**华为技术有限公司内部技术规范**

DKBA 7940.1-2017.04

**移动应用安全开发规范V2.1**

**Android篇**



华为技术有限公司

版权所有 侵权必究

**修订声明 Revision declaration**

**本规范拟制与解释部门：**

网络安全能力中心、电信软件网络安全工程部、消费者BG软件工程安全工程部

**本规范的相关系列规范或文件：**

*《Java语言编程规范 下卷 安全篇》*（DKBA6915-2014.07）*、《C&C++语言安全编程规范》*（DKBA 6914-2014.07） 、*《Web应用安全开发规范》*（DKBA 1606-2014.07）、*《隐私保护规范》*（DKBA 7981-2015.03）

**相关国际规范或文件一致性：**

无

**替代或作废的其它规范或文件：**

*《移动应用安全开发规范v2.0》*（DKBA 7940-2015.12）

**与相关规范或文件的关系：**

无

|  |  |  |  |
| --- | --- | --- | --- |
| **规范号** | **主要起草部门专家** | **主要评审部门专家** | **版本** |
| DKBA 7940-2015.01 | 电信软件业务部：陈辉军 190784 | 网络安全能力中心：何纲 152106，罗东 256178，朱喜红 210657，杜沸扬 102545，韩庆 265140，李赤阳 306646  电信软件业务部：刘高峰 257474，何伟祥 162822，韩斌 187717，丁鹏亮 192854，姚辉 195967，刑健 215251，朱小龙249319，冯经理284209郭亚奇291956，胡坤红104449，郑明达193729，桂莉莉 266347，刘辉275076  核心网产品线：王宏磊161798，黄树林234216，屈超富208114无线网络产品线：熊晓春257132  消费者BG：许汝波62966，张作强 66512，周雄201051，顾婷WX228487  交换机与企业通信产品线：龚连阳129383，李昂288242，唐正斌210664  中央硬件工程院：施文超303178，王少见 119800  IT产品线：吴鸿钟204126 | V1.0 |
| DKBA 7940.1-2015.12 | 网络安全能力中心：陈栋 291893，杜沸扬 102545，刘杰峰 296112，訾小超 306600，朱迪奇 268324  消费者BG：邓太生 210606、张作强 66512、东巍 208539  企业BG：张鹏 232167、徐海亮 184734、申长山 177870、刘刚 90004755  电信软件业务部：陈辉军 190784、王尚之 293504  核心网产品线：邹迈 185268 | 研发能力中心：郭曙光00121837、朱喜红00210657、余志刚00311016  交换机与企业通信产品线：杨志成00175422  HD固定网络研发管理部：魏建雄00222905  电信软件业务部：何伟祥00162822、陈辉军00190784  中央研究院：翟征德00212921  全球技术服务部：李鑫00178923 | V2.0  1.简单调整文档结构和规则分类  2.增加了部分Android安全相关规则，删除了部分其他规则已涵盖规则  3.优化了某些规则的描述和代码示例  4.所有条目统一称为规则，同时对一些规则添加例外情况描述 |
| DKBA 7940.1-2017.04 | 网络安全能力中心：陈栋 291893，刘杰峰 296112 | 研发能力中心：訾小超 00306600、朱迪奇 00268324  交换机与企业通信产品线：张鹏 00232167  电信软件业务部：陈辉军00190784  终端安全测试部：张悦 00290541、张中文 00377079、武昊英00348603  终端安全工程部：朱奎涛 00187968、袁中举 00152664、洪太亮 00266693、张作强 00329442、邓太生 00210606  能源：詹松烈 00173034  核心网：邹迈 00185268  网络安全设计部：汪泽宇 00357206 | V2.1  1.新增4条Android安全相关规则  2.优化了10条规则的描述和代码示例  3.调整了部分规则的规则名称及结构顺序 |

目录

[DKBA 7940.1-2017.04 1](#_Toc479599839)

[0 概述 6](#_Toc479599840)

[0.1 背景简介 6](#_Toc479599841)

[0.2 使用对象 6](#_Toc479599842)

[0.3 适用范围 6](#_Toc479599843)

[0.4 术语定义 6](#_Toc479599844)

[1 组件通用要求 7](#_Toc479599845)

[**规则 1.1 必须显式设置私有组件的exported属性为false** 7](#_Toc479599846)

[**规则 1.2 必须对对外交互的组件设置访问权限** 8](#_Toc479599847)

[**规则 1.3 禁止口令输入控件提供拷出功能** 9](#_Toc479599848)

[**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** 10](#_Toc479599849)

[**建议 1.5 涉及口令输入的应用界面，应避免使能截屏或录屏操作** 10](#_Toc479599850)

[2 Activity 11](#_Toc479599851)

[**规则 2.1 禁止对私有Activity设置taskAffinity属性** 11](#_Toc479599852)

[**规则 2.2 禁止对私有Activity设置launchMode属性** 11](#_Toc479599853)

[3 Broadcast 12](#_Toc479599854)

[**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** 12](#_Toc479599855)

[**规则 3.2 禁止使用Sticky Broadcast发送敏感数据** 13](#_Toc479599856)

[**规则 3.3 只在程序内部发送的广播须使用local Broadcast** 14](#_Toc479599857)

[**规则 3.4 Broadcast Receiver须分开处理系统广播和应用广播** 15](#_Toc479599858)

[4 Content Provider 16](#_Toc479599859)

[**规则 4.1 必须分别设置外部Content Provider的读写访问权限** 16](#_Toc479599860)

[**规则 4.2 禁止使用不可信数据直接拼接Content Provider查询语句** 17](#_Toc479599861)

[5 Service 19](#_Toc479599862)

[**规则 5.1 向Service传递敏感数据时，须验证Service的合法性** 19](#_Toc479599863)

[6 Intent 20](#_Toc479599864)

[**规则 6.1 必须对跨信任边界传入的Intent进行合法性判断** 20](#_Toc479599865)

[**规则 6.2 必须对Intent携带的敏感数据进行加密** 21](#_Toc479599866)

[**规则 6.3 使用PendingIntent触发事件时须传入显式Intent** 22](#_Toc479599867)

[**规则 6.4 禁止在隐式Intent中授予URI权限** 23](#_Toc479599868)

[**规则 6.5 禁止直接使用隐式Intent启动目标组件** 24](#_Toc479599869)

[7 WebView 25](#_Toc479599870)

[**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** 25](#_Toc479599871)

[**规则 7.2 通过解析网页uri来启动目的组件时须确保目的组件的合法性** 26](#_Toc479599872)

[**7.3 谨防WebView File域同源绕过漏洞** 28](#_Toc479599873)

[**规则 7.3.1 WebView无需使用File协议时，须禁用FileAccess功能** 28](#_Toc479599874)

[**建议 7.3.2 WebView加载的File协议url外部可控时，须实现FileAccess访问范围最小化** 29](#_Toc479599875)

[8 文件 30](#_Toc479599876)

[**规则 8.1 私有文件创建时必须指定MODE\_PRIVATE模式** 30](#_Toc479599877)

[**规则 8.2 不要信任MODE\_WORLD\_WRITABLE模式文件传入的数据** 32](#_Toc479599878)

[**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** 32](#_Toc479599879)

[**规则 8.4 禁止将未加密的敏感数据保存到外部设备上** 33](#_Toc479599880)

[9 运行环境 34](#_Toc479599881)

[**规则 9.1 禁止应用提供设备权限破解功能** 34](#_Toc479599882)

[**规则 9.2 禁止缓存敏感信息** 35](#_Toc479599883)

[**规则 9.3 敏感数据备份前应先加密** 36](#_Toc479599884)

[**规则 9.4 生产环境下必须设置android:debuggable为false** 37](#_Toc479599885)

[**规则 9.5 自定义权限时禁止将ProtectionLever属性设置为normal** 37](#_Toc479599886)

[**规则 9.6 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** 38](#_Toc479599887)

[**规则 9.7 禁止使用类加载器方法在不可信路径下加载和输出类** 39](#_Toc479599888)

[**规则 9.8 禁止使用危险权限检测方法** 40](#_Toc479599889)

[10 其他 41](#_Toc479599890)

[**规则 10.1 APK安装包须进行代码混淆** 41](#_Toc479599891)

[**规则 10.2 应用程序只申请业务必要的权限** 42](#_Toc479599892)

[**规则 10.3 禁止自定义权限及行为以系统命名空间开头** 42](#_Toc479599893)

[**规则 10.4 不同应用程序不要设置相同的UID** 43](#_Toc479599894)

[**规则 10.5 对外发布的应用程序须实现自动更新机制** 43](#_Toc479599895)

[**规则 10.6 应用的保持登陆选项，用户能够自主选择和取消** 43](#_Toc479599896)

[**规则 10.7 必须对涉及现实或虚拟货币操作的应用提供会话保护机制** 44](#_Toc479599897)

[**规则 10.8 必须对涉及现实或虚拟货币交易操作提供重认证** 45](#_Toc479599898)

[参考资料 47](#_Toc479599899)

概述

## 0.1 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，例如*The Cert Secure Coding Standard for Android*、*Android Secure Coding（Japan）*、*Android Security Guide*和*OWASP Mobile Top 10，*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。

Android应用通常使用Java语言开发，业务特性可能涉及认证、会话管理、隐私保护等内容，公司已有相应的《Java语言安全编程规范》、《会话管理安全设计规范》、《Web应用安全开发规范》、《隐私保护规范》，Android应用开发应遵从相关规范要求。

## 0.2 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。

## 0.3 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。

## 0.4 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

**显式Intent：**代码中指定了目标包名、类名、组件名中至少一种的Intent对象

**隐式Intent：**非显式Intent对象，代码中使用action、data、category中至少一种条件来匹配目标的Intent对象

**系统级应用：**Android系统原生应用及共享系统进程UID的自研应用

组件通用要求

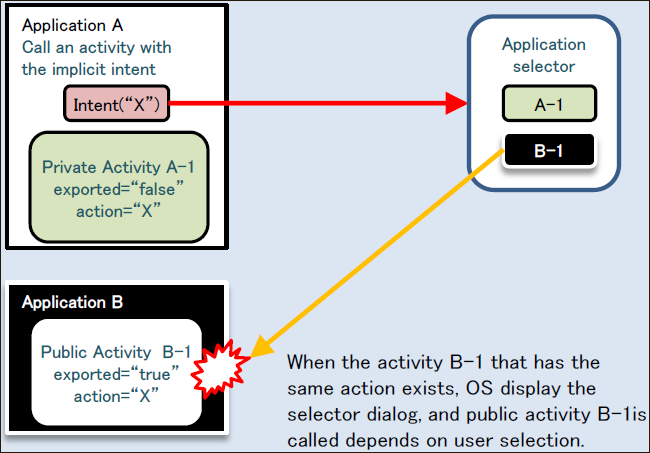
**规则 1.1** **必须显式设置私有组件的exported属性为false**

**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。

对于Content Provider组件，若未设置exported属性，则在Android 4.2版本之前，组件默认对外开放。对于Activity、Service、Receiver组件，若未设置exported属性，且设置了intent-filter标签，则组件默认对外开放。

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。

图1-1 私有组件声明<intent-filter>标签存在信息泄露风险

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.huawei.PrivateActivity"*

android:label=*"@string/app\_name"* >

**<intent-filter>**

**<action android:name=*"com.huawei.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.huawei.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.huawei.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，且声明了<intent-filter>标签，那么该组件默认是对外开放的。

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属性来设置访问权限。

例外情况：对于纯系统行为注册的开放组件可不必设置访问权限。

**错误示例：**

<activity

android:name=*"com.huawei.PublicActivity"*

android:label=*"@string/app\_name"* >

**<intent-filter>**

**<action android:name=*"com.huawei.action.EVENT\_CHANGE"* />**

**</intent-filter>**

</activity>

可以看出，任何指定com.huawei.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.huawei.permission.ACCESS"*

android:protectionLevel=*"signature"* >

</permission>

<application

android:allowBackup=*"false"*

android:icon=*"@drawable/ic\_launcher"*

android:label=*"@string/app\_name"*

android:theme=*"@style/AppTheme"* >

<activity

android:name=*"com.huawei.PublicActivity"*

android:label=*"@string/app\_name"*

**android:permission=*"com.huawei.permission.ACCESS"* >**

<intent-filter>

<action android:name=*"com.huawei.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。

对于认证公司内部应用，在<permission>标签中设置android:protectionLevel属性为signature，只允许拥有相同开发者签名的应用才能访问；对于认证非华为签名的外部应用，设置android:protectionLevel属性为dangerous,并需通过 “白名单”（如证书、包名）来认证外部应用来源。

**规则 1.3 禁止口令输入控件提供拷出功能**

说明：口令输入控件必须禁用拷出功能，以防止从输入控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控件的长按和双击功能来禁止选取文本。

错误示例：

<EditText

android:id=*"@+id/PWDEditText"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:inputType=*"textPassword"* />

EditText口令输入控件只设置了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选取文本的，因此涉及口令输入组件的情况下，应取消横屏界面UI。另外，Google原生口令输入控件默认已实现禁止拷出功能。

**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。

<EditText

android:id=*"@+id/PWDEditText"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:imeOptions=*"flagNoExtractUi"*

android:inputType=*"textPassword"*

**android:longClickable=*"false"* />**

也可以通过java代码实现：

editText.setLongClickable(**false**);

editText.setImeOptions(EditorInfo.*IME\_FLAG\_NO\_EXTRACT\_UI*);

**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。

**实施指导：**

* 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

**建议 1.5 涉及口令输入的应用界面，应避免使能截屏或录屏操作**

**说明：**口令输入界面主要涉及账号密码输入框及输入法弹出框，在用户输入口令时，应避免使能截屏或录屏操作。否则，一旦恶意软件骗取用户信任获得授权，窃取到用户操作的账号和密码，即可在其他设备中登录，造成用户隐私泄露。

针对第三方应用口令输入界面进行截屏或录屏操作时，系统应做出警告，提醒用户。

**实施指导：**

可以通过Google提供的对Window进行secure flag设置的方式：

1. 针对Activity界面，应在界面初始化时获取Window句柄

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.***activity\_main***);

Window win = getWindow();

win.addFlags(WindowManager.LayoutParams.***FLAG\_SECURE***);

}

1. 针对Dialog弹窗，可在获取Dialog实例后获取Window句柄

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.***activity\_main***);

Dialog dl = **new** Dialog(**this**);

Window win = dl.getWindow();

win.addFlags(WindowManager.LayoutParams.***FLAG\_SECURE***);

}

addFlags方法也可用setFlags方法替代，更多详情请参考[【参考资料9】](#OLE_LINK2)

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.huawei.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.huawei.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.huawei.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.huawei.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或singleTask。

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.huawei.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.huawei.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.huawei.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.huawei.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.huawei.action.EXAMPLE");

intent.putExtra(KEY, SENSITIVE\_DATA);

sendBroadcast(intent,"com.huawei.permission.TEST");

这里指定了Receiver权限，即只有声明了com.huawei.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.huawei.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.huawei.permission.TEST"*

android:protectionLevel=*"signature"* >

</permission>

然后，在静态<receiver>标签中添加相应权限的声明：

<receiver

android:name=*".ExampleReceiver"*

android:permission=*"com.huawei.permission.TEST"* >

<intent-filter>

<action android:name=*"com.huawei.action.EXAMPLE"* />

</intent-filter>

</receiver>

或是在动态注册receiver时添加相应权限的声明：

registerReceiver(receiver, filter, *"com.huawei.permission.TEST"*,**null**);

这样一来，该Receiver便只能接收来自具有com.huawei.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.huawei.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](#OLE_LINK1)

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

<permission

android:name=*"com.huawei.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.huawei.permission.TEST");

这里使用Broadcast替代Sticky Broadcast，并指定了Receiver权限，即只有具有com.huawei.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。

**错误示例:**

直接使用sendBroadcast()、sendOrderBroadcast()、endStickyOrderedBroadcast()或sendStickyBroadcast()方法发送程序内部使用的广播。

Intent intent=**new** Intent("com.huawei.action.PRIVATE\_BROADCAST");

intent.putExtra(KEY, SENSITIVE\_DATA);

sendBroadcast(intent);

**正确示例：**

首先获取LocalBroadcastManager实例：

LocalBroadcastManager lbm = LocalBroadcastManager.*getInstance*(**this**);

然后通过函数 registerReceiver方法来注册监听器：

lbm.registerReceiver(**new** BroadcastReceiver(){

@Override

**public** **void** onReceive(Context arg0, Intent arg1) {

// **TODO** Auto-generated method stub

}

}, **new** IntentFilter("com.huawei.action.PRIVATE\_BROADCAST")); lbm.registerReceiver(new BroadcastReceiver() {

最后调用sendBroadcast方法发送广播： @Override public void onReceive(Context context, Intent intent) { // TODO Handle the received local broadcast

Intent intent=**new** Intent("com.huawei.action.PRIVATE\_BROADCAST");

intent.putExtra(KEY, SENSITIVE\_DATA);

lbm.sendBroadcast(intent); }

}, new IntentFilter(LOCAL\_ACTION));

Read more: http://blog.chengyunfeng.com/?p=498#ixzz2l9b1fFR2

**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开注册可以降低两者之间的影响，通常情况下，系统广播不会设置访问权限，因此应用广播也不会设置访问权限，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，Broadcast Receiver分开注册、处理系统广播和应用广播可以有效防止外部恶意软件进行广播仿冒或混淆攻击。

**错误示例:**

<receiver android:name=*".MyBrocast"* >

<intent-filter android:priority=*"2147483647"*>

<!-- 注册混合广播行为 -->

<action android:name=*"com.forrest.action.mybroadcast"* />

<action android:name=*"android.intent.action.EMERGENCY\_DIAL"* />

<action android:name=*"android.intent.action.DELETE"* />

<action android:name=*"android.intent.action.SETTINGS"* />

</intent-filter>

</receiver>

这里Receiver混合注册了系统广播和应用广播的接收行为。

Broadcast Receiver调用onReceiver方法处理广播时若没有区分处理系统广播和应用广播；一旦系统广播涉及敏感数据或操作，恶意攻击者只需发送指定行为的应用广播即可触发广播处理操作，造成广播混淆攻击。

**正确示例（分开注册系统广播和应用广播）：**

<receiver android:name=*".MyBrocast1"* >

<intent-filter android:priority=*"2147483647"*>

<action android:name=*"android.intent.action.EMERGENCY\_DIAL"* />

<action android:name=*"android.intent.action.DELETE"* />

<action android:name=*"android.intent.action.SETTINGS"* />

</intent-filter>

</receiver>

<receiver android:name=*".ExampleReceiver"*

android:permission=*"com.huawei.permission.DEMO"* >

<intent-filter android:priority=*"2147483647"*>

<action android:name=*"com.forrest.action.mybroadcast"* />

</intent-filter>

</receiver>

若广播涉及敏感数据或操作，系统广播和应用广播须分开注册，可以从业务逻辑上隔离开不同广播的处理过程，保障系统广播安全的同时，能够对应用广播做细腻度的访问控制。

**正确示例（注册混合广播做访问权限控制）：**

<receiver android:name=*".MyBrocast"*

android:permission=*"com.huawei.permission.DEMO"* >

<intent-filter android:priority=*"2147483647"*>

<!-- 注册混合广播行为 -->

<action android:name=*"com.forrest.action.mybroadcast"* />

<action android:name=*"android.intent.action.EMERGENCY\_DIAL"* />

<action android:name=*"android.intent.action.DELETE"* />

<action android:name=*"android.intent.action.SETTINGS"* />

</intent-filter>

</receiver>

必须注册混合广播的情况下，应对Receiver组件做访问权限控制，这样可在不影响系统广播接收处理的前提下，确保攻击者无法仿冒或劫持应用广播。

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.huawei.ExampleProvider"*

android:authorities=*"com.huawei.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之前的版本中该组件就可以被其他应用任意访问。

错误示例：

<provider

android:name=*"com.huawei.ExampleProvider"*

android:authorities=*"com.huawei.provider"*

android:exported=*"true"*

android:Permission=*"com.huawei.permission.DEMO"*/>

应用程序直接在provider标签下设置android:Permission属性，则会导致该Content provider组件的读写权限一致，若第三方组件应只授予读或者写访问权限，则会违背权限最小化安全设计原则。

正确示例（分开设置provider的读写权限）：

<manifest

xmlns:android=*"http://schemas.android.com/apk/res/android"*

package=*"com.huawei"*

android:versionCode=*"1"*

android:versionName=*"1.0"* >

<uses-sdk android:minSdkVersion=*"8"* />

<permission android:name=*"com.huawei.permission.READ"*

android:protectionLevel=*"signature"* />

<permission android:name=*"com.huawei.permission.WRITE"*

android:protectionLevel=*"signature"* />

<application

android:icon=*"@drawable/ic\_launcher"*

android:label=*"@string/app\_name"* >

<provider

android:name=*"com.huawei.ExampleProvider "*

android:authorities=*"com.huawei.provider "*

android:exported=*"true"*

android:readPermission=*"com.huawei.permission.READ"*

android:writePermission=*"com.huawei.permission.WRITE"* />

</application>

</manifest>

该示例中，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的读权限）：

<provider

android:name=*"com.huawei.ExampleProvider"*

android:authorities=*"com.huawei.provider "*

android:exported=*"true"*

android:readPermission=*" com.huawei.permission.READ"*

android:writePermission=*"com.huawei.permission.WRITE"*>

<path-permission

android:pathPrefix=*"/hello"*

android:readPermission=*"com.huawei.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。

上述示例中，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.huawei.HELLO\_PERMISSION"*/>

如果需要在hello目录下写入数据就需要申请如下权限：

<uses-permission android:name=*"com.huawei.permission.WRITE"*/>

需要特别注意的是：如果<provider>中没有设置访问权限，而只在<path-permission>中设置，那么只有<path-permission>中定义的URI路径受到访问权限保护，该做法显然是不正确的，并且在android 2.3.3版本，只在<path-permission>设置的权限不会生效。

例外情况：系统级应用的Content Provider可直接设置permission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。

值得注意的是：若开发人员需要重写ContentProvider的openFile接口时，应对Uri路径解码后再过滤../字符串，防止路径遍历攻击。

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。

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.huawei.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.huawei.service.DemoService.**class**);

intent.setAction("com.huawei.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.huawei.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.huawei.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。

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校验。

错误示例：

Intent intent = getIntent();

Bundle bundle = intent.getExtras();

String str = **null**;

**if**(bundle != **null**){

str = (String)bundle.get("key");

// handle str

}

上面的例子对外部Intent数据直接进行类型转换操作，若攻击者传入一个非String类型的Intent数据，则可能导致应用程序崩溃。类似可能存在风险的操作还有数组边界和序列化。

正确示例：

Intent intent = getIntent();

Bundle bundle = intent.getExtras();

String str = **null**;

**if**(bundle != **null**){

**try** {

str = (String)bundle.get("key");

// handle str

} **catch** (Exception e) {

Log.*d*("TAG", MSG);

}

}

对外部Intent数据类型转换操作进行了异常捕获处理。

正确示例：

Intent intent = getIntent();

Bundle bundle = intent.getExtras();

String str = **null**;

Object obj = bundle.get("key");

**if**(bundle != **null** && obj **instanceof** String){

str = (String)obj;

// handle str

}

对外部Intent数据进行了类型转换前合法性校验。

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，除了对外部Intent数据进行合法性校验，还建议对外部Intent的来源进行校验，比如通过白名单的方式来筛选有效的来源，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。



图6-1 恶意应用通过注册动态接收器，截获Intent

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@huawei.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@huawei.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.huawei.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.huawei.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.huawei.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。

例外情况：使用PendingIntent触发事件时，允许传入纯系统行为的隐式Intent。

**规则 6.4 禁止在隐式Intent中授予**[**URI权限**](https://www.securecoding.cert.org/confluence/display/android/DRD05-J.+Do+not+grant+URI+permissions+on+implicit+intents)

**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。

**规则 6.5 禁止直接使用隐式Intent启动目标组件**

**说明：**隐式Intent是通过条件匹配的方式来筛选组件，凡是满足条件的组件都有被启动的可能：Activity会弹出选择框，由用户决定启动那个Activity；Service则是会启动被系统优先注册的那一个；而所有满足条件的Receiver都会启动广播处理操作。

因此，应用程序能够显式指定目标组件（类名、组件名、包名）的情况下，禁止直接使用隐式Intent。必须使用隐式Intent启动目标组件的情况下，须至少满足如下场景之一：

1. 纯系统行为的隐式Intent。如通过Intent.ACTION\_CALL行为启动拨打电话界面。
2. 目标组件已做访问权限控制，且隐式Intent中不携带敏感数据。如华为穿戴应用通过权限控制保障只与华为设备交互。
3. 启动前已通过“白名单”（如签名、证书等）方式校验目标组件合法性。如和第三方支付应用交互前“白名单”校验确保目标组件的合法性。
4. 隐式Intent中不携带敏感数据，且目标组件不涉及敏感操作。

**错误示例:**

Intent intent = **new** Intent();

intent.setAction("com.huawei.action.DEMO");

intent.putExtra("key", SENSITIVE\_DATA);

startActivity(intent);

这里直接使用隐式Intent启动Activity。攻击者一旦构造一个com.huawei.action.DEMO行为的恶意Activity，则可能造成Intent劫持攻击，导致敏感信息泄露；若目标组件涉及敏感操作（如文件下载），攻击者可以仿冒发送com.huawei.action.DEMO行为的Intent，从而导致无法预估的危害。

**正确示例（显式Intent启动目标组件）：**

Intent intent = **new** Intent();

intent.setClass(**this**, MyActivity.**class**);

intent.setAction("com.huawei.action.DEMO");

intent.putExtra("key", SENSITIVE\_DATA);

startActivity(intent);

正确示例中通过指定目标组件类名的方式，显式Intent方式启动Activity。

**正确示例（纯系统行为隐式Intent启动目标组件）：**

Intent intent = **new** Intent();

intent.setAction(Intent.***ACTION\_VIEW***);

intent.setData(Uri.*parse*(url));

startActivity(intent);

系统行为本身就是对外开放的，任何声明拥有系统权限的应用，即可访问注册了对应系统行为的组件。因此，在无法使用显式Intent方式启动目标组件的情况下，允许使用纯系统行为的隐式Intent来启动目标组件。

**正确示例（目标组件已做访问权限控制）：**

<uses-permission android:name="huawei.permission.DEMO"/>

Intent intent = **new** Intent();

intent.setAction("com.huawei.action.DEMO");

startActivity(intent);

目标组件：

<activity

android:name=*"myActivity"*

android:label=*"@string/app\_name"*

android:permission=*"huawei.permission.DEMO"*>

<intent-filter >

<action android:name=*"huawei.action.DEMO"*/>

<category android:name=*"android.intent.category.DEFAULT"*/>

</intent-filter>

</activity>

这里，启动方在AndroidManifest.xml配置文件中声明拥有huawei.permission.DEMO权限，通过隐式Intent的方式启动目标组件，目标组件做了huawei.permission.DEMO权限访问控制，确保不可被外界随意访问。然而，隐式Intent的方式启动目标组件存在Intent劫持攻击的风险，因此该场景下，隐式Intent不能携带敏感数据。

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，同时还应通过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

}

Android 4.4之前的版本必须移除以上三个系统接口，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。

可通过如下网页检测运行环境中的是否存在内置接口。

<html>

<head>

<script type=*"text/javascript"*>

function check() {

var div = document.getElementById("result")

div.innerHTML = "Found injected java object: <BR>"

for (var obj in window) {

if ("getClass" in window[obj]) {

div.innerHTML += obj + "<BR>"

}

}

}

</script>

</head>

<body>

<input type=*"button"* onclick=*"check()"* value=*"check"*/>

<div id=*"result"*></div>

</body>

</html>

**规则 7.2 通过解析网页uri来启动目的组件时须确保目的组件的合法性**

说明：Android系统提供一种服务，可以通过网页Uri来启动目标组件，该uri需遵循Intent Scheme Uri规则，通过使用parseUri方法将Uri解析为intent后，来启动目标组件。

Intent Scheme Uri基本语法如下：

intent:HOST/URI-path // Optional host

#Intent;

package=[string];

action=[string];

category=[string];

component=[string];

scheme=[string];

end;

例如

<a href=”intent://foobar/#Intent; action=myaction; type=text/plain; S.xyz=123; i.abc=678;component=com.android.browser/com.android.browser.BrowserActivity;end”>

等效于：

Intent intent = new Intent(“myaction”);

intent.setData(Uri.parse(“//foobar/”));

intent.putExtra(“xyz”, “123”);

intent.putExtra(“abc”, 678);

intent.setComponent(ComponentName.unflattenFromString(“com.android.browser/com.android.browser.BrowserActivity”));

基于以上的规则，类似的也可以通过Intent Uri打开指定的组件，值得说明的是，通过网页Uri的方式甚至可以启动该webview所在应用的私有activity，导致敏感信息泄露等危害。曾受该漏洞影响的包括遨游、Opera等一些知名浏览器，google chrome也出现过该问题，chrome虽然在parseUri之后通过类似setComponent(null)的方式防御了基于intent component的攻击方式，但并没有防范selector intent，因此导致了一个UXSS漏洞。

Chrome之UXSS：

Chrome的UXSS漏洞利用相对复杂。介绍之前，我们需要先了解一下关于Intent Selector的用法，详情见。简而言之，Intent Selector机制提供一种main intent不匹配的情况下可以设置替补的方案。比如A是main intent, B是A的selector intent，当startActiviy时，系统发现A无法匹配则会尝试用B去匹配。

Chrome相比于Opera，在intent过滤的步骤中添加了安全策略，代码如下：

Intent intent = Intent.parseUri(uri);

intent.addCategory(“android.intent.category.BROWSABLE”);

intent.setComponent(**null**);

context.startActivityIfNeeded(intent, -1);

从代码中，可以看到Chrome为了防御Intent Based攻击，做了不少限制，比如把category强置为”android.intent.category.BROWSABLE”，把component强置为null，相对之后比Opera强多了。然而，Chrome忽略了Intent Selector的用法，比如下面的用法:

intent:#Intent;S.xxx=123; SEL;component=com.android.chrome/.xyz;end

留意其中的关键字“SEL”，其实就是设置了一个component为com.android.chrome/.xyz的 selector intent，这种用法导致chrome的防御措施形同虚设。最后看一下Chrome UXSS的PoC：

<script>

    //通过WebAppActivity0我们先打开一个攻击的站点

    location.href = "intent:#Intent;S.webapp\_url=http://victim.example.jp;l.webapp\_id=0;SEL;compo nent=com.android.chrome/com.google.android.apps.chrome.webapps.WebappActivity0;end";

    // 停留2s或者更长时间, 然后注入javascript payload

    setTimeout(function() {

    location.href = "intent:#Intent;S.webapp\_url=javascript:(malicious javascript code);l.webapp\_id=1;SEL;component=com.android.chrome/com.google.android.apps.chrome.webapps.WebappActivity0;end";

    }, 2000);

</script>

这里的关键点是WebappActivity0对new intent的处理方式上。

第一次打开站点，并完成加载。第二次则是直接把javascript payload注入到目标网页。这个漏洞存在于在所有低于v.30.0.1599.92的chrome版本，而新版本修改了对new intent的处理方式，会创建new tab，这样就避免了javascript inject。

然而在新版中，依然没有屏避intent selector的使用，因此依然存在Chrome的私有组件和文件被读取的安全隐患。具体内容请参见[【参考资料8】](#OLE_LINK2)。

实施指导：

Selector intent是android 4.0.3引入的一个机制，该机制允许主intent附加一个selector intent，当一个主intent附加了selector intent后,如果主intent没有指定component组件名，则android会通过selector intent解析需要启动的目的组件。uri中可通过SEL参数指定selector intent，所有SEL之后end之前的参数都被解析为selector intent的内容，比如：

”intent: http://www.huawei.com#Intent;SEL; component = com.android.browser/com.android.browser.BrowserActivity;end”

等效于目的组件名被指定为

com.android.browser/com.android.browser.BrowserActivity

实际上还是会绕过setComponent(null)处理。

因此，当我们需要自行通过parseUri解析uri来启动一些应用时，需要特别注意对目的组件的合法性判断，可参考下面相对比较安全的解析处理：

// convert intent scheme URL to intent object

Intent intent = Intent.parseUri(uri);

// forbid launching activities without BROWSABLE category

intent.addCategory("android.intent.category.BROWSABLE");

// forbid explicit call

intent.setComponent(null);

// forbid intent with selector intent

intent.setSelector(null);

// start the activity by the intent

context.startActivityIfNeeded(intent, -1);

**7.3 谨防WebView File域同源绕过漏洞**

Android提供的“沙箱”保护机制实现了各应用间的数据相互隔离，即通常情况下，A应用是不能访问B应用的数据的。但不正确的使用WebView可能会打破这种隔离：假设B应用加载外部传入的url，且使能FileAccess功能，则A应用可以构造一个恶意file协议的url让B应用加载，从而获取B应用的内部私有文件。

FileAccess功能涉及的4个处理方法：

1. setAllowFileAccess：是否允许WebView使用File协议，默认是允许的。若设置为False，则WebView不能加载本地文件。
2. setJavaScriptEnabled：是否允许WebView使用JavaScript脚本，默认是不允许的。该方法主要和下面两个FileAccess功能方法配合使用。
3. setAllowFileAccessFromFileURLs：是否允许file url加载的JavaScript读取其他的本地文件，Android 4.1之前的版本默认是允许的，Android 4.1及之后的版本默认禁止。
4. setAllowUniversalAccessFromFileURLs：是否允许通过file url加载的JavaScript访问其他的本地文件和http、https等其他的源，Android 4.1之前的版本默认是允许的，Android 4.1及之后的版本默认禁止。若设置了该方法，setAllowFileAccessFromFileURLs方法的设置将不起作用。

**规则 7.3.1 WebView无需使用File协议时，须禁用FileAccess功能**

**说明：**Android系统默认允许WebView使用File协议。因此，若Webview无需使用File协议的情况下，应显式禁用FileAccess功能。

**错误示例:**

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.***activity\_main***);

WebView webView = (WebView)findViewById(R.id.my\_wv);

Intent intent = getIntent();

String url = intent.getData().toString();

//WebView未设置setAllowFileAccess

webView.loadUrl(url);

}

WebView加载来自外部的url，且未设置setAllowFileAccess方法，则默认允许使用File协议，攻击者只需构造一个恶意file url，即可获取应用或系统数据。

访问SD卡文件:adb shell am start –n package/.WebViewDemo –d file:///sdcard/attack.txt

访问系统配置文件：adb shell am start –n package/.WebViewDemo –d file:///etc/hosts

访问私有文件：adb shell am start –n package/.WebViewDemo –d file:///data/data/package/X.xml

**正确示例：**

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.***activity\_main***);

WebView webView = (WebView)findViewById(R.id.my\_wv);

webView.getSettings().setAllowFileAccess(**false**);

Intent intent = getIntent();

String url = intent.getData().toString();

webView.loadUrl(url);

}

这里，WebView不需要使用File协议，通过设置setAllowFileAccess属性值为false来禁用FileAccess功能。

**建议 7.3.2 WebView加载的File协议url外部可控时，须实现FileAccess访问范围最小化**

**说明：**若WebView必须使用File协议，且加载的File协议url外部可控时，应根据业务场景，可通过url路径合法性校验（路径标准化并确保在指定目录下）、url所指向文件完整性校验（校验文件的哈希值）、禁用JavaScript脚本等手段来确保FileAccess访问范围最小化，降低安全风险。

**错误示例:**

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.***activity\_main***);

WebView webView = (WebView)findViewById(R.id.my\_wv);

webView.getSettings().setAllowFileAccessFromFileURLs(**true**);

webView.getSettings().setJavaScriptEnabled(**true**);

Intent intent = getIntent();

String url = intent.getData().toString();

webView.loadUrl(url);

}

WebView加载来自外部的url，设置setAllowFileAccessFromFileURLs方法值为true，且使能JavaScript脚本，则允许通过file url加载的JavaScript读取其他的本地文件。

POC示例代码：fileUrlsAttack.html

<script>

var xmlHttpRep = **new** XMLHttpRequest();

xmlHttpRep.onload = function(){

var contentBox = document.getElementById("contentBox");

contentBox.innerHTML = xmlHttpRep.responseText;

};

**try**{

xmlHttpRep.open("get","/etc/hosts",**true**);

xmlHttpRep.send();

}**catch**(ex){

display(ex.message);

}

</script>

通过adb shell am start –n package/.WebViewDemo –d file:///sdcard/fileUrlsAttack.html命令即可访问/etc/hosts系统配置文件。

**正确示例：**

**protected** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.***activity\_main***);

WebView webView = (WebView)findViewById(R.id.my\_wv);

webView.getSettings().setAllowFileAccessFromFileURLs(**false**);

webView.getSettings().setJavaScriptEnabled(**false**);

Intent intent = getIntent();

String url = intent.getData().toString();

**if(isSecure(url))**

webView.loadUrl(url);

}

这里是通过禁用JavaScript脚本，来防止WebView解析JavaScript脚本来访问其他本地文件，且对url做合法性校验，来保障该WebView的FileAccess访问范围最小化。

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Shared Preferences文件属性，权限为“-rw-rw-r–”，说明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。

**错误示例(File):**

FileOutputStream outStream = **null**;

**try** {

outStream = **this**.openFileOutput("test.txt",

Context.MODE\_WORLD\_WRITEABLE);

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 ");

}

}

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。安全强哈希算法操作请参见《密码算法应用规范》。

**错误示例:**

String fileName = "/system/etc/media\_codecs.xml";

File testfile = **new** File(fileName);

**if** (testfile.exists()) {

**if** (testfile.canRead()) {

Log.i("testfile","filename exist canRead");

//Handle data

}

} **else** {

Log.e("testfile","media filename Not exist");

}

上述代码中，需要读取media\_codecs.xml文件存储的数据进行I/O操作，然而，仅仅判断该文件是否可读，若该文件内容可被其他应用程序篡改，则传入数据不可信。

**正确示例：**

String fileName = "/system/etc/media\_codecs.xml";

File testfile = **new** File(fileName);

**if** (testfile.exists()) {

**if** (testfile.canWrite()) {

**if** (SecureHashEquals(testfile)) {

//Handle data

} **else** {

Log.i("testfile","media file has been tamper!");

}

}

//Handle data

} **else** {

Log.e("testfile","media filename Not exist");

}

正确示例中，判断全局可写文件的哈希值是否与上次写入后的哈希值相等，若相等，则信任文件数据，否则认为文件数据已被篡改并记录日志。

**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。加密操作请参考《密码算法应用规范》。

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。

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。

**错误示例：**

**public final static** String *SHARE\_TEMP\_PICTURE\_NAME* = "DADKKWPOGJKA.png";

**private** Uri getBitmap(Context context) {

recycleBitmap();

FileOutputStream mFileOutPutStream = **null**;

Bitmap bitmap = getShareBitmap();

File file = **new** File(context.getCacheDir(), *SHARE\_TEMP\_PICTURE\_NAME*);

**try** {

mFileOutPutStream = **new** FileOutputStream(file);

**if** (**null** != bitmap) {

bitmap.compress(CompressFormat.JPEG, 30, mFileOutPutStream);

}

} **catch** (FileNotFoundException e) {

Log.e(TAG, e.getMessage());

} **catch** (Exception ex) {

Log.e(TAG, ex.getMessage());

}

**try** {

**if** (**null** != mFileOutPutStream) {

mFileOutPutStream.flush();

mFileOutPutStream.close();

}

} **catch** (IOException e) {

Log.e(TAG, e.getMessage());

}

recycleBitmap();

**if** (file.exists()) {

file.setReadable(**true**, **false**);

**return** Uri.fromFile(file);

} **else** {

**return null**;

}

}

上述代码的使用场景为，应用将自己的照片通过Android的分享机制，由用户选择一个应用（如MMS）分享给其他用户。为了能够让其他应用读取照片，需要创建一个临时文件，并且，该临时文件能够被其他应用访问。这样，如果临时文件包含敏感数据，并且一直存在，则存在信息泄露的风险。

**正确示例：**

<uses-permission android:name=*"android.permission.RECEIVE\_BOOT\_COMPLETED"* />

<receiver android:name=*".BootCompletedReceiver"* >

<intent-filter>

<action android:name=*"android.intent.action.BOOT\_COMPLETED"* />

</intent-filter>

</receiver>

**public** **class** BootCompletedReceiver **extends** BroadcastReceiver {

**public** **void** onReceive(Context context, Intent intent) {

**if** (**null** == intent || **null** == intent.getAction) {

**return**;

}

**if** (intent.getAction().equals(Intent.*ACTION\_BOOT\_COMPLETED*)) {

File dir = **new** File(context.getCacheDir());

**if** (**null** == dir || !dir.exists() || !dir.isDirectory()) {

**return**;

}

**for** (File file : dir.listFiles()) {

**if** (file.isFile()) {

file.delete(); // 删除所有文件

} **else** **if** (file.isDirectory()) {

deleteDir(); // 递规的方式删除文件夹

}

}

dir.delete();// 删除目录本身

}

}

}

上述代码注册了一个Broadcast Receiver，用于接收启动完成的广播。一旦手机启动完成，则删除临时目录下的文件。

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 敏感数据备份前应先加密**

说明：android:allowBackup属性允许用户对应用数据进行备份。特别是针对大批量的个人数据及用户通信内容等敏感数据，通常都是在数据库中明文存储，一旦被直接备份到不可信环境下（如外部SD卡、云盘等），则可能导致敏感数据泄露、篡改的风险，因此要求敏感数据备份前应先加密。

值得注意的是，若敏感数据已加密存储，备份操作时无需再加密，可直接备份；对于备份到本地的敏感数据，是否加密用户自己去定义（如苹果的本地备份就是有一个加密选项，缺省不加密，客户自己选择）。

**实施指导：**

备份操作需要在AndroidManifest.xml文件application标签下进行配置：

1. 无需备份数据的应用程序应显式设置android:allowBackup="false"
2. 应用程序必须备份数据的情况下，可不设置android:allowBackup属性，或设置android:allowBackup="true"；若涉及备份操作的敏感数据明文存储时，必须先加密后备份。

开发者也可以自实现BackupAgent接口来定义备份那些数据，具体操作请参见：<https://developer.android.com/reference/android/app/backup/BackupAgent.html>

**规则 9.4 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 9.5 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（Android 6.0及之后的版本是在运行时）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系统级应用，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.huawei.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.huawei.permission.TEST权限即可访问。

正确示例：

<permission

android:name=*"com.huawei.permission.TEST"*

android:protectionLevel=*"signature"* >

</permission>

根据业务需求，设置权限保护等级为signature。

**规则 9.6** **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。

**正确示例（root检测代码）：**

/\*\* 判断手机是否root \*/

**public static boolean** isRoot() {

String binPath = "/system/bin/su";

String xBinPath = "/system/xbin/su";

**if** (**new** File(binPath).exists() && isExecutable(binPath))

**return true**;

if (new File(xBinPath).exists() && isExecutable(xBinPath))

**return true**;

**return false**;

}

**private static boolean** isExecutable(String filePath) {

Process p = **null**;

**try** {

p = Runtime.getRuntime().exec("ls -l " + filePath);

// 获取返回内容

BufferedReader in = **new** BufferedReader(**new** InputStreamReader(

p.getInputStream()));

String str = in.readLine();

Log(TAG, str);

**if** (str != **null** && str.length() >= 4) {

**char** flag = str.charAt(3);

**if** (flag == 's' || flag == 'x')

**return true**;

}

} **catch** (IOException e) {

Log(TAG,e.getMessage());

} **finally** {

**if** (p != **null**) {

p.destroy();

}

}

**return false**;

}

这里考虑到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 9.7 禁止使用类加载器方法在不可信路径下加载和输出类**

**说明：**DexClassLoader和pathClassLoader类加载器方法用于Android动态加载dex文件、apk文件和jar文件，输出优化的dex文件。使用类加载器方法时，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。

**错误示例（类加载路径来自不可信的外部输入）：**

**public** **void** loadDexFile(File sourceFile) {

Context context = getApplicationContext();

File file = context.getDir("osdk", 0);// dex临时存储路径

DexClassLoader classLoader = **new** DexClassLoader(

sourceFile.getAbsolutePath(), file.getAbsolutePath(), **null**,

context.getClassLoader());

//… Handle classLoader

}

这里需要加载的类文件sourceFile来自不可信的外部输入，攻击者只需指定一个恶意的类文件，就会导致DexClassLoader类加载劫持攻击。

**错误示例（类加载路径来自不可信的环境）：**

**private** **void** loadDexFile(Context context) {

File sourceFile = **new** File(

Environment.*getExternalStorageDirectory*() + File.*separator*

+ "dextest.jar");// jar文件加载的存储位置

File file = context.getDir("osdk", 0);// dex临时存储路径

DexClassLoader classLoader = **new** DexClassLoader(

sourceFile.getAbsolutePath(), file.getAbsolutePath(), **null**,

context.getClassLoader());

//… Handle classLoader

}

这里需要加载的类文件是来自Environment.getExternalStorageDirectory()（通常是Android设备的外部SD卡）不可信环境下的dextest.jar文件，应用程序只需要声明了WRITE\_EXTERNAL\_STORAGE权限就可以对该环境下的资源进行读写操作；攻击者可以将恶意功能封装仿冒dextest.jar文件，从而实现DexClassLoader类加载劫持攻击。

**正确示例：**

**private** **void** loadDexFile(Context context) {

File sourceFile = **new** File(

context.getDir("secureDirectory", 0) + File.*separator*

+ "dextest.jar");// jar文件加载的存储位置

File file = context.getDir("osdk", 0);// dex临时存储路径

DexClassLoader classLoader = **new** DexClassLoader(

sourceFile.getAbsolutePath(), file.getAbsolutePath(), **null**,

context.getClassLoader());

//… Handle classLoader

}

这里在安全目录下加载dextest.jar文件，这样攻击者就不能通过加载路径和加载环境改变类文件。

需要注意的是，输出类文件到不可信路径也同样会存在类劫持攻击，应确保DexClassLoader方法输出的dex文件也保存在安全目录下。

**规则 9.8 禁止使用危险权限检测方法**

**说明：**checkCallingOrSelfPermission()、checkCallingOrSelfUriPermission()、enforceCallingOrSelfPermission()和enforceCallingOrSelfUriPermission()用来判定调用者是否具备访问某个服务或给定URI所需的权限。但是，若权限检测应用已申请待检测的权限，则利用此类函数可允许缺少相应权限的恶意应用越权访问，导致“混淆代理人攻击”，因此应禁止使用。

权限检测方法应使用checkCallingPermission()、checkCallingUriPermission()、enforceCallingPermission()和enforceCallingUriPermission()来替代。

错误示例：

**int** canProcess =

checkCallingOrSelfPermission("com.example.perm.READ\_INCOMING\_MSG");

**if** (canProcess != PERMISSION\_GRANTED)

**throw** **new** SecurityException();

checkCallingOrSelfPermission()检查自己或者其它调用者是否有com.example.perm.READ\_INCOMING\_MSG权限。但是，其他调用该程序的应用（没有该权限），可以利用该程序的com.example.perm.READ\_INCOMING\_MSG权限来绕过了系统的权限检查。



图9-1 恶意应用可绕过checkCallingOrSelfPermission权限检测

正确示例：

**int** canProcess = checkCallingPermission("com.example.perm.READ\_INCOMING\_MSG");

**if** (canProcess != PERMISSION\_GRANTED)

**throw** **new** SecurityException();

checkCallingPermission方法检查调用者是否有permission权限，如果调用者是自己，那么返回PackageManager.PERMISSION\_DENIED。那么缺乏相应权限的恶意应用程序不能通过混淆代理人攻击的方式获取com.example.perm.READ\_INCOMING\_MSG权限。

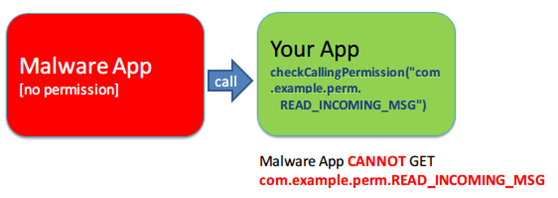


图9-2 恶意应用无法绕过checkCallingPermission权限检测

例外情况：

1. 同时检测调用者和应用本身是否具备访问权限的业务需要
2. 在AIDL Service中使用

其他

**规则 10.1 APK安装包须进行代码混淆**

说明：由于Android应用程序大都是Java代码，非常容易被反编译。代码混淆，为攻击者软件代码的分析设置障碍，增加其分析成本。

对于敏感数据的操作或者重要的关键逻辑处理，推荐放在Native层的C/C++代码实现中实现，因为对C/C++的代码进行反编译难度会比Java的大很多。

实施指导：

一、优先使用DexGuard工具软件进行代码混淆。DexGuard提供了反调试、防篡改、防仿冒、资源加密、Native代码混淆等强大功能。特别针对敏感应用，如游戏类、支付类等，须使用该工具进行混淆

二、次选ProGuard工具进行混淆，根据开发环境不同，相应的使用方式如下：

1. Eclipse+ADT开发环境

Android ADT集成了ProGuard代码混淆代工具，开发人员可利用该工具对代码进行混淆。

Android项目会在项目的根目录下生成一个proguard-project.txt文件，该文件是默认的混淆规则文件。

1、在proguard-project.txt文件中添加混淆的项目规则；

2、将project.properties文件中这段代码；

#proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt

前面的#号去掉即可实现代码混淆打包。

1. Android Studio开发环境

使用Android Studio创建的项目会在项目侧根目录下生成一个proguard-rules.pro文件，此文件便是混淆规则文件。混淆打包的步骤：

1、 在proguard-rules.pro文件中添加混淆的项目规则；

2、在Module下的build.gradle文件中将minifyEnabled false改为minifyEnabled true。如下示例：

buildTypes {

release {

minifyEnabled true

proguardFiles getDefaultProguardFile('proguard-android.txt'),

'proguard-rules.pro'

}

}

例外情况：业务风险评估不需要进行混淆的应用。

**规则 10.2 应用程序只申请业务必要的权限**

说明：应用程序应只申请业务必要的权限，禁止申请与业务无关的权限。如果应用申请了与业务无关的权限，会导致过度授权，违反了权限最小化原则。另外，设置过多的权限可能会适得其反，引起用户的反感和猜疑。例如，对于一个壁纸应用而言，申请短信权限是不合适的；而对于一个媒体播放器应用而言，申请通话权限会遭到用户的质疑。

错误示例：

<uses-permission android:name=*"android.permission.CAMERA"*/>

<uses-permission android:name=*"android.permission.WRITE\_EXTERNAL\_STORAGE"*/>

<!-- 申请发送短信权限 -->

<uses-permission android:name=*"android.permission.SEND\_SMS"*/>

<application

android:allowBackup=*"true"*

android:icon=*"@drawable/ic\_launcher"*

android:label=*"@string/camera"*

android:theme=*"@style/AppTheme"* >…</application>

这里，拍照应用申请了没必要的短信发送权限，易引起用户质疑。

正确示例：

<uses-permission android:name=*"android.permission.CAMERA"*/>

<uses-permission android:name=*"android.permission.WRITE\_EXTERNAL\_STORAGE"*/>

<application

android:allowBackup=*"true"*

android:icon=*"@drawable/ic\_launcher"*

android:label=*"@string/camera"*

android:theme=*"@style/AppTheme"* >…</application>

因此，正确的做法是：只申请与应用业务相关的权限。

若确实需要用到系统敏感功能，应通过调起该敏感服务的方式实现。比如不需申请SEND\_SMS权限也可通过调出短信界面由用户操作发送短信，这样的好处是用户能够知情、可控。

**规则 10.3 禁止自定义权限及行为以系统命名空间开头**

说明：使用系统命名空间开头自定义应用程序的权限或行为，当系统升级的时候，如果新版本系统定义了与应用程序已定义的权限或行为一模一样，安卓PMS（Package Management Service）会默认给应用程序授予系统权限或行为，而不必询问用户是否同意。

禁止应用程序自定义权限以android.permission系统命名空间开头、自定义行为以android.intent.action等系统命名空间开头。自定义权限和行为的命名应按照业务统一要求的进行命名。

错误示例：

<permission android:name=*"android.permission.TEST"*></permission>

这里应用程序在AndroidManifest.xml文件中自定义权限为android.permission.TEST，若下次Android系统升级时定义android.permission.TEST为系统权限，则会默认该应用程序已具有该权限，可能导致权限提升攻击。

正确示例：

<permission android:name=*"com.huawei.permission.TEST"*></permission>

正确示例中自定义权限以com.huawei.permission命名开头。

**规则 10.4 不同应用程序不要设置相同的UID**

说明：通常情况下, Android应用程序运行在自身的Linux进程上,并被分配一个唯一的用户UID,应用程序在运行期间内,用户UID保持不变。对于普通应用程序来说，由于每一个应用程序的UID和权限都不相同，因此能够达到保护敏感数据的目的。

然而，多个应用程序也可以共享一个UID。若多个应用程序使用了相同的签名，并在Manifest配置文件中分配了相同的UID，这就导致不同的应用程序运行在相同的Linux进程中，也就破坏了Android的安全沙箱机制，导致应用程序间资源与权限的共享；这种场景若被恶意攻击者利用，会导致未授权的资源及系统访问。

不同应用程序不要设置相同的UID。如果需要跨信任边界进行数据交互，应首选Intent

**例外情况：**

1. 当APK应用以插件的形式集成到一个应用程序内时，该应用程序及内置的APK插件允许通过设置相同UID的方式运行在同一个进程中；
2. 系统级应用程序

**规则 10.5 对外发布的应用程序须实现自动更新机制**

说明：Android应用程序应具备版本检测和自动更新的功能。应用程序应定期从服务器端获取版本信息，检测到有新版本时，弹窗提示用户更新，用户确认后自动下载和安装。自动更新机制能够快速的提醒用户更新版本，及时消除不安全版本中产品存在的安全问题。

实施指导：

1. 在应用程序启动时，在onCreate()方法中自动联网检测是否有新版本
2. 从服务器读取版本信息文件
3. 获取设备上应用程序版本信息
4. 解析服务端获取的版本信息文件，与客户端应用进行版本的比较
5. 显示更新提示框Dialog

三、下载新版APK文件

1. 实现“更新”按钮事件，将服务端APK文件下载到设备SD卡上
2. 下载完成取消进度条对话框并提示是否安装
3. 安装新版APK文件
4. 安装完成，广播通知删除SD卡上的APK文件

**规则 10.6 应用的保持登陆选项，用户能够自主选择和取消**

**说明：**对于需要与服务端交互的Android应用，不建议提供保持登陆选项，如果业务需要保持登陆或业界同类型应用保持登陆，用户应能够自主选择是否保持登录。

* 如果用户选择不保持登录，该应用需提供会话保护机制，参考[【规则10.7】](#OLE_LINK3)，以降低手机被他人获取从而导致仿冒攻击的风险；
* 如果用户选择保持登录，对涉及现实或虚拟货币及用户隐私的应用，不能仅通过后台发送普通请求来让服务端刷新会话有效期，而应该应使用OAuth2.0的令牌刷新机制或者采用类似的安全机制更换会话信息来防止会话劫持。

**实施指导：**

1、如果产品支持OAuth2.0，应该根据OAuth2.0的官方资料，选择使用如下令牌刷新机制：

1）从后端刷新

OAuth.refreshCredentials(oauthResult, req.session)

.then(function(newOAuthResult) {

//todo with newOAuthResult

})

.fail(function(e) {

//handle an error

});

});

2）从session刷新

OAuth.auth('facebook', req.session, {

force\_refresh: true

})

关于OAuth2.0的资料可以参考[【参考资料6】](#OLE_LINK2)，关于安卓下的OAuth2.0的使用，可以参见[【参考资料7】](#OLE_LINK2)。

2、对于不支持OAuth2.0的产品，也应当采取类似的措施，客户端每次发送保持会话信息的同时，服务端应更换会话信息（sessionID或者Token）。

**规则 10.7 必须对涉及现实或虚拟货币操作的应用提供会话保护机制**

**说明：**对于涉及现实或虚拟货币操作的应用，如银行、股票、支付、网上商店等类型应用，

会话管理过程中支持会话保护机制是保证用户财产和信息安全的必要手段。应用服务程序（包括容器）如果存在不恰当的会话管理，如用户退出系统后，旧的会话继续有效，会导致其他人员可以利用用户身份执行恶意操作。

如下情形需考虑实现会话保护机制：

1.用户主动点击退出按钮或点击关闭程序后，服务器端收到此退出信息，会话立即失效。

2.用户空闲一段时间过后（无请求），会话自动失效。

一、客户端主动发送会话结束请求，使用session.invalidate()来结束会话;

二、对于闲时超时情况，需由服务端实现会话超时检测、客户监听超时信息，具体场景参考如下：

1、服务端实现会话超时检测

a：服务端使用Web容器情况，在容器中配置闲时会话超时时间。具体指导请参见《会话管理安全设计规范1.0》规则4.2以及《Web应用安全开发规范2.0》规则5.6以及相关的容器配置规范。

b：服务端不是Web容器情况，如果网络使用SSL协议通信，使用SSL的会话机制，并在此基础上封装应用层会话：在连接建立后，把当前认证通过的用户信息绑定已经建立会话上，认证不通过则返回错误码给客户端程序，以便客户端重新认证或多次认证失败后释放此连接。

使用SSL协议，通过代码设置会话过期时间：

正确示例：

**int** sessionCacheSize = 0; // zero is no limits

**int** sessionTimeout = 86400; // seconds

SSLContext sslContext = SSLContext.getInstance("TLS");

sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), **null**);

SSLSessionContext sessionContext = sslContext.getServerSessionContext();

**if** (sessionContext != **null**) {

sessionContext.setSessionCacheSize(sessionCacheSize);

sessionContext.setSessionTimeout(sessionTimeout);

}

2、客户监听超时信息，使用HttpURLConnection或[HttpsURLConnection](https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html)连接组件情形下，可以使用Cookie维持会话。

正确示例（客户端使用监听器监听服务程序的会话超时消息）：

URL url = **new** URL("http://example.com/someop");

HttpURLConnection con= (HttpURLConnection) url.openConnection();

// 此处是getSessionId(con)的实现:

String cookieVal = con.getHeaderField("set-cookie");

String tempSessionId;

**if**(cookieVal != **null**) {

tempSessionId = cookieVal.substring(0, cookieval.indexOf(";"));

}

// sessionId是客户端登录认证通过后记录的会话ID或初始值空，同时判断getResponseCode有效

**if** (con.getResponseCode() == HttpServletResponse.SC\_OK && sessionId.equals(tempSessionId)) { // 服务器返回值为200，或判断HTTP状态码2XX

// 会话继续有效，处理服务器返回结果

InputStream inputStream = httpURLConnection.getInputStream();

//...

} **else** {

//服务器返回值为非200，可详细判断，比如401未授权

// 会话过期，得到新的tempSessionId

//提示重新登录界面或解锁界面

// 验证登录或解锁信息通过后，重新读取服务器产生的新的会话ID，赋值给sessionId

// sessionId = getSessionId(con);

//...

}

另外，客户端也可通过本地的定时器自动判断会话是否超时，超时后定位到解锁或登录页面。

**规则 10.8 必须对涉及现实或虚拟货币交易操作提供重认证**

说明：当用户登录后，做交易操作时，一旦输入了密码，交易即会开始。因此，对于虚拟或现实交易类应用在执行交易操作时，需要进行身份重认证，如采用动态交易口令、静态交易口令、短信验证码、指纹等，具体指导请参见《身份与访问管理安全设计规范》规则2.5.1。

身份重认证包括但不限于以下场景：

1.用户支付某产品或服务

2.用户预订某产品或服务

3.用户资金转移操作

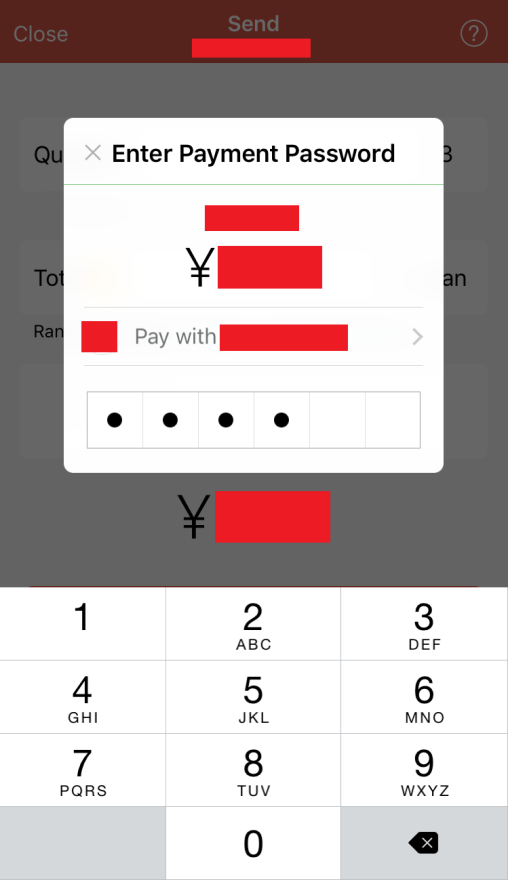


图10-1 启用支付口令确认支付

参考资料

1. Android Application Secure Design/Secure Coding Guidebook, <http://www.jssec.org/dl/android_securecoding_en.pdf>

2. CERT [Android Secure Coding Standard](https://www.securecoding.cert.org/confluence/display/android/Android+Secure+Coding+Standard?src=breadcrumbs), <https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=111509535>

3. CWE&SANS TOP 25, <http://www.sans.org/top25-software-errors/>

4. Fortify [A Taxonomy of Coding Errors that Affect Security](http://www.hpenterprisesecurity.com/vulncat/en/vulncat/intro.html), <http://www.hpenterprisesecurity.com/vulncat/en/vulncat/index.html>

5. 合作方交付件 Android Security Guide，<http://3ms.huawei.com/hi/group/9741/file_5132983.html>（英文原版）

<http://3ms.huawei.com/hi/group/9741/file_5132985.html>（中文版）



6. 关于令牌刷新的OAuth2.0官方资料，

<http://tools.ietf.org/html/rfc6749#section-1.5>

7. OAuth2.0在Android环境的应用示例，

<http://docs.oauth.io/?Android>

8. IntentScheme白皮书，

<http://www.mbsd.jp/Whitepaper/IntentScheme.pdf>

9.针对window设置secure flag的官方资料

<https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html>