vivo移动应用安全编码基础规范

说明

配合公司《vivo移动终端和互联网应用安全基线》,以规范移动应用(以下简称移动App或App)的编码安全管理要求。

本规范为vivo企业强制性规定，主要目的在于提升vivo移动应用App基本安全防护能力，从设计初始阶段即导入基本信息安全概念，通过规范的重点要项，提醒App开发者强化信息安全意识，并逐步完善自身App安全防护能力。

本规范由vivo安全工程组提出并归口管理。

文档修订历史

版本	日期	文档修订	修订备注	起草专家	评审专家
0.1	2017-08-07	LuJinghui	新增		
1.0	2017-08-31	YeLingjun	起草	叶灵军 规范框架大纲及部份条目 马帅 组件安全 杨杰 组件安全 莫燕清 数据存储安全 张建平 加解密安全规则 郑伟滨 网络安全 杨凯 数据存储安全	曹春阳；陈辉-软件平台；陈匡斐；陈凌云；陈忠迅；胡吉祥；刘坤-软件项目；刘全才；马冬梅；王述磊；吴华琛；袁志强；张名凯；周孟；李广；刘绪森；孟庆杰；占美全；吴文沧；肖方旭；莫燕清；杨杰
1.1	2017-10-28	LuJinghui	校对，格式调整		
1.2	2018-10-24	YeLingJun	修订版本 新增： 三方合作安全	陈辉军、陈凌云、崔小帆、蒋智勇、焦四辈、李仁军、马帅、莫燕清、莫燕清、任勇勋、杨振涛、袁云丽、叶灵军	
1.3	2018-10-29	YeLingJun	整理内容，输出内审稿		
1.4	2018-11-07	YeLingJun	评审后整理内容		陈辉军、王江胜、李仁军、钱钰、蒋智勇、焦四辈、黎小红、叶灵军

2.0	2018-11-10	Yelingjun	版本调整到2.0，发布		

1 前言

移动应用App开发项目周期短、更新快，通过本规范，将安全软件发展生命周期(Secure Software Development Life Cycle, SSDLC)流程简化，结合敏捷式开发，并总结归纳移动应用App开发实现及安全代码范例供开发人员选用，设计及开发出符合公司安全基线移动应用app，避免因开发疏忽造成存在的信息安全漏洞及可防范恶意攻击的 App安全程序，以保护用户敏感数据。

1.1 适用对象

- 项目经理(Project Manager, PM)
- 系统分析人员(System Analyst, SA)
- 系统设计人员(System Designer, SD)
- 程序员(Programmer, PG)
- 信息安全人员(Information Security, IS)
- 软件检测人员(Quality Assurance, QA)
- 数据库管理员(Data Base Administrator, DBA)
- 信息运维人员(Information System Operator, ISO)
- 系统运维人员（System Operator）

1.2 引用标准

- 工业和信息化部《移动终端安全能力设计准则》
- 工业和信息化部《YD/T 2407-2013 移动终端安全能力技术要求》
- 工业和信息化部《YD/T 2408-2013 移动终端安全能力测试方法》
- 工业和信息化部《YD/T 1886-2009 移动终端安全技术要求和测试方法》
- GBT 35273-2017 信息安全技术 个人信息安全规范
- 美国国家标准技术研究所（NIST）FIPS 199 《联邦信息和信息系统安全分类标准》
- 美国国家标准技术研究所（NIST）FIPS 200 《联邦信息系统最小安全控制标准》
- 美国国家标准技术研究所（NIST）NIST Special Publication 800-163《移动应用App安全审核》
- 美国国家标准技术研究所（NIST）《加密算法验证项目》（CAVP), NIST
- 欧洲网络与信息安全局(ENISA)《智能手机开发人员安全开发指南》（SSDGAD）
- OWASP Mobile Security Project - Top Ten Mobile Risks
- ISO/IEC 14598:2001 (Information technology - Software product evaluation)
- ISO/IEC TR 9126-4:2004 (Software engineering - Product quality)
- 《vivo用户数据分类分级规范》
- 《vivo软件产品隐私保护要求和设计指南》

1.3 使用建议

本规范以移动应用开发人员为主要对象，但移动应用开发项目会因App的需求目的、目标用户、项目计划、及与其他服务得集成等因素，必须与需求提出人员、项目经理、系统分析、系统设计、测试/品质及信息安全人员等不同角色进行协同合作。

2 适用范围

本规范主要针对移动应用程序在移动设备端的安全提出基本的安全编码要求，本规范适用于移

动应用程序所需、具有共同性、相类似的基础功能，例如数据储存、传输保护机制或使用者身分认证机制等。

3 术语

- **移动应用程序（Mobile Application）**

设计给智能手机、平板电脑和其他移动设备使用的应用程序。

- **移动应用程序商店（Application Store）**

移动设备用户通过预装在设备中的移动应用店或通过网站对应用程序、音乐、杂志、书籍、电影、电视节目进行浏览、下载或购买。

- **敏感数据（Sensitive Data）**

用户行为或移动应用程序的运行，移动设备及其附属储存介质建立、储存或传输的信息，而该信息的泄漏有对用户造成损害的可能性，包括但不限于个人信息、应用程序配置信息、黑白名单、生成的中间数据、运行日志等。

- **个人信息（Personal Information）**

以电子或者其它方式记录的能够单独或者与其它信息结合识别特定自然人身份或者反映特定自然人活动情况的各个信息。

个人信息包括姓名、出生日期、身份证件号码、个人生物识别信息、住址、通信信讯联系方式、通信记录和内容、账号密码、财产信息、征信息信息、行踪轨迹、住宿信息、健康生理信息、交易信息等。

所有可以直接或间接方式识别自然人的数据都是个人信息，包括自然人的姓名、出生年月日、身分证编号、护照号码、特征、指纹、婚姻、家庭、教育、职业、性取向、病历、医疗、基因、健康数据、联络方式、财务情况、社会活动、国际移动设备标识符（International Mobile Equipment Identity，IMEI）、国际行动用户标识符（International Mobile Subscriber Identity，IMSI）及其他得以直接或间接方式识别该个人的数据。其他根据不同国家法律法规的合规性性要求归属于个人信息的数据。

了解更多请查阅《GBT 35273-2017 信息安全技术 个人信息安全规范》或《vivo用户数据分类分级规范》。

- **个人敏感信息（Personal Sensitive Information）**

一旦泄露、非法提供或滥用可能危害人身和财产安全、极易导致个人名誉、身心健康受到损害或歧视性待遇等的个人信息。

个人敏感信息包括身份证号码、个人生物识别信息、银行账号、通信记录和内容、财产信息、征信信息、行踪轨迹、住宿信息、健康生理信息、交易信息、14岁以下（含）儿童的个人信息等。

了解更多请查阅《GBT 35273-2017 信息安全技术 个人信息安全规范》或《vivo用户数据分类分级规范》。

- **口令（Password）**

能让用户使用系统或用以识别用户身分的字符串，本文中一般是指用户输入的密码字符串。

- **付费资源（Payment Resource）**

通过移动应用程序购买功能取得的额外功能、内容及订阅项目。

- **会话标识符（Session Identification，Session ID）**

指在建立联机事务时，分配给该联机事务的标识符，并做为联机期间的唯一标识符；当联机结束时，该标识符可释出并重新指派给新的联机事务。

- **服务器证书（Server Certificate）**

载有签名验证数据，提供服务器身分鉴别及数据传输加密。

- **发证机构（Certificate Authority）**

签发证书的部门、法人。

- **恶意代码（Malicious Code）**

在未经使用者同意的情况下，侵害使用者权益，包括但不限于任何具有恶意特征或行为的程序代码。

- **信息安全漏洞（Vulnerability）**

指移动应用程序安全方面的缺陷，使得系统或移动应用程序数据的保密性、完整性、可用性面临威胁。

- **函数库（Library）**

指将一些复杂或者牵涉到硬件层面的程序包装成函数（Function）或对象（Object）收集在一起，编译成二进制代码提供程序设计者使用。

- **注入攻击（Code Injection）**

指因移动应用程序设计缺陷而执行用户所输入的恶意指令，包括但不限于命令注入（Command Injection）、SQL注入（SQL Injection）。

- **基于用户口令加密（Password-based encryption）**

基于用户口令的加密方法，PBE（Password-based encryption）。

4 移动应用安全技术要求

参照《vivo 移动终端和互联网应用安全基线》。

5 本文命名规则

5.1 规则名称

ADRD-xxx，ADRD 代表 Android 代码规则，xxx 是规则序号
JAVA-xxx，JAVA 代表 Java 代码规则，xxx 是规则序号
C-xxx， C 代表 C 代码规则，xxx 是规则序号
CPP-xxx，CPP 代表 C++代码规则，xxx 是规则序号

5.2 动词约定

禁止	编码人员坚决不能出现的做法
应	本规范推荐使用的做法
不应	编码过程中不得出现的做法
可以	可以，允许，如执行人有更好的做法也能采用

6 安全编码规则

6.1 组件安全

6.1.1 组件使用总体原则

- **最小化组件暴露**

不参与跨应用调用的组件添加 android:exported="false"属性，这个属性说明它是私有的，只有同一个应用程序的组件或带有相同用户 ID 的应用程序才能启动或绑定该服务。

- **设置组件访问权限**

对参与跨应用调用的组件或者公开的广播、服务设置权限。只有具有该权限的组件才能调用这个组件。含有联网事件的组件，对外安全性务必添加相应的权限访问限制和权限等级要求。

应用程序内部的组件应设置为 Private，请参阅本章以下规则：

ADRD-03. 应用内部的 content provider 应设置为 Private

ADRD-11. 应用内部的 activity 应设置为 Private

ADRD-16. 应用程序内部的广播接收者应设置为 Private

ADRD-18. 应用程序内部的 Service 应置为 Private

- **暴露组件的代码检查**

Android 提供各种 API 在运行时检查、执行、授予和撤销权限。这些 API 是 android.content.Context 类的一部分，这个类提供有关应用程序环境的全局信息。

6.1.2 ADRD-01. 限制应用的敏感 content provider 访问

ContentProvider 类提供了和其他应用管理和分享数据的机制. 在分享一个内容提供者的数据和其他应用时，需要谨慎使用访问控制移阻止对敏感数据的非授权访问。

有三种方式限制对内容提供者的访问：

- Public
- Private
- Restricted access

6.1.2.1 Public

通过在 AndroidManifest.xml 文件中指定 android:exported 属性，一个 content provider 可以被设定为对其他应用公开. 对于 API Level 16 以前的 Android 应用，一个 content provider 的缺省设定为 public 除非指定 android:exported="false". 举例，

```
1. <provider android:exported="true" android:name="MyContentProvider" android:au
thorities="com.example.mycontentprovider" />
```

如果一个content provider设定为public，则存储在这个provider中的数据可被其他应用访问，因此它应被设计为只能处理非敏感数据。

6.1.2.2 Private

通过将AndroidManifest.xml设定android:exported的属性将一个content provider设置为private. 对于API Level 17以后的Android应用，一个content provider的缺省设定为private. 举例，

```
1. <provider android:exported="false" android:name="MyContentProvider" android:a
uthorities="com.example.mycontentprovider" />
```

如果你不希望共享一个content provider和其他的应用，应在manifest文件中申明 android:exported="false"。需要注意的是，在API Level 8之前，即使设定 android:exported="false"，content provider依然可以被其他应用访问。

6.1.2.3 Restricted Access

组件在声明所需要权限时，一定要对权限及其protectionLevel进行定义。

在声明所需要权限时，对权限及其protectionLevel进行定义（建议至少是signature级别），可防范其他应用恶意定义权限导致权限控制措施失效的问题。请参阅6.1.26 ADRD-25. 组件在声明

权限时，要对权限及其protectionLevel进行定义。

6.1.2.4 不合规代码示例

下面的代码中AndroidManifest.xml没有android:exported属性，在API Level 16之前的程序中，content provider被设定为public:

AndroidManifest.xml

```
1. <provider android:name=".content.AccountProvider" android:authorities="jp.co.vulnerable.accountprovider" />
```

Java Code

```
1. // check whether movatwi is installed.
2. try {
3.   ApplicationInfo info = getPackageManager().getApplicationInfo("jp.co.vulnerable", 0);[cjl5]
4. } catch (NameNotFoundException e) {
5.   Log.w(TAG, "the app is not installed.");
6.   return;
7. }
8. // extract account data through content provider
9. Uri uri = Uri.parse("content://jp.co.vulnerable.accountprovider");
10. Cursor cur = getContentResolver().query(uri, null, null, null, null);[cjl6]
11. StringBuilder sb = new StringBuilder();
12. if (cur != null) {
13.   int ri = 0;
14.   while (cur.moveToNext()) {
15.     ++ri;
16.     Log.i(TAG, String.format("row[%d]:", ri));
17.     sb.setLength(0);
18.     for (int i = 0; i < cur.getColumnCount(); ++i) {
19.       String column = cur.getColumnName(i);
20.       String value = cur.getString(i);
21.       if (value != null) {
22.         value = value.replaceAll("[\r\n]", "");
23.       }
24.       Log.i(TAG, String.format("\t%s:\t%s", column, value));
25.     }
26.   }
27. } else {
28.   Log.i(TAG, "Can't get the app information.");
29. }
```

6.1.2.5 合规代码示例

下面的例子在AndroidManifest.xml中将content provider设定为private，其他的应用不可以访问到数据：

```
1.  <provider android:name=".content.AccountProvider" android:exported="false" an
droid:authorities="jp.co.vulnerable.accountprovider" />
```

6.1.3 ADRD-02. Public ContentProvider 的访问

Public ContentProvider 根据使用场景的不同，可分为 public Provider(公共 Provider)，Partner Provider(合作 Provider)，IN-House Provider(内部 Provider)。

6.1.3.1 Public Provider

没有特定的使用者，只要匹配上 URL 在很多地方都可以使用，Provider 很容易被伪造，所以很容易被恶意软件攻击。使用 Public Provider 要注意以下两点：

（1）不要返回敏感信息。

（2）细心处理 UriMatcher 匹配的 URI。

（3）使用参数化查询的方式，SQLiteDatabase 的增删改查方法是安全的，禁止直接拼接 SQL 语句进行执行，请参阅 6.2.9ADRD-09. 使用参数化的 SQL 语句防止 SQL 注入攻击。

```
1.  public class PublicProvider extends ContentProvider {
2.      // ……
3.      @Override
4.      public boolean onCreate() {
5.          return true;
6.      }
7.      // ……
8.      @Override
9.      public Cursor query(Uri uri, String[] projection, String selection,
10.             String[] selectionArgs, String sortOrder) {
11.         // *** POINT 2 ***细心处理 UriMatcher 匹配的 URI
12.         //UriMatcher#match() and switch case。
13.         // *** POINT 1 ***不要返回敏感信息
14.         switch (sUriMatcher.match(uri)) {
15.         case DOWNLOADS_CODE:
16.         case DOWNLOADS_ID_CODE:
17.             return sDownloadCursor;
18.         case ADDRESSES_CODE:
19.         case ADDRESSES_ID_CODE:
20.             return sAddressCursor;
21.         default:
22.             throw new IllegalArgumentException("Invalid URI:" + uri);
23.         }
24.     }
25.     // ……
26. }
```

6.1.3.2 Partner Provider

不具备相同签名的不同应用，希望互通使用 Content Provider 访问对方的数据。其使用的场景一般是针对不同公司业务线产品，希望公开自我的 Content Provider，达到业务上的互相合作。

要注意以下几点：

(1) 验证请求应用证书是否在白名单中；

(2) 细心处理 UriMatcher 匹配的 URI。

```
1.  public class PartnerProvider extends ContentProvider {
2.      // ……
3.      @Override
4.      public boolean onCreate() {
5.          return true;
6.      }
7.      @Override
8.      public Cursor query(Uri uri, String[] projection, String selection,
9.              String[] selectionArgs, String sortOrder) {
10.         // *** POINT 1 ***验证请求应用证书是否在白名单中；
11.         if (!checkPartner(getContext(), getCallingPackage(getContext()))) {
12.             throw new SecurityException("Calling application is not a partner
 application.");
13.         }
14.         // *** POINT 2 **细心处理 UriMatcher 匹配的 URI，不要把不相关的数据也返回给
调用者
15.         switch (sUriMatcher.match(uri)) {
16.         case DOWNLOADS_CODE:
17.         case DOWNLOADS_ID_CODE:
18.             return sDownloadCursor;
19.         case ADDRESSES_CODE:
20.         case ADDRESSES_ID_CODE:
21.             return sAddressCursor;
22.         default:
23.             throw new IllegalArgumentException("Invalid URI:" + uri);
24.         }
25.     }
26.     // ……
27. }
```

6.1.3.3 IN-House Provider

内部 Content Provider，其使用的场景一般是同一个企业不同产品线之间的交互，这些应用之前都具有相同内部签名。

如我们系统应用都具有同样的签名，如果做只有签名相同限制的话，第三方应用就不可以访问。

主要注意一下几点：

(1) 定义一个内部 signature permission；

(2) 设置 Content Provider signature permission 属性；

(3) 验证 signature permission 是否是在内部定义的；

(4) 细心处理 UriMatcher 匹配的 URI;
(5) Return 可以返回敏感信息;
(6) 导出 apk 时和请求应用使用同样的签名。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.provider.inhouseprovider">
    <!-- *** POINT 1 *** 定义一个内部 signature permission -->
    <permission
        android:name="org.jssec.android.provider.inhouseprovider.MY_PERMISSION"
        android:protectionLevel="signature" />
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <!-- *** POINT 2 *** 设置 Content Provider signature permission 属性 -->
        <provider
            android:name=".InhouseProvider"
            android:authorities="org.jssec.android.provider.inhouseprovider"
            android:permission="org.jssec.android.provider.inhouseprovider.MY_PERMISSION"
            android:exported="true" />
    </application>
</manifest>
```

InHouse Provider

```
public class InhouseProvider extends ContentProvider {
    // ……
    // UriMatcher
    private static final int DOWNLOADS_CODE = 1;
    private static final int DOWNLOADS_ID_CODE = 2;
    private static final int ADDRESSES_CODE = 3;
    private static final int ADDRESSES_ID_CODE = 4;
    private static UriMatcher sUriMatcher;
    static {
        sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        sUriMatcher.addURI(AUTHORITY, Download.PATH, DOWNLOADS_CODE);
        sUriMatcher.addURI(AUTHORITY, Download.PATH + "/#", DOWNLOADS_ID_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH, ADDRESSES_CODE);
        sUriMatcher.addURI(AUTHORITY, Address.PATH + "/#", ADDRESSES_ID_CODE);
    }
```

```
    // ……
    private static final String MY_PERMISSION = "org.jssec.android.provider.inhouseprovider.MY_PERMISSION";
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
        }
        return sMyCertHash;
    }
    @Override
    public boolean onCreate() {
        return true;
    }
    // ……
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {
        // *** POINT 3 *** 验证 signature permission 是否是在内部定义的
        if (!SigPerm.test(getContext(), MY_PERMISSION, myCertHash(getContext()))) {
            throw new SecurityException("The in-house signature permission is not declared by in-house application.");
        }
        // *** POINT 4 ***细心处理 UriMatcher 匹配到的 URI
        // *** POINT 5 *** 可以 return 敏感信息
        switch (sUriMatcher.match(uri)) {
        case DOWNLOADS_CODE:
        case DOWNLOADS_ID_CODE:
            return sDownloadCursor;
        case ADDRESSES_CODE:
        case ADDRESSES_ID_CODE:
            return sAddressCursor;
        default:
            throw new IllegalArgumentException("Invalid URI:" + uri);
        }
    }
    // ……
}
```

6.1.4 ADRD-03. 应用内部的 content provider 应设置为 Private

仅在单个应用程序中使用的provider是不需要被其他应用程序访问的，由于系统默认把provider设置为public，恶意程序有可能对content provider造成攻击。

6.1.4.1 不合规示例

6.1.4.2 合规示例

6.1.4.3 编程建议

仅在单个应用程序中使用的内容提供者应被明确设置为private，在Android 2.3.1（API Level 9)或更高版本中，内容提供者可以通过在provider中指定android:exported="false"来把provider设置为私有。

AndroidManifest.xml

```
<!-- *** POINT 1 *** Set false for the exported attribute explicitly. -->
<provider
android:name=".PrivateProvider"
android:authorities="org.jssec.android.provider.privateprovider"
android:exported="false" />
```

6.1.5 ADRD-04. 设置 content provider 的访问权限

6.1.5.1 Provider 的读写权限

一个Provider里面可能有私有数据，也有公有数据。有些数据可以让别人修改，有些不能让别人修改。为防止恶意程序获取原本不应该外泄的数据，或篡改数据，Provider就需要设置读权限（android:readPermission），和写权限（android:writePermission），或者都设置（android:permission），也可以通过自定义权限来精确设置权限。

```
<provider android:name=".MyProvider" android:authorities="mytest.testProvider
"
    android:readPermission="test.provider.permission"
    android:multiprocess="true">
    <grant-uri-permission android:pathPrefix="/user/" />
</provider>
```

- 自定义权限

```
<application
    android:exported="true"
    android:label="@string/app_name"
    android:icon="@drawable/ic_launcher">
    <provider
        android:name="MyProvider"
        android:authorities="com.vivo.myprovider"
        android:permission="com.vivo.permission.myprovider">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
```

```
    </provider>
</application>

<permission
    android:name="com.vivo.permission.myprovider"
    android:protectionLevel="normal"
    android:label="@string/permission_label"
    android:description="@string/permission_description"
    />
```

在需要定义权限的ContentProvider中声明权限android:permission="com.vivo.permission.myprovider"，再定义permission。permission对应的name与声明的权限对应。

6.1.5.2 Provider 的 URI 权限

Provider是通过URI来识别需要操作的数据是什么，数据的限制就需要体现在对URI的控制上。实际使用中，provider可以只开放部分URI的权限，例如，我们可以只开放content://cn.wei.flowingflying.propermission.PrivProvider/hello路径下权限，不允许访问其他路径，如下声明：

```
<provider android:name=".PrivProvider"
    android:authorities="com.vivo.weather.provider"
    android:readPermission="wei.permission.READ_CONTENTPROVIDER"
    android:exported="true" >
    <path-permission android:pathPrefix="/hello" android:readPermission="READ_HELLO_CONTENTPROVIDER" />
</provider>
```

可以针对其中某个或某部分URI，单独进行权限设置。除了android:pathPrefix，还可以有android:path和android:pathPatten，例如android:pathPattern="/hello/.*"（注意，通配符*之前有个‘.’）。

6.1.6 ADRD-05. 规范 content provider 的 URL

通过使用ContentProvider.openFile()方法，可以为另一个应用程序提供访问应用程序数据（文件）的功能。根据ContentProvider的实现，使用该方法可能会导致目录遍历漏洞。因此，当通过provider交换文件时，路径应在使用之前进行规范化。

接受不受信任输入的应用程序应该在验证输入是否合法之前规范输入。规范化很重要，因为在Unicode中，相同的字符串可以有许多不同的表示。根据Unicode标准[Davis 2008]，附录15《Unicode规范化表格》（参考：http://unicode.org/reports/tr15/）所述，当实现将字符串保持为规范化形式时，可以确保等效字符串具有唯一的二进制表示形式。

6.1.6.1 不合规示例

该示例尝试通过调用 android.net.Uri.getLastPathSegment()从路径 paramUri 中检索最后一个段，该段 paramUri 应该表示一个文件名。该文件在预先配置的父目录 IMAGE_DIRECTORY 中被访问。示例尝试在执行归一化之前验证字符串。

```
private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();
```

```
2.  public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
3.      throws FileNotFoundException {
4.    File file = new File(IMAGE_DIRECTORY, paramUri.getLastPathSegment());
5.    return ParcelFileDescriptor.open(file, ParcelFileDescriptor.MODE_READ_ONLY);


6.  }
```

当这个文件路径被 URL 编码后，就意味着被访问的这个文件可能会存在于预配置的父目录之外的一个不可预知的目录中。

从 Android 4.3.0_r2.2 开始，Uri.getLastPathSegment()方法在内部调用了 Uri.getPathSegments()：

```
1.  public String getLastPathSegment() {
2.    // TODO: If we haven't parsed all of the segments already, just
3.    // grab the last one directly so we only allocate one string.
4.    List<String> segments = getPathSegments();
5.    int size = segments.size();
6.    if (size == 0) {
7.      return null;
8.    }
9.    return segments.get(size - 1);
10. }
```

Uri.getPathSegments()方法的部分代码如下：

```
1.  PathSegments getPathSegments() {
2.    if (pathSegments != null) {
3.      return pathSegments;
4.    }
5.    String path = getEncoded();
6.    if (path == null) {
7.      return pathSegments = PathSegments.EMPTY;
8.    }
9.    PathSegmentsBuilder segmentBuilder = new PathSegmentsBuilder();
10.   int previous = 0;
11.   int current;
12.   while ((current = path.indexOf('/', previous)) > -1) {
13.     // This check keeps us from adding a segment if the path starts
14.     // '/' and an empty segment for "//".
15.     if (previous < current) {
16.       String decodedSegment = decode(path.substring(previous, current));
17.       segmentBuilder.add(decodedSegment);
18.     }
19.     previous = current + 1;
20.   }
21.   // Add in the final path segment.
22.   if (previous < path.length()) {
23.     segmentBuilder.add(decode(path.substring(previous)));
24.   }
25.   return pathSegments = segmentBuilder.build();
```

```
26. }
```

Uri.getPathSegments() 方法首先通过调用 getEncoded()获取了文件路径，然后使用"/"作为分隔符将路径分割成几部分，任何被编码的部分都将通过 decode()方法进行 URL 解码。

如果文件路径被 URL 编码，那么分隔符就变成了"%2F"，而不再是"/"，getLastPathSegment()就可能不会正确地返回路径的最后一段，从而导致目录遍历漏洞的产生。

如果 Uri.getPathSegments()在进行路径分割之前对路径进行解码，那么经过 URL 编码的路径就会被正确地处理，不过并没有这么实现。

以下恶意代码可对以上不合规代码示例中的漏洞进行利用：

```
1.  String target = "content://com.example.android.sdk.imageprovider/data/" +
2.    "..%2F..%2F..%2Fdata%2Fdata%2Fcom.example.android.app%2Fshared_prefs%2FExam
ple.xml";
3.
4.  ContentResolver cr = this.getContentResolver();
5.  FileInputStream fis = (FileInputStream)cr.openInputStream(Uri.parse(target));
6.
7.  byte[] buff = new byte[fis.available()];
8.  in.read(buff);
```

6.1.6.2 合规示例

该解决方案在验证字符串之前对其进行规范化。字符串的替代表示被标准化为规范的尖括号。因此，输入验证正确地检测到恶意输入并引发 IllegalStateException。

```
1.  private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();
2.  public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
3.    throws FileNotFoundException {
4.      File file = new File(IMAGE_DIRECTORY, Uri.parse(paramUri.getLastPathSegme
nt()).getLastPathSegment());
5.      return ParcelFileDescriptor.open(file, ParcelFileDescriptor.MODE_READ_ONL
Y);
6.  }
```

第一次调用Uri.getLastPathSegment()函数，字符串通过Uri.parse()转换成了Uri对象第二次调用Uri.getLastPathSegment()，得到的结果是Example.xml，如果攻击者提供了一个特殊的字符串，该字符串在第一次调用Uri.getLastPathSegment()时不能被解码，那么就获取不到分割路径的最后一段。

在下述解决方案中，在使用文件路径前通过Uri.decode()对其进行了解码。同样的，在文件对象创建后，通过调用File.getCanonicalPath()将路径进行了规范，同时检查它是否存在于IMAGE_DIRECTORY目录中。

通过使用规范化后的路径，即使文件路径被双重编码，目录遍历漏洞也可以得到缓解。

```
1.  private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();
2.    public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
3.        throws FileNotFoundException {
4.      String decodedUriString = Uri.decode(paramUri.toString());
5.      File file = new File(IMAGE_DIRECTORY, Uri.parse(decodedUriString).getLast
PathSegment());
```

```
    if (file.getCanonicalPath().indexOf(localFile.getCanonicalPath()) != 0) {
      throw new IllegalArgumentException();
    }
    return ParcelFileDescriptor.open(file, ParcelFileDescriptor.MODE_READ_ONLY);
  }
```

6.1.6.3 编程建议

ContentProvider.openFile()方法提供了一种方便其它应用程序访问自己的数据(文件)的方式，但是使用这个方法会导致一个目录遍历漏洞。因此，当通过ContentProvider访问一个文件的时候，路径应该被规范化。

我们建议的做法是包括：

- 过滤限制跨域访问，对访问的目标文件的路径进行有效判断，使用ContentProvider.openFile()之前需要调用Uri.decode()先对Content Query Uri进行解码，再过滤如可通过“../”实现任意可读文件的访问的Uri字符串。
- 设置exported=“false”，这样就减少了很大一部分的攻击源，参考规则ADRD-02
- 给provider设置恰当的访问权限，参考ADRD-03
- 大部分开放的Provider，是提供给公司的其他应用使用的，打包签名APP的签名证书是一致的，Provider的android:protectionLevel应该设置为“signature”
- 去除没有必要的openFile()接口，如果应用的Content Provider组件没有必要实现openFile()接口，建议移除该Content Provider的不必要的openFile()接口
- 设置权限来进行内部应用通过Content Provider的数据共享

6.1.7 ADRD-06. 规范 content provider 的增删改查

传递给 ContentProvider 的参数应该被视为不可信的，不能直接用于 sql 查询。禁止直接执行拼接后的 SQL 语句，应该使用参数化查询的方式，以避免 SQL 注入。

6.1.7.1 不合规示例

```
// 拼接 SQL 语句直接用于查询
String sql = "select * from user where id='" + userInputID +"'";
Cursor curALL = db.rawQuery(sql, null);
// 拼接 SQL 语句直接执行
String sql = "INSERT INTO user values('"+userInputID +"','"+userInputName +"')";// 错误同上
db.execSQL(sql);
```

6.1.7.2 合规示例

```
// 使用 SQLiteStatement 绑定参数
SQLiteStatement sqLiteStatement = db.compileStatement("insert into msgTable(uid, msg) values(?, ?)");
sqLiteStatement.bindLong(1, 12);
sqLiteStatement.bindString(3, "text");
long newRowId = sqLiteStatement.executeInsert();
```

```
// 使用 SQLiteDatabase 的增删改查方法
DatabaseHelper dbhelper = new DatabaseHelper(SqliteActivity.this,"sqliteDB");

SQLiteDatabase db = dbhelper.getReadableDatabase();
// 查
Cursor cur = db.query("user", new String[]{"id","name"}, "id=? and name=?", new String[]{userInputID,userInputName}, null, null, null);
// 增
ContentValues val = new ContentValues();
val.put("id", userInputID);
val.put("name", userInputName);
db.insert("user", null, val);
// 改
ContentValues val = new ContentValues();
val.put("id", userInputName);
db.update("user", val, "id=?", new String[]{userInputID });
// 删
db.delete("user", "id=? and name=?", new String[]{userInputID , userInputName });
```

6.1.7.3 编程建议

在使用 Provider 进行增删改查的代码设计时应该注意避免直接构造 SQL 语句进行执行，因为从 Provider 的各个接口中传递进来的参数可能是人为精心伪造的，使用不可信的参数进行 SQL 语句构造会导致执行预期之外的 SQL 操作，进而导致数据安全问题。

我们建议的做法是直接使用 SQLiteDatabase 的增删改查方法或者使用 SQLiteStatement 进行参数绑定后执行。

6.1.8 ADRD-07. 限制敏感 activity 的访问

在 Android 上，为 AndroidManifest.xml 文件中的活动声明 intent filter 意味着该活动可能会导出给其他应用程序使用。如果活动仅用于应用程序的内部使用，却声明 intent filter，则任何其他应用程序（包括恶意软件）都可以激活该 activity 以进行恶意使用。导致敏感信息泄露，并可能受到绕过认证、恶意代码注入等攻击风险。

有三种方式限制对activity的访问：

- Public
- Private
- Restricted access

6.1.8.1 Public

通过在AndroidManifest.xml文件中指定android:exported属性，一个Activity可以被设定为对其他应用公开。

```
<activity
    android:name=".PublicActivity"
    android:label="@string/app_name"
```

```
    android:exported="true">
        <intent-filter>
        <action android:name="org.jssec.android.activity.MY_ACTION" />
        <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
</activity>
```

如果一个Activity设定为public,可以被其他的应用启动,所以在Intent中要注意敏感信息的传递。

6.1.8.2 Private

通过将AndroidManifest.xml设定android:exported的属性将一个Activity设置为private。这样的Activity只能内部调用，也是最安全的方式。

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

6.1.8.3 Restricted Access

组件在声明所需要权限时，一定要对权限及其 protectionLevel 进行定义。

在声明所需要权限时，对权限及其 protectionLevel 进行定义（建议至少是 signature 级别），可防范其他应用恶意定义权限导致权限控制措施失效的问题。请参阅 6.1.26 ADRD-25. 组件在声明权限时，要对权限及其 protectionLevel 进行定义。

6.1.8.4 不合规代码示例

利用应用程序中的漏洞，通过启动该 activity，另一个无权访问 SD 卡或网络的应用程序可以上传图像或使用该应用用户的帐户存储在 SD 卡上的电影到社交网络服务。

```
<activity android:configChanges="keyboard|keyboardHidden|orientation" android:name=".media.yfrog.YfrogUploadDialog" android:theme="@style/Vulnerable.Dialog" android:windowSoftInputMode="stateAlwaysHidden">
    <intent-filter android:icon="@drawable/yfrog_icon" android:label="@string/YFROG">
        <action android:name="jp.co.vulnerable.ACTION_UPLOAD" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="image/*" />
        <data android:mimeType="video/*" />
    </intent-filter>
</activity>
```

android:name 是指实现此活动的类的名称。包的名称为“jp.co.vulnerable”，因此实现此活动的类的完全限定名称为 jp.co.vulnerable.media.yfrog.YfrogUploadDialog。由于定义了 filter，因此此活动将导出到其他应用程序。

6.1.8.5 合规代码示例

通过 android:exported="false"限制 activity 的导出。

```
<activity android:configChanges="keyboard|keyboardHidden|orientation" android:name=".media.yfrog.YfrogUploadDialog" android:theme="@style/ VulnerableTheme.Dialog" android:windowSoftInputMode="stateAlwaysHidden" android:exported="false">
</activity>
```

通过在 AndroidManifest.xml 中声明 export =“false”，应用程序可以通过判定调用者来修复此漏洞。在 activity 的 onCreate()方法中，添加代码以检查调用者的包名是否与其自身的包名相同。如果包名不同，则该活动退出：

```
public void onCreate(Bundle arg5) {
    super.onCreate(arg5);
    // ...
    ComponentName v0 = this.getCallingActivity();
    if(v0 == null) {
        this.finish();
    } else if(!jp.r246.twicca.equals(v0.getPackageName())) {
        this.finish();
        } else {
            this.a = this.getIntent().getData();
            if(this.a == null) {
                this.finish();
            }
            // ...
        }
    }
}
```

由于 Android 开发人员可以选择任意一个包名称，不同的应用开发者可以选择相同的包名称。因此，通常不建议使用包名称来验证的调用者。建议的替代方法是检查开发人员的证书，而不是包名称。

不过由于只有一个包含特定包名称的应用可以在 Google Play 上存在。如果用户尝试安装程序包名称已经存在于设备上的应用程序，安装将会失败或覆盖以前安装的应用程序。所以，该解决方案可能对于漏洞利用是合乎逻辑和安全的。

6.1.8.6 编程建议

限制 activity 的访问，我们建议：

- 如果应用的 Activity 组件不必要导出，或者组件配置了 intent filter 标签，建议显示设置组件的“android:exported”属性为 false
- 如果组件必须要提供给外部应用使用，建议对组件进行权限控制

6.1.9 ADRD-08. 使用显示 Intent 明确指定要启动的 activity

6.1.9.1 显示和隐式 Intent

启动那个 Acitivity 有两种方式：implicit(隐藏) intent 和 explicit(明确) intent。

- Explicit Intent

明确的指定了要启动的 Acitivity，比如以下 Java 代码，明确指定了要启动 PictureActivity：

```
// Using an Activity in the same application by an explicit Intent
Intent intent = new Intent(this, PictureActivity.class);
```

```
3.  intent.putExtra("BARCODE", barcode);
4.  startActivity(intent);
```

- Implicit Intent

没有明确的指定要启动哪个 Activity ，而是通过设置一些 Intent Filter 来让系统去筛选合适的 Acitivity 去启动。

```
1.  // Using other applicaion's Public Activity by an explicit Intent
2.  Intent intent = new Intent();
3.  intent.setClassName(
4.      "org.jssec.android.activity.publicactivity",
5.      "org.jssec.android.activity.publicactivity.PublicActivity");
6.  startActivity(intent);
```

当使用 startActivity 时，隐式 Intent 解析到一个单一的 Activity。如果存在多个 Activity 都有能力在特定的数据上执行给定的动作的话，Android 会从这些中选择最合适的一个进行启动。

6.1.9.2 编程建议

即使通过显式 Intent 使用其他应用程序的 Public Activity，目标 activitye 有可能是恶意软件。因为即使您以包名称限制目标 activity，恶意应用程序仍然也可能会伪造相同的包名称，所以，为了消除这种风险，建议考虑把 activity 设置为 Partner 或 In-house 的属性。

6.1.10 ADRD-09. 不指定 activity 的 taskAffinity 属性

在 Android 操作系统中，activity 由任务管理地，简单说来 task 是一种 stack 的数据结构，先入后出。

Activity 的切换，也取决于 activity 的 launchMode 属性。在默认设置中，每个 activity 使用其包名称作为其 affinity 属性。根据应用程序分配任务，所以单个应用程序中的所有 activity 都属于同一任务。

6.1.10.1 不合规代码示例

要更改任务分配，可以在 AndroidManifest.xml 中定义 affinity 属性，或设置一个发送到该 Activity 的 Intent。

```
1.  android:taskAffinity="com.test.task"
2.  android:label="@string/app_name">
```

恶意软件中的 Activity 如果也声明为同样的 taskAffinity，那他的 Activity 就会启动到你的 task 中，就会有机会拿到你的 intent。

6.1.10.2 编程建议

一定不要在 AndroidManifest.xml 文件中指定 android: taskAffinity，请直接使用默认设置，使 affinity 属性与包名称一致，以防止发送或接收的 Intents 内的敏感信息被恶意程序获取。

6.1.11 ADRD-10. 不指定 activity 的 launchMode 属性

Android 中 Activity 的 LaunchMode 分成以下四种：

- Standard，这种方式打开的 Activity 不会被当作 rootActivity，会生成一个新的 Activity 的 instance，会和打开者在同一个 task 内。

- singleTop，和 standard 基本一样，唯一的区别在于如果当前 task 第一个 Activity 就是该 Activity 的话，就不会生成新的 instance。
- singleTask，系统会创建一个新 task(如果没有启动应用)和一个 activity 新实例在新 task 根部，然后，如果 activity 实例已经存在单独的 task 中，系统会调用已经存在 activity 的 onNewIntent()方法，而不是存在新实例，仅有一个 activity 实例同时存在。
- singleInstance，和 singleTask 相似，除了系统不会让其他的 activities 运行在所有持有的 task 实例中，这个 activity 是独立的，并且 task 中的成员只有它，任何其他 activities 运行这个 activity 都将打开一个独立的 task。

Activity 启动模式用于控制启动 activity 时创建新任务和 activity 实例的设置。默认设置为“standard”。在“standard”设置中，启动 activity 时会创建新实例，其任务即是调用者所在的任务，不会创建新任务。

在 Android 5.0（API Level 21）和更高版本中，使用 getRecentTasks()检索的信息已经被限制了调用者的任务，可能还有一些其他任务，如不敏感的 Home 任务。然而，支持 Android 5.0（API Level 21）以下版本的应用程序，应防止敏感信息泄露。

发送到任务的 root activity 的内容将添加到任务历史记录中。Root activity 是第一个任务启动的第一个 activiry。任何应用程序都可以通过使用 ActivityManager 类读取添加到任务历史记录中的 intent。要浏览任务历史记录，需要在 AndroidManifest.xml 文件中指定 GET_TASKS 权限。

6.1.11.1 编程建议

Activity 启动模式可以在 AndroidManifest.xml 文件里通过设置 android: launchMode 属性来显式指定。但基于上述原因，这个属性不应该显式指定，而应该保持为默认的“standard”。

```
AndroidManifest.xml
<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name" >
    <!-- Private activity -->
    <!-- *** POINT 2 *** Do not specify launchMode -->
    <activity
        android:name=".PrivateActivity"
        android:label="@string/app_name"
        android:exported="false" />
</application>
```

6.1.12 ADRD-12. 不应为启动 Activity 的 Intents 设置 FLAG_ACTIVITY_NEW_TASK

Activity 的启动模式可以在执行 startActivity()或 startActivityForResult()时进行更改，在某些情况下可能会生成新任务。这时要考虑在执行期间不更改活动的启动模式。要更改 Activity 启动模式，可以通过 setFlags()或 addFlags()设置 Intent 标志，并使用该 Intent 作为 startActivity()或 startActivityForResult()的参数。FLAG_ACTIVITY_NEW_TASK 这个标志用于创建新任务。

在 intent 中指明用 FLAG_ACTIVITY_NEW_TASK 模式的话，如果发现该 activity 不存在的话，就会强制新建一个 task。如果同时设置了 FLAG_ACTIVITY_MULTIPLE_TASK 和 FLAG_ACTIVITY_NEW_TASK，就无论如何都会生成新的 task，该 Activity 就会变成 rootActiviy。

携带敏感信息的 Intents，不应该使用 FLAG_ACTIVITY_NEW_TASK 和 FLAG_ACTIVITY_MULTIPLE_TASK。

```
Example of sending an intent
// *** POINT 6 *** Do not set the FLAG_ACTIVITY_NEW_TASK flag for the intent to start an activity.
Intent intent = new Intent(this, PrivateActivity.class);
intent.putExtra("PARAM", "Sensitive Info");
startActivityForResult(intent, REQUEST_CODE);
```

6.1.13 ADRD-11. 应用内部的 activity 应设置为 Private

Activity 的访问权限包括以下几类：

Type	Definition
Private Activity	外部应用程序无法调用，最安全的 activity
Public Activity	权限公开，所有应用程序都能调到
Partner Activity	可以被指定的应用访问
In-house Activity	可被同一开发商的其它应用程序访问

仅在应用内部使用的 activity 应设置为 Private，如下示例：

AndroidManifest.xml

```
<!-- Private activity -->
<!-- *** POINT 3 *** Explicitly set the exported attribute to false. -->
<activity
    android:name=".PrivateActivity"
    android:label="@string/app_name"
    android:exported="false" />
```

仅在单个应用程序中使用的 activity 不应设置 intent-filter。基于 intent-filter 的特征和工作原理，即使您打算将 Intent 发送到内部 activity，如果是通过 intent-filter 发送，可能会无意中启动另一个 Activity。这时需要通过 exported 属性和 intent-filter 定义情况来看：

Intent-filter	Exported true	Exported false	Exported 不指定
Intent-filter 有定义	Public		
Intent-filter 不定义	Public, Partner, In-house		

6.1.13.1 不合规代码示例

AndroidManifest.xml(Not recommended)

```
<!-- Private activity -->
<!-- *** POINT 3 *** Explicitly set the exported attribute to false. -->
<activity
```

```
    android:name=".PictureActivity"
    android:label="@string/picture_name"
    android:exported="false" >
    <intent-filter>
        <action android:name="org.jssec.android.activity.OPEN />
    </intent-filter>
</activity>
```

6.1.14 ADRD-13. Public Activity 的访问

根据业务类型不同，Public Activity 可以分为一下几种，如下：

Type	Definition
Public Activity	权限公开，所有应用程序都能调到
Partner Activity	可以被指定的应用访问
In-house Activity	可被同一开发商的其它应用程序访问

6.1.14.1 Public Activity

任意一个应用程序都可以发送 Intent 来启动公共的 Activity,意味着恶意程序很容易劫持 Intent 信息，伪造 Intent，安全性很低。

要注意一下几点：

(1) 细心处理接收到 Intent，防止系统异常等问题；

(2) Finish 的时候不要在 Intent 中返回敏感信息。

```
public class PublicActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            // *** POINT 1 *** 细心处理接到 Intent
        String param = getIntent().getStringExtra("PARAM");
            Toast.makeText(this, String.format("Received param: \"%s\"", para
m), Toast.LENGTH_LONG).show();
    }
    public void onReturnResultClick(View view) {
        // *** POINT 2 *** Finish 的时候不要在 Intent 中设置敏感信息
        Intent intent = new Intent();
        intent.putExtra("RESULT", "Not Sensitive Info");
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

6.1.14.2 Partner Activity

不具备相同签名的不同应用，希望调用对方的 Activity 进行交互，数据访问。一般是针对不同公司业务线产品，希望公开自我的 Activity，达到业务上的互相合作。

要注意以下几点：

创建 Patner Activity

(1) 不要声明 taskAffinity；
(2) 不要声明 launchMode；
(3) 不设置 intent filter；
(4) 使用白名单机制验证应用签名；
(5) 细心处理来自 Partner Activity 的 Intent；
(6) 仅返回给 Partner Activity 一些公开信息；

使用 Partner Activity

(7) 确认目标应用证书是否在白名单中；
(8) Start Activity 时不要设置 FLAG_ACTIVITY_NEW_TASK；
(9) 使用 putExtra()放公开且不敏感的信息；
(10) 显式 Intent 调用 Partner Activity,具体到包名，类名；
(11) 使用 startActivityForResult()调用 Partner Activity，进行返回验证；
(12) 细心处理接收到的数据，即使数据来自 Partner Activity。

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.activity.partneractivity" >
    <application
        android:allowBackup="false"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <!-- Partner activity -->
        <!-- *** POINT 1 *** 不要声明 taskAffinity -->
        <!-- *** POINT 2 *** 不要声明 launchMode -->
        <!-- *** POINT 3 *** 不设置 intent filter,设置 exported 为 true -->
        <activity
            android:name=".PartnerActivity"
            android:exported="true" />
    </application>
</manifest>
```

Partner Activity

```
public class PartnerActivity extends Activity {
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        boolean isdebug = Utils.isDebuggable(context);
        sWhitelists = new PkgCertWhitelists();
        // 注册 Partner application 证书的哈希值
```

```
7.          sWhitelists.add("org.jssec.android.activity.partneruser", "0EFB7236 3
28348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255");
8.          // 注册其他 Partner app 同上
9.      }
10.     private static boolean checkPartner(Context context, String pkgname) {
11.         if (sWhitelists == null) buildWhitelists(context);
12.             return sWhitelists.test(context, pkgname);
13.     }
14.     @Override
15.     public void onCreate(Bundle savedInstanceState) {
16.         super.onCreate(savedInstanceState);
17.         setContentView(R.layout.main);
18.         // *** POINT 4 ***使用白名单机制验证应用签名.
19.         if (!checkPartner(this, getCallingActivity().getPackageName())) {
20.             Toast.makeText(this, "Requesting application is not a partner app
lication.", Toast.LENGTH_LONG).show();
21.             finish();
22.             return;
23.         }
24.         // *** POINT 5 *** 细心处理来自 Partner Activity 的 Intent
25.         Toast.makeText(this, "Accessed by Partner App", Toast.LENGTH_LONG).sh
ow();
26.     }
27.     public void onReturnResultClick(View view) {
28.         // *** POINT 6 *** 只返回给 Partner Activity 一些公开信息
29.         Intent intent = new Intent();
30.         intent.putExtra("RESULT", "Information for partner applications");
31.         setResult(RESULT_OK, intent);
32.         finish();
33.     }
34. }
```

Use a Partner Activity

```
1.  public class PartnerUserActivity extends Activity {
2.      private static PkgCertWhitelists sWhitelists = null;
3.      private static void buildWhitelists(Context context) {
4.          boolean isdebug = Utils.isDebuggable(context);
5.          sWhitelists = new PkgCertWhitelists();
6.          // 注册 Partner Activity 应用的哈希值
7.          sWhitelists.add("org.jssec.android.activity.partneractivity", "0EFB72
36 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255");
8.          // 其他 Partner 应用注册同上
9.      }
10.     private static boolean checkPartner(Context context, String pkgname) {
11.         if (sWhitelists == null) buildWhitelists(context);
12.             return sWhitelists.test(context, pkgname);
```

```
    }
    private static final int REQUEST_CODE = 1;
    // 目标 partner activity 的信息
    private static final String TARGET_PACKAGE =  "org.jssec.android.activity.partneractivity";
    private static final String TARGET_ACTIVITY = "org.jssec.android.activity.partneractivity.PartnerActivity";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
    public void onUseActivityClick(View view) {
        // *** POINT 7 ***确认目标应用证书是否在白名单中
        if (!checkPartner(this, TARGET_PACKAGE)) {
            Toast.makeText(this, "Target application is not a partner application.", Toast.LENGTH_LONG).show();
            return;
        }
        try {
            // *** POINT 8 ***Start Activity 时不要设置 FLAG_ACTIVITY_NEW_TASK;
            Intent intent = new Intent();
            // *** POINT 9 *** 使用 putExtra()放公开且不敏感的信息
            intent.putExtra("PARAM", "Info for Partner Apps");
            // *** POINT 10 *** 显式 Intent 调用 Partner Activity,具体到包名，类名
            intent.setClassName(TARGET_PACKAGE, TARGET_ACTIVITY);
            // *** POINT 11 *** 使用 startActivityForResult()调用 Partner Activity;
            startActivityForResult(intent, REQUEST_CODE);
        }
        catch (ActivityNotFoundException e) {
            Toast.makeText(this, "Target activity not found.", Toast.LENGTH_LONG).show();
        }
    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        // *** POINT 12 *** 细心处理接收到的数据
        super.onActivityResult(requestCode, resultCode, data);
        if (resultCode != RESULT_OK) return;
            switch (requestCode) {
                case REQUEST_CODE:
```

```
51.                     String result = data.getStringExtra("RESULT");
52.                     // *** POINT 12 *** 消息处理接收到的数据，即使数据来自
Partner Activity。
53.                     Toast.makeText(this,String.format("Received result: \"%s\
"", result), Toast.LENGTH_LONG).show();
54.                     break;
55.         }
56.     }
57. }
```

6.1.14.3 IN-House Activity

具有相同签名的企业内部不同产品线的应用，具有相同的签名，业务上存在互相调用。

要注意以下几点：

（1） 定义 Activity 的签名权限为 signature;
（2） 不声明 taskAffinity;
（3） 不声明 launchMode;
（4） 需要内部声明的 signature permission;
（5） 不要定义 intent filter;
（6） 验证签名;
（7） 细心处理 Intent 的传输的数据;
（8） Finsh()时，敏感信息可以返回;

AndroidManifest.xml

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3.      package="org.jssec.android.activity.inhouseactivity" >
4.      <!-- *** POINT 1 *** 定义内部 signature permission -->
5.      <permission
6.          android:name="org.jssec.android.activity.inhouseactivity.MY_PERMISSIO
N"
7.          android:protectionLevel="signature" />
8.      <application
9.          android:allowBackup="false"
10.         android:icon="@drawable/ic_launcher"
11.         android:label="@string/app_name" >
12.         <!-- In-house Activity -->
13.         <!-- *** POINT 2 *** 不声明 taskAffinity-->
14.         <!-- *** POINT 3 *** 不声明 launchMode -->
15.         <!-- *** POINT 4 *** 需要内部声明的 signature permission  -->
16.         <!-- *** POINT 5 *** 不要定义 intent filter,设置 exported 为 true -->
17.         <activity
18.             android:name="org.jssec.android.activity.inhouseactivity.InhouseA
ctivity"
```

```
        android:exported="true"
        android:permission="org.jssec.android.activity.inhouseactivity.MY_PERMISSION" />
    </application>
</manifest>
```

IN-House Activity

```
public class InhouseActivity extends Activity {
    // 内部 Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.activity.inhouseactivity.MY_PERMISSION";
    private static String sMyCertHash = null;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            sMyCertHash = "D397D343 A5CBC10F 4EDDEB7C A10062DE 5690984F 1FB9E88B D7B3A7C2 42E142CA";
        }
        return sMyCertHash;
    }
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // *** POINT 6 *** 验证签名
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "The in-house signature permission is not declared by in-house application.",
                    Toast.LENGTH_LONG).show();
            finish();
            return;
        }
        // *** POINT 7 *** 小心处理 Intent 的传输的数据
        String param = getIntent().getStringExtra("PARAM");
        Toast.makeText(this, String.format("Received param: \"%s\"", param), Toast.LENGTH_LONG).show();
    }
    public void onReturnResultClick(View view) {
        // *** POINT 8 *** 敏感信息能被返回
        Intent intent = new Intent();
        intent.putExtra("RESULT", "Sensitive Info");
        setResult(RESULT_OK, intent);
        finish();
    }
}
```

6.1.15 ADRD-14. 限制敏感 Broadcast Receiver 的访问

在 Android 上，为 AndroidManifest.xml 文件或者代码中注册中的活动声明 intent filter 意味着该广播可能会导出给其他应用程序使用。如果广播仅用于应用程序的内部使用，却声明 intent filter，则任何其他应用程序（包括恶意软件）都可以激活该广播以进行恶意使用。

对于涉及敏感数据/操作的 Broadcast：

（1）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。

（2）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。

有三种方式限制对内容提供者的访问：

- Public
- Private
- Restricted access

6.1.15.1 Public

通过在AndroidManifest.xml文件中指定android:exported属性,或者代买中动态注册， 一个Broadcast Receiver可以被设定为对其他应用公开。

```
1. <receiver
2. android:name=".PublicReceiver"
3.         android:exported="true">
4.             <intent-filter>
5.                 <action android:name="org.jssec.android.broadcast.MY_BROADCAS
T_PUBLIC" />
6.             </intent-filter>
7. </receiver>
```

如果一个广播设定为 public,第三方可以发送恶意广播或者伪造广播接收器，进行对系统的攻击。

6.1.15.2 Private

通过将AndroidManifest.xml设定android:exported的属性将一个广播设置为private。这样的广播只能内部调用，也是最安全的方式。

```
1. <receiver
2.     android:name=".PrivateReceiver"
3.     android:exported="false" />
```

6.1.15.3 Restricted access

组件在声明所需要权限时，一定要对权限及其 protectionLevel 进行定义。

在声明所需要权限时，对权限及其 protectionLevel 进行定义（建议至少是 signature 级别），可防范其他应用恶意定义权限导致权限控制措施失效的问题。请参阅 6.1.26 ADRD-25. 组件在声明权限时，要对权限及其 protectionLevel 进行定义。

6.1.16 ADRD-15. Public Broadcast Receiver

根据业务类型不同，Public Recevier 可以分为一下几种：

（1） Public Broadcast Receiver ；
（2） IN-House Broadcast Receiver。

6.1.16.1 Public Broadcast Receiver

公共广播，没有固定的接收方，可以被一些未确定性的应用激活，同时也容易被伪造，安全性很低。

（1）细心处理 Intent 信息，防止 Intent 被篡改；
（2）Return result 的时候不要放敏感信息；
（3）不要发送敏感信息；

Public Receiver

```
1.  public class PublicReceiver extends BroadcastReceiver {
2.      private static final String MY_BROADCAST_PUBLIC =
3.          "org.jssec.android.broadcast.MY_BROADCAST_PUBLIC";
4.      public boolean isDynamic = false;
5.      private String getName() {
6.          return isDynamic ? "Public Dynamic Broadcast Receiver" : "Public Stat
ic Broadcast Receiver";
7.      }
8.      @Override
9.      public void onReceive(Context context, Intent intent) {
10.         // *** POINT 1***细心处理 Intent 信息
11.         if (MY_BROADCAST_PUBLIC.equals(intent.getAction())) {
12.             String param = intent.getStringExtra("PARAM");
13.             Toast.makeText(context,
14.                 String.format("%s:\nReceived param:\"%s\"",getName(),param),
15.                 Toast.LENGTH_SHORT).show();
16.         }
17.         // *** POINT 2 *** Return result 的时候不要放敏感信息
18.         setResultCode(Activity.RESULT_OK);
19.         setResultData(String.format("Not Sensitive Info from %s",getName()));
20.         abortBroadcast();
21.     }
22. }
```

Sending

```
1.  public class PublicSenderActivity extends Activity {
2.      private static final String MY_BROADCAST_PUBLIC =
3.          "org.jssec.android.broadcast.MY_BROADCAST_PUBLIC";
4.      public void onSendNormalClick(View view) {
5.          // *** POINT 3 ***不要发送敏感信息
6.          Intent intent = new Intent(MY_BROADCAST_PUBLIC);
7.          intent.putExtra("PARAM", "Not Sensitive Info from Sender");
8.          sendBroadcast(intent);
9.      }
10.     // ……
11.     private BroadcastReceiver mResultReceiver = new BroadcastReceiver() {
12.         @Override
```

```
        public void onReceive(Context context, Intent intent) {
            // *** POINT 1 *** 细心处理 Intent 信息
            // Omitted, since this is a sample. Please refer to "3.2 Handling Input Data Carefully and Securely."
            String data = getResultData();
            PublicSenderActivity.this.logLine(
                String.format("Received result: \"%s\"", data));
        }
    };
    // ……
}
```

6.1.16.2 IN-House Broadcast Receiver

内部广播，具有相同的签名，除了内部应用能够发送接收，其他应用都不能发送接收广播，较为安全。

要注意以下几点：

Receiver:

(1) 定义一个内部的 signature Permission 用于发送、接收广播；
(2) 声明使用内部的 signature Permission 接收结果；
(3) 静态广播需要在<recevier/>添加内部的 signature Permission；
(4) 动态广播注册，在注册时需要添加 signature Permission；
(5) 验证内部定义的 signature Permission；
(6) 细心处理 Intent 信息；
(7) 可以返回敏感信息；

Sending:

(8) 定义内部 signature permission 去接收结果；
(9) 声明内部 signature permission 去接收广播；
(10) 验证内部 signature permission；
(11) 可以返回敏感信息；
(12) 需要验证内部 signature permission；
(13) 小心处理接收到的数据；

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.broadcast.inhousereceiver" >
    <!-- *** POINT 1 ***定义一个内部的 signature Permission 用于发送、接收广播 -->
    <permission
        android:name="org.jssec.android.broadcast.inhousereceiver.MY_PERMISSION"
        android:protectionLevel="signature" />
    <!-- *** POINT 2 *** 声明使用内部的 signature Permission 接收结果 -->
    <uses-permission
        android:name="org.jssec.android.broadcast.inhousesender.MY_PERMISSION" />
```

```
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:allowBackup="false" >
        <!-- *** POINT 3 *** 设置 exported 为 true -->
        <!-- *** POINT 4 *** 静态广播需要在<recevier/>添加内部的 signature Permission -->
        <receiver
            android:name=".InhouseReceiver"
            android:permission="org.jssec.android.broadcast.inhousereceiver.MY_PERMISSION"
            android:exported="true">
            <intent-filter>
                <action android:name="org.jssec.android.broadcast.MY_BROADCAST_INHOUSE" />
            </intent-filter>
        </receiver>
        ……
    </application>
</manifest>
```

House Recevier

```
public class InhouseReceiver extends BroadcastReceiver {
    // Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.broadcast.inhousereceiver.MY_PERMISSION";
    // certificate hash value
    private static String sMyCertHash = null;
    private static final String MY_BROADCAST_INHOUSE="org.jssec.android.broadcast.MY_BROADCAST_INHOUSE";
    public boolean isDynamic = false;
    private static String myCertHash(Context context) {
        if (sMyCertHash == null) {
            sMyCertHash = "0EFB7236 328348A9 89718BAD DF57F544 D5CCB4AE B9DB34BC 1E29DD26 F77C8255";
        }
        return sMyCertHash;
    }
    private String getName() {
        return isDynamic ? "In-house Dynamic Broadcast Receiver" : "In-house Static Broadcast Receiver";
    }
    @Override
    public void onReceive(Context context, Intent intent) {
        // *** POINT 6 *** 验证内部定义的 signature Permission
        if (!SigPerm.test(context, MY_PERMISSION, myCertHash(context))) {
```

```
            Toast.makeText(context, "The in-house signature permission is not declared by in-house application.",
                    Toast.LENGTH_LONG).show();
            return;
        }
        // *** POINT 7 *** H 小心处理 Intent 信息
        if (MY_BROADCAST_INHOUSE.equals(intent.getAction())) {
            String param = intent.getStringExtra("PARAM");
            Toast.makeText(context,
            String.format("%s:\nReceived param: \"%s\"", getName(), param),
                    Toast.LENGTH_SHORT).show();
        }
        // *** POINT 8 *** 可以返回敏感信息
        setResultCode(Activity.RESULT_OK);
        setResultData(String.format("Sensitive Info from %s", getName()));
        abortBroadcast();
    }
}
```

Sending Broadcast:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.jssec.android.broadcast.inhousesender" >
    <uses-permission android:name="android.permission.BROADCAST_STICKY"/>
    <!-- *** POINT 8 *** 定义内部 signature permission 去接收结果 -->
    <permission
        android:name="org.jssec.android.broadcast.inhousesender.MY_PERMISSION"
        android:protectionLevel="signature" />
    <!-- *** POINT 9 ***声明内部 signature permission 去接收广播 -->
    <uses-permission
        android:name="org.jssec.android.broadcast.inhousereceiver.MY_PERMISSION" />
  </manifest>
```

Sending

```
public class InhouseSenderActivity extends Activity {
    // In-house Signature Permission
    private static final String MY_PERMISSION = "org.jssec.android.broadcast.inhousesender.MY_PERMISSION";
    // In-house certificate hash value
    private static String sMyCertHash = null;
    // ……
    public void onSendNormalClick(View view) {
```

```
        // *** POINT 10 ***验证内部 signature permission;
        if (!SigPerm.test(this, MY_PERMISSION, myCertHash(this))) {
            Toast.makeText(this, "The in-house signature permission is not declared by in-house application.",
                    Toast.LENGTH_LONG).show();
            return;
        }
        // *** POINT 11 *** 可以返回敏感信息;
        Intent intent = new Intent(MY_BROADCAST_INHOUSE);
        intent.putExtra("PARAM", "Sensitive Info from Sender");
        // *** POINT 12 ***需要验证内部 signature permission;
        sendBroadcast(intent, "org.jssec.android.broadcast.inhousesender.MY_PERMISSION");
    }
    private BroadcastReceiver mResultReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            // *** POINT 13 ***细心处理接收到的数据;
            String data = getResultData();
            InhouseSenderActivity.this.logLine(String.format("Received result: \"%s\"", data));
        }
    };
    // ……
}
```

6.1.17 ADRD-16. 应用程序内部的广播接收者应设置为 Private

仅在应用程序内部使用的广播接收器应设置为 private，以避免收到其他应用程序的广播。它可以阻止应用程序的功能滥用或异常行为。

仅在应用程序内部使用的接收器不应配置 Intent-filter。Intent-filter 的特性可能会导致同一应用程序中的私有接收器被调用时，其他应用程序的 public 接收器意外地通过 Intent 过滤器被调用。

```
AndroidManifest.xml(Not recommended)
<!-- Private Broadcast Receiver -->
<!-- *** POINT 1 *** Set the exported attribute to false explicitly. -->
<receiver
    android:name=".PrivateReceiver"
    android:exported="false" >
    <intent-filter>
        <action android:name="org.jssec.android.broadcast.MY_ACTION" />
    </intent-filter>
</receiver>
```

6.1.18 ADRD-17. 限制敏感 Service 的访问

在 Android 上，为 AndroidManifest.xml 文件中的活动声明 intent filter 意味着该活动可能会导出给其他应用程序使用。如果活动仅用于应用程序的内部使用，却声明 intent filter，则任何其他应用程序（包括恶意软件）都可以激活该 activity 以进行恶意使用。导致敏感信息泄露，并可能受到绕过认证、恶意代码注入等攻击风险。

有两种方式限制对内容提供者的访问：

- Public
- Private

6.1.18.1 Public

通过在AndroidManifest.xml文件中指定android:exported属性,， 一个Service可以被设定为对其他应用公开。

```
1.  <service android:name=".PublicStartService" android:exported="true">
2.      <intent-filter>
3.          <action android:name="org.jssec.android.service.publicservice.action.
startservice" />
4.      </intent-filter>
5.   </service>
```

如果一个服务设定为 public,恶意程序可以伪造服务，或者修改 Intent 信息。

6.1.18.2 Private

通过将AndroidManifest.xml设定android:exported的属性将一个Service设置为private。这样的Service只能内部调用，也是最安全的方式。

```
1.  <service android:name=".PrivateStartService" android:exported="false"/>
```

6.1.18.3 Public Service

根据业务类型的不同，Public Service 包括：

（1） Public Service(公共 Service)；

（2） Partner Service(合作 Service)。

6.1.18.4 Public Service

公共 Service,没有指定发送方的 Service,可以被大多数的应用程序启动，最不安全。

要注意以下几点：

Create a Service

(1) 细心处理接收到的 Intent;

(2) Return 的时候，不要包含敏感信息;

Using a Service

(3) 显式调用 Service;

(4) 不要发送敏感信息。

Public Service

```
1.  public class PublicStartService extends Service{
2.      // ……
```

```
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // *** POINT 1 *** 细心处理接收到的 Intent;
        String param = intent.getStringExtra("PARAM");
        Toast.makeText(this, String.format("Recieved parameter \"%s\"", param)
, Toast.LENGTH_LONG).show();
        // *** POINT 2 *** Return 的时候，不要包含敏感信息
        // 这里是 startService()，所以没有返回信息
        return Service.START_NOT_STICKY;
    }
    // ……
}
```

Start Service

```
public class PublicUserActivity extends Activity {
    // ……
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.publicservice_activity);
    }
    public void onStartServiceClick(View v) {
        Intent intent = new Intent("org.jssec.android.service.publicservice.a
ction.startservice");
        // *** POINT 3 *** 显式调用 Service
        intent.setClassName(TARGET_PACKAGE, TARGET_START_CLASS);
        // *** POINT 4 *** 不要发送敏感信息
        intent.putExtra("PARAM", "Not sensitive information");
        startService(intent);
    }
    // ……
}
```

6.1.18.5 Partner Service

不同应用，不同弄签名，只能由特定的应用程序使用，一般是不用企业不同的产品需要利用对方的 Service 来做一些业务上的功能，要进行互相身份的验证。

Crate a Service

(1) 不要定义 Intent filter;

(2) 验证请求应用的证书;

(3) 不要忽略在 onBind()，onStartCommand(),onHandleIntent()中对请求应用的权限验证，因为服务启动后，就不会走 onCreate,不做认证，存在风险;

(4) 细心处理接收到的 Intent;

(5) 返回信息，只能返回给合作的应用;

Using a Service

(6) 判断目标应用证书是否在白名单中；
(7) 返回信息，确认是 partner app 返回的；
(8) 显式的调用 Partner Service；
(9) 细心处理接收到的数据。

Partner Service

```
1.  public class PartnerAIDLService extends Service {
2.      private static final int REPORT_MSG = 1;
3.      private static final int GETINFO_MSG = 2;
4.      private int mValue = 0;
5.      private static PkgCertWhitelists sWhitelists = null;
6.      private static void buildWhitelists(Context context) {
7.          // ……
8.      }
9.      private static boolean checkPartner(Context context, String pkgname) {
10.         if (sWhitelists == null) buildWhitelists(context);
11.             return sWhitelists.test(context, pkgname);
12.     }
13.     private final RemoteCallbackList<IPartnerAIDLServiceCallback> mCallbacks
= new RemoteCallbackList<IPartnerAIDLServiceCallback>();
14.     private static class ServiceHandler extends Handler{
15.         private Context mContext;
16.         private RemoteCallbackList<IPartnerAIDLServiceCallback> mCallbacks;
17.         private int mValue = 0;
18.         public ServiceHandler(Context context, RemoteCallbackList<IPartnerAID
LServiceCallback> callback, int value){
19.         // ……
20.         }
21.         @Override
22.         public void handleMessage(Message msg) {
23.             switch (msg.what) {
24.             case REPORT_MSG: {
25.                 if(mCallbacks == null){
26.                     return;
27.                 }
28.                 final int N = mCallbacks.beginBroadcast();
29.                 for (int i = 0; i < N; i++) {
30.                     IPartnerAIDLServiceCallback target = mCallbacks.getBroadc
astItem(i);
31.                     try {
32.                         // *** POINT 5 ***返回信息，只能返回给合作的应用
33.                         target.valueChanged("Information disclosed to partner
 application (callback from Service) No." + (++mValue));
34.                     } catch (RemoteException e) {
35.                     }
36.                 }
37.                 // ……
38.             }
```

```
39.         }
40.         }
41.         protected final ServiceHandler mHandler = new ServiceHandler(this, mC
allbacks, mValue);
42.         private final IPartnerAIDLService.Stub mBinder = new IPartnerAIDLServ
ice.Stub() {
43.             private boolean checkPartner() {
44.                 Context ctx = PartnerAIDLService.this;
45.                 if (!PartnerAIDLService.checkPartner(ctx, Utils.getPackageNam
eFromUid(ctx, getCallingUid()))) {
46.                     mHandler.post(new Runnable(){
47.                         @Override
48.                         public void run(){
49.                             Toast.makeText(PartnerAIDLService.this, "Requesti
ng application is not partner application.", Toast.LENGTH_LONG).show();
50.                         }
51.                     });
52.                     return false;
53.                 }
54.                return true;
55.             }
56.         }
57.         public void registerCallback(IPartnerAIDLServiceCallback cb) {
58.             // *** POINT 2 *** 验证请求应用的证书
59.           if (!checkPartner()) {
60.                 return;
61.             }
62.           if (cb != null) mCallbacks.register(cb);
63.         }
64.         public String getInfo(String param) {
65.             // *** POINT 2 *** 验证请求应用的证书
66.             if (!checkPartner()) {
67.                 return null;
68.             }
69.             // *** POINT 4 *** 小心处理接收到的 Intent
70.             Message msg = new Message();
71.             msg.what = GETINFO_MSG;
72.             msg.obj = String.format("Method calling from partner application.
 Recieved \"%s\"", param);
73.             PartnerAIDLService.this.mHandler.sendMessage(msg);
74.             // *** POINT 5 ***返回信息，只能返回给合作的应用
75.             return "Information disclosed to partner application (method from
 Service)";
76.         }
77.         public void unregisterCallback(IPartnerAIDLServiceCallback cb) {
78.             // *** POINT 2 ***  验证请求应用的证书
79.           if (!checkPartner()) {
```

```
                return;
            }
           if (cb != null) mCallbacks.unregister(cb);
        }
    };
    @Override
    public IBinder onBind(Intent intent) {
        // *** POINT 3 *** 不要忽略在 onBind(), onStartCommand(),onHandleIntent()中对请求应用的权限验证
        return mBinder;
    }
    // ……
}
```

Start Service

```
public class PartnerAIDLUserActivity extends Activity {
    private boolean mIsBound;
    private Context mContext;
    private final static int MGS_VALUE_CHANGED = 1;
    // *** POINT 6 *** 判断目标应用证书是否在白名单中
    private static PkgCertWhitelists sWhitelists = null;
    private static void buildWhitelists(Context context) {
        // ……
    }
    private static boolean checkPartner(Context context, String pkgname) {
        if (sWhitelists == null) buildWhitelists(context);
        return sWhitelists.test(context, pkgname);
    }
    private static final String TARGET_PACKAGE =  "org.jssec.android.service.partnerservice.aidl";
    private static final String TARGET_CLASS = "org.jssec.android.service.partnerservice.aidl.PartnerAIDLService";
    private static class ReceiveHandler extends Handler{
        private Context mContext;
        public ReceiveHandler(Context context){
            this.mContext = context;
        }
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MGS_VALUE_CHANGED: {
                      String info = (String)msg.obj;
                    Toast.makeText(mContext, String.format("Received \"%s\" with callback.", info), Toast.LENGTH_SHORT).show();
                    break;
                }
```

```
29.                 default:
30.                     super.handleMessage(msg);
31.                     break;
32.             } // switch
33.         }
34.     }
35.     private final ReceiveHandler mHandler = new ReceiveHandler(this);
36.     private final IPartnerAIDLServiceCallback.Stub mCallback =
37.         new IPartnerAIDLServiceCallback.Stub() {
38.             @Override
39.             public void valueChanged(String info) throws RemoteException {
40.                 Message msg = mHandler.obtainMessage(MGS_VALUE_CHANGED, info);

41.                 mHandler.sendMessage(msg);
42.             }
43.     };
44.     private IPartnerAIDLService mService = null;
45.     private ServiceConnection mConnection = new ServiceConnection() {
46.         @Override
47.         public void onServiceConnected(ComponentName className, IBinder servi
ce) {
48.             mService = IPartnerAIDLService.Stub.asInterface(service);
49.             try{
50.                 mService.registerCallback(mCallback);
51.             }catch(RemoteException e){
52.                 // service stopped abnormally
53.             }
54.             Toast.makeText(mContext, "Connected to service", Toast.LENGTH_SHO
RT).show();
55.         }
56.         @Override
57.         public void onServiceDisconnected(ComponentName className) {
58.             Toast.makeText(mContext, "Disconnected from service", Toast.LENGT
H_SHORT).show();
59.         }
60.     };
61.     @Override
62.     public void onCreate(Bundle savedInstanceState) {
63.         super.onCreate(savedInstanceState);
64.         setContentView(R.layout.partnerservice_activity);
65.         mContext = this;
66.     }
67.     public void onStartServiceClick(View v) {
68.         doBindService();
69.     }
70.     public void onGetInfoClick(View v) {
71.         getServiceinfo();
```

```
72.     }
73.     public void onStopServiceClick(View v) {
74.         doUnbindService();
75.     }
76.     @Override
77.     public void onDestroy() {
78.         super.onDestroy();
79.         doUnbindService();
80.     }
81.     private void doBindService() {
82.         if (!mIsBound){
83.                 // *** POINT 6 *** 判断目标应用证书是否在白名单中
84.             if (!checkPartner(this, TARGET_PACKAGE)) {
85.                 Toast.makeText(this, "Destination(Requested) sevice application is not registered in white list.", Toast.LENGTH_LONG).show();
86.                 return;
87.             }
88.             Intent intent = new Intent();
89.                 // *** POINT 7 ***返回信息，确认是 partner app 返回的；
90.             intent.putExtra("PARAM", "Information disclosed to partner application");
91.             // *** POINT 8 ***显式的调用 Partner Service
92.             intent.setClassName(TARGET_PACKAGE, TARGET_CLASS);
93.           bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
94.           mIsBound = true;
95.         }
96.     }
97.     private void doUnbindService() {
98.         if (mIsBound) {
99.             if(mService != null){
100.                     try{
101.                         mService.unregisterCallback(mCallback);
102.                     }catch(RemoteException e){
103.                     }
104.                 }
105.                 unbindService(mConnection);
106.                 Intent intent = new Intent();
107.                  // *** POINT 8 ***显式的调用 Partner Service
108.                 intent.setClassName(TARGET_PACKAGE, TARGET_CLASS);
109.                 stopService(intent);
110.                 mIsBound = false;
111.             }
112.         }
113.         void getServiceinfo() {
114.             if (mIsBound && mService != null) {
115.                 String info = null;
116.                 try {
```

```
                    // *** POINT 7 ***返回信息，确认是 partner app 返回的；
                     info = mService.getInfo("Information disclosed to partner application (method from activity)");
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
                // *** POINT 9 *** 细心处理接收到的数据
                Toast.makeText(mContext, String.format("Received \"%s\" from service.", info), Toast.LENGTH_SHORT).show();
            }
        }
    }
```

6.1.19 ADRD-18. 应用程序内部的 Service 应设置为 Private

仅在应用程序内部使用的 Service 应设置为 private，以避免其他应用激活。它可以阻止应用程序的功能滥用或异常行为。

仅在应用程序内部使用的 Service 不应配置 Intent-filter。

AndroidManifest.xml(Not recommended)

```
<!-- Private Service -->
<service android:name=".PrivateStartService" android:exported="false"/>
```

6.1.20 ADRD-19. 捕获 Intent 处理数据时发生的异常

通过 Intent.getXXXExtra()获取的数据，处理时进行以下判断，以及用 try catch 方式进行捕获所有异常，以防止应用出现拒绝服务漏洞：空指针异常，类型转换异常，数组越界访问异常，类未定义异常，其他异常。

6.1.21 ADRD-20. 不应使用隐式 Intent 广播敏感信息

Android 应用程序的核心组件（如 Activity，Services 和 BroadcastReciver）通过 Intent 激活。应用程序可以使用广播将 Intent 发送到多个应用程序（即将事件通知给多个应用程序）。此外，应用程序可以接收系统广播出来的 Intent。

BroadcastReciver 通过使用指定的 intentFilter 作为参数调用 Context.registerReceiver()来动态地注册自己。或者，可以通过在 AndroidManifest.xml 文件中定义<receiver>标签来静态注册 Reciver。

恶意广播接收方可以通过枚举所有可能的操作，数据和类型来窃听所有应用程序的 public 广播。即使广播被读取，发送者和接收者也无法知晓。粘性广播被窃听的风险最大，因为它们持续存在并被不断广播到新的接收者;因此，粘性广播 Intent 有更多机会被读取，而且，粘性广播无法受权限保护。

如果广播是有序广播，则恶意应用可以以高优先级注册自身，以便先接收到广播。然后，它可以取消广播，防止其进一步传播，从而导致拒绝服务，或者，它可能将恶意数据结果注入到该广播。

当使用 sendBroadcast（）时，通常任何其他应用程序（包括恶意程序）都可以接收广播。

因此，应该对广播的接收者和 Intent 增加一些限制。限制 Reciver 的一种方法是使用显式 Intent。显式 Intent 可以通过 setComponent()指定一个组件或通过 setClass()指定一个类，这样只有指定的组件或类可以解析该意图。

也可以通过设定权限来限制 Intent 的 Reciver，或者，从 Android API 版本 4.0 开始，还可以使用 Intent.setPackage（）将广播安全地限制在单个应用程序中。

另一种方法是使用 LocalBroadcastManager 类。把 Broadcast 及 Intent 限制在当前进程。LocalBroadcastManager 与 Context.sendBroadcast（Intent）有很多优点：

- 正在广播的数据不会离开你的应用程序，所以不用担心泄露私人数据。
- 其他应用程序不可能将这些广播发送到您的应用程序，因此您不必担心有安全漏洞被利用。
- 比通过系统发送全局广播更有效率。

6.1.21.1 不合规代码示例

该示例展示了应用程序（com/sample/ServerService.java）有一个容易受攻击的方法 d()使用隐含 Intent v1 作为 this.sendBroadcast()的参数来广播 Intent。Intent 包括设备的 IP 地址（local_ip），端口号和连接到设备的密码等敏感信息。

```
1.  public class ServerService extends Service {
2.      // ...
3.      private void d() {
4.          // ...
5.          Intent v1 = new Intent();
6.          v1.setAction("com.sample.action.server_running");
7.          v1.putExtra("local_ip", v0.h);
8.          v1.putExtra("port", v0.i);
9.          v1.putExtra("code", v0.g);
10.         v1.putExtra("connected", v0.s);
11.         v1.putExtra("pwd_predefined", v0.r);
12.         if (!TextUtils.isEmpty(v0.t)) {
13.         v1.putExtra("connected_usr", v0.t);
14.         }
15.     }
16.     this.sendBroadcast(v1);
17. }
```

另一应用程序可以通过下面的 BroadcastReciver 接收该广播：

```
1.  final class MyReceiver extends BroadcastReceiver {
2.      public final void onReceive(Context context, Intent intent) {
3.          if (intent != null && intent.getAction() != null) {
4.              String s = intent.getAction();
5.              if (s.equals("com.sample.action.server_running") {
6.              String ip = intent.getStringExtra("local_ip");
7.              String pwd = intent.getStringExtra("code");
8.              String port = intent.getIntExtra("port", 8888);
9.              boolean status = intent.getBooleanExtra("connected", false);
```

```
            }
        }
    }
}
```

攻击者可以实现一个 Reciver 来接收应用程序发送的隐含 Intent:

```
final class MyReceiver extends BroadcastReceiver {
    public final void onReceive(Context context, Intent intent) {
        if (intent != null && intent.getAction() != null) {
            String s = intent.getAction();
            if (s.equals("com.sample.action.server_running") {
            String ip = intent.getStringExtra("local_ip");
            String pwd = intent.getStringExtra("code");
            String port = intent.getIntExtra("port", 8888);
            boolean status = intent.getBooleanExtra("connected", false);
            }
        }
    }
}
```

6.1.21.2 合规代码示例

如果 Intent 仅在同一应用程序内部广播和接收，则可以使用 LocalBroadcastManager，其他应用程序不能收到广播消息,这降低了泄露敏感信息的风险。注意不是使用 Context.sendBroadcast()，而是使用 LocalBroadcastManager.sendBroadcast()：

```
Intent intent = new Intent("my-sensitive-event");
intent.putExtra("event", "this is a test event");
LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
```

6.1.22 ADRD-21. 应对传入的 Intent 进行合法性判断

攻击者可以反编译获得应用组件<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 包括但不限于以下几种：

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

6.1.22.1 不合规代码示例

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

```
5.      }
6.  }
```

该示例需要引用外部 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。

6.1.22.2 合规代码示例

```
1.  Intent intent = getIntent();
2.  if (intent != null){
3.      if (intent.getBundleExtra("xxx") != null){
4.          if (intent.getBundleExtra("xxx").equals(XXX)){
5.              //...
6.          }
7.      }
8.  }
```

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，开发者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）。

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。

6.1.23 ADRD-22. 应用程序在响应调用者前确认其权限

如果应用程序正在使用授权的权限来响应调用者，则必须检查该调用应用程序是否具有该授权。否则，响应的应用程序可能会授予其不应拥有的呼叫应用程序的权限。可以使用 Context.checkCallingPermission（）和 Context.enforceCallingPermission（）方法来确保调用应用程序具有正确的权限。

6.1.24 ADRD-23. 不应为隐式 Intent 授予 URI 权限

存储在应用程序 Service Provider 中的数据可以被包含在 Intent 中的 URI 引用。如果 Intent 的接收者没有访问 URI 所需的权限，则 Intent 的发送者可以给 Intent 设置标志 FLAG_GRANT_READ_URI_PERMISSION 或 FLAG_GRANT_WRITE_URI_PERMISSION。如果 provider 指定可以授予 URI 权限，则 Intent 的收收者就可以读取或写入 URI 上的数据。

如果恶意组件能够拦截 Intent，则可以访问 URI 中的数据。任何组件都可以拦截隐式 Intent，所以如果数据是私有的，那么承载数据权限的 Intent 都必须被是显式的，而不是隐式的。（请参阅 6.1.21 ADRD-20. 不应使用隐式 Intent 广播敏感信息）。

6.1.25 ADRD-24. 私有组件应指定 exported 属性为 false

在 Android 应用程序中，如果组件的导出值在应用程序的清单文件中被明确标记为 false，那么该组件将变为私有的。如果不指定组件的访问权限，那么，任何应用程序都可以访问该组件。

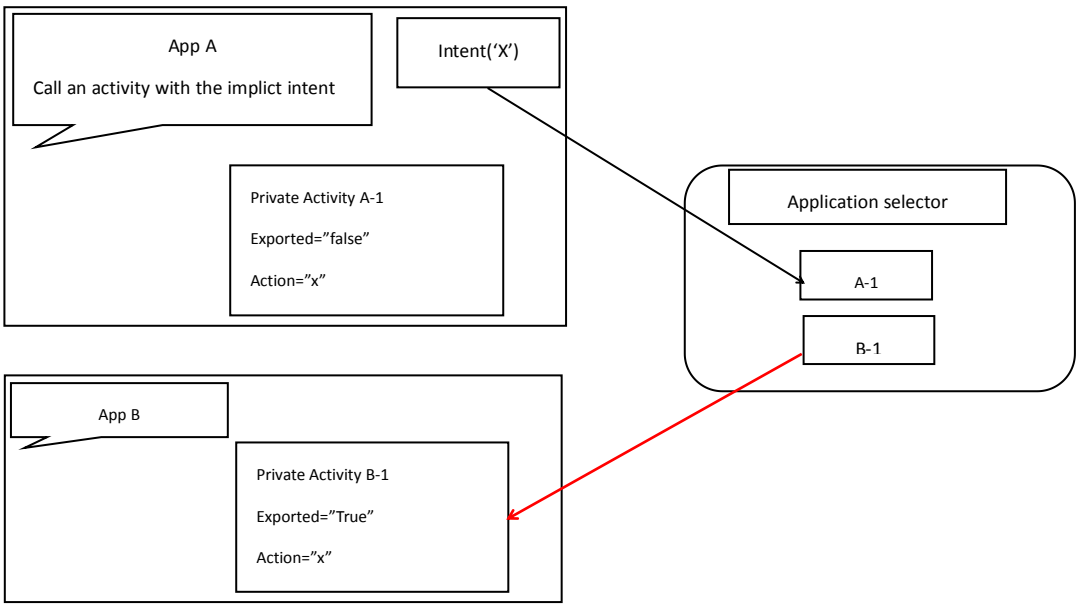
只在应用程序内部使用的组件必须设置为非公开，以防受到其他应用程序的调用。若应用组件未设置 exported 属性：

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。



6.1.25.1 不合规代码示例

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

6.1.25.2 合规代码示例

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。

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。

```
<activity
```

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

6.1.26 ADRD-25. 组件在声明权限时，要对权限及其 protectionLevel 进行定义

在声明所需要权限时，对权限及其 protectionLevel 进行定义（建议至少是 signature 级别），可防范其他应用恶意定义权限导致权限控制措施失效的问题。

6.1.27 ADRD-26. 应用程序应当慎用粘滞广播

粘滞广播不能使用权限来保护，可供任何接收者访问。如果这些广播包含敏感数据或发送到恶意接收者，可能会对应用程序造成危害，因此需要慎重使用粘滞广播。

6.1.27.1 不合规代码示例

发送粘滞广播的代码样例（不推荐）：

```
// ...
context.sendStickyBroadcast(intent);
// ...
```

由于粘滞广播无法使用权限来保护，因此只能在特殊情况下使用。重新评估使用粘滞广播的意图，了解所需的行为是否可以通过受接收者权限保护的常规广播来实现。这样可以限制仅为具有所需权限的接收者显示广播数据。此外，还应详细审查广播消息中包含的数据。

6.1.27.2 合规代码示例

发送常规广播并明确指定所需的接收者权限的代码样例（推荐）：

```
// ...
context.sendBroadcast(intent, "permission.ALLOW_INCOMING");
// ...
```

6.1.28 ADRD-27. 调用框架中新增的接口应有相应的权限校验

为了配合系统 app 实现某些功能，需要在框架里面增加新的接口（hide），调用加对应的权限校验，根据新增加的接口的重要性，可以增加权限、平台签名等校验。

- 权限校验

```
mContext.enforceCallingOrSelfPermission( String permission,  String message);
// 或者
ActivityManager.checkUidPermission(permissionType, uid);
```

- 平台签名校验

```
PackageManager pm = context.getPackageManager();
pm.checkSignatures("android", packageName)==PackageManager.SIGNATURE_MATCH
// 或者
pm.checkSignatures(1000, callingUid)==PackageManager.SIGNATURE_MATCH
```

6.2 数据存储安全

6.2.1 ADRD-01. allowBackup 属性须指定为 false

Android API Level 8 及其以上 Android 系统提供了为应用程序数据的备份和恢复功能，此功能的开关决定于该应用程序中 AndroidManifest.xml 文件中的 allowBackup 属性值，其属性值默认是 true。

当 allowBackup 标志为 true 时，用户即可通过 adb backup 和 adb restore 来进行对应用数据的备份和恢复，这可能会带来一定的安全风险，安全风险源于 adb backup 容许任何一个能够打开 USB 调试开关的人从 Android 手机中复制应用数据到外设，一旦应用数据被备份之后，所有应用数据都可被用户读取；adb restore 容许用户指定一个恢复的数据来源（即备份的应用数据）来恢复应用程序数据的创建。

因此，当一个应用数据被备份之后，用户即可在其他 Android 手机或模拟器上安装同一个应用，以及通过恢复该备份的应用数据到该设备上，在该设备上打开该应用即可恢复到被备份的应用程序的状态，比如通讯录应用，一旦应用程序支持备份和恢复功能，攻击者即可通过 adb backup 和 adb restore 进行恢复新安装的同一个应用来查看聊天记录等信息；对于支付金融类应用，攻击者可通过此来进行恶意支付、盗取存款等。因此务必将 allowBackup 标志值设置为 false 来关闭应用程序的备份和恢复功能，以免造成信息泄露和财产损失风险。

6.2.1.1 不合规代码示例

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.vivo.chat"
          android:versionCode="1"
          android:versionName="1.0">
    <uses-sdk android:minSdkVersion="10"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    <application
                android:label="@string/app_name">
        <activity android:name="LoginActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <activity android:name=".HomeActivity"/>
    </application>
</manifest>
```

6.2.1.2 合规代码示例

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.vivo.chat "
          android:versionCode="1"
```

```
5.          android:versionName="1.0">
6.      <uses-sdk android:minSdkVersion="10"/>
7.      <uses-permission android:name="android.permission.READ_PHONE_STATE" />
8.      <application
9.              android:allowBackup="false"
10.             android:label="@string/app_name">
11.         <activity android:name="LoginActivity"
12.                   android:label="@string/app_name">
13.             <intent-filter>
14.                 <action android:name="android.intent.action.MAIN"/>
15.                 <category android:name="android.intent.category.LAUNCHER"/>
16.             </intent-filter>
17.         </activity>
18.         <activity android:name=".HomeActivity"/>
19.     </application>
20. </manifest>
```

6.2.2 ADRD-02. 根据数据敏感程度选择最合适的存储方式

Android 提供了多种方式来保存永久性应用数据。我们要根据特定需求选择的最合适的解决方案，例如数据应该是应用的私有数据，还是可供其他应用（和用户）访问，以及数据需要多少空间等。

Android 的数据存储方式包括：

- 共享首选项

在键值对中存储私有原始数据。SharedPreferences 类提供了一个通用框架，可以保存和检索原始数据类型的永久性键值对。开发人员可以使用 SharedPreferences 来保存任何原始数据：布尔值、浮点值、整型值、长整型和字符串。此数据将跨多个用户会话永久保留，即使该应用已终止也会保留。

以下是在计算器中保存静音按键模式首选项的示例：

```
1.  public class Calc extends Activity {
2.      public static final String PREFS_NAME = "MyPrefsFile";
3.
4.      @Override
5.      protected void onCreate(Bundle state){
6.          super.onCreate(state);
7.          // ...
8.          // Restore preferences
9.          SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
10.         boolean silent = settings.getBoolean("silentMode", false);
11.         setSilent(silent);
12.     }
13.
14.     @Override
15.     protected void onStop(){
16.         super.onStop();
```

```
17.
18.         // We need an Editor object to make preference changes.
19.         // All objects are from android.context.Context
20.         SharedPreferences settings = getContext(). getSharedPreferences(PREFS
_NAME, 0);
21.         // Android7.0 及以上,可以使用以下方法创建 DE 路径下的 DB。默认是存放在 CE 下
22.         // SharedPreferences settings = getContext(). createDeviceProtectedSt
orageContext().getSharedPreferences(PREFS_NAME, 0);
23.
24.         SharedPreferences.Editor editor = settings.edit();
25.         editor.putBoolean("silentMode", mSilentMode);
26.
27.         // Commit the edits!
28.         editor.commit();
29.     }
30. }
```

- 内部存储

在设备内存中存储私有数据。可以直接在设备的内部存储中保存文件。默认情况下，保存到内部存储的文件是应用的私有文件，其他应用（和用户）不能访问这些文件。当用户卸载应用时，这些文件也会被移除。

创建私有文件并写入到内部存储的步骤：

（1）使用文件名称和操作模式调用 openFileOutput()。 这将返回一个 FileOutputStream。

（2）使用 write() 写入到文件。

（3）使用 close() 关闭流式传输。

```
1. String FILENAME = "hello_file";
2. String string = "hello world!";
3.
4. FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
5. fos.write(string.getBytes());
6. fos.close();
```

- 外部存储

在共享的外部存储中存储公共数据。在外部存储设备（例如 SD 卡）上创建的文件不受任何读取和写入权限的限制。对于外部存储设备中的内容，不仅用户可以将其移除，而且任何应用都可以对其进行修改，因此最好不要使用外部存储设备来存储敏感信息。参阅 6.2.11 ADRD-11. 不应把未加密的敏感数据存储到外部存储（SD 卡）。

外部存储设备中的数据不应该被应用程序信任，对外部存储设备中的数据要执行输入验证。

不要在动态加载前将可执行文件或类文件存储在外部存储设备中。如果应用程序确实需要从外部存储设备中检索可执行文件，请在动态加载前对这些文件执行签名和加密验证。

- SQLite 数据库

在私有数据库中存储结构化数据。创建新 SQLite 数据库的推荐方法是创建 SQLiteOpenHelper 的子类并覆盖 onCreate() 方法，在此方法中，您可以执行 SQLite 命令以创建数据库中的表。 例如：

```
1. public class DictionaryOpenHelper extends SQLiteOpenHelper {
2.
3.     private static final int DATABASE_VERSION = 2;
```

```
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE =
                "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
                KEY_WORD + " TEXT, " +
                KEY_DEFINITION + " TEXT);";

    DictionaryOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
        // Android7.0 及以上,可以使用以下方法创建 DE 路径下的 DB。默认是存放在 CE 下
        // super(context.createDeviceProtectedStorageContext(), DATABASE_NAME,
null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DICTIONARY_TABLE_CREATE);
    }
}
```

- 网络存储

在网络中使用网络服务器存储数据。敏感信息请务必选择恰当的加密方式进行数据加密。将敏感数据储存在服务器端取代储存在移动设备本地端，前提是安全联机机制已建立及服务器端储存的安全机制足够或优于移动设备端。

6.2.3 ADRD-03. 正确设置数据库文件路径

Android7.0 以下的 Android 版本，应用程序的数据库文件只能存储在 data/data/$package_name/databases 目录下面（$package_name 是该应的包名），且该目录禁止给 other 用户读写权限。

Android7.0 及以上，Google 引入 DirectBoot 模式，应用程序数据可分别指定存放的数据路径 CE、DE，默认为 CE。

CE 路径：data/user/0/$package_name。软连接到 data/data/，仅在开机（DirectBoot）解锁后方可进行读写。通过 Context.createCredentialProtectedStorageContext()获取 CE 的 Context

DE 路径：data/user_de/0/$package_name，在 DirectBoot 和解锁后均可读写。通过 context.createDeviceProtectedStorageContext()获取 DE 的 Context

其中，0 为 userID，即默认账户，vivo 的分身账户为 999。

应用可在 Mainfest 中声明如下属性，可将应用数据默认存放在 DE 中，并兼容 DirectBoot 下功能使用正常。

```
<application
    android:directBootAware="true"
    android:defaultToDeviceProtectedStorage="true">
```

6.2.4 ADRD-04. 控制数据库访问权限

数据库系统本身的文件及用户的数据文件需要严格控制访问权限，只有数据库进程运行帐户以及管理员帐户才具备读写权限；对数据库帐户授予的权限进行严格清晰的划分，所有数据库帐户只能具备执行其任务的最小权限。

6.2.5 ADRD-05. 多个应用程序共用的数据库通过 Content Provider 访问

ContentProviders 提供了一种结构存储机制，它可以限制你自己的应用，也可以允许其他应用程序进行访问。 如果你不打算向其他应用提供访问你的 ContentProvider 功能，那么在 manifest 中标记他们为 android:exported=false 即可。

要建立一个给其他应用使用的 ContentProvider，你可以为读写操作指定一个单一的 permission，或者在 manifest 中为读写操作指定确切的权限。建议你对要分配的权限进行限制，仅满足目前有的功能即可。通常新的权限在新功能加入的时候同时增加，会比把现有权限撤销并打断已经存在的用户更合理。

如果 Content Provider 仅在自己的应用中共享数据，使用签名级别 android:protectionLevel 的权限是更可取的。签名权限不需要用户确认，当应用使用同样的密钥获取数据时，这提供了更好的用户体验，也更好地控制了 Content Provider 数据的访问。

Content Providers 也可以通过声明 android:grantUriPermissions 并在触发组件的 Intent 对象中使用 FLAG_GRANT_READ_URI_PERMISSION 和 FLAG_GRANT_WRITE_URI_PERMISSION 标志提供更细致的访问。 这些许可的作用域可以通过 grant-uri-permission 进一步限制。

当访问一个 ContentProvider 时，使用参数化的查询方法，比如 query()，update()和 delete()来避免来自不信任源潜在的 SQL 注入。 注意，如果 selection 语句是在提交给方法之前先连接用户数据的，使用参数化的方法或许不够。 不要对“写”权限有一个错误的观念。 考虑“写”权限允许 sql 语句，它可以通过使用创造性的 WHERE 子句并且解析结果让部分数据的确认变为可能。攻击者可能在通话记录中通过修改一条记录来检测某个特定存在的电话号码，只要那个电话号码已经存在。如果 content provider 数据有可预见的结构，提供“写”权限也许等同于同时提供了“读写”权限。

6.2.5.1 编程建议

对 ContentProvider 的数据库内容进行加密，同时加上 signatureOrSystem 权限控制，避免第三方应用使用，提高安全性。

6.2.6 ADRD-06. 涉及敏感信息时必须实现对字段或数据库的加密

SQLite 是一个轻量的、跨平台的、开源的数据库引擎，它的读写效率、资源消耗总量、延迟时间和整体简单性上具有的优越性，使其成为移动平台数据库的最佳解决方案(如 Android、iOS)。Android 系统内置了 SQLite 数据库，并且提供了一整套的 API 用于对数据库进行增删改查操作

需要对各个数据库字段含义进行了解，并评估其中可能的安全问题，如果存在敏感信息，要考虑加密字段，或加密数据库。

将整个数据库整个文件加密,这种方式基本上能解决数据库的信息安全问题。目前已有的 SQLite 加密基本都是通过这种方式实现的。

目前流行的是一款开源的 SQLite 加密工具 SQLCipher ，微信也在使用。 SQLCipher 是完全开源的，其代码托管在 github 上。SQLCipher 使用 256-bit AES 加密，由于其基于免费版的 SQLite，主要的加密接口和 SQLite 是相同的，也增加了一些自己的接口。它有一个缺点就是使用该库之后会导致 Apk 会变大。

6.2.6.1 合规代码示例

```
db = sqLiteOpenHelper.getWritableDatabase(password);
```

Adb shell code

```
cd /data/data/com.example.sqlcipherdemo/databases
sqlite3 demo.db
.table

Error: file is encrypted or is not a database
```

结果：Error: file is encrypted or is not a database，可以发现找不到数据库文件，数据库文件已经被包装过了。

6.2.7 ADRD-07. 不应使用全局可读模式和全局可写模式创建数据库

针对 SharedPreferences，谷歌原生提供了几种模式供应用选择创建文件；但是对于 MODE_WORLD_WRITEABLE 和 MODE_WORLD_READABLE 在 Android N 以后的版本已经不再支持了，但兼容 Android N 以下的版本的应用，要考虑该问题。请参阅 6.2.10 ADRD-10. 应以 MODE_PRIVATE 类型创建私有文件。

6.2.7.1 不合规代码示例

Java code

```
SharedPreferences sharePref = getApplicationContext().getSharedPreferences("test_sp", Context.MODE_WORLD_WRITEABLE);
```

Adb shell code

```
PD1718:/data/data/com.example.createSharePref/shared_prefs # ls -al
ls -al
total 32
drwxrwx--x 2 u0_a146 u0_a146 4096 2017-10-29 21:27 .
drwxr-x--x 5 u0_a146 u0_a146 4096 2017-10-29 21:27 ..
-rw-rw--w- 1 u0_a146 u0_a146  140 2017-10-29 21:27 test_sp.xml
```

由于该 shared_pref 文件针对 other 用户具有写的权限，恶意程序可以通过如下的方法来篡改应用的 sp 文件：

```
Context context = createPackageContext("com.example.createSharePref", CONTEXT_IGNORE_SECURITY);
SharedPreferences sharePref = context.getSharedPreferences("test_sp", Context.MODE_WORLD_WRITEABLE);
```

6.2.8 ADRD-08. 禁止明文存储敏感数据

用户密码等关键信息明文存储导致用户信息泄露的案例不少，如比较典型的是 CSDN 的用户信息泄露。明文包含更多的信息，用户密码不能通过在代码里面进行编码运算后直接存数据库。

Android 应用程序的 assets 目录下，不能明文存储敏感数据，如程序配置信息，用户口令等。

对于密钥的存储，建议使用矛盾 SDK 加密后存储，或者使用 AndroidKeystore 存储。

保密要求高的数据，可以使用 SHA256，SHA256 在密文够复杂的情况下，用字典法破解很难，如果再限制验证数，基本不可能被破解，当然，如果密码很简单如 123456 这样的，也很容易破解。

保密要求高的数据也可以考虑使用 3DES 或 AES，暴力破解 MD5 比较容易，但穷举 DES 密钥几乎是不可能的任务，可以使用 3DES，密钥扩展为 128 位。现在 DES 已经逐渐退出应用领域了，AES 是更好的选择。进一步了解加解密算法，请参阅 6.6 加解密安全规则。

6.2.8.1 不合规代码示例

以下示例显示了在 PackageInstaller/assets/authentication_info.xml 中存储的黑白名单配置信息。

```
1.  {
2.      "whitelist_imei": ["865407010000009", "286988609700019", "286988683200019
"],
3.      "default_validity" : 0,
4.      "adb_valitity" : 600000,
5.      "whitelist_pkgname" : ["com.tencent.android.qqdownloader", "com.baidu.app
search"],
6.      "support_all_allow" : 1,
7.      "auth_type" : 1,
8.      "blacklist_pkgname" : ["com.mycheering.onekeyinstall", "com.droidyuri.ins
tallerclient"],
9.      "blacklist_control_time" : 15,
10.     "http_type" : 0,
11.     "special_launcher" : ["com.moxiu.launcher"]
12. }
```

6.2.8.2 合规代码示例

6.2.8.3 编程建议

始终存储并比较敏感数据的哈希，而永远不要存储明文密码本身。在进行哈希计算前，对每个密码应用随机且特定的盐值。使用加密的强哈希算法，例如 SHA-2 系列之一。

6.2.9 ADRD-09. 使用参数化的 SQL 语句防止 SQL 注入攻击

下面的代码，当域的用户名/密码行不存在时，使用参数化的 SQL 语句将数据插入本地数据库。当该域的用户名/密码行已经存在时，使用动态字符串链接构建 UPDATE SQL 语句。恶意网页控制 b 变量（用户名）和 a 变量（host），但不直接控制 c 变量（密码），因为密码是被加密编码过的。

6.2.9.1 不合规代码示例

```
1.  Cursor v1;
2.  SQLiteDatabase v0 = g.a().d();
3.  String v2 = "select * from mxautofill where host =?";
4.  h.f();
5.  try {
6.      v1 = v0.rawQuery(v2, new String[]{this.a});
```

```
    if(v1.getCount() <= 0) {
        ContentValues v2_1 = new ContentValues();
        v2_1.put("host", this.a);
        v2_1.put("username", this.b);
        v2_1.put("password", this.c);
        v0.insert("mxautofill", null, v2_1);
    }
    else {
        v1.moveToFirst();
        v1.getColumnIndexOrThrow("host");
        v2 = "update mxautofill set username = '" + this.b + "',passwrd = '"
+ this.c + "' where host = '" + this.a + "'";
        h.f();
        v0.execSQL(v2);
    }
}
```

攻击者可以构建一个 HTML 页面，通过调用 catchform 方法来利用 SQL 漏洞。攻击者尝试使用浏览器之前存储的信息来自动填充信息，因为 SQL 注入与 UPDATE 语句相关联，而不是最初的 INSERT 语句。因此攻击者可能选择流行的 URL 作为 documentURI 的值。

```
<html>
    <body>
        <script>
            var json = '{"documentURI":"https://accounts.google.com/","inputs
":[{"id":"username","name":"username","value":"loginsqltest@gmail.com''-alert(''SqlT
est:''+document.domain)-'''--"},{"id":"password","name":"password","value":"fakepass
word"}]}';
            mxautofill.catchform(json);
        </script>
    </body>
</html>
```

当用户访问恶意页面时，会提示用户“save your account？”，并且用户在 SQL 注入漏洞被利用之前点击 “Yes”。然后浏览器执行以下 SQL 语句。恶意程序在用户名字段注入 JavaScript，然后使用 SQL 注入注释掉其它的 SQL 语句，包括 WHERE 子语句，以便将更新限制为只有一行。

```
update mxautofill set username = 'loginsqltest@gmail.com''-alert(''SqlTest:''
+document.domain)-'''-- ',password = '3tJIh6TbL87pyKZJOCaZag%3D%3D' where host = 'ac
counts.google.com'
```

SQLite 数据库 mxautofill 表中的所有行会被更新了。就这样，本地 SQLite 数据库已被篡改。攻击者同样可以通过利用 SQL 漏洞远程提取 mxautofill 表中用户名和密码这类敏感用户信息。

6.2.9.2 合规代码示例

```
SQLiteStatement sqLiteStatement = db.compileStatement("insert into msgTable(u
id, msg) values(?, ?)");
sqLiteStatement.bindLong(1, 12);
sqLiteStatement.bindString(3, "text");
```

```
4. long newRowId = sqLiteStatement.executeInsert();
```

6.2.10 ADRD-10. 应以 MODE_PRIVATE 类型创建私有文件

文件操作模式有以下四种：

- MODE_PRIVATE 将创建文件（或替换相同名称的文件），并使其对您的应用程序是私有的。
- MODE_APPEND 如果文件已经存在，则将数据写入现有文件的末尾，而不是将其删除。
- MODE_WORLD_READABLE 所有程序均可读该文件数据（不建议）
- MODE_WORLD_WRITEABLE 即所有程序均可写入数据（不建议）

6.2.10.1 合规代码示例

```
1. String content = "abcdefghigk";
2. try {
3.     FileOutputStream os = openFileOutput("internal.txt", MODE_PRIVATE);
4.     // 写数据
5.     os.write(content.getBytes());
6.     os.close();// 关闭文件
7. } catch (FileNotFoundException e) {
8.     e.printStackTrace();
9. } catch (IOException e) {
10.    e.printStackTrace();
11. }
```

6.2.10.2 编程建议

不要创建允许其他应用程序具有读写权限的文件，以 MODE_PRIVATE 类型创建私有文件。

所有应用程序的外部存储的私有文件都放在 SD 卡根目录的 Android/data/下，目录形式为：/mnt/sdcard/Android/data/<package_name>/files,可以通过函数 Context.getExternalFilesDir()函数获得该目录。

6.2.11 ADRD-11. 不应把未加密的敏感数据存储到外部存储（SD 卡）

使用 sdcard 存储的数据，不限制只有本应用访问，任何可以有访问 Sdcard 权限的应用均可以访问。私有文件如果是存储在外部存储，也能被其他程序（包括恶意程序）访问。外部存储上，应用私有文件的价值在于卸载之后，这些文件也会被删除。

敏感的数据文件，也不能未经加密存储，如 AI 模型文件，一般格式为 tensorflow、snpe 等通用框架的文件格式，如果被导出，将导致模型算法和参数的泄露。

数据分类分级请参阅《vivo 用户数据分类分级规范》。

AndroidManifest.xml

```
1. <!-- 在 SD 卡中创建与删除文件权限 -->
2. <uses-permission
3.     android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS">
4. </uses-permission>
5.
```

```
<!-- 向 SD 卡中写入数据权限 -->
<uses-permission
      android:name="android.permission.WRITE_EXTERNAL_STORAGE">
</uses-permission>
```

6.2.12 ADRD-12. 不应在使用“android:sharedUserId”属性的同时对应用使用测试签名

通过 Shared User id,拥有同一个 User id 的多个 APK 可以配置成运行在同一个进程中。所以默认就是可以互相访问任意数据，也可以配置成运行成不同的进程，同时可以访问其他 APK 的数据目录下的数据库和文件。就像访问本程序的数据一样。

不应在使用“android:sharedUserId”属性的同时，对应用使用测试签名，否则其他应用拥有“android:sharedUserId”属性值和测试签名时，将会访问到内部存储文件数据。

6.2.13 C-13. 敏感信息不应以全局变量的形式存储及传递

全局变量形式存储的数据内存地址是固定的，且具有较长的生命周期，很容易被黑客追踪从而导致信息泄漏。如果是密码及密钥之类的敏感信息泄漏，直接会导致安全类问题。而局部变量是在栈中分配，动态内存是在堆中分配，这类分配空间地址是随机的，能够更好的保护敏感信息，因此敏感数据必须以局部变量或动态内存的形式存储及传递

6.2.13.1 不合规代码示例

```
char key[128];
void decode(char* inputData,char* outBuf){
    getkey(&key);
    do_decode(key, inputData, outBuf);
}
```

6.2.13.2 合规代码示例

```
void decode(char* inputData,char* outBuf) {
char key[128];
    getkey(&key);
    do_decode(key, inputData, outBuf);
}
void decode(char* inputData, char* outBuf) {
    char* key=(char*)malloc(128);
    getkey(&key);
    do_decode(key,inputData,outBuf);
    free(key);
}
```

6.2.14 ADRD-14. 使用敏感数据字段标识用户时应先做脱敏处理

作为用户标识字段需要提供给第三方或进行网络传时，不能直接使用明文的敏感数据字段（如 IMEI，手机号码等），应先做加密或脱敏处理。可以使用用不可逆算法（如 hash）重新编码后再使用，或者使用安全 SDK 进行重编码。例如将 IMEI 做不可逆的编码后，作为唯一标识符使用。

6.2.15 ADRD-15. Direct Boot 模式的应用敏感数据应存储到凭据保护存储区

在 Android7.0（不含 Android7.0）以前，Android 用的是全盘加密 FDE，而 7.0 及更高版本支持文件级加密 (FBE)。采用文件级加密时，可以使用不同的密钥对不同的文件进行加密，并且可以对加密文件单独解密。

借助文件级加密，Android 7.0 中引入了一项称为 DirectBoot 的新功能。该功能处于启用状态时，已加密设备在启动后将直接进入锁定屏幕。

为了支持 Dierect Boot 模式，系统提供了两个存储数据的区域：

- CE（Credential encrypted storage）

凭据保护存储区，凭据是指用户输入的锁屏密码，密码就是解锁的凭据。），也就是传统的 data/data/xxx 也即 data/user/0/xxx，data/user/0 是软链接到 data/data 的，所以其实是同一个目录，是默认存储数据的目录，仅在用户解锁手机后可用。

```
PD1728:/ # ls -al /dSystem.err: ata/user
ls -al /dSystem.err: ata/user
total 36
drwx--x--x   4 system system 4096  2018-07-01 08:00 .
drwxrwx--x  57 system system 4096  2018-08-27 09:16 ..
lrwxrwxrwx   1 root   root     10  1970-07-27 09:09 0 -> /dSystem.err: ata/dS
ystem.err: ata
drwxrwx--x 251 system system 12288 2018-07-01 08:00 10
drwxrwx--X 201 system system 12288 2018-08-27 09:16 11
```

- DE（Device encrypted storage）

设备保护存储区，指的就是根据设备硬件参数生成的密码保护。也就是 data/user_de/，主要对应的就是 Direct Boot 使用的存储空间。在 Direct Boot 模式下（即解锁手机前）和用户解锁手机后都可以使用的存储空间。

```
PD1728:/dSystem.err: ata/user_de/0 # ls
ls
android
android.ext.services
android.ext.shared
com.UCMobile
com.aliyun.ams.tyid
com.aliyun.uuid
com.android.BBKClock
```

Dierect Boot 的模式下可以访问 DE 的数据，但无法访问 CE 的数据，用户输入密码解锁前如果应用访问了 CE 的数据会出 IO 异常。

如果应用适配了 Direct Boot 模式，应尽量把敏感数据存储到 CE（凭据保护存储区）。例如，在短信应用中，应用可以存储一个仅能访问服务器上新消息数量的访问令牌，而其它的个人敏感信息（例如完整的短信历史记录和读/写访问令牌）仍应保存在凭据保护存储区（CE）中。

```
# DE 目录
AdministrSystem.err: ator@tgdn-17297 ~
$ adb shell ls -al dSystem.err: ata/user_de/0/com.vivo.daemonService/shared_prefs
total 20
drwxrwx--x 2 system system 4096 2018-07-05 00:00 .
drwx------ 5 system system 4096 2018-07-01 08:00 ..
-rw-rw---- 1 system system  701 2018-07-05 00:00 EWarranty_prefs.xml
-rw-rw---- 1 system system  119 2018-07-01 08:00 desktop_control_sp_config.xml
-rw-rw---- 1 system system  138 2018-07-01 08:00 night_pearl_updSystem.err: ate_default_time.xml

# CE 目录
AdministrSystem.err: ator@tgdn-17297 ~
$ adb shell ls -al dSystem.err: ata/dSystem.err: ata/com.vivo.daemonService/shared_prefs
total 20
drwxrwx--x 2 system system 4096 2018-07-01 08:00 .
drwx------ 7 system system 4096 2018-07-01 08:00 ..
-rw-rw---- 1 system system  213 2018-07-01 08:00 appbehavior_info.xml
-rw-rw---- 1 system system   65 2018-07-01 08:00 unifiedconfig_info.xml
```

了解更多关于 FBE 和 DirectBoot 的信息请参阅适配 DirectBoot 注意事项。

6.2.16 ADRD-16. 禁止新增或复用 Setting 字段以存储安全级别较高的数据

手机在不 root 的情况下，也可以通过 adb shell 指令更改 Setting 中 Secure 和 System 两个表里面的对应字段的值。

比如，更改自动锁屏时长为 5 分钟：

Adb shell code

```
adb shell settings put system screen_off_timeout 300000
```

6.2.17 不应把解密的结果存入缓存文件

敏感数据和文件解密后，应该直接在内存中使用。不能为了下次使用的速度快，而把解密后的中间结果缓存下来。缓存的结果如果以文件的形式存储，可能被攻击获取利用，导致加密文件的泄露。

6.2.17.1 不合规代码示例

6.2.17.2 合规代码示例

6.2.18 使用敏感数据文件时应进行文件的完整性校验

使用敏感的数据文件（或敏感数据的文件）时应对文件做完整性校验，防止文件被篡改而引发其它安全问题。保存在手机系统的文件可能被恶意获取，修改后重新导入，替换原文件。此时如果直接接在运行，可能会导致未知的错误。比如预训练的算法模型文件，如果被篡改后继续执行，会导致模型预测的结果错误，甚至输出恶意模型设定的结果。

6.2.18.1 不合规代码示例

6.2.18.2 合规代码示例

6.3 调试与日志安全

6.3.1 ADRD-01. 发布的应用程序 debuggable 应设置为 false

当 debuggable 标志值为 true 时，该程序可被任意调试，导致 APP 被恶意攻击者控制。

```
android:debuggable=false
```

6.3.2 ADRD-02. 禁止在日志上输出敏感信息

Android 为应用程序提供了输出日志记录信息和获取日志输出的功能。 应用程序可以使用 android.util.Log 类将信息发送到日志输出。要获得日志输出，应用程序可以执行 logcat 命令。日志输出级别包括：

Log.d (Debug)
Log.e (Error)
Log.i (Info)
Log.v (Verbose)
Log.w (Warn)

日志输出语句如下：

```
Log.v("method", Login.TAG + ", str1=" + str1);
Log.v("method", Login.TAG + ", str2=" + str2);
```

在 AndroidManifest.xml 中声明 READ_LOGS 权限，以便应用程序可以读取日志输出：

```
<uses-permission android:name="android.permission.READ_LOGS"/>
```

在应用程序调用 logcat：

```
Process mProc = Runtime.getRuntime().exec(new String[]{"logcat", "-d", "method:V *:S$Bc`W^(B)"});
BufferedReader mReader = new BufferedReader(new InputStreamReader(proc.getInputStream()));
```

在 Android 4.0 之前，任何具有 READ_LOGS 权限的应用程序都可以获得所有其他应用程序的日志输出。在 Android 4.1 之后，READ_LOGS 权限已被更改。即使具有 READ_LOGS 权限的应用程序也无法从其他应用程序获取日志输出。然而，通过将 Android 设备连接到 PC，可以获得其他应用程序的日志输出。

因此，重要的是应用程序不应该将敏感信息发送到日志输出。

6.3.2.1 不合规代码示例

如下代码，它发送访问令牌以以纯文本格式记录输出。

```
Log.d("Facebook-authorize", "Login Success! access_token="
        + getAccessToken() + " expires="
        + getAccessExpires());
```

下面这个例子。 Android 的天气预报应用将用户的位置数据发送到日志输出，如下所示：

```
I/MyWeatherReport( 6483): Re-use MyWeatherReport data
```

```
2. I/ ( 6483): GET JSON: http://example.com/smart/repo_piece.cgi?arc=0&lat=26.209026&lon=127.650803&rad=50&dir=-999&lim=52&category=1000
```

如果用户使用 Android OS 4.0 或之前版本，则具有 READ_LOGS 权限的其他应用程序可以在 AndroidManifest.xml 文件中声明 ACCESS_FINE_LOCATION 权限，从而获取用户的位置信息。导致用户敏感信息泄漏。

6.3.2.2 合规代码示例

```
1. public class LogActivity extends Activity {
2.     private static final String LOG_TAG = "ProGuardActivity";
3.     private static final boolean DEBUG = true;
4.     @Override
5.     public void onCreate(Bundle savedInstanceState) {
6.         super.onCreate(savedInstanceState);
7.         setContentView(R.layout.activity_proguard);
8.         // *** POINT 1 ***敏感信息不能用 Log.e()/w()/i(),System.out/erro 输出
9.         Log.e(LOG_TAG, "Not sensitive information (ERROR)");
10.        Log.w(LOG_TAG, "Not sensitive information (WARN)");
11.        Log.i(LOG_TAG, "Not sensitive information (INFO)");
12.        // *** POINT 2 ***开发调试阶段必须输出敏感信息时 ，用 Log.d()/v()
13.        //*** POINT 3 ***在发行 release 版本的时候，通过 Proguard 或者变量控制删除掉 Log.d()/v()所有 log。
14.        if (DEBUG) Log.d(LOG_TAG, "sensitive information (DEBUG)");
15.        if (DEBUG) Log.v(LOG_TAG, "sensitive information (VERBOSE)");
16.    }
17. }
```

6.3.3 ADRD-03. 程序发布时通过 Proguard 者变量控制 Log.d()/v()的 LOG 输出

proguard-protect.txt

```
1. // *** POINT 3 ***在发行 release 版本的时候，通过 Proguard 或者变量控制删除掉 Log.d()/v()所有 log。
2. -assumenosideeffects class android.util.Log {
3.     public static int d(...);
4.     public static int v(...);
5. }
```

6.4 网络安全

Android 软件通常使用 WIFI 网络与服务器进行通信。WiFi 并非总是可靠的，例如，开放式网络或弱加密网络中，接入者可以监听网络流量；攻击者可能 自己设置 WIFI 网络钓鱼。此外，在获得 root 权限后，还可以在 Android 系统中监听网络数据。

6.4.1 JAVA-01. 在 SSL/TLS 上应验证服务器证书

使用 SSL/TLS 协议进行安全通信的 Android 应用程序应正确验证服务器证书。 基本验证包括：

- 验证 X.509 证书的主题（CN）和 URL 匹配
- 验证证书是否由受信任的 CA 签名
- 验证签名是否正确
- 验证证书是否已过期

Android SDK 4.0 及更高版本提供了实现建立网络连接功能的软件包。例如，通过使用 java.net，javax.net，android.net 或 org.apache.http，开发人员可以创建服务器套接字或 HTTP 连接。org.webkit 提供了实现 Web 浏览功能所必需的功能。

开发人员可以自由地定制其 SSL 实现。开发人员应适当使用适用于应用程序和应用程序环境的 SSL。如果 SSL 未正确使用，用户的敏感数据可能会通过易受攻击的 SSL 通信渠道泄漏。

以下是几种 SSL 不安全使用的模式：

- 信任所有证书：开发人员实施 TrustManager 接口，以便它能够信任所有的服务器证书（无论谁签名，什么 CN 等等），攻击者可以利用中间人攻击窃听或干扰会话中的所有通信。
- 允许所有主机名：应用程序不会验证是否为客户端连接的 URL 发出证书。例如，当客户端连接到 example.com 时，它将接受为 some-other-domain.com 发出的服务器证书，攻击者可以利用中间人攻击窃听或干扰会话中的所有通信。
- 混合模式/无 SSL：开发人员在同一应用程序中混合安全和不安全的连接，或根本不使用 SSL。可能发生重定向或欺骗攻击，允许具有有效证书的恶意主机冒充受信任的主机
- 禁用证书撤销检查，可以连接到使用几乎可以肯定是恶意证书的被撤销证书的系统。

在 Android 上，建议使用 HttpURLConnection 进行 HTTP 客户端实现。

6.4.1.1 不合规代码示例

以下代码实现了继承 javax.net.ssl.SSLContext 的自定义 MySSLSocketFactory 类：

在该示例中，checkClientTrusted（）和 checkServerTrusted（）被重写以进行空白实现，SSLSocketFactory 不会验证 SSL 证书。MySSLSocketFactory 类用于在应用程序的另一部分创建一个 HttpClient 实例。

DefineRelease 类的静态成员 sAllowAllSSL 在其静态构造函数中初始化为 true。这样可以使用 SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER。 因此，在建立 SSL 连接时禁用了 SSL 连接从页不进行主机名验证，这样就相当于信任了所有证书。

```
1.  public class extends SSLSocketFactory {
2.      SSLContext sslContext;
3.      public MySSLSocketFactory (KeyStore truststore) throws NoSuchAlgorithmExc
eption, KeyManagementException,
4.      KeyStoreException, UnrecoverableKeyException {
5.          super(truststore);
6.          this.sslContext = SSLContext.getInstance("TLS");
7.          this.sslContext.init (null, new TrustManager[] { new X509TrustManager
() {
```

```
8.                  public void checkClientTrusted(X509Certificate[] chain, Strin
g authType) throws CertificateException
9.                  {
10.                     // ...
11.                 }
12.                 public void checkServerTrusted(X509Certificate[] chain, Strin
g authType) throws CertificateException
13.                 {
14.                     // ...
15.                 }
16.                 public X509Certificate[] getAcceptedIssuers() {
17.                     return null;
18.                 }
19.             }}, null);
20.     }
21.     public Socket createSocket() throws IOException {
22.         return this.sslContext.getSocketFactory().createSocket();
23.     }
24.     public Socket createSocket(Socket socket, String host, int port, boolean
autoClose) throws IOException, UnknownHostException {
25.         return this.sslContext.getSocketFactory().createSocket(socket, host,
port, autoClose);
26.     }
27. }
28.
29. public static HttpClient getNewHttpClient() {
30.     DefaultHttpClient v6;
31.     try {
32.         KeyStore v5 = KeyStore.getInstance(KeyStore.getDefaultType());
33.         v5.load(null, null);
34.         MySSLSocketFactory mySSLScoket = new MySSLSocketFactory(v5);
35.         if(DefineRelease.sAllowAllSSL) {
36.             ((SSLSocketFactory)mySSLScoket).setHostnameVerifier(SSLSocketFact
ory.ALLOW_ALL_HOSTNAME_VERIFIER);
37.         }
38.         BasicHttpParams v2 = new BasicHttpParams();
39.         HttpConnectionParams.setConnectionTimeout(((HttpParams)v2), 30000);
40.         HttpConnectionParams.setSoTimeout(((HttpParams)v2), 30000);
41.         HttpProtocolParams.setVersion(((HttpParams)v2), HttpVersion.HTTP_1_1);

42.         HttpProtocolParams.setContentCharset(((HttpParams)v2), "UTF-8");
43.         SchemeRegistry v3 = new SchemeRegistry();
44.         v3.register(new Scheme("http", PlainSocketFactory.getSocketFactory(),
 80));
45.         v3.register(new Scheme("https", ((SocketFactory)mySSLScoket), 443));
```

```
46.         v6 = new DefaultHttpClient(new ThreadSafeClientConnManager(((HttpPara
ms)v2), v3), ((HttpParams)v2));
47.     }
48.     catch(Exception v1) {
49.         v6 = new DefaultHttpClient();
50.     }
51.     return ((HttpClient)v6);
52. }
```

6.4.1.2 合规代码示例

合规解决方案可能会有各种形式，具体取决于实际实现。以下代码检查证书的可信性。

```
1.  // 获取系统凭证库里的 CA 证书，查验证书的可信性
2.  X509Certificate cert = Certificate; // 待查验的证书
3.  X509Certificate CAcert;
4.  KeyStore mCAStore = KeyStore.getInstance("AndroidCAStore");
5.  mCAStore.load(null, null);
6.  Enumeration<String> als = mCAStore.aliases();
7.  while (als.hasMoreElements()) {
8.      CAcert = (X509Certificate) mCAStore.getCertificate(als.nextElement());
9.      try {
10.         cert.verify(CAcert.getPublicKey());
11.         //
12.         break;
13.     } catch(Exception e) {
14.         CAcert = null;
15.     }
16. }
17. if (CAcert == null) {
18.     Log.i("CA not found.", "He he");
19. }
```

6.4.2 JAVA-02. 证书验证失败禁止继续通信

有些代码在出现证书验证错误之后，允许应用程序继续通信，也可以使用 HTTPS 与 Web 服务器。由于它们作为通过 HTTPS 与使用私有证书的 Web 服务器进行通信的方式被引入，使用了这些代码的应用程序大多数都容易受到中间人的袭击。

6.4.2.1 不合规代码示例

HTTPS 通信实施中同样存在许多缺陷，如以下几个代码片段，是导致易受攻击的 HTTPS 通信如下所示。

```
1.  // Risk: Case which creates empty TrustManager
2.  TrustManager tm = new X509TrustManager() {
3.
4.      @Override
5.      public void checkClientTrusted(X509Certificate[] chain,
```

```
        String authType) throws CertificateException {
            // Do nothing -> accept any certificates
        }
        @Override
        public void checkServerTrusted(X509Certificate[] chain,
        String authType) throws CertificateException {
            // Do nothing -> accept any certificates
        }
        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return null;
        }
};


// Risk:Case which creates empty HostnameVerifier
HostnameVerifier hv = new HostnameVerifier() {
        @Override
        public boolean verify(String hostname, SSLSession session) {
            // Always return true -> Accespt any host names
            return true;
        }
};

// Risk:Case that ALLOW_ALL_HOSTNAME_VERIFIER is used.
SSLSocketFactory sf;
// ...
sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
```

这时强烈建议用私有证书通过 HTTPS 进行通信。

6.4.2.2 合规代码示例

建议用私有证书通过 HTTPS 进行通信。示例请参阅 6.4.7 JAVA-07. 敏感数据应通过 HTTPS 传输并对证书进行验证。

6.4.3 ADRD-03. HTTPS（公有证书）的证书校验

https 在理论上是可以抵御中间人攻击和 DNS 劫持，但是由于开发过程中的编码不规范，导致 https 可能存在攻击风险，攻击者可以解密、篡改 https 数据和劫持 https。

https 的开发过程中常见的安全缺陷：

- 在自定义实现 X509TrustManager 时，checkServerTrusted 中没有检查证书是否可信，导致通信过程中可能存在中间人攻击，造成敏感数据劫持危害。
- 在自定义实现 HostnameVerifier 时，没有在 verify 中进行严格证书校验，导致通信过程中可能存在中间人攻击，造成敏感数据劫持危害。
- 在 setHostnameVerifier 方法中使用 ALLOW_ALL_HOSTNAME_VERIFIER，信任所有 Hostname，导致通信过程中可能存在中间人攻击，造成敏感数据劫持危害。

6.4.3.1 不合规代码示例

（1）禁止使用 HttpClient 进行 https 通信，原因：

- HttpClient 已停止维护，系统从 android2.3 已使用 HttpUrlConnection 代替 HttpClient
- HttpClient 不支持 SNI 类型的 https，在证书校验时出现如下异常：

```
1. System.err: javax.net.ssl.SSLException: hostname in certificate didn't match: <exappupgrade.vivoglobal.com> != <*.vivo.com.cn> OR <*.vivo.com.cn> OR <vivo.com.cn>
2. System.err:     at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:190)
3. System.err:     at org.apache.http.conn.ssl.BrowserCompatHostnameVerifier.verify(BrowserCompatHostnameVerifier.java:59)
4. System.err:     at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:119)
5. System.err:     at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:100)
6. System.err:     at org.apache.http.conn.ssl.SSLSocketFactory.createSocket(SSLSocketFactory.java:393)
7. System.err:     at org.apache.http.impl.conn.DefaultClientConnectionOperator.openConnection(DefaultClientConnectionOperator.java:170>
8. System.err:     at org.apache.http.impl.conn.AbstractPoolEntry.open(AbstractPoolEntry.java:169)
9. System.err:     at org.apache.http.impl.conn.AbstractPooledconnAdapter.open(AbstractPooledConnAdapter.java:124)
10. System.err:     at org.apache.http.impl.client.DefaultRequestDirector.execute(DefaultRequestDirector.java:365)
11. System.err:     at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:595)
12. System.err:     at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:518)
13. System.err:     at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:496)
14. System.err:     at com.vivo.upgradelibrary.network.QueryNetThread.doHttpGetRequest(Unknown Source)
15. System.err:     at com.vivo.upgradelibrary.network.QueryNetThread.tryToDoGetRequest(Unknown Source)
16. System.err:     at com.vivo.upgradelibrary.network.QueryNetThread.run(Unknown Source)
17. VivoNetworkControlller: SystemUI|mWifiEnabled && mWifiConnectd = true
18. System.err: javax.net.ssl.SSLException: hostname in certificate didn't match: <exappupgrade.vivoglobal.com> != <*.vivo.com.cn> OR <*.vivo.com.cn> OR <vivo.com.cn>
19. System.err:     at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:190)
20. System.err:     at org.apache.http.conn.ssl.BrowserCompatHostnameVerifier.verify(BrowserCompatHostnameVerifier.java:59)
```

```
    System.err:    at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:119)
    System.err:    at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:100)
```

（2）使用 HttpURLConnection 进行 https 通信时的错误用法

```
private HttpURLConnection createUrlConnection(URL url) throws IOException {
    HttpURLConnecticn httpURLConnection = (HttpURLConnection) url.openConnection();
    if (httpURLConnection instanceof HttpsURLConnection) {
        trustAllHosts((HttpsURLConnecticn) httpURLConnection);
    }
    return httpURLConnection;
}

private static void trustAllHosts(final HttpURLConnecticn httpsURLConnection) {
    SSLConntext sslContext = null;
    try {
        sslContext = SSLContext.getInstance("TLS");
        if (sslContext != null) {
            TrustManager tm = new X509TrustManager() {
                public X509Certificate[] getAcceptedIssuers() {
                    Log.i("test-https", "getAcceptedIssuers");
                    return null;
                }

                public void checkClientTrusted(X509Certificate[] chain, String authType) throws CertificateException {
                    Log.i("test-https", "checkClientTrusted");
                }

                // 没有对服务器证书进行校验
                public void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException {
                    Log.i("test-https", "checkServerTrusted");
                }
            };
            sslContext.inir(null, new TrustManager[]{tm}, null);
        }

    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    } catch (KeyManagementException e) {
        e.printStackTrace();
    }

```

```
    // 不对主机进行 ssl 验证
    // 将校验内容设置到建立 SSLSocket 的 SSLSocketFactory 中
    if (sslContext != null) {
        httpURLConnection.setSSLSocketFactory(sslContext.getSocketFactory());

    }
}

// ......

// 设置信任所有主机
httpsURLConnection.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);

// 允许所有主机的验证
// 没有对服务器证书域名进行校验
HostnameVerifier hostnameVerifier = new HostnameVerifier(){
    @Override
    public boolean verify(String hostname.SSLSession session) {
        return true;
    }
};
httpsURLConnection.setHostnameVerifier(hostnameVerifier);
```

（3）使用 OkHttp 进行 https 通信时的错误用法

```
public static void getUnsafeOkHttpClient() {
    try {
        // 不校检证书链
        final X509TrustManager[] trustAllCerts = new X509TrustManager[] {
            new X509TrustManager() {
                @Override
                public void checkClientTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {
                    // 不校验客户端证书
                }
                @Override
                public void checkServerTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {
                    // 不校验服务端证书
                }
                @Override
                public X509Certificate[] getAcceptedIssuers() {
                    return new X509Certificate[0];
                }
            }
        };

```

```
21.         final SSLContext sslContext = SSLConntext.getInstance("TLS");
22.         sslContext.init(null,trustAllCerts, null);
23.
24.         final HostnameVerifier hostnameVerifier = new HostnameVerifier() {
25.             @Override
26.             public boolean verify(String s, SSLSession sslSession) {
27.                 // 未真正校验服务器端证书域名
28.                 return true;
29.             }
30.         };
31.
32.         OkHttpClient okHttpClient = new OkHttpClient.Builder().hostnameVerifi
er(hostnameVerifier)
33.                 .sslSocketFactory(sslContext.getSocketFactory(), trustAllCert
s[0].build();
34.
35.         String url = "https://certs.cac.washington.edu/CAtest/";
36.
37.         Request request = new Request.Builder().url(url).build();
38.         Call call =okHttpClient.newCall(request);
39.         call.enqueue(new Callback() {
40.             @Override
41.             public void onFailure(Call call, IOException e) {
42.                 Log.d("onFailure", e.getMessage());
43.             }
44.
45.             @Override
46.             public void onResponse(Call call, Response response) throws IOExc
eption {
47.                 final String res = response.body().string();
48.                 Log.d("onResponse",res);
49.             }
50.         });
51.     } catch(Exception ex) {
52.         ex.printStackTrace();
53.         throw new RuntimeException(ex);
54.     }
55. }
```

6.4.3.2 合规代码示例

公有证书没有特殊要求，尽量不要自定义证书和域名的校验，使用系统默认的校验（HttpUrlConnection、OkHttp 都有默认的校验处理）即可。

6.4.3.3 编程建议

检查应用中是否存在 https 全信任证书的方法,在手机不安装抓包工具(charles)的证书的前提下，使用抓包工具可以查看应用中 https 请求的内容，此 https 请求使用全信任证书。

6.4.4 ADRD-04. HTTPS（私有证书）的证书校验

私有证书：未经过官方权威机构（CA）认证，自定义的证书

使用私有证书，抓包工具无法查看 https 的内容，但需要在客户端预埋证书（如果服务器端证书到期或者因为泄露等其他原因需要更换证书，必须强制用户进行客户端升级），并且需要重写校验证书链 TrustManager 中的方法，否则的话会出现 javax.net.ssl.SSLHandshakeException: java.security.cert.CertPathValidatorException: Trust anchor for certification path not found。

6.4.4.1 不合规代码示例

请参阅 6.4.3 ADRD-03. HTTPS（公有证书）的证书校验的不合规代码示例。

6.4.4.2 合规代码示例

【HttpUrlConnection】

```
1.  public static String getSafeFromServer(final InputStream certStream){
2.      // certStream 为服务器端证书输入流
3.      String response = "";
4.
5.      String httpsUrl3 = "https://certs.cac.washington.edu/CAtest/";
6.
7.      final String httpsUrl = httpsUrl3;
8.
9.      new AsyncTask<String, Void, Boolean>() {
10.         @Override
11.         protected Boolean doInBackground(String... Params) {
12.             try {
13.                 // 以 X.509 格式获取证书
14.                 CertificateFactory certificateFactory = CertificateFactory.ge
tInstance("X.509");
15.                 Certificate cert =certificateFactory.generateCertificate(cerS
tream);
16.                 Log.d("cert key", ((X509Certificate)cert).getPublicKey().
17.
18.                 // 生成一个包含服务器端证书的 KeyStore
19.                 String keyStoreType = KeyStore.getDefaultType();
20.                 Log.d("keystore type",keyStoreType);
21.                 KeyStore keyStore = KeyStore.getInstance(keyStroeType);
22.                 keyStore.load(null,null);
23.                 keyStore.setCertificateEntry("cert",cert);
24.
25.                 // 用包含服务器证书的 KeyStore 生成一个 TrustManger
26.                 String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm
();
27.                 Log.d("tmfAlgorithm",tmfAlgorithm);
28.                 TrustManagerFactory trustManagerFactory = TrustManagerFactory.
getInstance(tmfAlgorithm);
```

```
29.                 trustManagerFactory.init(keyStore);
30.
31.                 // 生成一个使用我们的 TrustManager 的 SSLContext
32.                 SSLConntext sslContext = SSLConntext.getInstance("TLS");
33.                 sslContext.init(null, trustManagerFactory.getTrustManagers(),
 null);
34.
35.                 URL url = new URL(httpsUrl);
36.                 HttpsURLConnecticn httpsURLConnection = (HttpsURLConnecticn)
url.openConnection();
37.                 httpsURLConnection.setSSLSocketFactory(sslContext.getSocketFa
ctory());
38.                 InputStream in = httpsURLConnection.getInputStream();
39.                 copyInputStreamToOutputStream(in ,System.out);
40.
41.             }catch (CertificateException e){
42.                 e.printStackTrace();
43.             }catch (KeyStoreException e) {
44.                 e.printStackTrace();
45.             }catch (NoSuchAlgorithmException e) {
46.                 e.printStackTrace();
47.             }catch (IOException e) {
48.                 e.printStackTrace();
49.             }catch (KeyManagementException e) {
50.                 e.printStackTrace();
51.             }
52.             return true;
53.         }
54.     }.execute(httpsUrl);
55.
56.     return response;
57. }
```

【OkHttp】

```
1.  try {
2.      SSLSocketFactory sslSocketFactory = HttpsUtils.getSslSocketFactory(mConte
xt.getAssets().open("srca.cer"));
3.      OkHttpClient client = new OkHttpClient.Builder().sslSocketFactory(sslSock
etFactory).build();
4.      public static SSLSocketFactory getSslSocketFactory(InputStream...certific
ates)
5.      {
6.          try
7.          {
8.              CertificateFactory certificateFactory = CertificateFactory.getIns
tance("X.509");
```

```
            KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType()
);
            keyStore.load(null);
            int index = 0;
            for (InputStream certificate : certificates)
            {
                String certificateAlias = Integer.toString(index++);
                keyStore.setCertificateEntry(certificateAlias, certificateFac
tory.generateCertificate(certificate));
                try
                {
                    if(certificate != null)
                        certificate.close();
                }catch (IOException e) {
                }
            }
        }
        SSLConntext sslContext = SSLConntext.getInstance("TLS");
        TrustManagerFactory trustManagerFactory trustManagerFactory =
                    TrustManagerFactory.getInstance(TrustManagerFactory.getDe
faultAlgorithm());
        trustManagerFactory.init(null, trustManagerFactory.getTrustManagers(),
 new SecureRandom());
        return sslContext.getSocketFactory();
    }
}
```

6.4.5 JAVA-05. 敏感数据处理时不应使用回环地址

在处理敏感数据时，不应使用环回，即将网络通信连接到本地主机端口。本地主机端口可由设备上的其他应用程序访问，因此它们的使用可能导致敏感数据被显示。应使用安全的 Android IPC 机制，例如 HttpsURLConnection 类或 SSLSocket 类。同样，安全通信不应该被绑定到 INADDR_ANY 端口，因为这将导致应用程序能够在任何地方接收请求。有关这些问题的更多信息，请参阅：[Android 安全]部分使用网络。

有些应用使用 localhost 网络端口处理敏感的 IPC。我们不建议采用这种方法，因为设备上的其他应用也可以访问这些接口。相反，应使用可通过 Service 等进行身份验证的 Android IPC 机制。绑定到 INADDR_ANY 比使用回环地址还要糟糕，因为这样一来，该应用可能会收到任何位置发来的请求。

6.4.5.1 不合规代码示例

```
SSLSocketFactory sf;
// ...
sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
```

6.4.5.2 合规代码示例

6.4.6 JAVA-06. 敏感数据在传输前应进行加密

敏感数据在传输前尽量加密处理，保证数据的传输安全，敏感数据主要有两类：

- 与用户相关的敏感数据，例如，手机号、银行卡、身份证等
- 与应用相关的敏感数据，例如，配置信息、黑白名单等，此类数据在传输前必须进行加密处理

6.4.6.1 不合规代码示例

```
<?xml version="1.0" encoding="UTF-8"?>
 <app>
     <!-- non sensitive data-->
     <userid>USERID</userid>
     <!-- sensitive data-->
     <password>PASSW0RD</password>
</app>
```

上面的 xml 文件我们需要传输的原始数据。如果直接传输敏感信息，是非常危险的，所以需要采用加密机制。为了在客户端和服务器端达成统一，我们在 xml 文件中需要指定目前双方所采用的加密算法。同时，我们设计这样的加密密钥：其只在一次传输操作中有效，而一旦离开当前会话立即失效。这样保证了密钥的安全，使数据得到进一步保障。

6.4.6.2 合规代码示例

经过加密机制处理后，XML 将呈现为如下所示的形式。其中，SessionID 用于标识本次会话，cipher 属性表示客户端和服务器端一致认可的加密算法。这样，经过加密机制的处理，敏感信息 password 的内容以加密后的形式呈现。服务端收到解密的信息后进行解密使用。

```
<?xml version="1.0" encoding="UTF-8"?>
<app SessionID="App-001e3751a6e6-00000d34-000000004ba7589e-0">
    <!-- non sensitive data-->
    <userid>USERID</userid>
    <!-- sensitive data-->
    <password cipher="aes128">78DFC347E201F24742030E4E03B8A034C83A4F072EA78DF6C63A9AF8DF06E57D42D73DC00D3A01773D1AB8A9DBCE759CACC324BD23D141A0CE4F68FAE6332970FD272250014A1C1CC82EB1637487A430</password>
</app>
```

6.4.7 JAVA-07. 敏感数据应通过 HTTPS 传输并对证书进行验证

在 HTTP 事务中，发送和接收的信息可能被嗅探或被篡改，连接的服务器可能被伪装。最危险的是直接使用 HTTP 协议登录账户或交换数据。例如，攻击者在自己设置的钓鱼网络中配置 DNS 服务器，将软件要连接的服务器域名解析至攻击者的另一台服务器在，这台服务器就可以获得用户登录信息，或者充当客户端与原服务器的中间人，转发双方数据。

敏感信息必须通过 HTTPS 发送和接收。

Android 网络运行机制与其他 Linux 环境差别不大，关键是确保对敏感数据使用合适的协议，如使用 HttpsURLConnection 来保证网络流量安全。我们建议您在服务器支持 HTTPS 的情况下一律使用 HTTPS（而非 HTTP），因为移动设备经常会连接到不安全的网络（如公共 WLAN 热点）。

6.4.7.1 不合规代码示例

```
1.  public static HttpClient getWapHttpClient() {
2.      try{
3.          KeyStore trustStore=KeyStore.getInstance(KeyStore.getDefaultType());
4.          trustStore.load(null, null);
5.          SSLSocketFactory sf = new MySSLSocketFactory(trustStore);
6.          sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
7.          // 此处信任手机中的所有证书，包括用户安装的第三方证书
8.          HttpParams params = new BasicHttpParams();
9.          HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
10.         HttpProtocolParams.setContentCharset(params, HTTP.UTF_8);
11.         SchemeRegistry registry = new SchemeRegistry();
12.         registry.register(new Scheme(“http”, PlainSocketFactory.getSocketFact
ory(), 80));
13.         registry.register(new Scheme(“https”, sf, 443));
14.         ClientConnectionManager ccm = new ThreadSafeClientConnManager(params,
 registry);
15.         return new DefaultHttpClient(ccm, params);
16.     } catch (Exception e) {
17.         return new DefaultHttpClient();
18.     }
19. }
```

在客户端中覆盖 google 默认的证书检查机制（X509TrustManager），并且在代码中无任何校验 SSL 证书有效性相关代码：

```
1.  public class MySSLSocketFactory extends SSLSocketFactory {
2.      SSLContext sslContext = SSLContext.getInstance(“TLS”);
3.      public MySSLSocketFactory(KeyStore truststore) throws NoSuchAlgorithmExce
ption, KeyManagementException, KeyStoreException, UnrecoverableKeyException {
4.          super(truststore);
5.              TrustManager tm = new X509TrustManager() {
6.                  public void checkClientTrusted(X509Certificate[] chain, Strin
g authType) throws CertificateException {
7.              }
8.              // 客户端并未对 SSL 证书的有效性进行校验，并且使用了自定义方法的方式覆盖
android 自带的校验方法
9.              public void checkServerTrusted(X509Certificate[] chain, String au
thType) throws CertificateException {
10.             }
11.             public X509Certificate[] getAcceptedIssuers() {
12.                 return null;
```

```
            }
        };
        sslContext.init(null, new TrustManager[] { tm }, null);
    }
}
```

如果用户手机中安装了一个恶意证书，那么就可以通过中间人攻击的方式进行窃听用户通信以及修改 request 或者 response 中的数据。

6.4.7.2 合规代码示例

优先使用由知名 CA 机构颁发的公有证书，如果使用的公有证书，则可以使用简单的代码创建安全请求：

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

使用 CA 机构颁发的证书，时间和成本太高，往往证书可能存在以下的情况。

- 颁发服务器证书的 CA 是未知的
- 服务器证书未由 CA 签名，但是是自签名的
- 服务器配置缺少中间 CA

下面是一个从 InputStream 获取特定 CA 的示例，使用它创建一个 KeyStore，然后用于创建和初始化一个 TrustManager。TrustManager 是系统用来验证服务器上的证书的方式，通过使用一个或多个 CA 从 KeyStore 创建一个证书，这些将是该 TrustManager 信任的唯一 CA。

给定新的 TrustManager，该示例初始化一个新的 SSLContext，它提供了一个 SSLSocketFactory，可用于从 HttpsURLConnection 覆盖默认的 SSLSocketFactory。这样连接将使用私有 CA 进行证书验证。

```
// Load CAs from an InputStream
// (could be from a resource or ByteArrayInputStream or ...)
CertificateFactory cf = CertificateFactory.getInstance("X.509");
// From https://www.washington.edu/itconnect/security/ca/load-der.crt
InputStream caInput = new BufferedInputStream(new FileInputStream("load-der.crt"));
Certificate ca;
try {
    ca = cf.generateCertificate(caInput);
    System.out.println("ca=" + ((X509Certificate) ca).getSubjectDN());
} finally {
    caInput.close();
}

// Create a KeyStore containing our trusted CAs
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);
```

```
19.
20. // Create a TrustManager that trusts the CAs in our KeyStore
21. String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
22. TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
23. tmf.init(keyStore);
24.
25. // Create an SSLContext that uses our TrustManager
26. SSLContext context = SSLContext.getInstance("TLS");
27. context.init(null, tmf.getTrustManagers(), null);
28.
29. // Tell the URLConnection to use a SocketFactory from our SSLContext
30. URL url = new URL("https://certs.cac.washington.edu/CAtest/");
31. HttpsURLConnection urlConnection = (HttpsURLConnection)url.openConnection();

32. urlConnection.setSSLSocketFactory(context.getSocketFactory());
33. InputStream in = urlConnection.getInputStream();
34. copyInputStreamToOutputStream(in, System.out);
```

6.4.8 ADRD-08. 敏感数据禁止在 http 的 URL 中明文携带

在无法避免使用 http 的情形，禁止在 URL 中携带敏感数据，如 IMEI 等。

6.4.8.1 不合规代码示例

```
1.  private URL buildUrl(String baseUrl) {
2.      URL url = null;
3.
4.      // ....
5.
6.      String urlStr = String.format(Locale.getDefault(), "%s?sk=%s&st=%s&imei=%s
&em=%s&model=%s&adrVerName=%s&av=%d" + "&sysVer=%s&romVer=%s&elapsedtime=%d&cs=%d&pi
xel=%s&globalAppVersion=%s&appVersion=%s" + "&thAppVersionName=%s&thResType=%d×tamp=
%d&rpkEngineVersion=%d", baseUrl, sk, st, imei, em, model, adrVerName, av, sysVer, r
omVer, elapsedtime, cs, pixel, globalAppVersion,appVersion, thAppVersionName, thResT
ype, timestamp, rpkEngineVersion);
7.
8.      Log.i(TAG, "buildUrl: urlStr = " + urlStr);
9.      String encodedUrl = Util.encodeURL(mContext, urlStr);
10.     Log.i(TAG, "buildUrl: encodedUri = " + encodedUrl);
11.     try {
12.         url = new URL(encodedUrl);
13.     } catch (MalformedURLException e) {
14.         Logit.d(TAG, "create URL exception!", e);
15.         return null;
16.     }
17.
18.     return url;
```

```
}

// http://172.25.20.164:8579/reports.htm#v10025/p10003/fileInstanceId=1041280&defectInstanceId=354278&mergedDefectId=57482
public boolean doSearch() {
    // ....
    // mHasTask = true
    int len = 500;
    // com.vivo.globalsearch.model.task.search.SearchServerHelper.buildUrl(java.lang.String) returns the mobile device identifier.
    URL url = buildUrl(Constans.SEARCH_BASE_URL);
    if (url == null) {
        Logit.d(TAG, "buildUrl fail!");
        return false;
    }

    // url.openConnection() uses the mobile device identifier url as a password, security token, or cryptographic key which suggests that url is used in an authentication scheme.
    // Do not use the mobile device identifier url to identify an application installation or as a shared secret. Instead, use java.util.UUID.randomUUID() or other cryptographically strong random bits.
    conn = (HttpsURLConnection) url.openConnection();
    conn.setReadTimeout(15000 /* milliseconds */);
    conn.setConnectTimeout(15000 /* milliseconds */);
    conn.setRequestMethod("GET");
    conn.setDoInput(true);
    Logit.d(TAG, "doInBackground   -- 04");

    // ....
}
```

6.4.8.2 合规代码示例

```
public static String encodeURL(Context ctx, String baseUrl) {
     try {
        if (!isEmpty(baseUrl)) {
            //
            SecurityCipher c = new SecurityCipher(ctx);
            // 对 url 加密，此处使用的是 JVQ 编码
            String result = c.encodeUrl(baseUrl);
            Logit.d(TAG, "encodeURL: result = " + result);
            return result;
        }
    } catch (JVQException e) {
          Logit.w(TAG, "encodeURL: JVQException: message", e);
    }
```

```
    return baseUrl;
}
```

6.4.8.3 编程建议

优先使用 https 协议进行网络传输，传输过程中相关的敏感数据（不局限于 IMEI）不能包含在 URL 中。建议对敏感信息进行加密，且采用 POST 方式，使用 https 传输。

6.4.9 ADRD-08. 应避免客户端应用程序对服务器发起集中请求

DDoS 攻击通过大量合法的请求占用大量网络资源，以达到瘫痪网络的目的。 主要的攻击方式可分为以下几种：

- 通过使网络过载来干扰甚至阻断正常的网络通讯
- 通过向服务器提交大量请求，使服务器超负荷
- 阻断某一用户访问服务器
- 阻断某服务与特定系统或个人的通讯

Application level floods 主要是针对应用软件层的，也就是高于 OSI 的。它是以大量消耗系统资源为目的，通过向网络服务程序提出无节制的资源申请来迫害正常的网络服务。

为避免服务器的资源被无节制地消耗，客户端在设定后台请求服务器任务时，应考虑请求时间的随机性，避免服务器出现集中访问，导致服务器宕机或者无法正常响应请求。

移动终端后台联网的触发条件应具有随机性，如监听亮灭屏(ACTION_SCREEN_ON/OFF)、充电插拔(ACTION_POWER_CONNECTED/DISCONNECTED)、网络连接改变广播(CONNECTIVITY_ACTION)，接收新短信(SMS_RECEIVED)等。不能触发联网任务，如监听零点广播(ACTION_DATE_CHANGED)。

对于定时任务触发的后台联网，应考虑间隔一定范围内的随机时间。

6.4.9.1 不合规代码示例

6.4.9.2 不合规代码示例

```
1. JobScheduler js = (JobScheduler) context.getSystemService(Context.JOB_SCHEDULER_SERVICE);
2. int delayedTime = new Random().nextInt(60 * 1000);
3. Calendar begin = Calendar.getInstance();
4. begin.set(Calendar.DAY_OF_YEAR, c.get(Calendar.DAY_OF_YEAR) + 1);
5. begin.set(Calendar.HOUR_OF_DAY, new Random().nextInt(8) + 6);// 最早启动时间[06:00,14:00)
6. begin.set(Calendar.MINUTE, new Random().nextInt(60));
7. begin.set(Calendar.SECOND, new Random().nextInt(60));
8. Calendar end = Calendar.getInstance();
9. end.set(Calendar.DAY_OF_YEAR, c.get(Calendar.DAY_OF_YEAR) + 1);
10. end.set(Calendar.HOUR_OF_DAY, new Random().nextInt(8) + 14); // 最晚启动时间[14:00,22:00)
11. end.set(Calendar.MINUTE, new Random().nextInt(60));
12. end.set(Calendar.SECOND, new Random().nextInt(60));
13. JobInfo job = new JobInfo.Builder(JOB_ID, new ComponentName(context, MyService.class))
14.     .setMinimumLatency(begin.getTimeInMillis() - System.currentTimeMillis())
```

```
    .setOverrideDeadline(end.getTimeInMillis() - System.currentTimeMillis())
    .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
    .build();
js.schedule(job);
```

6.5 WebView 安全规则

6.5.1 JAVA-01. 禁止 WebView 通过 file:schema 访问本地敏感数据

如果加载了恶意的 file:schema url 来源，通过 js（当 setJavaScriptEnabled(true)）能够攻击到目标应用：

- WebView XSS 跨域访问

WebSettings#setAllowFileAccessFromFileURLs 被允许，本地任意文件能够被恶意 js 通过 XMLHTTP 访问 file:schema 获取到。

- WebView 加载本地 JS 文件

WebSettings#setAllowUniversalAccessFromFileURLs 被允许，任意文件（包括本地和 http/https）能够被恶意 js 获取到。

- 防护方法：

禁用 file:schema(webView.getSettings().setAllowFileAccess(false);)
对于需要使用 file:schema 的，禁止 file 协议调用
js(webView.getSettings().setJavaScriptEnabled(false);)

WebView 类显示网页作为 Activity 布局的一部分。可以使用 WebSettings 对象来定制 WebView 对象的行为，WebSettings 对象可以从 WebView.getSettings()获取。WebView 的主要安全问题是关于 setJavaScriptEnabled()，setPluginState()和 setAllowFileAccess()方法。

- setJavaScriptEnabled()

setJavaScriptEnabled（）告诉 WebView 启用 JavaScript 执行。 其默认值为 false。如果启用 JavaScript 要设定为 True:

```
1. webview.getWebSettings().setJavaScriptEnabled(True)
```

- setPluginState()

setPluginState()方法告诉 WebView 启用，禁用或按需启用插件。

6.5.1.1 不合规代码示例

```
1. copyFile();  // 自定义函数，释放 filehehe.html 到 sd 卡上
2. String url = "file:///mnt/sdcard/filehehe.html";
3. Intent contIntent = new Intent();
4. contIntent.setAction("android.intent.action.VIEW");
5. contIntent.setData(Uri.parse(url));
6. Intent intent = new Intent();
7. intent.setClassName("com.qihoo.browser","com.qihoo.browser.BrowserActivity");

8. intent.setAction("android.intent.action.VIEW");
9. intent.setData(Uri.parse(url));
10. this.startActivity(intent);
```

以下是攻击的关键代码。

```
1. function getDatabase() {
2.     var request = false;
3.     if(window.XMLHttpRequest) {
4.      request = new XMLHttpRequest();
```

```
        if(request.overrideMimeType) {
            request.overrideMimeType('text/xml');
        }
    }
    xmlhttp = request;
    var prefix = "file:////data/data/com.qihoo.browser/databases";
    var postfix = "/webviewCookiesChromium.db"; // 取保存 cookie 的 db
    var path = prefix.concat(postfix);
    // 获取本地文件代码
    xmlhttp.open("GET", path, false);
    xmlhttp.send(null);
    var ret = xmlhttp.responseText;
    return ret;
}
```

6.5.1.2 合规代码示例

对 file 域的路径增加安全限制。

```
copyFile();  // 自定义函数，释放 filehehe.html 到 sd 卡上
String url = "file:///mnt/sdcard/filehehe.html";
if (!url.toLowerCase().startsWith("xxxx/xxxx"))
{
    return;
}
Intent contIntent = new Intent();
contIntent.setAction("android.intent.action.VIEW");
contIntent.setData(Uri.parse(url));
Intent intent = new Intent();
intent.setClassName("com.qihoo.browser","com.qihoo.browser.BrowserActivity");

intent.setAction("android.intent.action.VIEW");
intent.setData(Uri.parse(url));
this.startActivity(intent);
```

6.5.2 ADRD-02. Webview 只允许加载存储在 apk 内的 asset 或者 res 目录下的文件

Webview 只加载存储在 apk 的 asset 或者 res 目录下面的文件，可以是 html 文件。可以使用 JavaScript。但不要访问除 asset 和 res 以外的 file 资源。

```
public class WebViewAssetsActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        WebView webView = (WebView) findViewById(R.id.webView);
        WebSettings webSettings = webView.getSettings();
```

```
        // *** POINT 1 *** 不能访问除 asset 和 res 以外的 file 资源
        webSettings.setAllowFileAccess(false);
        // *** POINT 2 *** 可以使用 JavaScript
        webSettings.setJavaScriptEnabled(true);
        // 加载内容存储在 asset 目录下
        webView.loadUrl("file:///android_asset/sample/index.html");
    }
}
```

Webview 容易出现的漏洞有包括但不限于以下几类：

- 任意代码执行漏洞

（1）addJavascriptInterface 接口引起远程代码执行漏洞。

JS 调用 Android 的其中一个方式是通过 addJavascriptInterface 接口进行对象映射。当 JS 拿到 Android 这个对象后，就可以调用这个 Android 对象中所有的方法，包括系统类（java.lang.Runtime 类），从而进行任意代码执行。如可以执行命令获取本地设备的 SD 卡中的文件等信息从而造成信息泄露。

```
webView.addJavascriptInterface(new JSObject(), "myObj");
// 参数 1：Android 的本地对象
// 参数 2：JS 的对象
// 通过对象映射将 Android 中的本地对象和 JS 中的对象进行关联，从而实现 JS 调用 Android 的对象和方法
```

以下是攻击的 Js 核心代码：

```
function execute(cmdArgs)
{
    // 步骤 1：遍历 window 对象
    // 目的是为了找到包含 getClass （）的对象
    // 因为 Android 映射的 JS 对象也在 window 中，所以肯定会遍历到
    for (var obj in window) {
        if ("getClass" in window[obj]) {

            // 步骤 2：利用反射调用 forName（）得到 Runtime 类对象
            alert(obj);
            return  window[obj].getClass().forName("java.lang.Runtime")

            // 步骤 3：以后，就可以调用静态方法来执行一些命令，比如访问文件的命令
            getMethod("getRuntime",null).invoke(null,null).exec(cmdArgs);

            // 从执行命令后返回的输入流中得到字符串，有很严重暴露隐私的危险。
            // 如执行完访问文件的命令之后，就可以得到文件名的信息了。
        }
    }
}
```

Android 在 Android 4.2 版本中规定对被调用的函数以 @JavascriptInterface 进行注解从而避免漏洞攻击。在 Android 4.2 版本之前采用拦截 prompt（）进行漏洞修复。

（2）searchBoxJavaBridge_接口引起远程代码执行漏洞

在 Android 3.0 以下,Android 系统会默认通过 searchBoxJavaBridge_的 Js 接口给 WebView 添加一个 JS 映射对象：searchBoxJavaBridge_对象。该接口可能被利用，实现远程任意代码。解决方案是删除 searchBoxJavaBridge_接口。

```
// 通过调用该方法删除接口
removeJavascriptInterface（）；
```

- 密码明文存储漏洞

WebView 默认开启密码保存功能， mWebView.setSavePassword(true)，开启后，在用户输入密码时，会弹出提示框：询问用户是否保存密码；如果选择”是”，密码会被明文保到 /data/data/com.package.name/databases/webview.db 中，这样就有被盗取密码的风险。解决方案是关闭密码保存提醒。

```
WebSettings.setSavePassword(false);
```

- 域控制不严格漏洞

使用 file 域加载的 js 代码能够使用进行同源策略跨域访问，从而导致隐私信息泄露。

（1）同源策略跨域访问：对私有目录文件进行访问

（2）针对 IM 类产品，泄露的是聊天信息、联系人等等

（3）针对浏览器类软件，泄露的是 cookie 信息泄露。

如果不允许使用 file 协议，则不会存在上述的威胁；但同时也限制了 WebView 的功能，使其不能加载本地的 html 文件。

```
webView.getSettings().setAllowFileAccess(true);
```

6.5.2.1 编程建议

对于不需要使用 file 协议的应用，建议禁用 file 协议

```
setAllowFileAccess(false);
setAllowFileAccessFromFileURLs(false);
setAllowUniversalAccessFromFileURLs(false);
```

对于需要使用 file 协议的应用，建议禁止 file 协议加载 JavaScript。

```
// 需要使用 file 协议
setAllowFileAccess(true);
setAllowFileAccessFromFileURLs(false);
setAllowUniversalAccessFromFileURLs(false);

// 禁止 file 协议加载 JavaScript
if (url.toLowerCase().startsWith("file://")) {
    setJavaScriptEnabled(false);
} else {
    setJavaScriptEnabled(true);
}
```

由于某些系统函数对于 URL Scheme 大小写不敏感，因此需要将限定的 URL Scheme 进行统一大写或统一小写后再进行比较，防止大小写混写造成绕过。

6.5.3 JAVA-03. Webview 只允许访问安全的 URL

安全 URL,是指具有安全保证的，企业内部的或者者企业与企业之间达成了安全共识之间的访问 URL。开发者应妥善处理 SSL 错误并限制 URL 请求为 HTTPS。

webView.getSettings().setJavaScriptEnabled(true);会加载不可信的 URL，可能造成跨域攻击和 http 攻击。

6.6 加解密安全规则

6.6.1 总体原则

- **禁止使用私有的（自己设计或实现）、非标准的加解密算法，使用业界标准的算法实现。**

  私有、非标准的算法包括但不限于：

  （1）未公开的、自行设计的密码算法

  （2）自行对标准密码算法进行改造的

  （3）自行定义的通过变形/字符移位/替换等方式执行的数据转换算法

  （4）用编码的方式（如 Base64 编码）实现数据加密目的的伪加密实现

  （5）用差错控制编码（如奇偶校验、CRC）实现完整性校验

- **优先国际标准密码算法（国密算法在发往国外的产品中限制使用）**

  对于已公开的中国国密算法（SM2/SM3/SM4），在发往国外的产品中可以支持，但须默认关闭，除非客户明确要求或者明确声明不做要求。发往国外的产品中应禁止使用未公开的中国国密算法 SSF33/ SM1/SM7/SM9）。

国密算法名称	所属分类	是否公开
SSF33	分组加密算法	暂未公开
SM1	分组加密算法	暂未公开
SM2	椭圆曲线公钥密码算法	已公开
SM3	哈希算法	已公开
SM4	分组加密算法	已公开
SM7	分组加密算法	暂未公开
SM9	基于身份标识的非对称密码算法	暂未公开

- **禁止使用已知的不安全密码算法**

  下表列出业界已知的不安全密码算法推荐使用的算法：

是否安全	算法类型	算法
不安全	对称算法	DES 在所有场景下都不安全 3DES 在密钥长度 256 以下，k1=k2=k3 时不安全 SKIPJACK 和 RC2 在所有场景下都不安全 RC4 和 BlowFish 当密钥长度 128 以下时，不安全
	非对称算法	RSA 在密钥长度 1024 以下时不安全
	哈希算法	MD2,MD4 在所有场景下都不安全 MD5 不应用于数字签名或密码加密 SHA1 不应用于数字签名
	消息认证码算法	HMAC-MD4 在所有场景下都不安全
安全	对称算法	3DES 密钥长度 128 及以上，且 k1,k2,k3 互不相等 AES 密钥长度 128 及以上
	非对称算法	RSA 密钥长度 1024 及以上
	哈希算法	SHA2 密钥长度 256 及以上
	消息认证码算法	HMAC-SHA2 密钥长度 256 及以上

- **优先使用操作系统提供的 API 实现加解密**

6.6.2 C-01. 禁止使用私有密码算法

不使用私有的密码算法是安全从业人员的基本准则。

私有密码算法是指自己设计或实现的加密算法和协议。私有密码算法未经过大量的理论和攻击验证，其安全性依赖于密码的保密程度和作者的水平。建议使用使用业界标准的算法和已有的公开实现。

6.6.2.1 不合规代码示例

```
1.  nt convert_msg(uint8_t *buffer, int buffer_size)
2.
3.      int i = 0;
4.      uint8_t data;
5.      if(buffer == NULL)
6.      {
7.          return FAILD;
8.      }
9.      for(i = 0; i < buffer_size; i ++)
10.      {
11.          data = *(buffer + i);
12.          *(buffer + i) = data << ((i * ) % 8) ; // 自己写的简单偏移算法。
13.      }
14.      return SUCCESS;
15. }
```

6.6.2.2 合规代码示例

```
1.  static int32 keymaster_verify_signature(km_key_blob_type *key_blob,
2.                                          uint32 signed_data,
3.                                          uint32 dlen,
4.                                          uint32 signature,
5.                                          uint32 slen,
6.                                          keymaster_rsa_sign_params_t sign_para
m)
7.  {
8.      // ……
9.      key.type = QSEE_RSA_KEY_PUBLIC;
10.     ret = qsee_rsa_verify_signature(&key, QSEE_RSA_NO_PAD, NULL, QSEE_HASH_ID
X_NULL, (unsigned char *)signed_data, dlen, (unsigned char *)signature, slen);
11.     // 此标准的 RSA 签名验证算法
12.     // ……
13. }
```

6.6.3 JAVA-02. 敏感数据禁止硬编码在代码中

应用程序的敏感信息，如配置信息，个人敏感信息等不能硬编码在代码中。

将密钥硬编码在 Java 代码、文件中，会引起很大风险，要防范密钥硬编码的风险。信息安全的基础在于密码学，而常用的密码学算法都是公开的，加密内容的保密依靠的是密钥的保密，密钥如果泄露，对于对称密码算法，根据用到的密钥算法和加密后的密文，很容易得到加密前的明文；对于非对称密码算法或者签名算法，根据密钥和要加密的明文，很容易获得计算出签名值，从而伪造签名。

密钥硬编码在代码中，而根据密钥的用途不同，这导致了不同的安全风险，有的导致加密数据被破解，数据不再保密，有的导致和服务器通信的加签被破解。

密钥硬编码的主要形式有：

（1）密钥直接明文存在 sharedprefs 文件中，这是最不安全的。

（2）密钥直接硬编码在 Java 代码中，这很不安全，dex 文件很容易被逆向成 java 代码。

（3）将密钥分成不同的几段，有的存储在文件中、有的存储在代码中，最后将他们拼接起来，可以将整个操作写的很复杂，这因为还是在 java 层，只要花点时间，也很容易被逆向。

（4）用 NDK 开发，将密钥放在 so 文件，加密解密操作都在 so 文件里，这从一定程度上提高了的安全性，挡住了一些逆向者，但是有经验的逆向者还是会使用 IDA 破解的。

（5）在 so 文件中不存储密钥，so 文件中对密钥进行加解密操作，将密钥加密后的密钥命名为其他普通文件，存放在 assets 目录下或者其他目录下，接着在 so 文件里面添加无关代码（花指令），虽然可以增加静态分析难度，但是可以使用动态调式的方法，追踪加密解密函数，也可以查找到密钥内容。

保证密钥的安全确是件难事，涉及到密钥分发，存储，失效回收，APP 防反编译和防调试，还有风险评估。可以说在设备上安全存储密钥这个基本无解，只能选择增大攻击者的逆向成本，让攻击者知难而退。而要是普通开发者的话，做妥善保护密钥这些事情这需要耗费很大的心血。开发者应通过评估 APP 应用程序的重要程度及密钥用途来选择相应的技术方案。

6.6.3.1 不合规代码示例

该案例是某应用被反编译后，发现的 DES 算法：

```
1.  public class ai
2.  {
3.      private static byte[] a={1, 2, 3, 4, 5, 6, 7, 8};
4.
5.      public static String a(String paramString1, String paramString2)
6.      {
7.          new IvParameterSpec(a);
8.          SecretKeySpec localSecretKeySpec = new SecretKeySpec(paramString2.get
Bytes(), "DES");
9.          Cipher localCipher = Cipher.getInstance("DES/ECB/PKCS5Pandding");
10.         localCipher.init(1, localSecretKeySpec);
11.         return b.a(localCipher.doFinal(paramString1.getBytes()));
12.     }
13.
14.     public static String b(String paramString1, String paramString2)
15.     {
16.         byte[] arrayOfByte = b.a(paramString1);
17.         new IvParameterSpec(a);
18.         SecretKeySpec localSecretKeySpec = new SecretKeySpec(paramString2.get
Bytes(), "DES");
19.         Cipher localCipher = Cipher.getBytes("DES/ECB/PKCS5Pandding");
```

```
        localCipher.init(1, localSecretKeySpec);
        return new String(localCipher.doFinal(arrayOfByte));
    }
}
```

发现 DES 算法的密钥，硬编码为“yrdAppKe”：

```
.method static constructor <clinit>()v
    .locals 1

    .prologue
    .line 36
    const-string v0, "yrdAppKe"

    sput-object v0, Lcom/yirendai/ui/lockPattern/SetLocusPassWordView;->Ljava
/Lang/String
```

6.6.3.2 合规代码示例

6.6.4 JAVA-03. 应使用加密库的安全加密方法

使用系统提供的 API 实现加密时，应使用加密库的安全加密方法，Android Java 安全提供程序 API 默认使用了不安全的 AES 加密方法（AES 加密的 ECB 块加密模式），使用时应改用 CBC 或 CFB 模式。

使用系统提供的 API 实现加密时，应考虑更安全的加密方法，如：

（1）不使用 ECB 模式进行加密。

（2）使用随机 IV 进行 CBC 加密。

（3）对 PBE（Password-based encryption，基于用户口令的加密方法）增加迭代次数（如多于 1000 次），并使用随机盐值，随机盐可以有效的防止暴力破解。

在使用加密算法时，应指明加密模式。在 Android 应用程序中使用加密算法，主要使用 java.crypto 中的 Cipher 类。首先通过指定加密类型来创建一个 Cipher 类对象的实例使用。可以通过两种格式可以指定：“algorithm/mode/padding”和“algorithm”。后者，加密模式和填充将隐式设置为 Android 可能访问的加密服务提供商的相应默认值。选择这些默认值以优先考虑方便性和兼容性，但在某些情况下不是特别安全的选择。为此，为确保正确的安全保护，必须使用前者的格式，加密模式和填充是明确指定的。

6.6.4.1 不合规代码示例

6.6.4.2 合规代码示例

Android 提供的 AES 加密算法 API 默认使用的是 ECB 模式，所以要显式指定加密算法为：CBC 或 CFB 模式，可带上 PKCS5Padding 填充。AES 密钥长度最少是 128 位，推荐使用 256 位。

```
// 生成 KEY
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
keyGenerator.init(256);
// 产生密钥
SecretKey secretKey = keyGenerator.generateKey();
```

```
// 获取密钥
byte[] keyBytes = secretKey.getEncoded();

// 还原密钥
SecretKey key = new SecretKeySpec(keyBytes, "AES");
// 加密
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, key);
byte[] encodeResult = cipher.doFinal(plainText.getBytes());

```

6.6.5 JAVA-04. 不能使用弱加密或已知的不安全算法

优先使用移动设备操作系统提供的 API 实现储存加密机制，建议使用的加密算法与密钥长度：

- 加密密钥有效长度为 128 位、192 位或 256 位的先进加密标准(AES)算法。
- 加密密钥有效长度为 112 位或 168 位的三重数据加密算法(3DES)算法，且 k1，k2，k3 不相等。
- 签名/完整性保护建议使用 SHA-256 或更高强度的算法。

```
byte[] input = message.getBytes();
MessageDigest sha = MessageDigest.getInstance("SHA-256");
sha.update(input);
byte[] output = sha.digest();
String result = Base64.encodeToString(output, Base64.DEFAULT);
```

Android SDK 使用的 API 和 JAVA 提供的基本相似，由 Java Cryptography Architecture (JCA，java 加密体系结构），Java Cryptography Extension (JCE，Java 加密扩展包)，Java Secure Sockets Extension(JSSE，Java 安全套接字扩展包)，Java Authentication and Authentication Service(JAAS，Java 鉴别与安全服务)组成。

6.6.5.1 不合规代码示例

不建议使用 MD2、MD4、MD5、SHA-1、RIPEMD 算法来加密用户密码等敏感信息。这一类算法已经有很多破解办法，例如 md5 算法，网上有很多查询的字典库，给出 md5 值，可以查到加密前的数据。

```
MessageDigest md = MessageDigest.getInstance("MD5");
byte[] md5Bytes = md.digest(str.getBytes());
String result = Base64.encodeToString(md5Bytes, Base64.DEFAULT);
```

6.6.5.2 合规代码示例

使用 RSA 进行数字签名的算法，密钥长度不要低于 1024 位，建议使用 2048 位的密钥长度。如：

```
//生成密钥
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
keyPairGenerator.initialize(2048);
KeyPair keyPair = keyPairGenerator.generateKeyPair();
RSAPublicKey rsaPublicKey = (RSAPublicKey)keyPair.getPublic();
RSAPrivateKey rsaPrivateKey = (RSAPrivateKey)keyPair.getPrivate();
```

```
7.
8.  //签名
9.  PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(rsaPrivateKey.getEncoded());
10. KeyFactory keyFactory = KeyFactory.getInstance("RSA");
11. PrivateKey privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);
12. Signature signature = Signature.getInstance("SHA256withRSA");
13. signature.initSign(privateKey);
14. signature.update(src.getBytes());
15. byte[] result = signature.sign();
```

使用 RSA 算法做加密，RSA 加密算法应使用 Cipher.getInstance(RSA/ECB/OAEPWithSHA256AndMGF1Padding)，否则会存在重放攻击的风险。如：

```
1.  KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
2.  keyPairGenerator.initialize(2048);
3.  KeyPair keyPair = keyPairGenerator.generateKeyPair();
4.  RSAPublicKey rsaPublicKey = (RSAPublicKey) keyPair.getPublic();
5.  RSAPrivateKey rsaPrivateKey = (RSAPrivateKey) keyPair.getPrivate();
6.
7.  // 公钥加密
8.  X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(rsaPublicKey.getEncoded());
9.  KeyFactory keyFactory= KeyFactory.getInstance("RSA");
10. PublicKey publicKey = keyFactory.generatePublic(x509EncodedKeySpec);
11. Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA256AndMGF1Padding");
12. cipher.init(Cipher.ENCRYPT_MODE, publicKey);
13. byte[] result = cipher.doFinal(src.getBytes());
14. // ...
15. // 私钥解密
16. PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(rsaPrivateKey.getEncoded());
17. KeyFactory keyFactory2 = KeyFactory.getInstance("RSA");
18. PrivateKey privateKey = keyFactory2.generatePrivate(pkcs8EncodedKeySpec);
19. Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA256AndMGF1Padding");
20. cipher5.init(Cipher.DECRYPT_MODE, privateKey5);
21. byte[] result2 = cipher.doFinal(result);
```

6.6.6 JAVA-06. PBE 应给不同用户使用不同的盐值

基于用户口令生成加密密钥时应用盐值。另外，如果您为不同的用户提供功能相同的设备，建议每个用户使用不同的盐值。原因是，如果您生成加密密钥仅使用简单的散列函数而不使用盐值，可以使用称为“彩虹表(rainbow table)”的技术轻松恢复密码。使用不同盐值，从相同密码生成的密钥将是不同的（不同的哈希值），从而防止通过彩虹表搜索散列值。

6.6.6.1 不合规代码示例

6.6.6.2 合规代码示例

```
1.  public final byte[] encrypt(final byte[] plain, final char[] password) {
2.      byte[] encrypted = null;
3.      try {
4.          // *** POINT *** Explicitly specify the encryption mode and the paddi
ng.
5.          // *** POINT *** Use strong encryption methods (specifically, technol
ogies that m
6.          eet the relevant criteria), including algorithms, block cipher modes,
 and padding modes.
7.          Cipher cipher = Cipher.getInstance(TRANSFORMATION);
8.          // *** POINT *** When generating keys from passwords, use Salt.
9.          SecretKey secretKey = generateKey(password, mSalt);
10.         // ...
11.     }
12.     // ...
13. }
```

6.6.7 JAVA-05. PBE 不应把密码存储在移动设备

PBE（Password Based Encryption，基于口令加密）是一种基于口令的加密算法，其特点是使用口令代替了密钥，而口令由用户自己掌管，采用随机数杂凑多重加密等方法保证数据的安全性。PBE 算法在加密过程中并不是直接使用口令来加密，而是加密的密钥由口令生成，这个功能由 PBE 算法中的 KDF 函数完成。KDF 函数的实现过程为：将用户输入的口令首先通过“盐”（salt）的扰乱产生准密钥，再将准密钥经过散列函数多次迭代后生成最终加密密钥，密钥生成后，PBE 算法再选用对称加密算法对数据进行加密，可以选择 DES、3DES、RC5 等对称加密算法。

基于用户口令生成加密密钥时，不要在设备中存储密码，建议存储向量和生成参数。基于密码的加密的优点在于它不需要管理加密密钥，如果将密码存储在设备上将消除这个优势。在设备上存储的密码有被其它应用窃听的风险，出于安全考虑，也不应在设备上存储用户口令及生成的密钥（即明文和密文都不存储），可以存储盐值（每个用户应使用不同的盐值）及该盐值与某些信息的某种对应关系，通过这些对应关系来进行用户身份或权限验证。了解更多请参阅 PBE 加密、解密过程。

涉及密钥保存的情形，系统设计时需要具备在用户口令可能泄露的情况下及时修改并实时或准实时生效的能力；此场景一般用于服务端和客户端的配合。

6.6.7.1 不合规代码示例

6.6.7.2 合规代码示例

6.6.8 C-07. 存储敏感数据应选择 RPMB 或 SFS 并在存储前加密

一些重要信息，比如密钥的存储，需要选择安全的存储位置。RPMB 和 SFS 是 TEE 中提供的两种安全的存储方式。RPMB 是 EMMC 中的一块安全存储区域，这块区域的数据读写需要经过 HMAC 校验。

SFS 是 TEE 中提供的一套存储机制，TEE 会对存储在其中的数据做加密。RPMB 和 SFS 都只有 TEE 能访问，安全性较高。

保存敏感的数据文件，不管是文件系统中，应用程序 assets 目录，还是/data/目录下，都需要做文件的加密，以确保文件不被导出恶意使用。

6.6.8.1 不合规代码示例

```
1.  int vivo_write_data(char *file_name, char* data , uint32_t data_len)
2.  {
3.      FILE *fd;
4.      int error = 0;
5.
6.      // LOGD("start write feature file");
7.
8.      if(file_name == NULL || data == NULL || data_len == 0)
9.      {
10.         return -1;
11.     }
12.
13.     fd = fopen(file_name, "w+");
14.     if(fd == NULL){
15.         error = errno;
16.         LOGE("fopen failed: %s", strerror(error));
17.         return FACE_DETECT_ERROR_WRITE_FEATURE;
18.     }
19.
20.     fwrite(data,data_len, 1, fd);// 直接存储在 android 系统中
21.     fclose(fd);
22.     return 0;
23. }
```

6.6.8.2 合规代码示例

```
1.  int vivo_write_file(const char* file_name, uint8* data,uint32 data_len)
2.  {
3.      int32 ret =0;
4.      int file_handle = 0;
5.      char file_path[255] = {0};
6.      int write_size = 0;
7.
8.      sprintf(file_path,"%s%s", VIVO_FACE_CONFIG_PATH, file_name);
9.      file_handle = qsee_sfs_open(file_path, _O_RDWR | _O_CREAT|_O_TRUNC);
10.     if(file_handle == 0)
11.     {
12.         return -1;
13.     }
14.
15.     if(qsee_sfs_seek(file_handle,0,_SEEK_END) == -1)
```

```
    {
        qsee_sfs_close(file_handle);
        return -1;
    }

    write_size = qsee_sfs_write(file_handle, (char *)data, data_len);//存在SFS 中
    if(write_size != data_len)
    {
        qsee_sfs_close(file_handle);
        return -1;
    }
    qsee_sfs_close(file_handle);
    return ret;
}
```

6.6.9 C-08. 敏感数据加密所使用的密钥应一机一密

一般来说，一台机器一把密钥是最安全的，所以对于一些敏感数据的存储，最好使用一机一密的方式。可以根据手机唯一标识和其它参数通过 KDF 算法生成密钥，也可以使用 PBE 生成密钥。

6.6.9.1 不合规代码示例

```
int32  vivo_encrypt_aes(uint32 aes_buffer, uint32 aes_buffer_len, uint32 aes_plain_len, uint32 *aes_cihper_len)
{
    int status = E_SUCCESS,i = 0;
    qsee_cipher_ctx *aes_cipher_ctx = NULL;
    QSEE_CIPHER_ALGO_ET alg = QSEE_CIPHER_ALGO_AES_128;
    QSEE_CIPHER_MODE_ET mode = QSEE_CIPHER_MODE_CBC;
    QSEE_CIPHER_PAD_ET pad_mod = QSEE_CIPHER_PAD_PKCS7;
    uint8_t aes_cipher_key[16] =
    {
        0x01, 0x02, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03 // 固定的密钥进行加密
    };
    static uint8_t aes_cbc_iv[] =
    {
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
        0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04
    };
    uint8  *aes_cihper = NULL;
    // ...
    status = qsee_cipher_encrypt(aes_cipher_ctx,
                                 (uint8_t *)aes_buffer,
                                 aes_plain_len,
                                 (uint8_t *)aes_cihper,
```

```
                                    aes_cihper_len);
    // ...
}
```

6.6.9.2 合规代码示例

```
int vivo_security_storage_cipher_get_key(uint8* key,uint32* key_len)
{
    uint8 aes_inlay_key[VIVO_SECURITY_KEY_ENCRYPT_KEY_SIZE] = {bbbbbb};
    uint8 rsa_root_key[VIVO_SECURITY_RSA_KEY_SIZE_ALL] = {aaaaa};
    uint8 device_id[36] = {0};
    uint32 device_id_len = 36;
    int32 ret = VIVO_KM_ERROR_OK;
    if(key == NULL || key_len == NULL)
    {
        return -1;
    }

    if(get_uuid_string((char*)device_id,device_id_len,&device_id_len) != VIVO_KM_ERROR_OK)// 获取设备的一种唯一 ID
    {
        return -1;
    }


    if(qsee_kdf(device_id, device_id_len,
    (void*)(unsigned long)(aes_inlay_key), sizeof(aes_inlay_key),
    (void*)(unsigned long)(rsa_root_key), sizeof(rsa_root_key),
    key, VIVO_SECURITY_KEY_ENCRYPT_KEY_SIZE) != VIVO_KM_ERROR_OK)// 通过 kdf 算法生成密钥
    {
        return -1;
    }

    *key_len = VIVO_SECURITY_KEY_ENCRYPT_KEY_SIZE;
    return -1;
}
```

6.6.10 C-09. TEE 中使用的共享内存应进行合法性检查

TEE 的共享内存一般在 CA 端申请，在 TEE 中使用共享内存，需要对共享内存的合法性和长度进行检查，避免由于共享内存被篡改导致其它安全问题。

6.6.10.1 不合规代码示例

```
int handler_SetEnrollStatus( tci_t* pTci )
{
    int result =0;
    uint32_t status =  pTci->message.facepp_status.enroll_status;
    uint8_t* feature = (uint8_t*)pTci->message.facepp_status.feature;
    uint32_t feature_len =  pTci->message.facepp_status.feature_len;

    result = vivo _set_enroll_status(status, (char *)feature, feature_len); // 未经检测，直接使用共享内存
    return result;
}
```

6.6.10.2 合规代码示例

```
int handler_ReadImage( tci_t* pTci )
{
    int result = VIVO_FACE_OK;
    uint8_t* image_data = (uint8_t*)pTci->message.face_data.faceData;
    uint32_t* image_data_len =  &pTci->message.face_data.faceData_len;
    int i = 0;
    char *tci = (char *)pTci;

    if(image_data == NULL || image_data_len == NULL)
    {
        return VIVO_FACE_FAIL;
    }
    if( !tlApiIsNwdBufferValid(image_data, *image_data_len))// 检测共享内存合法性，该函数是 TA 厂商提供的
    {
        return TEE_ERR_NW_BUFFER;
    }
    result = vivo_read_image((char *)image_data, image_data_len);

    return result;
}
```

6.6.11 C-10. TEE 中定义的结构体必须严格内存对齐

TEE 中操作奇数地址会引发 TA 崩溃，造成 TEE 功能失效。这类问题特别难查，需要在编码定义结构体时对结构体里面的成员进行严格的内存对齐。避免出现崩溃。

6.6.11.1 不合规代码示例

```
#define MAX_FACE_NAME_LEN 256
typedef struct congfig {
```

```
    int name_len;
    char name[MAX_FACE_NAME_LEN + 1]; // name 长度为 257，内存不对齐
    char congfig_string[MAX_ENROLL_IMAGE_NUM][MAX_ENROLL_FEATURE_LEN]; // 操作此成员时出现崩溃
    int feature_version;
} face_congfig;
```

6.6.11.2 合规代码示例

```
#define MAX_FACE_NAME_LEN 256
typedef struct congfig {
    int name_len;
    char name[MAX_FACE_NAME_LEN + 4]; // name 长度为 260，内存对齐
    char congfig_string[MAX_ENROLL_IMAGE_NUM][MAX_ENROLL_FEATURE_LEN];
    int feature_version;
} face_congfig;
```

6.6.12 C-11. 不同 TA 禁止共用 RPMB 分区或者地址

高通平台 RPMB 分区的使用通过分区 ID 区分，谁先使用后面的就不能再使用。

MTK 平台的 RPMB 分区通过地址来区分，操作同一块 RPMB 地址，后面的会把前面的覆盖。不同 TA 使用同一块地址或者分区会造成功能缺失，或引发安全问题。

RPMB 在使用前需要确认分区或者地址是否已经被占用,不能与其他 TA 共用 RPMB 分区或者地址。

6.6.13 C-12. CA 和 TA 中禁止打印敏感数据的 log

在 TEE 开发过程中会加入很多 log 进行 debug，在开发完成后需要关闭相关 log 打印，防止敏感信息泄露，防止攻击者根据 log 信息推测代码逻辑，从而进行攻击。

请参阅 6.3.2 ADRD-02. 禁止在日志上输出敏感信息。

6.6.14 C-13. TEE 重要信息的存储需要有备份机制，需要存储内容哈希值

TEE 中存储的重要数据，需要有备份机制。防止由于 EMMC 位跳变或者黑客攻击等原因造成数据被更改却未察觉，从而使用不合法数据引发程序崩溃或安全隐患。存储数据是需存储一个备份，读取数据时如果发现主文件损坏，需要用备份数据恢复主文件。在使用数据前，先校验内容哈希值，确认内容完整性。

6.6.14.1 不合规代码示例

```
// 单独存储，未备份，未存储 hash
int vivo_write_file(const char* file_name, uint8* data,uint32 data_len)
{
    int32 ret =0;
    int file_handle = 0;
    char file_path[255] = {0};
    int write_size = 0;
```

```
8.
9.      sprintf(file_path,"%s%s", VIVO_FACE_CONFIG_PATH, file_name);
10.     file_handle = qsee_sfs_open(file_path, _O_RDWR | _O_CREAT|_O_TRUNC);
11.     if(file_handle == 0)
12.     {
13.         return VIVO_FACE_FAIL;
14.     }
15.
16.     if(qsee_sfs_seek(file_handle,0,_SEEK_END) == -1)
17.     {
18.         qsee_sfs_close(file_handle);
19.         return VIVO_FACE_FAIL;
20.     }
21.
22.     write_size = qsee_sfs_write(file_handle, (char *)data, data_len);
23.     if(write_size != data_len)
24.     {
25.         qsee_sfs_close(file_handle);
26.         return VIVO_FACE_FAIL;
27.     }
28.     qsee_sfs_close(file_handle);
29.     return ret;
30. }
```

6.6.14.2 合规代码示例

```
1.  int vivo_write_image_len(uint32_t image_len)
2.  {
3.      int result = VIVO_FACE_OK;
4.      uint8_t  data[VIVO_FACE_SHA256_HASH_SIZE + sizeof(uint32_t)] = {0};
5.      uint32_t data_hash_len = VIVO_FACE_SHA256_HASH_SIZE;
6.
7.      memcpy((char *)data, (char *)&image_len,sizeof(uint32_t));
8.      result = processSha256((uint8_t *)&image_len,sizeof(image_len),data + siz
eof(uint32_t),&data_hash_len);
9.      if(result != 0)
10.     {
11.         tlApiLogPrintf(TATAG"processSha256 error\n");
12.         return -1;
13.     }
14.
15.     result =  vivo_write_sfs(IMAGE_SFS_LEN_ID,(char *)data,sizeof(data));
16.     if(result != 0)
17.     {
18.         result =  vivo_write_sfs(IMAGE_SFS_LEN_ID,(char *)data,sizeof(data));
19.         if(result != 0)
20.         {
```

```
            tlApiLogPrintf(TATAG"vivo_face_write_image_len error\n");
            return -1;
        }
    }

    result =  vivo_write_sfs(IMAGE_SFS_LEN_ID + 1,(char *)data,sizeof(data));
    if(result != 0)
    {
        result =  vivo_write_sfs(IMAGE_SFS_LEN_ID + 1,(char *)data,sizeof(dat
a));
        if(result != 0)
        {
            tlApiLogPrintf(TATAG"vivo_face_write_image_len error\n");
            return -1;
        }
    }
    return 0;
}
```

6.6.15 C-14. TEE 中用到的密钥严禁出 TEE

TEE 在手机中被当做安全的执行环境，在 TEE 中做加解密，签名验签动作被认为是安全的，加解密签名验签的安全除了算法安全，运行环境安全，还有密钥安全，如果加解密签名验签所使用的密钥出现在 TEE 以外，就会影响整套机制的安全性，所以密钥严禁出 TEE。

6.6.16 C-15. TEE 中使用的算法和密钥长度要求

TEE 中使用的 AES 算法密钥必须使用 128 或者以上位数，最好是 256，CBC 模式，严禁使用 ECB 模式。

TEE 中严禁使用 MD5，需要使用 SHA256 计算哈希值。

TEE 中 RSA 密钥长度必须是 2048 位。

6.6.17 C-16. TEE 中进行关键信息比较时要使用安全的比较算法

TEE 中进行关键信息比较，如密码比较时，要使用安全的比较算法，不要用 C 标准库的比较函数，C 标准库函数以效率优先，不是以安全优先。

6.6.17.1 不合规代码示例

以下示例代码是以效率优先的字符串比较。

```
#include <stdio.h>
#include <stdlib.h>

int efficient_strcmp(const char* src, char* dst)
{
    char c1,c2;
```

```
    // 0 表示相等，非 0 表示不等
    unsigned char equal = 0;
    while(1)
    {
        c1 = *src++;
        c2 = *dst++;

        // 遇到第一个不相等的字符就得到结果并退出循环不再比较
        // 通过反复测试函数返回时间，将有机会知道是第几个字符不匹配
        if (c1 != c2)
        {
            equal = 1;
            break;
        }

        // 或者到字符串结尾退出循环
        if (c1 == '\0' || c2 == '\0')
            break;
    }

    return equal;
}
```

6.6.17.2 合规代码示例

以下示例代码是以安全优先的字符串比较。

```
#include <stdio.h>
#include <stdlib.h>

int secure_strcmp(const char* src, char* dst)
{
    char c1,c2;
    // 0 表示相等，非 0 表示不等
    unsigned char equal = 0;
    while(1)
    {
        c1 = *src++;
        c2 = *dst++;

        // 中间即使有不相等的字符，也不会退出循环；
        // 不会被人有机会发现是第几个字符不匹配的
        equal = equal | (c1 ^ c2);

        // 一直比较到字符串结尾才退出循环
        if (c1 == '\0' || c2 == '\0')
            break;
    }
```

```
22.
23.     return equal;
24. }
```

6.7 三方合作规则

6.7.1 禁止直接使用通过逆向手段获得的其它厂商的源代码和数据

通过逆向手段分析竞争对手的实现方案，是应对激烈市场竞争的必要手段，但为了追求进度，直接其它厂商的源代码和数据，会在产品里面留下被对方攻击的直接证据，同时也会带来很大的法律风险。即使从代码层面上，对函数、变量、文件重命名，也不能有效对抗这种风险。因此，可以参考竞争对手的实现方案，但必须自己重新定义数据结构，重新编写代码。

6.7.2 应给有版权的 SO 库增加双向验证机制

版权 SO 是版权方投入大量人力资源所开发的知识成果，如果能够轻易的剥离并在其它场合调用，则会给版权方带来巨大的损失。因此需要在 SO 内部设置一些框架依赖，确保不能直接在其它程序中调用，即不能独立剥离调用。

6.7.2.1 不合规代码示例

```
// So Code
void do_work(void) {
    // ...
}
// Caller Code
void work(void) {
    do_work();
}
```

6.7.2.2 合规代码示例

```
// So Code
void do_work(void){
    if(strcmp(sign,”aabbccdd”))
        return;
    // ...
}

// Caller Code
void work(void) {
    char* sign=”aabbccdd”;
    void work(void){
    do_work();
}
```

6.8 附录

6.8.1 个人信息

个人信息是指以电子或者其他方式记录的能够单独或者与其他信息结合识别特定自然人身份或者反映特定自然人活动情况的各种信息，如姓名、出生日期、身份证件号码、个人生物识别信息、住址、通信通讯联系方式、通信记录和内容、账号密码、财产信息、征信信息、行踪轨迹、住宿信息、健康生理信息、交易信息等。

判定某项信息是否属于个人信息，应考虑以下两条路径：一是识别，即从信息到个人，由信息本身的特殊性识别出特定自然人，个人信息应有助于识别出特定个人。二是关联，即从个人到信息，如已知特定自然人，则由该特定自然人在其活动中产生的信息（如个人位置信息、个人通话记录、个人浏览记录等）即为个人信息。符合上述两种情形之一的信息，均应判定为个人信息。

下表是个人信息举例：

个人基本资料	个人姓名、生日、性别、民族、国籍、家庭关系、住址、电话号码、电子邮箱等
个人身份信息	身份证、 军官证、护照、驾驶证、工作证、出入证、社保卡、居住证等
个人生物识别信息	个人基因、指纹、声纹、掌纹、耳廓、虹膜、面部特征等
网络身份标识信息	系统账号、 IP 地址、 邮箱地址及与前述有关的密码、口令、口令保护答案、用户个人数字证书等
个人健康生理信息	个人因生病医治等产的相关记录，如症、住院志、医嘱单、检验报告、手术及麻醉记录、护理记录、用药记录、药物食物过敏信息、生育信息、以往病史、诊治情况，家族病史、现病史、传染病史等，以及与个个身体健康状况产生的相关信息，及体重、身高、肺活量等
个人教育工作信息	个人职业、职位、工作单位、学历、学位、教育经历、工作经历、培训记录、成绩单等
个人财产信息	银行账号、(口令)、存款信息（包括资金数量、支付收款记录等）、房产信息、贷记录、征交信息、交易和消费流水等，以及虚拟货币、虚拟交易、游戏类兑换码等虚拟财产信息
个人通信息	通信记录和内容、短彩、彩信、电子邮件，以及描述个人通信的数据（常通称为元数据）等
联系人信息	通讯录、好友列表、群列表、电子邮件地址列表等
个人上网记录	指通过日志储存的用户操作记录，包括网站浏览记录、软件使用记录、点击记录等
个人常用设备信息	指包括硬件序列号、设备 MAC 地址、软件列表、唯一设备识别码（如 IMEI/androidID/IDFA/OPENUDID/GUID、SIM 卡 IMSI 信息等）等在内的描述个人常用设备基本情况的信息
个人位置信息	包括行踪轨迹、精准定位信息、住宿信息、经纬度等
其他信息	婚史、宗教信仰、性取向、未公开的违法犯罪记录等

6.8.2 个人敏感信息

个人敏感信息是指一旦泄露、非法提供或滥用可能危害人身和财产安全，极易导致个人名誉、身心健康受到损害或歧视性待遇等的个人信息。通常情况下，14 岁以下（含）儿童的个人信息和自然人的隐私信息属于个人敏感信息。可从以下角度判定是否属于个人敏感信息：

泄露：个人信息一旦泄露，将导致个人信息主体及收集、使用个人信息的组织和机构丧失对个人信息的控制能力，造成个人信息扩散范围和用途的不可控。某些个人信息在泄漏后，被以违背个人信息主体意愿的方式直接使用或与其他信息进行关联分析，可能对个人信息主体权益带来重大风险，应判定为个人敏感信息。例如，个人信息主体的身份证复印件被他人用于手机号卡实名登记、银行账户开户办卡等。

非法提供：某些个人信息仅因在个人信息主体授权同意范围外扩散，即可对个人信息主体权益带来重大风险，应判定为个人敏感信息。例如，性取向、存款信息、传染病史等。

滥用：某些个人信息在被超出授权合理界限时使用（如变更处理目的、扩大处理范围等），可能对个人信息主体权益带来重大风险，应判定为个人敏感信息。例如，在未取得个人信息主体授权时，将健康信息用于保险公司营销和确定个体保费高低。

下表给出个人敏感信息的举例：

个人财产信息	银行账号、鉴别信息（口令）、存款信息（包括资金数量、支付收记录等）、房产信息、贷贷记录、征信信息、交易和消费记录、流水记录等，以及虚拟货币、虚拟交易、游戏类兑换码等虚拟财产信息
个人健康生理信息	个人因生病医治等产生的相关记录，如病症、住院志、医嘱单、检验报告、手术及麻醉记录、护理记录、用药记录、药物食物过敏信息、生育信息、以往病史、诊治情况、家族病史、现病史、传染病史等，以及与个人身体健康状况产生的相关信息等
个人生物识别信息	个人基因、指纹、声纹、掌纹、耳廓、虹膜、面部识别特征等
个人身份信息	身份证、军官证、护照、驾驶证、工作证、社保卡、居住证等
网络身份标识信息	系统账号、邮箱地址及与前述有关的密码、口令、口令保护答案、用户个人数据证书等
其他信息	个人电话号码、性取向、婚史、宗教信仰、未公开的违法犯罪记录、通信记录和内容、行踪轨迹、网页浏览记录、住宿信息、精准定位信息等

6.8.3 常用密码算法标准

算法名称	标准编号	标准名称	发布机构	发布时间
DES	FIPS 46-3	Data Encryption Standard (DES)	NIST	1999
AES	FIPS 197	Advanced Encryption Standard	NIST	2001
3DES	SP 800-67 Rev. 1	Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher	NIST	2012
RC4			Ron Rivest	1994
EEA3		Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3.Document 2: ZUC Specification	3GPP	
SM4	GM/T 0002-2012	SM4分组密码算法	国家密码管理局	2012
RSA	PKCS#1V2.2	PKCS#1 V2.2 RSA Cryptography Standard	RSA Lab	2012
	RFC3447	Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1	IETF	2003
DSA	FIPS 186-3	Digital Signature Standard (DSS)	NIST	2009
	FIPS 186-3 Proposed Change	DRAFT Proposed Change Notice for Digital Signature Standard (DSS)	NIST	2012

ECDSA	ANS X9.62-2005	Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).	ANSI	2005
SM2	GM/T 0003-2012	SM2椭圆曲线公钥密码算法	国家密码管理局	2012
DH	RFC2631	Diffie-Hellman Key Agreement Method	IETF	1999
	PKCS#3 V1.4	Diffie-Hellman Key Agreement Standard	RSA Lab	1993
	IEEE Std. 1363-2000	Standard Specifications for Public Key Cryptography	IEEE	2000
SHA1	FIPS 180-4	Secure Hash Standard (SHS)	NIST	2012
	RFC3174	US Secure Hash Algorithm 1 (SHA1)	IETF	2001
SHA2	FIPS 180-4	Secure Hash Standard (SHS)	NIST	2012
MD5	RFC1321	The MD5 Message-Digest Algorithm	IETF	1992
SM3	GM/T 0004-2012	SM3密码杂凑算法	国家密码管理局	2012
HMAC	FIPS 198-1	The Keyed-Hash Message Authentication Code (HMAC)	NIST	2008
	RFC2104	HMAC: Keyed-Hashing for Message Authentication	IETF	1997
工作模式	SP 800-38 A	Recommendation for Block Cipher Modes of Operation - Methods and Techniques	NIST	2001
	SP 800-38 A - Addendum	Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode	NIST	2010
	SP 800-38 B	Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication	NIST	2005
	SP 800-38 C	Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality	NIST	2004
	SP 800-38 D	Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC	NIST	2007
	SP 800-38 E	Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices	NIST	2010
	SP 800-38 F	Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping	NIST	2012
PBE基于口令的加密	PKCS#5 V2.1	PKCS #5: Password-Based Cryptography Standard	RSA Lab	2012
	RFC2898	PKCS #5: Password-Based Cryptography Specification Version 2.0	IETF	2000
消息格式	PKCS#7 V1.5	PKCS #7: Cryptographic Message Syntax Standard	RSA Lab	1993
	RFC3852	Cryptographic Message Syntax (CMS)	IETF	2004

KDF密钥导出函数	PKCS#5 V2.1	PKCS #5: Password-Based Cryptography Standard	RSA Lab	2012
	RFC5869	HMAC-based Extract-and-Expand Key Derivation Function (HKDF)	IETF	2010
	SP 800-108	Recommendation for Key Derivation Using Pseudorandom Functions	NIST	2009