局的可读权限 MODE_WORLD_READABLE。

危害情况：攻击者恶意读取文件内容，获取敏感信息。

修复建议：开发者确认该文件是否存储敏感数据，如存在相关数据，去掉文件全局可读属性。

③ 全局文件可写。

风险详情：openFileOutput(String name，int mode)方法创建内部文件时，将文件设置了全局的可写权限 MODE_WORLD_WRITEABLE。

危害情况：攻击者恶意写文件内容，破坏 App 的完整性。

修复建议：开发者确认该文件是否存储敏感数据，如存在相关数据，去掉文件全局可写属性。

④ 全局文件可读写。

风险详情：openFileOutput(String name，int mode)方法创建内部文件时，将文件设置了全局的可读写权限。

危害情况：攻击者恶意写文件内容，破坏 App 的完整性；或者攻击者恶意读取文件内容，获取敏感信息。

修复建议：开发者确认该文件是否存储敏感数据，如存在相关数据，去掉文件全局可读写属性。

(4) 私有文件泄露风险或漏洞

① 配置文件可读。

风险详情：使用 getSharedPreferences 打开文件时，第二个参数设置为 MODE_WORLD_READABLE。

危害情况：文件可以被其他应用读取，导致信息泄露。

修复建议：如果必须设置为全局可读模式供其他程序使用，需保证存储的数据是非隐私数据或加密后存储。

② 配置文件可写。

风险详情：使用 getSharedPreferences 打开文件时，第二个参数设置为 MODE_WORLD_WRITEABLE。

危害情况：文件可以被其他应用写入，导致文件内容被篡改、影响应用程序的正常运行或更严重的问题。

修复建议：使用 getSharedPreferences 时，第二个参数设置为 MODE_PRIVATE。

③ 配置文件可读写。

风险详情：使用 getSharedPreferences 打开文件时，第二个参数设置为 MODE_WORLD_READABLE|MODE_WORLD_WRITEABLE。

危害情况：文件可以被其他应用读取和写入，导致信息泄露、文件内容被篡改、影响应用程序的正常运行或更严重的问题。

修复建议：使用 getSharedPreferences 时，第二个参数设置为 MODE_PRIVATE。禁止使用 MODE_WORLD_READABLE | MODE_WORLD_WRITEABLE 模式。

④ AES 弱加密。

风险详情：在 AES 加密时，使用 AES/ECB/NoPadding|AES/ECB/PKCS5Padding 模式。

危害情况：ECB 是将文件分块后对文件块做同一加密，破解加密只需要针对一个文件块进行解密，降低了破解难度和文件安全性。

修复建议：禁止使用 AES 加密的 ECB 模式，显式指定加密算法为 CBC 或 CFB 模式，可带上 PKCS5Padding 填充。AES 密钥长度最少是 128 位，推荐使用 256 位。

⑤ 随机数不安全使用。

风险详情：调用 SecureRandom 类中的 setSeed 方法。

危害情况：生成的随机数具有确定性，存在被破解的可能性。

修复建议：用/dev/urandom 或/dev/random 初始化伪随机数生成器。

⑥ AES/DES 硬编码密钥。

风险详情：使用 AES 或 DES 加解密时，密钥在程序中采用硬编码。

危害情况：通过反编译获取密钥可以轻易解密 App 通信数据。

修复建议：密钥加密存储或变形后进行加解密运算，不要硬编码到代码中。

(5) 文件目录遍历类漏洞

① Provider 文件目录遍历。

风险详情：当 Provider 被导出且覆写了 openFile 方法时，没有对 Content Query URI 进行有效判断或过滤。

危害情况：攻击者可以利用 openFile 接口进行文件目录遍历以达到访问任意可读文件的目的。

修复建议：一般情况下无须覆写 openFile 方法，如果必要，对提交的参数进行“../”目录跳转符或其他安全校验。

② unzip 解压缩漏洞。

风险详情：解压 ZIP 文件，使用 getName 方法获取压缩文件名后未对名称进行校验。

危害情况：攻击者可构造恶意 ZIP 文件，被解压的文件将会进行目录跳转，被解压到其他目录，覆盖相应文件，导致任意代码执行。

修复建议：解压文件时，判断文件名是否有“../”特殊跳转符。

(6) 文件格式解析类漏洞

① FFmpeg 文件读取。

风险详情：使用了低版本的 FFmpeg 库进行视频解码。

危害情况：在 FFmpeg 的某些版本中可能存在本地文件读取漏洞，可以通过构造恶

意文件获取本地文件内容。

修复建议：升级 FFmpeg 库到最新版。

② Janus 漏洞。

漏洞详情：向原始的 App APK 的前部添加一个攻击的 classes.dex 文件(A 文件)，Android 系统在校验时计算了 A 文件的哈希值，并以"classes.dex"字符串作为 key 保存。然后 Android 系统计算原始的 classes.dex 文件(B 文件)，并再次以"classes.dex"字符串作为 key 保存，这次保存会覆盖掉 A 文件的哈希值，导致 Android 系统认为 APK 没有被修改，完成安装。APK 程序运行时，系统优先以先找到的 A 文件执行，忽略了 B 文件，导致漏洞的产生。

危害情况：该漏洞可以让攻击者绕过 Android 系统的 Signature Scheme v1 签名机制，直接对 App 进行篡改。由于 Android 系统的其他安全机制也是建立在签名和校验的基础上，因此该漏洞相当于绕过了 Android 系统的整个安全机制。

修复建议：禁止安装有多个同名 ZipEntry 类的 APK 文件。

(7) 内存堆栈类漏洞

① 未使用编译器堆栈保护技术。

风险详情：为了检测栈中的溢出，引入了 Stack Canaries 漏洞缓解技术。在所有函数调用发生时，向栈帧内压入一个额外的被称为 canary 的随机数，当栈中发生溢出时，canary 将被首先覆盖，之后才是 EBP 寄存器和返回地址。在函数返回前，系统将执行一个额外的安全验证操作，将栈帧中原先存放的 canary 和.data 中副本的值进行比较，如果两者不吻合，说明发生了栈溢出。

危害情况：不使用 Stack Canaries 栈保护技术，发生栈溢出时系统并不会对程序进行保护。

修复建议：使用 NDK 编译 so 库时，在 Android.mk 文件中添加“LOCAL_CFLAGS :=-Wall -O2 -U_FORTIFY_SOURCE-fstack-protector-all”。

② 未使用地址空间随机化技术。

风险详情：PIE 全称 Position Independent Executables，是一种地址空间随机化技术。当 so 库被加载时，在内存里的地址是随机分配的。

危害情况：不使用 PIE，将会使 shellcode 的执行难度降低，攻击成功率增加。

修复建议：NDK 编译 so 库时，加入“LOCAL_CFLAGS := -fpie -pie”开启对 PIE 的支持。

③ libupnp 栈溢出漏洞。

风险详情：使用了低于 1.6.18 版本的 libupnp 库文件。

危害情况：构造恶意数据包可造成缓冲区溢出，造成代码执行。

修复建议：升级 libupnp 库到 1.6.18 版本或以上。

(8) 动态类漏洞

① DEX 文件动态加载。

风险详情：使用 DexClassLoader 加载外部的 APK、JAR 或 DEX 文件，当外部文件的来源无法控制或被篡改时，无法保证加载的文件是否安全。

危害情况：加载恶意的 DEX 文件将会导致任意命令的执行。

修复建议：加载外部文件前，必须使用校验签名或 MD5 等方式确认外部文件的安全性。

② 动态注册广播。

风险详情：使用 registerReceiver 动态注册的广播在组件的生命周期里是默认导出的。

危害情况：导出的广播可以导致拒绝服务、数据泄露或越权调用。

修复建议：使用带权限检验的 registerReceiver API 进行动态广播的注册。

(9) 校验或限定不严导致的风险或漏洞。

① Fragment 注入。

风险详情：通过导出的 PreferenceActivity 的子类，没有正确处理 Intent 的 Extra 值。

危害情况：攻击者可绕过限制访问未授权的界面。

修复建议：当使用高于 targetSdk19 的版本时，强制实现了 isValidFragment 方法；当使用低于 targetSdk19 的版本时，在 PreferenceActivity 的子类中都要加入 isValidFragment。两种情况下在 isValidFragment 方法中进行 Fragment 名的合法性校验。

② 隐式意图调用。

风险详情：封装 Intent 时采用隐式设置，只设定 action 属性，未限定具体的接收对象，导致 Intent 可被其他应用获取并读取其中数据。

危害情况：Intent 隐式调用发送的意图可被第三方劫持，导致内部隐私数据泄露。

修复建议：可将隐式调用改为显式调用。

(10) 命令行调用类相关的风险或漏洞。

动态链接库中包含执行命令函数。

风险详情：在 Native 程序中，有时需要执行系统命令，在接收外部传入的参数执行命令时没有过滤或检验。

危害情况：攻击者传入任意命令，导致恶意命令的执行。

修复建议：对传入的参数进行严格的过滤。

3.2.2　行业应用安全

2020 年《网络安全法》实施已逾两年，GDPR 正式生效，信息安全早已上升至国家层面，早在 2018 年中国人民银行发布的《关于开展支付安全风险专项排查工作的通知》，将金融等重点行业应用安全要求推至新的高度。

1. 金融行业信息泄露隐患严重

2019 年第三季度，金融类 App 数量约为 15 万款，较 2019 年年初增幅超过 15%，同时金融行业由于其业务的特殊性及敏感性，也是我国信息安全重点关注的行业。

2019 年，Testin 云测安全实验室累计扫描金融类 App 46 273 款，共发现漏洞 1 102 160 个，平均每个金融类 App 存在 30 个漏洞。金融类 App 高危漏洞 Top10 如下。

(1) ContentProvider URI 用户敏感信息泄露。

(2) 不安全的 ZIP 文件解压。

(3) 服务端证书弱校验。

(4) 客户端 XML 外部实体注入。

（5）Intent Scheme URL 攻击。

（6）WebView 远程代码执行。

（7）不安全的内部存储文件权限。

（8）Fragment 注入。

（9）Janus 签名漏洞。

（10）不安全的 SharedPreference 文件权限。

如图 3-10 所示，高危漏洞中出现频率最高的漏洞为 ContentProvider URI 用户敏感信息泄露。该漏洞属于组件安全范畴，是指 App 在使用 ContentProvider 提供对外数据访问接口时，未设置合理权限，攻击者利用此漏洞盗取账户信息及账户资金，直接危及用户和企业的安全。Testin 云测安全实验室建议可通过为导出的 ContentProvider 组件设置合理的调用权限进行漏洞修复。

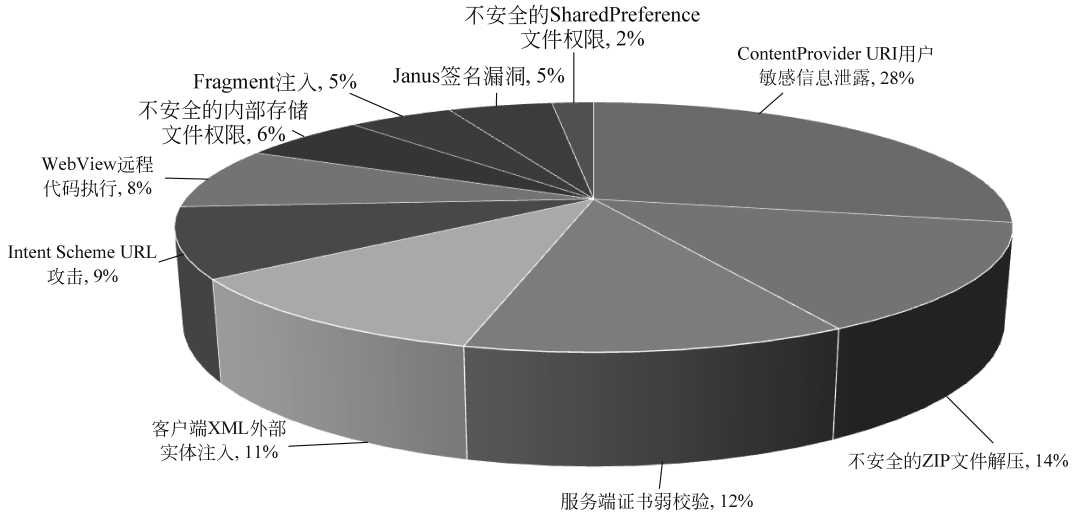

图 3-10　金融类 App 高危漏洞 Top10

高危漏洞中出现频率第二的漏洞为不安全的 ZIP 文件解压。该漏洞属于代码安全范畴，是指 App 在解压文件时使用 ZipEntry.getName 方法。该方法返回值里面会将路径原样返回。如果 ZIP 文件中包含路径字符串，同时没有进行防护，继续解压缩操作，就会将解压文件创建到其他目录中，覆盖掉敏感文件。造成敏感文件篡改，恶意代码执行等威胁。

2. 电商行业恶意攻击行为亟待防御

电商类 App 因涉及线上交易等业务且与用户账户资金密切相关，往往易成为黑灰产业的攻击对象，恶意刷券、虚假注册套取平台奖励等事件数见不鲜，一旦 App 潜在的漏洞隐患被非法利用，造成的损失难以估量。

2019 年，经由 Testin 云测安全实验室漏洞扫描引擎扫描的 96 456 款电商 App 中，共发现漏洞 3 071 250 个，其中高危漏洞 1 074 587 个，高危漏洞所占比例最高，高达 35%，平均每个电商应用存在 38 个漏洞。

高危漏洞中出现频率最高的漏洞为 Intent Scheme URL 攻击。该漏洞属于代码安全