中软国际公司内部技术规范

Android 移动应用安全开发规范

中软国际科技服务有限公司

修订声明

参考: 业界 Android 移动安全编码规范指南，华为 Android 移动安全编码规范等。

0.

日期	修订版本	修订描述	作者
2017-07-28	1.0	整理创建初稿	陈丽佳
2017-08-02	1.1	统一整理格式，并修正部分错误	陈丽佳

目录

概述

背景简介

本规范针对移动应用开发中的输入校验、组件安全、访问控制安全、越狱安全、认证安全和运行环境及平台安全等方面，描述可能导致安全漏洞或风险的常见编码错误。规范基于业界最佳实践，参考业界安全编码规范相关著作并总结了公司内部的编程实践。规范旨在减少组件访问权限控制不当、注入攻击、敏感信息泄露、不合法输入等安全问题的发生。

使用对象

本规范的读者及使用对象主要为使用 Android 平台的研发人员和测试人员等。

适用范围

该规范适用于基于 Android 平台的产品开发。除非有特别说明，所有示例代码是基于 Android 4.2 及以上版本。

术语定义

规则：编程时必须遵守的约定
说明：某个规则的具体解释
错误示例：违背某条规则的例子
正确示例：遵循某条规则的例子
例外情况：相应的规则不适用的场景
实施指导：针对如何实现/满足某条规则/建议，给出的具体操作指导
信任边界：位于信任边界之内的所有组件都是被系统本身直接控制的。所有来自不受控的外部系统的连接与数据，包括其他应用、服务端与外部系统，都应该被认为是不可信的，要先在边界处对其校验，才能允许它们进一步与本系统交互。

1. 组件通用要求

规则 1.1 必须显式设置私有组件的 exported 属性为 false

说明：只在应用程序内部使用的组件必须设置为非公开，以防受到其他应用程序的调用。若应用组件未设置 exported 属性：

- Android 4.2版本之前，exported属性默认值为true；
- Android 4.2及之后的版本，若应用组件未设置<intent-filter>标签，exported属性默认值为false，否则默认值为true。

因此，私有组件（不存在对外交互）应该在 AndroidManifest.xml 文件中显式设置该组件的 android:exported 属性值为 false，否则，可能会对外暴露内部接口，若被恶意应用利用，则会造成组件信息泄露或发起拒绝服务攻击。

需要注意的是，私有组件如果声明了<intent-filter>标签，即使显式设置 exported 属性为 false，也是存在组件信息泄露的风险。如图 1-1 所示，假设组件 A-1 是私有组件，应用程序 A 的某一组件根据组件 A-1 声明的行为 X 给组件 A-1 发送消息；若恶意应用程序 B 定义了公开组件 B-1，并声明了和组件 A-1 相同的行为 X，当应用 A 通过行为 X 发送敏感信息给组件 A-1 时，会给用户弹出消息接收选择框，用户在不知情的情况下可能会选择组件 B-1，导致恶意应用 B 接收到应用 A 发送的内部消息，造成信息泄露。

推荐做法是：

1. 显式设置私有组件的exported属性为false，且不要设置intent-filter标签；
2. 应用程序内部使用显式Intent访问私有组件。

错误示例：

```
<activity
    android:name="com.xcompany.PrivateActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="com.xcompany.action.EVENT_CHANGE" />
    </intent-filter>
</activity>
```

PrivateActivity 是一个私有组件，但是并未设置 android:exported 属性，并且声明了<intent-filter>标签，即任何指定 com.xcompany.action.EVENT_CHANGE 行为的外部应用都可以访问该组件。

正确示例：

```
<activity
    android:name="com.xcompany.PrivateActivity"
    android:label="@string/app_name"
    android:exported="false" >
</activity>
```

私有组件显示设置exported属性为false，即只能通过显式Intent进行访问。

规则 1.2 必须对对外交互的组件设置访问权限

说明：应用程序的组件不只是在应用程序内部使用，很多情况下需要与外部进行交互。一般有以下两种情况表明组件是对外交互的：

- 组件在AndroidManifest.xml文件中显式设置android:exported属性为true，则表明该组件允许对外交互的；
- 组件在AndroidManifest.xml文件中未设置android:exported属性，Android 4.2版本之前、

默认是对外交互的，Android 4.2及之后的版本，若声明了<intent-filter>标签，那么该组件也是对外交互的。

程序内组件一旦允许公开，则会存在跨信任边界数据交互，若被恶意应用程序利用，可能会导致组件信息泄露或拒绝服务攻击；因此，应用程序对外交互的组件必须在该组件标签（如<activity>）下，使用 android:permission 属性来设置权限。

需要注意的是,对于使用 pendingIntent 来启动外部组件的场景,由于 pendingIntent 无法设置访问权限,因此外部组件也不能设置访问权限，所以要求 pendingIntent 及外部组件中不能携带敏感数据;若需使用 pendingIntent 发送广播,则必须确保广播不涉及敏感数据或操作。

错误示例：

```
<activity
    android:name="com.xcompany.PublicActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="com.xcompany.action.EVENT_CHANGE" />
    </intent-filter>
</activity>
```

可以看出，任何指定 com.xcompany.action.EVENT_CHANGE 行为的外部应用就可以访问 PublicActivity 组件，恶意应用程序可能利用该组件获取敏感信息或发起拒绝服务攻击。

正确示例：

```
<permission
        android:name="com.xcompany.permission.ACCESS"
        android:protectionLevel="signature" >
</permission>
<application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.xcompany.PublicActivity"
            android:label="@string/app_name"
            android:permission="com.xcompany.permission.ACCESS" >
            <intent-filter>
                <action android:name="com.xcompany.action.EVENT_CHANGE" />
            </intent-filter>
        </activity>
</application>
```

在组件标签下设置 android:permission 属性，这样应用程序可以根据组件不同的安全策略，有针对性的设置访问权限，从而细粒度控制组件的访问权限，实现组件间权限隔离，更好的保护组件安全。

规则 1.3 禁止口令输入组件提供拷出功能

说明：口令输入组件必须禁用拷出功能，以防止从输入组件中通过拷出口令的方式来泄露用户口令，危害个人及应用的安全。可以通过禁用口令输入组件的长按键功能来禁止拷贝。

错误示例：

```
<EditText
        android:id="@+id/PWDEditText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:inputType="textPassword" />
```

EditText 口令输入组件只设置了 inputType 属性为 textPassword 来使口令信息非明文显示，但没有禁用拷出功能。

正确示例：

Android 3.0 及之后的版本：

```
editText.setCustomSelectionActionModeCallback(new ActionMode.Callback() {
        @Override
        public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
            return false;
        }
        @Override
        public void onDestroyActionMode(ActionMode mode) {
        }
        @Override
        public boolean onCreateActionMode(ActionMode mode, Menu menu) {
            return false;
        }
        @Override
        public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
            return false;
        }
});
```

Andorid 3.0 版本中增加了实现拷贝、粘贴等功能的自定义实现 API，具体响应事件由 ActionMode 定义，其中：

- onPrepareActionMode方法用于更新行为模式，行为更新返回true，否则返回false；
- onCreateActionMode方法会在ActionMode第一次创建时调用，return false表示允许创建该ActionMode，return false可以阻止该ActionMode的启用；
- onActionItemClicked方法用于处理ActionMode事件响应，响应事件处理返回true，阻止该响应则返回false。

上述代码中，通过设置以上几个方法返回值为 false，可以完全限制对该 EditText 的所有选择操作，只能用户输入口令。

需要注意的是，由于 EditText 在横屏编辑的时候会出现一个新的不同的编辑界面，这个界面里还是可以复制粘贴的，因此涉及口令输入组件的情况下，应取消横屏界面 UI。

Android 3.0 之前的版本：

只提供了长按事件来实现字符串拷贝功能，因此，可以通过设置 longClickable 属性值为 false 来禁用拷贝功能。

规则 1.4 必须对涉及现实或虚拟货币应用的口令输入框实现安全输入机制

说明：涉及到虚拟现实货币应用，在进行应用登录或支付操作时，用户口令或支付密码输入框若采用默认的键盘输入，其他恶意应用可以通过记录用户触摸屏幕的位置，来推测破解出用户输入口令信息。因此对于现实或虚拟货币应用的口令输入框不能使用默认的键盘输入，应采用安全的输入机制，如实现随机键盘布局。

推荐做法：

- 系统已实现安全输入机制情况下，应用可调用系统提供的安全机制：如EMUI的TUI

系统未实现安全输入机制情况下，需要借鉴业界最佳实践实现安全机制

2. Activity

规则 2.1 禁止对私有 Activity 设置 taskAffinity 属性

说明： Activity 的 taskAffinity 属性指明了其期望进入的 task。如果 Activity 没有显式指明 taskAffinity 属性值，那么默认就为 Application 指明的 taskAffinity 属性值（在 Application 没有指明 taskAffinity 属性值得情况下，其 taskAffinity 属性值就等于包名），在这种情况下，Application 中所有的 Activity 都是属于同一 task。
对于私有 Activity，若对 taskAffinity 属性设置不当，可能会导致分配与同一应用中的其他组件不同的 task，由于不同 task 间组件交互是采用 Binder 通信机制，存在共享内存的数据被其他恶意应用获取的可能。因此，对于私有 Activity，禁止指定 taskAffinity 属性。

错误示例：

```
<activity
    android:name="com.xcompany.InternalActivity"
    android:exported="false"
    android:taskAffinity="com.task.demo"
    android:theme="@android:style/Theme.NoTitleBar" >
</activity>
```

android:exported="*false*"表明这是一个私有 Activity；android:taskAffinity 属性将私有 Activity 放到一个新的 task 中，在不同 task 间传递 intent 消息时，intent 消息有可能被其他应用读取到。

正确示例：

```
<activity
    android:name="com.xcompany.InternalActivity"
    android:exported="false"
    android:theme="@android:style/Theme.NoTitleBar" >
</activity>
```

私有 Activity 不要设置 taskAffinity 属性。

规则 2.2 禁止对私有 Activity 设置 launchMode 属性

说明：Activity 的 launchMode 属性用于控制 task 和 Activity 实例的创建，Activity 有四种启动模式，见表 2-1。在不指定 Activity 的 launchMode 属性时，默认为 Standard 模式。

表 2-1 Activity 四种启动模式

启动模式	描述
standard	标准模式，每次调用startActivity()方法就会产生一个新的Activity实例，但不会创建新的task。
singleTop	如果已经有一个Activity实例位于task栈的顶部时，就不产生新的实例；如果不位于栈顶，就等同于standard模式。
singleTask	该模式下的Activity第一次启动时，会在一个新的task中产生这个Activity实例，以后每次调用都会使用已生成的Activity实例，不会再产生新的实例了。
singleInstance	启动时与singleTask一样，唯一的差别是，在此模式下Activity实例所处的task中，只能有该Activity实例，不能再有其他的Activity实例。

对于私有 Activity，若对 launchMode 属性设置为 singleTask 或 singleInstance，则可能会创建新的 task，task 间传递 intent 消息时，intent 消息有可能被其他应用读取到，造成信息泄露。因此，私有 Activity 禁止配置 launchMode 属性。

需要注意的是，由于 Intent 的 flag 标识也可以设置 launchMode 属性，因此，对于启动私有 Activity 的 Intent 设置 flag 时，也须禁止设置 launchMode 属性。

错误示例：

```
<activity
    android:name="com.xcompany.InternalActivity"
    android:exported="false"
    android:launchMode="singleInstance"
    android:theme="@android:style/Theme.NoTitleBar" >
</activity>
```

私有 Activity 设置 launchMode 属性值为 singleInstance，即该 Activity 所在的 task 中只有这一个 Activity 实例。在不同 task 间传递 intent 消息时，intent 消息有可能被其他应用读取到。

正确示例：

```
<activity
    android:name="com.xcompany.InternalActivity"
    android:exported="false"
    android:theme="@android:style/Theme.NoTitleBar" >
</activity>
```

私有组件显式设置 exported 属性为 false，且未设置 launchMode 属性。

例外情况：若不涉及敏感数据，且存在性能上的需求，可根据实际业务需要，设置私有 Activity 的 launchMode 属性值为 singleTop。

3. Broadcast

规则 3.1 必须对涉及敏感数据/操作的 Broadcast 进行访问权限控制

说明：广播（Broadcast）为组件间通信提供了一种消息传递机制，这些组件可能是属于同一应用程序、也可能是属于不同应用程序。不同应用程序间的广播（Broadcast）使用进程间通信机制，广播权限设置不当，会给程序或系统带来安全上的风险，如控制进程强制启动/退出或暴露用户敏感信息等。

对于涉及敏感数据/操作的 Broadcast：

- 广播不能让其他应用接收到，通过设置广播发送权限来限制谁有权接受广播消息，
- 只接收特定来源的广播，通过设置接收器权限来限制广播来源，避免被恶意的同样 ACTION的广播所干扰。

错误示例：

```
//Receiver tag
<receiver
       android:name=".ExampleReceiver"
       <intent-filter>
          <action android:name="com.xcompany.action.EXAMPLE" />
       </intent-filter>
</receiver>
```

Broadcast 的<receiver>标签中未声明 android:permission 属性来限制广播来源。

```
//Send Broadcast
Intent intent = new Intent("com.xcompany.action.EXAMPLE");
intent.putExtra(KEY, SENSITIVE_DATA);
sendBroadcast(intent);
```

发送携带敏感信息的广播也未指定 Receiver 权限，即任何声明拥有 com.xcompany.action.EXAMPLE 行为的 Receiver 都能接收此广播。

正确示例（谁有权接收我的广播）：

在这种情况下，可以在应用发广播时添加参数声明 Receiver 所需的权限。

首先，在 Broadcast 应用的 Androidmanifest.xml 中定义权限：

```
<permission
       android:name="com.xcompany.permission.TEST"
       android:protectionLevel="signature" >
</permission>
```

然后，在 Send Broadcast 时，将此权限参数传入：

```
Intent intent = new Intent("com.xcompany.action.EXAMPLE");
intent.putExtra(KEY, SENSITIVE_DATA);
sendBroadcast(intent,"com.xcompany.permission.TEST");
```

这里指定了 Receiver 权限，即只有声明了 com.xcompany.permission.TEST 权限的 Receiver 才能接收此广播。发送广播时指定 Receiver 权限，能够很好的防范广播劫持攻击；在广播应用中声明权限，能够降低广播仿冒攻击的风险。通常广播发送方是诉求提出的主体，因此，该做法也是最为推荐的。

同时在 Receiver 所在应用的 AndroidManifest.xml 中添加对应的权限：

```
<uses-permission android:name="com.xcompany.permission.TEST"/>
```

正确示例（谁有权给我发广播）：

在这种情况下，需要在 Receiver 应用的<receiver>标签中声明 android:permission 属性以及广播发送方所在应用具有的权限。

首先，在 Receiver 应用的 AndroidManifest.xml 中定义权限：

```
<permission
       android:name="com.xcompany.permission.TEST"
```

```
        android:protectionLevel="signature" >
</permission>
```

然后，在<receiver>标签中添加相应权限的声明：

```
<receiver
        android:name=".ExampleReceiver"
        android:permission="com.xcompany.permission.TEST" >
        <intent-filter>
            <action android:name="com.xcompany.action.EXAMPLE" />
        </intent-filter>
</receiver>
```

这样一来，该 Receiver 便只能接收来自具有 com.xcompany.permission.TEST 权限的应用发出的广播。

同时，在广播发送方所在应用的 AndroidManifest.xml 中声明使用该权限即可：

```
<uses-permission android:name="com.xcompany.permission.TEST"/>
```

规则 3.2 禁止使用 Sticky Broadcast 发送敏感数据

说明： sticky 类型的广播会在广播处理完成后继续保存上次广播的 intent。不管是广播发送前还是发送后注册的 Receiver，只要与 Sticky Broadcast 行为相同，立即就能接收到该广播。因此，若使用 Sticky Broadcast 发送敏感数据，容易导致敏感信息泄露。

错误示例：

```
Intent intent = new Intent("android.intent.action.sticky.broadcast");
intent.putExtra(KEY, SENSITIVE_DATA);
sendStickyBroadcast(intent);
```

任何注册 android.intent.action.sticky.broadcast 行为的 Receiver 都可以立即收到此广播消息。

正确示例（指定 Receiver 权限）：

禁止使用 Sticky Broadcast 发送敏感数据。可参考规则 1.3.1: 必须对涉及敏感数据/操作的 Broadcat 进行访问权限控制

首先，在 Broadcast 应用的 Androidmanifest.xml 中定义权限：

```
<permission
        android:name="com.xcompany.permission.TEST"
        android:protectionLevel="signature" >
</permission>
```

然后，在 Send Broadcast 时，将此权限参数传入：

```
Intent intent = new Intent("android.intent.action.broadcast");
intent.putExtra(KEY, SENSITIVE_DATA);
sendBroadcast(intent,"com.xcompany.permission.TEST");
```

这里使用 Broadcast 替代 Sticky Broadcast，并指定了 Receiver 权限，即只有具有 com.xcompany.permission.TEST 权限的 Receiver 才能接收此广播，降低了敏感信息泄露的风险。

规则 3.3 只在程序内部发送的广播须使用 local Broadcast

说明： Android 系统在 android-support-v4.jar 中引入了 Local Broadcast，用来在同一个应用内的不同组件间发送 Broadcast。

Local Broadcast 发送的广播不会离开广播所在的应用程序，因此可用来传播敏感信息；由于 Local Broadcast 不需要用到跨进程通信，因此相对 Broadcast 而言要更为高效；同时，使用 LocalBroadcastManager 注册的 Receiver 不会接收本应用外的广播，不用担心来自其他应用程序的破坏。

规则 3.4 Broadcast Receiver 须分开处理系统广播和应用广播

说明：Broadcast Receiver 注册时需要定义一个广播过滤器<intent-filter>，在<intent-filter>标签中声明需要接收的系统广播和应用广播。系统广播和应用广播分离处理可以降低两者之间的影响，否则，由于系统广播不会设置访问权限，因此应用广播也不能设置访问权限，从而导致应用广播对外暴露，造成信息泄露或危险操作。所以，应对系统广播和应用广播进行分别处理，以防止外部恶意软件进行广播仿冒或混淆攻击。

需要注意的是，若广播涉及敏感数据或操作，则须分别注册广播接收器来区分系统广播和应用广播，从业务逻辑上隔离开不同广播的处理过程。

4. Content Provider

规则 4.1 必须分别设置外部 Content Provider 的读写访问权限

说明：Content Provider 将本应用程序存储的数据以数据表的形式提供给访问者，以实现程序间数据的共享。如果 Content Provider 未进行访问权限控制，一旦被恶意应用程序访问，就会造成该 Content Provider 存储的数据可被任意读取和篡改；如果直接设置 Content Provider 的 android:permission 属性，则会默认将读写权限一起授予，违反公司权限最小化安全设计原则。因此，对外交互的 Content Provider 必须分别设置读写访问权限。

在使用 Content Provider 共享系统数据时，Content Provider 的权限不能低于访问系统数据所需要的权限。例如：应用 A 拥有访问系统某敏感数据的权限，并通过 Content Provider 共享，应用 B 没有访问该系统数据的权限，但是，若应用 B 拥有访问该 Content Provider 的权限，则可能导致系统数据不期望的泄露。推荐做法是：应用 B 通过应用 A 的 Content Provider 访问系统数据时，应用 A 中需校验应用 B 是否拥有访问该系统数据的权限。

错误示例：

```
<provider
      android:name="com.xcompany.ExampleProvider"
      android:authorities="com.xcompany.provider"/>
```

该示例未对访问权限进行控制， android 4.2 及以上版本 provider 的 exported 属性默认为 false, 在 android 4.2 以下版本中该组件就可以被其他应用任意访问。

正确示例（设置 provider 的读写权限）：

```
<manifest
      xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.xcompany"
      android:versionCode="1"
      android:versionName="1.0" >
      <uses-sdk android:minSdkVersion="8" />
      <permission android:name="com.xcompany.permission.READ"
                  android:protectionLevel="signature" />
      <permission android:name="com.xcompany.permission.WRITE"
                  android:protectionLevel="signature" />
      <application
         android:icon="@drawable/ic_launcher"
         android:label="@string/app_name" >
          <provider
             android:name="com.xcompany.ExampleProvider "
             android:authorities="com.xcompany.provider "
             android:exported="true"
             android:readPermission="com.xcompany.permission.READ"
             android:writePermission="com.xcompany.permission.WRITE" />
      </application>
</manifest>
```

Content Provider 分别设置了读权限 android:readPermission 和写权限 android:writePermission，从而细粒度控制组件的访问权限，实现权限最小化，更好的保护组件安全。

正确示例（设置 URI 的读写权限）：

```
<provider
      android:name="com.xcompany.ExampleProvider"
      android:authorities="com.xcompany.provider "
      android:exported="true"
```

```
        android:readPermission=" com.xcompany.MY_PERMISSION ">
        <path-permission
            android:pathPrefix="/hello"
            android:readPermission="com.xcompany.HELLO_PERMISSION" />
</provider>
```

可以针对部分 URI 单独进行权限设置，除了设置 android:pathPrefix 属性，也可以通过设置 android:path 或 android:pathPattern 属性来限制 URI 访问。

其他应用程序如果需要读取 hello 目录下的数据就需要申请如下权限：

```
<uses-permission android:name="com.xcompany.MY_PERMISSION"/>
<uses-permission android:name="com.xcompany.HELLO_PERMISSION"/>
```

需要特别注意的是，如果<provider>中没有设置访问权限，只在<path-permission>中设置了权限属性，那么在 android 2.3.3 版本中，path-permission 设置的权限是不会生效的。

例外情况：对于不涉及敏感数据的 Content Provider，可以根据业务实际情况和安全风险评估不设置访问权限。

规则 4.2 禁止使用不可信数据直接拼接 Content Provider 查询语句

说明：Content Provider组件提供了一组数据操作函数：query()，insert()，update()和delete()。通常，content provider是以SQL数据库的形式来管理数据并实现共享，对于跨信任边界输入的数据未经处理，直接用于拼接SQL语句，攻击者可以通过构造特殊形式的输入来改变程序中原本要执行的SQL逻辑，形成SQL注入攻击。

另外，content provider提供了URI来指定对外共享的数据地址，外部调用者也是根据该标识来访问数据。URI主要有scheme、authority和path三部分组成，其中path表示我们要操作的数据库，一般scheme、authority在程序中已经规定好，若攻击者可以输入path参数，则可能会改变URI路径，导致URI注入攻击。

错误示例（URI 注入）：

```
public void getLastName(String input)
{
      Uri dataUri = Uri.parse(WeatherContentProvider.CONTENT_URI + "/" + input);
      mCursor = mResolver.query(dataUri, null, null, null,null);
}
```

Content Provider 允许用户使用 Uri 来查询数据库，如果 Uri 从不可信环境传入，就存在注入的风险。

正确示例：

```
public void getLastName(String input)
{
       if (!input.equals("history") && !input.equals("picture"))
      {
          return;
      }
      Uri dataUri = Uri.parse(WeatherContentProvider.CONTENT_URI + "/" + input);
      mCursor = mResolver.query(dataUri, null, null, null,null);
}
```

使用硬编码的 Uri 或者在拼装 Uri 的时候对用户输入进行白名单校验可以防止攻击者利用 Uri 对 Content Provider 进行注入攻击。

错误示例（SQL 注入）：

```
public void getLastName(String input)
{
       String selection = "userID = " + input;
       String[] selectionArgs = {""};
       mCursor =
mResolver.query(mUri,mProjection,selection,selectionArgs,null);
```

```
}
```

对应的 provider：

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[]
       selectionArgs, String sortOrder) {
       SQLiteDatabase db = mOpenHelper.getReadableDatabase();
       String userQuery = "SELECT "+ projection[0]+
                  " FROM useraccounts WHERE "+ selection +selectionArgs[0];
       Cursor cursor = db.rawQuery(userQuery, selectionArgs);
       return cursor;
}
```

如果直接使用用户输入作为查询的参数，恶意用户可以输入类似 1' OR 'a' = 'a 的字符串，进行 SQL 注入攻击。因此不能直接使用用户输入作为查询字符串的一部分。

正确示例：

```
public void getLastName(String input){
       String selection = "userID = ?";
       String[] selectionArgs = {""};
       selectionArgs[0] = input;
       mCursor =
              mResolver.query(mUri,mProjection,selection,selectionArgs,null);
    }
```

对应的 provider：

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[]
       selectionArgs, String sortOrder) {
       SQLiteDatabase db = mOpenHelper.getReadableDatabase();
       String userQuery = "SELECT "+ projection[0]+
                  " FROM useraccounts WHERE "+ selection;
       SQLiteStatement prepStatement = db.compileStatement(userQuery);
       prepStatement.bindString(1, selectionArgs[0]);
       Cursor cursor = db.rawQuery(userQuery, selectionArgs);
       return cursor;
}
```

如果使用参数化查询，则在 SQL 语句中使用占位符表示需在运行时确定的参数值。参数化查询使得 SQL 查询的语义逻辑被预先定义，而实际的查询参数值则等到程序运行时再确定。参数化查询使得数据库能够区分 SQL 语句中语义逻辑和数据参数，以确保用户输入无法改变预期的 SQL 查询语义逻辑。

5. Service

规则 5.1 向 Service 传递敏感数据时，须验证 Service 的合法性

说明：通过包含敏感数据的 Intent 调用 Service，Service 可以获取到这些敏感数据，若 Service 不可信，则会造成敏感数据泄露。
因此，不要轻易把 Intent 传递给不可信的 Service，根据 Service 启动方式的不同有不同的验证方法：

- startService 方式，在所传递的 Intent 中显式指定要启动的 Service 组件名；
- bindService 方式，在 ServiceConnection 的 onServiceConnected 方法里检验 Service 的类名。

错误示例：

```
Intent intent = new Intent();
intent.setAction("com.xcompany.action.SEND_MSG");
intent.putExtra("data",SENSITIVE_MSG);
startService(intent);
```

通过特定的 action 启动 Service，如果恶意应用也申明了该 action，那么它也能接收该 intent，获取传递的敏感信息。

正确示例（startService 方式启动服务）：

```
Intent intent = new Intent(this,com.xcompany.service.DemoService.class);
intent.setAction("com.xcompany.action.SEND_MSG");
intent.putExtra("data",SENSITIVE_MSG);
startService(intent);
```

这里显式指定了需要启动的 Service 组件名。

正确示例（bindService 方式启动服务）：

```
boolean isBound = false;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    Intent intent = new Intent();
    intent.setAction("com.xcompany.action.SEND_MSG");
    intent.putExtra("data","1");
    ServiceConnection conn = new ServiceConnection() {
        @Override
        public void onServiceDisconnected(ComponentName arg0) {
            // TODO Auto-generated method stub
            isBound = false;
        }
        @Override
        public void onServiceConnected(ComponentName arg0, IBinder arg1) {
                // TODO Auto-generated method stub

    if(arg0.getClassName().equals("com.xcompany.service.DemoService")){
                    isBound = true;
                    //业务处理
            }
        }
    };
    bindService(intent, conn, BIND_AUTO_CREATE);
}
```

该示例中，对绑定服务的全路径类名进行合法性校验，只有在白名单内的 Service 才允许进行数据交互和业务处理。

6. Intent

规则 6.1 必须对跨信任边界传入的 Intent 进行合法性判断

说明：攻击者可以通过反编译方式获得应用组件<intent-filter>的内容，然后构造恶意代码为 Intent 设置相应的 Component 名，进而向指定应用组件发送空的 Intent 或携带恶意数据，如果应用接收该 Intent，却没有进行合法性判断，会导致应用程序崩溃。因此，应对外部传入的 Intent 内容进行合法性判断。

通常情况下，应用获取外部不可信 Intent 的方式如下，：

1. 使用 IPC 通信机制的组件通过 getIntent（）方法获取
2. 获取 Public 方法传入的 Intent 参数
3. 通过 parseUri（）方法获取
4. 通过 getParcelable（）方法获取
5. 通过 new Intent（intent）来传递外部 Intent 的方式获取

错误示例：

```
Intent intent = getIntent();
if (intent != null}
{
    if (intent.getBundleExtra("xxx").equals(XXX))
    {
        //...
    }
}
```

上面的例子需要引用外部 Intent 携带的 Bundle 数据，然而，未做合法性判断。若恶意应用程序传递一个携带 null 数据的 Intent，则会造成程序崩溃。

正确示例：

```
Intent intent = getIntent();
if (intent != null}
{
    if (intent.getBundleExtra("xxx") != null)
    {
        if (intent.getBundleExtra("xxx").equals(XXX))
        {
            //...
        }
    }
}
```

正确示例中对从外部接收的 Intent 和 Intent 携带的 Bundle 数据都进行了判空处理，避免了空 intent 导致应用程序崩溃的风险。当然，不仅仅是判空处理，开发人员应针对不同的业务场景做恰当的合法性判断（如边界、大小等限制）。

值得注意的是：从外部收到 Intent 时，如果来源是可预知的，或者有一定的范围，除了对 Intent 进行判空，还建议对接收到的外部 Intent 的来源进行校验，比如通过比对调用者名称摘要白名单的方法来筛选有效的来源，防止恶意应用发送恶意的 Intent 进行攻击。

规则 6.2 必须对 Intent 携带的敏感数据进行加密

说明：组件 Activity、Service 和 Broadcast 间传递数据通常要依赖 Intent，如果 Intent 中携带有敏感数据，不管是应用内或应用间的都需要对其进行加密，防止 Intent 被非法劫持，导致敏感数据泄露。

错误示例（未对敏感数据加密）：

```
Intent intent = new Intent(this, SecondActivity.class);
intent.putExtra("PASSWORD", "my_password");
startActivity(intent);
```

上面的例子没有对 Intent 中的敏感数据进行加密，存在信息泄漏风险。

正确示例：

```
Intent intent = new Intent(this, SecondActivity.class);
String encryptPwd = encryptUtil.encodeAES256("my_password");
intent.putExtra("PASSWORD", encryptPwd);
startActivity(intent);
```

上面的例子对 Intent 中的敏感数据按《密码算法应用规范》进行了加密。

值得注意的是：当使用一个 Activity 的时候，Intent 的如下内容会被 ActivityManager 输出到 LogCat 中，因此敏感数据信息不要放在这些内容中：

1. 目标Package的名称
2. 目标Class的名称
3. 被Intent设置的URI，例如setData()

错误示例（URI 泄露）：

```
Uri uri = Uri.parse("mailto:somebody@xcompany.com");
Intent intent = new Intent(Intent.ACTION_SENDTO, uri);
startActivity(intent);
```

上面的例子将导致 URI 数据被输出到 LogCat 中。

正确示例：

```
Uri uri = Uri.parse("mailto:");
Intent intent = new Intent(Intent.ACTION_SENDTO, uri);
intent.putExtra(Intent.EXTRA_EMAIL, new String[] {"somebody@xcompany.com"});
startActivity(intent);
```

通过使用 putExtra 可以避免 URI 数据被输出到 LogCat 中。

规则 6.3 使用 PendingIntent 触发事件时须传入显式 Intent

说明：PendingIntent 是一种特殊的 Intent，用来启动特定情境下触发的事件。当 A 应用设定一个原始 Intent，并据此创建 PendingIntent，然后将其传递给 B 应用时，B 应用就可以以 A 应用的身份来执行 A 应用预设的操作，并拥有 A 应用同样的权限（with the same Permission and identity）。因此，A 应用应当小心设置原始 Intent，采用在 Intent 中显示指定目的 Component 名称的方式，防止被恶意应用程序劫持，造成权限泄露。

错误示例：

```
private void sendNotification() {
    Intent intent = new Intent();
    // To set implicit Intent.
    intent.setAction("com.xcompany.action.MYACTION");
    int requestCode = 0;
    int flags = 0;
    PendingIntent pendingIntent = PendingIntent.getActivity(this,
            requestCode, intent, flags);
    String tickerText = "ticker text";
    CharSequence contentTitle = "content title";
    CharSequence contentText = "content text";
    Notification notification = new Notification(R.drawable.ic_launcher,
            tickerText, System.currentTimeMillis());
    notification.setLatestEventInfo(this, contentTitle, contentText,
            pendingIntent);
    NotificationManager notificationManager = (NotificationManager) this
            .getSystemService(Context.NOTIFICATION_SERVICE);
    final int NOTIFICATION_REF = 1;
    notificationManager.notify(NOTIFICATION_REF, notification);
```

```
}
```

上面的例子使用隐式 Intent 传给 PendingIntent，一旦触发点击指定通知事件，将会启动 com.xcompany.action.MYACTION 行为的 Activity，并将 PendingIntent 传递过去。此时恶意应用程序仅需指定 com.xcompany.action.MYACTION 行为即可获取原应用程序的权限。

正确示例：

```
private void sendNotification() {
    Intent intent = new Intent();
    // To set explicit component.
    ComponentName component = new ComponentName(
            "com.example.hellonotificationreceiver",
            "com.example.hellonotificationreceiver.MainActivity");
    intent.setComponent(component); //exlicit Intent
    int requestCode = 0;
    int flags = 0;
    PendingIntent pendingIntent = PendingIntent.getActivity(this,
            requestCode, intent, flags);
    String tickerText = "ticker text";
    CharSequence contentTitle = "content title";
    CharSequence contentText = "content text";
    Notification notification = new Notification(R.drawable.ic_launcher,
            tickerText, System.currentTimeMillis());
    notification.setLatestEventInfo(this, contentTitle, contentText,
            pendingIntent);
    NotificationManager notificationManager = (NotificationManager) this
            .getSystemService(Context.NOTIFICATION_SERVICE);
    final int NOTIFICATION_REF = 1;
    notificationManager.notify(NOTIFICATION_REF, notification);
}
```

正确示例中，在指定通知事件触发时，显式指定了需要启动的 Activity，能够有效的防止 Intent 劫持攻击。

规则 6.4 禁止在隐式 Intent 中授予 URI 权限

说明：如果应用程序没有访问某一 URI 的权限，可以通过在 Intent 中设置标签 Flag 为 FLAG_GRANT_READ_URI_PERMISSION 和 FLAG_GRANT_WRITE_URI_PERMISSION 的方式进行授权，同时还需指定相应 <provider>中 android:grantUriPermissions 属性值为 true，才可访问该 URI。若使用隐式 Intent，恶意应用程序一旦劫持到该 Intent，便会获得指定 URI 的读取或写入权限。

错误示例：

```
Intent intent = new Intent();
//To grant URI permissions on implicit intent
intent.setAction("com.android.activity.MYACTION");
intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
intent.setFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
startActivity(intent);
//Manifest.xml
<manifest...>
       <provider...
              android:grantUriPermissions="true" >
       </provider>
</manifest>
```

上面的例子允许授予对 URI 进行读取和写入，但使用了隐式 Intent，恶意应用程序一旦劫持到该 Intent，便会获得该 URI 的读取或写入权限。

正确示例：

```
Intent intent = new Intent(this, SecondActivity.class);
```

```
//To grant URI permissions on explicit intent
intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
intent.setFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
startActivity(intent);
//Manifest.xml
<manifest...>
       <provider...
              android:grantUriPermissions="true" >
       </provider>
</manifest>
```

使用显式 Intent 来允许授予 URI 的读取和写入权限，能够有效防止 Intent 劫持攻击。

7. WebView

规则 7.1 无法确认网页来源或数据合法性时，须禁用 WebView 中的 JavaScript

说明：Android 系统为开发人员提供了 WebView 控件，用于进行网页内容呈现。WebView 控件提供了一个特殊的方法 addJavascriptInterface，通过该方法可以将本地 java 对象提供给外部 javascript 代码调用，从而实现外部 javascript 代码与本地 java 代码的交互。

在 Android 4.2 版本之前，WebView 允许只要通过 addJavascriptInterface 暴露的对象，都可以通过 javascript 访问本地 java 对象的 Public 方法，甚至可以访问其父类的 Public 方法，结合 java 的反射机制，可以获取 java runtime 执行 shell 命令，给系统带来严重的安全隐患。因此，在无法确认网页来源或数据合法性时，须禁用 WebView 中的 JavaScript

Android 4.2 及以后版本，需要使用@JavascriptInterface 注解的本地 Java 对象公共方法才会暴露给外部 javascript 调用。因此，对于需要提供给外部 javascript 调用的本地方法需要进行注解，本地敏感方法还必须校验网页来源或数据的合法性。需要注意的是，使用注解方式必须确保 android:targetSdkVersion 不低于 17，并且应用运行的 Android 系统版本不低于 4.2。

实施指导：

一、如果明确网页内容不需要使用 javascript 脚本，应显式禁用使能 javascript

```
WebView wv = (WebView) findViewById(R.id.wv_broswer);
//禁用 webview 的 javascript 功能
wv.getSettings().setJavaScriptEnabled(false);
```

二、对于需要提供本地方法给外部 javascript 调用的情况

1、对于 Android 4.2 及之后的版本

为了防止 Java 层的函数被随便调用，规定允许被调用的函数必须使用@JavascriptInterface 进行注解，所以如果某应用运行的 Android 系统版本不低于 4.2，需要对外暴露的对象可以通过声明@JavascriptInterface 注解的方式开放接口。

```
class JsObject {
    @JavascriptInterface
    public String toString() { return "injectedObject"; }
}
webView.addJavascriptInterface(new JsObject(), "injectedObject");
webView.loadData("", "text/html", null);
webView.loadUrl("javascript:alert(injectedObject.toString())");
```

Androidmanifest.xml 关于应用最低运行版本要求的配置如下：

```
<uses-sdk
        android:minSdkVersion="17"
        android:targetSdkVersion="17" />
```

需要注意的是，使用@JavascriptInterface 注解声明对外开放的接口，需要确保应用运行环境的 API Level 不低于 17；由于对外接口可能存在不可信输入，需校验入参的合法性。

2、对于 Android 4.2 之前的版本

- 在确保网页来源安全性的前提下，可使能javascript功能。安全的网页来源包括如下两个方面：
  （1） 对于本地文件，应确保webview只访问应用程序内部资源目录下的文件。
  （2） 对于网络访问，应对网页url进行合法性校验（如通过白名单进行控制），对于

确认来源安全的url，建议使用为https访问，并且需要处理SSL错误。

- 在无法保证网页来源的安全性，并且必须使用javascript的情况下，应避免使用addJavascriptInterface功能，同时应通过removeJavascriptInterface方法移除Android系统内部的默认接口。

```
WebView wv = (WebView) findViewById(R.id.wv_broswer);
if (Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN_MR1) {
    wv.removeJavascriptInterface("searchBoxJavaBridge_");  // CVE-2014-1939
    wv.removeJavascriptInterface("accessibility");      // CVE-2014-7224
    wv.removeJavascriptInterface("accessibilityTraversal");  // CVE-2014-7224
}
```

对于移除以上三个系统接口的原因如下：

在2014年发现在Android系统中webkit中默认内置的一个searchBoxJavaBridge_接口，存在远程代码执行漏洞，该漏洞公布于CVE-2014-1939[7];

2014年香港理工大学的研究人员Daoyuan Wu和Rocky Chang发现了两个新的攻击向量存在于android/webkit/AccessibilityInjector.java中，分别是"accessibility"和"accessibilityTraversal"，该漏洞公布于CVE-2014-7224。

8. 文件

规则 8.1 私有文件创建时必须指定 MODE_PRIVATE 模式

说明：Android 系统提供了如下表中所示的四种数据存储的方式：

表 8-1 Android 系统四种数据存储方式

存储方式	描述
File	通常适合于没有固定格式的数据存储
Shared Preferences	以Key-Value的方式保存简要的信息，如配置信息等
SQLite	Android系统提供的轻量级数据库
Content Provider	实现不同应用程序间数据共享

除了Content Provider，若其他三种文件类型存储的数据只在应用程序内部使用，则创建时，必须指定为MODE_PRIVATE模式，否则可能造成私有文件对外暴露。

错误示例(Shared Preferences)：

```
EditText getName = (EditText)findViewById(R.id.findName);
String name = getName.getText().toString();
SharedPreferences.Editor editor = getSharedPreferences("settings", Context.
    MODE_WORLD_WRITEABLE).edit();
editor.putString("username", name);
editor.commit();
```

上述示例使用MODE_WORLD_WRITEABLE模式创建Shared Preferences,恶意应用程序只需指定需要访问的应用程序包名和Shared Preferences文件名即可对Shared Preferences文件进行写操作；同理，若使用MODE_WORLD_READABLE模式创建Shared Preferences文件，则其他应用对该Shared Preferences文件具备可读的权限；通过adb shell进入查看该应用创建的Shared Preferences文件属性，其权限为“-rw-rw-r-”，其中others具有可读权限。

正确示例：

```
EditText getName = (EditText)findViewById(R.id.findName);
String name = getName.getText().toString();
SharedPreferences.Editor editor = getSharedPreferences("settings", Context.
```

```
    MODE_PRIVATE).edit();
editor.putString("username", name);
editor.commit();
```

使用 MODE_PRIVATE 模式创建 Shared Preferences。

通过 adb shell 进入查看该应用创建的 shared Preferences 文件属性，其权限为“-rw-rw----”，其他应用程序对该 shared Preferences 文件没有读写权限。

若使用 MODE_WORLD_READABLE 模式创建文件，则其他应用对该文件具备可读的权限；若使用 MODE_WORLD_WRITEABLE 模式创建文件，则其他应用对该文件具备可写的权限。

正确示例：

```
FileOutputStream outStream = null;
try {
    outStream = this.openFileOutput("test.txt",
            Context.MODE_PRIVATE);
    outStream.write("test123".getBytes());
} catch (FileNotFoundException e) {
    Log.i(TAG, " FileNotFoundException ");
} catch (IOException e) {
    Log.i(TAG, " IOException ");
} finally {
    try {
        if (outStream != null)
            outStream.close();
    } catch (IOException e) {
        Log.i(TAG, " IOException ");
    }
}
```

使用 MODE_PRIVATE 模式创建 file，创建的文件保存在/data/data/xxx/files 目录下。通过 adb shell 进入查看该应用创建的 file 文件属性，其权限为“-rw-rw----”，others 的权限为不可读写；

错误示例（SQLite）：

```
context.openOrCreateDatabase("/data/data/com.android.providers.contacts/data
        bases/contacts2.db", Context.MODE_WORLD_READABLE, null);
```

仅在程序内部使用的 SQLite 数据库可被设备上的其他应用程序读取，容易造成敏感信息泄露。

正确示例：

```
context.openOrCreateDatabase("/data/data/com.android.providers.contacts/data
        bases/contacts2.db", Context.MODE_PRIVATE, null);
```

使用 MODE_PRIVATE 模式创建 contacts2.db。通过 adb shell 进入查看创建的内部存储文件的权限为“-rw-rw----”，其中 others 的权限为不可读写，与应用不同 uid 的用户查看数据库文件内容时提示“Permission denied”。

规则 8.2 不要信任 MODE_WORLD_WRITABLE 模式文件传入的数据

说明：文件的 MODE_WORLD_WRITEABLE 模式表示该文件可以被其他应用写操作，存在数据被其他应用程序篡改的风险。

推荐做法是，不允许文件创建时设置为 MODE_WORLD_WRITABLE 模式，若必须对外提供数据，推荐使用 Content Provider 组件共享数据；

需要注意的是，若应用程序必须对 MODE_WORLD_WRITABLE 模式文件进行读写操作，可以通过 Hash 的方式（推荐使用 HMAC）来校验文件的完整性。

规则 8.3 禁止将特权进程使用的配置文件声明为 MODE_WORLD_WRITABLE 模式

说明：驱动或服务使用的配置文件（如存放在 system/etc 下的文件）往往存在比较重要的

参数，正常情况下应该禁止其他不相关的程序或用户写访问，以免造成重要配置数据被篡改。Google 在 Android 4.4 版本 CTS 中增加对 others 群组权限判断，如果包含写权限无法通过 CTS 权限。

错误示例：

```
/dev/video0                0777  system    camera
```

其他应用有 others 群组权限后，可以随便写系统节点内容，若被恶意应用程序访问，会对系统造成难以预料的危害。

正确示例：

```
/dev/video0                0660  system    camera
/dev/video1                0660  system    camera
/dev/video2                0660  system    camera
```

节点不包含 others 权限。

规则 8.4 禁止将未加密的敏感数据保存到外部设备上

说明：若非业务需要，应用程序应在本地存储敏感数据（比如常见的记住口令场景）。SD 卡等外置设备存储的文件是公共可访问的，应用程序将敏感数据保存到外部设备时，数据可以被其他应用访问，容易造成信息泄露或被恶意篡改。

- Android 4.1版本之前，保存到外部设备上的文件可被随意读取；
- Android 4.1到4.3版本，任意应用程序只需声明WRITE_EXTERNAL_STORAGE权限即可修改外部SD卡存储的文件；
- Android 4.4及之后的版本，应用创建的外部文件会在其外部存储的包名目录下，可以通过设置用户组和MODE模式的方式来管理文件的访问权限。

另外，即便卸载了已将文件写入外部存储卡的应用程序，这些文件也不会被删除。这些问题会危及已写入外部存储的敏感信息，或者使攻击者能够通过修改合法程序所依赖的外部文件将恶意数据注入到合法程序中。

因此，若敏感数据必须存储在外部设备上（如 SD 卡），存储前应将敏感数据进行安全加密。

错误示例：

```
private String filename = "myfile";
private String string = "sensitive data such as credit card number";
FileOutputStream fos = null;
try {
    file file = new File(getExternalFilesDir(TARGET_TYPE), filename);
    fos = new FileOutputStream(file, false);
    fos.write(string.getBytes());
} catch (FileNotFoundException e) {
    // handle FileNotFoundException
} catch (IOException e) {
    // handle IOException
} finally {
    if (fos != null) {
        try {
            fos.close();
        } catch (IOException e) {
            // handle IOException
        }
    }
}
```

这里，包含敏感数据的文件 myfile 未做任何访问权限控制存储在外部设备上，可能会导致敏感信息泄露或篡改攻击。

正确示例（加密存储）：

```
private String filename = "myfile";
private String string = "sensitive data such as credit card number";
FileOutputStream fos = null;
try {
    file file = new File(getExternalFilesDir(TARGET_TYPE), filename);
    String encryptedString = EncryptUtil.encodeAES256(string);
    fos = new FileOutputStream(file, false);
    fos.write(encryptedString.getBytes());
} catch (FileNotFoundException e) {
    // handle FileNotFoundException
} catch (IOException e) {
    // handle IOException
} finally {
    if (fos != null) {
        try {
            fos.close();
        } catch (IOException e) {
            // handle IOException
        }
    }
}
```

敏感数据存储到外部设备前进行安全加密。

正确示例（指定为 MODE_PRIVATE 模式文件）：

```
private String filename = "myfile";
private String string = "sensitive data such as credit card number";
FileOutputStream fos = null;
try {
    fos = openFileOutput(filename, Context.MODE_PRIVATE);
    fos.write(string.getBytes());
} catch (FileNotFoundException e) {
    // handle FileNotFoundException
} catch (IOException e) {
    // handle IOException
} finally {
    if (fos != null) {
        try {
            fos.close();
        } catch (IOException e) {
            // handle IOException
        }
    }
}
```

数据存储在应用程序的目录下，并将文件属性设置为 MODE_PRIVATE，即其他应用程序无法访问此文件。

9. 运行环境

规则 9.1 禁止应用提供设备权限破解功能

说明：应用程序利用系统漏洞可以获取更高的权限（如：root 权限），进而实现设备权限破解的功能。如果应用程序提供上述设备权限破解功能，会导致以下风险：

1．用户或恶意程序执行该功能，会破坏系统的安全性，导致恶意应用可以访问应用数据，造成用户隐私泄露等问题；

2．如果应用程序提供了设备权限破解功能，会被外界质疑为后门；

Android 2.3 版本之前，可以利用 setuid 漏洞获取 root 权限：

adbd 是 Android 系统的一个守护进程，由 init 进程（root 用户）创建，但是创建后自身通过 setuid()设置运行的用户为 shell 用户。由于 linux 系统对于用户最大允许运行的进程数有限制，所以应用程序可以先结束当前 adbd 进程，然后 Init 进程会重新启动新的 adbd 进程，在这个过程中，应用程序需创建大量的子进程，并且这些子进程默认都是 shell 用户身份，从而使得 shell 拥有的进程数达到最大值。这样，导致新启动的 adbd 进程在调用 setuid()时失败，因为 shell 用户的进程已满，adbd 无法把自己运行的用户设置为 shell 用户，从而保持在刚创建时的 root 用户身份中运行。获取 root 权限后，就可以向系统中上传 su 程序。

应用程序可以通过以下代码获取 root 权限：

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

实施指导：

1．自研应用禁止利用系统漏洞获取更高权限；

2．自研应用禁止获取 root 权限执行危险命令；

3．对于敏感应用，应提供设备破解检测机制；

规则 9.2 禁止缓存敏感信息

说明：如果应用程序缓存敏感信息，尤其是缓存在公共存储区，那么这些敏感信息可能会被其他应用获取；若手机丢失或被盗，获得手机者可以通过缓存文件获取用户敏感信息。

可能造成敏感信息泄露的典型场景如下：

场景 1：缓存 WEB 应用的数据可能导致 URL 历史记录、HTTP 头、Cookies 等数据泄露。

场景 2：远程支付类应用通过摄像头拍摄支付（支票）照片并上传，会缓存照片在 Flash 空间上，导致支付信息泄露。

场景 3：涉及敏感信息的 Activity 再被置于后台时，系统会保存微缩图缓存，这可能导致攻击者通过浏览后台任务，导致敏感数据泄露。

场景 4：通过其他应用（如：彩信）分享文件时，缓存文件在公共存储区，导致文件泄露。

对于上述场景，如果想防止敏感信息泄露，可参考如下方式：

场景 1：不缓存敏感信息。

场景 2：不要使用照片，而是使用 SurfaceView 并显示摄像头的取景界面，增加一个拍照按钮，当拍照时，将照片信息保存在一个 buff 中，并在内存中转换为 JPEG 格式后上传到远端。可以通过 Camera 的 onPictureTaken()来实现。

场景 3：涉及用户隐私信息或企业安全的应用，当处于登陆状态放置到后台运行时，需跳转到安全认证界面(手势、指纹等)；也可在相应 Activity 的 onStop 方法中实现微缩图

敏感信息模糊操作（使用掩码或直接遮盖数据）。

场景 4：对于不再使用的临时文件，应及时删除。

由于通过其他应用分享的场景，本应用无法及时获取分享动作是否完成，因此无法及时删除临时文件，所以，可以选择在手机启动完成时删除。

规则 9.3 生产环境下必须设置 android:debuggable 为 false

说明：android:debuggable是为了方便开发人员对应用进行调试。但对于正式发货版本，如果应用设置为可调式模式，方便了攻击者对应用进行更深入的分析调试，不利于对应用的保护，因此要求正式版本须设置为不可调试模式。

对android:debuggable的设置需要在AndroidManifest.xml文件中进行配置。

错误示例：

```
<android xmlns:android="http://schemas.android.com/apk/res/android">
    <manifest>
        <application android:debuggable="true" />
    </manifest>
</android>
```

正确示例：

```
<android xmlns:android="http://schemas.android.com/apk/res/android">
    <manifest>
        <application android:debuggable="false" />
    </manifest>
</android>
```

或者将 android:debuggable 相关的显式配置全部从 AndroidManifest.xml 文件中删除，默认即为 false。

规则 9.4 自定义权限时禁止将 ProtectionLever 属性设置为 normal

说明：在开发过程中，如果使用的应用组件会与其他应用交互，需要为这些组件声明访问权限。可以通过在 AndroidManifest.xml 文件中定义权限的方式实现，使用者必须申请相同的权限才可以调用。对于自研组件一般情况下不会开放给外部使用，对安全级别要求较高，可以通过设置保护等级来提高安全性。

Android 系统为<permission>标签的 ProtectionLevel 属性提供了 4 种级别，如下表所示：

表 9-1 自定义权限四种保护级别

保护级别	描述
Normal	默认保护权限，具有较低安全防护，授予任何应用程序访问的权限
Dangerous	安装时提示用户高危权限，需要用户确认，是否授予访问者权限
Signature	用于限制具有相同证书签名的应用程序访问权限
SignatureOrSystem	系统会把访问权限授予Android系统预装的应用程序，或者是哪些具有相同证书签名的应用程序

对于自定义权限禁止将 ProtectionLevel 属性设置为 normal，而根据实际情况设置为其他保护级别，推荐设置权限保护等级为 signature 或 signatureOrSystem。

错误示例：

```
<permission
      android:name="com.xcompany.permission.TEST"
      android:protectionLevel="normal" >
```

```
</permission>
```

自定义权限的 ProtectionLevel 属性值设置为 normal，外部应用程序只需申请使用 com.xcompany.permission.TEST 权限即可访问。

正确示例：

```
<permission
      android:name="com.xcompany.permission.TEST"
      android:protectionLevel="signature" >
</permission>
```

根据业务需求，设置权限保护等级为 signature。

规则 9.5 涉及现实或虚拟货币及用户隐私的应用必须实现 root 检测及风险控制

说明：对于 Android 手机而言，root 意味着用户可以获取 Android 操作系统的 SuperUser 权限。Android 系统提供了“沙箱机制”来保障不同应用程序和进程之间的互相隔离，即在默认情况下，应用程序没有权限访问系统资源或其它应用程序的资源；然而，拥有 root 权限的应用程序则可以越过“沙箱机制”，无阻碍的与系统或其他应用程序进行交互。因此，涉及隐私及财产安全的应用必须进行 root 检测。

若检测到 Android 设备已被 root，根据应用类型的不同，风险控制策略也不一样：

1. 个人用户型应用程序，考虑到用户的服务体验，需要告知用户风险。
2. 企业型应用程序，考虑到公司的利益与信息安全，应禁用相关敏感功能。

考虑到 root 后的 Android 设备都会存在 root 权限控制文件，/system/bin/su 和 /system/xbin/su 路径下存放有相关的权限控制文件，如果仅仅通过判断这些文件是否存在来进行 root 检测会产生一些误报（如某米手机未 root 也有权限控制文件），若再加入权限控制文件是否具有可执行权限的判断，就可以确定 Android 设备是否 root 了。

规则 9.6 禁止 DexClassLoader 方法在不可信路径下加载和输出类

说明：DexClassLoader 方法用于 Android 动态加载 dex 文件、apk 文件和 jar 文件，输出优化的 dex 文件。使用 DexClassLoader 方法时，如果类的加载和输出路径来自不可信的外部输入或不可信的环境，可能导致 DexClassLoader 类劫持攻击。类加载操作必须在可信边界以内进行，具体可以是确保类的加载和输出路径是在安全目录下，这样外部应用程序既不能改变类的路径，也不能改变类文件。

规则 9.7 禁止使用 checkCallingOrSelfPermission 和 checkCallingOrSelfUriPermission 方法

说明：checkCallingOrSelfPermission() 和 checkCallingOrSelfUriPermission() 用来判定调用程序是否具备访问某个服务或给定 URI 所需的权限。但是，由于此类函数可允许缺少相应权限的恶意应用程序利用合法应用程序的权限进行访问，导致“混淆代理人攻击”，因此应禁止使用。

可使用 checkCallingPermission() 和 checkCallingUriPermission()来替代。