Android Security Study

lijin
r1.1

August 5, 2013

1 Overview

Android希望自己是具有高安全性以及高可用性的移动操作系统，与传统的OS相比，它更注重以下方面:

- 保护用户的数据
- 保护系统资源(包括网络访问)
- 提供应用程序之间的隔离

为了达到以上功能，Android提供了以下的关键特性:

- 在OS层通过linux kernel提供了健壮的安全性
- 所有的应用程序必须运行在“沙盒”中
- 安全的进程间通信
- 应用签名
- 应用定义权限，由用户批准使用

1.1 System and Kernel Level Security

1.1.1 Linux Security

在OS层，Android平台提供的安全主要来自于linux kernel，同时提供了另外一套IPC机制来支持运行在不同的进程中的两个应用程序来互相通信。作为移动计算环境的基础，linux kernel提供给了android几个关键的安全特性:

- 基于用户的安全模型
- 进程隔离
- 可以给予安全扩展的IPC机制
- 可以去掉kernel中不必要以及可能存在潜在威胁的部分

作为一个多用户的OS，linux kernel的一个基本的安全目标是将一个用户的资源同另一个的分离。Linux的安全哲学就是保护用户的资源，所以linux确保:

- 用户A去读用户B的文件
- 用户A不能消耗用户B的内存
- 用户A不能耗尽用户B的CPU配额
- 用户A不能访问用户B的设备（如电话模块，GPS模块以及蓝牙模块等）

1.1.2 The Application Sandbox

Android充分利用了Linux基于用户的权限管理以及资源隔离。Android系统为每一个应用程序分配了一个唯一的用户ID（UID），且运行在一个独立的进程中。这与传统的OS中，多个程序以同一个用户身份去运行是截然不同的。通过这样的设置，android就建立好了内核级别的应用程序沙盒。通过传统的linux工具，如分配给应用的UID/GID，内核就在进程级保证了应用之间的安全性。默认情况下，应用程序之间不能通信且只有有限的资源。如果应用A试图去读取应用B的数据，或者想要在没有权限的情况之下去打电话（该功能一般以另外一个应用的方式实现），由于应用A缺乏相应的权限，OS将会正确且及时地的进行阻止。这样的沙盒是简单的，可审计的，且基于已经存在了数十年的unix风格的用户之间进程以及文件隔离。

如同所有的安全特性一样，这种应用程序沙盒并不是不能被破解的，但是想要在一个配置的比较好的设备上做到这些，攻击者就必须要先破解Linux kernel（而这又是很难的）。

1.1.3 System Partition and Safe Mode

system分区包括内核、系统库、应用程序运行时环境、应用程序框架以及一些应用程序。该分区是只读的，当用户以Safe Mode启动时，只有这部分的应用程序是可用的。于是我们就保证了用户可以启动到一个没有第三方程序的安全环境。

1.1.4 Filesystem Permissions

在一个unix风格的环境里，文件系统的权限保证了一个用户不能读也不能修改另一个用户的文件。又由于android环境里每一个应用都以自己的用户ID运行，所以除非开发人员显式地将文件开放给其他应用程序，否则一个应用程序创建的文件是不能被另一个应用去访问的。

1.1.5 Cryptography

android提供了一系列的加密相关的API给应用使用，包括AES、RSA、DSA以及SHA。同时还提供具有更高级别抽象的SSL以及HTTPS协议API。4.0系统还添进了KeyChain类支持将private keys 以及certificate chains和系统的credential保存在一起。

1.2 User Security Features

1.2.1 Filesystem Encryption

Android 3.0以后开始提供了整个文件系统的加密，所有的用户数据都在内核级别通过dmcrypt进行加密。加密Key是由用户密码通过AES128算法算出的，以防止未提供用户密码而去访问用户数据。详细实现见这里。

1.2.2 Password Protection

Android可以配置为开机需要密码的模式，该密码同样被用来推算完整的文件系统加密key。

1.2.3 Device Administration

Android 2.2 以及以后的版本提供了设备管理API，在系统级别提供了了设备管理功能。例如，内置的emali应用使用了一些API去改善Exchange支持。Exchange管理员可以通过email客户端去设置其他设备的密码策略（例如密码必须为字母或者数字等）。管理员也可以远程清除丢失的或者被偷的设备上的数据（通过恢复出厂设置）。

1.2.4 Credential Storage

默认情况下，android预先包含了一些信任的Certificate Authorities(CAs)，浏览器之类的应用程序可以用它来建立SSL链接。在4.0以后用户可以在设置里disable掉这些预装的CA，还可以通过USB导入受信任的CA以及证书。4.1以后还允许OEM厂商去添加基于硬件专用存储的KeyChain storage，这样就可以将private key与特定的设备绑定在一起了（用来标示该设备）。

1.2.5 Virtual Private Network

Android提供了内置的VPN客户端以支持PPTP、L2TP以及IPsec VPNs。从4.0开始还提供了VpnService去支持第三方的VPN解决方案。4.2开始允许用户将某个VPN设置为“always on”，以指示应用程序只有通过已连接的VPN去访问网络。

1.3 Android Application Security

1.3.1 Elements of Applications

OS的核心层基于Linux kernel，android的应用程序却通常使用Java语言编写并运行在Dalvik虚拟机之中。当然，应用程序也可以使用native code去编写。

- AndroidManifest.xml(*This specifies which permissions are required.*)
- Activities
- Services
- Broadcast Receiver

1.3.2 The Android Permission Model: Accessing Protected APIs

默认情况下android应用程序只能访问相当首先的系统资源，由系统程序来管理那些若不正确或恶意使用会严重影响用户体验、网络数据以及用户数据的功能。

这种受限是通过几种不同的形式实现的。一些敏感的功能故意没有API去访问，例如根本就没有直接访问SIM卡的API。另外一些情况下，对角色的区分提供了一种安全机制，如每一个应用程序的数据是隔离的。另外一些实例就是通过Permission机制将敏感的API只提供给受信任的应用使用。

这些受保护的API包括：

- Camera functions
- Location data (GPS)
- Bluetooth functions
- Telephony functions
- SMS/MMS functions
- Network/data connections

这些资源只能通过OS去访问。若应用程序想要使用这些受保护的API，它必须在它的manifest文件里面去申明。当应用程序被安装时，系统将会有一个提示框提示给用户该应用申请的所有权限。用户可以选择继续安装或者终止，一旦用户选择继续安装，系统就会将该应用申请的所有权限一次性授予给它，用户不能选择只授予其中几条permission。该权限一直伴随着该应用程序一直到被卸载，在运行时不会再次提示需要用户确认权限。系统的核心应用或OEM应用不会向用户提示权限申请，默认就有。当应用程序卸载后再次安装的时候会再次提示用户是否允许。

若应用尝试使用自己没有申请的权限，该调用将会抛出一个安全异常以终止访问。受保护的API权限检查一般都在尽量低的层次去检查。系统的默认权限描述在这里. 应用程序也可以自己申明权限给其他的应用程序去使用。当定义一个权限的时候protectionLevel属性将告诉系统如何去对待该权限，以及在申请的时候是否需要提示用户。详细细节在这里.

还有一些与设备相关的特权第三方的应用程序是不可能拿到的，比如发送SMS广播intent。这些权限只给预安装的应用使用，且使用signatureOrSystem级别。

1.3.3 Interprocess Communication

应用程序之间的通信可以使用任何一个传统的unix风格的机制，如filesystem、local sockets或者信号量等，且linux的权限机制仍然生效。

同时Android提供了一套新的IPC机制：

- Binder

  一套轻量级的`remote procedure call`，专门为`in-process`和`cross-process call`的高性能通信设计。

- Services

  在`binder`的基础之上，提供特殊的`API`

- Intent

  一个`Intent`作为一种简单的消息对象描述了一定的“意图”。例如若应用程序想要显示网页内容，则将该目的表达于`Intent`之内，系统将会为该意图去选择一个合适的应用程序响应。同时`Intent`还可以被用来在整个系统内发送广播。

1.3.4 Cost-Sensitive APIs

一个cost sensitive API指代可能产生一定费用的功能。android将这些API列为受保护的API由OS去管理，第三方的用户必须由用户显式地去授予权限。这些API包括:

- Telephony
- SMS/MMS
- Network/Data
- In-App Billing
- NFC Access

Android 4.2在此之上添加了一个消息通知，当应用程序尝试往一个short code上发消息（这可能导致额外的消费费用）时给用户以提示，用户可以选择是否允许该消息往外发送。

1.3.5 SIM Card Access

底层的对SIM卡的访问对第三方应用程序不可见。OS负责所有与SIM卡的交互，包括读取联系人信息等。应用程序也不能通过AT命令集对此进行访问，所有的此类访问都是由Radio Interface Layer(RIL)来管理，并不提供高层接口。

1.3.6 Application Signing

代码签名允许应用程序开发者去标示一个应用程序的作者，且允许在升级程序时给予少量的用户提示。每一个android应用程序必须要有签名，若没有签名会被Google Play的拒绝，且package installer也不会进行安装。

应用程序可以定义在签名级别上的安全保护，只有具有相同签名的应用才可以使用这些权限。此外还可以允许相同签名的多个应用共享UID，详见这里。

2 Lock & Root

2.1 SIM lock

SIM lock功能一般由手机厂商所内置，网络运营商利用这项功能来限制手机所能使用的地方以及所使用的网络。通常该功能通过只允许使用特定模式的IMSI(International Mobile Subscriber Identities)号码，该号码由以下部分组成:

- Mobile Country Code(MCC)
- Mobile Network Code(MNC)
- Mobile subscriber identification number (MSIN)

该方法一般由合约机使用，以确保手机不能用其他运行商的服务（也可以在一定的时间段后解锁）。

2.2 Lock Bootloader

锁Bootloader的方法不同的厂商不一样，同时有的厂商不锁而有的锁。锁住了的Bootloader的主要作用就是不允许用户随意刷第三方的ROM。所以一般都会在recovery系统里对升级包的签名作验证。

2.3 Root

Root与锁Bootloader不同，root的作用是让应用程序获得root权限，以此来绕过android对系统的各种安全限制，从而第三方应用程序可以完成原本不可能提供的功能。获得root权限的过程就是不断的寻找android系统的漏洞，尝试是否有方法获得root权限。

2.4 Root Privilege Request

2.4.1 SUID/SGID

suid的含义是让可执行文件执行时具有owner的权限，sgid的含义是让其具有owner所在group的权限。

2.4.2 Implementation

- 步骤一：appA先通过某一种方式获得root权限
- 步骤二：appA将系统的system分区重新挂为可写，并将自带的su替换掉原系统的su,且将该su的所有者改为root，并设置SUID/SGID位
- 步骤三：若appB想获得root权限，调用su程序，appA自带的su程序会检查之前appB是否已被允许拥有root权限，若被允许则按照正常请求调用setuid/setgid;若没有则调用appA的一个activity去提示用户是否允许appB获得root权限。

3 Coding Security

3.1 Kernel-Assistant Implementation

有一些权限由于其特殊性（不通过框架就可以访问，C/C++程序调用socket()函数），其实现通过linux的group来实现。

- 内核选项：CONFIG_ANDROID_PARANOID_NETWORK
- 相关权限：android.permission.INTERNET,android.permission.BLUETOOTH

举例：init.rc中的service段中可以用group字段，来标识该service所属的组：

```
service mtpd /system/bin/mtpd
class main
socket mtpd stream 600 system system
user vpn
group vpn net_admin inet net_raw
disabled
oneshot
```

core/init/init_parser.c

```
474 static void parse_line_service(struct parse_state *state, int nargs, char *

526  case K_group:
527      if (nargs < 2) {
528          parse_error(state, "group option requires a group id\n");
529      } else if (nargs > NR_SVC_SUPP_GIDS + 2) {
530          parse_error(state, "group option accepts at most %d supp. groups\n
531                      NR_SVC_SUPP_GIDS);
532      } else {
533          int n;
534          svc->gid = decode_uid(args[1]);
```

```
535          for (n = 2; n < nargs; n++) {
536              svc->supp_gids[n-2] = decode_uid(args[n]);
537          }
538          svc->nr_supp_gids = n - 2;
539      }
540      break;
```

```
 90 static const struct android_id_info android_ids[] = {
 91  { "root",      AID_ROOT, },
 92  { "system",    AID_SYSTEM, },
 93  { "radio",     AID_RADIO, },
 94  { "bluetooth", AID_BLUETOOTH, },
 95  { "graphics",  AID_GRAPHICS, },
 96  { "input",     AID_INPUT, },
 97  { "audio",     AID_AUDIO, },
 98  { "camera",    AID_CAMERA, },
 99  { "log",       AID_LOG, },
100  { "compass",   AID_COMPASS, },
101  { "mount",     AID_MOUNT, },
102  { "wifi",      AID_WIFI, },
103  { "dhcp",      AID_DHCP, },
104  { "adb",       AID_ADB, },
....
```

```
43 static unsigned int android_name_to_id(const char *name)
44 {
45     struct android_id_info *info = android_ids;
46     unsigned int n;
47
48     for (n = 0; n < android_id_count; n++) {
49         if (!strcmp(info[n].name, name))
50             return info[n].aid;
51     }
52
53     return -1U;
54 }
```

kernel/net/ipv4/af_inet.c

```
121 #ifdef CONFIG_ANDROID_PARANOID_NETWORK
122 #include <linux/android_aid.h>
123
124 static inline int current_has_network(void)
125 {
```

```
126     return in_egroup_p(AID_INET) || capable(CAP_NET_RAW);
127 }
128 #else
129 static inline int current_has_network(void)
130 {
131     return 1;
132 }
133 #endif

291 if (!current_has_network())
292     return -EACCES;
```

相关信息可以从启动程序时的输出看出来:

```
I/ActivityManager( 2142): Start proc com.routon.wiki for activity
com.routon.wiki/.WiKiMenuActivity: pid=4718 uid=10053 gids={3003}
```

```
ActivityManagerService.java
1950  int uid = app.info.uid;
1951  int[] gids = null;
1952  try {
1953      gids = mContext.getPackageManager().getPackageGids(
1954              app.info.packageName);
1955  }
```

3.2 Binder

3.2.1 Principle

binder实现的功能为IPC，但实际实现却更接近于RPC，包括以下几个部分:

- 服务提供者标识
- 服务提供者提供的服务的标识
- 数据的marshalling/unmarshalling
- 客户端请求路由

Binder标识

每一个binder实体都有两种表示:

- binder实体

  包括用户层的BnXXX以及驱动里的binder_node;

实体存在于提供服务的进程内，内核会相应的创建一个binder_node数据结构来表示，并将相关联的用户态指针保存在内核态，这样一个用户态的进程就可以同时开启多个服务，在对其中一个服务的请求到来时内核会返回相应的用户态指针值给用户态处理。

- binder引用

  包括在用户层的BpXXX以及驱动里的binder_ref;

  引用号是驱动为引用分配的一个32位标识，在一个进程内是唯一的，而在不同进程中可能会有同样的值，这和进程的打开文件号很类似。引用号将返回给应用程序，可以看作Binder引用在用户进程中的句柄。除了0号引用在所有进程里都固定保留给SMgr，其它值由驱动动态分配。向Binder发送数据包时，应用程序将引用号填入binder_transaction_data结构的target.handle域中表明该数据包的目的Binder。驱动根据该引用号在红黑树中找到引用的binder_ref结构，进而通过其node域知道目标Binder实体所在的进程及其它相关信息，实现数据包的路由。

  就象一个对象有很多指针一样，同一个Binder实体可能有很多引用，不同的是这些引用可能分布在不同的进程中。

```
Binder node to reference

1594  case BINDER_TYPE_BINDER:
1595  case BINDER_TYPE_WEAK_BINDER: {
1596      struct binder_ref *ref;
1597      struct binder_node *node = binder_get_node(proc, fp->binder);
1598      if (node == NULL) {
1599          node = binder_new_node(proc, fp->binder, fp->cookie);
1606      }

If ref doesn't exist, create one '

1615      ref = binder_get_ref_for_node(target_proc, node);
1620      if (fp->type == BINDER_TYPE_BINDER)
1621          fp->type = BINDER_TYPE_HANDLE;
1622      else
1623          fp->type = BINDER_TYPE_WEAK_HANDLE;
1624      fp->handle = ref->desc;
1625      binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE,
1626                  &thread->todo);


Handle/Reference to Node or new Ref
```

```
1633  case BINDER_TYPE_HANDLE:
1634  case BINDER_TYPE_WEAK_HANDLE: {
1635      struct binder_ref *ref = binder_get_ref(proc, fp->handle);
```

```
Call Service
```

```
1644      if (ref->node->proc == target_proc) {
1645          if (fp->type == BINDER_TYPE_HANDLE)
1646              fp->type = BINDER_TYPE_BINDER;
1647          else
1648              fp->type = BINDER_TYPE_WEAK_BINDER;
1649          fp->binder = ref->node->ptr;
1650          fp->cookie = ref->node->cookie;
1651          binder_inc_node(ref->node, fp->type == BINDER_TYPE_BINDER,
                                                              0, NULL);
1656      } else {
```

```
Pass binder to another client, mainly used by ServiceManager
```

```
1657          struct binder_ref *new_ref;
1658          new_ref = binder_get_ref_for_node(target_proc, ref->node);
1663          fp->handle = new_ref->desc;
1664          binder_inc_ref(new_ref, fp->type == BINDER_TYPE_HANDLE, NULL);
```

数据传送

Linux内核实际上没有从一个用户空间到另一个用户空间直接拷贝的函数，需要先用copy_from_user()拷贝到内核空间，再用copy_to_user()拷贝到另一个用户空间。

为了实现用户空间到用户空间的拷贝，mmap()分配的内存除了映射进了接收方进程里，还映射进了内核空间。所以调用copy_from_user()将数据拷贝进内核空间也相当于拷贝进了接收方的用户空间。

drivers/staging/android/binder.c

```
Reserve kernel address space
2812  area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
map kernel address space
658   tmp_area.addr = page_addr;
659   tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
660   page_array_ptr = page;
661   ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
map user address apce
670   ret = vm_insert_page(vma, user_page_addr, page[0]);
```

3.2.2 Two types of binder

实名binder vs. 匿名binder

- 在ServiceManager注册了名字的Binder叫实名Binder，所有用户都可以使用
- 匿名Binder为通信双方建立一条私密通道，只要Server没有把匿名Binder发给别的进程，别的进程就无法通过穷举或猜测等任何方式获得该Binder的引用，向该Binder发送请求。

3.2.3 UID & PID

添加UID/PID per transaction，用于鉴别调用者身份,且其他进程无法sniff其中的交换数据(相对于socket, system V:key)

binder驱动在传输数据的时候会对发送数据者的pid以及euid做个记录，并传递给接受者:

```
drivers/staging/android/binder.c
    1540  t->sender_euid = proc->tsk->cred->euid;

    2424      tr.sender_euid = t->sender_euid;
    2425
    2426      if (t->from) {
    2427          struct task_struct *sender = t->from->proc->tsk;
    2428          tr.sender_pid = task_tgid_nr_ns(sender,
    2429                                  current->nsproxy->pid_ns);
    2430      } else {
    2431          tr.sender_pid = 0;
    2432      }
```

接受者服务将请求者的身份信息记录:

```
libs/binder/IPCThreadState.cpp
    988   mCallingPid = tr.sender_pid;
    989   mCallingUid = tr.sender_euid;
```

在上层checkPermission的时候用到上面两个参数:

```
core/java/android/app/ContextImpl.java
    1261  public int checkCallingOrSelfPermission(String permission) {
    1262      if (permission == null) {
    1263          throw new IllegalArgumentException("permission is null");
    1264      }
    1265
    1266      return checkPermission(permission, Binder.getCallingPid(),
    1267              Binder.getCallingUid());
    1268  }
```

在ActivityManagerService里面会实际使用这两个信息来进行判断:

services/java/com/android/server/am/ActivityManagerService.java

```
4500  // Root, system server and our own process get to do everything.
4501  if (uid == 0 || uid == Process.SYSTEM_UID || pid == MY_PID) {
4502      return PackageManager.PERMISSION_GRANTED;
4503  }
4504  // If there is a uid that owns whatever is being accessed, it has
4505  // blanket access to it regardless of the permissions it requires.
4506  if (owningUid >= 0 && uid == owningUid) {
4507      return PackageManager.PERMISSION_GRANTED;
4508  }
4509  // If the target is not exported, then nobody else can get to it.
4510  if (!exported) {
4511      Slog.w(TAG, "Permission denied: checkComponentPermission() owningUid="
4512      return PackageManager.PERMISSION_DENIED;
4513  }
4514  if (permission == null) {
4515      return PackageManager.PERMISSION_GRANTED;
4516  }
4517  try {
4518      return AppGlobals.getPackageManager()
4519              .checkUidPermission(permission, uid);
4520  } catch (RemoteException e) {
4521      // Should never happen, but if it does... deny!
4522      Slog.e(TAG, "PackageManager is dead?!?", e);
4523  }
4524  return PackageManager.PERMISSION_DENIED;
```

3.2.4 Security Analysis

访问服务

- 访问服务必须要有该服务的引用，不管是匿名传送过来的还是从ServiceManager处得到的
- 若随意填写handle的数字，会被驱动进行相关检查该引用是否存在
- 若随意填写的Handle数字存在，也没法猜测该引用代表的是哪一个服务，而在服务方一般会验证客户端发送过来的标识服务的一个字符串
- 若枚举该字符串，何不直接通过service client的接口进行访问?

伪造UID

该标识是在驱动中添加的，没法伪造

3.3 Permission in General

Android的APK包在安装时会向Android系统申请权限，用户在安装APK时会看到此APK索要的权限，用户要么全都批准，要么拒绝本次安装。Android的权限与Android的API对应，一个APK如果获得了某项权限，这就意味着它在运行时可以调用Android的Java API，从而实现对Android受限资源进行访问的目的。

Android的权限主要包括三部分信息:

- 权限名称
- 属于的组
- 保护级别

保护级别分为四种:

- Normal
  只要申请了就可以使用，不需要提示用户
- Dangerous
  安装时需要用户确认才可以使用
- Signature
  需要使用者的app和申明权限的app的签名一致
- SignatureOrSystem
  需要使用者的app和系统使用同一个数字证书,且需要在系统分区中(/system)

3.3.1 Declare a permission

系统中的Permission由系统的APK（包括frameworks-res.apk）里的AndroidManifest.xml里定义。其定义格式如下:

```
<permission-tree android:icon="drawable resource"
     android:label="string resource" ]
     android:name="string" />
<permission-group android:description="string resource"
     android:icon="drawable resource"
     android:label="string resource"
     android:name="string" />
<permission android:description="string resource"
     android:icon="drawable resource"
     android:label="string resource"
     android:name="string"
```

```
android:permissionGroup="string"
android:protectionLevel=["normal" | "dangerous" |
"signature" | "signatureOrSystem"] />
```

3.3.2 Examples

core/res/AndroidManifest.xml

```
134     <permission-group android:name="android.permission-group.COST_MONEY"
135         android:label="@string/permgrouplab_costMoney"
136         android:description="@string/permgroupdesc_costMoney" />
137
138     <!-- Allows an application to send SMS messages. -->
139     <permission android:name="android.permission.SEND_SMS"
140         android:permissionGroup="android.permission-group.COST_MONEY"
141         android:protectionLevel="dangerous"
142         android:label="@string/permlab_sendSms"
143         android:description="@string/permdesc_sendSms" />

1225     <permission android:name="android.permission.INSTALL_PACKAGES"
1226         android:label="@string/permlab_installPackages"
1227         android:description="@string/permdesc_installPackages"
1228         android:protectionLevel="signatureOrSystem" />
1229
1230     <!-- Allows an application to clear user data -->
1231     <permission android:name="android.permission.CLEAR_APP_USER_DATA"
1232         android:label="@string/permlab_clearAppUserData"
1233         android:description="@string/permdesc_clearAppUserData"
1234         android:protectionLevel="signature" />
```

3.3.3 Permission to group binding

由于在申请有些权限的时候要对应到特定的用户组，所以系统上需要有这么一张对应表:

frameworks/base/data/etc/platform.xml

```
37  <permission name="android.permission.BLUETOOTH_ADMIN" >
38      <group gid="net_bt_admin" />
39  </permission>
40
41  <permission name="android.permission.BLUETOOTH" >
42      <group gid="net_bt" />
43  </permission>
44
45  <permission name="android.permission.INTERNET" >
```

```
46      <group gid="inet" />
47  </permission>
48
49  <permission name="android.permission.CAMERA" >
50      <group gid="camera" />
51  </permission>
52
53  <permission name="android.permission.READ_LOGS" >
54      <group gid="log" />
55  </permission>
56
57  <permission name="android.permission.WRITE_EXTERNAL_STORAGE" >
58      <group gid="sdcard_rw" />
59  </permission>
```

此外该文件还会将一些permission分配给shell这个特殊的UID:

```
<!-- Standard permissions granted to the shell. -->
110 <assign-permission name="android.permission.WRITE_EXTERNAL_STORAGE"
                                                        uid="shell" />
111 <assign-permission name="android.permission.SEND_SMS"
                                                uid="shell" />
112 <assign-permission name="android.permission.CALL_PHONE"
                                                uid="shell" />
113 <assign-permission name="android.permission.READ_CONTACTS"
                                                uid="shell" />
114 <assign-permission name="android.permission.WRITE_CONTACTS"
                                                uid="shell" />
115 <assign-permission name="android.permission.READ_CALENDAR"
                                                uid="shell" />
116 <assign-permission name="android.permission.WRITE_CALENDAR"
                                                uid="shell" />
117 <assign-permission name="android.permission.READ_USER_DICTIONARY"
                                                uid="shell" />
118 <assign-permission name="android.permission.WRITE_USER_DICTIONARY"
                                                uid="shell" />
119 <assign-permission name="android.permission.ACCESS_FINE_LOCATION"
                                                uid="shell" />
120 <assign-permission name="android.permission.ACCESS_COARSE_LOCATION"
                                                uid="shell" />
121 <assign-permission name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS"
                                                uid="shell" />
122 <assign-permission name="android.permission.ACCESS_NETWORK_STATE"
                                                uid="shell" />
```

```
123 <assign-permission name="android.permission.ACCESS_WIFI_STATE"
                                                uid="shell" />
124 <assign-permission name="android.permission.BLUETOOTH"
                                                uid="shell" />
125 <!-- System tool permissions granted to the shell. -->
126 <assign-permission name="android.permission.GET_TASKS"
                                                uid="shell" />
127 <assign-permission name="android.permission.CHANGE_CONFIGURATION"
                                                uid="shell" />
128 <assign-permission name="android.permission.REORDER_TASKS"
                                                uid="shell" />
129 <assign-permission name="android.permission.SET_ANIMATION_SCALE"
                                                uid="shell" />
130 <assign-permission name="android.permission.SET_PREFERRED_APPLICATIONS"
                                                uid="shell" />
131 <assign-permission name="android.permission.WRITE_SETTINGS"
                                                uid="shell" />
```

在系统的/system/etc/permission目录里保存着该文件，以及下文会提到的硬件feature文件。

读取该permission配置代码：

```
1259 void readPermissions() {
1260 // Read permissions from .../etc/permission directory.
1261 File libraryDir = new File(Environment.getRootDirectory(),
                        "etc/permissions");
1262 if (!libraryDir.exists() || !libraryDir.isDirectory()) {
1263     Slog.w(TAG, "No directory " + libraryDir + ", skipping");
1264     return;
1265 }
1266 if (!libraryDir.canRead()) {
1267     Slog.w(TAG, "Directory " + libraryDir + " cannot be read");
1268     return;
1269 }
1270
1271 // Iterate over the files in the directory and scan .xml files
1272 for (File f : libraryDir.listFiles()) {
1273     // We'll read platform.xml last
1274     if (f.getPath().endsWith("etc/permissions/platform.xml")) {
1275         continue;
1276     }
1277
1278     if (!f.getPath().endsWith(".xml")) {
1279         Slog.i(TAG, "Non-xml file " + f + " in " +
```

```
                                libraryDir + " directory, ignoring");
1280            continue;
1281        }
                ...
1286
1287        readPermissionsFromXml(f);
1288    }
1289
1290    // Read permissions from .../etc/permissions/platform.xml
                                last so it will take precedence
1291    final File permFile = new File(Environment.getRootDirectory(),
1292            "etc/permissions/platform.xml");
1293    readPermissionsFromXml(permFile);
1294    }
```

3.3.4 Check permission

PackageManagerService.java

```
1874 public int checkUidPermission(String permName, int uid) {
1875     synchronized (mPackages) {
1876         Object obj = mSettings.getUserIdLPr(uid);
1877         if (obj != null) {
1878             GrantedPermissions gp = (GrantedPermissions)obj;
1879             if (gp.grantedPermissions.contains(permName)) {
1880                 return PackageManager.PERMISSION_GRANTED;
1881             }
1882         } else {
1883             HashSet<String> perms = mSystemPermissions.get(uid);
1884             if (perms != null && perms.contains(permName)) {
1885                 return PackageManager.PERMISSION_GRANTED;
1886             }
1887         }
1888     }
1889     return PackageManager.PERMISSION_DENIED;
1890 }
```

3.3.5 Shared-UID

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="string"
    android:sharedUserId="string"
    android:sharedUserLabel="string resource"
    android:versionCode="integer"
    android:versionName="string"
```

```
    android:installLocation=
                ["auto" | "internalOnly" | "preferExternal"] >
    . . .
</manifest>
```

系统启动一个普通的APK时,会为这个APK分配一个独立的UID,这就是userId。
如果APK要和系统中其它APK使用相同的UID的话,那就是sharedUserId。
共享shareUserId的apk要求要有相同的签名。

代表一个共享UID,通常,共同实现一系列相似功能的APK共享一个UID。这个共享ID所拥有的权限是所有申请这个共享ID的APP所申请的权限的集合。所有使用的同一个共享UID的APK运行在同一进程中,这个进程的UID就是这个共享UID,这些APK都具有这个共享UID的权限。

Examples

```
20 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
21   package="android" coreApp="true"
                android:sharedUserId="android.uid.system"
22   android:sharedUserLabel="@string/android_system_label">
```

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2    package="com.android.systemui"
3    coreApp="true"
4    android:sharedUserId="android.uid.system"
5    android:process="system"
6    >
```

3.3.6 Permission Store

相关文件存储在/data/system目录下:

```
157  mSettingsFilename = new File(systemDir, "packages.xml");
158  mBackupSettingsFilename = new File(systemDir, "packages-backup.xml");
159  mPackageListFilename = new File(systemDir, "packages.list");
160  mStoppedPackagesFilename = new File(systemDir, "packages-stopped.xml");
161  mBackupStoppedPackagesFilename = new File(systemDir, "packages-stopped-bacl
```

- packages.xml 文件中记录了a)系统中所有的权限b)系统安装的所有apk的属性权限的信息,当系统中的apk 安装,删除或升级时,改文件就会被更新c) share-user的相关信息。

```
<package name="com.routon.brow" codePath=
            "/data/app-private/com.routon.brow-1.apk"
            resourcePath="/data/app/com.routon.brow-1.zip"
            nativeLibraryPath="/data/data/com.routon.b
<sigs count="1">
```

```
<cert index="11" key="3082030d308201f5a003020102020437b3c353
                      300d06092a864886f70d01010b0500303731
                      0b3009060355040613025533110300e0603
                      55040a1307416e64726f69643116301406035504
</sigs>
<perms>
<item name="com.android.launcher.permission.INSTALL_SHORTCUT" />
<item name="android.permission.NFC" />
<item name="android.permission.USE_CREDENTIALS" />
<item name="android.permission.SET_WALLPAPER" />
<item name="android.permission.WRITE_EXTERNAL_STORAGE" />
<item name="android.permission.ACCESS_WIFI_STATE" />
<item name="android.permission.ACCESS_COARSE_LOCATION" />
<item name="android.permission.GET_ACCOUNTS" />
<item name="android.permission.WRITE_SYNC_SETTINGS" />
<item name="android.permission.WRITE_SETTINGS" />
<item name="android.permission.INTERNET" />
<item name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS" />
<item name="android.permission.READ_SYNC_SETTINGS" />
<item name="android.permission.ACCESS_FINE_LOCATION" />
<item name="android.permission.MANAGE_ACCOUNTS" />
<item name="android.permission.WAKE_LOCK" />
<item name="android.permission.ACCESS_NETWORK_STATE" />
<item name="com.android.browser.permission.READ_HISTORY_BOOKMARKS" />
</perms>
</package>

<shared-user name="android.media" userId="10004">
<sigs count="1">
<cert index="18" />
</sigs>
<perms>
<item name="android.permission.SEND_DOWNLOAD_COMPLETED_INTENTS" />
<item name="android.permission.RECEIVE_WAP_PUSH" />
<item name="android.permission.ACCESS_CACHE_FILESYSTEM" />
<item name="android.permission.WRITE_MEDIA_STORAGE" />
<item name="android.permission.WRITE_EXTERNAL_STORAGE" />
<item name="android.permission.MODIFY_NETWORK_ACCOUNTING" />
<item name="android.permission.ACCESS_MTP" />
<item name="android.permission.RECEIVE_BOOT_COMPLETED" />
<item name="android.permission.ACCESS_ALL_DOWNLOADS" />
<item name="android.permission.INTERNET" />
<item name="android.permission.WRITE_SETTINGS" />
<item name="android.permission.CONNECTIVITY_INTERNAL" />
```

```
<item name="android.permission.ACCESS_DOWNLOAD_MANAGER" />
<item name="android.permission.UPDATE_DEVICE_STATS" />
<item name="android.permission.ACCESS_NETWORK_STATE" />
<item name="android.permission.WAKE_LOCK" />
</perms>
</shared-user>
```

- packages.list的数据格式是:pkgName,userId,debugFlag,dataPath(包的数据路径),其包含的数据没有packages.xml全
- packages-stopped.xml记录处于停止状态的包的信息，主要包括包名、notLaunched状态等信息。

    注：从android 3.1开始package manager开始追踪处于stopped状态的应用，并提供了两个inteng的flag去指明intent是否应该激活这些应用（FLAG_INCLUDE_STOPPED_PACKAGES,FLAG_EXCLUDE_STOPPED_PACKAGES）。

    Launch controls

3.4 Application Signing

同一个开发者的多个程序尽可能使用同一个数字证书，这可以带来以下好处。

- 有利于程序升级，当新版程序和旧版程序的数字证书相同时，Android系统才会认为这两个程序是同一个程序的不同版本。如果新版程序和旧版程序的数字证书不相同，则Android系统认为他们是不同的程序，并产生冲突，会要求新程序更改包名。
- 有利于程序的模块化设计和开发。Android系统允许拥有同一个数字签名的程序运行在一个进程中，Android程序会将他们视为同一个程序。所以开发者可以将自己的程序分模块开发，而用户只需要在需要的时候下载适当的模块。
- 可以通过权限(permission)的方式在多个程序间共享数据和代码。Android提供了基于数字证书的权限赋予机制，应用程序可以和其他的程序共享概功能或者数据给那那些与自己拥有相同数字证书的程序。如果某个权限(permission)的protectionLevel是signature，则这个权限就只能授予那些跟该权限所在的包拥有同一个数字证书的程序。

在签名时，需要考虑数字证书的有效期:

- 数字证书的有效期要包含程序的预计生命周期，一旦数字证书失效，持有改数字证书的程序将不能正常升级。

- 如果多个程序使用同一个数字证书，则该数字证书的有效期要包含所有程序的预计生命周期。
- Android Market强制要求所有应用程序数字证书的有效期要持续到2033年10月22日以后。

Android系统不会安装运行任何一款未经数字签名的apk程序，无论是在模拟器上还是在实际的物理设备上。Android的开发工具(ADT插件和Ant)都可以协助开发者给apk程序签名，它们都有两种模式：调试模式(debug mode)和发布模式(release mode)。

在调试模式下，android的开发工具会在每次编译时使用调试用的数字证书给程序签名，开发者无须关心。当要发布程序时，开发者就需要使用自己的数字证书给apk包签名，可以有两种方法。

- 在命令行下使用JDK中的和Keytool(用于生成数字证书)和Jarsigner(用于使用数字证书签名)来给apk包签名。keytool -genkey -v -keystore android.keystore -alias android -keyalg RSA -validity 20000

  `-keystore ophone.keystore` 表示生成的证书，可以加上路径（默认在用户主目录下）；`-alias ophone` 表示证书的别名是`ophone`；`-keyalg RSA` 表示采用的`RSA`算法；`-validity 20000`表示证书的有效期是`20000`天。

- 使用ADT Export Wizard进行签名(如果没有数字证书可能需要生成数字证书)。

然后再用jarsigner或者ADT Export Wizard对应用程序进行签名。

3.5 Device Management

3.5.1 Feature

users-feature用来声明应用使用的硬件以及软件的feature。

```
<uses-feature
    android:name="string"
    android:required=["true" | "false"]
    android:glEsVersion="integer" />
```

例子:

```
<uses-feature android:name="android.hardware.bluetooth" />
<uses-feature android:name="android.hardware.camera" />
```

Android系统本身在安装APK前不去检查uses-feature节点，该节点一般由类似Google Play的服务检查。当应用声明了要使用某个feature且require=true的时候，若硬件设备不包含此feature时，Google Play会将此应用为该设备标记为不兼容。

有一个有趣的feature:

```
Television

android.hardware.type.television

The application is designed for a television user experience.
This feature defines "television" to be a typical living room
television experience: displayed on a big screen, where the user
is sitting far away and the dominant form of input is something
like a d-pad, and generally not through touch or a mouse/pointer-
device.
```

(其他的feature见http://developer.android.com/guide/topics/manifest/uses-feature-element.html)

当应用未申明uses-feature却使用了相关的permission时，会默认认为其使用了相关的feature。例如使用了BLUETOOTH_ADMIN可以暗含了使用android.hardware.bluetooth feature。

每一个可能的feature都在frameworks/base/data/etc/目录下，并在活动的feature对应的文件在/system/etc/permissions下，在PackageManagerService启动时读取。每添加一个硬件时，我们应该添加一个新的feature。

3.5.2 Permission

由于Feature不是给安卓系统用的，所以当应用程序需要访问对应的设备时，仍然使用标准的permission机制去获取访问权限。

3.6 Use permission

3.6.1 Activity

```
<activity android:allowTaskReparenting=["true" | "false"]
. . .
android:permission="string"
. . .
</activity>
```

3.6.2 Service

```
<service android:enabled=["true" | "false"]
. . .
android:permission="string"
. . .
</service>
```

3.6.3 Content Provider

声明一个provider

```
<provider android:authorities="list"
    android:enabled=["true" | "false"]
    android:exported=["true" | "false"]
    android:grantUriPermissions=["true" | "false"]
    android:icon="drawable resource"
    android:initOrder="integer"
    android:label="string resource"
    android:multiprocess=["true" | "false"]
    android:name="string"
    android:permission="string"
    android:process="string"
    android:readPermission="string"
    android:syncable=["true" | "false"]
    android:writePermission="string" >
. . .
</provider>
```

以上的申明可以包括以下字段:

```
<path-permission android:path="string"
android:pathPrefix="string"
android:pathPattern="string"
android:permission="string"
android:readPermission="string"
android:writePermission="string" />

<grant-uri-permission android:path="string"
android:pathPattern="string"
android:pathPrefix="string" />

<path-permission android:path="string"
android:pathPrefix="string"
android:pathPattern="string"
android:permission="string"
android:readPermission="string"
android:writePermission="string" />
```

说明

- 属性中的android:permission,android:readPermission,android:writePermission用来规定当访问这个provider的时候需要什么样的权限
- 属性中的android:grantUriPermissions字段可以为：true)本来没有权限的应用程序可以通过有权限的另一个程序给予权限来进行访问;

false)只有在<grant-uri-permission>中的元素可以通过grant permission的方式被访问

示例：

```
<provider android:name= “com.example.test.app1.MailProvider”
android:authorities= “com.example.test.app1.mailprovider”
android:readPermission= “com.example.perm.DB_READ”
android:writePermission= “com.example.perm.DB_WRITE” >
</provider>

<provider android:name= “com.example.test.app1.MailProvider”
android:authorities= “com.example.test.app1.mailprovider”
android:readPermission= “com.example.perm.DB_READ”
android:writePermission= “com.example.perm.DB_WRITE”
android:grantUriPermission= “true” >
</provider>

<provider android:name= “com.example.test.app1.MailProvider”
android:authorities= “com.example.test.app1.mailprovider”
android:readPermission= “com.example.perm.DB_READ”
android:writePermission= “com.example.perm.DB_WRITE” >
<grant-uri-permission android:path= “/attachments/” >
</provider>
```

- grant permission的含义：

  当application A需要使用Component B去访问Provider C，但是Component B并未添加Provider C的Read/Write permission，如果这个Provider设置了属性android:grantUriPermissions，那么就有办法使Component B访问C。通过设置A 启动B时的Inent属性FLAG_GRANT_READ_URI_PERMISSION and FLAG_GRANT_WRITE_URI_PERMISSION，即可授权B在启动后去访问C。

  一个典型的例子是当一个e-mail包含了一个附件时，mail应用程序就可以调用适当的浏览器来打开附件，即使这个浏览器没有查看所有内容提供器数据的权限。

```
uri = “content://com.example.test.provider1/attachments/42” ;
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
intent.setDataAndType(uri, “image/gif” );
startActivity(intent);
```

- 还有一种方法可以动态的授予权限：

```
uri = “content://com.example.test.provider1/attachments/42”;
Context.grantUriPermission(“com.example.test.app2”, uri,
                    intent.FLAG_GRANT_READ_URI_PERMISSION);
```

3.7 Install apk

- 当任意一个app想申请安装程序时，使用以下代码:

```
String apkName="getnumber.apk";
//安装文件apk路径
String fileName=Environment.getExternalStorageDirectory()+"/"+apkName;
//创建URI
Uri uri=Uri.fromFile(new File(fileName));
//创建Intent意图
Intent intent=new Intent(Intent.ACTION_VIEW);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);//启动新的activity
//设置Uri和类型
intent.setDataAndType(uri, "application/vnd.android.package-archive");
//执行安装
startActivity(intent);
```

  该Intent将由packages/apps/PackageInstaller/这个应用来响应。卸载应用的代码类似。PackageInstaller程序负责显示要安装的app的权限，以及要求用户确认或者否认安装。

- 从板子上安装

  需要android.permission.INSTALL_PACKAGES权限

  pm install xxx.apk

  注：可以在adb shell登上板子之后，su user_name切换后得到该user的shell之后进行测试。

- 通过adb安装

  adb安装使用shell权限:

  adb install xxx.apk

- 通过拷贝安装

  系统会检测/system/framework,/system/app,/data/app,/data/app-private,/vendor/app目录里文件的增加或者删除情况，所以可以通过向这几个目录拷贝apk进行安装。

3.8 Install package without prompting

3.8.1 Method 1

假设系统是我们自己做的，我们可以有某个拥有安装权限的代码段直接调用PackageManager来进行安装。

3.8.2 Method 2

修改PackageInstaller的代码，提供一条隐秘的通道来进行安装（比如响应一个特殊的Intent）。

3.9 Future

3.9.1 4.2

- Application Verification

  当应用程序被修改了以后再安装时能检测出来。

- installd hardening

  `installd`不再以`root`去运行，减少潜在的风险

- init script hardening

  应用了`O_NOFOLLOW`了语义，去掉符号链接相关的攻击

更多见这里

3.9.2 4.3

- Android sandbox reinforced with SELinux

  开始使用`SELinux`提供的强制访问控制`(MAC)`，当前仅使用在`permissive`模式，未来可能会改为`enforcing`(即使是`root`用户不能访问未经授权的对象)

  SELinux

- No setuid/setgid programs

  去掉了`setuid/setgid`的程序，直接废掉了以前的`root`机制。

- ADB Authentication

  从`4.2.2`开始，`adb`的访问要通过`RSA`的`key`，限制对`adb`的使用

- Restrict Setuid from Android Apps

  `/system`分区`mount`的时候添加了`nosuid`选项，以阻止`android`执行`setuid`的程序。

更多见这里

4 App & Money

4.1 credential & keychain

4.1.1 Encryption vs Signature

- Encryption

  加密是将数据资料加密，使得非法用户即使取得加密过的资料，也无法获取正确的资料内容，所以数据加密可以保护数据，防止监听攻击。其重点在于数据的安全性。

- Certificate

  身份认证是用来判断某个身份的真实性，确认身份后，系统才可以依不同的身份给予不同的权限。其重点在于用户的真实性。两者的侧重点是不同的。

4.1.2 Public vs Private Key

在现代密码体制中加密和解密是采用不同的密钥（公开密钥），也就是非对称密钥密码系统，每个通信方均需要两个密钥，即公钥和私钥，这两把密钥可以互为加解密。公钥是公开的，不需要保密，而私钥是由个人自己持有，并且必须妥善保管和注意保密。

公钥私钥的原则:

- 一个公钥对应一个私钥。
- 密钥对中，让大家都知道的是公钥，不告诉大家，只有自己知道的，是私钥。
- 如果用其中一个密钥加密数据，则只有对应的那个密钥才可以解密。
- 如果用其中一个密钥可以进行解密数据，则该数据必然是对应的那个密钥进行的加密。

非对称密钥密码的主要应用就是公钥加密和公钥认证，而公钥加密的过程和公钥认证的过程是不一样的。

基于公开密钥的加密过程

比如有两个用户Alice和Bob，Alice想把一段明文通过双钥加密的技术发送给Bob，Bob有一对公钥和私钥，那么加密解密的过程如下:

- Bob将他的公开密钥传送给Alice。
- Alice用Bob的公开密钥加密她的消息，然后传送给Bob。
- Bob用他的私人密钥解密Alice的消息。

基于公开密钥的认证过程

身份认证和加密就不同了，主要用户鉴别用户的真伪。这里我们只要能够鉴别一个用户的私钥是正确的，就可以鉴别这个用户的真伪。

还是Alice和Bob这两个用户，Alice想让Bob知道自己是真实的Alice，而不是假冒的，因此Alice只要使用公钥密码学对文件签名发送给Bob，Bob使用Alice的公钥对文件进行解密，如果可以解密成功，则证明Alice的私钥是正确的，因而就完成了对Alice的身份鉴别。整个身份认证的过程如下：

- Alice用她的私人密钥对文件加密，从而对文件签名。
- Alice将签名的文件传送给Bob。
- Bob用Alice的公钥解密文件，从而验证签名。

签名

还是上面这个例子，若Alice要给Bob写一封信：

- Alice在写完信后用一个HASH函数，生成信件的摘要（digest）
- Alice用私钥对摘要进行加密，生成了数字签名(signature)
- Alice将这个签名放在信件下面发给Bob
- Bob收到Alice的信件后，用Alice的公钥对签名进行解密，得到摘要
- Bob自己再对信件运行HASH函数生成摘要，若摘要相同，则说明信件未被修改过

证书

若Bob的电脑被crack掉了，cracker可以将系统中Alice的公钥替换为自己的公钥，从此就可以伪装自己为Alice给Bob写信了。

解决方法是Alice去找一个证书中心（certificate authority，简称CA），为公钥做认证。证书中心用自己的私钥，对鲍勃的公钥和一些相关信息一起加密，生成“数字证书”（Digital Certificate）。

以后Alice要给Bob写信在签名的同时附上数字证书。Bob在收到信后，使用CA的公钥解开证书，就可以得到Alice真实的公钥了。

4.1.3 Android Keystore

- X.509

  `android`使用的证书格式为`X.509`

- PKCS #12

  `PKCS #12`是将多个密码文件存为一个文件的容器。类似的有`JKS`，不过`android`只支持`PKCS #12`

KeyChain

由于证书需要在一些应用之间共享，从4.0（ICS）开始android提供了一套新的称为KeyChain的API来统一应用对系统的key store的访问，并且提供一种机制让用户选择是否给予应用对证书的访问。

此外，该API还允许应用安装X.509的证书以及PKCS#12的key store:

```
byte[] keystore = . . (read from a PKCS#12 keystore)

Intent installIntent = KeyChain.createInstallIntent();
installIntent.putExtra(KeyChain.EXTRA_PKCS12, keystore);
startActivityForResult(installIntent, INSTALL_KEYSTORE_CODE);
```

当key store配置好了之后，应用就可以请求使用certificate来与SSL server来鉴别了。当应用第一次请求的时候，将会有一个弹出框来让用户选择将哪一个certificate给应用使用，在用户给予了权限之后系统将会返回一个该证书的别名（alias），之后的访问都通过该别名。

```
KeyChain.choosePrivateKeyAlias(this,
  new KeyChainAliasCallback() {
      public void alias(String alias) {
      // Credential alias selected.  Remember the
                            alias selection for future use.
      if (alias != null) saveAlias(alias);
      }
  },
  // List of acceptable key types. null for any
  new String[] {"RSA", "DSA"},
  null,                          // issuer, null for any
  // host name of server requesting the cert, null if unavailable
  "internal.example.com",
  // port of server requesting the cert, -1 if unavailable
  443,
  // alias to preselect, null if unavailable
  null);
```

接下来就可以使用private key去签名了:

```
PrivateKey privateKey = KeyChain.getPrivateKey(context, savedAlias);
if (privateKey != null) {
    ...
    Signature signature = Signature.getInstance("SHA1withRSA");
    signature.initSign(privateKey);
    ...
}
```

Unifying Key Store Access in ICS

4.2 Disk Encryption

4.2.1 Storage Options

在需要存储数据时，有以下几个选择:

- Shared Preferences

  以`key-value`对的形式存储私有的“原始”`(booleans, floats, ints, longs and strings)`数据

- Internal Storage

  在设备的存储上存储复杂的数据。

- External Storage

  在外部存储上存储共享的数据。

- SQLite Databases

  在私有的数据库里存储结构化的数据。

- Network Connection

  通过网络存储数据。

这里要注意的是外部存储上的数据是共享的，大家都可以读，因为外部存储本就可以移出设备。具体使用方法见这里。

4.2.2 App Install Location

从API Level 8开始允许将应用安装到外部存储，可以通过manifest的属性android:installLocation 指定，有两个值a)preferExternal，优先安装在外部存储，但是当外部存储满时也会安装在内部存储b)auto，标明可以安装在外部存储，具体安装的地方由系统来选择c)internalOnly，必须安装在设备内置存储里。详细见这里

4.2.3 /data encryption

从3.0开始android支持对data分区的加密，其机制使用kernel的dm-crypt机制。由于dm-crypt机制作用于block层，而YAFFS直接作用于raw flash chip，所以基于yaffs的分区不能加密。内核首选的文件系统格式为ext4。第一版的加密为128 AES,CBC和ESSIV:SHA256,通过openssl库来实现。

为了支持对分区的加密，系统起来的时候先通过一个临时的data将系统运行，提示用户输入密码之后再将临时的framework关闭，继而挂载真正的data分区启动真正的framework。为了实现这些效果，init将系统中的服务分为了以下3类：

- core 用于启动以后再也不重新启动的服务
- main 当重新启动framework时需要重启的服务
- late_start 当真正的framework启动后才启动的服务

此外，对vold也添加了修改：

```
45 CommandListener::CommandListener() :
46                  FrameworkListener("vold", true) {
47     registerCmd(new DumpCmd());
48     registerCmd(new VolumeCmd());
49     registerCmd(new AsecCmd());
50     registerCmd(new ObbCmd());
51     registerCmd(new StorageCmd());
52     registerCmd(new XwarpCmd());
53     registerCmd(new CryptfsCmd());
54     registerCmd(new FstrimCmd());
55 }
```

具体的使用方法见这里。

4.3 App Security

假设有一个应用是需要付费的，当一个用户购买了一个应用了之后，是否可以转发给其他人使用呢?

4.3.1 Forward Locking

Forward Locking(Copy Protection)就是不允许用户将已经购买的东西转发给其他人用，在android上的实现是将一个包分为两个部分：

- 一个全局可读的部分，包含资源以及manifest，存放在/data/app
- 只有system可以读的部分，包含了可执行代码，存放在/data/app-private

```
drwxrwxrwx system   system   2013-08-01 10:22 app
drwxrwx--x system   system   2013-08-01 10:17 app-private
```

这种方式有一个问题：被root了的手机依然可以提取apk。

4.3.2 Encrypt Container

当应用安装到外部存储时，若使用原生的apk则可以任意被传播，所以需要加密。

/mnt/sdcard/.android_secure下面保存着加密了的apk（以.asec结尾），当需要用的时候需要通过vold解密之后挂载到/mnt/asec目录下：

```
dannoy@dannoy-hpc:/media/6334-3438$ ls .android_secure/
cn.wps.moffice-1.asec              com.flowerpig.lwp.circuit-2.asec
com.adobe.reader-1.asec            com.halfbrick.fruitninjahdB-1.asec
com.baidu.BaiduMap-1.asec          com.junefsh.game.talking3dcat-1.asec
com.chartcross.gpstestplus-1.asec  com.kingsoft-1.asec

shell@android:/ $ df
Filesystem             Size   Used   Free   Blksize
/mnt/secure/asec: Permission denied
/mnt/asec/com.kingsoft-1    19M    17M     2M   4096
/mnt/asec/com.tencent.mobileqq-1    17M    15M     1M   4096
/mnt/asec/cn.wps.moffice-1    14M    12M     1M   4096
/mnt/asec/com.baidu.BaiduMap-1    16M    14M     1M   4096
/mnt/asec/com.adobe.reader-1    16M    10M     5M   4096
/mnt/asec/com.tigerknows-2     7M     5M     1M   4096
/mnt/asec/com.youdao.note-1    14M    10M     3M   4096
/mnt/asec/com.tencent.mm-1    28M    26M     2M   4096
/mnt/asec/livio.pack.lang.en_US-1    11M     9M     1M   4096
/mnt/asec/com.qzone-1     8M     6M     1M   4096
/mnt/asec/com.UCMobile-1    18M    16M     1M   4096
```

有一点很奇怪，解密之后的apk实际上也是可以adb pull出来的（下面的pkg.apk）：

```
shell@android:/ $ ls /mnt/asec/com.UCMobile-1/ -l
dr-xr-xr-x system   root              2013-05-20 19:27 lib
-r-xr-xr-x system   root      8366119 2013-05-20 19:27 pkg.apk
```

这里有个解释：

```
Apps on SD card are stored in an encrypted container for
platform security purposes -- so that other applications
can not modify or corrupt them. When mounting the SD card,
```

```
these containers are mounted so that they can be accessed
in the same way as apps stored in internal storage.
```

也就是说还是不安全?

4.3.3 License Verification Library

为了解决前面几种方式仍然没有解决的Copy Protection，Google开发了另外一套东西—License Verification Library(LVL)，提供给Application Licensing使用。

该机制允许应用程序使用Google Play Licensing Server去检测当前机器的用户是否有响应的许可证。当应用需要进行此项检查时向Google Play的客户端去发起申请，然后由Google Play的客户端去向Server查询，并将结果返回给应用程序，应用程序根据返回的结果来执行接下来的行为是继续运行还是提示购买。

为方便应用开发者使用此项功能，Google开发了Google Market Licensing package。一次请求包括以下步骤:

- 应用程序提供：包名，用来验证服务器回应的nonce以及一个回调函数
- Google Play Client收集当前设备以及用户的相关信息，如Google账户名，IMSI等。然后以application的身份向服务器发起认证请求。
- Google Play Server通过在购买记录里查询Client发送过来的信息，返回license记录给Google Play Client，后者再将结果返回给发起请求的应用程序。

注:

- 只有当Google Play Client已安装且设备运行的android高于1.5（API Level 3）版本时才能使用
- 必须要有网络
- 对License检查结果的处理依赖于应用程序，licensing服务只提供检查的机制
- 当应用添加了licensing功能，且设备未安装Google Play Client的时候应当不影响应用程序的功能
- 即使是不需要收费的应用程序也可以使用该服务，从而去提供APK Expension Files的功能。

APK Expension Files

Google Play允许APK最大为50M，但总有些程序会需要很多其他的支持文件，为了方便这些软件的开发Google允许在发布应用的时候同时发布两个用来扩展程序的文件。

- 主扩展文件用来包括大部分的资源文件
- 补丁文件用来给主扩展文件打补丁，一般大小会小很多

以上两个文件都最大支持2G。由于大小原因，这些文件应当存储在外部存储中。

相关的信息见这里

4.3.4 In-App Billing

Gooel Play Service允许应用程序出售一些数字化的商品，包括两类：

- 可下载的文件：音视频，照片等
- 虚拟化的内容：游戏关卡，程序特性等

Google将这些内容分为两类：

- 标准商品：只收一次钱
- 可消耗的商品，需要重复购买，如游戏内的“血”、道具等

同LVL一样应用程序需要向Google Play Client发起请求，由Google Play Client来向服务器发起请求并返回结果。

安全考虑：

- 使用服务器来分发内容以保持内容更新
- 若将已购买的内容存储在本地的外部存储中，记得要加密，否则其他人都可以传播

相关信息见这里

5 Permissions

5.1 classes of permissions

5.1.1 Cost-Sensitive

- Telephony
- SMS/MMS
- Network/Data
- In-App Billing
- NFC Access

5.1.2 Info-Sensitive

- Contacts
- Photo
- Media

5.1.3 Lower-Sensitive

- Camera
- GPS
- Bluetooth

6 Links

- Android Security Overview
- Security Tips
- Manifest.permission
- Setuid
- <uses-feature>
- android安全机制
- 数字签名是什么
- 数据公钥加密和认证中的私钥公钥